

Rapport de Projet : génération et résolution de labyrinthes

Introduction :	1
Partie 1 :	1
Partie 2 :	2
Partie 3 :	3
Explication du display.ml :	4
Partie 4 :	5
Génération d'un labyrinthe :	5
Résolution d'un labyrinthe :	6
Partie 5 :	7
Partie hexagonale :	7
Makefile :	8
Partie interactive :	8
Conclusion :	8

Introduction :

Le but de ce projet est de générer et résoudre des labyrinthes.

On représente un labyrinthe avec un graphe, qui sera modélisé avec une Map en OCAML, ayant comme clé un sommet et comme valeurs un ensemble de sommets correspondant à ses arêtes.

README.md contient des informations complémentaires.

```
(0,0) => { (0,1), (1,0) }  
(0,1) => { (0,0), (0,2), (1,1) }  
...  
(2,1) => { (1,1), (2,0), (2,2), (3,1) }  
...  
(4,1) => { (3,1), (4,0), (4,2) }  
(4,2) => { (3,2), (4,1) }
```

Partie 1 :

La difficulté de cette partie pour moi a été la fonction **range2**, il m'a fallu relire l'énoncé plusieurs fois et m'entraîner en cours sur la récursivité avant de trouver une solution, j'ai donc demandé de l'aide au professeur qui m'a montré le fonctionnement général et la logique qui permet de la coder facilement.

Partie 2 :

Dans cette partie, ma difficulté a été de comprendre les lignes de code introduites par le sujet (ci-dessous) qui me permettront de faire **add_vertex** et **add_edges**. J'ai beaucoup cherché sur Internet et j'ai finalement compris qu'il fallait lire cette ligne de la droite vers la gauche "**type grid = CellSet.t CellMap.t ;;**" pour mieux comprendre : le type grid est une map contenant en valeur un ensemble de sommets (exemple dans l'introduction)

De plus, mon algorithme de génération de labyrinthes ne fonctionnait pas à cause du **add_vertex** car je ne vérifiais pas que le sommet donné en argument pouvait déjà être ajouté.

```
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 ;;
```

À propos de la méthode **create_grid**, j'ai d'abord essayé de la faire à l'aide de **2 fold_left** dans 2 fonctions séparées, mais lorsque je testais ma méthode, elle retournait une exception **NOT_FOUND**. c'était parce que ma map était vide car le **create_grid** ne remplissait pas bien la grille car ma méthode **add_neighbours_edges** remplissait la variable **grid** qui n'était pas la même que celle dans **create_grid**.

```
let add_neighbours_edges = fun c1 la lo grid ->  
  List.fold_left (fun acc c -> add_edges c1 c acc) (add_vertex c1 grid)  
  (get_neighbours la lo c1);;
```

```
let create_grid = fun la lo ->  
  let lc = range2 la lo in  
  List.fold_left (fun acc c1 -> add_neighbours_edges c1 la lo acc)
```

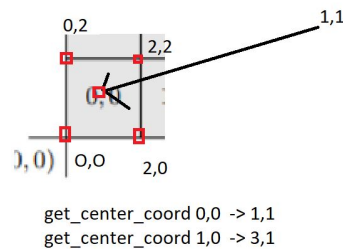
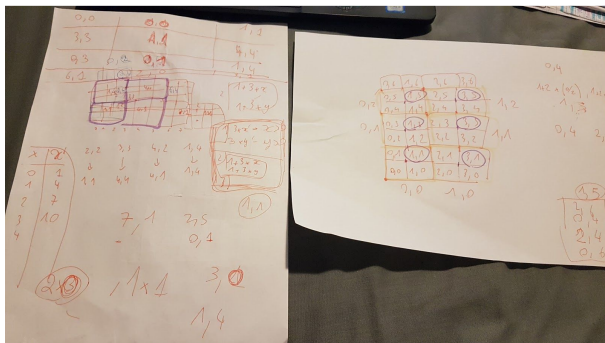
```
CellMap.empty lc
;;
(exemple de code qui ne fonctionnait pas)
```

Il fallait plutôt faire un `fold_left` pour parcourir la liste de sommet à rajouter dans la map, puis un autre pour parcourir les sommets ajoutés, puis un dernier pour parcourir les voisins du sommet courants à ajouter en tant qu'arête.
(Ce n'est pas la solution optimale mais fonctionne très bien)

Partie 3 :

Ensuite ma prochaine difficulté fut de visualiser ce que je voulais coder pour les méthodes : `get_center_coord`, `get_contour`, `get_wall`.

Pour ce faire, j'ai réalisé des schémas et quelques brouillons pour m'aider :



(Les schémas sont faux)

Pour commencer j'avais compris que :

- Une case était représentée par 9 cases de 3*3 et que la case du milieu serait le centre de cette case
- 4 cases de bases (2*2) et que les coordonnées des points de chaque case était la coordonnées inscrite sur la case à côté de celui-ci en haut à droite.
- Puis la bonne solution, qu'une case réel qui à comme nom 0,0 à comme point au centre 1,1 (il y a deux choses différentes ici : le nom de la case, et les coordonnées dans ce repère, qui a par ailleurs une échelle de 1 case = 2 unité car le module graphics de OCAML ne prend pas de valeur flottante)

j'ai donc finalement corrigé ma fonction `get_center_coord` : $1+2*x'$ et $1+2*y'$. en $1+2x$ et $1+2y$

Une de mes autres difficulté fut aussi que j'avais permuté le x et y dans `range2` et `create_grid`.

Explication du display.ml :

```
type status = Input | Output | Inside ;;
```

le type status permet de définir les 3 états différents d'une case, elle peut être

- l'entrée du labyrinthe : Input
- la sortie : Output
- une case basique. : Inside

```
let color_of_status = fun st -> match st with[]  
| Input -> Graphics.green  
| Output -> Graphics.red  
| Inside -> Graphics.rgb 192 192 192 (* light gray *)  
;;
```

Elle permet de colorier les cases en fonctions de leur types status

```
let do_the_thing = fun f -> fun u -> fun v ->  
match List.map f (get_wall u v) with  
| [(x1,y1); (x2,y2)] ->  
    let _ = Graphics.moveto x1 y1 in  
    Graphics.lineto x2 y2  
| _ -> ()  
;;
```

Cette méthode permet de dessiner un mur

```
let draw_vertex = fun f -> fun st -> fun v -> fun succ ->  
let contour = List.map f (get_contour v) in  
let contour' = Array.of_list contour in  
let _ = Graphics.set_color (color_of_status st) in  
let _ = Graphics.fill_poly contour' in  
let _ = Graphics.set_color Graphics.black in  
let _ = Graphics.draw_poly contour' in  
let _ = Graphics.set_color (color_of_status st) in  
let _ = CellSet.iter (fun u -> do_the_thing f u v) succ in  
Graphics.set_color Graphics.black  
;;
```

Cette méthode permet de dessiner une case coloré en fonction de son statut;

```
let get_status = fun v_in -> fun v_out -> fun v ->  
if v = v_in then Input  
else if v = v_out then Output  
else Inside  
;;
```

Cette méthode est un getter pour récupérer le status d'une case

```
let init = fun w -> fun h ->  
let _ = Graphics.open_graph "" in  
let _ = Graphics.resize_window w h in  
Graphics.clear_graph ()  
;;
```

Cette méthode ouvre et initialise la fenêtrés graphique.

```
let wait_and_close = fun () ->  
let _ = Graphics.wait_next_event [Graphics.Key_pressed] in  
Graphics.close_graph ()  
;;
```

Cette méthode est une lecture d'événement qui sur l'appuie d'une touche du clavier, ferme la fenêtre.

```
let draw_maze = fun f -> fun g -> fun v_in -> fun v_out ->
  CellMap.iter (fun v succ -> draw_vertex f (get_status v_in v_out v) v succ) g
;;
```

Cette méthode parcourt l'ensemble des cases de la grille et les colories

```
let rec draw_path_aux = fun f -> fun g -> fun v -> fun succ -> fun path ->
  match path with
  | [] -> ()
  | v::vs ->
    if CellSet.mem v' succ then
      try
        let succ' = CellMap.find v' g in
        let x, y = f (get_center_coord v) in
        let x',y' = f (get_center_coord v') in
        let _ = Graphics.moveto x y in
        let _ = Graphics.lineto x' y' in
        draw_path_aux f g v' succ' vs
      with _ ->
        let (i',j') = v' in Printf.printf "(%d,%d) is outside of the maze\n" i' j'
    else
      let (i,j) = v in
      let (i',j') = v' in
      Printf.printf "No path from (%d,%d) to (%d,%d)\n" i j i' j'
;;
```

cette méthode auxiliaire récursive permet de tracer le chemin proposé en solution.

```
let draw_path = fun f -> fun g -> fun path -> match path with
  | [] -> ()
  | v::vs ->
    try
      let succ = CellMap.find v g in
      let _ = Graphics.set_color Graphics.blue in
      draw_path_aux f g v succ vs
    with _ ->
      let (i,j) = v in Printf.printf "(%d,%d) is outside of the maze\n" i j
;;
```

Cette méthode utilise la méthode auxiliaire pour tracer le chemin donnée en arguments.

```
let test = fun (w,h) -> fun (z,sw,sh) -> fun g -> fun v_in -> fun v_out -> fun path ->
  let _ = init w h in
  let f = fun (x,y) -> (z*x+sw, z*y+sh) in
  let _ = draw_maze f g v_in v_out in
  let _ = draw_path f g path in
  wait_and_close ()
;;
```

cette méthode est l'équivalent de la boucle "idle" : boucle de "jeu" qui initialise la fenêtre, dessine le labyrinthe et affiche le chemin donnés en arguments

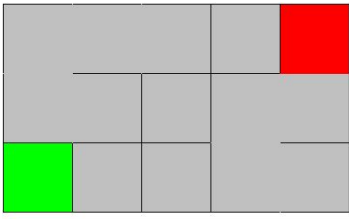
Partie 4 :

Génération d'un labyrinthe :

Ma 1ère erreur à été d'ajouter les arêtes avant d'ajouter les sommets, logiquement ça ne pouvait pas fonctionner.

De plus je n'ajoutais pas les 2 sommets v et v' mais seulement v.

J'obtenais ça :



```

# val maze : CellSet.t CellMap.t = <abstr>
# - : unit = ()
# (0,0) : (1,0)
(0,1) : (1,1)
(0,2) : (0,1)
(1,0) : (2,0)
(1,1) : (0,1)
(1,2) : (0,2)
(2,0) : (2,1)
(2,1) : (2,2)
(2,2) : (1,2)
(3,0) : (4,0)
(3,1) : (3,0)
(3,2) : (4,2)
(4,0) : (3,0)
(4,1) : (3,1)
(4,2) : (3,2)
U:**- *ocaml-topLevel* 99% L2586 (Tuareg-Interactive:run)

```

Résolution d'un labyrinthe :

Ma 1er méthode intuitive qui, selon moi, ressemble un peu à l'algorithme de trémaux :

Tant que je trouve pas la sortie, j'avance sur un voisin sans mur qui n'a jamais été visité (et donc l'ajoute à la liste), si je suis bloqué, je retourne à la case d'avant tant que je trouve pas un autre chemin sinon je continue.

Ma méthode récursive qui ne fonctionne pas:

si je suis arrivé à la sortie,

je retourne le chemin

sinon

je récupère les voisins sans mur

si il n'y en a aucun

je retourne une exception (chemin sans issues)

sinon

pour tout les voisins sans mur je recommence cette méthode (je **try catch** à ce moment)

(et c'est là, la magie du fonctionnelle , j'ai pas besoin de gérer le backtracking, il est géré automatiquement par le try catch et les appels récursifs)

Une méthode intéressante que j'ai trouvé pour mon solveur, left hand :

si je suis à la sortie je retourne le chemin

sinon

je regarde si il y a un mur à gauche

s'il y en a un,

je tourne ma direction de 90 degrés

sinon

je regarde si il y a un mur dans ma direction,

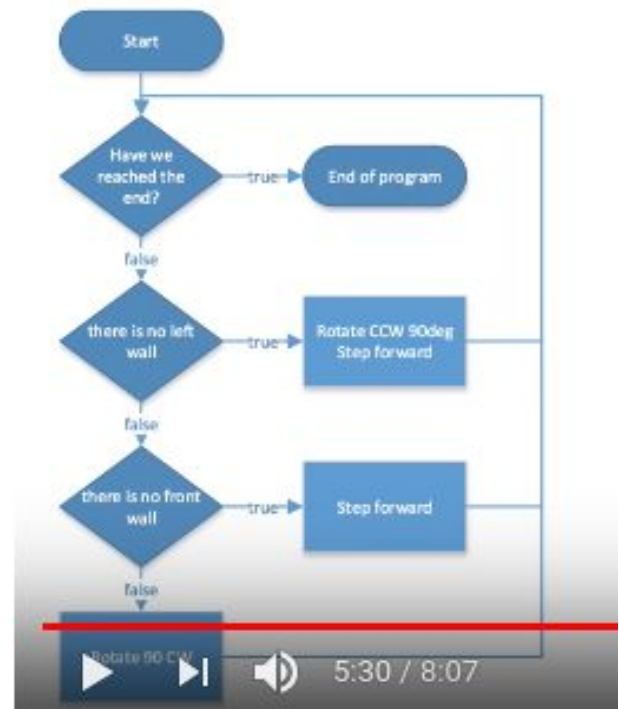
s'il y en a un,

j'avance,

sinon

je tourne ma direction de 90 degrés.

Au début ça ne fonctionnait pas et ça me donnait une boucle infini car dans le cas ou il n'y a pas de mur en face, je tournais à gauche, mais il fallait , en faite, changer de direction vers la droite (à 90° sens de l'aiguille d'une montre)



Autre algorithme vu sur internet : A*, très performant mais plutôt complexe à implémenter selon moi.

Partie 5 :

Partie hexagonale :

J'ai ré-implémenter les parties 2 et 3 pour permettre de générer un labyrinthe de grille hexagonale.

Mes difficultés on été principalement liée aux méthodes :

- `get_center_coord()`
- `get_neighbours()`

Pour `get_center_coord()` je partais de (0,0) montais en (0,y*8) (pour une case (5,0) , les coordonnées (0,5*8)). Ensuite on tourne autour de la couronnes courante et lorsqu'on tombe sur la case de nom (a,b), je m'arrête et je retourne les coordonnées trouvées.

Pour `get_neighbours()`, je récupère le centre des coordonnées des cases autours, et pour chaque coordonnée. je la cherche. donc monte puis tourne autour de la couronnes actuelle,

puis récursivement, je le fais pour toutes les couronnes supérieures tant que je ne trouve pas la coordonnée donnée en arguments. Je le fais pour les 6 voisins puis j'appelle `is_valid` pour vérifier que les cases existent bien dans notre grille.

Mes 2 fonctions (ci-dessus) fonctionnent bien mais ne sont pas optimisées du tout. car elles parcourent beaucoup de coordonnées inutilement. J'ai comparé le temps d'exécution de la génération d'une grille hexagonale de taille 200 avec un camarade, pour moi, il faudra attendre 15 minutes alors que pour mon camarade, seulement 1 minute ! (Vérifié avec la commande **time**)

Je conseille donc de lancer au maximum avec une grille hexagonale de taille 38 pour un rendu et un temps d'attente optimale.

Ensuite, malheureusement, mon algorithme de résolution de labyrinthe, ne fonctionne pas sur une grille hexagonale car ici, il n'y a plus de notion de "mur de gauche" ou "droite".

Makefile :

Ensuite, j'ai décidé de faire un makefile pour :

- générer un labyrinthe rectangulaire et lancer l'algorithme de résolution
- générer un labyrinthe et permettre à l'utilisateur de le résoudre lui-même
- générer un labyrinthe hexagonal (à résoudre aussi mais le solveur ne fonctionne pas)

J'ai remplacé `ocamlc` par `ocamlopt` car c'est beaucoup plus rapide à exécuter.

Partie interactive :

Pour ça, j'ai tout simplement repris le code des méthodes `:test`, `draw_path`, `draw_path_aux` dans le `display.ml` et retiré tout ce qui était lié aux paths donnés en argument, l'utilisateur choisit la direction en utilisant `z,q,s,d`.

Amélioration possible :

- garder le chemin parcouru en paramètre et retirer le dernier trait si on retourne sur ses pas.
- adapter le code pour une grille hexagonale.

Conclusion :

Pour conclure, ce projet m'a permis de mieux comprendre la programmation fonctionnelle, la récursivité et la connaissance de certains algorithmes de résolution de labyrinthe.

Le manque de documentation et d'aide sur Internet m'a obligé à généralement tout coder moi-même et devoir le comprendre pour debugger si besoin. C'était dur, mais ça m'a apporté beaucoup de bon réflexe.

