Recherche Opérationnelle 2ème année Travaux Pratiques

Filière Modélisation Mathématique et Science des Données

Implémentation de l'algorithme de Christofides

Présenté par : **LE GALL Mathis et KARINTHI Clément**

Supervisé par CHICOISNE Renaud

Année 2022/2023

Campus des Cézeaux, 1 rue de la Chébarde, TSA 60125, 63178 Aubière CEDEX

Table des matières

1 Présentation générale du problème du voyageur de commerce (TSP)

L'algorithme de Cristofides correspond à une méthode de résolution du problème du voyageur de commerce. Le voyageur de commerce a un problème, il doit passer par plusieurs villes avec une distance différente entre chaque ville le plus vite possible. Il faut donc trouver le plus court circuit en passant une fois par chaque ville.

FIGURE 1 – Exemple de problème du voyageur de commerce

2 Présentation générale de l'algorithme de Cristofides

L'algorithme de Cristofides est un algorithme conçu pour résoudre le problème du voyageur de commerce. Il a été inventé par Nico Cristofides en 1976. Il garantit de trouver un chemin de moins de 1.5 fois le chemin optimal. Pour se faire il faut d'abord trouver l'arbre couvrant de poids minimun (avec L'algorithme de Prim ici) puis créer un graphe complet avec les sommets de degré impair dans l'arbre couvrant. Pour ce graphe complet, on trouve les plus cours chemin entre les noeuds de degré impairs dans le graphe d'origine pour se faire on utilise l'algorithme de Dijsktra, ces plus courts chemins correspondront à la valeur des arètes entre les noeuds dans le graphe complet de sommets impairs. On cherche ensuite à trouver un matching parfait de poids minimum dans le graphe complet de sommets impairs. On unis l'arbre couvrant de poids minimums avec le matching puis on trouve un cycle eulerien avec les arètes avant de former un cycle hamiltonien en supprimant les doublons.

3 L'algorithme de Prim

L'algorithme de Prim est un algorithme glouton qui renvoit un arbre couvrant de poids minimums. Le code prend en entrée un pointeur vers un objet graph, qui contient une liste d'arêtes et de noeuds, ainsi que l'indice du noeud de départ (origin node) et un pointeur vers un objet tree qui sera rempli avec l'arbre couvrant de poids minimum trouvé.

Le code commence par réinitialiser toutes les arêtes et tous les noeuds pour ne pas être dans l'arbre. Ensuite, le noeud de départ est marqué comme étant dans l'arbre et l'algorithme de Prim commence.

L'algorithme de Prim consiste à ajouter répétitivement l'arête la moins chère qui relie un noeud de l'arbre à un noeud hors de l'arbre, jusqu'à ce que tous les noeuds soient dans l'arbre. Dans chaque itération, le code parcourt toutes les arêtes non marquées pour trouver l'arête la moins chère qui respecte les conditions mentionnées ci-dessus. Si une telle arête est trouvée, elle est ajoutée à l'arbre en la marquant comme étant dans l'arbre et en marquant ses noeuds comme étant dans l'arbre.

À la fin de l'algorithme, l'arbre de poids minimum est rempli dans l'objet tree en utilisant les arêtes marquées dans le graphe d'origine.

```
                (8,9)   [w=10.000000]
Starting PRIM's algorithm from node 0
XXXXXXXX tree XXXXXXXX
WEIGHT: 19.000000
Edge #2 (0,3)   cost:   0.000000
Edge #3 (0,4)   cost:   7.000000
Edge #5 (0,6)   cost:   2.000000
Edge #7 (0,8)   cost:   1.000000
Edge #16 (1,9)  cost:   4.000000
Edge #17 (2,3)  cost:   3.000000
Edge #25 (3,5)  cost:   1.000000
Edge #27 (3,7)  cost:   0.000000
XXXXXXXXXXXXXXXXXXXXX
```

FIGURE 2 – Arbre couvrant de poids minimums obtenus avec l'algorithme de Prim sur un graphe complet de taille 10

De cette arbre on va en extraire les sommets de degrés impairs soit les sommets 1,2,4,5,7,8. Il y a un sens à traiter ces sommets de degrés impairs car pour avoir un cycle hamiltonien il faut rajouter des connexions entres les arètes à degré impairs.

4 Création du graphe complet des sommets de degrés impairs

Pour choisir les arètes incidentes au sommets de degrés impairs à rajouter pour obtenir un graphe eulérien permettant d'avoir un cycle hamiltonien, on crée un graphe complet des sommets de degrés impairs. La valeur des arètes entre les sommets de degrés impairs correspond au plus court chemin (obtenu avec Dijsktra) du graphe initial. Avec ce graphe, nous pourrons réaliser un couplage minimums pour trouver les arètes à garder.

4.1 L'algorithme de Dijsktra

L'algorithme de Dijkstra est un algorithme de plus court chemin utilisé pour trouver le chemin entre deux noeuds d'un graphe pondéré. On commence par initialiser tous les noeuds comme étant non connecté à l'arbre de plus court chemin, on leur donne ainsi une distance à l'origine infinie. On visite le noeud ayant la plus petite distance par rapport à l'origine et on le déclare comme noeud visité en mettant sa distance à l'origine comme la valeur de l'arète qui les sépare. Pour chaque voisin non visité du noeud courant, calculer la distance en passant par le noeud courant. Si cette distance est plus courte que la distance actuellement stockée pour le voisin, mettre à jour la distance stockée. On répète ces étapes jusqu'à avoir soit atteint le noeud souhaité soit visité tous les noeuds. Si on attend d'avoir visité tous les noeuds on aura alors un arbre de Dijsktra consistent ou toutes les distances à l'origines seront minimales et ou les distances entre les points ne peuvent être améliorées. Cependant les distances entre les points qui ne sont pas l'origine ne sont pas minimales. En outre, ce n'est pas notre approche ici, nous voulons les distances entre les points de degrés impairs et donc pour chaque distance une fois l'arrivée trouvée on arrète l'algorithme pour des raisons de complexité et on obtient la distance du plus court chemin mais l'arbre de plus court chemin obtenu n'est pas toujours consistent car l'algorithme n'a pas terminé sa création. Ainsi, nous avons appliqué l'algorithme pour tous les couples de sommets de degrés impairs pour avoir les distances entre ces points et donc la valeur des arètes du graphe complet. Si nous avons tout de même stocker les arbres de plus courts chemins pour

chaque couple de points pour vérifier le bon déroulement nous avons changer la valeur cost de Tree pour qu'elle prenne la valeur de la distance entre notre point d'origine et le point d'arrivée.

```
Node 5 and 7
Dijkstra failed for edge 1 between nodes 2 and 0: 5.000000+10.000000 = 15.000000 < 17.000000
Dijkstra failed for edge 2 between nodes 3 and 0: 1.000000+0.000000 = 1.000000 < 17.000000
Dijkstra failed for edge 6 between nodes 7 and 0: 1.000000+15.000000 = 16.000000 < 17.000000
Dijkstra failed for edge 13 between nodes 1 and 6: 8.000000+6.000000 = 14.000000 < 15.000000
Dijkstra failed for edge 15 between nodes 1 and 8: 8.000000+5.000000 = 13.000000 < 18.000000
Dijkstra failed for edge 16 between nodes 1 and 9: 8.000000+4.000000 = 12.000000 < 13.000000
Dijkstra failed for edge 17 between nodes 3 and 2: 1.000000+3.000000 = 4.000000 < 5.000000
Dijkstra failed for edge 20 between nodes 2 and 6: 5.000000+8.000000 = 13.000000 < 15.000000
Dijkstra failed for edge 26 between nodes 3 and 6: 1.000000+6.000000 = 7.000000 < 15.000000
Dijkstra failed for edge 28 between nodes 3 and 8: 1.000000+3.000000 = 4.000000 < 18.000000
Dijkstra failed for edge 29 between nodes 3 and 9: 1.000000+10.000000 = 11.000000 < 13.000000
Dijkstra failed for edge 32 between nodes 7 and 4: 1.000000+7.000000 = 8.000000 < 18.000000
Dijkstra failed for edge 41 between nodes 9 and 6: 13.000000+1.000000 = 14.000000 < 15.000000
Dijkstra failed for edge 42 between nodes 7 and 8: 1.000000+2.000000 = 3.000000 < 18.000000
Potentials are NOT consistent. Disjktra failed.
```

FIGURE 3 – Application de l'algorithme de Dijsktra uniquement avec arrèt une fois le point d'arrivée trouvée

Ainsi, les potentiel ne sont pas consistent car le noeud d'arrivé a été trouvé sans visiter tous les noeuds.

On introduit une fonction all in tree qui permet d'arréter l'algorithme de Dijkstra seulement quand tous les noeuds ont été visités.

```
Node 4 and 1
Potentials are consistent. Disjktra ended successfully.
Node 4 and 2
Potentials are consistent. Disjktra ended successfully.
Node 4 and 5
Potentials are consistent. Disjktra ended successfully.
Node 4 and 7
Potentials are consistent. Disjktra ended successfully.
Node 8 and 1
Potentials are consistent. Disjktra ended successfully.
Node 8 and 2
Potentials are consistent. Disjktra ended successfully.
Node 8 and 5
Potentials are consistent. Disjktra ended successfully.
Node 8 and 7
Potentials are consistent. Disjktra ended successfully.
Node 1 and 2
Potentials are consistent. Disjktra ended successfully.
Node 1 and 5
Potentials are consistent. Disjktra ended successfully.
```

FIGURE 4 – Application de l'algorithme de Dijsktra jusqu'à avoir visité tous les noeuds

On obtient donc des arbres de plus courts chemins consistent avec toutes les distances à l'origine les plus courtes.

Quel que soit la méthode, les distances calculées sont les mêmes et donne bien les poids des arètes à mettre dans le graphe complet, mais calculer tous l'arbre de plus court chemin peut s'avérer couteux.

On peut ainsi avec l'arbre retracer le chemin pour aller du noeud 4 au noeud 8 : 4-¿0-¿8 de poids 7 + 1.

```
Distance between node 4 and 8 is 8.000000
XXXXXXXX tree XXXXXXXX
WEIGHT: 8.000000
Edge #3 (0,4)   cost:   7.000000
Edge #7 (0,8)   cost:   1.000000
Edge #16 (1,9)  cost:   4.000000
Edge #17 (2,3)  cost:   3.000000
Edge #27 (3,7)  cost:   0.000000
Edge #32 (4,7)  cost:   7.000000
Edge #34 (4,9)  cost:   7.000000
Edge #36 (5,7)  cost:   1.000000
Edge #41 (6,9)  cost:   1.000000
XXXXXXXXXXXXXXXXXXXXX
```

FIGURE 5 – Distance entre deux noeuds de degrés impairs, weight correspond aussi à la distance et non au poids de l'arbre

4.2 Fonction de création du graphe complet

La fonction de création du graphe complet prend en entrée le tableau des arbres de Dijkstra ou chaque arbre donne la distance entre deux points définis, un graphe Kn qui sera alloué et représente le graphe complet des noeuds de degrés impairs et le tableau des noeuds de degrés impairs nécessaire à l'instanciation. On alloue d'abord l'espace mémoire des noeuds et arètes du graphe complet. Puis, pour chaque paire de noeud on y met la valeur de cost de l'arbre de Dijkstra correspondant en y mettant. La fonction met aussi à jour le degré de chaque noeud. On stocke aussi la valeur maximale des arètes dans un attribut qui s'appelle max edge cost.

On retrouve donc les même sommets de degrés impairs avec en valeurs des arètes la distance minimale entre les couples de points.

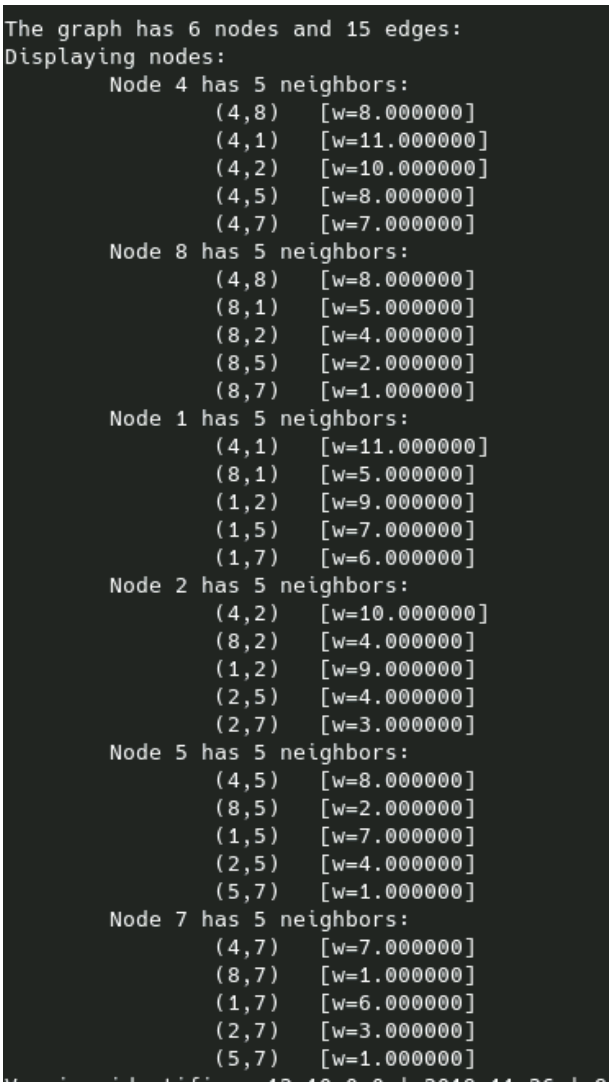

FIGURE 6 – Graphe complet des sommets de degrés impairs de l'arbre couvrant de poids minimums sur un graphe de départ complet de taille 10 K10

5 Couplage parfait de poids minimums dans le graphe complet de degrés impairs

On va effectuer un couplage parfait de poids minimums dans le graphe complet de noeuds de degrés impairs pour trouver les arètes à rajouter nécessaire à la création d'un tour Eulerien. Pour ce faire, on utilise la bibliothèque CPLEX un solveur pour les problèmes d'optimisation mathématique.

La fonction min weight perfect matching prend en entrée un pointeur vers une structure de données de graphe Gptr et un pointeur vers une structure de données d'arête M, qui stockera les arêtes de l'appariement. La fonction renvoie une valeur entière qui est la valeur optimale de la fonction objectif.

Tout d'abord, la fonction initialise une structure de problème IP problem ip prob qui stockera toutes les données requises par CPLEX pour résoudre le problème. Ensuite, elle crée un environnement CPLEX en appelant la fonction CPXopenCPLEX. Si l'environnement ne peut pas être créé, le programme quitte.

Ensuite, la fonction crée un problème MIP (Programmation mixte en nombres entiers) en appelant la fonction CPXcreateprob. Elle définit ensuite certains paramètres en utilisant les fonctions CPXsetintparam et CPXsetdblparam.

La fonction génère alors des variables et des contraintes nécessaires pour le problème d'appariement parfait en utilisant les données du graphe d'entrée. En particulier, elle crée une variable binaire xe pour chaque arête dans le graphe, et génère des contraintes pour s'assurer que chaque noeud dans le graphe est apparié avec exactement un autre noeud. La fonction objectif est définie pour minimiser la somme des poids des arêtes incluses dans l'appariement.

Enfin, la fonction appelle CPXmipopt pour résoudre le problème, et récupère la solution optimale en appelant CPXgetx. La fonction stocke les arêtes appariées dans le tableau M, et renvoie la valeur optimale de la fonction objectif.

```
Version identifier: 12.10.0.0 | 2019-11-26 | 843d4de
CPXPARAM_Read_DataCheck                          1
Found incumbent of value 18.000000 after 0.01 sec. (0.00 ticks)
Found incumbent of value 16.000000 after 0.01 sec. (0.00 ticks)
Tried aggregator 1 time.
Reduced MIP has 6 rows, 15 columns, and 30 nonzeros.
Reduced MIP has 15 binaries, 0 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.01 sec. (0.02 ticks)
Probing time = 0.00 sec. (0.01 ticks)
Tried aggregator 1 time.
Detecting symmetries...
Reduced MIP has 6 rows, 15 columns, and 30 nonzeros.
Reduced MIP has 15 binaries, 0 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.00 sec. (0.02 ticks)
Probing time = 0.00 sec. (0.01 ticks)
Clique table members: 6.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 32 threads.
Root relaxation solution time = 0.01 sec. (0.01 ticks)

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

*     0+    0                           16.0000        0.0000           100.00%
      0     0        cutoff             16.0000                     4    0.00%

Root node processing (before b&c):
  Real time             =    0.05 sec. (0.10 ticks)
Parallel b&c, 32 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.05 sec. (0.10 ticks)
Edge #1 ((4,1) [11.000000]) is used in the perfect matching
Edge #6 ((8,2) [4.000000]) is used in the perfect matching
Edge #14 ((5,7) [1.000000]) is used in the perfect matching
```

FIGURE 7 – Interface d'utilisation de Cplex, rend compte du traitement et donne le couplage parfait minimum trouvé en fin d'éxécution sur le même exemple que précedemment avec les noeuds de degrés impairs 1,2,4,5,7,8

6 Union du couplage parfait et de l'arbre couvrant de poids minimum

L'union du couplage parfait et de l'arbre couvrant de poids minimum est réalisée à l'aide de la fonction union matching tree. Cette union permet de créer un nouveau graphe qui combine les arêtes de l'arbre couvrant minimum et du matching parfait minimum créant les conditions nécessaires pour construire un cycle eulérien.

La fonction prend en entrée un pointeur vers une structure de graphe Gptr, un arbre T, une liste d'arêtes M et la taille de la liste de M. Il crée un nouveau graphe Gptr qui est l'union de l'arbre T et de la liste d'arêtes M.

Cette fonction commence par initialiser les noeuds de Gptr en leur attribuant un identifiant unique, en marquant qu'ils ne sont pas dans l'arbre T, en initialisant leur degré et un degré temporaire à 0. Ensuite, il crée les arêtes de Gptr en copiant les arêtes de T et de M, en mettant à jour leur coût et leur marquage comme étant dans l'arbre T ou pas, et en mettant à jour les degrés des noeuds adjacents à chaque arête.

Le code alloue de la mémoire pour les listes d'arêtes adjacente à chaque noeud de Gptr, et met à jour ces listes pour chaque noeud en utilisant les arêtes du graphe.

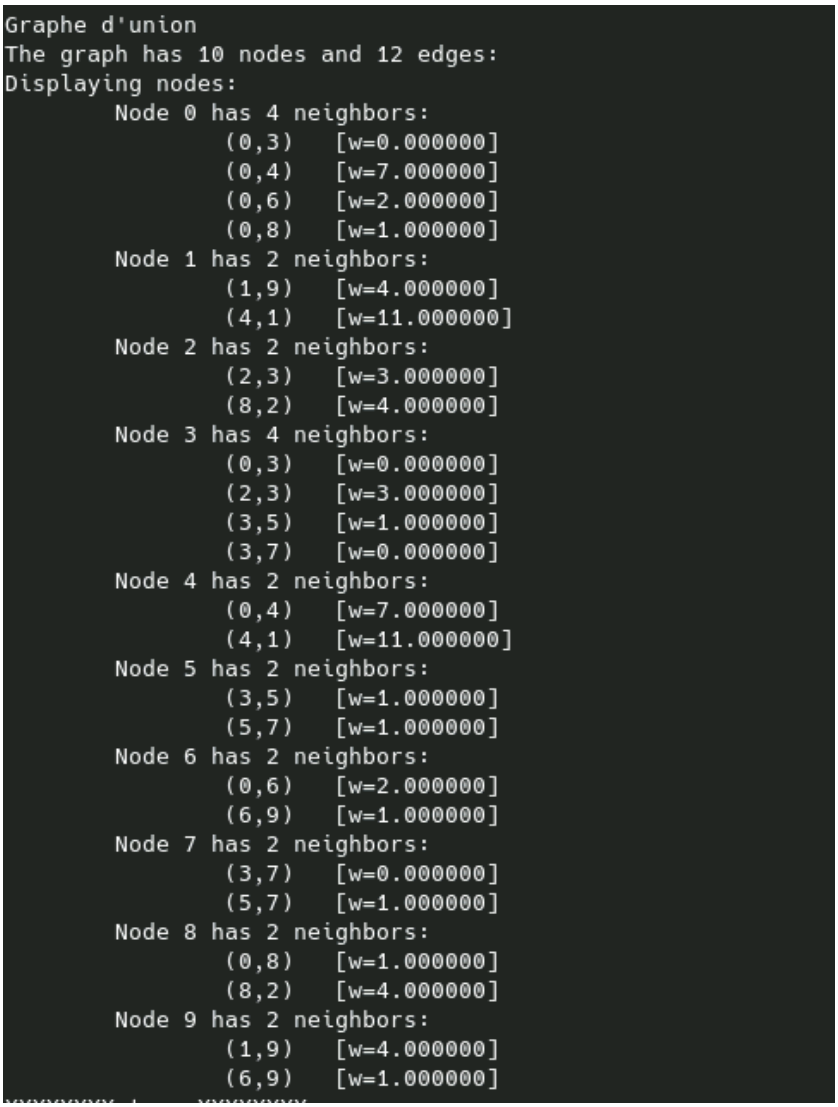

FIGURE 8 – Graphe d'union de l'arbre de poids minimum et du couplage de poids minimums on voit bien le rajout des 3 arètes selectionnées par le couplage

7 Cycle Eulérien

Afin d'obtenir un cycle eulérien sur l'union du couplage parfait et de l'arbre couvrant de poids minimum, nous allons utiliser l'algorithme de Fleury. Il faut tout d'abord savoir si une arête est un pont pour le bon fonctionnement de l'algorithme. Pour ce faire, nous avons rajouté dans la structure edge un entier displayed pour savoir si cette arête est supprimé ou non de la structure graph/tree. Cet entier vaut 1 si l'arête existe toujours, 0 sinon.
Nous utilisons une première fonction *is_bridge*. Cette fonction prend en paramètre un pointeur vers une structure de graphe Gptr, un pointeur vers une structure de edge e, un sommet défini par un entier s, et un pointeur sur le nombre de noeuds accessibles représenté par nbnodes. On supprime premièrement l'arête du graphe et on choisit un des deux sommets du l'arête supprimée pour être le sommet de départ. Si l'un des deux sommets se retrouve exclus du graphe, nous choisissons l'autre sommet. Ensuite, nous faisons un $DFS\_explore$ à partir du sommet de départ sélectionné et nous comptons le nombre de sommets visités après application du $DFS\_explore$. Si le nombre de sommets visités est égal au nombre de sommets accessibles dans le graphe alors l'arête n'est pas un pont. Sinon l'algorithme n'a pas exploré le graphe entièrement et

possède donc plusieurs composantes connexes après supression de l'arête, ce qui signifie que cet arête est un pont.
Le chemin eulérien est stocké dans une structure tree. La fonction fleury prend en paramètre un pointeur vers une structure de graphe Gptr, un pointeur vers une structure d'arbre T, et un sommet de départ s. Tant que le graphe n'est pas vide (il est vide si toutes les arêtes ne sont pas displayed), nous choisissons une arête incidente au sommet choisi. Si cette arête n'est pas un pont, nous l'ajoutons au tableau d'arêtesincludegraphics[width1cm de l'arbre, nous choisissons le noeud incident comme sommet de départ et nous mettons à jour le coût du chemin. Sinon, nous passons à l'arête suivante.

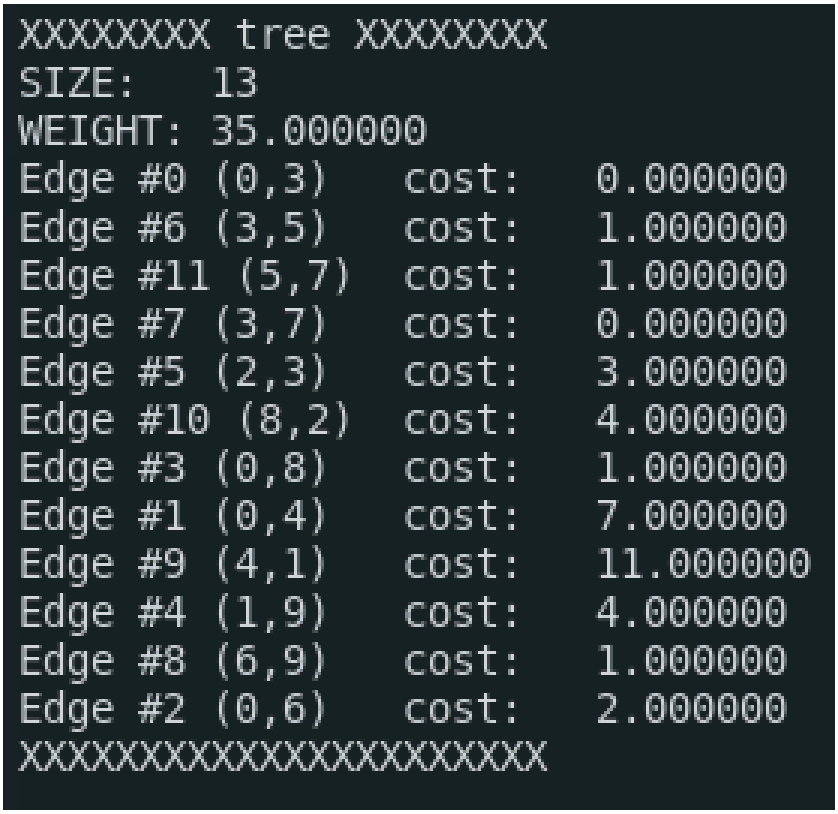

FIGURE 9 – Chemin eulérien de l'union du couplage parfait et de l'arbre couvrant de poids minimum à partir du sommet 0

8 Cycle Hamiltonien

A partir du cycle eulerien on crée un cycle hamiltonien qui passe par tous les noeuds une seule fois. L'algorithme consiste à parcourir le cycle eulérien et supprimer les doublons pour qu'on ne visite chaque noeud qu'une fois (il en sauvegarde cependant le cout). L'algorithme prend en entrée le graphe G de départ, ainsi que deux arbres (cycleE et cycleH) qui seront utilisés pour stocker le cycle Eulerien obtenu précédemment (un cycle qui passe par chaque arête du graphe exactement une fois) et le cycle Hamiltonien. L'algorithme utilise le fait que les arètes du cycle Eulerien soient triés dans l'ordre de parcours pour crée un cycle Hamiltonien lui aussi trié. (Attention le noeud de départ change au cours de l'algorithme pour prendre la valeur du dernier noeud visité, soit le nouveau départ de la prochaine arète du graphe).

L'algorithme commence par initialiser tous les noeuds du graphe à n'être pas dans l'arbre de cycle (isintree = 0). Il sélectionne ensuite le premier noeud du cycle Eulerien (cycleE) et l'ajoute à l'arbre de cycle (isintree = 1).

Ensuite, pour chaque arête du cycle Eulerien (à l'exception de la dernière), l'algorithme vérifie si le noeud actuel est l'extrémité i ou j de l'arête. S'il s'agit de i et que j n'est pas déjà dans l'arbre de cycle, l'algorithme

ajoute une arête entre le noeud de départ i et j (en stockant le coût total de l'arête) et marque j comme faisant partie de l'arbre de cycle. Les noeuds actuel et de départ deviennent alors j. Si le noeud actuel est déjà dans l'arbre de cycle, l'algorithme ajoute simplement au coût local et avance le noeud courant sans avancer le noeud de départ.

Si le noeud actuel est j et que i n'est pas déjà dans l'arbre de cycle, l'algorithme ajoute une arête entre le nœud de départ et i (en stockant le coût total de l'arête) et marque i comme faisant partie de l'arbre de cycle. Les noeuds actuel et de départ deviennent alors i. Si le noeud actuel est déjà dans l'arbre de cycle, l'algorithme ajoute simplement le coût de l'arête au coût local et avance le noeud courant sans avancer le noeud de départ.

Si le noeud actuel n'est ni i ni j, cela signifie qu'il y a une erreur dans la création du cycle Hamiltonien ou que le cycle Eulerien d'entrée était mal instancié, l'algorithme affiche alors un message d'erreur et sort de la boucle.

Après avoir parcouru toutes les arêtes sauf la dernière du cycle Eulerien, l'algorithme ajoute la dernière arête du cycle Eulerien au coût total de l'arbre. Il crée ensuite une arête entre le dernier noeud visité et le premier noeud du graphe (en stockant le coût total de l'arête) et ajoute cette arête à l'arbre de cycle Hamiltonien.

Enfin, l'algorithme retourne le cycle Hamiltonien (le poids est toujours le même que celui du cycle Eulérien).

```
XXXXXXXX tree XXXXXXXX
SIZE:   11
WEIGHT: 35.000000
Edge #0 (0,3)   cost:   0.000000
Edge #6 (3,5)   cost:   1.000000
Edge #11 (5,7)  cost:   1.000000
Edge #5 (7,2)   cost:   3.000000
Edge #10 (8,2)  cost:   4.000000
Edge #1 (8,4)   cost:   8.000000
Edge #9 (4,1)   cost:   11.000000
Edge #4 (1,9)   cost:   4.000000
Edge #8 (6,9)   cost:   1.000000
Edge #2 (6,0)   cost:   2.000000
XXXXXXXXXXXXXXXXXXXXXX
```

FIGURE 10 – Chemin hamiltonien sur un graphe initial complet de taille 10

9 Conclusion

Pour conclure, le voyageur de commerce n'a plus de problème, il a désormais un algorithme qui donne avec un graphe d'entrée de villes et de distances un trajet satisfaisant infèrieur à 1.5 fois la solution optimale quel que soit la compléxité du graphe d'entrée. La réalisation de cet algorithme très complet a été intéressante, car elle a nécessité l'utilisation de nombreux concepts et algorithmes de la théorie des graphes. De plus, son implémentation concrétise la théorie des graphes car l'algorithme de Christofides a été utilisé pour diverses applications, telles que la planification de tournées de véhicules, la conception de circuits intégrés, la planification de trajets de robots et l'ordonnancement de tâches. Enfin, il convient de souligner que l'algorithme de Christofides a également servi de base à de nombreuses autres méthodes heuristiques pour le TSP, ce qui en fait un outil précieux et polyvalent. En ce sens, ce travail a été passionnant et utile à réaliser.