Algorithmes de Recherche

Boi Faltings

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

Exemple: diagnostic d'un circuit électrique

- interrupteur connecté à une ampoule:
  $\texttt{enclenché(int)} \wedge \neg\ \texttt{défectueux(amp)} \Rightarrow \texttt{allumé(amp)}$

$\Rightarrow$ Règle diagnostic "expert":
$\texttt{enclenché(int)} \wedge \texttt{éteint(amp)} \Rightarrow \texttt{défectueux(amp)}$

- Supposons qu'on découvre que l'interrupteur peut également tomber en panne $\Rightarrow$ nouvelle règle:
  $\texttt{enclenché(int)} \wedge \texttt{éteint(amp)} \Rightarrow \texttt{défectueux(int)}$
- Problème:
  $\texttt{enclenché(int)} \wedge \texttt{éteint(amp)} \vdash$
  $\texttt{défectueux(amp)}$ **et** $\texttt{défectueux(int)}$

  mais juste une des deux doit être vraie!
- Contredit la monotonicité de la déduction!

Monotonicité de la déduction

- Une inférence ne peut pas être falsifiée par de nouvelles connaissances.

$\Rightarrow$ la déduction est monotone: on augmente continuellement la base de connaissances.

- Ne convient pas pour modéliser le diagnostic: une fois que la cause est trouvée, tous les autres candidats ne sont plus valables.

Inférences logiques

Considérez:

a) (prémisse) `oiseau(Tweety)`

b) (conclusion) `vole(Tweety)`

c) (modèle) $(\forall$ `x)` `oiseau(x)` $\Rightarrow$ `vole(x)`

3 Types d'inférences:

- **déduction**: a), c) $\rightarrow$ b)
- **induction**: a), b) $\rightarrow$ c)
- **abduction**: b), c) $\rightarrow$ a)

L'abduction n'est valable que sous l'hypothèse d'un monde clos:

Il n'existe pas de règles qui n'ont pas été considérées par l'algorithme.

Meilleure formulation: diagnostic = abduction

Modèle du circuit:

1. $\texttt{enclenché(int)} \wedge \neg \texttt{défectueux(amp)} \wedge \neg \texttt{défectueux(int)} \Rightarrow \texttt{allumé(amp)}$
2. $\texttt{défectueux(amp)} \Rightarrow \texttt{éteint(amp)}$
3. $\texttt{défectueux(int)} \Rightarrow \texttt{éteint(amp)}$

Diagnostic: *expliquer* $\texttt{éteint(amp)}$ par abduction utilisant 2. et 3.

Autres problèmes abductifs

L'abduction construit des hypothèses des causes qui produisent un certain effet:

- diagnostic: observations $\Rightarrow$ éléments défectueux
- configuration: spécifications $\Rightarrow$ composants
- ordonnancement: contraintes $\Rightarrow$ horaire
- planification: buts à atteindre $\Rightarrow$ opérations à effectuer

Element central de l'abduction: *recherche* d'une solution.

Résolution de problèmes par recherche

But d'un algorithme de recherche:

trouver une solution dont l'évaluation remplit des critères de succès

Par exemple:

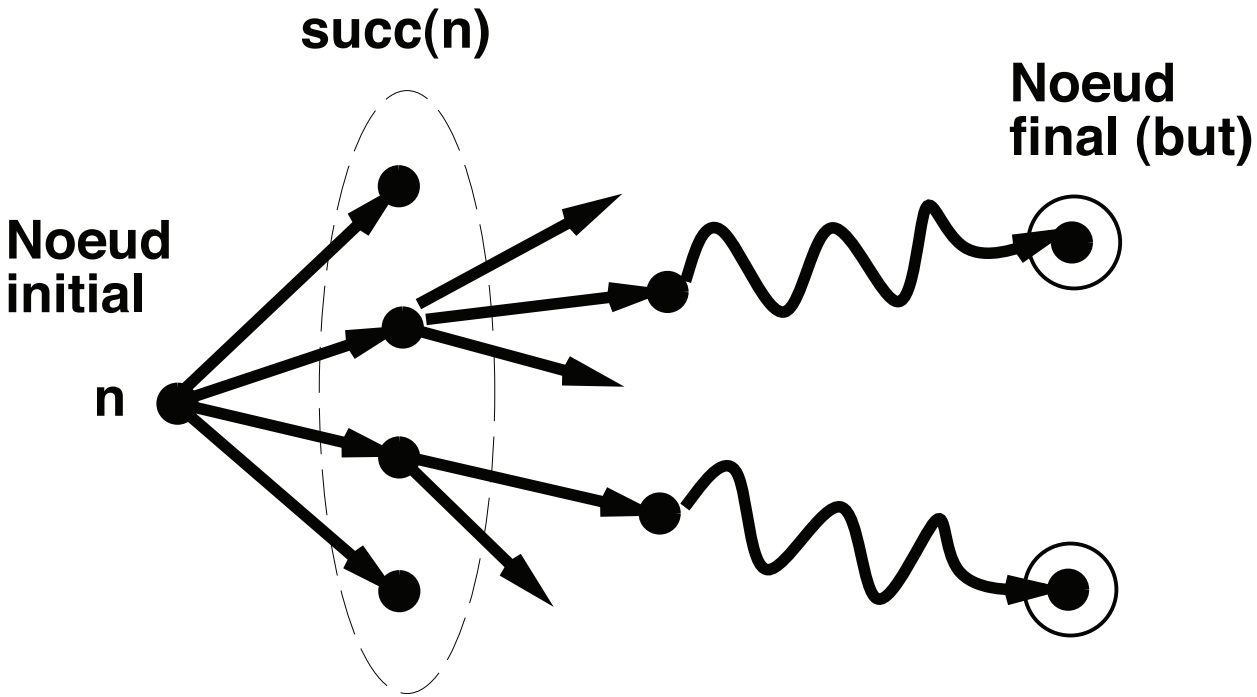
- diagnostic: trouver un diagnostic qui explique les observations.
- planification: trouver un plan qui est faisable et arrive au but.
- mais aussi déduction en chaînage arrière: trouver une séquence d'inférences qui finit par les conditions initiales.

Formalisation d'algorithmes de recherche

3 élements qui caractérisent le problème:

- Noeuds de recherche
- Fonction de successeur
- Critère de succès

Graphe de recherche



$\Rightarrow$ algorithmes généraux pour trouver le noeud final dans un temps minimal

Noeud de recherche

Décrit l'état courant
Exemples:

- solution abstraite: diagnostic partiellement détaillée.
- solution partielle: état intermédiaire dans un plan.
- ensemble des buts dans un systéme à chaînage arrière.

Fonction de génération de successeurs

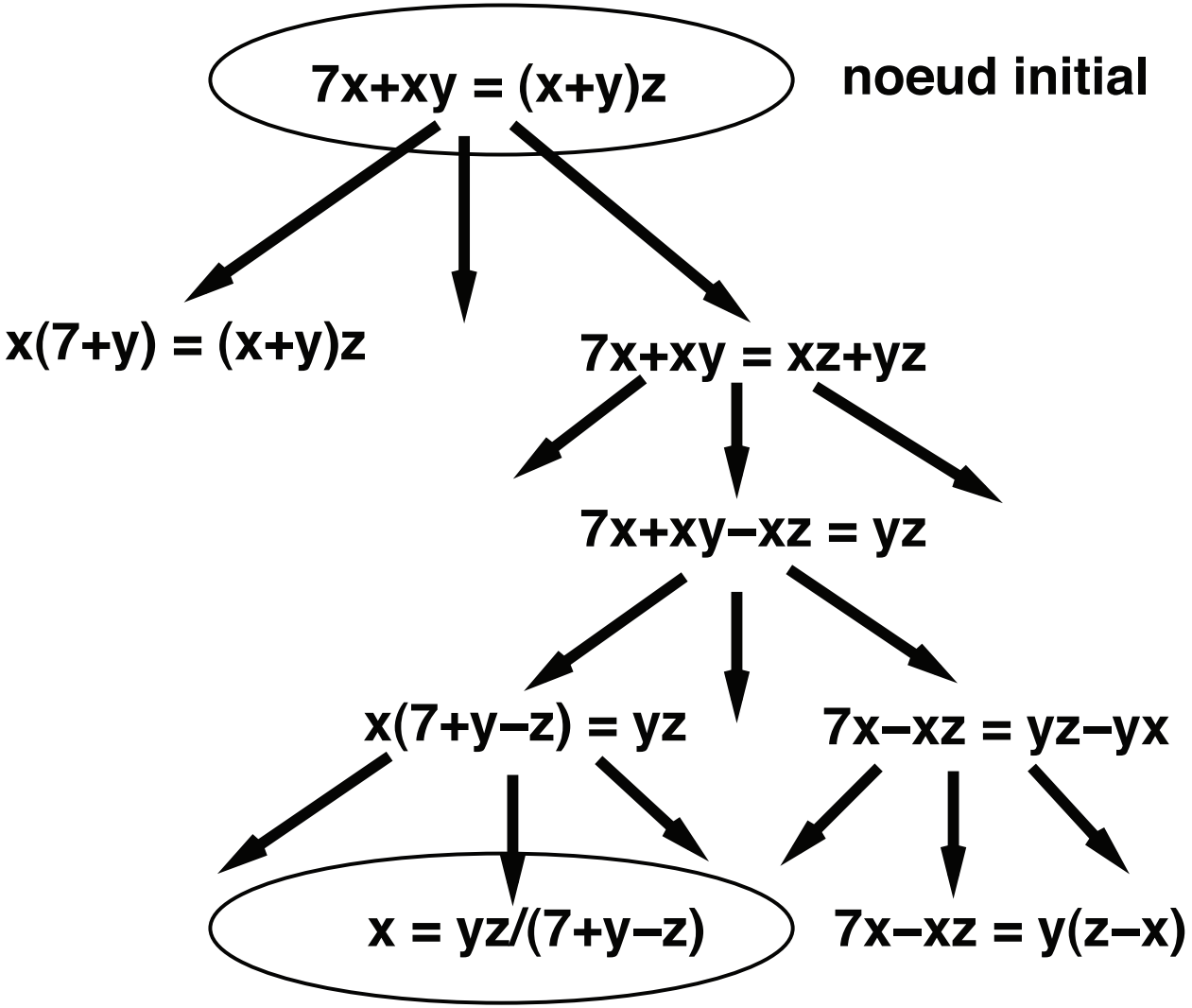
succ(n) = *liste des noeuds atteignables de n*
Exemples:

- diagnostic: affiner les candidats.
- planification: génération des situations atteignables par application des actions disponibles.
- inférence logique: application des règles dans un moteur d'inférence.

Critère de succès

- La recherche commence avec un ou plusieurs *noeuds initiaux*, et se termine avec un noeud qui remplit la *condition de terminaison*, appelée le *noeud final*.
- Le noeud initial et la fonction de successeurs $succ(n)$ définissent un *espace de recherche*. Il s'agit d'un graphe orienté dont les arcs représentent les noeuds atteignables par la fonction $succ(n)$.
- Un algorithme de recherche effectue une exploration locale du graphe de manière à trouver la solution avec un temps de calcul minimal.
- Normalement, on cherche une seule solution.

Exemple: simplification algebrique



L'optimalité dans la recherche

La solution peut être:

a) **le noeud final**: par exemple, dans un système d'inférence ou de conception.

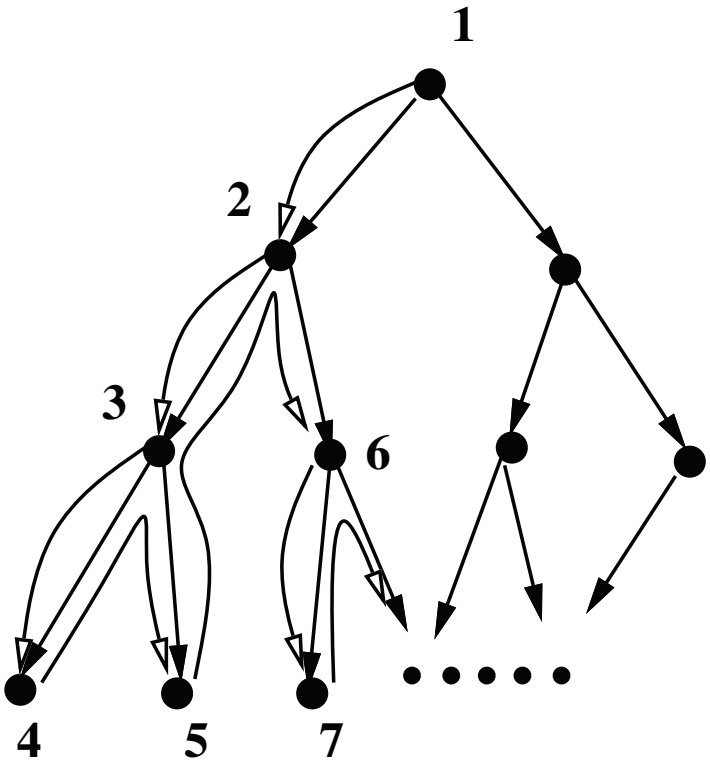
b) **le chemin au noeud final**: par exemple, dans un système de planification.

Surtout dans le cas b), on cherche le chemin à moindre *coût*:

- chaque transition d'un noeud à son successeur a un coût
- le coût d'un noeud est la somme des coûts du chemin
- certains algorithmes garantissent que la solution trouvée a un coût minimal.

La recherche en profondeur d'abord (DFS)

Espace de recherche consideré comme arbre inversé:

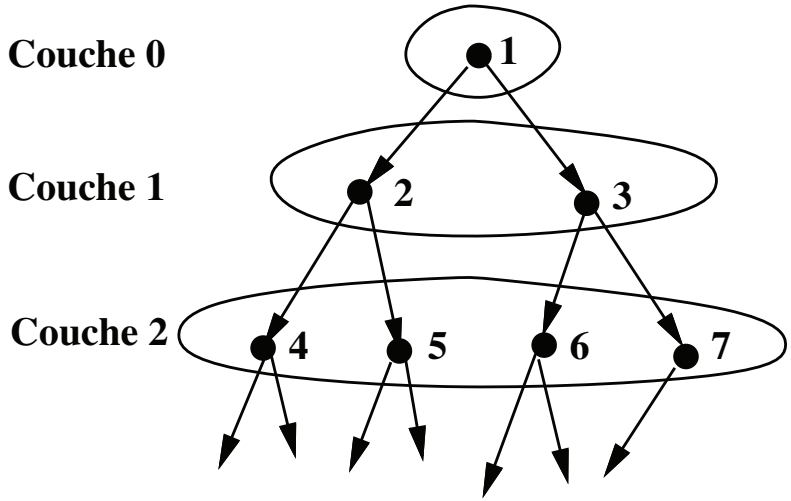


Profondeur d'abord (Depth-first):

- expansion du premier noeud trouvé jusqu'à ce qu'il n'y a plus de successeurs.
- retour en arrière (backtrack): retour à un niveau supérieur et essai de la prochaine possibilité.

Avantage: peu de mémoire requise: liste des noeuds "ouverts" et de leurs successeurs encore non explorés.

La recherche en largeur d'abord (BFS)



- Largeur d'abord (breadth-first): génération de l'arbre de recherche couche par couche.
- Trouve toujours le chemin le plus court.
- Exige beaucoup de mémoire pour stocker toutes les alternatives à toutes les couches.

Algorithmes

Profondeur d'abord:

1: Function DFS (Noeud-initial)
2: Q $\leftarrow$ (Noeud-initial)
3: **repeat**
4: n $\leftarrow$ first(Q), Q $\leftarrow$ rest(Q)
5: **if** n est un noeud but, **return** n
6: S $\leftarrow$ succ(n)
7: Q $\leftarrow$ **append**(S, Q)
8: **until** Q est vide
9: return ECHEC

Largeur d'abord: échanger l'ordre dans le pas 7:

7. Q $\leftarrow$ **append**(Q, S)

Recherche en profondeur limitée (DLS)

- Faiblesse de la recheche en largeur d'abord: exige une quantité de mémoire exponentielle.
  pas corrigeable!
- Faiblesse de la recherche en profondeur d'abord: peut descendre très loin dans un chemin inutile.
- $\Rightarrow$ limiter à une profondeur maximale l:
  pour tout noeud à profondeur l, on ne considère pas ses successeurs.

DLS

Profondeur d'abord:

```
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
```

Comment choisir la limite *l*?

Approfondissement itérative

Incrémenter la limite:

Function Iterative-deepening(Noeud-initial)
$l \leftarrow 2$
repeat
$solution \leftarrow DLS(Noeud - initial, l)$
$l \leftarrow l + 1$
until $solution \neq \{\}$

Chaque étape refait tout le travail de la précedente...
est-ce que c'est coûteux?

Complexité

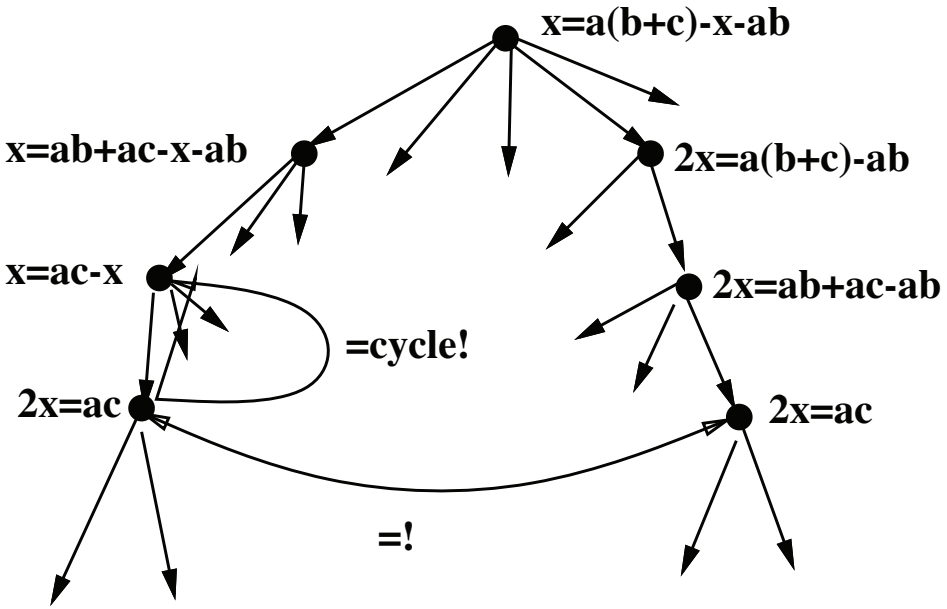
Si solution à profondeur l, l'algorithme a exploré tous les espaces de profondeur l, $l-1$, ..., 2.
Pour un nombre de noeuds $c(i) = b^{i+1} - 1$, la complexité totale est:

$$\begin{aligned}
\sum_{i=2}^{l} c(i) &= \sum_{i=3}^{l+1} (b^i - 1) \\
&= \left(b^{l+1} \sum_{i=0}^{l-2} b^{-i} \right) - (l-1) \\
&< \left(b^{l+1} \cdot \frac{b}{b-1} \right) - (l-1) \leq 2c(l)
\end{aligned}$$

si $b \geq 2, l \geq 3$
$\Rightarrow$ complexité pas plus que doublée.

La recherche de graphes

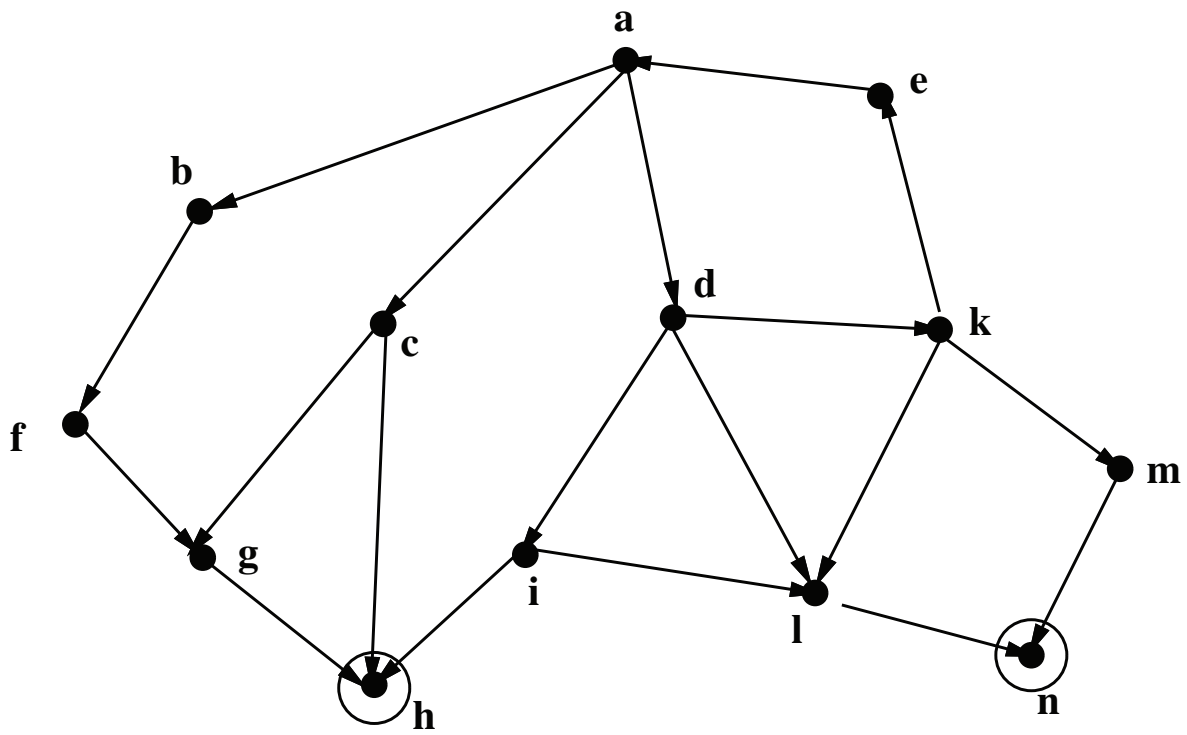
Un même noeud peut être atteint par plusieurs chemins $\Rightarrow$ duplication du travail:



Recherche avec détection de cycles

```
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
```

Exemple d'une recherche



Recherche en profondeur d'abord: a $\Rightarrow$ b $\Rightarrow$ f $\Rightarrow$ g $\Rightarrow$ h

Recherche en largeur d'abord: a $\Rightarrow$ {b,c,d} $\Rightarrow$ {f,g,h,i,l,k}

Optimisation de coûts

- Supposons qu'on cherche à minimiser le *coût* de la solution.
- Modèle: chaque génération de successeurs rajoute un coût $c(n', n)$: si n est successeur de n', alors le coût $g(n)$:

$$g(n) = c(n', n) + g(n') = c(n', n) + \sum_{n', n'' \in ancetres(n)} c(n', n'')$$

- Exemple: planification:
  - chaque expansion correspond à une action
  - coût du plan = somme des coûts des actions

Optimisation par DFS

Quand un noeud but est trouvé:

- mémoriser si son coût est meilleur que le meilleur trouvé
- ne pas générer des successeurs

Continuer la recherche jusqu'à ce que la list `Q` devient vide.

```
1: Function DFS (Noeud-initial)
2: Q ← (Noeud-initial); c ← ∞; sol ← 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:         Q ← append(S, Q)
10: until Q est vide
11: return sol
```

Branch-and-bound

Le coût d'un successeur n'est jamais inférieur à son ancètre.
$\Rightarrow$ si le coût d'un noeud n dépasse la coût d'un noeud but déjà trouvé (le *bound*), on peut l'ignorer.

```
1: Function DFS (Noeud-initial)
2: Q ← (Noeud-initial); c ← ∞; sol ← 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
```

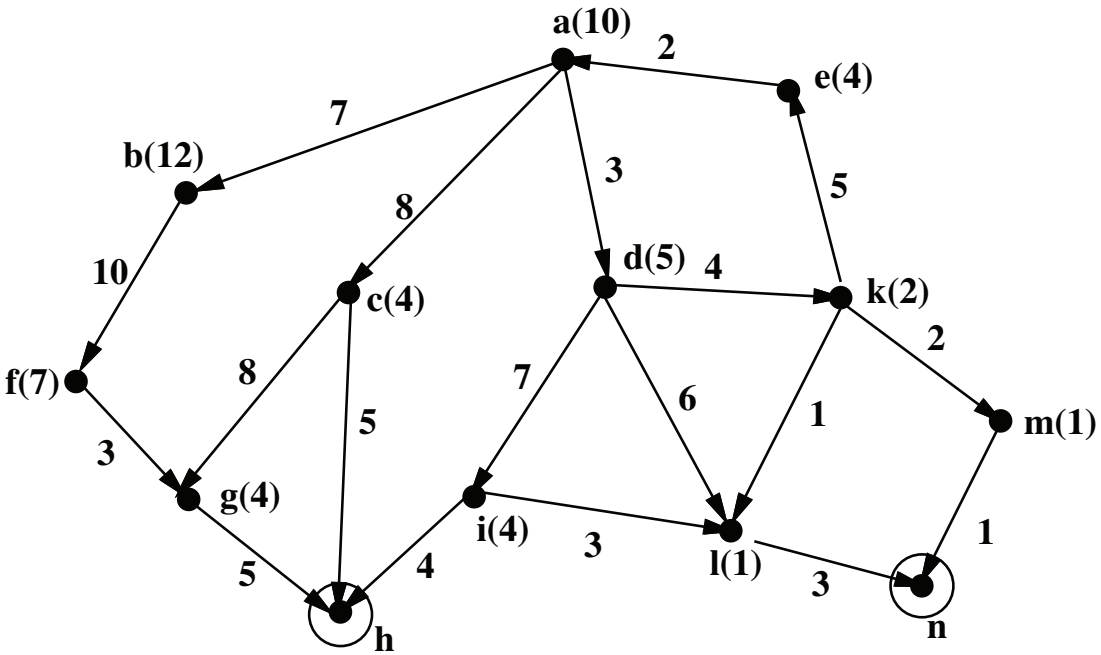
La recherche heuristique

- Approche: guider la recherche pour explorer d'abord les solutions les plus prometteuses.
- Cela peut être exprimé par une fonction *heuristique*:
  h(n) = estimation du coût minimal à partir du noeud n jusquà un noeud but.
- Si g(n) = coût du chemin jusqu'au noeud n, alors la fonction d'évaluation:
  $$f(n) = g(n) + h(n)$$
  est une estimation du coût du chemin optimal qui passe par n.
- $\Rightarrow$ privilégier l'exploration des noeuds dont la valeur de f(n) est basse: best-first (meilleur d'abord).

L'algorithme A*

1: Function A*(Noeud-initial)
2: Q $\leftarrow$ (Noeud initial)
3: C $\leftarrow$ vide
4: **repeat**
5: n $\leftarrow$ first(Q), Q $\leftarrow$ rest(Q)
6: **if** n $\notin$ C, ou n $\equiv$ n' $\in$ C mais g(n) $<$ g(n') **then**
7: **if** n est un noeud but, **return** n
8: ajouter n à C
9: S $\leftarrow$ succ(n)
10: S $\leftarrow$ sort(S,f)
11: Q $\leftarrow$ merge(S, Q, f)
12: **until** Q est vide
13: return ECHEC

Exemple d'une recherche heuristique



1. (a(10)) ⇒
2. (d(8),c(12),b(19)) ⇒
3. (k(9),l(10),c(12),i(14),b(19)) ⇒
4. (l(9),m(10),c(12),i(14),e(16),b(19)) ⇒
5. (m(10),n(11),c(12),i(14),e(16),b(19)) ⇒
6. (n(10),c(12),i(14),e(16),b(19)) ⇒ solution!

L'optimalité de la solution de A*

- Si toujours `h(n) = 0`, on explore les noeuds dans l'ordre de leur coût: la solution optimale est garantie.
- Lorsque l'algorithme doit choisir entre deux noeuds n et n', si `h(n)` *surestime* le vrai coût h*(n) restant, on peut avoir:
  `g(n')+h(n')<g(n)+h(n)` (`>g(n)+h*(n)`)
  même si le coût total du chemin par n est moins important que celui par n':
  `g(n')+h*(n')>g(n)+h*(n)`
  $\Rightarrow$ l'algorithme pourrait s'arrêter avec la solution n' qui est sous-optimale!
- On peut prouver que A* trouve toujours la solution optimale tant que `h(n)` **ne sur-estime pas** le vrai coût jusqu'au noeud final.

Complexité polynomiale

La complexité de A^* est meilleure que tout autre algorithme qui garantit l'optimalité de la solution, mais toujours exponentielle dans la longueur de la solution.
Si l'erreur de la fonction heuristique ne croit pas plus rapidement que le logarithme de sa valeur:

$$|h^*(n) - h(n)| \leq O(log h^*(n))$$

alors le nombre de noeuds exploré par A^* est polynomial en fonction de la longueur.

Trouver une fonction heuristique

Simplifier le problème $\Rightarrow$

- plus simple à optimiser.
- coût réduit.

Exemples:

- trouver un chemin dans un graphe $\Rightarrow$ bouger en ligne droite.
- ignorer les contraintes de précedence dans un ordonnancement.

Limitations de mémoire

La queue `Q` de A* peut devenir très longue.
Idées:

- "Beam Search": garder les n meilleurs noeuds seulement, jamais explorer les autres.
- Iterative Deepening A^*: effectuer un depth-first search jusqu'à un certain seuil de la fonction d'évaluation. Attention: on ne peut augmenter le seuil qu'en très petits incréments!
- Memory-bounded A^*: techniques pour "oublier" et regénerer des noeuds.

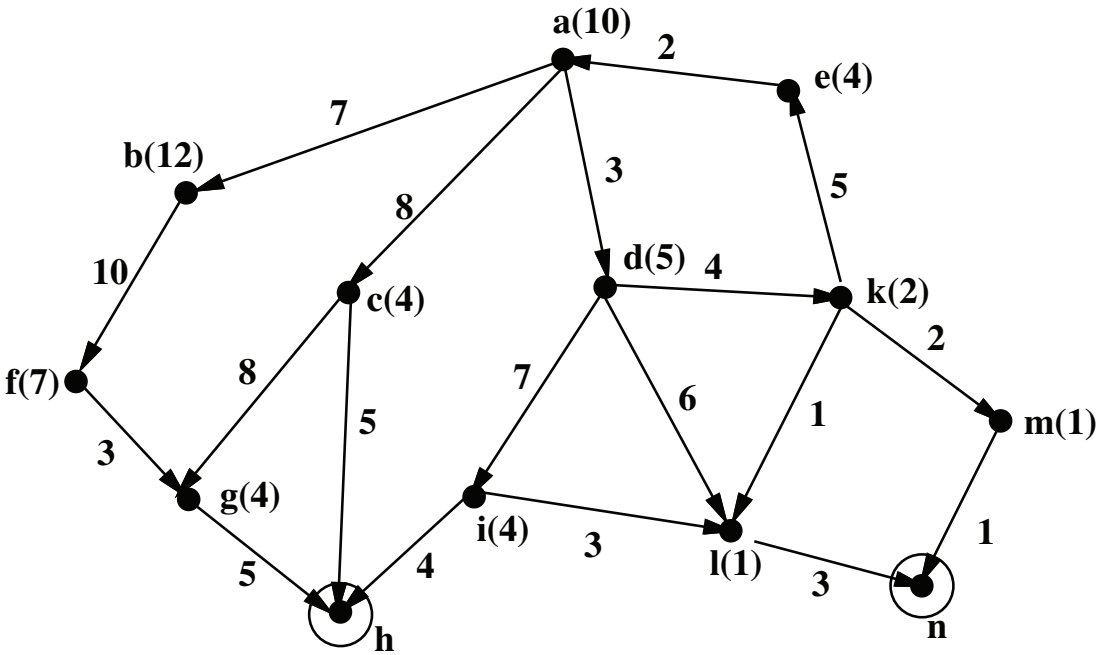
Chemins multiples

Quand on trouve un nouveau chemin à un noeud n, déjà visité, il se peut que le chemin soit meilleur:

1. si n n'avait pas encore été étendu:
   mettre à jour son coût, chemin et position dans Q.
2. si n avait déjà été étendu ($\in$ C): mettre à jour les successeurs de n dans Q.

Exemple: chemin d $\rightarrow$ l remplacé par d $\rightarrow$ k $\rightarrow$ l

Exemple



1. (a(10)) ⇒
2. (d(8),c(12),b(19)) ⇒
3. (k(9),l(10),c(12),i(14),b(19)) ⇒
4. (l(9),m(10),c(12),i(14),e(16),b(19)) ⇒
5. (m(10),n(11),c(12),i(14),e(16),b(19)) ⇒
6. (n(10),c(12),i(14),e(16),b(19)) ⇒ solution!

Le critère de monotonicité

- Si h(k) = 4, on aurait étendu le noeud l avant de trouver le meilleur chemin par k $\Rightarrow$ suite de la recherche doit être repetée.
- Cela peut être evité par la restriction de *monotonicité*:

$$h(n_1) - h(n_2) \leq c(n_1, n_2)$$

= la différence entre les fonctions heuristiques de deux noeuds ne doit jamais dépasser le coût nécessaire pour passer de l'un à l'autre.

Profondeur/Largeur d'abord...

- A* peut simuler l'algorithme de profondeur d'abord: $g(n) = 0$, $h(n) = <$ ordre dans succ(n) $>$
- A* peut aussi simuler l'algorithme de largeur d'abord: $g(n) = <$ longueur du chemin $> h(n) = 0$
- $\Rightarrow$ le comportement de A* se trouve entre les deux algorithmes.

Temps de calcul d'un algorithme de recherche

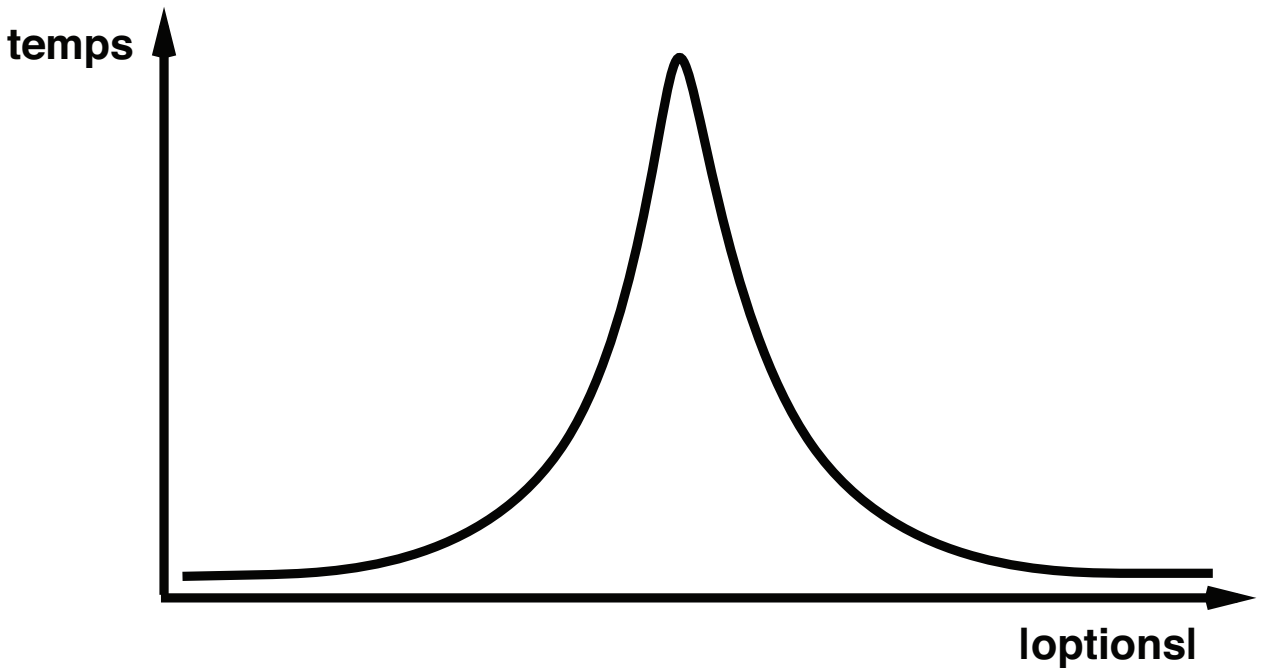
- Temps minimal $\simeq$ longueur du chemin à la solution
- Temps maximal $\simeq$ exploration de tout le graphe
- Dépend uniquement de l'ordre de l'exploration!
- $\Rightarrow$ très imprévisible

Temps de calcul moyen

3 cas:

- beaucoup d'options, beaucoup de noeuds finaux:
  $\Rightarrow$ probabilité elevée de tomber sur une solution
  $\Rightarrow$ temps de calcul faible
- peu d'options, pas de noeuds finaux:
  $\Rightarrow$ graphe de recherche petit
  $\Rightarrow$ parcours complet en peu de temps
- beaucoup d'options, un seul noeud final:
  $\Rightarrow$ parcourir la moitié du graphe
  $\Rightarrow$ temps de calcul élevé

Transitions de phase



$\Rightarrow$ classification des algorithmes selon la forme de la courbe.

Application: routage de véhicules autonomes

- Les véhicules autonomes (Autonomous Guided Vehicle) transportent des matériaux à l'intérieur d'usines.
- Normalement, ils sont programmés par des règles de comportement.
- Ils ne tiennent pas compte de l'interaction entre vehicules, ni de pannes
- Système de recherche heuristique basé sur A^* pour chercher les meilleures mouvements a été mis au point par Lookahead Decisions:
  - améliore le débit de 83%
  - réduit le temps moyen de parcours de 25%
  - réduit le nombre d'arrêts de 48%

Résumé

- Les algorithmes de recherche sont l'outil de base pour le raisonnement abductif.
- Recherche d'arbres: l'approfondissement iteratif combine les avantages de la recherche en profondeur et largeur d'abord.
- Recherche de graphes: la detection de cycles est important mais consomme de la memoire.
- Recherche heuristique: A* peut trouver la solution optimale de manière efficace.