Vincent Breault (111 226 718) 2019/12/07
GLO-2100 Département d'informatique et de génie logiciel

Section 1 Travail à faire:

La charge de travail de ce TP consiste à effectuer une recherche afin d'intégrer un algorithme de plus court chemin le plus efficace possible qui est valide peu importe l'instance en l'appliquant au réseau de bus du RTC dans la méthode plusCourtChemin() issue de la classe Graphe. Ce travail requiert de bien comprendre l'application d'un cas réel et de bien étudier la structure de ce réseau implémenté sous forme de graphe pour pouvoir appliquer les connaissances théoriques du cours à une vraie situation pratique. Un algorithme est déjà fourni et l'objectif est de trouver un algorithme qui a un facteur d'accélération d'au moins 10. La méthodologie utilisée pour effectuer le travail sera la suivante : 1. Analyser et comprendre le réseau de transport du RTC (graphe) 2. Effectuer une recherche d'algorithmes de plus courts chemins qui s'appliquent à la situation (algorithmes présentés dans le cours ou autres) et distinguer ceux qui semblent les plus efficaces tout en s'assurant qu'ils sont valides 3. Implémenter les algorithmes distingués et les tester sous différentes formes pour trouver la meilleure implémentation et trouver le plus efficace 4. Présenter les résultats de cet algorithme implémenté et des divers algorithmes testés tout en prenant soin de choisir le plus efficace.

Section 2 Observations et algorithmes envisagés:

Le graphe de ReseauGTFS est un graphe dirigé peu dense avec 344253 arcs et 119402 sommets qui sont tous les arrêts possibles plus le point d'origine et d'arrivée, ainsi avec la formule qui détermine la densité d'un graphe orienté D=|E|/(|V|(|V|-1)) où E est le nombre d'arcs et V le nombre de noeuds on obtient une densité de 0.000023405. Ceci représente une densité très faible comme la densité maximale est 1 où un graphe avec 1 de densité a des arcs qui relient tous les sommets ensemble et un graphe avec 0 de densité n'a aucun arc. Le graphe ne possède aucun cycle comme les noeuds sont des arrêts, et les arrêts sont temporels, il est donc impossible de retourner dans le temps en faisant un trajet d'autobus. Le graphe ne contient aucun poids négatif comme le temps pour se rendre d'un arrêt à l'autre ne peut pas être négatif, mais peut être nul. Avec ces observations deux algorithmes sont considérés, le premier est Dijkstra implémenté avec un tas minimum, et le deuxième est l'algorithme de Bellman-Ford pour les graphes orientés acycliques qui utilisent le tri topologique.

Bellman-Ford pour les graphes orientés acycliques:

L'algorithme de Bellman-Ford qui utilise le tri topologique est un choix judicieux, comme son temps d'exécution est moindre à celui de Dijkstra sur papier soit O(n + m). Cet algorithme fait usage d'une propriété de notre graphe, soit qu'il est acyclique et permet de réduire le temps d'exécution. En effectuant un tri topologique (m itérations) du graphe avant d'effectuer la boucle principale, celle-ci a besoin de boucler une seule fois comme les noeuds sont positionnés dans un ordre logique (n itérations). Cet algorithme est valide pour tous les graphes dirigés, acycliques et n'ayant pas de poids négatifs, car cet algorithme relâche les noeuds en ordre topologique étant donné du tri précédant la boucle principale.

Dijsktra avec tas minimum:

Une autre approche pour améliorer l'algorithme de Dijkstra avec listes d'adjacence déjà en place est d'ajouter un tas minimum (priority queue) et ajouter le noeud ayant la plus courte distance à parcourir sur le dessus du tas à chaque relâchement. Comme aucune méthode n'est disponible pour accéder à un noeud dans le tas, il est donc nécessaire d'ajouter une copie de plus du noeud présent avec la distance minimale. Ceci règle le problème de l'accès à un noeud dans un tas, mais introduit des copies de noeud dans le tas ce qui n'a pas d'impact sur la validité, car on considère seulement le noeud avec la distance minimale, en ignorant les copies. Cet algorithme est valide si, on ajoute la contrainte des noeuds visités soit un vecteur qui retient un booléen de la visite d'un noeud, ainsi il est impossible de revisiter un noeud ou une copie d'un noeud et de le rajouter dans le tas min.De plus, l'algorithme met à jour les distances et les noeuds solutionnés à chaque itération (relâchement) ce qui prouve sa validité. En implémentant la priority queue avec Dijkstra on peut réduire le temps d'exécution de $O(n^2)$ à $O((m+n)\log n)$.

Autres:

Les autres algorithmes ont été rejeté sans être testés, car leur temps d'exécution ne serait pas bénéfique comme le but du travail est d'optimiser le temps d'exécution de l'algorithme de plus court chemin. Bellman-Ford traditionnel n'est pas un choix judicieux comme son exécution est en $O(nm)$ ce qui est supérieur à l'algorithme de Dijkstra déjà implémenté en ce moment (comme $m>>n$). L'algorithme de Floyd-Warshall en $O(n^3)$ n'apporte pas un gain au niveau du temps d'exécution non plus. Le dernier algorithme considéré, mais qui ne peut pas être implémenté est A*, comme on n'a aucune heuristique valide, car certains arcs ont un poids de 0.

Choix:

L'algorithme choisi pour l'implémentation finale est l'algorithme de Dijkstra avec un tas min, comme sur papier l'algorithme de Bellman-Ford pour les graphes orientés acycliques semble plus efficace, mais comme le m et n n'est pas énorme, et après l'implémentation des deux et plusieurs tests l'algorithme de Dijkstra est plus efficace.

Section 3 Étapes algorithmiques:

Au début de la démarche, il fallait discriminer entre lesquels des deux algorithmes sélectionnés était le plus efficace en les implémentant et les testant. Comme sur papier Bellman-Ford avec le tri semblait le plus prometteur, ce fut le premier à être implémenté et testé. En premier, il fallait implémenter le tri topologique, en passant en paramètre : le vecteur des noeuds visités, l'index du noeud courant et le stack utilisé pour la boucle principale de l'algorithme. Ensuite, la boucle était très simple à implémenter en effectuant le relâchement sur chaque arc qui minimise la distance. Après avoir testé cet algorithme, il était clair qu'il ne fournirait pas un facteur d'accélération assez grand. Ainsi, il a fallu implémenter le tas min dans l'algorithme de Dijkstra

qui serait le choix de l'algorithme final. La STL : std::priority_queue<size_t, vector <pair<size_t, size_t >>, greater<pair < size_t, size_t >>> a permis d'implémenter le tas min.. En implémentant cet algorithme, il a fallu modifier légèrement celui du professeur en modifiant le relâchement pour ajouter le noeud relâché sur le dessus du tas. Un problème rencontré lors de l'implémentation fut que les noeuds visités n'étaient pas pris en compte au tout départ, ce qui causait une redondance au niveau de l'algorithme qui revisitait les noeuds copiés dans le tas, ne relâchant pas bien les noeuds et donnant le même temps d'exécution moyen que l'algorithme de base. Après avoir bien implémenté le vecteur booléen de visite, le facteur d'accélération a augmenté de façon significative, ce qui représentait le succès de l'implémentation. Dans les deux algorithmes implémentés il a fallu garder les deux boucles while à la fin de l'algorithme de base et de conserver un vecteur de prédécesseurs, car c'est ce vecteur et ces deux boucles qui permettent d'éventuellement reconstruire le chemin et les stocker dans p_chemin.

Section 4 Résultats:

Voici les temps d'exécution moyens pour chaque algorithme implémenté et testé plusieurs fois:

Algorithme de base (10000 itérations) : 14558 microsecondes

Bellman-Ford avec tri topologique (10000 itérations): 12861 microsecondes

Dijkstra avec tas min (10000 itérations): 805.072 microsecondes

Il est donc à noter que l'algorithme de Dijkstra avec tas min procure un facteur d'accélération de 18.08, tandis que l'algorithme de Bellman-Ford avec tri topologique procure 1.1319 comme facteur. Ceci est explicable par le fait que le graphe est peu dense et l'algorithme de Dijkstra avec tas min est avantageux pour ce type de graphe même si sur papier l'algorithme de Bellman-Ford semble meilleur.

Section 5 Conclusion :

En conclusion, ce travail a permis d'initier l'élève à un travail de recherche d'algorithme de plus court chemin pour une situation réelle qui a comme optique l'optimisation. De plus, il supporte le fait que sur papier un choix peut être meilleur qu'un autre, mais il n'est pas nécessairement le meilleur en pratique. Il serait intéressant de considérer une mémoire pour l'algorithme de plusCourtChemin() dans l'objet Graphe, soit qu'après chaque itération la classe Graphe garde en mémoire le vecteur de distance et de prédécesseurs, ainsi après chaque itération subséquente, le temps d'exécution serait en temps constant ou linéaire comme les plus courts chemins seraient tous déjà définis par ces vecteurs. Comme Dijkstra trouve le plus court chemin pour tous les noeuds du graphe, cette approche serait valide.