Rapport : Résolution du problème du Voyageur de Commerce avec l'algorithme de Little

M. Migne*A. El Ahmar†M. Kouakou‡

6 octobre 2023

*Matthieu.Migne@grenoble-inp.org>
†Anas.ElAhmar@grenoble-inp.org
‡Mouahe.Kouakou@grenoble-inp.org

Table des matières

1 Présentation du problème

Un commerçant part d'une ville et souhaite visiter n-1 autres villes une seule et unique fois et ensuite retour dans la ville de départ. Le problème est de déterminer dans quel ordre devrait-il visiter les villes pour minimiser la distance totale qu'il va devoir parcourir. Le coût entre chaque couple de villes est connu. Le problème peut être asymétrique, c'est-à-dire : soit c(i,j) le coût pour aller de la ville i à la ville j, c(i,j) peut être différent de c(j,i). Le nombre minimum de villes est 3, car en-dessous il ne serait pas possible de faire un parcours des villes circulaires.

2 Présentation de l'alogrithme de Little

Il y a (n-1) ! parcours possibles. Le temps nécessaire pour tester chacune de ces solutions et les comparer entre elles étant particulièrement élevé, nous appliquer un algorithme *Branch and Bound* pour résoudre le problème de manière beaucoup plus efficace.

Nous allons diviser le problème en plusieurs sous-problèmes et calculer pour chacun d'eux la borne inférieure du coût du meilleur tour pour ce sous-problème. Les bornes vont guider l'ordre dans lequel nous allons partionner les sous-problèmes et les résoudre. Les sous-problèmes sont des nœuds d'un arbre, et le processus de partionnement est une branche. Chaque nœud indique si l'on a prit ou pas un couple de ville dans le partionnement du problème de la branche correspondante.

Nous avons implémenté une variante de l'algorithme de Little disponible dans la source à la fin de ce rapport, nous avons notamment décidé de stocker la matrice réduite du nœud asscoié dans chaque nœud, et non pas de recalculer à chaque fois la matrice réduite à chaque qu'on créé un nouveau nœud, cela permet d'améliorer la rapidité d'exécution de l'algorithme en sacrifiant un peu d'espace mémoire.

Aussi, nous ne supprimons pas de lignes et de colonnes de notre matrice, mais plaçons juste des infinis sur les couples de villes qu'il n'est plus possible de prendre. Cela est plus ou moins équivalent à l'approche du papier et est simplement un choix d'implémentation.

Pour résoudre la difficulté d'éviter de compléter un parcours sans être passer pour toutes les villes, nous allons pour chaque nœud éviter que les nœuds fils directement puisse choisir un couple correspondant à la complétion d'un tour, excepté si ces nœuds fils correspondent à des feuilles finales.

3 Comparaison avec l'algorithme Brute Force

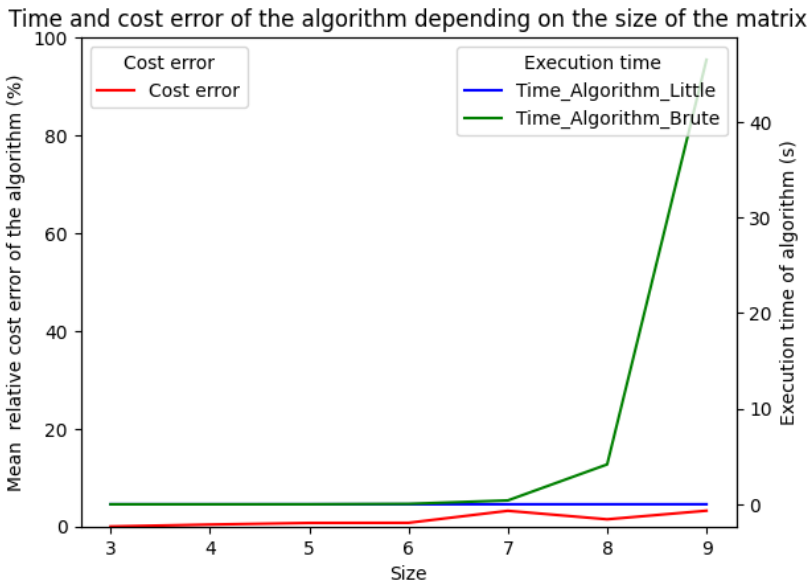

FIGURE 1 – Temps d'exécution de Little et Brute Force par rapport à la taille de la matrice, ainsi que l'erreur relative de Little par rapport à Brute Force

Nous avons généré le graphique ci-dessus en testant 100 itérations pour chaque taille de la matrice entre 3 et 9, puis en effectuant la moyenne pour chaque taille des temps d'exécution ainsi que l'erreur de coût relative de Little par rapport à Brute Force en pourcentages.

Nous pouvons observer que le temps d'exécution est négligeable devant celui du Brute Force, et nous pouvons voir que le temps d'exécution de Brute Force a bien une allure hyperexponentielle comme attendue.

Nous voyons également que l'erreur n'est pas strictement nulle, bien qu'étant relativement faible (majoritairement à 0% et a des pics à 5%) il apparait que notre implémentation ne trouve aps toujours une solution optimale au problème, bien que la solution trouvé n'est jamais très loin de celle optimale.

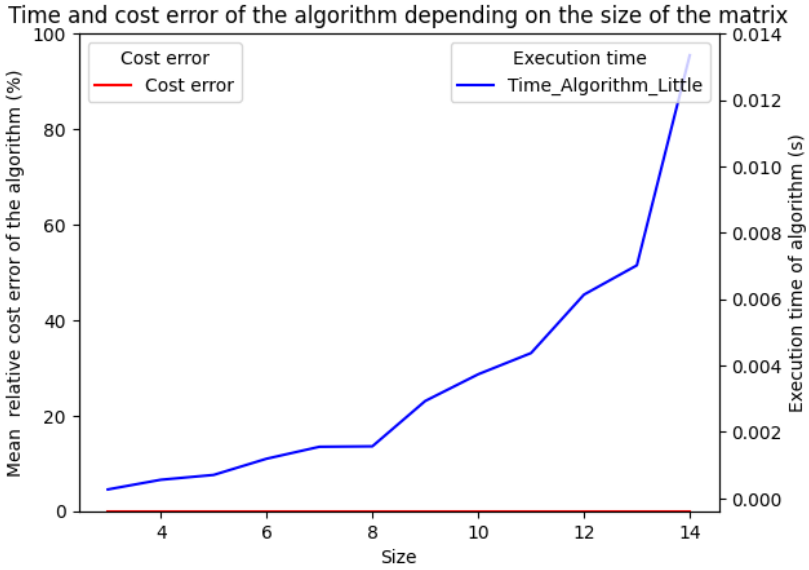

FIGURE 2 – Temps d'exécution de Little par rapport à la taille de la matrice

Nous avons générer le graphique ci-dessus avec uniquement la moyenne du temps d'exécution de l'algorithme de Little et l'erreur de coût relative à Brute Forces, sur une moyenne de 100 itérations par taille de matrice.

Nous pouvons observer que le temps d'exécution de l'algorithme Little est également exponentielle voire hyperexponentielle, ce qui n'était pas visible sur le graphique précédent étant donné que bien qu'étant tous les deux exponentielles, Brute Force a une exponentielle beaucoup plus forte.

4 Améliorations possibles

L'erreur relative du coût devrait être nulle jusuqu'à une taille de matrice bien supérieure à 13, le fait que nous ayons un peu d'erreur même sur des matrices de taille relativement faible est sûrement liée à une erreur d'implementation quelque part dans notre algorithme.

Concernant les villes de départ et d'arrivée de chaque chemin pour chaque nœud, dans notre implémentation actuelle nous les recalculons à chaque nœud, mais il y aurait sûrement une manière plus efficace de sauvegarder ces villes.

Nous pourrions trouver d'abord la feuille dans le cas on où on prend chacun des chemins à chaque fois, cela permettrait d'obtenir rapidement une borne supérieure intéressante.

Il serait également possible de trouver une manière plus efficace de stocker nos sous-problèmes qu'en utilisant un arbre qui sur une matrice de grande taille générera une structure particulièrement importante.

Enfin, si nous prenions uniquement des matrices symétriques en entrée, nous pourrions accélérer grandement le temps d'exécution en réduisant grandement le nombre de calculs.

5 Sitographie

AN ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM : http://www.heuristieken.nl/resources/(1963)Littleetal-_refurbished_AlgorithmforTSP.pdf