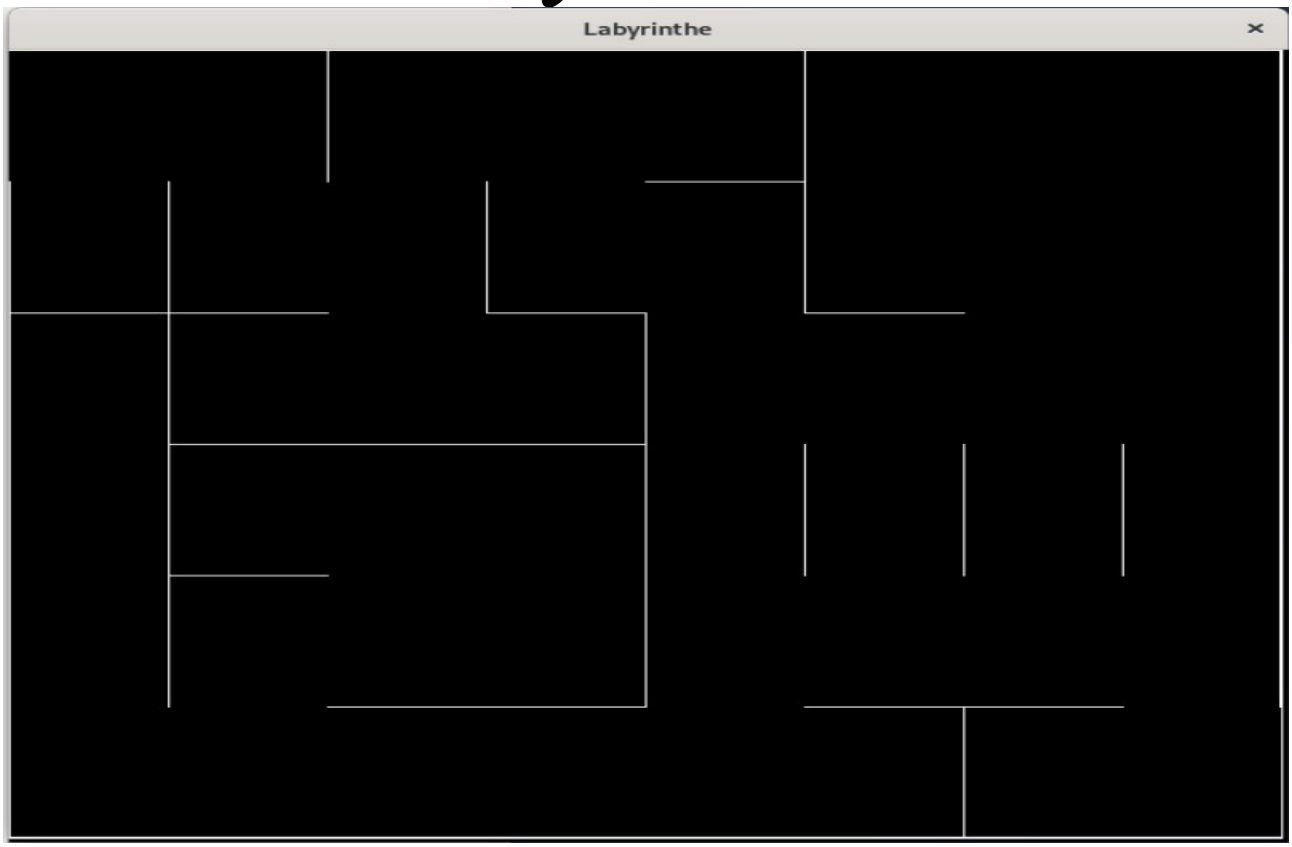
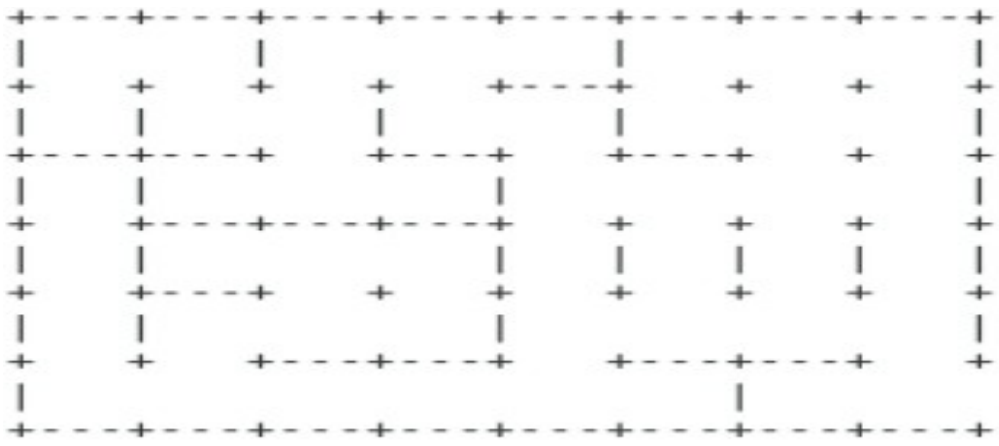
Génération aléatoire de labyrinthe





Canet Dylan – TP 3 – L2 Informatique – Université Gustave Eiffel
Diagne Ben – TP 3 – L2 Informatique – Université Gustave Eiffel

Le projet consiste en créer un générateur aléatoire de labyrinthe en implémentant l'idée d'arbres de recherche que nous avions été enseigné dans ce module.

I – Partie Utilisateur

Compilation :
gcc -Wall -ansi Labyrinthe.c -o Labyrinthe

Éxécution :
./Labyrinthe (--option)

Liste des options :
A la suite de l'éxécution, l'utilisateur a le choix d'utiliser les options ci-dessous :

- --taille=mxn : Définit la taille du labyrinthe généré, m étant la longueur et n la largeur.
- --graine=X : Fixe la graine du labyrinthe, ce qui permet de sélectionner un labyrinthe généré en particulier. X étant l'identifiant du labyrinthe.
- --attente=X : Ajoute un délai de X seconde(s) à chaque suppression de mur, ce qui permet de voir comment est généré le labyrinthe plus en détails.
- --mode=texte : Affiche une version textuelle du labyrinthe dans le terminal.
- --unique : Permet d'avoir un labyrinthe à chemin unique.

II – Partie Développeur

On cherche à écrire un programme qui permet de créer des labyrinthes aléatoires. Le concept sera de créer d'abord un labyrinthe plein avec tout les murs actifs, puis supprimer aléatoirement les murs jusqu'à tomber sur un labyrinthe valide.

Initialisation

Pour la composition du labyrinthe, deux structures sont à notre disposition :

-**Cellule**, qui correspond à une case du labyrinthe. Elle est composé de x et y, qui sont les coordonnées de la case. Elle va être utilisé pour établir un tableau dynamique de cette structure qui représentra l'arbre des cellules. Elle contiendra les coordonnées de toutes les cellules d'un arbre. On nommera ce tableau « peres ».

-**Labyrinthe**, qui représente un mur du labyrinthe. Elle est composé des variables mur_hori (mur horizontal) et mur_vert (mur vertical). Ces variables vont représenter l'état d'un mur (0 si non présente, 1 si présente). Similairement à la structure Cellule, elle va être utilisé pour établir un tableau dynamique de cette structure nommé « mur » contenant l'état de tout les murs. Les indices de ces murs représentent leurs coordonnées.

Un tableau dynamique supplémentaire nommé « rang » est utilisé, qui va représenter la hauteur d'une cellule. Ce tableau sera important puisque c'est lui qui va permettre la suppression des murs à l'aide de l'implémentation de fusion par rang. Le tableau aura la même taille que le tableau de cellules pour les deux dimensions, ce qui permettra d'identifier le rang de chaque cellule à l'aide des indices.

Les fonctions

Fusion par rang : La suppression des murs

Chaque arbre représente une classe. Deux cellules sont dans le même arbre s'il existe un chemin entre les deux. Donc dans ce cas, on dit aussi qu'ils sont de même classe.
Comme supprimer un mur entre deux cellules va créer un nouveau chemin entre eux, alors l'une de ces deux cellules va être ajouté dans l'arbre de l'autre. Pour cela, on procède à la **fusion par rang.** Pour qu'elle soit faite, trois fonctions vont être utilisés :

Cellule TrouveNaif(Cellule **peres, Cellule cell) :

Cette fonction prend en paramètre une classe "peres" et une cellule "cell" et va permettre de trouver le représentant de classe de la cellule. Cette fonction va parcourir les nœuds de "peres" jusqu'à atteindre la racine, qui est le représentant de la classe. On va alors prendre les coordonnées de cette racine et les renvoyer. Cette fonction est importante car pour effectuer la fusion, il faut qu'on ait le représentant des deux cellules concernés.

Int MemeClasse(Cellule **peres, Cellule cel1, Cellule cel2) :

Cette fonction prend en paramètre une classe "peres" et deux cellule "cel1" et "cel2" et retourne 1 si les deux cellules sont dans la même classe et 0 sinon. Pour cela, on appelle la fonction TrouveNaif pour avoir le représentant pour les deux cellules et on vérifie si les deux représentants sont les mêmes. Si oui, alors ils sont de même classe. Cette fonction va être utile pour la boucle principale du programme, là où on effectue la fusion.

void FusionRang(Cellule **peres, int **rang, Cellule cell, Cellule cell2) :

Cette fonction prend en paramètre une classe "peres", le tableau "rang" et deux cellule "cel1" et "cel2" et permet d'effectuer la fusion des rangs. On appelle, pour les deux cellules, leur représentant respectif. On vérifie d'abord s'ils ont le même représentant. Si c'est pas le cas, on vérifie si le rang du représentant de cel1 est supérieur ou égal au rang du représentant de cel2. Si oui, le représentant de cel2 devient celle du cel1 et ainsi la fusion sera faite. On vérifie aussi le cas inverse. Si la fusion a été effectué par deux arbres de même rang, on augmente le rang de l'arbre fusionné de 1 pour montrer qu'une nouvelle cellule a été ajouté dans l'arbre.

L'affichage du labyrinthe

Il existe deux types d'affichages du labyrinthe : La version graphique et la version textuelle. Ces deux types d'affichages sont respectivement supportés par les deux fonctions suivantes :

void AffichageGraphique(Labyrinthe **mur, int n, int m)

Prend en paramètres la liste des murs du labyrinthe, la largeur (n) et la longueur (m) et affiche le labyrinthe sous forme graphique à l'aide des outils de MLV. La fonction, à l'aide d'une boucle imbriqué, va parcourir la liste selon un certain pas qu'on définit au début de la fonction. Ce pas a pour but d'élargir les cases en fonction de la taille de la fenêtre. On vérifie si la valeur à l'indice du compteur de la boucle est actif (c'est à dire si elle est égale à 1) et on vérifie cette condition pour les deux types de murs. Si c'est le cas, on trace la ou les ligne(s) à la position de l'indice.

void AffichageTexte(Labyrinthe **mur, int n, int m)

Prend en paramètres la liste des murs du labyrinthe, la largeur (n) et la longueur (m) et dessine le labyrinthe sous forme textuelle sur le terminal. On procède de la même façon que l'affichage graphique mais la principale différence va venir de la boucle : Puisqu'on a pas besoin d'élargir les cases, on enlève l'idée de pas. De plus, on va ajouter une condition supplémentaire qui va nous permettre de passer du mur horizontal au mur vertical à chaque fois qu'on saute une ligne.

La boucle principale

Les options

Pour les quatre ptions, l'idée est la même : on parcourt la chaine de caractère trouvé en argument pendant l'éxécution du programme et on la compare avec les quatre options disponibles à l'aide de strcmp.

--taille :
On parcourt les caractères suivant l'option –taille, à l'aide de deux boucles : Une pour extraire la longueur et une pour la largeur. Une fois extraite, on vérifie si ce sont des chiffres pour ensuite les convertir en int. On stocke alors ces nouvelles valeurs aux variables de longueur et largeur correspondantes.

--attente :
On procède de la même façon. La variable stockant la valeur d'attente va être utilisé en tant qu'argument pour l'appel de la fonction MLV_wait_seconds.

--graine :
On procède de la même façon. La variable stockant la graine va être utilisé en tant qu'argument pour l'appel de la fonction srand. Cela va permettre d'éviter de toujours avoir des labyrinthes différents à chaque éxécution du programme.

--mode=texte :
Si cette option figure, alors l'affichage graphique par défaut sera remplacé par l'affichage textuelle. Pour cela, on a une variable de condition "verif" (0 pour affichage textuel, 1 pour graphique) qu'on vérifiera à chaque fois qu'on aura besoin de faire l'affichage du labyrinthe.

--unique :
Pour cela, il suffit d'ajouter une nouvelle condition dans la partie où on supprime un mur. La variable de condition sera nommé "unique" et est utilisé de la même façon que vérif.

La formation du labyrinthe

On crée une boucle qui a comme condition d'arrêt la situation dans laquelle la première cellule (l'entrée du labyrinthe) et la dernière cellule (la sortie) sont de la même classe, c'est à dire s'il existe un chemin entre eux, ce qui signifiera que le labyrinthe est valide. C'est ici qu'on appele la fonction MemeClasse.

Sinon, on initialise une cellule test et on stocke des valeurs aléatoires sur ses coordonnées. On initialise également une variable aléatoire « mur_check » d'intervalle [0, 1] qui vérifie si on doit faire la fusion avec le mur horizontal (0) ou le mur vertical (1).

On initialise également une seconde cellule test qui va faire la fusion avec la première cellule. La cellule choisie va dépendre de quel type de mur va être supprimé : Si c'est un mur vertical, alors ce sera la cellule voisine du dessous de la premère cellule. Si c'est un mur horizontal, ce sera la cellule voisine de droite. Pour éviter de prendre une cellule inconnue (dans le cas ou on étudie une cellule au bord du labyrinthe), on va juste prendre la cellule voisine du haut/de gauche en tant que première cellule et la cellule au bord du labyrinthe en tant que deuxième cellule. Une fois la deuxième cellule choisi, on effectue la fusion et on actualise la fenêtre. On fait ce processus en boucle jusqu'à que le labyrinthe soit valide.