DM1 - Informatique Option

Arsène MALLET - MP*

Question 1

Une solution optimale pour $F = \{42\}$ est $\{1, 2, 4, 5, 10, 20, 21, 42\}$. Cette solution est de profondeur 7. (pas plus petit puisque en binaire $42 = \underline{101010}_2$)

Question 2

On suppose $s \in \mathbb{N}^* \cup \{+\infty\}$ (le cas où $s = 0$ étant triviale) la profondeur de la solution optimale (en notant $s = +\infty$ si il n'y a pas de solution au jeu). Afin de montrer l'équivalence, il faut montrer qu'à chaque p-ième tour de boucle `tant que` , A contient tous les états de profondeur p (où $p < s \in \mathbb{N}$). Montrons le par récurrence :
Initialisation : Si $p = 0$ alors $A = \{e_0\}$, et donc contient bien l'unique état de profondeur 0. L'initialisation est vérifiée.
Hérédité : Soit $p \in \mathbb{N}$ tel que $p < s$, on suppose que A contienne tous les états de profondeur p en sortant dans le p-ième tour de la boucle `tant que`, montrons que A contient tous les états de profondeur $p + 1$ en entrant dans le $(p + 1)$-ième tour de la boucle `tant que`.

Comme $p < s$, $A \cap F = \varnothing$. Ainsi, grâce à la boucle `pour tout`, $B = \{s(x) \mid x \in A\}$, or comme A contient tous les états de profondeur p (*Hypothèse de récurrence*), alors B contient tout les états de profondeur $p + 1$. Comme en fin de bouble `tant que`, $A \leftarrow B$, en entrant dans le $(p + 1)$-ième tour de la boucle `tant que`, A contient tous les états de profondeur $p + 1$

Conclusion : à chaque p-ième tour de boucle `tant que` , A contient tous les états de profondeur p (où $p < s \in \mathbb{N}$)

Ainsi, le parcours en largeur renvoie `VRAI` si et seulement si il existe une profondeur $p' \in \mathbb{N}$ tel que $A_{p'} \cap F \neq \varnothing$, si et seulement si il existe une solution.

Question 3

Soit $p \in \mathbb{N}$, la profondeur de la solution trouvée.
On considère le nombre d'états total que l'on ajoute à A au cours des itérations de boucle comme étant l'ordre de la compléxité temporelle (puisque le nombre d'opérations élémentaires dépend de ce nombre total d'états).
Ainsi on ajoute enfaite à A tous les états atteignables depuis e_0. En considérant les potentiels doublons, et sachant qu'à chaque état de profondeur $i < p \in \mathbb{N}$, il y a deux choix d'états de profondeur $i + 1$ alors le nombre total d'états ajouté à A est majoré par:

$$\begin{aligned}\sum_{i=0}^{p}(2^i) &= 2^{p+1} - 1 \\ &< 2^{p+1}\end{aligned}$$

Ainsi la complexité temporelle est majorée par 2^{p+1}

On cherche maintenant un minorant du nombre d'états ajouté à A. Pour cela on peut montrer que pour tout $i \in \mathbb{N}$, on ajoute tout les entiers entre 1 et 2^i en $2i$ itérations de boucle :

Soit i $\in \mathbb{N}$ et $n \in [\![1, 2^i]\!]$. On note $\underline{b_k b_{k-1} ... b_0}_2$ où $k \leq i, b_k = 1$ et $\forall l \in [\![0, k-1]\!], b_l \in \{0, 1\}$ la représentation binaire de n. Ainsi,

$$n = \sum_{l=0}^{k} (b_l 2^l) = b_0 + 2(b_1 + 2(... + 2b_k)$$

et, comme $\forall l \in [\![0, k-1]\!], b_l \in \{0, 1\}$, n est atteignable en maximum $2k$ étapes (en partant de la dernière parenthèse, on multiplie par deux et/ou on ajoute 1 au fur et à mesure). Ainsi en p itérations de boucle, on ajoute au minimum l'ensemble $[\![1, 2^{\lfloor p/2 \rfloor}]\!]$ à A, d'où le minorant. Ainsi la complexité temporelle est bornée par deux fonctions exponentielles en p.

La complexité spatiale est de l'ordre du cardinal maximal de A au cours des itérations, qui est majoré par le nombre d'états de profondeur p, lui-même majoré par 2^p (séquence de p éléments et 2 choix par états). De plus, comme on ajoute au moins $2^{\lfloor p/2 \rfloor}$ éléments à A en p étapes (cf. complexité temporelle), il y a au moins une itération de boucle où $Card(A) \geq \frac{2^{\lfloor p/2 \rfloor}}{p}$ éléments. Ainsi la complexité spatiale est également bornée par deux fonctions exponentielles en p.

Question 4

```
let bfs () =
  let rec dissimulateWhile a b p=
    match a with
    |[] -> (match b with
          |[] -> -1
          |_ -> dissimulatewhile b [] p+1)
    |h::t -> if final h then p else dissimulateWhile t ((suivants(h)) b) @ p
in dissimulateWhile [initial] [] 0 ;;
```

Question 5

Comme montré à la question 2, l'ensemble A contient tous les états possibles de profondeur p en sortant dans la p-ième itération de boucle. Ici la p-ième itération correspond à un appel à `dissimulateWhile` de second argument `[]`et de troisième arguement `p`. En effet ce troisième indice est incrémenté de 1 à chaque fois que a est complétement vidé et que $a \leftarrow b$. Ainsi tous les états de prodondeur p sont vérifiés avant de tester les états de profondeur $p+1$, ainsi `bfs` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Question 6

Si une solution $e = e_0\ e_1\ ...\ e_p$ de profondeur $p \leq m$ existe , alors `DFS`(m, e_p, p) renvoie `VRAI` donc `DFS`(m, e_{p-1}, $p-1$) renvoie `VRAI` également, ainsi par récurrence, `DFS`(m, e_0, $p-1$) renvoie `VRAI`. D'où ($\Longrightarrow$).
Si `DFS`($m, e_0, 0) renvoie `VRAI` alors il existe $p \in \mathbb{N}$ et $x \in E$ tel que $p < m$ et `DFS`(m, x, $p+1$) = `VRAI` , donc $x \in F$. Ainsi x est une solution de profondeur $p+1 \leq m$, d'où ($\Longleftrightarrow$) .

Question 7

```
let ids () =
  let rec dfs m e p =
    if p>m then false
    else if final e then true else
      let flag = ref false in
      let voisins = ref (suivants e) in
      while (!voisins <> []) && (!flag = false) do
        flag := dfs m (List.hd !voisins) (p+1);
        voisins := List.tl !voisins
      done;
```

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

Question 8

Comme montrer à la question 6, `DFS(m,e_0,0)` renvoie `VRAI` si et seulement si une solution de profondeur inférieure ou égale à m existe. Or comme `ids` incrémente à chaque fois la valeur de m de 1 (en partant de 0) et fait ensuite appel à `DFS(m,e_0,0)` alors tant qu'une solution de profondeur p n'existe pas m continue d'augmenter. Ainsi lorsque `DFS(m,e_0,0)` renvoie `VRAI` pour la première fois, alors `ids` renvoie la valeur de m tel que `DFS(m,e_0,0) = true` , c'est donc d'une part qu'une solution existe, mais également que c'est une solution de profondeur optimale.

Question 9

1. On suppose qu'il y a exactement un état à chaque profondeur p : Comme il n'y a qu'un seul état pour chaque profondeur, pour le parcours en largeur, $Card(B) = 1$ (et $Card(A) = Card(B)$) durant chaque itération de boucle. Si la profondeur optimal est $s \in \mathbb{N}$, alors il y aura s itération(s) de boucle. Ainsi la complexité spatiale est $O(1)$ et la complexité temporelle $O(s)$. Pour ce qui est de l'algorithme de recherche itérée en profondeur, la fonction étant récursive, il est nécessaire qu'on stock la pile d'appel, et donc, suivant l'hypothèse, cette file est $O(s)$, d'où la complexité spatiale. En temporelle, l'algorithme cherche les solutions pour toutes les profondeurs p où $p \in [\![1; s]\!]$. Comme chaque test des solutions de profondeur p est un $O(p)$, alors en sommant, la complexité temporelle de l'algo est $O(s^2)$.
2. On suppose qu'il y a exactement 2^p états à la profondeur p. L'hypothèse permet d'affirmer que pour chaque itération de boucle, donc pour chaque profondeur p, $Card(B) = Card(A) = 2^p$. Donc la complexité spatiale est $O(2^s)$. Temporellement, on explore $\forall p \in [\![1, s]\!], \forall k \in [\![1, p]\!]$ les 2^k états, soit une complexité en $0(2^s)$. Pour le parcours en profondeur itérée, encore une fois la complexité spatiale correspond à la taille de la pile d'appel, qui est dans ce cas également $O(s)$. En temps, on devra $\forall p \in [\![1, s]\!], \forall k \in [\![1, p]\!]$ explorer les 2^k états, soit encore une complexité en $O(2^s)$.

Question 10

```
let min = ref 0 ;;

let rec dfs m e p =
  let c = p + (h e.value) in
  if c > m then
    ((if c < !min then min := c) ; false)
    else if (final e) then true else
      let flag = ref false in
      let voisins = ref (suivants e) in
      while (!voisins <> []) && (!flag = false) do
        flag := dfs m (List.hd !voisins) (p+1);
        voisins := List.tl !voisins
      done;
  !flag ;;

let idastar () =
  let m = ref (h initial.value) in
  let flag = ref false in
```

```
let m_final = ref !m in
while !m <> max_int && not !flag do
  begin
    min := max_int ;
    if dfs !m initial 0 then (flag := true ; m_final := !m);
    m := !min;
  end
done;
!m_final - 1 ;;
```

Question 11

En prenant:

$$\begin{array}{rcccl} h & : & E & \to & \mathbb{N} \\ & & x & \mapsto & \begin{cases} 0, & \text{si } n = t \\ 1, & \text{sinon} \end{cases} \end{array}$$

On peut montrer que h est bien admissible :
Soit $e \in E$ un état quelconque. Si $e = t$ alors $e \in F$ et donc l'état est solution. Ainsi $h(e) = 0$. Sinon, il reste au moins une étape avant d'atteindre un état solution donc la distance de e à t est supérieur ou égale à 1, or $h(e) \leq 1$ d'où l'admissibilité de h.

Question 12

(Non réussie)

Question 13

Sachant qu'il y a 16 cases, et que chaque état correspond à un placement des 16 valeurs disponibles dans chacune des 16 cases, on peut ainsi conclure qu'il y a 16! états possibles. En espace mémoire, peu importe comment on représente un état (Liste, Array, etc...), stocker de nombreux états parait difficielement envisageable. Sachant qu'en plus un état initial quelconque peu mener à une solution optimale de profondeur de l'ordre de 50 (cf. énoncé), cela confirme qu'il est difficile d'envisager un parcours en largeur afin de résoudre le jeu du taquin.

Question 14

Afin de montrer que h est admissible, il faut montrer que h ne peut renvoyer un entier supérieur au nombre d'états nécéssaires afin d'obtenir une solution optimale. Or le terme $\mid e_v^i - \lfloor v/4 \rfloor \mid$ représente la distance en terme de ligne ou distance verticale par rapport à la bonne position de v, de même, le terme $\mid e_v^j - (v \mod 4) \mid$ représente la distance horizontale par rapport à la bonne position. Or comme il faut au moins une étape pour bouger chaque écart, que ce soit horizontale ou vertical, le nombre d'étapes nécéssaire pour déplacer v, où $v \in [\![0, 14]\!]$, à la bonne position est minoré par la somme des distances verticales et horizontales. Enfin, commme le mouvement d'une case se fait à chaque fois par rapport à la case blanche, le nombre d'étapes afin d'atteindre le bon placement de toutes les cases est minoré par la somme du nombre d'étapes nécéssaires au bon placement de chaque cases. Ce qui montre l'admissibilité de h telle que définie.

Question 15

```
let h_terme i j v =
  abs (i - v/4) +  abs (j - v mod 4)

let move i j =
  let v = grid.(i).(j) in
  begin
    grid.(!li).(!lj) <- grid.(i).(j) ;
```

```
    grid.(i).(j) <- -1 ; (* Représentation de la case vide comme étant la case contenant -1*)
    h := !h - (h_terme i j v) + (h_terme !li !lj v);
    lj := j;
    li := i
  end;;
```

Question 16

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

Question 17

Pour pouvoir tester tous les cas possibles, il faut être capable de revenir en arrière sur un mouvement afin de tester les solutions suivantes dans le cas où un mouvement ne serai optimal. Pour cela, on définit la fonction `annule: unit -> bool`, qui annule le dernier mouvement et met à jour les positions des cases.

```
let annule () =
  match !solution with
    |[] -> false
    |h::t -> begin
              (match h with
              |Gauche -> droite ()
              |Droite -> gauche ()
              |Haut -> bas ()
              |Bas -> haut () );
              solution := t ;
              false
            end
;;
```

On peut maintenant procédé au codage de la fonction `dfs` (en considérant que `!h` correspond à la valeur de la fonction addmissible h de la question **14**).

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

```
    end
;;
```

Question 18

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