Computers and Operations Research 89 (2018) 82–93

Contents lists available at ScienceDirect

Computers and Operations Research

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

Compressed data structures for bi-objective {0,1}-knapsack problems

Pedro Correia[a,b,*], Luís Paquete[a], José Rui Figueira[b]

[a] CISUC, Department of Informatics Engineering, University of Coimbra, Portugal
[b] CEG-IST, Instituto Superior Técnico, Universidade de Lisboa, Portugal

ARTICLE INFO

Article history:
Received 20 June 2016
Revised 11 August 2017
Accepted 13 August 2017
Available online 17 August 2017

Keywords:
Multi-objective optimization
Implicit enumeration techniques

ABSTRACT

Solving multi-objective combinatorial optimization problems to optimality is a computationally expensive task. The development of implicit enumeration approaches that efficiently explore certain properties of these problems has been the main focus of recent research. This article proposes algorithmic techniques that extend and empirically improve the memory usage of a dynamic programming algorithm for computing the set of efficient solutions both in the objective space and in the decision space for the bi-objective knapsack problem. An in-depth experimental analysis provides further information about the performance of these techniques with respect to the trade-off between CPU time and memory usage.

1. Introduction

Dealing with a large amount of solutions for further processing is a key concern in the field of multi-objective combinatorial optimization. Such processing include, for example, gathering or producing a collection of data sets within a limited memory (internal or external), extraction of important pieces of information from the whole data, manage the data, which deals with several operations, and process these operations in a reasonable amount of CPU-time. These aspects require the use of efficient data structures.

Solution methods for multi-objective combinatorial optimization (MOCO) problems typically require a large usage of memory resources. Parametric and recursive programming (Ebem-Chaime, 1996; Przybylski et al., 2010), approximation methods (Erlebach et al., 2002), metaheuristics (Köksalan and Phelps, 2007), or exact methods (Bazgan et al., 2009b; Delort and Spanjaard, 2013; Figueira et al., 2013) are example of approaches that require more memory usage due to the large number of potential solutions that need to be kept during the search process. For instance, the experimental analysis reported in Figueira et al. (2013) shows that more than five million solutions need to be kept in memory in order to solve bi-objective knapsack problem instances with less than one thousand items; see similar results reported in Paquete et al. (2013) for a related problem and using a similar approach. Noteworthy, the implementations described in the literature only keep the *outcome vectors* in the objective space in memory, as for example, the algorithms by Bazgan et al. (2009b) and Figueira et al. (2013). Therefore, the memory requirements for keeping also the solutions should be much larger than those reported in the literature, making it infeasible for the usual memory capacity of current personal computers.

In this paper, we consider the bi-objective knapsack problem (BOKP). Several approaches to solve the BOKP exactly or with a good approximation quality have been proposed. Klamroth and Wiecek (2000) suggested five models to solve multi-objective integer knapsack problems (MOIKP). Each model is based on a network in which each state represents the set of all non-dominated solutions of a sub-problem. The authors show how these models can be adapted to different variants of the knapsack problem. Modeling the BOKP into a bi-objective shortest path problem over an acyclic network was proposed in Captivo et al. (2003). The model was solved by using a labeling algorithm. Three complementary dominance relations were proposed in Bazgan et al. (2009b) to be used in a dynamic programming algorithm. These relations were applied amongst potential solutions and used to fathom states that do not lead to efficient solution. The quality of the fathoming process of Bazgan et al. (2009b) was improved in Figueira et al. (2013) by proposing new prunning techniques for such a method. Delort and Spanjaard (2013) proposed a technique using an hybrid dynamic programming approach for a two phased algorithm to solve the BOKP. Some approaches provide quality guarantee approximations for the BOKP. In Erlebach et al., 2002 a fully polynomial time approximation scheme (FPTAS) scheme was developed to guarantee that for each efficient solution, another that is at most at a factor $(1+\epsilon)$ on all objective values is found. Bazgan et al. (2009a) proposed the usage of dominance relations to develop a new FPTAS scheme to solve the BOKP.

* Corresponding author at: Departamento de Engenharia Informática Faculdade de Ciências e Tecnologia Universidade de Coimbra Pólo II - Pinhal de Marrocos 3030-290 Coimbra Portugal.

E-mail addresses: pamc@dei.uc.pt (P. Correia), paquete@dei.uc.pt (L. Paquete), figueira@tecnico.ulisboa.pt (J.R. Figueira).

http://dx.doi.org/10.1016/j.cor.2017.08.008
0305-0548/

In this paper we study the impact of using different data structures in the case of the bi-objective knapsack problem when solutions in the *space of decision variables* (*e.g.*, a binary string) should be kept in memory. Two main data structures are investigated: Binary decision diagrams (Akers, 1978) and differencing methods based on spanning tree structures (Kang et al., 1977). Although these techniques are well-known, they have never been applied in the context of compression of solutions during the optimization process in a multi-objective framework. For benchmark purpose, we compare them against more naïve approaches, such as compression algorithms based on the Lempel–Ziv–Welch variant (Welch, 1984). We remark that any compression procedure has a significant overhead on CPU-time, even if the update is performed incrementally. Therefore, we are interested in understanding the effect of these techniques in terms of the trade-off between memory and CPU-time. In fact, our computational results suggest that some of these techniques can be located in different places of this trade-off.

This paper is organized as follows. Section 2 provides theoretical background. Section 3 is devoted to the presentation of the data structures implemented. Section 4 deals with other methods developed for benchmarking purposes. Section 5 presents a computational study. Finally, some conclusions and avenues for future research are provided.

2. Theoretical background

In the following, we present the fundamental concepts, definitions, and notation for MOCO problems and for the bi-objective knapsack problem as well as the fundamental framework needed for our algorithmic developments, with an illustrative example.

2.1. Fundamental concepts, definitions and notation

Let "max"$\{z_1(x), \ldots, z_m(x), \ldots, z_d(x) : x \in X\}$ denote the mathematical programming formulation of the generic linear MOCO problem, where $d \geq 2$ is the number of linear objective functions, x represents a vector of *decision variables*, $(x_1, \ldots, x_j, \ldots, x_n)$, and X is the *feasible set* (or region) in the *decision space* $\{0, 1\}^n$, in general resulting from the intersection of a set of linear constraints. Note that $z_m : X \to \mathbb{R}$, for all $m = 1, \ldots, d$. Note also that $z(x')$, or simply z', is the outcome vector with respect to solution x'. The feasible set is a finite set or it possesses a countable discrete structure. The elements $x \in X$ are characterized by some combinatorial properties (permutations, combinations, etc.). The "max" operator used in above definition of a MOCO problem is mathematically not meaningful since it is impossible to maximize all the objective functions at the same time; it barely serves here to state that each objective function is to be maximized. Since generally the objectives are conflicting each other, and there is no common optimal solution for all the objective functions, it leads to a different important concept of optimality, which can be derived from the binary relation defined below. (See chap. 6 in Steuer, 1986.)

Definition 1 (Dominance)**.** Let $z', z'' \in \mathbb{R}^d$ denote two outcome vectors. Then, z' *dominates* z'', denoted by $z' \Delta z''$, if and only if $z'_m \geqslant z''_m$, for all $m = 1, \ldots, d$ with at least one strict inequality.

Let $Z \subset \mathbb{R}^d$ denote the image of the feasible set X in the *objective space*. Let $N(Z)$ denote the whole *set of non-dominated outcome vectors*; $Z^{\leqslant} = \{z \in \mathbb{R}^d : z \leq 0\}$ denote the *negative cone* formed by all the possible coordinates of the negative orthant; $Z^{\leqslant}_{\tilde{z}} = \tilde{z} \oplus Z^{\leqslant}$ denote the displaced cone $Z^{\leqslant}$ at point $\tilde{z}$; $\hat{Z} = \{z \in \mathbb{R}^d : z \in Z^{\leqslant}_{\tilde{z}}$ for all $\tilde{z} \in N(Z)\}$ denote the set formed by $Z^{\leqslant}_{\tilde{z}}$ for all $\tilde{z} \in N(Z)$; $\mathcal{Z} = \text{conv}(\hat{Z})$ denote the convex hull of $\hat{Z}$; $\text{int}(\mathcal{Z})$ denote the interior of $\mathcal{Z}$; and $\text{bd}(\mathcal{Z})$ the boundary of $\mathcal{Z}$. Now, we can easily distinguish between *supported non-dominated outcome vectors*, those that belong to $N(Z)$ and are placed in $\text{bd}(\mathcal{Z})$, which can easily be computed, for example, by the approach proposed by Ehrgott-Chaime (1996), and *unsupported non-dominated vectors*, those that belong to $N(Z)$ and are located in $\text{int}(\mathcal{Z})$. Let $E(X)$ denote the *set of efficient solutions* in the decision space, the image of which is $N(Z)$. Note that $|E(X)| \geq |N(Z)|$.

2.2. The bi-objective $\{0, 1\}-$knapsack problem

The *bi-objective* {0, 1}*-knapsack* problem (BOKP) can be stated as follows. (See chap. 10 in Ehrgott, 2005.)

$$\begin{aligned}
\text{maximize} \quad & z_1(x_1, \ldots, x_j, \ldots, x_n) = \sum_{j=1}^{n} c_j^1 x_j \\
\text{maximize} \quad & z_2(x_1, \ldots, x_j, \ldots, x_n) = \sum_{j=1}^{n} c_j^2 x_j \\
\text{subject to:} \quad & \sum_{j=1}^{n} w_j x_j \leqslant W \\
& x_j \in \{0, 1\}, \quad j = 1, \ldots, n.
\end{aligned} \tag{BOKP}$$

where x_j are the {0, 1} decision variables (unknowns), c_j^m are the coefficients of such variables in the linear objective functions, w_j are the weights of variables x_j in the linear (side) constraint, $j = 1, \ldots, n$, $m = 1, 2$, and W is the right-hand side of the constraint. To avoid trivial solutions, we assume that the data parameters of this model are positive integers.

2.3. Branching programs and binary decision trees

We describe the branching program for building a binary decision tree that generates the efficient set to a given BOKP instance. This is the basis of the most efficient dynamic programming approaches for this problem (Bazgan et al., 2009b; Figueira et al., 2013; Klamroth and Wiecek, 2000).

Definition 2 (Binary decision tree)**.** A *binary decision tree* (BDT) is a rooted binary tree organized by levels, $j = 0, \ldots, n$, where $n + 1$ is its *depth*. The level 0 contains the root vertex and each of the following levels, $j = 1, \ldots, n$, contains vertices that are at distance j to the root. Every non-leaf vertex v has at most two children, denoted by $S_0(v)$ and $S_1(v)$, and a parent denoted by $F(v)$, except if it is the root vertex.

Procedure 1 provides the main algorithmic procedure to solve a BOKP instance, which is referred to as P. Lists L^j, $j = 0, \ldots, n$ represent ordered lists of sub-problems generated at iteration j. Problem P_i^j, $i = 1, \ldots, |L^j|$ is the sub-problem i at iteration j. Note that P_1^0 corresponds to the original instance P. The procedure is based on two main phases: branching and pruning. In the branching phase of each iteration j, for $j = 1, \ldots, n$, every sub-problem is separated in two, by fixing $x_j = 0$ and $x_j = 1$. Line 7 of Procedure 1 represents the branching phase, which is performed for each problem in the list generated at iteration $j - 1$, L^{j-1}. Note that after pruning, both generated problems are enqueued to the list of generation j, L^j (line 8 and 9). In the pruning phase, each sub-problem is subject to four pruning rules that are used to discard sub-problems, as stated in Definition 3. Line 10 shows the pruning step. Note that the procedure returns both sets $E(X)$ and $N(Z)$ of instance P, which are extracted from list L^n in line 11.

The procedure is implemented by keeping a list R^j that, at the end of each iteration j, for $j = 1, \ldots, n$, contains a list of partial solutions with fixed values of $x_1, \ldots, x_j$. A partial solution at iteration