DM13

Dans ce problème, les programmes sont écrits en langage OCaml.

Dans un jeu à un joueur, une configuration initiale du jeu est donnée et le joueur effectue une série de déplacements pour parvenir à une configuration gagnante. Des casse-têtes 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 première partie introduit la notion de jeu à un joueur et un premier algorithme pour trouver une solution optimale. Les deux parties suivantes proposent d'autres algorithmes, plus efficaces. La dernière partie présente une solution optimale au jeu de taquin. Les trois premières parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes. La dernière partie suppose qu'on a lu entièrement l'énoncé de la partie précédente.

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 $0 \leqslant i < p$. 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.

Question 1. Voici un exemple de jeu :

$$E = \mathbb{N}^* \qquad e_0 = 1 \qquad s(n) = \{2n, n+1\}$$

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

Question 2. 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 dans l'algorithme 1. 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.

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.

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

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. L'algorithme 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.

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.

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 $h : E \to \mathbb{N}$ [1] 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 $x_0 \dots x_k$ avec $x_{i+1} \in s(x_i)$ pour tout $0 \leqslant i < k$, 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. L'algorithme 4 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 utilise une référence globale `min` dont on précise 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 $t \in \mathbb{N}^*$. 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.

Application au jeu du taquin

Le jeu de 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. Un état initial possible est représenté ci-dessous, à gauche.

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

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

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

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 représenté au milieu, ci-dessus. Le but du jeu de taquin est de parvenir à l'état final représenté à droite, ci-dessus. 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 de 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.

Question 14. Montrer que cette fonction h est admissible.

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.

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

1. Une remarque similaire s'appliquerait à l'algorithme bfs, mais notre version est susceptible d'être optimisée parce que récursive terminale.

booléen renvoyé indique si le déplacement vers la gauche a été effectué. On veillera à mettre à jour solution lorsque c'est nécessaire.

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.

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.

Algorithmes

Algorithme 1 : parcours en largeur

```
1  fonction BFS()
2      A ← {e_0}
3      p ← 0
4      tant que A ≠ 0 faire
5          B ← 0
6          pour x ∈ A faire
7              si x ∈ F alors
8                  renvoyer Vrai
9              B ← s(x) ∪ B
10         A ← B
11         p ← p + 1
12     renvoyer Faux
```

Algorithme 2 : parcours en profondeur limité par une profondeur maximale m

```
1  fonction DFS(m, e, p)
2      si p > m alors
3          renvoyer Faux
4      si e ∈ F alors
5          renvoyer Vrai
6      pour chaque x ∈ s(e) faire
7          si DFS(m, x, p + 1) alors
8              renvoyer Vrai
9      renvoyer Faux
```

Algorithme 3 : parcours en profondeur DFS*

```
1  fonction DFS*(m, e, p)
2      c ← p + h(e)
3      si c > m alors
4          si x < min alors
5              min ← c
6          renvoyer Faux
7      si e ∈ F alors
8          renvoyer Vrai
9      pour chaque x ∈ s(e) faire
10         si DFS*(m, x, p + 1) alors
11             renvoyer Vrai
12     renvoyer Faux
```

Algorithme 4 : pseudo-code de l'algorithme IDA*

```
1  fonction IDA*()
2      m ← h(e_0)
3      tant que m ≠ ∞ faire
4          min ← ∞
5          si DFS*(m, e_0, 0) alors
6              renvoyer Vrai
7          m ← min
8      renvoyer Faux
```