DM13 (éléments de réponse)

D'après un sujet de X2017

Parcours en largeur

Question 1. On peut lister les nombres qu'on peut atteindre avec une profondeur donnée.

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

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

Question 2.

Algorithme 1 : parcours en largeur

```
1  fonction BFS()
2      A ← {e_0}
3      p ← 0
4      tant que A ≠ 0 faire
5          B ← 0
6          pour x ∈ A faire
7              si x ∈ F alors
8                  renvoyer VRAI
9              B ← s(x) ∪ B
10         A ← B
11         p ← p + 1
12         renvoyer FAUX
```

Question 3. 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 $e_0 \ldots e_p$ 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 $s\left(e_k\right)$ 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 4. 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 $O\left(4^n\right)$.

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 $O\left((3/2)^n\right)$.

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 OCaml.

```
initial : etat
suivants : etat -> etat list
final : etat -> bool
```

Question 5.

```
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()
- : init  = 7
```

Question 6. 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.

Parcours en profondeur

Question 7. 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 $DFS\left(m, e_{p-1}, p-1\right)$ 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 8.

```
let rec dfs m e p =
  p <= m && (final e || tester_succ (suivants e) m p)
and tester_succ lst m p = match lst 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
;;
```

`dfs` réalise la fonction dfs directement, avec les mêmes arguments. `tester_succ lst 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 9. Supposons qu'une solution existe et soit m la profondeur optimale. Alors l'algorithme ids va tester DFS $(k, e_0, 0)$ pour $k < m$ et ne trouvera pas de solution (par optimalité), puis testera DFS $(m, e_0, 0)$ et en trouvera une, par simple application de la question 6 . La valeur trouvée est donc bien optimale.

Question 10.

- 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 $O\left(m^2\right)$ à la fin.
- 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 $O\left(2^m\right)$ pour la dernière étape. Le temps de calcul sera également en $O\left((2+\varepsilon)^m\right)$ (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 $O\left((2+\varepsilon)^m\right)$.

Parcours en profondeur avec horizon

Question 11.

```
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 12. 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 quatrième partie, 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 $\left\lfloor \log_2(t/n) \right\rfloor$ coups, et il suffit donc de donner cette valeur à $h(n)$.

Question 13. 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'avantdernier 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.

Application au jeu du taquin

Question 14. Si on suppose que toutes les configurations sont accessibles, il y a 16! configurations possibles au taquin. On peut raisonnablement minorer par 10^12 ce nombre. S'il faut le démontrer, écrivons que $16! = 2^{15} \cdot 3^6 \cdot 5^3 \cdot 7^2 \cdot 11 \cdot 13 > 2^{10} \cdot (2 \cdot 5)^3 \left(2^2 \cdot 27\right) \cdot \left(3^3 \cdot 49\right) \cdot 11 \cdot 13. > 10^3 \cdot 10^3 \cdot 10^2 \cdot 10^3 \cdot 10 \cdot 10 > 10^{13}$. 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 :

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

Question 15. Pour que cette fonction h s'annule, il faudrait avoir pour chaque v, $e_v^i = \lfloor v/4 \rfloor$ et $e_v^j = v \bmod 4$. 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 array array
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 16.

```
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 valeur 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 17.

```
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 18. 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 19. 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)
;;
```