

A toolbox for solving knapsack problems

David Pisinger, DIKU, University of Copenhagen
pisinger@diku.dk

April 7, 2003

How to use the software

The Knapsack demo may be downloaded from the site

<http://www.diku.dk/pisinger/KNAPDEMO/>

The program is written in Java, hence it should in principle run in any browser having Java enabled. The present version has mainly been tested using the Windows browser *Explorer*, hence if you have problems, please use the Windows computers in *Multimedia Lab*, 1st floor, south.

You may also download all the relevant files *index.html*, *AppletTest.jar*, *data.1*, *data.2*, *data.3* to your home directory and use the command

```
java -jar AppletTest.jar
```

to start the program.

In the following demonstrations are marked with a **D**, while exercises are marked with a **E**.

1 Introduction

The 0-1 knapsack problem is one of the most important models in combinatorial optimization, having numerous real-life applications as well as being an important subproblem in several solution algorithms for more complex problems. For a recent survey see Pisinger and Toth [6]. Assume that n items are given, each item having a nonnegative integer *profit* p_j and *weight* w_j . The problem is to select a subset of the items so that their overall profit is maximized without having the overall weight to exceed a given *capacity* c . Formally we may define the problem as:

$$\text{maximize } \sum_{j=1}^n p_j x_j \quad (1)$$

$$\text{subject to } \sum_{j=1}^n w_j x_j \leq c \quad (2)$$

$$x_j \in \{0, 1\}, \quad j = 1, \dots, n. \quad (3)$$

where the binary variable x_j is one if item j is selected and x_j is zero otherwise. The optimal solution is denoted x^* and the corresponding solution value z^* .

We will represent each item by a rectangle having height w_j and area p_j . In this way the problem can be restated as stacking a subset of the items such that their overall height does not exceed c and such that the overall area of the items is maximized. The width p_j/w_j of an item may be interpreted as the *efficiency* of the item, i.e. a measure of the profit obtained pr. weight unit.

1.1 Simple packing

Despite the simplicity of the problem, it is NP-hard to solve, and hence we do not know any efficient algorithm for its solution.

Try to experiment with the following instance. By clicking on an item you can pack it into the knapsack or remove it from the knapsack. In the table at the top of the screen you may see the current solution value $z = \sum_{j=1}^n p_j x_j$ and the current weights $\sum_{j=1}^n w_j x_j$. The optimal solution value is $z = 30$, but even with this information it is not easy to find an optimal solution even for this simple instance.

D1 [Knapsack, data.3]

1.2 Greedy heuristic

If we sort the items according to decreasing efficiencies p_j/w_j , we may pack the knapsack in a greedy way. In each step we simply choose the item with the largest efficiency which fits into the knapsack.

Run the greedy algorithm on the following instance.

D2 [Knapsack, data.1, sorted]

1.3 Solving the linear problem

Assume that we were allowed to cut the items into smaller pieces. In this case the greedy heuristic always returns an optimal solution, by simply filling the knapsack with the most efficient items until the first item s does not fit into the knapsack. In this case a suitable fraction of item s is chosen, so that all the remaining capacity is used. The item s will be denoted the *split item*. The solution becomes $x_j = 1$ for $j = 1, \dots, s-1$, and $x_s = \frac{c - \sum_{j=1}^{s-1} w_j}{w_s}$. The corresponding solution value is

$$z_{LP} = \sum_{j=1}^{s-1} p_j + (c - \sum_{j=1}^{s-1} w_j) \frac{p_s}{w_s} \quad (4)$$

The solution value z_{LP} to the linear problem is an upper bound U on the solution value to the original knapsack problem. This means that $z^* \leq z_{LP}$ for the optimal solution value z^* . Any feasible solution is on the other hand denoted a lower bound z . Obviously we have $z \leq z^*$.

2 Upper bounds

In the following we will assume that the items are sorted according to decreasing efficiencies p_j/w_j .

The simplest upper bound for KP can be derived as

$$U_0 = \lfloor \frac{c p_1}{w_1} \rfloor.$$

E1 Prove correctness (including the truncation) of the bound, and derive the time complexity.

D3 [Bounds, data.3, U0]

By allowing items to be cut into smaller pieces we obtain the so-called *Dantzig bound* [1].

$$U_1 = \lfloor z_{LP} \rfloor.$$

E2 Prove correctness of the bound, and derive the time complexity.

E3 Prove that $U_1 \leq U_0$.

D4 [Bounds, data.3, U1]

Considering separately the case of packing the split item s or not, Martello and Toth [3] derived the following upper bound

$$U_2 = \max \left\{ \left\lfloor \hat{p} + (c - \hat{w}) \frac{p_{s+1}}{w_{s+1}} \right\rfloor, \left\lfloor \hat{p} + p_s + (c - \hat{w} - w_s) \frac{p_{s-1}}{w_{s-1}} \right\rfloor \right\} \quad (5)$$

where $\hat{p} = \sum_{j=1}^{s-1} p_j$ and where $\hat{w} = \sum_{j=1}^{s-1} w_j$.

E4 Prove correctness of the bound, and derive the time complexity.

E5 Prove that $U_2 \leq U_1$.

D5 [Bounds, data.3, U2]

3 Branch-and-bound

The knapsack problem may be solved by brute force by simply enumerating all feasible packings. Basically, we draw a tree, where each node corresponds to the inclusion or omission of an item. Since we have n items, there will be up to 2^n nodes. Since several of the packings will not be feasible, the tree will have fewer nodes in practice.

Try complete enumeration for the following instance

D6 [Branch-and-bound, data.3, primal, no bounds]

3.1 Simple branch-and-bound

As the name indicates, a branch-and-bound algorithm is based on two fundamental principles: branching and bounding. Assume that we wish to solve the following maximization problem

$$\max_{x \in X} f(x) \quad (6)$$

where X is a finite solution space. In the *branching* part, a given subset of the solution space $X' \subseteq X$ is divided into a number of smaller subsets X_1, \dots, X_m . These subsets may in principle be overlapping but their union must span the whole solution space X , thus $X_1 \cup X_2 \cup \dots \cup X_m = X'$. The process is in principle repeated until each subset contains only a single feasible solution. By choosing the best of all considered solutions according to the present objective value, one is guaranteed to find a global optimum for the stated problem.

The *bounding* part of a branch-and-bound algorithm derives upper and lower bounds for a given subset X' of the solution space. A lower bound z_ℓ may be chosen as the best solution encountered so far in the search. If no solutions have been considered yet, one may choose z_ℓ as the objective value of a heuristic solution. An upper bound $U_{X'}$ for a given solution space $X' \subseteq X$ is a real number satisfying

$$U_{X'} \geq f(x), \quad \text{for all } x \in X' \quad (7)$$

The upper bound is used to prune parts of the search space as follows. Assume that $u_{X'} \leq z_\ell$ for a given subset X' . Then (7) immediately gives that

$$f(x) \leq u_{X'} \leq z_\ell, \quad \text{for all } x \in X' \quad (8)$$

meaning that a better solution than z_ℓ cannot be found in X' and thus we need not investigate X' further.

Most primal methods are based on the framework proposed by Horowitz and Sahni [2]. In a recursive formulation each iteration corresponds to a dichotomic branch on the most efficient free variable x_b , where $x_b = 1$ is investigated before $x_b = 0$ according to the greedy principle. Having branched on variables $x_j, j < b$, the profit and weight sum of the currently fixed variables is \bar{p} resp. \bar{w} , hence if $\bar{w} > c$ for a given node, we may backtrack. Also, if an upper bound for the remaining problem is less or equal to the current lower bound z , we may backtrack. The lower bound z is improved during the search and the algorithm returns true if an improved solution was found in the present node or any descendant node. This makes it possible to define the optimal solution vector x^* during the backtracking as will be described below. At the beginning the lower bound should be initialized to $z = 0$ before calling Primal-Branch(1, 0, 0).

The algorithm backtracks if we have reached the bottom of the search tree ($b > n$), or if the bound U shows that no improved solution can be found in descending nodes ($\bar{p} + U \leq z$).

The optimal solution vector x^* is partially defined during backtracking of the algorithm.

E6 Show how the solution vector x^* can be found in $O(n)$ time, once the algorithm has terminated.

```

Algorithm Primal-Branch( $b, \bar{p}, \bar{w}$ ): boolean;
improved := false
if  $\bar{w} > c$  then return improved
if  $\bar{p} > z$  then  $z := \bar{p}$ ;  $b^* := b$ ; improved := true
if  $j > n$  or  $c - \bar{w} < \min_{i=j, \dots, n} w_i$  then return improved
derive upper bound  $U$  with capacity  $c - \bar{w}$ 
if  $\bar{p} + U \leq z$  then return improved
if Primal-Branch( $b + 1, \bar{p} + p_b, \bar{w} + w_b$ ) then  $x_b := 1$ ; improved := true
if Primal-Branch( $b + 1, \bar{p}, \bar{w}$ ) then  $x_b := 0$ ; improved := true
return improved

```

Figure 1: The Primal-Branch algorithm is a recursive branch-and-bound algorithm which in each step branches on the variable x_b . The local variable `improved` is used to indicate whether an improved solution was found at the present or descending branching nodes. The Primal-Branch algorithm returns the value of `improved`

- E7** Discuss the effect on the time complexity of defining the solution vector during the backtracking of the algorithm, as opposed to saving the whole solution vector x^* each time an improved solution z has been found.
- E8** If the time used for deriving a bound U is $t(U)$, derive the worst-case time and space complexity of the algorithm.

3.2 Primal-dual branch-and-bound

The normal branch-and-bound approach is to start with an empty knapsack and repeatedly add an item into it. Pisinger [4] noticed that for large-sized instances this may be quite inefficient since the algorithm needs to consider several branching nodes before getting to a sufficiently filled knapsack. One could imagine that it would be more effective to use a branching strategy which keeps the knapsack sufficiently filled.

The so-called *primal-dual* algorithm starts by branching on the split item s . The algorithm repeatedly inserts an item into the solution if the current weight sum is less than c ; Otherwise it removes an item. In this way one only needs to consider knapsacks which are “appropriately filled”. On the other hand infeasible solutions must be considered in a transition stage.

The variables x_{a+1}, \dots, x_{b-1} are fixed at some value and for the corresponding profit and weight sum \bar{p}, \bar{w} , we implicitly assume that all items $j \leq s$ were also packed into the knapsack. Thus we insert an item when $\bar{w} \leq c$ and remove an item when $\bar{w} > c$. In contrary to the Primal-Branch algorithm we cannot backtrack immediately whenever $\bar{w} > c$, as infeasible solutions may become feasible at a later stage by removal of some items. Instead we must rely on the upper bound U_0 to prune the search tree. The resulting algorithm is called by

```

Algorithm Primal-Dual-Branch( $a, b, \bar{p}, \bar{w}$ ): boolean
improved := false
if  $\bar{w} \leq c$  then
    try to insert item b
    if  $\bar{p} > z$  then  $z := \bar{p}$ ;  $a^* := a$ ;  $b^* := b$ ; improved := true
    if  $b > n$  then return improved
    derive upper bound  $U$  with capacity  $c - \bar{w}$ 
    if  $\bar{p} + U \leq z$  then return improved
    if Primal-Dual-Branch( $a, b + 1, \bar{p} + p_b, \bar{w} + w_b$ ) then  $x_b := 1$ ; improved := true
    if Primal-Dual-Branch( $a, b + 1, \bar{p}, \bar{w}$ ) then  $x_b := 0$ ; improved := true
else
    try to remove item a
    if  $a < 1$  then return improved
    derive upper bound  $U$  with capacity  $c - \bar{w}$ 
    if  $\bar{p} + U \leq z$  then return improved
    if Primal-Dual-Branch( $a - 1, b, \bar{p} - p_a, \bar{w} - w_a$ ) then  $x_a := 0$ ; improved := true
    if Primal-Dual-Branch( $a - 1, b, \bar{p}, \bar{w}$ ) then  $x_a := 1$ ; improved := true
return improved

```

Figure 2: The primal-dual algorithm Primal-Dual-Branch starts from the split solution, and in each iteration it either inserts an item b or removes an item a according to the weight sum \bar{w} of the currently included items. The local variable improved is used to indicate whether an improved solution was found at the present or descending branching nodes. The value of improved is returned from the algorithm.

Primal-Dual-Branch($s - 1, s, \hat{p}, \hat{w}$), where \hat{p} and \hat{w} are the profits of the split solution defined

$$\hat{p} = \sum_{j=1}^n p_j \hat{x}_j, \quad \hat{w} = \sum_{j=1}^n w_j \hat{x}_j. \quad (9)$$

Initially the lower bound must be set to $z = 0$.

When inserting item b a bound U is derived on items $j \geq b$. When removing item a the knapsack is over-filled and thus U is derived by minimizing the profit removed such that the overall weight sum comes below c .

- E9** Show how the solution vector x^* can be found in $O(n)$ time, once the algorithm has terminated.
- E10** If the time used for deriving a bound U is $t(U)$, derive the worst-case time and space complexity of the algorithm.
- E11** Based on the range of \bar{w} , discuss appropriate choices of upper bounds for each of the two algorithms, taking into account processing time and quality of bound.

- E12** Describe the possible range of \bar{w} , using respectively Primal-Branch and Primal-Dual-Branch. How does this affect the worst-case solution time? Which algorithmic technique could be used to improve the complexity of the algorithm when the possible range of \bar{w} is small?

3.3 Experiments with the algorithms

In the window at the top of the demo you find information on computational effort: `nodes` denotes the number of branch-and-bound nodes (i.e. recursive calls to the above algorithm). `iterations` denotes the number of steps used in calculating the bounds (i.e. number of items considered).

- E13** Based on the above exercises design an algorithm which you would expect to minimize the number of nodes, resp. number of iterations. You may choose between primal or primal-dual branching strategy, and the various upper bounds.

Next, try experimentally to verify your guess, running one of the following versions of the code:

- D7** [Branch-and-bound, data.3, primal, no bounds]
- D8** [Branch-and-bound, data.3, primal, U0]
- D9** [Branch-and-bound, data.3, primal, U1]
- D10** [Branch-and-bound, data.3, primal, U2]
- D11** [Branch-and-bound, data.3, primal-dual, no bounds]
- D12** [Branch-and-bound, data.3, primal-dual, U0]
- D13** [Branch-and-bound, data.3, primal-dual, U1]
- D14** [Branch-and-bound, data.3, primal-dual, U2]

4 Variable reduction

Before running a branch and bound algorithm or dynamic programming algorithm it may be advantageous to fix some variables at their optimal values. In this way the number of items n may be reduced, and hence the expensive enumeration will take less time. Since the worst-case time complexity of branch and bound and dynamic programming is very bad (exponential time, resp. pseudo-polynomial time), any effort possible should be made to reduce the instance size.

Variable reduction algorithms for the KP may be seen as a special case of the branch-and-bound paradigm described in section 3, where we only consider branching on a single variable at the root node. All variables are in turn chosen as the branching variable at the root node, and different branches are investigated corresponding to the domain of the variable. If a specific

branch is inferior then we may remove the corresponding values from the domain of the variable. The situation becomes particularly simple if the involved variables are binary as the elimination of one choice from the variable's domain immediately states the optimal value of the variable.

Let U_j^0 be an upper bound corresponding to the branch $x_j = 0$ and in a similar way U_j^1 an upper bound on the branch $x_j = 1$. Moreover, assume that an incumbent solution z has been found in some way, e.g. by using the greedy algorithm from Section 1.2. If $U_j^0 \leq z$ then we know that the branch x_j does not lead to an improved solution and thus we may fix the variable to $x_j = 1$. In a similar way $U_j^1 \leq z$ implies that $x_j = 0$ in every improved solution. All variables fixed at their optimal value may be removed from the problem thus decreasing the size of the instance.

As in all branch-and-bound algorithms the incumbent solution x corresponding to z should be saved. This is important as the reduction attempts to fix variables at their optimal value in every improved solution. If no improved solution is found, one should go back to the last incumbent solution.

As an upper bound in the reduction, any technique described in Section 2 can be used.

We will investigate the effect of the reduction algorithm using various upper bounds. First, a lower bound should be derived using the greedy algorithm:

E14 [Knapsack, data.3, sorted, 0-1 knapsack]

Then, the reduction algorithm may be applied. Try the different bounds, and observe their effect.

D15 [Reduction, data.3, U0]

D16 [Reduction, data.3, U1]

D17 [Reduction, data.3, U2]

D18 [Reduction, data.3, U3]

Having reduced the instance using one of the above bounds, the remaining instance may be solved using branch-and-bound or dynamic programming. Try to solve the reduced instance using branch-and-bound. Compare the number of branch-and-bound nodes to the number of nodes investigated without reduction.

Acknowledgements

The interactive toolbox was developed by Allan Odgaard on basis of a previous DOS-based toolbox developed by David Pisinger.

The present project was supported by NIK — Natural Science IT Competence Center — at University of Copenhagen.

References

- [1] G.B. Dantzig (1957), “Discrete Variable Extremum Problems”, *Operations Research*, **5**, 266–277.
- [2] E. Horowitz and S. Sahni (1974), “Computing partitions with applications to the Knapsack Problem”, *Journal of ACM*, **21**, 277–292.
- [3] S. Martello, P. Toth (1977), “An upper bound for the zero-one knapsack problem and a Branch and Bound algorithm”, *European Journal of Operational Research* **1**, 169–175.
- [4] D. Pisinger (1995), “An expanding-core algorithm for the exact 0-1 knapsack problem”, *European Journal of Operational Research* **87**, 175–187, (1995).
- [5] D. Pisinger (1999), “Linear time algorithms for knapsack problems with bounded weights”, *Journal of Algorithms* **33**, 1–14.
- [6] D. Pisinger, P. Toth (1998), “Knapsack Problems”, in D.Z. Du, P. Pardalos (eds.) *Handbook of Combinatorial Optimization*, Kluwer, 1–89.