Organisation du travail :

Pour ce projet on s'est réuni plusieurs fois afin de travailler ensemble sur la mise en place de la structure de donnée représentant nos labyrinthe.
On a donc créé ensemble le type grille ainsi que ses différents attributs, sous attributs.
On a crée des fonctions de vérifications de grille, de lecture de grille et d'affichage.
On a également implémenté la fonction de gestion de ligne de commande.

En revanche pour la partie résolution d'un labyrinthe et génération d'un labyrinthe aléatoire on s'est partagé le travail : Lounès KEBDI s'est chargé de la mise en place de fonctions nécessaires pour résoudre tout labyrinthe de taille n*m tandis que Yassine ALIF s'est lui occupé de la partie génération de labyrinthe aléatoire (n*m).

Chaque fonction écrite a été testé immédiatement pour éviter d'avoir des bugs complexes à gérer.

Nous avons rédigé ensemble ce rapport.

Présentation générale de la structure de données de grille

Un labyrinthe peut-être modéliser de différentes manières. En effet, on peut représenter un labyrinthe mathématiquement comme un arbre, ce qui relève du domaine de la théorie des graphes, comme on peut le représenter par un ensemble de matrices avec par exemple une matrice pour les murs verticaux et une matrice pour les murs horizontaux.

Nous avons pris la décision de modéliser nos labyrinthes à l'aide d'un type enregistrement grille qui a pour attribut un array de array représentant toutes les cases de la grille, ainsi que deux entiers représentants ses dimensions.

Chaque élément de notre array de array représente une case qui est caractérisée par plusieurs attributs : deux entiers pour ses coordonnées, un entier représentant la distance de sortie ainsi qu'un type somme situation qui représente la case. Une case peut être un mur horizontal, vertical, vide, une entrée, une sortie, un coin ou encore un chemin.

Illustration des types implémentés :

```
type situation = V | MH | MV | C | E | S | Chemin ;;
type case = {l: int ; c: int ; mutable s: situation ; mutable dis_S: int };;
type grille ={t: case array array; nb_l : int; nb_c: int};;
```

Pour la vérification de nos grilles on a créé les fonctions suivantes :

```
val contientEntreeEtSortie : grille -> unit
val tableau_correct : grille -> unit
val verifieGrille : grille -> unit
```

Ces dernières permettent de vérifier que toute grille (Labyrinthe) contient bien une entrée et une sortie que les différentes cases de la grille sont correctes en particulier celles représentant l'enceinte du labyrinthe …

Implémentation de la grille de référence :

On a implémenté une fonction qui crée une grille vide de taille n*m avec pour particularité que toutes les cellules du labyrinthe sont entourées de 4 murs (gain de temps pour la génération de labyrinthes aléatoires) :

```
val grille_vide : int -> int -> grille
```

Fonctionnalités de lecture et d'affichage

On a par la suite implémenté deux types d'affichage :
un basique correspondant aux fichiers tests donnés, et un autre utilisant des séquences Unicode graphique.

```
val affiche_situation : situation -> unit
val affiche_case : case -> unit
val affiche_ligne : case array -> unit
val affiche_grille : grille -> unit
val affiche_case_complex : case -> int -> int -> unit
val affiche_ligne_complex : case array -> int -> int -> unit
val affiche_grille_complex : grille -> unit
```

Les fonctions de lectures ont-elles été données par le sujet. On a seulement pris le soin d'ajouter une fonction qui converti le fichier lu en une grille.

Fonctionnalités de résolution :

```
val trouve_E : grille -> int * int
val trouve_S : grille -> int * int
val resolution : grille -> unit
val coord_prochain_chemin : grille -> int -> int -> int*int
val grille_resolue : grille -> unit
```

Fonctionnalités de générations :

On a utilisé l'algorithme donné par le professeur. Directions aléatoires permet de parcourir la grille aléatoirement.

```
val caseAleatoire : int -> int -> int * int
val directionsAleatoires : string list
val choisiCaseNonVisitee : grille -> int -> int -> int * int
val generate : int -> int -> grille
```

Présentation d’une difficulté rencontrée dans l’implémentation

Dans le cadre de notre projet, nous avons fait face à une difficulté assez importante lors de la phase de modélisation de la structure du labyrinthe. En effet, notre première implémentation de structure étant assez rudimentaire, elle ne nous permettait pas de prendre en compte des éléments cruciaux d'un labyrinthe complexe.

Une première légère difficulté fut l’orientation des murs. Nous avions initialement limité les murs à une seule orientation, ne permettant pas de prendre en compte des configurations plus complexes, telles que des murs pouvant être aussi bien verticaux qu'horizontaux, voire formant des coins.

Face à cette limitation, nous avons pris la décision conjointe de réviser notre approche pour améliorer la représentation du labyrinthe. Notre première modification a consisté à étendre notre type somme pour inclure la possibilité de représenter des murs sous différentes orientations, améliorant ainsi la flexibilité de notre modélisation.

La modification la plus importante de notre projet a été l’ajout d’un attribut de distance à la sortie pour chaque case du labyrinthe. Cette ajout a simplifié notre processus de résolution du labyrinthe et a eu un impact significatif sur notre génération aléatoire. Nous avons modifié cet attribut pour indiquer si nous étions déjà passés sur une case particulière, optimisant ainsi notre processus de génération aléatoire en utilisant la distance à la sortie comme critère de construction.

En collaborant sur ce projet, nous avons tiré des leçons importantes sur l'importance de la communication et de la révision régulière de nos conceptions. Les difficultés que nous avons rencontrées ont souligné la nécessité d'une approche flexible et itérative dans notre processus de développement logiciel collaboratif.

En conclusion, les ajustements successifs apportés à notre structure de labyrinthe ont résolu la difficulté initiale, améliorant notre représentation, notre résolution et notre génération aléatoire du labyrinthe. Cette expérience a renforcé notre compréhension des enjeux et défis de modélisation de structures.

Description des tests faits :

Pour la partie lecture et conversion d'un fichier en une grille, on a commencé par faire des tests visuels. C'est-à-dire que l'on a simplement fait appel à nos fonctions d'affichage pour vérifier si nos fonctions étaient correctes.

Ensuite on a implémenté des fonctions vérifiant vraiment les attributs d'un pseudo-labyrinthe. On a donc écrit une fonction verifieGrille qui prend en paramètre une grille et vérifie si elle contient bien une entrée une entrée et une sortie et que les différentes cases de la grille sont correctes (enceinte fermée, les cellules soient de même tailles, etc).

```
val contientEntreeEtSortie : grille -> unit
val tableau_correct : grille -> unit
val verifieGrille : grille -> unit
```

Pour la fonction d'initialisation de grille vide on l'a juste exécuté avec la commande ./maze.exe init 8 8 > test/test_initialisation_8x8.laby qui redirige la sortie standard dans le fichier test/test_initialisation_8x8.laby. On l'a testé sur différentes tailles de grilles (carrées ou non).

Pour la partie résolution du projet, on a résolu tous les labyrinthes proposés par le professeur et on les affiché afin de vérifier que la résolution trouvée reliée bien les bonnes cases et correspondait bien au plus cours chemin possible (on ne passe par plusieurs fois par la même case). En particulier on a la commande suivante ./maze.exe solve --pretty test/maze_100x100.laby 100 100 > test/test_resolution_100x100.laby qui résolve le labyrinthe taille 100x100 et redirige la résolution dans le fichier test/test_resolution_100x100.laby.

Enfin pour la partie aléatoire on a vérifié les grilles crées avec la fonction vérifie_grille présentée précédemment et on les a résolue pour vérifier si il respecté bien la connexité demandé par le labyrinthe. En particulier on a la commande suivante ./maze.exe random --pretty 100 100 > test/test_random_100x100.laby qui génère un labyrinthe taille 100x100 et redirige le labyrinthe ainsi que sa solution dans le fichier test/test_random_100x100.laby.

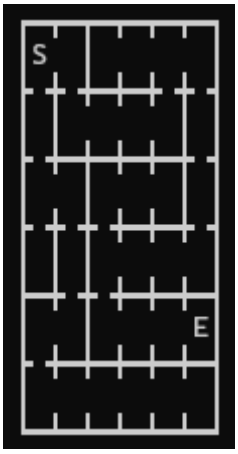
Pour finir on a testé notre affichage complexe avec les fonctions précédentes.

Description du type d'affichage amélioré choisi

On a choisi comme affichage amélioré celui nécessitant des séquences Unicodes graphiques. Pour cela, on seulement du remplacer les caractères utilisés pour l'affichage basique par des caractères plus jolis que l'on a copié collé sur le site suivant : https://en.wikipedia.org/wiki/Box-drawing_character

Avec cet affichage on est passé d'un affichage basique comme celui-ci-dessous :

à cet affichage :



qui est tout de même plus joli à voir.