Chapter 2
The Knapsack Problem and Straightforward Optimization Methods

In the previous chapter we gave some examples for optimization problems in the application area of production and logistics. Recall the cargo-loading problem we described at last which consists in choosing an optimal subset of available products for shipping. In the theory of optimization this task is categorized under a special class of problems, called *packing problems*. Precisely speaking, we are facing a subclass of packing problems, called *knapsack problems*. The basic idea of optimally packing items into a single object, i.e. a knapsack in the simplest case, serves as an abstract model for a broad spectrum of packing, loading, cutting, capital budgeting or even scheduling problems. In order to provide a general basis for the subsequent chapters, we will first introduce an example knapsack optimization problem and then discuss various different approaches to solve it.

2.1 The Reference Problem

Consider a set of items each of which has a predefined weight and a monetary value (*profit*). These items have to be packed into a bag whose maximum weight is limited in such a way that it can not carry all of the available items. The problem consists in choosing a subset of items which

1. fit into the bag with respect to the weight limit and
2. yield a maximum total profit

At first glance, one might think that this task of choosing some items does not really appear to be a problem. However, in the following we will give a concrete example in order to illustrate the difficulty of such a task.

Let us consider a bag which can hold at most 113 pounds and assume that there are ten items available as listed in Table 2.1. The items add up to a total weight of 227 pounds whereas the bag's capacity is only 50 % of this value. Hence we have to make a decision which items to put into the bag such that we gain the highest possible profit. But how can we find a solution to this decision problem?

G. Zäpfel et al., *Metaheuristic Search Concepts: A Tutorial with Applications to Production and Logistics*, DOI 10.1007/978-3-642-11343-7_2, © Springer-Verlag Berlin Heidelberg 2010

Table 2.1: Items of the example problem

Item	Weight	Profit
1	32 lbs.	727 $
2	40 lbs.	763 $
3	44 lbs.	60 $
4	20 lbs.	606 $
5	1 lbs.	45 $
6	29 lbs.	370 $
7	3 lbs.	414 $
8	13 lbs.	880 $
9	6 lbs.	133 $
10	39 lbs.	820 $

Table 2.2: Items sorted by profit

Item	Weight	Profit
8	13 lbs.	880 $
10	39 lbs.	820 $
2	40 lbs.	763 $
1	32 lbs.	727 $
4	20 lbs.	606 $
7	3 lbs.	414 $
6	29 lbs.	370 $
9	6 lbs.	133 $
3	44 lbs.	60 $
5	1 lbs.	45 $

An intuitive approach could be the following: Let us first sort the given items by profit in descending order as shown in Table 2.2. Now we add one item after the other to the bag as long as the capacity limit is not exceeded. In each step of this procedure we check if the next item in turn fits into the bag. If this is not the case, we skip this item and try the next one.

As for our example we would first add items #8, #10 and #2 to the bag, which leads to a total profit of 2463 $ at 92 pounds. In the next step we encounter item #1 with a weight of 32 lbs. Adding this item would result in a total weight of 124 lbs., which is too much for our bag. So we skip it and proceed with the next one (item #4). We can add this item since we have 112 pounds afterwards. Now we just have one pound left in our bag and therefore the only matching item is #5. Finally we obtain a completely filled bag yielding a total profit of 3114 $. Table 2.3 gives a detailed overview of the procedure we have described so far.

The solution we achieved can be written in a set notation just listing the items we have chosen in no specific order:

$$\{8, 10, 2, 4, 5\}$$

Table 2.3: Choosing the items to be added to the bag

Step	Item	Add ?	Total weight	Total profit
1	8	yes	13 lbs.	880 $
2	10	yes	52 lbs.	1700 $
3	2	yes	92 lbs.	2463 $
4	1	no		
5	4	yes	112 lbs.	3069 $
6	7	no		
7	6	no		
8	9	no		
9	3	no		
10	5	yes	113 lbs.	3114 $

Another way to display our solution is to specify for each item whether it is to be included in the bag or not:

Item #	1	2	3	4	5	6	7	8	9	10
Included ?	no	yes	no	yes	yes	no	no	yes	no	yes

Alternatively we can store this information in a vector, where the first position corresponds to item #1 and the last one to item #10:

$$(\text{no}, \text{yes}, \text{no}, \text{yes}, \text{yes}, \text{no}, \text{no}, \text{yes}, \text{no}, \text{yes})$$

If we adopt a more mathematical notation, the vector from above can be transformed by replacing "yes" and "no" with the numerical values 1 and 0 respectively:

$$(0,1,0,1,1,0,0,1,0,1)$$

So we have determined a feasible solution[1] for our example, but the following questions still remain open:

- Does there exist a different subset of items which yield an even higher total profit?
- What is the best possible (optimal) solution for our problem and how far are we still away from that solution?

In the following sections we will deal with these questions and thereby introduce methods which enable us to answer them at least partly. We will repeatedly refer

[1] In the strict sense, we may only call the optimal subset of items a "solution" to the problem, because besides feasibility, the requirement for optimality is part of the problem statement. Consequently, an arbitrary feasible solution has to be referred to as a *solution candidate* or *candidate solution* as long as its optimality status is unclear. However, in accordance with related literature, we drop this rigorous distinction and use the terms "solution candidate" and "solution" equivalently in this book. The best possible solution (candidate) will be explicitly referred to as the *optimal solution*.

to the example problem presented here and use it to explain the concepts and ideas behind the different approaches to solve optimization problems.

2.2 An Additional Greedy Approach

In the preceding section we presented an intuitive solution procedure for the knapsack problem. In each step of this procedure we tried to increase the total profit as much as possible by choosing the most valuable item out of the remaining ones. This principle can be generalized to other optimization problems where some target parameter has to be maximized: One would choose in each step the solution component with the greatest impact on the the target parameter. Such a course of action is called a *greedy approach* or also *greedy algorithm* in optimization theory.

In the following we will describe a further greedy algorithm for the knapsack problem. We adopt the same core strategy as we did in Section 2.1, but the rating of items is performed in a more sophisticated way. Instead of simply using the profit values we are now interested in how much profit we can gain *per unit of weight*.

Consider items #4 and #7 from our example. According to our previous strategy we would first add item #4 because it yields a higher profit (606 $ vs. 414 $). But if we take a closer look we realize that item #7 is much lighter than item #4 (3 lbs. vs. 20 lbs.). Is it possibly better to prefer the lighter item although its profit is inferior? In order to answer this question we have to compute the *profit to weight ratio* or *efficiency* for each item:

$$\text{item \#4:} \qquad \frac{606\ \$}{20\ \text{lbs.}} = 30.30\ \text{lbs. per pound} \tag{2.1}$$

$$\text{item \#7:} \qquad \frac{414\ \$}{3\ \text{lbs.}} = 138.00\ \text{lbs. per pound} \tag{2.2}$$

Indeed item #7 has a greater efficiency: In proportion to its weight it yields a higher profit than item #4 and occupies less capacity of the bag.

In analogy to equations (2.1) and (2.2) we determine the efficiency of every item as shown in Table 2.4.

Now we can apply the greedy principle to the profit to weight ratios, resulting in the sequence of steps displayed in Table 2.5. We finally obtain the following subset of items:

$$\{7,8,5,4,1,9,6\}$$

Again this can be written as:

$$(1,0,0,1,1,1,1,1,1,0)$$

This solution is slightly better than the one obtained by the simple value-based greedy approach. So may we conclude that the efficiency-based version is superior?

Table 2.4: Items of the example problem sorted by efficiency.

Item	Weight	Profit	Efficiency
7	3 lbs.	414 $	138.00 $ per pound
8	13 lbs.	880 $	67.69 $ per pound
5	1 lbs.	45 $	45.00 $ per pound
4	20 lbs.	606 $	30.30 $ per pound
1	32 lbs.	727 $	22.72 $ per pound
9	6 lbs.	133 $	22.17 $ per pound
10	39 lbs.	820 $	21.03 $ per pound
2	40 lbs.	763 $	19.08 $ per pound
6	29 lbs.	370 $	12.76 $ per pound
3	44 lbs.	60 $	1.36 $ per pound

Table 2.5: Choosing the items to be added to the bag

Step	Item	Add ?	Total weight	Total profit
1	7	yes	3 lbs.	414 $
2	8	yes	16 lbs.	1294 $
3	5	yes	17 lbs.	1339 $
4	4	yes	37 lbs.	1945 $
5	1	yes	69 lbs.	2672 $
6	9	yes	75 lbs.	2805 $
7	10	no		
8	2	no		
9	6	yes	104 lbs.	3175 $
10	3	no		

Unfortunately this is not the case and the performance of both greedy algorithms actually depends on the problem instance [121].

2.3 Solving the Knapsack Problem by Enumeration

In Section 2.1 we raised the question whether we can find a better solution for the example problem than the intuitive one. The previous section revealt that it is indeed possible to find a better solution by using a slightly modified approach. But we still do not know if this is already the best possible selection of items.

The easiest way to find an answer to this question is to enumerate *all* possible subsets of items and to check each such selection for:

1. *Feasibility*:
   Is the total weight of the given items lower or equal to the bag's weight limit?
2. *Total Profit*:
   The total profit is only computed if the examined item selection is feasible.

Remark that this approach is completely different from the greedy-based one. A greedy method tries to generate *one* single solution by successively assembling preferable solution components, whereas in the case of enumeration we examine one entire solution *per step* until we have seen all possible ones. Hence we perform an *exhaustive search* for the optimum solution.

In order to enumerate all solutions, we have to build all possible combinations of zeros and ones in the solution vector. Concerning our example, this can be accomplished in the following manner: Initially we create a vector with all elements set to zero, which is our first feasible solution. As a next step we set the rightmost element (item #10) to 1. Then we proceed by resetting element #10 and setting #9 to 1. Our next solution is a vector with element #10 also set to 1. We continue by resetting #9 and #10 to 0 and putting a 1 at position #8. Table 2.6 gives an overview of the enumeration process for the 10-item example problem. It can be seen that solution vector #96 yields a total profit of 3223 $ which is better than both greedy solutions obtained so far. Solutions #590 and #592 further improve the profit to 3447 $ and finally 3580 $ - the actual optimum solution for the example problem.

In view of this result - an 11.3 % improvement over the best greedy solution - one may ask why we are even bothering with more advanced approaches. The reason is that solving the problem in an enumerative way is only practical for knapsack problems with a small number of items, since the amount of possible subsets of items (solutions) increases by a factor of 2 for each additional item. The formula by which we can compute the total number of possible solutions is given by

$$\textit{Number of possible solutions} = 2^{\textit{number of items}}$$

Hence given 11 items we already have 2048 possible solutions. This kind of growth is called *exponential* in mathematical terminology.

Table 2.7 lists the total number of solutions for various different problem sizes. The computation times have been determined by running the enumeration method on a standard personal computer[2]. Experiments yielded an average performance of 1,677,000 examined solutions per second, which appears to be very much at first glance. But when looking at the number of solutions that have to be checked in order to solve problems with more than 40 items, this number becomes almost insignificant.

As a consequence, the exhaustive search for the optimum solution cannot be regarded as a competitive means for solving problems of reasonable size. However, this does not disqualify the principle of enumeration entirely. When applied in the context of sophisticated techniques for restricting the number of solutions to look at, this concept may exhibit a considerable increase in performance. In the following section we introduce a technique by which we can omit whole sets of solutions during the search as soon as we realize that they are negligible.

[2] Pentium-4 CPU 2.66 GHz, 1 GB RAM, Microsoft .NET Framework 2.0, C# Programming Language

Table 2.6: Enumeration of all possible solution vectors for the example problem

Solution #	Vector	Weight	Feasible ?	Profit	Best so far
1	(0,0,0,0,0,0,0,0,0,0)	0 lbs.	yes	0 $	
2	(0,0,0,0,0,0,0,0,0,1)	39 lbs.	yes	820 $	#2
3	(0,0,0,0,0,0,0,0,1,0)	6 lbs.	yes	133 $	
4	(0,0,0,0,0,0,0,0,1,1)	45 lbs.	yes	953 $	#4
5	(0,0,0,0,0,0,0,1,0,0)	13 lbs.	yes	880 $	
⋮	⋮	⋮	⋮	⋮	⋮
88	(0,0,0,1,0,1,0,1,1,1)	107 lbs.	yes	2809 $	#88
89	(0,0,0,1,0,1,1,0,0,0)	52 lbs.	yes	1390 $	
90	(0,0,0,1,0,1,1,0,0,1)	91 lbs.	yes	2210 $	
91	(0,0,0,1,0,1,1,0,1,0)	58 lbs.	yes	1523 $	
92	(0,0,0,1,0,1,1,0,1,1)	97 lbs.	yes	2343 $	
93	(0,0,0,1,0,1,1,1,0,0)	65 lbs.	yes	2270 $	
94	(0,0,0,1,0,1,1,1,0,1)	104 lbs.	yes	3090 $	#94
95	(0,0,0,1,0,1,1,1,1,0)	71 lbs.	yes	2403 $	
96	(0,0,0,1,0,1,1,1,1,1)	110 lbs.	yes	3223 $	#96
⋮	⋮	⋮	⋮	⋮	⋮
128	(0,0,0,1,1,1,1,1,1,1)	111 lbs.	yes	3268 $	#128
⋮	⋮	⋮	⋮	⋮	⋮
154	(0,0,1,0,0,1,1,0,0,1)	115 lbs.	no	1664 $	
155	(0,0,1,0,0,1,1,0,1,0)	82 lbs.	yes	977 $	
156	(0,0,1,0,0,1,1,0,1,1)	121 lbs.	no	1797 $	
⋮	⋮	⋮	⋮	⋮	⋮
590	(1,0,0,1,0,0,1,1,0,1)	107 lbs.	yes	3447 $	#590
591	(1,0,0,1,0,0,1,1,1,0)	74 lbs.	yes	2760 $	
592	(**1**,**0**,**0**,**1**,**0**,**0**,**1**,**1**,**1**,**1**)	**113 lbs.**	**yes**	**3580** $	**#592**
⋮	⋮	⋮	⋮	⋮	⋮
622	(1,0,0,1,1,0,1,1,0,1)	108 lbs.	yes	3492 $	
⋮	⋮	⋮	⋮	⋮	⋮
639	(1,0,0,1,1,1,1,1,1,0)	104 lbs.	yes	3175 $	
⋮	⋮	⋮	⋮	⋮	⋮
1024	(1,1,1,1,1,1,1,1,1,1)	227 lbs.	no	4818 $	

The solution space

Beforehand a concept is introduced which is later referred to, the so called *solution space*.

The solution space is the set of all candidate solutions. This is what was just seen in search by enumeration, e. g. in the case of the 10 items knapsack problem there are 1024 solutions (cf. table 2.6 with solution #1, #2, ..., #1024), those solutions build the solution space.

Table 2.7: Number of possible solutions dependent on the number of items

# of items	# of solutions	Time
10	1024	$<$ 0.1 sec.
11	2048	$<$ 0.1 sec.
12	4096	$<$ 0.1 sec.
15	32,768	$<$ 0.1 sec.
20	1,048,576	$\approx$ 0.6 sec.
25	33,554,432	$\approx$ 20 sec.
30	1,073,741,824	$\approx$ 40 sec.
40	1,099,511,627,776	$\approx$ 7.5 days
50	1,125,899,906,842,624	$\approx$ 21 years
60	1,152,921,504,606,846,976	$\approx$ 21790 years

Figure 2.1 shows a possible visualization of the solution space for the knapsack problem[3].

Every point on the x-axis equals one solution (from solution #1 to solution #1024), where on the y-axis the respective solution quality is plotted. The solutions are sorted in order of appearance, i. e. which is determined by the employed enumeration scheme. Hence, if the solutions would be produced in an different way, the chart would look completely different.

Although if you look at the figure it may seem obvious which solution is the best, this is not the case. To be able to draw the chart all solutions must be evaluated.

In practice you do not know how the search space looks like nor do you know which is the best solution. Typically the solution space is a rather abstract concept. So the best thing is to think of the solution space as set of solutions and not let you mislead by the pictures included here and in the forthcoming chapters.

This illustration is here to show you the difference in the amount of solution evaluations and search performance (i. e. the amount of generated solutions).

Advanced search methods try to explore the solution space more efficiently. In the upcoming chapters we will see how this can be done and what benefits, pitfalls and drawbacks one must cope with.

To summarize this short description: the solution space is the set of all possible solutions. The difficulty dealing with the solution space is that for most problems it is not possible to explore it completely.

In the next section we will introduce a search strategy which (under certain conditions) is able to find the best solution without the need of exploring the whole solution space, as it enumerates solutions in a much more efficient manner.

[3] In this chart infeasible solutions are included too.

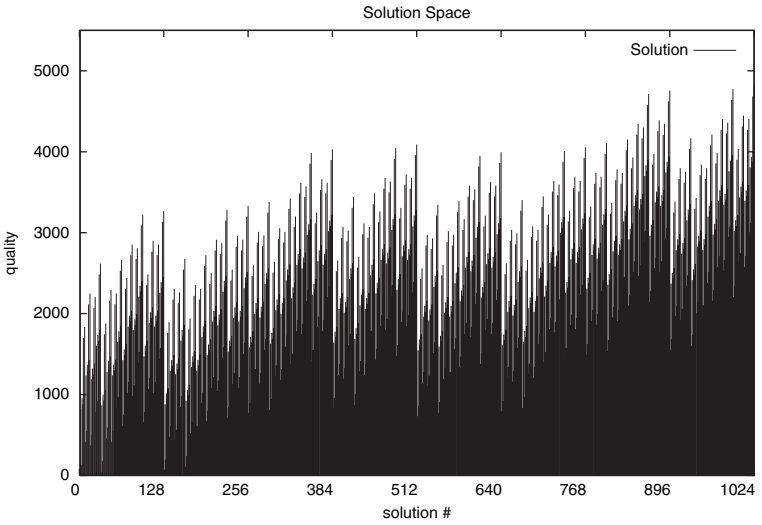


Fig. 2.1: Solutions produced by complete enumeration

2.4 Branch and Bound

Clearly, to look at each possible solution is not the shortest path to the optimum because you have to look at many solutions and a lot of them are worse than the best known so far. So, are there other, i.e. more intelligent, ways to find the best possible solution?

One idea is to look at promising solutions only. That means if we have a solution we look at those solutions which are capable to outperform the current one.

The question is: How can we achieve this?

The basic idea is to try only those combinations of items which can potentially lead to better solutions then the current best one. On the other hand, if we found an item configuration which definitely leads to a worse solution, we may not further investigate this option.

How does this work? The solution is built step by step. At each iteration one property of the solution is added. The first thing we need is a rule which decides which item is added next. For the sake of convenience we use profit to weight ratio here too, take a look at table 2.4 to recall the values. We add items in descending order of their ratio. Other strategies, i. e. highest profit as in the first greedy approach, would be possible, too.

We start with an empty bag. The first decision we have to make is: do we include item #7 in the final solution or not? This decision leads to two different solutions. Figure 2.2 illustrates this situation. The node (the squares a referred to as nodes) on the top represents the empty bag. The node on the left is a solution where item #7 is included and on the right there is a solution where it is excluded. This decision is

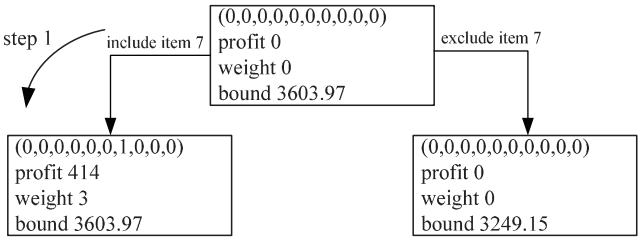


Fig. 2.2: First step of the Branch&Bound algorithm

fixed for all following nodes on the respective sides, i. e. any node connected on the left side has item #7 included.

Each node has 4 different properties[4]: At the top there is the solution vector, the second line is the profit, followed by the weight of the actual solution vector and finally the *bound*.

Most of the information was already present in the previous methods explained, but the entry labeled *bound* is new. This value tells us how much maximal profit the bag can yield with the remaining space. Here it is calculated in the following way: It is started with the actual fixed item configuration, which gives the profit and weight. Now items are "virtually" added to the bag until none can further be inserted without violating a constraint, i.e. the bag exceeds the maximum weight. Two situations can arise:

- The bag is full, then the bound is the profit of the virtually filled bag.
- The bag has some space left, then the remaining space is filled with a fraction of an not yet included item.

To visualize this, the bounds for figure 2.2 are calculated as example. On left left path item #7 is already in the bag, so we start with a profit of 414 $. Now we add items #8, #5, #4, #1, #9, which gives a profit of 414 $ + 880 $ + 45 $ + 606 $ + 727 $ + 133 $ = 2805 $ and a weight of 3 lbs. + 13 lbs. + 1 lbs. + 20 lbs. + 32 lbs. + 6 lbs. = 75 lbs. Because item #10 has a weight of 39 lbs. it does not fit completely in the bag, there are only 38 lbs. left. But we do not care about such things, we simply add as much of the last item as possible. In this case it is $38\text{lbs.}/39\text{lbs.} \cdot 820\$ = 798.97\$$. So the theoretical absolute maximum which could be inserted into the bag is 2805 $ + 798.97 $ = 3603.97. This value is called the upper bound (in a maximization problem).

It is clear that there exists no combination of items which could supersede this value. Let us calculate this value for the right path in figure 2.2: Item #7 is already excluded in this solution, so we add item #8, #5, #4, #1, #9, #10. This results in a profit of 3211 $ and a weight of 111 lbs. Here we take $2\text{lbs.}/40\text{lbs.}$ of item #2. So we get the following bound $3211\ \$ + 1\text{lbs.}/20\text{lbs.} \cdot 763\ \$ = 3249.15\ \$$.

This bound is much smaller than the bound on the left side, because item #7 is excluded from the solution.

[4] For easier reading units are omitted in the illustrations.

Now that we calculated all properties we need, we continue the search. We can either take the left node (path "include item #7") or the right node (path "exclude item #7"). As the left node promises higher maximum profit (the bound value of the left node is higher than the bound value in the right node), we continue there[5]. This step is called *branching*.

There we have the choice of adding item #8 or not. So, again, we get two child nodes. Figure 2.3 illustrates this situation. We see that on the left subnode (path "include item #8") the bound stays the same and on the right side it decreases again, because of that we take the left path (step 2).

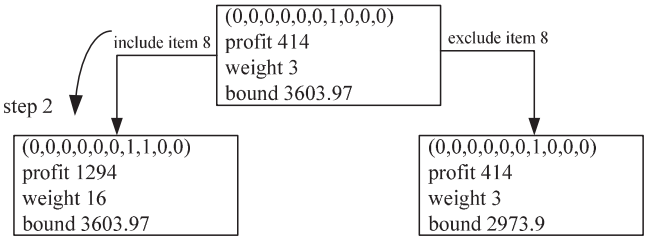


Fig. 2.3: Second choice; include item 8 in final solution or not

As the bag is not full yet, we continue to add items. We proceed with the left node, because it has a better bound and consider item #5:

- (0, 0, 0, 0, 1, 0, 1, 1, 0, 0), profit 1339 $, weight 17 lbs., bound 3603.97
- (0, 0, 0, 0, 0, 0, 1, 1, 0, 0), profit 1294 $, weight 16 lbs., bound 3580

Figure 2.4 illustrates the first three choices that are made.

We continue to expand the nodes with the better bound (i.e. walk further down the search tree). The next step is to decide whether include or exclude item #4 (step 4, figure 2.5):

- (0, 0, 0, 1, 1, 0, 1, 1, 0, 0), profit 1945 $, weight 37 lbs., bound 3603.97
- (0, 0, 0, 0, 1, 0, 1, 1, 0, 0), profit 1339 $, weight 17 lbs., bound 3381.43

Now the decision for item #1 must be made (step 5).

- (1, 0, 0, 1, 1, 0, 1, 1, 0, 0), profit 2672 $, weight 69 lbs., bound 3603.97
- (0, 0, 0, 1, 1, 0, 1, 1, 0, 0), profit 1945 $, weight 37 lbs., bound 3489.32

The last step which is illustrated in figure 2.5 concerns item #9.

- (1, 0, 0, 1, 1, 0, 1, 1, 1, 0), profit 2805 $, weight 75 lbs., bound 3603.97
- (1, 0, 0, 1, 1, 0, 1, 1, 0, 0), profit 2672 $, weight 69 lbs., bound 3587.38

The following steps are depicted in figure 2.6, step 7; we process item #10:

[5] The choice which node is used next is very important. Here we use a depth first search strategy. Another strategy would be breadth first search, where the nodes on the same level are expanded first.

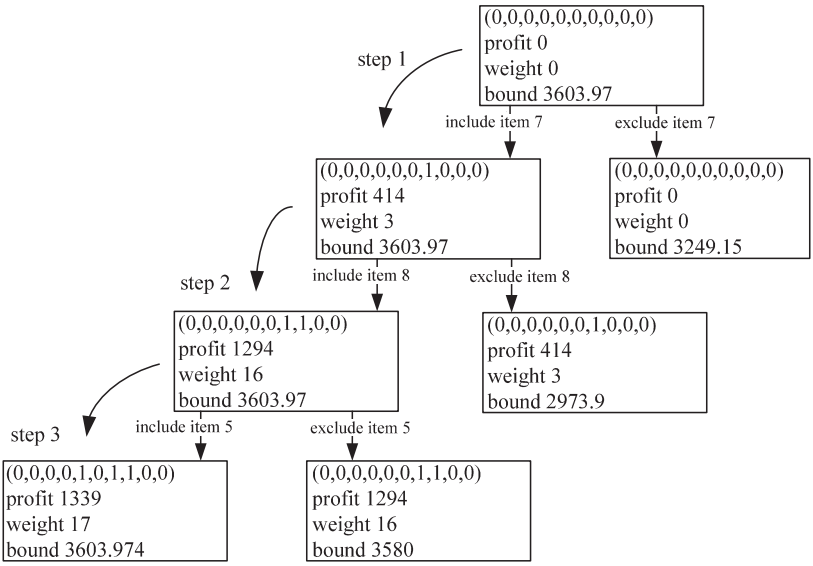


Fig. 2.4: First three steps of the branch and bound algorithm

- (1, 0, 0, 1, 1, 0, 1, 1, 1, 1), profit 2625 $, weight 114 lbs., bound 3625
- (1, 0, 0, 1, 1, 0, 1, 1, 1, 0), profit 2805 $, weight 75 lbs., bound 3529.85

Now we face our first problem, item 10 does not fit in the bag. It is not possible to further add any items, so we can skip calculations with the remaining items #2, #6 and #3.

We go one step back and from there exclude item #10 (step 8). The best combination of items we found so far is: (1, 0, 0, 1, 1, 0, 1, 1, 1, 0) with a profit of 2805 $ and a weight of 75 lbs. But this solution is not yet complete, we still have to decide what we do with the remaining items #2, #3 and #6.

The next item to process is #2. But if we include it (step 9) the weight exceeds the maximum weight. So we exclude it (step 10) and get the following item:

- (1, 0, 0, 1, 1, 0, 1, 1, 1, 0), profit 2805 $, weight 75 lbs., bound 3187.27

Now we must decide what we do with item #6:

- (1, 0, 0, 1, 1, 1, 1, 1, 1, 0), profit 3175 $, weight 104 lbs., bound 3187.27
- (1, 0, 0, 1, 1, 0, 1, 1, 1, 0), profit 2805 $, weight 75 lbs., bound 2856.82

Item #6 fits in the bag, so we continue with included item #6, step 11.
The last decision is what we do with item #3:

- (1, 0, 1, 1, 1, 1, 1, 1, 1, 0), profit 3235 $, weight 148 lbs., bound 3235
- (1, 0, 0, 1, 1, 1, 1, 1, 1, 0), profit 3175 $, weight 104 lbs., bound 3175

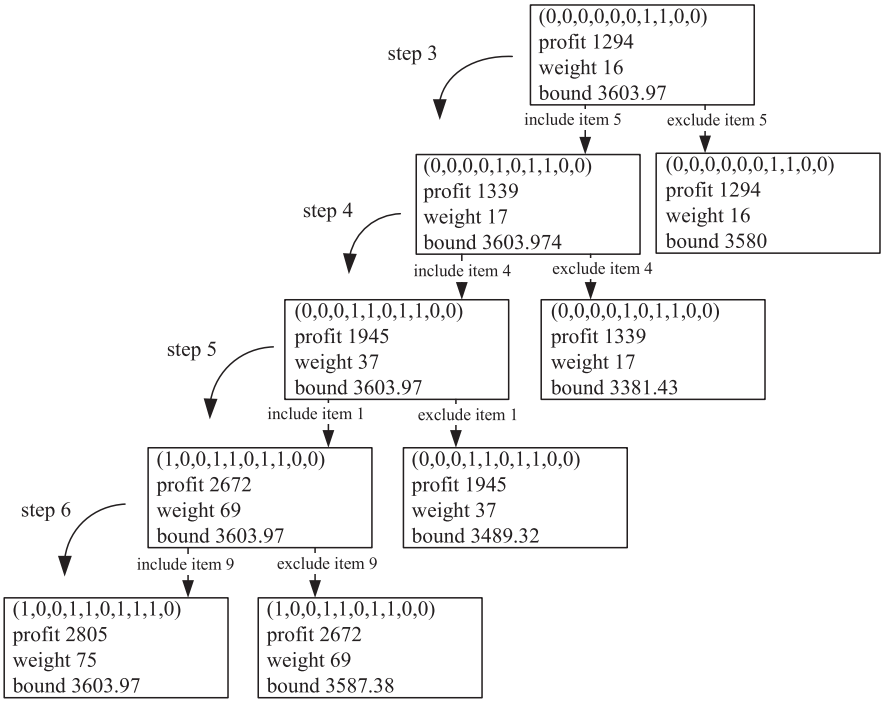


Fig. 2.5: The algorithm continues to add items to the solution

We see in step 12 that item #3 is too big, so we do not include it. The first complete solution is (1, 0, 0, 1, 1, 1, 1, 1, 1, 0), found in step 13, with a profit of 3175 $ and a weight of 104 lbs.

We already know this solution; it is exactly the one the greedy method found. But we know there exists a better solution, because the enumeration gave us a better one.

We call the profit of the solution (3175 $) the *lower bound*. This is the quality of the currently best known solution. Now we continue the search and as we encounter nodes, we can check if the theoretical maximum is higher or lower as the current lower bound. If it is lower we may not look closer at that solution, because it can not get better than the current one. This is the key to Branch&Bound.

Let us now continue the search. We already explored the part where item #6 is included. But what happens if we exclude item #6? We already calculated the node which does not contain item #6, here are again the corresponding nodes (resulting from step 11 and step 14):

- (1, 0, 0, 1, 1, 1, 1, 1, 1, 0), profit 3175 $, weight 104 lbs., bound 3187.27
- (1, 0, 0, 1, 1, 0, 1, 1, 1, 0), profit 2805 $, weight 75 lbs., bound 2856.82

As the bound with excluded item #6 (step 14) is lower than the current lower bound, we can skip this subtree.

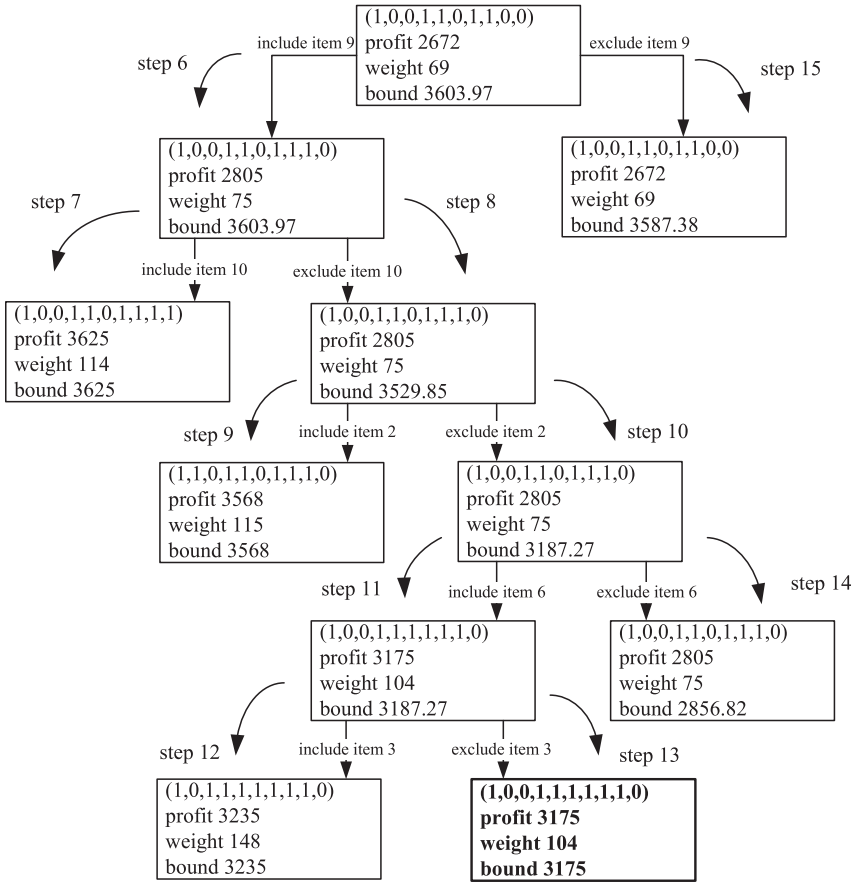


Fig. 2.6: The next search steps; the first feasible solution is marked bold

But where is the best solution. Figure 2.6 shows the subtree containing item #9. In contrast Figure 2.7 shows the corresponding other part of the tree, where item #9 is not included.

Recall, the lower bound is currently at 3175. Item #9 is permanently excluded in this subtree (step 15).

We proceed by calculating bounds for item #10, which gives the following nodes:

- (1, 0, 0, 1, 1, 0, 1, 1, 0, 1), profit 3492 $, weight 108 lbs., bound 3587.38
- (1, 0, 0, 1, 1, 0, 1, 1, 0, 0), profit 2672 $, weight 69 lbs., bound 3486.03

Item #10 fits in the bag and the bound is higher than the current lower bound, we branch (with step 16) into the subtree. Next item #2 is included (step 17), but the weight exceeds the limit. We follow the path “exclude item 2” (step 18), which gives the node:

- (1, 0, 0, 1, 1, 0, 1, 1, 0, 1), profit 3492 $, weight 108 lbs., bound 3555.79

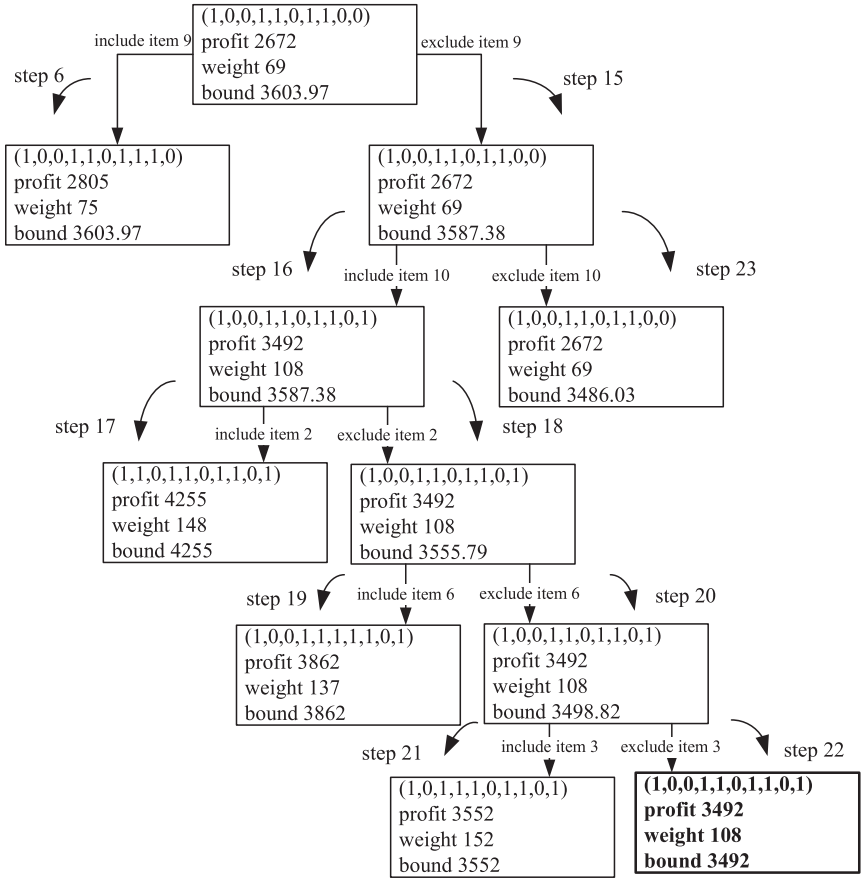


Fig. 2.7: Subtree where the second complete solution is found

Item #6 does not fit in the bag (step 19), so we continue without item #6 (step 20):

- (1, 0, 0, 1, 1, 0, 1, 1, 0, 1), profit 3492 $, weight 108 lbs., bound 3498.82

Lastly, we must decide if we include item #3. But, as with the other items, it is not enough space in the bag (step 21). So we do not include item #3, which results in the complete solution:

- (1, 0, 0, 1, 1, 0, 1, 1, 0, 1), profit 3492 $, weight 108 lbs., bound 3492

We have found a new solution, which is better than the old one. It has a profit of 3492 $ and so a lower bound of 3492.

We examined all branches in this subtree except the one excluding item #10 (step 23). We see that the upper bound is lower than our current lower bound, so we prune it.

Now we track back in the search tree to the position where the decision if item #1 is included or not is made. Please refer to figure 2.5 on page 19. As we see there if we exclude item #1 the upper bound is 3489.32, so we prune this subtree and go one step back, where we exclude item #4. There the upper bound is 3381.43 which is lower then the lower bound, again we skip this subtree and go upwards.

We are at the position where only items #7 and #8 are fixed. The next subtree is illustrated in figure 2.8.

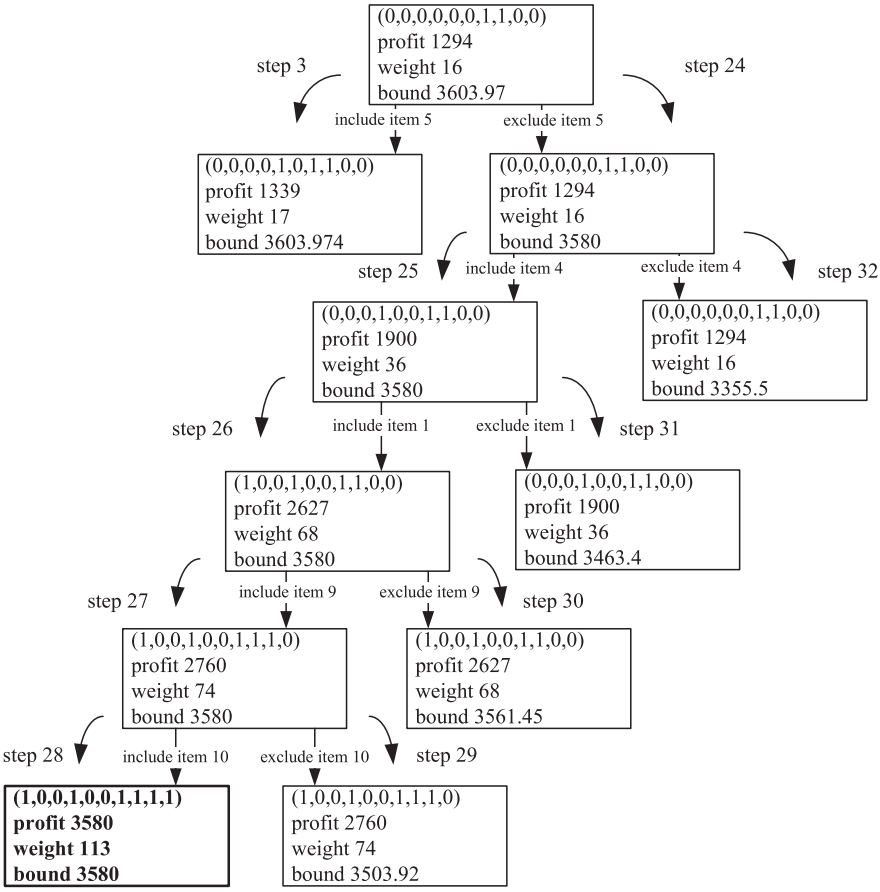


Fig. 2.8: Subtree where item #5 is excluded

By permanently excluding item #5, we get the following node (step 24):

- (0, 0, 0, 0, 0, 0, 1, 1, 0, 0), profit 1294 $, weight 16 lbs., bound 3580

This node looks promising because the upper bound is higher than the current lower bound. So we branch into this subtree.

After we excluded item #5, we take a closer look at item #4:

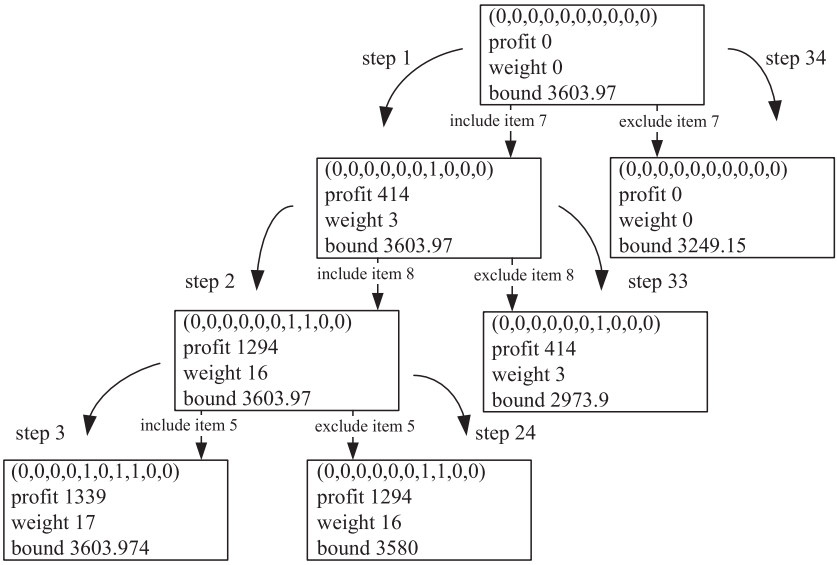


Fig. 2.9: Every node must be checked, if it can lead to better solutions

- (0, 0, 0, 1, 0, 0, 1, 1, 0, 0), profit 1900 $, weight 36 lbs., bound 3580
- (0, 0, 0, 0, 0, 0, 1, 1, 0, 0), profit 1294 $, weight 16 lbs., bound 3355.5

With step 25 we branch into the subtree containing item #4 because it has a higher bound than without item #4:

- (0, 0, 0, 1, 0, 0, 1, 1, 0, 0), profit 1900 $, 36 lbs., 3580
- (0, 0, 0, 0, 0, 0, 1, 1, 0, 0), profit 1294 $, 16 lbs., 3355.5

Then we decide whether to include item #1:

- (1, 0, 0, 1, 0, 0, 1, 1, 0, 0), profit 2627 $, weight 68 lbs., bound 3580
- (0, 0, 0, 1, 0, 0, 1, 1, 0, 0), profit 1900 $, weight 36 lbs., bound 3463.4

We take step 26 and inspect the influence of item #9 to the solution:

- (1, 0, 0, 1, 0, 0, 1, 1, 1, 0), profit 2760 $, weight 74 lbs., bound 3580
- (1, 0, 0, 1, 0, 0, 1, 1, 0, 0), profit 2627 $, weight 68 lbs., bound 3561.45

We include item #9 (step 27) and finish the branch by including (step 28) and excluding (step 29) item #10:

- (1, 0, 0, 1, 0, 0, 1, 1, 1, 1), profit 3580 $, weight 113 lbs., bound 3580
- (1, 0, 0, 1, 0, 0, 1, 1, 1, 0), profit 2760 $, weight 74 lbs., bound 3503.92

We found a new best known solution: (1, 0, 0, 1, 0, 0, 1, 1, 1, 1) with a profit of 3580 $ and a weight of 113 lbs. Now we replace the current lower bound of 3492 with the new one (3580).

We have found the optimal solution. We know this, because the enumerative method already gave us the optimal solution. But normally we would not know that this solution is optimal, so we must continue the search until we have expanded every subtree which we can not prune.

So let us take a look at the rest of the search tree. If we exclude item #10, the solution is feasible but surely worse than if we include item #10.

We go back and exclude item #9 (step 30 in figure 2.8):

- (1, 0, 0, 1, 0, 0, 1, 1, 0, 0), profit 2627 $, weight 68 lbs., bound 3561.45

Here the upper bound is lower than the lower bound and we can discard the subtree.

Next follows the exclution of item #1 (step 31):

- (0, 0, 0, 1, 0, 0, 1, 1, 0, 0), profit 1900 $, weight 36 lbs., bound 3463.4

Again, the upper bound is worse than the lower bound. No further search in this subtree is required.

Does the exclusion of item #4 yield to better solutions:

- (0, 0, 0, 0, 0, 0, 0, 1, 1, 0), profit 1294 $, weight 16 lbs., bound 3355.5

As we see, the answer is no.

Now we have two choices left (see figure 2.9, step 33 and step 34) whereas in both cases the upper bound is lower than the current lower bound. We discard the subtrees, the search is finished. We found the optimal solution: (1, 0, 0, 1, 0, 0, 1, 1, 1, 0) with a profit of 3580 and a weight of 113.

Figure 2.10 shows the complete Branch&Bound tree.

The optimal solution was found after only looking at 38 nodes, instead of 1024 in the case of exhaustive serach. And the best thing is, we know for sure that we found the optimal solution.

In the case of the knapsack problem the Branch&Bound algorihm is intuitive and works very well. This is because the problem can be divided into independent decisions which build the solution and for every non complete solution the upper bound can be calculated easily, i. e. for every item there is a branch where it is included and a branch where it is excluded.

But what is really necessary to solve a problem with the Branch&Bound approach?

In case of the knapsack problem it was easy. For every item two branches are created, one in which the item is included and one in which the item is excluded, i. e. two independet sets of solutions are built. This is the first key property of Branch&Bound. Each decision must partition the solutions into independent subsets.

The other key property is the calculation of the bound, which is an approximation of the quality achievable by the actual solution configuration. In the knapsack example the sum of the actual profit and the fractional weight and profit of the last item which fills the bag completely was used. It can be seen in the example that the upper bound always overestimates the real achievable quality, which is a necessary condition for Branch&Bound algorithms to work. However the value should

be as close as possible but must not lie under the real quality of the best solution obtainable using the current configuration.

Another important decision is the branching strategy, i.e. in which order the nodes are processed. In the example a depth first search strategy was used, were a complete subtree was explored until a leaf was reached (the upper bound converges to the real value). An alternative approach is the breadth first strategy, which first expands all nodes on the same depth and then goes one level deeper. Both strategies are illustrated in figure 2.11. The nodes are numbered in order of their visiting. A third possible traversing strategy is best first, which means the node with the best bound is expanded next.

The chosen strategy has a big influence on the performance of the search. So decisions must be made with care or based on experimentation. Here for example we used the profit to weight ratio, pack first, depth first strategy. That means always the item which has the best profit to weight ratio is chosen and added to the bag. An alternative strategy would have been to exclude that item. As you can image this strategy would be worse, because excluding promising items is not wise. With such a strategy the search must expand 334 nodes (in the given knapsack example), that is ten times more than with the best ratio inclusion strategy. But nonetheless much fewer than in exhaustive search.

Branch&Bound can dramatically reduce the effort in search but it is not always as easily applicable as in this case. Sometimes it is very hard to calculate an effective upper bound, so for other problems it is harder applicable.

One pleasant property has not yet been mentioned. If Branch&Bound does not give the optimal solution in reasonable time, it is possible to abort the search and take the best found solution so far as answer. Because we know the quality of that solution and the upper bound of the empty solution, we know the maximal discrepancy to the optimal solution in the worst case. Because the upper bound is an approximation to the optimal solution, we do not know the difference to its real value.

But again for big problem instances this may not be sufficient, because even to find a reasonable good solution may take quite a while.

2.5 Summary

In this chapter we have given some indications for how difficult it can be to find an optimal solution for an optimization problem, even if it appears to be very simple at first glance. Based on a concrete knapsack problem we demonstrated different ways to select the best possible combination of items to be placed into a bag such as to maximize the total profit.

In Sections 2.1 and 2.2 we presented an intuitive strategy for placing items in the bag *one after the other* according to priorities which correspond to

- the highest profit among all items not yet contained in the bag or
- the highest profit per pound ratio

where the latter prioritization principle turned out to be more effective. Indeed we can expect this method to yield a solution of reasonable quality, not only for our example but also for knapsack problems in general. However, this is in no way guaranteed and it is also not possible to make a statement on how close to the optimum we actually may get by this. In the area of problem solving, methods which exhibit the latter properties are called "heuristics" (cf. e.g. [76], [178]). This term originates from the greek word "heuriskein" which means "to discover" or "to find". According to [162], heuristics are

> "popularly known as rules of thumb, educated guesses, intuitive judgments or simply common sense. In more precise terms, heuristics stand for strategies using readily accessible though loosely applicable information to control problem-solving processes in human beings and machine(s)."

In fact, the term "heuristic" is fundamental to this book, though very general in its meaning since heuristics cover a broad spectrum of conceptually different solution methods. We provide a more detailed differentiation of heuristic methods in the forthcoming chapters.

As a counterpart to the intuitive approach, we introduced two methods for determining the *actual* optimum solution for our knapsack problem in Sections 2.3 and 2.4. The solution these methods provide is provably optimal, which means that we know *for sure* that there does not exist any better solution. Methods of this type are called *exact* in the mathematical sense.

Obviously, the complete enumeration approach is of almost no practical applicability. It simply becomes infeasible for knapsack problems involving more than 40 items[6] due to excessive computation time. For other kinds of optimization problems, even smaller instances may turn out to be intractable using this method, depending on their structural properties.

On the other hand, despite the theoretically effective concept of pruning unnecessary parts of the solution space, the Branch and Bound procedure generally remains computationally expensive. This is mainly due to the fact that the effort may still increase exponentially with the problem size in the worst case. Besides the problem structure, the actual computation time depends on several additional factors such as the effectiveness of the applied bounding scheme and the (optional) incorporation of still more advanced techniques. As a consequence, the price paid for an exact solution of an optimization problem is usually high, often too high to be affordable.

The question, if finding a provably optimal solution is really mandatory, inevitably arises, but can not be answered in general. In some situations it might be actually necessary to find such a solution, whereas in many others a near-optimal or even very good solution is sufficient. Hence we have to find a compromise between the very simple and fast intuitive heuristic approaches and computationally expensive exact ones. The methods we are looking for are still heuristics, because they do not guarantee a (provably) optimal solution, but they rely on more advanced strategies than the ones we have seen so far.

---

[6] Given now-dated standard PC hardware

Before we actually get into the description of such methods in the following chapter, we propose a first rough taxonomy of solution approaches, as outlined in Figure 2.12. Note that this taxonomy is intended to be as general as possible, hence we only captured concepts which are generalizable and not specific to the knapsack problem. We can see the main differentiation between exact and heuristic approaches as already indicated above. Under the exact methods we can already classify the Branch & Bound method and the complete enumeration approach. As far as the heuristic methods are concerned, we are not yet able to provide a full taxonomy, because they cover a much broader spectrum of methods than this chapter was intended to describe. In the following chapters we will go deeper into the area of heuristic approaches and successively complete the heuristic part of our taxonomy.

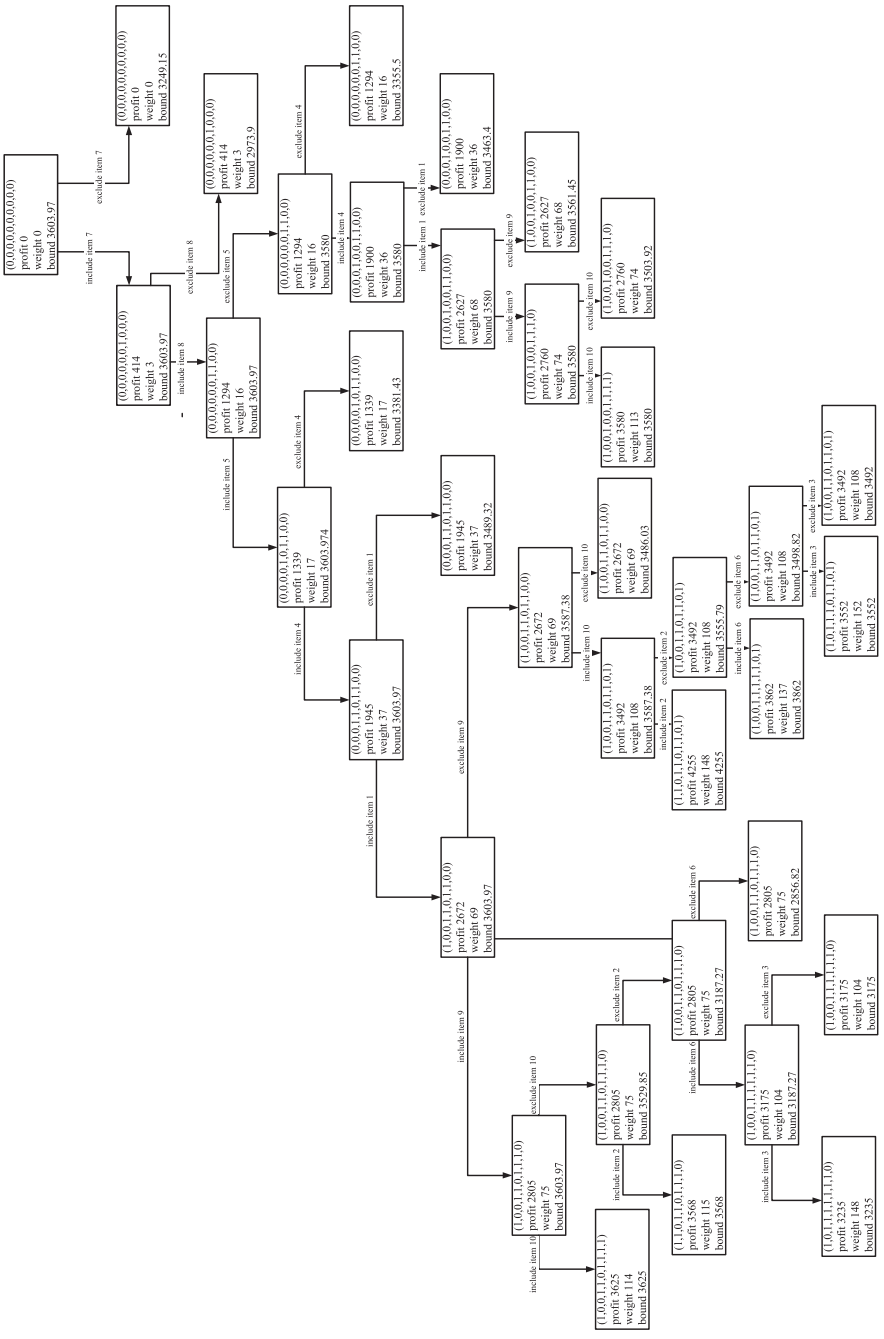
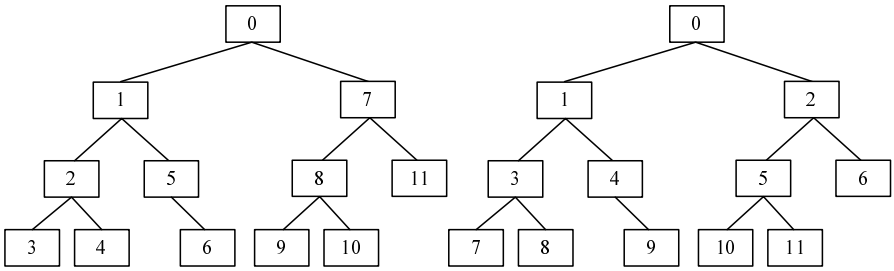


Fig. 2.10: Complete Branch&Bound tree



(a) Depth first search (b) Breadth first search

Fig. 2.11: Different strategies to traverse a tree (nodes are numbered according to their number of visit)

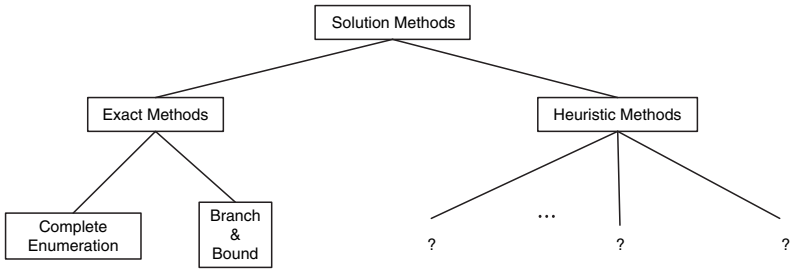


Fig. 2.12: A first rough taxonomy of solution methods for optimization problems

http://www.springer.com/978-3-642-11342-0

Metaheuristic Search Concepts
A Tutorial with Applications to Production and Logistics
Zäpfel, G.; Braune, R.; Bögl, M.
2010, X, 316p. 101 illus.., Hardcover
ISBN: 978-3-642-11342-0