Construction Objets Avancée

Giuseppe Lipari

March 25, 2019

Instructions

Vous devez rendre sur gitlab le code demandé avec un fichier README.md qui contient :

- Vos noms ;
- Pour chaque question :
  - Si vous avez réussi à coder les fonctionnalités demandées
  - La liste de tests de régression correspondants à la question

TP 6 : graph library

Un *graphe orienté* est une structure de données utilisé en mathematique et informatique pour representer des reseaux d'objets. Le but de ce TP est de construire une librarie template pour créer et manipuler de graphes.

Définitions

Un graphe orienté est un ensemble de noeuds connectés par des arêtes orientées (flèches). Il est possible d'associer des objets aux noeuds et aux arêtes. Un graphe a un (ou en générale plusieurs) points d'entrée.

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

Il y a plusieurs manières pour representer un graph: la plus simple est de représenter la liste de noeuds et la liste des arêtes. Chaque arête est une couple de noeud (source, destination). Pour faciliter la navigation, la liste des arêtes peut être implémentée avec une `map<>` qui associe le noeuds source à l'arête.

Question 1: Graph de strings

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

```
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 */
    /* todo */

public:

    Graph() {}
    Graph(const Graph &other) {/*todo*/}

    int add_node(const std::string &m) { /*todo*/ }
    int add_edge(const std::string &m, int source_id, int dest_id) {/*todo*/}
    int remove_node(int node_id);

    int search_node(const std::string &m) const {/*todo*/}
    std::string get_node(int node_id) const {/*todo*/}
    std::string get_edge(int edge_id) const {/*todo*/}

    int get_source(int edge_id) const {/*todo*/}
    int get_dest(int edge_id) const {/*todo*/}

    std::vector<int> get_successors(int node_id) const {/*todo*/}
    std::vector<int> get_predecessors(int node_id) const {/*todo*/}

    using Path = std::vector<int>;

    std::vector<Path> all_paths(int from, int to) const {/* todo */}

    Path shortest_path(int from, int to) const {/*todo*/}
};
```

Les methodes à implementer:

- 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()` et `get_edge()` 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 parcours dans le graphe.
- La fonction `all_paths` retourne la liste de tous le parcours possible d'un noeud `from` au noeud `to`. Si aucun parcours existe, il retourne un vecteur vide.
- La fonction `shortest_path` retourne le parcours plus court entre les deux noeuds en paramètre. Si aucun parcours existe, il retourne un parcour vide.

Implementer la classe. Écrire de tests pour vérifier qu'il marche bien. Pour la méthode `shortest path`, utilisez l'algorithme de Djikstra.

Question 2 : template

Generaliser la classe Graph pour associer aux noeuds et aux arêtes des objets d'un type quelconque.

Écrire de test de régression.

Question 3: Decoration

On voudrait decorer les edges avec des informations supplémentaires sans forcement modifier la classe associé aux arêtes.

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

Voici comment on fait:

- On prépare une classe template variadique `EdgeData` qui ne contient rien

```
template <typename ...Tp>
class EdgeData : public Tp ... {
};
```

- La classe graphe sera paramétré avec `EdgeData<std::string, RouteLength>`, et cette dernière est:

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

Tester le bon fonctionnement de cette technique. Ajouter une propriété `AverageTime` pour mémoriser le temps moyenne de parcours d'une route.

Question 4: Généralisation de `shortest_path`

On voudrait specialiser `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 methode template `shortest_path`, et tester avec la classe `EdgeData<std::string, RouteLenght>` implémentée dans la question précédente.

Question 5: shared pointers

Dans les questions précédentes il n'y a pas de methode pour changer les informations associés aux nodes et aux arêts. 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

```cpp
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;
   };
   /* todo */
};
```

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

  ```cpp
  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 villes dans le département du nord.
  2. Il calcule le parcours minimale en utilisant la technique implémenté à la question 4.
  3. Il modifie un distance.
  4. Il recalcule la parcours optimale et il obtient un parcours différent.

- Tester qu'une référence obtenue avec `get_node()` est toujours valide après avoir détruit le graph.

Question 6: copie profonde

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

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 verifie 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/`