The binary knapsack problem

Ghosh, Diptesh; Goldengorin, Boris

The Binary Knapsack Problem: Solutions with Guaranteed Quality

Diptesh Ghosh and Boris Goldengorin

SOM-theme A Primary Processes within Firms

Abstract

Binary knapsack problems are some of the most widely studied problems in combinatorial optimization. Several algorithms, both exact and approximate are known for this problem. In this paper, we embed heuristics within a branch and bound framework to produce an algorithm that generates solutions with guaranteed quality within very short times. We report computational experiments that show that for the more difficult strongly correlated problems, our algorithm can generate solutions within 0.01% of the optimal solution in less than 10% of the time required by exact algorithms.

Keywords: binary knapsack, accuracy parameter, branch and bound, local search

(also downloadable) in electronic version: `http://som.rug.nl`

1. Introduction

In a binary knapsack problem (BKP), we are given a set $E = \{e_j\}$ of n elements and a knapsack of 'weight capacity' c. Each element e_j has a 'profit' p_j and a 'weight' w_j. Our objective is to find the most profitable solution, i.e. subset of elements of E, that can be put in the knapsack without violating its weight capacity. The profitability of a subset of E is defined as the sum of the profits of the elements in the subset. If we denote the decision of including (or excluding) an element e_j in the knapsack by setting a variable x_j to 1 (0, respectively) then each solution can be represented as a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, and BKP can be represented by the following mathematical program:

$$\text{BKP:} \quad z^\star = \max \left\{ P(\mathbf{x}) = \sum_{j=1}^{n} p_j x_j \,\middle|\, C(\mathbf{x}) = \sum_{j=1}^{n} w_j x_j \le c, x_j \in \{0,1\}, j = 1, \ldots, n \right\}.$$

$z^\star$ is called the *optimal profit* for the instance, and any solution $\mathbf{x}^\star = (x_1^\star, x_2^\star, \ldots, x_n^\star)$ with $P(\mathbf{x}^\star) = z^\star$ and $C(\mathbf{x}^\star) \le c$ is called an *optimal solution*. In this paper, we will assume that p_j, w_j, and c are positive, $w_j < c$, for $j = 1, \ldots, n$, and $C(\mathbf{1}) > c$. We also assume, without loss of generality, that the elements in E are ordered according to non-increasing profit to weight ratios, i.e. for any two elements $e_i,\ e_j \in E$, $\frac{p_i}{w_i} > \frac{p_j}{w_j} \implies i > j$, ties being broken arbitrarily.

BKP is among the most widely studied problems of discrete optimization, being of interest to both practitioners and theoreticians. Practical interest in this problem stems from the fact that many practical situations are either modelled as binary knapsack problems (for example, capital budgeting and cargo loading) or solve such problems as subproblems (for example, cutting stock problems). Theoretical interest arises from the fact that, although it is among the simplest discrete optimization problems to state, it is often quite difficult to solve.

It is well-known that the optimization version of the BKP is NP-hard (refer, for example, to Garey and Johnson [1]). Exact algorithms to solve it are therefore based on branch and bound, dynamic programming, or a hybrid of the two. Comprehensive overviews of the exact solution techniques for the binary knapsack problem are available in Martello, Pisinger, and Toth [3] and Martello and Toth [5]. Algorithms based on branch and bound have been traditionally preferred to those based on dynamic programming because the latter require a lot of computer memory. Recently however, algorithms using dynamic programming have been used to solve large and difficult instances of the BKP (see, for example, algorithm `combo` in Martello, Pisinger, and Toth [4]). Heuristics for BKP, which generate feasible (and usually suboptimal) solu-

tions within short execution times are also an active area of research (refer to Martello and Toth [5], and Ghosh [2] for a treatment on heuristics for the BKP). These heuristics usually perform very well in practice, and output solutions that are very close to optimal. However, their performance guarantee is usually based on their worst case performance ratios, which form a very weak bound on the deviation of the profit of the heuristic solution to that of the optimal solution for BKP instances. Moreover the bounds obtained from such ratios is not instance-specific.

In this paper we present an algorithm α-MT1 that aims to rectify the situation. It embeds a heuristic inside a branch and bound framework. This allows us to compute a-posteriori, an upper bound to the deviation of the heuristic solution from an optimal solution, in terms of profits. If the deviation observed is more than an allowable limit, a backtracking operation allows us to use the heuristic with additional constraints and generate better solutions. Thus, in addition to the profit vector, the weight vector, and the knapsack capacity, α-MT1 takes a *prescribed accuracy* parameter α as input. The algorithm then guarantees that the profit (z^α) of the solution it outputs would satisfy the expression

$$z^\star - z^\alpha \leq \alpha.$$

The term $z^\star - z^\alpha$ will henceforth be called the *achieved accuracy*. The branch and bound framework that we use is the well-known mt1 algorithm in Martello and Toth [6]. We use this algorithm instead of more sophisticated ones because it is a typical example of branch and bound based algorithms for the BKP, and is one of the simplest among such algorithms. Moreover mt1 is sufficient to demonstrate the behavior we are interested in. Notice that the prescribed accuracy parameter in α-MT1 is not expressed as a percentage (as is common in ε-approximate algorithms), but as an absolute value. This ensures that the deviation from the optimal profit can be controlled irrespective of the actual value of the optimal profit.

The remainder of the paper is organized as follows. In the next section we describe the algorithm α-MT1. Section 3 presents results from computations carried out on randomly generated BKP instances belonging to well-known classes studied in the literature. We conclude the paper in Section 4 where we summarize the findings in the paper and suggest directions for future research.

2. The α-MT1 Algorithm

The α-MT1 algorithm that we propose in this paper embeds a local search heuristic in the branch and bound framework of the MT1 algorithm (refer to Martello and Toth [6] for a detailed description of the MT1 algorithm.) In order to achieve this, we make two modifications to MT1. First, we incorporate the prescribed accuracy factor in the fathoming procedure. Consider a subset S of the set of all feasible solutions, $\mathcal{S}$. We first use a good and relatively fast heuristic $\mathcal{H}$ to obtain a good solution in S. We also compute an upper bound ub_S to the profit of the solutions in S. If the profit from the heuristic solution $z^{\mathcal{H}}$ satisfies the condition: $ub_S - z^{\mathcal{H}} \leq \alpha$, then we know that we have found a solution whose profit is within α of the profit of the best solution in S. Second, we use a stopping rule that can stop the algorithm before it considers all of the subsets of $\mathcal{S}$ that it generates. Let ub_{init} be an global upper bound to the optimal profit of the instance. If at any subset S of $\mathcal{S}$, the heuristic $\mathcal{H}$ produces a solution $z^{\mathcal{H}}$ satisfying the condition $z^{\mathcal{H}} \geq ub_{init} - \alpha$, then we can stop the computations immediately. This is because $ub_{init} \geq z^{\star}$, which immediately implies that $z^{\star} - z^{\mathcal{H}} \leq \alpha$.

The MT1 algorithm proposed in Martello and Toth [6] uses a depth first search strategy to explore subproblems. However, in our implementation of α-MT1, we follow a best-first search strategy. For exact algorithms, this strategy is known to produce an optimal solution after evaluating the least number of subproblems. However, it requires more memory than algorithms using depth-first search strategies. Our best-first search strategy requires us to maintain a list of subproblems (which we call LIST), and terminate the algorithm when the list is empty, or when our stopping rule is triggered. In the pseudocode of α-MT1 a subproblem is denoted by a partial solution $\mathbf{x} = (x_1, \ldots, x_n)$ defined on an alphabet $\{0, 1, \bullet\}$. $x_j = 0$ or 1 has its usual connotations regarding e_j, and $x_j = \bullet$ denotes that no decision has been made on whether or not to include e_j in the solution. A partial solution where each of the components are either 0 or 1 is called a *complete* solution. We present a pseudocode of the α-MT1 algorithm in Figure 1. It assumes the presence of three procedures: $ub(\mathbf{x})$, which returns an upper bound to the profit of the best solution from subproblem $\mathbf{x}$; $\mathcal{H}(\mathbf{x})$, which returns a feasible solution to the subproblem $\mathbf{x}$; and *forward-move*$(\mathbf{x})$, which performs a 'forward move' described in MT1. We describe these procedures in detail in the remaining portion of this section.

INSERT FIGURE 1 HERE

- $ub(\mathbf{x})$: Upper bounds are computed in α-MT1 in the following manner (based on Martello and Toth [6]):

Let
$\Pi(\mathbf{x}) = \sum_{j:x_j=1} p_j$, $c_r(\mathbf{x}) = c - \sum_{j:x_j=1} w_j$, $s(\mathbf{x}) = \min\{j : \sum_{i=1,x_i=\bullet}^{j} w_i > c$, $s_-(\mathbf{x}) = \max\{j : x_j = \bullet, j < s(\mathbf{x})\}$, $s_+(\mathbf{x}) = \min\{j : x_j = \bullet, j > s(\mathbf{x})\}$, and $\overline{c}(\mathbf{x}) = c - \sum_{i=1,x_i=\bullet}^{s_-(\mathbf{x})} w_i$. Then

$$ub(\mathbf{x}) = \Pi(\mathbf{x}) + \sum_{i=1,x_i=\bullet}^{s_-(\mathbf{x})} p_i + \max\{\overline{c}(\mathbf{x})\frac{p_{s_+(\mathbf{x})}}{w_{s_+(\mathbf{x})}}, p_s - (w_s - \overline{c}(\mathbf{x}))\frac{p_{s_-}}{w_{s_-}}\}$$

is an upper bound to the optimal profit from the given instance.

- $\mathcal{H}(\mathbf{x})$: The heuristic $\mathcal{H}(\mathbf{x})$ is simply a local search heuristic with a 2-exchange neighborhood. It involves four steps. In the first step, it puts all the elements $e_j \in E$ with $x_j = 1$ in a set S, computes $c_r = c - \sum_{j:x_j=1} w_j$, and constructs a set E_r of elements $e_j \in E$ with $x_j = \bullet$. In the second step, it computes a greedy solution S_G by considering the elements $e_j \in E_r$ in the natural order and including them in a knapsack with weight capacity c_r whenever possible. The third step is a local search step which starts with S_G and improves it with local search using a 2-exchange neighborhood structure defined on the elements of E_r. The last step constructs the feasible solution output by $\mathcal{H}(\mathbf{x})$ by combining S and S_G.
- *forward-move*(**x**): The *forward-move* procedure in α-MT1 is identical to that in mt1. Let $j = \min\{j : x_j = \bullet\}$. We first construct the set N of the largest number of consecutive elements with $x_j = \bullet$ that we can include in the knapsack without exceeding the residual weight capacity $c_r = c - \sum_{j:x_j=1} w_j$. Set $x_j = 1$ for each j such that $e_j \in N$. If $\Pi(\mathbf{x}) + P(N) = ub(\mathbf{x})$, and this value is better than the profit of the best solution found so far, then we replace it by **x**, and direct α-MT1 to discard further search in the current subproblem. Otherwise, if $c - \sum_{j:x_j=1} w_j$ is less than the weight of any element in E, we carry out a dominance step, by which we try to replace the last element in N by two elements that are not in N. If the result of this dominance step is more profitable than the best solution found so far, then the best solution is updated. The *forward-move* procedure returns the modified **x** vector.

Notice that running α-MT1 with $\alpha = 0.0$ results in an optimal solution but makes α-MT1 run like mt1, while running α-MT1 with a large value of α makes it run like $\mathcal{H}(\mathbf{x})$, which in this case is local search with a 2-exchange neighborhood.

3. Computational Experiments

In this section we report our computational experience with the α-MT1 algorithm. We coded the algorithm in C, compiled it using the LCC compiler for Windows NT (due to Navia [7]), and ran it on a 733MHz Intel Pentium III machine with 128MB RAM.

We experimented with five different types of instances, viz.

Uncorrelated (UC) p_j and w_j values are uniformly and independently distributed in the interval $[L, H]$.

Weakly Correlated (WC) w_j values are uniformly and independently distributed in the interval $[L, H]$. For each j, p_j is uniformly random in $[w_j - 200, w_j + 200]$, so that $p_j > 0$.

Strongly Correlated (SC) w_j values are uniformly and independently distributed in the interval $[L, H]$. $p_j = w_j + 10$.

Inverse Strongly Correlated (ISC) p_j values are uniformly and independently distributed in the interval $[L, H]$. $w_j = p_j + 10$.

Almost Strongly Correlated (ASC) w_j values are uniformly and independently distributed in the interval $[L, H]$. For each j, p_j is uniformly random in $[w_j + 98, w_j + 102]$.

The weight capacity c for each of the instances was chosen to be $0.5 \sum_{j=1}^{n} w_j$. In Pisinger [8] it is shown that for UC type instances, setting $c = 0.35 \sum_{j=1}^{n} w_j$ generates instances that are most difficult for mt1. However, preliminary computations showed that the behavior of α-MT1 with $c = 0.35 \sum_{j=1}^{n} w_j$ was identical to the behavior when $c = 0.5 \sum_{j=1}^{n} w_j$ with respect to changes in α values. Thus, to maintain uniformity over all problem types, we chose $c = 0.5 \sum_{j=1}^{n} w_j$ for all problem types.

These instance types are similar to the ones used in Martello, Pisinger and Toth [4]. The only major class of instances mentioned in Martello, Pisinger and Toth [4] that we chose not to use in our computations are the class of even-odd problems. This is because we experiment with real-valued data, and 'even' and 'odd' concern integers only. Also, since we are concerned with approximate solutions, even-odd problems would invariably degenerate to instances very similar to the those of the strongly correlated class of problems.

As mentioned earlier, the data for each of the instances in our experiments are real-valued. This makes the instances more difficult to solve (since BKP is #P-Complete). For the UC and WC problem instances, we varied the size of the instances from 500 to 2000. For the other problems we varied the instance sizes from 50 to 1000. For each

instance size and instance type, we generated forty instances, divided into two sets of twenty instances each. In the first set, $\mathsf{L} = 1$ and $\mathsf{H} = 1000$. In the second set $\mathsf{L} = 1001$ and $\mathsf{H} = 2000$. Thus the spread of data in both the sets was the same, but the second set of problems did not contain any element with small weights, the presence of which often make the solution process easier.

We examined the behavior of α-MT1 in terms of the profit of the solution that it output, and the size of the BnB tree that it generated in order to solve instances corresponding to different instance sizes and values of α. (The size of a branch and bound tree is the number of nodes it contains.) mt1 is known to be able to solve moderate to large sized UC and WC type instances and is also known not to be able to solve any but small SC type instances (see, for example, Martello and Toth [5]). We took this behavior into account while designing our experiments. For the UC and WC type instances, we allowed α-MT1 to solve the instances exactly, and with α values of 5.0, 10.0, 15.0, 20.0, and 25.0. Given the data range for the instances, these α values each amount to less than 0.01% of the profit of an optimal solution. For SC, ISC, and ASC, we divided the instances into two categories, small and large. The small instances were of sizes varying from 50 to 150, and the large instances were of sizes varying from 200 to 1000. We allowed α-MT1 to solve the small instances exactly, and with α values of 5.0, 10.0, 15.0, 20.0, and 25.0. For the large instances, we computed the upper bound $\mathrm{ub}_{\mathrm{init}} = \mathrm{ub}(\ (\bullet, \bullet, \ldots, \bullet)\ )$ and used α values of 0.02%, 0.04%, 0.06%, 0.08%, 0.10%, and 0.12% of $\mathrm{ub}_{\mathrm{init}}$. α-MT1 was allowed a maximum execution time of 10 CPU minutes for each instance and each α value. We report the behavior of α-MT1 only for those sets where at least ten of the instances were solved within the given time for each α value.

For UC and WC type instances and small sized SC, ISC, and ASC instances, we could use α-MT1 to obtain optimal solutions. Thus we could compute the actual deviation of the solution output by α-MT1 from that of the optimal solution. But for large sized SC, ISC, and ASC instances, we could not obtain optimal solutions using α-MT1 within reasonable times. For these problems therefore, we measured deviations as a percentage of $\mathrm{ub}_{\mathrm{init}}$. These values therefore, form an upper bound to the actual deviations from the optimal profit for these instances. Different instances in the same problem set have widely different sizes of the BnB tree generated by α-MT1. So it is logical to present the size of the BnB tree generated for a certain α value as a percentage of the size of the tree generated for $\alpha = 0.0$. This is possible for UC and WC type instances and small sized SC, ISC, and ASC instances. For large sized SC, ISC, and ASC instances, we could not solve the instances with $\alpha = 0.0$; therefore we express the sizes of the BnB

trees as a percentage of the size of the trees generated when α is 0.02% of ub_{init}. Note however, that this difference makes it impossible to compare the percentage reductions in the sizes of the BnB trees for small and large instances of SC, ISC, and ASC type instances. Tables 4.1 through 4.10 present the results of our computational experiments.

INSERT TABLES 4.1 THROUGH 4.10 HERE

We now define two notations that will help to make the analysis of the results more readable. The first is $\Gamma(n, \alpha)$, which is the ratio of the achieved accuracy to the prescribed accuracy for instances of size n, and a prescribed accuracy parameter α. The second notation is $\Phi(n, \alpha, \alpha_0)$, which is the ratio of the size of the BnB tree for instances of size n and a prescribed accuracy parameter α to the size of the BnB tree for instances of size n and $\alpha = \alpha_0$.

For UC type instances (refer to Tables 4.1 and 4.2), the value of $\Gamma(n, \alpha)$ was always seen to be less than 0.5. Instances with data in the range $[1, 1000]$ had $\Gamma(n, \alpha)$ values that were almost constant for a given n, but increased when n increased. $\Phi(n, \alpha, 0.0)$ was also an almost linearly decreasing function of α for most of the instances that we studied. The slope of the decrease was initially steeper with increasing n, but when $\alpha = 25.0$ the slope was seen to decrease. At this stage, the size of the BnB trees for the largest instances were, on an average, approximately 6% of the size of the BnB tree when $\alpha = 0.0$. For instances with data in the range $[1001, 2000]$, $\Gamma(n, \alpha)$ increased with increasing with increasing α values as well as with increasing n values. The reduction in the size of the BnB trees for these problems was seen to be about half of the reduction observed for UC instances with the same n and α but with data in $[1, 1000]$.

For WC type instances (refer to Tables 4.3 and 4.4), the value of $\Gamma(n, \alpha)$ was seen to be almost constant with respect to changing α values. They were also seen to be less sensitive to changes in problem sizes. This behavior was valid when the data was in the range $[1, 1000]$ as well as when it was in the range $[1001, 2000]$. $\Phi(n, \alpha, 0.0)$ dropped rapidly to less than 20% when α was increased from 0.0 to 15.0. The $\Phi(n, \alpha, 0.0)$ values for instances where data was in the range $[1001, 2000]$ were seen to be about twice the $\Phi(n, \alpha, 0.0)$ values for instances where data was in the range $[1, 1000]$. An interesting feature is that the Φ values for both ranges were not sensitive to increases in α values when $\alpha \geq 15.0$. This, combined with the fact that $\Gamma(n, \alpha)$ values kept increasing when $\alpha \geq 15.0$, implies that increasing α values to more than 15.0 is not likely to improve the performance of α-MT1 on WC type instances.

The behavior of $\Gamma(n, \alpha)$ and $\Phi(n, \alpha, 0.0)$ for SC and ISC type instances (refer to Tables 4.5–4.8) were very similar to that for WC type instances. However, both these

types of instances were much more difficult to solve to optimality than WC type instances. In large instances of SC problems, the size of the BnB tree was fairly insensitive to increases in α values for $\alpha \geq 15.0$. The ISC type instances were observed to be extremely easy for α-MT1. Firstly, the $\Gamma(n, \alpha)$ values were close to 0.3 for these instances, where they were close to 0.5 for all the others that we have discussed so far. Also, when $\alpha \geq 10.0$, ISC instances led to very small BnB trees. For the large instances of ISC that we experimented with, in which data was drawn from the range $[1, 1000]$ we obtained a solution within the prescribed accuracy parameter at the root of the BnB tree for each of the instances with $n \geq 300$. Large ISC instances where the data was drawn from the range $[1001, 2000]$ were also easy to solve, and the sizes of the BnB trees for these instances did not vary with increasing α when $\alpha \geq 0.1ub_{init}$.

The behavior of α-MT1 on ASC instances (refer to Tables 4.9 and 4.10) were seen to be very different from those of the other instances that we experimented with. ASC instances where the data was in the range $[1001, 2000]$ were seen to be easier to solve than the instances where the data was in the range $[1, 1000]$. $\Gamma(n, \alpha)$ values were seen to be more sensitive to n than the WC, SC, and ISC type instances. Also, $\Phi(n, 25.0, 0.0)$ were seen to be more than 0.6 for most of these instances. (Other problem types usually had $\Phi(n, 25.0, 0.0)$ values close to 0.1.) The $\Phi(n, \alpha, 0.0)$ were seen to be increasing as n increased, and for larger instances, the size of the BnB tree was insensitive to increases in α when $\alpha \geq 0.05ub_{init}$. This means that α-MT1 would be less effective for larger sized ASC instances.

In summary, α-MT1 proved to be very efficient for most BKP instances, both in terms of the quality of solutions that it output and in terms of the reduction of size of the BnB tree during its execution. For most problems, the deviation of the solution output by α-MT1 was less than half the prescribed accuracy. In terms of the size of the BnB tree, for most of the problems, α-MT1 produced trees with around 10% of the number of nodes present in the BnB tree for mt1 when $\alpha \geq 15.0$. Considering the data ranges, $\alpha \geq 15.0$ implies a prescribed accuracy within 0.01%, making the reduction in the size of the BnB tree very impressive. The best results from α-MT1 were seen for ISC type of problems. This is partly because local search with 2-exchange neighborhoods are very effective for such problems. The worst results from α-MT1 were seen for ASC type of problems.

4. Summary and Discussions

In this paper, we present α-MT1, an algorithm to generate near-optimal solutions to binary knapsack problems, with bounds on the sub-optimality of the solution output. This algorithm embeds a local search based heuristic procedure within a branch and bound framework. As a result, α-MT1 is capable of producing a solution, whose profit is within a pre-specified amount of the profit of an optimal solution. Thus, the solutions generated by α-MT1 are insensitive to the actual numbers in the instance data for the problem. We tested the performance of α-MT1 on a wide variety of randomly generated knapsack instances belonging to types well-known in the literature (refer to Martello, Pisinger and Toth [4]). We observed that the algorithm performs well for all except the almost strongly correlated problem instances. In most cases we found out that the deviation achieved by α-MT1 was less than half the allowed deviation, and, when allowed a deviation of less than 0.01% of the profit of an optimal solution, solved problems in times that were an order of magnitude lower than the time required by exact algorithms. We chose the mt1 algorithm due to Martello and Toth [6] as the branch and bound algorithm on which we base our α-MT1 algorithm, since it is a typical branch and bound algorithm for binary knapsack problems. There are more sophisticated branch and bound algorithms, which could be used to solve larger problems more efficiently, and α-MT1-type algorithms could be devised based on such algorithms.

One of our current direction of research in this area is the following. In recent times, dynamic programming based algorithms are being proposed to solve instances of binary knapsack problems (refer, for example, to Martello, Pisinger and Toth [4]). These algorithms are shown to able to solve several classes of strongly correlated knapsack problems, which are traditionally difficult for pure branch and bound based algorithms. We are examining ways to incorporate ideas similar to the ones we propose here, into such dynamic programming based algorithms and obtain powerful algorithms for generating near-optimal solutions for a wider variety of binary knapsack problems.

Our other main area of research in this class of algorithms is to try to apply the concepts described here to find high-quality solutions to other hard combinatorial optimization problems such as facility location problems, quadratic cost partitioning, traveling salesperson problems, and scheduling problems.

References

[1] Garey MJ, Johnson DS. Computers and Intractability: A Guide to the Theory of NP-Completeness, San Francisco:Freeman, 1979.

[2] Ghosh D. Heuristics for Knapsack Problems: Comparative Survey and Sensitivity Analysis, Fellowship Dissertation, IIM Calcutta, India, 1997.

[3] Martello S, Pisinger D, Toth P. New trends in exact algorithms for the 0-1 knapsack problem, European Journal of Operational Research 2000;123:325–332.

[4] Martello S, Pisinger D, Toth P. Dynamic Programming and strong bounds for the 0-1 knapsack problem, Management Science 1999;45:414–424.

[5] Martello S, Toth P. Knapsack Problems: Algorithms and Computer Implementations, Chichester:John Wiley & Sons, 1990.

[6] Martello S, Toth P. An upper bound for the zero-one knapsack problem and a branch and bound algorithm, European Journal of Operational Research 1977;1:169–175.

[7] Navia J. LCC-Win32: a compiler system for Windows 95 – NT, `http://www.cs.virginia.edu/~lcc-win32/`.

[8] Pisinger D. Core problems in knapsack algorithms, Operations Research 1999;47:570–575.

Algorithm α-MT1
Input: Instance $I = \{(p_1, \ldots, p_n), (w_1, \ldots, w_n), c\}$, prescribed accuracy α.
Output: A solution to I within the prescribed accuracy α.
Code:

```
01 begin
02    ub_init ← ub( (•, •, ..., •) );
03    BestSolutionSoFar ← ∅;
04    BestSolutionValue ← −∞;
05    LIST ← { (•, •, ..., •) };
06    while LIST ≠ ∅ do
07    begin
08       Choose a subproblem x = (x_j) from LIST;
09       x^H = (x_j^H) = H(x);
10       if ub_init − P(x^H) ≤ α then                    (* New Stopping Rule *)
11          return x^H and stop;
12       ub_x ← ub(x)
13       if ub_x ≤ BestSolutionValue then                (* Discard this subproblem *)
14          goto 30;
15       if ub_x − P(x^H) ≤ α then     (* x^H is within the prescribed accuracy α *)
16       begin
17          Update BestSolutionSoFar and BestSolutionValue if necessary;
18          goto 30;
19       end;
20       x ← forward-move(x);
21 (* Creating new subproblems by branching *)
22       k ← min{j : x_j = •};
23       x^new ← x;
24       x_k^new ← 0;
25       LIST ← LIST ∪ {x^new};
26       x^new ← x;
27       x_k^new ← 1;
28       if C(x^new) ≤ c then
29          LIST ← LIST ∪ {x^new};
30    end;
31    return BestSolutionSoFar;
32 end.
```

Figure 1: Pseudocode of α-MT1.

Table 4.1: Performance of α-MT1 on UC knapsack instances (Data range $[1, 1000]$)

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
500	0.000	0.000	0.389	0.389	0.727	2.647	100.000	88.321	86.550	76.625	54.199	18.689
750	0.000	0.000	0.625	1.212	3.294	6.431	100.000	97.775	86.667	67.818	47.373	29.133
1000	0.000	0.000	0.094	1.919	4.119	4.231	100.000	94.908	60.896	29.807	22.506	21.326
1250	0.000	0.000	1.222	3.747	6.566	12.972	100.000	94.195	68.679	42.988	30.814	25.745
1500	0.000	0.000	1.719	2.731	9.944	12.262	100.000	87.058	60.616	41.562	20.086	14.721
1750	0.000	0.148	1.516	3.180	6.234	8.656	100.000	79.913	42.182	32.803	22.787	20.542
2000	0.000	0.050	2.938	4.588	8.588	12.538	100.000	89.807	34.000	21.927	12.333	6.071

Table 4.2: Performance of α-MT1 on UC knapsack instances (Data range $[1001, 2000]$)

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
500	0.000	0.000	0.000	0.003	0.084	0.084	100.000	99.511	98.000	95.049	82.485	70.642
750	0.000	0.000	0.000	0.181	0.938	3.138	100.000	99.273	96.125	85.448	80.657	61.318
1000	0.000	0.000	0.300	0.456	2.281	5.600	100.000	98.855	99.597	71.475	55.597	44.650
1250	0.000	0.000	0.113	2.400	4.713	7.350	100.000	98.691	88.002	51.925	43.996	41.266
1500	0.000	0.000	0.313	2.225	3.875	8.438	100.000	98.388	85.543	53.237	45.240	37.245
1750	0.000	0.000	0.400	2.513	6.213	12.363	100.000	98.276	75.723	58.043	47.007	34.132
2000	0.000	0.000	1.143	3.036	4.875	5.893	100.000	95.136	73.743	65.095	63.558	61.081

Table 4.3: Performance of α-MT1 on WC knapsack instances (Data range $[1, 1000]$)

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
500	0.000	0.277	2.778	7.219	8.659	10.609	100.000	73.182	27.142	8.123	6.125	4.267
750	0.000	0.644	3.386	6.534	7.412	8.567	100.000	49.321	13.058	3.739	3.046	2.633
1000	0.000	1.544	3.881	6.906	7.484	9.619	100.000	38.820	11.281	4.201	2.890	0.335
1250	0.000	1.613	4.072	7.444	7.719	7.719	100.000	27.397	8.249	0.545	0.137	0.098
1500	0.000	1.975	5.659	7.334	11.294	13.359	100.000	22.893	3.548	1.392	0.384	0.002
1750	0.000	2.103	5.081	8.197	9.516	12.144	100.000	19.534	2.333	0.784	0.418	0.001
2000	0.000	2.019	4.475	7.662	8.588	8.287	100.000	20.164	1.820	1.017	0.416	0.261

Table 4.4: Performance of α-MT1 on WC knapsack instances (Data range $[1001, 2000]$)

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
500	0.000	0.122	1.422	6.666	7.556	11.713	100.000	85.973	44.934	18.356	15.540	13.972
750	0.000	0.231	4.138	7.025	7.931	9.137	100.000	72.241	25.480	14.307	12.832	12.639
1000	0.000	0.850	2.694	3.313	3.313	6.075	100.000	31.695	10.308	9.683	9.481	9.415
1250	0.000	0.719	2.819	5.588	8.012	8.012	100.000	32.220	16.141	13.384	12.889	12.889
1500	0.000	1.563	3.750	6.950	6.950	9.175	100.000	49.092	32.626	29.163	29.163	28.318
1750	0.000	1.431	4.042	6.778	10.514	10.694	100.000	24.534	12.345	11.717	8.967	7.625
2000	0.000	1.729	3.667	7.750	8.938	10.250	100.000	30.524	14.483	13.526	13.395	12.266

Table 4.5: Performance of α-MT1 on SC knapsack instances (Data range $[1, 1000]$)

(a) Smaller sized instances with accuracy provided in absolute values.

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
50	0.000	0.406	1.819	4.552	7.629	9.376	100.000	72.004	14.800	4.930	4.427	4.364
75	0.000	0.479	1.905	3.924	6.223	10.744	100.000	59.251	2.338	2.108	1.792	1.769
100	0.000	0.550	3.098	4.416	6.096	9.132	100.000	51.935	9.085	8.849	8.329	7.933
125	0.000	0.409	2.790	4.517	7.345	10.177	100.000	40.994	1.781	1.427	1.424	1.422
150	0.000	0.668	2.938	4.774	6.329	10.025	100.000	16.199	2.873	2.815	2.807	0.892

(b) Larger sized instances with accuracy provided in percentage values.

n	Percentage deviation from upper bound ub_{init}						Percentage of number of subproblems required(*)					
	α values (%)						α values (%)					
	0.02	0.04	0.06	0.08	0.10	0.12	0.02	0.04	0.06	0.08	0.10	0.12
200	0.014	0.023	0.031	0.043	0.053	0.065	100.000	81.823	80.623	74.566	71.950	69.685
300	0.015	0.031	0.046	0.060	0.076	0.089	100.000	92.837	75.865	63.941	51.392	39.616
400	0.014	0.027	0.040	0.053	0.063	0.079	100.000	85.221	66.528	35.582	25.161	20.310
500	0.011	0.019	0.029	0.039	0.051	0.057	100.000	71.532	54.443	41.022	19.523	5.773
600	0.013	0.026	0.036	0.048	0.057	0.064	100.000	89.530	72.915	49.894	25.707	4.816
700	0.017	0.033	0.048	0.061	0.072	0.093	100.000	51.787	39.106	12.422	4.952	3.617
800	0.012	0.020	0.032	0.045	0.049	0.059	100.000	19.986	10.493	8.102	6.458	4.840
900	0.010	0.019	0.029	0.039	0.048	0.068	100.000	10.794	9.001	6.777	4.832	2.879
1000	0.014	0.028	0.037	0.043	0.048	0.068	100.000	20.200	13.845	9.629	5.150	2.494

(*) Percentages computed on number of subproblems generated when $\alpha = 0.02 ub_{init}$.

Table 4.6: Performance of α-MT1 on SC knapsack instances (Data range $[1001, 2000]$)

(a) Smaller sized instances with accuracy provided in absolute values.

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
50	0.000	0.209	2.084	5.194	7.709	9.992	100.000	68.458	0.287	0.264	0.249	0.223
75	0.000	0.761	4.533	5.908	10.181	11.401	100.000	7.966	1.510	1.215	1.141	1.139
100	Less than half of the instances could be solved within time											
125	Less than half of the instances could be solved within time											
150	Less than half of the instances could be solved within time											

(b) Larger sized instances with accuracy provided in percentage values.

n	Percentage deviation from upper bound ub_{init}						Percentage of number of subproblems required(*)					
	α values (%)						α values (%)					
	0.02	0.04	0.06	0.08	0.10	0.12	0.02	0.04	0.06	0.08	0.10	0.12
200	0.015	0.031	0.046	0.062	0.073	0.083	100.000	79.927	70.859	61.100	47.402	33.319
300	0.015	0.032	0.047	0.063	0.081	0.094	100.000	80.010	70.215	60.925	51.008	38.815
400	0.015	0.030	0.043	0.059	0.082	0.101	100.000	62.258	53.246	41.670	29.215	16.428
500	0.016	0.032	0.045	0.056	0.067	0.083	100.000	77.290	63.118	35.635	17.358	4.494
600	0.011	0.025	0.037	0.057	0.065	0.082	100.000	73.484	52.871	25.396	4.198	0.880
700	0.013	0.026	0.041	0.058	0.066	0.081	100.000	69.697	32.952	1.181	0.842	0.547
800	Less than half of the instances could be solved within time											
900	0.016	0.031	0.044	0.047	0.057	0.077	100.000	12.015	3.919	0.806	0.164	0.082
1000	Less than half of the instances could be solved within time											

(*) Percentages computed on number of subproblems generated when $\alpha = 0.02ub_{init}$.

Table 4.7: Performance of α-MT1 on ISC knapsack instances (Data range $[1, 1000]$)

(a) Smaller sized instances with accuracy provided in absolute values.

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
50	0.000	0.020	0.950	3.126	6.059	7.648	100.000	50.350	3.108	0.210	0.143	0.035
75	0.000	0.343	2.103	3.216	5.067	6.844	100.000	35.252	1.067	0.185	0.142	0.075
100	0.000	0.519	1.649	6.192	7.844	8.108	100.000	26.698	2.883	0.101	0.026	0.026
125	0.000	0.775	2.983	4.239	4.846	4.846	100.000	22.295	3.108	0.050	0.012	0.012
150	0.000	0.678	1.125	3.395	5.267	5.267	100.000	5.120	0.236	0.040	0.039	0.039

(b) Larger sized instances with accuracy provided in percentage values.

n	Percentage deviation from upper bound ub_{init}						Percentage of number of subproblems required(*)					
	α values (%)						α values (%)					
	0.02	0.04	0.06	0.08	0.10	0.12	0.02	0.04	0.06	0.08	0.10	0.12
200	0.013	0.017	0.018	0.018	0.018	0.018	100.000	6.818	0.455	0.455	0.455	0.455
300	0.011	0.011	0.011	0.011	0.011	0.011	α achieved at the root of the BnB tree					
400	0.008	0.008	0.008	0.008	0.008	0.008	α achieved at the root of the BnB tree					
500	0.005	0.005	0.005	0.005	0.005	0.005	α achieved at the root of the BnB tree					
600	0.005	0.005	0.005	0.005	0.005	0.005	α achieved at the root of the BnB tree					
700	0.004	0.004	0.004	0.004	0.004	0.004	α achieved at the root of the BnB tree					
800	0.003	0.003	0.003	0.003	0.003	0.003	α achieved at the root of the BnB tree					
900	0.002	0.002	0.002	0.002	0.002	0.002	α achieved at the root of the BnB tree					
1000	0.003	0.003	0.003	0.003	0.003	0.003	α achieved at the root of the BnB tree					

(*) Percentages computed on number of subproblems generated when $\alpha = 0.02 ub_{init}$.

Table 4.8: Performance of α-MT1 on ISC knapsack instances (Data range $[1001, 2000]$)

(a) Smaller sized instances with accuracy provided in absolute values.

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
50	0.000	0.146	1.205	3.159	3.975	4.551	100.000	35.624	1.533	0.723	0.723	0.723
75	0.000	1.190	2.541	3.913	5.171	6.808	100.000	6.100	0.916	0.870	0.866	0.860
100	Less than half of the instances could be solved within time											
125	Less than half of the instances could be solved within time											
150	Less than half of the instances could be solved within time											

(b) Larger sized instances with accuracy provided in percentage values.

n	Percentage deviation from upper bound ub_{init}						Percentage of number of subproblems required(*)					
	α values (%)						α values (%)					
	0.02	0.04	0.06	0.08	0.10	0.12	0.02	0.04	0.06	0.08	0.10	0.12
200	0.004	0.004	0.004	0.004	0.004	0.004	100.000	100.000	100.000	100.000	100.000	100.000
300	0.002	0.003	0.003	0.006	0.006	0.006	100.000	57.414	57.414	21.332	21.332	21.332
400	0.002	0.004	0.006	0.006	0.006	0.006	100.000	37.950	11.573	11.573	11.573	11.573
500	0.002	0.002	0.002	0.002	0.011	0.021	100.000	100.000	100.000	100.000	32.393	4.743
600	0.001	0.006	0.008	0.012	0.020	0.025	100.000	33.714	18.272	7.906	0.703	0.588
700	0.001	0.004	0.010	0.013	0.027	0.027	100.000	46.568	9.823	1.480	0.691	0.691
800	0.001	0.002	0.007	0.010	0.019	0.031	100.000	52.677	6.850	1.612	1.257	0.900
900	0.002	0.002	0.002	0.009	0.014	0.014	100.000	100.000	100.000	39.086	26.294	26.294
1000	0.001	0.002	0.004	0.004	0.018	0.028	100.000	23.050	1.922	1.922	1.104	0.559

(*) Percentages computed on number of subproblems generated when $\alpha = 0.02ub_{init}$.

Table 4.9: Performance of α-MT1 on ASC knapsack instances (Data range $[1, 1000]$)

(a) Smaller sized instances with accuracy provided in absolute values.

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
50	0.000	0.000	0.000	0.220	0.486	1.006	100.000	96.786	93.223	85.545	73.081	66.226
75	0.000	0.000	0.002	0.570	0.768	1.328	100.000	98.202	96.680	86.009	80.642	77.728
100	0.000	0.000	0.013	0.103	0.509	1.198	100.000	99.154	97.500	92.676	83.620	78.664
125	0.000	0.000	0.000	0.057	0.784	1.236	100.000	99.561	97.031	96.821	87.800	83.785
150	0.000	0.000	0.470	0.470	0.780	1.438	100.000	99.245	98.039	94.234	86.348	84.104

(b) Larger sized instances with accuracy provided in percentage values.

n	Percentage deviation from upper bound ub_{init}						Percentage of number of subproblems required(*)					
	α values (%)						α values (%)					
	0.02	0.04	0.06	0.08	0.10	0.12	0.02	0.04	0.06	0.08	0.10	0.12
200	0.004	0.005	0.053	0.056	0.058	0.061	100.000	98.438	69.531	37.257	0.027	0.025
300	Less than half of the instances could be solved within time											
400	Less than half of the instances could be solved within time											
500	Less than half of the instances could be solved within time											
600	Less than half of the instances could be solved within time											
700	0.014	0.019	0.026	0.027	0.041	0.059	100.000	1.300	0.483	0.382	0.286	0.191
800	0.011	0.022	0.028	0.033	0.038	0.046	100.000	10.770	8.131	7.637	7.284	6.836
900	0.014	0.024	0.030	0.045	0.050	0.062	100.000	18.609	4.646	3.083	2.775	2.155
1000	0.010	0.013	0.021	0.026	0.035	0.047	100.000	44.669	17.415	5.253	4.016	2.768

(*) Percentages computed on number of subproblems generated when $\alpha = 0.02 ub_{init}$.

Table 4.10: Performance of α-MT1 on ASC knapsack instances (Data range $[1001, 2000]$)

(a) Smaller sized instances with accuracy provided in absolute values.

n	Deviations from the optimal solution						Percentage of number of subproblems required					
	α values						α values					
	0.0	5.0	10.0	15.0	20.0	25.0	0.0	5.0	10.0	15.0	20.0	25.0
50	0.000	0.048	0.048	0.133	0.364	0.882	100.000	99.713	98.570	93.362	89.745	81.438
75	0.000	0.000	0.449	0.685	1.391	1.624	100.000	95.831	89.127	84.120	74.935	64.289
100	0.000	0.000	0.872	3.135	3.374	5.173	100.000	88.489	69.778	44.559	43.570	26.098
125	0.000	0.064	1.043	2.109	2.591	5.036	100.000	87.211	64.109	53.272	48.722	32.310
150	Less than half of the instances could be solved within time											

(b) Larger sized instances with accuracy provided in percentage values.

n	Percentage deviation from upper bound ub_{init}						Percentage of number of subproblems required(*)					
	α values (%)						α values (%)					
	0.02	0.04	0.06	0.08	0.10	0.12	0.02	0.04	0.06	0.08	0.10	0.12
200	Less than half of the instances could be solved within time											
300	0.011	0.016	0.022	0.033	0.047	0.062	100.000	2.652	0.557	0.294	0.245	0.209
400	0.014	0.028	0.037	0.054	0.061	0.065	100.000	72.876	50.928	18.643	3.864	2.404
500	0.012	0.025	0.045	0.059	0.075	0.085	100.000	58.878	35.572	16.915	9.626	3.522
600	0.012	0.024	0.037	0.052	0.060	0.066	100.000	62.175	45.802	26.826	5.014	0.707
700	0.012	0.020	0.034	0.052	0.063	0.072	100.000	66.276	20.585	1.201	0.699	0.337
800	0.014	0.024	0.039	0.047	0.061	0.073	100.000	10.780	3.995	0.644	0.184	0.115
900	0.011	0.025	0.035	0.037	0.049	0.057	100.000	5.697	2.141	0.290	0.144	0.091
1000	0.009	0.013	0.041	0.046	0.046	0.051	100.000	71.083	16.269	0.683	0.683	0.347

(*) Percentages computed on number of subproblems generated when $\alpha = 0.02 ub_{init}$.