

TRAVAUX PRATIQUES XIII

Algorithme A^* : jeu du Taquin

L'objectif de ce TP est d'utiliser l'algorithme A^* pour trouver une solution optimale au jeu du taquin (en nombre minimal de coups).

A Jeu du taquin

A.1 Présentation du jeu

Le jeu de taquin est constitué d'une grille $n \times n$ dans laquelle sont disposés les entiers de 0 à $n^2 - 2$, une case étant laissée libre.

Exemple : Voici un état initial possible pour $n = 4$ (qui est la version classique) :

	0	1	2	3
0	2	3	1	6
1	14	5	8	4
2		12	7	9
3	10	13	11	0

En pratique, le jeu est constitué de tuiles que l'on peut faire glisser sur le carré. On obtient un nouvel état du jeu en déplaçant dans la case libre le contenu de la case située au-dessus, à gauche, en dessous ou à droite, au choix. Si on déplace par exemple le contenu de la case située à droite de la case libre, c'est-à-dire 12, on obtient le nouvel état suivant :

2	3	1	6
14	5	8	4
12		7	9
10	13	11	0

Le but du jeu de taquin est de parvenir à l'état final dans lequel toutes les tuiles sont disposées dans l'ordre croissant. Par exemple, avec $n = 4$:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

Dans ce sujet, on s'intéresse à la résolution optimale du jeu du taquin, c'est-à-dire à déterminer une suite de déplacements légaux **de longueur minimale** permettant de passer d'une configuration initiale donnée à la configuration finale.

Dans l'exemple ci-dessus, la solution optimale est de longueur 50 (à partir de l'état initial, ou 49 à partir du deuxième état représenté).

A.2 Structures de données OCaml

En OCaml, une position sera représentée par le type suivant :

OCaml

```

1 type etat = {
2   grille : int array array;
3   mutable i : int; (* abscisse de la case libre *)
4   mutable j : int; (* ordonnée de la case libre *)
5   mutable h : int; (* valeur pour l'heuristique *)
6 }

```

Description :

- On suppose qu'une constante globale n a été définie.
 - `i` et `j` indiquent les coordonnées de la case libre.
 - `grille` est une matrice de dimensions $n \times n$ codant la grille. Le contenu de la case libre est arbitraire : si `s` est de type `etat`, alors `s.grille.(s.i).(s.j)` peut avoir n'importe quelle valeur.
 - `h` sera une heuristique estimant la distance de l'état actuel à l'état final, que nous définirons plus loin.
1. Écrire une fonction `affiche_etat : etat -> unit` qui affiche dans le terminal la grille du taquin. On ne demande pas d'effort particulier sur l'affichage. Par exemple, on pourrait obtenir l'affichage suivant :

Terminal

```

1 Grille :
2  0  1  2  3
3 12     5  6
4  8  4 10 11
5 13 14  7  9

```

Astuce : vous pouvez utiliser `Printf.printf "%2d"` pour afficher votre entier en 2 chiffres, peu importe sa longueur, et vous assurer que les nombres soient alignés.



B Graphe des états du jeu du taquin

Une configuration du taquin se code naturellement comme une permutation de $\llbracket 0, \dots, 15 \rrbracket$ (où le 15 correspond à la case vide). On peut alors définir le graphe (non orienté) G des états du taquin comme suit¹ :

- Les sommets sont les éléments de \mathcal{S}_{16} .
- Il y a une arête entre s et s' si et seulement si on peut passer de s à s' (en une étape) par l'un des quatre déplacements décrits plus haut.

On admet que ce graphe possède **exactement deux composantes connexes**, contenant chacune la moitié des sommets².

2. Quel est le nombre de sommets de ce graphe ? le nombre approximatif d'arêtes ? Est-il raisonnable de le stocker explicitement en mémoire ?

On code un déplacement par le type suivant, où `Haut` correspond à un déplacement  de la **case libre**  vers le haut, par exemple :

OCaml

```

1 type direction = Haut | Bas | Gauche | Droite | Rien
2
3 let delta = function
4   | Haut -> (-1, 0)
5   | Bas  -> (1, 0)
6   | Gauche -> (0, -1)
7   | Droite -> (0, 1)
8   | Rien  -> assert false

```

La valeur `Rien` ne sera utilisée qu'en fin de sujet.

3. Écrire une fonction `mouvements_possibles : etat -> direction list` qui renvoie la liste des directions de déplacement légales à partir d'un certain état.

1. Cette définition est cohérente avec celle donnée en cours dans le chapitre de théorie des jeux, et devrait vous servir de rappel/révision.
 2. Le prouver est un bon exercice... de mathématiques (qui implique la parité de la signature des permutations).

Pour orienter la recherche, on définit une heuristique h comme suit qui associe à chaque état du taquin un entier positif ou nul.

Pour e un état et $v \in \llbracket 0, n^2 - 2 \rrbracket$, on note e_v^i la ligne de l'entier v dans e et e_v^j sa colonne. On pose alors :

$$h(e) = \sum_{v=0}^{n^2-2} |e_v^i - \lfloor v/n \rfloor| + |e_v^j - (v \bmod n)|.$$

- Montrer que l'heuristique h est admissible et cohérente.

Indication : Cette question peut sembler difficile de prime abord mais elle est en fait très simple une fois qu'on a compris ce que représentait $h(e)$.

- Écrire une fonction `calcule_h : etat -> unit` qui prend en entrée un état, dans lequel le champ `h` a une valeur quelconque, et donne à ce champ la bonne valeur.

Aide : On pourra, pour cette question et la suivante, utiliser la fonction `distance` fournie.

- Écrire une fonction `delta_h : etat -> direction -> int` qui prend en entrée un état e et une direction d et renvoie la différence $h(e') - h(e)$, où e' est l'état que l'on atteint à partir de e en effectuant le déplacement d . On ne fera que les calculs nécessaires (on évitera donc de recalculer toute la somme définissant h).
- Écrire une fonction `applique : etat -> direction -> unit` qui modifie un état en lui appliquant un déplacement, que l'on supposera légal.

Remarque :

- On choisit de modifier l'état plutôt que d'en calculer un nouveau car cela nous sera utile en fin de sujet. Cependant, il sera souvent pratique de disposer d'une copie indépendante.
- Écrire une fonction `copie : etat -> etat` qui prend un état et en renvoie une copie. On pourra utiliser la fonction `Array.copy`, mais attention : `grille` est un tableau de tableaux...

C Utilisation de A^*

C.1 Structures fournies

Deux modules OCaml sont fournis :

- `Heap` (fichiers `heap.ml` et son interface `heap.mli`) pour une **file de priorité min** permettant l'opération `DECREASEPRIORITY`;
- `Vector` (fichiers `vector.ml` et son interface `vector.mli`) pour des tableaux dynamiques. Ce module est surtout utilisé de manière interne par le module `Heap`, vous n'aurez à vous en servir que tout à la fin du sujet.

Dans les deux cas, une rapide lecture du fichier `.mli` devrait vous permettre d'utiliser le module sans difficulté. Il n'est pas vraiment nécessaire d'aller lire les `.ml`.

D'autre part, vous aurez besoin dans le sujet d'utiliser le module `Hashtbl` qui fournit des tables de hachage.

On en rappelle quelques fonctions utiles en annexe. N'hésitez pas à aller lire la documentation !

C.2 Résolution du taquin

Rappelons que l'algorithme A^* est une fonction de parcours de graphe (dans un certain ordre). Un pseudo-code a été donné en cours, et je vous invite à vous y référer pour comprendre ce qu'on fait. Le but va être de parcourir le graphe des états du jeu du taquin à la recherche d'un chemin de longueur minimale. Dans le graphe des états du taquin, toutes les arêtes sont de poids 1, car on compte juste le nombre de déplacement.

Tout d'abord, comme le graphe n'est pas stocké explicitement, il nous faut un moyen d'accéder aux voisins d'un sommet.

- Écrire une fonction `successeurs : etat -> etat list` qui prend en entrée un état et renvoie la liste de ses successeurs dans le graphe (ou de ses voisins, d'ailleurs, le graphe n'étant pas orienté).

L'algorithme A^* va donc pouvoir explorer le graphe jusqu'au sommet destination et construire l'arbre du parcours effectué (structure appelée `parents` dans le cours). Il faudra alors à partir de cet arbre reconstruire le chemin trouvé en remontant les pères jusqu'à la source (ligne 13 du pseudo-code). C'est ce que nous allons coder maintenant.

Dans ce TP, on utilise la représentation suivante d'un arbre dans un tableau `t` (rappel) :

- $t[i] = i$ si et seulement si i est la racine de l'arbre ;

- $t[i] = j$ si j est le père de i .

On peut naturellement étendre cette définition à un `('a, 'a) Hashtbl.t` au lieu d'un tableau.

10. Écrire une fonction `reconstruit : ('a, 'a) Hashtbl.t -> 'a -> 'a list` qui prend en entrée un dictionnaire codant un arbre et un nœud x de l'arbre, et renvoie un chemin de la racine à x , sous la forme d'une liste de nœuds.
11. Écrire une fonction `astar : etat -> etat list` prenant en entrée un état initial et calculant un chemin de longueur minimale vers l'état final à l'aide de l'algorithme A^* . Cette fonction lèvera l'exception `Aucun_chemin` si aucun chemin n'existe.
Indication : Comme indiqué, on utilise un dictionnaire `parents` au lieu d'un tableau : il faudra faire de même pour les distances.
12. Tester cette fonction sur les différents exemples fournis dans le squelette (\triangle tous ne seront pas forcément traitables en un temps raisonnable!) et compter le nombre d'états explorés dans chaque cas.
Vous pouvez afficher les grilles successives de votre chemin grâce à `affiche_etat` et vérifier que la grille finale est bien la solution, et que l'on passe d'une grille à l'autre par des coups légaux.

D Pour aller plus loin : Algorithme IDA^*

Dans le cas de l'exploration d'un graphe infini, ou en tout cas suffisamment grand pour ne pas pouvoir être stocké en mémoire, on peut se retrouver limité par la mémoire plus que par le temps. En effet, explorer des centaines de millions de nœuds n'est pas forcément problématique sur une machine moderne, mais les stocker, et faire des tests d'appartenance à chaque étape, peut vite s'avérer prohibitif. On peut dans ce cas utiliser l'algorithme dit IDA^* , qui est un hybride entre le **parcours en profondeur itéré** et l'algorithme A^* .

D.1 Parcours en profondeur itéré

On considère l'algorithme suivant :

Pseudo-code

```

1  Entrées : G un graphe, s un sommet de G, final un sommet cible,
2            m un entier bornant la profondeur du parcours,
3            p la profondeur actuel du parcours (sera initialement 0)
4  Sortie : Vrai si le sommet cible \ttt{final} est atteignable depuis s
5            par un parcours de profondeur bornée par m dans G
6
7  DFS-MAX(G, s, final, m, p) :
8      Si p > m Alors :
9          Renvoyer FAUX
10     Si s = final Alors :
11         Renvoyer VRAI
12     Pour v successeur de s Faire :
13         Si DFS(G, v, final, m, p + 1) Alors :
14             Renvoyer VRAI
15     Renvoyer FAUX

```

Dans cet algorithme, s représente l'état actuel, p la profondeur actuelle (c'est-à-dire la longueur du chemin suivi de l'état initial au nœud actuel, longueur qui n'est pas nécessairement minimale) et m la profondeur maximale autorisée.

13. Montrer que $\text{DFS}(G, \text{init}, \text{final}, m, 0)$ renvoie VRAI si et seulement si le sommet final est à une distance inférieure ou égale à m de init .

Le parcours en profondeur itéré IDS consiste à effectuer des appels successifs à $\text{DFS}(0, \text{init}, 0)$, $\text{DFS}(1, \text{init}, 0)$, et ainsi de suite jusqu'à trouver un m pour lequel on obtient une réponse positive : ce m est alors la distance de init au sommet final.

14. Déterminer la complexité en temps et en espace d'un parcours en profondeur itéré depuis un sommet initial situé à distance n du sommet final dans les deux cas suivants :
 - le graphe contient exactement 1 sommet à distance k de init pour tout k ;
 - le graphe contient exactement 2^k sommets à distance k de init pour tout k .

Quel peut être l'intérêt d'effectuer un parcours en profondeur itéré plutôt qu'un parcours en largeur pour déterminer un plus court chemin ?

D.2 Algorithme IDA^*

L'algorithme IDA^* est obtenu en ajoutant à l'algorithme IDS une heuristique h admissible, et en effectuant les modifications suivantes :

- la borne ne concerne plus la profondeur p mais le coût estimé $h(e) + p$;
- si un parcours avec une borne de m a échoué, le parcours suivant se fait avec comme borne la plus petite valeur de $h(e) + p$ qui a dépassé m lors du parcours.

On va à nouveau parcourir plusieurs fois des fragments d'arbres, mais cette fois la croissance de ces fragments sera *orientée vers l'état final* (à condition que l'heuristique soit bonne).

Pseudo-code

```

1  IDA*(G, s0, h, final) :
2  m ← h(s0)
3  minimum ← ∞
4  fonction DFS*(m, s, p) :
5      c ← p + h(s)
6      Si c > m Alors :
7          minimum ← min(c, minimum)
8          Renvoyer FAUX
9      Si s = final Alors :
10         Renvoyer VRAI
11     Pour v successeur de s Faire :
12         Si DFS*(m, v, p + 1) Alors :
13             Renvoyer VRAI
14     renvoyer FAUX
15 Tantque m ≠ ∞ Faire :
16     minimum ← ∞
17     Si DFS*(m, s0, p) Alors :
18         Renvoyer VRAI
19     m ← minimum
20 Renvoyer FAUX

```

- Écrire une fonction `idastar_length : etat -> int option` qui calcule la longueur minimale d'un chemin du sommet fourni jusqu'au sommet final. On n'utilisera qu'un seul état que l'on modifiera au fur et à mesure du parcours (pas d'appels à la fonction `successeurs`, donc). La fonction renverra `None` si l'état est inaccessible (et qu'elle le détecte).
- Vérifier que la fonction traite correctement, et en un temps raisonnable, les exemples `ten`, `twenty` et `thirty`.
- Montrer que si l'heuristique h est admissible, la fonction `idastar_length` renvoie toujours la distance du sommet fourni au sommet final, à condition qu'un chemin existe.
- Apporter les modifications suivantes à cette fonction :
 - On souhaite obtenir le chemin gagnant, sous la forme d'un `direction Vector.t`.
 - On évitera de revenir immédiatement en arrière (on n'essaiera pas le coup `Gauche` si le dernier coup du chemin actuel est `Droite`). La présence de `Rien` dans le type `direction` peut ici s'avérer utile.

OCaml

```

1 val idastar : etat -> direction Vector.t option

```

Terminal

```

1 $print_idastar cinquante;;
2 Longueur 50
3 Down Left Left Up Right Right Up Left Down Left Left Up Right
4 Right Right Down Left Left Left Down Right Right Up Left Left
5 Up Right Right Up Left Left Down Right Right Right Up Left Left
6 Down Right Right Down Down Left Up Left Left Up Right Down$

```

Annexe : module Hashtbl

(`'a` est le type des clés et `'b` celui des valeurs).

- `Hashtbl.create : int -> ('a, 'b) Hashtbl.t` crée une table vide. L'entier fourni donne la capacité initiale mais n'a que peu d'importance (la table sera redimensionnée au besoin).
- `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` permet de tester si une clé est présente dans la table.
- `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` ajoute une association à la table.
- `Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` modifie une association existante, ou crée l'association si elle n'existait pas. On peut donc l'utiliser systématiquement à la place de `Hashtbl.add`.
- `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` renvoie la valeur associée à une clé, ou lève l'exception `Not_found` si la valeur n'est pas dans la table.
- `Hashtbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option` fait la même chose, mais utilise une option au lieu d'une exception.