ORO

Problème du voyageur : résolution par méthode *branch & bound*

Auteurs :
M. Sébastien Dubois

Professeur :
Devan SOHIER

14 Mai 2022

Table des matières

1 Introduction

Le projet porte sur le problème du voyageur. Une méthode *branch & bound* ainsi qu'une *bruteforce* ont été développées. Les codes compilés à l'aide d'un *Makefile* avec la commande `make` s'exécutent sur un exemple simple à l'aide de la commande `make run`. Il est possible d'exécuter les programmes selon diverses options :

— `./<program> -e` permet d'exécuter avec le fichier `configuration.txt` ;
— `./<program> -f my_file.txt` permet d'exécuter avec le fichier correspondant ;
— `./<program> -n number` permet d'exécuter selon des données de taille `number` générés aléatoirement.

avec :

— `<program>` à remplacer par `branch` pour exécuter selon la méthode de *Branch & Bound* ;
— `<program>` à remplacer par `bruteforce` pour exécuter selon la méthode de *Brutefroce*.

Vous pouvez créer votre propre fichier de configuration en utilisant des espaces comme séparateur. Les termes diagonaux sont à notifier par `inf`. Le programme fonctionne avec des valeurs flottantes positives.

Les codes fonctionnent correctement sur OS 5-17-3-ARCH1-1 avec `gcc@11.3.0`.

2 Présentation de la méthode et algorithme associe

2.1 Approche branch&bound

L'approche de *Branch & bound* - en français *Séparation et évaluation* consiste comme son nom l'indique, à l'exécution itérative ou récursive de deux phases. La phase de **séparation** opère une visite des combinaisons possibles au sein du jeu de données associé au problème. L'exploration s'effectue en tenant compte des résultats optimaux précédents. La phase d'**évaluation** consiste en la sélection ou l'abandon des branches précédemment explorées lorsque l'évaluation associée à une fonction coût ne lui est pas favorable. Une subtilité de la méthode réside dans le fait qu'une branche abandonnée temporairement peut redevenir rentable à explorer plus tard. Globalement, la méthode de *Branch & Bound* permet de s'affranchir de l'évaluation des branches *a priori* non optimales. La méthode permet ainsi de réduire l'explosion combinatoire, comme par exemple les problèmes dits **NP-complets**.
Il n'existe pas une unique méthode de *Séparation & évaluation*. Le critère d'évaluation et de sélection des branches joue un rôle important sur l'efficience de l'algorithme. Le critère de sélection doit être finement choisi afin de trouver le bon compromis entre efficience d'évaluation et précision. En effet, il existe parfois des stratégies visant à réduire considérablement la complexité du problème - et ainsi l'efficience algorithmique - mais réduisant l'efficience mathématique de celle-ci.

2.2 Problème du voyageur

Le problème du voyageur - appelé aussi du chemin hamiltonien sur un graphe complet - s'appuie sur l'écriture d'une matrice de coûts représentant une distance orientée entre un site a un autre. Le principe est le suivant :

— L'objectif est de déterminer le plus court chemin passant strictement une fois sur chaque site du graphe complet G ;
— l'écriture matricielle permet de définir un coût $c_{ij} \leq 0$ associée au trajet d'un site P_i a un site P_j suivant $c_{ij} = \|\vec{P_iP_j}\| = A_{ij}$.
— Les coûts associes au trajet d'un site à un autre ne sont pas nécessairement symétriques. Autrement dit, $A^T A$ en général.

Voici un exemple d'une matrice de transfert associé à un problème de voyageur sur un graphe G de cardinal $card(G) = 5$:

$$A_{ij} = \begin{bmatrix} \infty & 7 & 9 & 6 & 1 \\ 7 & \infty & 3 & 4 & 9 \\ 9 & 3 & \infty & 2 & 5 \\ 6 & 4 & 2 & \infty & 3 \\ 1 & 9 & 5 & 3 & \infty \end{bmatrix} \tag{1}$$

et voici une représentation d'un graphe complet composé de 5 sites ;

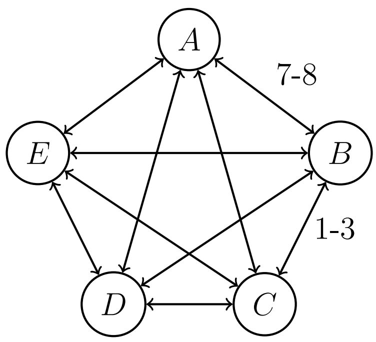


Figure 1

Les valeurs infinies sur la diagonale permettent de contraindre le problème en dissuadant l'algorithme de résolution d'explorer un tel chemin que l'on sait non-Hamiltonien.

Par exemple, la séquence de parcours $s = (A, B, C, D, E, A)$ possède un coût de $c(s) = 7 + 1 + 2 + 4 + 1.1 = 15.1$. L'objectif est de déterminer la séquence - ou du moins une séquence - optimale $s = \underset{s \text{ hamiltonien}}{\text{argmin}} \; (c(s))$.

2.3 Méthode utilisée

L'écriture matricielle propre au problème du voyageur se prête bien à l'utilisation de la méthode suivante :

— La matrice des coûts est interprétée par le parseur ;
— la matrice est réduire en ligne et en colonne, c'est-à-dire qu'une soustraction sur les lignes et sur les colonnes permet d'imposer que le terme de plus petite valeur est de valeur nulle ;
— Pour chaque itération d'avancement de l'exécution, le chemin de coût le plus faible est récupéré afin de l'explorer. Les nouveaux chemins sont alors sauvegardées ;
— En fonction des sites du graphe déjà explorés, la matrice propre au noeud est modifiée afin de ne permettre l'accès qu'à des noeuds non visités ;

— la procédure est répétée impérativement jusqu'à obtenir un chemin HAMILTONIEN. Il s'agit alors d'un chemin HAMILTONIEN de taille minimal.

Note : Encore une fois, on précise qu'il n'existe pas nécessairement qu'un seul chemin HAMILTONIEN optimal.

Du point de vue des structures de données, il a été utilise :

— afin de permettre la gestion de fichiers, une structure de données `s_problem` a été développée afin de stocker les données du fichier de configuration et de la matrice des couts ;

— une structure de `s_node`, permettant de stocker la matrice modifiée, le chemin parcouru, le coût ainsi que la hauteur dans l'arbre ;

— la matrice modifiée - aussi appelée `reduced_matrix`, est stockée sous forme de tableau unidimensionnel et non pas en tableau de tableau. Cela permet notamment d'améliorer les performances sans un surcout de temps de développement (bien au contraire) ;

— les chemins de chaque `s_node` sont stockés sous forme de tableaux de structures `s_segment` possédant un départ et une arrivée ;

— les noeuds sont stockes dans un tableau de noeuds `data.nodes`. Il a été nécessaire d'écrire des fonctions permettant de retirer un élément particulier ou d'en ajouter un à cette liste de noeuds. En effet, le noeud possédant le plus petit chemin est extrait et supprime de la liste à chaque itération de parcours. Puis, les nouveaux noeuds sont ajoutes à la fin de la liste.

En sommme, vous remarquerez notamment que chaque noeud du graphe possède sa propre matrice. Ainsi, la complexité en mémoire est très importante. Il existe certainement une alternative consistant à la recalculer à chaque fois.

3 Résultats

3.1 Comparaison avec la solution brute

Une solution brute est générée en calculant la longueur de chaque chemin HAMILTONIEN à l'aide d'une permutation. Ainsi, toutes les combinaisons sont testées et le plus petit chemin HAMILTONIEN est gardé. Cela sert de référence pour comparer avec la solution par algorithme de *Branch & Bound*. Les calculs ont été effectués pour n allant de 2 a $n = 14$ sans voir de différence entre les deux algorithmes. Notons que les deux codes utilisent des matrices aléatoires identiques. Voici quelques exemples de résultats :

```
./branch -e

--- Branch And Bound method ---

You selected the embedded file option.
Read file with success !
read data ..
Size of graph : 5

Solution :
 - Length : 16.000000
 - Path   : 0 - 4 - 3 - 2 - 1 - 0

Elapsed time :
 - s      : 0.000012
 - ms     : 0.012320
```

Listing 1 – Branch and Bound

```
./bruteforce -e

-----  Bruteforce method -----

You selected the embedded file option.
Read file with success !
read data ..
Size of graph  : 5

Solution :
 - Length : 16.000000
 - Path   : 0 - 1 - 2 - 3 - 4 - 0

Elapsed time :
 - s      : 0.000011
 - ms     : 0.011190
```

Listing 2 – Bruteforce

FIGURE 2 – Résultats de l'exécution des deux codes pour la matrice de référence

Les résultats sont identiques sur la matrice de référence. Notons que l'ordre des chemins est inversé entre les deux méthodes. Cela provient du fait que la matrice des coûts est symétrique et, ainsi, il existe au moins deux chemins optimaux. La taille des données est trop petite pour mettre en exergue des différences d'efficience entre les méthodes. Il existe au sein du dossier du projet un fichier de configuration nomme `configuration_large.txt`. Son interprétation au sein du code avec `./branch -f configuration_large.txt` permet d'obtenir les résultats suivants :

```
./branch -f configuration_large.txt

--- Branch And Bound method ---

You selected the file option.
Read file with success !
read data ..
Size of graph : 10

Solution :
 - Length : 37.000000
 - Path   : 0 - 6 - 4 - 2 - 3 - 5 - 8 - 9 - 1 - 7 - 0

Elapsed time :
 - s      : 0.000336
 - ms     : 0.335880
```

Listing 3 – Branch and Bound

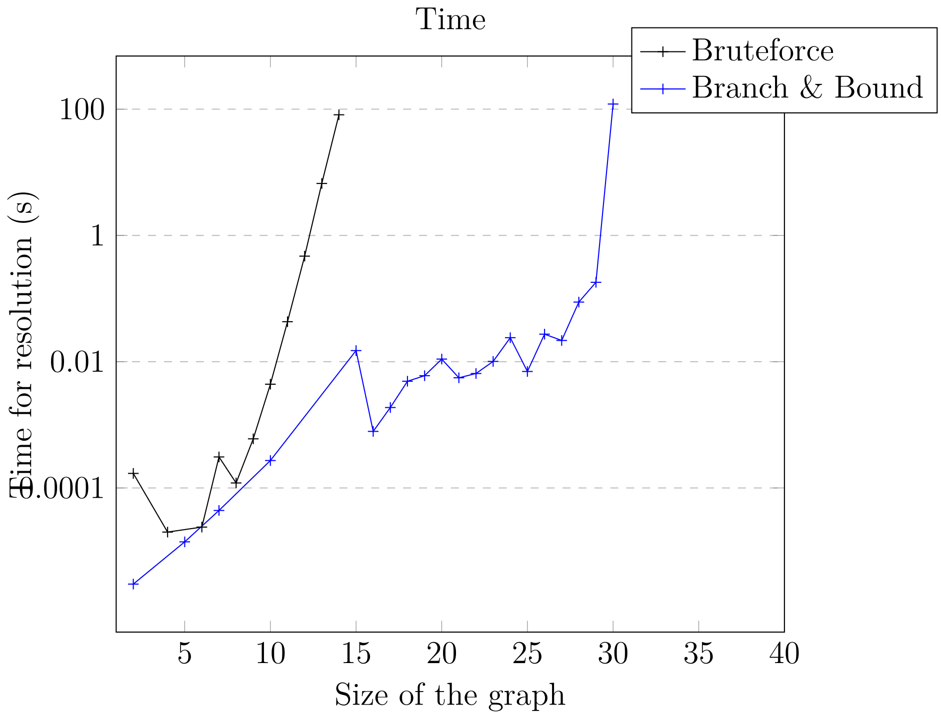
```
./bruteforce -f configuration_large.txt

-----  Bruteforce method -----

You selected the file option.
Read file with success !
read data ..
Size of graph  : 10

Solution :
 - Length : 37.000000
 - Path   : 0 - 6 - 4 - 2 - 3 - 5 - 8 - 9 - 1 - 7 - 0

Elapsed time :
 - s      : 0.004558
 - ms     : 4.557880
```

Listing 4 – Bruteforce

FIGURE 3 – Résultats de l'exécution des deux codes pour la matrice large

Il est désormais clair que la méthode *Branch and Bound* dépasse la méthode naïve sur cet exemple. De plus les résultats sont identiques.

La complexité de la méthode naïve est bien plus importante que celle de la méthode améliorée. Voici une étude des performances en fonction de la taille du problème. Notez que pour chaque n, la matrice générée aléatoirement est identique pour chacune des deux méthodes :



Les performances de l'approche naïve est clairement désastreuse. En revanche, il semble que les performances de la méthode *Branch & Bound* dépendent des valeurs du tableau. Dans le pire des cas, la complexité serait alors la même que l'algorithme *bruteforce*. On peut se servir de machines distribuées afin d'explorer plusieurs fois mais de façon différente l'arbre afin d'éviter ces cas.

4 Conclusion

La méthode *Branch & Bound* permet de résoudre des problèmes a la combinatoire très complexe de façon bien plus efficace que par des méthodes naïves. La complexité de développement est somme toute accessible et des méthodes plus générales peuvent certainement être développées afin de s'adapter aux différents problèmes. Une multitude de stratégies n'ont pas été explores a travers cet expose - ni même la parallélisation.