Représentation de la Connaissance
Complément Pratique

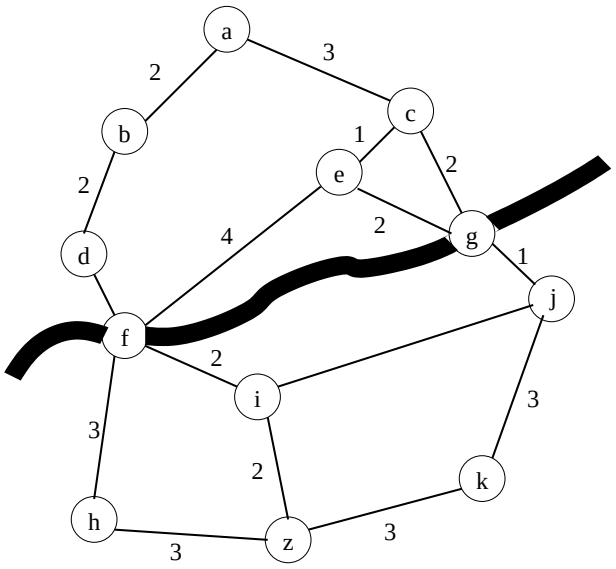
7 novembre 2006

Graphes AND/OR

1 Introduction

Certains problèmes peuvent être décomposés naturellement en plusieurs sous-problèmes indépendants. On peut représenter ces problèmes par un graphe AND/OR plutôt que par un espace d'états.

Soit par exemple le problème de la figure suivante :



On désire trouver un chemin de la ville a à la ville z. Nous savons qu'une rivière sépare les deux villes. Le seul moyen de la traverser est de passer par f ou g. Le problème

```
"trouver un chemin de a à z"
```

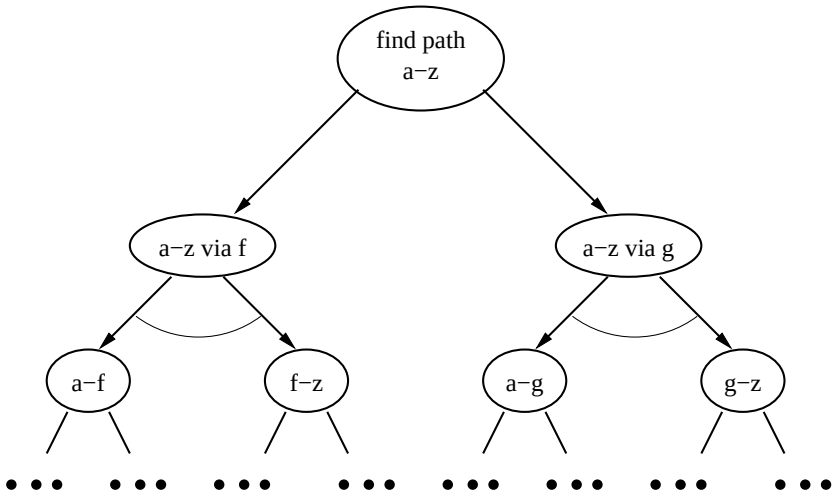
est donc équivalent à :

```
"trouver un chemin de a à z via f"
  ou
"trouver un chemin de a à z via g"
```

Chacun de ces problèmes peut à son tour être décomposé en deux nouveaux sous-problèmes :

1. `"trouver un chemin de a à g"` et `"trouver un chemin de g à z"`
2. `"trouver un chemin de a à f"` et `"trouver un chemin de f à z"`

Une telle décomposition est représentée par le graphe AND/OR suivant :

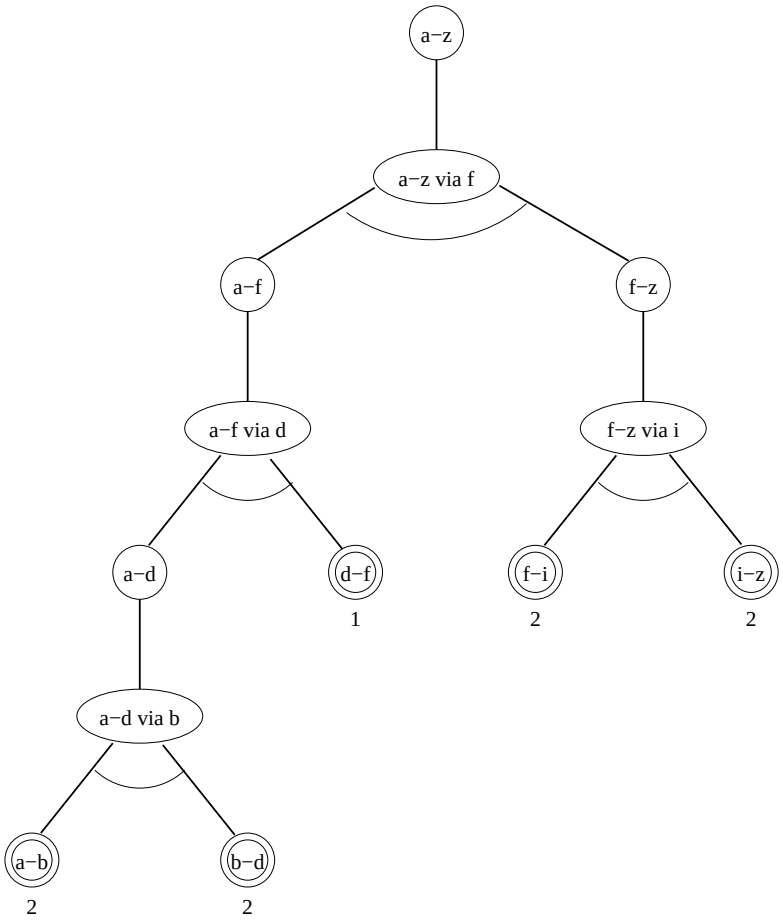


Ce graphe ne représente qu'une partie de l'arbre AND/OR complet. On pourrait continuer la décomposition en introduisant de nouvelles villes intermédiaires. Les fils d'un nœud représentent une décomposition du problème correspondant en sous-problèmes. Si les branches sont reliées par un arc, *tous* les sous-problèmes doivent être résolus pour trouver une solution au problème père (nœud AND). Dans le cas contraire, un seul sous-problème doit être résolu (nœud OR). La décomposition s'arrête à un nœud représentant un problème trivial : ce sont les nœuds goal du graphe AND/OR. Dans notre exemple, un tel problème serait la recherche d'un chemin entre a et b par exemple.

Dans le cas des représentations de problèmes par un espace d'états, une solution est donnée par un chemin dans cet espace. Dans le cas des graphes AND/OR, une solution prend la forme d'un *arbre*. Plus précisément, un tel arbre T est défini de la manière suivante :

- La racine de T est la racine de l'arbre AND/OR.
- Si T contient un nœud OR, alors exactement un des successeurs de ce nœud dans le graphe AND/OR est également un successeur de ce nœud dans T.
- Si T contient un nœud AND, alors *tous* les successeurs de ce nœud dans le graphe AND/OR sont également des successeurs de ce nœud dans T.

Dans le cas de notre exemple, une solution est donnée par l'arbre suivant :



Le chemin correspondant est $[a, b, d, f, i, z]$. On l'obtient en parcourant les feuilles de l'arbre de gauche à droite. Dans ce cas particulier, le coût de la solution est donné par la somme des coûts des feuilles de l'arbre. D'une manière plus générale, des coûts peuvent être associés à tous les nœuds de l'arbre ou aux arcs reliant les nœuds.

2 Recherche AND/OR

L'algorithme que nous allons voir ne tient pas compte des coûts éventuels et retourne donc une solution qui n'est pas nécessairement optimale.

Nous décidons de représenter un arbre AND/OR par une clause

```
Node ---> or : Subtrees.
```

dans le cas où la racine est un nœud OR, ou

```
Node ---> and : Subtrees.
```

dans le cas où la racine est un nœud AND. *Node* est la racine de l'arbre et *Subtrees* la liste des sous-arbres. Les opérateurs `--->` et `:` peuvent être définis par :

```
:- op( 600, xfx, --->).
:- op( 500, xfx, :).
```

La solution est également un arbre. Nous la représentons par

```
Node --> Subtree
```

dans le cas d'un nœud OR et par

```
Node --> and : Subtrees
```

dans le cas d'un nœud AND.

Pour résoudre un nœud N, on applique les règles suivantes :

1. Si N est un nœud goal, la solution est triviale
2. Si N est un nœud OR, on essaye de résoudre un de ses fils (on les essaye un par un jusqu'à en trouver un soluble).
3. Si N est un nœud AND, on essaye de résoudre *tous* ses fils (on les essaye un par un jusqu'à ce qu'ils soient tous résolus).

3 Une implémentation en Prolog

```
%  Depth-first AND/OR search
%  solve( Node, SolutionTree):
%    find a solution tree for Node in an AND/OR graph

:-  op( 500, xfx, :).
:-  op( 600, xfx, ---> ).

% Solution tree of goal node is Node itself
solve( Node, Node)  :-
  goal(Node).

% Node is an OR-node
% Select a successor Node1 of Node
solve( Node, Node ---> Tree)  :-
  Node ---> or:Nodes,
  member( Node1, Nodes),
  solve( Node1, Tree).

% Node is an AND-node
% Solve all Node's successors
solve( Node, Node ---> and:Trees)  :-
  Node ---> and:Nodes,
  solveall( Nodes, Trees).

% solveall( [Node1,Node2, ...],
%           [SolutionTree1,SolutionTree2, ...])

solveall( [], []).

solveall( [Node|Nodes], [Tree|Trees])  :-
  solve( Node, Tree),
  solveall( Nodes, Trees).
```

4 Code Prolog pour afficher l'arbre solution

```
%  Displaying a solution tree

% Display solution tree
% Indented by 0
show(Tree)  :-
  show(Tree,0), !.

% show( Tree, H): display solution tree indented by H

show( Node ---> Tree, H)  :-  !,
  write(Node), write(' ---> '),
  H1 is H + 7,
  show( Tree, H1).

% Display single AND tree
show( and:[T], H)  :-  !,
  show(T,H).

% Display AND-list of solution trees
show( and:[T|Ts], H)  :-  !,
  show( T, H),
  tab(H),
  show( and:Ts, H).

show( Node, H) :-
  write(Node), nl.
```

5 Best-first AND/OR search

L'algorithme AO^* permet de trouver la solution *optimale* (de coût minimal) d'un arbre AND/OR. Nous supposons ici qu'un coût est associé à chaque arc de l'arbre.

Soit un nœud N et N_1,N_2,... ses successeurs. Définissons le coût de N par :

1. $H(N) = min_i(cost(N, N_i) + H(N_i))$ si N est un nœud OR.
2. $H(N) = \sum_i(cost(N, N_i) + H(N_i))$ si N est un nœud AND.

Remarque : Ceci vient du fait que la solution ne contient qu'un seul successeur par nœud OR.

A présent, définissons la mesure $F(N)$ par :

$F(N) = cost(M, N) + H(N)$, où M est le nœud père de N.

On obtient donc :

1. $F(N) = min_i F(N_i)$ si N est un nœud OR.
2. $F(N) = \sum_i F(N_i)$ si N est un nœud AND.

Le principe de l'algorithme AO^* est identique à celui de A^* : on développe à chaque itération un sous-arbre de valeur minimale.