GIRARD Simon - BERAL Quentin

Projet RO

TDM - Balade en ville

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

Table des matières

1 Modéliser, définir le problème formel, associer une classe de complexité

1.1 Modélisation

Afin de définir la notion de distance, nous proposons le premier graphe suivant :

Les sommets sont les intersections et adresses, tandis que les arrêtes sont les routes. On considère le métro comme une route et ses arrêts comme des intersections.

$$G = (V, E)$$

$$V = \{x_i | i \in [|\text{intersection} \cup \text{arrêt de métro} \cup \text{adresse}|]\}$$

$$E = \{\{x_j, x_k\} | \exists \text{ une route ou une ligne de métro entre } x_j \text{ et } x_k, j, k \in [|V|], x \neq k \text{ et } x_j, x_k \in V\}$$

Ce graphe est non orienté, et les adresses sont étiquetées. Nous définissons ainsi la distance d'une adresse à une autre par le nombre d'arrêtes sur le plus cours chemin entre ces adresses qui ne passe par aucune autre adresse. Il peut être déterminé à l'aide d'un Dijkstra.

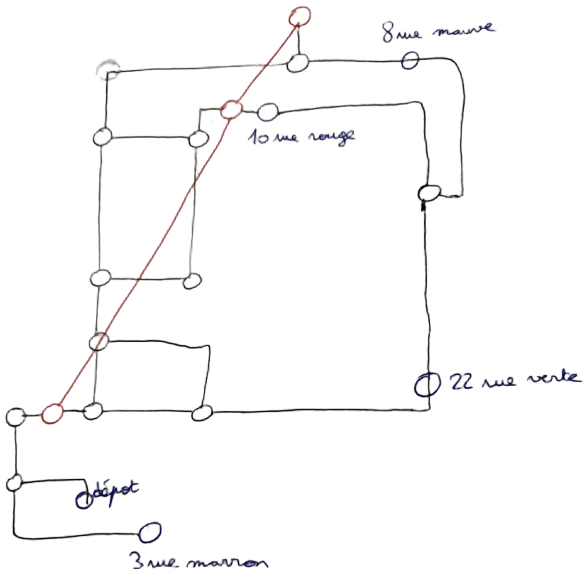

FIGURE 1 – Représentation sagittale

Nous proposons désormais un second graphe simplifié où seuls les adresses sont conservées en tant que noeuds, et que les adresses qui les relient sont valuées par la distance obtenue précédemment.

$$G_2 = (V_2, E_2)$$

$$V_2 = \{x_i | i \in [|\text{adresse}|]\}$$

$$E_2 = \{\{x_j, x_k\} | \exists \text{ un chemin entre } x_j \text{ et } x_k \text{ de distance } \rho(x_j, x_k), j, k \in [|V|], x \neq k \text{ et } x_j, x_k \in V\}$$

$\rho(x_j, x_k) =$ distance du plus court chemin de x_k à x_j qui ne passe par aucune autre adresse

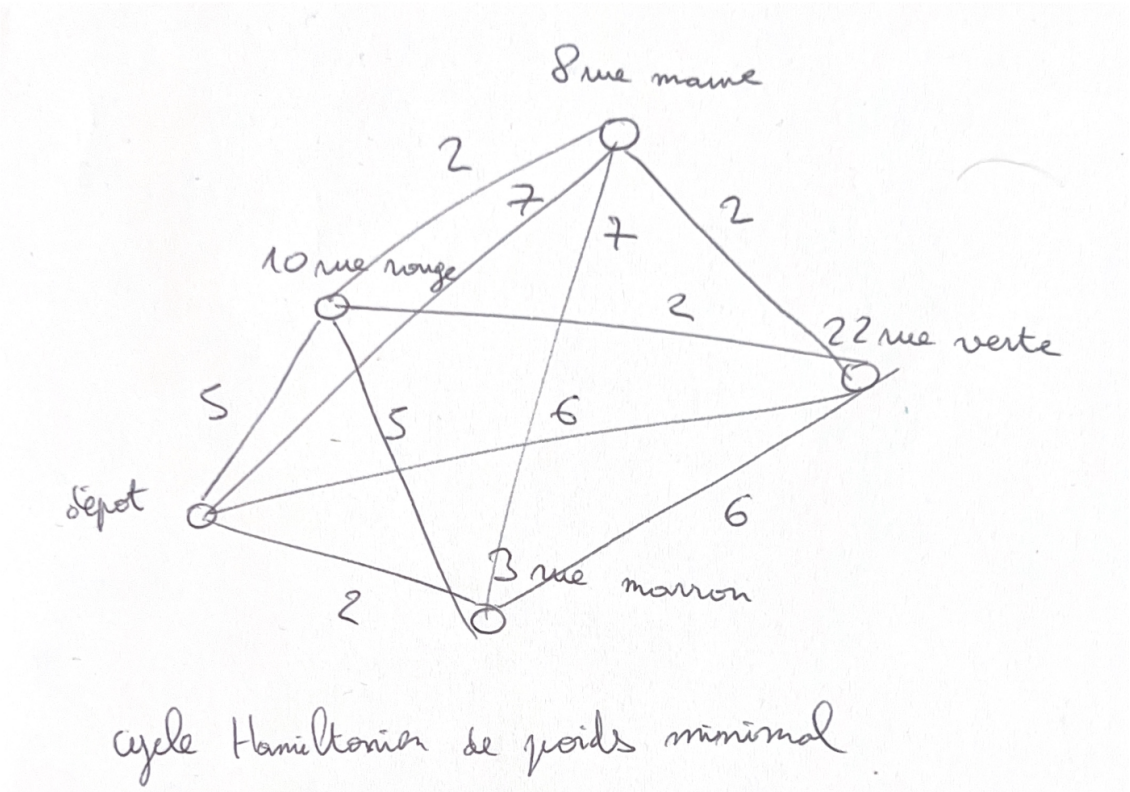

FIGURE 2 – Deuxième graphe valué

1.2 Problème formel sous-jacent

Trouver le cycle Hamiltonien de poids minimal qui passe par toutes les adresses, ce qui correspond à un problème du voyageur de commerce

1.3 Classe de complexité associée à ce problème

C'est un problème NP-difficile. L'énoncé nous demande de trouver toujours une solution exacte. Ceci-étant, ce n'est pas toujours possible suivant le nombre de nœuds. Il est envisageable de donner plutôt une solution approchée lorsque les graphes deviennent trop grands.

2 L'algorithme

2.1 État de l'art

Le problème du voyageur de commerce (TSP, pour Traveling Salesman Problem) est un des problèmes d'optimisation combinatoire les plus célèbres et les plus étudiés en informatique théorique, en recherche opérationnelle et en mathématiques discrètes. Formulé pour la première fois au début du 20ème siècle, il consiste à trouver le plus court chemin permettant de visiter un ensemble donné de villes exactement une fois et de revenir à la ville de départ.

Ce problème est NP-difficile, ce qui signifie qu'il est peu probable qu'il existe un algorithme de résolution efficace qui fonctionne pour toutes les instances de manière optimale en temps polynomial.

Le TSP a de nombreuses applications pratiques, notamment en logistique, en planification des tournées de véhicules, en fabrication et même en biologie computationnelle (par exemple, pour le séquençage de l'ADN). En raison de la complexité du problème, de nombreux algorithmes exacts et approchés ont été développés pour tenter de le résoudre, que ce soit pour des versions de petite ou de grande taille.

2.1.1 Modélisation et formulations mathématiques

Le TSP peut être formulé de plusieurs manières en tant que problème de graphe, où les villes représentent les nœuds et les distances (ou coûts de voyage) représentent les arêtes. Dans sa forme la plus simple, le TSP est modélisé sur un graphe complet pondéré, où chaque arête entre deux nœuds a un poids représentant la distance ou le coût de déplacement entre ces deux villes. L'objectif est de trouver un cycle hamiltonien minimal, c'est-à-dire un cycle qui visite chaque nœud exactement une fois et revient à son point de départ.

TSP Euclidien Une version spécifique du TSP est le TSP Euclidien, où les villes sont représentées par des points dans un espace euclidien, et les distances entre les villes sont les distances euclidiennes. Ce cas particulier est d'un grand intérêt pratique et théorique car les distances respectent la propriété de triangle, ce qui n'est pas toujours le cas dans les versions générales.

2.2 Algorithmes exacts

Les algorithmes exacts garantissent de trouver une solution optimale, mais leur complexité en temps est généralement exponentielle par rapport au nombre de villes, ce qui les rend peu pratiques pour des instances de grande taille.

Programmation dynamique : Algorithme de Bellman-Held-Karp L'une des approches exactes les plus connues est l'algorithme de Bellman-Held-Karp, basé sur la programmation dynamique. Cet algorithme a une complexité de $O(n^2 2^n)$, ce qui reste exponentiel, mais il réduit considérablement le temps de calcul par rapport à une énumération brute des solutions possibles. Il repose sur le fait que la solution optimale pour un sous-ensemble de villes peut être construite à partir de solutions optimales pour des sous-problèmes plus petits. [5]

Branch-and-Bound Le Branch-and-Bound est une autre approche exacte, qui consiste à explorer l'espace des solutions en formant un arbre de recherche. À chaque étape, l'algorithme divise le problème en sous-problèmes plus petits (branche) et utilise des bornes inférieures pour éliminer certaines branches non prometteuses (bound). Si une branche ne peut pas produire une meilleure solution que celle déjà trouvée, elle est éliminée. [3] Bien que cette méthode puisse être très efficace pour certaines instances, elle est toujours limitée pour des nombres élevés de villes.

Algorithmes de génération de colonnes et formulations linéaires Les formulations basées sur la programmation linéaire en nombres entiers (MILP, Mixed Integer Linear Programming) ont également été largement étudiées pour résoudre le TSP. Une formulation classique du TSP utilise des variables binaires pour représenter si une arête est incluse dans la solution. [2] Les contraintes incluent le fait que chaque ville doit être visitée exactement une fois et qu'il ne doit y avoir aucun sous-cycle.

Cependant, ces formulations peuvent mener à des relaxations du problème difficiles à résoudre directement, ce qui a conduit à l'usage de techniques comme la génération de colonnes et la séparation de coupes (comme les coupes de subtour) pour améliorer l'efficacité de la résolution.

2.3 Algorithmes approchés

Pour des instances de taille plus grande, où les algorithmes exacts deviennent inapplicables en raison de leur temps d'exécution exponentiel, on se tourne vers des algorithmes approchés, qui ne garantissent pas une solution optimale mais produisent des solutions de qualité proche de l'optimum dans un temps raisonnable.

Algorithmes gloutons Les algorithmes gloutons sont simples et rapides. Une approche courante est celle de l'algorithme "plus proche voisin" (nearest neighbor), qui commence par une ville aléatoire et sélectionne à chaque étape la ville la plus proche qui n'a pas encore été visitée. Bien que cet algorithme soit très rapide, il ne garantit pas une bonne approximation de l'optimum, surtout dans les versions non euclidiennes du TSP. [4] Une autre approche est de tenter de trouver un chemin qui suit les frontières.

Algorithmes de recherche locale Les techniques de recherche locale sont basées sur l'amélioration d'une solution initiale en effectuant des modifications locales (par exemple, échanger deux villes dans la tournée). Les algorithmes comme 2-opt et 3-opt sont des exemples classiques, où l'on tente d'améliorer une solution en supprimant respectivement deux ou trois arêtes et en les reconnectant de manière différente pour réduire le coût global de la tournée. [6]

Métaheuristiques Les métaheuristiques sont des algorithmes approchés plus sophistiqués, qui combinent des éléments de recherche locale avec des mécanismes globaux pour échapper aux minima locaux.

- Algorithmes génétiques : Inspirés par l'évolution naturelle, les algorithmes génétiques manipulent une population de solutions potentielles, et sélectionnent les meilleures solutions en combinant et mutant des individus pour explorer de nouvelles solutions.

- Recuit simulé (simulated annealing) : Inspiré du processus de refroidissement des métaux, le recuit simulé explore des solutions sous-optimales temporairement dans l'espoir de trouver de meilleures solutions à long terme. La probabilité d'accepter une solution pire diminue au fur et à mesure que l'algorithme progresse. [1]

- Algorithme des colonies de fourmis (Ant Colony Optimization) : Cet algorithme est basé sur le comportement collectif des fourmis cherchant de la nourriture. Les fourmis artificielles

déposent des phéromones sur des chemins courts, ce qui augmente la probabilité que d'autres fourmis suivent ces chemins, améliorant ainsi progressivement la qualité de la solution.

2.4 Pseudo-code de l'algorithme

La consigne impose très clairement l'utilisation d'un algorithme exact.

3 L'implémentation

Listing 1 – Transformation du graphe

```
# Fonction pour construire la matrice de distances en appelant Dijkstra
# pour chaque ville
def build_distance_matrix(graph, address_nodes):
    n = len(address_nodes)  # Nombre d'adresses a considerer
    distanceMatrix = [[0] * n for _ in range(n)]  # Initialiser une matrice n x n avec des zeros

    # Pour chaque adresse, utiliser Dijkstra pour calculer les distances vers toutes les autres
    for i in range(n):
        start_node = address_nodes[i]  # Prendre la izme adresse comme point de dzpart
        # Dijkstra retourne les distances du noeud 'start_node' a chaque autre noeud dans le graphe
        shortest_paths = dijkstra(graph, start_node)

        # Remplir la ligne i de la matrice de distance avec les resultats de Dijkstra
        for j in range(n):
            end_node = address_nodes[j]
            distanceMatrix[i][j] = shortest_paths[end_node]  # Distance entre start_node et end_node

    return distanceMatrix
```

Listing 2 – Résolution du TSP

```
# Fonction utilitaire pour calculer le cout du tour actuel
def calculate_tour_cost(tour, distance_matrix):
    cost = 0
    for i in range(len(tour) - 1):
        cost += distance_matrix[tour[i]][tour[i+1]]
    cost += distance_matrix[tour[-1]][tour[0]]  # Retour a la ville de depart
    return cost

# Fonction pour trouver une borne inferieure en utilisant une approximation, ici un simple MST
def calculate_lower_bound(current_tour, distance_matrix, n):
    min_cost = 0
    # Trouver les plus petites aretes possibles a partir de chaque ville non visitee
    for i in range(n):
        if i not in current_tour:
            min_edge = min([distance_matrix[i][j] for j in range(n) if j != i and j not in current_tour])
            min_cost += min_edge
    return min_cost

# Algorithme de Branch-and-Bound
def branch_and_bound(distance_matrix):
    n = len(distance_matrix)  # Nombre de villes
    best_tour = None          # Solution optimale
    best_cost = sys.maxsize   # Cout de la solution optimale
    queue = []                # File pour explorer les branches (LIFO)

    # Initialisation : commencer par la premiere ville (0)
    initial_tour = [0]
    initial_bound = calculate_lower_bound(initial_tour, distance_matrix, n)
    queue.append((initial_tour, initial_bound))  # Ajouter le premier noeud

    # Boucle de l'algorithme Branch-and-Bound
    while queue:
        # Extraire la tournee courante
        current_tour, current_bound = queue.pop()

        # Si toutes les villes sont visitees,
        # calculer le cout de cette tournee
        if len(current_tour) == n:
            current_cost = calculate_tour_cost(current_tour, distance_matrix)

            # Si on trouve une meilleure solution, on l'enregistre
            if current_cost < best_cost:
                best_cost = current_cost
                best_tour = current_tour
            continue
```

```
        # Si la borne actuelle est meilleure que le meilleur
        # cout trouve, on explore plus loin
        if current_bound < best_cost:
            for city in range(n):
                if city not in current_tour:
                    # Generer un nouveau sous-tour en ajoutant une ville non visitee
                    new_tour = current_tour + [city]

                    # Calculer la borne inferieure de ce sous-tour
                    new_bound = calculate_lower_bound(new_tour, distance_matrix, n)

                    # Si cette branche peut encore conduire a une
                    # meilleure solution, on continue
                    if new_bound < best_cost:
                        queue.append((new_tour, new_bound))

    return best_tour, best_cost
```

4 L'adaptation

Dans notre modèle, la ville est représentée par un graphe unique intégrant à la fois les routes et le métro. Chaque sommet du graphe correspond à un point de la ville (stations, carrefours), et chaque arc représente une connexion, qu'elle soit via une route ou une ligne de métro.

Quand le métro tombe en panne, notre solution a consisté à modifier le graphe en supprimant les arcs représentant les lignes de métro. Cela permet de conserver le même algorithme de calcul de trajet, mais sur un graphe réduit, où seules les routes sont disponibles pour le déplacement. Cette approche simplifie la gestion de la panne, car elle ne nécessite que de changer la structure du graphe.

Une alternative aurait été de changer l'implémentation sans modifier le graphe directement. Par exemple, nous aurions pu introduire des valuations négatives (ou très élevées) sur les arcs représentant les lignes de métro pour les rendre beaucoup moins attractifs dans le calcul du trajet. Cette méthode aurait permis de "désactiver" le métro sans supprimer ces arcs, mais aurait rendu l'algorithme un peu plus complexe à gérer, notamment en termes de gestion des coûts et des évaluations. Une autre solution aurait été d'introduire des conditions spécifiques dans l'algorithme pour ignorer les arcs correspondant au métro lorsque celui-ci est en panne, mais cela aurait nécessité des ajustements plus importants au niveau de l'implémentation.

Enfin, pour obtenir les deux résultats demandés (avec et sans métro), nous avons exécuté l'algorithme une première fois sur le graphe complet (routes et métro) et une seconde fois sur le graphe modifié sans les lignes de métro. Cela permet de visualiser les trajets avec et sans cette option de transport, en tenant compte de la panne.

5 Conclusion

L'algorithme fonctionne bien sur l'exemple donné et produit des résultats déterministes. Les résultats avec et sans l'optimisation sont, comme prévu, tout aussi bons. La solution optimale est bien retournée par le programme.

Il est difficile d'évaluer le gain de temps induit par l'optimisation à cause de la petite taille de l'exemple, qui fait que les mesures varient énormément.

Références

[1] Hüsamettin Bayram1 et Ramazan Şahin. *A new simulated annealing approach for the traveling salesman problem.* URL : https://www.researchgate.net/publication/283497778_A_new_simulated_annealing_approach_for_the_traveling_salesman_problem.

[2] Matteo Fortini. *LP-based Heuristics for the Traveling Salesman Problem (chapter 3).* URL : https://amsdottorato.unibo.it/339/1/tesi.pdf.

[3] Pierre Fouilhoux. *Branchement et Evaluation (Branch-and-Bound).* URL : https://webia.lip6.fr/~fouilhoux/MAOA/Files/MAOA_ROOC_branch_impr.pdf.

[4] Robb T. Koether. *The Traveling Salesman Problem Nearest-Neighbor Algorithm.* URL : https://people.hsc.edu/faculty-staff/robbk/Math111/Lectures/Fall%202016/Lecture%2033%20-%20The%20Nearest-Neighbor%20Algorithm.pdf.

[5] Quang Nhat Nguyen. *Travelling Salesman Problem and Bellman-Held-Karp Algorithm.* URL : https://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf.

[6] Jaap J. A. Slootbeek. *Average-Case Analysis of the 2-opt Heuristic for the TSP.* URL : https://essay.utwente.nl/72060/1/Slootbeek_MA_EEMCS.pdf.