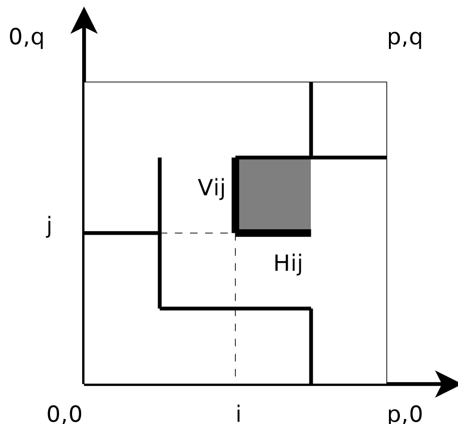
MPSI

Option Informatique
TP 5: Labyrinthes

David Baelde
david.baelde@ens-lyon.org

Nous allons nous attacher à générer aléatoirement de jolis labyrinthes, puis à les résoudre. On voit d'abord un labyrinthe de façon geométrique sur le plan $\mathbb{N}^2$ comme un ensemble de segments verticaux ou horizontaux inclus dans un certain carré délimitant le labyrinthe, dont on note par convention p la largeur et q la hauteur. On note H_{ij} le morceau de mur horizontal $[(i, j), (i + 1, j)]$; et V_{ij} le morceau vertical $[(i, j), (i, j + 1)]$. On interdit les murs inclus dans la frontière du carré, soit les H_{0j} et V_{i0}. Les cases sont les carrés de côté 1, on repèrera par convention une case pas la coordonnée de son coin inférieur gauche, soit (i, j) pour la case grisée sur le dessin.



Un labyrinthe sera aussi vu comme un graphe, dont les sommets sont les cases du labyrinthe, et les arêtes sont les absences de murs : deux cases voisines sont connectées s'il n'y a pas de mur entre elles.

Enfin, on représente en Caml un labyrinthe par un enregistrement, indiquant la taille du labyrinthe et la présence des arêtes : par exemple l'arête H_{ij} est présente dans `laby` si `laby.h.(i).(j) = true`.

```
type laby = { p : int ; q : int ;
              h : bool array array ; v : bool array array }
```

Je vous fournis un fichier contenant la définition du type ainsi qu'une fonction `new_laby : int -> int -> bool -> laby` pour créer un labyrinthe. `new_laby p q v` crée un labyrinthe de dimension (p,q) et initialise toutes les arêtes à `v`. De plus, je vous fournis un ensemble de fonctions pour dessiner des labyrinthes. Vous pourrez ainsi utiliser :

`fonction : type`	description
`Graph.draw : laby->unit`	affichage un labyrinthe
`Graph.wait : unit->unit`	attente d'un clic sur la fenêtre graphique
`Graph.pick : laby->int*int`	sélection d'une case par l'utilisateur
`Graph.mark : laby->int->int->unit`	coloriage d'une case du labyrinthe

1 Génération aléatoire

Nous allons générer aléatoirement un labyrinthe. Chercher à générer n'importe quel type de labyrinthe mène à des résultats assez moches ou peu intéressants. Nous chercherons donc des labyrinthes *connexes minimaux* : toute paire de sommets est reliée dans le graphe (connexité), et si on enlève une arête (ajoute un mur) la connexité est invalidée.

► Montrez qu'un tel graphe est sans cycle — un cycle est un chemin dans le graphe dont le point de départ et d'arrivée sont confondus.

► Quel type connu de graphe est caractérisé par la connexité et l'absence de cycles ?

Pour générer de tels labyrinthes nous partons d'un labyrinthe contenant tous les murs possibles. En enlevant petit à petit des murs, nous allons faire grandir un ensemble connexe minimal de cases I, jusqu'à ce qu'il recouvre tout le labyrinthe. Voici l'algorithme que nous allons implémenter : pour commencer, mettre dans I une case quelconque ; ensuite, tant qu'on peut, choisir au hasard une case de I qui a un voisin hors de I, et enlever le mur qui les sépare.

C'est dit simplement, mais c'est trop complexe algorithmiquement. Pour rendre cela raisonnablement efficace, nous allons maintenir tout au long de la construction l'ensemble des cases à la frontière de I : les cases de I qui ont un voisin hors de I.

Nous avons besoin de quatre opérations sur l'ensemble frontière : l'ajout et la suppression d'une case, le choix d'une case au hasard, et le test de vacuité. Une propriété essentielle va nous permettre de représenter la frontière de façon simple et efficace pour ces opérations : sa taille reste toujours bornée par le nombre de sommets. Nous allons donc créer un tableau de taille le nombre de sommets, et nous rangerons dans ce tableau les éléments de I, en stockant dans un `int ref` le premier index du tableau dont le contenu ne correspond pas à un élément de I.

Par exemple `(ref 0, [|1;3;4|])` correspond à l'ensemble vide, et `(ref 1, [|1;3;4|])` représente l'ensemble $\{1\}$.

► On définit : `type 'a set = int ref * 'a array`. Programmez les fonctions suivantes : création d'un ensemble (`int -> 'a -> 'a set`, où le second argument sert à initialiser le tableau), ajout d'un élément (`'a set -> 'a -> unit`), suppression d'un élément désigné par son index (`'a set -> int -> unit`), choix d'un élément au hasard (`'a set -> 'a`), et test de vacuité (`'a set -> bool`).

> **Le module `Random`** fournit de quoi faire des choix aléatoires. Vous avez à votre disposition `Random.int` qui étant donné un entier n renvoie un nombre au hasard entre 0 et $n-1$ inclus, ainsi que `Random.bool` qui renvoie un booléen au hasard. Pour ne pas avoir toujours le même résultat, il faut initialiser le générateur aléatoire, avec `Random.self_init ()`.

► Écrivez une fonction `set_remove_all` de type `'a set -> ('a -> bool) -> unit` telle que `set_remove_all set f` supprime tous les éléments `x` de `set` tels que `f x = true`.

On représente l'ensemble I par un tableau à deux dimensions contenant des booléens, `in.(i).(j)` indiquant si une case (i, j) est dans I.

► Écrire la fonction `neighbours_outside` qui étant donné un labyrinthe, un ensemble I et une case (i, j) renvoie la liste des voisins des (i, j) qui ne sont pas dans I.

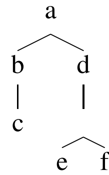
► Écrivez la génération aléatoire de labyrinthe, dont la boucle principale consiste à : choisir un élément dans la frontière, ajouter un de ses voisins à I en supprimant un mur, mettre à jour la frontière. Cette dernière étape consiste à ajouter la nouvelle case puis supprimer de la frontière les cases dont toutes les voisinnes sont désormais dans I. Ce filtrage pourra être fait ainsi : `set_remove_all front (fun e -> neighbours_outside e = [])` Vous testerez bien sûr en affichant graphiquement votre résultat.

► À voir rapidement ou à faire à la maison... Qu'est ce qui est inutilement coûteux ? Expliquer comment on peut améliorer l'algorithme en changeant les structures de données maintenues. Jusqu'à quelle taille pourrez-vous générer un labyrinthe en un temps raisonnable ?

2 Résolution d'un labyrinthe

Nous cherchons maintenant à relier deux points donnés dans un labyrinthe. Sur les labyrinthes générés par la procédure précédente il existe toujours un unique chemin, mais jouons le jeu et écrivons une procédure qui marche dans le cas général.

Chercher un chemin dans un graphe revient à le parcourir en partant d'un des points, en cherchant à atteindre l'autre. Il existe plusieurs types de parcours. Le parcours en *profondeur* : quand j'arrive à un noeud x, je choisis le premier voisin de x non exploré et je parcours le sous-graphe associé, puis je parcours le sous-graphe associé au second voisin non parcouru de y, etc. Le parcours en *largeur* parcourt tous les chemins à la fois : on part d'un ensemble F contenant un seul noeud, et à chaque étape on remplace l'ensemble F par l'ensemble des x voisins de F qui ne sont pas encore parcourus.



Sur cet exemple le parcours en profondeur visite les noeuds dans l'ordre a, b, c, d, e, f et le parcours en largeur énumère a, b, d, c, e, f.

Le parcours en largeur énumère les noeuds en s'éloignant petit à petit du noeud initial : on liste les noeuds à distance 1, puis 2, etc. Cela nous permet de chercher le plus court chemin d'un point à un autre, s'il en existe un. Cet algorithme est aussi appelé algorithme de la goutte d'eau : on imagine une goutte d'eau déposée en un point du labyrinthe, qui se propage en s'atténuant le long des couloirs du labyrinthe-buvard. Pour savoir si un point est accessible on regarde s'il est humidifié, le degré d'humidité indiquant la distance au point de départ.

► Implémenter l'algorithme de la goutte d'eau, en notant dans un tableau les distances au point de départ. Une fois atteint le point cherché, remontez au point de départ par le plus court chemin à l'aide du tableau des distances, et marquez dans la fenêtre graphique les cases de ce chemin.

> **Une extension possible** de la génération de labyrinthe est de ne pas partir d'un labyrinthe plein de murs, mais d'en enlever déja certains, selon un certain dessin. Du coup, l'extension de I ne se fera plus forcément case à case, puisque l'enlèvement d'un mur peut connecter à I plusieurs cases. Une fois cela pris en compte, l'algorithme devrait permettre de générer de jolis labyrinthes.