

```

BFS()
   $A \leftarrow \{e_0\}$ 
   $p \leftarrow 0$ 
  tant que  $A \neq \emptyset$ 
     $B \leftarrow \emptyset$ 
    pour tout  $x \in A$ 
      si  $x \in F$  alors
        renvoyer VRAI
       $B \leftarrow s(x) \cup B$ 
     $A \leftarrow B$ 
     $p \leftarrow p + 1$ 
  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 $s : E \rightarrow \mathcal{P}(E)$ 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

$$e_0 \ e_1 \ \dots \ e_p$$

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 :

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

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  $e \in F$  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 $h : E \rightarrow \mathbb{N}$ 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 $x_0 \ x_1 \ \dots \ x_k$ 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 $t \in \mathbb{N}^*$. 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.

<pre> DFS*(m, e, p) $\stackrel{\text{def}}{=}$ c $\leftarrow p + h(e)$ si c > m alors si c < min alors min $\leftarrow c$ renvoyer FAUX si e $\in F$ alors renvoyer VRAI pour chaque x dans s(e) si DFS*(m, x, p + 1) = VRAI alors renvoyer VRAI renvoyer FAUX </pre>	<pre> IDA*() $\stackrel{\text{def}}{=}$ m $\leftarrow h(e_0)$ tant que m $\neq \infty$ min $\leftarrow \infty$ si DFS*(m, e₀, 0) = VRAI alors renvoyer VRAI m $\leftarrow \min$ renvoyer FAUX </pre>
---	--

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	