Institut National des Sciences Appliquées
des Hauts-de-France

Département d'Informatique et de Cybersécurité

Rapport des Travaux Pratiques

Graphes et Optimisation : Recherche du plus court chemin et Problème du voyageur de commerce

Date : 15 avril 2024

Professeur : Raca TODOSIJEVIC - Associate Professor

Elias BOULANGER
Thomas AUBERT

Table des matières

Liste des Acronymes

TSP Traveling Salesman Problem

SPP Shortest Path Problem

Table des figures

Liste des tableaux

Chapitre 1

Introduction

aaa

Chapitre 2

Structure du projet

Dans ce chapitre, nous décrivons la structure de données et les choix techniques effectués pour la représentation et la manipulation des graphes utilisés dans ce TP. Nous implémentation se construit autour de deux types de problèmes : Shortest Path Problem (SPP) et Traveling Salesman Problem (TSP).

1 Représentation des noeuds

Nous avons choisi de représenter chaque noeud du graphe à l'aide d'une classe `Node`. Cette classe contient les informations nécessaires pour les algorithmes de recherche de chemin et les fonctions associées. Voici la description des attributs principaux :

— **position** : Un tuple `(x, y)` représentant les coordonnées du noeud dans le réseau.

— **neighbors** : Un dictionnaire où les clés sont des directions `(dx, dy)` et les valeurs sont les noeuds voisins correspondants. Cela permet de naviguer efficacement dans le graphe.

— **parent** : Le noeud parent, utilisé pour reconstruire le chemin après l'exécution de l'algorithme A*.

— **is_obstacle** : Un booléen indiquant si le noeud est un obstacle (non praticable). Les noeuds obstacles ne peuvent pas être voisins d'autres noeuds. Permet une structure de graphe dynamique.

— **g, h, f** : Ces attributs sont utilisés dans l'algorithme A*. `g` est le coût du chemin depuis le noeud de départ, `h` est une estimation heuristique du coût pour atteindre la destination, et `f` est la somme des deux (`f = g + h`).

2 Représentation du graphe

La classe `Graph` est utilisée pour représenter et manipuler le graphe. Elle gère deux types de problèmes : le SPP et le TSP.

2.1 Attributs principaux

— **start, objective** : Les noeuds de départ et d'arrivée pour SPP.

— **graph** : Une matrice de noeuds utilisée pour SPP. Notre (0, 0) est la node en haut à gauche du graphe, pour faciliter la manipulation et visualisation lignes/colonnes.

— **node_list** : Une liste de noeuds utilisée pour TSP.

— **cost** : Un dictionnaire où les clés sont des tuples de noeuds et les valeurs sont les coûts des arêtes entre ces noeuds.

— **shape** : Les dimensions du graphe. La valeur est dynamique, initialisée lors de la lecture du fichier d'entrée. Si les dimensions du graphe sont modifiés, cette valeur est alors mise à jour.

— **file_path, problem** : Le chemin vers le fichier d'entrée et le type de problème à résoudre.

Remarque : la présence de deux structures de graphes, `node_list`, et `graph` s'explique par la différence de représentation des noeuds pour les deux problèmes. Pour SPP, nous utilisons une matrice de noeuds pour faciliter l'implémentation des algorithmes de recherche, notamment CPLEX. Dans ce cas, les couts sont directement dépendant de coordonnées. Pour TSP, nous utilisons une liste de noeuds pour faciliter la manipulation des ensembles de noeuds. Dans ce cas, les couts sont définis par l'utilisateur, il n'y a pas de dépendance directe avec les coordonnées.

2.2 Méthodes principales

— **add_node** : Ajoute un noeud au graphe. Pour SPP, le graphe est étendu si nécessaire. Pour TSP, le noeud est simplement ajouté à la liste.

— **add_edge** : Ajoute une arête entre deux noeuds avec un coût spécifié. Cette méthode vérifie également que les noeuds sont voisins dans SPP.

— **remove_node** : Marque un noeud comme obstacle et déconnecte ses voisins.

— **solve** : Résout le problème spécifié en utilisant l'algorithme A* pour le plus court chemin ou une méthode d'énumération pour le voyageur de commerce.

— **get_neighbors** : Retourne les voisins d'un noeud donné. Prend en compte les obstacles et les limites du graphe.

— **get_edges** : Retourne une liste de toutes les arêtes du graphe.

Remarque : les méthodes `add_node`, `add_edge`, et `remove_node` sont implémentées pour les deux types de problèmes mais ne sont pas utilisées au sein du projet. Elles sont utiles si l'on veut manipuler dynamiquement le graphe ; ce qui est possible, mais n'a pas été necessaire pour les problèmes traités dans ce TP. Lorsque nous avions besoin de créer un graphe, nous initialisions les variables nécessaires via le traitement du fichier d'entrée.

3 Choix de performance et praticité

— **Utilisation de dictionnaires pour les voisins** : Cette structure permet un accès rapide et une gestion efficace des voisins d'un noeud.

— **Matrice de noeuds** : Pour SPP, cette représentation facilite l'implémentation des algorithmes de recherche et de parcours.

— **Liste de noeuds** : Pour TSP, cette représentation est plus adaptée car elle permet de manipuler facilement les ensembles de noeuds.

— **Coûts des arêtes** : Les coûts sont calculés à partir de la distance euclidienne pour SPP, ce qui permet une estimation réaliste des distances. Pour TSP, les coûts sont définis par l'utilisateur dans le fichier d'entrée.

4 Exécution du programme

Pour exécuter le programme, nous utilisons le fichier `main.py` qui contient les fonctions principales pour lancer les différents algorithmes sur les problèmes spécifiés. Voici une explication détaillée de la fonction principale et de son utilisation :

— **run** : Cette fonction exécute l'algorithme spécifié sur le problème donné. Les paramètres incluent le nombre d'itérations (`n_iter`), le type de problème (`problem`), l'algorithme à utiliser (`algo`), le chemin du fichier d'entrée (`file_path`), et des options pour afficher (`display`), sauvegarder (`save`), ou afficher les détails (`verbose`) du résultat. La fonction génère d'abord le graphe à partir du fichier spécifié, puis exécute l'algorithme et mesure le temps d'exécution pour chaque itération.

— **compare_algo** : Cette fonction compare les temps d'exécution des algorithmes A* et CPLEX pour SPP, ou des algorithmes de force brute et CPLEX pour TSP. Si un chemin de fichier est spécifié, la résolution est effectuée sur ce fichier, sinon un graphe aléatoire est généré.

Pour exécuter le programme, ouvrez un terminal et utilisez la commande suivante :

```
python main.py
```

Les modules python nécessaires sont spécifiés dans le fichier `requirements.txt`. Le module `docplex` est requis mais n'est pas dans le fichier car la licence pro ou académique est nécessaire pour résoudre les problèmes de plus de 30 noeuds.

Chapitre 3

Plus court chemin

1 Modélisation

On considère un graphe non orienté $G =< S, A >$ où S est l'ensemble des sommets et A l'ensemble des arêtes. Chaque arête a_{ij} est associée à un coût c_{ij}, qui vaudra 1 dans le cas où deux sommets sont reliés horizontalement ou verticalement, et $\sqrt{2}$ dans le cas où ils sont reliés en diagonale. On cherche à déterminer le plus court chemin entre un sommet de départ s et un sommet d'arrivée t.

1.1 Variables

— x_{ij} : vaut 1 si l'arête a_{ij} est empruntée, 0 sinon

1.2 Fonction objectif

On cherche à minimiser la somme des coûts des arêtes empruntées :

$$\min \sum_{(i,j)\in A} c_{ij} \cdot x_{ij} \tag{3.1}$$

1.3 Contraintes

— Le sommet de départ s est toujours relié à un sommet :

$$\sum_{j\in S} x_{sj} = 1 \tag{3.2}$$

— De même, le sommet d'arrivée t est toujours relié à un sommet :

$$\sum_{i\in S} x_{it} = 1 \tag{3.3}$$

— Le sommet de départ s n'a pas d'arête entrante :

$$\sum_{i\in S} x_{is} = 0 \tag{3.4}$$

— De même, le sommet d'arrivée t n'a pas d'arête sortante :

$$\sum_{j\in S} x_{tj} = 0 \tag{3.5}$$

— Chaque sommet a le même nombre d'arêtes entrantes et sortantes (sauf s et t) :

$$\sum_{j \in S} x_{ij} = \sum_{j \in S} x_{ji} \quad \forall i \in S \setminus \{s, t\} \tag{3.6}$$

— Notre graphe n'étant pas orienté, nous devons empêcher les sous-cycles, c'est-à-dire le cas où on trouve une arête a_{ij} et une arête a_{ji} dans le chemin :

$$\sum_{(i,j) \in A} x_{ij} + \sum_{(j,i) \in A} x_{ji} \leq 1 \quad \forall i, j \in S \setminus \{s, t\} \tag{3.7}$$

Nous avons implémenté et résolu ce problème en Python, en utilisant la librairie `docplex.mp.model` de CPLEX. Le code complet est disponible en annexe 1.

2 A Star

L'algorithme A* est une méthode heuristique utilisée pour trouver le plus court chemin entre deux sommets dans un graphe. Il combine les avantages de la recherche en profondeur et de la recherche en largeur tout en utilisant une fonction heuristique pour guider la recherche. Voici une description du pseudo-code de base de l'algorithme A* et des améliorations apportées.

2.1 Pseudo-code de base

Le pseudo-code de base de l'algorithme A* est le suivant :

1. Initialiser l'ensemble ouvert (open set) avec le noeud de départ.
2. Initialiser le coût g du noeud de départ à 0.
3. Calculer la valeur heuristique h pour le noeud de départ et mettre à jour sa valeur f (f = g + h).
4. Répéter jusqu'à ce que l'ensemble ouvert soit vide :
   (a) Extraire le noeud avec la plus petite valeur f de l'ensemble ouvert.
   (b) Si ce noeud est le noeud d'arrivée, reconstruire le chemin en remontant à travers les parents.
   (c) Pour chaque voisin du noeud courant :
      i. Calculer le coût g temporaire pour ce voisin.
      ii. Si ce coût g est inférieur au coût g actuel du voisin, mettre à jour les valeurs g, h et f du voisin, et définir le noeud courant comme parent du voisin.
      iii. Ajouter le voisin à l'ensemble ouvert s'il n'y est pas déjà.

Pour que cet algorithme fonctionne, il ne faut pas oublier d'initialiser la valeur g des noeuds à l'infini. Aussi, il faudrait réinitialiser les valeurs g, h et f des noeuds à l'infini à chaque itération de l'algorithme. Cela permet de recalculer les valeurs g, h et f correctement pour chaque itération.

2.2 Améliorations

Avec une heap queue

Pour améliorer l'efficacité de l'algorithme A*, nous utilisons une file de priorité (heap queue) pour l'ensemble ouvert. Cela permet d'extraire le noeud avec la plus petite valeur f en temps logarithmique. Les voisins des noeuds sont initialisés une seule fois au début du programme, ce qui réduit le coût de recalcul des voisins à chaque itération.

Avec un ensemble de noeuds visités

Pour réduire la vérification de présence d'un noeud dans l'ensemble ouvert, nous utilisons un ensemble (set en python) pour suivre les noeuds déjà visités. Cela permet de vérifier si un noeud est dans l'ensemble ouvert en temps constant.

2.3 Implémentation en Python

Les optimisations se traduisent dans le code Python suivant :

```
def a_star(start_node: Node, end_node: Node) -> Optional[List[Node]]:
    open_set = [start_node]
    heapq.heapify(open_set)
    opens_set_tracker = {start_node}

    start_node.g = 0
    start_node.h = heuristic(start_node, end_node)
    start_node.f = start_node.h

    while open_set:
        current_node: Node = heapq.heappop(open_set)
        opens_set_tracker.remove(current_node)

        if current_node == end_node:
            return reconstruct_path(current_node)

        for neighbor in current_node.neighbors.values():
            tentative_g = current_node.g + distance(current_node, neighbor)
            if tentative_g < neighbor.g:
                neighbor.parent = current_node
                neighbor.g = tentative_g
                neighbor.h = heuristic(neighbor, end_node)
                neighbor.f = neighbor.g + neighbor.h
                if neighbor not in open_set_tracker:
                    heapq.heappush(open_set, neighbor)
                    opens_set_tracker.add(neighbor)

    return None
```

Remarque : les fonctions `heuristic`, `distance`, et `reconstruct_path` sont des fonctions auxiliaires utilisées dans l'algorithme A*. Les deux premières permettent une flexibilité dans le calcul des coûts et des heuristiques, tandis que la dernière permet de reconstruire le chemin à partir du noeud d'arrivée via les parents. Par défaut une bonne mesure de distance est la distance de Manhattan, et une bonne heuristique est la distance euclidienne. Cette dernière permet le déplacement en diagonale. Nous n'utilisons pas la distance euclidienne pour le cout de déplacement, car elle favorise une augmentation du nombre de noeuds visités et donc une augmentation du temps de calcul.

2.4 Optimisation spatiale et temporelle

L'algorithme A* n'utilise pas l'ensemble complet de la matrice du graphe, mais uniquement les noeuds de départ et d'arrivée ainsi que les voisins nécessaires pour chaque itération. Cela permet d'optimiser l'utilisation de la mémoire et d'accélérer les opérations.

Les voisins des noeuds sont stockés dans un attribut de chaque noeud et initialisés une seule fois au début du programme. Ainsi, chaque résolution de A* n'appelle pas de fonctions `get_neighbors`, mais opère directement sur l'attribut `neighbors` de chaque noeud.

2.5 Complexité de l'algorithme A*

Complexité temporelle sans heap queue

Sans l'utilisation de heap queue, l'ensemble des nœuds ouverts (open set) est géré comme une liste non ordonnée. Les opérations de recherche du nœud avec le coût f minimal et l'insertion d'un nouveau nœud sont coûteuses :

- Extraction du nœud avec le coût f minimal :$\mathcal{O}(n)$
- Insertion d'un nouveau nœud : $\mathcal{O}(1)$
- Vérification de la présence dans open set : $\mathcal{O}(n)$

La complexité temporelle totale est alors $O(E \cdot V)$, où E est le nombre d'arêtes et V est le nombre de nœuds.

Complexité temporelle avec heap queue

Avec l'utilisation de heap queue, l'ensemble des nœuds ouverts (open set) est géré comme une heap binaire. Cela optimise les opérations suivantes :

- Extraction du nœud avec le coût f minimal : $\mathcal{O}(\log n)$
- Insertion d'un nouveau nœud : $\mathcal{O}(logn)$
- Vérification de la présence dans open set : $\mathcal{O}(n)$

Même si la vérification de la présence reste coûteuse, les opérations de base de l'algorithme, à savoir l'extraction et l'insertion dans la heap, dominent la complexité globale. La complexité temporelle totale est donc $\mathcal{O}(E \log V)$.

Complexité temporelle avec heap queue et ensemble (set)

En ajoutant un ensemble (set) pour suivre les nœuds dans open set, on optimise la vérification de la présence :

- Extraction du nœud avec le coût f minimal : $\mathcal{O}(\log n)$
- Insertion d'un nouveau nœud : $\mathcal{O}(\log n)$
- Vérification de la présence dans open set : $\mathcal{O}(1)$

Cependant, ces améliorations ne changent pas la complexité dominante de l'algorithme. L'opération d'extraction du nœud avec le coût f minimal et l'insertion dans la heap binaire restent les opérations les plus coûteuses et se produisent $\mathcal{O}(E)$ fois. Par conséquent, la complexité temporelle totale reste $\mathcal{O}(E \log V)$.

Note

La complexité temporelle - $\mathcal{O}(E \log V)$ ou $\mathcal{O}(E \cdot V)$ - est une borne supérieure. En pratique, à moins qu'il n'existe aucun chemin entre les deux nœuds de départ et d'arrivée, et pour une bonne heuristique, la complexité temporelle est bien plus faible.

3 Génération de graphes aléatoires

Afin de pouvoir comparer les deux implémentations, nous avons créé une fonction générant des graphes aléatoires. Cette fonction prend en paramètre le nombre de sommets n et la probabilité p qu'un sommet soit un obstacle.

```python
def gen_astar(n: int, p: float, file_name: str = "astar.txt", verbose: bool = False):
    """Generates a random shortest path problem with n^2 nodes and probability p of having an edge between nodes."""
    graph = [[Values.WALL for _ in range(n)] for _ in range(n)]

    # Generate nodes
    for i in range(n):
        for j in range(n):
            if random.random() < p:
                graph[i][j] = Values.EMPTY

    # Start and objective. Select a random start and objective from the empty nodes
    empty_nodes = [(i, j) for i in range(n) for j in range(n) if graph[i][j] == Values.EMPTY]
    start, objective = random.sample(empty_nodes, 2)
    graph[start[0]][start[1]] = Values.START
    graph[objective[0]][objective[1]] = Values.OBJECTIVE

    # Write to file
    Path("examples").mkdir(parents=True, exist_ok=True)
    with open(f"examples/{file_name}", "w") as f:
        f.write(f"{n} {n}\n")
        for i in range(n):
            for j in range(n):
                f.write(f"{graph[i][j]} ")
            f.write("\n")

    if verbose:
        print(f"File saved in examples/{file_name}")
```

FIGURE 3.1 – Fonction générant un graphe aléatoire

4 Résultats et comparaison des deux implémentations

Les deux méthodes étant fondamentalement différentes, nous pouvons observer de légères différences sur les résultats obtenus.

Les plots ont été générés grâce à la librairie `matplotlib` et networkx de Python. Cette dernière n'a été utilise que pour la visualisation des graphes, et non pour la résolution du problème. Une visualisation n'utilisant qu'opencv et nos structures dédiées existe également. Les graphiques suivants représentent le réseau de sommets liés par des arêtes, avec le sommet de départ en vert, le sommet d'arrivée en bleu clair, et le chemin trouvé en rouge.

Nous avons une légère différence dans le chemin trouvé par CPLEX et A Star sur le réseau `reseau_20_20_1` :

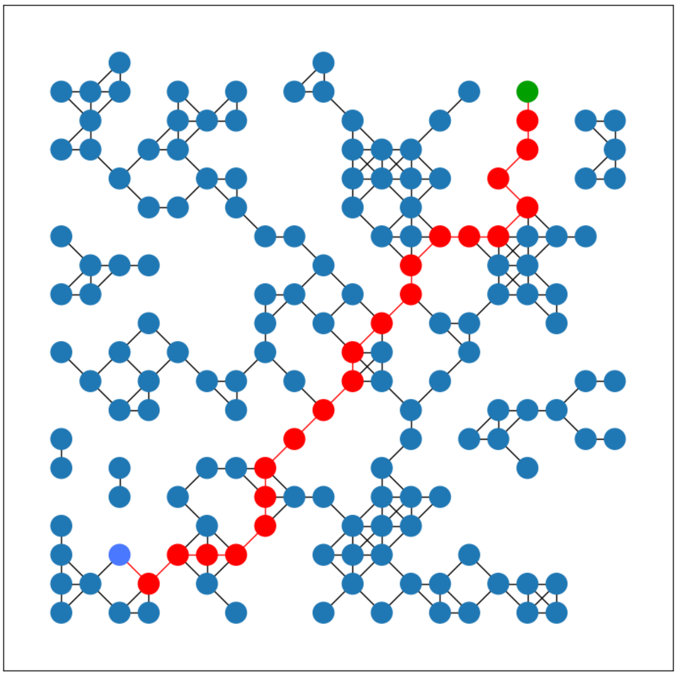

Figure 3.2 – Graphique de la solution trouvée par CPLEX

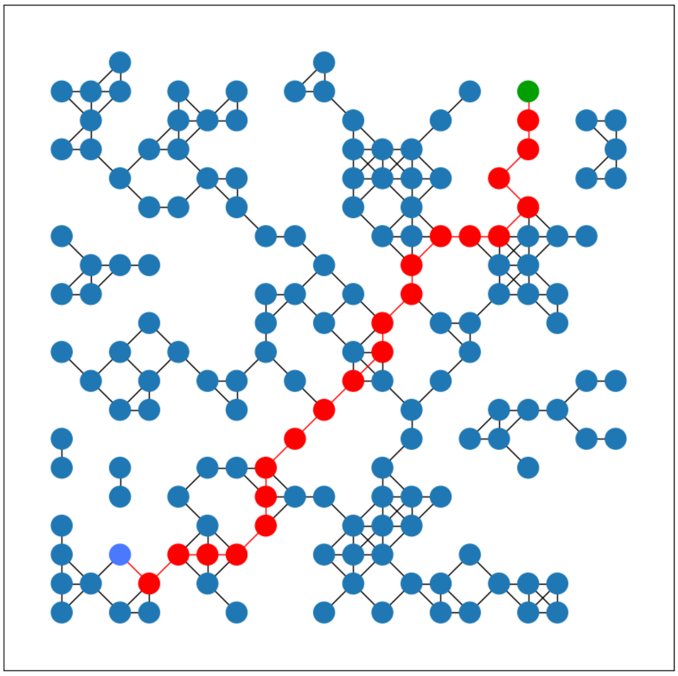

Figure 3.3 – Graphique de la solution trouvée par A Star

4.1 Comparaison des temps de calcul

Méthode / Taille du graphe	10^2	15^2	30^2	60^2	120^2
CPLEX	0.00009	0.00013	0.00051	0.00248	0.00804s
A*	0.02184	0.04013	0.19364	0.56905	2.36712s

Table 3.1 – Temps d'exécution (s) des deux méthodes en fonction de la taille du graphe. Moyenne obtenue sur 100 itérations. Densité des arêtes : 0.3

Changer la valeur de la fonction heuristique n'a que peu d'impact sur le temps de calcul. Par exemple pour un graphe 120×120 avec une densité de 0.3, le temps de calcul est de 0.00804s avec la distance euclidienne, de 0.01021s avec la distance de Manhattan, et de 0.00965s avec une heuristique nulle. Augmenter la taille du graphe ne semble pas change cette égalité : pour un graphe 500×500 avec une densité de 0.3, le temps de calcul est de 0.12993s avec la distance euclidienne, de 0.12056s avec la distance de Manhattan, et de 0.12651s avec une heuristique nulle.

Chapitre 4

Problème du voyageur de commerce

Le TSP consiste à trouver le plus court chemin passant par chaque ville une et une seule fois, et revenant à la ville de départ. Nous allons employer deux méthodes pour résoudre ce problème : une résolution linéaire avec CPLEX [Wik24] et une résolution par énumération des permutations possibles.

1 Modélisation

On considère un graphe non orienté $G =< S, A >$ où S est l'ensemble des sommets et A l'ensemble des arêtes. À chaque arête a_{ij} est associée une distance c_{ij}. On détermine le plus court chemin passant une fois par chaque sommet, et revenant au sommet de départ.

1.1 Variables

— x_{ij} : vaut 1 si l'arête a_{ij} est empruntée, 0 sinon

1.2 Fonction objectif

On cherche à minimiser la somme des distances des arêtes empruntées :

$$\min \sum_{(i,j)\in A} c_{ij} \cdot x_{ij} \tag{4.1}$$

1.3 Contraintes

— Chaque sommet doit être relié à une arête entrante :

$$\sum_{i\in S} x_{ik} = 1 \quad \forall k \in S \tag{4.2}$$

— Chaque sommet doit être relié à une arête sortante :

$$\sum_{j\in S} x_{kj} = 1 \quad \forall k \in S \tag{4.3}$$

— Empêcher les sous-cycles :

$$\sum_{(i,j)\in A} x_{ij} + \sum_{(j,i)\in A} x_{ji} \leq 1 \quad \forall i, j \in S \tag{4.4}$$

Le code Python équivalent à ce modèle est donné en annexe 2.

2 Résolution par énumération

On peut résoudre le TSP en énumérant toutes les permutations possibles des villes, et en calculant la distance totale pour chaque permutation. La solution optimale est celle qui minimise la distance totale.

Data: $graph$
Result: $best\_route$: liste des villes dans l'ordre optimal
$min\_cost \leftarrow \infty$;
$best\_route \leftarrow []$;
for $start\_end\_node$ *in* $graph.nodes$ **do**
    $remaining\_nodes \leftarrow graph.nodes - start\_end\_node$;
    for $permutation$ *in* $permutations(remaining\_nodes)$ **do**
        $route \leftarrow [start\_end\_node] + permutation + [start\_end\_node]$;
        $cost \leftarrow 0$;
        for $i \leftarrow 0$ **to** $len(route) - 1$ **do**
            $cost \leftarrow cost + graph.costs[route[i]][route[i+1]]$;
        end
        if $cost < min\_cost$ **then**
            $min\_cost \leftarrow cost$;
            $best\_route \leftarrow route$;
        end
    end
end
return $best\_route$;

Algorithm 1: tsp_brute_force

3 Génération de graphes aléatoires

Nous pouvons implémenter une fonction pour générer des graphes aléatoires de n villes avec des coûts aléatoires. Nous pourrons ainsi tester nos algorithmes sur des graphes de différentes tailles. Nous choisissons des coûts entre 10 et 50, et les villes ont une probabilité p d'être connectées.

```python
def gen_tsp(n: int, p:float, file_name:str = "tsp.txt"):
    """Generates a random TSP problem with n nodes and probability p of having an edge between nodes."""

    nodes = []
    cost = {}

    # Generate nodes
    for i in range(n):
        nodes.append(i)

    # Generate edges
    for i in range(n):
        for j in range(i+1, n):
            if random.random() < p:
                cost[(i, j)] = random.randint(10, 50)

    # Write to file
    Path("examples").mkdir(parents=True, exist_ok=True)
    with open(f"examples/{file_name}", "w") as f:
        f.write(f"{n} {len(cost)}\n")
        for (i, j), c in cost.items():
            f.write(f"{i} {j} {c}\n")

    print(f"File saved in examples/{file_name}")
```

FIGURE 4.1 – Fonction de génération de graphes aléatoires pour le TSP

4 Résultats et comparaison des méthodes

Nous pouvons tester nos algorithmes avec un exemple simple fourni par l'énoncé. Ils nous donnent le même résultat :

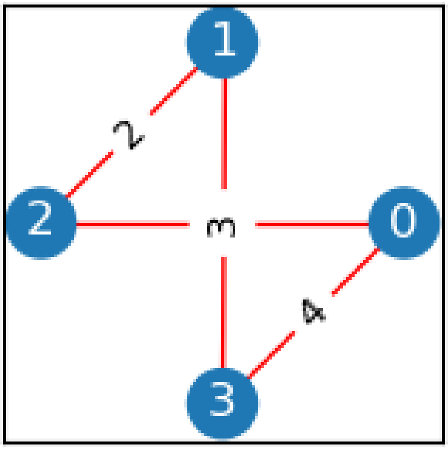

FIGURE 4.2 – Résolution d'un exemple simple de TSP

Nous pouvons désormais comparer le temps d'exécution des deux méthodes en fonction de la taille du graphe. Nous testons les méthodes sur des graphes dont les villes ont une probabilité de connexion de 0.8. Changer cette probabilité ne semble pas avoir un impact significatif sur les résultats.

Méthode / Taille du graphe	6	8	10	12	20	30
CPLEX	0.012	0.012	0.013	0.014	0.023	0.04
Énumération	0.00005	0.025	2.11	338.57	10^3	» 10^3

TABLE 4.1 – Temps d'exécution (s) des deux méthodes en fonction de la taille du graphe

Nous extrapolons les résultats par une régression linéaire pour obtenir une estimation du temps d'exécution pour des graphes de taille supérieure à 12 avec la méthode par énumération.

Nous constatons que la méthode par énumération devient rapidement impraticable pour des graphes de taille supérieure à 10, alors que la méthode CPLEX reste efficace pour des graphes de taille plus importante. Cela est dû à la complexité exponentielle de la méthode par énumération.

Voici un autre exemple de résolution du TSP avec un graphe de 20 villes généré aléatoirement :

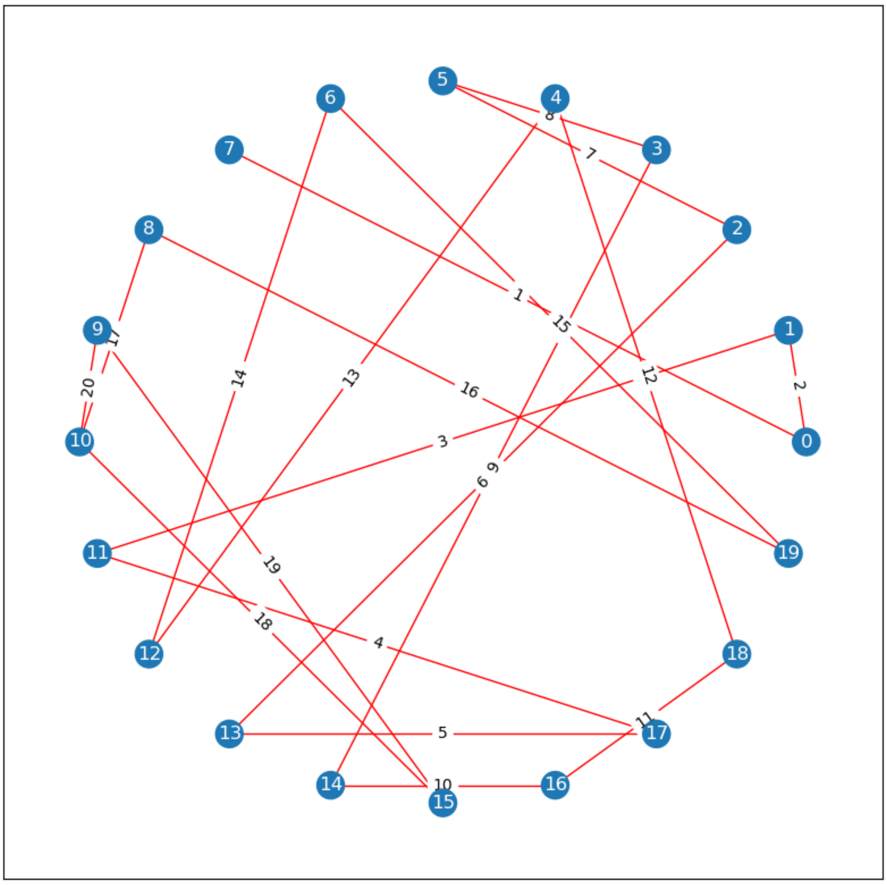

FIGURE 4.3 – Résolution d'un exemple de TSP avec 20 villes

Chapitre 5

Conclusion

eee

Annexe A

Algorithmes et Code

1 Plus court chemin

```python
def shortest_path_cplex_solver(graph, start_node, end_node):
    model = Model("Pathfinding")
    edges = graph.get_edges()
    cost = graph.cost

    # Variables for each edge: 1 if edge is used, 0 otherwise
    x = {e: model.binary_var(name=f"x_{e[0].position}_{e[1].position}") for e in edges}

    # Objective: Minimize the sum of the costs of the edges included in the path
    model.minimize(model.sum(cost[e] * x[e] for e in edges))

    # Constraints
    # Ensure exactly one outgoing edge from start and one incoming edge to end
    model.add_constraint(
        model.sum(x[(start_node, neighbor)] for neighbor in start_node.neighbors.values()
                  if (start_node, neighbor) in x) == 1
    )
    model.add_constraint(
        model.sum(x[(neighbor, start_node)] for neighbor in start_node.neighbors.values()
                  if (neighbor, start_node) in x) == 0
    )
    model.add_constraint(
        model.sum(x[(neighbor, end_node)] for neighbor in end_node.neighbors.values()
                  if (neighbor, end_node) in x) == 1
    )
    model.add_constraint(
        model.sum(x[(end_node, neighbor)] for neighbor in end_node.neighbors.values()
                  if (end_node, neighbor) in x) == 0
    )

    # Connectivity constraints
    for row in graph:
        for node in row:
            if node == start_node or node == end_node:
                continue
            model.add_constraint(
                model.sum(x[(node, neighbor)] for neighbor in node.neighbors.values() if (node, neighbor) in x) ==
                model.sum(x[(neighbor, node)] for neighbor in node.neighbors.values() if (neighbor, node) in x)
            )

    # No sub-tours
    for row in graph:
        for node in row:
            if node.is_obstacle or node == start_node or node == end_node:
                continue
            for neighbor in node.neighbors.values():
                if (node, neighbor) in x:
                    model.add_constraint(x[(node, neighbor)] + x[(neighbor, node)] <= 1)

    solution = model.solve()
    if solution:
        path = []
        # print("\nPath found with total cost:", model.objective_value)
        for e in x:
            if x[e].solution_value > 0.5:
                # print(f"{e[0].position} -> {e[1].position}")
                path.append(e[0])
        path.append(graph.objective)
        return path
    else:
        # print("No solution found")
        return None
```

2 Modélisation du Problème du Voyageur de Commerce

```python
def tsp_cplex_solver(graph) -> Optional[List]:
    """
    Solve the Traveling Salesman Problem (TSP) using the CPLEX solver
    :param graph: the graph to explore
    :return: the cheapest path found
    """
    nodes = graph.node_list
    costs = graph.cost

    # Create a new model
    mdl = Model('TSP')

    # Create variables: x[i, j] is 1 if edge (i, j) is part of the solution
    x = mdl.binary_var_dict(costs.keys(), name='x')

    # Objective: Minimize the total cost of the tour
    mdl.minimize(mdl.sum(x[i, j] * costs[i, j] for (i, j) in costs))

    # Constraints: Each node must be entered and left exactly once
    # Unique entering and leaving edges for each node
    for k in nodes:
        mdl.add_constraint(mdl.sum(x[i, k] for i in nodes if (i, k) in x) == 1, f'enter_{k}')
        mdl.add_constraint(mdl.sum(x[k, j] for j in nodes if (k, j) in x) == 1, f'leave_{k}')

    # No Sub tours
    for i in nodes:
        for j in nodes:
            if i != j and (i, j) in x:
                mdl.add_constraint(x[i, j] + x[j, i] <= 1, f'sub_tour_{i}_{j}')

    # Solve the model
    solution = mdl.solve()

    # Check if a solution exists
    if solution:
        edges = [(i, j) for i, j in x if x[i, j].solution_value > 0.5]
        return [edge[0] for edge in edges] + [edges[-1][1]]
    else:
        return None
```

Bibliographie

[Wik24] Wikipedia. *Problème du voyageur du commerce*. 2024. url : https://fr.wikipedia.org/w/index.php?title=Probl%C3%A8me_du_voyageur_de_commerce&oldid=215118809 (visité le 23/05/2024).