TRAVAUX PRATIQUES D'I.A. N° 1

ALGORITHME A* - APPLICATION AU TAQUIN

OBJECTIFS PEDAGOGIQUES

L'objectif est d'implémenter l'algorithme A* sous une forme générique et efficace en utilisant des structures de données adaptées afin de bénéficier d'opérations sur les états (recherche, insertion, suppression, modification) de faible complexité algorithmique.

L'algorithme doit être réutilisable pour tout type de problème modélisable comme la recherche d'un chemin optimal dans un graphe d'états, à condition de définir dans un fichier dédié :

- l'état initial,
- l'état final,
- les actions possibles pour changer d'état,
- le coût de chaque action,
- l'heuristique estimant la distance d'un état donné à un état final.

On étudiera expérimentalement le comportement et les limites de l'algorithme sur le problème du Taquin, en essayant :

- 2 heuristiques :
  - *le nombreb de pièces mal placées* dans la position courante par rapport à la situation désirée
  - *la somme des distances de Manhattan* de chaque pièce depuis sa position courante vers la position désirée.
- plusieurs niveaux de difficulté (solution en 2 coups, en 10 coups, 30 coups …),
- plusieurs tailles de problèmes (3×3 , 4×4 …).

TRAVAIL A EFFECTUER

1 Familiarisation avec le problème du Taquin 3×3

Le problème "*fil rouge*" utilisé pour la mise au point du programme A* est la recherche de la séquence d'actions optimale pour résoudre un taquin 3×3 défini par ses situations initiale (U_0) et finale (F) ci-dessous à gauche. Le programme doit pouvoir être facilement adapté à un taquin 4×4, dont un exemple de situation finale est donnée ci-dessous à droite.

b	h	c
a	f	d
g		e

Taquin 3×3
Situation initiale U_0

a	b	c
h		d
g	f	e

Taquin 3×3
Situation finale F

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Taquin 4×4
Situation finale F

1.1 Créer un répertoire de travail pour le tp n°1 d'I.A. et copier le fichier **taquin.pl**. présent sur moodle.insa-toulouse.fr (voir sujet de TP n°1 dans votre répertoire de travail.

1.2 Pour se familiariser avec la modélisation adoptée, charger le programme en Prolog et répondre aux questions suivantes :

a) Quelle clause Prolog permettrait de représenter la situation finale du Taquin 4x4 ?

b) A quelles questions permettent de répondre les requêtes suivantes :

```
?- initial_state(Ini), nth1(L,Ini,Ligne), nth1(C,Ligne, d).
?- final_state(Fin), nth1(3,Fin,Ligne), nth1(2,Ligne,P)
```

c) Quelle requête Prolog permettrait de savoir si une pièce donnée P (ex : a) est bien placée dans U_0 (par rapport à F) ?

On s'intéresse maintenant au prédicat **`rule/4`**. Comment l'utiliser pour répondre aux questions suivantes :

d) quelle requête permet de trouver une situation suivante de l'état initial du Taquin 3×3 (3 sont possibles) ?

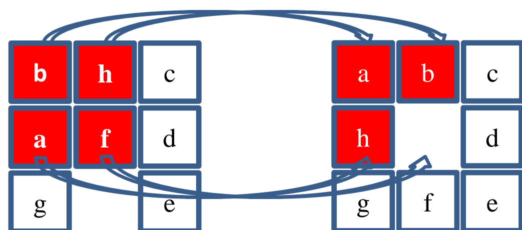
e) quelle requête permet d'avoir ces 3 réponses regroupées dans une liste ? (*cf.* **`findall/3`** en Annexe).

f) quelle requête permet d'avoir la liste de tous les couples [A, S] tels que S est la situation qui résulte de l'action A en U_0 ?

2. Développement des 2 heuristiques

2.1 L'heuristique du *nombre de pièces mal placées*.

Le nombre de déplacements restant à effectuer est estimé au *nombre de pièces mal placées* (comme si chaque pièce mal placée pouvait rejoindre sa place finale en 1 seule action). Pour calculer $h_1(U)$ on compare les matrices représentant les situations *U* et *F* et on compte combien de termes de *U* sont différents en *F* (l'emplacement vide n'est pas compté) ; exemple : calcul de $h_1(U_0)$.



situation initiale U_0 situation finale *F*

$h_1(U_0)$ = |{b, h, a, f}| = 4

Plus formellement,

$$h_1(U) = \sum_{X=1,2,3} \sum_{Y=1,2,3} diff(U_{XY}, F_{XY})$$

avec :

$$diff(vide, X) = 0 \quad \forall X$$

pour ne pas compter l'emplacement vide.

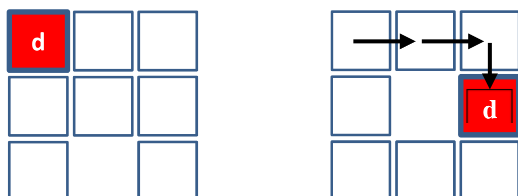
Le prédicat Prolog **`heuristique1(U, H)`** doit associer à un état U son évaluation heuristique selon la 1ère méthode.
Pour le coder, au moins 2 approches sont possibles (au choix) :

a. rechercher la liste de toutes les *pièces mal placées* (cf. prédicat **`findall`**) et calculer sa taille (cf. prédicat **`length`**). Une pièce (attention '**`vide`**' n'en est pas une) est *mal placée* en U si le terme de mêmes cordonnées dans l'état final F est un terme différent. Le prédicat liant un terme à ses coordonnées dans une matrice doit donc être défini.

b. évaluer récursivement le nombre de *différences entre deux matrices* :
si les 2 matrices sont vides (**`M1=M2=[]`**) la différence est 0 (cas trivial) ; sinon les différences entre les matrices **`M1=[L1|R1]`** et **`M2=[L2|R2]`** sont la somme des différences entre lignes **`L1`** et **`L2`** et des différences entre sous-matrices **`R1`** et **`R2`**. Les *différences entre deux lignes* peuvent être aussi établies récursivement : 0 si elles sont vides, sinon il faut évaluer la somme de la différence entre les 1er éléments et des différences entre sous-listes. Finalement, la *différence entre deux éléments* E1 et E2 c'est :
   - 1 si E1 \= E2 et si E1 \= vide (E1= vide n'est pas comptabilisé car ce n'est pas une pièce).
   - 0 sinon

2.2 L'heuristique basée sur la *distance de Manhattan*.

Cette heuristique évalue pour chaque pièce la distance minimale à parcourir pour rejoindre sa position finale ; ex :



Une situation U La situation finale F

$$DM(d, U, F) = 3$$

Plus formellement,

$$h_2(U) = \sum_{p=a,b,c,d,e,f,g,h} DM(p, U, F)$$

avec :

$$DM(p, U, F) = |X_F - X_U| + |Y_F - Y_U|$$

Le prédicat Prolog **`heuristique2(U, H)`** doit associer à un état U son évaluation heuristique selon la 2ème méthode.
Pour le coder il est conseillé de définir d'abord le prédicat **`coordonnees([L,C], Mat, Elt)`**.

Conseils et rappels

- Le prédicat **`findall(T,Goal, L)`** (*cf.* Annexe) permet de déterminer la liste des termes **`T`** qui satisfont la requête **`Goal`**. Ceci pourra s'avérer très utile pour établir la liste des pièces mal placées.
- En Prolog, le littéral **`V is abs(Expr)`** unifie la variable **`V`** à la valeur absolue de l'expression arithmétique **`Expr`**.

Ceci intervient dans le calcul de la distance de Manhattan liant la position d'un pièce E en U et celle en F.

- Le prédicat `nth1(Pos,List,Elt)` associe une position et un élément dans la liste List ; ex :

```
?- nth1(Pos,[a,b,c],Elt).
Pos = 1, Elt = a ;
Pos = 2, Elt = b ;
Pos = 3, Elt = c ;
no
```

- `sumlist(List, Sum)`, calcule la somme de tous les entiers de la liste `List`.

Aller à la fin du fichier `taquin.pl` et coder les prédicats logiques `heuristique1(U,H)` et `heuristique2(U,H)`.
Ces prédicats ont été temporairement implémentés par un "bouchon" (`true`) pour passer la compilation.

Tester les deux heuristiques sur les deux situations extrêmes (U_0 et F).

3. Implémentation de A*

3.1 Implémentation de P et Q par des arbres AVL

A* fait évoluer à chaque itération deux ensembles :
- l'**ensemble des états pendants, P** ; il s'agit des états non encore développés
- l'**ensemble des états clos, Q** ; il s'agit des états déjà développés.

P et Q ont des utilisations différentes :
- P contient la prochaine situation U à développer : celle qui a la valeur f(U) minimale ; d'autre part, après développement on effectue des tests d'appartenance sur P pour déterminer si les successeurs S de U sont des états en attente déjà rencontrés dans le passé, auquel cas il faut éventuellement mettre à jour les informations f(S), g(S) et pere(S).
- Q mémorise les situations C déjà développées et pour chacune la situation parente et l'action qui relie la situation parente à C ; il faut aussi faire des tests d'appartenance lorsqu'on veut savoir si un état a déjà été développé dans le passé et dans ce cas l'oublier (on ne doit pas développer 2 fois la même situation). Enfin Q permet de reconstruire la solution lorsque l'algorithme atteint l'état terminal F : pour reconstruire le chemin optimal il suffit de remonter depuis l'état final F vers l'état initial (à l'envers) en parcourant les liens pere(F), pere(pere(F)), … etc … jusqu'à I.

Au total, il faut réaliser les opérations suivantes sur les ensembles P et Q :
- recherche et suppression de l'état U qui minimise fU)
- pour chaque successeur d'un état U qui vient d'être développé, test d'appartenance à P et test d'appartenance à Q
- insertion dans P d'une nouvelle situation U
- mise à jour des valeurs f(S) g(S) père(S) d'un etat S de P et reclassement

L'implémentation de P et Q la plus favorable pour l'ensemble des opérations à effectuer est **l'arbre binaire de recherche équilibré (AVL)**. Recherche (appartenance), Insertion, Suppression et Modification sont de complexité $O(logN)$.
Un tas (heap) binaire aurait été intéressant pour rechercher l'état qui minimise f(U) avec une complexité de $O(1)$. Malheureusement comme il faut aussi disposer de la possibilité de mettre à jour des états de P lorsque leur valeur *f* diminue, ls structrure de tas ne convient pas car la mise à jour d'un élément quelconque d'un tas n'est pas une opération offerte par un tas. En revanche mettre à jour un élément dans un arbre binaire de recherche équilibré ne coûte que $O(logN)$.

Copier le fichier `avl.pl` fourni dans le répertoire du TP1 sur moodle.insa-toulouse.fr et sauvegarder dans votre répertoire. Lire la spécification des opérations fournies.

Dans ce module, les AVL sont des arbres de termes Prolog ordonnés lexicographiquement. En prolog la relation d'ordre lexicographique entre deux termes `T1` et `T2` quelconques est dénotée : `T1 @< T2` (voir Annexe).
Ouvrir ce fichier et tester les primitives disponibles (empty/1, insert/3, suppress/3, suppress_min/3, put_flat/3 …) à l'aide des exemples d'AVL fournis en fin de fichier.

Par définition un AVL ne peut pas être ordonné que par une seule relation d'ordre. Or, on a pourtant besoin :
- **d'ordonner P selon les valeurs [F, H]** croissantes afin de déterminer facilement l'état qui minimise f(U) ; dans le même temps on a également besoin de rechercher un état donné dans P et donc
- **d'ordonner P par situation croissante (lexicographiquement)**.

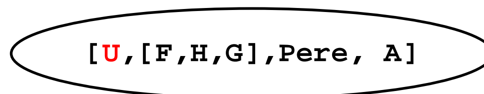
On décide donc d'utiliser non pas 1 mais 2 AVL (chaque état sera donc représenté 2 fois) en dédoublant P en 2 AVL, **Pf** et **Pu** :

Pf est un AVL représentant P, dont chaque nœud est un terme du type :

ex : **[[4,4,0],[[b,h,c],[a,f,d],[g,vide,e]]]** pour U_0 .

Les nœuds de l'arbre **Pf** sont ordonnés par valeurs F, puis H croissantes.
Le noeud « minimal » (qui possède la valeur F minimale) est le nœud situé le plus à gauche dans l'arbre.

Pu est un AVL représentant P, dont chaque nœud est un terme du type :



ex :**[[[b,h,c],[a,f,d],[g,vide,e]], [4,4,0], nil, nil]** pour U_0.

(**nil** signifie arbitrairement « aucun »).

Les nœuds de l'arbre **Pu** sont ordonnés lexicographiquement (selon la description de l'état).

Enfin, pour mémoriser les états développés et pouvoir les retrouver efficacement, on implémente **Q** par un AVL donc chaque nœud est du même type que ceux de **Pu** : les nœuds sont ordonnés en fonction de la description des états.

En résumé :
Pf sera l'AVL utilisé pour trouver facilement le prochain état à développer par A* (celui ayant f minimale).
Pu sera l'AVL permettant de mémoriser tous les états non encore développés, de répondre facilement à la question U∈P ? et de mettre à jour les valeurs [F,H,G] lorsqu'un meilleur chemin passant par U est trouvé par A*
Q permettra de réaliser facilement tests d'appartenance du type U∈Q pour affirmer qu'un état a déjà été développé par A*.

Désormais, <u>toute modification de P doit être dédoublée (dans Pf et dans Pu</u>).

3.2 Algorithme A* adapté aux structures AVL choisies

Le développement suivra le plan suivant :

Prédicat **main/0**

Ce prédicat représente le programme principal, qui doit :
- fixer la situation de départ S0 (l'état initial est défini dans le fichier **taquin.pl**)
- calculer les valeurs F0, H0, G0 pour cette situation
- créer 3 AVL Pf, Pu et Q initialement vides
- insérer un noeud **[[F0,H0,G0], S0]** dans Pf et un noeud **[S0, [F0,H0,G0], nil, nil]** dans Pu
- appeler **aetoile** sur Pf, Pu et Q

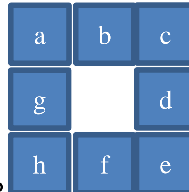
Prédicat **aetoile/3**
----
aetoile est défini récursivement :
- (cas trivial) si Pf et Pu sont vides, il n'y a aucun état pouvant être développé donc pas de solution au problème. Dans ce cas il faut afficher un message clair : « PAS de SOLUTION : L'ETAT FINAL N'EST PAS ATTEIGNABLE ! ».
- (cas trivial) si le nœud de valeur F minimum de Pf correspond à la situation terminale, alors on a trouvé une solution et on peut l'afficher (prédicat **affiche_solution**)
- (cas général) sinon
  - on enlève le nœud de Pf correspondant à l'état U à développer (celui de valeur F minimale) et on enlève aussi le nœud frère associé dans Pu
  - développement de U
    - déterminer tous les nœuds contenant un état successeur S de la situation U et calculer leur évaluation [Fs, Hs, Gs] (prédicat **expand**) connaissant Gu et le coût pour passer de U à S.
    - traiter chaque nœud successeur (prédicat **loop_successors**) :
      - si S est connu dans Q alors oublier cet état (S a déjà été développé)
      - si S est connu dans Pu alors garder le terme associé à la meilleure évaluation (dans Pu et dans Pf)
      - sinon (S est une situation nouvelle) il faut créer un nouveau terme à insérer dans Pu (idem dans Pf)
  - U ayant été développé et supprimé de P, il reste à l'insérer le nœud [U,Val,…,..] dans Q,
  - Appeler récursivement **aetoile** avec les nouveaux ensembles Pf_new, Pu_new et Q_new

Développer ensuite chaque prédicat (**affiche_solution**, **expand**, **loop_successors**), spécifier des tests unitaires et vérifier le bon fonctionnement de chacun avant de tenter le test d'intégration final. Pour rechercher les erreurs et faciliter la mise au point, il peut être utile d'afficher le contenu des AVL (prédicat **put_flat**).

3.3 Analyse expérimentale

- Noter le temps de calcul de A* et l'influence du choix de l'heuristique : quelle taille de séquences optimales (entre 2 et 30 actions) peut-on générer avec chaque heuristique (H1, H2) ? Présenter les résultats sous forme de tableau.
- Quelle longueur de séquence peut-on envisager de résoudre pour le Taquin 4x4 ?

A* trouve-t-il la solution pour la situation initiale suivante ?



Quelle représentation de l'état du Rubik's Cube et quel type d'action proposeriez-vous si vous vouliez appliquer A*?

ANNEXES : quelques rappels utiles en prolog

- **T1 = T2** est une unification ; ne pas confondre avec l'évaluation d'une expression arithmétique notée :
- **T is Expression**, ex : S **is** 2+3.

- La structure **si … alors … sinon ...** en Prolog, ex : mise en ordre de 2 éléments :

  Pour des raisons d'efficacité on évite de faire 2 fois la comparaison si la 1ère clause échoue, en utilisnt un "cut" :

```
ordonner(A,B,R) :- A =< B, !, R =[A, B].
ordonner(A,B,R) :- R =[B, A].
```

  La solution généralement adoptée en prolog (ci-dessous) masque l'usage du cut mais est totalement équivalente :

```
ordonner(A,B,R) :-
( A =< B   ->                          %        ( Condition_Goals ->
   R=[A,B]                             %              Then_Goals
;                                      %        ;
   R=[B,A]                             %              Else_Goals
).                                     %        )
```

  On retient la forme générale du si alors sinon : **(Condition_Goals -> Then_Goals ; Else_Goals)**

- Le prédicat **findall(Term, Generating_Goals, List_Of_Terms)**, par l'exemple :

```
?- List1 = [1,2,3], findall(Y,  (member(X, List1),Y is X+3),  List2) .
   List2 = [4,5,6]
?- List1 = [1,2,3], findall(toto, (member(X, List1)), List2) .
   List2 = [toto,toto,toto]
?- List1 = [1,2,3], findall([X,Y], (member(X, List1), Y is X+3), List2) .
   List2 = [[1, 4], [2, 5], [3, 6]]
```

- **Term1 @< Term2** est vrai si Term1 précède Term2 dans **l'ordre lexicographique**, exemples :

```
?- [[1,2,3], [[a,b],[c,d]]] @< [[2,3,4], [[a,b],[c,d]]]
   Yes
                % Généralement les termes à comparer sont instanciés…
?- X @< Y       % L'ordre entre 2 variables libres
   Yes
?- Y @< X       % … n'a aucun sens
   Yes
```

Pour une définition plus précise voir la documentation : http://eclipseclp.org/doc/bips/kernel/termcomp/AL-2.html