



ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE ET D'ANALYSE DES  
SYSTÈMES - RABAT

---

## Rapport de Projet de programmation : Labyrinthe

---

*Réalisé par :*

Maha DRISSI EL-BOUZAI  
Oulaya ELARGAB

*Encadré par :*

Pr. RADOUANE MOHAMED





## *Remerciements :*

Nous tenons à exprimer nos remerciements les plus sincères à notre encadrant et nos professeurs qui n'ont ménagé aucun effort pour contribuer à la réussite de ce projet en partageant leurs conseils et remarques les plus pertinents.



# Table des matières

<b>1</b>	<b>Présentation du projet</b>	<b>1</b>
1.1	Sujet . . . . .	1
1.2	Étymologie . . . . .	2
1.3	Principe du labyrinthe . . . . .	2
<b>2</b>	<b>Analyse théorique</b>	<b>3</b>
2.1	Cahier de charges . . . . .	3
2.1.1	Conception du labyrinthe . . . . .	3
2.1.2	Conception du programme . . . . .	3
2.2	Méthode de génération du labyrinthe . . . . .	4
2.3	Méthode de résolution . . . . .	5
<b>3</b>	<b>Réalisation du projet</b>	<b>6</b>
3.1	Difficultés rencontrées . . . . .	6
3.1.1	Compréhension du sujet . . . . .	6
3.1.2	Programmation des fonctions . . . . .	6
3.1.3	Rédaction et organisation du code . . . . .	6
3.1.4	Contrainte de temps . . . . .	6
3.1.5	Code Source. . . . .	7
3.1.6	Interface graphique SDL . . . . .	7
3.1.7	Rédaction du rapport . . . . .	7
3.2	Fonctions principales du code source . . . . .	8
3.2.1	Les directives des préprocesseurs . . . . .	8
3.2.2	Définition des constantes . . . . .	8
3.2.3	Définition de la structure utilisée . . . . .	8
3.2.4	Définition des variables globales utilisées . . . . .	8
3.2.5	Fonction <i>initParametres</i> . . . . .	8
3.2.6	Fonction <i>initCell</i> . . . . .	9
3.2.7	Fonction <i>index</i> . . . . .	9
3.2.8	Fonction <i>checkNeighbors</i> . . . . .	9
3.2.9	Fonction <i>removeWalls</i> . . . . .	9
3.2.10	Fonction <i>drawCell</i> . . . . .	9
3.2.11	Fonction <i>empiler</i> . . . . .	10
3.2.12	Fonction <i>depiler</i> . . . . .	10
3.2.13	Fonction <i>loadMap</i> . . . . .	10
3.2.14	Fonction <i>defineDistance</i> . . . . .	10
3.2.15	Fonction <i>findNextInPath</i> . . . . .	10
3.2.16	Fonction <i>findPath</i> . . . . .	11
3.2.17	Fonction <i>pathTo</i> . . . . .	11
3.2.18	Fonction <i>generateMaze</i> . . . . .	11
3.2.19	Fonction <i>resolveMaze</i> . . . . .	11
3.2.20	Fonction <i>createWindow</i> . . . . .	11
3.2.21	Fonction <i>main</i> . . . . .	12
3.2.22	functions.h . . . . .	13
<b>4</b>	<b>Interface graphique sur SDL</b>	<b>14</b>
4.1	Génération du labyrinthe . . . . .	14
4.2	Recherche du plus court chemin vers la première sortie aléatoire . . . . .	15
4.3	Recherche du plus court chemin vers la deuxième sortie aléatoire . . . . .	16



# Chapitre 1

## Présentation du projet

PROJET C s'agit de produire un programme d'environ 500 lignes de code afin de valider les compétences des cours : « Algorithmique », « Technique de programmation » et « Structures de données ». Le programme correspond à 20 heures de travail effectives en langage C. Les étudiants travaillent en binôme et bénéficient des conseils d'un professeur encadrant <sup>1</sup>

### 1.1 Sujet

Un labyrinthe est un tracé sinueux, muni ou non d'embranchements, d'impasses et de fausses pistes, destiné à perdre ou à ralentir celui qui cherche à s'y déplacer.(cf.fig.1.1) De nos jours, le terme de labyrinthe désigne une organisation complexe, tortueuse, concrète (architecture, urbanisme, jardins, paysages...) ou abstraite (structures, façons de penser...), où la personne peut se perdre. Le cheminement du labyrinthe est difficile à suivre et à saisir dans sa globalité.

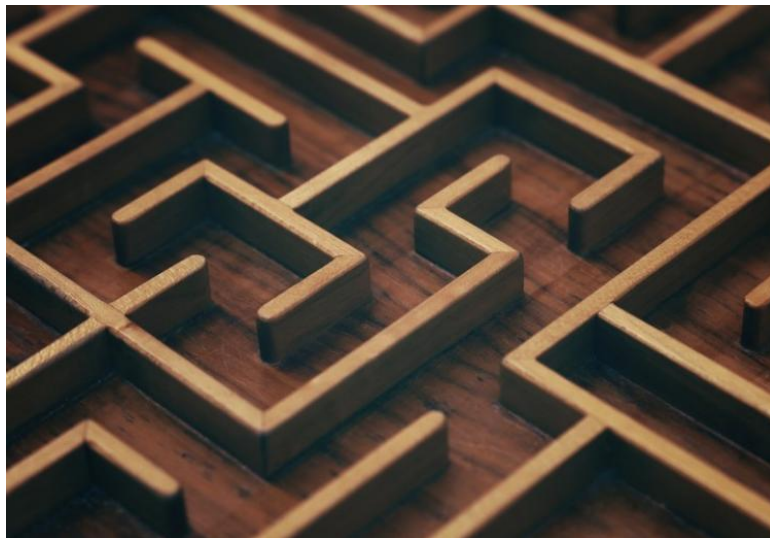


FIGURE 1.1 – Labyrinthe

---

1. Notice du projet de programmation 2020/2021

## 1.2 Étymologie

L'origine du mot est vraisemblablement préhellénique mais indo-européenne, s'expliquant comme le « tout-en-pierres ». On a tenté autrefois divers rapprochements étymologiques, par exemple avec le terme grec labrys, nom de la hache crétoise à double tranchant, avec laquelle aurait été creusé le labyrinthe. La forme la plus ancienne connue est da-pu2-ri-to- en mycénien de Cnossos (le signe pu2 note ordinairement phu), avec d à l'initiale et non l .

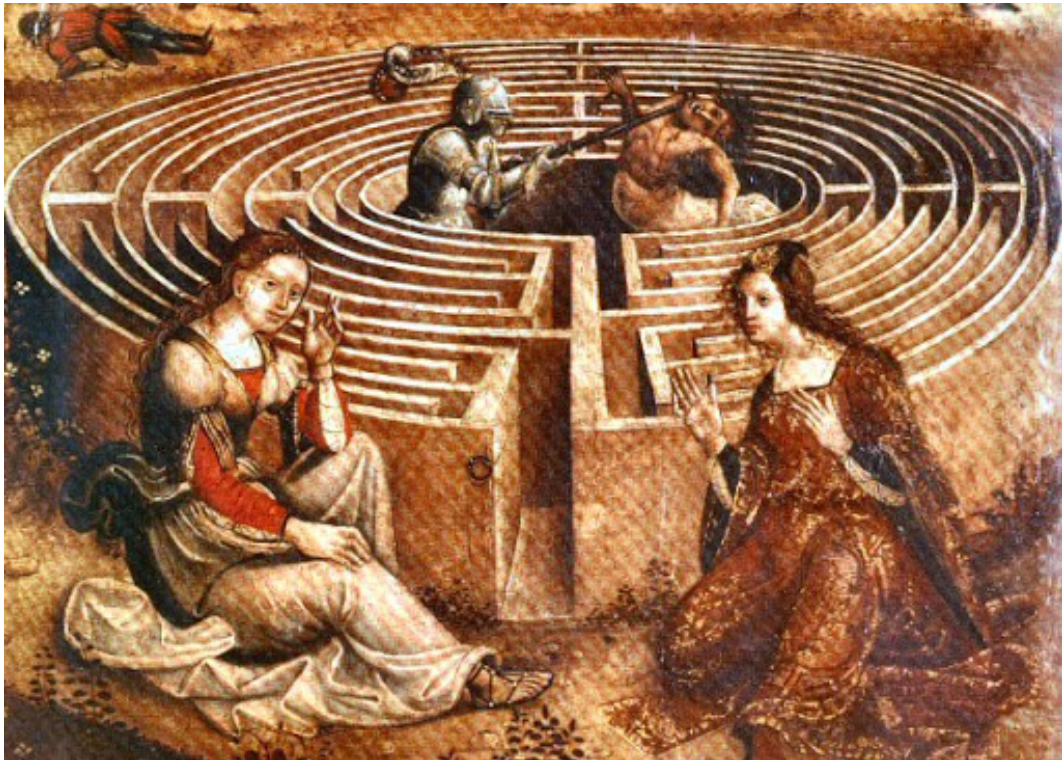


FIGURE 1.2 – Labyrinthe dans l'étymologie grecque

## 1.3 Principe du labyrinthe

Les labyrinthes peuvent être étudiés comme des objets mathématiques, c'est la modélisation mathématique du labyrinthe. Deux aspects importants de cette modélisation sont la génération automatique du labyrinthe et sa résolution.

En fait, la génération du labyrinthe consiste à construire une grille de cellules séparées par des murs. Ce labyrinthe peut être généré par plusieurs méthodes. On cite : **Randomized Prim's algorithm**, **Randomized Depth-First Search (DFS)**, **Randomized Kruskal's algorithm ...**

Quant à la résolution, elle consiste à trouver le plus court chemin qui mène à la sortie du labyrinthe modélisé mathématiquement. Il existe plusieurs algorithmes pour ce problème. On note : **Random mouse algorithm**, **Pledge algorithm**, **Trémaux's algorithm** et **Breadth First Search (BFS)**.

# Chapitre 2

## Analyse théorique

Ce chapitre permet de faire une analyse théorique du labyrinthe. En effet, cette conception est cruciale afin de comprendre la totalité des principes et les coder en se basant sur le cahier de charges fournit.

### 2.1 Cahier de charges

Le cahier de charges présente l'ensemble des instructions et contraintes qui cadrent la réalisation du labyrinthe. Le cahier de charges disponible donne des instructions qu'on va élaborer dans cette partie.

#### 2.1.1 Conception du labyrinthe

Il s'agit de concevoir un labyrinthe en 2D avec une entrée et deux sorties aléatoires, proposer et de concevoir un algorithme qui recherche efficacement l'une des deux sorties à partir de la position d'entrée :

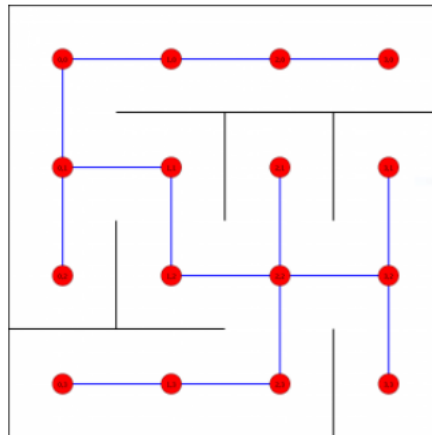


FIGURE 2.1 – Exemple proposé de labyrinthe

#### 2.1.2 Conception du programme

La conception du labyrinthe se fait en deux phases. La première phase consiste à la génération du labyrinthe à l'aide de l'algorithme DFS (Depth First Search). La deuxième phase tourne autour de la résolution du labyrinthe, en adoptant l'algorithme BFS (Breath First Search) qui consiste à trouver le plus court chemin liant une entrée aléatoire avec l'une des deux sorties.



Dans cette partie, nous allons discuter la methode sur laquelle s'appuie la g n ration du labyrinthe. L'algorithme de g n ration se base sur l'espace discr t s  dont les cellules carr es sont initialement remplies et s par es par des cloisons, selon les quatre directions (nord, sud, est et ouest). On part d'un labyrinthe o  tous les murs de toutes les cellules sont ferm s. Chaque cellule contient une variable bool enne qui indique si la cellule a d j   t  visit e ou non (i.e. les cellules visit es sont celles qui appartiennent au chemin du labyrinthe en cours de construction).

A central square labeled "Cellule" has four arrows pointing outwards. The top arrow is labeled "Nord", the bottom arrow is labeled "Sud", the left arrow is labeled "Ouest", and the right arrow is labeled "Est".

[illegible]

4

## 2.3 Méthode de résolution

Pour la phase de résolution nous avons utilisé l'algorithme BFS (Breath First Search) en divisant le travail en deux grandes étapes :

- La première étape de l'algorithme consiste à affecter à chaque cellule une distance la séparant de la cellule de sortie aléatoirement choisie. En fait, ceci revient à considérer cette dernière comme la racine d'un arbre, où les nœuds sont des cellules et les arêtes sont les murs les séparant et dont la longueur est égale à 1. On commence alors par donner à la racine le potentiel 0. Ensuite en explorant l'arbre constitué des cellules du labyrinthe et des murs les séparant, on affecte à chaque nœud le potentiel de son précédent plus la longueur de l'arête qui est égale à 1. A l'issue de ce traitement, le potentiel du nœud représente la distance le séparant de la racine. On répète ce processus jusqu'à avoir affecté une distance à toutes les cellules du labyrinthe.

- La deuxième étape est celle de la recherche du plus court chemin. Pour ceci, on part de la cellule d'entrée, et parmi les cellules l'avoisinant, on choisit celle qui a la plus courte distance par rapport à la cellule de sortie, préalablement déterminée dans la première étape. Ensuite, on répète le même algorithme pour la cellule choisie et ainsi de suite jusqu'à arriver à la cellule de sortie.

Le processus est le même pour la première et la deuxième sorties.

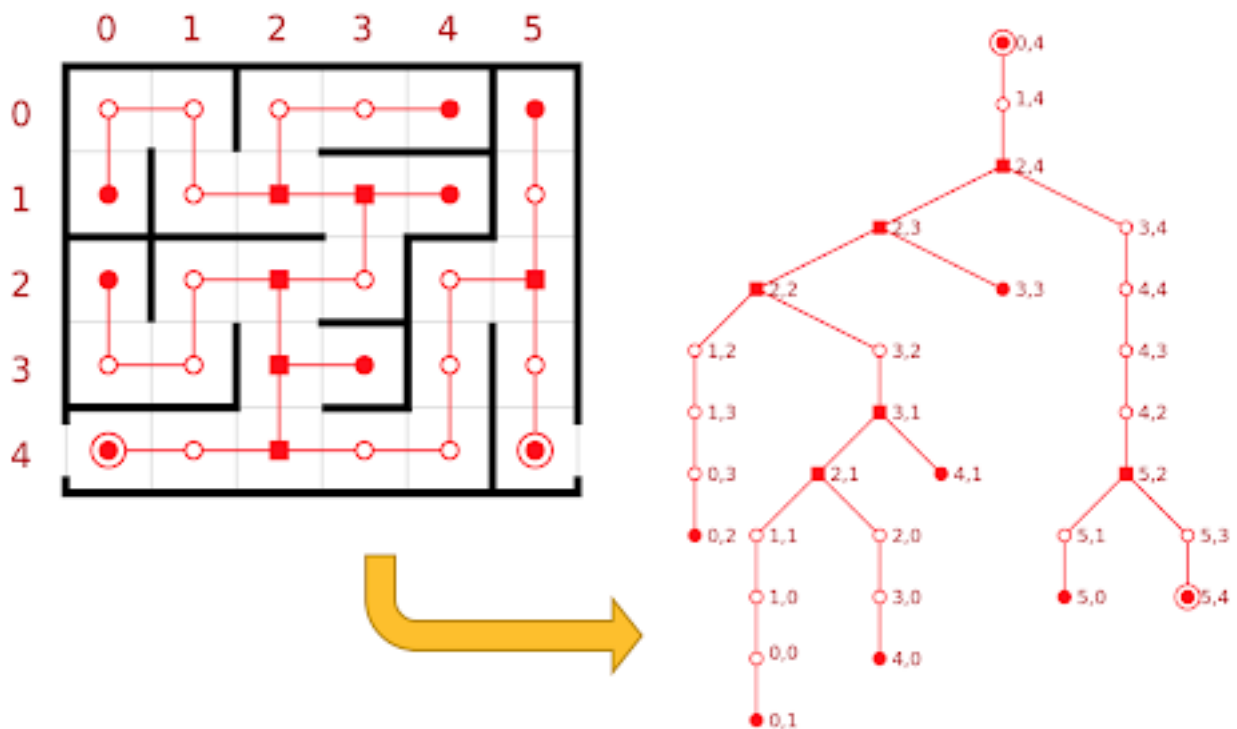


FIGURE 2.4 – Recherche du plus court chemin à l'aide de l'algorithme BFS

## Chapitre 3

# Réalisation du projet

Dans ce chapitre nous allons citer les plus grandes difficultés rencontrées lors de la réalisation du projet. Ainsi, nous allons expliquer le rôle des fonctions figurant dans le code source.

### 3.1 Difficultés rencontrées

Cette partie détaille la démarche de réalisation du projet et les problèmes qui l'ont accompagnée.

#### 3.1.1 Compréhension du sujet

La difficulté du sujet ne réside pas dans la compréhension du concept mais plutôt dans son implémentation avec un langage de programmation ( le cas ici du langage C )et l'utilisation de la bibliothèque SDL.

#### 3.1.2 Programmation des fonctions

Programmer ce labyrinthe s'est accompagné par plusieurs difficultés, car c'est notre premier contact réel avec le langage C. Les erreurs de compilations étaient indispensables partout dans le code source.

#### 3.1.3 Rédaction et organisation du code

Ecrire un *clean code* est un défi pour tous les développeurs. Nous avons essayé de garantir le maximum de la clarté. En effet, nous avons commenté le code d'une façon pertinente. De plus, nous avons divisé le code en plusieurs fichiers ".c " et des fichiers ".h ".

#### 3.1.4 Contrainte de temps

La découverte de plusieurs technologies durant ce projet a consommé pas mal de temps. Nous pensons que le délai était suffisant mais trop serré.

### 3.1.5 Code Source.

Le code source de ce projet qui a été partagé entre 3 fichiers, le fichier ".c", le fichier ".h" et le fichier main a été implémenté sur le compilateur de Code Blocks .



FIGURE 3.1 – Logo de Code Blocks

### 3.1.6 Interface graphique SDL

Afin de représenter graphiquement l’affichage du code source, nous avons essayé de manipuler la librairie SDL (Simple DirectMedia Layer). Découvrir la méthode de fonctionnement de cette librairie était un obstacle majeur.



FIGURE 3.2 – Logo de la librairie SDL

### 3.1.7 Rédaction du rapport

La documentation professionnelle nécessite la manipulation du logiciel de traitement de texte LaTeX. Travailler avec ce dernier est inévitable tôt ou tard, donc nous avons voulu exploiter cette opportunité et explorer LaTeX.



FIGURE 3.3 – Logo du logiciel LaTeX

## 3.2 Fonctions principales du code source

Comme vu précédemment dans le premier chapitre, chaque instruction nécessite une ou plusieurs fonctions. Dans cette partie nous allons élaborer les fonctions principales utilisés dans le code source.

### 3.2.1 Les directives des préprocesseurs

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include "functions.h"
5 #ifndef functions_h
6 #define functions_h
```

Ces directives ont comme rôle l'importation des bibliothèques et des fichiers *header* contenant les fonctions à utiliser dans le main.

### 3.2.2 Définition des constantes

```
1 #define MAX_CELL 6000
```

cette constante représente le nombre maximal de cellules qui définissent la forme du labyrinthe qu'on va générer (proposé dans le cahier des charges).

### 3.2.3 Définition de la structure utilisée

```
1 typedef struct _cellule
2 {
3     int i; // la ligne ou se trouve la cellule
4     int j; // colonne ou se trouve la cellule
5     int x; // top left abscisse
6     int y; // top left ordonnee
7     short int walls[4]; // tableau designant si un mur de la cellule existe (true) ou pas (
8     // false) walls[0] pour mur en haut ,// walls[1] pour mur a droite , walls[2] pour mur en bas
9     // , walls[3] pour mur a gauche.
10    short int visited; // c'est un boolean pour designer si une cellule a deja ete visitee
11    // durant le processus de generation du labyrinthe
12    int distance; // distance par rapport a la cellule de sortie;
13    int affected; // un boolean qui designe si une cellule on lui a deja affecte la
14    // distance
15    struct _cellule *next; // next voisin a visiter
16    struct _cellule *previous; // previous voisin for stacking
17 } cell;
```

### 3.2.4 Définition des variables globales utilisées

```
1 cell cells[MAX_CELL]; // tableau qui stocke les cellules du maze
2 cell *cellAct; // la cellule actuellement visitee par le programme
3 cell *cellSortie1; // la cellule de la premiere sortie
4 cell *cellSortie2; // la cellule de la deuxieme sortie
5 cell *cellEntree; // la cellule d'entree
6 int nbreCols; // nombre de colonnes du labyrinthe
7 int nbrLignes; // nombre de lignes du labyrinthe
8 int nbrCells; // nombre de cellules du labyrinthe
9 int cell_size; // la taille de la cellule inseree par l'utilisateur
10 SDL_Window *window; // la fenetre principale
11 SDL_Renderer *renderer; // le rendu de la fenetre
12 //PILE path; // la pile ou l'on stocke le chemin vers la sortie
```

### 3.2.5 Fonction *initParametres*

```
1 void initParametres(int width, int height, int cellsize);
```

Cette fonction a comme rôle l'initialisation des paramètres du labyrinthe, qui va être généré, à partir de données insérées par l'utilisateur .

Input : int width ; c'est la largeur de la fenêtre, int height : la longueur de la fenêtre, int cellsize : la taille de

chaque cellule du labyrinthe

Output : (void)

### 3.2.6 Fonction *initCell*

```
1 void initCell(cell *c, int i, int j);
```

Cette fonction a comme rôle la définition de la cellule Cell qui constitue l'unité de base de notre labyrinthe.

Input : c ( structure de type Cell ), i (la ligne où se trouve la cellule de type entier), j ( la colonne où se trouve la cellule de type entier)

Output : (void)

### 3.2.7 Fonction *index*

```
1 int index(int i, int j);
```

Cette fonction a comme rôle de repérer le positionnement de chaque cellule du labyrinthe.

Input : i (entier qui précise la ligne où se trouve le Cell), j (entier qui précise la colonne où se trouve le Cell)

Output : l'index de la cellule de la ième ligne jème colonne dans le tableau cells

### 3.2.8 Fonction *checkNeighbors*

```
1 cell * checkNeighbors(int i, int j);
```

Cette fonction a comme rôle d'examiner les cellules non encore visitées situées soit à gauche, à droite, en haut ou en bas. Elle teste les voisins de la cellule dont le numéro de ligne est i et le numéro de colonne est j.

Input : i (entier qui précise la ligne où se trouve le Cell), j (entier qui précise la colonne où se trouve le Cell)

Output : un pointeur de type Cell qui pointe sur la cellule choisie parmi les cellules voisines de la cellule dont les coordonnées sont i et j.

### 3.2.9 Fonction *removeWalls*

```
1 void removeWalls(cell* a, cell* b);
```

Cette fonction comme l'indique son nom élimine les deux walls qui existent entre les deux cells a et b afin de construire un chemin.

Input : a , b ( structure de type Cell )

Output : (void)

### 3.2.10 Fonction *drawCell*

```
1 void drawCell(cell *c);
```

Cette fonction a comme rôle la création de la cellule Cell du labyrinthe à l'aide de la bibliothèque SDL.

Input : c structure de type Cell

Output : (void)

### 3.2.11 Fonction *empiler*

```
1 void empiler(PILE **p, cell *c);
```

Cette fonction a comme rôle la sauvegarde des cellules visitées pour poursuivre le traitement avec les autres restantes.

Input : p pointeur de type PILE , c structure de type Cell

Output : (void)

### 3.2.12 Fonction *depiler*

```
1 cell* depiler(PILE **p);
```

On a recours à cette fonction lorsque on n'est dans une position où toutes les cellules qui encadrent la cellule courante sont déjà visitées. Elle retourne la cellule dépilée.

Input : p pointeur de type PILE

Output : structure de type Cell

### 3.2.13 Fonction *loadMap*

```
1 void loadMap(int lignes , int colonnes , int * cells , cell cellules []);
```

Cette fonction a comme rôle de dessiner la grille du départ cellule par cellule.

Input : lignes, colonnes de type int, cells pointeur de type int et un tableau de type structure Cell

Output : (void)

### 3.2.14 Fonction *defineDistance*

```
1 void defineDistance(cell * exitCell);
```

Cette fonction a comme rôle de déterminer la distance qui sépare nowCell de la cellule de sortie.

Input : exitCell structure de type Cell

Output : (void)

### 3.2.15 Fonction *findNextInPath*

```
1 cell * findNextInPath(cell* cellA);
```

Cette fonction a comme rôle de retourner la cellule optimale qui a la plus courte distance entre cellA et la cellule de sortie .

Input : cellA structure de type Cell

Output : cellule de type Cell

### 3.2.16 Fonction *findPath*

```
1 void findPath( cell * enterCell );
```

Cette fonction a comme rôle de trouver le plus court chemin vers la sortie.

Input : enterCell cellule de type Cell

Output : (void)

### 3.2.17 Fonction *pathTo*

```
1 void pathTo( cell* entree , cell* sortie , SDL_Color color );
```

Cette fonction a comme rôle de déterminer le plus court chemin entre entree et sortie et le dessiner avec la couleur color.

### 3.2.18 Fonction *generateMaze*

```
1 void generateMaze( cell* enterCell );
```

Cette fonction est définie pour générer le maze.

Input : enterCell cellule de type Cell

Output : (void)

### 3.2.19 Fonction *resolveMaze*

```
1 void resolveMaze( cell* enterCell , cell* exitCell1 , cell* exitCell2 );
```

Cette fonction a comme rôle la résolution du labyrinthe ;

Input : enterCell, exitCell1, exitCell2 : cellules de type Cell

Output : (void)

### 3.2.20 Fonction *createWindow*

```
1 void createWindow( int width , int height , int cellsize );
```

Cette fonction englobe les fonctions de la bibliothèque SDL nécessaires pour la génération du labyrinthe. C'est la fenêtre où l'on genere le labyrinthe et on le résout en faisant appel aux autres fonctions. Input : int width ; c'est la largeur de la fenêtre, int height : la longueur de la fenêtre, int cellsize : la taille de chaque cellule du labyrinthe

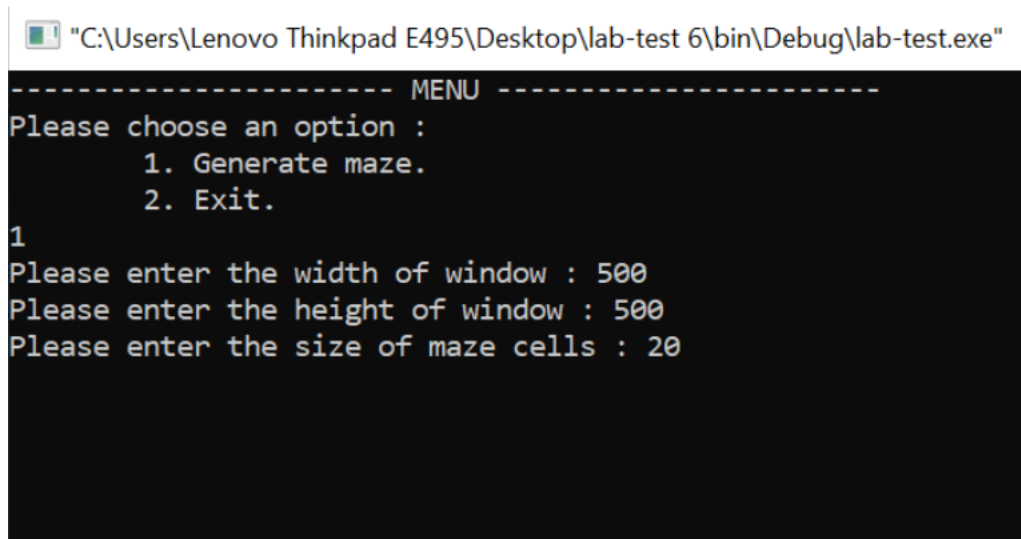
Output : (void)



### 3.2.21 Fonction *main*

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <SDL2/SDL.h>
4 #include "functions.h"
5
6
7 void menu(int *width, int *height, int *cellsize)
8 {
9     printf("Please enter the width of window : ");
10    scanf("%d",width);
11    printf("Please enter the height of window : ");
12    scanf("%d",height);
13    printf("Please enter the size of maze cells : ");
14    scanf("%d",cellsize);
15 }
16 int main(int argc, char* args[])
17 {
18     int choice=0, width,height,cellsize;
19     printf("----- MENU ----- \n");
20     printf("Please choose an option : \n");
21     printf("\t1. Generate maze.\n");
22     printf("\t2. Exit.\n");
23     scanf("%d",&choice);
24     if(choice == 1)
25     {
26         menu(&width,&height,&cellsize);
27         printf(" %d %d %d\n",width,height,cellsize);
28         createWindow(width,height,cellsize);
29     }
30     else if (choice == 2)      exit(0);
31     return EXIT_SUCCESS;
32 }
```

Cette fonction est le main de notre programme. L'utilisateur choisit les parametres du labyrinthe à générer ( ici c'est le cas d'un labyrinthe de longueur 500, largeur 500 et la taille de chaque cellule est 20 ).



```
"C:\Users\Lenovo Thinkpad E495\Desktop\lab-test 6\bin\Debug\lab-test.exe"
----- MENU -----
Please choose an option :
    1. Generate maze.
    2. Exit.
1
Please enter the width of window : 500
Please enter the height of window : 500
Please enter the size of maze cells : 20
```

FIGURE 3.4 – Output général la fonction *main* dans la console

### 3.2.22 functions.h

```
1 void initParametres(int width, int height, int cellsize);
2 void initCell(cell *c, int i, int j);
3 int index(int i, int j);
4 cell * checkNeighbors(int i, int j, int *prevRand);
5 void removeWalls(cell* a, cell* b);
6 void colorVisitedCell(cell* c, SDL_Color line_color);
7 void createWindow();
8 void drawCell(cell *c);
9 void empiler(PILE **p, cell *c);
10 cell* depiler(PILE **p);
11 void loadMap(int lignes, int colonnes, int * cells, cell cellules[]);
12 void defineDistance(cell * exitCell);
13 cell * findNextInPath(cell* cellA);
14 void findPath(cell * enterCell);
15 void colorCellPath(cell *c, SDL_Color line_color);
16 void pathTo(cell* entree, cell* sortie, SDL_Color color);
17 void generateMaze(cell* enterCell);
18 void resolveMaze(cell* enterCell, cell* exitCell1, cell* exitCell2);
19 void createWindow(int width, int height, int cellsize);
```

Ce fichier .h contient une appellation de toutes les fonctions utilisées.

## Chapitre 4

# Interface graphique sur SDL

Dans ce chapitre, nous allons valider les attentes du projet à travers une interface graphique en utilisant la bibliothèque SDL.

### 4.1 Génération du labyrinthe

Dans cette partie du programme, après l'exécution, l'utilisateur est demandé de donner les paramètres du labyrinthe qui veut créer selon son propre choix.

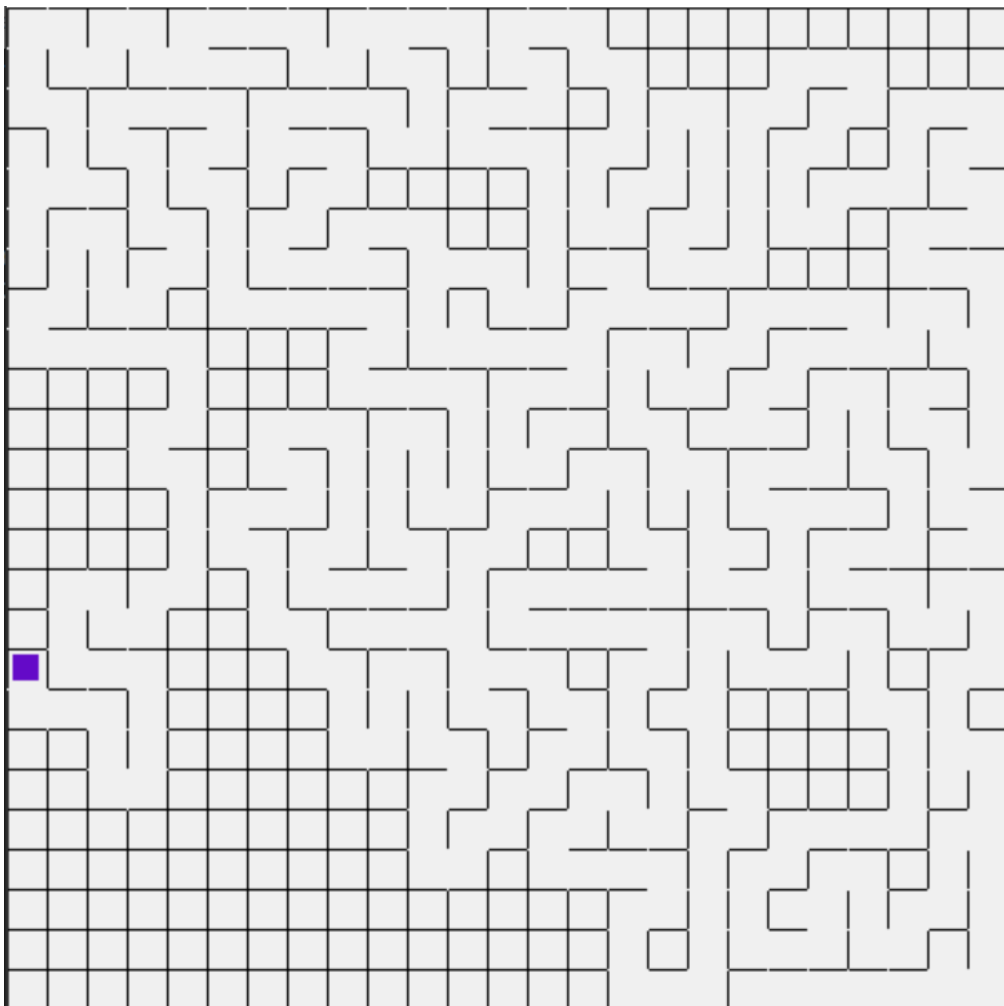


FIGURE 4.1 – Génération du labyrinthe à l'aide de l'algorithme DFS

## 4.2 Recherche du plus court chemin vers la premiere sortie aléatoire

Dans cette partie du programme, le chemin que la bille va traverser est shematisé par la couleur bleue comme si dessous, la bille traverse donc ce chemin depuis l'entrée choisie aléatoirement vers la premiere sortie aléatoire.

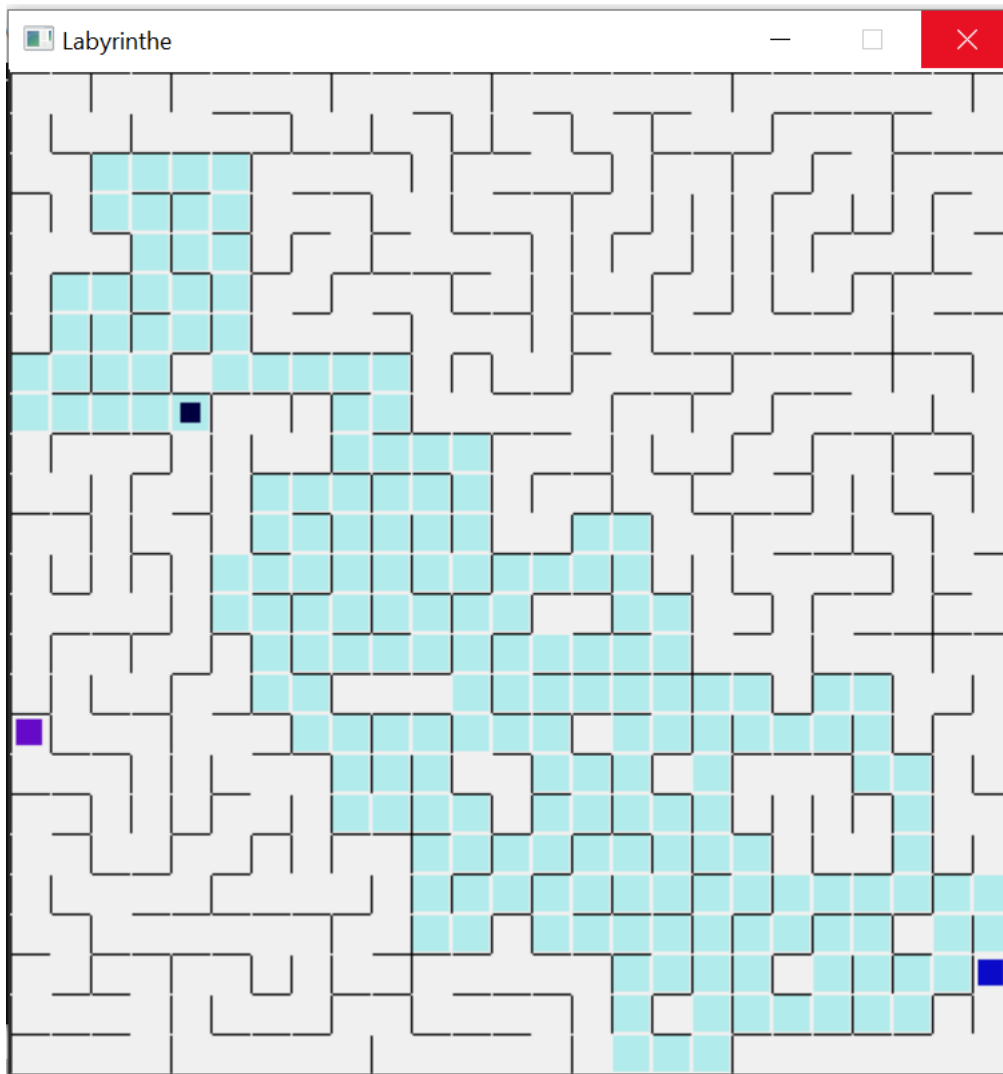


FIGURE 4.2 – Parcours du labyrinthe vers la premiere sortie en utilisant l'algorithme BFS

### 4.3 Recherche du plus court chemin vers la deuxième sortie aléatoire

Dans cette dernière partie du programme se fait la recherche du plus court chemin vers la deuxième sortie aléatoire schématisé en bleu, la bille rouge fait ce parcours depuis l'entrée vers cette sortie choisie aléatoirement.

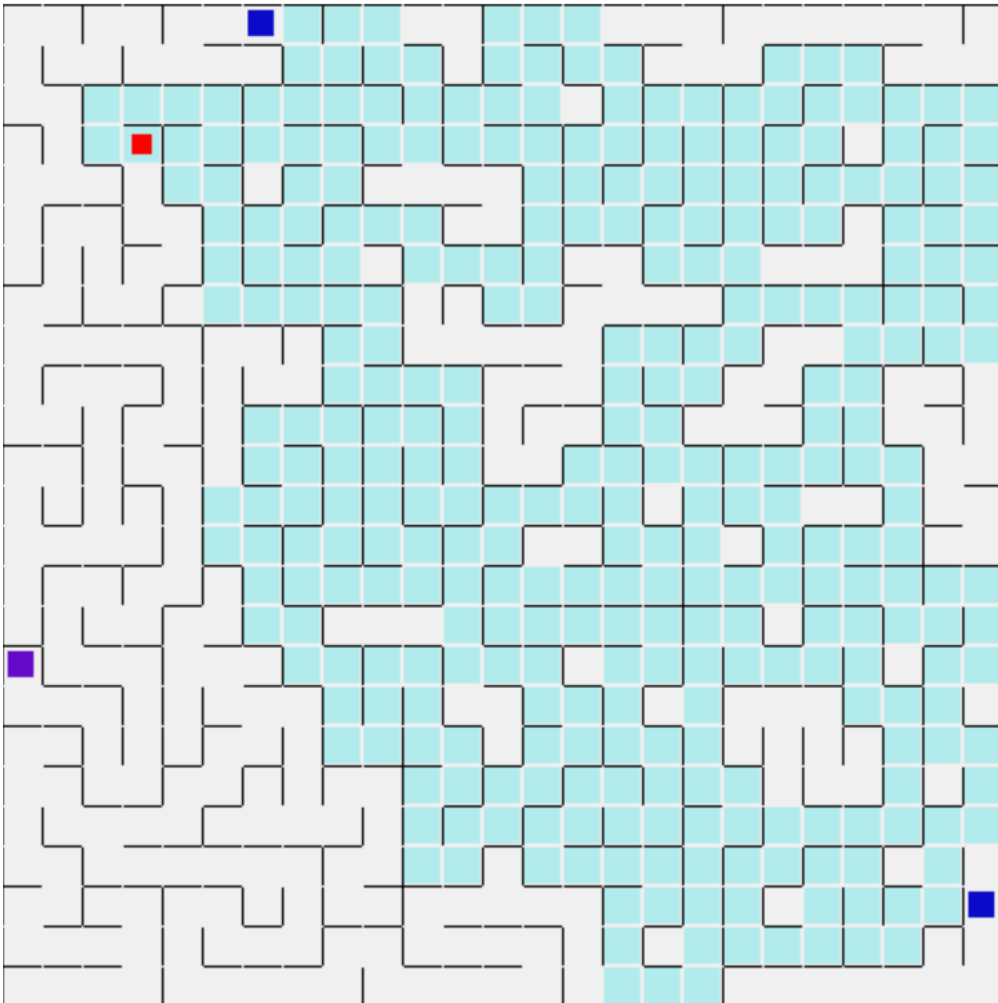


FIGURE 4.3 – Parcours du labyrinthe vers la deuxième sortie en utilisant l'algorithme BFS

# Bibliographie

- [1] <[https://en.wikipedia.org/wiki/Maze\\_generation\\_algorithm](https://en.wikipedia.org/wiki/Maze_generation_algorithm)>, 2020. [Online; accessed 9-January-2021].
- [2] <<https://thecodingtrain.com/CodingChallenges/010.1-maze-dfs-p5.html>>, 2020. [Online; accessed 9-January-2021].
- [3] <<https://www.overleaf.com/project>>, 2021. [Online; 25-January-2021 ].