COMPOSITION D'INFORMATIQUE (Option) n°2

Corrigé

⋆ ⋆ ⋆

I Déplacement d'un robot dans une grille

Question 1 Un exercice classique de programmation. La référence `i1` garde en mémoire un indice d'une valeur plus petite et `i2` d'une valeur **strictement** plus grande que `a`.

```
let dichotomie a t =
   let i1 = ref 0 and i2 = ref (Array.length t) in
   while !i2 - !i1 > 1 do
      let i = (!i1 + !i2) / 2 in
      if a < t.(i) then
         i2 := i
      else
         i1 := i
   done;
   !i1;;
```

La complexité est bien logarithmique, car la valeur $i_2 - i_1$ diminue de moitié (à partie entière près) à chaque itération.

Question 2 On commence par chercher l'indice de l'obstacle juste avant la position (en largeur et hauteur) et on reconstruit les positions atteignables :

```
let dpt_grille (a, b) =
   let i = dichotomie b ob_li.(a) and j = dichotomie a ob_co.(b) in
   [| a, ob_li.(a).(i) ; a, ob_li.(a).(i + 1) - 1 ;
      ob_co.(b).(j), b ; ob_co.(b).(j + 1) - 1, b |];;
```

Question 3 On construit une matrice vide qu'on remplit case par case avec la fonction précédente :

```
let mat_dpt () =
   let m = Array.make_matrix n n [||] in
   for a = 0 to n - 1 do
      for b = 0 to n - 1 do
         m.(a).(b) <- dpt_grille (a, b)
      done
   done;
   m;;
```

Notons que la complexité de `dpt_grille` est logarithmique en N, car on y fait deux appels à `dichotomie`. De par la présence des deux boucles `for`, on en déduit que la complexité de cette fonction est $O(N^2 \log N)$.

Question 4 On ne fait d'éventuelles modifications qu'en cas d'égalité d'une des deux coordonnées. On prend alors le plus proche des deux points entre l'obstacle et le robot. On suppose ici que les deux points sont distincts.

```
let modif t (a, b) (c, d) =
   if a = c then begin
      if b > d then t.(0) <- (a, max (d + 1) (snd t.(0)))
      else t.(1) <- (a, min (d - 1) (snd t.(1)))
   end;
   if b = d then begin
      if a > c then t.(2) <- (max (c + 1) (fst t.(2)), b)
      else t.(3) <- (min (c - 1) (fst t.(3)), b)
   end;;
```

Question 5 On commence par récupérer le vecteur de déplacements (qu'on copie) en utilisant la matrice de déplacements. La fonction auxiliaire `maj` prend en argument une liste de points et met à jour le vecteur de déplacements en utilisant la fonction précédente.

```
let dpt_robot (a, b) q =
   let t = Array.copy m.(a).(b) in
   let rec maj q = match q with
      | [] -> ()
      | (c, d) :: tl -> modif t (a, b) (c, d); maj tl in
   maj q;
   t;;
```

Question 6 À chaque itération, on peut déplacer chacun des 4 robots dans une des 4 directions. Cela fait un total de 16 déplacements possibles. On en déduit que le nombre de suite de k déplacements est 16^k. Pour connaître le résultat d'un déplacement, il faut utiliser la fonction précédente qui est exécutée en temps constant (car le nombre de robots est constant). La complexité totale est donc $O(N^2 \log N + 16^k)$ (car il est nécessaire de créer la matrice de déplacements).

II Tables de hachage

II.A Une famille de fonction h_w

Question 7 On utilise la règle de Horner pour faire le calcul et éviter de calculer les puissances de N.

```
let rec h w q = match q with
   | [] -> 0
   | (a, b) :: tl -> ((h w tl) * n * n + a + b * n) mod w;;
```

II.B Complexité pour une largeur fixe

Question 8 Dans un premier temps, notons que la recherche d'une clé dans une alvéole est linéaire en la taille de la liste de l'alvéole. De plus, de par le hachage uniforme, l'espérance de la complexité d'une clé non présente est égale à la moyenne des complexités de recherche dans chaque alvéole. Or, la taille moyenne d'une liste d'une alvéole est exactement $\alpha = \dfrac{n}{w}$. On en déduit que la complexité est bien $O(1+\alpha)$ (le 1 est présent car α peut être nul, et il faut $O(1)$ opération pour récupérer la liste de l'alvéole de toute façon).

Question 9 Si on prend une clé, chaque autre clé a une probabilité $1/w$ de se trouver dans la même alvéole. L'espérance de la taille de cette alvéole est donc $1 + \frac{n-1}{w} \leq 1 + \alpha$. La recherche se fera avec une complexité $O(1+\alpha)$ ici encore.

II.C Tables de hachage dynamique

Question 10

```
let creer_table h =
   { h = h; taille = 0; data = [| [] |]; w = 1 };;
```

Question 11 On crée une fonction `assoc` qui cherche l'élément associé à une clé k dans une liste, et on l'exécute dans le haché de k :

```
let recherche t k =
   let rec assoc k = function
      | [] -> failwith "Non trouvé"
      | (x, y) :: q when x = k -> y
      | _ :: q -> assoc k q in
   assoc k t.data.(t.h t.w k);;
```

Question 12 On utilise une fonction auxiliaire `ajout` qui ajoute tous les éléments d'une liste à un tableau de liste créé précédemment, en hachant chaque clé avec la nouvelle fonction h_{w_2}. On l'applique à toutes les listes de la table de hachage. Une fois que c'est terminé, on modifie la table et sa largeur.

```
let rearrange t w2 =
   let d = Array.make w2 [] in
   let rec ajout q = match q with
      | [] -> ()
      | (a, b) :: tl -> d.(t.h w2 a) <- (a, b) :: d.(t.h w2 a);
                        ajout tl in
   for i = 0 to t.w - 1 do
      ajout t.data.(i)
   done;
   t.data <- d;
   t.w <- w2;;
```

La complexité de création du vecteur de listes `d` est $O(w_2)$. La fonction `ajout` a une complexité linéaire en la taille de la liste. La boucle `for` est de taille w, et la complexité totale est de l'ordre de la somme des tailles de toutes les listes, c'est-à-dire $O(n)$. On en déduit bien une complexité totale en $O(n + w + w_2)$.

Question 13 On ajoute le couple à la liste du haché de k, puis on réarrange la table si nécessaire.

```
let ajout t k e =
   if not (recherche t k) then begin
      let i = t.h t.w k in
      t.data.(i) <- (k, e) :: t.data.(i);
      t.taille <- t.taille + 1
   end;
   if t.taille > 3 * t.w then
      rearrange t (3 * t.w);;
```

III Résolution du jeu des robots

III.A Parcours en largeur : étude théorique

Question 14 La terminaison ne concerne ici que la boucle **tant que**, car la boucle **pour** termine toujours. On utilise le variant de boucle $V_n = |F| + \sum_{v \in V} k_v \geq 0$ où k_v vaut 0 si $b[v]$ vaut **vrai** et 1 si $b[v]$ vaut **faux**. On a :

- V_0 vaut $|V| > 0$ avant de rentrer dans la boucle.
- Dans un passage de boucle, on défile un élément de F, puis on en rajoute autant que de sommets dont $b[v]$ devient **vrai**. On a donc $V_{n+1} = V_n - 1 < V_n$.

On en déduit que (V_n) est bien un variant de boucle et donc que l'algorithme termine.

Question 15 Soit $v_0 \to v_1 \to \cdots \to v_k = v$ un chemin de v_0 à v. Si on suppose que v n'est pas visité par l'algorithme, alors il existe $i \in [\![1, k]\!]$ tel que $b[v_i]$ est faux et $b[v_{i-1}]$ est vrai. Dans ce cas, le sommet v_{i-1} est passé par la file, et lorsqu'il a été défilé, tous ses voisins dont le marqueur valait **faux** ont été marqué en **vrai**, en particulier v_i. Par l'absurde, on en déduit le résultat.
Réciproquement, remarquons qu'un sommet v aura $b[v]$ qui est vrai si et seulement s'il est passé par la file. Or, on rajoute un sommet dans la file seulement s'il est voisin d'un sommet passé avant lui dans la file. Par une induction simple, on en déduit qu'il existe un chemin de v_0 à v (v_0 étant initialement l'unique sommet dans la file).

Question 16 Le tableau p permet de retrouver les prédécesseurs. On retrouve donc le chemin en le construisant à l'envers : en utilisant les mêmes notations que la question précédente, on a $v_{i-1} = p[v_i]$ pour tout $i \in [\![1, k]\!]$.

Question 17 Initialement, comme $n_{v_0} = 1$, la propriété est bien vérifiée. Supposons alors la propriété vraie pour tous les sommets jusqu'au numéro n et soit v le sommet tel que $n_v = n + 1$.

1. Supposons que $d_v < \ell_v$. Et soit un plus court chemin $v_0 \rightsquigarrow t \to v$. Notons de plus u le prédécesseur de v dans p. On a $n_u < n_v$ par hypothèse. De plus, comme on a supposé $d_v < \ell_v$, alors $d_t < d_u$ et donc $n_t < n_u$. Dans ce cas, le sommet t a été défilé avant le sommet u, et donc le prédécesseur de v aurait dû être t et non u. Par l'absurde, on en déduit que $d_v = \ell_v$.
2. Supposons qu'il existe un sommet t tel que $d_t < d_v$ et $n_t > n_v$. Soit $v_0 \rightsquigarrow t' \to t$ un plus court chemin de v_0 à t. Alors $v_0 \rightsquigarrow t'$ est un plus court chemin de v_0 à t' et donc $d_{t'} = d_t - 1$.
   Soit alors v' le prédécesseur de v dans p. On a, par hypothèse, $n_{v'} < n_v$ et par le point prouvé précédemment, $d_{v'} = \ell_{v'} = \ell_v - 1 = d_v - 1$.
   Finalement, on a $d_{t'} < d_{v'}$, et donc par hypothèse d'induction $n_{t'} < n_{v'}$. Cela signifie que t' a été défilé avant v' et donc que t aurait dû être enfilé avant v. Par l'absurde on a bien le second point.

Question 18 Remarquons que chaque arc et chaque sommet n'est parcouru ou visité qu'une seule fois dans le pire des cas. On en déduit que la complexité totale est en $O(|V| + |E|)$.

Question 19 La fonction `enfiler_voisins` permet de rajouter à la file et de faire la modification de b pour chaque voisin correspondant. Le reste découle directement de l'algorithme.

```
let parcours g v0 =
   let n = Array.length g in
   let f = creer_file () in
   let b = Array.make n false and
       p = Array.make n 0 in
   for i = 0 to n - 1 do
      p.(i) <- i
   done;
   b.(v0) <- true; enfiler f v0;
   let rec enfiler_voisins = function
      | [] -> ()
      | v' :: q when not b.(v') -> b.(v') <- true; enfiler f v';
                                   enfiler_voisins q
      | _ :: q -> enfiler_voisins q in
   while not (vide f) do
      let v = defiler f in
      enfiler_voisins g.(v)
   done;
   b, p;;
```

III.B Graphe orienté associé au jeu des robots

Question 20 Pour déterminer un sommet, il faut dans un premier temps déterminer les positions de tous les robots, soit $\binom{N^2}{p}$ possibilités, puis choisir le robot principal, soit p possibilités. Il y a donc $p\binom{N^2}{p}$ sommets dans le graphe. Cela représente 699 170 560 sommets pour $p = 4$ et $N = 16$.

Question 21 On utilise une fonction auxiliaire `creer` qui crée un sommet (pour faciliter l'écriture). De plus, on utilise une fonction `supp` qui supprime un élément dans une liste. Pour la fonction `accessibles`, on crée une référence de liste qu'on modifiera chaque fois qu'il faudra ajouter un sommet. On commence par traiter le cas du déplacement du robot principal, puis on utilise une fonction récursive `ajout_dpt` qui, pour chaque autre robot, calcule ses déplacements possibles, crée les sommets correspondants et les ajoute à la liste. Enfin, on utilise une fonction d'insertion qui insère un élément dans une liste triée :

```
let creer x q = {r = x; q = q};;

let rec supp x q = match q with
   | [] -> []
   | y :: tl when x = y -> tl
   | y :: tl -> y :: (supp x tl);;

let rec insertion x = function
   | [] -> [x]
   | y :: q when x < y -> x :: y :: q
   | y :: q -> y :: (insertion x q);;
```

```
let accessibles v =
   let l = ref [] in
   let x = v.r and q = v.q in
   let d = dpt_robot x q in
   for i = 0 to 3 do
      l := (creer d.(i) q) :: !l
   done;
   let rec ajout_dpt q = match q with
      | [] -> ()
      | y :: tl -> let r = supp y q in
                   let d = dpt_robot y (x :: r) in
                   for i = 0 to 3 do
                      l := (creer x (insertion d.(i) r)) :: !l
                   done;
                   ajout_dpt tl in
    ajout_dpt q;
    !l;;
```

⋆⋆⋆