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

Table des figures

Liste des tableaux

Chapitre 1

Introduction

aaa

Chapitre 2

Structure du projet

bbb

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.

Variables

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

Fonction objectif

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

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

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

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. Par exemple, sur le graphe `reseau_20_20_1`, nous avons une différence :

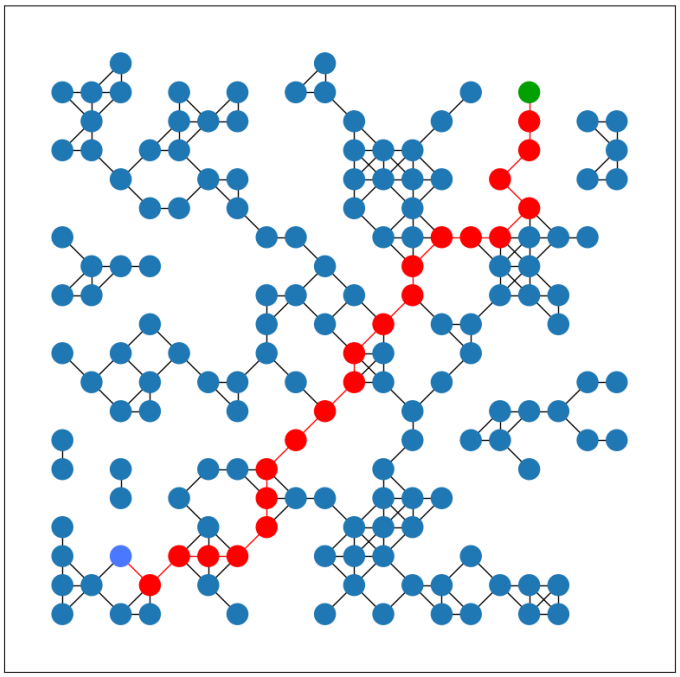

FIGURE 3.2 – Graphique de la solution trouvée par CPLEX

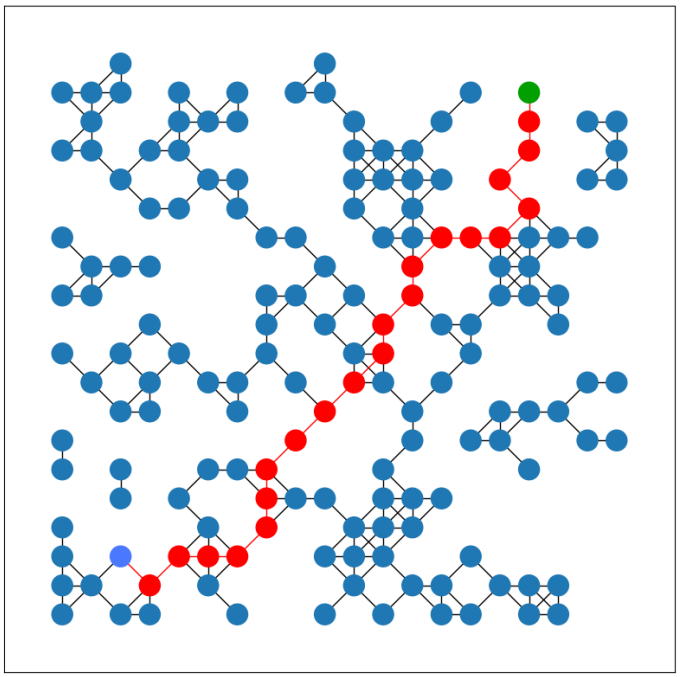

FIGURE 3.3 – Graphique de la solution trouvée par A Star

Chapitre 4

Problème du voyageur de commerce

Le problème du voyageur de commerce (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 cherche à déterminer le plus court chemin passant par chaque sommet une et une seule fois, et revenant au sommet de départ.

Variables

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

Fonction objectif

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

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

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.

Résolution par énumération

On peut résoudre le problème du 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

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

Résultats et comparaison des méthodes

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

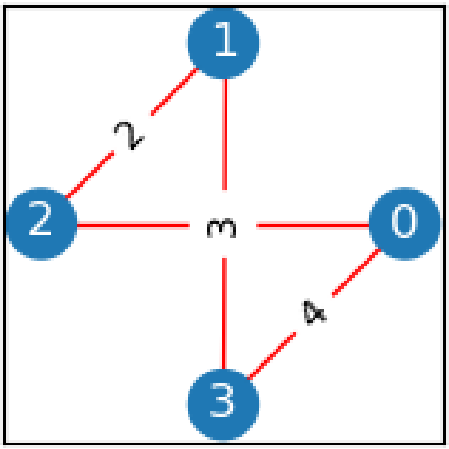

FIGURE 4.2 – Résolution d'un exemple simple de TSP

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	?	?

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

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.

TODO : comparer avec la complexité théorique

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).