IFT800: Projet #1 - Approximation (Bin-Packing)

Shawn Vosburg
voss2502
shawn.vosburg@usherbrooke.ca

Université de Sherbrooke — October 28, 2021

Introduction

Le problème de *bin packing* est un problème algorithmique d'optimisation qui tente de minimiser le nombre de conteneur (*bin* en anglais) nécessaire pour entreposer N items. Pour standardiser le problème, les conteneurs ont souvent une taille maximale de 1 et les N items ont une taille variant entre (0,1]. Les items ne peuvent pas être subdiviser en plusieurs petits items; ils sont atomiques. Les items doivent être tous placés dans les conteneurs, où la somme des items entreposés dans un conteneur ne doit pas être plus grand que la taille du conteneur, soit 1. La solution optimale est celle qui minimise le nombre de conteneur requis pour accomplir cette tâche, dénoté M_{OPT}.

Ce problème a des applications réelles. Par exemple, ce problème est étudié dans l'industrie du transport pour le chargement d'objets dans des conteneurs. Les camions ont un volume fixe et peuvent recevoir qu'un nombre fini d'objet qui ont aussi un volume fixe. Aussi, les camions peuvent avoir un poids maximale aussi. Le volume et le poids sont deux métriques qui peuvent être modélisé par la taille des conteneurs d'un problème de *bin packing* .

En technologie, dans les applications multi-médias, ce problème est rencontré quand il vient le temps de sauvegarder des données sur plusieurs disques durs. Les données sont souvent sous forme de fichier ayant une taille fixe et les disques durs ont un espace de stockage fixe étant plus grand qu'un seul fichier. Il tente alors de minimiser le nombre de disques durs à acheter pour stocker ces données.

Ce rapport résume mon analyse de quelques algorithmes d'approximation dans le cadre du cours *IFT800 - Algorithmique* pour le premier projet de session. Les expériences que je présente dans ce rapport porte sur les algrithmes *First Fit*, *Next Fit*, *Best Fit*, *First Fit Decreasing*, *Next Fit Decreasing* et *Best Fit Decreasing*. Ces algorithmes ont leurs points forts et faibles que je vais explorer dans la prochaine section.

1 Revue de littérature

Le problème de *bin packing* à été démontré d'appartenir à la catégorie de problèmes *NP-hard* [1]. Il n'y a alors pas d'algorithme en temps polynomial pour trouver la solution optimale au problème étant donné une liste L de N items. Cependant, il existe plusieurs algorithmes d'approximation pour trouver une solution faisable en temps polynomial. Cette section résume la recherche fait sur les algorithmes d'approximation explorés dans ce projet pour le problème de *bin packing*.

1.1 Next Fit (NF)

L'algorithme **Next Fit** (NF) est un algorithme *online* qui traite les items séquentiellement. Un algorithme *online* est un algorithme qui doit traiter les items dans l'ordre qu'ils sont dans la liste d'entrée. Cette classe d'algorithme est utile pour informer les décisions quotidiennes sur comment gérer un débit continu d'objet. Par exemple, il a été récemment question de comment mieux distribuer les vaccins de la COVID-19 en minimisant le nombre d'avions requis. Un algorithme *online* tel que NF doit être utilisé si la production de vaccin de chaque usine n'est pas connu à l'avance.

L'algorithme est très simple. Pour un item I de taille t, il place l'item dans le conteneur précédent, si possible. S'il ne rentre pas, il crée un nouveau conteneur et place l'item à l'intérieur. Ce processus est répété pour tous les items. Le pseudocode est disponible ci-bas.

Algorithm 1.1: NEXTFIT(I)

▷ Place les items de L dans le nombre minimum de conteneurs C.

$C \leftarrow \emptyset$

for each $I \in L$

do
- **if** C.dernier().taille() + I.taille() ≤ 1
  - **then** C.dernier() $\leftarrow$ C.dernier().ajouterItem(I)
  - **else**
    - $C_I \leftarrow \{I\}$
    - $C \leftarrow C$.ajouterArrière(C_I)

return (C)

Il est trivial d'implémenter cette algorithme en $\mathcal{O}(n)$, la seule boucle étant celle qui charge les données une à la fois [2].

L'algorithme est une 2-approximation [2]. Le pire des cas est celui où la moitié de l'espace des conteneurs est vide. Voici la famille de cas où NF offre une 2-approximation [2]. Pour les données d'entrée sous forme $L = \{\frac{1}{2}, \frac{2}{N}, \frac{1}{2}, \frac{2}{N}, ...\}$, où N est le nombre de données dans L et N est divisable par 4, la solution optimale serait de regrouper tous les items de taille $\frac{1}{2}$ dans $\frac{N}{4}$ conteneurs et les $\frac{2}{N}$ dans un seul conteneur, pour un total de $\frac{N}{4} + 1$ conteneurs. Alors, nous avons:

$$OPT = \frac{N}{4} + 1$$

Cependant, NF mettrait chaque pair de $(\frac{1}{2}, \frac{2}{N})$ dans un conteneur, pour un total de $\frac{N}{2}$ conteneurs. Cela veut dire que

$$APT = \frac{N}{2}$$

Alors, dans le pire des cas, nous obtenons

$$APP = 2 \times OPT - 2$$

démontrant alors la 2-approximation. Nul autre exemple ne gaspille autant d'espace que cette famille de cas.

Une propriété intéressante de NF est que le nombre de conteneurs retourné par l'algorithme pour une liste d'item $L = \{i_1, i_2, i_3, ..., i_N\}$ est le même que l'inverse de la liste, soit $L' = \{i_N, i_{N-1}, i_{N-2}, ..., i_1\}$ [3].

1.2 First Fit (FF)

L'algorithme **First Fit** (FF) est aussi un algorithme *online*. Il offre une amélioration sur NF en permettant de regarder tous les conteneurs précédant et de choisir le premier qui a assez d'espace pour le prochain item. Il y a alors, en général, moins d'espace gaspillé.

L'algorithme est très simple. Pour un item I de taille t, il place l'item I dans le premier conteneur tel que la somme du contenu du conteneur plus t est au maximum 1. Voici le pseudo-code de l'algorithme:

Algorithm 1.2: FIRSTFIT(I)

▷ Place les items de L dans le nombre minimum de conteneurs.

$C \leftarrow \emptyset$

for each $I \in L$

do
- estPlacé $\leftarrow$ false
- **for each** $C_I \in C$.enOrdre()
  - **do**
    - **if** $(\sum_{t \in C_I} t) +$ taille$(I) \leq 1$
      - **then**
        - $C_I \leftarrow C_I$.ajouterItem(I)
        - estPlacé $\leftarrow$ true
        - break
- **if** estPlacé $=$ false
  - **then**
    - $C_{nouveau} \leftarrow \{I\}$
    - $C \leftarrow C$.ajouterArrière$(C_{nouveau})$

return (C)

FF peut être naïvement implémenté en $\mathcal{O}(n^2)$ en bouclant sur les items et sur les conteneurs [2]. Par contre, il est possible de l'implémenter en $\mathcal{O}(n\log(n))$ avec les structures de données appropriées. Il suffit d'utiliser une variation d'un arbre binaire de recherche (*binary search tree* en anglais) pour entreposer les conteneurs de tel sorte que parcourir une branche spécifique de l'arbre via offre les conteneurs avec la taille d'espace restante en ordre croissant [2].

FF a été démontré d'être une $\lfloor 1.7 \rfloor$-approximation; l'algorithme ne retourne jamais plus que $\lfloor 1.7 \rfloor OPT$ conteneurs [4]. Cette borne est serrée [4]. Pour la preuve, les auteurs démontrent qu'une liste de $10k$ items de taille $\frac{1}{6} - \epsilon$, $10k$ items de taille $\frac{1}{3} - \epsilon$ et $10k$ items de taille $\frac{1}{2} + \delta$ est entreposé dans $17k$ conteneurs sous FF mais que l'optimale est $10k$, pour $\epsilon = 46 * 18^{k-1}\delta$ [4]. Le fait d'ajouter et d'enlever des items à différents endroit dans la liste est exploré pour prouver le ratio d'approximation [4].

1.3 Best Fit (BF)

L'algorithme **Best Fit** (BF) est un algorithme *online*. Il est très similaire à FF à l'exception qu'au lieu de regarder pour le premier conteneur que l'item peut être entreposé, il regarde pour le conteneur où la place restante après l'insertion est minimale. En autre mots, tous le conteneurs sont considérés quand il vient le temps d'insérer un nouveau item. L'ordre des conteneurs n'a pas d'importance.

Pour un item I de taille t, il place l'item dans le premier conteneur de la liste triée en décroissance C où l'item I rentre à l'intérieur. Le pseudocode de BF est disponible ci-bas.

Algorithm 1.3: BESTFIT(I)

▷ Place les items de L dans le nombre minimum de conteneurs.

$C \leftarrow \emptyset$

for each $I \in L$

do
- $C_I \leftarrow C$.rechercheBinaire()
- **if** $C_I \neq NULL$
  - **then**
    - $C \leftarrow C \backslash C_I$
    - $C_I \leftarrow C_I$.ajouterItem(I)
  - **else** $C_I \leftarrow \{I\}$
- \\ *Insertion t.q. C reste trié de façon que l'espace utilisé soit décroissant*
- $C \leftarrow C$.insertionTrier(C_I)

return (C)

Il est trivial d'implémenter BF en $\mathcal{O}(n\log(n))$, en gardant la liste de conteneurs triée en terme d'espace restant [2].

L'amélioration sur FF n'aide pas au ratio d'approximation, qui est aussi $\lfloor 1.7 \rfloor OPT$ [5]. La même famille de cas pour le pire cas de FF peut être utilisé pour trouver le pire cas de BF [5]. BF est alors une $\lfloor 1.7 \rfloor$-approximation.

1.4 Next Fit Decreasing (NFD)

Next Fit Decreasing (NFD) est un algorithme *offline*. C'est-à-dire que la liste L d'item I est complètement connu à l'avance et l'ordre des items peut être changé. La stratégie adopté par NFD est de trier la liste d'items de façon décroissante. Alors, les items de taille plus grands sont placés en premier dans des conteneurs séparés, prêt à recevoir les petits items. Après avoir trier la liste d'items, l'algorithme appelle NF. Le pseudocode est disponible ci-bas.

Algorithm 1.4: NEXTFITDECREASING(I)

▷ Place les items de L dans le nombre minimum de conteneurs.

$L \leftarrow L$.trierDécroissant()

return (NextFit(L))

Il est trivial d'implémenter NFD en $\mathcal{O}(n\log(n))$ avec un algorithme de triage éfficace. L'appel au NF se fait en temps $\mathcal{O}(n)$.

NFD est une 1.691-approximation, tel que NF [6]. C'est aussi un ratio d'approximation serré [6]. La preuve de la limite du ratio étant complexe, elle est omis de ce rapport. Le pire cas de NFD approche un ratio d'approximation de

$$\sum_{i=1}^{\infty} \frac{1}{a_i} = 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{42} + ... \approx 1.691$$

où $a_1 = 1$, $a_{i+1} = a_i(a_i + 1)\ \forall i \geq 1$. La preuve est est faite avec une liste composé de plusieurs sous-ensembles d'items. Chaque sous-ensemble i est composé de M items de taille $\frac{1}{a_i+1} + \epsilon\ \forall i \geq 1$. L'optimal est alors

$$OPT = M$$

tandis que la solution retournée par NFD est alors

$$APT = \sum_{i=1}^{\infty} \frac{M}{a_i} \approx 1.691M$$

Pour la même raison que NF retourne le même nombre de conteneurs pour entreposer une liste L et son inverse, NFD nécessite le même nombre de conteneurs que l'algorithme Next Fit Increasing [3].

1.5 First Fit Decreasing (FFD)

First Fit Decreasing (FFD) est un algorithme *offline*. La liste d'items est premièrement triée de façon décroissante. Ensuite, la fonction First Fit est appelée sur la liste triée. Le pseudocode est disponible ci-bas.

Algorithm 1.5: FIRSTFITDECREASING(I)

▷ Place les items de L dans le nombre minimum de conteneurs.

$L \leftarrow L$.trierDécroissant()

return (FirstFit(L))

Cet algorithme peut être implémenté en $\mathcal{O}(n\log(n))$ car le triage des données et l'appel de FF peuvent les deux se faire en $\mathcal{O}(n\log(n))$.

FFD est une $\frac{11}{9}$-approximation avec la limite de la solution étant $APT \leq \frac{11}{9}OPT + \frac{6}{9}$ [7]. La preuve étant complexe et hors de la portée de ce projet, elle est omise.

1.6 Best Fit Decreasing (BFD)

Best Fit Decreasing (BFD) est un algorithme *offline*. Similairement à FFD, l'algorithme trie initialement la liste d'items pour ensuite appeler BF. Voir le pseudocode ci-bas.

Algorithm 1.6: BESTFITDECREASING(I)

▷ Place les items de L dans le nombre minimum de conteneurs.

$L \leftarrow L$.trierDécroissant()

return (BestFit(L))

Cet algorithme peut être implémenté en $\mathcal{O}(n\log(n)$ pour les même raisons que FFD. BFD offre des performance similaires à FFD et est une $\frac{11}{9}$-approximation [8]. La preuve étant similaire à FFD, elle est omise pour sa complexité.

2 Méthodologie

Mon projet porte sur l'évaluation des algorithmes d'approximation sur le problème de *bin packing*. J'ai décidé d'écrire un programme en C++11 en utilisant que les librairies standards pour évaluer les algorithmes sur différentes métriques. Ce langage a été utilisé car il est possible de faire de la programmation en objet-orienté, il n'est interprété et le langage est facilement disponible sur toutes les platformes. Aussi, le langage est extrêmement rapide, ce qui est utile pour comparé les temps d'exécution. Tout le code que j'ai écrit est disponible avec ce rapport et est disponible sur ma page GitHub.com. Un fichier CMakeLists.txt est aussi disponible pour la compilation facile de mon programme. Voici les différentes composantes de mon programme.

2.1 Génération de données

J'ai implémenté quatre différents générateurs de données. Ils servent à évaluer la performance des algorithmes selon les pires cas de chacun. Aussi, j'ai inclut un générateur aléatoire pour voir la performance moyenne des algorithmes.

1. **Aléatoire**. Les items sont générés d'une distribution de probabilité uniforme avec des valeurs entre 0 et 1. Cette procédure tente de déterminer quel algorithme performe le mieux en moyenne.

2. **First Fit Worst Case (FFWC)**. Cette procédure envoie le pire cas de FF. Il est possible ensuite de vérifier comment les autres algorithmes performe dans ce cas-là. Pour des raisons de simplicité, j'assume que le pire cas est composé de $6C$ items de taille $\frac{1}{7} + \epsilon$, suivi de $6C$ items de taille $\frac{1}{3} + \epsilon$, suivi de $6C$ items de taille $\frac{1}{2} + \epsilon$ items, où $M = 6C$ est le nombre optimal de conteneurs. FF créera donc $\frac{69}{7}C$ conteneurs pour un ratio d'approximation de $\frac{23}{14}$.

3. **Next Fit Worst Case (NFWC)**. Similairement, nous testons la liste d'items qui offre la pire performance de NF. Selon moi, il est intéressant de voir la performance des autres algorithmes sur cette liste pour comprendre l'impact d'utiliser

NF. La liste d'items est alors $L = \{\frac{1}{2}, \frac{2}{N}, \frac{1}{2}, \frac{2}{N}, ...\}$, où N est le nombre d'items. L'optimalité est alors $M = \frac{N}{4} + 1$ conteneurs. NF va créer $\frac{N}{2}$ conteneurs, pour un ratio d'approximation de 2.

4. **First Fit Decreasing Worst Case (FFDWC)**. FFD et BFD ont les mêmes pire cas[5]. Étant donné que ces deux algorithmes on le plus petite ratio d'approximation limitrophe, soit $\frac{11}{9}$, il est intéressant de voir la performance des autres algorithmes, tel que ceux online et NFD, sur leur pire cas. La liste d'items qui donne le pire cas à FFDWC et BFDWC contient les ensembles suivants:

   (a) $B_1 := \{1/2 + \varepsilon, 1/4 + \varepsilon, 1/4 - 2\varepsilon\}$;

   (b) $B_2 := \{1/4 + 2\varepsilon, 1/4 + 2\varepsilon, 1/4 - 2\varepsilon, 1/4 - 2\varepsilon\}$

   Pour une liste contenant $(6k + 4) \times B_1$ et $(3k + 2) \times B_2$, avec une optimatité de $M = 9k + 6$ conteneurs, FFD et BFD vont avoir besoin de $11k + 8$ conteneurs, qui représente une $\frac{11}{9}$-approximation [7].

Chacun de ces ensembles est généré avec 180, 360, 720, 1440, 2880, 5760 et 11520 différents items pour voir la progression du temps d'exécution. En particulier, il serait intéressant de voir la performance de NF, implémenté en temps linéaire, et des autres algorithmes, implémenté pour une performance moyenne en $\mathcal{O}n(\log(n))$.

Ces générateurs de données ont été implémenté grâce à des classes C++ en utilisant le patron de méthode [1]. Chaque algorithme est une classe qui expose des fonctions à la même signature. De cette façon, le code est allégé et plus facile à suivre.

2.2 Les algorithmes d'approximation

Les algorithmes d'approximation sont des classes qui héritent toutes de la même classe abstrait appelée « BaseAlgorithme » qui standardise le constructeur, les méthodes et les membres. Chaque sous-classe de BaseAlgorithme doit définir la méthode *faireBinpacking* qui prends comme paramètres une référence la liste de conteneurs de sortie et une copie

[1] https://refactoring.guru/fr/design-patterns/template-method

des items d'entrée. C'est cette méthode qui est appelée pour obtenir les conteneurs générés part l'algorithme.

Comme mentionné, j'ai implémenté les algorithmes Next Fit, First Fit, Best Fit, Next Fit Decreasing, First Fit Decreasing, et Best Fit Decreasing. Pour FF, j'ai prit le temps nécessaire pour l'implémenter avec une complexité de $\mathcal{O}(n\log(n))$ sur des données aléatoires. Nous allons comparer les temps d'exécution et alors il était inacceptable de l'implémenter en $\mathcal{O}(n^2)$.

2.3 Mésures

Pour comparer les algorithmes, je calcule certaines critères pour chacune des entrées données aux algorithmes. Spécifiquement, je sauvegarde:

1. Le nombre de conteneurs requis.
2. Le nom du générateur de données.
3. La taille utilisée maximale d'un conteneur.
4. La taille utilisée minimale d'un conteneur.
5. La taille utilisée du conteneur médian.
6. Le nombre N d'items dans la liste d'items.
7. Le temps d'exécution en nanosecondes.
8. La taille utilisée moyenne des conteneurs.
9. La variance de l'espace utilisé des conteneurs.

Avec ces mesures, il va être possible d'obtenir une meilleure compréhension de la performance des algorithmes sur les données de plusieurs formes. Ces mesures sont sauvegarder dans un fichier CSV et puis ensuite charger dans un script Python avec un Jupyter Notebook pour la génération de graphiques.

2.4 Tests

Pour s'assurer que les algorithmes sont bien implémentés, j'ai écrit des tests unitaires, disponible dans le répertoire *src/test*. J'ai utilisé la bibliothèque Catch2, offert sous la licence *Boost Software License 1.0*. Ces tests génèrent un deuxième exécutable qui confirme que tout est en ordre et que les algorithmes produisent les résultats escomptés.

2.5 Les expériences comparatives

Pour évaluer les différents algorithmes d'approximation, je vais faire quelques expériences en changeant le générateur de données.

Premièrement, nous allons performer une évaluation de la performance moyenne, en terme de temps et du nombre de conteneurs trouvés, de chaque algorithmes d'approximation sur des données aléatoire. Cette expérience vise à évaluer quel algorithme performe le mieux sur des données qui ne contiennent aucun biais. Le meilleur algorithme serait idéalement celui qui a un temps d'exécution qui grandit le plus lentement et qui offre une solution avec le plus petit nombre de conteneurs. Le rapport va aussi présenter la distribution du nombre de conteneurs trouvé pour chaque algorithme pour voir la robustesse de l'algorithme.

Deuxièmement, nous allons comparer le nombre de conteneur trouvé par chaque algorithmes pour les générateurs de données **FFWC**, **NFWC** et **FFDWC**. Il est intéressant de voir quel algorithme performe le mieux durant les faiblesses d'un d'eux.

Finalement, la performance des algorithmes, en terme de temps d'exécution, sera évaluée pour le générateur de données **FFDWC** car ce générateur mélange l'ordre des items avant de donner la liste aux algorithmes. La performance de FFD devrait toujours être médiocre, car FFD trie la liste avant de s'exécuter, mais les algorithmes *online* peuvent être avantagés ou désavantagés par l'aspect aléatoire.

3 Résultats

Dans cette section, je présente les résultats obtenus sur les expériences décrites dans la section 2.5.

3.1 Données aléatoires

Pour l'expérience avec les données aléatoires, le but est de trouver quels algorithmes performent le mieux en moyenne selon le temps d'exécution et le nombre de conteneurs retournés.

Il est important de se rappeler que chaque algorithme est appelé avec les mêmes données en entrée dans le même ordre, pour une évaluation équitable.

Dans les figures suivantes, il est possible de remarquer quels algorithmes, selon la façon que je les ai implémenté, performent le mieux.

3.1.1 Temps d'exécution

La Figure 1 montre le temps d'exécution de chaque algorithme selon le nombre d'items aléatoires passés en entrée.

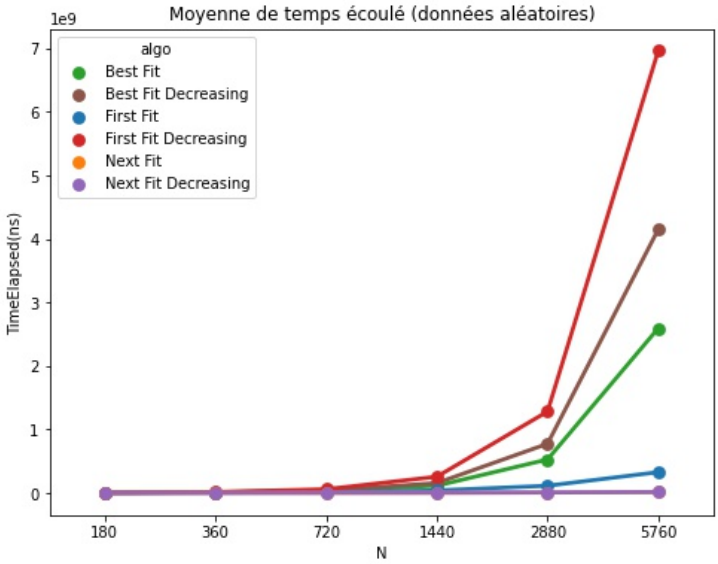

Figure 1: Temps d'exécution sur des données aléatoires en variant le nombre d'items.

Nous pouvons clairement voir que FFD performe horriblement en terme de temps

d'exécution sur de nombreux items. Cela s'explique au fait que j'ai implémenté un arbre de recherche binaire pour trouver le premier conteneur disponible pouvant héberger un item donné le plus rapidement possible, comme expliqué à la section 1.2. Cette arbre de recherche ne sert à rien pour FFD, car tous les items sont en ordre déjà. Pour les premiers items de taille $> \frac{1}{2}$, l'arbre de recherche ne va contenir seulement qu'une seule branche. Chaque nouveau ensemble va devenir un conteneur qui va devenir la racine de l'arbre dans ce cas-ci, et le restant de l'arbre devra être ré-inséré. Alors, il y a beaucoup de calcule et de manipulation de pointeurs et d'objet pour inverser les items de tailles $> 1/2$.

BFD souffre d'un problème similaire. Les items de taille $> 1/2$ sont déjà en ordre, mais l'algorithme passe trop de temps à tenter de garder la liste de conteneur triée. Il y a trop de comparaisons inutile qui augmente le temps d'exécution.

BF garde la liste de conteneurs triée en trouvant le premier conteneur de la liste où l'objet peut rentrer à l'intérieur. Ensuite, une opération de retrait et d'insertion est fait dans la liste de conteneur pour garder la liste triée. Selon la documentation, les opération d'insertion et de retrait sont des opération à complexité $\mathcal{O}$(M), où M est le nombre de conteneurs dans la liste de sortie [2] [3]. Alors, le temps d'exécution est négativement impacté.

FF a été implémenté de façon à ce que chaque item n'est comparé qu'au prochain conteneur ayant plus de place que le précédant. Alors, le temps d'exécution est minimale.

NF et NFD ont tous les deux très bien performés vis-à-vis le temps d'exécution. NF peut facilement être implémenté en temps linéaire et NFD en temps $\mathcal{O}$(nlog(n)), avec un algorithme de triage efficace.

Il est important de noter que le temps d'exécution est principalement dépendant de mon implémentation, qui peut ne pas être optimale.

[2] https://en.cppreference.com/w/cpp/container/vector/insert
[3] https://en.cppreference.com/w/cpp/container/vector/erase

3.1.2 Nombre de conteneur retourné

Figure 2 présente la distribution du nombre moyen d'items dans chaque conteneur, pour chaque algorithme. Étant donné qu'on ne connaît pas l'optimalité, nous pouvons que comparer le ratio moyen d'items par conteneur de chaque algorithme.

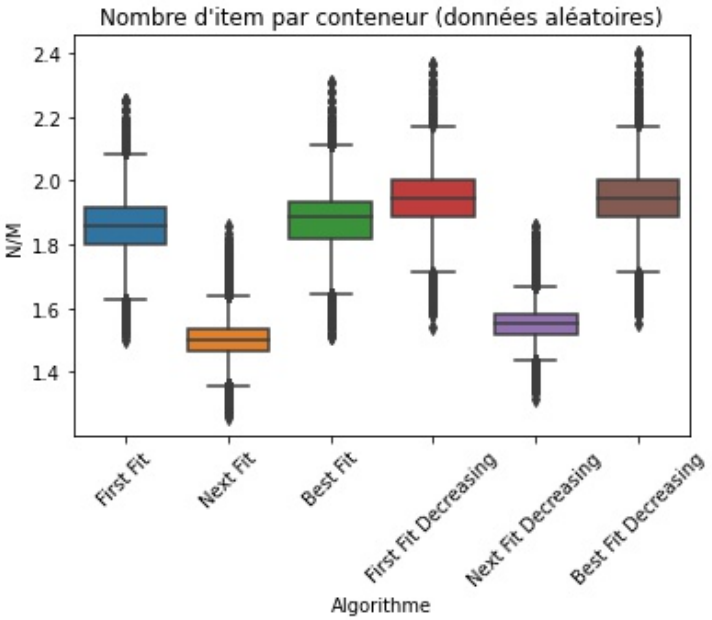

Figure 2: Temps d'exécution sur des données aléatoire en variant le nombre d'item.

Puisque que nous donnons les même données à tous les algorithmes, la figure 2 nous informe du nombre de conteneur retourné par l'algorithme. Le nombre d'items N est partagé par tous les algorithmes et donc le nombre de conteneur retourné M est la seule variable de l'axis vertical qui peut changer. Alors, une grande valeur $\frac{N}{M}$ est désirable car cela veut dire que M est petit.

Nous avons que FFD et BFD performe similairement le mieux, en réduisant le nombre de conteneur requis. BFD offre cependant le meilleur ratio $\frac{N}{M}$, signifiant qu'il y a une exécution où qu'il a réussi à mettre en moyenne 2.4 items par conteneur.

FF et BF offre des performances similaire. Ceux-ci sont moindre que leurs version *offline*.

NF et NFD offrent des piètres performances, ce qui était attendu selon leur construction.

3.2 First-Fit Worst Case (FFWC)

Dans cette section, la performance des algorithmes au pire cas de FF est observée. La création du pire cas de FF est décrite dans la section 2.1. FF offre un ratio d'approximation de $\frac{23}{14}$ [4]. La Figure 3 offre la performance de chaque algorithme.

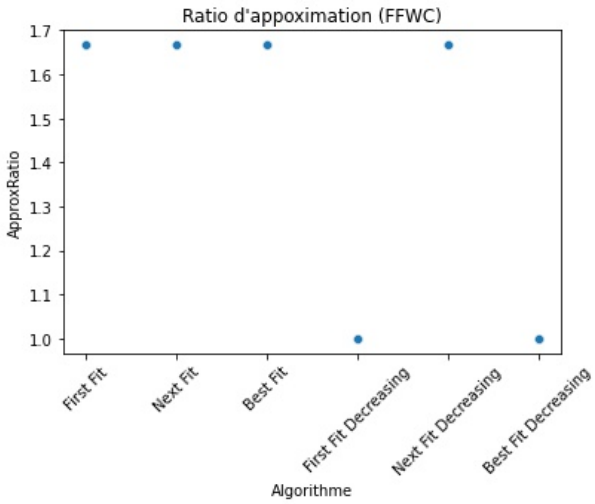

Figure 3: Ratio d'approximation pour les algorithmes sur les données FFWC.

De l'image, on peut voir que FF a un ratio d'approximation extrêmement proche de sa limite théorique, soit $\lfloor 1.7 \rfloor$. Dû à leur similitude, BF va créer les même conteneurs que FF pour le générateur FFWC. NF et NFD vont aussi créer les mêmes conteneurs, donc vont obtenir le même ratio d'approximation de $\lfloor 1.7 \rfloor$.

FFD et BFD trient la liste de données d'entrées en premier lieu et peuvent mettre les items dans les conteneurs précédents. Alors, les items $\frac{1}{7} + \epsilon$ et $\frac{1}{3} + \epsilon$ peuvent remplir les conteneurs contenant un item de taille $\frac{1}{2} + \epsilon$. Alors, l'optimalité est attenit par ces deux algorothimes.

3.3 Next-Fit Worst Case (NFWC)

Dans cette section, je compare les algorithmes d'approximation avec le générateur de données NFWC. Ce générateur offre la pire performance de l'algorithme NF en terme de ratio d'approximation. La liste générée par NFWC est décrite à la section 2.1. La Figure 4 montre le ratio obtenu pour chaque algorithme en faisant varié le nombre d'item en entrée, soit N.

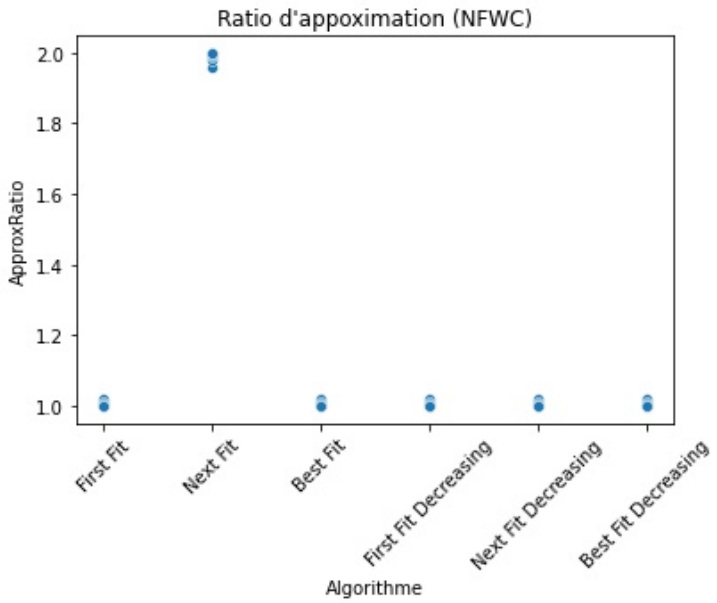

Figure 4: Ratio d'approximation pour les algorithmes sur les données NFWC.

On peut voir que tous les algorithmes performes très bien sur NFWC, à l'exception de NF. Plusieurs points peuvent être observés pour chaque algorithme car il y a une constante dans le ratio d'approximation qui disparaît quand le nombre d'item N devient grand.

La performance des autres algorithmes s'explique par le fait qu'il y a plusieurs façon d'obtenir l'optimalité.

1. Tous les items de taille $\frac{1}{2}$ peuvent être mis dans $\frac{N}{4}$ conteneurs et les $\frac{2}{N}$ peuvent être mis en 1 conteneur.

2. On peut créer deux conteneurs qui contiennent un item de taille $\frac{1}{2}$ et $\frac{N}{4}$ items de taille $\frac{2}{N}$. Les $\frac{N}{2}-2$ items restants de taille $\frac{1}{2}$ peuvent être mis dans $\frac{N}{4}-1$ conteneurs. Le nombre de conteneur total est alors de $\frac{N}{4}+1$.

Les versions *offline* des algorithmes rentrent dans la première catégorie. FF et BF rentre dans la deuxième catégorie.

3.4 First-Fit-Decreasing Worst Case (FFDWC)

Dans cette section, je vais présenter les algorithmes et leurs performances sur les données FFDWC. Le générateur de données FFDWC est expliqué en détail à la section 2.1. Je

vais spécifiquement présenter le temps d'exécution et les ratios d'approximation obtenus. Je vous rappelle que le générateur de données FFDWC contient toujours la même liste d'items mais que l'ordre de ces items est aléatoire. Alors, les algorithmes *online* vont performer différemment à chaque exécution tandis que les algorithme *offline* vont toujours s'exécuter pareillement. Les figures ci-bas démontrent les résultat de l'expérience.

3.4.1 Temps d'exécution

Je vais d'abord parler du temps d'exécution des algorithmes avec les données FFDWC. La Figure 5 présente cette information.

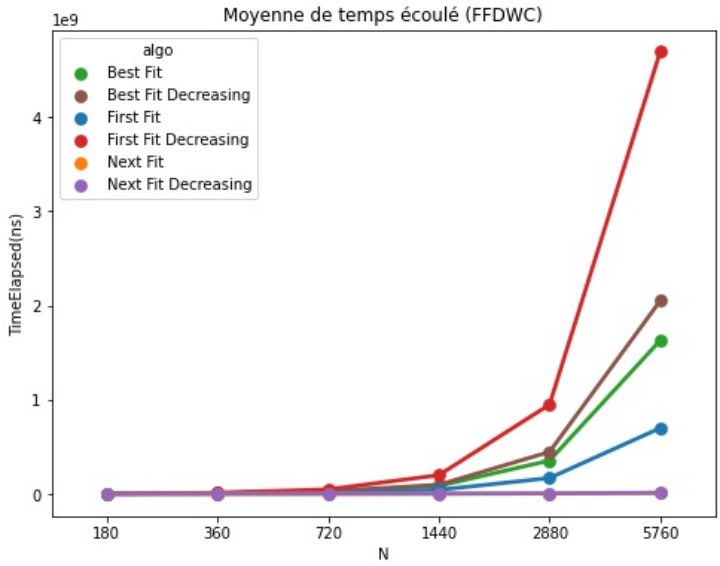

Figure 5: Temps d'exécution pour les algorithmes sur les données FFDWC.

Nous obtenons des temps d'exécution similaire à si les données étaient aléatoire, comme à la Figure 1. Cela voudrait dire que le temps d'exécution n'est pas dépendant de la taille des items mais bien de la quantité d'items, tel qu'attendu.

3.4.2 Ratio d'approximation

Le ratio d'approximation est une mésure importante d'un algorithme d'approximation car il décrit la qualité d'une solution comparé à l'optimalité. La Figure 6 présente le ratio d'approximation de chaque algorithmes sur les données FFDWC.

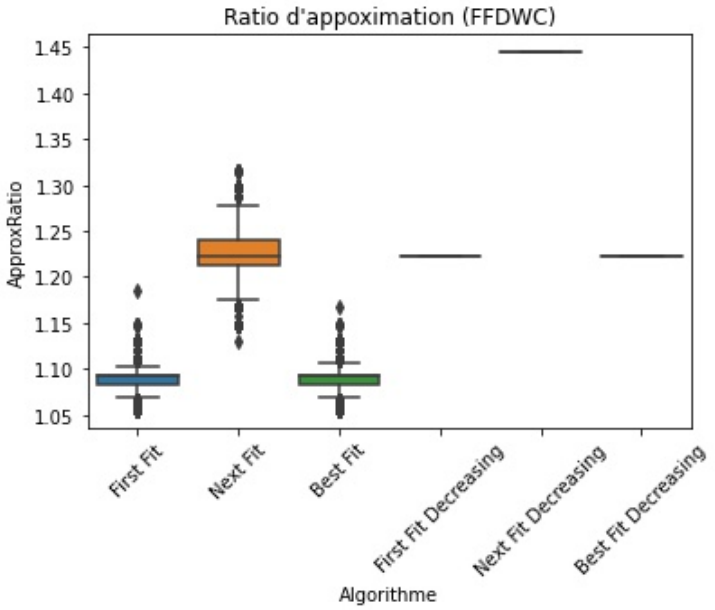

Figure 6: Temps d'exécution pour les algorithmes sur les données FFDWC.

Nous obtenons des résultats intéressants. FFD atteint son ratio d'approximation limitrophe théorique de $\frac{11}{9}$, tel que BFD. NFD offre une pire performance.

Les algorithmes *online* sont généralement mieux dans ce cas, avec FF et BF offrant un ratio d'approximation médian sous 1.1. NF est pire, comme attendu, et offre des une performance médiane comparative à FFD et BFD.

4 Discussion

En terme de temps d'exécution, NF et NFD performe très bien car ils ont une complexité lineaire. BF et FF sont bien mais ont une complexité $\mathcal{O}(nlog(n))$. Leurs versions *offline* performe très mal sur le temps d'exécution car les optimisations qui sont fait pour des données aléatoire *offline* sont souvent inutile avec une liste d'item triée, et ne fait qu'augmenter le calcule à faire. Il serait intéressant de voir le temps d'exécution de BFD et de FFD sans l'optimisation de BF et FF respectivement. En particulier, j'aimerais voir FFD sans l'arbre de recherche binaire que j'ai implémenté pour FF.

Il est important de noter que ceci est ma propre implémentation et n'est pas nécessairement optimale. Le temps d'exécution ne reflète alors pas ce qui peut être optimal en pratique.

Vis-à-vis le ratio d'approximation, NF est généralement le pire algorithme. Ses ratios d'approximation sont souvent très haut comparativement aux autres. Je ne le recommanderais pas alors. Cependant, il performe parfois mieux que les algorithmes *offline* pour les données FFDWC. FF et BF offrent des ratio très similaire.

Après mes recherches, je peux offrir quelques recommandations.

Si la liste d'item arrive en temps réel, je recommande soit FF ou soit BF. Les deux sont toujours très comparable en terme de performance et ils sont mieux que NF. La seule raison d'utiliser NF est si une contrainte de temps important est imposée et que les ressources sont limitées.

Si la liste d'item est complètement disponible au départ, je recommande l'utilisation d'un algorithme *offline* et un algrithme *online* pour être certain d'obtenir la meilleur solution possible. L'utilisation de NFD et de NF devrait être fait seulement si les contraintes de temps et de ressources computationelles sont importants.

5 Conclusion

Le problème de *BinPacking* apparaît dans plusieurs industries tel que le transport et l'architecture informatique. Étant NP-Hard, il est important de trouver des algorithmes d'approximation pour obtenir une solution faisable qui optimise l'utilisation des conteneurs.

Dans cette recherche pratique, j'ai présenté la performance des algorithmes FF, BF et NF et les versions *offline*. NF et NFD performe très bien au niveau du temps mais offre généralement une pire approximation que les autres.

Très peu de différence est visible entre FF et BF, except au niveau du temps d'exécution. Il serait intéressant de passer plus de temps pour tenter d'optimiser ces algorithmes en programmant des structures de données *ad hoc* pour ceux-ci. Par exemple, j'ai utilisé des std::vector pour BF mais ceux-ci aurait peut-être bénéficié de l'utilisation d'une *LinkedList* écrit pour le problème à la place.

FFD et BFD sont des très bons algorithmes qui ont un ratio d'approximation limitrophe

de $\frac{11}{9}$ qui est excellent. Cependant, leur temps d'exécution est souvent très grand, ce qui les rends moins désirable.

J'ai très aimé travailler sur ce projet d'envergure théorique et de l'implémenter en pratique. C'était ma première fois que je comparais des algorithmes qui performent la même tâche. J'ai eu à utiliser plusieurs outils moderne tel que C++, CMake, Python (avec matplotlib) pour mon expérience. J'ai aussi eu à utiliser une machine virtuelle avec Google Cloud pour exécuter mon projet sans monopoliser mon ordinateur portable. Ce fût un bon projet d'apprentissage pour moi, spécifiquement sur comment passer de la théorie à la pratique.

Références

[1] Bernhard H. Korte and Jens Vygen. “Bin Packing”. In: *Combinatorial optimization: Theory and algorithms*. Springer, 2006, pp. 449–467.

[2] Subhash Suri. *Approximation Algorithms*. Nov. 2017. URL: https://sites.cs.ucsb.edu/~suri/cs130b/BinPacking.

[3] David C Fisher. “Next-fit packs a list and its reverse into the same number of bins”. In: *Operations Research Letters* 7.6 (1988), pp. 291–293. ISSN: 0167-6377. DOI: https://doi.org/10.1016/0167-6377(88)90060-0. URL: https://www.sciencedirect.com/science/article/pii/0167637788900600.

[4] György Dósa and Jirı Sgall. “First Fit bin packing: A tight analysis”. In: *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2013.

[5] György Dósa and Jiří Sgall. “Optimal Analysis of Best Fit Bin Packing”. In: *Automata, Languages, and Programming*. Ed. by Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 429–441. ISBN: 978-3-662-43948-7.

[6] B. S. Baker and E. G. Coffman Jr. “A Tight Asymptotic Bound for Next-Fit-Decreasing Bin-Packing”. In: *SIAM Journal on Algebraic Discrete Methods* 2.2 (1981), pp. 147–152. DOI: 10.1137/0602019. eprint: https://doi.org/10.1137/0602019. URL: https://doi.org/10.1137/0602019.

[7] György Dósa. “The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I) 11/9OPT(I) + 6/9”. In: *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Ed. by Bo Chen, Mike Paterson, and Guochuan Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–11. ISBN: 978-3-540-74450-4.

[8] David S Johnson. “Near-optimal bin packing algorithms”. PhD thesis. Massachusetts Institute of Technology, 1973.