Projet :
Génération aléatoire de Labyrinthe.

Introduction

Le but du projet est d'implémenter un programme qui génère des labyrinthes d'une façon aléatoire.
Ici on implémentera spécifiquement l'algorithme de Union-Find.

Développement

I. Structures

```
typedef struct unionfind_t {
        coordonnnes_t taille;
        coordonnnes_t  **pere;
        unsigned int **rang;
} unionfind_t;

typedef struct labyrinthe_t  {
        coordonnees_t taille;
        int ** murs_hori;
        int ** murs_vert;
}labyrinthe_t;
```

Nous avons choisi la première structure Séparer Cases et Murs, car nous avons eu l'idée de représenter les cases et les murs séparément, pourquoi notre choix était déjà fait dés le début.

II. Fonctions principales.

a) Fonction d'implémentation du labyrinthe.

int initialiser_labyrinthe(labyrinthe_t* laby, int x, int y) :
Cette fonction initialise les murs du labyrinthe et alloue de la mémoire pour les murs **horizontaux** et **verticaux**, l'entier 1 si le murs est présent, donc au début on initialise tout à 1, sauf la première case (en haut à gauche) et la dernière (en bas à droite), où x et y représente le couple de cordonnées des murs.
Renvoi 1 si tout s'est bien passer, 0 sinon.

int initialiser_case(unionfind_t *grille ,int x ,int y)
Pareillement, même principe que la fonction dessus, on alloue et on déclare le **père** et le **rang** de chaque case. Initialement le **rang** vaut 0, et chaque case est **père** de lui même, où x et y représente le couple de cordonnées des cases.
Renvoi 1 si tout s'est bien passer, 0 sinon.

coordonnees_t trouver(unionfind_t *grille , coordonnees_t casee) ;
Fonction qui trouve le représentent d'un ensemble, en faisant une compression de chemin. Elle retrouve et renvoi le représentent de l'ensemble qui contient la case **casee** passer en paramètre, nous avons eu l'inspiration de la fonction du cours.

void casser_murs_aleatoirememnt(labyrinthe_t *laby , unionfind_t *grille) ;
Cette fonction se charge de générer aléatoirement des couples de coordonnées qui, ensuite, vérifie si le mur vaut 1, dans ce cas il casse le mur, soit le mur **horizontal** ou **vertical**, encore une fois ce choix est fait alétoirement par une variable **int h_ou_v** qui vaut 0 ou 1, l'entier 0 pour casser le mur **vertical** et 1 pour **horizontal**.
Une fois le mur casser on fusionne (appelle de la fonction **Fusion par rang**)

void fusion_par_rang(unionfind_t *grille , coordonnees_t ensemble1 , coordonnees_t ensemble2) ;

Cette fonction fusionne entre deux cases voisines en utilisant la fonction trouver et en comparent les rang de chaque cases. En appliquant la technique **de compression des chemins**, les opérations peuvent être optimisées en **O(Log n)** dans le pire des cas. L'idée est de toujours attacher un arbre de plus petite profondeur sous la racine de l'arbre plus profond, ce qui explique le nom de notre fonction car cette technique se nomme **l'union par rang**. Les deux techniques se complètent. La complexité temporelle de chaque opération devient encore plus petite que **O(Log n).** En fait, la complexité du temps amorti devient effectivement petite constante.

int est_chemin_valide(unionfind_t grille) ;

Cette fonction se charge de tester s'il existe un chemin valide, or il vérifie si le représentant de la première case est dans même ensemble que le représentant de la dernière case.
Renvoi 1 si le chemin est valide et 0 sinon.

b) Fonction d'affichage.

void affichage_case(unionfind_t grille) ;
Cette fonction n'est pas utilisé dans le code que nous avons rendu, elle se charge d'afficher les pére de chaque case sous forme de couple de cordonnées, si nous souhaitons savoir qui est le père de qui, appeler **affichage_case() ;**

void affichage_ascii(labyrinthe_t laby) ;
Cette fonction est le premier affichage (version ascii).

void affichage_ascii_utf8(labyrinthe_t laby) ;
Cette fonction affiche le labyrinthe version amélioré (chaine de caractère de UTF8).

void affichage_graphique(labyrinthe_t laby, int hauteur , int largeur) ;
Fonction qui permet d'utiliser la bibliothèque MLV ou hauteur et largeur, représente la hauteur et la largeur du labyrinthe et non pas la taille de la fenêtre, la taille de la fenêtre est définie par une macro.
#define TAILLE_FENETRE_X 1000
#define TAILLE_FENETRE_Y 1000

int main(int argc ,char *argv[]) ;

Fonction du main, le parti la plus délicat du projet, au départ nous avons eu des difficulté pour récupérer les arguments, ensuite, nous avons penser à consulter le manuel de la page **strcmp()** et que voilà qu'on tombe sur **strncmp() ,** cette fonction nous a facilité la tâche car on pouvais comparé l'argument saisie par l'utilisateur, pour ensuite décomposer et récupéré la valeur de ce que l'utilisateur a saisie. C'était pas évident de procéder d'une autre manière.

Améliorations :

Une fois les arguments récupérés on traite les options qui sont des améliorations :
Nous avons réussis à faire toutes les améliorations demander sauf la 7éme (chemin victorieux), l'idée est présente, sauf qu'on ignore comment faire l'algorithme qui permet d'utiliser nos deux fonction **en_filer()** et **de_filer()**, Nous n'avons pas mentionné avant ni la structures **File**, ni les deux dernière fonctions. Effectivement, nous avons déclaré une structure basée sur le principe *premier entré, premier sorti :*

```
typedef struct element {
        coordonnees_t coordonnees;
        struct element *suivant;
}Element;

typedef struct file{
        Element *debut;
        Element *fin;
        int taille;
}File;
```

Mais, cela n'a pas pu éclairé la suite de comment on fait l'algorithme.

Manuel d'utilisation

Si vous voulez utiliser votre fichier voici les étapes à suivre :

> Ouvrir le Terminal
> Se placer dans le dossier du projet
> Entrer la commande "**gcc labyrinthe.c –ansi –Wall –pedantic –o laby -lMLV** "

Ensuite, pour exécuter :
> Entrer la commande "**./laby [OPTIONS]**"

Où [OPTIONS] représente les différents options listé ci-dessous avec chacune leur fonction.

- ./**laby –mode=texte :**

  Cette option oblige le programme a procéder a l'affichage ascii amélioré, en l'absence de cette option, l'affichage graphique est choisi par défaut.

- ./**laby –taille=HxL:**

  Cette option permet de modifier la taille du labyrinthe qui initialement est par défaut à 6x8, où **H** et **L** sont des entiers.

- ./**laby –graine=X:**

  Cette option permet modifier srand() qui, par default est srand(time(NULL)) ; Ainsi, deux appels a votre programme avec une même graine devront créer le même labyrinthe, où **X** est un entier.

- ./**laby –attente=X:**

  Cette option en mode texte utilise **usleep(X**), cela permet de voir l'évolution du labyrinthe pour trouver un chemin valide, nous avons utilisé **MLV_wait_milliseconds(X)** pour la version graphique.
  X est le nombre de milliseconde d'attente entre chaque suppression de mur, par défault c'est fixé à 0.

- ./**laby –unique :**

  Cette option empêche la suppression d'un mur s'il sépare deux cellules appartenant déjà a la même classe, en utilisant la fonction trouver, on regarde avant de casser un mur entre deux cases voisines, si ils ont le même représentent sinon on casse le murs.

- ./**laby –access :**

  Cette option fait que toutes les cases appartiennes a la même classe en comptent le nombre du fusion. On s'arrête de casser les murs lorsque le nombre de fusion = (hauteur * largeur) -1 , ce qui permet de rendre tout les ensemble accessible.

Remarque importantes :

- Attention : Certaines options ne s'utilisent pas sur la même commande.
- L'écriture des options lors de l'exécution doit respecter la taille de l'option c'est à dire, il faut pas qu'il y est d'espace en plus : exemple, --graine = 12 fonctionnera pas, par contre --graine=12

Conclusion

Ce projet nous a aidés à comprendre nos cotés forts et cotés faibles et aussi la maîtrise des cours du 1er semestre (allocation, structures, pointures, tableaux arguments main, lib MLV…..) et le cours Find-Union en adaptant les fonctions du cour avec nos structures. Nous réalisons également que répartir le travail en binôme est plus efficaces.