POO & C++ : TP 4

EISE4 2014—2015

Pablo Rauzy

rauzy@enst·fr

pablo.rauzy.name/teaching.html#epu-cpp

17 octobre 2014

N'oubliez pas :

- Les TPs doivent être rendus par courriel à `rauzy@enst·fr` au plus tard le lendemain du jour où ils ont lieu avec `[EISE4]` suivi de vos noms dans le sujet du mail.
- Le code rendu doit être propre !
- Le TP doit être rendu dans une archive tar gzippée : `tar czvf NOMS.tgz NOMS` où `NOMS` est le nom du répertoire dans lequel il y a votre code (idéalement, les noms des gens du groupe) ; attention à ne pas rendre un répertoire avec des fichiers temporaires dus à la compilation (pensez à faire un `make clean` avant `tar`).
- Dans le répertoire avec vos noms, chaque exercice doit être dans son propre répertoire.
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt`.
- Le code est de préférence en anglais, avec des commentaires en français ou anglais, en restant cohérent.
- Le code doit être proprement indenté.
- Respectez les conventions de nommage :
  - variables : `nom_explicite` (*e.g.*, `game_type`),
  - fonctions : `verbeAction` (*e.g.*, `launchNewGame`),
  - classes : `NomExplicite` (*e.g.*, `BoardGame`),
  - membres privés ou protégés : comme variable ou fonction + `_` (*e.g.* `max_number_of_player_`, ou `updateState_`),
  - accesseurs d'affectation : `set_` + nom de l'attribut sans le `_` final (*e.g.*, `set_max_number_of_player`),
  - accesseurs de consultation : nom de l'attribut sans le `_` final (*e.g.*, `max_number_of_player`),
  - constantes : `NOM_EXPLICITE` (*e.g.*, `VERSION_NUMBER`).
- Pensez à utiliser les outils de développement comme le debugger `gdb` ou le profiler `valgrind`.
- **N'hésitez jamais à chercher de la documentation par vous-mêmes sur le net !**
  Par exemple la STL est documentée ici : `http://en.cppreference.com/w/cpp`.

Dans ce TP :

- la STL (Standard Template Library).
- Algorithme de Dijkstra.

Exercice 0.

Récupération des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP devrait être dans `~/cpp/tp4/`.

2. Récupérez les fichiers nécessaires au TP sur la page du cours.

3. Une fois que vous l'avez extrait de l'archive (`tar xzf tp4.tgz`), renommez le répertoire `tp4` en les noms de votre groupe (par exemple en `Dupont-Dupond`). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.

Exercice 1.

Un *graphe* est une structure de donnée très commune en informatique. Elle consiste en un ensemble de *sommets* et un ensemble d'*arêtes* reliants ces sommets. Formellement, on note un graphe $G = (V, E)$ où V est l'ensemble des sommets et où $E \subset V \times V$ est l'ensemble des arêtes, c'est à dire qu'une arrête est une paire de sommets.

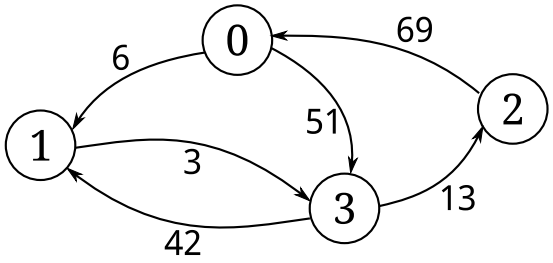
Par exemple, on peut voir le réseau des routes nationales comme un graphe où les villes sont des sommets et les routes sont des arêtes. Dans ce cas il y aura une arête entre deux sommets v_1 et v_2 si et seulement si il y a une route nationale directe de la ville représentée par v_1 et la ville représentée par v_2.

Si les arêtes d'un graphe ont un sens, on dit que le graphe est *orienté*. Dans ce cas on appelle les éléments de la paire représentant l'arête respectivement sa *source* et sa *cible*.

Par exemple on pourrait imaginer que la nationale reliant v_1 à v_2 soit à sens unique, dans ce cas on a une arête qui va de v_1 à v_2 mais pas de v_2 à v_1.

Une autre information importante dans un graphe est ce qu'on appelle le *poids* des arêtes. C'est une information attachée à chaque arête. Un graphe qui contient un poids pour chacune de ses arêtes est dit *pondéré*.

Dans notre exemple, le poids des arêtes pourrait correspondre à la longueur en kilomètre des routes. Dans ce cas le poids de l'arête (v_1, v_2) sera égale au nombre de kilomètres que fait la nationale entre les villes v_1 et v_2.



Dans ce graphe, $V = \{0, 1, 2, 3\}$ et $E = \{(0, 1), (0, 3), (1, 3), (2, 0), (3, 1), (3, 2)\}$.

1. Pour représenter un graphe, on a besoin de connaître son nombre de sommets S et on numérote les sommets de 0 à $S - 1$. Pour représenter les arêtes on peut utiliser ce qu'on appelle des *listes d'adjacences*. C'est à dire que pour chaque sommet du graphe on a la liste des sommets qui lui sont directement reliés. Formellement, dans un graphe $G = (V, E)$, si l est la liste d'adjacence du sommet $v \in V$, alors v' est un élément de l si et seulement si $(v, v') \in E$.

Pour représenter un graphe non-orienté, on ajoute simplement deux arêtes : une dans chaque sens.

Quand le graphe est pondéré, les éléments des listes d'adjacences contiennent aussi le poids de l'arête en plus du sommet cible.

Une représentation de ces listes pour le graphe donnée en exemple plus haut serait :

```
|- 0
|  `--(51)--> 3
|  `--(6)--> 1
|
|- 1
|  `--(3)--> 3
|
|- 2
|  `--(69)--> 0
|
|- 3
|  `--(13)--> 2
|  `--(42)--> 1
```

→ Quel sont selon vous les bon conteneurs de la STL à utiliser pour représenter un graphe ? Justifiez votre choix.

Réponse :
Plusieurs réponses possibles.
Le plus évident est d'utiliser un vecteur de listes ou directement un tableau de listes si on est sûr que le nombre de sommets ne changera pas. Le vecteur ou tableau permet d'accéder directement en temps constant à l'ensemble des arrêtes partant de n'importe quel sommet.
Les listes permettent l'ajout en temps constant et le parcours en temps linéaire. C'est en fait un choix informé par la suite du TP et la connaissance de l'algorithme de Dijkstra. Par exemple si on avait trouvé plus important de pouvoir rapidement vérifier l'existence d'une arrête entre deux sommets, on aurait pu choisir un tableau ou un vecteur là aussi.
Les éléments des listes peuvent être des pairs avec par exemple le premier élément qui désigne le sommet cible de l'arête et le second son poids.

```
1 typedef std::pair<uint, uint> edge_t;
2 typedef std::list<edge_t> adjacency_list_t;
3 std::vector<adjacency_list_t> edges_;
```

2. La classe `Graph` utilisée dans le `main` qui vous a été fourni n'existe pas encore, il faut donc la créer.

→ Dans le fichier `graph.hpp`, vous allez déclarer une classe `Graph` qui doit :

- avoir un un constructeur qui prend en argument le nombre de sommet du graphe ;
- avoir une méthode `addEdge` pour ajouter des arêtes au graphe, qui prend trois arguments :
  - le sommet de départ de la nouvelle arête,
  - son sommet d'arrivée,

  - son poids ;
- déclarer les types
  - de vos listes d'adjacences (en fonction des conteneurs de la STL que vous aurez choisi),
  - des éléments de ces listes (qui doivent représenter le sommet cible et le poids de l'arête) ;
- posséder en membre privé la structure de donnée représentant le graphe (et qui doit correspondre à votre réponse à la question 1).

Réponse :

```hpp
#ifndef GRAPH_HPP___
#define GRAPH_HPP___

#include <vector>
#include <list>
#include <utility>

class Graph
{
public:
  typedef std::pair<uint, uint> edge_t;
  typedef std::list<edge_t> adjacency_list_t;

  Graph (uint vertex_count);

  void addEdge (uint from, uint to, uint weight);

private:
  std::vector<adjacency_list_t> edges_;
};

#endif // GRAPH_HPP___
```

3. → Dans `graph.cpp`, implémentez les fonctions membres de votre classe `Graph`.

Réponse :

```cpp
Graph::Graph (uint vertex_count)
{
  edges_.resize(vertex_count);
}

void Graph::addEdge (uint from, uint to, uint weight)
{
  edges_[from].push_front(std::make_pair(to, weight));
}
```

4. Il est maintenant indispensable de tester votre code. Le fichier `main.cpp` qui vous a été fourni contient une fonction `readGraph` qui lit un graphe pondéré et orienté sur le flux d'entrée qu'on lui passe en argument.

Cette fonction commence par lire un premier nombre qui sera le nombre de sommets du graphe. Ensuite elle lit le nombre d'arêtes qu'elle devra ajouter au graphe, puis les arêtes une par une en les ajoutant au graphe avec la méthode `addEdge`. Pour chaque arête trois nombres sont lu : sa source, sa cible, et son poids.

Par exemple, si on appelle la fonction `readGraph` avec `std::cin` en argument comme c'est fait dans le `main`, le graphe d'exemple dessiné plus haut peut être donné sur l'entrée du programme comme ceci :

```
4
6
0 1 6
0 3 51
1 3 3
2 0 69
3 1 42
3 2 13
```

→ Vérifiez bien que votre code compile et ne plante pas à l'exécution.

5. → Implémentez un affichage de la structure de votre graphe dans une méthode `displayStructure`. Pensez à utiliser les itérateurs de la STL si vous en avez besoin.

N'oubliez pas de tester régulièrement votre code : ajoutez un appel à la méthode `displayStructure` dans votre `main` pour afficher les graphes que vous avez lu sur l'entrée standard. Vérifiez que votre programme fonctionne comme attendu.

Réponse :

```
void Graph::displayStructure (std::ostream &out)
{
  adjacency_list_t::iterator e;
  for (uint v = 0; v < edges_.size(); ++v) {
    out << "|- " << v << "\n";
    out << "| ";
    for (e = edges_[v].begin(); e != edges_[v].end(); ++e) {
      out << "`--(" << e->second << ")--> " << e->first << "\n";
      out << "| ";
    }
    out << "\n";
  }
}
```

Exercice 2.

Une question pertinente qu'on peut se poser est "quel est le moyen le plus rapide d'aller d'une ville à une autre ?". Autrement dit, quand on dispose d'un graphe, "Étant donné un graphe orienté et pondéré, quel est le plus court chemin entre deux sommets choisis de ce graphe ?".

Par exemple, sur le graphe dessiné dans l'exercice 1, il y a deux chemins pour aller de 0 à 2 : $0 \to 3 \to 2$, ou $0 \to 1 \to 3 \to 2$. Si on tient compte de la pondération des arêtes, c'est le second chemin qui est le plus court ($6+3+13 = 22$ alors que $51 + 13 = 64$).

Le but de cet exercice est d'implémenter un algorithme de recherche de plus court chemin : l'algorithme de Dijkstra.

Le principe de cet algorithme est assez simple : il s'agit d'une construction par récurrence des plus courts chemins d'un sommet source vers tous les autres sommets du graphe. À la fin il n'y a plus qu'à remonter depuis le sommet cible le long des chemins calculés pour avoir le plus court chemin du sommet source au sommet cible.

- L'idée va être, en partant du sommet source, de construire progressivement l'ensemble des chemins les plus courts possibles vers les autres sommets du graphe.
- Mettons qu'on veuille aller du sommet s au sommet c.
- Au départ, on note qu'aucun sommet n'a été visité et qu'ils sont tous considérés comme étant à distance infinie de s : pour tous sommet v on a sa distance à s $d_v = \infty$.
- On commence par le plus court chemin partant de s qui soit connu (base de la récurrence), c'est à dire le chemin de s vers lui même : c'est le chemin vide de longueur 0.
- On met donc s avec pour distance $d_s = 0$ dans l'ensemble $\mathcal{V}$ des sommets à visiter.
- Tant que l'ensemble des sommets à visiter n'est pas vide :
  - On sélectionne le sommet v avec la plus petite distance d_v dans l'ensemble des sommets à visiter $\mathcal{V}$, et on le retire de l'ensemble $\mathcal{V}$.
  - Si on l'a déjà visité, on le jette et on recommence à l'étape précédente.
  - Pour chaque sommet w directement voisin de v dans le graphe :
    - On calcul la distance de s à w en prenant le plus cours chemin connu passant par v (par récurrence) : $d'_w = d_v + p$ où p est le poids de l'arête (v, w).
    - Si d'_w est plus petit que le d_w déjà connu, alors :
      - On met à jour $d_w = d'_w$.
      - On ajoute le sommet w à l'ensemble des sommets à visiter $\mathcal{V}$, avec pour distance la nouvelle valeur de d_w.
      - On marque que dans le chemin optimal pour aller de s à w, le sommet précédant w est v.
- Ensuite, on a plus qu'à remonter les plus courts chemins par les sommets précédent en partant c jusqu'à arriver à s pour avoir le plus court chemin de s à c.

L'algorithme de Dijkstra est décrit avec des exemples sur sa page Wikipédia : `https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`.

1. Dans l'ensemble des sommets à visiter on veut pouvoir ajouter un élément étiqueté par un poids, et récupérer l'élément de poids minimal.

→ Quel est le bon conteneur à utiliser pour cela ? Justifiez votre réponse.

Réponse :
Le mieux est d'utiliser un tas, c'est à dire une `priority_queue` dans la STL. Le coût amorti est logarithmique pour les deux opérations.
Les éléments de la `priority_queue` sont des sommets pondérés, c'est à dire la même chose que les éléments de nos listes d'adjacences.

```
struct compvert {
  bool operator () (const Graph::edge_t &a, const Graph::edge_t &b) const {
    return a.second < b.second;
  }
};
std::priority_queue<edge_t, std::vector<edge_t>, compvert> to_visit;
```

2. Les autres conteneurs dont vous aurez besoin (l'ensemble des distances d_v, les marqueurs pour savoir si un sommet a déjà été visité ou pas, et les précédents) seront mieux représentés par de simples tableaux.

→ Quel sera la longueur et le type de chacun ?

Réponse :

- distances : `new uint[edges_.size()];`
- précédents : `new edge_t[edges_.size()];`
- marqueurs de visite : `new bool[edges_.size()];`

3. → En hésitant pas à vous aidez de recherches sur le web, implémentez une méthode `getOptimalPath` dans la classe `Graph` qui prend en argument deux sommets et qui utilise l'algorithme de Dijkstra pour calculer et renvoyer le chemin le plus cours entre ces deux sous forme d'une liste d'adjacence (sauf qu'ici la liste représente un chemin plutôt qu'un ensemble de voisins).

Commentez bien votre code pour expliquer ce qui correspond à chaque étape de l'algorithme.

Réponse :

```
#define INFINITY INT_MAX

Graph::adjacency_list_t *
Graph::getOptimalPath (uint from, uint to)
{
  std::priority_queue<edge_t, std::vector<edge_t>, compvert> to_visit;
  uint *dist = new uint[edges_.size()];
  edge_t *prev = new edge_t[edges_.size()];
  bool *visited = new bool[edges_.size()];

  std::fill(dist, dist + edges_.size(), INFINITY);
  std::fill(visited, visited + edges_.size(), false);

  dist[from] = 0;
  to_visit.push(std::make_pair(from, 0));

  while (!to_visit.empty()) {
    edge_t next = to_visit.top();
    int vertex = next.first;
    to_visit.pop();

    if (visited[vertex]) continue;
    else visited[vertex] = true;

    for (adjacency_list_t::iterator e = edges_[vertex].begin();
         e != edges_[vertex].end();
         ++e) {
      uint d = dist[vertex] + e->second;
      if (dist[e->first] > d) {
        dist[e->first] = d;
        to_visit.push(std::make_pair(e->first, dist[e->first]));
        prev[e->first] = std::make_pair(vertex, e->second);
      }
    }
  }

  adjacency_list_t *path = new adjacency_list_t;
  while (to != from) {
    path->push_front(std::make_pair(to, prev[to].second));
    to = prev[to].first;
  }

  return path;
}
```