Introduction

Le **problème d'affectation quadratique** consiste en attribuer au mieux des tâches et des agents. Les problèmes d'affectation auxquels nous sommes confronté sont les instances de Taillard. Celles-ci consiste en trouver le positionnement de n équipements sur n emplacements tout en minimisant le coût total d'affectation, définie par la fitness ci-dessou.

Soit W une fonction de poids définie entre les équipements et D une fonction de distance entre les emplacements des équipements :

$$\Sigma\, W(a,b)\; D(f(a),f(b))\; avec\; a,b\; \in\; P, l'ensemble\; des\; équipements$$

Pour répondre à ces problèmes, nous avons implémenté deux méthodes de résolution à savoir le **recuit simulé** et **tabou**. Nous avons choisi de les implémenter en code Java.

Dans ce rapport, nous commencerons par décrire les instances de Taillard utilisé pour les tests. Puis nous argumenterons nos choix d'implémentation et de paramétrage. Pour finir, nous ferons une étude comparative des deux méthodes implémentées.

I - Les instances de Taillard

Pour commencer, il faut savoir qu'il existe plusieurs instances de Taillard décrite dans plusieurs fichiers. Elles se différencies par leurs nombre d'équipements. Ainsi l'objectif sera de trouver la solution optimale le nombre d'équipements décrit.

Un fichier d'une instance de Taillard est composé des informations nécessaires au calcul de la fitness.

Nous avons une première matrice à disposition permettant de récolter le coût entre deux équipements. Pour illustrer, dans taillard 12, nous avons 12 équipements à placer. Soit a,b $\in$ $[1,12]$ deux équipements, le point de coordonnées [a,b] représente **le coût(poids)** des deux équipements.

Autrement, nous avons une seconde matrice dans chaque instance de Taillard décrivant la distance entre deux emplacements. Pour illustrer, soit a un équipement placé à l'indice x et b un autre équipement placé à l'indice y alors le point de coordonnées $[x, y]$ donne **la distance** entre ces deux emplacements x et y.

Comme demandé, dans notre projet nous nous sommes uniquement intéréssé aux instances de "taillardXX**a**". Celles-ci sont décrite dans des matrices symétrique.

Pour récupérer ces données, nous passons donc par une lecture de fichier implémentée dans la classe Outils qui initialise les matrices *poids* et *distance*.

II - Paramètres

1. Solution initiale

Afin de réaliser des calculs pertinents, nous initialisons une solution initiale complétement aléatoire.

2. Notion de voisinage

Les deux algorithmes utilisants des voisins, nous avons choisis d'abord un voisinage où l'on fait une transformation locale, en intervertissant l'emplacement de deux machines, par exemple :

Si = (4 , 6 , 11 , 1 , 7 , 2 , 10 , 9 , 8 , 12 , 3 , 5)

V(Si) = (**5** , 6 , 11 , 1 , 7 , 2 , 10 , 9 , 8 , 12 , 3 , **4**)

Concernant la méthode Tabou, un autre voisinage a pu être testé. Nous avons fait en sorte de permuter toutes les pairs d'équipements possible. Il est logique que cette notion de voisinage est plus coûteuse en temps de calcul, mais nous nous demandons si elle peut s'avérer payante.

3. Calcul de la fitness

Les premières instances de Taillards étants de petites tailles, nous recalculions entièrement la fitness d'une nouvelle solution, et cela même si l'on intervertissait seulement deux machines. Hors, pour les instances de plus grandes ampleurs, le temps de calcul d'une fitness devient conséquent, il a été donc nécessaire d'optimiser ce calcul, ce qui nous a permit de gagner un temps non négligeable.

Prenons l'exemple d'une solution possible d'une instance de Taillards 12 :

S0 = (4 , 6 , 11 , 1 , 7 , 2 , 10 , 9 , 8 , 12 , 3 , 5)

Nous générons un voisin de S0, en permutant les machines aux index **1** et **12** :

V(S0) = (**5** , 6 , 11 , 1 , 7 , 2 , 10 , 9 , 8 , 12 , 3 , **4**)

Afin de calculer la fitness de V(S0), nous calculons dans S0 la fitness liée aux machines 4 et 5, puis les autres machines :

- fitness(4) = $\Sigma\, W(\boldsymbol{4},b)\; D(\boldsymbol{1},f(b))\; avec\; b \in P, l'ensemble\; des\; \boldsymbol{é}quipements$
- fitness(5) = $\Sigma\, W(\boldsymbol{5},b)\; D(\boldsymbol{12},f(b))\; avec\; b \in P, l'ensemble\; des\; \boldsymbol{é}quipements$

Pour les autres machines, nous calculons la fitness associée aux index qui ont été changés :

- fitness(x) = $\Sigma\, W(a,\boldsymbol{4})\; D(f(a),\boldsymbol{1})\; avec\; a \in P, l'ensemble\; des\; \boldsymbol{é}quipements$
- fitness(y) = $\Sigma\, W(b,\boldsymbol{5})\; D(f(b),\boldsymbol{12})\; avec\; b \in P, l'ensemble\; des\; \boldsymbol{é}quipements$

Ensuite, nous additionnons ces fitness afin d'avoir les fitness qui seront modifiées après la permutation, fitnessAvantPermutation = fitness(4) + fitness(5) + fitness(x) + fitness(y).

Nous répétons le même processus, sur V(S0) cette fois-ci, afin d'obtenir la fitness des machines ayant changées :

Nous calculons donc, sur la solution voisine, la fitness des machines aux index venant d'être permuttés :

- fitness(5) = $\Sigma\, W(\boldsymbol{5},b)\; D(\boldsymbol{1},f(b))\; avec\; b \in P, l'ensemble\; des\; \boldsymbol{é}quipements$
- fitness(4) = $\Sigma\, W(\boldsymbol{4},b)\; D(\boldsymbol{12},f(b))\; avec\; b \in P, l'ensemble\; des\; \boldsymbol{é}quipements$

Pour les autres machines, nous calculons la fitness associée aux index qui ont été changés :

- fitness(x) = $\Sigma\, W(a,\boldsymbol{5})\; D(f(a),\boldsymbol{1})\; avec\; a \in P, l'ensemble\; des\; \boldsymbol{é}quipements$
- fitness(y) = $\Sigma\, W(b,\boldsymbol{4})\; D(f(b),\boldsymbol{12})\; avec\; b \in P, l'ensemble\; des\; \boldsymbol{é}quipements$

Pareil, nous additionnons ces fitness afin d'obtenir la fitness des éléments impactés par la permutation, fitnessApresPermutation = fitness(4) + fitness(5) + fitness(x) + fitness(y).

Afin d'obtenir la fitness finale, la fitness totale de V(S0), nous procédons comme suit :

fitness(V(S0)) = fitnessApresSwap - fitnessAvantSwap + fitness(S0).

Afin d'illustrer cette optimisation, nous mesurons le temps d'exécution sur une instance Taillards100a, avec un refroidissement de 0.99, un nombre d'itération avec la même température de 500, et un température et nombre d'itération généré selon les principes expliqués précédemment :

	Recuit avec fitness non optimisée	Recuit avec fitness optimisée	Gain de temps
Taillard100a	13228 ms	2713 ms	10 515 ms (-79%)
Taillard150b	29924 ms	3851 ms	26 073 ms (-87%)
Taillard256c	90526 ms	8335 ms	82 191 (-91%)

Ainsi, nous pouvons remarquer un gain de temps conséquent, avec un gain de temps évoluant très rapidement selon la taille du problème, jusqu'à 82 secondes de gagner.

III - Etude comparative

De manière générale, le protocole afin de réaliser des études statistiques, est de réaliser 1000 fois l'algorithme testé avec à chaque fois une solution initiale aléatoire, puis nous en extrayons la moyenne, ainsi que d'autres valeurs.

1. Tests Recuit Simulé

A) Définition

Le recuit simulé, comme vu en cours, est une métaheuristique, qui est inspirée par le domaine de la métallurgie. On alterne des cycles de refroidissements (diminution de la température) et de réchauffement (conservation de la température), afin de diminuer l'énergie (ici, la fitness).

Cette métaheuristique requiert quelques paramètres à son exécution :

- Température initiale (T0) :

Ce paramètre est important, car s'il est trop proche de 0, nous allons moins accepter de moins bonnes solutions que ce que nous avons actuellement. Ainsi, il est nécessaire d'avoir un t0 suffisamment élevé, afin d'accepter un large panel de solutions pour ne pas tomber dans un minimum local. Comme vu dans le cours, nous initialisons t0 de sorte que nous acceptions 80% des moins bonnes solutions :

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

Le $-\Delta f$, représente la différence entre deux solutions, la plus grande trouvée, sur un ensemble de test.

- N1

Ce paramètre est le nombre de fois où nous atténuons la température, il est calculé de sorte à avoir 1 chance sur 100 d'avoir la même mauvaise solution :

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

Il doit être suffisamment grand pour diminuer la température de manière efficace, afin de ne pas accepter de de mauvaises solutions.

B) Tests avec paramètres générés

Les tests s'effectueront avec les paramètres N et T0 calculés comme expliqué précédemment.

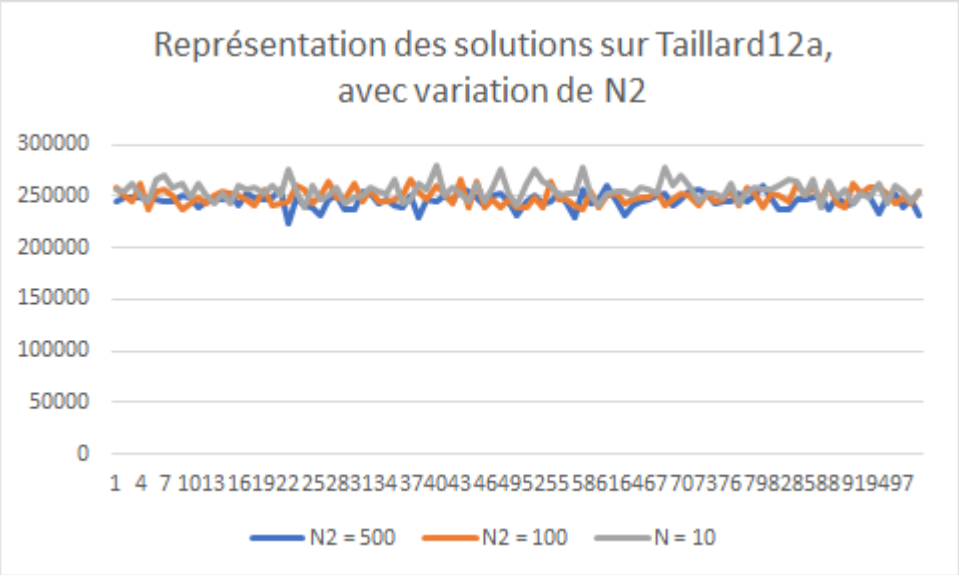
Paramètres fixes (générés) :

- N1 : 4322
- Température initiale : 475218

Paramètre variable :

- N2 : Nombre d'itération avec la même température

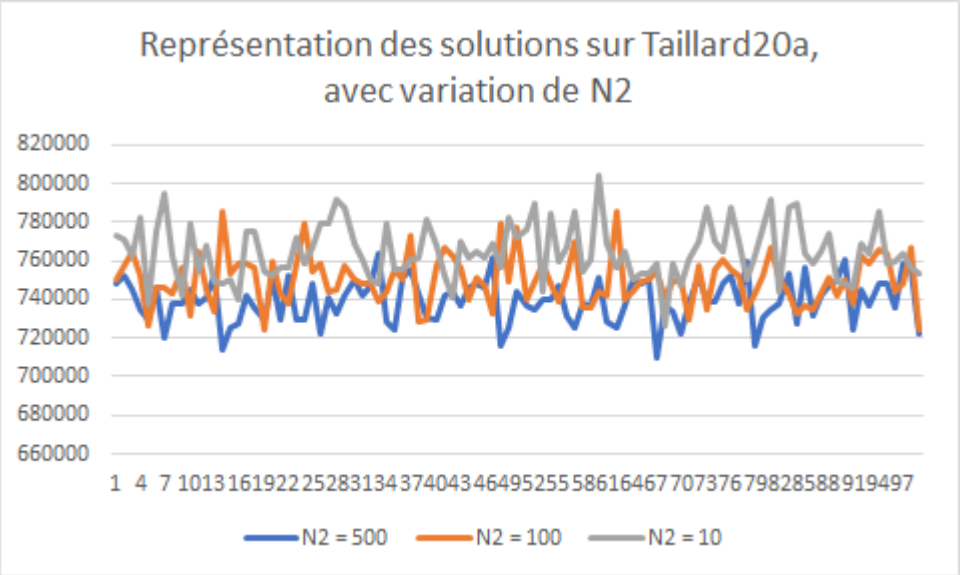
	N2	µ	Fitness Moyenne	Amélioration moyenne	Nombre de fois où la fitness optimale est trouvée
Taillard12a	500	0.99	245889	20.7%	16 / 1000
	100	0.99	249764	19.5%	6 / 1000
	10	0.99	255726	17.5%	0 / 1000



- N1: 4322
- Temperature : 769773

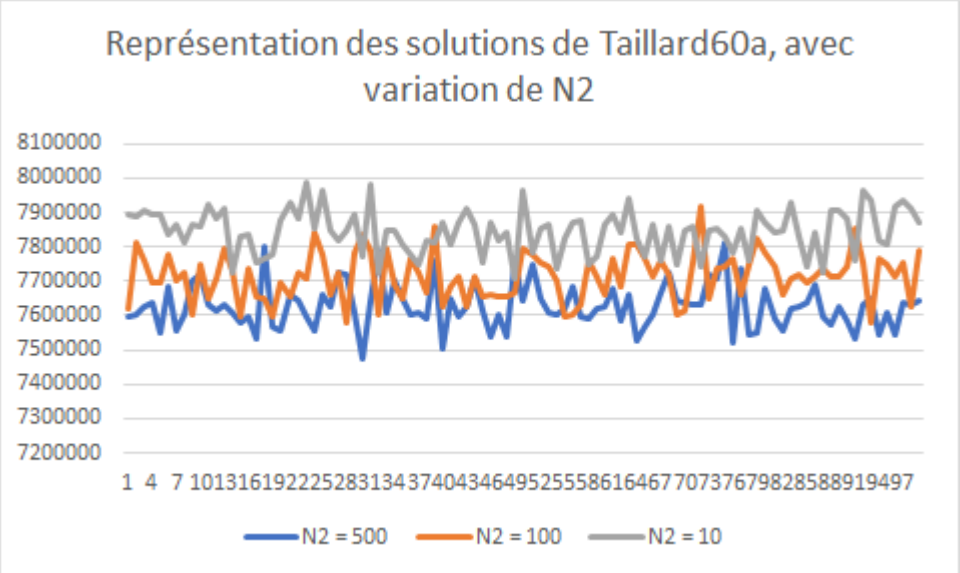
	N2	µ	Fitness Moyenne	Amélioration moyenne	Série
Taillard20a	500	0.99	739127	17%	0 / 1000

	100	0.99	749196	15.9%	0 / 1000
	10	0.99	825732	14.1%	0 / 1000



- N1: 4322
- Temperature : 2327559

	N2	μ	Fitness Moyenne	Amélioration moyenne	Série
Taillard60a	500	0.99	7623638	10%	0 / 1000
	100	0.99	7708466	9%	0 / 1000
	10	0.99	7850750	7.3%	0 / 1000



Nous remarquons que la fitness ainsi que la solution se dégrade en réduisant le nombre d'itération avec la même température, et que la solution s'améliore en utilisant plus de fois la même température pour le recuit.

En effet, plus la température se rapproche de 0, moins il y'a de chance d'accepter de moins bonnes solutions, ainsi nous tomberons plus facilement sur des minimums locaux.

Le fait d'avoir un grand N2 permet d'accepter plus de moins bonnes solutions, et donc, d'explorer un plus large éventail de solution, ce qui apporte plus de chances d'avoir une meilleure solution globalement.

Concernant le paramètre μ, il diminue plus rapidement la température si il est faible, et ainsi on refusera à nouveau de moins bonnes solutions, ce qui reviendra à un algorithme glouton.

2. Tests Tabou

a. Algorithme de descente:

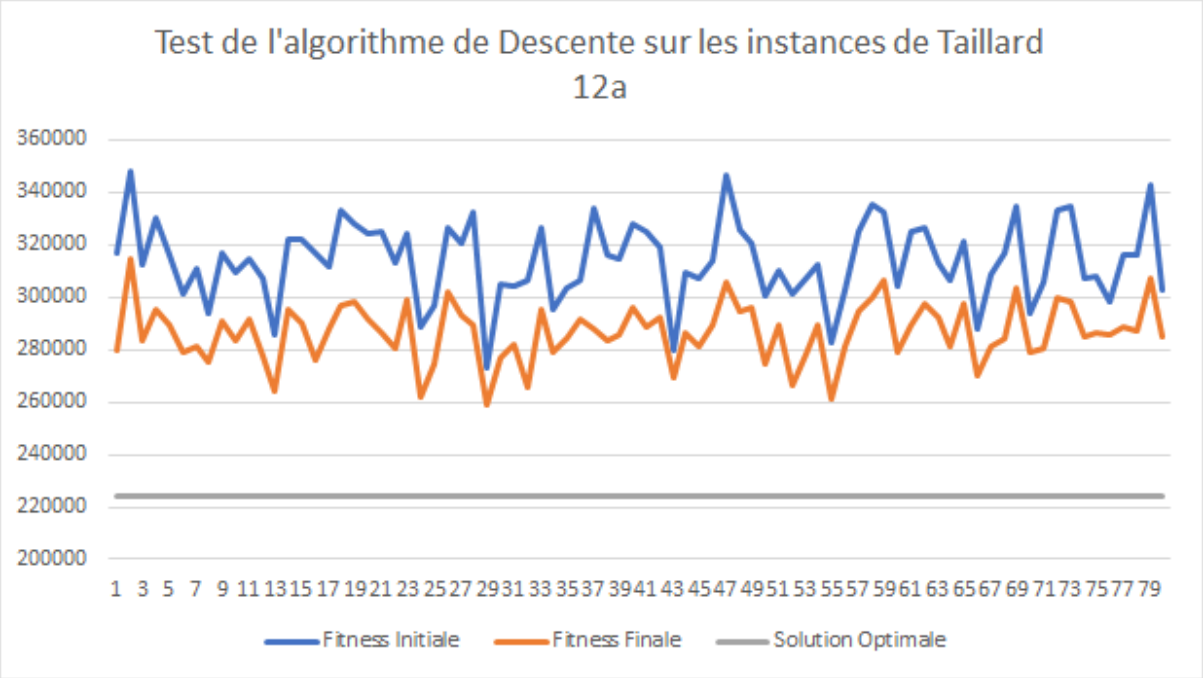
Tabou est une amélioration de l'algorithme de descente. Ainsi, nous allons nous intéressé à l'algorithme de descente avant de nous intéresser à l'algorithme Tabou.

L'algorithme de descente se déroule sur une unique itération. Il part d'une solution initial puis entame une recherche de voisins et détermine si l'un des ses voisins à une fitness inférieur à la fitness actuelle. Si c'est le cas alors, l'algorithme modifiera sa solution optimale, sinon il retournera celle de la solution initiale.

Comme type de voisins, nous avons choisi d'implémenter toutes les pairs d'équipements réalisable.

Le seul paramètre pouvant être touché est le voisinage.

Voici quelques tests réalisés sur cet algorithme:
Pour les prochains tests, nous avons relevé un échantillons de 1000 test.



Ici, nous avons une représentation graphiques des 80 premiers tests. Sur ce graphique nous pouvons voir que les résultats sont généralements loins de la solution optimale.

Indicateurs	Fitness initiale	Fitness Final	Delta
Moyen	312588,0579	286339,8342	26248,2238
Min	donnée non pertinente	255112	7180
Max	donnée non pertinente	323358	55584

Dans le tableau ci-dessus sont représentés les 1000 tests. Le tableau confirme ce que nous avions déduit du graphique. En effet, on peut voir en moyenne la finesse final est supérieur de d'environ 62000.
Aussi, nous pouvons voir que l'algorithme de descente est assez instable puisque d'une itération à une autre nous pouvons trouver un résultat différent. En effet, dans notre cas nous pouvons voir que le delta (Fitness Initiale- Finale) est d'en moyenne 26248 mais avec un minimum à 7180 et un maximum à 55584.

Néanmoins le principal avantage de cette méthode est qu'elle est très rapide. En effet, sur nos test nous avons constaté un temps d'exécution moyen de 0.644ms.

Nous avons constaté la faiblesse de l'algorithme de descente sur une petit jeu de données. Ainsi, nous préférons passer à Tabou.

b. L'algorithme Tabou

L'algorithme Tabou est une amélioration de l'algorithme de descente. En effet premièrement Tabou à un nombre d'itération supérieur à 1. Deuxièmement Tabou est aussi un algorithme explorer. Dans le cas où les voisins d'une solution ont tous une fitness supérieur à la solution, l'algorithme explorera dans le voisin ayant la fitness minimum. Cet aspect permet de sortir des minimums locaux. De plus, Tabou implémente une liste d'interdit, permettant d'éviter de revenir à une solution précédente. Pour toutes ces améliorations, Tabou devrait avoir des résultats supérieur à l'algorithme de descente.

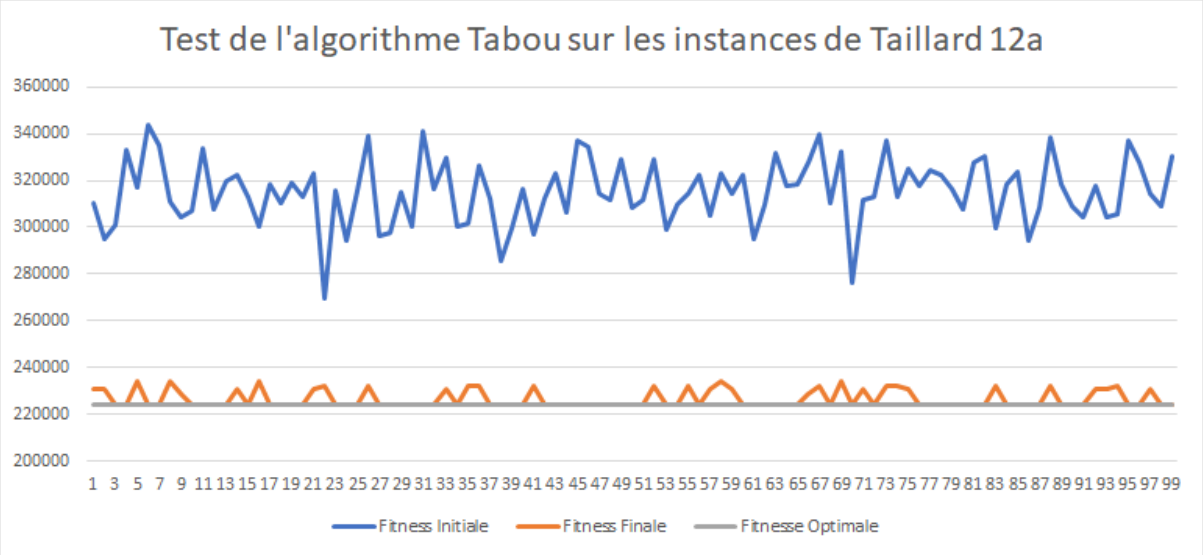
Ainsi les paramètre qui peuvent être influencé pour faire évoluer les résultats sont:

- n : le nombre d'itérations
- m: la taille de la liste tabou
- v: le voisinage

Réalisons des tests:

Encore une fois, nous avons fait 1000 test sur le problème de Taillard 12.a.
Nous avons utilisé les paramètres suivant:

- n=12
- m=12
- v=toutes les pairs d'équipements possibles



Sur ce cas, l'algorithme Tabou a été très efficace. En effet, on peut voir que sur 1000 itérations, il a trouvé la solution optimale dans 663 cas soit 66,3%. Ainsi il apparaît très clair que le nombre d'itération a un impact fort en comparant cet algorithme à celui de descente. De plus, on peut clairement voir que les résultats du Tabou sont très peu dispersés. Ainsi, l'algorithme, en utilisant ce voisinage, est très fiable. Néanmoins, il est assez coûteux en temps(13,865 ms par test).

Indicateurs	Fitness initiale	Fitness Final	Delta
Moyen	312252,232	226882,072	85370,16

Min	donnée non pertinente	224416	37626
Max	donnée non pertinente	239302	130088
Ecart	15436,54927	3708,13338	15852,8017

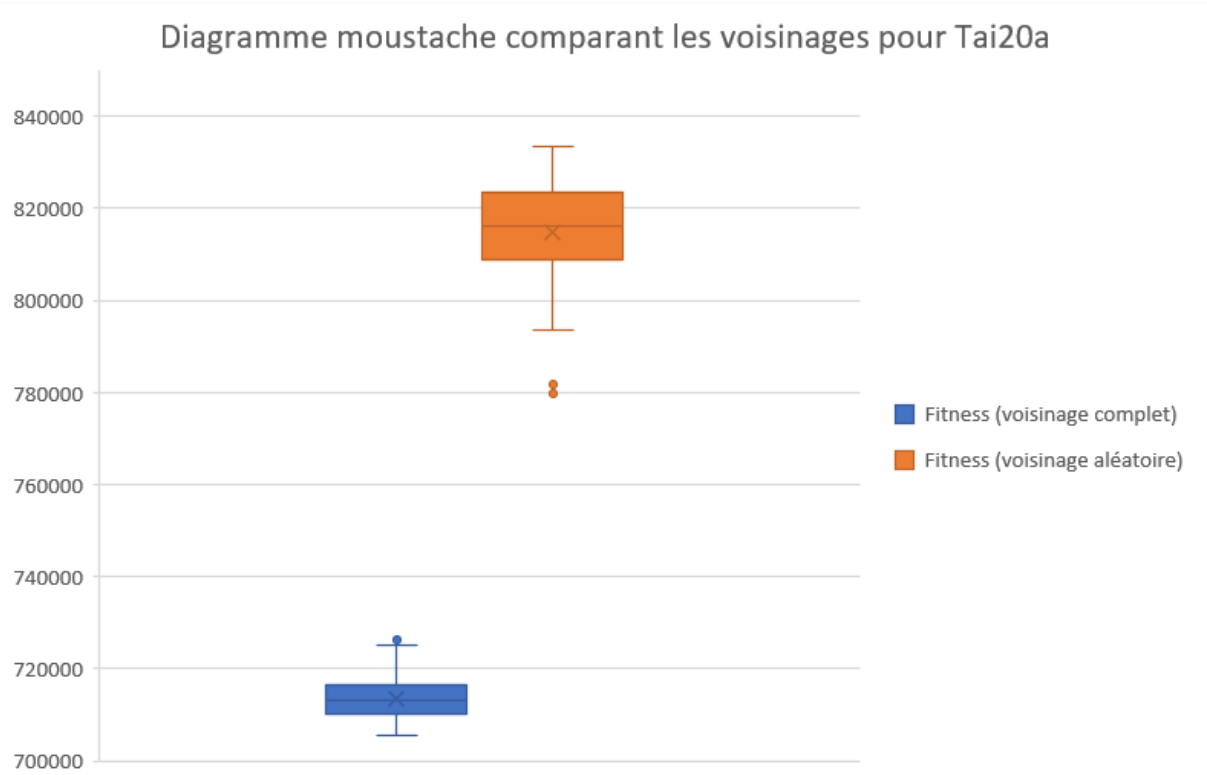
1. influence du voisinage:

Dans les cas précédents, nous avons utilisé comme voisinage l'ensemble des pairs d'équipement. Comme dit précédemment, ceci est coûteux en temps puisque l'on génère à chaque itération n*(n-1)/2 voisins où n le nombre d'équipements.
Ainsi, nous allons tenter de décrypter l'influence du voisinage:

- n : 1000
- m: 12
- v: on sélectionne aléatoirement x un équipement et le permute avec x-1 ou x+1 selon la fitness minimum

Indicateurs	Fitness Final	Delta	Fitness Final(ancien voisinage)	Delta(ancien voisinage)
Moyen	265091,412	47408,328	226882,072	85370,16
Min	243824	2620	224416	37626
Max	278358	105344	239302	130088
ecart	5579,994192	16097,3331	3708,13338	15852,8017

Comme attendu, le temps de calcule à été réduit et ici divisé par 13 (0.906ms en moyenne par itération contre 13,865 avec l'autre voisinage). Néanmoins les performances ont été impacté. En effet, on remarque que la fitness moyenne est beaucoup plus élevé (+39000 environ). Aussi, les résultats sont davantage dispersé. Ici on constate un écart type de la solution finale beaucoup plus élevé lorsque l'on utilise le voisinage défini précédemment.



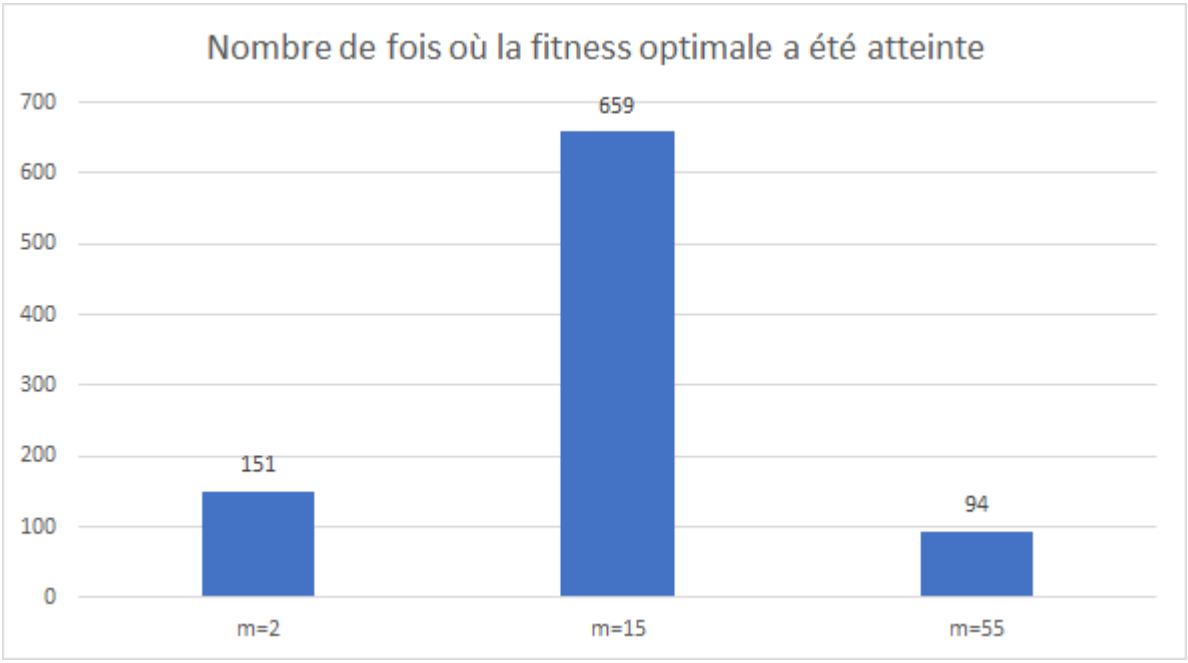
On constate que plus la taille du problème est élevé plus l'écart de performance mais aussi de calcul est élevé.

2. influence de la taille de la liste tabou:

Pour cette partie voici les paramètre que nous allons utiliser les paramètres suivants:

- n : 1000
- v: toutes les pairs d'équipements possibles

Nous avons réalisé 1000 tests sur l'instance de taillard 12a.



Sur ce graphique on voit clairement que la définition de la liste Tabou à un impact direct sur la performance de l'algorithme. En effet, il apparaît très clairement que ce nombre doit ni être trop faible pour éviter de tomber dans des minimums locaux ni trop élevé pour éviter de rester coincé dans un minimum local.

La taille de la liste tabou doit être proportionnel au nombre de voisins générés.

3. Comparaison

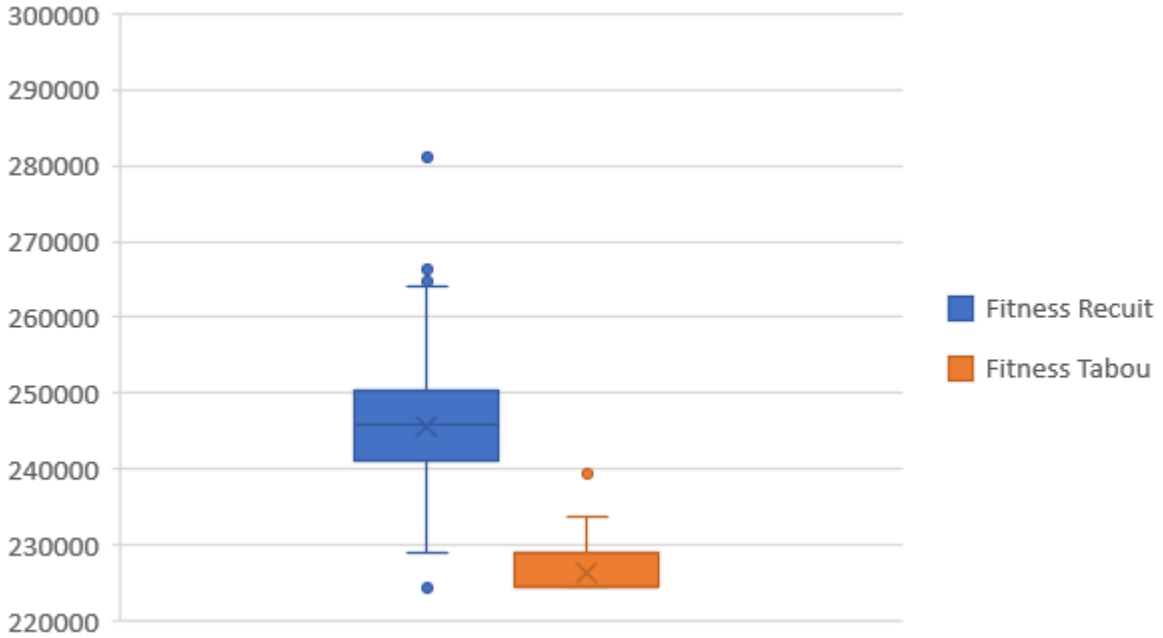
Taillard12a:

Solution initiale : (7 , 2 , 6 , 5 , 12 , 3 , 11 , 8 , 10 , 4 , 9 , 1)
Fitness = 297402

	Fitness moyenne	Amélioration moyenne	Temps d'exécution
Recuit simulé	245742	16.6%	1209 ms (pour 1000 test)
Méthode Tabou	224416		13.46 ms (pour 1 test)

Pour les même paramètre, l'algorithme Tabou est déterministe. Ici en testant avec une solution de départ, on peut voir que Tabou trouve la solution optimale quand le recuit ne l'a trouve pas même avec 1000 itération.

Nous avons tenté 1000 test aléatoire avec les paramètre optimaux de Tabou et du Recuit.



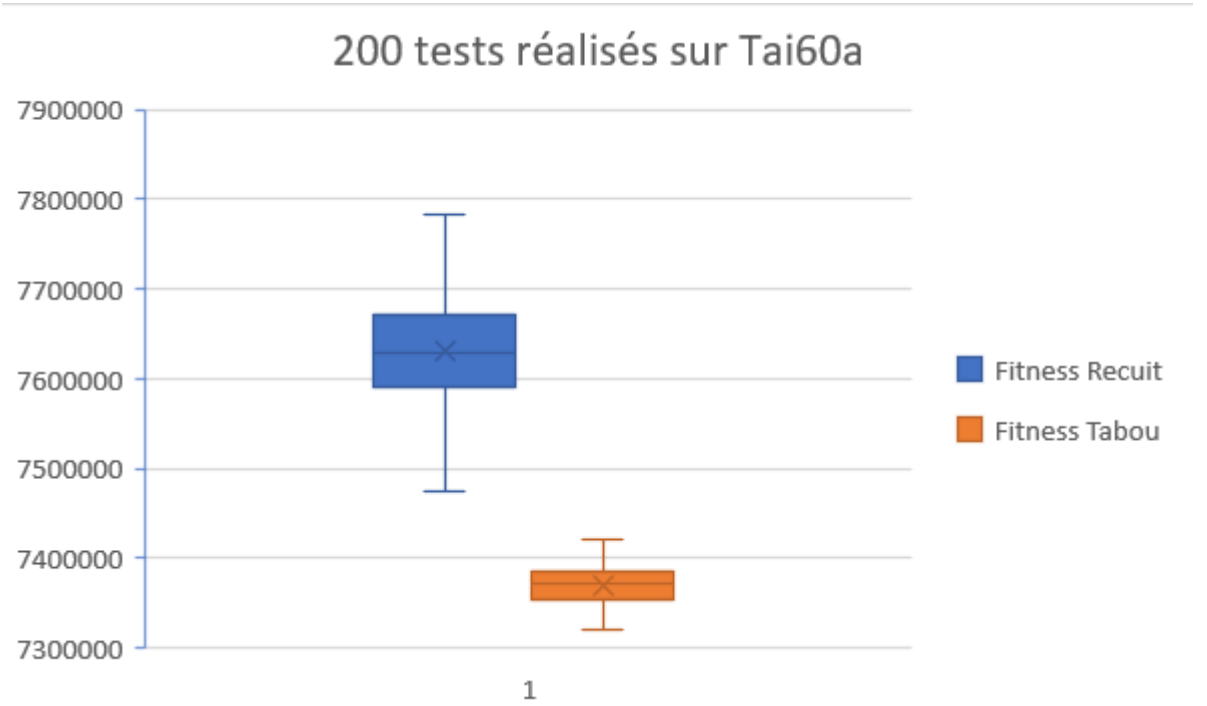
Ici, il apparaît très clair que Tabou est bien plus correct que l'algorithme du recuit simulé. En effet, on peut voir que près de 50 des tests de tabou ont atteint la valeur optimal contre 1 pour le recuit. De plus, la dispersion dans l'algorithme du recuit est beaucoup plus grande que celle de l'algorithme tabou.
Néanmoins, l'algorithme du recuit simulé s'avère plus performant en terme de temps de calcul. En effet, celui ci prend 10 fois moins de temps (1209ms contre 13406ms).

Taillard60a:

Solution initiale : (45 , 8 , 50 , 38 , 60 , 58 , 28 , 15 , 27 , 59 , 42 , 18 , 55 , 1 , 39 , 17 , 20 , 25 , 3 , 54 , 10 , 53 , 43 , 41 , 56 , 14 , 7 , 44 , 4 , 49 , 6 , 11 , 34 , 24 , 23 , 57 , 26 , 33 , 16 , 22 , 46 , 35 , 21 , 36 , 12 , 30 , 37 , 31 , 40 , 5 , 51 , 2 , 47 , 9 , 13 , 52 , 48 , 29 , 19 , 32)

Fitness = 8504590

	Fitness moyenne	Amélioration moyenne	Temps d'exécution
Recuit simulé	7623617	10%	1855 ms
Méthode Tabou			131218



Cette fois nous avons réalisé 200 tests et ce sur un problème conséquent. Encore une fois nous constaté les mêmes résultat en terme de performance. Néanmoins en termes de temps de calcul les résultats se sont aggravés. En effet Tabou a été 65 fois plus long (130000ms contre 2000ms).

Conclusion

Pour conclure, ce projet a été l'occasion pour nous d'intégrer un problème connu de l'optimisation discrète à savoir les instances de Taillard. De plus, nous avons pu mettre en pratique nos connaissances théoriques et comprenant les enjeux des algorithmes d'optimisation combinatoire.

Pour conclure concernant les algorithmes étudiés, concernant le recuit simulé, nous concluons qu'il permet de sortir des minimums locaux avec les bons paramètres, mais le problème est le choix de ces derniers, il peut être difficile de trouver les bons paramètres pour le problème étudié, et ainsi trouver une bonne solution, avec le recuit simulé. Le recuit est un algorithme rapide mais ne trouvant pas la réponse optimale.
Concernant l'algorithme de descente, nous avons pu voir que celui-ci n'était pas abouti. En effet, le nombre d'itérations à un impact direct sur les algorithmes que nous avons étudiés. Pour finir, nous conclurons en pensant que Tabou l'algorithme le plus sûr en terme de résultats. Néanmoins cet algorithme est très sensible à la liste Tabou, au nombre d'itération et à la notion de voisinage. Ainsi, il nécessite de connaître son cas avant de l'employer. Pour finir, une attention particulière est à retenir sur la notion de voisinage qui rend les calculs très long.