Sujet X-ENS 2017 : correction

Question 1
On passe de 1 à 2, puis à 4, puis à 5, puis à 10, puis 20, 21 et 42. Cette solution a profondeur 7.
Question 2
On montre par récurrence sur k que lorsque la variable p a valeur k, l'ensemble A contient toutes les valeurs à profondeur p. En effet, le résultat est vrai avant la boucle.
Supposons le au début de la k-ème boucle. Lors d'un nouveau passage de boucle, soit on trouve un élément dans F et l'algorithme s'arrête, soit B va contenir toutes les valeurs que l'on peut obtenir à partir d'une valeur de A. Comme les éléments de A sont ceux de profondeur $p = k$, B contiendra les éléments de profondeur $k + 1$. A la fin de ce passage de boucle, p devient $p + 1 = k + 1$ et A contient toutes les valeurs de profondeur $k + 1$ qui est la nouvelle valeur de p.
Ainsi, si le jeu possède une solution x de profondeur k, à la boucle k, on renverra VRAI dans l'énumération des éléments de A. Si le jeu ne possède pas de solutions, pour tout k, il n'y aura aucun élément de F à profondeur k, donc on ne renverra jamais la valeur VRAI.
Question 3
Notons a_k le nombre de valeurs de A au passage k de boucle. Au passage de boucle k, on parcourt l'ensemble A en faisant à chaque fois une comparaison et une affectation de B. En plus de cela, on fait 3 opérations. Le nombre d'opérations effectué à la boucle k est $2a_k + 3 = \Theta(a_k)$.
De même, l'espace en mémoire est en $\Theta(a_k)$.
A chaque valeur x de A à l'instant k correspond au plus deux valeurs à l'instants $k + 1$ que sont $x + 1$ et $2x$. Au plus, le nombre de valeurs de A à l'instant $k + 1$ est le double de celui à l'instant k. On a donc pour tout k, $a_{k+1} \leq 2a_k$. On montre alors par récurrence que $a_k \leq a_0 2^k$, c'est-à-dire $a_k = O(2^k)$.
De plus, pour chaque valeur de A à la boucle k, il y a au moins une valeur associé à celle-ci dans la boucle $k+1$ par l'opération $x \mapsto x + 1$. En outre, pour tout $k \geq 1$, il y a au moins $\frac{a_k}{2}$ valeurs pairs (celle obtenus par l'opération $x \mapsto 2x$ à la boucle précédente). Dans la boucle $k+1$, ces valeurs pairs donneront toutes une valeur impair par l'opération $x \mapsto x+1$. On a aussi au moins a_k valeurs pairs dans A à l'instant $k + 1$ données par $x \mapsto 2x$. Dans la boucle $k + 1$, A aura au moins a_k valeurs pairs et $\frac{a_k}{2}$ valeurs impaires. On a donc $a_{k+1} \geq \frac{3a_k}{2}$. Ainsi, $a_k \geq a_0(\frac{3}{2})^k$.
Pour aller jusqu'à la profondeur p, on aura donc une complexité (spatiale et temporelle) au plus de $\sum_{k=1}^{p} \Theta(2^k) = \Theta(\sum_{k=1}^{p} 2^k) = \Theta(2^{p+1} - 1) = \Theta(2^p)$ et au moins de $\sum_{k=1}^{p} \Theta((\frac{3}{2})^k) = \Theta((\frac{3}{2})^p)$.
Par conséquent, pour une solution de profondeur p, la complexité spatiale et temporelle de "bfs" est majorée par une fonction en $\Theta(2^p)$ et minorée par une fonction en $\Theta((\frac{3}{2})^p)$.
Question 4 Voici deux versions : la première récursive, la seconde itérative :

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

```
let bfs =
   let A=ref [initial] and p=ref 0;
   while !A<>[] do
      let B = ref [] in
      while !A<>[] do
         let x::q=!A in
         if final(x) then begin
                                !A=[]
                                !B=[]
                                end
         else B:=suivants(x)@(!B)
         A:=q
         done;
```

```
      A:=!B;
      p:=!p+1
      done;
   !p;;
```

Question 5

Nous avons déjà montré qu'au passage de boucle k, la variable A contient les valeurs à profondeur p. S'il y a une solution de profondeur p, avec p minimal, alors BFS ne pourra renvoyer une réponse avant ce p-ème passage de boucle. Elle renverra bien une solution de profondeur p au p-ème passage.

Question 6

Il existe une solution de profondeur inférieure à p en partant de e_0 si et seulement si e_0 est solution ou il existe une solution de profondeur inférieure à $p-1$ en partant d'un voisin de e_0.

De plus, $DFS(m, e, p)$ est vrai si et seulement si $p \leq m$ et e est dans F (c'est-à-dire est solution) ou un voisin x de e vérifie $DFS(m, x, p+1)$.

On en déduit, par induction, que $DFS(m, e, p)$ est vrai si et seulement si il existe une solution à partir de e en au plus $m-p$ étapes (c'est-à-dire à profondeur au plus $m-p$).

En effet, le résultat est vrai si $p < m$ (car DFS renvoie faux et $m - p < 0$). Supposons ce résultat pour p fixé. Alors $DFS(m, e, p)$ équivaut à $p \leq m$ et e est dans F ou un voisin x de vérifie $DFS(m, x, p+1)$; ceci, d'après l'hypothèse d'induction, équivaut à $p \leq m$ et e est dans F ou il existe un voisin x de e à profondeur $h \leq m-p-1$ solution ; dans ce deuxième cas, e est à profondeur $h' = h+1 \leq m-p$. Ainsi $DFS(m, e, p)$ équivaut à ce qu'il existe une solution de profondeur inférieure à $m-p$ à partir de e.

Par induction, $DFS(m, e_0, p)$ renvoie vrai si et seulement si il existe une solution à partir de e_0 de profondeur au plus $m-p$. En particulier, $DFS(m, e_0, 0)$ est vrai si et seulement si il existe une solution à partir de e_0 de profondeur au plus m.

Question 7

```
let rec dfs m p l = match l with
   | _ when p>m -> false
   | [] -> false
   | x::q when final(x) -> true
   | x::q -> (dfs m (p+1) suivants(x) )||( dfs m p q);;

let ids =
   let m=ref 0 in
   while not(dfs !m 0 [initial]) do
      m:=!m+1
      done;
   !m;;
```

Question 8

La fonction "dfs" renvoie un entier naturel p si et seulement si elle se termine ; d'après la question précédente, elle se termine si et seulement si il existe une solution à partir de e de profondeur au plus $m-p$. L'entier p est alors la profondeur de la solution par rapport à e.

La boucle de "ids" s'arrête dès que l'on trouve une valeur m pour laquelle "dfs" renvoie une valeur positive. S'il y a une solution de profondeur optimale h, la fonction "dfs" ne renverra pas de réponse positive pour $m < h$ et renverra une réponse positive pour la valeur $m = h$. Cette valeur sera la profondeur p (à savoir h) recherchée.

Question 9

Pour un parcours en largeur, la complexité spatiale comme temporelle à l'étape de la boucle k est proportionnelle au nombre d'état à profondeur k.

Dans le premier cas, cela donne $\Theta(1)$ comme complexité temporelle à chaque passage de boucle, soit au total $\Theta(p)$ pour une solution à profondeur p. La complexité spatiale (l'espace pris par la liste A principalement) est $\Theta(1)$.

Dans le second cas, on a une complexité temporelle à chaque passage de boucle de $\Theta(2^k)$, soit au total $\sum_{k=1}^{p} \Theta(2^k) = \Theta(2^p)$. La complexité spatiale est proportionnelle à la taille de A qui est au maximum $\Theta(2^p)$.

Pour un parcours en profondeur itéré (ou itérée), pour aller jusqu'à la profondeur k dans le premier cas il faut $\sum_{j=1}^{k} \Theta(1) = \Theta(k)$ opérations. Si la première solution se trouve à profondeur p, il faut faire le parcours pour $k = 1$, $k = 2, \cdots$, $k = p$, soit au total $\sum_{k=1}^{p} \Theta(k) = \Theta(p^2)$ opérations. Pour la complexité temporelle, il garder en mémoire les appels récursifs effectués, soit $\Theta(k)$ pour aller à la profondeur k. L'espace total utilisé ne dépasse donc pas $\Theta(p)$ pour une solution optimale à la profondeur p.

Dans le second cas, pour aller à profondeur k, il faut $\sum_{j=1}^{k} \Theta(2^j) = \Theta(2^k)$ opérations. La complexité temporelle pour une solution optimale à profondeur p sera donc en $\sum_{k=1}^{p} \Theta(2^k) = \Theta(2^p)$. La complexité spatiale pour aller à la profondeur k est

$\Theta(k)$. Pour une solution optimale à profondeur p, elle est donc de $\Theta(p)$.
Moralité : pour un nombre limité d'états, le parcours en largeur est plus efficace, notamment en temps ; pour un nombre très grand d'états, la mémoire utilisée par le parcours en largeur pourrait être limitante, ce qui ne sera pas le cas pour un parcours en profondeur itéré.
Question 10

```
let rec dfsstar m p l =
   let c = p + h(e) in match l with
   | _ when c>m ->
   if (c<!min || !min=-1) then min:=c; false
   | [] -> false
   | x::q when final(x) -> true
   | x::q -> (dfsstar m (p+1) suivants(x) )||( dfsstar m p q);;

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

Question 11
Prenons $h(n) = \lfloor \log_2(t/n) \rfloor$.
Le plus grand entier que l'on plus atteindre en m coups depuis n est $2^m n$. Si $F = \{t\}$, comme $2^{h(n)}n \leq \frac{t}{n}n = t$, le plus grand nombre que l'on puisse atteindre depuis n en $h(n)$ coups est t. En un coup de moins, on ne peut pas dépasser $t/2$. Par conséquent, la fonction h est bien admissible.
Question 12
Comme pour la précédente fonction DFS, on montre que $DFS*(m, e, p)$ est vrai si et seulement si il existe une solution à profondeur au plus $m - p$ (en plus, il s'arrête si $m - p < h(e)$ puisqu'il ne peut y avoir de telles solutions) ; en outre, $DFS*(m, e, p)$ modifie min en la valeur minimale de $p + h(e)$ pour les différents états e parcourus.
Les valeurs prises par m augmentent (m prend la valeur "min", sachant que "min" prend la plus petite des valeurs de $h(e) + p$).
Si l'algorithme $IDA*$ renvoie VRAI, c'est que le dernier appel à $DFS*(m, e_0)$ a renvoyé VRAI et que les précédents ont renvoyé FAUX. Notons m_1 et m_2 les deux dernières valeurs de m. Comme les valeurs prises par m augmentent et que l'avant-dernier appel à $DFS*$ a renvoyé faux, il n'y avait pas de de solution à profondeur m_1 ; en outre, on a trouvé une solution à profondeur m_2 puisque $DFS*(m_2, e0, 0)$ a renvoyé VRAI. Supposons qu'il existe une solution à profondeur $m_1 < m \leq m_2$. Alors, en considérant un chemin de longueur m menant à cette solution, le m_1-ème sommet e est à distance $m - m_1$ d'une solution. On a donc $h(e) \leq m - m_1$ donc $m_1 + h(e) \leq m \leq m_2$. Or m_2 est la valeur minimale prise par les divers $p + h(e)$. On a donc $m = m_2$. m_2 est donc bien la profondeur optimale.
Si l'algorithme $IDA*$ renvoie FAUX, soit m_1 la dernière valeur non infinie prise par m. Le dernier appel à $DFS*$ n'a pas modifié min et n'a donc pas trouvé d'état e de profondeur p tel que tel que $p + h(e) > m_1$. Ainsi, tous les états visités e de profondeur p vérifient $p + h(e) \leq m_1$. Les profondeurs possibles p sont donc bornées, on ne peut donc pas trouver de nouvel état à profondeur supérieure à la précédente. Il n'y a donc pas de solution si l'on n'a pu en trouver jusque là.
Question 13
Représenter un état demande un tableau de taille 16 ou une matrice de taille 4×4. Il y a 16! permutations de $[\![1, 16]\!]$, ce qui fait plus que $16! = 32 \times 30 \times 56 \times 65 \times 72 \times 77 \times 80 \times 9 = (32 \times 77) \times (30 \times 80) \times (56 \times 9) \times (65 \times 72) \geq 2400 \times 2400 \times 500 \times 4500 = 24^2 \times 5 \times 10^8 \geq 2 \times 10^{11}$. Notons toutefois que toutes les représentations du jeu ne sont pas accessibles ; cependant, il devrait y avoir environ cet ordre de grandeur d'états possibles (partons du principe que le but de l'énoncé n'est pas de déterminer quelles sont les permutations accessibles...).
Stocker un entier demande de l'ordre d'un octet. Représenter toutes les représentations possibles du jeu de taquin demande de l'ordre de $10^{11} \times 16$ octets, soit plus de 10^{12} octets ce qui est davantage que la mémoire de la RAM d'un ordinateur (qui est de l'ordre de 10^9 octets). L'espace mémoire demandé sera de l'ordre de ce nombre d'états (cf. question 9). A l'inverse, le parcours en profondeur itéré est beaucoup moins "gourmand" en espace mémoire.
Question 14
On remarque que pour tout chaque entier v l'entier $|e_v^i - \lfloor v/4 \rfloor| + |e_v^j - (v \mod 4)|$ est la distance séparant l'entier v de sa position finale.
Au cours d'un déplacement, cette distance est modifiée de 1 (une seule pièce est déplacée). La position finale est la seule pour laquelle $h(e) = 0$.
Ainsi, pour arriver à la position finale, il faut au moins $h(e)$ coups.
Question 15

```
let move i j =
   let v = grid.(i).(j) in grid.(!li).(!lj) <- v;
   li:=i;
   lj:=j;
   h:=0;
   h:=!h-abs(i-v/4)-abs(j-v mod 4)+abs(!li-v/4)+abs(!lj-v mod 4);;
```

Question 16

```
let tente_gauche = match (!lj,!solution) with
   | 3,_ ->false
   | _,Droite::_ -> false
   | _,_ ->move (!li) (!lj +1);
              solution:=Gauche:: !solution ;
              true;;
```

Question 17
On crée une fonction "annule" qui permet de revenir en arrière dans nos déplacements si nécessaire.

```
let annule = let x::q=!solution in
    match x with
    | Gauche -> move (!li) (!lj-1)
    | Droite -> move (!li) (!lj+1)
    | Haut -> move (!li-1) (!lj)
    | Bas -> move (!li+1) (!lj);
    solution:=q;
    false;;

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

Question 18
On suppose donnée une valeur initiale de h et une grille initiale.

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