GROUPE INP La Prépa

Promo P31 - 2ème année
Année Universitaire 2024-2025

Compte-rendu du problème 1

Romain BOUILLON
Noreddine BELMEGUENAÏ
Rémy GAUDIN
Gaël BOTET
Adrien CANDELA

Mercredi 16 Avril 2025

Table des matières

1 Introduction

Le problème 1 nous demande de trouver l'itinéraire le plus court possible que le jardinier d'un parc de la ville doit prendre pour visiter tous les arbres du parc, en partant de l'entrée (représentée comme étant l'origine du repère de coordonnées $(0, 0)$) en en y revenant à la fin. Par chance, le jardinier (nous) dispose des coordonnées de chacun des arbres du parc. L'objectif du problème est de découvrir et d'implémenter de nouveaux algorithmes permettant de se rapprocher d'une solution optimale tout en gardant des temps d'exécutions de programme viables.

2 Résolution du problème

On a cherché chacun de notre côté une solution au problème, en partageant nos idées et en s'organisant pour éviter d'être trop nombreux sur une même idée.

Au final, l'idée générale est d'approcher une solution sous optimale puis de l'améliorer petit à petit grâce à des heuristiques locales.

2.1 Première approche de la solution optimale

On a commencé par la méthode exhaustive, qui teste toutes les possibilités et renvoie la meilleure, qui prend beaucoup de temps à s'exécuter pour des jardins avec plus de 10 arbres bien qu'elle renvoie toujours la solution optimale.

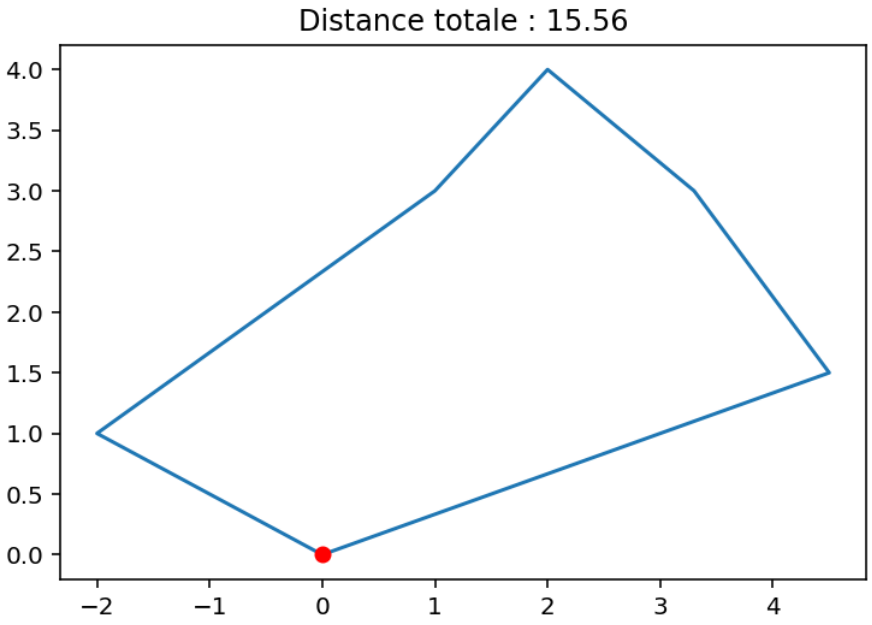

Figure 2.1 – Exemple 1 avec la méthode exhaustive

Ensuite la méthode gloutonne, qui prend toujours l'arbre le plus proche. Elle s'exécute rapidement et converge toujours, cependant elle est rarement optimale. On se sert alors de différentes heuristiques locales que l'on présentera plus tard.

La solution suivante est d'optimiser la méthode gloutonne en la pondérant. Pour choisir le point suivant on regarde le nombre de points qui l'entoure dans un rayon donné en paramètre. On multiplie ensuite sa distance au point actuel par le poids obtenu (le poids étant le nombre de points autour de lui multiplié par un facteur). On prend au départ un point aléatoire dans le graphe, et on remarque que le point de départ faire varier le résultat. On itère alors N fois pour avoir le meilleur point de départ (qui est arbitraire car c'est un cycle).

On notera tout de même que le programme d'origine nous faisait toujours partir du point de coordonnées (0 ,0), donc on l'ajoute dans notre set de coordonnées au début du programme.

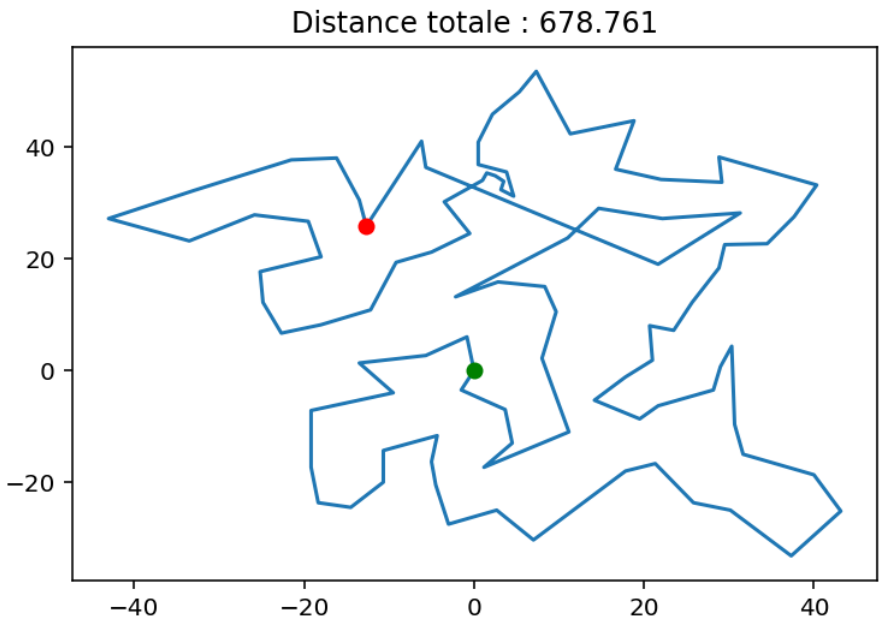

Figure 2.2 – Exemple 2 avec la méthode gloutonne pondérée, le point de départ du programme est en rouge et le point (0, 0) est en vert

Notre idée fut d'utiliser la très fameuse dichotomie, en découpant notre plan 2d en 2, puis en 2, puis en 2, ... On commence par trier la liste de coordonnées selon les x croissants, puis on la coupe en 2 pour répéter l'opération. Pour éviter d'obtenir un graphe horrible tout en vertical comme ceci :

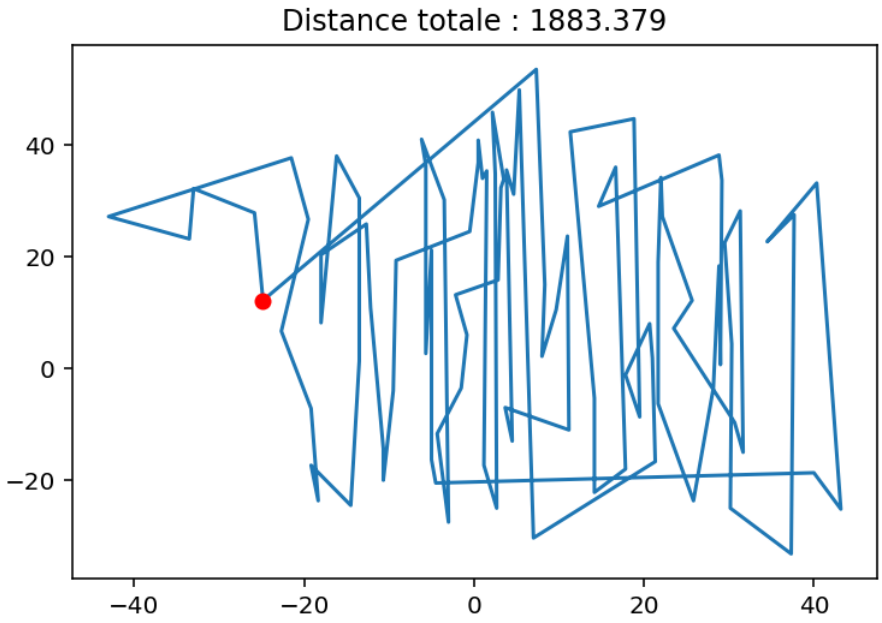

Figure 2.3 – Dichotomie que selon l'axe x

On change d'axe à chaque coupe : on coupe d'abord selon x puis selon y. Le problème de la dichotomie c'est de recoller les morceaux après les avoir décoller. L'idée est de trouver la meilleure option entre recoller le dernier terme de l'une avec le premier terme de l'autre ou le premier terme de l'une avec le dernier terme de l'autre, etc...

Au final cette méthode n'est que très rarement optimale on obtient des distances plutôt grandes, bien que parfois pour des certains graphes la solution semble plutôt optimale.

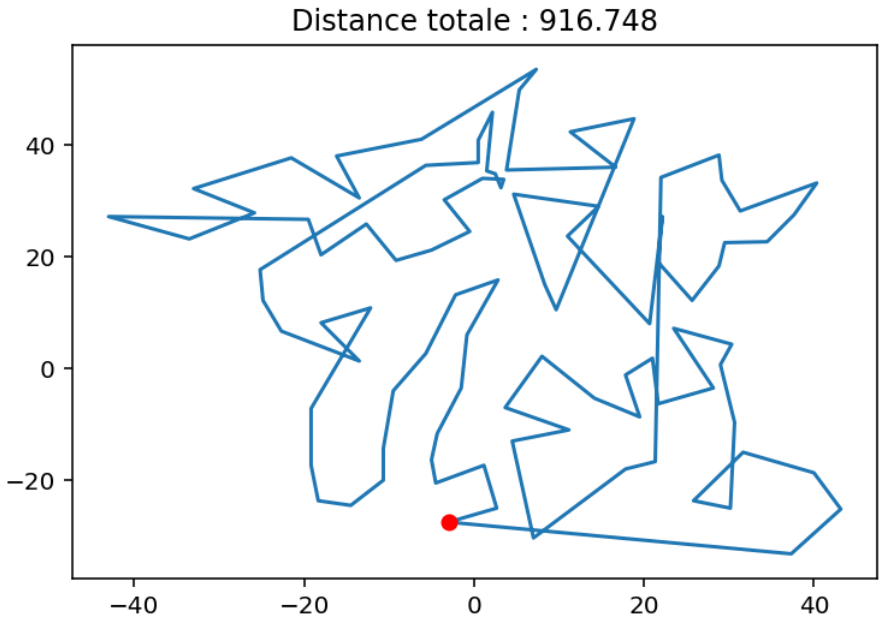

Figure 2.4 – La solution dichotomique n'est pas exceptionnelle pour l'exemple 2

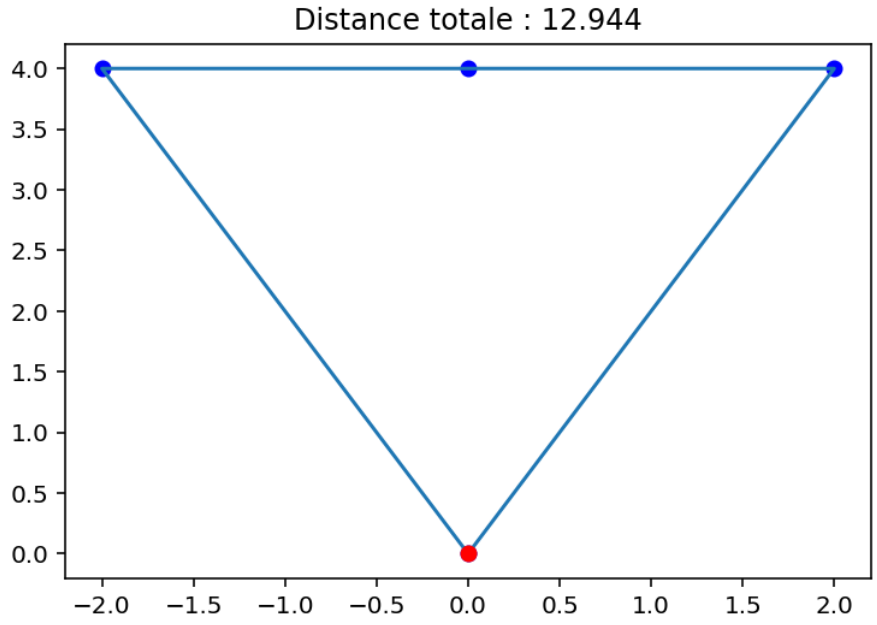

Figure 2.5 – La solution dichotomique est optimale pour l'exemple 3

2.2 Heuristiques locales

Le but des heuristiques locales est de partir d'une solution sous optimale, et de l'améliorer jusqu'à obtenir le meilleur chemin possible.

Heuristique 1

Nous savons que la méthode exhaustive est très efficace pour des petits jardins (c'est à dire avec peu d'arbres). On a utilisé l'efficacité de la méthode exhaustive pour optimiser la méthode gloutonne pondérée.

On découpe notre solution presque optimale en paquet de taille assez petite pour que la méthode exhaustive soit rapide et utile, On a donc codé une nouvelle fonction exhaustive qui prend en argument une liste de points et retourne le plus court chemin entre le premier et le dernier point de la liste. On est alors sur que cette solution pour ce tronçon est optimale et on observe par l'expérience de meilleur résultat qu'avec seulement la méthode gloutonne pondérée :

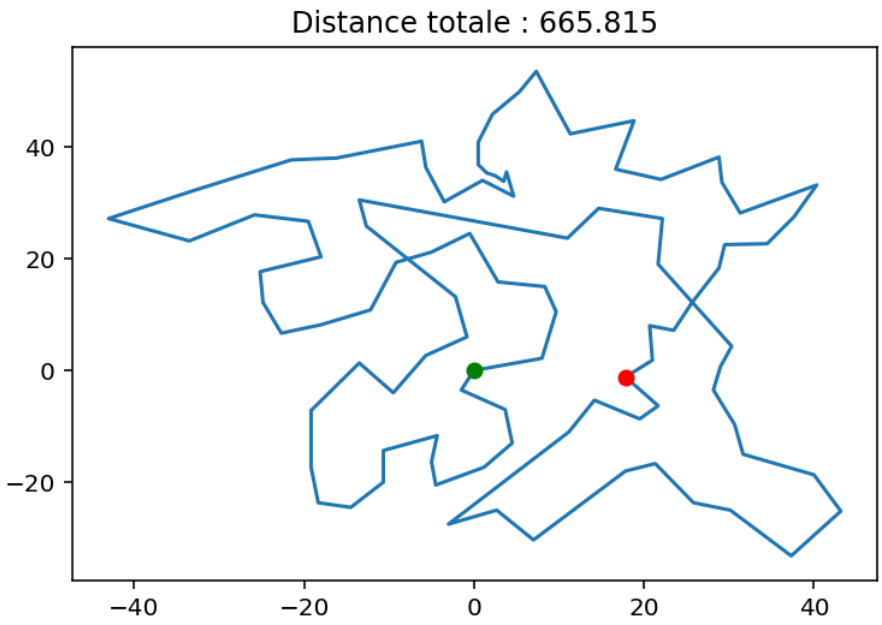

Figure 2.6 – Exemple 2 traité par le glouton pondéré, amélioré par l'heuristique locale

Heuristique 2

On a aussi essayé une heuristique 'glouton2' de parcours inspirée d'un algorithme glouton, modifiée pour améliorer la qualité du chemin via des exhaustives locales.

L'idée générale est la suivante :

— On part d'un point initial (souvent l'origine).

— À chaque étape, on sélectionne le **point le plus proche** du dernier point ajouté.

— Au lieu d'ajouter un seul point, on identifie un **groupe de points proches** (selon un facteur α qui étend le voisinage).

— Sur ce groupe, on applique un **algorithme exhaustif local** (brute-force) pour déterminer l'ordre optimal de visite.

— Ce sous-chemin est ensuite concaténé au chemin global.

— Les points du sous-chemin sont ensuite supprimés de la liste des points restants.

Cette méthode permet d'éviter certains pièges du pur glouton, qui peut faire des choix trop locaux et sous-optimaux sur le long terme.

Limites et perspectives d'amélioration

Même si cette approche améliore la qualité du chemin par rapport à un algorithme glouton classique, elle présente encore certaines limites :

— L'algorithme **ne prend pas encore en compte la destination suivante** (le futur point minimal situé hors du groupe local), ce qui limite sa capacité d'anticipation.

— Le choix du sous-groupe pourrait être affiné en considérant aussi leur **position relative au chemin global**, et non uniquement leur distance entre eux.

— **Pour certains jeux de données**, en particulier ceux où les points sont **éparpillés et équidistants**, l'approche ne donne pas de gain significatif par rapport à une solution gloutonne simple. En revanche, elle est plus performante lorsque les points ont tendance à se **regrouper localement**.

Exemple qui illustre cela :

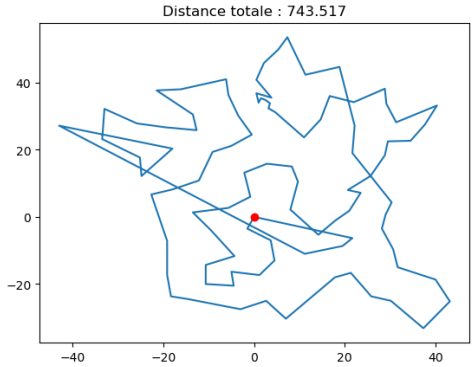

Figure 2.7 – Exemple 2 traité par le glouton basique

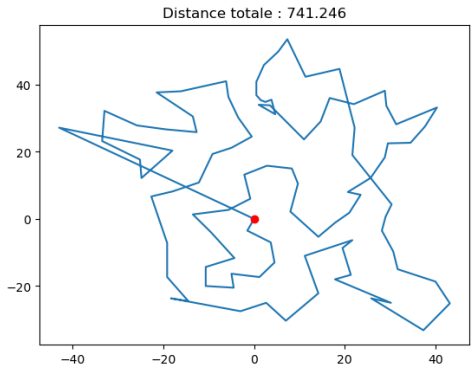

Figure 2.8 – Exemple 2 traité par le glouton avec de l'exhaustif

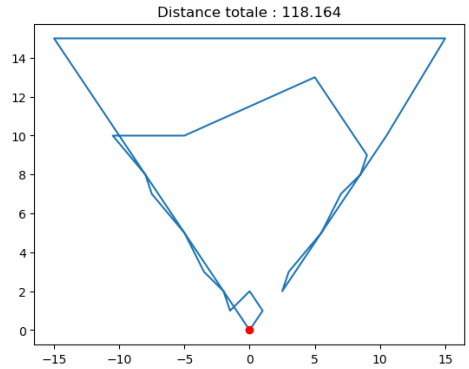

Figure 2.9 – Exemple 4 traité par le glouton basique

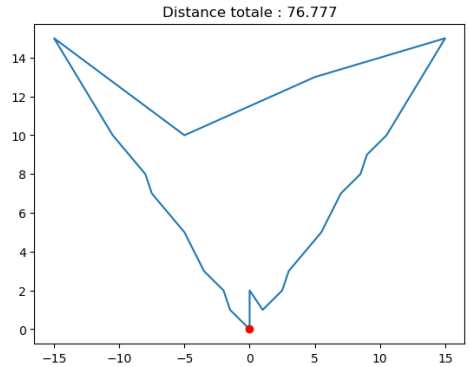

Figure 2.10 – Exemple 4 traité par le glouton avec de l'exhaustifs

2.3 Metaheuristiques

Une métaheuristique correspond à un algorithme d'optimisation qui permet en général d'obtenir des solutions presque optimales. Un tel algorithme n'est pas spécifique à un problème donné, il peut être utilisé pour tout problème remplissant les conditions suivantes :

— **Evaluation** : Pouvoir évaluer et comparer la qualité des solutions.
— **Voisinage** : Être capable de créer à partir d'une solution une nouvelle solution proche.

Dans le cas d'une structure de problème remplissant ces conditions, nous pouvons envisager une métaheuristique.Nous avons essayé un algorithme de *recuit simulé*. Le principe est le suivant :

Etape 1 : Initialisation de la solution

On commence par fabriquer une solution non-optimale comme une base de travail. On peut la générer aléatoirement mais le mieux est de fabriquer une solution relativement bonne, cela va simplifier l'optimisation de la solution. Un algorithme glouton, par exemple, peut nous donner - *dans la plupart des cas* - rapidement une première combinaison raisonnablement qualitative.

Etape 2 : Optimisation

Une fois que on possède une proposition de solution, on va chercher à l'améliorer, un peu à la manière d'une heuristique locale. La grande différence, qui représente le vrai intérêt d'une métaheuristique, est que la nouvelle solution crée à partir de celle précédente n'as pas forcément une meilleure évaluation. On effectue un grand nombre d'itérations, dans laquelle à chaque fois on réalise les actions suivantes :

— *On crée de manière aléatoire une solution 'voisine' (proche) de la solution précédente (`new = voisin(solution)`)*
— *On évalue la qualité de la nouvelle solution (`E = evaluation(new)`)*
— *On remplace la solution selon la probabilité de 1 si elle est meilleure, sinon de $P = e^{\frac{-\Delta E}{T}}$ avec ΔE la différence d'évaluation entre la nouvelle solution et l'ancienne. T correspond à une fonction température, qui peut être n'importe quelle fonction positive décroissante du numéro de l'itération. Son rôle étant de conditionner la probabilité d'accepter une solution d'évaluation plus grande que la précedente. Plus T est grand, plus la probabilité P est importante, ainsi plus on avance dans les itérations moins l'algorithme doit accepter des solutions plus "mauvaises".*

Pourquoi accepter des solutions "moins bonnes" ?

Même si on possède une solution correcte, il est possible qu'on ne puisse pas l'améliorer jusqu'à la quasi-optimalité avec des heuristiques locales. En effet, une heuristique locale, comme son nom l'indique, modifie la solution localement, mais ne change pas profondément sa structure. Or, dans certains cas, il faudrait modifier plus globalement la solution pour converger ensuite vers la quasi-optimalité. Prenons l'exemple de la tournée du jardinier. Voici un exemple de bon parcours, mais pas optimal.

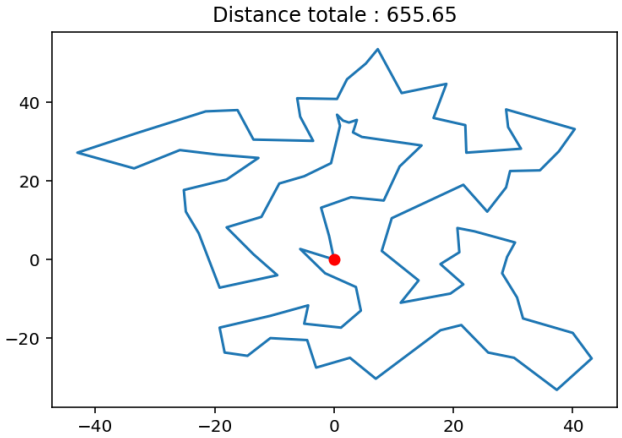

Figure 2.11 – Exemple de parcours

On voit que il n'y a aucun croisement, le parcours ne semble pas être améliorable avec des changements locaux. On peut parler de "minimum local". En effet, ce parcours correspond au meilleur parcours dans cette structure globale, et toutes les solutions proches sont visiblement moins bonnes (moins bonne évaluation), pourtant on sait qu'il n'est pas optimal, donc il ne correspond pas au parcours donnant l'évaluation minimale globale. Si ce parcours ne nous satisfait pas, il faut donc revoir toute sa structure, et donc accepter d'envisager des parcours d'évaluation moins bonne. C'est ce que permet une métaheuristique comme le recuit simulé.

L'enjeu d'un algorithme de recuit simulé est de bien choisir les fonctions température et `voisin`. Nous avons implémenté 4 fonctions pouvant faire office de températures, dans le but de choisir la plus performante dans le problème étudié.

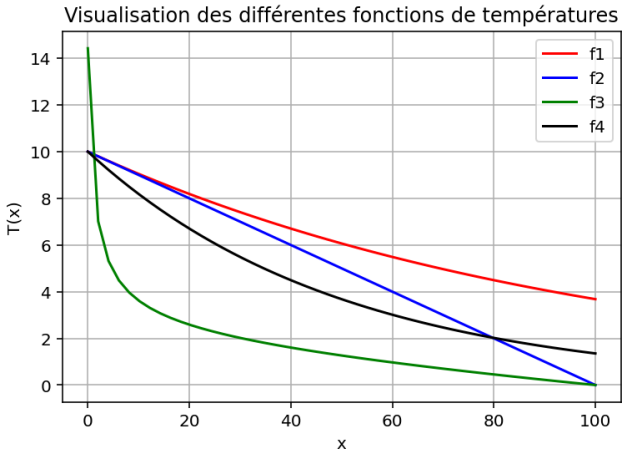

Figure 2.12 – Comparaison des candidats pour température

Le choix de température a un vrai impact sur l'efficacité de notre algorithme. Si on choisit f_4 par exemple, la température chuter rapidement au début, donc l'algorithme n'accepte de grands changements seulement au toutes premières itérations. En revanche, si on choisit $f1$, la température diminue constamment mais conserve une valeur importante, donc continue jusqu'au bout d'accepter des solutions d'évaluations moins bonnes. L'algorithme fonctionne bien

avec les 4 fonctions mais nous avons choisi $f4$, celle qui visiblement nous donnait les meilleurs résultats.

Le choix de la fonction voisin est aussi important. Nous avons implanté 3 fonctions voisin qui crée toutes trois d'une manière différente une nouvelle solution proche. Les trois fonctions travaillent sur une copie de la solution précédente.

- `voisin1` échange deux points choisis aléatoirement. Sa complexité est de $\mathcal{O}(n)$ a cause de la création d'une copie d'un tableau de taille (n,2).
- `voisin2` choisis aléatoirement une séquence de la solution et inverse l'ordre des points. Sa complexité est également de $\mathcal{O}(n)$ a cause de la création d'une copie d'un tableau de taille (n,2).
- `voisin3` choisis pour chaque couple de point de la solution si il échange les deux points ou non, avec une probabilité de 0.5

Le point commun de ces trois versions est que le degré de liberté est maximal. En effet, avec de nombreuses itérations, on peut créer tous les parcours possibles. Ce ne serait pas le cas si on modifiait la solution en la découpant identiquement à chaque fois. La création d'une solution proche constitue la plus grosse partie des calculs dans une itération. Étant donné que on compte faire de nombreuses itérations, l'algorithme est d'autant plus rapide que la fonction voisin s'exécute rapidement. Ici nos trois fonctions ont la même complexité, donc le choix se porte plus sur leur efficacité. On utilise plutôt `voisin2`, qui est dans notre cas en moyenne plus efficace.

Remarque

La méthode d'initialisation est aussi un paramètre à choisir pour améliorer le recuit simulé. En effet, nous avons clairement obtenu de meilleurs résultats en utilisant un algorithme glouton amélioré pour créer la première solution au lieu d'un algorithme glouton basique qui choisit l'arbre le plus proche. L'algorithme glouton amélioré produit une solution initiale plus proche de la quasi-optimalité, et de meilleure structure, ce qui rend plus facile et efficace l'étape d'optimisation.

3 Calcul de complexité des algorithmes

3.1 Algorithme exhaustif

Plus court chemin exhaustif sur l'ensemble des points

L'algorithme explore récursivement tous les ordres possibles dans lesquels on peut visiter les n arbres donnés. À chaque appel récursif, il :

- ✓ choisit un arbre parmi ceux restants,
- ✓ calcule la distance à partir du dernier point du chemin courant,
- ✓ met à jour le chemin et les arbres restants,
- ✓ et relance l'appel récursif.

À chaque niveau de récursion, un arbre est choisi parmi ceux restants. Il y a donc un total de $n!$ permutations possibles, et donc :

$$\text{Nombre total d'appels récursifs} = n!$$

À chaque appel récursif, les opérations suivantes sont effectuées :

- ✓ calcul d'une distance : coût constant $\mathcal{O}(1)$,
- ✓ suppression d'un arbre de la liste (via `np.delete`) : $\mathcal{O}(n)$,
- ✓ ajout d'un point au chemin (via `np.vstack`) : $\mathcal{O}(n)$.

Le coût d'un appel récursif est donc au plus $\mathcal{O}(n)$.
L'algorithme effectue $n!$ appels récursifs, chacun de coût $\mathcal{O}(n)$.

Ainsi, la complexité en temps est :

$$\boxed{\mathcal{O}(n \cdot n!)}$$

Plus court chemin exhaustif sur un tronçon de points donnés

Calculons la complexité de la fonction. Les étapes coûteuses en complexité de la fonction sont :

- ✓ Copie des points (arbres) du problème dans une liste. Complexité en $O(n)$ avec n le nombre d'arbres du problème.
- ✓ Une boucle qui itère sur chaque permutations possibles de l'ensemble des arbres du parc. On itère donc $n!$ fois. En effet, la ligne suivante

```
1  itertools.permutations(np.array(points_intermediaires)):
```

génère toutes les permutations possibles des éléments de `points_intermédiaires`. Chaque permutation est un tuple contenant les éléments dans un ordre différent. L'ensemble des permutations possibles de n élément est de cardinal $n!$, en effet :

- Pour le premier rang, on a le choix entre n éléments
- Pour le second rang il reste $n-1$ choix possibles
- pour le $(n-1)$-ième rang, il nous reste 2 choix possible
- Et pour le dernier il n'en reste plus qu'un

Ce qui nous donne $n(n-1)(n-2) \times \cdots \times 2 \times 1$ permutations possibles.

✓ Pour chaque itérations (permutation), on ajoute le point de départ au début et le point d'arrivée à la fin de la permutation. Complexité en $O(n)$.

✓ on utilise la fonction `distance_totale` qui a une complexité en $O(n)$

✓ on fait une copie de la permutation si jamais elle a une meilleure distance. Donc complexité en $O(n)$.

Ainsi, on a des opérations de complexité en $O(n)$ dans une boucle qui itère $n!$ fois. La complexité finale de cette fonction est en

$$\boxed{O(n \cdot n!)}$$

3.2 Parcours glouton

La fonction gloutonne construit la tournée en prenant à chaque étape l'arbre non visité le plus proche. Ainsi, à chaque étape ($O(n)$), pour identifier le plus proche, il compare la distance avec tous les autres $O(n)$. Ainsi la complexité est de $O(n)$

3.3 Recuit simulé

La complexité de l'étape d'initialisation dépend de l'algorithme utilisé. On va l'appeler C_{init}

L'étape d'optimisation est une boucle. Calculons la complexité au sein d'une itération :
A chaque itération, on effectue les actions suivantes :

✓ On évalue la solution actuelle $\rightarrow \mathcal{O}(n)$

✓ On crée une nouvelle solution avec `voisin` $\rightarrow \mathcal{O}(n)$

✓ On évalue la nouvelle solution $\rightarrow \mathcal{O}(n)$

✓ On modifie ou non la solution actuelle $\rightarrow \mathcal{O}(n)$

La complexité d'une itération est donc $C_{iter} = \mathcal{O}(n)$ Comme on effectue `nb_iter` itérations, la complexité globale de l'algorithme `recuit_simulé` est :

$$\boxed{C_{\texttt{recuit\_simulé}} = \mathcal{O}(nb\_iter \cdot (n + C_{\texttt{init}})}$$

3.4 Inverser arête et inverser arbre

En parcourant chaque arêtes (liaison entre 2 arbres) de la tournée et en examinant ensuite la permutation avec toutes les autres arêtes, la complexité est d'ordre n^2. C'est visible par la présence de 2 `for i in range(n)` imbriqués. Il en est de même pour la fonction `inverser_arbre`. On peut voir ci dessous la tendance quadratique de la complexité grâce au temps d'exécution des fonctions, mais c'est à nuancer car ces fonction se répètent sur elles même tant qu'elles sont efficaces ce qui explique les modulations de la courbe.

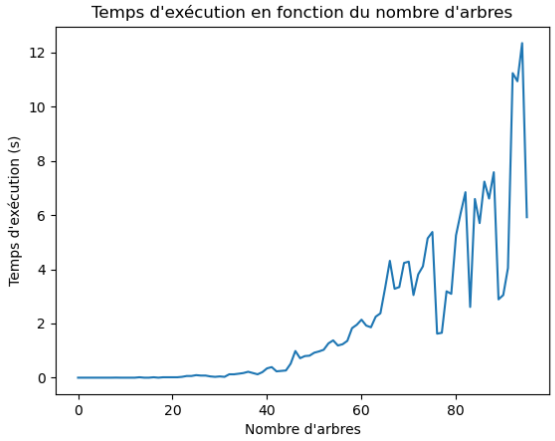

Figure 3.1 – Temps d'exécution de inverse_arbre en fonction du nombre d'arbres

4 Analyse expérimentale des algorithmes et comparaison avec la méthode exhaustive

Nous avons tenté plusieurs méthodes, avec différents résultats pour chacune. Tout d'abord, commençons par la méthode exhaustive : c'est la seule qui donne à tous les coups un résultat optimal cependant elle est très lente à converger. C'est donc une méthode pratique pour résoudre des petits problèmes comme l'exemple 1. Comme l'on peut le voir sur la FIGURE 4.1, le temps d'exécution de l'algorithme exhaustif évolue en $n!$ où n est le nombre d'arbres dans le parc, par exemple, pour un problème à 12 arbres, le temps d'exécution atteignait déjà l'ordre de la minute.

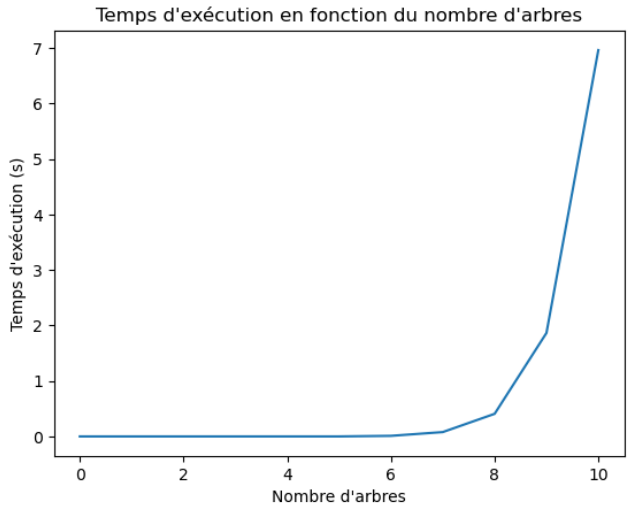

Figure 4.1 – Temps d'exécution de l'algorithme exhaustif en fonction du nombre d'arbres

Ensuite les méthodes gloutonne et gloutonne pondérée sont deux méthodes très similaires, bien que la seconde génère des solutions qui semblent plus simples à améliorer car nos heuristiques locales fonctionnaient mieux sur celles-ci. Nos heuristiques locales semblent être efficaces car elle améliore déjà significativement les solutions données par les méthodes gloutonnes. La méthode metaheurisitique recuit simulé est très efficace pour transformer une solution gloutonne en quasi-optimale. Son temps d'exécution est largement raisonnable devant l'algorithme exhaustif (solution quasi-optimale en moins de 30 secondes).Néanmoins, on obtient un résultat minimum de 627.001 en distance minimale pour les données du fichier exemple2.txt, en effectuant à la suite le recuit simulé et des heuristiques locales. Cette combinaison des méthodes permet de sélectionner une structure de solution quasi-optimale avec le recuit simulé, puis de l'améliorer encore le plus possible avec les heuristiques locales. Lorsque l'on affiche cette tournée de distance 627.001, il devient compliqué de trouver des améliorations possibles à l'œil nu.

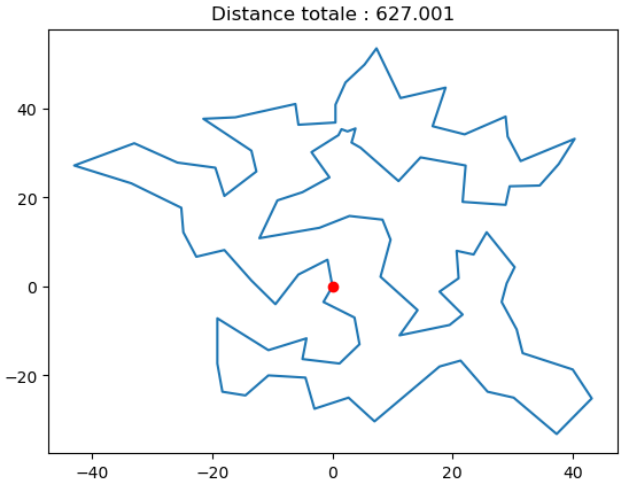

Figure 4.2 – Meilleur solution obtenue

C'est donc en mixant approche presque optimale, heuristiques, métaheuristiques, que l'on obtient le meilleur résultat.

En cherchant des données sur internet et en partageant avec nos camarades, nous avons un résultat optimal, à quelques mètres près puisqu'il semble exister une tournée de taille 619. On espère alors que le jardinier pourra réaliser ces quelques pas supplémentaires...