Automne 2017

Algorithmes et structures de données pour ingénieurs

GLO-2100

Travail à faire individuellement
À rendre avant la date indiquée
sur le portail du cours
(Voir modalités de remise à la fin de l'énoncé)

Tout travail remis constitue une contribution originale et distincte des travaux remis par d'autres. Le plagiat est strictement défendu. Tout travail plagié sera transmis au Commissaire aux infractions relatives aux études de l'Université Laval et sera donc passible de sanctions très punitives. De plus, les étudiants ou étudiantes ayant collaboré au plagiat seront soumis aux sanctions normalement prévues à cet effet par les règlements de l'Université.

Travail Pratique #2

Algorithmes sur graphes pour trouver les itinéraires de voyage sur le réseau de la RTC

UNIVERSITÉ LAVAL
Faculté des sciences et de génie
Département d'informatique
et de génie logiciel

1 Objectif général

Ce TP constitue le deuxième TP d'une séquence de 3 TPs portant sur les données du réseau de transport de la capitale (RTC) de la ville de Québec. Au TP1, vous deviez implémenter certaines méthodes de la classe DonneesGTFS vous permettant de gérer l'accès aux données GTFS de la RTC. Pour ce TP2, vous devez implémenter certaines méthodes de la classe **ReseauGTFS** vous permettant de construire un graphe sur lequel vous pourrez exécuter un algorithme de plus court chemin vous donnant le meilleur itinéraire d'un voyage utilisant le réseau d'autobus de la RTC, quelque soient les points d'origine et de destination fournis. Pour y arriver, ce TP utilisera la classe DonneesGTFS que vous étiez sensé compléter au TP1. Si vous n'avez pas réussi votre TP1, vous pouvez utiliser la librairie statique libTP1.a fournie avec cet énoncé qui implémente toutes les classes du TP1, y compris DonneesGTFS. En fait, nous recommandons à tous d'utiliser libTP1.a au cas où votre TP1 comporterait certaines erreurs n'ayant pas été détectées lors de la correction. Par contre, notez que libTP1.a ne s'utilise que dans la machine virtuelle du cours. Pour utiliser libTP1.a, vous devez utiliser tous les fichiers .h fournis avec cet énoncé et utiliser le CMakeLists.txt fourni avec cet énoncé pour la compilation avec CMake.

2 Les classes fournies

En plus de toutes les classes du TP1, nous vous fournissons la totalité de la classe **Graphe**. Dans ce TP, on vous demande de ne pas modifier cette classe, mais vous devrez l'utiliser dans les méthodes de ReseauGTFS que l'on vous demande d'implémenter.

La classe fournie **Graphe** est une implémentation très simple (mais efficace) d'un graphe à l'aide de listes d'adjacence. Elle est constituée d'une sous-classe privée Arc (qui est simplement une paire constituée du noeud destination et du poids de l'arc). Cette classe possède un seul attribut membre m_listesAdj qui est un vecteur de listes d'arcs. Les sommets du graphes vont de 0 à n-1 (n étant le nombre de sommets du graphe) et constituent les indexes du vecteur m_listesAdj. Dans cette classe, vous y trouverez uniquement les méthodes que vous aurez besoin pour votre TP, par exemple :

- Le constructeur Graphe(size_t n) vous permettant de créer un graphe de n listes d'adjacences vides.
- La méthode ajouterArc(size_t i, size_t, j, unsigned int poids) permettant d'ajouter un arc au graphe.
- La méthode enleverArc(size_t, i, size_t j) permettant d'enlever l'arc (i,j).
- La méthode resize(size_t p) permettant de modifier la taille du graphe afin qu'il contienne p nœuds.
- La méthode plusCourtChemin(p_origine, p_destination, p_chemin) qui est une implémentation simple et naïve de l'algorithme de Dijkstra vous permettant d'obtenir le plus court chemin du nœud p_origine au nœud p_destination.

Pour les autres méthodes et plus de détails, voir l'implémentation de Graphe.

La classe **ReseauGTFS** vous permet de construire un objet de type Graphe à partir d'un objet DonneesGTFS, de compléter ce graphe en fonction du point origine et du point destination (qui sont décrits par leurs coordonnées GPS) choisis et de trouver un itinéraire sur le réseau de la RTC en exécutant l'algorithme de plus court chemin de Graphe.

Le constructeur de ReseauGTFS (fourni) construit l'attribut membre m_leGraphe à l'aide d'un objet du type DonneesGTFS. Pour réaliser cette tâche, il utilise 3 méthodes privées que vous devez implémenter.

Ce TP a été conçu afin d'obtenir les estimations les plus réalistes possibles des temps de trajets pour aller d'un point d'origine à un point destination quelconque choisi par un utilisateur. Durant un trajet, il arrive régulièrement que l'on doive débarquer à une station et attendre pour prendre un autre bus allant dans une autre direction. Toute estimation réaliste doit donc prendre en compte ces temps d'attente qui représentent souvent une fraction significative du temps du trajet. Pour cette raison, **nous avons choisi d'utiliser les arrêts (et non les stations) pour représenter les nœuds du graphe** de m_leGraphe. Rappelez-vous qu'un arrêt représente un évènement spatiotemporel : un arrêt fait parti d'un voyage et s'effectue à une station et à un temps précis (donnée par son heure d'arrivée). Ainsi, chaque arrêt dans l'objet DonnesGTFS possèdera généralement un arc (de type « voyage ») vers le prochain arrêt du même voyage et un arc (de type « temporel » ou «d'attente») vers le prochain arrêt (d'un autre voyage) s'effectuant à la même station. Le poids de cet arc sera donné par le temps, en secondes, du délai de cette composante élémentaire du trajet. Le constructeur de ReseauGTFS devra construire le graphe en associant un nœud par arrêt et en créant tous les arcs de type « voyage » et de type « temporel » reliant les arrêts. Vous verrez que le graphe ainsi obtenu pour un objet DonneesGTFS contenant tous les voyages d'une journée contiendra généralement plus de 100000 nœuds!

Pour que les méthodes de ReseauGTFS puissent accomplir leurs tâches, elles devront utiliser les attributs membres m_arretDuSommet et m_sommetDeArret. L'attribut m_arretDuSommet est un vecteur de Arret::Ptr, donc m_arretDuSommet[i] nous donne le shared_ptr de l'arrêt représenté par le sommet i de m_leGraphe. Inversement m_sommetDeArret est un unordered_map<Arret::Ptr, size_t> de paires (clés, valeurs). Chaque clé est un Arret::Ptr et la valeur associée est un numéro de sommet. Donc, si a_ptr est un shared_ptr à un Arret, m_sommetDeArret[a_ptr] nous donne le numéro du sommet de m_leGraphe associé à cet arrêt. Nous utilisons un unordered_map car cet accès s'obtient presque toujours en temps O(1). Donc, à l'aide de ces deux conteneurs, nous pouvons obtenir, très efficacement, l'arrêt associé à un numéro de sommet et, inversement, le numéro de sommet associé à un arrêt. Ces deux conteneurs seront remplis par la méthode ajouterArcsVoyages (qui est appelée par le constructeur de ReseauGTFS) durant la construction du graphe m_leGraphe.

L'autre conteneur, qui est membre privé de ReseauGTFS, est le vecteur m_sommetsVersDestination qui contient les sommets possédant un arc vers le point destination. Ces sommets sont associés à des arrêts de stations où chacune de ces stations possède la propriété d'être à distance de marche du point destination. Le membre privé constant distanceMaxMarche fixe cette distance maximale de marche à 1.5 km. Donc toute station se trouvant, à « vol d'oiseau », à moins de 1.5 km du point origine pourra être utilisé et faire parti du trajet pour aller au point destination. L'utilisateur aura donc la possibilité de débarquer à n'importe quelle station se trouvant à moins de 1.5 km du point destination pour aller rejoindre à pieds le point destination. Le temps, en secondes, de ces trajets à pieds est donné par la distance parcourue (à vol d'oiseau) en km divisée par le vitesse de marche (donnée par l'attribut constant, vitesseDeMarche) fixée à 5 km/sec.

Nous vous fournissons la méthode ReseauGTFS::itineraire vous permettant d'obtenir l'itinéraire du voyage lorsque le graphe complet (incluant les points origine et destination) m_leGraphe aura été construit. Cette méthode utilise la méthode Graphe::plusCourtChemin pour obtenir le chemin de l'itinéraire et l'affiche à l'écran sous forme conviviale pour l'utilisateur. La méthode ReseauGTFS::itineraire retourne également le temps d'exécution, en microsecondes, de Graphe::plusCourtChemin. Au TP3, vous aurez à fournir un algorithme de plus court chemin qui sera substantiellement plus efficace que celui présentement fourni.

3 Travail à faire

Vous devez implémenter les méthodes suivantes de la classe **ReseauGTFS** :

- **La méthode ajouterArcsVoyages :** cette méthode privée, appelée par le constructeur de ReseauGTFS, ajoute dans m_leGraphe les arcs de type « voyage » et insère les arrêts (associés aux sommets) dans m_arretDuSommet et m_sommetDeArret. Le poids de chaque arc est la différence, en secondes, entre les heures d'arrivées des arrêts successifs.

- **La méthode ajouterArcsAttentes :** cette méthode privée, appelée par le constructeur de ReseauGTFS, ajoute dans m_leGraphe les arcs de type « temporel » (ou « attente »). L'intervalle de temps donnant le poids de l'arc est la différence, en secondes, entre les heures d'arrivées des arrêts successifs à la même station. Les conteneurs m_arretDuSommet et m_sommetDeArret ne sont pas modifiés par cette méthode car les arrêts ont déjà été ajoutés par la méthode ajouterArcsVoyages.

- **La méthode ajouterArcsTransferts :** cette méthode privée, appelée par le constructeur de ReseauGTFS, ajoute dans m_leGraphe les arcs dus aux transferts (ceux se trouvant dans l'objet DonnesGTFS) entre les stations. L'intervalle de temps donnant le poids de l'arc est donné par le temps de transfert entre les deux stations plus le temps d'attente nécessaire pour aboutir au prochain arrêt à la station d'arrivée (i.e., il faut prendre en compte le temps d'attente pour l'arrivée du prochain bus). Indice : pour tenir en compte ce temps d'attente, pensez à utiliser la méthode lower_bound() des conteneurs map et multimap. Chaque arc relie un arrêt de la station source à l'arrêt de la station destination qui est le premier à se réaliser après le temps de transfert.

- **La méthode ajouterArcsOrigineDestination :** cette méthode publique finalise la construction du graphe en ajoutant deux nœuds au graphe : l'un pour le point origine, l'autre pour le point destination et en ajoutant les arcs reliant ces deux nœuds au reste du graphe. Puisque l'on doit associer un arrêt par nœud du graphe, vous devez alors créer, à l'aide du constructeur de la classe Arret, un arrêt pour le point origine et un arrêt pour le point destination et mettre à jour les conteneurs m_arretDuSommet et m_sommetdeArret. Il faudra donc également créer un Arret::Ptr pour ces deux points GPS. Notez que pour créer un objet Arret, vous devez fournir un identificateur de station (station_id). Pour cela, utilisez les membres constants stationIdOrigine et stationIdDestination de ReseauGTFS. Vous devrez également fournir deux objets Heure pour chaque Arret et un identificateur de voyage (voyage_id). Utilisez des valeurs quelconques pour ces objets car vous n'aurez pas besoin de les utiliser. Vous devrez assigner à l'attribut m_sommetOrigine la valeur du numéro de sommet utilisé pour représenter le point d'origine et vous devrez assigner à m_sommetDestination la valeur du numéro de sommet assigné au point destination. Ces attributs vous seront utiles par la suite pour détruire partiellement le graphe. Puisque vous ajoutez deux sommets au graphe, vous devrez alors utiliser la méthode Graphe::resize afin d'ajouter deux listes d'adjacence additionnelles dans m_leGraphe. Ces listes seront remplies par les deux étapes consécutives suivantes de ajouterArcsOrigineDestination :

    - *Ajouts des arcs entre le point origine et les stations atteignables à pieds*. C'est l'attribut constant distanceMaxMarche qui détermine si une station est atteignable ou non. On suppose un trajet à pieds rectiligne direct (« à vol d'oiseau ») entre une paire de points. Le poids d'un arc représentant le trajet du point origine vers un arrêt d'une station atteignable est donné par le temps, en secondes, du parcours à pieds plus le temps d'attente (en secondes) pour que l'arrêt se réalise. Pensez donc à utiliser la

méthode lower_bound() des conteneurs map et multimap à cette fin. C'est simple et très efficace. Pour chaque station atteignable, vous devrez alors avoir un arc allant du point origine vers le premier arrêt de cette station qui se réalise après l'arrivée à pieds à la station. On ajoute donc un seul arc par station atteignable.

- *Ajouts des arcs entre les arrêts de certaines stations et le point destination*. Encore une fois, les stations concernées sont celles qui se trouvent à distance de marche du point destination. À cette étape, il faut ajouter un arc pour chaque arrêt de ces stations et le poids de chaque arc est donné uniquement par le temps, en secondes, du parcours à pieds pour se rendre au point destination. Contrairement au cas précédent, on a maintenant un arc par arrêt de station vers le point destination. Il y a donc généralement beaucoup plus d'arcs allant vers le point destination que d'arcs émergeant du point d'origine. Finalement, tout numéro de sommet de m_leGraphe que l'on connecte au sommet destination devra être insérer dans l'attribut m_sommetsVersDestination pour que la méthode suivante puisse s'exécuter efficacement.

- **La méthode enleverArcsOrigineDestination :** Cette méthode publique remet l'objet ReseauGTFS dans l'état qu'il était avant l'exécution de ajouterArcsOrigineDestination. Conséquemment, on doit donc enlever tous les arcs émergeant du point origine et tous les arcs aboutissant au point destination. Par souci d'efficacité, il est essentiel de faire cette opération **sans** parcourir tout le graphe. C'est pour cette raison que vous avez construit le conteneur m_sommetsVersDestination lors de l'exécution de ajouterArcsOrigineDestination. Donc, la première chose à faire sera de parcourir m_sommetsVersDestination et d'enlever, de m_leGraphe, tous les arcs émergeant de ces sommets et aboutissant au sommet m_sommetDestination à l'aide de la méthode Graphe::enleverArc. Vous devrez par la suite enlever de m_leGraphe les deux dernières listes d'adjacence à l'aide de Graphe::resize (ceci n'enlèvera que les listes d'adjacence pour le sommet origine et le sommet destination sans modifier le reste du graphe) et remettre à jour les attributs m_sommetDeArret, m_arretDuSommet, m_nbArcsStationsVersDestination, m_nbArcsOrigineVersStations, et m_origine_destination_ajoute. Cette méthode vous sera très utile au TP3 où vous aurez à tester votre algorithme de plus court chemin pour plusieurs paires (origine, destination) choisies aléatoirement.

Si vous avez implémenté ces méthodes correctement, la sortie de votre main() devrait être identique à ce que vous trouvez dans le fichier out.txt fourni avec cet énoncé. En modifiant légèrement votre main() pour appeler le constructeur de Date et celui de Heure sans paramètre, vous pourrez générer un itinéraire pour le moment présent pour une paire (origine, destination) que vous aurez choisie. Comparez l'itinéraire obtenu avec celui donné par Google (notez que vous pouvez obtenir les coordonnées GPS d'un point sur Google Maps en double-cliquant sur une position et en choisissant « plus d'info sur cet endroit »). Vous observerez que l'itinéraire donné par votre programme est généralement plus rapide que celui donné par Google. Cela peut possiblement être attribué aux causes suivantes :

- Google utilise probablement les données du fichier transfers.txt et, comme vous le savez, un temps de transfert est prévu pour les transferts d'une station à elle même; ce que nous n'utilisons pas. Dans ce cas, notre algorithme peut choisir de débarquer à une station et de prendre le prochain bus même si celui-ci est prévu passer 30 secondes après notre arrêt. Si le temps de transfert prévu pour cette station à elle-même est de 3 minutes, Google choisira de ne pas effectuer ce transfert trop serré (car moins de 3 minutes) et produira un itinéraire plus long que celui trouvé par notre algorithme (qui est, en fait, plus risqué).

- Nos parcours à pieds s'effectuent à « vol d'oiseau ». Ils s'effectuent donc plus rapidement que les parcours proposés par Google qui empruntent les rues existantes.

Important : notez que vous nous fournissons le fichier CMakeLists.txt pour votre projet. Dans ce cas, les exécutables et les librairies statiques produites se trouvent dans le même répertoire que les fichiers sources. Le répertoire build (ou le répertoire cmake-build-debug lorsque l'on utilise CLion) se trouve également dans ce répertoire. C'est ce CMakeLists.txt qu'utiliseront les correcteurs pour corriger votre TP sur la machine virtuelle du cours. **Vous devez donc vous assurer que votre programme compile (et s'exécute) sans erreur sur la machine virtuelle du cours avec le CMakeLists.txt fourni.**

4 Questions ?

Si vous avez des questions pertinentes à propos de ce TP, nous vous demandons de les poser sur le forum des TPs qui se trouve sur le portail du cours. Comme cela, la réponse sera visible à toute la classe, et non seulement à un seul individu. Toute question envoyée directement au professeur par courriel sera laissé sans réponse afin d'éviter de donner de l'information pertinente à un seul individu. Essayez de ne pas multiplier vos questions sur le forum et choisissez-les judicieusement. Finalement, tout TP de cette nature requiert de votre part une certaine dose de recherche à effectuer par soi-même afin de trouver la façon d'implémenter ce qui est demandé.

5 Fichiers à remettre

Vous devez remettre uniquement les fichiers ReseauGTFS.h et ReseauGTFS.cpp que vous devez insérer dans une archive ayant pour nom NomDeFamille_Prénom.zip. SVP, ne remettre aucun autre fichier. Vous devez remettre votre travail dans la boite de dépôt du portail du cours. Aucune remise par courriel ne sera acceptée. Tout travail remis en retard perdra 3 points par heure de retard. Donc, un retard de 1 seconde à 3599 secondes occasionne la perte de 3 points, et après un retard de plus de 33 heures, votre note tombera automatiquement à zéro. Vous pouvez remettre autant de versions que vous le désirez. Seul le dernier dépôt sera corrigé.

ATTENTION : vous avez la responsabilité de vérifier l'intégrité des fichiers que vous avez remis dans la boite de dépôt. Donc, nous vous demandons de vérifier ce que vous avez remis (en téléchargeant sur votre poste ce que vous avez téléversé dans la boite de dépôt) afin d'en vérifier l'intégrité. Le mécanisme de téléversement du Portail ne corrompt pas les fichiers. Cependant, il peut arriver que votre archive ait été corrompue si, lorsque vous l'avez créée, certains des fichiers étaient ouverts par d'autres applications (Eclipse ou CLion par exemple...). Vérifiez donc l'intégrité de votre archive après l'avoir construite!

Tout fichier remis qui est inutilisable sera considéré comme un fichier non remis.

6 Critères de correction

- L'exactitude de chaque méthode que vous devez implémenter vaut la grande majorité des points (environ 80%). En gros, vos méthodes sont sensées faire ce qui est demandé.
- La clarté du code des méthodes implémentées vaut le reste des points (environ 20%).
- Une version de ReseauGFTS.h et ReseauGTFS.cpp qui, avec les modules fournis, ne compile pas sur la machine virtuelle du cours avec le CMakeLists.txt fourni se verra attribuée la note de zéro (c'est inacceptable de remettre un programme qui ne compile pas).
- Si le programme résultant de cette compilation plante lors de son exécution, une note de zéro se verra attribuée au travail (c'est inacceptable de remettre un programme qui plante).

7 Plagiat

Tel que décrit dans le plan de cours, le plagiat est interdit. Une politique stricte de tolérance zéro est appliquée en tout temps et sous toutes circonstances. Tous les cas seront référés à la direction de la Faculté.