TRP –TP 4 et 5

Application au taquin

Durée : 5 h environ

Description du problème et choix d'une représentation interne

C'est le jeu de taquin tel qu'il a été vu en TD. Il peut être étendu de différentes manières, par exemple en augmentant la taille du plateau de jeu (passage du 3x3 au 4x4...), en augmentant une dimension ou en travaillant avec des cases de tailles variables (jeu de l'âne rouge). On peut également donner un coût différent aux 4 actions de déplacement ou aux cases que l'on déplace. Nous conserverons ici un taquin carré, de taille maximale 5x5. En tout état de cause, la taille de l'espace d'états d'un taquin n x n est considérable, puisque égale à $(n^2)!$ (l'espace de recherche est de taille $(n^2)\ !\ /2$). Certains problèmes de taquin 4x4 demeurent déjà difficiles à résoudre pour les machines actuelles.

Travail à réaliser

Vous devrez écrire un programme représentant ce jeu. Ce programme utilisera la fonction A* implémentée durant les TP précédents.

1. Commencez à réfléchir aux **deux types** : `etat` et `operation` pour représenter respectivement les états de la recherche (configurations possibles du taquin) et les opérations réalisables (ici, les déplacements qui permettent de passer d'une configuration à une autre). On pourra par exemple utiliser une liste de listes pour le type `etat` puis utiliser la fonction `List.nth` pour atteindre le nième élément d'une liste donnée, on pourra également utiliser un tableau. L'état pourra éventuellement fournir directement la position de la case vide.

Une opération est caractérisée par exemple par le déplacement de la case vide. Pour veiller à conserver un caractère générique dans votre programmation, vous devrez écrire un programme *adaptable* à un plateau carré *quelconque*. De plus, on peut considérer que les déplacements ont des coûts qui dépendent par exemple du poids des tuiles à déplacer. Pour cela, vous utiliserez un type `tuile` contenant une description (chaîne de caractères par exemple "A") et aussi une pondération qui pourra être fixée à 1 pour les exemples de base (dans lesquels toutes les tuiles ont le même poids).

2. Définissez **des exemples de constantes** pour chacun de ces types.

On définira notamment l'état `e1` :

A	C	D
H	F	B
G		E

et l'état `b1` suivant :

A	B	C
H		D
G	F	E

Vous pourrez utiliser le signe = pour tester l'égalité entre deux états.

3. Définir la fonction **`deplacer`** de type `etat -> operation -> etat * int` telle que `deplacer e op` renvoie l'état obtenu en effectuant l'opération op depuis l'état e, associé au coût

du déplacement. Elle génère une exception `DeplacementImpossible` si le déplacement est impossible car la case vide est sur l'un des bords du plateau.
Attention : Si vous avez choisi la représentation par tableau, la fonction `deplacer` doit créer un nouveau tableau dans lequel le déplacement est effectué ; elle ne doit pas modifier le tableau fourni en paramètre.

Afin d'utiliser la fonction générique `recherche` pour trouver une solution (séquence de déplacements) de coût optimal, il vous faudra fournir les différentes données et fonctions décrivant le problème ainsi qu'une fonction heuristique. Pour mémoire, pour utiliser la fonction recherche, vous devrez donner les arguments suivants :
- `testEtatBut`, de type `etat -> bool`, caractérisant les buts (renvoie vrai si un état donné est un but, c'est-à-dire si la configuration des piques est la configuration recherchée) ; cette fonction peut être définie à partir d'une fonction `egal` de type `etat -> etat -> bool`, dans ce cas `(egal but)` sera bien du type désiré et cette définition permettra de changer facilement de but.
- `opPoss`, de type `etat->(operation*etat*int) list`, la fonction qui retourne les opérations possibles à partir d'un état, associées aux états d'arrivée et aux coûts de ces opérations.
- `hEtat`, de type `etat -> int`, la fonction heuristique qui donne le coût minimal estimé pour aller d'un état donné à un état but.
- `depart` de type `etat` l'état de départ de la recherche.

Les arguments `testEtatBut` et `opPoss` seront directement déduits des fonctions que vous venez d'écrire. Il reste à définir hEtat.

4. Pour commencer, vous pourrez implémenter l'heuristique simple suivante :
Heuristique W, cases non encore en place : Cette heuristique est minorante. Elle consiste à prendre pour `h(e)` le nombre de cases qui ne sont pas, dans l'état `e`, à la place exigée par l'état-but `b`. Cette heuristique est simple mais elle ne tient pas compte du nombre de coups pour amener une case pleine à sa destination dans l'état-but `b` selon qu'elle en est plus ou moins éloignée. Cette heuristique pour le taquin dépend de l'état but, on pourra donc écrire une première fonction `hW1` de type `etat -> etat -> int`, telle que `hW1 b1 e1` donne le coût estimé pour aller de l'état `e1` à `b1`. Ainsi, pour un `but` donné, la fonction `(hW1 but)` a le type `etat -> int` et retourne, pour un état donné, le coût estimé pour aller de cet état au `but`.

5. Lancez la fonction recherche avec les arguments requis, en prenant **W** comme fonction heuristique. Vous pourrez tester votre programme sur un taquin avec état initial e1 et pour état but but1 et sur les jeux de tests décrits plus loin.

6. Pour visualiser les états et les opérations, vous pourrez créer des fonctions d'affichage

On pourrait par exemple avoir :
```
#afficherEtat e1;;
-------------
| A | C |D |
-------------
| H | F |B |
-------------
| G |   |E |
 -----------
- : unit = ()
```

Amélioration de l'heuristique

Plusieurs autres heuristiques ont été utilisées pour aider à résoudre le problème du taquin en estimant la longueur d'un chemin-solution passant par l'état courant. Nous présentons dans ce qui suit les deux heuristiques les plus employées.

7. Heuristique P, somme des distances « en pâté de maison » : On utilise la distance « `pâté de maison` » (en anglais « `city-block` » ou « `Manhattan` » distance). Cette heuristique est également minorante; elle est plus proche de la réalité que **W**, mais ne tient pas compte de la difficulté à inverser deux cases voisines.
Pour une tuile *t* , un état courant *e* et un état but *b*, soit $L_e(t)$ et $C_e(t)$ (respectivement. $L_b(t)$ et $C_b(t)$) les numéros de la ligne et de la colonne de *t*, dans l'état *e* (resp. dans l'état-but *b*), on définit :

$$d(t,e,b) = L_e(t)\ L_b(t) + C_e(t)\ C_b(t)$$

Alors $hP(e) = \sum_{t \in T} d(t,e,b)$ où *T* est l'ensemble de toutes les tuiles.

8. Heuristique P + 3S : Contrairement à **W** et **P**, cette heuristique tient compte de la difficulté à inverser deux tuiles voisines mais elle ne peut s'utiliser que dans le cas du taquin 3x3. Pour un état `e`, on définit d'abord `S` de la façon suivante :

- On choisit un sens de parcours des cases non centrales autour de la case centrale. On ne considère que les cases non centrales contenant des tuiles. A chacune d'entre elles, on attribue :
  - un poids de `0` si la tuile correspondante est aussi non centrale dans l'état-but `b` et si dans les deux états (`e` et `b`) la tuile suivante (dans le sens de parcours choisi) est la même.
  - `2` sinon.
- A la case centrale, si elle est pleine dans l'état `e`, on attribue :
  - `0` si elle contient déjà la bonne tuile par rapport à `b` ;
  - `1` sinon.

Alors S(e)= *somme des poids des cases pleines dans l'état e* et hP3S(e) = hP(e) + 3×S(e).

Exemples de calculs des heuristiques : étant donnés l'état but *b2* et les états *e2* et *e2'* :

e2

A	B	C
D	E	F
	H	G

e2'

A	B	C
D	E	F
	G	H

b2

A	B	C
D	E	F
G	H	

on obtient : $hW(e2)=1$, $hP(e2)=2$, $S(e2)=6$ et $hW(e2')=2$, $hP(e2')=2$, $S(e2')=4$

Jeux de tests

Problèmes avec un coût uniforme

On considère que tous les déplacements ont un coût de 1. Testez votre implémentation, avec les différentes heuristiques, sur les six problèmes suivants. Pour chacun d'entre eux, vous donnerez : le nombre de nœuds créés, le nombre de nœuds développés, le chemin-solution optimal et son coût.

e3

A	C	D
H	F	B
G		E

e3'

B	H	C
A	F	D
G		E

b3

A	B	C
H		D
G	F	E

e4

H	G	F
E	D	C
B	A	

e4'

C	A	D
	B	E
H	G	F

b4

	A	B
C	D	E
F	G	H

e5

A	E	B	C
D	F		G
H	N	J	K
L	I	M	O

e5'

O	K	H	L
N	J	I	M
B	F	A	D
C	G	E	

b5

	A	B	C
D	E	F	G
H	I	J	K
L	M	N	O

Problèmes avec un coût non uniforme

9. Les déplacements ont maintenant des coûts qui dépendent des cases que remplace la case vide. On peut ainsi considérer que chaque case correspond à un objet qu'il est plus ou moins coûteux de déplacer (objet plus ou moins lourd nécessitant plus ou moins d'énergie pour son déplacement par exemple). On recherche un chemin-solution de coût minimal. Testez votre programme, avec les différentes heuristiques, sur les six problèmes précédents, en prenant comme poids pour chaque tuile le rang de la lettre dans l'ordre alphabétique (A=1, B=2, ...). Pour chacun d'entre eux, vous donnerez : le nombre de nœuds créés, le nombre de nœuds développés, le chemin-solution optimal et son coût.