# Représentation de la Connaissance
# Complément Pratique

16 mars 2006

## Résolution de problèmes à l'aide d'heuristiques

## 1 Introduction

Les méthodes vues jusqu'à présents étaient toutes des méthodes de parcours *exhaustif*. Le problème est que l'espace d'état est généralement d'une taille très élevée pour la plupart des problèmes intéressants. Considérons par exemple le jeu d'échecs ; on estime que le nombre de positions légales différentes est de l'ordre de $10^{120}$. Il est donc impossible de parcourir la totalité de l'espace d'état en un temps plus petit que le nombre de nanosecondes écoulées depuis le big-bang !

Il est donc nécessaire de définir des *heuristiques* afin de déterminer dans quel ordre les différents chemins doivent être analysés, c'est-à-dire qui nous permettent de décider quels sont les chemins les plus prometteurs à tester en premier. Une *heuristique* est donc un ensemble de règles, dependant du problème, permettant de sélectionner des chemins ayant une grande probabilité d'atteindre un noeud solution. Les heuristiques ne sont donc pas des théorèmes : même les meilleures stratégies peuvent échouer.

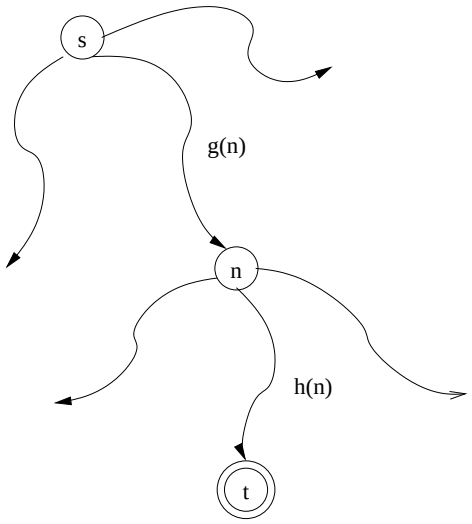
La stratégie consistant à explorer les chemins les plus prometteurs en premier, en se basant sur une heuristique, s'appelle le *best-first search*. Les humains utilisent cette stratégie naturelle tous les jours pour résoudre les problèmes habituels auxquels ils sont confrontés. Par exemple, lorsqu'une personne veut rendre visite à un vieil ami, elle ne va certainement pas faire une recherche exhaustive de toutes les routes possibles en commandant sur Internet toutes les cartes de toutes les routes du monde et en essayant toutes les combinaisons de routes les unes après les autres. La personne utilisera plutôt son expérience (donc une heuristique interne qui lui dicte des règles de *bon-sens*) pour déterminer une route proche de l'optimal. De même, les joueurs d'échecs ne considèrent pas tous les coups possibles dans toutes les positions atteignables possibles, mais n'analyse que ceux qui leurs parraissent potentiellement efficace d'après leur expérience.

## 2 Best-first search

Le *best-first search* ressemble très fortement au *breadth-first search* : le programme commence au noeud racine et garde en mémoire un ensemble de chemins candidats. Le principe du *breadth-first search* est de développer un chemin de taille minimale. Le *best-first search* consiste à développer le chemin qui a le plus de chances d'aboutir à une solution optimale (i.e. de coût minimal).

On définit une fonction $f(n)$ prenant comme argument un noeud $n$ et retournant une estimation du coût d'un chemin optimal passant par ce noeud. A chaque itération, l'algorithme développe un chemin tel que le noeud ajouté à ce chemin a une valeur $f(n)$ minimale.

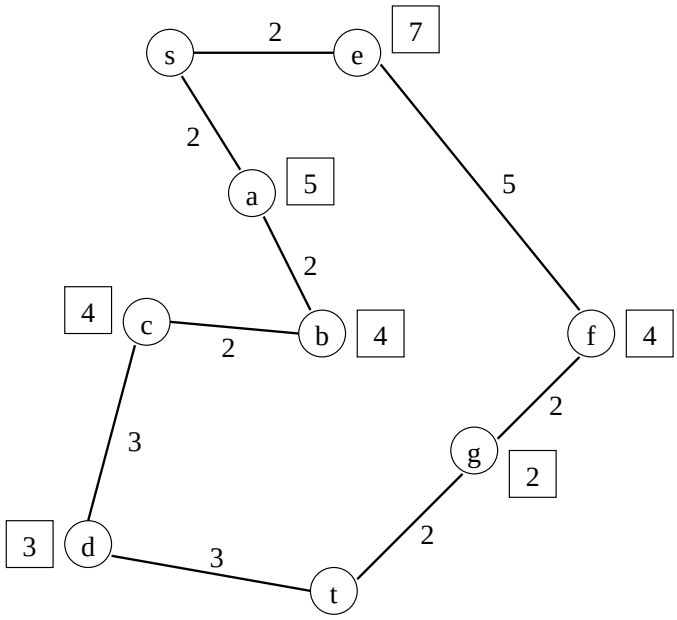
L'algorithme $A^*$ est un exemple de *best-first search*. On utilise dans ce cas la fonction $f(n) = g(n) + h(n)$, où $g(n)$ est une estimation du coût du meilleur chemin entre le noeud racine et $n$ et $h(n)$ est une estimation du coût restant.




Le terme $g(n)$ est calculé de la manière suivante : si on demande d'évaluer le noeud $n$, c'est qu'on a trouvé un chemin de la source à ce noeud. On estime $g(n)$ par le coût de ce chemin. C'est une estimation car ce chemin n'est pas nécessairement optimal. Le terme $h(n)$ est plus problématique ; il revient au programmeur de définir une estimation adéquate.

## 2.1 Exemple de recherche d'un chemin entre deux villes

Considérons le problème suivant : nous devons nous rendre de la ville $s$ à la ville $t$ en passant par le plus court chemin.




Les nombres associés aux divers segments représentent les distances physiques séparant les villes adjacentes. Les nombres encadrés représentent les estimations $h(n)$ du chemin qu'il reste à parcourir à partir de chaque ville. Nous avons défini $h(n)$ par la distance à vol d'oiseau entre le noeud $n$ et la destination $t$. La figure de la page suivante représente l'arbre de recherche ainsi que l'ordre de parcours des noeuds.

Initialement, l'unique chemin pouvant être développé est constitué du noeud racine $s$. Nous avons le choix entre les noeuds $a$ et $e$. On obtient :

$f(a) = 2 + 5 = 7$
$f(e) = 2 + 7 = 9$

Nous choisissons le noeud $a$ car $f(a) < f(b)$. L'ensemble des chemins candidats est maintenant composé de $[s]$ et $[s, a]$.

Nous pouvons mainteant développer soit $[s, a]$ vers $b$, soit $[s]$ vers $e$.

$f(b) = 4 + 4 = 8$
$f(e) = 2 + 7 = 9$

C'est $b$ qui a la plus petite valeur, c'est donc vers ce noeud que nous nous dirigeons. L'exécution se poursuit ainsi jusqu'à obtention de la solution optimale $[s, e, f, g, t]$.

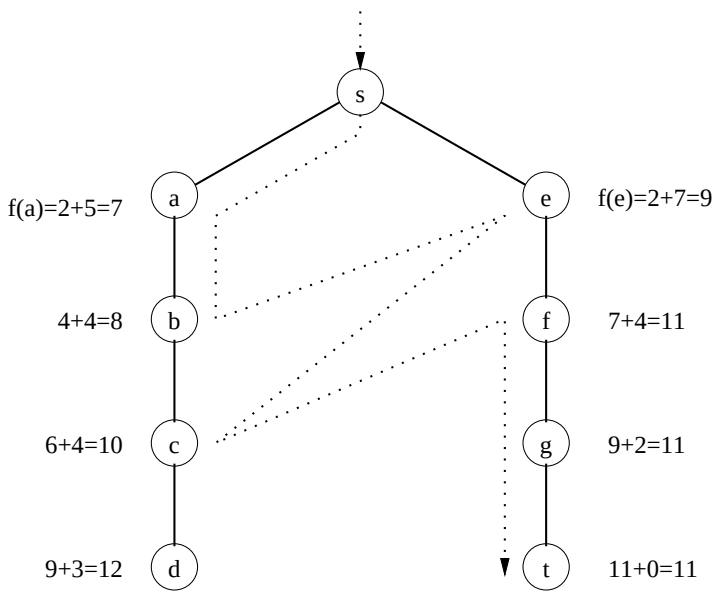



FIG. 1 – Arbre de recherche

## 2.2 Admissibilité

Un algorithme de recherche est dit *admissible* s'il produit toujours une solution optimale, en supposant qu'il existe au moins une solution. Si l'algorithme retourne plusieurs solutions, on peut dire qu'il est admissible si la première solution trouvée est optimale.

Soit, pour chaque noeud $n$ de l'espace d'états, $h^*(n)$ le coût d'un chemin optimal entre $n$ et un noeud de destination. On peut montrer que l'algorithme $A^*$ est admissible si $h(n) \leq h^*(n)$ pour tout noeud $n$.

Dans le cas de l'exemple précédent, cette inéquation est vérifiée car la distance à vol d'oiseau entre deux ville est bien toujours inférieure ou égale à la distance réelle. L'algorithme $A^*$ trouvera donc toujours le chemin le plus court.

## 2.3 Une implémentation en Prolog

Rappelons que nous devons a tout instant garder en mémoire l'arbre (les chemins candidats) généré jusqu'à présent. Par convention, nous représentons un arbre de la manière suivante :

1. **l(N,F/G)** si l'arbre est composé d'un seul noeud. $N$ est le noeud correspondant de l'espace d'états, $F$ et $G$ représentent $f(N)$ et $g(N)$ res-

pectivement.

2. **t(N,F/G,Subs)** représente un arbre dont les sous-arbres ne sont pas vides. $N$ est le noeud de l'espace d'états correspondant à la racine de l'arbre et $G$ représente $g(N)$. $F$ ne représente pas $f(N)$, mais la valeur de l'arbre entier, i.e. la plus petite valeur de ses fils (donc la plus petite valeur des feuilles de l'arbre). *Subs* est la liste des sous-arbres, par ordre croissant de valeurs.

Le principe consiste à développer l'arbre à chaque itération. Pour ce faire, on développe le sous-arbre de valeur minimale jusqu'à ce que sa valeur excède celle d'un autre sous-arbre, après quoi on passe au sous-arbre suivant. Dans le cas où l'arbre n'a pas de sous-arbre (est une feuille), on génère les successeurs avant de les développer à leur tour.

Le développement d'un arbre peut avoir trois issues différentes :

1. On a trouvé une solution. On utilisera une variable *Solved* qu'on instenciera à "yes" pour le signaler.
2. On n'a pas trouvé de solution mais l'arbre peut encore être développé. On a dû arrêter le développement car la valeur de l'arbre a dépassé la limite imposée. La variable *Solved* est instanciée à "no" pour le signaler.
3. On n'a pas trouvé de solution et l'arbre ne peut plus être développé. La variable *Solved* est instanciée à "never" pour le signaler.

### 2.3.1 Code Prolog

```
% bestfirst(+Start, -Solution)
% Solution is a path from Start to a goal

bestfirst(Start, Solution) :-
  expand([], l( Start, 0/0),  9999, _, yes, Solution).
        %  Assume 9999 is greater than any f-value
```

```
% expand(+Path, +Tree, +Bound, -Tree1, -Solved, -Solution)
% Path is path between start node of search and subtree Tree,
% Tree1 is Tree expanded within Bound,
% if goal found then Solution is solution path and Solved = yes

%  Case 1: goal leaf-node, construct a solution path

expand(P, l(N, _), _, _, yes, [N|P])  :-
   goal(N).

%  Case 2: leaf-node, f-value less than Bound
%  Generate successors and expand them within Bound.

expand(P, l(N,F/G), Bound, Tree1, Solved, Sol)  :-
  F  =<  Bound,
  (bagof(M/C, (s(N,M,C), not member(M,P)), Succ),
   !,                        % Node N has successors
   succlist(G, Succ, Ts), % Make subtrees Ts
   bestf(Ts, F1),      % f-value of best successor
   expand(P, t(N,F1/G,Ts), Bound, Tree1,  Solved, Sol)
   ;
   Solved = never % N has no successors - dead end
  ) .

%  Case 3: non-leaf, f-value less than Bound
%  Expand the most promising subtree; depending on
%  results, procedure continue will decide how to
%  proceed

expand(P, t(N,F/G,[T|Ts]), Bound, Tree1, Solved, Sol)  :-
  F  =<  Bound,
  bestf(Ts, BF), min(Bound, BF, Bound1),
  expand([N|P], T, Bound1, T1, Solved1, Sol),
  continue(P, t(N,F/G,[T1|Ts]), Bound, Tree1, Solved1, Solved, Sol).
```

```
%  Case 4: non-leaf with empty subtrees
%  This is a dead end which will never be solved

expand(_, t(_,_,[]), _, _, never, _) :- !.

%  Case 5:  f-value greater than Bound
%  Tree may not grow.

expand(_, Tree, Bound, Tree, no, _)  :-
  f(Tree, F), F > Bound.

% continue(+Path, +Tree, +Bound, -NewTree, +SubtreeSolved,
%          -TreeSolved, -Solution)

continue(_, _, _, _, yes, yes, Sol).

continue(P, t(N,F/G,[T1|Ts]), Bound, Tree1, no,
         Solved, Sol)  :-
  insert(T1, Ts, NTs),
  bestf(NTs, F1),
  expand(P, t(N,F1/G,NTs), Bound, Tree1, Solved, Sol).

continue(P, t(N,F/G,[_|Ts]), Bound, Tree1, never,
         Solved, Sol)  :-
  bestf(Ts, F1),
  expand(P, t(N,F1/G,Ts), Bound, Tree1, Solved, Sol).

% succlist(+G0, +[ Node1/Cost1, ...], -[l(BestNode,BestF/G), ...])
% make list of search leaves ordered by their F-values

succlist(_, [], []).

succlist(G0, [N/C|NCs], Ts)  :-
  G is G0 + C,
  h(N, H),  % Heuristic term h(N)
  F is G + H,
  succlist(G0, NCs, Ts1),
  insert(l(N,F/G), Ts1, Ts).
```

```
% Insert T into list of trees Ts preserving order
% with respect to f-values

insert(T, Ts, [T|Ts])  :-
  f(T, F), bestf( Ts, F1),
  F  =<  F1, !.

insert(T, [T1|Ts], [T1|Ts1])  :-
  insert(T, Ts, Ts1).


% Extract f-value

f(l(_,F/_), F).     % f-value of a leaf
f(t(_,F/_,_), F).   % f-value of a tree

bestf([T|_], F)  :- % Best f-value of a list of trees
  f(T, F).
bestf([], 9999).    % No trees: bad f-value

min(X, Y, X)  :-
  X =<  Y, !.

min(X, Y, Y).
```

## 2.4 Complexité de l'algorithme $A^*$

L'algorithme $A^*$ permet en général de réduire le nombre de noeuds de l'espace d'états explorés durant la recherche. Malgré cette amélioration, la complexité reste exponentielle en la profondeur de l'arbre de recherche. Ceci est valable pour le temps d'exécution ainsi que l'espace mémoire utilisé, vu que l'algorithme garde tous les noeuds générés en mémoire.

Il existe des variations de $A^*$ permettant d'économiser l'espace mémoire requis. Un exemple est l'algorithme $IDA^*$, (*iterative deepening* $A^*$). Cet algorithme consiste à effectuer à chaque itération une recherche en profondeur d'abord, mais limitée à une certaine valeur $f(n)$ des noeuds. Cette valeur est incrémentée d'une itératrion à la suivante.

Un autre exemple est l'algorithme $RBFS$ (*recursive best-first search*). Son fonctionnement est semblable à celui de $A^*$ mais il ne garde pas la totalité des noeuds explorés en mémoire. Plus précisément, il "oublie" un sous-arbre lorsque son développement doit être suspendu.