Problème du voyageur de commerce

Mohamed El Khache'

1 Introduction

Le **problème du voyageur de commerce** (ou *Travelling Salesman Problem*, TSP) est l'un des problèmes les plus célèbres et complexes en informatique théorique. Il représente également un cas fondamental en recherche opérationnelle.

Le problème se formule ainsi :

Étant donné une liste de villes et les distances entre chaque paire de villes, quel est le plus court circuit possible qui visite chaque ville exactement une fois et revient à la ville de départ ?

2 Solutions

La solution par **force brute** consiste à générer toutes les permutations possibles des villes, puis à calculer la longueur de chaque circuit afin de trouver le plus court.

Cependant, cette méthode devient rapidement irréaliste lorsque le nombre de villes n augmente. Dès que $n \geq 10$, le nombre de combinaisons devient trop grand pour être traité efficacement, et pour $n \geq 20$, il est pratiquement impossible de résoudre le problème avec cette approche sur un ordinateur ordinaire.

2.1 Arbre couvrant de poids minimal (MST)

Un **arbre couvrant de poids minimal** (ou *Minimum Spanning Tree*, MST) est un sous-ensemble des arêtes d'un graphe pondéré et connexe qui relie tous les sommets sans former de cycles, tout en minimisant la somme des poids des arêtes utilisées.

Autrement dit, c'est la façon la plus économique de connecter tous les nœuds d'un réseau sans créer de boucles.

Un MST peut être calculé efficacement à l'aide d'algorithmes comme ceux de **Prim** ou de **Kruskal**.

Bien qu'un MST ne fournisse pas directement une solution au TSP (car il ne forme pas un cycle), il est très utile pour **approcher une solution** au problème. En effet, le coût d'un MST constitue une **borne inférieure** pour la longueur du circuit optimal du TSP. Il est donc souvent utilisé comme base pour des algorithmes d'approximation comme celui de **Christofides**.

2.1.1 Algorithme de Prim

L'algorithme de **Prim** est un algorithme glouton qui permet de construire un arbre couvrant de poids minimal. Il commence par un sommet arbitraire et ajoute progressivement les arêtes de poids minimal qui relient un sommet déjà inclus dans l'arbre à un sommet encore non inclus, jusqu'à ce que tous les sommets soient connectés.

Voici une illustration pas à pas de l'algorithme de Prim :

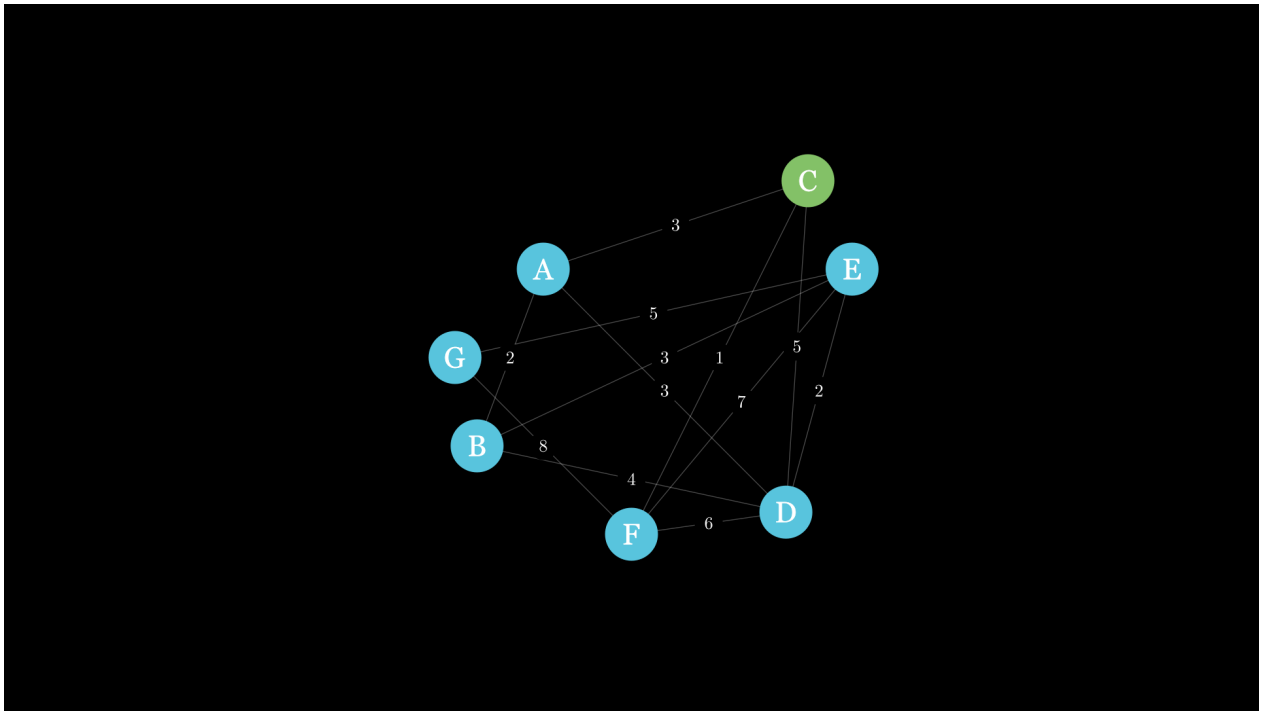


FIGURE 1 – On commence par le sommet **C**, puis on choisit l'arête de plus faible poids.

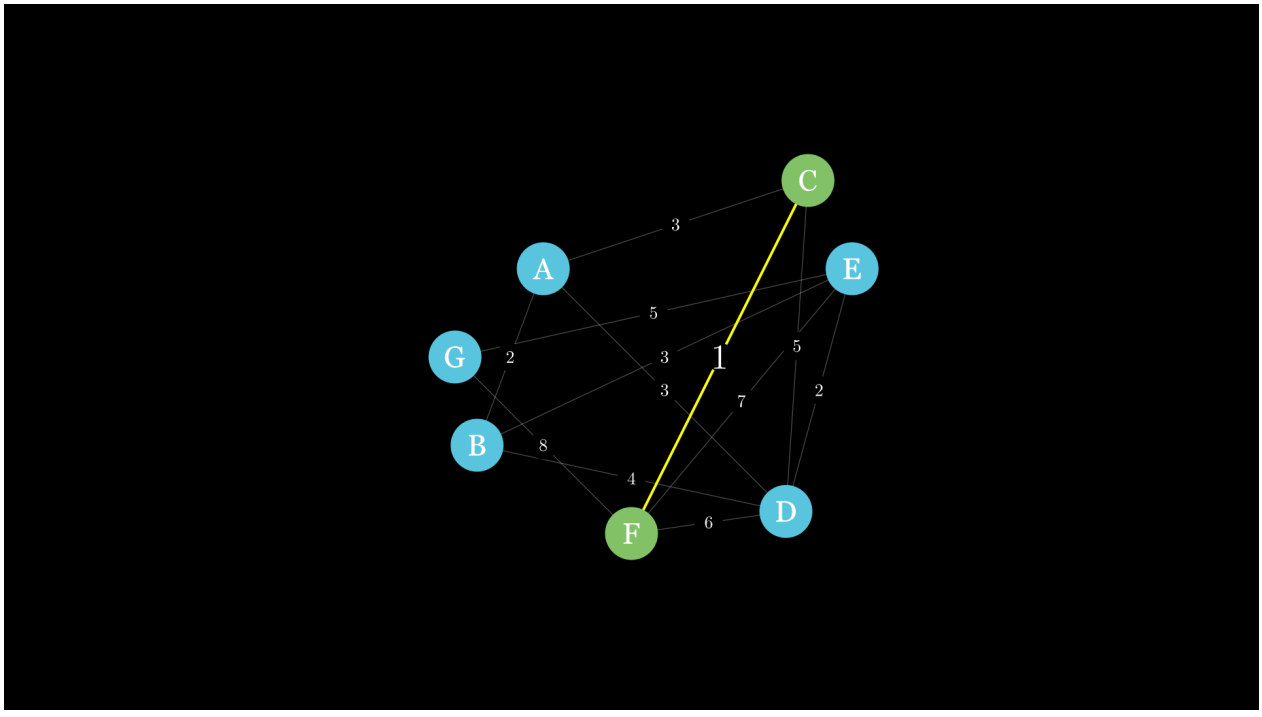


FIGURE 2 – L'arête de coût minimal connecte **C** à **F** avec un poids de **1**.

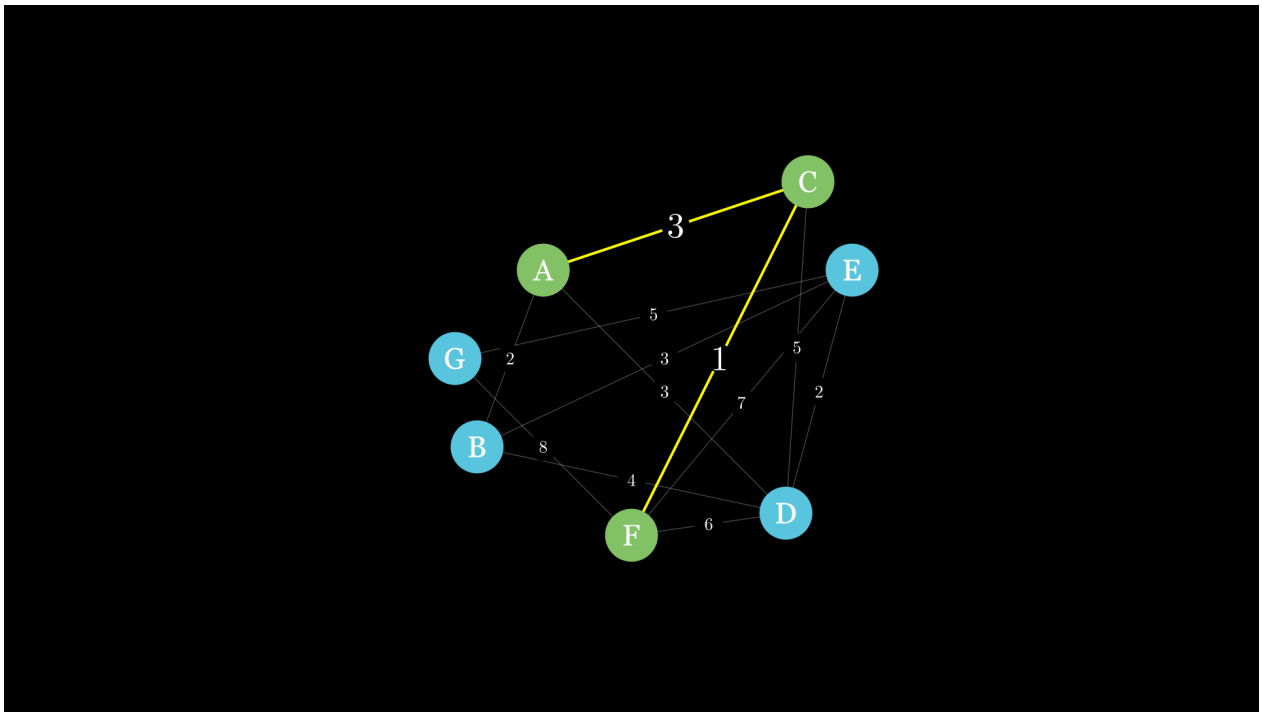


FIGURE 3 – L'arête suivante de poids minimal relie **C** à **A** avec un poids de **1**.

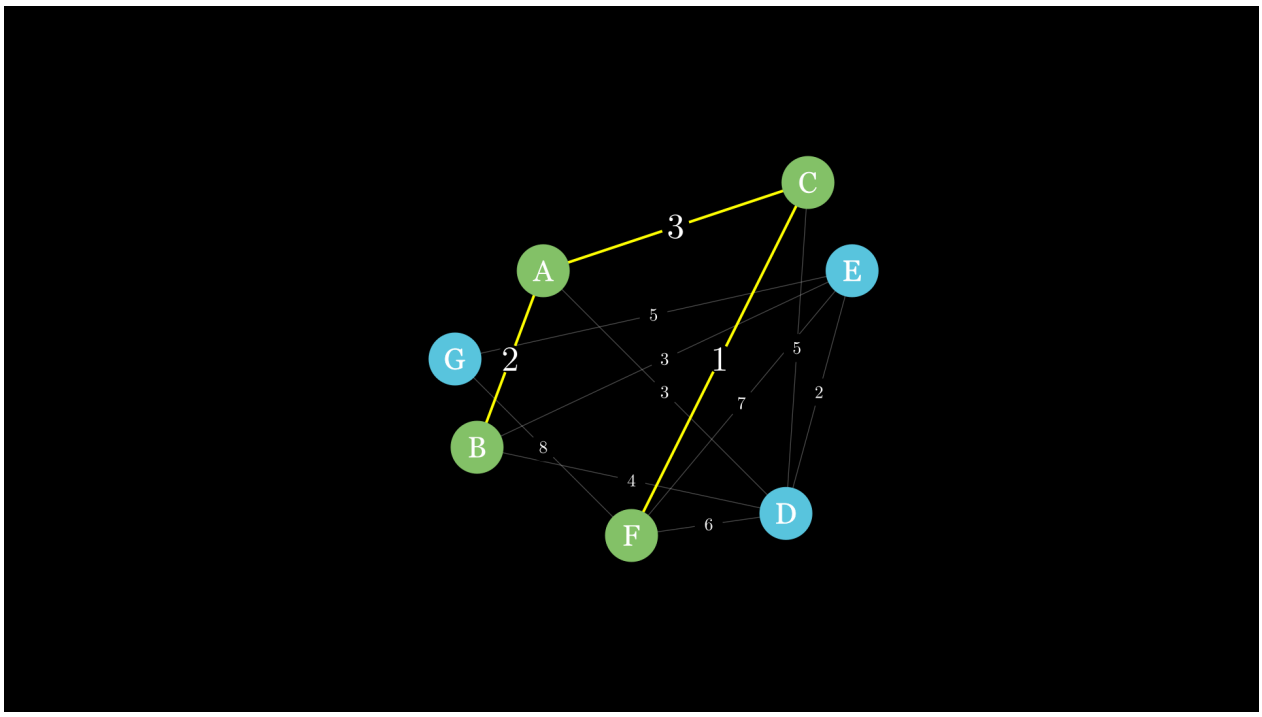


FIGURE 4 – L'arête suivante relie **A** à **B** avec un coût de **2**.

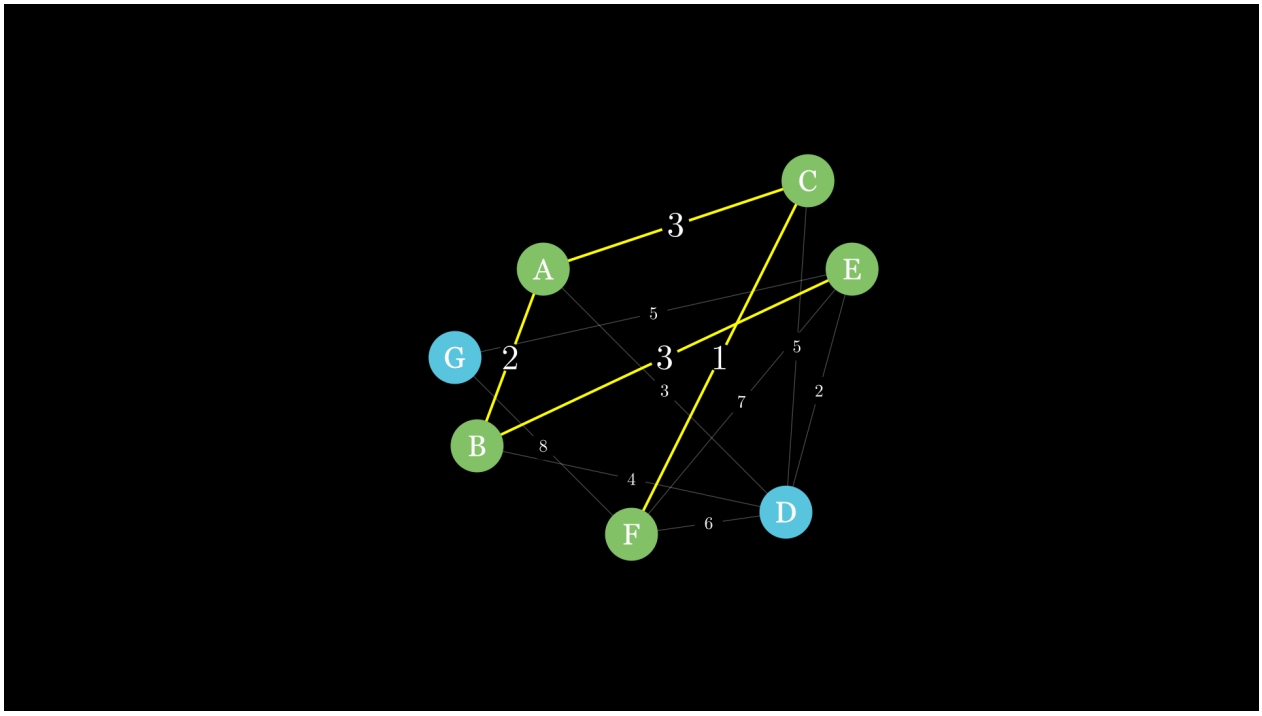


FIGURE 5 – Puis on ajoute **E**, connecté par une arête de coût **3**.

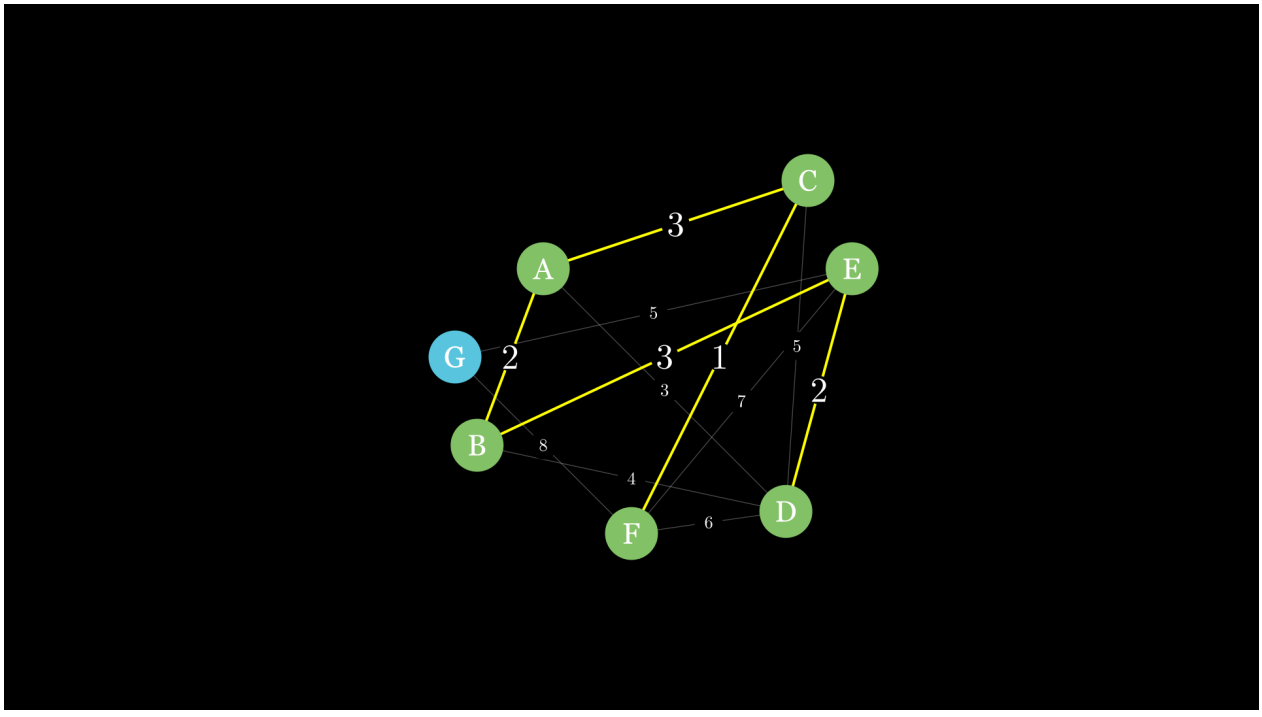


FIGURE 6 – Enfin, on ajoute **D**, connecté à **E** avec un coût de **2**.

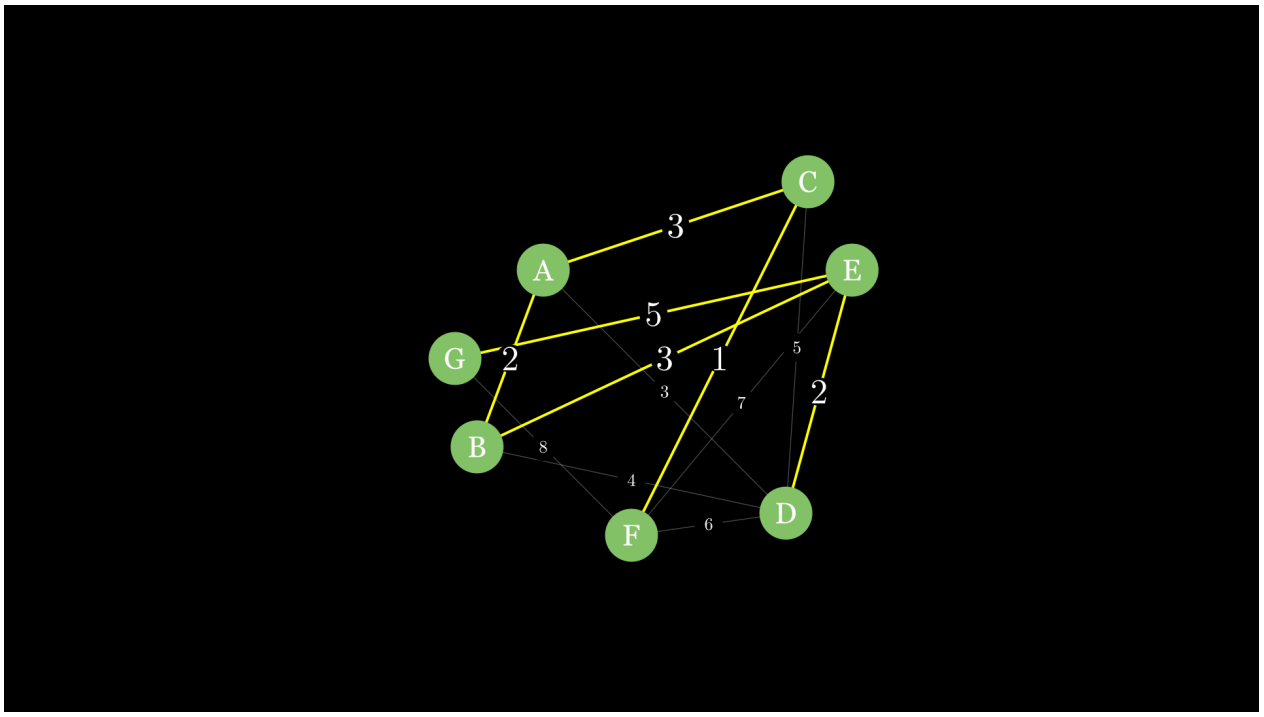


FIGURE 7 – Tous les sommets sont désormais connectés par des arêtes de poids minimal, sans cycles. L'arbre couvrant est complet.

2.1.2 code

```
def prim_mst(cost_matrix):
n = len(cost_matrix)
selected = [False] * n
key = [float('inf')] * n
parent = [None] * n
key[0] = 0 # Start from node 0

for _ in range(n):
    min_key = float('inf')
    u = -1
    for v in range(n):
        if not selected[v] and key[v] < min_key:
            min_key = key[v]
            u = v

    selected[u] = True

    for v in range(n):
        if cost_matrix[u][v] < key[v] and not selected[v]:
            key[v] = cost_matrix[u][v]
            parent[v] = u

# Collect the MST edges
mst_edges = []
total_cost = 0
for v in range(1, n):
    u = parent[v]
    mst_edges.append((u, v))
```

```
        total_cost += cost_matrix[u][v]

    return mst_edges, total_cost
```

2.2 L'algorithme du plus proche voisin (Nearest Neighbour)

L'algorithme du plus proche voisin est une heuristique simple pour résoudre le problème du voyageur de commerce (TSP). Il commence par une ville choisie au hasard, puis, à chaque étape, visite la ville non visitée la plus proche. Ce processus se poursuit jusqu'à ce que toutes les villes soient visitées, puis le circuit se termine en revenant à la ville de départ.

Bien que cette méthode soit rapide et facile à implémenter, elle peut parfois négliger des chemins plus courts, facilement perceptibles à l'œil humain, à cause de sa nature gloutonne.

2.2.1 Étapes de l'algorithme :

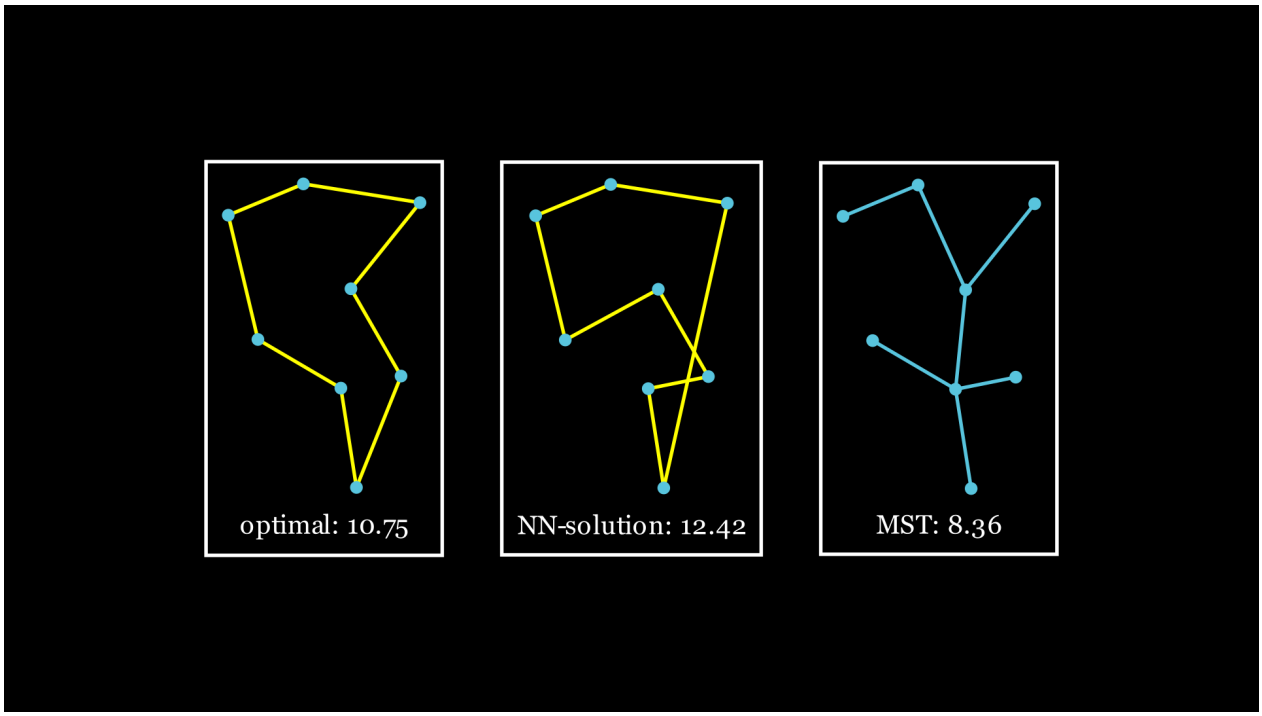
1. Marquer tous les sommets comme non visités.
2. Sélectionner un sommet arbitraire, le définir comme sommet courant **U**, et le marquer comme visité.
3. Trouver le chemin le plus court reliant le sommet courant **U** à un sommet non visité **V**.
4. Définir **V** comme sommet courant et marquer **U** comme visité.
5. Répéter jusqu'à ce que tous les sommets soient visités, puis revenir au sommet de départ pour former un cycle.

2.2.2 Code

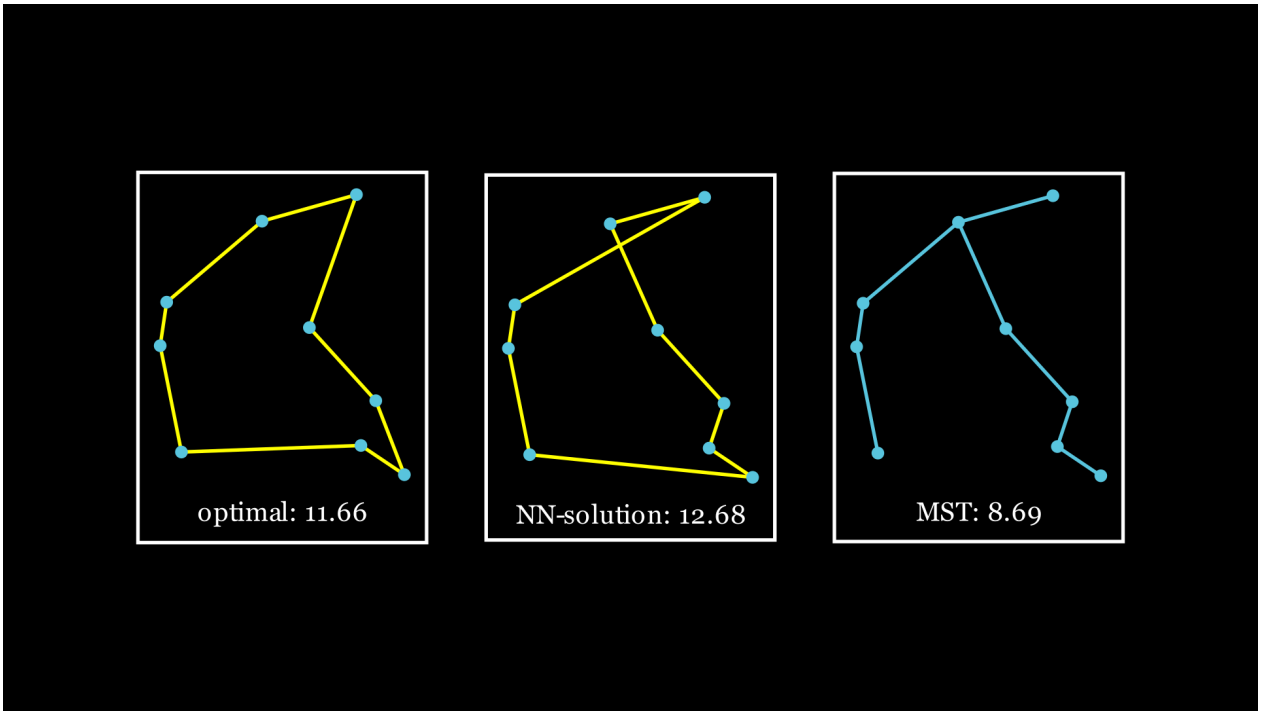
```
def tsp_nearest_neighbor(distnces):

    n = len(distnces) # nombres des villes
    start=0
    visited = [False] * n
    path = [start] # add the first city to the path
    visited[start] = True # mark the city as visited
    total_cost = 0 # null at the first city
    current = start

    for _ in range(n-1):

        next_node = 0 # holds the index of the node with lowest cost
        min_cost = float('inf')

        for i in range(n):
            if not visited[i] and distnces[current][i] < min_cost:
                # the curent point is the one with the minimum cost for now
                min_cost = distnces[current][i]
                next_node = i
       # after this loop next will have the index of the city with the lowest cost
        path.append(next_node) #we add it to the path
        visited[next_node] = True # mark it as visited

        total_cost += min_cost
        current = next_node
    total_cost += distnces[current][0]
    path.append(0)

    return path, total_cost
```

```
costs = [
#    A , B, C, D, E
    [0 , 2, 9,10,12],# A
    [1 , 0, 6, 4, 4],# B
    [15, 7, 0, 8,10],# C
    [6 , 3,12, 0, 3],# D
    [6 , 3,12 ,3, 0] # E
]
path, cost = tsp_nearest_neighbor(costs)
print("Tour:", path)
print("Total cost:", cost)
```

2.2.3 Exemple



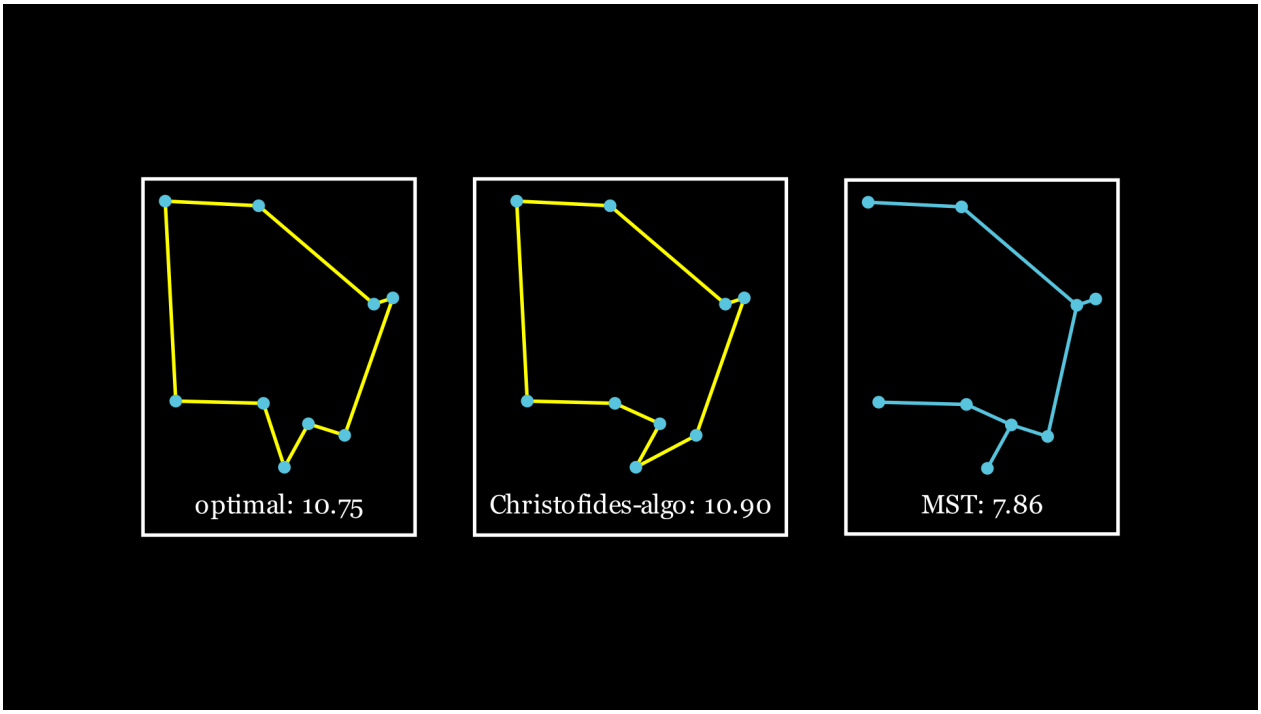
Dans cet exemple, on peut observer à quel point la solution proposée par l'algorithme du plus proche voisin peut s'écarter de la solution optimale, même avec un petit nombre de sommets. Néanmoins, cette approche reste bien moins coûteuse que celle par **force brute**, car elle a une complexité temporelle de $O(N^2)$.



Dans cet exemple, on constate que la solution optimale et celle obtenue via l'**arbre couvrant de poids minimal** (**MST**) partagent plusieurs arêtes communes. Cela pose la question : et si l'on utilisait le **MST** comme base pour construire une solution au **TSP** ?

2.3 Algorithme de Christofides

L'algorithme de Christofides est une méthode d'approximation pour le TSP qui garantit une solution dont le coût ne dépasse pas 1,5 fois celui de la solution optimale. Il commence par construire un arbre couvrant de poids minimal, puis ajoute un appariement de poids minimal entre les sommets de degré impair afin de rendre tous les degrés pairs. Ensuite, il construit un circuit eulérien et le convertit en un tour valide pour le TSP en sautant les sommets répétés. L'inégalité triangulaire garantit que cette étape de raccourcissement n'augmente pas le coût total.



On peut remarquer que la solution donnée par l'algorithme de Christofides ressemble davantage à l'**MST** et qu'elle est plus proche du chemin optimal. À titre de comparaison, l'algorithme du plus proche voisin donne un circuit de longueur 11,24 pour les mêmes points.

Conclusion

Le problème du voyageur de commerce (TSP) est bien plus qu'un simple casse-tête mathématique ; il représente une classe de problèmes fondamentaux qui apparaissent dans de nombreux domaines pratiques, tels que :

— La logistique et la gestion des tournées de livraison,
— L'optimisation des trajets dans les réseaux de transport,
— La planification de circuits dans la fabrication de circuits imprimés (PCB),
— Le séquençage d'ADN en bioinformatique.

Les algorithmes présentés dans ce document le **plus proche voisin** et l'**algorithme de Christofides** — offrent chacun une approche différente pour résoudre ou approximer ce problème complexe. Bien que certaines de ces méthodes soient simples et rapides, elles ne garantissent pas toujours la solution optimale. D'autres, comme Christofides, proposent un bon compromis entre performance et précision.

Cependant, les résultats fournis par ces heuristiques peuvent être encore améliorés. Une approche populaire pour affiner les solutions approximatives est d'appliquer des **échanges k-opt**, en particulier le **2-opt**, qui consiste à retirer deux arêtes du circuit et à reconnecter les sommets différemment pour obtenir un circuit plus court si possible. Ce procédé est souvent utilisé en post-traitement des solutions heuristiques pour améliorer leur qualité.

Resources

- Vidéo 1
- Vidéo 2
- Wikipedia : Travelling Salesman Problem
- Wikipedia : Nearest Neighbour Algorithm
- Wikipedia : Prim's Algorithm