Sorbonne Université
Master ANDROIDE

Compte-Rendu projet MAOA

MAOA - M2

Natacha Riviere – Nicolas loayza

2025

Table des matières

Chapitre 1

Introduction

Le but de ce projet est d'implémenter des nouvelles heuristiques pour la résolution du Travelling Thief Problem (TTP) une combinaison de deux problèmes célèbres étant le Travelling Salesman Problem avec le Knapsack problem. Dans ce problème, nous cherchons à optimiser le temps de trajet du voyageur tout en maximisant le profit du sac à dos. La vitesse du voyageur de commerce dépend du poids du sac à dos, plus le sac est plein, moins le voyageur va vite.

Nous allons étudier deux fonctions objectives différentes, une linéaire et une non linéaire et proposer des heuristiques gloutonnes ainsi qu'itératives pour ces deux problèmes différents.

Données d'entrée/contraintes du problème :

- Un ensemble de n villes $N = \{1..n\}$
  - Avec des distances d_{ij} entre chaque paire (i, j) de villes
- Un ensemble de m_i objets pour chaque ville : $M_i = \{1, ..., m_i\}$ où chaque objet k dans la ville i est caractérisé par :
  - Un poids w_{ik} et sa valeurs p_{ik}
- Un sac à dos avec une capacité maximale W
  - Sous la contrainte de sac à dos : $\sum_{i=1}^{n} \sum_{k=1}^{m_i} w_{ik} \leq W$
- Un coût de location R ou K du sac à dos (pour chaque unité de temps)
- Une vitesse de déplacement variable $v(w)$ qui dépend du poids w transporté
  - La vitesse diminue quand le poids augmente
  - La vitesse maximale est v_{max} et la vitesse min est v_{min}
- Le parcours du voleur : tournée $\Pi = (x_1, ..., x_n)$
- Les objets à voler à chaque ville : plan de collecte $P = (y_{21}, ..., y_{2m_2}, ..., y_{nm_n})$ avec $y_{ik} = 1$ si et seulement si l'objet k à la ville i est sélectionné

Chapitre 2

Fonction objective linéaire

Cette fonction objective est défini de la façon suivante :

$$\sum_{i=1}^{n}\sum_{k=1}^{m_i} p_{ik}y_{ik} - K\left(d_{x_n x_1}W_{x_n} + \sum_{i=1}^{n-1} d_{x_i x_{1+1}}W_{x_i}\right)$$

Comme cette fonction ne considère pas la vitesse (v_{min}, v_{max}) les heuristiques ne seront pas les mêmes, notamment, tant que le sac à dos est vide, le trajet ne compte pas, car si W_{x_i} vaut zéro, alors le terme correspondant de la somme vaut zéro.

2.1 Solution Gloutonne

Notre idée pour cette solution est d'ordonner tous les objets en fonction de leur ratio $profit/poids$ et en fonction de leur distance à l'origine. Plus un objet est loin de l'origine, moins il sera intéressant car il impactera négativement une plus longue distance.
Cela nous donne une liste des meilleurs objets à prendre qui nous pénaliseront le moins. Une fois que nous avons choisi les objets à voler (remplir le sac-à-dos) nous mettons les villes contenant ces objets à la fin du tour et nous construisons le tour de "proche en proche".

Entrées *La liste d'objets l avec leur valeur v et poids p, la liste des villes c avec leur coordonnées et leurs objets*;
Résultat : Une liste s d'objets à voler, et le tour t à faire
Calculer le ratio $r_i = \frac{v_i}{p_i}$ pour chaque objet i ;
Calculer le ratio $r'_i = \frac{r_i}{d_i}$ avec d_i étant la distance à la ville de départ de l'objet i ;
Trier tous les objets en ordre décroissant de leur valeur r'_i;
Prendre dans l'ordre tous les objets possibles qui rentrent dans le sac et les mettre dans une liste s;
$i = 0$;
while *c is not empty* **do**
    $tmp = closest\_city(c[i])$;
    $t = t + tmp$;
    $i = i + 1$;
end
return t,s ;

Algorithme 1 : Algorithme glouton objectif linéaire

Avec cette fonction objectif, on peut remarquer que ne prendre aucun objet nous donne une valeur de zéro. Avec cette heuristique gloutonne, sur les instances fournies, nous trouvons toujours une valeur négative. Nous pouvons donc estimer que cette heuristique n'est pas très bonne car il suffit de ne pas prendre d'objets pour optimiser la solution. Nous allons tenter d'améliorer ce résultat avec l'heuristique itérative.

Nous remarquons aussi que la distance des villes influe beaucoup sur la valuation finale, même en mettant toutes les villes où on a volé à la fin, si les distances sont grandes et les valeurs des objets comparativement petites, nous avons un résultat final qui est très négatif même si ont prends les "meilleurs" objets.

2.2 Solution Itérative

Pour la version itérative nous partons de notre solution initiale gloutonne et nous essayons de l'améliorer en rajoutant de l'aléatoire.

Comme la distance entre les villes est l'élément qui influe le plus notre fonction objectif, parfois il est mieux de pas prendre un objet qui serait en théorie meilleur (d'après notre ordre), et le remplacer par un autre qui permettrait de réduire la distance parcourue avec un sac chargé. On donne donc une probabilité de prendre ou pas un objet. Cela change les objets volés et donc modifie le tour comme nous gardons les villes où on a pris des objets pour la fin.

La construction du tour à partir du nouveau sac à dos part de la solution précédente. Nous enlevons du tour les villes où se trouvent les objets sélectionnés, construisons un tour de proche en proche sur ces villes, puis ajoutons ce chemin à la fin du tour. En faisant cela, seule une petite partie du tour est recalculée. Comme seules les distances parcourues chargées comptent, nous n'avons pas besoin d'optimiser le reste du tour.

Pour s'assurer que les algos ne tournent pas pendant des heures pour trouver la meilleur solution, nous introduisons une limite de temps MAX_TEMPS (2 minutes) ainsi qu'une limite d'itérations sans amélioration $MAX\_ITERATIONS$ (10_000), l'algorithme ne tourne que quand ces deux conditions sont respectés.

Entrées *La liste d'objets l avec leur valeur v et poids p, la liste des villes c avec leur coordonnées et leurs objets*;
Résultat : Une liste s d'objets à voler, et le tour t à faire
$t, s = Algo\_Glouton\_Lineaire(l, v, p, c)$;
$i = 0$;
$temps\_init = Time.time()$;
while $((Time.time() - temps\_init) < MAX\_TEMPS)$ *and* $(i < MAX\_ITERATIONS)$ **do**
    $s' = Randomiser\_Sac(s)$;
    $t' = Creer\_Tour(s)$;
    $val = valuation\_lineaire(s, t)$;
    $val' = valuation\_lineaire(s', t')$;
    if $val' > val$ **then**
        $val = val'$;
        $s = s'$;
        $t = t'$;
        $i = 0$;
    end
    $i = i + 1$;
end
return t,s ;

Algorithme 2 : Algorithme itératif objectif linéaire

Résultats

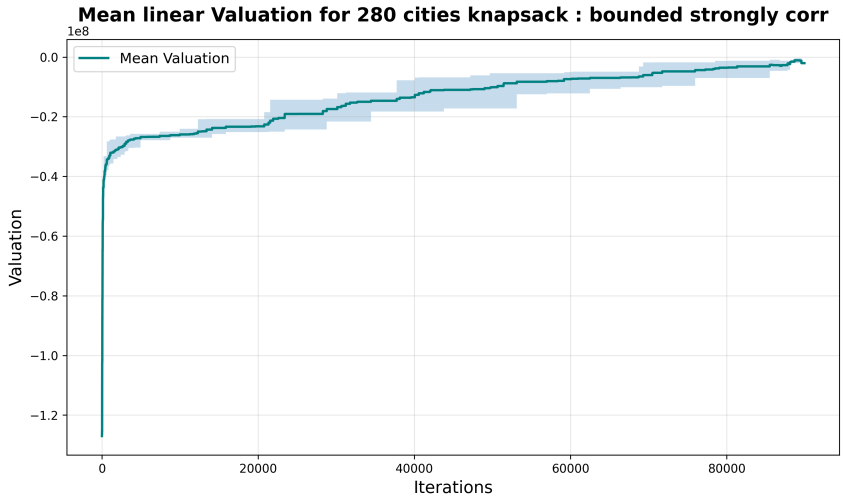


FIGURE 2.1 – Valeur moyenne de la fonction objective linéaire sur l'instance a280_n279_bounded-strongly-corr_01

Sur la courbe précédente, nous pouvons constater que notre algorithme améliore très vite la solution initiale, puis continue à améliorer progressivement la solution. Nous finissons par trouver des résultats positifs pour cette instance, donc meilleurs que ne prendre aucun objet.

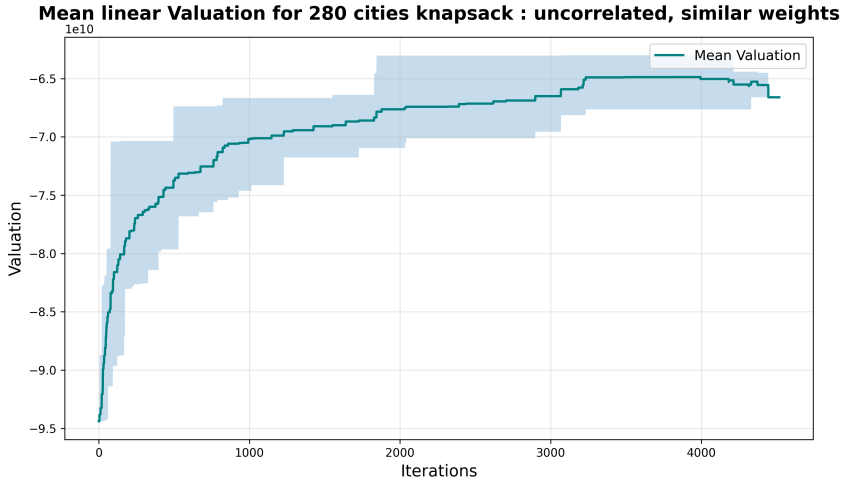


FIGURE 2.2 – Valeur moyenne de la fonction objective linéaire sur l'instance a280_n279_uncorr-similar-weights_05

Lorsque les poids et les valeurs ne sont pas corrélés, nous remarquons davantage de variance sur nos valeurs, ce qui signifie que le sac à dos varie beaucoup en fonction de la stochasticité. Malgré cela, les valeurs grandissent tout de même tout du long. Sur notre courbe, nous observons une baisse à la fin, C'est une erreur sur la représentation. En réalité, c'est lié à la limité de temps. Nos 10 instance n'ont pas fait autant d'itérations et nous avons donc moins que 10 valeurs à moyenner pour les dernière itérations. Comme ici ce sont les moins bonnes valuations qui ont fait plus d'itérations, cela donne l'impression que nous avons une valeur qui diminue.

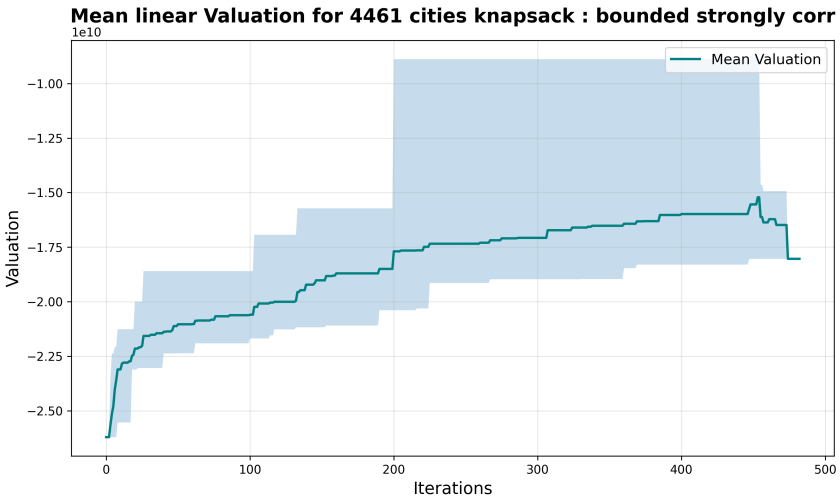
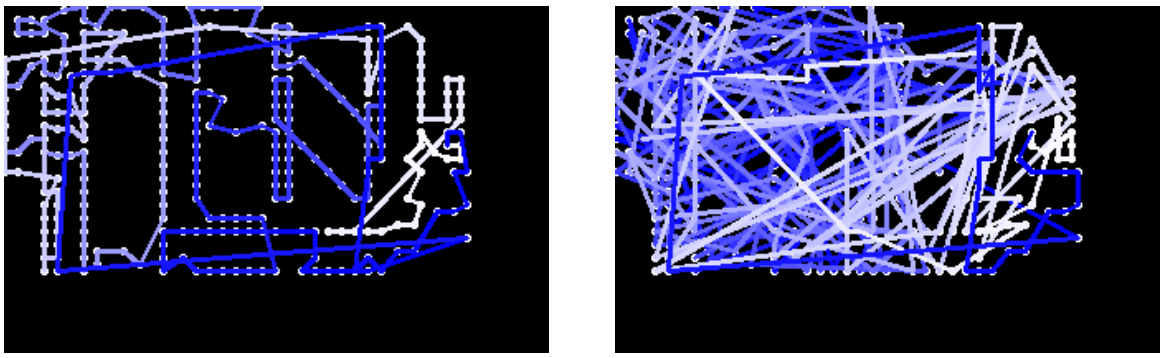


FIGURE 2.3 – Valeur moyenne de la fonction objective non linéaire sur l'instance fnl4461_n4460_bounded-strongly-corr_01_linear

Sur une instance de grande taille (4461 villes) notre heuristique améliore toujours la solution initiale, mais fait 10 fois moins d'itérations. On observe une très grande variabilité, due à la stochasticité du programme.



(a) solution gloutonne (b) après optimisation

FIGURE 2.4 – Représentation du chemin avant et après optimisation

Cette figure présente le chemin pris sur une instance de 280 villes avant et après optimisation de la solution gloutonne. Le résultat peut paraitre étonnant mais comme nous n'essayons pas d'optimiser le chemin entre deux villes si le poids est nul, nous optenons un chemin évidemment sous optimal. En revanche, la portion de chemin sur laquelle le voyageur est chargée, est optimisée. Ici ce sont les villes en bas à droite de l'image.

Chapitre 3

Fonction objective non linéaire

La fonction objective non linéaire est similaire à la précédente mais diffère dans le membre de droite, elle est défini de la façon suivante :

$$\sum_{i=1}^{n}\sum_{k=1}^{m_i} p_{ik}y_{ik} - R * \left(\frac{d_{x_n x_1}}{v_{max} - vW_{x_n}} + \sum_{i=1}^{n-1} \frac{d_{x_i x_{i+1}}}{v_{max} - vW_{x_i}} \right)$$

Cette fonction objective prends en compte la vitesse du voleur lorsqu'il est chargé, donc un objet influe tout le trajet une fois pris. Comme le membre de gauche reste pareil pour les deux fonctions, prendre de bons objets reste pertinent, mais maintenant on doit être plus judicieux sur le parcours. En effet, contrairement à la première fonction, une part de chemin parcourue avec un sac à dos vide a tout de même un impact négatif sur la valeur objectif. En revanche, l'impact du poids est moins important (car il impacte la vitesse, en dénominateur, et non directement la valeur).

3.1 Solution Gloutonne

Notre approche gloutonne reste similaire à celle d'avant. On cherche a optimiser le sac-à-dos en premier pour en fixer le tour juste après. Cette fois-ci au lieu de calculer le ratio par rapport au poids, valeur et distance, on le calcule que par rapport au poids et la valeur. Comme avant, une fois les objets choisis on met les villes où on a trouvé ces items à la fin du parcours. Ici on cherche davantage à optimiser le sac à dos, pour maximiser le pofit plutôt que le tour.

Entrées *La liste d'objets l avec leur valeur v et poids p, la liste des villes c avec leur coordonnées et leur objets*;
Résultat : Une liste s d'objets à voler, et le tour t à faire
Calculer le ratio $r_i = \frac{v_i}{p_i}$ pour chaque objet i ;
Trier tous les objets en ordre décroissant de leur valeur r_i;
Prendre tous les objets possibles qui rentrent dans le sac et les mettre dans une liste s;
$cities\_without\_object$ = [city for city in c if there is no object taken in city];
$cities\_with\_objects = c - cities\_without\_objects$;
$t = [first\_city]$;
while *cities_without_objects is not empty* **do**
    $tmp = closest\_city(t[-1], cities\_without\_objects)$;
    $t+ = tmp$;
    $cities\_without\_objects.remove(tmp)$;
end
while *cities_with_objects is not empty* **do**
    $tmp = closest\_city(t[-1], cities\_with\_objects)$;
    $t = t + tmp$;
    $cities\_with\_objects.remove(tmp)$;
end
return t,s ;

Algorithme 3 : Algorithme glouton objectif non linéaire

Cet algorithme, à nouveau n'est pas très performant, il a l'avantage bien sûr d'être rapide même s'il ne donne pas les meilleurs résultats. Cependant il est suffisamment utile pour servir de base à notre version itérative.

3.2 Solution Itérative

Pour notre solution itérative, nous avons exploré plusieurs pistes pour améliorer notre solution gloutonne, au début une idée était de chercher un meilleur tour avec un sac à dos donné en échangeant des villes de façon aléatoire dans le tour pour voir si cela réduisait la longueur totale. Cependant cette approche n'a pas donné de résultats satisfaisants.
Nous avons donc essayé une nouvelle approche, celle de prendre l'arête la plus longue du tour et de la modifier, cette méthode donnait de meilleurs résultats que la version aléatoire mais ne permettait pas d'améliorer la solution gloutonne.
Enfin, nous avons décidé de chercher un algorithme qui permettrait d'améliorer le tour de façon fiable, et après un peu de recherche nous avons décidé d'implémenter et d'utiliser l'algorithme 2-OPT pour améliorer la longueur totale du tour.

Nous nous sommes donc basés sur l'algorithme 2-opt pour élaborer notre heuristique. L'algorithme 2-OPT est un algorithme de recherche locale pour résoudre le Travelling Salesman Problem, le principe de ce algorithme est de prendre deux arêtes, et voir si en "croisant" les deux arêtes nous pouvons avoir une meilleure valuation. En pratique, l'algorithme échange toutes les arêtes deux à deux et garde la modification si elle améliore la longueur du tour.

Pour notre algorithme, nous commençons par calculer une première solution avec notre heuristique gloutonne, puis nous l'améliorons. A chaque étape, l'algorithme utilise 2-opt pour trouver un nouveau tour plus court, et calcule un nouveau sac à dos pour ce tour. Si la valuation du nouveau tour avec la nouvelle valuation est meilleure, alors la modification est gardée.
Pour calculer un nouveau sac à dos, nous classons les objets par ratio ($profit/poids$) multiplié par l'indice de la ville où se trouve l'objet dans le tour, et prenons les objets dans cet ordre tant qu'ils rentrent dans le sac à dos.
Si une modification améliore le tour mais pas la valeur de la fonction objective, elle n'est pas gardée. De plus, nous introduisons des probabilités de prendre un objet qui grandissent au fur et à mesure des itérations (la probabilité de choisir de prendre un objet diminue quand la valuation n'a pas changé à l'itération précédente).

Comme pour la fonction objective précédente, nous limitons le temps de calcul de l'algorithme, ainsi que les nombre d'itérations maximales sans amélioration avec les variables globales MAX_TEMPS (fixé à 2 minutes) et $MAX\_ITERATIONS$ (10000 itérations), de cette façon même si l'instance est grande on s'assure que nous aurons une solution.

Entrées *La liste d'objets l avec leur valeur v et poids p, la liste des villes c avec leur coordonnées*;
Résultat : Une liste s d'objets à voler, et le tour t à faire
$t, s = Algo\_Glouton\_Non\_Lineaire(l, v, p, c)$;
$i = 0$;
$temps\_init = Time.time()$;
while $((Time.time() - temps\_init) < MAX\_TEMPS)$ *and* $(i < MAX\_ITERATIONS)$ **do**
    $t' = 2opt(t)$;
    if $length(t') < length(t)$ **then**
        $s' = knapsack(s)$;
    end
    $val = valuation\_non\_lineaire(s, t)$;
    $val' = valuation\_non\_lineaire(s', t')$;
    if $val' > val$ **then**
        $val = val'$;
        $s = s'$;
        $t = t'$;
        $i = 0$;
        continue;
    end
    $i = i + 1$;
end
return t,s ;

Algorithme 4 : Algorithme itératif objectif non linéaire

Résultats

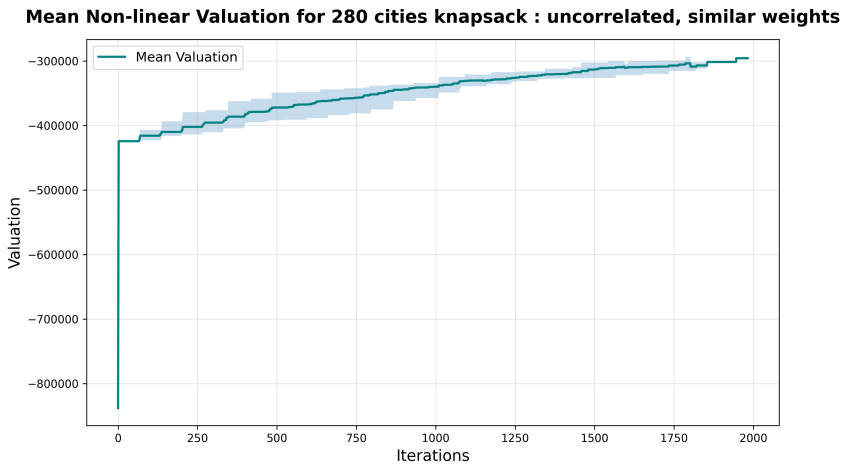


FIGURE 3.1 – Valeur moyenne de la fonction objective non linéaire sur l'instance a280_n1395_uncorr-similar-weights_05

Notre heuristique permet une nette amélioration de la solution gloutonne initiale, en moins de 10 itérations, puis, sur les instances testées, nous obtenons une amélioration continue de la valeur objective sur le temps.
Sur le graphe ci-dessus nous observons une augmentation progressive de la valeur dans le temps. Comme notre heuristique est stochastique, nous avons utilisé 10 run pour tracer cette courbe. Nous pouvons voir que l'aléatoire permet d'obtenir parfois de meilleures valeurs plus rapidement. La fin de la courbe ne présente plus de variance car comme nos runs étaient limitées en temps, nous avons tout simplemetnt

moins de valeurs sur ces itérations. Nous n'avons donc plus de vraie moyenne à la fin, ce sont seulement les valeurs d'une seule run.

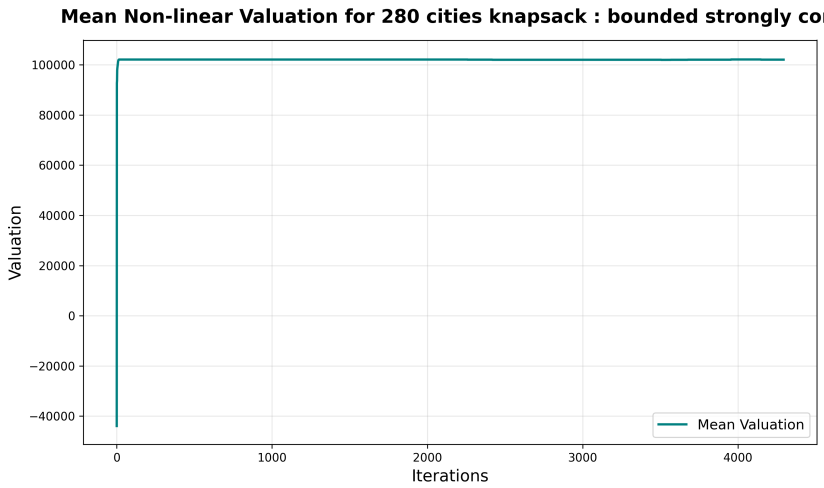


FIGURE 3.2 – Valeur moyenne de la fonction objective non linéaire sur l'instance a280_n279_bounded-strongly-corr_01

Sur cette figure suivante, nous observons que lorsque la valeur d'un objet dans le sac à dos est fortement corrélé à son poids, nous stagnons après le pic initial d'amélioration. Même avec de la stocasticité, cela suggère que le sac à dos reste toujours le même, car il n'y a aucune variance. Notre heuristique permet donc de trouver un maximum local, mais pas d'en sortir.

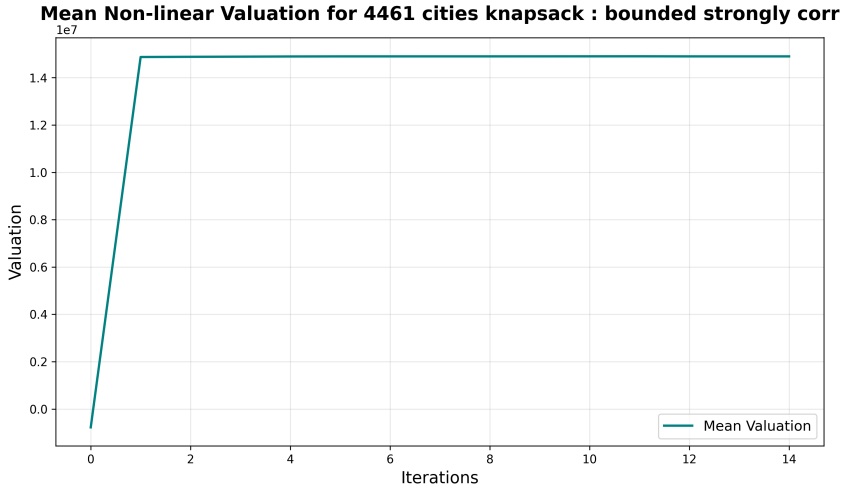
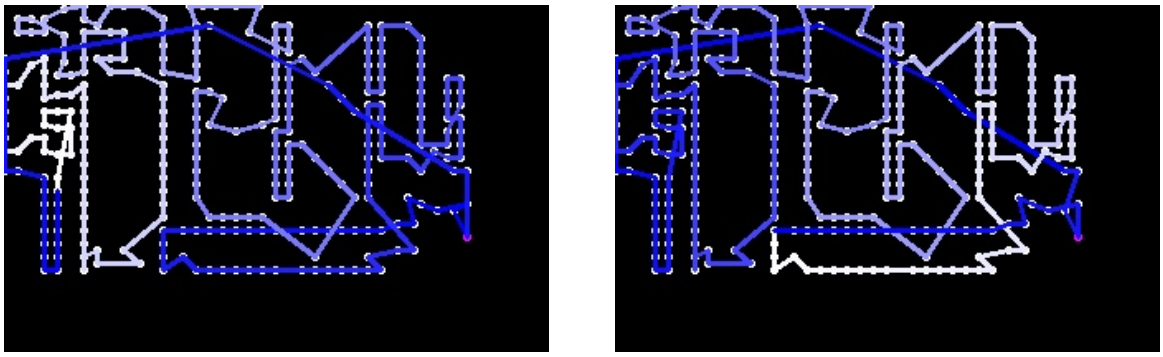


FIGURE 3.3 – Valeur moyenne de la fonction objective non linéaire sur l'instance fnl4461_n4460_bounded-strongly-corr_01

Sur des instances de grande taille (ici 4461 villes) notre heuristique est beaucoup plus lente. Nous avons ici 14 itérations en 2 minutes, contre environ 5000 pour l'instance précédente. En si peu d'itérations, la stochasticité, qui est introduite progressivement lorsque le programme est bloqué sur une valuation



(a) solution gloutonne (b) après optimisation

FIGURE 3.4 – Représentation du chemin avant et après optimisation

Sur cette figure, nous pouvons voir le chemin avant et après optimisation de la solution initiale. Contrairement à la version linéaire, tout le chemin doit être optimisé donc nous tendons vers un chemin le plus court possible. Ici nous pouvons constater notamment que l'optimisation, a retourné le sens du chemin, car celà devait être plus intéressant pour le contenu du sac à dos

Chapitre 4

Formulation PLNE du Problème

Variables de décision

- $x_{ij} \in \{0, 1\}$: Indique si l'arc (i, j) est emprunté (1 si oui, 0 sinon).
- $y_k \in \{0, 1\}$: Indique si l'objet k est collecté (1 si oui, 0 sinon).
- $W_x(i) \geq 0$: Poids total transporté à la ville i.
- $u_i \geq 1$: Ordre de visite de la ville i (utilisé pour éliminer les sous-tours).
- $z_{ij} \geq 0$: Poids transporté sur l'arc (i, j).

Données

- n : Nombre total de villes.
- m : Nombre total d'objets.
- d_{ij} : Distance entre les villes i et j.
- p_k : Profit associé à l'objet k.
- w_k : Poids de l'objet k.
- r : Ratio de location (coût par unité de poids par unité de distance).
- c : Capacité maximale du sac à dos.

Fonction objectif

Maximiser :

$$\text{Profit total} - \text{Coût total de trajet} = \sum_{k \in \mathcal{K}} p_k y_k - r \sum_{(i,j) \in \mathcal{A}} z_{ij} d_{ij}$$

Contraintes

1. Contrainte de départ (poids initial)

$$W_x(1) = 0$$

2. Conservation du poids à chaque ville Pour toute ville $i \neq 1$:

$$W_x(i) = \sum_{j \neq i} z_{ji} + \sum_{k \in \mathcal{K}(i)} w_k y_k$$

avec $\mathcal{K}(i)$ l'ensemble des objets situés à la ville i.

3. Contrainte de capacité du sac à dos

$$\sum_{k \in \mathcal{K}} w_k y_k \leq c$$

4. Relation entre z_{ij}, $W_x(i)$, et x_{ij} Pour tout arc (i, j) :

$$z_{ij} \leq c \cdot x_{ij}$$

$$z_{ij} \leq W_x(i)$$

$$z_{ij} \geq W_x(i) - c \cdot (1 - x_{ij})$$

$$z_{ij} \geq 0$$

5. Contraintes de visite des villes (TSP) Pour toute ville i :

$$\sum_{j \neq i} x_{ij} = 1 \quad \text{(partir de } i\text{)}$$

$$\sum_{j \neq i} x_{ji} = 1 \quad \text{(arriver à } i\text{)}$$

6. Élimination des sous-tours (MTZ) Pour toute paire de villes $i, j \neq 1$:

$$u_i - u_j + 1 \leq n \cdot (1 - x_{ij})$$

$$u_1 = 1 \quad \text{(ordre de visite de la ville de départ)}$$

$$2 \leq u_i \leq n \quad \text{pour toute ville } i \neq 1$$

Chapitre 5

Conclusion

Le Travelling Thief Problem étant une combinaison de deux problèmes NP-difficiles, ne permet pas l'utilisation d'algorithmes mal optimisés sans sacrifier en temps de calcul ou en qualité de la solution. Dans cette optique nos heuristiques gloutonnes sont très rapides indépendamment de la taille de l'instance en question, mais donnent des résultats moins que convenables, nous obligeant de faire des compromis surtout dans le tour à effectuer. Cependant elle permettent d'avoir une solution convenable sur laquelle itérer et améliorer nos solutions sur les autres heuristiques, qui sont bien meilleures même sur des instances un peu plus grandes.

Nos solutions restent moins que parfaites mais au moins nous avons pu améliorer nettement nos solutions grâce aux heuristiques itératives. Nos solutions restent probablement bloquées dans un maximum local car s'approcher de la meilleure solution du voyageur de commerce ne ga De plus, devoir optimiser pour deux fonctions objectives similaires mais légèrement différentes à rajouté une couche de difficulté, car les solutions implémentées ne pouvaient pas être exactement la même pour les deux.

Nous n'avons pas eu le temps de comparer nos heuristiques implémentées avec celles codées en Java et présentées dans le papier de référence. Cette limitation est principalement due aux différences de performance entre les langages de programmation. En effet, Java et Python ont des caractéristiques intrinsèques qui influencent leur vitesse d'exécution, notamment en ce qui concerne la gestion de la mémoire et l'optimisation du code au moment de l'exécution. Ces différences rendent difficile une comparaison équitable sans un effort supplémentaire pour aligner les environnements d'exécution ou réimplémenter les heuristiques dans un langage commun. Par conséquent, nous avons choisi de concentrer notre temps sur l'optimisation et l'analyse des performances de nos propres solutions.