# Une métaheuristique pour les problèmes de packing orthogonal en deux dimensions

**S. Grandcolas**[1] **C. Pain-Barre**[1]

[1]LSIS – UMR CNRS 7296 Av. Escadrille Normandie-Niemen, F-13397 Marseille Cedex 20 - France
{stephane.grandcolas,cyril.pain-barre}@lsis.org

**Résumé**

Nous présentons dans cet article une métaheuristique de type recherche locale pour les problèmes de packing orthogonal dans un espace à deux dimensions (OPP-2). Le processus, basé sur l'approche de F. Clautiaux et al. [8], consiste à chercher d'abord les positions des objets sur l'axe horizontal. Chaque fois qu'un placement satisfaisant est rencontré, une autre procédure tente de trouver les positions des objets sur l'axe vertical, de façon à ce qu'il n'y ait pas de chevauchements. Nous avons développé deux métaheuristiques distinctes, l'une pour la recherche des positions horizontales et l'autre pour la recherche des positions verticales, en envisageant différentes stratégies. Ces métaheuristiques peuvent facilement être utilisées pour traiter des problèmes de strip packing (SPP), et nous avons implémenté deux approches différentes pour SPP. Nous avons comparé nos résultats avec ceux obtenus par les systèmes les plus performants sur une sélection de problèmes connus.

**Abstract**

In this paper we describe a local search metaheuristic for solving orthogonal packing problems in a two-dimensional space (OPP-2). The method, based on F. Clautiaux et al. approach [8], consists first in searching the positions of the items on the horizontal axis. Each time a satisfying placement is discovered, another procedure searches the positions of the items on the vertical axis, so as that no two items overlap. We propose two distinct metaheuristics, one to search the positions of the items on the horizontal axis, and the other to search the positions of the items on the vertical axis. Several strategies have been developped and tested. These metaheuristics can easily be used to solve the strip packing problem (SPP), and we implemented two approaches for SPP. We compared our results with those obtained by the most efficient systems on a selection of SPP instances.

## 1 Introduction

Les problèmes de packing constituent une grande famille de problèmes dans le domaine de la recherche opérationnelle (bin packing, packing orthogonal, strip-packing, sac à dos en deux dimensions,...). Nous nous intéresserons dans cet article aux problèmes de packing en deux dimensions. Ces problèmes ont fait l'objet de nombreux travaux, du fait que leur description est très simple, et qu'ils correspondent à des problèmes réels très courants. Le problème de packing orthogonal (OPP) est le plus simple : il consiste à déterminer si $n$ objets rectangulaires de largeurs $w_1, \dots w_n$ et de hauteurs $h_1, \dots, h_n$ peuvent être placés à l'intérieur d'une boîte rectangulaire de largeur $W$ et de hauteur $H$ sans qu'il y ait de chevauchements entre les objets. Le problème de strip packing (SPP) est plus connu : il consiste à placer un ensemble d'objets rectangulaires dans une bande de largeur donnée mais de hauteur infinie, de façon à ce que les objets ne se chevauchent pas et que la hauteur occupée dans la bande soit la plus petite possible. Nous considèrerons dans cet article que les objets ont une orientation fixe pour ces deux problèmes (on trouve dans la littérature des méthodes qui envisagent la possibilité de rotation des objets).

SPP est un problème NP-difficile. Il est très lié au problème OPP. En effet, il est toujours possible de traiter SPP en testant successivement des problèmes OPP de hauteurs différentes. Les approches exactes les plus connues pour OPP sont la procédure de branch and bound de S. Martello et al. [14], la méthode de S. P. Fekete and J. Schepers [9] basée sur une représentation des placements des objets sur chaque axe à l'aide de graphes d'intervalles, et l'approche de F. Clautiaux et al. [8] dans laquelle on cherche d'abord les positions des objets sur l'axe horizontal avant de chercher leurs positions sur l'axe vertical. Ces approches ont des coûts élevés qui ne permettent pas de traiter des problèmes de taille importante. Il existe de nombreuses mé-

thodes à base d'heuristiques pour le problème SPP. Certaines utilisent l'algorithme *Bottom left* de B.S. Baker et al. [2] ou l'algorithme *Bottom left fill* de B. Chazelle [6]. Ces algorithmes placent les objets dans la boîte les uns après les autres de gauche à droite en commençant par le bas de la bande. Le placement obtenu dépend uniquement de l'ordre dans lequel sont pris les objets. Ainsi de nombreuses métaheuristiques utilisent l'algorithme BL en modifiant à chaque itération l'ordre des objets afin d'améliorer le placement. On en trouve à base de recherche tabou (M. Iori et al. [12]) ou d'algorithmes génétiques, ou encore qui modifient l'ordonnacement aléatoirement en se basant sur des probabilités (N. Lesh et al. [13]).

L'algorithme *Best Fit* de E.K. Burke et al. [5] est très différent de l'approche Bottom Left. A chaque étape on choisit l'objet qui convient le mieux pour être placé dans l'espace libre disponible le plus bas. R. Alvarez-Valdes et al. [1] proposent de construire une première solution de façon similaire à BF puis de l'améliorer avec un processus aléatoire de construction de type GRASP (Greedy Randomized Adaptive Search Procedure). Enfin, récemment B. Neveu et al. [15] ont décrit une approche originale de type recherche locale qui s'attache à modifier le placement des objets dans la boîte à partir des *trous*.

Nous proposons une nouvelle métaheuristique pour le problème OPP, basée sur l'approche exacte de F. Clautiaux et al., en utilisant une recherche locale. Cette métaheuristique peut être utilisée de deux façons différentes pour les problèmes de strip packing. La première consiste en une simple recherche dichotomique sur la hauteur. La deuxième consiste à modifier la recherche locale lors de la recherche des positions sur l'axe horizontal, en diminuant la hauteur au fur et à mesure que de nouveaux placements sont découverts. Nous avons implémenté ces deux approches et nous les avons comparées avec des méthodes performantes sur des jeux de problèmes classiques.

Nous présentons la métaheuristique pour le traitement du problème OPP dans la partie 2. La partie 3 est consacrée à l'adaptation de la métaheuristique au problème SPP. Les résutats de nos expérimentations figurent dans la partie 4. Enfin nous concluons dans la 5e partie.

## 2 Une métaheuristique pour OPP

L'approche de F. Clautiaux et al. [8] consiste à chercher les positions des objets sur l'axe horizontal (c'est à dire dans la largeur de la boîte), puis à déterminer les positions des objets sur l'axe vertical (c'est à dire dans la hauteur de la boîte). Leur procédure de recherche énumère tous les placements des objets sur l'axe des $x$ qui pourraient correspondre à une solution du problème de packing. Lors de cette recherche, qualifiée de *phase externe*, chaque fois qu'un placement satisfaisant $P_x$ est découvert, une autre procédure de recherche est appelée afin de déterminer s'il existe un placement $P_y$ des objets sur l'axe vertical qui constitue avec $P_x$ une solution du problème de packing. La recherche de $P_y$ est appelée la *phase interne*.

La figure 1 reproduit une solution d'un problème OPP. Les placements $P_x$ et $P_y$ sont représentés par des ensembles d'intervalles figurant en-dessous de la boîte pour $P_x$ et sur la gauche de la boîte pour $P_y$. Ils correspondent à la *projection* des objets sur l'axe des $x$ pour $P_x$ ou sur l'axe des $y$ pour $P_y$. Ainsi la longueur des intervalles dans $P_x$ correspond à la largeur des objets, tandis que la longuer des intervalles dans $P_y$ correspond à leurs hauteurs. Le couple $(P_x, P_y)$ constitue une solution du problème de packing si tous les intervalles sont contenus dans la largeur de la boîte pour $P_x$ ou dans sa hauteur pour $P_y$, et s'il n'existe pas deux objets dont les intervalles ont une intersection non vide dans $P_x$ et dans $P_y$. En effet, si c'était le cas, il y aurait un chevauchement entre ces deux objets dans la boîte.

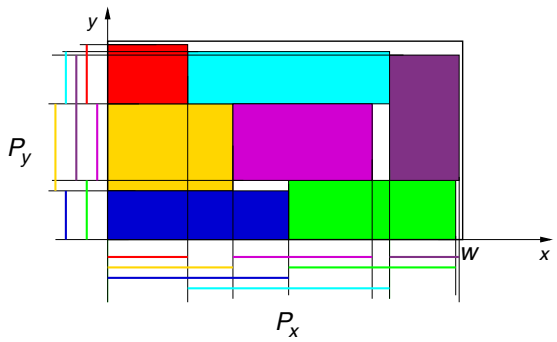


FIGURE 1 – Solution du problème OPP

Remarquons que si $(P_x, P_y)$ est une solution du problème de packing, alors en tout point $p$ de l'axe des $x$ la somme des hauteurs des objets dont l'intervalle correspondant dans $P_x$ contient $p$ est nécessairement inférieure ou égale à la hauteur de la boîte. Lors de la recherche de $P_x$, il est donc inutile de considérer les placements qui n'ont pas cette propriété. Nous appellerons cette contrainte la *contrainte de hauteur*. Elle constitue une condition nécessaire pour que $P_x$ puisse être étendu à une solution du problème de packing. Cette vérification (pour les largeurs) est inutile lorsqu'on cherche $P_y$. En effet, puisque à ce moment là les positions des objets sont fixées dans $P_x$, et puisque les intervalles correspondant à deux objets $i$ et $j$ ne doivent pas se chevaucher à la fois dans $P_x$ et dans $P_y$, un ensemble $I$ d'objets dont les intervalles s'intersectent deux à deux dans $P_y$ n'admet aucune intersection dans $P_x$. La somme des largeurs des intervalles correspondant aux objets de $I$ dans $P_x$ est donc inférieure ou égale à la largeur $W$ de la boîte.

L'approche de F. Clautiaux et al. est exacte : toutes les configurations possibles sont explorées dans la phase externe, et pour chaque placement $P_x$ qui satisfait la contrainte de hauteur, une procédure explore toutes les façons pos-

sibles de contruire $P_y$. Récemment S. Grandcolas et al. [10] ont présenté des améliorations de cette approche, qui permettent de réduire l'espace de recherche. Cependant, la taille des problèmes qui peuvent être traités en temps raisonable reste très limitée. Nous proposons dans cet article d'implémenter l'approche de F. Clautiaux et al. en calculant les placements des objets en phase externe et en phase interne à l'aide de métaheuristiques. La recherche est alors incomplète, mais il devient possible de traiter des problèmes de tailles importantes. Nous décrivons ci-dessous deux métaheuristiques pour la recherche de $P_x$ et de $P_y$, basées sur des recherches locales.

### Recherche du placement $P_x$ (phase externe)

Les positions des objets dans $P_x$ (ainsi que dans $P_y$) sont comprises entre 0 et l'infini, la position 0 correspondant à la face gauche de la boîte (ou au bas de la boîte pour $P_y$). Ainsi les objets peuvent "dépasser" à droite mais pas à gauche (ou en haut mais pas en bas dans $P_y$). La figure 2 illustre la recherche du placement $P_x$ lors de la phase externe. Nous avons ajouté au-dessus des intervalles une représentation en deux dimensions de l'espace vertical occupé par les objets en chaque point de l'axe des $x$ (c'est à dire la somme des hauteurs des objets qui recouvrent ce point). Dans cette représentation chaque objet apparaît à la position sur l'axe des $x$ qu'il occupe dans $P_x$. Les objets sont empilés les uns sur les autres dans un ordre arbitraire, et éventuellement découpés en plusieurs parties afin qu'il n'y ait aucun espace libre. Le rectangle grisé rappelle les dimensions de la boîte.

La recherche d'un placement $P_x$ satisfaisant consiste, à partir de positions des intervalles choisies aléatoirement, à appliquer des modifications élémentaires dans le but de satisfaire la contrainte d'inclusion (les intervalles doivent être contenus dans la largeur de la boîte) et la contrainte de hauteur. Une modification élémentaire se décompose en trois étapes : (1) enlever un intervalle du placement, (2) réarranger éventuellement le placement, et (3) réinsérer l'intervalle dans le placement. En fait, lors de l'insertion d'un intervalle $i$ dans le placement, seules sont considérées la position 0 (la face gauche de la boîte) et les positions correspondant à des bornes droites d'intervalles déja dans le placement (nous qualifierons ces intervalles de *supports* de l'intervalle $i$). Cette restriction n'a aucun effet sur l'issue de la recherche, si le problème de packing a des solutions alors il en existe de ce type. Remarquons enfin qu'un intervalle peut perdre tous ses supports pendant la recherche. Nous avons envisagé de réarranger $P_x$ chaque fois qu'un intervalle est enlevé, afin que tous les intervalles aient au moins un support à tout moment, en *tassant* le placement vers la gauche chaque fois que c'est nécessaire (en déplaçant les intervalles non soutenus vers la gauche, et en propageant ces décalages). Cette option est implémentée mais ne produit pas de meilleurs résultats sur les instances que nous avons traitées.

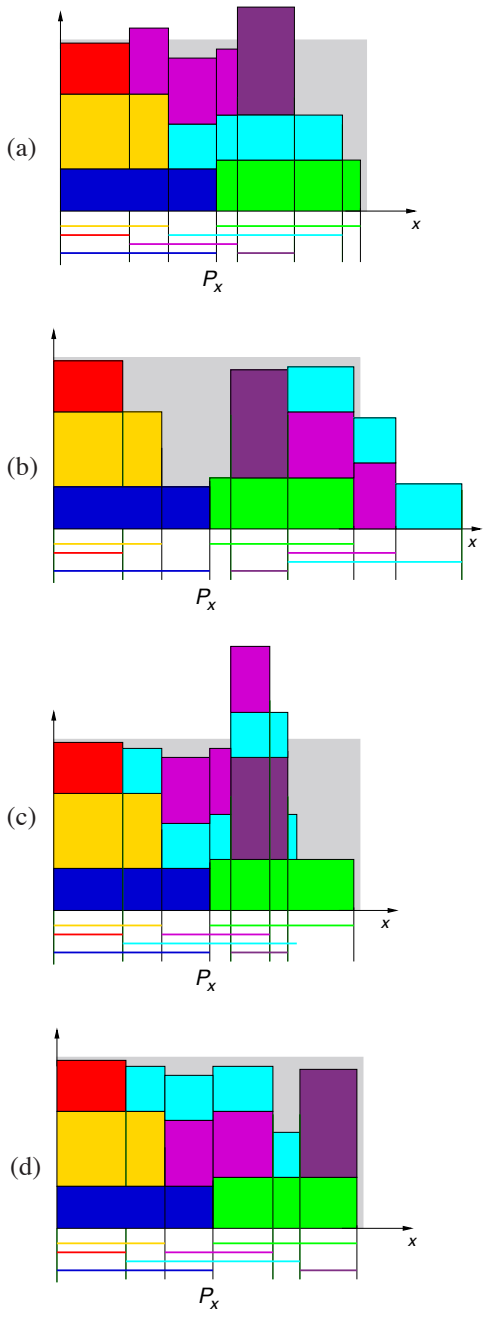


FIGURE 2 – Phase externe : contraction et écrasement

La procédure de recherche alterne des périodes de *contraction* et des périodes d'*écrasement* du placement $P_x$, tant que les contraintes d'inclusion et de hauteur ne sont pas satisfaites, et que le nombre de modifications effectuées est en-dessous d'une limite donnée. Chaque période a un objectif différent. Le changement de période intervient quand l'objectif de la période est atteint, ou quand le

nombre de modifications effectuées durant la période courante dépasse une limite donnée (fixée à 10 pour nos expérimentations).

**Contraction.** L'objectif est de placer les intervalles de $P_x$ dans la largeur de la boîte. On choisit des intervalles qui *dépassent* à droite, et on les insère dans la largeur de la boîte, quitte à générer ou à renforcer une violation de la contrainte de hauteur (voir la transition de (b) à (c) dans la figure 2 : l'objet bleu turquoise et l'objet rose, qui violent la contrainte d'inclusion, sont déplacés vers la gauche de façon à ce que les intervalles correspondants soient contenus dans la largeur de la boîte). S'il en existe, on choisit comme destination pour l'intervalle déplacé une position qui ne produit pas de dépassement de la hauteur, en privilégiant les positions qui maximisent l'espace libre dans la bande verticale délimitée par l'intervalle. Si il n'existe pas de telles positions, on privilégiera les positions qui minimisent la surface exédentaire.

**Ecrasement.** Le but est de satisfaire la contrainte de hauteur : les intervalles choisis pour être déplacés sont des intervalles impliqués dans la violation de la contrainte de hauteur. Nous avons implémenté différents choix pour les destinations des intervalles déplacés. Les meilleurs résultats ont été obtenus en choisissant une destination telle que la contrainte de hauteur soit satisfaite en tout point de l'intervalle déplacé, en privilégiant celles qui maximisent la hauteur occupée au niveau de la borne gauche de l'intervalle. Dans la figure 2, la transition de (a) à (b) avec le déplacement de l'objet rose et de l'objet bleu turquoise, et la transition de (c) à (d) avec le déplacement de l'objet violet illustrent la phase d'écrasement. En (d) le placement $P_x$ satisfait la contrainte de hauteur et la contrainte d'inclusion.

### Recherche du placement $P_y$ (phase interne)

La recherche du placement $P_y$ est assez semblable à la recherche de $P_x$, modulo le fait que, puisque les positions des objets sur l'axe des $x$ sont déja fixées, deux intervalles ne peuvent se chevaucher dans $P_y$ si les intervalles correspondant à ces mêmes objets dans $P_x$ ont une intersection non vide. Nous dirons que deux objets sont en *conflit* si les intervalles correspondant à ces objets dans $P_x$ et dans $P_y$ s'intersectent dans les deux placements. Nous avons pris le parti de considérer, pendant la recherche de $P_y$, uniquement des placements sans conflit. Afin d'éviter l'apparition de conflits chaque fois qu'un intervalle est inséré dans $P_y$, les intervalles qui sont en conflit avec cet intervalle sont décalés vers le haut (jusqu'à la borne haute de l'intervalle). Un mécanisme de propagation prend en charge ces décalages qui peuvent à leur tour générer des conflits. Le processus s'arrête lorsqu'il n'y a plus de conflits dans $P_y$.

La recherche de $P_y$ consiste, partant d'un placement aléatoire mais sans conflit, à déplacer des intervalles en éliminant chaque fois les conflits, dans le but de faire rentrer tous les intervalles de $P_y$ dans la hauteur de la boîte. Les intervalles déplacés sont choisis parmis ceux qui dépassent le sommet de la boîte. Les destinations possibles sont la position 0 (qui correspond au bas de la boîte) et les bornes supérieures des intervalles du placement. La recherche de $P_y$ prends fin dès que tous les intervalles de $P_y$ sont contenus dans la hauteur, ou dès que le nombre de déplacements effectués atteint une limite donnée.

Nous avons retenu les options suivantes pour le choix des déplacements :

**lowest increase width** On considère le déplacement d'un intervalle $i$ à la position $p$ seulement si (1) $i$ est *supporté* par un objet en conflit à la position $p$ (i.e. un intervalle correspondant à un objet qui recoupe l'objet correspondant à $i$ dans $P_x$ se termine en $p$ ; en deux dimensions cela signifie que l'objet correspondant à $i$ sera soutenu par un objet situé en-dessous) et si (2) il n'y a pas d'intervalle en conflit avec $i$ qui recouvre la position $p$. Cette dernière condition signifie que pour résoudre les conflits il n'y aura pas à déplacer des intervalles débutant avant $p$. On favorise parmi les déplacements qui satisfont ces deux conditions ceux dont la destination est la plus basse, et pour lesquel la largeur occupée au niveau de la destination augmente le plus (l'augmentation correspond à la différence entre la largeur de l'objet et la somme des largeurs des objets en conflit qui seront décalés vers le haut).

**minimize area/minimize height** Choix d'une destination telle que l'aire en conflit (ou la hauteur cumulée des conflits) est minimale.

**random** Choix d'un objet et d'une position au hasard. Si c'est possible, on choisit une position sans conflit.

Pour nos expérimentations nous avons utilisé dans 10 pour cent des cas le choix **random**, sinon le choix **lowest increase width**, et s'il n'y a pas de candidats le choix **minimize area/minimize height**.

## 3 Problèmes de Strip Packing

Le problème de strip packing (SPP) consiste à trouver la hauteur minimale qui permet de placer $n$ objets dans une bande de largeur donnée $l$. De nombreuses méthodes complètes ou non ont été publiées pour ce problème. Nous proposons deux approches. La première consiste en une recherche dichotomique sur la hauteur : la fonction SPP-dicho calcule la hauteur minimale d'une bande de largeur $l$ dans laquelle on doit placer l'ensemble d'objets $O$. Initialement la plage de recherche correspond à des bornes $inf$ et $sup$ triviales. La borne $sup$ est obtenue avec un algorithme connu, le *shelf algorithm*, qui consiste à poser les objets les uns après les autres sur un même niveau de la gauche vers la droite (comme sur une étagère d'où le

nom *shelf algorithm*), en ouvrant un nouveau niveau dès qu'il n'y a plus assez d'espace pour poser l'objet suivant. La borne $inf$ correspond simplement à la surface totale des objets divisée par la largeur de la boîte $l$. Tant que la plage $[inf, sup]$ contient plus d'une valeur, une hauteur intermédiaire $h$ est testée à l'aide de la procédure décrite dans la section précédente pour le problème OPP. Dans notre implémentation nous avons pris comme hauteur intermédiaire la moyenne des bornes $inf$ et $sup$. Initialement, le placement $P_x$ (resp. $P_y$) est généré aléatoirement par la fonction genere_$P_x()$ (resp. genere_$P_y()$). La fonction ameliore_$P_x()$ (resp. ameliore_$P_y()$) tente d'améliorer $P_x$ (resp. $P_y$) tant qu'il n'est pas valide, en effectuant au plus $nSteps_x$ (resp. $nSteps_y$) modifications, comme cela a été décrit dans la partie précédente pour le problème OPP. En phase externe le placement $P_x$ doit satisfaire la contrainte d'inclusion et la contrainte de hauteur, tandis qu'en phase interne le placement $P_y$ est astreint à respecter la contrainte d'inclusion et la contrainte de non-chevauchement définie par le graphe d'intersection des intervalles de $P_x$, noté $G_x$.

Si la procédure trouve une solution pour la hauteur $h$ on diminue la borne $sup$, sinon on augmente la borne $inf$ (on prendra par exemple la moyenne entre les bornes $inf$ et $sup$). Dans notre implémentation, si le placement $P_x$ est valide, la recherche de $P_y$ est réitérée un certain nombre de fois, en changeant les paramètres afin d'intensifier la recherche, tant qu'on ne trouve pas un placement valide. Enfin, la fonction SPP-dicho peut être appelée plusieurs fois, en mettant à jour la borne $sup$ avec le meilleur résultat rencontré lors des essais précédents.

La deuxième procédure que nous avons développée pour SPP, nommée SPP*, détermine une valeur minimale de la hauteur lors de la recherche du placement $P_x$ dans la phase externe (fonction optimise_$P_x$). Initialement, la valeur de la hauteur pour la contrainte de hauteur est $sup - 1$, où $sup$ est la borne produite par le *shelf algorithm*. Chaque fois qu'un placement $P_x$ satisfaisant la contrainte d'inclusion et la contrainte de hauteur est découvert, la fonction optimise_$P_x$ réitère la recherche avec la valeur de la hauteur diminuée d'une unité. Ainsi la hauteur diminue jusqu'à ce que la recherche locale soit incapable de produire un placement satisfaisant les contraintes d'inclusion et de hauteur. Le placement $P_x$ retenu pour la phase interne est le dernier placement valide rencontré pendant cette phase d'amélioration, c'est donc celui dont la hauteur maximale occupée $h$ est la plus petite. Si la recherche du placement $P_y$ est infructueuse, la fonction SPP* renvoie comme résultat la borne $sup$ produite par le *shelf algorithm*.

**Réglage des paramètres.** Pour chaque procédure les paramètres $nSteps_x$ et $nSteps_y$ sont choisis en fonction de la taille du problème (c'est à dire du nombre de variables). La procédure est exécutée plusieurs fois, et chaque fois les paramètres sont durcis. L'idée est de renforcer la recherche au fur et à mesure que la plage de recherche de la hauteur se restreint. Il est en effet inutile d'effectuer des recherches coûteuses lors des premiers essais, alors que les hauteurs testées peuvent être très éloignées de la hauteur optimale, et la recherche très facile ou au contrainte sans solution. Le temps de calcul est à peu près doublé entre deux essais successifs (le nombre d'étapes augmente tandis que le nombre d'essais diminue).

**Terminaison.** La recherche s'arête après un certain nombre d'essais, ou lorsque la meilleure hauteur trouvée n'évolue plus pendant plusieurs essais successifs, ou encore lorsque qu'un temps limite donné est dépassé.

**Remarque sur le coût.** Le temps de calcul de la fonction SPP-dicho est fonction de la taille du problème (le nombre

**Function** SPP-dicho($O$, $l$, $nSteps_x$, $nSteps_y$)

```
begin
    sup := borne_sup(O,l);
    RESTART :
    inf := borne_inf(O,l);
    while inf < sup do
        h := (inf + sup)/2;
        P_x := genere_Px(O);
        P_x := ameliore_Px(P_x,O,l,h,nSteps_x);
        if valide_Px(P_x,l,h) then
            G_x := calcul_chevauchements(P_x);
            P_y := genere_Py(O,G_x);
            P_y := ameliore_Py(P_y,O,h,G_x,nSteps_y);
            if valide_Py(P_y,G_x,h) then
                sup := h;
            else
                inf := h;
    if not time_out then
        if sup ≠ borne_inf(O,l) then
            goto RESTART;
    return sup;
end
```

**Function** SPP*($O$, $l$, $nSteps_x$, $nSteps_y$)

```
begin
    P_x := genere_Px(O);
    (P_x,h) := optimise_Px(P_x,O,l,nSteps_x);
    G_x := calcul_chevauchements(P_x);
    P_y := genere_Py(O,G_x);
    P_y := ameliore_Py(P_y,O,h,G_x,nSteps_y);
    if valide_Py(P_y,G_x,h) then
        return h;
    else
        return borne_max(O,l);
end
```

de variables détermine le nombre d'essais et le nombre d'étapes), mais aussi de sa *granularité*. En effet, le nombre d'exécutions de la boucle dépends de la largeur de la plage de recherche : si $inf$ et $sup$ sont les bornes initiales, le nombre d'appels est $O(\log\ (sup - inf + 1))$ comme pour une recherche dichotomique.

## 4 Résultats expérimentaux

Nous avons comparé les procédure de recherche SPP-dicho et SPP* avec des approches récentes et performantes, sur des séries de problèmes classiques (avec orientation fixe des objets). Nous avons choisi de reporter les résultats de GRASP de R. Alvarez-Valdes et al. [1] et du système de B. Neveu et al. [15] (l'option retenue est ID Walk avec l'heuristique HH et le réarrangement P, qui semble donner les meilleurs résultats). Les résultats figurent dans la table 1. Nous nous sommes limité aux jeux d'instances *gcut* de J.E. Beasley [3], *cgcut* de N. Christofides et al. [7], et *beng* proposées par B.E. Bengtsson [4]. Il s'agit de problèmes *généraux*, dans le sens où les solutions optimales ne sont pas toujours connues, et sont susceptibles de contenir de l'espace vide. Pour chaque instance figurent la moyenne des hauteurs et la hauteur minimale trouvée par chaque solveur sur une suite d'essais (10 essais avec un temps limite de 60 secondes pour GRASP et 10 essais avec un temps limite de 100 secondes pour ID Walk).

Nous avons reporté les résultats que nous avons obtenu en lançant nos procédures de recherche 12 fois pour chaque instance avec un temps limite de 1000 secondes pour chaque essai (ce temps limite a été atteint uniquement avec les instances les plus difficiles) sur des machines équipées de processeurs Xeon 2.4 GHz. Les paramètres de SPP-dicho et SPP* sont fixés au lancement du programme. Ils dépendent uniquement du nombre de variables du problème (au delà de 50 variables les paramètres n'évoluent plus). Pour certains problèmes, *cgcut*3 par exemple, il arrive que la moyenne des résultats soit fortement dégradée du fait que certains essais ont terminé sans produire de résultat, et nous avons alors utilisé la borne supérieure correspondant à un placement trivial des objets. C'est un défaut de SPP* de terminer sans avoir établi de résultat : en général la phase 1 produit très rapidement un placement $P_x$ de bonne qualité (de hauteur très petite). La recherche du placement $P_y$ en phase interne est souvent fastidieuse, et peut même ne jamais aboutir avant le temps limite. Cela explique les mauvais résultats de SPP*, qui n'est jamais meilleur que SPP-dicho.

Pour chaque série de problèmes figurent les moyennes des résultats moyens et des meilleurs résultats. Pour les problèmes *gcut* nous n'avons pas intégré dans ces moyennes les résultats des instances *gcut*9,...,*gcut*13 (indisponibles pour GRASP).

Sur les instances *gcut*, SPP-dicho est meilleur que ses concurrents (exception faite de l'instance *gcut*13 pour laquelle la hauteur minimale trouvée par ID Walk est inférieure). Les résultats qui constituent une amélioration significative apparaissent en gras. C'est le cas notamment pour les instances *gcut*4, *gcut*8, *gcut*13*r* qui sont difficiles. Ses moyennes sur la série sont meilleures que celles de GRASP et ID Walk. Inversement, sur la série *beng* SPP-dicho n'est pas performant, surtout sur les instances dont le nombre de variables est supérieur à 60. Nous pourrions faire les mêmes observations sur les instances de Hopper et Turton [11]. Dans ce cas les solutions optimales sont connues et ne laissent aucun espace vide dans la boîte. D'autre part, il y a très peu de diversité dans les dimensions des objets pour les instances *beng* (pour *beng*4 par exemple on compte 80 objets et seulement 8 largeurs différentes, alors que pour *gcut*8 on compte 50 objets avec 44 largeurs différentes). Cela conduit à des placements dans lesquels on trouve souvent des objets de même largeur (resp. hauteur) placés côte à côte qui constituent des agrégats rectangulaires. Notre méthode ne tient pas compte de ce types de configurations (GRASP par exemple tente de constituer des blocs plaçant côte à côte plusieurs pièces de mêmes dimensions). Il semble que SPP soit plus efficace lorsque les pièces sont peu nombreuses, avec des dimensions rarement égales mais assez proches (peu dispersées). D'autre part les modifications du placement $P_y$ dans la phase interne ont tendance à bouleverser le placement courant. L'insertion d'un objet produit le décalage des objets en conflits, décalages qui se propagent dans le placement et qui peuvent le changer de façon importante. Ce comportement a tendance à freiner la convergence vers un placement valide.

## 5 Conclusion

Nous avons présenté une métaheuristique pour les problèmes de packing orthogonal en deux dimensions basé sur l'approche (exacte) de F. Clautiaux [8]. Nous avons implémenté la méthode présentée, et nous l'avons adaptée au traitement de problèmes de Strip Packing. Les deux procédures de recherche SPP-dicho et SPP* que nous avons développées ont été comparées avec des approches existantes qui font référence sur des jeux d'instances classiques. Les résultats sont bons sur les instances dans lesquelles les objets sont peu semblables mais leurs dimensions assez proches.

Nous allons chercher à rendre plus efficace la recherche locale de la phase 2, en introduisant de nouvelles modifications élémentaires et en privilégiant celles qui ne modifient pas le placement de façon importante.

## Références

[1] R. Alvarez-Valdes, F. Parreño, and J.M. Tamarit. Reactive grasp for the strip-packing problem. *Com-*

| Instance | | | | GRASP | | IDW (+HH+P) | | SPP-dicho | | SPP* | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Name* | *n* | *w* | *LB* | *moy.* | *min* | *moy.* | *min* | *moy.* | *min* | *moy.* | *min* |
| gcut1 | 10 | 250 | 1016 | 1016,0 | 1016 | 1016,0 | 1016 | 1016,0 | 1016 | 1016,0 | 1016 |
| gcut2 | 20 | 250 | 1133 | 1191,0 | 1191 | 1203,9 | 1195 | 1187,0 | 1187 | 1191,1 | 1187 |
| gcut3 | 30 | 250 | 1803 | 1803,0 | 1803 | 1803,0 | 1803 | 1803,0 | 1803 | 1803,0 | 1803 |
| gcut4 | 50 | 250 | 2934 | 3002,0 | 3002 | 3032,0 | 3020 | 3003,7 | **3000** | 3009,1 | 3004 |
| gcut5 | 10 | 500 | 1172 | 1273,0 | 1273 | 1273,0 | 1273 | 1273,0 | 1273 | 1273,0 | 1273 |
| gcut6 | 20 | 500 | 2514 | 2627,0 | 2627 | 2651,1 | 2639 | 2622,0 | 2622 | 2622,0 | 2622 |
| gcut7 | 30 | 500 | 4641 | 4693,0 | 4693 | 4710,1 | 4704 | 4693,0 | 4693 | 4693,0 | 4693 |
| gcut8 | 50 | 500 | 5703 | 5912,2 | 5908 | 5947,7 | 5895 | **5877,6** | **5872** | 5934,9 | 5890 |
| gcut9 | 10 | 1000 | 2022 | | | 2317,0 | 2317 | 2317,0 | 2317 | 2317,0 | 2317 |
| gcut10 | 20 | 1000 | 5356 | | | 5972,0 | 5969 | 5964,0 | 5964 | 5964,0 | 5964 |
| gcut11 | 30 | 1000 | 6537 | | | 6994,1 | 6966 | 6869,5 | 6866 | 6894,8 | 6875 |
| gcut12 | 50 | 1000 | 12522 | | | 14690,0 | 14690 | 14690,0 | 14690 | 14690,0 | 14690 |
| gcut13 | 32 | 3000 | 4772 | | | 4975,7 | 4914 | 4953,4 | 4932 | 5133,8 | 4992 |
| gcut9r | 10 | 1000 | 2022 | 2256,0 | 2256 | 2251,3 | 2241 | 2241,0 | 2241 | 2241,0 | 2241 |
| gcut10r | 20 | 1000 | 5356 | 6393,0 | 6393 | 6427,3 | 6422 | 6393,0 | 6393 | 6393,0 | 6393 |
| gcut11r | 30 | 1000 | 6537 | 7736,0 | 7736 | 7736,0 | 7736 | 7736,0 | 7736 | 7736,0 | 7736 |
| gcut12r | 50 | 1000 | 12522 | 13172,0 | 13172 | 13213,0 | 13172 | 13172,0 | 13172 | 13172,0 | 13172 |
| gcut13r | 32 | 3000 | 4772 | 5009,5 | 5009 | 5075,4 | 5028 | 5015,5 | **5007** | 5070,7 | 5028 |
| | | | | 4314,1 | 4313,7 | 4333,8 | 4318,7 | 4310,2 | **4308,8** | 4319,6 | 4312,1 |
| cgcut1 | 16 | 10 | 23 | 23,0 | 23 | 23,0 | 23 | 23,0 | 23 | 23,0 | 23 |
| cgcut2 | 23 | 70 | 63 | 65,0 | 65 | 65,0 | 65 | 64,9 | 64 | 65,1 | 65 |
| cgcut3 | 62 | 70 | 636 | 661,0 | 661 | 667,9 | 662 | 662,0 | 661 | 721,9 | 663 |
| beng01 | 20 | 25 | 30 | 30,0 | 30 | 30,4 | 30 | 30,0 | 30 | 30,0 | 30 |
| beng02 | 40 | 25 | 57 | 57,0 | 57 | 58,0 | 58 | 58,0 | 58 | 58,5 | 58 |
| beng03 | 60 | 25 | 84 | 84,0 | 84 | 84,5 | 84 | 88,2 | 87 | 92,0 | 86 |
| beng04 | 80 | 25 | 107 | 107,0 | 107 | 107,9 | 107 | 119,4 | 116 | 126,0 | 126 |
| beng05 | 100 | 25 | 134 | 134,0 | 134 | 134,0 | 134 | 152,7 | 149 | 153,0 | 153 |
| beng06 | 40 | 40 | 36 | 36,0 | 36 | 36,0 | 36 | 36,0 | 36 | 36,0 | 36 |
| beng07 | 80 | 40 | 67 | 67,0 | 67 | 67,3 | 67 | 74,9 | 73 | 76,6 | 72 |
| beng08 | 120 | 40 | 101 | 101,0 | 101 | 101,0 | 101 | 109,0 | 109 | 109,0 | 109 |
| beng09 | 160 | 40 | 126 | 126,0 | 126 | 126,0 | 126 | 140,0 | 140 | 140,0 | 140 |
| beng10 | 200 | 40 | 156 | 156,0 | 156 | 156,0 | 156 | 171,0 | 171 | 171,0 | 171 |
| | | | | 89,8 | 89,8 | 90,1 | 89,9 | 97,9 | 96,9 | 99,2 | 98,1 |

TABLE 1 – Hauteurs moyennes et hauteurs minimales (SPP-2)


*puters & Operations Research*, 35(4) :1065 – 1083, 2008.

[2] B. Baker, E. Coffman, and R. Rivest. Orthogonal packing in two dimensions. *SIAM J. of Computing*, 9(4) :846–855, 1980.

[3] J. E. Beasley. Algorithms for unconstrained two-dimensional guillotine cutting. *The Journal of the Operational Research Society*, 36(4) :297–306, 1985.

[4] B.E. Bengtsson. Packing rectangular pieces - a heuristic approach. *Comput. J.*, 25(3) :353–357, 1982.

[5] E. K. Burke, G. Kendall, and G. Whitwell. A new placement heuristic for the orthogonal stock-cutting problem. *Oper. Res.*, 52(4) :655–671, August 2004.

[6] B. Chazelle. The bottom-left bin-packing heuristic : An efficient implementation. *IEEE Transactions on Computers*, 32(8) :697–707, 1983.

[7] N. Christofides and C. Whitlock. An Algorithm for Two-Dimensional Cutting Problems. *Operations Research*, 25(1) :30–44, January 1977.

[8] F. Clautiaux, J. Carlier, and A. Moukrim. A new exact method for the two-dimensional orthogonal packing problem. *European Journal of Operational Research*, 183(3) :1196–1211, 2007.

[9] S. P. Fekete and J. Schepers. A combinatorial characterization of higher-dimensional orthogonal packing. *Mathematics of Operations Research*, 29(2) :353–368, 2004.

[10] S. Grandcolas and C. Pinto. A new search procedure for the two-dimensional orthogonal packing problem. *Journal of Mathematical Modelling and Algorithms in Operations Research*, 14(3) :343–361, 2015.

[11] E. Hopper and C.H. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128(1) :34–57, 2001.

[12] Manuel Iori, Silvano Martello, and Michele Monaci. *Optimization and Industry : New Frontiers*, chapter Metaheuristic Algorithms for the Strip Packing Pro-

blem, pages 159–179. Springer US, Boston, MA, 2003.

[13] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. New heuristic and interactive approaches to 2d rectangular strip packing. *J. Exp. Algorithmics*, 10, December 2005.

[14] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *Journal on Computing*, 15(3) :310–319, 2003.

[15] B. Neveu, G. Trombettoni, I. Araya, and M.C. Riff. A strip packing solving method using an incremental move based on maximal holes. *International Journal on Artificial Intelligence Tools*, 17(5) :881–901, 2008.