Master 1 CRYPTIS
Algorithmique et Programmation

Rapport
Projet Labyrinthe

Fatma KOUIDER
Lucie MOUSSON
Moerava PIEDERRIERE
Morgane VOLLMER

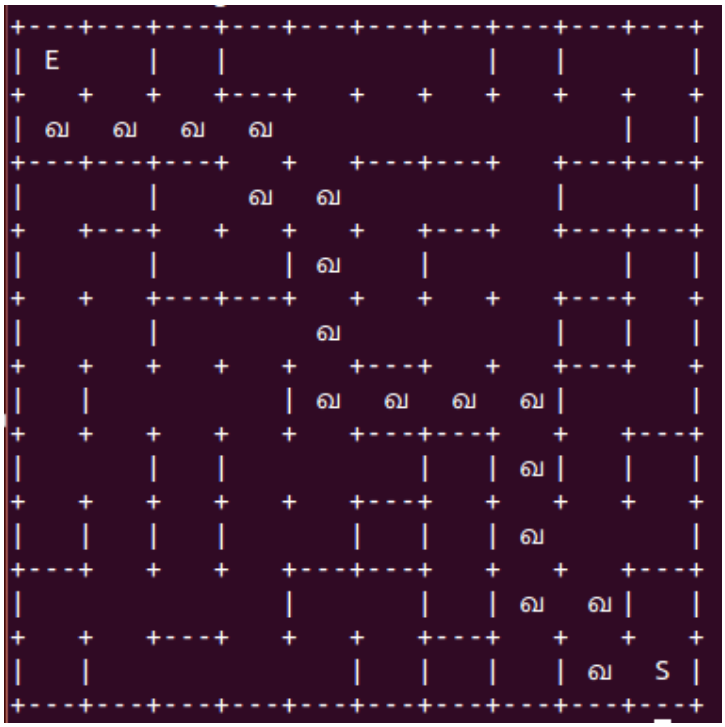


Table des matières

Introduction

Dans le cadre de l'UE programmation et algorithmique, nous avons réalisé le projet de modélisation d'un labyrinthe.
Pour cela, nous nous sommes séparé le travail de la manière suivante :

— Fatma s'est occupée des trois générations aléatoires.
La génération dite "intuitive", puis une autre où elle a imposé la place de l'entrée et de la sortie, pour finir une génération avec piles a été ajoutée.

— Lucie a géré le traitement de fichier, File.c, ainsi que la partie Validity.c. Puis elle a construit le main, et décidé de faire un Makefile. Pour finir, elle a fait le fichier.tex a partir de l'ensemble des parties de chacune qui lui ont été envoyées.

— Moerava a géré tout l'affichage, premièrement sans puis avec chemin.

— Morgane s'est occupée de toute la partie chemin ainsi que du calcul du temps d'éxécution.

Nous avons choisi suite à des divergences d'écritures, d'utiliser un git pour collaborer plus facilement.

Chapitre 1

Les structures

1.1 Nos choix techniques

Nous avons choisi de créer plusieurs structures. Il y a tout d'abord la structure Maze principale qui comporte les éléments principaux d'un labyrinthe. Mais pour plus de clarté et de facilité, nous avons décidé d'en utiliser d'autres.

1.2 Les différentes structures

1.2.1 La structure Maze

```
typedef struct Maze
{
unsigned short ** Matrix;
int Lin;
int Col;
int In[2];
int Out[2];
int Find[2];
} Maze;
```

La structure Maze est la structure principale de notre programme. Elle est utilisée par quasiment toutes les fonctions et est composée d'un tableau Matrix qui a comme variables les entiers affectés à chaque case du tableau, le nombre de ligne, qui peut être différent de celui des colonnes pour former un labyrinthe rectangle, et deux tableaux In et Out pour les coordonnées d'entrées et de sorties.

1.2.2 La structure Position

```
typedef struct Position {
   int X;
   int Y;
} Position;
```

Elle permet de regrouper les positions des cases dans une structure.
Cette structure nous a été utile dans le choix de l'entrée et la sortie dans MazeRand2.

1.2.3 La structure Node

```
typedef struct Node {
int Data[4];
struct Node* Next;
}Node;
```

Elle sert à stocker les cases qui n'ont pas été déjà visitées dans une pile (dans la fonction MazePile).
Elle contient :
1 - un tableau de 4 éléments dont nous allons utiliser pour mettre la position de la case courante et la position de la case précédente.
2 - un Pointeur "Node" qui sert à pointer sur un autre noeud (liaison entre les noeuds)
Cette structure est utilisée dans la fonction mazepile.

1.2.4 La structure Path

```
typedef struct
{
Position * Way;
int distance;
}Path;
```

Cette structure nous permet de tester le plus court chemin une fois celui-ci trouvé.
Il contient une distance correspondant à la longueur du chemin, et un tableau de position.
Le tableau de position contient distance +1 cases, nous considérons en effet que lorsque l'entrée et la sortie sont sur la même case, le chemin est de longueur 0.

Les différentes positions du chemin sont enregistrées dans le tableau, la première case correspondant à la sortie, et la dernière à l'entrée.

Chapitre 2

Les générations aléatoires

Dans cette partie nous allons expliquer les différentes méthodes que nous avons mises en pratique afin de générer un labyrinthe.

2.1 Génération aléatoire intuitive

Cette méthode est représentée par une fonction qui prend en entrée deux entiers, le nombre de lignes et le nombre de colonnes du labyrinthe. Nous avons procédé comme suit :

2.1.1 Première étape

Après la création d'une matrice "Matrix" nous allons la parcourir ligne par ligne tout en la remplissant par une valeur aléatoirement choisit entre 0 et 15 en distinguant les cas suivants :

Cas un

- Si Matrix[i][j] = Matrix[0][0] : Nous allons prendre une valeur aléatoire quelconque. - Si Matrix[i][j] = Matrix[0][1], Matrix[0][2], Matrix[0][3], ... , Matrix[0][Col-1] : Après un choix aléatoire de la valeur de Matrix[i][j] nous devons faire attention qu'il n'y soit pas d'anomalie entre les mures que partagent Matrix[i][j] et Matrix[i][j-1] ce qui revient à retester la valeur de Matrix[i][j] jusqu'à obtenir la même valeur du bit b0 de la valeur de Matrix[i][j] avec le bit b2 de la valeur Matrix[i][j-1].

Cas deux

- Si Matrix[i][j] = Matrix[1][0],... , Matrix[Lin-1][0] : Après un choix aléatoire de la valeur de Matrix[i][j] nous devons faire attention qu'il n'y soit pas d'anomalie entre les mures que partagent Matrix[i][0] et Matrix[i-1][0] ce qui revient à choisir une autre valeur aléatoire à Matrix[i][0] jusqu'à obtenir la même valeur du bit b3 de la valeur de Matrix[i][0] avec le bit b1 de la valeur Matrix[i-1][j].

Cas trois

- SiMatrix[i][j] = Matrix[1][1],Matrix[1][2],...,Matrix[1][Col-1], Matrix[2][1],Matrix[2][2],..., Matrix[2][Col-1],...,Matrix[Lin-1][1],Matrix[Lin-1][2],...,Matrix[Lin-1][Col-1] : Après un choix aléatoire de la valeur de Matrix[i][j] nous devons faire attention qu'il n'y soit pas d'anomalie entre les mures que partagent d'une part Matrix[i][j] avec Matrix[i-1][j] et d'autre part Matrix[i][j] avec

Matrix[i][j-1] ce qui revient à choisir une autre valeur aléatoire à Matrix[i][j] jusqu'à obtenir la même valeur du bit b3 de la valeur de Matrix[i][j] avec le bit b1 de la valeur Matrix[i-1][j] et la même valeur du bit b0 de la valeur de Matrix[i][j] avec le bit b2 de la valeur Matrix[i][j-1].

2.1.2 Deuxieme étape

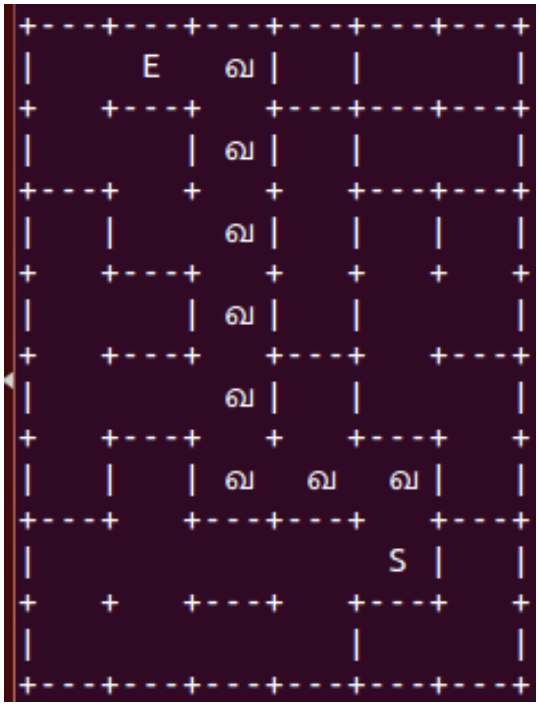
Maintenant que la matrice est bien remplie il faut retourner sur le contour (les bords) de la matrice et vérifier qu'il y a bien des murs sur les bords externes de la matrice si ce n'est pas le cas nous allons mettre les bits concernés à 1. -Cas : Matrix[0][0] : Mettre le b0 et le b3 à 1. -Cas : Matrix[0][1],...,Matrix[0][Col-2] : Mettre le b3 à 1. -Cas : Matrix[0][Col-1] : Mettre le b3 et le b2 à 1. -Cas : Matrix[1][0],Matrix[2][0],...,Matrix[Lin-2][0] : Mettre le b0 à 1. -Cas : Matrix[Lin-1][0] :Mettre le b0 et le b1 à 1. -Cas : Matrix[Lin-1][1],...,Matrix[Lin-1][Col-2] : Mettre le b1 à 1. -Cas : Matrix[Lin-1][Col-1] : Mettre le b1 et le b2 à 1. -Cas : Matrix[1][Col-1],...,Matrix[Lin-2][Col-1] : Mettre le b2 à 1.

2.1.3 Troisième étape

Nous allons à présent générer l'entrée et la sortie et pour ceci nous proposons deux choix :

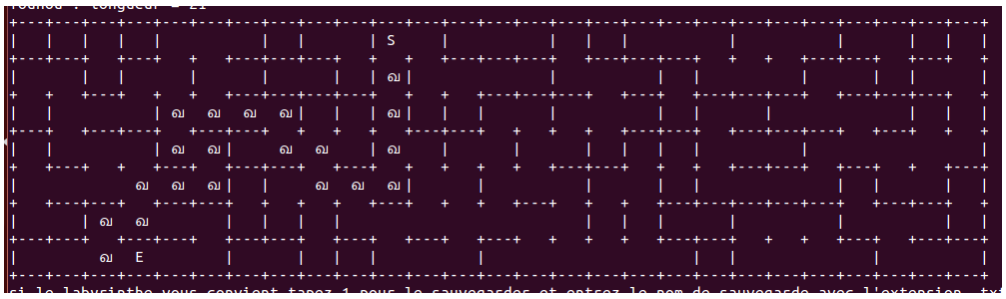
Choix un : Placement aléatoire avec MazeRand1

L'entrée et la sortie positionnées n'importe où dans la matrice (ne coïncident pas) : Il suffit de prendre une valeur aléatoire entre 0 et le nombre de ligne-1 pour In[0] et une valeur aléatoire entre 0 et le nombre de colonnes-1 pour In[1]. Pareil pour la position de la sortie sauf que nous vérifions si cette position ne coïncide pas avec l'entrée et nous l'attribuons à Out[2]. Ce choix est mis en pratique dans la fonction MazeRand1.
Exemple d'éxécution de MazeRand1 :



Choix deux : Placement sur les cotés avec MazeRand2

L'entrée et la sortie positionnées sur les bords de la matrice (ne coïncident pas) :
Dans un tableau de positions nous avons sauvegardé les positions des bords de la matrice, ensuite nous prenons aléatoirement une position du tableau que nous allons l'affecter à l'entrée du labyrinthe In[2]. En ce qui concerne la sortie elle est aussi choisie de la même manière sauf que nous avons ajouté un test pour que l'entrée et la sortie soient assez espacées et ne coïncident pas et nous nous l'attribuons à Out[2]. Ce choix est mis en pratique dans la fonction MazeRand2.
Exemple d'éxécution de MazeRand2 :



2.2 Génération pseudo-aléatoire

Dans le but de générer un labyrinthe dit parfait où toutes les cases sont accessibles où nous sommes sûres que le labyrinthe à au moins un chemin ce qui satisfait les utilisateurs. Pour mettre en pratique cette génération de labyrinthe nous nous somme inspirer de l'algorithme stratégie de recherche en profondeur.

Nous avons utilisés deux matrices de même taille : - Une matrice qui contient le labyrinthe - Une matrice binaire qui indique si une case du labyrinthe a été visitée où pas.

2.2.1 Première étape

Nous commençons par initialiser toutes les cases de la matrice du labyrinthe par 15 ce qui signifie que chaque case possède 4 murs et la matrice binaire par des 0 ce qui signifie que pour l'instant aucune case n'a été visitée.

2.2.2 Deuxième étape

Nous choisissons une position aléatoirement du labyrinthe qui va être le premier élément de la pile. Tant que la pile n'est pas vide nous allons récupérer le premier élément de la pile, ce dernier va être testé s'il a un parent donc nous allons distingué deux cas.

Cas 1

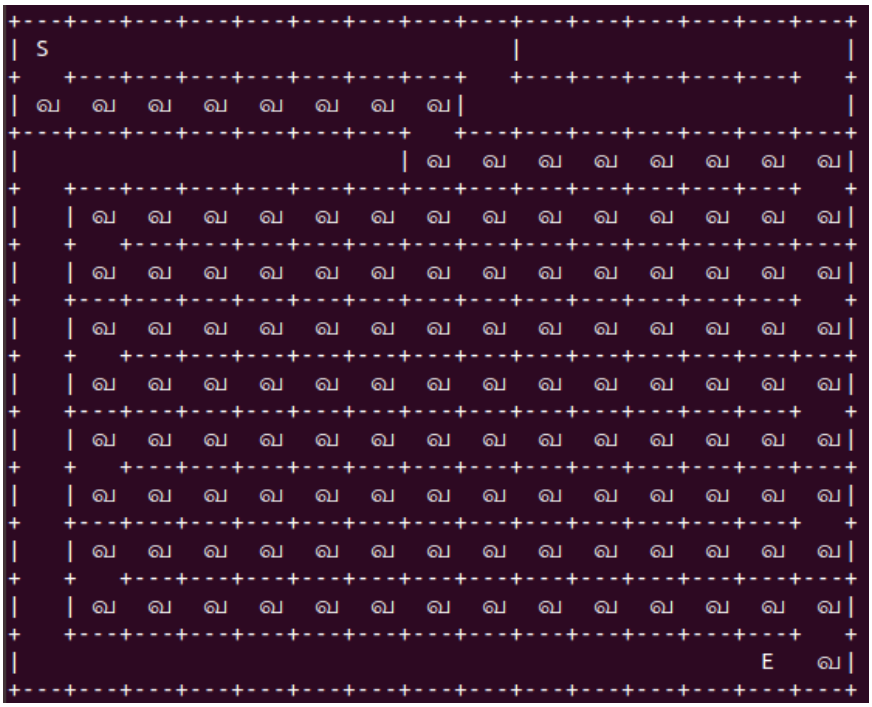
La case ne possède pas de parents (La première case avec qui nous allons commencer) : Nous allons ajouter les cases qui lui sont voisines (haut/bas/gauche/droite) dans la pile.

Cas 2

La case possède un parent : Nous allons éliminer les mures que partagent les deux cases ensuite nous vérifions si cette case (dépilé) possède des cases voisines accessibles qui n'ont pas été déjà visitées si c'est le cas nous ajoutons ces cases voisines (haut/bas/gauche/droite) à la pile. Ce processus va être répété jusqu'à épuisement des éléments de la pile (au fur et à mesure nous sommes en train de rajouter les voisins à la pile jusqu'à ce que les cases du labyrinthe seront toutes visitées).

2.2.3 Troisième étape

En ce qui concerne la génération de l'entrée et sortie nous avons opté pour la génération sur les bords du labyrinthe utilisé dans la fonction MazeRand2.
Exemple d'éxécution de MazePile :



2.2.4 Les autres fonctions utilisées dans MazePile

La fonction Push

Elle nous permet d'inserer un élément dans la pile .

La fonction pop

Elle nous permet de retirer le premier élément de la pile

La fonction Empty

Elle nous permet de vérifier si la pile est vide.

2.3 Les difficultés, erreurs et améliorations possibles

2.3.1 difficultés rencontrées

Les principales difficultés rencontrées ont été :
- La gestion des murs que partagent les cases.
- Créer un labyrinthe où il existe un chemin entre toutes les cases (cases toutes accessibles).

2.3.2 Améliorations possibles

On pourrait explorer d'autre méthode pour générer des labyrinthes parfaits.

2.3.3 Erreurs faites et limites

- L'utilisation de la fonction Rand avec plusieurs conditions prend beaucoup de temps à trouver la valeur adéquate ce qui alourdit notre fonction de génération aléatoire.

- La génération aléatoire est simple à implémenter mais les labyrinthes générés ne sont pas toujours parfaits et ils n'ont pas forcément un chemin, ce qui est désagréable pour un utilisateur.

- La génération d'un labyrinthe pseudo-aléatoire qui utilise la méthode de recherche en profondeur est assez simple à implémenter, par rapport à d'autres méthodes plus complexes.

Nous avons réussi à générer des labyrinthes où toutes les cases sont accessibles, donc il y aura toujours un chemin.
Par contre en ce qui concerne l'aspect visuel, cette méthode donne des labyrinthes simples ce qui peut être considéré comme un niveau trop facile par l'utilisateur.

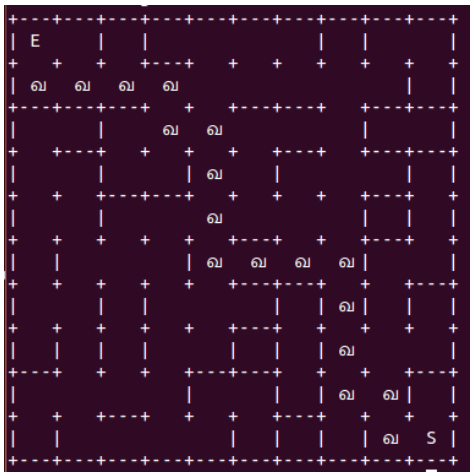
Chapitre 3

La manipulation de fichiers

Nous avons choisi de placer les fonctions traitant les fichiers dans un fichier File.c, qui comporte donc les fonctions lectFic et saveMaze expliquées ci dessous.

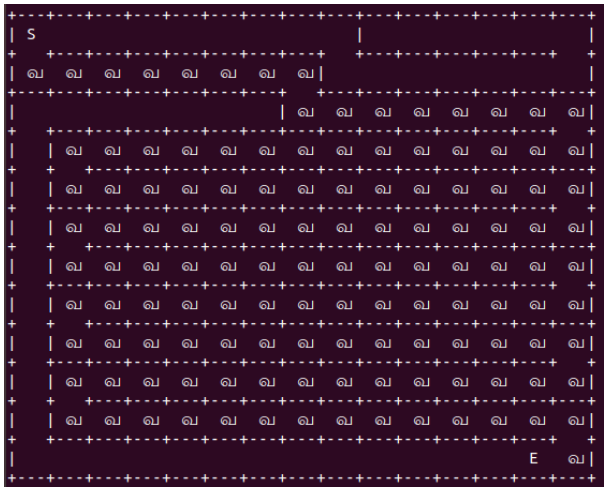
3.1 Lecture dans un fichier

Pour la fonction génération d'un labyrinthe à partir d'un fichier, qui prend en paramètre le nom du fichier, nous testons l'ouverture du fichier donné en paramètre. L'utilisateur a le choix entre un de ses fichiers, et un fichier par défaut (celui de l'énoncé). Si le fichier donné ne s'ouvre pas correctement (valeur de retour égale à NULL), nous affichons l'erreur et renvoyons un labyrinthe dont le nombre de ligne est égale à zéro. De sorte que dans le main, un seul test suffise à déterminer s'il y a un problème d'ouverture. Si la valeur retournée est différente de NULL, nous scanons les valeurs de la première ligne puis, après avoir alloué dynamiquement un tableau à deux dimensions, nous faisons une double boucle, à partir des valeurs ligne et colonne récupérées précédemment. Calloc est préféré à Malloc pour l'allocation, car il initialise l'ensemble à zéro. Il ne reste plus qu'à fermer le fichier, puis à retourner la structure Maze, remplie par les valeurs du fichiers.
Voici le labyrinthe généré avec le fichier par défaut de l'énoncé dont le chemin le plus court a été tracé à l'aide de la fonction waysearch vue ci après.



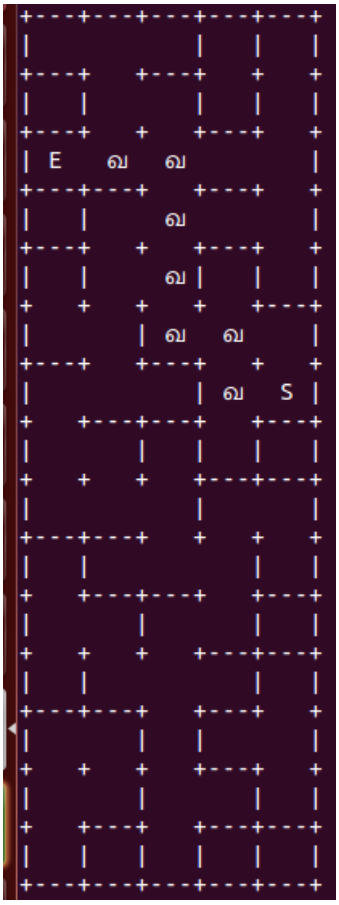
3.2 Sauvegarde dans un fichier

Nous avons choisi également de créer une fonction saveMaze, qui prend en paramètre un nom de fichier (avec extension .txt) et la structure du labyrinthe venant d'être généré. Que celui ci soit aléatoire, pseudo aléatoire ou formé avec les Piles, cette fonction permet à l'utilisateur d'enregistrer s'il le souhaite le labyrinthe venant d'être généré, ainsi que le plus court chemin trouvé. De cette façon, l'utilisateur peut sauvegarder les résultats qui l'intéressent. Cette fonction est programmé comme suit : elle ouvre un nouveau fichier dont le nom est donné en paramètre, le nom a été demandé à l'utilisateur. Elle teste s'il s'ouvre correctement. Si la valeur de retour d'ouverture est différente de NULL, elle écrit les différentes caractéristiques du labyrinthe dans le fichier sous le format donné dans l'énoncé. Puis elle ferme le fichier courant. Cette fonction ne retourne rien. Notons que parmi les améliorations possibles, nous aurions pu sauvegarder le fichier sans le chemin le plus court, en plus de sauvegarder celui où il est déjà dessiné.
Il n'y a pas eu de grosses difficultés de rencontrées durant le traitement des fichiers.
Sauvegarde d'un labyrinthe MazePile dont le chemin a déjà été tracé.




```
12 15 11 13 0 0
2089 65530 65530 65530 65530 65530 65530 65530 65532 65531 65530 65530 65530 65530 65532
2067 2058 2042 2026 2010 1994 1978 1964 65523 65530 65530 65530 65530 65530 65526
361 378 394 410 426 442 462 1939 1930 1914 1898 1882 1866 1850 1836
341 1609 1626 1642 1658 1674 1690 1706 1722 1738 1754 1770 1786 1802 1814
325 1587 1578 1562 1546 1530 1514 1498 1482 1466 1450 1434 1418 1402 1388
309 1161 1178 1194 1210 1226 1242 1258 1274 1290 1306 1322 1338 1354 1366
293 1139 1130 1114 1098 1082 1066 1050 1034 1018 1002 986 970 954 940
277 713 730 746 762 778 794 810 826 842 858 874 890 906 918
261 691 682 666 650 634 618 602 586 570 554 538 522 506 492
245 265 282 298 314 330 346 362 378 394 410 426 442 458 470
229 243 234 218 202 186 170 154 138 122 106 90 74 58 44
211 202 186 170 154 138 122 106 90 74 58 42 26 10 22
```

Puis la sauvegarde d'un fichier MazeRand2 avec, toujours, le chemin.



Labyrinthe.txt

```
15 5 2 0 6 4
75 56 78 65533 109
65535 33 60 65527 85
11 18 32 58 68
65535 73 48 74 84
141 81 68 125 103
115 100 83 96 124
137 114 142 113 134
145 172 221 135 65535
163 178 196 249 268
349 235 210 228 279
321 316 265 246 65535
343 291 272 302 65533
65529 65532 293 65531 65524
65521 65526 305 334 65527
65527 65535 327 65535 65535
```

Chapitre 4

Vérification de la cohérence du labyrinthe

Une des fonctions indispensable était bien sûr la fonction testant si les valeurs du labyrinthe étaient cohérentes. Qu'il soit généré par un fichier ou aléatoirement. Même si le cas d'erreur le plus probable reste celui du fichier, les fonctions aléatoires étant écrite de sorte à éviter ce type d'erreur, nous testons dans le main tous les labyrinthes générés par cette fonction. La fonction MazeValid s'appuie sur la fonction itob, qui prend un entier et le décompose en binaire dans un tableau qu'elle retourne. Elle fonctionne avec l'utilisation de divisions euclidienne et donc du modulo. MazeValid prend donc en paramètre une structure labyrinthe. Elle teste si les coordonnées de l'entrée et de la sortie sont bien dans le labyrinthe, si l'entrée est positionnée différemment de la sortie, et retourne le type d'erreur commise. Ensuite, Elle teste si les murs correspondent : Si deux cases adjacentes ont bien des murs correspondants, ou le cas échéant, aucun mur. De la sorte, On ne peut pas avoir de mur "simple" entre deux cases. De même elle retourne d'où vient l'erreur s'il y en a une. Après avoir libéré la mémoire allouée avec free, la fonction retourne 1 si le labyrinthe est valide, 0 sinon. Un exemple d'erreur par lecture sur un fichier dont les valeurs de deux cases adjacentes ne correspondent pas.

```
donnez le nom de votre fichier avec l'extension .txt, sinon tapez non et ce sera un fichier par défaut
test.txt
droite gauche 1 0 0 3
labyrinthe non valide
```

Ici, c'est le mur de droite de la case ligne 1 colonne 0 qui ne correspond pas au mur de gauche de la case ligne 0 colonne 3.

Chapitre 5

Chemin

5.1 Recherche de chemin

Pour trouver un chemin entre l'entrée et la sortie, et par la suite le chemin le chemin le plus court, nous avons dû trouver un moyen de marquer les cases que nous avons testées et par lesquelles nous sommes passées. Pour cela, nous avons décidé d'utiliser les douze derniers bits disponibles et d'initialiser toutes les cases à 4095.
Cette distance correspond à la distance maximale pouvant être implémentée sur 12 bits (soit 2^12 - 1). Il est donc possible que, sur des labyrinthes trop grands (possédant plus de 4095 cases), notre algorithme annonce qu'il n'a pas trouvé de chemin, alors qu'un chemin existe.
Dans le cas où un chemin de longueur exactement 4095 existe, la case de sortie est donc implémentée d'une distance 4095 et nous testons si une des cellules voisines de la sortie comporte une distance 4094. Si ce n'est pas le cas, les structures données et l'espace utilisé pour conserver les distances ne nous permettent pas de faire la différence entre la non existence de chemin et le cas où le chemin existe mais passe par plus de 4095 cases.

Pour initialiser nos cellules à la distance maximale, nous utilisons la fonction « set_distance_max() » prenant en argument un labyrinthe et affectant à toutes les case la distance maximale, exceptée la case d'entrée initialisée à distance 0. Cette fonction utilise une autre fonction nous permettant d'affecter une distance quelconque à une case définie (fonction « set_distance »).

Nous commençons désormais la recherche de chemin. Pour cela, nous avons construit une fonction récursive « ft_mapping_maze » prenant en argument un labyrinthe et une distance. Cette fonction est initialement utilisée avec la distance 0, nous permettant de construire une carte de chemin suivi. Dans un premier temps, nous initialisons la position du curseur à la position de l'entrée, et regardons si la position du curseur est la même que celle de la sortie, si c'est le cas nous affichons la distance du chemin entrée (0 en début d'algorithme), sinon nous entrons dans une boucle for. Pour chaque direction pouvant être empruntée (1 correspondant à gauche, 2 correspondant à en bas, 4 correspondant à droit et 8 correspondant à en haut), nous regardons si nous pouvons accéder à la case la plus proche dans cette direction (soit s'il n'y a pas de mur nous séparant de cette case), et nous regardons également si cette case à déjà été atteinte par un chemin plus court, cela est fait grâce à la fonction « ft_can_go_there ». Dans le cas où nous pouvons aller dans cette direction, nous bougeons le curseur dans cette case, incrémentons la distance d'un, ce qui signifie que nous avons un chemin construit depuis l'entrée jusqu'à cette case de cette distance. Nous appliquons une nouvelle fois notre fonction à notre labyrinthe, avec le curseur déplacé et la distance implémentée

d'un. Dans le cas où nous tombions sur une case qui ne peux accéder qu'à des cases ayant déjà été atteintes par un chemin plus court, nous faisons marche-arrière en décalant notre curseur dans la position inverse et en décrémentant la distance de notre case d'un, puis appliquons de nouveau notre fonction pour emprunter un nouveau chemin.
Notre algorithme cartographie donc notre labyrinthe de différentes façons, jusqu'à trouver le chemin le plus optimal. Celui-ci est inscrit dans les cases de notre labyrinthe sous forme de distance associée à chaque case. Nous avons donc décidé de récapituler ces informations dans une structure Path pour rendre son affichage plus simple.
Pour construire cette structure (comportant une distance et un tableau de position), nous nous y sommes pris de la manière suivante : la longueur du chemin est retrouvée en prenant la distance inscrite dans la case de sortie. Nous construisons ensuite un tableau de position contenant longueur+1 cases, et dont la première case contient la position de la sortie. Nous appliquons ensuite à chacune des cases la fonction pos_adjacent_cell permettant de trouver la case adjacente de notre position dont la distance et égale à la distance de notre case -1. De proche en proche, nous remplissons notre tableau jusqu'à remplir la dernière case par la position de notre entrée.

Chapitre 6

Affichage

Dans cette partie, nous allons parler de la partie affichage appelé Display dans notre programme. Nous avons programmé cette partie en deux fois. La première était d'afficher le labyrinthe avec des murs et la deuxième était d'afficher le labyrinthe avec le plus court chemin ainsi que l'entrée et la sortie.

6.1 Affichage simple, sans chemin

6.1.1 Nos choix techniques

Les murs du labyrinthe sont codés par les 4 bits de point faible le b3 code le mur du haut, le b2 le mur de droite, le b1 le mur du bas et le b0 le mur de gauche. Nous sommes passées par la division et le modulo.
En effet, le bit b0 correspond à 1, b1 à 2, b2 à 4 et b3 à 8 et en faisant la divisions par 2, 4, ou 8 nous obtenons 1 ou 0 car nous travaillons avec des entiers, donc ici la division nous donne la partie entière seulement. C'est ici que va intervenir le modulo 2 car si le nombre modulo 2 donnait 1 cela voulait dire qu'il y avait un mur et si ça donnait 0 c'est qu'il n'y en avait pas.
Ainsi, nous avons pu déterminer des conditions pour la construction des murs.

Pour construire le labyrinthe, nous avons fait le choix de lire deux fois la même case. La première fois étant pour afficher les murs de droite et de gauche et la deuxième fois pour afficher les murs du bas. Nous n'avions pas à afficher les murs du haut, sauf le tout premier, car nous avons une fonction qui teste la validité de notre labyrinthe donc si il y avait un mur du bas pour la case du dessus c'est qu'il y avait un mur en haut pour la case du dessous.

De là, nous avons construit deux boucles successives de for, la première pour parcourir les lignes et la deuxième pour parcourir les colonnes. Pour afficher les murs, nous avons fait une succession de if avec les conditions que nous vous avons expliquées plus tôt.

Les difficultés que nous avons rencontrées sont les doubles murs, le fait que les intersections n'étaient pas marquées avec notre première façon d'afficher le labyrinthe, ou encore les décalages au niveau des espaces.

Pour régler notre problème des doubles murs, nous avons juste rajouté une condition supplémentaire, prenant en compte la configuration de la case précédentes et les murs qu'il y avait déjà

d'affichés.

Nous avions aussi un problème d'espace insuffisant ou trop conséquent, ce qui décalait tous les murs et les intersections. Pour pouvoir le régler, nous avons fait des conditions en fonction de la configuration de la case précédentes.

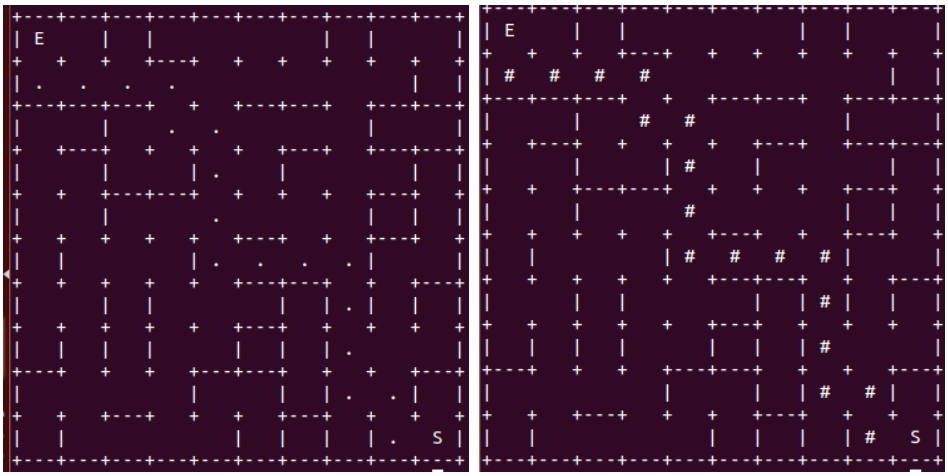
Pour notre problème d'intersections, nous avons dû changer notre façon de représenter les murs du bas pour pouvoir afficher les intersections de cases quand nous avions une succession de cases ne possédant pas de murs.

Nous sommes passées de « _ » a « +—+ ». Dans cette nouvelle façon d'afficher, nous avons rencontré un autre problème, celui d'afficher une double intersection. Ce qui décalait tout le labyrinthe et donc les murs ne correspondait plus aux bonnes intersections. Nous l'avons réglé de la même manière que les doubles murs.

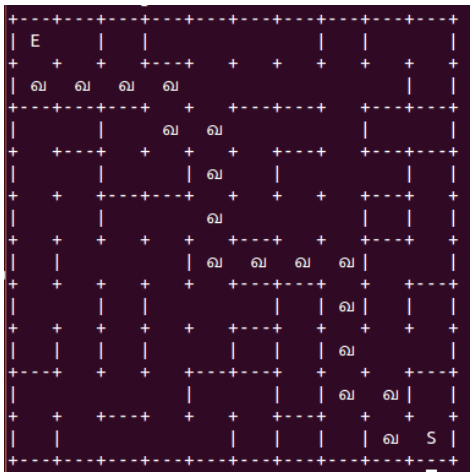
6.2 Affichage avec chemin

Nous avons travaillé sur la fonction de base afin d'y ajouter l'affichage de l'entrée, de la sortie et du chemin.
Dans un premier temps, nous avons juste ajouté des conditions supplémentaires pour afficher le chemin, l'entrée, la sortie ou un vide.
Mais cela est vite devenu illisible et les conditions commençaient à se confondre.
Donc, nous avons choisi de créer une fonction qui s'appelle way qui teste :
si le chemin passe dans cette case elle renvoit 1,
si l'entrée est dans cette case elle renvoit 2,
si la sortie est dans cette case elle renvoit 3,
sinon elle renvoit 0.
De cette fonction, on a fait un switch dans l'affichage, ce qui allège la fonction et améliore la lisibilité.

Pour afficher le chemin, nous avons testé plusieurs choix de représentation comme suit :



Nous avons choisit la troisième option (image suivante) car la première n'était pas assez visible, la deuxième ne nous paraissait pas très jolie et puis la troisième nous a paru assez ludique et visible pour nos utilisateurs.



Chapitre 7

Main et Makefile

7.1 Main

7.1.1 Choix techniques

Nous avons choisi d'utiliser un switch pour alléger le programme déjà conséquent du main.c. De sorte, une seule valeur est testée, et cela évite de trop nombreux tests secondaires. Nous déclarons les variables locales à la fonction au début. Parmi celles ci la déclaration d'une chaîne de caractères est nécessaire pour allouer le nom du fichier de sauvegarde, et l'initialisation du srand pour la génération aléatoire.
Une boucle while est programmée pour que l'utilisateur n'ait pas à relancer le programme à chaque fois qu'il veut générer un labyrinthe. De la sorte, il aura juste à taper 0 pour arrêter le programme, ou s'il veut continuer à générer, choisir parmi les quatre générations possibles. Au bout de quelques générations, on atteint une erreur mémoire.
La désallocation de la mémoire allouée avec les free donnait des erreurs de segmentation. Ils sont donc laissés en commentaires dans le main.

7.1.2 Les différents cas traités

Case 1 : D'après un fichier donné ou par défaut

Le fichier par défaut est celui de l'énoncé du projet qui a été joint au mot ENTREE avec un #define de sorte que l'on puisse changer ce fichier plus facilement en ne le modifiant que dans le Maze.h. Après avoir testé si le fichier s'ouvrait, (s'il ne retournait pas une structure dont le nombre de ligne égale zéro) nous vérifions qu'il est valide avec MazeValid, puis on génère le chemin en appelant way_search. Il ne reste plus qu'à afficher le labyrinthe avec dis.

Case 2 : Génération totalement aléatoire du labyrinthe

Après avoir confirmé à l'utilisateur que le labyrinthe allait être généré aléatoirement, on lui demande s'il veut imposer la taille ou s'il préfère que ce soit totalement aléatoire. On le prévient ici que s'il choisit m, soit le nombre de colonnes, supérieur à 35 l'affichage risque d'être faussé car la largeur de l'écran sera atteinte. La taille aléatoire sera modulo 34 + 1 car au delà il y aura une incohérence d'affichage en raison de la largeur du terminal.Nous appelons la fonction chemin, puis nous lui proposons de sauvegarder le labyrinthe avec saveMaze.
Nous procédons de même pour le case 3 (génération pseudo aléatoire) et le case 4 (génération avec

Piles pour atteindre toutes les cases depuis l'entrée).
Notons qu'au bout de plusieurs générations successives il y a des erreurs de différents types qui apparaissent. Certainement dues à la mémoire.

7.2 Makefile

7.2.1 Choix techniques

Le principe du Makefile est de découper son code en plusieurs morceaux distincts de façon à le rendre plus compréhensible. C'est également plus facile de travailler en groupe de la sorte. Au vu de l'envergure du projet, et car nous étions au nombre de quatre, il était impératif d'en faire un. De sorte, il y a au total un main.c, qui est le main général. Un fichier .h qui est inclus dans tous les autres fichiers .c. Et sept fichiers.c distincts, validite.c, fichiers.c, affichage.c, mazerand1.c, mazerand2.c, mazepile.c et waysearch.c. Tous ces fichiers sont donc liés par le Makefile ce qui permet de distinguer leur utilité et de les modifier plus facilement au besoin.

7.2.2 En pratique

Nous avons décidé en faisant le Makefile, de ne pas alourdir l'écriture avec les différentes commandes possibles. Nous avons simplement rajouté un "clean" pour supprimer les fichiers .o générés lors de la compilation :

```
clean:
rm *.o
```

Il a été quelque peu compliqué de déterminer s'il y avait dépendance ou non entre les fichiers, et comment traiter le cas du .h. Cependant, après quelques recherches sur internet, étayée par le polycopié distribué en début d'année, cela s'est vite réglé.
En conclusion, il suffit d'écrire la commande make dans le terminal, puis la commande ./lab pour exécuter le projet.

Conclusion

Nous avons donc implémenté toutes les fonctionnalités possibles d'un labyrinthe avec plus court chemin auxquelles nous avons pensées.
Un ajout possible aurait pu être une fonction RatioReussite() qui renvoyait le pourcentage de labyrinthe avec plus court chemin selon la taille et la génération, ainsi nous aurions pu optimiser encore la création de labyrinthe.
Cependant, notre code étant déjà conséquent, nous ne l'avons pas ajoutée.
Il va de soit qu'étant quatre sur le projet, malgré l'utilisation d'un git, il y a certainement eu des "redondances", des morceaux de codes qui se croisent. C'est une des optimisations possibles.