

ORO

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

Auteurs :

M. Sébastien DUBOIS

Professeur :

14 Mai 2022

Table des matières

1	Introduction	2
2	Présentation de la méthode et algorithme associe	2
2.1	Approche branch&bound	2
2.2	Problème du voyageur	2
2.3	Methode des regrets	3
3	Resultats	3
3.1	Comparaison avec la solution brute	3
4	conclusion	5

1 Introduction

Le projet se porte sur le problème du voyageur. Une methode *branch & bound* ainsi qu'une *bruteforce* ont été développées. Les codes compiles a l'aide d'un *Makefile* avec la commande **make** s'exécutent sur un exemple simple a 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.

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

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, a l'exécution itérative ou récursive de deux phases. La phase de séparation opère une visite des combinaisons possibles au sein du jeu de données associe 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 a 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 a explorer. 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'efficacité de l'algorithme. Le critère de sélection doit être finement choisi afin de trouver le bon compromis entre efficacité d'évaluation et précision. En effet, il existe parfois des stratégies visant a réduire considérablement la complexité du problème - et ainsi l'efficacité algorithmique - mais réduisant l'efficacité 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 cout $c_{ij} \leq 0$ associée au trajet d'un site P_i a un site P_j suivant $c_{ij} = \|P_i \vec{P}_j\| = A_{ij}$.

- Les coûts associés au trajet d'un site a un autre ne sont pas nécessairement symétriques. Autrement dit, $A^T A$ en général.

Voici un exemple d'une matrice de transfert associe a un problème de voyageur sur un graphe G de cardinal $\text{card}(G) = 5$:

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

et voici une representation d'un graphe complet compose de 5 sites ;

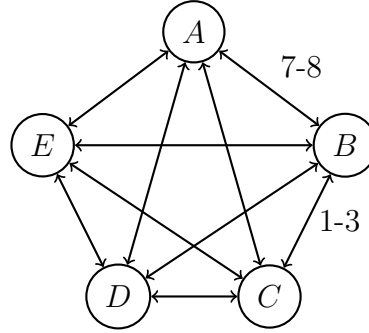


FIGURE 1

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 $s = \underset{s \text{ hamiltonien}}{\operatorname{argmin}} (c(s))$.

2.3 Methode des regrets

L'écriture matricielle propre au problème du voyageur se prête bien a l'utilisation de la methode

3 Resultats

3.1 Comparaison avec la solution brute

Une solution brute est générée en calculant la longueur de chaque chemin hamiltonien a l'aide d'une permutation. Ainsi, toutes les combinaisons sont testées et le plus petit chemin hamiltonien est garde. Cela sert de référence pour comparer avec la solution par algorithme de *Branch & Bound*. Les calculs ont été effectues 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 :

```

1 ./branch -e
2
3 ——— Branch And Bound method ———
4
5 You selected the embedded file option.
6 Read file with success !
7 read data ..
8 Size of graph : 5
9
10 Solution :
11 - Length : 16.000000
12 - Path : 0 - 4 - 3 - 2 - 1 - 0
13
14 Elapsed time :
15 - s : 0.000012
16 - ms : 0.012320

```

Listing 1 – Branch and Bound

```

1 ./bruteforce -e
2
3 ——— Bruteforce method ———
4
5 You selected the embedded file option.
6 Read file with success !
7 read data ..
8 Size of graph : 5
9
10 Solution :
11 - Length : 16.000000
12 - Path : 0 - 1 - 2 - 3 - 4 - 0
13
14 Elapsed time :
15 - s : 0.000011
16 - ms : 0.011190

```

Listing 2 – Bruteforce

FIGURE 2 – Resultats de l’executuion des deux codes pour la matrice de reference

Les résultats sont identiques sur la matrice de référence. Notons que l’ordre des chemins est inverse 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’efficacité entre les méthodes. Il existe au sein du dossier du projet un fichier de configuration nommé `configuration_large.txt`. Son interprétation au sein du code avec `./branch -f configuration_large.txt` permet d’obtenir les résultats suivants :

```

1 ./branch -f configuration_large.txt
2
3 ——— Branch And Bound method ———
4
5 You selected the file option.
6 Read file with success !
7 read data ..
8 Size of graph : 10
9
10 Solution :
11 - Length : 37.000000
12 - Path : 0 - 6 - 4 - 2 - 3 - 5 - 8 - 9 - 1 - 7 - 0
13
14 Elapsed time :
15 - s : 0.000336
16 - ms : 0.335880

```

Listing 3 – Branch and Bound

```

1 ./bruteforce -f configuration_large.txt
2
3 ——— Bruteforce method ———
4
5 You selected the file option.
6 Read file with success !
7 read data ..
8 Size of graph : 10
9
10 Solution :
11 - Length : 37.000000
12 - Path : 0 - 6 - 4 - 2 - 3 - 5 - 8 - 9 - 1 - 7 - 0
13
14 Elapsed time :
15 - s : 0.004558
16 - ms : 4.557880

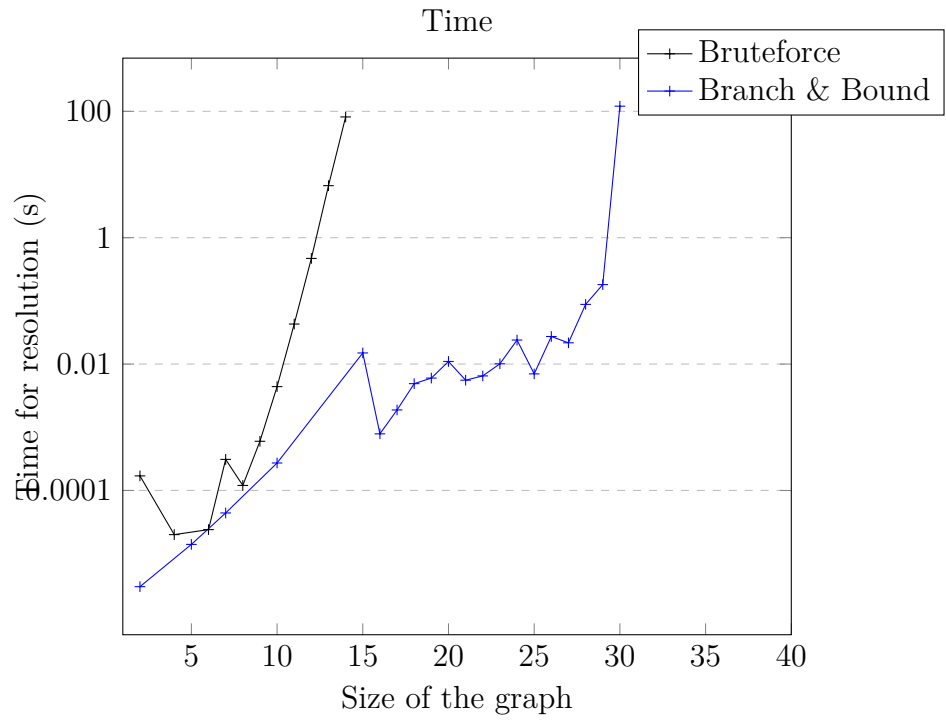
```

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 :



4 conclusion