# Stochastic Tree Search: An Illustration with the Knapsack Problem

João Pedro Pedroso and Mikio Kubo

Technical Report Series: DCC-2009-2
http://www.dcc.fc.up.pt/Pubs/


Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL
Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

# Stochastic Tree Search: An Illustration with the Knapsack Problem

João Pedro Pedroso and Mikio Kubo

April 2009

**Abstract**

This paper presents stochastic tree search, an alternative method to local search. Stochastic tree search consists of the exploration of a search tree, making use of heuristics for guiding the choice of the next branch, and a combination of diving and randomized selection of the path for exploring the tree.

Stochastic tree search efficiency relies on avoiding the repetition of the search on the same areas of the space, thus being unlikely to be trapped in a particular area of the search tree. By concentrating first on the most promising parts of the search tree, it allows to quickly determine solutions of very good quality.

## 1 Introduction

Combinatorial optimization is a field on the intersection between operations research, mathematics, and computer science. Problems in this field are among the most important in what concerns practical applications; and unfortunately there is no known efficient algorithm to solve for most of the interesting ones. As a way for tackling practical applications, *heuristics* — algorithms that can find good solutions in reasonable amounts of time — have been developed for many specific problems. A further step was done through the development of *metaheuristics*, i.e., algorithms that can be applied with few modifications to a wide range of problems. Metaheuristics have been very successful on finding very good solutions to most of the difficult problems with practical interest.

Problem-dependency is a major concern on the development of heuristics, as it implies that whenever a new problem is met a significant amount of time of a very skilled person has to be used for solving it. The idea behind the development of metaheuristics was to mitigate this requirement. Metaheuristics have been successful to a certain extent, but for obtaining solutions of very high quality they have to incorporate a good deal of problem-specific knowledge.

There are many works on the usage of exact, tree search methods, especially in a context of solving academic problems (e.g.[1], [2]). Even though some seminal works stressed the importance of enumerative methods [3], and some essays on this have been done [4], [5], research has to a large extent neglected the experimentation of exact methods applied to solve real-world, large size problems. This is arguably due to two reasons: the first is that these algorithms have to be interrupted before reaching a final, proven optimal solution; and the other is the difficulty inherent to the implementation of problem-specific, exact algorithms. Our claim in this text is that neither of these reasons is valid nowadays. Interrupting an exact algorithm, even in early stages of the search, most of the times still leads to state-of-the-art solutions; and the progress in computer programming languages makes the implementation of problem-specific exact algorithms a reasonable task, not substantially harder than the fine tuning of metaheuristics. Using exact methods on some easy problems, where complete search is possible in a reasonable amount of time, has an additional advantage over heuristics: that of being certain that the solution is optimal.

For the sake of clarity, we will illustrate our ideas on a simple, very well known problem: the knapsack problem.

## 2 The knapsack problem

For introducing the knapsack problem (KP) in a formal way, let us define a set $\mathcal{J} = \{1, \ldots, n\}$ of items available, and the capacity $c$ of the knapsack. Each item $j$ has associated a value denoted by $p_j$, and a weight denoted by $w_j$. For solving this problem we have to decide, for each of the items, if it will be selected or not; let us define a vector of variables $x = (x_1, \ldots, x_n)$ and assign the meaning $x_j = 1$ to *select item* $j$ and $x_j = 0$ otherwise. With these definitions, the total value of the items selected is the sum of $p_j x_j$ for all selected items. The capacity constraint implies that the sum of $w_j x_j$ for the selected items must be at most equal to $c$. This can be formulated as a linear integer program as follows:

$$\begin{aligned} \text{(KP)} \quad \text{maximise} \qquad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to :} \qquad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_j \in \{0,1\}, \qquad j = 1, \ldots, n. \end{aligned} \tag{1}$$

Notice that a solution to the knapsack problem may also be defined as the set of indices of the items put in the knapsack (i.e., the set of variables equal to one, $X = \{j \in \{1, \ldots, n\} | x_j = 1\}$.

For an in-depth analysis of this problem and its variants, the most comprehensive references are [6] and [7].

### 2.1 A greedy approach

A quick solution for the knapsack problem (albeit many times of poor quality) can be obtained by means of the greedy heuristics. Intuitively, an item is good if it has a high profit and a low weight. We can thus determine the profit-to-weight ratio $p_j/w_j$ for each item — called its *efficiency* — , and then start putting in the knapsack the items with highest efficiency. We continue doing this until no more items fit.

The algorithm starts thus by sorting the items by non-increasing order of efficiency, i.e., in such a way that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}. \tag{2}$$

Then, start with an empty knapsack and go through all the items (in this order), putting in the knapsack all the items whose inclusion does not violate the capacity constraint.

Algorithm 1 presents the main steps of the greedy algorithm for the knapsack problem, where $c$ is the capacity of the knapsack, $p$ and $w$ are vectors with the items' values and weights, respectively, and $\mathcal{J}$ is the list of items, in non-increasing order of efficiency. The output is $x$, the set of items put in the knapsack, and the corresponding profit ($\bar{z}^G$) and weight ($\bar{w}$).

**Algorithm 1**: A greedy heuristics for the knapsack problem

**Data**: $c, p, w, \mathcal{J}$
**Result**: $x, \bar{z}^G, \bar{w}$

```
1 x ← ∅
2 w̄ ← 0
3 z̄^G ← 0
4 for j ∈ J do
5     if w_j + w̄ ≤ c then
6         x ← x ∪ {j}
7         w̄ ← w̄ + w_j
8         z̄^G ← z̄^G + p_j
```

A variant of this algorithm can be used for obtaining an upper bound to the objective value. For the first item that does not fit into the knapsack, compute the fraction that could be inserted, and add the corresponding value to the sum of $p_j$ for the items in the knapsack. This is detailed in Algorithm 2.

**Algorithm 2**: An upper bound to the knapsack problem.

```
Data: c, p, w, J
Result: z̄^UB
z̄^UB ← 0
for j ∈ J do
    if w_j ≤ c then
        c ← c − w_j
        z̄^UB ← z̄^UB + p_j
    else
        z̄^UB ← z̄^UB + p_j c/w_j
        return z̄^UB
return z̄^UB
```

## 2.2 Complete search

We have seen in the greedy heuristics a method to obtain a solution in linear time (after the items are sorted): for each item, the decision of putting it on the knapsack or not is taken only once, and the alternative decision is not analyzed. This method can be easily extended in order to do a *complete search* of the solution space, by dividing the problem into two subproblems at each stage; in one of the subproblems the item is put in the knapsack, and in the other one it is not. This branching process is repeated for each variable, creating a tree with all the possible alternatives.

An example of the complete search tree for a small instance is shown in Figure 1. In this example the knapsack has capacity 8, and there is a set of four items available, with values $(10, 12, 14, 16)$ and weights $(2, 3, 4, 5)$. The knapsack is empty at the beginning (node 1). At each node it is decided if some item is put on the knapsack or not, until having considered all the items. Thus, at node 1 (the *root node*), we divide the main problem into two subproblems; in one of them (the left branch) it is decided that the item with $p = 10$ is put on the knapsack; in the right branch it is decided not to include that item. At each node the value of the current knapsack is shown at the top (in bold face), and the weight at the bottom.

The list of items is sorted in non-increasing order of efficiency; the greedy solution is to select items with $p = 10$ and $p = 12$; the items with $p = 14$ and $p = 16$ will not be included, as the capacity constraint would be violated. Therefore, if on the exploration of the tree we take the left branch whenever the capacity constraint is respected, and the right branch otherwise, we will reach the greedy solution; in this case the greedy rule would lead to node 19.

Observing the tree we can verify that the optimum is at node 26, corresponding to the selection of items $p = 12$ and $p = 16$; the value of these items is 28 and their weight is 8. There are nodes with a higher value, but none of them respects the knapsack's capacity.

This process, complete enumeration without further refinements, is a brute force method, which only works for very small instances. The size of the tree duplicates for each variable analyzed, and thus for $n$ variables there will be $2^n$ nodes to process — even with very fast computers, only very small instances can be solved this way.

Even though no known algorithm for solving the knapsack problem is better than this on the worst case, several improvements allow avoiding exploring a substantial part of the search tree in most of the practical situations. A simple set of rules for fathoming more nodes in which we are sure not to find the optimum solution, during the implicit enumeration of the tree is the following:

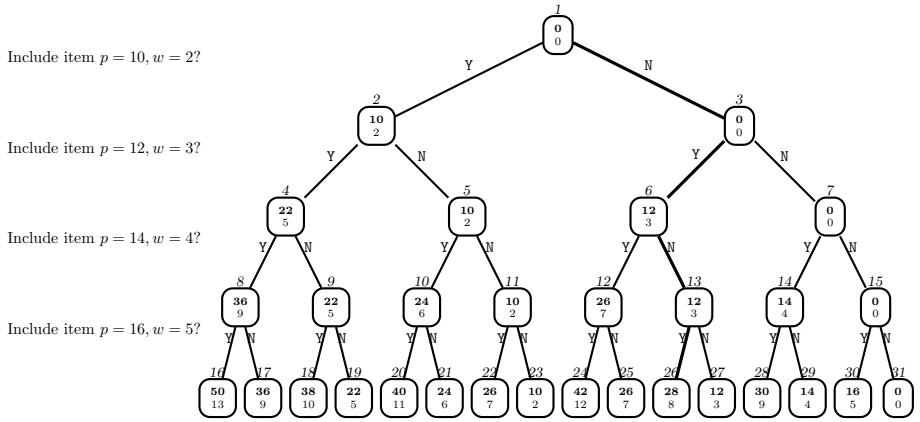


Figure 1: Complete search tree for a knapsack with capacity 8 and a set of four items, with values $(10, 12, 14, 16)$ and weights $(2, 3, 4, 5)$. The thick path leads to the optimal solution.

- when the size of the smallest item available does not fit in the knapsack, the node (and all its descendants) can be fathomed;
- when all the items available can be put in the knapsack, they can be immediately included (without analyzing alternative possibilities);
- if the upper bound for inserting the items not yet analyzed on the remaining capacity, plus the value of the items currently put on the knapsack, does not exceed the value of the best known solution, then the node can be fathomed.

Even though these rules are very simple, they allow a very dramatic reduction of the effort required to (implicitly) explore the tree.

There are several ways for exploring the nodes of a search tree; we will analyze them next. A generic algorithm for doing so, making use of a priority queue, is presented in Procedure `tree_search`. In this procedure, what is stored on each element of the queue $Q$ is the data necessary to explore a node (and its descendants) in the search tree: the set of items currently included in the knapsack, and the set of items available for future inclusion (those that were not analyzed yet).

The arguments of Procedure `tree_search` are the capacity $c$ of the knapsack, vectors with the items' values $p$ and weights $w$, and the set $\mathcal{J}$ of items available. Here we denote a solution as a set $X$ of selected items. The procedure starts by creating a queue (line 1) and inserting there the information concerning the root node: a tuple with an empty solution and the initial set $\mathcal{J}$ of items (line 2). The value of the best found solution is initialized in line 3, and updated in line 6. The main cycle, iterating as long as the queue is not empty, is in lines 4 to 11. The information of the node to analyze on the current cycle, i.e., the current solution $X$ and the set $\mathcal{J}$ of available items, is popped from the queue in line 5. The selection of the next item to analyze from the currently available set $\mathcal{J}$ is made at line 8; this can be done, e.g., by means of the same rule used in the greedy heuristics (decreasing efficiency). Then, information for the left recursion node is pushed into the queue at line 10 (if the item can be included in the knapsack without violating capacity), and for the right recursion at line 11. When the queue becomes empty, the cycle finishes and the best objective found is returned at line 12.

This procedure relies on two ordering processes: one for extracting elements of the queue $Q$ (line 5), and the other for extracting elements of the set $\mathcal{J}$ of items (line 8). Knowing a good lower bound to the current solution early on the tree exploration generally allows fathoming more nodes, and thus reducing the effort to find the optimum. For this purpose, these two choices are very important.

Concerning the selection of the next item of $\mathcal{J}$ to analyze, one possibility is to use the rule proposed for the greedy heuristics in Equation 2: analyze the remaining item with the greatest profit-to-weight ratio.

**Procedure** `tree_search`$(c, p, w, \mathcal{J})$

```
1  create empty queue Q
2  push ({}, J) into Q
3  z* ← −∞
4  while Q not empty do
5  |  pop (X, J) from Q
6  |  if Σ_{j∈X} p_j ≥ z* then z* ← Σ_{j∈X} p_j
7  |  if J ≠ ∅ then
8  |  |  j ← an element of J
9  |  |  if w_j + Σ_{k∈X} w_k ≤ c then
10 |  |  |  push (X ∪ {j}, J\{j}) into Q
11 |  |  push (X, J\{j}) into Q
12 return z*
```

As for the order of the nodes under which the tree is explored, the two most common possibilities are breadth-first search and depth-first search.

### 2.2.1 Breadth-first

One possibility for exploring the tree presented in Figure 1 is to explore all the nodes in a level of the tree (i.e., all the nodes that are at the same distance to the root) before starting exploring the next level (i.e., before going down one layer on the tree). In this case, when following the left branch first and without fathoming any nodes, the order of visit is node 1-2-3-4-5-...-31.

This can be implemented by means of a first-in-first-out queue on $Q$, and by selecting the item with highest efficiency when picking elements from the set $\mathcal{J}$ in Procedure `tree_search`.

### 2.2.2 Depth-first

One possibility for exploring the tree presented in Figure 1 is to descend on the tree as far as possible; when a node with no children is reached, backtrack to its upper node.

When using depth-first search on the example provided in Figure 1, following the left branch first and without fathoming any nodes, the order for visiting the tree is 1-2-4-8-16-17-9-18-19-5-10-20-21-11-22-23-3-6-12-24-25-13-26-27-7-14-28-29-15-30-31. This can be implemented by selecting the item with highest efficiency when picking elements from the set $\mathcal{J}$ in Procedure `tree_search`, and by adding the left child to the front of queue $Q$, and the right child to its rear.

## 2.3 Incomplete search

Even when advanced methods are used to fathom parts of the search tree, complete search takes too long for large problems; the meaning of *large* depends on the problem being tackled, on the elaboration of the methods used for determining bounds, on the hardware used, etc. But for difficult problems, above a certain instance size, the exact exploration of the search tree is not practical. In this case, the search must be interrupted when the time (or other resource, like space/memory) used exceeds a reasonable limit. The solution found is no longer a proven optimum, but hopefully it is a good feasible solution (i.e., a lower bound to the optimum, for maximization problems); additionally, tree search also provides an upper bound (or lower bound, for minimization problems). For many practical situations this information is enough.

### 2.3.1 Depth-first

Interrupting depth-first search is trivial: it suffices to check, at each node, if the allowed time has been exceeded; if so, return immediately and report the current bounds. Depth-first search usually

keeps a low number of open nodes, so interruption due to space limits is rare.

If the time allowed to the search is severely limited, relatively to the time required to explore the whole tree, the solution found (if some) can be of poor quality. In this case, the quality of the solution is critically dependent on the item selected at each node. Hence, a good heuristic for choosing the item to explore next is essential. On the knapsack example, the profit-weight ratio was proposed; in this case, if time allows reaching a leaf of the tree, the solution found is at least as good as the greedy solution obtained using the same rule.

#### 2.3.2 Breadth-first

Breadth-first search can also trivially be interrupted based on time; it suffices to check at each node if time has been exceeded, and return without branching if so. However, in many situations the number of open nodes grows very large, quickly using all the available space. In these cases, it may be useful to include also a criterion for stopping whenever the number of open nodes is too large.

In the case of the knapsack problem, if search is interrupted when the nodes being explored are close to the root, the knapsacks probably have much space still available. For other problems, a feasible solution may not have been found yet. In this situation, it is generally rewarding to descent the tree until reaching a leaf, at least in some of the nodes. This process is called *diving*. An example of diving for the knapsack problem is to complete the solution at each node in a greedy way, and use the corresponding objective as a lower bound for the node.

## 3 Stochastic tree search

Tree search can be put into a setting of stochastic search, by means of adding a random component to any of the two main selection steps in the search: the selection of the next item of $\mathcal{J}$ to analyze, and the selection of the next node of the tree to visit. In this paper we focus on the latter.

For the design of the stochastic algorithm we want to take advantage of two facts. The first is that, in general, diving quickly allows obtaining complete solutions; if the heuristics guiding the dive is of good quality, so will hopefully be the solutions obtained. The second fact is that making use of random start on tree search many times dramatically improves the results; this was observed, for example, for the satisfiability problem [8], due to the existence of heavy tails in the solutions' distribution.

The strategy devised for taking these considerations into account relies on two queues. One queue is a stack, were the left-hand side child of each processed node is placed; the other queue is used for placing the right-hand side child, as detailed in Procedure `stochastic_tree_search`,

The usage of two queues implements the desired behaviors described earlier. Departing from any point of the tree, the stack $D$ will drive the search into a dive, until reaching a leaf. In this process, the right-hand child of the visited nodes is inserted into $D$ if it is the only choice (line 14), or into $Q$ if the left branch is feasible (line 12); in the latter case, the left-hand child is pushed into the stack $D$ (line 11). When stack $D$ is empty, the next node to visit is randomly popped from the queue $Q$; this implements the desired random behavior. However, as no part of the tree is neglected, the whole solution space is enumerated, except in the case of interruption. We are, thus, in a case of stochastic implicit enumeration.

As said earlier, for difficult, large problems, complete search of the tree is usually not affordable. We claim that if we interrupt the method proposed here, it will provide very good results. The reason is that, by concentrating the search mostly on the left part of the tree, it will visit first the most interesting solutions. This is especially true if the heuristics for selecting the branching item in each node is carefully designed for the problem being tackled.

### 3.1 Experimental results

In order to show experimentally the merit of the ideas developed in this section we devised a simple test, comparing stochastic tree search to standard depth-first search. The data were generated making use of the properties of knapsack instances reported in [9], so as to provide difficult instances with

**Procedure** `stochastic_tree_search`$(c, p, w, \mathcal{J})$

```
create empty queues D, Q
push ({}, J) into D
z* ← −∞
while D or Q not empty do
    if D not empty then pop (X, J) from D
    else randomly pop (X, J) from Q
    if Σ_{j∈X} p_j ≥ z* then z* ← Σ_{j∈X} p_j
    if J ≠ ∅ then
        j ← an element of J
        if w_j + Σ_{k∈X} w_k ≤ c then
            push (X ∪ {j}, J\{j}) into D
            push (X, J\{j}) into Q
        else
            push (X, J\{j}) into D
return z*
```

small coefficients. Notice that no advanced concepts like the core [10], or other kinds of preprocessing, were used; the results reported here are, therefore, not expected to be close to the state-of-the-art in knapsack solving.

Table 1 reports the average number of nodes required for the (complete) solution of the series of instances for depth-first search and stochastic tree search, for a series of 25 instances or each size. As can be seen, the reduction on the resource usage brought by stochastic tree search is considerable. There was no instance for which stochastic tree search was not, at least, as good as depth-first search.

The most interesting usage of stochastic tree search is likely to be for determining good solutions in a short time. Table 2 reports the best solution found with respect to the number of nodes explored, for a typical instance. The initial values obtained are identical to both methods; these are the solutions obtained in the first dive. After exploring 17 nodes, the path of the two methods is different, with a clear advantage to stochastic tree search.

# 4 Conclusions

This paper presents stochastic tree search, an alternative method to local search. Stochastic tree search consists of the exploration of a search tree, making use of heuristics for guiding the choice of the next branch, and a combination of diving and randomized selection of the path for exploring the tree.

The strongest points in favor of such method are the following:

- it avoids the repetition of the search on the same areas of the space, which is unavoidable even for carefully implemented metaheuristics;
- as opposed to depth-first search or breadth-first search, the method is not likely to be trapped in a particular area of the search tree, thanks to combination of diving and random selection of the nodes to explore;
- by concentrating first on the most promising parts of the search tree, it allows to quickly determine solutions of very good quality.

For the sake of simplicity, we have made an illustration of the application of the methods proposed to the knapsack problem. This is not meant to be a state-of-the-art solver for this problem, as many important features (like preprocessing, identification of the core variables, etc.) were not implemented.

| Instance size ($n$) | $z^G$ | $z^{UB}$ | $z^*$ | #nodes dfs | #nodes sts | % improvement |
|---|---|---|---|---|---|---|
| 4 | 202.0 | 320.0 | 206.2 | 5.6 | 5.5 | 2.1 |
| 5 | 305.2 | 392.1 | 319.4 | 8.7 | 8.6 | 0.9 |
| 6 | 371.9 | 469.5 | 397.1 | 16.4 | 15.2 | 7.6 |
| 7 | 458.8 | 543.8 | 502.3 | 23.8 | 21.7 | 9.1 |
| 8 | 536.2 | 623.5 | 573.8 | 42.9 | 38.0 | 11.5 |
| 9 | 646.5 | 690.9 | 661.4 | 57.1 | 52.4 | 8.3 |
| 10 | 716.0 | 767.9 | 743.8 | 86.6 | 62.5 | 27.9 |
| 11 | 823.9 | 860.7 | 841.0 | 115.4 | 95.0 | 17.6 |
| 12 | 923.2 | 954.7 | 935.8 | 176.7 | 141.3 | 20.0 |
| 13 | 975.1 | 1027.0 | 1005.7 | 323.0 | 243.2 | 24.7 |
| 14 | 1071.7 | 1123.3 | 1105.8 | 407.9 | 276.6 | 32.2 |
| 15 | 1165.4 | 1212.1 | 1192.7 | 707.0 | 510.9 | 27.7 |
| 16 | 1267.0 | 1307.1 | 1290.0 | 769.0 | 584.4 | 24.0 |
| 17 | 1346.1 | 1384.1 | 1367.8 | 1051.6 | 652.7 | 37.9 |
| 18 | 1426.0 | 1468.1 | 1448.3 | 1914.7 | 1149.9 | 39.9 |
| 19 | 1516.8 | 1559.2 | 1544.0 | 3214.7 | 1704.8 | 47.0 |
| 20 | 1609.8 | 1647.5 | 1634.0 | 4442.6 | 2755.5 | 38.0 |
| 21 | 1671.9 | 1716.8 | 1700.6 | 6029.6 | 4581.8 | 24.0 |
| 22 | 1768.7 | 1796.5 | 1780.4 | 6797.0 | 4444.7 | 34.6 |
| 23 | 1849.2 | 1869.8 | 1856.2 | 14815.7 | 10509.4 | 29.1 |
| 24 | 1926.7 | 1956.2 | 1938.6 | 24997.2 | 19246.3 | 23.0 |
| 25 | 1993.6 | 2026.0 | 2012.2 | 35953.2 | 25332.6 | 29.5 |
| 26 | 2080.6 | 2115.6 | 2099.2 | 46732.8 | 32780.4 | 29.9 |
| 27 | 2170.9 | 2192.5 | 2179.4 | 112437.0 | 97351.2 | 13.4 |
| 28 | 2261.7 | 2293.2 | 2280.5 | 173883.3 | 139366.2 | 19.9 |
| 29 | 2337.1 | 2363.3 | 2349.1 | 220552.9 | 152719.1 | 30.8 |
| 30 | 2448.5 | 2478.3 | 2462.0 | 442854.3 | 408471.7 | 7.8 |
| 31 | 2510.0 | 2542.6 | 2528.4 | 499979.4 | 412342.6 | 17.5 |
| 32 | 2584.9 | 2614.1 | 2601.0 | 411892.6 | 263107.7 | 36.1 |

Table 1: Results for the knapsack problem: number of items, greedy solution, initial upper bound to $z$, optimal $z$, number of searched nodes for depth-first search (dfs) and stochastic tree search (sts), percent reduction on nodes searched by the latter (average for 25 instances of each size).

| dfs | | sts | |
|---|---|---|---|
| # nodes | z | # nodes | z |
| 2 | 190 | 2 | 190 |
| 3 | 380 | 3 | 380 |
| 4 | 551 | 4 | 551 |
| 5 | 703 | 5 | 703 |
| 6 | 836 | 6 | 836 |
| 7 | 931 | 7 | 931 |
| 8 | 1026 | 8 | 1026 |
| 9 | 1121 | 9 | 1121 |
| 10 | 1178 | 10 | 1178 |
| 11 | 1235 | 11 | 1235 |
| 12 | 1292 | 12 | 1292 |
| 13 | 1349 | 13 | 1349 |
| 14 | 1406 | 14 | 1406 |
| 15 | 1425 | 15 | 1425 |
| 16 | 1444 | 16 | 1444 |
| 17 | 1463 | 17 | 1463 |
| 34 | 2063 | 19 | 2095 |
| 65 | 2082 | 42 | 2117 |
| 158 | 2095 | 3683 | 2117 |
| 3407 | 2117 | | |
| 6741 | 2117 | | |

Table 2: Log of the best solution found with respect to the number of nodes explored for a typical instance, for depth-first search (left) and stochastic tree search (right).

On the directions for future research, one interesting path to follow is to use a non-uniform way for selecting the next node to explore, giving larger probabilities to the most promising nodes, according to some heuristics. Another direction is the application of this method to other problems; the only limitation in this respect is the requirement of a good heuristics for supporting the branching decision.

# References


[1] Artigues, C., Gendreau, M., Rousseau, L.M., Vergnaud, A.: Solving an integrated employee timetabling and job-shop scheduling problem via hybrid branch-and-bound. Computers & Operations Research **36**(8) (2009) 2330 – 2340

[2] Sbihi, A.: A best first search exact algorithm for the multiple-choice multidimensional knapsack problem. J. Comb. Optim. **13**(4) (2007) 337–351

[3] Glover, F., Tangedahl, L.: Dynamic strategies for branch and bound. Omega **4**(5) (1976) 571–576

[4] Fadlalla, A., Evans, J.R.: Improving the performance of enumerative search methods-I. exploiting structure and intelligence. Computers & OR **22**(6) (1995) 605–613

[5] Fadlalla, A., Evans, J.R., Levy, M.S.: Improving the performance of enumerative search methods - part II: Computational experiments. Computers & OR **22**(10) (1995) 987–994

[6] Martello, S., Toth, P.: Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc., New York, NY, USA (1990)

[7] Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Berlin, Germany (2004)

[8] Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reasoning **24**(1/2) (2000) 67–100

[9] Pisinger, D.: Where are the hard knapsack problems? Computers & Operations Research **32**(9) (2005) 2271 – 2284

[10] Balas, E., Zemel, E.: An Algorithm for Large Zero-One Knapsack Problems. Operations Research **28**(5) (1980) 1130–1154