Rapport du TP OPTIMISATION

BACHI Yasmine (CdE) SAADI Fatma Zohra Khaoula
NOUALI Sarah MOUSSAOUI Meroua
MIHOUBI Lamia Zohra

22 juin 2020

Table des matières

Introduction

Le problème du bin packing, dans lequel un ensemble d'objets de différents poids doit être rangé dans un nombre minimum de boîtes identiques de capacité C est un problème NP-difficile, c'est à dire qu'il n y a aucune chance de trouver une méthode de résolution qui fournit la solution exacte en un temps polynomiale, sauf si l'égalité NP=P est prouvée. Durant le dernier siècle, divers efforts ont été consacrés pour étudier ce problème, dans le but de trouver des algorithmes heuristiques rapides pour fournir de bonnes solutions approximatives. Dans ce projet, nous allons mettre en place un plateforme de résolution du problème du Bin Packing. pour cela , nous implémenterons 4 types de méthodes :

1. *méthodes exactes :* fournissant la solution optimale, mais qui sont très limités par la taille du problème.
2. *heuristiques :* qui sont des méthodes approchées spécifiques au problème.
3. *métaheuristiques :* qui sont des méthodes approchées génériques.
4. *hybridation d'une métaheuristique avec une recherche locale :* qui est notre contribution principale dans la résolution de ce problème.

Nous commencerons par la présentation du problème, sa formulation mathématique, et une étude des méthodes de résolutions existantes dans la littérature.[Etat de l'Art]. Ensuite, nous présenterons la conception détaillée de chaque méthode implémentée, ainsi que les résultats des test de ces méthodes effectués sur des benchmark connus.[Conception & Tests] On distingue 2 types de tests :

1. *les tests empiriques :* dont le but de trouver la meilleure configuration des paramètres de nos méthodes implémentées.
2. *les tests comparatifs :* où on doit comparer les résultats obtenus des méthodes implémentées et sélectionner la meilleure méthode de résolution pour chaque instances. la comparaison se fait en terme de qualité de la solution et du temps d'exécution.

Première partie

Présentation du Problème de Bin Packing (BPP) :

0.1 Domaines d'Application :

Le BPP a de nombreuses applications dans le domaine industriel, informatique, etc. Parmi lesquelles on trouve :

- Chargement de conteneurs.
- Placement des données sur plusieurs disques.
- Planification des travaux.
- Emballage de publicités dans des stations de radio / télévision de longueur fixe.
- Stockage d'une grande collection de musique sur des cassettes / CD, etc.

0.2 Formulation Mathématique

Etant donné m boites de capacité C et n articles de volume v_i chacun. Soient :

$$x_{ij} = \begin{cases} 1 & \text{article } j \text{ rangé dans la boîte } i \\ 0 & \text{sinon} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{boîte } i \text{ utilisée} \\ 0 & \text{sinon} \end{cases}$$

La formulation du problème donne ainsi le programme linéaire suivant

$$(PN) \begin{cases} Z(min) = \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} x_{ij} = 1 \\ \sum_{j=1}^{n} v_j x_{ij} \leq C y_i \\ y_i \in \{0,1\} \\ x_{ij} \in \{0,1\} \end{cases}$$

La première contrainte signifie qu'un article j ne peut être placé qu'en une seule boite La deuxième fait qu'on ne dépasse pas la taille d'une boite lors du rangement

Deuxième partie

Etat de l'Art :

Après une étude ciblée des travaux existants sur le problème du Bin Packing, nous avons abouti à une synthèse des méthodes de résolution les plus connues –dans chacune des trois catégories : méthodes exactes, et méthodes approchées : heuristiques et métaheuristiques– que nous allons exposer dans ce qui suit.

0.3 Méthodes Exactes

Les méthodes exactes permettent d'avoir des solutions optimales, cependant le temps de calcul peut être très long pour certaines instances du problème. Il n'existe pas un grand nombre de méthodes exactes pour résoudre le problème du Bin Packing , nous allons présenter dans ce qui suit la méthode MTP (basée sur le Branch and Bound) et une méthode de programmation dynamique DP-flow.

0.3.1 Branche and Bound

Cette méthode a été utilisée pour la première fois dans les années cinquante pour résoudre qui a été modélisé par un programme linéaire en nombres entiers. Afin de rendre le processus de résolution plus rapide, cet algorithme utilise une borne inférieure. Plusieurs techniques ont été proposés pour obtenir cette dernière.

Borne inférieure évidente : Soient C la capacité des boîtes utilisés, A l'ensemble des articles a_i de volumes v_i de l'instance I.

$$BI(I) = \frac{\sum_{i=1}^{n} v_i}{C}$$

Borne de Martello and Toth L_2 : Soit α un entier tels que :

$$0 \leq \alpha \leq C/2$$

On definie des classes d'articles suivantes :

$$C_1 = \{a_i, \quad C - \alpha < v_i\}$$

$$C_2 = \{a_i, \quad C/2 < v_i \leq C - \alpha\}$$

$$C_3 = \{a_i, \quad \alpha < v_i \leq C/2\}$$

$BI(I)$ est donnée par la formule suivante :

$$BI(I) = max\{L(\alpha), \quad 0 \leq \alpha \leq C/2\}$$

Avec

$$L(\alpha) = |C_1| + |C_2| + max(0, \left\lceil \frac{\sum_{j \in C_3} v_j - (|C_2| * C - \sum_{j \in C_2} v_j)}{C} \right\rceil)$$

cette borne est calculé en un temps $o(nlogn)$.

Borne inférieure L_3 : Une autre technique a été utilisé dans l'algorithme de résolution MTP pour déterminer une borne inférieure. Soient n_1 le nombre de boîtes obtenus après la première application de la technique de réduction MTP qui consiste à réduire l'instance du problème en rangeant l'article le plus petit, soit Ir^1 l'instance résiduelle de l'instance I après cette première application ie l'ensemble des articles restants après l'opération de réduction

$$L'_1 = n_1 + L_2(Ir^1) \geq L_2(I)$$

On refait ce processus jusqu'à ce que l'instance résiduelle soit vide (i.e. : tous les articles ont été rangés). A l'itération k, on obtiendra :

$$L'_K = \sum_{i=1}^{k} n_i + L_2(Ir^k)$$

La borne inférieure L_3 est obtenue en appliquant la formule suivante , par la suite :

$$L_3 = max\{L'_1, L'_2, \cdots, L'_{kmax}\}$$

Un des algorithmes proposés pour cette méthode est l'algorithme MTP :

L'algorithme MTP (Martello and Toth Procédure) :

Le meilleur algorithme existant pour trouver la solution optimale du problème Bin Packing est celui proposée par *Martello et Toth (Martello & Toth 1990a ; 1990b)* [1] le principe est le suivant : Les articles sont initialement triés selon des poids décroissants. À chaque nœud de décision, le premier élément libre est attribué, à son tour, aux boîtes existantes qui peuvent le contenir (on parcourt les boîtes par ordre de création) et à une nouvelle boîte. À chaque nœud de l'arbre de recherche

a. Une borne inférieure L_3 de la solution restante est calculée et utilisée pour élaguer le nœud de l'espace de recherche et réduire le problème actuel.
   - Si la borne inférieure du nœud actuel est supérieure à la borne inférieure du problème d'origine (nœud racine), le nœud est supprimé. Sinon (b)

b. Des algorithmes approximatifs FFD, BFD et WFD (qu'on présentera dans la partie Méthodes approximatives) sont appliqués au problème actuel, et chacune des solutions approximatives obtenues est comparée à la borne inférieure L_3.
   - Si le nombre de boîtes utilisées par l'une des solutions approximatives est égal à la borne inférieure du nœud actuel, aucune autre recherche n'est effectuée sous ce nœud.
   - Si le nombre de boîtes utilisées dans une solution approximative est égal à la borne inférieure L_3 du problème d'origine (nœud racine), l'algorithme se termine, renvoyant cette solution comme optimale.

De plus, La principale source d'efficacité de l'algorithme de Martello et Toth est une méthode pour réduire la taille des sous-problèmes restants, appelée **critère de dominance**.

0.3.2 Programmation Dynamique

La programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation. Le concept a été introduit au début des années 1950 par Richard Bellman. Elle consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires. Elle s'appuie sur *le principe d'optimalité de Bellman* : une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes. Un modèle pseudo-polynomial simple, pour résoudre le problème bin-packing, est obtenu en associant des variables aux décisions prises dans une table de programmation dynamique (DP) classique.

DP-flow :

Dans le modèle BPP proposé par *Cambazard et O'Sullivan* [2], connu sous le nom de DP-flow, les états DP sont représentés par un graphique dans lequel un chemin qui commence à partir d'un nœud initial et se termine à un nœud terminal représente un remplissage possible d'une boîte. Notons

$(j,d)(j = 0,\cdots,n\,et\,d = 0,\cdots,c)$ un état DP où les articles de 0 à j ont déjà
été étudié (les décisions de placer les articles de 0 à j dans la boîte ou non ont déjà été reprises) et entraînent un remplissage partiel de la boîte de d unités. Notons également par $((j,d),(j+1,e))$ un arc reliant les états (j,d) et $(j+1,e)$. Un tel arc exprime la décision d'emballer ou non l'article $j+1$ à partir de l'état actuel (j,d) : l'état atteint par l'arc est $(j+1,d+w_j+1)$ si l'article $j+1$ est emballé, et $(j+1,d)$ sinon. Soit A l'ensemble de tous les arcs. Comme un remplissage réalisable d'une boîte est représenté par un chemin qui commence à partir du nœud $(0,0)$ et se termine au nœud $(n+1,c)$, le BPP consiste à sélectionner le nombre minimum des chemins qui contiennent tous les éléments.
Pour cette instance BPP, une solution optimale est produite par les deux chemins mis en évidence dans la figure ci-dessous, à savoir
$[(0,0),(1,4),(2,4),(3,7),(4,7),(5,9),(6,9),(7,9)]$
et $[(0,0),(1,0),(2,4),(3,4),(4,7),(5,7),(6,9),(7,9)]$.

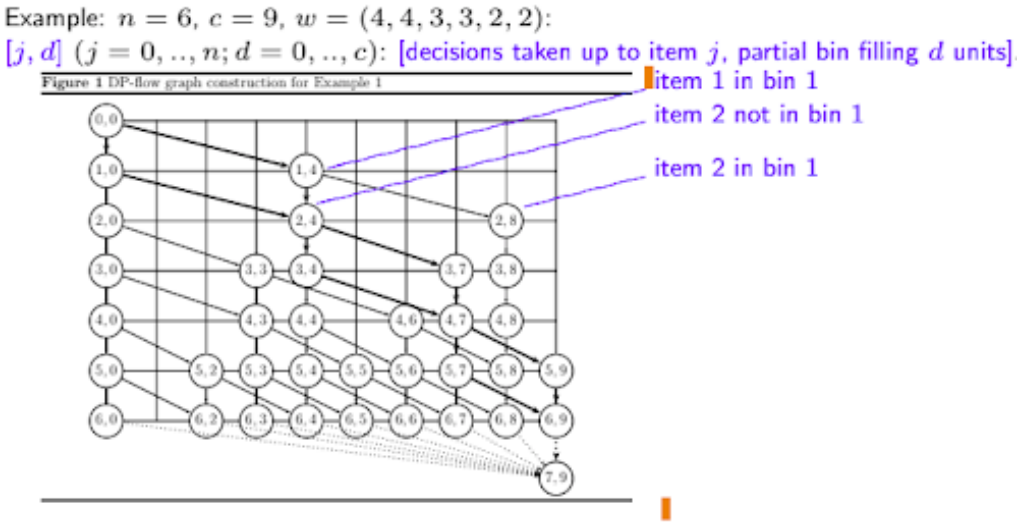


Figure 1 DP-flow graph construction for Example 1

0.4 Heuristiques

Les méthodes approchées sont classées en plusieurs catégories :

0.4.1 Algorithmes ON-LINE :

Ces algorithmes considèrent l'hypothèse que les articles arrivent un à la fois en un ordre connu, chaque article doit être rangé avant de passer à

l'article suivant.

0.4.1.1 Next Fit (NF) :

Lors du rangement de l'article a, NF vérifie s'il tient dans la même boîte que le dernier article (Soit la boîte du dernier article rangé Bj). Si c'est le cas, il place l'article dans la boîte Bj, laissant cette boîte ouverte. Sinon, il ferme la boîte Bj et place l'article a dans une nouvelle boîte Bj+1, qui devient maintenant la boîte ouverte.

Remarque :
Il existe des variants pour cet algorithme, permettant d'améliorer ses performances :

— **Next-KFIT (NFk) :** autorise k boîte ouvertes à la fois, c'est à dire qu'il vérifie dans les K boîtes ouvertes s'il y a assez d'espace pour ranger l'article a, sinon il ouvre une nouvelle boîte. Si K=1 on retombe sur l'algorithme NF.

0.4.1.2 First Fit (FF) :

Lors du rangement de l'article a, NF vérifie s'il tient dans la même boîte que le dernier article (Soit la boîte du dernier article rangé Bj). Si c'est le cas, il place l'article dans la boîte Bj, laissant cette boîte ouverte. Sinon, il ferme la boîte Bj et place l'article a dans une nouvelle boîte Bj+1, qui devient maintenant la boîte ouverte.

0.4.1.3 Best Fit (BF) :

L'article a est rangé dans une des boîtes ouvertes de sorte que le plus petit espace vide soit laissé. Si l'article ne tient dans aucune boîte existante, il sera placé dans une nouvelle boîte.

Remarque :
Il existe des variants pour cet algorithme, permettant d'améliorer ses performances :

◦ **K-Bounded Best Fit (BBFk) :** utilise le même principe que Best Fit, sauf qu'il restreint le nombre de boîtes ouvertes à k boîtes. ie : l'article a est rangé dans une des k-boîtes ouvertes de sorte que le plus petit espace vite soit laissé.

0.4.1.4 Worst Fit (WF) :

L'article a est rangé dans la boîte avec le plus grand espace vide, si cette dernière ne peut pas contenir l'article, il sera placé dans une nouvelle boîte.

Remarque :
Il existe des variants pour cet algorithme, permettant d'améliorer ses performances :

- **Almost Worst Fit (AWF) :** l'article a est rangé dans la 2ème boîte la plus vide, jusqu'à ce qu'il reste qu'une seule boîte qui peut contenir l'article. Dans ce cas il est placé dans cette boîte. Si l'article ne tient dans aucune boîte existante, il sera placé dans une nouvelle boîte.

0.4.1.5 Harmonic K (Hk) :

Cet algorithme est basé sur une partition de l'intervalle $[0, 1]$ en K sous-intervalles I_k, où $I_k =]\frac{1}{K+1}; \frac{1}{K}]$, à chacun de ces sous-intervalles correspond une seule boîte ouverte, et seuls les articles appartenant à ce sous-intervalle (i.e. : $v_i/C \in I_k$ avec v_i le volume d'un article) sont regroupés dans cette boîte. Si un nouvel article arrive et ne rentre pas dans sa boîte ouverte correspondante, la boîte est fermée et une nouvelle boîte est ouverte.
Remarque :
Il existe des variants pour cet algorithme, permettant d'améliorer ses performances :

- **Simplified HarmonicK (SHk) :** se base sur une structure d'intervalle qui est plus compliquée.

0.4.1.6 Autres algorithmes :

- **ABFk :** utilise le principe de rangement du Best Fit avec le principe de fermeture des boîtes du First Fit.
- **AFBk :** utilise le principe de rangement du First Fit avec le principe de fermeture des boîtes du Best Fit.

Remarque :
Dans le cas où $K = 1$, ces algorithmes sont équivalent à l'algorithme Next Fit.

0.4.2 Algorithmes OFF-LINE :

Dans ce type d'algorithmes on a accès à tous les articles avant de commencer le rangement dans les boîtes. *Quelques algorithmes off-line :*

0.4.2.1 First Fit Decreasing (FFD) :

Ordonner les articles par ordre décroissant des poids, ensuite appliquer l'algorithme First Fit.

0.4.2.2 Best Fit Decreasing (BFD) :

Ordonner les articles par ordre décroissant des poids, ensuite appliquer l'algorithme Best Fit

0.4.3 Algorithmes SEMI-ONLINE :

Les algorithmes dits semi-en ligne (SOL) se situent entre les online et les offline. Ils relâchent la prescription online de manière à permettre quelques opérations supplémentaires où l'algorithme a un peu de connaissance de l'avenir, au moins une des opérations suivantes est autorisée : comme reconditionner un nombre fini d'articles déjà emballés, prétraiter les articles en les commandant en fonction des tailles ou en tamponnant certains articles avant de les emballer. *Quelques algorithmes semi-online :*

0.4.3.1 MMP (Mostly Myopic Helps) : Fully Dynamic Algorithm

Dans cet algorithme [3], en partant de l'hypothèse que l'emballage peut être réarrangé arbitrairement pour accueillir les articles arrivant et partant, on suppose :

- l'emballage d'un article se fait en une abstraction totale des articles déjà emballés d'une taille plus petite(c'est à dire, on suppose que leurs places dans les boîtes sont vides, où le nouvel article pourra être affecté dans cet espace, et ces articles de petites tailles pourront être réarrangés).
- Regroupement des articles plus petits dans des lots(un groupe d'articles plus petit que ϵ peut être déplacé comme un seul article).
- Le nombre d'articles uniques ou de lots de très petits articles qui doivent être réarrangés est délimité par une constante.

Cet algorithme nécessite du temps $\Theta(\log n)$ par opération (c'est-à-dire pour une insertion ou une suppression d'un article). Il est presque aussi bon que celui des meilleurs algorithmes offlines pratiques.

0.4.3.2 Harmonic Fit avec (4 ou 6) partitions :

Le HF avec 4 partitions [4], en utilisant une structure de données appropriée, permet de traiter de grandes collections de petits articles en :

- les groupant et les déplaçant tous à la fois en temps constant
- les classifiant en 4 types par taille
- ne permettant au plus que les déplacements de 3 lots d'articles ou d'un seul article lorsqu'un nouvel article doit être attribué

Le HF avec 6 partitions [4] utilise :

- six types d'articles classés par taille
- au plus 7 mouvements de lots/ article par nouvel article

0.4.4 Algorithmes d'Espace Borné (Bounded Space) :

les algorithmes décident où un élément doit être emballé sur la base du contenu actuel d'un nombre fini k de bacs, où k est un paramètre de l'algorithme. Notez que FF et BF ne sont pas des algorithmes d'espace bornés, mais NF l'est, avec $k = 1$.

0.5 Métaheuristiques

0.5.1 La recherche tabou :

Une méthode de résolution a été proposée par Fernandes Muritiba Et al.[5] ,qui prend en entrée une population initiale obtenue par une heuristique et applique un opérateur de croisement sur ces solutions .Chaque solution obtenue est améliorée par la suite en utilisant une recherche tabou, qui consiste à se déplacer dans un espace de recherche contenant des solutions réalisables partielles où certains articles ne sont pas affectés à des boîtes. L'amélioration consiste de passer d'une solution partielle de valeur K à une solution complète de la même valeur. La fonction objective utilisée est basée sur la somme pondérée des tailles des articles. Pour la diversification de l'espace de recherche, On utilise une procédure basée sur un opérateur de croisement.

0.5.2 L'algorithme ILWOA (Improved Lévy WOA) :

L'algorithme WOA (Whale Optimization Algorithm) [6] est basé sur une méthode inspirée de la nature plus exactement d'une stratégie d'alimentation des baleines à bosse, où la recherche des proies représente l'exploration de l'espace de recherche et la libération des bulles représente l'exploitation. Une amélioration de cet algorithme a été proposée par Abdel-Basset, M., Manogaran, G., Abdel-Fatah, L. et al [7].Il s'agit de l'utilisation de fonctions logistiques et d'autres méthodes probabilistes pour assurer une convergence plus rapide

0.5.3 L'algorithme FFA (FireFly Algorithm) :

L'algorithme FireFly (FFA) [8] est une métaheuristique génétique, ins-

pirée par le comportement clignotant des lucioles. Le but principal du flash d'une luciole est d'agir comme un système de signal pour attirer d'autres lucioles. Il y a trois règles. La première règle, chaque luciole attire tous les autres lucioles avec des flashs plus faibles. Deuxièmement, l'attractivité est proportionnelle à leur luminosité qui est inversement proportionnelle à leurs distances. Pour deux lucioles clignotantes, la moins brillante se déplacera vers la plus brillante. L'attractivité est proportionnelle à la luminosité et ils diminuent tous les deux à mesure que leur distance augmente. S'il n'y en a pas plus brillant qu'une luciole particulière, il se déplacera de façon aléatoire. Enfin, aucune luciole ne peut attirer la luciole la plus brillante, cette dernière se déplace d'une façon aléatoire. Le FFA est construit par analogie, en appliquant ces trois règles sur une population de solutions initiale pour la faire évoluer en une population contenant la solution approchée.

Troisième partie

Méthodes Exactes :

Chapitre 1

Conception :

Dans ce chapitre,nous allons présenter la conception détaillée des méthodes exactes sur lesquelles notre choix d'implementation s'est porté :

1. Le branch and bound
2. Une version améliorée du branch and bound
3. La recherche exhaustive
4. La programmation dynamique

Dans le but de montrer l'applicabilité de ces méthodes, comparer leurs performances et montrer leurs limites, nous effectuerons des tests empiriques et comparatifs sur des benchmarks d'un coté, et sur des instances générées par notre propre générateur d'instances d'un autre coté.

1.1 Branch and bound

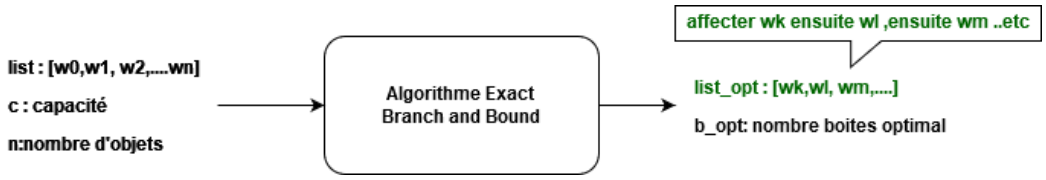
— L'algorithme Branch-and-Bound (B &B) que nous avons implémenté tente de ranger un objet à la fois en fonction de l'ordre initial des objets.

— Au niveau j de l'arbre, B &B crée un noeuds fils pour chaque boite ouverte et range l'objet j dans cette boîte si c'est possible. il crée aussi un noeuds supplémentaire qui représente l'ouverture d'une nouvelle boîte, et il range l'objet j dans cette boîte.

— En pratique, au niveau 1 de l'arbre l'objet 1 est rangé dans la boîte 1 , au niveau 2 l'objet 2 est rangé dans la boîte 1 ou dans une nouvelle boîte 2 ,...etc

— A chaque noeud, on résout un sous problème de taille (n-k) du bin packing, où les k premiers objets ont déjà été emballés.

— L'opération du rangement d'un objet i au niveau k consiste à permuter entre les éléments list(K) et list(i). On va avoir comme sortie une liste d'objets ordonnées selon l'ordre de rangement, il suffit ensuite de remplir les boîtes par les objets dans leur nouvel ordre pour générer la solution (l'emplacement de chaque objet dans les boîtes)



1.1.1 Pseudo-Code

Soient :

— n : le nombre d'articles

— list[0... n-1] : la liste des articles en entrées

— opt_list : la list ordonnée fournissant la solution optimale en sortie

— opt_cost : le nombre de boîtes optimal

— C :la capacité maximale d'une boîte.

L'algorithme proposé est une fonction récursive packBins ayant comme paramètres :

— k :l'ordre de l'élément à être ranger (le niveau dans l'arbre).

— sumwt :la somme des poids des éléments restants à être rangés

— bcount :la somme cumulée des boîtes déjà utilisées (depuis la racine jusqu'à ce nœud)

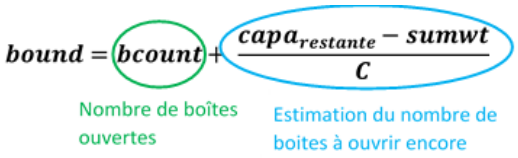
— capa_restante :l'espace libre restant dans la boîte ouverte.

Algorithm 1 Branch & Bound

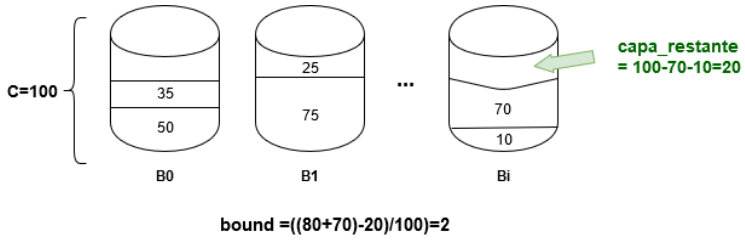
if $n = k$ **then**
  //noeud feuille (n objets rangés)
  if $bcount < opt\_cost$ **then**
    //solution exacte obtenue par cette branche est meilleure que celle trouvée auparavant
    màj du coût optimal $opt\_cost = bcount$.
    sauvegarder la solution liste $opt\_list = liste$.
  else
    continuer le parcours (monter d'un niveau dans l'arbre).
  end if
else
  //L'ensemble des articles restants sont dans les positions list[k... n-1].
  //chaque nœud fils i signifie qu'on a rangé le ième article parmi les articles restants (ayant la position k+i dans la liste list) à la position k.
  for chaque nœud fils i **do**
    Mettre l'article list[k+i] dans une boîte en permutant l'article[k+i] avec l'article[k] : permuter(k+i,k) .
    Incrémenter le nombre de boîtes utilisées (bcount) si on a ouvert une nouvelle boite.
    Mettre à jour la capacité restante (capa_restante).
    Mettre à jour la somme des volume des articles restants à être ranger (Sumwt)
    Calculer l'évaluation du nœud fils courant (borne L1) : $Bound = bcount + \frac{(sumwt - capa\_restante)}{C}$
    Comparer l'évaluation du nœud avec la solution optimale courante :
    if $bcount \geq opt\_cost$ **then**
      //solution exacte obtenue par cette branche est meilleure que celle trouvée auparavant
      le nœud est éliminé. Dans ce cas on re-permute pour revenir à l'état précédant (permute (k+i,k)).
      sauvegarder la solution liste $opt\_list = liste$
    else
      on exploite le nœud courant encore, en faisant un appel récursif à la fonction avec la valeur k+1 dans le 1 paramètre,en utilisant les nouvelles valeurs des autres paramètres.
    end if
  end for
end if

Evaluation d'un noeud (Borne L1)

L'évaluation d'un nœud est calculée en sommant 2 parties, le nombre de boîtes déjà utilisées bcount et une estimation du nombre de boîtes qu'on va ouvrir encore pour contenir les objets restants. Cette estimation est obtenue en divisant la somme des poids restants sumwt sur la capacité d'une boîte. On soustrait de la somme des poids restants, l'espace vide restant dans la dernière boîte ouverte, car ce dernier peut contenir des objets. On obtient ainsi la formule suivante :



exemple 1 : List= 10,50,25,80,70,75,35,70 ; C = 100

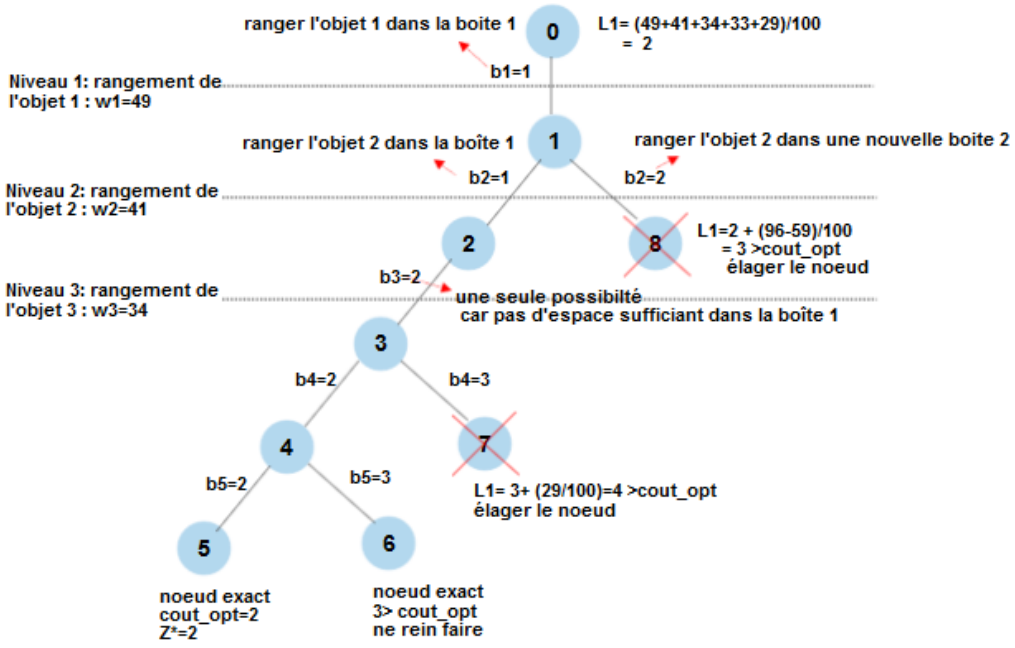


exemple 2 : n= 5 ; W_j=49,41,34,33,29 ; c=100 on pose : b_j= le numéro de boîte qui contient l'objet j.

1.2 Branch and bound amélioré

Une version améliorée de l'algorithme Branch and Bound présenté ci-dessus. L'amélioration s'est faite en 2 étapes :

1. Utilisation de l'heuristique WFD (Worst Fit Decreasing) pour initialiser la solution optimale.
2. Changement de la borne L1 utilisée par une autre borne plus puissante appelée L2.



Evaluation d'un noeud (Borne L2)

Il a été prouvé que la borne L1 n'est efficace que quand les poids des objets sont petits, c'est à dire qu'on peut mettre plusieurs objets dans la même boîte. Si ce n'est pas le cas, et que les objets ont de grands poids (proches de C) , cette borne n'aura aucun effet et l'algorithme fera une recherche exhaustive. C'est pour cela que la borne L2 à été proposée par Martello et Toth ,pour remédier à ce problème. on rappelle la formule de la borne L2, qui a été déjà présentée dans l'état de l'art :

rappel Soit α un entier tels que :

$$0 \leq \alpha \leq C/2$$

On definie des classes d'articles suivantes :

$$C_1 = \{a_i, \quad C - \alpha < v_i\}$$

$$C_2 = \{a_i, \quad C/2 < v_i \leq C - \alpha\}$$

$$C_3 = \{a_i, \quad \alpha < v_i \leq C/2\}$$

$BI(I)$ est donnée par la formule suivante :

$$BI(I) = max\{L(\alpha), \quad 0 \leq \alpha \leq C/2\}$$

Avec

$$L(\alpha) = |C_1| + |C_2| + max(0, \lceil \frac{\sum_{j \in C_3} v_j - (|C_2| * C - \sum_{j \in C_2} v_j)}{C} \rceil)$$

Explication de la formule : Etant donnée que les objets des classes C_1 et C_2 ont un poids supérieur à $C/2$ chacun d'eux sera placé dans une boîte séparée pour le contenir, donc $|C_1| + |C_2|$boîtes sont utilisées quelque soit la solution. De plus,aucun objet de l'ensemble C_3 ne peut être rangé dans une boîte contenant un objet de C_1 (à cause de la contrainte de capacité). La capacité résiduelle (espace libre) des $|C_2|$ boîtes est de : $C^* = |C_2| * c - \sum_{j \in C_2} w_j$ Donc dans le meilleur des cas, cette capacité résiduelle va être remplie par les objets de C_3, et dans ce cas le nombre de nouvelles boîtes qu'on doit ouvrir est de : $\frac{\sum_{j \in C_3 - C^*} w_j}{c}$ (cette dernière formule utilise le même principe que la borne L1).

1.2.1 Pseudo-Code

Algorithm 2 Branch and bound amélioré

Appliquer WFD sur l'instance pour initialiser le coût optimal : cout_opt=WFD(problème)
Appliquer l'algorithme Branch and Bound sur le problème en utilisant la borne L2

1.3 Recherche exhaustive

Dans cette 3ème solution, on a implémenté une recherche exhaustive, qui consiste à parcourir l'ensemble des nœuds et leurs fils, sans aucun élagage de nœuds. Donc on aura le même algorithme que celui du branch and bound, en supprimant l'étape de l'évaluation du nœud pour décider de son élagage.

1.4 La programmation dynamique

principe Étant donnée une liste L de N articles à ranger et la capacité d'une boîte C :

structure de donnée :

— La table de vérité m : une matrice de (C+1) colonnes, et N lignes, tel que m[i][j] désigne si l'article i peut être rangé dans une capacité j.

Etapes :

1. Remplir la table de vérité correspondante au problème actuel (L,C), puis ouvrir une nouvelle boite
2. En parcourant la table de vérité, choisir les articles à mettre dans cette nouvelle boite.
3. Ranger les articles choisis dans la boîte, et mettre à jour la liste L (retirer ces articles de L).
4. Réitérer le processus jusqu'à avoir rangé tous les articles (N=0)

1.4.1 Pseudo-Code

1.4.1.1 Méthodes utilisées :

la méthode : get_truth_table(capacité, items) :

paramètres :

— capacité : la capacité d'une boîte

— items : la liste des articles de poids W_j à ranger

Rôle : Création et initialisation de la table de vérité

Algorithm 3 get_truth_table(capacité, items)

```
Initialiser toutes les cases de la table de vérité m à "true"
Parcourir les articles un à un //les lignes de la matrice m
for chaque article i do
  for chaque colonne j do
    ///parcourir la capacité une unité par une unité
    if i = 0 then
      //premier article
      if j > 0 et j ≠ W_j then
        //c'est à dire cette capacité ne pourra pas accueillir l'article courant
        Mettre "faux" dans la case
        //revient à remplir la première ligne par "false", sauf la case1 et la case qui correspond au volume de l'article
      end if
    else
      //pour le reste des articles, utiliser la relation suivante sur la table
      if j < W_j then
        m[i][j] = m[i-1][j] //la valeur de la case en dessus
      else
        m[i][j] = m[i − 1][j] ∨ m[i − 1][j − (W_i)]
      end if
    end if
  end for
end for
return m
```

_pick_items(m)

paramètres :

— m : table de vérité

Rôle : Choisir les articles à mettre dans la boîte

Algorithm 4 _pick_items(m)

K = (nombre de lignes de m) - 1 // indice de la dernière ligne
Initialiser la liste des indices des articles choisis à la liste vide : picked_items_indices = []
if $k \geq 0$ **then**
  $k = max(jtelquem[k][j] = true)$ //prendre la plus grande capacité totale
end if
while $k \geq 0$ **do**
  //tant qu'il nous reste encore de lignes à visite
  if $k = 0 et j > 0$ **then**
    //première ligne
    ajouter l'article k à picked_items_indices
  else
    if $m[k-1][j] = false$ **then**
      ajouter l'article k à picked_items_indices
      j = j - W_k
    end if
  end if
  $k = k - 1$ //allez à la ligne de dessus
end while
return picked_items_indices

la méthode : _move_items_to_bin(list_of_items_indices, bin_index)

paramètres :

— list_of_items_indices : liste des indices des articles de poids W_j

— bin_index : le numéro de la boîte de destination

Rôle : Ranger les articles dans la boîte

Algorithm 5 _move_items_to_bin(list_of_items_indices, bin_index)

for chaque article à indice dans list_of_items_indices **do**
  Ranger l'article dans la boîte ayant l'indice bin_index
end for

1.4.1.2 Méthode principale :

La méthode : Pack_items()

Rôle : ranger les articles dans un nombre min de boîte , en retournant la solution optimale et son coût (nombre de boîtes utilisées)

Algorithm 6 _move_items_to_bin(list_of_items_indices, bin_index)

while il reste encore des articles à ranger dans la liste L **do**
Construire la table de vérité m : m=get_truth_table(capacité, items)
bin_index = Ouvrir une nouvelle boîte
Ajouter la nouvelle boîte à la liste des boîtes
picked_items = _pick_items(m) // choix des avrticles à ranger dans la boîte i
_move_items_to_bin(picked_items, bin_index) // ranger les articles dans la boîte i
Mettre à jour la liste L, en retirant les articles rangés
end while

Exemple :

Capacité=5 ;Articles=1,5,2

première itération : Construction de la table de vérité :

	C=0	C=1	C=2	C=3	C=4	C=5
w0=1	true	true	false	false	false	false
w1=5	true	true	false	false	false	true
w2=2	true	true	true	true	false	true

— Ouvrir une nouvelle boîte B_0 avec une capacité = 5
— Choisir les articles à mettre dedans : il choisit l'article W_1=5
— Mettre l'article W_1 choisi dans la boîte B_0
— Enlever l'article rangé de la liste des articles à ranger.

deuxième itération : Construction de la table de vérité :

	C=0	C=1	C=2	C=3	C=4	C=5
w0 =1	true	true	false	false	false	false
w2 = 2	true	true	true	true	false	false

— Ouvrir une nouvelle boîte B_1 avec une capacité = 5

— Choisir les articles à mettre dedans : il choisit l'article W_0=1 et l'article W_2=2

— Mettre les articles W_0 et W_2 choisis dans la boîte B_1

— Enlever les articles rangés de la liste des articles à ranger.

troisième itération :

— Liste vide

ARRÊT DE L'ALGORITHME

La solution optimale :

— B_0 contiendra l'article W_1 avec un taux d'occupation=$5/5 = 100\%$

— B_1 contiendra les articles W_0 et W_2 avec un taux d'occupation=$(1 + 2)/5 = 60\%$

Chapitre 2

Test et Résultats :

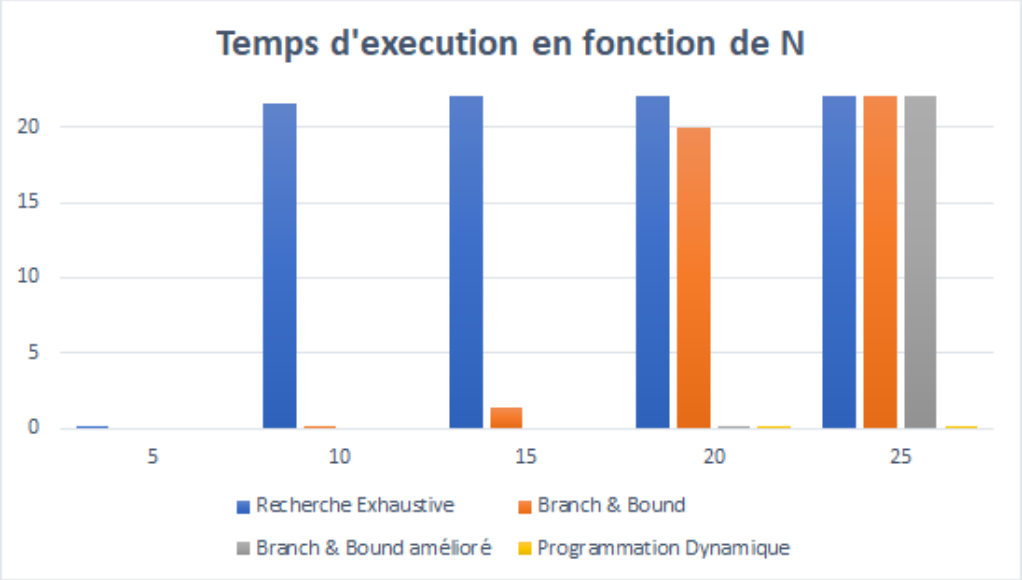
Dans cette partie, nous allons comparer les performances de nos algorithmes implémentés, pour cela, en premier lieu on va utiliser notre propre générateur d'instances pour comparer les 4 algorithms : recherche exhaustive , Branch & Bound , Branch & Bound amélioré et la programmation dynamique. ensuite on utilisera les instances du benchmark Scholl.

Remarque : Les algorithmes ont été développés en utilisant le langage de programmation Python, et exécutés sur un **HP probook [Intel Core i7-6500U CPU @2.50GHz, 8Go RAM]** en utilisant l'IDE Intellij pycharm Le générateur d'instances utilise la fonction random() de la bibliothèque random de Python, cette fonction utilise le Mersenne Twister qui est un générateur de nombres pseudo-aléatoires, réputé pour sa qualité.

2.1 Instances générées :

On génère plusieurs instances du problèmes avec la valeur $C = 100$ pour la capacité de la boîte, et un nombre d'articles croissant : $N \in \{5, 10, 15, 20, 25\}$. Les volumes des articles sont générés aléatoirement dans l'interval $]0, 100]$. le tableau ci dessous résume les résultats en termes de temps d'exécution en secondes de chacun des 4 algorithmes sur les instances générées.

N (nombre d'articles)	Recherche Exhaustive	Branch & Bound	Branch & Bound amélioré	Programmation Dynamique
5	0.0156	0.0	0.0	0.0
10	21.5747	1.3121	0.0	0.0
15	-	0.0156	0.0	0.0
20	-	-	0.0624	0.0156
25	-	-	-	0.03124



2.1.1 Analyse des résultats :

1. On remarque d'un côté qu'en augmentant la taille du problème, le temps d'exécution augmente très rapidement.
2. D'un autre côté, les performances de la DP sont meilleures que celle du Branch and Bound amélioré , suivie du Branch & Bound classique, et enfin vient la rechercher exhaustive qui prend un temps énorme pour résoudre des instances de taille petite.
3. Le Branch & Bound et la recherche exhaustive arrivent rapidement à leur limite ,qui est de $N = 15$ et $N = 20$ respectivement, suivi du Branch and Bound pour $N = 25$ dans ces instances générées. Ceci signifie que ces méthodes ne sont pas efficaces pour de grandes instances.

2.1.2 Interprétation des résultats :

On justifie les résultats obtenus et la grande différence entre les temps d'exécution des 4 méthodes comme suit :

1. La recherche exhaustive, donne des temps d'exécution les plus long, car cette dernière ne possède aucun mécanisme de réduction du problème, donc elle vas parcourir toute les permutations possibles des articles.
2. Le Branch & Bound, offre une petite amélioration par rapport à la recherche exhaustive, grâce à la borne inférieure L1 utilisée pour réduire quelques branches qu'on est sure qu'elles ne contiennent pas la solution optimale, mais cet algorithme arrive à sa limite rapidement, car la borne L1 n'est efficace que lorsque les volume des articles sont petits par rapport à la capacité de la boîte, sinon, on retombe sur une recherche exhaustive.
3. Le Branch & Bound amélioré, augmente les performances du Branch & Bound classique, à cause de la borne L2 qui couvre des cas de réduction plus large que la borne L1, de plus , l'utilisation des heuristiques permet d'accélérer le temps de trouver un noeud exacte.
4. Finalement, la programmation dynamique a pu résoudre le plus grand nombre d'instances, en un temps beaucoup plus petit que les autres méthodes ; on peut justifier ce bon comportement par la nature de notre problème, qui passe dans sa résolution par les même sous problèmes plusieurs fois (dans un parcours d'arbre, on repasse par le sous problème (noeud) où k objets ont été rangé, [n-k] fois pour choisir l'objet k+1 à ranger dans l'étape suivante).Donc, on a bien exploiter le point fort de la méthode qui est la table de programmation dynamique pour réutiliser les résultats des sous problèmes sans les recalculer.

2.2 Scholl Benchmark :

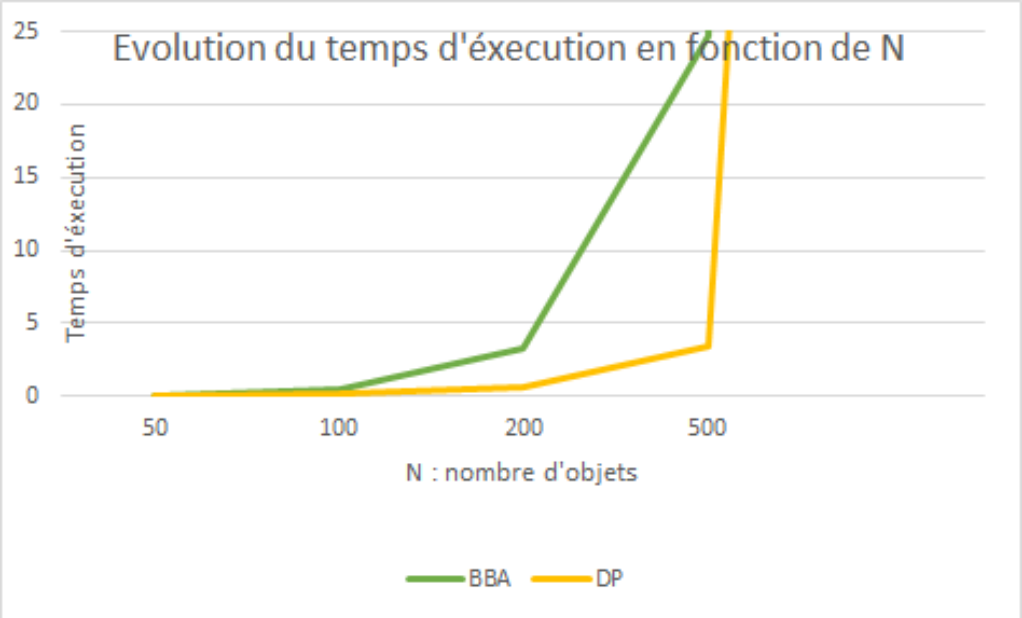
le Scholl benchmark est composé de 3 différentes classes, les volume des articles sont uniformément distribués entre 50 et 500. la capacité C de la boîte est entre 100 et 150 dans la première classe (Scholl1), égale à 1000 dans la classe 2 (Scholl2) et égale à 100 000 dans la 3ème classe (Scholl3). les deux algorithmes Recherche exhaustive et Branch & Bound sont incapable de résoudre les instances de ce benchmark à cause de leurs tailles et difficulté relativement élevés. Donc, dans cette partie nous allons faire une comparaison entre les 2 algorithmes Branch & Bound amélioré [BBA] et la programmation dynamique [DP].

Pour chaque classe, on a 4 valeurs de N (50,100,200,500) et pour chaque couple (N,C) on prends 5 instances afin de calculer le temps d'exécution moyen, ceci est dû à la génération aléatoire des volume des articles, ce qui peut rendre quelques instances plus difficiles que d'autres, même s'ils ont la même valeur du couple (N,C) . les résultats en temps d'exécution sont présentés dans le tableau suivant :

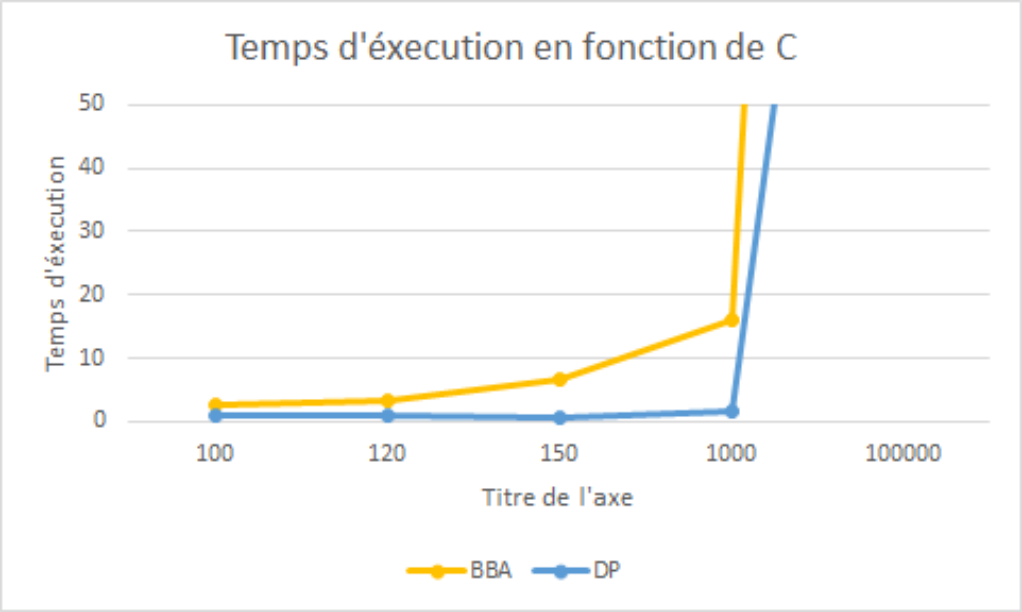
Classe	N	Temps d'exécution (secondes)							
		N1=50		N2=100		N3=200		N4=500	
		BBA	DP	BBA	DP	BBA	DP	BBA	DP
1	C1=100	0.04010	0.02318	0.16994	0.14423	1.83338	0.72787	8.87928	3.31377
	C2=120	0.05078	0.02943	0.37584	0.10916	2.01203	0.48869	10.40025	3.369926
	C3=150	0.06517	0.02437	0.57027	0.09249	2.91527	0.44331	22.92290	1.72647
2	C=1000	0.10653	0.06642	0.83099	0.26568	6.25429	0.92168	56.91714	5.53843
3	C=100.000	-	-	-	-	-	-	-	200

2.2.1 Analyse des résultats :

1. La programmation dynamique (DP) trouve la solution optimale en un temps meilleur que le branch & Bound amélioré
2. En augmentant N le nombre d'articles, le temps d'exécution augmente d'une façon exponentielle



3. En fixant le nombre d'articles, l'augmentation de la capacité C produit une augmentation dans les temps d'exécution



4. Pour la 3eme classe, qui contient les instances les plus difficiles, seul l'algorithme DP arrive à terminer son exécution(en 200 secondes), ce qui n'est pas très intéressant comme temps d'exécution.

2.2.2 Conclusion

Malgré les améliorations apportées aux algorithmes exactes (utilisation des heuristiques pour l'initialisation de la solution optimale, utilisation d'un évaluation plus performante ..), ces derniers suivent toujours la courbe exponentielle en terme de temps d'exécution en augmentant la taille du problème. En d'autres termes, ce type d'algorithmes arrivent rapidement à leur limite, sans même pas pouvoir résoudre des instances de taille moyenne. De nos jours, les données étants d'une très grande taille (qui dépasse les milliers), l'utilisation des méthodes exactes, quelques soit leurs performances, est impossible même avec les ordinateurs les plus rapides du monde. C'est pour cela que les chercheurs se sont dirigés vers des méthodes approchées qui fournissent une solution proche de l'optimal mais en un temps polynomial. Ce qui fait l'article des prochaines parties de notre projet.

Quatrième partie

Worst Case Analysis :

Les méthodes approchées cherchent à trouver une solution la plus proche possible de la solution optimale, pour mesurer la qualité de cette solution obtenue, nous utiliserons l'analyse du pire des cas *"worst Case analysis"*.

Dans cette analyse, les performances d'un algorithme sont mesurées par l'écart de la solution du pire cas (l'instance où l'algorithme donne la pire solution) à la solution optimale. L'une des métriques les plus utilisées dans l'analyse du pire des cas est le *Worst Case Ratio*.

Worst Case Ratio

Ce rapport mesure la déviation maximal de la solution obtenue par l'heuristique, par rapport à la solution optimale.

Soit :

- **L** : une instance du bin packing.
- **A(L)** : le nombre de boîtes utilisées en appliquant l'heuristique A sur L.
- **OPT(L)** : le nombre de boîtes optimal.

Le rapport est donné par la formule suivante :

$$Ra \equiv \{r \geq 1 : \frac{A(L)}{OPT(L)} \geq r \textit{ pour toute instance } L\}$$

Pour calculer le ratio, on calcul le rapport $A(L)/OPT(L)$ pour chaque instance, ensuite on prend le plus petit des majorants de ces rapports. Il est claire que la valeur idéale est un "1", ce qui correspond au cas où la solution obtenue est égale à la solution optimale pour toutes les instances, et dès que la solution obtenue s'éloigne de la solution optimale le rapport va augmenter.

Cinquième partie

Méthodes Heuristiques :

Chapitre 3

Conception :

Les heuristiques sont des méthodes spécifiques qui exploitent au mieux la structure du problème dans le but de trouver une solution raisonnable (non nécessairement optimale) en un temps réduit. L'utilisation de ce type d'algorithmes s'impose car les méthodes de résolution exactes sont de complexité exponentielle, et échouent à trouver la solution pour des instances de tailles moyennes voir petites, comme on la constater lors du chapitre précédant. L'usage des heuristiques est donc pertinent pour surmonter ces limites.

Dans ce chapitre, nous allons présenter la conception détaillée des heuristiques sur lesquelles notre choix d'implémentation s'est porté et qui sont :

1. Next Fit (NF)
2. Next Fit Decreasing (NFD)
3. First Fit (FF)
4. First Fit Descreasing (FFD)
5. Best Fit (BF)
6. Best Fit Decreasing (BFD)

Dans le but d'explorer ces méthodes, comparer leurs performances, montrer leurs avantages et découvrir leurs limites, nous effectuerons des tests empiriques et comparatifs sur les mêmes benchmarks utilisés pour les tests des méthodes exactes (Benchmark Scholl).

3.1 Next Fit (NF)

3.1.1 principe

Si l'article tient dans la même boite que l'article précédent, il est placé avec ce dernier. Sinon, on ouvre une nouvelle boite et le mettre là-dedans.

— NF est un algorithme simple d'une complexité de O(n).

3.1.2 Pseudo-Code

Algorithm 7 Next Fit

```
for Tous les articles i = 1, 2,. . . , n do
  if l'article i s'inscrit dans la boîte actuelle then
    Ranger l'article i dans la boîte actuelle
  else
    Créer une nouvelle boîte, en faire la boîte actuelle et ranger l'article i
    dedant.
  end if
end for
```

3.2 Next Fit Decreasing (NFD)

3.2.1 principe

Le NFD est une amélioration de l'algorithme Next-Fit. Cet algorithme ordonne es articles par ordre décroissant des poids,puis applique l'algorithme NF.

3.2.2 Pseudo-Code

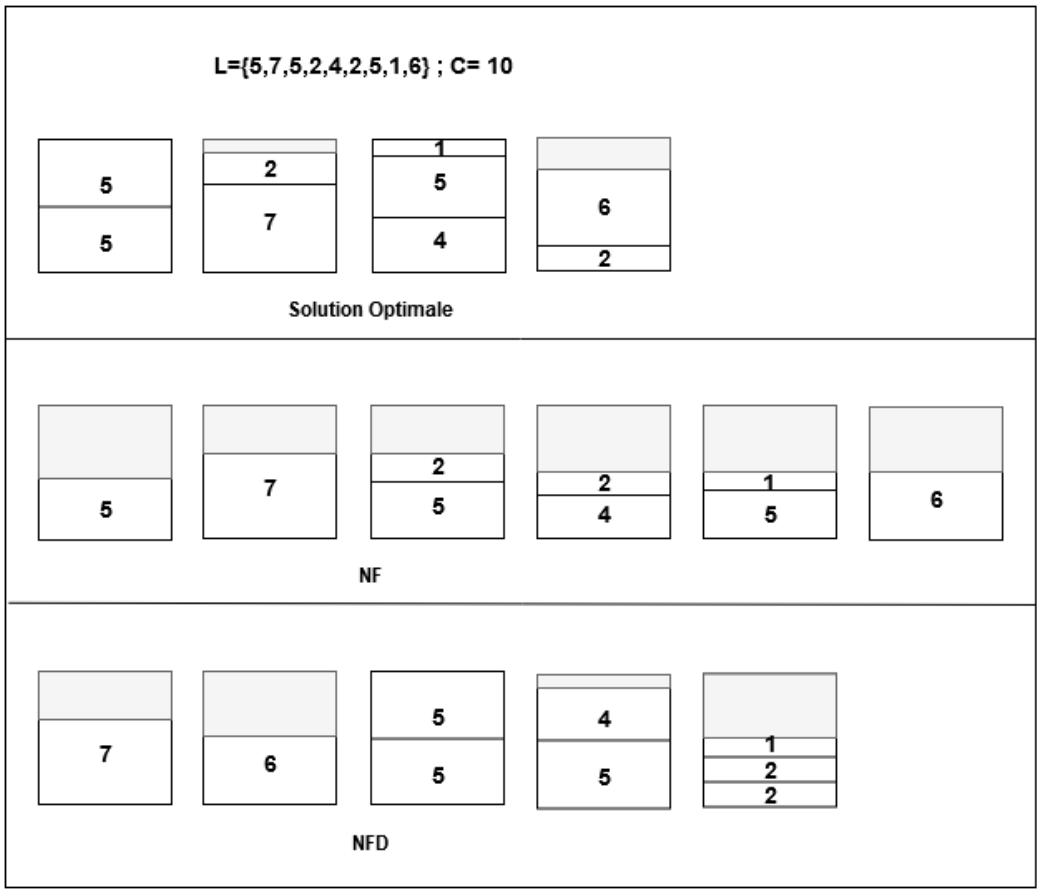


FIGURE 3.1 – Exemple NF et NFD

3.3 First Fit (FF)

3.3.1 principe

Ranger chaque article courant dans la première boîte, entre celles déjà ouvertes, qui peut le contenir sinon ouvrir une nouvelle boîte et on le range dedans.

— L'algorithme First Fit implémenté a une complexité de $O(n^2)$.

3.3.2 Pseudo-Code

Algorithm 9 First Fit

```
for Tous les articles i = 1, 2,. . . , n do
  for Tous les boîtes j = 1, 2,. . . m do
    if l'article i s'inscrit dans la boîte j then
      Ranger l'article i dans la boîte j
      Quitter la boucle ( passer à l'article suivant)
    end if
  end for
  if l'article i ne rentre dans aucune boîte disponible then
    Créer une nouvelle boîte et ranger l'article i dedans
  end if
end for
```

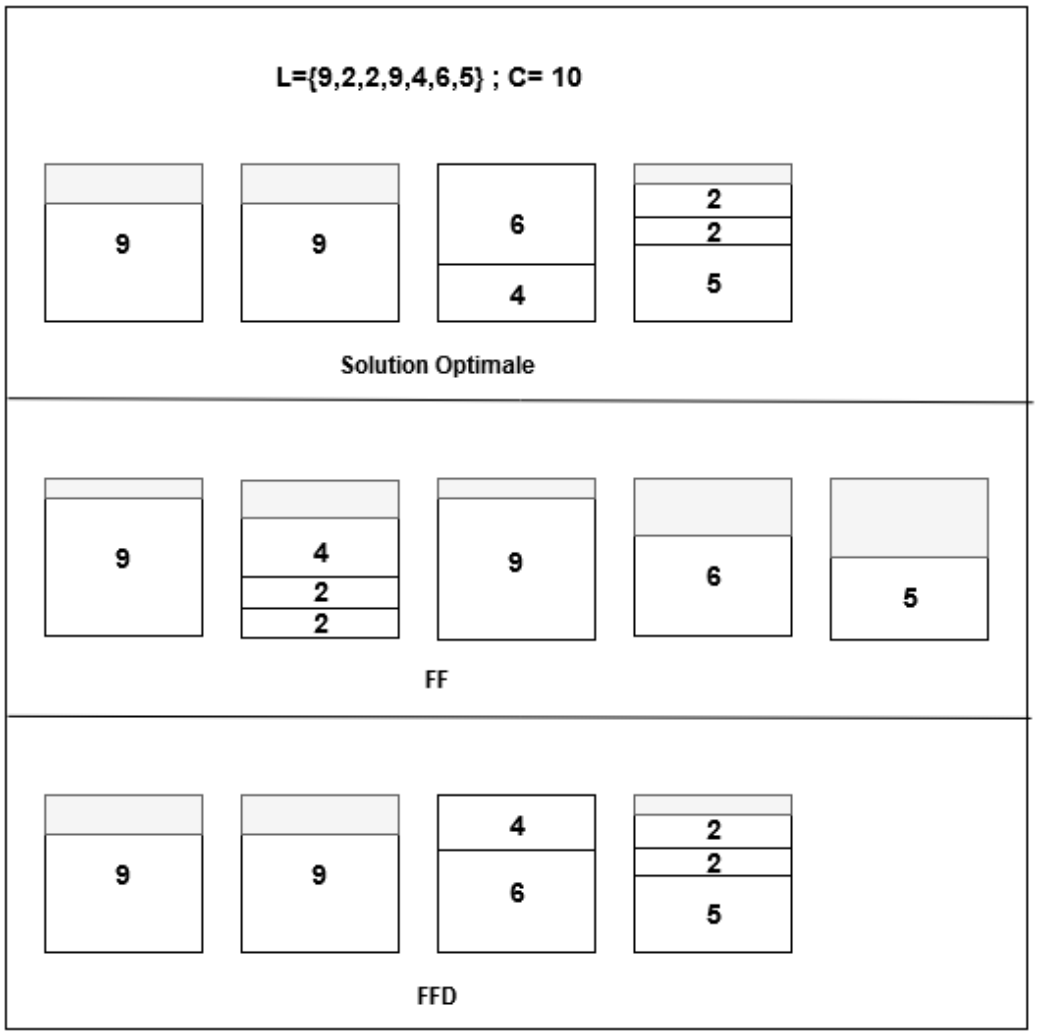
3.4 First Fit Descreasing (FFD)

3.4.1 principe

Le FFD est une amélioration de l'algorithme First-Fit. Cet algorithme ordonne les poids dans le sens décroissant puis lui applique l'algorithme FF.

— L'algorithme First Fit peut être implémenté en O(nlog n) en utilisant les arbres de recherche binaires

3.4.2 Pseudo-Code



figure[Exemple FF et FFD]Exemple FF et FFD

3.5 Best Fit (BF)

3.5.1 principe

Ranger chaque article

courant dans la boîte la mieux remplie, entre celles déjà ouvertes, qui peut le contenir sinon ouvrir une nouvelle boîte et on le range dedans.

L'algorithme Best Fit implémenté a une complexité de $O(n^2)$.

3.5.2 Pseudo-Code

Algorithm 11 Best Fit

— **for** Tous les articles i = 1, 2,. . . , n **do**
  for Tous les boîtes j = 1, 2,. . . m **do**
    if l'article i s'inscrit dans la boîte j **then**
      Calculer la capacité restante dans la boîte j une fois l'article
    end if
  end for
  Ranger l'article i dans la boîte j, où j est la boîte ayant la capacité restante minimale après avoir ajouté l'article(c'est-à-dire que "l'article convient le mieux").
  if une telle boîte n'existe pas (l'article ne peut être rangé dans aucune boîte) **then**
    Créer une nouvelle boîte et ranger l'article i dedans
  end if
end for

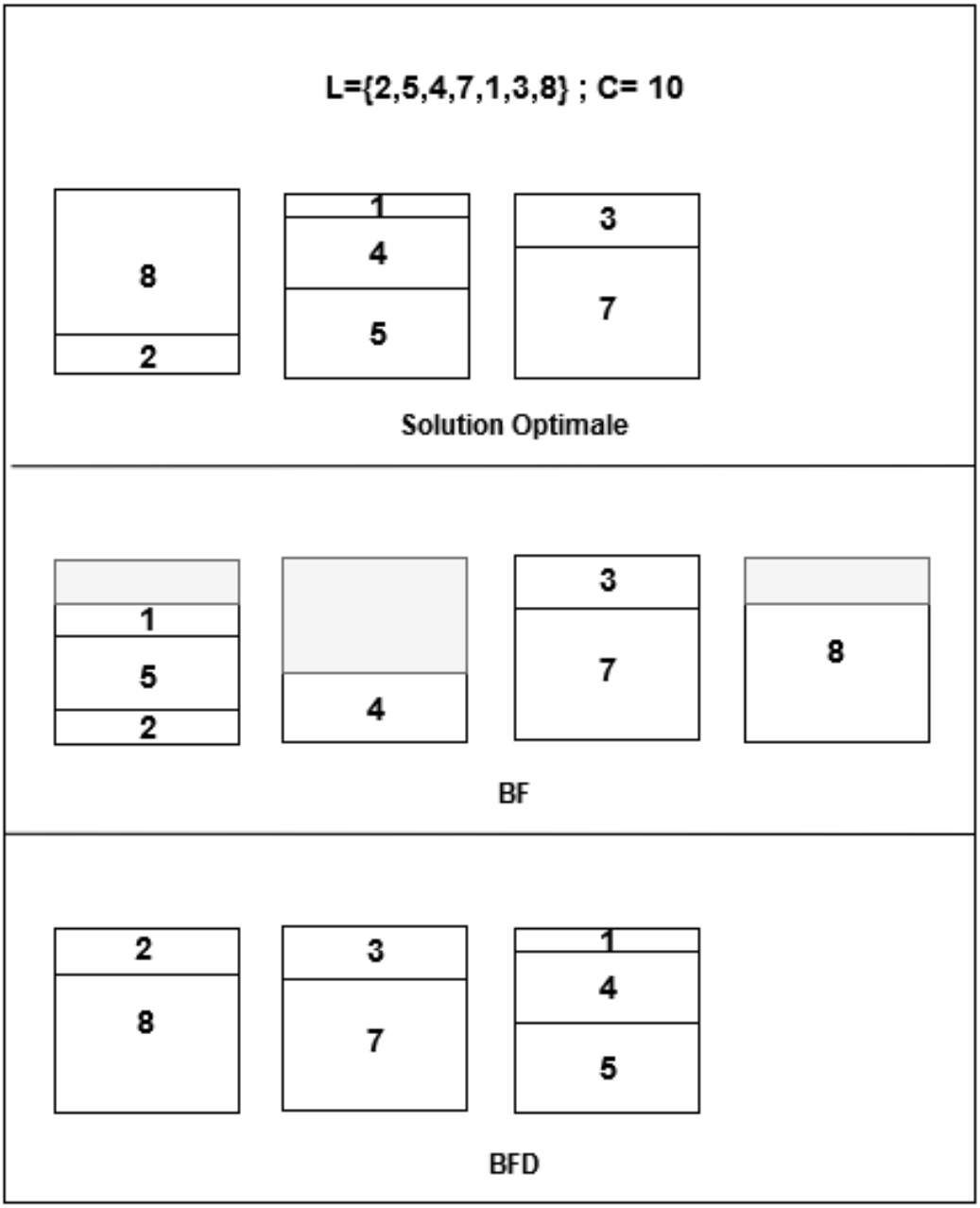
3.6 Best Fit Decreasing (BFD)

3.6.1 principe

Le BFD est une amélioration

de l'algorithme Best-Fit. Cet algorithme ordonne les poids dans le sens décroissant puis applique l'algorithme BF.

3.6.2 Pseudo-Code



figure[Exemple BF et BFD]Exemple BF et BFD