Travail pratique 3

Travail présenté à Kim Rioux-Paradis
GLO-2100

réalisé par
Mathieu Boisvert ;
536 887 692

5 décembre 2021

1 But de ce travail pratique

Le but de ce travail est d'effectuer une démarche d'investigation afin de trouver le meilleur algorithme du plus court chemin possible sur les données du réseau du RTC et de l'appliquer sur celles-ci. Tout d'abord, il faudra examiner les différents algorithmes vus dans le cours selon leurs propriétés et leur complexité afin de sélectionner l'algorithme qui sera le plus performant, lorsqu'appliqué sur données du réseau du RTC. Puis, il faudra écrire cet algorithme en C++ afin de l'appliquer sur les données réelles du réseau du RTC pour prouver que celui-ci est bel et bien le meilleur algorithme possible.

2 Réflexion sur les algorithmes envisagés

Les algorithmes qui ont été considérés comme étant ceux qui permettent de déterminer le meilleur itinéraire d'un voyage en utilisant le réseau du RTC ont été les algorithmes de **Djikstra**, de **Bellman-Ford** et de **Floyd**, ainsi que l'algorithme **A***. Ceux de **Bellman-Ford**, **Floyd** et **A*** ont été rejetés pour ces raison :

- L'algorithme de **Bellman-Ford** n'a pas été utilisé, car nous n'avons pas à prendre en compte les arcs de poids négatifs dans le contexte de notre problème, et celui-ci est en pire cas en $O(|S||A|)$, ce qui est plus lent que l'algorithme de Djikstra qui est en pire cas en $O(|E| + |V|log|V|)$
- L'algorithme de **Floyd** a été rejeté puisque dans le contexte de notre problème, nous n'avons pas à calculer le plus court chemin entre toutes les paires de nœuds à la fois ; nous avons simplement à calculer le plus court chemin entre deux nœuds seulement. Sa complexité en pire cas étant $O(|V|^3)$, l'algorithme est aussi beaucoup plus lent que celui de Djikstra.
- L'algorithme **A*** a été rejeté car celui-ci nécessite l'utilisation d'une heuristique valide, donc l'utilisation d'un graphe ne possédant pas d'arcs dont le poids est plus petit ou égal à 0. Dans notre cas, certains arcs peuvent avoir un poids de 0. Ainsi, cet algorithme ne pourrait pas trouver de chemins plus courts dans certains cas.

Par conséquent, une version améliorée de l'algorithme de **Djikstra** a été choisie afin d'effectuer les calculs des chemins les plus courts entre deux nœuds du graphe dans ce travail. En effet, cette amélioration consiste en l'utilisation d'un *tas min*. Pour un graphe de n sommets et de m arcs, le temps d'exécution de l'algorithme est donc de $O((n + m) \log n)$ tel que vu dans les notes de cours. Puisque le graphe est relativement dense ($\pm$ 5 arcs par sommets), le fait d'opter pour l'utilisation d'un *tas min* nous permet s'assurer que chaque nœud est traité exactement une fois, ce qui nous permet de gagner en performance.

3 Description des étapes algorithmiques implémentées

Tout d'abord, trois vecteurs différents sont initialisés afin de contenir le poids de chacun des arcs (`std::vector<size_t>`), les nœuds solutionnés (`std::vector<bool>`) ainsi que les prédécesseur de chacun de ces nœuds solutionnés (`std::vector<size_t>`). La taille initiale

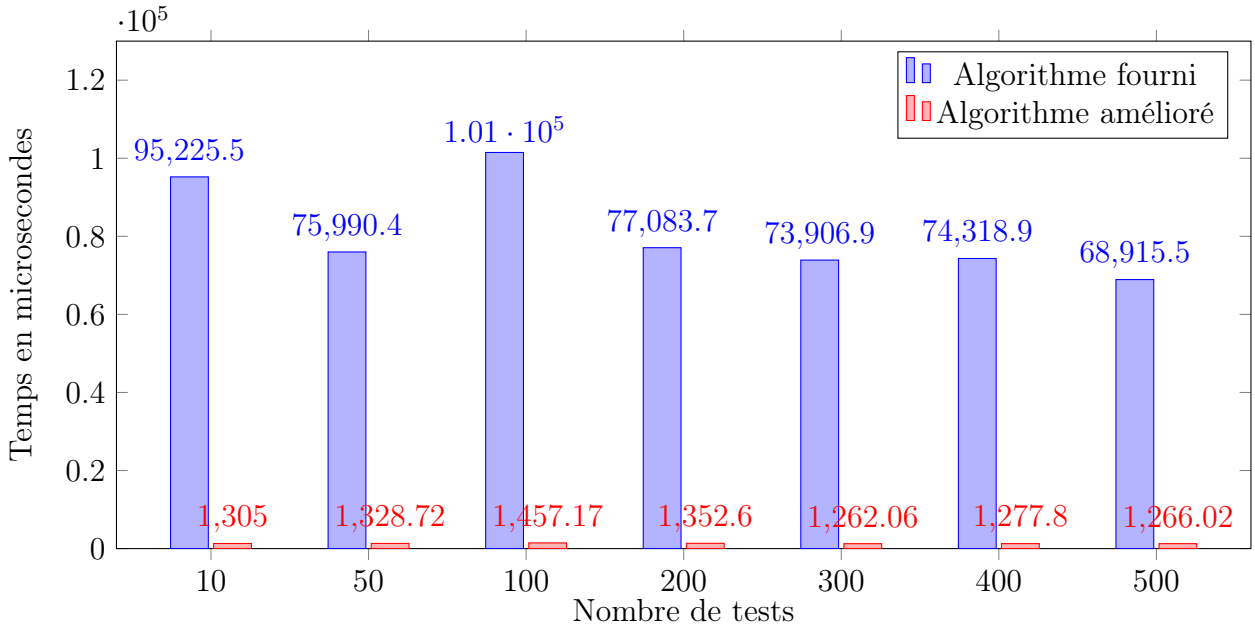
de ces 2 vecteurs correspond au nombre d'arcs du graphe, et tous les éléments des vecteurs sont remplis avec la valeur maximum possible pouvant être contenue par leur type de données (`true` dans le cas du vecteur contenant les nœuds solutionnés). Quoique la complexité temporelle de l'algorithme n'est pas altérée à la suite suite de l'implémentation de cette modification, le fait de remplir les éléments de ces vecteurs directement par le constructeur de ceux-ci nous évites de devoir ajouter une boucle ne servant qu'à l'attribution de valeurs aux éléments des vecteurs avant la boucle principale de l'algorithme.

Ensuite, comme mentionné dans le paragraphe précédent, la modification la plus importante apportées à l'algorithme de Djikstra a été l'utilisation d'une `std::priority_queue` afin de stocker les nœuds solutionnés en bordure des nœuds non solutionnés. Puisqu'il est impossible de récupérer un élément à l'intérieur d'une `std::priority_queue`, l'algorithme permet la duplication de clés afin de pouvoir modifier un élément de la file sans devoir reconstruire celle-ci lors de la modification de l'élément. En combinant ceci avec l'utilisation d'un comparateur (`std::greater<T>`) lors de l'initialisation de la `std::priority_queue`, on évite de devoir reconstruire l'objet à chaque relâchement d'arc et on peut toujours avoir accès au nœud dont la distance est la plus proche tout en ignorant les autres nœuds dupliqués.

Par ailleurs, avec l'ajout des modifications mentionnées, le temps d'exécution de l'algorithme est grandement réduit. En effet, la complexité temporelle de celui-ci devient, en pire cas, $O(|E|log|V|)$, contrairement à, en pire cas, $O(|E|+|V|log|V|)$ pour l'algorithme de base de Djikstra.

4 Présentation des résultats obtenus

Tout d'abord, j'ai testé mon algorithme sur 10 itinéraires consécutifs identiques à ceux fournis dans le fichier *out.txt* dans le but de pouvoir comparer mes résultats avec ceux ayant été compilés à l'aide de l'algorithme donné dans ce travail. Les résultats démontrent que mon algorithme choisit toujours le même itinéraire que lorsqu'on utilise l'algorithme fourni, mais en étant beaucoup plus rapide. Ensuite, j'ai testé mon algorithme sur 200 tests différents (l'arrêt de départ et l'arrêt d'arrivée ayant été générés de façon aléatoire) afin de déterminer le facteur de rapidité entre mon algorithme et celui qui a été fourni et aussi afin de savoir si mon algorithme était en mesure de déterminer un itinéraire valide à 100% du temps. Après analyse des résultats, ceux-ci ont démontrés que mon algorithme retourne toujours un itinéraire valide, soit un itinéraire partant du point d'origine et se rendant au point destination, et ce, à une rapidité phénoménale. Le tableau ci-dessous présente les résultats obtenus lors de l'exécution l'exécution de mon algorithme sur le graphe, versus l'exécution de l'algorithme fourni sur celui-ci encore une fois. Tous les tests ont été effectués dans le même environnement, soit sur un Macbook Pro 2019, avec le minimum d'applications ouvertes lors des tests :



Afin d'obtenir le facteur d'accélération moyen de mon algorithme par rapport à celui fourni, on peut simplement diviser chacun des résultats compilés pour l'algorithme fourni avec ceux pour mon l'algorithme, et ce, pour chacun des nombres de tests, et d'effectuer une moyenne avec chacun de ces résultats tels que :

$$\frac{\frac{95225.5\mu s}{1305\mu s} + \frac{75990.4\mu s}{1328.72\mu s} + \frac{101476\mu s}{1457.17\mu s} + \frac{77083.7\mu s}{1352.6\mu s} + \frac{73906.9\mu s}{1262.06\mu s} + \frac{74318.9\mu s}{1277.8\mu s} + \frac{68915.5\mu s}{1266.02\mu s}}{7} \approx 57$$

Ainsi, mon algorithme est en moyenne 57 fois plus rapide dans le contexte de notre problème que celui fourni dans ce travail.

5 Conclusion

Déterminer le chemin le plus court entre deux nœuds d'un graphe est un problème intéressant, mais complexe. Plusieurs mathématiciens et informaticiens ont déjà publié leurs propres recherches proposant un algorithme permettant de déterminer le chemin le plus court entre deux nœuds d'un graphe, mais il reste important de prendre en compte les caractéristiques de ces algorithmes avant de conclure qu'un de ceux-ci fonctionne correctement pour notre type de données. Dans mon cas, en analysant la structure des données représentés dans le graphe et en comparant différents algorithmes populaires, j'ai été en mesure d'en sélectionner un compatible et d'y apporter une amélioration afin de réduire son temps d'exécution dans le contexte du problème énoncé par le travail. Ceci étant dit, je crois qu'il est important de prendre le temps d'analyser les algorithmes de plus court chemin s'offrant à nous, car en fonction de la constitution de nos données, le temps d'exécution d'un algorithme peut largement différer d'un autre.