Rapport Projet RP

Zeyu TAO
Jiahua LI

April 2025

1 Introduction

Ce projet porte sur une généralisation du problème du voyageur canadien couvrant(CCTP), appelée k-CCTP. Dans ce cadre, un voyageur doit visiter tous les sommets d'un graphe complet et revenir à son point de départ, tout en découvrant dynamiquement qu'au plus k routes peuvent être bloquées. Deux algorithmes ont été implémentés dans ce projet : CR et CNN (détaillés dans la section suivante). L'objectif est d'évaluer et de comparer leur performance.

2 Implémentation

Durant ce projet, nous avons besoin d'utiliser les différents types de graphes, tels que graphe non orienté pondéré, graphe orienté pondéré et multigraphe. Pour faciliter notre implémentation, nous avons décidé d'utiliser les structures de graphe fournies par la bibliothèque **NetworkX**.

2.1 Implémentation des algorithmes

2.1.1 Christofides

Nous avons commencé par implémenter l'algorithme de Christofides, un algorithme d'approximation pour le problème du voyageur de commerce (TSP), avec un facteur d'approximation $\frac{3}{2}$. Les deux algorithmes, CR et CNN ont utilisé le résultat généré par Christofides - un circuit hamiltonien, qui passe exactement une fois par tous les sommets du graphe - comme base pour calculer ensuite un itinéraire qui passe par tous les sommets (au moins une fois) et retourne au point de départ aussi vite que possible dans un environnement incertain avec des arêtes potentiellement bloquées.

Nous avons implémenté plusieurs algorithmes dans le cadre de l'algorithme de Christofides, notamment l'algorithme de Prim pour créer un arbre couvrant de poids minimum, ainsi qu'un algorithme inspiré de celui d'Hierholzer pour déterminer un circuit eulérien. Pour l'étape du couplage parfait de poids minimum, nous avons expérimenté plusieurs approches possibles, comme l'algorithme d'Hongrois, qui malheureusement ne s'applique qu'aux graphes bipartis, l'algorithme glouton, qui fournit bien une solution réalisable, mais sans garantie d'optimalité et l'algorithme de Blossom, qui est l'algorithme classique pour résoudre le problème du couplage parfait de poids minimum ou maximum, mais très compliqué à implémenter. Finalement, nous avons décidé d'utiliser la fonction **networkx.min_weight_matching()** de la bibliothèque **NetworkX** pour cette étape, celle-ci étant une implémentation de l'algorithme de Blossom.

2.1.2 CR (Cyclic Routing)

Nous avons ensuite implémenté l'algorithme CR, un algorithme itératif reposant sur le principe suivant : il commence par calculer un circuit hamiltonien (une tournée) en utilisant l'algorithme

de Christofides. Nous notons cette tournée $P : s \to v_1 \to v_2 \to \cdots \to v_n \to s$.

L'algorithme CR procède ensuite de manière itérative jusqu'à ce que tous les sommets du graphe soient visités. À chaque itération m, il tente de visiter tous les sommets non encore visités en suivant un chemin P_m, construit à partir de la tournée P, dans le sens des aiguilles d'une montre ou inversement. Plus précisément, P_m commence au dernier sommet visité à l'itération $m-1$, puis suit l'ordre de P en ne prenant que les sommets non visités.

Lorsqu'une arête $\{u, v\}$ se trouve bloquée, l'agent tente de contourner le blocage en empruntant un sommet w déjà visité à l'itération précédente, à condition que w se trouve entre u et v dans P, c'est-à-dire, qu'il existe une sous chemin $u \to \cdots w \cdots \to v$, et que les arêtes $\{u, w\}$ et $\{w, v\}$ ne soient pas bloquée. Si un tel w n'existe pas, le sommet v est ignoré pour cette itération (i.e. il reste non visité pour l'itération courante) et l'agent passe au sommet suivant.

À la fin de la dernière itération, soit t le dernier sommet visité. Si l'arête $\{t, s\}$ (avec s le sommet de départ) est libre, l'agent l'emprunte pour retourner à la source et terminer l'algorithme. Dans le cas contraire, il cherche un sommet w déjà visité (peu importe lequel, puisque tous les sommets sont visités), tel que les arêtes $\{t, w\}$ et $\{w, s\}$ soient non bloquées, afin de rejoindre la source en contournant le blocage.

2.1.3 CNN (Christopides Nearest Neighbor)

Enfin, nous avons implémenté l'algorithme CNN, un algorithme qui combine l'algorithme de Christofides et l'algorithme Nearest Neighbor. L'algorithme CNN se déroule en quatre étapes principales : il commence par calculer une tournée en utilisant l'algorithme de Christofides, cette étape suit exactement le même processus que dans l'algorithme CR. Nous notons également cette tournée $P : s \to v_1 \to v_2 \to \cdots \to v_n \to s$.

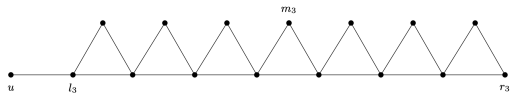
Ensuite, l'algorithme tente de suivre cette tournée dans l'ordre défini, en sautant les sommets inaccessibles en raison de blocages, c'est-à-dire, lorsqu'une arête $\{u, v\}$ est bloquée, l'agent ignore directement le sommet v et passe au sommet suivant dans la tournée, aucune tentative de contournement n'est effectuée à cette étape. Cette étape est très similaire à la première itération de l'algorithme CR, à la différence près que CNN mémorise également, en parallèle, toutes les arêtes bloquées adjacentes aux sommets visités. À la fin de cette étape, l'algorithme tente de revenir à la source, soit v' le dernier sommet visité, et si l'arête $\{v', s\}$ ne soit pas bloquée, l'agent l'emprunte pour retourner directement à la source. Dans le cas contraire, il retourne en arrière en suivant le chemin inverse emprunté jusqu'à v'. Par exemple : soit $P' : s \to v_1 \to v_2 \to \cdots \to v'$, le chemin emprunté pour atteindre le sommet v'. Si l'arête $\{v', s\}$ n'est pas bloquée, alors on construit un chemin $P_1 : s \to v_1 \to v_2 \to \cdots \to v' \to s$. Sinon l'agent retourne en arrière en suivant exactement le chemin inverse, et on obtient $P_1 : s \to v_1 \to v_2 \to \cdots \to v' \to \cdots \to v_2 \to v_1 \to s$.

Soit U l'ensemble des sommets non visités. Dans cette nouvelle étape, l'algorithme construit un **multigraphe** $G' = (U_s, E')$, où $U_s = U \cup \{s\}$ inclut tous les sommets non visités ainsi que la source. Pour chaque couple $(u, v) \in U_s^2$, il existe au plus deux arêtes qui les relient : la première étant l'arête originale entre u et v, et la seconde étant un plus court chemin entre u et v, où seuls les sommets déjà visités peuvent être utilisés comme sommets intermédiaires, de plus, les informations concernant les blocages découverts sont utilisées pour garantir que ce plus court chemin reste accessible. Nous avons implémenté l'algorithme de Dijkstra pour le calcul du plus court chemin dans cette étape.

Enfin, la dernière étape de cet algorithme consiste à appliquer l'algorithme Nearest Neighbor sur le multigraphe G' afin de trouver une tournée qui part du sommet source, visite tous les sommets de G', et revient ensuite au sommet source. Nous avons implémenté une première version de l'algorithme Nearest Neighbor, qui, à chaque itération, choisit le sommet non encore exploré dont la distance au sommet courant est minimale et qui ne soit pas bloqué, c'est une

approche purement gloutonne. Nous notons P_2 la tournée retournée par l'algorithme Nearest Neighbor, puis nous concaténons P_1 et P_2 afin de former une solution au problème du k-CCTP.

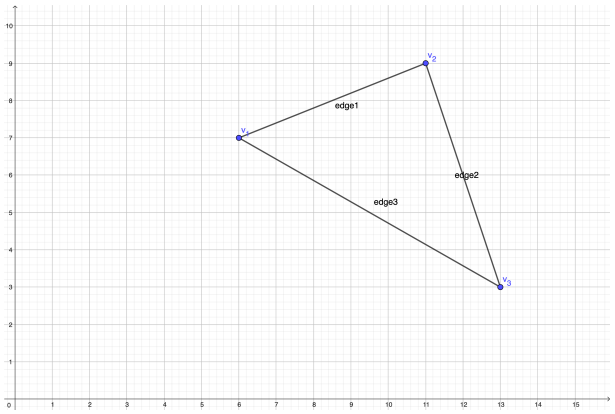
Nous avons constaté que cette dernière étape (algorithme Nearest Neighbor) ne fonctionne pas toujours. Par exemple, considérons l'instance décrite dans le papier de CNN, où seul le sommet u a été visité lors de la deuxième étape, et l_3 constitue le sommet source. L'application directe de notre algorithme Nearest Neighbor conduit à un chemin en zigzag allant de l_3 à r_3. Une fois l'agent atteint le sommet r_3, il devient impossible de revenir à l_3, puisqu'il n'existe pas d'arête reliant directement ces deux sommets, et il n'existe pas non plus de plus court chemin accessible en utilisant uniquement le sommet u comme intermédiaire. Pour traiter ce cas particulier, et pour que notre algorithme fonctionne pour cette instance décrite, nous avons modifié la dernière étape de notre algorithme, si aucun chemin ne permet de revenir à la source à la fin de l'itération, l'agent effectue un retour arrière en empruntant le chemin inverse comme la stratégie utilisée dans la deuxième étape. Malheureusement, il reste possible que certaines instances et certains choix inappropriés conduisent notre algorithme à l'échec. Par exemple, toujours considérons l'instance ci-dessous, si notre algorithme décide de prendre le chemin tout droite de l_3 à r_3, dans ce cas, notre algorithme échoue et retourne une solution qui ne visite pas tous les sommets. Nous n'avons pas adopté de stratégie spécifique pour résoudre ce problème, puisque cela rendrait l'algorithme plus complexe et il ne serait alors plus un algorithme Nearest Neighbor.



2.2 Création des instances de tests

Dans le cadre de ce projet, nous travaillons avec des hypothèses et des contraintes particulières s'appliquent à nos instances. Premièrement, l'état d'une arête quelconque ne change plus une fois qu'elle a été explorée par l'agent. Deuxièmement, le graphe doit rester connecté même après l'élimination des toutes les arêtes bloquées. Ces deux hypothèses sont imposées par les algorithmes CR et CNN, pour garantir ces deux hypothèses, il suffit, lors de la construction des instances, de vérifier qu'à chaque fois qu'une arête est marquée comme bloquée, le graphe doit rester connecté même en éliminant cette arête, et de ne plus modifier l'état des arêtes après la construction de l'instance.

De plus, une autre contrainte, imposée par l'algorithme de Christofides, qui exige que le graphe doit respecter l'inégalité triangulaire pour garantir le facteur d'approximation de $\frac{3}{2}$. Pour satisfaire cette contrainte, nous adoptons la méthode suivante : considérons le graphe $G = (V, E)$, où chaque sommet $v \in V$ est associé à une coordonnée $(x, y) \in \mathbb{N}^2$, et le poids d'une arête $e = \{u, v\}$, avec $u = (x_1, y_1)$ et $v = (x_2, y_2)$, est défini par leur distance d'euclidienne : $d(u, v) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Par construction, tout triplets $(v_1, v_2, v_3) \in V^3$ forme un triangle dans l'espace, et comme le poids des arêtes : $\{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}\}$ sont données par leurs distances d'euclidienne, donc l'inégalité triangulaire est nécessairement satisfait (Voici la figure ci-dessous).



3 Méthodologie expérimentale

3.1 Variables et notations

— n : taille du graphe (nombre de sommets) ;
— k : nombre d'arêtes bloquées ;
— Les performances sont évaluées en temps de calcul et qualité de la solution(le poids total du chemin obtenu).

3.2 Plan d'expériences

1. **Influence de k à taille de graphe fixe**

   — Paramètre fixé : $n = 50$.
   — Variable étudiée : $k \in \{\lceil 0.1n \rceil, \lceil 0.2n \rceil, \ldots, \lceil 0.9n \rceil, n-2\}$.
   — Objectif : analyser la sensibilité des algorithmes `CR` et `CNN` face à l'augmentation progressive de la proportion d'arêtes bloquées.

2. **Influence de n pour un k "maximal"($k = n-2$)**

   — Paramètre fixé : $k = n - 2$.
   — Variable étudiée : $n \in \{10, 12, 14, \ldots, 50\}$.
   — Objectif : évaluer la sensibilité des algorithmes lorsque la taille du graphe croît.

3. **Scénario à blocage très élevé**

   — Condition modifiée : $k = \lfloor 0.2n^2 \rfloor$, sous réserve que le problème demeure soluble.
   — Variable étudiée : n identique au deuxième protocole.
   — Objectif : étudier la robustesse des deux algorithmes lorsque la proportion d'arêtes bloquées atteint un niveau extrême.

3.3 Procédure expérimentale

Dans ce projet, nous avons choisi de nous concentrer sur l'évaluation empirique des performances moyennes des algorithmes CR et CNN. Les instances correspondant au cas extrême sont rares. De plus, les deux articles de référence fournissent déjà une analyse rigoureuse du cas extrême, accompagnée de démonstrations formelles. Ainsi, nous avons réalisé une approche basée sur des graphes aléatoires pour observer le comportement moyen des algorithmes.

Pour chaque configuration (n, k), nous générons aléatoirement 50 graphes distincts. Sur chacun d'eux, nous choisissons uniformément 5 sommets comme sources et exécutons successivement les

algorithmes `CR` et `CNN`. Chaque exécution produit une mesure individuelle ; les 50×5 observations ainsi obtenues pour une même configuration sont ensuite agrégées (médiane ou moyenne) afin de comparer les performances.

Dans un premier temps, nous avons utilisé la moyenne pour agréger les résultats. Cependant, en raison de la présence de valeurs dispersées dans certaines configurations (notamment pour l'algorithme `CNN`), nous avons choisi d'utiliser principalement la médiane, plus robuste pour comparer les performances.

L'ensemble des données est exporté au format `CSV` pour une analyse ultérieure.

4 Analyse comparative des algorithmes `CR` et `CNN`

4.1 Influence de k pour une taille de graphe fixe ($n = 50$)

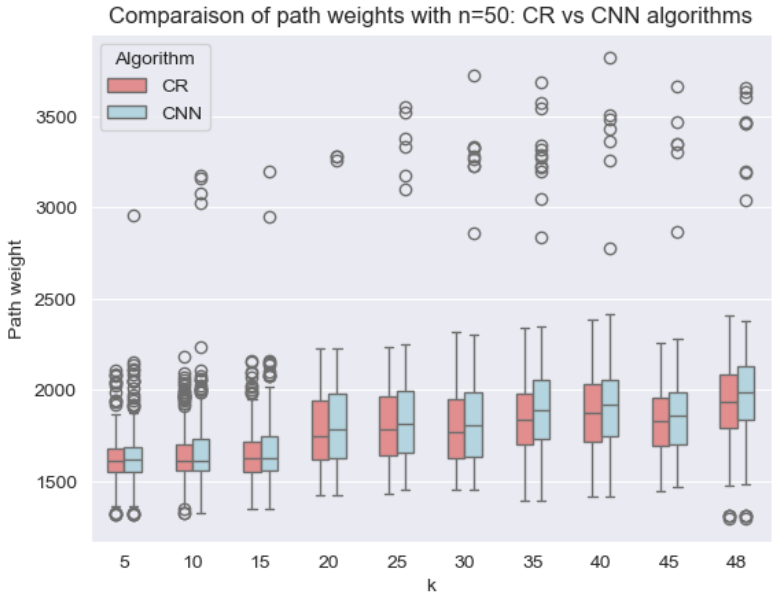


FIGURE 1 – Poids des chemins pour $n = 50$ lorsque k varie.

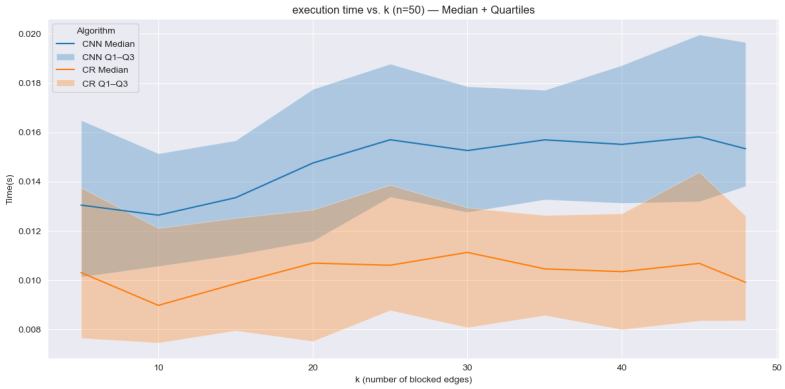


FIGURE 2 – Temps d'exécution pour $n = 50$ lorsque k varie.

Les figures 1 et 2 illustrent respectivement la variation des poids de chemins et des temps d'exécution en fonction du nombre d'arêtes bloquées k, pour un graphe de taille fixe $n = 50$.

Lorsque k est faible, la différence entre les algorithmes `CR` et `CNN` est petite, car les chemins optimaux restent facilement accessibles. En revanche, lorsque k augmente, `CR` tend à fournir des

chemins de meilleure qualité, tandis que `CNN` présente une variabilité plus élevée et tombe plus souvent sur des optima locaux.

En ce qui concerne les temps d'exécution, `CNN` est plus lent que `CR`, mais aucune tendance claire ne se dégage en fonction de k. Cela indique que le temps de calcul est davantage influencé par la structure particulière de chaque instance que par la quantité d'arêtes bloquées.

4.2 Influence de n pour un blocage "maximal" (k = n-2)

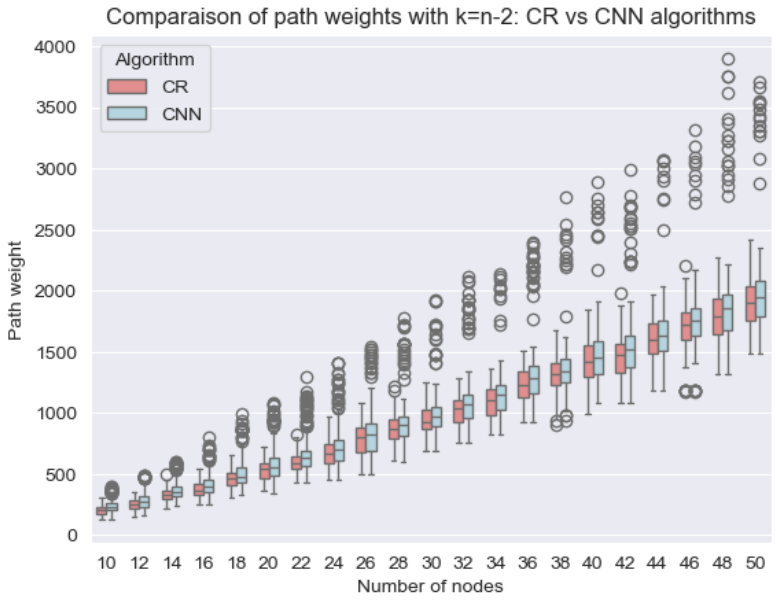


FIGURE 3 – Poids des chemins pour $k = n-2$ lorsque n varie.

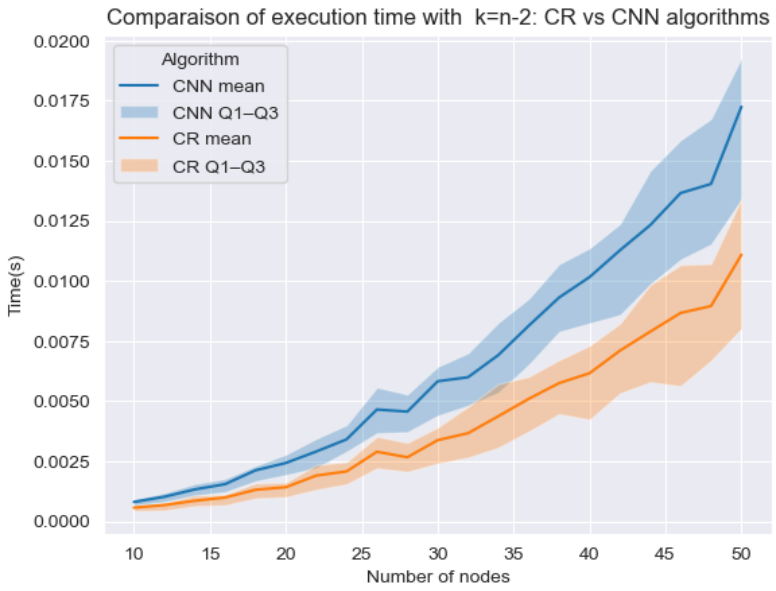


FIGURE 4 – Temps d'exécution pour $k = n-2$ lorsque n varie.

Les figures 3 et 4 présentent respectivement l'évolution des poids de chemins et des temps d'exécution lorsque la taille du graphe n varie, avec un nombre d'arêtes bloquées fixé à $k = n - 2$.

Les résultats confirment les tendances observées dans la section 4.1. L'algorithme `CR` produit des

solutions de meilleure qualité que `CNN`, avec des poids de chemins plus faibles et une dispersion moindre. En revanche, `CNN` montre une variabilité plus importante et génère davantage de valeurs aberrantes, notamment pour les grands graphes.

4.3 Scénario de blocage très élevé ($k = 0.2\,n^2$)

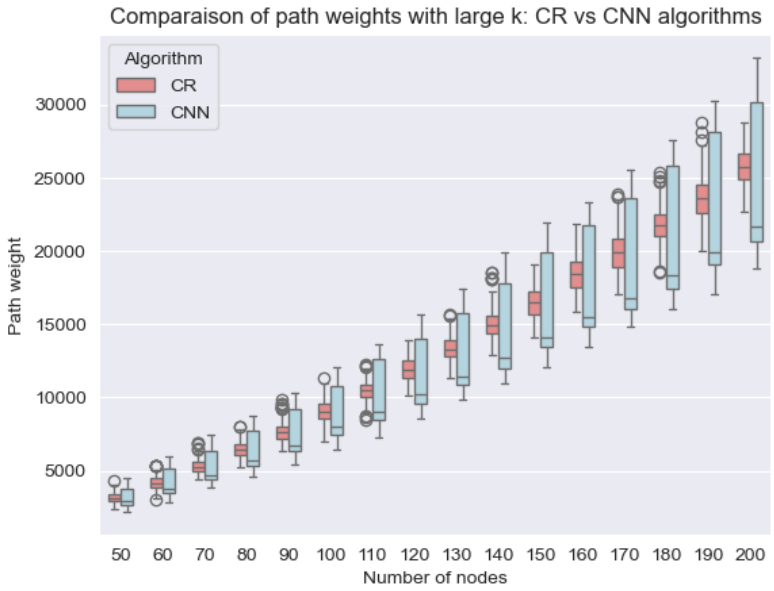


FIGURE 5 – Poids des chemins pour un blocage élevé ($k = 0{,}2\,n^2$).

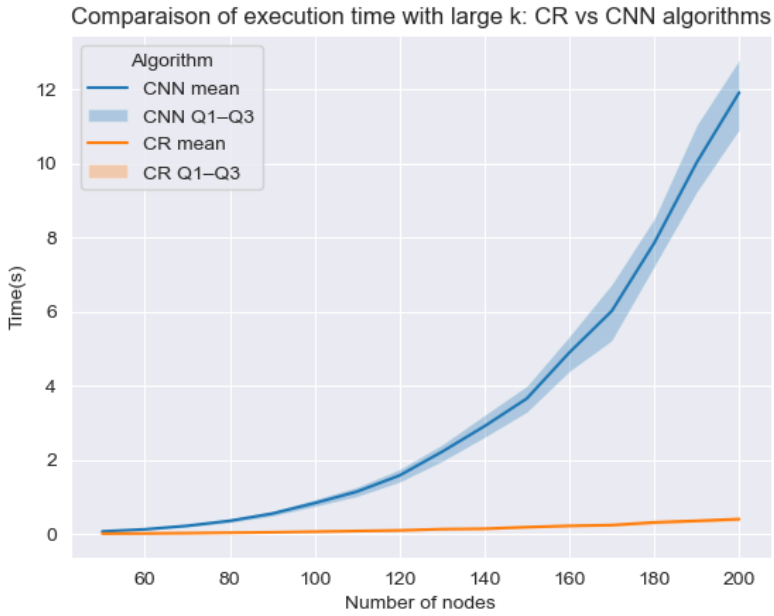


FIGURE 6 – Temps d'exécution pour un blocage élevé ($k = 0{,}2\,n^2$).

Ce troisième scénario, bien que hors du cadre strict du sujet (où $k < n-1$), vise à explorer le comportement des algorithmes lorsque le graphe est fortement obstrué, avec $k = 0{,}2\,n^2$. L'objectif est de voir si `CNN`, généralement moins performant dans les expériences précédentes, peut tirer avantage d'un environnement extrêmement contraint.

Les figures 5 et 6 présentent respectivement les poids des chemins et les temps d'exécution pour

n allant jusqu'à 200.

Nous observons que `CR` reste très stable en termes de qualité de solution, avec une faible dispersion, même lorsque n devient très grand. À l'inverse, `CNN` présente des résultats beaucoup plus dispersés : bien que sa médiane soit inférieure à celle de `CR`, indiquant parfois de meilleures solutions, la large étendue entre le premier et le troisième quartile montre une forte instabilité. Cela signifie que `CNN` peut trouver de très bonnes solution mais aussi échouer selon l'instance.

Concernant le temps d'exécution, le coût de `CNN` augmente de manière beaucoup plus rapide que celui de `CR` avec n.

5 Conclusion

Dans l'ensemble des expériences menées, l'algorithme `CR` se montre globalement plus performant que `CNN`, tant en termes de stabilité que de qualité moyenne des solutions. Il est également plus rapide, en particulier lorsque la taille du graphe augmente.

Cependant, dans des scénarios de blocage extrême, `CNN` peut parfois produire des chemins plus courts que `CR`, bien que de façon instable. Si la contrainte de temps de calcul n'est pas prioritaire, il peut être judicieux d'exécuter `CNN` plusieurs fois sur la même instance pour espérer obtenir une solution de meilleure qualité.