Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique parcours ANDROIDE

Rapport de projet

MAOA : Création d'un métro circulaire

Réalisé par :
Lou MOULIN-ROUSSEL Maxence MAIRE

Année 2023-2024

Table des matières

Table des figures

Chapitre 1

Introduction

Dans ce projet, nous approchons le problème de création d'un métro circulaire dans une zone définie. Un "bon" métro devra satisfaire au mieux trois critères :

— Critère de coût : moins un métro coûte cher, meilleur il est (les stations ont un coût fixe, et les tronçons ont un coût au kilomètre)

— Critère de temps : plus le temps moyen pour aller d'un point à un autre est bas, meilleur le métro est

— Critère de ratio : plus le ratio trajet à pied/trajet en métro est bas, meilleur le métro est

Ce rapport décrira en détails la manière dont l'on a approché le projet, les algorithmes utilisés et les résultats obtenus. Nous présenteront les conclusions tirées de ces résultats et les limitations et suites potentielles du projet en dernière partie du rapport.

L'archive Github du projet est en accès libre. Le programme a été développé en Python, en utilisant les librairies citées dans le manuel utilisateur.

Chapitre 2

Contribution

2.1 Algorithme méta-heuristique

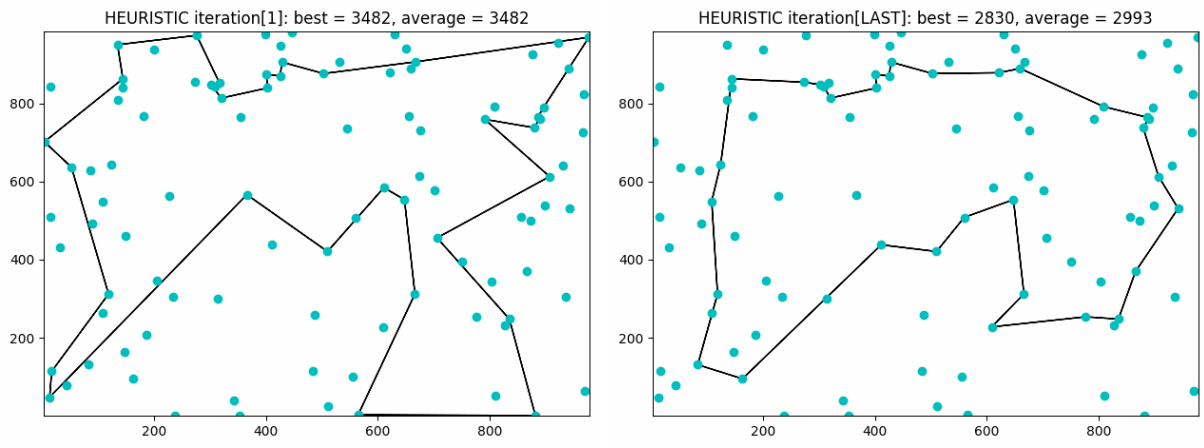

Figure 2.1 – Exemple d'évolution utilisant la méthode méta-heuristique

2.1.1 Approche utilisée

Description générale

En premier lieu, nous avons développé une approche méta-heuristique pour résoudre le problème.
Dans cette approche, on utilise tout d'abord une heuristique randomisée pour définir les stations choisies. Ensuite on utilise un TSP heuristique (approche du recuit simulé).
On utilise ensuite une recherche locale pour améliorer la solution. On modifie une station au hasard en la supprimant et en la remplaçant par un de ses points voisins, puis l'on insère ce point dans le trajet du métro. On analyse les solutions (en $O(n^2)$) et l'on conserve les x meilleure. On itère, en utilisant à chaque fois les x meilleures solutions comme parents pour générer de nouvelles solutions enfants.
Si la meilleure solution cesse de s'améliorer (stagne trop longtemps), on arrête la recherche locale, et l'on fait un dernier TSP heuristique pour améliorer la solution finale.

Avantages et défauts

L'algorithme méta-heuristique a pour avantages d'être rapide, de pouvoir fonctionner sur de larges instances et de pouvoir fournir des résultats assez performants. En revanche, ces résultats ne sont pas garantis, et l'algorithme peut se bloquer sur un maximum local (cf exemple ci-dessous). De plus, sur de grosses instances, on doit réduire les performances de l'algorithme afin de conserver un temps de calcul raisonnable.
À noter que l'on peut également lancer l'algorithme plusieurs fois afin d'essayer d'améliorer encore la solution.
Dans l'exemple ci-dessous, on voit que l'exécution renvoit une solution mauvaise, bloquée par un maximum locale (problème important dans les recherches heuristiques).

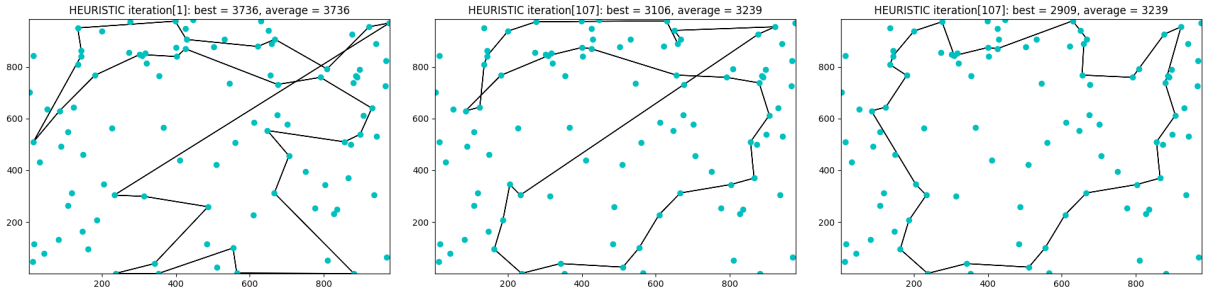

FIGURE 2.2 – Exécution retournant un maximum local avec l'approche méta-heuristique

Ci-dessous, on a une meilleure solution, où l'on est pas bloqué par un maximum local. On voit clairement une amélioration entre la première itération (gauche) et la solution finale retournée (droite).

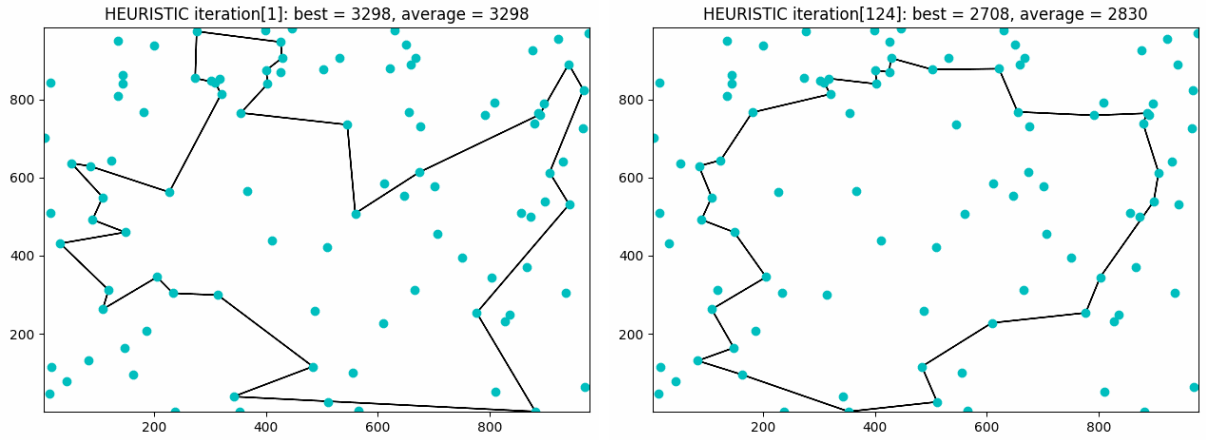

FIGURE 2.3 – Bonne exécution avec l'approche méta-heuristique

2.1.2 Algorithme

Ci-dessous, l'algorithme utilisé pour la recherche heuristique.

Algorithm 1 solveHeuristics(*points*, $numberOfStations$, $stagnationThreshold = 30$, $nbChildren = 7$, $nbParents = 4$, $lastTspIterations = 15$, $saveImg = False$)

Require: $0 \leq power \leq 1024$

stations ← *numberOfStations* STATIONS (HEURISTIQUE RANDOMISÉE RÉPÉTÉE)
solutions ← [SIMULATED ANNEALING TSP(*stations*)]
bestSolution ← *solutions*[0]
bestSolutionAge ← 0
while *bestSolutionAge* < *stagnationThreshold* **do**
    rankedSolutions ← SORT(*solutions*)
    if VALUE(*bestSolution*) < VALUE(*solutions*[0]) **then**
        bestSolution ← *solutions*[0]
        bestSolutionAge ← 0
    else
        bestSolutionAge ← *bestSolutionAge*+1
    end if
    if *saveImg* **then**
        SAUVEGARDE DE LA MEILLEURE SOLUTION EN IMAGE
    end if
    solutions ← []
    for $p \leftarrow 1$ to *nbParents* **do**
        for $c \leftarrow 1$ to *nbChildren* **do**
            solutions ← *solutions*+NEW CHILD SOLUTION(*rankedSolutions*[*p*])
        end for
    end for
end while
bestSolution ← SIMULATED ANNEALING TSP(*bestSolution*)
Return *bestSolution*

Temps de calcul et complexité

Un point important de l'algorithme est qu'il a un temps d'exécution incertain : il s'arrête quand la solution cesse de s'améliorer. On peut donc facilement jouer sur ses paramètres expérimentalement pour augmenter ou diminuer le temps d'exécution, selon la qualité de solution recherchée et la taille de l'instance. Les parties de l'algorithme à complexité fixée ne vont pas au delà de $O(n^2)$.

Utilisation et paramètres

On peut voir, dans l'exemple ci-dessous, que l'algorithme peut malgré tout sortir d'extrema locaux. Avec expérimentation, nous avons conclu que la meilleure combinaison de paramètres pour trouver une bonne solution tout en étant résistant aux extrema locaux et en en gardant un temps d'exécution raisonnable, il faut avoir un nombre elevé de parents (p=32) et un nombre réduit d'enfants par parent (c=4). Ainsi, les enfants (devenant parents) ont le temps de se développer et on a un ensemble de solution plus varié qu'avec un nombre de parents plus réduit (où ils sont majoritairement enfants de la meilleure solution) ou qu'avec un nombre d'enfants plus élevé (où encore une fois, on a peu de variété puisque rapidement des solutions très similaires).

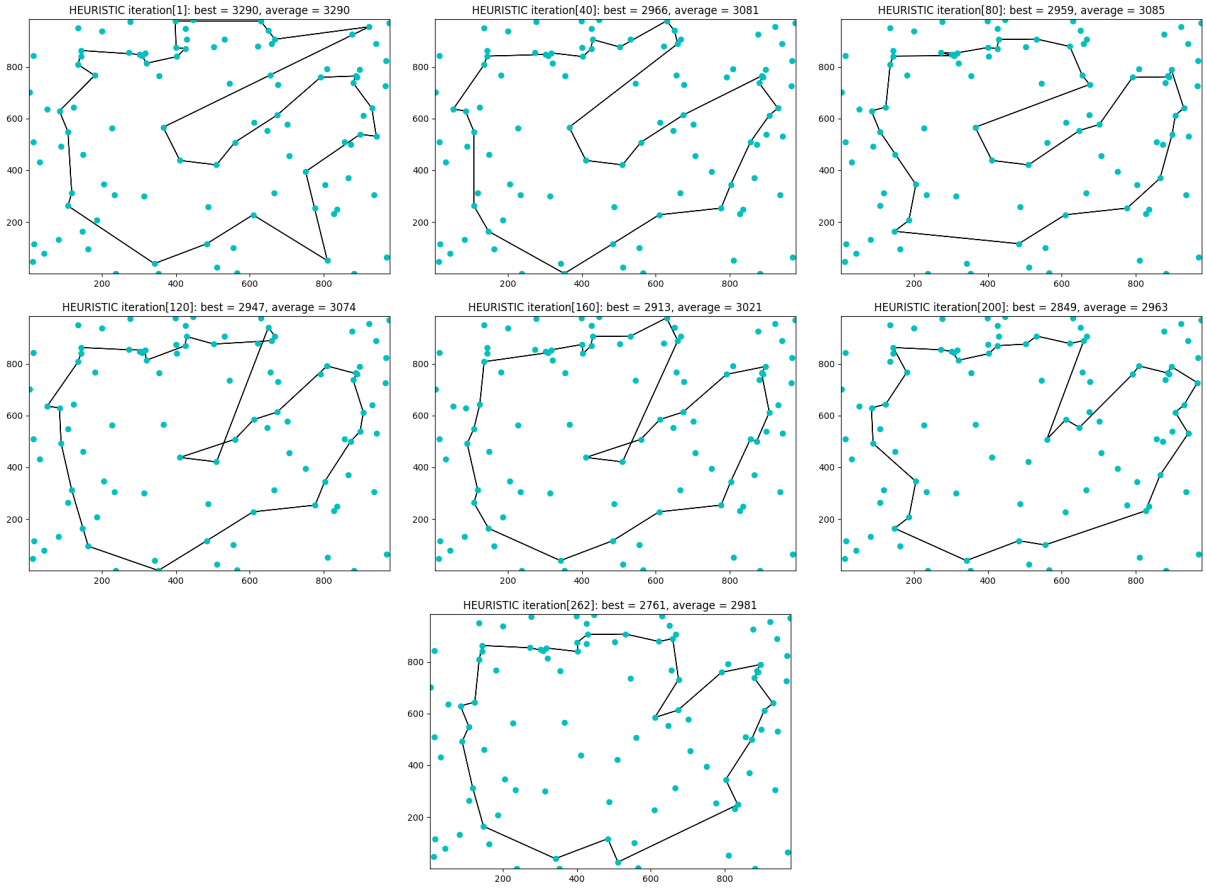

FIGURE 2.4 – Exécution où la solution sort d'un maximum local

Performances

Average execution time : 606.318
Average time : 2450.021
Average ratio : 0.0213
Average cost : 360254.157

2.2 PLNE - Formulation compacte

2.2.1 Approches utilisées

Nous avons tenté d'utiliser plusieurs approches utilisant des formulations compactes avec le solveur *Gurobi*.

Première approche

Nous avons tout d'abord développé une approche en deux étapes pour résoudre le problème : nous résolvons le problème du p-médian en premier lieu à l'aide d'un programme linéaire en formulation compacte puis nous résolvons un TSP avec une formulation compacte également. Cette approche a l'avantage d'être simple et les résultats sont exacts pour les deux sous-problèmes.

Autres approches

Nous nous sommes ensuite demandés si les résultats obtenus en créant des *clusters* puis en choisissant une station par cluster seraient très différents. Nous avons donc expérimenté différentes façons de créer les clusters, tout en utilisant la même technique pour choisir les stations pour chaque cluster dans un deuxième temps : nous résolvons le problème *p-médian* mais avec la contrainte supplémentaire qu'il doit y avoir une et une seule station par cluster puis nous résolvons le TSP correspondant à ces stations.

- Clustering selon le problème du p-médian : on résoud le *p-médian* et on considère que tous les points attribués à un médian forment un cluster autour de ce médian.
- Clustering autour de p points éloignés les uns des autres : on choisit p points le plus éloignés les uns des autres possible puis on attribue les autres points à l'un de ces médians en minimisant la somme des distances aux médians
- Clustering par densité (algorithme DBSCAN) : ici, on ne choisit pas un nombre de clusters mais une distance seuil telle que si deux points sont à une distance inférieure à ce seuil alors ils appartiennent au même cluster. Nous avons testé ces méthodes sur une même instance (*rd100.tsp*) avec $p = 20$ (à gauche, le résultat pour p-médian + TSP, au centre, pour un clustering à partir des points éloignés les uns des autres, et à droite, pour le clustering avec DBSCAN). On obtient 21 clusters avec DBSCAN avec les paramètres utilisés : $epsilon = 80$, $minsamples = 2$.

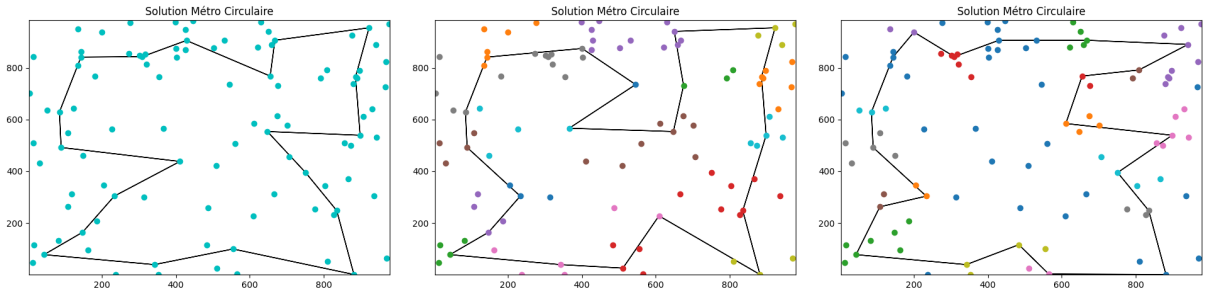

FIGURE 2.5 – Résultats sur la même instance pour différentes méthodes

Le tableau ci-dessous contient les coûts sur les différents critères retenus pour chacune de ces méthodes sur cette instance. Nous remarquons que la première approche est celle qui a obtenu les meilleurs résultats.

Méthode	Temps Moyen	Ratio	Coût
p_médian + TSP	2218.650654	21.99897284	243134.5239
clusters + p-médian + TSP	2388.266502	22.88315189	245640.2679
dbscan + p-médian + TSP	2247.54593	25.77529595	251735.5562

TABLE 2.1 – Résultats expérimentaux

2.3 PLNE - Formulation non compacte

Nous avons tenté de résoudre le problème de l'anneau-étoile avec une formulation non-compacte dans le fichier **anneau_etoile.py** mais nous n'avons pas réussi à implémenter un *callback Lazy* pour la séparation entière fonctionnel. Nous n'avons donc pas obtenu de solution à comparer avec celles obtenues pour les autres méthodes.

Chapitre 3

Conclusion

Algorithme heuristique

Finalement, notre algorithme heuristique a fourni des résultats assez puissants, mais il nous a fallu du temps pour le faire fonctionner correctement. Trouver les bons paramètres d'algorithme a en particulier été un travail ayant requis un certain nombre d'essais avant d'en avoir un ensemble adéquat.
Cependant, avec les paramètres actuels, nous avons un bon équilibre entre résistance aux extrema locaux, temps d'exécution, et qualité de la solution finale.
Nous avons malgré tout relevé quelques points qui pourraient être améliorés avec davantage de temps :

— TSP en recherche locale : une méthode pouvant améliorer les solutions serait de lancer un TSP heuristique sur une solution (par exemples toutes les 20 itérations) afin de casser les noeuds potentiels provenant du fait que les stations sont simplement insérés sur le chemin actuel, sans recalculer ce dernier.

— Adaptation aux instances particulières : notre algorithme s'adapte bien aux instances ayant des villes distribuées aléatoirement, mais moins bien si les villes sont disposées de manière régulière (exemple ci-dessous). Nous pourrions changer l'algorithme pour chercher une meilleure adaptation à ces situations.

— Dépendance du tirage initial : le tirage intial des stations influence beaucoup le résultat de l'exécution. Ceci pourrait être un point d'amélioration.

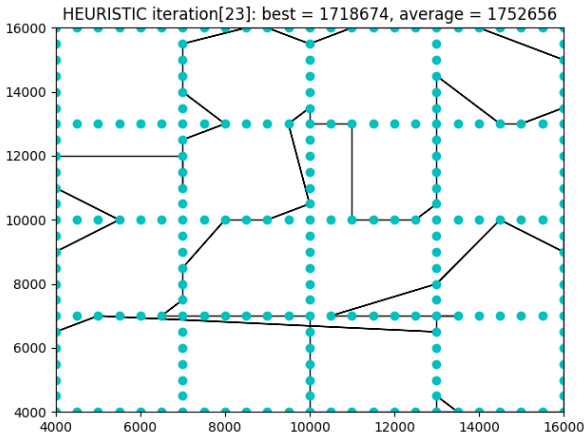

Figure 3.1 – Mauvaise exécution, sur une instance "particulière"

Algorithmes de PLNE

Pour résoudre ce problème, nous l'avons décomposé en plusieurs sous-problèmes que nous résolvons de manière exacte à l'aide du solveur *Gurobi*. Même si les solutions de ces sous-problèmes sont exactes, la solution finale trouvée, elle, ne l'est pas. Cette méthode permet cependant de trouver une ou des solutions se rapprochant de la solution optimale avec une complexité moindre.

Il serait également intéressant de résoudre le problème de l'anneau-étoile pour comparer les résultats avec ceux que nous avons obtenus.

Critères

Nous pourrions poursuivre ce projet en réfléchissant aux critères à optimiser et à comment prendre une décision étant donnés plusieurs tracés possibles.

Chapitre 4

Manuel d'utilisateur

4.0.1 Télécharger l'archive :

Installer depuis Github.

4.0.2 Installer les dépendances :

Python 3.10 ou supérieur. Pour installer une librairie Python :

```
$ pip install numpy
```

Librairies utilisées :

- numpy
- matplotlib
- natsort
- gurobipy
- shutil
- imageio

4.0.3 Exécution du projet :

Se placer dans l'archive du projet, ouvrir un terminal et exécuter la commande

```
$ python main.py
```

Cette commande lance l'exécution du programme (méthode méta-heuristique, PLNE compacte et PLNE non compacte), ainsi que la sortie visuelle.