Satisfaction de Contraintes

Boi Faltings

Laboratoire d'Intelligence Artificielle
boi.faltings@epfl.ch
http://moodle.epfl.ch/

Faiblesse des algorithmes de recherche

- Algorithmes de recherche très généraux, mais...
- ...conduisent à une explosion combinatoire.
- Peut-on définir une cadre restreint qui permet des méthodes plus efficaces?
- Oui: la satisfaction de contraintes/satisfiabilité.

Satisfaction de contraintes

Un grand nombre de problèmes pratiques s'expriment en terme d'une *satisfaction de contraintes*, par exemple dans les domaines de:

- l'ordonnancement et la planification de tâches: le but est de trouver un ensemble d'actions qui respecte les contraintes décrivant le but à atteindre et les moyens à disposition.
- la conception ou la configuration: le but est de trouver un ensemble de composants et de connections respectant toutes les contraintes de fonctionnalité.
- la vision: le but est de trouver une interprétation consistante avec les observations.

Formulation d'un problème de satisfaction de contraintes

Etant donné:

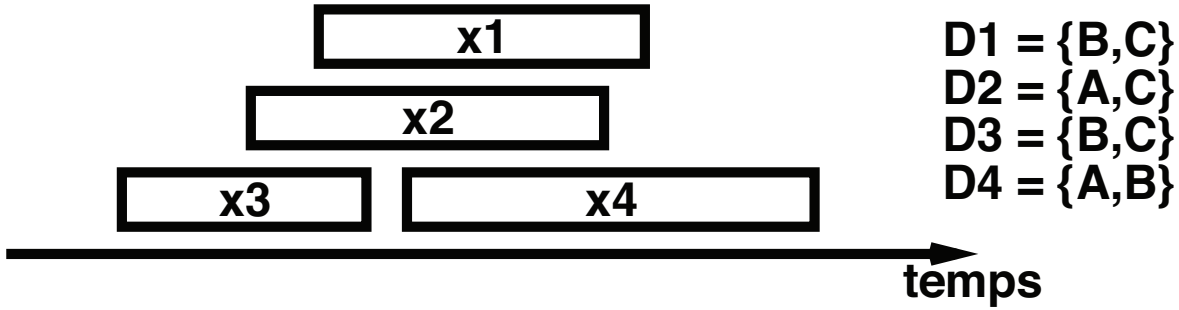
- Variables $X = \{x_1, x_2, ..., x_n\}$
- Domaines $D_1, D_2, ..., D_n$ des variables
- Contraintes $C = \{c_1(x_k, x_l, ...), c_2, ..., c_m\}$,
  chacune n'admettant que certaines combinaisons de valeurs

Trouver:
Des solutions: $\{x_1 = v_k, x_2 = v_l, ..., x_n = v_o\}$
de sorte que toutes les contraintes soient satisfaites.

Nous considérons des variables *discrètes*: les domaines $D_1, ..., D_n$ sont finis.

Exemple d'un PSC: allocation de ressources

But: effectuer les tâches T1 à T4 avec 3 ressources:



Formulation comme PSC:

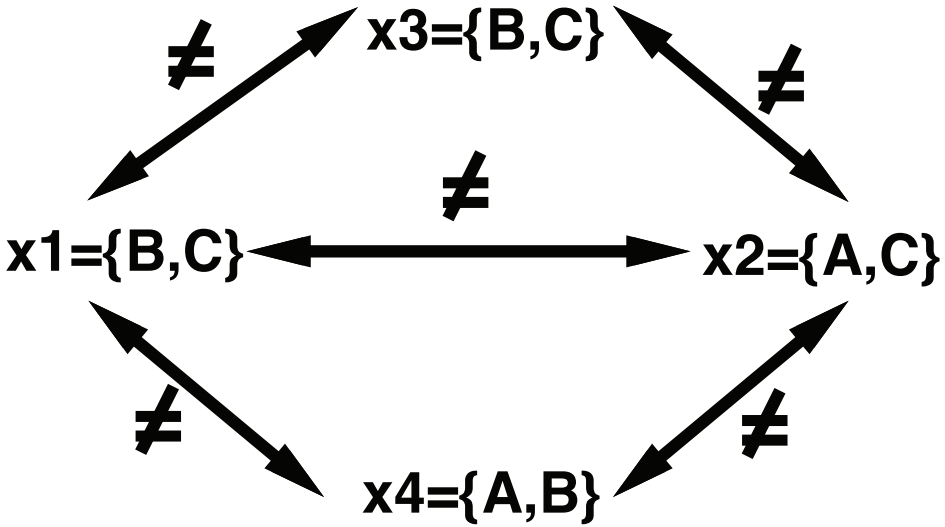
- Variables $x_1, x_2, ..., x_4$ correspondants aux tâches (valeur = *une* ressource)
- Domaines = ressources $D_1 = \{B, C\}, ...$
- Contraintes = deux tâches se chevauchant dans le temps ne peuvent être effectuées par la même ressource.

Réseaux de contraintes

Un problème de satisfaction de contraintes *binaires* peut être représenté comme un graphe:

- noeuds $\simeq$ variables
- arcs $\simeq$ contraintes

Exemple: allocation de ressources



Contraintes non-binaires

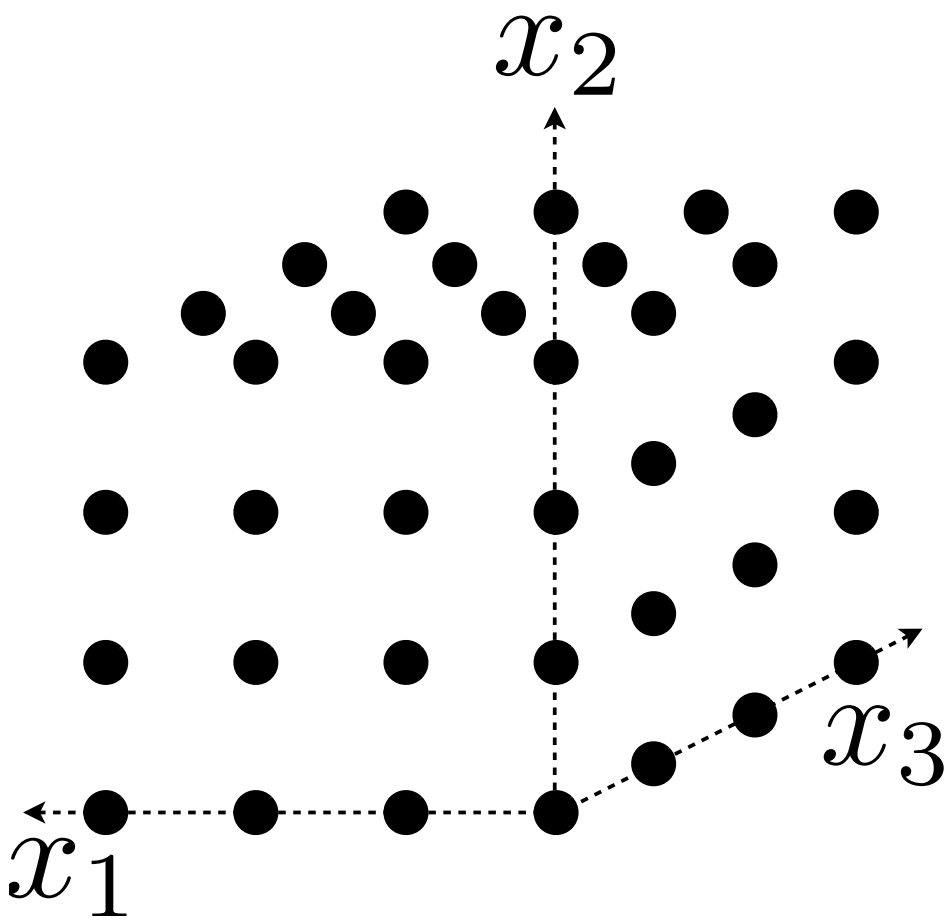
- La plupart des travaux traitent des contraintes *binaires* s'avérant plus adaptées à la propagation.
- Pour traiter des problèmes avec des contraintes non-binaires, on les transforme en contraintes binaires par une des méthodes suivantes:
  1. projection
  2. variables supplémentaires
  3. graphe dual

Solution de PSC

- L'algorithme le plus général et le plus simple pour la résolution de PSC discrets est celui de *generate-and-test*: essayer toutes les combinaisons des valeurs admissibles pour les variables en retenant celles qui respectent toutes les contraintes.
- Exemple: allocation de ressources: (x_1, x_2, x_3, x_4)

(BABA)	(BABB)	(BACA)	(BACB)
(BCBA)	(BCBB)	(BCCA)	(BCCB)
(CABA)	**(CABB)**	(CACA)	(CACB)
(CCBA)	(CCBB)	(CCCA)	(CCCB)

Espace des nœuds de recherche



Complexité de la solution

- La complexité d'une solution fournie par generate-and-test est en général exponentielle par rapport au nombre de variables. Or, un PSC typique implique un grand nombre de variables.
- Exemple: si $|D| = 4, |X| = 20$: 1'099'511'627'776 combinaisons à examiner!

$\Rightarrow$ on cherche des méthodes plus efficaces applicables à certaines classes de PSC.

- Ces méthodes sont en général basées sur le caractère local des contraintes: chaque contrainte n'implique que très peu de variables.

Algorithmes pour résoudre des PSC

Méthodes basées sur la recherche systematique:

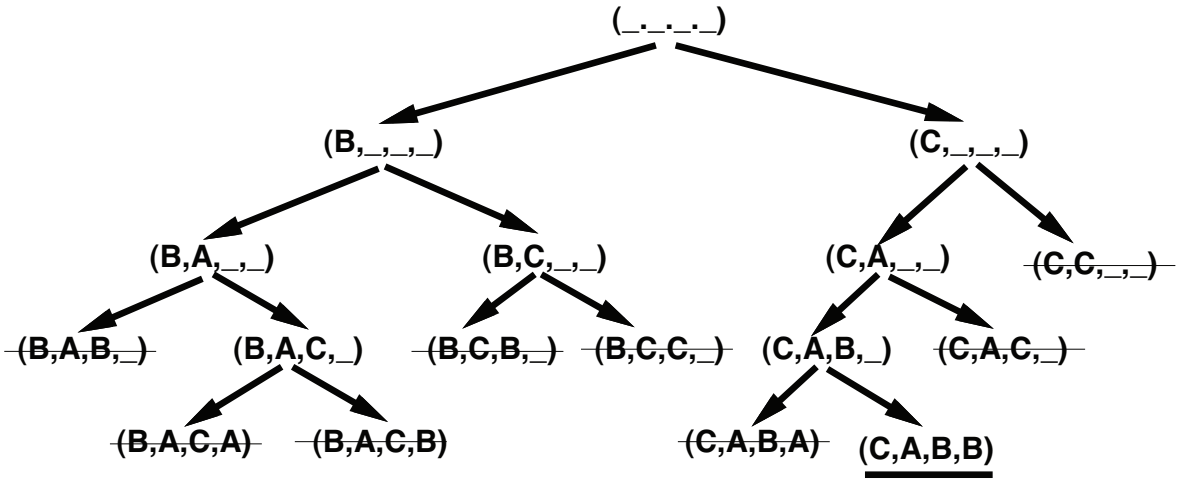
- backjumping
- forward checking
- lookahead

Méthodes basées sur la recherche locale:

- recuit simulé
- GSAT
- min-conflicts

Exploiter le caractère local d'un PSC

La recherche en profondeur d'abord permet d'effectuer des retour-arrières dès qu'une solution partielle ne respecte pas toutes les contraintes:



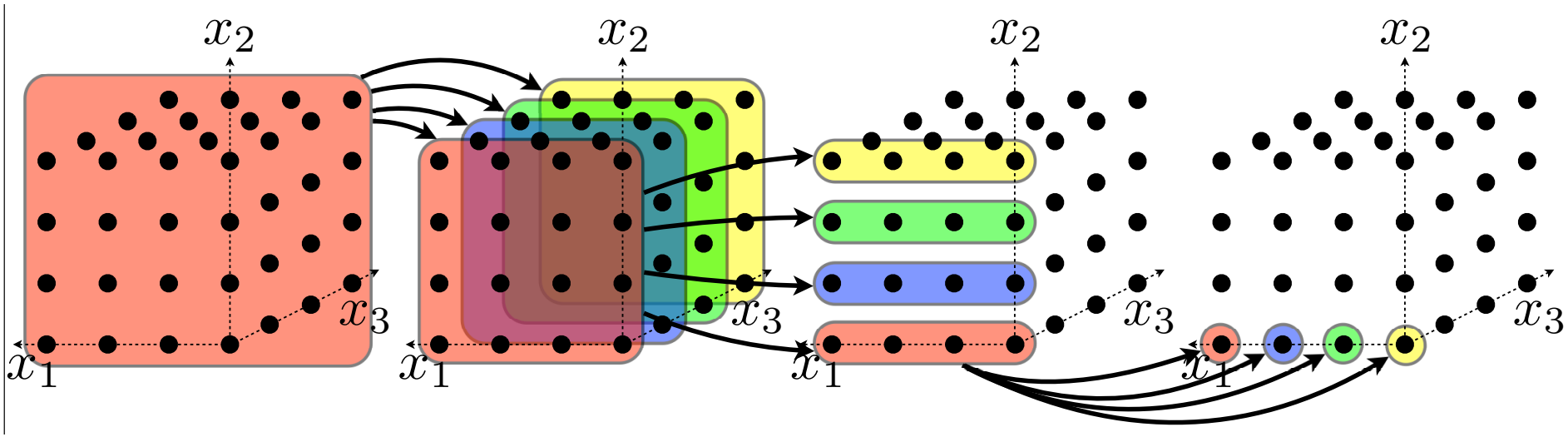
$\Rightarrow$ permet une solution plus efficace, mais toujours de complexité exponentielle.
Il existe des heuristiques qui réduisent la complexité de manière importante!

Résolution d'un PSC par recherche

Formulation comme problème de recherche:

- noeud de recherche = instantiation de variables
  $x_1 = v_1, x_2 = v_2, ..., x_k = v_k$
- fonction de successeur = instantation de la variable
  $x_{k+1} = v_{k+1}$ de manière à respecter
  toutes les contraintes avec $x_1, ..., x_k$.
- noeud initial = instantiation vide
- noeud but = instantiation de toutes les variables $x_1, ..., x_n$

Résolution d'un PSC par recherche systématique



Depth-First Search: DFS

x = tableau de n variables, rempli jusqu'à k
d = domaines des variables

```
Function DFS(x,d,k)
if k ≥ n then
  return x
else
  for v ∈ d[k+1] do
    consistent ← true
    for i ← 1 to k do
      if ¬ consistent(v,x[i],C(i,k+1)) then consistent ←
      false
    if consistent then
      x[k+1] ← v
      rest ← DFS(x,d,k+1)
      if rest ≠ :echec then return rest
return :echec
```

Exemple: allocation de ressources

Variables:

$x_1 \in \{B, C\}$
$x_2 \in \{A, C\}$
$x_3 \in \{B, C\}$
$x_4 \in \{A, B\}$

Contraintes:

$C(x_1, x_2) : \{(B, A), (B, C), (C, A)\}$
$C(x_1, x_3) : \{(B, C), (C, B)\}$
$C(x_1, x_4) : \{(B, A), (C, B), (C, A)\}$
$C(x_2, x_3) : \{(A, B), (A, C), (C, B)\}$
$C(x_2, x_4) : \{(A, B), (C, A), (C, B)\}$

Solution par DFS

`d = [(B,C),(A,C),(B,C),(A,B)]`

inégalité entre toutes les x sauf x_3/x_4

Etape	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!

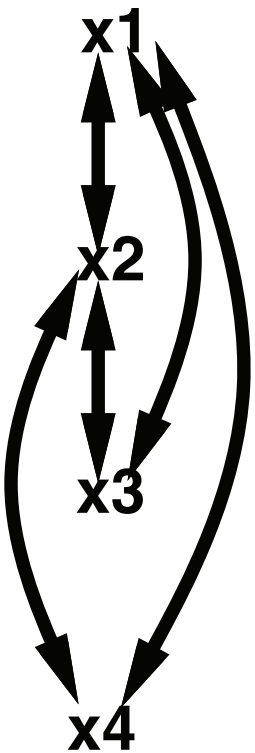
Améliorer la performance

La recherche peut être rendue plus efficace en optimisant:

- la manière de revenir en arrière d'un conflit.
- les valeurs considérées.
- l'ordre d'instantiation des variables.

Backjumping

Si l'assignation à x_{k+1} échoue, il faut changer au moins une variable qui y est liée par une contrainte.



- backjumping: revenir à la dernière variable qui a une contrainte avec x_{k+1}. (ex: $x_4 \rightarrow x_2$)
  (évite le pas 5).
- Conflict-directed backjumping: revenir à la dernière variable qui avait un *conflit* avec x_{k+1}. (ex: $x_4 \rightarrow x_1$)

Forward checking

Eviter des valeurs qui ne laissent plus aucune possibilité d'instantiation consistante pour d'autres variables.

- Ajouter un label `l[i]` à chaque variable.
- Initialement: label = domaine
- Forward checking:

  à chaque instantiation, éliminer du label des variables non-instantiées toutes les valeurs inconsistantes avec l'instantiation.

  Label vide $\Rightarrow$ instantiation non admise, backtrack
- Lookahead:

  Processus itératif: vérifier aussi entre toute paire de variables non-instantiés.

Exemple: Forward Checking

`d = [(B,C),(A,C),(B,C),(A,B)]`

inégalité entre tous les x sauf x_3/x_4

k	x[1]	x[2]	x[3]	x[4]	I[1]	I[2]	I[3]	I[4]	
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

Exemple: Lookahead

`d = [(B,C),(A,C),(B,C),(A,B)]`

inégalité entre toutes les x sauf x_3/x_4

k	x[1]	x[2]	x[3]	x[4]	I[1]	I[2]	I[3]	I[4]	
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

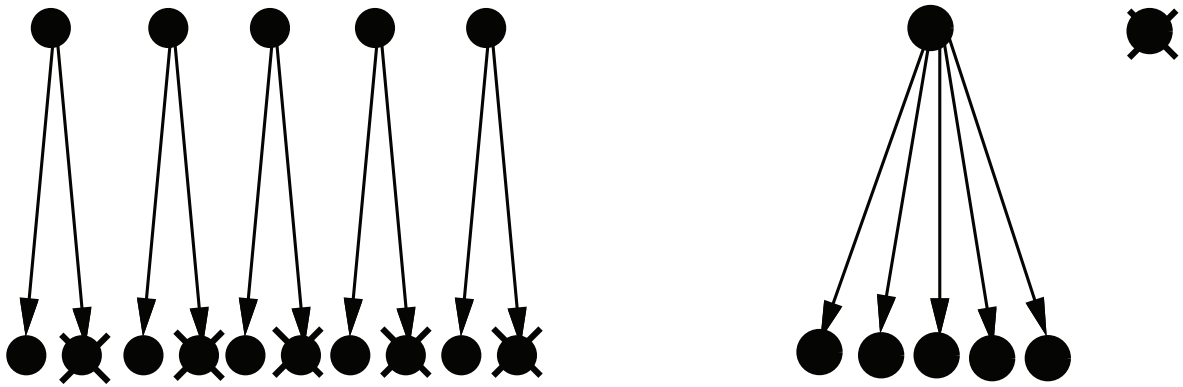
Ordonner les variables

Influencer la recherche par l'ordre d'assignation des variables.
Heuristiques d'ordre:

- dynamic variable ordering (DVO): prendre la variable dont le label est le plus petit (combinaison avec forward checking/lookahead)
- min-width ordering: prendre la variable qui a des contraintes avec le plus petit nombre de variables encore ouvertes.
- max-degree ordering: prendre la variable la plus connectée dans le PSC original.

Justification (DVO)

DVO réduit l'espace de recherche:



First-fail:
arbre plus petit en évitant de créer beaucoup de branches!

Exemple: ordonnancement de variables par DVO

e.	x[1]	x[2]	x[3]	x[4]	I[1]	I[2]	I[3]	I[4]	
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

Combinaison DVO + tiebreaking avec min-width

e.	x[1]	x[2]	x[3]	x[4]	I[1]	I[2]	I[3]	I[4]	
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

$\Rightarrow$ on ne peut pas faire mieux!

Comparaison des heuristiques

- DFS très peu efficace
- Backjumping mieux, mais visite au moins autant de noeuds que forward checking
- Forward checking/lookahead plus fort
- Forward checking + ordonnancement des variables
  $\Rightarrow$ toute forme de backjumping est inutile

Algorithmes de consistance

En pratique, il est important de connaître d'avance le temps requis pour la solution d'un PSC
Possible dans le cas de:

1. certaines topologies du graphe de contraintes
2. certaines topologies des contraintes mêmes

Pour cela, on applique des algorithmes de consistance.

Algorithmes de consistance partielle

Idée de la consistance = solution en 2 temps:

1. éliminer des valeurs ou combinaisons de valeurs ne pouvant respecter toutes les contraintes.
2. effectuer la recherche dans l'espace réduit

Exemple:
$x_1 = B$ ne peut jamais faire partie d'une solution!
$\Rightarrow$ l'éliminer d'avance de la recherche

La consistance des noeuds et arcs

- Considérez deux variables x_1 et x_2 ainsi que la contrainte $C(x_1, x_2)$ les liant.
- Une valeur de $v_1 \in l_1$ ne peut faire partie d'une solution du PSC que s'il existe au moins une valeur $v_2 \in l_2$ telle que $C(v_1, v_2)$ soit respectée.
  $\Rightarrow$ on peut donc éliminer toutes les valeurs de l_1 et l_2 qui ne respectent pas la contrainte.
- Par application itérative sur toutes les contraintes, on obtient un réseau qui satisfait la consistance des arcs.

Propagation locale

- Contraintes *binaires* (2 variables) et domaines discrets
- Chaque x_i porte un *label* l_i = ensemble de valeurs
  $\Rightarrow$ algorithme de Waltz:
- appliquer un opérateur de raffinement à chaque contrainte jusqu'à ce qu'il n'y ait plus de changement
- Garantit la consistance des arcs

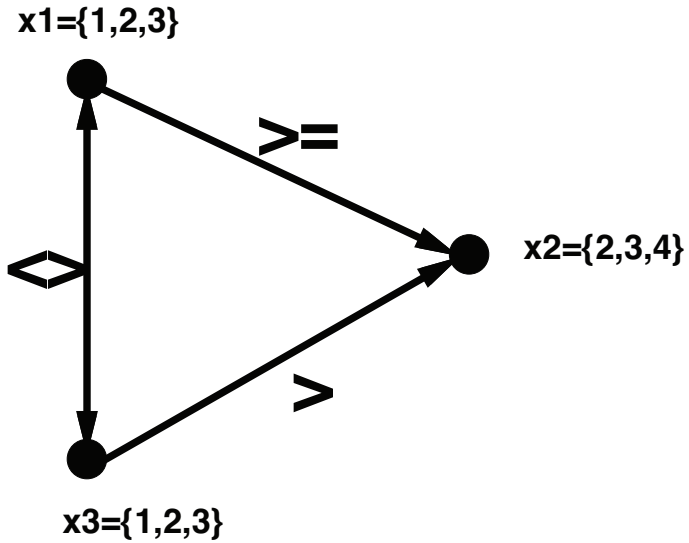
L'opérateur de raffinement

Pour réviser la contrainte entre x_i et x_j:

Function REVISER (i,j)
modifiée $\leftarrow$ **faux**
for chaque x $\in l_i$ **do**
if aucun y $\in l_j$ tel que $C(x, y)$ **then**
$l_i \leftarrow l_i \setminus x$
modifiée $\leftarrow$ **vrai**
return modifiée

Appliquer `REVISER(i,j)` à tous les combinaisons de variables `i` et `j` jusqu'à ce que le résultat soit **faux** pour tous les combinaisons.

Exemple: algorithme de Waltz



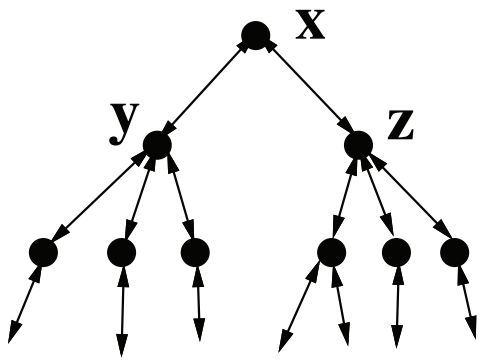
opération	l_1	l_2	l_3
c(1,2)	$\{\underline{1}, 2, 3\}$	$\{2, 3, 4\}$	$\{1, 2, 3\}$
c(2,1)	$\{\not{1}, 2, 3\}$	$\{2, 3, \underline{4}\}$	$\{1, 2, 3\}$
c(2,3)	$\{\not{1}, 2, 3\}$	$\{2, \underline{3}, \not{4}\}$	$\{1, 2, 3\}$
c(3,2)	$\{\not{1}, 2, 3\}$	$\{2, \not{3}, \not{4}\}$	$\{\underline{1}, \underline{2}, 3\}$
c(1,3)	$\{\not{1}, 2, \underline{3}\}$	$\{2, \not{3}, \not{4}\}$	$\{\not{1}, \not{2}, 3\}$
c(3,1)	$\{\not{1}, 2, \not{3}\}$	$\{2, \not{3}, \not{4}\}$	$\{\not{1}, \not{2}, 3\}$

La complexité de l'algorithme de Waltz

- Considérons e contraintes, n variables dont les domaines ont la taille maximale d.
- A chaque itération, `REVISER` est appliquée au plus $2e$ fois, une fois dans chaque direction.
- Pour que l'itération ne s'arrête pas, il faut enlever au moins une valeur $\Rightarrow$ au plus $n \cdot d$ itérations.
  $\Rightarrow$ complexité $= O(e \cdot n \cdot d)$
- Algorithme optimisé: $O(n^2 d)$

Solutions sans retour-arrière

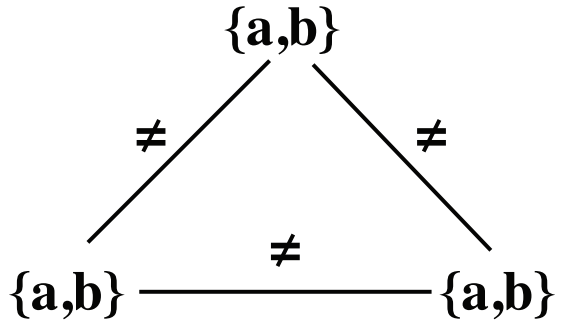
Dans le cas où le réseau de contraintes est un arbre (pas de cycles), la consistance des arcs garantit qu'une solution puisse être trouvée de manière très efficace, sans retour-arrière:



1. assigner n'importe quelle valeur à x.
2. la consistance des arcs garantit que l'on peut assigner des valeurs consistantes à y et z.
3. appliquer récursivement aux couches suivantes.

Cycles

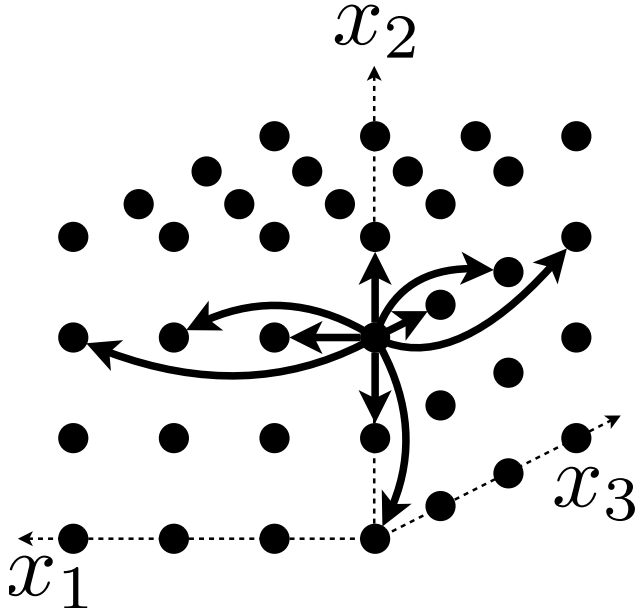
La présence de cycles rend la consistance des arcs beaucoup moins utile. Par exemple, le réseau suivant satisfait la consistance des arcs, mais n'admet aucune solution:



$\Rightarrow$ il faut définir des niveaux de consistance plus élevés.

Méthodes itératives (recherche locale)

Idée:
commencer par une assignation initiale, ensuite chercher des modifications locales qui réduisent le nombre de contraintes violées.



Hill-climbing

- Correspond à un "hill-climbing": continuellement augmenter le nombre de contraintes satisfaites.
- 2 versions:
  - déterministe (min-conflicts/GSAT)
  - probabiliste (recuit simulé)
- La performance est souvent très aléatoire: dépend du choix des valeurs initiales et de la séquence de modifications.

Min-conflicts

Idée:

changer l'assignation de la variable qui réduira le plus le nombre total de conflits.

- Descente dans la direction du plus fort changement: descente du gradient.
- La procédure tombera forcement dans des minima locaux, mais souvent ce sont déjà des solutions de bonne qualité.
- Version heuristique pour la recherche incrémentale: prendre la valeur qui aura le moins de possibilités de conflits avec de futures instantiations.

Algorithme (Min-conflicts)

X = variables
V = valeurs
C = contraintes

```
Function min-conflicts(X,V,C)
for i ← 1 to max-tries do
    V ← assignation aléatoire
    for j ← 1 to max-steps do
        nconf ← check(V,C)
        if nconf = 0 then
            return V
        else
            trouver k tel que changer v[k] donne un nombre minimal de conflits
            changer v[k]
retourner solution partielle V
```

Exemple: min-conflicts

Assignation initiale: (x1 = B, x2 = A, x3 = B, x4 = A)
$\Rightarrow$ 2 conflits: c(x1,x3) et c(x2,x4)

chang.	conflits	nconf
x1 $\rightarrow$ C	c(x2,x4)	1
x2 $\rightarrow$ C	c(x1,x3)	1
x3 $\rightarrow$ C	c(x2,x4)	1
x4 $\rightarrow$ B	c(x1,x3),c(x1,x4)	2

accepter (x1 $\rightarrow$ C): (x1 = C, x2 = A, x3 = B, x4 = A)
$\Rightarrow$ 1 conflit: c(x2,x4)

chang.	conflits	nconf
x1 $\rightarrow$ B	c(x1,x3), c(x2,x4)	2
x2 $\rightarrow$ C	c(x1,x2)	1
x3 $\rightarrow$ C	c(x1,x3),c(x2,x4)	2
x4 $\rightarrow$ B	-	0

accepter (x4 $\rightarrow$ B) $\Rightarrow$ solution:
(x1 = C, x2 = A, x3 = B, x4 = B)

Exemple 2: min-conflicts

Assignation initiale: (x1 = B, x2 = A, x3 = B, x4 = A)
$\Rightarrow$ 2 conflits: c(x1,x3) et c(x2,x4)

chang.	conflits	nconf
x1 $\rightarrow$ C	c(x2,x4)	1
x2 $\rightarrow$ C	c(x1,x3)	1
x3 $\rightarrow$ C	c(x2,x4)	1
x4 $\rightarrow$ B	c(x1,x3),c(x1,x4)	2

accepter (x2 $\rightarrow$ C): (x1 = B, x2 = C, x3 = B, x4 = A)
$\Rightarrow$ 1 conflit: c(x1,x3)

chang.	conflits	nconf
x1 $\rightarrow$ C	c(x1,x2)	1
x2 $\rightarrow$ A	c(x1,x3),c(x2,x4)	2
x3 $\rightarrow$ C	c(x2,x3)	1
x4 $\rightarrow$ B	c(x1,x3),c(x1,x4)	2

aucune amélioration possible: minimum local!

Recuit simulé

Problème:

le "hill-climbing" conduit à des minima locaux

$\Rightarrow$ on peut manquer la solution

Idée:

- parfois faire un changement non-optimal
- Algorithme inspiré de la physique: solidification de verres

Algorithme de recuit simulé

Function recuit(X,V,C)
V ← assignation aléatoire
T[max-steps] ← plan de réduction de "température"
for j ← 1 to max-steps **do**
    nconf ← check(V,C)
    if nconf = 0 **then** return V
    V' ← V avec une valeur v[k] changée aléatoirement
    if check(V',C) < check(V,C) **then**
        V ← V'
    else
        if random(0..1)<T[j] **then** V ← V'
return solution partielle V

Tableau T = liste de probabilités décroissantes

Exemple: recuit simulé

Supposons T = (0.7,0.5,0.3,0.1,0.01,0.001)
Assignation initiale: (x1 = B, x2 = A, x3 = B, x4 = A)
$\Rightarrow$ 2 conflits: c(x1,x3) et c(x2,x4)

chang.	conflits	mieux?	accepter?	assign.
x2 $\rightarrow$ C	c(x1,x3)	oui	oui	B,C,B,A
x4 $\rightarrow$ B	c(x1,x3), c(x1,x4)	non	oui	B,C,B,B
x1 $\rightarrow$ C	c(x1,x2)	oui	oui	C,C,B,B
x3 $\rightarrow$ C	c(x1,x2), c(x1,x3), c(x2,x3)	non	non	C,C,B,B
x2 $\rightarrow$ A	-	oui	oui	C,A,B,B

Satisfaction et Optimisation

- Souvent, il faut trouver une solution *optimale*.
- Optimalité = somme de critères:
  $\Rightarrow$ détruit la localité
  $\Rightarrow$ pas toutes les techniques sont applicables.
- Optimalité = max/min de critères:
  $\Rightarrow$ localité maintenue
  $\Rightarrow$ pratiquement toutes les techniques s'appliquent.
- Méthodes itératives plus simples à adapter.

Satisfiabilité

Variante de la satisfaction de contraintes:

- variables Booléennes, avec domaines vrai/faux.
- contraintes = clauses (disjonctions) logiques.
- solution = assignation qui rend toutes les clauses vrai (SAT)
- solution = assignation qui maximise le nombre de clauses qui sont vraies (Max-SAT).
- il existe des solvers très efficaces.

Application: Gestion de production d'automobiles

Usine Nissan de Sunderland (UK): deux chaines de production pour deux types de voitures
Alternatives pour produire un troisième type de voiture:

- construire une troisième chaine de production
- intercaler la production sur les chaines existantes

Les méthodes classiques (R.O.) ne permettaient pas de modéliser les contraintes relevant de la production de plusieurs modèles sur une même chaine

Solution par contraintes

La programmation par contraintes a permis de modéliser correctement le problème
Résultats:

- économie de la troisième ligne de production.
- augmentée la production de 236'000 à 337'000 voitures sans nouveaux équipements.
- planification mieux respectée: le pourcentage de voitures produites selon le plan a passé de 3% à 95%.

Résumé

PSC = formalisme général de modèlisation de problèmes abductifs
Résolution de PSC par recherche:

- Backjumping/forward checking/lookahead
- Ordonnancement de variables
- Solutions itératives

Techniques de consistance