# PROBLÈME D'ASSIGNATION QUADRATIQUE

OPTIMISATION DISCRÈTE

SUARD Tancrède 11505293 THOMAS Dylan 11606991

## Table des matières

# I. Introduction

Le rapport présenté ici s'inscrit dans le cadre de l'unité d'enseignement « optimisation discrète ». Le thème abordé dans la suite de ce document est basé sur la résolution des problèmes d'une assignation quadratique. Le travail est réalisé en binôme.

Le problème d'assignation quadratique est un problème d'optimisation combinatoire introduit par Koopmans et Beckmann en 1957. C'est un problème d'ordre NP-complet. Il est considéré comme l'un des problèmes les plus difficiles à résoudre de manière optimale. Le problème est défini dans le contexte suivant : on a un ensemble $A$ d'éléments de taille $n$ qui doivent être positionnées sur $n$ emplacements. La distance entre chaque emplacement est définie par $D_{i,j}$ avec $i, j \in A$ . De même, on définit un poids, ou bien un coût, pour relier chacun des éléments, représenté par une matrice $F_{i,j}$ avec $i, j \in A$.

Le problème consiste à trouver la bonne assignation de chacun des éléments par rapport à l'emplacement de tels buts à minimiser la fonction de coût suivante :

$$C = \sum_{i=1}^{n} \sum_{j=1}^{n} F_{i,j}\, D_{p(i),p(j)}$$

Où $p(i)$ représente la position associée à chacun des éléments $i$. Pour pouvoir stocker toutes ces informations, on utilise des matrices de taille $n$ pour pouvoir stocker les distances $D_{i,j}$ et les coûts $F_{i,j}$ respectivement de cette formule.

La résolution de ce type de problème est assez complexe, et on retrouve des solutions optimales que sur les petits ensembles de tailles $n \leq 36$. En règle générale, on utilise des approches heuristiques développées pour ce type de problème et applications des cas de taille inférieure à 100.

# II. Ressources et ensembles de données

L'étude réalisée ici sera effectuée sur un ensemble de Tai..a uniformément généré et proposé en Taillard:91[1]

Les ensembles utilisés sont de taille variable entre 12 et 100. Ils sont formatés de manière textuelle et brute, avec en première ligne la taille du nombre d'éléments suivi de la matrice des poids, suivie de la matrice des distances.

Le langage choisi pour travailler sur ces données et Java 8.

Les fichiers de données en sortie sont en format CSV pour pouvoir être exploités sur Excel et Tableau.

Le code implémenté tire profit des calculs multi-Thread.

[1] [Taillard:91]
E.D. TAILLARD. Robust tabu search for the quadratic assingnment problem. Parallel Computing, 17:443-455, 1991.

# III. Nomenclature

Avant de réaliser la partie étude des algorithmes sur les différentes instances de TAILLARD. Il convient de spécifier les différents termes utilisés dans la suite du rapport.

## 1. Landscape

Le terme Landscape est utilisé pour définir les données d'entrée. Elle regroupe la taille des matrices, elle y a différentes matrices de poids et de coûts. Dans l'implémentation, le Landscape est construit à partir d'un Parser qui s'occupe de lire les fichiers sources. Lorsque les algorithmes sont utilisés, le Landscape est passé en paramètre pour appliquer les opérations dessus.

## 2. Solution - Order

On désigne solution (Order dans l'implémentation) l'ordre de placement de chacun des éléments par rapport au Landscape d'entrée. Ainsi il y a autant de solutions que de façon possible d'affecter un élément sur chaque emplacement. Une solution est correcte si elle respecte la taille de la Landscape et qu'il apparaît une seule et unique fois chaque élément à l'intérieur. Pour un Landscape de taille $n$, il y a $n!$ Solutions. Soit dans le cas de Tai100a de taille $n = 100$, un peu moins de $10^{158}$ solutions.

## 3. Fitness

La fitness est le terme qui désigne le résultat de la fonction objectif spécifiée plus haut. Chaque fitness est calculé à partir d'une solution sur le Landscape donné. Le but des algorithmes qui suivent dans ce rapport est de trouver la meilleure solution qui minimise la fitness.

## 4. Opération

Pour calculer et définir les différentes solutions, il faut au préalable définir des opérations sur ces ensembles. Une opération consiste à modifier une solution pour en obtenir une différente. Dans la suite du rapport, on utilisera principalement la permutation qui échange de place deux éléments dans une solution. On notera que l'opération inverse de la permutation est la permutation elle-même.

## 5. Voisinage

On définit « voisinage » l'ensemble des solutions qui sont issues de différentes opérations à partir d'une solution source. Le nombre de voisins associés à une solution dépend donc des opérations que l'on effectue sur cette même solution. Dans l'implémentation détaillée dans ce rapport, la génération du voisinage est définie par des méthodes mapping.

## 6. Mapping

Le terme mapping définit ici le lien entre les solutions, le voisinage, et des opérations. Le principe consiste à définir des méthodes qui sont appliquées à une solution, suivant différentes opérations pour générer une liste de solutions, qui constituera le voisinage à cette solution d'entrée.

## 7. Algorithme

« Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre une classe de problèmes »

Wikipedia, 2019

Pour résoudre le problème donné, différents algorithmes sont utilisés dans cette implémentation. Ils implémentent tous les mêmes structures ce qui permet d'avoir une meilleure souplesse dans l'implémentation du code, et de respecter des normes d'entrée/sortie quant à l'exécution de code.

## 8. Benchmark

On définit par le terme « benchmark » une série de tests comprenant plusieurs paramètres d'entrée pour pouvoir réaliser une étude approfondie de l'utilisation des algorithmes sur les problèmes donnés. L'utilisation de benchmarks permet l'automatisation des algorithmes et ainsi de pouvoir générer énormément de résultats dans le but de pouvoir les comparer. Il est ainsi possible de définir des benchmarks avec différents algorithmes, différents paramètres, sur des Landscape différents.

Dans la suite de ce rapport, ce sera sur ces benchmarks que seront basés les différents graphiques utilisés et interprétations.

# IV. Algorithmes utilisés

Pour répondre aux problèmes d'assignation quadratique, il existe déjà de nombreux algorithmes qui ont fait leurs preuves dans ce type de problème. Quatre algorithmes sont utilisés et implémentés dans le cadre de ce travail : la marche aléatoire, la méthode Hill-Climbing, le recuit simulé et la méthode Tabou.

## 1. La marche aléatoire

Dans un premier temps, l'algorithme de marche aléatoire est implémenté dans le but de trouver naïvement une solution en itérant énormément de fois de manière aléatoire sur l'ensemble des Landscapes et de calculer la meilleure fitness. L'algorithme prend donc uniquement un nombre maximal d'itérations comme paramètre.

L'algorithme commence par tirer aléatoirement une solution et définit cette solution comme étant la meilleure et conserve sa fitness. Puis on tire aléatoirement un seul voisin de cette solution, s'il a une meilleure fitness que celle trouvée précédemment, alors on le définit comme étant la meilleure solution et on conserve sa fitness. Quelle que soit sa fitness, la solution est considérée comme étant la nouvelle solution actuelle et l'algorithme est réitéré à partir de cette solution. La boucle d'itérations s'arrête lorsque le nombre maximal d'itérations défini est atteint.

L'algorithme a une complexité linéaire (dépendant de la taille du Landscape) étant donné qu'il effectue très exactement le nombre d'itérations qui lui est demandé. Il n'y a aucune recherche de meilleurs voisins ni d'historique des voisins parcourus.

## 2. La méthode Hill-Climbing

La méthode Hill-Climbing consiste à trouver un minimum local. L'algorithme implémenté ici tire au hasard une solution dans le Landscape, puis itère sur les voisins de cette solution pour trouver lequel a la meilleure fitness par rapport à la solution actuelle. Le meilleur voisin est considéré comme étant la nouvelle solution sur laquelle itérer , puis lorsqu'aucun voisin n'améliore la fitness actuellement trouvée, on définit cette solution comme étant un minimum local. On compare la fitness de ce minimum local avec la fitness du meilleur minimum local trouvé sur l'ensemble de l'algorithme. L'algorithme prend en entrée un nombre maximal d'itérations. Une fois que le nombre maximal d'itérations est atteint, on considère comme solution du Landscape, le meilleur minimum local trouvé.

On notera qu'une légère variante a été implémentée également, car pour les grandes dimensions de TAILLARD, le nombre de voisins à une solution est extrêmement grand et comparer chacune de leur fitness prend du temps, tout comme le fait d'attendre de trouver un minimum local pour arrêter une itération de l'algorithme. En effet comme la taille du Landscape est très grande, la complexité de l'algorithme explose. Il a donc été rajouté un paramètre qui permet de définir le nombre de pourcentages de voisins que l'on souhaite tester avant de considérer une prochaine itération. Cela permet donc de tester par exemple 5 % des voisins de chaque solution.

## 3. La méthode du recuit simulé

Le recuit simulé est une méthode inspirée de la thermodynamique qui consiste à trouver un meilleur arrangement en s'autorisant de quelquefois dégrader la solution pour sortir des minimums locaux. Contrairement à la méthode Hill-Climbing définit juste au-dessus, grâce à un paramètre fictif : la température, on s'autorise occasionnellement à effectuer de l'exploration sur de mauvais voisins dans le but de sortir rapidement d'un minimum local pour en trouver un autre potentiellement meilleur. L'idée consiste à fixer une température initiale suffisamment grande pour s'autoriser à dégrader la solution trouvée dès les premières itérations, puis de réduire la température par un facteur $\mu$ avec $0 < \mu < 1$. Ainsi au fur et à mesure que la température diminue, l'algorithme tend à trouver le meilleur minimum local sur la zone testée.

L'algorithme prend donc en paramètres une solution initiale, une température initiale, et un nombre de mouvements sur les voisins réalisables avant de procéder à la prochaine itération sur la température.

L'implémentation de l'algorithme est basée sur celui étudié en cours. Toutefois, la variation de $\mu$ dans le temps n'a pas été prise en compte. $\mu$ est donc constant et la température diminue de manière géométrique tel que $t_n = \mu^n t_0$ avec $t_0$ la température initiale.

Différentes variations seront traitées pas la suite pour étudier les paramètres du recuit simulé.

## 4. La méthode Tabou

La méthode Tabou appliquée à ce problème d'assignation quadratique consiste ici à réaliser plusieurs tâches. Premièrement, on détermine par rapport au voisinage quel voisin minimise la fitness et pour chacun des voisins calculés on garde en mémoire l'opération qui est effectuée. Suivant certaines conditions, si une opération qui a été réalisée n'améliore pas la fitness, c'est-à-dire une opération qui augmente la fitness (un mauvais voisin donc), alors on décide d'effectuer le mouvement, pour sortir probablement d'un minimum local. Pour éviter de redescendre dans ce minimum local, on interdit l'opération inverse qui est associée à la création de ce voisin. Cette opération est ajoutée à l'intérieur d'une liste que l'on nomme la liste Tabou. Pour chaque itération supplémentaire sur le voisinage, on éliminera les voisins qui sont issus de cette opération. De plus, on limite la taille de la liste Tabou avec un certain nombre d'opérations. Dans une implémentation simple de la méthode Tabou, les opérations sont mises dans l'ordre d'ajout, et lorsqu'il faut ajouter une nouvelle opération qui dégrade une solution, et que la liste est remplie, on enlève le tout premier qui avait été ajouté (liste est considérée circulaire). Les deux principaux paramètres de la méthode Tabou sont le nombre d'itérations où l'algorithme est exécuté et la taille de la liste Tabou.

# V. Application des algorithmes sur les instances de Taillard.

Les algorithmes sont implémentés en Java, et exécutés sur les instances de Taillard. Comme décrit plus haut dans la nomenclature, toute une série de benchmarks a été réalisée pour pouvoir exécuter les algorithmes en faisant varier les paramètres de chacun d'entre eux. Les données sont extraites au format CSV. Toutefois pour maximiser le temps de calcul, les benchmarks sont exécutés en parallèle suivant un nombre défini de thread. Les données qui sont produites dans les fichiers CSV ne sont pas ordonnées. Il faut donc au préalable avant de pouvoir les traiter, les réordonner à l'aide de la fonction tri sous Excel.

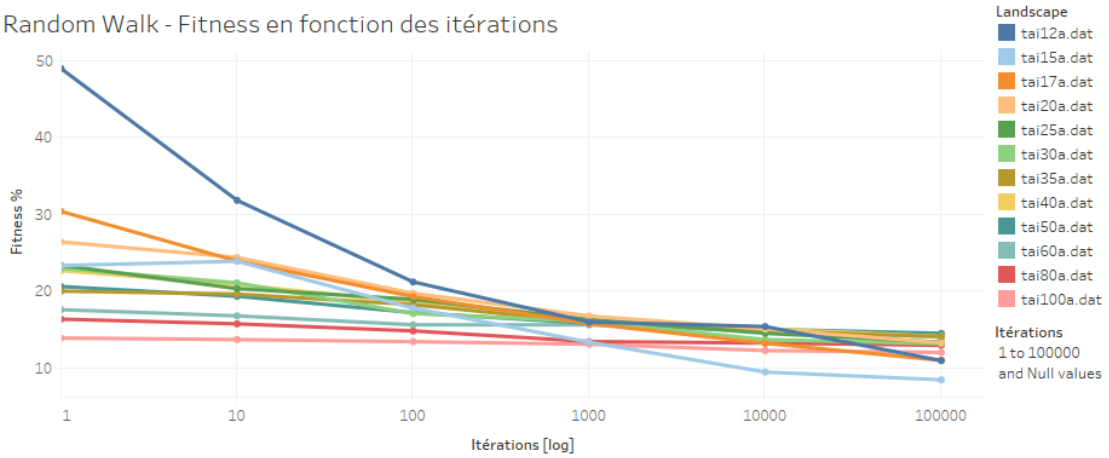
Étant donné le grand nombre d'informations qui en ressortent, et en fonction de grand nombre de paramètres qui varient, les représentations graphiques qui seront présentes dans la suite de ce rapport sont extraites d'une analyse réalisée à partir du logiciel Tableau. Les sources de ces analyses graphiques utilisables sous Tableau, seront mises en annexe ce document.

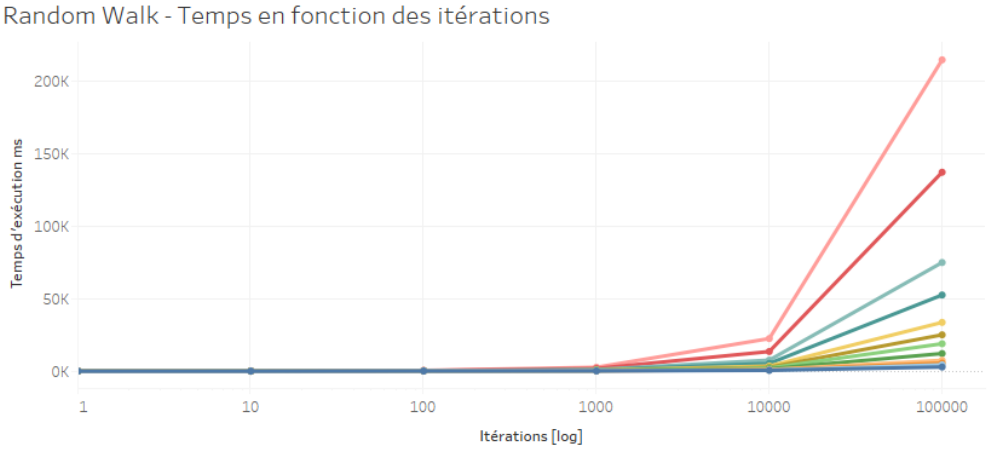
Dans le restant de l'analyse des algorithmes, deux critères d'évaluation vont être confrontés et mis en relation. Le premier concerne les résultats de la fitness sur les algorithmes en fonction de différents paramètres d'entrée. Le second concerne le temps d'exécution des algorithmes sur ces mêmes paramètres d'entrée. La principale analyse sera donc faite, non pas exclusivement sur la qualité de la fitness trouvé ou de la solution, mais également sur la complexité en temps qu'il faut pour arriver à atteindre ces solutions. Il sera donc question d'argumenter sur le fait de savoir s'il faut vraiment attendre très longtemps pour améliorer la fitness ou si l'on peut se contenter, suivant certains algorithmes et suivant certains paramètres, d'obtenir rapidement une estimation de la meilleure fitness trouvable.

Pour affiner les résultats de chacun des algorithmes, absolument tous les tests pour chacun des paramètres et pour chacun des algorithmes ont été réalisés 4 fois puis une moyenne du résultat des fitness du résultat du temps d'exécution a été récupérée pour former les graphiques suivants. C'est notamment pour ça qu'il y a rarement des fitness extrêmement basse pour les petites instances de Taillard, étant donné qu'il faut générer quatre fois d'affilée une toute petite fitness pour que la moyenne de ses fitness là soit basse. Le fait de réaliser quatre fois chacun des algorithmes permet de réduire considérablement un effet de brouillage et de désordre qui serait défini par certains tirages aléatoires notamment sur les voisins. Ainsi il en convient que les résultats sont plus stables et plus justes autant dans le temps que dans les résultats de la fitness.

## 1. La marche aléatoire

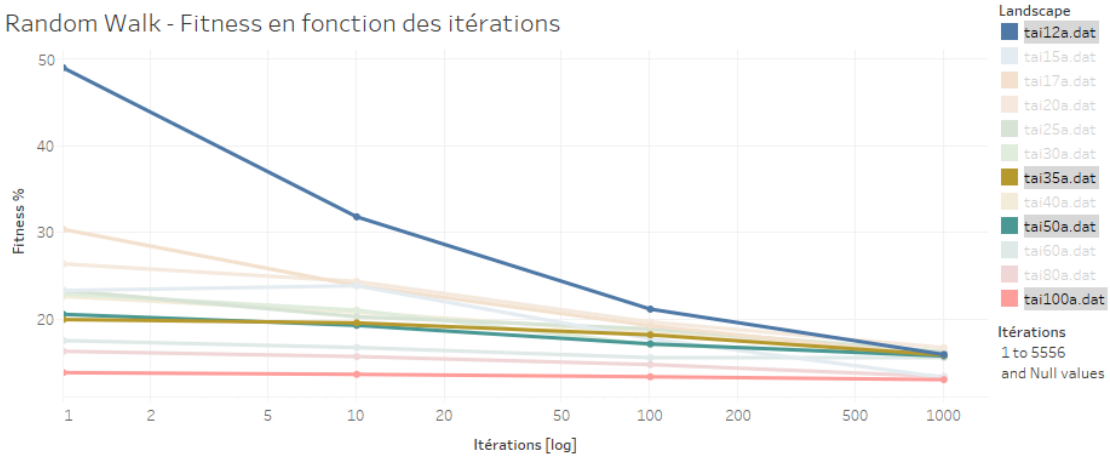
Pour tester une approche simple et naïve de la résolution du problème, l'algorithme de la marche aléatoire a été utilisé sur toutes les instances de Taillard. L'avantage de l'algorithme de la marche aléatoire c'est qu'il est extrêmement rapide à exécuter, en effet il n'y a aucune comparaison des fitness entre les voisins, pour chaque itération un voisin est élu et on vérifie s'il satisfait une fitness minimale sur l'ensemble de tous ceux qui ont été testés. Cela permet donc d'effectuer de très grande itération.

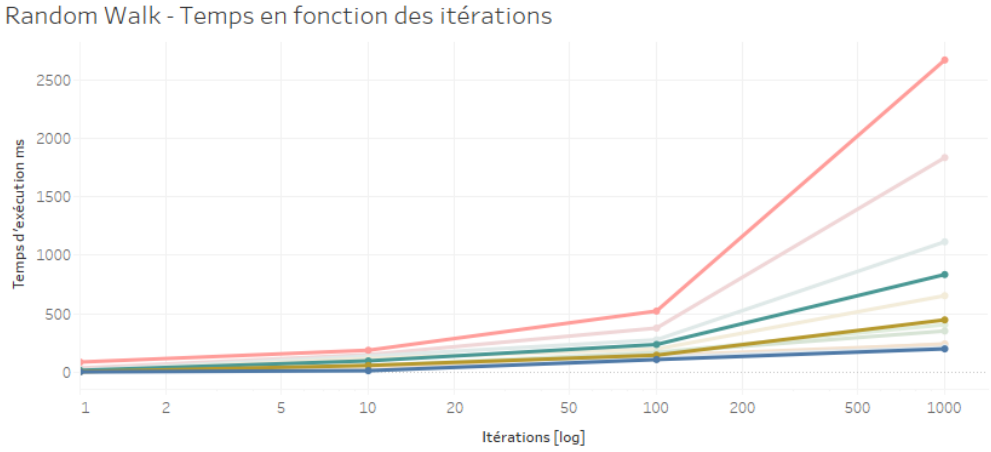



L'échelle des abscisses utilisées ici est en $\log(10)$. On remarque donc que très rapidement on trouve une bonne fitness à partir d'environ 100 itérations. De plus lorsque l'on dépasse 1000 itérations, la complexité le temps d'exécution explose. Sur l'ensemble des instances de Taillard utilisées ici, il convient d'estimer que 100 itérations ou à la rigueur 1000 est un bon nombre pour pouvoir s'assurer d'avoir une bonne fitness par rapport à l'algorithme.

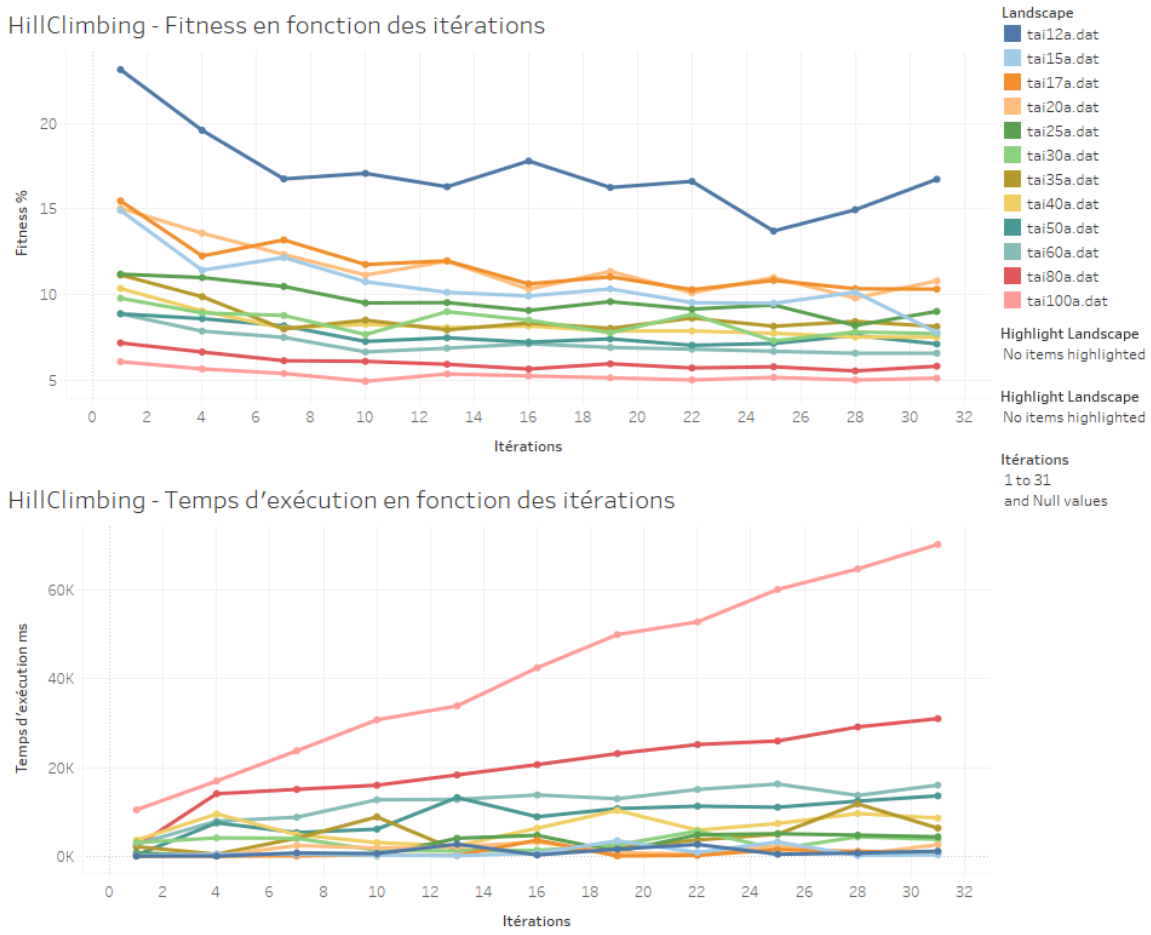
On remarque qu'une bonne fitness est trouvée dès lors qu'on itère autant de fois que la taille au carré de l'instance de Taillard. Avec un tel raisonnement, on effectue énormément de mouvement dans le Landscape, mais le procédé aléatoire tend à ce que toutes les solutions soient testées à force d'itérer.


Random Walk - Fitness en fonction des itérations
Fitness %
50
40
30
20
1
2
5
10
20
50
100
200
500
1000
Itérations [log]
Landscape
tai12a.dat
tai15a.dat
tai17a.dat
tai20a.dat
tai25a.dat
tai30a.dat
tai35a.dat
tai40a.dat
tai50a.dat
tai60a.dat
tai80a.dat
tai100a.dat
Itérations
1 to 5556
and Null values

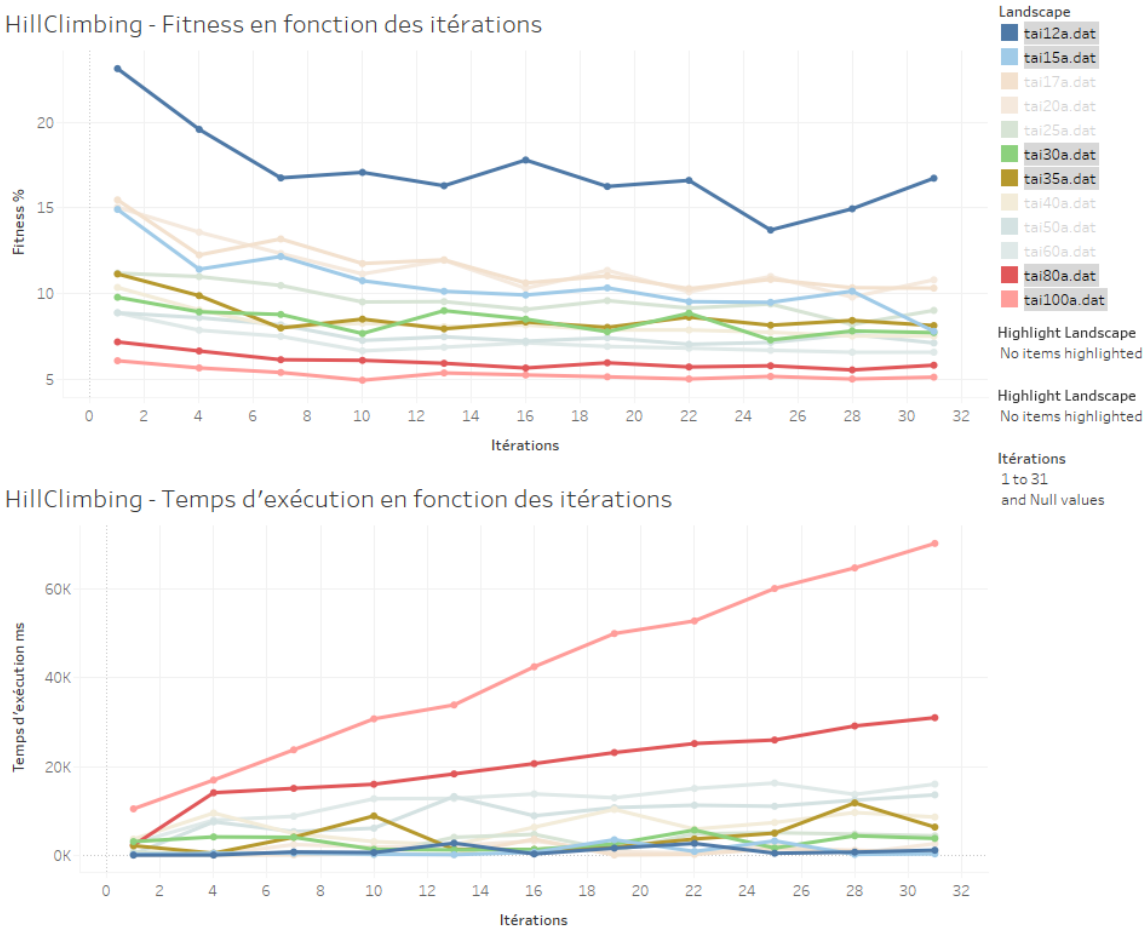

Random Walk - Temps en fonction des itérations
Temps d'exécution ms
2500
2000
1500
1000
500
0
1
2
5
10
20
50
100
200
500
1000
Itérations [log]

## 2. La méthode Hill-Climbing

La méthode Hill-Climbing utilisée ici a énormément moins d'itérations. En effet, les itérations illustrées ici représentent le nombre de fois qu'on débute algorithmes sur l'instance de Taillard. En effet pour chaque itération, on parcourt de manière quasiment illimitée les voisins jusqu'à trouver le minimum local. Il n'est donc pas nécessaire d'effectuer des milliers d'itérations. De plus, le temps qu'il faut pour trouver un minimum local va dépendre, finalement, de la solution initiale.




On remarque notamment avec les petites instances de Taillard que la fitness n'est pas très stable, étant donné qu'on arrive très rapidement à un minimum local, mais qu'à l'itération d'après, on peut trouver très rapidement un autre minimum local qui améliore potentiellement la fitness. À l'inverse, pour les instances de taille plus grande, on arrive assez rapidement à une très bonne fitness par rapport à la marche aléatoire que l'on a constatée au-dessus, mais avec un temps d'exécution beaucoup plus grand. En effet la qualité de la fitness est presque divisée par quatre, pour un temps d'exécution qui va être multiplié quasiment par 20. Étant donné que le temps d'exécution de cet algorithme reste quand même limité, largement inférieur à la seconde pour tous ces ensembles il en convient que la méthode Hill-Climbing semble une méthode clairement plus efficace que la marche aléatoire.


HillClimbing - Fitness en fonction des itérations
Fitness %
20
15
10
5
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Itérations
Landscape
tai12a.dat
tai15a.dat
tai17a.dat
tai20a.dat
tai25a.dat
tai30a.dat
tai35a.dat
tai40a.dat
tai50a.dat
tai60a.dat
tai80a.dat
tai100a.dat
Highlight Landscape
No items highlighted
Highlight Landscape
No items highlighted
Itérations
1 to 31
and Null values
HillClimbing - Temps d'exécution en fonction des itérations
Temps d'exécution ms
60K
40K
20K
0K
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Itérations

## 3. La méthode du recuit simulé

La méthode du recuit simulé prend plusieurs paramètres possibles en entrée. C'est pourquoi dans l'étude qui suit, certains paramètres vont être fixés et d'autres vont pouvoir varier pour éviter de générer des graphes qui soient complètement illisibles. Les paramètres de la méthode de recuit simulé fixé ici proviennent de l'algorithme fourni en cours.

On rappelle ici les paramètres utilisés :

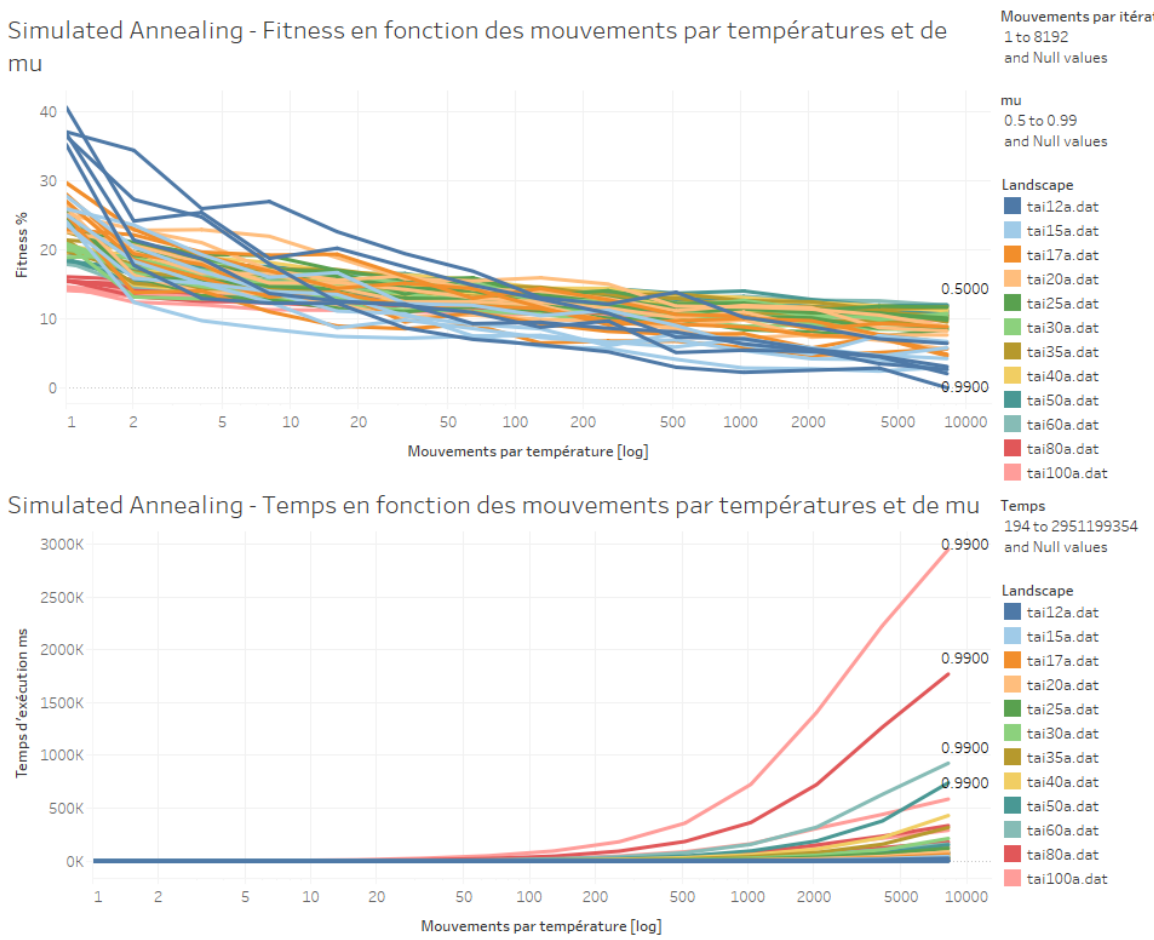
- la **variable $\mu$** qui fait varier la température à chaque itération : ici elle est variable telle que $\mu \in \{0.5; 0.8; 0.9; 0.95; 0.99\}$
- Le **nombre de mouvements par itérations de température $n_2$**.
- La **probabilité d'accepter de « moins bonnes solutions »** : ici, fixée à $\mathbf{0.8}$ comme dans le cours.
- La **probabilité d'accepter la même mauvaise solution que pour fixer la température initiale**. Fixée ici à **1%.**
- **La température initiale $t_0$** n'est pas passée en paramètres elle est calculée de telle sorte de pouvoir accepter de mauvaises solutions dès la première itération. Par la formule suivante :

Soit $\Delta f$ la pire différence de fitness des voisins de la solution initiale avec la fitness de la solution initiale.

$$t_0 = \frac{-\Delta \mathrm{f}}{\ln(0.8)}$$

- Le **nombre de changements de température $n_1$** tel que

$$n_1 = \frac{\ln(\frac{-\Delta f}{t_0 \ln(0.01)})}{\ln(\mu)}$$

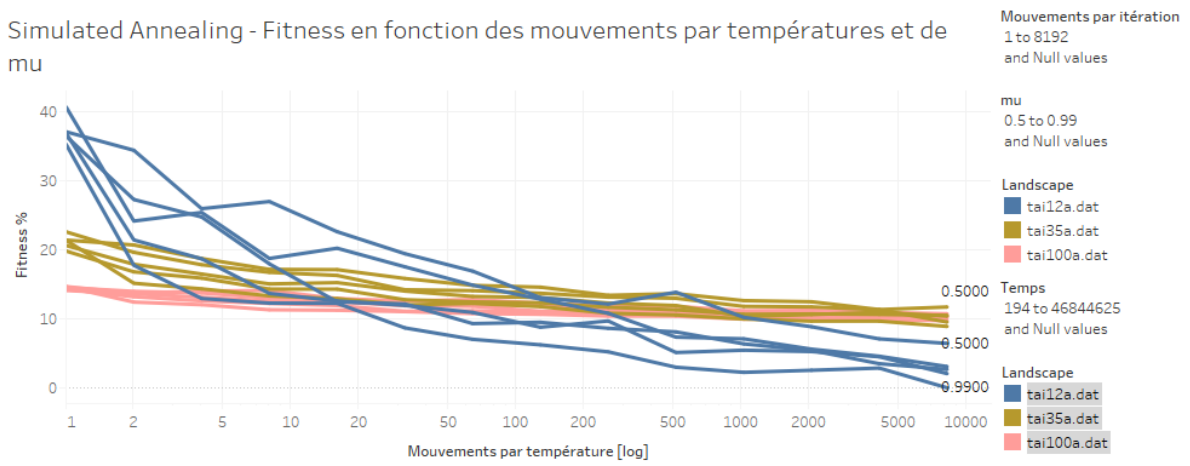


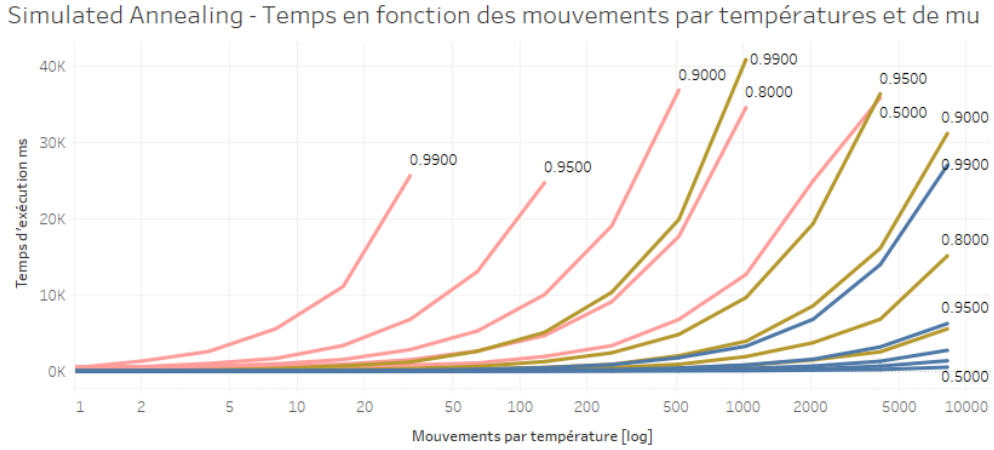

Ce graphique illustre les résultats en faisant varier le Landscape, le nombre de mouvements par température en $\log(10)$ en fonction des différents $\mu$.

On remarque premièrement que plus le nombre de mouvements de température est grand, plus les fitness solutions de l'algorithme convergent vers la bonne solution. On notera que dans le cas de Tai12a, la valeur de fitness en pourcentage atteint 0. Ce qui signifie que pour 10000 mouvements de température est un $\mu$ très proche de 1, sur 4 tests de l'algorithme, le recuit simulé a trouvé la solution optimale.

Quelque soit l'instance de Taillard utilisée, globalement, et quelque soit la valeur de $\mu$ la fitness convergent vers la fitness optimale plus le nombre de mouvements par température augmente.

Il est important de prendre en compte les temps pour les Taillard de taille plus grande que 35. En effet, le temps atteint facilement le million de milli secondes, c'est-à-dire plusieurs minutes. Dans le cas de Tai100a. L'algorithme pour calculer 8192 mouvements par température met environ 3M de milli secondes, soit 50 minutes. Étant donné que c'est une moyenne sur 4 tests de l'algorithme, faire ce rendu à pris un peu moins de 4h uniquement pour le dernier point de Tai100a. D'où l'importance de la gestion multithread précisée en introduction.



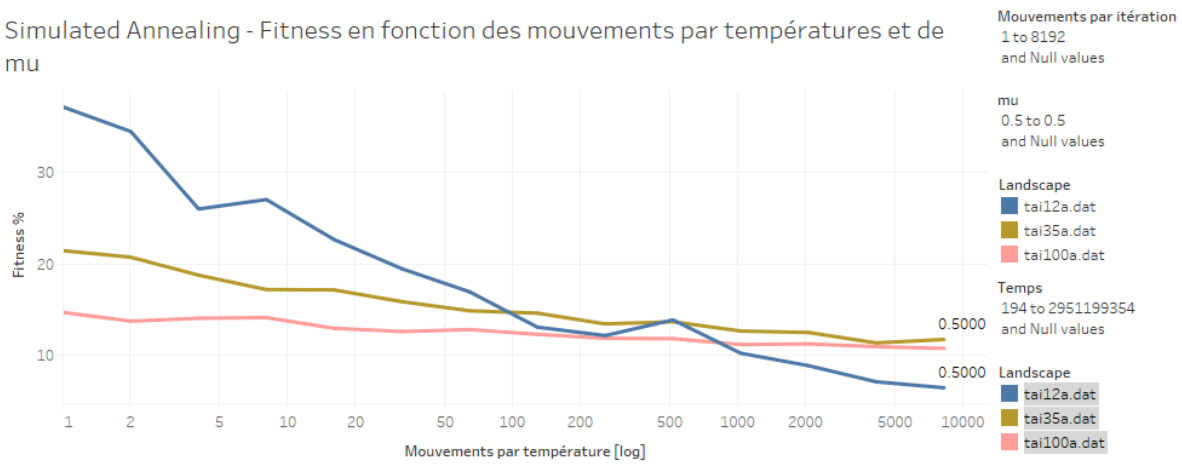

Étant donné les premières analyses du graphique global, voici une partie extraite réduite aux trois ensembles Tai12a, Tai35a et Tai100a. On peut donc comparer la performance de l'algorithme suivant un petit, moyen ou grand Landscape.

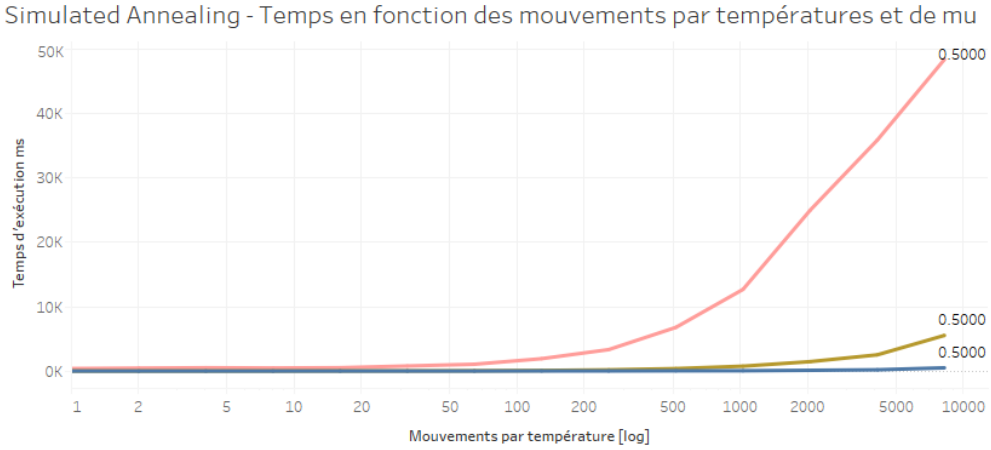
On reste sur une échelle logarithmique en $\log(10)$.

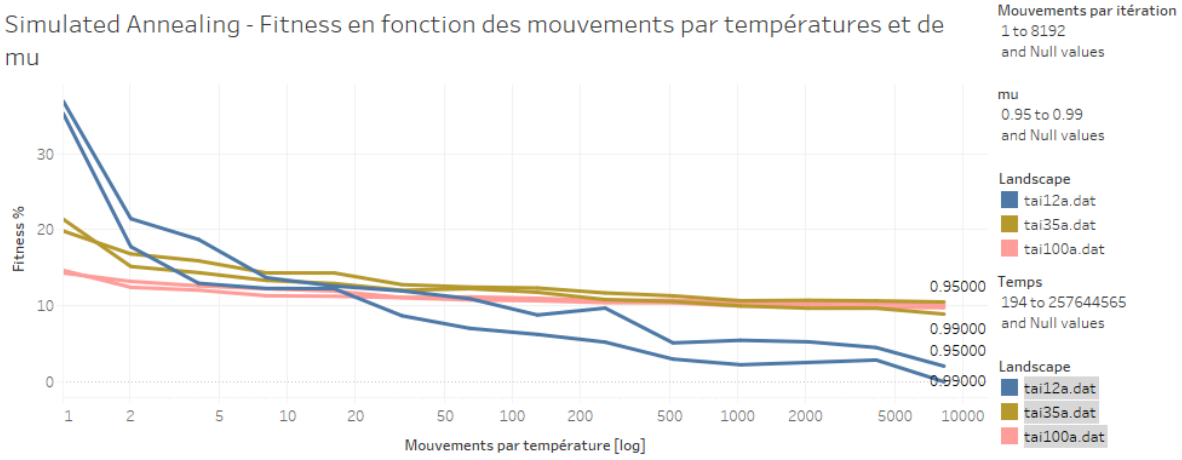
On remarque que les petites instances de Taillard produisent un résultat peu stable, mais qui converge. De plus, on remarque que plus $\mu$ tend vers 1 plus la meilleure fitness est trouvée vite.

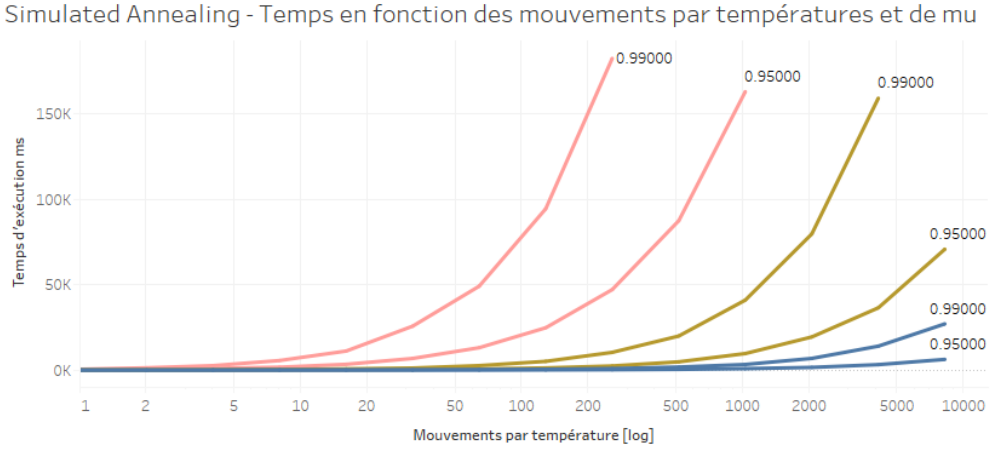
Pour Tai12a, à partir de 100 itérations, il devient difficile d'améliorer la fitness. Alors que pour Tai35a et Tai100a, seuls 2 à 8 mouvements suffisent pour tendre très vite vers une bonne solution de la fitness.

Quant à l'exécution, plus la taille du Landscape est grande, plus le temps d'exécution explose vite. On peut soit suggérer d'utiliser un nombre de mouvements élevé et un $\mu$ faible, soit utiliser un $\mu$ très proche de 0, mais réduire considérablement le nombre de mouvements par température. Sinon le temps d'exécution grandit trop vite.


Simulated Annealing - Fitness en fonction des mouvements par températures et de mu
Fitness %
30
20
10
1
2
5
10
20
50
100
200
500
1000
2000
5000
10000
Mouvements par température [log]
0.5000
0.5000
Mouvements par itération
1 to 8192
and Null values
mu
0.5 to 0.5
and Null values
Landscape
tai12a.dat
tai35a.dat
tai100a.dat
Temps
194 to 2951199354
and Null values
Landscape
tai12a.dat
tai35a.dat
tai100a.dat


Simulated Annealing - Temps en fonction des mouvements par températures et de mu
Temps d'exécution ms
50K
40K
30K
20K
10K
0K
1
2
5
10
20
50
100
200
500
1000
2000
5000
10000
Mouvements par température [log]
0.5000
0.5000
0.5000


Simulated Annealing - Fitness en fonction des mouvements par températures et de mu
Fitness %
30
20
10
0
1
2
5
10
20
50
100
200
500
1000
2000
5000
10000
Mouvements par température [log]
0.95000
0.99000
0.95000
0.99000
Mouvements par itération
1 to 8192
and Null values
mu
0.95 to 0.99
and Null values
Landscape
tai12a.dat
tai35a.dat
tai100a.dat
Temps
194 to 257644565
and Null values
Landscape
tai12a.dat
tai35a.dat
tai100a.dat


Simulated Annealing - Temps en fonction des mouvements par températures et de mu
Temps d'exécution ms
150K
100K
50K
0K
1
2
5
10
20
50
100
200
500
1000
2000
5000
10000
Mouvements par température [log]
0.99000
0.95000
0.99000
0.95000
0.99000
0.95000

Dans les deux graphiques précédents, on s'intéresse à l'influence de $\mu$.

Les valeurs de $\mu$ sont ici réduites à $\mu \in \{0.5\}$ dans le premier graphique et à $\mu \in \{0.95; 0.99\}$ dans le second.

On retrouve la convergence remarquée plus tôt. Toutefois on note que un $\mu \to 1$ fait tendre beaucoup plus vite la fitness vers celle optimale, mais augmente également significativement le temps de calcul. Les échelles de temps (ms) passent du simple au triple. Alors que les fitness ne sont pas trois fois mieux. Il faut alors se questionner si utiliser un $\mu$ plus petit sur un grand ensemble avec un nombre de mouvements par température élevée ne serait pas la méthode la plus efficace pour trouver une bonne fitness tout en restant dans un temps d'exécution raisonnable.

On rappelle que

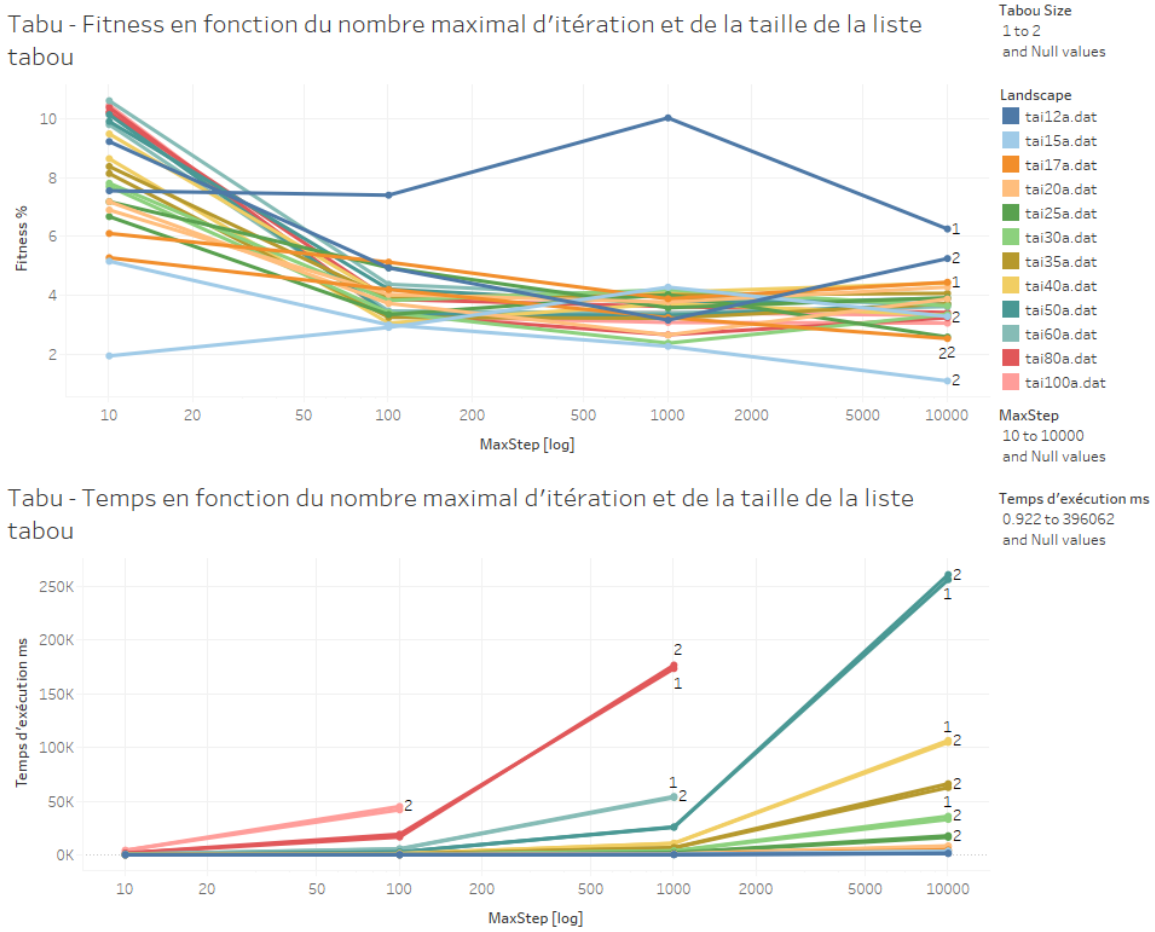
$$n_1 = \frac{\ln(\frac{-\Delta f}{t_0 \ln(0.01)})}{\ln(\mu)}$$

Et lorsque $\begin{cases}\mu \to 1 \\ \mu < 1\end{cases} \to \frac{1}{\ln(\mu)}$ augmente$\to n_1$ augmente.

Le nombre de changements de température est donc plus élevé avec un $\mu$ proche de 1 ce qui explique que la complexité en temps augmente significativement avec l'augmentation de $\mu$.

## 4. La méthode Tabou

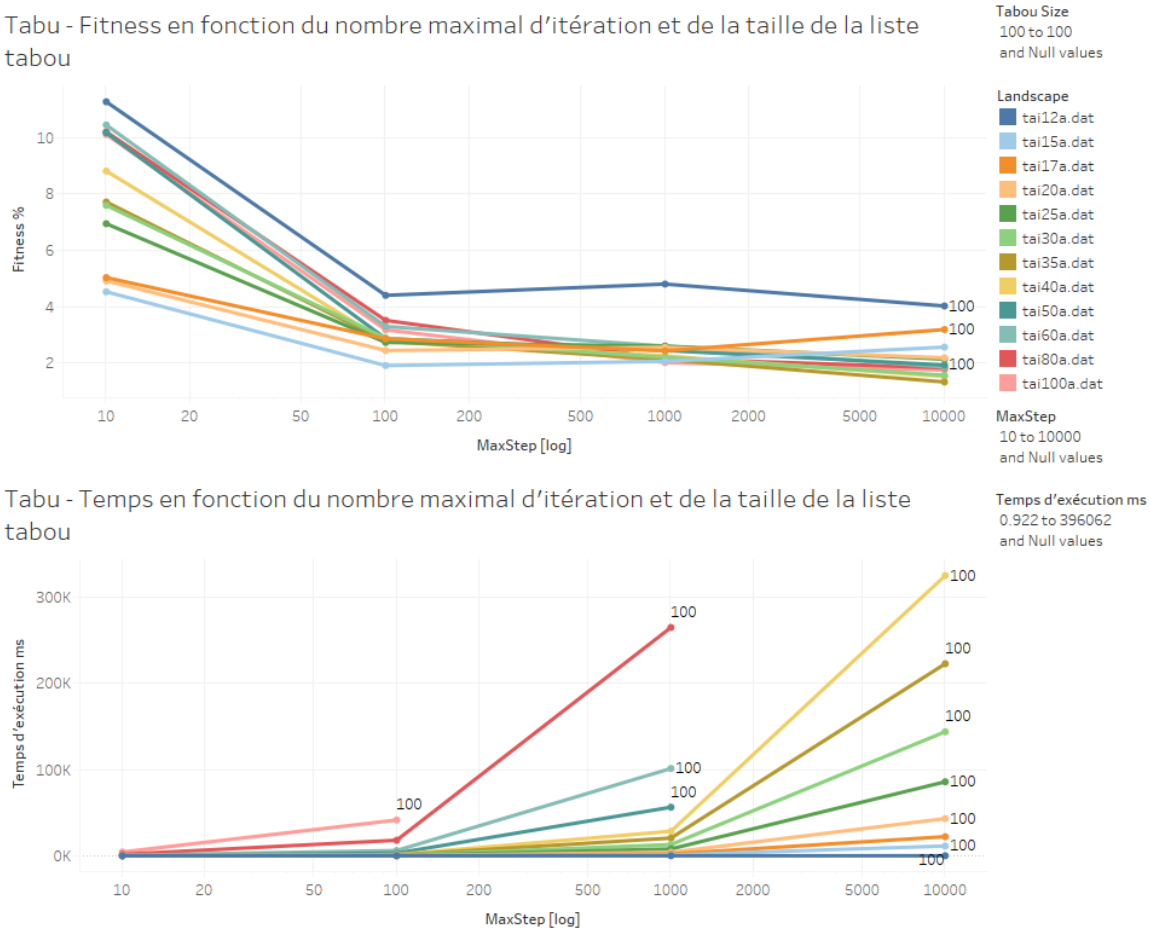
La méthode Tabou prend deux principaux paramètres, la taille de la liste Tabou et le nombre d'itérations de l'algorithme.




Le graphique représenté ici réduit l'algorithme avec uniquement des tailles de liste Tabou de 1 et 2 pour chacun des Landscape sur le nombre d'itérations maximum de l'algorithme en $\log(10)$.

On remarque que grossièrement une converge s'installe dès 100 itérations. De plus, d'exécution s'accroit très vite avec les différentes tailles de Landscape. Tout de même de manière linéaire entre la taille du Landscape et le nombre d'itérations.

Il est également à noter qu'on trouve de très bonne fitness en finalement pas beaucoup de temps comparé au recuit simulé. Pour 100 itérations avec une taille Tabou de 2, on obtient des fitness assez basses (3 à 6% de l'optimum) en moins de 50s. Alors qu'on est à 10% avec le recuit simulé pour le même temps d'exécution.




Ce graphique illustre les résultats avec une taille de liste Tabou de 100.

On a sensiblement les mêmes résultats, mais un temps d'exécution quand même plus grand. On notera que la fitness minimale trouvée tends dès 100 itérations, et que les 9900 itérations suivantes ne l'améliorent pas vraiment. On a quand même des résultats de l'ordre du 0.5-1% de fitness meilleur.

On remarque que plus la taille du Landscape est grande, mieux fonctionne l'algorithme, car il est moins sujet à interdire toutes les opérations réalisables. Les comparatifs en fonction des Landscape se trouvent en Annexe.

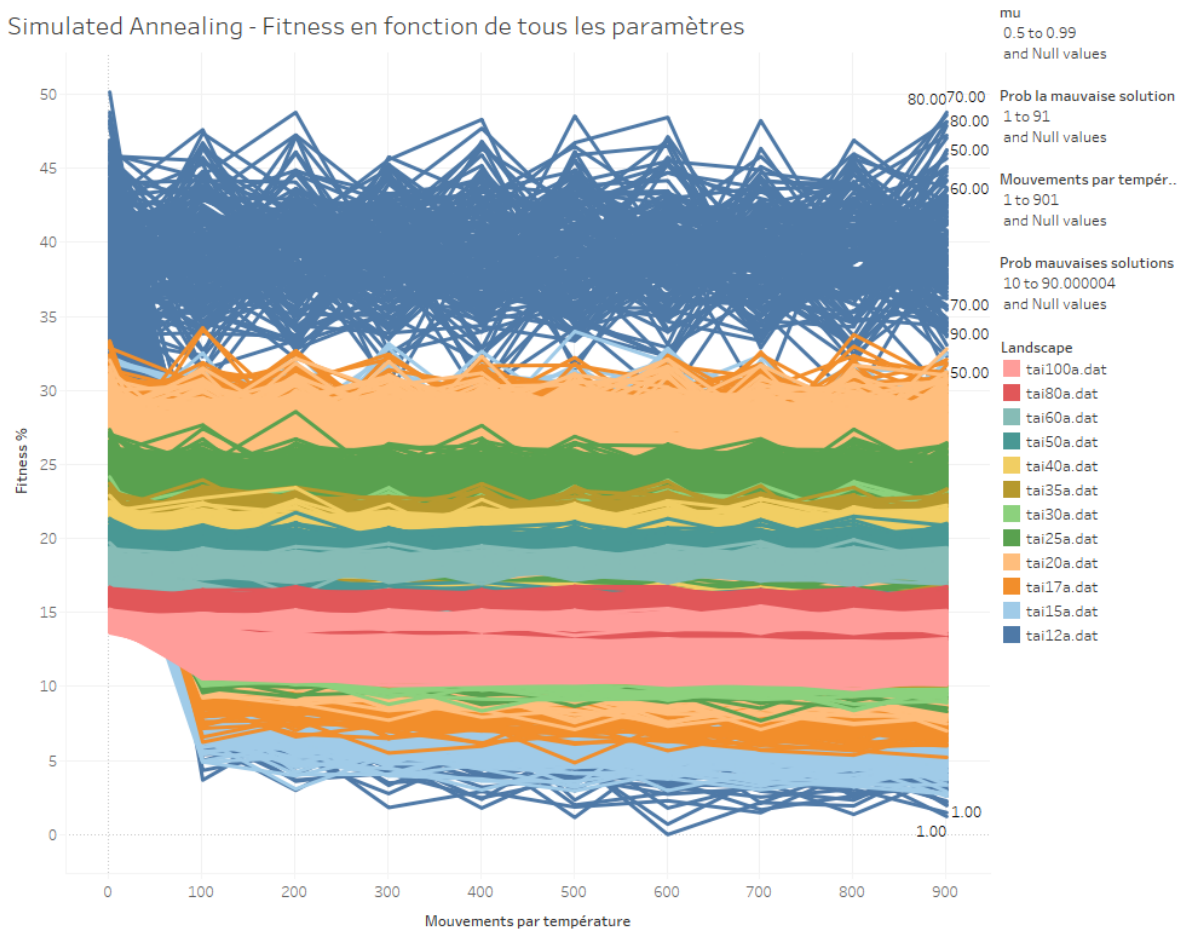
## 5. Travail complémentaire

Jusqu'à présent nous avons étudié le recuit simulé avec des valeurs fixées, récupérées des exemples de cours.

On s'intéresse alors à l'influence de ces valeurs par rapport aux résultats étudiés.

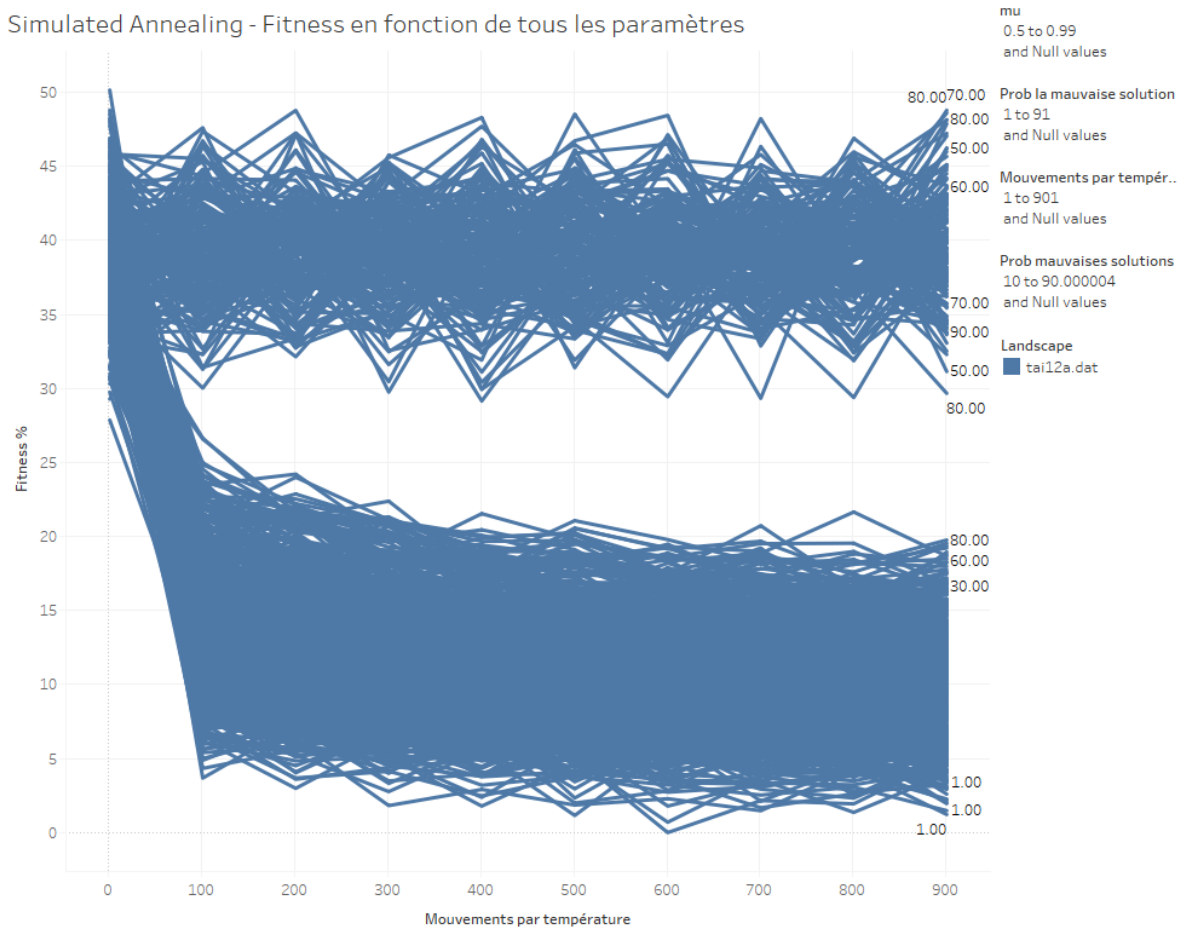
De nouveaux tests ont été réalisés cette fois-ci en faisant varier la probabilité de prendre la mauvaise solution, et la probabilité de considérer les mauvaises solutions. Respectivement fixé avant à 1% et 80%.

Voici la représentation brute des données :



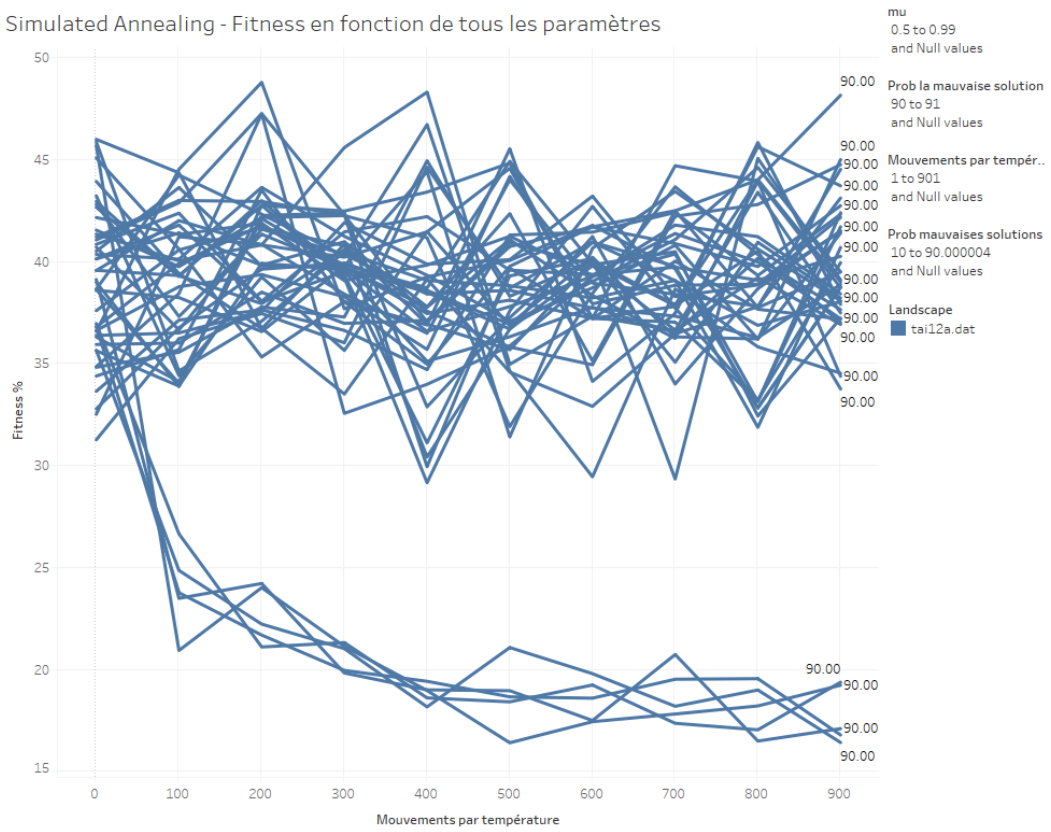

Cette fois-ci les instances de Taillard plus grande sont positionnées en priorité.On remarque qu'il y a une similarité en échelle avec des résultats qui convergent vers la solution optimale et d'autres résultats qui restent mauvais, quel que soit le nombre de mouvements par température.

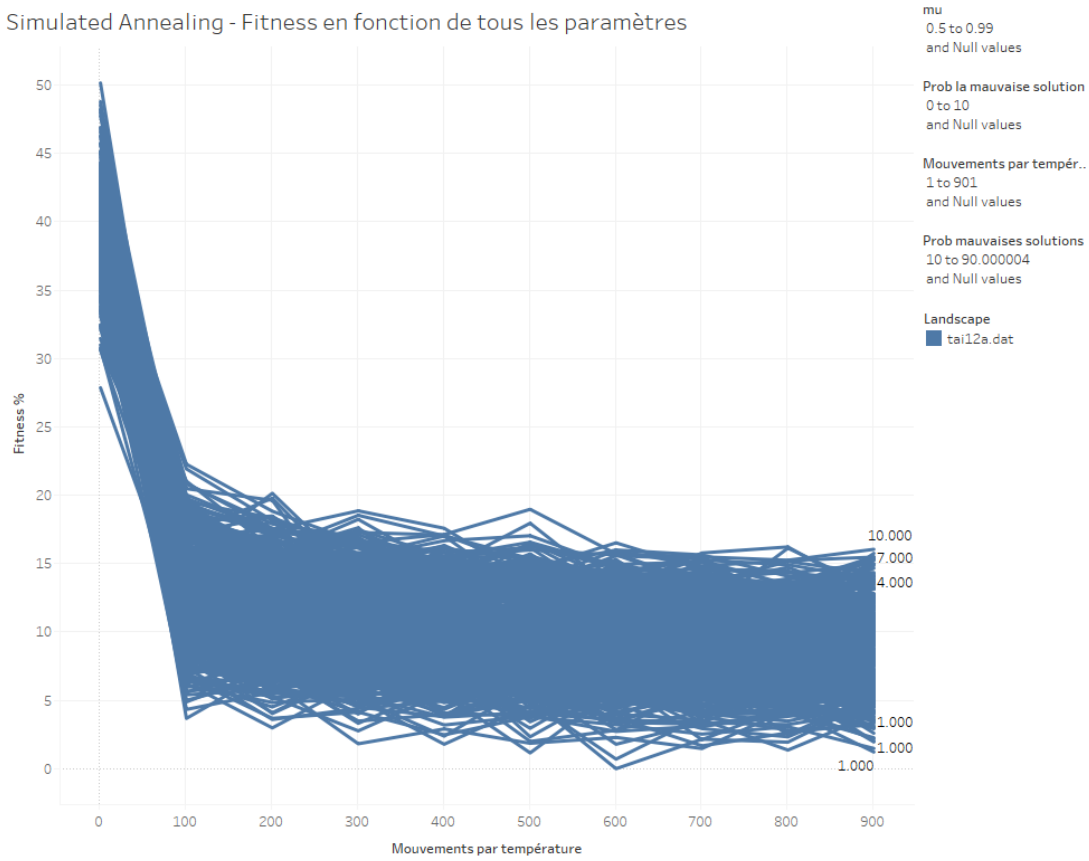
Maintenant on essaie d'analyser pour une instance de Taillard où l'on voit clairement la séparation entre les données qui font converger l'algorithme vers la solution optimale et les données qui ne fonctionnent pas.




On remarque bien la séparation des valeurs qui convergent et celles qui restent vers 40% d'erreurs.

On s'intéresse à quelles valeurs l'algorithme ne converge plus.



On remarque qu'à partir du moment où l'on prend qu'à 10% de chance (ou inférieur) de prendre une mauvaise solution, l'algorithme converge. Au-dessus de 10%, certains résultats d'algorithme ne convergent plus, puis au-dessus de 90% de chance de prendre la mauvaise solution, l'algorithme ne converge plus.

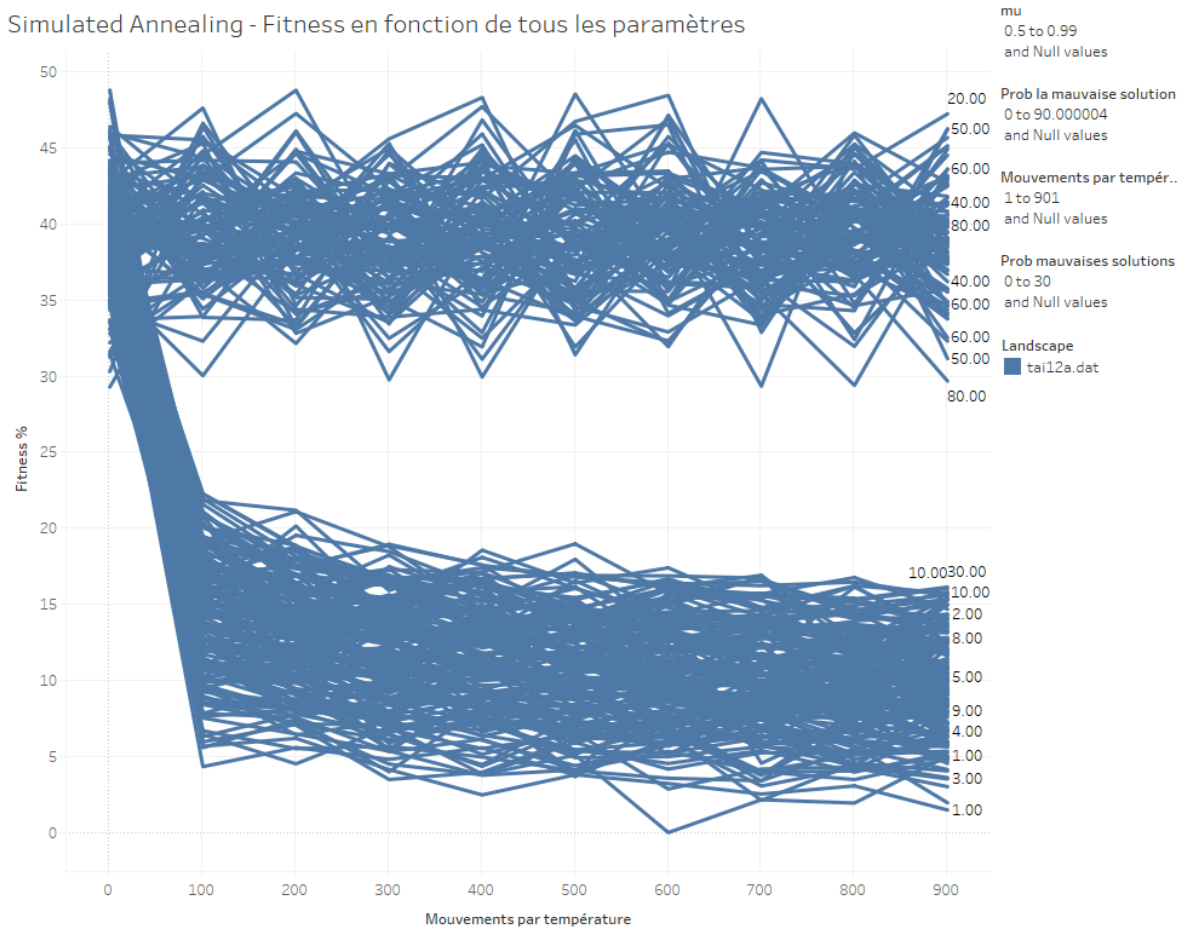
On pourra donc conclure que l'algorithme du recuit simulé ne converge pas forcément et que grâce au cours qui nous recommandait de prendre 1%, notre algorithme est dans un cas de convergence, quelles que soient les autres valeurs.

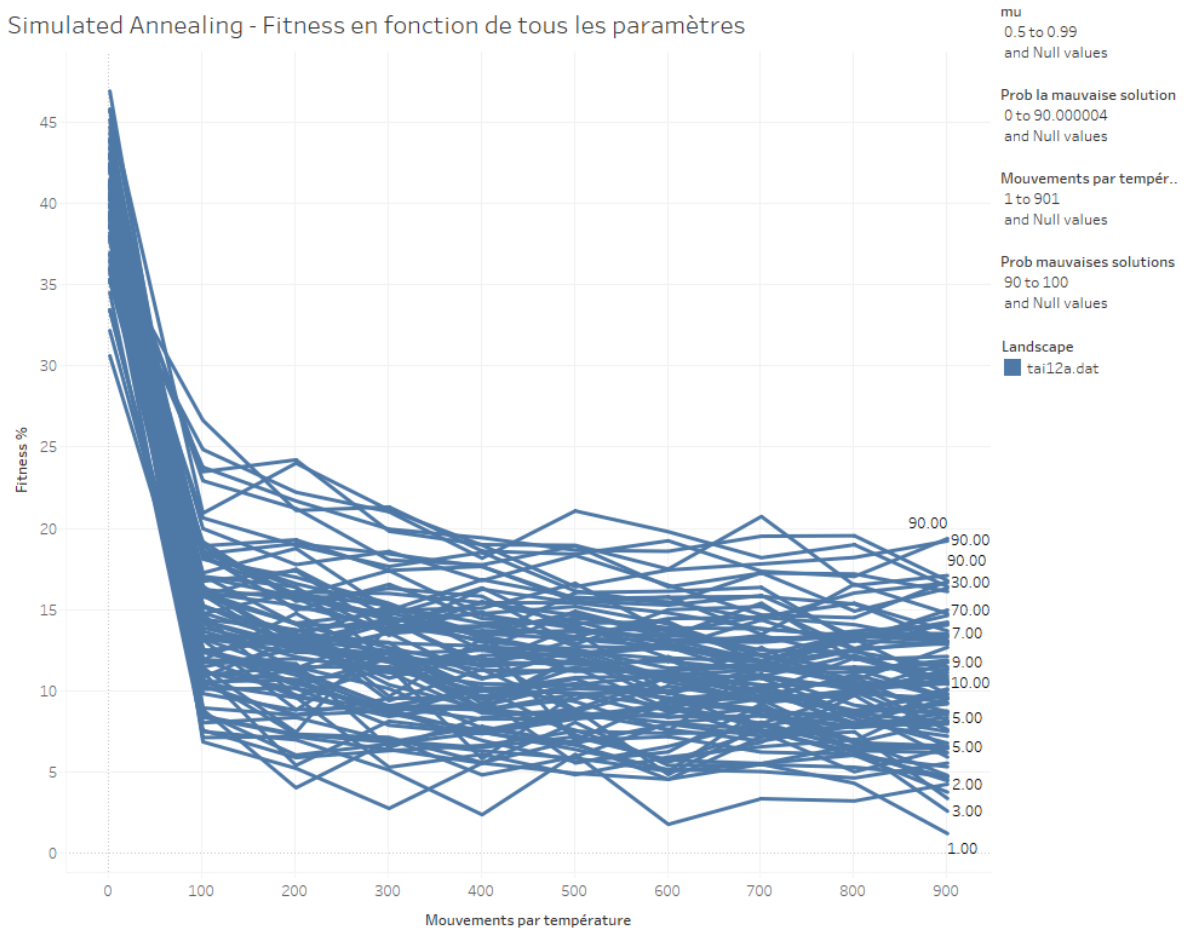
On notera également que cette séparation s'applique pour les plus grandes instances de Taillard, mais en étant plus rapprochée graphiquement.

Maintenant, on s'intéresse à la probabilité de considérer les mauvaises solutions.

Dans la première analyse, cette probabilité était celle du cours fixée à 80%.

On remarque que cette probabilité permet également de déterminer la convergence.






À partir de 90% de chance de considérer ces mauvaises ,solutions l'algorithme converge, en dessous il y a des résultats qui ne sont pas bons. En effet, on peut hypothétiquement penser que l'algorithme tend trop vite vers un minimum local et en ne considérant pas assez souvent les mauvaises solutions, il n'arrive pas a sortir du minimum local. Ce qui provoque de mauvaises fitness en résultat.

# VI. Conclusion

Pour conclure, le problème d'assignation quadratique est difficile à résoudre, étant donné qu'il est d'ordre NP complet. Avec différent jeu de données sous al forme d'instance de Taillard avec matrice symétrique (instance a), il est possible d'appliquer plusieurs algorithmes pour essayer de déterminer la fitness minimale.

Avec une approche naïve (la marche aléatoire), on peut obtenir en très peu de temps des fitness proche de 15% de la fitness optimale. C'est un bon début par rapport au temps, mais le nombre d'itérations ne permettra pas (en un temps raisonnable) de déterminer de meilleures fitness. Augmenter significativement le nombre d'itérations s'apparenterait simplement à du bruteforçage, ce qui n'a pas beaucoup d'intérêt.

Puis avec une seconde approche simple, la méthode Hill-Climbing, on obtient de très bonne fitness entre 6 et 13% de l'optimale. On converge en peu de temps vers une bonne solution, mais le nombre d'itérations de l'algorithme ne permet pas de passer en dessous.

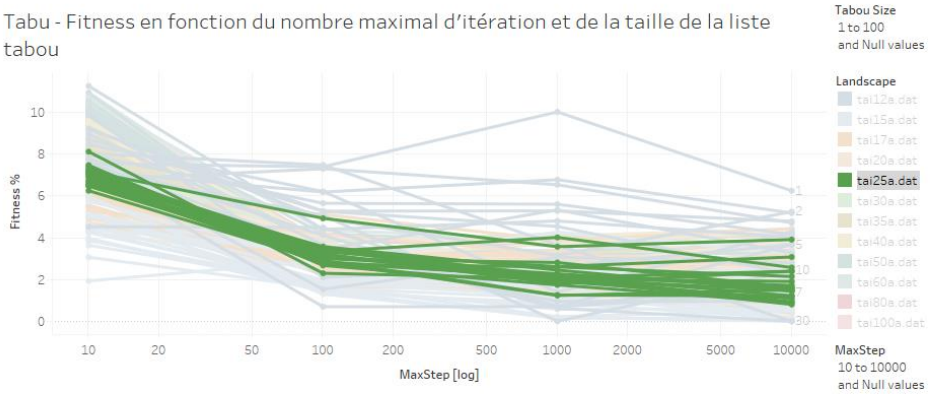
Il est alors intéressant d'utiliser l'algorithme du recuit simulé pour déterminer le meilleur minimum local de l'instance de Taillard en acceptant une marge d'exploration de mauvaise solution. L'algorithme utilise toutefois plusieurs paramètres d'entrée. On se rend compte qu'avec une gestion de la température qui diminue progressivement, une faible probabilité de prendre une mauvaise solution et une probabilité raisonnable de tout de même considérer les mauvaises solutions, on obtient un algorithme qui converge assez vite (en nombre de mouvement par température et en temps) vers la solution optimale.

Dans le domaine de considérer les mauvaises solutions, on peut utiliser la méthode Tabou. On a remarqué que pour la taille des instances, une grande liste Tabou n'est pas très utile, puisqu'elle va finir par rajouter du bruit dans les solutions. Toute fois en un temps de calcul assez ong, on trouve assez vite de très bonnes fitness, inférieures à 5% en moins d'une minute de calcul. C'est bien sûr un temps long, mais la taille de l'instance de Taillard intervient directement dans le calcul du voisinages, ce qui consomme beaucoup de temps de calcul.

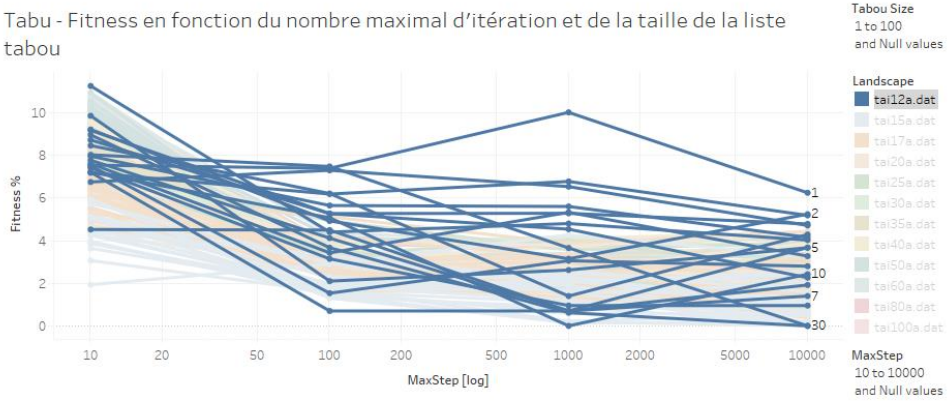
Ces algorithmes permettent efficacement de tenter de résoudre les problèmes d'assignation quadratique des instances de Taillard. On remarque que la taille des instances entre directement en considération sur le temps d'exécution des algorithmes. De plus un paramétrage fin avec de bons paramètres permet de maximiser la performance des algorithmes.

# VII. Annexes

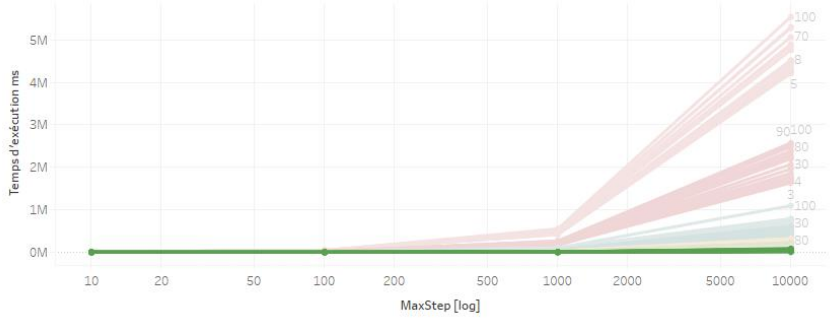
Tabu - Fitness en fonction du nombre maximal d'itération et de la taille de la liste tabou



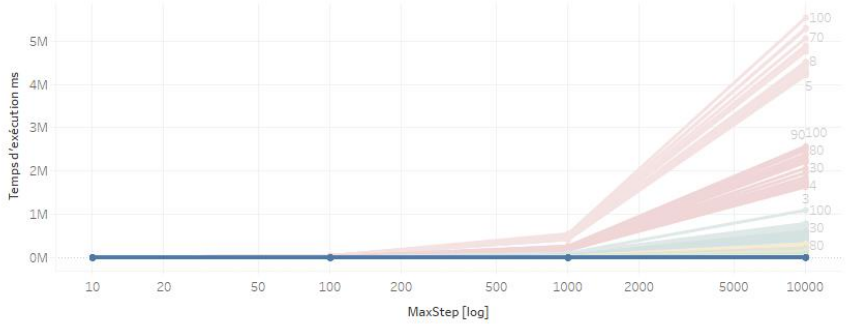

Tabu - Fitness en fonction du nombre maximal d'itération et de la taille de la liste tabou

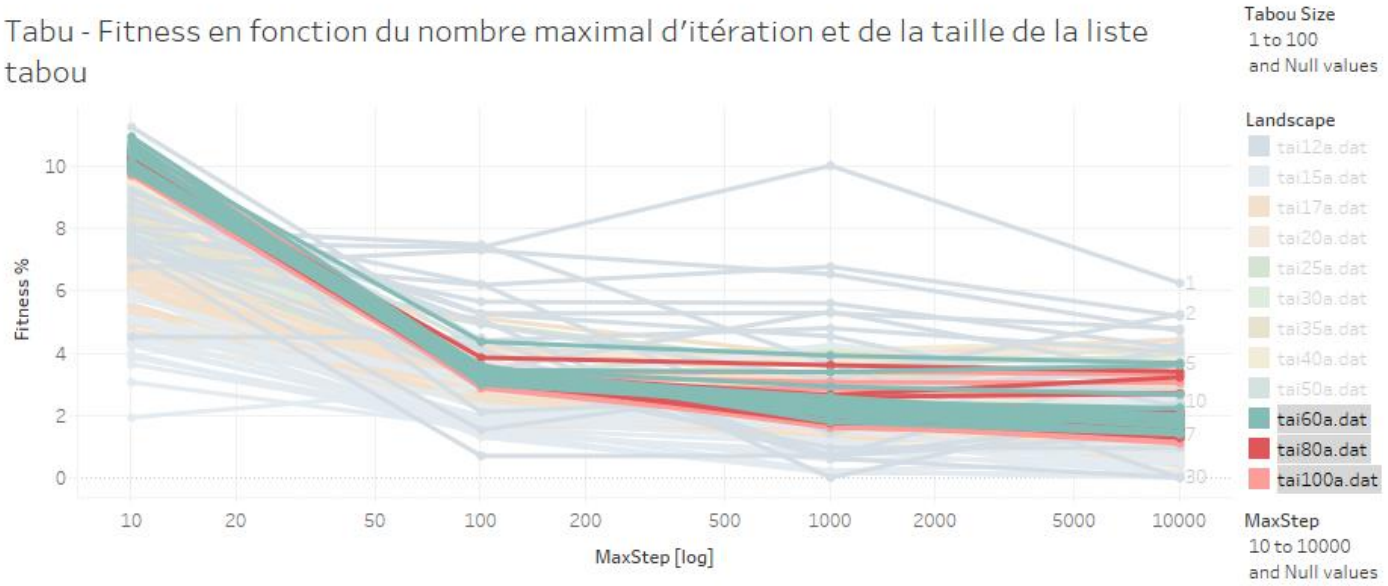


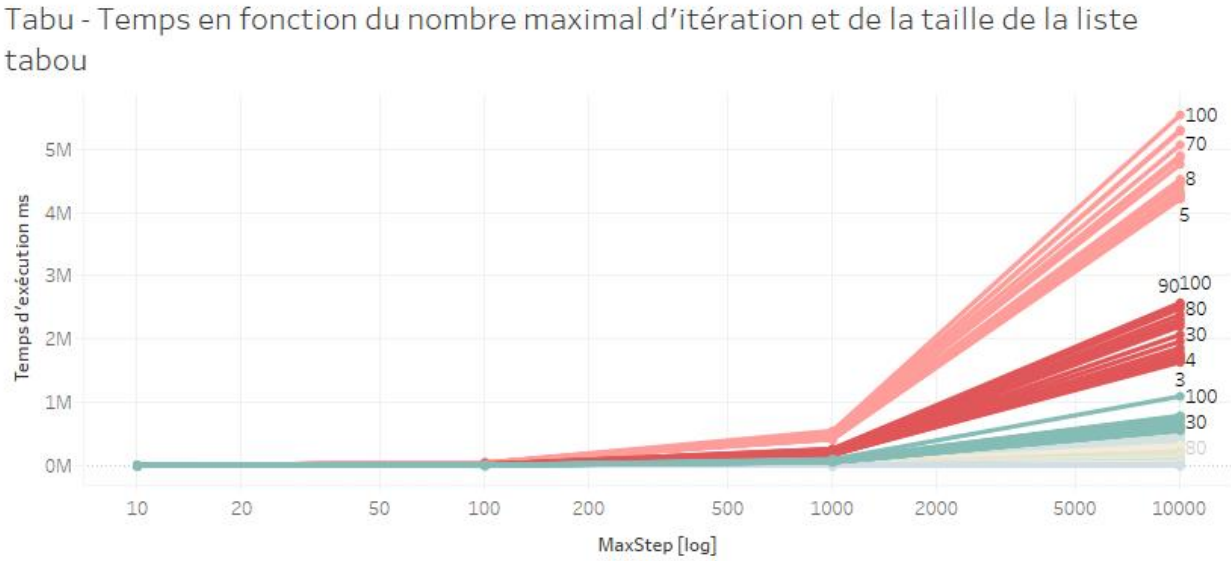

Tabu - Temps en fonction du nombre maximal d'itération et de la taille de la liste tabou




Tabu - Temps en fonction du nombre maximal d'itération et de la taille de la liste tabou




Tabu - Fitness en fonction du nombre maximal d'itération et de la taille de la liste tabou
Fitness %
0
2
4
6
8
10
10
20
50
100
200
500
1000
2000
5000
10000
MaxStep [log]
1
2
5
10
7
30
Tabu Size
1 to 100
and Null values
Landscape
tai12a.dat
tai15a.dat
tai17a.dat
tai20a.dat
tai25a.dat
tai30a.dat
tai35a.dat
tai40a.dat
tai50a.dat
tai60a.dat
tai80a.dat
tai100a.dat
MaxStep
10 to 10000
and Null values


Tabu - Temps en fonction du nombre maximal d'itération et de la taille de la liste tabou
Temps d'exécution ms
0M
1M
2M
3M
4M
5M
10
20
50
100
200
500
1000
2000
5000
10000
MaxStep [log]
100
70
8
5
90
100
80
30
4
3
100
30
80