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

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, 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.

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}

▶ **Réponse 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.

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.

```
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

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

▶ **Réponse 2** À la lecture de l'algorithme, on constate que A et B ne contiennent que des états atteignables depuis e_0. En effet, A est initialisé à e_0 et ne peut être remplacé que par B, et B est réinitialisé à chaque passage dans la boucle **tant que** et on ne lui ajoute que les ensembles $s(x)$ où x est dans A. Ainsi, s'il n'existe aucun état final atteignable depuis e_0, alors A ne contiendra aucun état atteignable, donc l'algorithme ne renverra pas VRAI.

Inversement, supposons qu'une solution $e_0 \dots e_p$ existe. Alors par construction de l'ensemble B, B (puis A) contiendra à la fin du k-ème passage dans la boucle **tant que** l'élément e_k (ou aura déjà renvoyé VRAI avant). En effet, si e_k est dans A après le k-ème passage, alors on ajoute $s(e_k)$ à B, et donc en particulier on aura e_{k+1} dans B après la $(k+1)$-ème itération.

Finalement, e_p étant dans F, si l'algorithme n'a pas renvoyé VRAI avant, il s'arrêtera au p-ème passage car $e_p \in A$ et $e_p \in F$.

▶ **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.

▶ **Réponse 3** Montrons d'abord qu'on peut borner supérieurement cette complexité temporelle (et donc spatiale : il faut au moins une opération pour écrire une valeur en mémoire) par une fonction exponentielle. Supposons donc qu'il existe une solution de profondeur p.

D'après la réponse précédente, on fera au plus p passages dans la boucle **tant que** de l'algorithme BFS(). On considère qu'on peut faire les différentes affectations en $O(1)$, il faut donc simplement évaluer le temps pour réaliser la ligne $B \leftarrow s(x) \cup B$. Or on sait que $s(x)$ contient exactement deux éléments pour chaque x. Ainsi, par une induction élémentaire, B contient au plus deux fois plus d'éléments que A et la taille de A au k-ème passage est donc bornée par 2^k. Le calcul de B au k-ème passage se fait donc par une union d'au plus 2^k éléments, ce qui peut se réaliser en $2^k \times 2^k$ opérations (en majorant largement). Le temps de calcul total est donc borné par la somme des 4^k, qui est donc un $O(4^n)$.

Pour la borne inférieure, nous allons montrer qu'au k-ème passage de l'algorithme, l'ensemble A est de taille minorée par un nombre exponentiel, et qu'il faut donc cette complexité exponentielle en temps et en espace pour le construire. Pour cela montrons que la taille de A est multipliée par $3/2$ au moins à chaque passage. Et pour arriver à ce résultat, nous allons démontrer en même temps que A contient toujours au moins une moitié d'entiers pairs après la première itération.

En effet, quel que soit le contenu de A à un passage, alors l'ensemble A au passage suivant se construit avec :

— Tous les entiers de A augmentés de 1

— Tous les entiers de A multipliés par 2

Il est clair que les entiers de la deuxième catégorie sont tous pairs, et tous distincts, et qu'ils représentent donc au moins la moitié des nombres dans A.

Montrons maintenant que la taille de A est multipliée par au moins $3/2$. Soit n le nombre d'entiers dans A. Il est clair qu'au passage suivant, A contiendra tous les entiers de A multipliés par deux, et qu'ils sont tous distincts, ce qui représente n valeurs. De plus, les entiers pairs de A sont au moins $n/2$, et deviennent impairs quand on leur ajoute 1. Il ne peut alors pas y avoir de collision avec les entiers doublés, et cela ajoute donc au moins $n/2$ valeurs dans l'ensemble.

Ainsi, la taille des A est multipliée au moins par $3/2$ à chaque passage dans la boucle, et on ne peut donc pas être en moins que $O\left(\left(\frac{3}{2}\right)^n\right)$.

Note : On peut aussi montrer que tous les entiers de 1 *à* 2^k *seront passés dans A au bout de* $2k$ *itérations, en considérant les décompositions en binaires, et montrer que cela implique en particulier une taille exponentielle pour l'ensemble A.*

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`.

▶ **Réponse 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**.

Notons que le fait de ne pas chercher à éliminer les doublons peut conduire à des augmentations de complexité (on peut le vérifier sur l'exemple `let suivants k = [0;1;k+1];;` dont la complexité deviendrait linéaire si on éliminait les doublons).

On peut tester ce programme avec

```
let initial = 1 ;;
let suivants k = [k+1;2*k] ;;
let final k = (k = 42) ;;
bfs()
- : int = 7
```

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

▶ **Réponse 5** On a établit à la question 2 que s'il existe une solution de jeu de longueur p, alors l'algorithme termine à la p-ème itération et renvoie VRAI. Dans notre situation, cela signifie que la condition `final t` aura été validée, et notre algorithme renvoie alors p.

S'il existait une solution meilleure, l'algorithme ne serait pas arrivé jusque là.

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 (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.

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.

▶ **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.

▶ **Réponse 6** Supposons qu'il existe une solution de profondeur inférieure ou égale à m : $e_0 \dots e_p$. Alors il est clair que $DFS(m, e_p, p)$ va renvoyer VRAI. Donc $DFS(m, e_{p-1}, p-1)$ également, et par suite, $DFS(m, e_0, 0)$ également.

Inversement, si $DFS(m, e_0, 0)$ renvoie VRAI, c'est soit que e_0 est final (auquel cas la solution existe et est de longueur nulle, ou bien qu'un successeur de e_0, que l'on notera e_1, valide $DFS(m, e_1, 1)$. La même dichotomie s'applique alors, et on peut reconstruire ainsi directement la solution. En effet, elle sera de longueur finie car bornée par m, par construction.

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.

▶ **Réponse 7**

```
let rec dfs m e p =
    p <= m && (final e || tester_succ (suivants e) m p)
and tester_succ liste m p = match liste with
       [] -> false
      |(t::q) -> dfs m t (p+1) || tester_succ q m p ;;

let ids() = let m = ref 0 in
    while not (dfs (!m) initial 0) do
          incr m ; done ; !m ;;
```

Explications : `dfs` réalise la fonction `dfs` directement, avec les mêmes arguments. `tester_succ liste m p` regarde si un des éléments de `liste` vérifie `DFS(m,x,p+1)`. La ligne du haut est rendue plus lisible en transformant les `if` en une formule booléenne (et l'évaluation paresseuse évite d'avoir des problèmes ici).

Pour `ids`, on utilise simplement une variable `m` qui contient la profondeur maximale autorisée à chaque itération.

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

▶ **Réponse 8** Supposons qu'une solution existe et soit m la profondeur optimale. Alors l'algorithme `ids` va tester $\mathrm{DFS}(k, e_0, 0)$ pour $k < m$ et ne trouvera pas de solution (par optimalité), puis testera $\mathrm{DFS}(m, e_0, 0)$ et en trouvera une, par simple application de la question 6. La valeur trouvée est donc bien optimale.

▶ **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.

▶ **Réponse 9**

1. 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 $O(1)$. Si on note m la profondeur optimale, le calcul de B (et de A) se fait en temps $O(m)$.

   Dans cette situation, l'algorithme de recherche itérée en profondeur n'a besoin lui aussi que de $O(1)$ en espace (l'algorithme doit stocker un nombre borné de variable, comme la valeur courante de m). Mais il ne faut pas oublier les appels récursifs, indispensables pour que l'algorithme sache où il en est des diverses explorations. Ceci rajoute $O(m)$ appels à stocker sur la pile en mémoire [1].

   En terme de complexité temporelle, cet algorithme doit explorer avec profondeur 1, puis 2, puis etc. jusqu'à arriver à m. Chaque exploration demande un temps linéaire en la profondeur souhaitée, soit une complexité $O(m^2)$ à la fin.

2. 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 $O(2^m)$ pour la dernière étape. Le temps de calcul sera également en $O((2+\varepsilon)^m)$ (on peut considérer qu'il y aura des temps polynomiaux en plus pour filtrer les doublons).

   Dans ce contexte, le parcours en profondeur itéré utilisera encore $O(m)$ en espace, uniquement pour stocker la pile d'appels récursifs. En temps de calcul, on devra pour chaque valeur de p entre 1 et m et pour chaque valeur de k entre 1 et p explorer les 2^k états. (p est le paramètre d'itération, et k parce qu'on explore tous ces états). Au final, la complexité temporelle est ici aussi dominée par $O((2+\varepsilon)^m)$.

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 $h : E \to \mathbb{N}$

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

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. 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.

```
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*.

▶ **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 ∞.

▶ **Réponse 10**

```
let rec dfsstar m e p =
    let c = p + h e in
    if c > m then begin
       if c < !min || !min = -1 then
           min := c ; false ;
       end
```

```
    else begin
       if final e then true
       else tester_star (suivants e) m p ; end
and tester_star l m p = match l with
     [] -> false ;
   |(t::q) -> dfsstar m t (p+1) || tester_star q m p ;;

let idastar () =
   let rec idastar_aux m =
       if m = -1 then -1
                 else begin
                      min := -1 ;
                      if dfsstar m initial 0 then
                         m
                      else
                         idastar_aux (!min) ;
                      end; in
    idastar_aux (h initial) ;;
```

▶ **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.

▶ **Réponse 11** En étant taquin, on pourrait proposer la fonction suivante : $h(n) = 0$ si $n \leqslant t$ et $h(n) = n^2$ si $n > t$. Cette fonction répond bien à la question posée, car dans le jeu (1), il n'est pas possible d'atteindre l'état t depuis une valeur qui lui est supérieure, ni en moins de 0 coup.

Mais le taquin n'étant abordé qu'à la partie IV, on propose ici une fonction plus sérieuse : on conserve l'idée que $h(n)$ prend les valeurs que l'on veut si $n > t$. Si $n \leqslant t$, on note qu'en un coup, on ne peut pas multiplier l'état par plus que 2. Ainsi, pour atteindre l'état t depuis l'état n, on doit au moins utiliser $\lfloor \log_2(t/n) \rfloor$ coups, et il suffit donc de donner cette valeur à $h(n)$.

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

▶ **Réponse 12** Supposons que la fonction h est admissible.
On note que $DFS*(m, e, p)$ renvoie une profondeur inférieure à m si elle existe.
On note également que les valeurs prises par la variable *min* sont croissantes.
Enfin, on note que si un appel à $DFS*(m, e_0, 0)$ trouvait une solution, elle serait renvoyée.
Les issues possibles de l'algorithme sont donc :

— L'algorithme renvoie FAUX. Cela signifie que m vient de passer à ∞, et on note m_f sa valeur précédente. On sait donc que le dernier appel à $DFS*(m_f, e_0, 0)$ n'a pas modifié la valeur de la variable *min*. Ce qui signifie que pour tout état e exploré de profondeur p, on avait $m_f \geq p + h(e)$. Donc en particulier, p est borné, ce qui signifie que l'exploration ne peut plus trouver de nouvel état, aucun des états explorés n'est final, et il n'y avait donc pas de solution.

— L'algorithme renvoie VRAI. On va noter m_a et m_f l'avant-dernière valeur et la dernière valeur (respectivement) données comme argument à DFS* dans le code de IDA*.

L'appel $DFS*(m_a, e_0, 0)$ ayant renvoyé FAUX, on sait qu'il n'existe pas de solution de longueur inférieure ou égale à m_a.

L'appel $DFS*(m_f, e_0, 0)$ ayant renvoyé FAUX, on sait qu'il y a une solution de longueur m_f. Il nous suffit donc de prouver que m_f est en réalité la longueur optimale.

Supposons qu'il existe une solution de longueur m, avec $m_a < m < m_f$. Sur cette solution, considérons le m_a-ème sommet. Il a été exploré au moins une fois par $DFS*$ lors de l'avant-dernier appel (avec $p = m_a$). De plus, ce sommet était à distance inférieure à $m_f - m_a$ d'un état final, donc comme h minore cette distance, on avait bien $c > m_a$ et $c < m_f$. Cela signifie que la variable *min* aurait dû prendre une valeur inférieure à c (au lieu de m_f), ce qui est bien une contradiction.

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.

► **Réponse 13** Il y a – si on suppose que toutes les configurations sont accessibles – 16! configurations possibles au taquin. On peut raisonnablement minorer par 10^12 ce nombre. S'il faut le démontrer, écrivons que $16! = 2^{15}.3^6.5^3.7^2.11.13 > 2^{10}.(2.5)^3(2^2.27).(3^3.49).11.13. > 10^3.10^3.10^2.10^3.10.10 > 10^13$. Il faut donc plus de 40 bits pour stocker un état (car $2^40 \sim 10^13$), et il faudrait donc plus de

40×10^13 octets de RAM pour une exploration exhaustive. Bref, c'est trop (plus de 10000 fois trop pour un ordinateur personnel).

Ceci reste de plus une minoration grossière, car le stockage d'un état serait sans doute moins optimisé de toute façon.

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 $0 \leqslant i \leqslant 3$, 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.

▶ **Réponse 14** Pour que cette fonction h s'annule, il faudrait avoir pour chaque v, $e_v^i = \lfloor v/4 \rfloor$ et $e_v^j = v \mod 4$. Ce n'est possible que si chaque case est exactement à sa position.

Par ailleurs, tout déplacement du taquin ne peut réduire que de 1 au maximum la valeur de h (en modifiant soit une valeur de e_v^i soit une valeur de e_v^j, de 1).

Ainsi, pour une configuration donnée e, il faut bien au moins $h(e)$ coups pour arriver à annuler la fonction h, donc pour arriver à l'état final.

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.

▶ **Réponse 15**

```
let ecart val i j =
    let oi = val / 4 and oj = val mod 4 in
    abs (oi - i) + abs (oj - j) ;;

let move i j =
```

```
let val grid.(i).(j) in
h := !h - ecart val i j + ecart val (!li) (!lj) ;
grid.(!li).(!lj) := grid.(i).(j) ;
li := i ;
lj := j ;;
```

La fonction `ecart` sert ici à calculer la contribution du nombre `val` dans la grille quand il est situé en position (i, j). On peut alors calculer la nouvelle valeu de `h` en retranchant la contribution en `(i,j)` et en ajoutant la contribution en `(!li,!lj)`.

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 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.

▶ **Réponse 16**

```
let tente_gauche () =
   match (!lj,!solution) with
     | 3, _ -> false
     | _,(Droite::q) ->  false
     | _ -> solution := Gauche :: (!solution) ;
            gauche() ; true ;;
```

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.).

▶ **Réponse 17** On introduit d'abord une fonction `cancel ()` qui annule le dernier coup joué (et renvoie toujours `false` pour les besoins de la fonction suivante) :

```
let cancel () = (match hd (!solution) with
          |Haut -> bas() ;
          |Bas -> haut() ;
          |Droite -> gauche() ;
          |Gauche -> droite() ;);
        solution := tl (!solution) ;
        false;;
```

On peut alors écrire la fonction `dfsstar` elle-même.

```
let rec dfsstar m p =
    let c = p + !h in
    if c > m then begin
       if c < !min || !min = -1 then
           min := c ;
       false;
       end
    else begin
       if !h = 0 then true
       else
         (tente_bas() && (dfsstar m (p+1) || cancel() ) ) ||
         (tente_haut() && (dfsstar m (p+1) || cancel() ) ) ||
         (tente_gauche() && (dfsstar m (p+1) || cancel() ) ) ||
         (tente_droite() && (dfsstar m (p+1) || cancel() ) );
       end;;
```

Si la valeur de `c` est trop élevée, on renvoie `false` après avoir mis à jour `min` si besoin. Si la solution est atteinte (`!h = 0`) alors on renvoie `true`. Sinon, on va tester successivement les quatre déplacements. Chaque ligne de la proposition booléenne revient à dire qu'on effectue un déplacement si c'est possible. Si non, on passe au suivant. Si oui, on continue l'exploration en profondeur, et on annule le déplacement si l'exploration n'a rien donné.

Ici, une version avec des exceptions serait sans doute plus intuitive (on peut aussi utiliser des structures `if` imbriquées, mais le résultat est moins lisible).

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

▶ **Réponse 18** Le sujet ne précise pas vraiment comment la variable `h` est initialisée. Dans le doute, on peut la fixer avec :

```
let calcul_h grid = let res = ref 0 in
  for i = 0 to 3 do
     for j = 0 to 3 do
        if (i != !li) or (j != !lj) then
              res := !res + ecart (grid.(i).(j)) i j;
      done;
  done ; !res ;;
```

```
let h = ref (calcul_h grid) ;;
```
(avec la fonction `ecart` plus haut).
De même, on pourrait poser : `let min = ref -1 ;;` .
Enfin...

```
let taquin () =
   let rec taquin_aux m =
       if m = -1 then -1
                 else begin
                      min := -1 ;
                      if dfsstar m 0 then
                           m
                      else
                           taquin_aux (!min) ;
                      end ; in
   taquin_aux (!h) ;;
```

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.