Bin Packing 1D

Optimisation discrète

Filière Informatique par apprentissage

Promotion n° 9

Année 2020-2021

Thomas BECHET
Michaël BUGNONE

Question 1:

Borne inférieure : C'est le nombre minimum de bins nécessaire afin de ranger tous les items. Elle est définie en partant du principe que les items peuvent s'empiler afin de parfaitement remplir chaque bins.

Avec ***n*** le nombre d'items et ***p*** la taille d'un bin.

$$\frac{\sum_{i=0}^{n} xi}{p}$$

Si la somme de tous les items modulo la taille des bins n'est pas égale à 0 alors on ajoute 1.

Borne supérieure : C'est le nombre maximal de bins nécessaire afin de pouvoir ranger tous les items. Elle est définie par le fait que chaque item à ranger occupe un seul bin.

Elle est donc égale au nombre d'items.

Question 2:

Afin de résoudre ce problème de bin packing, il est possible d'utiliser l'algorithme "FirstFitDecreasing". Cela nous permet de ranger nos items dans un minimum de bin avec un algorithme de faible complexité.

Question 3:

A partir du package de programmation linéaire "lpsolve", le temps de calcul était bien trop élevé pour trouver la solution optimale du problème "binpack1d_00.txt". Nous avons donc créé un problème moins complexe composé de 8 items que nous avons appelé "binpack1d_simple.txt".

En fonction de la taille des items, nous obtenons un temps de calcul pouvant varier de moins d'une seconde à plus de 3 secondes. De plus, lorsque nous avons plus de 8 items, en fonction de la valeur des items, le temps de calcul devient très long (plus de 20 secondes).

Taille bin	Nombre d'item	Item	Temps de calcul(s)	Résultat
5	9	3;4;2;4;2;3;2;1;4	26	6
5	9	3;4;2;4;2;3;2;1;3	0.013	5
5	8	3;4;2;4;2;3;2;4	26	6

5	8	3;4;2;4;2;3;2;2	0.011	5

Question 5 :

Les opérateurs de voisinage ont été conçus pour être déterministes. C'est-à-dire que d'une exécution à une autre, il faut générer les mêmes échanges. Pour cela, nous utilisons un générateur pseudo-aléatoire. De cette manière, il a été plus simple de débugger certains problèmes. Il faut également, que pour une génération, on ait une génération unique. La façon dont nous avons écrit l'algorithme assure que s'il reste une combinaison possible pour l'opérateur, elle sera choisie.

a. Déplacer un objet

Pour déplacer un objet, on choisit d'abord un bin cible et un bin source de manière aléatoire. Ensuite, on choisit un item dans le bin source qui sera déplacée dans le bin cible. Bien sûr, ce déplacement doit respecter la taille maximale du bin. Si la solution n'est pas possible, on relance l'algorithme sur des combinaisons qui n'ont pas été générées. Notre algorithme peut être assez lent dans le cas où beaucoup de solutions ne sont pas possibles.

Pseudo-code :

- On génère la liste des bins cibles
- Temps qu'il reste des bins cibles
    - On génère les bins sources
    - Temps qu'il reste des bins sources
        - On génère la liste des items dans le bin source
        - Temps qu'il reste des items
            - Si l'item peut être déplacé, on le choisit

Dans le cas où toutes les combinaisons ont été parcourues et qu'aucun item n'a pu être déplacé, on lève une exception.

b. Échanger deux objets

L'algorithme de d'échange de deux items est similaire au déplacement d'un item. On ajoute simplement une boucle en plus pour chercher deux objets permettant un échange.

Pseudo-code :

- On génère la liste des bins cibles
- Temps qu'il reste des bins cibles
    - On génère les bins sources
    - Temps qu'il reste des bins sources
        - On génère la liste des items dans le bin source
        - Temps qu'il reste des items source
            - On génère la liste des items dans le bin cible
            - Temps qu'il reste des items bin cible
                - Si le couple d'item peut être inversé, on les choisit.

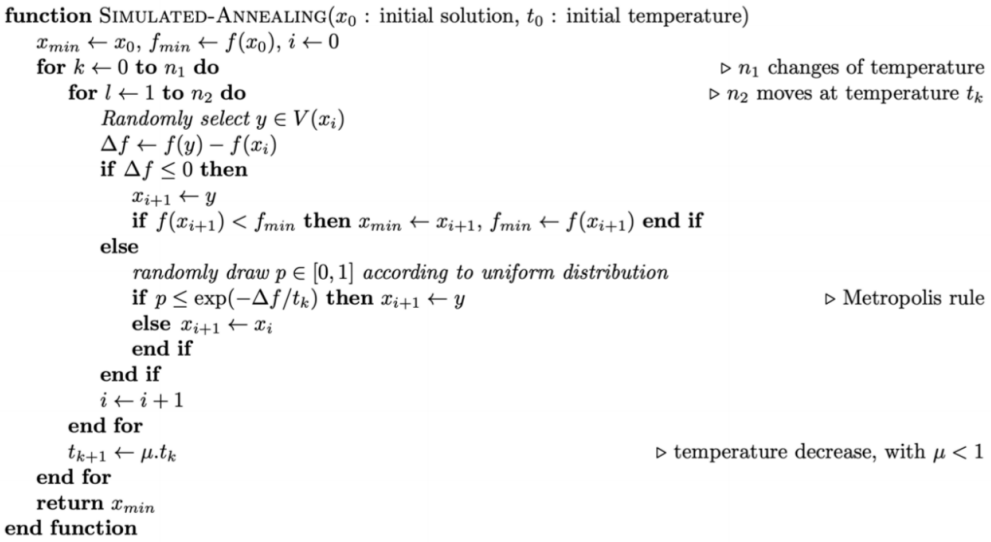
Pour générer le voisinage d'une solution, on applique (pseudo) aléatoirement soit l'opérateur échangeant deux items ou celui déplaçant un item.

Question 6 et 9 - Le recuit simulé :

L'algorithme du recuit simulé a été implémenté en se basant sur le pseudo-code présenté en cours. Ne réussissant pas à obtenir une convergence suffisamment rapide de la solution, nous avons implémenté le même algorithme mais en l'adaptant à la fitness à maximiser. C'est-à-dire que la fitness vaudra la somme des valeurs, le tout au carré. Une meilleure solution aura donc une fitness plus grande. Bien que le terme de finesse n'est plus trop de sens dans ce cas, nous avons préféré le conserver pour rester compréhensible.

Pour adapter l'algorithme, nous avons changé les signes de comparaison pour trouver la solution ayant une finesse maximale et le calcul du delta a été inversé pour obtenir un delta négatif quand la solution trouvée est meilleure que celle sauvegardée.

Pseudo-code - Cours_02_Meta_Voisinages.pdf



function SIMULATED-ANNEALING(x_0 : initial solution, t_0 : initial temperature)
  $x_{min} \leftarrow x_0$, $f_{min} \leftarrow f(x_0)$, $i \leftarrow 0$
  for $k \leftarrow 0$ **to** n_1 **do** ▷ n_1 changes of temperature
    for $l \leftarrow 1$ **to** n_2 **do** ▷ n_2 moves at temperature t_k
      Randomly select $y \in V(x_i)$
      $\Delta f \leftarrow f(y) - f(x_i)$
      if $\Delta f \leq 0$ **then**
        $x_{i+1} \leftarrow y$
        if $f(x_{i+1}) < f_{min}$ **then** $x_{min} \leftarrow x_{i+1}$, $f_{min} \leftarrow f(x_{i+1})$ **end if**
      else
        randomly draw $p \in [0,1]$ *according to uniform distribution*
        if $p \leq \exp(-\Delta f / t_k)$ **then** $x_{i+1} \leftarrow y$ ▷ Metropolis rule
        else $x_{i+1} \leftarrow x_i$
        end if
      end if
      $i \leftarrow i + 1$
    end for
    $t_{k+1} \leftarrow \mu . t_k$ ▷ temperature decrease, with $\mu < 1$
  end for
  return x_{min}
end function

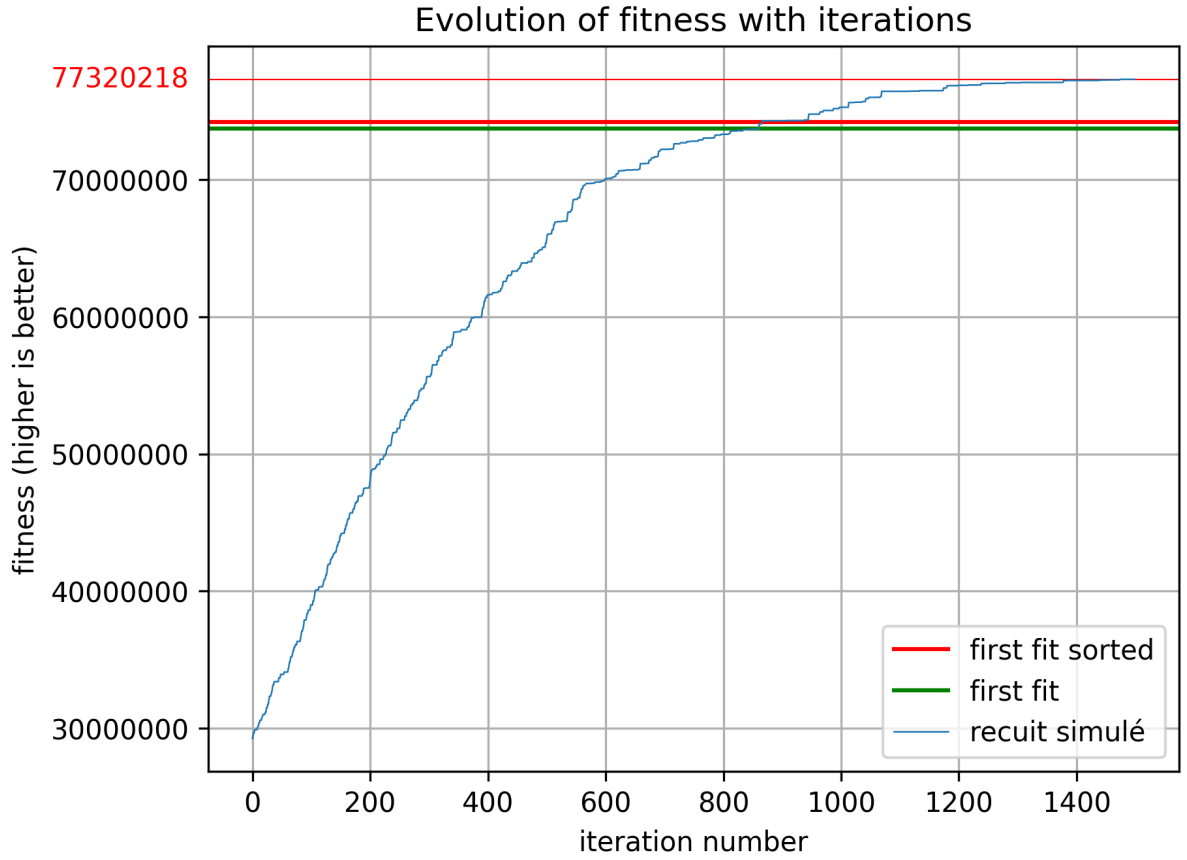
Les paramètres suivants ont été utilisés :

- La température initiale
- Le facteur de réduction qui, normalement, se situe entre 0 et 1. Il peut théoriquement être supérieur à 1, mais il doit rester différent de 0.
- Le nombre de changements de température qui correspond au *n1* dans le pseudo-code. Il s'agit du nombre de multiplications de la température courante par le facteur de réduction.
- Le nombre d'itérations par température qui correspond au *n2* dans le le pseudo-code. C'est le nombre d'itérations pour une même température. Ainsi, on peut faire plusieurs tirages de voisins pour une même température.
- Le nombre de voisins générés qui correspond au nombre de solutions générées à partir de la solution courante.

Exemple n°1:

Pour les tests ci-dessous, nous avons utilisé le jeu de données “binpack1d_05.txt” contenant 249 items et une taille de bin de 1000. Le nombre d’items est suffisamment grand pour mettre en valeur la recherche discrète de solutions.

Paramètres	**Valeurs**
Température initiale	10000
Facteur de réduction	0.99
Nombre de changement de température	300
Nombre d’itération pour une température	15
Voisins générés	100

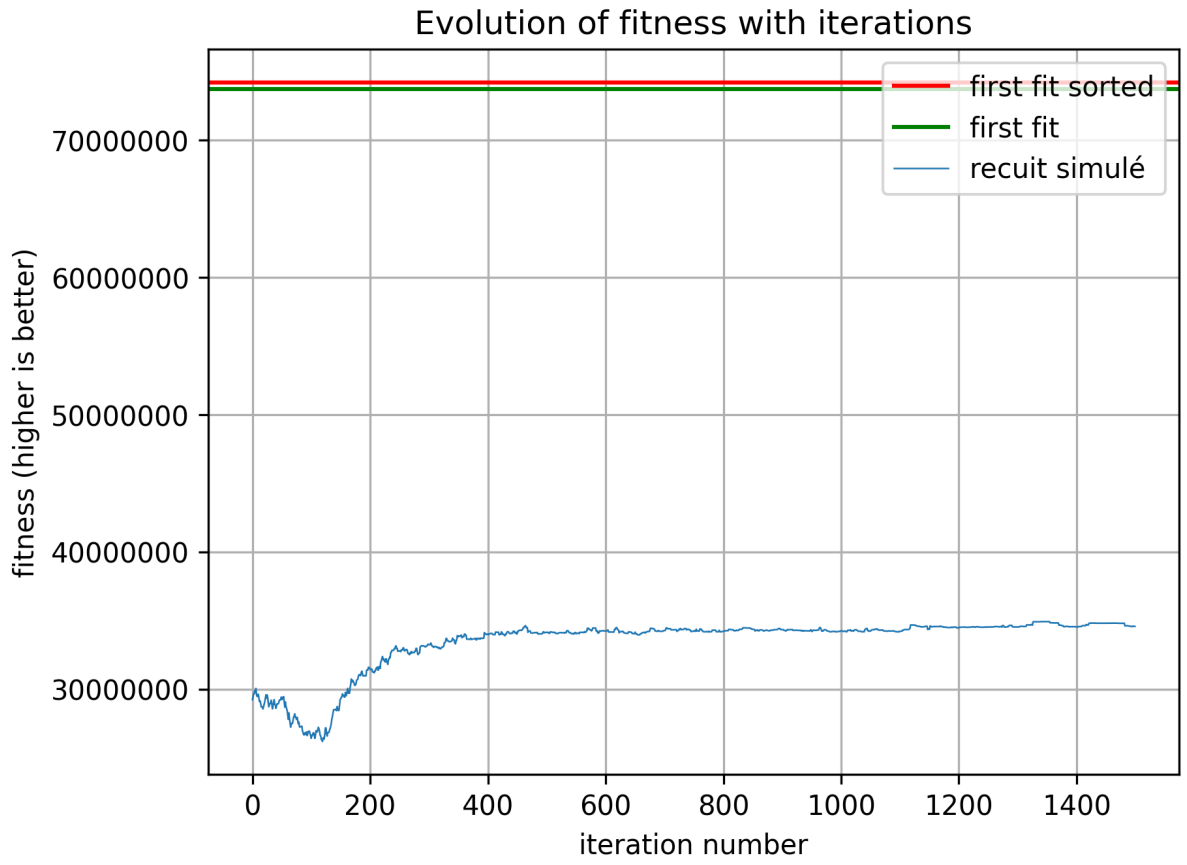


Comme nous pouvons le constater à travers le graphique ci-dessus, l’algorithme du recuit simulé nous permet d’obtenir une meilleure fitness que le first fit sorted et le first fit. La convergence de la solution est assez rapide. En seulement 1500 itérations, on atteint une meilleure solution que les méthodes “naïves”. De plus, on observe qu’il n’y a presque pas de solutions dégradées qui ont été utilisées.

Exemple n°2:

Dans cet exemple, nous avons essayé d'obtenir au maximum des solutions dégradées pour mettre en évidence le fonctionnement de la température dans l'algorithme du recuit simulé.

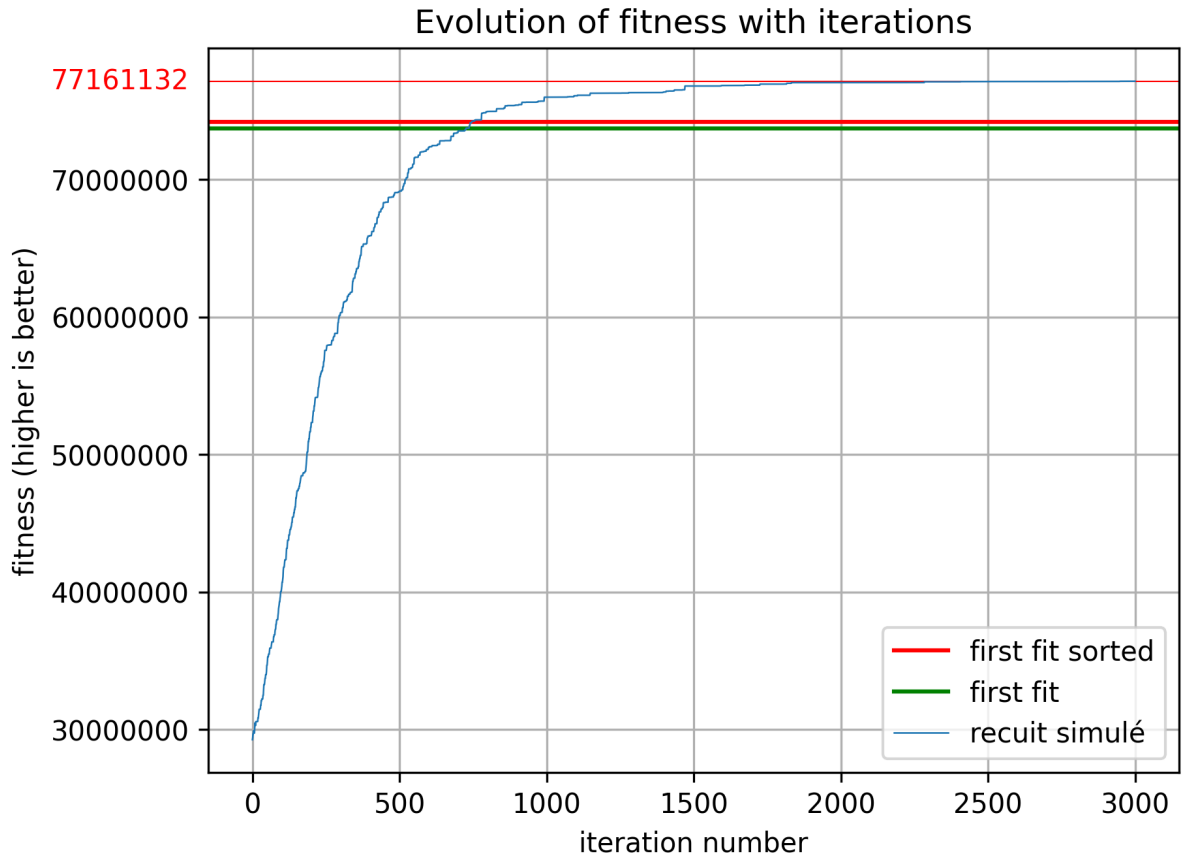
Paramètres	**Valeurs**
Température initiale	1000000
Facteur de réduction	0.99
Nombre de changement de température	300
Nombre d'itération pour une température	5
Voisins générés	100



Pour obtenir de moins bonnes solutions, nous faisons en sorte de conserver une température élevée sur les 1500 itérations. Avec cette température bien plus élevée, l'algorithme ne parvient pas à trouver une bonne solution. Cela est dû au fait qu'il va avoir une forte chance de choisir un plus mauvais voisin à chaque itération à cause de son paramètre de température.

Exemple n°3:

Paramètres	Valeurs
Température initiale	100000
Facteur de réduction	0.80
Nombre de changement de température	600
Nombre d'itération pour une température	5
Voisins générés	100



Dans cet exemple, nous avons essayé de faire jouer le facteur de réduction de température. Pour autoriser des solutions moins bonnes dans les premières itérations, nous avons augmenté la température et réduit le facteur de réduction. Ainsi, l'algorithme doit pouvoir autoriser des moins bonnes solutions au début puis progressivement restreindre ses recherches. On peut remarquer que ce paramètre n'a que peu d'effets dans cet exemple. En effet, le problème du bin packing 1D en utilisant une fitness croissante semble converger assez vite dès les premières itérations.

Question 7 et 9 - La recherche tabou :

L'algorithme de recherche tabou se base sur une liste d'opérateurs. La difficulté a été de stocker ces opérateurs et d'être capable de les comparer entre eux. Pour cela, nous avons deux classes d'opérateurs (SolutionMoveOperator et SolutionSwapOperator). Ces opérateurs sont appliqués sur une solution donnée pour transformer la solution avec la méthode *apply*. En modifiant la solution, les opérateurs conservent les indices des bins et des items qu'ils ont déplacés.

Pour comparer deux opérateurs entre eux. Il faut d'abord qu'ils soient du même type (avec un *instanceof*) mais aussi vérifier leurs indices. Sachant qu'un opérateur transforme un item d'un état A vers B (le déplacer d'un bin A vers B par exemple), on veut aussi vérifier que cette transformation ne se fasse pas de B vers A pour que la recherche tabou soit utile. En effet déplacer un item de A vers B est un opérateur différent de B vers A mais nous souhaitons tout de même le bannir. C'est pourquoi on vérifie à chaque fois si la source de la solution A et de la solution B sont les mêmes et inversement.

Pour développer l'algorithme, nous avons respecté celui vu en cours et en TP.

Pseudo-code :

- On initialise la liste tabou
- Pour chaque itérations
    - On génère les voisins de la solution courante
    - On trie la liste des voisins selon leur finesse
    - Pour chaque voisins de cette liste
        - Si l'opérateur n'est pas présent dans la liste
            - On choisit cette solution
            - On ajoute l'opérateur dans la liste tabou
            - Si la liste est pleine
                - On retire le plus vieil élément
            - On quitte la boucle

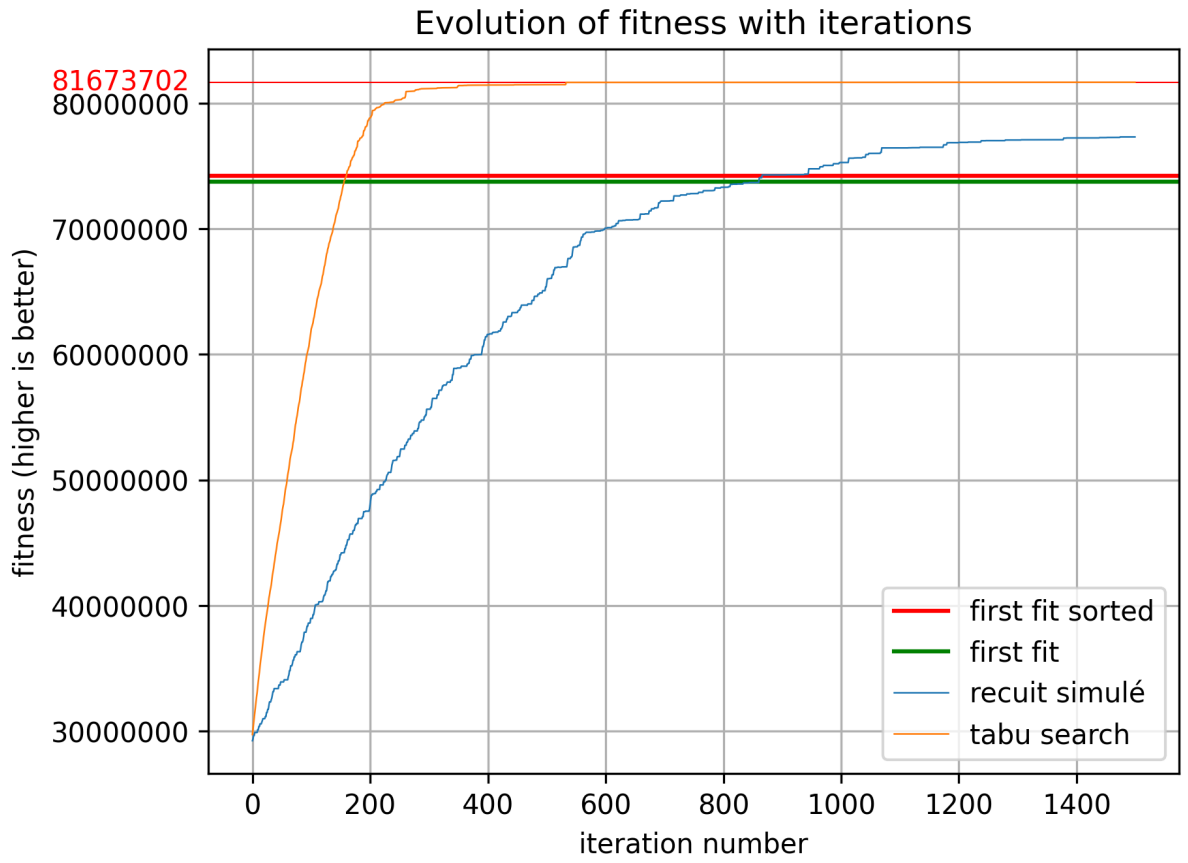
Les paramètres sur lesquelles nous travaillons sont les suivants :

- **Le nombre d'itérations** qui correspond au nombre de fois que l'on va appliquer la recherche tabou.
- **La longueur de la liste Tabou** qui correspond au nombre total d'opérateurs à bannir que la liste peut contenir.
- **Le nombre de voisins générés** qui correspond au nombre de solutions voisines à la solution courante générée. C'est principalement ce paramètre qui influence la qualité de la recherche.

Exemple n°1 :

Dans cet exemple, nous avons choisi le même nombre d'itérations que celui du recuit simulé pour avoir une comparaison de la vitesse de convergence de la solution.

Paramètres	**Valeurs**
Nombre d'itération	1500
Longueur de la liste tabou	5
Voisins générés	600



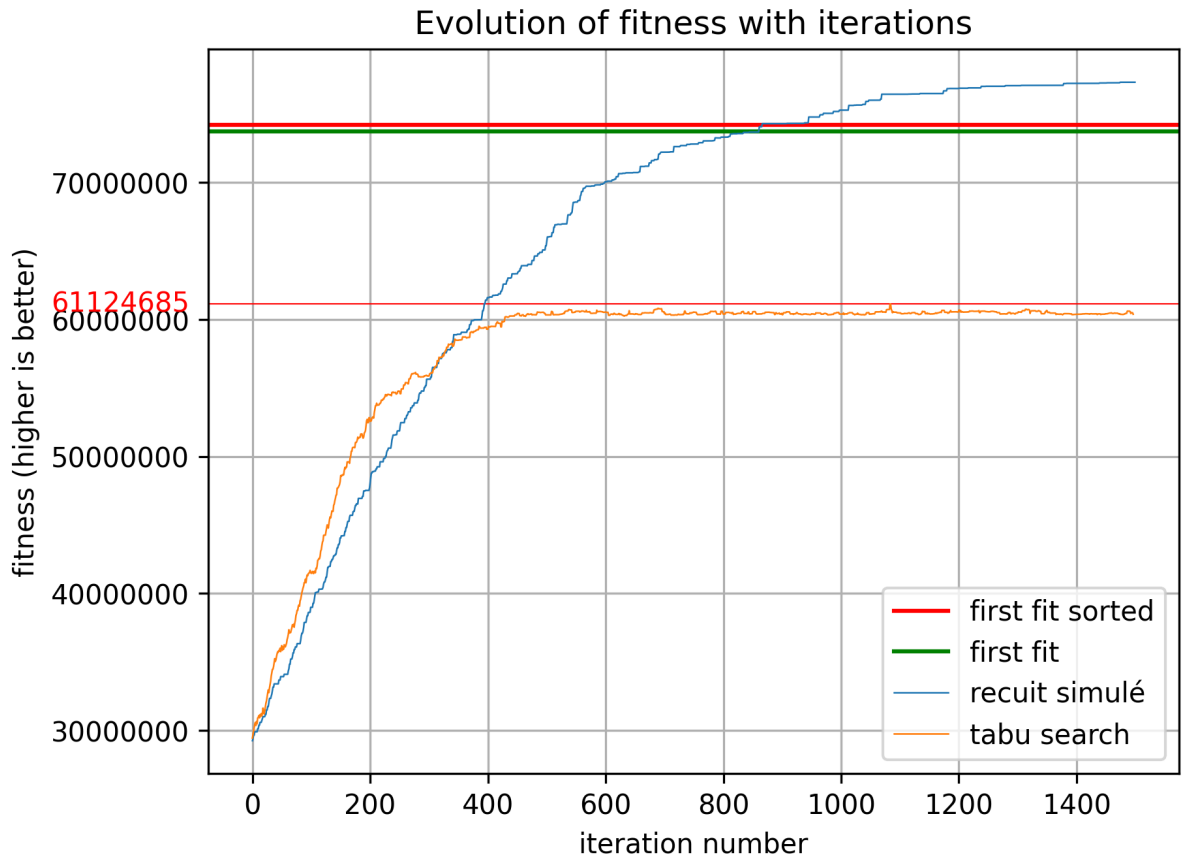
On remarque tout d'abord qu'en utilisant l'algorithme de recherche Tabou, on obtient une meilleure solution que toutes les autres méthodes. De plus, la convergence de cet algorithme est bien plus rapide. En seulement 200 itérations, nous obtenons une solution meilleure que toutes les autres méthodes et proche de la solution optimale théorique. Cette convergence rapide s'explique par le choix des voisins. En effet, nous prenons la meilleure solution parmi les voisins alors que le recuit simulé prend un voisin aléatoire. Étant donné que l'on peut générer beaucoup de voisins, il est fort probable de tomber sur une meilleure solution que celle courante.

Faire varier la longueur de la liste ne modifie pas beaucoup la courbe. En effet, si on regarde le nombre de fois que la liste tabou est utilisée, il n'est que de 13 utilisations sur 1500 itérations. Cela montre que pour une solution donnée, il y a beaucoup de voisins.

Exemple n°2 :

Dans cet exemple, nous avons réduit drastiquement le nombre de voisins générés et augmenté la taille de la liste Tabou au nombre d'itérations. Ainsi, une solution générée ne pourra pas jamais "reculer" et sera probablement forcée de prendre parfois de moins bonnes solutions que celle actuelle.

Paramètres	**Valeurs**
Nombre d'itération	1500
Longueur de la liste tabou	1500
Voisins générés	2



Tout d'abord, on remarque que la solution trouvée par la recherche Tabou est moins bonne que toutes les autres méthodes. Bien que la convergence est rapide au début, la recherche atteint un seuil à partir de 500 itérations. On peut aussi voir que les voisins générés ne sont pas toujours bons. En effet, à l'itération numéro ~300, on peut voir une baisse de la finesse, ce qui montre que la solution s'est dégradée sur plusieurs itérations.

Analyse et conclusion :

Après avoir réalisé de nombreux tests sur les différents jeux de données, ainsi que sur les différents paramètres des différents algorithmes, nous nous sommes rendu compte que le problème du bin packing 1D a un paysage très varié et que pour une solution donnée, il y a de nombreux voisins. Cela rend malheureusement l'utilité de la variation de la température peu efficace dans le cadre du recuit simulé. De plus, le fait de partir d'une solution très mauvaise (un item par bin) rend la recherche d'un voisin dégradant difficile au début. Résultat, la convergence est souvent très rapide sur les premières itérations mais va beaucoup dépendre du nombre de voisins générés ensuite. Dans le cas de la recherche Tabou, la convergence est bien plus rapide parce que la meilleure solution est choisie parmi les voisins générés. Etant donné qu'il existe beaucoup de voisins à une même solution, il est difficile de retomber sur une solution précédente (avec l'opérateur inverse donc).

De manière générale, l'algorithme de recherche Tabou est bien plus performant que celui du recuit simulé. Sur l'exemple présenté, on voit que 200 itérations suffisent à obtenir une bonne solution. Le recuit simulé aurait eu un plus grand intérêt si le nombre de voisins possible pour une solution donnée était plus faible.

Pour conclure, le problème du bin packing 1D peut être appliqué aux algorithmes de recuit simulé et de recherche Tabou mais ne les met pas en valeur. Dans le cas du recuit simulé, la gestion température n'est pas très utile et dans le cas de la recherche Tabou, la taille de la liste n'affecte que peu l'algorithme car le bannissement des opérateurs n'est que peu fréquent.