
Rapport Master M1

Master Calcul Haute Performance Simulation (CHPS)

# Implémentation d'algorithme de Résolution du Problème du voyageur de commerce avec la Méthode de Regret

Réalisé par: Aicha Maaoui

Mai 2022

Institut des Sciences et Techniques des Yvelines (ISTY)

# Contents

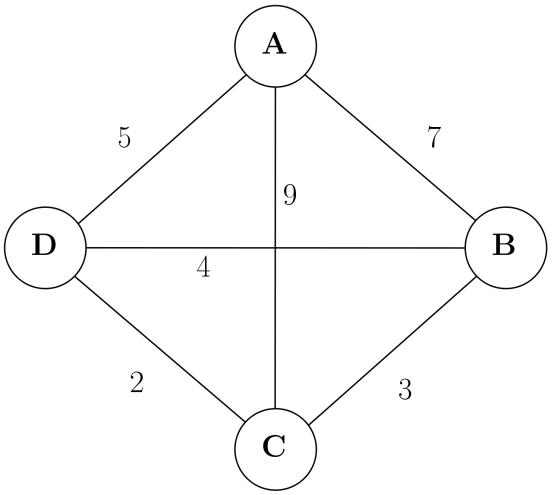
# List of Figures

# List of Tables

## 0.1 Description du problème

Le problème de voyageur de commerce est un problème classique qui consiste à trouver le trajet optimal pour parcourir une liste de villes. Le trajet entre chaque deux villes a un certain coût, et le but est de trouver un trajet qui parcourt toutes les villes et qui minimise le coût total. Le problème peut être représenté par un graphe complet, où les nœuds représentent les villes. On peut prendre un exemple avec 4 villes nommées de A à D dont le graphe complet et la matrice de coût sont illustrés par la figure 1.

La matrice de coût dans cet exemple est symétrique. Il est envisageable d'avoir une matrice de coût non symétrique si par exemple le entre deux deux villes ne coûte pas de même manière dans un sens ou dans l'autre. Dans ce projet on va implémenter un algorithme basé sur la méthode de regret pour résoudre ce problème.




| | **A** | **B** | **C** | **D** |
|---|---|---|---|---|
| **A** | $\infty$ | 7 | 9 | 5 |
| **B** | 7 | $\infty$ | 3 | 4 |
| **C** | 9 | 3 | $\infty$ | 2 |
| **D** | 5 | 4 | 2 | $\infty$ |

Figure 1: Exemple de graphe complet et matrice de coût.

## 0.2 Description de l'algorithme de résolution

Afin d'illustrer l'algorithme de résolution, on va résoudre petit à petit le problème donné par la matrice de coût donnée par la figure 1. La résolution repose sur la manipulation de la matrice de coût accompagnée par la construction d'un arbre de recherche sur plusieurs étapes. On va reprendre le problème précédent (figure 1), afin d'illustrer les différentes étapes comme le montre la figure 2.

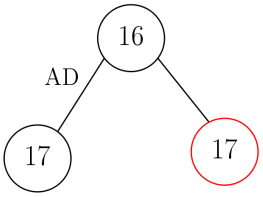
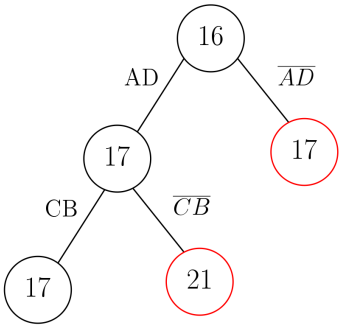
<table>
<tr><th></th><th>Arbre de recherche</th><th>Matrice de recherche</th></tr>
<tr><td>Etape 0</td><td> 
</td><td>
<table>
<tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>min</th></tr>
<tr><td>A</td><td>∞</td><td>7</td><td>9</td><td>5</td><td>5</td></tr>
<tr><td>B</td><td>7</td><td>∞</td><td>3</td><td>4</td><td>3</td></tr>
<tr><td>C</td><td>9</td><td>3</td><td>∞</td><td>2</td><td>2</td></tr>
<tr><td>D</td><td>5</td><td>4</td><td>2</td><td>∞</td><td>2</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>12</td></tr>
</table>
<table>
<tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th></tr>
<tr><td>A</td><td>∞</td><td>2</td><td>4</td><td>0</td></tr>
<tr><td>B</td><td>4</td><td>∞</td><td>0</td><td>1</td></tr>
<tr><td>C</td><td>7</td><td>1</td><td>∞</td><td>0</td></tr>
<tr><td>D</td><td>3</td><td>2</td><td>0</td><td>∞</td></tr>
<tr><td>min</td><td>3</td><td>1</td><td>0</td><td>0</td><td>4</td></tr>
</table>
<table>
<tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th></tr>
<tr><td>A</td><td>∞</td><td>1</td><td>4</td><td>0(1)</td></tr>
<tr><td>B</td><td>1</td><td>∞</td><td>0(1)</td><td>1</td></tr>
<tr><td>C</td><td>4</td><td>0(1)</td><td>∞</td><td>0(0)</td></tr>
<tr><td>D</td><td>0(1)</td><td>1</td><td>0(0)</td><td>∞</td></tr>
</table>
</td></tr>
<tr><td>Etape 1</td><td> 
</td><td>
<table>
<tr><th></th><th>A</th><th>B</th><th>C</th></tr>
<tr><td>A</td><td>1</td><td>∞</td><td>0(1)</td></tr>
<tr><td>B</td><td>4</td><td>0(5)</td><td>∞</td></tr>
<tr><td>C</td><td></td><td>1</td><td>0(1)</td></tr>
<tr><td>min</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
</table>
<table>
<tr><th></th><th>A</th><th>B</th><th>C</th></tr>
<tr><td>B</td><td>0(4)</td><td>∞</td><td>0(1)</td></tr>
<tr><td>C</td><td>4</td><td>0(5)</td><td>∞</td></tr>
<tr><td>D</td><td></td><td>1</td><td>0(1)</td></tr>
</table>
</td></tr>
<tr><td>Etape 2</td><td> 
</td><td>
<table>
<tr><th></th><th>A</th><th>C</th></tr>
<tr><td>B</td><td>0</td><td></td></tr>
<tr><td>D</td><td></td><td>0</td></tr>
</table>
</td></tr>
</table>

Figure 2: Description de l'algorithme de résolution.

- **Etape 0:** On commence par chercher la valeur minimale dans chaque ligne et les soustraire. Ensuite, on fait de même pour les colonnes. En parallèle, on commence par construire un arbre de recherche binaire. Le poids du nœud racine est égale la somme des valeurs minimales sur les lignes et les colonnes. La matrice de coût obtenue est dite "réduite". Les cases avec les valeurs zéro présentent des trajets potentiels à choisir. Pour ces cases, on va calculer les regrets en prenant la valeur minimale de la ligne et de la colonne. On va choisir le trajet qui correspond à la valeur de regret maximale. Si plusieurs cases ont la valeur maximale, on peut choisir parmi eux le trajet de manière arbitraire. Ici par exemple, on a choisi le trajet $AD$.

- **Etape 1:** On va inclure deux nœuds dans l'arbre de recherche, une qui va inclure le trajet choisi à l'étape précédente et un autre qui va l'exclure. Le nœud qui correspond au trajet exclu prend le poids du nœud parent plus la valeur du regret. On supprime de la matrice de coût la ligne et la colonne du trajet. On supprime aussi les cases qui mènent à la fermeture du trajet avec un court-circuit. Ici par exemple, on supprime le trajet DA pour ne pas avoir $AD \rightarrow DA$. On procède ensuite à la réduction de la matrice comme à l'étape précédente. On obtient le regret maximal pour le trajet $CB$. Le poids du nœud qui inclut le trajet le trajet à l'étape précédente est égale au poids du nœud parent plus la somme des minima des lignes et des colonnes.

- **Etape 2:** On ajoute deux nœuds pour inclure et exclure le trajet $CB$. Dans notre exemple, on exclut la case $BC$ pour ne pas avoir la fermeture $CB \rightarrow BC$. Il ne reste plus que deux trajets qu'on va les inclure dans notre chemin.

  **Vérification:** Après l'arrivée au bout du chemin, on a une solution au problème. Le coût de ce chemin est égal poids du dernier nœud. Il faut vérifier maintenant que c'est la meilleure solution. Pour cela il faut remonter dans l'arbre de recherche et comparer le coût de ce chemin avec les poids des différents nœuds exclus. S'il y a un nœud exclut avec un poid inférieur au coût de notre chemin, alors il y a potentiellement une solution plus optimale et il faut refaire la recherche à partir de ce nœud exclut. La vérification s'arrête si on vérifie tous les nœuds exclus, c'est-à-dire on remonte jusqu'au nœud racine.

  **Solution:** Maintenant on a une suite de trajet, il faut les ordonnée en commençant par n'importe quel ville pour obtenir un chemin fermé. Par exemple dans notre cas illustratif: $AD \rightarrow DC \rightarrow CB \rightarrow BA$.

## 0.3 Description de l'implémentation

L'algorithme de résolution décrit précédemment a été implémenté en C. L'idée principale de la résolution repose sur la récursivité pour parcourir un arbre binaire. La figure 3 présente le flowchart de la fonction ***main*** du programme.

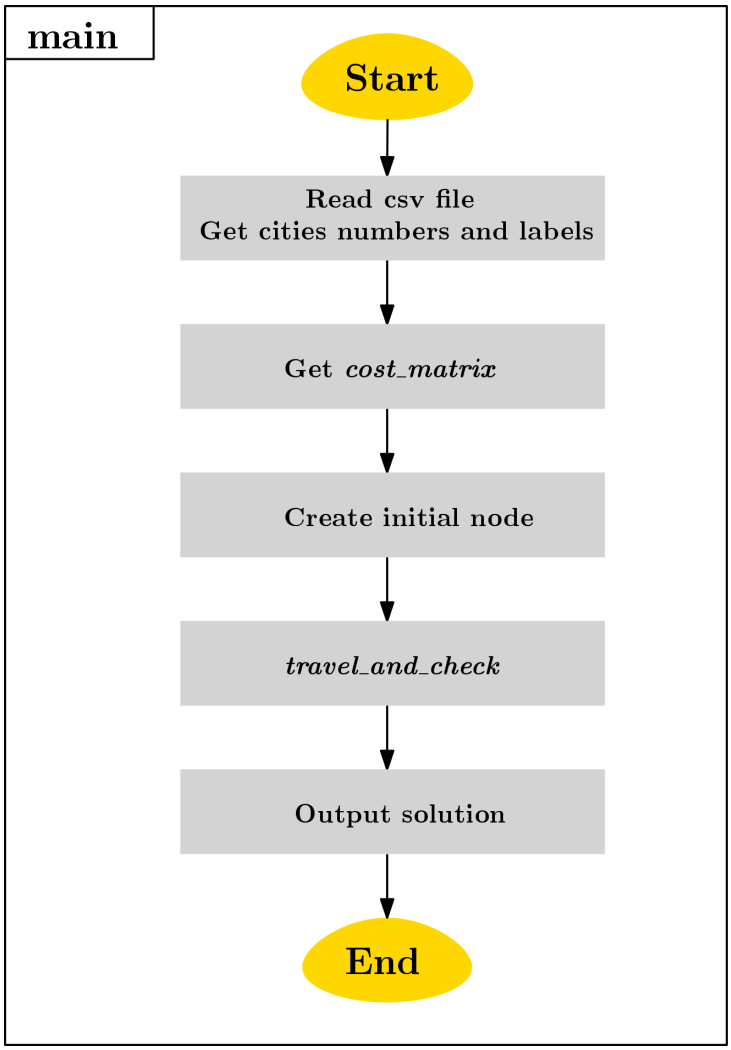



Figure 3: Flowchart de la fonction ***main*** du programme.

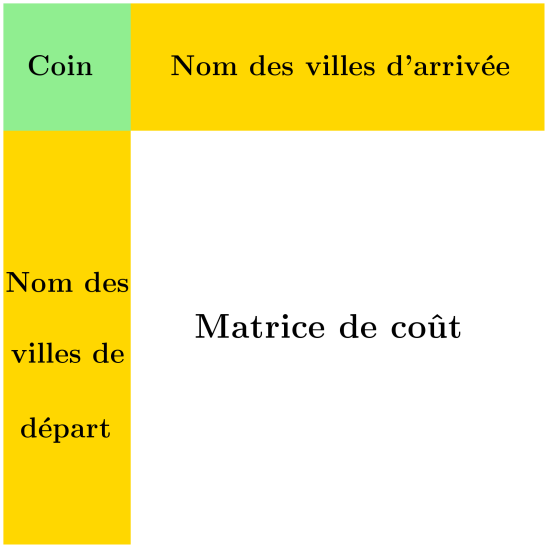
### 0.3.1 Format du fichier d'entrée

Le programme commence par lire le problème à partir d'un fichier csv facile à visualiser et à modifier. Le nom du fichier doit être donné en argument lors du lancement du programme, comme le montre le listing 1.

Listing 1: Commande pour lancer le programme.

```
1 $ ./regret exemple.csv
```

La figure 4 illustre la composition de ce fichier et une capture d'un exemple ouvert avec un logiciel de tableur.




(a) Composition du tableau à fournir.

| | Standard | Standard | Standard | Standard | Standard |
|---|---|---|---|---|---|
| 1 | coin | A | B | C | D |
| 2 | A | -1 | 7 | 9 | 5 |
| 3 | B | 7 | -1 | 3 | 4 |
| 4 | C | 9 | 3 | -1 | 2 |
| 5 | D | 5 | 4 | 2 | -1 |

(b) Capture de la matrice de coût fournie dans exemple.csv.

Figure 4: Composition et exemple du fichier à fournir.

Le nombre de villes est quelconque, ainsi que le nombre de caractères dans le nom des villes. Aucune limite n'a été spécifiée lors de la lecture des lignes du fichier. Tout est automatiquement déterminé. Cependant l'utilisateur doit respecter trois conditions :

- **Condition 1:** L'ordre des villes de départ et de d'arrivée doit être le même.
- **Condition 2:** Les valeurs de coût doivent être des entiers.
- **Condition 3:** La diagonale de la matrice de coût doit être remplie de $-1$.

La lecture du fichier d'entrée se fait en deux étapes. D'abord la fonction *get_cities_label* parcourt les noms des villes de départ et d'arrivée. Cette fonction vérifie qui est mis dans le bon ordre, ensuite elle retourne le nombre des villes et leurs noms. Si une incohérence est trouvée dans l'ordre des villes, le programme s'arrête avec le message d'erreur suivant **"Inconsistant cities labels in the cost matrix"**. Ensuite le fichier sera lu encore une fois par la fonction *get_cost_matrix* qui retourne la matrice de coût. Si une valeur de la diagonale est différente de

$-1$, un message d'avertissement s'affiche et la valeur sera imposé à $-1$.

### 0.3.2 Structure du Node

Après la lecture des données d'entrée, Le nœud racine de l'arbre de recherche sera initialisé. Un nœud est implémenté dans une structure Node dont les attributs et leurs types sont les suivants:

- **int type:** Le type du nœud (ROOT, INCLUSIVE ou EXCLUSIVE qui sont des constantes définies).
- **int nbr_cities:** Le nombre de villes qui restent à entrer (et aussi à sortir).
- **int *row_cities:** Un tableau d'entier de taille nbr_cities, contenant les indices les villes de départ qui restent encore dans l'algorithme.
- **int *column_cities:** Un tableau d'entier de taille nbr_cities, contenant les indices les villes de destinations qui restent encore dans l'algorithme.
- **int **cost_matrix:** La matrice de coût de taille (nbr_cities $\times$ nbr_cities).
- **int **regret_matrix:** La matrice de regret de taille (nbr_cities $\times$ nbr_cities).
- **int weight:** Le poids du nœud.
- **int nbr_paths:** Le nombre de trajets inclus du nœud racine jusqu'à ce noeud.
- **int **current_paths:** Un tableau de taille nbr_paths contenant les trajets incluses du nœud racine jusqu'à ce nœud. Chaque élément du tableau est un tableau de deux entiers indiquant respectivement les indices des villes de départ et d'arrivée.
- **Node *mother:** Un pointeur vers le nœud parent.
- **struct Node *inclusive_mother:** Un pointeur vers la "fille" qui inclue le trajet sélectionné.
- **struct Node *exclusive_mother:** Un pointeur vers la "fille" qui exclue le trajet sélectionné.

La déclaration et l'initialisation de la racine se fait avec la fonction *initial_node*, qui prend en argument le nombre de villes à visiter et la matrice de coût. La mémoire est allouée pour les attributs suivants : *row_cities*, column_cities, *cost_matrix* et *regret_matrix*. *row_cities* et *column_cities* sont remplis de 0 jusqu'à *nbr_cities-1* en ordre croissant. On rappelle que les vrais noms des villes ne sont pas passés dans l'algorithme de l'arbre de recherche, mais seulement leurs indices. La matrice de coût entrée par l'utilisateur est copiée dans la matrice de coût

attachée au nœud racine. Le type du nœud est évidemment mis sur ROOT. Les attributs weight et *nbr_paths* sont initialisés à zéro. L'attribut *current_paths* est omis de l'initialisation de la racine.

### 0.3.3 *Travel_and_check*

Après l'initialisation du nœud racine, on peut commencer l'algorithme de recherche du chemin optimal. Tout l'algorithme est implémenté principalement dans la fonction récursive *travel_and_check*. Cette fonction, illustrée dans le flowchart de la figure 5, prend en argument un nœud et suit l'évolution du meilleur coût (*best_cost*) et le meilleur chemin (*best_path*), les deux aussi en argument. La chaîne de récursivité se termine si l'algorithme de recherche revient au nœud racine, ce qui veut dire que toutes les possibilités ont été vérifiées.

La création d'un nœud, autre que la racine, se fait grâce à la fonction *add_node*. Cette fonction prend en argument le nœud parent (mother) ainsi que le type du nœud à créer (INCLUSIVE ou EXCLUSIVE). L'initialisation est différente s'il s'agit d'un type ou de l'autre. S'il s'agit du type INCLUSIVE, la fonction *add_node* prend en plus en argument le trajet choisi à l'étape précédente. S'il s'agit du type EXCLUSIVE, la fonction prend en plus le regret calculé à l'étape précédente. On alloue la mémoire aux attributs suivants : *row_cities*, *column_cities*, *cost_matrix*, *regret_matrix* et *current_paths*. La taille de la mémoire à attribuer pour chacun de ces attributs va dépendre de *nbr_cities* et *nbr_paths*. L'initialisation des différents attributs est faite comme l'indique le tableau suivant:

| **Attribut** | **INLUSIVE** | **EXCLUSIVE** |
|---|---|---|
| *nbr_paths* | *mother* $\rightarrow$ *br_paths* + *1*, on ajoute celui qu'on a sélectionné à l'étape précédente | *mother* $\rightarrow$ *nbr_paths* |
| *nbr_cities* | *mother* $\rightarrow$ *nbr_cities*–*1*, on a une ville au moins à visiter | *mother* $\rightarrow$ *nbr_cities* |
| *row_cities* et *column_cities* | *mother* $\rightarrow$ *row_cities* et *mother* $\rightarrow$ *column_cities* | *mother* $\rightarrow$ *row_cities* et *mother* $\rightarrow$ *column_cities* |
| *cost_matrix* | *mother* $\rightarrow$ *cost_matrix*, sauf la ligne et la colonne correspondantes au trajet choisi à l'étape précédente | *mother* $\rightarrow$ *cost_matrix* |
| *regret_matrix* | Non initialisé | *mother* $\rightarrow$ *regret_matrix* |
| *current_paths* | *mother* $\rightarrow$ *current_paths* | Plus le trajet choisi à l'étape précédente |
| weight | *mother* $\rightarrow$ *weight* | *mother* $\rightarrow$ *weight* + *regret* |

Table 1: L'initialisation des différents attributs du noeud en fonction de son type.

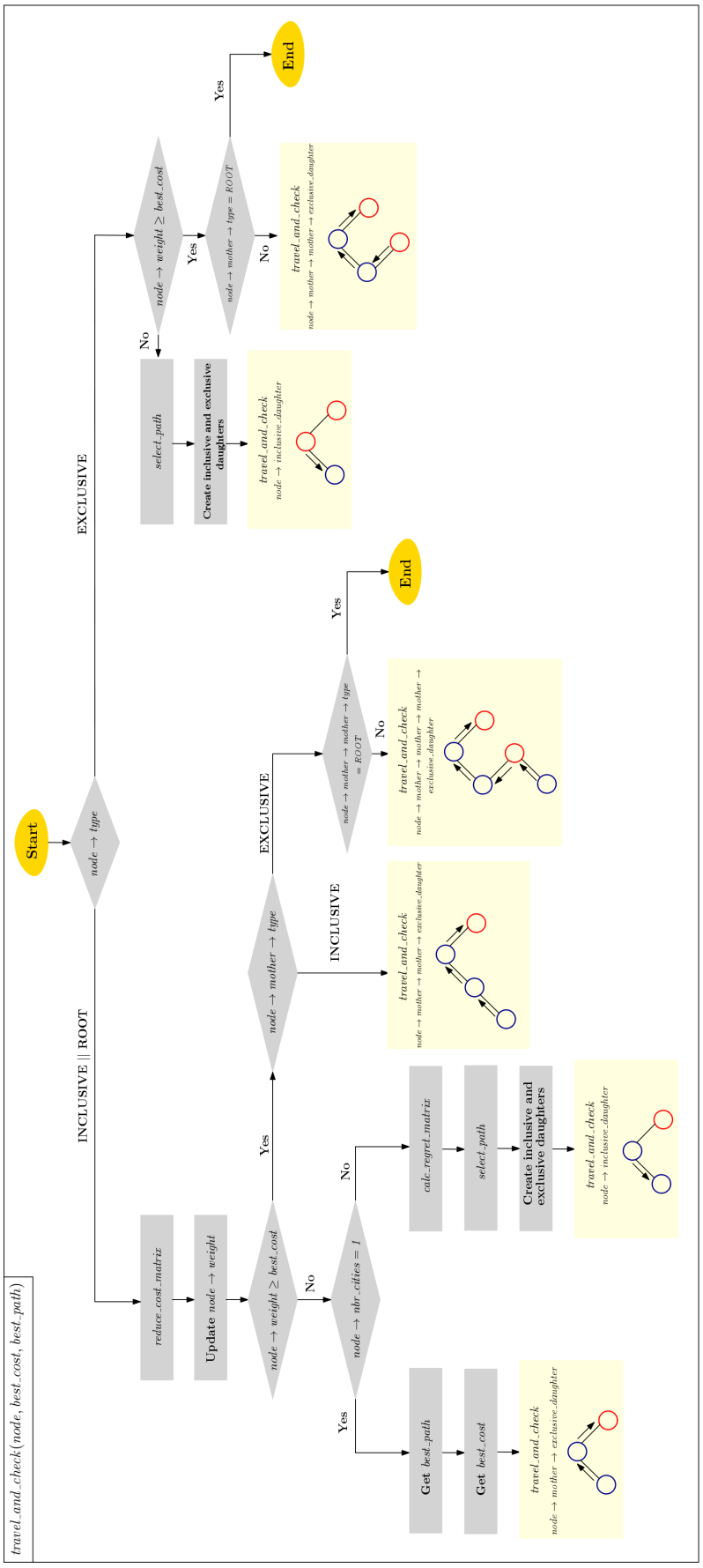



Figure 5: Flowchart de la fonction récursive *travel_and_check*.

La fonction *reduce_cost_matrix* réduit la matrice de coût. Pour chaque ligne, on cherche la valeur minimale et on soustrait cette valeur de toute la ligne. Les valeurs $-1$ ne sont pas prises en compte. Ensuite on répète la même opération pour les colonnes. La fonction *calc_regret_matrix* prend en argument la matrice de coût réduit et retourne la matrice des regrets. Le regret est calculé pour un trajet si seulement le coût réduit est égal à zéro. Les autres trajets lui sont attribués la valeur $-1$ dans la matrice des regrets, c'est-à-dire qu'ils ne font pas partie des trajets potentiels pour ce nœud dans l'arbre de recherche. Les regrets pour chaque trajet sont calculés par la fonction *calc_regret*. C'est la somme des valeurs minimales de la ligne et de la colonne auxquelles la case appartient. Les valeurs $-1$ ne sont pas prises en compte. Si la colonne ou la ligne ne contient aucune valeur, c'est la valeur zéro qui est prise en compte.

La fonction *select_path* prend en entrée la matrice des regrets et effectue les tâches suivantes:

- Recherche le regret maximal et retour de sa valeur,
- Retour aussi les indices des villes de départ et d'arrivée correspondant au regret maximal,
- Transformation du regret maximal en $-1$ dans la matrice des regrets d'exclure ce trajet définitivement dans la suite de l'arbre de recherche

Si plusieurs trajets ont la valeur du regret maximal, le trajet retourné et choisi de manière arbitraire parmi eux. De manière pratique, il s'agit du premier trajet trouvé.

### 0.3.4 Sortie du trajet optimal

Lorsque l'algorithme de recherche se termine et le trajet optimal est trouvé, la solution est affichée dans le terminal en utilisant les noms des villes comme le montre l'exécution de l'exemple du problème pris précédemment illustré dans la figure 6.

```
aicha@aicha-Vostro-3500:/media/aicha/DATA/regret_projet/regret$ make clean
rm -Rf *~ regret main input travel node *.o
aicha@aicha-Vostro-3500:/media/aicha/DATA/regret_projet/regret$ make all
gcc  -Wall -c input.c -o input
gcc  -Wall -c node.c -o node
gcc  -Wall -c travel.c -o travel
gcc  -Wall -c main.c -o main
gcc  -Wall -o regret main input travel node
aicha@aicha-Vostro-3500:/media/aicha/DATA/regret_projet/regret$ ./regret exemple.csv
****** Lecture du problème à partir de "exemple.csv" ******
Lecture du problème avec succès. Le nombre de villes est 4 .
****** Début de la résolution avec la méthode des regrets ******
Terminaison de la résolution avec succès
Le chemin optimal est:
B -> A -> D -> C -> B
****** Fin de la résolution avec la méthode des regrets  ******
```

Figure 6: Résultat de l'exécution avec le fichier exemple.csv.

Et ainsi, on a un programme qui permet de facilement lire le problème du voyageur de commerce et le résoudre.

## 0.4 Conclusion

Au cours de ce projet, on a pu résoudre le problème de voyageur de commerce grâce à la méthode de regret. L'algorithme repose sur le parcours d'un arbre de recherche binaire. On a implémenté cet algorithme dans un programme en c, qui se repose sur la récursivité. On a essayé de proposer une implémentation générique avec la lecture facile du problème à partir d'un fichier csv fourni par l'utilisateur. La solution optimale est affichée sur le terminal comme une chaîne fermée des villes parcourues.

# Appendix A

# Dépôt GitHub

Le code associé à ce rapport est déposé dans le dépot git **Methode_De_Regret**. Le code SSH de ce dépot est le suivant:

**git@github.com:Chaichas/Methode_De_Regret.git**