7 Branch and Bound, and Dynamic Programming

7.1 Knapsack

An important combinatorial optimization problem is the Knapsack Problem, which can be defined as follows:

Given nonnegative integers $n, c_1, \ldots, c_n, w_1, \ldots, w_n$ and W. Find a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} c_i$ is maximal.

In the standard formulation c_i and w_i are the costs and weight of item i, respectively, and W is the capacity of the knapsack. The problem is to put items into the knapsack such that the total weight does not exceed the capacity, and the total costs are maximal. The above definition leads straightforwardly to the following ILP formulation:

$$\begin{array}{rll} \text{Maximize} & \sum_{i=1}^n c_i x_i & \\ \text{subject to} & \sum_{i=1}^n w_i x_i \leq W & \\ & 0 \leq x_i \leq 1 & (i = 1, \ldots, n) \\ & x_i \text{ integral} & (i = 1, \ldots, n) \end{array}$$

Here x_i is a decision variable that gets value 1 if item i is in the knapsack, and 0 otherwise. This ILP formulation of the knapsack problem has the advantage that it is very easy to solve its LP-relaxation.

Proposition 1 *Let $c_1, \ldots, c_n, w_1, \ldots, w_n$ and W be nonnegative integers with*

$$\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n},$$

and let

$$k := \min \left\{ j \in \{1, \ldots, n\} : \sum_{i=1}^j w_i > W \right\}.$$

Then an optimum solution of the LP-relaxation of the ILP model of the knapsack problem is defined by:

$$x_i = \begin{cases} 1 & i = 1, \ldots, k-1, \\ \frac{W - \sum_{i=1}^{k-1} w_j}{w_k} & i = k, \\ 0 & i = k+1, \ldots, n. \end{cases}$$

The value of this LP-relaxation is clearly an upper bound for the optimal value of the knapsack problem. Notice that the solution of the LP-relaxation has at most one variable with a fractional value. If all weights (w_i) are smaller than W, then the two solutions $\{1, \ldots, k-1\}$ and $\{k\}$ are both feasible the better of these two solutions achieves at least half the optimum value. To be more precise:

Proposition 2 *Let for an instance I of the knapsack problem $OPT(I)$ be its optimal value and $LP(I)$ be the value of the LP-relaxation. If all weights are at most W then*

$$\frac{1}{2} LP(I) < OPT(I) \leq LP(I).$$

7.2 Branch and Bound

The branch and bound method is based on the idea of intelligently enumerating all the feasible points of a combinatorial optimization problem. In the branch and bound method we search for an optimal solution based on successive partitioning of the solution space. The *branch* in branch and bound refers to this partitioning process; the *bound* refers to lower bounds that are used to eliminate parts of the solution space that will not contain an optimum. First we shall develop the method for ILP, and then put things in a more abstract framework. Consider the ILP problem

$$\begin{array}{lllcl} \text{Problem 0:} & \text{minimize} & z & = & \mathbf{c}^\top \mathbf{x} \;=\; c(\mathbf{x}) \\ & \text{subject to} & A\mathbf{x} & \leq & \mathbf{b} \\ & & \mathbf{x} & \geq & \mathbf{0},\ \mathbf{x} \in \mathbb{Z}\ . \end{array}$$

If we solve the LP relaxation, we obtain a solution $\mathbf{x}^0$, which in general is not integer. The cost $c(\mathbf{x}^0)$ of this solution is, however, a lower bound on the optimal cost $c(\mathbf{x}^*)$ (where $\mathbf{x}^*$ is the optimal solution to Problem 0), and if x^0 were integer, we would in fact be done. In the cutting plane algorithm, we would now add a constraint to the relaxed problem that does not exclude feasible solutions. Here, however, we are going to split the problem into two subproblems by adding two mutually exclusive and exhaustive constraints. Suppose that component x_i^0 of $\mathbf{x}^0$ is noninteger, for example. Then the two subproblems are

$$\begin{array}{lllcl} \text{Problem 1:} & \text{minimize} & z & = & \mathbf{c}^\top \mathbf{x} \;=\; c(\mathbf{x}) \\ & \text{subject to} & A\mathbf{x} & \leq & \mathbf{b} \\ & & \mathbf{x} & \geq & \mathbf{0},\ \mathbf{x} \in \mathbb{Z} \\ & & x_i & \leq & \lfloor x_i^0 \rfloor \end{array}$$

and

$$\begin{array}{lllcl} \text{Problem 2:} & \text{minimize} & z & = & \mathbf{c}^\top \mathbf{x} \;=\; c(\mathbf{x}) \\ & \text{subject to} & A\mathbf{x} & \leq & \mathbf{b} \\ & & \mathbf{x} & \geq & \mathbf{0},\ \mathbf{x} \in \mathbb{Z} \\ & & x_i & \geq & \lfloor x_i^0 \rfloor + 1 \end{array}$$

The solution to the original problem must lie in the feasible region of one of these two problems, simply because one of the following two statements must be true.

$$x_i^* \leq \lfloor x_i^0 \rfloor\ , \quad x_i^* \geq \lfloor x_i^0 \rfloor + 1\ .$$

We now choose one of the subproblems, say Problem 1, which is after all an LP, and solve it. The solution $\mathbf{x}^1$ will in general not be integer, and we may split Problem 1 into two subproblems just as we split Problem 0, creating Problems 3 and 4. We can visualize this process as a successive finer and finer subdivision of the feasible region. Each subset in a given partition represents a subproblem i, with relaxed solution $\mathbf{x}^i$ and lower bound $z_i = c(\mathbf{x}^i)$ on the cost of any solution in the subset. We can also visualize this process as a tree, as is shown in Figure 1. The root represents the original feasible region and each node represents a subproblem. Splitting the feasible region at a node by the addition of the inequalities is represented by the branching to the node's two children. If the original ILP has a bounded feasible region, this process cannot continue indefinitely, because eventually the inequalities at a node in the branching tree will lead to an integer solution to the corresponding LP, which is an optimal solution to the original ILP. The branching process can fail at a particular node for one of two reasons: (1) the LP solution can be integer; or (2) the LP problem can be infeasible.

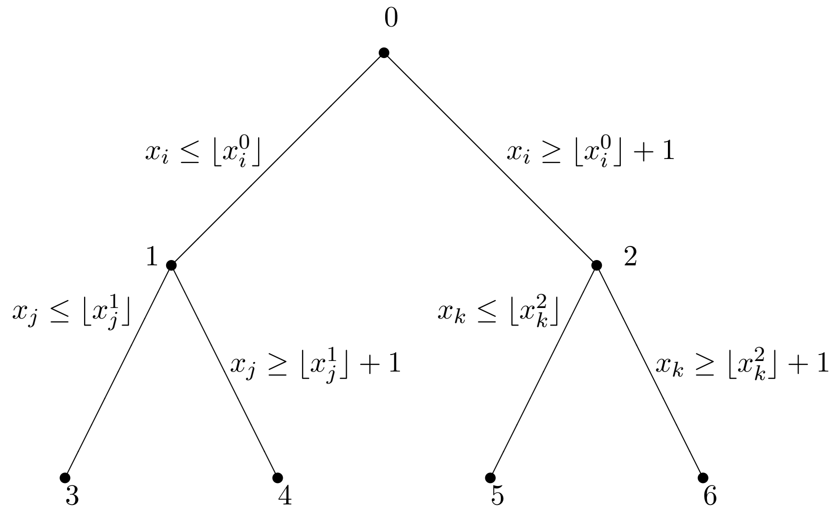

Figure 1: Representation of solution space subdivision by a binary tree

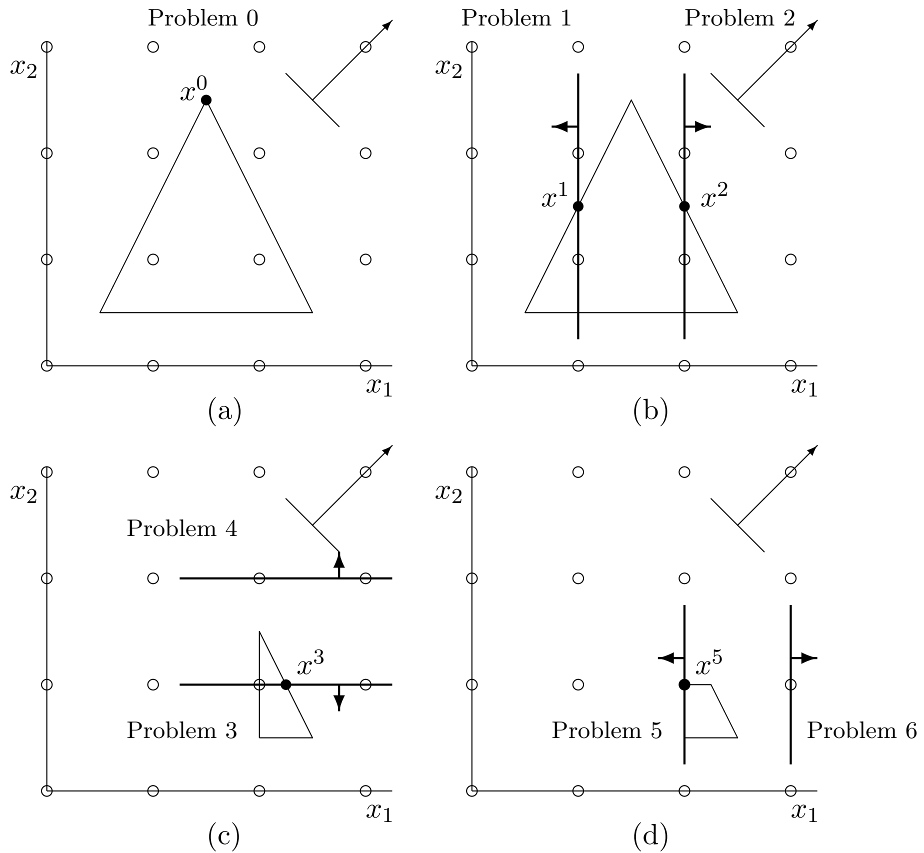

Figure 2: Stages in the solution of an ILP by branch and bound

A simple ILP is shown in Figure 2(a); the solution is $\mathbf{x}^* = (2,1)$ and $c(\mathbf{x}^*) = -(x_1+x_2) = -3$. The initial relaxed problem has the solution $\mathbf{x}^0 = (\frac{3}{2}, \frac{5}{2})$ with cost $c(\mathbf{x}^0) = -4$. Figure 2(b) shows the two subproblems generated by choosing the noninteger component $x_1^0 = \frac{3}{2}$ and introducing the constraints $x_1 \leq 1$ and $x_1 \geq 2$. If we continue branching from Problem 2 in Figure 2, we obtain the branching tree shown in Figure 3. Three leaves are reached in the right subtree; these leaves correspond to two feasible LP's and one LP with an integer solution $\mathbf{x}^5 = (2,1)$ with cost $z_5 = c(\mathbf{x}^5) = -3$. What we have

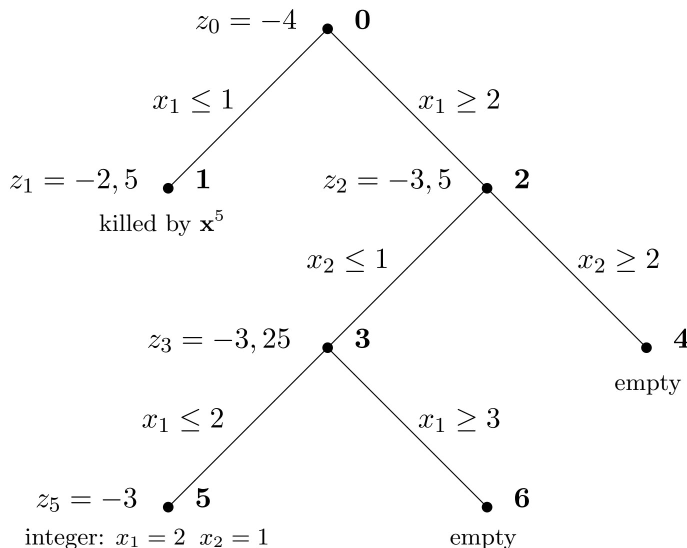

Figure 3: The binary tree leading to a solution to the problem

described up to this point comprises the branching part of the branch and bound. If we continue the branching process until all the nodes are leaves of the tree and correspond either to integer solutions or infeasible LP's, then the leaf with the smallest cost must be the optimal solution to the original ILP. We come now to an important component of the branch and bound approach: Suppose at some point the best complete integer solution obtained so far has cost z_m and that we are contemplating branching from a node at which the lower bound $z_k = c(\mathbf{x}^k)$ is greater than or equal to z_m. This means that any solution x that would be obtained as a descendent of x^k would have cost

$$c(\mathbf{x}) \geq z_k \geq z_m$$

and hence we need not proceed with a branching from $\mathbf{x}^k$. In such a case, we say that the node $\mathbf{x}^k$ has been *killed* or *fathomed*, and refer to it as *dead*. The remaining nodes, from which branching is still possibly fruitful, are referred to as *alive*. For our example (Figures 2, 3), the node corresponding to Problem 1 has associated with it a lower bound of $-2\frac{1}{2}$, which is greater than the solution cost of -3 associated with node 5. It is therefore killed by node 5, as shown. Since no live nodes remain, node 5 must represent the optimal solution.

There are now still two important details in the algorithm that need to be specified: We must decide how to choose, at each branching step, which node to branch from; and we must decide how to choose which noninteger variable is to determine the added constraint. The first choice is referred to as the *search strategy*. The two most common strategies are *depth first search* and *frontier search*. In case of depth first search we first consider the

last generated nodes to branch from. It has to be specified in which order these last nodes are considered. This strategy has the advantage that the amount of storage needed is limited and furthermore that in an early stage feasible solutions are found. In case of frontier search we first consider the node with the lowest lower bound because this is the most promising one. This strategy looks more effective, but for larger problems the number of subproblems that have to be stored is too large. For the second choice-the variable to add the constraint- the best strategy is to find that constraint which leads to the largest increase in the lower bound after that constraint is added, and to add either that constraint or its alternative. The motivation is to find the branch from a given node that is most likely to get killed, and in this way keep the search tree shallow. There are no theorems to tell us the best strategy, and computational experience and intuition are the only guides to the design of fast algorithms of this type.

If the feasible region is bounded, it is not difficult to see that the algorithm terminates. Indeed, then each variable can only be chosen a finite number of times for splitting.

7.3 Branch and Bound in a General Context

The idea of branch and bound is applicable not only to a problem formulated as an ILP (or mixed ILP), but to almost any problem of a combinatorial nature. We next describe the method in a very general context. Two things are needed to develop the tree in the branch and bound algorithm for ILP:

1. *Branching* A set of solutions, which is represented by a node, can be partitioned into mutually exclusive sets. Each subset in the partition is represented by a child of the original node.

2. *Lower Bounding* An algorithm is available for calculating a lower bound on the cost of any solution in a given subset.

No other properties of ILP were used. We may therefore formulate the method for any optimization problem in which (1) and (2) are available. Specification of the branching, the lower bounding and the search strategy leads to a branch and bound algorithm for this optimization problem. There is always some set of subproblems, or active nodes. This set is initialized by the original problem and the algorithm stops as soon as this set is empty. In each step of the algorithm one subproblem is selected. The choice is specified by the search strategy. There might be three reasons to eliminate this subproblem.

1. The subproblem is infeasible.

2. The lower bound is larger than or equal to the upper bound (cost of the best solution so far).

3. The subproblem can be solved. If the cost are less than the upper bound, the upper bound is replaced by these cost and the solution is stored.

If the subproblem can not be eliminated, it is split in a finite number of smaller subproblems that are added to the set of subproblems. There are now many choices in how we implement a branch and bound algorithm for a given problem; we shall discuss some of them.

- First, there is a choice of the branching itself; there may be many schemes for partitioning the solution space for that matter.

- Next, there is the lower bound calculation. One often has a choice here between bounds that are relatively tight but require relatively large computation time and bounds that are not so tight but can be computed fast.
- Third, there is the choice at each branching step of which node to branch from. The usual alternatives are least lower bound next (frontier search), last in first out (depth first search) or first in first out.
- Still another choice must be made at the start of the algorithm. It is often practical to generate an initial solution by some heuristic construction. This gives us an initial upper bound and may be very useful for killing nodes early in the algorithm. As usual, however, we must trade off the time required for the heuristic against possible benefit.

It should be clear from now that the branch and bound idea is not one specific algorithm, but rather a very wide class. Its effective use is dependent on the design of a strategy for the particular problem at hand. In this course we discuss two different branch and bound algorithms one for ILP (which we just did), and one for the traveling salesman problem (see Section 10).

7.4 Knapsack with Branch and Bound

The branch and bound algorithm for ILP has been illustrated in case of two variables, because then the LP-relaxations can easily be solved by hand. Another case where the LP-relaxation are easy to solve if for the knapsack problem. We illustrate this for the following example.

$$\begin{array}{ll} \text{Maximize} & 60x_1 + 60x_2 + 40x_3 + 10x_4 + 16x_5 + 9x_6 + 3x_7, \\ \text{subject to} & 3x_1 + 5x_2 + 4x_3 + x_4 + 4x_5 + 3x_6 + x_7 \leq 10, \\ & x_i \in \{0,1\}\,,\ i = 1, \ldots, 7. \end{array}$$

This problem can be solved using the branch and bound algorithm with the following specifications:

(A) *Branching.* Partition a subset F_i of the feasible region into two disjunct subsets by adding an extra condition: $x_j = 0$ and $x_j = 1$ respectively. The choice of the variable x_j follows from the calculation of the upper bound for the subset F_i. This corresponds to solving the LP-relaxation of some smaller knapsack problem. The solution of this relaxation has at most one variable with a fractional value. This variable is chosen as splitting variable.

(B) *Upper Bound calculation.* A subproblem F_i of the knapsack problem consists of the original knapsack where some variables are forced to be 0 or 1. This is again the feasible region of a knapsack problem since if some $x_i = 0$ this item can be removed and if $x_i = 1$, the knapsack capacity can be reduced by w_i. The upper bound calculation for F_i is again solving the LP-relaxation of a knapsack problem.

(C) *Search Strategy.* As search strategy we take the 'Last-in-First-out'-rule. Of the last two generated subsets we first consider the one with '$x_j = 1$'.

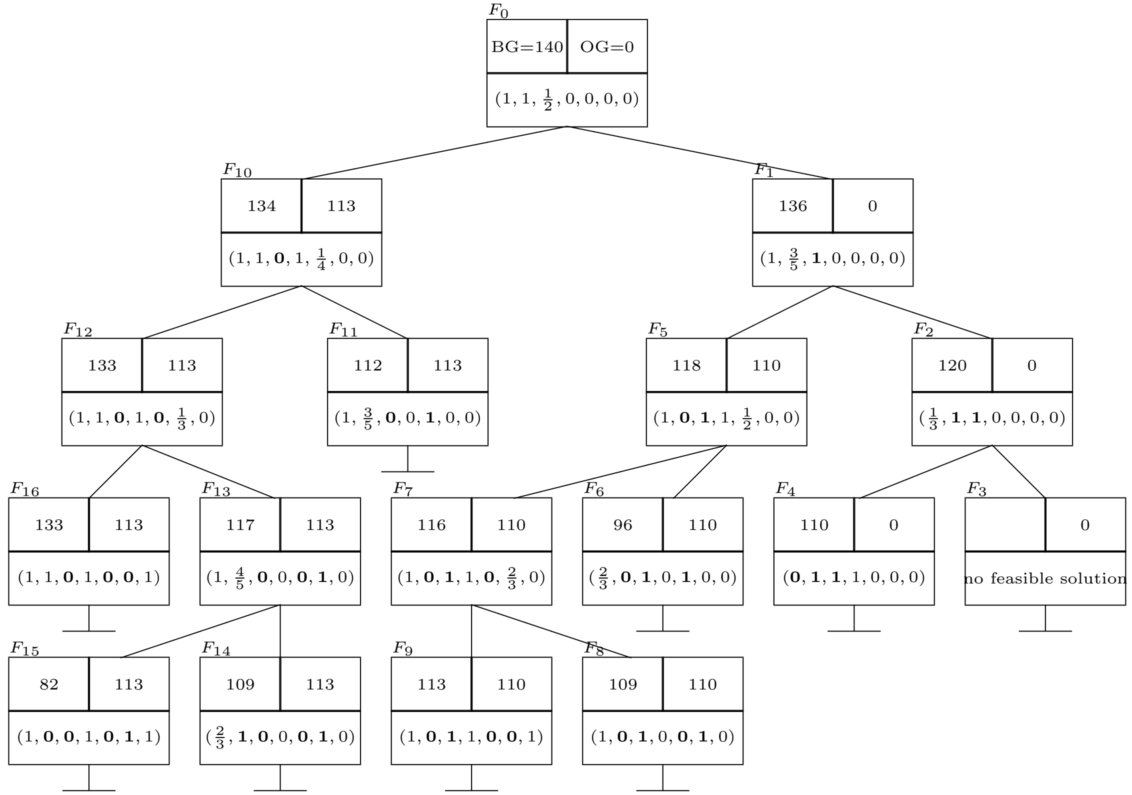

Figure 4: The Branch and Bound tree of the knapsack problem under consideration

Using these rules we get the following results. The items are already sorted by nonincreasing ratio $\frac{c_i}{w_i}$. The boldface 0's and 1's denote the variables that are fixed to 0 or 1 during the branching. The corresponding branch and bound tree is shown in Figure 4.

Name	Solution LP-relaxation	U.B.	L.B.	Next Action
F_0	$(1, 1, \frac{1}{2}, 0, 0, 0, 0)$	140	0	branch w.r.t. x_3
F_1	$(1, \frac{3}{5}, \mathbf{1}, 0, 0, 0, 0)$	136	0	branch w.r.t. x_2
F_2	$(\frac{1}{3}, \mathbf{1}, \mathbf{1}, 0, 0, 0, 0)$	120	0	branch w.r.t. x_1
F_3	no feasible solution		0	F_3 empty, backtrack
F_4	$(\mathbf{0}, \mathbf{1}, \mathbf{1}, 1, 0, 0, 0)$	110	0	integral solution, backtrack
F_5	$(1, \mathbf{0}, \mathbf{1}, 1, \frac{1}{2}, 0, 0)$	118	110	branch w.r.t. x_5
F_6	$(\frac{2}{3}, \mathbf{0}, \mathbf{1}, 0, \mathbf{1}, 0, 0)$	96	110	u.b.<l.b., backtrack
F_7	$(1, \mathbf{0}, \mathbf{1}, 1, \mathbf{0}, \frac{2}{3}, 0)$	116	110	branch w.r.t. x_6
F_8	$(1, \mathbf{0}, \mathbf{1}, 0, \mathbf{0}, \mathbf{1}, 0)$	109	110	integral solution and u.b.<l.b., backtrack
F_9	$(1, \mathbf{0}, \mathbf{1}, 1, \mathbf{0}, \mathbf{0}, 1)$	113	110	integral solution, backtrack
F_{10}	$(1, 1, \mathbf{0}, 1, \frac{1}{4}, 0, 0)$	134	113	branch w.r.t. x_5
F_{11}	$(1, \frac{3}{5}, \mathbf{0}, 0, \mathbf{1}, 0, 0)$	112	113	u.b.<l.b., backtrack
F_{12}	$(1, 1, \mathbf{0}, 1, \mathbf{0}, \frac{1}{3}, 0)$	133	113	branch w.r.t. x_6
F_{13}	$(1, \frac{4}{5}, \mathbf{0}, 0, \mathbf{0}, \mathbf{1}, 0)$	117	113	branch w.r.t. x_2
F_{14}	$(\frac{2}{3}, \mathbf{1}, \mathbf{0}, 0, \mathbf{0}, \mathbf{1}, 0)$	109	113	u.b.<l.b., backtrack
F_{15}	$(1, \mathbf{0}, \mathbf{0}, 1, \mathbf{0}, \mathbf{1}, 1)$	82	113	integral solution and u.b.<l.b., backtrack
F_{16}	$(1, 1, \mathbf{0}, 1, \mathbf{0}, \mathbf{0}, 1)$	133	113	integral solution, backtrack, finished

The optimal solution has value 133 and is given by:

$$x_1 = 1,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1,\ x_5 = 0,\ x_6 = 0,\ x_7 = 1$$

The number of considered subproblems can be 2^n and solving the LP-relaxation of a subproblem takes $\mathcal{O}(n)$. So the complexity of the branch and bound algorithm for knapsack is $\mathcal{O}(n2^n)$.

7.5 Dynamic Programming

Dynamic programming (DP) is, like branch and bound, an enumerative method to solve combinatorial optimization problems. The method can be characterized by the fact that it uses some recurrence relation to solve the problem. This recurrence relation relates the optimal solution of some problem to optimal solutions of smaller problems of the same type. For these smaller problems the same recurrence relation holds etcetera. Like the tree for the branch and bound method, one can define a directed graph to represent the dynamic programming method. The vertices of the graph correspond to the subproblems and there is an arc from subproblem Π_1 to Π_2 if Π_1 appears in the recurrence relation for Π_2. The original problem is then the 'endpoint' of the graph and some trivial subproblem(s) is(are) the starting point(s) of the graph. We will call this graph the DP-graph. The optimal solution of some subproblem can be determined using the recurrence relation if the optimal solutions of its predecessors are known. Since the graph is acyclic, one can solve all subproblems one-by-one starting with the trivial problems and ending with the original problem. In some cases solving a problem reduces to finding the shortest or longest path from the starting point to the end point in the DP-graph. This is for instance the case if the optimum $OPT(\Pi)$ of some subproblem Π is equal to the maximum or minimum of $\{OPT(\Pi_1) + c(\Pi_1, \Pi), \ldots, OPT(\Pi_k) + c(\Pi_k, \Pi)\}$ where $\Pi_1, \ldots, \Pi_k$ are the predecessors of problem Π and $c(.,.)$ are costs that can be computed easily for some problem instance. De complexity of some dynamic programming algorithm depends on the number of subproblems that have to be considered and the amount of work per subproblem. Often the total number of arcs in the DP-graph is a good measure.

In this course we have seen examples of dynamic programming algorithms already: The algorithms of Bellman-Ford and Floyd.

We will now discuss a dynamic programming algorithm for the knapsack problem. Let $1 \leq m \leq n$ en $0 \leq W' \leq W$. Define $f_m(W')$ as the optimal solution of the knapsack problem with items $1, 2, \ldots, m$ en knapsack capacity W'. Then clearly

$$\begin{aligned} f_0(W') &= 0 \quad W' = 0, 1, \ldots, W \\ f_1(W') &= \begin{cases} 0 & W' = 0, 1, \ldots, w_1 - 1 \\ c_1 & W' = w_1, w_1 + 1, \ldots, W \end{cases} \end{aligned}$$

We have the following recurrence relation for $f_m(W')$:

$$f_m(W') = \begin{cases} f_{m-1}(W') & W' = 0, 1, \ldots, w_m - 1 \\ \max\{f_{m-1}(W'), f_{m-1}(W' - w_m) + c_m\} & W' = w_m, w_m + 1, \ldots W \end{cases}$$

$f_n(W)$ is the optimum solution of the original problem. Dynamic programming now consists of calculating the values $f_m(W')$. This is done from $m = 0$ to $m = n$ and for some m from $W' = W$ downto $W' = 0$. This leads to the following algorithm:

At the start of the mth step of the algorithm the variables have the following values:

$$P[W'] = f_{m-1}(W') \quad \text{for} \quad W' = 0, 1, \ldots, W$$

$$X[W'] = S \subseteq \{1, 2, \ldots, m-1\} \quad \text{for} \quad W' = 0, 1, \ldots, W$$

where S is the subset of the set of items $\{1, 2, \ldots, m-1\}$ that leads to the maximum value $P[W']$ for capacity W'.

Input: $n, W, (c_i), (w_i)$;
Output: z;
begin
  for $W' = 0$ **to** W **do**
  begin
    $P[W'] := 0$;
    $X[W'] := \emptyset$;
  end
  for $m := 1$ **to** n **do**
    for $W' = W$ **downto** w_m **do**
      if $P[W'] < P[W' - w_m] + c_m$ **then**
      begin
        $P[W'] := P[W' - w_m] + c_m$;
        $X[W'] := X[W' - w_m] \cup \{m\}$
      end;
$z := P[W]$;
end.

This algorithm has time complexity $\mathcal{O}(nW)$ since the number of considered subproblems is nW and the amount of work per subproblem does not grow when the problem becomes larger. One only has to choose the best of two alternatives. Notice that the size of the input is $\mathcal{O}(n \log W + n \log C)$, where C is equal to the largest profit of the items. For our example we find the following values for $f_m(W')$:

W' ↑									
	10	0	60	120	120	130	130	130	133
	9	0	60	120	120	130	130	130	130
	8	0	60	120	120	120	120	120	120
	7	0	60	60	100	100	100	100	100
	6	0	60	60	60	70	70	70	73
W'	5	0	60	60	60	70	70	70	73
	4	0	60	60	60	70	70	70	70
	3	0	60	60	60	60	60	60	60
↑	2	0	0	0	0	10	10	10	13
	1	0	0	0	0	10	10	10	10
	0	0	0	0	0	0	0	0	0
		0	1	2	3	4	5	6	7
				→		m			

So, again the optimal solution has value 133 and is realized by choosing the items 1, 2, 4 and 7.

Exercises

1. Solve the following ILP problem by use of Branch and Bound with frontier search and draw the search tree. Minimize $11x + 10y$ subject to

$$
\begin{aligned}
13x + 10y &\geq 65 \\
8x + 17y &\geq 68 \\
x \geq 0,\ &y \geq 0 \\
x, y &\text{ integer.}
\end{aligned}
$$

2. Give an ILP formulation of the knapsack problem given below with knapsack capacity 17. Solve the ILP problem with Branch and Bound with frontier search. Draw the search tree.

object	1	2	3	4	5	6
value	8	16	20	6	4	9
weight	3	7	9	3	2	5

3. Solve the following knapsack problem with knapsack capacity 7, by use of dynamic programming.

object	1	2	3	4	5
value	4	7	3	5	4
weight	5	3	2	2	1

4. Show that if for an instance I of the knapsack problem not all weights are at most the capacity W, then $LP(I)/OPT(I)$ can be arbitrarily large.

5. Consider a knapsack problem with capacity W, items $1, \ldots, n$, weights $w_1, \ldots, w_n$, and values (costs) $c_1, \ldots, c_n$. Define $C = \sum_{j=1}^{n} c_j$. For an integer $m \leq n$, define $g_m(C')$ as the minimal total weight of a subset $S \subset \{1, 2, \ldots, m\}$ for which $\sum_{i \in S} c_i = C'$ (take $g_m(C') = \infty$ if such a set doesn't exist).

   a. What is $g_0(C')$ for $0 \leq C' \leq C$?

   b. How can $OPT(I)$ be derived from the values of $g_n(C')$ $(0 \leq C' \leq C)$?

   c. Derive a recursion formula for $g_{m+1}(C')$ in terms of the $g_m(C')$ and $g_m(C' - c_{m+1})$ (provided $C' \geq c_{m+1}$).

   d. Show that this gives a dynamic programming algorithm with complexity $\mathcal{O}(nC)$.

   e. Use this algorithm to solve the knapsack problem of Exercise 3.