I - Construire des labyrinthes

Mélange de Knuth

1. La description de l'algorithme se traduit directement en la fonction suivante :

```
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 (let tmp = a.(j) in
              a.(j) <- a.(i);
              a.(i) <- tmp) done;;
```

2. Montrons que l'on a l'invariant de boucle suivant :

$$\forall p, q \in \{0, i-1\},\ \Pr(x_p = q) = \frac{1}{i} \quad \text{et} \quad \forall p \in \{i, n-1\},\ x_p = p$$

Pour cela, supposons que l'invariant est vérifié en début de boucle et notons x_p (resp. y_p) la valeur de la p-ième case de `a` au début (resp. à la fin) de l'exécution du corps de la boucle.
Notons tout d'abord qu'il est clair que :

$$\forall\, p \in \{i, n-1\},\ y_p = x_p = p$$

Concernant les probabilités, commençons par $y_{\mathtt{i}-1}$.
— si $y_{\mathtt{i}-1} = i-1$, la seule possibilité a été que l'on a eu $\mathtt{j} = i-1$, ce qui est arrivé avec une probabilité $\frac{1}{i}$:

$$\Pr(y_{\mathtt{i}-1} = i-1) = \frac{1}{i}$$

— Sinon, en notant $\alpha = y_{\mathtt{i}-1} < i-1$, on a eu $\mathtt{j} < i-1$ alors $x_{\mathtt{j}} = \alpha$. Cela arrive avec une probabilité

$$\Pr(y_{\mathtt{i}-1} = \alpha) = \sum_{\mathtt{j}=0}^{\mathtt{i}-2} \frac{1}{\mathtt{i}} \Pr(x_{\mathtt{j}} = \alpha) = (i-1) \times \frac{1}{i} \times \frac{1}{i-1} = \frac{1}{i}$$

Pour les $y_{\mathtt{q}}$ avec $\mathtt{q} < \mathtt{i - 1}$, à nouveau, deux cas sont possibles :
— si $y_{\mathtt{q}} = \mathtt{i - 1}$, c'est que l'on a eu $\mathtt{j} = \mathtt{q}$, ce qui est arrivé avec une probabilité $\frac{1}{\mathtt{i}}$.
— Sinon, en notant $\beta = y_{\mathtt{q}}$, on avait $\mathtt{j} \neq \mathtt{q}$ (avec une probabilité $\frac{\mathtt{i}-1}{\mathtt{i}}$) et $x_{\mathtt{q}} = \beta$ (avec une probabilité $\frac{1}{\mathtt{i}-1}$), d'où

$$\Pr(y_{\mathtt{q}} = \beta) = \frac{\mathtt{i}-1}{\mathtt{i}} \times \frac{1}{\mathtt{i}-1} = \frac{1}{\mathtt{i}}$$

On a ainsi montré que

$$\forall\, p, \mathtt{q} \in \{0, i-1\},\ \Pr(y_p = \mathtt{q}) = \frac{1}{\mathtt{i}}$$

ce qui conclut la preuve de notre invariant de boucle.
Puisque l'invariant est vérifié au début de l'exécution de la boucle (ce qui revient à remarquer que l'on a $x_p = p$ pour tout `p`), il est aussi vérifié en fin de boucle. Ainsi, à l'issue de l'exécution de l'algorithme, on a

$$\forall\, p, \mathtt{q} \in \{0, n-1\},\ \Pr(y_p = \mathtt{q}) = \frac{1}{\mathtt{n}}$$

Parcours en profondeur aléatoire

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

   Notons que lors du parcours, un sommet n'a pas été encore visité s'il n'a pas de voisins dans `laby`.

```
let labyrinthe1 graphe =
  let laby = graphe_vide graphe.n in
  let rec aux sommet =
    let voisins = Array.of_list graphe.adj.(sommet) in
    melange_knuth voisins ;
    Array.iter
      (fun s ->
        match laby.adj.(s) with
        | [] ->
            ajoute_arete laby sommet s ;
            aux s
        | _ -> () (* le sommet a déjà été visité *)
      )
      voisins
  in aux 0 ;
  laby;;
```

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 couvrant intégralement le graphe initial, c'est-à-dire un labyrinthe parfait.

Classes disjointes

5.
```
let rec cd_trouve cd n =
  if cd.lien.(n) = n
  then n
  else cd_trouve cd cd.lien.(n)
```

6. Il faut faire attention à deux choses : au cas où les deux éléments appartiennent à une même classe, et au cas où les deux classes ont le même rang, unique cas le rang d'une classe augmente.

```
let cd_union cd i j =
    (* on détermine les représentants des classes de i et j *)
    let repr_i = cd_trouve cd i
    and repr_j = cd_trouve cd j in
    if repr_i <> repr_j
    then begin (* on récupère les rangs *)
         let rang_i = cd.rang.(repr_i)
         and rang_j = cd.rang.(repr_j) in
         if rang_i < rang_j
         then cd.lien.(i) <- j
         else if rang_i > rang_j
              then cd.lien.(j) <- i
              else (* égalité, on choisit de fusionner en i *)
                   (cd.lien.(j) <- i ;
                    cd.rang.(i) <- cd.rang.(i) + 1)
        end;;
```

7. Pour la première propriété, si avant fusion, toutes les classes de rang k contiennent au moins 2^k éléments, alors lors de la fusion des classes (distinctes) de i et `j`, si elles ont des rangs différents, alors le rang de la classe obtenue est égal au rang d'une des classes de départ alors que son cardinal a augmenté. Ainsi, si la classe obtenue est de rang k, elle contient au moins 2^k éléments, ce qui était déjà le cas au départ.

   Sinon, si les classes sont de même rang k, la classe obtenue est de rang $k+1$ et consiste en la fusion de 2 classes ayant chacunes au moins 2^k éléments. La classe obtenue a donc bien au moins 2^{k+1} éléments.

   Concernant la seconde propriété, à nouveau, si l'on fusionne des classes de rangs différents, le résultat est direct. Par contre si l'on fusionne deux classes distinctes de représentants i et `j` et de même rang k, en supposant que l'on lie `j` à i, alors par hypothèse d'induction :
   — tous les chemins des classes initiales vers leurs représentant est de longueur au plus k ;
   — il existe un sommet `s` de classe de `j` dont le chemin jusqu'à `j` est de longueur k.
   Après fusion, tous les chemins des classes initiales vers i (le représentant de la classe obtenue) sont de longueur au plus $k+1$, et le chemin de `s` à i est bien de longueur $k+1$.

8. Pour une relation d'équivalence obtenue à partir de `cd_init` en appliquant `cd_fusion`, d'après l'invariant précédent (vérifié au départ où chaque classe est de rang 0), ainsi, tout classe de rang k doit au moins contenir 2^k éléments. Or, si la relation d'équivalence est sur $\{0, n-1\}$, le rang k doit vérifier $2^k \leq n$ et donc $k \leq \log_2(n)$.

   De plus, comme le rang k d'une classe majore la longueur maximale du chemin d'un élément `s` à son représentant, la complexité de l'exécution de `cd_trouve(s)` est majorée par k et donc par $\log_2(n)$. Ainsi, la fonction `cd_trouve` est en $O(\log n)$.

Application à la construction d'un labyrinthe

9. On traduit directement l'algorithme proposé.

```
let labyrinthe2 graphe =
  let rel = cd_init graphe.n
  and tableau_aretes = aretes graphe
  and laby = graphe_vide graphe.n in
  melange_knuth tableau_aretes ;
  Array.iter (fun (s1, s2) ->
              let r1 = cd_trouve rel s1
              and r2 = cd_trouve rel s2 in
              if r1 <> r2
              then begin (* on ajoute une arete *)
                   ajoute_arete laby s1 s2 ;
                   (* et on fusionne les classes correspondantes *)
                   cd_union rel r1 r2 end)
             tableau_aretes ;
  laby;;
```

10. Il faut uniquement vérifier que l'invariant est préservé lorsque l'on modifie effectivement le graphe et la relation d'équivalence, i.e. lorsque les deux sommets ne sont pas de la même classe d'équivalence de la relation X.

    Or, dans ce cas, par hypothèse d'induction, les deux sommets initiaux $\mathtt{s}_1$ et $\mathtt{s}_2$ appartiennent à deux sous-graphes h_1 et h_2 qui sont des labyrinthes parfaits des sous-graphes g_1 et g_2 (ces sous-graphes étant respectivement ceux de `h` et `g` induits par les classes d'équivalence de $\mathtt{s}_1$ et $\mathtt{s}_2$). L'ajout d'une arête entre $\mathtt{s}_1$ et $\mathtt{s}_2$ conduit à un labyrinthe parfait h_+ du sous-graphe g_+ de `g` induit par la nouvelle relation d'équivalence. En effet,
    — le sous-graphe h_+ obtenu est connexe, étant formé par deux graphes connexes que l'on a reliés par une arête,
    — il existe un unique chemin reliant toute paire de sommets de h_+ (on rappelle que l'on ne considère que les chemins *simples*). En effet, deux sommets de h_1 et de h_2 sont reliés dans h_+ par un unique chemin, et pour tous sommets u de h_1 et v de h_2, il existe un unique chemin de h_+ allant de u à v, constitué de l'unique chemin de h_1 de u à $\mathtt{s}_1$, puis de la nouvelle arête de $\mathtt{s}_1$ à $\mathtt{s}_2$, puis de l'unique chemin de h_2 de $\mathtt{s}_2$ à v.

    Ainsi, à la fin de l'exécution de l'algorithme, la relation d'équivalence X ne contient plus qu'une unique classe, et donc `h` est un labyrinthe parfait de `g`.

Algorithme d'Eller

11. La boucle de l'algorithme d'Eller vérifie l'invariant suivant (où ℓ désigne la ligne sur laquelle on va appliquer l'étape (a)) :

    (a) 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 ;

    (b) tout sommet de hauteur strictement supérieure à ℓ est le seul membre de sa classe d'équivalence ;

    (c) les sous-graphes induits par la relation d'équivalence sont des arbres.

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

    Ensuite, 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 $\ell + 1$.

    Enfin, 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, à l'issue de la phase 1. de l'algorithme, d'après notre invariant (qui est clairement vérifié au départ), 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.

Puisqu'à l'étape 2, 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.

12. Étant donné un labyrinthe parfait h de g, on peut l'obtenir avec une probabilité non-nul à l'aide de l'algorithme d'Eller. En effet,
    - pour chaque étape de la phase 1, on peut considérer que les arêtes sont parcourues dans l'ordre et que pour chaque arête candidate ne reliant pas deux sommets d'une même classe d'équivalence, la sélection aléatoire de l'arête coïncide avec sa présence dans le labyrinthe. Ensuite, on ajoute les arêtes vers la ligne suivante correspondant au labyrinthe voulu (c'est correct puisque l'on ajoute au moins une arête par classe d'équivalence).
    - pour l'étape 2, on peut considérer que l'on parcourt les arêtes de la dernière ligne en commençant par celles appartenant au labyrinthe. Celles-ci 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.

II - Résoudre un labyrinthe

Files à deux bouts

13.
```
let ajoute_debut f a =
  let cap = Array.length f.contenu in
  if f.taille < cap
  then begin
       f.debut <- (f.debut - 1 + cap) mod cap;
       f.contenu.(f.debut) <- a;
       f.taille <- f.taille + 1 end
  else failwith "La file est pleine";;
```

14.
```
let retire_debut f =
  let cap = Array.length f.contenu in
  if f.taille > 0
  then begin
       let a = f.contenu.(f.debut)
       f.debut <- (f.debut + 1) mod cap;
       f.taille <- f.taille - 1;
       a end
  else failwith "La file est vide";;
```

Parcours en largeur

Il n'est pas demandé d'écrire la fonction, elle sera améliorée ensuite, mais en voici une écriture possible.

```
let minimum g src dst =
  let distance = Array.make g.n (-1) in
  let f = file_vide g.n in
  ajoute_fin f src;
  distance.(src) <- 0;
  let v = ref src in
  let traiter s =
    if distance.(s) = -1
    then (distance.(s) <- distance.(!v) + 1;
          ajoute_fin f s) in
  while f.taille > 0 || !v <> dst do
    List.iter traiter g.adj.(!v);
    v := retire_debut f done;
  distance.(dst));;
```

15. On note $d(s)$ la longueur minimale d'un chemin de `src` vers s, sa distance.

    On montre qu'à chaque moment

    (a) la file d'attente contient des sommets s_1, s_2, ..., s_p (dans l'ordre de début à la fin) tels que $d(s_1) \leq d(s_2) \leq \cdots \leq d(s_p) \leq d(s_1) + 1$,

    (b) tous les sommets de distance $d(s_1)$ ou moins ont été insérés,

    (c) 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.

    (a) 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 $d(s_2) \leq \cdots d(s_p) \leq d(s_1) + 1 = d(t_1) \cdots = d(t_q) = d(s_1) + 1 \leq d(s_2) + 1$.

    (b) 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.

    (c) 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.

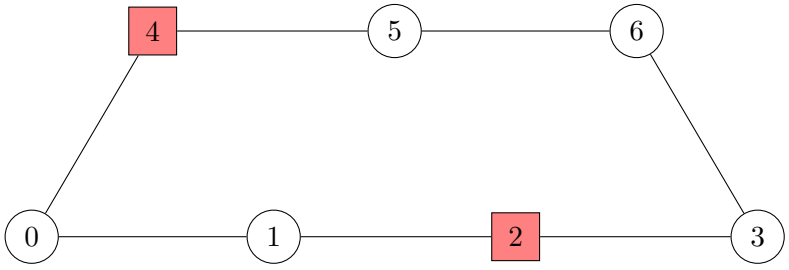
    Comme le graphe est connexe, le sommet `dst` est atteint donc la valeur de `distance.(dest)` est bien la valeur minimale des longueur des chemins de `src` à `dst`.

En croisant un minimum de monstres

16. La rédaction de la fonction est différente de celle proposée ci-dessus, en particulier on ne teste pas si la file est vide car on sait qu'on va trouver `dst` ; on peut donc utiliser une récursivité.

```
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 ();;
```

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

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

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.
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 $n.m_1 + l_1 < n.m_1 + n = n.(m_1 + 1) \leq n.m_2 \leq n.m_2 + l_2$
    - 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.