Computers & Operations Research 40 (2013) 2625–2631

Contents lists available at SciVerse ScienceDirect

Computers & Operations Research

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

Exact solution of the robust knapsack problem ☆

Michele Monaci [a,*], Ulrich Pferschy [b], Paolo Serafini [c]

[a] DEI, University of Padova, Via Gradenigo 6/A, I-35131 Padova, Italy
[b] Department of Statistics and Operations Research, University of Graz, Universitaetsstrasse 15, A-8010 Graz, Austria
[c] DIMI, University of Udine, Via delle Scienze 206, I-33100 Udine, Italy

ARTICLE INFO

Available online 18 May 2013

Keywords:
Knapsack problem
Robust optimization
Dynamic programming

ABSTRACT

We consider an uncertain variant of the knapsack problem in which the weight of the items is not exactly known in advance, but belongs to a given interval, and an upper bound is imposed on the number of items whose weight differs from the expected one. For this problem, we provide a dynamic programming algorithm and present techniques aimed at reducing its space and time complexities. Finally, we computationally compare the performances of the proposed algorithm with those of different exact algorithms presented so far in the literature for robust optimization problems.

1. Introduction

The classical *Knapsack Problem* (KP) can be described as follows. We are given a set $N=\{1,\dots,n\}$ of items, each of them with positive profit p_j and positive weight w_j, and a knapsack capacity c. The problem asks for a subset of items whose total weight does not exceed the knapsack capacity, and whose profit is a maximum. It can be formulated as the following Integer Linear Program (ILP):

$$(KP)\quad \max \sum_{j\in N} p_j x_j \tag{1}$$

$$\sum_{j\in N} w_j x_j \le c \tag{2}$$

$$x_j\in\{0,1\},\quad j\in N. \tag{3}$$

Each variable x_j takes value 1 if and only if item j is inserted in the knapsack.

This problem is NP-hard, although in practice fairly large instances can be solved to optimality within reasonable running time. Furthermore, dynamic programming algorithms with pseudopolynomial running time are available. A comprehensive survey on all aspects of (KP) was given by Kellerer et al. [11].

In this paper we consider the following variant of (KP), aimed at modeling uncertainties in the data, in particular in the weights: for each item j the weight may deviate from its given *nominal value* w_j and attain an arbitrary value in some known interval $[w_j-\underline{w}_j, w_j+\overline{w}_j]$. A feasible solution must obey the capacity constraint (2) no matter what the actual weight of each item turns out to be. However, uncertainty is bounded by an integer parameter Γ indicating that at most Γ items in the solution can change from their nominal value w_j to an arbitrary value in the interval. Clearly a diminution of a weight below the nominal value does not affect feasibility and in the worst case all changed weights reach their upper limit. Hence a feasible solution consists of a subset of items $J\subseteq N$ such that

$$\sum_{j\in J} w_j + \sum_{j\in \hat{J}} \overline{w}_j \le c \quad \forall \hat{J}\subseteq J,\ |\hat{J}|\le \Gamma. \tag{4}$$

We call this problem the *Robust Knapsack Problem* (RKP). It was recently considered by Monaci and Pferschy [16] who studied the worst-case ratio between the optimal solution value of (KP) and that of (RKP), as well as the ratio between the associated fractional relaxations. A similar setting with the restriction that $\overline{w}_j=\delta w_j$ for all j for some given constant $\delta>0$ was introduced by Bertsimas and Sim [4]. Clearly this is a particular case of the model considered in this paper.

In the following we assume, without loss of generality, that all input data are integer and items are sorted according to non-increasing $\overline{w}_j$ values. For notational simplicity we define the increased weights by $\hat{w}_j=w_j+\overline{w}_j$ for all j. In addition, for any given set S of items, we will denote by $p(S)=\sum_{j\in S}p_j$ and $w(S)=\sum_{j\in S}w_j$ the total profit and weight, respectively, of the items in S.

In this paper we review exact solution algorithms for (RKP). Although it is an NP-hard problem, exact solutions can be found in reasonable time even for large instances (see in Section 6 the computing times for instances up to 5000 items). Hence it is adequate to look for exact methods in solving (RKP) and it is

* Corresponding author. Tel.: +39 49 8277920.
E-mail addresses: monaci@dei.unipd.it (M. Monaci), pferschy@uni-graz.at (U. Pferschy), paolo.serafini@uniud.it (P. Serafini).

0305-0548/$ - see front matter © 2013 The Authors. Published by Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cor.2013.05.005

interesting to compare the behavior of different algorithms. The algorithms proposed in the literature up-to-date present quite distinct features, although two of them can be shown to be very tightly intertwined.

In Section 2 we present a dynamic programming algorithm. The algorithm mimics the well known algorithm for the standard knapsack problem, but is able to take care of the upper weights once the items are sorted according to non-increasing $\overline{w}_j$ values. This algorithm, developed by the authors, is investigated in detail. In Section 2 we show its correctness, and, as in the usual knapsack algorithm, we show that a similar version obtained by exchanging the roles of weights and values can be also formulated thus paving the way to approximation schemes.

In Section 3 we address the delicate issue of implementing the algorithm with a reduced amount of memory, since, with a large number of items and large data coefficients, space requirements can constitute a problem.

Other exact approaches are presented in Section 4. In particular we review the integer programming model by Bertsimas and Sim [4] (Section 4.1), an improvement on the iterative approach by Bertsimas and Sim [3] which requires solving $O(n)$ knapsack instances (Section 4.2) and the Branch-and-Cut algorithm by Fischetti and Monaci [8] in which the robustness requirements are modeled by cutting inequalities (Section 4.3).

The problem we investigate is a special combinatorial optimization problem that has been motivated by a particular modeling of the problem uncertainties. By and large this is the model which has received most attention in the literature, although a lot of research has been done to face problems with uncertain data (see, e.g. the recent survey by Bertsimas, Brown and Caramanis [2]). Recently, Poss [19] pointed out drawbacks of this approach from a probabilistic point of view.

As to uncertainty in knapsack problems, few contributions were proposed. (RKP) was first introduced by Bertsimas and Sim [3]. Klopfenstein and Nace [12] defined a robust chance-constrained variant of the knapsack problem and studied the relation between feasible solutions of this problem and those of (RKP). A polyhedral study of (RKP) was conducted by the same authors in [13], where some computational experiments with small instances (up to 20 items) were given. Recently, Büsing et al. [5,6] addressed the robust knapsack problem within the so-called recoverable robustness context in which one is required to produce a solution that is not necessarily feasible under uncertainty, but whose feasibility can be recovered by means of some legal moves. In [5,6], legal moves correspond to the removal of at most K items from the solution, so as to model a telecommunication network problem. For this problem, the authors gave different ILP formulations, cut separation procedures and computational experiments.

2. A dynamic programming algorithm

In this section we present an exact dynamic programming algorithm for (RKP). Note that the same problem was considered by Klopfenstein and Nace [12] who sketched a related dynamic programming recursion in their Theorem 3. While the brief description of the algorithm in [12] relies on a modification of a dynamic program for the nominal knapsack problem, we present a detailed algorithm explicitly designed for (RKP) which allows for an improvement of the complexities. In Section 3 we will analyze time and space complexities of our algorithm, and propose possible methods for reducing both of them; finally, the algorithm will be used in Section 5 to derive a fully polynomial approximation scheme for (RKP).

Our approach is based on the following two dynamic programming arrays: Let $\overline{z}(d,s,j)$ be the highest profit for a feasible solution with total weight d in which only items in $\{1,\dots,j\}\subseteq N$ are considered and *exactly* s of them are included, all with their upper weight bound $\hat{w}_j$. Let $z(d,j)$ be the highest profit for a feasible solution with total weight d in which only items in $\{1,\dots,j\}\subseteq N$ are considered and exactly Γ of them change from their nominal weight to their upper bound. Clearly, $d=0,1,\dots,c$, $s=0,1,\dots,\Gamma$, and $j=0,1,\dots,n$.

A crucial property for the correctness of our approach is the assumption that items are sorted by non-increasing weight increases $\overline{w}_j$. This implies the following lemma. For a subset of items $J\subseteq N$ denote by j_Γ the index of the $\Gamma-$th item in J if $|J|\geq\Gamma$, otherwise j_Γ is the index of the last item in J.

Lemma 1. *A subset $J\subseteq N$ is feasible if and only if*

$$\sum_{j\in J|j\leq j_\Gamma}\hat{w}_j+\sum_{j\in J|j>j_\Gamma}w_j\leq c$$

Proof. The largest increase of $w(J)$ caused by items attaining their upper weight is due to the subset of Γ items for which the increase $\overline{w}_j$ is largest. If J is feasible with respect to this subset, it is feasible for any other subset of J. □

Now we can compute all array entries by the following dynamic programming recursions:

$$\begin{aligned}\overline{z}(d,s,j)&=\max\{\overline{z}(d,s,j-1),\overline{z}(d-\hat{w}_j,s-1,j-1)+p_j\}\\&\quad\text{for } d=0,\dots,c,\ s=1,\dots,\Gamma,\ j=1,\dots,n,\\z(d,j)&=\max\{z(d,j-1),z(d-w_j,j-1)+p_j\}\\&\quad\text{for } d=0,\dots,c,\ j=\Gamma+1,\dots,n\end{aligned}\tag{5}$$

The initialization values are $\overline{z}(d,s,0)=-\infty$ for $d=0,\dots,c$ and $s=0,\dots,\Gamma$. Then we set $\overline{z}(0,0,0)=0$. The two arrays are linked together by initializing $z(d,\Gamma)=\overline{z}(d,\Gamma,\Gamma)$ for all d. Obviously, all entries with $d<\hat{w}_j$ (respectively $d<w_j$) are not used in definition of $\overline{z}$ (respectively z) in recursion (5). The optimal solution value of the robust knapsack problem can be found as

$$z^*=\max\begin{cases}\max\{z(d,n)|d=1,\dots,c\}\\\max\{\overline{z}(d,s,n)|d=1,\dots,c,\ s=1,\dots,\Gamma-1\}\end{cases}$$

and consumes a total capacity $c^*\leq c$.

Intuitively, the dynamic programming algorithm operates in two phases: first, it determines the best solution consisting of (at most) Γ items with increased weight. Then, this solution is possibly extended with additional items at their nominal weight. This separation into two phases is possible because the sorting by non-increasing $\overline{w}_j$ guarantees that in any solution the items with smallest indices, i.e. those that were packed into the knapsack earlier, are those that will attain their increased weight (see Lemma 1).

Theorem 2. *The dynamic programming recursion (5) yields an optimal solution of (RKP).*

Proof. We build an acyclic directed graph and show that the recursion corresponds to a longest path in the graph. The nodes are labeled as (d,s,j), with $d=0,\dots,c$, $j=0,\dots,n$, $s=0,\dots,\Gamma$. The node $(0,0,0)$ is the source and an additional node, labeled t, is the destination.

The arcs are defined as follows: within each group of nodes with the same label s (let us denote them as a "stage") there are arcs $(d,s,j-1)\to(d,s,j)$ with value 0. Let us denote these arcs as "empty". Using an empty arc corresponds to never inserting item j. Moreover, there are other empty arcs with value 0 from each node (d,s,n) to the destination t to model situations in which the solution includes less than Γ items.