TP 10 : L'algorithme A*

Vous utiliserez dans ce TP les modules suivant :

- `Heap` (fichier `heap.ml`) : implémentant des files de priorité minimum.
- `Vector` (fichier `vector.ml`) : implémentant des tableaux dynamiques. Ce module est utilisé par le module `Heap` et vous l'utiliserez dans la fin du sujet.
- `Hashtbl` (bibliothèque standard) : implémentant des tableaux associatifs (vous pouvez facilement trouver sa documentation en ligne).

Le jeu du taquin se joue sur une grille de taille $n \times n$ (classiquement, $n = 4$ et ce sera le cas dans tout le TP) contenant les entiers de $[\![0, n^2 - 2]\!]$. Voici un exemple de configuration initiale :

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

Les règles du jeu autorisent à déplacer dans la case vide le contenu d'une des cases voisines (en haut, en bas, à droite ou à gauche) de cette case vide. Si on déplace par exemple le contenu de la case située à droite de la case libre, c'est-à-dire 12, on obtient la configuration fille suivante :

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

Le but du jeu du taquin est de parvenir à la configuration finale suivante :

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

L'objectif de ce TP est de déterminer une suite de déplacements légaux **de longueur minimale** permettant de passer d'une configuration initiale donnée à la configuration finale.

En OCaml, une configuration sera représentée par le type d'enregistrement suivant :

```
type state = {
  grid : int array array;
  mutable i : int;
  mutable j : int;
  mutable h : int}
```

- On suppose qu'une constance globale `n` a été définie.
- `i` et `j` indiquent les coordonnées de la case libre.
- `grid` est une matrice $n \times n$ représentant la grille. La valeur de l'élément correspondant à la case libre est quelconque (c'est une valeur qu'on ignore en pratique).
- `h` permettra de stocker la valeur associée à cette configuration par l'heuristique.

Exercice 1 : Graphe du taquin

Une configuration du taquin s'encode naturellement comme une permutation de $[\![0, 15]\!]$ avec 15 représentant la case vide. Le graphe (non orienté) du taquin est défini comme suit :

- Ses sommets sont étiquetés par les éléments de $\mathfrak{S}_{16}$,
- Il y a une arête de s à t si et seulement si on peut passer de s à t en une étape par l'un des au plus quatre déplacements possibles selon les règles du jeu du taquin.

1) Quel est l'ordre de ce graphe ? Sa taille approximative ? Peut-on raisonnablement le stocker explicitement en mémoire ?

Trouver une suite de déplacements minimale pour résoudre un taquin revient à calculer un plus court chemin dans le graphe du taquin entre la configuration initiale et la configuration finale. Au vu de la taille de ce graphe, on souhaite effectuer cette recherche de manière à l'aide de l'algorithme A*.

On se dote dans cette partie de quelques fonctions utilitaires et d'une heuristique permettant de guider A*.

On code un déplacement à l'aide du type suivant :

```
type direction = U | D | L | R | N
```

Le constructeur `U` correspond à un déplacement de la case libre vers le haut (up), `D` à un déplacement de la case libre vers le bas, etc.

Le constructeur `No_move` sera utilisé vers la fin du sujet pour représenter une absence de déplacement.

On pourra utiliser lorsque cela est nécessaire la fonction suivante :

```
let delta = function
  | U -> (-1, 0)
  | D -> (1, 0)
  | L -> (0, -1)
  | R -> (0, 1)
  | No_move -> assert false
```

2) Écrire `possible_moves : state -> direction list` une fonction qui renvoie la liste des directions de déplacement légales à partir d'une configuration donnée.

Pour chaque configuration c du jeu du taquin, et tout $v \in [\![0, n^2-2]\!]$, on note c_v^i la ligne de l'entier v dans la configuration c et c_v^j sa colonne. L'heuristique h que l'on choisit pour orienter la recherche associe à la configuration c l'entier suivant :

$$h(c) = \sum_{v=0}^{n^2-2} \left(\left| c_v^i - \lfloor v/n \rfloor \right| + \left| c_v^j - (v \mod n) \right| \right)$$

3) (à faire chez-soi) Montrer que l'heuristique est admissible et monotone.

4) Expliquer ce que calcule cette fonction :

```
let distance (i:int) (j:int) (value:int) :int =
  let i_target = value / n in
  let j_target = value mod n in
  abs (i - i_target) + abs (j - j_target)
```

Recopier cette fonction dans votre programme en écrivant ses spécifications.

5) Écrire une fonction `compute_h : state -> unit` qui modifie le champ de la configuration c prise en entrée de sorte qu'il vaille $h(c)$.

6) Écrire une fonction `delta_h : state -> direction -> int` prenant en entrée une configuration c, une direction d et renvoyant la différence $h(c') - h(c)$ où c' est la configuration fille de c lorsqu'on fait le déplacement d. On ne fera que les calculs nécessaires (autrement dit, on ne recalculera pas toute la somme définissant h depuis le début).

7) Écrire une fonction `apply : state -> direction -> unit` qui modifie une configuration en lui appliquant le déplacement donné par la direction en entrée (qu'on supposera légal sans le vérifier).

Dans cette dernière fonction, on a choisi de modifier la configuration plutôt que d'en recréer une nouvelle suite à un coup. Il sera néanmoins parfois utile de disposer d'une copie indépendante d'une configuration :

8) Écrire une fonction `copy : state -> state` prenant une configuration en entrée et en renvoyant une copie. On pourra utiliser `Array.copy` en prenant garde au fait que `grid` est un tableau de tableaux.

Exercice 2 : Implémentation de A*

9) Écrire une fonction `sucessors : state -> state list` prenant en entrée une configuration et renvoyant la liste des configurations que l'on peut atteindre en un seul déplacement.

Attention aux partages de mémoire.

Comme pour la structure de donner unir-trouver, on peut implémenter des arbres par un table T de la façon suivante :

- $T[i] = i$ si et seulement si i est la racine de l'arbre.
- $T[i] = j \neq i$ si j est le père de i dans l'arbre.

Ce principe s'étend des tableaux aux dictionnaires : les clés et les valeurs sont alors des sommets du graphe et la présence du couple (s, p) dans le dictionnaire signale que le père de s dans l'arbre est p.

10) Écrire une fonction `reconstruct : ('a, 'a) Hashtbl.t -> 'a -> 'a list` qui prend en entrée un dictionnaire encodant un arbre et un noeud x de cet arbre et renvoie le chemin de la racine de l'arbre à x sous forme d'une liste de noeuds.

11) Écrire une fonction `astar : state -> state` 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 générera une erreur si aucun chemin n'existe. On utilisera des dictionnaires pour stocker les distances depuis l'origine et les prédécesseurs sur un plus court chemin plutôt que des tableaux.

12) Utiliser le contenu du fichier `tests.ml` pour tester vos fonctions.

Vérifier dans chaque cas que la longueur du chemin trouvé est bien la bonne.

Exercice 3 : L'algorithme IDA*

Le facteur limitant dans ce problème est principalement la mémoire au vu de la taille du graphe à explorer. Afin d'améliorer les performances de la partie précédente, on s'intéresse à l'algorithme IDA* qui est un hybride entre l'algorithme A* et le parcours en profondeur itéré, IDS (Iterative Depth Search).

Ce dernier repose sur l'algorithme de parcours en profondeur limité à une profondeur maximale suivant :

```
DFS(m, e, p):
si p > m :
    renvoyer faux
si e est l'état final :
    renvoyer vrai
pour x successeur de e :
    si DFS(m, x, p + 1) :
        renvoyer vrai
renvoyer faux
```

Dans cet algorithme, e représente le sommet actuel exploré, p la profondeur actuelle (la longueur du chemin suivi depuis le sommet initial vers celui actuel) et m la profondeur maximale d'exploration autorisée. On renvoie vrai s'il existe un chemin du sommet e jusqu'au sommet final et faux sinon.

13) Montrer que DFS(m, s, 0) renvoie vrai si et seulement si le sommet final est à une distance inférieure à m de la configuration initiale s.

Le parcours en profondeur itéré IDS consiste à effectuer des appels successifs à DFS(0, s, 0), DFS(1, s, 0)... jusqu'à trouver un m pour lequel on obtient une réponse positive, dans l'objectif de déterminer la distance séparant un sommet source s du sommet but b.

14) Déterminer la complexité en temps et en espace d'un parcours en profondeur itéré depuis un sommet initial s situé à distance n du sommet final dans les deux cas suivants :

a) Le graphe contient exactement un sommet à distance k de s pour tout k.

b) Le graphe contient exactement 2^k sommets à distance k de s 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 ?

L'algorithme IDA* est obtenu à partir de IDS en lui ajoutant une heuristique admissible h 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 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.

Si l'heuristique est bonne, on va parcourir des fragments d'arbres qui vont croître dans une direction orientée vers le sommet final. Les pseudo-codes pour la variante d'IDS qu'on utilise ici (qu'on note DFS*) puis IDA* suivent ; la variable minimum définie dans IDA* est utilisée et mise à jour dans DFS* :

```
DFS*(m, e, p):
c ← p + h(e)
si c > m :
    minimumn ← min(c, minimum)
    renvoyer faux
si e est l'état final :
    renvoyer vrai
pour x successeur de e :
    si DFS*(m, x, p + 1) :
        renvoyer vrai
renvoyer faux
```

```
IDA*(s):
m ← h(s)
tant que m ≠ ∞ :
    minimum ← ∞
    si DFS*(m, s, 0) :
        renvoyer vrai
    m ← minimum
renvoyer faux
```

15) Écrire une fonction `idastar_length : state-> int option` qui calcule la longueur minimale entre la configuration initiale donnée en entrée et la configuration finale. On ne fera aucun appel à `successeurs` ; à la place on utilisera une configuration qu'on mutera au fur et à mesure du parcours. Si la fonction détecte que la configuration finale est inaccessible, elle renverra `None`.

16) Montrer que si l'heuristique h est admissible, la fonction `idastar_length` renvoie effectivement la longueur d'un plus court chemin de la source au but s'il existe.

17) Modifier `idastar_length` en une fonction `idastar : state -> direction Vector.t option` qui renvoie un chemin minimal de coups sous la forme d'un `direction Vector.t` pour résoudre le taquin étant donnée une configuration initiale. Elle renverra `None` si un tel chemin n'existe pas.

On évitera dans `idastar` de revenir immédiatement sur ses pas (autrement dit, on ne testera pas le coup `L` si le dernier coup sur le chemin actuel était `R`). La direction `No_move` peut s'avérer utile.

18) Comparer les performances de `astar` et `idastar` sur le taquin `fifty`.