Sorbonne Université

Résolution de Problèmes
M1

Rapport The Covering Canadian Traveller Probleme

Étudiant-1 :
Ziming Zhao
21401298
ANDROIDE

Étudiant-2 :
Yassin Lahbib
28718491
ANDROIDE

27/04/2025

Table des matières

Table des figures

1 Introduction

Dans ce projet, nous nous intéressons au problème du voyageur de commerce avec contraintes de blocages sur les arêtes du graphe. L'objectif est de construire des tournées de coût minimal en s'adaptant dynamiquement aux blocages rencontrés lors du parcours.

Pour générer les instances de graphe nécessaires aux expérimentations, deux méthodes principales ont été utilisées :

- **Lecture de fichier :** nous avons implémenté une fonction permettant de parser des fichiers au format standard, afin d'extraire les coordonnées des différentes villes. Cette méthode permet de travailler sur des instances réalistes provenant de bases de données existantes.
- **Génération aléatoire :** nous avons également développé une fonction capable de générer aléatoirement des positions de villes dans un espace bidimensionnel, avec des coordonnées distribuées uniformément dans une zone carrée. Les distances entre les villes sont alors calculées à l'aide de la distance euclidienne, garantissant ainsi que l'**inégalité triangulaire** est respectée.

Le respect de l'inégalité triangulaire est essentiel pour assurer la validité des algorithmes utilisés qui reposent sur cette propriété pour construire des chemins approximativement optimaux.

Dans les sections suivantes, nous présenterons l'algorithme **Cyclic Nearest Neighbor (CNN)**, que nous avons utilisé et adapté pour résoudre les instances générées, puis nous analyserons ses performances expérimentales en comparaison avec l'algorithme **Constructive Reasoning (CR)**.

2 Introduction à l'algorithme de Christofides

L'algorithme de Christofides est une méthode classique pour trouver une solution approximative au problème du voyageur de commerce (TSP) symétrique. Il garantit que la longueur du chemin final ne dépasse pas 1,5 fois celle de la solution optimale. Le processus global peut être divisé en plusieurs étapes principales :

- Construction de l'arbre couvrant de poids minimum (MST) ;
- Identifier les sommets de degré impair ;
- Appariement parfait minimum sur les sommets impairs ;
- Fusionner MST et appariement pour obtenir un graphe eulérien ;
- Trouver un circuit eulérien ;
- Transformer le circuit eulérien en cycle hamiltonien ;

2.1 Fonction `ACPM()`

La fonction `ACPM()` construit un **arbre couvrant de poids minimum** (MST) en utilisant une approche inspirée de l'algorithme de Prim :

- On démarre d'un sommet initial (par défaut 0), et on construit progressivement l'arbre.
- À chaque étape, on ajoute au MST l'arête de coût minimal reliant un sommet déjà sélectionné à un sommet encore non sélectionné.
- On maintient la liste des arêtes sélectionnées et la structure de l'arbre sous forme d'un dictionnaire d'adjacence.

2.2 Fonction `odd_degree_vertex()`

Après avoir construit le MST, la fonction `odd_degree_vertex()` identifie tous les sommets de degré impair :

- Chaque sommet est examiné selon son nombre de voisins.
- Si le nombre d'arêtes incidentes est impair, le sommet est ajouté à la liste des sommets impairs.

2.3 Fonctions `inducted_distance_matrix` et `minimum_weight_perfect_matching`

Après avoir trouvé les sommets impairs, nous cherchons à effectuer un **appariement parfait de poids minimum** :

- `inducted_distance_matrix` extrait la sous-matrice des distances entre les seuls sommets impairs à partir de la matrice initiale.
- `minimum_weight_perfect_matching` construit un graphe complet pondéré entre les sommets impairs, puis utilise NetworkX pour calculer un appariement parfait de coût minimal.

2.4 Fonction `union_matching_ACPM`

La fonction `union_matching_ACPM` combine les arêtes du MST et celles du couplage parfait afin de créer un **multigraphe eulérien** où tous les sommets ont un degré pair.

2.5 Fonction `euler_tour`

À partir du multigraphe obtenu, `euler_tour` utilise l'algorithme de Hierholzer pour trouver un **circuit eulérien**, en supprimant progressivement les arêtes traversées et en empilant les sommets parcourus.

2.6 Fonction `euler_to_hamiltonian`

Enfin, `euler_to_hamiltonian` transforme le circuit eulérien en un **cycle hamiltonien** :

- En parcourant le tour eulérien, chaque sommet est ajouté à la tournée uniquement la première fois où il est rencontré.
- Le sommet de départ est réajouté en fin de parcours pour fermer la boucle.

3 Algorithme routage cyclique (CR)

3.1 Introduction

Après avoir implémenté l'algorithme de Christofides, nous avons développé l'algorithme **routage cyclique (CR)**. L'algorithme CR repose sur un chemin hamiltonien produit par Christofides et ajuste dynamiquement son parcours en fonction des arêtes connues comme ouvertes ou bloquées, afin d'éviter les obstacles et de retourner au point de départ.

Le processus général est :

- Partir du point initial et tenter de rejoindre le prochain sommet dans l'ordre donné ;
- Si l'accès direct est impossible, chercher un raccourci via les nœuds déjà visités ;
- Visiter un groupe de sommets à chaque ronde, changer dynamiquement de direction selon certaines conditions (empêchement ou risque de boucle) ;
- Après avoir visité tous les sommets, retourner explicitement au point de départ.

Toutes les fonctionnalités sont encapsulées dans la classe `ConstructiveReasoning`.

3.2 Structure générale de l'algorithme

- `__init__`(graph, hamiltonian_path) : Initialiser le graphe et les variables d'état (current, path, visited, unvisited, direction, round_count, max_rounds).
- `get_subpath_nodes`(src, dest, direction) : Identifier les nœuds déjà visités entre src et dest selon la direction.
- `try_shortcut`(src, dest, direction) : Chercher un chemin direct ou via d'autres nœuds visités (BFS).
- `run`() : Boucle principale :
  - Obtenir la liste ordonnée des nœuds à visiter ;
  - Vérifier si un changement de direction est nécessaire ;
  - Construire dynamiquement le chemin pour chaque ronde ;

    - Retourner au point de départ à la fin.
- `_get_ordered_unvisited`() : Retourner les nœuds non visités triés selon la direction actuelle.
- `_return_to_start`() : Revenir au point de départ après avoir tout visité.

3.3 Détail des modules

3.3.1 Fonction `get_subpath_nodes`

La fonction `get_subpath_nodes` identifie les nœuds déjà visités entre deux sommets (`src` et `dest`) sur le chemin Hamiltonien, en fonction de la direction de parcours :

- Si `dest` est le successeur immédiat de `src` (en sens normal), aucun nœud intermédiaire n'est nécessaire.
- Sinon, parcourir le chemin de `src` à `dest` (sens normal ou inversé), en collectant les nœuds visités.

3.3.2 Fonction `try_shortcut`

La fonction `try_shortcut` cherche un chemin entre deux sommets :

- Si une connexion directe existe entre `src` et `dest`, elle est utilisée.
- Sinon, une recherche en largeur (BFS) est réalisée en utilisant uniquement les nœuds intermédiaires visités pour tenter de relier `src` à `dest`.

3.3.3 Fonction `run`

La fonction `run` pilote l'algorithme Constructive Reasoning :

- Tant qu'il reste des nœuds non visités :
    - Générer la liste ordonnée des nœuds accessibles.
    - Décider d'un éventuel changement de direction selon deux critères :
        - Le début de la ronde diffère de la fin de la précédente.
        - Le même ensemble de nœuds serait visité en changeant de direction.
    - Tenter de construire des raccourcis pour visiter les nœuds.
    - Mettre à jour le chemin et l'état des nœuds visités.
- À la fin, revenir au point de départ pour fermer la boucle.

3.4 Fonction `_get_ordered_unvisited`

Cette fonction trie les nœuds non visités selon l'ordre ou l'ordre inverse du chemin Hamiltonien, en fonction de la direction actuelle.

3.4.1 Fonction `_return_to_start`

Dans la phase finale, `_return_to_start` assure le retour explicite au sommet de départ :

- Tenter d'abord dans la direction actuelle.
- Si nécessaire, inverser la direction et réessayer.
- Ajouter le chemin de retour trouvé au parcours final.

4 Algorithme Cyclic Nearest Neighbor (CNN)

4.1 Présentation de l'algorithme

L'algorithme **Cyclic Nearest Neighbor (CNN)** est une approche constructive destinée à résoudre le problème du *voyageur cyclique avec contraintes de blocages*. Il repose sur l'utilisation d'un *chemin de référence* calculé à l'aide de l'algorithme de **Christofides**, sans connaissance préalable des arêtes bloquées.

Le principe de CNN est de suivre autant que possible l'ordre donné par Christofides :

— À chaque sommet, l'algorithme tente de rejoindre directement le **prochain sommet prévu**.

— Si l'arête est bloquée, il choisit parmi les **voisins non visités** celui dont la distance est la plus courte.

— Ce processus se poursuit jusqu'à ce que tous les sommets soient visités, puis l'algorithme revient au sommet de départ pour fermer le cycle.

Cette stratégie permet à CNN de s'adapter localement aux blocages sans recalculer entièrement son itinéraire. En contrepartie, lorsque les blocages sont nombreux, la prise de décisions purement locale peut conduire à des parcours sensiblement moins optimaux.

Pour notre projet, nous nous sommes appuyés sur l'article *The Covering Canadian Traveller Problem Revisited* de Hahn et Xefteris, en particulier sur le pseudocode proposé, que nous avons adapté à nos besoins spécifiques.

5 Études

5.1 Analyse de robustesse face aux blocages

L'objectif de cette étude est d'analyser la capacité des algorithmes à s'adapter face à un grand nombre d'arêtes bloquées.

Nous avons fait varier le nombre k d'arêtes bloquées afin d'observer l'évolution du coût moyen des tournées. Les expérimentations ont été réalisées sur un graphe de 300 sommets, en effectuant 5 exécutions indépendantes pour chaque valeur de k, afin d'obtenir une estimation fiable du coût moyen.

Les résultats montrent que l'algorithme **CNN** génère systématiquement des tournées de coût inférieur à celles obtenues par **CR**, indiquant ainsi de meilleures performances globales. La courbe de variation du coût pour CR présente des pics plus marqués, traduisant une sensibilité plus importantes aux blocages. Cependant, les deux algorithmes suivent globalement la même tendance d'évolution du coût en fonction du nombre d'arêtes bloquées. CNN conservant en moyenne des coûts systématiquement plus faibles que CR.

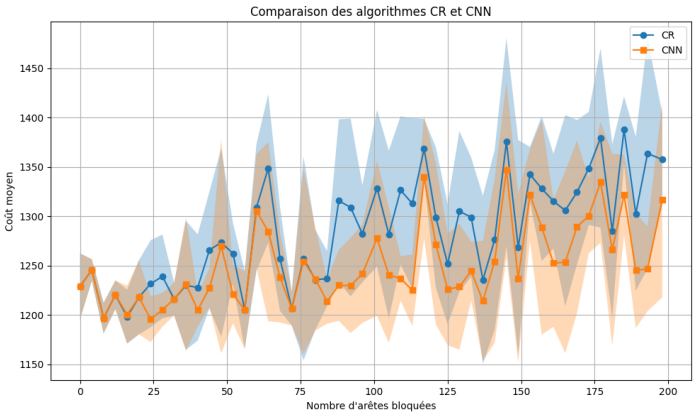


FIGURE 1 – Évolution du coût moyen en fonction du nombre d'arêtes bloquées pour un graphe de 300 sommets

5.2 Impact de la taille du graphe sur le coût moyen

L'objectif de cette étude est de comparer le coût moyen des tournées obtenues par les algorithmes **CR** et **CNN** en faisant varier la taille du graphe. Ici, le nombre d'arêtes bloquées est fixé à $k = n - 2$, où n représente le nombre de sommets du graphe.

Les résultats montrent que le coût moyen augmente avec la taille du graphe, ce qui est attendu compte tenu de la complexité croissante du problème. Globalement, **CNN** génère des tournées de coût

plus faible que **CR**. Toutefois, on observe quelques pics où le coût trouvé par CNN dépasse ponctuellement celui obtenu par CR. Malgré ces fluctuations locales, les deux courbes restent relativement proches, avec une tendance générale favorable à CNN.

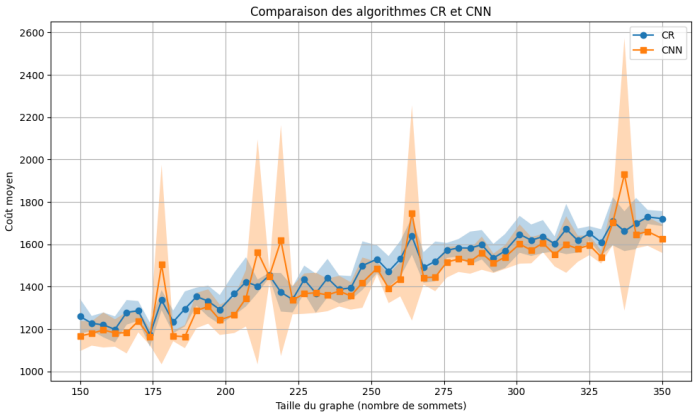


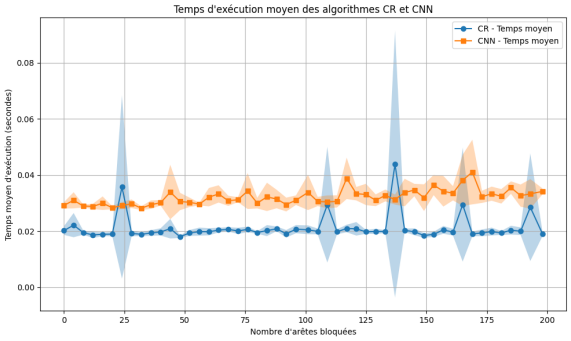
FIGURE 2 – Évolution du coût moyen en fonction de la taille du graphe

5.3 Temps moyen d'exécution des algorithmes

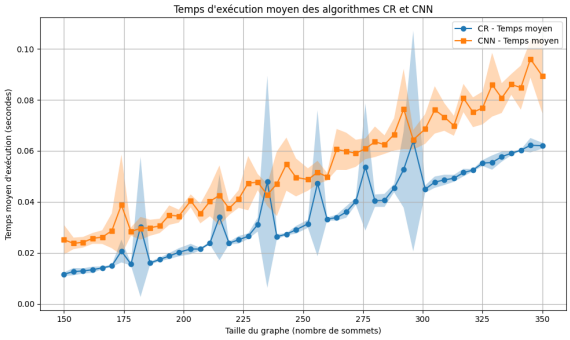
Nous avons mesuré le temps moyen nécessaire à l'exécution des algorithmes **CR** et **CNN**.

La figure (a) montre que le nombre d'arêtes bloquées n'a pas d'influence significative sur le temps d'exécution des algorithmes pour un graphe de 300 sommets. En revanche, comme illustré dans la figure (b), lorsque le nombre de sommets augmente, le temps moyen d'exécution croît de manière approximativement linéaire.

Nous observons également que l'algorithme **CNN** est globalement plus lent que **CR**, bien que quelques pics apparaissent ponctuellement sur la courbe de CR.



(a) Temps moyen en fonction du nombre d'arêtes bloquées (graphe de 300 sommets)



(b) Temps moyen en fonction de la taille du graphe

FIGURE 3 – Évolution du temps moyen d'exécution des algorithmes CR et CNN

La différence de temps d'exécution entre CNN et CR vient principalement de leur manière de gérer les blocages. Lorsqu'un blocage est rencontré, **CR** utilise une méthode rapide : il modifie légèrement son chemin sans chercher à optimiser à chaque étape.

À l'inverse, **CNN** doit, à chaque blocage, explorer les voisins disponibles pour trouver le sommet le plus proche accessible. Cela revient à résoudre un petit problème de plus court chemin local, ce qui demande plus de calculs.

Cette manière de procéder permet à CNN de trouver de meilleurs chemins en général, mais elle rend l'algorithme plus lent que CR, surtout quand il y a beaucoup de blocages ou lorsque le graphe est grand.

6 Conclusion générale

Dans ce projet, nous avons comparé l'algorithme **Cyclic Nearest Neighbor (CNN)** à **Constructive Reasoning (CR)** pour résoudre des instances du voyageur de commerce avec blocages.

Les résultats expérimentaux montrent que **CNN** génère en moyenne des tournées de meilleur coût que **CR**, tout en restant relativement proche en temps d'exécution

En définitive, **CNN** s'impose comme une approche plus efficace pour ce type de problème, en conciliant qualité des solutions et adaptation aux perturbations.