Devoir surveillé 3 c

X 2020

Option informatique MP1, MP2 & MP3

Labyrinthes

I Construire des labyrinthes

Mélange de Knuth

Question 1

```
let melange_knuth a =
   let n = Array.length a in
   for i = 1 to n - 1 do
      let j = Random.int (i + 1) in
      if j < i
      then begin let tmp = a.(j) in
           a.(j) <- a.(i);
           a.(i) <- tmp end done;;
```

Question 2

Montrons que l'on a l'invariant de boucle suivant : au départ de la boucle pour l'indice i on a

$$\{x_0, x_1, \ldots, x_{i-1}\} = \{0, 1, \ldots, i-1\} \tag{1}$$

$$\forall p, q \in \{0, i-1\},\ \Pr(x_p = q) = \frac{1}{i} \tag{2}$$

$$\forall p \in \{i, \ldots, n-1\},\ x_p = p \tag{3}$$

Où x_p est la valeur de `a.(p)`.

Initialisation Au départ, pour $i = 0$, on a $x_p = p$ pour tout $p \in \{0, \ldots, n-1\}$.

Hérédité On suppose que la propriété est valide au début de la boucle pour l'indice i.

On note y_p est la valeur de `a.(p)` à la fin de l'exécution du corps de la boucle. On va prouver que les propriétés de l'invariant sont vérifiées, pour $i+1$, en remplaçant x_p p ar y_p.

(3) Les valeurs de `a.(p)` sont inchangées pour $p \geqslant i+1$ donc $y_p = p$ pour $p \geqslant i+1$.

(1) On a permuté les valeurs : $y_i = x_j$ et $y_j = x_i$ avec $0 \leqslant j \leqslant i$ (même pour $j = i$) donc les valeurs des y_0, y_1, …, y_i sont bien une permutation de $\{0, 1, \ldots, i-1\}$.

La valeur de y_i est celle de x_j avec j choisi aléatoirement entre 0 et i, or les valeurs de x_0, …, x_i sont les entiers de 0 à i donc $\Pr(y_i = q) = \frac{1}{i+1}$ pour $q \in \{0, i\}$.

Pour $0 \leqslant p < i$, $y_p = x_p$ si $j \neq p$ donc avec la probabilité $\frac{i}{i+1}$ et $y_p = i$ si $j = p$ donc avec la probabilité $\frac{1}{i+1}$.

Ainsi $\Pr(y_p = q) = \frac{i}{i+1} \Pr(p_p = q) \frac{i}{i+1} \frac{1}{i} = \frac{1}{i+1}$ pour $q < i$ et $\Pr(y_p = i) = \frac{1}{i+1}$: (2) est valide.

Preuve L'invariant pour $i = n$ donne bien $\forall p, q \in \{0, n-1\},\ \Pr(x_p = q) = \frac{1}{n}$?

Parcours en profondeur aléatoire

Question 3

Pour réaliser un labyrinthe à partir d'un graphe, on initialise un nouveau graphe vide `laby`, et on ajoute les arêtes au fur et à mesure du parcours en profondeur du graphe *initial*.

```
let labyrinthe1 g =
   let laby = graphe_vide g.n in
   let vus = Array.make g.n false in
   let rec  traiter s =
      let voisins = Array.of_list g.adj.(s) in
      melange_knuth voisins;
      let k = Array.length voisins in
      for i = 0 to (k-1) do
         let t = voisins.(i) in
         if not vus.(t)
         then begin vu.(t) <- true;
              ajoute_arete laby s t
              traiter t end done;
    vus.(0) <- true;
    traiter 0;
    laby;;
```

Question 4

L'algorithme précédent admet pour invariant que toutes les arêtes du graphe en cours de construction forment un arbre dont l'un des sommets est `0` et les autres sommets sont ceux visités ensuite. À chaque étape, on ajoute une arête entre un sommet de l'arbre et un sommet qui n'en fait pas partie, ce qui préserve la propriété que les arêtes constituent un arbre.
Comme cet invariant est vérifié au début du parcourt, il l'est encore à la fin. Or, ayant supposé que le graphe initial est connexe, tous ses sommets auront été visité, donc on aura obtenu un arbre contenant tous les sommets, c'est-à-dire un labyrinthe parfait.

Classes disjointes

Question 5

```
let rec cd_trouve cd n =
let p = cd.lien.(n) in
   if p = n
   then n
   else cd_trouve cd p;;
```

Question 6

```
let cd_union cd i j =
    let ci = cd_trouve cd i in
    let cj = cd_trouve cd j in
    let ri = cd.rang.(ci) in
    let rj = cd.rang.(cj) in
    if ri < rj
    then cd.lien.(ci) <- cj
    else if ri > rj
         then cd.lien.(cj) <- ci
         (* on ne fait rien si les 2 classes sont égales *)*
         else if ci <> cj
              then begin cd.lien.(cj) <- ci;
                   cd.rang.(ci) <- cd.rang.(ci) + 1 end;;
```

Question 7

Nombre d'éléments

On suppose, qu'avant une fusion, chaque classe admet au mois 2^k élément si son rang est k.
Si les deux classes fusionnées ont des rangs différents, alors le rang de la classe obtenue est égal au rang d'une des classes de départ, k et son cardinal a augmenté. Ainsi, le cardinal est toujours minoré par 2^k, les autres classes sont inchangées.
Si les classes fusionnées sont de même rang k, la classe obtenue est de rang $k+1$ et consiste en la fusion de 2 classes ayant chacune au moins 2^k éléments. La classe obtenue a donc bien au moins 2^{k+1} éléments.
La première propriété est donc conservée à chaque fusion ; elle est vraie pour toute relation d'équi

Hauteur

On suppose, qu'avant une fusion, chaque classe de rang k admet un élément de hauteur k et que toutes les hauteurs sont majorées par k.
Si les deux classes fusionnées ont des rangs différents avec $k_1 < k_2$, l'élément de hauteur maximale de la seconde classe est conservé, à la hauteur k_2, tous ses autres éléments sont de hauteur k_2 au plus et ceux de la première classe on une hauteur augmentée de 1 donc majorée par $k_1 + 1 \leqslant k_2$.
Si les deux classes fusionnées ont des rangs différents avec $k_2 < k_1$.
Si les classes fusionnées sont de même rang k, les élément de la classe augmentée ont une hauteur toujours majorée par k, ceux de la classe incluse , une hauteur majorée par $k+1$ et son élément qui était de hauteur k est maintenant de hauteur $k+1$.
La seconde propriété est donc conservée à chaque fusion.

Question 8

On initialise une relation d'équivalence avec `cd_init` : toutes les classes ont un rang 0 et contiennent 1 élément, on a bien $1 \geqslant 0$ et toutes les classes ont un rang 0 et contiennent un seul élément, de hauteur 0. Ainsi les deux propriétés précédentes sont vérifiées au départ.
Comme elles sont conservées lors de chaque fusion, elles sont vraies pour chaque classe.
Pour un sommet s appartenant à une classe de rang k, la complexité de l'exécution de `cd_trouve (s)` est égale à la hauteur augmentée de 1 donc est majorée par $k+1$ et la classe contient 2^k éléments donc $2^k \leqslant n$. La complexité est majorée par $1 + \log_2(n)$, c'est un $\mathcal{O}\big(\ln(n)\big)$.

Application à la construction d'un labyrinthe

Question 9

```
let labyrinthe2 g =
   let cc = cd_init g.n in (* cc pour composantes connexes *)
   let ar = aretes g in
   let laby = graphe_vide g.n in
   melange_knuth ar;
   let p = Array.length ar in
   for i = 0 to (p-1) do
      let (s, t) = ar.(i) in
      if cd_trouve cc s <> cd_trouve cc t
      then begin ajoute_arete laby s t;
                 cd_union cc s t end done;
   laby;;
```

Question 10

Lorsque l'on traite une arête qui relie deux sommets appartenant à la même composante, on ne change rien, le sous graphe reste une forêt de labyrinthes parfaits.
Lorsque l'on traite une arête qui relie deux classes d'équivalence distinctes h_1 et h_2 par les sommets s_1 et s_2 on crée un sous-graphe h.

- h est connexe car formé par deux graphes connexes que l'on a reliés par une arête,
- Un chemin simple dans h entre deux sommets de h_1, s'il ne restait pas dans h_1 devrait parcourir l'arête (s_1, s_2) pour sortir de h_1 puis la parcourir de nouveau pour revenir dans h_1, ce qui contredit la simplicité du chemin. Ainsi un chemin simple entre deux sommets de h_1 reste danns h_1 : il est donc unique car h_1 est un labyrinthe.
- De même il existe un unique chemin simple dans h reliant deux sommets de h_2.
- Un chemin simple dans h entre deux sommets de h_1 et de h_2, t_1 et t_2, doit passer par l'arête (s_1, s_2) et ne peut la parcourir qu'une fois. Il est donc composé de l'unique chemin simple dans h_1 entre t_1 et s_1, de l'arête (s_1, s_2) et de l'unique chemin simple dans h_2 entre s_2 et t_2.
- Ainsi h est un labyrinthe.

À chaque étape on obtient bien une forêt de labyrinthe, à la fin on a donc un unique arbre qui est un labyrinthe.

Algorithme d'Eller

Question 11

On va prouver qu'avant le traitement de la ligne ℓ on a l'invariant suivant :

1. tous les sommets de hauteur inférieure ou égale à ℓ sont dans la même classe d'équivalence qu'au moins un sommet de hauteur ℓ,
2. tout sommet de hauteur strictement supérieure à ℓ est le seul membre de sa classe d'équivalence,
3. les sous-graphes induits par la relation d'équivalence sont des arbres.

Pour $\ell = 0$, il n'y a rien à prouver.
Si la propriété est vraie avant les étapes (a) et (b) pour une ligne qui n'est pas la dernière alors

- lors de l'étape (a), on ne relie que des sommets appartenant à des classes d'équivalences distincts, donc les sous-graphes induits par la relation d'équivalence après cette étape restent des arbres,

- après l'étape (b), tout sommet de hauteur au plus ℓ admet (au moins) un sommet de hauteur ℓ dans sa classe d'équivalence, et on a ajouté à chaque classe d'équivalence contenant des sommets de hauteur ℓ (autrement dit, toutes les classes d'équivalences contenant des sommets de hauteur au plus ℓ) au moins une arête vers un sommet de hauteur $\ell + 1$.

De plus, les modifications ne concernant que les les sommets de hauteur au plus $\ell + 1$ (avant l'incrémentation finale de ℓ), les sommets de hauteur au moins $\ell + 2$ restent les seuls éléments de leur classe d'équivalence.
Ainsi l'invariant est conservé au rang $\ell + 1$.

Lors du traitement de la dernière ligne (étape 2.), toutes les classes d'équivalences contiennent au moins un sommet de la dernière ligne, et les sous-graphes induits par les classes d'équivalences sont des arbres. On ajoute le maximum d'arêtes possibles en reliant à chaque fois deux classes d'équivalences, on obtient à la fin une unique classe d'équivalence comprenant tous les sommets du graphe initial, et le graphe obtenu est un arbre couvrant cette classe d'équivalence, c'est donc bien un labyrinthe parfait.

Question 12

Étant donné un labyrinthe parfait h de g, on peut l'obtenir avec une probabilité non nulle à l'aide de l'algorithme d'Eller.

- Pour chaque lignes avant la dernière, les arêtes horizontales de h ne peuvent relier deux sommets d'une même classe d'équivalence : elles peuvent donc être les choix faits avec la probabilité $\frac{1}{2}$ pour chacune donc la probabilité de les choisir est non nulle.
- Il existe au moins une arête verticale de h pour chaque classe d'équivalence donc les arêtes verticales entre deux lignes de h peuvent être le résultat du choix aléatoire de (b) avec une probabilité non nulle.
- L'ordre aléatoire du choix des arêtes de la dernière ligne peut être celui qui commence par les arêtes de h, auquel cas elles vont alors toutes être sélectionnées, et aucune des suivantes.

À la fin, on obtient bien le labyrinthe obtenu et avec une probabilité non nulle, chacun des choix opportuns (en nombre fini) ayant une probabilité non nulle.
Il n'est pas demandé de prouver que tous les labyrinthes sont équi-probables, je ne sais pas si cela est le c as

II Résoudre un labyrinthe

Files à deux bouts

Question 13

Dans OCaml, `a mod n` est négatif si a est négatif d'où l'ajout de `cap`.

```
let ajoute_debut f a =
   let cap = Array.length f.contenu in
   if f.taille < cap
   then begin f.taille <- f.taille + 1;
        f.debut <- (f.debut - 1 + cap) mod cap;
        f.contenu.(f.debut) <-  end
   else failwith "La file est pleine";;
```

Question 14

```
let retire_debut f =
  let cap = Array.length f.contenu in
  if f.taille > 0
  then begin f.taille <- f.taille - 1;
       let a = f.contenu.(f.debut)
       f.debut <- (f.debut + 1) mod cap;
       a end
  else failwith "La file est vide";;
```

Parcours en largeur

Question 15

On note $d(s)$ la longueur minimale d'un chemin de `src` vers s, sa distance.
On montre qu'à chaque moment

1. la file d'attente contient des sommets s_1, s_2, …, s_p (dans l'ordre de début à la fin) tels que $d(s_1) \leqslant d(s_2) \leqslant \cdots \leqslant d(s_p) \leqslant d(s_1) + 1$,
2. tous les sommets de distance $d(s_1)$ ou moins ont été insérés,
3. toutes les valeurs de `distance.(i)` sont les distances des sommets correspondants.

Cette propriété est vrai lors de l'initialisation (étape 3) car la file ne contient que `src` qui est à distance 0 et c'est le seul sommet de distance 0.
On suppose que cette propriété est vraie avant un passage ; on retire le sommet s_1 et on ajoute les voisins de s_1 non encore marqués, t_1, …, t_q.

1. On accède à ces sommets depuis s_1 donc leur distance est au plus $d(s_1) + 1$, comme les sommets à distance $d(s_1)$ ont été déjà ajoutés, leur distance est exactement $d(s_1) + 1$. On a donc $d(s_2) \leqslant \cdots d(s_p) \leqslant d(s_1) + 1 = d(t_1) \cdots = d(t_q) = d(s_1) + 1 \leqslant d(s_2) + 1$.
2. Si on a $d(s_2) = d(s_1)$ alors tous les sommets de distance $d(s_2)$ au plus ont été insérés.
   Si on a $d(s_2) = d(s_1) + 1$ alors tous les sommets de distance $d(s_1)$ ont étés traités donc tous les sommets de distance $d(s_1) + 1 = d(s_2)$ ont été insérés.
3. Les valeurs données dans le tableau `distance` sont `distance.(s1)+ 1` qui est égale, selon l'hypothèse de récurrence à $d(s_1) + 1$, ce qui est bien la distance des t_j.

Ainsi la propriétés reste vraie.

Comme le graphe est connexe, le sommet `dst` est atteint donc la valeur de `distance.(dest)` est bien la valeur minimale des longueur des chemins de `src` à `dst`.

En croisant un minimum de monstres

Question 16

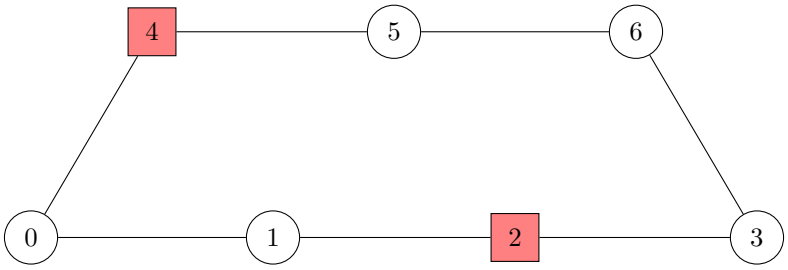
```
let minimum_monstres g monstre src dst =
   let distance = Array.make g.n (-1, -1) in
   let f = file_vide g.n in
   ajoute_fin f src;
   distance.(src) <- (0, 0);
   let rec loop () =
      let v = retire_debut f in
      if v = dst
      then distance.(dst)
      else begin
          List.iter
          (fun w -> let m, d = distance.(v) in
                    if fst distance.(w) = -1
                    then if monstre.(w)
                         then (distance.(w) <- (m+1, d+1);
                               ajoute_debut f w)
                         else (distance.(w) <- (m, d+1);
                               ajoute_fin f w))
          g.adj.(v);
         loop () end in
   loop ();;
```

Un boucle while semble plus lisible

```
let minimum_monstres g monstre src dst =
   let distance = Array.make g.n (-1, -1) in
   let f = file_vide g.n in
   ajoute_fin f src;
   distance.(src) <- (0, 0);
   while distance.(dst) = (-1, -1) do
     let v = retire_debut f in
     let m, d = distance.(v) in
     let traiter w =
        if distance.(w) = (-1, -1)
        then if monstre.(w)
             then begin distance.(w) <- (m+1, d+1);
                        ajoute_fin f w end
             else begin distance.(w) <- (m, d+1);
                        ajoute_debut f w end in
     List.iter traiter g.adj.(v) done;
   distance.(dst);;
```

Question 17

Voici un exemple possible.



On choisit `src` = 0 et `dst` = 3. La file et le tabeau des distances ont les valeur successives ci-dessus (le début est en gras). Les monstres sont en rouge.

0	-1	-1	-1	-1	-1	-1	(0, 0)	(-1,-1)	(-1,-1)	(-1,-1)	(-1,-1)	(-1,-1)	(-1,-1)
1	4	-1	-1	-1	-1	-1	(0, 0)	(0,1)	(-1,-1)	(-1,-1)	(1,1)	(-1,-1)	(-1,-1)
1	**4**	2	-1	-1	-1	-1	(0, 0)	(0, 1)	(1, 2)	(-1,-1)	(1,1)	(-1,-1)	(-1,-1)
1	**5**	2	-1	-1	-1	-1	(0, 0)	(0, 1)	(1, 2)	(-1,-1)	(1,1)	(1, 2)	(-1,-1)
1	**6**	2	-1	-1	-1	-1	(0, 0)	(0, 1)	(1, 2)	(-1,-1)	(1,1)	(1, 2)	(1, 3)
1	**3**	2	-1	-1	-1	-1	(0, 0)	(0, 1)	(1, 2)	(1, 4)	(1,1)	(1, 2)	(1, 3)

On aboutit ainsi à une distance renvoyée de 4 alors qu'il existe un chemin avec 1 monstre aussi mais de longueur 3.

Minimum de monstres et longueur minimale

Question 18

On indique les sommets dans les files par un couple : numéro du sommet (k) et distance, la première colonne indique le sommet traité.

k	`f`	`sources_courantes`	`sources_suivantes`	m
	(0, 0)	$\emptyset$	$\emptyset$	0
0	(1, 1)	$\emptyset$	(4, 1)	0
1	(2, 2)	$\emptyset$	(4, 1), (5, 2)	0
2	$\emptyset$	$\emptyset$	(4, 1), (5, 2), (3, 3), (6, 3)	0
	$\emptyset$	(4, 1), (5, 2), (3, 3), (6, 3)	$\emptyset$	1
	(4, 1)	(5, 2), (3, 3), (6, 3)	$\emptyset$	1
4	(8, 2) , (5, 2)	(3, 3), (6, 3)	$\emptyset$	1
8	(5, 2), (12, 3), (3, 3), (6, 3)	$\emptyset$	(9, 3)	1
5	(12, 3), (3, 3), (6, 3)	$\emptyset$	(9, 3)	1
12	(3, 3), (6, 3), (13, 4)	$\emptyset$	(9, 3)	1
3	(6, 3), (13, 4), (7, 4)	$\emptyset$	(9, 3)	1
6	(13, 4), (7, 4)	$\emptyset$	(9, 3), (10, 4)	1
13	(7, 4), (14, 5)	$\emptyset$	(9, 3), (10, 4)	1
7	(14, 4), (11, 5)	$\emptyset$	(9, 3), (10, 4)	1
14	(11, 5), (15, 5)	$\emptyset$	(9, 3), (10, 4)	1
11	(15, 5)	$\emptyset$	(9, 3), (10, 4)	1

Question 19

Les invariants qui permettraient de prouver la correction peuvent être donnés sous la forme suivante. m est le nombre de monstre attribués au sommet et l est la distance attribuée.

- Le nombre de monstres est constant pour les sommets dans `f`, m.
- Le nombre de monstres est constant et vaut m pour les sommets dans `sources_courantes`.
- Le nombre de monstres est constant et vaut $m+1$ pour les sommets dans `sources_suivantes`.

- Tous les sommets avec un nombre de monstres égal à m au plus dans le chemin minimal ont été ajoutés (et éventuellement retirés).
- `f` contient des sommets s_1, s_2, ..., s_p (dans l'ordre de début à la fin) avec des longueurs croissantes et ne prenant que 2 valeurs, l et $l+1$
- Les longueurs des sommets dans `sources_courantes` sont croissantes et minorées par $l+2$.
- Les longueurs des sommets dans `sources_suivantes` sont croissantes.

Question 20

Si le graphe contient n sommets, les chemins minimaux contiennent aux plus $n-1$ arêtes. L'ordre lexicographique sur les couples (m, l) peut donc se ramener à l'ordre naturels sur les entiers $n.m+l$. En effet

- pour $m_1 < m2$, on a $n.m_1 + l_1 < n.m_1 + n = n.(m_1+1) \leqslant n.m_2 \leqslant n.m_2 + l_2$
- et $n.m_1 + l_1 < n.m_2 + l_2$ pour $m_1 = m_2$ et $l_1 < l_2$.

Si on donne le poids 1 à une arête d'un sommet quelconque vers un sommet non monstre et le poids n à une arête d'un sommet quelconque vers un sommet monstre, le poids d'un chemin sera $n.m + l$ donc le chemin de poids minimal dans le graphe pondéré sera le chemin de coût minimal.