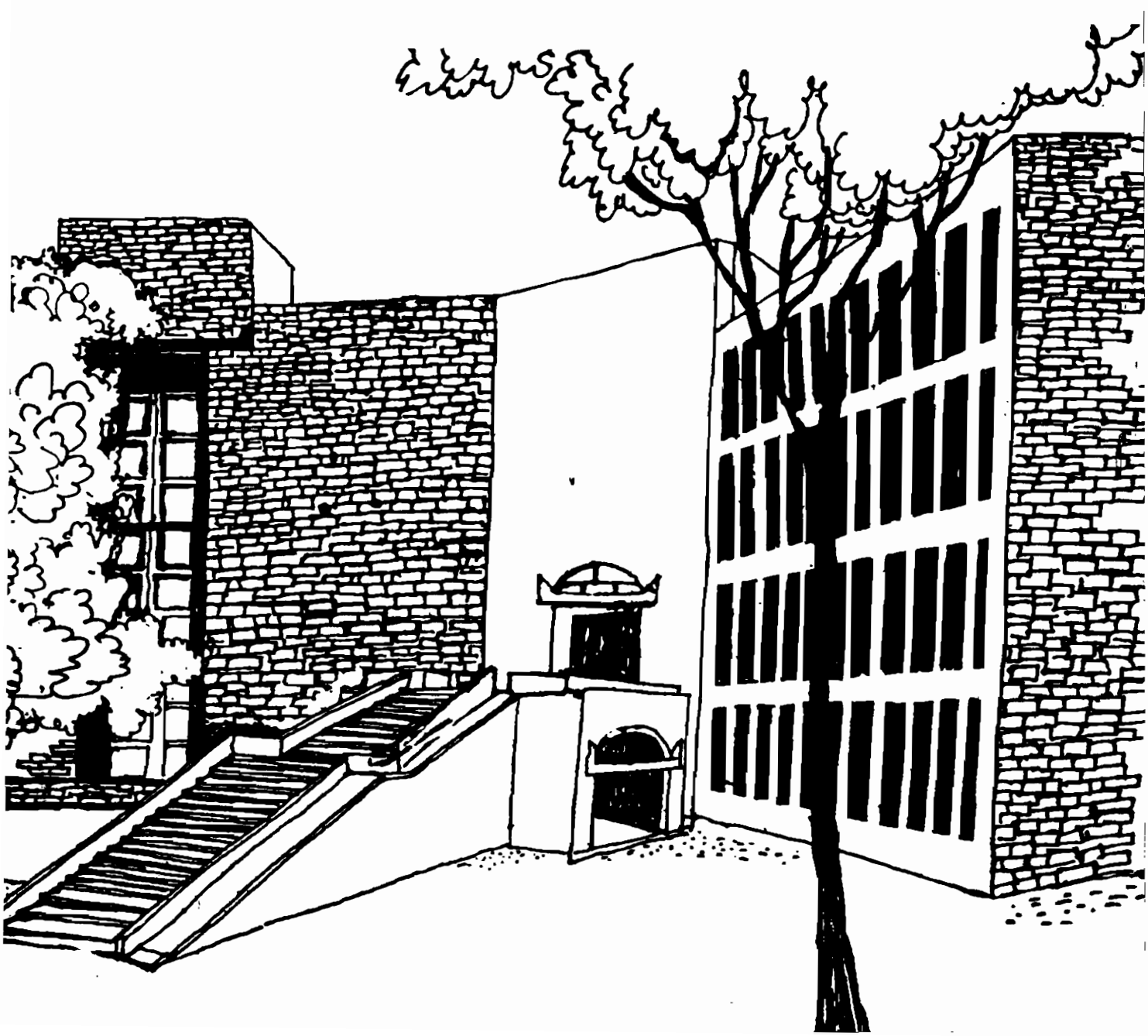
विद्याविनियोगाधिकारः

IIM

AHMEDABAD

Working Paper



Obtaining Near-Optimal Solutions for the Binary Knapsack Problem

Boris Goldengorin
Diptesh Ghosh

W.P.No. 2002-02-03
February 2002 116 89

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD-380 015
INDIA

Obtaining Near-Optimal Solutions for the Binary Knapsack Problem

Boris Goldengorin
Faculty of Economic Sciences, University of Groningen,
P.O. Box 800, 9700AV Groningen, The Netherlands
B.Goldengorin@eco.rug.nl

Diptesh Ghosh
Production & Quantitative Methods Area, IIM Ahmedabad,
Vastrapur, Ahmedabad 380015, India
diptesh@iimahd.ernet.in

February 11, 2002

Abstract

In this paper we consider the well-known binary knapsack problem. We propose a method of embedding heuristics in a branch and bound framework to obtain solutions with profits within a pre-specified quality parameter within very short times. Our computational experiments on the more difficult problems show that algorithm can generate solutions with profits within 0.01% of that of an optimal solution in less than 10% of the time required by exact algorithms based on similar principles.

1 Introduction

Given a set of n elements, where p_j and w_j respectively are the profit and the weight of element j, and a knapsack of weight capacity c, the binary knapsack problem (BKP) is one of selecting a subset of the elements so as to maximize the profit $P(\mathbf{x}) = \sum_{j=1}^{n} p_j x_j$ of a solution $\mathbf{x} = (x_1, \ldots, x_n)$ subject to the solution weight $W(\mathbf{x}) = \sum_{j=1}^{n} w_j x_j \leq c$, where

$$x_j = \begin{cases} 1 & \text{if element } j \text{ is selected to be in the subset,} \\ 0 & \text{otherwise.} \end{cases}$$

In this paper, we will assume that p_j, w_j, and c are positive, $w_j < c$, for $j = 1, \ldots, n$, and $W(\mathbf{1}) > c$. We also assume, without loss of generality, that the elements are ordered according to non-increasing profit to weight ratios, ties being broken arbitrarily.

BKP is among the most widely studied problems of discrete optimization, since many practical problems 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).

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]. Design of heuristics for BKP, which generate feasible but usually suboptimal solutions within short execution times is also an area of research (see Martello and Toth [5], and Ghosh [2]). These heuristics usually perform very well in practice, and output solutions that are very close to optimal. The theoretical performance measure for such heuristics is usually based on their worst case performance ratio which, for BKP, form a very weak bound on the deviation of the profit of the heuristic solution to that of the optimal solution. Moreover such measures are heuristic-specific and not instance-specific.

In this paper we present an algorithm α-MT1 to obtain solutions with performance guarantees in absolute terms. 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 solution 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, our algorithm takes an *prescribed accuracy* parameter α as input. α-MT1 guarantees that the profit (z^{α}) of the solution it outputs for an instance would satisfy the expression $z^{\star} - z^{\alpha} \leq \alpha$, where $z^{\star}$ is the profit of an optimal solution to that instance. The term $z^{\star} - z^{\alpha}$ will be called the *achieved accuracy*.

The branch and bound framework that we use is based on the well-known mt1 algorithm in Martello and Toth [6]. More sophisticated and efficient solution techniques are available for solving the BKP, but mt1 is easy to implement and is sufficient to illustrate the applicability of the idea we propose and to demonstrate the rapid reduction in execution time with increasing values of achieved accuracy. Notice that the 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.

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 classes known from the literature to be difficult for branch and bound based algorithms. We summarize our results of the paper in Section 4 and suggest directions for future research.

2 The α-MT1 Algorithm

The α-MT1 algorithm is based on framework of branch and bound (BnB). However α-MT1 differs from the standard branch and bound algorithm in two ways.

First, we incorporate the prescribed accuracy factor in the bounding procedure. Consider a subset S of the set of feasible solutions, $\mathfrak{S}$. We first compute an upper bound ub_S to the profit of the solutions in S. We also use a good and relatively fast heuristic $\mathcal{H}$ to obtain a good solution 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 also use a stopping condition that can stop the algorithm before it evaluates the whole of $\mathfrak{S}$. At any point during the execution of α-MT1, let ub be an upper bound to the optimal profit of the instance. If, at any subset S of $\mathfrak{S}$, the heuristic $\mathcal{H}$ produces a solution $z^{\mathcal{H}}$ satisfying the condition $Z^{\mathcal{H}} + \alpha \geq ub$, then since $ub \geq z^{\star}$, it immediately follows that $z^{\star} - Z^{\mathcal{H}} \leq \alpha$. This implies that we can stop the computations immediately.

The α-MT1 algorithm is based on the branch and bound algorithm mt1 proposed in Martello and Toth [6]. However, contrary to mt1 in our implementation 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 to terminate the algorithm when the list is empty, or when the stopping condition mentioned earlier is satisfied. 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 their usual connotations, and $x_j = \bullet$ denotes that no decision has been made on whether or not to include element j in the solution. A partial solution where each of the components are either 0 or 1 is called a *complete solution*. The pseudocode of α-MT1 is given in Figure 1. We assume 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 the branch and bound algorithm in Martello and Toth [6]. We describe these procedures in detail in the remaining portion of this section.

- $ub(\mathbf{x})$: Upper bounds are computed in α-MT1 in the following manner (due to Martello and Toth [6]:
  Let $\Pi = \sum_{j:x_j=1} p_j$, $c_r = c - \sum_{j:x_j=1} w_j$, $s = \min\{j : \sum_{i=1,x_i=\bullet}^{j} w_i > c\}$, $s_- = \max\{j : x_j = \bullet, j < s\}$, $s_+ = \min\{j : x_j = \bullet, j > s\}$, and $\bar{c} = c - \sum_{i=1}^{s_-} w_i$. Then

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

  is an upper bound to the optimal profit for the given instance. Note that $ub(\mathbf{x})$ may not correspond to the profit of any feasible solution.

- $\mathcal{H}(\mathbf{x})$: The heuristic $\mathcal{H}(\mathbf{x})$ is a local search heuristic with a 2-exchange neighborhood. It involves four steps. In the first step put all the elements j with $x_j = 1$ in a set S, compute $c_r = c - \sum_{j:x_j=1} w_j$, and construct a set E_r of elements j with $x_j = \bullet$. In the second step, we compute a greedy solution S_G by considering the elements $j \in E_r$ 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 $j = \min\{j : x_j = \bullet\}$, and $\Pi = \sum_{x_j=1} p_j$. We first construct the set N of the largest number of consecutive elements with $x_j = \bullet$ which 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 $\Pi + P(N) = ub(\mathbf{x})$, 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 at most 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.

Algorithm α-MT1
Input: BKP 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 ← ub( (•,•,...,•) ); BestSolnSoFar ← ∅; BestSolnValue ← −∞;
03    LIST ← { (•,•,...,•) };
04    while LIST ≠ ∅ do
05    begin
06       Choose a subproblem x = (x_j) from LIST;
07       x^H = (x_j^H) = H(x);
08       if ub − P(x^H) ≤ α then                          (* New Stopping Rule *)
09          return x^H and stop;
10       ub_x ← ub(x)
11       if ub_x ≤ BestSolnValue then                     (* Discard this subproblem *)
12          goto 28;
13       if ub_x − P(x^H) ≤ α then          (* x^H is within the prescribed accuracy α *)
14       begin
15          Update BestSolnSoFar, BestSolnValue and ub if required;
16          goto 28;
17       end;
18       x ← forward-move(x);
19 (* Creating new subproblems by branching *)
20       k ← min{j : x_j = •};
21       x^new ← x;
22       x_k^new ← 0;
23       LIST ← LIST ∪ {x^new};
24       x^new ← x;
25       x_k^new ← 1;
26       if W(x^new) ≤ c then
27          LIST ← LIST ∪ {x^new};
28    end;
29    return BestSolnSoFar;
30 end.
```

Figure 1: Pseudocode of α-MT1.

Notice that running α-MT1 with $\alpha = 0.0$ will return an optimal solution but will

make α-MT1 run like mt1, while running α-MT1 with a large value of α will make it run like $\mathcal{H}(\mathbf{x})$, i.e. in this case like local search with a 2-exchange neighborhood.

3 Computational Experiments

We coded the algorithm in C, compiled it using the LCC compiler for Windows NT [7], and ran it on a 733MHz Intel Pentium III machine with 128MB RAM. The data for the instances are real-valued. There are nine different classes of randomly generated BKP instances studied in the literature (see Martello, Pisinger and Toth [4]). Four of these belong to the so-called "even-odd" classes. We did not use them in our computations. 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. Of the remaining five classes, the classes of uncorrelated problems and of weakly correlated problems are relatively easy. For our experiments therefore, we chose the following three types of instances:

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 $0.5 \sum_{j=1}^{n} w_j$.

As mentioned earlier, the data for each of the instances were real-valued. In our experience, this makes the problems more difficult to solve. For each of the three instance types, we varied the instance sizes from 50 to 1000. For each instance size and instance type, we generated twenty instances. The value of L and H were chosen to be 1001 and 2000 respectively. The data range was chosen in this manner so that the problems did not contain any element with small weights, whose presence often makes the solution process easier.

We examine the behavior of α-MT1 in terms of the profit of the solution that it outputs, and the size of the BnB tree that it generates to solve instances corresponding to different instance sizes and values of α. Since mt1 is known to be unable to solve moderate to large sized SC, ISC, and ASC type instances, we divide these instances into two categories, small and large. The small instances are of sizes varying from 50 to 150, and the large instances are of sizes varying from 200 to 1000. We allow 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 compute an upper bound ub and use α values of 0.02%, 0.04%, 0.06%, 0.08%, 0.10%, and 0.12% of ub. α-MT1 is 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 have been solved within the given time for each α value.

For the small sized instances, we can use α-MT1 to obtain optimal solutions. Thus we can compute the actual deviation of the solution output by α-MT1 from that of the optimal solution. But for the large sized instances, we cannot obtain optimal solutions using α-MT1 within reasonable times. For these problems therefore, we measure deviations as a percentage of ub. These values form an upper bound to the actual deviations from the optimal profit for these instances. Since different instances in the same problem set have widely different sizes of the BnB tree generated by α-MT1, (the *size* of a BnB tree is the number of nodes in the tree,) we 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 small sized SC, ISC, and ASC instances. For the large sized instances, we cannot 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 $\alpha = 0.02\%$ of ub. This procedure makes it impossible to compare the percentage reductions in the sizes of the BnB trees for small and large instances. Tables 1 through 3 present the results of our computational experience.

Let $\Gamma(n, \alpha)$ be the ratio of the achieved accuracy to the prescribed accuracy for instances of size n and a prescribed accuracy parameter α. Also let $\Phi(n, \alpha, \alpha_0)$, be 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 SC type instances (refer to Table 1), $\Gamma(n, \alpha)$ is independent of the value of n and increases linearly with α. For small instances of SC problems, the size of the BnB tree was fairly insensitive to increases in α values for $\alpha \geq 15.0$. $\Phi(n, \alpha, 0.0)$ initially drops steeply with increasing α, but then increases at a linear rate when α is large enough. It is also independent of n.

The behavior of $\Gamma(n, \alpha)$ and $\Phi(n, \alpha, 0.0)$ for ISC type instances (refer to Table 2) was mostly independent of n. 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 the SC type instances. The value of $\Gamma(n, \alpha)$ increased with n for the smaller instances, but were relatively independent of n in the larger instances. Also, when $\alpha \geq 10.0$, small sized ISC instances led to very small BnB trees. 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 $\alpha \geq 0.1\%$ of ub.

The behavior of α-MT1 on ASC instances (refer to Table 3) was seen to be very different from that of the other instances that we experimented with. $\Gamma(n, \alpha)$ values were seen to be more sensitive to n than the 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 .) $\Phi(n, \alpha, 0.0)$ was seen to be increasing as n increased, and for larger instances, the size of the BnB tree was insensitive to increases in α when $\alpha \geq 0.05\%$ of ub. 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 (with the exception of ASC type 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. The deviation of the solution output by α-MT1 was less than half the prescribed accuracy, and in terms of the size of the BnB tree, for most instances, 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 which embeds a local search based heuristic procedure within a branch and bound framework to produce a solution whose profit is within a pre-specified amount of the profit of an optimal solution. A characteristic of the solutions generated by α-MT1 is that their suboptimality is insensitive to the actual numbers in the instance data for the problem. We tested the performance of α-MT1 on a variety of difficult 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.

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. A challenging direction of further research in the type of algorithms that we propose here is to incorporate similar ideas into such dynamic programming based algorithms and obtain powerful algorithms for generating near-optimal solutions for a wider variety of binary knapsack problems.

Applying the concepts described here to find near-optimal solutions to other hard combinatorial optimization problems is also another promising area of research.

References

[1] M.J. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco (1979).

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

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

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

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

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

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

Table 1: Performance of α-MT1 on SC knapsack instances

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

n	Deviations from the optimal solution						%-age of number of subproblems reqd.					
	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	%-age deviation from ub						%-age of number of subproblems reqd.(*)					
	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											

(*) %-ages computed on number of subproblems generated when $\alpha = 0.02\%$ of ub.

Table 2: Performance of α-MT1 on ISC knapsack instances

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

n	Deviations from the optimal solution						%-age of number of subproblems reqd.					
	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	%-age deviation from ub						%-age of number of subproblems reqd.(*)					
	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

(*) %-ages computed on number of subproblems generated when $\alpha = 0.02\%$ of ub.

Table 3: Performance of α-MT1 on ASC knapsack instances

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

n	Deviations from the optimal solution						%-age of number of subproblems reqd.					
	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	%-age deviation from ub						%-age of number of subproblems reqd.(*)					
	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

(*) %-ages computed on number of subproblems generated when $\alpha = 0.02\%$ of ub.