# Approximation du problème de tourné de véhicule avec des méta-heuristiques


- Benoit Leroux
- Justin Bauer


## Description du problème

Etant donné $n$ points du plan représentant des clients, un point spécial appelé dépôt et un entier $m$, trouvez l'ensemble de poids minimal de $m$ cycles ayant pour origine le dépôt et tel que chaque point est visité une unique fois.

Où le poids d'une solution est la somme des longueur des parcours de chaque cycle (on peut prendre la distance euclidienne pour trouver les longueurs).

## Algorithme génétique

### Idée générale

On laisse évoluer une population (ie. un ensemble d'individu représentant chacun une solution potentielle de notre problème) initiale dans un environnement. A chaque itération, les meilleurs solutions sont sélectionnés de manière probabiliste et se reproduise entre elles pour former la prochaine génération.

La sélection permet à l'algorithme de se rapprocher d'une bonne solution (c'est ce qui fait converger l'algorithme).
Le fait d'avoir un grand ensemble d'individu permet une grande diversité de solution, ce qui permet d'éviter d'être bloquer dans des minimums locaux.

### Implémentation

#### Représentation d'une solution

On représente une solution à l'aide d'une liste de float compris entre $0$ et $m-1$ inclus. Une telle solution est généré à partir de la fonction suivante :

```
def random_float_cycles(n: int, m: int) -> cycles:
    return np.array([randint(0, m-1) + uniform(0, 1) for _ in range(n)])
```

La sémantique d'une telle solution est la suivante :

- chaque valeur de la liste est indexé par un entier de $0$ à $n-1$ représentant un point
- La partie entière de la $i$ème valeur désigne le numéro du véhicule qui va parcourir ce point
- Pour une sous-liste de point parcouru par le même véhicule (ie. même partie entière), l'ordre de parcours est donné par la permutation des indices engendré par le tri d'une telle sous-liste

Exemple pour un ensemble de 6 points et 2 véhicules:

| **0** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1.2 | 0.5 | 1.04 | 0.75 | 1.3 | 1.15 |

On tri cette liste :

| **1** | **3** | **2** | **5** | **0** | **4** |
|---|---|---|---|---|---|
| 0.5 | 0.75 | 1.04 | 1.15 | 1.2 | 1.3 |

#### Crossover [1]

Le but de cette étape est de générer une solution fille à partir de deux solutions parentes $p_1$ et $p_2$.

Pour cette étape il faut introduire un paramètre $p$ représentant le seuil de probabilité préférentiel entre $p_1$ et $p_2$.

La solution fille est alors généré en ajoutant soit un élément de $p_1$ soit un élément de $p_2$ respectivement si $p \geq v$ ou si $p < v$ où $v$ est une valeur aléatoire entre 0 et 1 généré à la volée.

Une telle fonction est implémenté par le code python suivant :

```
def crossover(self, p1: cycles, p2: cycles) → cycles:
        """
        @retval a cycles sharing similarity with p1 and p2
        """
        return [p1[i] if uniform(0, 1) ⩾ self.cross_proba else p2[i] for i in range(self.n)]
```

### Fitness et itération

Pour la fonction fitness, il suffit de parcourir la solution et de sommer les distances entre chaque points.

La fonction d'itération est implémenté par la fonction python suivante :

```
def evolve(self) → None:
        pop = self.init_pop
        for _ in range(self.iteration):
            pop = self.next_gen(pop)

        self.final_pop = pop
```

où la fonction `next_gen` est la suivante :

```
def next_gen(self, pop: list[cycles]) → list[cycles]:
        """
        TODO
        - Select random individuals in pop
        - Class these individuals by fitness (priority queue insertion)
        - Reproduce best individuals while next_pop size ⩽ pop_size
        """
        next_gen = []
        while len(next_gen) < self.pop_size:
            sub_select = sorted(sample(pop, 10), key=lambda x:self.fitness(x))
            sub_select = sub_select[:5]
            p1 = choice(sub_select)
            p2 = choice(sub_select)

            next_gen.append(self.crossover(p1, p2))

        self.scores.append(self.fitness(next_gen[0]))

        return next_gen
```

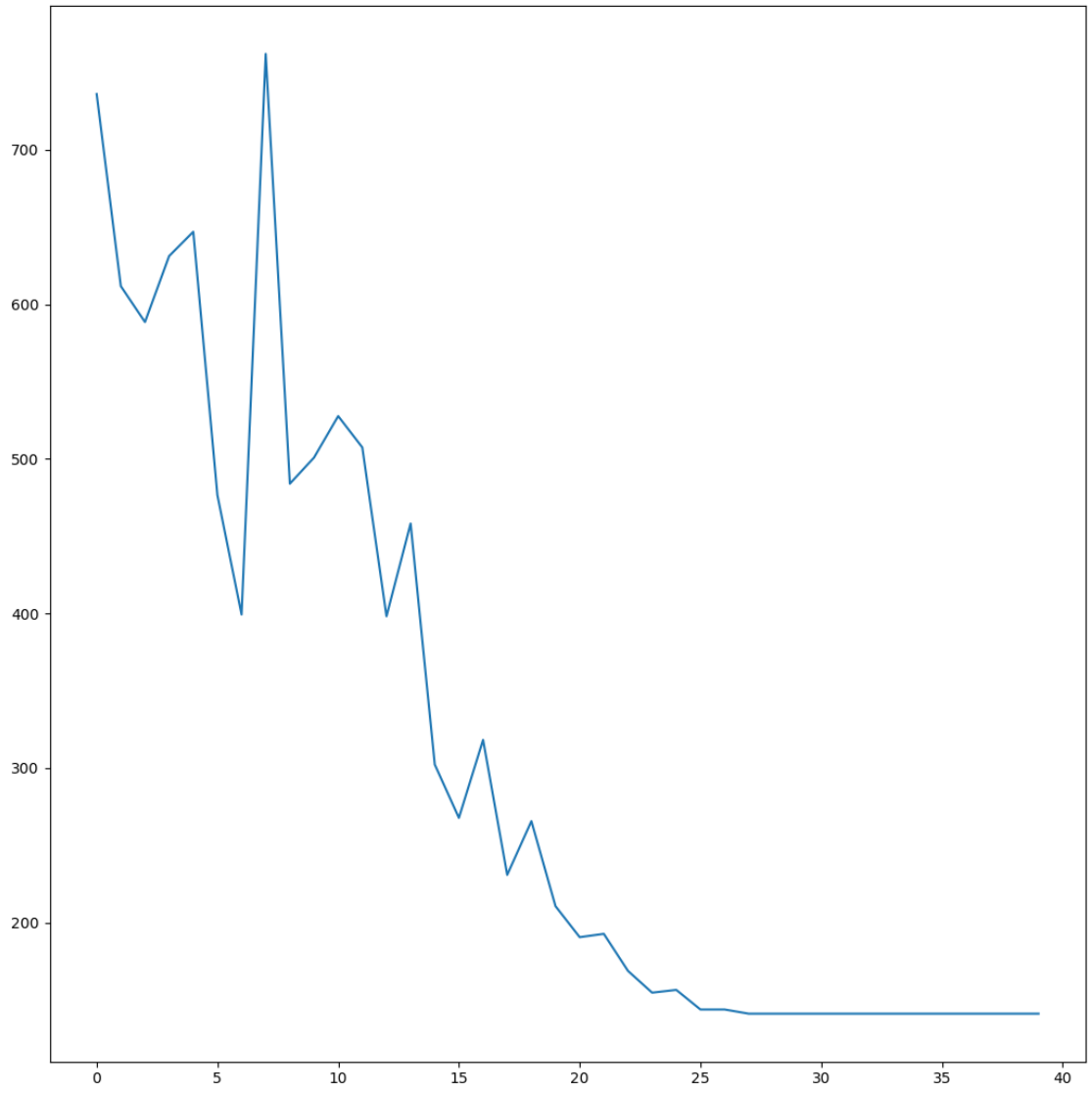
On pourrait former la nouvelle génération d'une autre manière, mais celle là marche assez bien avec les données testées et nécessite peu de calcul.

## Quelques résultats

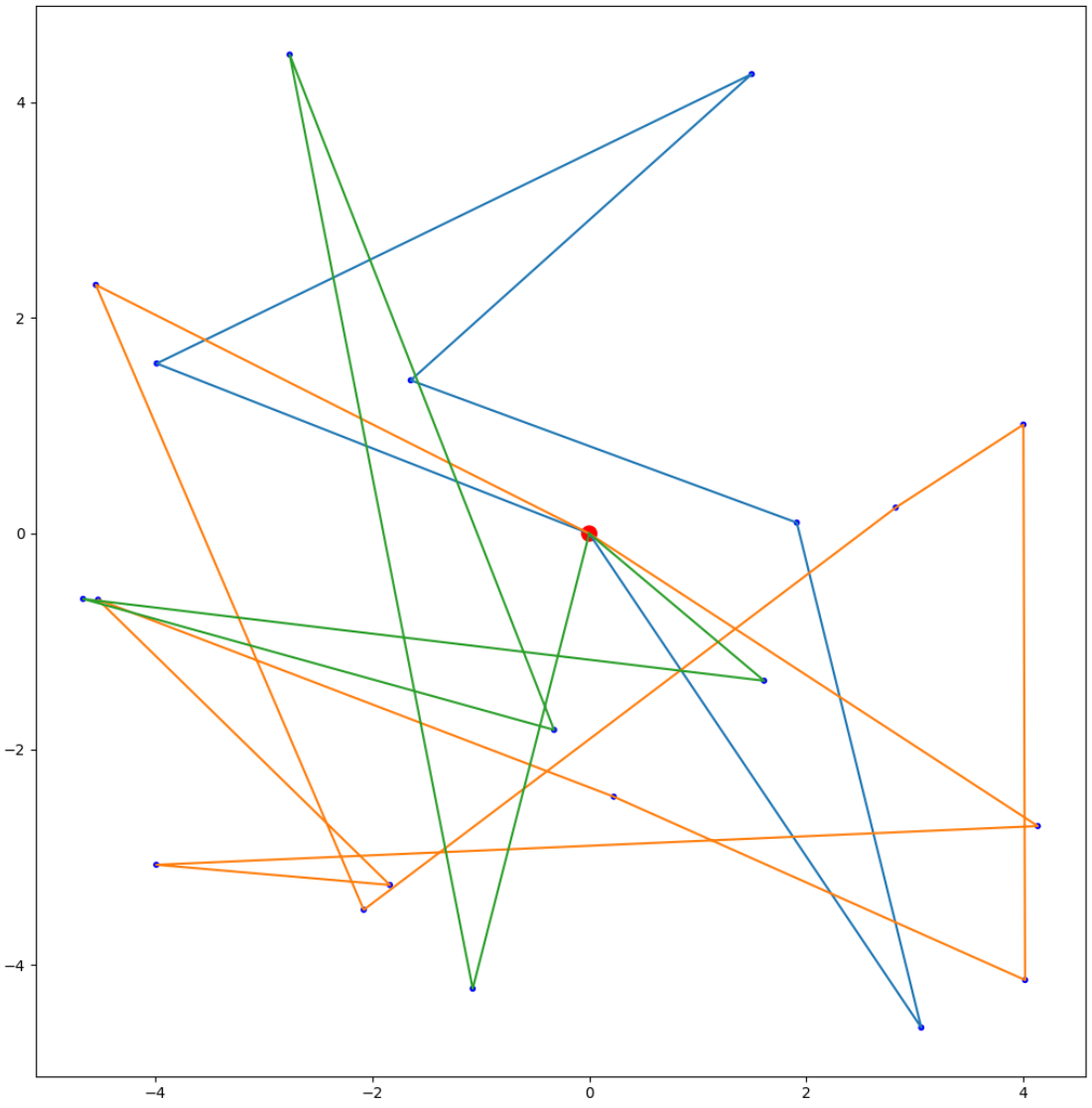
### Algo génétique

Les tests ont été effectué à partir de donné généré aléatoirement. La complexité de l'algorithme n'a pas encore été optimisé (il y a beaucoup de calcul que l'on fait en double), ce qui fait que le calcul d'une bonne solution prend beaucoup de temps sur le dataset donné. Pour optimiser cette complexité, il faudrait mettre en cache la permutation d'indice qui ordonne une solution donné, de plus pour mettre à jour une telle information lors d'un crossover, il faudrait utiliser un tas min un peu modifié pour extraire une valeur et en insérer une autre de manière optimisé.

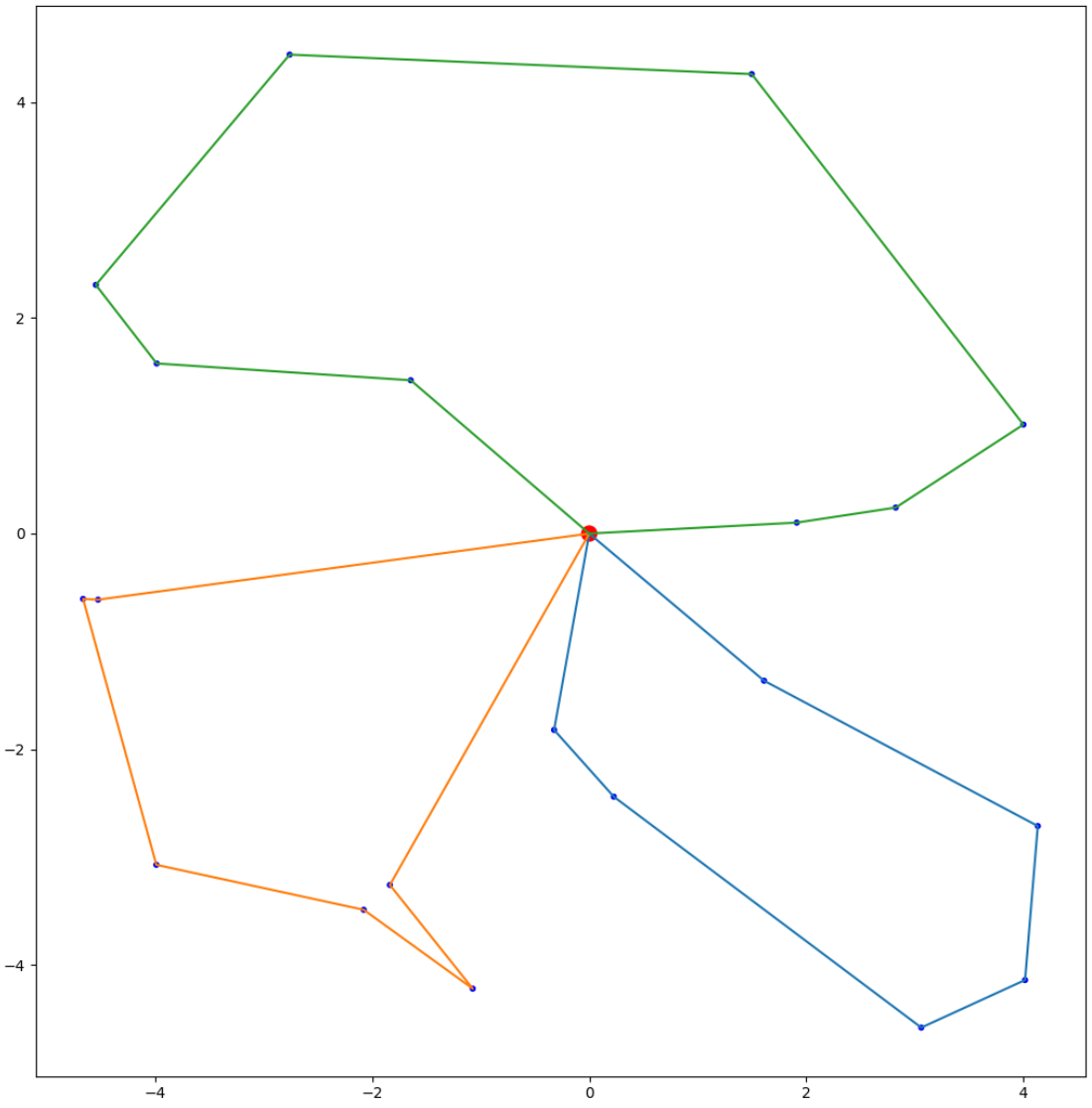
Fitness:



Solution initiale :



Solution finale (sélectionné aléatoirement dans la population finale, probablement pas la meilleur):




## Références

1. Genetic Algorithms and Random Keys for Sequencing and Optimizations - James C. Bean - 1992↩