Un mouvement incrémental pour le problème du strip-packing

Bertrand Neveu **Gilles Trombettoni** **Ignacio Araya**

INRIA Sophia-Antipolis, Université de Nice-Sophia, ENPC, Projet COPRIN,
INRIA 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis cedex
Departamento de Informatica, Universitad Tecnica Federico Santa Maria, Valparaiso, Chile
{neveu, trombe, Ignacio.Araya}@sophia.inria.fr

Résumé

Pour traiter les problèmes de placement de rectangles en 2D (bin-packing), de nombreux algorithmes complets et incomplets maintiennent une propriété dite BL (bottom-left) : tout rectangle placé dans un conteneur est calé en bas et à gauche.

Alors qu'il est facile de maintenir la propriété BL lors de l'ajout d'un rectangle, ce maintien est plus coûteux après le retrait d'un rectangle. Cela rend difficile de concevoir des mouvements incrémentaux dans des métaheuristiques ou des algorithmes complets d'optimisation.

Cet article explore la possibilité de violer la propriété BL. A la place, nous proposons de maintenir simplement un ensemble de "trous maximaux", ce qui permet l'ajout mais aussi le retrait incrémental de rectangles.

Pour valider notre approche alternative, nous avons conçu un mouvement incrémental, qui maintient les trous maximaux, pour le problème du strip-packing, une variante du problème de placement de rectangles. Nous avons également utilisé des heuristiques initiales gloutonnes standard. et la métaheuristique `ID Walk` pour réaliser une recherche locale basée sur ce mouvement.

Les résultats expérimentaux montrent que l'approche est compétitive avec les meilleurs algorithmes incomplets, plus spécialement avec les autres métaheuristiques (dotées de mécanismes pour sortir des mimima locaux).

1 Introduction

Les problèmes de découpe consistent à placer un ensemble de pièces donné dans des conteneurs sans recouvrement. Les variantes diffèrent dans la dimension considérée (1D, 2D ou 3D), dans le type de pièces (rectangles, polygones en 2D), ou dans des contraintes additionnelles : les opérations de découpe doivent-elles se faire sur toute la largeur de la pièce (coupe guillotine) ou non, l'orientation des objets est-elle fixée ou à déterminer ? Le problème du strip-packing en 2D étudié dans cet article consiste à trouver une façon de placer des rectangles de hauteur et largeur données sans recouvrement sur une bande de largeur donnée et de hauteur infinie. Le but est alors de minimiser la hauteur de bande utilisée.

Ces problèmes ont de nombreuses applications pratiques, comme la découpe de pièces de tissu ou de métal. En 3D, ils apparaissent dans le transport d'objets dans des conteneurs. La plupart de ces problèmes sont NP-difficiles et les principales méthodes pour les résoudre sont à base d'algorithmes combinatoires complets, d'heuristiques de placement gloutonnes, de métaheuristiques et d'algorithmes génétiques.

Pour limiter la combinatoire, la plupart des algorithmes en 2D se limitent à l'étude de configurations ayant la propriété BL (Bottom-Left), c-à-d où tous les rectangles sont calés en bas et à gauche. Ils maintiennent aussi cette propriété quand un rectangle R est enlevé du conteneur (ou placé ailleurs), ce qui implique souvent que les rectangles au dessus et à droite de R soient bougés aussi. Cette propriété est justifiée par le fait qu'il existe toujours une solution optimale BL et que l'on peut toujours transformer une configuration en configuration BL de hauteur inférieure ou égale. Mais cette opération n'est pas locale et peut modifier l'ensemble des rectangles.

Après un rappel des algorithmes existants en partie 2, nous présentons en partie 3 comment ajouter/enlever un rectangle. Ces opérations sont originales dans le fait qu'elles maintiennent de manière incrémentale un ensemble de *trous maximaux*, sans garder forcément la propriété BL.

La partie suivante montre de manière expérimentale que des algorithmes de recherche locale, sans maintenir la propriété BL, ne fragmentent pas la configuration, c-à-d ne produisent pas de nombreux trous. La partie 4 introduit un nouveau mouvement incrémental pour le strip packing 2D, qui maintient la structure de données des trous maximaux lors de l'addition et du retrait d'un rectangle. Les expérimentations présentées dans la partie 5 montrent l'intérêt d'une métaheuristique utilisant ce mouvement.

2 Algorithmes existants pour le strip-packing 2D

De nombreux algorithmes différents ont été proposés pour traiter le problème du strip-packing.

Les approches exactes sont en général limitées à de petites instances [13]. L'algorithme de branch and bound proposé par Martello et al. est intéressant [15] et peut résoudre certaines instances jusqu'à 200 rectangles. Cet algorithme calcule de bonnes bornes obtenues par des considérations géométriques et une relaxation du problème.

E. Hopper, dans sa thèse [10] décrit de manière exhaustive les algorithmes incomplets existants alors. Nous rappelons juste ici quelques heuristiques à partir d'algorithmes constructifs gloutons jusqu'à des métaheuristiques complexes et des algorithmes génétiques.

Bottom Left Fill (BLF) [9] est une généralisation de l'heuristique gloutonne proposée par Baker et al. [2] en 1980. BLF place les rectangles dans un ordre prédéfini, par exemple par largeur, hauteur ou surface décroissantes. Un rectangle R est placé dans le premier emplacement qui peut le contenir. Les emplacements (i.e., trous) sont triés suivant leur ordonnée comme premier critère et leur abcisse comme second, de telle sorte qu'un rectangle est placé le plus en bas et le plus à gauche possible. C'est pourquoi la configuration construite possède la propriété BL. Contrairement à l'algorithme présenté dans cet article, de nombreuses métaheuristiques considèrent un mouvement qui échange deux rectangles dans l'ordre suivi par BLF. C'est en particulier le cas de l'algorithme hybride génétique tabou conçu par Iori et al. [12].

Hopper [11] a présenté une stratégie améliorée pour BLF appelée BLD, où les objets sont ordonnés selon plusieurs critères (hauteur, largeur, surface, périmètre) et l'algorithme sélectionne le meilleur résultat obtenu. Lesh et al. in [14] ont amélioré l'heuristique BLD. Leur stratégie BLD^* répère des placements gloutons avec un ordre modifié aléatoirement autour de celui obtenu par un critère classique jusqu'à atteindre une limite de temps.

L'heuristique gloutonne (BF) proposée par Burke et al. dans [7] adopte une autre stratégie de calage en bas. A chaque pas, on considère l'emplacement le plus bas dans la solution et on y place le rectangle remplissant au mieux cet emplacement, s'il en existe un [1]. Dans [8], Burke et al. améliorent leur approche en utilisant une métaheuristique (recherche taboue, recuit simulé ou algorithme génétique) pour réparer la dernière partie de la solution obtenue par BF.

Finalement, deux dernières approches doivent être mentionnées et constituent nos principaux compétiteurs. Bortfeldt [6] propose un algorithme génétique sophistiqué qui travaille directement sur la géométrie (en forme de couches) de la configuration. Le meilleur algorithme pour traiter le strip-packing avec des rectangles d'orientation fixe est l'algorithme GRASP réactif [1] suivant : tous les rectangles sont d'abord placés avec une heuristique gloutonne BF améliorée et rendue aléatoire. Quelques rectangles au dessus sont alors enlevés et replacés avec l'algorithme glouton (dans un ordre différent). Plusieurs étapes sont ainsi réalisées avec un nombre croissant de rectangles, selon une stratégie de type VNS (recherche à voisinage variable) [17].

3 maintenir les "trous maximaux"

L'idée principale de notre approche est de maintenir de manière incrémentale l'ensemble des trous maximaux quand des rectangles sont ajoutés ou enlevés.

Definition 1 *(Trou maximal) Considérons un conteneur C partiellement rempli avec un ensemble S de rectangles. Un trou maximal H (pour C et S) est un rectangle vide dans C tel que :*

- *H n'intersecte aucun rectangle de S (i.e., H est un trou),*
- *H est maximal, i.e., il n'existe pas de trou H' tel que H soit strictement inclus dans H' (notation : l'inclusion d'un rectangle H dans un rectangle H' est notée par $H \subset H'$)*[2]

La plupart des algorithmes existants peuvent utiliser un tel ensemble de trous maximaux pour implanter leurs opérations atomiques. En particulier, les heuristiques gloutonnes BF et BLF peuvent implanter les emplacements possibles pour placer les rectangles avec l'ensemble des trous maximaux.

L'intérêt d'une telle structure est plus important encore. Nous affirmons qu'il est possible de concevoir des

[1] Trois variantes existent en fait où le rectangle est calé à gauche, vers le rectangle le plus haut ou le plus bas. L'heuristique retourne la meilleure solution obtenue par ces trois variantes.

[2] Cette propriété implique que H est calé en bas et à gauche par rapport au conteneur ou aux rectangles de S.

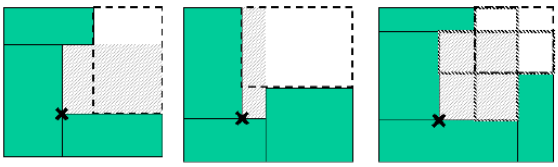

FIG. 1 – Exemples de trous maximaux

algorithmes où le nombre de trous maximaux reste petit en pratique durant la recherche, même si les rectangles sont retirés en violant la propriété BL. L'idée est la suivante. Quand un rectangle est retiré, nous ne modifions pas la solution partielle pour la rendre BL. Nous nous contentons de mettre à jour l'ensemble des trous maximaux. Ainsi, un rectangle R placé ensuite dans le conteneur sera calé en bas et à gauche s'il est placé dans un trou maximal pas trop grand. [3] L'intérêt principal est de limiter l'ensemble des emplacements possibles pour les rectangles aux trous maximaux en préservant le caractère incrémental d'un retrait de rectangle. Dans un sens, les bons résultats obtenus par la métaheuristique présentée dans cet article justifient cette approche.

Opérations atomiques entre deux trous

Dans la suite, un rectangle est représenté par quatre coordonnées : $R.x_L$, $R.y_B$, $R.x_R$, $R.y_T$: $(R.x_L, R.y_B)$ représente le coin bas-gauche de R tandis que $(R.x_R, R.y_T)$ est le coin haut-droit.

Les ajouts et retraits de rectangles dans un conteneur sont basées sur deux opérations entre rectangles et trous maximaux. L'opération **Minus**(H, R) entre un trou H et un rectangle R est utilisée quand R est ajouté dans le conteneur. Elle retourne l'ensemble des trous maximaux qui restent quand R intersecte H. Un calcul simple des nouveaux trous maximaux créés (`Holes`) est réalisé comme suit :

1. `Holes` $\leftarrow \emptyset$
2. `If` $R.x_R < H.x_R$ `then Holes` $\leftarrow$ `Holes` $\cup$ $\{(R.x_R, H.y_B, H.x_R, H.y_T)\}$ `EndIf`
3. `If` $R.y_T < H.y_T$ `then Holes` $\leftarrow$ `Holes` $\cup$ $\{(H.x_L, R.y_T, H.x_R, H.y_T)\}$ `EndIf`
4. `If` $H.x_L < R.x_L$ `then Holes` $\leftarrow$ `Holes` $\cup$ $\{(H.x_L, H.y_B, R.x_L, H.y_T)\}$ `EndIf`
5. `If` $H.y_B < R.y_B$ `then Holes` $\leftarrow$ `Holes` $\cup$ $\{(H.x_L, H.y_B, H.x_R, R.y_B)\}$ `EndIf`

[3] Dans la métaheuristique présentée ensuite, nous avons laissé un "petit" rectangle mis dans un grand trou ne pas être BL. Nous laissons à la fonction objectif le soin de sélectionner entre les voisins candidats.

`Minus`(H, R) peut créer moins de quatre trous car les conditions testées ne sont souvent pas toutes remplies simultanémenent.

La seconde opération **Plus**(H_1, H_2) est appliquée à deux trous maximaux. Si H_1 et H_2 s'intersectent ou sont contigus, `Plus` retourne au plus les deux nouveaux trous maximaux suivants :

- ($\max(H_1.x_L, H_2.x_L)$, $\min(H_1.y_B, H_2.y_B)$, $\min(H_1.x_R, H_2.x_R)$, $\max(H_1.y_T, H_2.y_T)$)
- ($\min(H_1.x_L, H_2.x_L)$, $\max(H_1.y_B, H_2.y_B)$, $\max(H_1.x_R, H_2.x_R)$, $\min(H_1.y_T, H_2.y_T)$)

Ajout et retrait de rectangles

Nous basant sur ces deux opérations, nous présentons maintenant les deux procédures principales utilisées dans la plupart des algorithmes traitant un problème de placement de rectangles 2D. `AddRectangle` et `RemoveRectangle`.

`AddRectangle`(in R, in/out C, in/out S) met à jour l'ensemble des trous maximaux de S quand le rectangle R est placé dans le conteneur. `AddRectangle` applique essentiellement l'opérateur `Minus` à R et aux trous de S qui intersectent R, comme suit :

1. Ajouter R dans C.
2. Pour chaque trou de S intersectant R, ajouter dans un emsemble `setH` les trous retournés par `Minus`(H, R).
3. Filtrer `setH` et ne garder que les trous *maximaux*.

Remarques :

- Il peut arriver que deux trous H_1 and H_2, créés par deux appels différents à `Minus`, vérifient $H_1 \subset H_2$. Cela justifie la troisième étape.
- (Correction) Une démonstration par l'absurde nous permet de comprendre qu'un trou nouvellement créé n'a pas besoin d'être fusionné avec un trou contigu H' qui n'intersecte pas R pour (récursivement) construire un trou plus grand. Sinon, cela signifierait que H' n'était pas maximal. Ce point a un impact significatif et positif sur l'efficacité de la procédure qui ne prend en compte que les trous intersectant R (et non leurs "voisins").

La procédure `RemoveRectangle` est un peu plus complexe. `RemoveRectangle` (in R, in/out C, in/out S) met à jour l'ensemble des trous maximaux de S quand le rectangle R est retiré du conteneur C. Il remplace R par un trou H et applique l'opération `Plus` sur H et ses trous contigus (s'il y en a). Un processus de point fixe est appliqué pour obtenir la complétude. Des détails sont donnés ci-après.

Proposition 1 *(Terminaison et correction de* `RemoveRectangle`*) Soit R un rectangle à retirer d'un conte-*

```
algorithm RemoveRectangle (in R, in/out C, in/out S)
    Remove R from C ; Add R in S
    Initialize a list Holes with holes in S that are contiguous with R
    Initialize a list HolePairs with pairs (R, H) such that H is in Holes
    while HolePairs is not empty do
        Select a pair of holes (hole1, hole2) in Holepairs
        Remove (hole1, hole2) from HolePairs
        newHoles ← Plus (hole1, hole2) /* newHoles contains at most 2 holes */
        for every newHole in newHoles do
            for every hole in Holes /* Ensure maximality */ do
                if newHole ⊂ hole then
                    delete newHole from newHoles ; break
                else
                    if hole ⊂ newHole then
                        delete hole from Holes
                    end
                end
            end
            if newHole ∈ newHoles /* Update Holes and HolePairs */ then
                Add newHole to Holes
            else
                Add to HolePairs all the pairs (newHole, H) such that H is in Holes
            end
        end
    end
end.
```

neur C, soit S l'ensemble correspondant de trous maximaux.

Un appel à `RemoveRectangle` *termine et met à jour S avec l'ensemble de tous les trous maximaux de C.*

Démonstration. (esquisse ; la démonstration complète est faite par récurrence).

La terminaison est basée sur plusieurs points :

- Comme pour `AddRectangle`, il n'est pas nécessaire de voir les trous qui n'ont pas été mis au départ dans `HolePairs` (c-à-d, la procédure ne visite que les voisins de R).
- Comme l'opération `Plus` ne retourne pas plus de deux trous, le nombre de trous dans `Holes` n'augmente jamais durant l'exécution de `RemoveRectangle`.
- Les points ci-dessus et la définition de `Plus` impliquent que l'union de tous les trous créés durant l'exécution de `RemoveRectangle` ne change pas. En d'autres termes, la "surface" couverte par tous les trous considérés (R et les trous voisins) est constante durant l'exécution de `RemoveRectangle`.
- L'opération `Plus`(H_1, H_2) crée au plus deux trous H'_1, H'_2 qui sont plus grands ou égaux aux trous H_1 et H_2.

Ces points expliquent pourquoi un point fixe est atteint. La correction est assurée par l'application exhaustive de l'opération `Plus` à chaque paire possible. □

La procédure `RemoveRectangle` peut être utilisée par un algorithme de placement 2D satisfaisant la propriété BL : quand un rectangle est retiré (ou placé ailleurs dans le conteneur), les rectangles déjà placés doivent être bougés pour retrouver la propriété BL, et ainsi limiter la "fragmentation" du conteneur. Cependant, cet article explore la possibilité de ne rien faire après un retrait de rectangle. Une telle approche est décrite ci-après.

4 Un mouvement incrémental pour le strip-packing

Le strip-packing est une variante du placement de rectangles 2D. Un ensemble de rectangles doivent être placés dans *un* conteneur, appelée *bande*. Dans toute la suite, cette bande sera considérée comme verticale : elle a donc une largeur fixée et une hauteur variable. Le but est de placer tous les rectangles sur la bande sans recouvrement en utilisant une hauteur minimale.

Comme cela a été mentionné dans l'introduction, comme on travaille en plus d'une dimension, les objets placés dans le conteneur sont très dépendants et il est difficile de concevoir une réparation incrémentale (et donc non coûteuse) de la solution courante. Ceci ex-

plique pourquoi les métaheuristiques existantes ou les algorithmes génétiques ne sont pas réellement incrémentaux. La plupart des approches sont basées sur une heuristique gloutonne classique, comme BLF ou BF, et un mouvement consiste par exemple à échanger deux rectangles dans l'ordre avec lequel l'heuristique gloutonne va les placer. Ainsi, échanger deux rectangles i and j, i ayant été placé avant j, dans l'ordre implique de repositionner tous les rectangles qui avaient été placés après i, et dans le pire des cas, de replacer tous les rectangles.

Dans cet article, nous proposons une métaheuristique générique basée sur un mouvement plus incrémental. Cet algorithme de recherche locale utilise une métaheuristique quelconque M (comme la recherche taboue ou le recuit simulé) et un algorithme glouton G (comme BF ou BLF). Il utilise aussi un mouvement qui va prendre un rectangle sur le haut du placement courant et va le mettre ailleurs sur la bande, en modifiant *localement* les placements des autres rectangles pour respecter la contrainte de non-recouvrement. Plus précisément, un mouvement est implanté comme suit :

1. Prendre un rectangle R dont le haut est sur la ligne la plus haute de la bande (il peut y avoir plusieurs rectangles candidats).
2. Sélectionner un emplacement R', qui est un rectangle placé ou un trou maximal , tel que quand R sera placé dans le coin bas gauche de R' :
   - R reste à l'intérieur de la bande,
   - le haut de R est strictement plus bas que dans son ancienne position.
3. Avec `RemoveRectangle`, enlever l'ensemble S des rectangles qui intersectent R dans sa nouvelle position, incluant R' si c'est un rectangle (les rectangles de S devront être replacés ailleurs).
4. Placer R dans la nouvelle position choisie à l'étape 2.
5. Replacer les rectangles de S avec l'heuristique gloutonne G.

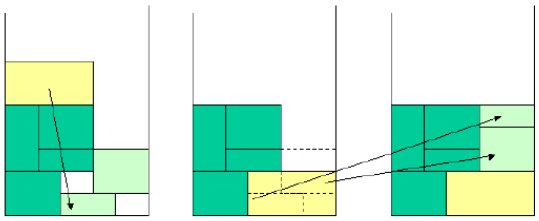

FIG. 2 – un mouvement complet

Les étapes 1 et 2 recherchent des meilleures solutions de manière agressive (intensification). Un mouvement similaire avait été mentionné dans [1] mais n'avait pas été utilisé dans l'algorithme final qu'ils ont proposé.

L'évaluation de l'objectif à minimiser est l'ordonnée du plus haut segment d'un rectangle sur la bande. Cependant, nous avons choisi une fonction objectif plus fine égal à $w \times h + n$, où w est la largeur de la bande, h l'ordonnée la plus haute occupée et n le nombre d'unités remplies par des rectangles sur la plus haute ligne. Cette fonction permet de mieux discriminer les différentes configurations qui ont la même hauteur, en favorisant celles qui ont une dernière ligne peu occupée et devraient être plus faciles à améliorer.

Pour la variante qui permet aux rectangles de tourner de 90 degrés, le mouvement est modifié de la manière suivante. L'étape 1 doit aussi choisir une orientation pour le rectangle R (aléatoirement avec probabilité 0.5 de tourner).

Nous indiquons dans le tableau 1 quelques statistiques obtenues dans les expérimentations réalisées sur les instances de Hopper et Turton (5). Ces résultats montrent que le nombre d'emplacements possibles pour un rectangle (trous maximaux et rectangles placés) croît de manière linéaire en fonction du nombre de rectangles et que le nombre de rectangles déplacés dans un mouvement reste toujours très faible en moyenne.

Classe	N	maxpl	rectmax	rectav
C1	16–17	33	8	1.9
C2	25	44	10	1.6
C3	28–29	55	11	2.0
C4	49	94	16	2.5
C5	73	128	18	2.4
C6	97	175	19	2.6
C7	196–197	342	25	2.8

TAB. 1 – Statistiques sur les instances H-T : N : nombre de rectangles, maxpl : nombre maximum d'emplacements possibles, rectmax (rectav) : nombre maximum (moyen) de rectangles déplacés dans un mouvement.

Métaheuristique et heuristiques gloutonnes choisies

Nous avons voulu proposer une métaheuristique générique pouvant être spécialisée avec de nombreuses métaheuristiques et heuristiques gloutonnes.

Nous avons essayé dans des expérimentationss préliminaires les principales métaheuristiques disponibles dans la bibliothèque logicielle `INCOP` en `C++` [18] développée par le premier auteur : recherche taboue, recuit simulé, sa variante Metropolis à température constante et `ID(best)` [19] qui est une variante simple

de `ID Walk` avec un seul paramètre [4]. Le recuit simulé a été écarté, ayant obtenu la plus mauvaise performance lors de ces essais préliminaires. La recherche taboue, `ID Walk` avec deux paramètres et `ID(best)` ont donné de bons résultats, et nous avons choisi `ID(best)` pour sa simplicité. En particulier, la procédure de réglage automatique fournie par `INCOP` [19] est plus robuste quand il n'y a qu'un paramètre à régler.

`ID(best)` est une stratégie à liste de candidats qui utilise un paramètre `MaxNeighbors` pour réaliser un mouvement d'une configuration x à une configuration x' de la manière suivante :

1. `ID(best)` choisit aléatoirement des voisins candidats un par un et les évalue. Le *premier* voisin x' avec une évaluation meilleure ou égale à celle de x est accepté.
2. Si `MaxNeighbors` voisins ont été rejetés, alors le *meilleur* voisin parmi eux (avec une valeur de la fonction objectif strictement plus grande que celle de x) est choisi.

`ID(best)` a un comportement commun avec la recherche taboue quand tous les candidats ont été rejetés (point 2 ci-dessus). Cependant, les différences consistent en l'absence d'une liste taboue et en une autre manière de choisir un x' (étape 1 ci-dessus) : avec la recherche taboue (quand seulement une portion du voisinage est examiné), tous les `MaxNeighbors` candidats sont examinés et le meilleur d'entre eux (et non le premier) non tabou est retourné.

Notre algorithme fonctionne avec une heuristique gloutonne standard : BF ou BLF, en les considérant avec trois ordres possibles pour les rectangles : largeur décroissante (w), hauteur décroissante (h), surface décroissante (s), donnant ainsi six combinaisons possibles : `BFw`, `BFh`, `BFs`, `BLFw`, `BLFh`, `BLFs`.

Le fonctionnement de la métaheuristique est le suivant :

1. l'heuristique gloutonne place tous les rectangles sur la bande (configuration initiale),
2. la procédure de réglage automatique gère les appels à `ID(best)` avec différentes valeurs de `MaxNeighbors` le long de la marche pour réparer la première solution.

Une variante prometteuse de cette métaheuristique incorpore directement le choix de l'heuristique gloutonne dans le voisinage : en plus du choix du rectangle R à replacer et de l'emplacement où le mettre, on choisit aussi l'heuristique gloutonne pour replacer les rectangles déplacés pour faire la place à R.

[4] `ID Walk` a deux paramètres `MaxNeighbors` et `SpareNeighbor`. `SpareNeighbor` peut prendre une valeur entre 1 (`ID(any)`) and `MaxNeighbors` (`ID(best)`).

Cette variante présente deux avantages. D'abord, elle évite à l'utilisateur d'avoir à choisir parmi les (six) heuristiques de placement disponibles. Plus précisément, on obtient ainsi avec `ID(best)` et le réglage automatique de son paramètre `MaxNeighbors` une métaheuristique sans paramètre. Ensuite, la variante évite certains biais dus à l'utilisation systématique d'une heuristique. Par exemple, il est reconnu que pour le placement 2D avec rotation possible des rectangles, BF (avec la propriété BL) place beaucoup de rectangles verticalement sur la droite de la bande.

On peut trouver une description de la procédure de réglage automatique dans [19]. Chaque essai est indépendant et est interrompu quand le temps CPU alloué est atteint. Un essai est une succession d'une phase de réglage automatique, où le paramètre `MaxNeighbors` est réglé de manière dichotomique sur des marches courtes et d'une phase d'exploration pendant laquelle le paramètre est gardé fixe pour une longue marche. Ces phases sont ensuite répétées avec des nombres de mouvements croissants.

Le même protocole a été suivi pour les exemples testés. Dans un même essai, la première phase de réglage réalise en tout 24 marches (avec 6 valeurs différentes pour le paramètre) de 200 mouvements chacune ; la première phase d'exploration est une marche de 10000 mouvements ; la deuxième phase de réglage réalise 24 marches de 800 mouvements et la deuxième phase d'exploration une marche de 40000 mouvements, et ainsi de suite. Ainsi, environ 30% du temps est consacré au réglage.

La valeur initiale de `MaxNeighbors` a été arbitrairement mise à la moitié du nombre de rectangles à positionner. Pour la variante avec rotation des rectangles, cette valeur est égale au nombre de rectangles. Bien que simple, cette procédure s'est avérée robuste (elle produit une bonne configuration) dans la grande majorité des essais.

5 Expérimentations

Problèmes testés

Nous avons réalisé des expérimentations sur 5 séries regroupant 547 instances. Les 21 instances sans vide de Hopper et Turton [11] sont réparties en 7 classes de nombres de rectangles croissants. Les résultats sont dans le tableau 2.

Le tableau 3 montre les résultats obtenus sur les 13 instances *gcut* de Beasley [3], les 3 instances *cgcut* et les 10 instances *beng* de Bengtsson [4]. Les résultats obtenus sur les 12 instances *ngcut* proposées aussi par Beasley ne sont pas indiqués car ils sont tous résolus de manière optimale par notre métaheuristique (en moins

de 3 secondes) et par les concurrents. On notera que les instances *gcut* ont un nombre raisonnable de gros rectangles mais ont une grande largeur de bande allant de 250 à 3000 unités.

Le tableau 4 inclut les résultats obtenus sur les 500 instances proposées par Martello et Vigo [16], Berkey et Wang [5]. Ce grand nombre d'instances est divisé en 10 classes, elles-mêmes subdivisées en 5 séries de 10 instances. Les classes définissent différentes largeurs de bande allant de 10 à 300. Les 5 séries définissent des instances avec respectivement 20, 40, 60, 80 ou 100 rectangles.

Finalement, le tableau 5 montre les résultats obtenus sur les instances de Hopper et Turton [11] pour la variante du problème de strip-packing où les rectangles peuvent tourner avec un angle de 90 degrés.

Compétiteurs

Tous les compétiteurs n'ont pas testé les cinq séries de problèmes. Ils ont aussi adopté des conditions expérimentales différentes.

L'algorithme hybride tabou/génétique de Iori [12] a été exécuté pendant 300 secondes sur un Pentium III à 800 Mhz. L'algorithme BLD* de Lesh et al. [14] a tourné sur un Pentium à 2 Ghz. Les deux résultats présentés correspondent à des limites de temps de respectivement 60 s and 3600 s.

Les résultats présentés par Burke correspondent à son heuristique BF améliorée avec de la recherche taboue, du recuit simulé ou un algorithme génétique, Cette heuristique est exécutée 10 fois avec une limite de temps de 60 secondes par exécution sur un Pentium IV à 2 Ghz. L'algorithme génétique de Bortfeldt est exécuté 10 fois sur chaque instance avec un temps moyen de 160 secondes sur un Pentium à 2 Ghz. L'algorithme GRASP [1] est lui exécuté 10 fois sur chaque instance avec une limite de temps de 60 secondes sur un Pentium IV Mobile à 2 Ghz.

Conditions expérimentales

Pour une catégorie donnée de problèmes, notre métaheuristique suit un ou les deux protocoles suivants. Dans toute la suite, notre métaheuristique notée `ID Walk` correspondra à la variante `ID(best)` décrite précédemment. Une première approche exécute `ID Walk` avec les deux heuristiques gloutonnes les plus prometteuses, 10 exécutions pour chacune (les deux heuristiques sélectionnées sont indiquées dans les tableaux) : une instance est ainsi résolue 20 fois, avec un temps de 100 (parfois 1000) secondes sur un Pentium IV à 2.66 Ghz pour chaque résolution. Une seconde approche utilise la variante de notre métaheuristique sans paramètre : une instance est résolue seulement 10 fois dans les mêmes conditions. Ainsi, nous donnons à chaque instance un temps total de 2000 ou 1000 (parfois 20000 ou 10000) secondes selon le protocole sélectionné.

Pour toutes les heuristiques, nous indiquons les meilleurs résultats obtenus. Les résultats en moyenne ne sont pas indiqués, car ils ne sont pas toujours présents dans la littérature et sont même sans signification pour certains algorithmes (y compris notre métaheuristique avec deux heuristiques gloutonnes différentes).

Les tableaux 2, 3, 4 regroupent les résultats expérimentaux obtenus pour les problèmes à orientation fixe tandis que le tableau 5 montre ceux de la variante avec rectangles à orientation variable.

Commentaires

- Tableau 2 : GRASP surclasse les autres algorithmes, surtout sur les classes 6 and 7 qui ont le plus de rectangles. `ID Walk` est généralement meilleur que les autres compétiteurs (GRASP excepté). Bortfeldt se comporte bien sur la classe 7.
- Tableau 3 : sur les 3 catégories présentées, on peut noter aussi que GRASP est meilleur que `ID Walk`, lui-même meilleur que l'algorithme de Iori. On notera cependant que `ID Walk` est meilleur que GRASP sur 3 instances *gcut* qui ont une bande large (en gras).
  `ID Walk` (même en 1000 secondes) donne des résultats similaires à ceux de Lesh en 3600 secondes. `ID Walk` trouve la solution optimale des instances *beng*, sauf *beng07*.
- Tableau 4 : du meilleur au plus mauvais, l'ordre entre les compétiteurs est GRASP, `ID Walk`, algorithme de Bortfeldt, algorithme de Lesh, algorithme de Iori.
- Tableau 5 : pour la variante de strip-packing avec une orientation des rectangles variable, l'algorithme de Bortfeldt se comporte très bien, sauf sur les petites instances[5]. L'approche de Hopper et Turton n'est pas compétitive avec notre métaheuristique.

Remarques générales :

- L'algorithme de Lesh se comporte bien, mais n'améliore pas beaucoup sa solution en dépensant plus de temps (de 60s à 3600s). Cela montre la limite des approches gloutonnes pures, même répétées sans métaheuristique pour s'échappper des minimums locaux.
- La variante automatique de `ID Walk` se comporte bien. Elle est souvent meilleure que l'algorithme

[5] A notre avis, cela peut être dû à la partie de post-traitement pour traiter les instances sans coupes guillotines.

Classe	Larg.	N	Opt.	IDW-1000	IDW-2*1000	IDW-100	Iori	Lesh	Burke	Bortfeldt	GRASP
C1	20	16–17	20	0.00	0.00	0.00	1.59	–	0.00	1.59	0.00
C2	40	25	15	0.00	0.00	0.00	2.08	–	6.25	2.08	0.00
C3	60	28–29	30	1.08	1.08	2.15	2.15	–	3.23	3.23	1.08
C4	60	49	60	1.64	1.64	1.64	4.75	–	1.64	2.70	1.64
C5	60	73	90	1.10	1.46	1.81	3.92	2.17	1.46	1.46	1.10
C6	80	97	120	1.37	1.37	1.37	4.00	1.64	1.37	1.64	0.83
C7	160	196.3	240	1.64	1.90	1.77	–	–	1.77	1.23	1.23
# solutions optimales				8/21	8/21	7/21	5/18	0/6	3/21	4/21	8/21

TAB. 2 – Comparaison sur les instances de Hopper et Turton. Chaque classe contient 3 instances avec une largeur donnée (colonne *Larg.*), un nombre donné de rectangles (N), et un optimum connu (*Opt.*). Les cellules donnent l'écart moyen de l'optimum en pourcentage. Les résultats pour l'algorithme de Burke sont ceux de la meilleure métaheuristique : BF + recuit simulé. Les résultats pour Lesh et al. ont été obtenus en 3600s. Les colonnes `IDW-1000` et `IDW-2*1000` indiquent les résultats de notre métaheuristique en resp. 1000 secondes par exécution (10 exécutions avec la variante avec heuristique gloutonne variable) et 1000 secondes par exécution (10 exécutions avec BFs and 10 exécutions avec BLFw). La troisième colonne comprend les résultats obtenus par la variante `IDW Walk` à heuristique variable en seulement 100 secondes par exécution. On notera qu'un réglage manuel de `ID Walk` avec 1000 secondes par exécution nous permet d'atteindre un écart de l'optimum de 1.37 pour la classe 7.

de base demandant à l'utilisateur de choisir une ou deux heuristiques gloutonnes et cela en deux fois moins de temps. (Cette comparaison n'est pas concluante sur les instances *gcut*.)

– Enfin, sur les problèmes de strip-packing avec des rectangles à orientation fixe, `ID Walk` est derrière GRASP, mais est généralement meilleur que les autres (algorithme de Bortfeldt inclus). Sur la variante avec orientation variable, `ID Walk` se comporte bien, mais est battu par l'algorithme de Bortfeldt sur les grosses instances.

6 Discussion

La contribution décrite dans cet article est double. Premièrement, nous avons proposé des opérateurs incrémentaux pour maintenir l'ensemble des trous maximaux lors de l'ajout ou le retrait de rectangles dans un conteneur pour tout problème de placement 2D. Nous avons suggéré de relaxer la propriété BL, qui est respectée par la plupart des algorithmes complets et incomplets. Deuxièmement, nous avons conçu une métaheuristique pour le problème du strip-packing 2D avec un mouvement incrémental portant directement sur la configuration des rectangles placés. En particulier, nous avons proposé une variante sans paramètre à régler manuellement et sans heuristique gloutonne à spécifier.

Les expérimentations nous amènent aux observations suivantes. Les meilleures heuristiques pour le strip-packing exploitent la géométrie de la configuration. En effet, les expérimentations ont écarté les métaheuristiques travaillant sur l'ordre des rectangles à placer par une heuristique gloutonne. Au contraire, l'approche de Bortfeldt et notre métaheuristique ont montré leur efficacité en travaillant directement sur la configuration.

Malgré ces résultats positifs, nous ne pouvons affirmer qu'il est vraiment intéressant de proposer des mouvements incrémentaux permettant la violation de la propriété BL. Considérons en effet les résultats obtenus par Lesh [14] et surtout ceux obtenus par GRASP, pour le moment limités dans leur implantation aux rectangles à orientation fixe. Cette approche n'est pas incrémentale. Plus précisément, elle replace tous les rectangles au dessus d'un certain seuil dans sa partie réparation de type VNS. Ces deux approches par ailleurs refont de nombreux placements initiaux complets.

Deux modifications de notre métaheuristique devraient être considérées. Peut-elle être améliorée en utilisant une heuristique gloutonne sophistiquée comme celle utilisée par GRASP ? La seconde amélioration serait de forcer de temps en temps la satisfaction de la propriété BL lors de la recherche locale.

Références

[1] R. Alvarez-Valdes, F. Parreño, and J.M. Tamarit. Reactive grasp for the strip packing problem. In *Proceedings Metaheuristic Conference MIC*, 2005.

[2] B.S. Baker, E.G. Coffman, and R.L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9 :846–855, 1980.

Instance	Larg.	N	LB	`ID Walk`	Variante	Iori	Lesh 60	Lesh 3600	GRASP
beng01	25	20	30	30	30	31	–	–	30
beng02	25	40	57	57	57	58	–	–	57
beng03	25	60	84	84	84	86	–	–	84
beng04	25	80	107	108	108	110	–	–	107
beng05	25	100	134	134	134	136	–	–	134
beng06	40	40	36	36	36	37	–	–	36
beng07	40	80	67	68	68	69	–	–	67
beng08	40	120	101	101	101	–	–	–	101
beng09	40	160	126	126	126	–	–	–	126
beng10	40	200	156	156	156	–	–	–	156
cgcut01	10	16	23	23	23	23	–	–	23
cgcut02	70	23	63	65	65	65	–	–	65
cgcut03	70	62	636	669	669	676	–	–	661
gcut01	250	10	1016	1016	1016	1016	1016	1016	1016
gcut02	250	20	1133	1204	1205	1207	1211	1195	1191
gcut03	250	30	1803	1803	1803	1803	1803	1803	1803
gcut04	250	50	2934	3066	3022	3130	3072	3054	3002
gcut05	500	10	1172	1273	1273	1273	1273	1273	1273
gcut06	500	20	2514	2656	2665	2675	2682	2656	2627
gcut07	500	30	4641	4694	4694	4758	4795	4754	4693
gcut08	500	50	5703	6192	6191	6240	6181	6081	5908
gcut09	1000	10	2022	2317	2317	–	–	–	2256
gcut10	1000	20	5356	**5973**	**5979**	–	–	–	6393
gcut11	1000	30	6537	**7037**	**6997**	–	–	–	7736
gcut12	1000	50	12522	14690	14690	–	–	–	13172
gcut13	3000	32	4772	**4962**	**4995**	–	–	–	5009

TAB. 3 – Comparaison sur les instances *beng*, *cgcut* et *gcut*. La colonne LB indique des valeurs de borne inférieure connue. Les cellules donne la meilleure solution obtenue par l'algorithme correspondant. Nous indiquons les meilleures solutions obtenues par l'algorithme de Lesh en 60 et 3600 secondes. Deux colonnes correspondent aux résultats de `ID Walk`. La première (`ID Walk`) correspond à 10 exécutions de 100 s chacune avec BLFw et 10 exécutions de 100 s avec BFw (i.e., un total de 2000 secondes). La seconde colonne (Variante) correspond à la variante avec heuristique gloutonne variable en 1000 secondes (10 exécutions de 100 secondes chaque).

Classe	Larg.	ID Walk	Glouton	Iori	Lesh 60	Lesh 3600	Bortfeldt	GRASP
01	10	0.67	BFw+BLFw	0.64	0.81	0.68	0.75	0.63
02	30	0.58	BFw+BFh	1.78	1.12	0.42	0.88	0.10
03	40	2.16	BFw+BLFw	3.05	2.71	2.23	2.52	1.73
04	100	3.47	BFw+BFh	5.08	4.41	3.54	3.19	2.02
05	100	2.20	BFw+BLFw	3.15	2.85	2.43	2.59	2.05
06	300	4.86	BFw+BFh	5.99	6.45	5.13	4.96	3.08
07	100	1.12	BFw+BLFw	1.16	1.17	1.12	1.19	1.10
08	100	4.19	BFw+BLFw	6.16	5.99	4.93	3.85	3.57
09	100	0.07	BFw+BLFw	0.07	0.07	0.07	0.07	0.07
10	100	3.12	BFw+BLFw	4.67	4.11	3.48	3.05	2.93
Overall		2.24%		3.17%	2.97%	2.40%	2.31%	1.73%

TAB. 4 – Comparaison sur les 500 instances proposées par Martello, Vigo, Berkey, Wang [16, 5]. Les cellules indiquent l'écart moyen en pourcentage avec la borne inférieure (l'optimum est généralement inconnu). `ID Walk` est exécuté 20 fois par instance, avec deux heuristiques gloutonnes (10+10) spécifiées dans la colonne *Glouton*. Cela signifie qu'un pourcentage dans une cellule provient de 1000 essais (20 essais sur les 50 instances de la classe). Un essai a une limite de temps de 100 secondes.

Classe	Larg.	N	Opt.	IDW-1000	IDW-2*1000	IDW-100	Hopper-Turton	Bortfeldt
C1	20	16–17	20	0.00	0.00	0.00	4	1.70
C2	40	25	15	0.00	0.00	0.00	6	0.00
C3	60	28–29	30	2.22	2.22	3.33	5	2.22
C4	60	49	60	1.67	1.67	1.67	3	0.00
C5	60	73	90	1.11	1.11	1.11	3	0.00
C6	80	97	120	1.11	1.11	1.67	3	0.33
C7	160	196.3	240	1.25	1.53	1.67	4	0.33
# solutions optimales				7(10)/21	7/21	6/21	0/21	15/21

TAB. 5 – Comparaison sur les instances de Hopper et Turton avec rectangles à orientation variable [11]. IDW-1000 and IDW-100 correspondent à la variante automatique de ID Walk en resp. 1000 secondes and 100 secondes par exécution. IDW-2*1000 est lancé avec BLFw et BFs (1000 secondes par exécution). Par ailleurs, ID Walk trouve les 3 optimums de la classe 3 et un optimum de la classe 5 (0.74) en 1000 secondes par exécution quand il est réglé manuellement avec BFs.

[3] J.E. Beasley. Algorithms for unconstrained two-dimensional guillotine cutting. *J. of the operational research society*, 33 :49–64, 1985.

[4] B.E. Bengtsson. Packing rectangular pieces – a heuristic approach. *The computer journal*, 25 :353–357, 1982.

[5] J.O. Berkey and Wang P.Y. Two-dimensional finite bin packing algorithms. *J. of the operational research society*, 38 :423–429, 1987.

[6] A. Bortfeldt. A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. *European Journal of Operational Research*, 172 :814–837, 2006.

[7] E. Burke, G. Kendall, and G. Whitwell. A new placement heuristic for the orthogonal stock cutting problem. *Operations Research*, 52 :697–707, 2004.

[8] E. Burke, G. Kendall, and G. Whitwell. Metaheuristic enhancements of the best-fit heuristic for the orthogonal stock cutting problem. *submitted in INFORMS, second revision*, : , 2006.

[9] B. Chazelle. The bottom left bin packing heuristic : an efficient implementation. *IEEE Transactions on Computers*, 32 :697–707, 1983.

[10] E. Hopper. *Two-Dimensional Packing Utilising Evolutionary Algorithms and other Meta-Heuristic Methods*. PhD. Thesis Cardiff University, 2000.

[11] E. Hopper and B.C.H. Turton. An empirical investigation on metaheuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128 :34–57, 2001.

[12] M. Iori, S. Martello, and M. Monaci. *Metaheuristic algorithms for the strip packing problem*, pages 159–179. Kluwer Academic Publishers, 2003.

[13] N. Lesh, J. Marks, A. Mc. Mahon, and M. Mitzenmacher. Exhaustive approaches to 2d rectangular perfect packings. *Information Processing Letters*, 90 :7–14, 2004.

[14] N. Lesh, J. Marks, A. Mc. Mahon, and M. Mitzenmacher. New heuristic and interactive approaches to 2d rectangular strip packing. *ACM Journal of Experimental Algorithmics*, 10 :1–18, 2005.

[15] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *INFORMS Journal of Computing*, 15 :310–319, 2003.

[16] S. Martello and D. Vigo. Excact solution of the two-dimensional finite bin packing problem. *Management science*, 15 :310–319, 1998.

[17] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11) :1097–1100, 1997.

[18] B. Neveu and G. Trombettoni. Incop : An Open Library for INcomplete Combinatorial OPtimization. In *Proc. Constraint Programming, CP'03*, volume LNCS 2833, pages 909–913, 2003.

[19] B. Neveu, G. Trombettoni, and F. Glover. ID Walk : A Candidate List Strategy with a Simple Diversification Device. In *Proc. Constraint Programming, CP'04, LNCS 3258*, pages 423–437, 2004.