Exact Solution of the Robust Knapsack Problem

Michele Monaci[1] and Ulrich Pferschy[2] and Paolo Serafini[3]

[1] *DEI, University of Padova, Via Gradenigo 6/A, I-35131 Padova, Italy.* monaci@dei.unipd.it

[2] *Department of Statistics and Operations Research, University of Graz, Universitaetsstrasse 15, A-8010 Graz, Austria.* pferschy@uni-graz.at

[3] *DIMI, University of Udine, Via delle Scienze 206, I-33100 Udine, Italy.* paolo.serafini@uniud.it

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.

keywords: knapsack problem, robust optimization, dynamic programming.

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 follow Integer Linear Program (ILP):

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

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

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

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, Pferschy and Pisinger [10].

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

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

We call this problem the *Robust Knapsack Problem* (RKP). It was recently considered by Monaci and Pferschy [14] 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 [3]. 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 5,000 items). Hence it is adequate to look for exact methods in solving (RKP) and it is 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 [3] (Section 4.1), the iterative approach by Bertsimas and Sim [2] which requires solving $n+1$ knapsack instances (Section 4.2) and the Branch-and-Cut algorithm by Fischetti and Monaci [7] in which the robustness requirements is 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 [1]). As to uncertainty in knapsack problems, few contributions were proposed. (RKP) was first introduced by Bertsimas and Sim [2], while Klopfenstein and Nace [11] defined a robust chance-constrained variant of the knapsack problem and studied the relation between feasible solutions of this problems an those of (RKP). A polyhedral study of (RKP) was conducted by the same authors in [12], where some computational experiments with small instances (up to 20 items) were given. Recently, Büsing, Koster and Kutschka [4, 5] 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 [4, 5], 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 [11] who sketched a related dynamic programming recursion in their Theorem 3. While the brief description of the algorithm in [11] 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.

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 $\{1, \ldots, j\} \subseteq N$ 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 $\{1, \ldots, j\} \subseteq N$ are considered and Γ of them change from their nominal weight to their upper bound. Clearly, $d = 0, 1, \ldots, c$, $s = 0, 1, \ldots, \Gamma$, and $j = 0, 1, \ldots, n$.

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

Γ-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 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. $\square$

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

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

The initialization values are $\overline{z}(d,s,0) = -\infty$ for $d = 0,\ldots,c$ and $s = 0,\ldots,\Gamma$. 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 $d < \hat{w}_j$ (resp. $d < w_j$) are not used in definition of $\overline{z}$ (resp. z) in recursion (5). The optimal solution value of the robust knapsack problem can be found as

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

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 $d = 0,\ldots,c$, $j = 0,\ldots,n$, $s = 0,\ldots,\Gamma$. 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

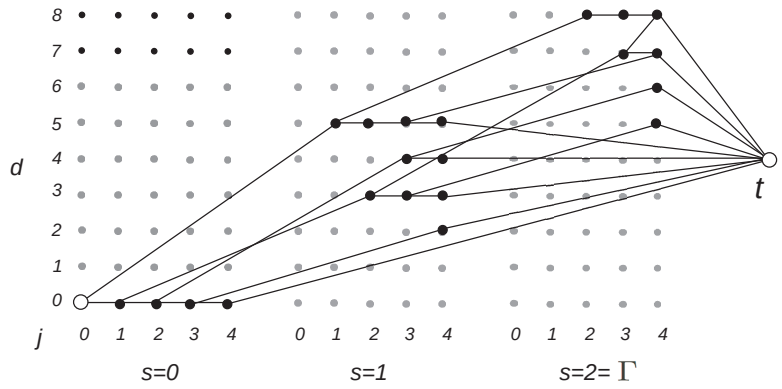

Figure 1: Directed graph model for the dynamic programming recursion.

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.

From stage $s-1$ to stage s there are arcs $(d - \hat{w}_j, s-1, j-1) \to (d, s, j)$, if $d \geq \hat{w}_j$, with value p_j. Let us denote these arcs as "heavy". Using a heavy arc corresponds to inserting item j and assuming it will take the upper weight $\hat{w}_j$.

Finally, within stage $s = \Gamma$ there are arcs $(d - w_j, \Gamma, j-1) \to (d, \Gamma, j)$, if $d \geq w_j$, with value p_j. Let us denote these arcs as "light". Using a light arc corresponds to inserting item j and assuming it will take its nominal weight.

Figure 1 shows an example of the graph associated with an instance with $n = 4, w = (1, 1, 3, 1)$, $\hat{w} = (5, 3, 4, 2), c = 8$ and $\Gamma = 2$. The source and the destination are the larger white nodes. The nodes in gray are non reachable and their arcs are not shown.

We claim that there is a one-to-one correspondence between paths from source to destination and feasible solutions.

Consider any path from node $(0, 0, 0)$ to node t. Note that this path moves from a stage to the next one only by heavy arcs. In particular, it will reach the final stage only after using exactly Γ heavy arcs. Note also that each path visits the nodes in increasing order of indices j. By construction, the indices of the light arcs (if any) are greater than the indices of the heavy arcs. Hence, by Lemma 1, the solution given by the path is feasible. Of course, this is true also in case a light arc is used to go from a node (d, s, n) to the destination t; indeed, in this case, only $s \leq \Gamma$ are inserted in the solution, all of them with increased weight $\hat{w}_j$.

As to the reverse, take any feasible solution J, and let $\hat{J}$ be its first Γ items if $|J| > \Gamma$, otherwise $\hat{J} = J$. As before j_Γ is the last index in $\hat{J}$. We build the path as follows: for each index $j \leq j_\Gamma$ we choose an empty arc if $j \notin J$ and a heavy arc if $j \in J$, until we reach the node $(\sum_{j \in \hat{J}} \hat{w}_j, |\hat{J}|, j_\Gamma)$. If $|J| \leq \Gamma$ we use an empty arc from this node till the destination node t. If $|J| > \Gamma$, then the node $(\sum_{j \in \hat{J}} \hat{w}_j, |\hat{J}|, j_\Gamma)$ is actually $(\sum_{j \in \hat{J}} \hat{w}_j, \Gamma, j_\Gamma)$, i.e., from this node we use

```
Solve_RKP(N, Γ, c)
begin
initialization
1.   for d := 0 to c do
2.       for s = 1 to Γ do
3.           z̄(d, s) = −∞
4.   z̄(0, 0) = 0;
5.   for j := 1 to |N| do
         possibly pack item j with nominal weight (if Γ items have already been inserted)
6.       for d := c down to w_j do
7.           if z̄(d − w_j, Γ) + p_j > z̄(d, Γ) then
8.               z̄(d, Γ) = z̄(d − w_j, Γ) + p_j;
         possibly pack item j with robust weight
9.       for s := Γ down to 1 do
10.          for d := c down to ŵ_j do
11.              if z̄(d − ŵ_j, s − 1) + p_j > z̄(d, s) then
12.                  z̄(d, s) = z̄(d − ŵ_j, s − 1) + p_j;
13.  z* = max{z̄(d, s) | d = 1, ..., c; s = 1, ..., Γ}
end
```

Figure 2: Dynamic programming algorithm for (RKP).

arcs within stage Γ for all $j > j_\Gamma$, in particular empty arcs if $j \notin J$ and light arcs if $j \in J$, until we reach the node $(\sum_{j\in\hat{J}} \hat{w}_j + \sum_{j\in J\setminus\hat{J}} w_j, \Gamma, n)$ and from this node we go to the destination.

Now the recursions (5) are exactly the optimality conditions for a longest path in the graph. □

3. Time and space complexity

In this section we analyze the time and space complexity of the dynamic programming algorithm resulting from (5). In many practical applications the space requirements are the main obstacle for using a dynamic programming algorithm. Hence, we will pay special attention to this issue.

A straightforward evaluation of the recursion (5) starts with array $\overline{z}$. In principle, we go through all items j from 1 to n in an outer loop. For each j, all cardinalities $s \leq \Gamma$ of a solution are considered, and in the inner-most loop all capacity values d from 0 to c are evaluated. Then array z is initialized with the outcome of $\overline{z}$ for $s = \Gamma$. Again, an outer loop goes over all items and an inner loop over all capacities. This would yield a pseudopolynomial running time of $\mathcal{O}(\Gamma\, n\, c)$ for computing the optimal *solution value*.

For the above recursion, at each iteration j only values from the previous

iteration $j-1$ are required. Thus we can construct a more refined implementation for computing z^* as given by procedure Solve_RKP in Figure 2. The above procedure applies a better utilization of the dynamic programming array and avoids copying array entries from one iteration to the next, yielding a space requirement $\mathcal{O}(n + \Gamma\, c)$ and a time complexity $\mathcal{O}(\Gamma\, n\, c)$. However, in this case it would not be possible to reconstruct the optimal solution set, but only the optimal solution value.

To obtain also the optimal *solution set* there are two straightforward possibilities. On one hand, one could store the dynamic programming arrays for all values of j, which allows a reconstruction of the solution set by backtracking; however, this procedure increases the space requirement by a factor n. On the other hand, one could store the current solution set for each entry of the dynamic programming array. An efficient approach would use a bit representation of the at most n items of each solution which yields a total space requirement of $\mathcal{O}(n + \Gamma\, c \log n)$. But in this case it should be noted that the computation of a new entry of the dynamic programming array requires copying a solution set from a previous entry. Such a transfer of n bits cannot be done in constant time but induces an increased running time of $\mathcal{O}(\Gamma\, n\, c \log n)$.

We will now present a more involved approach based on a recursive partitioning scheme. It is related to the general framework given in Pferschy [15] (see also Kellerer, Pferschy and Pisinger [10, Sec. 3.3]) but requires special considerations since the conditions for directly applying the framework of [15] are not met by (RKP).

The main idea is a bipartitioning of the item set $N = N_1 \cup N_2$ with $N_1 = \{1, \ldots, n/2\}$ and $N_2 = \{n/2 + 1, \ldots, n\}$ (assuming n to be even). After computing the optimal solution value for the whole set N we reconstruct the optimal solution set recursively for each set N_i.

We use the above recursion in a dynamic programming procedure Solve_RKP(c, Γ, N) which returns the optimal solution value of the robust knapsack problem for *every* capacity value $d = 1, \ldots, c$. In addition we also store for each array entry a counter $k(d, \Gamma)$ which indicates how many items in the corresponding solution set were taken from the first half of items, i.e. from N_1. It is trivial to update this counter whenever an item from N_1 is inserted during the dynamic programming recursion. Note that items are never removed from a solution set. The counter value associated to the optimal solution value z^* will be denoted by k^*.

In the recursion we will also use two non-robust variants of the knapsack problem. On one hand we will use the standard knapsack problem (KP) introduced above. It is well known that the optimal solution set of (KP) can be computed in $\mathcal{O}(n\, c)$ time and $\mathcal{O}(n + c)$ space by dynamic programming (see e.g. Kellerer, Pferschy and Pisinger [10]). On the other hand we will use a cardinality constrained knapsack problem (E-kKP), where a strict cardinality bound k is added to (KP) such that

$$\sum_{j \in N} x_j = k. \tag{6}$$

It was shown by Caprara et al. [6] that the optimal solution set of (E-kKP) can be computed in $\mathcal{O}(k\,n\,c)$ time and $\mathcal{O}(n + k\,c)$ space by dynamic programming. Note that both of these algorithms also makes use of the recursive partitioning scheme from Pferschy [15]. Moreover, they both compute the respective optimal solution values for all capacity values $d = 1, \ldots, c$.

Our partitioning argument is based on the following observation.

Lemma 3 *If $k^* \geq \Gamma$, then the optimal solution consists of the solutions of a robust knapsack problem with parameter Γ for N_1 and of an instance of (KP) for N_2 with nominal weights.*

Otherwise, if $k^* < \Gamma$, then the optimal solution consists of the solutions of an instance of (E-kKP) with parameter k^* and increased weights $\hat{w}_j$ for N_1 and a robust knapsack problem with parameter $\Gamma - k^*$ for N_2.

Proof. It was pointed out in Lemma 1 that in any feasible solution, and thus also in an optimal solution, the Γ items with lowest index, i.e. indices up to j_Γ, are those which contribute an increased weight to the capacity; all remaining items with higher indices contribute their nominal weight.

If $k^* \geq \Gamma$, then $j_\Gamma \in N_1$ and all items in the optimal solution belonging to N_2 (if any) contribute only their nominal weight. If $k^* < \Gamma$, then $j_\Gamma \in N_2$ and all k^* items in the optimal solution belonging to N_1 contribute their increased weight. The remaining at most $\Gamma - k^*$ weight increases are contributed by items in N_2. Note that this second case includes the possibility that the optimal solution contains less than Γ items. □

Based on Lemma 3 an algorithm for the robust knapsack problem is presented in Figure 3.

Theorem 4 *The recursive partitioning algorithm of Figure 3 to compute an optimal solution of (RKP) can be performed in $\mathcal{O}(\Gamma\,n\,c)$ time and $\mathcal{O}(n + \Gamma c)$ space.*

Proof. Using the dynamic programming scheme described above each call to $\mathsf{Solve\_RKP}(c, \Gamma, N)$ requires $\mathcal{O}(\Gamma\,n\,c)$ time and $\mathcal{O}(n + \Gamma\,c)$ space. Moreover, summarizing the above discussion we can also perform each call to $\mathsf{recursion}(z^*, k^*, c^*, \Gamma, N)$ in $\mathcal{O}(\Gamma\,|N|\,c^*)$ time: Indeed, the main effort in the recursion for $k^* \geq \Gamma$ is the execution of $\mathsf{Solve\_RKP}(c^*, \Gamma, N_1)$ requiring $\mathcal{O}(\Gamma\,\frac{|N|}{2}c^*)$ time and the solution of an instance of (KP) with item set N_2 which requires only $\mathcal{O}(\frac{|N|}{2}c^*)$ time. For $k^* < \Gamma$ we have to solve an instance of (E-kKP) in $\mathcal{O}(\Gamma\frac{|N|}{2}c^*)$ time and execute $\mathsf{Solve\_RKP}(c^*, \Gamma - k^*, N_2)$ also in $\mathcal{O}(\Gamma\frac{|N|}{2}c^*)$ time. To report the solutions sets for (KP) resp. (E-kKP) we have to rerun the respective algorithms with capacity c_1 resp. c_2 since they compute the optimal solution sets only for the given capacity value c^* but not for all capacity values. For both cases finding the combination $c_1 + c_2 = c^*$ can be done by a simple pass through the dynamic programming arrays.

begin
1. call Solve_RKP(c, Γ, N) and determine z^*, c^* and k^*;
2. call recursion$(z^*, k^*, c^*, \Gamma, N)$;

end

recursion$(z^*, k^*, c^*, \Gamma, N)$
begin
0. **if** $|N| = 1$ **then** output trivial solution.
1. partition N into $N_1 = \{1, \ldots, \lceil |N|/2 \rceil\}$ and $N_2 = \{\lceil |N|/2 \rceil + 1, \ldots, |N|\}$;
2. **if** $k^* \geq \Gamma$ **then**
3. call Solve_RKP(c^*, Γ, N_1) returning $z_1(c^*)$;
4. solve (KP) with N_2, c^* and weights w_j returning $z_2(c^*)$;
5. find a combination $c_1 + c_2 = c^*$ such that $z_1(c_1) + z_2(c_2) = z^*$;
6. output the solution set of (KP) for $z_2(c_2)$;
7. let k_1^* be the counter associated to $z_1(c_1)$;
8. recursion$(z_1(c_1), k_1^*, c_1, \Gamma, N_1)$
9. **else**
10. solve (E-kKP) with parameter k^*, N_1, c^* and weights $\hat{w}_j$ returning $z_1(c^*)$;
11. call Solve_RKP$(c^*, \Gamma - k^*, N_2)$ returning $z_2(c^*)$;
12. find a combination $c_1 + c_2 = c^*$ such that $z_1(c_1) + z_2(c_2) = z^*$;
13. output the solution set of (E-kKP) for $z_1(c_1)$;
14. let k_2^* be the counter associated to $z_2(c_2)$;
15. recursion$(z_2(c_2), k_2^*, c_2, \Gamma - k^*, N_2)$
16. **endif**

end.

Figure 3: Recursive partitioning algorithm to find the optimal solution set of (RKP).

Summing up the running time over all recursion levels we get a total running time of

$$\Gamma\, n\, c + \sum_{i=0}^{\log n} \Gamma\, n/2^i\; c \leq 3\, \Gamma\, n\, c$$

which proves the claimed time complexity.

The space requirements are trivially bounded by $\mathcal{O}(|N| + \Gamma\, c)$. Note that before each call to recursion all previously used space can be set free and thus reused in the next recursion level. □

Note that the results by Bertsimas and Sim [2, 3] (see also Section 4.2) ensure that (RKP) can also be solved to optimality by solving $n+1$ nominal knapsack problems (KP), see Section 4.2. Since (KP) can be solved in $\mathcal{O}(n\, c)$ time and

$\mathcal{O}(n+c)$ space this approach would require only $\mathcal{O}(n+c)$ space, but $\mathcal{O}(n^2\,c)$ time.

4. Other exact approaches to (RKP)

In this section we review some other algorithms for the optimal solution of (RKP).

4.1 A compact MILP-formulation

The robust counterpart of an uncertain ILP was formulated by Bertsimas and Sim [3] as a Mixed ILP (MILP) by adding a polynomial number of variables and constraints. For the special case of (KP), the associated robust counterpart looks as follows:

$$\begin{array}{lll} \max & \displaystyle\sum_{j\in N} p_j\,x_j & \\ & \displaystyle\sum_{j\in N} w_j\,x_j + \sum_{j\in N}\pi_j + \Gamma\rho \le 1 & \\ & -\,\overline{w}_j\,x_j + \pi_j + \rho \ge 0 & j\in N \\ & x_j\in\{0,1\},\ \pi_j\ge 0,\ \rho\ge 0 & j\in N. \end{array} \tag{7}$$

The resulting model, referred to as BS_{MILP} in the following, involves n binary and $n+1$ continuous variables, and $n+1$ constraints. Hence, from a theoretical point of view, enforcing robustness does not increase the computational complexity of the problem.

4.2 Iterative solution of the nominal problem

A different approach for a special class of ILPs was presented again by Bertsimas and Sim [2]. This algorithm applies to those problems in which all variables are binary and only the coefficients in the objective function are subject to uncertainty.

In this case, it is enough to observe that an optimal solution exists in which variable ρ takes a value from a set of at most $n+1$ candidates; once ρ is fixed, the resulting problem is a nominal problem in which some coefficients are changed. Hence, one can guess the "correct" value of ρ, solve $n+1$ nominal problems and take the best of the associated solutions as the optimal robust solution. It is easy to see that, if one swaps the role of the objective function with that of the uncertain constraint, (RKP) immediately fits within these settings, which allows to adopt the scheme above. By solving $n+1$ nominal (KP) instances, obtained by suitably changing the items' weights, one yields the optimal solution of a given (RKP) instance. The algorithm above requires $\mathcal{O}(n\,T)$ time, where $\mathcal{O}(T)$ denotes the computing time for solving the nominal (KP).

4.3 Branch-and-cut algorithm

Fischetti and Monaci [7] noted that robustness, defined according to the definition of Bertsimas and Sim [2], can

be enforced with no need of introducing additional variables, by working on the space of the original x_j variables. To do this, one has to separate some *robustness cuts*, that, for the knapsack problem, have the following structure

$$\sum_{j \in N} w_j \, x_j + \sum_{j \in S} \overline{w}_j \, x_j \leq c, \qquad \forall S \subseteq N : |S| \leq \Gamma. \tag{8}$$

For a general MILP, given the current solution x^*, the separation of robustness cuts associated with a given row requires $\mathcal{O}(n)$ if the x^* is integer; otherwise separation can be carried out in $\mathcal{O}(n \log n)$ time.

Note that the formulation of Section 4.1 and this Branch-and-Cut model have the same lower bounds given by the LP relaxations as can be seen from [3].

5. FPTAS and Robust Knapsack

Approximation algorithms are an obvious alternative to the computation of exact solutions. In particular, the construction of a fully polynomial approximation scheme (FPTAS) for (RKP) is an interesting issue.

Note that the recursion (5) could be alternatively stated by exchanging the roles of profit and weights. Without lengthy explanations we briefly state the resulting recursion. Let $y(p, s, j)$ be the smallest weight for a feasible solution with total profit p in which only items in $\{1, \ldots, j\} \subseteq N$ are considered and exactly s of them are included, all with their upper weight bound $\hat{w}_j$. Let $y(p, j)$ be the smallest weight for a feasible solution with total profit p in which only items in $\{1, \ldots, j\} \subseteq N$ are considered and Γ of them change from their nominal weight to their upper bound.

The entries of the dynamic programming array can be computed in analogy to (5).

$$\begin{aligned} y(p, s, j) = {} & \min\{y(p, s, j-1),\, y(p - p_j, s - 1, j - 1) + \hat{w}_j\} \\ & \text{for } p = 0, \ldots, p(N), s = 1, \ldots, \Gamma, j = 1, \ldots, n, \\ y(p, j) \quad = {} & \min\{y(p, j-1),\, y(p - p_j, j - 1) + w_j\} \\ & \text{for } p = 0, \ldots, p(N), j = \Gamma + 1, \ldots, n \end{aligned} \tag{9}$$

with the obvious initializations. Then the optimal solution value is given by $\max\{p \mid y(p, n) \leq c\}$. The straightforward running time of this approach would be $\mathcal{O}(\Gamma \, n \sum_{j \in N} p_j)$.

Since the running time is now pseudopolynomial in the sum of profits, one can apply a standard scaling argument (cf. [10, Sec. 2.6]) to obtain an FPTAS from our dynamic programming scheme. Indeed, using scaled profits $\tilde{p}_j := \lfloor p_j \, n / (\varepsilon \, p_{\max}) \rfloor$, where $p_{\max}$ is the largest profit of an item, one obtains

an FPTAS for (RKP) with running time $\mathcal{O}(\Gamma\, n^3/\varepsilon)$ (neglecting the details of retrieving the corresponding solution set, cf. Section 3).

However, one could also use the iterative method from Section 4.2 to reach an FPTAS. Indeed, the iterative method can be applied also with any approximation algorithm for (KP). Again, $n+1$ iterations yield a feasible solution for (RKP) and preserve the approximation ratio of the algorithm for the nominal problem. Therefore, one can take advantage of the highly tuned FPTAS for (KP) (see Kellerer and Pferschy [8, 9]) with a running time complexity of $\mathcal{O}(n \log n + 1/\varepsilon^3 \log^2(1/\varepsilon))$ and obtain an FPTAS for (RKP) with $\mathcal{O}(n^2 \log n + n \cdot 1/\varepsilon^3 \log^2(1/\varepsilon))$ running time. This will dominate the above approach for most reasonable parameter settings.

Of course, it would be possible to apply some of the technical features of [8, 9] to speed up the FPTAS arising directly from the dynamic program for (RKP). However, it seems that this is a futile exercise with little hope for an overall improved running time complexity.

6. Computational experiments

In this section we computationally evaluate, on a large set of instances, the dynamic programming algorithm of Section 2 and the other exact algorithms for (RKP) described in Section 4. To the best of our knowledge, no (RKP) instances were proposed so far in the literature, but those used by Klopfenstein and Nace [12]; as already mentioned, these are however quite small instances (up to 20 items) that cannot be used to compare dynamic programming with the algorithms of Section 4.

Thus, we randomly generated the problems in our testbed in the following way.

Test instances

In order to produce hard (RKP) instances, we first generated hard (KP) instances and then defined the robust weight of each item accordingly. According to Pisinger [16], hard (KP) instances were obtained by taking profits and weights as independent uniformly distributed values in a given interval; in particular, the following 5 classes of (KP) instances were generated:

- UN, uncorrelated instances: p_j and w_j are integer values randomly chosen in $[1, c]$;
- WC, weakly correlated instances: w_j values are integer values randomly generated in $[1, c]$, and each p_j is chosen among the integers in $[\max\{1, w_j - c/10\}, w_j + c/10]$;
- SC, strongly correlated instances: each weight w_j is an integer value randomly chosen in $[1, c]$ and the associated profit is $p_j = w_j + c/10$;

- IC, inverse strongly correlated instances: each profit p_j is an integer value randomly chosen in $[1, c]$ and the associated weight is $w_j = \min\{c, p_j + c/10\}$;
- SS, subset sum instances: weights w_j are integer values randomly generated in $[1, c]$ and $p_j = w_j$.

To test the performances of the algorithms with problems of different sizes, we generated instances with $c = 100$ and $n \in \{100, 500, 1000, 5000\}$, and instances with $n = 5000$ and $c \in \{1000, 5000\}$. For each combination of the parameters, 10 instances were generated; thus our testbed includes 300 (KP) instances.

As to uncertainty, in all the instances the robust weight $\overline{w}_j$ of each item j was randomly generated in $[w_j, c]$. Finally, we considered different values of Γ, namely $\Gamma \in \{1, 10, 50\}$.

All instances (and the corresponding solutions) are available at `http://www.or.deis.unibo.it/research.html`.

Algorithms

We compared the following exact approaches for (RKP):

- BS_{MILP}, i.e., the compact formulation (7) for (RKP) as proposed by Bertsimas and Sim [3].
- BS_2, i.e., the approach proposed by Bertsimas and Sim [2] consisting in the iterative solution of nominal (KP) instances. To solve nominal (KP) instances, we run as a black box the publicly available routine `combo` by Martello, Pisinger and Toth [13], which is actually considered the state-of-the-art for the exact solution of (KP) problems.
- BC, i.e., the branch-and-cut algorithm to robust optimization recently proposed by Fischetti and Monaci [7]; in our implementation, separation is carried out at each node of the enumerative tree.
- DP, i.e., the dynamic programming algorithm Solve_RKP presented in Section 3, but without the recursive scheme for space reduction.

All algorithms were coded in C language and run on a PC Intel Q6600 CPU@2.40GHz with 4GB RAM. For solving both MILP (7) and all LPs during enumerative algorithm BC, the commercial solver IBM-ILOG Cplex 11.2 was used. For each instance, a time limit of 300 CPU seconds was given to each algorithm. For some classes of instances the CPU times are very small; thus, in order to obtain a reliable estimate, it was necessary to compute, for each instance, the CPU time spent while solving it 100 times and correspondingly divide the total time.

Results

Table 1 reports the outcome of our experiments on the 300 instances of our testbed for the 3 different values of Γ. In particular, each line in the table refers to a set of 10 instances, characterized by number of items n, capacity c and type $t \in \{UN, WC, SC, IC, SS\}$, and reports, for each value of Γ, the average computing time (arithmetic mean, in seconds) required by each algorithm (for unsolved instances, we counted a computing time equal to the time limit).

Computational results show that our dynamic programming algorithm is very effective in solving (RKP) instances where a small number of coefficients change, as its complexity grows linearly with Γ. For larger values of Γ the iterative solution of (KP) instances, as proposed by Bertsimas and Sim [2], turns out to be the best algorithmic approach. However, the computing time of BS_2 may vary in a substantial way when solving instances of different type – although of the same size – which is not the case for algorithm DP, whose computing time is more stable. This is particularly evident when instances with $c = 5000$ are considered.

The two algorithms above outperform the remaining two algorithms, BS_{MILP} and BC. This is not surprising, as the latter are based on the use of a general purpose ILP solver, which is usually less efficient than ad hoc algorithms for knapsack problems. In addition, cutting planes are not the best suitable way to enforce robustness when problems with integer variables are considered, as noted by Fischetti and Monaci [7], which leads to a number of unsolved instances in our testbed.

7. Conclusions

In this paper we considered the robust knapsack problem, i.e., the uncertain variant of the well-known knapsack problem in which robustness is enforced according to the Bertsimas-Sim definition (see, [3]). For this problem we presented a dynamic programming algorithm and studied its time and space complexities, as well as techniques aimed at reducing the space complexity. We then computationally tested this algorithm on a large set of randomly generated instances, and compared the performances of the algorithm with those of other exact approaches proposed in the literature for (RKP). Computational results showed that dynamic programming is a viable way for solving (RKP) even for large-sized instances.

References

[1] D. Bertsimas, D.B. Brown, and C. Caramanis. Theory and applications of robust optimization. *SIAM Review*, 53:464–501, 2011.

[2] D. Bertsimas and M. Sim. Robust discrete optimization and network flows. *Mathematical Programming*, 98:49–71, 2003.

instances			$\Gamma = 1$				$\Gamma = 10$				$\Gamma = 50$			
n	c	type	BS_{MILP}	BS_2	BC	DP	BS_{MILP}	BS_2	BC	DP	BS_{MILP}	BS_2	BC	DP
100	100	UN	0.022	0.000	0.016	0.000	0.014	0.000	0.031	0.000	0.009	0.000	0.010	0.000
100	100	WC	0.026	0.000	0.038	0.000	0.011	0.000	0.019	0.000	0.009	0.000	0.009	0.000
100	100	SC	0.036	0.000	0.050	0.000	0.025	0.000	0.082	0.000	0.014	0.000	0.034	0.000
100	100	IC	0.002	0.000	0.001	0.000	0.002	0.000	0.001	0.000	0.002	0.000	0.000	0.000
100	100	SS	0.008	0.000	0.046	0.000	0.003	0.000	0.017	0.000	0.003	0.000	0.005	0.000
500	100	UN	0.177	0.002	0.144	0.000	0.104	0.001	1.632	0.000	0.037	0.001	0.737	0.001
500	100	WC	0.217	0.002	0.434	0.000	0.080	0.001	0.451	0.000	0.044	0.001	0.273	0.001
500	100	SC	0.290	0.002	0.416	0.000	0.155	0.001	8.109	0.000	0.041	0.001	3.517	0.001
500	100	IC	0.009	0.002	0.003	0.000	0.009	0.001	0.001	0.000	0.009	0.001	0.002	0.001
500	100	SS	0.170	0.001	3.441	0.000	0.034	0.001	0.893	0.000	0.034	0.001	0.692	0.001
1000	100	UN	0.526	0.003	0.527	0.000	0.288	0.002	14.682	0.001	0.077	0.002	6.779	0.002
1000	100	WC	0.541	0.003	0.952	0.000	0.232	0.002	2.900	0.001	0.100	0.002	1.733	0.002
1000	100	SC	1.255	0.003	2.395	0.000	0.443	0.002	94.304	0.001	0.106	0.002	82.292	0.003
1000	100	IC	0.023	0.003	0.003	0.000	0.023	0.002	0.003	0.001	0.023	0.002	0.002	0.002
1000	100	SS	0.056	0.002	18.346	0.000	0.061	0.002	3.663	0.001	0.058	0.001	3.524	0.003
5000	100	UN	4.584	0.012	14.873	0.001	2.183	0.011	300.000	0.004	0.574	0.011	300.000	0.012
5000	100	WC	5.219	0.012	10.831	0.001	2.664	0.011	300.000	0.004	0.671	0.010	262.481	0.012
5000	100	SC	33.573	0.013	110.717	0.001	3.152	0.011	300.000	0.004	0.696	0.011	300.000	0.012
5000	100	IC	0.056	0.017	0.014	0.001	0.056	0.009	0.013	0.003	0.056	0.008	0.015	0.010
5000	100	SS	0.289	0.010	148.443	0.001	0.273	0.007	50.436	0.004	0.272	0.007	42.420	0.012
5000	1000	UN	12.584	0.120	21.089	0.015	3.975	0.097	300.000	0.032	0.850	0.094	300.000	0.099
5000	1000	WC	12.916	0.119	27.905	0.015	3.118	0.094	300.000	0.033	0.982	0.091	300.000	0.099
5000	1000	SC	59.797	0.141	135.139	0.015	10.107	0.097	300.000	0.035	1.702	0.094	300.000	0.101
5000	1000	IC	0.059	0.153	0.014	0.014	0.060	0.089	0.014	0.027	0.060	0.073	0.014	0.081
5000	1000	SS	1.556	0.084	300.000	0.016	0.813	0.068	300.000	0.035	0.719	0.063	300.000	0.101
5000	5000	UN	12.715	0.381	19.629	0.073	2.965	0.291	300.000	0.156	1.193	0.276	300.000	0.474
5000	5000	WC	8.477	0.380	23.695	0.073	3.686	0.284	300.000	0.158	1.057	0.268	300.000	0.476
5000	5000	SC	99.400	0.544	191.060	0.073	6.609	0.282	300.000	0.164	1.865	0.264	300.000	0.483
5000	5000	IC	0.060	0.836	0.014	0.068	0.060	0.316	0.014	0.129	0.060	0.213	0.015	0.389
5000	5000	SS	2.342	0.423	270.010	0.080	0.634	0.349	300.000	0.165	0.670	0.218	300.000	0.482
avg			8.566	0.109	43.341	0.015	1.395	0.068	115.909	0.032	0.400	0.057	113.485	0.095

Table 1: Computational experiments on random (RKP) instances

[3] D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52:35–53, 2004.

[4] C. Büsing, A.M.C.A. Koster, and M. Kutschka. Recoverable robust knapsacks: Γ-scenarios. In *Proceedings of INOC 2011, International Network Optimization Conference*, Lecture Notes in Computer Science 6701, pages 583–588. Springer, 2011.

[5] C. Büsing, A.M.C.A. Koster, and M. Kutschka. Recoverable robust knapsacks: the discrete scenario case. *Optimization Letters*, 5:379–392, 2011.

[6] A. Caprara, H. Kellerer, U. Pferschy, and D. Pisinger. Approximation algorithms for knapsack problems with cardinality constraints. *European Journal of Operational Research*, 123:333–345, 2000.

[7] M. Fischetti and M. Monaci. Cutting plane versus compact formulations for uncertain (integer) linear programs. *Mathematical Programming Computation*, 4:239–273, 2012.

[8] H. Kellerer and U. Pferschy. A new fully polynomial approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3:59–71, 1999.

[9] H. Kellerer and U. Pferschy. Improved dynamic programming in connection with an FPTAS for the knapsack problem. *Journal of Combinatorial Optimization*, 8:5–12, 2004.

[10] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[11] O. Klopfenstein and D. Nace. A robust approach to the chance-constrained knapsack problem. *Operations Research Letters*, 36:628–632, 2008.

[12] O. Klopfenstein and D. Nace. Cover inequalities for robust knapsack sets - application to the robust bandwidth packing problem. *Networks*, 59:59–72, 2012.

[13] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 3:414–424, 1999.

[14] M. Monaci and U. Pferschy. On the robust knapsack problem. submitted manuscript, available as Optimization Online 2011-04-3019, 2012.

[15] U. Pferschy. Dynamic programming revisited: Improving knapsack algorithms. *Computing*, 63:419–430, 1999.

[16] D. Pisinger. Where are the hard knapsack problems? *Computers and Operations Research*, 32:2271–2284, 2005.