

Construction Objets Avancée : TP 4

Giuseppe Lipari

April 5, 2021

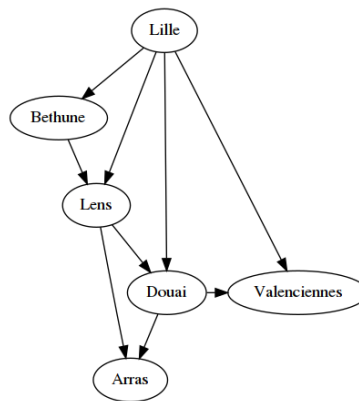
Graph library

Un *graphe orienté* est une structure de données utilisé en mathématique et en informatique pour représenter des réseaux d'objets. Le but de ce TP est de construire une *graph template library* pour créer et manipuler de graphes.

Définitions

Un *graphe orienté* est un ensemble de noeuds connectés par des arêtes orientées (*edges*). Il est possible d'associer des objets (étiquettes) aux noeuds et aux arêtes.

Par exemple, on veut représenter le réseau des routes dans le Nord. Chaque noeud représente une ville et les arêtes sont les routes. Dans ce cas, à chaque noeud on associe un objet de type **Ville** et à chaque arête on associe un objet de type **Route**.



Il y a plusieurs manières de représenter un graphe. Dans ce TP nous utiliserons une implémentation très simple.

- Chaque noeud est représenté par une structure de donnée. Un noeud possède un entier non négatif qui sert comme identifiant unique, et l'objet associé au noeud (par exemple une string) ;

- La liste de noeuds est gardé dans une *map* qui associé à chaque identifiants le noeud correspondant ;
- Une arête est une couple de noeuds, et elle est associé avec une étiquette ; elle possède un identifiant unique.
- La liste des arêtes est gardée aussi dans une *map* qui associé à chaque identifiant la structure représentant l'arête.
- Finalement, on utilise une *map* qui associe à chaque noeuds la liste des arêtes *sortantes* du noeud (c'est à dire : les arêtes qui ont le noeud comme source).

Voici la squelette de code que nous utiliserons.

```

class Graph {
    struct Node {
        int node_id;
        std::string data;
    };
    struct Edge {
        int edge_id;
        std::string data;
        int source_id;
        int dest_id;
    };

    /* data structures */
    std::map<int, Node> nodes;
    std::map<int, Edge> edges;
    std::map<int, std::vector<int>> dests;
    int id_counter;
    int edge_counter;

public:

    Graph() : id_counter{0}, edge_counter{0} {}
    Graph(const Graph &other) :
        nodes(other.nodes), edges(other.edges), dests(other.dests),
        id_counter(other.id_counter), edge_counter(other.edge_counter) {}

    inline int add_node(const std::string &m) { /* todo */ }

    inline bool node_exist(int id) const { /* todo */ }

    inline int add_edge(const std::string &m, int source_id, int dest_id) {
        /* todo */
    }

    inline void remove_node(int node_id) {
        /* todo */
    }

    inline int search_node(const std::string &m) const {
        /* todo */
    }

    inline std::string get_node_data(int node_id) const {
        /* todo */
    }

    inline std::string get_edge_data(int edge_id) const {
        /* todo */
    }

    inline int get_edge_source(int edge_id) const {
        /* todo */
    }

    inline int get_edge_dest(int edge_id) const {

```

Le but est de faire une librairie. D'abord on développe une classe non-template, où les noeuds et les arêtes sont associés à des strings.

Question 1 : coder la classe

Les méthodes à implémenter:

- un *copy constructor* ;
- `add_node` ajoute un noeud dans le graph et retourne l'identifiant unique du noeud (un entier).
- `add_edge` ajoute une arête entre deux noeuds à partir de leur ids. Il retourne l'identifiant unique de l'arête (un entier).
- `get_node_data()` et `get_edge_data()` retournent les contenus à partir des ids.
- pour une arête, `get_source` et `get_dest` retournent les identifiants des noeuds source et destination.
- pour un noeud, `get_successors` retourne un vecteur d'arêtes sortants; `get_predecessor` retourne un vecteur d'arêtes entrants
- Un `Path` est juste un vecteur d'arêtes qui marque un chemin dans le graphe.
- La fonction `all_paths` retourne la liste de tous les chemins possibles d'un noeud `from` au noeud `to`. Si aucun chemin existe, il retourne un vecteur vide. **Attention : pour simplifier le codage, on assume que le graphe ne contient jamais de boucles !**
- Pour l'algorithme `shortest_path`, on utilisera l'algorithme de Dijkstra.

Implémenter la classe. Écrire de tests pour vérifier qu'elle fonctionne correctement.

Question 2 : template

Généraliser la classe `Graph` pour associer aux noeuds et aux arêtes des objets d'un type quelconque :

- La classe `Graph` devient une classe template:

```
template<class ND, class ED>
class Graph {
    ...
};
```

- Les fonctions `get_node_data(int node_id)` et `get_edge_data(int edge_id)` doivent retourner les objets correspondants :

```
template<class ND, class ED>
class Graph {
    ...
public:
    ND get_node_data(int node_id) const { /* todo */ }
    ED get_edge_data(int edge_id) const { /* todo */ }
};
```

Vérifier que les tests sont encore correctes quand on spécifie des strings pour cette classe.

Suggestion : codez la classe template dans un fichier différent, par exemple dans `graph_t.hpp`. Il faut inclure l'un ou l'autre.

Question 3: Décoration

On voudrait décorer les arêtes avec des informations supplémentaires sans forcément modifier la classe associée aux arêtes.

Par exemple, supposez que le graphe représente les routes dans le département du Nord. On associe un `std::string` aux arêtes avec le nom de la route. Plus tard, on voudrait ajouter l'information sur la longueur en Km de la route.

Voici comment on fait:

- On prépare une classe template variadique `EdgeData` qui contient des strings.

```
template <typename ...Tp>
class EdgeData : public Tp ... {
    std::string str;
public:
    void set_string(const std::string &s) { str = s; }
    std::string get_string() const { return str; }
};
```

- On déclare une class `RouteLenght` :

```
class RouteLenght {
    double l;
public:
    void set_lenght(double len) { l = len; }
    double get_lenght(double len) const { return l; }
};
```

- Maintenant, la classe `EdgeData<RouteLenght>` contient une `string` et un `double`, et on peut l'utiliser comme dans le code suivant :

```
EdgeData<RouteLenght> data;
data.set_string("Lille-Valenciennes");
data.set_lenght(44.14);
```

- On déclare une instance de `Graph` ayant comme paramètre la classe `EdgeData<RouteLenght>` :

```
Graph<string, EdgeData<RouteLenght>> mygraph;

int lille = mygraph.add_node("Lille");
int valen = mygraph.add_node("Valenciennes");
mygraph.add_edge(data, lille, valen);
```

Tester le bon fonctionnement de cette technique. Ajouter aussi une deuxième propriété `AverageTime` pour mémoriser le temps moyenne de parcours d'une route, et vérifier que tout fonctionne correctement.

Question 4: Généralisation de `shortest_path`

On voudrait spécialiser `shortest_path` pour prendre en compte une propriété générique des edges. Par exemple, dans la cas d'un graphe qui représente les routes du département, on voudrais calculer le parcours plus court, ou le parcours avec le plus grande nombre de stations d'essence, etc.

Pour faire ça, la méthode devient une méthode *template* qui prends comme paramètre une fonction d'évaluation de la métrique sur les arêtes.

Écrire la méthode template `shortest_path`, et tester avec la classe `EdgeData<RouteLenght>` implémentée dans la question précédente.

Question 5: shared pointers

Dans les questions précédentes il n'y a pas manière de changer les informations associés aux noeuds et aux arêtes. Pour permettre ça, on va changer de strategie.

- Dans la struct `Node` et dans la struct `Edge` on memorise un `shared_ptr` vers l'objet associé

```

template <typename N, typename E>
class Graph {
    struct Node {
        int node_id;
        std::shared_ptr<N> data;
    };
    struct Edge {
        int edge_id;
        int source_id;
        int dest_id;
        std::shared_ptr<E> data;
    };
    /* etc. */
};

```

- Les fonctions `get_node_data_` et `get_edge_data` retournent un `shared_ptr<>` vers l'objet associé qu'on peut modifier après:

```

std::shared_ptr<N> get_node(int node_id) {/*todo*/}
std::shared_ptr<E> get_edge(int edge_id) {/*todo*/}

```

Implémentez cette nouvelle version.

- Tester le scénario suivant :
 1. Un utilisateur crée un graphe de distances entre des villes dans le département du nord.
 2. Il calcule le chemin minimale en utilisant la technique implémenté à la question 4.
 3. Il modifie un distance.
 4. Il recalcule la chemin optimale et il obtient un parcours différent.
- Tester qu'une référence obtenue avec `get_node_data()` est toujours valide après avoir détruit le graphe.

Question 6: copie profonde

Le copy constructor par défaut fait une copie *shallow* du graph, et donc les objets pointés par le `shared_ptr` ne sont par copiés.

Ajouter une fonction pour faire la copie profonde du graph:

```
template <typename N, typename E>
class Graph {
    /* ... */
public:
    Graph() {}
    Graph (const Graph &other) { /* shallow copy */ }
    Graph deep_copy() const { /* deep copy */ }
    /* ... */
};
```

La fonction vérifie si le type `N` est polymorphique: si oui, on utilise la fonction `clone`, si non on utilise le copy constructor. Même chose pour le type `E` (il faut utiliser la technique `if constexpr` du C++17 vue en cours).

Tester que le copie sont effectué correctement: en particulier, dans le cas d'une copie profonde, si on modifie l'objet original, la copie n'est pas modifiée.

Curiosité

Les distances entre villes du département du Nord:

<http://www.lion1906.com/departements/nord/>