

C++ / Tableaux et pointeurs

Robotique/UE 3.1 - TD 3 - Octobre 2019

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

L'objectif de ce TD est de manipuler des pointeurs d'objets et de comprendre leur intérêt pour construire des graphes et les parcourir. On cherche à résoudre un problème de labyrinthe.

Les murs du labyrinthe étudié sont sans épaisseur. On choisit donc une représentation par graphe plutôt que par matrice (ou double tableau). Chaque nœud du graphe correspond à une cellule du labyrinthe, *i.e.* une case carrée de côté 1. Les connexions entre les cellules sont matérialisées par des arcs entre les nœuds.

Un labyrinthe se définit par :

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

A. Une cellule

On commence par implémenter la classe `Cell` :

1. Son constructeur crée l'objet positionné en $(\text{int } x, \text{int } y)$.
On laissera pour le moment tous les attributs de `Cell` en `public`.

2. On ajoute d'autres attributs de classe dans `Cell.h` :

- `int m_nb_neighb = 0;`
- `Cell **m_neighb = NULL;`

qui renseignent le nombre et les pointeurs vers les cellules voisines. `m_neighb` est un tableau de pointeurs vers des objets `Cell`.

3. On implémente la méthode `add_neighb(Cell *c)` qui ajoute un pointeur au tableau `m_neighb` par allocation dynamique, s'il n'est pas déjà présent dans le tableau. Comme il s'agit d'un graphe non orienté, on appelle de manière récursive `c->add_neighb(this)`; à la fin de la méthode¹.
4. Pour faciliter la création du labyrinthe, on ajoute également une méthode `add_neighb(Cell *c1, Cell *c2)`.

B. Un labyrinthe

5. Proposer un *type complexe* (`struct`) définissant très simplement un labyrinthe, et nommé `Maze`. Attention : la définition d'un labyrinthe par un tableau (ou `vector`) de cellules n'est pas adaptée à une représentation par graphe. Réfléchir à une représentation des données qui ne soit pas redondante.

¹En C++, au sein d'une classe, le mot-clé `this` est un pointeur sur l'objet lui-même.

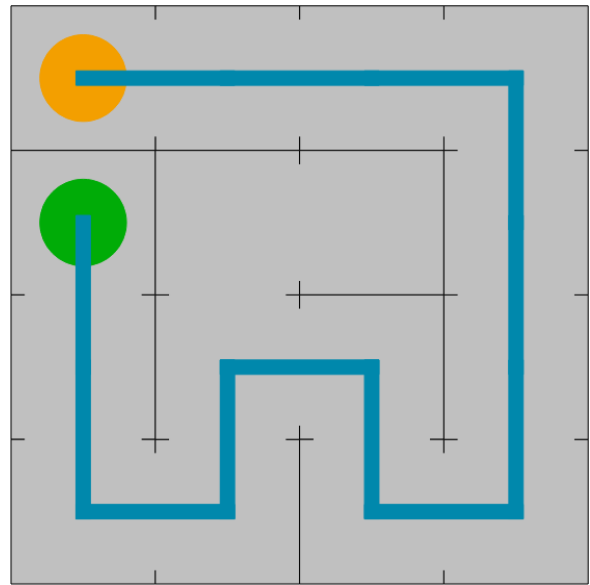


Figure 1: Un labyrinthe 4x4.

C. Affichage

6. Un affichage récursif du labyrinthe est proposé. Il repose sur un flag `m_displayed` (un booléen) qui est levé pour chaque cellule après affichage. Ajouter cet attribut dans la classe (`false` par défaut) et intégrer les méthodes suivantes dans `main.cpp` :

```
void draw_gate(const Cell *n1, const Cell *n2)
{
    vibes::drawBox(min(n1->m_x, n2->m_x) + 0.1, max(n1->m_x, n2->m_x) + 0.9,
                   min(n1->m_y, n2->m_y) + 0.1, max(n1->m_y, n2->m_y) + 0.9,
                   "lightGray[lightGray]");
}

void display_graph(Cell *cell)
{
    vibes::drawBox(cell->m_x, cell->m_x + 1, cell->m_y, cell->m_y + 1, "[lightGray]");
    cell->m_displayed = true;

    for(int i = 0 ; i < cell->m_nb_neighb ; i++)
    {
        if(!cell->m_neighb[i]->m_displayed)
            display_graph(cell->m_neighb[i]);
        draw_gate(cell, cell->m_neighb[i]);
    }
}
```

7. En vous inspirant du TD précédent pour les fonctionnalités de VIBes, tester l'affichage d'une cellule en (0,0) dans une vue graphique².

D. Création d'un labyrinthe

8. Dans `main.cpp`, créer un labyrinthe 4x4 à travers une fonction `create_maze()`. On pourra s'inspirer de la Figure 1. La fonction retournera un variable de type complexe `Maze`.
- commencer par allouer dynamiquement chaque cellule du labyrinthe (16 pointeurs d'objets).
 - à l'aide de la méthode `set_neighb()`, définissez les connexions (arcs du graphe) entre les cellules.
 - créer un labyrinthe à partir de ces pointeurs.
9. Implémenter une procédure `void display(Maze m)` dessinant le labyrinthe (avec éventuellement une mise en évidence des points de départ et d'arrivée).

E. Le chemin

Le chemin entre la cellule de départ et celle d'arrivée est matérialisé par une liste chaînée de cellules à parcourir.

Une *liste chaînée* est une structure de données représentant une collection d'éléments de même type. Cet ensemble est ordonné et de taille arbitraire. Son implémentation se fait sous la forme d'une succession d'éléments contenant un objet ainsi qu'un pointeur vers l'élément suivant. L'accès aux éléments d'une liste se fait donc de manière séquentielle : chaque élément permet l'accès au suivant (contrairement au tableau dans lequel l'accès se fait de manière directe, par adressage de chaque cellule dudit tableau).

10. Proposer une classe `Path` implémentant une liste chaînée de pointeurs sur des objets `Cell`. La définition de la classe `Path` est donnée par :

²Pour un labyrinthe 4x4, utiliser : `vibes::axisLimits(0-0.5, 4+0.5, 0-0.5, 4+0.5);`

```

class Path
{
public:

    Path(const Cell *c);
    ~Path();
    void add_to_path(const Cell *c);
    // ...

protected:

    const Cell *m_c;
    Path *m_next = NULL;
};

```

11. Implémenter une méthode simple et récursive affichant le chemin.

F. Résoudre le labyrinthe

12. Proposer une méthode pour résoudre le labyrinthe. Par exemple avec une approche simple et récursive :

```
bool find_path(Cell *c, Cell *cf, Path *path)
```

où `c` pointe sur la cellule courante et `cf` sur l'objectif. On ajoutera un autre flag à la classe `Cell` permettant l'exploration du graphe.

13. Afficher le chemin sur le labyrinthe.
14. S'assurer que toutes les allocations dynamiques ont été désallouées à la fin de l'exécution du programme. Utiliser le programme `valgrind` pour détecter les éventuelles fuites mémoire.

G. (optionnel) Optimisation

15. Construire un labyrinthe plus grand et plus complexe.
16. Proposer un algorithme de recherche de chemin plus optimisé, retournant le chemin le plus court pour sortir du labyrinthe. S'inspirer d'algorithmes déjà bien connus.