C++ / Tableaux et pointeurs

Robotique/UE 3.1 - TD 03 - 2025/2026

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.

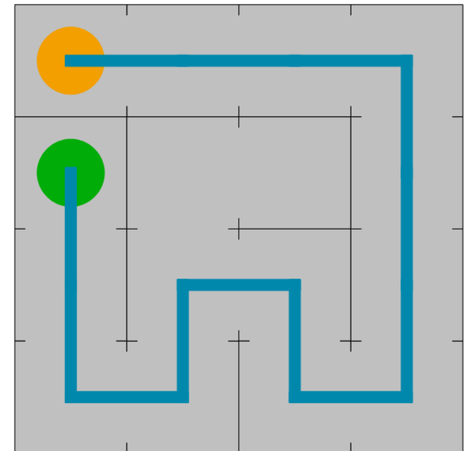

Figure 1 – Un labyrinthe 4x4.

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 créé l'objet positionné en `(int x, 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 _nb_neighb = 0;`
   - `Cell **_neighb = nullptr;`

   qui renseignent le nombre et les pointeurs vers les cellules voisines. `_neighb` est un tableau de pointeurs vers des objets de type `Cell`.
3. On implémente la méthode `add_neighb(Cell *c)` qui ajoute un pointeur au tableau `_neighb` par allocation dynamique, s'il n'est pas déjà présent dans le tableau. Comme il s'agit d'un graphe non orienté (une cellule est voisine de sa voisine), on appelle de manière récursive `c->add_neighb(this);` à la fin de la méthode [1].
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, selon le principe DRY [2].

C. Affichage

6. Un affichage récursif du labyrinthe est proposé. Il repose sur un flag `_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` :

1. En C++, au sein d'une classe, le mot-clé `this` est un pointeur sur l'objet lui-même.
2. `https://en.wikipedia.org/wiki/Don%27t_repeat_yourself`

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

void display_graph(Cell *cell)
{
  vibes::drawBox(cell->_x, cell->_x + 1, cell->_y, cell->_y + 1, "[lightGray]");
  cell->_displayed = true;

  for(int i = 0 ; i < cell->_nb_neighb ; i++)
  {
    if(!cell->_neighb[i]->_displayed)
      display_graph(cell->_neighb[i]);
    draw_gate(cell, cell->_neighb[i]);
  }
}
```

7. Ajouter des commentaires pour chaque instruction du code fourni à la Question 6.
8. En s'inspirant du TD précédent pour les fonctionnalités de VIBes, tester l'affichage d'une cellule en `(0,0)` dans une vue graphique [3]. Ne pas oublier d'initialiser VIBes avec les instructions `vibes::beginDrawing()`, `vibes::newFigure(..)`, *etc.*

D. Création d'un labyrinthe

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

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

E. Le chemin

Le chemin entre la cellule de départ et celle d'arrivée est matérialisé par une liste chaînée de cellules à parcourir, comme représenté sur la Figure 2.

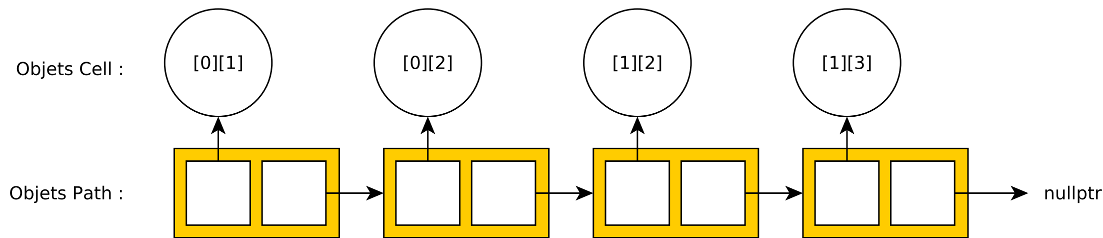

FIGURE 2 – Le `Path` (chemin solution du labyrinthe) est une *liste chaînée* d'éléments pointant vers des objets `Cell`. Chaque élément chaîné (en jaune) contient donc un pointeur vers une cellule du labyrinthe, et un pointeur vers l'élément `Path` suivant. Un objet `Path` représente donc à la fois une liste chaînée, ainsi que le premier élément de cette liste. Le dernier élément n'existe pas : il est `nullptr` (*i.e.* un pointeur ne pointant sur rien).

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).

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

```
class Path
{
  public:

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

  protected:

    const Cell *_c;
    Path *_next = nullptr;
};
```

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

F. Résoudre le labyrinthe

13. 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. Par *récursif*, on comprend que `find_path(..)` s'appellera elle-même. Cela permet de définir une technique de résolution du problème très simple. Elle sera en revanche peu optimale [4], mais cela ne sera pas critique sur des labyrinthes de la taille de ceux traités dans ce TD et dans le suivant.
14. Afficher le chemin sur le labyrinthe.
15. 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.

4. ≪ *La récursion est la racine du calcul car elle échange la description contre du temps* ≫, Alan J. Perlis, 1982.

G. (optionnel) Optimisation

16. Construire un labyrinthe plus grand et plus complexe.
17. 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.