Constructions et explorations de labyrinthes

Nous nous intéressons ici à l'étude de labyrinthes. La partie I propose plusieurs méthodes pour construire des labyrinthes parfaits. La partie II étudie des algorithmes pour explorer des labyrinthes en présence de monstres. Les deux parties peuvent être traitées indépendamment.

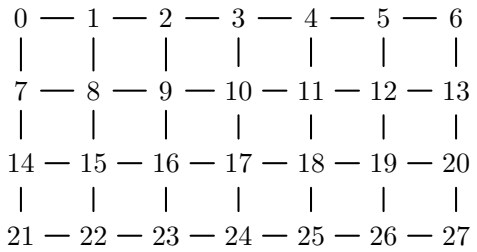
Graphes. Dans tout ce sujet, on considère des graphes non orientés, sans boucles. On note $\mathtt{v} — \mathtt{w}$ l'arête reliant les sommets `v` et `w`. Un chemin de longueur $\mathtt{n} \geq 0$ du sommet $\mathtt{v}_0$ au sommet $\mathtt{v}_\mathtt{n}$ est une séquence $\mathtt{v}_0 — \mathtt{v}_1 — \cdots — \mathtt{v}_{\mathtt{n}-1} — \mathtt{v}_\mathtt{n}$ de sommets reliés par des arêtes. Dans tout l'énoncé, les chemins considérés sont simples, c'est-à-dire que leurs sommets sont tous distincts. Un graphe `g` est connexe si toute paire de sommets de `g` est reliée par un chemin.

Un sous-graphe d'un graphe `g` est un graphe dont l'ensemble des sommets est un sous-ensemble de celui de `g` et l'ensemble des arêtes est un sous-ensemble de celui de `g`. Pour un graphe `g` et un sous-ensemble `X` de sommets de `g`, on définit le sous-graphe de `g` induit par `X` comme le graphe dont les sommets sont `X` et dans lequel deux sommets sont reliés s'ils sont reliés dans `g`.

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

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

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



Langage OCaml. On peut créer des tableaux avec les deux fonctions `Array.make` et `Array.of_list`. L'appel de `Array.make n x` crée un tableau contenant `n` copies de l'élément `x`. L'appel de `Array.of_list l` crée un tableau contenant, dans l'ordre, les éléments d'une liste `l`. Les cases d'un tableau sont numérotées à partir de 0. La fonction `Array.length` renvoie la taille d'un tableau. Pour un tableau `a`, on accède à l'élément d'indice `i` avec `a.(i)` et on le modifie avec `a.(i) <- v`.

La fonction `List.iter: ('a -> unit) -> 'a list -> unit` est telle que l'appel de `List.iter f [a1 ; ... ; an]` applique la fonction `f` aux éléments `a1`, ..., `an` dans cet ordre. Elle est donc équivalente à `begin f a1; f a2; ...; f an; () end`.

Pour un entier $\texttt{n} > 0$, l'appel `Random.int n` renvoie un entier tiré aléatoirement dans l'ensemble $\{0, 1, \ldots, \texttt{n} - 1\}$. On supposera que ce tirage est uniforme (tous les entiers sont équiprobables).

Représentation d'un graphe en OCaml. On représente un graphe en OCaml avec 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 $\{0, 1, \ldots, \texttt{n} - 1\}$. Pour un sommet `v`, la liste d'adjacence `adj.(v)` contient tous les voisins de `v` dans un ordre arbitraire et sans doublons. On suppose donnée une fonction

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

telle que pour $0 \leq \texttt{v} \leq \texttt{g.n} - 1$ et $0 \leq \texttt{w} \leq \texttt{g.n} - 1$, l'appel de `ajoute_arete g v w` ajoute au graphe `g`, si elle n'est pas déjà présente, une arête entre les sommets `v` et `w`, c'est-à-dire ajoute `v` à `g.adj.(w)` et `w` à `g.adj.(v)`. On se donne également une fonction

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

telle que `graphe_vide n` renvoie un nouveau graphe à `n` sommets et sans aucune arête (toutes les listes `adj.(v)` sont vides). On se donne enfin une fonction

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

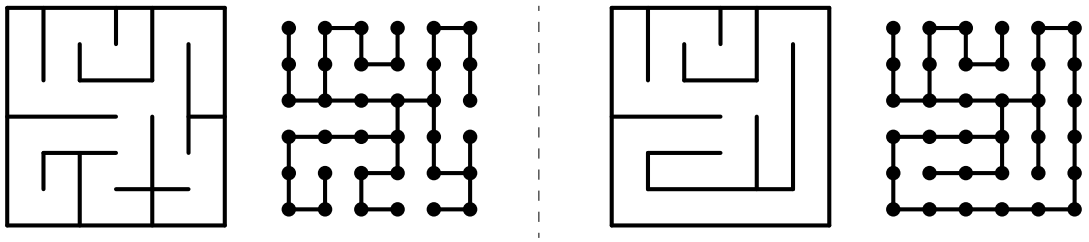
qui renvoie un tableau contenant toutes les arêtes d'un graphe donné, sous la forme de paires d'entiers. Le graphe étant non orienté, la paire $(\texttt{v}, \texttt{w})$ et la paire $(\texttt{w}, \texttt{v})$ apparaissent toutes les deux dans ce tableau.

Complexité. Par *complexité* (en temps) d'un algorithme A on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc) nécessaires à l'exécution de A dans le cas le pire. Lorsque la complexité dépend d'un ou plusieurs paramètres $\kappa_0, \cdots, \kappa_{r-1}$, on dit que A a une complexité en $\mathcal{O}(f(\kappa_0, \cdots, \kappa_{r-1}))$ s'il existe

une constante $C > 0$ telle que, pour toutes les valeurs de $\kappa_0, \cdots, \kappa_{r-1}$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $\kappa_0, \cdots, \kappa_{r-1}$, la complexité est au plus $C\, f(\kappa_0, \cdots, \kappa_{r-1})$.

Partie I. Construire des labyrinthes

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. Le 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.

Mélange de Knuth. On commence par décrire l'algorithme du *mélange de Knuth* qui permet de permuter aléatoirement les éléments d'un tableau, et qui sera utilisé dans les trois algorithmes de construction de labyrinthe. Pour un tableau `a` de taille `n`, cet algorithme effectue `n` échanges : pour $0 \leq \texttt{i} < \texttt{n}$, la `i`-ième étape échange l'élément `a.(i)` avec l'élément `a.(j)`, pour un entier `j` pris au hasard entre 0 et `i` *inclus*.

Question 1. Écrire une fonction `melange_knuth: int array -> unit` qui reçoit en argument un tableau et le permute aléatoirement avec le mélange de Knuth.

On souhaite montrer que le mélange de Knuth effectue une permutation aléatoire uniforme du tableau, c'est-à-dire que toutes les permutations sont équi-probables.

Question 2. Soit `a` le tableau de taille `n` dans lequel `a.(p)` $=$ `p` pour tout `p`. On appelle la fonction `melange_knuth` sur le tableau `a`, et on note $x_{\texttt{p}}$ la valeur de la `p`-ième case de `a` après l'appel. Montrer que la probabilité que $x_{\texttt{p}}$ soit égal à `q`, pour $0 \leq \texttt{p} < \texttt{n}$ et $0 \leq \texttt{q} < \texttt{n}$, vaut

$$\Pr(x_{\texttt{p}} = \texttt{q}) = \frac{1}{\texttt{n}}\,.$$

On suppose que `Random.int` renvoie un entier aléatoire uniforme.
Indication : en notant toujours $x_{\texttt{p}}$ la valeur courante de la `p`-ième case du tableau à l'étape `i`, montrer que l'algorithme satisfait l'invariant de boucle :

1. pour tous $0 \leq \texttt{p} < \texttt{i}$ et $0 \leq \texttt{q} < \texttt{i}$, on a $\Pr(x_{\texttt{p}} = \texttt{q}) = \frac{1}{\texttt{i}}$,
2. pour tout $\texttt{i} \leq \texttt{p} < \texttt{n}$, on a $x_{\texttt{p}} = \texttt{p}$.

Parcours en profondeur aléatoire. On rappelle qu'on considère un graphe `g` connexe et qu'on appelle labyrinthe sur `g` un sous-graphe connexe de `g` qui a le même ensemble de sommets que `g`. Pour construire un labyrinthe aléatoire sur `g`, on peut utiliser l'algorithme suivant. On effectue un *parcours en profondeur* (noté DFS par la suite) du 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`—`w` au labyrinthe et on traite `w`.

Question 3. Écrire une fonction `labyrinthe1: graphe -> graphe` 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. On ne demande pas de vérifier que `g` est connexe.

Question 4. Montrer que le labyrinthe construit par la fonction `labyrinthe1` est parfait.
Indication : identifier un invariant du parcours en profondeur réalisé par `labyrinthe1`.

Classes 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 $\{0, 1, \ldots, \mathtt{n}-1\}$. 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 choisi les représentants 5, 1 et 3. On se donne le type OCaml suivant pour cette structure de données :

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

Question 5. Écrire une fonction `cd_trouve: classes_disjointes -> int -> int` qui prend en arguments une relation d'équivalence sur $\{0, 1, \ldots, \mathtt{n}-1\}$ et un entier `i`, avec $0 \leq \mathtt{i} < \mathtt{n}$, et renvoie le représentant de la classe de `i`.

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 les classes d'équivalence de deux éléments `i` et `j` donnés. L'idée consiste à chercher les représentants `ri` et `rj` 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 `ri` ou `rj` pour la classe fusionnée n'est pas anodin, car il affecte le coût des opérations `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 positions $0, 2, 4, 6$ 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.

Question 6. Écrire une fonction `cd_union: classes_disjointes -> int -> int -> unit` qui prend en arguments une relation d'équivalence sur $\{0, 1, \ldots, \mathtt{n} - 1\}$ et deux entiers `i` et `j`, avec $0 \leq \mathtt{i}, \mathtt{j} < \mathtt{n}$, 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.

Question 7. On appelle rang d'une classe le rang de son représentant. Montrer que la fonction `cd_union` préserve les deux invariants suivants :

1. tout classe de rang k possède au moins 2^k éléments ;
2. 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 $\mathtt{n} \geq 0$ et renvoie une relation d'équivalence sur $\{0, 1, \ldots, \mathtt{n} - 1\}$ dont toutes les classes sont des singletons. Pour tout `i`, on a donc `lien.(i)` $= \mathtt{i}$ et `rang.(i)` $= 0$.

Question 8. Déduire de la question 7 que, pour une relation d'équivalence sur $\{0, 1, \ldots, \mathtt{n}-1\}$ construite à partir de `cd_init` et uniquement avec des appels à `cd_union`, la complexité de `cd_trouve` est en $O(\log \mathtt{n})$.

Application à la construction d'un labyrinthe. On utilise maintenant la structure de classes disjointes pour construire un labyrinthe parfait `h` à partir d'un graphe `g` connexe à `n` sommets. L'algorithme procède ainsi :

1. on construit une relation d'équivalence sur $\{0, 1, \ldots, \mathtt{n}-1\}$ avec `cd_init` ;
2. on construit le tableau de toutes les arêtes du graphe `g`, puis on le mélange avec `melange_knuth` ;
3. on parcourt ce tableau mélangé et, pour chaque arête `v`—`w`, si `v` et `w` ne sont pas dans la même classe d'équivalence, on ajoute l'arête `v`—`w` au labyrinthe `h` et on fusionne les classes de `v` et `w` avec `cd_union`.

Question 9. Écrire une fonction `labyrinthe2: graphe -> graphe` qui prend en argument un graphe `g` connexe et renvoie un labyrinthe sur `g` construit avec cet algorithme. On rappelle l'existence de la fonction `aretes` qui renvoie un tableau contenant toutes les arêtes d'un graphe donné, sous la forme de paires d'entiers. On ne demande pas de vérifier que `g` est connexe.

Question 10. Montrer que l'étape 3 de l'algorithme maintient l'invariant suivant : pour toute classe d'équivalence X, le sous-graphe de `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.

Algorithme d'Eller. On considère maintenant un troisième algorithme pour construire un labyrinthe parfait, dans le cas particulier où le graphe `g` de départ est la grille de taille $\mathtt{n} \times \mathtt{m}$ définie dans le préambule. Comme pour l'algorithme précédent, on utilise une relation d'équivalence sur les sommets $\{0, 1, \ldots, \mathtt{n} \times \mathtt{m} - 1\}$ de la grille `g`. Initialement, toutes les classes d'équivalence sont des singletons. Dans la suite, on dit qu'on *connecte* deux sommets `i` et `j`, voisins dans `g`, pour dire qu'on fusionne leurs classes d'équivalence avec `cd_union` et qu'on ajoute l'arête `i`—`j` dans le labyrinthe en construction.

L'algorithme d'Eller procède en balayant la grille de haut en bas :

1. pour chaque ligne, sauf la dernière :
   (a) on parcourt toutes les arêtes `v`—`(v+1)` de cette ligne, dans un ordre aléatoire. Si les sommets `v` et `v+1` ne sont pas dans la même classe d'équivalence, on les connecte avec probabilité $1/2$;
   (b) on choisit un sous-ensemble aléatoire de sommets de cette ligne qui contient au moins un sommet de chaque classe d'équivalence. On relie chacun des sommets de ce sous-ensemble avec le sommet situé juste en dessous sur la ligne suivante.

2. on parcourt toutes les arêtes $\mathtt{v}$—$(\mathtt{v}{+}1)$ de la dernière ligne, dans un ordre aléatoire. On connecte les sommets $\mathtt{v}$ et $\mathtt{v}+1$ si ils ne sont pas dans la même classe d'équivalence.

Question 11. Montrer que l'algorithme d'Eller construit un labyrinthe parfait.

Question 12. Prouver que tout labyrinthe parfait sur la grille $\mathtt{g}$ peut être obtenu par l'algorithme d'Eller.

Partie II. 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. Dans cette partie, on considère trois variantes de la recherche d'un chemin.

Files à deux bouts. On commence par programmer une structure de données de file à deux bouts, qui sera utilisée dans les trois algorithmes de recherche de chemin. 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 une taille `cap`, qui est la capacité maximale de la file. Si `debut` + `taille` $\leq$ `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 `debut` + `taille` $-1-$ `cap`. Dans tous les cas, on a les inégalités suivantes : $0 \leq$ `debut` $<$ `cap` et $0 \leq$ `taille` $\leq$ `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_debut: 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 opérations `ajoute_fin` et `retire_fin` opèrent de manière similaire à la fin de la file.

Question 13. Donner le code de la fonction `ajoute_debut`.

Question 14. 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 le sommet destination `dst`. On procède ainsi :

1. créer un tableau `distance` de taille `n`, initialisé avec la valeur -1 dans toutes les cases ;
2. créer une file à deux bouts `f` de capacité `n` ;
3. ajouter le sommet source `src` dans `f` et initialiser `distance.(src)` à 0 ;
4. tant que la file `f` n'est pas vide :
   (a) retirer l'élément `v` au début de `f` ;
   (b) si `v` $=$ `dst` alors on a terminé et la réponse est `distance.(dst)` ;
   (c) 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`.

Question 15. Montrer que cet algorithme renvoie bien la longueur d'un plus court chemin de la source `src` à la destination `dst`. On rappelle que le graphe est supposé connexe.

En croisant un minimum de monstres. On suppose maintenant qu'il y a des monstres sur certains sommets du labyrinthe (les sommets `src` et `dst` pouvant, comme les autres, accueillir des monstres). Notre but est maintenant de chercher des chemins du sommet source `src` au sommet destination `dst` qui passent par un minimum de monstres (en comptant les monstres sur `src` et `dst` s'il y en a). Pour cela, on peut modifier l'algorithme de parcours en largeur donné plus haut de la façon suivante :

- le tableau `distance` contient maintenant, pour chaque sommet `v` déjà atteint, un couple (m, ℓ) où m est le nombre de monstres et ℓ la longueur d'un chemin de `src` à `v` qui passe par un minimum de monstres ;
- quand on atteint un nouveau sommet `w` pour la première fois, on le rajoute à la fin de la file `f` s'il y a un monstre sur le sommet `w` et au début sinon.

Question 16. Compléter le code de la fonction `minimum_monstres: graphe -> bool array -> int -> int -> int * int` ci-dessous qui implémente cet algorithme. Les quatre arguments sont le graphe `g`, un tableau de booléens `monstre` qui indique pour chaque sommet s'il y a un monstre, et les sommets source `src` et destination `dst`. La fonction renvoie le couple (m, ℓ) où m est le nombre de monstres et ℓ la longueur d'un chemin de `src` à `v` qui passe par un minimum de monstres.

```
let minimum_monstres g monstre src dst =
  let distance = Array.make g.n (-1,-1) in   (* 1 *)
  let f = file_vide g.n in                   (* 2 *)
  ...                                        (* 3 *)
  let rec loop () =                          (* 4 *)
    let v = retire_debut f in                  (* a *)
    if v = dst then distance.(v) else begin    (* b *)
      List.iter (fun w ->                      (* c *)
        ...            (* pour chaque voisin w de v *)
      ) g.adj.(v);
      loop ()
    end in
  loop ()
```

La valeur initiale `(-1,-1)` dans le tableau `distance` est utilisée pour indiquer qu'un sommet n'a pas encore été vu. La syntaxe de la fonction `List.iter` est rappelée dans le préambule. Les commentaires dans le code fourni font référence aux étapes de l'algorithme du parcours en largeur. On pourra se contenter de donner uniquement les morceaux de code correspondant aux étapes (3) et (4c).

Question 17. On admet que l'algorithme de la question 16 trouve un chemin qui passe par un minimum de monstres. En revanche, il ne donne pas nécessairement un chemin de longueur minimale parmi ceux qui passent par un minimum de monstres. Donner un exemple de labyrinthe pour lequel la fonction `minimum_monstres` trouve un chemin dont la longueur n'est pas minimale parmi ceux qui passent par un minimum de monstres.

Minimum de monstres et longueur minimale. On considère maintenant un algorithme qui cherche un chemin de longueur minimale parmi ceux qui passent par un minimum de monstres.

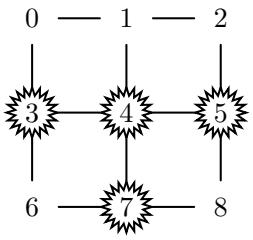
Le coût d'un chemin est défini comme la paire (m, ℓ) où m est le nombre de monstres le long de ce chemin (y compris à ses extrémités) et ℓ la longueur de ce chemin. On ordonne les coûts lexicographiquement : $(m_1, \ell_1) < (m_2, \ell_2)$ si et seulement si $m_1 < m_2$ ou $m_1 = m_2$ et $\ell_1 < \ell_2$. La distance entre deux sommets est définie comme le coût minimal d'un chemin entre ces sommets.

Pour trouver la distance entre la source et la destination, l'idée consiste à effectuer plusieurs parcours en largeur successifs, pour des nombres de monstres rencontrés de plus en plus grand. On fait un premier parcours en largeur sur les chemins qui ne passent par aucun monstre. Si la destination est atteinte, on a bien trouvé un plus court chemin (par la question 15) ne passant par aucun monstre. Sinon, on a trouvé tous les monstres à distance de type $(1, \ell)$. On démarre alors un nouveau parcours en largeur depuis les monstres à distance $(1, \ell)$ avec ℓ minimale. Pendant ce parcours, on prend soin d'insérer au début de la file d'attente du parcours en largeur chaque monstre à distance de type $(1, \ell)$ quand la longueur de chemin ℓ est atteinte par le parcours en largeur. Si la destination est atteinte, on a trouvé un plus court chemin passant par un seul monstre. Sinon, on a trouvé tous les monstres à distance de type $(2, \ell)$ et on continue.

Pour mettre en œuvre cet algorithme, on utilise trois files, `f`, `sources_courantes` et `sources_suivantes`. Par ailleurs, on utilise toujours un tableau `distance` qui contient maintenant, pour chaque sommet `v` déjà atteint, sa distance à la source sous la forme d'un couple (m, ℓ). On procède ainsi :

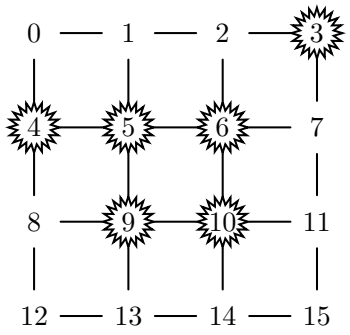
1. initialisation :
   - créer un tableau `distance` de taille `n`, initialisé avec la valeur $(-1, -1)$ dans toutes les cases,
   - créer trois files `f`, `sources_courantes` et `sources_suivantes` de capacité `n`,
   - ajouter le sommet source `src` dans `f` et initialiser `distance.(src)` à $(0, 0)$ s'il n'y a pas de monstre sur le sommet `src` et à $(1, 0)$ sinon ;
2. tant que la destination n'est pas atteinte :
   (a) si la file `f` est vide,
      i. si la file `source_courantes` est vide, déplacer tous les éléments de `sources_suivantes` dans `sources_courantes`,
      ii. déplacer tous les éléments de `sources_courantes` avec ℓ minimal dans `f` ;
   (b) retirer l'élément `v` au début de `f` et soit $(m, \ell) =$ `distance.(v)` ;
   (c) si `v` $=$ `dst` alors on a terminé et la réponse est (m, ℓ) ;
   (d) tant qu'il y a au début de `sources_courantes` un sommet à distance $(m, \ell + 1)$, le retirer de `sources_courantes` et l'ajouter à la fin de `f` ;
   (e) pour chaque voisin `w` de `v` pour lequel `distance.(w)` vaut $(-1, -1)$,
      - s'il n'y a pas de monstre sur `w`, ajouter `w` à la fin de `f` et donner à `distance.(w)` la valeur $(m, \ell + 1)$;
      - s'il y a un monstre sur `w`, ajouter `w` à la fin de `sources_suivantes` et donner à `distance.(w)` la valeur $(m + 1, \ell + 1)$.

Par exemple, on considère le labyrinthe de neuf sommets ci-dessous, avec `src` $= 0$ et `dst` $= 8$ et des monstres sur les sommets encerclés. Le tableau ci-dessous déroule l'algorithme sur cet exemple. La première colonne indique les étapes effectuées, la deuxième les affectations faites dans le tableau `distance` et les trois dernières l'état des trois files à la fin de chaque étape (on n'écrit rien lorsqu'il n'y a pas de changement).



étape	affectations `distance`	`f`	`sources_courantes`	`sources_suivantes`
1	$0 \leftarrow (0,0)$	0	$\emptyset$	$\emptyset$
2be	$1 \leftarrow (0,1)$ $3 \leftarrow (1,1)$	1		3
2be	$2 \leftarrow (0,2)$ $4 \leftarrow (1,2)$	2		3, 4
2be	$5 \leftarrow (1,3)$	$\emptyset$		3, 4, 5
2a		3	4, 5	$\emptyset$
2bde	$6 \leftarrow (1,2)$	4, 6	5	
2bde	$7 \leftarrow (2,3)$	6, 5	$\emptyset$	7
2be	$8 \leftarrow (1,4)$	6, 8		
2b		8		
2bc		$\emptyset$		

Question 18. Dérouler de la même façon l'algorithme sur le labyrinthe à 16 sommets suivant, où les monstres sont sur les sommets encerclés, et avec `src` $= 0$ et `dst` $= 11$:



Question 19. Montrer que cet algorithme renvoie bien la distance (pour le coût lexicographique défini plus haut) de la source `src` à la destination `dst`. On rappelle que le labyrinthe est supposé connexe. Le candidat pourra se contenter de donner une liste complète d'invariants permettant de déduire la correction de l'algorithme.

Question 20. Montrer qu'il est également possible de trouver un chemin de coût minimal dans le labyrinthe avec les monstres (pour le coût lexicographique défini plus haut) en construisant un graphe orienté pondéré et en utilisant un algorithme de plus court chemin (de type Dijkstra ou Floyd-Warshall).

* *
*