TP3 : Taquin

07 et 14 octobre

Ce TP est inspiré d'un TP proposé par Jean Baptiste Bianqui.

1 STRUCTURES FOURNIES

Deux modules sont fournis :

— `Heap` (fichiers `heap.ml` et `heap.mli`) pour une file de priorité min permettant l'opération DECREASEPRIORITY ;

— `Vector` (fichiers `vector.ml` et `vector.mli`) pour des tableaux dynamiques. Ce module est surtout utilisé de manière interne par le module `Heap`.

D'autre part, vous aurez besoin dans le sujet d'utiliser le module `Hashtbl` qui fournit des tables de hachage. On rappelle ci-dessous quelques fonctions utiles (`'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.

▷ **Question 1.** Un tas est représenté de manière usuelle dans un tableau `keys` correspondant au parcours en largeur de l'arbre binaire associé. La structure dispose de plus d'un tableau `priority` qui stocke dans sa case `i` la priorité de l'élément `keys.(i)`. Afin de garder une trace de l'emplacement des clés dans le tableau `keys`, on maintient à jour une table de hachage `mapping` telle que les clés sont les clés du tas et les valeurs associées sont les indices du tableau `keys` dans lesquelles elles se trouvent. Les tableaux utilisés sont implémentés à l'aide du module `Vector` qui fournit des tableaux dynamiques ce qui permet de redimensionner facilement la taille du tas si besoin.

Compléter le fichier `heap.ml` afin de définir les fonctions `left`, `right` et `parent` qui fournissent l'indice de fils gauche, fils droit et parent d'une clé située dans la case `i` du tableau `keys`.

Compléter la fonction récursive `sift_up` qui permet de reformer la structure de tas après insertion d'un nouvel élément dans l'arbre. Cette fonction sert d'outil à la fonction fournie d'insertion dans un tas. ◁

▷ **Question 2.** Vous trouverez dans le squelette une fonction `dijkstra` qui calcule les plus courts chemins depuis `s` dans un graphe orienté pondéré à coefficients positifs. Adapter cette fonction pour obtenir une fonction `astar` qui calcule le plus court chemin de `s` à `t` dans un graphe orienté pondéré à coefficients positifs en utilisant une heuristique `h` passée en argument. ◁

2 Jeu du taquin

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. 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

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 suivant :

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).

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

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

— On suppose qu'une constante globale n a été définie.
— `i` et `j` indiquent les coordonnées de la case libre.
— `grid` est une matrice $n \times n$ codant la grille. Le contenu de la case libre est arbitraire : si `s` est de type `state`, alors `s.grid.(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.

3 Graphe du taquin

Une configuration du taquin se code naturellement comme une permutation de $[0 \dots 15]$ (où le 15 correspond à la case vide). On peut alors définir le graphe (non orienté) G du taquin comme suit :
— 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 admettra que ce graphe possède exactement deux composantes connexes, contenant chacune la moitié des sommets.

▷ **Question 3.** 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 U, par exemple, correspond à un déplacement de la **case libre** vers le haut :

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

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

▷ **Question 4.** Écrire une fonction possible_moves qui renvoie la liste des directions de déplacement légales à partir d'un certain état.

```
val possible_moves : state -> direction list
```

◁

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 $v \in [0, \dots, n^2 - 2]$, on note e_v^i la ligne de l'entier v dans e et e_v^j sa colonne. On pose alors :

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

▷ **Question 5.** Montrer que l'heuristique h est admissible et monotone.

Remarque 1. *Cette question peut sembler difficile mais elle est en fait très simple*, une fois qu'on a compris ce que représentait $h(e)$.

◁

▷ **Question 6.** Écrire une fonction compute_h qui prend en entrée un état, dans lequel le champ h a une valeur quelconque, et donne à ce champ la bonne valeur.

Remarque 2. *On pourra, pour cette question et la suivante, utiliser la fonction* distance *fournie.*

```
val compute_h : state -> unit
```

◁

▷ **Question 7.** Écrire une fonction delta_h 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).

```
val delta_h : state -> direction -> int
```

◁

▷ **Question 8.** Écrire une fonction apply qui modifie un état en lui appliquant un déplacement, que l'on supposera légal.

```
val apply : state -> direction -> unit
```

◁

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.

▷ **Question 9.** Écrire une fonction copy qui prend un état et en renvoie une copie. On pourra utiliser la fonction Array.copy, mais attention : grid est un tableau de tableaux...

```
val copy : state -> state
```

◁

4 Utilisation de $A^\star$

▷ **Question 10.** Écrire une fonction `successors` 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).

```
val successors : state -> state list
```

◁

▷ **Question 11.** Nous avons souvent codé des arbres dans des tableaux de la manière suivante :
— $t[i] = i$ 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`. Écrire une fonction `reconstruct` 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.

```
val reconstruct : ('a, 'a) Hashtbl.t -> 'a -> 'a list
```

◁

▷ **Question 12.** Écrire une fonction `astar` prenant en entrée un état initial et calculant un chemin de longueur minimale vers l'état final à l'aide de l'algorithme $A^\star$. Cette fonction lèvera l'exception `No_path` si aucun chemin n'existe.

```
val astar : state -> state list
```

Remarque 3. *Comme indiqué, on utilise un dictionnaire* `parents` *au lieu d'un tableau : il faudra faire de même pour les distances.*

◁

▷ **Question 13.** Tester cette fonction sur les différents exemples fournis (tous ne seront pas forcément traitables en un temps raisonnable !). ◁

5 Algorithme $IDA^\star$

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 $IDA^\star$, qui est un hybride entre le *parcours en profondeur itéré* et l'algorithme $A^\star$.

5.1 Parcours en profondeur itéré

On considère l'algorithme suivant :

Algorithme 1 Parcours en profondeur limité par une profondeur maximale m.

```
fonction DFS(m, e, p)
    si p > m alors
        renvoyer FAUX
    fin si
    si e est l'état final alors
        renvoyer VRAI
    fin si
    pour x successeur de e faire
        si DFS(m, x, p + 1) alors
            renvoyer VRAI
        fin si
    fin pour
    renvoyer FAUX
fin fonction
```

Dans cet algorithme, e 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.

▷ **Question 14.** Montrer que DFS$(m, init, 0)$ 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 DFS$(0, init, 0)$, DFS$(1, init, 0)$, 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.

▷ **Question 15.** 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 ? ◁

5.2 Algorithme $IDA^\star$

L'algorithme $IDA^\star$ 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).

Algorithme 2 Pseudo-code de l'algorithme IDA*.

```
fonction IDA*( )
    m ← h(e_0)
    minimum ← ∞
    fonction DFS*(m, e, p)
        c ← p + h(e)
        si c > m alors
            minimum ← min(c, minimum)
            renvoyer FAUX
        fin si
        si e est l'état final alors
            renvoyer VRAI
        fin si
        pour x successeur de e faire
            si DFS*(m, x, p + 1) alors
                renvoyer VRAI
            fin si
        fin pour
        renvoyer FAUX
    fin fonction
    tant que m ≠ ∞ faire
        minimum ← ∞
        si DFS*(m, e_0, 0) alors
            renvoyer VRAI
        fin si
        m ← minimum
    fin tant que
    renvoyer FAUX
fin fonction
```

▷ **Question 16.** Le squelette fournit une fonction `idastar_length` qui calcule la longueur minimale d'un chemin du sommet fourni jusqu'au sommet final. On utilise qu'un seul état que l'on mutera au fur et à mesure du parcours. La fonction renvoie `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`.

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 `L` si le dernier coup du chemin actuel est `R`). La présence de `No_move` dans le type `direction` peut ici s'avérer utile.

Vous devriez maintenant avoir une solution pour l'exemple `fifty` dont la solution est ci-dessous :

Exemple test :

```
print_idastar fifty;;
Length 50
Down Left Left Up Right Right Up Left Down Left Left Up Right
Right Right Down Left Left Left Down Right Right Up Left Left
Up Right Right Up Left Left Down Right Right Right Up Left Left
Down Right Right Down Down Left Up Left Left Up Right Down
```

◁

Remarque 4. *On remarque un point important ici, la fonction IDA* a une complexité temporelle plus élevée que A* puisqu'elle va visiter beaucoup plus de sommets mais le gain en complexité spatiale la rend beaucoup plus efficace en pratique. Si vous testez l'exemple* `sixty_four`*, IDA* prendra quelques minutes alors que A* échouera faute d'espace mémoire suffisant.*