Slim Ben Yahia

111 150 230

Alex Gilbert

111 152 662

Algorithme et structures de données pour ingénieur
GLO-2100

Livrable 3

Département d'informatique et de génie logiciel
Université Laval
(Automne 2016)

Maintenant que nous avons une façon efficace de traiter les données du RTC, il faut maintenant trouver une façon rapide de déterminer quel est le meilleur chemin à prendre pour aller d'un point A à un point B en utilisant ces données. Au travail pratique 2, nous avions programmé plusieurs algorithmes pour effectuer cette tâche, notamment Dijsktra et Bellman-Ford. Comme ces algorithmes étaient en $O(n^2)$ et $O(nm)$ respectivement pour un graphe de n sommets et m arcs, ces algorithmes étaient très lents pour traiter notre graphe qui contient un très grand nombre de sommets et d'arcs, respectivement 4604 et 19322. Maintenant, il faudrait trouver des algorithmes plus rapides pour effectuer cette même tâche, de façon à rendre le programme plus rapide et donc beaucoup plus utilisable.

Durant notre recherche, nous avons investigué plusieurs algorithmes. Le premier était l'algorithme de Bellman-Ford pour graphe acyclique avec tri topologique. Cet algorithme est très rapide, de par son temps d'éxécution en $O(n+m)$, donc si on arrivait à l'implementer, cela accélèrerais beaucoup notre recherche du plus court chemin. Par contre, un des prérequis à l'utilisation de cet algorithme est que le graphe soit acyclique, ce qui n'est pas le cas de notre réseau du RTC. Il existe énormément de cycles de toutes tailles dans notre graphe, majoritairement parce que tous les arrêts qui sont suffisamment proches ont un arc de déplacement à pied dans les deux sens entre les deux, ce qui crée des dizaines de cycles instantanément. Cet algorithme n'est donc pas utilisable.

Un autre algorithme que nous avons vu durant les cours est l'algorithme de FLoyd-Warshall. Cet algorithme trouve le chemin le plus court entre toutes les paires de sommets du graphe, au lieu d'une seule paire, ce qui le rend peu adapté à nos besoins. Puisqu'il cherche beaucoup plus d'informations que ce dont nous avons besoin, il est aussi très lent. Floyd-Warshall s'éxécute en $O(n^3)$, ce qui est pire que nos algorithmes courant. Nous avons donc rapidement abandonné cette piste.

Le prochain algorithme sur notre liste est l'algorithme de Dijsktra utilisant un monceau pour les noeuds non-solutionnés. Cet algorithme s'éxécute en $O((n+m)\ Log(n))$, ce qui est bien plus rapide que nos deux algorithmes. De plus, cet algorithme n'a pas de spécification particulière sur la nature du graphe sur lequel on l'utilise, donc il est compatible avec notre graphe.

Le dernier algorithme que nous avons investigué est l'algorithme A* (A-star). A* est un algorithme particulier. Il dépend d'une fonction heuristique qui donne une estimation du temps restant pour atteindre le but à partir d'un noeud donné. A* se fie ensuite à cette

heuristique pour évaluer le graphe intelligemment. En pire cas, avec une très mauvaise heuristique et une grande malchance, A* s'éxécutera en $O(n^2)$. Par contre, dans tous les autres cas, A* s'éxécutera beaucoup plus rapidement. Si on peut prouver que l'heuristique est admissible, ce qui veut dire qu'elle ne donnera jamais un résultat plus grand que la réalité pour la distance restante pour atteindre le but, A* retournera forcément le chemin le plus court. Dans notre problème, trouver une telle heuristique était possible. En effet, aucun trajet d'autobus ne planifie que le chauffeur dépasse la limite de vitesse, et la limite de vitesse maximum au Québec est 100km/h. Les trajets a pied vont tous exactement à 5 km/h. L'heuristique choisie a donc été le temps nécéssaire pour atteindre le but si on se déplaçais à vol d'oiseau (plus efficacement qu'un autobus) à 100 km/h (également ou plus rapidement qu'un bus). Dans notre graphe, ceci constitue donc une heuristique qui sera toujours plus petite ou égale au vrai coût pour atteindre le but. A* est donc un algorithme valide pour notre problème.

Comme la vitesse d'éxécution de A* est très variable en fonction de la qualité de l'heuristique et de la forme du graphe, il est très difficile d'estimer sa performance. Nous avons donc dû l'implementer pour tester sa performance. Nous avons aussi implementé l'algorithme de Dijsktra utilisant un monceau, car il était clair de par sa complexité qu'il allait être comparable. Pour finir, nous avons aussi implementé Floyd-Warshall bien qu'il était prouvé mathématiquement que cet algorithe allait être moins efficace que les autres, par souci de validation. Ce dernier prend tellement de temps que nous ne l'avons pas lancé des milliers de fois comme les autres pour comparer, car une seule itération prenait déjà plusieurs heures, en comparaison aux délais en secondes des autres algorithmes.

Toujours à des fins d'optimisation, dans l'algorithme de Dijsktra avec un monceau, nous avons utilisé pour le dit monceau un std::vector de std::pair pour les données. Cela permettait non seulement d'accéder à une position arbitraire en temps constant, en plus de nous permettre de garder les données triées en permanence, dans le but d'effectuer toutes les opérations de base le plus vite possible considérant le grand nombre de sommets et d'arcs mentionné ci-haut.

Pour ce qui est de A*, nous avons simplement utilisé des std::unordered_map pour stocker les données locales de l'algorithme. Ces données étaient seulement utilisées pour ajouter et retirer des noeuds à traiter et déjà traité, ainsi que pour conserver le coût pour atteindre chaque noeud et son prédécesseur. Comme il fallait lier ces valeurs à des numéros de sommets, une map était le contenur idéal pour stocker ces données et y accéder efficacement.

Pour ce qui est de Floyd-Warshall, toutes nos sources semblaient en accord pour dire qu'on ne pouvait, en aucun cas, l'accélérer pour qu'il aille plus vite que $O(n^3)$, nous n'avons donc pas investi de temps pour pouvoir l'améliorer un peu en changeant certains

conteneurs. Cet algorithme n'allait pas pouvoir compétitionner même avec les deux fournis dans tous les cas.

Un raisonnement similaire s'applique à l'algorithme de Bellman-Ford pour graphe acyclique, qui était inutilisable sur notre graphe cyclique. Aucun temps n'a été investi sur l'optimisation d'un algorithme qui ne correspondais aucunement à notre besoin.

Voici les résultats sur 20 itérations de 1000 éxécutions de chaque algorithme comparable (en secondes)

	Temps pour 1000 éxécutions (en secondes)				
Itération	Dijsktra	Bellman-Ford	Dijstra avec un monceau	A*
1	1551.565777	636.545959	39.17788	113.019096
2	1513.68857	628.367805	38.959443	112.220657
3	1532.705488	632.664152	38.995468	111.847314
4	1507.150621	633.01275	39.452386	113.321972
5	1501.893537	605.970276	35.846197	104.085826
6	1331.165604	624.306706	40.0558	114.249915
7	1541.910884	639.332729	39.74383	114.595064
8	1574.28261	637.94778	39.60704	113.674872
9	1451.051988	461.752434	30.538916	90.996137
10	1268.509749	437.075132	30.415079	90.143252
11	1251.800292	438.628744	30.60011	91.017075
12	1254.477438	437.707217	30.371799	90.727011
13	1253.449946	437.055203	30.358935	90.649005
14	1255.055964	436.67477	30.394916	90.193708
15	1251.545795	437.028707	30.258743	90.471553
16	1253.622606	436.669957	30.525571	90.520214
17	1253.307023	436.212511	30.496914	90.838718
18	1252.624551	436.808281	30.418905	90.46599
19	1316.968815	473.411276	31.491187	93.214379
20	1305.974364	482.760393	32.502839	95.040837
Total	27422.751622	10389.932782	680.211958	1991.292595
Moyenne	1371.1375811	519.4966391	34.0105979	99.56462975

Comme les résultats le montrent, l'algorithme de Dijsktra avec un monceau est le plus rapide, avec un résultat environ 3 fois plus rapide que l'algorithme A* avec notre heuristique. Comme nous n'avons pas réussi à créer une meilleure heuristique qui était prouvée comme admissible, ceci est donc la meilleure performance de A* dans ces circonstances. Dijsktra avec un monceau est environ 15 fois plus rapide que Bellman-Ford, qui était lui-même plus de 2 fois plus rapide que Dijsktra normal. Cet algorithme est donc le meilleur que nous avons trouvé, et nous le recommanderions à quiconque recherche un plus court chemin sur le réseau du RTC, car il arrive à trouver 1000 plus court chemins aléatoires en moins de 35 secondes, ce qui veut dire que pour un seul chemin, moins de 0.035 secondes seraient nécéssaires, ce qui est parfaitement acceptable pour un logiciel rendu disponible à un utilisateur.