Rapport
de
Projet de Programmation C

Génération aléatoire de labyrinthe

KHEDR Nader
SURAT Rémi

Année universitaire 2019 / 2020
L2 Informatique

Table des matières

Introduction

Ce projet nous a été demandé dans le cadre de la matière *Algorithmique des arbres* enseignée par Mr. *Marc Zipstein* au second semestre de la deuxième année de licence informatique à l'UPEM (Université Paris-Est Marne-la-Vallee).

Le but est d'écrire un programme qui génère des labyrinthes aléatoirement et les affichent sur le terminal ou sur une fenêtre de la librairie MLV.

Plusieurs options peuvent êtres sélectionnées lors du lancement du programme, nous les détaillerons toutes dans le manuel utilisateur.

Certains algorithmes du programme sont détaillés dans la partie développeur ainsi que les difficultés rencontrées et un aspect global du programme (modules, fonctions principales).

Manuel utilisateur

Exécution du programme

Le projet Labyrinthe est conçu en plusieurs modules. Un Makefile contenu dans le dossier du projet permet de faciliter la compilation.

Pour compiler le programme il faut se placer dans le dossier, la ou est contenu le fichier ***Main.c*** et taper la commande :

make

Un exécutable de nom ***Labyrinthe*** sera créé, pour l'exécuter il faut taper la commande suivante :

./Main --option1 --option2 … --optionN

où ***option1, option1 … optionN*** sont les options sélectionnés pour la génération du labyrinthe.

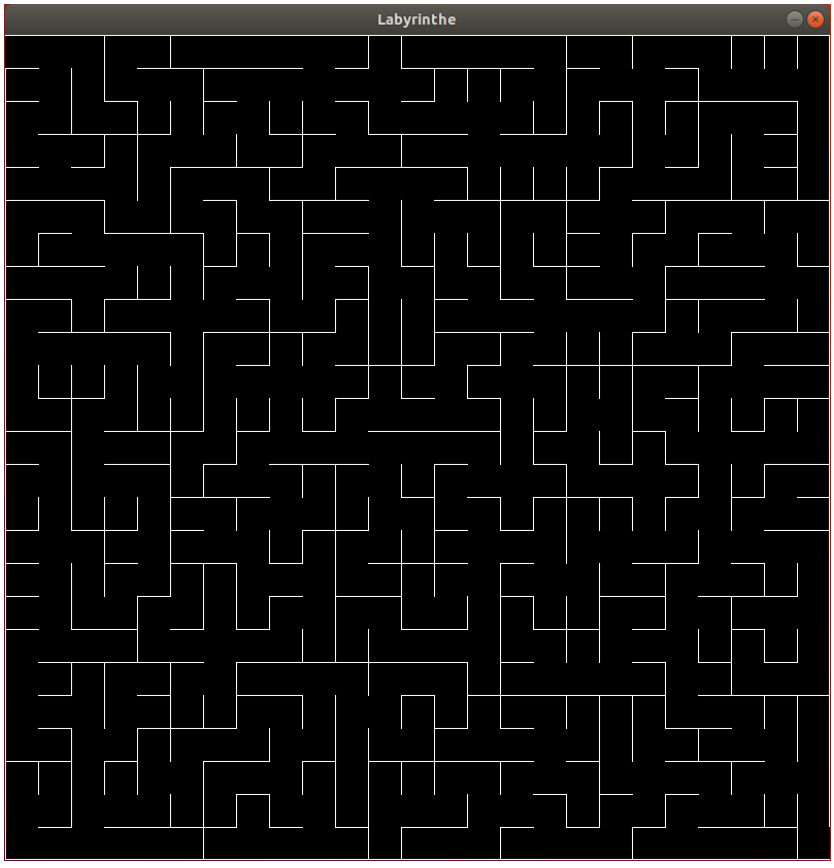

Illustration 1.

Options disponibles

Par défaut, l'exécution du programme sans options lance un affichage en mode graphique d'un labyrinthe de taille 25x25.

Le labyrinthe est généré de façon pseudo-aléatoire à l'aide d'une graine initialisée par le temps courant. Le programme lancé plusieurs fois à des heures différentes affiche donc des labyrinthes différents.

De plus le programme affiche un labyrinthe ne contenant que des cellules totalement fermées et attend une entrée clavier de l'utilisateur pour supprimer un mur, jusqu'à ce que le labyrinthe soit dit « valide », c'est à dire qu'il existe un chemin entre l'entrée et la sortie.

Certains aspects de la génération du labyrinthe peuvent être modifiés, voici la liste exhaustive de toutes les options disponibles lors de l'exécution du programme :

- --***mode=texte :*** Lance le programme en mode texte (sans interface graphique).
- --***taille=axb :*** Modifie la taille du labyrinthe (hauteur (« a ») et largeur (« b »)).
- --***graine=X :*** Fixe la graine d'aléatoire (ainsi deux appels au programme avec une même graine génère un même labyrinthe).
- --***attente=0 :*** Affiche sans attente la labyrinthe final.
- --***attente=X :*** Le programme attends « X » millisecondes entre chaque suppression de mur lors de la génération du labyrinthe.
- --***unique :*** Empêche la suppression d'un mur s'il sépare deux cellules appartenant déjà à un même chemin. Cette option rend le labyrinthe plus esthétique.
- --***acces :*** Rend toutes les cellules accessibles. Cette option renforce la difficulté du labyrinthe.
- --***victor :*** Affiche le chemin de longueur minimale reliant l'entrée et la sortie.
- --***help :*** Affiche le manuel utilisateur.

Ainsi l'exécution du programme selon la commande suivante :

./Main --taille=25x25 --attente=0 --unique --acces

Affiche directement (sans afficher les étapes de suppression des murs) un labyrinthe en mode graphique de hauteur 25 et de largeur 25 où toutes les cellules sont accessibles *(Illustration 1)*.

Partie développeur

Choix de conception et fonctions importantes

Modules

Notre réflexion lors du choix des différents modules pour notre projet à été la suivante. Il fallait distinguer, selon nous, trois aspects distincts du programme :

- Un concernant l'affichage (graphique et terminal).
- Un pour la conception à proprement dite du labyrinthe (initialisation, génération et libération de l'espace mémoire alloué pour le labyrinthe).
- Un pour la structure de donnée *Unionfind* qui permettra de représenter les chemins de cellules par des ensembles disjoints. Ici nous aurons besoin d'initialiser la structure et de faire les fonctions permettant de trouver le représentant d'un ensemble et d'unir deux ensembles.
- Nous nous sommes rapidement rendus compte que nous aurions besoin de fonctions pour afficher et libérer les tableaux utilisés dans la structure du labyrinthe et dans la structure d'*Unionfind*.
- Puis nous avons choisit de créer un module pour gérer tous les affichages d'erreurs et les manuels d'utilisation des différentes options du programme.
- Finalement, après avoir conçu la partie principale du programme, nous avons choisis d'implémenter l'amélioration permettant d'afficher le plus court chemin qui relie l'entrée à la sortie, nous avons créé pour cela un module pour gérer les files.

Les modules sont donc les suivants

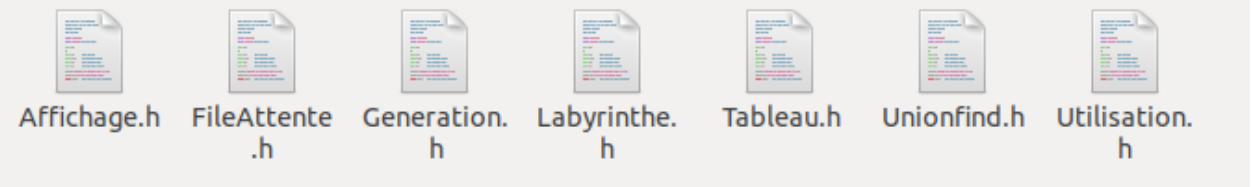

Illustration 2.

Graphe d’inclusions

Certains modules dépendent les uns des autres, voici le graphe d’inclusion représentant les dépendances de chaque module :

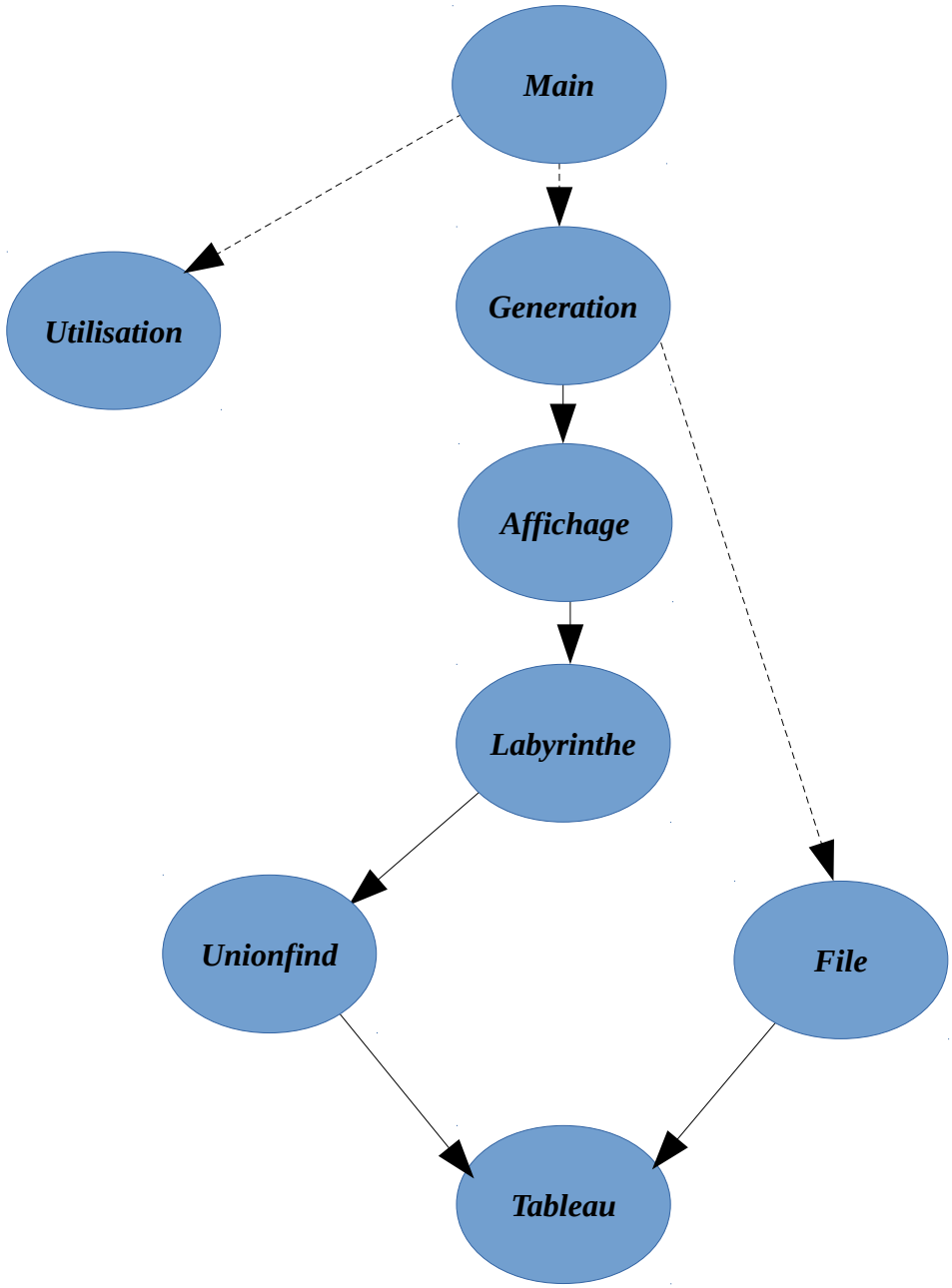

Figure 1.

Les flèches en pointillées représentent l’inclusion d’un fichier .h dans un fichier.c, celles pleines représentent l’inclusion d’un fichier.h dans un fichier.h.

Génération du labyrinthe

1. **Sélection des options selon les arguments fournis par l'utilisateur**

   Pour cet aspect nous avons choisis d'utiliser la librairie ***getopt*** qui permet de faciliter la gestion des options et de diminuer considérablement le code.

2. **Initialisation du labyrinthe**

   La structure ***OptionsArg*** contient toutes les valeurs des options choisies. Par défaut, les valeurs sont les suivantes :

   mode_texte = 0 ***attente*** = -1 ***taille.abscisse*** = 25

   unique = 0 ***acces*** = 0 ***taille.ordonnee*** = 25

   graine = srand(time(NULL))

```
10  typedef struct OptionsArg {
11      Coordonnees_t taille;
12      int mode_texte;
13      int graine;
14      int attente;
15      int unique;
16      int acces;
17      int victor;
18  } OptionsArg;
19
```

Illustration 3.

3. **Génération du labyrinthe**

   La fonction ***genererLabyrinthe*** du module ***Generation*** est la fonction principale du programme, elle génère et renvoie les ensembles de cellules correspondant au labyrinthe dans une structure de donnée *Unionfind.*

   La génération du labyrinthe se produit de la façon suivante :

   a) La fonction ***initialiserEnsembles*** initialise ***n * m*** ensembles de cellules contenant chacun une unique cellule qui est donc son propre représentant.

   b) La condition d'arrêt de l'algorithme est sélectionnée selon l'option ***acces.***

   c) Un mur est aléatoirement sélectionné (cet aspect sera détaillé dans la suite de ce rapport) et si l'option ***unique*** est sélectionnée, un mur n'est supprimé que s'il séparait deux cellules n'appartenant pas déjà dans le même ensemble.

   d) Finalement le programme produit ou non un affichage selon la valeur de l'option ***attente*** et selon le mode d'affichage précisé.

Implémentation des options

Chemin unique

La fonction ***estDansMemeClasse*** du module ***Unionfind*** permet d'implémenter l'option **unique**.

Elle détermine simplement si les représentants des deux cellules séparées par le mur sont les mêmes et empêche dans ce cas la suppression du mur si l'option est sélectionnée.

```
88
89  int estDansMemeClasse(Unionfind_t ensembles_cel, Coordonnees_t cel_1, Coordonnees_t cel_2){
90      Coordonnees_t repre_cel_1;
91      Coordonnees_t repre_cel_2;
92
93      repre_cel_1 = trouverRepresentant(ensembles_cel.pere, cel_1);
94      repre_cel_2 = trouverRepresentant(ensembles_cel.pere, cel_2);
95
96      if((repre_cel_1.abscisse == repre_cel_2.abscisse) && (repre_cel_1.ordonnee == repre_cel_2.ordonnee))
97          return 1;
98      return 0;
99  }
```

Illustration 4.

Accessibilité des cellules

Pour implémenter cette option nous avons choisi d'ajouter une variable ***cardinalite*** à la structure ***Unionfind_t.***

Cette variable représente le nombre d'ensembles de cellules et est donc initialisée à ***n * m*** où n et m sont la hauteur et la largeur du labyrinthe.

Chaque fois que deux ensembles sont réunis, la cardinalité est décrémentée, ainsi l'algorithme de génération du labyrinthe ne s'arrête que lorsque la cardinalité vaut 1 et que donc toutes les cellules sont dans le même ensemble.

De cette façon la complexité est minimale car chaque cellule n'est pas systématiquement vérifiée.

```
 9  typedef struct unionfind_t {
10      Coordonnees_t taille;
11      Coordonnees_t **pere;
12      unsigned int **rang;
13      int cardinalite;
14  } Unionfind_t;
15
```

Illustration 5.

Meilleur affichage en mode texte

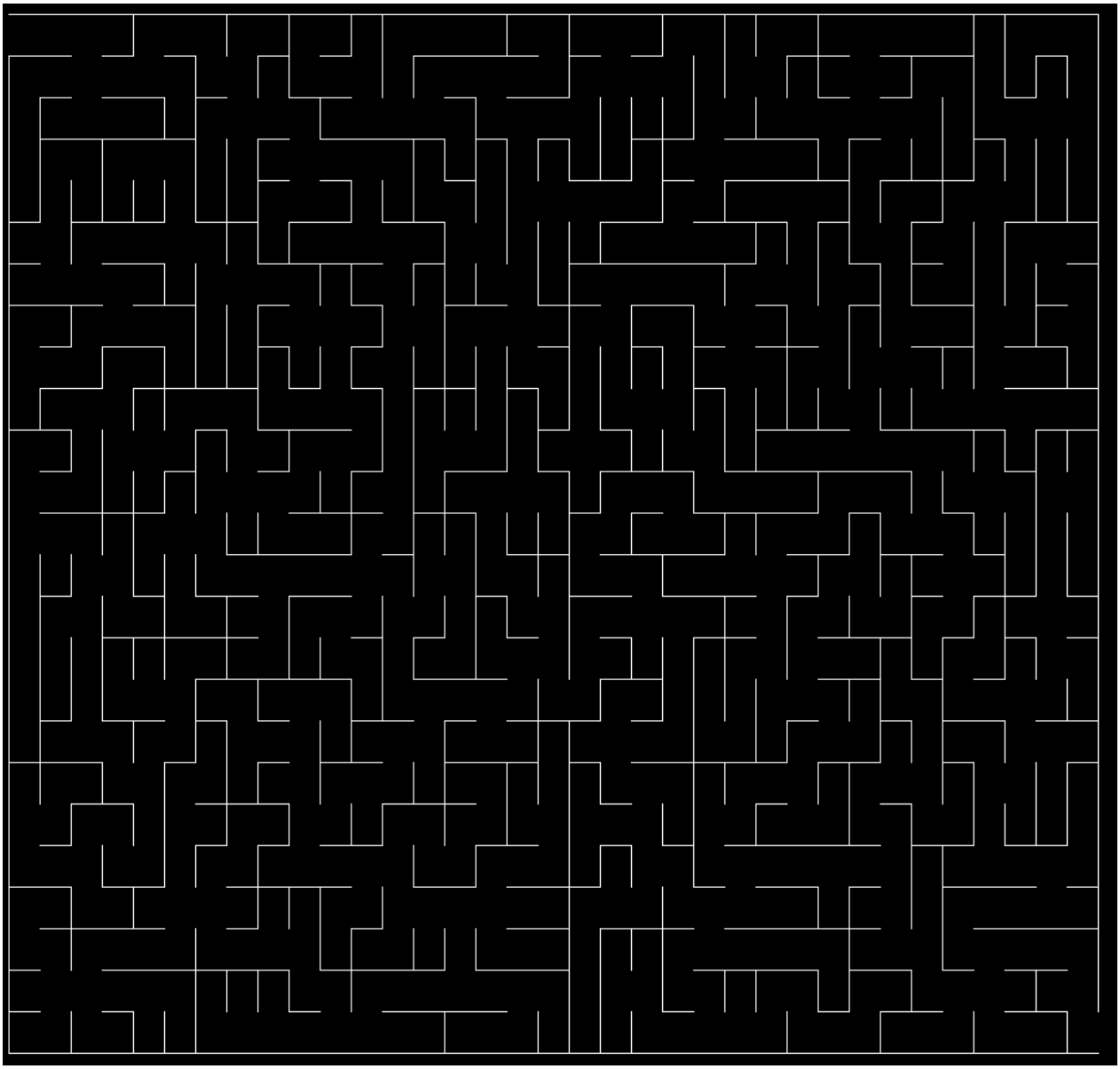

Illustration 6.

Le labyrinthe ci dessus à été affiché en mode texte utf8.

Pour implémenter cette fonctionnalité nous sommes partis d’un labyrinthe de taille 3*3 pour trouver les lignes de codes nécessaires, puis nous avons généralisé la conjecture.

Chemin victorieux

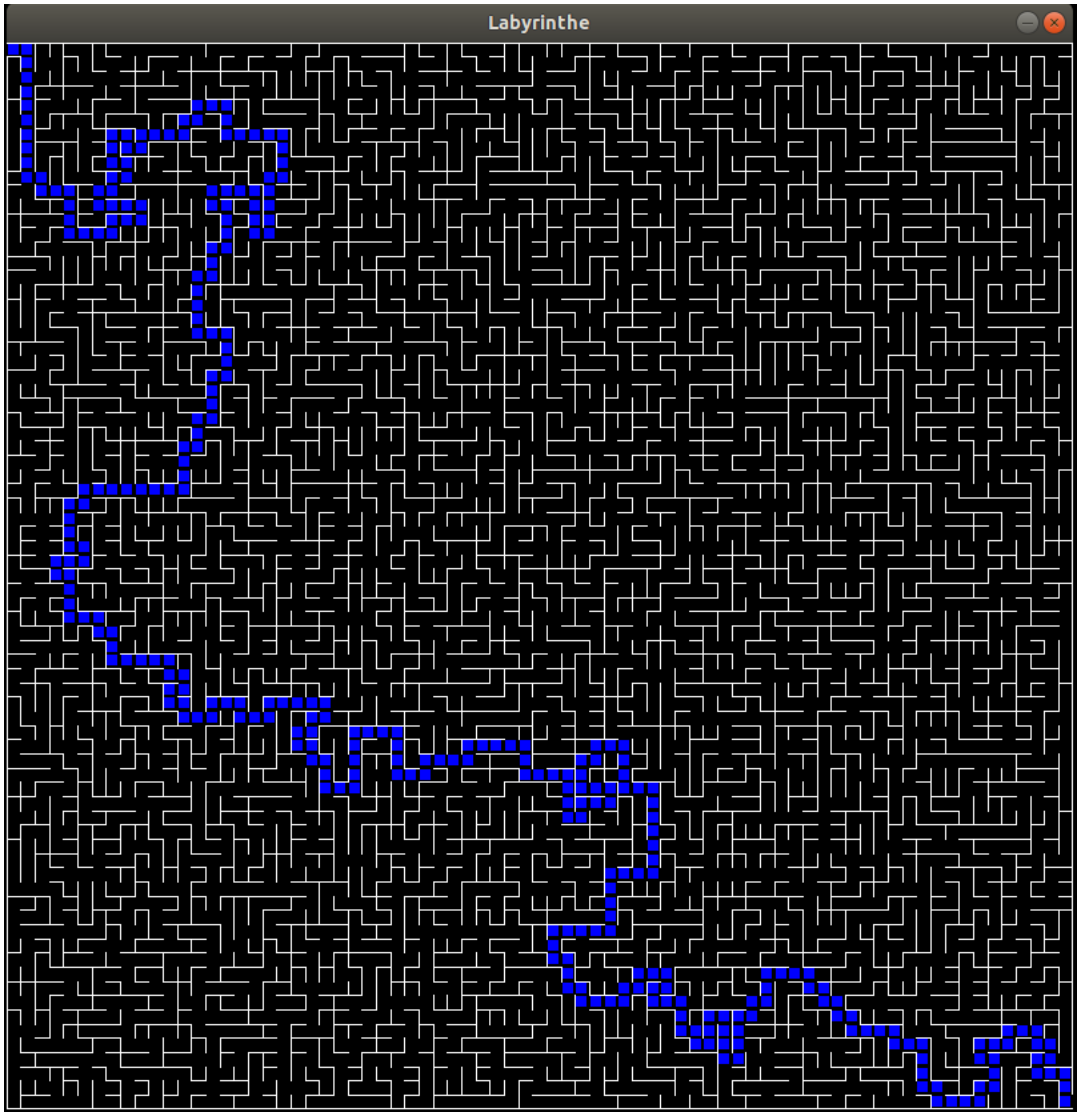

Illustration 7.

L'implémentation de cette dernière amélioration à été la plus difficile et intéressante.

Nous avons choisi d'utiliser les murs du labyrinthe pour afficher le plus court chemin. L'algorithme utilisé est un algorithme de « ***backtracking*** », il test tous les chemins et revient à une étape précédente lorsqu'il se trouve dans une impasse.

Nous réutilisons ici le tableau de pères du labyrinthe, nous nous en servirons pour afficher le plus court chemin à la fin de l'algorithme. Celui ci se déroule de la façon suivante.

Une cellule est dite voisine lorsqu'elle n'est pas séparée par un mur et qu'elle n'a pas déjà été visitée. Dans l'algorithme on remarquera qu'une cellule (i, j) déjà visitée n'est plus son propre père dans le tableau de pères.

Nous avons codé les fonctions nécessaires à l'implémentation des files dans le module ***FileAttente.***

L'algorithme utilisé :

- On part de la cellule (0, 0) et on enfile ses cellules voisines en notant pour chacune d'elles que la cellule (0, 0) est leur cellule père dans le tableau.
- Tant que la file n'est pas vide :
  - On enfile les cellules dites voisines en notant leur cellule père (la cellule actuelle).
  - On défile la cellule actuelle
- On ajoute dans une liste chaînée le plus court chemin en partant de la cellule (n, m) où n et m sont respectivement la largeur et la hauteur du labyrinthe, il faut ajouter en début de file afin d'inverser l'ordre et finalement d'afficher le plus court chemin en partant de l'entrée.
- On par parcours la liste chaînée et on colorie chaque cellule.

Le schéma ci dessous montre un exemple de l'algorithme sur un labyrinthe de taille 3 * 3.

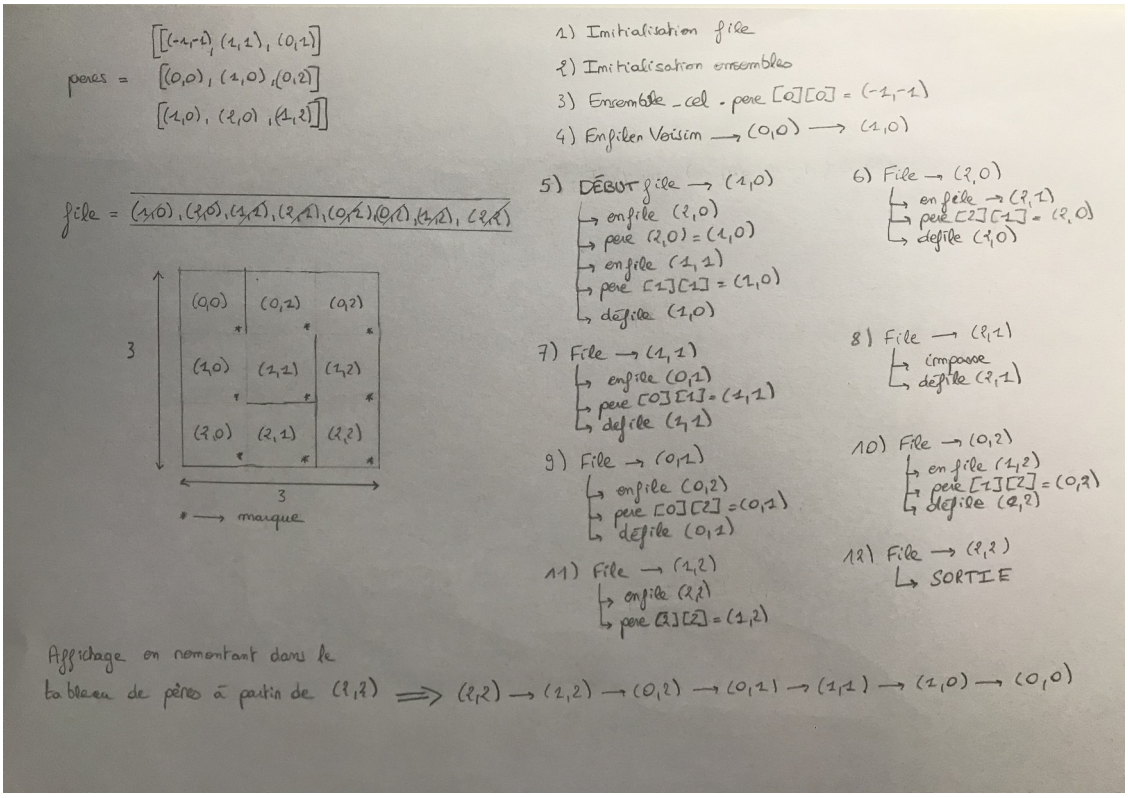

Illustration 8.

Performances

Temps d’exécution selon les options choisies

Nous avons mesuré les performances du programme selon les options choisies et selon différentes tailles de labyrinthe, voici un tableau récapitulatif :

Taille / Options	**Sans option**	**--acces**	**--unique**
25 * 25	instantané	instantané	instantané
100 * 100	~ 20 ms	~ 75 ms	~ 20 ms
200 * 200	~ 200 ms	~ 1,175 s	~ 100 ms
300 * 300	~ 5 s	~ 8 s	~ 3 s
400 * 400	~ 10 s	~ 25 s	~ 8 s
500 * 500	~ 30 s	~ 65 s	~ 27 s

Ces calculs sont très approximatifs mais ils nous montrent que l’option **acces** fait très rapidement augmenter le temps d’exécution et que l’option **unique** le fait baisser très légèrement.

Les autres options ne modifient pas le temps d’exécution.

Nous avons utilisé les fonctions de mesure du temps de la bibliothèque standard **<*sys/time.h*>** car elles permettent une mesure optimal.

La fonction ***intervalle_ms*** permet de mesurer le temps écoulé, à la milliseconde près entre deux temps.

```
18  /* Renvoie en millisecondes la durée écoulée entre 'debut' et 'fin' */
19  static int intervalle_ms(struct timeval debut, struct timeval fin) {
20      int temps_ecoule = (fin.tv_sec - debut.tv_sec) * 1000;
21      return temps_ecoule + (fin.tv_usec - debut.tv_usec) / 1000;
22  }
23
```

Illustration 9.

Amélioration du tirage aléatoire des murs

Une amélioration majeure des performances de génération du labyrinthe consiste à améliorer le tirage aléatoire des murs.

En effet il a fallut trouver un moyen d'empêcher de tirer aléatoirement un mur déjà supprimé du labyrinthe, ce qui diminue nettement la complexité.

Nous avons pour cela créé un tableau d'entiers de taille ***n * m * 2*** où n et m sont respectivement la largeur et la hauteur du labyrinthe.

Il faut multiplier par 2 la taille du tableau pour représenter les murs verticaux et les murs horizontaux.

Nous avons ensuite initialisé le tableau de sorte qu'il contienne les ***n * m - 1*** premiers entiers naturels, ils représenterons toutes les combinaisons de coordonnées possibles.

Ensuite nous mélangeons ce tableau plusieurs fois pour rendre le tirage des murs pseudo aléatoire puis nous parcourons simplement le tableau pour choisir le mur à supprimer.

L'utilisation d'un tableau permet une bien meilleure complexité car l'accès à un élément est en O(1).

Pertes de mémoire

Nous avons veillé à faire attention aux pertes de mémoires lors de la conception de ce projet, nous avons pour cela utilisé ***valgrind.***

Les fonctions ***libererEnsemblesCel*** et ***libererLabyrinthe*** désallouent l'espace mémoire précédemment alloué.

Nous avons cependant remarqué que la librairie MLV causait des pertes de mémoires irrécupérables, le projet n'est donc pas parfait sur cet aspect.

Difficultés rencontrées

Nous avons eu quelques difficultés lors de la conception de l'amélioration pour l'affichage du plus court chemin.

Nous avons d'abord tenté, en vain, d'utiliser la structure de donnée ***Unionfind*** afin de créer l'arbre correspondant au tableau de pères et d'implémenter un parcours en largeur sur celui ci, chaque niveau de l'arbre représentant une étape du chemin.

Cependant nous avons été rapidement confrontés à plusieurs problèmes, en effet la fusion par rang et la compression des chemins utilisée dans ***Unionfind*** fausse immanquablement la construction de l'arbre et il est impossible de retracer le plus court chemin.

De plus le tableau de pères représente un arbre dont les liens sont représentés vers le haut, la parcours en largeur est donc difficile…

Conclusion

Ce projet nous a permis de travailler plusieurs aspects de la programmation avancée en C.

Modularité, algorithmes avancés (chemin victorieux), meilleure gestion des options à l'aide de librairie ***getopt.***

Nous avons notamment observé l'importance des performances, surtout qu'ici il ne s'agit que d'un « petit » projet et nous avons donc utilisé des moyens pour réduire la complexité.

Nous sommes assez fiers de ce projet, il nous aura notamment permit de parfaitement comprendre l'utilisation de la structure de donnée ***Unionfind*** qui peut être utilisée dans de nombreux projets.