MSM3M02a: *Autumn* 2010

Integer Programming

Lecture 9:

Solving Knapsack Problems —— By Dynamic Programming

Some Motivation: Shortest paths
Given a directed graph with nonnegative arc distances, and initial node s, the problem is to find the shortest path from s to every other node v.

Observation: If the shortest path from s to t passes by node p, the sub-paths (s, p) and (p, t) are shortest paths from s to p, and p to t respectively.

If this were not true, the shorter sub-path would allow us to construct a shorter path from s to t, leading to a contradiction.

Observation: Let $d(v)$ denote the length of a shortest path from s to v. Then

$$d(v) = \min_{i \in V^{-1}(v)} \{d(i) + c_{iv}\}$$

where c_{iv} is the distance between node i and v, and $V^-(i)$ means the set of predecessors of node i.

In other words, if we know the lengths of the shortest paths from s to every neighbor (predecessor) of v, then we can find the length of the shortest path from s to v.

We will discuss an approach by which some IP problems can be solved by recursively reducing the problem dimension. This is called ***dynamic programming*** **(DP for short)**.

In this lecture, we will see that ***knapsack problems*** with integer coefficients are gen-

erally well-solved via dynamic programming.

DP is an effective approach for Knapsack problems if the size of the data is restricted. Consider the integer knapsack problem

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ s.t. \quad & \sum_{j=1}^{n} a_j x_j \leq b \\ & x \in Z_+^n \end{aligned}$$

where the data $a_j (j = 1, ..., n)$ and b are integral.

First, let's see how the 0-1 knapsack problem can be solved via DP, and then in the next lecture how the general case can be solved by DP.

0-1 Knapsack Problems

Let $\{a_j\}_{j=1}^n$ and $b \in N$, i.e., nonnegative integers. Consider the $0-1$ knapsack problem

$$\begin{aligned} (P) \qquad \max \quad & \sum_{j=1}^{n} c_j x_j \\ s.t. \quad & \sum_{j=1}^{n} a_j x_j \leq b, \\ & x \in B^n. \end{aligned}$$

DP for 0-1 knapsack problem:

- We need to set up a recursion that relates the optimal solution of (P) to those of certain subproblems.
- For this purpose we vary the right-hand side over the range $\lambda = 0, ..., b$, and we vary the number of variables from $r = 1, ..., n$.

For these values we consider the problems

$$(P_r(\lambda)) \qquad f_r(\lambda) = \max \sum_{j=1}^{r} c_j x_j$$

$$s.t. \quad \sum_{j=1}^{r} a_j x_j \leq \lambda,$$
$$x \in B^r.$$

Then $z = f_n(b)$ gives us the optimal value of the knapsack problem, and furthermore, all the problems $(P_r(\lambda))$ are all knapsack problems of the same type but of smaller size.

To arrive at a recursive relationship between the values $f_r(\lambda)$, we distinguish two cases:

- If $x_r^* = 0$, then the choice of the remaining variables $x_1, ..., x_{r-1}$ must be optimal for the problem $(P_{r-1}(\lambda))$. In other words, we then have

$$f_r(\lambda) = f_{r-1}(\lambda).$$

- If $x_r^* = 1$, then the choice of the remaining variables $x_1, ..., x_{r-1}$ must be optimal for $(P_{r-1}(\lambda - a_r))$, and hence,

$$f_r(\lambda) = c_r + f_{r-1}(\lambda - a_r).$$

Remark: Since we are looking for the maximum objective value, we just compare the two function values above and pick the one that produces the larger value! That is the value for $f_r(\lambda)$.

Thus, we arrive at the recursion

$$f_r(\lambda) = \max\left\{f_{r-1}(\lambda), c_r + f_{r-1}(\lambda - a_r)\right\}.$$

To initialize the recursion, we set the obvious boundary values

$$f_r(0) = 0, \quad r = 1, ..., n,$$

$$f_0(\lambda) = 0, \quad \lambda = 0, ..., b.$$

And for any other r and λ, we have the following recursions:

$$f_r(\lambda) = \max\{f_{r-1}(\lambda), c_r + f_{r-1}(\lambda - a_r)\}.$$

This is based on the following observations:

- When $x_r^* = 0$ (for instance when $0 \leq \lambda < a_r$), we have

$$f_r(\lambda) = f_{r-1}(\lambda),$$

- When $(x_r^* = 1)$, we have

$$f_r(\lambda) = c_r + f_{r-1}(\lambda - a_r).$$

We use the recursion to successively calculate $f_1, f_2, f_3, ..., f_n$ for all integral values of $\lambda = 0, ..., b$.

Once we have all these values, the question is how to find an associated optimal solution. This could be done by iterating back from the optimal value $f_n(b)$:

- If $f_n(b) = f_{n-1}(b)$, we set $x_n^* = 0$ and continue by looking for an optional solution of value $f_{n-1}(b)$.
- If $f_n(b) = c_n + f_{n-1}(b - a_n)$, we set $x_n^* = 1$ and then continue by looking for an optimal solution of value $f_{n-1}(b - a_n)$.

The computation of the backtracking procedure can be avoided altogether by keeping track of the indicator variables

$$p_r(\lambda) = \begin{cases} 0 & \text{if } f_r(\lambda) = f_{r-1}(\lambda), \\ 1 & \text{if } f_r(\lambda) = c_n + f_{r-1}(\lambda - a_n) \end{cases}$$

Thus, the above procedure becomes

- If $p_n(b) = 0$, we set $x_n^* = 0$ and continue by checking value $p_{n-1}(b)$.
- If $p_n(b) = 1$, we set $x_n^* = 1$ and then continue by checking the value $p_{n-1}(b - a_n)$.

Therefore, the DP algorithm for 0-1 knapsack problem can be stated as follows:

Algorithm 9.1. [DP for 0-1 Knapsack with integer coefficients (maxization)]

1. Initialisation: Set $f_r(0) = 0, (r = 1, ..., n), f_0(\lambda) = 0, (\lambda = 0, ..., b)$.

2. Forward Recursion: For $r = 1, 2, ..., n$, repeat

   i) $f_r(\lambda) = f_{r-1}(\lambda),$ for $\lambda = 0, ..., a_r - 1,$

   ii) $f_r(\lambda) = \max\{f_{r-1}(\lambda), c_r + f_{r-1}(\lambda - a_r)\},$ for $\lambda = a_r, ..., b.$

3. Backward Recursion:

   i) Set $\lambda = b,\ r = n,\ x = 0.$

   ii) While $r, \lambda > 0$, if $p_r(\lambda) = 1$ set $x_r = 1$ and $\lambda \longleftarrow \lambda - a_r,$

   end.

   $r \longleftarrow r - 1.$

   Repeat the step ii).

Example 9.2. Let us apply the above Algorithm to the $0 - 1$ knapsack problem

$$\begin{aligned} \max \quad & 10x_1 + 7x_2 + 25x_3 + 24x_4 \\ s.t. \quad & 2x_1 + x_2 + 6x_3 + 5x_4 \leq 7, \\ & x \in B^4. \end{aligned}$$

- Starting the forward recursion, $f_1(1) = f_0(1) = 0, p_1(1) = 0,$
- $f_1(2) = \max\{f_0(2), c_1 + f_0(2 - a_1)\} = \max\{0, 10 + 0\} = 10,\ p_1(2) = 1,$
- and likewise $f_1(3) = \cdots = f_1(7) = 10,\ p_1(3) = \cdots = p_1(7) = 1.$
- Next, $f_2(1) = \max\{f_1(1), c_2 + f_1(1 - a_2)\} = \max\{0, 7 + 0\} = 7, p_2(1) = 1,$
- $f_2(2) = \max\{f_1(2), c_2 + f_1(2 - a_2)\} = \max\{10, 7 + 0\} = 10,\ p_2(2) = 0,$
- $f_2(3) = \max\{f_1(3), c_2 + f_1(3 - a_2)\} = \max\{10, 7 + 10\} = 17,\ p_2(3) = 1,$
- and likewise, $f_2(4) = \cdots = f_2(7) = 17,\ \ p_2(4) = \cdots = p_2(7) = 1,$
- Continuing this way, we find the values

	f_1	f_2	f_3	f_4	p_1	p_2	p_3	p_4
$\lambda = 0$	0	0	0	0	0	0	0	0
1	0	7	7	7	0	1	0	0
2	10	10	10	10	1	0	0	0
3	10	17	17	17	1	1	0	0
4	10	17	17	17	1	1	0	0
5	10	17	17	24	1	1	0	1
6	10	17	25	31	1	1	1	1
7	10	17	32	34	1	1	1	1

Back-tracking is easy, using the $p_r(\lambda)$,

- $p_4(7) = 1$, thus $x_4^* = 1$.
- Therefore, we next have to check $p_3(7 - a_4) = p_3(2) = 0$, so $x_3^* = 0$,
- so we check $p_2(2) = 0$, showing that $x_2^* = 0$,
- and finally, $p_1(2) = 1$, so $x_1^* = 1$. Thus, $x^* = (1, 0, 0, 1)$ is an optimal solution.

Example 9.3. Solve the following 0-1 knapsack problem *by dynamic programming*

$$\begin{aligned} \max \quad & 7x_1 + 8x_2 + 3x_3 \\ s.t. \quad & x_1 + 2x_2 + x_3 \leq 3, \\ & x_1,\ x_2,\ x_3 \in \{0, 1\}. \end{aligned}$$

Lecture 10:

Solving Integer Knapsack Problems by DP

We now extend the DP method to integer knapsack problems

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ s.t. \quad & \sum_{j=1}^{n} a_j x_j \leq b \\ & x \in Z_+^n \end{aligned}$$

where again the coefficients $b, a_j (j = 1, ..., n)$ are positive integers.

We can again vary the right-hand side $\lambda = 0, ..., b$ and the number of variables $r = 0, ..., n$ and consider all of the following subproblems,

$$\begin{aligned} (P_r(\lambda)) \quad g_r(\lambda) := \quad \max \quad & \sum_{j=1}^{r} c_j x_j \\ s.t. \quad & \sum_{j=1}^{r} a_j x_j \leq \lambda \\ & x \in Z_+^r \end{aligned}$$

Clearly, our original knapsack problem is $(P_n(b))$, so that its optimal value is given by $g_n(b)$.

The following boundary (starting) values are obvious,

$$g_r(0) = 0, \quad (r = 0, ..., n),$$

$$g_0(\lambda) = 0, \quad (\lambda = 0, ..., b).$$

If x^* is an optimal solution to $P_r(\lambda)$ giving value $g_r(\lambda)$, then we consider the value of x_r^*, which can be any integer value of the following

$$0, ..., \lfloor \frac{\lambda}{a_r} \rfloor.$$

If $x_r^* = t$ then using the principle of optimality, we have that

$$g_r(\lambda) = c_r t + g_{r-1}(\lambda - t a_r)$$

for some

$$t = 0, 1, ..., \lfloor \frac{\lambda}{a_r} \rfloor.$$

We find the recursion formula

$$g_r(\lambda) = \max \left\{ t c_r + g_{r-1}(\lambda - t a_r) : \quad t = 0, ..., \lfloor \frac{\lambda}{a_r} \rfloor \right\}.$$

We may find a simpler formula than this.

Observe that

- If $x_r^* = 0$, then the vector $(x_1^*, ..., x_{r-1}^*)$ must be optimal for the problem $(P_{r-1}(\lambda))$, so that $g_r(\lambda) = g_{r-1}(\lambda)$.

- If $x_1^* \geq 1$, then the vector $(x_1^*, ..., x_{r-1}^*, x_r^* - 1)$ must be optimal for $(P_r(\lambda - a_r))$, so that $g_r(\lambda) = c_r + g_r(\lambda - a_r)$.

Therefore, we can arrive at the recursion

$$g_r(\lambda) = \max \left\{ g_{r-1}(\lambda), c_r + g_r(\lambda - a_r) \right\}.$$

Again we set the indicator variables

$$p_r(\lambda) = \begin{cases} 0, & \text{if } g_r(\lambda) = g_{r-1}(\lambda) \\ 1, & \text{otherwise.} \end{cases}$$

Algorithm 10.1. [DP for Integer Knapsack]

1. Initialisation: Set $g_r(0) = 0, (r = 1, ..., n), g_0(\lambda) = 0, (\lambda = 0, ..., b)$.

2. Forward recursion: For $r = 1, ..., n$, repeat

   i) $g_r(\lambda) = g_{r-1}(\lambda), \lambda = 0, ..., a_r - 1,$

   ii) $g_r(\lambda) = \max\{g_{r-1}(\lambda), c_r + g_r(\lambda - a_r)\},\ (\lambda = a_r, ..., b).$

3. Backward Recursion:

   i) Set $\lambda = b, r = n, x = 0.$

   ii) While $\lambda, r > 0$, repeat

   if $p_r(\lambda) = 0$, set $r \longleftarrow r - 1,$

   elseif $p_r(\lambda) = 1,$

   set $x_r \longleftarrow x_r + 1$, and $\lambda \longleftarrow \lambda - a_r,$

   end.

To back-track, we then start checking $p_n(b)$:

- If $p_n(b) = 1$ then $x_n^* \geq 1$ and we must check $p_n(b - a_n)$ to see whether $x_n^* \geq 2$ etc.

- If $p_n(b) = 0$ then $x_n^* = 0$ and we must check $p_{n-1}(b)$ to see whether $x_{n-1}^* \geq 1$ etc.

Example 10.2. Consider the integer knapsack problem

$$\begin{aligned} \max \quad & 7x_1 + 9x_2 + 2x_3 + 15x_4 \\ s.t. \quad & 3x_1 + 4x_2 + x_3 + 7x_4 \leq 10, \\ & x \in Z_+^4. \end{aligned}$$

Applying Algorithm 10.1, we find the following table of values for $g_r(\lambda), p_r(\lambda)$,

	g_1	g_2	g_3	g_4	p_1	p_2	p_3	p_4
$\lambda = 0$	0	0	0	0	0	0	0	0
1	0	0	2	2	0	0	1	0
2	0	0	4	4	0	0	1	0
3	7	7	7	7	1	0	0	0
4	7	9	9	9	1	1	0	0
5	7	9	11	11	1	1	1	0
6	14	14	14	14	1	0	0	0
7	14	16	18	18	1	1	1	0
8	14	18	18	18	1	1	0	0
9	21	21	21	21	1	0	0	0
10	21	23	23	23	1	1	0	0

Back-tracking:

- $p_4(10) = 0$, so $x_4^* = 0$.
- $p_3(10) = 0$, so $x_3^* = 0$.
- $p_2(10) = 1$, so $x_2^* \geq 1$.
- $p_2(10 - a_2) = p_2(6) = 0$, so $x_2^* \not\geq 2$, and hence, $x_2^* = 1$.
- $p_1(6) = 1$, so $x_1^* \geq 1$.
- $p_1(6 - a_1) = p_1(3) = 1$, so $x_1^* \geq 2$.
- $p_1(3 - a_1) = p_1(0) = 0$, so $x_1^* \not\geq 3$, and hence, $x_1^* = 2$.

We have found that $x^* = (2, 1, 0, 0)$ is an optimal solution.

Example 10.3. Solve the following integer programming problem *by dynamic programming*

$$\begin{aligned} \max \quad & 11x_1 + 7x_2 + 5x_3 \\ s.t. \quad & x_1 + 2x_2 + x_3 \leq 5, \\ & x_1,\ x_2,\ x_3 \geq 0 \text{ and integer.} \end{aligned}$$

MSM3M02a: *Autumn* 2010

Integer Programming

Lecture 11:

Branch and Bound (I)
—— Solving general integer programming problems

So far, we have learned

- how to solve the class of IP problems with total unimodularity,
- how to solve Knapsack problems by dynamic programming

For general IP problems we have learned

- how to find dual bounds (i.e., upper bounds for maximization problems) on integer programming problems by solving certain relaxations,
- how to solve some relaxations (i.e. "easy" optimization problems).

What we haven't learned yet is how to use these bounds to solve the general IP problems, which is the problem we are going to attack next!

A small-size problem is usually much easier to solve than a large-size (large-scaled) problem. Given a large problem, a natural approach solving the problem is to break it into a series of small problems.

Consider the problem

$$z = \max\{c^T x : \ x \in S\}.$$

Question:

How can we break the problem into a series of smaller problems that are easier, solve the smaller problems, and then put the information together again to solve the original problem?

Proposition 11.1. [Divide and Conquer] Consider the problem

$$z = \max\{c^T x : x \in S\}.$$

If the set S of feasible solutions can be decomposed into a union of simpler sets $S = S_1 \bigcup \cdots \bigcup S_k$ and if

$$z^j := \max\{c^T x : x \in S_j\} \quad (j = 1, ..., k),$$

then

$$z = \max_j z^j.$$

A typical way to represent such a divide and conquer approach is via an enumeration tree.

Example 11.2a. Let $S \subseteq B^3$. How we break the set into smaller set, and construct the enumera tion tree?

Example 11.2b. Let S be the set of feasible tours of the travelling salesman problem on a network of 4 cities. Let node 1 be the departure city.

- S can be subdivided into the disjoint sets of tours that start with an arc (12), (13) or (14) respectively, i.e.,

  $$S = S(12) \bigcup S(13) \bigcup S(14),$$

  where $S(1i)$ means the tour starting with arc $(1i)$.

- Each of the sets $S(12), S(13)$ and $S(14)$ can be further subdivided according to the choice of the second arc, $S(12) = S(12)(23) \bigcup S(12)(24)$ etc.

- Each of these sets corresponds to a specific TSP tour and cannot be further subdivided. We have found an enumeration tree of the TSP tours.

We see that S was first decomposed on a first level, and then each of the constituent parts was further decomposed on a further level and so on.

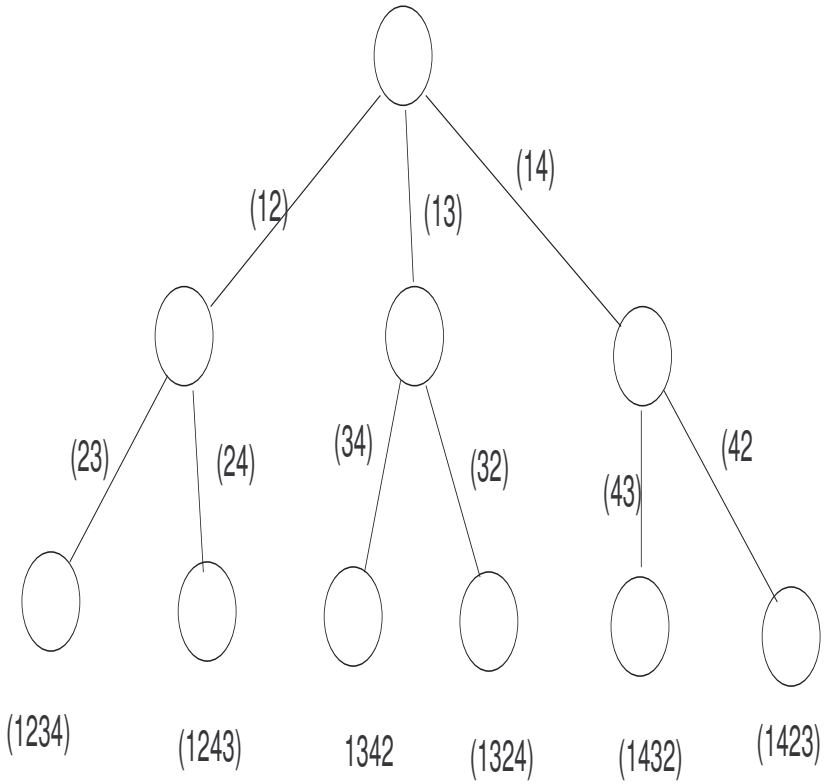

Figure 1: TSP enumeration tree

Thus, Proposition 11.1 allows us to decompose a hard problem into a possibly large number of easier branch problems, and to find an optimal solution of S by comparing the solutions found for the branch problems.

However, for even quite moderately sized problems such a tree can no longer be explicitly enumerated, as the number of leaves grows exponentially in the problem size.

The idea of implicit enumeration is based on building up the enumeration tree as we explore it, and to *prune* certain parts that are not worth looking at before those parts are even generated. The *pruning* mechanisms are based on the following insight:

Proposition 11.3. Consider the problem

$$z = \max\{c^T x : x \in S\},$$

and let

$$S = S_1 \bigcup ... \bigcup S_k$$

be a decomposition of its feasible domain into smaller sets. Let

$$\underline{z}^j \leq z^j \leq \bar{z}^j$$

be lower and upper bounds on

$$z^j = \max\{c^T x : x \in S_j\}$$

for all j. Then

$$\underline{z} := \max_j \underline{z}^j \leq z \leq \max_j \bar{z}^j =: \bar{z}$$

gives an upper and lower bound on z.

Pruning Mechanisms

We speak of pruning a branch when we detect that we need no longer explore it further. This can happen for a variety of reasons.

1. Pruning by Bound: A branch S_j can be pruned when $\bar{z}^j \leq \underline{z}$.

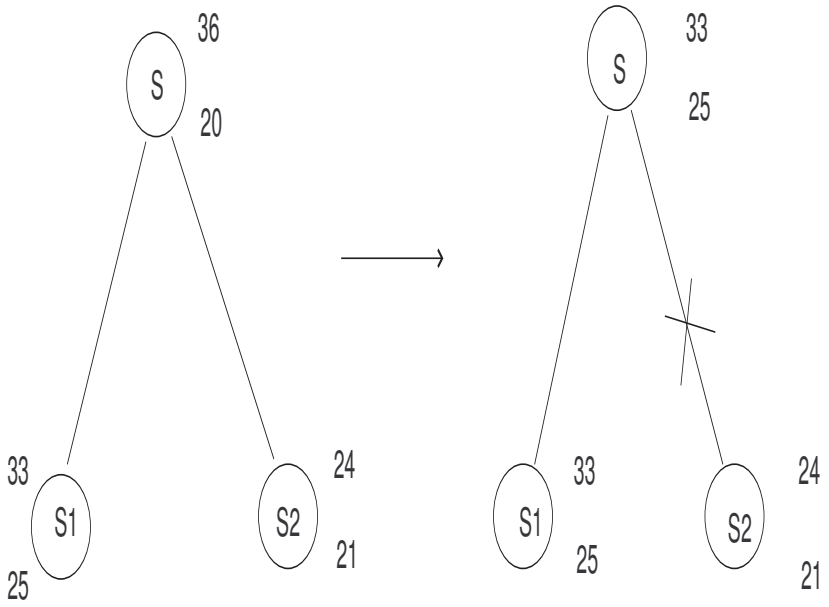

Figure 2: Pruning by bound

2. Pruning by Infeasibility: If $S_j = \emptyset$; then the corresponding branch can be pruned.

Why would anyone introduce an empty S_j into the decomposition of S?
Remember that we don't set up the decomposition tree explicitly but use an implicit enumeration, typically by introducing more and more extra constraints as we trickle down towards the leaves of the enumeration tree.

As we proceed, the constraints may become incompatible and correspond to an empty set S_j, but this may not be a-priori obvious and has to be detected.

3. Pruning by Optimality:

When $\underline{z}^j = \bar{z}^j$ for some j, then the branch corresponding to S_j no longer has to be considered further, as an optimal solution $z^j = \underline{z}^j = \bar{z}^j$ for this branch is already available. However, we will not throw this solution away, as it may later turn out to be optimal for the parent problem S.

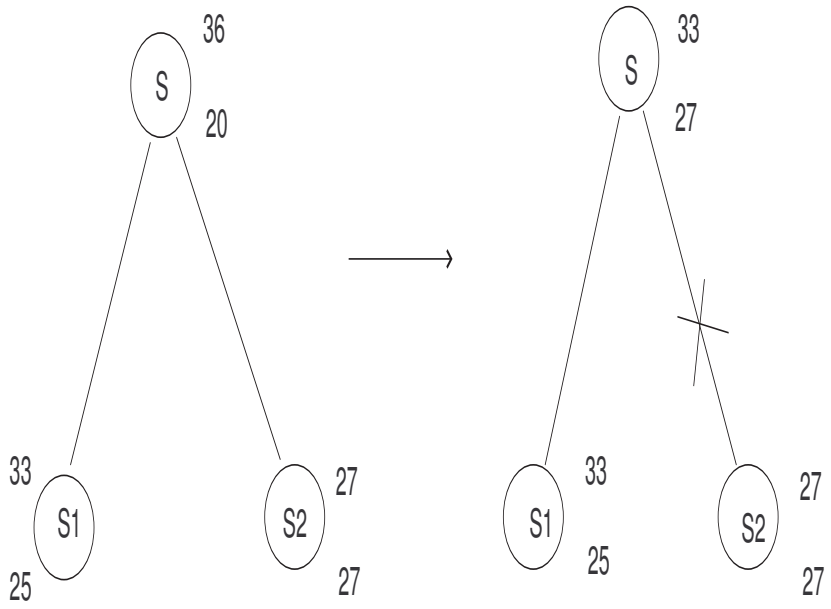

Figure 3: Pruning by Optimality

Branch and Bound Method

A branch and bound method systematically exploits all of the above to break down the solution of a hard optimization problem $\max\{c^T x : x \in S\}$ into easier parts:

- Implicitly build an enumeration tree for S by subdividing $S = S_1 \bigcup ... \bigcup S_k$ on the first level, and then subdividing each S_j further on the next level and so on.
- Only build the parts of the tree that are actually explored.
- For each subproblem S_j (corresponding to a branch of the enumeration tree), compute primal and dual bounds: use heuristics for primal bounds, relaxation for dual bounds.
- Use Proposition 11.3 to tighten bounds at the root.
- Prune branches that need not be explored.

If LP relaxation is used for generating dual bounds in a branch-and-bound system, we speak of LP based branch-and-bound. We will now explore this framework in more detail.

Algorithm 11.4. [LP based branch-and-bound] Throughout the algorithm we will maintain and update a list of active nodes List, a primal bound $\underline{z}$ on $\max\{c^T x : x \in S\}$, and an incumbent x^*, that is, the best solution encountered so far.

1. Initialisation: Set $List := \{S\}, \underline{z} := -\infty$ and $x^* := \emptyset$.
2. While $List \neq \emptyset$, repeat:

   i) Choose a problem $S_j \in List$ with formulation P_j, and solve the LP relaxation $x^j := \arg\max\{c^T x : x \in P_j\}$,
   if an optimal solution x^j exists, store $\bar{z}^j = c^T x$,
   elseif the LP is unbounded, store $\bar{z}^j = +\infty$,
   else store "$P_j = \emptyset$",
   end.

   ii) Prune or branch: if "$P_j = \emptyset$",

   $$List \longleftarrow List \setminus \{S_j\}, \quad \text{(prune by infeasibility)},$$

   elseif $\bar{z}^j \leq z$,

   $$List \longleftarrow List \setminus \{S_j\}, \quad \text{(prune by bound)},$$

elseif $x^{[j]}$ is feasible for S,

$$z \longleftarrow z^j, \quad \text{(update primal bound)},$$

$$x^* \longleftarrow x^j, \quad \text{(update incumbent)},$$

$$List \longleftarrow List \setminus \{S_j\}, \text{(prune by optimality)},$$

else branch S_j into two subproblems $S_j^{[1]}, S_j^{[2]}$,

$$List \longleftarrow (List \setminus \{S_j\}) \cup \{S_j^{[1]}, S_j^{[2]}\}.$$

end.

3. Stop with incumbent x^* optimal ($x^* = \emptyset$ is a certificate of infeasibility of the problem).

A variety of modification of the above algorithm can be made. For instance, the efficiency of the algorithm can be improved if in step 2.i) we also compute a primal bound for S_j and use this to update $\underline{z}$ as well as an upper bound $\bar{z}$.

Four Questions:

1. *How are the bounds are to be obtained?*

   This question has been answered already. The primal (lower) bounds are provided by feasible solutions, and dual (upper) bounds by relaxation or duality.

2. *How should the feasible region be separated into smaller regions*?

   One simple idea is to choose an integer variable that is fractional in the linear programming solution, and split the problem into two about this fractional value.

   If $x_j = \widetilde{x}_j \notin Z^1$, one can take

   $$S_1 = S \cap \{x : x_j \leq \lfloor \widetilde{x}_j \rfloor\}$$

   $$S_2 = S \cap \{x : x_j \geq \lceil \widetilde{x}_j \rceil\}$$

   It is clear that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$.

Example 11.5: If $x_j = 4/3$ is the solution of relaxation problem, and S is the current feasible region. Then it can be split up as $S = S_1 \cup S_2$ where

$$S_1 = S \cap \{x : x_j \leq \lfloor \frac{4}{3} \rfloor\} = S \cap \{x : x_j \leq 1\}$$

$$S_2 = S \cap \{x : x_j \geq \lceil \frac{4}{3} \rceil\} = S \cap \{x : x_j \geq 2\}$$

3. *If LP relaxation is used, How to solve the subproblems efficiently?*

   - We split up the feasible region by adding an inequality to the current LP problems.
   - Thus, we may using the sensitivity analysis method of LP to reoptimize the slightly changed LP problems without starting again from scratch.
   - As we have just added one single upper and lower bound constraints to the linear program, our previous optimal basis remains dual feasible, and it is therefore natural to reoptimize from this basis using the **dual simplex algorithm**.
   - Let x^*, y^* be optimal solutions for the primal and dual LP instances

     $$(P) \quad \max\{c^T x : Ax \leq b, x \geq 0\}$$

     and

     $$(D) \quad \min\{b^T y : A^T y \geq c, y \geq 0\}.$$

     Adding a new constraint, LP becomes

     $$(P') \quad \max\{c^T x : \quad Ax \leq b, \quad a_{m+1}^T x \leq b_{m+1}, \quad x \geq 0\}$$

     This corresponds to the situation where a new variable appears in the dual problem

     $$(D') \quad \min\{b^T y + b_{m+1} y_{m+1} : \quad A^T y + a_{m+1} y_{m+1} \geq c, \quad y, y_{m+1} \geq 0\}.$$

4. *In what order should the subproblems (nodes in enumerate tree) be examined?*

   There is no single answer that is best for all instances. We will discuss this question further.

Lecture 12:

Branch and Bound (II): An example

Example 12.1: Solve the following IP.

$$\begin{array}{lrll} (IP) & z = & \max & 4x_1 - x_2 \\ & & s.t. & 7x_1 - 2x_2 \leq 14 \\ & & & x_2 \leq 3 \\ & & & 2x_1 - 2x_2 \leq 3 \\ & & & x \in Z^2_+ \end{array}$$

1. ***Bounding:*** To obtain the first upper bound, solve the LP relaxation

$$\begin{array}{lrll} (IP) & z = & \max & 4x_1 - x_2 \\ & & s.t. & 7x_1 - 2x_2 \leq 14 \\ & & & x_2 \leq 3 \\ & & & 2x_1 - 2x_2 \leq 3 \\ & & & x \in R^2_+ \end{array}$$

   Let x_3, x_4, x_5 be slack variables. By simplex method, the resulting optimal basis representation is

$$\begin{array}{rlllllll} \bar{z} = \max \frac{59}{7} & & & -\frac{4}{7}x_3 & -\frac{1}{7}x_4 & & & \\ & x_1 & & +\frac{1}{7}x_3 & +\frac{2}{7}x_4 & & = & \frac{20}{7} \\ & & x_2 & & +x_4 & & = & 3 \\ & & & -\frac{2}{7}x_3 & +\frac{10}{7}x_4 & +x_5 & = & \frac{23}{7} \\ & x_1 & x_2 & x_3 & x_4 & x_5 & \geq & 0 \end{array}$$

   from which we obtain the nonintegral solution $(x_1, x_2) = (\frac{20}{7}, 3)$ and the upper (or dual) bound $\bar{z} = 59/7$

   Is there any straightforward way to find a feasible solution? Apparently not. By convention, as no feasible solution is yet available, we set $\underline{z} = -\infty$.

2. ***Branching:*** Since $\underline{z} < \bar{z}$, (IP) is not solved to optimality yet, and we need to branch. As $x_1 = \frac{20}{7}$, we take

$$S_1 = S \cap \{x : x_1 \leq 2\}, \quad S_2 = S \cap \{x : x_1 \geq 3\}.$$

We now have the tree down in the following figure.

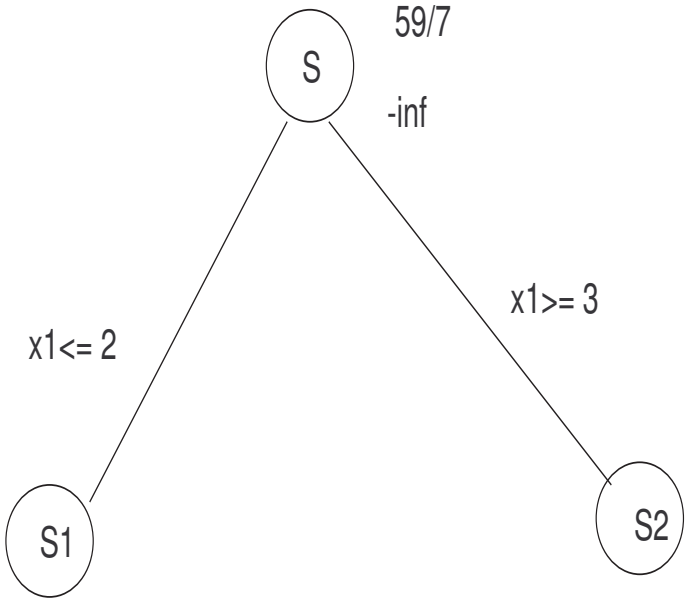

Figure 4: Branching 1

The subproblem (nodes) S_1, S_2 that must still be examined are called ***active.*** The node S on the other hand has been processed and is ***inactive.***

3. ***Choosing an an active node:***

   During the run of the algorithm, we maintain a list of active nodes. This list currently consists of S_1, S_2.

   Later we will discuss breath-first versus depth-first choices of the next active node to be processed. For now we arbitrarily select S_1.

4. ***Bounding:***

   Next we derive a bound $\bar{z}^1$ by solving the LP relaxation

$$\begin{array}{lrll} (IP) & z = & \max & 4x_1 - x_2 \\ & & s.t. & 7x - 1 - 2x_2 \leq 14 \\ & & & x_2 \leq 3 \end{array}$$

$$2x_1 - 2x_2 \leq 3$$
$$x_1 \leq 2$$
$$x \in R^2_+$$

of the problem (IP_1). $\bar{z}^1 = max\{c^T x : x \in S_1\}$ that corresponds to node S_1. Note that (IP_1) differs from (LP) by having one more constraint $x_1 \leq 2$ imposed.

Using $x_1 = \frac{20}{7} - \frac{1}{7}x_3 - \frac{2}{7}x_4$ from the optimal representation of (LP), express the constraint $x_1 \leq 2$ as

$$-\frac{1}{7}x_3 - \frac{2}{7}x_4 + s = -\frac{6}{7}$$

where s is a new slack variable. Thus we have the dual feasible representation:

$$\begin{array}{rlccccccc}
\bar{z} = \max \frac{59}{7} & & & -\frac{4}{7}x_3 & -\frac{1}{7}x_4 & & & & \\
& x_1 & & +\frac{1}{7}x_3 & +\frac{2}{7}x_4 & & & = & \frac{20}{7} \\
& & x_2 & & +x_4 & & & = & 3 \\
& & & -\frac{2}{7}x_3 & +\frac{10}{7}x_4 & +x_5 & & = & \frac{23}{7} \\
& & & -\frac{1}{7}x_3 & -\frac{2}{7}x_4 & & +s & = & -\frac{6}{7} \\
& x_1, & x_2, & x_3, & x_4, & x_5, & s & \geq & 0
\end{array}$$

After two simplex pivots, the linear program is reoptimized, giving with $\bar{z}^1 = 15/2$ and $(\bar{x}^1_1, \bar{x}^1_2) = (2, \frac{1}{2})$.

5. ***Branching.*** S_1 is not solved to optimality and cannot be pruned, so using the same branching rule as before, we have two new nodes

$$S_{11} = S_1 \cap \{x : x_2 \leq 0\}, \quad S_{12} = S_1 \cap \{x : x_2 \geq 1\},$$

and add then to the node list. The tree is now as shown in the folloiwng figure:

and the new list of active nodes is S_{11}, S_{12}, S_2.

6. ***Choosing an active node:*** We arbitrarily choose S_2 for processing.

7. ***Bounding:*** We compute a bound $\bar{z}^2$ by solving the LP relaxation

$$\begin{array}{rlll}
(IP) \qquad z = & \max & 4x_1 - x_2 \\
& s.t. & 7x - 1 - 2x_2 \leq 14 \\
& & x_2 \leq 3 \\
& & 2x_1 - 2x_2 \leq 3 \\
& & x_1 \geq 3 \\
& & x \in R^2_+
\end{array}$$

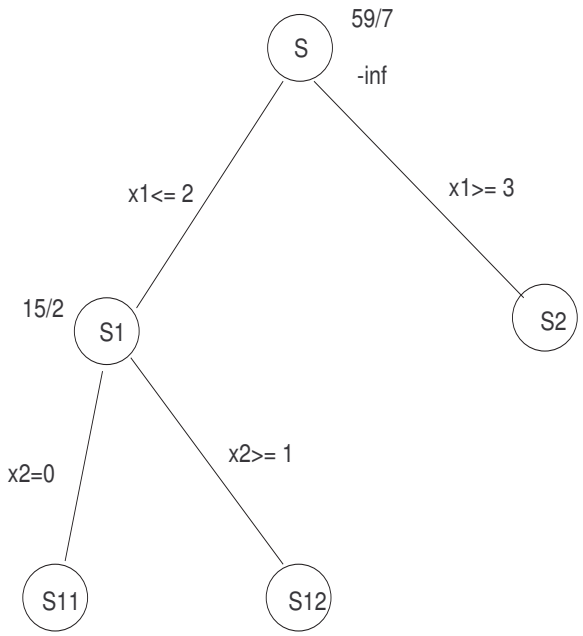

Figure 5: Branching 2

of the problem (IP_2) $z^1 = \max\{c^T x : x \in S_2\}$. To solve $LP(S_2)$, we use the dual simplex algorithm in the same way as above. The constraint $x_1 \geq 3$ is first written as $x_1 - t = 3, t \geq 0$ which expressed in terms of the nonbasic variable becomes:

$$\frac{1}{7}x_3 + \frac{2}{7}x_4 + t = -\frac{1}{7}.$$

From inspection of this constraint, we see that the resulting linear program

$$\begin{array}{rlllllll}
\bar{z} = \max \frac{59}{7} & & -\frac{4}{7}x_3 & -\frac{1}{7}x_4 & & & & \\
x_1 & & +\frac{1}{7}x_3 & +\frac{2}{7}x_4 & & & = & \frac{20}{7} \\
& x_2 & & +x_4 & & & = & 3 \\
& & -\frac{2}{7}x_3 & +\frac{10}{7}x_4 & +x_5 & & = & \frac{23}{7} \\
& & \frac{1}{7}x_3 & +\frac{2}{7}x_4 & & +t & = & -\frac{1}{7} \\
x_1, & x_2, & x_3, & x_4, & x_5, & s & \geq & 0
\end{array}$$

But this LP is infeasible, because the last constraint contradicts $x_6 \geq 0$. Hence, S_2 can be pruned by infeasibility.

8. ***Choosing an active node:*** The list of active nodes is S_{11}, S_{12}. We arbitrarily choose S_{12}.

9. ***Bounding:*** $S_{12} = S \cap \{x : x_1 \leq 2, x_2 \geq 1\}$. The resulting linear program has optimal solution $\bar{x}^{12} = (2, 1)$ with value 7. Since this is an integer solution, $z^{12} = 7$.

10. ***Updating the incumbent:*** We store (2,1) as the best integer solution found so far and update the lower bounds $\underline{z} = \max\{\underline{z}, 7\}$, and S_{12} is now *pruned by optimality.*

11. ***Choosing an active node:*** Only S_{11} is active, so choose this node.

12. ***Bounding:*** $S_{11} = S \cap \{x : x_1 \leq 2, x_2 \leq 0\}$. The resulting linear program has optimal solution $\bar{x}^{11} = (3/2, 0)$ with value $6 < \underline{z} = 7$. Thus the node is *pruned by bound.*

13. ***Termination:*** There are no further active nodes left, and the algorithm terminates, returning the optimal solution $z = 7$ and the maximiser $x = (2, 1)$ that achieves it.

The complete branch-bound tree is shown in the following figure:

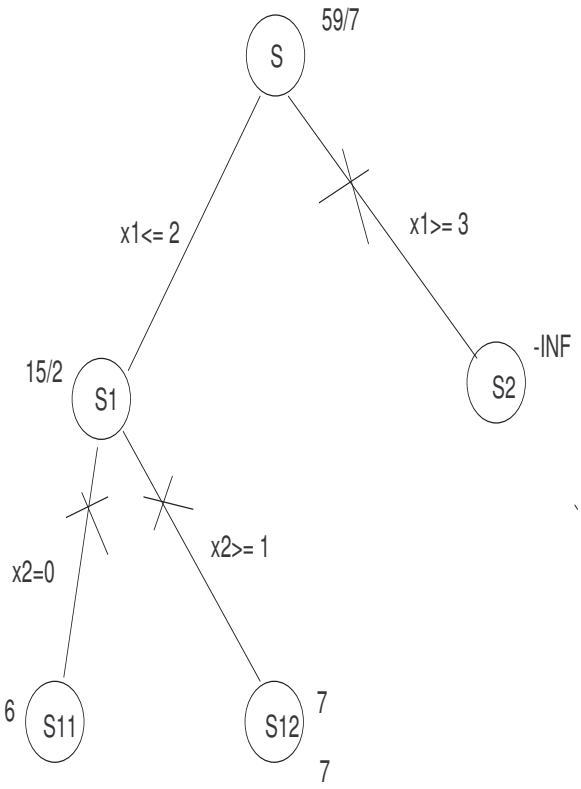

Figure 6: Complete branch and bound tree