Projet Licence 2 Informatique S4: Génération aléatoire de Labyrinthe

Sommaire

Introduction

Ce projet a pour but de programmer un générateur aléatoire de labyrinthe en C. Ce labyrinthe devait être structuré de la manière suivante, une grille de format ***nxm***, avec ***n*** *lignes* et ***m*** *colonnes*. Cette grille est donc composée de cases nommées *« cellules »* qui ont chacune des coordonnées sous le format (i , j) avec i allant de 0 à (***n*** – 1) et j de 0 à (***m*** – 1). De plus, chaque cellule est séparée de ses voisines par des murs.

Chaque labyrinthe possède aussi une entrée en (0, 0) et une sortie en (***n*** – 1 , ***m*** – 1). Le générateur que nous avons codés doit donc sélectionner aléatoirement puis supprimer un mur qui sépare deux cellules. On répète cette opération jusqu'à ce qu'un chemin *« valide »* relie l'entrée à la sortie du labyrinthe.

Pour réaliser ce projet nous avons dû utiliser :

- L'algorithme d'*Union-Find* vu en cours d'algorithmique des arbres.
- La bibliothèque MLV.
- Nous avons aussi procédés à plusieurs améliorations, dans le but de rendre le projet plus fonctionnel.

Projet Licence 2 Informatique S4: Génération aléatoire de Labyrinthe

Le Labyrinthe

Notre code est donc organisé de la manière suivante :

On commence déjà par initialiser notre labyrinthe. Celui-ci est donc composé uniquement de cellules disjointes, séparées par des murs. Chaque cellule appartient à une classe différente au début. Une classe contient les cellules reliées entre-elles. Autrement dit, il existe un chemin qui relie les cellules d'une même classe. Chaque classe possède un représentant qui est l'une de ses cellules. Et de manière générale chaque cellules a un père et le représentant est son propre père. Tout ceci peut être représenté sous la forme de plusieurs arbres au début et au fur et à mesure on réduis le nombre d'arbre en les reliant. Pour cela on utilise l'algorithme d'***Union-Find*** .

Déroulé de l'algorithme d'***Union-Find*** :

- Relie les deux cellules qui étaient séparées par le mur.
- Cherche le représentant de chaque cellule
- Fait en sorte que le représentant de la plus grande classe devienne le père du représentant de l'autre classe.
- Fusionne les classes en une seule.

Pour cela nous avons structuré notre code de la manière suivante :

Les Structures

- coordonnees_t :
  stocke les coordonnées d'une cellule (abscisse, ordonnée)
- case_t :
  stocke la présence ou non des murs Est et Sud de chaque cellule.
  Stocke les coordonnées du père de la cellule.
  Stocke son rang, plus précisément, la hauteur potentielle de la cellule sur l'arbre.
  stocke son père, la cellule au-dessus dans l'arbre.
- laby_t :
  stocke ses dimensions.
  stocke le tableau dynamique à deux dimensions qui représente les cellules.

Projet Licence 2 Informatique S4: Génération aléatoire de Labyrinthe

Déroulé du Main

- Choix des arguments.
  - Si le mode texte est activé (**–mode=texte**) affichage console, sinon affichage graphique par défaut.
- On initialise le labyrinthe.
- Tant que l'entrée et la sortie ne sont pas reliées (on vérifie avec **laby_valide**).
  - On affiche le labyrinthe.
  - On utilise la fonction **supprime_mur.**
    - On sélectionne des coordonnées aléatoires.
    - On sélectionne un mur entre Sud et Est.
    - On recommence tant qu'on n'a pas trouvé un mur encore existant.
    - On supprime le mur.
    - On applique l'algorithme ***Union-Find*** sur les cellules qui ne sont plus séparées.

Les Améliorations

Comme il est précisé dans le sujet, nous avons ajouté plusieurs améliorations au code.

- L'argument qui permet de changer la taille du labyrinthe ***--taille=nxm*** ((***n***, ***m***) dimensions du labyrinthe). Par défaut, les dimensions étaient définies par des variables globales. Utilise fonction **lire_taille_laby()** qui lit l'argument et récupère ***n*** et ***m*** pour modifier la taille du labyrinthe.

- L'argument qui permet de fixer la graine. Permet lors d'une génération aléatoire de pouvoir rappeler plusieurs fois le même labyrinthe pour une même graine. On utilise **lire_entier()** qui va récupérer la valeur de la graine ***x*** dans l'argument ***--graine=x***.

- L'argument qui permet de rythmer le programme ***--attente=x.*** Si ***x*** = 0, on affiche directement le résultat final. Si ***x*** > 0, on supprime un mur toute les ***x*** millisecondes. Par défaut, la suppression d'un mur attend une entrée clavier de l'utilisateur. On utilise aussi **lire_entier()** pour récupérer ***x***.

- L'argument ***--unique*** qui empêche la destruction de murs entre deux cellules d'une même classe. Pour cela on utilise une version modifiée de **supprime_mur()** qui vérifie si le mur sélectionné est bien entre deux cellules de classes distinctes.

- L'argument --***acces*** ajoute comme contrainte le fait que toutes les cellules doivent être dans la même classe à la fin de la génération. Pour cela on ajoute dans la structure **laby_t** un entier qui compte le nombre de classe du labyrinthe. On décrémente ce nombre à chaque fusion de classes. L'accessibilité du labyrinthe est vérifiée lorsque le labyrinthe est valide.

- De plus, on a un affichage dans le terminal du temps d'exécution du programme .

- Par défaut, notre gestion de l'aléatoire n'était pas optimisée. En effet, nous devions sans arrêt vérifier l'existence ou non d'un mur, ce qui augmentait le temps d'exécution du programme à cause d'une redondance de calcul. Donc, avec l'argument --***fast*** nous utilisons deux structures supplémentaire. Il y a **position** qui est un type enum permettant la sélection du mur Est ou Sud. Et **cibleMur** stocke la position d'un mur par rapport aux coordonnées d'une cellule et les coordonnées elles-même.
  On crée un tableau qui stocke dans l'ordre croissant des coordonnées tous les murs internes du labyrinthe. On utilise **melange_tableau_mur()** qui échange de manière aléatoire les coordonnées du tableau. On utilise une nouvelle version de **supprime_mur()**, **supprime_mur_par_tableau()** qui parcourt les cases du tableau et supprime les murs associés, ainsi que son homologue **supprime_mur_par_tableau_unique()** qui ne supprime que les murs qui séparent deux cellules de classes différentes.

- Notre affichage en mode texte étant peu esthétique (principalement composé de « + » de « - » et de « | », nous avons, grâce au module ***structure.h***, créé une fonction d'affichage en UTF-8. Le résultat est plus soigné et plus esthétique, se rapprochant du dessin en mode graphique.

Conclusion :

Ce projet nous a donné l'occasion d'utiliser de manière un peu différente la notion d' ***Union-Find***. L'utilisation de la bibliothèque graphique MLV ne nous a guère causé de difficultés. Nous avons pu, avec grande satisfaction, réussir les améliorations demandées pour le projet, excepté celle recherchant un chemin victorieux. Nous sommes très satisfaits de notre programme et nous n'avons pas eu de problème particulier.