TP16 - Labyrinthe

4 mai 2023

L'objectif de ce TP est la génération de labyrinthes. On se place pour cela sur une grille carrée de taille $n \times m$. Le labyrinthe sera modélisé par un graphe dans lequel les cases de la grille sont les sommets et deux cases sont connectées si on peut passer de l'une à l'autre (i.e. adjacentes et pas de mur entre les deux).

1. Les graphes utilisés sont-ils orientés ou non orientés ?
2. Comment donner un numéro unique à chacune des cases de la grille ? Implémentez deux fonctions `int coord_to_id(int i, int j, int n, int m)` et `void coord_from_id(int id, int n, int m, int* x, int* y)` pour passer des coordonnées dans la grille à ce numéro unique et réciproquement.
3. Implémentez une fonction `bool is_adjacent(int id_1, int id_2, int n, int m)` qui détermine si deux cases de numéro `id_1` et `id_2` sont adjacentes.

Pour travailler avec des graphes, on décide d'utiliser une représentation par matrice d'adjacence. (Le listes en C, c'est désagréable ...).

4. Déterminez un type enregistrement (`struct`) qui représente un graphe et contient les informations utiles au sujet de ce graphe.
5. Implémentez une fonction `graph_t* create_graph(int nb_vertices)` qui crée un graphe avec `nb_vertices` sommets et aucun arc.
6. Implémentez une fonction `void free_graph(graph_t* g)` qui libère la mémoire utilisée par un graphe `g`.
7. Implémentez une fonction `bool is_neighbor(graph_t* g, int i, int j)` qui détermine si `j` est voisin de `i`.

Pour générer un labyrinthe, on va maintenant relier des cases les unes avec les autres : casser des murs pour créer des chemins.

8. Implémentez une fonction `add_edge(graph_t* g, int i, int j)` qui ajoute un arc du sommet `i` vers le sommet `j` dans le graphe `g`.
9. Implémentez une fonction de suppression d'arc `remove_edge(graph_t* g, int i, int j)`
10. Un labyrinthe sur une grille de taille $n \times m$ possède $n \times m$ sommet, mais il est pertinent de conserver l'information des dimensions pour pouvoir identifier les coordonnées des cases à partir de leur numéro en plus de la structure de graphe du labyrinthe. Implémentez une structure `labyrinth_t` pour réaliser ceci.
11. Implémentez une fonction `labyrinth_t* create_labyrinth(int n, int m)` qui crée un labyrinthe sur une grille de taille `n`×`m`.
12. Implémentez une fonction `free_labyrinth(labyrinth_t*)` qui libère la mémoire utilisée par un labyrinth.
13. Implémentez une fonction `void remove_wall(labyrinth_t* lab, int i_1, int j_1, int i_2, int j_2)` qui supprime le mur entre les cases de coordonnées i_1, j_1 et i_2, j_2.
14. Implémentez une fonction `void add_wall(labyrinth_t* lab, int i_1, int j_1, int i_2, int j_2)` qui ajoute un mur entre les cases de coordonnées i_1, j_1 et i_2, j_2.
15. Implémentez une fonction d'affichage d'un graphe interprété comme un labyrinthe. Pour plus de lisibilité on pourra procéder de la sorte : on utilisera un caractère pour un espace libre et un autre pour un mur. Les cases seront toujours des espaces libres. Une ligne sur deux représente les cases, et une sur deux les murs, même chose pour les colonnes. Voici un exemple de labyrinthe 3×3 dans lequel toutes les cases sont isolées.

```
#######
# # # #
#######
# # # #
#######
# # # #
#######
```

16. Testez cette fonction et les précédentes en créant manuellement un labyrinthe via un choix arbitraire des murs à casser. Affichez ce labyrinthe.

On souhaite créer un labyrinthe dans lequel il n'existe qu'un seul chemin d'une case à une autre. Pour cela on se sert d'une structure *union-find* dont le principe est le suivant : cette structure représente une partition $P = (P_i)$ d'un ensemble fini E c'est-à-dire qu'on a $\forall i \neq j,\ P_i \cap P_j = \emptyset$ et $E = \cup_i P_i$.
Son interface contient les fonctions suivantes :

— `init` qui initialise la structure avec la partition triviale telle que chaque élément soit seul dans son ensemble.

— `union` qui fusionne deux ensembles P_i et P_j dans la partition en un seul

— `find` qui renvoie le numéro de l'ensemble P_i de la partition auquel appartient un élément de E passé en argument.

Bien entendu, après la fusion de deux éléments de la partition via `union`, on souhaite que le numéro de la partition obtenu avec `find` tienne compte de la fusion.

17. Implémentez une structure *union-find* qui consiste en un tableau dont la case `i` contient le numéro de partition du sommet `i`.
18. Implémentez les fonctions `uf_t* create_uf(int nb_elements)`.
19. Implémentez les fonctions `void unite_classes(uf_t* partition, int i, int j)` et `int find(uf_t* partition, int element)`. (`union` est un mot clé réservé en C, on ne peut pas l'utiliser pour la fonction d'union)

Ici l'ensemble E est l'ensemble des sommets, et deux sommets sont dans la même partition s'il existe un chemin de l'un vers l'autre. Pour créer un labyrinthe on procède de la manière suivante :

— Choisit un arc (qui sont ici des murs à abattre ou non) au hasard

— On vérifie s'il relie deux cases qui sont dans la même partition. Si ce n'est pas le cas, on casse le mur et on met à jour la partition, sinon on passe au suivant.

— Quand on a considéré tous les murs, on a terminé, le labyrinthe est prêt.

Ici, il faut choisir des murs (qui sont des arcs, c'est-à-dire des couples de sommets du graphe). Pour déterminer le critère d'arrêt de l'algorithme, il faut garder trace des murs déjà considérés. Deux approches sont possibles :

— Générer un tableau mélangé de tous les murs et la parcourir du début à la fin

— Compter le nombre de murs déjà considérés. Tant que ce nombre n'a pas atteint le nombre total de murs, on tire un mur non traité au hasard. Pour cela on tire un numéro de mur au hasard tant qu'on n'a pas un mur non traité.

Remarque: Pour générer des nombres aléatoires en C, il est nécessaire d'inclure `#include <stdlib.h>` pour avoir accès à la fonction `rand`. Cette fonction ne prend pas d'argument et génère un entier pris uniformément entre `0` et `RAND_MAX` qui est une constante dont la valeur dépend de la machine.

Ainsi `rand() % n` génère un entier dans $[0, n-1]$, et `rand() % 2 == 0` un booléen. Pour obtenir un `float` dans $[0, 1]$ on peut utiliser `(float)rand()/RAND_MAX`. Cependant, pour utiliser la fonction `rand` il faut au préalable initialiser le générateur pseudo aléatoire. Pour cela on peut utiliser la fonction `srand` qui prend un entier et l'utilise comme graine pour le générateur pseudo aléatoire. Pour une graine fixée, le fonctionnement de `rand` sera déterministe : deux exécutions du programme seront identiques du point de vue du comportement de `rand`. Ce peut être pratique lors de la conception et de la phase de debug.

Pour une graine qui change à chaque exécution du programme on peut utiliser en début de fonction `main` `srand(time(NULL))` qui initialise le générateur pseudo aléatoire à l'aide de la date courante à l'exécution du programme. Pour avoir accès à la fonction `time`, un `#include <time.h>` est nécessaire.

20. Implémentez l'algorithme de génération de labyrinthe présenté plus haut en utilisant l'une des deux approches proposée pour obtenir des murs au hasard.
21. Affichez les labyrinthes obtenus.
22. (Bonus) Résolvez le labyrinthe à l'aide d'un algorithme de parcours.