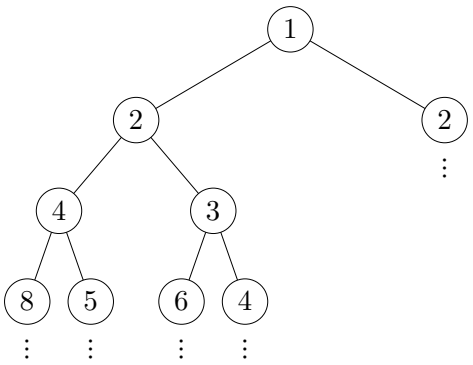
DS 1 d'option informatique - MP* - Sujet X/ENS 2017 - Correction

Question 1 – Voici une solution optimale :

1, 2, 4, 5, 10, 20, 21, 42

Pour vérifier que cette solution est bien optimale, on peut construire un arbre binaire complet où les noeuds à profondeur p sont étiquetés par tous les états accessibles en p coups :



On se rend compte qu'il faut au minimum 7 coups pour atteindre 42.

Question 2 – Soit $K \in \mathbb{N}^*$ le nombre de tours de boucle (le dernier tour peut éventuellement être incomplet à cause du retour anticipé) et numérotons ces tours par $k = 1, 2, \dots, K$. On remarque que $p = k - 1$ au début du tour et que $p = k$ à la fin du tour. Pour chaque $k \in [\![1, K]\!]$, notons A_k la valeur de l'ensemble A au début du tour k et montrons par itération finie que A_k est l'ensemble des états atteignables en $k-1$ coups.

Initialisation. Pour $k = 1$, $A_1 = \{e_0\}$ qui est bien l'ensemble des états atteignables avec 0 coup.

Hérédité. Soit $k \in [\![1, K-1]\!]$ et supposons que A_k contienne tous les états atteignables en $k-1$ coups. L'ensemble B construit lors du tour de boucle k contient tous les états atteignables en jouant un coup depuis un état de A_k. En d'autres termes A_{k+1} est bien l'ensemble de tous les états atteignables en k coups.

Montrons maintenant le résultat demandé dans l'énoncé par double implication.

($\Rightarrow$) Si la fonction renvoie VRAI, alors d'après le pseudo-code, il existe un élément de F qui appartient à A_K. Ainsi, cet élément peut être atteint en $K-1$ coups à partir de e_0. Donc une solution existe.

($\Leftarrow$) Si une solution existe, notons $p \in \mathbb{N}$ la profondeur minimale d'une solution. Remarquons d'abord que le tour de boucle numéro p existe : en effet si on avait $A_k = \varnothing$ pour un certain $k \leqslant p$, alors aucun état ne serait atteignable en p coups et donc e n'existerait pas ; de plus $A_k \cap F = \varnothing$ pour tout $k < p$. Finalement, au début du tour numéro p, on a $A_p \cap F \neq \varnothing$ et donc la fonction renvoie VRAI.

Question 3 – Pour cette question, on suppose que chaque ligne de l'algorithme s'exécute en temps constant, en particulier que $s(x) \cup B$ se calcule en temps constant.

$\star$ La principale difficulté est d'encadrer le cardinal de l'ensemble A à chaque étape. On reprend les notations K et A_k de la question précédente. De plus, on note P_k (resp. I_k) les éléments pairs (resp. impairs) de A_k. Pour tout $k \in [\![1, K-1]\!]$:

$$\begin{cases} P_{k+1} = \Big[\Big\{2n : n \in P_k\Big\} \uplus \Big\{2n : n \in I_k\Big\}\Big] \cup \Big\{n+1 : n \in I_k\Big\} \\ I_{k+1} = \Big\{n+1 : n \in P_k\Big\} \end{cases}$$

On a donc :

$$|P_1| = 0 \qquad |I_1| = 1 \qquad |P_2| = 1 \qquad |I_2| = 0$$

et pour tout $k \in [\![1, K-1]\!]$:

$$|P_k| + |I_k| \leqslant |P_{k+1}| \leqslant |P_k| + 2|I_k| \qquad |I_{k+1}| = |P_k|$$

On en déduit que :

$$|A_1| = |A_2| = 1 \qquad |P_1| = 0 \qquad |P_2| = 1 \qquad |P_3| \geqslant 1$$

et que pour tout $k \in [\![1, K-1]\!]$:

$$|A_{k+1}| = |P_{k+1}| + |I_{k+1}| \leqslant 2(|P_k| + |I_k|) = 2|A_k|$$

Par récurrence immédiate sur $k \geqslant 1$, on obtient $|A_k| \leqslant 2^{k-1}$. De plus, pour tout $k \in [\![2, K-1]\!]$:

$$|P_{k+1}| \geqslant |P_k| + |I_k| = |P_k| + |P_{k-1}|$$

Si on note $\phi = \dfrac{1+\sqrt{5}}{2}$ le nombre d'or ; alors en utilisant la relation $\phi^2 = \phi + 1$, on montre par récurrence immédiate sur $k \geqslant 2$ que $|P_k| \geqslant \phi^{k-3}$. Finalement :

$$\forall k \in [\![2, K]\!] : \phi^{k-3} \leqslant |A_k| \leqslant 2^{k-1}$$

⋆ Conformément à l'énoncé, supposons que le parcours en largeur de la figure 1 termine. D'après les inégalités ci-dessus, l'ensemble A ne peut pas être vide au début d'un tour de boucle et donc la fonction ne peut pas renvoyer FAUX, elle renvoie donc VRAI. Soit p la profondeur de la solution trouvée, alors le nombre de tours de boucle est $K = p + 1$ (le dernier tour étant incomplet).

Pour la complexité spatiale, on remarque que tous les ensembles construits sont de taille $\mathcal{O}(2^p)$ et que l'ensemble A_p construit lors de l'avant-dernier tour est de taille $\Omega(\phi^p)$.

La complexité spatiale est bien exponentielle en p.

Pour la complexité temporelle, on remarque que le tour de boucle k s'exécute en temps $\Theta(|A_k|)$. Ainsi, l'avant-dernier tour de boucle (le numéro p) s'exécute en temps $\Omega(\phi^p)$ et l'ensemble de l'algorithme s'exécute en temps :

$$\mathcal{O}\left(\sum_{k=1}^{p+1} 2^k\right) = \mathcal{O}(2^p)$$

Finalement :

La complexité temporelle est bien exponentielle en p.

Question 4 –

```
(* Variables globales pour le jeu (1) *)
let initial = 1;;
let suivants i = [2*i; i+1];;
let final i = (i = 42);;
```

```
(* ens_A et ens_B représentent les ensembles A et B de l'énoncé *)
let bfs() =
  let rec explorer ens_A ens_B = match ens_A with
    | [] -> 1 + explorer ens_B []
    | e :: q when final e -> 0
    | e :: q -> explorer q (suivants e @ ens_B) in
  explorer [initial] [];;
```

Question 5 – Fixons le triplet (E, e_0, s) représentant le jeu. Pour tout $e \in E$:

→ Si le jeu (E, e, s) n'a pas de solution, on pose $R(e) = +\infty$.

→ Sinon, $R(e) \in \mathbb{N}$ est la profondeur d'une solution optimale de (E, e, s).

Quels que soient $A \subset E$ et $B \subset E$, soit :

$$R(A, B) = \min\left[\left\{R(e) : e \in A\right\} \cup \left\{1 + R(e) : e \in B\right\}\right]$$

On munit $\mathbb{N}^2$ de l'ordre lexicographique. Montrons par induction sur $(k, \ell) \in \mathbb{N}^2$ la propriété :

$$\mathcal{P}(k,\ell) : \begin{cases}\text{Pour tout } A \subset E \text{ et } B \subset E, \text{ si } R(A,B) = k \text{ et } |A| = \ell, \text{ alors l'appel à}\\ \text{« } \texttt{explorer ens_A ens_B} \text{ » termine et renvoie } k.\end{cases}$$

Soit $(k, \ell) \in \mathbb{N}^2$, supposons $\mathcal{P}(k', \ell')$ pour tout $(k', \ell') < (k, \ell)$ et montrons $\mathcal{P}(k, \ell)$. Soient $A \subset E$, $B \subset E$ tels que $R(A, B) = k$ et $|A| = \ell$. Lors de l'exécution de « `explorer ens_A ens_B` », on se trouve dans l'un des 3 cas du filtrage de la fonction :

Cas 1 du filtrage. On a $A = \varnothing$, or :

$$R(\varnothing, B) = 1 + R(B, \varnothing)$$

Ainsi, si on pose $k' = k-1$ et $\ell' = |B|$, alors $(k', \ell') < (k, \ell)$ et l'hypothèse d'induction assure que « `explorer ens_B []` » renvoie k'. Finalement, l'appel à « `explorer ens_A ens_B` » renvoie $1 + k' = k$.

Cas 2 du filtrage. Le premier élément e de `ens_A` est un état gagnant donc :

$$R(e) = 0 \qquad \text{et} \qquad R(A, B) = 0$$

De plus, l'appel à « `explorer ens_A ens_B` » renvoie bien $0 = k$.

Cas 3 du filtrage. Le premier élément e de `ens_A` n'est pas un état gagnant. On a donc :

$$R(A, B) = R(A \setminus \{e\}, B \cup s(e))$$

Ainsi, si on pose $k' = k$ et $\ell' = \ell - 1$, alors $(k', \ell') < (k, \ell)$ et l'hypothèse d'induction assure que « `explorer q (suivants e @ ens_B)` » renvoie k'. Finalement, l'appel à « `explorer ens_A ens_B` » renvoie $k' = k$.

Conclusion. Supposons qu'une solution existe et notons $k \in \mathbb{N}$ la profondeur d'une solution optimale. On remarque que $k = R(e_0) = R(\{e_0\}, \varnothing)$. Ainsi, d'après $\mathcal{P}(k, 1)$, l'appel à « `explorer [initial] []` » renvoie k et donc la fonction `bfs` renvoie une profondeur optimale.

Question 6 – Montrons par récurrence sur $k = m - p \in \mathbb{Z}$ que $\mathrm{DFS}(m, e, p)$ renvoie VRAI si et seulement si une solution de profondeur inférieure ou égale à k existe pour le jeu (E, e, s).

Initialisation. Si $m - p < 0$, alors la fonction renvoie nécessairement FAUX (première condition du code). De plus, aucune solution de profondeur inférieure ou égale à $m - p$ ne peut exister. D'où le résultat.

Hérédité. Soit $k \in \mathbb{N}$. Supposons la propriété vraie pour $k - 1$ et montrons la pour k. Soient m, p tels que $m - p = k$, on a alors deux cas dans la fonction :

→ Si $e \in F$. Dans ce cas e est solution, c'est à dire que le jeu (E, e, s) admet une solution de profondeur 0 (qui est bien $\leqslant k$). De plus, la fonction renvoie VRAI, d'où le résultat.

→ Si $e \notin F$:

$$\begin{aligned}\mathrm{DFS}(m,e,p) = \text{VRAI} &\Leftrightarrow \exists x \in s(e) : \mathrm{DFS}(m, x, p+1) = \text{VRAI} && \text{(d'après le programme)}\\ &\Leftrightarrow \exists x \in s(e) : \begin{cases}\text{Une solution de profondeur inférieure ou égale à } m-p-1 \text{ existe}\\ \text{pour le jeu } (E, x, s).\end{cases} && \text{(Par hypothèse d'induction)}\\ &\Leftrightarrow \begin{cases}\text{Une solution de profondeur inférieure ou}\\ \text{égale à } m-p \text{ existe pour le jeu } (E, e, s).\end{cases}\end{aligned}$$

Conclusion. Lorsqu'on prend $p = 0$, on obtient :

> DFS$(m, e, 0)$ renvoie VRAI si et seulement si une solution de profondeur inférieure ou égale à m existe pour le jeu (E, e, s).

Question 7 –

```
(* Variables globales pour le jeu (1) *)
let initial = 1;;
let suivants i = [2*i; i+1];;
let final i = (i = 42);;
```

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

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

Question 8 – La fonction `dfs` est la traduction dans la syntaxe OCaml de la fonction DFS donnée dans l'énoncé. Ainsi, d'après la question 6, « `dfs m e p` » renvoie `true` si et seulement si une solution de profondeur inférieure ou égale à m existe.

Montrons la correction de `ids`. On remarque que cette fonction renvoie le plus petit entier m tel que DFS$(m, e_0, 0)$ renvoie VRAI. Supposons maintenant qu'une solution existe et notons k la profondeur d'une solution optimale. Par minimalité de k, pour tout $k' \in [\![0; k-1]\!]$, il n'existe pas de solution de profondeur inférieure ou égale à k'. Ainsi, d'après la question 6 :

→ $DFS(k', e_0, 0)$ renvoie FAUX pour tout $k' \in [\![0; k-1]\!]$.

→ $DFS(k, e_0, 0)$ renvoie VRAI.

> La fonction `ids` renvoie donc k qui est la profondeur optimale d'une solution.

Question 9 – Pour établir les complexités, on suppose qu'une solution existe et que la profondeur optimale vaut $p_0 \in \mathbb{N}$.

1. Commençons par le cas où il y a exactement un état à chaque profondeur p.

   Pour le parcours en largeur, à chaque instant de l'algorithme BFS, l'ensemble A est de taille 1 et l'ensemble B est de taille 0 ou 1. Ainsi, la complexité spatiale est en $\mathcal{O}(1)$. De plus, la boucle « tant que » fait $\mathcal{O}(p_0)$ tours et la boucle « pour tout » fait 1 tour. Au total, la complexité temporelle est en $\mathcal{O}(p_0)$.

   Pour la recherche itérée en profondeur, on remarque que la fonction DFS manipule uniquement des variables occupant une place $\mathcal{O}(1)$ en mémoire. De plus, lorsque $m \leqslant p_0$, l'arbre des appels récursifs de DFS$(m, e, 0)$ contient $m + 1$ noeuds et a pour hauteur m. Ainsi, DFS a pour complexité spatiale $\mathcal{O}(m)$ et pour complexité temporelle $\mathcal{O}(m)$. La fonction `ids` appelle DFS$(m, e, 0)$ pour $m = 0, 1, \dots p_0$. Ainsi, la complexité spatiale de `ids` est en $\mathcal{O}(m)$ et la complexité temporelle en :

$$\mathcal{O}\left(\sum_{m=0}^{p_0} m\right) = \mathcal{O}(p_0^2)$$

> BFS : $\begin{cases} \text{Spatiale} : \mathcal{O}(1) \\ \text{Temporelle} : \mathcal{O}(p_0) \end{cases}$ IDS : $\begin{cases} \text{Spatiale} : \mathcal{O}(p_0) \\ \text{Temporelle} : \mathcal{O}(p_0^2) \end{cases}$

2. Passons au cas où il y a exactement 2^p état à chaque profondeur p.

   Pour le parcours en largeur, si on numérote les tours de la boucle « tant que » par $p = 0, 1, \ldots$ alors l'ensemble A sera de taille 2^p au début du tour p et 2^{p+1} à la fin. De plus, l'ensemble B reste lui aussi de taille $\mathcal{O}(2^p)$. Ainsi, la complexité spatiale est en $\mathcal{O}(2^{p_0})$. À chaque tour de la boucle « tant que », on parcourt tous les éléments de A, soit une complexité temporelle en :

$$\mathcal{O}\left(\sum_{m=0}^{p_0} 2^m\right) = \mathcal{O}(2^{p_0})$$

   Lorsque $m \leqslant p_0$, l'arbre des appels récursifs de $\mathrm{DFS}(m, e, 0)$ a pour hauteur m et son nombre de noeuds est au plus :

$$\sum_{p=0}^{m} 2^p = 2^{m+1} - 1 = \mathcal{O}(2^m)$$

   Ainsi, la complexité spatiale de `ids` est $\mathcal{O}(m)$ et sa complexité temporelle est :

$$\mathcal{O}\left(\sum_{m=0}^{p_0} 2^m\right) = \mathcal{O}(2^{p_0})$$

BFS :		IDS :	
Spatiale : $\mathcal{O}(2^{p_0})$		Spatiale : $\mathcal{O}(p_0)$	
Temporelle : $\mathcal{O}(2^{p_0})$		Temporelle : $\mathcal{O}(2^{p_0})$	

Question 10 – La variable `min` sera de type `int` et on utilisera l'entier `-1` pour représenter ∞. Attention : lorsqu'il n'y a pas de solution, la fonction risque de partir dans une boucle infinie.

```
(* Variables globales pour le jeu (1) *)
let initial = 1;;
let suivants i = [2*i; i+1];;
let t = 42;;
let final i = (i = t);;
let inf = -1  (* Représente l'infini *)
let min = ref inf;;
```

```
(* Fonction heuristique de la question 11 *)
let h e =
  if e > t then 0 else
    let x = (float_of_int t) /. (float_of_int e) in
    int_of_float (ceil (log x /. log 2.));;
```

```
(* h est la fonction heuristique *)
let rec dfs_star m e p =
  let c = p + h e in
  if c > m then begin
      if !min = inf || c < !min then min := c;
      false
    end
  else if final e then true
  else if List.exists (fun x -> dfs_star m x (p+1)) (suivants e) then true
  else false;;
```

```
(* Boucle infinie si pas de solution *)
let idastar () =
  let rec chercher m =
    print_int m; print_newline();
    if m = inf then inf else begin
        min := inf;
        if dfs_star m initial 0 then m else chercher !min
      end in
  chercher (h initial);;
```

Question 11 – Pour le jeu (1), on pose :

$$\boxed{h : \begin{cases} \mathbb{N}^* \to \mathbb{N} & \\ e \mapsto \left\lceil \log_2\left(\frac{t}{e}\right) \right\rceil & \text{si } e \leqslant t \\ e \mapsto 0 & \text{si } e > t \end{cases}}$$

Montrons que h est admissible. Pour tout couple d'états $(n, m) \in E^2$, on note $\text{dist}(n, m)$ la distance entre n et m (si elle existe). Par récurrence immédiate sur $\text{dist}(n, m) \in \mathbb{N}$, on a :

$$m \leqslant n \times 2^{\text{dist}(n,m)}.$$

En inversant cette formule, on obtient :

$$\left\lceil \log_2\left(\frac{m}{n}\right) \right\rceil \leqslant \text{dist}(n, m).$$

On peut maintenant montrer que h est admissible pour le jeu (1). Pour tout $e \in E$:

→ Si $e > t$, alors il n'existe pas de chemin entre e et t et donc $\text{dist}(e, t)$ n'est pas définie. Dans ce cas, l'énoncé précise que $h(e)$ peut être quelconque.

→ Si $e \leqslant t$, alors d'après ce qui précède, $h(e) \leqslant \text{dist}(e, t)$.

Dans les deux cas, il n'existe pas d'état $f \in F$ situé à une distance de e strictement inférieure à $h(e)$.

$\boxed{h \text{ est admissible}}$

Remarque. Dans le cas du jeu (1), on peut éviter que la fonction de la question 10 ne parte dans une boucle infinie lorsqu'il n'y a pas de solution en autorisant h à renvoyer $+\infty$. Il suffit de reprendre la fonction h donnée ci-dessus en remplaçant $h(e) = 0$ par $h(e) = +\infty$ dans le cas où $e > t$.

Question 12 – On remarque que (voir les justifications ci-dessous) :

(i) Dans la fonction IDA* la variable m augmente strictement entre deux tours de boucle.

(ii) Si une solution existe alors à chaque tour de boucle de IDA*, on a $m \leqslant p_0$ où p_0 est la profondeur d'une solution optimale.

$\star$ Montrons (i). Dans la fonction DFS*, si la ligne « $min \leftarrow c$ » s'exécute alors $c > m$. Ainsi, juste avant l'exécution de « $m \leftarrow min$ », on a nécessairement $min > m$.

$\star$ Montrons (ii). Au premier tour de boucle, on a $m = h(e_0) \leqslant p_0$ car h est admissible. Il suffit donc de montrer que $m \leqslant p_0$ durant les autres tours de boucle. En d'autres termes, on doit montrer que juste avant l'exécution de la ligne « $m \leftarrow min$ », on a $min \leqslant p_0$.

Soit $(e_0, e_1 \ldots, e_{p_0})$ une solution optimale du jeu, alors l'appel à DFS$^\star(m, e_0, 0)$ va appeler récursivement DFS$^\star(m, e_1, 1)$, DFS$^\star(m, e_2, 2)$, …, DFS$^\star(m, e_i, i)$ jusqu'à un certain $i \leqslant p_0$. D'après le code de l'appel à DFS$^\star(m, e_i, i)$, on a trois cas :

→ La condition « $c > m$ » est vérifiée. Dans ce cas, la fonction renvoie FAUX et on a $min \leqslant i + h(e_i)$. Or $i + h(e_i) \leqslant p_0$ car h est admissible.

→ La condition « $e \in F$ » est vérifiée. Dans ce cas, la fonction renvoie VRAI ce qui signifie que la ligne « $m \leftarrow min$ » ne sera pas exécutée.

→ Sinon, un appel à DFS$^\star(m, e_{i+1}, i+1)$ est fait ce qui contredit la maximalité de i.

En résumé, dans la fonction IDA*, si on ne se trouve pas au dernier tour de boucle, alors juste avant l'exécution de la ligne « $m \leftarrow min$ », on a $min \leqslant p_0$.

$\star$ En combinant (i) et (ii), on en déduit que si une solution existe, alors l'appel à IDA* termine et renvoie un entier m vérifiant $m \leqslant p_0$. De plus, si DFS* renvoie VRAI, c'est qu'il a trouvé une solution vérifiant $p \leqslant m$. La seule possibilité est donc $p = m = p_0$ et `idastar` renvoie toujours une profondeur optimale.

Question 13 – Pour représenter un état du taquin, on peut utiliser une matrice de taille 4×4 dont chaque case contient un nombre entre 0 et 15 où 15 représente une case libre. Pour représenter chaque nombre de $[\![0, 15]\!]$, il faut au minimum 4 bits. Au total il faut au moins $4 \times 16 = 64$ bits pour représenter un état du taquin.

Chaque état correspond à une permutation des éléments de l'état final, soit 16! états possibles. Au total, pour stocker simultanément tous les états, le nombre de bits nécessaires serait de l'ordre de :

$$\begin{aligned} 64 \times 16! &= 2^{21} \times 3^6 \times 5^3 \times 7^2 \times 11 \times 13 \\ &= 1024 \times 256 \times 729 \times 10^3 \times 7007 \\ &\geqslant 10^3 \times 2.10^2 \times 7.10^2 \times 10^3 \times 7.10^3 \\ &= 98 \times 10^{13} \\ &\approx 10^{15} \end{aligned}$$

Soit plus de 1 péta-bits de données. Pour rappel, la mémoire vive d'un ordinateur de bureau peut contenir quelques giga-octets et le disque dur quelques tera-octets.

Même si tous les états ne sont pas stockés en même temps lors du parcours en largeur, la première phrase de la page 6 indique qu'il faut au minimum 50 déplacements pour résoudre le taquin donné au début de la partie IV. Pour chaque configuration, il y a 2, 3 ou 4 déplacements possibles. Ainsi, si on utilise la fonction `bfs` de la question 4 dans laquelle les doublons ne sont pas supprimés, alors au début du dernier tour de boucle, la liste `ens_A` contiendra au moins 2^{50} configurations, c'est à dire plus de 64 péta-bits de données.

Il n'est pas réaliste d'envisager le parcours en largeur de la figure 1 pour chercher une solution optimale du taquin.

Remarque. En réalité, la moitié des 16! permutations ne sont pas accessibles depuis l'état gagnant (voir wikipédia).

Remarque. Le rapport du jury indique que « l'exemple d'un chemin optimal de longueur 50 donné dans l'énoncé doit être utilisé pour justifier un nombre trop grand d'états visités par le parcours en largeur ».

Question 14 – On remarque que :

→ Si t représente l'état final du taquin alors $h(t) = 0$. En effet, pour tout $v \in [\![0; 14]\!]$:

$$t_v^i = \lfloor v/4 \rfloor \qquad \text{et} \qquad t_v^j = v \bmod 4$$

→ S'il est possible de passer d'un état e à un état f avec un seul déplacement alors $h(f) = h(e) - 1$ ou $h(f) = h(e) + 1$. En effet, il existe $(i, j) \in [\![0; 3]\!]^2$ tel que passer de e à f consiste à déplacer le contenu v_0 de la case de coordonnées (i, j) vers la droite, la gauche, le haut ou le bas. Dans le premier cas (les trois autres se traitent de manière similaire), :

$$\forall v : e_v^i = f_v^i \qquad \forall v \neq v_0 : e_v^j = f_v^j \qquad f_{v_0}^j = e_{v_0}^j + 1$$

et donc $h(f) \in \{h(e) - 1, h(e) + 1\}$.

Pour conclure, si on note t l'état final du taquin, une récurrence immédiate sur $\text{dist}(e, t)$ montre $\text{dist}(e, t) \leqslant h(e)$ pour tout e.

h est admissible.

Partie IV

```
(* Variables globales évoquées dans l'énoncé *)
type deplacement = Gauche | Bas | Droite | Haut;;
let solution: deplacement list ref = ref [];;
let inf = -1  (* Représente l'infini *)
let min = ref inf;;
```

```
(* Grille exemple de l'énoncé *)
let grid = [| [|  2;  3;  1;  6|];
              [| 14;  5;  8;  4|];
              [| -1; 12;  7;  9|];
              [| 10; 13; 11;  0|] |];;
```

```
(* Initialisation de li et lj à partir de grid.
   Suppose que la case vide contient -1 *)
let li = ref (-1) and lj = ref (-1);;
let n = Array.length grid in
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        if grid.(i).(j) = -1 then begin
            li := i;
            lj := j;
          end
      done;
    done;;
```

```
(* Initialisation de h à partir de grid.
   Suppose que li et lj sont initialisés *)
let h  = ref (-1);;
let n = Array.length grid in
    h := 0;
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        if i <> !li || j <> !lj then begin
            let v = grid.(i).(j) in
            h := !h + abs (i - v/n);
            h := !h + abs (j - (v mod n));
          end
      done;
    done;;
```

Question 15 –

```
let move (i: int) (j: int): unit =
  let n = Array.length grid in
  h := !h - abs (i - grid.(i).(j)/4) - abs (j - (grid.(i).(j) mod n));
  h := !h + abs (!li - grid.(i).(j)/4) + abs (!lj - (grid.(i).(j) mod n));
  grid.(!li).(!lj) <- grid.(i).(j);
  grid.(i).(j) <- -1;
  li := i;
  lj := j;;
```

```
(* Fonctions données dans l'énoncé *)
let haut   () = move (!li + 1) !lj;;
let gauche () = move !li (!lj + 1);;
let bas    () = move (!li - 1) !lj;;
let droite () = move !li (!lj - 1);;
```

Question 16 –

```
let tente_gauche (): bool =
  let n = Array.length grid in
  match !solution with
  | Droite :: _ -> false
  | _ when !lj = n-1 -> false
  | _ -> gauche();
         solution := Gauche :: !solution;
         true;;
```

```
(* Fonction non demandée par l'énoncé *)
let tente_droite (): bool =
  (* let n = Array.length grid in *)
  match !solution with
  | Gauche :: _ -> false
  | _ when !lj = 0 -> false
  | _ -> droite();
         solution := Droite :: !solution;
         true;;
```

```
(* Fonction non demandée par l'énoncé *)
let tente_haut (): bool =
  let n = Array.length grid in
  match !solution with
  | Bas :: _ -> false
  | _ when !li = n-1 -> false
  | _ -> haut();
         solution := Haut :: !solution;
         true;;
```

```
(* Fonction non demandée par l'énoncé *)
let tente_bas (): bool =
  (* let n = Array.length grid in *)
  match !solution with
  | Haut :: _ -> false
  | _ when !li = 0 -> false
  | _ -> bas();
         solution := Bas :: !solution;
         true;;
```

Question 17 –

```
(* Annule le dernier coup. *)
let annuler_coup() = match !solution with
  | [] -> failwith "annuler_coup: aucun coup n'a été joué"
  | c :: q -> solution := q;
              match c with
              | Gauche -> droite()
              | Droite -> gauche()
              | Bas -> haut()
              | Haut -> bas();;
```

```
(* On écrit 2 fonctions récursives mutuelles.
   - Dans "tente": f = tente_gauche ou tente_droite ou tente_bas ou tente_haut
   - Dans dfs: donner e en entrée n'est pas nécessaire. *)

let rec tente m p (f: unit -> bool): bool =
  if f() then
    if dfs m (p+1) then true
    else (annuler_coup(); false)
  else false

and dfs m p =
  let c = p + !h in
  if c > m then begin
      if !min = inf || c < !min then min := c;
      false
    end
  else if !h = 0 then true
  else tente m p tente_gauche || tente m p tente_droite ||
         tente m p tente_bas || tente m p tente_haut;;
```

Question 18 –

```
let taquin () =
  solution := [];
  let rec chercher m =
    print_int m; print_newline();
    if m = inf then failwith "Taquin: pas de solution";
    min := inf;
    if dfs m 0 then !solution else chercher !min in
  chercher !h;;
```