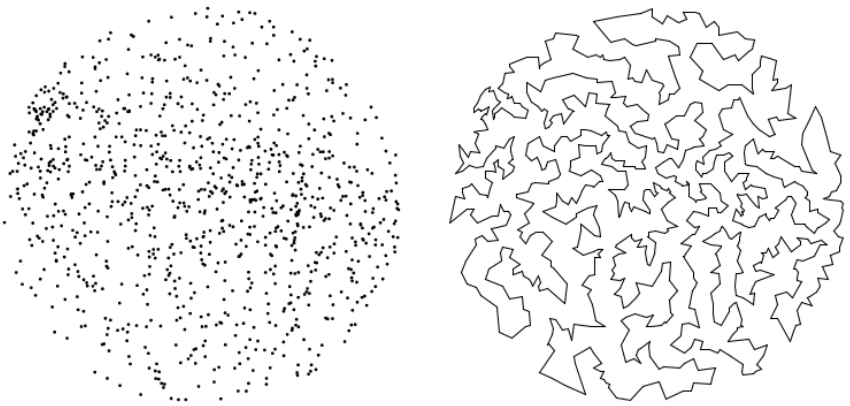
Optimisation et Recherche Opérationnelle

Résolution du problème du voyageur de commerce par séparation et évaluation

Auteur :
M. Gabriel Dos Santos

Enseignant référent :
M. Devan Sohier



M1 High Performance Computing and Simulation

14 mai 2022

Table des matières

1 Introduction

Le problème que nous avons choisi de résoudre pour ce projet est celui du voyageur de commerce, *travelling salesman problem* (ou TSP) en anglais. L'objectif est déterminer une solution à ce problème à l'aide de la méthode de séparation et évaluation (ou *branch-and-bound).* L'implémentation présentée ci-après a été réalisée en langage C.

Dans la suite de ce rapport, nous commencerons par présenter brièvement le problème du TSP ainsi que l'algorithme de séparation et évaluation. Nous verrons ensuite quels ont été les choix d'implémentation, avec une attention particulière sur les structures de données utilisées. Nous poursuivrons avec une démonstration du code et des résultats obtenus sur des graphes de différentes tailles. Enfin, nous conclurons avec quelques idées d'amélioration, notamment en terme de performance.

2 Présentation du problème et de la méthode de *branch-and-bound*

Le problème du voyageur de commerce est un problème d'optimisation sur un graphe complet (ou clique) G tel que $G = (V, A, \omega)$ avec V un ensemble de sommets, A un ensemble d'arcs reliant tous les sommets deux à deux et ω une fonction de coût sur ces arcs. Il consiste en le calcule du plus court chemin passant par tous les sommets du graphe exactement une seule fois (recherche d'un cycle hamiltonien).

Un exemple de cas concret :
Une entreprise possède quatre locaux (A, B, C et D) à travers la région. Pour des raisons de logistique, la direction de l'entreprise a besoin de visiter l'ensemble de ses locaux et souhaite minimiser le coût du trajet entre eux. À cette fin, elle va chercher à déterminer le chemin le plus court reliant les sites. Ce problème peut être modélisé à l'aide d'un graphe comme présenté dans la figure 1.

En se servant de la méthode de séparation et évaluation, il est possible de calculer le cycle optimal à travers ce graphe. L'algorithme de *branch-and-bound* est rappelé dans la figure 2 ci-après.

En appliquant ce dernier sur le graphe, il est possible de déterminer le parcours optimal suivant : $A \longrightarrow B \longrightarrow C \longrightarrow D$, ayant un coût total de 62.

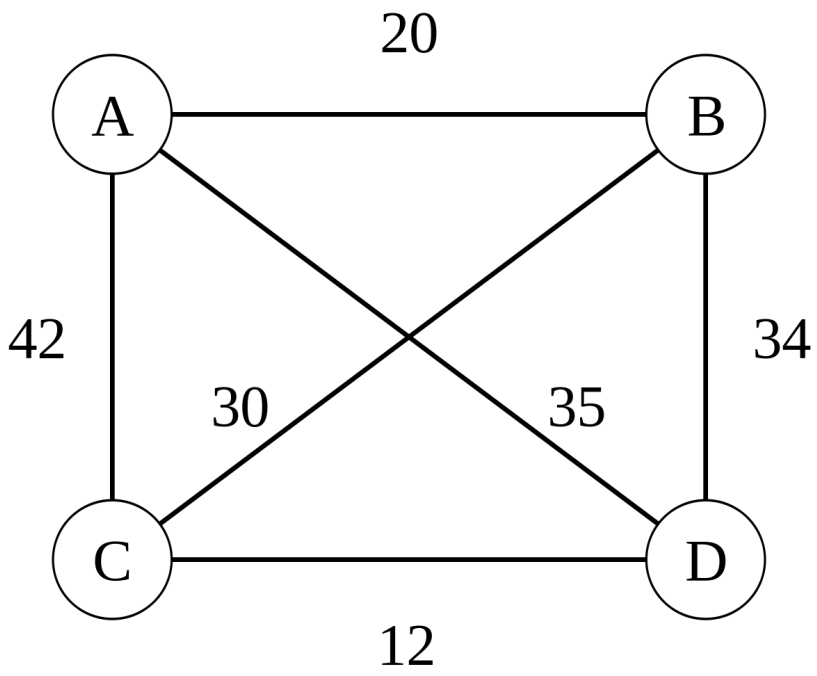


FIGURE 1 – Représentation du problème sous la forme d'une clique

A : liste des noeuds en attente
BS : valeur de la meilleure solution connue
S_0 : racine de l'arborescence = probleme general
Tant que $A \neq \emptyset$ faire
  SELECTIONNER S, le noeud de plus grande priorite de A
  SEPARER S : soient S_i ses fils
  Pour chaque S_i, faire :
    Si S_i est une solution (feuille)
      ALORS Si (val(S_i) < BS)
        BS = Val(S_i) et **ELAGUER** A
      SINON
        EVALUER S_i : $g(S_i)$
        Si (g(S_i) < BS) Alors Inserer S_i dans A
        (sinon Elaguer S_i)

FIGURE 2 – Pseudo-code pour l'algorithme de *branch-and-bound*

3 Choix d'implémentation

L'ensemble du travail réalisé pour ce projet est disponible sur le dépôt GitHub suivant : github.com/tsp

Pour compiler et exécuter le projet, veuillez vous servir du Makefile dont les commandes sont détaillées dans le README fourni avec le code source.

3.1 Fichier de configuration

Afin de faciliter l'exécution du code sur des graphes de taille différente, le programme prend en entrée le chemin d'un fichier de configuration. Ce dernier doit contenir la matrice d'adjacence du graphe à étudier, où les poids sont des entiers (signés ou non), au format suivant :

Structure du fichier de configuration

```
NOMBRE_DE_NOEUDS
POIDS_0-0 POIDS_0-1 POIDS_0-2 ...
POIDS_1-0 POIDS_1-1 POIDS_1-2 ...
POIDS_2-0 POIDS_2-1 POIDS_2-2 ...
...
```

La diagonale principale ne doit contenir que des zéros. Dans le cas du graphe vu dans la section précédente, le fichier de configuration serait :

Fichier : `sample_config.txt`

```
4
0  20 42 35
20 0  30 34
42 30 0  12
35 24 12 0
```

3.2 Architecture et organisation du code

L'architecture du projet se veut modulaire, ainsi l'organisation du code source se fait de la façon suivante :

- Le dossier `ext` contient le code source des librairies externes au projet (présentées dans la sous-section suivante) ;
- Le dossier `include` contient les fichiers d'en-têtes dans lesquels sont définies les différentes structures de données utilisées ainsi que les fonctions nécessaires :
  - `config.h` définit la structure de configuration ainsi que les fonctions nécessaires à son utilisation ;

— `solver.h` définit la structure du *solver* ainsi que les fonctions nécessaires à son utilisation ;
— `utils.h` définit des fonctions utilitaires mais qui ne se rapporte pas directement au *solver* ou à la configuration du problème.

— Le dossier `src` contient les fichiers sources dans lesquels sont implémentées les fonctions déclarées dans les fichiers d'en-tête. La fonction d'entrée du programme (`main`) y est définie, dans le fichier `main.c` ;
— Le dossier `target` contient les artefacts de la compilation (sous-dossier `deps`) ainsi que l'exécutable obtenu, `tsp` ;
— Le dossier `datasets` contient des fichiers de configuration supplémentaires, avec des graphes de différentes tailles afin de pouvoir tester le programme.

Pour plus de clarté, la structure est la suivante :

```
$ tree
```

```
Permissions Name
drwxr-xr-x  .
drwxr-xr-x  |-- ext
.rw-r--r--  |  |-- *.so   # External libraries
drwxr-xr-x  |-- include
.rw-r--r--  |  |-- *.h    # Header files
drwxr-xr-x  |-- src
.rw-r--r--  |  |-- *.c    # Source code files
drwxr-xr-x  |-- target
drwxr-xr-x  |  |-- deps
.rw-r--r--  |  |  |-- *.o # Object files
.rwxr-xr-x  |  |-- tsp
drwxr-xr-x  |-- datasets
.rw-r--r--  |  |-- *.txt  # Other configuration files
.rw-r--r--  |-- Makefile
.rw-r--r--  |-- sample_config.txt
```

3.3 Structure de données utilisées

Pour tous les "vecteurs" présentés dans la suite de cette section, nous utiliserons la structure générique `vec_t`, implémentée dans le dossier `ext/vec`. Le code source intégrale de celle-ci est disponible sur GitHub, à l'adresse suivante : github.com/libvec. J'ai moi-même implémentée cette structure dans le cadre d'un projet personnel l'été dernier. `vec_t` se base sur la même conception que la classe C++ `std::vector` et permet de stocker des éléments dans un tableau dynamique de façon générique (i.e. peu importe leur type). Elle offre diverses fonctions permettant de manipuler le vecteur et gère toutes les allocations mémoires nécessaires.

Ceci permet d'avoir une plus grande flexibilité et offre une API commune pour tous les types de tableaux dynamiques utilisés au sein d'un programme.

Structure de configuration :
La structure de configuration `config_t` définie dans le header `include/config.h` permet de stocker les informations lues dans le fichier de configuration. La matrice d'adjacence est stockée dans un vecteur en une seule dimension, ce qui permet d'éviter des indirections lors des accès mémoires et de les optimiser. La fonction utilitaire `adj_matrix_get` permet d'accéder aisément à une valeur de la matrice pour des indices i et j donnés.

Structure du *solver* :
La structure `solver_t` permet de garder en mémoire les informations de l'exécution de l'algorithme de *branch-and-bound*. Elle est composée d'un entier stockant le coût minimum courant pour le trajet le plus court, ainsi que de trois vecteurs :

- `visited_nodes` est un tableau de booléen de même dimension que le matrice d'adjacence stockant l'état des noeuds du graphe, *true* s'ils ont été visités, *false* sinon ;
- `path_taken`, de dimension *nb_noeuds* + 1, stocke le chemin couramment parcouru dans le graphe ;
- `optimal_path`, de même dimension que `path_taken`, stocke le chemin le plus optimal (de poids minimal) connu dans le graphe ;

La résolution du problème est implémentée à l'aide des fonctions `solve_tsp` (appelée depuis le `main`) et `solve_branch_and_bound` (appelée récursivement afin de construire l'arbre de recherche de l'algorithme de séparation et évaluation). À chaque fois que l'algorithme détermine un chemin plus optimal que celui connu jusqu'alors, on met à jour le valeur du coût de minimal de ce chemin ainsi que le `optimal_path` dans la structure du *solver*.

Pour plus de détails techniques sur l'implémentation, nous vous invitons à lire le code source du fichier `src/solver.c`, intégralement commenté et documenté.

4 Démonstration

Afin de tester l'implémentation de la résolution du problème du voyageur de commerce, nous avons récupéré des jeux de données de différentes tailles sur le site suivant : TSP datasets. Ceux-ci sont disponibles dans le dossier `datasets`.

Les listings suivants présentent la sortie obtenu à l'exécution du programme (sur des graphes de 13 et 17 noeuds, respectivement) :

```
$ make run
```

```
target/tsp sample_config.txt
Travelling Salesman Problem configuration:
  Number of nodes: 13
  Adjacency matrix:
     0 2451  713 1018 1631 1374 2408  213 2571  875 1420 2145 1972
  2451    0 1745 1524  831 1240  959 2596  403 1589 1374  357  579
   713 1745    0  355  920  803 1737  851 1858  262  940 1453 1260
  1018 1524  355    0  700  862 1395 1123 1584  466 1056 1280  987
  1631  831  920  700    0  663 1021 1769  949  796  879  586  371
  1374 1240  803  862  663    0 1681 1551 1765  547  225  887  999
  2408  959 1737 1395 1021 1681    0 2493  678 1724 1891 1114  701
   213 2596  851 1123 1769 1551 2493    0 2699 1038 1605 2300 2099
  2571  403 1858 1584  949 1765  678 2699    0 1744 1645  653  600
   875 1589  262  466  796  547 1724 1038 1744    0  679 1272 1162
  1420 1374  940 1056  879  225 1891 1605 1645  679    0 1017 1200
  2145  357 1453 1280  586  887 1114 2300  653 1272 1017    0  504
  1972  579 1260  987  371  999  701 2099  600 1162 1200  504    0

Minimum cost: 7293
Path taken: 0 -> 7 -> 2 -> 3 -> 4 -> 12 -> 6 -> 8 -> 1 -> 11 -> 10 -> 5 ->
↪  9 -> 0
Finished in 0.022s
```

```
$ target/tsp dataset/17_nodes.txt
```

```
Travelling Salesman Problem configuration:
  Number of nodes: 17
  Adjacency matrix is too big to print, sorry!

Minimum cost: 2085
Path taken: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9
↪  -> 1 -> 4 -> 8 -> 11 -> 15 -> 0
Finished in 10.577s
```

On observe que le temps d'exécution est conséquemment plus grand lorsque l'on augmente la taille du graphe, soulignant la complexité polynomiale de l'algorithme de *branch-and-bound*.

5 Conclusion

Le problème du voyage de commerce est un problème d'optimisation couramment rencontré dans la vie réelle, ainsi une implémentation efficace et optimale de sa résolution est important. L'algorithme de *branch-and-bound* permet d'obtenir

un résultat optimal mais son coût en calcul augmente exponentiellement avec la taille du problème à résoudre.

Pour aller plus loin, il sera intéressant de modifier la sortie dans le terminal afin de rendre la matrice d'adjacence plus lisible dans le cas de grands graphes. Étant symétrique, il serait également utile d'optimiser son utilisation mémoire en ne stockant que la partie triangulaire inférieure ou supérieure. On pourrait aussi implémenter une méthode de *branch-and-cut*, généralement moins coûteuse que la méthode de séparation et évaluation. Enfin, il serait intéressant de tenter de paralléliser l'algorithme au moyen de MPI et/ou d'OpenMP, afin de mieux exploiter l'architecture multi-coeurs des ordinateurs modernes.