EPU 4

Documentation Mini Projet 2

Complexité et algorithmique

Romain SUMEROT

Aghiles DZIRI

Elie BAROGHEL

Dylan RITROVATO

2017 - 2018

Polytech'Nice Sophia

S O M M A I R E

Introduction

Comme stipulé dans le sujet, on souhaite traiter le problème de bin-packing pour un client industriel. Le langage utilisé comme support est le langage **Java**.

Problème :

On dispose d'un certain nombre d'objets qu'on souhaite ranger dans des containers. L'objectif étant d'utiliser le moins de containers possible. Ce problème est NP-difficile mais il existe quand même des solutions qui permettent d'approcher du résultat parfait.

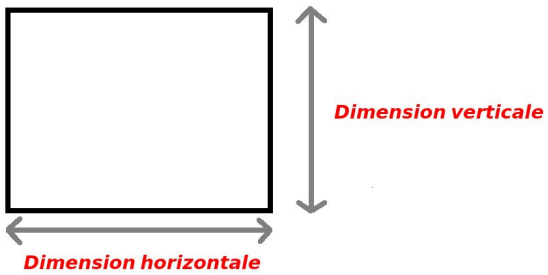


Figure 1 : Représentation d'un container ou d'un objet en 2D

Ce problème prend en entrée un certain nombre d'objets (que l'on note n) ayant un certain volume et on demande le volume souhaité des containers (on ne s'intéresse pas à l'unité du volume et on la considère fixée).

En sortie, on obtient alors un certain nombre de conteneurs qui contiennent ces objets.

Exemple :

Entrée : 4 objets et un conteneur de volume 100

- Objet 1 : Rouge de volume 20
- Objet 2 : Vert de volume 60
- Objet 3 : Bleu de volume 18
- Objet 4 : Jaune de volume 100

Sortie : on obtient par exemple 3 containers représentés sur la figure ci-dessous

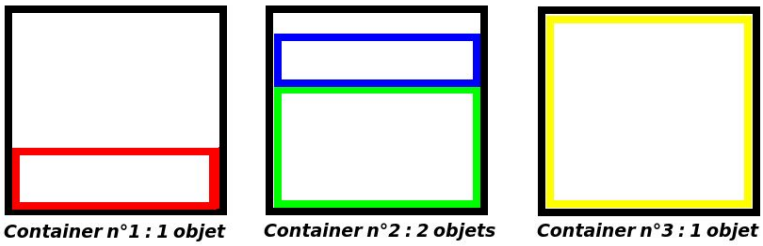


Figure 2 : Représentation de la sortie de l'exemple en 2D

Organisation du rapport :

Ce problème présente de nombreuses solutions. Pour cela, le sujet nous propose 5 algorithmes connus qui répondent à ce même problème mais dont les solutions en sortie sont différentes. L'objectif de ce rapport est de comparer ces algorithmes pour déterminer le plus efficace en terme de rangement et de temps d'exécution.

Ce dossier comporte 4 parties. La première pose les bases des outils et calculs mathématiques nécessaires pour nos calculs de complexités qui nous permettent de discuter de l'efficacité des algorithmes en temps d'exécution. La deuxième partie a pour objectif d'expliquer le principe des algorithmes et d'exposer leur complexité. La troisième partie présente les fonctionnalités implémentées dans notre projet et son mode d'emploi. Enfin, la dernière partie présente nos simulations et compare les algorithmes. Ces 4 parties sont également succédées par une conclusion générale.

1. Base pour les calculs de complexité et conventions

Les complexités sont calculées avec les dominations de fonction (notation grand O). Pour rappel, on dit que f = O(g) s'il existe une constante K positive et un rang M entier positif tels que : $|f(n)| \leq K\,|g(n)|$ pour tout entier $n \geq M$.

Quelques propriétés du grand O :

- Soient f_1 et f_2 deux fonctions telles que $f_1 = O(g_1)$ et $f_2 = O(g_2)$, on a alors : $f_1 + f_2 = O(|g_1| + |g_2|)$ et $f_1 \times f_2 = O(g_1 \times g_2)$.
- si les fonctions f, g, et h vérifient f = O(g) et g = O(h), alors on a aussi f = O(h).

Notations :

Pour les calculs de complexité, on notera **n** le nombre d'objets (également appelés items dans le sujet).

Conventions :

Les exemples fournis en annexe du sujet définissent en entrée un volume qui est identique pour tous les containers. Nos algorithmes ont alors pour convention que chaque container ait le même volume. Cependant, dans notre code, un container est une structure qui possède son propre volume. Il nous faudrait alors peu de changements pour traiter des containers de volumes différents.

2. Présentation des 5 algorithmes de référence et calculs de leur complexité

Structure du code

Nous avons défini une classe Bin, qui représente un container, et une classe Item qui représente un item. Un objet de type Bin est défini par un volume total (maxCapacity), un ensemble d'items (une liste) et le volume total (capacity) qu'occupent l'ensemble de ces items dans le container.
Un objet de type Item est quand à lui défini par son volume (size) et par un id qui permet d'identifier l'item courant.

En ce qui concerne les 5 algorithmes, ils utilisent ces objets. Comme nous programmons en langage Java, nous avons également défini une classe abstraite AbstractBP pour éviter de dupliquer du code comme souhaité dans le sujet. Cette classe abstraite a plusieurs avantages : dans Algo.java et Stat.java, il suffit d'ajouter de nouveaux algorithmes qui héritent de AbstractBP dans la liste des algorithmes à exécuter ou simuler et tout s'exécute de manière automatisée. Il est donc simple de comparer des algorithmes entre eux. Ensuite, cette classe abstraite oblige les 5 algorithmes à utiliser les mêmes objets. Un nouveau développeur dans le projet devra donc suivre la rigueur imposée par AbstractBP.

Ainsi, les 5 algorithmes ont la même entrée : une liste d'Item et le volume total que doit avoir chaque Bin (il est le même pour tous comme expliqué dans nos conventions). Les algorithmes vont ensuite donner leur solution et leur temps de leur exécution.

De plus, dans la spécification, le nombre minimum de Bin est égal à 1. Ainsi, la liste de Bin est initialisée avec 1 Bin vide.

Sont expliqués ci dessous leur principe et les calculs de complexités :

NEXT FIT

On parcourt l'ensemble des items. À chaque itération, on teste si on peut ajouter l'item actuel au dernier Bin enregistré, c'est à dire qu'au dernier Bin de la liste des Bin, si le bin a la capacité suffisante pour accueillir l'item, celui-ci est ajouté. Si ce n'est pas le cas, on ajoute à la liste des Bin un nouveau Bin vide.

Nos listes sont de type ArrayList, l'ajout et get sont donc en O(1). De plus, vérifier si un bin peut accueillir un item revient à comparer 2 int entre eux. La vérification est donc en O(1).

D'autre part, on dispose de **n** items. Comme on parcourt l'ensemble de ces items et qu'on ne fait qu'un ajout d'item et une vérification, on a une complexité totale égale à O(n * O(1)) = O(n) par opération sur les complexités.

Complexité = O(n)

AVL

Le sujet qui nous explique le principe des algorithmes nous informe qu'il est possible d'implémenter les 4 algorithmes restants avec une complexité O(n*log n) à condition d'utiliser une structure adaptée. Nous avons d'abord commencé par implémenter les algorithmes avec une complexité en O(n^2). Comme l'objectif est cependant de comparer par la suite les algorithmes entre eux, nous avons implémenté les algorithmes avec une complexité O(n*log n) grâce à des arbres AVL étudiés l'an passé.

L'intérêt d'utiliser les AVL tree est que ces structures sont connues, fiables, fonctionnelles et leurs complexités ont déjà été étudiées en ASD l'an passé. L'action d'insert dans l'arbre est en O(log k) pour k = nombre de bin. Toutefois comme le nombre de bin peut être égal, au maximum, au nombre n d'item (un bin par item), on considère la complexité de l'action à O(log n).

FIRST FIT

Pour l'algorithme First Fit, on stocke les bins dans un arbre AVL.
On boucle sur chaque items que l'on doit ranger dans les bins.

A chaque tour de boucle, on prend le bin à la racine de l'arbre. S'il n'a plus assez de place pour prendre l'item, on parcourt l'arbre vers la droite (vers les bin avec le plus de place dispo) jusqu'à en avoir un avec l'espace suffisant. Si aucun n'est disponible, on créé un nouveau bin pour prendre l'item que l'on mettra ensuite dans l'arbre.

Le parcours de l'arbre se fait en O(log n) puisqu'on se deplace toujours sur la branche droite. L'action la plus coûteuse en temps à chaque tour de boucle est l'insertion d'un bin dans l'arbre (qui peut se faire après le parcours de l'arbre). Donc on a une complexité de O(log n) par tour de boucle. Si on considère qu'on a n items alors notre complexité est de O(n log n).

Complexité = O(n*log n)

BEST FIT

Cet algorithme ressemble au précédent mais au lieu de s'arrêter au premier bin pouvant contenir l'item, on continue le parcours vers la droite en comparant à chaque fois les bins dispo

pour trouver celui répondant au mieux aux conditions de l'algorithme (avoir une place disponible de taille assez proche de la taille de l'item).
Ça ne change rien à la complexité qui reste à O(n log n).

Complexité = O(n*log n)

WORST FIT

Pour l'algorithme Worst Fit, on stocke les bins dans un Tas binaire (PriorityQueue en Java) classé dans l'ordre décroissant de place restante (le plus vide en haut du tas).

On boucle sur chaque items que l'on doit ranger.

A chaque tour de boucle, on prend le bins au sommet du tas et on vérifie que l'item peut rentrer dedans. Si oui, on l'insère et on remet le bin dans le tas. Sinon, on réinsère le bin, on créé un nouveau bin qui prend notre item et on l'insère dans le tas.
L'action la plus complexe réalisée à chaque tour de boucle est l'insertion dans le tas du bin qui est en O(log n). Comme il y a n items qui répète cette action, on a une complexité de O(n log n).

Complexité = O(n*log n)

ALMOST FIT

L'algorithme Almost Worst Fit est implémenté de manière quasi-identique à Worst Fit. On ne fait juste que quelques retraits et insertions supplémentaire dans la tas à chaque tour de boucle.

Comme l'augmentation de l'utilisation de ces actions reste constante, on peut dire qu'on a une complexité de O(n log n).

Complexité = O(n*log n)

3. Fonctionnalités de notre projet

algo.ex

Ce premier exécutable exécute l'ensemble des 5 algorithmes sur les exemples **"exemple100.txt", "exemple500.txt", "exemple1000.txt" et "monexemple.txt"** qui se trouvent dans le dossier exemple.

Pour chaque fichier d'entrée, on génère le fichier de sortie correspondant (le nom du fichier généré contient "output"). Si un fichier ne s'y trouve pas, un message d'erreur sera affiché dans la console.

Contenu d'un fichier output :

1) On affiche le volume des containers lu par le parser
2) Les items sont numérotés pour pouvoir êtres identifiés et on affiche leur poids correspondant
3) On affiche ensuite les résultats des exécutions des algorithmes :

Ainsi, pour chaque algorithme, on affiche :

Son nom : nombre de containers rangés

La liste des bin rangés où :

bin n°i = [volume total occupé par les items, {Items avec leur poids}]
où volume total occupé par les items = somme des poids de ces items

Son temps d'exécution

Exemple :

Si en entrée on a :

Taille bin
100
Objets
19, 4, 71, 45, 4, 24

Alors en sortie on aura :

Max capacity of the bins : 100

Item n°1 : 19
Item n°2 : 4
Item n°3 : 71
Item n°4 : 45
Item n°5 : 4
Item n°6 : 24
Item n°7 : 9
Item n°8 : 9

NextFit : Size = 2

[94, {n°1:19 n°2:4 n°3:71 }]
[91, {n°4:45 n°5:4 n°6:24 n°7:9 n°8:9 }]

time = 0.012395 ms

FirstFit : Size = 2

[98, {n°1:19 n°2:4 n°3:71 n°5:4 }]
[87, {n°4:45 n°6:24 n°7:9 n°8:9 }]

time = 0.046977 ms

BestFit : Size = 2

[98, {n°1:19 n°2:4 n°3:71 n°5:4 }]
[87, {n°4:45 n°6:24 n°7:9 n°8:9 }]

time = 0.029919 ms

WorstFit : Size = 2

[94, {n°1:19 n°2:4 n°3:71 }]
[91, {n°4:45 n°5:4 n°6:24 n°7:9 n°8:9 }]

time = 0.02698 ms

AlmostWorst : Size = 2

[98, {n°1:19 n°2:4 n°3:71 n°5:4 }]
[87, {n°4:45 n°6:24 n°7:9 n°8:9 }]

time = 0.019555 ms

Stat.ex

Pour comparer les algorithmes, nous avions besoin de comparer leur temps d'exécution ainsi que le nombre que containers rangés. Cependant, des traces d'exécution sur 100 simulations et 1000 items sont dures à analyser pour un humain. D'autant qu'il y a plusieurs façons de générer aléatoirement les données.

Nous avons donc besoin d'histogrammes pour analyser les simulations. Ainsi, pour chaque simulation, 2 histogrammes sont produits : un histogramme qui dresse le temps d'exécution de chaque algorithme et l'autre qui compare le nombre de bin de chaque algorithme.

Enfin, un dernier histogramme général faisant la moyenne des histogrammes est généré. C'est lui qui permet vraiment de faire des analyses.

Les histogrammes sont enregistrés sous format image .jpeg dans le dossier simulations/graphs.

Pour chaque simulation, on enregistre également les traces comme pour Algo.ex dans le dossier simulations/traces. Ce qui nous a permi de vérifier nos histogrammes

Enfin, on demande la manière de générer et différents paramètres pour faire nos statistiques. On demande par exemple le nombre d'items, la taille des bins et la loi de probabilité désirée.

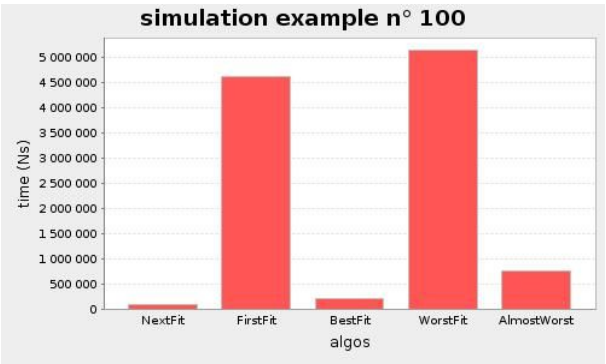
Les bibliothèques utilisées :

JFreeChart qui est une API Java open source permettant de créer des graphiques et des diagrammes. Cela nous a permet de créer des graphiques représentant le nombre de bins et le temps d'exécution pour chaque algorithme. Cette API nous facilite énormément le travaille pour l'affichage des statistiques.

4. Simulations et analyses

Pour permettre de faire des simulation nous avons mis en place des dialogues pour demander à l'utilisateur la taille du container (bin), le nombre d'items à générer et le nombre de simulations. Par la suite nous avons ajouté différentes manière de génération d'exemples et cela avec les lois de probabilités.

Les exemples fournis :



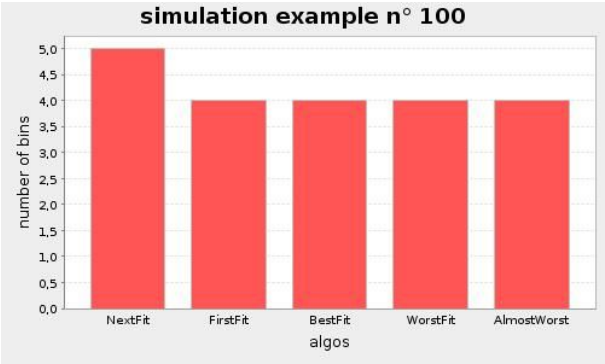
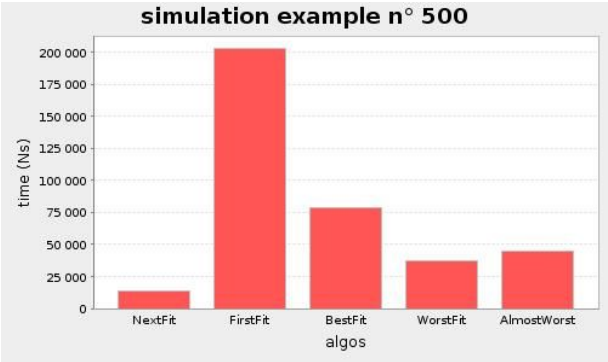


Figure 1: exemple100.txt.



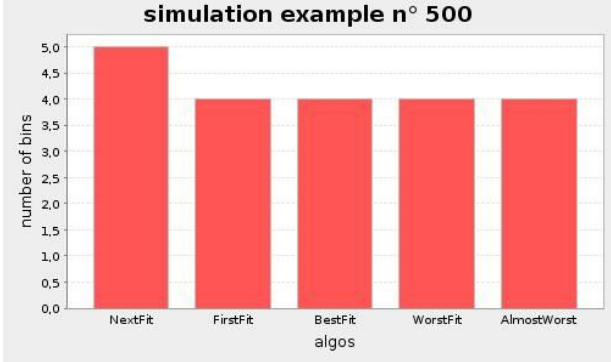
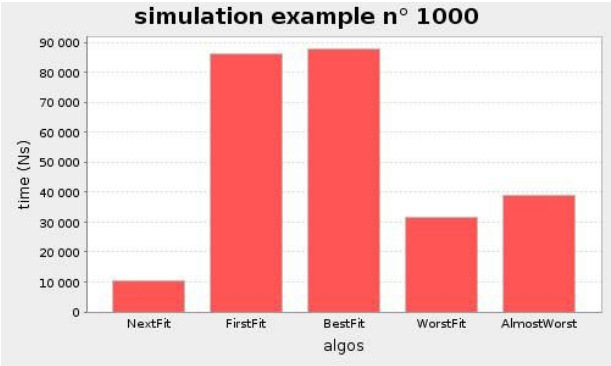


Figure 2: exemple500.txt.



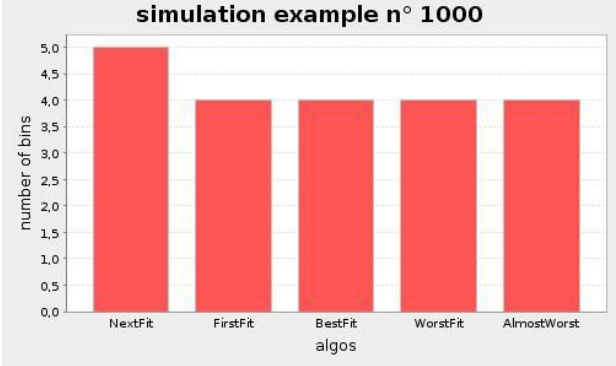


Figure 3: exemple1000.txt.

D'après les exemples fournis, nous constatons que Next fit a une plus grosse marge d'erreur, donc il fait plus de bins (containers) que les autres dans la majorité des cas. Par contre Next fit est plus rapide que les autres, donc il est plus adapté au items qui sont de taille constante.

First fit est plus stable par rapport aux autres, comme on peut voire que Best Fit est rapide quand il s'agit de traitement d'un nombre d'items petit, ici quand on a 100 items Best Fit

est le meilleur si on prend en compte les bins de plus de Next Fit par rapport aux autres. Avec 1000 items on voit vraiment la différence car Best Fit est plus long que First Fit.

Worst Fit est plus adapté aux situation où nous avons besoin de ranger un grand nombre d'items.

Almost Worst devient aussi de plus en plus long au fur et à mesure que le nombre d'items augmente.

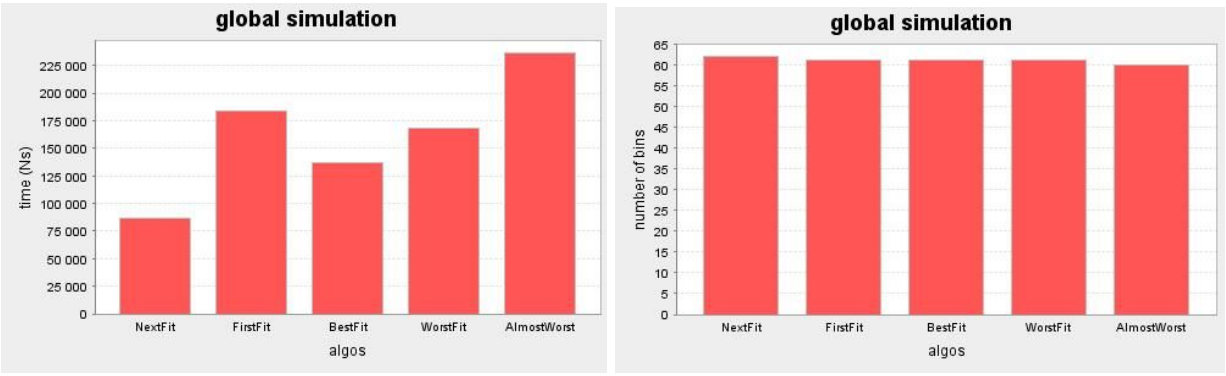


Figure 4:Moyenne de 200 simulations de 1200 items avec la loi de poisson lambda = 5 sizebin=100.

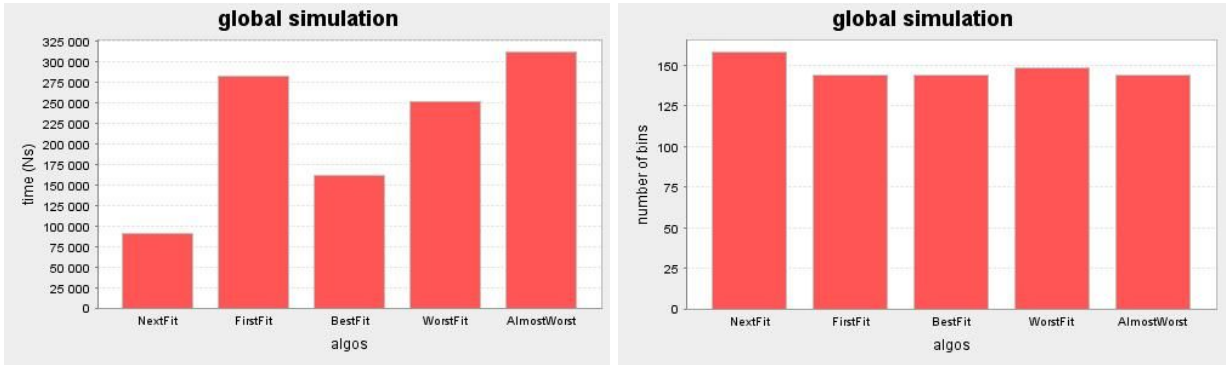


Figure 5:Moyenne de 200 simu de 1500 items avec la loi de Exponnentielle lambda = 0.1 sizebin=100.

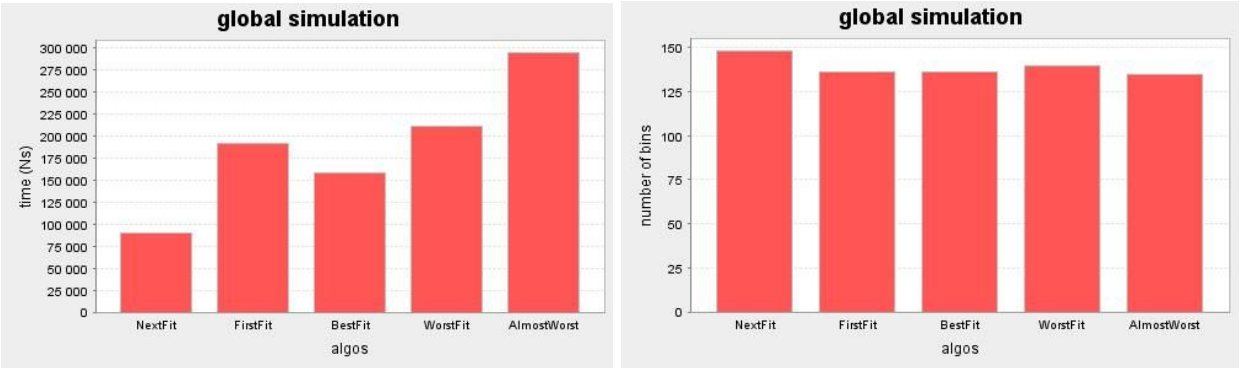


Figure 6:Moyenne de 200 simu de 1500 items avec la loi de Geometrique proba =0.1 sizebin=100 :

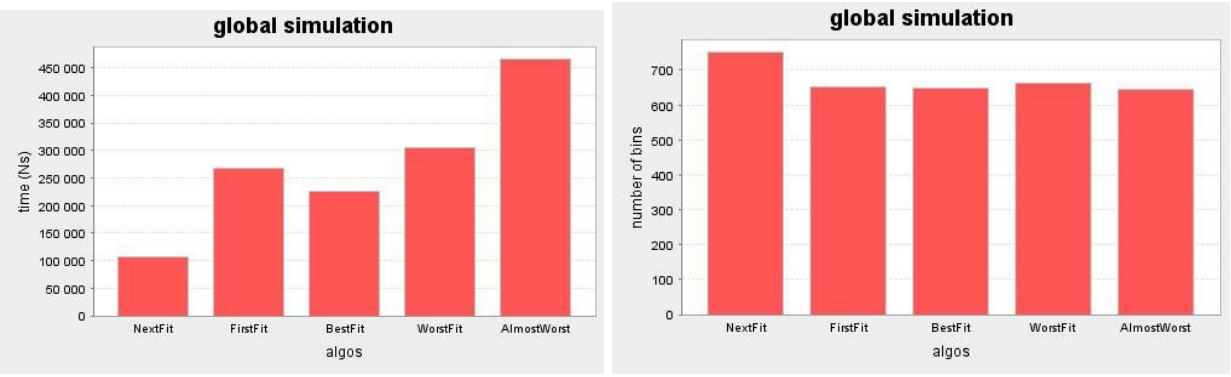


Figure 7:Moyenne de 200 simu de 1500 items avec la loi de Uniforme borneInf = 20 borneSup= 60 sizebin100.

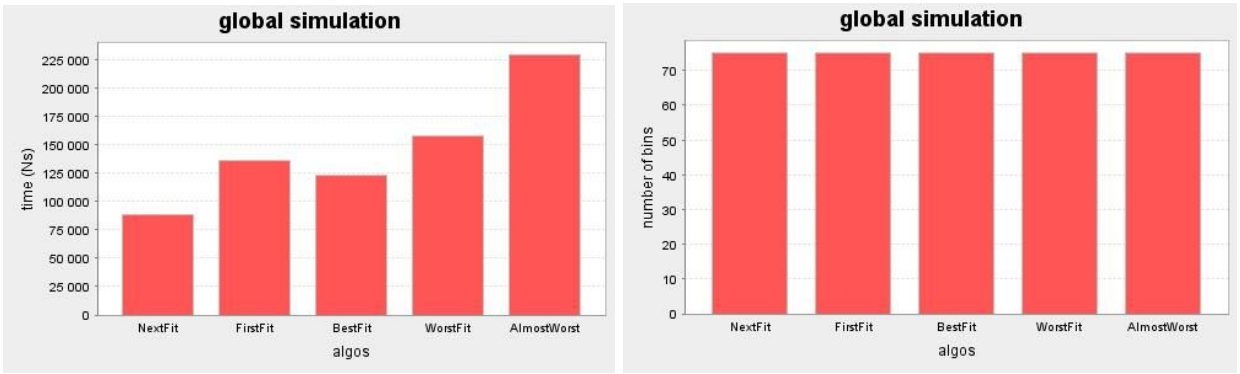


Figure 8:Moyenne de 200 simu de 1500 items avec les items de taille 5 sizebin100.

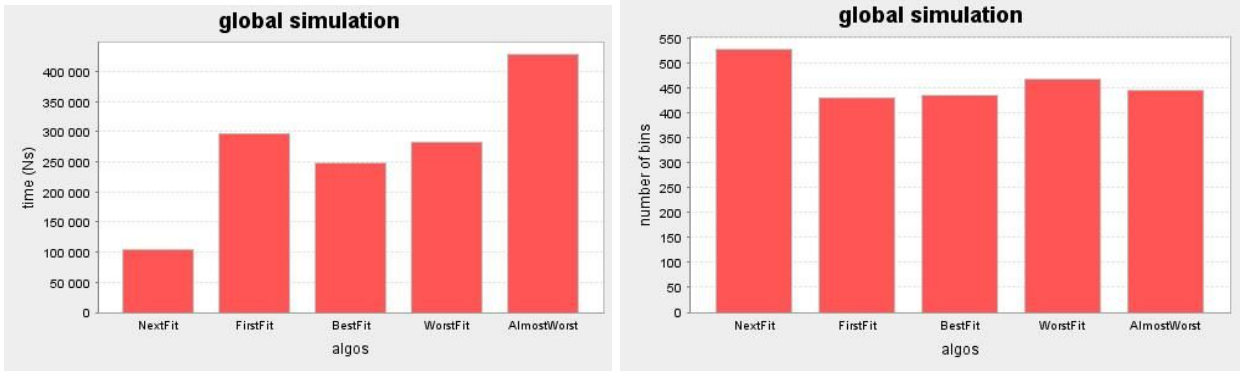


Figure 9:Moyenne de 200 simu de 1500 items avec la loi de Normale moyenne=20 variance= 25 sizebin100.

Avec des moyennes regroupant les mesures de 200 simulations, on fini par obtenir des résultats assez stables et assez représentatifs. En ce qui concerne le score (c'est-à-dire le nombre de bins utilisés), Next Fit est clairement le plus faible des algorithmes, les autres obtenant des scores assez semblables les uns aux autres. En revanche, il se rattrape facilement sur le temps d'exécution qui est beaucoup plus rapide que les autres. Almoste Worst

Fit reste le pire en temps d'exécution mais obtient de meilleurs scores en nombres de bins que Worst Fit (ce qui est le but de cet algorithme).

Finalement, il semblerait que l'algorithme le plus recommandable soit Best Fit qui a le meilleur ratio score / temps d'éxecution. Il est certe deuxième en temps d'exécution par rapport à Next Fit mais ses résultats en nombre de bins sont bien meilleurs. En revanche, Next Fit reste toujours intéressant si tous les items à ranger ont la même taille car il fait le même score que les autres mais avec un bien meilleur temps.

Conclusion

Après de nombreuses simulations, nous avons fait varier le nombre d'items, la taille des bin et nous avons utilisé de nombreuses lois de probabilité afin de couvrir le plus de situations possibles que les industriels peuvent rencontrer.

Deux grands cas ont un vrai intérêt pour les industriels : si les items qu'ils considèrent sont environ de même taille, alors l'algorithme Next Fit est le plus performant en temps et en rangement. En revanche, si un industriel veut faire des chargements d'un grand nombre d'items dans des containers, le meilleur algorithme est Best Fit qui donne la meilleur efficacité de rangement.