C++ / Fichiers et conteneurs

Robotique/UE 3.1 - TD 04 - 2025/2026

Supports de cours disponibles sur
`www.simon-rohou.fr/cours/c++`

L'objectif de ce TD est de manipuler des fichiers et des conteneurs de la bibliothèque standard. On cherche à importer / exporter des problèmes de labyrinthes dans le TD précédent. On reprendra le code déjà développé, au moins jusqu'à la question TD3/9 incluse.

Les murs du labyrinthe étudié sont sans épaisseur, ce qui a mené à un choix de représentation par graphe. Il s'agit maintenant de *sérialiser* cette structure dans un fichier. Dans un premier temps et pour faciliter le développement, nous travaillerons sur un fichier texte. Une exportation sous forme binaire pourra ensuite être proposée.

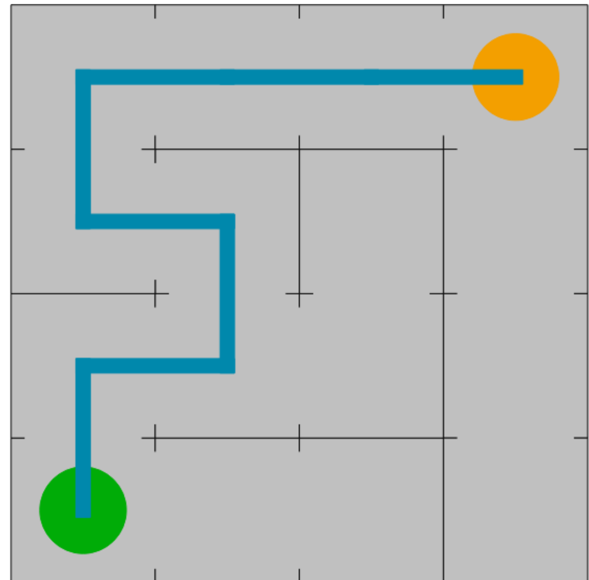

Figure 1: Un labyrinthe 4x4 simple.

A. Format de données

Rappel : un labyrinthe se définit par :

- un graphe non orienté de cellules
- une cellule de départ
- une cellule d'arrivée

Pour exporter un labyrinthe en fichier texte, on commencera par renseigner la cellule de départ, celle d'arrivée, puis l'ensemble des cellules (départ et arrivée comprises) avec le nombre de voisins et leurs positions.
L'exemple suivant correspond au labyrinthe 4x4 présenté en Figure 1 (le chemin solution n'est pas enregistré dans le fichier). Les deux labyrinthes présentés ci-contre sont téléchargeables sur le site du cours.

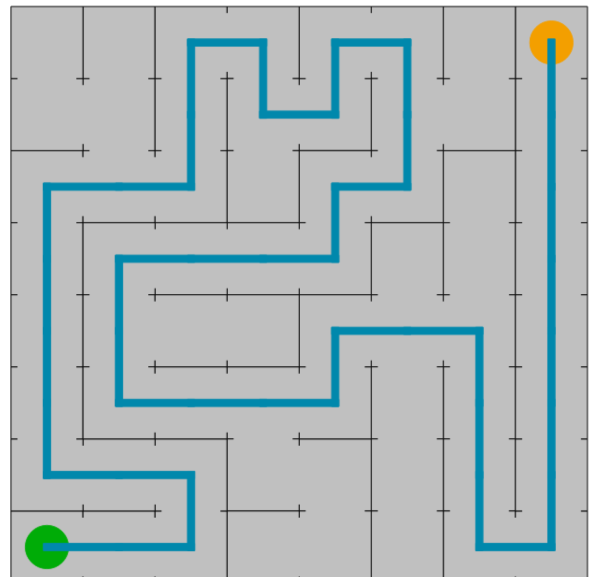

Figure 2: Un labyrinthe 8x8 plus complexe.

```
# Start:
(0,0)
# End:
(3,3)
# Cells:
(0,0)2(1,0)(0,1)
(1,0)2(0,0)(2,0)
(2,0)1(1,0)
(0,1)2(1,1)(0,0)
(1,1)3(0,1)(2,1)(1,2)
(2,1)2(1,1)(2,2)
(2,2)1(2,1)
(1,2)2(1,1)(0,2)
(0,2)2(1,2)(0,3)
(0,3)2(0,2)(1,3)
(1,3)2(0,3)(2,3)
(2,3)2(1,3)(3,3)
(3,3)2(2,3)(3,2)
(3,2)2(3,1)(3,3)
(3,1)2(3,2)(3,0)
(3,0)1(3,1)
```

B. Sérialiser un labyrinthe

Rappel : un labyrinthe est représenté par un type complexe `Maze` :

```
struct Maze
{
  Cell *c0, *cf;
};
```

1. Dans la classe `Cell` déjà implémentée, définir l'opérateur `<<` permettant de sérialiser un objet `Cell` dans un `ostream`. On respectera la convention `(x,y)` comme dans l'exemple de la section précédente.
2. Dans le programme principal, créer la procédure enregistrant un `Maze` dans un fichier :

   ```
   void save_maze(const Maze& maze, const string& file_name)
   ```

   On veillera à respecter le format de données présenté en Section A.
   NB : on pourra utiliser un implémentation récursive, comme pour l'affichage du labyrinthe (TD3, Section C), avec une autre procédure :

   ```
   void save_cell(Cell *cell, ofstream& f)
   ```

   Il faudra pour cela enrichir la classe `Cell` d'un attribut booléen dédié à cette sérialisation[1].
3. Enregistrer le labyrinthe du TD précédent.

C. Désérialiser un labyrinthe

Le parcours d'un fichier ne se fait pas nécessairement dans l'ordre permettant la construction du graphe. En particulier, pour qu'une cellule puisse définir ses voisins, ces derniers doivent avoir été instanciés et donc parcourus dans le fichier de données. Une approche est d'indexer toutes les cellules instanciées dans une `std::map` pendant la lecture du fichier. Chaque objet `Cell` de ce *dictionnaire* sera référencé par une clé `<x,y>`, unique puisqu'un espace du labyrinthe ne peut être occupé que par une cellule. Cela nous permet de découvrir le conteneur `std::map` de la librairie standard. Aussi, les clés de ce dictionnaire seront implémentées par un autre conteneur : `std::pair`.

4. Dans la classe `Cell`, définir l'opérateur `>>` permettant de désérialiser un objet `Cell` à partir d'un `istream`. On pourra utiliser la fonction `getline` permettant de réduire le stream à la portion qui nous intéresse (voir la documentation officielle en ligne), et la méthode `istream::get()` pour lire un caractère.
5. Dans le programme principal, créer une fonction retournant un labyrinthe à partir d'un fichier :

   ```
   Maze read_maze(const string& file_name)
   ```

   On utilisera par exemple une structure temporaire `map<pair<int,int>,Cell*> _cells` pour enregistrer les cellules (par pointeur) au fur et à mesure de la lecture du fichier.

   Note : la boucle `while(!f.eof()) { ... }` permet de parcourir un fichier de manière itérative. La méthode `eof()` (pour End Of File) retourne `true` une fois la fin de fichier atteinte.

D. (optionnel) Sérialisation binaire

6. Proposer une sérialisation plus compacte avec lecture/écriture de fichiers binaires.

[1] Modifier la classe `Cell` à des fins d'affichage ou de sérialisation n'est pas une approche correcte : la classe `Cell` doit se restreindre à la représentation numérique d'une cellule du labyrinthe. Ici, on y ajoute des booléens facilitant l'exploration du graphe uniquement par soucis de simplicité, au détriment d'une représentation correcte des données.