Représentation de la Connaissance
Complément Pratique

24 octobre 2006

Stratégies de base de résolution de problèmes

1 Introduction

Nous nous intéressons ici à une classe très générale de modélisation de problèmes, à savoir la modélisation par *espace d'état* (state space). Un espace d'état est un graphe dont les noeuds correspondent à des situations du problème, et le problème lui-même consiste à trouver un chemin permettant de passer d'une situation initiale du système à la situation finale désirée.

Cette classe de problème est très générale en ce sens que nombre de problèmes rencontrés habituellement en intelligence artificielle peuvent être traités de la sorte. Par exemple un jeu peut être décrit par un graphe dont les noeuds correspondent à des situations valides du jeu et les arcs des coups possibles pour passer d'une situation à une autre.

Nous allons tout d'abord décrire différentes stratégies basiques permettant de trouver un chemin dans un graphe et tenter de montrer les avantages et inconvénients de chacunes d'entre-elles. Nous verrons que ces stratégies de base sont très efficaces pour des problèmes peu complexes mais ne tiennent pas la route lorsque la taille de l'espace d'état et le facteur de branchement sont élevés.

2 Depth-first search

La recherche la plus simple d'un chemin dans un graphe consiste à parcourir le graphe à l'aide d'une méthode de recherche *en profondeur d'abord*, sans aucun autre mécanisme supplémentaire. L'idée en est très simple : si le noeud courrant est le noeud solution du problème la recherche s'arrête car la solution est déjà trouvée. Sinon, la liste des successeurs du noeud courrant est générée, et la recherche continue à partir du premier d'entre-eux. Les autres noeuds ne seront explorés qu'une fois que toutes les possibilités d'étendre le premier noeud auront été épuisées sans succès.

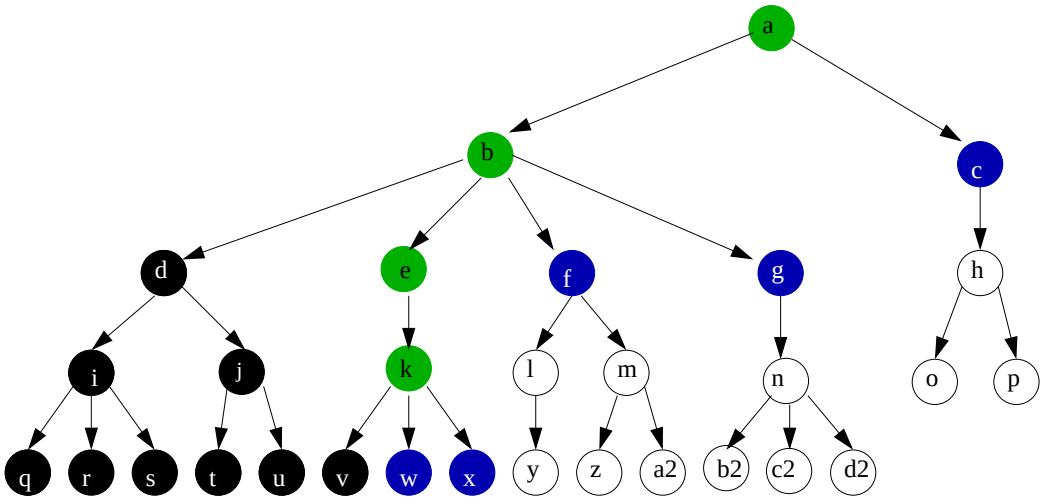

FIG. 1 – Parcours en profondeur d'abord de l'espace d'état

Le code Prolog qui implémente cette méthode est très simple étant donné que Prolog lui-même fonctionne en profondeur d'abord.

```
solve(N,[N]) :- goal(N).
solve(N,[N|Sol1]) :- s(N,N1),solve(N1,Sol1).
```

où `goal(N)` est vrai si `N` est un noeud solution du problème
et `s(N,N1)` est vrai s'il existe un arc entre les noeuds `N` et `N1`.

Cette méthode très simple fonctionne bien dans certains cas particuliers. Un exemple classique est le problème des huits reines.

2.1 Problème des huits reines

Pour rappel, le problème consiste à placer huit reines sur un échiquier sans qu'aucune d'entre-elles ne puisse en prendre une autre. Une modélisation du problème par espace d'état qui pourrait être utilisée par notre méthode *solve* peut être formulée comme suit :

- Les noeuds du graphe sont des positions de l'échiquier contenant entre zéro et huit reines placées dans des colonnes consécutives de l'échiquier.
- Un noeud successeur est obtenu en placant une reine dans la prochaine colonne de manière à ce qu'elle n'attaque aucune autre reine.
- Le noeud de départ est celui de l'échiquier vide et sera représenté par une liste vide.

– Un noeud solution est n'importe quel noeud avec huit reines (la règle du successeur garantissant que les reines ne s'attaquent pas entre-elles).

Représentant l'échiquier par la liste des coordonnées des lignes dans lesquelles les reines se trouvent, on obtient facilement un code pour modéliser le problème :

```
s (Queens,[Queen|Queens]) :- member(Queen,[1,2,3,4,5,6,7,8]),
                             noattack(Queen,Queens).
goal(_,_,_,_,_,_,_,_).
```

Le prédicat `noattack(Queen,Queens)` est vrai si la reine `Queen` n'attaque pas les reines de la liste `Queens` et son implémentation (simple) est laissée au lecteur. La question

```
?- solve([],Solution).
```

va produire toutes les solutions possibles au problème des huit reines par backtracking. La méthode simple de recherche en profondeur d'abord fonctionne bien pour ce problème pour la simple raison que le graphe construit dégénère en un arbre, et donc qu'aucun cycle ne peut apparaître.

2.2 Détection des cycles

Les problèmes dont l'espace d'état est un graphe cyclique ne pourront être résolus par la méthode simple vue jusqu'à présent. Un graphe cyclique apparaît lorsque le problème permet de revenir dans un état précédent par une série de coup. Pour illustrer cela, partons de l'espace d'états suivants :

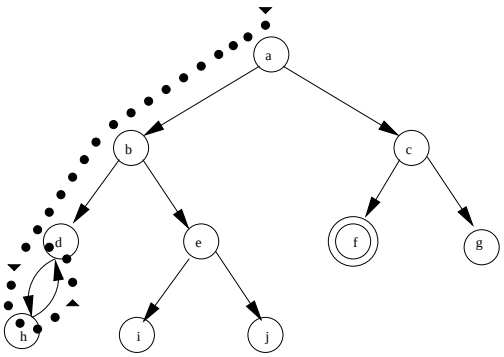

Fig. 2 – Exemple d'espace d'état cyclique

Face à un graphe tel que celui de la figure 2 le code simple que nous avons vu ne permettrait pas de trouver un chemin vers le noeud solution f étant donné que la recherche en profondeur d'abord va rester continuellement sur la branche $a - b - d - h - d - h - d - h$−...

Pour ce genre de problèmes, il nous faut donc un mécanisme de détection des cycles. La solution la plus simple consiste à ignorer les noeuds qui sont des clones d'un noeud visité précédement.

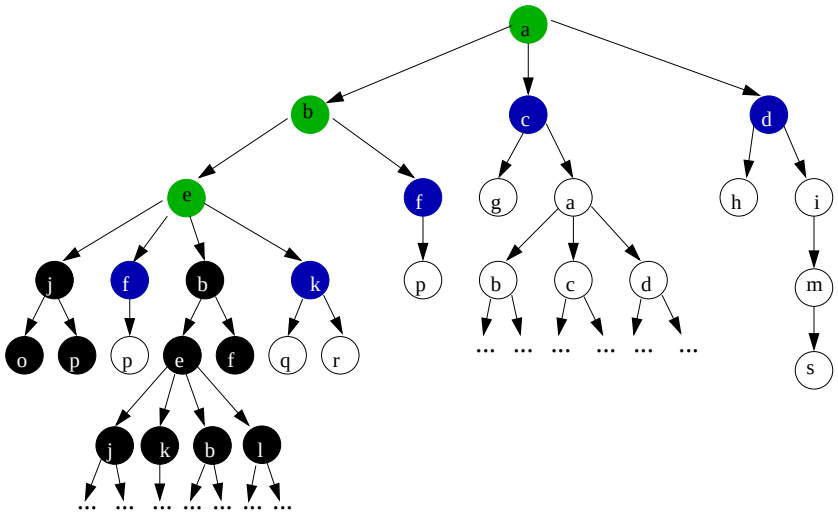

FIG. 3 – Parcours en profondeur d'abord avec détection de cycles

Pour faciliter l'algorithme, nous produiront le chemin inverse du chemin solution. Voici le code Prolog qui implémente cette idée :

```
% solve(+Node,-Solution)
% Solution is an acyclic path (in reverse order) between Node and a goal

solve(Node, Solution)  :-
  depthfirst([], Node, Solution).

% depthfirst(+Path,+Node,-Solution)
% extending the path [Node|Path] to a goal gives Solution

depthfirst(Path, Node, [Node|Path])  :-
  goal(Node).
```

```
depthfirst(Path, Node, Sol)  :-
  s(Node, Node1),
  not member(Node1, Path),
  depthfirst([Node|Path], Node1, Sol).
```

Ce programme permet donc bien de traiter des graphes tel que celui que nous venons de prendre en exemple, mais par contre ne fonctionnera pas lorsque l'espace d'état est infini. Ce qui est d'ailleurs assez couramment le cas. En effet, l'algorithme que nous venons de voir pourrait très bien échouer dans sa recherche d'un noeud solution si ce dernier est situé à droite d'une branche infinie du graphe.

2.3 Depth-limited, depth-first search

Une méthode qui palie au problème rencontré précédemment est la recherche *en profondeur d'abord avec profondeur de recherche bornée*. L'idée est très simple, il s'agit de limiter la profondeur de l'arbre à une certaine borne fixée a priori. Ainsi, si le graphe est infini mais possède des solutions situées à une profondeur inférieure ou égale à la borne, elles seront trouvées.

Voici un code Prolog qui implémente cette idée (sans détection de cycles) :

```
% depthfirst2(+Node,-Solution,+Maxdepth)
% Solution is a path, not longer than Maxdepth, from Node to a goal

depthfirst2(Node, [Node], _)  :-
  goal(Node).

depthfirst2(Node, [Node | Sol], Maxdepth)  :-
  Maxdepth > 0,
  s(Node, Node1),
  Max1 is Maxdepth - 1,
  depthfirst2(Node1, Sol, Max1).
```

Il est evidemment également possible de combiner détection de cycle et limitation de la profondeur de recherche si le problème spécifique le nécessite. Le code n'est pas difficile à construire à partir des deux codes précédents et est laissé en exercice au lecteur.

3 Iterative deepening

Un des difficulté d'utilisation du programme de recherche en profondeur d'abord avec borne sur la profondeur de recherche est de déterminer une borne adéquate. Si nous choisissons une borne très petite nous risquons de ne trouver aucune solution et si nous choisissons une borne trop élevée la complexité va devenir inacceptable.

Pour contourner ce problème, il est possible d'exécuter la méthode de manière itérative en augmentant la profondeur de recherche d'une unité à chaque itération. Cette méthode est appellée la recherche par *approfondissement itératif*. Le code du programme peut s'écrire facilement en écrivant une fonction auxilliaire qui appelle `depthfirst2` en augmentant la valeur de `Maxdepth` d'une unité à chaque appel récursif.

Cette procédure est très simple et bien qu'elle ne fasse rien de vraiment intelligent elle a une efficacité satisfaisante en pratique pour des problèmes qui ne nécessitent pas le recourt à des méthodes plus compexes basée sur des heuristiques. Une comparaison plus précise des performanes des différentes méthodes de base sera faites dans la suite.

4 Breadth-first search

Contrairement aux différentes méthodes de recherche en profondeur d'abord, la méthode de recherche *en largeur d'abord* choisi de visiter les noeuds les plus proches de la racine en premier lieu. La figure 4 illustre très bien le mécanisme mis en place.

Point de vue algorithmique, cette méthode est plus dure à programmer. En effet, il ne nous suffit plus maintenant d'étudier les chemins candidats un à un et de les éliminer de la mémoire au fur et à mesure comme c'était le cas pour la recherche en profondeur d'abord. Il nous faut impérativement maintenir un ensemble de chemins candidats en mémoire.

Pour montrer la raison de ce fait, intéressons nous par exemple au moment où notre recherche à atteint le noeud f de la figure 4. A ce moment de l'exécution, on voit directement qu'il est impératif que les chemins aboutissant aux noeuds bleus soient toujours présents en mémoire. On voit donc qu'au fur et à mesure que notre recherche explore l'arbre, et donc que la largeur de l'arbre augmente, le nombre de chemins candidats à conserver en mémoire augmente.

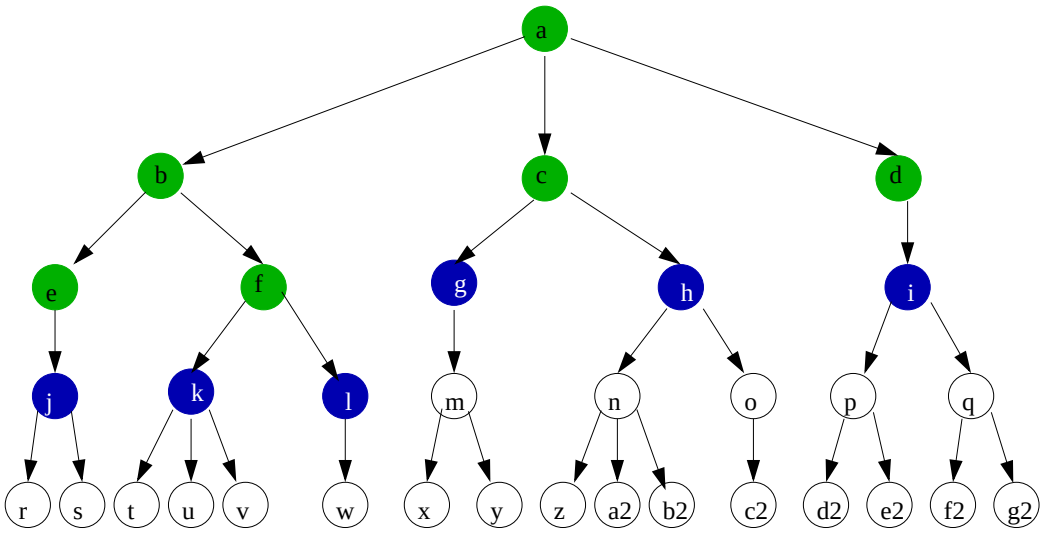

FIG. 4 – Parcours en largeur d'abord

Voici un code Prolog (utilisant la notation par différence de liste) qui implémente la recherche en largeur d'abord :

```
solve(Start, Solution)  :-
  breadthfirst([[Start]|Z] - Z, Solution).

breadthfirst([[Node|Path]|_] - _, [Node|Path])  :-
  goal(Node).

breadthfirst([Path|Paths] - Z, Solution)  :-
  extend(Path, NewPaths),
  append(NewPaths, Z1, Z),
  Paths \== Z1,
  breadthfirst(Paths - Z1, Solution).
```

5 Analyse des différentes méthodes

Les examples fournis jusqu'à présent pourraient vous avoir donné l'impression que les différentes méthodes de recherche utilisées ne fonctionnent que pour des arbres et non en général pour des graphes. En réalité cependant, lorsqu'on fait une recherche dans un graphe, il ne s'agit en fait que d'une recherche dans un arbre dégénéré où certains noeuds ont été dupliqués.

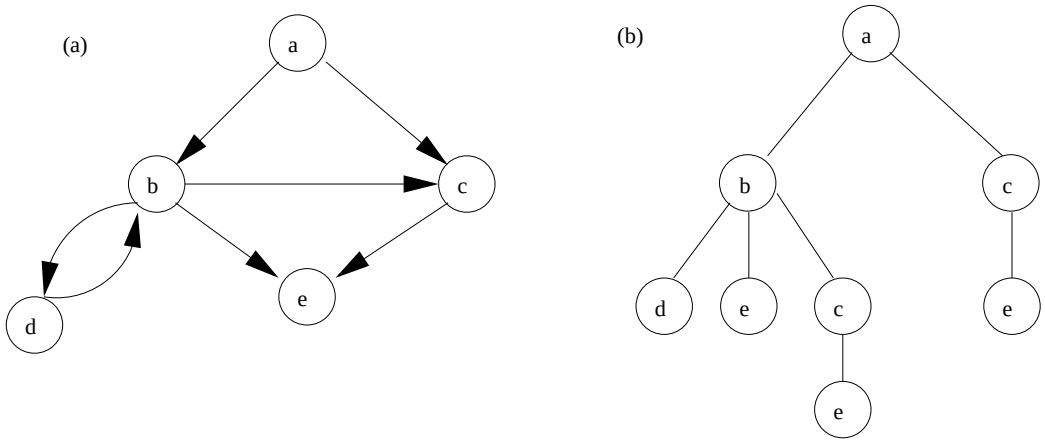

FIG. 5 – (a) Un espace d'état (b) L'arbre de tous les chemins non-cycliques à partir de a

Les programmes fonctionnent donc aussi pour des graphes quelconques bien qu'ils risquent de dupliquer le travail pour les noeuds qui pouvent être atteints par plusieurs chemins.

Pour éviter cela, dans le cadre de la recherche en profondeur d'abord, il suffit de tester la répétition de noeuds dans *tous* les chemins candidats et non uniquement dans le chemin dans lequel le noeud a été généré. Pour ce qui est des différentes méthodes de recherche en profondeur d'abord par contre il faudra utiliser le mécanisme de détection de cycles.

Pour des problèmes non-triviaux l'espace d'état va devenir tellement grand que la complexité va jouer un rôle important. En Prolog, on étudie toujours deux types de complexité :

- **la complexité spaciale** : il s'agit de donner un ordre de grandeur de l'espace mémoire qui sera nécessaire lors de l'exécution du programme. Nous mesurerons cette complexité par le nombre maximum de noeuds du graphe qu'il faut maintenir en mémoire lors de l'exécution.

- **la complexité temporelle** : il s'agit de donner un ordre de grandeur du temps qu'il faudra pour trouver une solution. Nous mesurerons cette complexité par le nombre de noeuds visités avant de trouver une solution.

Afin de comparer les différentes méthodes, étudions un espace d'état assez génériques qui serait un arbre dont chaque noeud à exactement b *successeurs* et dont la première solution se trouve à une *profondeur* d de l'arbre.

Il va de soi que la méthode basique de recherche en profondeur d'abord aura des complexités spaciales et temporelles pouvant être infinie et qu'il est donc impossible de les comparer aux autres en toute généralité. Nous étudierons donc plutôt la recherche en profondeur d'abord limitée à une *profondeur de dmax*.

Le nombre de noeuds de l'arbre augmente de manière exponentielle avec la profondeur, la recherche en largeur d'abord nécessitant de générer tous ces noeuds et de les garder tous en mémoires, les complexités spaciales et temporelles seront donc de $1 + b + b^2 + b^3 + ...$ c'est à dire une complexité de l'ordre $O(b^d)$.

La recherche en profondeur d'abord limitée va par contre nécessiter en toute généralité de générer tous les noeuds de l'arbre jusqu'à une profondeur de $dmax$ mais de ne garder en mémoire à chaque fois qu'une seule branche de l'arbre ce qui nous donne donc des complexités spaciales et temporelles respectivement de $dmax$ et $O(b^{dmax})$.

L'iterative deepening réalise $(d + 1)$ recherches en profondeur d'abord limitées respectivement à des profondeurs de $0, 1, ..., d$. L'efficacité spaciale sera donc de d étant donné que la plus grande branche qu'il nous faudra garder en mémoire au cours de l'exécution sera de longueur d.

Le noeud racine sera visité $d + 1$ fois, les noeuds de profondeur 1 seront visités d fois, etc. Dans le pire des cas, le nombre de noeuds générés est donc :

$$(d + 1) * 1 + d * b + (d - 1) * b^2 + ... + 1 * b^d$$

Cela conduit également à une complexité temporelle de $O(b^d)$. En réalité d'ailleurs, le surplus de travail par rapport à la recherche en largeur d'abord est vraiment faible. On peut prouver sans trop de difficulté que la ratio entre le nombre de noeuds générés par l'iterative deepening par rapport au nombre de noeuds générés par la recherche en largeur d'abord est proche de $b/(b-1)$ pour $b \geq 2$. Ce petit surplus de travail est donc risible par rapport à l'énorme gain de mémoire En ce sens, l'iterative deepening combine les meilleurs propriétés des recherches en largeur d'abord (espace) et en profondeur d'abord (temps) et est donc une méthode fort utilisée en pratique. La table suivante résume les différentes complexités trouvées.

	Complexité temporelle	Complexité spaciale	Solution la plus courte garantie
Breadth-first	b^d	b^d	oui
Depth-first limited	b^{dmax}	$dmax$	non
Iterative deepening	b^d	d	oui

N'oublions pas cependant que les méthodes basiques que nous venons de voir ne font rien de très subtil pour lutter contre l'explosion combinatoire étant donné qu'elles considèrent tous les chemins comme étant aussi prometteurs les uns que les autres. Elles sont en fait non-informées des spécificités du problème et ne sont pas suffisante dans le cas de problèmes à grande échelle. Dans ce cas, il faudra plutôt songer à des méthodes de résolution basées sur des heuristiques.