Avant-propos

Le sujet de l'Intelligence Artificielle fait partie du plan d'études de la section d'Informatique de l'EPFL depuis plus de 20 ans. Ce livre est le résultat d'une évolution constante des notes de cours et des exercices pendant cette période. Il contient donc à la fois des méthodes très anciennes, comme les algorithmes d'inférence et l'apprentissage de concepts, et des méthodes très récentes, comme la planification par contraintes et certaines techniques d'apprentissage. La sélection a été réalisée selon trois critères :

1) *Utilité pratique* : toutes les méthodes présentées dans ce livre trouvent leur application dans de nombreux logiciels en application pratique aujourd'hui, et le livre couvre la majorité de telles techniques.
2) *Couverture* : l'ensemble des techniques donne un aperçu équilibré des différents sujets traités en Intelligence Artificielle, à l'exclusion du traitement de la langue naturelle et de la robotique qui sont considérés comme des sujets propres.
3) *Simplicité* : les méthodes sont suffisamment simples pour être comprises et implémentées par un étudiant *bachelor* de dernière année. Aucune méthode ne demande des connaissances mathématiques ou informatiques particulièrement élevées. Par contre, elles ouvrent le chemin vers des méthodes plus sophistiquées.

Le contenu théorique est accompagné de nombreux exercices qui apprennent au lecteur à programmer pas à pas la majorité des algorithmes. C'est certainement l'une des fortes spécificités de l'ouvrage. Même si les programmes représentent des versions simples et en soi peu efficaces de l'implémentation des techniques du livre, ils sont utiles pour bien les comprendre et constituent une base pour des réalisations plus sophistiquées.

Remerciements

Nous voulons sincèrement remercier les contributions de nombreux collaborateurs du Laboratoire d'Intelligence Artificielle de l'EPFL. Jeff Primus, Eric Sauthier, Jamila Sam-Haroud, Esther Gelle et Paolo Viappiani ont contribué à la mise au point des exercices. Christian Frei a réalisé un gros effort de documentation de ceux-ci et de leurs solutions. Vincent Schickel, Bruno Alves, Clément Humbert et Thomas Léauté ont traduit le code de LISP en différentes versions de Python.

Pour cette deuxième édition entièrement revue, Marina Boia et Stéphane Martin ont énormément amélioré le contenu des exercices et certains chapitres. Nous remercions aussi les générations d'étudiants de l'EPFL qui ont contribué indirectement à ce livre par leurs commentaires et questions.

Table des matières

Avant-propos **v**

Table des matières **vii**

1 Introduction **1**
1.1 Connaissances : données non structurées 3
1.2 Modélisation du monde et des connaissances 3
1.3 Inférence : manipuler les connaissances 6
1.4 Historique de l'IA 7
1.5 Les domaines d'application de l'Intelligence Artificielle 11
1.5.1 Systèmes à base de connaissances 12
1.5.2 Raisonnement basé sur modèles : systèmes de planification 13
1.5.3 Systèmes d'apprentissage 14
1.6 Structure du livre 14

Systèmes à base de connaissances **17**

2 Connaissances et inférence **21**
2.1 Modèles et représentations 21
2.1.1 Modélisation et représentation 22
2.1.2 Les connaissances ont une interprétation unique 23
2.2 Représentation de la connaissance 25
2.3 Règles d'équivalence 26
2.4 Exemple de modélisation 27
2.5 Inférence 28
2.6 Exercice 30

3 Algorithmes d'inférence **31**
3.1 Forme normale 31
3.2 Inférence par résolution 32
3.3 Inférence propositionnelle par chaînage 35
3.4 Chaînage avant sans variables 35
3.5 Expression de connaissances générales grâce aux quantificateurs 38
3.5.1 Appliquer des connaissances quantifiées : unification . . 40
3.5.2 Filtrage (*pattern matching)* 40
3.5.3 Unification 40
3.5.4 Chaînage avant avec variables 41
3.6 Inférence par résolution avec variables 43
3.7 Exercices 47

4 Représentation structurée des connaissances 67
4.1 Cadres . . . 68
4.2 Réseaux à héritage structurés . . . 70
4.3 Logiques descriptives . . . 71
4.4 Exercices . . . 77

5 Raisonnement basé sur des règles et systèmes experts 79
5.1 Systèmes experts . . . 79
5.1.1 Inférence à chaînage arrière . . . 80
5.1.2 Critères de choix . . . 82
5.1.3 Formulation de questions . . . 83
5.1.4 Explication du raisonnement . . . 83
5.2 Problèmes spécifiques liés à l'inférence . . . 85
5.2.1 La négation . . . 85
5.2.2 Inférences non monotones . . . 86
5.2.3 Systèmes de maintenance de la cohérence . . . 87
5.3 Exercices . . . 91

6 Traitement de l'information incertaine 111
6.1 De la logique floue à une représentation de l'incertitude . . . 113
6.2 Les facteurs de certitude . . . 114
6.3 Réseaux bayésiens . . . 116
6.3.1 Chaînage d'inférences . . . 117
6.3.2 Importance de la causalité . . . 118
6.3.3 Inférence abductive . . . 121
6.3.4 Chemins de causalité multiples . . . 126
6.3.5 Applications . . . 129
6.4 Exercices . . . 130

Le raisonnement basé sur modèles 143

7 Résolution de problèmes par recherche 147
7.1 Arbres et graphes de recherche . . . 147
7.2 Algorithmes de recherche en profondeur-d'abord (DFS) et en largeur-d'abord (BFS) . . . 149
7.3 Recherche en profondeur limitée . . . 152
7.4 Détection explicite de cycles . . . 153
7.5 Recherche d'une solution optimale . . . 154
7.6 Exercices . . . 159

8 Satisfaction de contraintes 171
8.1 Définition des problèmes de satisfaction de contraintes (PSC) . 172
8.2 Formulation d'un réseau de contraintes binaires 174
8.3 Solution d'un PSC par recherche 177
8.3.1 Méthodes basées sur la recherche en profondeur-d'abord 177
8.3.2 Méthodes itératives 182
8.4 Solution par propagation 185
8.5 Consistance et complexité de la recherche 188
8.5.1 Consistance des nœuds 189
8.5.2 Consistance des arcs 189
8.6 Contraintes globales 190
8.7 Satisfiabilité 192
8.8 Optimisation sous contraintes 193
8.9 Exercices 195

9 Diagnostic 209
9.1 Trois manières d'implémenter un diagnostic 210
9.1.1 Diagnostic par abduction explicite 211
9.1.2 Transformation en déduction 212
9.1.3 Abduction par raisonnement incertain 213
9.2 Diagnostic basé sur la consistance 214
9.3 Proposition de mesures 217
9.4 Modèles de défaillances 219
9.5 Exercices 220

10 Génération de plans 227
10.1 Représentation d'un environnement changeant 227
10.2 Planification par chaînage arrière 230
10.3 Macro-opérateurs 233
10.4 La complexité du problème de la planification 234
10.5 Planification par inférence logique 235
10.6 Buts multiples 236
10.7 Extensions pour améliorer la flexibilité 241
10.8 Exercices 244

Apprentissage automatique 255

11 Induction de modèles paramétriques à partir d'exemples 259
11.1 Représentation 260
11.2 Biais 261
11.3 Apprentissage par recherche 262
11.3.1 Apprentissage par spécialisation 262
11.3.2 Apprentissage par généralisation 262
11.4 Frontières de décision 264

11.5 Régression 268
11.6 Classification par régression logistique 271
11.7 Classification probabiliste 272
11.8 Exercices 274

12 Apprentissage de classifications structurées **277**
12.1 Apprentissage de classifications disjonctives 277
12.2 Apprentissage d'arbres de décision : l'algorithme ID3 280
12.3 Bagging et boosting : combinaison de différentes techniques d'apprentissage 289
12.4 Exercices 293

13 Apprentissage non supervisé **305**
13.1 Apprentissage de sous-classes 305
13.2 Clustering hiérarchique 307
13.3 Clustering de partitionnement 312
13.4 Clustering probabiliste 314
13.5 Apprentissage semi-supervisé 317
13.6 Exercices 320

14 Apprentissage bio-inspiré **331**
14.1 Réseaux de neurones artificiels 331
14.2 Algorithmes génétiques 336

Solutions des exercices **341**

Bibliographie **399**

Index **405**

Chapitre 1

Introduction

Dès le début de l'informatique, les chercheurs se sont tout particulièrement intéressés à la reproduction de l'intelligence humaine sur ordinateur. C'était la motivation principale pour aller au-delà de la simple machine à calculer vers des automates capables de traiter l'information en général. Dans les années 1940, des chercheurs comme Turing, Von Neumann et Shannon ont tous apporté des contributions importantes dans ce sens.

L'Intelligence Artificielle (IA) a donc longtemps été synonyme d'informatique tout court. Elle se distinguait des mathématiques numériques qui voyaient dans les ordinateurs plutôt des machines à calculer. En 1956, l'IA est finalement devenue une discipline en soi lors de la Conférence de Dartmouth qui réunissait notamment McCarthy, Newell, Simon et Minsky, des personnages clés qui ont fortement influencé le développement de la discipline.

Depuis ces débuts, l'Intelligence Artificielle a subi un développement fulgurant et les techniques qui furent développés sont à la base de l'informatique telle que nous la connaissons aujourd'hui : la programmation orientée objet et la programmation fonctionnelle en sont issues, de même que les techniques d'analyse de données et d'apprentissage automatiques, qui sont largement répandues. Les progrès de l'IA se sont signalés par des succès comme le système `WATSON` qui a battu en 2011 les meilleurs joueurs humains dans le jeu télévisé *Jeopardy*, ou encore l'assistant personnel `SIRI` disponible sur certains téléphones mobiles ou les voitures autonomes.

Avant toute discussion sur les techniques de l'IA, il est nécessaire de prendre connaissance de ses principaux objectifs et raisons d'être. Une définition intuitive est facile à donner : il s'agit de l'étude des programmes informatiques qui simulent la pensée ou l'intelligence humaine. Mais comment définir exactement en quoi cela consiste ?

Si les premiers ordinateurs symboliques ont effectivement été appelés des *cerveaux électroniques*, l'évolution des techniques informatiques n'a pas tardé à démontrer le caractère trop péremptoire de cette vision des choses. Les définitions précises, comme celle qui prétend qu'un être intelligent est un être capable de *réagir* à son environnement, s'avèrent trop simplistes : un thermostat réagit aux modifications de son environnement, tout comme un être humain, mais rares sont les personnes qui lui attribueront de l'*intelligence*. Par contre, la *complexité* des comportements semble jouer un rôle important dans la définition de l'*intelligence*. Aujourd'hui, il apparaît clairement que la complexité de la pensée humaine est sans commune mesure avec tout ce qu'un ordinateur existant est à même de réaliser.

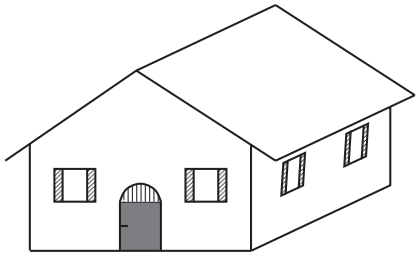


Fig. 1.1 *Dessin d'une maison en 3-dimensions ou texture sur une feuille de papier ?*

Une meilleure approche pour définir l'intelligence consisterait à étudier de plus près ce que nous, êtres humains, considérons comme intelligent. Examinons, par exemple, le dessin de la figure 1.1.

Imaginons ce qu'un robot équipé d'une caméra vidéo peut nous en dire. Une réponse *correcte* serait : « Je vois un morceau de papier sur lequel est dessiné un ensemble de lignes ». La réponse *intelligente*, « Je vois une maison », est paradoxalement fausse en soi, mais c'est très probablement celle qu'on aurait espérée d'un ordinateur intelligent. Ce dernier point dégage le critère selon lequel nous décidons que la réponse est intelligente ou ne l'est pas. Le fait de voir une maison sur la feuille de papier et non pas simplement un ensemble de lignes, est basé sur un grand nombre de *conventions* et de *connaissances* : les humains utilisent des dessins pour communiquer de l'information sur des objets tridimensionnels, l'élaboration de ces dessins obéit à des règles et la forme du dessin correspond à celle d'une maison. La plupart des programmes d'IA utilisent une quantité significative de connaissances humaines et sont généralement connus sous l'appellation de *systèmes basés sur la connaissance*. L'exemple que nous venons de voir nous permet de formuler une nouvelle définition de l'IA : un programme intelligent est un programme qui résout des problèmes en utilisant les règles et conventions propres aux humains (ou du moins qui s'en inspirent).

Si l'élaboration de programmes intelligents est déjà un objectif intéressant en soi, le succès de l'IA comme discipline d'ingénierie provient largement du fait qu'elle offre des solutions à des problèmes d'intérêt pratique. D'un point de vue technologique, l'IA fournit des moyens plus souples et plus efficaces pour produire du logiciel informatique. Par exemple, les techniques d'IA ont permis la mise au point de programmes qui peuvent conduire des véhicules autonomes, réagir à des pannes de circuits électriques, reconfigurer des réseaux de communication, ou traiter des rapports de sinistre d'une assurance. Un comportement intelligent est important, car il permet d'éviter les obstacles rencontrés en pratique plus facilement que les techniques algorithmiques classiques.

Dans les paragraphes suivants, nous verrons ce qui distingue actuellement la résolution de problèmes par un être humain de la manière de procéder des ordinateurs. Nous verrons aussi quels sont les problèmes pour lesquels l'IA peut être d'un meilleur apport que les techniques classiques.

1.1 Connaissances : données non structurées

Science et ingénierie sont toutes deux dominées par des *principes*, c'est-à-dire par des règles générales dont l'application uniforme résout *tous* les problèmes d'une même classe. Cela est particulièrement apparent en mathématiques, où le but principal consiste à réduire des problèmes, en apparence complexes, à un ensemble minimal de principes. Les ordinateurs ont été développés par des mathématiciens, il n'est donc pas étonnant que ce soient des machines particulièrement bien adaptées à la mise en œuvre de principes. Les premiers ordinateurs n'étaient en fait que des calculateurs implémentant les principes les plus généraux de l'algèbre, et qui ne pouvaient résoudre que des problèmes se ramenant à un ensemble de calculs algébriques. Aujourd'hui, même si les ordinateurs peuvent traiter un ensemble plus vaste d'opérations, les notions algorithmiques de base sous-jacentes aux programmes procéduraux reflètent toujours le concept de machines de calcul : les programmes sont des *séquences* d'étapes opérant sur des *données*. Cet aspect a conduit au développement des superordinateurs dans lesquels certains principes (par exemple le calcul de vecteurs) sont câblés (physiquement implémentés) afin d'être exécutés de façon extrêmement rapide.

En revanche, le comportement humain est régi par très peu de principes. Un adage bien connu affirme dans cet esprit qu'il n'y a pas de règle sans exceptions. En fait, les principes s'accommodent mal d'exceptions et il est ainsi difficile d'utiliser des ordinateurs dans des domaines où peu de principes connus sont clairement applicables.

En termes informatiques, on peut faire une distinction analogue entre le traitement de données structurées et non structurées. L'informatique classique est particulièrement adaptée au traitement de données structurées, comme par exemple des chiffres, des codes et d'autres contenus de bases de données. Par contre, le comportement humain repose sur beaucoup de données non structurées : des règles, des normes sociales ou encore des informations incertaines. Ces données sont difficilement formalisables sous un format structuré. Elles sont en général incomplètes, ambiguës et parfois aussi inconsistantes. En IA, elles sont appelées des *connaissances*.

1.2 Modélisation du monde et des connaissances

Comme le montre la figure 1.2, le premier pas pour tout système intelligent est de modéliser le monde par une représentation logique qui identifie les objets, propriétés et relations importants pour une tâche donnée. Par exemple, si la tâche est de conduire un véhicule autonome, il faut reconnaître les piétons, les autres véhicules, les voies de circulation, etc. Cette reconnaissance se fait par un système de vision ou par d'autres capteurs, et fournit une première représentation du monde. Toute tâche intelligente – comme le raisonnement, la résolution de problèmes ou l'apprentissage – repose sur une telle représentation, car l'ordinateur n'a pas d'autre connexion avec le monde.

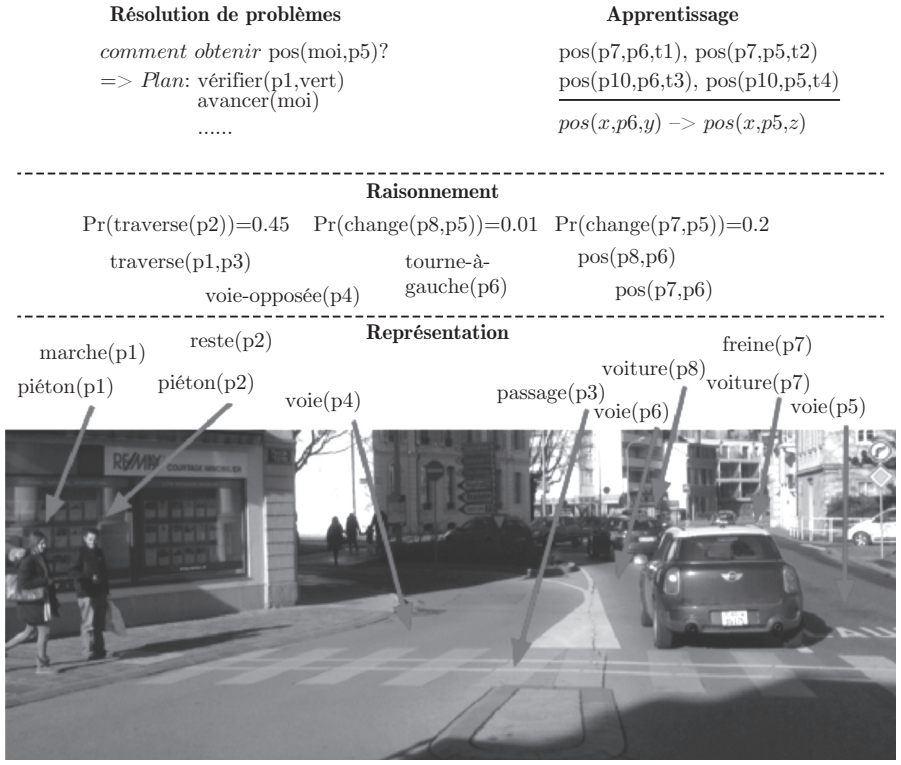


Fig. 1.2 *Modélisation logique du monde pour un véhicule autonome et exemples de raisonnement.*

Pour exprimer ces données non structurées, la société se sert des langues naturelles, telles que le français, l'anglais ou le chinois. Depuis le temps des Grecs, on a cherché à formaliser la signification des expressions du langage naturel. Le formalisme qui a été développé pour ce faire est la *logique des prédicats*. Elle repose sur des prédicats qui peuvent être vrais ou faux. Dans ce livre, nous exprimerons toujours nos modèles du monde dans le formalisme de la logique des prédicats. Même si cette représentation n'est pas obligatoirement présente de façon explicite dans tout logiciel IA, elle est utile au moins comme notation de base.

En général, il y a une correspondance entre prédicats et mots d'une langue. Sur la base des prédicats, on construit des expressions utilisant des connecteurs tels que *et* et *ou*. Ces expressions correspondent à des phrases. La signification d'une phrase devient donc : si la phrase est vraie, l'expression correspondante en calcul des prédicats l'est aussi.

L'avantage de ce formalisme est que l'on peut maintenant définir des règles qui permettent de juger si deux phrases sont consistantes entre elles, ou si une phrase est la conséquence d'une autre. Par exemple, les trois phrases :

1) « Si l'enfant de Mme Dupont est une fille, ce n'est pas un garçon »

2) « L'enfant de Mme Dupont est une fille »

3) « L'enfant de Mme Dupont est un garçon »

sont clairement inconsistantes, mais on ne peut pas s'imaginer un algorithme qui permettrait de détecter une telle inconsistance avec une certaine généralité. En formalisant :

- F = « L'enfant de Mme Dupont est une fille »
- G = « L'enfant de Mme Dupont est un garçon »

on obtient :

1) F $\Rightarrow \neg$ G

2) F

3) G

On peut alors appliquer des algorithmes généraux pour détecter l'inconsistance, ou encore faire l'inférence que G doit être faux sur la base des deux premières expressions.

Dans cet exemple, nous avons inventé des prédicats qui eux-mêmes correspondent déjà à beaucoup plus que des mots de la langue. Pour obtenir une traduction plus générale, le calcul des prédicats admet également des *variables* et des *quantificateurs*. Dans ce cas précis, nous pouvons utiliser le prédicat $F(x)$ qui est vrai si l'argument x est une fille. Nous pouvons alors introduire une constante E qui représente « l'enfant de Mme Dupont », et exprimer « l'enfant de Mme Dupont est une fille » par $F(E)$. Mieux encore, nous pouvons introduire un prédicat $E(x, y)$ qui dit « x est enfant de y », et utiliser la quantification existentielle $\exists x$ qui dit « il existe un x » pour écrire : $(\exists x)E(x, Dupont)$ *et* $F(x)$. Cela nous permet d'avoir un formalisme plus général, que nous pouvons appliquer non seulement à l'enfant de Mme Dupont, mais aussi à d'autres enfants.

On peut ensuite songer à exprimer également une autre information, qui est donnée par chacune des phrases, celle qu'il n'existe qu'un seul enfant de Mme Dupont. Il n'existe donc pas une seule et unique façon de traduire des phrases en calcul des prédicats, surtout quand les phrases sont ambiguës. En fait, souvent, on souhaite conserver une certaine ambiguïté, par exemple en diplomatie. La traduction automatique de la langue en calcul des prédicats reste donc un rêve qui ne sera probablement jamais réalité – on aura toujours besoin d'un humain pour résoudre les ambiguïtés.

C'est pour cette raison que l'IA ne permet pas de traiter directement les connaissances non structurées, mais suppose un processus qui aura traduit préalablement les connaissances en calcul des prédicats. Ces traductions existent de plus en plus, par exemple dans le « *text mining* » ou le web sémantique. L'utilisation du calcul des prédicats permet ensuite de construire des programmes capables de manipuler des connaissances non structurées telles que les expressions en langue naturel. C'est là l'importance essentielle des techniques d'IA dans l'informatique d'aujourd'hui.

1.3 Inférence : manipuler les connaissances

Il n'est pas possible de manipuler des connaissances par des opérations arithmétiques. Le mécanisme de base pour la manipulation de connaissances est le raisonnement utilisant des principes d'inférence logique.

Il existe trois mécanismes de raisonnement : la déduction, l'induction et l'abduction. Dans la *déduction*, un ensemble de propositions initiales (les *prémisses*) ainsi qu'un ensemble de règles sont utilisés pour inférer un ensemble de *conclusions*. Dans l'*abduction* par contre, ce sont les conclusions et les règles qui sont utilisées pour retrouver les prémisses desquelles découlent les conclusions. Dans l'*induction* enfin, prémisses et conclusions sont utilisées pour inférer l'ensemble des règles qui permettent de passer des premières aux secondes.

L'exemple suivant illustre les trois types de raisonnement. À partir des trois propositions suivantes :

`Oiseau(Titi)` : *Titi est un oiseau*
`(∀ x) oiseau(x) ⇒ chante(x)` : *Les oiseaux chantent*
`chante(Titi)` : *Titi chante*

on peut imaginer les inférences suivantes :

déduction :

$$\frac{\texttt{oiseau(Titi)} \quad (\forall \texttt{x})\ \texttt{oiseau(x)} \Rightarrow \texttt{chante(x)}}{\texttt{chante(Titi)}}$$

abduction :

$$\frac{\texttt{chante(Titi)} \quad (\forall \texttt{x})\ \texttt{oiseau(x)} \Rightarrow \texttt{chante(x)}}{\texttt{oiseau(Titi)}}$$

induction :

$$\frac{\texttt{oiseau(Titi), oiseau(Fred)} \quad \texttt{chante(Titi), chante(Fred)}}{(\forall\ \texttt{x})\ \texttt{oiseau(x)} \Rightarrow \texttt{chante(x)}}$$

Ces trois modes de raisonnement définissent les trois grands domaines applicatifs de l'IA. La déduction s'applique surtout à la modélisation directe de la pensée humaine, donc par exemple dans des programmes qui appliquent des règles. L'abduction a un grand nombre d'applications dans des problèmes de diagnostic, de planification et de conception. Les applications de l'induction se trouvent dans des systèmes d'apprentissage à partir d'exemples.

Dans la figure 1.2, nous montrons des exemples des trois types de raisonnement dans le cas de la conduite d'un véhicule autonome :

- Le raisonnement déductif est nécessaire pour établir les relations et propriétés des objets reconnus dans le monde. Il peut aussi tirer des conclusions incertaines avec une estimation de leur probabilité, par exemple pour décider si un piéton va traverser la route ou pas.
- Le raisonnement abductif est utilisé pour la planification : quelles actions faut-il accomplir pour se déplacer à un endroit voulu ? A quoi faut-il faire attention ?

- L'induction sert à l'apprentissage afin de prévoir le comportement des autres usagers de la route : comment se déplaceront-ils dans les différentes situations ?

Les résultats de l'abduction et de l'induction sont en général ambigus et ne sont pas tous nécessairement valides :

- L'abduction donne autant de résultats que de règles qui permettent l'inférence de sa prémisse.
- Il existe plusieurs règles qui peuvent résulter d'une induction sur le même ensemble de propositions. Par exemple, on aurait pu conclure par $(\forall$ `x) chante(x)` $\Rightarrow$ `oiseau(x)`.

Cependant, l'abduction et l'induction sont bien fondées sous l'hypothèse d'un *monde clos*. Cette hypothèse s'explique comme suit :

- Pour l'abduction : toutes les règles sont connues, et on ne pourra donc pas découvrir une autre explication.
- Pour l'induction : toutes les prémisses et conclusions sont connues, ce qui signifie qu'on ne pourra jamais découvrir un contre-exemple à la règle trouvée.

Comme on ne peut pas toujours assurer un monde clos, le seul mécanisme d'inférence dont on peut garantir le bien-fondé inconditionnel est la déduction. Il est à la base de la grande majorité des systèmes informatiques : les algorithmes classiques sont basés sur une déduction du résultat à partir des entrées. Cependant, sous l'hypothèse d'un monde clos, l'abduction et l'induction deviennent également fondées et peuvent être implémentées par un programme informatique. C'est ici que se trouve une grande partie de l'intérêt de l'IA par rapport à l'informatique classique.

1.4 Historique de l'IA

Le but des premiers ordinateurs était de réaliser de grands calculs, en particulier de trajectoires d'obus d'artillerie. Leur première utilisation en dehors des calculs numériques fut de casser des codes cryptographiques dans un projet mené par le mathématicien Alan Turing. Il fut l'un des premiers à développer une vision beaucoup plus large des ordinateurs et à formaliser cette vision dans des modèles théoriques comme la machine de Turing. Il considérait les ordinateurs comme de véritables cerveaux électroniques, capables de beaucoup plus que du simple calcul.

Pendant l'été 1956, un groupe de chercheurs s'est réunit au collège de Dartmouth (New Hampshire, USA) pour une conférence d'un mois. Il y avait là des chercheurs qui allaient devenir très influents, tels que l'organisateur John McCarthy et Herbert Simon, qui allait recevoir plus tard le prix Nobel. À cette époque, la puissance des ordinateurs progressait rapidement, et il paraissait évident qu'ils allaient égaler ou dépasser l'intelligence humaine au bout de peu de temps. La conférence a donc inventé le terme « Intelligence Artificielle ».

En 1955 déjà, Alan Newell et Herbert Simon avaient créé un programme intitulé le *Logic Theorist*, qui pouvait trouver des preuves – parfois très élégantes – de certains théorèmes mathématiques. Ils ont ensuite développé ce programme en une théorie générale de la résolution de problèmes, le *General Problem Solver* (GPS). En 1958, ils osèrent deux prédictions pour les dix années suivantes : un ordinateur serait le premier à trouver la démonstration d'un théorème important et un ordinateur deviendrait champion du monde d'échec. La première prédiction fut réalisée en 1974 avec le théorème des quatre couleurs, mais il fallut attendre beaucoup plus longtemps pour la deuxième. Il semblait alors qu'il n'y aurait aucune limite à ce qu'un ordinateur pourrait faire ; les seules limitations provenaient de leur capacité et de leur vitesse, et les deux faisaient des progrès rapides.

En même temps, on faisait des progrès importants dans la quête de deux autres objectifs : la compréhension du langage humain et la vision par ordinateur. En 1964, Daniel Bobrow a développé un programme, du nom de STUDENT, qui était capable de résoudre des exercices de mathématiques du niveau du lycée. Un autre programme, appelé ELIZA et développé par Joe Weizenbaum en 1966, simulait la conversation d'un psychologue avec son patient. Il était tellement convaincant que les gens oubliaient rapidement qu'ils étaient en train de communiquer avec un ordinateur. Ce fut le premier « chatterbot ». Le couronnement des ces recherches fut le programme SHRDLU, construit par Terry Winograd en 1970. Il permettait une conversation, raisonnait lui-même et planifiait des actions dans un monde simulé de blocs. Ainsi, il apparaissait comme un véritable collègue intelligent.

En vision, David Waltz développa des programmes qui pouvaient interpréter des images faites de traits (*line drawings*) comme des structures tridimensionnelles. Cette technique permettait aussi d'identifier dans des images des objets par leurs contours. Cette technique reste le principe de base de pratiquement toutes les techniques de vision par ordinateur utilisées aujourd'hui.

L'IA créa donc des attentes immenses, suivant un schéma qui plus tard sera identifié comme la courbe « hype » de Gartner (voir fig. 1.3) : certains succès impressionnants créèrent des attentes qui ne purent jamais être satisfaites. L'IA rencontrait ainsi des difficultés de deux côtés. Tout d'abord, la critique fondamentale la plus importante venait de Hubert Dreyfus, qui affirmait que l'intelligence devrait impliquer plus que du raisonnement, car les symboles utilisés en IA n'avaient pas de signification pour l'ordinateur qui les traite.

De plus, de sérieux problèmes apparaissaient dans le développement de l'algorithmique. Il devenait de plus en plus clair qu'il existait une certaine classe de problèmes, appelés NP-durs, pour lesquels on n'arrivait pas à trouver des algorithmes dont le temps de calcul n'explosait pas exponentiellement avec la taille du problème. En fait, presque tous les problèmes que traitaient l'IA tombaient dans cette catégorie. On fut donc amené à douter que les succès, qui avaient été obtenus sur des problèmes de petite taille, pourraient se généraliser à des problèmes plus grands. Cette prise de conscience a eu pour effet un arrêt presque total des recherches en IA.

Heureusement, il y eut aussi certains succès sur des problèmes d'intérêt pratique, en particulier ceux des *systèmes experts*. Le système DENDRAL, dont la

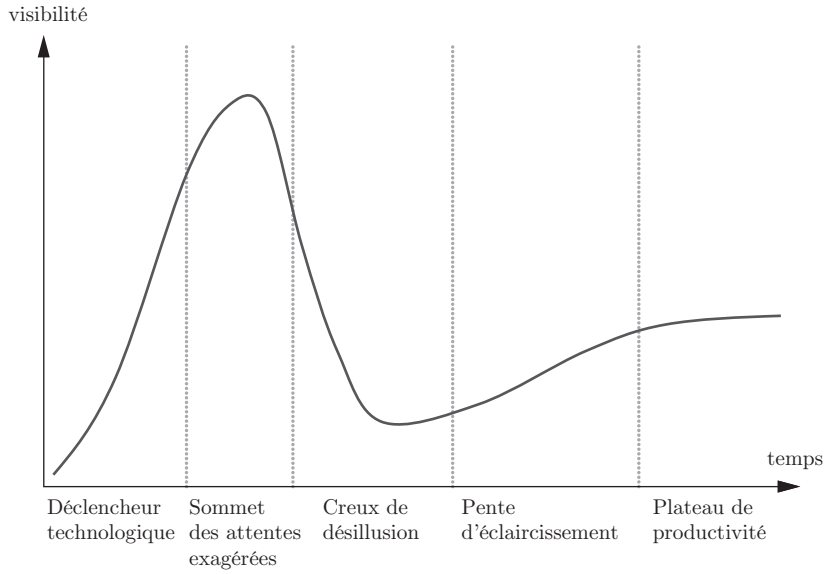


Fig. 1.3 *La courbe hype.*

mise au point avait commencé en 1965 déjà, pouvait identifier des molécules chimiques avec la même précision qu'un spécialiste humain et donc automatiser une tâche qui jusque-là nécessitait de très hautes qualifications. Le système MYCIN, dont le développement avait débuté en 1972, identifiait des maladies infectieuses et arrivait à proposer des traitements. Des analyses des performances du système ont montré qu'elles étaient supérieures à tout médecin humain. Cela peut être attribué au fait que le programme arrive à combiner les connaissances d'un grand nombre d'experts. Un autre grand succès industriel fut le système XCON, un système expert pour la configuration d'ordinateurs, développé par la Digital Equipment Corporation : il aurait permis des économies de $40 millions par an, une somme énorme pour l'époque. Les millieux industriels en prirent note, et bientôt les systèmes experts furent à la base d'une industrie naissante comportant de nombreuses entreprises comme Teknowledge, Inference, ou Intellicorp. En même temps on développa des ordinateurs spécialisés dans le traitement de données symboliques, ainsi que le langage de programmation `LISP`, produits par Symbolics et Texas Instruments. Presque chaque grande entreprises avait un département d'IA, et les espoirs mises dans la révolution des systèmes experts étaient énormes.

Cependant, on découvrit bientôt que les bases théoriques n'étaient pas encore assez mûres pour réaliser toutes ces attentes. Pour réussir dans de vrais problèmes de grande taille, il fallait résoudre des problèmes de base comme la logique non monotone et le raisonnement incertain. Or ces problèmes n'étaient pas très bien compris. Par conséquence, vers la fin des années 1980, le grand boom des systèmes experts toucha à sa fin, et l'IA entra dans la phase du creux de désillusion de la courbe de Gartner (voir fig. 1.3).

Cette phase a conduit à l'« hiver de l'IA », une période de déception qui a duré jusqu'en 1995. Certains sous-domaines sont devenus indépendants de l'IA. Le traitement du langage et celui de la parole ont adopté des approches statistiques, ne reposant plus sur des grammaires formelles développées par des linguistes, ce qui a conduit aux méthodes de recherche d'information utilisées dans les moteurs de recherche d'aujourd'hui. La vision par ordinateur est devenue un domaine en soi et a développé ses connexions avec le traitement des signaux et des images. L'accent a été mis plutôt sur la reconnaissance d'objets que sur la compréhension de l'environnement. La robotique s'est concentrée sur des robots autonomes avec peu de capacités, ne possédant souvent aucune connexion avec un raisonnement ou avec le langage. Même l'apprentissage automatique, clairement un sujet central de l'IA et fortement connecté avec le raisonnement symbolique, est souvent vu comme un domaine séparé. Là aussi, l'apparition des réseaux de neurones artificiels a eu tendance à démentir l'utilité des connaissances explicites, en réalisant l'apprentissage de façon implicite.

Cependant, le boom des années 1980 a produit de nombreux chercheurs en IA, qui, pendant les années 1990, ont réalisé des progrès importants sur les bases théoriques qui posaient tant de problèmes pour les systèmes experts. Pour le raisonnement, les techniques de satisfaction de contraintes ont permis le passage à des techniques d'inférence beaucoup plus puissantes et à un traitement propre des problèmes de logique non monotone. Les techniques des réseaux bayésiens ont fourni un cadre théorique solide pour le traitement des informations incertaines. Pour l'apprentissage, les *support vector machines* ont remplacé les techniques simples d'induction et les réseaux de neurones. Par conséquent, aujourd'hui, on retrouve des techniques telles que la programmation par règles (par exemple les *business rules*), l'inférence bayésienne, la programmation par contraintes, les résolveurs SAT et l'apprentissage automatique dans presque toutes les applications informatiques. L'IA est devenue une des bases de l'informatique et a finalement atteint le plateau de productivité de la courbe de Gartner (fig. 1.3).

Les années 1990 ont connu également un développement important des techniques de raisonnement probabilistes, notamment sur la base de modèles graphiques, tels que les réseaux de Bayes. La capacité de raisonner avec des informations incertaines a permis l'utilisation de connaissances qui ne sont pas exacts à 100%, et donc d'échapper à la contrainte de cohérence absolue qu'impose le cadre d'un raisonnement logique. Par exemple, dans le traitement du langage naturel, ceci a permis de raisonner à partir d'informations textuelles sans en avoir une compréhension parfaite. Dans des techniques de *machine reading*, la redondance des informations contenues dans des bases textuelles permet alors de tolérer un certain taux d'erreur dans la compréhension de textes individuels et même de corriger des erreurs par la suite en raisonnant sur la cohérence. Des modèles probabilistes se sont également imposés pour l'induction : l'apprentissage vise à apprendre des modèles probabilistes plutôt que strictement logiques. Ceci permet de construire les connaissances nécessaires à l'inférence par des techniques d'apprentissage.

Avec l'arrivée de l'internet dans la vie de tous les jours, l'Intelligence Artificielle a quitté le monde des laboratoires et de l'industrie spécialisée. La dis-

ponibilité de presque tout les textes en format numérisé via l'internet a fourni le cadre pour de nombreuses applications qui traitent automatiquement ces informations, un phénomène désormais connu sous le terme de *Big Data*. Citons quelques exemples :

- La compagnie Google fournit des services de traduction automatique grâce à des modèles obtenus par l'analyse de grandes quantités de textes bilingues trouvés sur le web.
- L'assistant SIRI, disponible sur les iphones de la compagnie Apple, parvient à désambiguïser les questions posées par son utilisateur, avant d'établir un plan de recherche des informations nécessaires à la réponse, de l'exécuter et de synthétiser une réponse.
- Le programme WATSON, lors d'une démonstration organisée par IBM, a battu les meilleurs joueurs humains du jeu télévisé *Jeopardy*, en cherchant en temps réel des informations spécialisées et en construisant un raisonnement qui permet de trouver les réponses demandées.

L'intelligence artificielle a aussi trouvé son utilisation dans de nombreux autres domaines, tels que des voitures auto-conductrices ou le négoce automatisé d'actions en bourse. Et, pour ne pas l'oublier : le 11 novembre 1997, avec presque 30 ans de retard sur la prédiction, le programme Deep Blue devenait champion du monde d'échec en battant le champion de l'époque, Gary Kasparov, dans un tournoi régulier.

Les techniques qui sont centrales à l'IA d'aujourd'hui, et qui font son succès dans les applications, sont basées sur l'inférence logique, les algorithmes de recherche et d'optimisation et diverses techniques statistiques utiles pour l'apprentissage automatique. Elles sont essentielles pour résoudre des problèmes tels que :

- le traitement d'informations non structurées, comme par exemple des textes ou le contenu de pages Web ;
- l'opérationnalisation de données, par exemple la génération de règles qui peuvent être appliquées automatiquement pour implémenter une certaine stratégie ;
- le calcul abductif, par exemple pour planifier ou ordonnancer des opérations afin d'atteindre certains buts ;
- le calcul inductif, par exemple l'apprentissage des préférences d'un utilisateur, la prévision des mouvements de la bourse, ou la détection d'anomalies dans une grande base de données.

Ce livre présente une introduction à ces techniques, développées par une communauté de milliers de chercheurs au cours des cinquante dernières années.

1.5 Les domaines d'application de l'Intelligence Artificielle

L'IA est un domaine très vaste qui a de nombreuses applications. Comme première classification, on peut en distinguer trois types :

- des programmes qui imitent des capacités *cognitives* et reproduisent un raisonnement humain, par exemple un diagnostic médical, la configuration d'un central téléphonique, la planification d'une mission spatiale ou la recherche de régularités dans une grande base de données ;
- des programmes qui imitent des capacités *sensorielles* et sont capables de reconnaître des formes ou des objets, ou bien de comprendre la parole en langage naturel ;
- des programmes qui imitent des capacités *sensomotrices* et sont capables de réagir de façon autonome à leur environnement, par exemple des robots ou agents autonomes.

Dans ce livre, nous traitons essentiellement d'applications du premier type, bien qu'une grande partie des techniques s'appliquent également aux autres applications. Les méthodes utilisées pour la vision et le traitement de la parole s'inspirent de plus en plus de modèles neuronaux du cerveau et de la statistique qui ne sont pas facilement applicables à des tâches de raisonnement et sont devenus des domaines en soi. Les agents autonomes posent des problèmes supplémentaires, comme la réactivité en temps réel et l'optimisation du comportement par rapport à leur environnement. Ils dépassent le cadre de ce livre et sont traités dans les domaines des agents autonomes et des systèmes multi-agents.

Pour illustrer les problèmes auxquels s'appliquent les techniques décrites dans ce livre, nous allons considérer quelques exemples.

1.5.1 Systèmes à base de connaissances

Une des premières applications de l'IA est l'automatisation de tâches complexes. Par exemple, des assurances pourraient souhaiter automatiser les décisions sur des dossiers de sinistres, des fabricants d'imprimantes souhaiteraient fournir des outils de diagnostic de pannes et on aimerait automatiser la configuration et la reconfiguration d'installations informatiques. Quand ces tâches dépassent une certaine complexité, elles impliquent une quantité importante de *connaissances* et il n'est plus rentable de les implémenter par des algorithmes. Cela est le cas surtout quand les connaissances changent, comme par exemple pour une assurance qui doit s'adapter à des règlements qui varient régulièrement.

Pour de telles applications, on modélise les connaissances sous-jacentes directement sous forme logique, en calcul des prédicats ou dans un langage spécialisé qui en est dérivé. Cela permet alors d'appliquer des algorithmes généraux, appelés moteurs d'inférence, pour obtenir les raisonnements qui en découlent. Un premier avantage est que le temps de développement se trouve fortement raccourci. De plus, il devient facile d'adapter le système à des changements de connaissances. On peut les changer directement au lieu de devoir développer à nouveau un algorithme qui en découle.

La complexité des connaissances peut varier entre des systèmes très simples et très complexes. Pour les applications dans les systèmes d'information, les

connaissances sont souvent des règles, lois ou *policies* qui doivent être appliquées automatiquement. Pour ce faire, on utilise des techniques relativement simples comme les *business rules*. À l'autre extrême, on trouve des systèmes qui modélisent le comportement d'un expert, par exemple pour aider à établir un diagnostic médical ou estimer des risques financiers. De tels systèmes, appelés aussi *systèmes experts*, peuvent faire appel à des techniques parfois très sophistiquées telles que la logique non monotone et le raisonnement incertain.

Une autre application des systèmes à base de connaissances se trouve dans le web sémantique. Il se base sur des langages standardisés qui permettent d'ajouter des connaissances logiques à des pages web. Il devient alors possible de faire des raisonnements complexes utilisant les connaissances disponibles sur le World Wide Web.

1.5.2 Raisonnement basé sur modèles : systèmes de planification

Un deuxième type d'application est la résolution de problèmes, tels que des casse-têtes, où des systèmes IA sont plus performants que les humains grâce à leur capacité de comparer un grand nombre de possibilités et de choisir la meilleure. Parmi les applications pratiques, le problème de la planification se pose de façon très nette pour la gestion des procédures impliquant des humains et des machines et dont le degré de complexité est tel qu'il est difficile d'en avoir une vision d'ensemble. Prenons pour exemple le cas d'une mission spatiale. Il existe des centaines de buts à réaliser pendant le laps de temps relativement court que dure la mission. De plus, dans le cas où les problèmes rencontrés n'ont jamais été étudiés auparavant, il faudra pouvoir très vite modifier les plans pour s'adapter aux imprévus. Les processus industriels de grande envergure, comme la construction d'avions, impliquent souvent des millions d'opérations différentes. Ils constituent un autre exemple typique de systèmes nécessitant une planification automatique par ordinateur.

Il est très intéressant d'utiliser des systèmes basés sur la connaissance dans le domaine de la planification : d'une part, parce que la tâche en elle-même est compliquée, d'autre part parce que par nature, elle ne peut être traitée algorithmiquement. En pratique, les systèmes de planification sont développés sur la base de règles heuristiques établies par des experts.

Les applications des systèmes de planification sont nombreuses. Par exemple, les missions de la navette spatiale américaine ont été organisées par un système de planification. Il en va de même pour la logistique des opérations militaires américaines. On utilise des systèmes d'IA pour planifier l'utilisation des installations d'usines chimiques ainsi que pour la production d'avions. Ils sont également utilisés pour planifier les mouvements d'avions au sol dans les grands aéroports.

Avec l'avènement du raisonnement probabiliste, la planification a également pu être appliquée à des problèmes peu structurés, tels que la planification d'acteurs de synthèse dans les jeux vidéo. Il est quasiment impossible d'imaginer de programmer leurs mouvements d'une façon réaliste sans faire appel aux techniques de planification de l'Intelligence Artificielle.

L'ordonnancement est une version simplifiée de la planification, qui se trouve à l'intersection de l'IA et de la recherche opérationnelle. On en trouve de nom-

breuses applications dans des domaines divers, tels que les mouvements d'avions sur un porte-avions, le placement de conteneurs dans un port ou encore la confection d'horaires de cours.

1.5.3 Systèmes d'apprentissage

Pour un grand nombre de personnes, c'est la capacité d'*apprendre* qui est le trait caractéristique de l'intelligence. L'apprentissage de nouvelles connaissances a donc été largement étudié en IA. Le processus qui est le mieux connu est celui de l'*induction*. Dans un système inductif, on présente à l'ordinateur des exemples positifs et négatifs caractérisant un *concept*. À partir de ces exemples, le programme construit une description compacte du concept.

On distingue deux types d'apprentissage : supervisé et non supervisé. Dans un apprentissage supervisé, les exemples présentés à l'ordinateur sont classées d'avance, et l'apprentissage construit un modèle qui reproduit cette classification. Comme ce type d'apprentissage a un but et une mesure de performance clairs, il a été longtemps au centre des préoccupations et on connaît aujourd'hui des techniques puissantes. La problématique inhérente à l'apprentissage est d'apprendre un modèle aussi fiable que possible avec aussi peu de données que possibles. On constate en fait souvent que la quantité des données est insuffisante pour permettre l'apprentissage avec la qualité voulue, surtout quand il faut fournir les classifications des exemples à l'entrée.

On s'est donc intéressé à exploiter la grande masse de données qu'on peut trouver sur le web, mais qui a normalement pas de classification associée, le *Big Data*. Apprendre néanmoins des modèles utilisables est le but des techniques d'apprentissage *non supervisée* ou *semi-supervisée*, qui ont connu un développement fulgurant ces dernières années. Ces algorithmes d'apprentissage permettent de tirer des leçons inhérentes aux données qui sont collectées, de les transformer en connaissances, et d'agir en conséquence. Le public ne voit pas toujours d'un oeil positif ces techniques, qui sont perçues comme une surveillance généralisée qui rappelle le *Big Brother*. Cependant, ils fournissent aussi de nombreux services, comme la recommandation de produits, l'optimisation de l'adaptation des ressources aux besoins de leurs utilisateurs ou la détection d'épidémies et la mise au point de nouvelles hypothèses scientifiques.

1.6 Structure du livre

Le contenu de ce livre est structuré en trois parties selon les trois modes d'inférences utilisés. La première partie traite des systèmes à base de connaissances (*knowledge-based systems*). Ces systèmes utilisent des moteurs d'inférence déductifs basés sur les principes de la logique et s'appliquent surtout à automatiser des tâches qui exigent des connaissances complexes.

La deuxième partie traite des systèmes utilisant l'abduction, appelés systèmes de *raisonnement à base de modèles* (*model-based reasoning*). Ces systèmes sont utilisés afin de trouver des solutions à des problèmes complexes tels que la planification de missions spatiales ou de processus de production.

La troisième partie traite des systèmes utilisant l'induction, appelés systèmes d'apprentissage automatique (*machine learning*). Ces systèmes ont pour but de trouver des nouvelles connaissances sur la base d'une grande quantité de données et s'utilisent par exemple dans la détection de fraudes.

Littérature

Le livre de référence de loin de plus cité sur l'Intelligence Artificielle est celui de Russel et Norvig [1], qui existe également en version française. Le livre de Poole et Mackworth [2] est plus compact et plus récent. Pour des synthèses avec une perspective historique par un des plus anciens chercheurs du domaine, consulter les livres de Nilsson [3, 4].

Les revues principales pour la publication des résultats de recherche en Intelligence Artificielle sont les revues *Artificial Intelligence* (Elsevier), qui est la plus prestigieuse, et le *Journal of Artificial Intelligence Research (JAIR)*, qui se trouve sur le web

`http://www.cs.washington.edu/research/jair/home.html`

La revue *Intelligent Systems* (IEEE Press) est une bonne revue focalisée surtout sur les applications.

Application : Python

L'IA demande la possibilité de traiter des données symboliques et d'autres structures dynamiques comme des listes. Pendant de nombreuses années, c'est le langage de programmation LISP qui était le seul à fournir efficacement ces possibilités.
Récemment, Python a été développé comme une alternative. Ce langage met à disposition des fonctionnalités similaires à LISP, mais dans une syntaxe plus proche des langages de programmation courants.
De nombreuses sociétés basent leur logiciels sur Python, comme par exemple Google (voir le site web `www.python.org`).

Première partie

Systèmes à base de connaissances

Les systèmes basés sur la connaissance sont des logiciels dont le comportement se base sur des connaissances qui sont généralement des informations non-structurées. Ils sont particulièrement utiles pour l'automatisation du raisonnement d'un expert humain, comme par exemple pour définir un ensemble de règles qui permettent de traiter des cas d'assurance ou qui aident au diagnostic de maladies.

Pour rendre des connaissances non structurées utilisables par un programme, on les traduit généralement dans un formalisme logique équivalent à la logique des prédicats. Nous supposons donc dans cette partie du livre que les connaissances sont exprimées sous cette forme, bien que certains outils puissent utiliser un autre format plus restreint. La représentation des connaissances est le contenu du premier chapitre de cette partie.

Sur la base de cette formulation logique, les systèmes à base de connaissances appliquent en général une inférence *déductive*. Cette partie du livre, et plus spécifiquement le deuxième chapitre, est donc consacrée à l'inférence déductive qui permet l'implémentation d'un raisonnement humain sur ordinateur.

Dans les cinq chapitres suivants, nous considérons trois sujets qui sont importants en relation avec des systèmes à base de connaissances : la représentation de connaissances par des règles et des représentations structurées, les techniques d'inférence et leur utilisation, ainsi que le traitement de connaissances incertaines.

Chapitre 2

Connaissances et inférence

Afin d'assurer la flexibilité qui caractérise les programmes d'IA, il est nécessaire de doter l'ordinateur de mécanismes lui permettant de représenter aussi bien les problèmes que leurs solutions. En effet, construire un système basé sur la connaissance requiert avant tout de pouvoir *représenter* des connaissances sur un ordinateur. Dans ce chapitre, nous nous intéresserons à ce qui distingue les *connaissances* des *données*, puis nous présenterons le formalisme du *calcul des prédicats* appliqué à la représentation des connaissances. Ensuite, nous montrerons comment utiliser ce formalisme pour construire des *moteurs d'inférence* automatiques.

2.1 Modèles et représentations

Si un programme est avant tout destiné à tirer des conclusions sur des situations du monde réel, il n'en reste pas moins incapable de dériver et de formuler ces conclusions à partir du monde réel lui-même. Il lui faudra d'abord disposer d'un *modèle* sur lequel il pourra travailler. Généralement, les programmes se basent sur des modèles mathématiques décrivant le monde réel comme un ensemble d'*entités* caractérisées par des *propriétés* et liées entre elles par des *relations*. L'exemple de la masse accrochée à un ressort, décrit par la figure 2.1, illustre ce type de modélisation.

```
Entités : m, s, g
propriétés : masse(m), ressort(s), fixé(g)
Relations : connecté(m,s), connecté(s,g)
```

Un tel modèle au niveau des objets et des relations reste cependant insuffisant pour effectuer des calculs sur le système. Si l'on désire par exemple simuler le

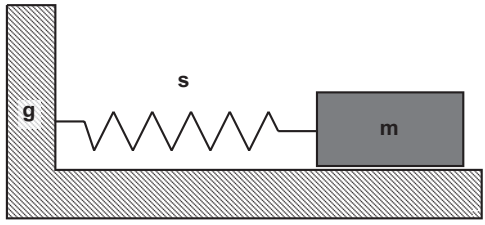


Fig. 2.1 *Une masse attachée à un ressort.*

mouvement du bloc, il est nécessaire de modéliser également sa position, sa vitesse, sa masse ainsi que la force agissant sur lui :

```
Entités : x, v, a, F, t
Relations : position(m, x), vitesse(m, v),
 accélération(m, a) force(s, F), temps(t),


 égal(v, dx/dt), égal(a, dv/dt), égal(F, -Dx), égal(F, ma)
```

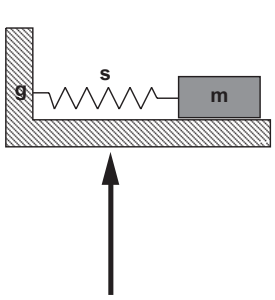
Notons toutefois que ce modèle présuppose que seul le mouvement de Newton de la masse sous l'influence de la force du ressort est intéressant. On pourrait aussi très bien modéliser les propriétés électriques du ressort et en particulier ses propriétés en tant que bobine magnétique. Lorsque l'on modélise un système physique, il est toujours nécessaire d'émettre des hypothèses limitant la taille du modèle. Pour élaborer un programme simulant ou analysant le système masse-ressort dans un langage de programmation algorithmique tel que C ou Java, seule la seconde partie du modèle, comportant les variables et les équations, est utile. Il n'est pas nécessaire – ni possible – de représenter dans le programme l'information concernant les objets et leurs relations. Nous verrons dans ce qui suit les différences de base existant entre la représentation des *données* dans les programmes algorithmique et la représentation des *connaissances* dans les systèmes basés sur la connaissance.

2.1.1 Modélisation et représentation

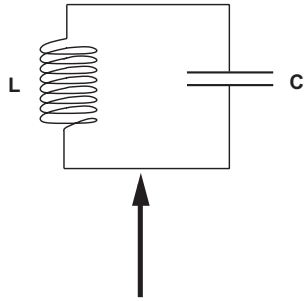
Un programme travaille sur des données pour dériver des prédictions. Données et prédictions sont toutes deux formulées dans les termes propres au modèle choisi. Pour réaliser cela, le programme manipule une *représentation* du modèle stockée dans la mémoire de l'ordinateur. Dans les langages de programmation les plus conventionnels, cette représentation consiste en un ensemble de *variables*, dont chacune représente une instance particulière choisie pour un élément du modèle. Dans le cas de la masse accrochée au ressort de la figure 2.1, un programme de simulation tiendra compte tout au plus de quatre variables, une pour chacun des **x**, **v**, **a** et **F**. L'ensemble constituera le modèle du système à simuler (fig. 2.2). En fait, il suffit de représenter **x** et **v** ainsi que le temps **t**.

Il est important de noter que si nous considérons que le programme de simulation modélise le système masse-ressort, ce n'est qu'une simple question d'*interprétation*. En effet, on pourrait utiliser exactement le même programme pour simuler, par exemple, un oscillateur électrique constitué d'un condensateur et d'une inductance : tout ce qu'il sera nécessaire de faire sera d'interpréter à nouveau la position de la masse comme une charge, sa vitesse comme le courant et le ressort comme une tension. Cela est illustré par la figure 2.2.

Ce dernier exemple démontre en fait l'importante distinction existant entre un *modèle* et sa *représentation* : le programme de simulation contient non seulement la représentation d'un modèle du système masse-ressort mais aussi




```
Program Oscillator;
const s=3.452; m=7.342
var x,v,t: real;
      k:integer;
begin
 t:=0; v:=0; x:=–1.0;
 for k:=1 to 100 do begin
   v:=v ! 0.1*s*x/m;
   x:=x + 0.1*v;
   t:=t + 0.1;
   print(t,x); end;
 end.
```




```
Program Oscillator;
const L–inv=3.452; C=7.342
var i,q,t: real;
       k:integer;
begin
 t:=0; i:=0; q:=!1.0 ;
 for k:=1 to 100 do begin
   i:=i – 0.1*L!inv*q/C;
   q:=q + 0.1*i;
   t:=t + 0.1;
   print(t,q); end;
end.
```

Fig. 2.2 *Le modèle d'un programme algorithmique s'applique de la même manière à beaucoup de systèmes en changeant les noms des variables.*

celle de nombreux autres oscillateurs. L'interprétation de la représentation dans les termes d'un modèle particulier est strictement l'affaire de l'utilisateur. En termes techniques, la *sémantique* du programme est fixée par l'utilisateur.

2.1.2 Les connaissances ont une interprétation unique

Les programmes d'Intelligence Artificielle, même s'ils sous-tendent des idées originales, n'en demeurent pas moins des *programmes*. Ils sont, par conséquent, sujets aux mêmes limitations que des programmes conventionnels. Les faits du monde réel, quant à eux, doivent être *représentés* conformément à un certain modèle. Il faut cependant noter que les modèles liés à un programme typique d'IA sont souvent beaucoup plus complexes que ceux utilisés par des programmes conventionnels. Ainsi, si la sémantique de l'interprétation des données et des résultats devait toujours être imposée par l'utilisateur, celui-ci serait confronté à l'énorme tâche de maîtriser toutes les connaissances intégrées dans le programme ainsi que leurs significations. La complexité du modèle rendrait l'utilisation du programme beaucoup trop difficile[(1)]. Un problème plus complexe encore est celui de l'interprétation du monde réel par un système autonome comme un robot : si la relation existant entre le modèle informatique et la réalité n'est pas unique, il n'est pas évident d'interpréter les observations effectuées sur le monde réel.

[(1)] Ce phénomène peut également être observé dans les systèmes d'exploitation comme UNIX : peu de personnes en connaissent toutes les fonctionnalités.

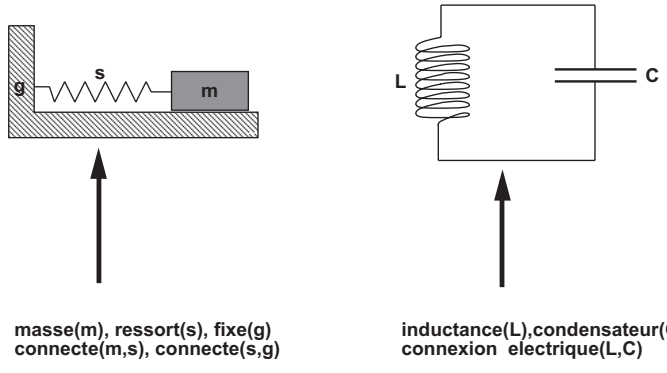


Fig. 2.3 *Les connaissances ne peuvent pas s'interpréter à nouveau en changeant de noms de variables.*

Une convention importante distingue les connaissances des données. Elle consiste à considérer que la signification des symboles représentant la connaissance est non seulement fixe mais qu'elle est en plus proche de celle que les êtres humains lui attribuent : le symbole `rouge` se réfère à la couleur rouge et le symbole `voiture` à une voiture. Ainsi les *connaissances* ne sont autres que des données formulées selon un modèle cognitif humain et pour lesquelles il n'y a qu'une seule interprétation possible.

Le modèle sous-jacent aux échanges d'informations entre êtres humains par le biais du langage ou de diagrammes est en fait un modèle cognitif humain dont les interprétations sont généralement connues. Comme éléments communément admis de ce modèle cognitif figurent par exemple les objets liés à une situation particulière, leurs couleurs, formes et autres caractéristiques. Dans un environnement très spécifique, par exemple celui des échanges bancaires, les objets manipulés comme les chèques et autres lettres de crédit auront une interprétation bien précise. À l'inverse de ceux des modèles numériques, les éléments d'un modèle cognitif sont en correspondance exacte avec la perception que les humains en ont. En conséquence, ils ne peuvent être librement réinterprétés : les modèles d'un oscillateur mécanique et électrique, décrits par la figure 2.3, ne peuvent pas être transformés l'un en l'autre en changeant des noms de variables.

Cela nous permet d'affirmer qu'un programme dont la représentation incorpore par exemple des faits sur les ressorts *connaît* ces faits : il sait comment les interpréter puisque leur signification est connue et admise par tous les utilisateurs. L'existence de cette interprétation unique et unanimement admise est le principal aspect différenciant la connaissance des données[(2)].

La capacité de l'IA de représenter des connaissances est particulièrement mise en valeur lorsque le modèle sous-jacent au programme est identique à celui qui sous-tend la perception humaine. L'utilisateur d'un programme basé sur

[(2)] Notons, cependant, que certains philosophes affirment que tant que l'interprétation reste fixée par convention et non par nécessité, on ne peut toujours pas parler de connaissances.

la connaissance n'a, en conséquence, nul besoin de connaître quoi que ce soit sur le programme pour en interpréter les résultats : cette interprétation est unique et peut être formulée en langue naturelle par exemple. Cela rend possible la construction de systèmes basés sur la connaissance beaucoup plus complexes que des logiciels conventionnels, mais néanmoins utilisables sans qu'il soit nécessaire de connaître beaucoup de détails sur leur fonctionnement. Il devient également possible au programme lui-même de changer son propre modèle en cours d'exécution, par apprentissage ou en réponse aux modifications de son environnement.

Il existe des bases de connaissances standardisées, appelées *ontologies*, qui sont utilisées pour formaliser l'échange d'informations entre différents programmes et bases de données. Le consortium WWW a standardisé des langages pour leur formalisation. Cela permet notamment d'organiser des symboles dans des hiérarchies d'héritage caractérisées par des relations de classe et sous-classe des objets que représentent les symboles. Ils permettent également de définir les propriétés associées à ces objets. De nombreuses organisations mettent à disposition des ontologies qui définissent des termes.

2.2 Représentation de la connaissance

Représenter un modèle cognitif sur un ordinateur nécessite des structures de données beaucoup plus riches que celles offertes par les langages de programmation classiques. Le formalisme communément admis pour représenter des connaissances est celui de la *logique*, plus précisément le *calcul des prédicats du 1er ordre*.

En logique, l'unité d'information élémentaire est la *proposition* qui représente un fait particulier. Des exemples de propositions sont donnés ci-dessous :

```
Jacques pèse 78 kilos.
Lausanne est situé en Amérique.
```

Une proposition peut-être *vraie* ou *fausse* : le premier de nos deux exemples peut aussi bien être une proposition vraie que fausse, alors que le second est sûrement une proposition fausse. Néanmoins, ces propositions sont toutes deux licites.

En calcul des prédicats, une proposition s'exprime par des *formules bien formées*, c'est-à-dire des expressions qui respectent les règles d'écriture du calcul des prédicats. Les formules bien formées sont en fait des expressions construites au moyen des éléments suivants :

- instances : représentées par des symboles comme `Jacques`, `Vert`, `Bloc` ;
- prédicats : symboles prenant un nombre fixe d'arguments, voire aucun argument. Exemples : `Il-pleut`, `Etudiant(Jacques)`, `Age(Jacques,25)` ;
- fonctions : retournant les propriétés d'un symbole, par exemple : `Oncle(Jacques)` ($\rightarrow$ `Pierre`), `Age(Jacques)` ($\rightarrow$ 25) ;
- connecteurs : permettant les combinaisons de formules bien formées :
  - $\wedge$: **et**

  - $\vee$: **ou**
  - $\Rightarrow$: **implique**
  - $\neg$: **non**
- quantificateurs : comme $\forall$ (pour tout) et $\exists$ (il existe), qui sont introduits ici mais ne seront expliqués que plus tard.

Des exemples de formules bien formées peuvent être obtenus par combinaison de ces éléments, comme dans les exemples suivants :

`1.Pèse(Jacques,78)`
« Jacques pèse 78 (kilos). »
`2.¬ Ecrivain(Mozart)`
« Mozart n'est pas un écrivain »
`3.Cousin(Fils(Jean),Neveu(Jean))`
« le fils et le neveu de Jean sont cousins »
`4.Lit(Pierre,Candide)` $\wedge$ `Auteur(Voltaire,Candide)`
« Pierre lit Candide dont l'auteur est Voltaire »
`5.Humain(Socrate)` $\Rightarrow$ `Mortel(Socrate)`
« Socrate est humain donc il est mortel »

Ces éléments sont combinés en formules bien formées selon les règles syntaxiques suivantes, FBF désignant une *formule bien formée* :

$FBF ::=$
$Prédicat \mid FBF \vee FBF \mid FBF \wedge FBF \mid \neg FBF \mid FBF \Rightarrow FBF \mid (FBF)$

Comme pour des expressions algébriques, des parenthèses peuvent être utilisées pour clarifier la priorité des opérateurs. En l'absence de parenthèses, $\wedge$ (et) correspond à la multiplication et est prioritaire par rapport à $\vee$ (ou) qui correspond à l'addition. $\neg$ (non) correspond alors à la négation.

Il faut noter que cette définition du calcul des prédicats de 1er ordre n'est pas minimale. On peut en effet très bien remplacer $F_1 \Rightarrow F_2$ par $\neg F_1 \vee F_2$ et il n'est donc pas nécessaire d'introduire l'implication. L'objectif est plutôt de permettre une expression aussi confortable que possible : il y a des cas où $F_1 \Rightarrow F_2$ est heuristiquement préférable à $\neg F_1 \vee F_2$ et vice versa.

2.3 Règles d'équivalence

En général, le calcul des prédicats ne représente pas forcément un état du monde réel de manière unique. Il existe souvent plusieurs possibilités de formulation. Cela est lié aux *règles d'équivalence* présentées dans ce qui suit :

- $\neg(\neg X_1)$ est équivalent à X_1
  $X_1 \vee X_2$ est équivalent à $\neg X_1 \Rightarrow X_2$
- les lois de Morgan :
  $\neg(X_1 \wedge X_2)$ est équivalent à $\neg X_1 \vee \neg X_2$
  $\neg(X_1 \vee X_2)$ est équivalent à $\neg X_1 \wedge \neg X_2$

- les lois distributives :
  $X_1 \wedge (X_2 \vee X_3)$ est équivalent à $(X_1 \wedge X_2) \vee (X_1 \wedge X_3)$
  $X_1 \vee (X_2 \wedge X_3)$ est équivalent à $(X_1 \vee X_2) \wedge (X_1 \vee X_3)$
- les lois commutatives :
  $X_1 \wedge X_2$ est équivalent à $X_2 \wedge X_1$
  $X_1 \vee X_2$ est équivalent à $X_2 \vee X_1$
- les lois associatives :
  $(X_1 \wedge X_2) \wedge X_3$ est équivalent à $X_1 \wedge (X_2 \wedge X_3)$
  $(X_1 \vee X_2) \vee X_3$ est équivalent à $X_1 \vee (X_2 \vee X_3)$
- la loi de la contraposée :
  $X_1 \Rightarrow X_2$ est équivalent à $\neg X_2 \Rightarrow \neg X_1$

Ces lois justifient en fait certaines simplifications d'écriture des formules bien formées. Ainsi les lois d'associativité permettent d'écrire la conjonction $X_1 \wedge X_2 \wedge \ldots \wedge X_N$ sans aucune parenthèse.

2.4 Exemple de modélisation

Le pays de Lointainie veut automatiser le traitement des voyageurs à la douane. Le règlement actuel prévoit entre autres :

- Tout voyageur adulte a le droit d'importer hors taxe une petite quantité de marchandise.
- Sont considérées comme petites quantités :
  - pour le cognac, moins d'un litre ;
  - pour le vin, moins de deux litres ;
  - pour toute marchandise, une valeur de moins de cent francs.

Le but est de mettre au point un programme permettant de décider si un voyageur doit être taxé ou non. Pour cela, il faut d'abord modéliser le problème formellement. Pour ce faire, on identifie les objets :

voyageur (v), marchandise (m)

et leurs propriétés :

adulte, hors-taxe, petite-quantité, cognac, <-1-litre, <-2-litres, <-100-Frs.

Ensuite, on peut identifier des règles :

adulte(v) $\wedge$ petite-quantité(m) $\Rightarrow$ hors-taxe(m,v)
cognac(m) $\wedge$ <-1-litre(m) $\Rightarrow$ petite-quantité(m)
vin(m) $\wedge$ <-2-litres(m) $\Rightarrow$ petite-quantité(m)
<-100-Frs.(m) $\Rightarrow$ petite-quantité(m)

L'inférence permettra alors d'obtenir des conclusions sur la base du modèle.

2.5 Inférence

L'utilité de la représentation en calcul de prédicats est qu'elle permet de tirer des *inférences* par un programme. Une inférence logique suit le schéma suivant :

> Étant donné un ensemble $\mathcal{P}$ de propositions, trouver des propositions x telles que si toutes les propositions de $\mathcal{P}$ sont vraies, x le sera aussi.

On écrit alors $\{\mathcal{P}\} \vdash x$.

Il existe des preuves que tout calcul algorithmique peut être formulé sous la forme d'un problème d'inférence logique. Il existe donc des langages de programmation généraux (par exemple `PROLOG`) qui sont basés uniquement sur l'inférence logique.

Les possibilités d'inférence peuvent être formalisées par des *règles d'inférence*. Une règle d'inférence prend la forme suivante :

$$P_1 \wedge P_2 \wedge ... \wedge P_n \Rightarrow Q$$

ou $P_1, ..., P_n \in \mathcal{P} \vdash Q$.

Si les règles d'inférence peuvent être formulées sans grande difficulté, leur utilisation sous forme de programme efficace est plus difficile. En fait, la plupart des systèmes pratiques se limitent à la règle du *modus ponens* sur la base de laquelle on peut concevoir des programmes simples et efficaces.

La règle d'inférence du *modus ponens* se définit comme suit :

$$\{(p \Rightarrow q)\} \wedge p \Rightarrow q$$

On peut imaginer d'autres règles d'inférence :

$$\begin{aligned} p \wedge q &\Rightarrow \{p \wedge q\} \quad \text{(Introduction ET)} \\ \{(p \Rightarrow q)\} \wedge \{(q \Rightarrow r)\} &\Rightarrow \{(p \Rightarrow r)\} \quad \text{(Transitivité)} \end{aligned}$$

Une *preuve* logique d'une proposition Q est la trace d'inférence de Q à partir d'un ensemble de faits $\mathcal{P}$. Pour décrire des preuves, nous allons utiliser la notation suivante :

	Proposition	*Justification*
1.	p(A)	prémisse
2.	q(A)	prémisse
3.	p(A) ∧ q(A)	IE 1 2
4.	...	....

où la justification correspond toujours à la manière dont une proposition a été déduite. Nous avons utilisé « IE 1 2 » pour « Introduction d'un Et entre propositions 1 et 2 ». D'autres abréviations sont par exemple « MP » pour *modus ponens*, « IO » pour « Introduction d'un Ou ».

Nous allons voir plus tard des algorithmes servant à construire automatiquement des preuves logiques. Une preuve est utile non seulement pour le résultat

	Proposition	*Justification*
1.	`cognac`	prémisse
2.	`<-1-litre`	prémisse
3.	`adulte`	prémisse
4.	(`cognac` ∧ `<-1-litre`) ⇒ `petite-quantité`	prémisse
5.	(`adulte` ∧ `petite-quantité`) ⇒ `hors-taxe`	prémisse
6.	`cognac` ∧ `<-1-litre`	IE 1 2
7.	`petite-quantité`	MP 6 4
8.	`adulte` ∧ `petite-quantité`	IE 3 7
9.	`hors-taxe`	MP 5 8

Fig. 2.4 *Exemple d'une preuve logique.*

qu'elle fournit, mais également parce qu'elle permet de donner une explication de ce résultat. Par exemple, on peut demander une explication de la conclusion `hors-taxe` du raisonnement de la figure 2.4 :

« Pourquoi hors-taxe ? »

La trace de la preuve peut alors être reformatée pour donner l'explication :

parce-que :

1) `petite-quantité`

2) `adulte`

3) Règlement : `adulte` ∧ `petite-quantité` ⇒ `hors-taxe`

L'interrogation pourrait se poursuivre récursivement par exemple en demandant ensuite :

« Pourquoi petite-quantité ? »

La réponse peut également être obtenue par simple reformulation de la preuve logique. Cette facilité d'explication est souvent importante. Dans l'exemple du règlement douanier, elle protège des erreurs ou des manipulations. Dans une telle application, cette possibilité est indispensable pour une utilisation pratique, car le système doit pouvoir prouver que sa réponse est juste. Il en va de même pour beaucoup d'autres applications de systèmes à base de connaissances, par exemple pour des décisions concernant les droits au dédommagement par une assurance. Elles sont en général très difficiles à donner dans un cadre de programmation procédural où des explications doivent être explicitement programmées.

Un autre avantage de systèmes à base de connaissances est la facilité d'adaptation à des changements. Par exemple, si le règlement change pour introduire une nouvelle manière de traiter des parfums, on pourra simplement ajouter une règle, par exemple :

(`parfum(m)` ∧ `<-50-ml(m)`) ⇒ `petite-quantité`

Dans un programme procédural, par contre, toute la structure logique doit être revue. Il en va de même pour d'autres modifications. Par exemple, l'introduction

d'une nouvelle condition de non-résidence conduit à un changement local d'une règle :

$$(\texttt{petite-quantité}(\mathrm{m}) \wedge \texttt{non-résident(v)}) \Rightarrow \texttt{hors-taxe(m,v)}$$

Son implémentation par un programme classique pourrait être beaucoup plus complexe.

Littérature

Il existe de nombreux livres sur la logique des prédicats, qui est plutôt un sujet de la philosophie et des mathématiques. Le livre de Kowalski [5] a été le premier à évoquer son utilisation pour la programmation.

2.6 Exercice

Pour ce chapitre, nous ne vous proposons pas d'exercice particulier. Par contre, nous vous suggérons, selon votre niveau, d'apprendre ou de réviser le langage de programmation Python, puisque celui-ci est utilisé tout au long de ce livre. Vous pouvez vous documenter si nécessaire au travers des nombreuses ressources en ligne et par les livres existants.

Chapitre 3

Algorithmes d'inférence

Dans un système à base de connaissances, le « calcul » se fait par des *moteurs d'inférence*, des programmes généraux qui permettent de trouver automatiquement des preuves logiques. En général, un moteur d'inférence construit des implications :

$$\{P\} \vdash q$$

La procédure est :

- *fondée* si q est toujours une conséquence de $\{P\}$,
- *complète* si elle trouve toutes les q fondées,
- *complète pour la réfutation* si elle trouve toutes les contradictions de $\{P\}$, c'est-à-dire elle ne manquera jamais $q = \perp$ (contradiction).

L'idéal serait d'avoir un algorithme qui est à la fois fondé et complet. Hélas, par le théorème de Gödel, un tel algorithme ne peut pas exister. Par contre, pour le calcul de prédicats de 1$^{\text{er}}$ ordre, il existe une procédure d'inférence qui est complète pour la réfutation. Il s'agit de la *résolution*. Une procédure qui est complète pour la réfutation peut être utilisée pour construire des preuves indirectes.

3.1 Forme normale

Les équivalences entre expressions ont pour conséquence qu'il y a de nombreuses manières d'exprimer les mêmes connaissances en calcul de prédicats. Cela complique aussi enormément les algorithmes d'inférence qui doivent alors être capables de s'adapter à toutes les manières d'expression. On adopte alors une convention d'une unique *forme normale*.

Nous allons appeler une proposition formée par un prédicat appliqué à des instances ou fonctions d'instances une *proposition simple*, et nous supposons que deux propositions simples ne sont égales que s'elles sont identiques. Nous acceptons ainsi de ne pas considérer des équivalences comme $plus - grand(a, b) \Leftrightarrow plus - petit(b, a)$.

On distingue alors deux formes normales : dans la *forme normale conjonctive*, l'expression est transformée en une conjonction de *clauses* :

$$c_1 \wedge c_2 \wedge ... \wedge c_n$$

où les clauses c_i sont des disjonctions de propositions simples :

$$c_i = p_1 \vee p_2 \vee ... \vee p_m$$

En revanche, la *forme disjonctive normale* représente les expressions sous forme d'une disjonction de conjonctions.

On utilise exclusivement la forme normale conjonctive car elle permet facilement d'exprimer plusieurs expressions qui sont simultanément vraies comme une seule conjonction de toutes les clauses qu'elles contiennent, en supprimant les clauses qui apparaissent en double. Par contre, pour la forme disjonctive cela demanderait une reécriture couteuse des expressions.

À cause de la commutativité, il reste une ambiguïté quant à l'ordre des expressions. On considère que les clauses forment un ensemble sans ordre particulier, appellé une *base de connaissances*. À l'intérieur des clauses, on peut trier les propositions simples par exemple par ordre lexique.

En appliquant les règles d'équivalence dans le bon sens, toute expression logique peut être transformée en forme conjonctive normale. Le plus important est de transformer les implications ($\Rightarrow$) en disjonction, donc $A \Rightarrow B$ devient $\neg A \vee B$. Nous allons supposer que les premisses $\{P\}$ sont représentés comme un ensemble de clauses.

Par exemple, l'expression :

$$(A \vee B) \Rightarrow (C \wedge D)$$

sera réécrit comme :

$$\begin{aligned}
&\neg(A \vee B) \vee (C \wedge D)\\
&(\neg A \wedge \neg B) \vee (C \wedge D)\\
&(\neg A \vee C) \wedge (\neg A \vee D) \wedge (\neg B \vee C) \wedge (\neg B \vee D)
\end{aligned}$$

et donc l'ensemble des quatre clauses :

$$\begin{aligned}
&(\neg A \vee C)\\
&(\neg A \vee D)\\
&(\neg B \vee C)\\
&(\neg B \vee D)
\end{aligned}$$

3.2 Inférence par résolution

La règle de résolution prend deux clauses qui contiennent la même proposition C une fois de façon positive et une fois de façon négative et en produit une nouvelle clause qui est composée du reste des deux clauses :

$$\begin{aligned}
c_i : &\quad C \quad \vee X; X = a \vee ...\\
c_j : \neg &\quad C \quad \vee Y, Y = b \vee ...\\
\Rightarrow c_{ij} : &\qquad X \vee Y
\end{aligned}$$

Un moteur d'inférence basé sur la résolution applique cette règle de façon itérative pour ainsi compléter une base de données $\mathcal{BD}$ avec toutes les inférences qui peuvent être obtenues.

1: Fonction Résolution(P)
2: $\mathcal{BD} \leftarrow \{P\}$
3: **repeat**
4: sélectionner deux clauses $\in \mathcal{BD}$:

$$p_1 : \quad a_1 \ \vee \ ... \ \vee \ a_n \ \vee \ C$$
$$p_2 : \quad b_1 \ \vee \ ... \ \vee \ b_m \ \vee \ \neg C$$

5: appliquer la règle de *résolution* pour obtenir :

$$p_n : a_1 \ \vee \ ... \ \vee \ a_n \ \vee \ b_1 \ \vee \ ... \ \vee \ b_m$$

6: **if** p_n est une clause vide **then**
7: arrêter : contradiction
8: **else**
9: éliminer de p_n toutes les propositions doublées ;
10: **if** $p_n \notin \mathcal{BD}$ **then**
11: ajouter p_n à $\mathcal{BD}$
12: **until** il ne reste plus de combinaisons de clauses qui n'ont pas encore été utilisées
13: retourner $\mathcal{BD}$

Si la fonction Résolution finit sans jamais trouver une clause vide, l'ensemble fourni au départ doit être consistant. Cela peut être prouvé par le fait que la procédure suivante ne manquera pas d'assigner un modèle (vrai ou faux) consistant à toutes les propositions élémentaires qui apparaissent dans $\{P\}$:

1) ordonner les propositions $p_1, ..., p_k$
2) assigner toutes les propositions à `faux`
3) $BD \leftarrow$ Résolution(P)
4) for $i \leftarrow 1$ to k
5) if $\exists clause c \in BD$ qui en plus de p_i ne contient que des $p_j, j < i$ qui sont tous assignées à `faux`, (donc qui ne peut être vrai que si $p_i =$ `vrai`), assigner p_i à `vrai`.
6) end for

La seule façon dont cette procédure pourrait échouer est s'il existe à la fois des clauses qui demandent que p_i soit `vrai` et d'autres qui demandent que p_i soit `faux`. Comme les premiers doivent contenir p_i et les derniers $\neg p_i$, la résolution aurait été appliqué à toutes ces paires et aurait obtenu des clauses ne contenant que des $p_j, j < i$ et qui sont fausses. Donc, la procédure aurait du échouer à

une itération précédente déjà. Pour $i = 1$, soit il y a p_1 et $\neg p_1$, ce qui aurait donné une clause vide, soit la procédure trouve une assignation. Comme elle ne peut pas échouer après, elle fournira une assignation qui satisfait à toutes les clauses.

Cette procédure peut être exploitée pour trouver n'importe quelle preuve en appliquant le principe de la *preuve indirecte*, c'est-à-dire on réduit la preuve de $\{P\} \vdash q$ à la preuve :

$$(\{P\} + \neg q) \vdash \bot$$

où $\bot$ est le symbole de la contradiction qui est toujours faux. Si en fait $\{P\} \vdash q$ est vraie, la procédure de résolution est alors garantie de s'arrêter avec la contradiction qui fait preuve de cela. Lors de la résolution, cette contradiction se manifestera dans la découverte d'une clause vide :

$$X \wedge \neg X \Rightarrow \bot$$

Un moteur d'inférence pour le calcul de prédicats de 1$^{\text{er}}$ ordre basé sur le principe de la résolution est donc garanti de s'arrêter quand cette contradiction sera déduite.

Si la résolution est très générale, elle est aussi très peu efficace. En pratique, on a donc tendance à baser l'inférence logique plutôt sur des règles plus simples dont la plus importante est le *modus ponens* :

$$\{p, p \Rightarrow q\} \vdash q$$

Ce type de règle est la base du *chaînage* de règles, qui se retrouve dans presque tous les moteurs d'inférence utilisés dans la pratique. Le chaînage est complet pour la réfutation de *clauses de Horn*. Une clause de Horn est une implication de la forme :

$$cond_1 \wedge cond_2 \wedge ... \Rightarrow assertion$$

qui n'admet qu'une seule conclusion à la fois. En fait, pour des clauses de Horn le *modus ponens* n'est rien d'autre que la règle de résolution ! Il est important que l'application de la règle de résolution à deux clauses de Horn donne toujours comme résultat une autre clause de Horn. La classe des clauses de Horn est donc *fermée* sous la règle de résolution.

Certaines formes de règles peuvent être traduites en clauses de Horn équivalentes. Tel est le cas quand il y a des disjonctions de conditions :

$$p \vee q \Rightarrow r \simeq p \Rightarrow r,\ q \Rightarrow r$$

ou quand il y a une conjonction des conclusions :

$$p \Rightarrow q \wedge r \simeq p \Rightarrow q,\ p \Rightarrow r$$

Une telle transformation n'est par contre pas possible quand il y a une disjonction des conclusions :

$$p \Rightarrow q \vee r$$

Cela signifie qu'une telle connaissance ne peut pas être traitée par un moteur d'inférence à chaînage, mais seulement par la résolution.

3.3 Inférence propositionnelle par chaînage

La forme la plus classique d'une inférence logique est celle du *modus ponens* :

	`(∀ x) oiseau(x) ⇒ vole(x)`	*« Tous les oiseaux volent »*
	`oiseau(Tweety)`	*« Tweety est un oiseau »*
alors	`vole(Tweety)`	*« Tweety vole »*

La plupart des déductions logiques effectuées dans les systèmes basés sur la connaissance s'appuie sur cette règle, qui s'appelle aussi le *chaînage*. Elle s'applique à des connaissances sous la forme de clauses de Horn.

Rappelons que l'on peut obtenir une procédure d'inférence complète sur clauses de Horn en utilisant le *modus ponens* et des preuves indirectes. Comme la seule manière d'obtenir une contradiction entre clauses de Horn est de trouver une proposition et sa négation, il suit que l'application du *modus ponens* en soi est suffisante pour produire toutes les propositions qui découlent de l'ensemble de prémisses et de règles. On n'a donc pas besoin de spécifier la conclusion souhaitée pour obtenir une contradiction – il suffit d'attendre que la procédure d'inférence produise la proposition souhaitée.

La résolution d'un problème par chaînage implique :

- Une *base de données*, qui contient des propositions jugées vraies. Au départ, elle contient les *prémisses* du problème.
- Une *base de connaissances*, qui contient les règles générales qui seront utilisées pour trouver une solution.
- La spécification d'une solution, qui peut être une proposition complètement ou partiellement spécifiée, ou bien un critère quelconque qui permet de vérifier si une proposition donnée est considérée comme solution.

Il existe alors deux manières de trouver les solutions :

- En chaînage « avant » : à partir des prémisses, appliquer les règles pour produire toutes les conséquences jusqu'au moment où une solution est trouvée. Le chaînage avant est la procédure naturelle d'inférence logique.
- En chaînage « arrière » : à partir de la solution, produire toutes les étapes intermédiaires hypothétiques (sous-buts) qui permettront de déduire une solution jusqu'au moment où un tel sous-but est satisfait par les prémisses. Le chaînage arrière a été proposé comme une procédure qui est plus proche du raisonnement humain. Nous allons considérer le chaînage arrière plus tard dans ce livre.

3.4 Chaînage avant sans variables

La procédure d'inférence à chaînage avant consiste à appliquer toutes les règles possibles à l'ensemble des faits connus, en ajoutant chaque nouvelle conclusion à cet ensemble. Par application itérative du processus, chaque conclusion peut

elle-même satisfaire les conditions d'une autre règle, ce qui conduit à un *chaînage avant* des règles. Le processus s'arrête lorsqu'aucune règle n'est applicable à l'ensemble des faits, ou quand une solution satisfaisante est trouvée.

Nous considérons d'abord un moteur d'inférence pour la logique *propositionnelle* (sans variables) et nous allons y rajouter la possibilité d'utiliser des variables par la suite.

Comme exemple d'inférences que notre moteur d'inférence sera capable de faire, considérons l'exemple suivant où l'on doit décider si une certaine marchandise peut passer hors taxe ou non. On suppose qu'au départ on a les faits et règles suivants :

R1. `vin` $\wedge$ `<-2-litres` $\Rightarrow$ `petite-quantité`
R2. `cognac` $\wedge$ `<-1-litre` $\Rightarrow$ `petite-quantité`
R3. `<-100-Euro` $\Rightarrow$ `petite-quantité`
R4. `petite-quantité` $\wedge$ `adulte` $\Rightarrow$ `hors-taxe`
F1. `vin`
F2. `<-2-litres`
F3. `<-100-Euro`
F4. `adulte`

et que le but sera de prouver :

`hors-taxe`

La procédure de chaînage avant résoudrait ce problème par la suite d'inférences suivante :

R1, F1, F2 $\rightarrow$ F5 : `petite-quantité`
R3, F3 $\rightarrow$ F5 : `petite-quantité`
R4, F5, F4 $\rightarrow$ F6 : `hors-taxe` $\Rightarrow$ **but**

L'efficacité de la procédure dépend essentiellement du nombre de fois auquel on arrive à éviter les inférences inutiles (par exemple, F5 a été trouvé deux fois).

Un *moteur d'inférence* à chaînage avant est appelé avec un ensemble de prémisses, **F** et un ensemble de règles sous la forme de clauses de Horn. Il fait appel aux structures de données suivantes :

- une *base de règles* constituée par les règles qui permettent de réaliser les déductions,
- une *base de données* de faits, qui sont soit des prémisses soit des propositions déduites des prémisses,
- une *file d'attente* pour éviter les boucles.

Le flux d'informations entre ces divers éléments est décrit par la figure 3.1 et l'algorithme par la figure 3.2. La base de règles **R** est passée au moteur d'inférence et ne change pas pendant l'exécution. Les prémisses **F** sont insérées dans une file d'attente. L'état courant de l'inférence est représenté par une *base de données* de faits, qui contient tous les faits qui sont connus comme étant vrais par le moteur d'inférence. Avant son insertion dans cette base de données, tout nouveau fait est d'abord comparé à la base de règles pour déterminer s'il permet de nouvelles inférences. Pour éviter des boucles, chaque fait est examiné dans

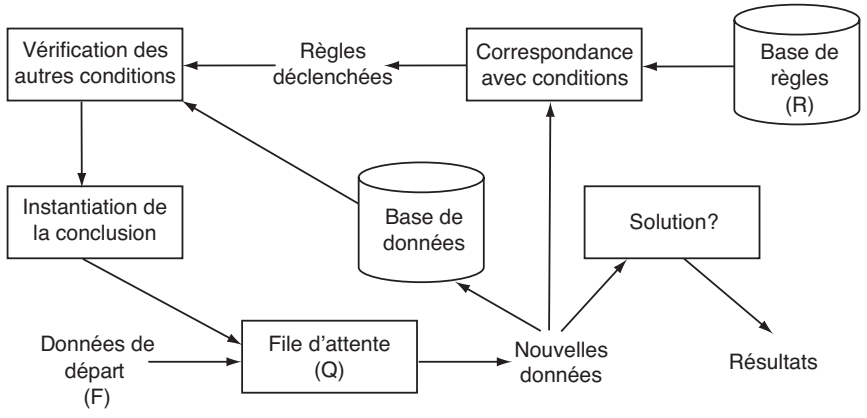


Fig. 3.1 *Flux des informations entre composants d'un moteur d'inférence à chaînage avant.*

```
1: Procédure chaînage-simple(F,R)
2: Q ← {F}
3: while Q n'est pas vide do
4:    q ← first(Q) ; Q ← rest(Q)
5:    if q ∉ base de données then
6:       ajouter q à la base de données
7:       if q est un noeud but then imprimer q
8:       for r ∈ {R} do
9:          if q ∈ conditions(r)
            and conditions(r) ⊆ base de données then
10:            ajouter conclusion(r) à la queue Q
```

Fig. 3.2 *Algorithme pour un moteur d'inférence en chaînage avant, pour calcul propositionnel, sans nécessité de substitution de variables.*

l'ordre de sa découverte à l'aide d'une file d'attente **Q**. Les prémisses passent elles aussi par le même processus.

Si un nouveau fait correspond aux conditions d'une règle, le moteur d'inférence vérifie que toutes les autres conditions font également partie de la base de données. Si c'est le cas, la règle est applicable (on dit aussi qu'elle est *déclenchée*) et la conclusion est un nouveau fait, à condition qu'elle n'existe pas déjà dans la base de données ou la file d'attente **Q**. Le nouveau fait est alors inséré dans la file d'attente pour être traité lors d'une prochaine itération. Notons que la file d'attente sert à éviter que des inférences se fassent en double.

Le processus s'arrête au moment où un nouveau fait constitue une solution satisfaisante au problème, selon une spécification donnée par l'utilisateur. Cette

spécification peut être une proposition précise qui doit être prouvée, une proposition contenant des variables dont on cherche les valeurs, ou des contraintes qui permettent de vérifier qu'une proposition remplit les critères d'une solution. Le test se fait au moment de l'insertion dans la base de données, ce qui permet également de traiter le cas où la solution se trouve déjà parmi les prémisses.

3.5 Expression de connaissances générales grâce aux quantificateurs

La logique propositionnelle nous permet uniquement d'exprimer des connaissances relatives à des situations *particulières*. Elle est insuffisante pour décrire des connaissances générales s'appliquant à plusieurs situations. Cela est dû à l'absence de *variables*. L'utilisation de variables permet d'exprimer qu'une proposition est vraie pour toutes les substitutions possibles d'une variable donnée.

En calcul des prédicats, il est possible d'utiliser des variables comme substitut des instances. Chaque variable doit alors être définie au moyen d'un quantificateur. En général, le calcul des prédicats du 1er ordre permet les deux quantificateurs suivants :

- $\forall$: c'est le quantificateur universel. Placé devant une formule $P(x)$, il donne une proposition vraie pour toute interprétation vérifiant P, *quelles que soient* les instances du domaine de x.
- $\exists$: c'est le quantificateur existentiel. Placé devant une formule $P(x)$, il donne une proposition vraie pour toutes les interprétations pour lesquelles *il existe au moins* une instance du domaine de x vérifiant P.

Grâce à la quantification, on peut maintenant exprimer des connaissances générales comme :

`(∀ x) [Eléphant (x) ⇒ Couleur (x, Gris)]`
« Tous les éléphants sont gris »
ou encore
`(∃ x) [Eléphant (x) ∧ Fait-des-numéros-de-cirque(x)]`
« Il existe au moins un éléphant qui fait des numéros de cirque »

Le calcul des prédicats avec quantification permet de distinguer des ambiguïtés qui existent dans la langue naturelle. La phrase : « Tout le monde parle une langue » peut être traduite par deux alternatives :

a. `(∀ x) personne(x) ⇒ (∃ y) (langue(y) ∧ parle(x,y))`

b. `(∃ y) langue(y) ∧ (∀ x) (personne(x) ⇒ parle(x,y))`

qui distinguent les cas (a) où tout le monde parle une langue quelconque et (b) où il existe une langue précise que tout le monde parle.

Selon le type de quantification admis, on distingue le calcul de prédicats de

- ordre 0 : aucune quantification possible ;
- 1er ordre : quantification sur les individus ;
- 2^{e} ordre : quantification sur les individus et les prédicats.

Des algorithmes d'inférence existent uniquement pour la logique d'ordre 0 et du 1er ordre. Par conséquent, le calcul des prédicats de 2e ordre n'est presque jamais utilisé. Il serait cependant nécessaire pour exprimer des connaissances telles que :

> « Toutes les propriétés des liquides sont également valables sur la lune. »

Heureusement, on n'a que rarement besoin de formuler de telles connaissances.

Le logicien Skolem a montré qu'on peut éliminer tout quantificateur existentiel en introduisant une fonction, appelée *fonction de Skolem*. Considérons un quantificateur existentiel qui se trouve à l'intérieur du domaine d'un autre quantificateur universel, comme dans le schéma général suivant :

`(∀ x) [ (∃ y) p(x,y) ]`

On introduit une *fonction de Skolem* $\mathtt{f}_y\mathtt{(x)}$ qui retournera pour tout `x` un `y` tant que `p(x,y)` est vrai. Par la suite, on écrit alors :

$\mathtt{(\forall\ x)\ p(x,f}_y\mathtt{(x))}$

On utilisera une fonction de Skolem différente pour chaque quantificateur existentiel et toutes les variables à quantification universelle dont le domaine s'intersecte avec celui du quantificateur existentiel seront des arguments de la fonction. Si le quantificateur existentiel se trouve en dehors de tout quantificateur universel, la fonction de Skolem sera une constante.

Une fois que les quantificateurs existentiels sont éliminés, toutes les variables restantes seront soumises à une quantification universelle et il n'y aura plus d'interdépendance entre elles. On peut donc laisser tomber tous les quantificateurs universels en introduisant la convention que toute variable libre est toujours quantifiée de manière universelle. On arrive ainsi à une formulation plus compacte et plus lisible.

Par exemple, l'expression

`(∀ x) personne(x) ⇒ (∃ y) (langue(y) ∧ parle(x,y))`

peut ainsi être transformée d'abord en :

$\mathtt{(\forall\ x)\ personne(x)} \Rightarrow \mathtt{(langue(f}_{langue}\mathtt{(x))} \wedge \mathtt{parle(x,f}_{langue}\mathtt{(x))}$

et ensuite en :

$\mathtt{personne(x)} \Rightarrow \mathtt{(langue(f}_{langue}\mathtt{(x))} \wedge \mathtt{parle(x,f}_{langue}\mathtt{(x))}$

D'après la signification des quantificateurs, il est également possible de formuler les lois d'équivalences suivantes :

$(\exists x)P(x)$ est équivalent à $\neg\ (\forall x)[\neg P(x)]$
$(\forall x)P(x)$ est équivalent à $\neg\ (\exists x)[\neg P(x)]$
$(\forall x)[P(x) \wedge Q(x)]$ est équivalent à $(\forall x)P(x) \wedge (\forall y)Q(y)$
$(\exists x)[P(x) \vee Q(x)]$ est équivalent à $(\exists x)P(x) \vee (\exists y)Q(y)$
$(\forall x)P(x)$ est équivalent à $(\forall y)P(y)$
$(\exists x)P(x)$ est équivalent à $(\exists y)P(y)$

3.5.1 Appliquer des connaissances quantifiées : unification

Pour que de nouvelles propositions puissent être déduites d'un ensemble donné d'autres propositions, il faut qu'il existe des connaissances quantifiées, généralement valides, applicables au problème.

Pour cela, un moteur d'inférence doit d'abord trouver des *correspondances* entre une représentation des données quantifiées et la représentation d'une proposition. Ces correspondances seront exprimées par des valeurs attribuées aux variables qui apparaissent dans les formules quantifiées. La détermination de ces correspondances fait l'objet du mécanisme de *filtrage* (*pattern matching*) et plus généralement de celui de l'*unification*.

3.5.2 Filtrage (*pattern matching)*

Le mécanisme de filtrage consiste, comme son nom l'indique, à *filtrer* une donnée avec une expression contenant éventuellement des variables (cette expression jouant le rôle de filtre). Cela permet de dégager les correspondances existant entre la donnée et le filtre. Dans l'exemple suivant, il s'agit de filtrer la donnée `Paul regarde Pierre` avec le filtre `(?X regarde ?Y)` contenant les deux variables `?X` et `?Y` :

```
(filtrer '(Paul regarde Pierre) '(?X regarde ?Y))
       → ((?X Paul) (?Y Pierre))
```

Le résultat obtenu consiste en une liste d'associations indiquant pour quelles substitutions des variables le filtre correspond à la donnée. Si le filtre ne contenait aucune variable, la fonction `filtrer` se contenterait de retourner une liste vide lorsque le filtre et la donnée sont identiques et `ECHEC` dans le cas contraire.

Un algorithme de filtrage récursif est donné à la figure 3.3. La notation $\{E_i/V_i\}$ signifie que l'on construit une association de la variable V_i avec le terme E_i.

3.5.3 Unification

L'unification correspond en fait à un mécanisme de filtrage pour lequel on admet que la donnée elle-même contient également des variables (et non pas seulement le filtre comme c'est le cas pour le filtrage). En d'autres termes, il s'agit d'*unifier* deux filtres pour y déceler des correspondances. L'exemple suivant décrit ce que l'on entend par unification :

```
(unifier '(?X est un éléphant) '(?Y est un ?ANIMAL))
       → ((?X ?Y) (?ANIMAL éléphant))
```

L'écriture d'un unificateur pose un peu plus de problèmes que celle d'un mécanisme de filtrage. En effet, étant donné que des variables peuvent apparaître dans les deux arguments à filtrer, des phénomènes de circularité indésirables peuvent se produire, ainsi que le démontrent les exemples suivants (circularités directe et indirecte) :

```
(unifier '?x '(f ?x))
```

ou

```
(unifier '(?x ?z) '(?z (f ?x))
```

1: **Function** Filtrer(datum,pattern)
2: **if** pattern est un symbol **then**
3: **if** datum et pattern sont identiques **then return** {}
4: **if** pattern est une variable **then return** {pattern/datum}
5: **return** ECHEC
6: **if** datum est un symbol **then return** ECHEC
7: $F_1 \leftarrow$ premier élément de datum, $T_1 \leftarrow$ reste de datum
8: $F_2 \leftarrow$ premier élément de pattern, $T_2 \leftarrow$ reste de pattern
9: $Z_1 \leftarrow$ FILTRER(F_1,F_2)
10: **if** Z_1 = ECHEC **then return** ECHEC
11: $G_1 \leftarrow T_1$
12: $G_2 \leftarrow$ remplacer les variables de T_2 par les unifications Z_1
13: $Z_2 \leftarrow$ FILTRER(G_1,G_2)
14: **if** Z_2 = ECHEC **then return** ECHEC
15: **return** { $Z_1 \cup Z_2$ }

Fig. 3.3 *Algorithme récursif de filtrage.*

Il est par conséquent nécessaire de tester ce genre de configurations circulaires avant d'engager le mécanisme d'unification à proprement parler.

La figure 3.4 présente un algorithme d'unification récursif. Cet algorithme n'est autre que celui de la figure 3.3 auquel ont été ajoutés des tests détectant la circularité.

3.5.4 Chaînage avant avec variables

Le moteur d'inférence en chaînage avant peut maintenant être étendu pour admettre également des règles qui contiennent des variables. L'introduction de variables changera peu le schéma de la figure 3.1. La mise en correspondance d'un nouveau fait avec la base de règles est remplacée par l'application de l'algorithme de filtrage et il se peut qu'il en résulte plusieurs correspondances avec des substitutions de variables différentes. Chacune de ces substitutions est utilisée pour une instanciation différente de la conclusion. Le schéma modifié est donné par la figure 3.5. La figure 3.6 donne la version modifiée de l'algorithme simple de la figure 3.2.

Comme exemple pour illustrer le processus, considérons les prémisses suivantes :

F1 : `père(Jacques,Charles)`,
F2 : `frère(Charles,François)`,
F3 : `frère(Jacques,Pierre)`

une base de connaissances qui consiste en deux règles :

R1 : `père(?x,?y)` $\wedge$ `frère(?y,?z)` $\Rightarrow$ `père(?x,?z)`
R2 : `père(?x,?y)` $\wedge$ `frère(?x,?z)` $\Rightarrow$ `oncle(?z,?y)`

```
1: Function Unifier(E1,E2)
2: if E1 ou E2 est un symbole then
3:    Interchanger les arguments de E1 et de E2 (si nécessaire) de sorte que E1 soit
      un atome
4:    if E1 et E2 sont identiques then return {}
5:    if E1 est une variable then
6:       if E1 apparaît dans E2 then return ECHEC
7:       return {E1 / E2}
8:    if E2 est une variable then return {E2 / E1}
9:    return ECHEC
10: F1 ← premier élément de E1, T1 ← reste de E1
11: F2 ← premier élément de E2, T2 ← reste de E2
12: Z1 ← UNIFIER(F1,F2)
13: if Z1 = ECHEC then return ECHEC
14: G1 ← remplacer les variables de T1 par les substitutions Z1
15: G2 ← remplacer les variables de T2 par les substitutions Z1
16: Z2 ← UNIFIER(G1,G2)
17: if Z2 = ECHEC then return ECHEC
18: return { Z1 ∪ Z2 }
```

Fig. 3.4 *Algorithme récursif d'unification.*

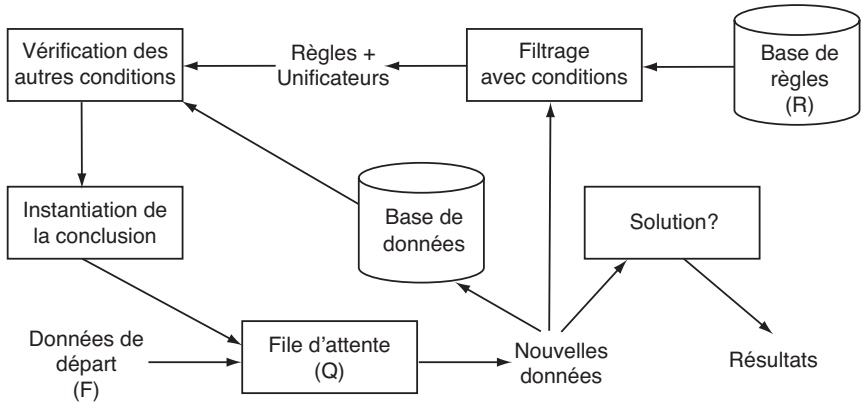


Fig. 3.5 *Flux des informations entre composants d'un moteur d'inférence à chaînage avant avec variables.*

et le but de l'inférence :

`oncle(?x,François)`

La procédure de chaînage avant résoudrait le problème décrit ci-dessus par les inférences suivantes :

R1, F1, F2 → F4 : `père(Jacques,François))`
R2, F1, F3 → F5 : `oncle(Pierre,Charles)`
R2, F4, F3 → F6 : `oncle(Pierre,François)` ⇒ **but**

L'efficacité de la procédure dépend essentiellement du degré auquel on arrive à limiter les inférences inutiles (par exemple F5 dans l'exemple ci-dessus).

L'algorithme d'inférence, décrit en détail dans la figure 3.6, consiste principalement à filtrer les faits avec les déclencheurs des règles de la base de connaissances. Chaque paire de (fait, règle) où le fait correspond à une condition de la règle (par filtrage), est placée avec ses substitutions dans la file. Lorsque tous les filtrages sont opérés, l'algorithme construit itérativement les conclusions des éléments de la file, en commençant par le premier élément. Chaque nouvelle conclusion est immédiatement comparée à la base de connaissances pour voir si elle correspond à l'une des règles (c'est-à-dire si elle est susceptible de la *déclencher*). Toutes les correspondances sont insérées en fin de file. Le processus s'arrête si une nouvelle conclusion constitue une solution satisfaisante au problème.

```
1: Procédure Chaînage-Avant-Variables(F,R)
2: Q ← faits de départ F
3: while Q n'est pas vide do
4:    q ← premier(Q), Q ← reste(Q)
5:    if q ∉ base de données then
6:       imprimer q and ajouter q à la base de données
7:       for chaque règle r de la base de règles R do
8:          if ∃ c ∈ conditions(r) FILTRER(q,c) ≠ ECHEC then
9:             for toute combinaison C d'éléments de la base de données tel que FIL-
               TRER(C,conditions(r)) ≠ ECHEC do
10:               n ← instanciation de conclusion(r) avec les substitutions de FIL-
                  TRER(C,conditions(r))
11:               if n ∉ base de données then ajouter n en queue de Q
```

Fig. 3.6 *Algorithme pour un moteur d'inférence en chaînage avant avec variables.*

3.6 Inférence par résolution avec variables

Si la règle du *modus ponens*, qui est à la base du moteur d'inférence à chaînage-avant, constitue un moyen très naturel de faire des déductions, elle reste cependant limitée à des clauses de Horn. La règle de la *résolution*, quant à elle,

permet une procédure de preuve complète. Elle s'utilise dans le contexte d'une *preuve indirecte* : une proposition but G se déduit d'un ensemble de prémisses {P} et d'un ensemble de règles {R} si et seulement si l'ensemble

$$\{\mathrm{P}\} \cup \{\mathrm{R}\} \cup \{\neg\ \mathrm{G}\ \}$$

est contradictoire. Avant d'appliquer une telle procédure, les règles d'inférence seront traduites en leurs équivalents conformément à la règle suivante :

$$(\mathrm{A} \Rightarrow \mathrm{B}) \Leftrightarrow (\neg\ \mathrm{A} \vee \mathrm{B})$$

Toute expression est alors simplifiée en une forme canonique selon les étapes suivantes : élimination des quantificateurs, rendre les noms uniques, transformer en conjonction de disjonctions. Ensuite, chaque disjonction dans la conjonction devient une *clause* et toutes les clauses sont valables simultanément.

Un moteur d'inférence basé sur la résolution ajoute la négation d'un but donné à l'ensemble des faits existants et utilise l'algorithme pour déduire une contradiction dans cet ensemble. Du fait que la proposition but doit être précisément spécifiée à la procédure de résolution, de tels moteurs d'inférences sont aussi appelés *démonstrateurs de théorèmes.*

La résolution se base sur les deux règles d'inférence suivantes :

1) résolution binaire :

$$(L_1 \vee A), (\neg L_2 \vee B),\ Unificateur(L_1, L_2) = U$$
$$\Rightarrow (U(A) \vee U(B))$$

2) factorisation :

$$(L_1 \vee L_2 \vee A),\ Unificateur(L_1, L_2) = U$$
$$\Rightarrow (U(L_2) \vee A)$$

La règle de résolution binaire est appliquée à toute paire de clauses qui contient des parties L_1 et $\neg L_2$ telles que L_1 et L_2 peuvent s'unifier. La règle de factorisation s'applique quand une clause contient deux parties L_1 et L_2 qui peuvent s'unifier entre elles et sont donc redondantes.

Un moteur d'inférence par résolution applique ces deux règles de façon itérative à tous les endroits possibles jusqu'à ce qu'il n'y ait plus aucune possibilité d'application. Cette procédure est garantie de trouver une contradiction (p et $\neg p$) si elle existe. Par contre, elle peut être très inefficace.

Pour illustrer le fonctionnement de la résolution, considérons l'exemple suivant. En partant des faits initiaux :

1. $\neg lapin(x) \vee animal(x)$
   $(=$ `lapin(x)` $\Rightarrow$ `animal(x)`$)$
2. $\neg animal(y) \vee bouge(y)$
   $(=$ `animal(y)` $\Rightarrow$ `bouge(y)`$)$

on veut prouver :

$$\neg lapin(z) \vee bouge(z)$$
$$(=\texttt{lapin(z)} \Rightarrow \texttt{bouge(z)})$$

La transformation de ce but en sa négation donne :

$$lapin(z) \wedge \neg bouge(z)$$

ce qui se traduit en deux clauses élémentaires qui sont ajoutées à la base de faits initiale :

3. $lapin(z)$
4. $\neg bouge(z)$

En appliquant la règle de résolution binaire deux fois (1.+2., 4.+5.), on trouve :

5. $\neg lapin(x) \vee bouge(x)$
6. $\neg lapin(z)$

et donc une contradiction entre 6. et 3. qui sert comme preuve de l'hypothèse.

Comme dans cet exemple il n'y avait aucun conflit entre les noms des variables, la factorisation n'était pas nécessaire.

Littérature

Robinson a publié la règle de résolution en 1965 dans [6]. Les clauses de Horn ont été introduites dans [7], et [8] a décrit pour la première fois leur utilisation dans un moteur d'inférence. Le livre de Gallier [9] donne un aperçu plus complet des méthodes d'inférence.

Outils - domaine public

Il y a plusieurs moteurs d'inférence pour les business rules, notamment `Drools` :

`http ://www.drools.org`

et `OpenRules` (disponible aussi en version commerciale) :

`http ://www.openrules.com`

L'outil `CLIPS`, également en domaine public, existe depuis longtemps mais est toujours mise à jour et utilisé dans de nombreuses projets :

`http://clipsrules.net/`

Il existe plusieurs logiciels de démonstration automatique de théorèmes qui sont disponibles dans le domaine public. Un des premiers était le système `Otter` et son successeur `Prover9`, qui utilisent le principe de la résolution :

`http://en.wikipedia.org/wiki/Otter_theorem_prover`

Plus récent, on peut considérer `E`, developpé depuis 1998 et parmi les plus puissants aujourd'hui :

`http://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html`

Une autre méthode est de prouver des théorèmes par réduction à SAT, comme le système `KODKOD` issu du MIT :

`http://alloy.mit.edu/kodkod/`

Ce type de logiciel ne se base cependant pas sur l'inférence explicite, mais sur la satisfaction de contraintes que nous allons voir plus tard dans ce livre au chapitre 8.

Outils - commercial

Les moteurs d'inférence sont le plus souvent integrés dans des outils de business rules. Ils sont commercialisés par presque toute les grandes fournisseurs de logiciel, parmi eux IBM Business Rules Management System :

`http://www-01.ibm.com/software/websphere/products/business-rule-management/`

Microsoft Business Rules Engine :

`https://msdn.microsoft.com/en-us/library/aa561216.aspx`

ou encore SAP Business Rule Framework :

`https://en.wikipedia.org/wiki/BRFplus`

Plus généralement, on trouve des moteurs d'inférence dans de nombreuses outils pour l'implémentation de systèmes intelligents. Le fournisseur de logiciel le plus engagé est IBM avec sa division de *cognitive computing*.

Application : Traitement de requêtes de transfert par business rules

La société Thames Water s'occupe de l'approvisionnement d'eau de toute la région de Londres. Elle doit traiter chaque année 250 000 demandes de transfert concernant la tarification de l'approvisionnent d'eau pour des constructions ou rénovations de la région.
Auparavant, ces demandes étaient traitées par une équipe de trente personnes. Vu la complexité des situations et des règles appliquées, il paraissait impossible d'automatiser leur traitement. À partir de l'an 2000, la société a néanmoins réussi à le faire par des *business rules*, des règles à chaînage-avant.
Le nouveau système a permis aux employés de se concentrer sur la réalisation des changements demandés et a ainsi diminué les délais de réponse de 50%. Les coûts ont été réduits de trois millions de livres par an. De plus, on a pu corriger des calculs erronés et ainsi identifier un découvert de 1.4 million de livres qui a pu être récupéré.

On observe là un résultat typique de l'introduction de *business rules* pour automatiser des processus administratifs.

(Source : "Britain's Thames Water Delivers Millions in Cost Benefits Using ILOG JRules." The Free Library, 31 May 2005. `http ://www.thefreelibrary.com/`)

3.7 Exercices

Exercice 3.1 Première partie - sans variable
L'inférence à chaînage avant est à la base de la plupart des systèmes de raisonnement automatiques utilisés aujourd'hui. Cette série d'exercices a pour but d'introduire cette technique. La réalisation d'un tel système se fera en plusieurs étapes, la première se limitant à des règles sans variables. Dans les étapes suivantes, vous implémenterez un moteur d'inférence qui utilisera des règles avec variables.

Modules squelettes

Les modules de cette section fournissent le squelette du programme que nous allons développer. Le module `exemple_impots_sans_variables.py` représente le code d'un fichier test.

Module `.../moteur_sans_variables/regle_sans_variables.py` :

```
class RegleSansVariables:
    def __init__ ( self , conditions , conclusion ):
        self .conditions = set(conditions)
        self .conclusion = conclusion

    def depend_de(self, fait ):
        print('à compléter')

    def satisfaite_par ( self , faits ):
        print('à compléter')

    def __repr__ ( self ):
        print('à compléter')
```

Module `.../moteur_sans_variables/connaissance.py` :

```
class BaseConnaissances:
    def __init__ ( self , constructeur_de_regle ):
        self . faits = []
        self . regles = []
        self . constructeur_de_regle = constructeur_de_regle

    def ajoute_un_fait ( self , fait ):
        self . faits .append(fait)

    def ajoute_faits ( self , faits ):
        self . faits .extend( faits )
```

```
    def ajoute_une_regle( self , description ):
        regle = self . constructeur_de_regle ( description )
        self . regles . append(regle)

    def ajoute_regles ( self , descriptions ):
        for description in descriptions :
            self . ajoute_une_regle ( description )
```

Module `.../moteur_sans_variables/chainage.py` :

```
class Chainage:
    __indentation = 4 * ' '

    def __init__ ( self , connaissances):
        self . trace = []
        self . solutions = []
        self . connaissances = connaissances

    def reinitialise ( self ):
        self . trace = []
        self . solutions = []

    def chaine( self ):
        # Nous retournons un ensemble vide dans ce cas.
        return self. solutions

    def affiche_trace ( self , indent=None):
        if indent is None:
            indent = Chainage.__indentation

        print('Trace:')
        for evenement in self. trace :
            print('{}{}'.format(indent, evenement))

    def affiche_solutions ( self , indent=None):
        if indent is None:
            indent = Chainage.__indentation

        if len( self . solutions ) > 0:
            print('Faits d\'eduits :')
            for fait in self . solutions :
                print('{}{}'.format(indent, fait))
        else:
            print('Aucun fait trouv\'e.')
```

Module `.../moteur_sans_variables/chainage_avant_sans_variables.py` :

```
from .chainage import Chainage

class ChainageAvantSansVariables(Chainage):
    def chaine( self ):
        print('à compléter')
```

Module `.../exemple_impots_sans_variables.py` :

```
from sys import argv, exit
from moteur_sans_variables.regle_sans_variables import RegleSansVariables
from moteur_sans_variables.connaissance import BaseConnaissances
from moteur_sans_variables.chainage_avant_sans_variables import ChainageAvantSansVariables

# La description d'une règle est une liste de deux éléments:
# une liste de conditions et une conclusion.
regles = [
            [['pas-d-enfants'], 'réduc-enfant-0'],
            [['enfants'], 'réduc-enfant-100'],
            [['bas-salaire'], 'réduc-loyer-200'],
            [['moyen-salaire'], 'réduc-loyer-100'],
            [['haut-salaire'], 'réduc-loyer-0'],
            [['pas-de-loyer'], 'réduc-loyer-0'],
            [['petit-trajet'], 'réduc-trajet-0'],
            [['réduc-enfant-0', 'long-trajet'], 'réduc-trajet-100'],
            [['réduc-loyer-0', 'long-trajet'], 'réduc-trajet-100'],
            [['réduc-enfant-100', 'réduc-loyer-100', 'long-trajet'],
             'réduc-trajet-50'],
            [['réduc-enfant-100', 'réduc-loyer-200', 'long-trajet'],
             'réduc-trajet-0'],
            [['réduc-enfant-0', 'réduc-loyer-0', 'réduc-trajet-0'],
             'réduc-0'],
            [['réduc-enfant-100', 'réduc-loyer-0', 'réduc-trajet-0'],
             'réduc-100'],
            [['réduc-enfant-0', 'réduc-loyer-100', 'réduc-trajet-0'],
             'réduc-100'],
            [['réduc-enfant-100', 'réduc-loyer-100', 'réduc-trajet-0'],
             'réduc-200'],
            [['réduc-enfant-0', 'réduc-loyer-200', 'réduc-trajet-0'],
             'réduc-200'],
            [['réduc-enfant-100', 'réduc-loyer-200', 'réduc-trajet-0'],
             'réduc-300'],
            [['réduc-enfant-0', 'réduc-loyer-0', 'réduc-trajet-50'],
             'réduc-50'],
            [['réduc-enfant-100', 'réduc-loyer-0', 'réduc-trajet-50'],
             'réduc-150'],
            [['réduc-enfant-0', 'réduc-loyer-100', 'réduc-trajet-50'],
             'réduc-150'],
            [['réduc-enfant-100', 'réduc-loyer-100', 'réduc-trajet-50'],
             'réduc-250'],
            [['réduc-enfant-0', 'réduc-loyer-200', 'réduc-trajet-50'],
             'réduc-250'],
            [['réduc-enfant-0', 'réduc-loyer-200', 'réduc-trajet-50'],
             'réduc-250'],
            [['réduc-enfant-100', 'réduc-loyer-200', 'réduc-trajet-50'],
             'réduc-350'],
            [['réduc-enfant-0', 'réduc-loyer-0', 'réduc-trajet-100'],
             'réduc-100'],
            [['réduc-enfant-100', 'réduc-loyer-0', 'réduc-trajet-100'],
             'réduc-200'],
            [['réduc-enfant-0', 'réduc-loyer-100', 'réduc-trajet-100'],
             'réduc-200'],
            [['réduc-enfant-100', 'réduc-loyer-100', 'réduc-trajet-100'],
             'réduc-300'],
            [['réduc-enfant-0', 'réduc-loyer-200', 'réduc-trajet-100'],
             'réduc-300'],
```

```
            [[ 'réduc-enfant-100', 'réduc-loyer-200', 'réduc-trajet-100'],
             'réduc-400'],
        ]

if len(argv) < 2 or argv[1].lower() not in ('a', 'b'):
    print('On attend au moins un arguments: A ou B')
    exit(1)

if argv[1].lower() == 'a':
    faits_initiaux = ['bas-salaire', 'loyer', 'enfants', 'long-trajet']
elif argv[1].lower() == 'b':
    faits_initiaux = ['pas-d-enfants', 'pas-de-loyer',
                      'haut-salaire', 'long-trajet']

bc = BaseConnaissances(lambda descr: RegleSansVariables(descr[0], descr[1]))
bc.ajoute_faits( faits_initiaux )
bc.ajoute_regles( regles )

moteur = ChainageAvantSansVariables(bc)
moteur.chaine()

moteur.affiche_solutions()

if len(argv) > 2 and argv[2].lower() == 'trace':
    # Utile durant le déboggage.
    moteur.affiche_trace()
```

Idée de base

L'idée de base d'un moteur d'inférence à chaînage avant est de déduire toutes les faits possibles à partir d'un ensemble de règles et de faits initiaux, c'est-à-dire de propositions qui sont tenues pour vraies dès le départ. Chaque fois qu'un nouveau fait est déduit, l'ensemble des règles doit être appliqué à nouveau à la base des faits : il est en effet possible que le fait nouvellement déduit permette le déclenchement d'une règle qui a déjà été essayée auparavant sans succès. Le processus d'inférence se termine lorsque plus aucun fait nouveau ne peut être déduit.

Exercice 3.1.1 Les faits et les règles

Dans cette série, les propositions ne contiendront pas de variables et seront représentées par des chaînes de caractères (`string`) contenant leur description en langage naturel. Les faits seront donc des propositions. En outre, les règles seront des clauses de Horn, et donc composées de deux parties :

- Un ensemble de conditions (des propositions qui doivent être toutes satisfaites pour que la règle se déclenche) ;
- Une seule conclusion (une proposition qui pourra le cas échéant être inséré dans la base des faits).

Les règles seront ainsi représentées par la classe `RegleSansVariables` du module `regle_sans_variables.py`. Cette classe possède deux attributs : une liste de propositions, qui représentent les conditions, et une proposition, qui représente

la conclusion. Pour utiliser `RegleSansVariables`, vous devez donc compléter les méthodes :

- `depend_de(self, fait)`, qui doit retourner `True` si le fait passé en argument fait partie des conditions ;
- `satisfaite_par(self, faits)`, qui doit retourner `True` si toutes les conditions de déclenchement de la règle sont présentes dans la liste de faits passée en argument ;
- `__repr__`, qui retourne une représentation d'une règle sous forme de `string`. Cela nous permettra d'afficher les règles de manière plus pratique en utilisant la syntaxe `print(règle)`, au lieu de `print(règle.conditions)` et `print(règle.conclusion)`.

Pensez à utilisez l'opérateur `in` pour écrire `depend_de` et la méthode `issubset` de la classe `set` pour implémenter `satisfaite_par`.

Les faits et les règles pertinents pour un problème seront collectés dans la classe `BaseConnaissances`, qui est décrite dans le module `connaissance.py`.

Exercice 3.1.2 Le moteur d'inférence à chaînage avant sans variables

Vous disposez maintenant du code nécessaire pour implémenter le moteur à chaînage avant sans variables. Ce code est à implémenter dans `chainage_avant_sans_variables.py`, en complétant la classe `ChainageAvantSansVariables`. Notez que cette dernière est une sous-classe de la classe `Chainage` du module `chainage.py` et qu'elle hérite par conséquent des deux méthodes `affiche_solutions` et `affiche_trace`, qui servent à afficher les résultats et le parcours de l'algorithme. La classe `ChainageAvantSansVariables` doit recevoir une instance de `BaseConnaissances` en tant que paramètre de son constructeur. C'est à partir du contenu de cette base de connaissance qu'elle recherchera des faits nouveaux.

Votre tâche consiste à implémenter la méthode `chaine` dans la classe `ChainageAvantSansVariables`. N'oubliez pas de placer les faits déduits dans la variable `self.solutions` au moment où ils sont découverts. Vous pouvez également ajouter les règles et les faits à `self.trace` à mesure qu'ils interviennent dans l'inférence.

Pour rappel, l'algorithme à implémenter est le suivant :

```
ChainageAvantSansVariables(faits_depart, regles)
1.  solutions <− liste vide
2.  Q <− faits_depart
3.  WHILE Q n'est pas vide DO
4.      q <− premier(Q)
5.      Q <− reste(Q)
6.      IF q n'est pas dans solutions THEN
7.          ajouter q à solutions
8.          FOR EACH règle r de regles DO
9.              IF r.depend_de(q) et r. satisfaite_par (solutions) THEN
10.                 ajouter la conclusion de r en queue de Q
11.             END IF
12.         END FOR
13.     END IF
14. END WHILE
15. RETURN solutions
END ChainageAvantSansVariables
```

Rappelez-vous qu'en Python, les listes peuvent s'employer comme des queues grâces aux méthodes `append` et `pop`.

Test du programme

`exemple_impots_sans_variables.py` contient les règles et les faits nécessaires pour le calcul du montant d'une réduction d'impôts. Après avoir écrit votre programme, testez-le sur un premier exemple en ajoutant l'option `A` et vérifiez que le fait `'réduc-300'` est correctement déduit. Vous pouvez afficher la trace en utilisant l'option `trace`.

```
python3 exemple_impots_sans_variables.py A
python3 exemple_impots_sans_variables.py A trace
```

Le module contient un deuxième exemple. Quelle devrait être alors la réduction d'impôts ?

```
python3 exemple_impots_sans_variables.py B
python3 exemple_impots_sans_variables.py B trace
```

Solutions à la page 343

Exercice 3.2 Deuxième partie - avec variables

Le but de cette série est de développer un moteur d'inférence à chaînage avant capable de manipuler des règles comportant des variables. Dans un premier temps, vous devrez étendre l'implémentation des règles en complétant quelques fonctions utilitaires. Puis vous construirez un filtre, qui permettra de comparer deux propositions dont l'une pourra contenir des variables. Ensuite, en utilisant votre filtre, vous implémenterez un moteur d'inférence avec variables.

Vous aurez également la possibilité d'implémenter un unificateur et de le tester sur votre moteur à chaînage avant avec variables. Un unificateur permet aussi de comparer deux propositions. La différence fondamentale avec le filtre est que l'unificateur accepte la présence de variables dans les deux expressions, ce qui rend possible de l'utiliser dans le chaînage arrière.

Modules squelettes

Les modules qui suivent constituent le squelette du programme que nous allons développer. Le dernier, `exemple_impots_avec_variables.py`, est un module de test.

Module `.../moteur_avec_variables/proposition_avec_variables.py` :

```
def est_atomique(proposition):
    print('à compléter')

def est_une_variable(proposition, marqueur='?'):
    print('à compléter')
```

```
def tete(proposition):
    if est_atomique(proposition):
        raise Exception("Proposition atomique: Impossible de la segmenter.")
    elif len(proposition) > 0:
        return proposition[0]
    else:
        raise Exception("Proposition vide: Impossible de la segmenter.")

def corps(proposition):
    if est_atomique(proposition):
        raise Exception("Proposition atomique: Impossible de la segmenter.")
    elif len(proposition) > 0:
        return proposition[1:]
    else:
        raise Exception("Proposition vide: Impossible de la segmenter.")

def lister_variables (proposition):
    variables = set()
    if est_atomique(proposition):
        if est_une_variable(proposition):
            variables.add(proposition)
    else:
        for sous_prop in proposition:
            variables.update( lister_variables (sous_prop))
    return variables
```

Module `.../moteur_avec_variables/regle_avec_variables.py` :

```
class RegleAvecVariables:
    def __init__ ( self , conditions , conclusion ):
        self.conditions = conditions
        self.conclusion = conclusion

    def depend_de(self, fait , methode):
        print('à compléter')

    def satisfaite_par ( self , faits , cond, env, methode):
        print('à compléter')

    def __repr__ ( self ):
        return '{} => {}'.format(str(self.conditions), str( self.conclusion))
```

Module `.../moteur_avec_variables/chainage_avant_avec_variables.py` :

```
from moteur_sans_variables.chainage import Chainage
from .filtre import Filtre

class ChainageAvantAvecVariables(Chainage):
    def __init__ ( self , connaissances, methode=None):
        Chainage.__init__ ( self , connaissances)

        if methode is None:
            self.methode = Filtre()
        else:
            self.methode = methode
```

```
    def instancie_conclusion(self, regle, envs):
        print('à compléter')

    def chaine(self):
        print('à compléter')
```

Module `.../moteur_avec_variables/filtre.py` :

```
from .proposition_avec_variables import est_atomique, est_une_variable, tete, corps

class Filtre:
    echec = 'échec'

    def substitue(self, pattern, env):
        print('à compléter')

    def filtre (self, datum, pattern):
        print('à compléter')

    def pattern_match(self, datum, pattern, env=None):
        print('à compléter')
```

Module `.../moteur_avec_variables/unificateur.py` :

```
from .proposition_avec_variables import est_atomique, est_une_variable, tete, corps

class Unificateur:
    echec = 'échec'

    def substitue(self, pattern, env):
        print('à compléter')

    def unifie(self, prop1, prop2):
        print('à compléter')

    def pattern_match(self, prop1, prop2, env=None):
        print('à compléter')
```

Module `.../exemple_impots_avec_variables.py` :

```
from sys import argv, exit
from moteur_avec_variables.regle_avec_variables import RegleAvecVariables
from moteur_sans_variables.connaissance import BaseConnaissances
from moteur_avec_variables.filtre import Filtre
from moteur_avec_variables.unificateur import Unificateur
from moteur_avec_variables.chainage_avant_avec_variables import
    ChainageAvantAvecVariables

faits_initiaux = [
    ('add', '0', '0', '0', '0'),
    ('add', '100', '100', '0', '0'),
    ('add', '100', '0', '100', '0'),
    ('add', '200', '100', '100', '0'),
    ('add', '200', '0', '200', '0'),
    ('add', '300', '100', '200', '0'),
    ('add', '50', '0', '0', '50'),
    ('add', '150', '100', '0', '50'),
```

```
    ('add', '150', '0', '100', '50'),
    ('add', '250', '100', '100', '50'),
    ('add', '250', '0', '200', '50'),
    ('add', '350', '100', '200', '50'),
    ('add', '100', '0', '0', '100'),
    ('add', '200', '100', '0', '100'),
    ('add', '200', '0', '100', '100'),
    ('add', '300', '100', '100', '100'),
    ('add', '300', '0', '200', '100'),
    ('add', '400', '100', '200', '100'),
    # Paul
    ('bas-salaire', 'Paul'),
    ('loyer', 'Paul'),
    ('enfants', 'Paul'),
    ('long-trajet', 'Paul'),
    # Marc
    ('moyen-salaire', 'Marc'),
    ('loyer', 'Marc'),
    ('enfants', 'Marc'),
    ('long-trajet', 'Marc'),
    # Jean
    ('haut-salaire', 'Jean'),
    ('pas-de-loyer', 'Jean'),
    ('pas-d-enfants', 'Jean'),
    ('long-trajet', 'Jean'),
]

regles = [
    # Reduction enfants
    [[('pas-d-enfants', '?x')], ('réduc-enfant', '0', '?x')],
    [[('enfants', '?x')], ('réduc-enfant', '100', '?x')],
    # Reduction loyer
    [[('bas-salaire', '?x'), ('loyer', '?x')], ('réduc-loyer', '200', '?x')],
    [[('moyen-salaire', '?x'), ('loyer', '?x')], ('réduc-loyer', '100', '?x')],
    [[('haut-salaire', '?x'), ('loyer', '?x')], ('réduc-loyer', '0', '?x')],
    [[('pas-de-loyer', '?x')], ('réduc-loyer', '0', '?x')],
    # Reduction transport
    [[('petit-trajet', '?x')], ('réduc-trajet', '0', '?x')],
    [[('réduc-enfant', '0', '?x'), ('long-trajet', '?x')],
     ('réduc-trajet', '100', '?x')],
    [[('réduc-loyer', '0', '?x'), ('long-trajet', '?x')],
     ('réduc-trajet', '100', '?x')],
    [[('réduc-enfant', '100', '?x'), ('réduc-loyer', '100', '?x'),
      ('long-trajet', '?x')], ('réduc-trajet', '50', '?x')],
    [[('réduc-enfant', '100', '?x'), ('réduc-loyer', '200', '?x'),
      ('long-trajet', '?x')], ('réduc-trajet', '0', '?x')],
    # Reduction totale
    [[('réduc-enfant', '?a', '?x'), ('réduc-loyer', '?b', '?x'),
      ('réduc-trajet', '?c', '?x'), ('add', '?res', '?a', '?b', '?c')],
     ('réduc', '?res', '?x')],
]

if len(argv) < 2 or argv[1].lower() not in ('filtre', 'unificateur'):
    print('On attend un argument: Filtre ou Unificateur')
    exit(1)

if argv[1].lower() == 'filtre':
    methode = Filtre()
elif argv[1].lower() == 'unificateur':
```

```
    methode = Unificateur()

bc = BaseConnaissances(lambda descr: RegleAvecVariables(descr[0], descr[1]))
bc. ajoute_faits ( faits_initiaux )
bc. ajoute_regles ( regles )

moteur = ChainageAvantAvecVariables(connaissances=bc, methode=methode)
moteur.chaine()

moteur. affiche_solutions ()

if len(argv) > 2 and argv[2].lower() == 'trace':
    moteur. affiche_trace ()
```

Le code de cette série d'exercices s'appuie sur le code développé pour l'inférence sans variables. Il est donc important de respecter strictement la structure des dossiers que nous vous fournissons. Sinon, Python ne pourra pas importer correctement les modules.

Les faits et les règles

Au cours des exercices précédents, vous avez manipulé des faits simples et des règles sans variables. Dans cette série, les faits pourront être composés et les règles pourront contenir des variables. Nous parlerons plus généralement de *propositions* qui sont définies récursivement comme étant :

- un atome, présenté sous la forme d'une `string` et représentant soit une variable, soit une valeur ;
- ou un `tuple` contenant des propositions.

Vous trouverez ici[(1)] des informations détaillées sur la syntaxe des `tuples` en Python. L'essentiel à retenir pour cet exercice est qu'un `tuple` est une séquence de valeurs construite en alignant plusieurs éléments séparés par des virgules : `t3 = 'str', 0, []`. Pour plus de clarté, on entoure généralement ces valeurs de parenthèses : `t3 = ('str', 0, [])`. Notez enfin qu'un tuple composé d'un seul élément doit contenir une virgule finale. Ici aussi, il est préférable d'utiliser des parenthèses : `t1 = ('unique',)`.

Par convention, une variable sera un atome qui commence par un point d'interrogation. Exemple :

```
'?x'
'?qui'
```

Les faits seront des propositions sans variables. Voici par exemple deux descriptions de faits :

```
'Paul'
('réduc−loyer', '200', 'Michel')
```

(1) https ://docs.python.org/3.5/tutorial/datastructures.html#tuples-and-sequences

En général cependant, une proposition pourra contenir des variables, comme dans ces exemples :

```
('réduc−loyer', '200', '?x')
('bas−salaire', '?z')
('réduc−trajet', '?x', '?y')
```

Comme dans la série précédente, les règles seront constituées d'une liste de conditions et d'une conclusion. Voici les descriptions de deux règles possibles :

```
[('pas−d−enfants', '?x')], ('réduc−enfant', '0','?x')
[[('bas−salaire', '?x'), ('loyer', '?x')], ('réduc−loyer', '200', '?x')
```

La classe `RegleSansVariables` de la série précédente doit ainsi être adaptée à l'utilisation de variables. Nous créerons donc une nouvelle classe `RegleAvecVariables` qui devra redéfinir les méthodes `depend_de` et `satisfaite_par` en faisant appel aux méthodes de pattern matching (filtrage ou unification). Vous devrez compléter ces méthodes une fois le pattern matching implémenté.

Exercice 3.2.1 Le filtre

La technique du filtrage permet d'établir des correspondances entre deux propositions. Plus précisément, un filtre détermine les substitutions variables-valeurs qui permettent de retrouver une proposition sans variables (le datum) à partir d'une autre (le pattern) qui peut contenir des variables.

Pour commencer, vous devrez coder deux fonctions utilitaires dans le module `proposition_avec_variables.py`, afin de faciliter l'implémentation du filtrage : La fonction `est_atomique(prop)`, qui doit retourner `True` si la proposition `prop` est une `string` et la fonction `est_une_variable(prop)`, qui doit retourner `True` si `prop` est un atome et si le premier caractère de sa description indique une variable (`'?'`).

La méthode `Filtre.substitue`

Vous pouvez maintenant vous attaquer à la classe `Filtre` du module `filtre.py` et implémenter sa méthode `substitue`. Cette méthode doit retourner un pattern dont les variables auront été remplacées par les valeurs disponibles dans l'environnement `env` qui est passé en paramètre. `env` est un dictionnaire qui contient des substitutions variable-valeur. N'oubliez pas que le pattern est une proposition, donc soit un atome, soit un tuple pouvant contenir d'autres propositions. Pensez donc à utiliser une méthode récursive pour traiter ces cas.

Pour vous guider, voici quelques exemples du fonctionnement de la méthode :

```
substitue('doctorant', {'?z': 'Paolo', '?y': 'Michel', '?x': 'Vincent'})
    −> 'doctorant'

substitue('?x', {'?z': 'Paolo', '?y': 'Michel', '?x': 'Vincent'})
    −> 'Vincent'

substitue(('?x', 'est un', 'doctorant'), {})
    −> ('?x', 'est un', 'doctorant')
```

```
substitue(('?x', 'est un', 'doctorant'), {'?x': 'Vincent'})
    -> ('Vincent', 'est un', 'doctorant')

substitue(('?x', 'est un', 'doctorant'), {'?y':'Michel'})
    -> ('?x', 'est un', 'doctorant')

substitue(('?x', 'est un', ('?a')), {'?y': 'Michel', '?a': 'Vincent'})
    -> ('?x', 'est un', ('Vincent'))
```

La méthode `Filtre.filtre`

Nous allons maintenant écrire la méthode `filtre` de la classe `Filtre`, qui implémente l'algorithme de filtrage. La méthode retournera :

- Un environnement, `{'?x' :'toto', ..., '?y' :'titi'}`, dans le cas où le processus aboutit à des substitutions ;
- Un environnement vide, `{}`, si le processus réussit sans aucune substitution, c'est-à-dire lorsque les deux propositions sont identiques ;
- La constante `Filtre.echec`, en cas d'erreur de filtrage (si `datum` et `pattern` sont incompatibles).

La méthode possède la signature `filtre(datum, pattern)`, où `datum` est une proposition sans variables, et `pattern` une proposition pouvant contenir des variables.

Voici son pseudo-code :

```
Filtre (datum, pattern)
1.  IF pattern == () AND datum == () THEN RETURN {}
2.  ELSE IF pattern == () OR datum == () THEN RETURN échec
3.  ELSE IF pattern est un atome THEN
4.      IF pattern et datum sont identiques THEN RETURN {}
5.      ELSE IF pattern est une variable THEN RETURN {pattern: datum}
6.      ELSE RETURN échec
7.      END IF
8.  ELSE IF datum est un atome THEN RETURN échec
9.  ELSE
10.     F1 <- premier (datum)
11.     T1 <- reste (datum)
12.     F2 <- premier(pattern)
13.     T2 <- reste(pattern)
14.     Z1 <- Filtre(F1, F2)
15.     IF Z1 == échec THEN RETURN échec END IF
16.     G1 <- T1
17.     G2 <- remplacer les variables de T2 selon les substitutions de Z1
18.     Z2 <- Filtre(G1, G2)
19.     IF Z2 == échec THEN RETURN échec END IF
20.     RETURN {Z1 UNION Z2}
21. END IF
END Filtre
```

Et voici quelques exemples de son usage :

```
filtre ('Vincent', '?x')
    -> {'?x': 'Vincent'}

filtre (('Vincent', 'est un', 'doctorant'), ('?x', 'est un', 'doctorant'))
```

```
    -> {'?x': 'Vincent'}

filtre (('Vincent', 'est un', 'doctorant') , ('Vincent', 'est un', 'doctorant'))
    -> {}

filtre (('Vincent', 'est un', 'doctorant'), ('?x', 'est un', '?x'))
    -> échec

filtre (('Vincent', 'est un', ('doctorant')) , ('Vincent', 'est un', ('?y')))
    -> {'?y': 'doctorant'}
```

Vous trouverez ici[2] quelques fonctions utiles pour manipuler des dictionnaires. La méthode `update` est particulièrement commode pour obtenir l'union de deux dictionnaires. Une façon brève et élégante pour retourner un dictionnaire à un élément est : `return {key : value}`.

La vraie fonction d'interface : `Filtre.pattern_match`

La fonction `filtre` n'est pas très pratique pour un programme hôte car il n'est pas possible de lui fournir en entrée un environnement. Dans le processus de chaînage avant que vous allez écrire, chaque condition d'une règle doit être vérifiée avant de pouvoir être utilisée. Cela implique que chaque condition soit filtrée avec succès par un fait existant. Comme plusieurs conditions peuvent être présentes, il est nécessaire de tester chacune en respectant l'environnement obtenu lors des filtrages précédents. Il faut donc pouvoir fournir à la fonction de filtrage un environnement déjà établi.

Vous devez ainsi compléter la méthode `pattern_match` de la classe `Filtre`, qui permettra de prendre en compte un environnement de substitutions déjà existantes. Cette méthode prend en paramètres deux propositions, un datum et un pattern, accompagnés d'un environnement sous forme d'argument optionnel, et retourne un nouvel environnement. Elle s'appuiera bien évidemment sur les méthodes `filtre` et `substitue`.

Voici une liste d'exemples qui prennent en compte des environnements préexistants :

```
pattern_match(('Vincent', 'est un', 'doctorant'), ('?x', 'est un', 'doctorant') ,
  {'?y':'doctorant'})
    -> {'?y': 'doctorant', '?x': 'Vincent'}

pattern_match(('Vincent', 'est un', 'doctorant'), ('?x', 'est un', '?y'),
  {'?y':'doctorant'})
    -> {'?y': 'doctorant', '?x': 'Vincent'}

pattern_match('Vincent', '?x', {})
    -> {'?x': 'Vincent'}
```

Les arguments optionnels d'une fonction Python obéissent à la syntaxe `argument=valeur`. À cause de problèmes de mutabilité, il est déconseillé d'utiliser

[2] `http://docs.python.org/py3k/library/stdtypes.html\#typesmapping`

`{}` comme valeur par défaut. Il est préférable d'utiliser la valeur `None` et d'assigner un dictionnaire vide à la variable à l'intérieur de la méthode. Vous pouvez consulter cet article[3] pour plus d'informations.

Exercice 3.2.2 De retour aux règles

La méthode `RegleAvecVariables.depend_de`

La classe `RegleAvecVariables`, qui est à implémenter dans le module `regle_avec_variables.py`, reprend les noms des méthodes de la classe `RegleSansVariables` que nous avons utilisée dans l'exercice précédent, mais avec une implémentation passablement différente. La méthode `dépend_de` doit ainsi vérifier qu'un fait passé en paramètre est un déclencheur des conditions de la règle, et doit retourner un dictionnaire associant à chaque condition de la règle l'environnement résultant du pattern match entre cette condition et le fait. Si la recherche de substitutions aboutit à un échec pour une condition, il n'est pas nécessaire de la mentionner dans le dictionnaire. Il faudra donc comparer chaque condition de la règle avec le fait à l'aide de la méthode `pattern_match` du filtre et recueillir les environnements résultants.

La méthode `dépend_de` prend en entrée deux paramètres : `fait`, un fait à tester, et `methode`, l'objet de pattern matching utilisé, soit un filtre soit un unificateur (filtre par défaut). N'oubliez pas que cette fonction doit vérifier toutes les conditions de la règle. Pour vous aider, voici quelques exemples :

```
règle = RegleAvecVariables([('père', '?x', '?y'), ('père', '?y', '?z')],
                           ('grand-père', '?x', '?z'))

# Le fait ('père', 'Jean', 'Paul') peut satisfaire la première ou la seconde condition :
methode.pattern_match(('père', 'Jean', 'Paul'), ('père', '?x', '?y'))
    -> {'?x': 'Jean', '?y': 'Paul'}

methode.pattern_match(('père', 'Jean', 'Paul'), ('père', '?y', '?z'))
    -> {'?y': 'Jean', '?z': 'Paul'}

# La méthode depend_de renvoie donc un dictionnaire avec ces deux conditions associées
# à leurs environnements respectives :
depend_de(('père', 'Jean', 'Paul'), Filtre())
    -> {('père', '?x', '?y'): {'?x': 'Jean', '?y': 'Paul'},
        ('père', '?y', '?z'): {'?y': 'Jean', '?z': 'Paul'}}
```

La méthode `RegleAvecVariables.satisfaite_par`

La méthode `satisfaite_par` de la classe `RegleAvecVariables` vérifie que les faits passés en paramètres satisfont toutes les conditions de la règle tout en respectant l'environnement de départ inféré par `dépend_de`. La méthode prend en entrée quatre paramètres :

- `faits` : une liste de faits ;
- `cond` : la condition qui a servi à découvrir l'environnement `env` ;

[3] http ://www.deadlybloodyserious.com/2008/05/default-argument-blunders/

- `env` : l'environnement déjà établi par `depend_de` ;
- `methode` : l'objet de pattern matching utilisé, soit un filtre soit un unificateur (filtre par défaut).

La méthode `satisfaite_par` retourne une liste d'environnements qui correspondent à toutes les substitutions possibles entre les conditions de la règle et les faits. Il s'agit donc de trouver les environnements qui satisfont chacun toutes les conditions. Si ce n'est pas possible, la méthode doit retourner une liste vide. En outre, chaque nouvel environnement construit devra être testé lors de la vérification de la prochaine condition. Voici un petit exemple pour clarifier les choses :

```
faits  = [('père', 'Jean', 'Paul'),
          ('père', 'Florent', 'Paul'),
          ('père', 'Paul', 'Michel')]

regle  = RegleAvecVariables([('père', '?x', '?y'), ('père', '?y', '?z')],
                            ('grand-père, '?x', '?z'))

satisfaite_par (faits, ('père', '?x', '?y'), {'?x': 'Jean', '?y': 'Paul'})
    -> [{'?x': 'Jean', '?y': 'Paul', '?z': 'Michel'}]

satisfaite_par (faits, ('père', '?y', '?z'), {'?y': 'Jean', '?z': 'Paul'})
    -> []
```

Et voici le pseudo-code de la méthode :

```
SatisfaitePar (regle, faits, cond, env)
1.  conditions_a_tester <- conditions de regle sauf cond
2.  environnements_satisfaisants <- [env]
3.  FOR EACH condition cond1 de conditions_a_tester DO
4.      environnements_nouveaux <- liste vide
5.      FOR EACH fait de faits DO
6.          FOR EACH environnement env1 de environnements_satisfaisants DO
7.              env1 <- nouvel environnement déterminé par pattern_match(fait, cond1, env1)
8.              IF NOT env1 == échec THEN
9.                  ajouter env1 à environnements_nouveaux
10.             END IF
11.         END FOR
12.     END FOR
13.     IF environnements_nouveaux est vide THEN
14.         RETURN liste vide
15.     END IF
16.     environnements_satisfaisants <- environnements_nouveaux
17. END FOR
18. RETURN environnements_satisfaisants
END SatisfaitePar
```

Exercice 3.2.3 Le moteur d'inférence à chaînage avant avec variables

La méthode `ChainageAvantAvecVariables.instancie_conclusion`

Une fois que les conditions d'une règle ont été validées, il faut instancier la conclusion en accord avec la liste des environnements ainsi obtenus afin de déduire de nouvelles propositions. La fonction `instancie_conclusion` de la classe

`ChainageAvantAvecVariables` prend comme paramètres une règle et une liste d'environnements, et retourne une liste de nouveaux faits (un par environnement). Exemples :

```
règle = RegleAvecVariables(liste_de_conditions, ('?x', '?y'))
instancie_conclusion(règle, [{'?x': 'X','?y': 'Y'}])
    -> [('X', 'Y')]
```

Vous pouvez implémenter cette méthode de façon simple avec une boucle `for` itérant sur les environnements, mais il est aussi possible de l'écrire en une ligne sous forme de 'list comprehension'[4] (un peu plus complexe mais plus élégant). Pensez en outre à utiliser la fonction `substitue` de la classe de pattern match.

La méthode `ChainageAvantAvecVariables.chaine`

Grâce aux changements apportés aux règles, grâce à la méthode `instancie_conclusion` et au module de filtrage que nous avons développé, nous pouvons maintenant réaliser une nouvelle version de notre moteur d'inférence à chaînage avant, avec la capacité de manipuler des règles comportant des variables.

Implémentez l'algorithme de chaînage avant dans la méthode `chaine` de la classe `ChainageAvantAvecVariables`. Pour rappel, l'algorithme à implémenter est le suivant :

```
ChainageAvantAvecVariables(faits_depart, regles)
1.  solutions <- liste vide
2.  Q <- faits_depart
3.  WHILE Q n'est pas vide DO
4.      q <- premier(Q)
5.      Q <- reste(Q)
6.      IF q n'est pas dans solutions THEN
7.          ajouter q à solutions
8.          FOR EACH règle r de regles DO
9.              IF une condition de r dépend de q THEN
10.                 FOR EACH condition cond et environnement env déduits de
                    r.depend_de(q) DO
11.                     envs <- r.satisfaite_par(solutions, cond, env)
12.                     instances <- instancier la conclusion de r selon envs
13.                     ajouter instances en queue de Q
14.                 END FOR
15.             END IF
16.         END FOR
17.     END IF
18. END WHILE
19. RETURN solutions
END ChainageAvantAvecVariables
```

Test du programme : Chaînage avant avec filtre

Le module `exemple_impots_avec_variables.py` contient les règles et les faits nécessaires pour le calcul du montant d'une réduction d'impôts. Après avoir

[4] Introduction à la list comprehension de Python : `https://docs.python.org/3/tutorial/datastructures.html\#list-comprehensions`.

écrit votre programme, testez-le en exécutant dans le terminal avec l'option `filtre`. Vous pouvez afficher la trace en ajoutant l'option `trace`. Quelle devrait être la réduction d'impôts ?

```
python3 exemple_impots_avec_variables.py filtre
python3 exemple_impots_avec_variables.py filtre trace
```

Exercice 3.2.4 L'unificateur

Nous allons maintenant construire un unificateur. C'est un outil analogue à un filtre mais bien plus puissant. À la différence du filtre, qui compare deux expressions dont l'une seulement peut comporter des variables, l'objectif d'un unificateur est de comparer deux expressions pouvant toutes deux contenir des variables. L'unificateur est donc une version généralisée du filtre. Il fournit comme résultat les correspondances entre les deux propositions sous la forme de substitutions variable-proposition (lorsqu'il en existe).

Pour mieux comprendre l'utilité de l'unificateur, voyons quelques exemples :

```
unifie (('Vincent', 'est un', 'doctorant'), ('Vincent', 'est un', 'doctorant'))
  -> {}

unifie (('Vincent', 'est un', 'doctorant'), ('Michel', 'est un', 'doctorant'))
  -> échec

unifie (('foo', '?x', ('?y', 'bar', 'Jean')), ('foo', 'Jean', ('Marc', 'bar', '?x')))
  -> {'?y': 'Marc', '?x': 'Jean'}

unifie (('p', '?x', ('f', '?y')), ('p', ('f','a'), '?x'))
  -> {'?y': 'a', '?x': ('f', 'a')}`
```

Nous utiliserons des conventions analogues à celles que nous avons appliquées dans le cas du filtrage. Désormais cependant un environnement pourra associer des **propositions, pas nécessairement des valeurs,** à des variables. Le résultat final de l'unification contiendra plutôt des substitutions variable-proposition sans variables, mais les étapes intermédiaires pourront aussi renvoyer des environnements qui associent des propositions contenant des variables à d'autres variables.

La méthode `Unificateur.substitue`

La fonction `substitue` de la classe `Unificateur` doit permettre d'instancier une proposition étant donné un ensemble de substitutions variable-proposition. Exemples :

```
substitue(('?x', 'est un', 'doctorant') , {})
  -> ('?x', 'est un', 'doctorant')

substitue(('p', '?x') , {'?y': ('g','?z'), '?x': ('f', '?y'), '?z': ('a')})
  -> ('p', ('f', ('g', ('a'))))

substitue(('p', '?x'), {'?y': ('g','?z'), '?x': ('f','?y'), '?z': ('?q')})
  -> ('p', ('f', ('g', ('?q'))))
```

Lorsque vous rédigerez le code de cette méthodes, rappelez-vous qu'une variable doit parfois être remplacée par une définition de substitution qui contient

elle-même des variables. Il faut donc veiller à aussi remplacer toutes les variables qui figurent dans la proposition associée à une autre variable. Pensez à implémenter une solution récursive.

La méthode `Unificateur.unifie`

Vous avez maintenant tout ce qu'il faut pour implémenter la fonction `Unificateur.unifie`. L'algorithme d'unification est très proche de l'algorithme de filtrage :

```
Unifie(prop1, prop2)
1.  IF prop1 ou prop2 est un atome THEN
2.      si nécessaire, échanger prop1 et prop2 pour que prop1 soit un atome
3.      IF prop1 et prop2 sont identiques THEN RETURN {}
4.      ELSE IF prop1 est une variable THEN
5.          IF prop1 apparaît dans prop2 THEN RETURN échec
6.          ELSE RETURN {prop1: prop2}
7.          END IF
8.      ELSE IF prop2 est une variable THEN RETURN {prop2: prop1}
9.      ELSE RETURN échec
10.     END IF
11. ELSE
12.     F1 <- premier(prop1)
13.     T1 <- reste(prop1)
14.     F2 <- premier(prop2)
15.     T2 <- reste(prop2)
16.     Z1 <- Unifie(F1, F2)
17.     IF Z1 == échec THEN RETURN échec END IF
18.     G1 <- remplacer les variables de T1 selon les substitutions de Z1
19.     G2 <- remplacer les variables de T2 selon les substitutions de Z1
20.     Z2 <- Unifie(G1, G2)
21.     IF Z2 == échec THEN RETURN échec END IF
22.     RETURN {Z1 UNION Z2}
23. END IF
END Unifie
```

La vraie fonction d'interface : `Unificateur.pattern_match`

Comme dans le cas de la méthode `filtre`, `unifie` n'est pas très pratique pour un programme hôte. Il faut donc aussi coder une fonction `Unificateur.pattern_match` qui permette de prendre en compte un environnement de substitutions déjà existantes. Cette fonction prendra en paramètres deux expressions pouvant contenir des variables et un environnement à titre d'argument optionnel. Elle doit retourner un nouvel environnement ou la constante `Unificateur.echec`, selon que le pattern matching a réussi ou échoué. La méthode s'appuiera bien évidemment sur `unifie` et `substitue`.

Voici une liste d'exemples qui prennent en compte des environnements préexistants :

```
pattern_match(('foo', '?x', ('?y', 'bar', 'Jean')), ('foo', 'Jean', ('Marc',
    'bar', '?x')), {'?y': 'Marc'})
  -> {'?y': 'Marc', '?x': 'Jean'}

pattern_match(('foo', '?x', ('?y', 'bar', 'Paul')), ('foo', 'Jean', ('Marc',
```

```
 'bar', '?x')), {})
−> échec
```

Pour coder cette méthode, il convient donc de s'assurer que l'environnement est valide, puis de remplacer les variables des deux propositions par les définitions de l'environnement, et enfin de procéder à l'unification.

Test du programme : Chaînage avant avec unificateur

Essayez d'utiliser l'unificateur à la place du filtre, en lançant `exemple_impots_avec_variables.py` avec l'option 'unificateur'. Que constatez-vous et pourquoi ?

```
python3 exemple_impots_avec_variables.py unificateur
python3 exemple_impots_avec_variables.py unificateur trace
```

Est-il vraiment nécessaire d'utiliser un unificateur dans le chaînage avant ?

Solutions à la page 343

Chapitre 4

Représentation structurée des connaissances

Au-delà de la logique des prédicats, on peut imaginer une structuration des connaissances qui permet de plus facilement gérer des bases de connaissances. Des objets du monde comme *chien*, *livre*, *personne*, *montagne* ou *couleur* sont des instances de *concepts*. Étant donné les objectifs fondamentaux de l'intelligence artificielle, il va de soi que cette notion peut être représentée au sein d'un programme. Savoir ce qui définit exactement un concept est une question philosophique. Ce problème est d'ailleurs troublé par l'existence d'exceptions : par exemple, un oiseau est en général capable de voler, mais cette propriété ne s'applique pas aux autruches. Dans le cadre de l'intelligence artificielle, on s'intéresse toujours à des applications spécifiques, ce qui permet une définition pragmatique des concepts (qui ne satisfait évidemment pas nécessairement les philosophes). En général, un concept est défini comme étant une combinaison de *propriétés* de définition : les oiseaux ont des ailes, ils volent, ils chantent, etc. Une combinaison de propriétés définit une *classe* d'individus et si l'on ajoute davantage de propriétés de définition, il est aussi possible de définir des sous-classes : la classe des oiseaux est ainsi une sous-classe des animaux qui se distingue par des propriétés telles que `peut-voler` ou `a-des-ailes`.

Dans le cadre du calcul des prédicats, il n'existe qu'un seule type d'objets : les *entités*. Comme la plupart des règles ne s'appliquent qu'à des concepts particuliers, le champ d'une entité doit être restreint en précisant les propriétés définissant le concept. C'est en fait un moyen extrêmement lourd de définir des connaissances, principalement parce que qu'on ne tire pas parti des relations existant entre concepts : une grande partie des propriétés d'une entité peuvent être *héritées* des concepts englobant dont elle est l'instance. Des techniques particulières de structuration de la connaissance, représentées par les *réseaux à héritage structuré*, ont été développées en vue de profiter de la structuration conceptuelle des classes.

Les entités qui se conforment à la définition d'une classe sont nommées ses *instances*. Si dans une représentation des connaissances toutes les entités sont structurées en classes, on peut alors les distinguer en représentant uniquement les propriétés spécifiques les différenciant des autres membres de la même classe. Par exemple, une instance du concept `table` peut désormais être identifiée par un ensemble limité de propriétés : sa hauteur, sa taille, le matériau qui la compose et sa position dans l'espace. Pour chaque concept, on peut ainsi définir

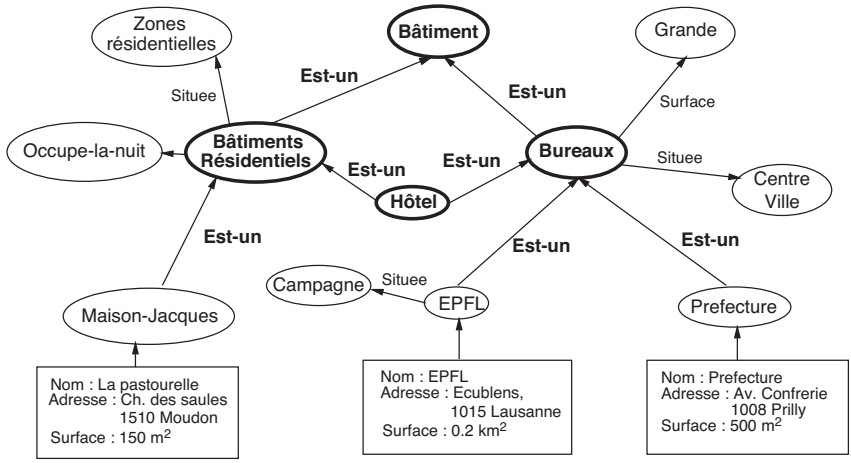


Fig. 4.1 *Structuration utilisant des réseaux à héritage et cadres. Les propriétés communes sont représentées par le réseau, les instances distinguées par les cadres.*

un ensemble de propriétés distinctives qui définissent une instance du concept. C'est l'idée des *cadres* (*frames*) : pour chaque classe, on regroupe l'ensemble des propriétés distinctives, appelées des *slots*, sous forme d'un *cadre*.

La structuration des connaissances qui résulte de l'utilisation parallèle de réseaux à héritage et de cadres est décrite dans la figure 4.1. Elle permet de définir les concepts avec un maximum d'économie. Cette structuration a notamment inspiré la programmation orientée objet, pour laquelle la plupart des langages intègrent une méthodologie analogue.

La notion des réseaux à héritage structurés peut être généralisée en permettant non seulement des liens entre sous-classes, mais également d'autres relations entre concepts. De tels types de réseaux sont connus sous le nom de *réseaux sémantiques*.

4.1 Cadres

Un *cadre* est une structure de données qui regroupe un ensemble de propriétés et correspond ainsi à un *enregistrement*, notion connue des langages de programmation classiques. Un cadre décrit un objet, qui peut être :

- une classe, dont les propriétés *communes* à toutes les instances sont regroupées dans le cadre ;
- une instance d'une classe, dont les propriétés qui la *distinguent* des autres instances sont représentées par le cadre.

La figure 4.2 donne un exemple de cette notion pour la représentation d'un étudiant.

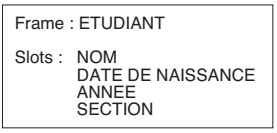


Fig. 4.2 *Exemple d'un cadre. À chaque slot doit être attribuée une propriété, qui peut être une référence à un autre cadre. Des valeurs par défaut peuvent remplir les propriétés indéfinies.*

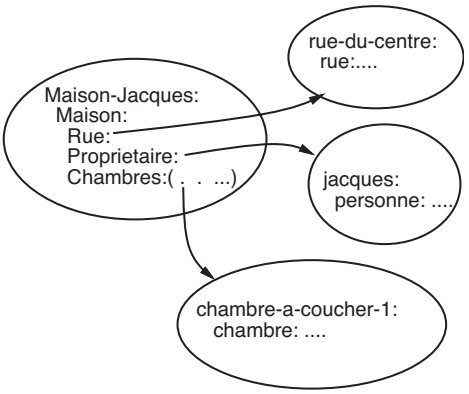


Fig. 4.3 *Les slots qui contiennent des pointeurs à d'autres cadres créent une structure de mémoire associative.*

Si un cadre ressemble beaucoup à un enregistrement, il existe néanmoins des différences importantes. Les propriétés que regroupe un cadre, appelées des *slots*, peuvent en effet contenir des valeurs qui sont d'autres cadres, créant ainsi une structure de mémoire associative. La figure 4.3 montre un exemple d'une telle structure.

Cette structure associative permet de reproduire dans un programme la nature associative de la mémoire humaine : le cadre qui représente la maison de Jacques peut susciter les cadres qui représentent la personne ou des détails de cette maison. Une telle structuration permet de simplifier considérablement des bases de connaissances car les objets importants peuvent être rapidement retrouvés par les liens associatifs.

La notion de cadre a été développée dans le contexte de systèmes de vision artificielle afin de mieux permettre la reconnaissance d'objets. La reconnaissance proprement dite d'un objet se base alors sur la vérification des propriétés de la classe à laquelle appartient le cadre. Une fois reconnue, l'information que le système doit obtenir sur l'objet est définie par les slots du cadre correspondant. Les cadres définissent alors quelle est l'information importante que le système doit avoir concernant un objet, un problème qui est difficile à résoudre autrement. Les cadres définissent un contexte de la structure de mémoire dont le choix d'objets est limité : la tâche de reconnaissance est alors considérablement simplifiée.

4.2 Réseaux à héritage structurés

Pour de nombreux domaines, les relations de définition liant les éléments de la représentation des connaissances sont en quantité non négligeable. L'exemple le plus important est celui des *classifications hiérarchiques* : un moineau est un type d'oiseau, un oiseau est un type d'animal, etc. Chaque membre d'une certaine classe partage des propriétés avec les autres membres de cette classe : par exemple, les oiseaux volent et les animaux ont besoin de nourriture. Au lieu d'écrire des ensembles de règles exprimant cette connaissance, il est alors préférable d'utiliser un mode structuré de représentation permettant une expression directe de ce type de connaissances. L'exemple de la figure 4.1 illustre la notion de *réseau à héritage structuré*.

La plupart des représentations structurées de la connaissance utilisent la notion de graphes ou *réseaux sémantiques*. L'idée sur laquelle se basent les réseaux sémantiques puise ses sources de certaines théories psychologiques de la mémoire humaine. Les nœuds correspondent aux *concepts*, et les arcs aux relations entre concepts. Un concept est une classe d'entités partageant certaines propriétés, comme les animaux, les oiseaux ou les pingouins. Selon des théories psychologiques, la mémoire humaine s'articule et se structure autour de concepts.

Un concept est défini par l'ensemble des prédicats définissant les propriétés communes à toutes ses entités. Comme le montre la figure 4.1, des propriétés sont attachées aux nœuds du réseau. Les concepts sont *hiérarchisés* afin d'exprimer l'héritage des propriétés. Le réseau de la figure 4.1 donne un exemple de telles hiérarchies. Les concepts sont par ailleurs liés à des sous-concepts via des liens `EST-UN`, et c'est grâce à ces liens qu'ils héritent des propriétés englobantes. Les réseaux sémantiques qui ne contiennent que des liens `EST-UN` sont désignés sous le nom de *réseau à héritage structuré* et sont de loin les plus fréquemment utilisés.

Dans les classifications hiérarchiques réelles, il existe de nombreuses exceptions aux propriétés générales. Ces exceptions nécessitent un mécanisme particulier afin de construire efficacement les hiérarchies : il s'agit de considérer que les propriétés définies sur les niveaux les plus bas ont la précédence sur celles spécifiées sur les niveaux les plus hauts. Par exemple, la propriété d'être située en centre ville, généralement propre à la classe des bureaux, ne s'étend pas à l'Ecole polytechnique fédérale de Lausanne (EPFL) qui fait pourtant bien partie de cette classe. Ceci s'exprime dans le réseau de la figure 4.1 par l'adjonction au nœud représentant l'EPFL d'une propriété explicite spécifiant qu'elle se situe en banlieue.

Une autre complication est donnée par le fait que de nombreux concepts partagent des liens d'héritage avec *plusieurs* classes. Par exemple, un hôtel partage à la fois des caractéristiques d'un bâtiment résidentiel – il est occupé la nuit – et des caractéristiques d'un bâtiment commercial : il est normalement situé en centre ville. Une telle situation exige l'utilisation de l'*héritage multiple* de plusieurs classes.

Les règles d'héritage qui tiennent compte des exceptions et de l'héritage multiple sont les suivantes :

- Les propriétés d'une instance x de la classe $\mathcal{C}$ sont données par l'union des propriétés du cadre et les propriétés de la classe $\mathcal{C}$.
- Les propriétés d'une classe $\mathcal{C}$ sont données par les propriétés de $\mathcal{C}$ et de toutes les classes avec lesquelles $\mathcal{C}$ partage un lien `EST-UN`. Des conflits entre propriétés sont résolus suivant les règles de précédence suivantes :
  - les propriétés de $\mathcal{C}$ ont la précédence sur toutes les super-classes ;
  - entre les super-classes de la classe $\mathcal{C}$, la précédence est fixée par une liste.

Les réseaux à héritage font partie intégrante de la plupart des outils de systèmes experts actuels. Ils sont particulièrement utiles pour exprimer, de manière compacte, les connaissances de définition liées aux concepts d'un domaine. La notion d'héritage hiérarchique a également été adoptée dans de nombreux langages de programmation orientée-objets, dans le but de structurer les classes d'objets.

4.3 Logiques descriptives

Les logiques descriptives (*description logics*) sont une formalisation des cadres et de leurs relations d'héritage. Elles permettent en particulier de faire des inférences sur les propriétés de concepts et les relations entre classes.

Dans une logique descriptive, les cadres sont représentés par des *concepts*, généralement décrits par des lettres majuscules comme C ou D. Les propriétés (slots) sont décrites par des relations. Pour une relation R, la notation $C.R$ donne les concepts D tels que $R(C, D)$. On s'intéresse ensuite à l'héritage, qui est une relation de sous-ensemble : C hérite les propriétés de D si les instances de C sont un sous-ensemble des instances de D et on écrit : $C \sqsubseteq D$. Finalement, on peut construire la conjonction de concepts C et D par l'intersection des instances $(C \sqcap D)$ et la disjonction par l'union $(C \sqcup D)$.

Par rapport aux réseaux d'héritage, les logiques descriptives ajoutent notamment des possibilités de quantification et de restrictions sur les valeurs. On décrit par $\forall R.C$ les concepts dont toutes les instances qui sont en relation R sont des instances de C :

$$\forall R.C = x | \forall y : R(x, y) \Rightarrow C(y)$$

De manière analogue, on écrit $\exists R.C$ pour celles pour lesquelles il existe une instance de C en relation R :

$$\exists R.C = x | \exists y : R(x, y) \wedge C(y)$$

Parmi les restrictions, notons des restrictions de nombre ; par exemple $(\leq nR)$ donne toutes les instances qui sont en relation R avec au plus n autres instances.

Considérons comme exemples les expressions suivantes :

1) `père` $\sqsubseteq$ `personne` $\sqcap \exists$`enfant`
   le concept `père` est un sous-ensemble des personnes qui ont des enfants.
2) `père` $=$ `personne` $\sqcap$ `masculin` $\sqcap \exists$`enfant`
   le concept `père` est défini comme une personne masculine qui a des enfants.
3) $\forall$ `enfant.masculin`
   toutes les instances dont tous les enfants sont masculins.
4) $\exists$ `enfant.masculin` $\sqcup \exists$ `enfant.feminin`
   toutes les instances qui ont un fils ou une fille.
5) $\exists$ `enfant` (sans qualification)
   toutes les instances qui ont un enfant.

L'importance des logiques descriptives se retrouve dans le fait qu'elles admettent des procédures de raisonnement spécialisées et efficaces. Le raisonnement peut avoir différents buts :

- Satisfiabilité de concepts : $\Sigma \not\models C \equiv \bot$
  Est-ce que C est compatible avec la base de connaissances ?
- Subsumption : $\Sigma \models C \sqsubseteq D$?
  Est-ce que C est une sous-classe de D ?
  Une telle inférence est utile pour la classification de concepts : trouver les concepts D tel que $C \sqsubseteq D$. Elle a par exemple les applications suivantes :
  - intégrer un concept dans une base de connaissances ;
  - lier une requête à une classification d'informations ;
  - traduire des informations entre différentes représentations.
- Consistance : $\Sigma \not\models \bot$
  Elle est utile pour vérifier la cohérence : est-ce qu'une base de connaissances est contradictoire ?
- Vérification d'instances : $\Sigma \models C(a)$
  Est-ce que a est une instance de C ? Cette inférence est utile pour retrouver les propriétés d'une instance. Les applications sont par exemple de trouver les propriétés d'un objet (base de données) ou de filtrer les objets pour trouver ceux qui répondent à certains critères.

Il existe d'ailleurs les réductions suivantes :

- Subsumption $\Rightarrow$ Satisfiabilité :
  $\Sigma \models C \sqsubseteq D$ ssi $\Sigma \models (C \sqcap \neg D) \equiv \bot$
- Vérification d'instances $\Rightarrow$ Consistance :
  $\Sigma \models C(a)$ ssi $\Sigma \cup \{\neg C(a)\} \models \bot$

La figure 4.4 montre les constructeurs admis par une logique descriptive simple, la logique $\mathcal{FL}^-$. Considérons donc, à titre d'exemple, l'algorithme de subsumption ($C \sqsubseteq D$?) pour la logique $\mathcal{FL}^-$. Supposons d'abord que la T-Box (voir ci-après) est vide, c'est-à-dire qu'il n'y a pas d'autres connaissances sur les classes que celles des expressions mêmes. L'inférence progresse alors en deux pas :

1) Transformation en forme canonique :
   - $A \sqcap (B \sqcap C) \Rightarrow A \sqcap B \sqcap C$
   - $\forall R.C \sqcap \forall R.D \Rightarrow \forall R.(C \sqcap D)$

   $\Rightarrow C = C_1 \sqcap C_2 \sqcap C_3 ... \sqcap C_m$
   $\Rightarrow D = D_1 \sqcap D_2 \sqcap D_3 ... \sqcap D_n$
2) Vérification de chaque facteur D_i :
   - si D_i est un atome, ou de la forme $\exists P$, alors il doit y avoir un $C_j = D_i$,
   - si D_i a la forme $\forall P.D'$, alors il doit y avoir un $C_j = \forall P.C'$ tel que $C' \sqsubseteq D'$.

La complexité est quadratique dans la taille des descriptions : $O(|C| \times |D|)$.

Nom	Syntaxe	Exemple
Conjonction	$A \sqcap B$	`personne` $\sqcap$ `jeune`
Quantification Universelle	$\forall R.C$	$\forall$ `enfant.mâle`
Quantification Existentielle	$\exists R.\top$	$\exists$ `enfant`

Fig. 4.4 *Les constructeurs admis par la logique descriptive* $\mathcal{FL}^-$.

Considérons un exemple d'une inférence en $\mathcal{FL}^-$: prouver que la classe C de toutes les personnes qui ont un garçon mineur est une sous-classe de la classe D des personnes qui ont un enfant mineur. Il s'agit donc d'une subsumption $C \sqsubseteq D$:

$C =$ `personne` $\sqcap \exists$ `enfant` $\sqcap$
  $\forall$ `enfant.mineur` $\sqcap \forall$ `enfant.masculin`
$\Rightarrow C' =$ `personne` $\sqcap \exists$ `enfant` $\sqcap$
  $\forall$ `enfant.`(`mineur` $\sqcap$ `masculin`)
$D = D' =$ `personne` $\sqcap \exists$ `enfant` $\sqcap$
  $\forall$ `enfant.mineur`

et la subsumption est donc vérifiée.

La puissance des logiques descriptives est l'utilisation de bases de connaissances, notamment des définitions de concepts. On distingue deux bases de connaissances :

- La T-box, qui contient les connaissances *terminologiques* concernant les classes et qui fait l'objet de mécanismes d'inférence spécifiques.
- La A-Box, qui contient des *assertions* quelconques concernant les instances et fait l'objet d'un raisonnement logique classique. Elle correspond à la base de connaissances classique.

On distingue plusieurs types de T-box :

- **primitive** : admet des spécifications de concepts :
  `A` $\sqsubseteq$ `C`, `A` = nom, `C` = expression
  Exemple :
  `professeur` $\sqsubseteq$ `personne` $\sqcap \exists$ `enseigne`

- **simple** : admet en plus des définitions de concepts :
  $\texttt{A} = \texttt{C}$
  Exemple :
  $\texttt{parent} = \texttt{personne} \sqcap \exists\ \texttt{enfant}$
- **libre** : admet en plus des relations entre concepts :
  $\texttt{C} \sqsubseteq \texttt{D}$, $\texttt{C} = \texttt{D}$
  Exemple :
  $\texttt{personne} \sqcap \texttt{jeune} \sqsubseteq \texttt{étudiant} \sqcup \texttt{écolier}$

Nous allons nous limiter à des T-boxes simples et acycliques. Pour des T-boxes libres, le test de subsumption est très complexe (*EXPTIME*) ; elles sont donc peu étudiées. Pour les T-boxes cycliques, la subsumption est *PSPACE-complete* ; en plus, il y a peu de raison d'avoir des définitions cycliques dans la pratique.

Sous l'hypothèse d'une T-box simple et acyclique, on peut réduire le test de subsumption à un test entre concepts en appliquant itérativement les règles de transformation :

- $T(C \sqcap D) \rightarrow T(C) \sqcap T(D)$
- $T(\forall R.C) \rightarrow \forall R.T(C)$
- $T(\exists R) \rightarrow \exists R$
- $T(A) \rightarrow$ définition de A

S'il y a plusieurs spécifications de concepts, il peut y avoir plusieurs résultats de réécriture, et il faut considérer leur union dans l'algorithme de subsumption. Cela fait que la réécriture peut engendrer une explosion combinatoire, selon la terminologie.

Comme exemple, considérons la T-Box :

$$\begin{aligned} \Sigma \quad = \{ \quad & \texttt{femme} = \texttt{personne} \sqcap \texttt{feminin} \\ & \texttt{ecolier} \sqsubseteq \texttt{personne} \sqcap \texttt{mineur} \\ & \texttt{ecolier} \sqsubseteq \texttt{personne} \sqcap \texttt{curieux}\}\} \end{aligned}$$

On peut alors prouver la subsumption $C \sqsubseteq D$ entre :

$$C = \texttt{femme} \sqcap \exists\ \texttt{enfant} \sqcap \\ \forall\ \texttt{enfant.écolier} \sqcap \forall\ \texttt{enfant.masculin}$$

et

$$D = \texttt{personne} \sqcap \exists\ \texttt{enfant} \sqcap \\ \forall\ \texttt{enfant.mineur}$$

en tenant compte de la T-box Σ par les inférences suivantes :

$$C = \texttt{femme} \sqcap \exists\ \texttt{enfant} \sqcap \\ \forall\ \texttt{enfant.écolier} \sqcap \forall\ \texttt{enfant.masculin} \\ \Rightarrow C' = \texttt{personne} \sqcap \texttt{féminin} \sqcap \exists\ \texttt{enfant} \sqcap \\ \forall\ \texttt{enfant.}(\texttt{personne} \sqcap \texttt{mineur} \sqcap \texttt{masculin})$$

$\Rightarrow C'' =$ `personne` $\sqcap$ `féminin` $\sqcap \exists$ `enfant` $\sqcap$
$\forall$ `enfant.`(`personne` $\sqcap$ `curieux` $\sqcap$ `masculin`)
$D = D' =$ `personne` $\sqcap \exists$ `enfant` $\sqcap$
$\forall$ `enfant.mineur`
Subsumption $C' \sqsubseteq D$!

La réécriture des concepts suivant les définitions de la T-box peut produire une explosion combinatoire surtout dans le cas où les définitions contiennent des quantifications, par exemple :

$$\texttt{pauvre} = \forall \texttt{compte.vide}$$

Dans ce cas, il faut créer une version pour toutes les substitutions des quantificateurs possibles, par exemple pour chaque compte. Quand il y a plusieurs quantificateurs, il faut générer toutes les combinaisons, conduisant ainsi à une explosion exponentielle.

La logique $\mathcal{FL}^-$ n'est pas très puissante. Il existe d'autres logiques descriptives qui sont plus expressives. La figure 4.5 montre la logique $\mathcal{AL}$, et la figure 4.6 montre la complexité du raisonnement (subsumption) pour différentes extensions.

Une autre classe très expressive de logiques descriptives est donnée par les logiques $\mathcal{SHIQ}$ et $\mathcal{SHIN}$. La figure 4.7 montre les expressions qui sont possibles dans ces logiques. Ces logiques sont surtout importantes car elles sont

Nom	Syntaxe	Exemple
Conjonction	$A \sqcap B$	`personne` $\sqcap$ `jeune`
Quantification Universelle	$\forall R.C$	$\forall$ `enfant.mâle`
Quantification Existentielle	$\exists R.\top$	$\exists$ `enfant`
Tautologie	$\top$	
Contradiction	$\bot$	
Négation d'atomes	$\neg A$	$\neg$ `personne`

Fig. 4.5 *Opérateurs et exemples d'expressions de la logique $\mathcal{AL}$.*

Expressivité		$\models C \sqsubseteq D$	$\models C(a)$
$C \sqcap D$ $\forall R.C$ $\exists R$	$\mathcal{FL}^-$	P	P
$\neg A$	$\mathcal{AL}$	P	P
$\exists R.C$	$\mathcal{ALE}$	NP	PSPACE
$\neg C$	$\mathcal{ALC}$	PSPACE	
$\{a_1, ...\}$	$\mathcal{ALCO}$	PSPACE	
	$\mathcal{SHIQ}$	EXPTIME	

Fig. 4.6 *Complexité de l'opération de subsumption pour différents degrés d'expressivité.*

Nom	Syntaxe	Exemple	
Conjonction	$A \sqcap B$	`personne` $\sqcap$ `jeune`	
Disjonction	$A \sqcup B$	`vieux` $\sqcup$ `jeune`	
Négation	$\neg C$	$\neg$ (`personne` $\sqcap$ `jeune`)	$\mathcal{S}$
Q.U.	$\forall R.C$	$\forall$ `enfant.mâle`	
Q.E.	$\exists R.C$	$\exists$ `enfant.mâle`	
Rôle transitif	R.R	`soeur.enfant`	
Rôles hiérarchiques	$R \sqsubseteq S$	`mère` $\sqsubseteq$ `parent`	$\mathcal{H}$
Inversion de rôles	R^-	`enfant` $\Leftrightarrow$ `parent`	$\mathcal{I}$
Restriction de nombre	$\geq nR$	≥ 2 `enfant`	$\mathcal{N}$
Restriction qualifiée	$\geq nR.C$	≥ 2 `enfant.mâle`	$\mathcal{Q}$

Fig. 4.7 *Expressivités des logiques $\mathcal{SHIQ}$ et $\mathcal{SHIN}$.*

la base du langage OWL (*ontology web language*) de formalisation de connaissances, standardisé par le consortium du World Wide Web (W3C). Ce langage est destiné à la formalisation d'ontologies, des bases de connaissances taxonomiques utilisées pour la classification d'information sur le Web. Une ontologie permet par exemple d'automatiser l'intégration de l'information provenant de différents sites web et de construire des moteurs de recherche plus puissants et permettant par exemple l'utilisation de synonymes.

Par exemple, en utilisant une ontologie qui précise que `pomme` $\sqsubseteq$ `fruit`, un site qui offre des `pomme`s peut être trouvé comme résultat d'une requête qui cherche des `fruit`s. En plus, l'inférence peut être faite au niveau de la requête même au lieu de chaque instance, ce qui la rend efficace. Une ontologie permettrait également d'exprimer des équivalences entre différentes langues ainsi qu'entre différentes classifications.

À part leur utilisation dans le web, les logiques descriptives ont trouvé de nombreuses autres applications dans la formalisation et la fédération de bases de données, des outils de configuration, et d'autres outils d'aide à la décision. Citons par exemple `LOOM`, outil du ISI (University of Southern California), incomplet, existe depuis 1987 et disponible sous :
`http://www.isi.edu/isd/LOOM/LOOM-HOME.html`

Littérature

Le concept des *frames* et de la représentation structurée des connaissances a été introduite par Marvin Minsky dans [10]. John Sowa a été l'un des moteurs du développement dans ce domaine [11]. La collection [12] donne un aperçu des recherches sur les réseaux sémantiques. Le domaine des logiques de descriptions, bien que très vaste, est bien résumé dans [13]. Le World Wide Web Consortium publie des standards pour la représentation des connaissances ontologiques, en particulier le langage OWL [14].

Outils - domaine public

De nombreuses outils pour les langages du world wide web ont été développés par des universités, comme par exemple :
`KAON2`, pour `OWL`, de l'Université de Karlsruhe, distribué sous :

`http://kaon2.semanticweb.org/`

`FaCT++`, pour `OWL`, de l'Université de Manchester, distribué sous :

`http://owl.man.ac.uk/factplusplus/`

La page

`http://owl.cs.manchester.ac.uk/tools/list-of-reasoners/`

donne une liste d'outils disponibles pour les logiques descriptives.

4.4 Exercices

Nous souhaitons mettre à disposition des contribuables un système informatique qui leur permette de connaître les déductions auxquelles ils ont droit dans leur déclaration d'impôts. Le code d'imposition fixe les déductions par classes de contribuables.

Exercice 4.1 Modélisation

Il existe des personnes et des contribuables. Parmi les contribuables, il y a des salariés et des indépendants. Chaque contribuable déclare :

- ses enfants (des personnes)
- l'âge de chaque enfant ($<$12,12$-$18)
- son revenu (faible, moyen, élevé)
- son loyer (faible, moyen, élevé)
- son trajet au travail (faible, moyen, élevé)

Modélisez le problème par des concepts et relations en logique descriptive. Quels sont les concepts et quelles sont les relations ?

Solution à la page 349

Exercice 4.2 Déductions

On admet les déductions suivantes :

1) *enfants* : lorsque le contribuable a au moins un enfant ;
2) *loyer* : lorsque le contribuable a un loyer élevé et un revenu faible ;
3) *trajet* : lorsque le contribuable est un salarié qui a un trajet élevé ;
4) *pension* : lorsque le contribuable est un indépendant, ou salarié à revenu élevé.

Formalisez les classes qui ont droit aux déductions comme expressions en logique descriptive.

Solution à la page 349

Exercice 4.3 Raisonnement(1)

Calculez les déductions de Charles étant donné :

```
salarié(Charles)
enfant(Charles,Jacques)
trajet(Charles,élevé)
loyer(Charles,élevé)
revenu(Charles,faible)
```

Solution à la page 349

Exercice 4.4 Raisonnement(2)

Le gouvernement veut changer la loi en adaptant les déductions pour loyer de façon à ce qu'elles s'appliquent à quelqu'un qui a un loyer élevé et qui a :

- *version a)* au moins un enfant âgé entre 12-18 ans, ou
- *version b)* au moins deux enfants.

Pour les deux versions, caractérisez les groupes de personnes qui perdent le droit à la déduction et ceux qui gagnent le droit à la déduction. Quelle puissance de la logique faut-il pour exprimer ces classes ?

Solution à la page 350

Chapitre 5

Raisonnement basé sur des règles et systèmes experts

Parmi toutes les méthodes de résolution de problèmes, la préférence doit sans doute aller à celles se rapprochant le plus du raisonnement humain. Cela se justifie à plus d'un titre : d'abord un tel type de procédé facilite l'interaction entre utilisateur et programme, puisque l'utilisateur *comprend* ce que fait le programme. Un second point, non négligeable, est lié au fait qu'il est souvent important de savoir *pourquoi* un résultat a pu être obtenu. Par une explication, on se réfère à la manière par laquelle un problème a pu être résolu. Elle n'est par conséquent compréhensible que si elle se conforme au raisonnement humain. La dernière raison (et peut-être la plus importante) est liée à des questions d'efficacité : grâce à leurs expériences, les humains ont développé des stratégies de résolution remarquablement efficaces. Il est intéressant de s'inspirer de ces stratégies pour obtenir des algorithmes performants.

5.1 Systèmes experts

Inspirés par le succès de systèmes d'inférence à base de règles tel que le General Problem Solver (GPS) [15], les chercheurs ont tenté de modéliser le raisonnement humain dans des problèmes d'intérêt pratique. GPS a introduit le principe de l'analyse moyens-buts, où le raisonnement est motivé par un but à atteindre au lieu d'enchaîner aveuglement des inférences. Ce principe a été repris sous la forme de déduction en *chaînage arrière*, qui a permis des systèmes très efficaces. Le chaînage arrière peut ainsi être vu comme une stratégie d'inférence qui est très proche du raisonnement humain.

L'un des premiers programmes s'inspirant d'une telle démarche fut DENDRAL, un système analysant automatiquement des données sur la spectroscopie de masse. L'analyse de spectroscopies de masse est habituellement effectuée par des experts ayant une solide expérience en la matière. DENDRAL a modélisé le raisonnement des experts en utilisant un formalisme basé sur des règles, similaire à celui utilisé par GPS. C'est ainsi que fut créé le premier *système expert* (même si l'appellation n'existait pas encore à l'époque) de compétence comparable à celle des humains.

Les premières recherches sur les systèmes experts ont révélé que la modélisation du raisonnement d'un expert pouvait être d'une surprenante simplicité.

En général, quelques centaines de règles suffisent à reproduire la résolution de problèmes par des experts ! Cela a conduit aux nombreux succès de l'IA dans les années 1970 et au début des années 1980. Les systèmes experts sont jusqu'à ce jour utilisés en industrie. De plus, la technologie des systèmes experts a démontré clairement que c'est la *connaissance* et non pas les *algorithmes* qui est garante du comportement *intelligent* d'un programme.

Les systèmes experts sont généralement utilisés comme outils de résolution dans les applications où la conclusion est unique et bien définie, comme le diagnostic ou encore l'interprétation de données. Des exemples bien connus de systèmes experts aux résultats concluants, sont donnés ci-dessous :

- DENDRAL, le premier système expert, utilisé pour l'interprétation de spectroscopies de masse.
- MYCIN, pour diagnostiquer des infections et recommander des antibiotiques, est le système expert le mieux documenté dans la littérature.
- PROSPECTOR, utilisé pour l'interprétation de données géologiques en vue de détecter des gisements miniers.
- XCON, un des plus grands systèmes experts (plus de 10 000 règles), utilisé en son temps pour configurer des systèmes d'ordinateurs VAX.

5.1.1 Inférence à chaînage arrière

La procédure d'inférence à chaînage arrière commence par le *but* et consiste à appliquer toutes les règles possibles dans un sens "arrière" pour le réduire à des *environnements* consistant en *sous-buts*. Par sa construction, chaque environnement permet alors l'inférence du but donné. Par application itérative du processus, chaque sous-but sera à nouveau réduit, ce qui conduit à un *chaînage arrière* des règles. Le processus s'arrête lorsqu'aucune règle n'est applicable à l'ensemble des sous-buts, ou quand un environnement de sous-buts est entièrement satisfait par la base de données. Par exemple, la procédure de chaînage arrière résoudrait le problème exemple du paragraphe 3.5.4 de la manière suivante.

Le premier pas consiste à réduire successivement les buts à des environnements de sous-buts :

$$\begin{aligned} \text{R2, But} \rightarrow \{\ & \text{SB1} = \texttt{père(?y,François)}, \\ & \text{SB2} = \texttt{frère(?y,?x)}\ \} \\ \text{R1, SB1} \rightarrow \{\ & \text{SB2} = \texttt{frère(?y,?x)}, \\ & \text{SB3} = \texttt{frère(?w, François)}, \\ & \text{SB4} = \texttt{père(?y,?w)}\ \} \end{aligned}$$

Ensuite, comme les sous-buts de ce dernier environnement peuvent tous être satisfaits par la base de données, on les remplace par les substitutions de variables qu'ils impliquent :

$$\begin{aligned} \text{SB2} \rightarrow & \begin{array}{l} \text{E1} : \texttt{\{?y=Charles,?x=Francois\}}\ , \\ \text{E2} : \texttt{\{?y=Jacques,?x=Pierre\}} \end{array} \\ \text{SB3} \rightarrow & \texttt{\{?w=Charles\}} \\ \text{SB4} \rightarrow & \texttt{\{?y=Jacques,?w=Charles\}} \end{aligned}$$

Par combinaison des substitutions possibles, on obtient deux *environnements* qui permettent la déduction de la solution voulue :

E1 : { (?y=Charles, ?x=Francois) ∧ (?w=Charles) ∧
(?y=Jacques, ?w=Charles) }
E2 : { (?y=Jacques, ?x=Pierre) ∧ (?w=Charles) ∧
(?y=Jacques, ?w=Charles) }

Cependant, le premier de ces deux environnements est *contradictoire*, et par conséquent ne peut pas servir comme solution. La procédure rend donc le résultat `?x=Pierre`.

Les structures de données utilisées par une procédure de chaînage arrière sont essentiellement les mêmes que celles utilisées pour le chaînage avant. Cependant, le flux d'informations, décrit par la figure 5.1, est différent.

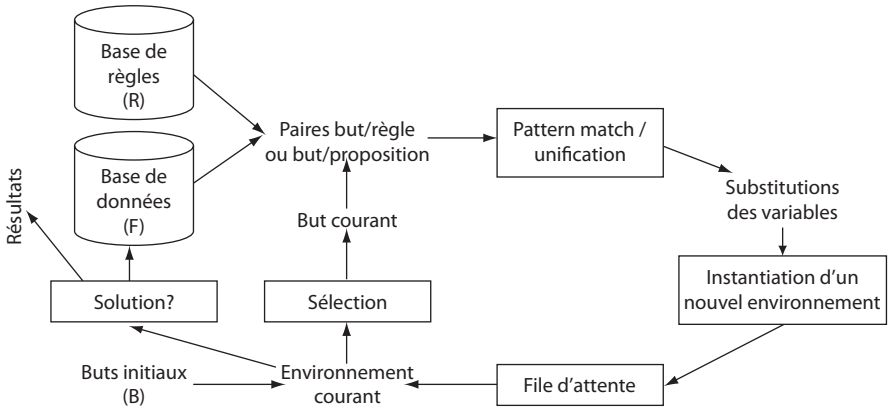


Fig. 5.1 *Flux des informations entre composants d'un moteur d'inférence à chaînage arrière.*

En fait, un algorithme de chaînage arrière est beaucoup plus compliqué à implémenter de manière efficace, car il faut tenir compte des liens entre sous-buts et environnements pour empêcher une duplication du travail :

- Si on traite chaque sous-but de façon isolée, on risque d'en résoudre plus que nécessaire : il faut uniquement satisfaire *un* des environnements générés.
- Si on traite chaque environnement de façon isolée, on répète le même travail pour chaque sous-but qui apparaît dans plus d'un environnement.

La figure 5.2 décrit une version simplifiée de l'algorithme d'inférence. Elle parcourt tous les environnements courants et vérifie d'abord si l'environnement ne contient que des substitutions de variables, c'est-à-dire si la procédure peut être arrêtée. Sinon, elle parcourt tous les buts de l'environnement pour soit les unifier avec une proposition de la base de données, soit les réduire encore à d'autres sous-buts. Notons que la procédure est simplifiée, car elle ne tient pas compte d'occurrences multiples d'un même sous-but : le travail est alors répété.

```
1: Fonction Chaînage-Arrière (R,F,B)
2: envs ← { buts-initiaux } = B
3: repeat
4:    e ← premier(envs), envs ← reste(envs)
5:    if tous les buts de e sont résolus then
6:       return instancier(buts(e),unificateurs(e))
7:    for tout but b ∈ e do
8:       for toute proposition p de la base de données F do
9:          if U ← UNIFIER(p,b) ≠ ECHEC then
10:            ajouter (e \ b) ∪ U à la file envs
11:      for toute règle r de la base de règles R do
12:         if U ← UNIFIER(droit(r),b) ≠ ECHEC then
13:            ajouter (e \ b) ∪ instancier(gauche(r),U) à la file envs
14: until envs = vide
```

Fig. 5.2 *Algorithme pour un moteur d'inférence en chaînage arrière.*

5.1.2 Critères de choix

Le choix entre le chaînage avant et arrière peut être fait sur la base du type de règles :

- `le-plus-cher(?x)` ⇒ `plus-cher(?x,?y)` :
  utilisation en chaînage arrière uniquement, car le chaînage avant exige l'unification avec un nombre infini d'individus.
- `père(?x,?y)` ⇒ `masculin(?x)` :
  analogue, mais doit être utilisée en chaînage avant.

En général, le chaînage arrière est plus efficace pour des problèmes bien ciblés tels que le planification ou le diagnostic. Par contre, le chaînage avant est plus adapté aux problèmes d'interprétation des données et est d'ailleurs beaucoup plus simple à implémenter.

Notons qu'il est envisageable de combiner les deux stratégies en déclarant certaines règles comme étant du type *avant* et d'autres du type *arrière*. Les règles de chaînage avant pourront alors être appliquées chaque fois qu'un but a été résolu par chaînage arrière. Les résultats du chaînage avant seront déposés dans la base de données et pourront servir à une vérification plus rapide des sous-buts par chaînage arrière.

Lorsque plusieurs règles concourent au même but, chaque alternative doit être explorée séparément. Ceci aboutit à une *recherche* des ensembles de sous-buts qu'il est possible d'obtenir par application des différentes règles. Cette recherche peut s'effectuer au moyen d'algorithmes de recherche que nous verrons plus tard dans ce livre. Dans certains systèmes experts, des *méta-règles* sont

employées pour guider la recherche vers les règles à privilégier. Par exemple, une méta-règle de MYCIN pourrait suggérer :

```
Si le but est de ....., essayer les règles 15, 27, 45
... ne pas essayer les règles 7, 98, 162
```

En plus de sa fonction principale qui consiste à trouver des solutions, un système expert peut être doté des trois fonctionnalités suivantes :

- il peut poser des questions à l'utilisateur lorsque les informations disponibles sont insuffisantes pour résoudre le problème,
- il peut fournir des explications sur les raisons permettant d'aboutir à une conclusion,
- il est souvent doté de mécanismes lui permettant de traiter de l'information incertaine.

Ces fonctionnalités supplémentaires se justifient pour un type d'*utilisation* du système visant à en faire une *aide* à la décision plutôt qu'un résolveur de problèmes indépendant.

5.1.3 Formulation de questions

De nombreuses tâches résolues par les systèmes experts s'apparentent au diagnostic ou à l'interprétation de données. Dans de telles applications, il arrive souvent que l'information ne puisse être obtenue que par des mesures coûteuses. Aussi, par souci d'économie, un système expert ne devrait nécessiter de telles informations que lorsqu'elles sont indispensables au traitement. Cela peut être réalisé en dotant le système de mécanismes lui permettant de *demander* explicitement un complément d'informations lorsque le besoin s'en fait sentir.

À titre d'exemple, considérons MYCIN, le système diagnostiquant les infections bactériennes. La session de diagnostic démarre au moyen de plusieurs informations initiales concernant le patient : son nom, son âge, son poids et les symptômes visibles. Sur la base de ces informations initiales, MYCIN élabore des hypothèses quant aux infections possibles et propose des tests additionnels pour confirmer ou infirmer ces hypothèses. Ces tests additionnels peuvent être extrêmement coûteux et MYCIN dispose d'heuristiques lui permettant de privilégier ceux qui sont les plus économiques ou encore de grouper les tests pouvant s'exécuter ensemble. Notons au passage que cela reflète exactement le comportement d'un praticien.

En pratique, il n'est généralement pas très aisé d'obtenir une grande quantité d'informations en se basant exclusivement sur les questions posées à l'utilisateur : le système question-réponse est en effet un moyen de communication très inefficace. Une meilleure solution consiste bien sûr à donner au départ toutes les informations nécessaires à la résolution du problème par le biais d'une interface graphique.

5.1.4 Explication du raisonnement

Les systèmes experts sont souvent utilisés comme moyen d'*aide* dans les cas où les décisions à prendre peuvent avoir des conséquences très importantes. Dans

MYCIN par exemple, on pourrait envisager qu'un diagnostic erroné engendre des répercussions catastrophiques sur un patient. Il n'est pas d'usage de faire une confiance aveugle aux recommandations données par un ordinateur dans des cas aux implications si graves. Il est important qu'un système expert puisse justifier ses conclusions en faisant explicitement état des raisons qui lui ont permis d'y aboutir. Cela fournit à l'utilisateur un moyen de valider les recommandations de l'ordinateur avant de les appliquer.

Dans les systèmes basés sur des règles, chaque conclusion se base sur l'application d'une chaîne de règles à l'information initialement fournie par l'usager. Cette chaîne de règles, exprimées en langage naturel, est utilisée comme explication des résultats obtenus. Dans la plupart des cas, elles sont données à l'utilisateur sous la forme d'un dialogue dans lequel les résultats intermédiaires sont expliqués les uns après les autres. Les dialogues de la figure 5.3 ainsi que ceux de la figure 5.4 illustrent les différents types d'explications fournies par le système MYCIN.

En plus des explications relatives au résultat final, il est parfois intéressant de connaître les raisons pour lesquelles le système pose certaines questions à

```
Question WHY

In this series, how many of the cultures for the e-coli (ORGANISM-1)?
                                        (question typique posée par MYCIN)

* WHY      (Requête de l'utilisateur pour déterminer ce qui a motivé la question)

[i.e. Why is it important to determine the number of culture which were
positive for the organism?]

            (Mycin reformule la question en langue naturelle)

[1.0] This will help to determine whether there is a significant disease associated
 with the occurence of the organism.

    It has already been established that
       [1.1] the culture was taken from a sterile source
       [1.2] the number of cultures in the series is greater than 2

    Therefore, if
       [1.3] the number of cultures which were positive for
         ORGANISM-1 is greater than 2

    Then
       there is a suggestive evidence (.5) that there is a significant disease associated with
       this occurence of the e-coli (ORGANISM-1)
       [RULE 049]

            (Répétition de la question WHY pour tester le bien-fondé
             des explications fournies)
** WHY
  [ie Why is it important to determine if there is a significant disease associated with the
  occurence of the organism]

[2.0] This will help to determine a diagnosis for which the infection should be treated
.....
```

Fig. 5.3 *Explications données par* MYCIN *à la suite d'une requête WHY.*

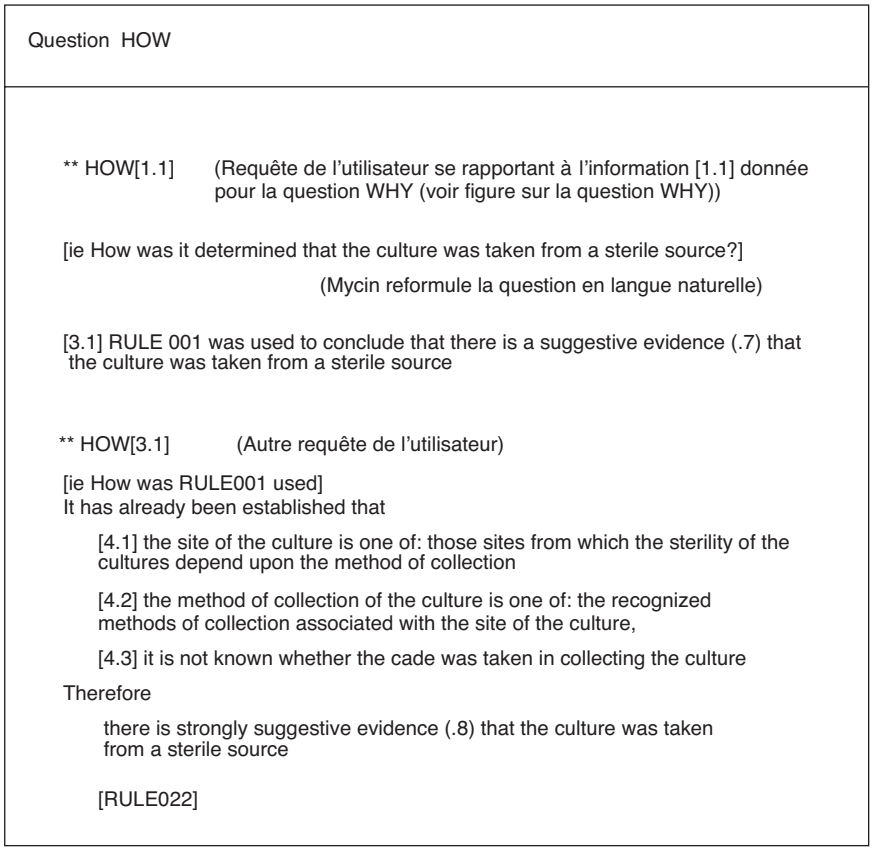


Fig. 5.4 *Explications données par* MYCIN *à la suite d'une requête HOW.*

l'usager. Un mécanisme similaire à celui précédemment vu permet de fournir ce genre d'explications car chaque question est engendrée par une règle que le système aimerait appliquer. Dans un système de diagnostic comme MYCIN, ces règles correspondent aux hypothèses de diagnostic que le système essaie de vérifier. Elles donnent par conséquent une bonne explication des raisons pour lesquelles un test donné est nécessaire.

5.2 Problèmes spécifiques liés à l'inférence

5.2.1 La négation

Dans la pratique, il peut aussi arriver qu'une conclusion soit conditionnée par le fait qu'une proposition soit fausse. Par exemple, on pourrait imaginer une règle :

$$\texttt{petite-quantité} \wedge \neg\, \texttt{frontalier} \Rightarrow \texttt{hors-taxe}$$

Cette règle n'est pas une clause de Horn, car son équivalent logique contient plus d'une proposition positive :

$\neg$ `petite-quantité` $\vee$ `frontalier` $\vee$ `hors-taxe`

On pourrait corriger ce problème en introduisant une proposition `non-frontalier` qui serait vraie exactement dans les cas où `frontalier` est fausse. Cependant, si aucune règle ne permet l'assertion explicite de $\neg$ `frontalier` ou `non-frontalier`, comment pourra-t-on satisfaire cette condition ? Du point de vue de la logique, cela ne pourrait jamais arriver, car aucune inférence ne permet de conclure la condition.

Cependant, dans la pratique, on voudrait probablement quand même déclencher la règle si on n'arrive pas à prouver l'inverse. Ceci s'appelle le principe de *Negation as failure* :

> Si le moteur d'inférence n'arrive pas à prouver `p`, alors il faut supposer que $\neg$`p` est vrai.

Cette règle semble correcte sous l'hypothèse d'une procédure d'inférence complète qui garantit que le moteur d'inférence permettra de prouver `p`.

Cependant, ce raisonnement pose un deuxième problème : on n'a pas un moyen de savoir *quand* il ne faut plus s'attendre à une preuve d'une hypothèse `p`. On pourrait penser qu'on doive attendre que la procédure ne trouve plus aucune inférence. Mais que fera-t-on si :

$\neg$ `p` $\Rightarrow$ `q`
$\neg$ `q` $\Rightarrow$ `p`

Si on déclenche d'abord la première règle, la deuxième ne sera jamais déclenchée ; on aura donc $\neg$ `p` et `q`. Si par contre, on commence par la deuxième, on finira avec `p` et $\neg$ `q`, l'inverse du premier cas ! Ce problème est en fait plus profond et est lié au fait que la négation introduit un caractère non monotone qui n'était pas prévu dans la logique classique : une proposition qui était vraie peut devenir fausse par la suite.

5.2.2 Inférences non monotones

L'application des règles d'inférence fait que l'ensemble des propositions considérées vraies est toujours en croissance monotone. On parle alors de logique *monotone*. Or, il est connu que le raisonnement humain est souvent *non monotone* : des inférences sont souvent *révisées* au moment où de nouvelles informations deviennent disponibles. Par exemple, étant donné que $Tweety$ est un oiseau, la conclusion $vole(Tweety)$ semble justifiée. Mais au moment où on découvre que $Tweety$ est en fait une autruche, l'inférence doit être révisée. En fait, le problème est que la règle :

$$oiseau \Rightarrow vole$$

n'est pas correcte et devrait être remplacée par la règle :

$$oiseau \wedge \neg autruche \wedge \neg ailes - coupées \wedge \neg ... \Rightarrow vole$$

Mais de telles règles ne peuvent être appliquées que si un grand nombre de faits sont connus. Elles sont donc peu utiles dans un système pratique ! Une formulation utile est celle de la *logique des défauts*, où des règles peuvent inclure des conditions négatives :

$$oiseau \wedge \neg anormal \Rightarrow vole$$

et où l'inférence sera faite même en absence de la connaissance explicite des conditions négatives. Si par la suite on découvre qu'une telle condition n'est effectivement pas satisfaite, le système d'inférence doit retirer la conclusion et toutes ses conséquences pour que la validité du raisonnement soit maintenue.

On ne peut cependant pas se contenter de retirer simplement une assertion qui s'est avérée invalide. En effet, cette dernière constitue peut-être la base d'une chaîne complète de déductions ultérieures qui deviennent, de ce fait, elles aussi invalides. Lorsqu'on découvre qu'en réalité l'oiseau ne vole pas, une assertion affirmant qu'il peut construire son nid sur un toit devient discutable. Le processus de rétraction doit par conséquent pouvoir remonter toute la chaîne d'inférences établies à partir d'un certain fait. L'automatisation de ce traitement fait l'objet de ce qu'il est convenu d'appeler des *systèmes de maintenance de la cohérence* (RMS : *Reason Maintenance Systems*).

Dans le cas de déductions logiques classiques, le nombre de faits déduits croît toujours de manière monotone. On parle alors d'inférence *monotone*. Par opposition, les moteurs d'inférence permettant la rétraction de faits sont dits *non monotones*.

5.2.3 Systèmes de maintenance de la cohérence

Un système de maintenance de la cohérence comporte deux aspects importants. Le premier est la représentation des faits, qui permettra aussi d'exprimer l'incertitude quant à la validité d'une proposition. Le deuxième est la manière d'ajouter et d'enlever des assertions dans la base de données ; celle-ci doit maintenir la cohérence de l'ensemble.

Représentation

Dans un système de maintenance de la cohérence (SMC), chaque proposition devient un *nœud* qui est doté d'un état explicite, traduisant sa crédibilité : `IN` ou `OUT`. `IN` veut dire que le moteur d'inférence peut prouver la proposition, soit comme prémisse ou comme conséquence d'une prémisse. Par contre, `OUT` veut dire que le moteur d'inférence n'a pas d'informations sur la véracité de la proposition. Un SMC opte pour différentes *attitudes* selon la nature du nœud qu'il traite :

- `(OUT n), (OUT (NOT n))` : le système ne sait rien quant à la véracité de n
- `(IN n), (OUT (NOT n))` : le système croit que n est vrai
- `(OUT n), (IN (NOT n))` : le système croit que n est faux
- `(IN n), (IN (NOT n))` : contradiction

Lorsqu'un nœud marqué par `IN` est retiré, c.à.d. qu'il devient `OUT`, un algorithme de *maintenance de la cohérence* doit assurer que toutes les assertions qui en découlent sont également retirées. Si on découvre qu'un nœud marqué par `IN` est contradictoire, cela indique également qu'il y a une contradiction dans les antécédents qui ont permis de déduire ce nœud.

Pour qu'on puisse retrouver ces antécédents, chaque nœud d'un SMC contient des *justifications* qui indiquent tous les chemins d'inférence par lesquels le nœud a été déduit. Une justification contient :

- la règle qui a donné la proposition associée au nœud comme conclusion,
- les nœuds qui ont été utilisés pour satisfaire les conditions de la règle.

Afin d'améliorer l'efficacité de traitement, un SMC maintient en plus explicitement des pointeurs entre tout nœud et ses conséquences. On peut représenter un nœud par une liste :

`(IN/OUT` $<$ *proposition* $>$ $<$ *justification* $>$`)`

On peut ainsi aussi admettre plusieurs justifications qui seront ajoutées à la fin de la liste, par exemple quand la même proposition a été déduite par plusieurs inférences.

Par exemple, en utilisant les notations d'un SMC, la règle qui permet de déduire qu'un oiseau peut voler peut être formulée par une règle `LES-OISEAUX-VOLENT` comme suit :

```
(IN oiseau(?x)?j1) ∧ (OUT anormal(?x)?j2 ⇒
(IN vole(?x) (LES-OISEAUX-VOLENT?j1 (OUT?j2)))
```

Si `A` et `B` sont les justifications des deux conditions,

```
(LES-OISEAUX-VOLENT A (OUT B))
```

sera la justification de la conclusion construite par l'application de la règle.

Algorithme d'assertion/rétraction

Un système de maintenance de la cohérence est lié à un moteur d'inférence de sorte que toute assertion dans la base de données s'opère via le SMC, de même que pour les requêtes concernant l'état des propositions existantes. Le SMC peut être appelé dans deux fonctions :

1) Si une proposition est déduite par le moteur d'inférence, le SMC doit l'ajouter comme nœud à la base de données s'il n'existe pas déjà. Ensuite, il doit mettre l'état du nœud à `IN` et installer la justification qui correspond au nouveau chemin de raisonnement.
2) Si on découvre une contradiction et on veut donc retirer un fait, le nœud correspondant doit être mis à `OUT`.

Chaque opération déclenche l'*algorithme de maintenance de la cohérence* qui en propage les effets sur la base de données. L'implémentation détaillée dépendra beaucoup du contexte de l'application, donc nous nous contentons ici d'un schéma de l'algorithme :

1) Si un nœud existe déjà pour la proposition, ajouter la nouvelle justification sinon créer un nouveau nœud `N` en lui associant la justification. Mettre l'état du nœud à `IN` ou `OUT` selon l'assertion effectuée.
2) Construire une liste L contenant toutes les conséquences de `N`.
3) Pour tout nœud de L, réévaluer les justifications pour voir s'il en existe une valide indépendamment de `N`. Si elles sont toutes invalides, mettre le nœud à `OUT` et appliquer récursivement la procédure de maintenance de la cohérence. Si, par contre, une justification valide existe, marquer le nœud avec un `IN`. Si le nœud est marqué `IN`, réévaluer les justifications de toutes ses conséquences (qui peuvent alors être devenues valides).
4) Contrôler s'il y a une contradiction. Si un nœud permet de déduire un nœud *nogood*, il y a une contradiction. Le système détecte une contradiction lorsqu'un nœud *nogood* devient `IN`, ce qui est signalé au moteur d'inférence qui doit alors retirer des nœuds de sorte à lui restituer un état `OUT`.

Le fait que l'on propage toute modification de l'état d'un nœud sur ses conséquences constitue la principale caractéristique de l'algorithme de maintenance de la cohérence. Comme chaque nœud peut avoir été dérivé de différentes manières, il est nécessaire de réévaluer ses justifications pour voir si elles sont toujours valables. Le processus de propagation peut devenir très coûteux car il est possible qu'un nœud puisse avoir un nombre important de conséquences. Les systèmes de maintenance de la cohérence doivent, pour cela, être utilisés avec précaution.

Lorsque des contradictions sont découvertes, il est utile de les mémoriser afin que les déductions qui les ont engendrées puissent être évitées par la suite. Cela peut être réalisé en marquant explicitement les nœuds contradictoires comme *nogood*.

Le *nogood* classique est une contradiction logique forte, telle que (P et $\neg P$). Il peut cependant être utile d'introduire d'autres types de contradictions. Par exemple, si une variable x ne peut prendre qu'une valeur unique, une contradiction peut être déduite à partir de ($x = a$, $x = b$ et $a \neq b$).

Lorsqu'une contradiction se produit dans le processus de raisonnement, cela signifie qu'une des valeurs conflictuelles doit être *retirée*. Comme il n'existe pas de règles générales permettant de décider laquelle des valeurs il faut retirer, le choix est délégué au moteur d'inférence qui peut s'aider d'heuristiques pour l'accomplir.

Preuves conditionnelles

Les systèmes de maintenance de la cohérence permettent de définir des hypothèses pour construire des argumentations générales. Par exemple, un SMC peut être utilisé pour démontrer la transitivité de la règle d'implication :

```
(1) A ⇒ B (prémisse)
(2) B ⇒ C (prémisse)
(3) A (hypothèse)
```

```
(4) B (modus ponens (1) (3))
(5) C (modus ponens (2) (4))
(6) A ⇒ C (preuve conditionnelle (5) (3) (1) (2))
```

En traçant la structure de justification associée à la dérivation de C, il est possible de vérifier que, pour conclure C, l'hypothèse A et deux prémisses suffisent. Une stratégie générale de preuve conditionnelle produit ainsi une règle explicite pour déduire la transitivité. Un processus similaire aurait pu être appliqué, même si les dérivations opérées étaient beaucoup plus complexes.

Une seconde forme de raisonnement basé sur des hypothèses est donnée par les *démonstrations indirectes* : le contraire du but est supposé, et il s'agit de détecter une contradiction. L'exemple suivant illustre cette forme de preuve indirecte :

```
(1) A ⇒ B (Prémisse)
(2) B ⇒ C (Prémisse)
(3) (NOT (A ⇒ C)) (Hypothèse preuve indirecte)
(4) (AND A (NOT C)) (Equivalence 3)
(5) A (élimination-et (4))
(6) B (modus ponens (5) (1))
(7) C (modus ponens (6) (2))
(8) (NOT C) (élimination-et (4))
(9) CONTRADICTION ((7) (8))
(10)(A ⇒ C) (preuve indirecte (1) (2) (9))
```

Littérature

Le système GPS qui était l'ancêtre des systèmes experts est décrit dans l'article [15], paru en 1963. De nombreux ouvrages ont été publiés sur les systèmes experts. [16] est une bonne référence générale. Le premier système expert DENDRAL et ses développements font l'objet du livre [17]. Le livre [18] contient une description de MYCIN, un système expert pour le diagnostic d'infections qui a été beaucoup analysé. Les *business rules* sont un développement plus récent, dont [19] donne une bonne introduction.

Outils - domaine public

Parmi de nombreux outils de systèmes experts disponibles dans le domaine public, l'outil CLIPS est le mieux développé et est toujours mis à jour par ses auteurs :

```
http://clipsrules.sourceforge.net/
```

Il existe d'ailleurs un livre édité par deux des auteurs de CLIPS [20].

Outils - commercial

Comme les moteurs d'inférence, les outils pour les systèmes experts sont de plus en plus integrés dans des systèmes intelligents de plus grande envergure. La division de *cognitive computing* de IBM, par exemple, intègre souvent des systèmes experts en combinaison avec d'autres techniques tels que la compréhension du langage naturel et l'apprentissage des connaissances. Cependant, il reste des entreprises de plus petite envergure tels que Exsys qui a mis au point de nombreux systèmes avec des démonstrations en ligne (`http://www.exsys.com`).

Application : Prédiction de toxicité par le système DEREK

Lors de la synthèse d'une nouvelle substance, il est important de savoir si elle est toxique. Comme la toxicité est un phénomène complexe, sa prédiction nécessite beaucoup de connaissances.
Le système DEREK constitue une des ressources les plus utilisées par des chimistes pour la prédiction de la toxicité. Il s'agit d'un système expert qui a été d'abord développé par la compagnie Schering Agrochemical et ensuite donné à une organisation à but non lucratif (Lhasa Limited). DEREK est constamment mis à jour et intègre ainsi les connaissances cumulées d'innombrables experts du domaine.
(source : `http://www.lhasalimited.org/`)

5.3 Exercices

Exercice 5.1 Comparaison du chaînage avant et arrière

Le but de cet exercice est de comparer les avantages et les inconvénients des deux différents types de chaînage utilisés dans les systèmes experts, c'est-à-dire les chaînages avant et arrière. Nous allons explorer cette question à l'aide d'un exemple.

Un négociant en vin s'est constitué un mini-système expert pour l'aider à gérer sa cave. Ce système permet de déclasser et de bonifier des vins de différentes régions viticoles selon leur ancienneté. Un vin devient bon à partir d'un certain nombre d'années. Passé un second seuil, le vin sera déclassé. Ces seuils varient en fonction de la provenance. Voici la liste des faits :

1. Stock−Vin(Bordeaux, 1997)
2. Stock−Vin(Bordeaux, 1990)
3. Stock−Vin(Bordeaux, 1961)
4. Stock−Vin(Bordeaux, 1965)
5. Stock−Vin(Bordeaux, 1977)
6. Stock−Vin(Bordeaux, 1978)
7. Stock−Vin(Bordeaux, 1981)
8. Stock−Vin(Bourgogne, 1995)

9. Stock−Vin(Bourgogne, 1986)
10. Stock−Vin(Bourgogne, 1990)
11. Stock−Vin(Bourgogne, 2002)
12. Problème−de−Bouchon(Bordeaux, 1986)
13. Problème−de−Bouchon(Bourgogne, 1998)
14. Année−Courante = 2007

Nous définissons des règles associées. Nous supposons que le nom d'une variable commence toujours par un point d'interrogation ; par exemple : `?vin`.

1. Stock−Vin(Bordeaux, ?Année−Vin)
   => Déclasser(Bordeaux, ?Année−Vin, ?Année−Vin+40)
2. Stock−Vin(Bourgogne, ?Année−Vin)
   => Déclasser(Bourgogne, ?Année−Vin, ?Année−Vin+20)
3. Stock−Vin(Bordeaux, ?Année−Vin)
   => Bonifier(Bordeaux, ?Année−Vin, ?Année−Vin+20)
4. Stock−Vin(Bourgogne, ?Année−Vin)
   => Bonifier(Bourgogne, ?Année−Vin, ?Année−Vin+10)
5. Déclasser(?Vin, ?Année−Vin, ?Année) AND ?Année > 1900 AND ?Année < 2020
   => Déclasser(?Vin, ?Année−Vin, ?Année+1)
6. Bonifier(?Vin, ?Année−Vin, ?Année) AND ?Année > 1900
   AND ?Année < 2020 AND NOT Déclasser(?Vin, ?Année−Vin, ?Année+1)
   => Bonifier(?Vin, ?Année−Vin, ?Année+1)
7. Déclasser(?Vin, ?Année−Vin, ?Année) AND ?Année = Année−Courante
   => Eliminer(?Vin, ?Année−Vin)
8. Problème−de−Bouchon(?Vin, ?Année−Vin)
   => Eliminer(?Vin, ?Année−Vin)

Nous supposons ici que le moteur d'inférence est capable d'accomplir des calculs arithmétiques et que la négation fonctionne ("not A" est vrai s'il n'est pas possible de déduire "A" à partir des faits contenus dans la base de données).

Le négociant voudrait savoir que répondre aux deux requêtes suivantes :

- Quels sont les stocks de bon Bourgogne pour l'année courante (2007) ?
- Quels vins faut-il éliminer de la cave ?

Répondez aux questions suivantes :

1) Comment exprimer ces requêtes dans le langage de notre système expert (en chaînage avant, puis en chaînage arrière) ?
2) Si l'on utilise le chaînage arrière, quel est, en détail, le comportement du système pour la première requête, sachant que le Bourgogne auquel on s'intéresse est celui de 1995 ? C'est-à-dire, est-ce que le Bourgogne 1995 est encore bon en 2007 ?
3) Comment se comporte le système pour la seconde requête, en supposant cette fois que le mécanisme d'inférence est le chaînage avant ? Quel est le problème ?
4) Quel est le sous-ensemble de règles qui nous permet de répondre à cette requête par chaînage avant ?
5) À quoi sert la condition `?Année > 1900` dans la règle 6 ? Peut-on trouver un exemple de requête pour laquelle l'absence de cette condition pose un problème ?

6) Quel est l'inconvénient d'utiliser toutes les règles en chaînage arrière pour la première requête ?
7) Que peut-on conclure de tout ce qui précède ?

Solution à la page 351

Exercice 5.2 Programmation du chaînage arrière

Vous allez maintenant programmer un moteur d'inférence à chaînage arrière. Le principe est simple. Sachant que l'on dispose d'une base de faits et de règles, l'idée consiste à poser des questions au moteur d'inférence. Une question est une proposition, contenant éventuellement des variables, qui est interprétée comme un but à satisfaire. Le moteur d'inférence tente de satisfaire ce but en trouvant un ou plusieurs faits qui y correspondent, c'est-à-dire qui peuvent lui être unifiés. Ces faits peuvent soit exister directement dans la base de faits, soit être déduits par une combinaison de règles et d'autres faits.

Dans le premier cas, le moteur d'inférence retourne le ou les faits correspondants. Dans le second cas, il cherche l'ensemble des règles dont la conséquence correspond au but. Lorsqu'il en trouve, il applique à nouveau récursivement le même raisonnement en considérant l'ensemble des conditions de la règle comme de nouveaux sous-buts qu'il faut atteindre. Ce processus peut être interprété comme une recherche dans un espace de solutions, l'idée étant de trouver les faits unifiables avec le but.

Chaque étape de l'algorithme dépend d'un environnement, c'est-à-dire d'un ensemble de valeurs possibles pour les variables, et construit une liste de buts qui restent à satisfaire pour obtenir une preuve du but initial. Nous appelons la structure de données qui contient cette liste de buts un *nœud*. L'unification d'un but avec un fait ou avec la conclusion d'une règle donne lieu à un ou plusieurs nœuds *successeurs*, traités à leur tour dans les étapes ultérieures. Chaque noeud contient en outre une instanciation du but initial, laquelle constitue une solution lorsqu'il ne reste plus aucun sous-but à satisfaire.

Modules squelettes

Les modules suivants fournissent le squelette du programme que nous allons développer. Les modules `exemple_classification_animale.py`, `exemple_genealogie.py` et `exemple_cycle.py` permettront de le tester :

Module `.../moteur_chainage_arriere/connaissance.py` :

```
from moteur_avec_variables.proposition_avec_variables import *

class BaseConnaissances:
    def __init__ ( self , constructeur_de_regle ):
        self . faits = {}
        self . regles = {}
        self . constructeur_de_regle = constructeur_de_regle
        self .sym = 0
```

```
def ajoute_un_fait(self, fait):
    clef = tete(fait)
    if clef in self.faits:
        self.faits[clef].append(fait)
    else:
        self.faits[clef] = [fait]

def ajoute_faits(self, faits):
    for fait in faits:
        self.ajoute_un_fait(fait)

def ajoute_une_regle(self, description):
    regle = self.constructeur_de_regle(description)
    clef = tete(regle.conclusion)
    if clef in self.regles:
        self.regles[clef].append(regle)
    else:
        self.regles[clef] = [regle]

def ajoute_regles(self, descriptions):
    for description in descriptions:
        self.ajoute_une_regle(description)

def choisir_faits_interessants(self, pattern):
    clef = tete(pattern)
    faits = self.faits.get(clef, [])
    return faits

def choisir_regles_interessantes(self, pattern, unificateur):
    regles_interessantes = []
    clef = tete(pattern)
    if clef in self.regles:
        anciennes_regles = self.regles[clef]
        for regle in anciennes_regles:
            regles_interessantes.append(self.nouvelle_instance(regle, unificateur))
    return regles_interessantes

def nouvelle_instance(self, regle, unificateur):
    env = {}
    variables = set()
    for cond in regle.conditions:
        variables.update(lister_variables(cond))
    variables.update(lister_variables(regle.conclusion))
    for var in variables:
        self.sym = self.sym + 1
        nouvelle_var = '?{}'.format(self.sym)
        env[var] = nouvelle_var

    conditions = [unificateur.substitue(cond, env) for cond in regle.conditions]
    conclusion = unificateur.substitue(regle.conclusion, env)

    return self.constructeur_de_regle((conditions, conclusion))
```

Module `.../moteur_chainage_arriere/noeud.py` :

```
from moteur_avec_variables.proposition_avec_variables import *
```

```
class Noeud:
    def __init__(self, but, sous_but_courant, sous_buts_a_tester, profondeur):
        self.but = but
        self.sous_but_courant = sous_but_courant
        self.sous_buts_a_tester = sous_buts_a_tester
        self.profondeur = profondeur

    def est_terminal(self):
        print('à compléter')

    def est_solution(self):
        print('à compléter')

    def successeur(self, env, nouveaux_sous_buts, unificateur):
        print('à compléter')

    def description_standardisee(noeud):
        sous_buts = [noeud.sous_but_courant] if len(noeud.sous_but_courant) > 0 else []
        sous_buts.extend(noeud.sous_buts_a_tester)
        sous_buts = sorted(sous_buts, key=lambda prop: prop)
        but_et_sous_buts = [noeud.but] + sous_buts

        return but_et_sous_buts

    def __repr__(self):
        return '<{},{},{},{}>'.format(self.but,
                                      self.sous_but_courant,
                                      self.sous_buts_a_tester,
                                      self.profondeur)
```

Module `.../moteur_chainage_arriere/noeuds_testes.py` :

```
from moteur_avec_variables.proposition_avec_variables import *
from .noeud import Noeud

class NoeudsTestes:
    echec = 'échec'

    def __init__(self):
        # Nous utiliserons une liste standard pour stocker les descriptions des\
        # noeuds déjà testés.
        self.descriptions = []

    def ajoute(self, noeud):
        description = noeud.description_standardisee()
        self.descriptions.append(description)

    def __contains__(self, noeud):
        if not isinstance(noeud, Noeud):
            raise ValueError("Seul un noeud peut être testé.")

        description = noeud.description_standardisee()
        for descr in self.descriptions:
            if self.inclut(descr, description):
                return True

        return False
```

```
def match(self, prop1, prop2, env):
    prop1 = prop1[:]
    prop2 = prop2[:]
    env = env.copy()

    # Si les deux propositions sont vides, le processus est terminé.
    if len(prop1) == 0 and len(prop2) == 0:
        return env

    # Si une des propositions seulement est vide, les deux n'ont pas la même
    # longueur, donc elles sont bien différentes.
    elif len(prop1) == 0 or len(prop2) == 0:
        return NoeudsTestes.echec

    # Si les deux sont des variables,
    elif est_une_variable(prop1) and est_une_variable(prop2):
        # on teste si la substitution a déjà été trouvée
        if prop1 in env:
            if env[prop1] == prop2:
                return env

        # sinon on l'ajoute a l'environnement courant.
        else:
            if prop1 != prop2:
                env[prop1] = prop2
            return env

    # Si l'une des propositions est un atome, on retourne l'environnement.
    elif est_atomique(prop1) or est_atomique(prop2):
        if prop1 == prop2:
            return env
    else:
        tete1 = tete(prop1)
        reste1 = corps(prop1)
        tete2 = tete(prop2)
        reste2 = corps(prop2)
        env_tete = self.match(tete1, tete2, env)
        if env_tete == NoeudsTestes.echec:
            return NoeudsTestes.echec
        env_reste = self.match(reste1, reste2, env_tete)
        if env_reste == NoeudsTestes.echec:
            return NoeudsTestes.echec
        return env_reste

    # Aucune tentative n'a donné de résultat.
    return NoeudsTestes.echec

def inclut_sous_buts(self, sous_buts1, sous_buts2, env):
    if len(sous_buts1) == 0:
        return True

    sb1 = tete(sous_buts1)
    sous_buts_restants1 = corps(sous_buts1)

    for sb2 in sous_buts2:
        nouvel_env = self.match(sb1, sb2, env)
        if nouvel_env != NoeudsTestes.echec:
            # On teste le match avec les sous-buts restants.
            sous_buts_restants2 = [sb for sb in sous_buts2 if sb != sb2]
```

```
                if self.inclut_sous_buts(sous_buts_restants1, sous_buts_restants2, nouvel_env):
                    return True

        return False

    def inclut(self, descr1, descr2):
        env = self.match(tete(descr1), tete(descr2), {})
        if env != NoeudsTestes.echec:
            # On compare les sous-buts non-résolus.
            return self.inclut_sous_buts(corps(descr1), corps(descr2), env)
        return False
```

Module `.../moteur_chainage_arriere/chainage_arriere.py` :

```
from moteur_sans_variables.chainage import Chainage
from .noeud import Noeud
from .noeuds_testes import NoeudsTestes

class ChainageArriere(Chainage):
    def __init__(self, connaissances, unificateur):
        self.connaissances = connaissances
        self.unificateur = unificateur

    def successeurs(self, noeud):
        print('à compléter')

    def backchain(self, noeud_depart):
        print('à compléter')

    def chaine(self, pattern):
        # Retourne les solutions par chaînage arrière.
        noeud_depart = Noeud(pattern, pattern, [], 0)
        solutions = self.backchain(noeud_depart)

        return solutions
```

Module `.../exemple_classification_animale.py` :

```
from sys import argv
from moteur_chainage_arriere.connaissance import BaseConnaissances
from moteur_chainage_arriere.chainage_arriere import ChainageArriere
from moteur_avec_variables.regle_avec_variables import RegleAvecVariables
from moteur_avec_variables.unificateur import Unificateur

faits = [
    ('a-des-poils', 'blaireau'),
    ('a-des-bébés-formes', 'blaireau'),
    ('température-stable', 'blaireau'),
    ('a-des-poils', 'écureuil'),
    ('a-des-bébés-formes', 'écureuil'),
    ('température-stable', 'écureuil'),
    ('chimpanzé', 'cheetah'),
    ('gorille', 'bozo'),
    ('singe', 'babouin'),
    ('singe', 'paresseux'),
    ('chien', 'bill'),
    ('loup', 'loup-1'),
```

```
    ('lycaon', 'lycaon-1'),
    ('chat', 'mistigri'),
    ('lion', 'minet'),
    ('tigre-du-bengale', 'tigre-du-bengale-1'),
    ('a-des-poils', 'kangourou'),
    ('a-des-bébés-foetaux', 'kangourou'),
    ('température-stable', 'kangourou'),
    ('pond-des-oeufs', 'ornythorinque'),
    ('pond-des-oeufs', 'nouveau-spécimen'),
    ('a-des-poils', 'ornythorinque'),
    ('température-stable', 'nouveau-spécimen'),
    ('température-stable', 'ornythorinque'),
]

regles = [
    [[('placentaire', '?x')], ('mammifère', '?x')],
    [[('marsupial', '?x')], ('mammifère', '?x')],
    [[('monotrème', '?x')], ('mammifère', '?x')],
    [[('placentaire-1', '?x')], ('placentaire', '?x')],
    [[('genre-placentaire', '?x')], ('placentaire', '?x')],
    [[('a-des-poils', '?x'), ('a-des-bébés-formes', '?x'),
      ('température-stable', '?x')],
     ('placentaire-1', '?x')],
    [[('a-des-poils', '?x'), ('a-des-bébés-foetaux', '?x'),
      ('température-stable', '?x')],
     ('marsupial', '?x')],
    [[('pond-des-oeufs', '?x'), ('a-des-poils', '?x'),
      ('température-stable', '?x')],
     ('monotrème', '?x')],
    [[('singe', '?x')], ('genre-placentaire', '?x')],
    [[('primate', '?x')], ('singe', '?x')],
    [[('lemurien', '?x')], ('singe', '?x')],
    [[('chimpanzé', '?x')], ('primate', '?x')],
    [[('gorille', '?x')], ('primate', '?x')],
    [[('canidé', '?x')], ('genre-placentaire', '?x')],
    [[('chien', '?x')], ('canidé', '?x')],
    [[('loup', '?x')], ('canidé', '?x')],
    [[('lycaon', '?x')], ('canidé', '?x')],
    [[('félin', '?x')], ('genre-placentaire', '?x')],
    [[('chat', '?x')], ('félin', '?x')],
    [[('lion', '?x')], ('félin', '?x')],
    [[('tigre', '?x')], ('félin', '?x')],
    [[('tigre-du-bengale', '?x')], ('tigre', '?x')],
    [[('tigre-de-l-himalaya', '?x')], ('tigre', '?x')],
]

questions = [
    ('placentaire', '?quel-animal'),
    ('félin', '?quel-animal'),
    ('félin', 'mistigri'),
    ('mammifère', 'ornythorinque'),
    ('félin', 'ornythorinque'),
]

bc = BaseConnaissances(lambda descr: RegleAvecVariables(descr[0], descr[1]))
bc.ajoute_faits(faits)
bc.ajoute_regles(regles)

moteur = ChainageArriere(bc, Unificateur())
```

```
for question in questions:

    print()
    print('Question: ' + str(question))

    moteur. reinitialise ()
    moteur.chaine(question)

    moteur. affiche_solutions ()

    if len(argv) > 1 and argv[1].lower() == 'trace':
        moteur. affiche_trace ()
```

Module `.../exemple_genealogie.py` :

```
from sys import argv
from moteur_chainage_arriere.connaissance import BaseConnaissances
from moteur_chainage_arriere.chainage_arriere import ChainageArriere
from moteur_avec_variables.regle_avec_variables import RegleAvecVariables
from moteur_avec_variables.unificateur import Unificateur

femmes = ['françoise', 'julie', 'alphonsine', 'véronique',
          'charlotte', 'brigitte', 'constance']
hommes = ['jean', 'pierre', 'marc', 'philippe', 'gustave',
          'octave', 'antoine', 'hyacynthe', 'rodolphe']
tous = femmes + hommes

faits = []
faits.extend([('femme', f) for f in femmes])
faits.extend([('homme', f) for f in hommes])
faits.extend([('différent', x, y) for x in tous for y in tous if x != y])

faits.extend([
    ('parent', 'jean', 'pierre'),
    ('parent', 'françoise', 'pierre'),
    ('parent', 'philippe', 'alphonsine'),
    ('parent', 'charlotte', 'alphonsine'),
    ('parent', 'pierre', 'julie'),
    ('parent', 'alphonsine', 'julie'),
    ('parent', 'antoine', 'véronique'),
    ('parent', 'constance', 'véronique'),
    ('parent', 'octave', 'marc'),
    ('parent', 'brigitte', 'marc'),
    ('parent', 'marc', 'gustave'),
    ('parent', 'véronique', 'gustave'),
    ('parent', 'gustave', 'hyacynthe'),
    ('parent', 'julie', 'hyacynthe'),
    ('parent', 'philippe', 'rodolphe'),
    ('parent', 'brigitte', 'rodolphe'),
])

regles = [
    [('parent', '?x', '?y'), ('femme', '?x')], ('mère', '?x', '?y')],
    [('parent', '?x', '?y'), ('homme', '?x')], ('père', '?x', '?y')],
    [('parent', '?x', '?y')], ('enfant', '?y', '?x')],
    [('parent', '?x', '?y'), ('homme', '?y')], ('fils', '?y', '?x')],
    [('parent', '?x', '?y'), ('femme', '?y')], ('fille', '?y', '?x')],
```

```
    [[('parent', '?x', '?z'), ('parent', '?z', '?y')],
     ('grand-parent', '?x', '?y')],
    [[('parent', '?x', '?z'), ('parent', '?z', '?y'), ('parent', '?y', '?u')],
     ('arrière-grand-parent', '?x', '?u')],
    [[('grand-parent', '?x', '?y'), ('femme', '?x')], ('grand-mère', '?x', '?y')],
    [[('grand-parent', '?x', '?y'), ('homme', '?x')], ('grand-père', '?x', '?y')],
    [[('arrière-grand-parent', '?x', '?y'), ('femme', '?x')],
     ('arrière-grand-mère', '?x', '?y')],
    [[('arrière-grand-parent', '?x', '?y'), ('homme', '?x')],
     ('arrière-grand-père', '?x', '?y')],
    [[('parent', '?x', '?z'), ('parent', '?x', '?y'), ('homme', '?y'),
      ('différent', '?z', '?y')], ('frère', '?y', '?z')],
    [[('parent', '?x', '?z'), ('parent', '?x', '?y'), ('femme', '?y'),
      ('différent', '?z', '?y')], ('soeur', '?y', '?z')],
    ]

questions = [
    ('arrière-grand-mère', '?qui', '?qui-d-autre'),
    ('arrière-grand-parent', '?qui', '?qui-d-autre'),
    ('soeur', '?qui', '?qui-d-autre'),
    ('frère', '?qui', '?qui-d-autre'),
    ('fils', '?qui', '?qui-d-autre'),
    ('fils', 'gustave', 'marc'),
    ('fils', '?qui', 'marc'),
    ('fils', 'gustave', 'hyacynthe'),
    ('fils', 'gustave', 'ferdinand'),
]

bc = BaseConnaissances(lambda descr: RegleAvecVariables(descr[0], descr[1]))
bc.ajoute_faits(faits)
bc.ajoute_regles(regles)

moteur = ChainageArriere(bc, Unificateur())

for question in questions:

    print()
    print('Question: ' + str(question))

    moteur.reinitialise()
    moteur.chaine(question)

    moteur.affiche_solutions()

    if len(argv) > 1 and argv[1].lower() == 'trace':
        moteur.affiche_trace()
```

Module `.../exemple_cycle.py.py` :

```
from sys import argv
from moteur_chainage_arriere.connaissance import BaseConnaissances
from moteur_chainage_arriere.chainage_arriere import ChainageArriere
from moteur_avec_variables.regle_avec_variables import RegleAvecVariables
from moteur_avec_variables.unificateur import Unificateur

faits = [('r', 'd')]
regles = [[[('r', '?a')], ('q', '?b')], [[('q', '?b')], ('r', '?a')]]
questions = [('r', '?qui')]
```

```
bc = BaseConnaissances(lambda descr: RegleAvecVariables(descr[0], descr[1]))
bc. ajoute_faits ( faits )
bc. ajoute_regles ( regles )

moteur = ChainageArriere(bc, Unificateur())

for question in questions:

    print()
    print('Question: ' + str(question))

    moteur. reinitialise ()
    moteur.chaine(question)

    moteur. affiche_solutions ()

    if len(argv) > 1 and argv[1].lower() == 'trace':
        moteur. affiche_trace ()
```

Étape initiale

Le premier nœud contiendra la même proposition, la requête initiale, comme but et comme sous-but. Par exemple, si l'on recherche toutes les personnes dont Jean est le grand-père, on aura :

Base des faits :

```
('père', 'Jean', 'Pierre'), ...
```

Base des règles :

```
R1 : ('père', '?x', '?z) AND ('père', '?z', '?y') => ('grand−père', '?x', '?y')
R2 : ('père', '?x', ?z) AND ('mère', '?z', '?y') => ('grand−père', '?x', '?y')
```

Requête initiale :

```
('grand−père', 'Jean', '?x')
```

Nœud initial : `n1`, avec

```
But : ('grand−père', 'Jean', '?x')
Sous−buts : ('grand−père', 'Jean', '?x')
```

Les antécédents

Pour gérer les nœuds au cours du processus, nous devons définir deux listes, qui seront mises à jour au fur et à mesure du chaînage arrière : i) *Nœuds-à-Tester* est la liste des nœuds qui restent à visiter ; ii) *Nœuds-Testés* est la liste des nœuds qui ont déjà été visités (elle sert à éviter les cycles). Nous aurons ainsi, dans la première étape, *Nœuds-à-Tester* qui équivaut à `[n1]` (le premier nœud), et *Nœuds-Testés* égale à `[]`. En explorant `n1`, nous trouvons deux nouveaux nœuds (`n2` et `n3`), que nous ajoutons aux nœuds à tester tandis que `n1` est ajouté à *Nœuds-Testés* et ainsi de suite.

Exploration de `n1` : calcul des successeurs de `n1` en tentant de satisfaire un de ses sous-buts :

- Utilisation de la règle R1 : nouveau nœud `n2`, avec :

  But : ('grand−père', 'Jean', '?x')
  Sous−buts : ('père', 'Jean', '?z'), ('père', '?z', '?x')

- Utilisation de la règle R2 : nouveau nœud `n3`, avec :

  But : ('grand−père', 'Jean', '?x')
  Sous−buts : ('père', 'Jean', '?z'), ('mère', '?z', '?x')

Nœuds-à-Tester : `[n2, n3]`

Nœuds-Testés : `[n1]`

Exploration de `n2` : calcul des successeurs de `n2` en tentant de satisfaire un de ses sous-buts :

- Utilisation du fait `('père', 'Jean', 'Pierre')` pour satisfaire le premier sous-but de `n2` : nouveau nœud `n4`, avec :

  But : ('grand−père', 'Jean', '?x')
  Sous−buts : ('père', 'Pierre', '?x')

- Utilisation d'autres faits, ou de règles qui ont "père" comme conséquence...

Nœuds-à-Tester : `[n3, n4]`

Nœuds-Testés : `[n1, n2]`

...

Les classes utilitaires

Nous avons tout d'abord besoin d'un unificateur. Nous vous suggérons de reprendre celui que vous avez programmé vous-même, ou d'utiliser la solution que nous vous avons proposée.

La classe `BaseConnaissances` servira à contenir les faits et les règles. Elle possède des méthodes permettant d'ajouter de nouveaux faits et de nouvelles règles lors de l'initialisation, ainsi que les deux méthodes `choisir_faits_interessants` et `choisir_regles_interessantes`, qui méritent un commentaire. `choisir_faits_interessants` permet d'éviter de vérifier des faits inutiles. Elle prend en paramètre une proposition (contenant éventuellement des variables), et retourne la liste de tous les faits dont le premier élément est identique au premier élément de la proposition. Par exemple, si les faits enregistrés dans la base de connaissances sont `('père', 'Jean', 'Paul')`, `('mère', 'Martina', 'Marie')` et `('père', 'Paul', 'Martina')`, on a :

```
choisir_faits_interessants (('père', 'Marc', '?x'))
  −> [('père', 'Jean', 'Paul'), ('père', 'Paul', 'Martina')]

choisir_faits_interessants (('père', '?x', 'Jean'))
```

```
    -> [('père', 'Jean', 'Paul'), ('père', 'Paul', 'Martina')]

  choisir_faits_interessants (('mère', '?x', '?y'))
    -> [('mère', 'Martina', 'Marie')]

  choisir_faits_interessants (('cousin', '?x', 'Jean'))
    -> []
```

`choisir_regles_interessantes` prend en paramètre une proposition (la conséquence d'une règle), et retourne la liste de toutes les règles dont le premier élément de la conséquence est identique au premier élément de la proposition. Par exemple, si la base de connaissances contient les règles :

```
R1 : ('père', '?x', '?z') AND (père, '?z', '?y') => (grand−père, '?x', '?y')
R2 : ('père', '?z', '?y') AND (marié', '?z', '?x') => (mère', '?x', '?y')
```

on a :

```
choisir_regles_interessantes (('grand−père', 'Marc', '?x'))
  -> [('père', '?45', '?46') AND (père', '?46', '?47') => (grand−père', '?45', '?47')]

choisir_regles_interessantes (('grand−père', '?x', '?y'))
  -> [('père', '?1', '?2') AND (père', '?2', '?3') => (grand−père', '?1', '?3')]

choisir_regles_interessantes (('mère', '?x', '?y'))
  -> [('père', '?75', '?74') AND (marié', '?75', '?73') => (mère', '?73', '?74')]

choisir_regles_interessantes (('cousin', '?x', 'Jean'))
  -> []
```

Comme vous pouvez le remarquer, `choisir_regles_interessantes` ne retourne pas les règles avec leurs variables d'origine. Chaque invocation de cette fonction doit générer des copies des règles originales dans lesquelles les variables auront été remplacées par des variables uniques, jamais encore employées. Cette précaution est absolument vitale dans un moteur d'inférence à chaînage arrière. Elle permet d'utiliser les mêmes variables lors de la définition de règles différentes, voire de définir des règles récursives, sans qu'il n'y ait de risque de confusion. Nous pouvons illustrer ceci par un exemple en prenant le fait `('père', 'Jean', 'Marie')` et les règles suivantes :

```
R1 : ('parent', '?y', '?x') => ('ancêtre', '?y', '?x')
R2 : ('père', '?x', '?y') => ('parent', '?x', '?y')
```

Si l'on n'utilise pas des copies des règles avec variables uniques et que l'on veut connaître tous les ancêtres du système, on aura comme but `('ancêtre', '?y', '?x')` et comme sous-but par R1 `('parent', '?y', '?x')`. Il faut alors satisfaire le sous-but `('parent', '?y', '?x')`. La règle R2 permet de poursuivre le chaînage : il faut donc unifier le sous-but avec la conséquence de R2, ce qui conduit à une circularité `'?x' => '?y'` et `'?y' => '?x'` et donc un échec. Et même si l'on s'assure que deux règles différentes n'ont jamais de variables en commun, la génération de copies des règles avec des variables uniques reste nécessaire à cause des règles récursives.

`BaseConnaissances` définit donc une variable `sym`, qui va nous permettre de construire des symboles uniques. C'est la méthode `nouvelle_instance` qui est chargée de créer de nouvelles instances de chaque règle en utilisant la valeur de

`sym` (un entier) pour créer de nouveaux noms de variables et en l'incrémentant en même temps.

Les faits sont enregistrés dans un dictionnaire `faits`, qui est une variable d'instance (`self.faits = {}`). Les valeurs de ce dictionnaire sont des listes, qui regroupent les faits selon leur premier atome. Chaque clé d'accès du dictionnaire est donc le premier atome de tous les faits de la liste associée. Par exemple, avec les clés `'grand-père'` et `'grand-mère'`, nous pourrions avoir :

```
self.faits = {
    'grand-père': [
        ('grand-père', 'Paul', 'Jacques'),
        ('grand-père', 'Jean', 'Marc')
    ],
    'grand-mère': [
        ('grand-mère', 'Mathilde', 'Pierre'),
        ('grand-mère', 'Véronique', 'Françoise')
    ]
}
```

Nous représenterons les règles au moyen de la classe `RegleAvecVariables`, déjà définie précédemment, et qui possède deux attributs : `conditions`, une liste de propositions, et `conclusion`, qui est aussi une proposition. `BaseConnaissances` possède également une variable `regles`, pour stocker les règles. `regles` est un dictionnaire dont les clés sont des atomes et les valeurs des listes de règles, classées de telle sorte que chaque liste est associée au premier atome de la conclusion de tous ses éléments.

Par exemple, avec la clé d'accès `'grand-père'`, on pourrait avoir :

```
self.regles = {
    'grand-père': [
        RegleAvecVariables([('père', '?x', '?z'),
                            ('père', '?z', '?y')],
                            ('grand-père', '?x', '?y')),
        RegleAvecVariables([('père', '?x', '?z'),
                            ('mère', '?z', '?y')],
                            ('grand-père', '?x', '?y'))
    ]
}
```

Création et gestion des nœuds

Un nœud doit regrouper plusieurs informations. Nous définissons donc une classe `Noeud`, qui contient les éléments suivants :

- `but` : le but principal (la requête initiale) ;
- `sous_but_courant` : le sous-but courant ;
- `sous_buts_a_tester` : l'ensemble des sous-buts restants, qu'il faut encore satisfaire pour répondre à la requête initiale ;
- `profondeur` : la longueur du chemin exploré depuis le nœud initial.

`Noeud` possède la méthode `description_standardisee`, qui retourne une description univoque du nœud et qui nous sera utile pour le traitement de la liste

des nœuds déjà testés. Plus exactement, cette méthode retourne une liste formée du but, puis des sous-buts du nœud (y compris le sous-but courant), les sous-buts étant triés par ordre croissant selon le nom du prédicat (le premier élément de chaque sous-but).

Un nœud est-il une solution ?

Un nœud est une solution lorsqu'il ne lui reste plus de sous-buts à satisfaire et que son but principal ne contient plus de variables. Pour tester la première condition, implémentez donc la méthode `est_terminal`, qui retournera la valeur `False` si et seulement si le nœud contient encore des sous-buts à examiner.

```
class Noeud:
    ...

    def est_terminal( self ):
        ...
```

Vous pouvez ensuite utiliser cette méthode pour implémenter `est_solution`, qui doit retourner `True` si le nœud courant correspond à une solution, et `False` sinon. Dans le premier cas, le but du nœud constituera une solution, c'est-à-dire une instance de la requête initiale dont les variables éventuelles auront été remplacées.

```
class Noeud:
    ...

    def est_solution ( self ):
        ...
```

Extension d'un nœud

Lorsqu'un nœud est sélectionné pour exploration et qu'il n'est pas une solution, il faut tenter de satisfaire ses sous-buts un par un. Il convient donc d'essayer d'abord de satisfaire l'un des sous-buts, dans notre cas le sous-but courant. Un sous-but peut être satisfait de deux manières différentes :

Cas 1. Il est unifiable à un fait existant. Le sous-but est alors directement satisfait et l'on peut se pencher sur les autres. Cela permet de créer un nouveau nœud en tenant compte du résultat de l'unification qui vient d'être réussie et de l'environnement E qui en résulte. Ce nouveau nœud aura les caractéristiques suivantes : i) le but principal reste le même, sauf qu'il faut tenir compte de l'unification, c'est-à-dire qu'il faut remplacer ses variables en accord avec E ; ii) les sous-buts à satisfaire sont les sous-buts restants du nœud père, dont on aura remplacé les variables selon E. Prenons un exemple :

Nœud examiné :

```
But : ('grand−père', '?x', '?y')
Sous−buts : [('père', '?x', '?z'), ('père', '?z', '?y')]
```

Fait :

('père', 'Jean', 'Marc')

Unification du fait avec le premier sous-but :
→ environnement `{'?x' : 'Jean', '?z' : 'Marc'}`

Nouveau nœud :

But : ('grand−père', 'Jean', '?y')
Sous−buts : [('père', 'Marc', '?y')]

Cas 2. Il est unifiable à la conséquence d'une règle : il faut alors satisfaire les conditions de cette règle. Cela permet de créer un nouveau nœud, en tenant compte des nouveaux sous-buts et du résultat de l'unification entre le sous-but et la conséquence de la règle, c'est-à-dire de l'environnement résultant E. (Cette unification est nécessaire puisque le sous-but et la conséquence n'ont pas les mêmes variables *a priori*). Ce nouveau nœud aura les caractéristiques suivantes : i) le but principal reste le même, sauf qu'il faut tenir compte de l'unification en remplaçant ses variables selon E ; ii) les sous-buts à satisfaire sont les sous-buts restants du nœud père, dont on aura remplacé les variables selon E, plus les conditions de la règle que l'on veut appliquer. Prenons un exemple :

Nœud examiné :

But : ('grand−oncle', '?x', '?y')
Sous−buts : [('grand−père', '?z', '?y'), ('frère', '?x', '?z')]

Règle :

('père', '?a', '?b') AND ('père', '?b', '?c') => ('grand−père', '?a', '?c')

Unification de la conséquence de la règle avec le premier sous-but :
→ environnement `{'?z' : '?a', '?y' : '?c'}`

Nouveau nœud :

But : ('grand−oncle', '?x', '?c')
Sous−buts : [('père', '?a', '?b'), ('père', '?b', '?c'), ('frère', '?x', '?a')]

Dans les deux cas précités, la génération d'un nouveau nœud est très similaire, mis à part les sous-buts supplémentaires qui sont introduits lorsqu'un sous-but est remplacé par les conditions d'une règle. Écrivez donc dans la classe `Noeud` une méthode `successeur`, qui doit prendre comme paramètres :

- le nœud père (ici désigné par `self`) ;
- l'environnement résultant de l'unification du sous-but avec un fait ou avec la conséquence d'une règle ;
- la liste des sous-buts supplémentaires (qui peut être vide le cas échéant) ;
- l'objet unificateur nécessaire à la réalisation des substitutions (par sa méthode `substitue`).

```
class Noeud:
    ...
    def successeur( self , env, nouveaux_sous_buts, unificateur ):
        ...
```

Nœuds testés

La liste des nœuds déjà testés sera constituée par une instance de la classe `NoeudsTestes`. Celle-ci définit la méthode `__contains__`, une *méthode spéciale* de Python[1], qui permet d'utiliser la syntaxe `noeud in NoeudsTestes` pour vérifier si `noeud` est contenu dans `NoeudsTestes`, c'est-à-dire s'il a déjà été exploré. De façon complémentaire, la méthode `ajoute` permet d'ajouter un nœud à la liste chaque fois qu'il est examiné.

Un nœud sera donc considéré comme déjà exploré, et pourra être ignoré, si et seulement s'il existe déjà dans la liste un nœud ayant le même but et les mêmes sous-buts, à des substitutions variable-variable près. En d'autres termes, s'il existe un ensemble de substitutions de variables par d'autres variables qui permettent de retrouver la description d'un nœud déjà enregistré à partir du nœud en question. C'est cette vérification qu'accomplissent les méthodes de `NoeudsTestes`.

La méthode `inclut` vérifie ainsi si deux descriptions de nœuds correspondent aux variables près. Elle s'appuie sur `inclut_sous_buts`, qui tente le filtrage spécial (en s'appuyant sur `match`) de sous-buts en tenant compte d'un environnement existant. `match` teste si deux expressions sont identiques à des substitutions de variables près. Elle retourne un environnement (un dictionnaire de substitutions) si le matching a réussi, ou la constante `NoeudsTestes.echec` s'il a échoué.

Le chainage arrière

La classe `ChainageArriere` contiendra l'algorithme principal du chaînage arrière. Elle hérite de la classe `Chainage`, que nous connaissons déjà. Sa méthode `successeurs` continue le processus de génération des successeurs que nous avons discuté plus haut. Elle doit trouver tous les successeurs possibles d'un nœud, en s'appuyant sur les faits et les règles de la base de connaissances. Le processus est décrit dans l'algorithme suivant :

```
Successeurs(noeud):
1.  nouveaux_noeuds <− liste vide.
2.  FOR EACH règle r de la base des règles DO
3.      env <− unification du sous−but courant de noeud avec la conséquence de r
4.      IF env n'est pas échec ( unification réussie ) THEN
5.          ajouter à nouveaux_noeuds un nouveau noeud où:
              i ) le but est celui de noeud, dont les variables ont été remplacées
                  selon env ;
              ii ) le sous−but courant est l'un des sous−buts restants de noeud, dont
                  les variables ont été remplacées selon env ;
              iii ) les sous−buts restants sont ceux de noeud, moins le sous−but
                  courant, augmentés des conditions de la règle r ; les variables
                  auront été remplacées selon env.
6.      END IF
7.  END FOR
8.  FOR EACH fait f de la base des faits DO
9.      env <− unification du sous−but courant de noeud avec f
```

[1] `http://docs.python.org/3/reference/datamodel.html#specialnames`

```
10.     IF env n'est pas échec ( unification réussie ) THEN
11.         ajouter à nouveaux_noeuds un nouveau noeud où:
              i ) le but est celui de noeud, dont les variables ont été remplacées
                  selon env ;
              ii ) le sous−but courant est un des sous−buts restants de noeud, dont les
                  variables ont été remplacées selon env ;
              iii ) les sous−buts restants sont ceux de noeud, moins le sous−but
                  courant, dont les variables auront été remplacées selon env.
12.     END IF
13. END FOR
14. RETURN nouveaux_noeuds
END Successeurs
```

Écrivez la méthode `successeurs` de `ChainageArriere`, qui doit prendre comme paramètre un nœud et qui retourne tous les successeurs possibles de celui-ci. Nous vous suggérons d'utiliser les méthodes `choisir_regles_interessantes` et `choisir_faits_interessants` décrites plus haut. Celles-ci sont disponibles dans l'objet `connaissances` qui constitue un attribut de `ChainageArriere`. N'oubliez pas non plus de faire appel à `Noeud.successeur`.

Nous pouvons maintenant passer à la méthode principale de `ChainageArriere`, qui devra réaliser l'algorithme de chaînage proprement dit. Écrivez donc une méthode `backchain`, qui prenne comme argument un nœud initial et retourne la liste de toutes les solutions trouvées (sans doublons), ou une liste vide s'il n'en existe pas. Nous vous suggérons l'algorithme suivant :

```
Backchain(noeud_initial)
1.  noeuds_testes <− vide
2.  solutions <− liste vide
3.  noeuds_a_tester <− liste contenant noeud_initial
4.  WHILE noeuds_a_tester n'est pas vide DO
5.      n <− premier élément de noeuds_a_tester
6.      noeuds_a_tester <− reste de noeuds_a_tester
7.      IF n ne fait pas partie de noeuds_testes THEN
8.          noeuds_testes <− ajouter n a noeuds_testes
9.          IF n est une solution THEN
10.             ajouter n à solutions
11.         ELSE
12.             nouveaux_noeuds <− générer les successeurs de n
13.             ajouter nouveaux_noeuds en tête de noeuds_a_tester
14.         END IF
15.     END IF
16. END WHILE
17. RETURN solutions
END Backchain
```

```
class ChainageArriere:
    ...
    def backchain(self, noeud_depart):
        ...
```

Interface

Il est temps d'écrire une méthode qui permette d'interroger le moteur. Soit `chaine` cette méthode, qui prend comme argument une proposition à satisfaire, nous aurons :

- `ChainageArriere.chaine(('grand-père', 'Jean', 'Marc'))` demande si Jean est bien le grand-père de Marc.
- `ChainageArriere.chaine(('grand-père', 'Jean', '?x'))` permet de recenser tous les petits-enfants de Jean.
- `ChainageArriere.chaine(('grand-père', '?x', '?y'))` permet de connaître toutes les relations "grand-père".

Écrivez cette méthode, qui retournera les résultats obtenus. Elle invoquera `backchain` avec un nœud initial dont le but principal sera la proposition fournie à titre de requête, et dont les sous-buts seront réduits à un seul : la même proposition.

```
class ChainageArriere:
    ...
    def chaine(self, pattern):
        ...
```

Test du programme

Vous pouvez finalement tester votre moteur d'inférence avec le module `exemple_classification_animale.py`, qui contient un exemple de règles et de faits inspirés de la classification animale. Essayez d'inventer différentes requêtes, du style :

- Quels sont les mammifères connus du système ?

  ```
  moteur.chaine(('mammifere', '?x'))
  ```

- L'ornithorynque est-il un mammifère ?

  ```
  moteur.chaine(('mammifère', 'ornithorynque'))
  ```

- Quels animaux sont des félins ?

  ```
  moteur.chaine(('félin', '?quels-animaux'))
  ```

Vous pouvez aussi utiliser le module `exemple_genealogie.py`, qui présente des règles formalisant les relations de parenté et le module `exemple_cycle.py`, qui offre un exemple de règles circulaires.

Solutions à la page 353

Chapitre 6

Traitement de l'information incertaine

Les raisonnements logiques s'appuient sur une distinction nette entre des propositions vraies et des propositions fausses. Dans la réalité cependant, nous sommes souvent confrontés à des situations d'incertitude, dans lesquelles il est difficile d'affirmer que telle ou telle proposition est absolument vraie ou absolument fausse. Ceci arrive pour diverses raisons :

- manque de précision dans les données de départ,
- utilisation d'un raisonnement abductif, dont la conclusion est ambiguë,
- présence de facteurs non-observables qui ont une influence sur la validité du raisonnement.

Considérons comme exemple une maison intelligente. Elle dispose de nombreux capteurs, parmi lesquels un détecteur de mouvement dans le hal (M), un détecteur d'ouverture de la porte d'entrée (E), et un détecteur de bris de vitre (V). On aimerait construire un système qui permette d'interpréter ces capteurs afin de contrôler la maison, par exemple en déclenchant une alarme lors d'un cambriolage ou en éteignant la lumière lorsque personne n'est présent. Comme chaque maison est différente, on souhaiterait en outre construire un seul système de règles qui s'applique à toutes les maisons quel que soient les capteurs et leur positionnement.

Pour modéliser logiquement la situation, on commence par élaborer une déscription des éléments et des relations d'influence entre ces éléments. Pour la fonction d'alarme, on peut ainsi établir le diagramme de la figure 6.1. Le but est de construire un système de règles qui décide si une alarme est due à un cambrioleur, ou s'il s'agit d'une fausse alerte. Par exemple, on peut utiliser le fait que seul un cambrioleur briserait la vitre, mais qu'en absence de mouvement dans la maison, il s'agit probablement d'une fausse alerte.

Bien qu'un tel système nécessite un raisonnement logique, il est difficile d'y appliquer une distinction absolue entre propositions vraies et fausses, surtout si la maison comporte de nombreux capteurs :

1) le capteur de mouvement donne des mesures continues et peut être activé à un degré variable.

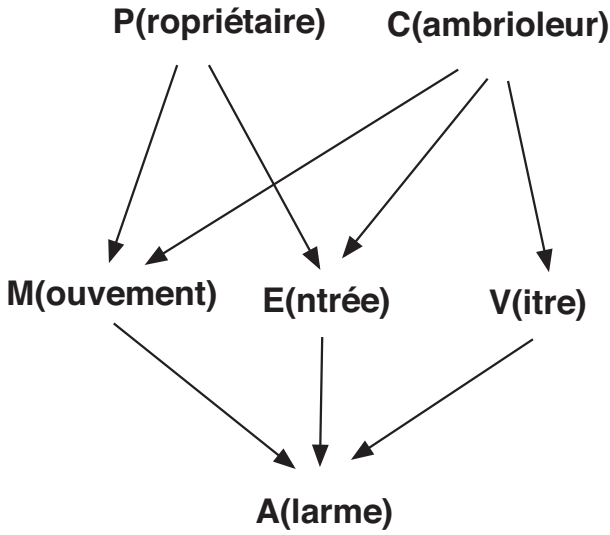


Fig. 6.1 Influence entre présence de personnes, capteurs et alarme.

2) on ne peut conclure à la présence d'un cambrioleur sur la base des signaux des capteurs qu'à travers un raisonnement abductif et ambigu : le propriétaire pourrait également en être la cause ;
3) il manque des informations sur la présence de personnes ailleurs que dans la zone couverte par le détecteur de mouvements.

Ces trois facteurs font qu'il est difficile d'implémenter une solution par raisonnement purement logique. Une telle situation est en fait très courante dans presque toutes les applications de systèmes intelligents et souligne le besoin de pouvoir tirer des raisonnements en présence d'incertitude.

Une approche qui s'est largement imposée, dans des systèmes de diagnostic médical, dans le moteur WATSON de IBM, comme dans des voitures autonomes, peut se résumer comme suit :

1) par un moteur d'inférence logique, on construit différents chemins d'inférence possibles,
2) on évalue la vraisemblance de chaque raisonnement sur la base des informations dont on dispose,
3) on choisit le ou les raisonnements qui semblent les plus vraisemblables.

Evidemment, on peut alterner les étapes 2 et 3 avec la première pour gagner en efficacité et ne pas poursuivre des hypothèses invraisemblables.

Dans notre cas, un moteur d'inférence identifierait donc d'abord les hypothèses C(ambrioleur) et P(ropriétaire) comme conséquences possibles sur la base des données A(larme), M(ouvement), E(ntrée) et V(itre). Par la suite, la vraisemblance des deux hypothèses sera évaluée par les techniques que nous allons voir dans ce chapitre.

Il existe plusieurs formalismes qui permettent le calcul utilisant des informations incertaines :

- la *logique floue* et ses variantes, comme les *facteurs de certitude*, faciles à appliquer mais sans bases théoriques solides et donc parfois incorrectes ;
- les *calculs probabilistes*, fondés sur la théorie des probabilités, mais qui s'avèrent trop complexes à appliquer en pratique ;
- les *réseaux bayésiens*, bien fondés dans leur théorie et faciles à appliquer, mais nécessitant un modèle *causal* de la réalité.

Dans ce chapitre, nous allons exposer ces techniques en détail.

6.1 De la logique floue à une représentation de l'incertitude

L'idée principale de la *logique floue* (*fuzzy logic*) est de traduire des valeurs numériques en prédicats dont la validité est « floue », comme dans l'exemple de la figure 6.2. Cette distribution exprime le pourcentage de sujets qui jugeraient une personne d'une certaine taille comme étant grande. Si cette personne est un enfant d'un mètre, seuls d'autres enfants pourraient peut-être lui appliquer ce prédicat. La plupart des gens placeraient plutôt la limite entre 1,7 et 1,8 mètre, et c'est donc l'endroit où la courbe enregistre sa plus grande croissance. Si l'on considère enfin une personne de deux mètres ou plus, tout le monde sera d'accord pour dire qu'elle est `grande` et la probabilité que le prédicat s'applique sera donc proche de 1.

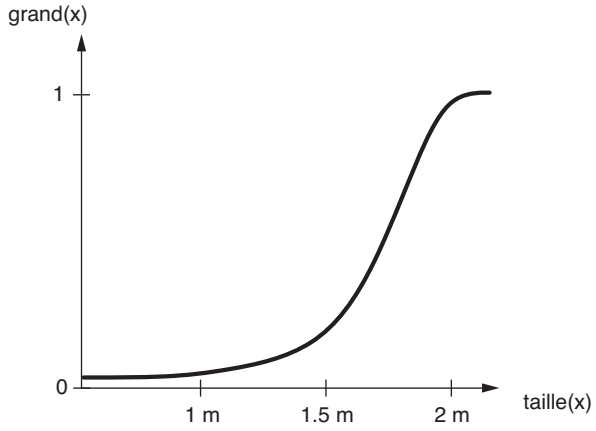


Fig. 6.2 *Exemple d'une distribution « floue » pour le prédicat* `grand`.

La distribution modélise donc l'incertitude qui résulte de la traduction d'un attribut continu en un prédicat qui ne connaît que les valeurs `vrai` et `faux`. En principe, chaque proposition qui figure dans un raisonnement peut faire l'objet d'une telle incertitude. On associe donc à chaque proposition une mesure numérique de vraisemblance, laquelle indique le degré auquel cette proposition peut s'appliquer.

L'incertitude pourra aussi se présenter en association avec des inférences. Par exemple, un cambrioleur ne déclenchera pas forcément le détecteur de mouvement, puisqu'il pourrait prendre un autre chemin dans la maison. Même si l'on est sûr qu'un cambrioleur se trouve dans la maison, le résultat d'une telle inférence restera incertain. Il faut donc prévoir que les règles d'inférence peuvent aussi être incertaines.

Finalement, une autre source d'incertitude résulte du fait que le raisonnement souhaité est souvent de nature abductive. Par exemple, l'inférence que le détecteur de mouvement indique la présence d'un cambrioleur est incertaine non pas parce la relation est incertaine, mais parce qu'il existe une autre explication, qui est la présence du propriétaire. Si on modélise une telle inférence abductive comme une inférence logique par clauses de Horn, elle doit fournir une résultat incertain.

Pour faire face à ces besoins, nous souhaitons représenter l'incertitude par des chiffres de sorte que que :

- l'incertitude de chaque proposition et de chaque inférence est caracterisée par une tel chiffre,
- l'incertitude de la conclusion d'une inférence est une fonction des incertitudes des prémisses et des règles utilisées,
- l'incertitude peut être adaptée quand de nouvelles informations apparaissent.

Nous allons d'abord examiner un formalisme simple, celui des *facteurs de certitude*, qui remplit ces exigences mais ne fournit pas toujours un résultat correct.

Partant de la reconnaissance des faiblesses de cette technique, nous discuterons ensuite de la méthode des *réseaux bayesiens*, qui permettent un vrai calcul probabiliste mais avec certaines restrictions, notamment celle d'une reconnaissance explicite de la causalité.

6.2 Les facteurs de certitude

La plupart des systèmes experts modélisent l'incertitude en attachant un certain degré de confiance aux conclusions obtenues. Dans MYCIN par exemple, on tient compte de *facteurs de certitude* (CF) représentés par des nombres réels compris entre -1.0 et 1.0. Un facteur de certitude de 1.0 signifie qu'un fait est absolument certain, une valeur de 0.5 signifie que le fait est vraisemblable, une valeur de 0.0 implique que l'on ignore totalement s'il est vrai ou faux et enfin un facteur de certitude de -1.0 indique que le fait est faux avec une certitude absolue.

Les facteurs de certitude sont attachés aussi bien aux faits qu'aux règles. Lorsqu'une règle donnée est appliquée pour déduire un nouveau fait, le facteur de certitude de ce fait est calculé par combinaison des facteurs de certitude des antécédents de la règle et de celui de la règle elle-même. Ce calcul s'effectue selon la formule suivante :

$$CF(résultat) = max(min(CF(conditions)), 0) \cdot CF(règle)$$

En prenant le minimum des CF des conditions, on considère que les conditions sont satisfaites au degré du maillon le plus faible. Si une des conditions n'est plutôt pas satisfaite, on utilise la valeur 0 et on n'attribue aucune certitude à la conclusion. L'utilisation des opérateurs *max* et *min* pour effectuer la combinaison des certitudes des conditions est la caractéristique essentielle de la *logique floue* qui est à la base des facteurs numériques de certitude.

Quand le même résultat a été trouvé par application de plusieurs règles différentes, il faut *combiner* leurs facteurs de certitude. Même s'il est très difficile de formuler ce calcul d'une manière telle que le résultat soit garanti correct, il existe différentes formules qui donnent de bons résultats en pratique. Par exemple, dans le système MYCIN, la combinaison de deux facteurs de certitude x et y, attachés à la même proposition, s'effectue comme suit :

$$CF_{\text{COMBINE}}(x,y) = \begin{cases} x + y - xy & \text{si } x \geq 0, y \geq 0 \\ \frac{x+y}{1-min(|x|,|y|)} & \text{si } x < 0, y > 0 \\ -CF_{\text{COMBINE}}(-x,-y) & \text{si } x \leq 0, y \leq 0 \end{cases}$$

Considérons maintenant l'application de ce formalisme à notre exemple. Supposons que nous voulons détecter la présence d'un cambrioleur sur la base des observations des capteurs, c'est-à-dire :

```
P1(CF=0.8) : M(ouvement)
P2(CF=0.7) : V(itre)
P3(CF=0.1) : E(ntrée)
```

et des règles :

```
R1(CF=0.9) : V(itre) ∧ M(ouvement) ⇒ C(ambrioleur)
R2(CF=0.2) : E(ntrée) ∧ M(ouvement) ⇒ C(ambrioleur)
```

ce qui permet l'inférence :

```
P4(CF=0.??) : C(ambrioleur)
```

La proposition `P4` peut être inférée soit par la règle `R1`, soit par la règle `R2`, avec les facteurs de certitude suivants :

```
R1 : CF(P4)= max(min(0.7,0.8),0) * 0.9 = 0.63
R2 : CF(P4)= max(min(0.1,0.8),0) * 0.2 = 0.06
```

Les deux valeurs sont alors combinées par la formule de combinaison pour arriver au résultat :

$$\texttt{CF(P4) = CF}_{combine}\texttt{(0.63,0.02)} = 0.63 + 0.02 - 0.63 \cdot 0.02 = 0.637$$

et il est donc assez probable qu'un cambrioleur soit présent, à cause du bris de la vitre.

L'utilisation des facteurs de certitude pour « simuler » un raisonnement abductif peut conduire à des résultats erronés, notamment dans le cas où des règles déductives sont mélangées avec des règles qui simulent l'abduction. Par exemple, les règles :

```
R1(CF=0.9) : E(ntrée) ∧ M(ouvement) ⇒ P(ropriétaire)
R2(CF=0.5) : C(ambrioleur) ⇒ E(ntrée)
R3(CT=0.99) : C(ambrioleur) ⇒ M(ouvement)
```

permettent la chaîne d'inférence :

`C(ambrioleur) (CF=1.0)` $\Rightarrow$ `E(ntrée) (CF=0.5), M(ouvement) (CF=0.99)`
$\Rightarrow$
`Propriétaire (CF=0.45)`

Ce n'est pas du tout raisonnable : le fait qu'un cambrioleur soit présent explique déjà le déclencement des détecteurs, et ne donne donc pas d'information quant à la présence du propriétaire ! Le problème de ce raisonnement, c'est que les cas dans lesquels $R1$ est vrai sont justement ceux dans lesquels $R2$ est faux : on n'a pas tenu compte de l'interdépendence entre les deux.

6.3 Réseaux bayésiens

Les *interdépendances* entre propositions sont donc le problème principal du raisonnement incertain. La logique floue et les facteurs de certitude n'en tiennent simplement pas compte. Malheureusement, en pratique, on peut observer une forte dépendance entre les règles. Les conclusions tirées sans tenir compte de cette constatation sont donc le plus souvent fausses.

Une possibilité pour pallier ce problème consiste à utiliser explicitement des probabilités. En fait, nous allons caractériser l'incertitude par une probabilité :

$p(A)$ = probabilité que la proposition A soit vraie.
$p(\neg A) = 1 - p(A)$ = probabilité que la proposition A soit fausse.
$P(A) = [p(A), p(\neg A)]$ = *distribution* de probabilité de A.

La définition classique des probabilités veut qu'une probabilité mesure la *fréquence* à laquelle un événement se produit. Ceci ne permettrait pas qu'une probabilité change au cours du raisonnement. On adopte alors une autre interprétation, qu'on appelle *bayésienne*. Dans cette interprétation, la probabilité est une *croyance* portant sur la fréquence de l'événement, croyance qui peut évoluer en fonction de nouvelles informations.

On peut définir de telles probabilités par une expérience hypothétique : supposons qu'on pose à un expert une question sous forme de pari :

> Soit un pari dans lequel vous gagnerez 100 CHF. si A est vrai. Quel est la plus grande somme x que vous seriez prêt à mettre pour y participer ?

La réponse x indique alors la probabilité bayésienne que cet expert attribue à A : $p(A) = x/100$.

On aimerait propager ces probabilités dans un raisonnement utilisant le modus ponens. Ainsi, si nous avons la règle $A \Rightarrow B$, et que $p(A)$ est connue, $p(B)$ pourrait se calculer comme suit :

$$p(B) = p(B|A) \cdot p(A) + p(B|\neg A)(1 - p(A))$$

où

$$p(B|A) = p(A, B)/p(A)$$

est la probabilité conditionnelle de B étant donné A.

Une différence importante par rapport aux facteurs de certitude est que l'incertitude de la règle s'exprime non seulement à travers $p(B|A)$, mais aussi à travers la probabilité $p(B|\neg A)$, appelée *contrefactuelle* (*counterfactual*). Cette dernière est essentielle pour exprimer l'interdépendence entre les événements, bien qu'elle rende les calculs plus complexes.

6.3.1 Chaînage d'inférences

Considérons maintenant le *chaînage* de plusieurs inférences, en rajoutant une autre règle $B \Rightarrow C$. Le calcul le plus simple serait de simplement enchaîner le même calcul deux fois. Considérons l'exemple de la maison intelligente. Soit :

$P =$ « Propriétaire présent », $p(C) = 0.9$
$M =$ « Mouvement détecté », $p(M|P) = 0.9, p(M|\neg P) = 0.01$
$A =$ « Alarme déclenchée », $p(A|M) = 1.0, p(A|\neg M) = 0.01$

le chaînage $C \rightarrow M \rightarrow A$ nous donne le résultat :

$$\begin{aligned} p(M) &= p(M|P) \cdot p(P) + p(M|\neg P)(1 - p(P)) \\ &= 0.9 \cdot 0.9 + 0.1 \cdot 0.011 = 0.811 \\ p(A) &= p(A|M) \cdot p(M) + p(A|\neg M)(1 - p(M)) \\ &= 0.811 \cdot 1 + 0.189 \cdot 0.01 \simeq 0.811 \end{aligned}$$

ce qui semble être correct.

Par contre, considérons le calcul analogue sur un autre exemple :

$P =$ « Propriétaire présent », $p(P) = 0.9$
$M =$ « Mouvement détecté », $p(M|P) = 0.9, p(M|\neg P) = 0.01$
$C =$ « Cambrioleur présent »,$p(C|M) = 0.1, p(C|\neg M) = 0.01$

Pour le chaînage $P \rightarrow M \rightarrow C$, le calcul des probabilités nous donne :

$$\begin{aligned} p(M) &= 0.9 \cdot 0.9 + 0.1 \cdot 0.01 = 0.811 \\ p(C) &= 0.811 \cdot 0.1 + 0.189 \cdot 0.01 \simeq 0.0813 \end{aligned}$$

Cependant, si nous avons déjà identifié la présence d'un cambrioleur comme la raison pour laquelle un mouvement a été detecté, il semble peu probable que le propriétaire soit présent en même temps ! Ici, nous avons une forte dépendance entre les règles : la detection du mouvement s'explique par la présence d'un cambrioleur justement dans les cas où le propriétaire n'est pas présent ! Le calcul correct devrait donc tenir compte des dépendances :

$$\begin{aligned} p(M) &= 0.811 \\ p(C) &= \underbrace{p(C|M,P)}_{=0} p(M,P) \\ &\quad + \underbrace{p(C|\neg M,P)}_{=0} p(\neg M,P) \\ &\quad + \underbrace{p(C|M,\neg P)}_{=0.9} \underbrace{p(M,\neg P)}_{=0.001} \\ &\quad + \underbrace{p(C|\neg M,\neg P)}_{=0.01} \underbrace{p(\neg M,\neg P)}_{=0.5} \\ &= 0.001 \cdot 0.9 + 0.01 \cdot 0.5 = 0.0059 \end{aligned}$$

Observons que ce calcul utilise la distribution jointe $P(C|M,P)$, ce qui est plus complexe que les distributions $P(M|P)$ et $P(C|M)$. Si on considérait aussi toutes les autres causes imaginables de M, comme les mouvements du chat, l'air chaud, la fille du propriétaire, etc., la probabilité conditionelle devrait inclure toutes ces variables aussi. Avec 10 causes $Y_1, .., Y_{10}$, on aurait besoin de la distribution $P(C|M, Y_1, .., Y_{10})$ qui compte 11 dimensions et 2048 valeurs distinctes ! Pire, si le raisonnement implique un chaînage, on doit tenir compte de toutes les variables qui y sont mentionnées.

Comment peut-on savoir quand il est nécessaire d'utiliser ce calcul plus complexe ? Une issue à ce dilemme est de profiter de la structure du monde pour identifier les endroits où l'on peut s'attendre à des dépendances. Plus précisement, il faut considérer la *causalité* entre les événements. La figure 6.1, que nous avons montrée au début du chapitre, représente un graphe qui exprime cette causalité sous forme d'arcs dirigés. En général, nous avons l'habitude d'attribuer des liens de causalité aux phénomènes du monde qui nous entoure ; de tels modèles ne sont donc pas difficiles à concevoir.

6.3.2 Importance de la causalité

Nous pouvons constater que la première inférence :

$$P \rightarrow M \rightarrow A$$

correspond à une chaîne causale, tandis que la deuxième :

$$P \rightarrow M \leftarrow C$$

n'a pas de telle correspondance, puisque P et C sont toutes les deux des causes possibles de M. Nous pouvons donc formuler l'hypothèse que le chaînage des probabilités est possible quand l'inférence suit un chemin causal dans une seule direction consistante. Mais comment concrétiser cette observation ?

La théorie des *réseaux bayésiens* formalise cette observation en utilisant la notion d'indépendance *conditionnelle*. Rappelons que deux événements A et C sont indépendants si :

$$p(C|A) = p(C|\neg A) = p(C)$$

Nous définissons l'indépendance *conditionnelle* de A et C étant donné B comme suit :

$$p(C|A,B) = p(C|\neg A,B) = p(C|B)$$

et de même pour $\neg B$:

$$p(C|A,\neg B) = p(C|\neg A,\neg B) = p(C|\neg B)$$

L'indépendance conditionnelle est fortement liée à la causalité. En fait, on peut définir la causalité comme suit :

> $Y_1, ..., Y_n$ sont les causes de X si X est conditionnellement indépendant de tous les autres événements étant donné $Y_1, .., Y_n$.

Donc, un graphe qui exprime les liens causaux entre événements nous révèle également leurs relations d'indépendance causale. Souvent, les règles utilisées pour établir un raisonnement expriment déjà des liens de causalité. Parfois cependant les règles d'un système expert correspondent à un raisonnement abductif, comme par exemple l'inférence M(ouvement) $\Rightarrow$ P(ropriétaire). Dans un tel cas, la règle suit la direction inverse de la causalité. On peut néanmoins supposer que les inférences identifient correctement les paires de nœuds du graphe qui doivent être liées par un arc.

L'indépendance conditionnelle est fort utile, puisqu'elle permet d'ignorer la plupart des dépendances lors du calcul des probabilités dans une chaînage d'inférences. Par exemple, dans une inférence $A \rightarrow B \rightarrow C$:

$$p(C|A) = p(C|A,B) \cdot p(B|A) + p(C|A,\neg B) \cdot p(\neg B|A)$$

Si A et C sont conditionnellement indépendants étant donné B :

$$p(C|A) = p(C|B) \cdot p(B|A) + p(C|\neg B) \cdot (1 - p(B|A))$$

nous n'avons pas besoin de connaître la distribution jointe $P(A,C)$ ni celle de $P(C|A,B)$ pour calculer $P(C)$, mais nous pouvons propager la probabilité localement.

L'indépendance conditionelle permet de décomposer une distribution jointe de probabilités en un produit de complexité réduite. En analogie avec des variables indépendantes A et B, où $P(A,B) = P(A) \cdot P(B)$, si A et B sont conditionellement indépendants étant donnée C on a

$$P(A,B,C) = P(A|C)P(B|C)P(C).$$

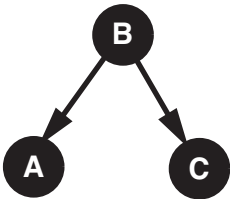
On peut donc représenter la distribution jointe des variables dans la figure 6.1 comme :

$$P(P,C,M,E,V,A) = P(P)P(C)P(M|P,C)P(E|P,C)P(V|C)P(A|M,E,V)$$

donc, au lieu d'une distribution qui compte $2^6 = 64$ valeurs, on peut faire avec $2 + 2 + 8 + 8 + 4 + 16 = 40$ valeurs. Si la différence n'est pas impressionante, supposons qu'on rajoute 3 autres variables dont A est la seule cause, et la comparaison sera entre $2^9 = 512$ et $40 + 4 + 4 + 4 = 56$ valeurs.

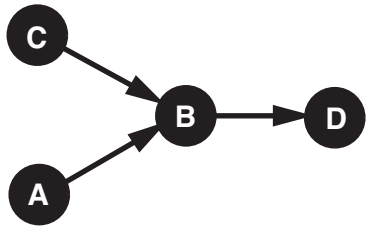
L'indépendance conditionnelle apparaît dans deux types de structures, soit dans des chaînes causales :

soit lorsque plusieurs événements ont une cause commune :



Dans un graphe causal qui ne contient que ces deux structures comme sous-graphes, il suffit de connaître les distributions de probabilités conjointes de toutes les paires d'événements pour propager les probabilités dans les deux sens.

Par contre, dans une structure où plusieurs causes conduisent au même événement B, comme :



on rencontre une *dépendance* conditionnelle qui bloque le chaînage de la propagation des probabilités. A et C sont indépendants, mais deviennent dépendants quand B est connu :

$$\begin{aligned} p(A|C) &= p(A|\neg C) = p(A) \\ p(A|B,C) &\neq p(A|B) \end{aligned}$$

et cela s'étend également aux descendants de B tels que D. La propagation est néanmoins possible, mais il faut utiliser la distribution $P(A|B,C)$ (ou bien $P(B|A,C)$ ou $P(C|A,B)$, selon la direction de propagation - tous peuvent se calculer à partir de $P(A,B,C)$).

En combinant ces trois types de structures, nous pouvons propager les probabilités dans n'importe quelle structure causale. Comme une structure causale ne peut pas contenir de cycles, elle établit parmi les variables un ordre tel que les effets suivent les causes. Voici l'ordre pour l'exemple de la figure 6.1 :

variable	nœud	parents	descendants
x_0	P	$\{\}$	$\{M,E\}$
x_1	C	$\{\}$	$\{M,E,V\}$
x_2	M	$\{P,C\}$	$\{A\}$
x_3	E	$\{P,C\}$	$\{A\}$
x_4	V	$\{C\}$	$\{A\}$
x_5	A	$\{M,E,V\}$	$\{\}$

L'algorithme général consiste à prendre les variables dans l'ordre et à calculer $P(x_i)$ comme suit : en supposant l'indépendence des variables dans $\{parents(x_i)\}$, on obtient d'abord la distribution jointe $p(x_i, parents(x_i))$:

$$p(x_i, \{parents(x_i)\}) \leftarrow p(x_i|\{parents(x_i)\}) \prod_{y_k \in parents(x_i)} p(y_k) \qquad (6.1)$$

et ensuite on marginalise les $y_k \in parents(x_i)$ un par un dans une itération d'opérations :

$$p(x_i, y_1, .., y_{k-1}) \leftarrow \sum_{y_k \in \{vrai, faux\}} p(x_i, y_1, .., y_k)$$

qui élimine la dernière variable de la distribution et va donc finalement laisser que $p(x_i)$.

Pour appliquer cet algorithme, nous devons connaître pour chaque variable la distribution conditionelle $p(x_i|\{parents(x_i)\})$. En pratique, il faut donc que la structure soit telle que chaque nœud n'ait que peu de parents, afin d'éviter des calculs prohibitifs. En tout cas, ce calcul sera plus efficace qu'un calcul réalisé sans utilisation de la causalité et dans lequel on devrait considérer la probabilité conditionelle non seulement des parents directs, mais de tous les antécédants.

La méthode ci-dessus nous permet de calculer la probabilité des effets à partir des causes. Cependant, pour estimer la probabilité de la présence d'un cambrioleur, nous devons procéder dans le sens inverse, c'est-à-dire trouver la probabilité des causes à partir des conséquences. C'est la question que nous allons aborder ci-dessous.

6.3.3 Inférence abductive

Considérons un lien causal $A \rightarrow B$. Nous avons vu que l'inférence de $P(B)$ peut se faire par propagation :

$$p(B) = p(B|A) \cdot p(A) + p(B|\neg A)(1 - p(A))$$

En supposant qu'on connaisse une probabilité à priori $p(A)$, on peut aussi renverser le sens de la propagation, en utilisant la règle de Bayes :

$$\begin{aligned} p(A|B) &= \frac{p(A,B)}{p(B)} = \frac{p(B|A)p(A)}{p(B)} \\ &= \alpha p(B|A)p(A) \end{aligned}$$

où on remplace $p(B)$, qui n'est normalement pas connu, par $1/\alpha$, car α peut être trouvé après coup en normalisant $p(A|B)+p(\neg A|B) = 1$. On peut ainsi calculer $p(A)$ étant donné n'importe quel nombre d'effets B en les intégrant ceux-ci l'un après l'autre. Par exemple, pour un événement Y qui a k conséquences $X_1....,X_k$:

$$p(Y|X_1,..,X_k) = \alpha p(Y) \prod_{i=1}^{k} p(X_i|Y) \tag{6.2}$$

où l'on obtient α simplement par le fait que la somme des probabilités pour toutes les valeurs possibles de Y doit être égale à 1 :

$$\alpha[p(Y) \prod_{i=1}^{k} p(X_i|Y) + (1 - p(Y)) \prod_{i=1}^{k} p(X_i|\neg Y)] = 1$$

C'est en raison de cette utilisation de la règle de Bayes qu'on appelle souvent de tels réseaux des *réseaux bayésiens* (*Bayesian networks*).

Reprenons notre exemple de la maison intelligente. La question la plus intéressante est de distinguer entre la présence du propriétaire et celle du cambrioleur. Supposons que les détecteurs M et V sont actifs, mais pas E. En utilisant le modèle causal, nous pouvons obtenir :

$$\begin{aligned} p(C|M,\neg E,V) &= \alpha_c \underbrace{p(C)}_{=0.01}\underbrace{p(M|C)}_{=0.9}(1-\underbrace{p(E|C)}_{=0.5})\underbrace{p(V|C)}_{=0.5} \\ &= 2.25\cdot 10^{-3}\alpha_c \end{aligned}$$

et nous pouvons obtenir α_c :

$$\begin{aligned} p(\neg C|M,\neg E,V) &= \alpha_c \underbrace{p(\neg C)}_{=0.99}\underbrace{p(M|\neg C)}_{=0.5}(1-\underbrace{p(E|\neg C)}_{=0.5})\underbrace{p(V|\neg C)}_{=0.0005} \\ &= 1.2375\cdot 10^{-4}\alpha_c \\ \Rightarrow \alpha_c &= 1/(0.00225+0.00012375) = 1/0.00237375 = 421.27 \end{aligned}$$

et donc conclure $p(C|M,\neg E,V) = 0.948$.

Par contre, le calcul analogue pour la présence du propriétaire :

$$p(P|M,\neg E,V) = \alpha_p \underbrace{p(P)}_{=0.5}\underbrace{p(M|P)}_{=0.9}(1-\underbrace{p(E|P)}_{=0.99})\underbrace{p(V|P)}_{=0.001} = 0.0000045\alpha_p$$

et

$$\begin{aligned} p(\neg P|M,\neg E,V) &= \alpha_p \underbrace{p(\neg P)}_{=0.5}\underbrace{p(M|\neg P)}_{=0.01}(1-\underbrace{p(E|\neg P)}_{=0.005})\underbrace{p(V|\neg P)}_{=0.005} = 0.000024875\alpha_p \\ \Rightarrow \alpha_p &= 1/(0.000045+0.0000495) = 1/0.000029375 = 34042.55 \end{aligned}$$

et donc $p(P|M,\neg E,V) = 0.153$. Le système peut donc bien utiliser l'information que seul le cambrioleur est susceptible de briser la vitre pour obtenir une meilleure estimation. Il est évident qu'on peut ainsi construire des systèmes d'alarme qui intègrent de nombreux capteurs et évitent des fausses alertes en utilisant le comportement habituel des habitants. Cette technique d'inférence, qu'on appelle aussi « naive Bayes », est très répandue dans la pratique.

Le raisonnement bayésien permet également de combiner des inférences dans le sens de la causalité et dans le sens opposé. Par exemple, considérons une chaîne :

avec les propositions suivantes :

A = « il y a une épidémie de méningite »
B = « le patient a la méningite »
C = « le patient a mal à la tête »

et

$$p(B|A) = 1/100, p(B|\neg A) = 1/50\,000, p(C|B) = 0.5, p(C) = 1/20$$

Dans le cas où il n'y a pas d'épidémie, nous avons $p(B) = p(B|\neg A) = 1/50\,000$ et la règle de Bayes donne le résultat :

$$\begin{aligned} p(B|C) &= \frac{p(C|B)p(B)}{p(C)} \\ &= \frac{0.5/50\,000}{1/20} \\ &= 1/5000 \end{aligned}$$

Donc, la méningite n'est pas facilement associée à un mal de tête. Par contre, s'il y a une épidémie, $p(B) = 1/100$ et le calcul donne $p(B|C) = 0.1$.

La propagation des probabilités peut se concevoir comme une propagation de « messages » entre les nœuds du graphe causal : des message du type $\pi(B)$ qui arrivent au nœud B depuis les parents (A), dans le sens de la causalité, et des message du type λ qui arrivent depuis les descendants (C), dans le sens inverse.

Considérons d'abord le cas où A et C sont observés avec certitude. On peut alors calculer :

$$p(B) \leftarrow \alpha\pi(B)\lambda(B) \tag{6.3}$$

où

$$\begin{aligned} \pi(B) &= p(B|A) \\ \lambda(B) &= p(C|B) \\ \alpha &= \text{constante de normalisation} \end{aligned}$$

Afin de pouvoir facilement calculer la constante de normalisation, il convient de propager pour chaque événement une combinaison de valeurs $(p(vrai), p(faux))$. Cela permet à tout moment d'appliquer la normalisation pour obtenir des probabilités.

Comme la chaîne causale assure que A et C sont conditionnellement indépendants étant donné B, le calcul correspond à :

$$\begin{aligned} p(B) &= p(B|A,C) \\ &= p(C|A,B)\frac{p(B|A)}{p(C|A)} \\ &= \underbrace{p(B|A)}_{\pi}\underbrace{p(C|B))}_{\lambda}\underbrace{\frac{1}{p(C|A)}}_{\alpha} \end{aligned}$$

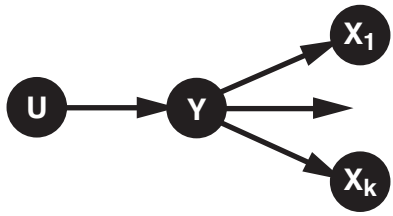
d'où on peut déjà savoir que $\alpha = \frac{1}{p(C|A)}$. En général, on peut déterminer α par une normalisation normalisation qui assure que $p(B) + p(\neg B) = 1$:

$$\begin{aligned} & p(B) + p(\neg B) \\ = \; & \alpha(p(C|A,B)p(B|A) + p(C|A,\neg B)p(\neg B|A)) \\ = \; & \alpha(p(C,B|A) + p(C,\neg B|A)) \\ = \; & \alpha p(C|A) \end{aligned}$$

ce qui nous donne également $\alpha = 1/p(C|A)$, mais s'applique de manière générale indépendamment de la complexité du calcul.

Si les valeus de A et C ne sont pas connues exactement, mais par des distributions de probabilité $P(A)$ et $P(C)$, on peut appliquer le même calcul dans une propagation (belief propagation). On décompose $\pi(B) = P(B|A) \cdot \pi(A) = \sum_{a \in A} p(a)P(B|a)$ et $\lambda(B) = P(C|B) \cdot \lambda(B) = \sum_{c \in C} P(c|B)p(c)$ et applique le même calcul de la formule 6.3.

La propagation s'applique également si il y a plusieurs déscendants. Considérons un nœud Y qui a un parent U et k descendants $X_1..X_k$:



On calcule pour chaque valeur y de Y :

$$p(y) = \alpha \cdot \pi(y) \cdot \lambda(y)$$

où

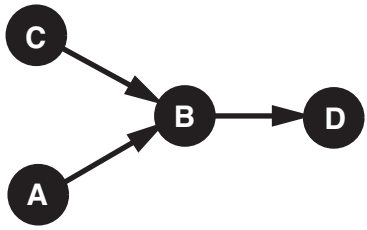
- $\pi(y) = \sum_u p(y|u)\pi_Y(u)$,
  où $\pi_Y(U)$ est le message reçu de U et contient une probabilité pour chaque valeur u de U.
- $\lambda(Y) = \prod_{j=1}^{k} \lambda_{X_j}(y)$,
  où $\lambda_{X_j}(y)$ est le message reçu de X_j.
- α est un facteur de normalisation pour que la somme des $p(y)$ pour toutes les valeurs de Y soit $= 1$.

et on envoie les messages :

- au parent U : $\lambda_X(u) = \sum_x \lambda(x)p(x|u)$
- aux descendants Y_j : $\pi_{Y_j}(x) = \alpha\pi(x) \prod_{i \neq j} \lambda_{Y_j}(x)$

Si Y n'a aucun parent, alors $\pi(y)$ est la probabilité à priori de y. Si Y est une feuille, l'ensemble X est vide, et donc le produit $\lambda(Y) = 1$. Si Y est observé, alors $p(y)$ est la probabilité après l'observation, et $\lambda(y)$ est la distribution de probabilités de y.

Notons cependant qu'une structure où un événement à plusieurs causes possibles, comme suit :



conduit à une dépendance conditionnelle qui bloque le chaînage de la propagation des probabilités. A et C sont indépendants, mais deviennent dépendants quand B est connu :

$$\begin{aligned} p(A|C) &= p(A|\neg C) = p(A) \\ p(A|B,C) &\neq p(A|C) \end{aligned}$$

L'intuition est la suivante : si A explique déjà B, C devient moins probable comme explication. Donc, il y a une dépendance entre A et C. Cette dépendance s'étend aussi aux effets de B, comme par exemple D. On dit alors que B et ses descendants sont des nœuds qui *bloquent* le chemin entre A et C, et il n'est en général pas possible de propager des probabilités sur de telles structures.

Une propagation locale des influences à travers une telle structure, par exemple pour le calcul de $P(A)$ à partir de $P(B)$, n'est possible que si toutes les autres causes sont indépendantes dans le graphe, dans l'exemple que A et C sont indépendants tant que B n'est pas connu. Dans un tel cas, on peut calculer $P(A,B,C) = P(B|A,C)P(A)P(C)$ et $P(A|B)$ par marginalisation des valeurs de C : $P(A|B) = \sum_{x \in C} P(A|B,C) = \alpha \sum_{x \in C} P(A,B,C)$. Une telle indépendance existe notamment aussi si les valeurs des autres causes sont connues exactement, ou bien si on n'a aucune information (même indirecte) sur leur valeur.

Si une telle indépendance n'est pas donnée, par exemple parce qu'il existe des chemins causaux entre une autre variable Z et A et entre Z et C, la solution est plus complexe. Pour une solution exacte, un graphe causal qui contient une telle structure doit être transformé en un graphe qui ne la contient pas par des méthodes de *clustering*. L'alternative est une solution approximative par une simulation, ce que nous allons voir plus tard. La méthode du clustering consiste en deux étapes :

1) Transformation du graphe en un graphe *moral* : pour tout nœud qui a plusieurs parents, on « marie » les parents en ajoutant un arc.
2) On regroupe les cliques dans le graphe résultant en clusters et on construit un supernœud pour chacun.

La propagation devra alors utiliser la distribution jointe de probabilités de tous ces événements, et devient vite très coûteuse.

Dans l'exemple, de la figure 6.3, le clustering pourrait générer deux nœuds :

$N_1 = \{O, Y, Z\}$
$N_2 = \{X, Y, Z\}$

liés par un lien causal qui exprime les probabilités conjointes de ces combinaisons d'événements. Si les méthodes de clustering rendent la technique des réseaux bayésiens parfaitement générale, ils augmentent considérablement la complexité des réseaux et sont de ce fait difficiles à appliquer.

Une alternative au clustering est de calculer les probabilités par une simulation des différents cas de figure qui peuvent se présenter dans le réseau et de leurs fréquences d'occurrence. Cette méthode a eu beaucoup de succès en pratique, bien qu'elle ne soit pas garantie de produire des résultats corrects.

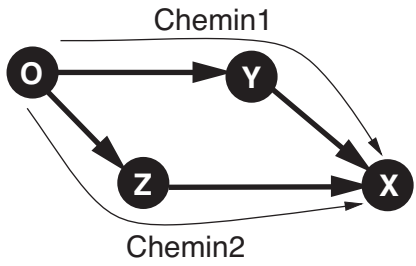


Fig. 6.3 *Exemple d'une partie d'un réseau qui contient un cycle.*

6.3.4 Chemins de causalité multiples

Même si les réseaux bayesiens réduisent fortement la complexité de l'inférence probabiliste, les distributions de probabilité conditionelle qu'on doit y formuler demeurent parfois très complexes. Considérons notre système d'alarme : si l'on devait y ajouter une vingtaine d'autres capteurs, comme on le ferait dans une grande maison, la distribution $P(A|\{capteurs\})$ serait beaucoup trop complexe pour être calculée explicitement lors de la propagation, comme l'exige l'opération 6.1. Une telle explosion survient chaque fois qu'il existe des chemins causaux multiples entre les causes et leurs conséquences.

Comme le calcul explicite de la distribution de probabilité jointe n'est pas un but en soi, mais un moyen pour le calcul, on peut utiliser d'autres méthodes. Une méthode qui s'est largement imposée consiste à réaliser une *simulation* du raisonnement sur un grand nombre de cas de figure précis. Pour estimer $P(A|C)$ dans un réseau Bayesien qui comporte plusieurs nœuds intermédiaires, on peut utiliser l'algorithme de simulation suivant :

1) Ordonner les nœuds par ordre causal.
2) Générer des instances d'état du réseau en commençant par les nœuds sans parents, et en suivant les chaînes causales :
   - générer une valeur pour chaque descendant X dès que tous les parents ont obtenus une valeur, selon la distribution $P(X|\{parents\})$ du réseau ;
   - itérer jusqu'à ce que toutes les variables aient une valeur.
3) Enregistrer la fréquence des paires de valeurs pour C et A jusquà ce qu'un nombre suffisant de valeurs aient été obtenu.
4) Estimer $P(A, C)$, et donc $P(A|C) = P(A, C)/P(C)$, par la fréquence observée.

Dans l'exemple du système d'alarme, on simulerait différents comportements du propriétaire et du cambrioleur, selon les probabilités données. Rappelons la structure du graphe :

nœud	parents	descendants
P	$\{\}$	$\{M, E\}$
C	$\{\}$	$\{M, E, V\}$
M	$\{P, C\}$	$\{A\}$
E	$\{P, C\}$	$\{A\}$
V	$\{C\}$	$\{A\}$
A	$\{M, E, V\}$	$\{\}$

On commence par générer un échantillon pour P et C selon leur distribution à priori. Ensuite on génère aléatoirement des valeurs précises pour chaque variable descendante, en accord avec les probabilités conditionelles telles qu'elles figurent dans le réseau. On répète ensuite ce processus un grand nombre de fois pour que les probabilités se stabilisent. Par exemple, on pourrait générer les échantillons :

	P	C	(M,E,V)	A
1	1	0	(1,0,0)	1
2	0	0	(0,0,0)	0
			...	
50	0	1	(1,0,1)	1
			...	

pour finalement estimer les probabilités selon les fréquences. Si parmi 10 000 échantillons, on en trouve 53 dans lesquels le propriétaire a déclenché l'alarme sur 754 dans lequels le propriétaire était présent, on peut estimer :

$$p(A|P) = \frac{count(A, P)}{count(P)} = \frac{53}{754} = 0.07$$

Si on trouve 4 échantillons dans lesquels le cambrioleur a déclenché l'alarme sur 5 où il était présent, cela nous donne :

$$p(A|C) = \frac{count(A, C)}{count(C)} = \frac{4}{5} = 0.8$$

Le point faible est évidement que sur les 10 000 échantillons, on n'en a utilisé que 5 pour estimer $p(A|C)$! En pratique, il convient donc de ne simuler que les scénarios qui ont effectivement un intérêt, ou en tout cas de surpondérer ces événements. Ici, on aimerait surpondérer les événements où le cambrioleur est présent.

Dans un cas où la variable dont nous voulons surpondérer la probabilité n'a pas de parents, cela peut se faire facilement : on ne génère que les échantillons avec les valeurs voulues, c'est-à-dire ici les échantillons où le cambrioleur est présent. Dans le cas présent, on peut se limiter aux cas où le cambrioleur est présent et générer un nombre d'échantillons beaucoup plus significatif que les 5 qu'on a obtenu par un échantillonage purement aléatoire.

Que faire, cependant, si on veut changer la pondération d'une variable dépendante, comme M ? On devrait traduire cela en une condition sur les parents, C et P, mais il n'y a pas de calcul systématique qui nous permet de générer des échantillons de C et P qui aboutissent sur la distribution voulue de M.

Dans un tel cas, il convient de générer une séquence d'échantillons, dont chacun est fonction du précédent. Comme les échantillons forment une chaîne de Markov (dans laquelle chaque élément dépend uniquement du précedent), on parle alors de *Markov Chain Monte Carlo* (MCMC). La technique la plus connue est celle de l'échantillonage de Gibbs (*Gibbs sampling*).

Dans cette méthode, on fixe les variables connues à leur valeur, et ne les fait jamais varier. Pour toutes les autres variables, on génère à chaque itération de nouvelles valeurs en fonction des valeurs des variables voisines. Dans un réseau bayésien, une variable x_i est conditionellement indépendant de *toutes* les autres variables étant donnée sa *couverture de Markov* (*Markov blanket*) $MB(x_i)$, définie comme ses parents, descendants directs et parents de ces descendants. Comme ces variables ont toutes des valeurs connues, on peut calculer la distribution de x_i étant donné l'état de toutes les variables du réseau :

$$P(x_i|MB(x_i)) = \alpha P(x_i|parents) \prod_{Y_i \in descendants(X_i)} P(y_i|parents(Y_i))$$

où α est un facteur de normalisation qui reflète les probabilités des variables de $MB(x_i)$ et qui sera calculé tel que $\sum_{x_i} P(x_i|MB(x_i)) = 1$.

À chaque itération, on choisit donc une variable x_i dont la valeur n'est pas fixée par une pondération et génère l'état suivant en lui attribuant une nouvelle valeur selon la distribution calculée ci-dessus. La mise à jour se déroule dans un ordre tel que les parents prennent les valeurs de l'itération courante, tandis que leurs descendants gardent les valeurs de l'itération précédente tant qu'ils ne sont pas mis à jour. On appelle cette manière de procéder *l'échantillonage selon Gibbs* (*Gibbs sampling*). On peut montrer que la distribution des échantillons générés par le *Gibbs sampling* converge vers celle qu'on pourrait observer dans le sous-ensemble des échantillons d'une simulation complète dont les variables fixes possèdent les valeurs voulues.

On procède aussi parfois en deux phases : stabilisation des distributions, puis échantillonage.

Dans notre exemple, supposons que nous voulons estimer $p(A|C, M)$. On ne génère alors que des échantillons où les valeurs de C et M sont fixées à 1 :

	(C,M)	P	E	V	A
1	(1,1)	0	1	0	1
2	(1,1)	1	1	0	0
3	(1,1)	0	0	1	1
4	(1,1)	0	0	1	1
5	(1,1)	1	0	1	0

...

L'estimation de la probabilité $p(A|C, M)$ se fait simplement en comptant la fréquence de A ; par exemple si on a observé $A = 1$ dans 55 échantillons sur 100 :

$$\Rightarrow Pr(A|C, M) = \frac{\text{count(A)}}{\text{nombre d'echantillons}} = \frac{55}{100} = 0.55$$

On peut appliquer ces techniques soit pour effectuer une inférence spécifique, soit pour faire une sorte de compilation d'un réseau complexe, dont on élimine

les variables intermédiaires pour ne garder que la relation conditionelle entre les variables dont la valeur est fixée à l'entrée et le résultat final. Par exemple, pour un diagnostic médical, on pourrait extraire la distribution conditionelle p(maladie|symptômes visibles) afin de pouvoir l'appliquer de manière efficace sans passer par toute la chaîne causale.

6.3.5 Applications

Les réseaux bayésiens ont connu un fort succès pratique, notamment dans des applications de diagnostic. Leur utilité repose sur deux observations. La première est qu'il est très facile pour un expert d'identifier les liens qui devraient figurer dans le réseau. La deuxième est que les résultats obtenus, c'est-à-dire la proposition qui est jugée la plus probable, sont très peu sensibles au choix précis des probabilités : les performances du système sont acceptables même si les probabilités sont estimées de manière grossière.

Parmi de nombreuses applications, on trouve :

- des systèmes de diagnostic médical,
- des systèmes de diagnostic de réacteurs d'avions,
- la plupart des filtres anti-spam,
- les divers assistants dans les logiciels de Microsoft.

Littérature

La logique floue a été présentée dans [21] et a ensuite été mentionnée dans de nombreuses autres publications. La modélisation et l'inférence dans les réseaux probabilistes bayésiens sont décrites dans [22], qui est la référence la plus connue, et aussi dans [23]. Le livre de Koller et Friedman [24] est une reférence très complète sur les techniques d'inférence Bayesiens, et celui de Darwiche [25] est un ouvrage plus compact et axé sur la pratique.

Les aspects liés à la modélisation de la causalité sont décrites dans un excellent livre de Pearl [26].

Outils - domaine public

Il existe de nombreux outils pour le raisonnement et la construction de réseaux bayésiens. Citons par exemple les *Bayesian Network tools in Java* (BNJ) :

`http://sourceforge.net/projects/bnj`

On peut mentionner également les extensions pour des programmes comme Maple.

Pour l'utilisation pratique de l'inférence probabiliste, il convient souvent de compiler le modèle pour permettre une inférence rapide. Le logiciel Ace est un tel compilateur :

`http://reasoning.cs.ucla.edu/ace/`

Outils - commercial

Les techniques de raisonnement incertain et surtout les méthodes des réseaux bayésiens font partie de toutes les logiciels d'Intelligence Artificielle commercialisés par les grands fournisseurs tels que IBM, Microsoft ou HP. Comme societé qui commercialise un outil spécifique pour les réseaux Bayesiens, citons Hugin Expert (`http://www.hugin.com`).

Application : voitures autonomes

Conduire une voiture peut être un plaisir, mais aujourd'hui, pour beaucoup, il s'agit plutôt d'une corvée dont on aimerait bien se passer. Qui n'aimerait pas avoir son propre chauffeur et utiliser le temps du parcours à travailler ou faire une sieste ?

Le progrès de la technologie a été rapide : lors d'un premier concours de l'agence de recherche US DARPA en 2004, aucun des véhicules n'a réussi à traverser plus de 5% du parcours de 150 miles dans le désert. Déjà une année après, une VW modifée par une équipe de Stanford a effectué le trajet en moins de 7 heures. En 2009, la même équipe, reprise par Google, lançait les premières voitures autonomes sur les routes publiques en Californie. On parle d'une commercialisation à large échelle avant 2020.

La technologie qui a rendu possible cette performance est celle des réseaux bayésiennes. Les voitures reconnaissent leur position, les panneaux et autres participants par des capteurs et systèmes de vision. L'information est assemblée dans un réseau bayésien pour donner une image précis de la situation du vehicule et du traffic, et ainsi permettre la planification du mouvement.

L'inférence bayésienne est essentielle pour réussir l'intégration cohérente des informations des capteurs et la situation dynamique du traffic qui nécessite une adaptation constante du modèle par des inférences logiques. Par des années d'essais, Google a construit une base de connaissances assez complète des situations qui peuvent se produire dans le traffic habituel, ce qui en fait un conducteur « système expert » chevronné !

(Source : Google Self-Driving Car Project : How it works `https ://www.google.com/selfdrivingcar/how/` (chargée le 26.5.2016))

6.4 Exercices

Exercice 6.1 Première partie - Réseaux Bayésiens

La première série d'exercices traitera des réseaux bayésiens, mais sans programmation. Le but est de passer en revue les principes du raisonnement probabiliste, la modélisation des liens de causalité et l'inférence dans un réseau bayésien.

Dans la deuxième partie, vous modifierez le moteur d'inférence à chaînage avant avec variables que nous avons développé dans les exercices des chapitres précédents afin de prendre en compte la possibilité d'incertitudes dans les faits et les règles.

Exercice 6.1.1 Raisonnement probabiliste
Après votre bilan de santé annuel, vous recevez par courrier les résultats de vos examens : vous avez été testé positif à une maladie grave. Les taux de faux positifs et négatifs sont de 1 %, c'est-à-dire que la probabilité que le test classifie un patient sain comme étant malade ou un malade comme étant sain est de 0.01. Par bonheur, cette maladie est extrêmement rare : elle ne frappe qu'une personne sur dix mille.

Question 1. À l'aide d'un raisonnement probabiliste, expliquez pourquoi la rareté de la maladie est une chance pour vous. Indication : utilisez la règle d'inférence bayésienne, $T =$ Test et $M =$ Maladie constituant les événements concernés.

Exercice 6.1.2 Causalité
Les liens de causalité expriment des relations du type $si\ A = 1, alors\ B = 1$, c'est-à-dire, selon le formalisme de la logique propositionnelle, des règles du type $A \Rightarrow B$, A et B étant deux prédicats booléens. Lorsque le raisonnement est incertain, la causalité peut être établie via un raisonnement probabiliste :

$(A \Rightarrow B)$ devient ($A \Rightarrow B$ avec probabilité $P(B|A)$)

L'inférence bayésienne[(1)] consiste alors généralement à partir d'une observation sur B afin d'établir la probabilité selon laquelle A est vrai, c'est-à-dire que l'on remonte la chaîne de causalité. Cet exercice a pour but de vous donner une idée de la raison pour laquelle cela est généralement le cas.

Considérons une petite histoire. L'inspecteur Smith se trouve à Priory School[(2)]. Il attend le détective Holmes et son ami, le docteur Watson, pour enquêter sur la disparition d'un professeur. Mais ils sont en retard. Tous deux ont la réputation d'être mauvais conducteurs et l'inspecteur Smith se demande si la route est gelée, ce qui leur causerait certainement de gros ennuis. Il téléphone à sa secrétaire, Miss Lovelace, qui l'informe que le Dr Watson a effectivement eu un accident. Voici leur conversation :

> *Smith* — Un accident ? Bien, la route étant probablement gelée, il est probable que Holmes ait aussi eu un accident !
> *Lovelace* — La route, gelée ? Non, certainement pas : il ne fait pas si froid et les routes ont été sablées.
> *Smith* — Pas de chance pour Watson alors. Attendons Holmes encore dix minutes...

(1) `http://www.ai.mit.edu/courses/6.825/fall02/pdf/6.825-lecture-15.pdf`
(2) `https://en.wikipedia.org/wiki/The_Adventure_of_the_Priory_School`

Le lendemain, l'épouse de Watson, Mary, lit le journal. Elle découvre l'accident de son mari et apprend que le professeur de Priory School a été sauvé et les ravisseurs arrêtés. Voici ses pensées :

> — Ils ont arrêté les ravisseurs ? Alors, Holmes n'a probablement pas eu d'accident, donc la route n'était probablement pas gelée.

Elle lit ensuite qu'il ne faisait pas si froid et que les routes avaient été sablées. Elle se dit :

> — Décidément, Holmes a évité l'accident.

Modélisation du problème

Question 1. Formalisez les relations entre les événements de cette petite histoire à l'aide d'un réseau bayésien à quatre nœuds :

- I=Route-Gelée
- H=Accident-Holmes
- W=Accident-Watson
- S=Professeur-Sauvé

Question 2 (Inférence déductive). Si l'on connaît la probabilité de l'événement I, $P(I)$:

- Comment peut-on calculer les probabilités $P(H)$ et $P(W)$ des événements H et W ? De quelles informations faut-il disposer quant aux relations entre I, H et W ?
- Comment peut-on calculer la probabilité $P(S)$ de l'événement S ? Quelles informations nous faut-il sur les relations entre I, H, W et S ?

Question 3 (Inférence abductive). Comment peut-on exprimer la probabilité de l'événement I lorsqu'on sait si l'événement W s'est produit ou pas ($W = 1$ ou $W = 0$) ? Si l'on connaît $P(I)$, quelles informations nous faut-il sur la relation entre I et W pour calculer cette probabilité ?

Question 4 (Déduction et abduction). Si on connaît $P(I)$:

- Comment peut-on exprimer la probabilité de l'événement H lorsqu'on sait si l'événement W s'est produit ou pas ? Quelles informations nous faut-il sur les relations entre I, W et H pour calculer cette probabilité ?
- Comment peut-on exprimer la probabilité de H lorsqu'on sait si les événements W et S se sont produits ou pas ? Quelles informations nous faut-il sur les relations entre I, H, W et S pour calculer cette probabilité ?
- Comment peut-on exprimer la probabilité de H lorsqu'on sait si les événements I, W et S se sont produits ou pas ? Quelles informations nous faut-il sur les relations entre I, H, W et S pour calculer cette probabilité ?

Calcul probabiliste

Ajoutons maintenant les informations sur les relations de ce réseau. Sachant qu'il est probable que la route soit gelée et que Holmes et Watson soient de mauvais conducteurs, nous considérerons que :

$P(I = 1) = 0.7$
$P(H = 1|I = 1) = 0.9, P(W = 1|I = 1) = 0.7$
$P(H = 1|I = 0) = 0.1, P(W = 1|I = 0) = 0.5$

Ensuite, sachant que la présence de Holmes est essentielle pour la résolution d'une affaire, nous considérerons que :

$P(S = 1|H = 1, W = 1) = 0.1$
$P(S = 1|H = 1, W = 0) = 0.2$
$P(S = 1|H = 0, W = 1) = 0.8$
$P(S = 1|H = 0, W = 0) = 1$

Question 5. Complétez les tableaux des probabilités conditionnelles $P(H|I)$ et $P(W|I)$.

$P(H\|I)$	$I = 1$	$I = 0$	$P(W\|I)$	$I = 1$	$I = 0$
$H = 1$			$W = 1$		
$H = 0$			$W = 0$		

Question 6. À partir de ces tableaux, calculez les probabilités que Holmes et Watson aient eu un accident, $P(H = 1)$ et $P(W = 1)$.

Question 7. Dès que Miss Lovelace l'a informé que Watson a eu un accident, Smith déduit que les routes sont probablement gelées. Calculez la probabilité $P(I = 1|W = 1)$.

Question 8 (Dépendance). Ensuite, il en déduit que Holmes a probablement eu un accident aussi. Quelle est la probabilité $P(H = 1|W = 1)$? Comparez-la avec $P(H = 1)$: H et W sont deux événements dépendants.

Question 9 (Indépendance conditionnelle). Dès que Miss Lovelace l'informe que les routes ne sont pas gelées, Smith revient sur ses conclusions. Quelle est la probabilité $P(H = 1|W = 1, I = 0)$? Comparez-la avec $P(H = 1|I = 0)$: H et W sont deux événements indépendants, étant donné I.

Question 10. Complétez le tableau de la probabilité conditionnelle $P(S|W, H)$:

$P(S\|W, H)$	$H = 1$, $W = 1$	$H = 1$, $W = 0$	$H = 0$, $W = 1$	$H = 0$, $W = 0$
$S = 1$				
$S = 0$				

Question 11 (Causes multiples). À partir de ce tableau, calculez la probabilité que le professeur ait été sauvé, $P(S = 1)$.

Question 12. En découvrant l'accident de son mari et le dénouement de l'affaire de la Priory School, Mary conclut que Holmes a évité l'accident. Quelle est la probabilité $P(H = 1|W = 1, S = 1)$?

Question 13 (Abduction avec plusieurs conséquences). Ensuite, elle déduit que les routes n'étaient pas gelées. Quelle est la probabilité $P(I = 1|W = 1, S = 1)$?

Question 14. Finalement, elle découvre l'état des routes et déduit que Holmes n'a certainement pas eu un accident. Quelle est la probabilité $P(H = 1|I = 0, W = 1, S = 1)$?

Question 15 (Dépendance conditionnelle). Et si Mary n'était pas au courant de l'accident de Watson ? Quelle est la probabilité $P(H = 1|I = 0, S = 1)$? Comparez-la avec $P(H = 1|I = 0, W = 1, S = 1)$: H et W sont deux événements dépendants, étant donnés I et S.

Question 16. Finalement, quelle cause pourrait expliquer l'accident de Watson ? Peut-être ses pneus sont-ils trop vieux ? Modifiez le réseau en ajoutant le nœud V=Vieux-Pneus-Watson.

Question 17. Comment peut-on exprimer la probabilité de l'événement V dès lors qu'on sait si les événements I et W se sont produits ou pas ? Si l'on connaît $P(V)$, de quelles informations faut-il disposer sur les relations entre I, W et V pour calculer cette probabilité ?

Solutions à la page 356

Exercice 6.2 Deuxième partie - Facteurs de Certitude

Dans cet exercice, nous allons modifier notre moteur d'inférence à chaînage avant de façon à intégrer la notion d'incertitude dans les faits et les règles. Les modules ci-dessous représentent le squelette du programme que nous allons développer.

Notez que le code de cette série s'appuie sur des modules développés dans les chapitres précédents. Nous ne les reproduisons pas ici, mais il est nécessaire de pouvoir les importer. Veillez à organiser vos dossiers en conséquence.

Module `.../moteur_avec_variables_fc/facteurs_certitude.py` :

```
def fc_ou(fc1, fc2):
    print('à compléter')

def fc_et(fc1, fc2):
    print('à compléter')
```

Module `.../moteur_avec_variables_fc/regle_avec_variables_fc.py` :

```
from .facteurs_certitude import fc_et

class RegleAvecVariables_FC:
    def __init__(self, conditions, conclusion, fc=1.0):
        print('à compléter')
```

```
    def depend_de(self, fait , methode):
        envs = {}

        for condition in self .conditions:
            # Si au moins une des conditions retourne un environnement,
            # nous savons que la proposition satisfait une des conditions.
            env = methode.pattern_match(fait, condition, {})
            if env != methode.echec:
                envs[condition] = env

        return envs

    def satisfaite_par ( self , faits , cond, env, env_fc , methode):
        print('à compléter')

    def __repr__ ( self ):
        return '{} => {}, {}'.format(str(self.conditions),
                                     str( self .conclusion ),
                                     str( self . fc ))
```

Module `.../moteur_avec_variables_fc/connaissance_fc.py` :

```
from .facteurs_certitude import fc_ou
from moteur_avec_variables_fc. regle_avec_variables_fc import RegleAvecVariables_FC

class BaseConnaissances_FC:
    def __init__ ( self ):
        self . faits = {}
        self . regles = []

    def ajoute_un_fait ( self , fait ):
        print('à compléter')

    def ajoute_faits ( self , faits ):
        for fait in faits :
            self . ajoute_un_fait ( fait )

    def ajoute_une_regle( self , description ):
        print('à compléter')

    def ajoute_regles ( self , descriptions ):
        for description in descriptions :
            self . ajoute_une_regle( description )
```

Module `.../moteur_avec_variables_fc/chainage_avant_avec_variables_fc.py` :

```
from moteur_sans_variables.chainage import Chainage
from moteur_avec_variables. filtre import Filtre

class ChainageAvantAvecVariables_FC(Chainage):
    def __init__ ( self , connaissances, methode=None):
        Chainage. __init__ ( self , connaissances)

        if methode is None:
            self .methode = Filtre()
        else:
            self .methode = methode
```

```
    def instancie_conclusion(self, regle, envs_et_fcs):
        print('à compléter')

    def chaine(self):
        print('à compléter')
```

Module `.../exemple_animaux.py` :

```
from sys import argv, exit
from moteur_avec_variables_fc.regle_avec_variables_fc import RegleAvecVariables_FC
from moteur_avec_variables_fc.connaissance_fc import BaseConnaissances_FC
from moteur_avec_variables.unificateur import Unificateur
from moteur_avec_variables.filtre import Filtre
from moteur_avec_variables_fc.chainage_avant_avec_variables_fc import
    ChainageAvantAvecVariables_FC

regles = [
    ([('placentaire', '?x')], ('mammifère', '?x')),
    ([('marsupial', '?x')], ('mammifère', '?x')),
    ([('monotrème', '?x')], ('mammifère', '?x')),
    ([('placentaire-1', '?x')], ('placentaire', '?x')),
    ([('genre-placentaire', '?x')], ('placentaire', '?x')),
    ([('a-des-poils', '?x'), ('a-des-bébés-formes', '?x'),
      ('température-stable', '?x')],
     ('placentaire-1', '?x'), 9.0/10.0),
    ([('a-des-poils', '?x'), ('a-des-bébés-foetaux', '?x'),
      ('température-stable', '?x')],
     ('marsupial', '?x'), 95.0/100.0),
    ([('a-des-oeufs', '?x'), ('a-des-poils', '?x'),
      ('température-stable', '?x')],
     ('monotrème', '?x')),
    ([('singe', '?x')], ('genre-placentaire', '?x')),
    ([('primate', '?x')], ('singe', '?x')),
    ([('lémurien', '?x')], ('singe', '?x')),
    ([('chimpanzé', '?x')], ('primate', '?x')),
    ([('gorille', '?x')], ('primate', '?x')),
    ([('canidé', '?x')], ('genre-placentaire', '?x')),
    ([('chien', '?x')], ('canidé', '?x')),
    ([('loup', '?x')], ('canidé', '?x')),
    ([('lycaon', '?x')], ('canidé', '?x')),
    ([('félin', '?x')], ('genre-placentaire', '?x')),
    ([('chat', '?x')], ('félin', '?x')),
    ([('lion', '?x')], ('félin', '?x')),
    ([('tigre', '?x')], ('félin', '?x')),
    ([('tigre-du-bengale', '?x')], ('tigre', '?x')),
    ([('tigre-de-l-himalaya', '?x')], ('tigre', '?x')),
    ]

if len(argv) < 2 or argv[1].lower() not in ('a', 'b'):
    print('On attend au moins un arguments: A ou B')
    exit(1)

if argv[1].lower() == 'a':
    faits_initiaux = [
        (('a-des-poils', 'blaireau'),),
        (('a-des-bébés-formes', 'blaireau'),),
        (('température-stable', 'blaireau'),),
```

```
        (('a-des-poils', 'écureuil' ),),
        (('a-des-bébés-formes', 'écureuil'),),
        (('température-stable', 'écureuil' ),),
        (('chimpanzé', 'cheetah' ),),
        (('gorille', 'bozo' ),),
        (('singe', 'babouin'),),
        (('singe', 'paresseux' ),),
        (('chien', 'bill' ),),
        (('loup', 'loup-1'),),
        (('lycaon', 'lycaon-1'),),
        (('chat', 'mistigri' ),),
        (('lion', 'minet' ),),
        (('tigre-du-bengale', 'tigre-du-bengale-1'),),
        (('a-des-poils', 'kangourou'),),
        (('a-des-bébés-foetaux', 'kangourou'),),
        (('température-stable', 'kangourou'),),
        (('a-des-oeufs', 'ornythorinque' ),),
        (('a-des-oeufs', 'nouveau-spécimen'), 5.0/10.0),
        (('a-des-poils', 'ornythorinque' ),),
        (('température-stable', 'nouveau-spécimen'), 9.0/10.0),
        (('température-stable', 'ornythorinque' ),),
    ]
elif argv[1].lower() == 'b':
    faits_initiaux = [
        (('a-des-oeufs', 'nouveau-spécimen'), 5.0/10.0),
        (('température-stable', 'nouveau-spécimen'), 9.0/10.0),
        (('a-des-poils', 'nouveau-spécimen'), 9.0/10.0),
        (('a-des-poils', 'écureuil' ),),
        (('a-des-bébé-formes', 'écureuil'),),
        (('température-stable', 'écureuil' ),),
        (('a-des-poils', 'un-spécimen-qui-ressemble-à-un-écureuil'), 0.9),
        (('a-des-bébé-formes', 'un-spécimen-qui-ressemble-à-un-écureuil'), 0.7),
        (('température-stable', 'un-spécimen-qui-ressemble-à-un-écureuil'), 0.8),
    ]

bc = BaseConnaissances_FC()
bc.ajoute_faits( faits_initiaux )
bc.ajoute_regles( regles )

moteur = ChainageAvantAvecVariables_FC(connaissances=bc, methode=Filtre())
moteur.chaine()

moteur.affiche_solutions ()

if len(argv) > 2 and argv[2].lower() == 'trace':
    moteur.affiche_trace ()
```

Exercice 6.2.1 Manipulation des facteurs de certitude

Dans cet exercice, nous aurons besoin de pouvoir combiner les facteurs de certitude de faits entrant dans une relation logique les uns avec les autres. Commencez donc par compléter la fonction `fc_et` de `facteurs_certitude.py`, qui prend comme paramètres deux facteurs de certitude devant exister conjointement et qui retourne le minimum des deux. Cette fonction sera utilisée par la méthode `RegleAvecVariables_FC.satisfaite_par`, que nous verrons ci-dessous, pour calculer le facteur de certitude associé à un ensemble de conditions.

Dans le même module, codez aussi une fonction `fc_ou` qui prend comme paramètres les facteurs de certitude de deux faits identiques par leur contenu (même proposition) et qui retourne le facteur de certitude résultant. Cette fonction sera utilisée par la méthode `BaseConnaissances_FC.ajoute_un_fait` pour réaliser la mise à jour des facteurs de certitude.

Exercice 6.2.2 Le faits et les règles
Des facteurs de certitude doivent être associés aux faits et aux règles. Le langage Python ne permettant pas d'utiliser des structures au sens des `struct` du C, nous ferons usage de `tuples` de la forme `(fait, fc)`, où `fait` est le fait que l'on veut définir et `fc` un facteur de certitude. Nous aurons ainsi par exemple :

```
fait = (('père', 'Jean', 'Paul'), 9/10)
```

D'une façon analogue, la classe `RegleAvecVariables_FC` implémentera une règle avec trois attributs :

- `conditions` : la liste des conditions de la règle ;
- `conséquence` : la conséquence de la règle ;
- `fc` : le facteur de certitude associé à la règle.

Nous aurons ainsi par exemple :

```
regle = RegleAvecVariables_FC([('père', '?x', '?z'), ('père', '?z', '?y')],
                              ('grand−père', '?x', '?y'),
                              8/10)
```

`RegleAvecVariables_FC` s'inspire donc de la classe `RegleAvecVariables` du moteur de chaînage avant en lui ajoutant un attribut `self.fc`, qui contiendra la valeur du facteur de certitude associé. Le constructeur de la règle, que vous devez compléter, devra prendre trois arguments, en accord avec la définition ci-dessus. Pour simplifier la création de nouvelles règles, vous pouvez stipuler que l'argument correspondant au facteur de certitude soit optionnel, avec une valeur par défaut de 1.0.

La méthode `RegleAvecVariables.satisfaite_par` *:* La méthode `depend_de` de la classe `RegleAvecVariables` peut être reprise telle quelle. Il faut en revanche modifier la méthode `satisfaite_par` de sorte qu'elle prenne comme arguments :

- La liste des faits déjà connus ;
- La condition testée avec succès par `depend_de` ;
- L'environnement résultant de l'appel à `depend_de` ;
- Le facteur de certitude associé au fait qui a déclenché la règle ;
- La classe de pattern matching (filtre ou unificateur) utilisée.

La valeur de retour doit être une liste de pairs d'environnements, accompagnés chacun par le facteur de certitude associé. Le facteur de certitude d'un nouvel environnement est donné par le minimum des facteurs de cet environnement et du fait passé en argument à l'appel de la fonction `pattern_match` qui a conduit à la découverte du nouvel environnement.

Prenons par exemple (par simplification, les faits et règles ne sont pas écrits selon le format interne) :

```
règle = ([('vole', '?x', '?y'), ('est−découvert', '?x')],
         ('en−prison', '?x', 'pour−vol−de', '?y'), 8/10)

faits = [
         (('vole', 'Jean', 'bijoux'), 9/10),
         (('vole', 'Paul', 'voiture'), 8/10),
         (('est−découvert', 'Paul'), 6/10),
         (('est−découvert', 'Jean'), 8/10))
         ]

# Fait déclencheur :
fait_declencheur = (('est−découvert', 'Jean'), 8/10)

# Paires de conditions et environnements retournées par règle.depend_de :
envs = {('est−découvert', '?x') : {'?x': 'Jean'}}

# L'essai de la règle par le moteur àchaînage avant doit provoquer l'appel
# suivant de règle. satisfaite_par :
satisfaite_par (faits, ('est−découvert', `?x'), {'?x': 'Jean'}, 8/10))
    −> [({'?x': 'Jean', '?y': 'bijoux'}, 8/10)]
```

La base de connaissances : La base de connaissances `BaseConnaissances_FC` doit aussi être modifiée en adaptant les méthodes `ajoute_un_fait` et `ajoute_une_regle` aux nouvelles définitions des faits et des règles :

- `ajoute_un_fait(('père', 'Jean', 'Paul'), 9/10))` ajoute le fait `('père', 'Jean', 'Paul')` à la base de connaissances avec un facteur de certitude de 9/10 ;
- `ajoute_un_fait(('père', 'Paul', 'Marc'))` ajoute le fait `('père', 'Paul', 'Marc')` à la base de connaissances avec une facteur de certitude de 1 (valeur par défaut) ;
- `ajoute_une_regle(([('père', '?x', '?z'), ('père', '?z', '?y')], ('grand-père', '?x', '?y'), 8/10))` ajoute une règle à la base de connaissances avec un facteur de certitude de 8/10 ;
- `ajoute_une_regle(([('père', '?x', '?z'), ('père', '?z', '?y')], ('grand-père', '?x', '?y')))` ajoute une règle à la base de connaissances avec un facteur de certitude de 1.

La fonction `ajoute_un_fait` continue de jouer le même rôle que par le passé : elle ajoute un fait à la base des faits si celui-ci n'est pas déjà connu. Cependant, elle doit être modifiée pour tenir compte des facteurs de certitude. En effet, si le fait à ajouter n'est pas nouveau (il a donc été démontré auparavant par un chemin différent), il faut mettre à jour le facteur de certitude associé pour refléter la nouvelle valeur de certitude. Utilisez la méthode `fc_ou` dans ce cas.

Exercice 6.2.3 Le moteur d'inférence à chaînage avant avec variables

La méthode `ChainageAvantAvecVariables.instancie_conclusion` *:* Comme auparavant, la méthode `instancie_conclusion` de la classe `ChainageAvantAvecVariables_FC` doit instancier la conséquence d'une règle au moyen d'une liste d'environnements. Cependant, les nouveaux faits doivent maintenant se voir affecter un facteur de certitude, qui dépendra du facteur de certitude de la règle et de celui qui résulte des conditions. Elle doit donc prendre en paramètre une liste de paire d'environnements associés à leurs facteurs de certitude, et retourner les instanciations de la conclusion, munies chacune de leur facteur de certitude.

Prenez bien en compte la formule de calcul du facteur de certitude résultant de l'application d'une règle. Exemple :

```
règle = ([('vole', '?x', '?y'), ('est-découvert', '?x')],
         ('en-prison', '?x', 'pour-vol-de', '?y'), 8/10)

instancie_conclusion(règle,
                     [({'?x' : 'Jean', '?y' : 'bijoux'}, 8/10),
                      ({'?x' : 'Paul', '?y' : 'voiture'}, 6/10)])
  -> [(('en-prison', 'Jean', 'pour-vol-de', 'bijoux'), 16/25),
      (('en-prison', 'Paul', 'pour-vol-de', 'voiture'), 12/25)]
```

La méthode `ChainageAvantAvecVariables_FC.chaine` *:*Les faits dont le facteur de certitude est négatif ne doivent pas être utilisés comme déclencheurs, même s'ils sont consignés dans la base des faits. Implémentez donc la méthode de chaînage avant `chaine` de `ChainageAvantAvecVariables_FC` en tenant compte des modifications apportées aux autres fonctions et en vous inspirant de l'algorithme suivant :

```
ChainageAvantAvecVariables_FC(faits_depart, regles)
1.  solutions <- liste vide
2.  Q <- faits_depart
3.  WHILE Q n'est pas vide DO
4.      q <- premier(Q)
5.      Q <- reste(Q)
6.      IF q n'est pas dans solutions THEN
7.          ajouter q à solutions
8.          IF le facteur de certitude fc de q est plus grand que 0 THEN
9.              FOR EACH règle r de regles DO
10.                 envs <- toute les paires de conditions et environnements issues
                    du pattern matching réussi entre q et les conditions de r
11.                 FOR chaque condition cond et environnement env de envs DO
12.                     envs1 <- toutes les paires environnements et fcs établis
                        par le pattern matching des conditions restantes
                        de r étant donné env
13.                     FOR chaque environnement env1 et facteur fc1 de envs1 DO
14.                         instances <- instanciation de la conclusion de r selon
                            env1 et fc1
15.                         ajouter instances en queue de Q
16.                     END FOR
17.                 END FOR
18.             END FOR
19.         END IF
20.     END IF
```

21. END WHILE
22. RETURN solutions
END ChainageAvantAvecVariables_FC

Test du programe

Le module `exemple_animaux.py` contient des règles et des faits modélisant une petite partie d'un arbre de classification des mammifères. Après avoir écrit votre programme, testez-le sur le premier exemple en ajoutant l'option `A`. Vous pouvez afficher la trace en utilisant l'option `trace`.

```
python3 exemple_animaux.py A
python3 exemple_animaux.py A trace
```

Le module contient un deuxième exemple que vous pouvez exécuter avec la commande :

```
python3 exemple_animaux.py B
python3 exemple_animaux.py B trace
```

Solutions à la page 361

Deuxième partie

Le raisonnement basé sur modèles

L'informatique classique procède selon le principe de la *déduction* : des entrées bien définies sont transformées en un ou plusieurs résultats. La transformation elle-même s'effectue de manière indépendante du contexte ; un programme fonctionne de la même manière dans toutes les situations.

En Intelligence Artificielle, on considère en plus la possibilité qu'un programme déductif puisse être *appris* en observant un grand nombre de paires entrées-résultats. Par exemple, on peut programmer une voiture autonome par un apprentissage à partir d'observations portant sur le comportement d'un conducteur humain.

Cependant, le résultat souhaité dépendra souvent du contexte et des objectifs ; souvent aussi, on ne possède pas d'expérience antérieure qui puisse servir de modèle. Une voiture autonome peut ainsi se retrouver rentrer dans des situations pour lesquelles aucune expérience n'est disponible, par exemple sur des chantiers ou lors d'accidents de la route.

Dans de tels cas, il faut *chercher* une nouvelle solution en utilisant comme *modèle* les connaissances déductives programmées ou apprises dans des situations analogues et plus fréquentes. Il s'agit d'*imaginer* des situations hypothétiques dont on ne possède aucune experience préalable et d'en tirer la meilleure solution du point de vue des objectifs que l'on s'est donnés. Cette capacité d'imagination est une des caractéristiques qui expliquent la puissance de l'intelligence humaine.

En termes plus formels, on cherche alors un ensemble de paramètres qui, associés à un modèle, rendront possibles des conclusions qui répondent aux exigences du problème. Le raisonnement déductif part d'un modèle et de paramètres pour trouver des conclusions :

$$\text{modèle} \wedge \text{paramètres} \vdash \text{conclusions}$$

Dans le cas qui nous intéresse, nous disposons d'un modèle et de conclusions et nous cherchons les paramètres qui rendront valide la dérivation des conclusions :

$$\text{modèle} \wedge \text{conclusions} \vdash \text{paramètres}$$

Par exemple,

- Dans une tâche de diagnostic portant sur un dispositif en panne, on possède un modèle du fonctionnement du dispositif et, comme conclusions, les observations de son comportement actuel. On cherche comme paramètres les composantes défectueuses qui expliquent ce comportement.
- Dans un problème de planification, le modèle consiste en l'ensemble des opérateurs plus la situation de départ. Les conclusions sont les objectifs à atteindre. On cherche comme paramètres une séquence d'opérateurs qui garantisse que les buts seront atteints.

Souvent, ce raisonnement implique aussi une *optimisation* : on cherche le diagnostic le plus probable, ou bien le plan le plus efficace.

Ce type de raisonnement logique s'appelle *l'abduction* ; on parle aussi de *raisonnement basé sur des modèles* ou de *résolution de problèmes*. L'abduction n'est correcte que si on fait l'hypothèse d'un *monde clos*, c'est-à-dire si toutes les possibilités d'obtenir les conclusions sont supposées connues. Par exemple,

si on découvre un nouveau type de défaut d'un dispositif, un diagnostic qui n'a pas pris en compte cette possibilité ne sera plus correct. Toutes les méthodes pour résoudre un problème abductif font cette hypothèse du monde clos, et il existe des différences importantes entre les méthodes quant à la façon de la mettre en œuvre.

Dans le *raisonnement basé sur des modèles*, qui donne son titre à cette partie, l'hypothèse d'un monde clos n'est prise en compte qu'au moment où le problème est résolu. L'abduction se fait donc directement sur la base du *modèle*, qui pourrait également s'utiliser de façon déductive. Cela veut dire que le modèle peut changer entre différentes applications du système sans qu'une reprogrammation soit nécessaire. Le prix à payer pour cette avantage est cependant que l'abduction exige toujours une *recherche* entre les différentes valeurs possibles des paramètres. Les principaux algorithmes d'abduction reposent donc sur des algorithmes de recherche.

Dans cette deuxième partie, nous passerons d'abord en revue des algorithmes de recherche généraux et ensuite des algorithmes adaptés à une classe plus restreinte de problèmes abductifs, celle des problèmes de satisfaction de contraintes. Enfin, nous examinerons deux applications, le diagnostic et la planification. Le diagnostic est le plus simple car dans ce cas, l'espace des solutions possibles est clos : il se limite aux combinaisons des différentes composantes. Par contre, la planification est un problème ouvert, car il n'y a aucune limite aux nombre d'opérations requises.

Chapitre 7

Résolution de problèmes par recherche

Comme nous l'avons déjà souligné dans l'introduction, les systèmes basés sur la connaissance utilisent des méthodes de résolution différentes de celles proposées par l'algorithmique classique. Ces méthodes s'appliquent aussi bien à des problèmes admettant plusieurs solutions qu'à ceux qui n'en ont aucune. Dans les systèmes algorithmiques par contre, pour tout problème, c'est une solution unique qui est atteinte par un traitement direct. Les systèmes basés sur la connaissance atteignent généralement un but en *cherchant* une solution satisfaisante dans un espace d'alternatives. La recherche peut retourner une, plusieurs ou même aucune solution.

Un processus de recherche est constitué de deux parties : un *générateur* de solutions et un *évaluateur* du progrès effectué. Le moteur à chaînage-avant, vu précédemment, constitue l'exemple le plus simple de systèmes de recherche. Il génère les solutions possibles en appliquant à la base de données la règle d'inférence du *modus ponens*. L'évaluateur, pour sa part, contrôle si le but désiré est atteint ou pas. Ce processus présente en fait deux inconvénients :

- Le mécanisme à chaînage-avant est incapable de discerner des alternatives mutuellement exclusives : tous les éléments de la base de faits se doivent d'être simultanément valides. Cela implique également qu'il est impossible de formuler des règles disjonctives de la forme : `C` ⇒ `A ou B`.
- Les états intermédiaires sur le chemin menant à la solution ne sont pas évalués. La recherche est donc complètement aveugle jusqu'au moment où la solution est effectivement atteinte.

Dans ce qui suit, nous verrons par quelles extensions il est possible d'éliminer ces deux problèmes. Le premier problème peut être contourné par la construction d'un *graphe de recherche* explicite des différents *environnements* d'alternatives, ce qui a déjà été utilisé dans l'algorithme de chaînage arrière. La solution au second problème consiste à utiliser des heuristiques d'évaluation permettant de décider quelle partie du graphe il faut explorer.

7.1 Arbres et graphes de recherche

L'espace exploré par un processus de recherche est en fait un arbre constitué de *nœuds* et d'*arcs*. Chaque nœud représente une étape d'inférence : un environnement de buts dans un système à chaînage arrière, ou bien une solution

abstraite ou partielle. Les arcs correspondent aux applications des règles qui transforment une solution partielle en une autre.

L'arbre de recherche décrit par la figure 7.1 montre une partie de l'espace exploré en vue de résoudre le problème des quatre reines. Le but de ce problème est de placer les reines sur un échiquier de seize cases de telle sorte qu'elles ne soient pas en conflit, selon les règles du jeu d'échec (deux reines ne peuvent occuper la même ligne, la même colonne ou la même diagonale). La recherche de la figure 7.1 commence par une configuration alignant les 4 reines sur la première ligne. Les règles sont des transformations d'équivalence déplaçant une reine jusqu'à une position voisine au sein de sa colonne. Pour générer tous les successeurs d'un nœud dans l'arbre, il suffit de lui appliquer toutes les règles possibles. Notons qu'une telle recherche ne peut se faire par simple chaînage-avant, car les états résultants sont mutuellement incompatibles (une même reine occupe plusieurs positions en même temps).

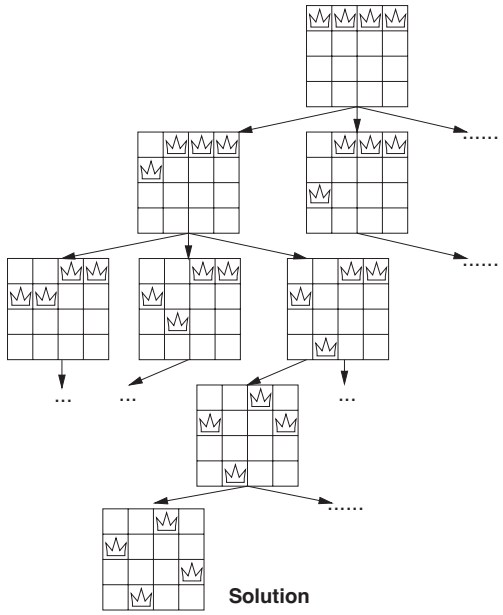


Fig. 7.1 *Partie d'un arbre de recherche pour le problème des 4 reines.*

Si deux règles s'appliquent dans un ordre différent à un nœud de l'arbre, la situation résultante est identique. L'existence potentielle de tels cycles signifie que l'espace de recherche n'est pas vraiment un arbre, mais plutôt un *graphe* d'alternatives. Une plus grande économie peut être réalisée par détection explicite des cycles dans le graphe de recherche, ce qui évite d'explorer plusieurs fois des nœuds identiques.

Le graphe de recherche peut être représenté implicitement par une fonction $succ(n)$ qui retourne une liste des *successeurs* liés par un arc au nœud n. En général, la fonction *succ* correspond à l'application de toutes les règles d'inférence au nœud n.

Un algorithme de recherche commence toujours avec un ou plusieurs *nœuds initiaux*. Il se termine avec la génération d'un *nœud final* qui remplit une *condition de terminaison* qui l'identifie comme étant une solution au problème. En général, la recherche s'arrête dès qu'une solution est trouvée, et un algorithme est *optimal* s'il trouve une solution dans un temps minimal.

La *solution* au problème peut être donnée par :

- la description du nœud final : par exemple un placement des quatre reines ;
- le *chemin* qui mène d'un nœud initial au nœud final : par exemple, dans un problème de planification où l'on désire connaître la séquence d'opérations à effectuer.

Dans le deuxième cas, il existe un deuxième critère d'optimalité, celui du *coût* total des opérations sur le chemin trouvé. On n'admet alors que des algorithmes qui trouvent effectivement la solution optimale au problème.

7.2 Algorithmes de recherche en profondeur-d'abord (DFS) et en largeur-d'abord (BFS)

Il existe deux techniques extrêmes d'exploration d'un arbre de recherche : la recherche en profondeur-d'abord (*Depth First Search*, ou DFS) et la recherche en largeur-d'abord (*Breadth First Search*, ou BFS). La première tente d'atteindre la solution le plus vite possible en explorant immédiatement les successeurs de tout nœud généré, alors que la seconde étend l'arbre en générant les nœuds couche par couche.

L'algorithme de recherche en profondeur-d'abord est décrit par la figure 7.2, et la figure 7.3 met en évidence son fonctionnement sur un exemple.

À chaque étape, l'algorithme met à jour la file des nœuds non explorés. C'est toujours le premier nœud de la file qui est étendu. Les nœuds résultants sont ajoutés en *tête* de la file de sorte qu'ils soient explorés en premier dans les étapes ultérieures. Dans l'exemple de la figure 7.3, l'algorithme explore donc

```
1: Function Recherche-DFS (Noeud-initial)
2: Q ← (Noeud-initial)
3: repeat
4:     n ← first(Q), Q ← rest(Q)
5:     if n est un noeud but, return n
6:     S ← succ(n)
7:     Q ← append(S, Q)
8: until Q est vide
9: return ECHEC
```

Fig. 7.2 *Algorithme de recherche en profondeur-d'abord.*

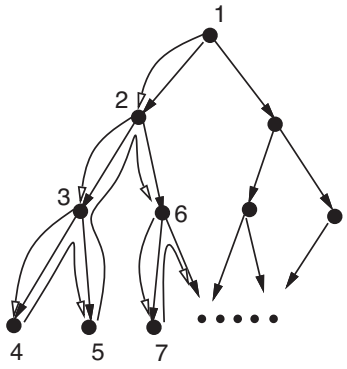


Fig. 7.3 *Exemple d'une recherche en profondeur-d'abord. La numérotation indique la séquence d'exploration des nœuds.*

tout d'abord les nœuds 1 à 4. Lorsqu'il n'existe aucun successeur, l'algorithme poursuit son exploration en étendant le prochain nœud non encore visité. En d'autres termes, il effectue un *retour-arrière* au niveau précédent. Les passages des nœuds 4 à 5 et 5 à 6 sont des exemples de tels retours-arrière.

Les avantages de la stratégie profondeur-d'abord sont la simplicité de son implémentation et le fait que l'algorithme ne requiert que très peu de mémoire : uniquement le chemin entre le nœud initial et le nœud courant ainsi que les alternatives non explorées sur ce chemin doivent être mémorisés.

L'un des problèmes majeurs de cette méthode de recherche se pose lorsque le graphe contient des cycles. Cette possibilité est donnée lorsque les opérateurs permettent des opérations réversibles. Par exemple, s'il était possible de revenir sur le placement d'une reine après l'avoir posée, l'arbre de recherche pourrait contenir, entre autres, le cycle décrit par la figure 7.4. L'algorithme peut alors boucler à l'infini sans trouver de solutions. Il est toutefois possible de remédier à cette situation en détectant les cycles et en évitant d'étendre des nœuds déjà visités lors d'une étape précédente. Cela risque cependant de faire exploser le temps de calcul, car chaque nœud doit être comparé à tous ceux qui l'ont précédé.

Une alternative à la recherche en profondeur-d'abord est donnée par la méthode de recherche en largeur-d'abord, dont l'algorithme est décrit par la figure 7.5, et la figure 7.6 donne un exemple de fonctionnement.

Du point de vue algorithmique, la seule différence entre les recherches en largeur et en profondeur-d'abord réside dans le fait que les successeurs d'un nœud sont placés en *fin* de file au lieu d'être insérés en *tête* (ligne 7 de la figure 7.5). Cette modification implique que l'algorithme ne visite un nœud donné de la couche $(n + 1)$ qu'après avoir exploré tous ceux de la couche n, comme le montre la numérotation de la figure 7.6. Cela signifie que la recherche en largeur-d'abord nécessite beaucoup plus d'espace mémoire que celle en profondeur-d'abord vu que l'ensemble des nœuds d'un niveau donné devient rapidement important. En revanche, l'algorithme en largeur-d'abord trouvera

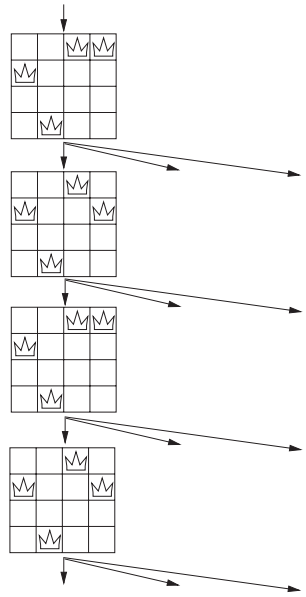


Fig. 7.4 *Une situation où l'algorithme de recherche en profondeur-d'abord entre dans une boucle infinie.*

```
1: Function Recherche-BFS (Noeud-initial)
2: Q ← (Noeud-initial)
3: repeat
4:     n ← first(Q), Q ← rest(Q)
5:     if n est un noeud but, return n
6:     S ← succ(n)
7:     Q ← append(Q, S)
8: until Q est vide
9: return ECHEC
```

Fig. 7.5 *Algorithme de recherche en largeur-d'abord.*

toujours une solution, s'il en existe une, et cette solution sera de surcroît *optimale* : ce sera la solution nécessitant le moins d'applications de règle. Par contre, l'algorithme largeur-d'abord exige beaucoup de mémoire, car tous les nœuds d'une même couche doivent être mémorisés.

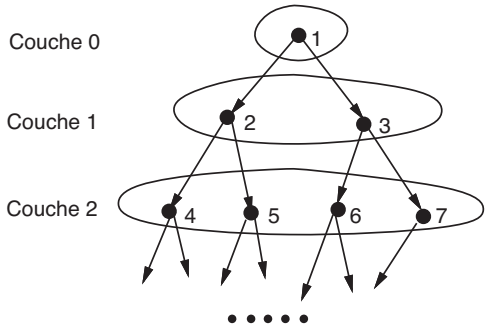


Fig. 7.6 *Exemple d'une recherche en largeur-d'abord. La numérotation indique l'ordre dans lequel les nœuds sont explorés.*

7.3 Recherche en profondeur limitée

La recherche en largeur-d'abord a le défaut qu'elle utilise beaucoup de mémoire. Pour cette raison, elle n'est presque jamais utilisée en pratique, et on applique surtout la recherche en profondeur-d'abord. Celle-ci présente cependant le problème que le chemin trouvé peut être loin d'être optimal. Même si une solution peut être atteinte en générant peu de nœuds, si elle ne se trouve pas sur le premier chemin visité, la recherche peut se perdre en examinant des nœuds d'une grande profondeur sans trouver une solution.

On peut corriger ce problème si on savait déjà d'avance à quelle profondeur l va se situer la solution dans l'arbre de recherche. On peut alors limiter à l la profondeur d'un nœud pour la recherche en profondeur-d'abord à l. Dès qu'un nœud atteint cette profondeur, on ne génère aucun successeur. On obligera ainsi l'algorithme de revenir en arrière et d'examiner également les autres nœuds, un peu comme le ferait la recherche en largeur d'abord. Cela nous donne l'algorithme DLS (*depth-limited search*) qui prend comme paramètre la profondeur maximale l :

```
1: Function DLS (Noeud-initial,l)
2: depth-limit(noeud-initial) ← l
3: Q ← (Noeud-initial)
4: repeat
5:    n ← first(Q), Q ← rest(Q)
6:    if n est un noeud but, return n
7:    S ← succ(n)
8:    for nn ∈ S do
9:       depth-limit(nn) ← depth-limit(n)-1
10:      if depth-limit(nn) ≥ 0 then Q ← append(nn,Q)
11: until Q est vide
12: return ECHEC
```

Malheureusement, en général on ne connaît pas d'avance la bonne valeur de l. Une valeur trop petite ferait que l'algorithme manque la solution, tandis qu'une valeur trop élevée signifierait un coût de recherche excessif.

On peut pallier cet inconvénient par la méthode appelée *iterative deepening*, c'est-à-dire en commençant avec une valeur petite pour l qui sera augmentée si on ne trouve pas de solution :

Function Iterative-deepening(Noeud-initial)
$l \leftarrow 2$
repeat
  $solution \leftarrow DLS(l)$
  $l \leftarrow l + 1$
until $solution \neq \{\}$

Comme cet algorithme répète toujours la recherche de tous les niveaux précédents, on pourrait penser qu'il est peu efficace. Cependant, on peut montrer que la complexité, exprimée en nombre de nœuds de l'espace exploré, ne dépasse pas le double de ce qu'on aurait obtenu si on avait commencé avec la bonne valeur de la limite l. Cela est dû à la croissance exponentielle des nœuds dans chaque couche de l'arbre de recherche. Supposons que chaque nœud a b successeurs. Le nombre de nœuds dans un arbre de profondeur k est alors :

$$c(k) = 1 + b + ... + b^k = \frac{b^{k+1} - 1}{b - 1}$$

Si la première solution se trouve à la profondeur l, l'algorithme a exploré tous les espaces de profondeur l, $l - 1$, ..., 1. Donc la complexité totale est :

$$\begin{aligned}
\sum_{i=1}^{l} c(i) &= \frac{1}{b-1} \sum_{i=1}^{l} (b^{i+1} - 1) \\
&= \frac{b^{l+1}}{b-1} \left[(\sum_{i=0}^{l-1} b^{-i}) - l \right] \\
&\leq \frac{(b^{l+1} - 1)}{(b-1)} \frac{(1 - b^{-l})}{(1 - 1/b)} \\
&< c(l) \cdot 2
\end{aligned}$$

pour autant que $b \geq 2$. Donc, on obtient une méthode qui utilise peu de mémoire, trouve la solution la moins profonde et dont la complexité n'est pas plus que doublée par rapport à une méthode qui connaît la bonne profondeur d'avance. L'algorithme du *iterative deepening* est donc très souvent appliquée en pratique.

7.4 Détection explicite de cycles

Afin d'éviter la répétition de traitements précédemment effectués et de contourner les boucles infinies de la recherche en profondeur-d'abord, il est utile de

pouvoir détecter explicitement si un nœud donné a déjà été visité plus tôt. Cette détection de cycles nécessite la maintenance d'une liste de nœuds déjà explorés. En ajoutant cette liste C à la structure de l'algorithme de recherche en profondeur-d'abord, on obtient l'algorithme modifié de la figure 7.7.

```
1: Function DFS-cycle(Noeud-initial)
2: Q ← (Noeud initial)
3: C ← vide
4: repeat
5:    n ← first(Q), Q ← rest(Q)
6:    if n n'est pas membre de C then
7:       if n est un noeud but, return n
8:       ajouter n à C
9:       S ← succ(n)
10:      Q ← append(S, Q)
11: until Q est vide
12: return ECHEC
```

Fig. 7.7 *Algorithme de recherche en profondeur-d'abord avec détection de cycles.*

Avant d'étendre un nœud donné, l'algorithme de recherche en profondeur-d'abord modifié contrôle qu'il n'a jamais été visité. Si un nœud a fait l'objet d'une précédente exploration, et que, par conséquent, ses successeurs ont déjà été générés, les expansions ultérieures de ce nœud sont simplement abandonnées. Même s'il évite les cycles, ce traitement est sous-optimal. En effet, il se peut très bien que le nouveau chemin emprunté par l'algorithme pour atteindre le nœud soit plus court que le chemin original. Une version plus efficace consisterait à n'abandonner l'exploration du nœud que si son coût est plus élevé que celui d'au moins une de ses précédentes explorations. Nous verrons plus loin comment tenir compte de telles considérations.

7.5 Recherche d'une solution optimale

Souvent, on ne veut pas seulement trouver une solution, mais trouver la solution qui a le moindre *coût*. Un modèle de coût couramment utilisé est de supposer que chaque génération de successeurs rajoute un coût $c(n', n)$. Donc, si n est successeur de n', alors le coût $g(n)$:

$$g(n) = c(n', n) + g(n') = c(n', n) + \sum_{n', n'' \in ancetres(n)} c(n', n'')$$

Ce modèle s'applique par exemple à la planification, en supposant que chaque expansion correspond à une action au plan incomplet, et le coût du plan est égal à la somme des coûts des actions.

Une manière simple de trouver la solution optimale est de générer toutes les solutions possibles et retenir celles qui ont le moindre coût. On peut exploiter le fait que le coût augmente avec la profondeur de recherche pour éliminer des possibilités qui ne peuvent pas être optimales avant de les générer : dès que le coût de la solution partielle dépasse le coût d'une solution complète, celle-ci ne pourra pas faire partie d'une solution optimale car son coût sera forcément plus élevé. Cette observation nous mène à la méthode *branch-and-bound*, qui consiste à modifier l'algorithme DFS pour que chaque fois qu'un nœud est trouvé, l'algorithme mémorise si son coût est meilleur que le meilleur trouvé et ne génère plus de successeurs. On continue alors la recherche au-delà du premier nœud but jusqu'à ce que la liste `OPEN` devienne vide. La figure 7.8 montre l'algorithme résultant.

```
1: Function Recherche-DFS-BB (Noeud-initial)
2: Q ← (Noeud-initial) ; c ← ∞ ; s ← ECHEC
3: repeat
4:    n ← first(Q), Q ← rest(Q)
5:    if n est un noeud but then
6:       if coût(n) < c then c ← coût(n) ; sol ← n
7:    else
8:       S ← succ(n)
9:       for m ∈ S do
10:         if coût(m) < c then Q ← append(m, Q)
11: until Q est vide
12: return sol
```

Fig. 7.8 *Algorithme DFS modifié pour optimisation par branch-and-bound.*

L'algorithme DFS-BB s'avère pourtant toujours très gourmand en temps de calcul, car il génère une grande partie de tous les nœuds de recherche possibles dont beaucoup sont loin de la solution optimale. Il serait bien de mieux guider la recherche vers la meilleure solution par exemple en choisissant des opérateurs de moindre coût d'abord, ou en utilisant les opérateurs qui rapprochent le plus rapidement du but. Ceci est l'idée de la recherche heuristique et notamment de l'algorithme A*, décrit par la figure 7.9.

Le comportement de A* se trouve entre les deux extrêmes de la recherche en profondeur-d'abord et la recherche en largeur-d'abord. La seule différence entre l'algorithme de recherche en profondeur-d'abord et celui de recherche en largeur-d'abord est liée à l'ordre selon lequel les nouveaux nœuds sont ajoutés à la file Q des nœuds à explorer. Pour A*, les nouveaux nœuds ne sont pas simplement rajoutés à Q, mais ils sont fusionnés de telle sorte que Q soit toujours ordonnée par ordre croissant d'une certaine fonction d'évaluation f. La fonction d'évaluation f(n) se calcule en sommant deux facteurs : le coût g(n) de transformation du nœud initial en nœud n, et une estimation *heuristique* h(n) du coût de la transformation de n en un nœud but. La fonction d'évaluation

```
1: Function Recherche-A*(Noeud-initial)
2: Q ← (Noeud initial)
3: C ← vide
4: repeat
5:    n ← first(Q), Q ← rest(Q)
6:    if n ∉ C, ou n ≡ n' ∈ C mais g(n) < g(n') then
7:       if n est un noeud but, return n
8:       ajouter n à C
9:       S ← succ(n)
10:      S ← sort(S,f)
11:      Q ← merge(S, Q, f)
12: until Q est vide
13: return ECHEC
```

Fig. 7.9 *A*, un algorithme de recherche heuristique optimal.*

donne une estimation du coût total d'un chemin menant du nœud initial à un nœud but en passant par le nœud n. En explorant les nœuds par ordre croissant de cette estimation, l'algorithme privilégie les nœuds les plus prometteurs, c'est à dire ceux qui ont le plus de chance d'aboutir à une solution optimale.

Comme exemple d'une recherche heuristique, considérez le graphe de la figure 7.10. La figure montre entre parenthèses l'estimation heuristique de chaque nœud et sur les arcs le coût associé à l'opération.

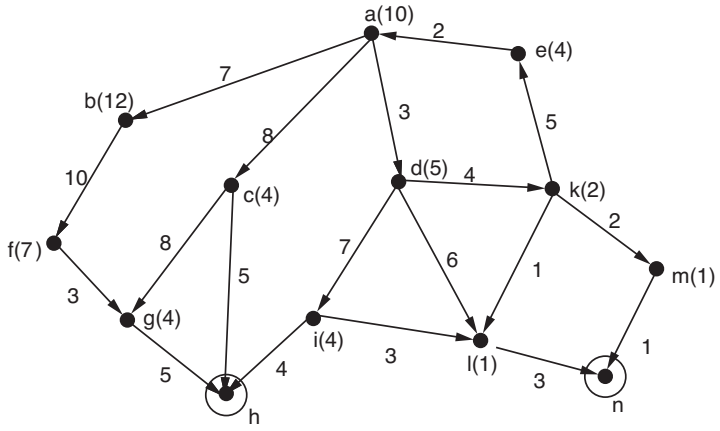


Fig. 7.10 *Exemple d'une recherche heuristique.*

La recherche par A* procédera alors dans comme suit (les valeurs d'évaluation de chaque nœud sont indiquées entre parenthèses) :

```
1. Q = (a(10)) ⇒
2. Q = (d(8),c(12),b(19)) ⇒
3. Q = (k(9),l(10),c(12),i(14),b(19)) ⇒
4. Q = (l(9),m(10),c(12),i(14),e(16),b(19)) ⇒
5. Q = (m(10),n(11),c(12),i(14),e(16),b(19)) ⇒
6. Q = (n(10),c(12),i(14),e(16),b(19)) ⇒
solution !
```

Notons que l'algorithme A* admet qu'on puisse trouver un nouveau chemin à un nœud `n` qui a déjà été visité et se trouve donc sur la liste `C`. Si le nœud `n` n'a pas encore été étendu, il suffit alors de mettre à jour son coût, son chemin et sa position dans Q. Cela se fait à l'étape 3 de la trace ci-dessus, où le chemin `d` → `l` est remplacé par `d` → `k` → `l`, qui est moins coûteux. Si la valeur heuristique du nœud `k` avait été $h(k) = 4$ au lieu de 2, on aurait étendu `l` avant `k`. Dans ce cas, on aurait dû également mettre à jour les évaluations de toutes les conséquences de `l` au moment où l'on trouve le nouveau chemin par `k`.

Une caractéristique importante de A* apparaît quand l'estimation heuristique $h(n)$ *sous-estime* toujours le coût de transformation de n en un nœud but. Dans ce cas, on peut prouver que le premier nœud but trouvé par l'algorithme sera toujours la solution *optimale* au problème. La recherche peut donc s'arrêter, puisque l'on sait qu'aucune solution meilleure ne peut être trouvée lors d'explorations ultérieures.

Si la fonction $h(n)$ remplit en plus la restriction de monotonicité :

$$|h(n_1) - h(n_2)| \leq c(n_1, n_2)$$

où $c(n_1, n_2)$ est le coût de l'arc qui lie n_1 et n_2, alors on peut garantir que l'algorithme découvre chaque nœud par le chemin optimal, c'est-à-dire que les chemins trouvés par l'algorithme ne doivent jamais être révisés.

Le comportement de la recherche dépend fortement de la formulation des fonctions h et c. Plus h se rapproche des coûts réels, plus l'algorithme converge rapidement. En fait, si l'estimation fournie par h est totalement exacte, l'algorithme n'explore aucun nœud inutile. Une bonne façon de trouver une heuristique est de considérer une version simplifiée du problème dont on peut facilement trouver une solution. Par exemple, lors d'une recherche dans un graphe, on peut ignorer la contrainte qu'on peut se déplacer uniquement à travers les arcs du graphe et permettre aussi d'autres mouvements. Dans le problème des reines, on peut ignorer certaines possibilités qu'ils peuvent se capturer. Le coût de la solution du problème simplifié est alors une heuristique qui sous-estime le coût de la solution au vrai problème.

Dans les problèmes où le coût c n'a pas grande importance, on peut choisir $c(n) = 0$. En revanche, l'estimation heuristique du nombre de transformations à effectuer avant d'atteindre le but reste très utile pour accélérer le processus de recherche. Finalement, on peut remarquer que la recherche en largeur-d'abord n'est autre que A* avec $c(n) =$ nombre de transformations déjà effectuées et $h(n) = 0$.

Comme pour la recherche en largeur-d'abord, l'algorithme A^* peut devenir très gourmand en mémoire à cause de la liste Q. On peut alors appliquer des techniques similaires à l'*iterative deepening* :

- dans le *beam search*, on garde seulement les n meilleurs nœuds de Q et on écarte les autres. C'est la manière la plus simple de limiter les besoins en mémoire. Cette méthode ne garantit pas que la solution optimale soit trouvée.
- *Iterative Deepening* A^* : on effectue une recherche en profondeur-d'abord jusqu'à un certain seuil de la fonction d'évaluation. Si aucune solution n'est trouvée, on augmente le seuil par petites incrémentations. La première solution trouvée sera alors optimale.
- *Memory-bounded* A^* : ces techniques sont plus complexes. Elles permettent d'« oublier » et de régénérer des nœuds de Q.

Littérature

Les principaux algorithmes de recherche sont traités dans des introductions à l'algorithmique. L'article [27] donne un résumé des techniques. La méthode de l'*iterative deepening* a été présentée pour la première fois dans [28]. L'algorithme A^* a été introduit dans [29] et l'optimalité de l'algorithme a été discutée dans [30]. La référence [31] présente une version adaptée à un espace mémoire limité.

Application : Routage de véhicules autonomes

Pour transporter des pièces et des matériaux entre les machines qui les traitent, les usines modernes utilisent souvent des véhicules autonomes. Ceux-ci sont habituellement programmés par des règles de comportement qui définissent une façon de traiter les tâches sans collisions. Les règles doivent être développées pour les situations spécifiques par des experts hautement qualifiés. De plus, il est difficile de concevoir des règles qui fonctionnent même en présence d'imprévus comme des pannes de véhicules.
L'entreprise Lookahead Decisions a remplacé un système à règles par une méthode qui cherche les meilleures combinaisons de chemins par recherche heuristique (A*). Par rapport aux comportements fixes, cette méthode a amélioré le débit de 83%, le temps moyen pour réaliser des tâches de 25% et aussi le nombre d'arrêts de véhicules (une mesure de leur usure) de 48%. En plus, le nouveau système résiste mieux aux changements et ne demande pas d'expert pour la mise à jour des règles de comportement.

(Source : Lookahead Decisions Case Study : Real-time routing of automated guided vehicles, `www.lookaheaddecisions.com`, 2003.)

7.6 Exercices

Exercice 7.1 Résolution de problèmes par recherche

Dans cette série d'exercices, vous allez vous familiariser avec les trois principaux algorithmes de recherche :

- en profondeur-d'abord (DFS),
- en largeur-d'abord (BFS),
- par A*.

À titre d'exemple, vous les utiliserez pour découvrir un chemin entre deux villes étant donné leurs positions géographiques et les routes qui les connectent.

Modules squelettes

Les modules qui suivent constituent le squelette des programmes que vous allez implémenter. Les deux derniers modules, `exemple_carte_simple.py` et `exemple_carte_suisse.py`, permettront de les tester.

Module `.../moteurs_recherche/element.py` :

```
class Element:
    def __init__(self, nom=''):
        print('à compléter')

    def distance(self, element):
        return 1

    def __eq__(self, autre):
        print('à compléter')

    def __hash__(self):
        return hash(str(self))

    def __repr__(self):
        return '{}'.format(self.nom)
```

Module `.../moteurs_recherche/ville.py` :

```
from math import sqrt
from .element import Element

class Ville(Element):
    def __init__(self, x, y, nom=''):
        Element.__init__(self, nom)
        print('à compléter')

    def distance(self, ville):
        print('à compléter')

    def __eq__(self, autre):
        print('à compléter')
```

```
    def __hash__(self):
        return hash(str(self))

    def __repr__(self):
        return '{}({}, {})'.format(self.nom, self.x, self.y)
```

Module `.../moteurs_recherche/espace.py` :

```
from copy import copy

class Espace:
    def __init__(self, elements=None, arcs=None):
        self.elements = []
        if elements is not None:
            self.elements = sorted(self.elements, key=lambda e: e.nom)

        self.arcs = []
        if arcs is not None:
            self.ajoute_arcs(arcs)

    def ajoute_arcs(self, arcs):
        print('à compléter')

    def trouve_voisins(self, element):
        print('à compléter')

    def __repr__(self):
        rep = ''
        for element in self.elements:
            rep += '{}, '.format(element)
            rep += 'avec voisins: '
            voisins = self.trouve_voisins(element)
            rep += ', '.join(map(str, voisins))
            rep +='\n'

        return rep
```

Module `.../moteurs_recherche/noeud.py` :

```
from math import sqrt

class Noeud:
    def __init__(self, element, parent=None, cout=0, cout_f=0):
        print('à compléter')

    def __repr__(self):
        rep = '<{}, {}, {}>'.format(self.element,
                                    round(self.cout),
                                    round(self.cout_f))
        return rep
```

Module `.../moteurs_recherche/recherche.py` :

```
from .noeud import Noeud

class Recherche:
    echec = 'échec'
```

```
    def __init__(self, espace, optimisee=False):
        self.espace = espace
        self.optimisee = optimisee

    def recherche(self, depart, but):
        # L'heuristique à utiliser (utile uniquement pour A\*).
        self.h = lambda e: e.distance(but)

        noeud_depart = Noeud(depart, None, 0, self.h(depart))
        noeud_but = Noeud(but)

        return self.recherche_chemin(noeud_depart, noeud_but)

    def recherche_chemin(self, noeud_depart, noeud_but):
        print('à compléter')

    def trouve_chemin(self, noeud):
        chemin = []
        while noeud is not None:
            chemin.insert(0, noeud.element)
            noeud = noeud.parent

        return chemin

    def detecte_cycle(self, trace, noeud):
        return noeud.element in trace

    def trouve_successeurs(self, noeud):
        print('à compléter')

    def ajoute_successeurs(self, queue, successeurs):
        # Nous retournons une liste vide pour éviter de déclencher une exception,
        # mais cette méthode doit être surchargée dans les sous-classes.
        return []
```

Module `.../moteurs_recherche/bfs.py` :

```
from .recherche import Recherche
from .noeud import Noeud

class RechercheBFS(Recherche):
    def ajoute_successeurs(self, queue, successeurs):
        print('à compléter')
```

Module `.../moteurs_recherche/dfs.py` :

```
from .recherche import Recherche
from .noeud import Noeud

class RechercheDFS(Recherche):
    def ajoute_successeurs(self, queue, successeurs):
        print('à compléter')
```

Module `.../moteurs_recherche/astar.py` :

```
from moteurs_recherche.recherche import Recherche
from moteurs_recherche.noeud import Noeud

class RechercheAStar(Recherche):
    def detecte_cycle( self , trace , noeud):
        print('à compléter')

    def ajoute_successeurs( self , queue, successeurs ):
        print('à compléter')
```

Module `.../exemple_carte_simple.py` :

```
from moteurs_recherche.element import Element
from moteurs_recherche.ville import Ville
from moteurs_recherche.espace import Espace
from moteurs_recherche.dfs import RechercheDFS
from moteurs_recherche.bfs import RechercheBFS
from moteurs_recherche.astar import RechercheAStar

# Les éléments dans l'espace de recherche (les  villes ).
elements = {
    'A': Ville (0, 16, 'A'),
    'B': Ville (5, 13, 'B'),
    'C': Ville (0, 10, 'C'),
    'D': Ville (5, 8, 'D'),
    'E': Ville (11, 18, 'E'),
    'F': Ville (15, 13, 'F'),
    'G': Ville (29, 18, 'G'),
    'H': Ville (26, 0, 'H'),
    'I': Ville (12, 10, 'I' ),
    'J': Ville (17, 7, 'J' ),
    'K': Ville (11, 3, 'K'),
    'L': Ville (22, 16, 'L'),
    'M': Ville (25, 12, 'M'),
    'N': Ville (24, 6, 'N'),
    'O': Ville (20, 0, 'O'),
    'P': Ville (5, 0, 'P'),
}

# Les arcs liant les éléments (les routes).
arcs = [
    (elements['A'], elements['B']),
    (elements['A'], elements['E']),
    (elements['B'], elements['C']),
    (elements['B'], elements['E']),
    (elements['B'], elements['D']),
    (elements['C'], elements['D']),
    (elements['C'], elements['P']),
    (elements['D'], elements['I' ]),
    (elements['D'], elements['K']),
    (elements['E'], elements['F']),
    (elements['E'], elements['L']),
    (elements['F'], elements['I' ]),
    (elements['F'], elements['L']),
    (elements['F'], elements['M']),
    (elements['G'], elements['H']),
    (elements['G'], elements['L']),
```

```
    (elements['G'], elements['M']),
    (elements['I'], elements['J']),
    (elements['J'], elements['K']),
    (elements['J'], elements['N']),
    (elements['K'], elements['O']),
    (elements['K'], elements['P']),
    (elements['M'], elements['N']),
    (elements['N'], elements['O']),
    (elements['B'], elements['I']),
    (elements['B'], elements['F']),
    (elements['P'], elements['O'])
]

# L'espace de recherche.
espace = Espace(elements.values(), arcs)
print('L\'espace de recherche:\n{}'.format(espace))

print('Recherche DFS:')
dfs = RechercheDFS(espace, False)
chemin = dfs.recherche(elements['A'], elements['P'])

print('Chemin: {}'.format(chemin))

##################################################

print('\nRecherche BFS:')
bfs = RechercheBFS(espace, False)

chemin = bfs.recherche(elements['A'], elements['P'])
print('Chemin: {}'.format(chemin))

##################################################

print('\nRecherche A*:')
astar = RechercheAStar(espace, False)
chemin = astar.recherche(elements['A'], elements['P'])

print('Chemin: {}'.format(chemin))
```

Module `.../exemple_carte_suisse.py` :

```
from moteurs_recherche.element import Element
from moteurs_recherche.ville import Ville
from moteurs_recherche.espace import Espace
from moteurs_recherche.dfs import RechercheDFS
from moteurs_recherche.bfs import RechercheBFS
from moteurs_recherche.astar import RechercheAStar

# Les éléments dans l'espace de recherche (les  villes ).
elements = {
    'Lausanne': Ville(110, 260, 'Lausanne'),
    'Genève': Ville(40, 300, 'Genève'),
    'Sion': Ville(200, 300, 'Sion'),
    'Neuchâtel': Ville(150, 170, 'Neuchâtel'),
    'Bern': Ville(210, 280, 'Bern'),
    'Basel': Ville(230, 65, 'Basel'),
    'Fribourg': Ville(175, 200, 'Fribourg'),
    'Zürich': Ville(340, 90, 'Zürich'),
```

```
    'Aarau': Ville(290, 95, 'Aarau'),
    'Luzern': Ville(320, 155, 'Luzern'),
    'St-Gallen': Ville(85, 455, 'St-Gallen'),
    'Thun': Ville(235, 210, 'Thun'),
}

# Les arcs liant les éléments (les routes).
arcs = [
    (elements['Lausanne'], elements['Genève']),
    (elements['Sion'], elements['Lausanne']),
    (elements['Neuchâtel'], elements['Lausanne']),
    (elements['Fribourg'], elements['Lausanne']),
    (elements['Fribourg'], elements['Bern']),
    (elements['Sion'], elements['Thun']),
    (elements['Neuchâtel'], elements['Bern']),
    (elements['Basel'], elements['Bern']),
    (elements['Zürich'], elements['Aarau']),
    (elements['Zürich'], elements['Luzern']),
    (elements['Bern'], elements['Aarau']),
    (elements['Bern'], elements['Luzern']),
    (elements['Luzern'], elements['Aarau']),
    (elements['St-Gallen'], elements['Zürich']),
    (elements['Thun'], elements['Bern']),
    (elements['Basel'], elements['Zürich']),
]

# L'espace de recherche.
espace = Espace(elements.values(), arcs)
print('L\'espace de recherche:\n{}'.format(espace))

print('Recherche DFS:')
dfs = RechercheDFS(espace, False)
chemin = dfs.recherche(elements['Lausanne'], elements['Zürich'])

print('Chemin: {}'.format(chemin))

################################################

print('\nRecherche BFS:')
bfs = RechercheBFS(espace, False)
chemin = bfs.recherche(elements['Lausanne'], elements['Zürich'])

print('Chemin: {}'.format(chemin))

################################################

print('\nRecherche A*:')
astar = RechercheAStar(espace, False)
chemin = astar.recherche(elements['Lausanne'], elements['Zürich'])

print('Chemin: {}'.format(chemin))
```

Exercice 7.1.1 Classes de base

La classe `Element`

Dans cet exercice, vous devrez manipuler des éléments situés dans un espace de recherche — plus précisément, des villes dans un espace à deux dimensions. L'espace de recherche est muni d'une notion de distance appliquable à chaque paire d'éléments. L'objectif des algorithmes de recherche que nous allons programmer sera de trouver le chemin le plus court entre deux éléments.

La classe `Element` du module `element.py` représentera donc un élément générique placé dans un espace de recherche. Cet élément sera caractérisé par un seul attribut, `self.nom`. Commencez par compléter le constructeur de la classe, qui doit initialiser cet attribut au moyen d'une valeur passée en paramètre. Ensuite, complétez la méthode `__eq__`, qui doit vérifier l'égalité entre l'élément courant et un autre. Deux éléments seront réputés égaux (`__eq__` retourne `True`) lorsqu'ils possèdent des noms égaux. Notez que `Element` contient aussi une fonction qui retourne la distance entre l'élément courant et un autre élément. Cette fonction n'a ici qu'une implémentation triviale et devra être surchargée de manière appropriée dans les sous-classes.

La classe `Ville`

La classe `Element` nous fournit un modèle que les éléments de recherche doivent spécialiser. Notre but dans cet exercice est de trouver des chemins entre des villes, qui sont des éléments dans un espace à deux dimensions. Nous définissons donc une sous-classe `Ville`, qui étend `Element` en lui ajoutant deux attributs :

- `x` : la position de l'élément sur l'axe des x ;
- `y` : la position de l'élément sur l'axe des y.

Commencez par coder le constructeur de la classe, qui doit initialiser ces attributs à partir des valeurs passées en paramètre. Ensuite, surchargez la méthode d'égalité `__eq__`, afin qu'elle compare les noms et les coordonnées. Finalement, surchargez la fonction de distance, afin de retourner la distance euclidienne entre deux villes. Pour cette dernière opération, vous pouvez utiliser l'opérateur de mise à la puissance de Python, qui est `**` (par exemple : `3**2 == 9`). En outre, `from math import sqrt` permet d'importer uniquement la fonction racine carrée du module `math`.

La classe `Espace`

Dans l'espace de recherche, chaque élément sera lié à d'autre éléments placés à proximité — ses voisins. Par exemple, si l'espace de recherche représente une carte de la Suisse, Lausanne sera parmi les éléments voisins de Genève. Nous indiquerons la proximité entre deux éléments par un tuple `(element_1, element_2)`. Du point de vue formel, un tuple représente ainsi un arc dans le graphe constitué par les éléments et leurs relations de vicinité.

Nous utiliserons la classe `Espace` pour représenter un espace de recherche. `Espace` stockera tous ses éléments dans une liste `self.elements` et tous ses arcs

dans `self.arcs`. Le constructeur initialise les listes d'éléments et d'arcs, soit avec des collections passées en paramètre, soit comme des listes vides lorsqu'il est appelé sans arguments.

La classe contient aussi les méthodes suivantes, que vous devez implémenter :

- `ajoute_arcs(self, arcs)` : prend en argument une collection de tuples `(a, b)` représentant des arcs, qu'elle ajoute aux collections de l'objet courant. (Cette méthode n'est pas absolument indispensable, car les listes peuvent aussi bien être remplies lors de la construction, mais il est parfois plus pratique et plus lisible d'ajouter des arcs par le biais d'une méthode.).
- `trouve_voisins(self, element)` : retourne la liste de tous les voisins d'un élément (par exemple, si l'arc `(a, b)` est le seul de l'espace `e` : `e.voisins(a)` doit retourner `[b]`.

La classe `Noeud`

Lors de la recherche, chaque `Element` sera modélisé par un nœud dans un arbre de recherche. Chaque nœud contiendra une référence sur un élément. Les nœuds seront créés de façon dynamique au cours de l'exploration de l'espace de recherche par l'algorithme.

Nous avons donc besoin d'une classe `Noeud` (squelette disponible dans `noeud.py`), permettant de modéliser un nœud. Cette classe contiendra quatre attributs :

- `element` : une référence sur un objet de type `Element`.
- `cout_c` : contient le coût $c(n)$, c'est-à-dire le coût depuis le nœud de départ jusqu'au nœud en question. Il s'agit de la somme minimale des longueurs des arcs entre l'élément référencé par le nœud de départ et l'élément référencé par le nœud courant. Dans notre cas, la longueur d'un arc est donnée par la distance entre ses deux éléments.
- `cout_f` : contient le coût $f(n)$, c'est-à-dire le coût heuristique utilisé par l'algorithme A*, qui est égal à $c(n) + h(n)$. Dans notre cas, la fonction heuristique $h(n)$ calcule la distance euclidienne entre l'élément contenu par le nœud courant et l'élément but ; le coût heuristique modélise ainsi la distance au but, de manière à privilégier l'exploration du nœud le plus prometteur ;
- `parent` : le nœud parent du nœud courant, c'est-à-dire le nœud qui a conduit au nœud courant durant la recherche.

Écrivez donc un constructeur qui initialise ces quatre attributs à partir de valeurs passées en paramètres.

Exercice 7.1.2 Algorithmes de recherche

Nous allons maintenant développer un outil de recherche qui implémente les trois principaux algorithmes : DFS (*Depth-first search* : Recherche en profondeur-d'abord), BFS (*Breadth-first search* : Recherche en largeur-d'abord) et A* (Recherche par A*).

Le principe de tout algorithme de recherche est de trouver un élément but (dans notre exemple, la ville de destination) à partir d'un élément initial (la ville de départ). Lors de l'exploration de l'espace des éléments, la recherche traverse un arbre constitué de nœuds qui sont liés à leurs nœuds successeurs. Chaque nœud de recherche correspond à une étape dans la recherche. L'exploration d'un nœud de recherche permet, s'il en a, de trouver ses successeurs — qui deviennent de nouveaux nœuds de recherche.

Les algorithmes que nous allons étudier ne se différencient que par la gestion de la liste des nœuds ouverts `Q` :

- DFS (en profondeur-d'abord) : place les nouveaux nœuds en tête de `Q` ;
- BFS (en largeur-d'abord) : place les nouveaux nœuds en queue de `Q` ;
- A* : insère les nouveaux nœuds de telle sorte que `Q` soit toujours ordonnée selon le coût heuristique croissant de ces nœuds.

Le pseudocode de l'algorithme de recherche vous est donné ci-dessous :

```
Recherche(noeud_depart, noeud_but, methode)
 1. Q <− [noeud_depart]
 2. WHILE Q n'est pas vide DO
 3.     n <− premier(Q)
 4.     Q <− reste(Q)
 5.     IF n == noeud_but THEN
 6.         RETURN chemin de noeud_depart àn
 7.     ELSE
 8.         S <− successeurs de n
 9.         Q <− AjouterSuccesseurs(Q, S, methode)
10.     END IF
11. END WHILE
12. RETURN échec
END Recherche
```

Comme vous pouvez le constater, l'algorithme de base est le même quelle que soit la méthode utilisée (DFS, BFS et A*). La seule différence réside dans la façon d'ajouter les successeurs à la liste `Q`. Plus concrètement, la fonction qui ajoute les successeurs à `Q` est définie comme suit :

```
AjouteSuccesseurs(Q, S, methode)
 1. IF methode == DFS THEN
 2.     RETURN S + Q
 3. ELSE IF methode == BFS THEN
 4.     RETURN Q + S
 5. ELSE IF methode == A∗ THEN
 6.     Q <− Q + S
 7.     Q <− Q trié par coût heuristique
 8.     RETURN Q
 9. ELSE
10.     RETURN échec
11. END IF
END AjouteSuccesseurs
```

Les classes de recherche

L'algorithme de recherche doit être implémenté dans la classe `Recherche` de `recherche.py`. Un constructeur de la classe, qui initialise le graphe de recherche,

vous est déjà donné. Nous avons également donné une fonction d'interface `recherche`, qui prend en paramètres deux éléments, crée deux nœuds de recherche contenant ces éléments et les passe en arguments à la méthode `recherche_chemin`. C'est dans celle-ci que vous devez implémenter l'algorithme. Appelez les méthodes `trouve_successeurs` et `ajoute_successeurs` aux étapes 8 et 9 de l'algorithme. Afin de retourner le chemin depuis le nœud de départ, appelez la méthode `trouve_chemin`, qui vous est déjà donnée.

Ensuite, complétez la méthode `trouve_successeurs`. Cette méthode doit retourner une liste contenant tous les successeurs d'un nœud passé en paramètre. La méthode doit retourner tous les éléments voisins de l'élément encapsulé par le nœud courant, eux-mêmes contenus dans de nouveaux nœuds de recherche (sauf le parent du nœud courant, pour éviter les cycles...). Notez qu'en créant ces nouveaux nœuds, vous devez aussi initialiser leurs coûts, à partir du coût du nœud courant et en utilisant la fonction heuristique `self.h`. Cette dernière est initialisée à l'aide d'une fonction lambda dans la méthode d'interface `recherche`.

Ensuite, vous pouvez passer à la méthode `ajoute_successeurs`. Il convient d'implémenter cette méthode différemment pour chacun des trois algorithmes, dans les sous-classes de `Recherche` : `RechercheDFS` (`dfs.py`), `RechercheBFS` (`bfs.py`) et `RechercheAStar` (`astar.py`)

Test du programme

Testez vos algorithmes sur les fichiers `exemple_carte_simple.py` et `exemple_carte_suisse.py`.

```
python3 exemple_carte_simple.py
python3 exemple_carte_suisse.py
```

Qu'en concluez-vous ? Quelle est l'importance de l'heuristique utilisée par A* ? Regardez surtout le nombre de nœuds de recherche examinés. Que pouvez-vous conclure sur les algorithmes en regardant la longueur des chemins (en nombre de villes traversées) ?

Pourquoi l'algorithme DFS boucle-t-il à l'infini sur les deux cartes ? Testez maintenant en commentant le code appelant l'algorithme DFS. Quel résultat obtenez-vous ?

Exercice 7.1.3 Algorithmes de recherche avec détection de cycles

Comme vous avez pu le constater, l'algorithme n'est pas très efficace car, dans un espace de recherche cyclique, certains nœuds peuvent être visités à plusieurs reprises. Pire, l'algorithme peut se retrouver prisonnier d'une boucle infinie.

Détecter et éviter les cycles nécessitent de maintenir une liste des nœuds déjà explorés. Dans le cas du DFS et du BFS, l'algorithme devra contrôler si ce nœud est déjà présent dans la liste avant de chercher ses successeurs. Lorsque c'est le cas, nous savons que les successeurs ont déjà été construits, et il n'est pas utile de recommencer. L'algorithme A* est un peu plus compliqué de ce point de vue : on doit revisiter un nœud si et seulement si le coût `f(n)` est inférieur au coût `f(n)` du nœud la dernière fois qu'il a été exploré

En résumé, l'algorithme de recherche optimisé est le suivant :

```
RechercheOptimisee(noeud_depart, noeud_but, methode)
 1. Q <− [noeud_depart]
 2. C <− []
 3. WHILE Q non vide DO
 4.     n <− premier(Q), Q <− reste(Q)
 5.     IF n == noeud_but THEN
 6.         RETURN chemin de noeud_depart àn
 7.     ELSE
 8.         IF n not in C or
                (n=n' in C and f(n)<f(n') and methode is "A∗") THEN
 9.             S <− successeurs de n
10.             Q <− AjouterSuccesseurs(Q, S, methode)
11.             ajoute n dans C
12.         END IF
13.     END IF
14. END WHILE
15. RETURN échec
END RechercheOptimisee
```

En vous basant sur le pseudocode ci-dessus, implémentez l'algorithme de recherche optimisé. Vous devrez d'abord implémenter la méthode `detecte_cycle`, qui teste si un nœud a déjà été exploré, étant donné le nœud courant et la collection `trace` des nœuds déjà explorés. Notez que vous devez surcharger cette méthode dans la classe `RechercheAStar`, afin de tolérer les cycles qui permettent de trouver un chemin plus court.

Nous vous recommandons de ne pas implémenter une nouvelle méthode `recherche_chemin`, mais de modifier la version existante afin de traiter le cas où l'attribut de la classe `self.optimisee` prend la valeur `True`. En outre, afin d'optimiser la détection des cycles, nous vous suggérons d'implémenter `trace` comme un dictionnaire qui associe chaque élément au nœud de recherche qui le contient.

Lorsque vous avez terminé, vous pouvez tester à nouveau votre programme. Qu'observez-vous cette fois ?

Exercice 7.1.4 Labyrinthe

À titre d'exercice supplémentaire, vous pouvez tester votre implémentation en lui faisant découvrir un chemin entre deux points d'un labyrinthe. Pour simplifier, vous coderez ce dernier sous la forme d'une matrice de caractères `L`, telle que `L[i][j]` prenne la valeur `'x'` si la cellule `(i,j)` est occupée et ne peut pas être explorée.

Vous pouvez ensuite implémenter une méthode qui traduira cette matrice en un objet `Espace`, qui contiendra les déplacements possibles depuis chaque cellule (par exemple, en haut, en bas, à gauche et à droite si les cellules voisines sont libres). Appliquez les trois algorithmes de recherche pour construire le chemin conduisant d'un point quelconque à un autre et comparez les résultats.

Solutions à la page 365

Chapitre 8

Satisfaction de contraintes

On peut observer que les algorithmes de recherche généraux ont une complexité exponentielle en la taille du problème. Cela n'est souvent pas acceptable, et on doit alors chercher des heuristiques valables pour le domaine en question qui permettent souvent de résoudre le problème spécifique avec un temps de calcul beaucoup plus raisonnable.

Il serait alors intéressant de définir un cadre limité de problèmes qui soit à la fois suffisamment spécifique pour admettre des heuristiques efficaces, mais aussi suffisamment général pour s'appliquer à une large gamme d'applications pratiques. Le paradigme de la *satisfaction de contraintes* est un tel modèle qui a rencontré beaucoup de succès en pratique.

Un problème de satisfaction de contraintes (PSC) se caractérise comme suit :

- Le problème peut être décrit par un ensemble de variables. Une solution est donnée par l'affectation d'une valeur à chaque variable. La plupart des problèmes auxquels s'intéresse l'IA (comme la conception, la planification ou encore la programmation logique) sont susceptibles d'être formalisés de cette manière.
- La valeur d'une variable appartient à un *domaine*, qui est soit un ensemble fini de valeurs discrètes (satisfaction symbolique), soit un ensemble d'intervalles numériques (satisfaction continue).
- Les contraintes de consistance agissent sur les variables. On distingue les contraintes *unaires*, qui déterminent la valeur d'une seule variable, les contraintes *binaires* qui concernent les combinaisons des valeurs de deux variables et les contraintes *multiples* qui concernent des combinaisons de plus de deux variables.

Cette formulation est applicable à de nombreux problèmes pratiques. Citons par exemple :

- Dans l'ordonnancement et la planification de tâches, le but est de trouver un ensemble d'actions qui respecte les contraintes dérivant du but à atteindre et des moyens à disposition. Les variables sont alors les tâches, leurs domaines, les ressources et les intervalles de temps durant lesquels celles-ci peuvent s'effectuer. Les contraintes exigent qu'aucune ressource ne puisse être affectée à deux tâches simultanément et que les tâches doivent être accomplies dans les délais.

- Dans la conception ou configuration, le but est de trouver un ensemble de composants et de connections qui respectent toutes les contraintes fonctionnelles. Les variables sont alors les fonctionnalités, leurs domaines les composants qui peuvent les réaliser, et les contraintes de la compatibilité, le respect des coûts, etc.
- Dans la vision, le but est de trouver une interprétation qui soit consistante avec les observations. Les variables sont alors les observations, leurs domaines les interprétations et les contraintes des contraintes de compatibilité entre observations et interprétations.

8.1 Définition des problèmes de satisfaction de contraintes (PSC)

Étant donné la généralité du problème, il est utile de définir des méthodes générales pour le résoudre de manière efficace. Formellement, un PSC = (X,D,C) s'exprime de la manière suivante :

Étant donné :

- Les variables X = $x_1, x_2, ..., x_n$.
- Les domaines D = $D_1, D_2, ..., D_n$ associés aux variables.
- Les contraintes C = $C_1(x_k, x_l, ...), C_2, ..., C_m$, qui restreignent les combinaisons de valeurs possibles pour les variables sur lesquelles elles portent.

Trouver :

Toutes les solutions :
$\{x_1 = v_k, x_2 = v_l, ..., x_n = v_o\}$ telles que toutes les contraintes soient satisfaites.

Dans ce livre, nous traitons uniquement des techniques pour variables à valeurs *discrètes*, c'est-à-dire que les domaines $D_1, ..., D_n$ sont des ensembles finis de valeurs. Il existe des différences importantes entre des problèmes à domaines finis et des problèmes à domaines infinis.

Dans le cas où toutes les contraintes sont binaires, on peut formuler le problème comme un graphe où les nœuds sont les variables et les arcs sont des contraintes entre ces variables. On parle alors d'un graphe ou d'un *réseau* de contraintes.

Le formalisme des PSC a permis de développer un grand nombre d'heuristiques et de méthodes efficaces valables pour tout problème formalisé sous cette forme. On a même développé des *langages de programmation logique par contraintes* tels que PROLOG 3 et Eclipse.

Un premier exemple d'un PSC est l'allocation de ressources à des tâches échelonnées dans le temps (fig. 8.1). On a ici un certain nombre de tâches (T_1 à T_4), dont chacune peut être exécutée par une ressource parmi un ensemble

restreint. Le but est alors de trouver une assignation des ressources aux tâches de manière à ce qu'aucune ressource n'effectue plus d'une tâche en même temps. Le problème se formule facilement comme PSC :

- Variables $x_1..x_n$ correspondant aux tâches $T_1, T_2..T_n$ (valeur = *une* ressource).
- Domaines = ressources qui peuvent effectuer la tâche. Par exemple, $D_1 = \{B, C\}$.
- Contraintes = deux tâches se chevauchant dans le temps ne peuvent être effectuées par la même ressource.

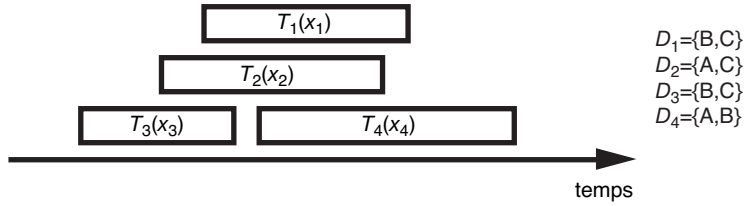


Fig. 8.1 *Quatre tâches T_1 à T_4 à effectuer avec 3 ressources.*

Ce problème peut être résolu par un algorithme de *generate-and-test* : essayer toutes les combinaisons des valeurs admissibles pour les variables et retenir celles qui respectent toutes les contraintes. Pour l'exemple de la figure 8.1, on a deux ressources possibles pour chacune des quatre tâches, donc il y a $2^4 = 16$ combinaisons de ressources pour $T_1T_2T_3T_4$:

(BABA)	(BABB)	(BACA)	(BACB)
(BCBA)	(BCBB)	(BCCA)	(BCCB)
(CABA)	(CABB)	(CACA)	(CACB)
(CCBA)	(CCBB)	(CCCA)	(CCCB)

En vérifiant les contraintes de non-simultanéité pour chaque combinaison, on peut observer que seule la combinaison BBAC satisfait toutes les contraintes et est donc une solution. Il est évident que la complexité d'une solution fournie par *generate-and-test* est toujours exponentielle par rapport au nombre de variables. Un PSC typique implique un grand nombre de variables, ce qui rend cette méthode inapplicable. Des algorithmes plus efficaces ont donc été définis.

L'idée sous-jacente à presque toutes les méthodes de satisfaction de contraintes consiste à vérifier les contraintes sur des instanciations partielles de valeurs à un sous-ensemble de variables. Une instanciation partielle est une solution partielle seulement si toutes les contraintes entre les variables auxquelles on a assigné une valeur sont satisfaites. Une instanciation partielle n'est alors étendue que si elle est une solution partielle.

Cela donne lieu à une recherche où les nœuds sont des solutions partielles et la fonction de successeur consiste à assigner une prochaine valeur à l'une des variables qui n'a pas encore de valeur. On vérifie alors les contraintes pour

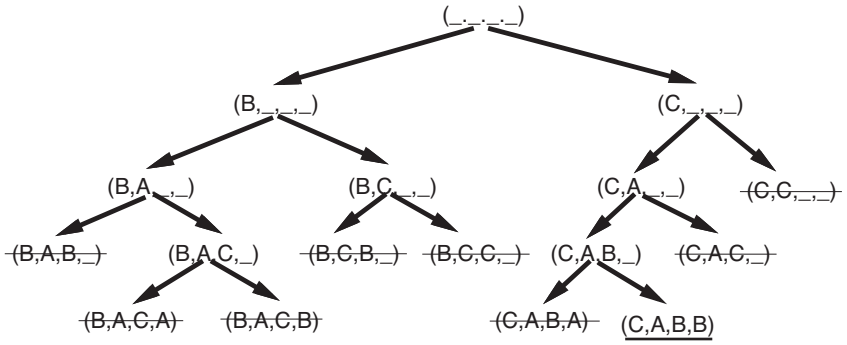


Fig. 8.2 *Arbre de recherche pour la résolution d'un problème d'allocation de ressources par une recherche en profondeur-d'abord.*

filtrer les nœuds qui ne sont pas consistants. La figure 8.2 montre l'arbre de recherche résultant pour l'exemple de la figure 8.1.

Il existe un certain nombre d'heuristiques pour rendre encore plus efficace une telle recherche. Si la recherche évite d'explorer plus loin des instanciations inconsistantes, les tests d'inconsistance ne peuvent s'appliquer que dans l'ordre d'instanciation des variables. À chaque backtrack, on perd l'information sur les combinaisons consistantes de valeurs pour les dernières variables instanciées. Donc, il arrive souvent que l'algorithme explore plusieurs fois la même combinaison contradictoire de valeurs.

Pour pallier ce désavantage, on pourrait imaginer une application d'un filtrage uniforme sur toutes les variables et non seulement sur celles qui sont déjà instanciées. C'est l'idée des algorithmes de *consistance partielle* : d'abord limiter l'espace de recherche à un sous-ensemble prometteur en éliminant des combinaisons de valeurs qui dans aucun cas ne peuvent respecter toutes les contraintes. Ce processus peut s'appliquer comme prétraitement avant ou entre différentes étapes de recherche. Par exemple, imaginons qu'une contrainte implique que la valeur $T_1 = A$ ne puisse jamais faire partie d'une solution. On éliminera alors ce choix et la recherche ne va plus considérer du tout l'assignation $T_1 = A$. Comme l'élimination d'une valeur en entraîne souvent l'élimination d'autres, on parle d'une *propagation* des contraintes.

8.2 Formulation d'un réseau de contraintes binaires

La plupart des travaux traitent des contraintes *binaires* qui s'avèrent les plus adaptées à la propagation. Un problème de satisfaction de contraintes binaires peut alors être représenté par un graphe où :

- les nœuds sont les variables,
- les arcs représentent les contraintes.

On appelle ceci un *réseau de contraintes*. Par exemple, le problème d'allocation de ressources de la figure 8.1 donnera lieu au réseau de contraintes

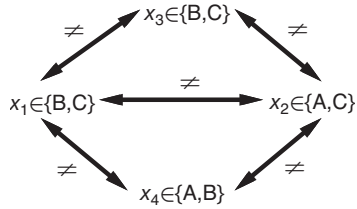


Fig. 8.3 *Réseau de contraintes pour l'exemple de l'allocation de ressources.*

de la figure 8.3. Pour cet exemple, toutes les contraintes sont des contraintes d'inégalité ($\neq$) signifiant que les valeurs assignées aux deux variables doivent être différentes.

Si la représentation comme graphe est bien adaptée à un réseau de contraintes binaires, la plupart des problèmes pratiques conduisent à des contraintes multiples. Pourvu qu'il s'agisse d'un problème à valeurs discrètes, il existe deux méthodes pour transformer un réseau n-aire en un réseau binaire : la *projection* et la *transformation en réseau dual*.

Une contrainte impliquant plus de deux variables peut être approximée par un réseau de contraintes binaires, sa *projection*. La figure 8.4 montre un exemple : la contrainte R est représentée par trois contraintes à deux variables en retenant les combinaisons de valeurs qui figurent dans la liste des combinaisons admises par R.

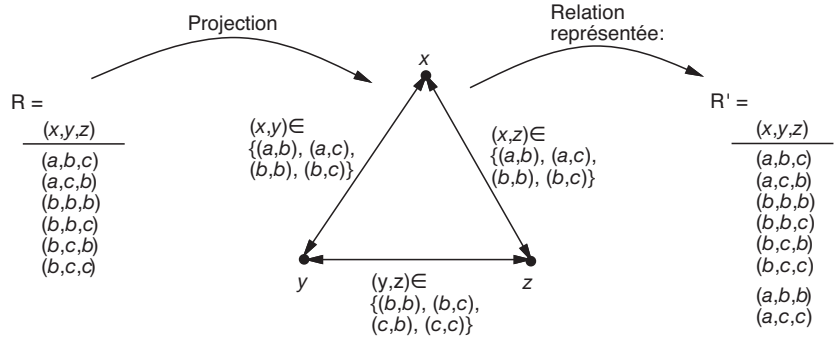


Fig. 8.4 *Projection d'une contrainte R à trois variables. Le réseau de trois contraintes qui en résulte admet deux combinaisons de valeurs en plus :* (a, b, b) *et* (a, c, c).

Cependant, une perte d'information s'ensuit souvent d'une projection : dans l'exemple, le réseau qui en résulte admet en effet deux combinaisons de valeurs en trop ! Cette perte d'information est inévitable : une relation à n variables avec des domaines de taille m représente un volume d'information de m^n bits. Sa projection consiste en $n \cdot (n-1)/2$ contraintes dont chacune ne représente que m^2 bits d'information. Comme en général $m^n < n \cdot (n-1)/2 \cdot m^2$, l'information de la contrainte à n variables ne peut pas être complètement représentée.

La deuxième possibilité consiste à transformer l'hypergraphe du PSC avec contraintes à variables multiples en un problème binaire équivalent en construisant un graphe *dual* G_D au réseau de contraintes G :

- nœuds de G_D = contraintes de G,
- arcs de G_D = ensembles de variables en commun : la contrainte exige que les instanciations soient les mêmes des deux côtés.

On peut montrer qu'il n'y a alors aucune perte d'informations : le PSC transformé admet exactement les mêmes solutions que le problème original.

La figure 8.5 montre un exemple d'un PSC avec des contraintes à plus de 2 variables.

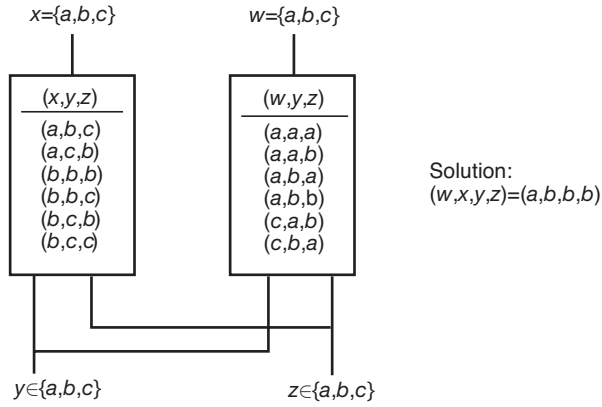


Fig. 8.5 *Un problème impliquant des contraintes à variables multiples.*

Sa transformation en graphe dual donnera deux nœuds :

1) $\alpha = (x, y, z) \in$
   $\{(a, b, c), (a, c, b), (b, b, b), (b, b, c), (b, c, b), (b, c, c)\}$
2) $\beta = (w, y, z) \in$
   $\{(a, a, a), (a, a, b), (a, b, a), (a, b, b), (c, a, b), (c, b, a)\}$

et une contrainte :

$$C : y(\alpha) = y(\beta), z(\alpha) = z(\beta) : \{((b, b, b), (a, b, b)\}$$

et il y a une seule combinaison qui satisfait la contrainte :

$$\alpha = (b, b, b), \beta = (a, b, b)$$

Comme il est alors possible de transformer tout PSC *discret* en un problème à contraintes binaires, nous nous concentrerons par la suite sur les réseaux de contraintes binaires uniquement. Nous ne considérons explicitement des algorithmes pour des contraintes non binaires que pour le cas de variables continues, où une telle transformation n'est pas possible.

8.3 Solution d'un PSC par recherche

Tout PSC discret peut être résolu par un algorithme de recherche : on énumère toutes les combinaisons imaginables de valeurs des variables, et on retient celles qui respectent les contraintes. Cependant, un PSC comporte souvent des centaines de variables, dont les domaines peuvent également prendre une taille considérable. Un algorithme de recherche sans aucune optimisation serait donc tellement inefficace que l'on ne peut envisager une telle solution. Dans le contexte des PSC, on utilise donc des heuristiques spécifiques qui permettent une recherche plus efficace.

En fait, il y a deux types de méthodes pour la résolution d'un PSC par recherche. Le premier prend comme base un algorithme de recherche en profondeur-d'abord et y ajoute certaines heuristiques et méthodes de propagation de labels qui le rendent beaucoup plus efficace. Le deuxième type de méthode s'inspire de l'optimisation et se base sur une modification itérative d'une assignation complète pour minimiser et finalement éliminer tout conflit avec les contraintes. Il s'agit donc d'une propagation de valeurs.

8.3.1 Méthodes basées sur la recherche en profondeur-d'abord

Pour résoudre un PSC, on peut utiliser un algorithme de recherche en profondeur-d'abord où :

- nœud de recherche = instanciation de variables $x_1 = v_1, x_2 = v_2, ..., x_k = v_k$ (k est la profondeur du nœud dans l'arbre de recherche, le nœud racine étant par convention à la profondeur 0) ;
- fonction de successeur = instanciation de la variable $x_{k+1} = v_{k+1}$ de manière à respecter toutes les contraintes avec $x_1, ..., x_k$;
- nœud initial = instanciation vide ;
- nœud but = instanciation de toutes les variables $x_1, ..., x_n$.

La figure 8.6 montre comment l'algorithme de recherche en profondeur-d'abord (aussi appelé « backtrack ») peut être adapté à la résolution d'un PSC. La recherche peut être rendue plus efficace par :

- les valeurs considérées : pour limiter leur choix, on utilise les méthodes du *forward checking* ou du *lookahead*, basées sur la consistance partielle des contraintes ;
- l'ordre d'instanciation des variables : on utilise des heuristiques basées sur la structure du réseau de contraintes.

En pratique, ces méthodes conduisent à des améliorations très sensibles de l'efficacité des algorithmes.

x = tableau de n variables, rempli jusqu'à k
d = domaines des variables

```
1:  Function DFS(x,d,k)
2:  if k > n then
3:     return x
4:  else
5:     for v ∈ d[k] do
6:        consistent ← true
7:        for i ← 1 to k-1 do
8:           if ¬ consistent(v,x[i],C(i,k)) then consistent ← false
9:        if consistent then
10:          x[k] ← v
11:          rest ← DFS(x,d,k)
12:          if rest ≠ :echec then return rest
13: return :echec
```

Fig. 8.6 *Algorithme pour la solution d'un PSC par recherche en profondeur-d'abord.*

Comme exemple, considérez à nouveau le problème d'allocation de ressources que montre la figure 8.7.

Nous avons les variables et domaines :

$$D_1 = \{B, C\}$$
$$D_2 = \{A, C\}$$
$$D_3 = \{B, C\}$$
$$D_4 = \{A, B\}$$

et les contraintes :

$$C(x_1, x_2) : \{(B, A), (B, C), (C, A)\}$$
$$C(x_1, x_3) : \{(B, C), (C, B)\}$$
$$C(x_1, x_4) : \{(B, A), (C, B), (C, A)\}$$
$$C(x_2, x_3) : \{(A, B), (A, C), (C, B)\}$$
$$C(x_2, x_4) : \{(A, B), (C, A), (C, B)\}$$

L'algorithme `DFS` sera alors appelé avec `d = [(B,C),(A,C),(B,C),(A,B)]` et trouve la solution en 12 pas, comme le montre la figure 8.8.

Backjumping

Si l'assignation à x_{k+1} échoue, il faut changer au moins une variable qui y est liée par une contrainte. Dans la trace de la figure 8.8, le premier retour-arrière

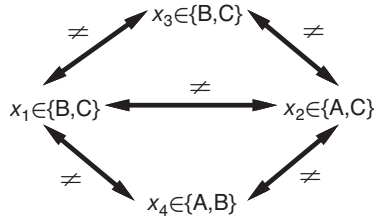


Fig. 8.7 *Problème d'allocation de ressources.*

Pas	k	x[1]	x[2]	x[3]	x[4]	
1	1	B	-	-	-	
2	2	B	A	-	-	
3	3	B	A	C	-	
4	4	B	A	C	*	retour-arrière !
5	3	B	A	*	-	retour-arrière !
6	2	B	C	-	-	
7	3	B	C	*	-	retour-arrière !
8	2	B	*	-	-	retour-arrière !
9	1	C	-	-	-	
10	2	C	A	-	-	
11	3	C	A	B	-	
12	4	C	A	B	B	solution !

Fig. 8.8 *Trace de l'exécution du DFS simple. Les « * » indiquent qu'il n'y a aucune valeur qui n'est pas en conflit avec les assignations déjà effectuées.*

aurait pu directement revenir sur x_2, puisque aucune contrainte lie x_3 et x_4. Ceci est l'idée de la règle du *backjumping* :

> Quand il n'y a pas de valeur consistante pour la variable x_{k+1}, revenir directement à la dernière variable qui a une contrainte avec x_{k+1}.

Dans l'exemple, cela évite le pas 5. La méthode peut être rendue encore plus ciblée dans le *conflict-directed backjumping* :

> Quand il n'y a pas de valeur consistante pour la variable x_{k+1}, revenir à la dernière variable qui avait un *conflit* avec une des valeurs du domaine de x_{k+1}.

Forward checking

Si le *backjumping* produit un gain d'efficacité dans le cas où un retour-arrière a effectivement lieu, le *forward checking* a pour but d'éviter à l'avance des instanciations inconsistantes en appliquant le critère de la consistance des arcs pendant la recherche.

Il exige qu'on ajoute un label `l[i]` à chaque variable, qui sera initialement égal à son domaine. La règle du *forward checking* met à jour ces labels en appliquant la règle suivante :

> À chaque instanciation d'une variable x_k, éliminer toutes les valeurs inconsistantes avec x_k des labels des variables qui ne sont pas encore instanciées.

Lookahead et consistance des arcs

Le *forward checking* est très semblable à la consistance des arcs, mais diffère car la propagation n'est appliquée qu'aux contraintes impliquant la dernière variable assignée. Si on étend cette méthode pour continuer la propagation récursivement, l'heuristique s'appelle le *lookahead*. Si en plus on fait une itération jusqu'à ce qu'il n'y a plus de changement, on obtient la consistance des arcs à chaque pas de la recherche. Les versions plus poussées ont l'avantage d'éliminer au maximum le besoin de recherche, mais elles sont elles-mêmes coûteuses à appliquer. En pratique, il semblerait que les heuristiques plus poussées soient avantageuses pour des grands problèmes.

Comme le montre la trace de la figure 8.9, le *forward checking* permet de résoudre l'exemple avec uniquement deux retours-arrière. La trace du *lookahead* (fig. 8.10) donne un résultat encore meilleur : un seul retour-arrière suffit.

k	x[1]	x[2]	x[3]	x[4]	l[1]	l[2]	l[3]	l[4]	
0	-	-	-	-	B,C	A,C	B,C	A,B	
1	B	-	-	-	C	A,C	C	A	
2	B	A	-	-	C	C	C	-	retour-arrière !
2	B	C	-	-	C	-	-	A	retour-arrière !
1	C	-	-	-	-	A	B	A,B	
2	C	A	-	-	-	-	B	B	
3	C	A	B	-	-	-	-	B	
4	C	A	B	B	-	-	-	-	solution !

Fig. 8.9 *Trace de la recherche avec forward checking.*

k	x[1]	x[2]	x[3]	x[4]	l[1]	l[2]	l[3]	l[4]	
0	-	-	-	-	B,C	A,C	B,C	A,B	
1	B	-	-	-	-	A	C	-	retour-arrière !
1	C	-	-	-	-	A	B	B	
2	C	A	-	-	-	-	B	B	
3	C	A	B	-	-	-	-	B	
4	C	A	B	B	-	-	-	-	solution !

Fig. 8.10 *Trace de la recherche avec lookahead.*

Ordonnancement des variables

L'autre possibilité pour influencer la recherche consiste à influencer l'ordre dans lequel on considère l'assignation des variables. Il y a plusieurs heuristiques d'ordre qui sont proposées dans la littérature. Parmi les plus importantes, on trouve :

- *dynamic variable ordering* (DVO) : prendre la variable non instanciée dont le label est le plus petit (combinaison avec *forward checking/lookahead*) ;
- *min-width ordering* : prendre la variable non instanciée connectée au plus petit nombre de variables non instanciées ;
- *max-degree ordering* : prendre la variable la plus connectée dans le graphe original ; ceci est un ordre statique qui ne dépend pas de l'état de la recherche.

Les deuxième et troisième heuristiques sont souvent couplées à la première : dans les cas où l'ordre produit n'est pas unique, on utilise la deuxième puis la troisième heuristique pour décider. Cette combinaison est, selon l'état des connaissances en 1996, la plus performante pour les problèmes pratiques.

La figure 8.11 montre la trace d'une recherche qui utilise uniquement l'heuristique DVO en combinaison avec le forward checking. Maintenant, un seul retour-arrière suffit pour trouver la solution. Ce résultat peut encore être amélioré en utilisant en plus l'heuristique *min-width*, comme le montre la figure 8.12 qui trouve la solution sans aucun retour-arrière. Pour cet exemple, on ne peut pas faire mieux !

Pas	x[1]	x[2]	x[3]	x[4]	l[1]	l[2]	l[3]	l[4]	
0	-	-	-	-	B,C	A,C	B,C	A,B	
1	B	-	-	-	C	A,C	C	A	
2	B	-	C	-	-	A	-	A	
3	B	A	C	-	-	-	-	-	retour-arrière !
4	C	-	-	-	-	A	B	A,B	
5	C	A	-	-	-	-	B	B	
6	C	A	B	-	-	-	-	B	
7	C	A	B	B	-	-	-	-	solution !

Fig. 8.11 *Trace de la recherche utilisant le dynamic value ordering (DVO).*

Sur un exemple très simple comme celui que nous avons vu ici, les gains en performance qui sont obtenus grâce à ces algorithmes ne sont pas très impressionnants. Cependant, sur des grands problèmes, il est normal de voir des gains très importants qui rendent possibles la solution de grands systèmes de contraintes. Comme résultats théoriques comparatifs, on peut retenir :

- la recherche par retour-arrière simple est la moins efficace ;
- l'heuristique du *backjumping* (simple) est meilleure, mais visite au moins autant de nœuds que le forward checking et est donc strictement moins puissante que celui-ci ;.

Pas	x[1]	x[2]	x[3]	x[4]	l[1]	l[2]	l[3]	l[4]	
0	-	-	-	-	B,C	A,C	B,C	A,B	
1	-	-	B	-	C	A,C	C	A,B	
2	C	-	B	-	-	A	-	A,B	
3	C	A	B	-	-	-	-	B	
4	C	A	B	B	-	-	-	-	solution !

Fig. 8.12 *Trace de la recherche utilisant l'heuristique DVO suivie de l'heuristique min-width.*

- le *forward checking* et sa version plus poussée du *lookahead* sont les meilleures heuristiques ;
- le *conflict-directed backjumping* peut améliorer la performance du *forward checking*.

8.3.2 Méthodes itératives

À côté des méthodes de recherche incrémentale, qui ne considèrent que des solutions partielles consistantes, il existe des méthodes itératives qui commencent avec n'importe quelle assignation de valeurs et effectuent des changements locaux pour éliminer les conflits avec les contraintes l'un après l'autre. Ceci correspond plutôt à la propagation de valeurs. Nous allons examiner deux algorithmes de ce type :

- l'heuristique min-conflicts,
- le recuit simulé.

Les différences entre les méthodes itératives se situent dans la manière dont les changements locaux sont faits. Ils peuvent se faire :

- de manière déterministe (min-conflicts), c'est-à-dire qu'il y a un critère fixe et que le changement se fait toujours quand ce critère est satisfait ;
- probabiliste (recuit simulé), c'est-à-dire que le critère ne donne que les *probabilités* pour un changement.

Pour toutes les méthodes itératives, la performance dépend essentiellement du choix des valeurs initiales : si elles sont proches d'une solution, il y a évidemment moins de pas qui restent à faire. Ceci rend difficile la comparaison des performances observées sur ces algorithmes avec la recherche incrémentale.

L'idée de l'heuristique min-conflicts est de changer à chaque étape l'assignation de la variable qui réduira le plus le nombre total de conflits. La figure 8.13 montre l'algorithme qui en résulte. Le choix de la variable à changer se fait dans le pas 10 ; on compare le nombre de conflits qui existeront après avoir changé la valeur de la variable. La figure 8.14 montre la trace d'une exécution sur l'exemple de la figure 8.7 que nous avons déjà traité auparavant.

L'heuristique min-conflicts correspond directement à la procédure du *hill-climbing* bien connue dans l'optimisation. Elle est donc susceptible d'amener

X = variables
V = valeurs
C = contraintes

```
1: Function min-conflicts(X,V,C)
2: for i ← 1 to max-tries do
3:    V ← assignation aléatoire
4:    for j ← 1 to max-steps do
5:       nconf ← check(V,C)
6:       if nconf = 0 then
7:          return solution V
8:       else
9:          trouver k tel que changer v[k] donne un nombre minimal de conflits
10:         changer v[k]
11: retourner solution partielle V
```

Fig. 8.13 *Algorithme* `min-conflicts`.

1er pas :
Assignation initiale : (x1 = B, x2 = A, x3 = B, x4 = A)
⇒ 2 conflits : c(x1,x3) et c(x2,x4)

variable changée	conflits avec	nombre total de conflits
x1 → C	c(x2,x4)	1
x2 → C	c(x1,x3)	1
x3 → C	c(x2,x4)	1
x4 → B	c(x1,x3),c(x1,x4)	2

accepter (x1 → C) : (x1 = C, x2 = A, x3 = B, x4 = A)
⇒ 1 conflit : c(x2,x4)

2e pas :

variable changée	conflits avec	nombre total de conflits
x1 → B	c(x1,x3), c(x2,x4)	2
x2 → C	c(x1,x2)	1
x3 → C	c(x1,x3),c(x2,x4)	2
x4 → B	-	0

accepter (x4 → B) : (x1 = C, x2 = A, x3 = B, x4 = B)
⇒ solution !

Fig. 8.14 *Solution d'un PSC par l'algorithme min-conflicts.*

la recherche vers des minima locaux. La figure 8.15 montre une trace où en acceptant un changement différent mais aussi valable que celui de la figure 8.14, l'algorithme tombe dans un minimum local où il n'y a aucun changement local qui apporte une réduction du nombre de conflits. C'est pour cette raison qu'on admet un certain nombre d'essais destinés à donner plusieurs chances de trouver une solution. Il semblerait par contre qu'en pratique, ces minima locaux ne présentent que peu de problèmes. La méthode de solution de PSC par min-conflicts est utilisée par exemple pour l'ordonnancement et la planification du télescope spatial HUBBLE.

1er pas :
Assignation initiale : (x1 = B, x2 = A, x3 = B, x4 = A)
⇒ 2 conflits : c(x1,x3) et c(x2,x4)

variable changée	conflits avec	nombre total de conflits
x1 → C	c(x2,x4)	1
x2 → C	c(x1,x3)	1
x3 → C	c(x2,x4)	1
x4 → B	c(x1,x3),c(x1,x4)	2

accepter (x2 → C) : (x1 = B, x2 = C, x3 = B, x4 = A)
⇒ 1 conflit : c(x1,x3)

2e pas :

variable changée	conflits avec	nombre total de conflits
x1 → C	c(x1,x2)	1
x2 → A	c(x1,x3),c(x2,x4)	2
x3 → C	c(x2,x3)	1
x4 → B	c(x1,x3),c(x1,x4)	2

aucun changement réduisant le nombre de conflits ⇒ fin

Fig. 8.15 *Un autre exemple où l'algorithme min-conflicts s'arrête dans un minimum local et ne trouve donc pas la solution, bien qu'elle existe.*

Notons finalement qu'on pourrait également utiliser l'heuristique min-conflicts pour un ordonnancement des valeurs dans la recherche incrémentale. Elle consisterait alors à considérer d'abord la valeur qui aura le moins de possibilités de conflits avec les variables qui restent encore à instancier.

Pour éviter le problème des minima locaux, la méthode du *recuit simulé* admet qu'on puisse accepter, avec une probabilité faible, des changements qui n'améliorent pas la qualité de la solution. Cet algorithme est inspiré de la physique et plus particulièrement du processus de la solidification des verres. Il utilise une température T qui donne la probabilité d'accepter un changement qui n'améliore pas la solution ; en choisissant $0 \leq T \leq 1$, on peut utiliser T comme une probabilité. Cette température, et donc la probabilité, est décroissante en fonction des itérations. La figure 8.16 montre l'algorithme, très similaire à l'algorithme min-conflicts. La figure 8.17 montre une trace d'exécution

Tableau T = liste de probabilités décroissantes

```
1: Function recuit(X,V,C)
2: V ← assignation aléatoire
3: T[max-steps] ← plan de réduction de "température"
4: for j ← 1 to max-steps do
5:    nconf ← check(V,C)
6:    if nconf = 0 then return V
7:    V' ← V avec une valeur v[k] changée aléatoirement
8:    if check(V',C) < check(V,C) then
9:       V ← V'
10:   else
11:      if random(0..1)<T[j] then V ← V'
12: return solution partielle V
```

Fig. 8.16 *Algorithme du recuit simulé.*

Assignation initiale : (x1 = B, x2 = A, x3 = B, x4 = A)
⇒ 2 conflits : c(x1,x3) et c(x2,x4)

changement	conflits	mieux ?	accepter ?	assignation
x2 → C	c(x1,x3)	oui	oui	B,C,B,A
x4 → B	c(x1,x3),c(x1,x4)	non	oui	B,C,B,B
x1 → C	c(x1,x2)	oui	oui	C,C,B,B
x3 → C	c(x1,x2),c(x1,x3),c(x2,x3)	non	non	C,C,B,B
x2 → A	-	oui	oui	C,A,B,B

Fig. 8.17 *Trace d'exécution du recuit simulé sur l'exemple de la figure 8.7.*

de l'algorithme, à nouveau sur l'exemple de la figure 8.7. Dans cet exemple, on a supposé que la température prend les valeurs suivantes, selon les itérations : T=(0.7,0.5,0.3,0.1,0.01,0.001).

8.4 Solution par propagation

Une autre manière de résoudre des problèmes de satisfaction de contraintes est par une *propagation* locale d'informations suivant la structure du réseau de contraintes. De telles méthodes, en général de complexité polynômiale dans la taille du problème, peuvent s'utiliser comme pré-traitement avant une recherche ou parfois même comme seul algorithme de solution.

Le terme *propagation de contraintes* est utilisé pour plusieurs différentes méthodes bien distinctes. Par exemple, il peut s'agir de :

- la propagation de valeurs,
- la relaxation de valeurs,
- la propagation de labels,
- la propagation de contraintes à proprement dit.

Clarifions un peu ces différentes notions.

Propagation de valeurs

Étant donné un PSC (X,D,C) où :

- certaines variables $\{x_i\}$ ont des valeurs v_i,
- les valeurs des autres variables $\{x_j\}$ sont inconnues.

On utilise les contraintes pour attribuer des valeurs consistantes aux $\{x_j\}$. Ceci est possible surtout avec des contraintes fonctionnelles, comme $x = y + z$. En général, les contraintes ne permettent pas de déduire des valeurs uniques. On doit alors faire un choix arbitraire de valeurs, et la convergence de la procédure n'est pas garantie.

Relaxation de valeurs

Étant donné un PSC (X,D,C) où :

- toutes les variables $\{x_i\}$ ont chacun une valeur v_i,
- les valeurs ne sont pas consistantes avec les contraintes.

On change incrémentalement les valeurs de certaines variables pour arriver à une solution consistante avec toutes les contraintes. La procédure peut suivre un régime de minimisation du nombre de conflits et revient alors à une optimisation. Les problèmes qui se posent sont :

- il n'est pas certain de trouver la meilleure solution,
- on ne peut pas distinguer des solutions multiples,
- il y a une forte possibilité de cycles.

Une variation de la relaxation est l'algorithme *Tabou*, dont l'élément important est qu'on mémorise des étapes précédentes pour éviter qu'il y ait des cycles.

Propagation de labels

Étant donné un PSC (X,D,C) où :

- chaque variable x_i a un *label* $l_i \subseteq D_i$,
- les labels indiquent les valeurs considérées comme encore consistantes.

On change itérativement les labels des variables pour éliminer toutes les valeurs ne pouvant être consistantes.

Le résultat d'une telle propagation est un ensemble de labels qui remplissent un certain degré de *consistance*. La consistance des nœuds est vérifiée si toutes les valeurs du label l_i respectent toutes les contraintes unaires $C(x_i)$. La consistance des arcs concerne des paires de deux variables x_i et x_j ainsi que la contrainte $C(x_i, x_j)$ les liant. Une valeur de $x_i = v_i \in l_i$ ne peut faire partie d'une solution du PSC que s'il existe au moins une valeur $x_j = v_j \in l_j$ telle que $C(v_i, v_j)$ soit respectée. On peut donc éliminer toutes les valeurs de l_i et l_j qui ne respectent pas la contrainte. Par application itérative sur toutes les contraintes, on obtient des labels qui satisfont la consistance des arcs.

```
1: Function REVISER (i,j,C)
2: modifiée ← faux
3: for chaque x ∈ l_i do
4:   if aucun y ∈ l_j tel que C(x,y) then
5:     l_i ← l_i \ x
6:     modifiée ← vrai
7:   return modifiée
```

Fig. 8.18 *Fonction élémentaire pour la consistance des arcs : REVISER(i,j,C_{ij}) effectue le raffinement de C_{ij} et sera appelée pour toutes les contraintes jusqu'à ce qu'il n'y ait plus de changement.*

Ceci est le fonctionnement de l'algorithme de Waltz, le premier algorithme à introduire la notion de propagation de contraintes. Il consiste à appliquer REVISER (i, j, C_{ij}) (fig. 8.18) à toutes les combinaisons de variables i et j jusqu'à ce que le résultat soit **faux** pour toutes les combinaisons. Par exemple, si nous avons le PSC suivant :

Variables : x_1, x_2, x_3
Domaines = labels initiaux
$l_1 = \{a, b, c, d\}$,
$l_2 = \{a, b, c, d\}$,
$l_3 = \{a, b, c, d\}$
Contraintes :
$\mathcal{C}_1(x_1, x_2) = \{(a, b), (a, c), (c, d)\}$,
$\mathcal{C}_2(x_1, x_3) = \{(a, b), (b, c), (c, d)\}$,
$\mathcal{C}_3(x_2, x_3) = \{(a, b), (a, c), (d, d)\}$

la propagation des labels effectuée par l'algorithme de Waltz sera la suivante :

1) $\mathcal{C}_1(x_1) \Rightarrow l_1 = \{a, c\}$
   $\mathcal{C}_1(x_2) \Rightarrow l_2 = \{b, c, d\}$
2) $\mathcal{C}_2(x_1) \Rightarrow l_1 = \{a, c\}$
   $\mathcal{C}_2(x_3) \Rightarrow l_3 = \{b, d\}$

3) $\mathcal{C}_3(x_2) \Rightarrow l_2 = \{d\}$
   $\mathcal{C}_3(x_3) \Rightarrow l_3 = \{d\}$
4) $\mathcal{C}_1(x_1) \Rightarrow l_1 = \{c\}$

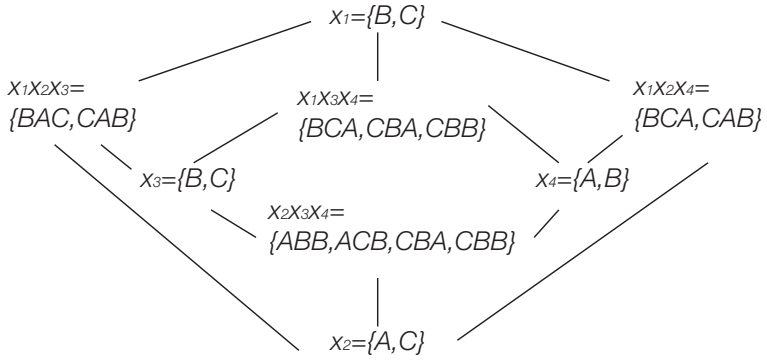
ce qui donne déjà une solution unique sans aucune recherche :

$$x_1 = c, x_2 = d, x_3 = d$$

La propagation de contraintes

Dans la propagation de contraintes à proprement parler, il s'agit de déterminer incrémentalement les contraintes implicites entre variables qui résultent de la composition des contraintes d'un PSC. À partir de contraintes à k variables, on construit donc des contraintes à (k+1) variables qui représentent les combinaisons consistantes de valeurs de ces (k+1) variables. On termine avec une seule contrainte qui englobe toutes les variables et représente toutes les solutions globalement consistantes. Cette opération ressemble fortement au « join » connu dans les bases de données.

Comme exemple, considérons le PSC que montre la figure 8.3. Il contient uniquement des contraintes à 2 variables, la propagation des contraintes construit d'abord un PSC dont les contraintes impliquent 3 variables :



et ensuite une seule contrainte à 4 variables :

$$C(x_1, x_2, x_3, x_4) = \{(C, A, B, B)\}$$

Dans le cas d'un PSC à k variables, la procédure s'arrêtera avec une seule contrainte à k variables qui contient toutes les solutions. On appelle alors le PSC *(k-1)-consistant.*

8.5 Consistance et complexité de la recherche

Si une utilisation des algorithmes de propagation peut être de trouver directement une solution à un PSC, une autre possibilité est de les utiliser pour un

pré-traitement du problème avant ou pendant sa solution par recherche. Ceci est notamment utile pour les algorithmes de consistance. Il est évident que ces méthodes peuvent servir à réduire le temps que met un algorithme de la recherche pour trouver une solution. Maintenant, nous allons établir des relations précises entre la forme d'un réseau de contraintes, le degré de consistance et la complexité de recherche nécessaire pour trouver une solution.

Trouver toutes les solutions à un PSC général est un problème de complexité exponentielle, puisque le nombre de solutions qu'admet le PSC peut être en croissance exponentielle avec le nombre de variables du problème ; dans ce cas, la complexité de les énumérer toutes ne peut pas être moins qu'exponentielle. Par contre, pour trouver *une* seule solution, on peut espérer de limiter la complexité à un temps polynomial.

Nous allons considérer les notions de consistance suivantes :

- consistance des nœuds : toutes les valeurs admissibles pour un nœud satisfont l'ensemble des contraintes unaires sur ce nœud ;
- consistance des arcs : pour chacune des valeurs admissibles pour un nœud x_i, il existe des valeurs admissibles pour les autres nœuds x_j telle que chaque contrainte entre x_i et x_j est satisfaite.

Pour chacune de ces notions de consistance, nous allons examiner la complexité de recherche d'*une* solution dans un réseau qui remplit cette consistance.

8.5.1 Consistance des nœuds

Si les variables associées au problème prennent leurs valeurs dans un domaine discret, l'ensemble des valeurs admissibles pour une variable peut être représenté sous la forme d'une liste. Pour satisfaire les contraintes unaires (consistance des nœuds), il suffit alors de disposer d'un algorithme qui élimine simplement de ces listes les valeurs engendrant une inconsistance. Par contre, les garanties que peut donner la consistance de nœuds ne sont pas très fortes : pour le cas d'un réseau sans contraintes entre variables, la consistance des nœuds donne une garantie que n'importe quelle combinaison de valeurs est une solution admissible.

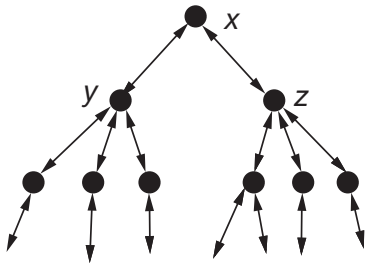
8.5.2 Consistance des arcs

L'algorithme le plus simple pour réaliser la consistance des arcs est l'algorithme de Waltz. Il consiste à appliquer itérativement une procédure `REVISER` (fig. 8.18) à toutes les contraintes jusqu'au moment où aucune modification ne peut être apportée (état stationnaire). L'algorithme est terminé quand les appels à `REVISER` ont retourné **faux** pour toutes les paires de nœuds (i, j).

La complexité de l'algorithme de Waltz peut être estimée comme suit. Considérons m contraintes, n variables dont les domaines ont la taille maximale d. À chaque itération, `REVISER` est appliquée au plus $2m$ fois, une fois dans chaque direction. Pour que l'itération ne s'arrête pas, il faut enlever au moins une valeur. Il ne peut donc pas y avoir plus que $n \cdot d$ itérations. Donc, la complexité totale ne peut pas dépasser $O(m \cdot n \cdot d)$.

La propagation de Waltz peut être facilement généralisée à des contraintes à n variables : `REVISER` doit alors veiller à ce que pour chaque valeur du label L_i, il existe toujours une combinaison de valeurs admises par les labels des autres $(n-1)$ variables pour que la contrainte soit respectée.

Pour des réseaux de contraintes *binaires* qui ont la topologie d'un arbre, c'est-à-dire qui n'ont pas de cycles, la consistance des arcs garantit qu'on peut trouver une solution de manière très efficace par une recherche sans retour-arrière :



En commençant par un nœud initial quelconque x, on affecte des valeurs aux variables couche par couche. La consistance des arcs garantit alors qu'à chaque étape le label d'une variable w contient au moins une valeur qui est consistante avec la valeur qui a été choisie pour son unique nœud parent. Une solution peut ainsi être trouvée avec un temps de calcul linéaire en fonction du nombre de variables. Même si cette classe de PSC n'est pas la seule qui peut être résolue en temps polynomial, elle est la plus importante car elle est facile à caractériser. Il existe d'ailleurs des méthodes de transformation qui permettent de regrouper des variables d'un PSC dans des méta-variables qui alors forment un arbre entre elles. Un tel *clustering* permet de réduire la complexité de la recherche qui n'est désormais exponentielle que dans la taille des clusters.

8.6 Contraintes globales

Dans le cas de grands problèmes pratiques, on rencontre souvent des structures de contraintes relativement régulières qui créent une grande complexité lors de la résolution par recherche. Considérons par exemple un problème d'allocation de ressources où il s'agit d'assigner quatre tâches simultanées $x_1..x_4$ aux ressources $\{a, b, c, d, e\}$ en respectant les domaines suivants :

- $x_1 \in \{a, b, c, d\}$
- $x_2 \in \{a, b\}$
- $x_3 \in \{a, b, c\}$
- $x_4 \in \{a, c\}$

La contrainte qu'aucune ressource ne puisse être assignée à deux tâches peut être représentée par un ensemble de contraintes binaires d'inégalité entre les variables. Cependant, cela conduit à un grand effort de recherche, parce que

pour les trois premières valeurs de x_1, c.à.d. a, b et c, l'algorithme doit effectuer la recherche jusqu'à x_4 pour trouver qu'il n'y a aucune solution. Les algorithmes de consistance locale n'effectuent aucune réduction avant la dernière étape et ne sont donc pas efficaces.

Une alternative serait de reconnaître qu'on trouve ici une structure spéciale qu'on peut appeler `alldifferent` (tous différents). On peut alors formuler ainsi une contrainte additionnelle que l'on nomme *contrainte globale* :

`alldifferent`(x_1, x_2, x_3, x_4)

L'intérêt de cette contrainte globale est que l'on peut formuler un algorithme de propagation spécialisé et plus efficace qui la prend en compte.

Représentons la structure d'une contrainte `alldifferent` par un graphe qui contient comme nœuds :

- les variables,
- les valeurs de leurs domaines.

et un arc qui lie chaque variable avec toutes les valeurs de son domaine (voir fig. 8.19).

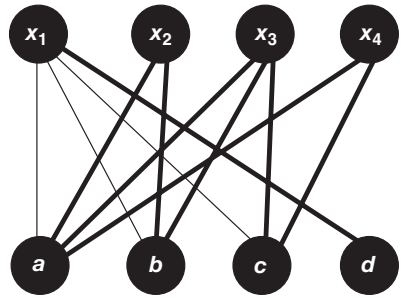


Fig. 8.19 *Représentation de la contrainte* `alldifferent` *comme graphe. Les arcs qui peuvent apparaître dans un matching maximal sont indiquées en gras.*

Une assignation de valeurs aux variables peut être représentée par un *matching* de taille maximale qui associe tous les nœuds variables aux nœuds valeurs. Il existe un algorithme efficace qui permet de trouver tous les arcs qui peuvent participer à un tel matching. Pour l'exemple, ils sont identifiées par des traits gras dans la figure 8.19.

Un tel filtrage permet d'éliminer dès le départ les valeurs a, b et c du domaine de x_1, et donc de trouver une solution de manière beaucoup plus efficace. Il pourrait également s'appliquer lors de la recherche, par exemple pour un *forward checking*, et ainsi obtenir des gains de performance impressionnants.

Pour profiter de telles possibilités, les outils de programmation par contraintes mettent à disposition de nombreuses contraintes globales dont la propagation est implémentée par des algorithmes spécialisés. Citons par exemple le *global cardinality constraint*, qui permet d'indiquer que chaque valeur doit être assignée à au moins x et au plus y variables parmi un certain ensemble. D'autres exemples sont de nombreuses contraintes de cycles qui se réfèrent à l'existence et à des propriétés de cycles dans les assignations des variables.

8.7 Satisfiabilité

Le problème de la satisfiabilité (SAT) se définit comme suit :

Étant donné

- un ensemble de *littéraux* $x_1, ..., x_n$
- et un ensemble de *clauses* $\pm x_i \vee \pm x_j \vee ... \vee \pm x_k$,

trouver une assignation de valeurs $x_i = v_i \in \{vrai, faux\}$ pour tous les x_i tel que toutes les clauses soient vraies.

Pour des clauses de longueur trois ou plus, la satisfiabilité est un problème de complexité NP.

Il est évident que la satisfiabilité est un cas spécial de problème de satisfaction de contraintes. Cependant, il existe deux simplifications : toutes les variables ont le même domaine de deux valeurs uniquement, et les contraintes ont une forme restreinte. On peut ainsi utiliser des méthodes spécifiques pour la résolution de ces problèmes.

Comme pour les PSC, il existe des méthodes complètes, basées sur la recherche et l'inférence, et des méthodes itératives qui sont incomplètes. Elles fonctionnent essentiellement comme les PSC.

Un algorithme de recherche construit une assignation de valeurs à toutes les variables (littéraux) de façon incrémentale. Cependant :

- On remplace le *forward checking* par *unit propagation*. Suite à l'assignation d'une valeur v_i à une variable x_i, on peut substituer cette valeur dans tous les clauses. Il y aura alors :
  - des clauses qui contiennent x_i et où la valeur v_i satisfait la clause ; ces clauses sont désormais satisfaites et ne doivent plus être considérées ;
  - des clauses qui contiennent x_i, et où la valeur v_i *ne* satisfait *pas* la clause ; on peut éliminer x_i de la clause ; si elle ne contient désormais aucune variable, on a trouvé une contradiction qui nécessite un backtrack ;
  - des clauses qui ne contiennent pas x_i et ne sont donc pas affectées.

  Cette propagation est en général beaucoup plus forte que dans le cas d'un PSC.
- On remplace également le *dynamic variable ordering* par une heuristique qui sélectionne la prochaine variable basée sur son occurrence dans les clauses qui ne sont pas encore satisfaites. Par exemple, on prend la variable qui apparaît dans le plus grand nombre de clauses pas encore satisfaites.

Un algorithme itératif commence avec une assignation aléatoire de valeurs aux variables. Pour améliorer le nombre de conflits, un changement d'assignation doit forcement affecter une clause qui n'est pas satisfaite. Une technique utilisée peut consister à sélectionner une clause non satisfaite et à changer l'assignation d'une des variables. On répète ce processus jusqu'à ce qu'une solution soit trouvée. Cette méthode ne peut pas tomber dans des optima locaux, mais elle peut résulter en une boucle infinie. Pour éviter cela, il suffit de choisir les

clauses à changer de façon aléatoire : un boucle infinie apparaîtra alors avec une probabilité très petite. La méthode peut donc être complète. Par contre, si le problème n'admet pas de solution, elle ne s'arrêtera jamais.

8.8 Optimisation sous contraintes

Le paradigme de la satisfaction de contraintes peut être étendu à celui de l'optimisation en exigeant que la solution trouvée ne soit non seulement conforme à toutes les contraintes, mais également la meilleure suivant un certain critère d'optimisation. En fait, la recherche de solution optimale se retrouve dans presque toutes les applications pratiques. Des méthodes d'optimisation font donc partie de pratiquement tous les outils de programmation par contraintes.

Du point de vue formel, le critère d'optimisation est normalement défini comme une fonction de coût qui se calcule comme la somme des valeurs de diverses *relations*. On peut ainsi compléter la definition d'un PSC comme suit :

Étant donné :

- les variables $\mathrm{X} = x_1, x_2, ..., x_n$;
- les domaines $\mathrm{D} = D_1, D_2, ..., D_n$ associés aux variables ;
- les contraintes $\mathrm{C} = C_1(x_k, x_l, ...), C_2, ..., C_m$, qui restreignent les combinaisons de valeurs possibles pour les variables sur lesquelles elles portent ;
- les relations $\mathrm{R} = R_1(x_k, x_l, ...), R_2, ..., R_l$, où chacune donne un coût pour chaque combinaison de valeurs des arguments ;

Trouver :

la solution :
$\{x_1 = v_k, x_2 = v_l, ..., x_n = v_o\}$ telle que toutes les contraintes soient satisfaites et que la somme $R_1 + R_2 + ... + R_l$ soit minimale parmi tous les assignations consistantes.

L'optimisation sous contraintes est une alternative plus générale à la programmation linéaire. Si la programmation linéaire et ses variantes discrètes développées en recherche opérationnelle présentent un avantage en efficacité, l'optimisation sous contraintes admet des contraintes et des relations arbitraires. Cette dernière est donc plus généralement applicable.

Il existe parfois des problèmes où des algorithmes efficaces permettent d'optimiser l'un ou l'autre aspect isolé, mais pas leur combinaison. Par exemple, dans l'optimisation de transports, il peut être important de chercher à la fois le chemin le plus court et un ordonnancement optimal. L'optimisation sous contraintes permet de combiner ces deux critères.

L'algorithme de recherche en profondeur-d'abord pour la solution de PSC peut être adapté à l'optimisation en introduisant le mécanisme de *branch-and-bound*. Le principe est le suivant :

- On maintient pour chaque nœud de recherche (c.à.d. une assignation partielle) une borne inférieure BI(n) sur le coût d'une assignation consistante

qui contient l'assignation partielle. La borne inférieure se calcule comme la somme des coûts de toutes les relations qui peuvent être évaluées étant donné les assignations déjà faites. Pour une relation où uniquement une partie des variables a une valeur, on compte le coût le plus bas qui est possible étant donné les assignations déjà fixées.

- L'algorithme maintient également une borne supérieure BS qui est initialisée à $+\infty$. Chaque fois qu'on trouve une assignation complète, si le coût c est inférieur à BS, on met à jour $BS \leftarrow c$, et on mémorise l'assignation comme la solution provisoire.
- Tout nœud n tel que $BI(n) \geq BS$ ne peut pas conduire à une meilleure solution que la solution provisoire et est donc abandonné.
- S'il ne reste aucun nœud non exploré, la solution provisoire est la solution optimale avec coût BS.

Il est également possible de généraliser à l'optimisation les techniques de *forward checking*, de *lookahead* et de la consistance des arcs.

L'adaptation des algorithmes itératifs pour trouver une solution optimale est plus simple : au lieu de minimiser uniquement les violations de contraintes, on associe à chaque violation de contrainte un coût qui se rajoute au coût de la solution. On choisit alors les modifications qui réduisent le plus possible le coût de la solution.

Une variante de l'optimisation sous contraintes est la satisfaction partielle de contraintes. Dans cette variante, on considère que toute contrainte peut être violée avec un certain coût. On cherche alors la solution qui minimise le coût de tous les violations. On peut transformer un tel problème en un problème d'optimisation sous contraintes en considérant chaque contrainte comme une relation qui aurait un coût de zéro pour toutes les assignations admises et un coût de c pour toutes celles qui ne sont pas admises. On parle alors aussi de préférences ou de *soft constraints*.

Littérature

Plusieurs livres sur la programmation par contraintes sont apparus récemment. La collection [32] est la plus complète et couvre pratiquement tous les sujets. Les livres [33] et [34] présentent des approches plus spécifiques, le premier avec un accent sur la programmation dynamique et le deuxième avec un accent sur la programmation logique par contraintes.

L'idee de la consistance locale et plus particulièrement la consistance des arcs a été introduit dans [35]. La combinaison du *forward checking* et *dynamic variable ordering* a été introduite en 1980 dans [36], et le fait que la consistance des arcs garantit une recherche sans retour-arrière a été démontré dans [37] et développé dans [38]. On ne connaît pas l'inventeur du principe de la recherche locale, mais le recuit simulé a été introduit dans [39] et le GSAT dans [40].

Outils - domaine public

Il existe de nombreux outils pour la satisfaction de contraintes, mais beaucoup sont vieux ou mal documentés. Le système `Choco` est un développement récent qui est beaucoup utilisé :

`http://choco.sourceforge.net/`

Application : Gestion de production d'automobiles

Appartenant à la société Nissan, l'usine de Sunderland (UK) était déjà considérée comme l'usine de voitures la plus efficace d'Europe. L'usine fabriquait deux modèles sur ses deux chaînes de production.
En 1999, Nissan voulait y produire un troisième modèle, mais sans augmenter le nombre de chaînes de production. On a réussi à le faire grâce à l'introduction d'un outil d'ordonnancement basé sur la satisfaction de contraintes. La technologie a permis de gérer les contraintes qui résultent du fait que trois modèles sont produits simultanément sur deux chaînes de production, ce qui n'était pas possible avec les outils de la recherche opérationnelle.
Nissan a ainsi augmenté le nombre de voitures produit par année de 236 000 a 337 000 sans introduire de nouveaux équipements. L'alternative, la construction d'une troisième chaîne de production, aurait coûté plus de cinq cent millions de dollars ! De plus, avec le nouveau système, la planification est beaucoup plus souvent respectée : si, avant, seulement 3% des voitures étaient produites selon le plan, en utilisant la satisfaction de contraintes, ce taux est passé à 95%.

(Source : ILOG SA : Success Story : Nissan, `http://www.ilog.com`)

8.9 Exercices

Exercice 8.1 Première partie - Algorithme de Backtrack

Dans cette série d'exercices, vous allez programmer des algorithmes destinés à résoudre des problèmes de satisfaction de contraintes :

- la consistance de nœuds et d'arcs,
- l'algorithme de recherche par backtracking.

Dans la série suivante, vous appliquerez ces algorithmes au jeu du Sudoku.

Modules squelettes

Les modules `moteur_psc` suivants contiennent le squelette du programme que nous allons développer. Le module `exemple_backtracking.py` permettra de le tester.

Module `.../moteur_psc/variable.py` :

```
class Variable:
    def __init__(self, nom, domaine, val=None):
        self.nom = nom
        self.domaine = domaine
        self.val = val

    def taille_domaine(self):
        return len(self.domaine)

    def __eq__(self, that):
        return self.nom == that.nom

    def __hash__(self):
        return sum(map(ord, self.nom))

    def __repr__(self):
        return '{} = {}, domaine: {}'.format(self.nom, self.val, self.domaine)
```

Module `.../moteur_psc/contrainte.py` :

```
class Contrainte:
    def __init__(self, variables):
        self.variables = tuple(variables)

    def dimension(self):
        return len(self.variables)

    def est_valide(self):
        return False

    def __repr__(self):
        return 'Contrainte: {}'.format(self.variables)

    def __eq__(self, that):
        return self.variables == that.variables

    def __hash__(self):
        return sum([v.__hash__ for v in self.variables])

class ContrainteUnaire(Contrainte):
    def __init__(self, var, op):
        Contrainte.__init__(self, (var,))
        self.op = op

    def est_valide(self, val):
        print('à compléter')

class ContrainteBinaire(Contrainte):
    def __init__(self, var1, var2, op):
        Contrainte.__init__(self, (var1, var2))
        self.op = op

    def est_valide(self, var, val):
        print('à compléter')

    def est_possible(self, var):
        print('à compléter')
```

```
    def reviser( self ):
        print('à compléter')
```

Module `.../moteur_psc/moteur_psc.py` :

```
class PSC:
    def __init__ ( self ,  variables ,  contraintes ):
        self . variables  = variables
        self . contraintes  = contraintes

        self . iterations  = 0
        self . solutions  = []

    def consistance_noeuds( self ):
        print('à compléter')

    def consistance_arcs ( self ):
        print('à compléter')

    def consistance_avec_vars_precedentes ( self ,  k ):
        print('à compléter')

    def backtracking( self ,  k=0, une_seule_solution=False):
        print('à compléter')

    def  affiche_solutions ( self ):
        print('Recherche terminée en {} itérations ' . format(self. iterations ))

        if  len( self . solutions ) == 0:
            print('Aucune solution trouvée')
            return

        for  sol  in  self . solutions :
            print('Solution ')
            print('========')
            for  (nom, var) in sorted(sol.items ()):
                print('\tVariable  {}:  {}' . format(nom, var))
```

Module `.../exemple_backtracking.py` :

```
from moteur_psc.variable import Variable
from moteur_psc.contrainte import ContrainteUnaire, ContrainteBinaire
from moteur_psc.psc import PSC

variables  = [
              Variable('a',  [2,  3]),
              Variable('b',  list (range(12))),
              Variable('c',  list (range(3))),
              Variable('d',  list (range(3))),
              Variable('e',  list (range(12))),
              ]

contraintes  = [
    ContrainteUnaire(variables [1],  lambda x: x < 4),
    ContrainteBinaire(variables [0],  variables [1],  lambda x, y: x != y),
    ContrainteBinaire(variables [1],  variables [2],  lambda x, y: x != y),
```

```
    ContrainteBinaire(variables [1], variables [3], lambda x, y: x != y),
    ContrainteBinaire(variables [1], variables [4], lambda x, y: x != y),
    ContrainteBinaire(variables [2], variables [3], lambda x, y: x != y),
    ContrainteBinaire(variables [2], variables [4], lambda x, y: x != y),
    ContrainteBinaire(variables [3], variables [4], lambda x, y: x != y),
    ContrainteBinaire(variables [4], variables [0], lambda x, y: x < y),
]

psc = PSC(variables, contraintes)

psc.consistance_noeuds()
psc. consistance_arcs ()
psc.backtracking(0, False)

psc. affiche_solutions ()
```

Exercice 8.1.1 Consistance des nœuds et des arcs
Les modules `variable.py` et `contrainte.py` contiennent les classes `Variable` et `Contrainte`, ainsi que les sous-classes de cette dernière, `ContrainteUnaire` et `ContrainteBinaire`. Le module `psc.py` implémente la classe `PSC` - une librairie pour la gestion d'un ensemble de variables et de contraintes, ainsi que pour la résolution d'un système de contraintes.

Comme vous pouvez le constater, les fonctions `consistance_noeuds` et `consistance_arcs` la classe `PSC` ne sont pas implémentées. Vous pouvez commencer par compléter ces fonctions, en suivant les indications suivantes :

- La fonction `consistance_noeuds` doit appeler la méthode `est_valide` de la classe `ContrainteUnaire`, que vous devez aussi compléter.
- La fonction `consistance_arcs` doit appeler la méthode `reviser` de la classe `ContrainteBinaire`, dans laquelle vous devez implémenter l'algorithme de Waltz (consistance d'arcs). Comme les contraintes binaires sont bidirectionnelles, vous devrez implémenter la fonction `reviser` de telle sorte que les domaines des deux variables de la contrainte soient réduits (si possible) par l'appel à `reviser`.
- À son tour, la méthode `reviser` s'appuie sur les méthodes `est_valide` et `est_possible` de la classe `ContrainteBinaire`, que vous devez aussi compléter.

Exercice 8.1.2 Algorithme du backtrack
Le Backtracking est un algorithme de recherche en profondeur-d'abord avec les charactéristiques suivantes :

- un nœud de recherche est une instanciation de variables $x_1 = v_1$, $x_2 = v_2$, ..., $x_k = v_k$ (où k est la profondeur du nœud dans l'arbre de recherche),
- la fonction de successeur ajoute une nouvelle instanciation $x_{k+1} = v_{k+1}$ de manière à respecter toutes les contraintes pour les variables x_1, ..., x_k,
- le nœud initial est une instanciation vide,
- un nœud but consiste en une instanciation de toutes les variables x_1, ..., x_n.

L'algorithme de recherche, dont nous vous donnons le pseudo-code ci-dessous, doit ainsi être implémenté dans la méthode `backtracking` de la classe `PSC` :

```
solutions <− []
variables <− [v1, v2, ..., vn]

Backtracking(k, une_seule_solution)
1.  IF k >= n THEN
2.      IF NOT une_seule_solution THEN
3.          ajouter la solution actuelle à solutions
4.      ELSE
5.          RETURN solutions = [solution actuelle]
6.      END IF
7.  ELSE
8.      v <− variables[k]
9.      FOR EACH valeur de domaine d de la variable v DO
10.         assigner la valeur d à la variable v
11.         vé rifier la consistance de v=d avec les variables précédentes
12.         IF v=d est consistant THEN
13.             reste <− Backtracking(k+1, une_seule_solution)
14.             IF reste != échec THEN
15.                 RETURN reste
16.             END IF
17.         END IF
18.     END FOR
19. END IF
20. RETURN échec
END Backtracking
```

`backtracking` prend deux paramètres :

- `k` : la profondeur courante (commence à 0),
- `une_seule_solution` : si vrai, alors retourne la première solution trouvée, sinon retourne toutes les solutions possibles.

Les étapes 11 et 12 de l'algorithme ci-dessus seront implémentées à l'aide de la fonction `consistance_avec_vars_precedentes` de la classe `PSC`, que vous devez aussi compléter.

Les solutions seront stockées dans la variable de classe `self.solutions`, chacune étant représentée par un dictionnaire qui associera le nom de la variable à sa valeur. Comme ces solutions sont conservées dans un champ de la classe, il n'est donc pas indispensable de les retourner. En outre, au lieu de retourner une valeur spéciale en cas d'échec, la méthode `backtracking` peut se terminer simplement sans valeur de retour.

5

Test du programme

Une fois que vous avez terminé, vous pouvez tester votre implémentation sur le module `exemple_backtracking.py` :

```
python3 exemple_backtracking.py
```

Exercice 8.1.3 Permutations

S'il vous reste du temps, vous pouvez utiliser l'algorithme du backtrack pour afficher toutes les permutations possibles des entiers de 0 à N. Par permutation, nous entendons une solution dont les clés seront des indices et les valeurs, des entiers de 0 à N. Chacune de ces solutions doit contenir chaque valeur une fois et une fois seulement. Vous devrez donc définir un système de contraintes pour vous assurer que cette condition soit respectée. Testez votre implémentation pour plusieurs valeurs de N. (Ne choisissez pas des valeurs trop élevées pour N, faute de quoi l'algorithme risque de tourner pendant des heures.)

Essayez ensuite d'imaginer d'autres contraintes pour restreindre l'ensemble des solutions et implémentez-les. Par exemple, on peut exiger que deux valeurs consécutives ne soient jamais toutes deux paires ou toutes deux impaires.

Solutions à la page 370

Exercice 8.2 Deuxième partie - Forward Checking

Dans cette série, vous allez programmer de nouveaux algorithmes de résolution de systèmes de contraintes, plus sophistiqués que ceux de la leçon précédente :

- les heuristiques du *Variable Ordering* et du *Dynamic Variable Ordering*,
- programmer l'heuristique du *Forward Checking*.

Vous les testerez ensuite sur le jeu du Sudoku.

Modules squelettes

Commencez par copier les modules suivants, qui implémentent partiellement le code de l'exercice. Le module `moteur_psc_heuristique` fournit le squelette du programme que nous allons développer. Les modules `exemple_forward_checking.py` et `exemple_sudoku.py` sont là pour vous permettre de tester votre programme.

Module `.../moteur_psc_heuristique/variable_avec_label.py` :

```
from moteur_psc.variable import Variable

class VariableAvecLabel(Variable):
    def __init__(self, nom, domaine, val=None):
        Variable.__init__(self, nom, domaine, val)

        # Le label est initialement égal au domaine.
        self.label = domaine[:]

    def __repr__(self):
        return '{} = {}, domaine: {}, label: {}'.format(self.nom,
                                                        self.val,
                                                        self.domaine,
                                                        self.label)
```

Module `.../moteur_psc_heuristique/contrainte_avec_propagation.py` :

```
from moteur_psc.contrainte import ContrainteBinaire

class ContrainteAvecPropagation(ContrainteBinaire):
    def __init__(self, var1, var2, op):
        ContrainteBinaire.__init__(self, var1, var2, op)

    def reviser(self):
        # Nous appliquons d'abord la méthode reviser() de la classe-mère pour
        # réviser les domaines de chaque variable.
        domaines_modifies = ContrainteBinaire.reviser(self)

        # Puis, s'il y a lieu, nous nous assurons que les labels sont toujours
        # identiques aux domaines.
        if domaines_modifies:
            for var in self.variables:
                var.label = var.domaine[:]

        return domaines_modifies


    def propage(self, var):
        print('à compléter')
```

Module `.../psc_heuristique/moteur_psc_heuristique.py` :

```
from moteur_psc.psc import PSC

class PSCHeuristique(PSC):

    def __init__(self, variables, contraintes):
        PSC.__init__(self, variables, contraintes)

        self.reinitialise()

    def reinitialise(self):
        self.initialise_labels()
        self.solutions = []
        self.iterations = 0

    def initialise_labels(self):
        for var in self.variables:
            var.label = var.domaine[:]

    def consistance_noeuds(self):
        # Nous appelons d'abord la méthode de la classe-mère PSC pour réduire
        # les domaines.
        PSC.consistance_noeuds(self)

        # Puis, nous nous assurons que les labels sont identiques aux domaines.
        self.initialise_labels()

    def variable_ordering(self):
        print('à compléter')

    def dynamic_variable_ordering(self, k):
        print('à compléter')
```

```
    def propagation_consistante(self, k):
        print('à compléter')

    def forward_checking(self, k=0, une_seule_solution=False):
        print('à compléter')
```

Module `.../sudoku.py` :

```
from moteur_psc_heuristique.variable_avec_label import VariableAvecLabel
from moteur_psc.contrainte import ContrainteUnaire
from moteur_psc_heuristique.contrainte_avec_propagation import
        ContrainteAvecPropagation
from moteur_psc_heuristique.psc_heuristique import PSCHeuristique

class Sudoku:
    def __init__(self, grille, taille=9, sous_taille=3):
        if taille % sous_taille != 0:
            raise ValueError('Taille et sous-taille de grille incompatibles.')
        self.taille = taille
        self.sous_taille = sous_taille

        # Génère une variable par case.
        self.variables = [VariableAvecLabel('{}{}'.format(i, j),
                                            list(range(1, self.taille + 1)))
            for i in range(self.taille) for j in range(self.taille)]

        # Initialise les cases dont les valeurs sont connues.
        for i in range(self.taille):
            for j in range(self.taille):
                # Si la case est instanciée avec une valeur valide,
                # assigne la variable et restreint son domaine à la valeur initiale.
                if isinstance(grille[i][j], int):
                    if not (1 <= grille[i][j] <= self.taille):
                        raise ValueError('Valeur invalide dans la grille de départ.')
                    var = self.variables[i * self.taille + j]
                    var.val = grille[i][j]
                    var.domaine = [grille[i][j]]
                    var.label = [grille[i][j]]

        self.contraintes = []
        self.genere_contraintes()

    def __genere_contraintes_sous_grille(self, x, y):
        # Parcours de la sous-grille.
        for i in range(x, x + self.sous_taille):
            for j in range(y, y + self.sous_taille):
                # Pour chaque case qui n'est ni dans la même ligne i ni dans la même
                # colonne j, on ajoute une contrainte. (les autres cases sont couvertes
                # par les contraintes de lignes et de colonnes.)
                for k in range(x, x + self.sous_taille):
                    for l in range(y, y + self.sous_taille):
                        if i != k and j != l:
                            self.contraintes.append(
                                ContrainteAvecPropagation(self.variables[i * self.
                                                                         taille + j],
                                                          self.variables[k * self.
                                                                         taille + l],
                                                          lambda x,y: x != y)
```

```
                        )

def genere_contraintes(self):
    self.contraintes = []

    for i in range(0, self.taille):
        for j in range(0, self.taille):
            # Contraintes sur les case d'une ligne.
            for k in range(j + 1, self.taille):
                self.contraintes.append(
                    ContrainteAvecPropagation(self.variables[i * self.taille + j],
                                              self.variables[i * self.taille + k],
                                              lambda x,y: x != y)
                )
            # Contraintes sur les cases d'une colonne.
            for k in range(j + 1, self.taille):
                self.contraintes.append(
                    ContrainteAvecPropagation(self.variables[j * self.taille + i],
                                              self.variables[k * self.taille + i],
                                              lambda x,y: x != y)
                )

    # Contrainte sur les cases d'une sous-grille.
    # Le troisième argument de range permet de régler l'incrément.
    # Ex.: range(0, 5, 2) génère la séquence 0, 3.
    #      range(10, 5, -1) génère la séquence 10, 9, 8, 7, 6.
    # '//' est l'opérateur de division entière.
    for i in range(0, self.taille, self.taille // self.sous_taille):
        for j in range(0, self.taille, self.taille // self.sous_taille):
            self._genere_contraintes_sous_grille(i, j)

def resoudre(self, methode):
    psc = PSCHeuristique(self.variables, self.contraintes)
    psc.consistance_noeuds()
    psc.consistance_arcs()

    if methode == 'forward_checking':
        psc.forward_checking(une_seule_solution=True)
    elif methode == 'backtracking':
        psc.variable_ordering()
        psc.backtracking(une_seule_solution=True)
    else:
        raise ValueError('Méthode inconnue: ' + str(methode))
    print('Méthode: ' + methode)
    print('Recherche terminée en {} itérations'.format(psc.iterations))
    for i in range(self.taille):
        for j in range(self.taille):
            nom = '{}{}'.format(i, j)
            self.variables[i * self.taille + j].val = psc.solutions[0][nom]

def __repr__(self):
    def val(e):
        if e is None:
            return '-'
        else:
            return e

    ret = ''
    for i in range(self.taille):
```

```
            for j in range(self. taille ):
                ret += '{} '.format(val(self. variables [i * self . taille + j]. val))
            ret += '\n'

        return ret
```

Module `.../exemple_forward_checking.py` :

```
from moteur_psc_heuristique.variable_avec_label import VariableAvecLabel
from moteur_psc.contrainte import ContrainteUnaire
from moteur_psc_heuristique.contrainte_avec_propagation import
        ContrainteAvecPropagation
from moteur_psc_heuristique.psc_heuristique import PSCHeuristique

variables = [
            VariableAvecLabel('a', [2, 3]),
            VariableAvecLabel('b', list (range(12))),
            VariableAvecLabel('c', list (range(3))),
            VariableAvecLabel('d', list (range(3))),
            VariableAvecLabel('e', list (range(12))),
            ]

contraintes = [
    ContrainteUnaire(variables [1], lambda x: x < 4),
    ContrainteAvecPropagation(variables[0], variables [1], lambda x, y: x != y),
    ContrainteAvecPropagation(variables[1], variables [2], lambda x, y: x != y),
    ContrainteAvecPropagation(variables[1], variables [3], lambda x, y: x != y),
    ContrainteAvecPropagation(variables[1], variables [4], lambda x, y: x != y),
    ContrainteAvecPropagation(variables[2], variables [3], lambda x, y: x != y),
    ContrainteAvecPropagation(variables[2], variables [4], lambda x, y: x != y),
    ContrainteAvecPropagation(variables[3], variables [4], lambda x, y: x != y),
    ContrainteAvecPropagation(variables[4], variables [0], lambda x, y: x < y),
]

psc = PSCHeuristique(variables, contraintes)

psc.consistance_noeuds()
psc. consistance_arcs ()
psc. variable_ordering ()

psc.backtracking()

print('Backtracking avec variable ordering: ')
psc. affiche_solutions ()

psc. reinitialise ()
psc.forward_checking()

print('Forward checking: ')
psc. affiche_solutions ()
```

Module `.../exemple_sudoku.py` :

```
from sys import argv, exit
from moteur_psc_heuristique.variable_avec_label import VariableAvecLabel
from moteur_psc.contrainte import ContrainteUnaire
from moteur_psc_heuristique.contrainte_avec_propagation import
```

```
        ContrainteAvecPropagation
from moteur_psc_heuristique.psc_heuristique import PSCHeuristique
from sudoku import Sudoku

grilleA = [
    [ 9 ,'-','-','-','-','-','-','-', 2],
    [ 3 ,'-', 7 , 1 ,'-','-', 4 ,'-', 8],
    ['-', 1 ,'-','-', 5 , 4 ,'-', 6 ,'-'],
    ['-','-', 1 ,'-','-','-','-', 7 ,'-'],
    ['-','-', 4 ,'-','-','-', 9 ,'-','-'],
    ['-', 2 ,'-','-','-','-', 8 ,'-','-'],
    ['-', 8 ,'-', 3 , 2 ,'-','-', 4 ,'-'],
    [ 7 ,'-', 3 ,'-','-', 6 , 2 ,'-', 1],
    [ 4 ,'-','-','-','-','-','-','-', 5]
]

grilleB = [
    ['-','-','-','-','-','-','-','-','-'],
    ['-','-', 7 , 8 , 3 ,'-', 9 ,'-','-'],
    ['-','-', 5 ,'-','-', 2 , 6 , 4 ,'-'],
    ['-','-', 2 , 6 ,'-','-','-', 7 ,'-'],
    ['-', 4 ,'-','-','-','-','-', 8 ,'-'],
    ['-', 6 ,'-','-','-', 3 , 2 ,'-','-'],
    ['-', 2 , 8 , 4 ,'-','-', 5 ,'-','-'],
    ['-','-','-','-', 9 , 6 , 1 ,'-','-'],
    ['-','-','-','-','-','-','-','-','-']
]

if len(argv) < 3:
    print('On attend deux arguments: grille (A ou B) ' +\
          'et méthode (forward_checking ou backtracking)')
    exit(1)

if argv[1].lower() == 'a':
    sudoku = Sudoku(grilleA)
elif argv[1].lower() == 'b':
    sudoku = Sudoku(grilleB)
else:
    print('Le premier argument doit être A ou B')
    exit(1)

if argv[2] in ('backtracking', 'forward_checking'):
    methode = argv[2]
else:
    print('Le second argument doit être forward_checking ou backtracking')
    sys.exit(1)

print('Grille ' + argv[1])
print(sudoku)

sudoku.resoudre(methode)

print(sudoku)
```

Comme les méthodes que vous allez programmer dans cet exercice sont une extension du code des chapitres précédents, il est plus commode de les implémenter dans des classes-filles, qui héritent des classes que nous avons déjà développées. Nous vous fournissons ainsi la classe `VariableAvecLabel`, qui étend

la classe `Variable`, la classe `ContrainteAvecPropagation`, qui étend la classe `ContrainteBinaire`, et la classe `PSCHeuristique`, qui hérite de la classe `PSC`.

Exercice 8.2.1 Variable ordering
Dans le cas du Backtracking, comme vous avez pu le constater, les variables sont instanciées les unes après les autres dans l'ordre où elles apparaissent dans `self.variables`. L'heuristique du *Variable Ordering* consiste à trier ces variables de façon à instancier d'abord celles dont le domaine est le plus restreint. L'idée est de commencer par les variables les plus restrictives, car ce sont celles-ci qui ont le plus de chance d'aboutir à une instanciation inconsistante, et donc à un backtrack.

Le premier exercice consiste à implémenter cet algorithme dans la classe `PSCHeuristique`. Pour trier les variables, vous pouvez appeler la méthode `sort()` de la liste `self.variables` en passant comme paramètre `key` une fonction lambda qui retourne la taille du domaine. Vous pouvez vous inspirer de la documentation disponible ici[(1)].

Vous pouvez ensuite tester votre algorithme sur le fichier `exemple_forward_checking.py` en vérifiant s'il améliore la performance du Backtracking.

Exercice 8.2.2 Algorithme du forward checking
Des heuristiques peuvent aussi être employées afin d'améliorer la recherche par rapport au Backtracking. Vous allez ainsi programmer l'heuristique connue sous le nom de *Forward Checking*. Elle a pour but d'éviter à l'avance des instanciations inconsistantes en appliquant le critère de la consistance des arcs pendant la recherche. Pour cela, il faut ajouter à chaque variable un attribut `self.label`, qui sera initialement égal au domaine de celle-ci. Le forward checking met alors à jour ces labels en appliquant la règle suivante : à chaque instanciation d'une variable x_k, on retire toutes les valeurs inconsistantes avec x_k des labels des variables qui ne sont pas encore instanciées.

Dans la méthode `forward_checking` de la classe `PSCHeuristique`, programmez donc l'algorithme ci-dessous :

```
solutions <- []
variables <- [v1, v2, ..., vn]

ForwardChecking(k, une_seule_solution)
1.  IF k >= n THEN
2.      IF NOT une_seule_solution THEN
3.          ajouter la solution actuelle à solutions
4.      ELSE
5.          RETURN solutions = [solution actuelle]
6.      END IF
7.  ELSE
8.      v <- variables[k]
9.      sauvegarde_labels <- labels des variables vk, ..., vn
10.     FOR EACH valeur de label d de la variable v DO
11.         assigner la valeur d à la variable v
12.         réduire le label de v à la seule valeur d
```

(1) http ://wiki.python.org/moin/HowTo/Sorting

```
13.          propager v=d aux labels des variables suivantes
14.          IF v=d est consistant THEN
15.              reste <− ForwardChecking(k+1, une_seule_solution)
16.              IF reste != échec THEN
17.                  RETURN reste
18.              END IF
19.          END IF
20.          labels des variables <− sauvegarde_labels
21.      END FOR
22. END IF
23. RETURN échec
END ForwardChecking
```

La méthode `forward_checking` prend deux paramètres :

- `k` : la profondeur courante (commence à 0),
- `une_seule_solution` : si `True`, retourne la première solution trouvée, sinon retourne toutes les solutions.

Les solutions seront stockées dans la variable de classe `self.solutions`. Chaque solution sera représentée par un dictionnaire qui, à un nom de variable donné, associera la valeur de cette variable.

Les étapes 13 et 14 de l'algorithme ci-dessus sont à implémenter à l'aide de la fonction `propagation_consistante` de la classe `PSCHeuristique`, que vous devez compléter. Pour chaque contrainte portant sur la variable courante et sur au moins une deuxième variable non encore instanciée, cette fonction doit appeler la méthode `propage` de la contrainte pour tenter de réduire le label de la deuxième variable.

Vous devez aussi implémenter la méthode `propage` de la classe `ContrainteAvecPropagation`. Pour chaque valeur possible de la deuxième variable, `propage` vérifiera si cette valeur est consistante avec la contrainte et la retirera du label de la variable si ce n'est pas le cas. Les deux méthodes `propagation_consistante` et `propage` devront retourner `True` si les contraintes peuvent être satisfaites, et `False` si au moins une des variables non encore instanciées n'a plus aucune valeur possible dans son label après propagation.

Lorsque vous avez terminé, vous pouvez tester votre implémentation du forward checking sur le fichier `exemple_forward_checking.py` en vérifiant s'il améliore la performance du Backtracking.

Exercice 8.2.3 Dynamic variable ordering

L'heuristique du variable ordering ne trie la liste des variables qu'une seule fois, avant la recherche. Le *Dynamic Variable Ordering* est une heuristique encore plus efficace, qui trie la liste des variables par ordre croissant de la taille du label à chaque étape de la recherche. L'idée est donc de retrier la liste des variables à chaque étape k, mais seulement à partir de la position k (car les variables précédentes ont déjà été instanciées).

Implémentez cette heuristique dans la classe `PSCHeuristique`, et appelez-la dans l'algorithme du forward checking entre les lignes 7 et 8 du pseudocode ci-dessus. Notez que vous n'avez pas besoin de trier toute la liste. En supposant que vous êtes à l'étape k, il est plus efficace de chercher la variable possédant

le plus petit label à partir de la position k et de l'échanger avec la variable d'indice k. Pour échanger les valeurs de deux variables `a` et `b` en Python, vous pouvez utiliser la syntaxe `a, b = b, a`.

Testez finalement à nouveau votre algorithme sur le fichier `exemple_forward_checking.py`, et comparez les résultats.

Exercice 8.2.4 Sudoku

Afin de comprendre ces algorithmes plus en détail, testez-les sur le fichier `exemple_sudoku.py`, qui contient deux grilles de Sudoku. Essayez la grille `A`, puis la grille `B`. Que constatez-vous ?

```
python3 exemple_sudoku.py A forward_checking
python3 exemple_sudoku.py B forward_checking
```

Essayez maintenant d'utiliser le Backtracking. Que constatez-vous ?

```
python3 exemple_sudoku.py A backtracking
python3 exemple_sudoku.py B backtracking
```

Solutions à la page 373

Chapitre 9

Diagnostic

Effectuer un diagnostic d'un système défaillant est une tâche fréquente et présente un grand domaine d'application de l'intelligence artificielle. Des outils automatiques sont utiles car :

- le problème est clairement défini et peut donc être résolu par un programme,
- le diagnostic nécessite souvent un raisonnement complexe qui dépasse les capacités des personnes confrontées au problème,
- une panne, par exemple dans un réseau électrique ou dans une installation industrielle, est souvent très coûteuse et doit être réparée dans un délai aussi bref que possible.

Un algorithme de diagnostic automatique prend comme entrées un modèle du comportement du système (MS) et des observations de ce comportement (OBS). Il doit produire un ensemble de candidats de diagnostic (CAND), dans lequel chaque candidat indique une combinaison de composants défectueux. L'algorithme peut trier les candidats selon la probabilité qu'ils constituent le bon diagnostic. Une autre fonctionnalité peut être de proposer des observations ou mesures qui permettent d'affiner l'ensemble des candidats.

Un système est défaillant si son comportement n'est pas consistant avec son modèle :

$$MS \cup OBS \vdash \perp (\text{Contradiction})$$

Nous allons d'abord considérer des méthodes qui *expliquent* les observations, c'est-à-dire que :

$$MS \cup CAND \vdash OBS$$

Il est évident que ce problème est un problème *abductif* et non pas déductif.

La difficulté principale pour un algorithme de diagnostic est que pour un certain ensemble d'observations, il y a souvent une grande variété de causes possibles. Par exemple, il y a beaucoup de raisons pour lesquelles une lampe peut ne pas s'allumer quand on enclenche l'interrupteur : l'ampoule peut être cassée ou mal vissée, la lampe peut avoir un mauvais contact ou la prise peut manquer de courant. Uniquement la prise en compte d'autres observations permet de rétrécir cet ensemble de possibilités.

Ce processus est difficile à modéliser par un raisonnement déductif comme l'exprime un système de règles. Un tel raisonnement impliquerait que chaque

observation peut rajouter de nouveaux candidats au lieu d'en éliminer. En fait, l'implémentation d'un processus de raisonnement abductif pose des difficultés liées au fait qu'il n'y a pas de correspondance directe avec un raisonnement déductif. Dans ce chapitre, nous allons voir trois manières de résoudre ce problème :

- par abduction explicite,
- par transformation en un problème déductif,
- par raisonnement incertain.

Ces techniques s'appliquent aussi à l'implémentation d'autres moteurs d'inférence abductifs, et le diagnostic peut être considéré comme un exemple d'un tel processus.

Par la suite, nous allons considérer un diagnostic basé sur la notion de consistance au lieu d'abduction. Cette technique est spécifique au diagnostic.

9.1 Trois manières d'implémenter un diagnostic

Considérons le circuit électrique de la figure 9.1. Le circuit contient trois ampoules a_1, a_2 et a_3 dont on peut observer le fonctionnement. Ces ampoules sont connectées à une source de courant. Nous supposons que cette dernière ne tombe jamais en panne. La connexion est faite par quatre câbles c_1, c_2, c_3 et c_4 qui peuvent être défectueux, ce qu'on ne peut pas observer directement. Le problème est de trouver quels câbles sont défectueux.

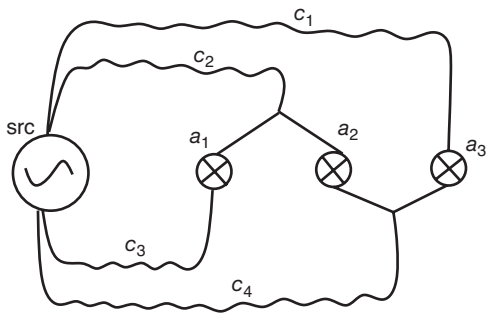


Fig. 9.1 *Circuit électrique dont on souhaite un diagnostic.*

La modélisation des dispositifs joue un rôle important, car elle fixe la granularité du diagnostic. Le modèle doit se situer au bon niveau pour identifier avec suffisamment de précision les composants défectueux. Normalement, on modélisera comme composants des unités qui peuvent être remplacées. En même temps, le modèle doit être aussi abstrait que possible pour limiter la complexité du raisonnement. Dans l'exemple, on suppose qu'on peut échanger les ampoules ou bien réparer les câbles individuellement.

Le problème peut être formalisé en logique comme suit. Nous décrivons le circuit et sa topologie par les propositions suivantes :

$\texttt{ampoule}(a_1), \texttt{ampoule}(a_2), \texttt{ampoule}(a_3)$
$\texttt{câble}(c_1), \texttt{câble}(c_2), \texttt{câble}(c_3), \texttt{câble}(c_4)$
$\texttt{connexion}(c_1, \texttt{src}, a_3), \texttt{connexion}(c_2, \texttt{src}, a_1),$
$\texttt{connexion}(c_2, \texttt{src}, a_2), \texttt{connexion}(c_3, a_1, \texttt{src}),$
$\texttt{connexion}(c_4, a_2, \texttt{src}), \texttt{connexion}(c_4, a_3, \texttt{src})$

Pour les prédicats qui figurent dans ces propositions, on peut formuler les règles suivantes, valables pour n'importe quel circuit :

$$\texttt{ampoule(x)} \wedge \texttt{cable(y)} \wedge \texttt{connexion(y,src,x)} \wedge \texttt{défectueux(y)} \Rightarrow \texttt{éteint(x)}$$
$$\texttt{ampoule(x)} \wedge \texttt{cable(y)} \wedge \texttt{connexion(y,x,src)} \wedge \texttt{défectueux(y)} \Rightarrow \texttt{éteint(x)}$$

Grâce à ces règles, par déduction on peut tirer les inférences suivantes :

$\texttt{défectueux}(c_1) \Rightarrow \texttt{éteint}(a_3)$
$\texttt{défectueux}(c_2) \Rightarrow \texttt{éteint}(a_1)$
$\texttt{défectueux}(c_2) \Rightarrow \texttt{éteint}(a_2)$
$\texttt{défectueux}(c_3) \Rightarrow \texttt{éteint}(a_1)$
$\texttt{défectueux}(c_4) \Rightarrow \texttt{éteint}(a_2)$
$\texttt{défectueux}(c_4) \Rightarrow \texttt{éteint}(a_3)$

qui décrivent les effets d'un câble défectueux.

À partir de l'état des composants, ces règles permettent donc de déduire les observations sur l'état des ampoules. Or, ce que nous voulons atteindre, c'est la possibilité de déduire l'état des composants à partir des observations sur l'état des ampoules. Nous allons décrire ci-après trois manières de réaliser un tel diagnostic.

9.1.1 Diagnostic par abduction explicite

Une première possibilité pour trouver un diagnostic de ce circuit est un raisonnement abductif explicite. On fera alors l'hypothèse du monde clos en supposant que seuls les quatre câbles peuvent être défectueux. Soient les observations :

$\texttt{éteint}(a_1), \texttt{allumé}(a_2), \texttt{éteint}(a_3)$

On fera alors une recherche entre toutes les combinaisons de câbles défectueux imaginables en retenant celles pour lesquelles le modèle du circuit (développé ci-dessus) prédit les observations qui ont été faites. La recherche est formulée comme suit :

- un nœud de recherche représente un ensemble de défauts,
- le nœud initial est l'ensemble vide,
- la fonction de successeur consiste à ajouter un défaut,
- le critère de terminaison consiste à ce que les défauts permettent de déduire les observations.

La figure 9.2 montre une trace de recherche pour l'exemple cité ci-dessus. Dans l'exemple, uniquement la combinaison :

`défectueux(c1) ∧ défectueux(c3)`

peut donner lieu aux observations. Il peut y avoir de nombreuses solutions qui expliquent les mêmes observations. Pour un diagnostic, on aimerait avoir l'explication la plus simple : elle est la plus probable, et montre la manière la plus simple de résoudre le problème. L'algorithme de recherche doit donc fournir la solution la plus simple, c.à.d. celle qui est la plus proche du nœud initial. L'algorithme A^* est souvent utilisé pour satisfaire ce critère.

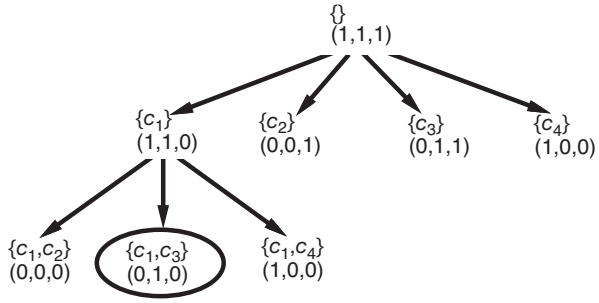


Fig. 9.2 *Exemple d'une recherche pour trouver un diagnostic.*

Notons que l'exactitude de la réponse dépend d'une hypothèse d'un monde clos : toutes les combinaisons de défauts possibles doivent être explorées lors de la recherche. Cette hypothèse doit cependant être faite uniquement au moment de la recherche. On peut donc changer les connaissances entre différentes instances de diagnostic.

9.1.2 Transformation en déduction

Une deuxième possibilité consiste à effectuer la recherche de diagnostic pour toutes les observations possibles. On peut donc construire un tableau de toutes les hypothèses justifiées selon les observations. Pour le circuit exemple, on obtient (1 = marche, 0 = éteint) :

a_1	a_2	a_3	Diagnostic
1	1	1	{}
1	1	0	c_1
1	0	0	c_4
1	0	1	-
0	0	1	c_2
0	1	1	c_3
0	1	0	$c_3 \wedge c_1$
0	0	0	$(c_1 \wedge c_2) \vee (c_3 \wedge c_4) \vee (c_2 \wedge c_4)$

Dans certains cas, il se peut que d'autres éléments soient également en panne : par exemple, les observations `1 0 0` peuvent aussi s'expliquer par une panne simultanée de c_1 et c_4. Les hypothèses données dans le tableau sont des hypothèses *minimales* dans le sens que toute hypothèse qui explique les observations doit au moins la contenir. Notons que cela implique également qu'elles sont des conclusions *nécessaires* des observations. Elles peuvent donc être transformées en règles, comme nous le verrons ci-dessous.

Il est important d'observer qu'une telle construction n'est valide que sous la condition d'une hypothèse de monde clos. Si, par exemple, on admettait également que les ampoules mêmes peuvent être défectueuses, l'espace des possibilités serait plus grand. Un diagnostic qui se base sur ce tableau sera donc forcement dépendant de la validité de cette hypothèse, et tout changement de connaissances qui viole l'hypothèse de monde clos oblige de recalculer le tableau.

Le tableau qui donne la liste de tous les cas de figures possibles sous l'hypothèse d'un monde clos peut être exploité pour construire un système déductif pour le diagnostic de ce circuit :

1) $\texttt{éteint}(a_3) \wedge \neg\ \texttt{éteint}(a_2) \Rightarrow \texttt{défectueux}(c_1)$

2) $\texttt{éteint}(a_1) \wedge \texttt{éteint}(a_2) \wedge \neg\ \texttt{éteint}(a_3) \Rightarrow \texttt{défectueux}(c_2)$

3) $\texttt{éteint}(a_1) \wedge \neg\ \texttt{éteint}(a_2) \Rightarrow \texttt{défectueux}(c_3)$

4) $\texttt{éteint}(a_2) \wedge \texttt{éteint}(a_3) \wedge \neg\ \texttt{éteint}(a_1) \Rightarrow \texttt{défectueux}(c_4)$

5) $\texttt{éteint}(a_1) \wedge \texttt{éteint}(a_2) \wedge \texttt{éteint}(a_3) \Rightarrow$
$(\texttt{défectueux}(c_1) \wedge \texttt{défectueux}(c_2)) \vee \ldots$

Ces règles correspondent alors à un système expert ou un programme conventionnel pour le diagnostic de ce circuit précis sous l'hypothèse d'un certain ensemble de défauts possibles. Un tel système pourrait par exemple être fourni par un constructeur de machines avec son produit. L'établissement des règles peut être obtenu par la programmation classique. Cependant, si le circuit est modifié ou si de nouveaux défauts sont possibles, le système doit être complètement réécrit. Le système fournira en plus des résultats faux s'il se trouve dans une situation où l'hypothèse du monde clos n'est pas vérifiée, par exemple quand il y a un court-circuit.

9.1.3 Abduction par raisonnement incertain

Une autre possibilité de résoudre un problème abductif est d'explicitement représenter l'incertitude qui résulte du fait que l'abduction n'est pas bien fondée. On peut l'exprimer en utilisant des *chiffres de certitude* (`CF`), qui sont plus ou moins équivalents à des probabilités. Chaque règle et chaque fait porteront un `CF`, et l'abduction peut alors être approximée par des règles déductives. Pour cet exemple, on pourrait formuler les règles incertaines comme suit :

1) $\texttt{ampoule(x)} \wedge \texttt{câble(y)} \wedge \texttt{connexion(y,src,x)} \wedge \texttt{éteint(x)} \overset{CF=0.5}{\Rightarrow} \texttt{défectueux(y)}$

2) $\texttt{ampoule(x)} \wedge \texttt{câble(y)} \wedge \texttt{connexion(y,x,src)} \wedge \texttt{éteint(x)} \overset{CF=0.5}{\Rightarrow} \texttt{défectueux(y)}$

3) $\texttt{ampoule(x)} \wedge \texttt{câble(y)} \wedge \texttt{connexion(y,src,x)} \wedge \neg\ \texttt{éteint(x)} \overset{CF=-1.0}{\Rightarrow} \texttt{défectueux(y)}$

4) $\texttt{ampoule(x)} \wedge \texttt{câble(y)} \wedge \texttt{connexion(y,x,src)} \wedge \neg\ \texttt{éteint(x)} \overset{CF=-1.0}{\Rightarrow} \texttt{défectueux(y)}$

en notant qu'un chiffre de certitude négatif représente la probabilité de la négation.

Si nous appliquons ces règles à l'observation où toutes les trois lampes sont éteintes, nous aurons :

$\texttt{éteint}(a_3) \Rightarrow$:

$\texttt{défectueux}(c_1)$,CF=0.5
$\texttt{défectueux}(c_4)$,CF=0.5

$\texttt{éteint}(a_1) \wedge \texttt{éteint}(a_2) \Rightarrow$:

$\texttt{défectueux}(c_2)$,CF=0.75 (2 règles)
$\texttt{défectueux}(c_3)$,CF=0.5
$\texttt{défectueux}(c_4)$,CF=0.5

Nous allons examiner les formules à utiliser pour un tel raisonnement ainsi que le problème général du raisonnement incertain en détail plus tard.

Notons ici qu'une des utilités d'un raisonnement incertain est qu'il permet de formuler des règles déductives indépendantes du contexte pour des problèmes de nature abductive. Il est donc très intéressant pour la mise au point de systèmes intelligents.

9.2 Diagnostic basé sur la consistance

Un diagnostic par abduction doit *prédire* le comportement défectueux qui est observé. Cela le rend bien sûr plus crédible, mais il peut être trop difficile à obtenir. Par exemple, un circuit logique défectueux a un comportement très complexe et il peut être très coûteux de caractériser exactement ce comportement. En plus, en pratique, il n'est pas important de connaître la défaillance exacte, mais plutôt d'identifier correctement le composant en cause.

Considérons par exemple un circuit arithmétique comme le montre la figure 9.3. Le circuit peut avoir des défaillances très complexes qui font que le résultat du calcul est faux uniquement pour certaines entrées. Pour un diagnostic, il n'est normalement pas important de connaître ces détails, car on ne pourra qu'échanger des modules entiers. Il est donc suffisant de savoir quels sont les modules qui ont des défaillances.

Formellement, cette idée peut s'exprimer comme suit. Au lieu d'expliquer les observations, un candidat au diagnostic doit spécifier les composants à enlever du modèle du système pour le rendre consistant avec les observations, c'est-à-dire :

$$(MS - CAND) \cup OBS \nvdash \bot$$

Appelons cela un diagnostic *basé sur la consistance*.

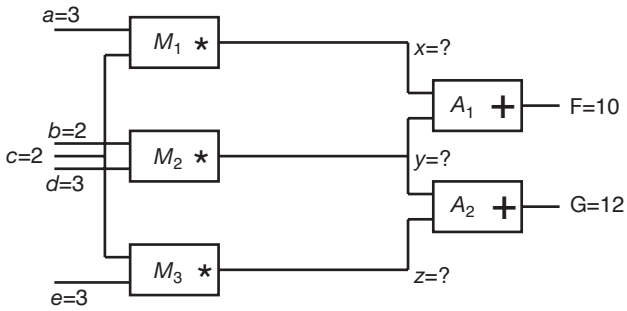


Fig. 9.3 *Modèle de circuit digital constitué de deux additionneurs et de trois multiplicateurs.*

Dans l'exemple de la figure 9.3, supposons que les trois unités de multiplication et les deux unités d'additions sont des composants qui peuvent être échangés de façon indépendante. Le diagnostic sera donc exprimé en termes de ces composants.

Chaque composant aura des lois de comportement. Notons qu'on modélise uniquement le comportement correct – aucun modèle d'un comportement défaillant est requis. Dans l'exemple de la figure 9.3, l'unité de multiplication M_1 aura un comportement décrit par la loi $x = A * C$. Utilisant les lois de comportement, on peut calculer des prédictions sur la base des observations. Par exemple, dans la figure 9.3, on peut calculer F = 12 à partir des observations $A = 3, B = 2, C = 2, D = 3$. Cette prédiction peut être *justifiée* par un comportement correct des trois composants M_1, M_2 et A_1.

Un candidat de diagnostic sera une combinaison de composants défectueux. Le diagnostic se base sur le principe suivant. Si un composant est défectueux, aucune des prédictions justifiées par son comportement correct n'est valable. Un candidat est donc valable si toutes les prédictions du modèle qui contredisent une observation dépendent d'un composant qui fait partie du diagnostic candidat.

Dans l'exemple de la figure 9.3, supposons que la mesure de F est 10 et non pas 12, comme prédit par le modèle. On appelle une telle paire un *symptôme*. La déduction F = 12 est valable à condition que M_1, M_2 et A_1 fonctionnent correctement. Étant donnée cette divergence, tous les composants de l'ensemble $\{M_1, M_2, A_1\}$ ne peuvent pas fonctionner correctement. On appelle cette combinaison un *conflit*. Les défaillances candidates se traduiraient par les candidats suivants :

$$\{M_1\}, \{M_2\}, \{A_1\},$$
$$\{M_1, M_2\}, \{M_1, A_1\}, \{M_2, A_1\}$$
$$\text{et } \{M_1, M_2, A_1\}$$

Cependant, en général on ne s'intéresse qu'aux candidats *minimaux* qui contiennent un minimum de composants. Dans cet exemple, les candidats minimaux sont :

$$D = \{\{M_1\}, \{M_2\}, \{A_1\}\}$$

Par propagation, la mesure F = 10 conduira aussi à d'autres conflits. Par exemple, sous l'hypothèse que M_1 fonctionne, le modèle prédit $x = 6$. En même temps, M_3 nous donne $z = 6$, avec A_2 et G=12 nous obtenons $y = 6$, et avec F = 10 et A_1 nous obtenons $x = 4$ avec la justification que $\{A_1, A_2, M_3\}$ fonctionnent tous. On aura donc le conflit C_2 entre $x = 4$ et la prédiction du modèle $x = 6$ qui implique les composants : $\{A_1, A_2, M_1, M_3\}$. Il est donc important qu'un moteur d'inférence, par exemple un moteur de chaînage avant, fasse toutes les inférences possibles pour obtenir tous les symptômes et, par conséquent, tous les conflits.

Un candidat de diagnostic doit rendre consistants tous les conflits. Cela veut dire qu'il doit contenir au moins un élément de chaque conflit. Lors d'un diagnostic, on découvre les conflits séquentiellement en prenant des mesures, et on aimerait donc maintenir l'ensemble des candidats de façon incrémentale aussi. Cela est possible par une procédure en deux étapes :

- générer un nouvel ensemble de candidats qui contient pour chaque candidat CAND et chaque composant c du conflit un candidat $C' = CAND \cup \{c\}$,
- filtrer l'ensemble des nouveaux candidats en éliminant tous ceux qui sont en double ou qui ne sont pas minimaux, c'est-à-dire éliminer tous les candidats X qui sont des sur-ensembles stricts d'un autre candidat Y ($X \supseteq Y$).

Par exemple, pour l'ensemble de candidats :

$$D_1 = \{\{M_1\}, \{M_2\}, \{A_1\}\}$$

et le nouveau conflit $\{A_1, A_2, M_1, M_3\}$, on génère d'abord un nouvel ensemble :

$$\begin{aligned} D' = \{ \quad & \{M_1, A_1\}, \{M_2, A_1\}, \{A_1\} \\ & \{M_1, A_2\}, \{M_2, A_2\}, \{A_1, A_2\} \\ & \{M_1\}, \{M_2, M_1\}, \{A_1, M_1\} \\ & \{M_1, M_3\}, \{M_2, M_3\}, \{A_1, M_3\} \quad \} \end{aligned}$$

Ensuite, on élimine les candidats qui ne sont pas minimaux, et on obtient ainsi le diagnostic :

$$D_2 = \{\{M_1\}, \{A_1\}, \{M_2, M_3\}, \{A_2, M_2\}\}$$

Il faut noter cependant qu'un composant défectueux n'a pas forcément toujours un comportement anormal. Par exemple, un multiplicateur défectueux qui omettrait le dernier bit du résultat aurait en apparence un comportement correct lorsque le résultat est pair, l'erreur n'étant visible que lorsqu'il est impair. Cela signifie en fait que si un diagnostic constitué de l'ensemble D de composants défectueux explique une divergence, tout sur-ensemble contenant D est aussi un diagnostic correct. Cependant, le proposer comme diagnostic serait trop pessimiste : on ne soupçonnera pas des composants d'avoir un défaut sans qu'ils aient été impliqués dans un symptôme de défaillance.

Notons enfin qu'une mesure donnée n'est utilisée par le système que si elle est en désaccord avec une prédiction issue du modèle. Cela peut parfois ne pas sembler très naturel si l'on considère qu'une observation en accord avec le modèle atteste du fonctionnement correct de certains composants. En basant le diagnostic sur la notion de *circonscription*, il est possible de tenir compte de ces considérations positives afin d'exonérer certains composants.

Un processus typique de diagnostic ne se poursuit pas jusqu'à ce qu'il n'y ait plus qu'un candidat unique, il s'achève plutôt dès qu'un candidat particulier se révèle beaucoup plus probable que les autres. Un critère simple de détection d'un tel candidat pourrait consister en un comptage : s'il persiste un candidat contenant un seul composant défectueux, il sera choisi comme le meilleur candidat. Mais une façon plus efficace de procéder consiste plutôt à calculer explicitement la probabilité d'occurrence de chaque candidat et d'arrêter la recherche dès que l'un d'entre eux semble plus probable que les autres.

Ce calcul est basé sur les probabilités de défaillance $P(c)$ de chaque composante c et l'hypothèse d'indépendance de défauts. Si on considère qu'un candidat X décompose les composantes en deux sous-ensembles :

$\{ D(X) =$ composantes défectueuses $\}$ et
$\{ N(X) =$ composantes non défectueuses $\}$

on peut alors calculer la probabilité d'un candidat X comme

$P(X) = \prod_{c \in D(X)} P(c) \cdot \prod_{c \in N(X)} (1 - P(c))$

Dès qu'il y a un candidat dont la probabilité est particulièrement élevée, le diagnostic peut être considéré comme terminé.

9.3 Proposition de mesures

Dans la plupart des cas, un système de diagnostic a la possibilité de proposer par lui-même les mesures à effectuer sur le dispositif à l'étape suivante. Le système doit en particulier proposer d'effectuer les mesures permettant d'aboutir à la meilleure discrimination dans l'ensemble courant des candidats. De telles *propositions de mesures* sont basées sur des *informations* relatives aux ensembles candidats, établies elles-mêmes par le biais de mesures. Une mesure de ces informations est donnée par la théorie de l'information :

$$I(X;Y) = E(X) - E(X|Y) = I(Y;X) = E(Y) - E(Y|X)$$

où $I(X;Y)$ est l'information que la mesure de Y donne sur la valeur de X. Cette information est d'ailleurs, par un théorème remarquable de la théorie de l'information, aussi égale à $I(Y;X)$. $E(X)$ est l'entropie de X :

$$E(X) = \sum_i -P(x_i) log(P(x_i))$$

et $E(X|Y)$ est l'entropie de X étant donnée la mesure de Y :

$$E(X|Y) = \sum_j P(y_j) \left\{ \sum_i -P(x_i|y_j) log(P(x_i|y_j)) \right\}$$

La mesure de Y est optimale si elle donne le maximum d'informations sur les candidats potentiels, c'est-à-dire si elle maximise la somme de $I(X;Y)$ pour l'ensemble courant des candidats possibles. Si on suppose que $E(Y)$ est la même pour tout Y, l'information est maximisée lorsque $E(Y|X)$ est minimisée, et par conséquent, proposer une mesure optimale revient à calculer $E(Y|X)$ pour toutes les mesures possibles de Y et à conserver celle qui est minimale. $E(Y|X)$ est calculée à partir des probabilités des résultats possibles de mesure pour Y étant donné l'ensemble courant des candidats.

L'entropie de la valeur d'une mesure Y se calcule par la formule suivante :

$$E(Y) = -\sum_{k} p(Y = val_k) * log(p(Y = val_k))$$

La probabilité de mesurer la valeur k pour la variable i se calcule de la manière suivante :

$$p(var_i = val_{ik}) = \sum_{c \in P_{ik}} p(c) + \sum_{c \in U_i} p(c)/m$$

ou P_{ik} est l'ensemble de candidats qui prédisent la valeur k pour la variable i, et U_i regroupe les candidats qui ne donnent aucune valeur pour la mesure i. Pour que le calcul soit correct, il faut cependant normaliser les probabilités pour que

$$\sum_{CAND} p(CAND) = 1$$

comme on sait qu'un des candidats doit être correct.

Dans notre exemple, les différents candidats prédisent des valeurs différentes pour les mesures X, Y et Z. Si on suppose que pour tous les composants, la probabilité de panne est de 0.01, on obtient les prédictions et probabilités suivantes :

Candidat	Prévision	Probabilité	Normalisé
$\{M_1\}$	$xyz = (4, 6, 6)$	0.01	0.495
$\{A_1\}$	$xyz = (6, 6, 6)$	0.01	0.495
$\{M_2, A_2\}$	$xyz = (6, 4, 6)$	10^{-4}	0.005
$\{M_2, M_3\}$	$xyz = (6, 4, 8)$	10^{-4}	0.005

et donc les probabilités des différentes valeurs :

Mesure	Justifications	Candidats	Probabilité
$X = 4$	$(\{M_2, A_1\}, \{M_3, A_1, A_2\})$	$\{M_1\}$	0.495
$X = 6$	$(\{M_1\})$	$\{A_1\}, \{M_2, A_2\}, \{M_2, M_3\}$	0.505
$Y = 6$	$(\{M_2\}, \{M_3, A_2\})$	$\{M_1\}, \{A_1\}$	0.99
$Y = 4$	$(\{M_1, A_1\})$	$\{M_2, A_2\}, \{M_2, M_3\}$	0.01
$Z = 6$	$(\{M_3\}, \{M_2, A_2\})$	$\{M_1\}, \{A_1\}, \{M_2, A_2\}$	0.995
$Z = 8$	$(\{M_1, A_1, A_2\})$	$\{M_2, M_3\}$	0.005

où il faut observer qu'un candidat prédit une mesure si au moins une des justifications de la mesure ne contient aucun élément qui fait partie du candidat.

On obtient donc les entropies :

$$\begin{aligned} X : -P(6)log(P(6)) - P(4)log(P(4)) &= \\ -0.505log(0.505) - 0.495log(0.495) &= 0.99993bit \\ Y : -P(6)log(P(6)) - P(4)log(P(4)) &= 0.0808bit \\ Z : -P(6)log(P(6)) - P(8)log(P(8)) &= 0.0454bit \end{aligned}$$

et on décide donc de mesurer X.

9.4 Modèles de défaillances

La technique de diagnostic basée sur la consistance souffre d'un problème fondamental dû à l'absence de modélisation des comportements anormaux, et qui engendre parfois des diagnostics candidats physiquement inconcevables. Un exemple d'une telle situation est donné par la figure 9.4, où la plupart des candidats inclut des défaillances physiquement irréalisables. De tels problèmes peuvent être résolus en utilisant des *modèles de défaillances* décrivant toutes les défaillances possibles dont est susceptible de souffrir un composant donné. Un comportement défectueux général est admis comme candidat dans le seul cas où le modèle de défaillances n'admet aucun des candidats possibles.

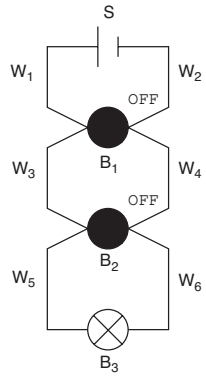


Fig. 9.4 *Exemple pour lequel le processus général de GDE produit un diagnostic inacceptable. Il existe 22 candidats minimaux, incluant la prédiction selon laquelle la batterie ainsi que l'ampoule B_3 sont défectueuses : la batterie ne produit pas de courant et B_3 est allumée sans courant. En éliminant ces défaillances physiquement impossibles, l'ensemble des candidats se restreint à seulement 4 candidats.*

Le diagnostic utilisant les modèles des défaillances procède en introduisant successivement les différentes défaillances dans l'ordre de leur probabilité, et en choisissant le premier qui explique le comportement observé. L'hypothèse d'une certaine défaillance est ainsi justifiée par le seul fait qu'il n'y a pas d'autre explication plus convaincante, ce qui constitue une forme de raisonnement par défaut.

Littérature

Les principes du diagnostic développés ici sont issus d'une synthèse de nombreux éléments. Le concept du diagnostic basé sur la consistance fut introduit dans [41], et la collection [42] réunit plusieurs travaux dans cette direction. Le papier plus récent [43] présente une vision à plus long terme de l'utilisation du diagnostic dans un contexte de systèmes autonomes.

Outils

Application : Diagnostic de systèmes spatiaux

En 1999, le vaisseau Mars Polar Lander faisait sa descente vers la surface de Mars. La descente était guidée par un altimètre jusqu'à une altitude de quarante mètres. Ensuite, le moteur freinait l'appareil et devait s'arrêter au moment où des capteurs placés dans les pieds détectaient le contact avec le sol. Or, la vibration a endommagé l'un de ces capteurs qui détectait le contact en permanence, ce qui a provoqué l'arrêt du moteur. La sonde est alors tombée de son altitude de quarante mètres. Le résultat, c'est qu'elle ne fonctionnait plus. Les ingénieurs n'avaient pas prévu tous les défauts possibles...
La même année, un autre vaisseau, la sonde Deep Space 1, subissait également un certain nombre de problèmes. Par exemple, une vanne d'une de ses fusées risquait de se bloquer, provoquant ainsi un dysfonctionnement du moteur. Heureusement, la sonde était équipée du système Livingstone, un outil de diagnostic basé sur modèles. Celui-ci aurait non seulement détecté le problème, mais également trouvé la manière de reconfigurer les vannes pour rendre le moteur à nouveau fonctionnel.

(Source : Wade Roush : Immobots take control, *MIT Technology Review* Dec. 2002/Jan. 2003, pp. 36-41.

Brian Williams *et al.* : Model-based Programming of Fault-Aware Systems, *AI Magazine* 24(4), 2003, pp. 61-75.)

9.5 Exercices

Exercice 9.1 Diagnostic d'un réseau par abduction explicite

Dans cet exercice, vous allez programmer un exemple d'abduction explicite. Pour ce faire, nous considérerons un réseau électrique formé d'un ensemble de blocs recevant des signaux les uns des autres. Chaque bloc prend deux signaux en entrée et produit une seule sortie. Le but, en supposant qu'une panne est survenue, est de découvrir quels sont les blocs qui ne fonctionnent pas.

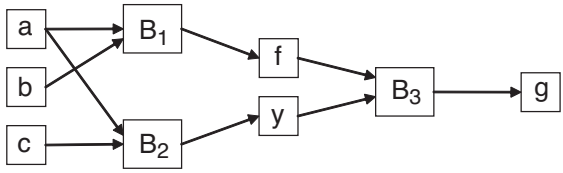


Fig. 9.5 Exemple d'un réseau.

Considérons un exemple concret illustré à la figure 9.5. Nous décrivons ce réseau de la manière suivante :

```
circuit = Circuit()

circuit.ajouter_bloc('B1', 'a', 'b', 'f')
circuit.ajouter_bloc('B2', 'a', 'c', 'y')
circuit.ajouter_bloc('B3', 'f', 'y', 'g')
```

Or sur ces blocs, on peut faire des observations. Par exemple, pour signaler que la sortie `g` n'est pas correcte, on invoque `circuit.observe_incorrect('g')` ; pour annoncer que la sortie `f` fonctionne normalement, on utiliserait `circuit.observe_correct('f')`. Nous verrons plus loin ces méthodes en détail.

Modules squelettes

Avant toute chose, voici le squelette du programme que nous allons développer. Nous discuterons des cinq premiers modules dans les paragraphes qui suivent. Les deux derniers, `exemple_1.py` et `exemple_2.py`, sont des modules de tests qui vous permettront de vérifier votre implémentation. Nous vous recommandons de respecter la structure des dossiers telle qu'elle est exprimée dans les noms des modules, sous peine de devoir modifier les `import`.

Module `.../reseau/bloc.py` :

```
class Bloc:
    def __init__(self, nom, entree_1, entree_2, sortie):
        self.nom = nom
        self.entree_1 = entree_1
        self.entree_2 = entree_2
        self.sortie = sortie

    def __lt__(self, autre):
        if not isinstance(autre, Bloc):
            raise ValueError("Seul un bloc peut être comparé àun bloc.")
        return self.nom < autre.nom

    def __eq__(self, autre):
        if not isinstance(autre, Bloc):
            raise ValueError("Seul un bloc peut être comparé àun bloc.")
        return autre.nom == self.nom

    def __hash__(self):
        return hash(self.nom)
```

```
    def __repr__(self):
        return 'Bloc({})'.format(self.nom)
```

Module `.../reseau/circuit.py` :

```
from .bloc import Bloc
from .disjonction import Disjonction
from .conjonction import Conjonction

class Circuit:
    def __init__(self):
        self.blocs = {}
        self.conflits = []
        self.no_goods = []

    def ajouter_bloc(self, nom, entree_1, entree_2, sortie):
        bloc = Bloc(nom, entree_1, entree_2, sortie)
        self.blocs[sortie] = bloc

    def observe_incorrect(self, signal):
        chemin = self.trouver_un_chemin(signal)
        conflit = self.traduire_chemin_en_disjonction(chemin)
        self.conflits.append(conflit)

    def observe_correct(self, signal):
        chemin = self.trouver_un_chemin(signal)
        for el in chemin:
            self.no_goods.append(Conjonction([el]))

    def trouver_un_chemin(self, signal):
        chemin = []
        sorties = [signal]
        while len(sorties) > 0:
            sortie = sorties.pop(-1)
            bloc = self.blocs.get(sortie)
            if bloc is not None:
                chemin.append(bloc)
                sorties.append(bloc.entree_1)
                sorties.append(bloc.entree_2)
        return chemin

    def traduire_chemin_en_disjonction(self, chemin):
        return Disjonction([Conjonction([etape]) for etape in chemin])
```

Module `.../reseau/conjonction.py` :

```
class Conjonction(frozenset):
    def __or__(self, autre):
        elements = list(self)
        elements.extend(autre)
        return Conjonction(elements)

    def __repr__(self):
        elements = [str(element) for element in sorted(self)]
        conj = ' & '.join(elements)
        if len(elements) > 1:
            return '({})'.format(conj)
```

```
        else:
            return conj
```

Module `.../reseau/disjonction.py` :

```
class Disjonction(set):
    def combiner(self, disjonction):
        if len(self) == 0:
            return disjonction
        elif len(disjonction) == 0:
            return self

        ret = Disjonction()
        for conj_1 in self:
            for conj_2 in disjonction:
                ret.add(conj_1 | conj_2)

        return ret

    def __repr__(self):
        if len(self) == 0:
            return '()'
        elements = [str(element) for element in sorted(self)]
        return ' | '.join(elements)
```

Module `.../reseau/abduction.py` :

```
from .disjonction import Disjonction
from .conjonction import Conjonction

class Abduction:

    def __init__(self, conflits, no_goods):
        self.conflits = conflits
        self.no_goods = no_goods

    def combiner_conflits_observations(self, disjonctions):
        print('à compléter')

    def retire_subsumes(self, conjonctions):
        print('à compléter')

    def retire_no_goods(self, conjonctions, no_goods):
        print('à compléter')

    def calcule_conflit_minimal(self, afficher_etapes=False):
        # 1. Combine les conflits.
        conflit_minimal = self.combiner_conflits_observations(self.conflits)
        if afficher_etapes: print('Conflit combiné :', conflit_minimal)

        # 3. Supprime les candidats subsumés.
        conflit_minimal = self.retire_subsumes(conflit_minimal)
        if afficher_etapes: print('Non subsumés :', conflit_minimal)

        # 4. Supprime les candidats contenant les no-goods.
        conflit_minimal = self.retire_no_goods(conflit_minimal, self.no_goods)
        if afficher_etapes: print('Sans no-goods :', conflit_minimal)
```

```
        return conflit_minimal
```

Module `.../exemple_1.py` :

```
from reseau.circuit import Circuit
from reseau.bloc import Bloc
from reseau.abduction import Abduction

circuit = Circuit()

circuit.ajouter_bloc('B1', 'a', 'b', 'f')
circuit.ajouter_bloc('B2', 'a', 'c', 'y')
circuit.ajouter_bloc('B3', 'f', 'y', 'g')

circuit.observe_incorrect('f')
circuit.observe_incorrect('g')

abduc = Abduction(circuit.conflits, circuit.no_goods)

conflit_minimal = abduc.calcule_conflit_minimal( afficher_etapes =True)

print('Conflit minimal :')
print(conflit_minimal)
```

Module `.../exemple_2.py` :

```
from reseau.circuit import Circuit
from reseau.bloc import Bloc
from reseau.abduction import Abduction


circuit = Circuit()

circuit.ajouter_bloc('B1', 'a', 'b', 'f')
circuit.ajouter_bloc('B2', 'a', 'c', 'g')
circuit.ajouter_bloc('B3', 'c', 'd', 'h')
circuit.ajouter_bloc('B4', 'c', 'e', 'i')

circuit.ajouter_bloc('B5', 'f', 'h', 'j')
circuit.ajouter_bloc('B6', 'g', 'i', 'k')

circuit.ajouter_bloc('B7', 'j', 'k', 'l')

circuit.observe_incorrect('l')
circuit.observe_correct('j')


abduc = Abduction(circuit.conflits, circuit.no_goods)

conflit_minimal = abduc.calcule_conflit_minimal( afficher_etapes =True)

print('Conflit minimal :')
print(conflit_minimal)
```

Abduction

Chaque observation génère un conflit. Un conflit est un ensemble d'éléments qui ne peuvent pas fonctionner tous en même temps. À partir de ces conflits, l'abduction explicite va générer des listes de candidats. Un ensemble de candidats est une liste d'éléments qui peuvent expliquer les observations. Donc, si on a un conflit entre a, b et c, on dira qu'un candidat parmi a, b, ou c doit être fautif.

Nous représenterons les candidats sous la forme de conjonctions exprimant un ET logique. La conjonction de A, B, et C s'interpète donc comme $A \wedge B \wedge C$. Nous définissons ainsi une classe `Conjonction`, qui hérite de la classe `frozenset` de Python. Il s'agit d'une collection immutable, non ordonnée et ne pouvant contenir qu'une instance d'un object donné. Elle possède une *méthode spéciale* importante pour nous : `__or__`, qui permet d'utiliser la syntaxe `conj = conj_1 | conj_2` pour créer une conjonction portant sur tous les éléments présents dans `conj_1` ou dans `conj_2` (ou éventuellement dans les deux).

Un ensemble de candidats sera une disjonction de conjonctions (un OU logique), que nous représenterons par la classe `Disjonction`. Cette dernière spécialise la classe `set` de Python. Si `conj_1 = Conjonction(['A', 'B'])` et `conj_2 = Conjonction(['D', 'E'])` sont deux conjonctions, c'est-à-dire deux candidats, correspondant à $(A \wedge B)$ et $(D \wedge E)$, leur disjonction `Disjonction([conj_1, conj_2])` exprimera le fait $(A \wedge B) \vee (D \wedge E)$. `Disjonction` contient essentiellement la méthode `combine`, qui combine l'objet courant avec une autre disjonction afin de produire une nouvelle disjonction contenant toutes les conjonctions qui résultent de l'union deux-à-deux des conjonctions des disjonctions originales.

Nous aurons également besoin d'une liste de *nogoods*, c'est-à-dire de valeurs contradictoires. Un *nogood* est une liste d'éléments qui ne peuvent être présents en même temps. Par exemple, si `Conjonction(['A', 'E'])` est un *nogood*, cela signifie que A et E ne peuvent être la source du problème en même temps.

La classe `Abduction` formalise le processus abstrait de l'abduction. Elle contient les deux attributs `self.conflits` et `self.no_goods` pour les conflits et les *nogoods*. Sa méthode principale `calcule_conflit_minimal` est une fonction générale qui permet de trouver les candidats minimaux. Nous vous en donnons le code. `calcule_conflit_minimal` procède en quatre étapes :

1) elle génère un ensemble de combinaisons de candidats à partir des candidats de chaque conflit,
2) elle supprime les candidats subsumés,
3) et enfin elle supprime les candidats contenant les *nogoods*.

Chacune de ces étapes est réalisée par une méthode spécialisée. Ce sont ces méthodes que vous devez implémenter en suivant les descriptions que nous vous donnons :

- `combiner_conflits_observations` : combine itérativement tous les candidats pour chaque conflit. Cette fonction doit s'appuyer sur `Disjonction.combine`.

- `retire_subsumes` : enlève les candidats qui sont subsumés. Par exemple, $(a \wedge b \wedge c) \vee (a) \vee (b \wedge c) \vee (b \wedge c \wedge d)$ doit donner $(a) \vee (b \wedge c)$.
- `retire_no_goods` : enlève les candidats qui contiennent un *nogood*.

```
class Abduction:
    ...

    def combiner_conflits_observations( self ,  disjonctions ):
        ...

    def retire_subsumes( self ,  conjonctions ):
        ...

    def retire_no_goods( self ,  conjonctions ,  no_goods):
        ...
```

Blocs et circuits

Une fois le moteur d'abduction créé, nous pouvons l'appliquer à notre exemple du réseau électrique. Tout d'abord, nous devons définir une classe `Bloc`, qui modélise un bloc. Elle possède un attribut `nom`, qui identifie chaque bloc, et trois attributs correspondant aux identifiants des deux entrées et de la sortie. Les blocs sont assemblés en un réseau modélisé par la classe `Circuit`. Cette dernière contient trois attributs `self.blocs`, `self.conflits`, `self.no_goods`, qui recueilleront respectivement les blocs, les conflits et les *nogoods*. Les deux derniers attributs sont des listes standards, le premier est un dictionnaire dans lequel chaque bloc est identifié par son signal de sortie.

La méthode `ajouter_bloc` de `Circuit` permet de construire le circuit en ajoutant des blocs un à un, étant donné leurs noms et les identifiants de leurs entrées et sorties. `observe_incorrect` notifie l'observation d'un signal erroné, `observe_correct` en revanche garanti qu'un signal fonctionne convenablement. `trouver_un_chemin` et `traduire_chemin_en_disjonction` permettent de définir un conflit une fois qu'un signal défectueux à été enregistré. La première méthode collecte toutes les causes possibles de l'observation, et la seconde les traduit en une disjonction de conjonctions, c'est-à-dire un conflit.

Étant donné un circuit, il suffit ensuite de passer le contenu de ses attributs `conflits` et `no_goods` à un moteur d'abduction, et d'appeler `calcule_conflit_minimal` pour trouver le conflit minimal susceptible d'expliquer les observations.

Test du programme

Vous pouvez tester maintenant votre solution sur les deux modules de test `exemple_1.py` et `exemple_2.py`. Essayez en outre de créer des réseaux plus compliqués, en introduisant davantage de blocs.

Solutions à la page 376

Chapitre 10

Génération de plans

Les humains sont capables de planifier des *actions* en vue d'atteindre un objectif. Pour de nombreuses tâches, comme la programmation d'un robot autonome, l'ordinateur est amené à s'inspirer de cette activité intelligente propre à l'homme. Pour qu'un robot puisse fonctionner correctement, il doit évidemment être en mesure de formuler des plans et de les exécuter pour atteindre les objectifs souhaités. Nous nous intéressons dans ce qui suit à ce problème particulier pour lequel nombre de travaux ont été réalisés. Nous illustrerons les méthodes au moyen de l'exemple de STRIPS qui est l'un des premiers travaux dans ce domaine. Il présente l'intérêt d'être à la base de plusieurs mécanismes présents dans les systèmes de planification actuels.

Le problème de planification se pose de façon très nette pour la gestion des procédures impliquant des humains et des machines qui ont un degré de complexité tel qu'il est difficile d'en avoir une vision d'ensemble. Nombre de systèmes pratiques sont en effet impossibles à gérer proprement, car on ne peut développer des plans d'action optimaux pour les opérations à réaliser. Prenons pour exemple le cas d'une mission spatiale : il existe des centaines de buts à réaliser pendant le laps de temps relativement court de la mission. De plus, dans le cas où les problèmes rencontrés n'ont jamais été étudiés auparavant, il faudra pouvoir très vite modifier les plans pour s'accommoder des particularités inattendues. Un autre exemple peut être pris des processus industriels de grande envergure, comme la construction d'avions. Ceux-ci impliquent souvent des millions d'opérations différentes qui nécessitent une planification automatique par ordinateur.

Il est très intéressant d'utiliser des systèmes basés sur la connaissance dans le domaine de la planification, parce que, d'une part, la tâche en elle-même est compliquée et, d'autre part, elle ne peut être traitée algorithmiquement. En pratique, les systèmes de planification sont développés sur la base de règles heuristiques établies par des experts.

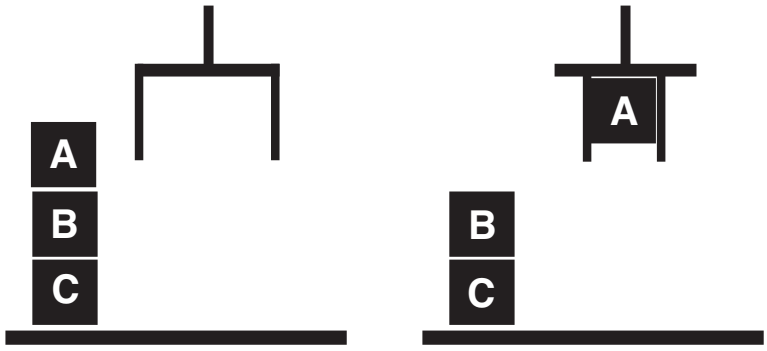
10.1 Représentation d'un environnement changeant

Un système de planification doit représenter un monde dont il est lui-même l'agent des changements effectués : le monde ne reste pas inchangé comme c'est le cas pour un système de diagnostic. Lorsqu'une action donnée d'un plan est exécutée, on dit que le monde change de *situation*. Pour enchaîner correctement

les actions d'un plan, le modèle du monde doit donc pouvoir distinguer les différentes situations auxquelles il peut être confronté durant l'exécution du plan.

Comme exemple, considérons le *monde des blocs*, illustré par la figure 10.1. Chaque *état* du monde est représenté par une *situation*, qui est une description partielle de l'état en calcul des prédicats. Ces représentations utilisent les prédicats suivants :

- `ON(x,y)` : le bloc `x` se trouve directement au-dessus du bloc `y`.
- `ONTABLE(x)` : le bloc `x` se trouve sur la table.
- `CLEAR(x)` : il n'y a rien sur le bloc `x`.
- `HOLDING(x)` : la main tient le bloc `x`.
- `HANDEMPTY` : la main est vide.




```
ON(A,B)
ON(B,C)
CLEAR(A)
HANDEMPTY
ONTABLE(C)
```

```
HOLDING(A)
ON(B,C)
CLEAR(B)
ONTABLE(C)
```

Fig. 10.1 *Deux états du monde des blocs et leur représentation sous la forme de situations en calcul de prédicats.*

Les deux états illustrés dans la figure 10.1 sont logiquement incompatibles entre eux : `HANDEMPTY` est en contradiction avec `HOLDING(A)`. Par conséquent, la représentation doit distinguer les deux situations. Le *calcul de situations*, développé pour le système de planification de robots STRIPS, est un exemple de modèle d'environnement capable d'intégrer la notion de changement de situation. Dans ce modèle chaque nouvelle situation créée par l'exécution d'un plan est indexée par un symbole particulier. Tous les prédicats utilisés pour modéliser un état du monde sont modifiés de sorte à incorporer comme nouvel argument la situation dans laquelle ils s'appliquent. Lorsque l'on passe d'une situation S_1 à une situation S_2, la plupart des prédicats valides dans S_1 le restent dans S_2, mais ce fait doit cependant être explicitement déduit à partir d'un ensemble d'*axiomes cadres*. Le problème relatif à la formulation de ces

axiomes est généralement désigné sous le nom de *problème de cadres*. Le terme de *cadre* établit en fait une analogie avec les changements de situations et de cadres (décors) dans un film : de nombreuses situations ont lieu dans un cadre quasi identique. Le passage d'un cadre à un autre se fait par l'intermédiaire d'une *action*, modélisée comme l'application d'un *opérateur*.

La façon la plus naturelle de formuler des axiomes cadres consiste à utiliser la règle par défaut suivante : tout ce qui n'est pas explicitement mentionné dans la description d'un opérateur demeure inchangé après son exécution. Cette règle a été adoptée dans le calcul de situations, qui spécifie explicitement pour chaque opérateur la liste `DELETE` des propositions qui ne seront plus valides après son exécution et la liste `ADD` des propositions qui au contraire deviendront valides.

En plus, pour appliquer une action, il faut respecter certaines *préconditions*. Par exemple, pour qu'un robot puisse saisir un objet, il faut satisfaire au préalable que l'objet soit dégagé et que la main du robot soit libre. La transformation qui résulte de l'action établit une nouvelle situation dans laquelle le robot porte l'objet. Dans le formalisme de STRIPS, cela s'exprime au moyen d'un opérateur **PICKUP**(x), où x est une variable qui représente l'objet à saisir :

PICKUP(x)
PRECONDITIONS (P) = `HANDEMPTY, CLEAR(x),ONTABLE(x)`
DELETE (D) = `HANDEMPTY, CLEAR(x), ONTABLE(x)`
ADD (A) = `HOLDING(x)`

Les autres opérateurs utilisés dans le monde des blocs sont :

- `PUTDOWN(x)` :
  P= `HOLDING(x)`, D = P,
  A= `ONTABLE(x)`, `CLEAR(x)`, `HANDEMPTY`
- `PUTON(x,y)` :
  P = `HOLDING(x)`, `CLEAR(y)`, D = P,
  A = `HANDEMPTY`, `ON(x,y)`, `CLEAR(x)`
- `UNSTACK(x,y)` :
  P = `HANDEMPTY`, `ON(x,y)`, `CLEAR(x)`,
  D = P, A = `HOLDING(x)`, `CLEAR(y)`

La principale difficulté posée par cette formulation vient du fait que le nombre de prédicats qui doivent figurer dans les listes `ADD`, `DELETE` et `PRECONDITIONS` devient rapidement très important. Cela est particulièrement vrai lorsque des propositions additionnelles peuvent être ajoutées aux listes par le biais de règles d'inférences : la liste `DELETE` d'un opérateur doit alors aussi supprimer toutes les conséquences inférées sur la base de faits qui ne sont plus vrais.

La représentation que nous venons de voir est formalisée dans le langage `PDDL` (Planning Domain Definition Language) qui a permis de standardiser l'interface aux algorithmes de planification et ainsi de les comparer sur différents problèmes test.

10.2 Planification par chaînage arrière

L'algorithme de planification est lancé avec une situation initiale et une situation but, qui, comme toute expression de situation, ne donnent qu'une description *partielle* des états à transformer. La planification se fait par chaînage-arrière à partir du but, de façon analogue au système à chaînage-arrière utilisé par les systèmes experts. La raison pour laquelle on utilise le chaînage-arrière est qu'en général, le nombre d'actions possibles à partir d'un certain état est beaucoup plus grand que le nombre d'actions qui peuvent servir aux buts courants ; donc, la recherche en chaînage arrière est beaucoup plus efficace.

La figure 10.2 montre un exemple d'une réduction effectuée par un tel processus. L'utilisation de cette stratégie se justifie par le fait que les systèmes de planification comme STRIPS sont basés sur une analyse des objectifs et des moyens (moyens-buts) qui requiert de lier une opération à un but donné. Comme il n'existe généralement aucun opérateur permettant d'atteindre directement le but en un seul pas d'inférence, STRIPS réduit itérativement les buts en sous-buts jusqu'à satisfaire la situation initiale. Il est à noter que la classification des opérateurs par rapport aux buts qu'ils permettent de réaliser est implicitement obtenue par le contenu des listes `ADD`.

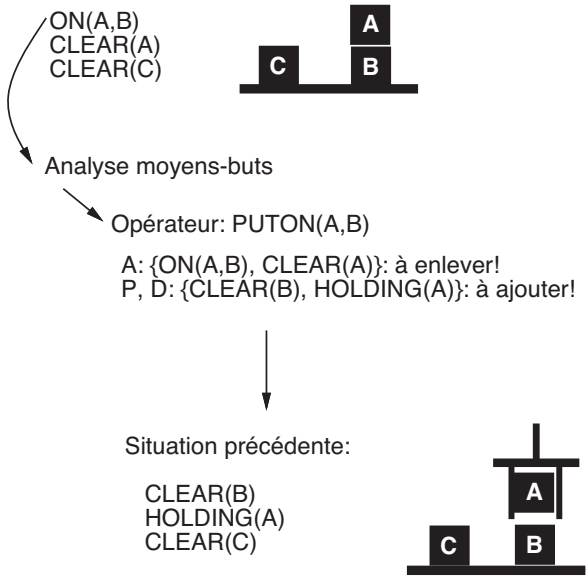


Fig. 10.2 *Chaînage arrière pour réduire la situation but à une situation sous-but.*

La réduction d'une situation but S en une situation sous-but S' s'effectue en planifiant l'application d'un opérateur O transformant S' en S. Dans le contexte de STRIPS, une description de S' s'obtient comme suit (fig. 10.2) :

- les buts (propositions de la description partielle de l'état) existant dans S sont unifiés avec les propositions de la liste `ADD` de l'opérateur O ne figurant pas dans S',

- les propositions de la précondition de O qui n'apparaissent pas dans S', ou qui font partie de la liste `DELETE`, sont ajoutées comme sous-buts à la description de S',
- toutes les autres propositions sont reportées dans S.

Comme la description des situations S et S' en terme de buts à atteindre n'est que partielle, il est possible que le processus de régression génère des descriptions ambiguës de situations, ce qui peut conduire à de nouveaux branchements. Si nous considérons, par exemple l'application de l'opérateur :

UNSTACK`(x,y)`
`P = D = HANDEMPTY, CLEAR(x), ON(x,y)`
`A = HOLDING(x), CLEAR(y)`

pour atteindre le but `HOLDING(A)` en présence d'un autre but `CLEAR(C)`, l'instanciation de l'opérateur fixera $x = A$, mais aucune valeur pour y. Si $y = C$, ce que le système de planification ne peut pas savoir, on n'aura plus besoin d'une action pour obtenir `CLEAR(C)` ; si $y \neq C$, il faut le maintenir comme but. La régression de `CLEAR(C)` impliquera donc la disjonction `(y=c)` $\vee$ `CLEAR(C)`. Les deux possibilités doivent être poursuivies séparément.

La planification construit la chaîne d'opérations en appliquant un algorithme de *recherche* tel que l'algorithme A*. Par exemple, supposons que le but est de construire une tour de trois blocs, décrite par la situation :

`ON(A,B), ON(B,C)`

à partir de la situation :

`ON(C,A), CLEAR(B), CLEAR(C)`

L'arbre de recherche qui résulte pour ce problème est décrit par la figure 10.3. A l'exception du premier niveau, la trace ne montre pas tous les chemins possibles, mais uniquement le chemin qui conduit à la solution. L'arbre de recherche complet est beaucoup plus grand et contient de nombreux chemins inutiles.

Pour minimiser le coût du plan résultant, on utilise l'algorithme A* où le coût se compose des coûts des opérateurs impliqués dans le plan, et la fonction heuristique peut se baser sur la différence entre la situation actuelle et la situation but.

Notons l'importance de détecter des situations qui sont en fait contradictoires aussitôt que possible afin d'éviter des recherches inutiles. La détection des inconsistances est aussi importante pour éviter des plans inconsistants qui peuvent être générés sinon. Par exemple, considérons la situation but précédente :

`ON(A,B), ON(B,C)`

En appliquant l'opérateur `PUTON(B,C)`, on obtient la situation sous-but contradictoire (et physiquement impossible) :

`ON(A,B), HOLDING(B), CLEAR(C)`

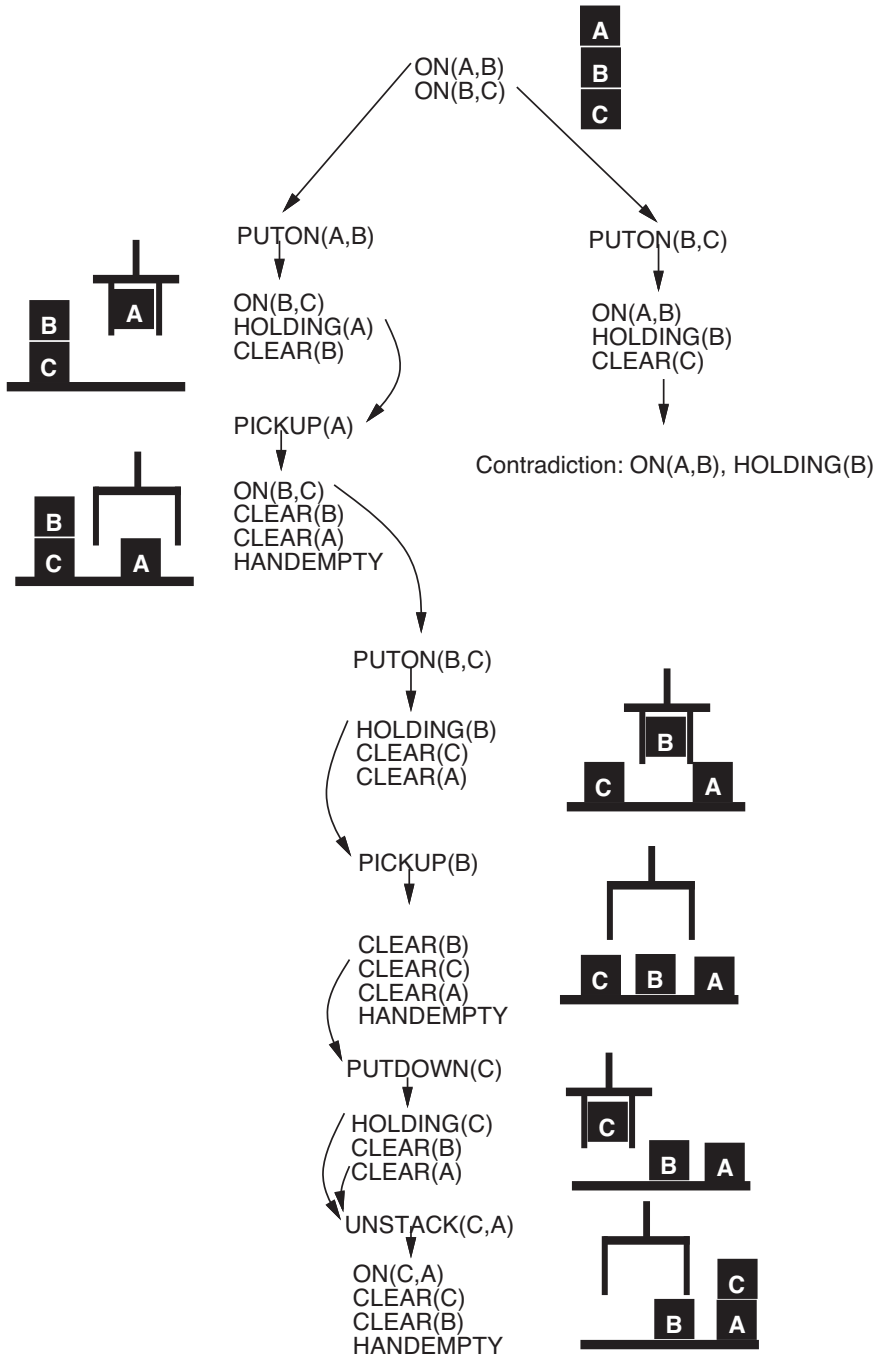


Fig. 10.3 *Arbre de recherche pour la construction d'une tour de 3 blocs.*

à laquelle on peut cependant appliquer la séquence d'opérateurs `PICKUP(B)`, `PUTON(A,B)` et `PICKUP(A)` pour obtenir la situation initiale parfaitement consistante :

`CLEAR(A), CLEAR(B), CLEAR(C)`

Si l'inconsistance n'est pas détectée, le système considérerait ce plan comme une solution, bien qu'il soit physiquement impossible à réaliser ! Il est donc important d'arrêter la recherche aussitôt qu'un état inconsistant est détecté.

Notons également quelques lacunes de l'analyse moyens-buts qui est à la base de l'algorithme de planification. Une première lacune concerne la formulation de la représentation. Supposons que celle-ci inclut un prédicat `ABOVE(x,y)` qui spécifie que `x` se trouve quelque part au-dessus de `y`. Une tour de 3 blocs `A, B, C` pourrait alors être spécifiée par les buts :

`ABOVE(A,C)` $\wedge$ `ON(A,B)`

Or, l'analyse moyens-buts est incapable de trouver les opérateurs qui permettent d'atteindre cette combinaison de buts : le but `ABOVE(A,C)` ne permet que de proposer un opérateur `PUTON(A,C)`, mais pas l'opérateur `PUTON(B,C)` qui serait nécessaire pour atteindre ce but en combinaison avec le but `ON(A,B)`. Un autre problème concerne le fait qu'il est impossible d'introduire dans un plan l'utilisation d'objets non mentionnés dans le but. Ce problème est plus profondément lié au problème théorique faisant qu'une planification permettant l'introduction d'éléments supplémentaires n'est pas calculable par un ordinateur.

10.3 Macro-opérateurs

Dans STRIPS, les séquences d'actions réalisant un type particulier de buts, peuvent être précalculées et stockées sous forme de macro-opérateurs (`MACROP`). Un macro-opérateur permet au système de planification de décrire les plans à un niveau plus élevé que celui donné par des opérateurs élémentaires. Les `MACROP` sont représentés par des *tables triangulaires*, ayant l'aspect décrit par l'exemple de la figure 10.4.

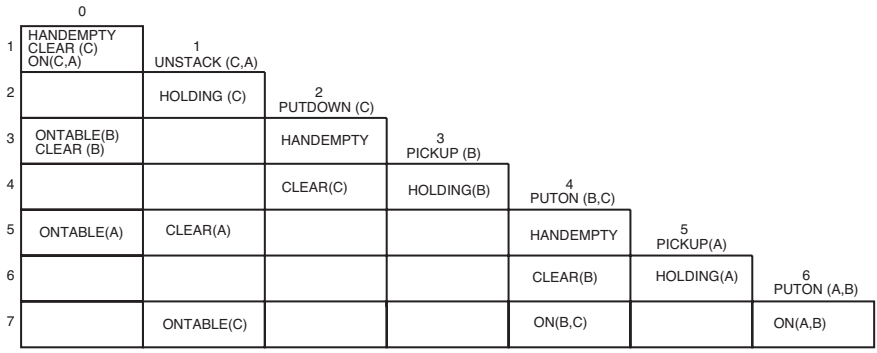


Fig. 10.4 *Exemple de tables triangulaires utilisées par STRIPS.*

La table triangulaire que montre cette figure représente un plan pour construire une tour de trois blocs A, B et C. On peut le réappliquer comme un macro-opérateur suivant :

```
P = D = HANDEMPTY,CLEAR(C),ON(C,A),ONTABLE(B),CLEAR(B),ONTABLE(A)
A = ON(B,C), ON(A,B), ONTABLE(C)
```

Sur la diagonale de la table triangulaire figurent les séquences d'actions du plan. Chaque colonne située en-dessous d'une action donne la description des situations qui s'ajoutent par l'exécution de cette action. Sur les lignes sont spécifiées les propositions supprimées par exécution des actions de la diagonale.

Considérées plus attentivement, ces tables triangulaires présentent une particularité intéressante : l'ensemble total des propositions ajoutées et supprimées par la séquence d'opérateurs, peut être entièrement lu sur la colonne 0 (liste `DELETE` de `MACROP`) et sur la dernière ligne (liste `ADD` de `MACROP`). Tout ceci établit le fait que les tables triangulaires constituent une structure adéquate pour la représentation d'opérateurs plus abstraits que ceux utilisés à la base.

10.4 La complexité du problème de la planification

Le problème de générer un plan utilisant le formalisme du calcul des situations a été analysé en informatique théorique. On considère en fait le problème `PLANMIN` : est-ce qu'il existe un plan pour un certain problème donné avec une longueur d'au plus k opérateurs ? Si le nombre d'opérateurs ou d'objets n'est pas fini, le problème ne peut pas être résolu par une machine de Turing et il est inutile de parler de sa complexité. Autrement, le problème de la planification peut se réduire directement à différentes versions du *halting problem* pour machines de Turing.

La figure 10.5 montre différentes classes du problème et leur complexité. Chaque classe est caractérisée par des conditions sur les opérateurs et parfois les buts :

- 0,1,2,* indique le nombre de pré(post)conditions,
- + veut dire que la condition doit être une proposition sans négation,
- g buts veut dire que le nombre de buts ne dépasse pas une constante g.

Le cas le plus général ainsi que trois sous-classes indiquées dans la figure 10.5 sont équivalents à une machine de Turing et donc PSPACE-complets. Les cas qui peuvent être résolus en temps polynomial se limitent à des opérateurs sans préconditions ou bien à un nombre limité de buts et au plus une précondition par opérateur – autrement, on pourrait construire des sous-buts et surmonter la limitation du nombre de buts. Ces cas ne sont en général pas intéressants dans la pratique.

On voit donc que la planification est un problème très difficile et pour sa résolution, on doit compter sur de bonnes méthodes heuristiques comme elles ont été développées dans le cadre de moteurs d'inférence ou de la satisfaction de contraintes. Nous allons donc voir comment profiter de ces outils pour la planification.

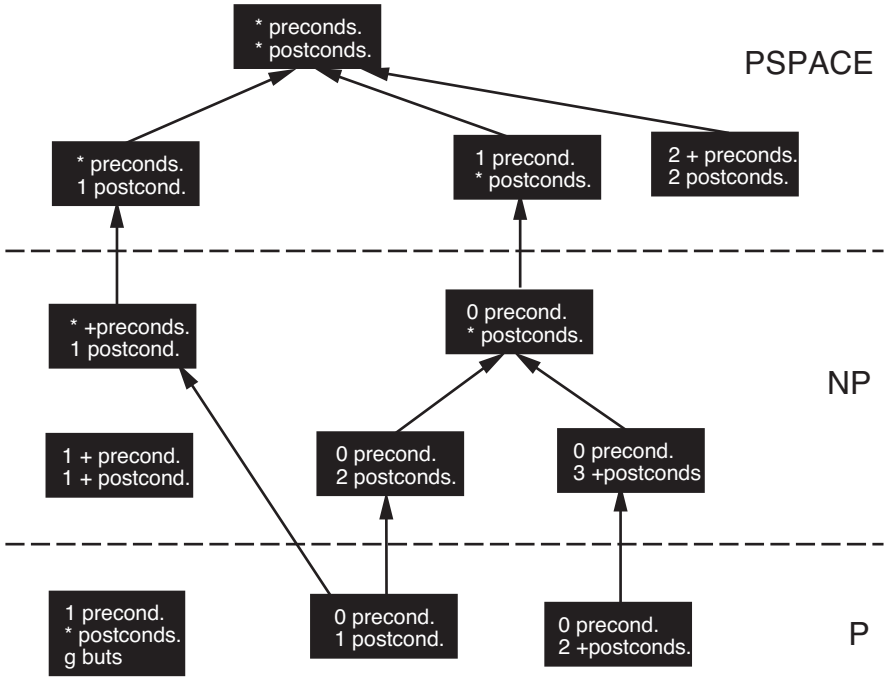


Fig. 10.5 *Hiérarchie de complexité du problème* PLANMIN.

10.5 Planification par inférence logique

Un problème de planification peut également être posé comme un problème d'inférence, ce qui permet d'utiliser un moteur d'inférence quelconque comme base du système. Une manière de faire cela consiste à ajouter à chaque prédicat utilisé pour décrire une situation un argument qui indique la situation dans laquelle le prédicat est valable :

ON(A,B,S_1)

et en définissant une fonction d'évolution des états :

S_1 = apply(PUTON(A,B), S_0)

Au lieu de formuler les opérateurs avec des listes ADD, DELETE, etc., on formule des règles d'inférence comme ceci :

(CLEAR(x,s) $\wedge$ HOLDING(y,s)) $\Rightarrow$
(ON(y,x,apply(PUTON(y,x),s)) $\wedge$ ....

Un plan sera alors construit de la manière suivante. Supposons que nous partons de la situation initiale S_0 :

ONTABLE(A,S_0), ON(B,C,S_0), HANDEMPTY(S_0)

et que nous cherchons à satisfaire le but `ON(A,B,x)`, où la variable `x` désigne une éventuelle situation finale que le système ne connaît pas encore. Si toutes les règles d'inférence sont bien formulées, un moteur d'inférence retournera par exemple le résultat :

```
x = apply(PUTON(A,B),
          apply(PICKUP(A),
                apply(PUTDOWN(B),
                      apply(UNSTACK(B,C),S0))))
```

ce qui est une substitution de la variable x qui rendra le résultat sous la forme d'une conséquence de la situation initiale.

Cependant, cette manière de résoudre le problème pose à nouveau le problème des *cadres* : pour chaque combinaison de prédicat P et d'opérateur O, on est obligé de formuler des règles d'inférences qui indiquent si P reste valable ou non dans la situation qui résulte de l'application de l'opérateur O. Si nous pensions, par exemple, à tous les blocs qui peuvent se trouver ailleurs sur une table pendant l'exécution d'un plan, il est évident que le nombre de règles qu'on aurait à formuler serait énorme. Il faut donc formuler des *axiomes cadres*, c'est-à-dire des règles générales du type :

$$(\forall\ \mathtt{P})\ (\mathtt{P} \neq \mathtt{CLEAR}) \wedge (\mathtt{P} \neq \ldots) \Rightarrow$$
$$(\mathtt{P(s)} \Rightarrow \mathtt{P(apply(PUTON(x,y),s))})$$

Malheureusement, il s'agit là d'une règle du calcul de prédicats de deuxième ordre, ce qui ne peut pas être utilisé dans une procédure d'inférence algorithmique. Une issue à cela est de générer un individu pour chaque prédicat utilisé dans la planification et de l'associer à un état spécifique par un prédicat `HOLDS` :

$$\mathtt{HOLDS(ON(A,B),S_0)}$$

ce qui fait que les axiomes de cadre deviennent des règles de 1$^{\text{er}}$ ordre :

$$(\forall\ \mathtt{P})\ (\mathtt{TYPE(P)} \neq \mathtt{CLEAR}) \wedge (\mathtt{TYPE(P)} \neq \ldots) \Rightarrow$$
$$(\mathtt{HOLDS(P,s)} \Rightarrow \mathtt{HOLDS(P,(apply(PUTON(x,y),s))))}$$

La formulation logique du problème de la planification permet son intégration avec d'autres considérations, notamment le raisonnement temporel qui permet de planifier des opérations qui se déroulent en parallèle.

10.6 Buts multiples

Quand un problème de planification implique plusieurs buts à réaliser en même temps, on peut souvent exécuter les actions qui y mènent dans n'importe quel ordre. La planification par A^* effectue une recherche linéaire entre les différentes séquences d'états, ce qui conduit à une explosion combinatoire de nœuds de recherche. La figure 10.6 en montre un exemple : les quatre actions de poser une ampoule et un couvercle sur deux lampes (gauche et droite) peuvent s'exécuter dans six ordres différents, qui seront tous distingués lors d'une recherche par A^*.

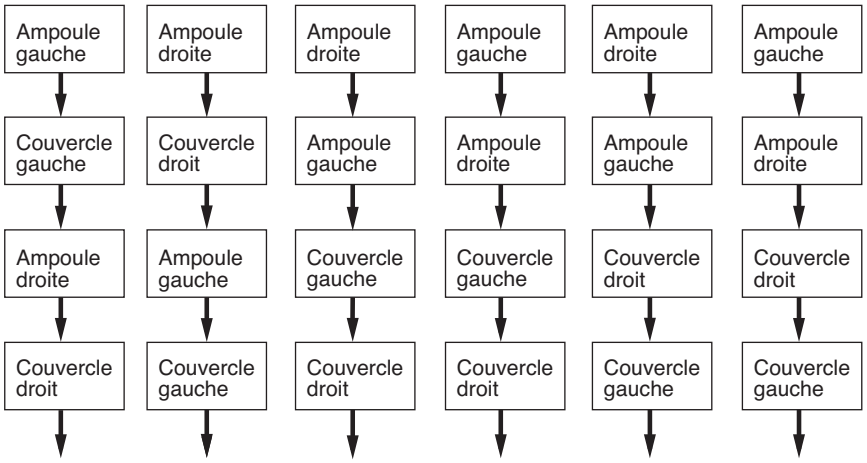


Fig. 10.6 *Plans linéaires pour mettre ampoules et couvercles.*

L'idée de la planification non linéaire est de séparer la sélection des actions qui feront partie du plan de leur ordonnancement dans une séquence précise. Un plan non linéaire est un graphe dont les nœuds sont des actions et les arcs représentent des contraintes sur l'ordre des opérations. Par exemple, on représentera les différentes séquences pour mettre les ampoules et les couvercles par un seul plan non-linéaire (fig. 10.7). Dans des situations comme celle-ci, où certaines actions peuvent être exécutées dans n'importe quel ordre, la planification non linéaire permet une représentation plus compacte. C'est un exemple de l'idée du *least commitment*, c'est-à-dire de ne pas faire de choix avant que ce soit vraiment nécessaire.

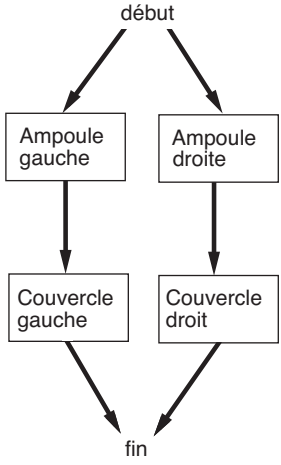


Fig. 10.7 *Plan non linéaire pour ampoules et couvercles.*

Il existe de nombreux algorithmes pour la génération de plans non linéaires. Ils impliquent une recherche relativement complexe d'opérateurs et construisent les contraintes d'ordonnancement par une analyse des préconditions et des conséquences des opérations. Au lieu d'entrer dans les détails de ces algorithmes, nous allons montrer une manière de traduire un problème de planification en un problème de satisfaction de contraintes, plus précisément en un problème de satisfiabilité (SAT). Cela permet ensuite l'application des algorithmes efficaces de PSC que nous avons vus.

Le plan construit par cette technique sera exprimé dans une autre représentation qui est une séquence linéaire d'états, où l'on associe à chaque état un ensemble d'actions qui peuvent toutes s'exécuter en parallèle. La figure 10.8 en montre un exemple. On peut montrer que s'il existe un plan non-linéaire, il existe aussi un plan sous cette forme. Donc ce choix n'affecte pas la généralité de l'approche.

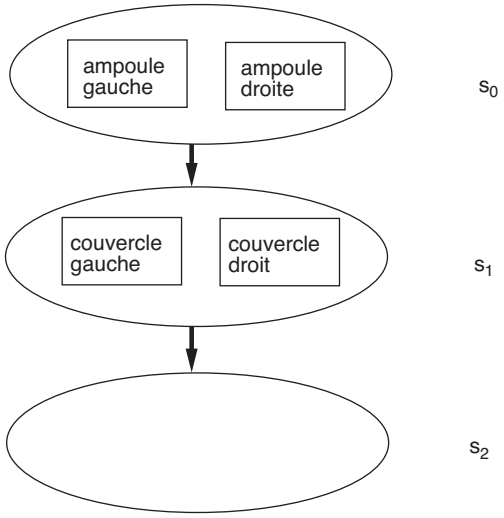


Fig. 10.8 *Une représentation alternative d'un plan non-linéaire : une séquence d'états avec des actions qui peuvent s'exécuter en parallèle.*

La méthode repose sur deux hypothèses :

- On connaît une longueur maximale l du plan, c'est-à-dire le nombre maximal d'états. Cela peut être satisfait de manière analogue à la recherche en *iterative deepening* : on choisit d'abord une limite petite que l'on augmente si aucune solution n'est trouvée.
- On a déterminé par avance toutes les actions qui pourraient être nécessaires pour obtenir un plan d'actions complet.

Nous allons maintenant décrire un problème de satisfaction de contraintes dont n'importe quelle solution correspond à un plan valable pour atteindre les conditions buts. Le PSC ne contiendra que des variables booléennes, c'est-à-

dire des variables dont la valeur sera choisie entre `vrai` et `faux`. On appelle un tel PSC un problème de satisfiabilité (SAT).

En partant de la limite l sur la longueur, on définit d'abord la séquence d'états $S_1, S_2, ..., S_{l+1}$. Le dernier état représente l'état final qui n'admet plus aucune action. Pour chaque état, le PSC contient deux types de variables :

- pour chaque action qui pourrait faire partie du plan, il existe une variable qui prendra la valeur `vrai` si l'action est exécutée dans cet état et `faux` si elle ne l'est pas,
- pour chaque propriété qui caractérise les états dans le formalisme STRIPS, il existe une variable qui indique si la propriété est vraie ou fausse dans cet état.

Notons que les valeurs des variables qui correspondent aux actions définissent un plan sous le format de la figure 10.8.

Considérons à nouveau l'exemple des lampes. Il existe quatre actions :

mettre ampoule gauche/droite $\simeq$ `ag/ad`
mettre couvercle gauche/droite $\simeq$ `cg/cd`

et supposons qu'il y ait trois états : `s0,s1,s2`. Le PSC aura alors les variables suivantes pour les actions :

```
ag(s0),ad(s0),cg(s0),cd(s0)
ag(s1),ad(s1),cg(s1),cd(s1)
```

En plus, supposons qu'on représente les états par quatre propositions :

posée ampoule gauche/droite $\simeq$ `pag/pad`
posé couvercle gauche/droite $\simeq$ `pcg/pcd`

On aura alors en plus les variables :

```
pag(s0),pad(s0),pcg(s0),pcd(s0)
pag(s1),pad(s1),pcg(s1),pcd(s1)
pag(s2),pad(s2),pcg(s2),pcd(s2)
```

Le deuxième élément d'un PSC, ce sont les contraintes sur les valeurs des variables :

- Chaque variable qui représente une action a des contraintes avec chaque variable qui représente une précondition dans le même état. Elles assurent que si l'action est exécutée, alors les préconditions doivent être satisfaites. On a donc l'expression logique :

  action $\Rightarrow$ précondition

  Par exemple, pour mettre l'ampoule gauche (`ag`), il faut que le couvercle gauche ne soit pas encore posé ($\neg$ `pcg`), donc :

  `ag(s0)` $\Rightarrow \neg$ `pcg(s0)`

  et ainsi pour toutes les autres instances de `ag`. Notons que cette expression sera représentée comme une contrainte, c'est-à-dire comme matrice de valeurs admissibles :

	pcg	
ag	vrai	faux
vrai	0	1
faux	1	1

- Des contraintes analogues existent pour les postconditions et les suppressions de chaque action, donc par exemple :

  `ag(s0)` $\Rightarrow$ `pag(s1)`

- Les conditions initiales donnent lieu à des contraintes sur les variables du premier état. Par exemple, si au début on n'a rien posé, on aura les contraintes :

```
C1:pag(s0)=faux;
C2:pad(s0)=faux;
C3:pcg(s0)=faux;
C4:pcd(s0)=faux;
```

- Les buts deviennent des contraintes sur les variables du dernier état, par exemple :

```
C1:pag(s2)=vrai;
C2:pad(s2)=vrai;
C3:pcg(s2)=vrai;
C4:pcd(s2)=vrai;
```

- Les axiomes de cadre seront également exprimés par des contraintes. Rappelons que les axiomes de cadre expriment le fait que si une variable d'état n'est pas touchée par une action, alors sa valeur au prochain état doit rester inchangée. Il est équivalent de dire que si la valeur change, il doit y avoir eu une action qui en est responsable. Donc, pour toute paire d'états successifs, et toute variable d'état, il existe une contrainte entre les variables correspondantes et toutes les actions qui peuvent l'affecter :

  si la valeur de la variable change, une des actions doit l'avoir comme postcondition.

  Par exemple, si le couvercle est posé alors qu'il ne l'était pas avant, cela doit être parce qu'on l'a posé ; s'il n'est plus posé, c'est une contradiction, car il n'y a pas d'opérateur pour l'enlever. On peut exprimer cela par les contraintes suivantes :

  $\neg$ `pcg(s0)` $\wedge$ `pcg(s1)` $\Rightarrow$ `cg(s0)`
  `pcg(s0)` $\wedge$ $\neg$ `pcg(s1)` $\Rightarrow$ $\bot$

- Supposons qu'une action a_1 ait la précondition p et une autre action a_2 ait une postcondition $\neg p$. Si les deux actions sont exécutés simultanément, il pourra y avoir un conflit car a_2 annulera la précondition de a_1. Donc, il faut établir une contrainte d'exclusion mutuelle (mutex) entre a_1 et a_2, et en général entre toute paire d'actions tel que l'une influence la précondition de l'autre.

Le problème de satisfaction de contraintes qu'on aura ainsi défini n'admettra comme solution qu'un plan valable. Pour l'exemple, une solution (qui correspond à la figure 10.8) pourrait être :

```
pag(s0) = faux; pad(s0) = faux; pcg(s0) = faux; pcd(s0) = faux;
ag(s0) = vrai; ad(s0) = vrai; cg(s0) = faux; cd(s0) = faux;
pag(s1) = vrai; pad(s1) = vrai; pcg(s1) = faux; pcd(s1) = faux;
ag(s1) = faux; ad(s1) = faux; cg(s1) = vrai; cd(s1) = vrai;
pag(s2) = vrai; pad(s2) = vrai; pcg(s2) = vrai; pcd(s2) = vrai;
```

Notons que si le nombre de variables paraît élevé, il ne croît que de façon linéaire avec le nombre d'états, d'opérateurs et de variables d'état. Il reste donc maîtrisable même pour de grands problèmes de planification.

Bien entendu, il peut y avoir plusieurs solutions et donc plusieurs plans. Il est d'ailleurs possible d'ajouter d'autres critères, par exemple on pourrait ajouter des contraintes pour tenir compte de la disponibilité des ressources. Cela permettra alors de planifier l'exécution de certaines opérations en parallèle, ce qui correspond plus à la réalité dans beaucoup de problèmes industriels tels que la productique. Il s'agit alors de satisfaire deux types de contraintes :

- précédence : un opérateur qui satisfait la précondition d'un autre doit venir avant ;
- ressource : deux opérations qui utilisent la même ressource ne peuvent pas être exécutées en même temps.

En général, la planification non-linéaire permet d'améliorer de façon significative l'efficacité de traitement d'un système de planification. Cela s'explique par le fait que l'effort de calcul le plus important, l'ordonnancement des tâches, peut se faire en utilisant des méthodes de satisfaction de contraintes plus efficaces que les méthodes de recherche simples. Il existe des techniques très efficaces qui permettent de résoudre des problèmes de satisfiabilité avec des millions de variables et rendent donc possible des plans très complexes.

10.7 Extensions pour améliorer la flexibilité

Opérateurs avec variables

Si la planification non linéaire permet déjà de profiter du principe du *least commitment* en ce qui concerne l'ordre des opérations, les opérateurs sont toujours complètement instanciés, c'est-à-dire que les objets auxquels ils se réfèrent sont décidés au moment de la sélection de l'opérateur. Pour pousser le principe du *least commitment* plus loin, on pourrait aussi imaginer d'avoir des opérateurs contenant des variables, par exemple :

```
PUTON(A, ?x)
... UNSTACK(A, ?x)
```

pour entreposer `A` sur un autre bloc et libérer la main. Cela évitera de formuler un grand nombre d'alternatives pour tous les différents endroits où on pourrait entreposer le bloc A.

Cependant, les variables qu'on introduit ainsi doivent satisfaire à des contraintes pour éviter des situations inconsistantes. Dans l'exemple, `?x` ne peut pas être égal à `A` ni à un bloc qui est utilisé dans une autre opération. Ces contraintes peuvent s'intégrer facilement dans un système de planification par satisfaction de contraintes.

Opérateurs avec disjonctions

Une autre possibilité est qu'un opérateur puisse contenir des disjonctions. S'il s'agit d'une disjonction des préconditions, par exemple $A \vee B$, on peut remplacer l'opérateur par deux copies identiques ayant comme préconditions A et B seules. Si, par contre, il s'agit d'une disjonction de postconditions, cela veut dire qu'il peut y avoir plusieurs contextes pour poursuivre la planification. Cela pourrait se produire par exemple si l'opérateur implique une mesure dont le résultat est incertain. Dans ce cas, il n'est pas possible de trouver une formulation équivalente sans disjonctions. En fait, de telles disjonctions conduisent à une explosion combinatoire des possibilités qui est à éviter si possible.

Planification hiérarchique

Souvent, le domaine dans lequel se fait la planification admet une structuration hiérarchique. La figure 10.9 montre un exemple d'une telle décomposition. En développant un plan à plusieurs niveaux d'abstraction, on peut exploiter cette structure pour améliorer l'efficacité de la planification. On construit d'abord un plan au plus haut niveau d'abstraction. On définit ainsi une suite de sous-problèmes à résoudre aux niveaux inférieurs. C'est évidemment plus efficace, mais peut cacher certaines possibilités et rendre donc le processus incomplet. Par exemple, il se peut que l'on doive reconfirmer le voyage de retour pendant le séjour, mais la décomposition ne permettrait pas d'intercaler une telle action dans la planification du séjour.

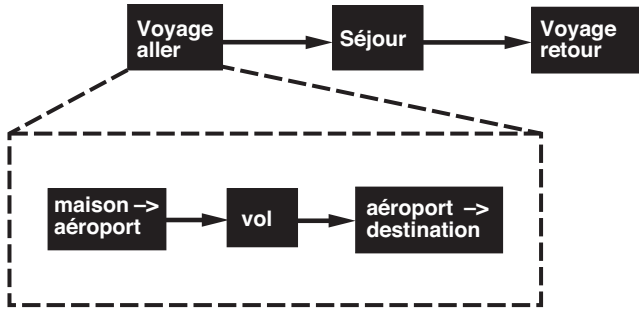


Fig. 10.9 *Exemple d'une structuration hiérarchique.*

Pour éviter ce genre d'interférence, on a developpé des techniques qui

construisent une hiérarchie sans interférences en analysant les dépendences visibles par les préconditions et postconditions des opérateurs. Par exemple, le système `Fast Downward` [52] construit ainsi un *graphe causal* au dessus de la structure du problème de satisfaction de contraintes. Ce graphe est alors la base d'heuristiques pour la résolution du problème de satisfaction de contraintes en tirant profit de son hiérarchie ainsi découverte. Cette technique permet des gains importants en complexité par rapport aux heuristiques générales pour la satisfaction de contraintes.

En pratique, la plupart des systèmes de planification utilisés actuellement sont de type non linéaire. Ils s'appliquent à une large variété de problèmes, citons par exemple :

- la planification de missions spatiales (DEVISER),
- les opérations d'un porte-avions (SIPE),
- l'opération d'un ensemble d'ascenseurs (Schindler).

En raison de l'énorme économie de ressources induite par une planification intelligente, c'est dans le domaine de la planification que l'apport de l'IA s'est particulièrement démarqué.

Littérature

La planification a été l'un des premiers problèmes considérés par l'IA. Le papier [44] considère les problèmes principaux de modélisation et formule le problème des cadres. [45] introduit le système de planification STRIPS, et [46] analyse le calcul de situations qui est sous-jacent à pratiquement tous les travaux sur la planification. [47] fait une analyse de la complexité de calcul inhérente au problème de la planification.

[48] présente la planification non linéaire qui s'est développée pendant les années 1980 et 1990. La dernière génération d'algorithmes de planification basés sur les contraintes a été introduite dans [50] pour l'algorithme Graphplan et [49] pour le système Satplan. La méthode décrite dans ce chapitre prend également des éléments de [51]. La methode hiérachique du fast downward est décrite dans [52].

On trouve une bonne synthèse récente dans le livre de Geffner et Bonet [53].

Outils - domaine public

Les systèmes de planification ont plusieurs générations et il existe de nombreux logiciels qui sont maintenant dépassés. Un système qui est relativement proche de l'état de l'art et similaire à la technique présenté ici est `Satplan` et son successeur `Madagascar`, disponibles ici :

`http://users.ics.aalto.fi/rintanen/satplan.html`

Une méthode qui gagne en efficacité en exploitant les structures hiérachiques du problème est le système `Fast Downward` dont le développeur se trouve à l'Université de Basle :

`http://www.fast-downward.org/`

Il existe aussi un site qui regroupe les développements qui se sont faits autour du formalisme de représentation `PDDL` :

`http://planning.domains/`

Application : Planification des mouvements d'ascenseurs

Les gratte-ciel contiennent une multitude d'ascenseurs qui doivent servir de nombreuses demandes. Quand tous les ascenseurs sont réglés par un même régime simple, on peut observer qu'ils ont tendance à tous servir les mêmes étages en même temps. C'est évidemment inutile, parce qu'un seul ascenseur est suffisant pour accueillir toutes les personnes qui attendent sur un étage. La société Schindler a mis au point un système de planification, basé sur un formalisme STRIPS et traduit en PSC, qui planifie les mouvements de tous les ascenseurs de façon coordonnée. Ainsi, pour chaque étage il n'y aura qu'un seul ascenseur qui viendra chercher les personnes qui y attendent. Le système réduit le temps de parcours des usagers entre 10 *et* 50%, permet d'économiser une partie des ascenseurs ou d'introduire des ascenseurs à plusieurs étages pour multiplier la capacité sans prendre plus de surface au bâtiment.

(Source : Jana Koehler et Daniel Ottiger : An AI-Based Approach to Destination Control in Elevators, *AI Magazine* 23(3), 2002, pp. 59-78.)

10.8 Exercices

Exercice 10.1 Planification - Modélisation

L'objectif de cette série d'exercices consistera à planifier la traversée d'un groupe de cannibales et de missionnaires de la rive gauche à la rive droite d'une rivière. Ceux-ci disposent pour ce faire d'un bateau à deux places, qui ne peut être piloté que par un missionnaire. Nous supposons qu'il y a en tout deux missionnaires M_1 et M_2 et deux cannibales C_1 et C_2. La traversée ne peut se faire qu'en empruntant un unique bateau B.

Cette série se compose de deux parties. Dans la première partie, correspondant à l'exercice 1, vous devrez concevoir sur papier un modèle PSC qui représente le problème de planification donné au paragraphe précédent et qui permette de le résoudre au moyen d'algorithmes de résolution de PSC. Dans la seconde partie, vous implémenterez ce modèle en Python et vous le résoudrez en appliquant le module PSC implémenté au cours des exercices précédents.

Notez qu'il n'est pas nécessaire de lire l'énoncé de la deuxième partie pour accomplir la première. Vous pourriez en fait éprouver une certaine difficulté à comprendre cet énoncé avant d'avoir lu la solution de la première partie, étant donné qu'il y fait référence.

Modules squelettes

Les modules qui suivent constituent le squelette du programme que nous allons développer. `exemple_missionnaires.py` sert en particulier à définir le problème et vous permettra de tester votre implémentation lorsque vous aurez terminé.

Module `.../moteur_psc_planification/axiomecadre.py` :

```
from moteur_psc.contrainte import Contrainte

class ContrainteAxiomeCadre(Contrainte):
    def __init__(self, var_pre, ops, var_post):
        Contrainte.__init__(self, (var_pre, var_post) + tuple(ops))

        self.var_pre = var_pre
        self.var_post = var_post
        self.vars_ops = ops

    def est_valide(self, var, val):
        print('à compléter')

    def propage(self, var):
        print('à compléter')

    def reviser(self):
        return False

    def __repr__(self):
        return 'Axiome de cadre:\n\t{}\n\t{}\n\t{}'.format(self.var_pre,
                                                          [op for op in self.vars_ops],
                                                          self.var_post)
```

Module `.../moteur_planification/operateur.py` :

```
class Operateur:
    def __init__(self, nom, precond, postcond):
        self.nom = nom

        self.precond = precond
        self.postcond = postcond

    def __repr__(self):
        return self.nom
```

Module `.../moteur_planification/etat.py` :

```
from moteur_psc_heuristique.variable_avec_label import VariableAvecLabel

class Etat:
    def __init__(self, no_etat, propositions, operateurs, etat_prec=None):
        self.no_etat = no_etat
        self.etat_prec = etat_prec

        self.operateurs = { op.nom: op for op in operateurs }

        self. vars_initiales  = {}
        self. vars_finales  = {}
```

```
        self . construire_vars_operateurs (operateurs)
        self . construire_vars_propositions (propositions)

    def construire_vars_operateurs ( self , ops):
        self . vars_operateurs = {}

        for op in ops:
            var_nom = '{} état {}'.format(op.nom, self.no_etat)
            self . vars_operateurs [op.nom] = VariableAvecLabel(var_nom,
                                                                [True, False])

    def construire_vars_propositions ( self , props):
        print('à compléter')

    def variables ( self ):
        return (list( self . vars_initiales . values ()) +
                list ( self . vars_finales . values ()) +
                list ( self . vars_operateurs . values ()))
```

Module `.../moteur_planification/planification.py` :

```
from moteur_psc.contrainte import ContrainteUnaire
from moteur_psc_heuristique.contrainte_avec_propagation import
        ContrainteAvecPropagation
from moteur_psc_heuristique.psc_heuristique import PSCHeuristique
from moteur_psc_planification.axiomecadre import ContrainteAxiomeCadre
from .etat import Etat

class Planification :
    def __init__ ( self , propositions , operateurs,
                   mutex_propositions, mutex_operateurs,
                   depart, but, nb_etats ):
        self . operateurs = operateurs
        self . mutex_propositions = mutex_propositions
        self . mutex_operateurs = mutex_operateurs

        self . depart = depart
        self . but = but

        self . nb_etats = nb_etats

        self . propositions = propositions

        self . etats = []
        self . construire_etats ()

        self . psc = PSCHeuristique(self.variables (), self . construire_contraintes ())

    def construire_etats ( self ):
        print('à compléter')

    def variables ( self ):
        # Utiliser un set évite les doublons entre variables finales et
        # initiales .
        variables = set()
        for etat in self . etats :
            variables . update(etat. variables ())
```

```
    return list(variables)

def construire_contraintes(self):
    return (self.construire_contraintes_propositions() +
            self.construire_contraintes_operateurs() +
            self.construire_contraintes_conditions() +
            self.construire_contraintes_axiomes_cadre() +
            self.construire_contraintes_initiales() +
            self.construire_contraintes_finales())

def construire_contraintes_propositions(self):
    print('à compléter')

def construire_contraintes_operateurs(self):
    print('à compléter')

def construire_contraintes_conditions(self):
    print('à compléter')

def construire_contraintes_axiomes_cadre(self):
    print('à compléter')

def construire_contraintes_initiales(self):
    print('à compléter')

def construire_contraintes_finales(self):
    print('à compléter')

def resoudre(self):
    self.psc.consistance_noeuds()
    self.psc.consistance_arcs()
    self.psc.variable_ordering()

    self.psc.forward_checking(0, True)
    self.sol = self.psc.solutions

    return self.sol

def affice_solutions(self):
    print('Recherche terminée en {} itérations'.format(self.psc.iterations))

    if len(self.psc.solutions) == 0:
        print('Aucune solution trouvée')
        return

    for sol in self.psc.solutions:
        print('Solution')
        print('========')
        for etat in self.etats:
            print('État {}: '.format(etat.no_etat))
            print('  Propositions initiales :')
            for nom, var in sorted(etat.vars_initiales.items()):
                if sol[var.nom]:
                    print('    ' + nom)

            print('  Opérateurs:')
            for nom, var in sorted(etat.vars_operateurs.items()):
                if sol[var.nom]:
```

```
                print('     ' + nom)

        print('  Propositions finales :')
        for nom, var in sorted(etat.vars_finales.items()):
            if sol[var.nom]:
                print('     ' + nom)
        print()
```

Module `.../exemple_missionnaires.py` :

```
from moteur_planification.operateur import Operateur
from moteur_planification.planification import Planification

def format_g(acteur):
    return 'g({})'.format(acteur)

def format_d(acteur):
    return 'd({})'.format(acteur)

def format_dg(bateau, pilote):
    return 'dg({}, {})'.format(bateau, pilote)

def format_gd(bateau, pilote, passager):
    return 'gd({}, {}, {})'.format(bateau, pilote, passager)


bateaux = ['B']
missionnaires = ['M1', 'M2']
cannibales = ['C1', 'C2']

acteurs = bateaux + missionnaires + cannibales

# Ajoute les propositions pour la position des acteurs.
propositions = []
print('à compléter')

# Ajoute les opérateurs de déplacement.
operateurs = []
print('à compléter')

# Ajoute les mutex de proposition (un acteur ne peut pas être sur les deux rives
# simultanément).
mutex_propositions = []
print('à compléter')

# Ajoute les mutex d'opérateurs.
mutex_operateurs = []
print('à compléter')

# Ajoute les contraintes initiales (tous les acteurs à gauche).
depart = []
print('à compléter')

# Ajoute les contraintes finales (but: tous les acteurs à droite).
but = []
print('à compléter')

# Transforme le problème de planification en PSC.
```

```
plan = Planification(propositions, operateurs,
                     mutex_propositions, mutex_operateurs,
                     depart, but,
                     nb_etats=5)
plan.resoudre()

plan. affice_solutions ()
```

Veillez à respecter la structure des dossiers telle qu'elle est reflétée dans les noms des modules ci-dessus, sous peine de devoir modifier les instructions `import`.

Exercice 10.1.1 Modélisation sur papier
Avant de commencer à coder, vous devez modéliser le problème de planification sous forme de PSC. Pour ce faire, il convient de procéder en deux étapes :

- définition du problème de planification en termes de propositions et d'opérateurs,
- définition d'un PSC correspondant, en termes de variables et de contraintes sur ces variables.

Définition du problème de planification

Un problème de planification peut être défini par trois éléments :

- Un ensemble de propositions qui décrivent complètement l'état du monde à un moment donné. Certaines de ces propositions peuvent être mutuellement exclusives. Il faut alors expliciter les contraintes d'exclusion.
- Deux ensembles d'instanciations partielles de ces propositions, qui décrivent respectivement l'état initial et l'état final, qui est le but à atteindre.
- Un ensemble d'opérateurs qui permettent de faire évoluer le monde d'un état à un autre.

Rappelons qu'une proposition est, par définition, une affirmation portant sur l'état d'une partie du monde et qui peut être vraie ou fausse. Les opérateurs, quant à eux, se définissent comme des actions dont l'exécution nécessite que certaines propositions (leurs préconditions) soient vraies ou fausses et qui ont pour conséquence d'imposer à certaines propositions (leurs postconditions) d'être vraies ou fausses.

Pour commencer, proposez donc une définition d'un problème de planification qui corresponde à la description informelle du problème telle qu'elle vous est donnée en introduction.

Privilégiez un modèle simple, qui ne contienne pas trop d'opérateurs superflus. Par exemple, il est inutile d'introduire des opérateurs pour décrire le fait qu'un missionnaire ou un cannibale embarque sur le bateau ou qu'il en débarque. Pour chaque acteur, vous pouvez utiliser une proposition indiquant s'il se trouve ou non sur la rive gauche et une autre proposition indiquant s'il se trouve ou non sur la rive droite.

Définition d'un PSC correspondant

Par définition, un PSC est décrit par :

- un ensemble de variables, qui prennent des valeurs dans des domaines définis,
- un ensemble de contraintes sur ces variables, qui définissent les combinaisons de valeurs admissibles.

Proposez un modèle PSC pour le problème de planification. Les variables doivent décrire complètement les propositions et les opérateurs lors de chaque état. Les contraintes, quant à elles, doivent exprimer les propriétés et les limitations du problème. Par exemple, un bateau ne peut contenir que deux acteurs au maximum, et l'un d'eux (le pilote) doit être un missionnaire. Ou encore, le bateau doit initialement être à gauche pour pouvoir faire la traversée de gauche à droite.

Afin de reformuler le problème de planification sous la forme d'un PSC, vous devrez faire une hypothèse sur le nombre d'états nécessaires pour l'existence d'un plan aboutissant à une solution (c'est-à-dire sur le nombre d'applications successives d'opérateurs). Notez que votre modèle PSC n'est pas obligé de se limiter aux contraintes unaires ou binaires. Il peut comporter des contraintes n-aires avec $n > 2$. Par exemple, les contraintes correspondant aux axiomes de cadre.

Solutions à la page 378

Exercice 10.2 Planification - Implémentation

Exercice 10.2.1 Construction du problème de planification
Le module `exemple_missionnaires.py` contient une routine qui, pour commencer, définit les acteurs, opérateurs, mutex et conditions de départ et de fin du problème, puis construit le problème de planification en utilisant la classe `Planification` du module `planification.py`. Une fois l'objet `Planification` construit, sa méthode `resoudre` utilise les outils de résolution de PSC développés les séries précédentes pour résoudre le problème.

Construction des propositions : rédigez le code nécessaire pour contruire les propositions du problème. Une proposition doit être simplement représentée par une string. Stockez toutes les propositions dans la liste `propositions`.

Construction des opérateurs : construisez à présent les opérateurs du problème, qui seront représentés par la classe `Operateur` du module `operateur.py`. Le constructeur de cette classe prend comme arguments trois paramètres :

- le nom de l'opérateur à créer (similaire à la représentation des propositions,
- la liste des préconditions (une liste de propositions),
- la liste des postconditions (une liste de propositions).

Spécification des mutex de propositions : ajoutez le code nécessaire à construire les mutex de propositions et qui les stockera dans une liste sous forme de tuples `(prop1, prop2)`. `prop1` et `prop2` sont ainsi deux propositions qui ne doivent pas être vraies en même temps.

Spécification des mutex d'opérateurs : ajoutez ensuite le code qui définira les mutex d'opérateurs avec le même format que les mutex de propositions, c'est-à-dire comme une liste de tuples `(op1, op2)`, avec `op1` et `op2` deux opérateurs qui ne doivent pas être exécutés en même temps.

Déclaration des contraintes initiales et finales : spécifiez maintenant les contraintes initiales et finales du problème de planification avec deux listes de tuples `(proposition, valeur)`.

Exercice 10.2.2 Implémentation des axiomes de cadre, états et planificateur
Les modules `axiomecadre.py`, `etat.py` et `planification.py` contiennent les classes et les algorithmes qui permettent de modéliser un problème de planification comme un PSC, avant de résoudre celui-ci pour trouver un plan valide. Le module PSC utilisé sera celui qui a été développé au cours de la série d'exercices 8.2 (p. 200).

La classe `ContrainteAxiomeCadre`

Cette classe, définie dans `axiomecadre.py`, est une sous-classe de la classe `Contrainte` et implémente une contrainte d'axiome de cadre pour un état S_i donné et une proposition prop donnée :

> Si $\text{prop}(S_i) =$ `False` et $\text{prop}(S_{i+1}) =$ `True`, alors, pour au moins un opérateur op qui a prop comme postcondition, on a $\text{op}(S_i) =$ `True`.

Cette contrainte est une contrainte n-aire qui porte sur plus de deux variables. Ces variables sont les attributs de la classe :

- `var_pre` est la variable $\text{prop}(S_i)$,
- `var_post` est la variable $\text{prop}(S_{i+1})$,
- `vars_ops` est la liste des variables correspondant aux opérateurs qui ont prop comme postcondition.

La méthode `est_valide` *:* il vous faut tout d'abord implémenter la méthode `est_valide`. Notez que contrairement au cas des contraintes unaires et binaires, cette méthode peut être appelée alors que toutes les variables de la contrainte ne sont pas encore instanciées (c'est-à-dire même quand leur valeur est `None`). Traitez donc ce cas en premier et faites une hypothèse de présomption de validité : la contrainte est présumée valide tant qu'on n'a pas pu prouver qu'elle était violée (c'est-à-dire tant qu'au moins une de ses variables n'est pas encore instanciée).

La méthode `propage` *:* implémentez ensuite la méthode `propage`. Cette méthode est appelée juste après qu'une valeur ait été choisie pour une variable de la contrainte. Elle tente alors de propager les conséquences de ce choix aux variables non encore instanciées de la contrainte pour réduire leurs labels. En effet, il est possible que l'assignation d'une valeur à une variable rende incompatibles certaines valeurs des labels des variables non encore instanciées. Cette méthode doit retourner `True` si et seulement si aucune inconsistance n'a été découverte.

Imaginons par exemple que la seule variable déjà instanciée est $\text{prop}(S_i) =$ `False`, et qu'on désire propager aux variables d'opérateurs les conséquences de l'assignation $\text{prop}(S_{i+1}) =$ `False`. Il est clair que la contrainte sera alors toujours vérifiée et que la méthode retournera `True` sans avoir pu découvrir aucune valeur incompatible dans les labels des variables d'opérateurs. Inversement, si l'on choisit l'assignation $\text{prop}(S_{i+1}) =$ `True`, on peut en déduire qu'au moins un des labels des variables d'opérateurs doit contenir la valeur `True`. Si ce n'est pas le cas, cette assignation est inconsistante. Si seulement une des variables d'opérateurs possède un label qui contient `True`, alors on peut d'ores et déjà conclure que seule la valeur `True` est possible pour cette variable et on peut retirer la valeur `False` de son label.

Comme vous le soupçonnez peut-être déjà sur la base de cet exemple, l'implémentation d'un algorithme de propagation performant pour une contrainte n-aire s'avère être un problème difficile dans le cas général, surtout si l'on veut accélérer la recherche en découvrant le plus tôt possible les inconsistances et en réduisant au maximum les labels des variables non encore instanciées.

Dans cet exercice, nous vous proposons d'en implémenter une version simple, peu performante mais suffisante pour le problème de planification qui nous occupe. Cette implémentation paresseuse consiste à ne tenter de réduire les labels et de détecter les inconsistances que lorsqu'il ne reste plus qu'une seule variable de la contrainte qui ne soit pas encore instanciée. Lorsque c'est le cas, vérifiez simplement les valeurs du label de cette variable une par une et retirez du label celles qui ne respectent pas la contrainte. Retournez `True` si et seulement si le label résultant n'est pas vide. Dans le cas contraire, lorsqu'au moins deux variables de la contraintes ne sont pas encore instanciées, utilisez la même hypothèse de présomption de validité que pour la méthode `est_valide` et retournez systématiquement `True` sans vous mettre en peine de réduire les labels de ces variables.

Remarque sur la méthode `reviser` *:* notez que l'implémentation de la fonction `reviser` qui vous est fournie retourne simplement `False`, c'est-à-dire qu'elle n'essaie pas de réduire les domaines des variables en appliquant la consistance des arcs. La raison en est que la consistance des arcs n'est pas définie pour des contraintes n-aires. Pour ces contraintes, on parle plutôt de consistance des arcs généralisée (*Generalized Arc Consistency*, ou GAC) : pour chaque valeur du domaine de chaque variable, il doit exister une combinaison de valeurs pour toutes les autres variables qui satisfasse la contrainte. Mais dans l'exemple simple qui nous occupe, il n'est pas nécessaire d'implémenter la GAC, de même qu'il n'est pas nécessaire d'implémenter une méthode de propagation très performante.

La classe `Etat`

Un état contient six attributs :

- `vars_initiales` : une liste de variables correspondant aux propositions au début de l'état (égales à celles qui existent à la fin de l'état précédent). Cette « liste » est en fait un dictionnaire qui associe les propositions à leurs variables respectives.
- `vars_finales` : un dictionnaire de variables associées aux propositions à la fin de l'état (égales à celles du début de l'état suivant).
- `vars_operateurs` : un dictionnaire de variables associées aux opérateurs pour cet état.
- `no_etat` : le numéro de l'état, inclus dans l'intervalle [0, `Planification.nb_etats`).
- `etat_prec` : l'objet `Etat` qui précède l'état courant dans le plan.

Le constructeur vous est donné, et appelle les méthodes `construire_vars_operateurs` et `construire_vars_propositions`, qui remplissent les attributs `vars_operateurs`, et `vars_initiales` et `vars_finales` respectivement.

La première de ces méthodes est déjà implémentée. Vous devez coder la seconde en vous inspirant de la première. Nommez les variables à l'aide du numéro de l'état au début duquel se trouve la variable. N'oubliez pas que les variables finales d'un état doivent être les mêmes que les variables initiales de l'état suivant.

La classe `Planification`

La classe `Planification` est la classe centrale du planificateur. Elle transforme un problème de planification en un PSC afin de découvrir un plan valide. Cette classe possède les attributs suivants :

- `propositions`, `operateurs`, `mutex_propositions`, `mutex_operateurs`, `depart` et `but`, qui correspondent aux listes construites dans `exemple_missionnaires.py`,
- `nb_etats` : le nombre d'états dans le plan, c'est-à-dire la longueur de celui-ci,
- `etats` : la liste des états du problème,
- `psc` : l'instance de PSC qui représente le problème modélisé en PSC.

Les méthodes de la classe : vous allez maintenant implémenter les méthodes de la classe `Planification`. Ce sont les suivantes :

- `contruire_etats` : construit tous les états de la planification et les ajoute à la liste `self.etats`. Faites en sorte que la liste soit triée par ordre croissant du numéro de l'état.
- `constuire_contraintes_propositions` : construit les contraintes binaires d'exclusion mutuelle entre propositions.

- `construire_contraintes_operateurs` : construit les contraintes binaires d'exclusion mutuelle entre opérateurs.
- `construire_contraintes_initiales` et `construire_contraintes_finales` : ajoutent les contraintes initiales sur les propositions de l'état 0 et les contraintes finales sur les propositions de l'état final.
- `constuire_contraintes_conditions` : ajoute les contraintes de pré- et post-conditions entre propositions et opérateurs.
- `construire_contraintes_axiomes_cadre` : ajoute les contraintes d'axiomes de cadre en utilisant la classe `ContrainteAxiomeCadre`.

Test du programme

Finalement, testez votre programme en lançant `exemple_missionaires.py`.

Solutions à la page 385

Troisième partie

Apprentissage automatique

Un aspect important de l'intelligence est la capacité *d'apprendre* de nouvelles connaissances sur la base d'exemples issus de l'observation du monde. Parmi les trois modes d'inférence, l'apprentissage correspond à l'induction, c'est-à-dire à des raisonnement qui tirent des conclusions universelles à partir de prémisses particulières. Par exemple, si nous considérons les propositions :

a) `oiseau(Tweety)`
b) `vole(Tweety)`
c) `(∀ x) oiseau(x) ⇒ vole(x)`

l'induction construit la règle c) à partir de a) et b).

On ne peut pas garantir que le résultat d'une telle inférence soit correct sauf si on ajoute l'hypothèse d'un monde clos :

> Tous les exemples ont été considérés, et il n'existe donc aucun contre-exemple aux règles pour autant qu'elles soient consistantes avec ces exemples.

En général, on ne peut garantir cette propriété qu'avec une certaine probabilité. Même si cette probabilité augmente avec le nombre d'exemples, on n'atteint jamais une certitude absolue. Par contre, il existe une théorie de l'apprentissage qui permet de donner des bornes à la probabilité qu'un résultat soit correct en fonction du nombre d'exemples utilisés dans l'apprentissage (expliquée à la section 12.2).

On peut distinguer deux types d'apprentissage :

- L'*apprentissage supervisé* : on fournit au système des exemples avec la bonne classification ou la bonne prédiction ; le système doit alors reproduire cette classification ou prédiction aussi bien que possible. L'apprentissage supervisé s'applique par exemple à l'apprentissage de règles pour reconnaître les mauvais payeurs ou les conditions de dysfonctionnement d'un appareil.
- L'*apprentissage non supervisé* : le système doit lui-même proposer une classification raisonnable, par exemple pour optimiser ses propres critères de performance du système. L'apprentissage non-supervisé peut ainsi servir à grouper les clients d'un site web en classes typiques pour optimiser leur structure d'accès ou à classifier de segments de génome pour distinguer les portions importantes de celles qui ne le sont pas.

L'apprentissage supervisé vise à obtenir un modèle capable de prédire une variable-« cible » pour de nouveaux exemples, en utilisant un ensemble d'exemples pour lesquels la valeur de cette variable est déjà connue. On distingue entre la *classification*, dans laquelle la variable-cible est catégorique, et la *régression*, dans laquelle la variable-cible prend des valeurs numériques. Les modèles peuvent en outre se diviser en deux catégories :

- des modèles simples, appelées également *paramétriques*, qui couvrent tous les exemples avec une seule expression, dont il s'agit de trouver les paramètres,
- des modèles structurés, appelés également *non paramétriques*, dans lesquels on décompose l'ensemble des exemples en sous-ensembles afin d'apprendre une classification simple pour chacun d'entre eux.

L'apprentissage non-supervisé ne s'appuie pas sur des variables-cibles préalablement données, mais vise à obtenir un modèle qui regroupe des exemples similaires. Il est particulièrement utile quand on dispose d'une grande quantité de données non-interprétées. Le but de l'apprentissage est alors de découvrir la structure inhérente à ces données, en général sous la forme d'un regroupement en classes similaires (des *clusters*).

Comme l'apprentissage s'observe avant tout chez les êtres vivants, plusieurs techniques s'inspirent de la biologie. Nous considérons ainsi notamment les réseaux de neurones artificiels, qui imitent la structure du cerveau sous une forme idéalisé, et les algorithmes génétiques. Tous les deux permettent un apprentissage supervisé.

Littérature

Parmi les nombreux ouvrages qui traitent de l'apprentissage automatique, citons-en deux : le livre de Bishop [54] et celui de Murphy [55]. [56] presente le domaine sous un angle de la statistique.

Certains livres sont accompagnés de logiciels qui permettent d'expérimenter avec les techniques : par exemple, [57] présente la boîte à outils open-source WEKA, une collection qui contient pratiquement tous les algorithmes d'apprentissage. Elle constitue un outil extraordinaire pour construire des applications de l'apprentissage automatique.

Pour une perspective de l'influence que les techniques d'apprentissage promettent sur l'informatique, le livre de Domingos [58] est intéressant.

Chapitre 11

Induction de modèles paramétriques à partir d'exemples

Dans ce chapitre, nous allons nous occuper de l'apprentissage d'un seul modèle simple qui doit couvrir du mieux possible tous les exemples fournis à l'entrée. Un modèle simple est une forme type qui contient certains paramètres, comme par exemple :

- un polynôme d'un certain degré k,
- une expression logique,
- un hyperplan dans un espace de traits qui sépare des exemples positifs et négatifs.

Apprendre un modèle simple revient en général à déterminer les *paramètres* de la classification qui donnent la meilleure approximation des exemples considérés. Par exemple, il existe des techniques pour trouver les paramètres d'un polynôme qui réduisent au minimum l'erreur d'approximation d'un ensemble de points de mesure, où l'on exprime l'erreur par la somme des carrés des différences. On parle donc aussi d'un apprentissage paramétrique.

Considérons d'abord l'apprentissage d'expressions logiques tels que des règles. Le but de l'apprentissage est donc de construire une règle du type :

`condition logique` $\Rightarrow$ `classification`

où la `condition logique` décrit un *concept* ou l'ensemble des situations pour lesquelles `classification` est vraie. L'apprentissage construit donc une *classification* qui permet de classer une situation en *instance* ou *non-instance* du concept.

Par exemple, supposons que vous fassiez un voyage dans un pays tropical, où la nourriture a tendance à contenir des piments très piquants. Vous souhaitez alors utiliser vos expériences des types de piments :

`grand,allongé,rouge,piquant`
`grand,rond,vert,`$\neg$`piquant`
`petit,allongé,jaune,`$\neg$`piquant`
`petit,allongé,rouge,piquant`
`petit,rond,rouge,`$\neg$`piquant`

pour trouver une règle :

`allongé,rouge` $\Rightarrow$ `piquant`

qui vous permettra à l'avenir d'éviter les piments trop piquants.

À l'entrée du système d'apprentissage, nous avons donc un ensemble d'exemples classés :

- un ensemble $\mathcal{P}$ d'instances d'un concept et
- un ensemble $\mathcal{N}$ de non-instances du concept.

L'apprentissage a alors pour but de trouver une description logique qui couvre toutes les instances de $\mathcal{P}$ et aucune instance de $\mathcal{N}$.

11.1 Représentation

Les *exemples* peuvent être représentés par des traits, qui sont soit des attributs à valeurs binaires (vrai/faux) :

`piquant,rouge,...`

des attributs à valeurs multiples :

`couleur = {rouge,bleu,vert},longueur` $\in$ `[1..4.5]`

ou des relations :

`(couleur ?x rouge),...`

Le choix des traits à utiliser dans la représentation est essentiel pour la performance du système d'apprentissage. Le problème est complexe car il n'y a pas de critère qui permette à coup sûr de savoir si un attribut est important pour la classification ou pas. Souvent, l'utilité des attributs peut se révéler uniquement dans leur combinaison. Considérons par exemple trois attributs $x, y, z \in [-1..1]$ qui sont dérivés du contenu de textes e-mail. Le message est considéré comme un spam si $x \cdot y \cdot z > 0$. Par contre, aucun des attributs x, y et z ni aucune paire d'entre eux n'a de corrélation avec le spam. Dans ce cas, il est impossible de déterminer l'importance de ces attributs sans les considérer tous ensemble. En général, pour être sûr de considérer tous les attributs importants, il faudra examiner toutes les combinaisons et leur pertinence pour la tâche d'apprentissage, ce qui revient à une recherche combinatoire coûteuse. Malheureusement, on ne connaît pas à ce jour de meilleure méthode, et le problème de la sélection d'attributs reste un important problème ouvert !

La représentation la plus générale d'une description apprise serait une expression logique quantifiée :

$$\texttt{homme}(?\texttt{x}) \wedge (\exists ?y)\texttt{père}(?\texttt{x}, ?\texttt{y})$$

Mais ce genre de descriptions n'est en général pas applicable à cause de la complexité de l'apprentissage. Il est plus réaliste d'utiliser une expression sans quantificateurs :

$$\texttt{homme}(?\texttt{x}) \wedge \texttt{père}(?\texttt{x}, \texttt{Charles})$$

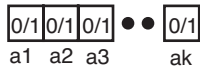
Or toute expression sans quantificateurs possède une forme *normale*, sous la forme d'une disjonction de conjonctions :

$$(\texttt{p1} \wedge \texttt{p2} \wedge \texttt{p3}) \vee (\texttt{p4} \wedge \texttt{p2}) \vee (\texttt{p1} \wedge \texttt{p5} \wedge \texttt{p6})$$

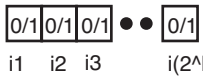
En interprétant chaque conjonction comme un sous-concept, la forme normale se réduit à un ensemble de sous-concepts. Si on divise au préalable les exemples en sous-ensembles correspondant aux différents sous-concepts, on peut alors apprendre les conjonctions qui représentent ces sous-concepts en appliquant à chaque sous-ensemble un algorithme adapté à l'apprentissage d'une conjonction d'attributs. C'est l'idée de base des classifications structurées que nous verrons plus en détail dans le chapitre 12.

11.2 Biais

L'apprentissage peut en principe être effectué par un algorithme de recherche qui parcourt l'espace de toutes les descriptions possibles. En fait, pour le problème général on ne connaît aucun autre algorithme. Considérons cependant la complexité d'une telle recherche. L'apprentissage d'un concept divise les exemples en deux sous-classes : membres et non-membres. Si les instances sont décrites par k attributs binaires différents, il y a alors 2^k instances différentes possibles :



et donc 2^{2^k} manières différentes d'attribuer l'ensemble des instances à une classes ou à l'autre :



Par exemple, avec 10 attributs, il y aura plus de 10^{300} possibilités. Une recherche exhaustive à travers toutes ces possibilités est clairement exclue.

Pour combler ces lacunes, on introduit des *biais* dans la recherche sous-jacente à l'apprentissage. Ces biais peuvent se situer à trois niveaux :

- Dans la représentation même : on limite par exemple les descriptions considérées à des conjonctions d'attributs et de négation d'attributs. Nous définissons alors pour chaque attribut trois valeurs possibles (positif, négatif ou absent). Cela donne lieu à un espace de descriptions qui comprend 3^k possibilités.
- Dans la recherche : on utilise des heuristiques qui privilégient des descriptions simples qui sont souvent plus adéquates.
- Par le domaine d'application : souvent, certaines combinaisons d'attributs peuvent être exclues *à priori* en s'appuyant sur la connaissance du domaine dans lequel se déroule l'apprentissage. Par exemple, pour reconnaître un mauvais payeur le prénom ne devrait jouer aucun rôle.

11.3 Apprentissage par recherche

Un processus de recherche se définit par deux éléments : les nœuds et la fonction de successeur. Dans le cas de l'apprentissage de descriptions conjonctives, les nœuds correspondront à des descriptions candidates, et la fonction de successeur soit à une spécialisation, soit à une généralisation de cette description. Chaque nœud sera filtré par les ensembles $\mathcal{P}$ (instances du concept) et $\mathcal{N}$ (non-instances du concept) ; toute description qui :

- n'est pas satisfaite par un exemple de $\mathcal{P}$ ou
- est satisfaite par un exemple de $\mathcal{N}$

n'est pas valable et ne sera plus développée par la suite.

La spécialisation et la généralisation sont deux règles opposées qui donnent lieu à des algorithmes très différents. La spécialisation part d'un concept général (au début, une description vide) qui s'applique à n'importe quel exemple. On ajoute ensuite des attributs qui en limitent l'applicabilité jusqu'à ce que la description ait atteint un maximum de spécificité.

La généralisation fonctionne dans le sens inverse : on part d'un exemple précis et on retire des attributs jusqu'à ce que la description soit suffisamment générale pour couvrir l'ensemble des exemples positifs. Dans les deux cas, chaque description aura plusieurs successeurs et l'algorithme effectuera donc une recherche.

11.3.1 Apprentissage par spécialisation

Pour l'apprentissage par spécialisation, on commence avec une description vide et on y ajoute des attributs jusqu'à ce que la description ne couvre plus aucun exemple négatif. On applique donc un algorithme de recherche, par exemple en profondeur d'abord ou en largeur d'abord, dans lequel :

- les nœuds sont des descriptions candidates (conjonctions d'attributs),
- les successeurs d'un nœud sont obtenus en rajoutant un attribut de plus à la description,
- le nœud initial est une description vide,
- un nœud final est une description qui couvre tous les exemples de $\mathcal{P}$, et aucun exemple de $\mathcal{N}$.

Notons qu'on peut éliminer les nœuds qui ne couvrent pas tous les exemples de $\mathcal{P}$, car il ne serait pas possible de les couvrir en spécialisant encore la description. Aucun de leurs successeurs ne pourrait donc être un nœud final.

En pratique, l'apprentissage par spécialisation n'est pas toujours très efficace, car il implique une recherche peu ciblée parmi de nombreuses descriptions possibles. Il est surtout utilisé quand il s'agit d'apprendre des descriptions partielles pour construire des classifications structurées.

11.3.2 Apprentissage par généralisation

Dans l'apprentissage par généralisation, il convient de construire comme description de départ une description $\mathcal{D}_0$, constituée par l'intersection de tous les

attributs communs à tous les exemples positifs. Aucun sur-ensemble de $\mathcal{D}_0$ ne pourra s'appliquer à tous les exemples positifs, et donc toute description valable D doit être un sous-ensemble : $D \subseteq \mathcal{D}_0$.

Ainsi, on peut savoir tout de suite que si $\mathcal{D}_0$ couvre également un exemple négatif, il n'existe pas de description conjonctive valable !

Par exemple, pour les piments :

```
grand,allongé,rouge,piquant
grand,rond,vert,¬piquant
petit,allongé,jaune,¬piquant
petit,allongé,rouge,piquant
petit,rond,rouge,¬piquant
```

les attributs partagés par tous les exemples positifs sont les suivants :

$\mathcal{D}_0$ = `allongé, rouge`

On peut ensuite simplifier $\mathcal{D}_0$ pour trouver une description qui ne couvre aucun exemple négatif. Cela peut se faire ici aussi par un algorithme de recherche. Dans ce cas cependant, les successeurs sont obtenus par généralisation en écartant des attributs. Une description correspond à un nœud final si elle ne couvre plus aucun exemple négatif.

Au lieu de commencer par la conjonction des attributs communs, on peut aussi débuter la recherche par des exemples spécifiques, appelés des *noyaux*. Cela permet de trouver un résultat partiel même quand il n'existe pas de description conjonctive qui couvre tous les exemples. Une telle recherche est surtout intéressante si, outre la règle de généralisation simple qui consiste à ôter des attributs un à un, on applique également d'autres règles de généralisation, telles que :

- changer des constantes en variables :
  `Nom=François` $\Rightarrow$ `Nom=x`
- monter dans un arbre de généralisation de concepts :
  `type = Macintosh` $\Rightarrow$ `type = pomme` $\Rightarrow$ `type = fruit`

En admettant de nouveaux prédicats, on peut aussi :

- ajouter une disjonction d'intervalles :
  `[1..5]` $\Rightarrow$ `[1..5]` $\vee$ `[7..10]`
- fermer un intervalle :
  `[1..4]` $\vee$ `[5..10]` $\Rightarrow$ `[1..10]`
- trouver les extrêmes d'un ordre partiel :
  `ON(A,B),ON(B,C)` $\Rightarrow$ `LOWEST(C), HIGHEST(A)`

L'avantage de ces règles, dont la plupart sont utilisables soit en généralisation soit en spécialisation, est qu'on peut parfois trouver des descriptions pour des jeux d'exemples qui n'admettraient autrement aucune possibilité de description conjonctive. Par exemple, supposons une nouvelle instance positive :

```
petit,allongé,vert,piquant
```

qui fait qu'il n'y a aucune description conjonctive qui réponde à nos critères, c'est-à-dire qui recouvre tous les exemples positifs. En utilisant une hiérarchie où les couleurs `rouge,vert` sont une instance de `foncé`, on peut généraliser à

```
allongé,foncé
```

qui est ainsi une description valable du concept des piments piquants.

Néanmoins, il y aura des cas où il sera impossible de trouver une seule description conjonctive. Par exemple, si nous avons les exemples positifs $\mathcal{P}$:

```
grand,allongé,rouge,piquant
petit,allongé,rouge,piquant
petit,rond,vert,piquant
```

et les exemples négatifs $\mathcal{N}$:

```
grand,rond,vert,¬piquant
petit,allongé,jaune,¬piquant
petit,rond,rouge,¬piquant
```

il n'y a aucune conjonction d'attributs qui réponde aux critères d'une description valable, et il faut avoir utiliser des classifications structurées que nous allons voir au chapitre 12.

11.4 Frontières de décision

On a souvent affaire à des attributs qui ne sont pas des propositions logiques, mais qui prennent des valeurs dans des domaines continus. Par exemple, pour détecter les dysfonctionnements d'une machine, on part de mesures continues telles que des pressions, températures, etc. L'apprentissage doit alors apprendre des frontières de décision dans un espace à plusieurs dimensions continues, et on ne peut pas les apprendre par une recherche entre différentes descriptions. Par contre, il existe des algorithmes spécialisés pour apprendre de telles frontières.

On se limite en général à des frontières linéaires. La figure 11.1 en montre un exemple. Ces frontières se décrivent sous le format général suivant :

$$w_1 \cdot x_1 + w_2 \cdot x_2 + ...w_n \cdot x_n \geq s$$

ou, en remplaçant le seuil s par $-w_0$:

$$\delta(W, X) = w_0 \cdot 1 + w_1 \cdot x_1 + w_2 \cdot x_2 + ...w_n \cdot x_n \geq 0$$

Un exemple décrit par le vecteur X est alors classé selon qu'il se trouve d'un côté ou de l'autre de la frontière :

$$C(X) = \begin{cases} 1(+) & \text{si } \delta(W, X) \geq 0 \\ 0(-) & \text{autrement} \end{cases}$$

S'il existe une telle frontière qui sépare les deux classes, on dit qu'elles sont *linéairement séperables.*

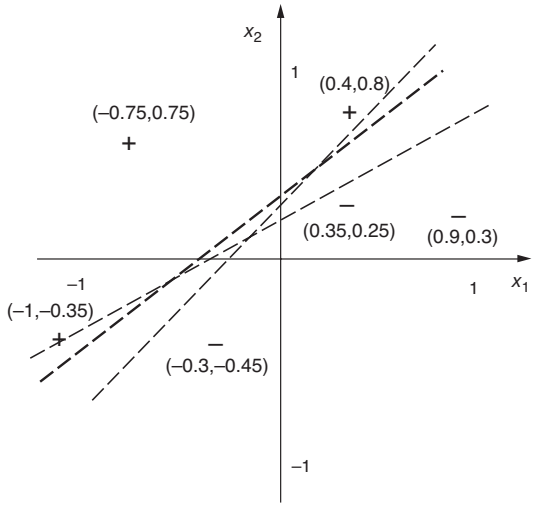


Fig. 11.1 *Frontières de décision dans un espace continu.*

Le problème est alors comment apprendre une telle frontière à partir d'exemples. Un premier algorithme est donné par la règle du *perceptron*, qui s'inspire d'un ancien modèle du fonctionnement d'un neuron humain. L'algorithme du perceptron, décrit à la figure 11.2, est très simple : on commence avec une règle quelconque et on s'en sert pour classifier les exemples. Chaque fois que la règle se trompe, on applique une correction qui consiste simplement à ajouter ou à déduire l'exemple du vecteur des poids. Le *seuil* w_0 reste égal à 1. Le paramètre σ indique la vitesse de convergence désirée ; il peut conduire à des problèmes de stabilité s'il est trop élevé. On peut prouver que si les classes sont linéairement séparables, et il est en fait possible de trouver une (seule) frontière de décision linéaire qui sépare les exemples des différentes classes, la règle du perceptron en trouvera une. Cependant, si une telle frontière n'existe pas, la règle ne converge vers aucun résultat.

1: **Function** PERCEPTRON($\mathcal{P}$,$\mathcal{N}$)
2: W $\leftarrow$ (1,0,0,...,0)
3: **for** $X_i \in \mathcal{P} \cup \mathcal{N}$ **do**
4: **if** $C(W, X_i) \neq class(X_i)$ **then**
5: **if** $class(X_i) = +$ **then**
6: W $\leftarrow$ W + $\sigma \cdot X_i$
7: **else**
8: W $\leftarrow$ W - $\sigma \cdot X_i$
9: **return** W

Fig. 11.2 *Règle du perceptron pour l'apprentissage d'une seule frontière de décision linéaire.*

La trace suivante donne un exemple du fonctionnement de la règle du perceptron, avec $\sigma = 1$:

Exemple	W_{init}	Class.	W_{modif}
(0.4,0.8) +	(0,0)	-	(0.4,0.8)
(0.35,0.25) -	(0.4,0.8)	-	(0.4,0.8)
(0.9,0.3) -	(0.4,0.8)	-	(0.4,0.8)
(-0.75,0.75) +	(0.4,0.8)	-	(-0.35,1.55)
(-1,-0.35) +	(-0.35,1.55)	-	(-1.35,1.2)
(-0.3,-0.45) -	(-1.35,1.2)	-	(-1.35,1.2)
(-1,-0.35) +	(-1.35,1.2)	-	(-2.35,0.85)
(0.4,0.8) +	(-2.35,0.85)	-	(-1.95,1.65)
....			

On peut observer qu'un seul exemple doit passer plusieurs fois, et que la convergence peut être très lente.

Si la règle du perceptron est utile parce qu'elle a la garantie de converger vers une frontière adéquate lorsqu'une telle frontière existe, elle ne débouche pas forcément sur la meilleure solution. Parmi les différentes frontières possibles, on préférerait en effet trouver celle qui *maximise* la séparation des exemples, comme le montre la figure 11.3. On souhaite donc trouver des poids W tels que :

- pour toutes les instances positives, la distance δ à la frontière est $> \delta_0$,
- pour toutes les instances négatives, la distance $\delta < -\delta_0$ et
- δ_0 est maximal.

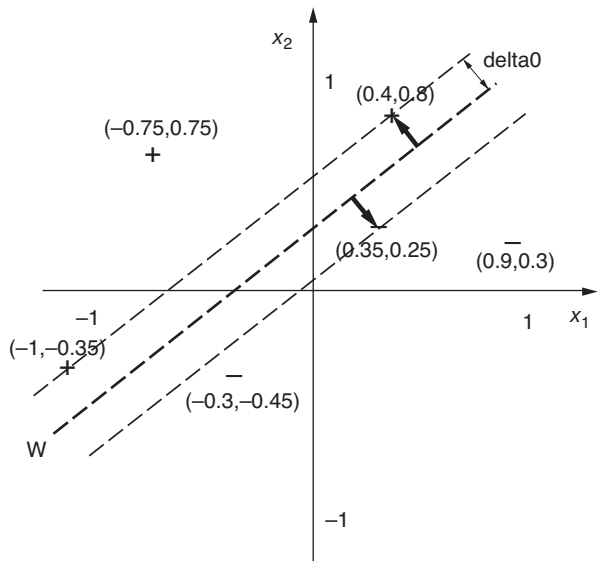


Fig. 11.3 *La meilleure frontière de décision est celle qui maximise la distance entre les exemples.*

Ce choix assure que la frontière sépare au mieux les exemples, et s'avère le meilleur critère en pratique. Elle est donc la solution d'un problème d'optimisation quadratique :

- objectif : maximiser δ_0 = minimiser $|W|^2$
- contraintes :
  pour tout exemple positif, $X \cdot W > \delta_0$
  pour tout exemple négatif, $X \cdot W < -\delta_0$

Un tel problème peut être résolu en temps polynomial par rapport au nombre d'exemples, donc de façon assez efficace. On peut observer que dans la solution optimale, certains exemples se trouveront à une distance δ_0 de la surface de décision. On appelle les vecteurs entre la surface de décision et ces exemples les plus proches des *vecteurs de support* (*support vectors*), et la méthode s'appelle donc *support vector machine (SVM).*

Lorsque les exemples ne sont pas séparables par une frontière linéaire, on peut soit :

- Admettre que certains exemples ne sont pas correctement classifiés. On peut alors déduire la somme des erreurs de δ_0 lors de la maximisation de la surface et ainsi optimiser une combinaison des critères.
- Introduire une transformation non linéaire qui les rend séparables (*Kernel function*).

Le recours à une fonction de noyau (*kernel function*) est souvent considéré comme un élément principal des SVM. Il consiste à transformer les coordonnées en appliquant une fonction non linéaire $\phi(x)$ et ainsi à projeter les exemples dans un autre espace d'exemples. Dans l'exemple de la figure 11.4, on peut utiliser les fonctions :

$$\begin{aligned} x'_1 &= \phi(x_1) = x_1^2 \\ x'_2 &= \phi(x_2) = x_2^2 \end{aligned}$$

pour rendre séparables les exemples positifs et négatifs qui ne l'étaient pas selon les coordonnées initiales.

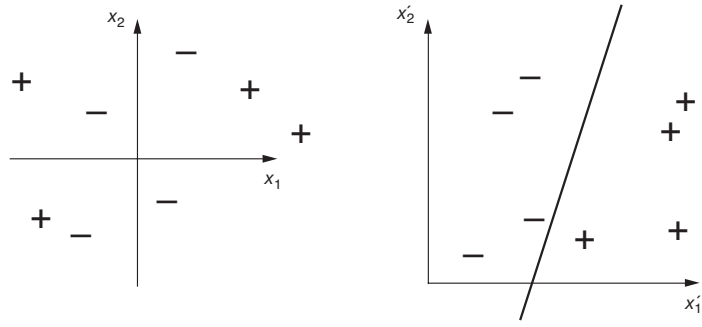


Fig. 11.4 *L'application d'une fonction de noyau rend séparables les exemples positifs et négatifs.*

Notons que l'optimisation qui est sous-jacente à la construction d'une SVM n'utilise que les produits scalaires de vecteurs :

$$\min \underline{\phi(W)} \cdot \underline{\phi(W)}$$

sous les contraintes définies par les exemples X_i :

$$\underline{\phi}(X_i) \cdot \underline{\phi}(W) > 1 \quad \text{ou} \quad < -1 \quad \text{selon la classe de l'exemple.}$$

On n'a donc pas vraiment besoin de construire $\phi(W)$ et $\phi(X_i)$; il suffit de construire leur produit scalaire. On remplace donc $\phi(X) \cdot \phi(Y)$ par une *fonction de noyau* K(X,Y), et on peut alors considérer des fonctions noyaux particulièrement simples, comme par exemple :

$$\begin{aligned} K(X,Y) &= (X \cdot Y)^2 \Rightarrow \underline{\phi} = (x_1^2, \sqrt{(}2)x_1x_2, x_2^2) \\ K(X,Y) &= x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2 = \underline{\phi}(X) \cdot \underline{\phi}(Y) \end{aligned}$$

Les contraintes de la SVM s'expriment alors sur la fonction noyau :

$$\min K(W,W), \text{ sous contraintes } K(X_i,W) > 1/ < -1$$

et on n'a jamais besoin de construire ϕ.
Ceci est particulièrement intéressant pour certaines fonctions noyaux, par exemple la *Radial basis function* :

$$K(X,Y) = e^{-|X-Y|^2/2\sigma^2}$$

pour laquelle ϕ serait d'une dimensionnalité infinie. Il est donc heureux que l'on puisse se passer de la calculer explicitement. En général, on choisit comme fonctions noyaux des expressions qui mesurent la similarité entre exemples. La *radial basis function* en montre un bon exemple, car elle prend des valeurs d'autant plus élevées que les vecteurs arguments sont plus semblables.
Les *support vector machines* sont beaucoup utilisées en pratique, surtout pour la reconnaissance de formes, mais également dans de nombreuses applications de classification.

11.5 Régression

La régression est une technique statistique qui a pour but de prédire la valeur d'une variable aléatoire y sur la base d'un ensemble de variables $x_1, .., x_k$. On parle aussi d'une *explication* de la valeur de y en termes des x_i. La forme la plus connue est la régression linéaire, dans laquelle le modèle prend la forme :

$$y = w_0 + w_1x_1 + ... + w_kx_k + \mathcal{L}$$

où $\mathcal{L}$ est une erreur résiduelle. L'apprentissage d'un tel modèle doit donc trouver les paramètres w_i qui minimisent $\mathcal{L}$ sur l'ensemble des exemples.

Le plus souvent, on suppose que l'erreur résiduelle pour l'exemple i est distribuée selon une distribution Gaussienne :

$$p(i) = p(\mathcal{L}(i)) = \alpha e^{-w\mathcal{L}(i)^2}$$

Selon le principe de la maximisation de vraisemblance, on choisit les paramètres du modèles de façon à maximiser la probabilité $\Pi_i p(i)$ d'observer l'ensemble des exemples. Comme il est plus simple de maximiser une somme, et comme la fonction du logarithme est monotone, il est préférable de maximiser $\sum_i \log p(i)$ à la place. Dans le cas d'une distribution Gaussienne, cela revient à *minimiser* :

$$\sum_i \log p(i) \propto \sum_i \mathcal{L}(i)^2$$

ce qui est aussi connu comme l'approximation des *moindres carrés*.

Les w_i qui correspondent à une telle approximation aux moindres carrés peuvent être obtenus en résolvant un système de k équations linéaires. La figure 11.5 montre un exemple de régression linéaire.

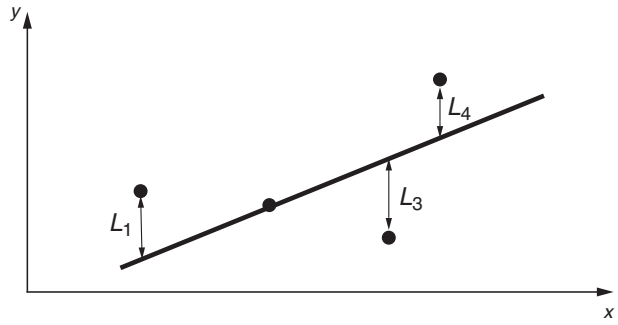


Fig. 11.5 *Exemple d'une régression linéaire pour trouver un modèle qui couvre quatre exemples. L'erreur résiduelle est la somme des* $L_1, .., L_4$.

La régression peut aussi s'étendre facilement à une approximation par polynômes, car un polynôme est une fonction linéaire dans un espace qui inclut les termes polynomiaux $x_i \cdot x_j$ comme dimensions supplémentaires. Par exemple, pour 2 traits x_1 et x_2, une régression polynômiale de degré 2 :

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_{11} x_1^2 + w_{12} x_1 x_2 + w_{22} x_2^2 + \mathcal{L}$$

est linéaire dans un espace de 5 au lieu de 2 dimensions. On peut ainsi étendre la régression linéaire à des courbes plus complexes, et obtenir un modèle avec moins d'erreur. Pour le même exemple que montre la figure 11.5, on peut par exemple obtenir une erreur beaucoup plus faible, comme le montre la figure 11.6.

Cependant, un tel procédé a tendance à produire des modèles peu vraisemblables, qui obtiennent de bons résultats sur les exemples utilisés pour l'apprentissage, mais pas sur de nouveaux exemples (comme les cercles dans la figure 11.6). Il y a en fait un compromis à faire entre :

- le biais du modèle : la capacité du modèle à représenter la réalité,
- la variance de l'apprentissage : erreur du modèle induite par une courbe déterminée trop fortement par les erreurs d'observation.

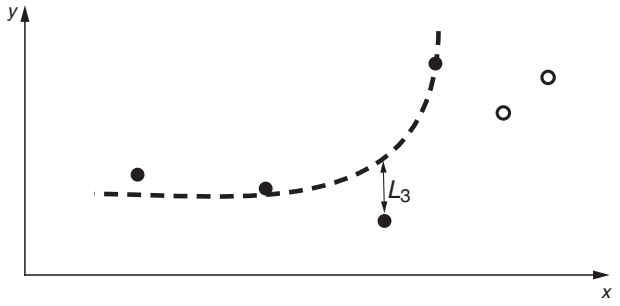


Fig. 11.6 *Exemple d'une régression polynomiale pour trouver un modèle qui couvre quatre exemples. L'erreur résiduelle est beaucoup plus faible, mais d'autres exemples tels que les cercles produisent une erreur nettement plus grande.*

Un modèle trop complexe va introduire trop de variance, un phénomène qu'on appelle le *surapprentissage* (*overfitting*). Pour éviter ce phénomène, on introduit une *régularisation*, qui va pénaliser des modèles peu vraisemblables.

La régularisation peut être vue comme un remplacement du critère de maximisation de la vraisemblance $P(X|W)$ par un critère de maximisation de la probabilité du modèle lui-même :

$$p(W|X) = p(X|W)\frac{p(W)}{p(X)}$$

Comme $p(X)$ n'est pas influencé par W, il suffit de choisir W pour *minimiser* :

$$-\ln p(X|W) - \ln p(W)$$

ce qui revient à modifier le critère des maximisation de la vraisemblance par l'ajout du terme $\ln p(W)$, qu'on appelle *régularisateur*. Son influence dépend de la probabilité qu'on attribue au modèle W :

- si tous les W sont equiprobables, il n'a pas d'influence ;
- si on s'attend à ce que les w_i soient distribués selon Gaussienne avec moyenne 0, alors le régularisateur sera $\sum_k -w_k^2$; ceci est le cas le plus courant ;
- si la distribution des w_i est Laplacienne (exponentielle), alors le régularisateur sera $\sum_k -w_k$; ce cas s'appelle *LASSO* ;
- d'autres distributions supportent d'autres régularisateurs.

Le régularisateur permet de combattre le surapprentissage de manière simple et efficace. Des instruments analogues sont également utilisés dans d'autres techniques d'apprentissage, en imposant une penalité pour des modèles complexes. Leur justification est souvent moins claire que dans le cas de la régression.

11.6 Classification par régression logistique

Pour appliquer la régression à la classification, il faut fixer y à deux valeurs discrètes, par exemple 0 ou 1. Ceci ne correspond évidement plus à l'hypothèse d'une erreur Gaussienne, et il convient donc d'introduire une transformation de y :

- D'abord, on rend y continu en utilisant la probabilité $p(y = 1|X)$. Comme on peut le voir dans l'exemple de la figure 11.7, la régression linéaire donne toujours une fonction qui dépasse l'intervalle [0..1], et qui peut donc pas toujours être interprétée comme probabilité.
- Ensuite, on limite le domaine à l'intervalle [0..1] par la transformation logistique :

$$p(y = 1|X) = \frac{e^{w_0+w_1x_1+...+w_kx_k}}{1 + e^{w_0+w_1x_1+...+w_kx_k}}$$

Cette opération limite le domaine à l'intervalle [0..1] par une transformation sigmoïde, comme montré dans la figure 11.7, où le modèle serait :

$$p(y = 1|x) = \frac{e^{w_0+w_1x}}{1 + e^{w_0+w_1x}}$$

La classification par régression logistique apprend donc les paramètres w_i du modèle qui minimisent les erreurs sur l'ensemble des exemples fournis. Plus précisément, nous souhaitons trouver les paramètres w qui sont les plus probables étant donnés les exemples D, c'est-à-dire maximiser :

$$p(W|D) = p(D|W)\frac{p(W)}{p(D)}$$

par la règle de Bayes. Comme $p(D)$ n'est pas influencé par w, il suffit de maximiser le produit des deux autres termes. Il s'avère d'ailleurs là aussi plus pratique de maximiser leur logarithme :

$$\ln p(D|W) + \ln p(W)$$

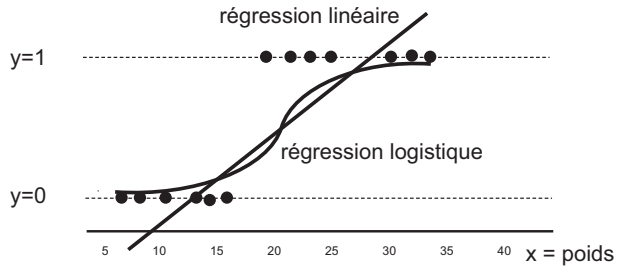


Fig. 11.7 *Régression de la classe y (1=chien, 0 = chat) en fonction du trait x (poids).*

Considérons d'abord une probabilité uniforme pour tous les choix de w. Le terme $p(w_0, w_1, ..., w_k) = c$ est constant et peut être ignoré. On doit donc maximiser :

$$p(D|W) = \prod_i p(y = y_i|W, X_i)$$

où i varie sur tous les exemples. On observe que :

$$p(y = 0|X) = 1 - \frac{e^{W \cdot X}}{1 + e^{W \cdot X}} = \frac{1}{1 + e^{W \cdot X}}$$

et

$$p(y = +1|X) = \frac{e^{W \cdot X}}{1 + e^{W \cdot X}} = \frac{1}{1 + e^{-(W \cdot X)}}$$

et donc

$$p(y = y_i|X) = \frac{1}{1 + e^{(1-2y_i)(W \cdot X_i)}}$$

La maximisation de la probabilité revient donc à choisir w pour maximiser :

$$\sum_i - \ln(1 + e^{(1-2y_i)(W \cdot X_i)}) \tag{11.1}$$

et existe des routines d'optimisation convexe qui peuvent fournir la solution optimale de façon stable et efficace.

La régression logistique est utilisée avec énormément de succès pour apprendre des modèles probabilistes de données. On s'intéresse non seulement au classificateur, mais également à la caractérisation de l'influence des variables x_i qui est mise en évidence par les coefficients de régression correspondents w_i.

11.7 Classification probabiliste

La classification naïve Bayesienne, (*naive Bayes*) est une autre forme de classification, similaire à la régression linéaire, et qui fait appel aux techniques d'inférence probabiliste introduites au chapitre 6. Elle s'applique lorsqu'une structure causale avec une seule cause Y, qui ne peut être observée directement, produit k conséquences X_i observables, de sorte que ces conséquences sont toutes conditionnellement indépendantes étant donné Y. C'est-à-dire lorsque :

$$P(X_i|Y, X_j, j \neq i) = P(X_i|Y)$$

et que les conséquences peuvent être observées. La figure 11.8 en montre un exemple.

Dans ce cas, on sait que $P(Y)$ est proportionnel au produit des probabilités conditionnelles. Nous reprenons ici l'équation 6.2 du chapitre 6 :

$$p(Y|X_1, .., X_k) = \alpha p(Y) \prod_{i=1}^{k} p(X_i|Y)$$

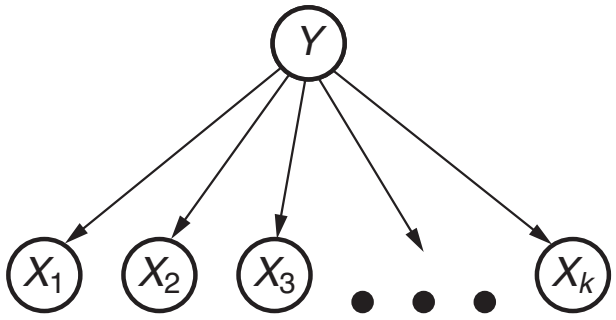


Fig. 11.8 *Réseau causal d'une cause Y avec k conséquences X_i.*

De manière similaire à la régression, elle permet la prédiction de $P(Y|X)$, mais l'apprentissage peut alors se faire par une simple estimation des probabilités conditionnelles $p(X_i|Y)$, par exemple sur la base des fréquences des effets X_i quand la cause Y est également présente.
Par exemple, pour prédire si un e-mail est du spam ou pas (Y), on peut considérer l'ensemble des noms et adjectifs (X_i) du message comme des effets causés par le type du message. Sur la base d'un échantillon de messages déjà triés en spam et non-spam, il suffit d'estimer les probabilités $p(X_i|Y)$, par exemple en calculant la fréquence de chaque mot dans les deux classes. On peut alors estimer la probabilité qu'un message précis soit du spam en appliquant l'équation 6.2 ci-dessus. Pour trier les messages en fonction de cette probabilité, il suffit de les comparer entre elles et on n'a donc pas besoin de connaître $p(Y)$ qui est identique pour chaque message. Cette technique est très largement utilisée dans des filters à spam.

Littérature

La méthode d'induction par spécialisation ou généralisation est décrite en détail dans [59]. Le perceptron a été introduit pour la première fois par Rosenblatt [60], puis développé par de nombreux autres auteurs. Un survol des techniques se trouve dans [61]. Les *support vector machines* ont été introduites par Vapnik [62].

Le livre de Bishop [54] donne des détails sur la régression logistique.

Application : Filtre à spam pour e-mails

L'envoi de messages non sollicités (ce que l'on appelle *spam*) constitue un grave problème pour tout utilisateur d'e-mail d'aujourd'hui. SpamAssassin est un logiciel gratuit qui est très largement utilisé pour le filtrage du spam. Le logiciel est aussi intégré comme noyau dans de nombreux outils commerciaux de gestion d'e-mail.

Comme la plupart des outils anti-spam, SpamAssassin permet d'apprendre la classification spam/non-spam sur la base d'exemples de messages classifiés par l'utilisateur. Certains logiciels se contentent d'apprendre des corrélations statistiques (comme la méthode Bayésienne naïve). Dans sa troisième version, SpamAssassin obtient une meilleure performance en utilisant l'algorithme du perceptron pour apprendre une pondération de différents critères. D'autres études ont montré que la performance pourrait être encore améliorée en utilisant une *support vector machine* ; cependant son implémentation est trop complexe pour un outil qui se veut simple.

(Source : `http://spamassassin.apache.org/`)

11.8 Exercices

Exercice 11.1 Apprentissage avec attributs discrets

Supposons que l'on veuille apprendre à reconnaître les champignons vénéneux sur la base des exemples suivants :

Ordre	chapeau	couleur du chapeau	pied	lamelles	décision
1	convexe	brun	épais	étroites	poison
2	convexe	brun	fuselé	larges	ok
3	en cloche	pourpre	épais	larges	poison
4	convexe	pourpre	épais	étroites	poison
5	en cloche	gris	fuselé	larges	ok
6	en cloche	gris	épais	étroites	poison
7	convexe	gris	fuselé	larges	ok
8	en cloche	pourpre	fuselé	larges	ok

Répondez aux questions suivantes :

1) Donnez une ou plusieurs descriptions conjonctives pour les champignons non-vénéneux. Laquelle de ces descriptions est la meilleure ?
2) Pourquoi est-ce qu'on ne prend pas simplement l'intersection de tous les attributs que partagent les champignons non-vénéneux ?
3) Supposons que l'on rajoute l'exemple suivant :

Ordre	chapeau	couleur du chapeau	pied	lamelle	décision
9	... convexe	gris	fuselé	étroit	poison

Pouvez-vous trouver une description conjonctive de la classe des champignons non vénéneux ? Quel critère permet de répondre à cette question facilement ?

4) Supposons qu'on prenne note également de la couleur de la partie supérieure des champignons, en distinguant les valeurs rouge, brun, blanc et noir.

   Peut-on utiliser ce nouvel attribut pour apprendre une classification conjonctive ? Quel technique permettrait de le faire (pensez aux similarités entre couleurs).

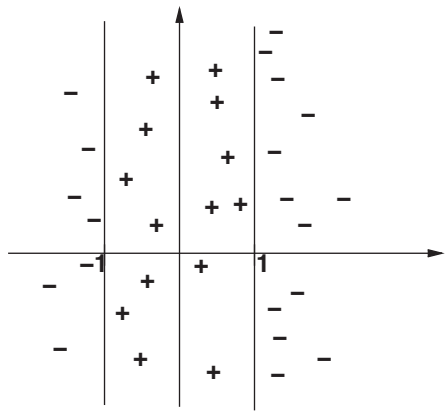
Solution à la page 392

Exercice 11.2 Apprentissage avec attributs numériques

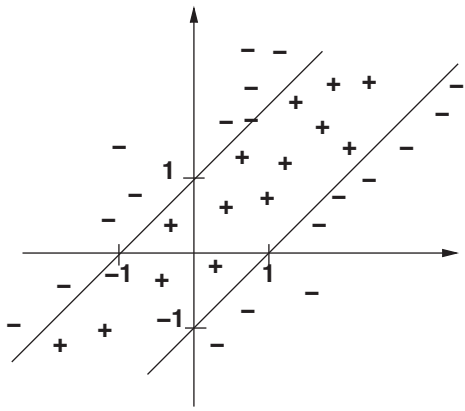
La séparation d'exemples décrits par des attributs numériques exige qu'il soit possible de séparer les exemples positifs et négatifs par une frontière linéaire. Même dans les cas où une frontière linéaire n'existe pas, il est souvent possible d'en trouver une en rajoutant des coordonnées redondantes.

Quelle transformation de coordonnées rendrait ces distributions linéairement séparables ?

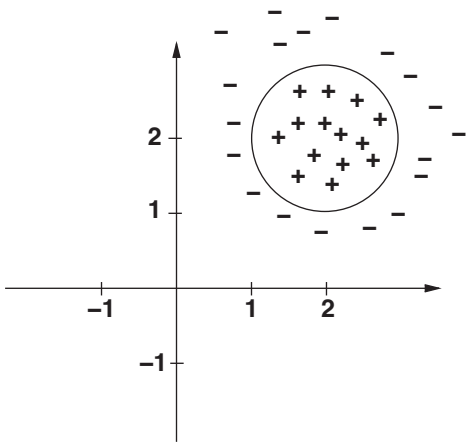
1) Les exemples positifs occupent une bande de largeur 2 autour de l'axe Y (coordonnées X entre entre -1 et 1).



2) Même bande, mais tournée de 45 degrés autour de l'origine.



3) Intérieur du cercle de rayon 1 centré en (2,2).



Solution à la page 392

Chapitre 12

Apprentissage de classifications structurées

Souvent, il est impossible de trouver une seule classification simple qui donne une performance suffisante sur l'ensemble des exemples. Dans le cas où nous avons les exemples positifs $\mathcal{P}$:

```
grand,allongé,rouge,piquant
petit,allongé,rouge,piquant
petit,rond,vert,piquant
```

et les exemples négatifs $\mathcal{N}$:

```
grand,rond,vert,¬piquant
petit,allongé,jaune,¬piquant
petit,rond,rouge,¬piquant
```

il n'existe aucune classification simple sous forme de conjonction d'attributs qui donne le résultat correct sur tous les exemples.

On peut alors appliquer le principe de *diviser pour régner* : en séparant les exemples dans des sous-ensembles, on peut apprendre des classifications simples plus fiables pour chacun de ces sous-ensembles. Lors de la classification de nouveaux exemples, on appliquera le même critère de séparation pour décider quelle classification simple est à appliquer.

Nous considérons trois méthodes pour l'apprentissage d'une description structurée. La première consiste à modifier les méthodes d'apprentissage de classifications logiques simples afin d'apprendre des classifications disjonctives. La deuxième consiste à construire des arbres de décision, et la troisième est le *boosting* qui apprend une combinaison de classifications.

12.1 Apprentissage de classifications disjonctives

Il y a deux manières d'adapter l'algorithme d'apprentissage de classifications logiques pour apprendre des concept disjonctifs :

1) Admettre des hypothèses `h` qui ne soient pas satisfaites par tous les exemples positifs. La description sera alors une disjonction qui, dans son ensemble, couvrira l'ensemble des exemples positifs.
2) Admettre des hypothèses `h` qui ne soient pas satisfaites par certains exemples négatifs. La description sera alors une *liste de décision*, une hiérarchie d'exceptions.

Apprentissage de disjonctions

La figure 12.1 montre un algorithme simple qui implémente la première solution. Il construit successivement des descriptions D qui couvrent une partie X des exemples positifs, tout en excluant l'ensemble des exemples négatifs, et les accumule dans $TERMS$. Chaque fois qu'une description a été trouvée, on peut éliminer les exemples X qui sont couverts par cette description de l'ensemble $PSET$ des exemples. La description finale sera alors une disjonction entre les descriptions de $TERMS$. Pour apprendre les descriptions individuelles, on utilise un apprentissage par spécialisation comme décrit dans le chapitre précédent. Cependant, comme on veut apprendre des descriptions qui ne couvrent pas tous les exemples, on ne peut pas partir de l'ensemble des attributs de tous les exemples positifs, mais il faut commencer avec une description vide.

1: **Function** DISJONCT($\mathcal{P}$,$\mathcal{N}$)
2: PSET $\leftarrow \mathcal{P}$
3: TERMS $\leftarrow$ {}
4: **repeat**
5: D $\leftarrow$ Description qui couvre X $\subseteq$ PSET mais aucun n $\in \mathcal{N}$
6: PSET $\leftarrow$ PSET - X
7: TERMS $\leftarrow$ TERMS $\cup$ D
8: **until** PSET = {}
9: **return** TERMS

Fig. 12.1 *Algorithme pour l'apprentissage de descriptions disjonctives.*

Sur l'exemple donné ci-dessus, l'algorithme trouve la description { (`allongé` $\wedge$ `rouge`) $\vee$ (`petit` $\wedge$ `rond` $\wedge$ `vert`) } par les deux étapes suivantes :

1) X = { { `grand,allongé,rouge` }, { `petit,allongé,rouge` } }
   TERMS = { `allongé` $\wedge$ `rouge` }
   PSET = { `petit,rond,vert` }
2) X = { { `petit,rond,vert` } }
   TERMS = { `(allongé` $\wedge$ `rouge)` $\vee$ `(petit` $\wedge$ `rond` $\wedge$ `vert)` }
   PSET = {}

Le résultat de cet algorithme simple dépend très fortement du choix des sous-ensembles X couverts par chaque partie de la description. Un algorithme plus performant pourrait être obtenu par une recherche entre différents choix pour ces sous-ensembles.

Apprentissage avec exceptions

Une autre manière de traiter les concepts disjonctifs consiste à utiliser une description qui admette des *exceptions*. Ainsi, dans l'exemple cité plus haut, l'exemple { `petit,rond,vert,piquant` } devient une *exception* à la règle !

En général, on peut représenter des exceptions par une *liste de décisions*. Il s'agit d'une liste de règles :

$$\texttt{D = (conj}_1 \Rightarrow \texttt{c}_1\texttt{, conj}_2 \Rightarrow \texttt{c}_2\texttt{, ..., c}_n\texttt{)}$$

où $\texttt{conj}_i$ est une conjonction d'attributs et $\texttt{c}_i$ une classification.

La règle qui donne la classification c_i a la priorité sur c_{i+1} : les règles sont parcourues dans l'ordre et la première règle qui est déclenchée donne la classification. Le dernier élément est une classification par défaut. Par l'ordre des règles, la règle $\texttt{c}_i$ agit comme *exception* à la règle $\texttt{c}_{i+1}$: tous les cas auxquels s'applique $\texttt{c}_i$ ne seront plus considérés par $\texttt{c}_{i+1}$.

La figure 12.2 montre un algorithme pour l'apprentissage automatique d'une telle liste de décision. Il part d'un ensemble $\mathcal{E}$ qui contient tous les exemples (positifs et négatifs). Il construit itérativement des descriptions partielles des exemples de la classe C qui ne sont pas encore correctement classifiés par la $DLIST$ courante. Il se peut que ce processus entre dans un boucle infinie, qui construit toujours la même description. Dans ce cas, on peut sélectionner un attribut A qui divise l'ensemble $MISSED$ des exemples mal classés en deux parties $M1$ et $M2$, et trouver des descriptions pour chacune d'entre elles qui seront alors rajoutées en tête de $DLIST$. Si ces descriptions sont à nouveau identiques à d'autres qui se trouvent déjà dans $DLIST$, on applique le même processus récursivement jusqu'à ce que les descriptions soient différentes.

```
1: Function DL(E)
2: DLIST ← {<classe par défaut>}
3: repeat
4:    MISSED ← p ∈ P ∪ N pas correctement classifié par DLIST
5:    C ← classe la plus commune de MISSED
6:    D ← conjonction qui couvre le plus d'instances de C∈MISSED
7:    if D ≠ first(DLIST) then
8:       DLIST ← cons(D ⇒ C,DLIST)
9:    else
10:      sélectionner un attribut A qui divise MISSED en M1 et M2
11:      récursion sur M1 et M2
12: until MISSED = {}
13: return DLIST
```

Fig. 12.2 *Algorithme pour l'apprentissage d'une liste de décision.*

Dans le cas d'un seul concept, on considère qu'il y a deux classes : membres et non-membres. Dans l'exemple des piments déjà cité plus haut, ces deux classes seront `piquant` et ¬ `piquant` :

a) `grand,allongé,rouge,piquant`
b) `petit,allongé,rouge,piquant`
c) `petit,rond,vert,piquant`

d) grand,rond,vert,¬piquant
e) petit,allongé,jaune,¬piquant
f) petit,rond,rouge,¬piquant

L'algorithme choisira piquant comme classification par défaut et donc comme valeur initiale de $DLIST$; ce choix est arbitraire car piquant et ¬piquant apparaissent avec la même fréquence. L'algorithme procédera alors selon les étapes suivantes :

1) MISSED = (d,e,f)
   D = rond
   DLIST = (rond ⇒ ¬ piquant,piquant)
2) MISSED = (c,e)
   D = petit ∧ rond ∧ vert
   DLIST =
   (petit ∧ rond ∧ vert ⇒ piquant, rond ⇒ ¬ piquant,piquant)
3) MISSED = (e)
   D = petit ∧ jaune ∧ allongé
   DLIST =
   (petit ∧ jaune ∧ allongé ⇒ ¬ piquant, ...)

12.2 Apprentissage d'arbres de décision : l'algorithme ID3

Le deuxième type de description structurée qui peut être appris automatiquement est celui des *arbres de décision* ou, plus précisément, des arbres de classification. Un arbre de classification, par exemple celui de la figure 12.3, est une structure qui permet de déterminer de façon univoque la *classe* (ou le concept) d'un exemple x.

Les différents éléments d'un arbre de classification ont la signification suivante :

- chaque nœud non terminal correspond à un test $P(x)$ qui évalue le prédicat P sur l'exemple x,
- chaque feuille désigne une classe.

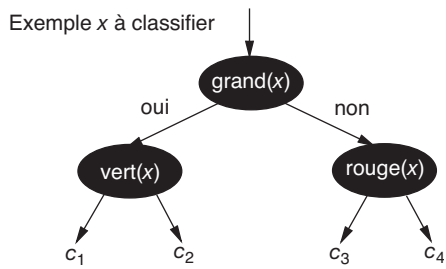


Fig. 12.3 *Exemple d'un arbre de classification.*

L'arbre peut être assimilé à une entrevue avec un expert. Pour classifier un exemple, il faut parcourir l'arbre de la racine aux feuilles, chaque nœud intermédiaire correspondant à une question de l'expert. Chaque réponse du client détermine alors la question suivante de l'expert, soit le chemin à emprunter dans l'arbre : si la réponse est affirmative, il faut prendre le chemin de gauche, sinon celui de droite. Une conversation respectant l'arbre de classification de la figure 12.3 pourrait être :

Expert : Votre exemple est-il grand ?
Vous : Oui.
Expert : Votre exemple est-il vert ?
Vous : Non.
Expert : Alors la classe de votre exemple est c_2.

Plus formellement, l'arbre doit être parcouru selon l'algorithme suivant (`x` est l'exemple à classifier et `N` est la racine de l'arbre) :

1: **Function** CLASSIFY(x,N)
2: **while** N n'est pas terminal **do**
3: **if** $P_N(x)$ **then**
4: N := noeud gauche
5: **else**
6: N := noeud droite
7: **return** la classe de N.

où P_N est le prédicat associé au nœud intermédiaire N.

Il est à remarquer que la description d'une classe c se déduit directement de l'arbre de classification : c'est la conjonction de tous les tests que l'on a effectués lors du parcours de l'arbre, de la racine à la feuille identifiant la classe c. Chaque terme de la description est soit le prédicat P associé au nœud intermédiaire si l'on est parti sur la gauche après le nœud, soit le complément du prédicat $\neg P$ si l'on est allé sur la droite. Ainsi, la description de c_2 de la figure 12.3 est :

$\mathcal{D}_{c_2}$: `grand(x)` $\wedge$ $\neg$`vert(x)`

L'algorithme `ID3` (fig. 12.4) est une méthode pour construire un arbre de classification optimal, c'est-à-dire l'arbre qui permet de classer un exemple en effectuant, en moyenne, un minimum de tests. Pour ce faire, `ID3` construit l'arbre de manière incrémentale en créant à chaque étape une feuille de l'arbre partiel courant :

- Si le parcours de la racine à cette feuille n'est possible que pour les exemples d'une seule classe c_i, elle constitue une feuille de l'arbre final et elle est libellée avec la classe c_i. Les instances pour lesquelles le parcours aboutit à ce nœud seront classifiées ainsi et ne seront bien évidemment plus prises en compte pour la suite de la construction de l'arbre.
- Sinon, la feuille correspond à un nœud intermédiaire de l'arbre final. Un prédicat P_j doit alors être choisi pour définir le test associé au nœud. P_j doit être sélectionné de manière à laisser aussi peu d'incertitude que

```
1: Function ID3(E)
2: if E = {} then
3:    return NIL
4: else
5:    if ∀ e ∈ E classe(e)=c then
6:       return c
7:    else
8:       P ← attribut ∈A qui reduit le plus l'entropie de la classe
9:       L ← {e|e ∈ E et P(e) = succès}
10:      R ← {e|e ∈ E et P(e) = échec }
11:      N ← noeud vide, N.P ← P
12:      N.left ← ID3(L), N.right ← ID3(R)
13:      return N
```

Fig. 12.4 *L'algorithme ID3.*

possible sur la classe de l'exemple après le test. Nous verrons par la suite qu'un critère *d'entropie* permet d'effectuer ce choix. Le prédicat P_j choisi ne pourra bien évidemment plus être utilisé par ses nœuds descendants.

Dans les exemples que nous avons pris ci-dessus, les prédicats sont booléens et ne donnent comme réponse que vrai ou faux, créant ainsi un arbre de décision binaire. On peut généraliser les questions posées dans les nœuds en prenant en compte toutes les valeurs possibles d'un attribut. Par exemple, pour une question sur l'attribut `hauteur ?`, on aurait comme valeurs possibles `grand`, `moyen` ou `petit`. L'algorithme ID3 doit alors construire plusieurs branches à la place des seules branches L et R. Ainsi, un prédicat booléen ne représente que le cas spécial d'un attribut aux deux valeurs, `vrai` ou `faux`.

Choisir de façon intelligente les attributs qui seront testés à chaque nœud de l'arbre constitue un élément clé de ID3. Pour obtenir un arbre compact, il faut qu'on choisisse des attributs pertinents, qui réduisent rapidement l'incertitude restant quant à la classe.

La théorie de l'information fournit une mesure précise de l'incertitude appelée *entropie*. Si l'on applique ce concept à notre problème de classification, l'entropie est une mesure de l'*incertitude* liée à la classification d'un exemple. Cette mesure est importante car elle nous permet de *quantifier* le degré d'optimalité d'un arbre de décision.

Mathématiquement, si un exemple appartient à l'une des n classes $c_1, \ldots, c_n$ et que la probabilité qu'un exemple fasse partie de la classe c_i est $p(c_i)$, l'entropie de la classification $H(\mathcal{C})$ est :

$$H(\mathcal{C}) = -\sum_{i=1}^{n} p(c_i) \cdot \log_2 p(c_i)$$

L'entropie d'une classification après avoir utilisé un test $\mathcal{A}$ (qui normalement correspond à la présence d'un certain attribut) nous permet de décider quel test il faut effectuer : le test retenu doit être celui qui entraîne la plus petite entropie, soit le moins d'incertitude.

Soit $H(\mathcal{C}|\mathcal{A})$ l'entropie de la classification après avoir utilisé $\mathcal{A}$. Afin de calculer la valeur de $H(\mathcal{C}|\mathcal{A})$, il faut tout d'abord déterminer l'entropie de chaque sous-arbre généré par le test $\mathcal{A}$. Dans le cas général, un test $\mathcal{A}$ peut fournir m résultats différents $a_1, \ldots, a_m$. L'entropie du sous-arbre où $\mathcal{A}$ vaut a_j est définie par :

$$H(\mathcal{C}|a_j) = -\sum_{i=1}^{n} p(c_i|a_j) \cdot \log_2 p(c_i|a_j)$$

$p(c_i|a_j)$ étant la probabilité qu'un exemple appartienne à la classe c_i si le test $\mathcal{A}$ fournit a_j.

Pour calculer l'entropie de la classification $H(\mathcal{C}|\mathcal{A})$ après le test $\mathcal{A}$, il suffit de prendre la moyenne pondérée des entropies de tous les sous-arbres du test :

$$H(\mathcal{C}|\mathcal{A}) = \sum_{j=1}^{m} p(a_j) \cdot H(\mathcal{C}|a_j)$$

L'algorithme `ID3` choisit alors à chaque itération le test $\mathcal{A}$ qui fournit la valeur la moins élevée de $H(\mathcal{C}|\mathcal{A})$. En pratique, ce critère revient le plus souvent à diviser les instances restantes de manière aussi égale que possible.

Pour illustrer le fonctionnement de `ID3`, considérons les exemples suivants :

a) `grand,allongé,rouge,piquant`
b) `petit,allongé,rouge,piquant`
c) `petit,rond,vert,piquant`
d) `grand,rond,vert,¬piquant`
e) `petit,allongé,jaune,¬piquant`
f) `petit,rond,rouge,¬piquant`

La figure 12.5 montre comment les différents attributs conduisent à répartir les exemples. La qualité de ces répartitions peut être chiffrée par l'entropie moyenne des ensembles restants. Par exemple, pour l'attribut `rouge`, nous avons les probabilités :

Pr(vrai) = 0.5 (a,b,f), Pr(faux) = 0.5 (c,d,e)

et les probabilités conditionnelles que le piment soit piquant ou non :

Pr(piquant|rouge) = 0.66, Pr(¬ piquant|rouge) = 0.33
Pr(piquant|¬rouge) = 0.33, Pr(¬ piquant|¬rouge) = 0.66

L'incertitude sur la classification qui reste dans le cas où `rouge` est vrai est alors :

$$\begin{aligned} H(C|rouge = vrai) &= -\sum_{v \in \{piq., \neg piq.\}} Pr(v|rouge = vrai) \cdot \log_2(Pr(v|rouge = vrai)) \\ &= -0.66 \ \log_2(0.66) - 0.33 \ \log_2(0.33) \\ &= 0.92 \text{ bit} \end{aligned}$$

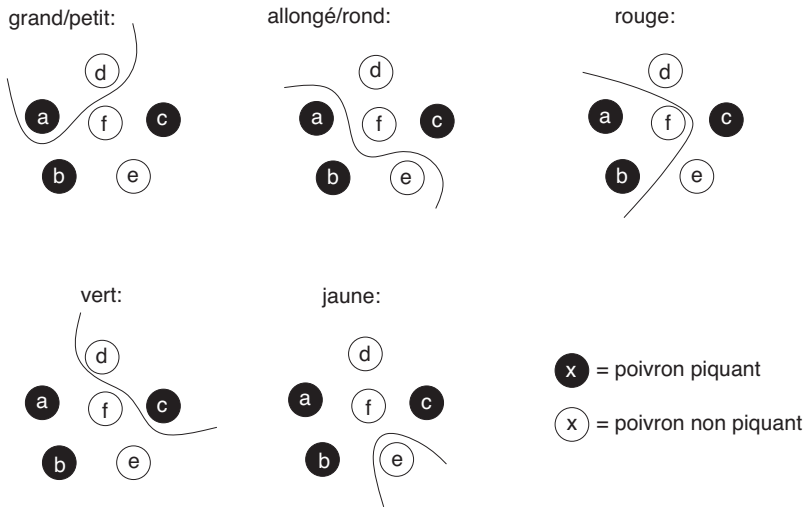


Fig. 12.5 *Répartition des exemples selon les différents attributs.*

et le même résultat sera obtenu pour $H(C|rouge = faux)$. L'entropie moyenne se calcule comme la moyenne pondérée sur les deux valeurs de `rouge`, c'est-à-dire :

$$\begin{aligned} H(C|rouge) &= H(C|rouge = vrai) \cdot Pr(rouge = vrai) + \\ &\quad H(C|rouge = faux) \cdot Pr(rouge = faux) \\ &= 0.5 \cdot 0.92 + 0.5 \cdot 0.92 = 0.92 \text{ bit} \end{aligned}$$

L'arbre est alors construit itérativement en appliquant les mêmes calculs à chaque sous-ensemble. Dans cet exemple, nous avons trois étapes :

1) (a,b,c,d,e,f)
   `grand, petit` ou `vert` : incertitude moyenne restante : 1 bit
   `rond,allongé` ou `rouge` : incertitude moyenne restante : 0.92 bit
   `jaune` : incertitude moyenne restante 0.81 bit
   ⇒ choisir `jaune`
2) (a,b,c,d,f)
   `grand` ou `petit` : incertitude moyenne restante 0.955
   `rond` ou `allongé` : incertitude moyenne restante 0.554
   `rouge` ou `vert` : incertitude moyenne restante 0.955
   ⇒ choisir `rond`
3) (c,d,f)
   `grand` ou `petit` : incertitude moyenne restante 0.66 bit
   `rouge` ou `vert` : incertitude moyenne restante 0.66 bit
   ⇒ choisir `vert`
4) (c,d)
   `grand` ou `petit` : incertitude moyenne restante 0 bit ⇒ `petit`

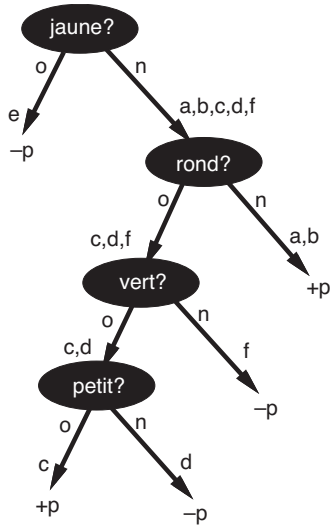


Fig. 12.6 *L'arbre de classification résultant de l'application de* `ID3` *sur l'exemple.*

La figure 12.6 montre l'arbre qui résulte de cette exécution de `ID3`.

Une petite lacune de l'algorithme tel qu'il a été présenté est que le critère d'entropie privilégie des prédicats à beaucoup de valeurs. Cela n'a pas d'importance dans l'exemple que nous avons vu, car tous les attributs ont des valeurs binaires (vrai/faux). Mais ce sera important pour des attributs à valeurs multiples ou même numériques. Il est alors possible de compenser cet effet en divisant la réduction d'incertitude par le nombre de branches générées, c'est-à-dire en choisissant le prédicat A qui maximise :

$$\frac{H(C) - H(C|A)}{|valeurs(A)|}$$

Si l'entropie est le critère le plus habituellement utilisé pour la construction d'un arbre de décision, il en existe d'autres, qui peuvent se révéler utiles. Dans le cas des classifications numériques, on peut notamment minimiser la variance de la classe pour les différents valeurs d'un attribut :

$$1/n \sum_{i=1}^{n} (c_i - \overline{c})^2$$

Une autre possibilité, pour n'importe quel type de classification, consiste à minimiser la fraction d'exemples n'appartenant pas à la classe la plus fréquente :

$$1/n|\{c_i|c_i \neq \text{classe la plus fréquente}\}|$$

`ID3` est un algorithme qui existe depuis longtemps et qui a prouvé sa grande efficacité dans de nombreuses applications pratiques. Cependant, il présente certaines lacunes importantes :

- Certaines décisions peuvent être dues aux hasards de la sélection des exemples, quand ces derniers sont peu nombreux.
- La classification de nouveaux exemples n'est pas toujours adéquate. En fait, il arrive souvent que la performance d'un arbre de classification soit *pire* que la stratégie qui consiste à assigner à tout exemple la classe la plus fréquente.
- Les arbres de classification qui en résultent sont souvent difficiles à interpréter.

Les deux premiers problèmes sont étroitement liés et sont tous les deux une conséquence du phénomène du *surapprentissage* (overfitting). La dernière lacune peut souvent être résolue en remplaçant l'arbre par un jeu de règles équivalent. Après quelques considérations sur la qualité d'un apprentissage en général, nous montrons deux solutions qui sont souvent utilisées aujourd'hui.

Qualité de l'apprentissage

Le critère statistique utilisé par `ID3` peut conduire à choisir des prédicats qui n'ont aucune signification. Par exemple, si la lettre attachée à chaque exemple serait aussi considérée comme un attribut, un prédicat très puissant serait :

indice de l'exemple $\leq$ "c" $\Rightarrow$ `piquant`
indice de l'exemple $\geq$ "d" $\Rightarrow \neg$`piquant`

Comme il est clair, dans cet exemple, qu'il n'existe aucune relation entre l'indice et la classification, on peut s'attendre à ce que l'arbre donne de très mauvais résultats pour de nouveaux exemples. Dans ce cas, il est évident que la numérotation n'est pas significative. Mais comment faire dans le cas où l'on n'a vraiment aucune information quant à la relation entre attributs et classification ?

Il existe en fait une théorie qui permet de chiffrer la qualité qu'on peut espérer d'un résultat d'apprentissage en fonction de sa complexité et du nombre d'exemples fournis en entrée : la théorie PAC. Ces trois lettres sont l'abréviation de :

- **P**robablement :
  la classification est approximativement correcte avec probabilité δ.
- **A**pproximativement :
  la probabilité d'erreur de classification est inférieure à ϵ.
- **C**orrect.

L'idée principale de la théorie est que la situation où un prédicat sans importance a une forte corrélation avec le résultat peut se produire uniquement dans le cas où le nombre d'exemples n'est pas assez élevé. Dans l'exemple de la numérotation, il serait rare dans un grand ensemble d'exemples que tous les exemples positifs viennent avant tous les exemples négatifs.

Le résultat principal de la théorie PAC suppose que l'algorithme rend un résultat correct sur tous les exemples fournis pour l'apprentissage. On fournit

alors N exemples, dont distribution exemples correspond à la réalité, et on cherche à apprendre un concept parmi $|P|$ possibilités. La théorie donne alors une relation entre N, $|P|$, δ et ϵ qui est valable pour n'importe quel algorithme d'apprentissage.
Cette relation est obtenue comme suit. Supposons que $h \in P$ ait un taux d'erreur qui dépasse ϵ. Cela veut dire que la probabilité que h soit néanmoins correct sur les N exemples est

$$Pr(correct(h, N)) < (1 - \epsilon)^N$$

En supposant que l'algorithme d'apprentissage n'a pas de biais parmi les descriptions possibles, la condition PAC est satisfaite si et seulement si la probabilité qu'il *existe* un h qui soit correct sur N exemples, mais a un taux d'erreur $> \epsilon$ sur de nouveaux exemples est $< \delta$:

$$\delta > |P| \cdot (1 - \epsilon)^N$$

Cette relation permet alors d'obtenir une limite sur N nécessaire pour garantir une borne δ_0 sur δ :

$$N \geq \frac{\log(\delta_0/|P|)}{\log(1 - \epsilon)}$$

Donc, par exemple, si le nombre de descriptions possibles $|P|$ croît de façon simplement exponentielle avec le nombre d'attributs, alors le nombre d'exemples requis croît de façon linéaire avec la complexité de la description. Cela est le cas notamment des descriptions conjonctives que nous avons vues précédemment. Ce sont donc des descriptions qui peuvent être apprises de manière efficace. Par contre, pour les arbres de classification, la croissance du nombre d'arbres possibles par rapport au nombre d'attributs est exponentielle par rapport au carré du nombre d'exemples, et donc le nombre d'exemples requis augment bien plus vite.

Par exemple, pour un domaine avec dix attributs, il y a au plus :

$$|P| = 11 \cdot 10^2 \cdot 9^4 \cdot 8^8 ... \cdot 2^{2^9} = 2,6579 \cdot 10^{35}$$

arbres différents possibles[(1)]. Donc, si on veut en plus atteindre une erreur de classification $\epsilon < 0.001$ avec une probabilité $\delta < 0.01$, la formule nous donne, pour le nombre d'exemples requis :

$$N \geq \frac{ln(\delta/|P|)}{ln(1 - \epsilon)} = 86\,131$$

ce qui est déjà un nombre très important pour un domaine aussi simple. Si, par contre, nous nous contentons d'une erreur de classification $\epsilon \leq 0.05$, il ne faudra plus que 1680 exemples.

En général, comme 2^{2^9} est plus grand que chacun des 10 autres facteurs de la somme, c'est ce dernier élément qui est déterminant pour la complexité (qui est donc doublement exponentielle par rapport au nombre d'attributs).

(1) Cela se calcule comme suit : on a le choix entre 10 attributs ou rien du tout pour le premier nœud, 9 attributs ou rien = 10 possibilités pour chacun des deux nœuds du 2^{e} niveau, 9 possibilités pour chacun des 4 nœuds du 3^{e} niveau, et ainsi de suite.

Élaguer un arbre de décision

Dans le contexte des arbres de classification, le problème de l'overfitting apparaît surtout près des feuilles de l'arbre, quand la taille des sous-ensembles utilisés pour construire les dernières parties de l'arbre devient trop peu importante pour assurer la qualité de la classification. Dans ce cas, on peut souvent obtenir de meilleurs résultats en assignant à tous les exemples la classe la plus probable de l'ensemble. Dans le cas idéal, on arrêterait la procédure `ID3` au moment où le nombre d'exemples restants ne suffit plus pour garantir une bonne qualité de classification. Cependant, il est difficile de faire cela car on ne connaît pas encore les performances de l'arbre qui en résulteraient. On procède donc plutôt par *élagage* d'un arbre une fois qu'il est complètement construit.

L'idée de l'élagage est de comparer la performance de l'arbre élagué avec celle de l'arbre original sur de nouveaux exemples. Cependant, comment peut-on effectuer cette comparaison sans disposer de nouveaux exemples ? On peut utiliser l'estimation suivante, qui provient de la statistique. Supposons que élaguer un nœud n implique E erreurs sur les N exemples du jeu original qui sont attribués à ce nœud. Supposons de plus que les erreurs pour chaque exemple sont statistiquement indépendantes :

$$Pr(\text{E erreurs}) = Pr(\text{erreur})^E \cdot (1 - Pr(\text{erreur}))^{(N-E)}$$

La statistique permet alors de donner des bornes sur $Pr(\text{erreur})$ qui assurent un certain degré de confiance. Par exemple, si un arbre de classification fait zéro erreur sur six exemples, la probabilité d'erreur avec un degré de confiance $> 25\%$ est 0.206. On peut ensuite comparer cela avec le nombre d'erreurs qui apparaissent si on remplace l'arbre qui classifie ces six exemples par une classification uniforme. Si en fait la classification uniforme est meilleure, on coupe l'arbre à partir de ce nœud.

Arbre ⇒ Règles

Les décisions de l'arbre sont toujours liées à une séquence de tests. Comme chaque nœud de l'arbre dépend alors de tous les nœuds qui le précèdent, on ne peut pas l'interpréter en soi. Il serait plus pratique de disposer de règles qui soient valables indépendamment du contexte, par exemple :

`jaune` ⇒ ¬ `piquant`
¬ `jaune` ∧ ¬ `rond` ⇒ `piquant`
¬ `jaune` ∧ `rond` ∧ ¬ `vert` ⇒ ¬ `piquant`
....

Un premier jeu de règles peut être généré facilement en observant que chaque parcours de l'arbre génère une règle indépendante. Pour l'arbre de la figure 12.6, nous avons les parcours et donc les règles suivantes :

1. `jaune` ⇒ ¬ `piquant`
2. ¬ `jaune` ∧ ¬ `rond` ⇒ `piquant`
3. ¬ `jaune` ∧ `rond` ∧ ¬ `vert` ⇒ ¬ `piquant`
4. ¬ `jaune` ∧ `rond` ∧ `vert` ∧ `petit` ⇒ `piquant`

5. ¬ `jaune` ∧ `rond` ∧ `vert` ∧ ¬ `petit` ⇒
¬ `piquant`

Cependant, ces règles sont beaucoup trop compliquées, car certaines des conditions ne sont pas nécessaires. On peut alors les simplifier comme suit :

3'. `rond` ∧ ¬ `vert` ⇒ ¬ `piquant`
4'. `rond` ∧ `vert` ∧ `petit` ⇒ `piquant`
4". `vert` ∧ `petit` ⇒ `piquant`
5'. `rond` ∧ `vert` ∧ ¬ `petit` ⇒ ¬ `piquant`
5". `vert` ∧ ¬ `petit` ⇒ ¬ `piquant`

Les simplifications peuvent être trouvées automatiquement en traitant une règle à la fois. On écarte itérativement un prédicat à la fois et on examine si la règle reste valide sur l'ensemble des exemples. S'il n'y a que très peu d'exemples qui ne satisfont pas la règle modifiée, on peut parfois tolérer ces exceptions avec une justification similaire à celle du élagage de l'arbre.

L'apprentissage inductif dont sont capables certains types de réseaux de neurones possède des caractéristiques semblables à `ID3`. Cependant, comme l'ensemble des neurones est fixe, la profondeur de la classification n'est pas adaptée automatiquement à la précision requise pour classer tous les exemples. Ainsi, dans certains cas, la classification construite par de tels réseaux est surdéterminée et permet donc également de compléter des informations manquantes. Par contre, si le nombre de neurones s'avère insuffisant, le réseau n'apprendra jamais une classification suffisamment précise pour classer correctement tous les exemples.

12.3 Bagging et boosting : combinaison de différentes techniques d'apprentissage

La troisième méthode pour l'apprentissage de classifications structurées consiste à combiner différents classificateurs par les méthodes du *bagging* et *boosting*. Il s'agit de techniques très générales qui permettent d'apprendre de façon ciblée des classifications simples qui peuvent s'intégrer dans une seule classification structurée.

Le *bagging* et le *boosting* font appel à des méthodes dites *méthodes d'apprentissage faibles*, qui apprennent des classifications simples, et dont on exige que le taux d'erreur soit inférieur à 50%, étant donné une certaine distribution de la probabilité des exemples. De telles méthodes ne sont pas difficiles à construire, puisque la plupart des méthodes d'apprentissage d'une description simple remplissent ce critère.

La méthode d'apprentissage simple prend ainsi comme entrée

- n exemples $(\underline{x}_i, c_i), i \in 1..n\,;\, c_i \in \{0,1\}$,
- une distribution de probabilités $p_i, \sum_{i=1}^{n} p_i = 1$,

et doit fournir une classification simple $h(\underline{x})$ avec une probabilité d'erreur inférieure à 50% :

$$\epsilon = \sum_{i=1}^{n} p_i |h(\underline{x}_i) - c_i| < 0.5$$

Le *bagging* consiste à appliquer un certain nombre n de méthodes d'apprentissage faibles, choisies de façon aléatoire et à réaliser un vote parmi les classificateurs : si au moins k sur n classificateurs donnent un résultat positif, on conclut à une classification positive, autrement elle reste négative.

Dans le cas du *boosting*, on choisit les méthodes d'apprentissage faibles de façon plus ciblée. L'algorithme ADABOOST construit k classifications simples en choisissant chaque fois une distribution de probabilité des exemples qui mette l'accent sur les exemples mal classés par les classifications déjà apprises.

Initialement, tous les n exemples auront le même poids $w_i = 1/n$. L'algorithme est une itération pour $t \in 1..k$ des opérations suivantes :

for $i \in 1..n$ **do**

$p_i \leftarrow \frac{w_i}{\sum_{i=1}^{n} w_i}$

$h \leftarrow$ résultat de la méthode faible, où les fréquences des examples sont données par les p_i

erreur $\epsilon \leftarrow \sum_{i=1}^{n} p_i |h(\underline{x}_i) - c_i|$

$\beta_t \leftarrow \frac{\epsilon}{(1-\epsilon)}$

for $i \in 1..n$ **do**

$w_i \leftarrow w_i \beta^{1-|h(\underline{x}_i)-c_i|}$

La première étape consiste à calculer les probabilités des exemples en normalisant la distribution des poids. On applique ensuite la méthode faible et on mesure le taux d'erreur ϵ sur les exemples avec leur distribution. Le facteur β qui sera associé à cette classification est calculé en fonction de ce taux d'erreur. Ensuite, les poids w_i sont mis à jour de façon à réduire le poids des exemples qui sont correctement classés. Lors de l'itération suivante, l'algorithme se focalisera donc sur les exemples pour lesquels la classification n'est pas encore correcte.

Le classificateur final consiste en une classification structurée, qui combine les différents classificateurs multipliés par leur poids :

$$h_f(\underline{x}) = \begin{cases} 1 & si \sum_{j=1}^{k} (-\log \beta_j)(h_i(\underline{x}) - 1/2) \geq 0 \\ 0 & sinon \end{cases}$$

On peut garantir que l'on obtient par cette méthode une classification de plus en plus fiable.

Considérons le *boosting* sur le problème de classification de piments suivant :

1) `grand,allongé,rouge,piquant`
2) `petit,allongé,rouge,piquant`
3) `petit,rond,vert,piquant`

4) `grand,rond,jaune,¬piquant`
5) `petit,allongé,jaune,¬piquant`
6) `grand,rond,rouge,¬piquant`

Supposons que nous utilisons un seul attribut qui prédit `piquant` ou `¬piquant` avec une fiabilité d'au moins 50% et que nous apprenons ainsi 3 classifications faibles :

1) h_1 : `allongé` $\Rightarrow$ `piquant`

erreurs	ϵ_1	β_1	w_1	w_2	w_3	w_4	w_5	w_6
3,5	1/3	1/2	1/12	1/12	1/6	1/12	1/6	1/12

$\Rightarrow P = (1/8, 1/8, 1/4, 1/8, 1/4, 1/8)$

2) h_2 : `jaune` $\Rightarrow$ `¬ piquant`

erreurs	ϵ_2	β_2	w_1	w_2	w_3	w_4	w_5	w_6
6	1/8	1/7	1/84	1/84	1/42	1/84	1/42	1/12

$\Rightarrow P = (1/14, 1/14, 1/7, 1/14, 1/7, 1/2)$

3) h_3 : `petit` $\Rightarrow$ `piquant`

erreurs	ϵ_3	β_3	w_1	w_2	w_3	w_4	w_5	w_6
5	1/7	1/6	1/84	1/84	1/42	1/84	1/42	1/12

Le résultat est donc la classification :

$$\begin{aligned} & h_f(\underline{x}) \\ = \; & \log(2)h_1(\underline{x}) + \log(7)h_2(\underline{x}) + \log(6)h_3(\underline{x}) - \frac{\log(2)+\log(7)+\log(6)}{2} \\ = \; & 0.69h_1(\underline{x}) + 1.95h_2(\underline{x}) + 1.8h_3(\underline{x}) - 2.22 \end{aligned}$$

Considérons la performance de la classification structurée sur les exemples :

Exemple	h_1	h_2	h_3	h_f
1) `grand,allongé,rouge,piquant`	1	1	0	$0.42 \rightarrow 1$
2) `petit,allongé,rouge,piquant`	1	1	1	$2.22 \rightarrow 1$
3) `petit,rond,vert,piquant`	0	1	1	$1.53 \rightarrow 1$
4) `grand,rond,jaune,¬piquant`	0	0	0	$-2.22 \rightarrow 0$
5) `petit,allongé,jaune,¬piquant`	1	0	1	$0.27 \rightarrow 1$
6) `grand,rond,rouge,¬piquant`	0	1	0	$-0.27 \rightarrow 0$

Même si dans ce cas, il reste toujours une erreur de classification, le *boosting* est une méthode très utilisée en pratique.

Une autre variante du *boosting*, appelée *Martingale boosting*, est apparue récemment. Dans cette méthode, on considère les classificateurs faibles dans un ordre et choisit le prochain classificateur en fonction du résultat des classifications précedentes, et plus précisement sur la base du nombre de classifications positives obtenues par les classificateurs précédents. Lors de l'apprentissage également, on apprend les classificateurs sur les sous-ensembles d'exemples qui ont obtenu un certain nombre de classifications positives par les classificateurs précédents. Cette méthode est dans un certain sens une combinaison entre un arbre de classification et le *boosting*, mais la croissance du nombre de nœuds

est plus faible que pour les arbres de décision. On peut prouver que si chaque classificateur correspond au critère d'une méthode d'apprentissage faible, le résultat convergera rapidement vers une classification fiable des exemples.

La figure 12.7 montre schématiquement une séquence de classificateurs qui pourrait être apprise par *Martingale boosting* sur notre exemple. Un exemple commence son parcours en haut à gauche. Chaque fois qu'un classificateur rend un résultat positif, on continue vers la droite, autrement vers la gauche. Les flèches en traits pointillés correspondent à un test nul qui échoue toujours (comme la classification est déjà parfaite).

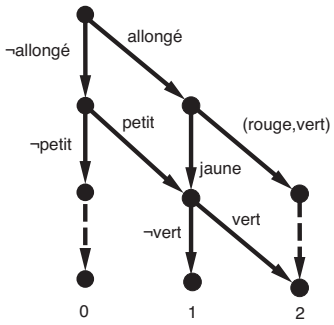


Fig. 12.7 *Séquence de classificateurs obtenue par Martingale boosting pour l'exemple des piments.*

Les exemples seront alors classés comme suit (notons que les exemples ne suivent pas tous les mêmes classificateurs) :

Exemple	h_1	h_2	h_3	*score*
1) `grand,allongé,rouge,piquant`	1	1	-	2
2) `petit,allongé,rouge,piquant`	1	1	-	2
3) `petit,rond,vert,piquant`	0	1	1	2
4) `grand,rond,jaune,¬piquant`	0	0	-	0
5) `petit,allongé,jaune,¬piquant`	1	0	0	1
6) `grand,rond,rouge,¬piquant`	0	0	-	0

En classifiant tous les piments avec au moins deux votes positifs comme piquants, on obtient donc un résultat correct.

Littérature

L'algorithme ID3 a été présenté dans [64]. La théorie PAC a été publiée dans [65]. Le *boosting* a été introduit à l'origine dans [66], puis amélioré dans [67]. Le *Martingale boosting* présenté dans ce chapitre a été publié pour la première fois dans [68].

Application : Prédiction de pannes de réseaux électriques

Le réseau de distribution d'électricité de la ville de New York contient de nombreux câbles très vieux, qui risquent de causer des courts-circuits lors de périodes de fortes demande. Pour maintenir la stabilité du réseau, il est important de remplacer ces câbles avant qu'ils ne lâchent sous la tension. La compagnie Consolidated Edison, fournisseuse d'électricité pour la plus grande partie de la ville, possède une procédure pour tester les câbles et détecter s'il faut les remplacer. Mais souvent c'est l'application même du test qui rend les câbles défectueux. Cette procédure a donc un coût important.

En 2005, la société a introduit un système d'apprentissage basé sur le *boosting* qui détecte les câbles les plus à risque sur la base de 150 attributs comme leur âge, leur type, leur charge habituelle et leur comportement en fonction de la charge. Le système utilise une version de *boosting* qui s'appelle *Martingale boosting* pour apprendre à établir un classement des câbles dont le risque de panne est le plus élevé. Tous les câbles qui sont effectivement tombés en panne se trouvaient dans la moitié que le système avait classée comme la plus à risque. De plus, 75% des pannes concernaient 25% des câbles classés comme les plus à risque.

En focalisant l'attention sur les câbles les plus à risque, le système a fait économiser des dizaines de millions de dollars de coûts à la société ConEdison. Une amélioration supplémentaire de la performance est attendue, ainsi qu'une utilisation plus grande de cette technique.

(Source : Philip Gross *et al.* : Predicting Electricity Distribution Feeder Failures Using Machine Learning Susceptibility Analysis, *IAAI-06*, pp. 1705-1711, 2006.)

12.4 Exercices

Exercice 12.1 Les arbres de décision (ID3)

Si vous voulez investir dans une compagnie informatique et que vous demandez conseil à un expert financier, avant de vous répondre, celui-ci vous posera toute une série de questions concernant l'entreprise. Il voudra connaître le type de concurrence à laquelle elle est confrontée, son âge, son secteur d'activité, etc. En admettant que vous possédiez de nombreux exemples de profils d'entreprise accompagnés des conclusions de l'expert, vous auriez en quelque sorte à votre disposition une partie de son expertise. Il serait intéressant de pouvoir la réutiliser sans avoir toujours recours à lui lorsque vous souhaitez analyser de nouvelles entreprises.

Un arbre de décision est une structure qui est souvent utilisée pour représenter des connaissances. Il permet justement de remplacer un expert humain lorsque l'on désire connaître la nature d'une certaine caractéristique d'un objet, caractéristique que nous appellerons la 'classe' de cet objet. Il s'agit d'une

structure en arbre qui modélise le cheminement intellectuel de l'expert et dans laquelle :

- Chaque nœud intermédiaire correspond à une question portant sur une propriété de l'objet. Nous appelons une telle propriété un « attribut ».
- Chaque arête correspond à une valeur de cet attribut.
- Chaque nœud terminal correspond à une collection d'objets appartenant à la même classe. Cette classe est donc associée au nœud. La même classe peut se manifester dans plusieurs nœuds terminaux.

En parcourant cet arbre, c'est-à-dire en répondant aux questions des nœuds intermédiaires et en suivant les arêtes correspondantes, on parvient à un nœud terminal qui nous renseigne sur la classe de l'objet.

Modules squelettes

Voici tout d'abord les squelettes de fichiers Python qui vous permettront de réaliser l'exercice. Les deux modules `exemple_profits.py` et `exemple_maladies.py` vous permettront de tester votre programme :

Module `moteur_id3/noeud_de_decision.py` :

```
class NoeudDeDecision:
    def __init__(self, attribut, donnees, enfants=None):
        self.attribut = attribut
        self.donnees = donnees
        self.enfants = enfants

    def terminal(self):
        return self.enfants is None

    def classe(self):
        if self.terminal():
            return self.donnees[0][0]

    def classifie(self, donnee):
        rep = ''
        if self.terminal():
            rep += 'Alors {}'.format(self.classe().upper())
        else:
            valeur = donnee[self.attribut]
            enfant = self.enfants[valeur]
            rep += 'Si {} = {}, '.format(self.attribut, valeur.upper())
            rep += enfant.classifie(donnee)
        return rep

    def repr_arbre(self, level=0):
        rep = ''
        if self.terminal():
            rep += '---'*level
            rep += 'Alors {}\n'.format(self.classe().upper())
            rep += '---'*level
            rep += 'Décision basée sur les données:\n'
            for donnee in self.donnees:
```

```
                rep += '---'*level
                rep += str(donnee) + '\n'

        else:
            for valeur, enfant in self.enfants.items():
                rep += '---'*level
                rep += 'Si {} = {}: \n'.format(self.attribut, valeur.upper())
                rep += enfant.repr_arbre(level+1)

        return rep

    def __repr__(self):
        return str(self.repr_arbre(level=0))
```

Module `moteur_id3/id3.py` :

```
from math import log
from .noeud_de_decision import NoeudDeDecision

class ID3:
    def construit_arbre(self, donnees):
        # Nous devons extraire les domaines de valeur des
        # attributs, puisqu'ils sont nécessaires pour
        # construire l'arbre.
        attributs = {}
        for donnee in donnees:
            for attribut, valeur in donnee[1].items():
                valeurs = attributs.get(attribut)
                if valeurs is None:
                    valeurs = set()
                    attributs[attribut] = valeurs
                valeurs.add(valeur)

        arbre = self.construit_arbre_recur(donnees, attributs)

        return arbre

    def construit_arbre_recur(self, donnees, attributs):
        print('à compléter')

    def partitionne(self, donnees, attribut, valeurs):
        print('à compléter')

    def p_aj(self, donnees, attribut, valeur):
        print('à compléter')

    def p_ci_aj(self, donnees, attribut, valeur, classe):
        print('à compléter')

    def h_C_aj(self, donnees, attribut, valeur):
        print('à compléter')

    def h_C_A(self, donnees, attribut, valeurs):
        print('à compléter')
```

Module `exemple_profits.py` :

```
from moteur_id3.noeud_de_decision import NoeudDeDecision
from moteur_id3.id3 import ID3


# Les données d'apprentissage.
donnees = [
    ['down', {
        'age': 'old',
        'competition': 'no',
        'type': 'software'
    }],
    ['down', {
        'age': 'midlife',
        'competition': 'yes',
        'type': 'software'
    }],
    ['up', {
        'age': 'midlife',
        'competition': 'no',
        'type': 'hardware'
    }],
    ['down', {
        'age': 'old',
        'competition': 'no',
        'type': 'hardware'
    }],
    ['up', {
        'age': 'new',
        'competition': 'no',
        'type': 'hardware'
    }],
    ['up', {
        'age': 'new',
        'competition': 'no',
        'type': 'software'
    }],
    ['up', {
        'age': 'midlife',
        'competition': 'no',
        'type': 'software'
    }],
    ['up', {
        'age': 'new',
        'competition': 'yes',
        'type': 'software'
    }],
    ['down', {
        'age': 'midlife',
        'competition': 'yes',
        'type': 'hardware'
    }],
    ['down', {
        'age': 'old',
        'competition': 'yes',
        'type': 'hardware'
    }],
]
```

```
id3 = ID3()
arbre = id3.construit_arbre(donnees)
print('Arbre de décision :')
print(arbre)
print()

print('Exemplification :')
print(arbre.classifie ({
        'age': 'midlife',
        'competition': 'no',
        'type': 'hardware'}))
```

Module `exemple_maladies.py` :

```
from moteur_id3.noeud_de_decision import NoeudDeDecision
from moteur_id3.id3 import ID3

# Les données d'apprentissage.
donnees = [
    ['angine-érythémateuse', {
        'fièvre': 'élevée',
        'amygdales': 'gonflées',
        'ganglions': 'oui',
        'gêne-à-avaler': 'oui',
        'mal-au-ventre': 'non',
        'toux': 'non',
        'rhume': 'non',
        'respiration': 'normale',
        'joues': 'normales',
        'yeux': 'normaux'}
    ],
    ['angine-pultacée', {
        'fièvre': 'élevée',
        'amygdales': 'points-blancs',
        'ganglions': 'oui',
        'gêne-à-avaler': 'oui',
        'mal-au-ventre': 'non',
        'toux': 'non',
        'rhume': 'non',
        'respiration': 'normale',
        'joues': 'normales',
        'yeux': 'normaux'}
    ],
    ['angine-diphtérique', {
        'fièvre': 'légère',
        'amygdales': 'enduit-blanc',
        'ganglions': 'oui',
        'gêne-à-avaler': 'oui',
        'mal-au-ventre': 'non',
        'toux': 'non',
        'rhume': 'non',
        'respiration': 'normale',
        'joues': 'normales',
        'yeux': 'normaux'}
    ],
    ['appendicite', {
        'fièvre': 'légère',
```

```
        'amygdales': 'normales',
        'ganglions': 'non',
        'gêne-à-avaler': 'non',
        'mal-au-ventre': 'oui',
        'toux': 'non',
        'rhume': 'non',
        'respiration': 'normale',
        'joues': 'normales',
        'yeux': 'normaux'}
    ],
    ['bronchite', {
        'fièvre': 'légère',
        'amygdales': 'normales',
        'ganglions': 'oui',
        'gêne-à-avaler': 'non',
        'mal-au-ventre': 'non',
        'toux': 'oui',
        'rhume': 'oui',
        'respiration': 'gênée',
        'joues': 'normales',
        'yeux': 'normaux'}
    ],
    ['coqueluche', {
        'fièvre': 'légère',
        'amygdales': 'normales',
        'ganglions': 'non',
        'gêne-à-avaler': 'oui',
        'mal-au-ventre': 'non',
        'toux': 'oui',
        'rhume': 'oui',
        'respiration': 'gênée',
        'joues': 'normales',
        'yeux': 'normaux'}
    ],
    ['pneumonie', {
        'fièvre': 'élevée',
        'amygdales': 'normales',
        'ganglions': 'non',
        'gêne-à-avaler': 'non',
        'mal-au-ventre': 'non',
        'toux': 'oui',
        'rhume': 'non',
        'respiration': 'rapide',
        'joues': 'rouges',
        'yeux': 'normaux'}
    ],
    ['rougeole', {
        'fièvre': 'légère',
        'amygdales': 'normales',
        'ganglions': 'non',
        'gêne-à-avaler': 'oui',
        'mal-au-ventre': 'non',
        'toux': 'oui',
        'rhume': 'oui',
        'respiration': 'normale',
        'joues': 'normales',
        'yeux': 'larmoyants'}
    ],
    ['rougeole', {
```

```
        'fièvre': 'légère',
        'amygdales': 'normales',
        'ganglions': 'non',
        'gêne-à-avaler': 'oui',
        'mal-au-ventre': 'non',
        'toux': 'oui',
        'rhume': 'oui',
        'respiration': 'normale',
        'joues': 'taches-rouges',
        'yeux': 'larmoyants'}
    ],
    ['rubéole', {
        'fièvre': 'légère',
        'amygdales': 'normales',
        'ganglions': 'oui',
        'gêne-à-avaler': 'non',
        'mal-au-ventre': 'non',
        'toux': 'non',
        'rhume': 'non',
        'respiration': 'normale',
        'joues': 'taches-rouges',
        'yeux': 'normaux'}
    ],
    ['rubéole', {
        'fièvre': 'non',
        'amygdales': 'normales',
        'ganglions': 'oui',
        'gêne-à-avaler': 'non',
        'mal-au-ventre': 'non',
        'toux': 'non',
        'rhume': 'non',
        'respiration': 'normale',
        'joues': 'taches-rouges',
        'yeux': 'normaux'}
    ],
    ['rubéole', {
        'fièvre': 'non',
        'amygdales': 'normales',
        'ganglions': 'oui',
        'gêne-à-avaler': 'non',
        'mal-au-ventre': 'non',
        'toux': 'non',
        'rhume': 'non',
        'respiration': 'normale',
        'joues': 'normales',
        'yeux': 'normaux'}
    ],
]


id3 = ID3()
arbre = id3.construit_arbre(donnees)
print('Arbre de décision :')
print(arbre)
print()

print('Exemplification :')
print(arbre. classifie ({
        'fièvre': 'non',
```

```
'amygdales': 'normales',
'ganglions': 'oui',
'gêne-à-avaler': 'non',
'mal-au-ventre': 'non',
'toux': 'non',
'rhume': 'non',
'respiration': 'normale',
'joues': 'normales',
'yeux': 'normaux'}))
```

L'algorithme ID3

ID3 est un algorithme de construction d'arbres de décision qui vise à minimiser le nombre de questions à poser. Il construit un arbre de décision à partir d'un ensemble de données constituées d'objets décrits par leurs attributs et leur classe. L'algorithme est le suivant :

```
ID3(données, attributs)
1.  IF données est vide THEN
2.      RETURN NULL
3.  ELSE IF toutes les données font partie de la même classe THEN
4.      # Nœud terminal
5.      RETURN un nœud terminal contenant tous les données
6.  ELSE
7.      # Nœud intermédiaire
8.      A <- l'attribut minimisant l'entropie de la classification
9.      valeurs <- liste des valeurs possibles pour A
10.     FOR v IN valeurs DO
11.         # Partitionnement:
12.         partitions[v] <- les données qui ont v comme valeur pour A
14.         # Calcul des sous-nœuds
15.         enfants[v] <- ID3(partitions[v], attributs - A)
16.     END FOR
17.     RETURN un nœud avec enfants comme successeurs
18. END IF
END ID3
```

Bien évidemment, la qualité de l'arbre de décision construit par ID3 dépend des données ; plus elles sont variées et nombreuses, plus la classification de nouveaux objets sera fiable.

Structures de données

Voyons les structures de données dont nous aurons besoin. Nous représenterons les donnée d'apprentissage (un objet avec sa classe) sous forme de listes composées du nom de la classe et d'un dictionnaire {attribut : valeur} :

```
donnée ::= [val-classe,
            {attribut-1: val-attribut-1,
             ... ,
            attribut-k: val-attribut-k}]
```

où `k` est le nombre d'attributs. Chaque donnée doit spécifier une valeur pour chaque attribut. Vous pouvez trouver des exemples de telles données d'apprentissage dans les modules de test `exemples_maladies.py` et `exemples_profits.py`.

Nous utiliserons aussi :

- un dictionnaire qui fait correspondre chaque attribut à son domaine de valeurs :

```
attributs ::= {attribut−1: [val−attribut−1−1, ...],
               ...,
               attribut−k: [val−attribut−1−k, ...]}
```

- une liste contenant toutes les classes des données d'apprentissage :

```
classes ::= [val−classe−1, ...]
```

La classe `NoeudDeDecision`

Les nœuds de l'arbre de décision seront modélisés par la classe `NoeudDeDecision`. La classe contient trois champs :

- `attribut` : l'attribut de partitionnement d'un nœud. Ce champ vaut `None` pour un nœud terminal.
- `donnees` : la liste des données qui tombent dans la sous-classification du nœud.
- `enfants` : un dictionnaire associant un fils (sous-nœud) à chaque valeur de l'attribut du nœud. Ce champ vaut `None` pour un nœud terminal.

Exercice 12.1.1 L'entropie

La classe `ID3` du module `id3.py` implémente l'algorithme ci-dessus. Elle contient une méthode qui construit un arbre de décision à partir des données d'apprentissage. Cette méthode s'appuie à son tour sur une méthode utilitaire qui calcule l'entropie conditionnelle de la classe étant donné un attribut qui partitionne les données.

L'entropie est une mesure de l'information, ou plutôt de l'incertitude, à l'égard de la classification d'un objet. ID3 utilise cette mesure comme une heuristique visant à minimiser la taille de l'arbre de décision, ne conservant à chaque étape que l'information absolument nécessaire pour classer un objet. Chaque fois que l'on doit choisir un attribut pour partitionner les données, on privilégie ainsi celui qui génère une classification dont l'entropie est minimale.

Nous notons $H(C|A)$ l'entropie de la classification après avoir partitionné les données selon la valeur de l'attribut A. Sa valeur est donnée par l'équation :

$$H(C|A) = \sum_{j=1}^{M} p(a_j) H(C|a_j)$$

où a_j est une valeur de l'attribut A, M est le nombre total de valeurs possibles de A et $p(a_j)$ est la probabilité que la valeur de l'attribut A soit a_j.

$H(C|a_j)$ est l'entropie de la classification parmi les données pour lesquelles l'attribut A prend la valeur a_j. Elle est définie par l'égalité :

$$H(C|a_j) = -\sum_{i=1}^{N} p(c_i|a_j) \log_2 p(c_i|a_j)$$

où N est le nombre de classes différentes, et $p(c_i|a_j)$ la probabilité conditionnelle qu'un objet appartienne à la classe c_i sachant que son attribut A vaut a_j.

Dans la classe `ID3`, écrivez donc une méthode `p_aj`, avec quatre arguments : `self`, `donnees`, `attribut` et `valeur`. Cette méthode doit retourner la probabilité `p(attribut=valeur)`, c'est-à-dire la probabilité $p(a_j)$, sur la base de `donnees`, que l'attribut `attribut` vaille `valeur`.

De façon similaire, écrivez une deuxième méthode `p_ci_aj` qui prenne un argument de plus : `classe`. Cette méthode doit retourner la probabilité conditionnelle `p(classe=classe|attribut=valeur)`, c'est-à-dire la probabilité $p(c_i|a_j)$ qu'une donnée appartienne à la classe `classe` lorsque son attribut `attribut` vaut `valeur`. Cette probabilité devra être calculée par rapport aux objets de `donnees`.

Ensuite, écrivez une méthode `h_C_aj`, avec quatre arguments : `self`, `donnees`, `attribut` et `valeur`, qui retourne l'entropie de la sous-classification $H(C|a_j)$, où a_j est la valeur `valeur` de l'attribut `attribut`. Aidez-vous de la deuxième équation ci-dessus. (Lorsque $p = 0$, le résultat de $p \log_2 p$ est indéfini. Il faut alors prendre la limite et traiter ce cas comme $p \log_2 p = 0$.)

Finalement, écrivez une méthode `h_C_A`, qui prenne quatre arguments : `self`, `donnees`, `attribut` et `valeurs`, et retourne l'entropie $H(C|A)$ de la classification des objets de `donnees` après avoir choisi l'attribut `attribut`. Aidez-vous de la première équation ci-dessus.

Exercice 12.1.2 La méthode `partitionne`
Dans la même classe, écrivez une méthode `partitionne` qui prendra trois paramètres (outre `self`) :

- `donnees` : les données d'apprentissage à partitionner ;
- `attribut` : l'attribut A de partitionnement ;
- `valeurs` : une liste contenant les valeurs a_j de l'attribut A.

La méthode doit retourner un dictionnaire qui associe à chaque valeur a_j de A une liste contenant les données pour lesquelles A vaut a_j. (Si une certaine valeur a_j n'apparaît pas dans `donnees`, la partition correspondante vaudra `[]`).

Exercice 12.1.3 La méthode `construit_arbre_recur`
Nous pouvons maintenant passer à la construction de l'arbre proprement dite. Pour cela, écrivez une méthode `construit_arbre_recur`, qui doit accepter trois paramètres :

- `self` : la classe ID3 ;
- `donnees` : les données de la sous-classification courante ;
- `attributs` : les attributs encore disponibles pour partitionner les exemples.

Cette méthode doit construire un arbre de décision en suivant l'algorithme ID3, dont le pseudo-code vous a été donné ci-dessus. Nous vous suggérons d'utilisez la méthode `h_C_A` pour construire une liste qui associe un attribut à son entropie, puis de choisir l'attribut dont l'entropie est la plus petite. Utilisez la méthode `partitionne` pour partitionner les exemples selon les valeurs de cet attribut.

Notez que `construit_arbre_recur` est appelée par `construit_arbre`, qui sert d'interface. Le code de cette dernière vous est donné et consiste en une routine qui extrait les domaines des attributs, avant de les passer à `construit_arbre_recur`.

Test du programme

Vous pouvez maintenant tester votre module avec les modules d'exemple :

- `exemple_profits.py` : présente des profils d'entreprises informatiques avec leurs espérances de profit ;
- `exemple_maladies.py` : essaie de trouver de quelle maladie souffre un enfant.

Utilisez les méthodes `repr_arbre` et `__repr__` de la classe `NoeudDeDecision` afin d'afficher l'arbre de décision résultant de `construit_arbre`. Essayez d'utiliser l'arbre de décision comme un expert humain de manière interactive à l'aide de la méthode `classifie` de la classe `NoeudDeDecision`.

Solutions à la page 393

Chapitre 13

Apprentissage non supervisé

Pour l'apprentissage de hiérarchies de classification, ainsi que pour l'apprentissage de descriptions caractéristiques, il est nécessaire que l'utilisateur indique pour chaque exemple la classe dont celui-ci est l'instance. Cette forme d'apprentissage est dite *supervisée*. On pourrait aussi imaginer que l'algorithme d'apprentissage décide lui-même des classes qui existent et de la classification de chaque exemple. Cet apprentissage, dit *non supervisé*, a donc pour but de former automatiquement des classes, processus appellé *clustering*.

L'apprentissage non supervisé a de nombreuses applications pratiques, surtout pour l'analyse de données et la découverte de nouvelles connaissances. Par exemple :

- dans un site web, on peut appliquer le clustering pour trouver des classes d'utilisateurs similaires ; on peut ainsi améliorer le site en leur offrant des chemins d'accès spécifiques ;
- dans un système de recherche d'informations, on peut regrouper les documents par sujets et ainsi les retrouver de manière plus fiable qu'avec une recherche par mots-clés ;
- dans un système d'analyse d'images, on peut séparer les différents objets représentés dans une image ;
- en bio-informatique, on peut appliquer le clustering ou des algorithmes génétiques afin de classer des segments d'ADN similaires pour en reconnaître la structure ;
- dans les sciences naturelles, on peut formuler des hypothèses de lois ou des théorèmes.

13.1 Apprentissage de sous-classes

L'apprentissage de sous-classes consiste à trouver une manière de regrouper des exemples en sous-classes naturelles, appellées des *clusters*. Certains regroupements montrent une plus grande cohérence que d'autres. Par exemple, étant donné les exemples :

A : grand,allongé,rouge
B : grand,rond,rouge
C : petit,rond,vert
D : petit,rond,rouge

on formers plutôt les groupes $\{A, B\}$ et $\{C, D\}$ que $\{A, C\}$ et $\{B, D\}$, car la description des classes serait beaucoup plus complexe dans le second cas.

Apprendre une sous-division implique une *recherche* parmi un certain nombre d'alternatives possibles pour sélectionner celle qui couvre les exemples de la meilleure manière. Cependant, dans la plupart de cas, cette recherche serait beaucoup trop complexe : pour n exemples, il y a k^n manières de les repartir en k clusters, ce qui rend une recherche exhaustive beaucoup trop complexe.

Le clustering fournit des méthodes pour regrouper les exemples en clusters de manière que chaque cluster contienne des exemples similaires. La base de ce processus est une mesure de distance $d(\underline{x}_1, \underline{x}_2)$ entre exemples. En général, les algorithmes de clustering sont des heuristiques, qui fonctionnent par approximations afin de garantir une complexité de calcul linéaire, ou presque linéaire, par rapport au nombre d'exemples.

Pour appliquer une méthode de clustering adéquate, il faut typiquement suivre les étapes suivantes :

- définir une représentation des exemples ;
- définir la mesure de similarité entre exemples, c'est-à-dire la distance (ou proximité) de deux exemples suivant le domaine d'application ; par exemple, pour des points sur un espace à deux dimensions, on utilise la distance euclidienne ;
- la phase du clustering proprement dit consiste alors à regrouper les exemples similaires entre eux suivant l'algorithme choisi.

Il existe deux types de clustering qui se distinguent par le critère de similarité qui est appliqué :

- Similarité *transitive* : on considère que si x est similaire à y, et y est similaire à z, alors x, y et z devraient faire partie du même cluster, même si x n'est pas similaire à z. Ce type de similarité se présente souvent quand il s'agit de partitionner des données, par exemple lors de la ségmentation d'objets dans une image ou la classification d'espèces en biologie. On représente généralement cette similarité sous forme de graphe.
- Similarité *non-transitive* : on considère que tous les exemples qui font partie d'un cluster doivent être similaires les uns aux autres. Cela veut dire que chaque cluster représente un prototype qui pourrait se substituer à n'importe lequel des exemples grâce à sa similarité. Ce type de similarité se présente quand il s'agit de simplifier des données en les résumant par des exemples prototypes, comme dans la classification, la recherche d'informations ou la recommendation, et de manière plus générale le *data mining* de régularités à partir d'une grande quantité de données. On représente généralement cette similarité par une mesure géométrique.

La partie gauche de la figure 13.1 montre un exemple où la transitivité de la similarité est nécessaire pour obtenir le clustering indiqué, tandis que dans l'exemple de droite tous les exemples sont suffisamment similaires pour que cette propriété ne soit pas nécessaire.

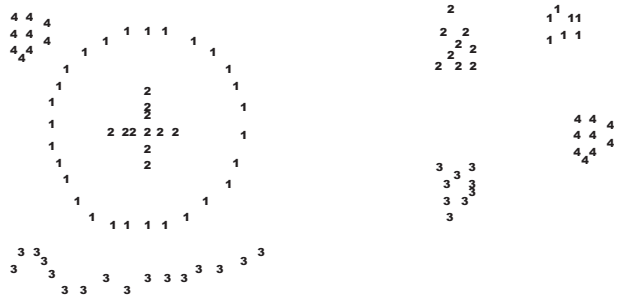


Fig. 13.1 *À gauche, le clustering nécessite une similarité transitive, tandis que l'exemple de droite n'a pas besoin de cette propriété.*

Il existe principalement trois grands groupes d'algorithmes de clustering : le *clustering hiérarchique*, *le clustering de partitionnement*, et le *clustering probabiliste*. Le premier créée une hiérarchie de regroupements, tandis que le deuxième définit un seul regroupement. Le troisième associe à chaque exemple une probabilité d'appartenance à chaque cluster. Nous les expliquons ci-après.

13.2 Clustering hiérarchique

Les algorithmes de clustering hiérarchique peuvent suivre deux approches suivant le type de structures de données crées au départ :

- Les méthodes par *agglomération* attribuent tout d'abord à chaque exemple un seul cluster. Cela signifie que pour n exemples, nous créons d'abord n clusters. Ces derniers sont alors fusionnés les uns après les autres jusqu'à ce qu'il n'en reste plus qu'un seul, qui couvre tous les exemples. L'algorithme peut s'arrêter lorsqu'il ne reste plus qu'un certain nombre de clusters, selon un seuil prédéfini.
- Les méthodes par *division* suivent le procédé inverse. Elles partent d'un cluster unique, qui englobe tous les exemples existants. Ce cluster est alors divisé au fur et à mesure suivant des critères précis.

Dendrogramme

Le clustering hiérarchique est une méthode qui construit une hiérarchie de clusters de manière à ce que ceux-ci soient regroupés à nouveau en clusters à un niveau supérieur. Le résultat sera ce que l'on nomme un *dendrogramme*. Un dendrogramme représente différents regroupements d'exemples et de niveaux de similarité (fig. 13.3). Il peut être coupé à différents niveaux, donnant lieu à différents clusters. Pour le montrer, considérons les exemples du plan de la figure 13.2. Pour le moment, nous faisons abstraction des trois clusters dessinés.

La figure 13.3 illustre le dendrogramme résultant d'un algorithme hiérarchique appliqué aux exemples de ce plan. Le bas du dendrogramme représente un clustering maximal, où chaque cluster ne contient qu'un seul exemple. Le

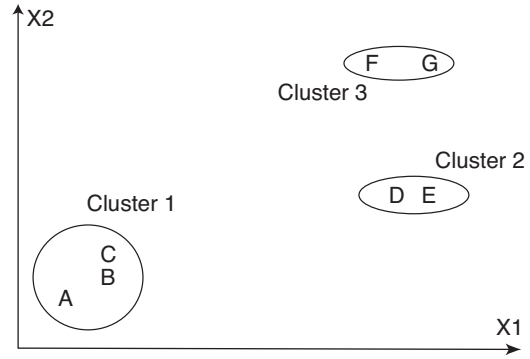


Fig. 13.2 *Exemples dans un plan avec trois clusters.*

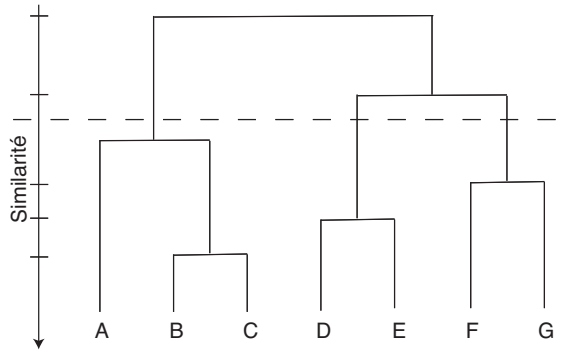


Fig. 13.3 *Le dendrogramme correspondant à la figure 13.2.*

haut montre le cluster qui regroupe tous les exemples. Si l'on prend la coupe en pointillé sur la figure 13.3, on obtient les trois regroupements représentés par les ovales de la figure 13.2.

Méthode par agglomération : single-link et complete-link

La manière la plus simple de construire un dendrogramme consiste à procéder par agglomération. Le schéma général est le suivant :

1) placer chaque exemple dans son propre cluster (singleton),
2) trouver la paire de clusters la plus similaire, étant donnée une définition de distance entre clusters, et la fusionner en un seul cluster,
3) calculer la distance entre ce nouveau cluster et tous les autres clusters,
4) répéter les pas 2 et 3 jusqu'à ce qu'il n'existe plus qu'un seul cluster contenant tous les exemples.

Sur la base de ce procédé général, il existe deux algorithmes principaux : le *single-link* et le *complete-link*. Ils se différencient par leur manière de calculer la distance entre deux clusters (étape 3) :

- dans l'algorithme *single-link*, la distance entre deux clusters est le minimum des distances entre toutes les paires d'exemples composées d'un élément de chaque cluster,
- dans l'algorithme *complete-link*, on prend le maximum des distances entre toutes les paires d'exemples appartenant à des clusters différents.

Il est intéressant de constater que ces deux algorithmes, qui sont très similaires, donnent des regroupements qui peuvent être de qualité très différente. La figure 13.4 illustre ce phénomène. Les exemples ont déjà été regroupés dans trois clusters 1, 2 et 3, et la question est alors de décider s'il faut regrouper cluster 1 avec 2 ou avec 3. Si on applique le critère *single-link*, on doit comparer les distances $d_{sl}(1,2)$ et $d_{sl}(1,3)$, et on regroupe donc 1 avec 3. Par contre, par le critère *complete-link*, on doit clairement regrouper 1 et 2.

Comme le *single-link* n'a besoin que d'une seule paire de membres similaires pour établir la connexion entre deux clusters, il se prête bien à des mesures de similarité transitives. Par contre, le *complete-link* exige que tous les membres soient similaires et ne suppose donc aucune transitivité de la mesure de similarité.

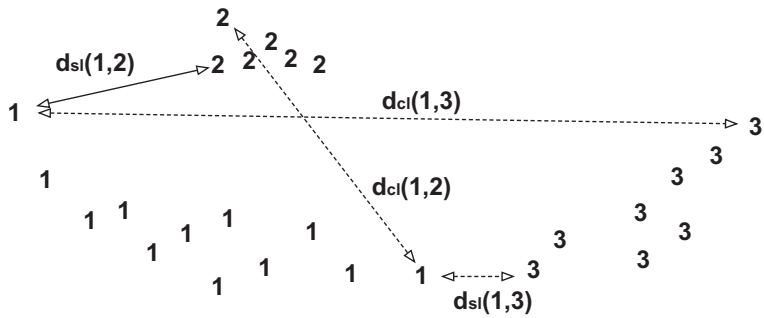


Fig. 13.4 *Différence entre les critères single-link et complete-link.*

Méthode par division

Si les algorithmes par agglomération s'efforcent de regrouper les exemples les plus similaires, les algorithmes par division se focalisent sur les dissimilarités en séparant les exemples les moins similaires.

Ces méthodes partent d'une représentation des similarités sous forme de graphe complet, dont les sommets correspondent aux exemples et dont chaque arête porte un poids égal à la similarité entre les exemples qu'elle lie. On divise alors le graphe en construisant des *coupes* qui contiennent des arêtes de poids minimal. Cela implique que la mesure de similarité soit considerée comme transitive, comme dans le critère single-link dans le clustering agglomératif.

L'algorithme le plus simple est basé sur les arbres couvrants. Avant de l'expliquer, rappelons quelques notions. Soit un graphe connecté et non dirigé. Un *arbre couvrant* (en anglais *spanning tree*) de ce graphe est un sous-graphe qui 1) est un arbre et 2) connecte tous les sommets. Un graphe peut avoir plusieurs arbres couvrants. Si un poids est attribué à chaque arête, on peut calculer le poids total de chaque arbre couvrant. Parmi tous les arbres couvrants possibles d'un graphe, l'*arbre couvrant de poids minimal* (en anglais *minimal spanning tree*, abrégé MST) est celui dont le poids est le plus faible pour tous les arbres couvrants.

Un algorithme de clustering peut utiliser cela pour créer un partitionnement. Il s'agit d'une méthode par division, qui procède en deux étapes principales :

- on crée tout d'abord l'arbre couvrant de poids minimum sur le graphe des exemples existants ;
- on efface l'arête dont le poids est le plus grand, afin de créer deux clusters ; cette deuxième étape est répétée jusqu'à obtenir un clustering satisfaisant selon un critère fixé d'avance.

La figure 13.5 montre comment appliquer cette méthode. Sur la base des exemples positionnés dans le plan, on établit un arbre couvrant de poids minimum. L'arête dont la longueur est la plus grande se trouve entre C et D. En l'éliminant, on obtient deux sous-graphes, qui définissent deux clusters.

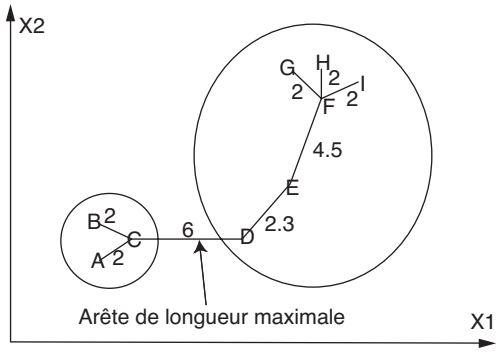


Fig. 13.5 *Algorithme utilisant un arbre couvrant de poids minimal.*

Il existe de nombreux algorithmes pour trouver un MST. Les meilleurs obtiennent une solution en un temps presque linéaire dans le nombre d'arêtes (donc quadratique dans le nombre d'exemples), et sa complexité n'est donc pas plus élevée que l'évaluation de toutes les similarités entre exemples qui est à la base du clustering.

Cependant, la méthode a deux points faibles :

- elle minimise le poids minimal des arêtes traversées par une coupe, mais ne prend pas en considération la somme des poids, qui peut parfois être plus significative,

- elle ne permet pas d'équilibrer la taille des sous-graphes qui résultent du fractionnement, et peut donc conduire à des arbres très inhomogènes.

Pour combler ces lacunes, on fait appel à des méthodes de *coupe à poids minimal* de la théorie des graphes. Ces méthodes permettent de minimiser la *somme* des poids des arêtes qui traversent une coupe, et non seulement le *minimum* des poids comme c'est le cas quand on utilise un MST. Cependant, la complexité de tels algorithmes est nettement plus élevée : la méthode la plus courante, celle de Ford-Fulkerson, a un temps de calcul quadratique dans le nombre d'arêtes, et est donc beaucoup plus élevé que l'approche basée sur le MST. De plus, cette méthode a tendance à sélectionner des coupes très déséquilibrées, qui isolent des petits groupes de noeuds.

On utilise donc une méthode de *clustering spectral* (*spectral clustering*), qui fait une approximation de la coupe à poids minimal et en même temps minimise la différence entre la taille des parties qui sont construites. Elle utilise une représentation du graphe de similarité sous forme de deux matrices quadratiques avec une rangée/colonne pour chaque nœud :

- la matrice W, qui représente les poids w des arrêtes entre noeuds et
- la matrice diagonale D, qui représente le degré (nombre d'arrêtes) de chaque noeud.[(1)]

On considère alors la matrice Laplacienne normalisée du graphe :

$$L = I - D^{-1}W$$

et plus précisément ses valeurs propres, qu'on suppose triées dans l'ordre croissant :

$$e_1 < e_2 < ... < e_n$$

La matrice Laplacienne normalisée d'un graphe connexe a les propriétés suivantes :

- la première valeur propre $e_1 = 0$;
- la deuxième valeur e_2 est associée à un vecteur propre qui définit une coupe entre deux sous-graphes C_1 et C_2 ; cette coupe minimise (approximativement) le critère NCut :

$$\left(\sum_{a \in cut} w(a)\right) \cdot \left(\frac{1}{\sum_{a \in C_1} w(a)} + \frac{1}{\sum_{a \in C_2} w(a)}\right)$$

  de sorte que les composantes positives du vecteur correspondent aux nœuds de C_1 et les composantes négatives à ceux de C_2.

(1) Comme le critère de coupe utilise la somme des poids, on suppose qu'on réduit le nombre d'arêtes en supprimant celles dont le poids est très faible, et qui ne comptent que très peu dans le poids total d'une coupe. Ceci permet de réduire la complexité.

Ce vecteur propre s'appelle aussi le *vecteur de Fiedler* (*Fiedler vector*). Le critère qu'il optimise prend en compte à la fois le poids de la coupe et l'équilibrage de la taille des deux parties, exprimée chacune par la somme des similarités entre les exemples qu'elle contient. Par application récursive, on peut donc construire sur cette base un clustering hiérarchique bien équilibré.

Une autre façon d'interpréter le vecteur de Fiedler consiste à le prendre comme une projection du graphe sur un axe, de sorte que les deux clusters se trouvent sur les moitiés positives et négatives de cet axe. On peut se demander quelle pourraît être la signification des vecteurs propres associés aux prochaines valeurs propres. Elles définissent en fait une projection du graphe dans d'autres dimensions, de telle sorte que les $2-(K+1)$-ème vecteurs définissent une projection en k dimensions, qui découpe le graphe en k clusters de points proches les uns des autres. On peut donc trouver une partition dite *spectrale* en k clusters, qui applique la transitivité de la similarité en utilisant des algorithmes de clustering de partitionnement dans l'espace défini par les k vecteurs propres. Ces algorithmes, notamment l'algorithme k-means, seront décrits dans la prochaine section.

Plus précisément, pour obtenir k clusters, on construit une matrice H de n rangées et k colonnes, qui consistent en des k vecteurs propres associées aux valeurs propres $e_2, .., e_{k+1}$. Ensuite, on procède au clustering des exemples dont les coordonnées sont données par les rangées de H, en utilisant comme similarité l'inverse de la distance Euclidienne. Ce clustering utilise l'algorithme k-means, décrit ci-dessous. L'avantage est qu'on peut obtenir un clustering de partitionnement avec un critère de similarité transitif, comme dans le cas de la segmentation d'image.

13.3 Clustering de partitionnement

Le clustering de partitionnement construit une partition unique des exemples, et non pas une structure hiérarchique. L'algorithme de clustering le plus connu et le plus appliqué est justement un algorithme de partitionnement : le *k-means*.

Algorithme k-means

Pour appliquer le *k-means*, on doit choisir au départ le nombre k de clusters que l'on veut construire. Comme initialisation, on choisit alors k *noyaux* $\underline{c}_j$, qui sont des exemples distincts, qui vont chacun caractériser un cluster C_j.

L'algorithme procède alors par itération :

- associer chaque exemple $\underline{x}_i$ au cluster dont le noyau est le plus proche, c'est-à-dire au C_j tel que $d(\underline{c}_j, \underline{x}_i)$ est minimale,
- pour chaque cluster C_j, remplacer le noyau par l'exemple qui est le plus au centre des exemples du cluster, c'est-à-dire par le $\underline{x_c} \in C_j$ tel que

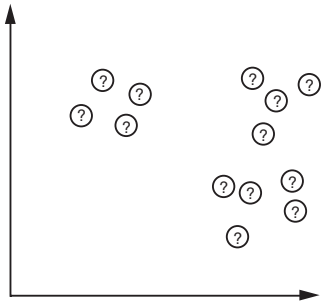
$$\sum_{\underline{x}_i \in C_j} d(\underline{x}_c, \underline{x}_i)^2$$

est minimal.

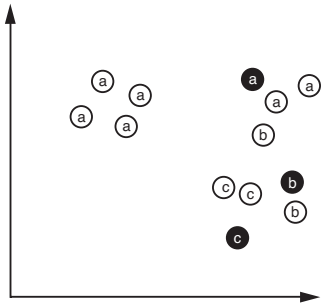
Après un certain nombre d'itérations, l'arrangement devrait se stabiliser et on obtient alors un regroupement en clusters. Notons qu'il s'agit d'un algorithme approximatif, qui ne donne aucune garantie quant à la qualité du résultat.

L'algorithme que nous venons de décrire est en fait la variante *k-médoïds* de *k*-means. L'algorithme *k*-means proprement dit s'applique uniquement à des attributs continus. Le noyau d'un cluster se calcule alors par les moyennes des attributs des instances qui en font partie. L'algorithme *k*-médoïds que nous présentons est une variante qui s'applique également à des attributs discrets, et nous allons utiliser le terme *k*-means pour cette variante.

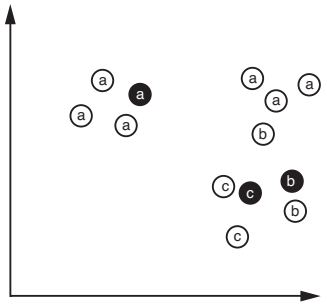
Comme exemple, considérons la situation suivante, dans laquelle les exemples ne sont pas encore classés :



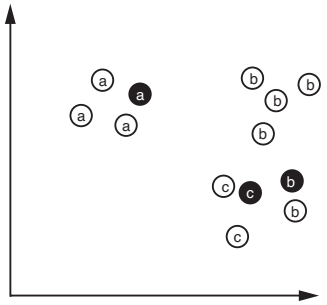
À l'initialisation, supposons que nous choisissons au hasard trois noyaux (indiqués en noir) et que nous classons les exemples ainsi :



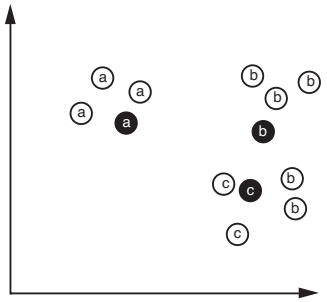
Ensuite, nous recalculons les noyaux :



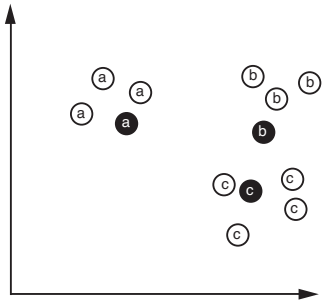
et nous classons les exemples à nouveau :



Après un nouveau calcul des noyaux :



et un reclassement des exemples :



nous avons construit un regroupement raisonnable en trois clusters.

L'application de *k-means* demande à l'utilisateur de faire le bon choix de k ainsi que des k noyaux d'apprentissage de départ. Il est également important d'utiliser une bonne mesure de distance, qui ait une signification réelle. Par exemple, pour des attributs symboliques ou logiques, la distance pourrait être le nombre d'attributs différents.

13.4 Clustering probabiliste

Nous avons passé en revue deux grands groupes de méthodes de clustering : le clustering hiérarchique et le clustering de partitionnement. Ces méthodes génèrent des partitions dans lesquelles chaque exemple n'appartient qu'à une et

une seule partition à la fois. Or, il existe aussi des applications pour lesquelles on souhaiterait disposer d'une certaine flexibilité dans le résultat, notamment en attribuant un exemple à plusieurs clusters à la fois. C'est ce que réalise le *clustering probabiliste*. Cette méthode permet à un exemple d'appartenir à plusieurs clusters, en utilisant une fonction qui définit une probabilité d'appartenance.

Prenons l'exemple illustré à la figure 13.6. Un algorithme déterministe attribuerait chaque exemple à un seul cluster, par exemple H1 ou H2. Un algorithme probabiliste, quant à lui, définirait deux clusters F1 et F2, et attribuerait à chaque exemple une probabilité d'appartenance. Pour notre exemple, nous aurions :

- F1 = (1,0.9), (2,0.8), (3,0.7), (4,0.6), (5,0.55), (6,0.2), (7,0.2), (8,0.0), (9,0.0)
- F2 = (1,0.0), (2,0.0), (3,0.0), (4,0.1), (5,0.15), (6,0.4), (7,0.35), (8,1.0), (9,0.9)

L'exemple 4 aurait alors une probabilité d'appartenance à F1 de 0.6 et une probabilité d'appartenance à F2 de 0.1.

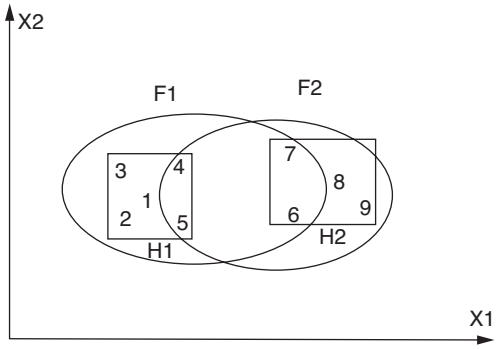


Fig. 13.6 *Un exemple de clustering probabiliste.*

Le modèle le plus courant est donné par un *mélange de Gaussiennes* tel qu'illustré par la figure 13.7 pour une seule dimension t. Dans un tel modèle, chaque cluster C_j est représenté par une distribution Gaussienne centrée sur Y_j avec variance σ_j^2. La probabilité qu'une instance appartenent au cluster C_j est égal à l'instance X_i est alors estimée comme :

$$P(X_i|j) = \frac{1}{\sigma_j\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{d(X_i,Y_j)}{\sigma_j}\right)^2}$$

où $d(X,Y)$ est la distance entre X_i et Y_j (dans le cas de la Figure 13.7, la différence en t).

Par exemple, t pourrait représenter le poids dans un ensemble de personnes, et on aimerait que l'algorithme trouve qu'il y a deux populations, les « o » (qui sont les femmes) et les « x » (qui sont les hommes).

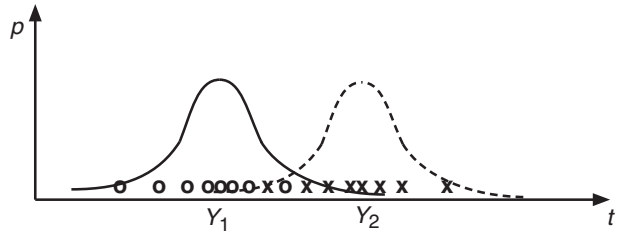


Fig. 13.7 *Un exemple d'un mélange de Gaussiennes dans une seule dimension t. Y_1 est le centre de la distribution des « o », Y_2 le centre des « x ». Chaque courbe Gaussienne indique la probabilité que le processus génère une instance de la classe correspondante avec cette valeur de t.*

Pour classer une instance donnée, nous aimerions connaître la probabilité qu'elle appartienne à une des classes. Nous exprimons l'appartenance par une variable *latente* $z_i \in \{1, .., k\}$, et la probabilité d'appartenance est alors :

$$P(z_i = j|X_i) = \frac{p(z_i = j)}{p(X_i)} P(X_i|j) \tag{13.1}$$

ce qui permet d'utiliser le modèle pour la classification des instances.

Pour apprendre le modèle, nous appliquons le principle du maximum de vraisemblance : nous cherchons

- pour chaque cluster, les paramètres Y_j et σ_j ainsi que
- pour chaque exemple, la probabilité d'appartenance $p(z_i = j)$,

de façon à maximiser la probabilité de l'ensemble des instances :

$$p(X_i) = \sum_{j=1}^{k} P(X_i|j)p(z_i = j)$$

Nous nous trouvons cependant confronté au problème de la poule et de l'œuf :

- supposons que nous connaissions pour chaque instance le cluster z_i auquel elle appartient, nous pouvons alors trouver les paramètres des distributions Gaussiennes qui maximisent la probabilité des instances en calculant Y_j comme moyenne des X_i avec $z_i = j$,
- supposons que nous connaissions les paramètre des distributions, alors nous pouvons estimer les valeurs les plus probables des z_i par l'équation 13.1.

Nous pouvons résoudre ce problème en utilisant un algorithme très similaire à k-means, l'algorithme de maximisation de l'espérance (*expectation maximisation*, EM). L'algorithme commence par une initialisation aléatoire des paramètres des clusters et de la distribution $p(z_i = j)$, et consiste ensuite en une itération de deux étapes :

- Espérance : pour chaque exemple X_i, on calcule la probabilité $L^t(i,j)$ d'appartenance à chaque cluster C_j, étant données les valeurs courantes des paramètres à l'itération t :

$$L^t(i,j) = \frac{p^t(z_i = j) g(X_i, \sigma_j^t, Y_j^t)}{\sum_{l=1}^{k} p^t(z_i = l) g(X_i, \sigma_l^t, Y_l^t)}$$

où $g(X, \sigma, Y)$ est la fonction de distribution Gaussienne centrée en Y avec variance σ, appliquée à X :

$$g(X, \sigma, Y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{d(X,Y)}{\sigma}\right)^2}$$

et $d(X,Y)$ est la distance entre X et Y, par exemple la distance Euclidienne $\sqrt{\sum_i (x_i - y_i)^2}$.

- Maximisation : pour chaque cluster, on recalcule le centre et la variance de la distribution de façon à maximiser la probabilité des exemples :

$$Y_j^{t+1} = \frac{\sum_{i=1}^{n} L^t(i,j) X_i}{\sum_{i=1}^{n} L^t(i,j)}$$

et

$$\sigma_j^{t+1} = \frac{\sum_{i=1}^{n} L^t(i,j) d^2(X_i, Y_j)}{\sum_{i=1}^{n} L^t(i,j)}$$

et la probabilité d'appartenance :

$$p_j^{t+1} = \frac{1}{n} \sum_{i=1}^{n} L^t(i,j)$$

Comme pour l'algorithme k-means, l'algorithme se termine quand il n'y a que peu de changement dans la classification.

On peut garantir que l'algorithme converge vers un optimum local. Il peut cependant en exister plusieurs, et l'on n'est pas sûr d'aboutir au meilleur. On applique donc souvent cet algorithme à plusieurs reprises, avec différentes initialisations, et on retient le résultat qui obtient les meilleures probabilités $P(X_i)$. Si on obtient ainsi différents modèles avec des probabilités similaires, cela signifie que les données ne suffisent pas pour identifier un seul modèle correct.

13.5 Apprentissage semi-supervisé

Souvent, on a la possibilité d'assigner des étiquettes à une petite partie des données à disposition, mais une grande partie reste sans étiquette. On aimerait donc appliquer à la fois un apprentissage supervisé sur les données dont on connaît la classification et un apprentissage non-supervisé sur le reste des données. L'apprentissage semi-supervisé permet de le faire.

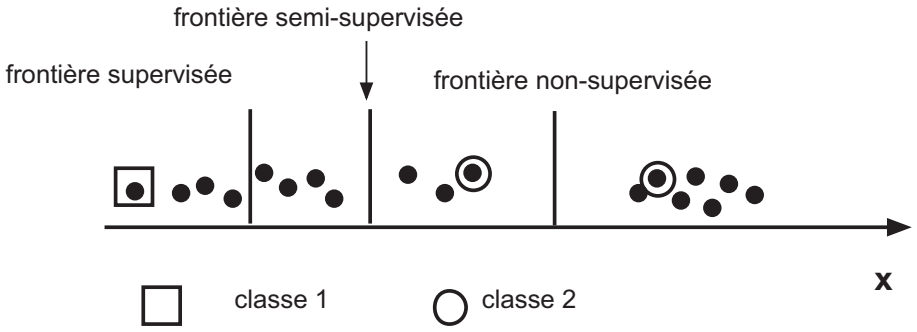


Fig. 13.8 *L'apprentissage semi-supervisé part d'un ensemble d'exemples dont certains seulement sont déjà classées.*

La figure 13.8 illustre l'intuition sous-jacente à l'apprentissage semi-supervisé [75]. On peut voir l'apprentissage semi-supervisé de deux manières :

- Un apprentissage supervisé qui utilise en plus les exemples non-classifiés. Le principe est que la classification apprise doit être aussi peu ambiguë que possible sur les exemples non-classifiées. Par exemple, lorsqu'il s'agit de construire une frontière de décision, on aimerait que celle-ci soit aussi éloignée que possible des exemples non-classifiés. Dans la figure 13.8, une analyse de la distribution permet ainsi de placer la frontière entre les deux classes d'une façon plus précise.
- Un apprentissage non-supervisé qui utilise en plus les exemples classées. Dans ce cas, on suppose que des exemples similaires doivent avoir la même classe, et on utilise donc les exemples classifiés comme contraintes pour la formation des clusters. Dans la figure 13.8, on sait que les exemples classifiés font partie de clusters différents et ceci permet de distinguer la bonne manière de les regrouper.

Pour le premier type, la technique la plus courante est de modifier la fonction d'optimisation d'une SVM (fig. 11.3) de façon à ce que, non seulement, on maximise la distance aux exemples classifiés les plus proches, mais également, dans une moindre mesure, la distance aux exemples non-classifiés. Ceci évite que la frontière soit placée dans une région dense, comme c'est le problème dans l'exemple de la figure 13.8.

Pour le deuxième type, on utilise habituellement les exemples classifiés comme noyaux des clusters, et étend ces derniers à partir d'eux. Ce principe s'applique évidement à des algorithmes comme k-means, mais également de façon plus large. Par exemple, si nous avons à construire un dictionnaire de mots positifs et négatifs, nous pouvons commencer avec les mots « bon » et « mauvais » et ensuite rajouter à chaque classe les mots qui sont similaires, c'est-à-dire qui apparaissent souvent dans la même phrase.

L'apprentissage semi-supervisé rencontre un grand succès en pratique et se trouve encore en plein développement.

Littérature

Les articles [69] et [70] sont des synthèses largement citées portant sur les algorithmes de clustering.

L'algorithme EM a été introduit à l'origine par [71] et est décrit également dans des livres généraux tels que celui de Bishop [54].

Le clustering spectral a été originalement introduit dans [72] et [73] ; [74] est un tutorial qui en explique les différents variantes.

[75] est un bon résumé des techniques d'apprentissage semi-supervisé.

Outils - domaine public

La bibliothèque `WEKA` réunit pratiquement tous les algorithmes d'apprentissage dans un seul cadre et est très commode pour construire rapidement des applications pratiques :

`http://www.cs.waikato.ac.nz/ml/weka/`

Il existe de nombreux autres outils, comme par exemple des environnements de programmation liés à l'*inductive logic programming*. La plupart sont disponibles dans le domaine public.

Application : Recommandation de produits

L'apprentissage non-supervisé, et plus particulièrement le clustering, est une méthode classique du marketing. En regroupant les produits qui sont souvent achetés ensemble, ou par les mêmes personnes, on peut savoir comment mieux les positionner dans un magasin, ou comment organiser la publicité.

Le clustering est utilisé dans les sites d'e-commerce afin de les rendre plus faciles à naviguer (les produits similaires sont regroupés sur la même page) ou pour donner des recommandations. Par exemple, Amazon.com donne une recommandation de produits qui sont susceptibles d'intéresser le client. Cela est fait sur la base des produits qu'il a déjà achetés et des groupes de produits achetés par d'autres clients.

Pour rendre ce processus plus efficace, Amazon utilise un processus de *collaborative filtering* qui est une forme de clustering spécifique pour la recommandation.

(Source : Greg Linden, Brenth Smith, Jeremy York : Amazon.com Recommendations, *IEEE Internet Computing* 7(1), pp. 76-80, 2003.)

13.6 Exercices

Exercice 13.1 Clustering

La notion de clustering recouvre un ensemble de procédés qui permettent, à partir d'une base de données, de regrouper les données similaires en agrégats, ou *clusters*, afin de réduire leur complexité apparente et de mettre à jour leur structure. Les classifications ainsi obtenues peuvent ensuite être utilisées dans diverses applications. Par exemple,

- en bio-informatique, des méthodes de clustering sont utilisées pour grouper les espèces d'êtres vivants similaires, afin de pouvoir généraliser à tout le groupe les propriétés connues d'une espèce particulière ; par exemple, un traitement efficace contre un certain agent pathogène peut s'avérer aussi efficace à l'égard d'autres micro-organismes du même cluster.
- dans le domaine de la vision, le nombre de pixels de différentes nuances de gris dans une image peut être réduit en regroupant plusieurs pixels en un même cluster et en leur attribuant une nuance de gris commune ;
- la classification des utilisateurs d'un site internet à partir de leurs habitudes de consultation du site peut permettre de concevoir des interfaces différentes pour chaque classe, adaptées à leurs attentes spécifiques ; de façon similaire, la classification du contenu d'un site internet peut aider à décider comment diviser ce contenu en rubriques thématiques.

Dans cette série d'exercices, nous vous proposons de travailler avec deux sortes de données :

- une liste de maladies avec leurs symptômes ; la classification de ces maladies doit permettre d'identifier des groupes d'affections similaires, susceptibles d'être traitées par des traitements similaires ;
- une liste d'entreprises actives dans le domaine de l'informatique, dont une classification en différents groupes doit permettre d'identifier les concurrents proches sur le marché.

Modules squelettes

Voici pour commencer les modules partiellement implémentés qui serviront de base à nos exercices. Le dernier constitue un module de test.

Module `.../moteurs_clustering/cluster.py` :

```
class Cluster:
    def __init__ ( self , donnees, nom=''):
        self .donnees = []
        self .ajoute_donnees(donnees)
        self .nom = nom

    def ajoute_donnee(self, donnee):
        self .donnees.append(donnee)
```

```
    def ajoute_donnees(self, donnees):
        for donnee in donnees:
            self.ajoute_donnee(donnee)
```

Module `.../moteurs_clustering/cluster_mean.py` :

```
from .cluster import Cluster

class ClusterMean(Cluster):
    def __init__(self, donnees, nom):
        Cluster.__init__(self, donnees, nom)
        self.noyau = self.donnees[0] if len(self.donnees) > 0 else None

    def centre(self, dist_f):
        print('à compléter')

    def vide(self, garde_noyau=False):
        if garde_noyau:
            self.donnees = [self.noyau]
        else:
            self.donnees = []
            self.noyau = None

    def __repr__(self):
        rep = 'Cluster {}: \n'.format(self.nom)
        for donnee in self.donnees:
            indent = '--->' if donnee == self.noyau else ' '*4
            rep += '{}{}\n'.format(indent, donnee)
        return rep
```

Module `.../moteurs_clustering/cluster_hierarchique.py` :

```
from .cluster import Cluster

class ClusterHierarchique(Cluster):
    def __init__(self, donnees, gauche=None, droite=None):
        Cluster.__init__(self, donnees)
        self.gauche = gauche
        self.droite = droite

    def est_terminal(self):
        return self.gauche is None and self.droite is None

    def repr_hierarchie(self, level=0):
        if self.est_terminal():
            rep = '    '*(level-1) + '|---' + str(self.donnees[0]) + '\n'
        else:
            rep = '    '*level + '|---' + '\n'

        if self.gauche is not None:
            rep += self.gauche.repr_hierarchie(level+1)
        if self.droite is not None:
            rep += self.droite.repr_hierarchie(level+1)

        return rep

    def __repr__(self):
        return 'Cluster racine: \n{}'.format(self.repr_hierarchie(level=0))
```

Module `.../moteurs_clustering/clustering.py` :

```
class Clustering:
    def __init__ ( self ):
        self . clusters = []

    def initialise_clusters ( self , donnees):
        return []

    def revise_clusters ( self ):
        return []

    def fini ( self , anciens_clusters ):
        return False

    def itere ( self , donnees):
        # Sauvegarde des clusters.
        anciens_clusters = []
        self . initialise_clusters (donnees)

        # Continue le clustering tant que les nouveaux clusters ont changé par
        # rapport à l' itération précédente.
        while not self. fini ( anciens_clusters ):
            anciens_clusters = self . clusters [:]
            self . revise_clusters ()
```

Module `.../moteurs_clustering/clustering_kmeans.py` :

```
from .cluster_mean import ClusterMean
from .clustering import Clustering

class ClusteringKMeans(Clustering):
    def __init__ ( self , k, dist_f ):
        super(). __init__ ()
        self .k = k
        self . dist_f = dist_f

    def noyaux(self, clusters ):
        return [cluster .noyau for cluster in clusters ]

    def initialise_clusters ( self , donnees):
        print('à compléter')

    def fini ( self , anciens_clusters ):
        print('à compléter')

    def revise_clusters ( self ):
        print('à compléter')

    def affiche_clusters ( self ):
        print('\n'. join ([str( cluster ) for cluster in self . clusters ]))
```

Module `.../moteurs_clustering/clustering_hierarchique.py` :

```
from .cluster_hierarchique import ClusterHierarchique
from .clustering import Clustering

class ClusteringHierarchique(Clustering):
```

```
    liens = {
        'single': min,
        'complete': max,
    }

    def __init__(self, type_lien, dist_f):
        super().__init__()
        self.dist_f = dist_f
        # Permet d'utiliser min ou max de manière générique en fonction du
        # paramètre type_lien.
        self.lien = self.liens[type_lien]

    def fusionne_clusters(self, cluster1, cluster2):
        donnees = cluster1.donnees + cluster2.donnees
        return ClusterHierarchique(donnees, cluster1, cluster2)

    def calcule_distance(self, cluster1, cluster2):
        print('à compléter')

    def initialise_clusters(self, donnees):
        print('à compléter')

    def fini(self, anciens_clusters):
        print('à compléter')

    def revise_clusters(self):
        print('à compléter')

    def affiche_clusters(self):
        print('\n'.join([str(cluster) for cluster in self.clusters]))
```

Module `.../exemple_clustering.py` :

```
from sys import argv, exit
from moteurs_clustering.clustering_kmeans import ClusteringKMeans
from moteurs_clustering.clustering_hierarchique import ClusteringHierarchique

def distance(donnee1, donnee2):
    if len(donnee1) != len(donnee2):
        raise Exception('Les deux données doivent avoir le même nombre d\'attributs.')

    return sum(attrib1 != attrib2 for attrib1, attrib2 in zip(donnee1, donnee2))

def est_entier_positif(s):
    " Teste si une string représente un entier positif. "

    try:
        return int(s) > 0
    except ValueError:
        return False

profits = [
    ('down', 'old', 'no', 'software'),
    ('down', 'midlife', 'yes', 'software'),
    ('up', 'midlife', 'no', 'hardware'),
    ('down', 'old', 'no', 'hardware'),
    ('up', 'new', 'no', 'hardware'),
    ('up', 'new', 'no', 'software'),
```

```
    ('up', 'midlife', 'no', 'software'),
    ('up', 'new', 'yes', 'software'),
    ('down', 'midlife', 'yes', 'hardware'),
    ('down', 'old', 'yes', 'software'),
]


maladies = [
    ('angine-érythémateuse', 'élevée', 'gonflées', 'oui', 'oui', 'non', 'non',
     'non', 'normale', 'normales', 'normaux'),
    ('angine-pultacée', 'élevée', 'points-blancs', 'oui', 'oui', 'non', 'non',
     'non', 'normale', 'normales', 'normaux'),
    ('angine-diphtérique', 'légère', 'enduit-blanc', 'oui', 'oui', 'non', 'non',
     'non', 'normale', 'normales', 'normaux'),
    ('appendicite', 'légère', 'normales', 'non', 'non', 'oui', 'non',
     'non', 'normale', 'normales', 'normaux'),
    ('bronchite', 'légère', 'normales', 'oui', 'non', 'non', 'oui',
     'oui', 'gênée', 'normales', 'normaux'),
    ('coqueluche', 'légère', 'normales', 'non', 'oui', 'non', 'oui',
     'oui', 'gênée', 'normales', 'normaux'),
    ('pneumonie', 'élevée', 'normales', 'non', 'non', 'non', 'oui',
     'non', 'rapide', 'rouges', 'normaux'),
    ('rougeole', 'légère', 'normales', 'non', 'oui', 'non', 'oui',
     'oui', 'normale', 'normales', 'larmoyants'),
    ('rougeole', 'légère', 'normales', 'non', 'oui', 'non', 'oui',
     'oui', 'normale', 'taches-rouges', 'larmoyants'),
    ('rubéole', 'légère', 'normales', 'oui', 'non', 'non', 'non',
     'non', 'normale', 'taches-rouges', 'normaux'),
    ('rubéole', 'non', 'normales', 'oui', 'non', 'non', 'non',
     'non', 'normale', 'taches-rouges', 'normaux'),
    ('rubéole', 'non', 'normales', 'oui', 'non', 'non', 'non',
     'non', 'normale', 'normales', 'normaux'),
]

if len(argv) < 4:
    print('On attend trois arguments: ' +
          'type des exemples ("profits", "maladies"), ' +
          'nombre de clusters (pour le clustering k-means), '
          'type de lien ("single", "complete", pour le clustering hiérarchique)')
    exit(1)

if argv[1].lower() == 'profits':
    donnees = profits
elif argv[1].lower() == 'maladies':
    donnees = maladies
else:
     print('Type des exemples accepté : profits ou maladies')
     exit(1)

if not est_entier_positif(argv[2]):
    print('Nombre de clusters accepté : entier positif')
    exit(1)

k = int(argv[2])

if argv[3].lower() not in ('single', 'complete'):
    print('Type de lien accepté : single ou complete')
    exit(1)
```

```
type_lien = argv[3]

clustering = ClusteringKMeans(k, distance)
clustering.itere(donnees)

print('Clustering k-means:')
clustering.affiche_clusters()

clustering = ClusteringHierarchique(type_lien, distance)
clustering.itere(donnees)

print('Clustering hiérarchique:')
clustering.affiche_clusters()
```

L'algorithme général de clustering

Les deux algorithmes de clustering que vous allez implémenter sont des algorithmes itératifs, qui construisent des clusters en suivant l'algorithme général ci-dessous :

```
clusters <- liste vide

Clustering(données)
1.  anciens_clusters <- liste vide
2.  clusters <- initialise_clusters(données)
3.  WHILE NOT fini(anciens_clusters) DO
4.      anciens_clusters <- sauvegarde des clusters
5.      clusters <- revise_clusters(clusters)
6.  END WHILE
END Clustering
```

La classe `Cluster`

La classe `Cluster` représente un cluster défini par un nom (optionnel) et par une liste de données. Cette classe contient donc deux attributs :

- `donnees` : la liste des données du cluster,
- `nom` : le nom (optionnel) du cluster.

Cette classe contient aussi des méthodes utilitaires qui permettent d'ajouter des données aux clusters.

La classe `Clustering`

La classe `Clustering` implémente l'algorithme général de clustering que nous avons vu ci-dessus. La méthode `itere` implémente cet algorithme en appelant trois méthodes : `initialise_clusters`, `revise_clusters` et `fini` ; elle joue donc le rôle d'un wrapper pour ces trois méthodes. `itere` initialise les clusters, puis les révise de manière itérative jusqu'à ce qu'ils soient stabilisés, et ne changent plus d'une itération à l'autre.

`Clustering` impose une structure générale que ses sous-classes sont contraintes de respecter. Afin d'obtenir le comportement désiré pour le clustering k-means

et le clustering hiérarchique, les méthodes `initialise_clusters`, `revise_clusters` et `fini` doivent cependant être implémentées différemment dans chaque sous-classe.

Exercice 13.1.1 Le clustering k-means (clustering de partitionnement)
L'algorithme k-means part d'une liste de noyaux, qui sont des données autour de chacune desquelles sera construit un cluster. Au cours d'une itération, chaque donnée est réaffectée au cluster du noyau duquel elle est le plus proche. Chaque cluster est ensuite recentré autour d'un nouveau noyau. L'algorithme se termine quand l'ensemble des noyaux n'a pas changé d'une itération à la suivante. Il existe plusieurs méthodes pour recentrer un noyau, qui donnent lieu à différentes variantes de l'algorithme. Dans cet exercice, nous appliquerons une variante adaptée à l'usage de données discrètes, qui est aussi appelée algorithme des k-médoïdes.

La classe `ClusterMean`

La classe `ClusterMean` étend la classe mère `Cluster` avec un attribut `noyau` qui doit aussi faire partie de la liste des données du cluster. Le constructeur initialise cet attribut au premier élément de la liste de données.

La méthode principale à implémenter est `centre`, qui met à jour le noyau d'un cluster afin que celui-ci soit au centre des données du cluster. Le nouveau noyaux sera la donnée qui minimise la somme quadratique des distances aux autres données du cluster. En d'autres termes, le noyau x_n du cluster C doit minimiser l'expression suivante :

$$\sum_{x \in C} d(x_n, x)^2$$

C'est pour cette raison que la méthode prend un argument `dist_f`, qui est la fonction de distance entre deux données d'un cluster. La distance que nous utiliserons sera calculée comme le nombre d'attributs différents dans les tuples représentant les éléments à comparer (voir `exemple_clustering.py`).

À plusieurs reprises, vous allez devoir parcourir une liste pour trouver l'élément qui minimise une certaine fonction. En Python, ceci peut être implémenté en une seule ligne, grâce à la fonction `min(liste, key=fonction)`, qui prend en paramètres une liste d'éléments et une fonction que l'on cherche à minimiser. Par exemple, pour trouver l'élément de la liste `[1, 2, -3]` qui a le plus petit carré, on peut utiliser la ligne de code `m = min([1, 2, -3], key=lambda x : x**2)`.

La classe `ClusteringKMeans`

La classe `ClusteringKMeans` étend la classe mère `Clustering` afin d'implémenter l'algorithme k-means. Elle ajoute deux nouveaux attributs :

- `k` : le nombre des clusters à construire,
- `dist_f` : la fonction de distance entre deux éléments.

Le constructeur de la classe prend ainsi en arguments le nombre de clusters souhaités et la fonction de distance.

Comme nous l'avons vu ci-dessus, les méthodes principales à implémenter sont `initialise_clusters`, `revise_clusters` et `fini`. La méthode `initialise_clusters` prend en argument la liste des données à regrouper. Elle initialise la liste `self.clusters` de sorte que celle-ci contienne k clusters avec comme noyaux les k premiers éléments de la liste de données. Tous les autres éléments sont affectés au premier cluster.

La méthode `revise_clusters` s'exécute en deux étapes :

- calcul des nouveaux clusters : initialisation des clusters avec uniquement les noyaux, calcul des distances de chaque élément aux noyaux et ajout de l'élément dans le cluster le plus proche ;
- pour chaque cluster ainsi obtenu, le noyau est mis à jour afin d'être au centre du cluster.

La méthode `fini` prend en argument la liste des anciens clusters. Elle compare cette liste avec les clusters actuels, afin de tester si l'algorithme a convergé. Afin d'implémenter cette méthode, vous pouvez comparer les listes de noyaux de ces deux ensembles de clusters : les noyaux ne changent pas si et seulement si les clusters restent les mêmes.

Astuce : Afin d'implémenter ces deux dernières méthodes, vous pouvez vous aider des méthodes `vide` de la classe `ClusterMean`, qui vide un cluster avec l'option de garder son noyau, et `noyaux` de la classe `ClusteringKMeans`, qui retourne les noyaux d'une liste de clusters.

Exercice 13.1.2 Le clustering hiérarchique

À la différence du clustering de partitionnement, le clustering hiérarchique par agglomération construit une classification en clusters de plus en plus larges, qui peut se présenter sous la forme d'un dendrogramme. La classification hiérarchique ainsi obtenue est organisée en groupes et en sous-groupes, afin de discerner les agrégats de similarité grossière des agrégats de similarité plus fine.

L'algorithme part d'un ensemble de clusters ne contenant chacun qu'une seule donnée, et, lors de chaque itération, fusionne les deux clusters les plus similaires. L'algorithme se termine quand toutes les données ont été regroupées en un seul cluster. La mesure de similitude entre deux clusters peut être calculée de différentes façons. Dans cet exercice, nous vous en proposons deux : la distance *single-link* et la distance *complete-link*. La distance single-link définit la similitude de deux clusters comme la plus courte distance entre deux données de ces clusters. À l'inverse, le complete-link considère la plus longue distance.

La classe `ClusterHierarchique`

La classe `ClusterHierarchique` représente un cluster (un nœud) dans le dendrogramme (un arbre binaire) construit par le clustering hiérarchique. Elle contient deux nouveaux attributs :

- `gauche` : le sous-cluster de gauche,
- `droite` : le sous-cluster de droite.

La classe `ClusteringHierarchique`

La classe `ClusteringHierarchique` implémente le clustering hiérarchique. Elle étend la classe mère `Clustering` avec deux nouveaux attributs :

- `type_lien` : le type de distance entre deux clusters (`'single'` ou `'complete'`),
- `dist_f` : la fonction de distance entre deux données.

Le constructeur de la classe prend en arguments la fonction de distance entre deux données et le type de distance entre deux clusters.

Les méthodes principales que vous devez implémenter sont `initialise_clusters`, `revise_clusters`, `clusters_distances` et `fini`. La méthode `initialise_clusters` initialise la liste `self.clusters` de sorte qu'elle contienne un cluster (un nœud) pour chaque donnée de la liste passée en argument.

La méthode `revise_clusters` cherche les deux clusters les plus proches et les fusionne en un seul grâce à la méthode `fusion` de la même classe, qui retourne le nouveau nœud. Elle les retire ensuite de la liste des clusters pour y ajouter le produit de leur fusion. Cette méthode s'appuie sur la méthode `calcule_distance`, qui doit, selon le type de lien indiqué, retourner le minimum ou le maximum des distances entre chaque paire des données des deux clusters. Selon que la distance entre deux clusters est définie comme le minimum des distances entre chaque paire d'éléments ou comme le maximum, on obtient en effet une distance de type *single-link* ou de type *complete-link*.

Rappelons que c'est la méthode `Clustering.itere` qui implémente l'algorithme général de clustering, et qui joue le rôle d'un wrapper pour les méthodes `initialise_clusters` et `revise_clusters`. Dans le cas du clustering hiérarchique, `itere` va donc initialiser les clusters, puis les fusionner de manière itérative jusqu'à ce qu'il ne reste qu'un seul élément dans la liste `self.clusters`. Cet élément constituera la racine de la hiérarchie construite par le clustering. La méthode `fini`, quant à elle, arrête l'algorithme quand la taille de la liste des clusters est réduite à un seul élément.

Test du programme

Une fois que vous avez terminé l'implémentation des méthodes manquantes, il ne vous reste plus qu'à tester ces deux algorithmes sur les deux exemples fournis : maladies et profits d'enterprises.

Pour le clustering k-means, faites varier le nombre de clusters (ainsi que l'initialisation des clusters et leurs noyaux, si vous êtes motivés). Que pensez-vous de la qualité des clusters ? Les données classifiées dans un même cluster sont-elles toujours très similaires ? Est-il facile de choisir le nombre de clusters ?

Pour le clustering hiérarchique, quelles différences obtenez-vous si vous alternez entre les méthodes *single-link* et *complete-link* ? Que pensez-vous de la qualité des clusters ainsi obtenus, en comparaison avec l'algorithme k-means ?

Solutions à la page 396

Chapitre 14

Apprentissage bio-inspiré

L'apprentissage est une caratéristique de nombreux êtres vivants. Il n'est donc pas surprenant que l'observation du monde naturel nous fournisse des sources d'inspiration. Deux techniques principales sont ainsi calquées sur des modèles biologiques :

- les *réseaux de neurones artificiels*, qui forment une structure imitant les réseaux de neurones du cerveau et
- les algorithmes *génétiques*, qui réalisent un processus d'optimisation en simulant l'évolution des organismes.

Dans les deux cas, bien que ces techniques s'inspirent de la nature, elles utilisent des modèles simplifiés, qui sont bien adaptés à l'implémentation informatique, mais assez éloignés de modèles biologiques corrects.

14.1 Réseaux de neurones artificiels

Nous avons déjà vu au chapitre 11 la méthode du perceptron pour la classification binaire. Le perceptron a été inspiré par le fonctionnement des neurones, dont il constitue un modèle fortement simplifié, qui néglige notamment la dynamique temporelle à l'œuvre dans les neurones biologiques.
Étant donnée cette analogie, il est naturel de regrouper de tels éléments de calcul dans un réseau qui imite le réseau qu'on retrouve dans le cerveau humain. Un tel réseau est bien plus puissant qu'un seul perceptron et peut implémenter des frontières de décision plus complexes. En outre, il peut réaliser d'un seul coup des classifications ou prédictions multiples, tandis que le perceptron est limité à séparer deux classes.

La version la plus simple d'un tel réseau utilise deux couches de neurones, comme montré dans la figure 14.1. On y trouve :

- une couche d'entrées $e_1, .., e_n$, qui représentent chacune une valeur d'activation d'un trait dans la situation ou dans l'exemple courant ; ces valeurs sont transmises à
- une couche de neurones cachés (*hidden units*) $H_1, .., H_k$, qui reçoivent des entrées depuis les neurones de la couche d'entrées et fournissent les résultats vers
- une couche de neurones de sortie $O_1, .., O_m$, qui retournent les résultats du réseau, par exemple des classifications ou des prédictions.

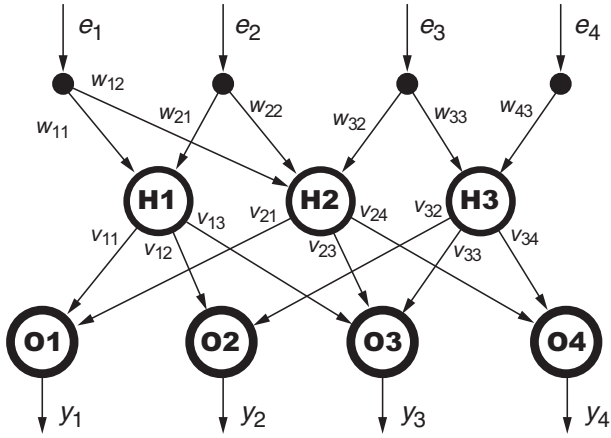


Fig. 14.1 *Exemple d'un réseau de neurones artificiels multicouche.*

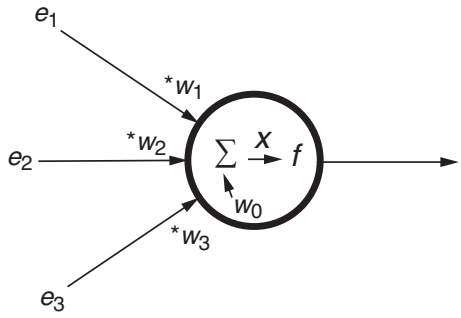


Fig. 14.2 *Fonctionnement d'un neurone artificiel.*

La couche cachée permet de construire des traits complexes en fournissant différentes classifications des entrées, qui peuvent être combinées de différentes manières par la couche de sortie. Ceci donne à une telle architecture, même avec seulement deux niveaux, une puissance largement supérieure à celle d'un seul perceptron.

À l'instar du perceptron, un neurone artificiel (fig. 14.2) calcule la somme de ses entrées e_i, pondérées par les poids de connexion, et ajoute un *biais* w_0 :

$$x = \sum_{e_i \in entrees} w_i e_i + w_0$$

Ensuite, on applique une fonction non-linéaire, dite *fonction d'activation*, pour obtenir la valeur de sortie. Cette fonction peut être une simple fonction de seuil (threshold) :

$$f(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}$$

ou une fonction sigmoïde :

$$f(x) = \frac{1}{1 + e^{-x}}$$

ou bien une fonction redresseur (rectifier) :

$$f(x) = \max(0, x)$$

La fonction sigmoïde est identique à la transformation logistique (voir sect. 11.6) et assure que la fonction d'activation prend des valeurs dans l'intervalle [0..1]. Un réseau à 2 couches présente donc une équivalence avec une regression logistique. La fonction redresseur est surtout importante dans un réseau possédant de nombreuses couches, où elle permet un meilleur apprentissage, comme nous le verrons par la suite.

La classification ou prédiction effectuée par le réseau est déterminée par les poids w_{ij}. Dans l'exemple de la figure 14.1, le neurone caché $H1$ calcule $f(w_{11} \cdot e_1 + w_{21} \cdot e_2)$, et la sortie $O3$ est obtenue par $f(v_{13} \cdot H1 + v_{23} \cdot H2 + v_{33} \cdot H3)$. En choisissant convenablement les connections et les poids, on peut donc construire une très large gamme de fonctions. En pratique, des réseaux de neurones artificiels peuvent représenter pratiquement toutes les fonctions, soit pour la classification, soit pour la régression et la prédiction, à condition qu'ils soient suffisement complexes.

L'apprentissage d'un réseau de neurones artificiels consiste donc à déterminer les poids qui permettront la meilleure performance sur les exemples fournis en entrée. Pour la couche de sortie, il est possible d'appliquer l'algorithme du perceptron, ou, plus généralement, la descente de gradient stochastique. Par contre, pour les couches cachées, on ne connaît pas directement l'erreur à attribuer à un neurone particulier. On peut cependant déterminer cette erreur par rétropropagation (*back-propagation*), en calculant la variation de l'erreur en fonction de chacun des poids sous la forme du gradient de la fonction d'activation.

Comme le montre la figure 14.3, on utilise les erreurs observées sur la couche de sortie $\epsilon_1, .., \epsilon_m$ pour mettre à jour les poids et rétropropager les erreurs vers la couche précédente. Chaque neurone artificiel (fig. 14.4) effectue les deux opérations suivantes :

1) *correction* de w_i par gradient

$$g_i = \frac{\delta \epsilon_j}{\delta w_i} = \frac{df}{dx}\frac{\delta x}{\delta w_i} = \frac{df}{dx} e_i$$

$\Delta w_i = -\alpha \epsilon g_i$ ($\alpha \in [0..1]$ = taux d'apprentissage)

2) *rétropropagation* de l'erreur vers la couche cachée par gradient

$$r_i = \frac{\delta \epsilon}{\delta e_i} = \frac{\delta f}{\delta x}\frac{\delta x}{\delta e} = \frac{\delta f}{\delta x} w_i$$

$\epsilon_i^c = \sum_j r_{ij} \epsilon_j$ (somme sur tous les neurones j connectées)

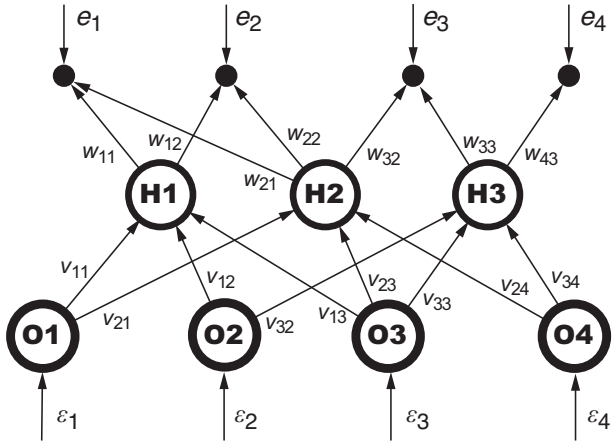


Fig. 14.3 *Apprentissage dans un réseau multicouches par rétropropagation des erreurs de sortie.*

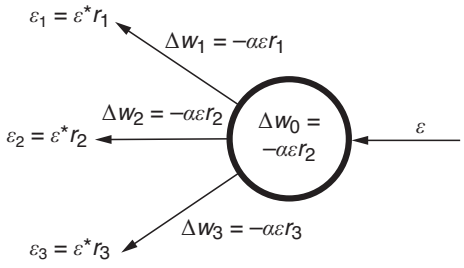


Fig. 14.4 *Rétropropagation sur un neurone.*

La rétropropagation dépend alors de facon cruciale de la forme du gradient de la fonction d'activation f utilisée par le neurone artificiel. Pour la fonction de seuil du percepton, le gradient est soit infini soit zéro, et ne permet donc aucune rétropropagation. Heureusement, d'autres fonctions sont mieux adaptées :

- pour la fonction sigmoïde $f(x) = \frac{1}{1+e^{-x}}$, le gradient peut se calculer en utilisant la valeur de la fonction même :

$$\frac{\delta f}{\delta x} = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)(1-f(x))$$

- pour la fonction redresseur, le gradient est encore plus simple : $f(x) = \max(0, x)$:

$$\frac{\delta f}{\delta x} = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}$$

Chaque neurone effectue donc la rétropropagation de l'erreur vers les neurones qui lui fournissent ses entrées. Ces neurones somment les erreurs qu'ils reçoivent à leur sortie et appliquent également à leur tour la même procédure.

Comme dans le cas du perceptron, on applique cet algorithme itérativement à des exemples sélectionnés de façon aléatoire. On obtient donc ainsi une descente de gradient stochastique. L'algorithme se termine quand le réseau atteint un taux d'erreurs acceptable.

Bien qu'une grande partie des applications de réseaux de neurones artificiels se contentent de deux couches (comme dans la figure 14.1), la rétropropagation en autorise un nombre nettement plus élevé, par exemple une trentaine. On parle alors de réseau profond (*deep neural net*) dont l'apprentissage constitue le *deep learning*. Dans un tel réseau profond, l'apprentissage est cependant beaucoup plus lent et nécessite une grande quantité de données.

Les réseaux profonds sont donc utiles surtout pour des applications dans lesquelles on dispose de données abondantes, comme la vision, la reconnaissance de la parole, ou des jeux qui peuvent être répetés indéfiniment (comme le programme `AlphaGo`, qui est ainsi devenu champion du monde de Go). Les couches multiples se révèlent utiles, car elles permettent de créer une hiérachie de détecteurs de traits de plus en plus complexes. Ainsi, dans un système de vision, on peut observer que les premières couches apprennent à reconnaître des traits simples, comme des lignes et des formes géométriques, qui seront ensuite composées pour former des détecteurs de formes plus complexes, tels que des éléments de visages, dans des couches supérieures.

Le développement de réseaux profonds a longtemps été empêché par le problème du *gradient disparaissant* (*vanishing gradient*) : la derivée de la fonction sigmoïde restant toujours inférieure à 1, le signal d'erreur rétropropagé d'une couche à l'autre devient de plus en plus faible. L'utilisation de la fonction redresseur a permis d'éliminer ce problème et ainsi d'entraîner des réseaux beaucoup plus profonds.

Un deuxième problème lié à la complexité de l'apprentissage est que la descente de gradient conduit vers un minimum local, et que dans un grand réseau il n'est pas facile de recommencer avec différents points de départ. On utilise alors la méthode du *dropout* pour créer des variations aléatoires, qui permettent d'échapper aux minima locaux. Elle consiste à enlever au hasard quelques neurones lors du calcul du gradient. L'apprentissage s'arrête alors uniquement si le résultat est robuste à l'égard des variations produites par cette méthode, ce qui donne des résultats plus proches de l'optimum.

Le deep learning est en fait une technique très ancienne, qui a été mise au point dans les années 1980 et qui a suscité un vif intérêt à cette époque déjà. Pendant longtemps, la technique n'a cependant fourni que des résultats médiocres. On pense aujourd'hui que cela était dû au manque de données et de puissance de calcul. Depuis 2010, le deep learning a accumulé de nombreux succès, en particulier dans les domaines de la vision et de la reconnaissance automatique de la parole. Des progrès supplémentaires ont été obtenus par l'intégration du deep learning avec d'autres techniques d'Intelligence Artificielle, ce qui a rendu possibles des applications dans d'autres domaines. Par exemple, le programme `AlphaGo`, qui intègre des réseaux profonds dans un système de planification avec adversaire, a été capable de battre le champion du monde de Go.

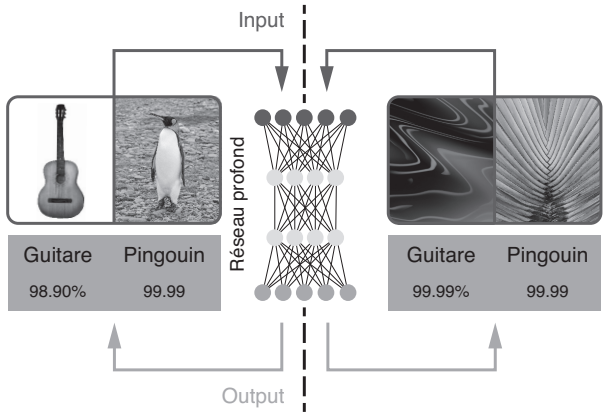


Fig. 14.5 *Ambiguïté du classificateur appris par* deep learning *: le réseau entrainé sur des guitares et des pingouins admet également d'autres structures à la place.*

Lors de l'utilisation des techniques du deep learning, on ne peut plus vraiment interpréter les contenus appris par le réseau comme de nouvelles connaissances explicites. Il est donc aussi difficile de vérifier et de garantir que ces connaissances sont effectivement correctes. Par exemple, un réseau entraîné pour reconnaître des guitares et des pingouins pourra également admettre des images plus abstraites comme étant de tels objets (fig. 14.5). Si on veut utiliser un système qui repose sur le deep learning pour la prise de décisions importantes, cette imprévisibilité peut poser de sérieux problèmes.

14.2 Algorithmes génétiques

Les algorithmes génétiques s'inspirent de l'évolution biologique et restent en effet très proches de celle-ci. L'idée est de chercher un résultat d'apprentissage en faisant évoluer une *population* de plusieurs modèles possibles, dont les qualités sont mises en compétition. Cette technique peut permettre d'éviter de rester bloqué dans des *optima locaux*. Les algorithmes génétiques permettent l'apprentissage et la découverte d'une structure qui satisfait un critère d'évaluation donné. L'apprentissage se fait par une recherche incrémentale et impose donc très peu de restrictions sur les modèles qui peuvent être appris.

En entrée, un algorithme génétique dispose :

- d'une population initiale de n solutions potentielles ; chaque solution est représentée par un « chromosome », qui est une version codée de toutes ses caractéristiques sous la forme d'une chaîne de symboles (en s'inspirant de la biologie) ;
- d'une fonction f d'évaluation des solutions ; f n'est pas forcément une fonction mathématique des chromosomes, elle peut aussi consister en une simulation de la performance des solutions dans le problème auquel elles doivent être appliquées ; cela permet de modéliser le principe de « sélection naturelle ».

À chaque itération, l'algorithme génère de nouvelles solutions en appliquant des opérateurs de *mutation* et de *combinaison* sur des membres de la population courante. Ces opérateurs s'inspirent également de la biologie :

- la *mutation* change un élément du chromosome de manière aléatoire,
- la *combinaison* de deux chromosomes crée un nouveau chromosome combinant des éléments des deux premiers.

Une nouvelle population est constituée à partir de la population courante en éliminant certaines solutions peu performantes et en en créant de nouvelles par mutation et combinaison. Par exemple, la génération suivante pourrait se constituer de :

- k solutions dont la fonction d'évaluation est la plus élevée (c'est à ce niveau qu'est effectuée la sélection naturelle) et
- un nombre $(n - k)$ de solutions choisies au hasard parmi les autres.

L'application itérative de la procédure conduit à l'optimisation de la population selon le critère d'évaluation choisi au départ.

Comme exemple d'une application, considérons un fabricant de chocolat qui veut optimiser la qualité de ses produits. On suppose que la qualité du chocolat dépend fortement de la quantité de sucre et de cacao ajoutés dans la pâte. La forme précise de cette relation, comme la montre la figure 14.6, est cependant inconnue et peut être évaluée uniquement en produisant les chocolats correspondants et en les mettant en vente.

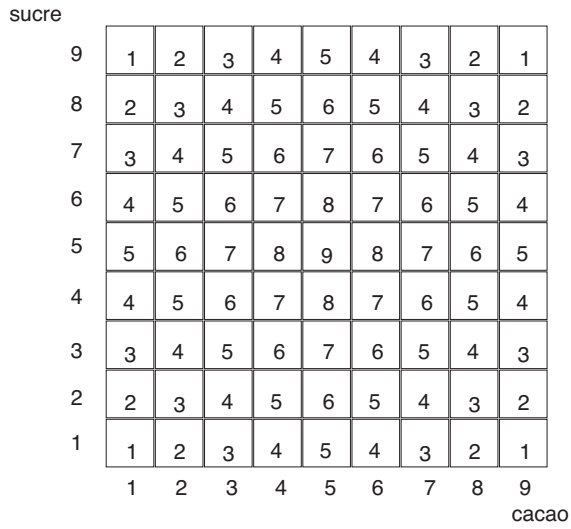
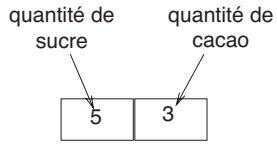


Fig. 14.6 *La qualité du chocolat en fonction de la quantité de sucre et de cacao dans la pâte.*

Une solution peut être représentée par un « chromosome » :



Les opérateurs sont alors :

- la mutation : la quantité de sucre ou de cacao augmente ou décroît de 1,
- la combinaison z de deux chromosomes x et y : $sucre(z) = sucre(x)$, $cacao(z) = cacao(y)$.

Pour un problème dont la fonction d'évaluation est simple, comme celle de la figure 14.6, la mutation suffit pour obtenir la meilleure solution. Elle équivaut à un processus de *hill-climbing* couramment utilisé en optimisation.

sucre									
9	1	2	3	4	5	4	3	2	1
8	2	0	0	0	0	0	0	0	2
7	3	0	0	0	0	0	0	0	3
6	4	0	0	7	8	7	0	0	4
5	5	0	0	8	9	8	0	0	5
4	4	0	0	7	8	7	0	0	4
3	3	0	0	0	0	0	0	0	3
2	2	0	0	0	0	0	0	0	2
1	1	2	3	4	5	4	3	2	1
	1	2	3	4	5	6	7	8	9
									cacao

Fig. 14.7 *Une fonction d'évaluation qui nécessite la combinaison en plus de la mutation.*

Par contre, si la fonction d'évaluation contient des minima ou maxima locaux, la combinaison est nécessaire pour les surmonter. La figure 14.7 montre un exemple d'une fonction d'évaluation dans laquelle les très bonnes solutions de la partie centrale sont entourées d'un « fossé » de zéros, qui les sépare des solutions médiocres des bords. Il est difficile, voire impossible, pour l'optimisation basée sur des critères locaux, de traverser le « fossé » constitué par les valeurs 0, car toute mutation qui se rapprocherait du bon résultat prendrait d'abord une valeur zéro et serait éliminée.

La combinaison permet de trouver un bon résultat en combinant deux solutions qui sont chacune localement optimales :

(5 1) avec (1 5) ⇒ (5 5)

Ce phénomène est typique des applications d'algorithmes génétiques. L'opérateur de combinaison est donc essentiel pour leur performance. Pour bien exploiter cet opérateur, il est important de maintenir une certaine diversité dans la population. Il peut donc être utile d'encourager explicitement cette diversité, par exemple en modifiant la fonction d'évaluation pour favoriser des distributions plus variées. On peut ainsi remplacer la fonction $f(s)$ par la somme $f(s) + g(s)$, où $g(s)$ caractérise le degré d'individualité :

$$g(s) = \frac{1}{\sum_i \frac{1}{d^2(s,s_i)}}$$

Les algorithmes génétiques sont souvent utilisés pour effectuer une optimisation dans des domaines où la fonction d'optimisation est peu claire, et qui nécessitent donc cette forme d'apprentissage au lieu d'un calcul direct de la solution optimale. Les applications sont très variées et incluent par exemple l'optimisation de la forme de turbines ou l'optimisation de stratégies de négociation sur des marchés financiers.

Littérature

Le livre général [54] contient une bonne introduction aux réseaux de neurones artificiels. Le livre [76] entre en plus de détail dans le deep learning.

Les livres [77] et [78] introduisent les algorithmes génétiques ; de nombreuses publications existent quant à leurs applications.

Solutions des exercices

Chapitre 3

Algorithmes d'inférence

Exercice 3.1 : Inférence à chaînage avant sans variables

Module `.../moteur_sans_variables/regle_sans_variables.py` :

```
class RegleSansVariables:
    def __init__(self, conditions, conclusion):
        self.conditions = set(conditions)
        self.conclusion = conclusion

    def depend_de(self, fait):
        return fait in self.conditions

    def satisfaite_par(self, faits):
        return self.conditions.issubset(faits)

    def __repr__(self):
        return '{} => {}'.format(str(list(self.conditions)),
                                 str(self.conclusion))
```

Module `.../moteur_sans_variables/chainage_avant_sans_variables.py` :

```
from .chainage import Chainage

class ChainageAvantSansVariables(Chainage):
    def chaine(self):
        queue = self.connaissances.faits[:]
        self.reinitialise()

        while len(queue) > 0:
            fait = queue.pop(0)

            if fait not in self.solutions:
                self.solutions.append(fait)
                self.trace.append(fait)

                # Vérifie si des règles sont déclenchées par le nouveau fait.
                for regle in self.connaissances.regles:
                    if regle.depend_de(fait) and regle.satisfaite_par(self.solutions):
                        queue.append(regle.conclusion)
                        self.trace.append(regle)

        return self.solutions
```

Exercice 3.2 : Moteur d'inférence à chaînage avant avec variables

Module `.../moteur_avec_variables/proposition_avec_variables.py` :

```
def est_atomique(proposition):
    return type(proposition) == type('')
```

```
def est_une_variable(proposition, marqueur='?'):
    return est_atomique(proposition) and proposition[0] == marqueur

def tete(proposition):
    if est_atomique(proposition):
        raise Exception("Proposition atomique: Impossible de la segmenter.")
    elif len(proposition) > 0:
        return proposition[0]
    else:
        raise Exception("Proposition vide: Impossible de la segmenter.")

def corps(proposition):
    if est_atomique(proposition):
        raise Exception("Proposition atomique: Impossible de la segmenter.")
    elif len(proposition) > 0:
        return proposition[1:]
    else:
        raise Exception("Proposition vide: Impossible de la segmenter.")

def lister_variables(proposition):
    variables = set()
    if est_atomique(proposition):
        if est_une_variable(proposition):
            variables.add(proposition)
    else:
        for sous_prop in proposition:
            variables.update(lister_variables(sous_prop))
    return variables
```

Module `.../moteur_avec_variables/regle_avec_variables.py` :

```
class RegleAvecVariables:
    def __init__(self, conditions, conclusion):
        self.conditions = conditions
        self.conclusion = conclusion

    def depend_de(self, fait, methode):
        envs = {}

        for condition in self.conditions:
            # Si au moins une des conditions retourne un environnement,
            # nous savons que la proposition satisfait une des conditions.
            env = methode.pattern_match(fait, condition, {})
            if env != methode.echec:
                envs[condition] = env

        return envs

    def satisfaite_par(self, faits, cond, env, methode):
        envs = [env]

        # On n'a pas besoin de tester ``cond`` car cela a été fait dans l'appel
        # à ``depend_de`` qui précède l'appel à cette méthode.
        conditions_a_tester = [cond1 for cond1 in self.conditions if cond1 != cond]

        # Pour chaque condition dans la liste des conditions, si la liste
        # des environnements n'est pas vide, on y ajoute les environnements
        # qui permettent de satisfaire une des conditions.
```

```
        for cond1 in conditions_a_tester :
            envs_nouveaux = []

            for fait in faits :
                for env1 in envs:
                    env1 = methode.pattern_match(fait, cond1, env1)
                    if env1 != methode.echec:
                        envs_nouveaux.append(env1)

            # Si au moins une condition n'est pas satisfaite ,
            # la règle ne l' est pas non plus.
            if len(envs_nouveaux) == 0:
                return []

            envs = envs_nouveaux

        return envs

    def __repr__( self ):
        return '{} => {}'.format(str(self.conditions), str( self . conclusion))
```

Module `.../moteur_avec_variables/chainage_avant_avec_variables.py` :

```
from moteur_sans_variables.chainage import Chainage
from . filtre import Filtre

class ChainageAvantAvecVariables(Chainage):
    def __init__ ( self , connaissances, methode=None):
        Chainage. __init__ ( self , connaissances)

        if methode is None:
            self . methode = Filtre()
        else:
            self . methode = methode

    def instancie_conclusion ( self , regle , envs):
        return [self . methode.substitue(regle.conclusion , env) for env in envs]

    def chaine( self ):
        queue = self.connaissances. faits [:]
        self . reinitialise ()

        while len(queue) > 0:
            fait = queue.pop(0)

            if fait not in self . solutions :
                self . trace . append(fait)
                self . solutions . append(fait)

                # Vérifie si des règles sont déclenchées par le nouveau fait.
                for regle in self . connaissances. regles :
                    cond_envs = regle.depend_de(fait, self . methode)
                    for cond, env in cond_envs.items():
                        # Remplace l'environnement par ceux qui satisfont
                        # toutes les conditions de la règle et pas seulement la
                        # première condition.
                        envs = regle. satisfaite_par ( self . solutions , cond, env,
                        self . methode)
```

```
                    # Ajoute la conclusion de la règle instanciée pour tous
                    # les environnements possibles.
                    if len(envs) > 0:
                        queue.extend(self.instancie_conclusion(regle, envs))
                        self.trace.append(regle)

        return self.solutions
```

Module `.../moteur_avec_variables/filtre.py` :

```
from .proposition_avec_variables import est_atomique, est_une_variable, tete, corps

class Filtre:
    echec = 'échec'

    def substitue(self, pattern, env):
        if est_atomique(pattern):
            if pattern in env:
                return env[pattern]
            else:
                return pattern

        pattern_subst = ()

        for sous_pattern in pattern:
            sous_pattern_subst = self.substitue(sous_pattern, env)
            pattern_subst = pattern_subst + (sous_pattern_subst,)

        return pattern_subst

    def filtre(self, datum, pattern):
        if len(pattern) == 0 and len(datum) == 0:
            return {}

        if len(pattern) == 0 or len(datum) == 0:
            return Filtre.echec

        if est_atomique(pattern):
            if datum == pattern:
                return {}
            if est_une_variable(pattern):
                return {pattern: datum}

            return Filtre.echec

        if est_atomique(datum):
            return Filtre.echec

        datum_tete = tete(datum)
        pattern_tete = tete(pattern)
        datum_reste = corps(datum)
        pattern_reste = corps(pattern)

        tete_env = self.filtre(datum_tete, pattern_tete)

        if tete_env == Filtre.echec:
            return Filtre.echec
```

```
        pattern_reste = self.substitue(pattern_reste, tete_env)
        reste_env = self.filtre(datum_reste, pattern_reste)

        if reste_env == Filtre.echec:
            return Filtre.echec

        tete_env.update(reste_env)

        return tete_env

    def pattern_match(self, datum, pattern, env=None):
        if env is not None:
            env = env.copy()
        else:
            env = {}

        pattern = self.substitue(pattern, env)
        resultat = self.filtre(datum, pattern)
        if resultat == Filtre.echec:
            return Filtre.echec

        env.update(resultat)
        return env
```

Module `.../moteur_avec_variables/unificateur.py` :

```
from .proposition_avec_variables import est_atomique, est_une_variable, tete, corps

class Unificateur:
    echec = 'échec'

    def substitue(self, pattern, env):
        if est_atomique(pattern):
            if pattern in env:
                return self.substitue(env[pattern], env)
            else:
                return pattern

        pattern_subst = ()

        for sous_pattern in pattern:
            sous_pattern_subst = self.substitue(sous_pattern, env)
            pattern_subst = pattern_subst + (sous_pattern_subst, )

        return pattern_subst

    def unifie(self, prop1, prop2):
        if len(prop1) == 0 and len(prop2) == 0:
            return {}
        if len(prop1) == 0 or len(prop2) == 0:
            return Unificateur.echec

        # Une des deux propositions est un atome => on essaie de le matcher.
        if est_atomique(prop1) or est_atomique(prop2):
            if prop1 == prop2:
                return {}
```

```
            if not est_atomique(prop1):
                prop1, prop2 = prop2, prop1

            if est_une_variable(prop1):
                if prop1 in prop2:
                    return Unificateur.echec
                else:
                    return {prop1: prop2}

            if est_une_variable(prop2):
                return {prop2: prop1}

            # Dans les autres cas, l'unification est un échec.
            return Unificateur.echec

        # Aucune des propositions n'est atomique : on unifie récursivement.
        prop1_tete = tete(prop1)
        prop2_tete = tete(prop2)
        prop1_reste = corps(prop1)
        prop2_reste = corps(prop2)
        tete_env = self.unifie(prop1_tete, prop2_tete)
        if tete_env == Unificateur.echec:
            return Unificateur.echec

        prop1_reste = self.substitue(prop1_reste, tete_env)
        prop2_reste = self.substitue(prop2_reste, tete_env)
        reste_env = self.unifie(prop1_reste, prop2_reste)
        if reste_env == Unificateur.echec:
            return Unificateur.echec

        tete_env.update(reste_env)
        return tete_env

    def pattern_match(self, prop1, prop2, env=None):
        if env is not None:
            prop1 = self.substitue(prop1, env)
            prop2 = self.substitue(prop2, env)
            env = env.copy()
        else:
            env = {}

        resultat = self.unifie(prop1, prop2)
        if resultat == Unificateur.echec:
            return Unificateur.echec

        env.update(resultat)
        return env
```

Chapitre 4

Représentation structurée des connaissances

Exercice 4.1 Modélisation
Il faut modéliser tous les concepts qui figurent dans le problème, c.à.d. les concepts suivants :

`personne, contribuable, salarié, indépendant, faible, moyen, élevé, <12, 12−18`

Le modèle contiendra également toutes les relations :

`enfant, âge, revenu, loyer, trajet`

Finalement, certains concepts font partie d'une hiérarchie :

`contribuable` $\sqsubseteq$ `personne`
`salarié` $\sqsubseteq$ `contribuable`
`indépendant` $\sqsubseteq$ `contribuable`

Exercice 4.2 Déductions
Les classes qui ont droit aux quatre déductions peuvent être modélisées par les expressions suivantes :

1) `déduction-enfant` = `contribuable` $\sqcap\exists$ `enfant`
2) `déduction-loyer` = `contribuable` $\sqcap$ `loyer.élevé` $\sqcap$ `revenu.faible`
3) `déduction-trajet` = `salarié` $\sqcap$ `trajet.élevé`
4) `déduction-pension` = `indépendant` $\sqcup$ (`salarié` $\sqcap$ `revenu.élevé`)

Exercice 4.3 Raisonnement (1)
La classe de Charles est décrite par l'expression :

(`salarié` $\sqcap$ `enfant.Jacques` $\sqcap$ `trajet.élevé` $\sqcap$ `loyer.élevé` $\sqcap$ `revenu.faible`)

Charles a droit à trois déductions : `déduction-enfant, déduction-trajet,` et `déduction-loyer`
On trouve `déduction-enfant` en utilisant de la T-box d'abord le fait qu'un salarié est un contribuable :

`salarié` $\sqsubseteq$ `contribuable`

pour construire une réécriture de l'expression qui décrit Charles, et ensuite la définition de la classe

`déduction-enfant` = `contribuable` $\sqcap\exists$ `enfant`

pour réécrire le but.
La procédure de subsumption sera alors appliquée pour montrer :

(`contribuable` $\sqcap$ `enfant.Jacques` $\sqcap$ `trajet.élevé` $\sqcap$ `loyer.élevé` $\sqcap$ `revenu.faible`) $\sqsubseteq$ (`contribuable` $\sqcap\exists$ `enfant`)

La même réécriture pourra être utilisée pour montrer la subsumption avec `déduction-trajet`, et la subsumption avec `déduction-loyer` peut être trouvée sans réécriture.

Exercice 4.4 Raisonnement (2)
Pour la version a), on peut caractériser la classe de ceux qui perdent par :

$$\texttt{contribuable} \sqcap \texttt{loyer.élevé} \sqcap \texttt{revenu.faible} \sqcap \neg\exists\ \texttt{enfant.âge.12-18}$$

et de ceux qui gagnent :

$$\texttt{contribuable} \sqcap \texttt{loyer.élevé} \sqcap \neg\ \texttt{revenu.faible} \sqcap\exists\ \texttt{enfant.âge.12-18}$$

Pour la version b), on peut caractériser la classe de ceux qui perdent par :

$$\texttt{contribuable} \sqcap \texttt{loyer.élevé} \sqcap \texttt{revenu.faible} \sqcap <\texttt{2.enfant}$$

et ceux qui gagnent par :

$$\texttt{contribuable} \sqcap \texttt{loyer.élevé} \sqcap \neg\ \texttt{revenu.faible} \sqcap >\texttt{1.enfant}$$

Pour la version a), la logique $\mathcal{AL}$ est suffisante. Pour la version b), la possibilité d'introduire des restrictions de nombre est nécessaire, ce qui rend le raisonnement beaucoup plus complexe.

Chapitre 5

Raisonnement basé sur des règles et systèmes experts

Exercice 5.1 Comparaison du chaînage avant et arrière

1) En chaînage avant, il suffit de lancer le système pour trouver tous les faits qui peuvent être établis d'après la base des faits initiaux et les règles. Ensuite on examine le contenu de la base de faits pour voir les faits qui correspondent à notre requête. On peut envisager que l'examen du contenu de la base de faits se fasse au moyen d'un mécanisme de *pattern matching*. Pour le chaînage arrière, il suffit de fournir au système le but à satisfaire :
   - Requête 1 : `Bonifier(Bourgogne,?Année-Vin,2007)`
   - Requête 2 : `Eliminer(?Vin,?Année-Vin)`
2) `Bonifier(Bourgogne,1995,2007)` est la requête à laquelle nous nous intéressons. Voici les règles qui sont appliquées :

```
REGLE 4)
BUT: Bonifier(Bourgogne,1995,2007)
consequence−regle: Bonifier(Bourgogne,?Année−Vin,?Année−Vin+10)
−>ECHEC

REGLE 6)
BUT: Bonifier(Bourgogne,1995,2007)
SOUSBUTS:
− 2006 > 1900 −> SUCCES
− 2006 < 2020 −> SUCCES
− NOT Déclasser(Bourgogne,1995,2007)
  REGLE 2)
  BUT: Déclasser(Bourgogne,1995,2007)
  SOUSBUTS:
  − 2007 − 1995 > 20−>ECHEC
  −>ECHEC
  REGLE 5)
  BUT: Déclasser(Bourgogne,1995,2007)
  SOUSBUTS:
  − 2006 < 2020 −> SUCCES
  − 2006 > 1900 −> SUCCES
  − Déclasser(Bourgogne, 1995, 2006)
    REGLE 5)
    BUT: Déclasser(Bourgogne, 1995, 2006)
    SOUSBUTS:
    − 2005< 2020 −> SUCCES
    − 2005> 1900 −> SUCCES
    − Déclasser(Bourgogne, 1995, 2005) ... jusqu'à 1900 ... −> ECHEC
    −> ECHEC
  −>ECHEC
  −>SUCCESS

− Bonifier(Bourgogne,1995,2006)
  REGLE 4)
  BUT:Bonifier(Bourgogne,1995,2006)
```

```
SOUSBUTS:
− Stock−Vin(Bourgogne,1995) −> SUCCES
− 2006 − 1995 > 10 −> ECHEC
−> ECHEC
REGLE 6)
BUT: Bonifier(Bourgogne,1995,2006)
SOUSBUTS:
− 2005 > 1900 −> SUCCES
− 2005 < 2020 −> SUCCES
− NOT Déclasser(Bourgogne,1995,2006)
  REGLE 2)
  BUT: Déclasser(Bourgogne,1995,2006)
   ...
  −> ECHEC
  −> SUCCES

− Bonifier(Bourgogne,1995,2005)
  REGLE 4)
  BUT: Bonifier(Bourgogne,1995,2005)
  SOUSBUTS:
  − Stock−Vin(Bourgogne,1995) −> SUCCES
  − 2005 − 1995 = 10 −> SUCCES
  −> SUCCES
−> SUCCES

−> SUCCES
```

3) Si on enclenche le système en chaînage avant, le mécanisme d'inférence va essayer de déduire tous les faits possibles en unifiant l'ensemble des règles avec les faits de la base (faits initiaux et faits inférés). Dans le cas de notre exemple, le système va surcharger la base de faits avec des informations inutiles, en déduisant pour chaque année tous les vins qui seront bons cette année-là (et ce jusqu'à 2020). Or, si un vin est bon une certaine année, il le restera tant qu'il n'est pas déclassé. Déduire explicitement qu'il sera bon toutes les années suivantes est inutile. Imaginons que le nombre de bons vins soit important, le nombre de faits inférés risque d'être très grand et va considérablement ralentir le fonctionnement de notre système expert. Ces séries d'inférences sont d'autant plus inutiles qu'elles n'ont rien à voir avec la requête qui nous intéresse, c.à.d. quels sont les vins à éliminer.
4) Les règles 1), 2), 5), 7) et 8) permettent de déduire tous les vins à déclasser et à éliminer en utilisant le chaînage avant. Cela peut-être constaté en partant du fait `Eliminer` et en regardant quelles sont les seules règles qui peuvent être successeurs de ce type de faits dans un processus de chaînage arrière.
5) Dans le cas du chaînage arrière, l'absence de cette condition va faire cycler le système à l'infini pour certaines requêtes : par exemple, si on désire savoir si le Bourgogne 1983 est un bon vin, la règle 4) ne sera jamais satisfaite car il n'y a pas de Bourgogne 1983 en stock et le système va donc récursivement essayer d'appliquer la règle 6) puis la règle 4) sans jamais s'arrêter.
6) Si toutes les règles sont appliquées en chaînage arrière, les vins déclassés sont à déduire à nouveau à chaque fois que l'on veut savoir si un vin

particulier est bon. De plus, on va essayer de satisfaire à plusieurs reprises les mêmes sous-buts, par exemple : `Stock-Vin(?Vin,?Année-Vin)` qui est impliqué dans les règles 1), 2) 3) et 4). Il est clair que si la satisfaction d'un sous-but est onéreuse, on aimerait éviter de le recalculer plusieurs fois.

7) Ce qu'on veut montrer, c'est qu'un processus de décision est souvent composé de parties en chaînage avant et de parties en chaînage arrière. Le choix du mécanisme d'inférence est souvent lié à la nature des requêtes (certaines requêtes se prêtent mieux au chaînage avant et d'autres au chaînage arrière). En principe, si les règles ont été conçues avec suffisamment de soin pour éviter les problèmes de cycles, on peut indifféremment utiliser l'un ou l'autre type de chaînage pour satisfaire une requête. Cela ne veut pas dire que les coûts (en capacité mémoire et en temps) soient les mêmes. C'est pour cela qu'en pratique, les systèmes experts sont souvent hybrides.

Exercice 5.2 Programmation du chaînage arrière

Module `.../moteur_chainage_arriere/noeud.py` :

```
from moteur_avec_variables.proposition_avec_variables import *

class Noeud:
    def __init__(self, but, sous_but_courant, sous_buts_a_tester, profondeur):
        self.but = but
        self.sous_but_courant = sous_but_courant
        self.sous_buts_a_tester = sous_buts_a_tester
        self.profondeur = profondeur

    def est_terminal(self):
        return len(self.sous_but_courant) == 0 and len(self.sous_buts_a_tester) == 0

    def est_solution(self):
        variables = lister_variables(self.but)
        return self.est_terminal() and len(variables) == 0

    def successeur(self, env, nouveaux_sous_buts, unificateur):
        # Les sous-buts àexplorer sont composés par les sous-buts
        # encore ouverts et les nouveaux sous-buts (cas d'une règle).
        sous_buts = nouveaux_sous_buts[:]
        sous_buts.extend(self.sous_buts_a_tester)

        if len(sous_buts) > 0:
            premier_sous_but = sous_buts[0]
        else:
            premier_sous_but = ()

        if len(sous_buts) > 1:
            reste_sous_buts = sous_buts[1:]
        else:
            reste_sous_buts = []

        successeur = Noeud(
                        unificateur.substitue(self.but, env),
```

```
                    unificateur.substitue(premier_sous_but, env),
                    [unificateur.substitue(sous_but, env) for sous_but in
                     reste_sous_buts],
                    self.profondeur + 1
                   )

        return successeur

    def description_standardisee(noeud):
        sous_buts = [noeud.sous_but_courant] if len(noeud.sous_but_courant) > 0 else []
        sous_buts.extend(noeud.sous_buts_a_tester)
        sous_buts = sorted(sous_buts, key=lambda prop: prop)
        but_et_sous_buts = [noeud.but] + sous_buts

        return but_et_sous_buts

    def __repr__(self):
        return '<{},{},{},{}>'.format(self.but,
                                      self.sous_but_courant,
                                      self.sous_buts_a_tester,
                                      self.profondeur)
```

Module `.../moteur_chainage_arriere/chainage_arriere.py` :

```
from moteur_sans_variables.chainage import Chainage
from .noeud import Noeud
from .noeuds_testes import NoeudsTestes

class ChainageArriere(Chainage):
    def __init__(self, connaissances, unificateur):
        self.connaissances = connaissances
        self.unificateur = unificateur

    def successeurs(self, noeud):
        nouveaux_noeuds = []

        if noeud.est_terminal():
            return nouveaux_noeuds

        # On se limite aux faits et aux règles intéressants à examiner.
        # En effet, si le sous-but examiné commence par 'grand-pere',
        # alors il est inutile de considérer des noeuds commençant
        # par 'oncle' par exemple.
        regles_interessantes = self.connaissances.choisir_regles_interessantes(noeud.
        sous_but_courant, self.unificateur)
        faits_interessants = self.connaissances.choisir_faits_interessants(noeud.
        sous_but_courant)

        # On parcourt les règles intéressantes.
        for regle in regles_interessantes:
            # Tentative d'unification entre le sous-but sélectionné
            # et la consequence de la règle.
            env = self.unificateur.unifie(noeud.sous_but_courant,
                                          regle.conclusion)
            if env != self.unificateur.echec:
                nouveau_noeud = noeud.successeur(env,
                                                 regle.conditions,
                                                 self.unificateur)
```

```
                nouveaux_noeuds.append(nouveau_noeud)

        # On parcourt les faits intéressants.
        for fait in faits_interessants :
            # Tentative d'unification entre le sous-but sélectionné et le fait.
            env = self.unificateur.unifie(noeud.sous_but_courant, fait)
            if env != self.unificateur.echec:
                nouveau_noeud = noeud.successeur(env, [], self.unificateur)
                nouveaux_noeuds.append(nouveau_noeud)
        # Retourne les nouveaux noeuds ainsi trouvés.
        return nouveaux_noeuds

    def backchain(self, noeud_depart):
        # Liste des noeuds à tester.
        queue = [noeud_depart]
        # Liste des noeuds déjà testés.
        noeuds_testes = NoeudsTestes()
        # Liste des solutions.
        self.solutions = set()

        # Tant qu'il y a des noeuds à tester dans la liste,
        while len(queue) > 0:
            # on sélectionne le noeud suivant.
            noeud = queue.pop(0)
            self.trace.append(noeud)
            # Si le noeud n'appartient pas encore à la liste des noeuds
            # déjà testés, on l'y ajoute pour éviter les cycles.
            if noeud not in noeuds_testes:
                noeuds_testes.ajoute(noeud)
                # Si le noeud est une solution, on l'ajoute à celles qu'on a
                # déjà trouvées.
                if noeud.est_solution():
                    self.solutions.add(noeud.but)
                else:
                    # On obtient des noeuds supplémentaires par chaînage arrière.
                    successeurs = self.successeurs(noeud)
                    successeurs.extend(queue)
                    queue = successeurs

        # Retourne la liste des solutions au problème.
        return self.solutions

    def chaine(self, pattern):
        # Retourne les solutions par chaînage arrière.
        noeud_depart = Noeud(pattern, pattern, [], 0)
        solutions = self.backchain(noeud_depart)

        return solutions
```

Chapitre 6

Traitement de l'information incertaine

Exercice 6.1 Réseaux de Bayes

6.1.1 Raisonnement probabiliste

Question 1. Informellement, la probabilité que vous soyez malade est toujours proportionnelle à la probabilité de cette maladie. Ainsi, sur 10'000 personnes, l'une d'entre elle sera effectivement malade et sera diagnostiquée comme telle par le test avec 99 % de chance. Cependant, pour les 9'999 personnes restantes, le test se trompera dans 1 % des cas. Environ 100 personnes seront donc positives au test bien que n'étant pas malade. Globalement, vous avez donc seulement 1 % de risque d'être effectivement malade si le test est positif.

Plus formellement, soit T la variable représentant le résultat du test, et M la variable représentant la maladie. Par définition des taux de faux négatifs et de faux positifs, on a $P(T = 1|M = 1) = 1-P(T = 0|M = 1) = 1-0.01 = 0.99$ et $P(T = 0|M = 0) = 1 - P(T = 1|M = 0) = 1 - 0.01 = 0.99$. Cependant, $P(M = 1) = 0.0001$ puisque la maladie ne frappe qu'une personne sur 10'000. La probabilité qui nous intéresse est $P(M = 1|T = 1)$, puisque nous souhaitons déterminer le risque que vous soyez effectivement malade sachant que votre test a été positif. Or,

$$\begin{aligned} P(M = 1|T = 1) &= \frac{P(T = 1|M = 1) \cdot P(M = 1)}{P(T = 1|M = 1) \cdot P(M = 1) + P(T = 1|M = 0) \cdot P(M = 0)} \\ &= \frac{0.99 \cdot 0.0001}{0.99 \cdot 0.0001 + 0.01 \cdot 0.9999} \\ &= 0.009804 \end{aligned}$$

Comme ce chiffre demeure très faible, vous ne devez pas forcément vous inquiéter. Bien sûr, cette conclusion est due à la faible fréquence de la maladie dans la population.

6.1.2 Causalité

Modélisation du problème

Question 1. Le réseau bayésien (fig. 1) possède les nœuds :

- I=Route-Gelée
- H=Accident-Holmes
- W=Accident-Watson
- S=Professeur-Sauvé

et les arcs :

- $I \to H$ et $I \to W$, car l'état des routes influence les accidents de Holmes et de Watson.
- $W \to S$ et $H \to S$, car les accidents de Holmes et de Watson influencent la résolution de l'affaire.

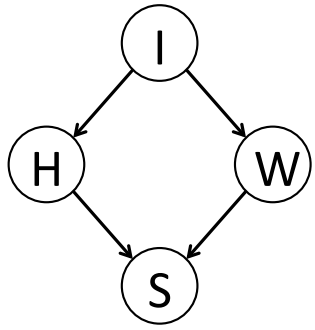


Fig. 1 Le réseau bayésien avec les noeuds : I=Route-Gelée, H=Accident-Holmes, W=Accident-Watson, S=Professeur-Sauvé

Question 2 (Inférence déductive).

- Pour calculer $P(H)$, il nous faut connaître $P(H|I)$:

$$P(H = h) = P(H = h|I = 1)P(I = 1) + P(H = h|I = 0)P(I = 0)$$

Il en va de même de la probabilité $P(W)$, pour laquelle il faut connaître $P(W|I)$:

$$P(W = w) = P(W = w|I = 1)P(I = 1) + P(W = w|I = 0)P(I = 0)$$

- Pour calculer $P(S)$, il nous faut en plus $P(S|H, W)$:

$$P(S = s) = \sum_{i,h,w \in \{1,0\}} P(S = s|H = h, W = w)P(H = h|I = i)P(W = w|I = i)P(I = i)$$

Question 3 (Inférence abductive). La probabilité que nous cherchons est $P(I|W)$. Pour la calculer, il nous faut $P(W|I)$. Par la règle de Bayes, nous avons :

$$P(I = i|W = w) = \frac{P(W = w|I = i)P(I = i)}{P(W = w)}$$

Question 4 (Déduction et abduction) :

- On cherche $P(H|W)$. Il nous faut donc $P(W|I)$ et $P(H|I)$:

$$P(H = h|W = w) = \sum_{i \in \{1,0\}} P(H = h|I = i)P(I = i|W = w)$$

- On cherche $P(H|W, S)$. Il nous faut donc en plus $P(S|H, W)$:

$$P(H = h|W = w, S = s) = \alpha P(S = s|H = h, W = w)P(H = h|W = w)$$

On détermine $\alpha = \frac{1}{P(S=s|W=w)}$ de sorte que :

$$\sum_{h\in\{1,0\}} P(H=h|W=w,S=s)=1$$

- On cherche $P(H|I,W,S)$:

$$P(H=h|I=i,W=w,S=s)=\alpha P(S=s|H=h,W=w)P(H=h|I=i)$$

On détermine $\alpha = \frac{1}{P(S=s|I=i,W=w)}$ de sorte que :

$$\sum_{h\in\{1,0\}} P(H=h|I=i,W=w,S=s)=1$$

Calcul probabiliste

Question 5.

$P(H\|I)$	$I=1$	$I=0$	$P(W\|I)$	$I=1$	$I=0$
$H=1$	0.9	0.1	$W=1$	0.7	0.5
$H=0$	0.1	0.9	$W=0$	0.3	0.5

Question 6.

$$\begin{aligned}P(H=1) &= P(H=1|I=1)P(I=1)+P(H=1|I=0)P(I=0)\\ &= 0.9\cdot 0.7+0.1\cdot 0.3=0.66\\ P(W=1) &= P(W=1|I=1)P(I=1)+P(W=1|I=0)P(I=0)\\ &= 0.7\cdot 0.7+0.5\cdot 0.3=0.64\end{aligned}$$

Question 7.

$$\begin{aligned}P(I=1|W=1) &= \frac{P(W=1|I=1)\cdot P(I=1)}{P(W=1)}\\ &= \frac{0.7\cdot 0.7}{0.64}\cong 0.766\end{aligned}$$

Il en résulte que $P(I=0|W=1)\cong 0.234$.

Question 8 (Dépendance).

$$\begin{aligned}P(H=1|W=1) &= P(H=1|I=1)\cdot P(I=1|W=1)\\ &\quad + P(H=1|I=0)\cdot P(I=0|W=1)\\ &= 0.9\cdot 0.766+0.1\cdot 0.234\cong 0.713\end{aligned}$$

On peut constater que la probabilité d'accident de Holmes a augmenté suite à la connaissance de l'accident de Watson, car cet accident laisse supposer que la route peut être gelée. H et W sont deux événements dépendants.

Question 9 (Indépendance conditionnelle). Si l'on sait que la route n'est pas gelée, $I = 0$ et l'accident de Watson n'a plus d'influence sur la probabilité d'accident de Holmes. En effet, l'accident de Watson n'a pas d'effet sur la probabilité que la route soit gelée puisque celle-ci est connue avec certitude. Ainsi, on a : $P(H = 1|I = 0, W = 1) = P(H = 1|I = 0) = 0.1$, sachant que I, H et W sont deux événements indépendants.

Question 10.

$P(S\|W,H)$	$H=1$, $W=1$	$H=1$, $W=0$	$H=0$, $W=1$	$H=0$, $W=0$
$S=1$	0.1	0.2	0.8	1
$S=0$	0.9	0.8	0.2	0

Question 11 (Causes multiples).

$$\begin{aligned}P(S=s) = \sum_{i,h,w\in\{1,0\}} & P(S=s|H=h,W=w)P(H=h|I=i) \\ & P(W=w|I=i)P(I=i) \\ &= 0.1\cdot(0.9\cdot 0.7\cdot 0.7 + 0.1\cdot 0.5\cdot 0.3) \\ &+ 0.2\cdot(0.9\cdot 0.3\cdot 0.7 + 0.1\cdot 0.5\cdot 0.3) \\ &+ 0.8\cdot(0.1\cdot 0.7\cdot 0.7 + 0.9\cdot 0.5\cdot 0.3) \\ &+ 1\cdot(0.1\cdot 0.3\cdot 0.7 + 0.9\cdot 0.5\cdot 0.3) \\ &= 0.3896\end{aligned}$$

Question 12.

$$\begin{aligned}P(H=1|W=1,S=1) &= \alpha P(S=1|H=1,W=1)P(H=1|W=1) \\ &= \alpha\cdot 0.1\cdot 0.713 = \alpha\cdot 0.0713 \\ P(H=0|W=1,S=1) &= \alpha P(S=1|H=0,W=1)P(H=0|W=1) \\ &= \alpha\cdot 0.8\cdot 0.287 = \alpha\cdot 0.23\end{aligned}$$

Il faut que $\alpha\cdot(0.0713+0.23) = 1$, donc on trouve $\alpha \cong 3.319$, $P(H=1|W=1,S=1) \cong 0.237$ et $P(H=0|W=1,S=1) \cong 0.763$.

Question 13 (Abduction avec plusieurs conséquences).

$$\begin{aligned}P(I=1|W=1,S=1) &= \sum_{h\in\{1,0\}} P(I=1|H=h,W=1)P(H=h|W=1,S=1) \\ &= \sum_{h\in\{1,0\}} \frac{P(H=h|I=1)P(W=1|I=1)P(I=1)}{P(H=h|W=1)P(W=1)} \\ &\qquad P(H=h|W=1,S=1) \\ &= \frac{0.9\cdot 0.7\cdot 0.7}{0.713\cdot 0.64}\cdot 0.237 + \frac{0.1\cdot 0.7\cdot 0.7}{0.287\cdot 0.64}\cdot 0.763 \\ &= 0.229 + 0.204 = 0.433\end{aligned}$$

Question 14.

$$\begin{aligned} P(H=1|I=0,W=1,S=1) &= \alpha P(S=1|H=1,W=1)P(H=1|I=0) \\ &= \alpha \cdot 0.1 \cdot 0.1 = \alpha \cdot 0.01 \\ P(H=0|I=0,W=1,S=1) &= \alpha P(S=1|H=0,W=1)P(H=0|I=0) \\ &= \alpha \cdot 0.8 \cdot 0.9 = \alpha \cdot 0.72 \end{aligned}$$

Il faut que $\alpha \cdot (0.01 + 0.72) = 1$, donc on trouve $\alpha = 1.370$ et $P(H = 1|I = 0, W = 1, S = 1) = 0.014$ et $P(H = 0|I = 0, W = 1, S = 1) = 0.986$.

Question 15 (Dépendance conditionnelle).

$$\begin{aligned} P(H=1|I=0,S=1) &= \alpha P(S=1|H=1,I=0)P(H=1|I=0) \\ &= \alpha \cdot \sum_{w\in\{1,0\}} P(S=1|H=1,W=w) \cdot P(W=w|I=0) \cdot P(H=1|I=0) \\ &= \alpha \cdot (0.1 \cdot 0.5 + 0.2 \cdot 0.5) \cdot 0.1 = \alpha \cdot 0.015 \\ P(H=0|I=0,S=1) &= \alpha P(S=1|H=0,I=0)P(H=0|I=0) \\ &= \alpha \cdot \sum_{w\in\{1,0\}} P(S=1|H=0,W=w) \cdot P(W=w|I=0) \cdot P(H=0|I=0) \\ &= \alpha \cdot (0.8 \cdot 0.5 + 1 \cdot 0.5) \cdot 0.9 = \alpha \cdot 0.81 \end{aligned}$$

Il faut que $\alpha \cdot (0.015 + 0.81) = 1$, donc on trouve $\alpha = 1.21$ et $P(H = 1|I = 0, S = 1) = 0.02$ et $P(H = 0|I = 0, S = 1) = 0.98$. Donc, si en plus on sait que Watson a eu un accident, la probabilité que Holmes en ait eu un aussi redescend un peu. Ainsi, H et W sont des événements dépendants sachant que S et I.

Question 16. On ajoute le noeud V=Vieux-Pneus-Watson et l'arc $V \to W$ (fig. 2).

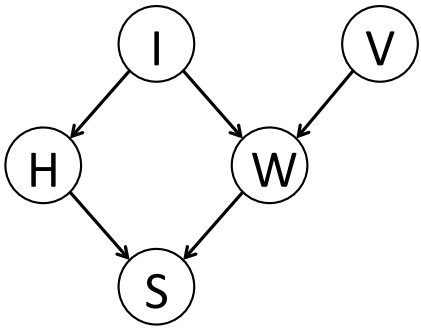


Fig. 2 Le nouveau réseau bayésien avec les noeuds : I=Route-Gelée, H=Accident-Holmes, W=Accident-Watson, S=Professeur-Sauvé, V=Vieux-Pneus-Watson

Question 17. On cherche $P(V|I,W)$. Il nous faut donc $P(W|I,V)$:

$$P(V=v|I=i,W=w) = \alpha P(W=w|I=i,V=v)P(V=v)$$

On détermine $\alpha = \frac{1}{P(W=w|I=i)}$ de sorte que :

$$\sum_{v\in\{1,0\}} P(V=v|I=i,W=w) = 1$$

Exercice 6.2 Facteurs de certitude

Module `.../moteur_avec_variables_fc/facteurs_certitude.py` :

```
def fc_ou(fc1, fc2):
    if fc1 > 0.0 and fc2 > 0.0:
        return fc1 + fc2 - (fc1 * fc2)
    elif fc1 < 0.0 and fc2 < 0.0:
        return fc1 + fc2 + (fc1 * fc2)
    else:
        return (fc1 + fc2) / (1.0 - min(abs(fc1), abs(fc2)))

def fc_et(fc1, fc2):
    return min(fc1, fc2)
```

Module `.../moteur_avec_variables_fc/regle_avec_variables_fc.py` :

```
from .facteurs_certitude import fc_et

class RegleAvecVariables_FC:
    def __init__(self, conditions, conclusion, fc=1.0):
        self.conditions = conditions
        self.conclusion = conclusion
        self.fc = fc

    def depend_de(self, fait, methode):
        envs = {}

        for condition in self.conditions:
            # Si au moins une des conditions retourne un environnement,
            # nous savons que la proposition satisfait une des conditions.
            env = methode.pattern_match(fait, condition, {})
            if env != methode.echec:
                envs[condition] = env

        return envs

    def satisfaite_par(self, faits, cond, env, env_fc, methode):
        envs_et_fcs = [(env, env_fc)]
        conditions_a_tester = [cond1 for cond1 in self.conditions if cond1 != cond]
```

```
        for cond1 in conditions_a_tester :
            nouveaux_envs_et_fcs = []

            for fait , fait_fc in faits :
                for env1, env_fc1 in envs_et_fcs :
                    env1 = methode.pattern_match(fait, cond1, env1)
                    if env1 != methode.echec:
                        fc_mix = fc_et(env_fc1, fait_fc )
                        nouveaux_envs_et_fcs.append((env1, fc_mix))

            if len(nouveaux_envs_et_fcs) == 0:
                return []

            envs_et_fcs = nouveaux_envs_et_fcs

        return envs_et_fcs

    def __repr__( self ):
        return '{} => {}, {}'.format(str(self.conditions),
                                     str( self .conclusion ),
                                     str( self . fc ))
```

Module `.../moteur_avec_variables_fc/connaissance_fc.py` :

```
from .facteurs_certitude import fc_ou
from moteur_avec_variables_fc. regle_avec_variables_fc import RegleAvecVariables_FC

class BaseConnaissances_FC:
    def __init__ ( self ):
        self . faits = {}
        self . regles = []

    def ajoute_un_fait ( self , fait ):
        if len( fait ) == 2:
            prop, fc = fait
        elif len( fait ) == 1:
            prop, fc = fait [0], 1.0
        else:
            raise ValueError("Fait mal formé: " + str( fait ))

        fc_deja_present = self . faits .get( prop)
        if fc_deja_present is not None:
            nouveau_fc = fc_ou(fc, fc_deja_present )
            self . faits [prop] = nouveau_fc
        else:
            self . faits [prop] = fc

    def ajoute_faits ( self , faits ):
        for fait in faits :
            self . ajoute_un_fait ( fait )

    def ajoute_une_regle( self , description ):
        if len(description) == 2:
            regle = RegleAvecVariables_FC(description[0], description [1])
        elif len(description) == 3:
            regle = RegleAvecVariables_FC(description[0], description [1], description [2])
        else:
```

```
            raise ValueError("Une règle doit avoir deux ou trois éléments. Reçu " +
                             str(description))

        self.regles.append(regle)

    def ajoute_regles(self, descriptions):
        for description in descriptions:
            self.ajoute_une_regle(description)
```

Module `.../moteur_avec_variables_fc/chainage_avant_avec_variables_fc.py` :

```
from moteur_sans_variables.chainage import Chainage
from moteur_avec_variables.filtre import Filtre

class ChainageAvantAvecVariables_FC(Chainage):
    def __init__(self, connaissances, methode=None):
        Chainage.__init__(self, connaissances)

        if methode is None:
            self.methode = Filtre()
        else:
            self.methode = methode

    def instancie_conclusion(self, regle, envs_et_fcs):
        nouveaux_faits = []

        for env, env_fc in envs_et_fcs:
            prop = self.methode.substitue(regle.conclusion, env)
            nouveau_fait = (prop, max(0.0, env_fc) * regle.fc)
            nouveaux_faits.append(nouveau_fait)

        return nouveaux_faits

    def chaine(self):
        queue = [p for p in self.connaissances.faits.items()]
        self.reinitialise()

        while len(queue) > 0:
            fait, fait_fc = queue.pop(0)

            if (fait, fait_fc) not in self.solutions:
                self.trace.append((fait, fait_fc))
                self.solutions.append((fait, fait_fc))

                # Si le facteur de certitude du fait est supérieur à 0.0
                if fait_fc > 0.0:

                    # on vérifie si des règles sont déclenchées par le nouveau fait.
                    for regle in self.connaissances.regles:
                        envs = regle.depend_de(fait, self.methode)

                        for cond, env in envs.items():
                            # On remplace l'environnement par ceux qui satisfont toutes
                            # les conditions de la règle et pas seulement la première
                            # condition.
                            envs1 = regle.satisfaite_par(self.solutions, cond, env,
                                                         fait_fc, self.methode)
```

```
                # On ajoute à la queue la conclusion de la règle instanciée
                # selon chaque environnement possible.
                if len(envs1) > 0:
                    queue.extend(self.instancie_conclusion(regle, envs1))
                    self.trace.append(regle)

        return self.solutions
```

Chapitre 7

Résolution de problèmes par recherche

Exercice 7.1 Codage des classes de bases

Module `.../moteurs_recherche/element.py` :

```
class Element:
    def __init__(self, nom=''):
        self.nom = nom

    def distance(self, element):
        return 1

    def __eq__(self, autre):
        return self.nom == autre.nom

    def __hash__(self):
        return hash(str(self))

    def __repr__(self):
        return '{}'.format(self.nom)
```

Module `.../moteurs_recherche/ville.py` :

```
from math import sqrt
from .element import Element

class Ville(Element):
    def __init__(self, x, y, nom=''):
        Element.__init__(self, nom)
        self.x = x
        self.y = y

    def distance(self, ville):
        return sqrt((self.x-ville.x)**2 + (self.y-ville.y)**2)

    def __eq__(self, autre):
        if not isinstance(autre, Ville):
            return False
        return self.x == autre.x and self.y == autre.y and self.nom == autre.nom

    def __hash__(self):
        return hash(str(self))

    def __repr__(self):
        return '{}({}, {})'.format(self.nom, self.x, self.y)
```

Module `.../moteurs_recherche/espace.py` :

```
from copy import copy

class Espace:
    def __init__(self, elements=None, arcs=None):
```

```
        self.elements = []
        if elements is not None:
            self.elements = sorted(self.elements, key=lambda e: e.nom)

        self.arcs = []
        if arcs is not None:
            self.ajoute_arcs(arcs)

    def ajoute_arcs(self, arcs):
        for element_1, element_2 in arcs:
            if not element_1 in self.elements:
                self.elements.append(element_1)
            if not element_2 in self.elements:
                self.elements.append(element_2)

            if not (element_1, element_2) in self.arcs:
                self.arcs.append((element_1, element_2))

        self.elements = sorted(self.elements, key=lambda e: e.nom)

    def trouve_voisins(self, element):
        voisins = []

        for element_1, element_2 in self.arcs:
            if element_1 == element:
                voisins.append(element_2)
            if element_2 == element:
                voisins.append(element_1)

        voisins = sorted(voisins, key=lambda v: v.nom)

        return voisins

    def __repr__(self):
        rep = ''
        for element in self.elements:
            rep += '{}, '.format(element)
            rep += 'avec voisins: '
            voisins = self.trouve_voisins(element)
            rep += ', '.join(map(str, voisins))
            rep +='\n'

        return rep
```

Module `.../moteurs_recherche/noeud.py` :

```
from math import sqrt

class Noeud:
    def __init__(self, element, parent=None, cout=0, cout_f=0):
        self.element = element
        self.parent = parent
        self.cout = cout
        self.cout_f = cout_f

    def __repr__(self):
        rep = '<{}, {}, {}>'.format(self.element,
                                    round(self.cout),
```

```
                                round(self.cout_f))
        return rep
```

Module `.../moteurs_recherche/recherche.py` :

```
from .noeud import Noeud

class Recherche:
    echec = 'échec'

    def __init__(self, espace, optimisee=False):
        self.espace = espace
        self.optimisee = optimisee

    def recherche(self, depart, but):
        # L'heuristique à utiliser (utile uniquement pour A\*).
        self.h = lambda e: e.distance(but)

        noeud_depart = Noeud(depart, None, 0, self.h(depart))
        noeud_but = Noeud(but)

        return self.recherche_chemin(noeud_depart, noeud_but)

    def recherche_chemin(self, noeud_depart, noeud_but):
        queue = [noeud_depart]
        iterations = 0
        trace = {}

        while len(queue) > 0:
            noeud = queue.pop(0)

            if self.optimisee and self.detecte_cycle(trace, noeud):
                continue

            iterations += 1
            print('Itération {}: {}'.format(iterations, noeud))

            if noeud.element == noeud_but.element:
                return self.trouve_chemin(noeud)
            else:
                trace[noeud.element] = noeud
                successeurs = self.trouve_successeurs(noeud)
                queue = self.ajoute_successeurs(queue, successeurs)

        return Recherche.echec

    def trouve_chemin(self, noeud):
        chemin = []
        while noeud is not None:
            chemin.insert(0, noeud.element)
            noeud = noeud.parent

        return chemin

    def detecte_cycle(self, trace, noeud):
        return noeud.element in trace

    def trouve_successeurs(self, noeud):
```

```
        successeurs = []
        voisins = self.espace.trouve_voisins(noeud.element)
        for voisin in voisins:

            # Évite les cycles à deux éléments a - b - a - b ...
            if noeud.parent is not None and noeud.parent.element == voisin:
                continue

            # Coût jusqu'au noeud successeur = coût jusqu'au noeud courant +
            # distance entre les deux noeuds.
            distance = noeud.element.distance(voisin)
            cout = noeud.cout + distance

            # Coût estimé = coût jusqu'au noeud successeur + coût estimé entre
            # le noeud successeur et le but.
            cout_f = cout + self.h(voisin)

            successeur = Noeud(voisin, noeud, cout, cout_f)
            successeurs.append(successeur)

        return successeurs

    def ajoute_successeurs(self, queue, successeurs):
        # Nous retournons une liste vide pour éviter de déclencher une exception,
        # mais cette méthode doit être surchargée dans les sous-classes.
        return []
```

Module `.../moteurs_recherche/bfs.py` :

```
from .recherche import Recherche
from .noeud import Noeud

class RechercheBFS(Recherche):
    def ajoute_successeurs(self, queue, successeurs):
        return queue + successeurs
```

Module `.../moteurs_recherche/dfs.py` :

```
from .recherche import Recherche
from .noeud import Noeud

class RechercheDFS(Recherche):
    def ajoute_successeurs(self, queue, successeurs):
        return successeurs + queue
```

Module `.../moteurs_recherche/astar.py` :

```
from moteurs_recherche.recherche import Recherche
from moteurs_recherche.noeud import Noeud

class RechercheAStar(Recherche):
    def detecte_cycle(self, trace, noeud):
        if noeud.element not in trace:
            return False

        autre = trace[noeud.element]
```

```
        return autre.cout_f <= noeud.cout_f

    def ajoute_successeurs( self ,  queue, successeurs ):
        queue = queue + successeurs
        queue = sorted(queue, key=lambda n: n.cout_f)

        return queue
```

Chapitre 8

Satisfaction de contraintes

Exercice 8.1 Consistance des nœuds et des arcs

Module `.../moteur_psc/contrainte.py` :

```
class Contrainte:
    def __init__(self, variables):
        self.variables = tuple(variables)

    def dimension(self):
        return len(self.variables)

    def est_valide(self):
        return False

    def __repr__(self):
        return 'Contrainte: {}'.format(self.variables)

    def __eq__(self, that):
        return self.variables == that.variables

    def __hash__(self):
        return sum([v.__hash__ for v in self.variables])

class ContrainteUnaire(Contrainte):
    def __init__(self, var, op):
        Contrainte.__init__(self, (var,))
        self.op = op

    def est_valide(self, val):
        return self.op(val)

class ContrainteBinaire(Contrainte):
    def __init__(self, var1, var2, op):
        Contrainte.__init__(self, (var1, var2))
        self.op = op

    def est_valide(self, var, val):
        var1, var2 = self.variables

        if var1 == var:
            return self.op(val, var2.val)
        elif var2 == var:
            return self.op(var1.val, val)
        else:
            # var n'est pas une des variables de la contrainte.
            raise ValueError('Mauvaise variable: ' + var.nom + '. ' +
                             'On attendrait ' + var1 + ' ou ' + var2)

    def est_possible(self, var):
        if var not in self.variables:
            # var ne fait pas partie des variables de la contrainte
            var1, var2 = self.variables
```

```
                raise ValueError('Mauvaise variable: ' + var.nom + '. ' +
                                 'On attendrait ' + var1 + ' ou ' + var2)

            for val in var.domaine:
                if self.est_valide(var, val):
                    # Il suffit d'une valeur valide.
                    return True

            # Aucune valeur val du domaine n'a retourné True pour
            # est_valide(var, val).
            return False

        def reviser(self):
            domaines_modifies = False

            # reversed() retourne l'inverse d'une liste ou d'un tuple.
            # Les paires sont donc (var1, var2) et (var2, var1).
            # Les tuples ne sont pas identiques mais ils contiennent des références
            # sur les mêmes objets Variable (modifier var1 dans le premier tuple\
            # modifie var1 dans le second).
            for var1, var2 in (self.variables, reversed(self.variables)):
                ancienne_valeur = var1.val
                for val in var1.domaine[:]:
                    var1.val = val

                    if not self.est_possible(var2):
                        var1.domaine.remove(val)
                        domaines_modifies = True
                var1.val = ancienne_valeur

            return domaines_modifies
```

Module `.../moteur_psc/moteur_psc.py` :

```
class PSC:
    def __init__(self, variables, contraintes):
        self.variables = variables
        self.contraintes = contraintes

        self.iterations = 0
        self.solutions = []

    def consistance_noeuds(self):
        for contrainte in self.contraintes:
            if contrainte.dimension() == 1:
                # Nous créons un nouveau domaine en ne gardant que les
                # valeurs valides.
                # Le plus simple est d'utiliser la `list comprehension' avec
                # une condition.
                contrainte.variables[0].domaine = [var for var in contrainte.variables[0].
                domaine if contrainte.est_valide(var)]

    def consistance_arcs(self):
        refaire = False
        for contrainte in self.contraintes:
            if contrainte.dimension() == 2 and contrainte.reviser():
                refaire = True
```

```
        if refaire :
            self.consistance_arcs()

    def consistance_avec_vars_precedentes(self, k):
        for contrainte in self.contraintes:
            # Si la variables courante est concernée.
            if self.variables[k] in contrainte.variables:
                for i in range(k):
                    # Si n'importe laquelle des variables précédentes est concernée.
                    if self.variables[i] in contrainte.variables:
                        if contrainte.est_valide(self.variables[k], self.variables[k].val):
                            break
                        else:
                            return False
        # Toutes les contraintes sont valides.
        return True

    def backtracking(self, k=0, une_seule_solution=False):
        if len(self.solutions) == 1 and une_seule_solution:
            return

        self.iterations += 1
        # On est parvenu àune solution.
        if k >= len(self.variables):
            sol = {}
            for var in self.variables:
                sol[var.nom] = var.val
            if len(self.solutions) == 0 or not une_seule_solution:
                self.solutions.append(sol)
        else:
            var = self.variables[k]
            for val in var.domaine:
                var.val = val
                if self.consistance_avec_vars_precedentes(k):
                    # On continue l'algorithme sur la variable k+1.
                    self.backtracking(k=k+1, une_seule_solution=une_seule_solution)
            var.val = None

    def affiche_solutions(self):
        print('Recherche terminée en {} itérations'.format(self.iterations))

        if len(self.solutions) == 0:
            print('Aucune solution trouvée')
            return

        for sol in self.solutions:
            print('Solution')
            print('========')
            for (nom, var) in sorted(sol.items()):
                print('\tVariable {}: {}'.format(nom, var))
```

Exercice 8.2 Consistance des nœuds et des arcs

Module `.../moteur_psc_heuristique/contrainte_avec_propagation.py` :

```
from moteur_psc.contrainte import ContrainteBinaire

class ContrainteAvecPropagation(ContrainteBinaire):
    def __init__(self, var1, var2, op):
        ContrainteBinaire.__init__(self, var1, var2, op)

    def reviser(self):
        # Nous appliquons d'abord la méthode reviser() de la classe-mère pour
        # réviser les domaines de chaque variable.
        domaines_modifies = ContrainteBinaire.reviser(self)

        # Puis, s'il y a lieu, nous nous assurons que les labels sont toujours
        # identiques aux domaines.
        if domaines_modifies:
            for var in self.variables:
                var.label = var.domaine[:]

        return domaines_modifies


    def propage(self, var):
        var1, var2 = self.variables

        if var == var1:
            fixe = var1
            variable = var2
        elif var == var2:
            variable = var1
            fixe = var2
        else:
            raise ValueError('Var est ' + var.nom + '. ' +
                             'on attendrait ' + var1.nom + ' ou ' + var2.nom)

        # On ne garde que les valeurs du label pour lesquelles variable
        # reste valide.
        for val in variable.label[:]:
            if not self.est_valide(variable, val):
                variable.label.remove(val)

        # S'il existe au moins une valeur possible, l'assignation est consistante.
        return len(variable.label) > 0
```

Module `.../psc_heuristique/moteur_psc_heuristique.py` :

```
from moteur_psc.psc import PSC

class PSCHeuristique(PSC):

    def __init__(self, variables, contraintes):
        PSC.__init__(self, variables, contraintes)

        self.reinitialise()

    def reinitialise(self):
```

```
        self . initialise_labels ()
        self . solutions = []
        self . iterations = 0

    def initialise_labels ( self ):
        for var in self . variables :
            var . label = var.domaine[:]

    def consistance_noeuds( self ):
        # Nous appelons d'abord la méthode de la classe-mère PSC pour réduire
        # les domaines.
        PSC.consistance_noeuds(self)

        # Puis, nous nous assurons que les labels sont identiques aux domaines.
        self . initialise_labels ()

    def variable_ordering ( self ):
        self . variables . sort (key=lambda x: len(x.domaine))

    def dynamic_variable_ordering( self , k):
        index = k
         taille_plus_petit_label = len(self . variables [index]. label )

        for i in range(k+1, len(self.variables )):
            if len( self . variables [ i ]. label ) < taille_plus_petit_label :
                index = i
                 taille_plus_petit_label = len(self . variables [ i ]. label )

        if k != index:
            self . variables [k], self . variables [index] = self . variables [index], self . variables [k]

    def propagation_consistante( self , k):
        # Pour chaque contrainte portant sur la variable courante,
        for contrainte in self . contraintes :
            if self . variables [k] in contrainte . variables :
                # si la contrainte porte sur une des variables suivantes,
                for i in range(k+1, len(self.variables )):
                    if self . variables [ i ] in contrainte . variables :
                        # on propage la nouvelle assignation.
                        if contrainte . propage(self . variables [k ]):
                            break
                        else:
                            # La contrainte ne peut pas être satisfaite .
                            return False
        return True

    def forward_checking( self , k=0, une_seule_solution=False):
        if len( self . solutions ) == 1 and une_seule_solution:
            return

        self . iterations += 1
        if k >= len(self.variables ):
            sol = {}
            for var in self . variables :
                sol [var . nom] = var.val
            self . solutions . append(sol)
        else:
            self . dynamic_variable_ordering(k)
            variable = self . variables [k]
```

```
# Conserve une copie des labels de départ.
sauvegarde_labels = { var: var.label [:] for var in self.variables }

for val in sauvegarde_labels[variable]:
    variable.val = val
    variable.label = [val]
    if self.propagation_consistante(k):
        # Continue l'algorithme sur la variable k+1.
        self.forward_checking(k=k+1, une_seule_solution=une_seule_solution)
        for var in self.variables:
            var.label = sauvegarde_labels[var]
variable.val = None
```

Chapitre 9

Diagnostic

Exercice 9.1 Diagnostic d'un réseau par abduction explicite

Module `.../reseau/abduction.py` :

```
from .disjonction import Disjonction
from .conjonction import Conjonction

class Abduction:

    def __init__(self, conflits, no_goods):
        self.conflits = conflits
        self.no_goods = no_goods

    def combiner_conflits_observations(self, disjonctions):
        combinaison = Disjonction()
        for disjonction in disjonctions:
            combinaison = combinaison.combiner(disjonction)
        return combinaison

    def retire_subsumes(self, conjonctions):
        sans_subsumes = Disjonction()

        for conjonction in sorted(conjonctions, key=lambda expl: len(expl)):
            conserver = True
            for conj in sans_subsumes:
                if conj.issubset(conjonction):
                    conserver = False
                    break
            if conserver:
                sans_subsumes.add(conjonction)

        return sans_subsumes

    def retire_no_goods(self, conjonctions, no_goods):
        sans_no_goods = Disjonction()

        for conjonction in conjonctions:
            conserver = True
            for no_good in no_goods:
                if no_good.issubset(conjonction):
                    conserver = False
                    break
            if conserver:
                sans_no_goods.add(conjonction)

        return sans_no_goods

    def calcule_conflit_minimal(self, afficher_etapes=False):
        # 1. Combine les conflits.
        conflit_minimal = self.combiner_conflits_observations(self.conflits)
        if afficher_etapes: print('Conflit combiné :', conflit_minimal)
```

```
# 3. Supprime les candidats subsumés.
conflit_minimal = self.retire_subsumes(conflit_minimal)
if afficher_etapes: print('Non subsumés :', conflit_minimal)

# 4. Supprime les candidats contenant les no-goods.
conflit_minimal = self.retire_no_goods(conflit_minimal, self.no_goods)
if afficher_etapes: print('Sans no-goods :', conflit_minimal)

return conflit_minimal
```

Chapitre 10

Génération de plans

Exercice 10.1 Modélisation

Planification : Modélisation sur papier

Il existe souvent différentes façons de modéliser un problème de planification donné sous la forme d'un Problème de Satisfaction de Contraintes. Le modèle que nous vous proposons ici correspond à celui qui est décrit dans le chapitre 10.6 du cours. Si vous avez développé une solution différente, il est cependant possible qu'elle soit tout aussi valide.

Définition du problème de planification

Rappelons qu'un problème de planification est défini par les éléments suivants :

- Une liste de propositions qui décrivent l'état du monde.
- Une liste d'opérateurs qui décrivent les actions qui peuvent être exécutées pour changer l'état du monde. Chaque opérateur possède des préconditions et des postconditions.
- Des conditions initiales, qui décrivent complètement l'état initial du monde en termes de propositions.
- Des conditions finales, qui décrivent complètement ou partiellement l'état du monde désiré en termes de propositions.
- Des mutex, qui stipulent des contraintes d'exclusion mutuelle entre les propositions et entre les opérateurs. Les mutex de propositions sont des paires de propositions qui ne peuvent être vraies en même temps. Par exemple, un missionnaire ne peut pas simultanément se trouver sur la rive gauche et sur la rive droite. Les mutex d'opérateurs définissent les paires d'opérateurs qui ne peuvent être exécutés en même temps. Par exemple, un même bateau ne peut pas être piloté à la fois par le missionnaire M_1 pour transporter le canibale C_1 et par M_2 pour transporter C_2.

Dans notre problème de planification, nous considérons trois types d'acteurs :

- Les bateaux, qui correspondent au type B. Dans notre exemple, il n'y a qu'un seul bateau, dénoté par B.
- Les missionnaires, qui correspondent au type M. Il y a deux missionnaires, M_1 et M_2.
- Les cannibales, qui correspondent au type C. Il y a deux cannibales, C_1 et C_2.

Le choix des propositions du problème de planification doit permettre de décrire complètement la position de chaque acteur.

Notre problème présente aussi deux types d'actions possibles :

- traversée du fleuve depuis la rive gauche jusqu'à la rive droite,
- traversée du fleuve depuis la rive droite jusqu'à la rive gauche.

Chacun de ces deux types d'actions peut faire intervenir un certain nombre d'acteurs. Par exemple, une traversée ne peut avoir lieu qu'à l'aide d'un bateau (c'est-à-dire un acteur de type B). Seul un missionnaire (c'est-à-dire un acteur de type M) peut conduire le bateau. Le bateau ne peut contenir qu'un passager supplémentaire, qui peut être indifféremment de type M (un missionnaire) ou de type C (un cannibale). La liste des opérateurs du problème de planification doit couvrir toutes les combinaisons d'acteurs pour les deux types d'actions.

Choix des propositions du problème de planification

Afin de décrire la position d'un acteur, nous vous proposons d'utiliser deux propositions, correspondant aux deux positions possibles de l'acteur (rive gauche ou rive droite). Pour un acteur A donné, nous introduisons donc les deux propositions suivantes :

- $g(A)$ est vraie si et seulement si l'acteur A est sur la rive gauche,
- $d(A)$ est vraie si et seulement si l'acteur A est sur la rive droite.

Ces deux propositions peuvent vous sembler redondantes, puisque A est nécessairement sur la rive droite s'il n'est pas sur la rive gauche ($d(A)$ est vraie si $g(A)$ est fausse et vice-versa). Nous avons cependant choisi cette représentation car elle se généralise aisément à des problèmes plus compliqués, dans lesquels un acteur peut occuper plus de deux positions.

Choix des opérateurs du problème de planification

Comme nous l'avons indiqué plus haut, des opérateurs seront nécessaires pour modéliser la traversée du fleuve par des groupes d'acteurs. Pour un état donné, les opérateurs seront les suivants :

- $gd(B, M, A)$ est l'opérateur décrivant la traversée du fleuve de gauche à droite, à bord du bateau B, piloté par le missionnaire M, avec pour passager A (qui peut par ailleurs être de type M ou C). Les préconditions de cet opérateur sont que les propositions $g(B)$, $g(M)$ et $g(A)$ doivent être vraies. Les postconditions sont que les propositions $d(B)$, $d(M)$ et $d(A)$ doivent être vraies.
- $dg(B, M)$ est l'opérateur décrivant la traversée de droite à gauche du missionnaire M à bord du bateau B. Les préconditions de cet opérateur sont que les propositions $d(B)$ et $d(M)$ doivent être vraies. Les postconditions sont que les propositions $g(B)$ et $g(M)$ doivent être vraies.

Le choix a été fait dans ce modèle de ne pas tenir compte de la possibilité pour un bateau de faire la traversée de droite à gauche avec un passager en plus du pilote. Ce choix limite l'éventail des plans valides possibles. Il a l'avantage de correspondre à un problème de planification plus petit, plus facile à résoudre,

et qui générera des plans plus courts (puisqu'il n'est alors pas permis de perdre du temps à faire traverser un cannibale dans un sens, puis dans l'autre sens). Dans cet exercice, sachant que les deux cannibales sont sur la rive gauche dans l'état initial, il est très facile de constater qu'il existe un plan valide qui ne fait jamais traverser un cannibale de la rive droite à la rive gauche. Il faut cependant se rappeler, que, dans le cas d'un problème plus général, un tel choix de modélisation pourrait déboucher sur un problème de planification infaisable, quand bien même le problème initial serait soluble.

Conditions initiales du problème de planification

Au début, tous les acteurs sont sur la rive gauche. Les conditions initiales sont donc les suivantes :

$$g(B) = g(M_1) = g(M_2) = g(C_1) = g(C_2) = \texttt{True}$$

Conditions finales du problème de planification

Le but est de faire passer tous les acteurs du côté droit de la rivière. Les conditions finales sont donc les suivantes :

$$d(B) = d(M_1) = d(M_2) = d(C_1) = d(C_2) = \texttt{True}$$

Mutex de propositions du problème de planification

Les mutex de propositions sont des paires de propositions qui ne peuvent être vraies en même temps. Dans notre problème, pour chaque acteur A, nous avons le mutex suivant :

$$[g(A), d(A)]$$

En effet, un acteur ne peut se trouver en même temps sur la rive gauche et sur la rive droite.

Mutex d'opérateurs du problème de planification

Les mutex d'opérateurs sont des paires d'opérateurs qui ne peuvent être exécutés en même temps. Dans notre problème, pour chaque paire d'opérateurs distincts op_1 et op_2, si les deux opérateurs ont un acteur en commun (qu'il soit de type B, M ou C), alors on a le mutex suivant :

$$[\text{op}_1, \text{op}_2]$$

Dans le cas où l'acteur en commun est de type B, ce mutex traduit le fait qu'un même bateau ne peut pas contenir deux équipages différents en même temps. Dans le cas où l'acteur en commun est de type M ou C, et si l'on a plusieurs bateaux à disposition, un même missionnaire ou un même cannibale ne peut se trouver sur deux bateaux différents en même temps.

Description du modèle PSC pour le problème de planification

Rappelons qu'un plan non linéaire est constitué d'une séquence d'états S_0, S_1, ..., S_n, chaque état S_i étant décrit par des propositions qui sont vraies ou fausses au début de l'état, un ensemble d'opérateurs qui sont exécutés pendant cet état et des propositions qui sont vraies ou fausses à la fin de l'état (après l'exécution des opérateurs), et qui correspondent aux propositions du début de l'état suivant.

Un opérateur est caractérisé par des préconditions, c'est-à-dire des propositions qui doivent être vraies au début de l'état S_i pour que l'opérateur puisse être exécuté durant S_i, et des postconditions, c'est-à-dire des propositions qui devront être vraies au début de l'état S_{i+1}.

Choix des variables PSC pour les propositions

Pour chaque état S_i, avec $i = 0...n+1$, et chaque acteur A, le PSC contiendra les variables booléennes suivantes, qui correspondent aux propositions du problème de planification introduites plus haut :

- $g(A, S_i) =$ `True` si et seulement si l'acteur A est sur la rive gauche à la fin de l'état S_{i-1} et au début de l'état S_i,
- $d(A, S_i) =$ `True` si et seulement si l'acteur A est sur la rive droite à la fin de l'état S_{i-1} et au début de l'état S_i.

Choix des variables PSC pour les opérateurs

Pour chaque état S_i, avec $i = 0...n$, et chaque opérateur op (par exemple, $dg(B, M_1)$), le PSC contiendra la variable booléenne op(S_i), qui indiquera si oui ou non l'opérateur est exécuté dans l'état S_i.

Expression des contraintes du PSC

Il existe six types de contraintes qui doivent être vérifiées pour qu'un plan soit valide :

- les contraintes sur l'état initial,
- les contraintes sur l'état final,
- les préconditions et les postconditions des opérateurs,
- les axiomes de cadre,
- les mutex de propositions,
- les mutex d'opérateurs.

Contraintes PSC correspondant aux contraintes sur l'état initial. Tous les acteurs sont initialement sur la rive gauche. Par conséquent, les contraintes sur l'état initial prennent une forme très simple, qui est la suivante :

$$g(B, S_0) = g(M_1, S_0) = g(M_2, S_0) = g(C_1, S_0) = g(C_2, S_0) = \texttt{True}$$

Contraintes PSC correspondant aux contraintes sur l'état final. Les contraintes sur l'état final sont comparables aux contraintes sur l'état initial, à la différence fondamentale près qu'il faut définir lequel des états S_0, S_1, ..., S_n correspond à cet état final. En d'autres termes, il faut choisir par avance la longueur des plans solutions que l'on veut considérer, c'est-à-dire le nombre total d'états.

Si l'on ne considère que des plans très courts, il est possible que l'on n'obtienne aucune solution, car il peut n'exister aucun plan qui parvienne à passer de l'état initial à l'état final avec si peu d'états intermédiaires. Si en revanche on autorise des plans très longs, la taille du PSC et la complexité de sa résolution explosent. Traditionnellement, on commence donc par des plans courts, et on augmente progressivement leur taille tant qu'aucune solution n'est découverte.

Dans notre cas, le problème se résout très facilement à la main. On peut donc tricher et choisir le nombre d'états en sachant qu'il existe un plan solution qui le contient. Les étapes suivantes constituent ainsi une solution possible, sachant que le but à atteindre est d'avoir tous les acteurs sur la rive droite :

- $gd(B, M_1, C_1)$,
- $dg(B, M_1)$,
- $gd(B, M_1, C_2)$,
- $dg(B, M_1)$,
- $gd(B, M_1, M_2)$.

Il suffit donc de cinq étapes, soit cinq états (sans compter l'état initial S_0) pour obtenir l'état final désiré. On choisira donc S_5 comme état final. Les contraintes sur l'état final sont par conséquent les suivantes :

$$d(B, S_5) = d(M_1, S_5) = d(M_2, S_5) = d(C_1, S_5) = d(C_2, S_5) = \texttt{True}$$

Contraintes PSC correspondant aux contraintes de préconditions et de postconditions des opérateurs. Reprenons les deux types d'opérateurs introduits précédemment et explicitons leurs préconditions et postconditions en termes de variables du PSC :

- L'opérateur $gd(B, M, A)$ a pour préconditions que $g(B)$, $g(M)$ et $g(A)$ doivent toutes être vraies. Les postconditions stipulent que $d(B)$, $d(M)$ et $d(A)$ doivent toutes être vraies.
- L'opérateur $dg(B, M)$ a pour préconditions que $d(B)$ et $d(M)$ doivent être vraies. Les postconditions stipulent que $g(B)$ et $g(M)$ doivent être vraies.

On n'a pas mentionné ici les suppressions de chaque opérateur. Par exemple, une autre postcondition de $dg(B, M)$ est que $d(B)$ doit être fausse. Il n'est cependant pas nécessaire d'expliciter ces postconditions négatives, car elles seront automatiquement imposées par les mutex de propositions introduits précédemment et présentés en détail plus bas.

En termes de PSC, ces préconditions et postconditions prennent la forme des contraintes suivantes sur les variables du PSC, pour chaque état S_i avec $i = 0...n$ et chaque opérateur op :

- pour chaque proposition prop qui est une postcondition de l'opérateur op, on a : $\text{op}(S_i) \Rightarrow \text{prop}(S_{i+1})$;
- pour chaque proposition prop qui est une précondition de l'opérateur op, on a : $\text{op}(S_i) \Rightarrow \text{prop}(S_i)$.

On utilise ici la notation $\Rightarrow$ pour indiquer l'implication logique. Ainsi, si prop est une postcondition de l'opérateur op, alors $\text{op}(S_i) \Rightarrow \text{prop}(S_{i+1})$ est équivalent à « si $\text{op}(S_i) =$ `True`, alors $\text{prop}(S_{i+1}) =$ `True` ». Ceci traduit bien le fait que si l'opérateur op est exécuté dans l'état S_i, alors sa postcondition prop doit être vraie à la fin de l'état S_i, c'est-à-dire au début de l'état S_{i+1}.

*Contraintes PSC correspondant aux axiomes de cadre*Les axiomes de cadre stipulent que si aucun opérateur ne vient, dans l'état S_i, modifier une proposition prop, alors elle reste identique à l'état S_{i+1}. De manière équivalente : si la valeur d'une proposition change d'un état S_i à l'état suivant S_{i+1}, c'est que l'opérateur $\text{op}(S_i)$ y est pour quelque chose. Par exemple, si $g(C_1, S_i) =$ `True` et $g(C_1, S_{i+1}) =$ `False`, alors $\text{op}(S_i)$ est nécessairement l'un des deux opérateurs suivants : $gd(B, M_1, C_1)$ ou $gd(B, M_2, C_1)$.

Remarquez que cette contrainte est de formulation relativement complexe, et en particulier qu'elle implique plus de deux variables : $\text{prop}(S_i)$, $\text{prop}(S_{i+1})$, et les variables pour l'état S_i de tous les opérateurs qui ont prop comme précondition ou postcondition. Plus précisément, les contraintes d'axiomes de cadre prennent les deux formes suivantes, pour chaque état S_i, et pour chaque proposition prop :

- si $\text{prop}(S_i) =$ `False` et $\text{prop}(S_{i+1}) =$ `True`, alors pour au moins un opérateur op qui a prop comme postcondition, on a $\text{op}(S_i) =$ `True`,
- si $\text{prop}(S_i) =$ `True` et $\text{prop}(S_{i+1}) =$ `False`, alors pour au moins un opérateur op qui a prop comme postcondition négative, on a $\text{op}(S_i) =$ `True`.

Comme indiqué précédemment, il n'est pas nécessaire de considérer le deuxième cas, qui comporte une postcondition négative. En effet, si nous prenons l'exemple de la proposition $d(M_1)$, lorsque celle-ci passe de `True` à `False`, il n'est pas nécessaire de vérifier que l'opérateur $dg(M_1)$ est exécuté, puisqu'alors les contraintes de mutex de propositions imposeront que $g(M_1)$ passe (à l'inverse) de `False` à `True`. Le premier cas ci-dessus suffit alors pour assurer que $dg(M_1)$ soit exécuté.

Rappelons que ces contraintes ne sont pas simplement unaires ou binaires, mais qu'elles ont une multiplicité plus grande que deux. Il faut donc aussi modifier le module PSC pour qu'il puisse manipuler de telles contraintes.

*Contraintes PSC correspondant aux mutex de propositions*Les mutex de propositions sont des contraintes qui stipulent que deux propositions sont mutuellement exclusives, c'est-à-dire qu'elles sont incompatibles et ne peuvent pas être vraies en même temps. Par exemple, on ne peut pas avoir $g(M_1)$ et $d(M_1)$ vraies en même temps, puisque le missionnaire M_1 ne peut pas se trouver à la fois sur la rive droite et sur la rive gauche.

Plus généralement donc, pour chaque état S_i, et pour chaque acteur A (de n'importe quel type), le modèle PSC contiendra les contraintes suivantes :

$$g(A, S_i) \text{ NAND } d(A, S_i)$$

Cette contrainte autorise virtuellement qu'un acteur puisse n'être ni sur la rive gauche, si sur la rive droite ($g(A, S_i) = d(A, S_i) =$ `False`). Mais, dans la pratique, cette situation ne pourra pas se produire. Ceci peut se démontrer par récurrence : Nous partons du principe que les contraintes sur l'état initial décrivent complètement la situation initiale. C'est-à-dire qu'au début de l'état 0, un acteur est nécessairement sur une rive donnée. Considérons maintenant l'état k, et supposons qu'un acteur se trouve sur la rive gauche ($g(A) =$ `True`) au début de cet état (le raisonnement est tout aussi valide si l'on remplace « gauche » par « droite » et « droite » par « gauche »). Deux possibilités se présentent alors pour les opérateurs exécutés dans l'état k : 1) aucun opérateur n'a $g(A)$ comme précondition, et alors les contraintes d'axiomes de cadre vont imposer que $g(A)$ reste vraie à la fin de l'état ; 2) au moins un opérateur a $g(A)$ comme précondition, et alors les contraintes de postconditions vont imposer que $d(A)$ soit vraie à la fin de l'état. Dans les deux cas, la rive sur laquelle se trouve l'acteur à la fin de l'état est clairement définie.

Contraintes PSC correspondant aux mutex d'opérateurs. De manière analogue aux mutex de propositions, chaque mutex d'opérateur $[\text{op}_1, \text{op}_2]$ va donner lieu, pour chaque état S_i, à la contrainte suivante :

$$\text{op}_1(S_i) \text{ NAND } \text{op}_2(S_i)$$

Résumé du modèle PSC

Au bout du compte, voici le modèle que nous obtenons :

Variables :

1) Variables booléennes $g(A, S_i)$, pour chaque acteur A et chaque état S_i.
2) Variables booléennes $d(A, S_i)$, pour chaque acteur A et chaque état S_i.
3) Variables booléennes $\text{op}(S_i)$, pour chaque opérateur op et chaque état S_i.

Contraintes :

1) Contraintes sur l'état initial :

$$g(B, S_0) = g(M_1, S_0) = g(M_2, S_0) = g(C_1, S_0) = g(C_2, S_0) = \texttt{True}$$

2) Contraintes sur l'état final :

$$d(B, S_5) = d(M_1, S_5) = d(M_2, S_5) = d(C_1, S_5) = d(C_2, S_5) = \texttt{True}$$

3) Contraintes de préconditions et postconditions des opérateurs, pour chaque état S_i et chaque opérateur op :

- Pour chaque proposition prop qui est une postcondition de l'opérateur op : $\text{op}(S_i) \Rightarrow \text{prop}(S_{i+1})$.
- Pour chaque proposition prop qui est une précondition de l'opérateur op : $\text{op}(S_i) \Rightarrow \text{prop}(S_i)$.

4) Contraintes d'axiomes de cadre, pour chaque état S_i et chaque proposition prop :
   - Si $\text{prop}(S_i) =$ `False` et $\text{prop}(S_{i+1}) =$ `True`, alors pour au moins un opérateur op qui a prop comme postcondition, on a $\text{op}(S_i) =$ `True`
5) Contraintes de mutex de propositions, pour chaque état S_i et pour chaque acteur A :
$$g(A, S_i) \text{ NAND } d(A, S_i)$$
6) Contraintes de mutex d'opérateurs, pour chaque état S_i, et pour chaque paire d'opérateurs distincts op_1 et op_2 qui ont un acteur en commun :
$$\text{op}_1(S_i) \text{ NAND } \text{op}_2(S_i)$$

Dans ce PSC, toutes les variables sont booléennes. Typiquement, plutôt qu'un algorithme PSC, on utiliserait un algorithme SAT pour résoudre ce genre de problèmes. Un algorithme SAT est en effet spécialisé dans la résolution de PSC exprimés sous la forme de clauses (des expressions qui peuvent être vraies ou fausses) ne comportant que des variables booléennes.

Exercice 10.2 Implémentation

Module `.../moteur_psc_planification/axiomecadre.py` :

```
from moteur_psc.contrainte import Contrainte

class ContrainteAxiomeCadre(Contrainte):
    def __init__(self, var_pre, ops, var_post):
        Contrainte.__init__(self, (var_pre, var_post) + tuple(ops))

        self.var_pre = var_pre
        self.var_post = var_post
        self.vars_ops = ops

    def est_valide(self, var, val):
        ancienne_valeur = var.val
        var.val = val

        # On part du principe que la contrainte est valide si au moins une
        # variable n'est pas instanciée.
        for var2 in self.variables:
            if var2.val is None:
                var.val = ancienne_valeur
                return True

        valide = False
```

```
        # Si toutes les variables sont instanciées et qu'une variable passe de
        # False à True.
        if self.var_pre.val == False and self.var_post.val == True:
            # Vérifie qu'au moins un des opérateurs est appliqué.
            for op in self.vars_ops:
                if op.val == True:
                    valide = True
                    break
        else:
            valide = True

        var.val = ancienne_valeur
        return valide

    def propage(self, var):
        var2 = None
        for var_i in self.variables:
            if var_i.val is None:
                if var2 is None:
                    var2 = var_i
                else:
                    # Il reste plus d'une variable à instancier.
                    return True

        # Teste les valeurs du label pour la dernière variable non instanciée.
        for val in var2.label[:]:
            if not self.est_valide(var2, val):
                var2.label.remove(val)

        return len(var2.label) > 0

    def reviser(self):
        return False

    def __repr__(self):
        return 'Axiome de cadre:\n\t{}\n\t{}\n\t{}'.format(self.var_pre,
                                                          [op for op in self.vars_ops],
                                                          self.var_post)
```

Module `.../moteur_planification/etat.py` :

```
from moteur_psc_heuristique.variable_avec_label import VariableAvecLabel

class Etat:
    def __init__(self, no_etat, propositions, operateurs, etat_prec=None):
        self.no_etat = no_etat
        self.etat_prec = etat_prec

        self.operateurs = { op.nom: op for op in operateurs }

        self.vars_initiales = {}
        self.vars_finales = {}

        self.construire_vars_operateurs(operateurs)
        self.construire_vars_propositions(propositions)

    def construire_vars_operateurs(self, ops):
        self.vars_operateurs = {}
```

```
        for op in ops:
            var_nom = '{} état {}'.format(op.nom, self.no_etat)
            self.vars_operateurs[op.nom] = VariableAvecLabel(var_nom,
                                                             [True, False])

    def construire_vars_propositions(self, props):
        if self.no_etat > 0:
            self.vars_initiales = self.etat_prec.vars_finales

        for prop in props:
            var_nom = '{} état {}'.format(prop, self.no_etat + 1)
            self.vars_finales[prop] = VariableAvecLabel(var_nom,
                                                        [True, False])
            if self.no_etat == 0:
                var_nom = '{} état 0'.format(prop)
                self.vars_initiales[prop] = VariableAvecLabel(var_nom,
                                                              [True, False])

    def variables(self):
        return (list(self.vars_initiales.values()) +
                list(self.vars_finales.values()) +
                list(self.vars_operateurs.values()))
```

Module `.../moteur_planification/planification.py` :

```
from moteur_psc.contrainte import ContrainteUnaire
from moteur_psc_heuristique.contrainte_avec_propagation import ContrainteAvecPropagation
from moteur_psc_heuristique.psc_heuristique import PSCHeuristique
from moteur_psc_planification.axiomecadre import ContrainteAxiomeCadre
from .etat import Etat

class Planification:
    def __init__(self, propositions, operateurs,
                 mutex_propositions, mutex_operateurs,
                 depart, but, nb_etats):
        self.operateurs = operateurs
        self.mutex_propositions = mutex_propositions
        self.mutex_operateurs = mutex_operateurs

        self.depart = depart
        self.but = but

        self.nb_etats = nb_etats

        self.propositions = propositions

        self.etats = []
        self.construire_etats()

        self.psc = PSCHeuristique(self.variables(), self.construire_contraintes())

    def construire_etats(self):
        self.etats.append(Etat(0, self.propositions, self.operateurs, None))

        for i in range(1, self.nb_etats):
            self.etats.append(Etat(i, self.propositions,
                                   self.operateurs, self.etats[-1]))
```

```
def variables(self):
    # Utiliser un set évite les doublons entre variables finales et
    # initiales.
    variables = set()
    for etat in self.etats:
        variables.update(etat.variables())

    return list(variables)

def construire_contraintes(self):
    return (self.construire_contraintes_propositions() +
            self.construire_contraintes_operateurs() +
            self.construire_contraintes_conditions() +
            self.construire_contraintes_axiomes_cadre() +
            self.construire_contraintes_initiales() +
            self.construire_contraintes_finales())

def construire_contraintes_propositions(self):
    contraintes = []
    nand = lambda x,y: not (x and y)

    # Construction des contraintes générées par les mutex de propositions
    # **pour chaque état**.
    for mutex in self.mutex_propositions:
        for etat in self.etats:
            contr = ContrainteAvecPropagation(etat.vars_initiales[mutex[0]],
                                              etat.vars_initiales[mutex[1]],
                                              nand)
            contraintes.append(contr)
            # Les mutex de propositions doivent aussi être valides pour les
            # variables finales du dernier état.
            if etat.no_etat == (self.nb_etats - 1):
                contr = ContrainteAvecPropagation(etat.vars_finales[mutex[0]],
                                                  etat.vars_finales[mutex[1]],
                                                  nand)
                contraintes.append(contr)

    return contraintes

def construire_contraintes_operateurs(self):
    contraintes = []
    nand = lambda x,y: not (x and y)

    for mutex in self.mutex_operateurs:
        for etat in self.etats:
            contr = ContrainteAvecPropagation(etat.vars_operateurs[mutex[0].nom],
                                              etat.vars_operateurs[mutex[1].nom],
                                              nand)
            contraintes.append(contr)

    return contraintes

def construire_contraintes_conditions(self):
    contraintes = []
    # Contraintes générées par les pré- et post-conditions.
    # Implication logique.
    imp = lambda x,y: (not x) or y
```

```
        for etat in self.etats:
            for op in self.operateurs:
                for precond in op.precond:
                    contr = ContrainteAvecPropagation(etat.vars_operateurs[op.nom],
                                                      etat.vars_initiales[precond],
                                                      imp)
                    contraintes.append(contr)
                for postcond in op.postcond:
                    contr = ContrainteAvecPropagation(etat.vars_operateurs[op.nom],
                                                      etat.vars_finales[postcond],
                                                      imp)
                    contraintes.append(contr)

        return contraintes

    def construire_contraintes_axiomes_cadre(self):
        contraintes = []

        for etat in self.etats:
            for prop in self.propositions:
                vars_ops = [etat.vars_operateurs[op.nom]
                            for op in self.operateurs
                            if prop in op.postcond]
                contr = ContrainteAxiomeCadre(etat.vars_initiales[prop],
                                              vars_ops,
                                              etat.vars_finales[prop])
                contraintes.append(contr)
        return contraintes

    def construire_contraintes_initiales(self):
        contraintes = []
        for contrainte in self.depart:
            eq = lambda x: x == contrainte[1]
            contr = ContrainteUnaire(self.etats[0].vars_initiales[contrainte[0]], eq)
            contraintes.append(contr)

        return contraintes

    def construire_contraintes_finales(self):
        contraintes = []

        for contrainte in self.but:
            eq = lambda x: x == contrainte[1]
            contr = ContrainteUnaire(self.etats[-1].vars_finales[contrainte[0]], eq)
            contraintes.append(contr)

        return contraintes

    def resoudre(self):
        self.psc.consistance_noeuds()
        self.psc.consistance_arcs()
        self.psc.variable_ordering()

        self.psc.forward_checking(0, True)
        self.sol = self.psc.solutions

        return self.sol

    def affice_solutions(self):
```

```
        print('Recherche terminée en {} itérations'.format(self.psc.iterations))

        if len(self.psc.solutions) == 0:
            print('Aucune solution trouvée')
            return

        for sol in self.psc.solutions:
            print('Solution')
            print('========')
            for etat in self.etats:
                print('État {}: '.format(etat.no_etat))
                print('  Propositions initiales :')
                for nom, var in sorted(etat.vars_initiales.items()):
                    if sol[var.nom]:
                        print('    ' + nom)

                print('  Opérateurs:')
                for nom, var in sorted(etat.vars_operateurs.items()):
                    if sol[var.nom]:
                        print('    ' + nom)

                print('  Propositions finales :')
                for nom, var in sorted(etat.vars_finales.items()):
                    if sol[var.nom]:
                        print('    ' + nom)
                print()
```

Module `.../exemple_missionnaires.py` :

```
from moteur_planification.operateur import Operateur
from moteur_planification.planification import Planification

def format_g(acteur):
    return 'g({})'.format(acteur)

def format_d(acteur):
    return 'd({})'.format(acteur)

def format_dg(bateau, pilote):
    return 'dg({}, {})'.format(bateau, pilote)

def format_gd(bateau, pilote, passager):
    return 'gd({}, {}, {})'.format(bateau, pilote, passager)


bateaux = ['B']
missionnaires = ['M1', 'M2']
cannibales = ['C1', 'C2']

acteurs = bateaux + missionnaires + cannibales

# Ajoute les propositions pour la position des acteurs.
propositions = []
for acteur in acteurs:
    propositions.append(format_g(acteur))
    propositions.append(format_d(acteur))

# Ajoute les opérateurs de déplacement.
```

```
operateurs = []
for bateau in bateaux:
    for pilote in missionnaires:
        # Déplacements du bateau sans passagers (droite àgauche).
        operateurs.append(Operateur(
            format_dg(bateau, pilote),
            [format_d(bateau), format_d(pilote)],
            [format_g(bateau), format_g(pilote)])
            )

        for passager in missionnaires + cannibales:
            # Déplacements du bateau avec passagers (gauche àdroite).
            if passager != pilote:
                operateurs.append(Operateur(
                    format_gd(bateau, pilote, passager),
                    [format_g(bateau), format_g(pilote), format_g(passager)],
                    [format_d(bateau), format_d(pilote), format_d(passager)])
                    )

# Ajoute les mutex de proposition (un acteur ne peut pas être sur les deux rives
# simultanément).
mutex_propositions = []
for acteur in acteurs:
    mutex_propositions.append((format_g(acteur), format_d(acteur)))

# Ajoute les mutex d'opérateurs.
mutex_operateurs = []
for i in range(len(operateurs)):
    for j in range(i+1, len(operateurs)):
        for acteur in acteurs:

            if ((format_d(acteur) in operateurs[i].precond or
                 format_g(acteur) in operateurs[i].precond)
                and
                (format_d(acteur) in operateurs[j].precond or
                 format_g(acteur) in operateurs[j].precond)):

                mutex_operateurs.append((operateurs[i], operateurs[j]))
                break

# Ajoute les contraintes initiales (tous les acteurs à gauche).
depart = []
for acteur in acteurs:
    depart.append((format_g(acteur), True))

# Ajoute les contraintes finales (but: tous les acteurs à droite).
but = []
for acteur in acteurs:
    but.append((format_d(acteur), True))

# Transforme le problème de planification en PSC.
plan = Planification(propositions, operateurs,
                     mutex_propositions, mutex_operateurs,
                     depart, but,
                     nb_etats=5)
plan.resoudre()

plan.affice_solutions()
```

Chapitre 11

Induction de classifications simples à partir d'exemples

Exercice 11.1 Attributs discrets

1) Les descriptions conjonctives suivantes sont possibles :
   - $fuselé \wedge large$,
   - $fuselé$.

   Le plus court le mieux : c'est donc $fuselé$ qui est la meilleure solution.
2) On ne prend pas l'intersection de tous les attributs que partagent les champignons venimeux, parce qu'on obtiendrait une description qui est trop spécifique, c'est-à-dire $fuselé \wedge large$.
3) Après l'ajout du nouvel exemple, il n'existe pas de description conjonctive de la classe des champignons non venimeux, parce qu'il n'y a aucun attribut qui est partagé par tous les exemples.
4) Non, on ne peut pas utiliser ce nouvel attribut pour apprendre une classification conjonctive, parce qu'aucune valeur de l'attribut n'est vraie pour tous les exemples d'une même classe. Cependant, si on regroupe $\{rouge, brun\} = coloré$, alors les champignons colorés sont ceux qui sont venimeux.

Exercice 11.2 Attributs numériques

1) Il faut rajouter la coordonnée x^2 qui permet la séparation par la frontière $x^2 \leq 1$.
2) La bande se caractérise par le fait que $y-x$ se situe entre -1 et 1, donc $(y-x)^2 \leq 1$. Elle peut s'exprimer par les trois coordonnées supplémentaires x^2, xy, y^2 : $x^2 - 2xy + y^2 \leq 1$
3) L'intérieur du cercle se caractérise par l'inégalité : $(x-2)^2 + (y-2)^2 < 1$ ou de manière équivalente : $4x + 4y - x^2 - y^2 > 7$, donc nous avons besoin des coordonnées x^2, y^2, x, y.

Chapitre 12

Apprentissage de classifications structurées

Exercice 12.1 Les arbres de décision (ID3)

Module `moteur_id3/id3.py` :

```
from math import log
from .noeud_de_decision import NoeudDeDecision

class ID3:
    def construit_arbre(self, donnees):
        # Nous devons extraire les domaines de valeur des
        # attributs, puisqu'ils sont nécessaires pour
        # construire l'arbre.
        attributs = {}
        for donnee in donnees:
            for attribut, valeur in donnee[1].items():
                valeurs = attributs.get(attribut)
                if valeurs is None:
                    valeurs = set()
                    attributs[attribut] = valeurs
                valeurs.add(valeur)

        arbre = self.construit_arbre_recur(donnees, attributs)

        return arbre

    def construit_arbre_recur(self, donnees, attributs):

        def classe_unique(donnees):
            if len(donnees) == 0:
                return True
            premiere_classe = donnees[0][0]
            for donnee in donnees:
                if donnee[0] != premiere_classe:
                    return False
            return True

        if donnees == []:
            return None

        # Si toutes les données restantes font partie de la même classe,
        # on peut retourner un noeud terminal.
        elif classe_unique(donnees):
            return NoeudDeDecision(None, donnees)

        else:
            # Sélectionne l'attribut qui réduit au maximum l'entropie.
            h_C_As_attribs = [(self.h_C_A(donnees, attribut, attributs[attribut]),
                               attribut) for attribut in attributs]

            attribut = min(h_C_As_attribs, key=lambda h_a: h_a[0])[1]
```

```
            # Crée les sous-arbres de manière récursive.
            attributs_restants = attributs.copy()
            del attributs_restants[attribut]

            partitions = self.partitionne(donnees, attribut, attributs[attribut])

            enfants = {}
            for valeur, partition in partitions.items():
                enfants[valeur] = self.construit_arbre_recur(partition,
                                                             attributs_restants)

            return NoeudDeDecision(attribut, donnees, enfants)

    def partitionne(self, donnees, attribut, valeurs):
        partitions = {valeur: [] for valeur in valeurs}

        for donnee in donnees:
            partition = partitions[donnee[1][attribut]]
            partition.append(donnee)

        return partitions

    def p_aj(self, donnees, attribut, valeur):
        # Nombre de données.
        nombre_donnees = len(donnees)

        # Permet d'éviter les divisions par 0.
        if nombre_donnees == 0:
            return 0.0

        # Nombre d'occurrences de la valeur a_j parmi les données.
        nombre_aj = 0
        for donnee in donnees:
            if donnee[1][attribut] == valeur:
                nombre_aj += 1

        # p(a_j) = nombre d'occurrences de la valeur a_j parmi les données /
        #          nombre de données.
        return nombre_aj / nombre_donnees

    def p_ci_aj(self, donnees, attribut, valeur, classe):
        # Nombre d'occurrences de la valeur a_j parmi les données.
        donnees_aj = [donnee for donnee in donnees if donnee[1][attribut] == valeur]
        nombre_aj = len(donnees_aj)

        # Permet d'éviter les divisions par 0.
        if nombre_aj == 0:
            return 0

        # Nombre d'occurrences de la classe c_i parmi les données pour lesquelles
        # A vaut a_j.
        donnees_ci = [donnee for donnee in donnees_aj if donnee[0] == classe]
        nombre_ci = len(donnees_ci)

        # p(c_i|a_j) = nombre d'occurrences de la classe c_i parmi les données
        #              pour lesquelles A vaut a_j /
        #              nombre d'occurrences de la valeur a_j parmi les données.
        return nombre_ci / nombre_aj
```

```
def h_C_aj(self, donnees, attribut, valeur):
    # Les classes attestées dans les exemples.
    classes = list(set([donnee[0] for donnee in donnees]))

    # Calcule p(c_i|a_j) pour chaque classe c_i.
    p_ci_ajs = [self.p_ci_aj(donnees, attribut, valeur, classe)
                for classe in classes]

    # Si p vaut 0 -> plog(p) vaut 0.
    return -sum([p_ci_aj * log(p_ci_aj, 2.0)
                 for p_ci_aj in p_ci_ajs
                 if p_ci_aj != 0])

def h_C_A(self, donnees, attribut, valeurs):
    # Calcule P(a_j) pour chaque valeur a_j de l' attribut A.
    p_ajs = [self.p_aj(donnees, attribut, valeur) for valeur in valeurs]

    # Calcule H_C_aj pour chaque valeur a_j de l' attribut A.
    h_c_ajs = [self.h_C_aj(donnees, attribut, valeur)
               for valeur in valeurs]

    return sum([p_aj * h_c_aj for p_aj, h_c_aj in zip(p_ajs, h_c_ajs)])
```

Chapitre 13

Apprentissage non supervisé

Exercice 13.1 Clustering

Module `.../moteurs_clustering/clustering_kmeans.py` :

```
from .cluster_mean import ClusterMean
from .clustering import Clustering

class ClusteringKMeans(Clustering):
    def __init__(self, k, dist_f):
        super().__init__()
        self.k = k
        self.dist_f = dist_f

    def noyaux(self, clusters):
        return [cluster.noyau for cluster in clusters]

    def initialise_clusters(self, donnees):
        if len(donnees) < self.k:
            raise Exception('Il faut au moins {} données'.format(self.k))

        # Crée les clusters autour des noyaux, qui sont les premières k données.
        noyaux = [(donnees[i], str(i + 1)) for i in range(self.k)]
        self.clusters = [ClusterMean([noyau[0]], noyau[1]) for noyau in noyaux]

        # Ajoute toutes les autres données au premier cluster.
        self.clusters[0].ajoute_donnees(donnees[self.k:])

    def fini(self, anciens_clusters):
        return self.noyaux(self.clusters) == self.noyaux(anciens_clusters)

    def revise_clusters(self):
        # Extrait toutes les données des anciens clusters, sauf les noyaux.
        donnees = []
        for cluster in self.clusters:
            donnees.extend([d for d in cluster.donnees if d != cluster.noyau])

        # Réinitialise les nouveaux clusters aux noyaux des anciens clusters.
        for cluster in self.clusters:
            cluster.vide(garde_noyau=True)

        # Assigne chaque donnée au cluster du noyau duquel il est le
        # plus proche.
        for donnee in donnees:
            distances = [(self.dist_f(donnee, cluster.noyau), cluster)
                         for cluster in self.clusters]
            cluster = min(distances, key=lambda x: x[0])[1]
            cluster.ajoute_donnee(donnee)

        # Recentre le noyau de chaque nouveau cluster.
        for cluster in self.clusters:
            cluster.centre(self.dist_f)
```

```
    def affiche_clusters (self):
        print('\n'.join([str(cluster) for cluster in self.clusters]))
```

Module `.../moteurs_clustering/clustering_hierarchique.py` :

```
from .cluster_hierarchique import ClusterHierarchique
from .clustering import Clustering

class ClusteringHierarchique(Clustering):
    liens = {
        'single': min,
        'complete': max,
    }

    def __init__(self, type_lien, dist_f):
        super().__init__()
        self.dist_f = dist_f
        # Permet d'utiliser min ou max de manière générique en fonction du
        # paramètre type_lien.
        self.lien = self.liens[type_lien]

    def fusionne_clusters(self, cluster1, cluster2):
        donnees = cluster1.donnees + cluster2.donnees
        return ClusterHierarchique(donnees, cluster1, cluster2)

    def calcule_distance(self, cluster1, cluster2):
        distances = []
        for donnee1 in cluster1.donnees:
            for donnee2 in cluster1.donnees:
                distances.append(self.dist_f(donnee1, donnee2))

        return self.lien(distances)

    def initialise_clusters(self, donnees):
        # Construit les clusters terminaux : un par donnée.
        # Les clusters seront ensuite fusionnés pour créer la hiérarchie.
        self.clusters = [ClusterHierarchique([donnee]) for donnee in donnees]

    def fini(self, anciens_clusters):
        return len(self.clusters) == len(anciens_clusters)

    def revise_clusters(self):
        if len(self.clusters) == 1:
            return

        # Calcule la distance entre chaque paire de clusters.
        distances = []
        for cluster1 in self.clusters:
            for cluster2 in self.clusters:
                if cluster1 != cluster2:
                    distance = self.calcule_distance(cluster1, cluster2)
                    distances.append((distance, cluster1, cluster2))

        # Trouve les deux clusters les plus proches.
        paire = min(distances, key=lambda x: x[0])
        cluster1 = paire[1]
        cluster2 = paire[2]
```

```
        # Fusionne ces deux clusters.
        nouveau_cluster = self.fusionne_clusters(cluster1, cluster2)
        self.clusters.remove(cluster1)
        self.clusters.remove(cluster2)
        self.clusters.append(nouveau_cluster)

    def affiche_clusters(self):
        print('\n'.join([str(cluster) for cluster in self.clusters]))
```

Bibliographie

[1] Stuart Russell and Peter Norvig, *Artificial Intelligence : A Modern Approach* (2nd Edition), Prentice Hall, 2003.
Version francaise : *Intelligence artificielle*, 2e édition. Traduit par Laurent Miclet et Fabrice Popineau. Paris : Pearson Education France, 2006.

[2] David L. Poole et Alan K. Mackworth, *Artificial Intelligence : Foundations of Computational Agents*, Cambridge University Press, 2010.

[3] Nils J. Nilsson, *Artificial Intelligence : A New Synthesis*, Morgan Kaufmann Publishers, 1998.

[4] Nils J. Nilsson, *The Quest for Artificial Intelligence*, Cambridge University Press, 2009.

[5] R. Kowalski, *Logic for Problem Solving*, Elsevier North Holland, 1979.

[6] J. Alan Robinson, « A Machine-Oriented Logic Based on the Resolution Principle », *Journal of the ACM (JACM)*, Vol. 12, Issue 1, pp. 23-41, 1965.

[7] Alfred Horn, « On sentences which are true of direct unions of algebras », *Journal of Symbolic Logic*, 16, pp. 14-21, 1951.

[8] Robert Kowalski and Donald and Kuehner, *Linear Resolution with Selection Function Artificial Intelligence*, Vol. 2, pp. 227-260, 1971.

[9] Jean H. Gallier, *Logic for Computer Science : Foundations of Automatic Theorem Proving*, Wiley, 1986.

[10] Marvin Minsky, « A framework for representing knowledge », in Patrick J. Winston, editor, *The Psychology of Computer Vision*, pp. 211-277, McGraw-Hill, New York, NY, 1975.

[11] John Sowa, *Conceptual Structures : Information Processing in Mind and Machine*, Ed. Addison-Wesley, 1983.

[12] John Sowa, éditeur, *Principles of Semantic Networks : Explorations in the representation of knowledge*, Morgan-Kaufmann, San Mateo, California, 1991.

[13] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider, *The Description Logic Handbook : Theory, Implementation, Applications*, Cambridge University Press, Cambridge, UK, 2003.

[14] OWL Web Ontology Language Reference. World Wide Web Consortium, 2004.

[15] Allen Newell, Herbert Simon, « GPS : A Program that Simulates Human Thought », in E.A. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, McGraw-Hill, 1963.

[16] Peter Jackson, *Introduction to Expert Systems* (3rd edition), Addison Wesley, 1998.

[17] R. Lindsay, B. Buchanan, E. Feigenbaum and J. Lederberg, *Applications of artificial intelligence to organic chemistry : The Dendra projects*, McGraw-Hill, 1980.

[18] Bruce G. Buchanan and Edward H. Shortliffe (eds.), *Rule-Based Expert Systems : The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, 1984.

[19] Barbara von Halle, Larry Goldberg, John Zachman, *The Business Rule Revolution*. Happy About Publishing, 2006.

[20] Joseph C. Giarratano, Gary D. Riley, *Expert Systems : Principles and Programming*, Fourth Edition, Course Technology, 2004.

[21] Lofti Zadeh, *Fuzzy Sets, Information and Control*, Vol. 8, pp. 338-353, 1965.

[22] Judea Pearl, *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*, Morgan Kaufmann, 1988.

[23] Robert G. Cowell, A.Philip Dawid, Steffen L. Lauritzen, David J. Spiegelhalter, *Probabilistic Networks and Expert Systems*, Springer, 1999.

[24] Daphne Koller and Nir Friedman, *Probabilistic Graphical Models : Principles and Techniques*, MIT Press, 2009.

[25] Adnan Darwiche, *Modeling and Reasoning with Bayesian Networks*, Cambridge University Press, 2009.

[26] Judea Pearl, *Causality : Models, Reasoning and Inference*, Cambridge University Press, 2009.

[27] Judea Pearl and Richard E. Korf, « Search techniques », *Annual Reviews Computer Science* 2, pp. 451-467, 1987.

[28] Richard E. Korf, « Depth-first iterative-deepening : an optimal admissible tree search », *Artificial Intelligence* 27(1), pp. 97-109, 1985.

[29] P.E. Hart, N.J. Nilsson, B. Raphael, « A Formal Basis for the Heuristic Determination of Minimum Cost Paths », *IEEE Transactions on Systems Science and Cybernetics SSC* 4(2), pp. 100-107, 1968.

[30] Rina Dechter, Judea Pearl, « Generalized best-first search strategies and the optimality of A* », *Journal of the ACM* 32 (3), pp. 505-536, 1985.

[31] Richard E. Korf, « Linear-space best-first search », *Artificial Intelligence* 62, pp. 41-78, 1993.

[32] Francesca Rossi, Peter van Beek, Toby Walsh (eds.), *Handbook of Constraint Programming*, Elsevier, 2006.

[33] Rina Dechter, *Constraint Processing*, Morgan Kaufmann, 2003

[34] Krzysztof Apt, *Principles of Constraint Programming*, Cambridge University Press, 2003.

[35] David Waltz, Understanding line drawings of scenes with shadows, in : P.H. Winston (ed.), *The Psychology of Computer Vision*, McGrawHill. pp. 19-92, 1975.

[36] R.M. Haralick and G.L. Elliot, « Increasing tree search efficiency for constraint satisfaction problems », *Artificial Intelligence* 14, pp. 263-314, October 1980.

[37] Eugene Freuder, « A sufficient condition for backtrack-free search », *Journal of the ACM* 29, pp. 24-32, March 1982.

[38] Eugene Freuder, « A sufficient condition for backtrack-bounded search », *Journal of the ACM* 32(14), pp. 755-761, 1985.

[39] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, « Optimization by Simulated Annealing », *Science* 220(4598), pp. 671-680, 1983.

[40] Bart Selman and Henry Kautz, Noise « Strategies for Improving Local Search », *Proceedings of AAAI-94*, 1994.

[41] Johan de Kleer and Brian C. Williams, « Diagnosing multiple faults », *Artificial Intelligence* 32(1), pp. 97-130, 1987.

[42] Walter Hamscher, Luca Console and Johan de Kleer, *Readings in Model-based Diagnosis*, Morgan Kaufmann Publishers, 1992.

[43] Nicola Muscettola, Pandu Nayak, Barney Pell and Brian Williams, « Remote agent : To boldly go where no AI system has gone before », *Artificial Intelligence* 103(1-2), August 1998.

[44] J. McCarthy and P. Hayes, « Some philosophical problems from the standpoint of artificial intelligence », in B. Meltzer and D. Michie (eds.), *Machine Intelligence* 4, pp. 463-502, Edinburgh University Press, 1969.

[45] R. Fikes and N. Nilsson, « STRIPS : a new approach to the application of theorem proving to problem solving », *Artificial Intelligence* 2, pp. 189-208, 1971.

[46] H. Levesque, F. Pirri and R. Reiter, « Foundations for the situation calculus », *Electronic Transactions on Artificial Intelligence* 2(3-4), pp. 159-178, 1998.

[47] T. Bylander, « The Computational complexity of propositional STRIPS planning », *Artificial Intelligence* 69, pp. 165-204, 1994.

[48] Daniel S. Weld, « An introduction to least commitment planning », *Artificial Intelligence Magazine* 15(4), pp. 27-61, 1994.

[49] H. A. Kautz and B. Selman, Planning as satisfiability, Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92), pp. 359-363, 1992.

[50] Avrim L. Blum and Merrick L. Furst, « Fast planning through planning graph analysis », *Artificial Intelligence* 90(1-2), pp. 281-300, 1997.

[51] Minh Binh Do and Subbarao Kambhampati, « Planning as constraint satisfaction : Solving the planning graph by compiling it into CSP », *Artificial Intelligence* 132(2), pp. 151-182, 2001.

[52] Malte Helmert, « The Fast Downward Planning System », *Journal of Artificial Intelligence Research* 26, pp. 191-246, 2006.

[53] Hector Geffner, Blai Bonet, *A Concise Introduction to Models and Methods for Automated Planning*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool, 2013.

[54] Christopher M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[55] Kevin P. Murphy, *Machine Learning : A Probabilistic Perspective*, MIT Press, 2012.

[56] Trevor Hastie, Robert Tibshirani et Jerome Friedman, *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*, Springer, 2009.

[57] Ian H. Witten and Eibe Frank, *Data Mining : Practical machine learning tools and techniques*, Morgan Kaufmann, 2005.

[58] Pedro Domingos, *The Master Algorithm : How the Quest for the Ultimate Learning Machine Will Remake Our World*, Basic Books, 2015.

[59] Pat Langley, *Elements of Machine Learning*, Morgan Kaufmann, 1996.

[60] Frank Rosenblatt, « The Perceptron : A Probabilistic Model for Information Storage and Organization in the Brain », Cornell Aeronautical Laboratory, *Psychological Review*, Vol. 65, No. 6, pp. 386-408, 1958.

[61] S. I. Gallant, « Perceptron-based learning algorithms », *IEEE Transactions on Neural Networks* 1(2), pp. 179-191, 1990.

[62] Vladimir Vapnik, *The Nature of Statistical Learning Theory*, Springer-Verlag, 1995.

[63] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabell, « Explanation-Based Generalization : A Unifying View », *Machine Learning* 1, pp. 1-33, 1986.

[64] J.R. Quinlan, « Induction of Decision Trees », *Machine Learning* 1, pp. 81-106, 1986.

[65] Leslie Valiant, « A theory of the learnable », *Communications of the ACM* 27(11), pp. 1134-1142, 1984.

[66] Yoav Freund and Robert E. Schapire, « A decision-theoretic generalization of on-line learning and an application to boosting », *Journal of Computer and System Sciences* 55(1), pp. 119-139, 1997.

[67] Robert E. Schapire et Yoram Singer, « Improved Boosting Algorithms Using Confidence-Rated Predictors », *Machine Learning* 37(3), pp. 297-336, 1999.

[68] P. Long and R. Servedio, « Martingale Boosting, Eighteenth Annual Conference on Computational Learning Theory (COLT) », pp. 79-94, 2005.

[69] A.K. Jain, M.N. Murty et P.J. Flynn, « Data Clustering : A Review », *ACM Computing Surveys* 31(3), pp. 264-323, 1999.

[70] R. Xu et D. Wunsch, « Survey of Clustering Algorithms », *IEEE Transactions on Neural Networks* **16**(3), 2005.

[71] Dempster, A.P. ; Laird, N.M. ; Rubin, D.B. (1977). « Maximum Likelihood from Incomplete Data via the EM Algorithm », *Journal of the Royal Statistical Society*, Series B 39 (1) : 1–38. JSTOR 2984875. MR 0501537.

[72] Jianbo Shi and Jitendra Malik, « Normalized Cuts and Image Segmentation », *IEEE Transactions on PAMI*, Vol. 22, No. 8, Aug 2000.

[73] Ng, A., Jordan, M., and Weiss, Y., « On spectral clustering : analysis and an algorithm », in T. Dietterich, S. Becker, and Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems* 14, pp. 849 – 856, MIT Press, 2002.

[74] Ulrike von Luxburg, « A Tutorial on Spectral Clustering », *Statistics and Computing*, 17 (4), 2007.

[75] Zhu, Xiaojin ; Goldberg, Andrew B., *Introduction to semi-supervised learning*, Morgan & Claypool, 2009.

[76] Ian Goodfellow, Yoshua Bengio, et Aaron Courville, *Deep Learning*, MIT Press, 2016.

[77] John Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, 1975.

[78] David E Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Kluwer Academic Publishers, Boston, 1989.

Index

A* (algorithme), 155
A-box, 73
abduction, 6, 145
    par raisonnement incertain, 213
agglomération, 307
    (méthode par), 308
algorithme(s)
    ADABOOST, 290
    de clustering, 306
    de maximisation de l'espérance, 316
    de Waltz, 187
    génétiques, 336
    ID3, 281
    k-médoïds, 313
    k-means, 312
`AlphaGo`, 335
apprentissage, 14
    automatique, 257
    bio-inspiré, 331
    d'arbres de décision, 280
    de classifications structurées, 277
    de disjonctions, 278
    de exceptions, 278
    de sous-classes, 305
    non supervisé, 257, 305
    par généralisation, 262
    par recherche, 262
    par spécialisation, 262
    semi-supervisé, 317
    supervisé, 257
arbre(s), 147
    couvrant, 310
    couvrant de poids minimal, 310
    de classification, 280
    de décision, 280
assertion (algorithme d'), 88
attitudes, 87
attributs numériques, 264
axiomes cadres, 228, 236

back-propagation, 333
backjumping, 178
    (conflict-directed), 179
backtrack, 177
bagging, 289
base de connaissances, 35
beam search, 158
biais, 261
Big Data, 11
boosting, 289
branch-and-bound, 155, 193
Breadth First Search (BFS), 149
business rules, 46
buts multiples, 236

cadre(s), 68, 229
calcul
    de situations, 228
    des prédicats, 21, 38
    des prédicats du 1er ordre, 25
candidats minimaux, 215
causalité, 118
chaînage, 34, 35
    arrière, 35, 79
    avant, 35, 36
chromosome, 337
circonscription, 217
classe, 67
classification(s), 257
    Bayesienne, 272
    disjonctives (apprentissage de), 277
    hiérarchiques, 70
    probabiliste, 272

structurées (apprentissage de), 277
clause, 31
de Horn, 34
clustering, 125, 306
(algorithmes de), 306
de partitionnement, 312
hiérarchique, 307
probabiliste, 314
spectral, 311
clusters, 305
cohérence, 87
(algorithme de maintenance de la), 88
(système de maintenance de la) (SMC), 87
combinaison (opérateur de), 337
complete-link (méthode par), 308
concepts, 67, 70, 71
(satisfiabilité de), 72
condition de terminaison, 149
conflict-directed backjumping, 179
conflit, 215
connaissance(s), 3, 12, 21, 25
(représentation structurée des), 67
consistance, 72
des arcs, 180, 189
des nœuds, 189
partielle, 174
contraintes, 171
binaires, 174
globales, 190
contrefactuelle, 117
couche de neurones cachés, 331
croyance, 116
cycles, 153
(détection de), 154

déduction, 6
deep learning, 335
deep neural net, 335
défaillances, 219
démonstrateurs de théorèmes, 44
démonstrations indirectes, 90
dendrogramme, 307
dépendance conditionnelle, 120, 125
Depth First Search (DFS), 149
Depth-Limited Search (DLS), 152
descente de gradient stochastique, 333
détection de cycles, 154
DFS-BB (algorithme), 155
diagnostic, 209
basé sur la consistance, 214
division, 307
(méthode par), 309
domaine, 171
dropout, 335
dynamic value ordering (DVO), 181

élagage, 288
enregistrement, 68
entités, 67
entropie, 282
environnements, 80
exceptions, 278
expectation maximisation, 316
explication, 83

facteurs de certitude, 114
factorisation, 44
filtrage, 40
filtre à spam, 273
fonction
d'activation, 332
de Skolem, 39
redresseur, 333
sigmoïde, 333
forme normale, 31
formules bien formées, 25
forward checking, 179
frames, 68
frontières de décision, 264

General Problem Solver, 8
generate-and-test, 173
Gibbs sampling, 128
gradient disparaissant, 335
graphe de recherche, 147

héritage
multiple, 70

structuré, 67, 70
heuristique(s), 174
min-conflicts, 182
hill-climbing, 182, 338
Horn (clause de), 34

ID3 (algorithme), 280
incertitude, 111
indépendance conditionnelle, 118
induction, 6, 257
de modèles simples, 259
inférence, 6, 21, 28
(algorithmes d'), 31
(moteur d', 21
(moteurs d'), 31
abductive, 121
monotone, 87
non monotone, 86, 87
instances, 67
(vérification d'), 72
intelligence, 1
iterative deepening, 153
A^*, 158

justifications, 88

(k-1)-consistant, 188
k-médoïds (algorithme), 313
k-means (algorithme), 312
kernel function, 267

LASSO, 270
least commitment, 237
LISP, 15
liste de décision, 277, 279
logique(s), 25
$\mathcal{AL}$, 75
$\mathcal{FL}^-$, 75
$\mathcal{SHIN}$, 75
$\mathcal{SHIQ}$, 75
des défauts, 87
des prédicats, 4
descriptives, 71
floue, 113
monotone, 86
propositionnelle, 36
lookahead, 180

macro-opérateurs, 233
Markov Chain Monte Carlo (MCMC), 128
martingale boosting, 291
max-degree ordering, 181
maximisation de l'espérance (algorithme de), 316
mélange de Gaussiennes, 315
memory-bounded A^*, 158
méta-règles, 82
min-conflicts, 182
min-width ordering, 181
minimal spanning tree (MST), 310
modèle(s)
de défaillances, 219
non paramétriques, 257
paramétriques, 257
simple, 259
modus ponens, 28, 34
monde
clos, 7
des blocs, 228
moteur d'inférence, 21, 31, 36
mutation (opérateur de), 337
MYCIN, 83

naive Bayes, 122, 272
négation, 85
neurone artificiel, 332
nœud(s)
final, 149
initiaux, 149
nogood, 89

opérateurs avec disjonctions, 242
optimisation sous contraintes, 193
ordering
(dynamic value) (DVO), 181
(max-degree), 181
(min-width), 181
ordonnancement des variables, 181
overfitting, 286
OWL (*ontology web language*), 76

PAC (théorie), 286
pattern matching, 40
perceptron, 265

(règle du), 265
plan(s), 227
non linéaire, 237
planification, 13, 227
hiérarchique, 242
non linéaire, 237
par chaînage arrière, 230
PLANMIN, 234
préférences, 194
preuve(s)
conditionnelles, 89
indirecte, 34, 44
logique, 28
problème
de cadres, 229
de satisfaction de contraintes (PSC), 172
des cadres, 236
projection, 175
propagation(s), 174
de contraintes, 186
de labels, 186
de valeurs, 186
propositions de mesures, 217
propriétés, 67
Python, 15

qualité de l'apprentissage, 286
quantificateur
existentiel, 38
universel, 38
quantification, 38

radial basis function, 268
raisonnement
basé sur des modèles, 146
basé sur des règles, 79
basé sur modèles, 145
Reason Maintenance Systems, 87
recherche
(graphes de), 147
(résolution de problèmes par), 147
en largeur-d'abord, 149
en profondeur limitée, 152
recherche en profondeur-d'abord, 149
recuit simulé, 182
règle(s)
(raisonnement basé sur des), 79
d'équivalence, 26
d'inférence, 28
du perceptron, 265
régression, 257, 268
logistique, 271
régularisation, 270
relaxation de valeurs, 186
représentation structurée des connaissances, 67
réseau(x)
à héritage structuré, 70
bayésiens, 116, 118
de contraintes, 174
de neurones artificiels, 331
dual, 175
multicouche, 332
sémantiques, 68, 70
résolution, 31, 43
binaire, 44
de problèmes par recherche, 147
retour-arrière, 150
rétraction (algorithme d'), 88
rétropropagation, 333

SAT (problème de la satisfiabilité), 192
satisfaction de contraintes, 171
satisfiabilité, 192
de concepts, 72
similarité
non-transitive, 306
transitive, 306
single-link (méthode par), 308
SIRI, 11
situation, 228
Skolem (fonction de), 39
slots, 68
soft constraints, 194
solution optimale, 154
sous-buts, 80
sous-classes, 305
(apprentissage de), 305

spam (filtre à –), 273
spanning tree, 310
STRIPS, 228
subsumption, 72
support vector machine (SVM), 267
support vectors, 267
surapprentissage (*overfitting*), 270
symptôme, 215
système(s)
    à base de connaissances, 19
    de maintenance de la cohérence (SMC), 87
    défaillant, 209
    experts, 79

T-box, 73
tables triangulaires, 233
théorèmes (démonstrateurs de), 44
théorie
    de l'information, 217
    PAC, 286
traitement de l'information incertaine, 111
transformation en déduction, 212

unification, 40
unit propagation, 192

vanishing gradient, 335
variable latente, 316
vecteur
    de Fielder, 312
    de support, 267
vérification d'instances, 72
voitures autonomes, 130

WATSON, 11