B I N - P A C K I N G
2 D

DIALLO Boubacar
SOUIKI Nouaïm

Table des matières

Introduction

1 Autour du problème

Ce projet consiste en une implémentation d'une heuristique, pour le problème d'optimisation connu sous le nom de « bin-packing », dans sa version en deux dimensions.

Ce problème peut s'énoncer de cette manière :
Etant donnés un conteneur rectangulaire, ainsi qu'un nombre arbitraire d'objets de cette même forme, calculer un remplissage du conteneur, par un sous-ensemble de ces objets, de manière à optimiser la surface occupée dans le conteneur.

Afin de faciliter la résolution de ce problème, on se contente d'un placement orthogonal des objets (c'est-à-dire que les objets sont placés parallèlement aux bords du conteneur).
De plus, nous nous interésserons facultativement à la version non orientée du problème, c'est-à-dire à la possibilité de tourner les objets de 90 degrés lors de leur placement.

Difficulté :
Ce problème a eté démontré *NP-difficile* : il n'existe donc pas d'algorithme polynomial pour le résoudre.
Les heuristiques utilisées ont alors des complexités exponentielles.

2 Présentation des programmes

Les programmes implémentés afin de résoudre ce problème sont au nombre de trois : un *solveur,* qui calcule une solution du problème pour des instances produites par un *génerateur* indépendant, et enfin un *dessinateur* qui offre une illustration de la solution proposée.
Nous faison remarquer ici que les figures utilisées tout au long de ce projet sont des captures d'écrans de remplissages effectués avec le fichier d'instance *« instance_de_reference »*, qui se trouve avec les sources dans le dossier du projet.
Cette instance est formée d'un conteneur de dimensions *200x180* , pour un ensemble de *cinq mille objets* de dimensions avoisinant celles du conteneur.

2.1 Générateur d'instances

Ce programme génère de façon pseudo-aléatoire les instances pour le remplissage, selon la ligne de commande saisie. Le module *« generateur »* implémenté pour ce programme est indépendant dans le projet.
La génération d'une intance se fait en fonction du nombre d'objets, requis par la ligne de commande, ainsi que des intervalles de leurs dimensions. Les objet sont ainsi générées dans la contrainte de ces intervalles. Une semence facultative peut être saisie manuellement, à défaut de quoi la suite aléatoire à génerer sera simplement initialisée par la date courante.

Avant de procéder à la génération, le programme vérifie les conditions de validités des dimensions minimales et maximales requises par l'utilisateur :

- ✗ Une dimension ne peut être nulle (auquel cas l'objet généré correspondra au mieux à une droite, au pire à un point, ce qui ne présente evidemment aucun d'intérêt pour notre problème).
- ✗ Une dimension minimale ne peut dépasser sa dimension maximale correspondante.

2.2 Solveur

Il s'agit du programme qui implémente l'heuristique proposant une solution pour le problème posé.
En premier lieu, ce programme réalise une lecture de l'instance du problème, puis calcule un remplissage du conteneur, lequel remplissage diffère selon les options de la ligne de commande. Pour plus d'informations sur ces options, saisir la commande : ./remplir_conteneur -help

Les modules ayant implémentés ce programme sont les suivants :

« lecture_probleme », « recherche_remplissage », « remplir_bande » , « utiles ».

Le programme principal se charge de la prise en compte des options de la ligne de commande, de l'allocation de toutes les structures sollicitées durant l'exécution -ce qui réduira les multiples allocations et libérations dans la mémoire durant la recherche ; il se charge également du prétraitement de l'instance, de l'appel de la procédure de recherche, et enfin de l'affichage des solutions selon la syntaxe requise -à laquelle nous ajoutons simplement deux dernières lignes représentant la surface du remplissage obtenu, ainsi que son pourcentage dans le conteneur.

2.3 Dessinateur de solution

Il s'agit là du programme qui dessine la solution trouvée dans une interface graphique.
La librairie de dessin utilisée est *EZ-Draw*. Les options reconnues par la ligne de commande sont, entre autres, l'affichage des noms des objets dans le conteneur même (option `--text-objs`) ou encore le coefficient d'agrandissement du dessin.
Les modules implémentant ce programme sont « dessiner_solution » ainsi que le module « ez-draw » du professeur Thiel qui contient les outils de dessin utilisés.

1 Lecture des instances

La lecture d'une instance se fait par tube ou par redirection depuis un flot contenant un ensemble bien formé d'objets. Le programme de lecture vérifie la validité de synatxe de l'instance, puis la sauvegarde dans la mémoire. Pour chaque objet de l'instance, une structure sauvegarde son identité, qui est construite selon la syntaxe requise (constituée ici du nom de l'objet et de ses dimensions), de telle manière à ce qu'un tableau de telles structures sauvegarde la totalité de l'instance.

2 Résolution

Avant d'attaquer l'optimisation proprement dite du problème, définissons dès abord notre méthode de recherche.

2.1 Principe

Nous utilisons ici une méthode *déterministe* pour résoudre le problème, c'est-à-dire qu'en théorie la recherche s'effectue sur toutes les solutions possibles, pour ne garder à la fin que la plus optimale.
Cependant, ce problème étant NP-difficile, et de plus l'énumération de toutes les solutions possibles étant dans la pratique irraisonnable -et ceci même pour un problème de taille moyenne, nous restreignions la recherche à un nombre limité de ces solutions, selon la largeur d'exploration choisie par l'utilisateur, qui est par defaut à 3.

En outre, le remplissage effectué ici étant un remplissage par bandes, la recherche énumère tous les remplissages possibles sur un nombre de largeurs de bande valides, en les remplissant chacune suivant une stratégie pré-définie (et qui peut être plus ou moins optimale).

Cette recherche énumérative construit une arborescence sur les différentes largeurs de bandes sélectionnées. Le nombre de largeurs à sélectionner est égale dans le meilleur des cas, à la largeur d'exploration de la procédure de recherche. En d'autres termes, si nous appelons k la largeur d'exploration de la procédure, l'algorithme de recherche aura idéalement k branches à explorer à chaque profondeur.

2.2 Algorithme de recherche

2.2.1 Corps de l'algorithme

Nous ré-expliquons ici, de la façon dont nous l'avons compris, l'algorithme principal du programme, proposé dans le sujet du projet.

Pour construire l'arborescence des choix, cet algorithme exécutera :

- Une boucle itérant sur un certain nombre de largeurs de bandes valides.
- A chaque tour de cette boucle, l'algorithme remplit la bande courante via la stratégie de remplissage choisie.
- L'algorithme se rappelle ensuite récursivement sur le reste du conteneur, en sélectionnant des nouvelles largeurs de bandes valides, etc ...
- Avant de sortir d'une itération (c'est-à-dire lorsqu'un remplissage possible du conteneur a eté calculé), les objets

ayant remplit la dernière bande sont libérés de cette bande.
- La procédure s'arrête après avoir itéré sur toutes les largeurs de bandes selelctionnées.

Cette arborescence effectue donc une recherche exhaustive pour la largeur d'énumération choisie. A chaque fois que l'algorithme atteint une feuille de l'arbre de récursion (c'est-à-dire qu'un remplissage possible du conteneur a eté calculé), la récursion le fait remonter à un ancien nœud de l'arbre, pour lequel il essaie d'autres largeurs possibles à remplir (sans avoir oublié au préalable de faire une mise à jour du meilleur remplissage).

La libération des objets de la dernière bande remplie assure un parcours complet de l'arbre des choix permis par la largeur d'exploration, puisqu'en remontant, l'algorithme aura possibilité de remplir d'autres bandes grâce à ces objets libérés.

2.2.2 Complexité

Un mot encore sur la complexité de cette procédure.
Si nous prenons comme convention de noms :

L := *largeur du conteneur,*

lmin := *la plus petite largeur de bande à remplir,*

bmax := *L / lmin,*

Alors, étant donné qu'à chaque profondeur de l'arbre, l'algorithme explore ***k*** branches dans le pire des cas, et que la hauteur de l'arbre de recherche est majorée par le nombre de bandes à remplir, qui est égal au plus à ***bmax ,*** nous pouvons alors en déduire que la complexité de notre algorithme est de

$$k^{bmax}$$

Il s'agit donc bien d'une complexité *exponentielle.*

3 Implémentation

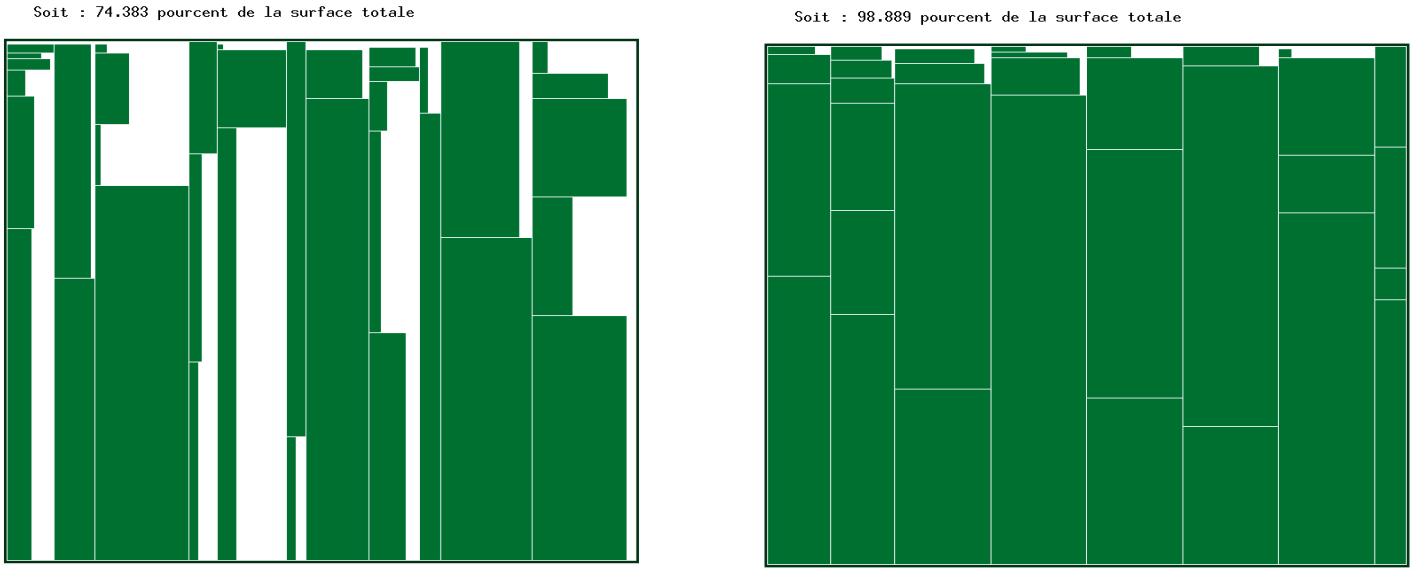
3.1 Prétraitement des instances

3.1.1 Cas trivial

Dans le cas où l'instance se réduit à un seul objet ou celui où un objet de l'instance a les mêmes dimensions du conteneur (à l'ordre près), il est alors inutile de rentrer dans la procédure de recherche, et ceci est alors traité à part dans le programme principal.

3.1.2 Tri préliminaire

Une fonctionnalité que nous avons choisi pour notre heuristique, du fait de son efficacité, est la suivante : après avoir lu, sauvegardé les objets de l'instance et juste avant l'appel de la procédure de recherche, le programme principal va trier l'ensemble des objets du prblèmes en fonction des largeurs dans le but d'avoir les objets les plus larges en fin de la liste.
L'algorithme de tri choisi pour cette effet est le tri par tas (essentiellement en vertu de sa complexité logarithmique).
A titre d'exemple : sur notre instance de réference, les placements (gloutons) obtenus *sans* et *avec* ce tri donnent des taux de remplissage respectifs de *74* et *99* % de la surface totale du conteneur (voir la figure 1), soit 25 % de différence en faveur du remplissage avec l' instance triée.



(a) Instance non triée (b) Instance triée.

Figure 1 - Illustration d'un remplissange glouton effectué sans et avec l'option `-tri`.

N.B : Dans le reste de la présentation, et sauf précision du contraire, l'option `-tri` sera utilisée implicitement en même temps que les fonctionnalités exposée.

3.2 De la disponibilité des objets

Afin de permettre un parcours correct de l'arborescence lors de la recherche, l'algorithme doit pouvoir essayer toutes largeurs de bandes sélectionnées à chaque nœud parcouru de l'arbre.
Il est donc indispensable qu'au moment de remonter dans la récursion, dans le but essayer d'autres largeurs de bandes, l'algorithme doit avoir à sa disposition des objets *libres* afin de tester les placement pour ces autres largeurs de bandes.

Ainsi, lors de la recherche, et juste avant de remonter dans la récursion, la procédure doit *« libérer »* les objets placés dans la dernière bande. Sans cela, le parcours de l'arbre risque partiel et certaines solutions possibles, permises par la largeur d'exploration, seront peut-être ignorées.

3.2.1 Caractérisation des objets libres

Signifier qu'un objet est *« libre »* - ou *« disponible »* se fait grâce à un tableau local d'entiers : le tableau `O[]`. Ce tableau, indexé sur les indices des objets, caractérise leur disponibilité de la façon suivante :
Soit *i* l'indice d'un objet dans l'instance du problème ;alors :

> *i est libre* **si et seulement si** `O[`i`]` $= i$
> **Sinon** : `O[`i`]` $=$ *-1*

3.2.2 Libération des objets

En premier lieu nous rappelons que pour le besoin de la récursion et afin d'éviter de mutilples allocations dans la mémoire, la sauvegarde des objets *pris* dans une bande se fait à l'aide d'une pile de tableaux d'entiers, nommée `Pile_Objets_Pris[][]` et qui est indexée sur la profondeur de l'arbre des choix.
De cette façon lors de ses *batracks*, l'algorithme de recherche retrouvera aisément les objets disponibles pour un placement. Ensuite, la libération d'objets consiste simplement à les caractériser « libres» dans le tableau `O[]` prévu à cet effet, sans oublier de les vider de la pile.

3.3 Sélection des largeurs de bandes

La sélection des largeurs de bandes valides a lieu à chaque profondeur de l'arbre de recherche.
La façon la plus naturelle de sélectionner des largeurs de bandes valides - c'est-à-dire aptes à être remplies par des objest

existants, consiste à ne considérer que des largeurs correspondants aux dimensions des objets effectivement libres au nœud courant de l'arbre.

Dans la même logique que pour les objets placés dans une bande, la sauvegarde des largeurs de bandes decidées valides se fait grâce à une pile de tableaux d'entiers, qui est indexée, de le même façon, sur la profondeur de l'arbre des choix. En référence au tableau des largeurs valides *« C »* du pseudo-code, nous avons nommé cette pile `Pile_C[][]`.

3.3.1 Sélection des largeurs par défaut

Dans un premier temps, et dans le but de réaliser avant toute chose la récursion correctement, nous avons choisi de ne remplir que des bandes pré-choisies par défaut qui sont de largeurs *dix, vingt* et *trente*, et même si les objets effectivement libres n'y correspondent pas (auquel cas le remplissage résultant sera très médiocre).

La fonction implémentée pour ce besoin dans notre programme est `Selectionner_k_LargeursValidesParDefaut()`, dont le seul mérite aura finalement été de nous permettre de mieux comprendre, dans un premier temps l'exécution de l'algorithme proposé.

A titre d'exemple : en première partie du projet, dans le compte rendu que nous avions écrit, nous affirmions avec fierté que notre programme calculait le placement d'un million d'objets en une vingtaine de secondes : il s'agissait en réalité là d'une erreur dans notre implémentation, en conséquence de laquelle notre programme parcourait mal l'arborescence de la recherche, ce qui expliquait la rapidité anormale de notre algorithme.
La simplicité de cette première procédure de sélection nous a permis, assez facilement de résoudre ce problème.

Il est à noter tout de même, pour la défense de cette procédure qu 'elle ne sélectionne une largeur par défaut dans {10, 20, 30}, que s'il existe au moins un objet valide et disponible pour cette largeur de bande.

3.3.2 Sélection des largeurs valides

La fonction `Selectionner_k_LargeurValides()` utilise l'autre stratégie de sélection précèdemment évoquée, et qui consiste à choisir les largeurs de bandes parmi les largeurs des objets libres du moment. Nous avons implémenté cette méthode, mais dans sa version améliorée proposée par le sujet.

3.3.3 Améliorations

Cette version améliorée consiste à sélectionner les largeurs les plus fréquentes parmi les dimensions des objets : c'est une sélection des largeurs selon leurs occurrences.
De cette façon, à une largeur de bande sélectionnée correspondra nécessairement le maximum d'objets valides.

- **Calcul des occurrences**

Pour le calcul des occurrences, nous avons repris et exploité une idée qui était assez en vogue entre nos différents groupe de binômes : un tableau d'entiers indexé sur les largeurs d'objets indique l 'occurrence de chaque largeur. Ce calcul, qui aurait pu être effectué avec une complexité en n^2 avec un algorithme classique en double parcours, s'en trouve de cette manière fortement simplifié.

Il suffit en effet de parcourir linéairement l'ensemble des largeurs des objets de l'instance, et d'incrémenter à chacune de ces largeurs l'élément de `Occs[]` d'indice correspondant. Supposons par exemple, que lors de notre parcours de l'instance, nous ayons à calculer l'occurrence d'une largeur l : alors il suffit d'incrémenter `Occs[`l`]` à chaque fois que nous rencontrons l dans parmi les largeurs d'objets.

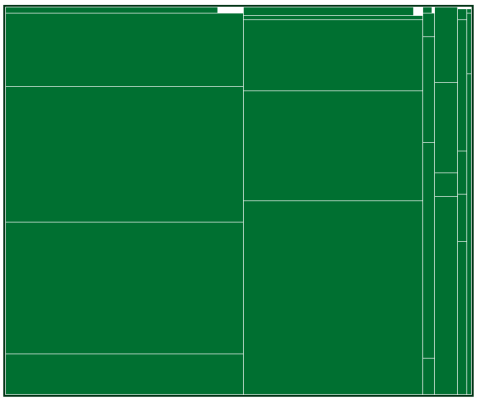
Pour illustrer l'efficacité de cette manière de calculer les occurrences, nous avons exécuté le solveur avec les deux méthodes précèdemment citées, sur une notre instance de référence.
La différence écrasante entre les deux temps d'éxecution produits nous a convaincu de la supériorité de la méthode du calcul linéaire des occurrences: en effet, le temps du remplissage a été aproximativement divisé par 20.

- **Sélection**

Une fois effectué le calcul des occurrences, `Selectionner_k_LargeurValides()` exécute un autre algorithme qui itère sur le nombre de largeurs valides à choisir, en extrayant à chaque itération la largeur la plus fréquente. L' occurrence

de cette largeur est ensuite annulée, afin d'éviter qu'elle ne soit reprise d'une itération à l'autre. La figure 2 illustre un placement des objets de l'instance de référence selon leurs occurrences (voir figure 1.b pour la sélection par défaut).

Figure 2 - Illustration d'un remplissange effectué avec l'option `-occs`.



N.B : Nous avons eu énormément de problèmes à faire fonctionner la sélection selon les occurrences, et cette dernière genère toujours des bugs, notamment une sortie de la profondeur que nous n'arrivons pas à gérer.

Pour visualiser ce problème, exécuter le solveur en lui donnant en entrée l'instance contenue dans le fichier *« segfault_profondeur ».*

3.4 Remplissage d'une bande

Une fois les largeurs de bandes valides sélectionnées, la procédure de recherche extrait la première d'entre elle et la remplit. Afin de ne pas re-remplir la même bande lors d'un batrack ultérieur, la valeur extraite est écrasée, dans la pile, par un *-1*.

Pour remplir la bande, deux stratégies ont eté adoptées : stratégie gloutonne et stratégie sac à dos.

3.4.1 Approche glouton

Cette méthode de placement est comme son nom l'indique, fort basique : nous parcourons linéairement l'ensemble des objets de l'instance, en plaçant les objets qui sont *libres* et *valides* dans la bande en question : il n'est pas question d'optimiser en aucune manière l'espace occupé dans la bande.

Afin de tirer profit du tri préliminaire des objets, le parcours de l'instance se fait à rebours : de cette façon le placement commencera à remplir chaque bande par l'objet le plus large disponible.

3.4.2 Approche sac à dos

Cette méthode consiste à assimiler une bande donnée à un sac à dos que nous voudrions remplir avec des objets, de certains poids et valeurs, sans dépasser la capacité du sac, et en maximisant la valeur totale du contenu.

Il suffit pour cela de considérer les caractéristques du sac en fonction des données de notre problème de remplissage de bande.

Les considérations prises sont les suivantes :

- ✔ La capacité du sac est assimilée à la hauteur initiale de la bande (c'est-à-dire la hauteur du conteneur).
- ✔ Le poids d'un objet est assimilé à la hauteur de cet objet.
- ✔ La valeur d'un objet est assimilée à sa surface.

On met à contribution, dans cette méthode, l'algorithme implémenté en première partie d'année, non sans avoir auparavant sélectionné les candidats éligibles pour la bande en question.

Esuite classiquement, une table de programmation dynamique est remplie par l'algorithme, qui calcule dedans les sous-solutions optimales pour le remplissage de la bande.

Enfin une procédure permet de reconstruire les solutions trouvées, en parcourant la table de programmation dynamique à rebours, et en plaçant dans la pile des objets pris les objets qui l'on effectivement été.

La stratégie sac à dos optimise fortement le remplissage du conteneur, comme le montre la figure 3 -qui illustre un

placement de *100 %*, mais en revanche augmente significativement son temps de calcul.

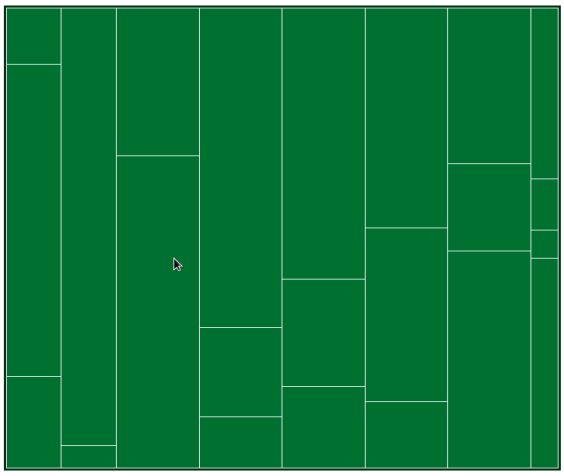


Figure 3 : placement des objets avec l'option `-sac`

Après avoir discuté de ces deux méthodes de placement, il serait à présent utile de présenter deux outils très importants pour l'une comme pour l'autre.

3.4.3 Outils

- **De la validité d'un objet**

Nous abordons ici une notion primordiale concernant le remplissage des bandes : la décision de la validité d'un objet selon sa largeur et des dimensions dans lesquelles il serait souhaitable de le placer.

La fonction chargé de renvoyer cette décision est `EstValide()`, qui, dans sa version basique, se contente de vérifier que les dimensions de l'objet sont inférieures aux dimensions dans lesquelles la procédure appelante veut le placer.

- **Coordonnées d'un objet solution**

Pour chaque objet, une structure munie des champs entiers abscisse et ordonnée permet de sauvegarder ses coordonnées une fois qu'il est placé; ainsi un tableau de telles structures contient les coordonnées de tous les objets qui se trouvent dans le conteneur.

Une fois l'objet placé dans une bande, et quelque soit la méthode de placement utilisée , il peut être alors utile de sauvegarder directement ses coordonnées dans le conteneur : ces coordonées sont calculées dans un repère orthonormal théorique dont l'origine est le coin inférieur gauche du conteneur, et de vecteurs unitaires normés d'un pixel. Afin de simplifier les calculs dans le repère choisi, nous prenons comme convention que les coordonnées d'un objet correspondent aux coordonnées de son coin inférieur gauche.

Le calcul se fait diffèremment selon que la bande soit remplie depuis le haut ou qu'elle le soit depuis le bas (si imbrication de bandes il y a).

3.4.4 Améliorations

- **Concernant la décision sur la validité d'un objet**

Deux amélioration ont été apportées à la fonction `EstValide()` :

- ***Favoriser la hauteur minimale dans la bande***

La fonction commence simplement par tourner l'objet de 90 degrés s'il s'avère que sa hauteur est supérieure à sa largeur, et aussi qu'une fois tourné, il restera (ou deviendra) valide.
Cette manipulation a pour but d'optimiser la hauteur restante de la bande dans laquelle l'objet est placé.

- ***Réaliser un pivot de l'objet***

La fonction tourne l'objet de 90 degrés, si cela lui assure sa validité pour une bande donnée.
Il s'agit en fait simplement de permuter les champs largeur et hauteur dans la structure représentant l'objet que l'on désire pivoter.

Grâce à ces deux améliorations, nous obtenons des remplissages plus optimaux, du fait que nous permettons à une bande d'acceuillir plus d'objets grâce à l'option `-hmin`, et que des objets ne pouvant à priori pas être placés, le peuvent désormais grâce à l'option `-pivot`.

■ Recoupement de la largeur de bande

L'algorithme de référence proposait de rappeler la recherche sur le reste du conteneur diminué de la largeur de bande dernièrement remplie.
L'inconvenient d'une telle méthode est qu'il arrive qu'une bande ne soit remplie que partiellement par les objets qui la remplissent : il y aura donc un gaspillage de surface entre la largeur de bande et l'objet le plus large qu'elle contient.
Pour éviter ce gaspillage, nous avons fait en sorte que la procédure de remplissage d'une bande renvoie, comme résultat, la dimension de l'objet le plus large placé.
Ainsi le rappel de l'algorithme de recherche se fait, de cette manière, sur le reste du conteneur diminué de cette valeur.
Nous pouvons vérifier sur la figure 4 que, sans cette méthode, le gaspillage d'espace entre bandes est augmenté (comparer avec la figure 1.a qui illustre un placement avec recoupement).

N.B : `-tri` n'a pas été utilisée ici, par souci de cohérence avec la figure 1.a.

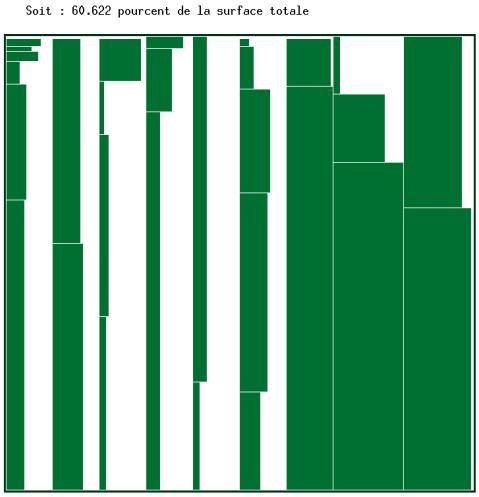


Figure 4 : sans le recoupement, le placement est moins optimal, les bandes étant très espacées.

N.B : Nous avons cependant remarqué que sur certaines instances cette amélioration devient caduque grâce au tri des largeurs, et qu'ainsi sur une certaine instance triée, avec ou sans recoupement des bandes, la qualité du placement reste inchangée.

■ Imbrication des bandes deux par deux

Nous avons sans succès essayé d'implémenter l'imbrication des bandes deux par deux au fur et à mesure du remplissage.
Dans la théorie, l'algorithme que nous avons implémenté nous semblait fonctionnel, mais l'éxécution produit ce qui nous semble être une itération infinie.

3.5 Mise à jour du meilleur remplissage

La mise à jour d'un meilleur remplissage du conteneur se fait à chaque feuille de l'arbre de récursion, lorsqu'il apparaît que le remplissage calculé tout au long de la branche courante est meilleur que le remplissage optimum déjà trouvé.
Cela se fait simplement en comparant la surface du placement courant avec celle de l'optimum trouvé : si la nouvelle surface courante est meilleure, on écrase l'ancien optimum par ce nouveau placement.

Le calcul des surfaces se fait à chaque fois qu'une solution possible est trouvée, et non pas au fur et à mesure du remplissage des bandes : on parcourt simplement les objets du placement en ajoutant leurs surfaces.

Un objet faisant partie d'un meilleur remplissage se voit sauvegarder son identité dans une structure dédiée à cet effet, qui contient en plus des informations initiales concernant l'objet, ses coordonnées dans le conteneur. Un tableau de telles structures contient les identités de tous les objets solutions.

3.6 Autour de quelques propriétés de coupures

Etant donné que la recherche parcourt - selon sa largeur d'exploration– la totalité de l'arborescence des choix, et cela même si le meilleur placement possible pour objets a déjà été trouvé, inopinément, dans un noeud précédent il devenait intéressant de savoir si l'on pouvait « couper » l'algorithme à un moment où il devenait évident qu'aucun autre meilleur remplissage ne pouvait en résulter.

3.6.1 Coupe sur le remplissage partiel

Nous implémentons pour cette coupe, la propriété de coupure énoncée dans le sujet du projet.
Cette propriété énonce que, si à un nœud donné le remplissage effectué laisse un espace vacant supérieur à l'espace vacant du meilleur remplissage déjà sauvegardé, il ne sert plus de continuer : on peut à ce moment-là couper la recherche – non sans avoir tout de même libéré les objets ayant remplis la dernière bande, ainsi l'algorithme remonte dans la récursion et s'en va parcourir les autres branches possibles de l'arbre.

En général, le temps de résolution est diminué de près de 50 %, grâce à l'application de cette propriété.

3.6.2 Coupe sur le remplissage total

Une autre coupe évidente peut-être faite lorsque l'on obtient un remplissage total du conteneur - c'est-à-dire lorsque 100 % de la surface du conteneur se trouve remplie, à un nœud donné. Il est trivial que dans un tel cas, il n'est plus utile de continuer la recherche.

Cette coupe est cependant rarement utile pour un placement glouton des objets; elle ne montre en fait son efficacité que lors de certains placements par sac à dos, stratégie qui, pour cerains types d'instances, produit (assez) souvent des remplissages à 100 %, auquel cas l'algorithme de recherche se voit coupé.

4 Changer la largeur d'énumération

Après avoir essayé des placements avec les différentes manières précèdement exposées, nous avons voulu répondre à la question suivante :

L'augmentation de la larguer d'exploration de l'algorithme de recherche influe-t-elle sur la qualité du remplissage ?

Nous savions d'ores et déjà, étant donnée la nature exponentielle de la recherche, qu'une largeur d'énumération plus importante coûterait énormèment plus en terme de temps, et que cette complexité pouvait facilement exploser pour certaines instances, si cette largeur devenait relativment importante.

Nous avons donc lancé notre algorithme sur notre instance de référence, avec deux largeurs d'énumération différentes, assez raisonnables : 2 et 5. Comme nous ne voulions pas complexifier encore plus la tâche avec l'algorithme du sac à dos, nous avons simplement exécuté le programme avec l'option `-occs` et `-tri`, pour un remplissage selon les occurrences des largeurs triées. Ainsi, nous avons laissé tourner le programme, et nous avons attendu ...

Voici les résultats :

Largeur d'exploration	*Temps d'exécution*	*Taux de remplissage*
2	0.4 secondes	99.78 %
5	2 minutes et 15 secondes	99.93 %

Il est donc indéniable que la largeur d'énumération influe sur la qualité du placement des objets dans le conteneur, mais aussi de façon significative sur le temps d'exécution de la recherche.

Ceci était d'ailleurs prévisible : en effet, une largeur d'exploration plus grande permet à l'algorithme de tester plus de

solutions, ce qui signifie qu'une solution plus optimale sera peut-être atteinte, mais en un temps évidemment plus long.

5 Conclusions

Au terme de cette étude, nous pouvons affirmer que l'heuristique que nous avons implémentée pour résoudre le problème posé remplit presque la totalité des objectifs que l'énoncé du projet proposait.

Cependant, nous avons constaté que, pour certaines instances, il était nécessaire d'augmenter l'instance d'un objet bidon pour résoudre une erreur de segmentation avec le sac à dos (qui ne prenait pas en compte le premier élement des instances).

Le seul regret que nous éprouvons est de n'avoir pas su imbriquer les bandes, bien que nous ayons vraiment essayé.