

DS 2

Devoir sur table

(4 heures)

Le sujet a pour but l'étude de labyrinthes.

Dans tous le sujet, on considère des graphes non orientés, sans boucles. On note $v \longleftrightarrow w$ l'arête reliant les sommets v et w .

La représentation d'un graphe en OCaml est donné par le type suivant :

```
type graphe = { n : int ; (* les sommets sont 0,1,...,n-1 *)
adj : int list array (* adj.(v) est la liste des voisins de v *)
};;
```

Les sommets sont les entiers $\{0, 1, \dots, n - 1\}$. Pour un sommet v du graphe g , la liste d'adjacence $g.adj.(v)$ contient tous les voisins de v dans un ordre arbitraire et sans doublon.

I. Fonctions utiles sur les graphes

Q1| Écrire la fonction `ajout_sans_doublon` qui prend en argument une liste l (sans doublon) et un élément e , et renvoie la liste sans doublon constituée des éléments de la liste l et de l'élément e .

```
ajout_sans_doublon : 'a list -> 'a -> 'a list
```

Q2| Écrire la fonction `ajoute_arete` qui prend en argument un graphe g et deux entiers i et j (avec $0 \leq i \leq g.n - 1$ et $0 \leq j \leq g.n - 1$). La fonction ajoute l'arête $i \longleftrightarrow j$ si elle n'était pas déjà présente.

```
ajoute_arete : graphe -> int -> int -> unit
```

Q3| Écrire la fonction `graphe_vide` qui prend en argument un entier n et renvoie un graphe à n sommets sans aucune arête.

```
graphe_vide : int -> graphe
```

Q4| Écrire la fonction `aretes` qui prend en argument un graphe, et renvoie la liste des couples (i, j) , tel que $i \longleftrightarrow j$ soit une arête de g . (Le graphe étant non orienté, si (i, j) est dans la liste, alors (j, i) y est aussi.)

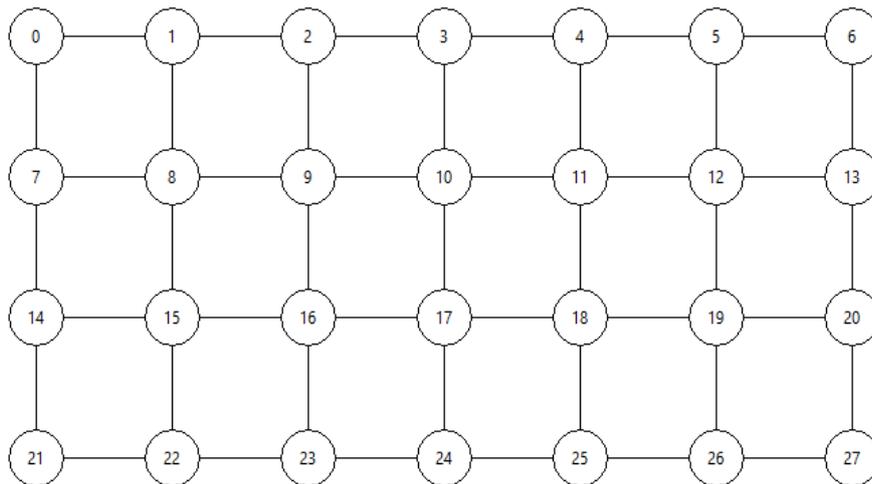
```
aretes : graphe -> (int * int) list
```

II. Construction d'un labyrinthe

Pour deux entiers n et m , la grille de taille $n \times m$ est le graphe dont les sommets sont $\{0, 1, \dots, n \times m - 1\}$ et dont les arêtes sont les paires de sommets qui sont :

- soit de la forme $v \longleftrightarrow (v + 1)$ pour $0 \leq v < nm$ tel que v modulo m est distinct de $m - 1$,
- soit de la forme $v \longleftrightarrow (v + m)$ pour $0 \leq v < (n - 1)m$.

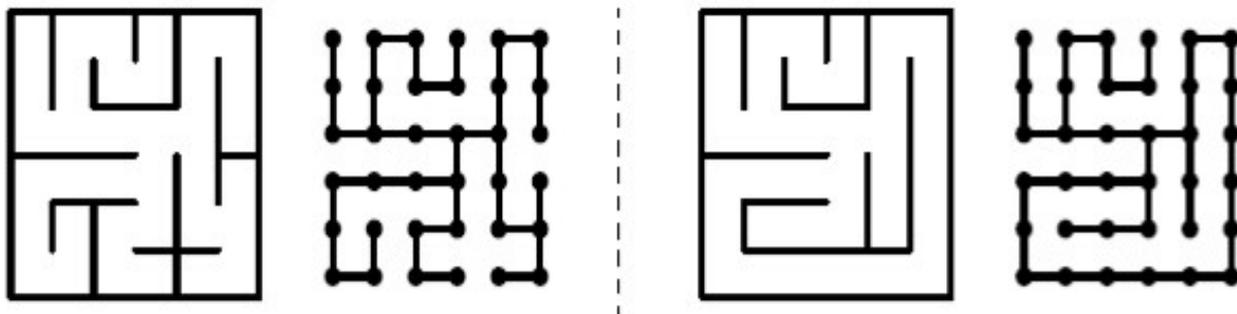
Par exemple, la grille de taille 4×7 est le graphe suivant :



On se donne un graphe g connexe. On appelle **labyrinthe** sur g un sous-graphe connexe de g qui a le même ensemble de sommets que g .

Par exemple, on peut partir d'un graphe g dont les sommets sont les cases d'une grille rectangulaire et dont les arêtes correspondent aux cases adjacentes (nord, sud, est, ouest). Les arêtes du labyrinthe correspondent alors aux ouvertures permettant de passer d'une case à une autre. Les arêtes de g qui ne sont pas dans le labyrinthe correspondent aux murs.

Voici deux exemples :



On dit qu'un labyrinthe est parfait lorsqu'il existe un et un seul chemin entre toute paire de sommets. La labyrinthe de gauche dans l'exemple ci-dessus est parfait, alors que celui de droite ne l'est pas. Dans cette partie, on cherche à construire des labyrinthes parfaits.

- Q5| Écrire la fonction `grille`, qui prend en argument deux entiers n et m , et renvoie le graphe correspondant à la grille de taille $n \times m$.

```
grille : int -> int -> graphe
```

- Q6| Afin d'obtenir un mélange aléatoire d'une liste, on va écrire deux fonctions :
La première fonction, `i_eme_element`, prend en argument une liste l et un indice i (nécessairement compris entre 0 et la taille de la liste moins 1) et retourne le couple $(element, reste)$ qui représente l'élément en position i , et le reste de la liste.

```
i_eme_element : 'a list -> int -> 'a * 'a list
```

Écrire la fonction `melange_liste` qui reçoit en argument une liste et permute aléatoirement les éléments de la liste. (rappel : la fonction `Random.int` prend un argument n et renvoie un entier aléatoire entre 0 et $n - 1$.)

```
melange_liste : 'a list -> 'a list
```

- Q7| Pour construire un labyrinthe aléatoire sur un graphe g , on peut utiliser l'algorithme suivant :

On effectue un parcours en profondeur (noté DFS par la suite) de graphe g . Lorsqu'un sommet v est traité, on permute aléatoirement la liste de ses voisins et on la parcourt. Pour chaque voisin w de v qui n'a pas encore été visité, on ajoute l'arête $v \longleftrightarrow w$ au labyrinthe et on traite w .

Écrire la fonction `labyrinthe1` qui prend en argument un graphe g connexe et renvoie un labyrinthe sur g construit avec un DFS aléatoire. On suggère d'écrire le parcours en profondeur comme une fonction récursive.

```
labyrinthe1 : graphe -> graphe
```

- Q8| Montrer que le labyrinthe construit par la fonction `labyrinthe1` est parfait.
Indication : identifier un invariant du parcours en profondeur réalisé par `labyrinthe1`.

III. Classe disjointes

On s'intéresse maintenant à une structure de données qui sera utilisée plus loin pour construire des labyrinthes par d'autres algorithmes. Cette structure de données représente une relation d'équivalence sur l'ensemble $\{0, 1, \dots, n - 1\}$. Pour cela on choisit un représentant arbitraire dans chaque classe d'équivalence et on se donne un tableau `lien` de taille n qui va permettre de retrouver ce représentant. Pour un représentant r , on a `lien.(r) = r`. Pour tout autre élément i de la classe de r , on aboutit à r en suivant les valeurs données par le tableau `lien` (c'est-à-dire que `lien.(i) = r`, ou `lien.(lien.(i)) = r`, ou `lien.(lien.(lien.(i))) = r`, etc). En particulier, i et `lien.(i)` sont toujours dans la même classe d'équivalence. Par exemple, le tableau

```
lien = [ 2 | 1 | 5 | 3 | 3 | 5 | 5 ]
```

représente la relation d'équivalence dont les classes sont $\{0, 2, 5, 6\}$, $\{1\}$ et $\{3, 4\}$, pour lesquelles on a choisis les représentants 5, 1 et 3.

On se donne le type OCaml suivant pour cette structure de données :

```
type classes_disjointes = {
  lien : int array ;
}
```

Q9| Écrire une fonction `cd_trouve` qui prend en arguments une relation d'équivalences sur $\{0, 1, \dots, n-1\}$ et un entier i , avec $0 \leq i < n$, et renvoie le représentant de la classe de i .

```
cd_trouve : classes_disjointes -> int -> int
```

On note que la complexité de `cd_trouve` dans le pire des cas est proportionnelle à la longueur du plus long chemin d'un élément à son représentant. Formellement, la longueur du chemin d'un élément i à son représentant r est définie comme le nombre d'éléments rencontrés en partant de i et en suivant les valeurs du tableau `lien` (en comptant i lui-même mais sans compter r).

On veut maintenant pouvoir modifier la relation d'équivalence, en offrant une opération `cd_union` permettant de fusionner la classes d'équivalence de deux éléments i et j donnés. L'idée consiste à chercher les représentants r_i et r_j des classes d'équivalence de i et j , avec la fonction `cd_trouve`, puis de faire pointer l'un vers l'autre en modifiant le tableau `lien`. Le choix du représentant r_i ou r_j pour la classe fusionnée n'est pas anodin, car il affecte le coût des opération `cd_trouve` ultérieures.

On introduit donc un second tableau `rang` de taille n . Pour chaque représentant r , la valeur de `rang.(r)` donne la longueur du plus long chemin d'un élément de la classe de r à r . Pour un élément i qui n'est pas un représentant, la valeur de `rang.(i)` n'est pas significative et ne sera jamais utilisée. Par exemple, les tableaux

`lien =`

2	1	5	3	3	5	5
---	---	---	---	---	---	---

`et` `rang =`

0	0	1	1	0	2	0
---	---	---	---	---	---	---

représentent la relation d'équivalence dont les classes sont $\{0, 2, 5, 6\}$, $\{1\}$ et $\{3, 4\}$, mais les valeurs du tableau `rang` en position 0, 2, 4, 6 ne sont pas significatives.

On modifie donc le type `classes_disjointes` de la façon suivante :

```
type classes_disjointes = {
  lien : int array ;
  rang : int array ;
}
```

Le code de la fonction `cd_trouve` reste inchangé.

Q10| Écrire une fonction `cd_union` qui prend en argument une relation d'équivalences sur $\{0, 1, \dots, n-1\}$ et deux entiers i et j , avec $0 \leq i, j < n$, et fusionne les classes d'équivalence de i et j . Lorsque i et j sont déjà dans la même classe, cette fonction ne fait rien. Sinon, elle choisit pour nouveau représentant celui dont le rang est maximal.

```
cd_union : classes_disjointes -> int -> int -> unit
```

Q11| On appelle rang d'une classe le rang de son représentant. Montrer que la fonction `cd_union` préserve les deux invariants suivants :

- (a) toute classe de rang k possède au moins 2^k éléments ;
- (b) dans une classe de rang k , la longueur du plus long chemin jusqu'au représentant est égale à k .

On se donne une fonction :

```
cd_init : int -> classes_disjointes
```

qui prend en argument un entier $n \geq 0$ et renvoie une relation d'équivalence sur $\{0, 1, \dots, n-1\}$ dont toutes les classes sont des singletons. Pour tout i , on a donc `lien.(i) = i` et `rang.(i) = 0`.

Q12| Dédurre de la question 7 que, pour une relation d'équivalence sur $\{0, 1, \dots, n-1\}$ construite à partir de `cd_init`, la complexité de `cd_trouve` est en $O(\log n)$.

IV Construction à l'aide des classes disjointes

On utilise maintenant la structure de classes disjointes pour construire un labyrinthe parfait `h` à partir d'un graphe `g` connexe à n sommets.

Le labyrinthe procède ainsi ;

- 1- On construit une relation d'équivalence sur $\{0, 1, \dots, n-1\}$ avec `cd_init` ;
- 2- On construit la liste de toutes les arêtes du graphe `g`, puis on la mélange ;
- 3- On parcourt la liste mélangée et, pour chaque arête $v \longleftrightarrow w$, si v et w ne sont pas dans la même classe d'équivalence, on ajoute l'arête $v \longleftrightarrow w$ au labyrinthe `h` et on fusionne les classes de v et w avec `cd_union` ;

Q13| Écrire une fonction `labyrinthe2` qui prend en argument un graphe `g` connexe et renvoie un labyrinthe sur `g` construit avec cet algorithme.

```
labyrinthe2 : graphe -> graphe
```

Q14| Montrer que l'étape 3 de l'algorithme maintient l'invariant suivant : pour toute classe d'équivalence X , le sous graphe `h` induit par X est un labyrinthe parfait du sous-graphe de `g` induit par X . En déduire que le labyrinthe construit par la fonction `labyrinthe2` est parfait.

V Résoudre un labyrinthe

On se pose maintenant la question de résoudre un labyrinthe, c'est-à-dire de chercher un chemin d'un sommet source `src` donné à un sommet destination `dst` donné. On suppose toujours que le labyrinthe est connexe. En revanche, on ne suppose pas le labyrinthe parfait, c'est-à-dire qu'il peut exister plusieurs chemins de la source à la destination.

Files à deux bouts.

On commence par programmer une structure de données de file à deux bouts . Une telle file est représentée par le type suivant :

```
type file = {
  contenu : int array ;
  mutable debut : int ;
  mutable taille : int
}
```

La file contient `taille` éléments, stockés dans le tableau `contenu` à partir de l'indice `debut` et dans le sens des indices croissants. Le tableau a un `taille cap`, qui est la capacité maximale de la file. Si $\text{debut} + \text{taille} \leq \text{cap}$, les éléments de la file sont stockés de façon consécutive dans le tableau `contenu`, entre `debut` et `debut + taille - 1`. Sinon, le stockage des éléments se poursuit au début du tableau `contenu`, jusqu'à la position $\text{debut} + \text{taille} - 1 - \text{cap}$. Dans tous les cas, on a les inégalités suivantes : $0 \leq \text{debut} < \text{cap}$ et $0 \leq \text{taille} \leq \text{cap}$.

Voici deux exemples de files de capacité 7 contenant 4 éléments (notés *X*) :

		X	X	X	X	
--	--	---	---	---	---	--

cap = 7 , debut = 2 ,taille = 4

X	X					X	X
---	---	--	--	--	--	---	---

cap = 7 , debut = 5 ,taille = 4

La file est construite par une fonction

```
file_vide : int -> file
```

qui prend la capacité en argument, puis manipulée avec les quatre opérations suivantes :

```
ajoute_debut :file -> int -> unit
retire_file : file -> int
ajoute_fin : file -> int -> unit
retire_fin : file -> int
```

L'appel de `ajoute_debut f e` ajoute l'élément `e` au début de la file `f` (sans déplacer ou modifier les éléments déjà présents), en supposant que la file n'est pas pleine. L'appel de `retire_debut f` retire et renvoie l'élément du début de la file `f` (sans déplacer ou modifier les autres éléments), en supposant que la file n'est pas vide. Les deux autres opérations `ajoute_fin` et `retire_fin` opèrent de manière similaire à la fin de la file.

Q15| Donner le code de la fonction `ajoute_debut`.

Q16| Donner le code de la fonction `retire_debut`.

Parcours en largeur

Pour trouver la longueur d'un plus court chemin dans un labyrinthe à n sommets, on effectue un parcours en largeur, en partant du sommet source `src` et en s'arrêtant quand on trouve de sommet destination `dst`. On procède ainsi :

- (a) créer un tableau `distance` de taille n , initialisé avec la valeur -1 dans toutes les cases ;
- (b) créer une file à deux bouts `f` de capacité n ;
- (c) ajouter le sommet source `src` dans `f` et initialiser `distance.(src)` à 0 ;
- (d) tant que la file `f` n'est pas vide :
 - i. retirer l'élément v au début de `f` ;
 - ii. si $v = \text{dst}$ alors on a terminé et la réponse est `distance.(dst)` ;
 - iii. sinon, pour chaque voisin w de v pour lequel `distance.(w)` vaut -1 ; ajouter w à la fin de `f` et donner à `distance.(w)` la valeur `distance.(v) + 1`.

Q17| Montrer que cet algorithme renvoie bien la longueur d'un plus court chemin de la source à la destination.

Q18| Écrire une fonction `resolution` qui prend en argument un graphe `g` et deux entiers `src` et `dst`, et renvoie la longueur d'un plus court chemin de la source à la destination.

`resolution : graphe -> int -> int -> int`