A hybrid dynamic programming approach to the biobjective binary knapsack problem

Charles Delort, Olivier Spanjaard

A Hybrid Dynamic Programming Approach to the Biobjective Binary Knapsack Problem

CHARLES DELORT, Laboratoire d'Informatique de Paris 6 (UMR CNRS 7606) – UPMC
OLIVIER SPANJAARD, Laboratoire d'Informatique de Paris 6 (UMR CNRS 7606) – UPMC

This paper is devoted to a study of the impact of using bound sets in biobjective dynamic programming. This notion, introduced by Villareal and Karwan [1981], has been independently revisited by Ehrgott and Gandibleux [2007], as well as by Sourd and Spanjaard [2008]. The idea behind it is very general, and can therefore be adapted to a wide range of biobjective combinatorial problem. We focus here on the biobjective binary knapsack problem. We show that using bound sets to perform a hybrid dynamic programming procedure embedded in a two phases method [Ulungu and Teghem 1995] yields numerical results that outperform previous dynamic programming approaches to the problem, both in execution times and memory requirements.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on discrete structures; G.2.1 [**Combinatorics**]: Combinatorial algorithms; I.2.8 [**Information Systems Applications**]: Dynamic programming

General Terms: Algorithms, Performance

Additional Key Words and Phrases: multiobjective optimization, biobjective binary knapsack problem

1. INTRODUCTION

Multiobjective combinatorial optimization (MOCO) deals with combinatorial problems where every solution is evaluated according to several objectives. Interest in this area has tremendously grown over the last two decades. A thorough presentation of the field can be found for instance in a book by Ehrgott [2005]. When multiple objectives are involved, and according to the available preferential information, one can either search for a single "best compromise" solution [Galand 2008] (if the type of compromise sought is known), or generate the whole set of Pareto optimal solutions (if the type is unknown), i.e. solutions that cannot be improved on one objective without being depreciated on another one. We are interested here in the latter approach. Most of the classical exact and approximate methods for finding an optimal solution in single objective discrete optimization have been revisited for finding the Pareto set under multiple objectives, e.g. dynamic programming [Daellenbach and De Kluyver 1980; Klamroth and Wiecek 2000], branch and bound [Bitran and Rivera 1982; Kiziltan and Yucaoglu 1983; Marcotte and Soland 1986; Mavrotas and Diakoulaki 1998], greedy algorithm [Serafini 1986], as well as many heuristic and metaheuristic methods [Ehrgott and Gandibleux 2004].

In order to perform implicit enumeration in multiobjective optimization problems, the formal notion of *bound set* needs to be introduced. This has been done several times in the literature. Roughly speaking, bound sets are *sets* of bounds. Indeed, due to the partial nature of the ordering relation between solutions, the use of a set of bounds instead of a single bound makes it possible to more tightly approximate the image set of the solutions in the objective space. To our knowledge, one of the first work mentioning that notion was performed by Villareal and Karwan [1981], and deals with branch and bound for multiobjective integer linear programming problems. Although the definition they proposed is general, no operational way to compute bound sets has

---

This work has been supported by ANR project GUEPARD (NT09_475088).
Authors' address: LIP6-CNRS, UPMC, 4 Place Jussieu, F-75252 Paris Cedex 05, France.
This paper is an extended version of "Using bound sets in multiobjective optimization: Application to the biobjective binary knapsack problem", published in SEA 2010.

been devised until recently where the bound set does not reduce to a singleton. In this concern, based on the convex hull of the image of the solutions in the objective space, new bound sets have been proposed that invole an infinite number of points (vectors) [Ehrgott and Gandibleux 2007; Sourd and Spanjaard 2008]. A multiobjective *branch and bound* making use of these new bound sets has proved very efficient for the biobjective spanning tree problem [Sourd and Spanjaard 2008]. This approach indeed enables to increase by an order of magnitude (10 to 1) the size of the instances that can be solved in a reasonable amount of time. However, no other attempt has yet been undertaken to investigate the impact of these new bound sets in other multiobjective optimization settings (i.e., other multiobjective optimization problems and/or procedures).

The present paper is a first step in this direction: we investigate the use of such bound sets in a hybrid *dynamic programming* procedure for the biobjective binary *knapsack problem*. The hybridization we propose is in the spirit of the dominance relations between states used in works by Bazgan *et al.* [2007; 2009], but enables huge savings in memory requirements as well as improvements in execution times thanks to the involved bound sets we use.

Two improvements of our method are also studied in the paper, both improvements reinforcing the impact of using involved bound sets to prune the search. A first direction of improvement is the embedding of our method in a two phases approach [Ulungu and Teghem 1995]. In a two phases method, one first computes a subset of the Pareto set so as to identify a subspace of the objective space to which all Pareto points belong. We will show that this shrinking of the objective space further improves the pruning power of the bound sets. A second direction of improvement is a preprocessing by *shaving* that is all the more efficient that the bound sets are accurate. This preprocessing procedure does indeed an intensive use of the bound sets in order to reduce the size of the problem before launching the solution phase itself. The main principles of shaving have been introduced by Martin and Shmoys [1996] in a single objective optimization setting. It has been recently proved to be also interesting in a multiobjective optimization setting [Sourd and Spanjaard 2008]. We show here that the impact of shaving becomes even greater when embedded in a two phases method.

To summarize, the contribution of the paper is twofold:

(1) we first explain how to hybridize multiobjective dynamic programming with the fathoming criterion provided by the bound sets;
(2) then we detail how hybrid multiobjective dynamic programming can be embedded in a two phases approach to further improve the method.

The paper is organized as follows. After recalling some preliminary definitions, and stating the biobjective binary knapsack problem studied in the paper (Section 2), we present a general framework for hybridizing dynamic programming using bound sets in a multiobjective setting (Section 3). Next, we show how to embed it in a two phases method (Section 4). Then, after briefly describing related works on the problem, we specify our resolution method for solving the biobjective binary knapsack problem (Section 5). Finally, we provide numerical results that show the interest of the approach (Section 6).

2. PRELIMINARIES

2.1. Preliminary Definitions

We first recall some preliminary definitions concerning MOCO problems. They differ from the standard single objective ones mainly in their cost structure, as solutions are valued by m-vectors instead of scalars. Let us denote by $\mathcal{X}$ the set of feasible solutions,

and by $\mathcal{Y}$ its image in the objective space. The image of solution $x \in \mathcal{X}$ is $f(x) = (f_1(x), \ldots, f_m(x))$. Comparing solutions in $\mathcal{X}$ amounts then to comparing m-vectors in $\mathcal{Y}$. In this framework, the following notions prove useful (in a maximisation setting):

DEFINITION 2.1. *The* weak dominance *relation on m-vectors of $\mathbb{Z}_+^m$ is defined, for all $y, y' \in \mathbb{Z}_+^m$, by $y \succcurlyeq y' \iff [\forall i \in \{1, \ldots, m\}, y_i \geq y'_i)]$. The* dominance *relation is defined as the asymmetric part of $\succcurlyeq$: $y \succ y' \iff [y \succcurlyeq y'$ and $y' \not\succcurlyeq y]$.*

DEFINITION 2.2. *Within a set $Y \subseteq \mathcal{Y}$, an element y is said to be* dominated *(resp.* weakly dominated*) if $y' \succ y$ (resp. $y' \succcurlyeq y$) for some y' in Y, and* non-dominated *if there is no y' in Y such that $y' \succ y$. The set of non-dominated elements in Y is denoted by $Y^\star$.*

By abuse of language, when $f(x) \succ f(x')$, we say that solution x *dominates* solution x'. Similarly, we use the term of *non-dominated* solutions. The set of non-dominated solutions of $X \subseteq \mathcal{X}$ is denoted by $X^\star$. Following Bazgan *et al.* [2007; 2009], we say that a set of non-dominated solutions is *reduced* if it contains one and only one solution for each non-dominated objective vector in $Y = f(X) = \{f(x) : x \in X\}$ (also named minimal complete set [Hansen 1980]). The non-dominated solutions that maximise a weighted sum of the objectives are called *supported*. A supported solution is called *extreme* if its image is a non-dominated vertex of the convex hull of $Y^\star$.

2.2. Biobjective Binary Knapsack Problem

An instance of the multiobjective binary knapsack problem (0-1 MOKP) consists of a knapsack of integer capacity c, and a set of items $N = \{1 \ldots n\}$. Each item j has a weight w^j and a m-vector profit $p^j = (p_1^j, \ldots, p_m^j)$, variables w^j, p_k^j ($k \in \{1, \ldots, m\}$) being integers. A solution is characterized by a binary n-vector x, where $x_j = 1$ if item j is selected. Furthermore, a solution x is feasible if it satisfies the constraint $\sum_{j=1}^{n} w^j x_j \leq c$. The goal of the problem is to find a reduced set of non-dominated feasible solutions, which can be formally stated as follows:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{j=1}^{n} p_k^j x_j & k \in \{1 \ldots m\} \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} w^j x_j \leq c & \\ & x_j \in \{0,1\} & j \in \{1, \ldots, n\} \end{array}$$

Note that several works in the literature aim at determining *all* Pareto optimal solutions. In our opinion, this goal does not coincide with the one studied in single objective optimization, where one looks for *one* optimal solution, and not *all* optimal solutions. We will focus in this paper on the biobjective binary knapsack problem (0-1 BOKP), where $m = 2$, and from now on we characterize an item j by a triple (p_1^j, p_2^j, w^j).

EXAMPLE 2.3. *Consider the following problem:*

$$\begin{array}{ll} \textit{maximize} & \begin{cases} 10x_1 + 2x_2 + 6x_3 + 9x_4 + 12x_5 + x_6 \\ 2x_1 + 7x_2 + 6x_3 + 4x_4 + x_5 + 3x_6 \end{cases} \\ \textit{subject to} & 4x_1 + 4x_2 + 5x_3 + 4x_4 + 3x_5 + 2x_6 \leq 6 \\ & x_j \in \{0,1\} \;\; j \in \{1, \ldots, 6\} \end{array}$$

There are ten feasible solutions, four of which are non-dominated, and their image set in the objective space is $\mathcal{Y}^\star = \{(13,4), (11,5), (10,7), (3,10)\}$.

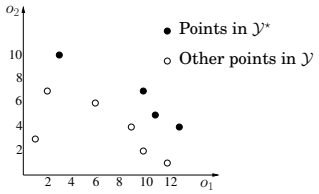

Fig. 1. Objective space in Example 2.3.

3. HYBRID MULTIOBJECTIVE DYNAMIC PROGRAMMING PROCEDURE

3.1. Multiobjective Dynamic Programming

Before hybridizing multiobjective dynamic programming using bound sets, we first recall the standard multiobjective dynamic programming framework. Let there be an integer $n \in \mathbb{N}$. As usual in dynamic programming, we assume that the problem we wish to solve can be broken into $n+1$ periods 0,...,n. We consider for each period j a set S_j representing all the possible states at the end of period j. Without loss of generality, we further assume that there is a unique initial state s_0 and that $S_j \cap S_k = \emptyset$ (if $j \neq k$). We will express by $S = \bigcup_{j=0}^{n} S_j$ the overall set of states. Furthermore, we denote by $A \subseteq S \times S$ the set of possible transitions between states. Note that $s \in S_j$ and $(s, s') \in A$ implies that $s' \in S_{j+1}$. An m-vector $v(s, s')$ is attached to each transition $(s, s') \in A$, and the m-vector assigned to a *policy* (i.e., a sequence of transitions) $\delta = (s_j, \ldots, s_k)$ is $v(\delta) = \sum_{i=j}^{k-1} v(s_i, s_{i+1})$. We will express by $\Delta(s)$ the set of feasible policies from state s_0 to state s, and by $\Delta(S_j)$ the set $\bigcup_{s \in S_j} \Delta(s)$. A multiobjective dynamic programming procedure aims at determining a reduced set $\Delta^\star(S_n)$ of non-dominated policies from s_0 to S_n. Unlike the scalar case, there possibly exist several non-dominated policies with distinct m-vectors to reach a given state in a multiobjective setting. Hence, one keeps a reduced set $\Delta^\star(s)$ of non-dominated policies at each state s, instead of a single optimal one. The multiobjective dynamic programming procedure is formally stated in Algorithm 1, where (δ, s_j) denotes, for $\delta = (s_i, \ldots, s_{j-1})$, policy $(s_i, \ldots, s_j)$, and notation RND$(\cdot)$ stands for a set function returning a reduced set of non-dominated elements. $S'_j \subseteq S_j$ represents the subset of states that one can obtain starting from initial state s_0 (it is therefore the set of states $s_j \in S_j$ such that $\Delta(s_j) \neq \emptyset$).

ALGORITHM 1: Multiobjective Dynamic Programming

1 $S'_0 \leftarrow \{s_0\}$; $\Delta^\star(s_0) \leftarrow \{()\}$ (where $()$ is the empty policy)
2 **for** $j = 1, \ldots, n$ **do**
2a compute $S'_j \leftarrow \{s_j : s_{j-1} \in S'_{j-1}$ and $(s_{j-1}, s_j) \in A\}$
2b **for** each $s_j \in S'_j$ **do**
  compute $\Delta^\star(s_j) \leftarrow$ RND$(\{(\delta, s_j) : \delta \in \Delta^\star(s_{j-1}), (s_{j-1}, s_j) \in A\})$
3 **return** RND$(\bigcup_{s_n \in S'_n} \Delta^\star(s_n))$

For solving the biobjective binary knapsack problem, we will use the following (well-known) dynamic programming formulation:

- a period corresponds to an item j;

- a state $(j, w) \in S_j$ ($j \in \{1, \ldots, n\}$) corresponds to the subsets of $\{1, \ldots, j\}$ of weight w;

- a transition from state (j, w) to state $(j+1, w')$ is possible if $w + w^{j+1} = w'$ or $w = w'$ (assuming $\max\{w, w'\} <= c$): in other words, a transition corresponds to the decision whether to take item $j+1$ or not;

- a policy δ can be seen as a (partial) instanciation of binary vector x, denoted by $\delta = \{x_j \leftarrow 0 \text{ or } 1, \ldots, x_k \leftarrow 0 \text{ or } 1\}$.

3.2. Bound Sets in MOCO Problems

It is well known that the computer storage requirements are an important issue in dynamic programming approaches. This is even more true in MOCO problems since one does not store a single subpolicy at each step, but a set of Pareto optimal subpolicies. To overcome this difficulty, a simple idea is the following: given an incumbent set of already known complete policies (computed by a heuristic), if a bound can prove that a subpolicy cannot be completed in a policy improving the incumbent set, then it is not necessary to store this policy (similarly to what is done in the single objective branch and bound strategy for dynamic programming [Morin and Marsten 1976]). In a multiobjective optimization setting, since one handles sets of m-vectors, the very notion of upper and lower bound has to be revisited. This work has been undertaken by Villareal and Karwan [1981]. They introduced the notion of *bound sets* (in the terminology of Ehrgott and Gandibleux [2007]). Since the formalism used here slightly differs from the one presented in these works, we give below our own definitions of upper and lower bound sets.

3.2.1. Upper Bound Set. The simplest idea that comes to mind to upper bound a set Y of vectors is to define a single vector y^I such that $y^I_i = \max_{y \in Y} y_i$ for $i = 1, \ldots, m$. This point is called the *ideal point* of Y. However, this ideal point is usually very "far" from the points in Y. For this reason, it is useful to define an upper bound from a *set* of vectors instead of a singleton. Such a set is then called an *upper bound set* [Ehrgott and Gandibleux 2007].

DEFINITION 3.1 (UPPER BOUND SET). *A set* $\mathbf{UB}$ *is an upper bound set of* Y *if* $\forall y \in Y,\ \exists u \in \mathbf{UB} : u \succcurlyeq y$.

This is compatible with the definition of an upper bound in the single objective case ($\mathbf{UB}$ reduces then to a singleton). As previously indicated, the upper bound set defined by $\mathbf{UB} = \{y^I\}$ is poor. In practice, a general family of good upper bound sets of Y can be defined as $\mathbf{UB}_\Lambda = \bigcap_{\lambda \in \Lambda} \{u \in \mathbb{R}^m : \langle \lambda, u \rangle \leq \mathbf{UB}_\lambda\}$, where the $\lambda \in \Lambda$ are weight vectors of the form $(\lambda_1, \ldots, \lambda_m) \geq 0$, $\langle ., . \rangle$ denotes the scalar product, and $\mathbf{UB}_\lambda \in \mathbb{R}$ is an upper bound for $\{\langle \lambda, y \rangle : y \in Y\}$. Clearly, the best upper bound set in this family is obtained for $\Lambda = \Lambda_c(Y)$ where $\Lambda_c(Y)$ characterizes the facets of the non-dominated boundary of the convex hull of Y. Interestingly, we will see in the following that this boundary can be efficiently computed in the biobjective case, provided $\max_{y \in Y} \langle \lambda, y \rangle$ can be determined within polynomial or pseudo-polynomial time.

3.2.2. Lower Bound Set. Similarly to the upper bound set, the simplest idea that comes to mind to lower bound a set Y of vectors is to define a single vector y^A such that $y^A_i = \min_{y \in Y} y_i$ for $i = 1, \ldots, m$. This point is called the *anti-ideal point* of Y. Here again, taking several points simultaneously into account in the lower bound enables to more tightly bound set Y. Such a set is then called a *lower bound set* [Ehrgott and Gandibleux 2007].

DEFINITION 3.2 (LOWER BOUND SET). *A set* $\mathbf{LB}$ *is a lower bound set of* Y *if* $\forall y \in Y,\ \exists l \in \mathbf{LB} : y \succcurlyeq l$.

As above, the compatibility with the single objective case holds. In the biobjective case, when Y only includes mutually non-dominated points, we will show in the next subsection a way to refine the lower bound set defined by $\mathbf{LB} = \{y^A\}$.

3.2.3. Comparing Bound Sets in Multiobjective Dynamic Programming. Hybridization is about eliminating subpolicies at step $2b$ of the multiobjective dynamic programming procedure by using bound sets. In order to perform the elimination, we need to evaluate if a subpolicy $\delta \in \Delta^\star(s_j)$ ($s_j \in S_j$) can be *extended* in a non-dominated complete policy. Let us denote by $\mathrm{Ext}(\delta)$ the set of potential *extensions* of a subpolicy $\delta \in \Delta^\star(s_j)$ ($s_j \in S_j$), i.e. feasible policies in $\Delta(S_n)$ (i.e., from state s_0 to a state in S_n) whose subpolicy from s_0 to S_j is δ. To do this, one compares an upper bound set $\mathbf{UB}$ of $f(\mathrm{Ext}(\delta))$ and a lower bound set $\mathbf{LB}$ of $f(\Delta^\star(S_n))$. These sets are computed in the course of the dynamic programming procedure (the details are given in the next subsection). Unlike the single objective case, the comparison is not trivial since one handles sets instead of scalars. We introduce here two notions that make it possible to simply define this operation in a multiobjective setting.

DEFINITION 3.3 (UPPER AND LOWER RELAXATIONS). *Given an upper bound set* $\mathbf{UB}$*, the upper relaxation* $\mathbf{UB}^{\preccurlyeq}$ *is defined as:* $\mathbf{UB}^{\preccurlyeq} = \{y \in \mathbb{R}^m_+, \exists u \in \mathbf{UB},\ u \succcurlyeq y\}$. *Similarly, given a lower bound set* $\mathbf{LB}$*, the lower relaxation* $\mathbf{LB}^{\succcurlyeq}$ *is defined as:* $\mathbf{LB}^{\succcurlyeq} = \{y \in \mathbb{R}^m_+, \exists l \in \mathbf{LB},\ y \succcurlyeq l\}$.

For the convenience of the reader, the different notions introduced so far are illustrated in Figure 2.

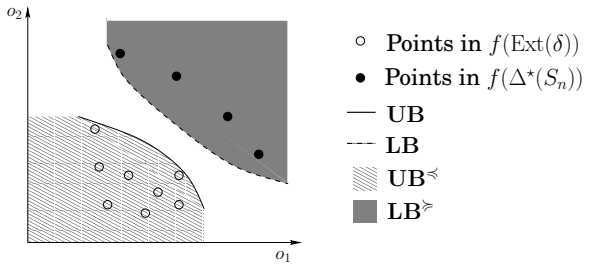

Fig. 2. Comparing bound sets.

Coming back to the comparison of $\mathbf{UB}$ and $\mathbf{LB}$, it is clear that $\mathbf{UB}^{\preccurlyeq} \supseteq f(\mathrm{Ext}(\delta))$ and $\mathbf{LB}^{\succcurlyeq} \supseteq f(\Delta^\star(S_n))$. Consequently, $\mathbf{UB}^{\preccurlyeq} \cap \mathbf{LB}^{\succcurlyeq} = \emptyset$ implies that $f(\mathrm{Ext}(\delta)) \cap f(\Delta^\star(S_n)) = \emptyset$. In this case, subpolicy δ can of course be safely pruned. In the hybrid version of multiobjective dynamic programming, step 2b of Algorithm 1 is thus replaced by:

2b **for** each $s_j \in S'_j$ **do**
  compute $\Delta^\star(s_j) \leftarrow \mathrm{RND}(\{(\delta, s_j) : \delta \in \Delta^\star(s_{j-1}), (s_{j-1}, s_j) \in A\})$
  for each $\delta \in \Delta^\star(s_j)$ **do**
    if $(\mathbf{UB}^{\preccurlyeq}_\delta \cap \mathbf{LB}^{\succcurlyeq} = \emptyset)$ **then** $\Delta^\star(s_j) \leftarrow \Delta^\star(s_j) \setminus \delta$

where $\mathbf{UB}^{\preccurlyeq}_\delta$ denotes an upper relaxation of $f(\mathrm{Ext}(\delta))$.

Note that this pruning condition can be refined by using the fact that one only looks for a reduced set of non-dominated solutions as well as the fact that valuations are integers. For simplicity, this refinement is not detailed here. The main point is now to be able to efficiently compute good lower and upper bound sets. In the following subsection, this issue will be answered for the biobjective case.

3.3. Hybrid Biobjective Dynamic Programming

We now detail the algorithms used in hybrid *biobjective* dynamic programming to compute the bound sets and perform their comparison.

Computation of an Upper Bound Set. Given a subpolicy $\delta \in \Delta(S_j)$, upper bound set $\mathbf{UB}_{\Lambda_c(f(\mathrm{Ext}(\delta)))}$ can be compactly represented by storing the *extreme points* of $Y = f(\mathrm{Ext}(\delta))$, i.e. the vertices of the non-dominated boundary of the convex hull of Y (points y^1, y^2, y^3 and y^4 in the left part of Figure 3). Note that, for readibility, notation $\mathbf{UB}_{\Lambda_c(f(\mathrm{Ext}(\delta)))}$ is replaced by $\mathbf{UB}_{\Lambda_c}(\delta)$ in the sequel. Aneja and Nair's method [1979] enables to efficiently compute these vertices in biobjective combinatorial problems whose single objective version is solvable within polynomial or pseudo-polynomial time. In order for the paper to be self-contained, we explain this method shortly. First, the two lexicographically optimal points (a point is lexicographically optimal if it achieves the best value on the second (resp. first) objective among points achieving the best value on the first (resp. second) objective) are computed by resorting to a single objective optimization algorithm. Second, the list L of extreme points is initialized with the two obtained points, and maintained in increasing order with respect to the first objective. In the biobjective binary knapsack problem, list L is then computed recursively as follows: given two consecutive points $y^1 = (y^1_1, y^1_2)$ and $y^2 = (y^2_1, y^2_2)$ in L, a new point is computed by solving a standard binary knapsack problem after scalarizing the vector valuations (p^j_1, p^j_2) of each item j by a weighted sum $\lambda_1 p^j_1 + \lambda_2 p^j_2$, with $\lambda_1 = y^2_2 - y^1_2$ and $\lambda_2 = y^1_1 - y^2_1$. The number of times the single objective solution method is launched is of course linear in the number of extreme points.

EXAMPLE 3.4. *Let us come back to Example 2.3. Assume that one wants to upper bound set $f(\mathrm{Ext}(\delta))$ where δ denotes the partial instanciation $\{x_6 \leftarrow 0\}$. Aneja and Nair's method yields the following list L of extreme points, characterizing $\mathbf{UB}_{\Lambda_c}(\delta)$: $L = ((12,1),(9,4),(6,6),(2,7))$. The corresponding upper relaxation $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)$ is represented in the left part of Figure 3.*

Computation of a Lower Bound Set. Given a subset $I \subseteq f(\Delta(S_n))$ (the incumbent set in the hybrid dynamic programming procedure), a tight lower bound set $\mathbf{LB}$ of $I^\star$ can be computed as follows. When there are two objectives and $\{(i^j_1, i^j_2) : 1 \leq j \leq k\}$ are the points of $I^\star$ maintained in lexicographical order (i.e., in decreasing order of the first objective, and increasing order of the second one), one can set $\mathbf{LB}_{\mathcal{N}(I)} = \{n^j = (i^{(j+1)}_1, i^j_2) : 0 \leq j \leq k\}$, where $i^0_2 = 0$ and $i^{(k+1)}_1 = 0$. Note that, if i^1_1 (resp. i^k_2) is the optimal value on the first component (resp. second component), the definition of $\mathbf{LB}_{\mathcal{N}(I)}$ can be restricted to $\{(i^{(j+1)}_1, j^i_2) : 1 \leq j \leq k-1\}$ since there cannot be any feasible

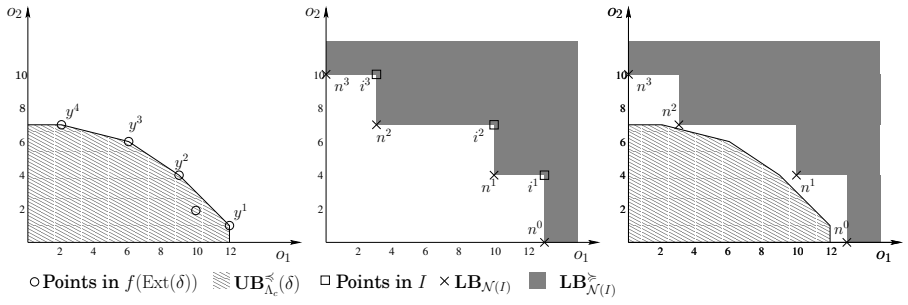

Fig. 3. Upper and lower bound sets in a biobjective setting.

solution with a value strictly greater than i_1^1 on the first component, nor a solution of value strictly greater than i_2^k on the second component. It typically occurs when the determination of the lexicographically optimal solutions can be done efficiently. The set $\mathbf{LB}_{\mathcal{N}(I)}$ can here be viewed as a generalization of the nadir point of I (whose components are the worst possible values among the points of $I^\star$). The points in $\mathbf{LB}_{\mathcal{N}(I)}$ are therefore sometimes called *local nadir points* [Ehrgott and Gandibleux 2007]. One can note that $\mathbf{LB}_{\mathcal{N}(I)}$ is also a lower bound set for $f(\Delta^\star(S_n))$.

EXAMPLE 3.5. *Let us come back to Example 2.3 once again, and consider the following subset of points in* $f(\Delta(S_n))$*:* $I = \{(13,4),(10,7),(3,10)\}$*. The lower bound set is then:* $\mathcal{N}(I) = \{(13,0),(10,4),(3,7),(0,10)\}$*. This lower bound set is represented in the middle part of Figure 3, as well as its lower relaxation* $\mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)}$.

As described above, to know if one can prune a subpolicy δ, one must compute the intersection of the relaxations of a lower bound set of $f(\Delta^\star(S_n))$ and an upper bound set of $f(\mathrm{Ext}(\delta))$. Testing if $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$ amounts to check that no element of $\mathbf{LB}_{\mathcal{N}(I)}$ is included in $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)$. It can be formally expressed by:

$$\forall n \in \mathbf{LB}_{\mathcal{N}(I)}, \exists \lambda \in \Lambda_c(f(\mathrm{Ext}(\delta))) : \lambda_1 n_1 + \lambda_2 n_2 > \max_{y \in \mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)} (\lambda_1 y_1 + \lambda_2 y_2)$$

EXAMPLE 3.6. *Continuing Example 3.4 and Example 3.5, we shall compare the two obtained relaxations. Both sets are represented in the right part of Figure 3. Their intersection is empty, meaning that subpolicy* δ *can be safely discarded.*

Hybrid Biobjective Dynamic Programming Procedure. The hybrid multiobjective dynamic procedure is summarized in Algorithm 2. Notation $\mathrm{ND}(\cdot)$ stands for a set function returning the subset of non-dominated points in a set of m-vectors, I denotes the *incumbent set* of non-dominated complete policies among the ones that have already been generated, and $\mathrm{Ext}_{\Lambda_c}(\delta)$ denotes the extreme solutions among the extensions of δ. For simplicity we return here the images in the objective space of the non-dominated policies instead of the non-dominated policies themselves. Note that the policies themselves could be obtained easily by using standard bookkeeping techniques. Furthermore, one takes advantage of the fact that the computation of $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)$ yields feasible complete policies, possibly non-dominated, by updating set I if policy δ is not discarded.

ALGORITHM 2: Hybrid Multiobjective Dynamic Programming

1 $S'_0 \leftarrow \{s_0\}$; $\Delta^\star(s_0) \leftarrow \{()\}$; $I \leftarrow \{(0,\ldots,0)\}$
2 **for** $j = 1,\ldots,n$ **do**
2a compute $S'_j \leftarrow \{s_j : s_{j-1} \in S'_{j-1}$ and $(s_{j-1}, s_j) \in A\}$
2b **for** each $s_j \in S'_j$ **do**
  compute $\Delta^\star(s_j) \leftarrow \mathrm{RND}(\{(\delta, s_j) : \delta \in \Delta^\star(s_{j-1}), (s_{j-1}, s_j) \in A\})$
  for each $\delta \in \Delta^\star(s_j)$ **do**
  if $(\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset)$ **then** $\Delta^\star(s_j) \leftarrow \Delta^\star(s_j) \setminus \delta$
  else $I \leftarrow \mathrm{ND}(f(\mathrm{Ext}_{\Lambda_c}(\delta)) \cup I)$
3 **return** I

4. THE TWO PHASES METHOD

The two phases method has been introduced by Ulungu and Teghem [1995]. The idea of this method is to divide the problem into two different tasks: first, finding extreme solutions (the first phase), then, finding the non-extreme solutions (the second phase).

This method best applies to problems whose associated single objective problems are efficiently solved, since the first phase heavily depends on the resolution of single objective problems. This is the case of knapsack problems, as well as assignment problems, spanning tree problems, etc. In biobjective problems, one can easily reduce the search space of the second phase by using solutions found in the first one: the triangles generated by two successive extreme points p^1, p^2 and their *local nadir point* (i.e. the point $(\min\{p_1^1, p_2^1\}; \min\{p_1^2, p_2^2\})$) are the only places where new non-dominated points can be found, as illustrated in the following example.

EXAMPLE 4.1. *The triangles that would be obtained in the problem described in Example 2.3 are represented in Figure 4. In a two phases method, the extreme points (in black) would be found during the first phase, and the other non-dominated points (in grey) would be found during the second phase.*

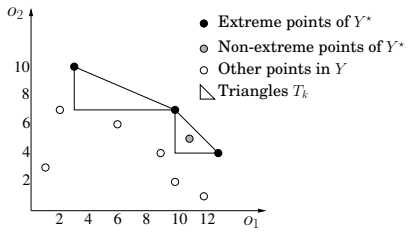

Fig. 4. Two phases method.

The second phase consists then in thoroughly examining each triangle using an enumerative method such as a branch and bound procedure or a ranking method. We call *ranking method* an algorithm able to enumerate the best solutions of a single objective optimization problem in order, until a stopping condition is fulfilled (the simplest condition is for instance to stop at a given rank k) : this makes it possible to explore a triangle by using the ranking method for a convenient weighted sum of the objectives (the one for which the hypotenuse of the triangle is an isoquant). Currently, the fastest algorithm solving the biobjective assignment problem [Przybylski et al. 2008] uses a two phases method with a ranking method for the second phase. A similar method has also been used for the biobjective spanning tree problem [Steiner and Radzik 2008]. Concerning the biobjective binary knapsack problem, a two phases method has already been presented [Visée et al. 1998], using a branch and bound in the second phase, but other approaches have since been proposed that outperform the two phases method: a labeling approach developed by Captivo et al. [2003], and a dynamic programming approach by Bazgan et al. [2009]. A more detailed description of these works is given below (in Section 5).

We propose here a two phases version of our hybrid dynamic programming procedure : instead of applying one single hybrid dynamic programming procedure directly on the problem instance at hand, one first computes the extreme solutions of the problem (the first phase of the method), and then applies the hybrid procedure for each triangle between two consecutive extreme points (the second phase). By subdividing the problem in this way, the fathoming criterion is more efficient. Indeed, let us denote by $T_1, \ldots, T_t$ the triangles to explore after the first phase has been performed. When examining a triangle T_k, for testing whether a subpolicy can be discarded using the fathoming criterion, computing the entire upper relaxation is not needed, only the part of the relaxation in the triangle is considered. More precisely, in step 2b of Algorithm 2, it amounts to replace test $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$ by test $(\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap T_k) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$.

In large problems, entire upper relaxation $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)$ may involve hundreds of vertices, while for a given triangle T_k, local relaxation $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap T_k$ only involves a few vertices. The computation of this latter relaxation is therefore much faster. Furthermore, note that incumbent set I is of course not reset during the second phase. The two phases version of our hybrid dynamic programming procedure is synthesized in Algorithm 3.

ALGORITHM 3: Two Phases Hybrid Biobjective Dynamic Programming

First phase
1 compute extreme points $\{y^1, \ldots, y^{t+1}\}$
2 $I \leftarrow \{y^1, \ldots, y^{t+1}\}$
3 **for** $k = 1, \ldots, t$ **do**
3a generate triangles T_k between y^k and y^{k+1}
Second phase
4 **for** $k = 1, \ldots, t$ **do**
4a launch Algorithm 2 to explore triangle T_k and update I accordingly
return I

For the convenience of the reader, we now provide an example of step 4a to illustrate the interest of the modified fathoming criterion.

EXAMPLE 4.2. *Suppose that triangle T_k (as represented in Figure 5) is being examined. Note that it is sufficient to determine the three encircled vertices of $\mathbf{UB}_{\Lambda_c}(\delta)$ to identify $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap T_k$ (see Figure 5). In this example, policy δ is discarded, because $(\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap T_k) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$. Note that $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} \neq \emptyset$ (due to the area enclosed in bold on the bottom right of Figure 5), so δ would not have been discarded without a two phases approach.*

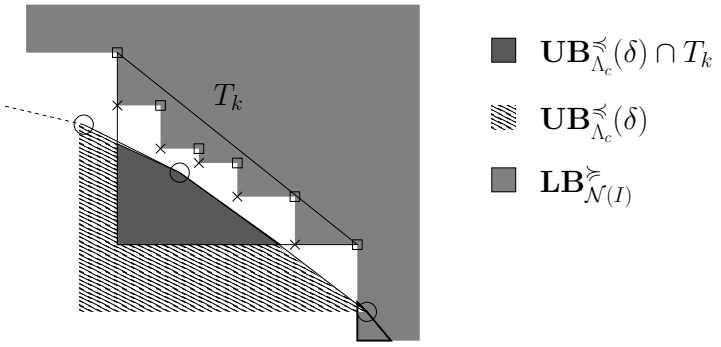

Fig. 5. Objective space in Example 4.2.

5. APPLICATION TO THE BIOBJECTIVE BINARY KNAPSACK PROBLEM

5.1. Related Works

Several methods have been developed in the previous years to solve 0-1 BOKP. We present here a brief state-of-the-art on exact methods. Note that, of course, there exist also approximation algorithms (with performance guarantee) and heuristic methods (genetic algorithms, tabu search...) for solving 0-1 BOKP. For a recent and more broad

survey on the multiobjective knapsack problem, the interested reader can refer to the article of Lust and Teghem [2010].

Among exact algorithms proposed for solving 0-1 BOKP, one can mention two phases methods, dynamic programming procedures and more specific problem-dependent methods. Visée et al. [1998] introduced a two phases method to solve the biobjective binary knapsack problem. They first calculate the set of extreme optimal solutions, then they compute the set of non-extreme Pareto optimal solutions located in the triangles generated in the objective space by two successive extreme solutions, by resorting to branch and bound procedures (one for each triangle). This method is memory consuming, and only small-sized instances can be solved. A labelling approach by Captivo et al. [2003] improved the resolution of this problem, enabling larger instances to be solved in a decent time, the main limitation being again the memory requirements. To take advantage of the particular structure of network problems, the biobjective knapsack model is transformed into a biobjective shortest path problem over an acyclic network. This makes it possible to use a labelling algorithm to search for all the non-dominated solutions. A recent work, done by Bazgan et al. [2009], presents a new approach based on dynamic programming. It relies on the use of several complementary dominance relations to discard partial solutions that cannot lead to new non-dominated solutions. In their paper, the most efficient dominance relation between two partial instanciations δ and δ' is based on the comparison between, on the one hand, an ideal point $y^I(\delta)$ obtained from δ (resp. δ') by computing an upper bound on each objective, and, on the other hand, a heuristic solution $x^h(\delta')$ obtained from δ' (resp. δ) by performing a greedy completion of the partial instanciation. If $f(x^h(\delta')) \succ y^I(\delta)$, then δ is dominated by δ' and can be safely discarded. The paper also features a comparison between this new approach, the labelling approach and an ε-constraint approach [Ehrgott 2005], showing the large enhancement in terms of both memory and computation time brought by the method introduced in the article. This is currently the most efficient dynamic programming method known for solving the biobjective binary knapsack problem, but it still needs a large amount of memory to stock all the non-dominated partial solutions, and the largest instances solved may require tens of millions of solutions to be stocked in order to find all non-dominated solutions. Our method will, all the while solving the problem faster, settle the memory issue.

5.2. Our Algorithm

Before describing our algorithm in detail, we would like to simply recall what is its aim: given a set of items, and a knapsack of a given capacity, find all the possible packings of the items in the knapsack (actually a reduced set) that have a total profit that is non-dominated by any other packing.

5.2.1. Initialization of the Algorithm. The initialization of our algorithm consists in reducing the size of the instance by using a preprocessing procedure called *shaving*, and in initializing the incumbent set I by heuristically computing good solutions for the problem. For 0-1 BOKP, the procedure we use works as follows: after initially setting $I = \{(0,0)\}$, for each item j two subproblems are created, one where item j is made mandatory (type M), and one where item j is made forbidden (type F). For each subproblem the upper relaxation $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)$ – where δ denotes the partial instanciation $\{x_j \leftarrow 0\}$ or $\{x_j \leftarrow 1\}$ – is computed, and compared to the current lower relaxation $\mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)}$. If some extreme points of $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)$ are non-dominated, they are inserted into I, and the possible dominated elements in I are removed. If $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$, the subproblem grants no non-dominated solutions for the problem, and thus item j can be excluded from the problem, by permanently setting $x_j = 0$ (resp. $x_j = 1$) if

the subproblem is of type F (resp. type M). This part of the algorithm is useful for at least two reasons: first it allows us to have a good incumbent set before starting the dynamic programming procedure, thus decreasing the number of subsolutions propagated, and second, depending on the instance, it can remove a lot of items, which is also a good means to reduce the computation time. For the convenience of the reader, we now provide an illustrative example.

EXAMPLE 5.1. *Consider the following biobjective binary knapsack problem:*

$$\begin{aligned} \textit{maximize} \quad & \sum_{j=1}^{5} p_k^j x_j \qquad k \in \{1,2\} \\ \textit{subject to} \quad & \sum_{j=1}^{5} w^j x_j \leq 5 \\ & x_j \in \{0,1\} \qquad j \in \{1,\ldots,5\} \end{aligned}$$

The set of items is $N = \{(10,10,1),(1,1,10),(3,9,3),(9,4,3),(5,5,2)\}$. *Initially, one sets* $I = \{(0,0)\}$. *The shaving process works as follows. One first computes the extreme points of* $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\{x_1 \leftarrow 1\})$*:* $\{(19,14),(13,19)\}$. *The incumbent set* I *is therefore updated:* $I = \{(19,14),(13,19)\}$. *One then computes the extreme points of* $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\{x_1 \leftarrow 0\})$*:* $\{(14,9),(8,14)\}$. *Since* $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\{x_1 \leftarrow 0\}) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$*, item* $(10,10,1)$ *is made permanently mandatory. One proceeds with the second item, and detects that* $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\{x_2 \leftarrow 1\}) = \emptyset$ *(no feasible solution includes the second item due to its weight). Consequently* $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\{x_2 \leftarrow 1\}) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} = \emptyset$*, and item* $(1,1,10)$ *is made permanently forbidden. Nothing happens when examining the third item (i.e., item* $(3,9,3)$ *is made neither mandatory nor forbidden). Next, when examining the fourth item one gets a single extreme point for* $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\{x_4 \leftarrow 1\})$*:* $\{(15,15)\}$. *The incumbent set is once again updated:* $I = \{(19,14),(15,15),(13,19)\}$ *(and item* $(9,4,3)$ *is made neither mandatory nor forbidden). Nothing happens when examining the fifth item. To summarize, the initial knapsack problem can be transformed into the following reduced problem:*

$$\begin{aligned} \textit{maximize} \quad & 10 + \sum_{j=1}^{3} p_k^j x_j \quad k \in \{1,2\} \\ \textit{subject to} \quad & \sum_{j=1}^{3} w^j x_j \leq 4 \\ & x_j \in \{0,1\} \qquad j \in \{1,\ldots,3\} \end{aligned}$$

where the set of items is now: $N = \{(3,9,3),(9,4,3),(5,5,2)\}$. *Moreover, the incumbent set* $I = \{(19,14),(15,15),(13,19)\}$ *has been initialized in a good way (in this example, all non-dominated solutions are in I).*

The shaving procedure is actually performed several times: once during the initialization of the algorithm, and once before examining each triangle (the shaving procedure is therefore launched once per triangle). In the second kind of shaving, much more items are made mandatory or forbidden since one takes advantage of the fact that one focuses on a very narrow area of the objective space. However the first shaving is still useful since it prevents from eliminating many times the same item.

5.2.2. Implementation of the Search for Extreme Solutions. In our resolution method, the search for extreme solutions is a critical primitive since it is extensively used, both for the first phase of the method and for the determination of upper approximations. The computation of an extreme solution amounts to solve a single objective binary

knapsack problem. This problem has been thoroughly studied, and there are a lot of good algorithms available. The best algorithms currently known for this problem have been compared in a book by Kellerer et al. [2004], where the *minknap* and *combo* algorithms prove to be the quickest[1]. To solve the single objective problems, we used the minknap algorithm [Pisinger 1997]. Combo algorithm [Martello et al. 1999] seems indeed to be a little bit slower for the type of instances we are solving. Most of the time, the single objective problems solved have no correlation between profits and weights, and when there is a correlation, it is only a weak correlation, making minknap a better choice.

5.2.3. Dynamic Programming. Among the dynamic programming approaches proposed for the multiobjective knapsack problem [Klamroth and Wiecek 2000], we adopt here a similar approach to Villareal and Karwan's one [1981], which is itself an extension of the approach of Garfinkel and Nemhauser [1972] to take into account multiple objectives. As indicated in Section 3, the set of states is defined as:

$$S = \{(j, w) : j = 1, \ldots, n;\ w = 0, \ldots, c\}$$

where (j, w) corresponds to the subsets of $\{1, \ldots, j\}$ of weight w (partial instanciations).

Let us denote by $\Delta(j, w) = \{\delta : \sum_{i=1}^{j} w^i \delta(x_i) = w,\ \delta(x_{j+1}) = 0, \ldots, \delta(x_n) = 0\}$ the set of partial instanciations δ at state (j, w), where $\delta(x_i) = 1$ (resp. $\delta(x_i) = 0$) if $(x_i \leftarrow 1) \in \delta$ (resp. $(x_i \leftarrow 0) \in \delta$), and by $Y(j, w)$ its image in the objective space. We show here how to compute $Y^\star(n, w)$ (the non-dominated elements of $Y(n, w)$) for $w \in \{0, \ldots, c\}$. As already mentioned, using standard bookkeeping techniques, it is not difficult to extend the algorithm such that it actually returns a reduced set of non-dominated solutions in $\cup_{w=0}^{c} \Delta(n, w)$. 0-1 BOKP can be solved by applying the following recursive equations:

$$\begin{array}{rcl} Y^\star(0, w) & = & \{(0, 0)\} \quad \text{for } w = 0 \ldots c \\ Y^\star(j, 0) & = & \{(0, 0)\} \quad \text{for } j = 1 \ldots n \\ Y^\star(j, w) & = & ND(Y^\star(j-1, w) \cup (p^j + Y^\star(j-1, w - w^j))) \\ & & \text{for } j = 1 \ldots n,\ w = 1 \ldots c \end{array} \tag{1}$$

where $ND(Y)$ denotes the non-dominated elements of a set Y, and $y + Y^\star(j, w)$ denotes $\{y + y' : y' \in Y^\star(j, w)\}$.

Implementation of the Recursion. For each state (j, w) we first compute the non-dominated subsolutions using Equation 1. We store each set $Y^\star(j, w)$ in a red-black tree enabling us to insert, remove or search an element in $O(\log |Y^\star(j, w)|)$. This allows us to compute $Y^\star(j, w)$ in $O(|Y^\star(j-1, w-w^j)| + |Y^\star(j-1, w)|)$ instead of $O(|Y^\star(j-1, w-w^j)| \times |Y^\star(j-1, w)|)$. To achieve this complexity, it is well known that one only needs the elements of sets $Y^\star(j-1, w)$ and $Y^\star(j-1, w-w^j)$ to be *lexicographically* ordered, i.e. decreasingly according to the first component and increasingly according to the second one. The algorithm to compute $Y^\star(j, w)$ from $Y^\star(j-1, w)$ and $Y^\star(j-1, w-w^j)$ is then simple. After initially setting $Y^\star(j, w) = \emptyset$, we compare the first element of $p^j + Y^\star(j-1, w-w^j)$ and the first element of $Y^\star(j-1, w)$, and we select the maximal one with respect to the lexicographical order (or arbitrarily if both elements are equal). We insert the selected element into $Y^\star(j, w)$ provided it is not dominated by the last element of $Y^\star(j, w)$ (if it exists). We repeat this elementary step with the non-selected elements of both sets until one of the sets contains only selected elements. We insert then into $Y^\star(j, w)$ the non-selected elements of the other set, granted they are non-dominated.

[1]Both algorithms are available at `http://www.diku.dk/hjemmesider/ansatte/pisinger/codes.html`.

EXAMPLE 5.2. *Assume that we want to compute* $Y^\star(j,w)$ *from* $p^j + Y^\star(j-1, w-w_j) = \{(10,1),(8,4)\}$ *and* $Y^\star(j-1,w) = \{(10,2),(6,2),(2,8)\}$. *First, we compare* $(10,1)$ *and* $(10,2)$. *Since* $(10,2)$ *is lexicographically greater than* $(10,1)$*, it is selected and inserted into* $Y^\star(j,w)$ *(which was empty). We now compare* $(10,1)$ *and* $(6,2)$*: element* $(10,1)$ *is selected but not inserted (dominated by* $(10,2)$*). Then,* $(6,2)$ *and* $(8,4)$ *are compared:* $(8,4)$ *is selected and inserted into* $Y^\star(j,w)$*, which becomes* $\{(10,2),(8,4)\}$. *Since all the elements of* $p^j + Y^\star(j-1,w-w_j)$ *have been selected, the non-selected elements of* $Y^\star(j-1,w)$ *are examined and* $(2,8)$ *is inserted into* $Y^\star(j,w)$*, because it is non-dominated. One finally obtains* $Y^\star(j,w) = \{(10,2),(8,4),(2,8)\}$.

Domination Relation between Elements of $Y^\star(j,w)$ *and* $Y^\star(j,w')$. Obviously, an element $y \in Y^\star(j,w)$ can be discarded if there exists an element $y' \in Y^\star(j,w')$ such that $w' < w$ and $y' \succcurlyeq y$ (better profits and smaller weight). To implement this domination relation, one uses a set $Y^\star(j)$ defined by $Y^\star(j) = ND(\cup_{w=0}^{c} Y^\star(j,w))$. In practice $Y^\star(j)$ is incrementally built, simultaneously to sets $Y^\star(j,w)$ $(w = 0,\ldots,c)$. More precisely, once a set $Y^\star(j,w)$ is completed, the elements that are dominated by one of $Y^\star(j)$ are discarded, and $Y^\star(j)$ is updated by using the following relation: $Y^\star(j) = ND(Y^\star(j) \cup Y^\star(j,w))$. We now illustrate this on a small example.

EXAMPLE 5.3. *At some step of the dynamic programming procedure, assume that we have :* $Y^\star(j,w) = \{(4,2),(3,3),(2,6)\}$ *and* $Y^\star(j) = \{(8,1),(5,2),(3,3),(1,9)\}$. *Elements* $(4,2)$ *and* $(3,3)$ *are discarded from* $Y^\star(j,w)$ *because they are weakly dominated by* $(5,2)$ *and* $(3,3)$ *respectively. In other words, there exists subsolutions with lower weights and greater profits. Afterwards set* $Y^\star(j)$ *is updated and becomes* $Y^\star(j) = \{(8,1),(5,2),(3,3),(2,6),(1,9)\}$.

Fathoming Criterion. The domination relation can reduce the number of elements in each set $Y^\star(j,w)$ for a low computational cost. Yet for large instances, this relation is not enough to efficiently limit the number of elements, leading the program to fall short of memory. In this concern, the fathoming criterion introduced in Section 3 makes it possible to considerably reduce the number of stored elements. Finding an upper relaxation $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)$ for a partial instanciation δ at state (j,w) can be done by applying Aneja and Nair's method (see Subsection 3.3) to find the extreme points of the following problem:

$$\begin{array}{ll} \text{maximize} & \displaystyle\sum_{i=j+1}^{n} p_k^i x_i + \sum_{i=1}^{j} p_k^i \delta(x_i) \quad k \in \{1,2\} \\ \text{subject to} & \sum_{i=j+1}^{n} w^i x_i \leq c - w \qquad x_i \in \{0,1\} \end{array}$$

where $\delta(x_i) = 1$ (resp. $\delta(x_i) = 0$) if $(x_i \leftarrow 1) \in \delta$ (resp. $(x_i \leftarrow 0) \in \delta$). Using this formulation enables us to remark that δ only appears in the objective function. To reduce the computational load, we can thus compute once the extreme points of the subproblem on $\{j+1,\ldots,n\}$ with capacity $c-w$, that is denoted by $P_{(j+1,w)}$, and is written:

$$\begin{array}{ll} \text{maximize} & \displaystyle\sum_{i=j+1}^{n} p_k^i x_i \qquad\qquad k \in \{1,2\} \\ \text{subject to} & \sum_{i=j+1}^{n} w^i x_i \leq c - w \quad x_i \in \{0,1\} \end{array}$$

One can then obtain the vertices of $\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta)$ (for a partial instanciation δ) by simply translating the extreme points of $P_{(j+1,w)}$ by vector $(\sum_{i=1}^{j} p_1^i \delta(x_i), \sum_{i=1}^{j} p_2^i \delta(x_i))$. Note that there are still some redundancies in the calculations, because an extreme solution $(x_j,\ldots,x_n)$ of $P_{(j,w)}$ is also an extreme solution of $P_{(j,w')}$ $(w' > w)$ provided $\sum_{i=j}^{n} w^i x_i \leq c - w'$. We take advantage of this property to speed up the computation time of the extreme solutions of $P_{(j,w')}$ knowing the extreme ones of $P_{(j,w)}$. Even when

only a subset of the extreme solutions of $P_{(j,w)}$ are still extreme solutions of $P_{(j,w')}$, we can use the common vertices to build the hull of $P_{(j,w')}$ faster. Indeed, all we have to do is to apply Aneja and Nair's method, not for the whole hull, but just for the *gaps* between vertices that already belonged to $P_{(j,w)}$. By *gap* we mean the interval between two common vertices, where the vertices of $P_{(j,w)}$ do not match with the ones of $P_{(j,w')}$. We now provide an example to illustrate this.

EXAMPLE 5.4. *Consider a biobjective binary knapsack problem with* $c = 30$*. Assume that, for a given* $j \in \{1, \dots, n\}$*, the feasible solutions of subproblem* $P_{(j,10)}$ *(i.e. finding the non-dominated subsets of* $\{j, \dots, n\}$ *with a maximum weight of* $30 - 10 = 20$*) are characterized by the following triples, where the first (resp. second) component denotes the value on the first (resp. second) objective, and the third component denotes the weight:* $S = \{(18, 4, 9), (16, 10, 17), (16, 8, 8), (12, 14, 9), (8, 15, 7), (6, 16, 13), (2, 16, 10)\}$*. The images of these solutions in the objective space are represented in Figure 6. Among these feasible solutions the extreme ones are characterized by:* $\{(18, 4, 9), (16, 10, 17), (12, 14, 9), (6, 16, 13)\}$*. Assume now that we want to use this result to compute the extreme solutions of* $P_{(j,15)}$ *(the maximum weight is therefore* $30 - 15 = 15$*). Points* $(18, 4)$*,*$(12, 14)$ *and* $(6, 16)$ *remain extreme in* $P_{(j,15)}$ *since they still satisfy the weight constraint. The only computation that is required is then to check whether there exist new extreme points in the gap between* $(18, 4)$ *and* $(12, 14)$*. This is actually the case: the resolution of an iteration of Aneja and Nair's method between these two points yields the solution characterized by* $(16, 8, 8)$*. The two recursive calls yields no new extreme points, and the procedure stops. The extreme solutions of subproblem* $P_{(j,15)}$ *are thus:* $\{(18, 4, 9), (16, 8, 17), (12, 14, 9), (6, 16, 13)\}$*. This output has been obtained with only 3 iterations of Aneja and Nair's method (and therefore 3 launchings of a single objective optimization procedure) instead of 7 iterations if it had been computed from scratch.*

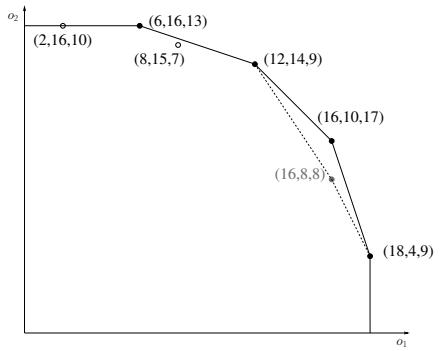

Fig. 6. Extreme solutions of $P_{(j,10)}$ (in black) and $P_{(j,15)}$ (in dotted line). The grey point is an extreme point in $P_{(j,15)}$ that was not an extreme point in $P_{(j,10)}$. The unfilled points are neither extreme points in $P_{(j,10)}$ nor in $P_{(j,15)}$.

Updating the Lower Bound. When a partial instanciation δ satisfies $(\mathbf{UB}^{\preccurlyeq}_{\Lambda_c}(\delta) \cap T_k) \cap \mathbf{LB}^{\succcurlyeq}_{\mathcal{N}(I)} \neq \emptyset$, then δ is not discarded but, before examining another partial instanciation, the algorithm checks whether a newly computed extreme point can be inserted into I. Indeed, we take advantage of the property that these extreme points correspond to feasible solutions in order to update I.

6. EXPERIMENTAL RESULTS

All experiments presented here were performed on an Intel® Core™ 2 Duo CPU E8400 @ 3.00GHz personal computer, endowed with 3.2GB of RAM memory, using a linux operating system. All algorithms were written in C++. The times shown here were calculated using the gettimeofday method, and memory requirements were calculated using the top command.

6.1. Instances

The types of instances considered here are the same as in the paper by Bazgan *et al.* [2009]:
Type A: random instances, where $p_1^j \in \{1,\dots,1000\}$, $p_2^j \in \{1,\dots,1000\}$ and $w^j \in \{1,\dots,1000\}$;
Type B: unconflicting instances, where $p_1^j \in \{101,\dots,1000\}$, $p_2^j \in \{p_1^j-100,\dots,p_1^j+100\}$ and $w^j \in \{1,\dots,1000\}$ (the values on both objectives are close);
Type C: conflicting instances, where $p_1^j \in \{1,\dots,1000\}$, $p_2^j \in \{\max\{900-p_1^j,1\},\dots,\min\{1100-p_1^j,1000\}\}$ and $w^j \in \{1,\dots,1000\}$ (the values on both objectives approximately sum up to 1000);
Type D: conflicting instances with correlated weights, where $p_1^j \in \{1,\dots,1000\}$, $p_2^j \in \{\max\{900-p_1^j,1\},\dots,\min\{1100-p_1^j,1000\}\}$ and $w^j \in \{p_1^j+p_2^j-200,\dots,p_1^j+p_2^j+200\}$ (the weights are positively correlated with the sum of both objectives).
All the parameters were uniformly randomly generated. Furthermore, for all these instances, we set $c = \lceil 0.5\sum_{j=1}^{n} w^j \rceil$. The average number of non-dominated points that are found in the numerical tests detailed in Section 6.2 below are indicated in Table I. If one evaluates the difficulty of an instance as the number of non-dominated points according to the size, it clearly shows that the instances of type B are the easiest ones while the instances of type D are the hardest ones.

Table I. Number of non-dominated points of 0-1 BOKP instances.

Type	Size	non-dominated points			Type	Size	non-dominated points		
		Min.	Avg.	Max.			Min.	Avg.	Max.
A	300	881	1126.7	1624	B	1000	99	152.8	217
	500	2276	2725.8	3301		2000	317	504.5	701
	700	4032	4881.5	6051		3000	763	954.3	1245
	1000	7642	9075.4	10197		4000	1187	1445.2	1823
C	200	982	1507.0	2030	D	100	1440	1687.5	1991
	300	2053	2858.4	3453		150	2802	3477.2	4172
	400	3484	4342.8	5457		200	4795	5541.5	6228
	500	5619	6613	7759		250	7083	8217.1	8880

6.2. Results

In this section, we will focus on three different aspects of our algorithm. First, the computational impact of the way the items are ordered in the biobjective dynamic programming procedure will be studied. Second, we will show the usefulness of all the different components of our algorithm, by comparing procedures where some components are disabled. Then, we will compare our method (named S2H hereafter, for Shaving, 2 phases method, and Hybrid dynamic programming) to the one of Bazgan *et al.* [2009] (named BHV hereafter, corresponding to the initials of the authors), since it is the most efficient dynamic programming method known to this date.

Table II. Ranking items according to O^{max}.

items	$(10,2,1)$	$(4,6,1)$	$(12,3,1)$
rank w.r.t p_1^j/w^j	2	3	1
rank w.r.t p_2^j/w^j	3	1	2
max rank	3	3	2
O^{max}	3	2	1

Table III. Computation times in seconds.

Type	Size	Time in seconds								
		O^{rand}			O^{max}			O^T		
		min.	avg.	max.	min.	avg.	max.	min.	avg.	max.
A	1000	427	588	757	424	556	705	250	325	407
B	4000	245	342	448	350	413	491	211	286	368
C	500	2006	4153	6631	983	1962	2636	866	1577	2180
D	250	2660	3930	5576	2121	2568	3283	1114	1397	1877

6.2.1. Sorting the Items. In order to limit time comsumption, the order in which the items are considered in the dynamic programming procedure is an important parameter. We compare here three ways to order items in the dynamic programming procedure for exploring each triangle :

— order O^{rand} (the same ordering for all triangles): the items are taken in a random order.

— order O^{max} (the same ordering for all triangles): this order has been introduced by Bazgan et al. [2009]. To obtain this ranking, items j are first ordered for each objective k according to the ratios p_k^j/w^j . In the biobjective case, two orderings are thus performed. Each item is then granted a number, which is the maximum rank in the two previous orderings. Finally, items are sorted increasingly according to these numbers (in order to discriminate items with the same rank, the sum of the ranks is used). For instance, consider set $N = \{(10,2,1),(4,6,1),(12,3,1)\}$ of items (we recall that the last component is the weight). The sorting according to O^{max} is obtained as indicated in Table II. Note that item $(4,6,1)$ is ranked better than item $(10,2,1)$ because the sum of the ranks of $(4,6,1)$ is $3+1=4$, while the one of $(10,2,1)$ is $2+3=5$.

— order O^T (a different ordering for each triangle): sorting the items decreasingly according to the ratio of their profits to their weights is a good ordering for the single objective knapsack problem. In order O^T, a different ordering is used in each triangle T_k, by ranking the items according to the ratio of a linear combination $\lambda_1 p_1^j + \lambda_2 p_2^j$ to w^j. Let y^k and y^{k+1} denote the two extreme points defining T_k. Coefficients λ_1 and λ_2 are set such that $\lambda_1 y_1^k + \lambda_2 y_2^k = \lambda_1 y_1^{k+1} + \lambda_2 y_2^{k+1}$.

The numerical tests were carried out on 30 randomly generated instances for each type and size. The average CPU times in seconds are presented in Table III. The results confirm that sorting items according to the zone of the Pareto frontier we explore (order O^T) leads to faster resolution times. The gain in time ranges from 20% to nearly 50% depending on the type of the instance. Therefore, this ordering will be used for the results presented hereafter unless otherwise specified.

6.2.2. Comparison of the Different Components of the Algorithm. We compared S2H and BHV by running both methods on the same instances[2] (and, of course, the same computer).

[2]We wish to thank Hadrien Hugot who kindly sent us the C++ code of the BHV method.

Table IV shows the time and memory spent to solve different types and sizes of instances. The first two columns indicate the type and size of the instances solved. For each type and size, 30 randomly generated instances have been solved using different methods, and the minimum, average and maximum times and memory requirements are indicated. Numbers in bold represent the best value for a given type and size. Shaving, hybridization and the implementation in two phases are the three main components of the algorithm presented in this paper. We evaluated some variations of our method in order to measure the importance of each component. Since using each part separately is sometimes meaningless (for instance using only the two phases method part and the dynamic programming procedure would lead to solve several times the very same problem), and always too much time consuming, we present methods using two out of the three parts: 2H is a two phases method using a hybridized dynamic programming (DP) procedure, SH is a hybridized DP procedure applied to a shaved problem, and finally S2 is a two phases method using simple DP on shaved problems. A time limit was set to 10000 seconds. Symbol "-" in the table denotes that at least one instance of this type and size reached this limit. Symbol "*" indicates that at least one instance couldn't be solved due to insufficient memory. Note that for method S2, we present here the results using O^{max}, because it yields better results. Indeed, the multiple sortings are very good for the bounds of our algorithm, it enhances the speed for the shaving part and the hybrid dynamic programming, but is not very efficient for classic dynamic programming. O^{max} was also used for method SH, since the implementation in two phases is disabled.

Shaving. The shaving procedure is particularly effective on the instances of type A or B: there is indeed a lot of items that are not interesting, such as items with low profits on both objectives, and high weights, and conversely items that have high profits and low weights, which will be taken in all non-dominated solutions. On the other hand, since all items in types C and D have conflicting profits, it is more difficult to shave items. Using the shaving procedure also enables to initialize the incumbent set accurately, leading to a lower computation time for all types of instances when used in combination with hybridization. Concerning memory requirements, the shaving procedure is also interesting for types A and B, but has almost no effect for the two other types.

Two phases. Using a two phases method makes it possible to divide the problem into several smaller problems, but there can be a lot of them (for problems of type A and size 1000, there are on average 155 subproblems to solve). The combination with the shaving procedure is interesting, because it further reduces the sizes of the subproblems. Furthermore, the combination with the hybridization is also effective, because it enables to compute only a fraction of the entire upper bound set (the one focusing on the area of interest). Finally, solving subproblems also requires less memory, but the memory space spared this way is not as important as that of the shaving for types A and B; the opposite is observed for types C and D.

Hybridization. Hybridizing the DP is the main part of our algorithm. Not only does it tremendously reduces the memory requirements, it also saves a lot of computation time for larger, or more difficult instances. This can be seen by looking at the results of method S2 on the instances of types C and D, both in terms of time and memory requirements.

6.2.3. Comparison between our Method and the BHV Method. From a memory consumption point of view, our method largely outperforms the BHV method for all sizes and types of instances (see Figure 7, continuous lines). This is a nice consequence of the

Table IV. Computation times in seconds, and memory requirements in megabytes of different methods to compute all non-dominated points of 0-1 BOKP.

Type	Size	Method	Time (s)			Memory (MB)			Type	Size	Method	Time (s)			Memory (MB)		
			Min.	Avg.	Max.	Min.	Avg.	Max.				Min.	Avg.	Max.	Min.	Avg.	Max.
A	300	S2H	**8.5**	**12.9**	**22.7**	**2.3**	**2.6**	**3.1**	B	1000	S2H	5.1	7.0	11	**1.8**	**2.0**	**2.3**
		BHV	27	51	103	57	80	113			BHV	**1.4**	**4.1**	**10**	9	11	13
		2H	21.1	30.7	50.2	5.5	5.8	6.3			2H	10	17	40	15	15.2	16
		SH	43	60	88	3.8	4.0	4.4			SH	7.3	10	16	2.2	2.5	2.8
		S2	62	113	208	9	13.4	16			S2	7.4	10	15	1.8	2.3	3.6
	500	S2H	**30.9**	**48.7**	**70.1**	**2.8**	**3.1**	**3.7**		2000	S2H	**28**	**41**	**60**	**2.2**	**2.5**	**2.8**
		BHV	387	564	1031	225	401	449			BHV	57	132	272	57	132	272
		2H	114	167	221	8.5	8.8	9.3			2H	79	135	226	29	30.1	31
		SH	343	448	679	5.7	6.2	6.6			SH	55	109	181	3.5	3.8	4.3
		S2	400	671	1045	22	38	56			S2	51	91	123	7	14	21
	700	S2H	**86.7**	**126**	**162**	**3.2**	**3.6**	**3.8**		3000	S2H	**102**	**130**	**171**	**2.7**	**2.9**	**3.2**
		BHV	1781	2740	4184	897	1308	1800			BHV	610	874	1292	449	449	449
		2H	355	538	695	11	11.1	12			2H	378	553	914	44	44.6	45
		SH	1443	2209	3353	8.0	8.7	9.4			SH	371	517	699	4.7	4.9	5.2
		S2	1555	2820	3624	81	116	159			S2	249	344	468	26	45	74
	1000	S2H	**250**	**325**	**407**	**3.8**	**4.2**	**4.7**		4000	S2H	**211**	**286**	**368**	**3.0**	**3.3**	**3.7**
		BHV	*	*	*	*	*	*			BHV	1985	3017	4184	897	1307	1800
		2H	1349	1733	2122	15	15.5	16			2H	1008	1518	2205	58	58.5	60
		SH	-	-	-	-	-	-			SH	1268	1648	2097	5.8	6.1	6.4
		S2	-	-	-	-	-	-			S2	755	970	1308	63	84	130
C	200	S2H	38	64	119	3.8	4.3	5.0	D	100	S2H	57	84	136	**4.7**	**5.1**	**6.0**
		BHV	**21**	**32**	**47**	57	63	113			BHV	**16**	**24**	**35**	57	80	113
		2H	31	53	89	4.4	4.8	5.3			2H	79	108	169	4.7	5.1	6.0
		SH	96	147	239	**2.8**	**3.1**	**3.4**			SH	100	125	165	5.3	6.0	6.7
		S2	1555	1835	2307	71	107	163			S2	1553	2138	3252	101	124	168
	300	S2H	152	255	365	**5.0**	**5.9**	**6.9**		150	S2H	175	232	352	**6.2**	**6.9**	**7.8**
		BHV	143	**206**	**288**	225	257	449			BHV	**113**	**154**	**228**	225	311	449
		2H	**131**	235	346	6.0	6.6	7.3			2H	218	302	432	6.8	7.5	8.8
		SH	602	788	1159	8.6	9.4	10			SH	652	698	1123	8.1	9.2	10
		S2	-	-	-	-	-	-			S2	-	-	-	-	-	-
	400	S2H	**369**	**723**	1062	**6.6**	**7.7**	**9.0**		200	S2H	492	657	1057	**8.3**	**8.9**	**10**
		BHV	486	748	**1006**	449	782	897			BHV	**441**	**572**	**770**	897	897	897
		2H	472	738	1050	7.9	8.9	9.9			2H	694	954	1742	**8.3**	**8.9**	**10**
		SH	1539	2806	3956	12	14.8	18			SH	2054	2689	3747	11	13.1	16
		S2	-	-	-	-	-	-			S2	-	-	-	-	-	-
	500	S2H	866	**1577**	**2180**	**8.4**	**9.6**	**10**		250	S2H	**1114**	**1397**	**1877**	**9.5**	**10.6**	**13**
		BHV	1447	2014	2651	897	1458	1800			BHV	1328	1581	1989	1100	1730	1800
		2H	**844**	1634	2086	9.7	10.4	11			2H	1679	2030	2698	9.7	**10.6**	**13**
		SH	-	-	-	-	-	-			SH	6025	6984	8516	14	18.1	21
		S2	-	-	-	-	-	-			S2	-	-	-	-	-	-

2H: method S2H without shaving SH: method S2H without two phases
S2: method S2H without hybridization

hybridization method, which enables to discard many intermediate results in dynamic programming. From the computation time perspective, the results depend on the types and sizes of instances (see Figure 7, dotted lines). For types A and B the S2H method is much faster than the BHV method, while for types C and D it becomes faster when the size of instances grows. The reason for this behaviour is that, as stated above, the shaving is less effective, and the fathoming criterion is rather time consuming, but this is compensated for bigger instances by the fact that a lot of computation time is saved thanks to the important number of elements that are fathomed.

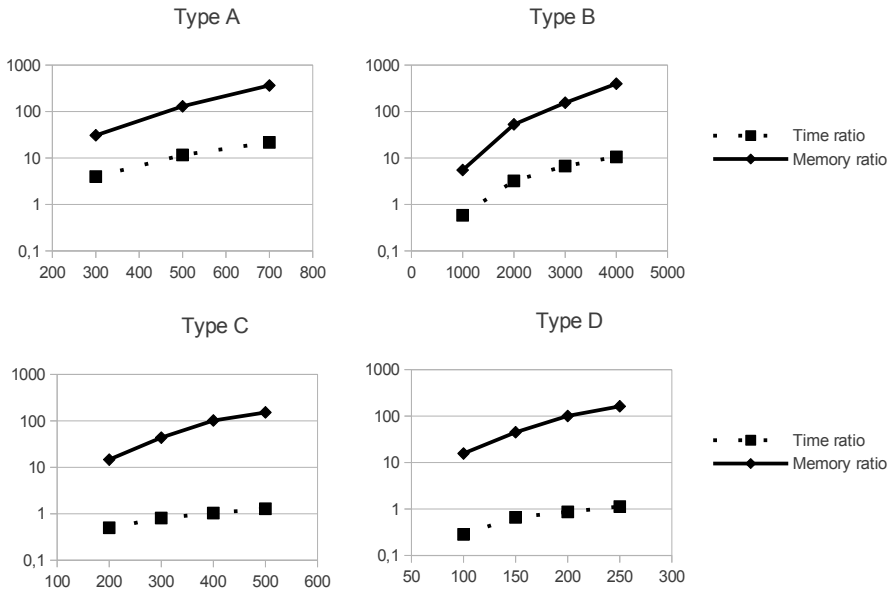

Fig. 7. Time and memory ratios BHV/S2H.

7. CONCLUSION

In this paper, we have presented a hybrid dynamic programming approach for multiobjective combinatorial optimization. It has been applied to the biobjective binary knapsack problem. Our approach outperforms previous dynamic programming methods from the viewpoint of memory requirements and resolution times. A natural extension of this work would be to investigate the impact of hybrid dynamic programming on other MOCO problems (e.g., multiobjective shortest path). Another extension would be to study how to improve the resolution times on conflicting instances of 0-1 BOKP. For this purpose, apart from implementation tips (e.g., by speeding up the memory allocation process), an incremental resolution of the single objective problems (which is the most cumbersome primitive in our method) is worth investigating. Finally, note that our fathoming criterion has only been implemented in the biobjective case up to now. The study of its practical implementation in problems involving more than two objectives is an interesting and potentially fruitful task in our opinion.

REFERENCES

ANEJA, Y. AND NAIR, K. 1979. Bicriteria transportation problem. *Management Science 25*, 73–78.

BAZGAN, C., HUGOT, H., AND VANDERPOOTEN, D. 2007. An efficient implementation for the 0-1 multi-objective knapsack problem. In *WEA*. 406–419.

BAZGAN, C., HUGOT, H., AND VANDERPOOTEN, D. 2009. Solving efficiently the 0-1 multi-objective knapsack problem. *Computers and Operations Research 36,* 1, 260–279.

BITRAN, G. AND RIVERA, J. 1982. A combined approach to solve binary multicriteria problems. *Naval Research Logistics Quarterly 29*, 181–201.

CAPTIVO, M., CLÌMACO, J., FIGUEIRA, J., MARTINS, E., AND SANTOS, J. 2003. Solving bicriteria 0-1 knapsack problems using a labeling algorithm. *Comput. Oper. Res. 30,* 12, 1865–1886.

DAELLENBACH, H. AND DE KLUYVER, C. 1980. Note on multiple objective dynamic programming. *Journal of the Operational Research Society 31*, 591–594.

EHRGOTT, M. 2005. *Multicriteria Optimization, second edition*. Springer.

EHRGOTT, M. AND GANDIBLEUX, X. 2004. Approximative solution methods for multiobjective combinatorial optimization. *Journal of the Spanish Statistical and Operations Research Society 12,* 1, 1–88.

EHRGOTT, M. AND GANDIBLEUX, X. 2007. Bound sets for biobjective combinatorial optimization problems. *Computers & Operations Research 34,* 9, 2674–2694.

GALAND, L. 2008. Méthodes exactes pour l'optimisation multicritère dans les graphes : recherche de solutions de compromis. Ph.D. thesis, Université Paris 6 - UPMC.

GARFINKEL, R. AND NEMHAUSER, G. 1972. *Integer Programming*. John Wiley and Sons, New York.

HANSEN, P. 1980. Bicriterion path problems. In *Multiple Criteria Decision Making: Theory and Applications, LNEMS 177*, G. Fandel and T. Gal, Eds. Springer-Verlag, Berlin, 109–127.

KELLERER, H., PFERSCHY, U., AND PISINGER, D. 2004. *Knapsack Problems*. Springer, Berlin, Germany.

KIZILTAN, G. AND YUCAOGLU, E. 1983. An algorithm for multiobjective zero-one linear programming. *Management Science 29,* 12, 1444–1453.

KLAMROTH, K. AND WIECEK, M. 2000. Dynamic programming approaches to the multiple criteria knapsack problem. *Naval Research Logistics 47*, 57–76.

LUST, T. AND TEGHEM, J. 2010. The multiobjective multidimensional knapsack problem: a survey and a new approach. *CoRR abs/1007.4063*.

MARCOTTE, O. AND SOLAND, R. 1986. An interactive branch-and-bound algorithm for multiple criteria optimization. *Management Science 32,* 1, 61–75.

MARTELLO, S., PISINGER, D., AND TOTH, P. 1999. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science 45*, 414–424.

MARTIN, P. AND SHMOYS, D. 1996. A new approach to computing optimal schedules for the job shop scheduling problem. In *Proceedings of the Fifth international IPCO conference, Vancouver, Canada*, S. M. W.H. Curnigham and M. Queyranne, Eds. LNCS 1084, 389–403.

MAVROTAS, G. AND DIAKOULAKI, D. 1998. A branch and bound algorithm for mixed zero-one multiple objective linear programming. *European Journal of Operational Research 107*, 530–541.

MORIN, T. AND MARSTEN, R. 1976. Branch and bound strategies for dynamic programming. *Operations Research 24*, 611–627.

PISINGER, D. 1997. A minimal algorithm for the 0-1 knapsack problem. *Operations Research 45*, 758–767.

PRZYBYLSKI, A., GANDIBLEUX, X., AND EHRGOTT, M. 2008. Two phase algorithms for the bi-objective assignment problem. *European Journal of Operational Research 185,* 2, 509–533.

SERAFINI, P. 1986. Some considerations about computational complexity for multiobjective combinatorial problems. In *Recent advances and historical development of vector optimization*, J. Jahn and W. Krabs, Eds. Lecture Notes in Economics and Mathematical Systems Series, vol. 294. Springer-Verlag, Berlin.

SOURD, F. AND SPANJAARD, O. 2008. A multi-objective branch-and-bound framework. Application to the bi-objective spanning tree problem. *INFORMS Journal of Computing 20,* 3, 472–484.

STEINER, S. AND RADZIK, T. 2008. Computing all efficient solutions of the biobjective minimum spanning tree problem. *Comput. Oper. Res. 35,* 1, 198–211.

ULUNGU, E. AND TEGHEM, J. 1995. The two phases method: An efficient procedure to solve bi-objective combinatorial optimization problems. *Foundations of Computing and Decision Sciences 20,* 2, 149–165.

VILLAREAL, B. AND KARWAN, M. 1981. Multicriteria integer programming: A (hybrid) dynamic programming recursive approach. *Mathematical Programming 21*, 204–223.

VISÉE, M., TEGHEM, J., PIRLOT, M., AND ULUNGU, E. L. 1998. Two-phases method and branch and bound procedures to solve the bi–objective knapsack problem. *J. of Global Optimization 12,* 2, 139–155.