ÉCOLE POLYTECHNIQUE — ÉCOLES NORMALES SUPÉRIEURES

CONCOURS D'ADMISSION 2017 **FILIÈRES MP SPECIALITÉ INFO**

COMPOSITION D'INFORMATIQUE – A – (XULCR)

(Durée : 4 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation sera **obligatoirement** `Caml`.

*
* *

Jeux à un joueur et solutions optimales

Nous nous intéressons ici à des jeux à un joueur où une configuration initiale est donnée et où le joueur effectue une série de déplacements pour parvenir à une configuration gagnante. Des casse-tête tels que le *Rubik's Cube*, le solitaire, l'âne rouge ou encore le taquin entrent dans cette catégorie. L'objectif de ce problème est d'étudier différents algorithmes pour trouver des solutions à de tels jeux qui minimisent le nombre de déplacements effectués.

La partie I introduit la notion de jeu à un joueur et un premier algorithme pour trouver une solution optimale. Les parties II et III proposent d'autres algorithmes, plus efficaces. La partie IV a pour objectif de trouver une solution optimale au jeu du taquin.

Les parties I, II et III peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes. La partie IV suppose qu'on a lu entièrement l'énoncé de la partie III.

Complexité

Par *complexité en temps* d'un algorithme A on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc) nécessaires à l'exécution de A dans le cas le pire. Par *complexité en espace* d'un algorithme A on entend l'espace mémoire minimal nécessaire à l'exécution de A dans le cas le pire. Lorsque la complexité en temps ou en espace dépend d'un ou plusieurs paramètres $\kappa_0, \cdots, \kappa_{r-1}$, on dit que A a une complexité en $\mathcal{O}(f(\kappa_0, \cdots, \kappa_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $\kappa_0, \cdots, \kappa_{r-1}$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $\kappa_0, \cdots, \kappa_{r-1}$, la complexité est au plus $C\, f(\kappa_0, \cdots, \kappa_{r-1})$.

```
BFS()
    A ← {e_0}
    p ← 0
    tant que A ≠ ∅
        B ← ∅
        pour tout x ∈ A
            si x ∈ F alors
                renvoyer VRAI
            B ← s(x) ∪ B
        A ← B
        p ← p + 1
    renvoyer FAUX
```

FIGURE 1 – Parcours en largeur.

Partie I. Jeu à un joueur, parcours en largeur

Un jeu à un joueur est la donnée d'un ensemble non vide E, d'un élément $e_0 \in E$, d'une fonction $s : E \to \mathcal{P}(E)$ et d'un sous-ensemble F de E. L'ensemble E représente les états possibles du jeu. L'élément e_0 est l'état initial. Pour un état e, l'ensemble $s(e)$ représente tous les états atteignables en un coup à partir de e. Enfin, F est l'ensemble des états gagnants du jeu. On dit qu'un état e_p est à la *profondeur* p s'il existe une séquence finie de $p+1$ états

$$e_0 \; e_1 \; \ldots \; e_p$$

avec $e_{i+1} \in s(e_i)$ pour tout $0 \leq i < p$. Si par ailleurs $e_p \in F$, une telle séquence est appelée une *solution* du jeu, de profondeur p. Une solution *optimale* est une solution de profondeur minimale. On notera qu'un même état peut être à plusieurs profondeurs différentes.

Voici un exemple de jeu :

$$\begin{aligned} E &= \mathbb{N}^\star \\ e_0 &= 1 \\ s(n) &= \{\, 2n,\, n+1 \,\} \end{aligned} \tag{1}$$

Question 1. Donner une solution optimale pour ce jeu lorsque $F = \{\, 42 \,\}$.

Parcours en largeur. Pour chercher une solution optimale pour un jeu quelconque, on peut utiliser un parcours en largeur. Un pseudo-code pour un tel parcours est donné figure 1.

Question 2. Montrer que le parcours en largeur renvoie VRAI si et seulement si une solution existe.

Question 3. On se place dans le cas particulier du jeu (1) pour un ensemble F arbitraire pour lequel le parcours en largeur de la figure 1 termine. Montrer alors que la complexité en temps et en espace est exponentielle en la profondeur p de la solution trouvée. On demande de montrer que la complexité est bornée à la fois inférieurement et supérieurement par deux fonctions exponentielles en p.

```
DFS(m, e, p)
    si p > m alors
        renvoyer FAUX
    si e ∈ F alors
        renvoyer VRAI
    pour chaque x dans s(e)
        si DFS(m, x, p + 1) = VRAI alors
            renvoyer VRAI
    renvoyer FAUX
```

FIGURE 2 – Parcours en profondeur (partie II), limité par une profondeur maximale m.

Programmation. Dans la suite, on suppose donnés un type `etat` et les valeurs suivantes pour représenter un jeu en Caml :

```
initial: etat
suivants: etat -> etat list
final: etat -> bool
```

Question 4. Écrire une fonction `bfs: unit -> int` qui effectue un parcours en largeur à partir de l'état initial et renvoie la profondeur de la première solution trouvée. Lorsqu'il n'y a pas de solution, le comportement de cette fonction pourra être arbitraire.

Indication : On pourra avantageusement réaliser les ensembles A et B par des listes, sans chercher à éliminer les doublons, et utiliser une fonction récursive plutôt qu'une boucle `while`.

Question 5. Montrer que la fonction `bfs` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Partie II. Parcours en profondeur

Comme on vient de le montrer, l'algorithme BFS permet de trouver une solution optimale mais il peut consommer un espace important pour cela, comme illustré dans le cas particulier du jeu (1) qui nécessite un espace exponentiel. On peut y remédier en utilisant plutôt un parcours en profondeur. La figure 2 contient le pseudo-code d'une fonction DFS effectuant un parcours en profondeur à partir d'un état e de profondeur p, sans dépasser une profondeur maximale m donnée.

Question 6. Montrer que DFS(m, e_0, 0) renvoie VRAI si et seulement si une solution de profondeur inférieure ou égale à m existe.

Recherche itérée en profondeur. Pour trouver une solution optimale, une idée simple consiste à effectuer un parcours en profondeur avec $m = 0$, puis avec $m = 1$, puis avec $m = 2$, etc., jusqu'à ce que DFS(m, e_0, 0) renvoie VRAI.

Question 7. Écrire une fonction `ids: unit -> int` qui effectue une recherche itérée en profondeur et renvoie la profondeur d'une solution optimale. Lorsqu'il n'y a pas de solution, cette fonction ne termine pas.

Question 8. Montrer que la fonction `ids` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Question 9. Comparer les complexités en temps et en espace du parcours en largeur et de la recherche itérée en profondeur dans les deux cas particuliers suivants :

1. il y a exactement un état à chaque profondeur p ;
2. il y a exactement 2^p états à la profondeur p.

On demande de justifier les complexités qui seront données.

Partie III. Parcours en profondeur avec horizon

On peut améliorer encore la recherche d'une solution optimale en évitant de considérer successivement toutes les profondeurs possibles. L'idée consiste à introduire une fonction $h : E \to \mathbb{N}$ qui, pour chaque état, donne un minorant du nombre de coups restant à jouer avant de trouver une solution. Lorsqu'un état ne permet pas d'atteindre une solution, cette fonction peut renvoyer n'importe quelle valeur.

Commençons par définir la notion de *distance* entre deux états. S'il existe une séquence de $k+1$ états $x_0\ x_1\ \cdots\ x_k$ avec $x_{i+1} \in s(x_i)$ pour tout $0 \leq i < k$, on dit qu'il y a un chemin de longueur k entre x_0 et x_k. Si de plus k est minimal, on dit que la distance entre x_0 et x_k est k.

On dit alors que la fonction h est *admissible* si elle ne surestime jamais la distance entre un état et une solution, c'est-à-dire que pour tout état e, il n'existe pas d'état $f \in F$ situé à une distance de e strictement inférieure à $h(e)$.

On procède alors comme pour la recherche itérée en profondeur, mais pour chaque état e considéré à la profondeur p on s'interrompt dès que $p + h(e)$ dépasse la profondeur maximale m (au lieu de s'arrêter simplement lorsque $p > m$). Initialement, on fixe m à $h(e_0)$. Après chaque parcours en profondeur infructueux, on donne à m la plus petite valeur $p + h(e)$ qui a dépassé m pendant ce parcours, le cas échéant, pour l'ensemble des états e rencontrés dans ce parcours. La figure 3 donne le pseudo-code d'un tel algorithme, appelé IDA*, où la variable globale min est utilisée pour retenir la plus petite valeur ayant dépassé m.

Question 10. Écrire une fonction `idastar: unit -> int` qui réalise l'algorithme IDA* et renvoie la profondeur de la première solution rencontrée. Lorsqu'il n'y a pas de solution, le comportement de cette fonction pourra être arbitraire. Il est suggéré de décomposer le code en plusieurs fonctions. On utilisera une référence globale `min` dont on précisera le type et la valeur retenue pour représenter ∞.

Question 11. Proposer une fonction h admissible pour le jeu (1), non constante, en supposant que l'ensemble F est un singleton $\{t\}$ avec $t \in \mathbb{N}^\star$. On demande de justifier que h est admissible.

Question 12. Montrer que, si la fonction h est admissible, la fonction `idastar` renvoie toujours une profondeur optimale lorsqu'une solution existe.

```
DFS*(m, e, p) ≝
    c ← p + h(e)
    si c > m alors
        si c < min alors
            min ← c
        renvoyer FAUX
    si e ∈ F alors
        renvoyer VRAI
    pour chaque x dans s(e)
        si DFS*(m, x, p + 1) = VRAI alors
            renvoyer VRAI
    renvoyer FAUX
```

```
IDA*() ≝
    m ← h(e_0)
    tant que m ≠ ∞
        min ← ∞
        si DFS*(m, e_0, 0) = VRAI alors
            renvoyer VRAI
        m ← min
    renvoyer FAUX
```

FIGURE 3 – Pseudo-code de l'algorithme IDA*.

Partie IV. Application au jeu du taquin

Le jeu du taquin est constitué d'une grille 4×4 dans laquelle sont disposés les entiers de 0 à 14, une case étant laissée libre. Dans tout ce qui suit, les lignes et les colonnes sont numérotées de 0 à 3, les lignes étant numérotées du haut vers le bas et les colonnes de la gauche vers la droite. Voici un état initial possible :

	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 du taquin est de parvenir à l'état final suivant :

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

Ici, on peut le faire à l'aide de 49 déplacements supplémentaires et il n'est pas possible de faire moins.

Question 13. En estimant le nombre d'états du jeu du taquin, et la place mémoire nécessaire à la représentation d'un état, expliquer pourquoi il n'est pas réaliste d'envisager le parcours en largeur de la figure 1 pour chercher une solution optimale du taquin.

Fonction h. On se propose d'utiliser l'algorithme IDA* pour trouver une solution optimale du taquin et il faut donc choisir une fonction h. On repère une case de la grille par sa ligne i (avec $0 \leq i \leq 3$, de haut en bas) et sa colonne j (avec $0 \leq j \leq 3$, de gauche à droite). Si e est un état du taquin et v un entier entre 0 et 14, on note e_v^i la ligne de l'entier v dans e et e_v^j la colonne de l'entier v dans e. On définit alors une fonction h pour le taquin de la façon suivante :

$$h(e) = \sum_{v=0}^{14} |e_v^i - \lfloor v/4 \rfloor| + |e_v^j - (v \bmod 4)|.$$

Question 14. Montrer que cette fonction h est admissible.

Programmation. Pour programmer le jeu de taquin, on abandonne l'idée d'un type `etat` et d'une fonction `suivants`, au profit d'un unique état global et modifiable. On se donne pour cela une matrice `grid` de taille 4×4 contenant l'état courant e ainsi qu'une référence `h` contenant la valeur de $h(e)$.

```
grid: int vect vect
h: int ref
```

La matrice `grid` est indexée d'abord par i puis par j. Ainsi `grid.(`i`).(`j`)` est l'entier situé à la ligne i et à la colonne j. Par ailleurs, la position de la case libre est maintenue par deux références :

```
li: int ref
lj: int ref
```

La valeur contenue dans `grid` à cette position est non significative.

Question 15. Écrire une fonction `move: int -> int -> unit` telle que `move` i j déplace l'entier situé dans la case (i, j) vers la case libre supposée être adjacente. On prendra soin de bien mettre à jour les références `h`, `li` et `lj`. On ne demande pas de vérifier que la case libre est effectivement adjacente.

Déplacements et solution. De cette fonction `move`, on déduit facilement quatre fonctions qui déplacent un entier vers la case libre, respectivement vers le haut, la gauche, le bas et la droite.

```
let haut   () = move (!li + 1) !lj;;
let gauche () = move !li (!lj + 1);;
let bas    () = move (!li - 1) !lj;;
let droite () = move !li (!lj - 1);;
```

Ces quatre fonctions supposent que le mouvement est possible.

Pour conserver la solution du taquin, on se donne un type `deplacement` pour représenter les quatre mouvements possibles et une référence globale `solution` contenant la liste des déplacements qui ont été faits jusqu'à présent.

```
type deplacement = Gauche | Bas | Droite | Haut
solution: deplacement list ref
```

La liste `!solution` contient les déplacement effectués dans l'ordre inverse, *i.e.*, la tête de liste est le déplacement le plus récent.

Question 16. Écrire une fonction `tente_gauche: unit -> bool` qui tente d'effectuer un déplacement vers la gauche si cela est possible et si le dernier déplacement effectué, le cas échéant, n'était pas un déplacement vers la droite. Le booléen renvoyé indique si le déplacement vers la gauche a été effectué. On veillera à mettre à jour `solution` lorsque c'est nécessaire.

On suppose avoir écrit de même trois autres fonctions

```
tente_bas   : unit -> bool
tente_droite: unit -> bool
tente_haut  : unit -> bool
```

Question 17. Écrire une fonction `dfs: int -> int -> bool` correspondant à la fonction DFS* de la figure 3 pour le jeu du taquin. Elle prend en argument la profondeur maximale m et la profondeur courante p. (L'état e est maintenant global.)

Question 18. En déduire enfin une fonction `taquin: unit -> deplacement list` qui renvoie une solution optimale, lorsqu'une solution existe.

Note : Trouver une solution au taquin n'est pas très compliqué, mais trouver une solution optimale est nettement plus difficile. Avec ce qui est proposé dans la dernière partie de ce sujet, on y parvient en moins d'une minute pour la plupart des états initiaux et en une dizaine de minutes pour les problèmes les plus difficiles.

* *
*