

## Intelligence Artificielle – Devoir n° 2

### 1 Modélisation et implémentation de l'algorithme de recherche A\*

L'algorithme A\* est une version de l'algorithme de recherche du meilleur d'abord avec graphe de recherche. Celui-ci est un algorithme informé, qui utilise une fonction d'évaluation (f) pour guider le choix du nœud à développer. Cette fonction s'applique à tous les nœuds de l'espace de recherche. Elle utilise des critères de nature heuristique (h).

Pour un nœud n, on a alors  $f(n) = g(n) + h(n)$  :

- $g(n)$  est le coût du trajet pour arriver à n.
- $h(n)$  est l'estimation du coût du trajet qu'il reste à parcourir.
- $f(n)$  est une estimation du caractère prometteur du nœud n (relativement à l'objectif visé).

Pour A\*, h est minorante, c'est à dire que  $h(n) \leq h^*(n)$  pour tout n,  $h^*(n)$  étant le coût véritable du trajet restant à parcourir.

Pour chaque nœud de l'espace de recherche, on doit alors être capable de connaître l'état qu'il représente (étiquette), son père, g, et f.

A\* gère les nœuds du graphe à l'aide de 2 ensembles, ouvert l'ensemble des nœuds à développer et fermé l'ensemble des nœuds développés. Il choisit, dans ouvert, un nœud ayant la fonction f minimum (état plus prometteur que les autres) et ajoute à ouvert ses fils à condition que aucun nœud de ouvert ou fermé ne représente la même étiquette que le fils avec une fonction f inférieure :

Ouvert <- (père0, e0, 0, h) (état initial)

Fermé <- vide

Succès <- faux

Tant que Ouvert <> vide et Succès = faux

- choix de (pèren, n, g, f) dans Ouvert tel que f(n) minimum
- Si n terminal  
  alors Succès = vrai et construire le chemin allant de e0 à n  
  sinon supprimer (pèren, n, g, f) de Ouvert et l'ajouter à Fermé

Pour chaque successeurs s de n

Si s n'appartient pas à Ouvert U Fermé

alors ajouter s à Ouvert

      père(s) <- n

$g(s) \leftarrow g(n) + k(n, s)$       ( $k(n, s)$  = coût pour aller de n à s)

sinon mise à jour de Ouvert si  $f(s) < f(\text{nœud déjà existant})$

En Prolog, un nœud sera représenté par nd(chemin, état, coût g) où chemin est la liste des états visités pour arriver à état en partant de l'état initial. Si le nœud est terminal, il suffit de renvoyer ce chemin

Il est utile de définir des prédicats d'accès aux composantes d'un nœud :

- etiquette récupère l'état représenté par un nœud.
- chemin récupère le chemin d'accès à l'état d'un nœud.
- g récupère le coût pour accéder à l'état d'un nœud.
- f calcule l'heuristique h du nœud et récupère  $f = g + h$ .

Voici le code en Prolog :

```
nd_init(nd([],_,0)). /*Le choix de l'état initial se fait à l'appel de a_etoile.*/

/*Accès aux composantes d'un nœud.*/
etiquette(nd(_,X,_),X).
chemin(nd(Ch,_,_),Ch).
g(nd(_,_,G),G).
f(nd(Ch,E,G),F):-
    heuristique(nd(Ch,E,G),H),
    F is G+H.

/*a_etoile crée le nœud correspondant à l'état initial et lance la recherche avec a_etoile_aux qui travaille avec Ouvert et Ferme.*/
a_etoile(EtatInit,CheminSol,Cout):-
    nd_init(nd([],EtatInit,0)),
    NI = nd([],EtatInit,0),
    a_etoile_aux([NI],[],CheminSol,Cout), /*Ouvert = {nœud initial} et Ferme = vide*/
    !.

/*a_etoile_aux prend le premier nœud de la liste Ouvert et vérifie s'il est terminal. Dans ce cas il construit le chemin solution. Remarquons que l'on prend bien le premier nœud de Ouvert car quand on ajoute un nœud à Ouvert, on le fait de façon à trier les nœud selon leur fonction f (voir ajoute_tri). Si le nœud n'est pas terminal, on l'ajoute à Ferme et on regarde ses fils et on les ajoute à Ouvert en faisant la vérification expliquée dans l'algorithme page 1. On relance alors a_etoile_aux avec les nouveaux Ouvert et Ferme.*/
a_etoile_aux([],_,_):- fail.
a_etoile_aux([Etat|_,_,CheminSol,Cout):-
    terminal(Etat),
    g(Etat,Cout),
    cons(Etat,CheminSol).
a_etoile_aux([Etat|ResteOuvert],Ferme,CheminSol,Cout):-
    not(terminal(Etat)),
    ajoute(Etat,Ferme,NFerme),
    findall(S,cherche_fils(Etat,S),ListeSucc),
    ajoute_verif(ListeSucc,ResteOuvert,NFerme,NOuvert),
    a_etoile_aux(NOuvert,NFerme,CheminSol,Cout).

/*cherche_fils un nœud successeur d'un état en lui donnant un Chemin et un cout.*/
cherche_fils(nd(Ch,E,G),nd(NCh,NE,NG)):-
    succ(_,E,NE,Cout),
    concat(Ch,[E],NCh),
    NG is G+Cout.

/*ajoute sert à ajouter les nœuds développés dans Ferme (il n'y a pas de contraintes).*/
ajoute(X,Liste,[X|Liste]).

/*appartient regarde si l'état d'un nœud se retrouve dans une liste sous un autre nœud. Il récupère ce nœud.*/
appartient(Etat,[nd(C,Etat,G)|_],nd(C,Etat,G)).
appartient(Etat1,[nd(C,_,G)|R],nd(C,Etat2,G)):-
    appartient(Etat1,R,nd(C,Etat2,G)).

/*ajoute_tri sert à ajouter des nœuds dans Ouvert. Il ajoute un nœud à Ouvert en fonction des valeurs de g et f du nœud.*/
ajoute_tri(X,[],[X]).
ajoute_tri(Etat1,[Etat2|R],[Etat1,Etat2|R]):-
    f(Etat1,F1),
    f(Etat2,F2),
    g(Etat1,G1),
    g(Etat2,G2),
    comparaison(F1,F2,G1,G2).
ajoute_tri(Etat1,[Etat2|R],[Etat2|NR]):-
    f(Etat1,F1),
    f(Etat2,F2),
    g(Etat1,G1),
    g(Etat2,G2),
    not(comparaison(F1,F2,G1,G2)),
    ajoute_tri(Etat1,R,NR).

/*comparaison fait les comparaisons entre les f et g de 2 nœuds pour décider de leur place dans Ouvert lors de ajoute_tri*/
comparaison(F1,F2,_):- F1<F2.
comparaison(F1,F2,G1,G2):- F1=F2,G1>=G2.

/*cons construit le chemin lorsque l'on a trouvé une solution*/
cons(nd(Ch,Etat,_),Chemin):- concat(Ch,[Etat],Chemin).
```

**/\*concaté fait la concaténation de 2 listes\*/**

```
concaté([],L,L).
concaté([X|R],L,[X|NL]):- concaté(R,L,NL).
```

**/\*retire retire simplement un élément d'une liste pour donner une nouvelle liste\*/**

```
retire(X,[X|R],R).
retire(X,[Y|R],[Y|L]):- retire(X,R,L).
```

**/\*ajoute\_verif(ListeSucc,Ouvert,Ferme,NOuvert) vérifie un par un les éléments de ListeSucc pour savoir si ils appartiennent au ensemble Ouvert ou Ferme et selon le cas il ajoute à Ouvert les nœuds qui ont correctement passés les vérifications, c'est à dire, par exemple, si un EtatSucc appartient déjà à Ouvert, il faut que son f soit inférieur pour que l'on retire l'ancien de Ouvert et qu'on le remplace par ce nouvel état.\*/**

```
ajoute_verif([],X,_,X).
ajoute_verif([EtatSucc|ResteListeSucc],Ouvert,Ferme,NNOuvert):-/*l'état n'appartient pas aux ensembles*/
    etiquette(EtatSucc,E),
    not(appartient(E,Ouvert,_)),
    not(appartient(E,Ferme,_)),
    ajoute_tri(EtatSucc,Ouvert,NOuvert),
    ajoute_verif(ResteListeSucc,NOuvert,Ferme,NNOuvert).
ajoute_verif([nd(C1,Etat1,G1)|ResteListeSucc],Ouvert,Ferme,NNNOuvert):-/*l'état appartient à Ouvert et son f est inférieur*/
    ap_ouvert(Etat1,Ouvert,Ferme,nd(C2,Etat2,G2)),
    f(nd(C1,Etat1,G1),F1),
    f(nd(C2,Etat2,G2),F2),
    comparaison(F1,F2,G1,G2),
    retire(nd(C2,Etat2,G2),Ouvert,NOuvert),
    ajoute_tri(nd(C1,Etat1,G1),NOuvert,NNNOuvert),
    ajoute_verif(ResteListeSucc,NNNOuvert,Ferme,NNNOuvert).
ajoute_verif([nd(C1,Etat1,G1)|ResteListeSucc],Ouvert,Ferme,NNNOuvert):-/*l'état appartient à Ferme mais pas à Ouvert et son f
est inférieur*/
    appartient(Etat1,Ferme,nd(C2,Etat2,G2)),
    not(appartient(Etat1,Ouvert,_)),
    f(nd(C1,Etat1,G1),F1),
    f(nd(C2,Etat2,G2),F2),
    comparaison(F1,F2,G1,G2),
    ajoute_tri(nd(C1,Etat1,G1),Ouvert,NOuvert),
    ajoute_verif(ResteListeSucc,NOuvert,Ferme,NNNOuvert).

ajoute_verif([nd(C1,Etat1,G1)|_],Ouvert,Ferme,NOuvert):-/*l'état appartient à Ouvert et son f est supérieur*/
    ap_ouvert(Etat1,Ouvert,Ferme,nd(C2,Etat2,G2)),
    f(nd(C1,Etat1,G1),F1),
    f(nd(C2,Etat2,G2),F2),
    not(comparaison(F1,F2,G1,G2)),
    NOuvert = Ouvert.
ajoute_verif([nd(C1,Etat1,G1)|_],Ouvert,Ferme,NOuvert):-/*l'état appartient à Ferme mais pas à Ouvert et son f est supérieur*/
    appartient(Etat1,Ferme,nd(C2,Etat2,G2)),
    not(appartient(Etat1,Ouvert,_)),
    f(nd(C1,Etat1,G1),F1),
    f(nd(C2,Etat2,G2),F2),
    not(comparaison(F1,F2,G1,G2)),
    NOuvert = Ouvert.
```

**/\*ap\_ouvert différencie les 2 cas où un état appartient à Ouvert : soit l'état appartient à Ouvert et pas à Ferme, soit il appartient aux deux.\*/**

```
ap_ouvert(Etat1,Ouvert,Ferme,nd(C2,Etat2,G2)):-
    appartient(Etat1,Ouvert,nd(C2,Etat2,G2)),
    not(appartient(Etat1,Ferme,_)).
ap_ouvert(Etat1,Ouvert,Ferme,nd(C2,Etat2,G2)):-
    appartient(Etat1,Ferme,_),
    appartient(Etat1,Ouvert,nd(C2,Etat2,G2)).
```

Je n'ai pas trouvé de moyen plus court pour coder ajoute\_verif, qui prend, je m'en rend compte, beaucoup de place pour différencier tous les cas.

On peut remarquer que utiliser un ensemble Ferme n'est pas obligatoire mais que cela évite de nombreuses redondances.

## 2 Le jeu de taquin 3x3

Un état au jeu de taquin sera représenté par une liste des 9 cases du jeu dans un ordre précis :

Pour un jeu dans la position

A	B	C
D	E	F
G	H	I

on aura la représentation [A,B,C,D,E,F,G,H,I].

L'état que l'on veut atteindre est

1	2	3
8		4
7	6	5

c'est à dire [1,2,3,8,0,4,7,6,5].

Le symbole 0 sera utilisé pour la case vide mais elle ne sera pas gérée au même titre que les autres cases.

Voici la liste de toutes les opérations que l'on peut effectuer selon les différentes configurations des états possibles. On utilise pour cela 4 opérateurs Haut, Bas, Gauche et Droite qui indique quelle case à côté de la case vide va bouger.

```
succ(gauche,[0,A,B,C,D,E,F,G,H],[A,0,B,C,D,E,F,G,H],1).
succ(haut,[0,A,B,C,D,E,F,G,H],[C,A,B,0,D,E,F,G,H],1).
```

```
succ(droite,[A,0,B,C,D,E,F,G,H],[0,A,B,C,D,E,F,G,H],1).
succ(gauche,[A,0,B,C,D,E,F,G,H],[A,B,0,C,D,E,F,G,H],1).
succ(haut,[A,0,B,C,D,E,F,G,H],[A,D,B,C,0,E,F,G,H],1).
```

```
succ(droite,[A,B,0,C,D,E,F,G,H],[A,0,B,C,D,E,F,G,H],1).
succ(haut,[A,B,0,C,D,E,F,G,H],[A,B,E,C,D,0,F,G,H],1).
```

```
succ(bas,[A,B,C,0,D,E,F,G,H],[0,B,C,A,D,E,F,G,H],1).
succ(gauche,[A,B,C,0,D,E,F,G,H],[A,B,C,D,0,E,F,G,H],1).
succ(haut,[A,B,C,0,D,E,F,G,H],[A,B,C,F,D,E,0,G,H],1).
```

```
succ(droite,[A,B,C,D,0,E,F,G,H],[A,B,C,0,D,E,F,G,H],1).
succ(bas,[A,B,C,D,0,E,F,G,H],[A,0,C,D,B,E,F,G,H],1).
succ(gauche,[A,B,C,D,0,E,F,G,H],[A,B,C,D,E,0,F,G,H],1).
succ(haut,[A,B,C,D,0,E,F,G,H],[A,B,C,D,G,E,F,0,H],1).
```

```
succ(bas,[A,B,C,D,E,0,F,G,H],[A,B,0,D,E,C,F,G,H],1).
succ(droite,[A,B,C,D,E,0,F,G,H],[A,B,C,D,0,E,F,G,H],1).
succ(haut,[A,B,C,D,E,0,F,G,H],[A,B,C,D,E,H,F,G,0],1).
```

```
succ(bas,[A,B,C,D,E,F,0,G,H],[A,B,C,0,E,F,D,G,H],1).
succ(gauche,[A,B,C,D,E,F,0,G,H],[A,B,C,D,E,F,G,0,H],1).
```

```
succ(droite,[A,B,C,D,E,F,G,0,H],[A,B,C,D,E,F,0,G,H],1).
succ(gauche,[A,B,C,D,E,F,G,0,H],[A,B,C,D,E,F,G,H,0],1).
succ(bas,[A,B,C,D,E,F,G,0,H],[A,B,C,D,0,F,G,E,H],1).
```

```
succ(bas,[A,B,C,D,E,F,G,H,0],[A,B,C,D,E,0,G,H,F],1).
succ(droite,[A,B,C,D,E,F,G,H,0],[A,B,C,D,E,F,G,0,H],1).
```

```
terminal(nd(_,[1,2,3,8,0,4,7,6,5],_)).
```

### Heuristique 1 : Nombre de cases mal placées.

```
heuristique([1,2,3,8,0,4,7,6,5],0):-!.
heuristique([A,B,C,D,E,F,G,I,J],H):-
```

```
    compare(A,1,R1),
    compare(B,2,R2),
    compare(C,3,R3),
    compare(D,8,R4),
    compare(E,0,R5),
    compare(F,4,R6),
    compare(G,7,R7),
    compare(I,6,R8),
    compare(J,5,R9),
    H is R1+R2+R3+R4+R5+R6+R7+R8+R9.
```

```
compare(0,_,0):-!.
compare(Lettre,Lettre,0).
compare(Lettre,Case,1):-not(Lettre=Case).
```

Cette heuristique se contente de compter le nombre de cases d'un état [A,B,C,D,E,F,G,I,J] qui ne sont pas placées comme dans l'état final [1,2,3,8,0,4,7,6,5].

### **Heuristique 2 : Distance en escalier des cases mal placées.**

```
heuristique([1,2,3,8,0,4,7,6,5],0):-!.
heuristique([A,B,C,D,E,F,G,I,J],H):-
    compare(A,1,R1,1),
    compare(B,2,R2,2),
    compare(C,3,R3,3),
    compare(D,8,R4,4),
    compare(E,0,R5,5),
    compare(F,4,R6,6),
    compare(G,7,R7,7),
    compare(I,6,R8,8),
    compare(J,5,R9,9),
    H is R1+R2+R3+R4+R5+R6+R7+R8+R9.
```

```
compare(0,_,0,_) :- !.
compare(Lettre,Lettre,0,_) .
compare(Lettre,Case,D,N):-
    not(Lettre=Case),
    place(Lettre,[1,2,3,8,0,4,7,6,5],N,D,1).
```

```
place(Lettre,[X|_],N,D,I):-
    Lettre = X,
    dist(I,N,D).
place(Lettre,[X|R],N,D,I):-
    not(Lettre = X),
    I is I+1,
    place(Lettre,R,N,D,I).
```

```
dist(1,1,0). dist(1,2,1). dist(1,3,2). dist(1,4,1). dist(1,5,2). dist(1,6,3). dist(1,7,2). dist(1,8,3). dist(1,9,4).
dist(2,1,1). dist(2,2,0). dist(2,3,1). dist(2,4,2). dist(2,5,1). dist(2,6,2). dist(2,7,3). dist(2,8,2). dist(2,9,3).
dist(3,1,2). dist(3,2,1). dist(3,3,0). dist(3,4,3). dist(3,5,2). dist(3,6,1). dist(3,7,4). dist(3,8,3). dist(3,9,2).
dist(4,1,1). dist(4,2,2). dist(4,3,3). dist(4,4,0). dist(4,5,1). dist(4,6,2). dist(4,7,1). dist(4,8,2). dist(4,9,3).
dist(5,1,2). dist(5,2,1). dist(5,3,2). dist(5,4,1). dist(5,5,0). dist(5,6,1). dist(5,7,2). dist(5,8,1). dist(5,9,2).
dist(6,1,3). dist(6,2,2). dist(6,3,1). dist(6,4,2). dist(6,5,1). dist(6,6,0). dist(6,7,3). dist(6,8,2). dist(6,9,1).
dist(7,1,2). dist(7,2,3). dist(7,3,4). dist(7,4,1). dist(7,5,2). dist(7,6,3). dist(7,7,0). dist(7,8,1). dist(7,9,2).
dist(8,1,3). dist(8,2,2). dist(8,3,3). dist(8,4,2). dist(8,5,1). dist(8,6,2). dist(8,7,1). dist(8,8,0). dist(8,9,1).
dist(9,1,4). dist(9,2,3). dist(9,3,2). dist(9,4,3). dist(9,5,2). dist(9,6,1). dist(9,7,2). dist(9,8,1). dist(9,9,0).
```

Pour chaque cases mal placées, cet heuristique cherche la place qu'elle devrait avoir et la compare à sa place actuel, puis retient la distance entre ces 2 places. A la fin, elle renvoie la somme des distances.

### **Heuristique 3 : Nombre d'éléments du pourtour qui casse l'ordre croissant.**

```
heuristique([1,2,3,8,0,4,7,6,5],0):-!.
heuristique([A,B,C,D,_,F,G,I,J],H):-
    L = [A,B,C,F,J,I,G,D],
    verif_liste_triee(L,0,H).

verif_liste_triee([_],I,I).
verif_liste_triee([0|R],I,N):-
    NI is I+1,
    verif_liste_triee(R,I,NN),
    N is NI+NN.
verif_liste_triee([X,Y|R],I,N):-
    not(X=0),
    X<Y,
    verif_liste_triee([Y|R],I,N).
verif_liste_triee([X,Y|R],I,N):-
    X>Y,
    NI is I+1,
    verif_liste_triee([X|R],I,NN),
    N is NI+NN.
```

Cette heuristique compte 1 à chaque fois qu'une case de la liste d'un état casse le caractère croissant du pourtour du jeu (et 1 si la case vide apparaît dans le pourtour). En effet, la liste du pourtour pour l'état final est [1,2,3,4,5,6,7,8].

Seul problème, pour l'état [0,1,2,7,0,3,6,5,4] de pourtour [0,1,2,3,4,5,6,7], on aura une estimation de 1 alors que intuitivement, on voit bien qu'il reste environ 7 coût à jouer.

### **Test des 3 heuristiques :**

Sur l'état initial [6,1,3,2,0,4,8,7,5], les 3 heuristiques ont permis de trouver le même chemin solution pour atteindre l'état final [1,2,3,8,0,4,7,6,5].

Ce chemin est [6,1,3,2,0,4,8,7,5] (1), [6,1,3,0,2,4,8,7,5] (2), [0,1,3,6,2,4,8,7,5] (3), [1,0,3,6,2,4,8,7,5] (4), [1,2,3,6,0,4,8,7,5] (5), [1,2,3,0,6,4,8,7,5] (6), [1,2,3,8,6,4,0,7,5] (7), [1,2,3,8,6,4,7,0,5] (8), [1,2,3,8,0,4,7,6,5] (9).

Pour chaque état les réponses heuristiques sont :

Heuristique 1 :  $h(1) = 5, h(2) = 5, h(3) = 5, h(4) = 4, h(5) = 3, h(6) = 3, h(7) = 2, h(8) = 1, h(9) = 0$ .

Heuristique 2 :  $h(1) = 8, h(2) = 7, h(3) = 6, h(4) = 5, h(5) = 4, h(6) = 3, h(7) = 2, h(8) = 1, h(9) = 0$ .

Heuristique 3 :  $h(1) = 2, h(2) = 2, h(3) = 2, h(4) = 2, h(5) = 1, h(6) = 1, h(7) = 1, h(8) = 1, h(9) = 0$ .

### **Coût restant réellement :**

$h^*(1) = 8, h^*(2) = 7, h^*(3) = 6, h^*(4) = 5, h^*(5) = 4, h^*(6) = 3, h^*(7) = 2, h^*(8) = 1, h^*(9) = 0$ .

Plus une heuristique est petite et plus le caractère prometteur d'un état est grand. Si la fonction d'évaluation sait être le minimum pour de bons états, elle sera efficace.

Pour cela, on pourrait penser que h3 est plus efficace que les 2 autres heuristiques mais elle donne des résultats petits même pour certains états où il reste pas mal de coups à jouer en réalité. En quelque sorte, h3 estime que n'importe quel état a autant de chance d'être un état du chemin solution.

On remarque alors que h2 a su estimer les coûts restant à parcourir par les coût réels. h2 est donc une bonne heuristique, ainsi que h1 qui devient de moins en moins précise quand on s'éloigne du but.

### 3 Le problème du voyageur de commerce

Un état pour ce problème se représente simplement par le nom de la ville.

Un nœud est solution si le chemin correspondant à l'état du nœud passe par toutes les villes.

Pour tester les différentes heuristiques, nous prendrons l'exemple suivant :

A	B	C	D
21			
12	7		
15	25	25	

(4 villes A, B, C et D avec les coûts indiqués entre elles)

En Prolog :

```
succ(vdc,a,b,21).
succ(vdc,a,c,12).
succ(vdc,a,d,15).
succ(vdc,b,a,21).
succ(vdc,b,c,7).
succ(vdc,b,d,25).
succ(vdc,c,a,12).
succ(vdc,c,b,7).
succ(vdc,c,d,25).
succ(vdc,d,a,15).
succ(vdc,d,b,25).
succ(vdc,d,c,25).
```

```
terminal(nd(Chemin,a,_):-
  member(a,Chemin),
  member(b,Chemin),
  member(c,Chemin),
  member(d,Chemin).
```

#### Heuristique 2 : Somme des n plus petits arcs du graphe.

```
heuristique(nd(Ch,a,_),H):-
  compte_nb_villes_restantes(Ch,[a,b,c,d],0,N),
  retire_n(N,[21,12,15,7,25],[],Lppa),
  somme(Lppa,H).
heuristique(nd(Ch,b,_),H):-
  compte_nb_villes_restantes(Ch,[a,b,c,d],0,N),
  retire_n(N,[21,12,15,7,25],[],Lppa),
  somme(Lppa,H).
heuristique(nd(Ch,c,_),H):-
  compte_nb_villes_restantes(Ch,[a,b,c,d],0,N),
  retire_n(N,[21,12,15,7,25],[],Lppa),
  somme(Lppa,H).
heuristique(nd(Ch,d,_),H):-
  compte_nb_villes_restantes(Ch,[a,b,c,d],0,N),
  retire_n(N,[21,12,15,7,25],[],Lppa),
  somme(Lppa,H).

somme([],0).
somme([X|R],S):-
  somme(R,S2),
  S is S2+X.

compte_nb_villes_restantes(_,[],N,N).
compte_nb_villes_restantes(Chemin,[X|R],N,NNN):-
  not(member(X,Chemin)),
  NN is N+1,
  compte_nb_villes_restantes(Chemin,R,NN,NNN).
compte_nb_villes_restantes(Chemin,[X|R],N,NN):-
  member(X,Chemin),
  compte_nb_villes_restantes(Chemin,R,N,NN).
```

```

retire_n(0,_,Lppa,Lppa).
retire_n(N,Lnd,Lppa,NLppa):-
    rech_plus_petit(Lnd,X),
    retire(X,Lnd,NLnd),
    ajoute(X,Lppa,Lppa2),
    NN is N-1,
    retire_n(NN,NLnd,Lppa,Lppa3),
    concat(Lppa2,Lppa3,NLppa).

rech_plus_petit([X],X).
rech_plus_petit([X,Y|R],E):-
    X<Y,
    rech_plus_petit([X|R],E).
rech_plus_petit([X,Y|R],E):-
    X>=Y,
    rech_plus_petit([Y|R],E).

```

compte\_nb\_villes\_restantes fait ce que son nom indique à l'aide d'une liste de villes (toutes celles du graphe) et du chemin déjà parcouru jusqu'à l'état considéré.  
 retire\_n retire n fois l'arête la plus petite du graphe en l'enlevant à chaque fois.  
 rech\_plus\_petit récupère le plus petit élément d'une liste.

#### **Heuristique 4 : Pour tout n, $h(n) = 0$ .**

```

heuristique(a,0).
heuristique(b,0).
heuristique(c,0).
heuristique(d,0).

```

On utilise en fait ici pas d'heuristique. Un nœud est choisi dans Ouvert selon son coût g. Plus le chemin parcouru jusqu'à un nœud est petit et plus on estime que ce chemin est le début du meilleur.

h est forcément minorante, puisque toujours égale à 0.

Si le dernier nœud à visiter nous coûte très cher, alors cette heuristique nous a fait perdre du temps à cause d'une mauvaise estimation.

#### **Heuristique 5 : $n \times$ coût de l'arc minimum du graphe.**

```

heuristique(nd(Ch,a,_),H):-
    compte_nb_villes_restantes(Ch,[a,b,c,d],0,N),
    rech_plus_petit([21,12,15,7,25],E),
    H is N*E.
heuristique(nd(Ch,b,_),H):-
    compte_nb_villes_restantes(Ch,[a,b,c,d],0,N),
    rech_plus_petit([21,12,15,7,25],E),
    H is N*E.
heuristique(nd(Ch,c,_),H):-
    compte_nb_villes_restantes(Ch,[a,b,c,d],0,N),
    rech_plus_petit([21,12,15,7,25],E),
    H is N*E.
heuristique(nd(Ch,d,_),H):-
    compte_nb_villes_restantes(Ch,[a,b,c,d],0,N),
    rech_plus_petit([21,12,15,7,25],E),
    H is N*E.

```

Cette heuristique fonctionne un peu comme la 2 (Somme des n plus petits arcs du graphe).

Elle est minorante car s'il reste n nœuds à parcourir et que chacun de ces nœuds coûte le plus petit coût possible, alors on ne peut en aucun cas dépasser la valeur réelle du coût qu'il reste à parcourir.

#### **Test des 3 heuristiques :**

Sur l'état initial a, les 3 heuristiques ont permis de trouver le même chemin solution pour atteindre l'état final a en passant par toutes les villes une et une seule fois.

Ce chemin est [a, c, b, d, a] de coût 59.

Pour chaque état les réponses heuristiques sont :

Heuristique 2 :  $h(a) = 59$ ,  $h(c) = 34$ ,  $h(b) = 19$ ,  $h(d) = 7$ ,  $h(a) = 0$ .

Heuristique 4 :  $h(a) = 0$ ,  $h(c) = 0$ ,  $h(b) = 0$ ,  $h(d) = 0$ ,  $h(a) = 0$ .

Heuristique 5 :  $h(a) = 28$ ,  $h(c) = 21$ ,  $h(b) = 14$ ,  $h(d) = 7$ ,  $h(a) = 0$ .



Coût restant réellement :

$h^*(a) = 59$ ,  $h^*(c) = 47$ ,  $h^*(b) = 40$ ,  $h^*(d) = 15$ ,  $h^*(a) = 0$ .

Comme on l'a vu précédemment, l'heuristique nulle estime que si le chemin déjà parcouru jusqu'à un état considéré est minimum, alors il a de grande chance d'être le bon chemin. Cette heuristique n'est pas du tout efficace ; par exemple sur un graphe ayant des états que l'on ne peut atteindre qu'à partir d'un certain moment et dont le coût des arêtes y amenant est très grande.

Les heuristiques 2 et 5 ont la même façon de raisonner mais la 2 fait une meilleure estimation car elle se base sur  $n$  arêtes différentes ( $n$  est le nombre de villes qu'il reste à visiter) qui pourraient être empruntées alors que la 1 se base  $n$  fois sur la même arête (alors qu'une même arête ne va pas être utilisée  $n$  fois).