C++ : STL - suite

TP C++
STL

Eric Ramat
eric.ramat@univ-littoral.fr

22 novembre 2023

Durée : 6 heures

---

1 Objectifs

Le quatrième TP a pour objectif d'utiliser au maximum les classes et algorithmes offerts par la STL. Nous allons nous intéresser aux algorithmes de graphe et en particulier, à la recherche de chemins sur une carte routière. Dans les TP précédents, nous avons développé un simulateur de traffic routier et nous l'avons testé sur une petite portion de carte. Dans ce nouveau TP, on va travailler avec l'ensemble des routes et des carrefours de l'agglomération de Calais.

2 Travail demandé

Avant de commencer le TP, il est nécessaire de forker un nouveau dépôt `git` contenant le projet avec un début de code. Vous devez voir apparaître dans votre compte gitlab (gitlab.dpt-info.univ-littoral.fr) le projet "traffic-calais". Le dépôt contient trois classes :

— `Graph` : le graphe qui est composé d'une liste de sommets et d'arcs
— `Edge` : les arcs du graphe avec un identifiant, une longueur (en mètres), le sommet source et le sommet destination
— `Vertex` : les sommets avec un identifiant, un type, la liste des arcs entrants et la liste des arcs sortants

Les identifiants sont issus d'OpenStreetMap, un projet libre de cartographie (https://www.openstreetmap.fr/) et le fichier de données est produit à l'aide de scripts Python et d'extractions obtenues avec l'API overpass-turbo (https://overpass-turbo.eu/). Il y a trois types de sommets :
— INPUT : sommet d'entrée avec uniquement des arcs sortants où les véhicules sont injectés dans le graphe
— OUTPUT : sommet de sortie avec uniquement des arcs entrants où les véhicules sortent du graphe
— INTERNAL : sommet qui possède à la fois des arcs entrants et sortants

Les rondpoints ne sont pas représentés en tant que tel. Un rondpoint est un ensemble d'arcs et de sommets où les sommets sont les points d'accès au rondpoint.

Question 1. Développer la méthode `search_shortest_path` en traduisant l'algorithme suivant. Cet algorithme recherche le chemin dont le coût est minimal. Dans notre contexte, le coût est la longueur d'un arc (c'est à dire d'un tronçon).

```
search_shortest_path(source, destination)
  score[i] = infini pour tout i tel que i est un sommet
  from[i] = plus proche voisin connu pour tout i tel que i est un sommet
  queue = liste des sommets en cours de traitement;

  score[source] = 0
  ajouter source dans queue
  tant que (queue n est pas vide) faire :
    courant est le premier élement de queue
    retirer le premier élément de queue
   si courant = destination alors :
     chemin = liste vide;
     tant que courant est différent de source faire :
       placer courant dans chemin
       courant = from de courant
     fin tant que
     placer source dans chemin
     inverser les éléments de chemin (étape optionnelle selon le type de chemin)
     retourner chemin
   sinon
     successeurs = les sommets entrants des arcs sortants de courant
     pour chaque successeur v de successeurs faire :
       si score[v] = infini ou score[v] + poids de l arc entre courant et v <
    score[v] alors :
            from[v] = current
            score[v] = score[current] + poids de l arc entre courant
            placer v dans queue
      fin si
     fin pour
   fin si
```

```
 fin tant que
retourner chemin vide
```

Dans la fonction `main`, ajouter le code ci-dessous pour tester votre code.

```
auto path = graph.search_shortest_path("generator/507", "end/366");

for (const auto &e:path) {
   std::cout << e << "␣";
}
std::cout << std::endl;
```

Vous devez obtenir :

```
generator/507 junction/2641 junction/2409 junction/2411 junction/2642
junction/1847 junction/2406 junction/2353 junction/964 junction/1674
junction/1675 junction/1673 junction/2522 junction/2521 junction/2518
junction/2519 junction/356 junction/354 junction/1676 junction/1677
junction/355 junction/2613 junction/2615 junction/711 junction/2618
junction/2619 junction/2620 junction/2621 junction/2622 junction/2164
junction/712 junction/713 junction/714 junction/1253 junction/1627 junction/2459
junction/2458 junction/341 junction/2464 junction/1138 junction/1141 junction/2465
junction/1139 junction/1136 junction/1638 junction/358 junction/359 junction/363
junction/367 junction/366 junction/1640 junction/1639 junction/342 junction/371
junction/370 junction/55 junction/1413 junction/1414 junction/1429 junction/1643
junction/858 junction/1642 junction/1641 junction/373 junction/868 junction/2177
junction/865 junction/867 end/366
```

Question 2. "Connecter" le code des TP précédents à ce nouveau code. La classe `Vertex` doit être la super-classe `Node` et la classe `Link` est la sous-classe de `Edge`. Il y a probablement des adaptations à faire.

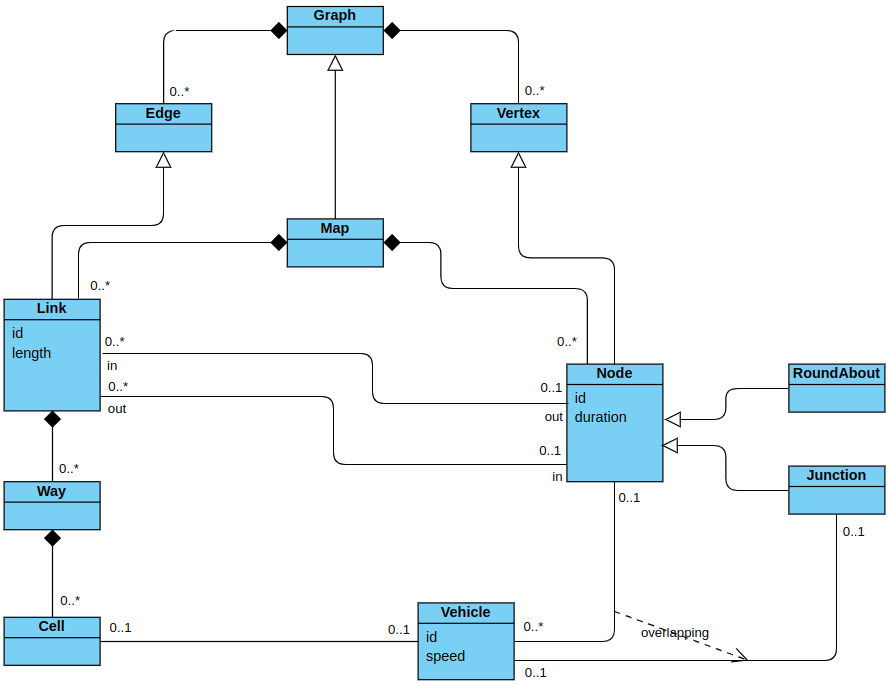

Diagramme de classes

Question 3. Écrire une nouvelle classe `Generator`. Son rôle est de créer aléatoirement des véhicules et de les injecter dans le réseau. Pour chaque nouveau véhicule, il faut tirer au hasard un sommets d'entrée et un sommet de sortie, puis calculer le chemin et placer le véhicule dans le n ?ud d'entrée. Tous les sommets d'entrée possèdent la même probabilité d'être utilisé. Toutes les secondes, on génère un véhicule.

Un exemple d'utilisation des générateurs aléatoires.

```
#include <iostream>
#include <random>

int main()
{
    std::mt19937 gen(772635); // engine with a seed
    std::uniform_int_distribution<> distrib(1, 10);

    // 20 integers generated into an int in [1, 6]
    for (int n = 0; n < 20; ++n) {
      std::cout << distrib(gen) << "␣";
     }
    std::cout << std::endl;
}
```

Question 4. Que faut-il ajouter pour connaître le temps min, max et moyen de présence des véhicules dans le réseau pour une durée de simulation donnée ?