# Projet - Optimisation combinatoire multiobjectif

Réalisé par
**Louis Bard**

# Table des matières

**1 Introduction** **2**

**2 Instances** **3**

**3 Filtrage de solutions dominées** **4**

**4 Algorithmes pour le mTSP** **6**
4.1 Approche scalaire . . . . . 6
4.2 Approche Pareto . . . . . 9

**5 Comparaisons des algorithmes** **11**

**6 Conclusion** **12**

# 1 Introduction

Le TSP (problème du voyageur de commerce, ou travelling salesman problem) est un problème classique de l'optimisation combinatoire. Étant donné un ensemble de villes, il s'agit de trouver le chemin 'optimal' qui relie toutes les villes, sans passer plus d'une fois par la même. L'optimalité se rapporte généralement à une distance parcourue. Dans sa variante mono-objectif, le TSP est connu comme un problème NP-difficile : on ne connait pas d'algorithme permettant de trouver une solution exacte en un temps polynomial. Néanmoins, de nombreuses applications réelles se résument en un problème de TSP, ou se doivent de résoudre le TSP comme sous-problème.

Dans sa variante multi-objectif, le mTSP (multi-objective TSP, avec m objectifs) vise non seulement à minimiser la distance totale, mais également le temps total, le coût total, etc. Par conséquent, il est supposé que plusieurs quantités, telles que la distance, le temps et le coût, sont affectées à chaque paire de villes.

Plus formellement, le mTSP peut être défini par un graphe complet G = (V,E), où V = v1,v2,...,vn est un ensemble de noeuds (les villes) et E = [vi,vj], vi,vj  V est un ensemble d'arcs. À chaque arc [vi,vj]  E est affecté un coût positif ck(i,j) par fonction objectif k  1,...,m. À chaque objectif correspond donc une matrice de coûts. Le but est de trouver l'ensemble des permutations cycliques simples  de taille n qui sont Pareto optimales pour f1,f2,...,fm, avec :

$$f_k(\pi) = \sum c_k(\pi(i), \pi(i+1)) + c_k(\pi(n), \pi(1)) \tag{1}$$

Dans le cas général, le mTSP est NP-difficile, et la taille de l'ensemble Pareto optimal croit exponentiellement avec la taille du problème (le nombre de noeuds n)

Les expérimentations ont été réalisées sous Python 3.7.6 avec un Macbook - 2,2 GHz - 16 GO RAM.

# 2 Instances

Les instances que nous allons utiliser proviennent de la page web de L. Paquete (instances de type aléatoires, à 100 villes). Un fichier d'instance contient les valeurs pour un seul type de coût entre chaque paire de noeuds (un fichier = une matrice de coûts = un objectif). Il faut donc lire autant de fichiers d'instance que d'objectifs que l'on veut prendre en compte.

Nous nous intéressons aux 6 fichiers d'instances suivants :

- randomA100.tsp
- randomB100.tsp
- randomC100.tsp
- randomD100.tsp
- randomE100.tsp
- randomF100.tsp

A partir de ces fichiers nous allons concevoir des instances de mTSP à deux objectifs, en fusionnant les fichiers d'instances deux par deux.

Le programme qui permet de lire les instances, ( voir github / annexe ) va permettre de stocker ces données dans une matrice de taille (100 , 100). Les lignes et colonnes de la matrice représente un couple de ville i,j , et la valeur à l'intersection est un tuple ( A , B ) contenant les deux objectifs.

Pour une instance donnée, la matrice est représentée comme ci-dessous :

| | Ville 1 | Ville 2 | ... | Ville 100 |
|---|---|---|---|---|
| Ville 1 | (a,b) | (a,b) | ... | (a,b) |
| Ville 2 | (a,b) | (a,b) | ... | (a,b) |
| ... | (a,b) | (a,b) | ... | (a,b) |
| Ville 100 | (a,b) | (a,b) | ... | (a,b) |

FIGURE 1 – Représentation des instances de données

On a aussi implémenté une fonction d'évaluation. Celle-ci se base sur la formule donnée en introduction. En prenant un chemin de solution ( i.e un ensemble de villes : 1,5,13 ... ) elle va renvoyer un tuple, qui consiste en la somme de la contribution de chacun des objectifs.

# 3 Filtrage de solutions dominées

Il s'agira ici d'implémenter deux algorithmes de filtrages, qui permettront de retourner les solutions non-dominées d'un ensemble de solutions. Nous nous baserons sur la définition suivante pour réaliser les algorithmes de filtrage :

Definition (Pareto dominance) An objective vector $u = (u_1, \ldots, u_n)$ is said to dominate $v = (v_1, \ldots, v_n)$ (denoted by $u \prec v$ ) if and only if no component of $v$ is smaller than the corresponding component of $u$ and at least one component of $u$ is strictly better, i.e. $\forall i \in \{1, \ldots, n\} : u_i \leq v_i \wedge \exists i \in \{1, \ldots, n\} : u_i < v_i$

**Off-line** : Dans une stratégie «off-line», tous les points que l'on désire filtrer sont connus «en même temps». L'algorithme prend un ensemble de solutions en entrée et retourne le sous-ensemble de solutions non-dominées. Les deux objectifs seront à minimiser. Cet algorithme contient deux étapes : il va évaluer la fitness de chaque solution, puis ensuite filtrer l'ensemble.

Pour vérifier le bon fonctionnement de ce premier algorithme nous avons générer 500 solutions aléatoires puis nous avons filtrer les solutions dominées. Un exemple de visualisation est donné ci-dessous.

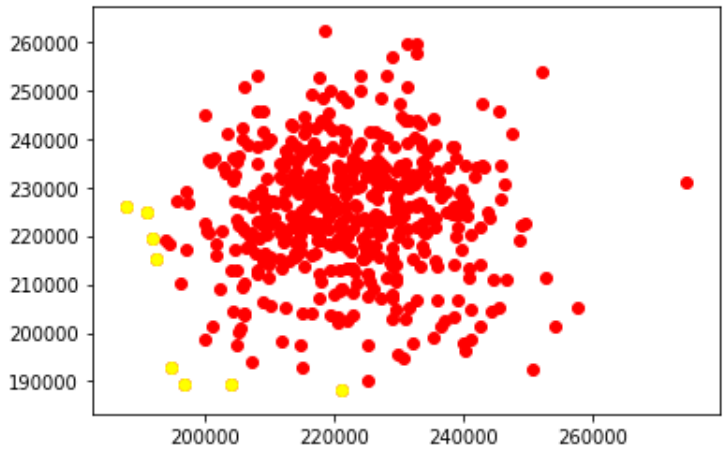


FIGURE 2 – Filtrage off-line avec les solutions non dominées en jaune. Objectif 1 en abscisse et objectif 2 en ordonnée

On observe bien que il existe un front pareto de solutions non dominées.

**On-line** : Dans une stratégie «on-line», l'algorithme maintient un ensemble de solutions que l'on appelle une archive A (initialement un ensemble vide). De nouvelles solutions lui arrivent l'une après l'autre. À chaque fois qu'il prend une nouvelle solution x en entrée, l'algorithme doit décider s'il ajoute la solution x à l'archive A, et doit éventuellement supprimer les solutions de A qui sont dominées par x. Une fois encore, les objectifs sont à minimiser.

Nous allons générer P solutions aléatoires et mettre l'archive à jour séquentiellement

avec chacune des solutions.

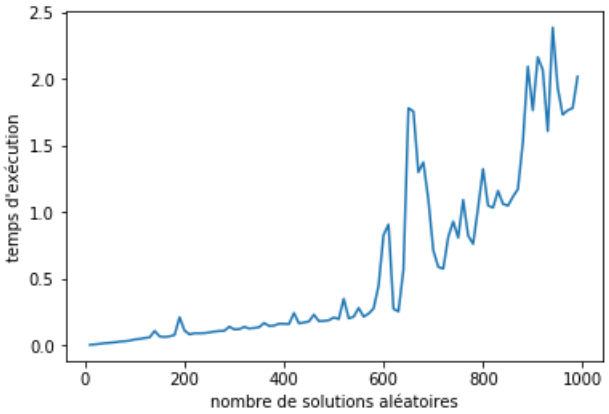


FIGURE 3 – Temps d'exécution en fonction de P

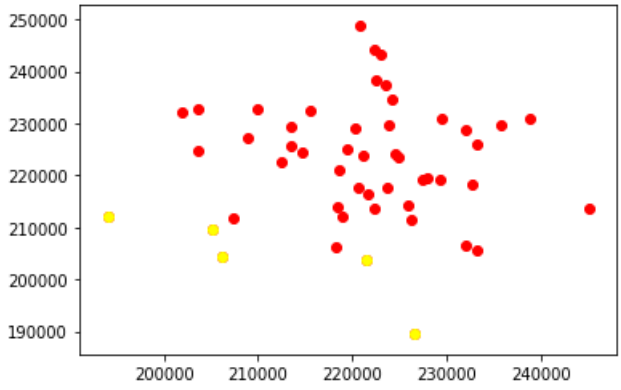


FIGURE 4 – Filtrage on-line avec les solutions non dominées en jaune. Objectif 1 en abscisse et objectif 2 en ordonnée

On constate tout d'abord sur la figure 4 que l'algorithme fonctionne bien, le front Pareto apparaît bien en jaune. Concernant le temps d'exécution , on remarque que celui-ci augmente beaucoup plus rapidement à partir de P = 600 solutions à traiter. L'algorithme prendra un temps plus considérable à filtrer un grand nombre de solutions.

# 4 Algorithmes pour le mTSP

Afin de résoudre le mTSP, nous allons nous intéresser à deux types d'approches : une approche scalaire et une approche Pareto.

## 4.1 Approche scalaire

Une approche scalaire recherche itérativement une solution Pareto approchée en se basant sur une fonction d'agrégation (transformant donc le problème initial en un problème mono-objectif) et en faisant varier (de façon intelligente) les paramètres de la fonction d'agrégation. Par exemple, en considérant une somme pondérée, on peut définir un certain nombre de poids possibles de façon à balayer l'ensemble du front Pareto. Il suffira ensuite de filtrer l'ensemble des solutions obtenues afin d'écarter les solutions dominées.

L'algorithme que nous allons implémenté est décrit ci-dessous :

**Algorithm 1 :** Restart search strategy

**forall** *vecteur de poids* $\lambda$ **do**
  *s est une solution aléatoire ;*
  $s^{'}$ = *ILS(s,* $\lambda$*)*
  *Ajouter* $s^{'}$ *à l'archive ;*
**end**
*Filtrer l'archive ;*
*Retourner l'archive ;*

Nous avons un ensemble de $m$ vecteurs de poids qui peuvent être utilisés. Chaque composant d'un vecteur de poids a la valeur $i/z$ où $i = 0,...,z$ et où $z$ est un paramètre. Tout les composants d'un vecteur de poids doivent sommés à 1.

L'algorithme de recherche local est une Iterated Local Search. Cet algorithme consiste en une recherche locale itérative pour atteindre rapidement un optimum local ( intensification) et une perturbation pour échapper aux optimas locaux ( diversification). Le critère d'acceptation est là pour contrôler la balance entre diversification et intensification.

**Algorithm 2 :** Algorithmic outline of iterated local search

s = solution initiale ;
$s^*$ = LocalSearch(s) ;
**while** *critère d'arrêt* **do**
  $s'$ = Perturbation($s^*$,history) ;
  $s^{*\prime}$ = LocalSearch($s^{'}$) ;
  $s^*$ = AcceptanceCriterion($s^*$,$s^{*\prime}$,history) ;
**end**

Nous avons les versions possibles pour cet algorithme :

- ☐ Solution initiale :
    1. solution aléatoire
- ☐ Local Search : Hill-Climbing
    1. Voisinage : 2-échange ou 3-échange
    2. Stratégie de selection : best improvement : meilleur solution améliorante parmis tout les voisins
- ☐ Perturbation
    1. 2-opt move ( aléatoire )
- ☐ Critère d'acceptation ; accepter seulement les optimas locaux améliorant

Pour le restart search strategy nous avons donc les paramètres suivants :

- ☐ Vecteur de poids :
    1. Nombre de scalarisations : 100,500,1000.
- ☐ Nombre d'itérations de l'ILS
    1. 0 , 50 ou 100

L'algorithme de Hill-Climbing consiste à partir d'une solution initiale à générer un ensemble de voisins ( avec un 2 ou 3 - échange ). Il s'agira ensuite de parcourir tout ces voisins et de sélectionner la solution à plus faible coût. Cette solution sera alors la nouvelle solution initiale. Il suffit alors de réiterer cette opération jusqu'a trouver un minimum local.

Nous allons donc évaluer l'influence de des paramètres ci-dessous sur les performances de l'algorithme.

| *Paramètres* | *Niveaux* |
|---|---|
| Nombres de scalarisations | [100,500,1000] |
| Nombres d'itérations de l'ILS | [0,50,100] |

FIGURE 5 – Paramètres de l'algorithme

Les performances ont été évaluées à l'aide d'un front de Pareto de référence et d'un indicateur d'hypervolume ( plus il est faible plus l'algorithme se rapproche du meilleur front connu). Puisque nos algorithmes ne sont pas déterministes et démarrent d'une solution initiale aléatoire, nous avons moyenné les résultats sur 30 runs. Nous avons obtenu les résultats suivants pour les trois instances :

| | AB | CD | EF |
|---|---|---|---|
| S = 100 | 0.86 | 0.84 | 0.86 |
| S = 500 | 0.83 | 0.83 | 0.84 |
| S = 1000 | 0.81 | 0.80 | 0.82 |

FIGURE 6 – Performance de l'algorithme en fonction du nombre de scalarisations et de l'instance

Ces mesures ont été réalisées avec un nombre d'itérations de l'ILS égale à 0. On observe qu'il y a une augmentation des performances de l'algorithme quand le nombre de scalarisations augmente. Cependant cela se fera au prix d'un temps d'exécutions plus conséquent.

Nous avons aussi évalué l'influence du nombre d'itérations de l'ILS avec un nombe de scalarisations S = 100 fixe. Nous avons obtenu les résultats suivants :

| | AB | CD | EF |
|---|---|---|---|
| I = 0 | 0.86 | 0.84 | 0.86 |
| I = 50 | 0.66 | 0.68 | 0.67 |
| I = 100 | 0.61 | 0.65 | 0.62 |

FIGURE 7 – Performance de l'algorithme en fonction d'itérations au sein de l'ILS et de l'instance

On constate qu'il une aumgentation significative des performances lorsqu'on passe d'un nombre d'itérations I = 0 à I = 50. ( près de 20 % ). Cependant cela se fera au prix d'un temps d'exécution beaucoup plus élevé. Choisir un nombre d'itérations de 50 peut apparaître comme un bon compromis.

## 4.2 Approche Pareto

Une approche Pareto ne maintient pas une solution unique, mais un ensemble de solutions. Elle se base généralement sur la relation de dominance Pareto. Elle vise à améliorer l'ensemble de solutions en même temps (et pas chaque solution l'une à la suite de l'autre). En règle général, une archive ou une population est maintenue et, à chaque itération, de nouvelles solutions sont crées (à l'aide d'opérateurs de voisinage ou de variations) et ajoutées dans l'archive. Il s'agit alors de mettre l'archive à jour en ne conservant que les meilleures solutions.

Nous avons décidé d'implémenter une Recherche Locale Pareto, qui est l'un des premiers algorithmes a avoir été conçu selon une approche Pareto.

Le pseudo code de cet algorithme est le suivant :

Algorithm 1 Pareto Local Search (standard sequential version)

input : An initial set of non-dominated solutions $A_0$
$\forall x \in A_0,$ set explored $(x) \leftarrow$ FALSE
$A \leftarrow A_0$
while $A_0 \neq \emptyset$ do
    $x \leftarrow$ a randomly selected solution from $A_0$
    for each $x'$ in the neighborhood of $x$ do
        if $x' \not\prec A$ then
            explored $(x') \leftarrow$ FALSE
            $A \leftarrow$ Update $(A, x')$
        end if
    end for
    explored $(x) \leftarrow$ TRUE
    $A_0 \leftarrow \{x \in A \mid$ explored $(x) =$ FALSE $\}$
end while
return $A$

(2)

Cet algorithme consiste donc à générer une archive contentant des solutions aléatoires non dominées. Ensuite il s'agira de sélectionner une solution au sein de cette archive, de générer un voisinage de cette solution et d'y trouver une solution non-dominée pour l'ajouter à l'archive. Après avoir parcouru le voisinage complet de la solution initiale celle-ci est marquée comme lue.

Le critère d'arrêt est atteint quand on aura parcouru tout les éléments de l'archive.

Nous avons le paramètrage suivant pour cet algorithme :

- ☐ Solution initiale :
    1. Solution aléatoire ou métaheuristiques
- ☐ Taille de l'archive initiale
- ☐ Voisinage
    1. Taille du voisinage
    2. Type de voisinage : 2-echange ou 3-echange

Nous avons choisis de tester cet algorithme avec une archive initiale de taille 100, contenant des solutions générées aléatoirement. La taille du voisinage sera de 100 et les voisins seront obtenus par un 2-échange.

On obtient les résultats suivants ( avec les mêmes indicateurs de performances que précèdement) :

| AB | CD | EF |
|---|---|---|
| 0.86 | 0.90 | 0.88 |

FIGURE 8 – Performance de l'algorithme en fonction d'itérations au sein de l'ILS et de l'instance

Cet algorithme apparait comme moins performant que l'approche scalaire. En effet en partant de solutions aléatoires initiales il pourrait avoir du mal à échapper à des optimas Pareto locaux. Une solution pourrait être de changer le type de voisinage ou de partir de solutions obtenues à l'aide d'heuristiques ( comme le 2-opt).

De nombreuses approches Pareto ont été développés à partir de celle proposé précdémment permettant une diversification plus importante et donc de meilleure performance, ce sont les algorithmes évolutionnaires multi-objectif.

# 5 Comparaisons des algorithmes

Nous avons procédé à une analyse expérimentale afin de comparer les performances des deux algorithmes.

Puisque nos algorithmes ne sont pas déterministes nous avons réalisé 10 runs de chacun. Le critère d'arrêt à été fixé à un temps d'exécution maximal de 1 min ( pertinent surtout pour l'approche scalaire). Pour l'approche scalaire nous avons choisis une scalarisation de 100, et un nombre d'itérations de 100. Pour l'approche Pareto nous avons choisis une archive de taille 1000, généré avec des solutions aléatoires.

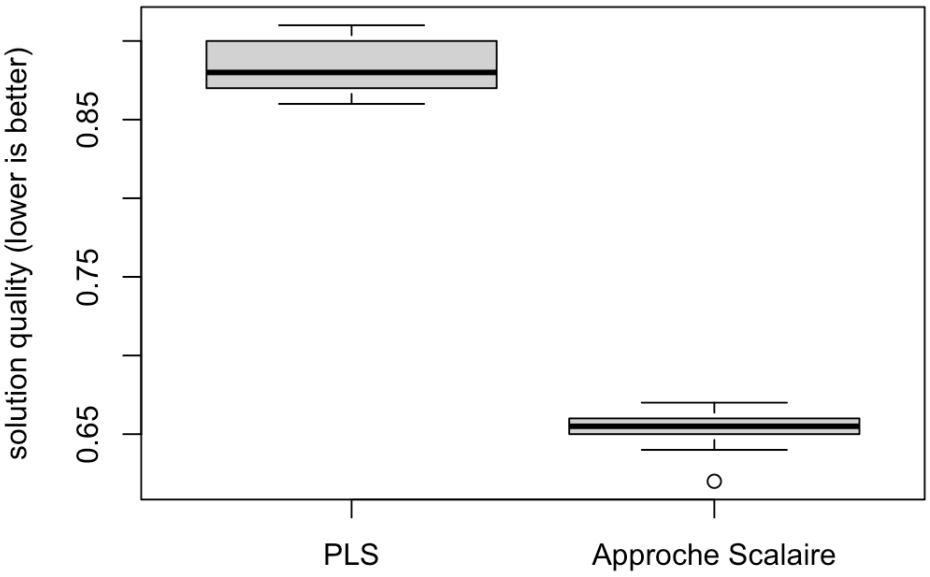


FIGURE 9 – Comparaison entre les deux algorithmes.

On peut alors constater que l'approche scalaire implementée est plus performante que la Recherche Locale Pareto.

# 6 Conclusion

Nous avons donc étudié deux types d'algorithmes possibles afin de résoudre le problème multiobjectif TSP à savoir une approche scalaire et une approche Pareto. Après avoir étudié les différentes configurations possibles de l'approche scalaire nous avons vu que celle-ci était plus performante ( sous sa meilleure configuration ) que l'approche Pareto. Ce résultat pourrait provenir du manque de complexité provenant de l'approche Pareto. En effet celle-ci part de solutions aléatoires, et ne permet pas une grande diversification avec la génération de son voisinage. En utilisant des heuristiques pour créer les solutions initiales ( comme le 2-opt ) ou en utilisant une diversification plus prometteuse ( comme celle des algorithmes génétiques) l'approche Pareto pourrait être améliorée.