

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

1	Introduction	1
2	Structure du projet	2
3	Plus court chemin	3
1	Modélisation	3
2	A Star	4
3	Comparaison des deux implémentations	4
4	Problème du voyageur de commerce	5
1	Modélisation	5
5	Conclusion	8
A	Algorithmes et Code	I
1	Modélisation du Problème du Voyageur de Commerce	I

Liste des Acronymes

Table des figures

4.1	Fonction de génération de graphes aléatoires pour le TSP	7
4.2	Résolution d'un exemple simple de TSP	7

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 $G = \langle S, A \rangle$ 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 :

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

Contraintes

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

$$\sum_{j \in S} x_{sj} = 1 \quad (3.2)$$

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

$$\sum_{i \in S} x_{it} = 1 \quad (3.3)$$

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

$$\sum_{j \in S} x_{ij} = \sum_{j \in S} x_{ji} \quad \forall i \in S \setminus \{s, t\} \quad (3.4)$$

- 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 :

$$\sum_{(i,j) \in A} x_{ij} + \sum_{(j,i) \in A} x_{ji} \leq 1 \quad \forall i, j \in S \setminus \{s, t\} \quad (3.5)$$

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 ??.

2 A Star

3 Comparaison des deux implémentations

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 $G = \langle S, A \rangle$ 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 :

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

Contraintes

- Chaque sommet doit être relié à une arête entrante :

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

- Chaque sommet doit être relié à une arête sortante :

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

- Empêcher les sous-cycles :

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

Le code Python équivalent à ce modèle est donné en annexe 1.

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 pourrions 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.

```
1 def gen_tsp(n: int, p: float, file_name: str = "tsp.txt"):
2     """Generates a random TSP problem with n nodes and probability p of having an edge between nodes."""
3
4     nodes = []
5     cost = {}
6
7     # Generate nodes
8     for i in range(n):
9         nodes.append(i)
10
11    # Generate edges
12    for i in range(n):
13        for j in range(i+1, n):
14            if random.random() < p:
15                cost[(i, j)] = random.randint(10, 50)
16
17    # Write to file
18    Path("examples").mkdir(parents=True, exist_ok=True)
19    with open(f"examples/{file_name}", "w") as f:
20        f.write(f"{n} {len(cost)}\n")
21        for (i, j), c in cost.items():
22            f.write(f"{i} {j} {c}\n")
23
24    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 :

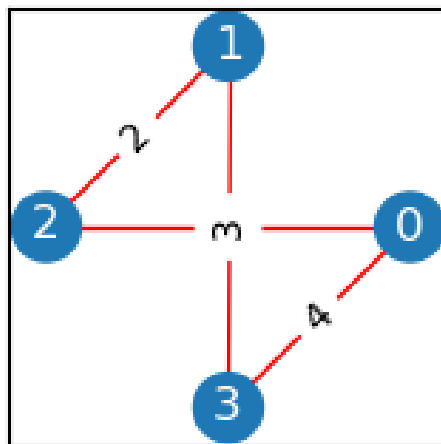


FIGURE 4.2 – Résolution d'un exemple simple de TSP

Chapitre 5

Conclusion

eee

Annexe A

Algorithmes et Code

1 Modélisation du Problème du Voyageur de Commerce

```
1 def tsp_cplex_solver(graph) -> Optional[List]:
2     """
3     Solve the Traveling Salesman Problem (TSP) using the CPLEX solver
4     :param graph: the graph to explore
5     :return: the cheapest path found
6     """
7     nodes = graph.node_list
8     costs = graph.cost
9
10    # Create a new model
11    mdl = Model('TSP')
12
13    # Create variables: x[i, j] is 1 if edge (i, j) is part of the solution
14    x = mdl.binary_var_dict(costs.keys(), name='x')
15
16    # Objective: Minimize the total cost of the tour
17    mdl.minimize(mdl.sum(x[i, j] * costs[i, j] for (i, j) in costs))
18
19    # Constraints: Each node must be entered and left exactly once
20    # Unique entering and leaving edges for each node
21    for k in nodes:
22        mdl.add_constraint(mdl.sum(x[i, k] for i in nodes if (i, k) in x) == 1, f'enter_{k}')
23        mdl.add_constraint(mdl.sum(x[k, j] for j in nodes if (k, j) in x) == 1, f'leave_{k}')
24
25    # No Sub tours
26    for i in nodes:
27        for j in nodes:
28            if i != j and (i, j) in x:
29                mdl.add_constraint(x[i, j] + x[j, i] <= 1, f'sub_tour_{i}_{j}')
30
31    # Solve the model
32    solution = mdl.solve()
33
34    # Check if a solution exists
35    if solution:
36        edges = [(i, j) for i, j in x if x[i, j].solution_value > 0.5]
37        return [edge[0] for edge in edges] + [edges[-1][1]]
38    else:
39        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).