D.S. 03+ X-ENS 2017

Jeux à un joueur

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 de 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 $k_0, \dots, k_{r-1}$, on dit que A a une complexité en $O(f(k_0, \dots, k_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $k_0, \dots, k_{r-1}$ suffisamment grandes (c'est à dire plus grandes qu'un certain seuil), pour toutes instance du problème de paramètres $k_0, \dots, k_{r-1}$, la complexité est au plus $Cf(k_0, \dots, k_{r-1})$.

1 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, $s(e)$ est l'ensemble des 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 $(e_0, e_1, \cdots, e_p)$ 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.

Voici un exemple de jeu :

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

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 ci-après.

Algorithme 1 BFS()

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

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 (I.1) pour un ensemble F arbitraire pour lequel le parcours en largeur BFS 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.

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.*

2 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 (I.1) qui nécessite un espace exponentiel.
On peut y remédier en utilisant plutôt un parcours en profondeur.
Voici 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.

Algorithme 2 DFS(m,e,p)

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

Question 6
Montrer que $\texttt{DFS}(m, e_0, 0)$ *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 $\texttt{DFS}(m, e_0, 0)$ 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.

3 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 $x_0 \ \ldots \ x_k$ 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.

Voici 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.

Algorithme 3 DFS*(m,e,p)

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

Algorithme 4 IDA*()

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

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 ∞.
La réponse devra contenir l'écriture de l'algorithme DFS*.

Question 11
Proposer une fonction h admissible pour le jeu (I.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.*

4 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. 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 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	

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.

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 $0 \leqslant i \leqslant 3$, de haut en bas) et sa colonne j (avec $0 \leqslant j \leqslant 3$, 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 :

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

Question 14
Montrer que cette fonction h est admissible.

4.1 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.*

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 déplacements 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* pour le jeu du taquin. Elle prend en argument la profondeur maximale m et la profondeur courante p.*
On rappelle que l'état e est maintenant global, déterminé par `grid`*.*

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.

5 Solutions

Solution de la question 1 - On peut lister les nombres qu'on peut atteindre avec une profondeur donnée :

- 0 : 1
- 1 : 2
- 2 : 3 ; 4
- 3 : 5 ; 6 ; 8
- 4 : 7 ; 9 ; 10 ; 12 ; 16
- 5 : 11 ; 13 ; 14 ; 17 ; 18 ; 20 ; 24 ; 32
- 6 : 15 ; 19 ; 21 ; 22 ; 25 ; 26 ; 28 ; 33 ; 34 ; 36 ; 40 ; 48 ; 64
- 7 : 23 ; 27 ; 29 ; 30 ; 35 ; 37 ; 38 ; 41 ; 42 ; 44 ; 49 ; 50 ; ...

On a alors la solution optimale : 1 ; 2 ; 4 ; 5 ; 10 ; 20 ; 21 ; 42.

Solution de la question 2 - On remarque que, lors du passage de la boucle `tant que` pour la valeur p,

- A ne contient que des états accessibles,
- ces états sont de profondeur au plus p,
- A contient tous les états de profondeur p.

On note $\mathcal{P}(p)$ ces propriétés.
$\mathcal{P}(0)$ est vraie car A contient initialement e_0, seul état accessible de profondeur 0.
Si la propriété $\mathcal{P}(p)$ est vérifiée et qu'aucun état de A n'appartient à F alors on place dans B les voisins des états de A :

- ces voisins d'états accessibles sont donc accessibles,
- ils sont voisins d'états de profondeur p au plus sont de profondeur $p+1$ au plus,
- comme tous les états de profondeur $p+1$ sont voisins d'états de profondeur p, éléments de A, ils appartiennent à B.

Après avoir remplacé p par $p+1$ et A par B on voit donc que $\mathcal{P}(p+1)$ est vérifiée.
On a donc prouvé par récurrence que $\mathcal{P}(p)$ est vérifiée pour tout p qui correspond à un passage de la boucle `tant que`.

Ainsi, si le parcours renvoie VRAI, c'est qu'on a trouvé un état accessible dans F donc qu'il existe solution. S'il existe une solution, il en existe une qui est optimale.

Inversement s'il existe une solution (optimale) alors il existe un état t accessible dans F.
Si t est à la profondeur p il sera découvert lors du $p+1$-ième passage, sauf si un autre sommet accessible a été découvert auparavant. Dans les deux cas l'algorithme renvoie VRAI.

Solution de la question 3 - On commence par encadrer la complexité spatiale.
Chaque élément de A admet deux voisins donc la taille de B est au plus le double de celle de A.
Si on note a_p la taille de A lors du passage de la boucle pour la valeur p on a $a_{p+1} \leqslant 2a_p$ donc, comme $a_0 = 1$, $a_p \leqslant 2^p$.
On a $B = B_1 \cup B_2$ avec $B_1 = \{x+1 \,;x \in A\}$ et $B_2 = \{2x \,;x \in A\}$.
B_1 et B_2 on le même cardinal que A et sont formés d'éléments distincts.
Touts les éléments de B_2 sont pairs donc la moitié, au moins, des éléments de B sont pairs.
Ainsi la moitié, au moins, des éléments de A sont pairs pour $p \geqslant 1$.
Ces élément pairs vont donner dans B_1 des éléments impairs donc distincts de ceux de B_2 donc il y a au moins $a_p/2 + a_p$ éléments distincts dans B ; ainsi $a_{p+1} \geqslant \frac{3}{2}a_p$ pour $p \geqslant 1$. Comme on a $a_1 = 2 \geqslant \frac{3}{2}$ on en déduit $a_p \geqslant \left(\frac{3}{2}\right)^p$.
Ainsi la complexité spatiale, qui est la somme des taille de A et de B donc égale à $a_p + a_{p+1}$, est encadrée par deux exponentielles :

$$\left(\frac{3}{2}\right)^p \leqslant \left(\frac{3}{2}\right)^p + \left(\frac{3}{2}\right)^{p+1} \leqslant a_p + a_{p+1} \leqslant 2^p + 2^{p+1} \leqslant 2^{p+2}$$

La complexité temporelle d'une étape est minorée par la taille de B car il faut au moins une instruction par élément ajouté.

Pour chaque x de A on fait un nombre fini d'opération élémentaires puis on ajoute chacun des deux voisins : pour effectuer une union il faut, au pire tester tous les éléments de B pour ne pas ajouter de doublon donc on effectue au plus $a_p(C + 2a_{p+1})$ instructions.
La complexité temporelle est donc majorée par
$\sum_{k=0}^{p} a_p(C + 2a_{p+1}) \leqslant \sum_{k=0}^{p} 2^{k+2}(C + 2.2^{k+3}) \leqslant C2^{p+3} + 2^{2p+6} \leqslant C'.4^p.$

Les deux complexités sont bien encadrées par des exponentielles.

Solution de la question 4 -

```
let bfs () =
  let rec successeurs A B p =
    match A with
    |[] -> successeurs B [] (p+1)
    |(t::q) -> if final t
               then p
               else successeurs q (suivants t @ B) p
    in successeurs [initial] [] 0 ;;
```

La fonction `successeurs` prend en entrée l'état des variables `A`, `B` et `p` décrites dans le sujet, et renvoie la solution. Le cas `A = []` correspond à la fin de la boucle **pour tout**.

Solution de la question 5 - On a montré à la question 2 que si l'algorithme termine en renvoyant VRAI au passage pour la valeur p alors l'état appartenant à F considéré est à la profondeur p au plus.
Il ne peut pas être à une profondeur $k < p$ car sinon il était dans l'ensemble A lors du passage pour la valeur p et l'algorithme aurait terminé lors de ce passage.
Ainsi la valeur renvoyée est la profondeur d'une solution.
Si ce n'était pas la la profondeur d'une solution optimale alors un état correspondant à une solution optimale aurait appartenu à A lors d'une valeur antérieure de p ce qui aurait fait terminer l'algorithme plus tôt : c'est impossible.
l'algorithme renvoie la profondeur d'une solution optimale.

Solution de la question 6 -
Supposons qu'il existe une solution de profondeur inférieure ou égale à m : $(e_0, \ldots, e_p)$.
e_p est un voisin de e_{p-1} donc DFS$(m, e_{p-1}, p-1)$ renvoie VRAI.
On prouve ainsi, par récurrence descendante, que DFS(m, e_k, k) renvoie VRAI pour tout $k \in \{0, \ldots, p\}$ donc, en particulier, DFS$(m, e_0, 0)$ renvoie VRAI.

Inversement, on démontre, par récurrence descendante sur p que si DFS(m, e, p) renvoie VRAI alors il existe une solution depuis e de profondeur $m - p$ au plus.
Pour $p = m$ on ne teste que e : on a donc $e \in F$ et (e) est une solution de profondeur 0.
On suppose la propriété vérifiée pour $p + 1$ et on suppose que DFS(m, e, p) renvoie VRAI.
Alors, soit $e \in F$ et on a une solution de profondeur $0 \leqslant m - p$,
soit un successeur de e, que l'on notera e', valide DFS$(m, e', p+1)$. D'après l'hypothèse de récurrence il existe une solution depuis e' de profondeur $m - p - 1$ au plus. On la prolonge en ajoutant e à gauche et on obtient une solution depuis e de profondeur $m - p$ au plus.

Solution de la question 7 - On commence par la recherche en profondeur bornée.
On commence par une fonction de test d'une fonction booléenne sur une liste qui renvoie `true` si et seulement si il existe au moins un élément de la liste qui vérifie la condition.

```
let rec tester f liste =
  match liste with
  |[] -> false
  |t::q -> (f t) || (tester f q);;
```

Cette fonction existe : `List.exists`.
On suit alors l'algorithme proposé

```
let rec dfs m e p =
  if p > m
  then false
  else if final e
       then true
       else tester (fun x -> dfs m x (p+1)) (suivants e);;
```

Pour trouver la profondeur minimale on teste par valeur de m croissante

```
let ids () =
  let rec aux m =
    if dfs m initial 0
    then m
    else aux (m+1) in
  aux 0;;
```

Solution de la question 8 -
Si une solution existe on note m la profondeur optimale d'une solution.
L'algorithme `ids` va tester `dfs k initial 0` pour $k < m$ qui renvoie Faux d'après l'exercice 6
On teste ensuite `dfs m initial 0` , l'exercice 6 indique que la fonction `dfs` renvoie VRAI donc `ids` renverra m : la valeur trouvée est donc bien optimale.

Solution de la question 9 - Un état à chaque profondeur

- Quand il y a exactement un état à chaque profondeur p, le parcours en largeur n'aura qu'un élément dans A (et dans B) à chaque passage, donc une complexité spatiale en $\mathcal{O}(1)$. Si on note m la profondeur optimale, le calcul de B (et de A) se fait en temps $\mathcal{O}(m)$.
- Le parcours en profondeur n'utilise aussi que des liste de taille 1. Cependant il faut ajouter la pile des appels récursif, en $\mathcal{O}(m)$.
  On explore aux profondeurs 1, puis 2, puis etc. jusqu'à arriver à m.
  Chaque exploration demande un temps linéaire en la profondeur souhaitée, soit une complexité totale en $\mathcal{O}\big(m^2\big)$.

2^p états à la profondeur p

- S'il y a exactement 2^p états à la profondeur p, le parcours en largeur aura besoin de stocker les 2^p à chaque étape, soit une complexité spatiale en $\mathcal{O}\big(2^m\big)$ pour la dernière étape.
  Le temps de calcul sera également de l'ordre de $\mathcal{O}\big(2^m\big)$.
- Le parcours en profondeur n'utilisera lui que la taille de sa pile d'appels, en $\mathcal{O}(m)$.
  Le temps de calcul sera toujours de l'ordre de $\mathcal{O}\big(2^m\big)$.

Solution de la question 10 -
On peut représenter l'infini par `let min = ref max_int`, `min` est alors un entier.

```
let rec dfsstar m e p =
  let c = p + h e in
  if c > m
  then (if c < !min then min := c;
        false)
  else begin
         if final e
         then true
         else tester (fun x-> dfsstar m x (p+1)) (suivants e)
       end;;
```

```
let idastar () =
 let rec idastar_aux m =
   if m = max_int
   then max_int
   else begin min := max_int ;
              if dfsstar m initial 0
              then m
              else idastar_aux (!min)
        end in
  idastar_aux (h initial) ;;
```

Solution de la question 11 -
Pour le jeu (1), un état q accessible à partir de p par un chemin de longueur k vérifie $q \leqslant 2^k.p$.
Pour tout p il existe un entier k tel que $2^{k-1}.p < t \leqslant 2^k.p$ où t est l'objectif : il faut au moins jouer encore k coups à partir de p avant de trouver une solution.
$h : p \mapsto \lceil \log_2(t/p) \rceil$ est donc une fonction admissible.

Solution de la question 12 - On commence par prouver que `DFS*`$(m, e_0, 0)$ renvoie VRAI si et seulement si il existe une solution de profondeur inférieure ou égale à m.
$(e_0, \ldots, e_p)$ est une solution avec $p \leqslant m$.
Pour tout k on doit avoir $h(e_k) \leqslant p - k$ car la distance de e_k à F est au plus $p - k$.
On a alors $e_p \in F$ et $p + h(e_p) = p \leqslant m$ donc DFS*(m, e_p, p) renvoie VRAI.
$p - 1 + h(e_{p-1}) \leqslant p - 1 + 1 \leqslant m$ et e_p est un voisin de e_{p-1} donc DFS*$(m, e_{p-1}, p-1)$ renvoie VRAI.
On prouve ainsi, par récurrence descendante, que DFS*(m, e_k, k) renvoie VRAI pour tout $k \in \{0, \ldots, p\}$ donc, en particulier, DFS*$(m, e_0, 0)$ renvoie VRAI.

La réciproque se démontre comme dans l'exercice 6.

Une solution doit avoir une profondeur au moins $h(e_0)$: c'est la première tentative faite.
Si DFS*$(m, e_0, 0)$ a renvoyé FAUX alors, lors du calcul on a ramené min à la plus petite valeur pour laquelle il pourrait exister une solution ayant cette profondeur. Cela démontre qu'il n'y a pas de solution de profondeur $k < min$.
On essaye alors avec DFS*$(min, e_0, 0)$.
Dès que DFS*$(m, e_0, 0)$ renvoie VRAI, m est bien la profondeur optimale.

Solution de la question 13 - Chaque état est une permutation des 16 éléments (case vide comprise) de l'état final : il y en a donc 16!.
Un entier de $\{0, 1, 2, \ldots, 15\}$ est représentable avec 4 bits donc chaque état est représentable avec 64 bits, 8 octets. L'ensemble des états demandera donc 16!.8 octets. Comme on a $16! > \left(\frac{16}{e}\right)^{16} > 5^{16}$, l'ensemble des états demande plus de $8.5^{16} > 10^{12}$ octets.
Chaque état admet au moins deux voisins donc, à la profondeur 49, on a un ensemble de voisins atteints de l'ordre de 2^{49}, qui est supérieur à 16!. La taille ci-dessus est un ordre de grandeur atteint par la liste A.
On atteint une taille supérieure à un téra-octet, non réaliste.

Solution de la question 14 - Dans l'état final, l'entier $k \in \{0, 1, \ldots, 14\}$ se trouve à la ligne `k/4` et à la colonne `k mod 4`. Chaque mouvement modifie soit la ligne soit la colonne d'une unité donc le nombre d'étapes pour remettre en place l'entier k est au moins $|e_k^i - k/4| + |e_k^j - (k \bmod 4)|$.
Ainsi $h(e)$ minore le nombre de coups à jouer à partir de e.

Solution de la question 15 - On modifie la valeur de `h` en ajoutant la valeur pour la nouvelle position `(li, lj)` et en enlevant la valeur pour l'ancienne position `(i, j)` devenue vide.

```
let move i j =
  let k = grid.(i).(j) in
  grid.(!li).(!lj) <- k;
  h := !h - abs (i - k/4) - abs (j - k mod 4)
          + abs (!li - k/4) + abs (!lj - k mod 4);
  li := i;
  lj := j;;
```

Solution de la question 16 - On commence par écrire la fonction testant si une liste ne commence pas par `droite`

```
let debut_non_droite liste =
  match liste with
  |Droite::_ -> false
  |_ -> true;;
```

```
let tente_gauche () =
  if !lj < 3 && debut_non_droite !solution
  then begin
         gauche ();
         solution := Gauche::!solution;
         true
       end
  else false ;;
```

Solution de la question 17 - La position finale est reconnue par `!h` nul.
Il ne faut pas oublier de revenir à la position initiale si un mouvement était possible mais n'aboutit pas.

```
let rec dfs m p =
  let c = p + !h in
  if c > m
  then (if c < !min then !min := c; false)
  else if !h = 0
       then true
       else begin
         let reponse = ref false in
         if tente_gauche ()
         then if dfs m (p+1) then reponse := true else droite
            ();
         if tente_droite ()
         then if dfs m (p+1) then reponse := true else gauche
            ();
         if tente_haut ()
         then if dfs m (p+1) then reponse := true else bas();
         if tente_bas ()
         then if dfs m (p+1) then reponse := true else haut();
         !reponse end;;
```

Solution de la question 18 - On commence par l'initialisation de `h` en fonction de la grille initiale.

```
let h = ref 0;
for i = 0 to 3 do
  for j = 0 to 3 do
    if i <> !li || j <> !lj
    then let k = grid.(i).(j) in
        h := !h + abs (i - (k/4)) + abs (j - (k mod 4)) done
           done;;
```

```
let taquin () =
   let m = ref !h in
   let reponse = ref (-1) in
   while !m <> max_int do
      mini := max_int;
      if dfs !m  0
      then begin reponse := !m;
                 m := max_int end
      else m := !mini done;
   List.rev !solution;;
```