Labyrinthes

1 Introduction

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `labyrinthes` contenant une source OCaml :

```
curl cdn.sci-phy.org/mp2i/tp19-labyrinthes.tgz | tar xvz
```

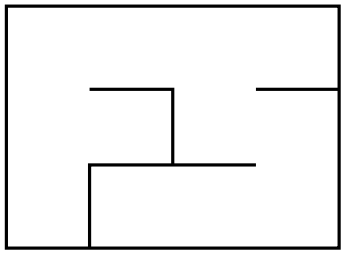
On y trouvera un fichier source `labyrinthes.ml` à compléter, contenant deux modules (l'un pour gérer des graphes non orientés, le second pour des classes d'équivalence), ainsi que quelques fonctions qui seront utiles dans cette séance.

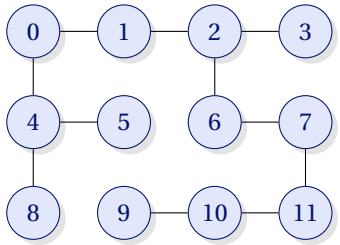
1.2 Objectifs

Nous nous intéresssons ici à la génération et à la résolution de labyrinthes. On se limite ici au cas de labyrintes tels que celui illustré ci-dessous : rectangulaires, consistant en un regroupement de « cases » carrées, donnant accès à certaines des cases immédiatement voisines (en 4-connexité).

Dans la suite, on notera h le nombre de cases du labyrinthe en hauteur, et w le nombre de cases en largeur.

D'un point de vue algorithmique, les labyrinthes seront modélisés comme des graphes non-orientés, où les « cases » sont numérotés de 0 à $h \times w - 1$ de la façon suivante :





Un labyrinthe est dit *parfait* s'il existe un chemin et un seul entre toute paire de cases (donc entre toute paire de sommets dans le graphe), un chemin étant défini comme une succession $s_0, s_1, \ldots, s_n$ de cases (de sommets dans le graphe) tous distincts deux à deux, et tels qu'il soit possible de passer de s_i à s_{i+1} (donc que les sommets correspondants soient voisins dans le graphe).

1.3 OCaml et affichage graphique

Dans ce TP, nous allons essayer d'afficher des labyrinthes à l'écran. Pour ce faire, OCaml dispose de modules graphiques, dont un (`graphics`) est inclus dans la bibliothèque standard, mais n'est plus distribué directement avec le compilateur depuis sa version `4.09`.

Pour activer `graphics` dans un mode interactif (Avec WinCaml, `ocaml`...), on dispose de plusieurs possibilités. Tout d'abord, on peut charger directement le module précompilé (s'il est disponible et que le compilateur est en mesure de le trouver) :

```
#load "graphics.cma"
```

Alternativement, on peut utiliser `ocamlfind` pour localiser et charger ledit module :

```
#use "topfind"
#require "graphics"
```

Les deux options sont présentes dans le fichier fourni, la seconde étant commentée. Vous êtes invité à activer celle adaptée à votre situation.

Si l'on travaille en compilant directement le fichier, nul besoin de ces commandes (on peut les supprimer) mais il faudra ajouter `graphics.cma` dans la liste des sources (en ajoutant éventuellement le chemin où le trouver grâce à l'option `-I` du compilateur).

Bien évidemment, le module `graphics` doit être installé. Si ce n'est pas le cas et que le gestionnaire `opam` est installé, on peut utiliser, dans un terminal :

```
opam install graphics
eval `opam config env`
```

On pourra également être amené à installer `ocamlfind` de la sorte pour disposer de `topfind`.

Si d'aventure il ne vous était pas possible de faire fonctionner la partie graphique, vous pouvez commenter les différentes fonctions graphiques dans le fichier, et utiliser la version dessinant les labyrinthes en ASCII dans le terminal.

1.4 Utilisation du module `Graph`

Pour manipuler des graphes non-orientés, on fournit un module `Graph`, qui propose différentes fonctions. Pour créer un nouveau graphe g, on écrira

```
let g = Graph.create ()
```

Les sommets du graphe peuvent être identifiés par la plupart des types OCaml (entiers, chaînes de caractères, n-uplets...) Pour ajouter une arête au graphe, entre deux sommets (par exemple identifiés par 37 et 42), on utilise :

```
Graph.add_edge g 37 42
```

Notons qu'il n'est pas besoin de déclarer les sommets d'un graphe, seules les arêtes importent.

Pour obtenir la liste des identifiants des sommets voisins du sommet 37, on utilise `Graph.neighbors` qui retourne la liste des identifiants des sommets voisins :

```
Graph.neighbors g 37
```

Enfin, pour tester si deux sommets sont voisins dans le graphe, on utilisera la fonction `Graph.exists_edge` qui retourne un booléen :

```
Graph.exists_edge g 37 42
```

1.5 Exemples de labyrinthes et affichage

On fournit également une fonction `gen_maze` qui prend en argument trois entiers `s`, `h` et `w` et retourne un labyrinthe de taille $h \times w$ (le paramètre `s` est une graine qui permet, selon la valeur fournie, d'obtenir des labyrinthes différents).

```
let my_maze = gen_maze 1 30 45
```

Pour afficher ce labyrinthe, on utilisera la fonction `draw_maze` qui prend en argument un graphe représentant un labyrinthe et ses dimensions :

```
draw_maze my_maze 30 45
```

Si vous ne parvenez pas à faire fonctionner l'interface graphique, vous pouvez commenter toutes les fonctions graphiques, et utiliser la fonction `ascii_maze` pour le labyrinthe en ASCII dans la sortie standard, en écrivant :

```
ascii_maze my_maze 30 45;
```

2 Étude de labyrinthes

2.1 Labyrinthes parfaits

1. Proposer une fonction `is_connex` qui prend en argument un graphe représentant un labyrinthe et ses dimensions h et w et retourne un booléen indiquant si le graphe est connexe (on pourra étudier le nombre de sommets atteints par une exploration en profondeur depuis le sommet 0)

2. Quels labyrinthes de taille 30×45 sont connexes parmi ceux générés par la fonction `gen_maze i 30 45` pour i entre 1 et 5?

Un labyrinthe parfait doit nécessairement vérifier la propriété précédente, mais ce n'est pas suffisant : il doit être acyclique pour que les chemins soient uniques.

3. Proposer une fonction `is_perfect` qui prend en argument un graphe représentant un labyrinthe et ses dimensions h et w et retourne un booléen indiquant si le labyrinthe est parfait (indice : il n'est **pas utile** ici d'essayer de détecter la présence d'un cycle, il y a bien plus simple!).

4. Quels labyrinthes de taille 30×45 sont parfaits parmi ceux générés par la fonction `gen_maze i 30 45` pour i entre 1 et 5?

2.2 Recherche de chemin

On souhaite à présent trouver un chemin dans un labyrinthe menant d'une case identifiée par `src` à une case identifiée par `dst`. Pour ce faire, on va écrire une fonction `path` qui prendra en argument un graphe, et les identifiants de deux sommets `src` et `dst`, et qui retournera une liste des identifiants des cases menant de `src` à `dst`.

Le premier élément de la liste devra être `src`, le dernier `dst`. On ne demande pas de chercher le chemin le plus court (de toute façon, si le labyrinthe est parfait, ce chemin est unique), on pourra donc utiliser une exploration en profondeur. **S'il n'existe aucun chemin, on retournera une liste vide**.

5. Proposer une implémentation de la fonction `path`.

6. Utiliser la fonction `path` pour déterminer des chemins menant de la case en haut à gauche vers la case en bas à droite dans les labyrinthes produits par `gen_maze i 30 45` (on remplacera les `???` ci-dessous par les valeurs adéquates). Pour afficher le résultat, on pourra appeler, immédiatement après avoir utilisé la fonction `draw_maze`, la fonction `draw_path` qui prend pour seul argument la liste des cases visitées, et l'affiche par-dessus le labyrinthe.

```
let my_path = path my_maze ??? ???;;
draw_maze my_maze 30 45;;
draw_path my_path 30 45;;
```

Pour ceux travaillant en mode texte, on utilisera :

```
let my_path = path my_maze ??? ???;;
ascii_maze ~path:my_path my_maze 30 45;;
```

3 Génération de labyrinthes

3.1 Méthode du parcours en profondeur

On souhaite cette fois créer notre propre labyrinthe parfait. Il existe de très nombreuses méthodes pour cela. Une première possibilité utilise un parcours en profondeur.

7. Proposer une fonction `no_wall h w` qui crée un graphe représentant un labyrinthe de taille $h \times w$ sans aucun mur (à part sur les bords, on doit pouvoir se déplacer dans les quatre directions depuis n'importe quelle case.

8. Écrire une fonction `gen_dfs h w` qui construit un labyrinthe de la façon suivante : elle construit un graphe sans mur, puis utilise une exploration en profondeur de ce graphe pour accéder à toutes ses cases en partant de la case 0. Lorsque le parcours en profondeur se déplace d'une case i à une voisine j (atteignant la case j pour la première fois), il mémorise ce déplacement. Une fois l'exploration terminée, on crée un nouveau graphe, vide, et pour tout déplacement enregistré entre une case `i` et sa voisine `j`, on ajoute dans ce nouveau graphe une arête entre `i` et `j`. La fonction retourne ce second graphe.

9. Utiliser la fonction précédente pour créer un labyrinthe de taille 30×45, et afficher le résultat obtenu. Que penser du résultat ?

Pour obtenir un résultat plus intéressant, il faut simplement, dans le parcours en profondeur, considérer les voisins de la case `i` dans un ordre aléatoire.

Pour mélanger les éléments d'une liste (en temps linéaire), on fournit la fonction `shuffle_list` qui prend en argument une liste et retourne une nouvelle liste où les éléments ont été ordonnés de façon aléatoire (méthode du mélange de Knuth).

10. Modifier `gen_dfs` pour que les voisins soient explorés dans un ordre aléatoire, et vérifier que le résultat obtenu est plus satisfaisant.

3.2 Méthode de Kruskal

La méthode précédente tend à créer de longs couloirs. Il existe de nombreuses méthodes pour générer des labyrinthes (méthode d'Eller, de Wilson qui sera proposée plus loin...) Nous allons voir une autre méthode possible.

11. Proposer une fonction `poss_walls h w` construisant une liste de tous les couples (i, j) tels que $i < j$ et i et j sont des identifiants de cases potentiellement voisines dans un labyrinthe de taille $h \times w$.

Pour générer un labyrinthe, on peut procéder de la façon suivante :

- on fait appel à la fonction `poss_walls` pour générer une liste de passages possibles ;
- on mélange cette liste ;
- on crée un graphe vide, qui deviendra le labyrithe ;
- on considère tous les couples (i, j) de la liste, et s'il n'existe pas encore de chemin menant de i à j, elle ajoute une arête entre i et j dans le graphe.

Le point délicat est de savoir s'il existe une chemin menant de i à j. On pourrait utiliser la fonction `path`, mais ce sera inefficace. Pour ce faire, on va utiliser le module `UnionFind` fourni. Il s'agit d'un outil utilisant le principe « Union-find » qui sera étudié en seconde année et permet de répondre efficacement (pratiquement en temps constant) à cette question.

Il s'agit d'une structure de données permettant de mémoriser des « classes d'équivalence ». Deux cases seront dans la même classe d'équivalence s'il existe un chemin entre les deux cases.

Initialement, chaque élément est seul dans sa propre classe d'équivalence. On initialise la structure de donnée en écrivant :

```
let uf = UnionFind.create ()
```

Pour indiquer qu'une case communique avec une autre case (et donc que les deux classes d'équivalence auxquelles elles appartiennent n'en font plus qu'une), par exemple 37 et 42, on utilise la fonction `UnionFind.union` :

```
UnionFind.union uf 37 42
```

Pour une case donnée d'identifiant i, on peut trouver un identifiant de case qui représente l'ensemble de la classe d'équivalence à laquelle appartient la case i (cet identifiant est le même pour toutes les cases de la classe d'équivalence entre deux appels à `UnionFind.union`). Par exemple, pour la case 37 :

```
let r = UnionFind.find uf 37
```

Enfin, pour tester s'il existe un chemin entre deux cases, par exemple 17 et 54, on utilisera la fonction `UnionFind.joined` qui retourne un booléen indiquant si les deux cases sont dans la même classe d'équivalence :

```
let exists_path = UnionFind.joined uf 17 54
```

12. Proposer une fonction `gen_Kruskal` générant un labyrinthe avec la méthode précédente.

13. Utiliser la fonction précédente pour générer quelques labyrinthes, les afficher et vérifier qu'ils sont parfaits.

4 Recherche de cycles

4.1 Objectif

La fonction `gen_maze` produit parfois des labyrinthes pour lesquels il existe plusieurs chemins allant d'une case à une autre. Lorsque cela arrive, il existe donc un *cycle*, c'est-à-dire un chemin $s_0, s_1, s_2, \ldots, s_{n-1}, s_n = s_0$ où toutes les cases, à l'exception de la dernière, sont distinctes, mais qui revient à son point de départ.

4.2 Cas d'un labyrinthe connexe

On suppose dans un premier temps le labyrinthe connexe. On peut trouver un cycle en effectuant un parcours en profondeur à partir de la case 0 : si, parmi les voisines de i, il y a une case j qui a déjà été visitée, et que on n'a pas atteint i en venant de la case j, alors on vient de trouver un cycle.

14. Proposer une fonction `find_cycle` qui prend en argument un labyrinthe connexe et retourne une liste de cases constituant un cycle (liste d'entiers constituant le cycle, la première case devant être égale à la dernière) s'il en existe un, et une liste vide sinon.

15. Utiliser la fonction `draw_path` pour afficher un cycle sur un labyrinthe connexe non parfait obtenu avec `gen_maze`.

4.3 Cas général

16. Modifier la fonction pour qu'elle trouve toujours un cycle s'il en existe un même si le labyrinthe n'est pas connexe.

5 Algorithme d'Eller

5.1 Principe

La méthode de génération de labyrinthe proposée par Eller est intéressante à plusieurs titres. Le point le plus remarquable est qu'elle produit des labyrinthes de très bonne qualité en procédant ligne par ligne (ce qui permet, moyennant quelques conditions, de réduire la quantité de mémoire nécessaire pour générer des labyrinthes de très grande taille).

Le principe est le suivant :

- pour toutes les lignes, sauf la dernière :
  - on parcourt toutes les paires de cases voisines de la ligne, dans un ordre aléatoire, et si les deux cases ne sont pas dans la même classe d'équivalence, on les connecte avec une probabilité `1/2` ;
  - on regroupe les cases de la lignes selon leur classe d'équivalence (on obtient une liste de liste de cases), et pour liste de cases, on la mélange, on connecte la première des cases avec la case inférieure, et on connecte les suivantes avec la case inférieure avec une probabilité `1/2` ;
- pour la dernière ligne, on parcourt toutes les paires de cases voisines de la ligne, dans un ordre aléatoire, et si les deux cases ne sont pas dans la même classe d'équivalence, on les connecte systématiquement.

17. Justifier que l'on obtient bien avec cette méthode un labyrinthe parfait, et que tout labyrinthe parfait peut être obtenu avec cette méthode.

5.2 Implémentation

18. Proposer une fonction `partition` qui prend en argument la largeur w des lignes, un numéro de ligne (entre 0 et $h-2$) et un objet `UnionFind` et retourne une partition des identifiants des cases de la ligne selon leur classe d'équivalence.

19. En déduire une implémentation `gen_Eller` de l'algorithme d'Eller.