Projet : génération et résolution de labyrinthes

Consignes :

- Lisez bien tout le sujet avant de commencer à coder.
- Déposez avant le **vendredi 1 février à 23h59** une archive au format `.tar.gz`, contenant votre code (fichiers `.ml`), vos fichiers de test éventuels et un rapport au **format PDF**, dans le dépôt `iprf_projet` sur `http://exam.ensiie.fr`.
- Pensez à commenter votre code. Au minimum, chaque déclaration de fonction devra être précédée d'un commentaire expliquant ce que la fonction est censée faire et sous quelle(s) hypothèse(s) sur les arguments.

L'objectif de ce projet est d'écrire un programme en OCaml qui génère un labyrinthe, puis qui le résout. Le sujet est composé de 5 parties :

1. un petit échauffement avec quelques fonctions utiles sur des listes,
2. la création d'une grille qui servira de base pour la génération d'un labyrinthe,
3. l'affichage et les premiers tests,
4. la génération d'une labyrinthe à partir d'une grille et sa résolution,
5. quelques pistes pour aller plus loin.

1 Quelques fonctions sur les listes

Le but de cette partie est de code quelques fonctions sur les listes qui vont nous servir pour la suite du projet.

1.1 Création de listes

Question 1. Écrire une fonction `range: int -> int -> int list` qui, sur la donnée de deux entiers `a` et `b`, construit la liste de tous les entiers entre `a` et `b` inclus.

Par exemple, `range 0 5` devra retourner `[0; 1; 2; 3; 4; 5]`.

Grâce à cette fonction, il est possible de traduire facilement des boucles en OCaml :

— 1 **pour** i *allant de* a *à* b **faire**
2 $\quad l[i] \leftarrow f(i)$;

se traduit par

```
List.map f (range a b)
```

— 1 `acc` $\leftarrow v_0$;
2 **pour** i *allant de* a *à* b **faire**
3 $\quad$ `acc` $\leftarrow f($`acc`$, i)$;
4 **retourner** `acc` ;

se traduit par

```
List.fold_left
    (fun acc i -> f acc i)
    v0
    (range a b)
```

Question 2. Écrire une fonction `range2: int -> int -> (int * int) list` qui, sur la donnée de deux entiers `m` et `n`, construit une liste contenant tous les couples (i, j) avec $0 \leq i < m$ et $0 \leq j < n$. Aucun ordre n'est imposé sur les couples.

1.2 Mélange d'une liste

Nous aurons besoin de mélanger des listes lors de la génération de labyrinthes. Pour cela, nous allons implanter l'algorithme 1.

Algorithme 1 : `shuffle`

Entrée : une liste ℓ
Sortie : une liste ℓ' avec les éléments que ℓ dans un ordre aléatoire

```
1 n ← longueur(ℓ)
2 si n = 0 alors
3 |  retourner []
4 sinon
5 |  i ← entier choisi aléatoirement entre 0 et n − 1
6 |  x ← i^e élément de ℓ
7 |  r ← ℓ privée de son i^e élément
8 |  r' ← shuffle(r)
9 |  retourner x :: r'
```

Question 3. Écrire une fonction récursive `remove_nth: int list -> int -> int list` qui, sur la donnée d'une liste ℓ et d'un indice `i`, renvoie une liste correspondant à ℓ privée de son élément d'indice `i` (dont on suppose l'existence).

Par exemple, `remove_nth [0;1;2;3] 2` devra retourner `[0; 1; 3]`.

Question 4. Écrire une fonction `extract_random: int list -> int * int list`, de telle sorte que `extract_random` ℓ renvoie le couple `(x,r)` où `x` est un élément de ℓ choisi aléatoirement et où `r` est la liste ℓ privée de `x`.

note : On utilisera `Random.int n` pour obtenir un nombre choisi aléatoirement entre 0 et $n-1$.

Question 5. Écrire la fonction `shuffle: int list -> int list`.

2 Représentation de la grille

Pour cette partie et les deux suivantes, nous allons travailler sur une grille rectangulaire de $m \times n$ cases, où m et n sont des paramètres. Pour représenter chaque case, nous utiliserons le type `Cell.t` défini à l'aide du code suivant :

```
module Cell =
  struct
    type t = int * int
    let compare = Pervasives.compare
  end
;;
```

La grille est alors représentée en OCaml comme un graphe, à l'aide du type `grid` défini par :

```
module CellMap = Map.Make(Cell) ;;
module CellSet = Set.Make(Cell) ;;
type grid = CellSet.t CellMap.t ;;
```

2.1 Fonctions sur les graphes

Question 6. Écrire une fonction `add_vertex: Cell.t -> grid -> grid` qui, sur la donnée d'une case `v` et d'un graphe `g`, renvoie un nouveau graphe constitué de `g` auquel on a ajouté le sommet `v` (sans successeurs).

note : Si `v` est déjà dans `g`, on veillera à renvoyer directement `g`.

Question 7. Écrire une fonction `add_edges: Cell.t -> Cell.t -> grid -> grid` qui, sur la donnée de deux cases `u` et `v`, et d'un graphe `g`, renvoie un nouveau graphe constitué de `g` auquel on a ajouté (si besoin) les arêtes de `u` à `v` et de `v` à `u`.

2.2 Génération d'une grille

Il s'agit maintenant de coder une fonction `create_grid: int -> int -> grid` qui, sur la donnée de m et n, génère le graphe correspondant à une grille rectangulaire de taille $m \times n$. On numérote les cases d'une telle grille de $(0,0)$ à $(n-1, m-1)$, avec m le nombre de lignes et n le nombre de colonnes.

Question 8. Écrire une fonction `is_valid: int -> int -> Cell.t -> bool` telle que `is_valid m n c` retourne `true` si `c` correspond bien à une case de la grille rectangulaire de taille $m \times n$, et `false` sinon.

Question 9. Écrire une fonction `get_neighbours: int -> int -> Cell.t -> Cell.t list` qui, sur la donnée de `m`, `n` et `c`, dresse la liste de tous les voisins de la case `c` sur une grille $m \times n$.

Question 10. Écrire la fonction `create_grid` en procédant de la façon suivante :

1. définir une variable locale `lc` contenant la liste de toutes les cases,
2. ajouter les éléments de `lc` à `CellMap.empty` comme nouveaux sommets du graphe,
3. pour chaque élément `c` de `lc`, ajouter les arêtes de `c` vers ses voisins.

3 Affichage et premiers tests

Nous allons maintenant mettre en place de quoi afficher une grille, ce qui permettra de tester le reste du code. L'affichage se fera à l'aide du code disponible à l'adresse :

`http://web4.ensiie.fr/~christophe.mouilleron/teaching/IPRF/projet/display.ml` .

Ce code repose sur le module `Graphics` de OCaml, qui fournit quelques primitives d'affichage simples. L'affichage d'une grille rectangulaire se fait comme illustré à la figure 1, chaque case étant un carré de taille 2×2 (cf figure 2).

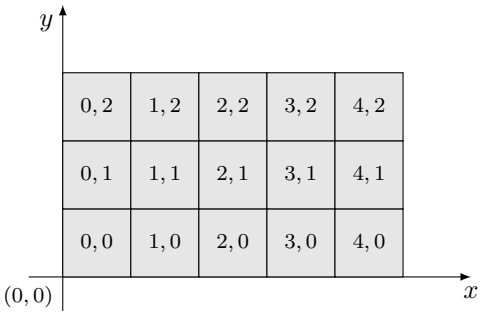

FIGURE 1 – Positionnement des cases à l'écran pour une grille 3×5.

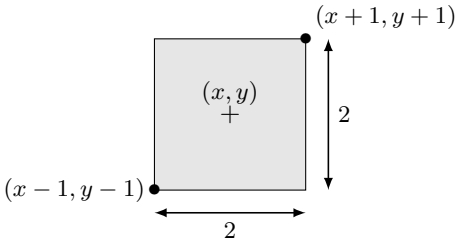

FIGURE 2 – Case centrée en (x, y).

Question 11. Écrire une fonction `get_center_coord: Cell.t -> int * int` qui, sur la donnée d'une case `c`, retourne les coordonnées (x, y) du centre de la case.

Question 12. Écrire une fonction `get_contour: Cell.t -> (int * int) list` qui, sur la donnée d'une case `c`, retourne la liste des coordonnées des quatre coins de `c`.

Question 13. Écrire une fonction `get_wall: Cell.t -> Cell.t -> (int * int) list` qui, sur la donnée de deux cases `c1` et `c2`, retourne la liste des coordonnées des coins communs à `c1` et `c2` (qui sont donc les extrémités du mur séparant `c1` et `c2`).

Ainsi, `get_wall (1,2) (0,2)` devra retourner `[ (2,4); (2,6) ]` ou `[ (2,6); (2,4) ]`, alors que `get_wall (3,1) (0,2)` devra retourner une liste vide car les cases $(3,1)$ et $(0,2)$ ne sont pas voisines.

Pour tester ce que vous avez fait jusqu'à présent, ajoutez à la fin de votre fichier le code suivant :

```
#use "display.ml" ;;
let g = create_grid 70 90 ;;
test (800,600) (4,0,0) g (0,0) (89,69) [] ;;
```

Normalement, vous devez obtenir un labyrinthe sans aucun mur à l'intérieur (rectangle gris), avec une entrée en vert et une sortie en rouge.

Pour obtenir ce résultat, vous devez :
— soit utiliser `Emacs` et mettre `ocaml graphics.cma` au lancement de l'interpréteur,
— soit lancer la commande `ocaml graphics.cma votre_fichier.ml` dans le terminal.
Dans tous les cas, faites bien attention à ne pas oublier l'argument `graphics.cma`.

Question 14. Faites quelques tests avec d'autres paramètres. On gardera `[]` comme dernier paramètre tant que la partie sur la résolution de labyrinthe n'est pas finie.

Dans votre rapport, expliquez avec le plus de détails possibles ce que font les différentes fonctions présentes dans le fichier `display.ml`, et comment elles le font.

note : N'hésitez pas à consulter la documentation en ligne du module `Graphics` si besoin.

4 Génération et résolution d'un labyrinthe

Maintenant que nous avons une grille (sous forme d'un graphe) comme support, nous allons générer un labyrinthe sur cette grille. Techniquement, il s'agit de construite un arbre couvrant de notre graphe, c'est-à-dire un arbre dont les sommets sont les mêmes que ceux du graphe et reliant tous ces sommets (sans faire de cycle puisqu'il s'agit d'un arbre).

Pour se faire, nous allons partir d'un sommet v quelconque du graphe, créer une branche en avançant de proche en proche aussi longtemps que possible, et recommencer la création de branche tant qu'il reste des voisins de v qui n'appartiennent à aucune branche. Cette approche est détaillée par l'algorithme 2.

Algorithme 2 : `generate_maze_aux`

Entrée : une grille g servant de support, un labyrinthe m en cours de construction, un sommet v et une liste ℓ
Sortie : un nouveau labyrinthe avec au minimum les mêmes murs que m
Hypothèses : les éléments de ℓ sont tous des voisins de v dans g.

si ℓ *est vide* **alors retourner** m
sinon
  $v' \leftarrow$ premier élément de ℓ
  $t \leftarrow \ell$ privée de son premier élément
  si v' *est déjà un sommet de* m **alors retourner** `generate_maze_aux`(g, m, v, t)
  sinon
    $m' \leftarrow m$ auquel on a ajouté le sommet v', et les arêtes de v à v' et de v' à v
    $\ell' \leftarrow$ liste des successeurs de v' dans g
    Mélanger aléatoirement ℓ'
    $m'' \leftarrow$ `generate_maze_aux`(g, m', v', ℓ')
    retourner `generate_maze_aux`(g, m'', v, t)

Question 15. Écrire une fonction `generate_maze_aux: grid -> grid -> Cell.t -> Cell.t list -> grid`. correspondant à l'algorithme 2.

Question 16. Écrire une fonction `generate_maze: grid -> grid` qui, sur la donnée d'une grille, retourner un labyrinthe sur cette grille.

note : Utilisez `CellMap.choose` pour récupérer une case quelconque de la grille.

Question 17. Écrire une fonction `solve_maze: grid -> Cell.t -> Cell.t -> Cell.t list` qui, sur la donnée d'un labyrinthe, de la case d'entrée et de la case de sortie, résout le labyrinthe.

Vous pouvez tester ce que vous avez fait dans cette partie en adaptant le code suivant :

```
#use "display.ml" ;;
Random.self_init () ;;
let g = create_grid 70 90 ;;
let m = generate_maze g ;;
let s = solve_maze m (0,0) (89,69) ;;
test (800,600) (4,10,10) m (0,0) (89,69) s ;;
```

5 Pour aller plus loin

Cette dernière partie propose deux améliorations.

5.1 Labyrinthe sur une grille hexagonale

L'approche à base de graphe pour la génération et la résolution d'un labyrinthe peut s'appliquer sur n'importe quel type de grille.

On se propose ici de passer à une grille hexagonale, comme celle de la figure 3. La taille de la grille est caractérisée par son rayon (nombre de couronnes) et les cases sont numérotées par un couple (i, j) où i est le numéro de la couronne et j le numéro de la case dans cette couronne (en comptant dans le sens des aiguilles d'une montre). Pour ce qui est des coordonnées des coins de chaque case (qui doivent être entières), vous pouvez vous référer à la figure 4.

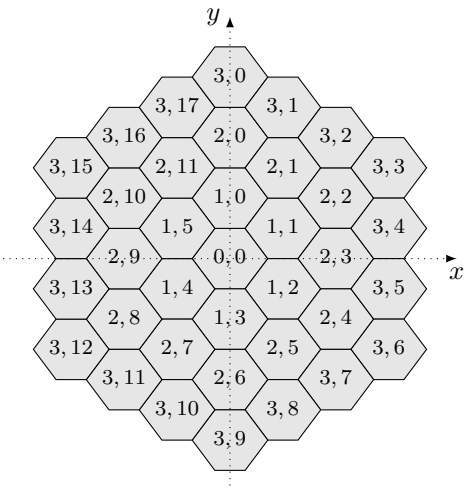

FIGURE 3 – Positionnement des cases pour une grille hexagonale de rayon 4.

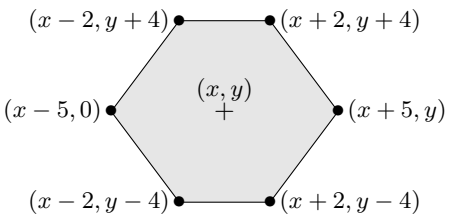

FIGURE 4 – Case hexagonale centrée en (x, y).

Question 18. Dans un nouveau fichier `.ml`, refaire les questions des parties 2 et 3 pour le cas d'une grille hexagonale. Vérifier à l'aide de `display.ml` que le code de la partie 4 fonctionne aussi dans ce cas.

Voici quelques remarques pour vous aider :
- Il est plus facile de traiter la case $(0, 0)$ à part pour `get_neighbours`, `get_contour`, etc.
- Pour `get_neighbours` et `get_center_coord`, il faudra traiter différemment la case (i, j) selon que j est un multiple de i ou non.
- Faites attention aux débordements dans `get_neighbours`. Par exemple, l'un des voisins de $(2, 11)$ est $(2, 0)$, et pas $(2, 12)$.
- Pour `get_wall`, le plus simple est de calculer l'intersection des contours de `c_1` et `c_2`.

Question 19. Séparer vos codes en plusieurs fichiers, et créer un Makefile qui se chargera de la compilation séparée et créera deux exécutables (un pour chaque type de grille).

note : Si vous compilez vos fichiers, vous ne pouvez plus utiliser `#use`. Pour utiliser la variable/fonction `bar` définie dans le fichier `foo.ml`, vous devrez soit l'appeler `Foo.bar`, soit mettre un `open Foo ;;` avant d'utiliser `bar`.

5.2 Version interactive

Plutôt que de faire résoudre le labyrinthe par la machine, on va inviter l'utilisateur à trouver lui-même le chemin reliant l'entrée à la sortie.

Question 20. Proposer une variante `test_interactif` de la fonction `test` fournie dans le fichier `display.ml`, dans laquelle le chemin se construit et s'affiche selon ce que l'utilisateur entre au clavier.

Remarques :
- Il s'agit, une fois la partie graphique initialisée, de faire appel à une fonction récursive dont le cas d'arrêt correspond à la réception d'une touche (disons `Esc` dont le code ascii est 27) pour quitter.
- Un appel à `Graphics.clear_graph ()` permet d'effacer tout ce qui a été dessiné précédemment.
- La récupération de la touche saisie par l'utilisateur se fait via :

```
let ev = Graphics.wait_next_event [Graphics.Key_pressed] in
let key = ev.Graphics.key in
...
```

- Je vous suggère d'écrire une fonction auxiliaire qui, étant donnée la case courante et la touche saisie par l'utilisateur, retourne la case d'arrivée (qu'elle soit valide ou non).