Discrete Applied Mathematics 158 (2010) 97–109

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

A multi-level search strategy for the 0–1 Multidimensional Knapsack Problem

Sylvain Boussier [a,*], Michel Vasquez [b], Yannick Vimont [b], Saïd Hanafi [c], Philippe Michelon [a]

[a] LIA, Université d'Avignon et des Pays de Vaucluse, 339 chemin des Meinajaries, BP 1228, 84911 Avignon Cedex 9, France

[b] LGI2P, École des Mines d'Alès, Parc scientifique Geroges Besse, 30035 Nîmes, France

[c] LAMIH, Université de Valenciennes et du Hainaut-Cambrésis Le Mont Houy - BP 311, 59304 Valenciennes, France

ARTICLE INFO

Article history:
Received 30 May 2008
Received in revised form 15 July 2009
Accepted 27 August 2009
Available online 24 September 2009

Keywords:
Resolution Search
Branch & Bound
Reduced costs
0–1 Multidimensional Knapsack Problem

ABSTRACT

We propose an exact method based on a multi-level search strategy for solving the 0–1 Multidimensional Knapsack Problem. Our search strategy is primarily based on the reduced costs of the non-basic variables of the LP-relaxation solution. Considering that the variables are sorted in decreasing order of their absolute reduced cost value, the top level branches of the search tree are enumerated following Resolution Search strategy, the middle level branches are enumerated following Branch & Bound strategy and the lower level branches are enumerated according to a simple Depth First Search enumeration strategy. Experimentally, this cooperative scheme is able to solve optimally large-scale strongly correlated 0–1 Multidimensional Knapsack Problem instances. The optimal values of all the 10 constraint, 500 variable instances and some of the 30 constraint, 250 variable instances of the `OR-Library` were found. These values were previously unknown.

1. Introduction

The 0–1 Multidimensional Knapsack Problem, denoted MKP, is a well-known optimization problem which can be viewed as a resource allocation model and can be stated as follows:

$$(P)\ \text{Maximize} \sum_{j=1}^{n} c_j x_j \tag{1}$$

$$\text{subject to} \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i \in M = \{1, \ldots, m\} \tag{2}$$

$$x_j \in \{0, 1\} \quad j \in N = \{1, \ldots, n\} \tag{3}$$

where n is the number of items and m is the number of knapsack constraints with capacities b_i ($i \in M$). Each item j ($j \in N$) yields c_j units of profit and consumes a given amount of resource a_{ij} for each knapsack i. In comparison to general 0–1 integer problems, the MKP coefficients are all positive integer values ($c \in \mathbb{N}^n$, $A \in \mathbb{N}^{m \times n}$, $b \in \mathbb{N}^m$) and there are usually few constraints compared to the number of variables. The MKP is a special case of 0–1 integer programs; it is known to be NP-hard, but not strongly NP-hard. This problem has been widely discussed in the literature, and efficient exact and approximate

* Corresponding author. Tel.: +33 0 466384016.
E-mail addresses: Sylvain.Boussier@univ-avignon.fr (S. Boussier), Michel.Vasquez@ema.fr (M. Vasquez), Yannick.Vimont@ema.fr (Y. Vimont), Said.Hanafi@univ-valenciennes.fr (S. Hanafi), Philippe.Michelon@univ-avignon.fr (P. Michelon).

0166-218X/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.dam.2009.08.007

algorithms have been developed for obtaining optimal and near-optimal solutions (the reader is referred to [10] and [11] for a comprehensive annotated bibliography). Some very efficient algorithms [19,20] exist when $m = 1$, but as m increases, exact methods usually fail to provide an optimal solution for even moderate size instances. A wide variety of different techniques have been used for solving optimally the MKP. Among others, we can mention dynamic programming approaches [13,16,28], tree search algorithms [24,8,12,27] and hybrid constraint programming and integer linear programming approaches [21].

Resolution Search was proposed by Chvátal (1997) [5] as an alternative to Branch & Bound for pure 0–1 linear programming problems. Resolution Search is strongly related to Dynamic Backtracking introduced in Artificial Intelligence literature by Ginsberg (1993) [14] to solve constraint satisfaction problems. Hanafi and Glover (2002) [17] show that the Dynamic Branch & Bound proposed by Glover and Tangedahl (1976) [15] yields similar branching strategies as Resolution Search, and other strategic alternatives in addition.

Recently Vimont et al. [27] proposed an efficient Branch & Bound algorithm for the MKP where the branching policy lies more on pruning the search tree than searching for good solutions. Although interesting in terms of computational results, this method has the disadvantage of requiring a (good) lower bound as starting point. In this paper, we propose a new exact method which hybridizes Resolution Search, a Branch & Bound algorithm inspired by [27] and a Depth First Search enumeration for the MKP. The proposed algorithm is self-sufficient and does not require any lower bound as starting value.

We propose an improvement of the *waning phase* of Resolution Search for the special case of MKP based on implied relationships that can be deduced between nodes of the search tree. We show that partitioning the set of feasible solutions by imposing a cardinality bound (*hyperplanes with a fixed number of items*) enables us to obtain a tighter upper bound and thus to prune efficiently the search tree. This decomposition leads us to consider a set of subproblems for which the number of items is set to an integer value. We show that the structure of Resolution Search enables these subproblems to be solved by switching from one to another instead of solving them one by one. This method induces some diversification in the search process and seems to enable good solutions to be found rapidly. Finally, we detail the combination of Resolution Search and the Branch & Bound in which subproblems of small size generated by Resolution Search are solved completely with Branch & Bound. The resulting algorithm gives an exact approach that is both able to obtain good feasible solutions rapidly and to solve hard instances in less CPU time than existing published approaches [18,27] and a commercial MIP solver. Moreover, for the first time (to our knowledge), we provide the optimal values of all the `OR-Library` instances with 10 constraints and 500 variables and the optimal values of seven of the instances with 30 constraints and 250 variables.

The paper is organized as follows: The principles of Resolution Search and an improvement of the method are explained in Section 2. In Section 3, we detail the implementation of our approach for the MKP namely the hyperplane decomposition and the iterative exploration of each of the subproblems. The combination of Resolution Search and the Branch & Bound is detailed in Section 4. We provide an example in Section 5. Finally, we present our computational experiments in Section 6 and we conclude in Section 7.

2. Resolution Search

In this section we introduce the main Resolution Search features and propose an improvement of one of its mechanisms called the *waning phase*. The reader is referred to the original paper of Chvátal [5] and to further studies made by [17,7,6,22,3] for more details.

2.1. Resolution Search principle

The Resolution Search algorithm progressively restricts the set of feasible solutions that offer a possibility to improve the best known solution. The method can be viewed as generating an enumeration tree where the root corresponds to the original problem instance. The Branch & Bound procedure usually corresponds to a tree search starting from the root and exploring the descendant nodes until all *terminal nodes* are reached (a terminal node is a node whose descendants are all infeasible or a node with no children). Conversely, Resolution Search explores the search tree starting from terminal nodes until the root is reached. Each time a terminal node is encountered, it is recorded in a specific way in order to discard this node from the search space and to provide the next node in the exploration. If at least one of the problem's constraints is violated by this node, a minimal partial instantiation of the variables responsible for the constraint violation is identified and recorded in order to discard the corresponding subtree from the search space. The specificity of the recording mechanism allows the algorithm to preserve memory space while keeping completeness.

Let us note $u = (u_1, \ldots, u_n)$ a vector in $\{0, 1, *\}^n$ corresponding to a partial or complete instantiation of the variables where $u_j = *$ if u_j is a free variable. Let u denote an arbitrary vector in $\{0, 1, *\}^n$, and define the associated index sets $N^0(u) = \{j \in N : u_j = 0\}$, $N^1(u) = \{j \in N : u_j = 1\}$ and $N^*(u) = \{j \in N : u_j = *\}$, we set $N(u) = N^0(u) \cup N^1(u)$. Each node u of the search tree corresponds to a subproblem $P(u)$ which is defined as :

$$(P(u)) \text{ Maximize } \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to } \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i \in M$$

$$x_j = u_j \quad j \in N(u)$$
$$x_j \in \{0, 1\} \quad j \in N$$

Thus, u is *terminal* if $P(u)$ is infeasible or if $N(u) = N$. We shall say that a node v is the *extension* of u (denoted $u \sqsubseteq v$) if $v_j = u_j$ whenever $u_j \neq *$. In other words, if $u \sqsubseteq v$ then v is a descendant of u.

Chvátal [5] proposes a data structure suited for recording the terminal nodes in a family denoted $\mathscr{F}$, called *path-like family*, with a size that does not exceed the number of binary variables n. The family $\mathscr{F}$ is composed of a list of terminal nodes and for each terminal node $u \in \mathscr{F}$, an index $\rho(u)$ is associated such that $\rho(u) \in N(u)$. For each family $\mathscr{F}$, we are able to construct a node $u^{\mathscr{F}}$ which is the extension of no nodes in $\mathscr{F}$. For each terminal node $u \in \mathscr{F}$ with its associated index $\rho(u)$, we define a node $\tilde{u}$ such that

$$\forall j \in N, \qquad \tilde{u}_j = \begin{cases} 1 - u_j & \text{if } j = \rho(u), \\ u_j & \text{otherwise.} \end{cases}$$

Given the nodes $\tilde{u} \in \mathscr{F}$, $u^{\mathscr{F}}$ can be constructed in the following way:

$$\forall j \in N, u_j^{\mathscr{F}} = \begin{cases} \tilde{u}_j & \text{if } j \in N(u) \text{ with } u \in \mathscr{F}, \\ * & \text{if } j \in \cap_{u \in \mathscr{F}} N^*(u). \end{cases}$$

This construction makes sense because some specific rules for constructing $\mathscr{F}$ impose that $\tilde{u}_j = \tilde{v}_j$ for all $u, v \in \mathscr{F}$ with $j \in N(u) \cap N(v)$ (see [5] for more details).

Starting with $u^{\mathscr{F}}$, a function called `obstacle` performs two different phases:

1. *The waxing phase* which replaces step by step the components $u_j^{\mathscr{F}} = *$ by 0 or 1 until a terminal node u^* is reached.
2. *The waning phase* which tries to find a minimal element u^- of $\{0, 1, *\}^n$ such that $u^- \sqsubseteq u^*$ and u^- is a terminal node. Such an element is called an *obstacle*.

While an obstacle u^- has been identified, it is added to the family $\mathscr{F}$ in order to discard the corresponding subtree from the search space.

Since each ρ_i determines one of the variables which will be fixed to their opposite value in the next iteration, the choice of ρ_i has an influence on the global behavior of the algorithm. Here we propose to choose the index in $N(u^i)$ of the variable with the greatest absolute reduced cost value (see Section 2.3). This strategy provided better results than other strategies tested like choosing the variable with the lower cost (c_j) or choosing the variable with the greatest consumption ($\sum_{i \in M} a_{ij}$).

The update of $\mathscr{F}$ is achieved according to a specific mechanism based on the observation that some terminal nodes can be simplified using a *resolvent* operator. Vectors u and v are said to *clash* if there is exactly one index $j \in N(u) \cap N(v)$, such that $u_j = 1 - v_j$, their *resolvent* which we denote $w = u \nabla v$ is defined by

$$\forall j \in N, \quad w_j = \begin{cases} u_j & \text{if } u_j = v_j \\ * & \text{if } u_j = 1 - v_j. \end{cases}$$

Trivially, if vectors u and v are clashing terminal nodes, their resolvent $u \nabla v$ is also a terminal node. For example, if $u = (1, 0, *,)$ and $v = (1, 1, *)$ are terminal nodes then $u \nabla v = (1, *, *)$ is also a terminal node. During the update of $\mathscr{F}$, the new terminal node u^- can be simplified in that way by applying the resolvent operator between u^- and the other recorded terminal nodes. If it appears after simplification that $u^- = (*, *, \ldots, *)$, then the enumeration is over and the best known solution is an optimal solution; otherwise, $u^{\mathscr{F}}$ will be the next starting node for the function `obstacle`.

The waxing and the waning phases of `obstacle` are achieved by considering an evaluation function called `oracle`. This function consists in solving the continuous relaxation of $P(u)$ (denoted $\bar{P}(u)$). It takes as parameter a problem P, a node u and is defined as follows:

$$\texttt{oracle}(P, u) = \begin{cases} -\infty & \text{if } \bar{P}(u) \text{ is infeasible} \\ v(\bar{P}(u)) & \text{otherwise} \end{cases}$$

where $v(\bar{P}(u))$ is the optimal value of $\bar{P}(u)$. Let x^* be a best known solution and $\texttt{LB} = cx^*$ the objective value corresponding to a lower bound of the optimal value $v(P)$. For a given node u, if $\texttt{oracle}\ (P, u) \leq \texttt{LB}$, then u is terminal. Algorithm 1 details the function `resolution_search` which takes as a parameter a problem P and a lower bound LB of P. Algorithm 2 details the function `obstacle` which takes as a parameter a problem P, the current node $u^{\mathscr{F}}$, a lower bound LB of P and an obstacle u^- (which is actually an output), it returns a value `bound` which is either equal to $\sum_{j \in N^1(u^+)} c_j$ if a feasible solution u^+ is found or $-\infty$ otherwise.

2.2. Implicit waning phase

In the function `obstacle` (Algorithm 2), the waning phase is achieved by reversing one by one the decisions taken during the waxing phase starting from the second to last and by checking at each step whether the partial instantiation is a terminal node or not.

In our experimentations, this approach turned out to slow down the execution time of the whole process. It seems that the time expended in solving the LP-relaxation at each step is not worth the time saved in generating the obstacle.

Algorithm 1 Function `resolution_search`

```
Resolution_Search(P,LB)
{
   F = ∅;
   while((*,*,...,*) ∉ F) {
      u⁻ = (*,*,...,*);
      try = obstacle(P,u^F, LB, u⁻);
      if(try > LB)  LB = try;
      add u⁻ to F and update F;
   }
}
```

Algorithm 2 Function `obstacle`

```
obstacle(P,u,LB,u⁻)
{
   //Waxing phase
   u⁺ = u;
   if(bound = oracle(P, u⁺) > LB){
      while(N*(u⁺) ≠ ∅) {
         choose an index j ∈ N*(u⁺) and a value α in {0, 1};
         u_j⁺ = α;
         if(bound = oracle(P, u⁺) ≤ LB) break;
      }
      if(bound > LB) LB = bound;
   }
   //Waning phase
   u⁻ = u⁺; J = N(u⁻);
   while(J ≠ ∅){
      choose an index j ∈ J; set u_j⁻ = *; and J = J − {j};
      if(oracle(P, u⁻) > LB) u_j⁻ = u_j⁺;
   }
   return bound;
}
```

We propose another method for identifying a minimal obstacle which we call *implicit waning phase*. In this method, we try to identify a set of *crucial* variables directly *during* the waxing phase. When a terminal node is reached, only the branching decisions linked to the set of *crucial* variables are kept for constructing u^-. This approach is done by taking into consideration an implied relationship that might be deduced between nodes of the search tree. Considering two vectors $u, v \in \{0, 1, *\}^n$, we say that u *implies* v and we note $u \Rightarrow v$ if the instantiation of the variables x_j with $j \in N(u)$ obliges the fixing of additional variables x_j with $j \in N^*(u)$ giving the vector v (obviously $u \sqsubseteq v$). Our implicit waning phase is justified by Proposition 1.

Proposition 1. *Let u, v and w be vectors in $\{0, 1, *\}^n$. If $u \Rightarrow v$ and $v \sqsubseteq w$ is a terminal node, then the node u^- defined by*

$$\forall j \in N, \quad u_j^- = \begin{cases} w_j & \text{if } j \in N - (N(v) \cap N^*(u)) \\ * & \text{if } j \in N(v) \cap N^*(u) \end{cases}$$

is a terminal node.

Proof. We are going to prove the proposition by induction on the cardinality of the set $J = N^*(u) \cap N(v)$. Let us suppose that $|J|$ is equal to one, $J = \{j_0\}$, then the node v' defined by $v'_j = u_j$ for $j \in N - \{j_0\}$ and $v'_{j_0} = 1 - v_{j_0}$ is a terminal node since $u \Rightarrow v$. Thus the node v'' defined by

$$v''_j = \begin{cases} w_j & \text{if } j \neq j_0 \\ 1 - v_{j_0} & \text{if } j = j_0 \end{cases}$$

is also a terminal node. Moreover the terminal nodes w and v'' clash since $v \sqsubseteq w$ hence $v_{j_0} = w_{j_0}$. Hence, the resolvent of w and v'' which is the node $u' = w \nabla v''$ is a terminal node. Now, let us suppose that the proposition is true for a set $J = N^*(u) \cap N(v)$ of length k. We will prove that the proposition is still true if J is of length $k + 1$. Let $j_0 \in N^*(u) \cap N(v)$, the node v' defined by $v'_j = v_j$ for $j \in N - \{j_0\}$ and $v'_{j_0} = 1 - v_{j_0}$ is a terminal node since $u \Rightarrow v$. Similarly, the node v'' defined by $v''_j = w_j$ if $j \neq j_0$ and $v''_{j_0} = 1 - v_{j_0}$ is also a terminal node. The terminal nodes w and v'' clash since $v_{j_0} = w_{j_0}$. Hence, the

resolvent of w and v'' which is the node $w' = w\nabla v''$ is a terminal node. Now, we can apply the hypothesis of induction on u, v' and w' where the node v' is defined now by $v'_j = v_j$ for $j \in N - \{j_0\}$ and $v'_{j_0} = *$. This completes the proof that node u' is a terminal node. □

This method provides obstacles at a lower computational cost compared to the original waning phase of Resolution Search. However, its efficiency is related to the structure of the problem we attempt to solve since implied relationships have to be deduced. We show in Section 2.3 that, for the special case of MKP, the implicit waning phase can be implemented using the information brought by the reduced costs at the optimality of the LP-relaxation solution.

2.3. Implicit waning phase using reduced costs

Let us consider the upper bound $\texttt{UB} = c\bar{x}$, where $\bar{x}$ is the LP-relaxation solution. Let $\bar{c}$ be the vector of reduced costs. If we know a lower bound $\texttt{LB} \in \mathbb{N}$ of the problem, then each better solution x must satisfy the following constraint [23,1,27]:

$$\sum_{j \in N^0(\bar{x})} |\bar{c}_j| x_j + \sum_{j \in N^1(\bar{x})} |\bar{c}_j|(1 - x_j) \leq \lfloor \texttt{UB} \rfloor - \texttt{LB} - 1 \tag{4}$$

The constraint (4) (known as the *reduced costs constraint*) enables us to fix the non-basic variables x_j so that $|\bar{c}_j| > \lfloor \texttt{UB} \rfloor - \texttt{LB} - 1$ to their optimal value $\bar{x}_j$ in the LP-relaxation solution. If we consider the right-hand side of this constraint $gap = \lfloor \texttt{UB} \rfloor - \texttt{LB} - 1$ as an available amount of reduced cost to fix non-basic variables to the opposite of their optimal value, then each node $u \in \{0, 1, *\}^n$ must satisfy the following constraint:

$$\sum_{j \in \mathscr{G}(u)} |\bar{c}_j| \leq \lfloor \texttt{UB} \rfloor - \texttt{LB} - 1 \tag{5}$$

where $\mathscr{G}(u) = \{j \in N(\bar{x}) \mid u_j = 1 - \bar{x}_j\}$ is the set of indexes of the non-basic variables x_j that are set to their opposite value in u.

At the start of function `obstacle`, we initialize $u = u^{\mathscr{F}}$ and check whether u satisfies the reduced costs constraint by updating the value gap (for each instantiation $u_j = 1 - \bar{x}_j$, we set $gap = gap - |\bar{c}_j|$). If this is not the case, we return $u^- = \{u_j \mid j \in \mathscr{G}(u)\}$ as the minimal node responsible for the infeasibility otherwise we fix the non-basic variables x_j so that $u_j = *$ and $|\bar{c}_j| > gap$ to their optimal value. Then, we continue the waxing phase by fixing free variables until we reach a terminal node. Once a terminal node is reached, the obstacle u^- is constructed by taking only the *crucial* variables of this terminal node into consideration (i.e the variables which have not been fixed by the reduced costs constraint). The branching strategy consists in choosing the non-basic free variable with the greatest reduced cost and to fix it to its optimal value. In case all the free variables are basic variables, we set the most fractional free variable to the closest integer value. The Algorithm 3 details the function `obstacle` embedding the implicit waning phase.

3. Hyperplane decomposition of the search space

In the proposed approach, the MKP is tackled by decomposing the problem into several subproblems where the number of items to choose is fixed at a given integer value (hyperplane). First we introduce the search space decomposition then we present a specific implementation of Resolution Search which permits to explore efficiently all these subproblems.

3.1. Hyperplane decomposition

Considering that LB is the objective value of the best know solution of the problem, we can define $X = \{x \mid Ax \leq b, cx \geq \texttt{LB} + 1, x \in \{0, 1\}^n\}$ as the set of feasible solutions strictly better than LB and consider the following problems:

$$P^+ : \text{ maximize } \{ex \mid x \in \bar{X}\} \quad \text{and} \quad P^- : \text{ minimize } \{ex \mid x \in \bar{X}\}$$

where e is the vector of dimension n with all its components equal to one and $\bar{X} = \{x \mid Ax \leq b, cx \geq \texttt{LB} + 1, x \in [0, 1]^n\}$. In what follows, we note $v(P)$ the optimal value of a given problem P. Let $k_{min} = \lceil v(P^-) \rceil$ and $k_{max} = \lfloor v(P^+) \rfloor$, then we have $v(P) = max\{v(P_k) \mid k_{min} \leq k \leq k_{max}\}$ where

$$P_k : \text{ maximize } \{cx \mid x \in X, ex = k\}.$$

Solving the MKP by tackling separately each of the subproblems P_k for $k = k_{min}, \ldots, k_{max}$ appeared to be an interesting approach [25–27]. Indeed, the additional constraint $(ex = k)$ provides a tighter upper bound than the classical LP-relaxation, thus it enables us to improve the pruning of the search tree and to get better variable fixing using the reduced costs constraint.

For example, let us consider the instance with 10 constraints and 500 variables taken from the `OR-Library` [2] (number 20) and suppose that we know a lower bound for this instance[1]. Table 1 shows the gains made by the hyperplane decomposition combined with the reduced costs constraint. The left column corresponds to the upper bound of the classical LP-relaxation with the corresponding number of fixed variables; and the right column shows the same data with the hyperplane decomposition.

[1] In this example, $\texttt{LB} = 304214$ is given by the `greedy` algorithm introduced in Section 3.2.

Algorithm 3 Function `obstacle` with implicit waning phase for the MKP

```
obstacle(P, u, LB, u⁻)
{
   gap = ⌊UB⌋ − LB − 1;
   for(j ∈ N(x̄))
      if(u_j = 1 − x̄_j) {gap = gap − c̄_j; u_j⁻ = u_j}
   if(gap < 0) return 0;
   u⁺ = u; u⁻ = u;
   if(bound = oracle(P, u⁺) > LB){
      //Implicit waning phase
      for(j ∈ N*(u⁺) ∩ N(x̄))
         if(|c̄_j| > gap) u_j⁺ = x̄_j;
       //Waxing phase
      while(N*(u⁺) ≠ ∅) {
         if(it exists j such that x_j is non-basic and j ∈ N*(u⁺))
            choose j in N*(u⁺) such that |c̄_j| is maximal and set α = x̄_j;
         else
            choose the index j in N*(u⁺) such that |x̄_j − 0.5| is minimal and
            set α = ⌈x̄_j⌉ if x̄_j > 0.5 and α = ⌊x̄_j⌋ otherwise;
         u_j⁺ = α; u_j⁻ = α;
         if(bound = oracle(P, u⁺) ≤ LB) break;
      }
      if(bound > LB) LB = bound;
   }
   return bound;
}
```

Table 1
Improving upper bound and variable fixing using $ex = k$ and the reduced cost constraint.

LP-relaxation		Hyperplane decomposition		
Upper bound	# fixed variables	Hyperplane	Upper bound	# fixed variables
304555.03	30	$e.x = 378$	304546.29	37
		$e.x = 379$	304553.62	30
		$e.x = 377$	304516.12	67
		$e.x = 380$	304539.29	43
		$e.x = 376$	304427.94	173
		$e.x = 381$	304502.74	88
		$e.x = 375$	304313.84	302
		$e.x = 382$	304425.70	168
		$e.x = 383$	304312.81	314

As we will show in Section 4, this last constraint permits also to solve efficiently small subproblems with a brute force algorithm.

3.2. Iterative resolution search

The structure of Resolution Search allows us to solve the original problem P by progressively exploring the search space of each subproblem P_k. More precisely, let $\mathscr{F}_k$ be the path-like family associated to Resolution Search for the subproblem P_k. At each step of the search, $\mathscr{F}_k$ provides all the information about the state of the search: the terminal nodes recorded at this stage and the next node $u^{\mathscr{F}_k}$ to explore. This makes it possible to execute some iterations of resolution search at a given P_k, then continue to another $P_{k'}$ and come back to the subproblem P_k again without any loss of information. The Algorithm 4 describes this way of exploration starting with an initial lower bound LB which becomes an optimal solution of P at the end. The starting lower bound can be generated by any heuristic or metaheuristic for the MKP. In our numerical experiments, this lower bound is generated with a greedy algorithm from the LP-relaxation solution of P, denoted $\bar{x}$, by fixing $x_j = 1$ for a maximum number of items j with the greatest value $\bar{x}_j$.

4. Combining Resolution Search, Branch & Bound and simple Depth First Search enumeration

In the proposed algorithm, when subproblem $P(u)$ is small enough in the waxing phase (less than a given parameter `spb_size`, see Section 6), we solve it with a Branch & Bound algorithm inspired by the algorithm published in [27].

Algorithm 4 Function `iterative_resolution_search`

```
iterative_resolution_search(P,LB)
{
   Compute the bounds k_min and k_max;
   Set K = {k_min, ..., k_max};
   for(k = k_min; k ≤ k_max; k++) F_k = ∅;
   While (K ≠ ∅) do{
      Choose k ∈ K;
      Choose Nb_Iter_k ≥ 1;
      iter = 0;
      while(iter < Nb_iter_k and (*,*,...,*) ∉ F_k)){
         try = obstacle(P_k, u^{F_k}, LB, u^-);
         if(try > LB)  LB = try;
         add u^- to F_k and update F_k;
         if((*,*,...,*) ∈ F_k)   {K = K − {k}; break;}
         iter++;
      }
   }
}
```

Unlike classical Branch & Bound procedures, this algorithm search the most unpromising parts of the search tree first. At each node, it follows the policy described below:

1. Solve the LP-relaxation of the problem with the simplex algorithm.
2. Branch on the non-basic variable (x_j) with the highest reduced cost ($|\bar{c}_j|$) and fix it to the opposite of its optimal value ($1 - \bar{x}_j$).
3. Update $gap = \lfloor \mathtt{UB} \rfloor - \mathtt{LB} - 1$ by $gap = gap - |\bar{c}_j|$.
4. Fix all the non-basic variables with a reduced cost greater than gap to their optimal value.

Instead of branching on the most fractional basic variable as it is usually done, the branching strategy consists of fixing the non-basic variable with the higher reduced cost to the *opposite* of its optimal value. The aim is to focus more on pruning the search tree than rapidly finding good solutions.

In this Branch & Bound algorithm, when the number of remaining free variables is less than a given parameter `dfs_size`, the corresponding subproblem is solved with a *brute force* Depth First Search where the branching strategy consists in fixing the first free variable to 0 then to 1. In this procedure, the variables are simply enumerated without solving the LP-relaxation at each node. However, in order to enforce the pruning of the search tree, additional constraints linked to the equality $ex = k$ are taken into consideration: let $\Omega_c(k)$ be the set of the indexes of the k variables with the greatest cost (c_j), then the sum of the costs of the variables in $\Omega_c(k)$ must be greater than or equal to the best known lower bound

$$\sum_{j \in \Omega_c(k)} c_j \geq \mathtt{LB}$$

and let $\Omega_i(k)$ be the set of the indexes of the k variables with the lowest consumption (a_{ij}), then the total consumption by the variables in $\Omega_i(k)$ must be lower than or equal to the capacity b_i

$$\sum_{j \in \Omega_i(k)} a_{ij} \leq b_i \quad i = 1, 2, \ldots, m.$$

The only difference between this Branch & Bound algorithm and the one published in [27] is that a specific reduced costs propagation has been deleted. This propagation consists in checking the validity of the current node with the reduced cost constraints of all the parent nodes. In our case, since we only solve small subproblems, the number of nodes pruned using this technique is not worth the time saved, hence we have removed this procedure.

Globally, the search tree is divided into three parts: the top branches (great reduced costs variables) are enumerated with Resolution Search, the middle branches (middle reduced costs variables) are enumerated with Branch & Bound and the lower branches (low reduced costs variables and basic variables) are enumerated with Depth First Search procedure (see Fig. 1).

5. Example

In order to illustrate the procedure `obstacle`, we take the following problem (P^{ex}) as an example:

$$\begin{array}{ll} (P^{ex}) \text{ Maximize} & 10x_1 + 10x_2 + 8x_3 + 8x_4 + 7x_5 \\ \text{subject to} & 5x_1 + 2x_2 + 10x_3 + 1x_4 + 3x_5 \leq 10 \\ & 2x_1 + 11x_2 + 2x_3 + 10x_4 + x_5 \leq 11 \\ & x = (x_1, x_2, x_3, x_4, x_5)^T \in \{0, 1\}^5. \end{array}$$

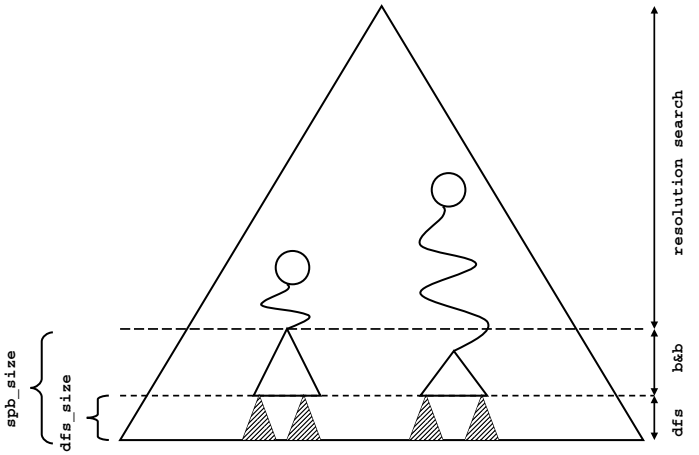

Fig. 1. Global view of the exploration process of an hyperplane.

Let us suppose that we are trying to solve the problem P_2^{ex} which corresponds to P^{ex} with the additional constraint ($ex = 2$). The resolution of the LP-relaxation of P_2^{ex} gives us an optimal value ($UB = 19.55$), an optimal solution $\bar{x}$ and the reduced costs vector $\bar{c}$ such that:

$$x = (1, 0.78, 0.22, 0, 0) \quad \text{and} \quad \bar{c} = (2, 0, 0, -1.78, -0.78).$$

Let us suppose that we know a lower bound $\texttt{LB} = 16$ of the problem, then the corresponding reduced cost constraint is

$$2(1 - x_1) + 1.78x_4 + 0.78x_5 \leq 2.$$

At the beginning of the procedure, the LP-relaxation of the problem is solved for each available hyperplane in order to give us the information needed for the reduced cost constraint. Let *gap* be the value $\lfloor \texttt{UB} \rfloor - \texttt{LB} - 1$.

The first step, called the *consistency phase*, consists of checking the feasibility of $u^{\mathscr{F}}$. If a constraint is violated, the descent phase stops and the corresponding partial instantiation is recorded as the terminal node u^- in $\mathscr{F}$. At the same time, the reduced cost constraint (5) is checked and the *gap* value is updated: for each non-basic variable set to the opposite of its optimal value ($1 - \bar{x}_j$), the absolute value of its reduced costs ($|\bar{c}_j|$) is subtracted from *gap*. If it happens that $gap < 0$ then the current partial solution is a terminal node. In this case, u^- is only composed of the variables set at the opposite of their optimal value in $u^{\mathscr{F}}$. For example, suppose that $u^{\mathscr{F}} = (0, 0, 0, 1, *)$, then $gap = gap - (|\bar{c}_1| + |\bar{c}_4|) = 2 - (2 + 1.78) = -1.78$. Since $gap < 0$, $u^{\mathscr{F}}$ is a terminal node and since $u_1^{\mathscr{F}}$ and $u_4^{\mathscr{F}}$ are the only instantiations responsible for the constraint violation, we set $u^- = (0, *, *, 1, *)$.

If $u^{\mathscr{F}}$ is feasible, we proceed to the *implicit waning phase*: for each remaining non-basic free variable x_j, if $|\bar{c}_j| > gap$ we set $u_j^+ = \bar{x}_j$. The branching decisions taken in this phase are just a consequence of the instantiated variables in $u^{\mathscr{F}}$. Then, according to Proposition 1, they are not included in u^-. In order to illustrate this process, suppose that `obstacle` starts with $u^{\mathscr{F}} = (0, *, *, *, *)$, then $gap = gap - |\bar{c}_1| = 0$. Since, $|\bar{c}_4| = 1.78$ is greater than *gap*, x_4 must be set to $\bar{x}_4 = 0$. Consequently, $u^{\mathscr{F}} = (0, *, *, *, *)$ implies $u^+ = (0, *, *, 0, *)$ so u_4^+ will be set to $*$ in u^-.

In the next phase, the algorithm begins with the *waxing phase* which consists in assigning values to free variables according to the following strategy: (1) if there exists at least one non-basic free variable, choose the free variable with the greatest absolute reduced cost value and fix it to its optimal value $\bar{x}_j$; (2) if the set of free variables is only composed of basic variables, fix the most fractional variable to the closest integer value. If we consider the previous example, $u^+ = (0, *, *, 0, *)$ after the implicit waning phase. Since x_5 is the non-basic free variable with the greatest absolute reduced costs value ($|\bar{c}_5| = 0.78$), it is fixed to its optimal value $\bar{x}_5$ ($u_5^+ = 0$).

Once the number of remaining free variables is less than or equal to `spb_size`, the waxing phase stops and the corresponding subproblem is solved with the Branch & Bound algorithm. This subproblem includes the free variables with the lowest reduced cost and the basic variables. Obviously, since the Branch & Bound algorithm explores the whole subtree corresponding to these variables, the obstacle u^- does not contain any branching choices made during this phase. Only the branching choices taken during the consistency phase *and/or* the waxing phase are considered. In the above example, let us suppose that $\texttt{spb\_size} = 2$, the enumeration of the remaining variables x_2 and x_3 gives the following terminal nodes:

$$\begin{aligned}
&u^* = (0, 0, 0, 0, 0), \quad cu^* \leq \texttt{LB},\\
&u^* = (0, 0, 1, 0, 0), \quad cu^* \leq \texttt{LB},\\
&u^* = (0, 1, 0, 0, 0), \quad cu^* \leq \texttt{LB},\\
&u^* = (0, 1, 1, 0, 0), \quad Au^* \not\leq b : \text{infeasible}.
\end{aligned}$$

Since no terminal node improves the best known solution, the obstacle $u^- = (0, *, *, *, 0)$ is added to $\mathscr{F}$. Note that `spb_size` can be updated dynamically, depending on whether we want to favor the Resolution Search part or the Branch & Bound part.

6. Experimental results

In this section, we first detail the settings of the parameters `NB_iter_k`, `spb_size` and `dfs_size`, then we describe the results obtained on the well-known hard instances of Chu and Beasley.

6.1. Parameter settings

Three parameters must be taken into account in this algorithm: the parameter `NB_iter_k` (Section 3.2) which corresponds to the number of Resolution Search iterations for each hyperplane, the parameter `spb_size` (Section 4) which corresponds to the size of the subproblems solved with the Branch & Bound algorithm and the parameter `dfs_size` (Section 4) which corresponds to the size of the subproblems solved with the Depth First Search algorithm.

In order to enforce the search diversification, we set `NB_iter_k` $= 1$. In this case, only one iteration of Resolution Search is executed successively for each hyperplane. In our preliminary experiments it was found that this setting found good solutions rapidly.

The value of `spb_size` has a great impact on our hybrid method because it determines the proportion of search space enumerated by Resolution Search and the one enumerated by the Branch & Bound algorithm. Experimentally, it appeared that a small value of `spb_size` obtains good solutions quickly but requires more CPU time to prove the optimality of the best solution found. On the other hand, a large value `spb_size` can give too much importance to the Branch & Bound and if we do not know a good solution, it will take time to solve each of the subproblems. In order to balance the advantages and the disadvantages of each extreme, we update `spb_size` dynamically, starting with a low value (in order to give priority to Resolution Search for finding good solutions) and then increasing this value (in order to give more priority to the Branch & Bound algorithm when a good solution is known). In our experimentations, `spb_size` is incremented every 100 iterations from 20 to 60 and is set to 20 again if a better solution is found.

The parameter `dfs_size` also has an impact on the global behavior of the algorithm. Its value determines the size of the subproblems solved with the Depth First Search algorithm. Since this algorithm does not compute a bound at each explored node of the search tree, it is not efficient if the subproblem to solve is too large. On another hand, it is very efficient if the problem to solve is small enough since it does not "waste" its time executing all the simplex computations required by a classical Branch & Bound algorithm. Experimentally, we observed that the algorithm performed well with `dfs_size` $= 20$.

6.2. Results on Chu and Beasley's benchmarks

We have evaluated our algorithm on the well-known set of MKP instances proposed by Chu and Beasley available on the OR-Library website [2]. These instances were generated using the procedure described in [9]. The website contains a collection of 270 problems with $n = 100$, 250 and 500 variables and $m = 5$, 10 and 30 constraints. Thirty problems were generated for each n–m combination. The name of each problem is **cbm.n_r**, where m is the number of constraints, n the number of variables and r the number of the instance. Our algorithm has been tested on a Dell Precision 690 2 Dual Core 3 GHz with 2 GB RAM. The results are compared to those produced by the commercial solver CPLEX 11 and our previous algorithm [27]. Computational results obtained on the cb10.250 and the cb5.500 instances are presented in Tables 2 and 3. The description of the columns is the following:

- *Instance* is the name of the instance.
- *Objective* is the best objective value found by our algorithm (with * if we proved that it is the exact optimal value).
- *Obj. (s)* is the time in seconds spent for obtaining the best objective value found.
- *Proof (s)* is the time in seconds spent for proving that the best objective value found is the optimal value of the problem or - if the proof has not been found.

The column RS + B&B corresponds to the results obtained with the combination of Resolution Search and Branch & Bound algorithms, the column B&B corresponds to the results obtained with the Branch & Bound published in [27] and the column CPLEX gives the results obtained by CPLEX 11 (ERROR if CPLEX exceeds the capacity of RAM memory). The results obtained with the 10 constraint, 250 variable instances are presented in Table 2 and those obtained with the 5 constraint, 500 variable instances are presented in Table 3. These two Tables clearly show the contribution of the proposed method in terms of computational results. On the one hand, the optimal values are rapidly found and on the other hand, the proof times are better for all the instances when compared to the previously published Branch & Bound algorithm [27] and CPLEX 11.

In addition, we evaluated our algorithm on the cb10.500 and the cb30.250 instances whose optimal values are unknown. The obtained results are presented in Tables 4 and 5. The description of the columns is the following:

- *Instance* is the name of the instance.
- *Best known* is the value of the best known solution found either by Chu and Beasley [4] (CB) or Vasquez and Vimont [26] (VV) or Wilbaut and Hanafi [29] (WH).
- *Objective* is the best objective value found by our algorithm (with * if we proved that it is the exact optimal value).

Table 2
Results for the 10 constraint, 250 variable instances of the OR-Library.

Instance	Objective	RS + B&B		B&B [27]	CPLEX
		Obj. (s)	Proof (s)	Proof (s)	Proof (s)
cb10.250_0	**59187***	0	**1995**	3075	14074
cb10.250_1	**58781***	195	**906**	26346	1356
cb10.250_2	**58097***	1	**477**	905	4099
cb10.250_3	**61000***	1143	**3858**	5448	36286
cb10.250_4	**58092***	442	**12449**	17273	ERROR
cb10.250_5	**58824***	3	**448**	659	4382
cb10.250_6	**58704***	0	**388**	631	2188
cb10.250_7	**58936***	2055	**34976**	52397	ERROR
cb10.250_8	**59387***	201	**1434**	2656	6153
cb10.250_9	**59208***	0	**2667**	4141	12825
cb10.250_10	**110913***	336	**2155**	2971	7792
cb10.250_11	**108717***	378	**2112**	4479	12206
cb10.250_12	**108932***	2	**1284**	2029	12396
cb10.250_13	**110086***	54	**5743**	9054	ERROR
cb10.250_14	**108485***	0	**951**	1383	4677
cb10.250_15	**110845***	56	**2478**	3572	16528
cb10.250_16	**106077***	20	**3078**	4720	10481
cb10.250_17	**106686***	489	**2017**	3322	11023
cb10.250_18	**109829***	10	**1758**	2890	11565
cb10.250_19	**106723***	14	**484**	830	2993
cb10.250_20	**151809***	211	**350**	583	2619
cb10.250_21	**148772***	0	**935**	1370	6918
cb10.250_22	**151909***	0	**175**	496	1380
cb10.250_23	**151324***	190	**337**	3293	1385
cb10.250_24	**151966***	319	**487**	559	3332
cb10.250_25	**152109***	1	**193**	490	1589
cb10.250_26	**153131***	4	**30**	143	183
cb10.250_27	**153578***	0	**1967**	5081	11905
cb10.250_28	**149160***	65	**217**	765	817
cb10.250_29	**149704***	0	**247**	495	869

Table 3
Results for the 5 constraint, 500 variable instances of the OR-Library.

Instance	Objective	RS + B&B		B&B [27]	CPLEX
		Obj. (s)	Proof (s)	Proof (s)	Proof (s)
cb5.500_0	**120148***	40	**73**	891	3962
cb5.500_1	**117879***	6	**10**	692	255
cb5.500_2	**121131***	3	**23**	745	2644
cb5.500_3	**120804***	9	**31**	422	3106
cb5.500_4	**122319***	3	**18**	373	859
cb5.500_5	**122024***	30	**44**	624	2139
cb5.500_6	**119127***	2	**34**	570	5233
cb5.500_7	**120568***	4	**18**	538	897
cb5.500_8	**121586***	32	**51**	785	3210
cb5.500_9	**120717***	0	**29**	1864	2494
cb5.500_10	**218428***	35	**42**	561	737
cb5.500_11	**221202***	0	**7**	1437	200
cb5.500_12	**217542***	148	**155**	1094	1919
cb5.500_13	**223560***	0	**54**	811	1592
cb5.500_14	**218966***	5	**7**	325	113
cb5.500_15	**220530***	14	**26**	545	894
cb5.500_16	**219989***	6	**14**	397	342
cb5.500_17	**218215***	0	**12**	543	346
cb5.500_18	**216976***	0	**24**	466	757
cb5.500_19	**219719***	4	**44**	649	1148
cb5.500_20	**295828***	0	**3**	513	21
cb5.500_21	**308086***	13	**29**	451	248
cb5.500_22	**299796***	0	**6**	524	72
cb5.500_23	**306480***	5	**14**	415	468
cb5.500_24	**300342***	0	**18**	408	87
cb5.500_25	**302571***	6	**10**	424	374
cb5.500_26	**301339***	3	**4**	312	90
cb5.500_27	**306454***	2	**4**	373	167
cb5.500_28	**302828***	2	**12**	485	80
cb5.500_29	**299910***	39	**62**	375	334

Table 4
Results for the 10 constraint, 500 variable instances of the OR-Library.

Instance	Best known	Objective	Obj. (h)	Proof (h)	Gap
cb10.500_0	117811 (VV)	**117821***	24.5	567.2	**10**
cb10.500_1	119232 (VV)	**119249***	68.4	272.9	**17**
cb10.500_2	119215 (VV)	119215*	18.6	768.3	0
cb10.500_3	118801 (WH)	**118829***	47.4	89.6	**4**
cb10.500_4	116514 (WH)	**116530***	86.1	2530.3	**16**
cb10.500_5	119504 (VV)	119504*	2.3	188	0
cb10.500_6	119827 (VV)	119827*	2.7	128	0
cb10.500_7	118333 (WH)	**118344***	161.7	179.6	**11**
cb10.500_8	117815 (VV)	117815*	86.3	219.9	0
cb10.500_9	119251 (VV)	119251*	3.1	354.9	0
cb10.500_10	217377 (VV)	217377*	≤0.1	515.8	0
cb10.500_11	219077 (VV)	219077*	0.5	437.6	0
cb10.500_12	217847 (VV)	217847*	≤0.1	5.5	0
cb10.500_13	216868 (VV)	216868*	≤0.1	104.4	0
cb10.500_14	213859 (VV)	**213873***	59.4	1382.1	**14**
cb10.500_15	215086 (VV)	215086*	≤0.1	43.9	0
cb10.500_16	217940 (VV)	217940*	13.4	36.1	0
cb10.500_17	219990 (VV)	219990*	150.8	348.8	0
cb10.500_18	214375 (VV)	**214382***	12.7	57.8	**7**
cb10.500_19	220899 (VV)	220899*	0.2	21.3	0
cb10.500_20	304387 (VV)	304387*	6.6	8.2	0
cb10.500_21	302379 (VV)	302379*	≤0.1	8.4	0
cb10.500_22	302416 (VV)	**302417***	67.2	105.5	**1**
cb10.500_23	300784 (VV)	300784*	0.9	3.8	0
cb10.500_24	304374 (VV)	304374*	0.1	16.8	0
cb10.500_25	301836 (VV)	301836*	29.7	30.9	0
cb10.500_26	304952 (VV)	304952*	≤0.1	18.5	0
cb10.500_27	296478 (VV)	296478*	1.1	9.3	0
cb10.500_28	301359 (VV)	301359*	8.1	39.1	0
cb10.500_29	307089 (VV)	307089*	1.2	4.4	0

- *Obj. (h)* is the time in hours spent for obtaining the best objective value found.
- *Proof (h)* is the time in hours spent for proving that the best objective value found is the optimal value of the problem or - if the proof has not been found.
- *Gap* is the gap between the best objective value found by our algorithm and the best known objective value (*Gap*= *Objective* − *Best known*).

The results on the cb10.500 and cb30.250 instances show that our algorithm is not really suitable for these instances since the times are huge for most of them. However they illustrate several qualities of the proposed method: (1) in spite of the large CPU times, our algorithm never ran out of memory and (2) the computational time required for finding the optimal solutions is quite low for the most part of the instances. The best known solutions of 8 of the cb10.500 instances and 27 of the cb30.250 instances have been improved. All the cb10.500 instances and 7 of the cb30.250 instances have been optimally solved. We also tried to execute CPLEX 11 on these instances and it appeared that it ran out of memory after at most 3 hours of computational time.

7. Conclusion

We proposed an exact algorithm for the 0–1 Multidimensional Knapsack Problem which combines Resolution Search, a Branch & Bound and a Depth First Search algorithm that exploit efficiently both the reduced costs and the fixed number of item constraints. This approach uses Resolution Search to guide the search toward promising parts of the search space and employs the Branch & Bound and the Depth First Search algorithms to solve small subproblems efficiently. We showed that the structure of Resolution Search enables us to explore iteratively different subproblems while keeping completeness. The proposed *iterative Resolution Search* algorithm enhances the diversification and improves the ability of the algorithm to obtain good solutions rapidly. Moreover, we introduced an alternative method for identifying the terminal nodes which we call *implicit waning phase*. This method can replace the original waning phase of *Resolution Search* and may improve its efficiency if some implied relationships are deduced between nodes of the search tree. The computational experiments made on several MKP instances show the contribution of the proposed method. Our algorithm is faster than exact existing approaches and a commercial solver for medium size instances of the OR-Library (the 10 constraint, 250 variable instances and the 5 constraint, 500 variable instances) and it enabled us to find new optimal solutions for large-scale instances (some of the 30 constraint, 250 variable instances and all the 10 constraint, 500 variable instances). Moreover, 35 of the best known solutions for these instances have been improved. These results can be useful for researchers in the field of metaheuristics who would like to evaluate their algorithms.

Table 5
Results for the 30 constraint, 250 variable instances of the OR-Library.

Instance	Best known	Objective	Obj. (h)	Proof (h)	Gap
cb30.250_0	56693 (CB)	**56842**	39.7	–	**149**
cb30.250_1	58318 (CB)	**58418**	30.9	–	**100**
cb30.250_2	56553 (CB)	**56614**	36.6	–	**88**
cb30.250_3	56863 (CB)	**56930**	≤0.1	—	**67**
cb30.250_4	56629 (CB)	56629	21.7	—	0
cb30.250_5	57119 (CB)	**57205**	9.8	–	**86**
cb30.250_6	56292 (CB)	**56348**	53.9	–	**56**
cb30.250_7	56403 (CB)	**56457**	0.2	–	**54**
cb30.250_8	57442 (CB)	**57447**	84.7	–	**5**
cb30.250_9	56447 (CB)	56447	≤0.1	–	0
cb30.250_10	107689 (CB)	**107755**	4.1	–	**66**
cb30.250_11	108338 (CB)	**108392**	0.7	–	**54**
cb30.250_12	106385 (CB)	**106442**	0.3	–	**57**
cb30.250_13	106796 (CB)	**106876**	86.9	–	**80**
cb30.250_14	107396 (CB)	**107414**	3.3	–	**18**
cb30.250_15	107246 (CB)	**107271**	4.9	–	**25**
cb30.250_16	106308 (CB)	**106372**	20.3	–	**64**
cb30.250_17	103993 (CB)	**104032**	0.1	–	**39**
cb30.250_18	106835 (CB)	**106856**	1.2	–	**21**
cb30.250_19	105751 (CB)	**105780**	3.6	–	**29**
cb30.250_20	150083 (CB)	**150163***	17.5	33.2	**80**
cb30.250_21	149907 (CB)	**149958***	0.9	14.5	**51**
cb30.250_22	152993 (CB)	**153007***	6	17.8	**14**
cb30.250_23	153169 (CB)	**153234***	≤0.1	14.8	**65**
cb30.250_24	150287 (CB)	150287*	≤0.1	19.4	0
cb30.250_25	148544 (CB)	**148574**	≤0.1	–	**30**
cb30.250_26	147471 (CB)	**147477***	2.4	23.8	**6**
cb30.250_27	152841 (CB)	**152912***	≤0.1	50	**71**
cb30.250_28	149568 (CB)	**149570**	16.8	–	**2**
cb30.250_29	149572 (CB)	**149668**	0.9	–	**96**

Acknowledgments

We would like to express our thanks to Christophe Wilbaut for his help on the numerical experimentations of this paper. We would also like to thank the anonymous referees for their valuable advice.

References

[1] E. Balas, C. Martin, Pivot and complement-heuristic for 0-1 programming, Management Science 26 (1) (1980) 86–96.
[2] J. Beasley, Or-library: Distributing test problems by electronic mail, Journal of Operational Research Society 41 (1990) 1069–1072. http://people.brunel.ac.uk/~mastjjb/jeb/info.html.
[3] S. Boussier, Étude de Resolution Search pour la programmation linaire en variables binaires, Ph.D. Thesis, University of Montpellier II, 2008.
[4] P.C. Chu, J.E. Beasley, A genetic algorithm for the multidimensional knapsack problem, Journal of Heuristics 4 (1) (1998) 63–86.
[5] V. Chvátal, Resolution search, Discrete Applied Mathematics 73 (1997) 81–99.
[6] G. Codato, M. Fischetti, Combinatorial Benders' Cuts, in: Lecture Notes in Computer Science, Integer Programming and Combinatorial Optimization (2004) 178–195.
[7] S. Demassey, C. Artigues, P. Michelon, An application of resolution search to the rcpsp, in: 17th European Conference on Combinatorial Optimization ECCO'04, Beyrouth, Lebanon, 2004.
[8] D. Fayard, G. Plateau, An algorithm for the solution of the 0-1 knapsack problem, Computing 28 (3) (1982) 269–287.
[9] A. Fréville, G. Plateau, An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem, Discrete Applied Mathematics 49 (1994) 189–212.
[10] A. Fréville, The multidimensional 0-1 knapsack problem: An overview, European Journal of Operational Research 155 (2004) 1–21.
[11] A. Fréville, S. Hanafi, The multidimensional 0-1 knapsack problem - bounds and computational aspect. Advances in operations research, in: M. Guignard, K. Spielberg (Eds.), Annals of Operations Research 139 (1) (2005), 195–227.
[12] B. Gavish, H. Pirkul, Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality, Mathematical Programming 31 (1985) 78–105.
[13] P. Gilmore, R. Gomory, The theory and computation of knapsack functions, Operations Research 14 (1966) 1045–1075.
[14] M.L. Ginsberg, Dynamic backtracking, Journal Of Artificial Intelligence Research 1 (1993) 25.
[15] F. Glover, L. Tangedahl, Dynamic strategies for branch and bound, OMEGA, The International Journal of Management Science 4 (5) (1976) 571–575.
[16] C. Green, Two algorithms for solving independent multidimensional knapsack problems, Management Science Report, no. 110, Carnegie Institute of Technology, Graduate School of Industrial Administration, Pittsburgh, USA, 1967.
[17] S. Hanafi, F. Glover, Resolution search and dynamic branch-and-bound, Journal Of Combinatorial Optimization 6 (4) (2002) 401–423.
[18] R. James, Y. Nakagawa, Enumeration methods for repeatedly solving multidimensional knapsack sub-problems, IEICE - Transactions on Information and Systems E88-D (2005) 2329–2340.
[19] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, ISBN: 3-540-40286-1, 2004, 546 pp.
[20] S. Martello, P. Toth, Knapsack problems: Algorithms and computer implementations, in: Series in Discrete Mathematics and Optimization, Wiley Interscience, 1990.
[21] C. Oliva, P. Michelon, C. Artigues, Constraint and linear programming: Using reduced costs for solving the zero/one multiple knapsack problem, in: International Conference on Constraint Programming, CP 01, Proceedings of the Workshop on Cooperative Solvers in Constraint Programming, CoSolv 01, 2001, pp. 87–98.

[22] M. Palpant, C. Artigues, C. Oliva, Mars: A hybrid scheme based on resolution search and constraint programming for constraint satisfaction problems, laas report 07194, Tech. Rep, LAAS-CNRS, Université de Toulouse, France, 2007.
[23] R.M. Saunders, R. Schinzinger, A shrinking boundary algorithm for discrete system models, IEEE Transactions On Systems Science And Cybernetics ssc-6 (2) (1970) 133–140.
[24] W. Shih, A branch and bound method for the multiconstraint zero-one knapsack problem, Journal of Operational Reasearch Society 30 (1979) 369–378.
[25] M. Vasquez, J. Hao, An hybrid approach for the 0-1 multidimensional knapsack problem, in: Proc. 17th International Joint Conference on Artificial Intelligence, IJCAI-01, Seattle, WA, 2001.
[26] M. Vasquez, Y. Vimont, Improved results on the 0-1 multidimensional knapsack problem, European Journal of Operational Research 165 (1) (2005) 70–81.
[27] Y. Vimont, S. Boussier, M. Vasquez, Reduced costs propagation in an efficient implicit enumeration for the 01 multidimensional knapsack problem, Journal of Combinatorial Optimization 15 (2008) 165–178.
[28] H. Weigartner, D. Ness, Methods for the solution of the multidimensional 0/1 knapsack problem, Operations Research 15 (1967) 83–103.
[29] C. Wilbaut, S. Hanafi, New convergent heuristics for 0–1 mixed integer programming, European Journal of Operational Research 195 (2009) 62–74.