Bin Packing with Conflicts: a generic Branch-and-Price algorithm

Ruslan Sadykov
RealOpt team, INRIA Bordeaux — Sud-Ouest, 351 cours de la Libération, 33405 Talence France
Ruslan.Sadykov@inria.fr

François Vanderbeck
Université Bordeaux I and RealOpt team, INRIA Bordeaux — Sud-Ouest, 351 cours de la Libération, 33405 Talence France fv@math.u-bordeaux1.fr

The bin packing problem with conflicts consists in packing items in a minimum number of bins of limited capacity while avoiding joint assignments of items that are in conflict. Our study demonstrates the comparatively good performance of a generic Branch-and-Price algorithm for this problem. We made use of our black-box solver BaPCod, relying on its generic branching scheme and primal heuristics, while developing a specific pricing oracle. For the case where the conflict graph is an interval graph, we developed a dynamic programming algorithm for pricing, while for the general case, we implemented a depth-first-search branch-and-bound approach. The algorithm is tested on instances from the literature where the conflict graph is an interval graph, as well as on newly generated instances with an arbitrarily conflict graph. The computational results show that our generic algorithm outperforms special purpose algorithms of the literature, closing all open instances in one hour of CPU time.

Key words: branch-and-price; bin packing; knapsack; conflict graphs; interval graphs.

1. Introduction

In the Bin Packing Problem (BPP), items of different sizes/weights must be packed into a minimum number of identical bins of given capacity. In the variant with conflicts (denoted BPPC), a graph is given where nodes represent items and edges represent conflicts between pairs of items: any two items that are linked by an edge cannot be assigned to the same bin. Thus, the problem is a combination of the Bin Packing Problem and the Vertex Coloring Problem. It arises in many real-world applications, such as examination scheduling (Laporte and Desroches, 1984), parallel computing and database storage (Jansen, 1999), product delivery (Christofides et al., 1979), resource clustering in highly distributed parallel computing
(Beaumont et al., 2008). The special case of an interval conflict graph is a realistic model on its own. It arises for instance in the mutual exclusion scheduling problem (Baker and Coffman, 1996; Gardi, 2009) in which all items or tasks are represented by time intervals associated to their schedule, and a conflict between tasks arises when the associated intervals overlap. A related application arise in workforce planning (Gardi, 2005): the problem is to assign tasks to a minimum number of workers; each task is defined by a start time and a completion time; tasks assigned to the same worker may not overlap in time; the total duration of the tasks assigned to a worker may not exceed a given bound.

The BPPC was considered by Jansen and Öhring (1997) and Jansen (1999) who developed approximation algorithms for special cases of conflicts graphs. Several computational studies on the problem have recently been published. Gendreau et al. (2004) have evaluated six heuristics and lower bounds for the problem. Different heuristics, lower bounds and an exact algorithm based on a branch-and-price approach were proposed by Fernandes Muritiba et al. (2009). A special purpose branch-and-price algorithm was developed by Elhedhli et al. (2010).

Here, we show that the BPPC can be efficiently solved by a generic branch-and-price algorithm: we use the generic software platform, BaPCod, that is developed in our team. It includes the generic branching scheme of Vanderbeck (2010) and the generic column generation based primal heuristics of Joncour et al. (2010). The pricing subproblem is a Knapsack Problem with Conflicts (KPC). We developed a specific branch-and-bound oracle for a general conflict graph and a dynamic programming solver for the special case of an interval graph. To the best of our knowledge, the latter dynamic program is an original contribution although a dynamic program for the more general case of the KPC in a chordal graph can be found in Pferschy and Schauer (2009). Our computational results demonstrates that our approach improves on the state-of-art branch-and-price algorithms proposed by Fernandes Muritiba et al. (2009) and by Elhedhli et al. (2010).

The paper is organized as follows. In Section 2, we provide a compact formulation and an extended (set covering) integer programming formulation of the problem. Our algorithm is presented in Section 3. Results of computational experiments are reported in Section 4. In Section 5, we draw conclusions.
2. Formulations of the problem

Formally, the BPPC can be described as follows. We are given a set $V = \{1, 2, \ldots, n\}$ of items and an infinite number of identical bins of capacity $W$. Each item $i \in V$ has a non-negative weight $w_i$ measuring its bin capacity consumption. We are also given a conflict graph $G = (V, E)$, where $E$ is a set of edges such that $(i, j) \in E$ when $i$ and $j$ are in conflict.

The problem is to assign items to bins, using a minimum number of bins, while ensuring that the total weight of the items assigned to a bin does not exceed the bin capacity, $W$, and that no two items that are in conflict are assigned to the same bin.

A natural and compact integer programming formulation makes use of binary variables $x_{ik}$ taking value 1 if item $i$ is assigned to bin $k$ and 0 otherwise, and binary variables $y_k$ taking value 1 if bin $k$ is used and 0 otherwise:

$$\min \sum_{k=1}^{K} y_k$$

s.t. $\sum_{k=1}^{K} x_{ik} \geq 1, \quad i = 1, \ldots, n,$

$$\sum_{i=1}^{n} w_i x_{ik} \leq W y_k, \quad k = 1, \ldots, K,$$

$$x_{ik} + x_{jk} \leq y_k, \quad (i, j) \in E, \quad k = 1, \ldots, K,$$

$$y_k \in \{0, 1\}, \quad k = 1, \ldots, K,$$

$$x_{ik} \in \{0, 1\}, \quad i = 1, \ldots, n, \quad k = 1, \ldots, K,$$

where $K$ is an upper bound on the number of used bins ($K \leq n$). Constraints (1b) require that each item is assigned to a bin; constraints (1c) impose the capacities of the bins; and constraints (1d) formulate the conflicts. The objective (1a) is to minimize the number of used bins.

The linear programming relaxation of formulation (1) is weak, even if valid inequalities are added (Martello and Toth, 1990). Instead, as for a standard bin packing problem, one can use a set covering reformulation of (1) (Fernandes Muritiba et al., 2009; Elhedhli et al., 2010). Such reformulation results from applying the Dantzig-Wolfe decomposition principle to (1) (Vanderbeck and Savelsbergh, 2006): once (1b) are dualized in a Lagrangian way, subsystem (1c-1f) decomposes into a subproblem for each bin $k$. Let $B$ be the set of all the subsets of items which are not in conflict and fit into one bin, i.e., the solutions to a
subproblem. Each subset $B \in \mathcal{B}$ is defined by an indicator vector $x^B$ ($x^B_i = 1$ if item $i$ is in set $B$) and associated with a binary variable $\lambda_B$ taking value $1$ if the corresponding subset of items is selected to fill one bin. The reformulation is:

$$\min \sum_{B \in \mathcal{B}} \lambda_B$$

s.t. $\sum_{B \in \mathcal{B}} x^B_i \lambda_B \geq 1, \quad i = 1, \ldots, n,$

$\lambda_B \in \{0, 1\}, \quad B \in \mathcal{B}.$

Here, constraints (2b) replace constraints (1b), all other constraints of the compact formulation (1) are built in the definition of feasible sets $B \in \mathcal{B}$.

Formulation (2) is tackled using a branch-and-price approach: at each node of a branch-and-bound tree, the linear relaxation of (2) is solved by column generation to provide a lower bound. The calculation of this lower bound is done by iteratively solving:

- the restricted master problem (RMP) which is the linear relaxation of (2) with a restricted number of variables;

- and the pricing problem which determines whether there exists a variable $\lambda_B$ to be added to (RMP) in order to improve its current solution; this amounts to searching for the set $B \in \mathcal{B}$, solution to subsystem (1c-1f), which yields the minimum reduced cost column for (RMP).

Let $\{\pi_i\}_{i \in \mathcal{V}}$ be a current dual solution of the (RMP). Then, the pricing problem can be formulated as

$$\max \sum_{i=1}^{n} -\pi_i z_i$$

s.t. $\sum_{i=1}^{n} w_i z_i \leq W \quad i = 1, \ldots, n,$

$z_i + z_j \leq 1, \quad (i, j) \in E,$

$z_i \in \{0, 1\}, \quad i = 1, \ldots, n.$

Model (3) is a Knapsack Problem with Conflicts (KPC), as studied, for example, by Hifi and Michrafy (2007).
3. A branch-and-price algorithm

Our algorithm for solving the BPPC is a branch-and-price procedure. It is to be distinguished from the branch-and-price algorithms used by Fernandes Muritiba et al. (2009) and Elhedhli et al. (2010) for the following features:

1. the way we solve the pricing problem (denoted KPC);
2. the branching rule that we use;
3. the use of a generic column generation based heuristic.

Before going into details, we briefly present the “features” of the branch-and-price algorithms described in the literature.

Fernandes Muritiba et al. (2009) use combinatorial lower bounds and a tabu search based heuristic as a “preprocessing” step. The pricing problem is solved by a greedy heuristic. The MIP solver *CPLEX 10* is called when the heuristic cannot find a column with a negative reduced cost. Branching is performed by implementing a disjunctive constraint on the $\lambda$ variable with the largest fractional part, with priority to the round-up branch. A depth first strategy is used.

Elhedhli et al. (2010) rely on the branching rule of Ryan and Foster (1981) in which two items are selected and put to the same bin at the first child node and constrained to be in different bins at the other child node. The implementation strategy is however specific: instead a classic binary search tree, several pairs of items are considered simultaneously. In the first branch, they are all considered for joint assignment; while the other child nodes enumerate the disjoint assignments for each pair. Primal solutions are obtained using a rounding heuristic for the set covering problem (2). The pricing problem is solved by the MIP solver *CPLEX* after adding maximal clique inequalities generated using the Qualex library (Busygin, 2006).

3.1. Solving the Knapsack Problem with Conflicts

In selecting an algorithm to solve the KPC with interval and arbitrary conflict graphs, the first obvious choice is to use an IP solver for formulation (3). However, our preliminary tests showed that very efficient IP solver such as CPLEX are not fast enough to be called many times during the column generation procedure. An alternative algorithm for the KPC was
proposed by Hifi and Michrafy (2007). It is faster than CPLEX only on a small fraction of instances with conflict graph density of around 1%. Therefore, we developed our own specialized algorithms for the KPC.

First consider the Knapsack Problem with Interval Conflict Graphs (KPICG). Formally, a graph \( G = (V, E) \) is an interval graph if, to each vertex \( v \in V \), one can associate an open interval \( I_v \) of the real line, i.e., \( I_v = (a_v, b_v) \) for \( a_v, b_v \in \mathbb{R} \) with \( a_v < b_v \), such as two distinct vertices \( u, v \in V \) are adjacent if and only if \( I_u \cap I_v \neq \emptyset \). The family \( \{I_v\}_{v \in V} \) is an interval representation of \( G \). See Figure 1 for an illustration.

![Figure 1: An interval graph and its interval representation](image)

Recently, Pferschy and Schauer (2009) proposed a pseudo-polynomial algorithm for the KPC with chordal conflict graphs which is a super-class of interval graphs. However, the high complexity \( O(nW^2) \) of this algorithm prevents us of using it within the branch-and-price method. Therefore, here we propose a new algorithm with a lower complexity for solving the KPICG.

Consider an instance of the KPICG. Let \( \{I_i\}_{i \in V} \) be the interval representation of the conflict graph. Note that, if such a representation is not explicitly given, it can be obtained in time \( O(n + m) \) (Corneil et al., 1998), where \( n \) is the number of vertices and \( m \) is the number of conflicts.

**Definition 1.** Considering an interval conflict graph, \( G = (V, E) \), assume the items, \( i \in V \), are indexed in non-decreasing order of the right endpoints of the corresponding intervals \( \{I_i = (a_i, b_i)\}_{i \in V} \) (the ties are resolved arbitrarily), i.e., \( b_i \leq b_j \) if \( i < j \). Let \( Q_i \) denote the set of items with indexes smaller than \( i \) that are not in conflict with \( i \):

\[
Q_i = \{ j : j < i, (i, j) \notin E \}, \quad \forall i \in V.
\]

Let \( \text{prev}_i \) be the item in \( Q_i \) with the largest index (or 0 if such item does not exist):

\[
\text{prev}_i = \begin{cases} 
\max\{j : j \in Q_i\}, & Q_i \neq \emptyset, \\
0, & Q_i = \emptyset.
\end{cases} \quad \forall i \in V.
\]
**Observation 1.** Consider an interval conflict graph, \( G = (V, E) \). Given the item indexing of Definition 1, for every pair \( i, j \in V \), such that \( 1 \leq j \leq \text{prev}_i \), \( i \) and \( j \) are not in conflict, i.e., \((i, j) \notin E\), while when \( \text{prev}_i < j < i \), \( i \) and \( j \) are in conflict, i.e., \((i, j) \in E\).

Let \( P(i, w) \) be the value of an optimal solution of the KPICG for the first \( i \) items and knapsack size \( w \). With this notation, the solution of model (3) gives \( P(n, W) \). By Observation 1, the solution set associated with \( P(i, w) \) either includes item \( i \) and cannot include any items \( j \) such as \( \text{prev}_i < j < i \); or it does not include item \( i \) and it reproduces the solution set for \( P(i - 1, w) \). Therefore,

**Observation 2.**

\[
P(i, w) = \max \{P(\text{prev}_i, w - w_i) + p_i, P(i - 1, w)\}.
\]

The value \( P(n, W) \) is the solution to KPICG. The associated solution set \( B \) can be retrieved by backtracking from value \( P(n, W) \) to value \( P(0, 0) \). The dynamic programming algorithm stemming from Observation 2 is formally presented as Function \( \text{DP} \). It is easy to see that the time and the space complexity of the dynamic programming algorithm are both \( O(nW) \) once the values \( \text{prev} \) are known. This complexity is more tractable in practice than that of the algorithm by Pferschy and Schauer (2009).

Next, we consider the knapsack problem with an arbitrary conflict graph. We developed the following simple enumeration algorithm for the KPC. This algorithm is a combination of the classic depth-first-search based branch-and-bound algorithm for the 0-1 Knapsack Problem (Kelleler et al., 2004, section 2.4) and the enumeration algorithm for solving the maximum clique (or maximum independent set) problem by Carraghan and Pardalos (1990). The latter also makes use of a depth-first-search strategy, while dual bounds are obtained by simply ignoring all conflicts between free vertices, i.e. vertices which have not yet been fixed via branching decisions.

**Definition 2.** For each item \( i \in V \), we define the list \( C_i \) of items in conflict with \( i \). At any node of the enumeration tree, we denote by \( S^1 \) the set of items that have been selected in the current partial knapsack solution and by \( S^0 \) the items that have been fixed to 0. The set \( F = (V \setminus (\cup_{i \in S^1} C_i \cup S^1 \cup S^0)) \) denotes free items that are not fixed to either 0 or 1 in previous branching decisions and are not in conflict with items in \( S^1 \). Assume that items are indexed in the non-decreasing order of their “efficiency”, i.e., of their ratio \( p_i/w_i \). Then, \( \text{succ}_i(F) \)
Function \( \text{DP}(n, p[1,\ldots,n], w[1,\ldots,n], W, \text{prev}[1,\ldots,n]) \)

\[
\begin{align*}
\text{for } w & \leftarrow 0 \text{ to } W \text{ do} \\
& \quad P(0, w) \leftarrow 0; \\
\text{for } i & \leftarrow 1 \text{ to } n \text{ do} \\
& \quad \text{for } w \leftarrow 0 \text{ to } w_i - 1 \text{ do} \\
& \quad \quad P(i, w) \leftarrow P(i - 1, w); \\
& \quad \quad l_{iw} \leftarrow 0; \\
& \quad \text{for } w \leftarrow w_i \text{ to } W \text{ do} \\
& \quad \quad P(i, w) \leftarrow P(i - 1, w); \\
& \quad \quad l_{iw} \leftarrow 0; \\
& \quad \quad \text{if } P(i, w) < P(\text{prev}_i, w - w_i) + p_i \text{ then} \\
& \quad \quad \quad P(i, w) \leftarrow P(\text{prev}_i, w - w_i) + p_i; \\
& \quad \quad \quad l_{iw} \leftarrow 1; \\
\end{align*}
\]

\( w \leftarrow W; \)

\( B \leftarrow \emptyset; \)

\text{for } i \leftarrow n \text{ downto } 1 \text{ do}

\[
\begin{align*}
\text{if } l_{iw} & = 1 \text{ then} \\
& \quad B \leftarrow B \cup \{i\}; \\
& \quad w \leftarrow w - w_i; \\
\end{align*}
\]

\text{return } B;

denotes the item following \( i \) in the sorted sub-list of items of \( F \). By extension \( \text{succ}_0(F) \)
denote the first element in \( F \), while \( \text{last}(F) \) the last element in \( F \) and \( \text{succ}_{\text{last}(r)}(F) = n + 1 \).

During the depth-first-search, upper (dual) bounds \( UB \) are computed at each node of the tree by solving the continuous relaxation of the residual knapsack problem on set \( F \), ignoring conflict constraints:

\[
\begin{align*}
UB & = \max \sum_{i \in F} p_i x_i + \sum_{i \in S^1} p_i \\
\text{s.t. } & \quad \sum_{i \in F} w_i x_i \leq W - \sum_{i \in S^1} w_i \quad i \in F, \\
& \quad 0 \leq x_i \leq 1, \quad i \in F.
\end{align*}
\]

As the items in \( F \) are sorted according to their efficiencies, problem (5) can be solved in \( O(n) \) time using a greedy algorithm (Kelleler et al., 2004).

If the current upper bound \( UB \) is larger than the value \( LB \) of the incumbent solution, we branch: for each item \( i \in F \), we consider a child node where \( i \) is added to \( S^1 \) and all items of \( F \) that are before \( i \) are added to \( S^0 \). As the items in \( F \) and in the conflict list \( C_i \) of the \( i \)-th item in \( F \) are sorted in the same order, each child node can be created in time \( O(n) \). Thus,
the time spent per node is linear. The depth-first-search algorithm for the KPC is formally presented as Function $\text{Node}(p, w, S^1, F, LB, B)$, where $p$ is the current profit, $w$ the current weight, $S^1$ the set of items fixed to 1, $F$ the set of free items, $LB$ the current lower bound, and $B$ the associated current incumbent solution. The algorithm can be launched by calling $\text{Node}(0, 0, \emptyset, V, 0, \emptyset)$.

\begin{verbatim}
Function \text{Node}(p, w, S^1, F, LB, B)
    if $p > LB$ then
        $LB \leftarrow p$;
        $B \leftarrow S^1$;
        $UB \leftarrow p$;
        $c \leftarrow w$;
        $i \leftarrow \text{succ}_0(F)$;
        while $c < W$ and $i \leq \text{last}(F)$ do
            if $c + w_i \leq W$ then
                $UB \leftarrow UB + p_i$;
                $c \leftarrow c + w_i$;
                $i \leftarrow \text{succ}_i(F)$;
            else
                $UB \leftarrow UB + (W - c) \cdot (p_i/w_i)$;
                $c \leftarrow W$;
        if $UB \leq LB$ then
            return $B$;
        $i \leftarrow \text{succ}_0(F)$;
        while $p + (W - w) \cdot (p_i/w_i) > LB$ and $i \leq \text{last}(F)$ do
            $j \leftarrow \text{succ}_i(F)$;
            $F \leftarrow F \setminus \{i\}$;
            if $w + w_i \leq W$ then
                $\hat{S}^1 \leftarrow S^1 \cup \{i\}$;
                $\hat{F} \leftarrow F \setminus C_i$;
                $B \leftarrow \text{Node}(p + p_i, w + w_i, \hat{S}^1, \hat{F}, LB, B)$;
            $i \leftarrow j$;
        return $B$;
\end{verbatim}

We also experimented with variants of the above algorithm. In a first variant, items were added to the partial solution in the non-increasing order of the ratio of profit by number of conflicts. In a second variant, dual bounds were computed by solving a continuous knapsack problem that takes into account some of the conflicts only. Indeed, continuous continuous knapsack problems with disjoint special ordered set (SOS) constraints can be solved using the $O(n^2)$ algorithm by Johnson and Padberg (1981). Therefore, we search for
a conflict sub-graph that define a family of disjoint cliques, \( C \subset V \), each of which defines a SOS constraint: \( \sum_{i \in C} x_i \leq 1 \). The family of cliques is constructed using the following iterative greedy procedure: items are indexed in non-decreasing order of their ratio \( p_i/w_i \); on each iteration, one clique is formed and the corresponding vertices are not considered in subsequent iterations; to build a clique, we select the remaining item with smallest index (maximum ratio \( p_i/w_i \)) and add subsequent items that are in conflict with it as long as it forms a clique. The two above variants were not as successful experimentally as that of Function \textbf{Node}(p, w, S^1, F, LB, B).

Note that, in the column generation procedure, we do not have to solve the pricing problem to optimality: the procedure can iterate as long as a solution with a negative reduced cost is found. Fernandes Muritiba et al. (2009) applied a heuristic to try to find such a solution before relying on an exact algorithm. This approach decreases the average time needed to solve the pricing problem. However, our computational study showed that the overall branch-and-price algorithm performance is not as good when using a non exact pricing problem solver because of the slower convergence of the column generation procedure.

3.2. The branching rule

The linear relaxation of (2) is traditionally called the master. It can be shown that the solution \( \bar{\lambda} \) to the master is binary, i.e. \( \bar{\lambda} \in \{0, 1\}^B \), if and only if for all item pairs \( i, j \in V \), the number of selected subsets that contain both \( i \) and \( j \) is either zero or one. Equivalently \( \bar{\lambda} \in \{0, 1\}^B \) if and only if solution \( \bar{\lambda} \) can be projected on a corresponding solution \((\bar{x}, \bar{y})\) to the integer problem (1) – the projection is defined in Vanderbeck (2010). The result is known for the bin packing problem, or vertex coloring, but it applies more generally for a set partitioning like master problem with a pure binary subproblem (Vanderbeck, 2010; Vanderbeck and Wolsey, 2010).

Therefore, a natural branching scheme is the following. At a given node of the branch-and-price tree, given master solution \( \bar{\lambda} \), one can identify two items \( i, j \in V \) such that \( 0 < \sum_{B: i, j \in B} \bar{\lambda}_B < 1 \), and branch by enforcing that these two items are either assigned to the same bin or must be in different bins. Such branching constraint can be enforced by removing inappropriate columns and constraining further column generation by adding a constraint in the subproblem. In the first child node, where item \( i \) and \( j \) must be assigned to the same bin, one adds constraints \( x_i = x_j \) to the pricing problem (3) and removes from the master all columns that do not satisfy it. In the second child, constraint \( x_i + x_j \leq 1 \) is added to
the subproblem and columns with \( x_i = x_j = 1 \) are removed from the master. This scheme was originally proposed by Ryan and Foster (1981). It was used in the approach of Elhedhli et al. (2010).

Observe that adding branching constraints to (3) implies modifications to the pricing problem that are compatible with the KPC model: constraint \( x_i = x_j \) amounts to contracting the associated vertices of the conflict graph into one vertex representing the item pair \( i, j \); while \( x_i + x_j \leq 1 \) amounts to adding a conflict edge. However, the special structure of the conflict graph might be lost. In particular, for applications where the initial conflict graph is an interval graph, additional conflicts introduced by branching will typically destroy this structure that made the pricing problem solvable in pseudo-polynomial time as we showed in Section 3.1. Thus, our dynamic programming algorithm on which the method rely for good performance cannot be used when the Ryan and Foster branching scheme is applied.

Instead, we use the generic branching scheme proposed by Vanderbeck (2010) that was specially designed to preserve the structure of the pricing problem. The scheme proceeds by progressively partitioning into column classes the set \( \mathcal{B} \) of feasible pricing problem solutions and by implementing separate pricing on each class. A class is defined by restricting the solution set via fixing some variables to zero or one. Hence, pricing over a class can be done using the initial oracle since the latter can handle some variable fixing. The implementation developed in Vanderbeck (2010) guarantees that the number of created classes remains polynomial in the input size. Fractional master solutions are eliminated by adding branching constraint in the master that force an integer lower bound on the number of columns selected in each defined class. The dual bounds after branching are proved to be as strong as if branching constraint had been defined in the subproblem.

To be more specific let us examine how the generic branching scheme of Vanderbeck (2010) applies to the present problem. It takes a form closely related to the Ryan and Foster branching scheme. At the root node, a pair \( i, j \in V \) is selected such that \( 0 < \sum_{B:B \ni i,j} \lambda_B < 1 \). Then, in Node 1, branching is implemented by requiring that \( \sum_{B:B \ni i,j} \lambda_B \geq \lambda_B \) and \( \sum_{B:B \ni i,j} \lambda_B \geq K - 1 \) in the master; there are two column classes and two associated pricing problems, the first consider solutions where \( x_i = x_j = 1 \) and the second solutions where \( x_i = x_j = 0 \). In Node 2, branching is implemented by requiring \( \sum_{B:B \ni i,j} \lambda_B \geq 1 \) and \( \sum_{B:B \ni i,j} \lambda_B \geq K - 1 \) in the master and there are 2 pricing problems, the first consider solutions where \( x_i = 1 \) and \( x_j = 0 \) and the second solutions where \( x_i = 0 \). At subsequent branch-and-price nodes, branching is implemented by further partitioning existing column
classes (for details see Vanderbeck (2010)).

### 3.3. Column generation based heuristic

As it was showed by previous research and by our own computational experiments, the set covering formulation is a very tight formulation that provides quite good dual bounds for the BPPC. Therefore, combined with a good primal heuristic, the column generation approach can be a very successful algorithm.

We use a generic diving heuristic which is a depth-first heuristic search in the branch-and-price tree that is presented in Joncour et al. (2010). Here, the branch-and-price enumeration is not driven by the branching scheme of Section 3.2, but simply by fixing $\lambda_B$ variables. At each branch-and-price node, the master is solved by column generation, then a branch corresponding to rounding-to-one a $\lambda_B$ variable is selected heuristically based on a greedy strategy. The master is then updated: deleting rows of (2) associated to items already covered and deleting columns covering those items. The master is re-optimized with a limit on the number of column generation iterations and the process is reiterated.

The solution obtained through the initial depth first exploration of the tree is considered as a reference incumbent solution. To further explore the solution space, we use limited backtracking as a diversification mechanism as developed in Joncour et al. (2010). This generic primal heuristic implemented in the software platform BaPCod relies on the concept of Limited Discrepancy Search (Harvey and Ginsberg, 1995). Specifically, we avoid choosing columns in a tabu list that consists of columns selected in previous branches from which we wish to diversify the search. The tabu list of columns at a branch-and-price node is the union of the tabu list of its ancestor and the columns chosen in previous child nodes of the ancestor. The tabu list of the root node is empty. A node which is not the first child node of its ancestor is explored only if the size of its tabu list is smaller or equal to $\text{maxDiscrepancy}$ and its depth is smaller or equal to $\text{maxDepth}$, where $\text{maxDiscrepancy}$ and $\text{maxDepth}$ are two control parameters. In our implementation, we set parameters $\text{maxDiscrepancy} = 2$ and $\text{maxDepth} = 3$. The resulting search tree is illustrated in Figure 2.

### 4. Computational experiments

Our algorithm was developed using the software platform BaPCod — a generic Branch-and-Price Code. BaPCod is a library of C++ classes developed within the INRIA research team.
Figure 2: The search tree of the diving heuristic with parameters \texttt{maxDepth} = 3, \texttt{maxDiscrepancy} = 2; a dotted line denotes a pure dive down in the branch-and-price tree.

RealOpt at the University of Bordeaux. Our algorithm relies on the generic features of the solver for the branching scheme, the primal heuristic, and basic preprocessing. Therefore, the only application specific implementation consists in providing the problem formulation and the oracles for solving the pricing problem.

4.1. Instances with interval conflict graphs

We tested our procedure on instances obtained using the generation procedure of Gendreau et al. (2004), which is itself based on the bin packing test instances of Falkenauer (1996). There are 8 classes of instances. In the first four classes referenced below by “u”, the items have an integer weight uniformly distributed in the range [20, 100] while bins have capacity 150. The number \( n \) of items is the same for each instance in a class, and grows from the first to the fourth class, taking value: \( n = 120, 250, 500 \) and 1000, respectively. The next four classes referenced below by “t” (for “triplets” ) involve items with weights, \( w_i \) uniformly distributed in the range [250, 500], to be packed into bins of capacity \( W = 1000 \). Items are generated by triplet: every third item, \( i_{3s} \), has a weight \( w_{i_{3s}} = W - w_{i_{3s-1}} - w_{i_{3s-2}} \) for \( s = 1, \ldots, \frac{n}{3} \). Thus, an optimal solution requires \( \frac{n}{3} \) bins that are filled at full capacity with exactly three items. The number of items is, respectively, \( n = 60, 120, 249 \) and 501.

Conflict graphs are characterized by different density values \( \delta \), varying from 0.1 to 0.9. This is done by assigning a value, \( \rho_i \), to each vertex, \( i \in V \), according to a continuous uniform distribution in [0, 1]. Then, a conflict is created for item pair \((i, j)\) if \((\rho_i + \rho_j)/2 \leq \delta\). For each class and each \( \delta \), 10 instances were generated. Thus, the resulting test-bed is composed of
800 instances in total. In the sequel, we assume that items are indexed in the non-increasing order of their values \( \rho_i \).

Observe that the above conflict generation scheme results in an interval conflict graph. The interval representation of such graph is similar to the one depicted in Figure 3. Therein, the intervals \( I_i \) associated to each item/vertex \( i \in V \) are shown from the bottom to the top in order of their index number, i.e., in the non-increasing order of their values \( \rho_i \). The definition of intervals \( I_i \) formalizes the condition that a conflict exists if \( (\rho_i + \rho_j)/2 \leq \delta \). If two items, \( i \) and \( j \) have both \( \rho \)-value larger or equal to \( \delta \) they cannot be in conflict; hence, they have non-overlapping intervals. Let \( L = \{ i : \rho_i > \delta \} \) and define interval \( I_i = (i - 1, i) \) for items \( i \in L \). If two items, \( i \) and \( j \) have both \( \rho \)-value smaller or equal to \( \delta \) they must be in conflict; hence, they have overlapping intervals. For items \( j \in V \setminus L \), one can set \( I_j = (a_j, |L| + 1) \) where \( a_j = \min\{i - 1 : i \in L \text{ and } \rho_i + \rho_j \leq 2\delta \} \). Note that all these intervals overlap on \((|L|, |L| + 1)\). Now consider two items \( i, j \in V \) with \( \rho_i > \delta \), \( \rho_j \leq \delta \), and therefore \( i < j \) and note that their interval overlap only if \( \rho_i + \rho_j \leq 2\delta \), with the property that if \( I_i \) overlaps with \( I_j \), then it must overlap with every \( I_l \) such as \( j < l \).

![Figure 3: Structure of the interval representation of the conflict graphs](image)

We have therefore showed that an interval definition exists that yields the desired conflict graph. However, for the purpose of our algorithm, one only needs the values of \( \text{prev}_i \) of Definition 1. They can be obtained as follows:

\[
\text{prev}_i = \begin{cases} 
  i - 1, & \rho_i > \delta, \\
  \max\{j : (\rho_j + \rho_i)/2 > \delta\}, & \rho_i \leq \delta
\end{cases}
\]

assuming items are indexed in the non-increasing order of their values \( \rho_i \).

In our numerical experiments, we first compared our algorithm and the algorithm of Fernandes Muritiba et al. (2009), which we denote MIMT. The comparison was carried on the test instances that were kindly provided to us by these authors. For each class and each density, there are 10 instances. So, we have 90 instances for each class. Here we tested two versions of our algorithm: with and without the diving heuristic. Our algorithm was run
using one thread on a Dell PowerEdge T300 workstation with an Intel Xeon X5460 3.16 GHz processor. Algorithm MIMT was run on a Pentium IV 3 GHz processor. By www.spec.org, our machine is roughly 3.75 times faster. Therefore, we multiplied our computing time by 3.75 for the purpose of this comparison.

<table>
<thead>
<tr>
<th>class</th>
<th>MIMT</th>
<th>Our w/o heur.</th>
<th>Our with heur.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>not opt.</td>
<td>av. time</td>
<td>not opt.</td>
</tr>
<tr>
<td>t60</td>
<td>0</td>
<td>38.7</td>
<td>0</td>
</tr>
<tr>
<td>t120</td>
<td>5</td>
<td>1860.3</td>
<td>1</td>
</tr>
<tr>
<td>t249</td>
<td>4</td>
<td>1582.1</td>
<td>2</td>
</tr>
<tr>
<td>t501</td>
<td>4</td>
<td>3163.6</td>
<td>0</td>
</tr>
<tr>
<td>u120</td>
<td>0</td>
<td>29.4</td>
<td>0</td>
</tr>
<tr>
<td>u250</td>
<td>0</td>
<td>107.1</td>
<td>0</td>
</tr>
<tr>
<td>u500</td>
<td>5</td>
<td>2195.4</td>
<td>1</td>
</tr>
<tr>
<td>u1000</td>
<td>2</td>
<td>1911.9</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Comparison of our algorithm with the algorithm of Fernandes Muritiba et al. (2009)

In Table 1, we report the number of test instances unsolved within the time limit, the average solution time, and the maximum solution time (for our algorithm). The time limit was 10 hours for MIMT and 1 hour for our algorithm. Our algorithm with the diving heuristic solved all instances to optimality and it is faster by an order of magnitude than MIMT. Using the heuristic allowed us to solve 4 more instances and it speeds up our algorithm considerably. Additionally, we observed that our root node lower bound was equal to the optimal solution for all instances but 3. All but 4 instances were solved at the root node (thanks to the diving heuristic).

Secondly, we compare our algorithm and the algorithm of Elhedhli et al. (2010), which we denote ELGN. The comparison was carried on the test instances that were kindly provided to us by these authors. For each class and each density, there are 20 instances. So, we have 180 instances for each class. The comparison involve results obtained on different computers: algorithm ELGN was run on a Sun Blade 2500 workstation with an Ultrasparc IIIi 1.6 GHz processor. By www.spec.org, our machine is roughly six times faster. Therefore, in this comparison, the solution time of our algorithm is multiplied by 6, and the ELGN time limit was divided by 6. The time limit was 1 hour for ELGN and 10 minutes for our algorithm.

In Table 2, we compare the number of test instances unsolved within the time limit (for algorithm ELGN, since our algorithm solved all instances to optimality), the average solution time, and the maximum solution time. Here we compare algorithm ELGN with our
algorithm including the diving heuristic (algorithm ELGN includes a primal heuristic too). Results of Table 2 indicates that our algorithm is an order of magnitude faster. All instances were solved at the root node thanks to the diving heuristic. The root node lower bound is equal to the optimal solution for all tested instances.

### 4.2. Instances with arbitrary conflict graphs

As there are no test instances of the problem with an arbitrary conflict graph available in the literature, we generated them ourselves. First, we took the same instances as those used above, but generated the conflict graphs randomly in the following way. We begin with the empty graph. We iteratively select an item pair \((i, j)\) at random (with uniform distribution); then edge \((i, j)\) is added to the graph if it is not already defined. The procedure is interrupted as soon as the desired graph density is reached.

The resulted eight classes are referenced below by “ta” and “ua”. For each class and each density, there are 10 instances. So, we have 90 instances per class. We tested our algorithm with and without the diving heuristic on these instances. In Table 3, we compare the number of test instances unsolved within 1 hour, the average solution time (only for the solved instances), and the average remaining gap (only for the unsolved instances). When the heuristic is not used, then generally, if an instance is not solved to optimality, no feasible solution is available. Therefore, for this version of the algorithm, the gap statistics are not provided.

For comparison purposes, Table 3 also reproduces on the right-hand the results for instance classes “u” and “t” with interval conflict graph (therein our computing times are not multiplied by a correction factor anymore). As shown in Table 3, instances with arbitrary conflict graphs are significantly harder to solve than the instances with interval conflict graphs. For arbitrary conflict graphs, the diving heuristic helps a lot when solving instances
Table 3: Results obtained by our algorithm on instances with arbitrary conflict graphs and comparison of solution time on instances with interval conflict graphs

<table>
<thead>
<tr>
<th>class</th>
<th>Our algo w/o heur.</th>
<th>Our algo with heur.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>not opt. av. time</td>
<td>not opt. av. time gap</td>
</tr>
<tr>
<td>ta60</td>
<td>0 0.3</td>
<td>0 0.2 0%</td>
</tr>
<tr>
<td>ta120</td>
<td>0 2.3</td>
<td>0 4.2 0%</td>
</tr>
<tr>
<td>ta249</td>
<td>7 97.6</td>
<td>6 137.3 1.2%</td>
</tr>
<tr>
<td>ta501</td>
<td>23 215.0</td>
<td>25 392.6 0.6%</td>
</tr>
<tr>
<td>ua120</td>
<td>0 1.4</td>
<td>0 0.7 0%</td>
</tr>
<tr>
<td>ua250</td>
<td>1 41.2</td>
<td>2 9.0 1.0%</td>
</tr>
<tr>
<td>ua500</td>
<td>27 234.3</td>
<td>8 39.0 0.5%</td>
</tr>
<tr>
<td>ua1000</td>
<td>33 713.0</td>
<td>8 286.2 0.3%</td>
</tr>
<tr>
<td></td>
<td>t60 0.2</td>
<td>t120 4.1</td>
</tr>
<tr>
<td></td>
<td>t249 8.6</td>
<td>t501 50.4</td>
</tr>
</tbody>
</table>

in class “ua”. On the contrary, it increases the running time when solving class “ta” instances. However, it guarantees to obtain a good feasible solution. Solving pricing problems using the depth-first-search branch-and-bound algorithm takes 32.9% of the overall running time on the average.

4.3. Instances with a larger number of items per bin

Observe that, in previous classes of instances, the number of items per bin does not exceed 3 on average. We generated additional classes of more difficult instances denote below by “d” and “da”. They consist of items with integer weights uniformly distributed in the range [500, 2500], to be packed into bins of capacity $W = 10000$. In class “d”, conflict graphs are interval graphs. There were generated using the same procedure as for classes “t” and “u”. In class “da”, conflict graphs are arbitrary, as in classes “ta” and “ua”. The number $n$ of items is 120, 250, and 500. There are on average 8 items per bin. For each class and each density, there are 10 instances. So, we have 90 instances per class again. We tested our algorithm with the diving heuristic on these instances. In Table 4, we report the number of test instances unsolved within 1 hour (out of 90), the average solution time (only for the solved instances), and the average remaining gap (only for the unsolved instances).

Our algorithm solved all class “d” instances. As for class “a” and “t”, the lower bound provided by the column generation procedure are very tight: it was always equal to the optimum solution. The diving heuristic is very good: it found optimum solutions for all but 3 instances. Our numerical experiment revealed that, for these instances where the bin capacity is large, the dynamic programming algorithm was slower than the depth-first-search
branch-and-bound algorithm developed the general KPC. Hence, we used the latter. The solution time for instances “d” are therefore much larger than for instances “u” and “t”. The increase computing time is also due to a slower convergence of the column generation algorithm. In our experiments the oracle for the KPC took on the average only 18.6% of the overall computation time. (The increasing computing times explain why we did not test instances in class “d” and “da” with 1000 jobs.)

The performance of our algorithm for class “da” instances is much worse than for other tested classes. Slightly less than half of the instances remains unsolved after one hour of computation time. The remaining gap is almost an order of magnitude larger than for class “ta” and class “ua” instances. The difficulty of these instances depends a lot on the conflict graph density. Details on this can be found in Section 4.5.

### 4.4. Efficiency of the diving heuristic

In this subsection, we present the results obtained using the diving heuristic only, without branching. We tested two variants of the heuristic: a pure diving approach (DH) without the Limited Discrepancy Search (meaning that the \texttt{maxDiscrepancy} parameter is equal to 0) and the variant used in the above test (DH with LDS) with the parameters \texttt{maxDiscrepancy} = 2 and \texttt{maxDepth} = 3.

<table>
<thead>
<tr>
<th>class</th>
<th>not opt.</th>
<th>av. time</th>
<th>gap</th>
<th>class</th>
<th>not opt.</th>
<th>av. time</th>
<th>gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>d120</td>
<td>0</td>
<td>8.9</td>
<td>0.0%</td>
<td>da120</td>
<td>23</td>
<td>23.2</td>
<td>4.7%</td>
</tr>
<tr>
<td>d250</td>
<td>0</td>
<td>53.4</td>
<td>0.0%</td>
<td>da250</td>
<td>40</td>
<td>23.3</td>
<td>3.7%</td>
</tr>
<tr>
<td>d500</td>
<td>0</td>
<td>486.8</td>
<td>0.0%</td>
<td>da500</td>
<td>41</td>
<td>137.6</td>
<td>3.9%</td>
</tr>
</tbody>
</table>

Table 4: Results obtained by our algorithm on hard instances with interval and arbitrary conflict graphs

<table>
<thead>
<tr>
<th>#items</th>
<th>“u”</th>
<th>“t”</th>
<th>“ua”</th>
<th>“ta”</th>
<th>“da”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>gap</td>
<td>time</td>
<td>gap</td>
<td>time</td>
</tr>
<tr>
<td>60</td>
<td>0.2</td>
<td>0.21%</td>
<td>0.2</td>
<td>0.21%</td>
<td>0.1</td>
</tr>
<tr>
<td>120</td>
<td>0.6</td>
<td>0.12%</td>
<td>0.9</td>
<td>0.45%</td>
<td>0.6</td>
</tr>
<tr>
<td>250</td>
<td>3.1</td>
<td>0.11%</td>
<td>5.7</td>
<td>0.31%</td>
<td>3.7</td>
</tr>
<tr>
<td>500</td>
<td>17.8</td>
<td>0.04%</td>
<td>34.7</td>
<td>0.16%</td>
<td>28.1</td>
</tr>
<tr>
<td>1000</td>
<td>110.0</td>
<td>0.02%</td>
<td>205.1</td>
<td>0.08%</td>
<td>205.1</td>
</tr>
</tbody>
</table>

Table 5: Results with our diving heuristic DH alone.
In Tables 5 and 6, we present, for all the above instance classes, the average running time of the heuristics and the average gap of the solutions found in per cent from the best known dual bound. The diving heuristic produces solutions of a high quality. The Limited Discrepancy Search improves significantly the performance of the heuristic, especially for instances of classes “u”, “t”, and “ua”. A disadvantage of the diving heuristic is that its running time increases rapidly with the number of items. Again, class “da” instances are the most difficult. The heuristic efficiency drops a lot for this class.

Fernandes Muritiba et al. (2009) have proposed a population based heuristic (PH) for the problem. It consists in a tabu search algorithm and a diversification procedure. In Table 7, we compare the two variants of our diving heuristic with PH, the heuristic of Fernandes Muritiba et al. (2009). As it was done above, the running time of our heuristics is multiplied by 3.75 to compensate the difference in the computers speed. The results of Table 7 show that DH is faster than PH for instances with less than 500 items and produce on average significantly better solutions that the population heuristic (except for class “u120”). DH LDS is only slightly slower than the PH and produces optimal solutions for all instances except one.

### 4.5. Impact of the density of the graph to the results

We proceed to show how the difficulty of instances depends on the density of the conflict graph. Instances here are grouped according to their classes. In Table 8, for our algorithm with the diving heuristic, we present the percentage of test instances unsolved within 1 hour (columns “¬opt.”), the average solution time (only for the solved instances), and the average remaining gap (only for the unsolved instances). For classes “u” and “t”, we present only the time statistic, as all these instances were solved to optimality.

One can observe that the impact of the graph density on the difficulty of instances highly
Table 7: Comparison of our diving heuristics with the population heuristic of Fernandes Muritiba et al. (2009).

<table>
<thead>
<tr>
<th>density</th>
<th>“u”</th>
<th>“t”</th>
<th>“ua”</th>
<th>“ta”</th>
<th>“da”</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>15.5</td>
<td>6.1</td>
<td>0%</td>
<td>71.7</td>
<td>0%</td>
</tr>
<tr>
<td>20%</td>
<td>16.7</td>
<td>11.0</td>
<td>0%</td>
<td>67.7</td>
<td>0%</td>
</tr>
<tr>
<td>30%</td>
<td>19.5</td>
<td>21.8</td>
<td>0%</td>
<td>67.8</td>
<td>0%</td>
</tr>
<tr>
<td>40%</td>
<td>30.1</td>
<td>10.8</td>
<td>0%</td>
<td>71.4</td>
<td>0%</td>
</tr>
<tr>
<td>50%</td>
<td>26.8</td>
<td>9.8</td>
<td>0%</td>
<td>64.8</td>
<td>0%</td>
</tr>
<tr>
<td>60%</td>
<td>19.6</td>
<td>7.5</td>
<td>0%</td>
<td>76.5</td>
<td>0%</td>
</tr>
<tr>
<td>70%</td>
<td>15.8</td>
<td>5.7</td>
<td>5%</td>
<td>101.7</td>
<td>0.7%</td>
</tr>
<tr>
<td>80%</td>
<td>13.1</td>
<td>4.5</td>
<td>15%</td>
<td>100.2</td>
<td>0.5%</td>
</tr>
<tr>
<td>90%</td>
<td>10.5</td>
<td>3.0</td>
<td>25%</td>
<td>115.9</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

Table 8: Impact of the conflict graph density on the results obtained by our branch-and-price algorithm.

<table>
<thead>
<tr>
<th>density</th>
<th>“u”</th>
<th>“t”</th>
<th>“ua”</th>
<th>“ta”</th>
<th>“da”</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>15.5</td>
<td>6.1</td>
<td>0%</td>
<td>71.1</td>
<td>0%</td>
</tr>
<tr>
<td>20%</td>
<td>16.7</td>
<td>11.0</td>
<td>0%</td>
<td>67.7</td>
<td>0%</td>
</tr>
<tr>
<td>30%</td>
<td>19.5</td>
<td>21.8</td>
<td>0%</td>
<td>67.8</td>
<td>0%</td>
</tr>
<tr>
<td>40%</td>
<td>30.1</td>
<td>10.8</td>
<td>0%</td>
<td>71.1</td>
<td>0%</td>
</tr>
<tr>
<td>50%</td>
<td>26.8</td>
<td>9.8</td>
<td>0%</td>
<td>64.8</td>
<td>0%</td>
</tr>
<tr>
<td>60%</td>
<td>19.6</td>
<td>7.5</td>
<td>0%</td>
<td>76.5</td>
<td>0%</td>
</tr>
<tr>
<td>70%</td>
<td>15.8</td>
<td>5.7</td>
<td>5%</td>
<td>101.7</td>
<td>0.7%</td>
</tr>
<tr>
<td>80%</td>
<td>13.1</td>
<td>4.5</td>
<td>15%</td>
<td>100.2</td>
<td>0.5%</td>
</tr>
<tr>
<td>90%</td>
<td>10.5</td>
<td>3.0</td>
<td>25%</td>
<td>115.9</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

Table 7: Comparison of our diving heuristics with the population heuristic of Fernandes Muritiba et al. (2009).

Table 8: Impact of the conflict graph density on the results obtained by our branch-and-price algorithm.

depends on the graph class. The most difficult instances with interval conflict graph are with an average and slightly below average density. On the contrary, the most difficult instances with arbitrary conflict graph are with a high density. Highly oscillating results for instance class “ta” can be explained by their particular structure: there are threshold values for the density of the conflict graph which change a lot the quality of the column generation dual bounds.

In Tables 9 and 10, we present the impact of conflict graph density on the performance of the diving heuristics DH and DH LDS: we report the average running time and the average gap of the solutions found. The observation is similar to the one concerning the impact of density on the efficiency of the branch-and-price algorithm. The heuristic DH sometimes requires more time and produces solutions with larger relative gap for instances with small density. This is due to the larger running time of pricing oracle and the fact that the absolute solution values are smaller.
These results also allow us to give the following recommendations. For the instances of classes “ua” and “da”, the pure diving heuristic should be used if the graph density is large. For other instances with an arbitrary conflict graph, the DH LDS requires a reasonable increase in the running time but produces significantly better solutions. For instances with an interval conflict graph, the DH LDS should always be used.

5. Conclusions

In this paper, we present a branch-and-price algorithm for the bin packing problem with conflicts that we implemented using the software platform BaPCod. The only problem specific “features” of this algorithm are the formulations and the oracles for solving the pricing problem. Our algorithm was tested on instances from the literature and newly generated ones. Our computational results can be summarized as follows:

<table>
<thead>
<tr>
<th>density</th>
<th>“u”</th>
<th>“t”</th>
<th>“ua”</th>
<th>“ta”</th>
<th>“da”</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>14.3</td>
<td>3.2</td>
<td>70.0</td>
<td>17.6</td>
<td>32.7</td>
</tr>
<tr>
<td>20%</td>
<td>15.9</td>
<td>3.8</td>
<td>67.6</td>
<td>16.2</td>
<td>31.1</td>
</tr>
<tr>
<td>30%</td>
<td>19.2</td>
<td>5.3</td>
<td>67.1</td>
<td>12.7</td>
<td>27.6</td>
</tr>
<tr>
<td>40%</td>
<td>28.1</td>
<td>9.3</td>
<td>64.7</td>
<td>11.1</td>
<td>24.0</td>
</tr>
<tr>
<td>50%</td>
<td>24.7</td>
<td>9.1</td>
<td>63.6</td>
<td>9.2</td>
<td>21.3</td>
</tr>
<tr>
<td>60%</td>
<td>18.3</td>
<td>6.8</td>
<td>59.2</td>
<td>8.5</td>
<td>25.6</td>
</tr>
<tr>
<td>70%</td>
<td>14.5</td>
<td>5.2</td>
<td>54.8</td>
<td>7.1</td>
<td>45.9</td>
</tr>
<tr>
<td>80%</td>
<td>12.7</td>
<td>4.1</td>
<td>47.8</td>
<td>8.1</td>
<td>26.5</td>
</tr>
<tr>
<td>90%</td>
<td>9.4</td>
<td>2.8</td>
<td>39.4</td>
<td>6.5</td>
<td>12.4</td>
</tr>
</tbody>
</table>

Table 9: Impact of the conflict graph density on the results obtained using heuristic DH.

<table>
<thead>
<tr>
<th>density</th>
<th>“u”</th>
<th>“t”</th>
<th>“ua”</th>
<th>“ta”</th>
<th>“da”</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>15.5</td>
<td>6.1</td>
<td>74.7</td>
<td>51.2</td>
<td>92.5</td>
</tr>
<tr>
<td>20%</td>
<td>16.7</td>
<td>11.0</td>
<td>74.4</td>
<td>42.8</td>
<td>84.5</td>
</tr>
<tr>
<td>30%</td>
<td>19.5</td>
<td>14.0</td>
<td>69.1</td>
<td>72.3</td>
<td>36.7</td>
</tr>
<tr>
<td>40%</td>
<td>30.1</td>
<td>10.8</td>
<td>69.7</td>
<td>112.8</td>
<td>79.8</td>
</tr>
<tr>
<td>50%</td>
<td>26.8</td>
<td>9.8</td>
<td>66.7</td>
<td>120.8</td>
<td>81.0</td>
</tr>
<tr>
<td>60%</td>
<td>19.6</td>
<td>7.5</td>
<td>81.5</td>
<td>80.1</td>
<td>384.4</td>
</tr>
<tr>
<td>70%</td>
<td>15.8</td>
<td>5.7</td>
<td>106.9</td>
<td>86.3</td>
<td>543.4</td>
</tr>
<tr>
<td>80%</td>
<td>13.1</td>
<td>4.5</td>
<td>171.8</td>
<td>9.4</td>
<td>348.2</td>
</tr>
<tr>
<td>90%</td>
<td>10.5</td>
<td>3.0</td>
<td>605.5</td>
<td>91.7</td>
<td>170.7</td>
</tr>
</tbody>
</table>

Table 10: Impact of the conflict graph density on the results obtained using heuristic DH LDS.
On instances from the literature, our algorithm outperforms the existing algorithms. These instances are rather specific: the number of items in a bin is small (3 on average) and the conflict graph is an interval graph.

Instances where the solution involves a higher number of items per bin and the conflict graph has no special structure, are much harder. In particular, the linear relaxation bound stemming from the set covering formulation is not as tight when the conflict graph is not an interval graph.

The generic diving heuristic built into BaPCod contributes a lot to the success of our algorithm, and compares favorably with the population based heuristics from the literature. The Limited Discrepancy Search improves significantly the efficiency of the diving heuristic.

The generic BaPCod solver is a competitive tool once a problem specific oracle is provided for solving the pricing problem.

In addition to the new benchmarks, the highlights of our study are:

- a novel dynamic programming algorithm of complexity $O(nW)$ for the knapsack problem with an interval conflict graph;
- a depth-first-search branch-and-bound algorithm for the Knapsack Problem with Conflicts that has proved to be quite efficient in practice and outperforms the CPLEX 11.0 solver on instances with conflict graphs of density 10% and more;
- an illustration of the interest of exploiting structure of solved instances (the fact that standard BPPC test instances of the literature had interval conflict graphs was not noticed in the previous research work).

Note that, although the fact that the conflict graphs are interval was not exploited explicitly in both the study of Fernandes Muritiba et al. (2009) and that of Elhedhli et al. (2010), the graphs’ special structure still favors their proposed algorithms. One particularity of the conflict graphs generated as described in Section 4.1 is that they always contain large cliques. The size of the maximum clique and the number of maximal cliques are roughly equal to $n\delta$. In algorithm MIMT, combinatorial lower bounds are used which are based on the computation of a maximal clique. In the algorithm ELGN, maximal clique inequalities are
generated for formulation (3) in order to speed up its solution by the MIP solver (CPLEX).
Moreover, as our results show, instances with arbitrary conflict graphs are much harder to
solve to optimality than instances with the interval conflict graphs in great part because
the quality of the lower bounds produced by the column generation procedure is worse.
Therefore, one could say that algorithms ELGN and MIMT implicitly “exploited” the good
quality of lower bounds of the instances on which they have been tested and hence the
interval graph structure.

References


Communication Complexity, Lecture Notes in Computer Science*, vol. 5058. Springer
Berlin / Heidelberg, 61–73.

Busygin, Stanislav. 2006. A new trust region technique for the maximum weight clique

Carraghan, Randy, Panos M. Pardalos. 1990. An exact algorithm for the maximum clique

A. Mingozzi, P. Toth, C. Sandi, eds., *Combinatorial optimization*. Wiley, Chichester, 315–
338.

Corneil, Derek G., Stephan Olariu, Lorna Stewart. 1998. The ultimate interval graph recog-
Discrete algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA,
USA, 175–180.

Elhedhli, Samir, Lingzi Li, Mariem Gzara, Joe Naoum-Sawaya. 2010. A Branch-and-Price
Algorithm for the Bin Packing Problem with Conflicts. *INFORMS Journal on Computing*
ijoc.1100.0406.


