On s'intéresse dans ce TP à la création de labyrinthe et à leur résolution. Pour tout le TP, un labyrinthe sera représenté par une matrice de 0 et 1 telle qu'un 0 représente un mur (qui sera dessiné en noir) et un 1 représente une case libre (dessinée en blanc). Les bords du labyrinthe seront toujours des murs.
Nous considérons qu'un labyrinthe est « bien construit » s'il n'existe pas de boucles entre deux cases libres. De ce fait, entre deux cases libres, il existera un unique chemin dans le labyrinthe.
Nous distinguerons différentes cases :

- Les **pièces** sont les cases de coordonnées impaires. À la fin de la construction, toutes les pièces seront libres, donc avec un 1 dans la matrice.
- Les **cloisons** sont les cases dont exactement une coordonnée est impaire. Une cloison se situe toujours entre deux pièces adjacentes. La forme du labyrinthe dépendra de quelles cloisons sont creusées.
- Les **piliers** sont les cases de coordonnées paires. Elles ne seront jamais creusées pendant la construction.

Ainsi, un labyrinthe peut-être vu comme un arbre ternaire, dont la racine est la pièce en bas à gauche, les nœuds sont les pièces, et les arêtes sont les cloisons. Une pièce ayant plus de 2 pièces voisines forme une intersection.
La résolution du labyrinthe consiste à trouver l'unique chemin de la racine au nœud correspondant à la case en haut à droite.

1 Génération de labyrinthes

1.1 Généralités

Il n'existe pas de « bonne » manière de construire aléatoirement un labyrinthe, mais plusieurs techniques construisant des labyrinthes aux propriétés différentes (taille et nombre des ramifications depuis le chemin de résolution, principalement). Nous proposons dans cette partie une méthode pour construire un labyrinthe, mais vous trouverez à la fin du sujet d'autres méthodes possibles.

Question 1 Écrire la matrice et dessiner l'arbre correspondant au labyrinthe suivant :

Le module graphique de caml s'ouvre de la manière suivante :

```
#load "graphics.cma";;
open Graphics;;
```

On dispose des commandes suivantes, qui nous permettront de dessiner les labyrinthes :

- `open_graph : string -> unit` : ouvre une fenêtre graphique dont les propriétés sont renseignées dans une chaîne de caractères. Nous l'utiliserons ici uniquement avec l'argument indiquant la taille, de la forme `"690x690"` par exemple. Les dessins ne sont possibles que lorsque la fenêtre graphique est ouverte.
- `set_color : color -> unit` : modifie la couleur de tracé. Les couleurs sont des constantes, comme `black`, `white`, `red` ou `green`. Nous utiliserons le tableau suivant :

  ```
  let couleurs = [|black;white;red;yellow;cyan;green|];;
  ```

- `fill_rect : int -> int -> int -> int -> unit` qui prend en arguments 2 entiers qui sont une abscisse et une ordonnée, ainsi qu'une largeur et une hauteur et trace le rectangle plein de la largeur et hauteur renseignées, dont le coin inférieur gauche se situe aux coordonnées renseignées. La couleur de remplissage est celle indiquée par la commande précédente.

Question 2 On pose la taille d'une case ou d'un mur du labyrinthe par :

```
let pixel = 20;;
```

Écrire une fonction `carre : (int * int) -> color -> unit` qui trace un carré aux coordonnées données en argument de la couleur donnée en argument, et de largeur et hauteur un pixel.

Question 3 En déduire une fonction `dessiner : int array array -> unit` qui trace un labyrinthe donné par une matrice de 0 et de 1. La fonction devra ouvrir une fenêtre graphique de la bonne taille par rapport à la taille de la matrice et d'un pixel, en utilisant `string_of_int`, et la concaténation de chaînes de caractères par `^`.

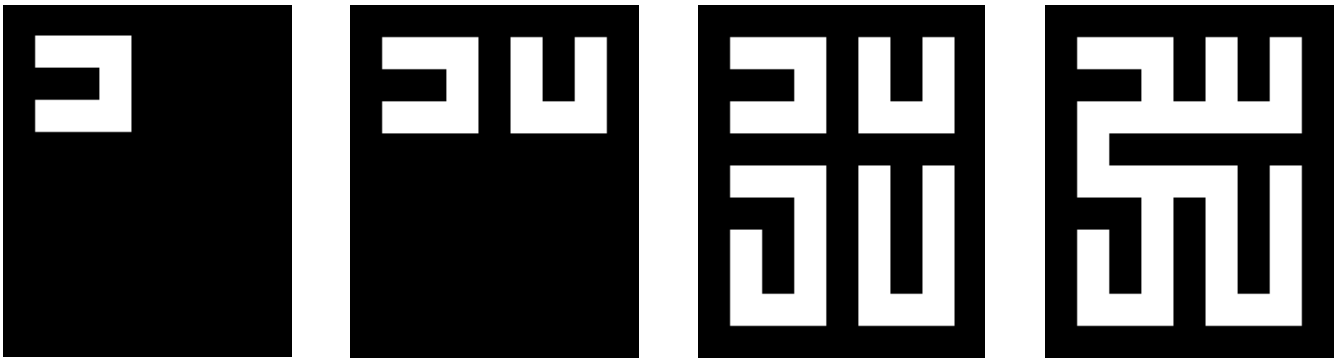
Question 4 Écrire une fonction `colorie : int array array -> int * int -> int -> unit` qui prend en argument une matrice m, un couple d'entiers (i, j) et un entier k et modifie `m.(i).(j)` pour lui donner la valeur `k`, et colorie le carré de coordonnées (i, j) par `couleurs.(k)`.

1.2 Une méthode récursive

La première méthode permet de créer des labyrinthes rectangulaires de la manière suivante :

- On découpe la grille en 4 zones, de tailles équilibrées au maximum.
- Dans 3 des quatre murs de séparation (choisis aléatoirement), on creuse un trou dans une section (choisie aléatoirement). On rappelle que `Random.int n` choisit un nombre aléatoire entre 0 et $n-1$.
- On recommence récursivement pour chacune des quatre zones, sauf si ce sont des pixels ou des rectangles dont l'une des dimensions vaut 1.

Voici par exemple des étapes de création d'un labyrinthe de taille 9×11 :



Question 5 Écrire une fonction `laby_rec : int -> int -> int array array` qui prend en argument deux entiers ℓ et h et renvoie une matrice représentant un labyrinthe de taille $\ell \times h$. Cette fonction devra utiliser une fonction auxiliaire récursive `laby : int * int -> int -> int -> unit` qui prend en argument un couple (i, j) et deux dimensions ℓ et h. On tracera le labyrinthe avec la fonction `dessiner`.

Il est toujours intéressant de voir se dessiner le labyrinthe au fur et à mesure. Cette méthode de tracé ne le permet pas, car le labyrinthe est dessiné d'un seul coup. Pour voir la chronologie, nous utiliserons la commande suivante, qui permet de stopper le processus pendant un temps donné :

```
#load "unix.cma";;
let minisleep (sec: float) =
    ignore (Unix.select [] [] [] sec);;
```

Ainsi, la commande `minisleep 0.01;` permet de mettre le système en pause pendant 1 centième de seconde.

Question 6 Modifier la fonction `colorie` pour qu'elle marque une pause après avoir dessiné. Modifier la fonction `laby_rec` pour qu'elle ouvre une fenêtre graphique, la remplisse de noir, puis dessine les cases libres au fur et à mesure que la matrice se remplit de 1. On choisira la bonne taille de la fenêtre lors de son ouverture.

2 Recherche de chemin

Nous nous intéressons ici à de la recherche de chemin dans un labyrinthe, à connaissance totale (c'est-à-dire qu'on connait à l'avance la structure du labyrinthe, contrairement à de la recherche à l'aveugle, à l'instar de Thésée dans le labyrinthe de Dédale).

Question 7 Pour évaluer les différentes techniques de recherche, nous souhaitons éviter de modifier la matrice du labyrinthe testé. Écrire une fonction `deepcopy : 'a array array -> 'a array array` qui copie une matrice donnée en argument.

Nous rappelons l'existence des modules `Stack` et `Queue` (représentant les piles et files), ainsi que les méthodes associées : `is_empty`, `push`, `pop` et `top`.

2.1 Parcours en profondeur

Comme son nom l'indique, il s'agit ici de faire un parcours en profondeur de l'arbre du labyrinthe. Pour cela, nous utiliserons une structure de pile. Le principe est le suivant :

- On commence par créer une pile contenant le sommet $(1, 1)$ et une matrice `pred` de prédécesseurs.
- Si la pile est non vide, on dépile un sommet, on cherche ses pièces voisines non visitées et on les ajoute à la pile. On modifie alors la matrice `pred` en y mettant les bons prédécesseurs pour les pièces et les cloisons concernées.
- Si l'un des sommets est le sommet en haut à droite, on reconstruit le chemin grâce au tableau de prédécesseurs.

Question 8 Écrire une fonction `parcours_prof : int array array -> unit` qui dessine une copie de la matrice, puis applique l'algorithme de parcours en largeur. La fonction devra utiliser des fonctions auxiliaires récursives. On coloriera en rouge les cases dans la pile, en jaune les cases explorées et en vert le chemin de résolution.

2.2 Parcours en largeur

Cette implémentation est quasi-identique à la précédente, sauf qu'on utilise une file plutôt qu'une pile.

Question 9 Écrire une fonction `parcours_larg : int array array -> unit` qui applique l'algorithme du parcours en largeur.

3 Générations avancées

3.1 Génération en profondeur

Cette génération crée de longues ramifications avec peu d'intersections. On écrira quelques fonctions utilitaires :

Question 10 Écrire une fonction `interieur : int array array -> int * int -> bool` qui indique si la case donnée en argument se trouve dans l'enceinte de creusage du labyrinthe donné en argument.
Écrire une fonction `cloisons : int array array -> int * int -> (int * int) list` qui renvoie la liste des cloisons voisines à une pièce telles que les pièces voisines n'ont pas déjà été creusées. On mélangera la liste dans un ordre aléatoire en utilisant la fonction suivante :

```
let shuffle l =
    let q = List.map (fun c -> (Random.bits (), c)) l in
    let r = List.sort compare q in
    List.map snd r;;
```

Le principe de la génération en profondeur est le suivant :

- Initialement, on creuse un trou dans le coin en bas à gauche, et on insère ses deux cloisons voisines dans un ordre aléatoire dans une pile vide.
- Tant que la pile est non vide, on y retire une cloison, et on procède comme suit :
  - On regarde si le labyrinthe crée une boucle, c'est-à-dire si son successeur (la pièce adjacente dans la direction opposée d'où on vient) est creusée ou non.
  - S'il n'y a pas de boucle et que les cases concernées sont dans l'enceinte du labyrinthe, on creuse la cloison retirée et son successeur, puis on ajoute chacun des 3 cloisons voisines non creusés du successeur à la pile.

Question 11 Écrire une fonction `laby_prof : int -> int -> int array array` qui prend en argument la largeur et la hauteur (impaires) du labyrinthe et qui renvoie sa matrice en générant un labyrinthe selon ce principe. On fera en sorte de dessiner le labyrinthe au fur et à mesure, comme indiqué en question 6. On coloriera en rouge les cases présentes dans la table aléatoire, en pensant à les remettre en noir dans le cas où une boucle est créée.
Remarque : On pourra prendre en compte le prédécesseur des cloisons dans la pile.

3.2 Traversée aléatoire

Pour cette génération, nous allons devoir utiliser une structure de données permettant de retirer un élément aléatoire de la structure. Comme usuellement, nous souhaitons des complexités raisonnables ($O(1)$ en moyenne) pour les différentes opérations. Pour éviter de créer un tableau trop grand et occuper trop d'espace mémoire, nous utiliserons

des tables de hachage. Afin d'éviter d'écrire le nom du module systématiquement, on peut charger toutes les fonctions par la commande :

```
open Hashtbl;;
```

On rappelle les opérations usuelles :

- `create : int -> ('a, b) t` : crée une table vide d'une certaine taille maximale (la taille n'a pas d'importance, car on parle ici de tables dynamiques).
- `add : ('a, 'b) t -> 'a -> 'b -> unit` : ajoute un couple dans une table.
- `find : ('a, 'b) t -> 'a -> 'b` : renvoie la valeur associée à une clé.
- `remove : ('a, 'b) t -> 'a -> unit` : supprime une association à une clé.
- `replace : ('a, 'b) t -> 'a -> 'b -> unit` : remplace une association.

Le type de table aléatoire utilisé sera le suivant :

```
type 'a rtbl = {data : (int, 'a) t; mutable n : int};;
```

La valeur n permet de garder en mémoire le nombre de données stockées. Elle vaut 0 initialement. Pour ajouter un élément, on l'insère dans la table avec la clé n, puis on incrémente n. Pour retirer un élément, on choisit un nombre aléatoire k entre 0 et $n-1$, on échange les valeurs aux clés k et $n-1$, puis on retire la valeur associée à $n-1$ (en modifiant n) en la renvoyant.

Question 12 Écrire des fonctions associées aux tables : `creer : unit -> 'a rtbl`, `vide : 'a rtbl -> bool`, `inserer : 'a -> 'a rtbl -> unit` et `retirer : 'a rtbl -> 'a` selon ce principe.

Le principe de la traversée aléatoire ressemble à celui de la génération en profondeur, mais l'ordre de traitement des cases est géré différemment, car la structure est la table aléatoire.

Question 13 Écrire une fonction `laby_trav : int -> int -> int array array` qui dessine un labyrinthe par une traversée aléatoire.

3.3 Algorithme de Wilson

L'algorithme de Wilson est particulier, car au lieu de partir du labyrinthe déjà creusé, il part d'une pièce non reliée, et crée un chemin aléatoirement pour relier cette pièce au reste du labyrinthe. Le principe est le suivant :

- On crée une pile dite de **sondage**, initialement vide. On creuse la pièce en bas à droite.
- Pour chaque pièce du labyrinthe, si elle n'est pas déjà creusée, on ajoute cette pièce à la pile et on lance une procédure de sondage :
  - On regarde la pièce du sommet de la pile et on choisit aléatoirement une de ses pièces voisines, qui n'est pas son prédécesseur.
  - Si la pièce choisie est une pièce du labyrinthe, alors on creuse toute la pile de sondage, et on termine la procédure de sondage.
  - Si la pièce choisie est une pièce déjà sondée, alors on enlève de la pile de sondage toutes les pièces jusqu'à retomber à nouveau sur la pièce déjà sondée, puis on relance la procédure de sondage.
  - Sinon, on ajoute à la pile de sondage la pièce choisie puis on relance la procédure de sondage.

L'algorithme peut être frustrant à ses débuts, car tant qu'il y a peu de pièces dans le labyrinthe, la procédure de sondage peut durer longtemps jusqu'à rencontrer une pièce déjà creusée.

Question 14 Écrire une fonction `wilson : int -> int -> int array array` qui génère un labyrinthe selon ce principe. On coloriera en rouge les pièces en cours de sondage (ainsi que les cloisons entre ces pièces).

4 Flooding

Le **flooding** consiste à colorier les cases creusées d'un labyrinthe en fonction de leur distance par rapport à l'origine. Il permet de constater les différences de structure entre les différents algorithmes de création.

Question 15 Écrire une fonction `couleur : int -> color` qui prend en argument un nombre entre 0 et 767 et renvoie une couleur du module graphique. On utilisera la fonction `rgb : int -> int -> int -> color` qui renvoie une couleur selon le code RGB (Red, Green, Blue). On parcourra les couleurs continument dans l'ordre Rouge →

Vert → Bleu.

Pour le flooding, on utilise un parcours en largeur, et on ajoute à la file d'attente les coordonnées d'un point, ainsi que son numéro de couleur. Lors du défilement, on colorie le sommet par sa couleur et chaque voisin est alors ajouté, en incrémentant le numéro de couleur (modulo 768).

Question 16 Écrire une fonction `flooding : int array array -> unit` qui remplit le labyrinthe de couleurs selon ce principe.

5 Utilisation d'une file de priorité

Une file de priorité est une structure de données permettant de stocker des données associées à une priorité. Lorsqu'on retire un élément de la file de priorité, on retire celui avec la priorité la plus faible (ou la plus élevée, selon l'implémentation). Cette structure sera étudiée plus en détail en deuxième année. Il n'existe pas de structure déjà implémentée en Caml, mais nous recopierons le code donné en exemple pour la création de module, à l'adresse `https://caml.inria.fr/pub/docs/manual-ocaml/moduleexamples.html`. Nous utiliserons alors les commandes `PrioQueue.empty`, `PrioQueue.insert` et `PrioQueue.remove_top` pour manipuler la structure (les signatures sont indiquées une fois le code exécuté).

Remarque

> Attention, il s'agit d'une structure persistante et non impérative. On utilisera des fonctions récursives.

5.1 L'algorithme A*

Cet algorithme améliore l'efficacité de la recherche de chemin. Sa structure est exactement la même que celle des parcours en profondeur et en largeur, mais utilise une file de priorité. Lorsqu'on doit choisir un sommet dans la file de priorité, on choisit celui qui « a le plus de chance de nous rapprocher de l'arrivée », c'est-à-dire celui qui minimise la distance euclidienne au sommet d'arrivée.

Question 17 Écrire une fonction `astar : int array array -> unit` qui fait la recherche du chemin dans un labyrinthe selon cet algorithme.

5.2 L'algorithme de Prim

Cet algorithme permet une nouvelle génération de labyrinthe. Son principe est très proche de la traversée aléatoire, en utilisant une file de priorité. Ce qui change est que chaque fois qu'une cloison est ajoutée à la file de priorité, elle est ajoutée avec une priorité aléatoire (qui doit être un nombre entier).

Question 18 Écrire une fonction `prim : int -> int -> int array array` qui génère un labyrinthe selon ce principe.

6 Conclusion

Question 19 Tester les différents algorithmes de recherche de chemin et le flooding sur les différents types de labyrinthes créés. On modifiera notamment la taille du pixel à 1, on enlèvera la pause lors du dessin d'un carré, et on dessinera des labyrinthes de taille de la résolution de l'écran.

Question 20 S'inspirer des fonctions écrites pour proposer de nouvelles versions de générations de labyrinthes (forme autre que rectangulaire, à maille hexagonale, nouvelles contraintes d'algorithmes, ...)
On pourra effectuer des recherches pour trouver de nouveaux algorithmes. Les meilleures idées d'implémentations qui me seront envoyées par mail recevront un point bonus au premier devoir de l'année de spé (si je vous ai en cours) !