IE 511: Integer Programming, Spring 2021 **1 Apr, 2021**

Lecture 20: Unstructured IPs, Pre-processing, Branch and Bound

Lecturer: Karthik Chandrasekaran *Scribe: Karthik*

In the previous lecture, we saw three applications of the dynamic programming technique. In this lecture, we will see one more application of this technique and subsequently move on to solving techniques for unstructured IPs.

20.1 DP Application 4: Maximum weight subtree problem with edge costs

This is a more complex version of the maximum weight subtree problem. This also arises in the context of service providers aiming to decide which customers to connect. The problem is formally stated as follows:

Given: Tree $T = (V, E)$ rooted at $r \in V$, profits $p : V \to \mathbb{R}$, $c : E \to \mathbb{R}_+$
Goal: Subtree rooted at r with maximum profit, i.e.,
$\max \left\{ \sum_{v \in V(T')} p(v) - \sum_{e \in E(T')} c(e) : T' \text{ is a subtree rooted at } r \right\}$.

Exercise. Formulate an IP for this problem.

We will see an algorithm to solve this problem that is based on DP.

States: Let $h(u) :=$ maximum weight of a subtree rooted at u. We need to find $h(r)$.

Deriving the recursion for $h(u)$: If u is a leaf, then $h(u) = \max\{0, p(u)\}$. Suppose u has children $v_1, \ldots, v_k$. Let T' be a maximum weight subtree rooted at u. We have exactly one of two possibilities for T':

1. Either T' is empty,
2. Or T' is composed of subtrees rooted at v_is in which case any subtree rooted at v_i included in T' should be a max-weight subtree rooted at v_i (principle of optimality).

Note that the subtree rooted at v_i in T' can be non-empty only if the edge $uv_i \in E(T')$. Therefore, we have the following recurrence:

Recurrence:

$$h(u) = \max \left\{ 0, p(u) + \max_{x \in \{0,1\}^k} \left\{ \sum_{i=1}^{k} \left(h(v_i) - c(uv_i) \right) x_i \right\} \right\}.$$

Now, how do we compute $h(u)$ if know $h(v_i)$ for the children $v_1, \ldots, v_k$ of u? It seems like we need

to solve a BIP already! Well, observe that this BIP is easy to solve. This is because, the problem

$$\max_{x\in\{0,1\}^k} \sum_{i=1}^{k} \left(h(v_i) - c(uv_i)\right) x_i$$

can be solved in $O(k)$ time by setting

$$x_i^* := \begin{cases} 1 & \text{if } h(v_i) - c(uv_i) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, $h(u)$ can be computed in $O(k)$ time.

Thus, using the recursion, we can compute the $h(\cdot)$ values in a bottom-up fashion in the tree T. This leads to the following theorem:

Theorem 1. *Maximum weight subtree problem with edge costs can be solved in $O(|V|)$ time.*

20.2 Solving techniques for unstructured IPs

So far, we have focused on structured IPs and seen how to solve them—via either reduction to LPs or primal or primal-dual or DP. Next, we move to solving techniques for general IPs, i.e., unstructured IPs. These solving techniques may not necessarily be polynomial-time. We will see solving techniques and also emphasize the general theory underlying these solving techniques.

Recall that a general/unstructured IP is of the form

$$\max\{c^T x : Ax \le b, x \in \mathbb{Z}^n\}.$$

20.3 Pre-Processing

Given an unstructured IP, we first pre-process the IP. This involves three steps, namely

1. tightening bounds,
2. removing redundant constraints, and
3. fixing variables.

These steps are also performed while solving LPs. Let us illustrate these steps with an example.

Example:

$$\begin{aligned}
\max\ & 2x_1 + x_2 - x_3 \\
& 5x_1 - 2x_2 + 8x_3 \le 15 \\
& 8x_1 + 3x_2 - x_3 \ge 9 \\
& x_1 + x_2 + x_3 \le 6 && \longrightarrow \text{remove} \\
& 0 \le x_1 \le 3 && \xrightarrow{\text{after tightening bounds}} 7/8 \le x_1 \le 9/5 \\
& 0 \le x_2 \le 1 \\
& 1 \le x_3 && \xrightarrow{\text{after tightening bounds}} 1 \le x_3 \le 101/64 \\
& x_1, x_2, x_3 \in \mathbb{Z}
\end{aligned}$$

1. *Tighten bounds:*
   - $5x_1 \leq 15 + 2x_2 - 8x_3 \leq 15 + 2 - 8 \implies x_1 \leq 9/5$
   - $8x_3 \leq 15 - 5x_1 + 2x_2 \leq 15 - 0 + 2 = 17 \implies x_3 \leq 17/8$
   - $8x_1 \geq 9 - 3x_2 + x_3 \geq 9 - 3 + 1 = 7 \implies x_1 \geq 7/8$
   - $8x_3 \leq 15 - 5x_1 + 2x_2 \leq 15 - 5(7/8) + 2 = 101/8 \implies x_3 \leq 101/64$
2. *Remove redundant constraints*: $x_1 + x_2 + x_3 \leq 6$ becomes redundant so remove it.
3. *Fix Variables*:
   - Increasing x_2 will not violate the constraints and will improve the objective. Therefore, $x_2 = 1$ in the optimal solution (if it exists).
   - $1 \leq x_3 \leq 101/64$ and $x_3 \in \mathbb{Z}$ which implies that $x_3 = 1$ in the optimal solution (if it exists).
   - $7/8 \leq x_1 \leq 9/5$ and $x_1 \in \mathbb{Z}$ which implies that $x_1 = 1$ in the optimal solution (if it exists).

After fixing variables, we need to verify feasibility. We see that $x_1 = 1, x_2 = 1, x_3 = 1$ does not violate any constraint and is hence an optimal solution.

In the above example, we were lucky to obtain an optimal solution purely by pre-processing. This lucky situation may not happen with all unstructured IPs. However, pre-processing goes a long way towards simplifying the instance and getting to the difficult part of the instance.

20.4 Branch and Bound (B&B)

Divide and Conquer is a popular technique in algorithm design. B&B is an extension of Divide and Conquer to solve IPs.

20.4.1 Branching

Consider the IP

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

where $S \subseteq \mathbb{Z}^n$ is some feasible region. We break it into subproblems: Let $S = S_1 \cup S_2 \cup \cdots \cup S_k$ be a decomposition of S and let

$$z^i := \max\{c^T x : x \in S_i\} \quad \forall i \in [k].$$

We observe that $z = \max\{z^i : i \in [k]\}$. This is a simple observation. It turns out to be helpful for many IPs.

A convenient way to represent the branching part of the B&B technique (i.e., the dividing aspect of the Divide and Conquer approach) is via an *enumeration tree*. We illustrate the *enumeration tree* with two examples.

Example 1. Suppose $S \subseteq \{0,1\}^3$. Then a possible enumeration tree is as shown in Figure 20.1 where $S_0 := \{x \in S : x_1 = 0\}$, $S_1 := \{x \in S : x_1 = 1\}$ and proceed recursively. This style of enumeration tree for $S \subseteq \{0,1\}^n$ will have as many as 2^n vertices in the enumeration tree. In particular, in order to find a min cost TSP tour on n cities (complete graph instance with input cost c_{ij} on edge ij) if we use X_{ij} as indicator variables to determine if city i is followed by city j in the tour, then we would have an enumeration tree with $2^{\binom{n}{2}}$ nodes.

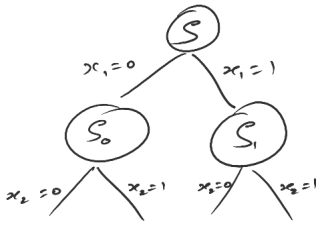
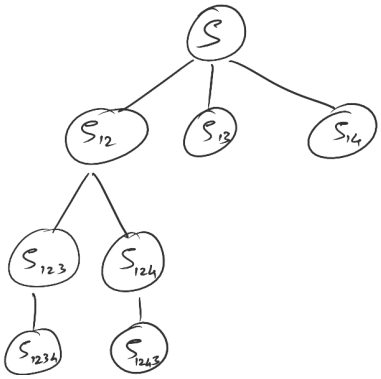


Figure 20.1: Example of an enumeration tree

Example 2: Say we want to find a min cost TSP tour on n cities. Let S be the set of all tours. S can be partitioned into three types: S_{ij} := set of tours in S such that city j immediately follows city 1. If we have n cities, then the number of vertices in this type of enumeration tree is $(n-1)!$ ($\approx n^n \approx 2^{n \log n}$). Note that this enumeration tree has much fewer vertices for the n-city TSP than the previous one.



The main issue with the branching technique is that we could produce a large enumeration tree, i.e., an enumeration tree where the number of vertices is exponential in the input size. Note that larger enumeration tree results in more computation time. So, complete enumeration would take too much time. We will avoid complete enumeration by cleverly pruning the enumeration tree. We will prune unnecessary parts of the tree. To prune, we will exploit relaxation and bounding techniques.

20.4.2 Pruning Enumeration Trees

We will discuss pruning rules by considering the objective to be maximization. If the objective is minimization, then you can either bring the problem to maximization form and apply the same rules that we discuss or modify the rules suitably to apply them for the minimization problem.

We will see pruning rules based on bounds. The subproblem corresponding to node S may not

be solvable efficiently. However, typically we will be able to get some feasible solution and solve a relaxation quickly.

- Solving a relaxation of a maximization problem gives an upper bound u on the objective value of the original problem.
- The objective value of a feasible solution to a maximization problem gives a lower bound l on the objective value of the original problem.

Therefore, for S, we will be able to get l, u such that $l \leq z_S \leq u$ quickly. We will use these bounds to cleverly prune the tree. We will illustrate the pruning rules through examples.

Example 1: See Figure 20.2.

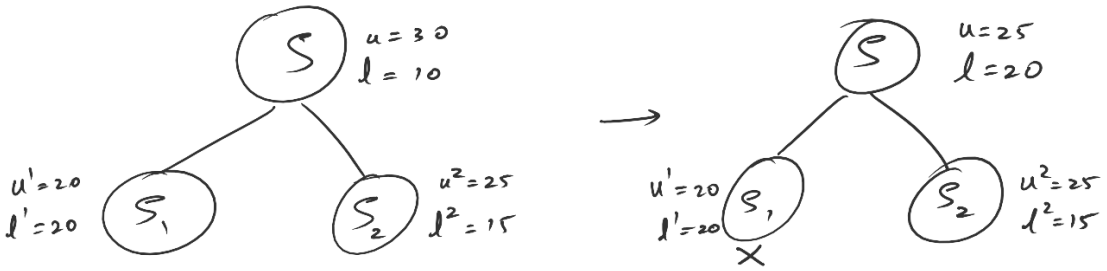


Figure 20.2: The bounds of the sub-problems allow us to (1) update the bounds for z_S and (2) then prune subproblem S_1 as it has already been solved to optimality.

In Figure 20.2, we found better lower and upper bounds for z using the upper and lower bounds for subproblems. Furthermore, branch S_1 has achieved optimality because $c^1 = l^1$. So, we do not have to explore branch S_1 further.

These observations lead to the following general propositions:

Proposition 2 (Updating bounds)**.** *Suppose $l^i \leq z^i \leq u^i \ \forall i \in [k]$. Then*

$$\max\{l^i : i \in [k]\} \leq z_S \leq \max\{u^i : i \in [k]\}.$$

Proposition 3 (Pruning by optimality)**.** *If $l^i = u^i$ for some i, then the optimum solution for the i^{th} subproblem $z^i := \max\{c^T x : x \in S_i\}$ is known, so we do not have to explore branch S_i further, i.e., prune S_i.*

Any other reason to prune? Here is an example:

Example 2: See Figure 20.3

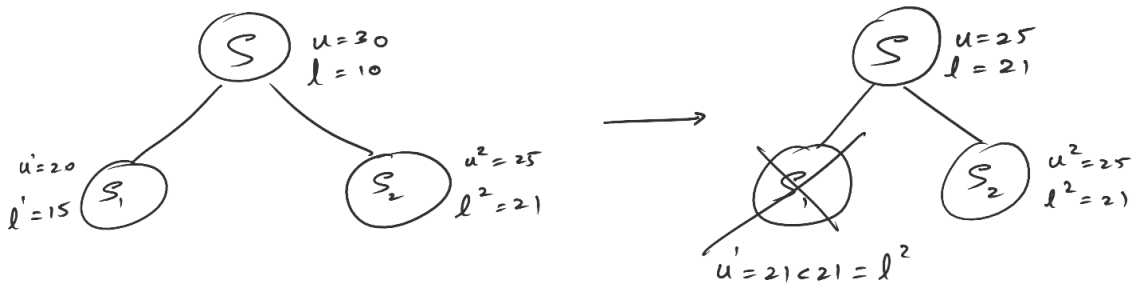


Figure 20.3: The bounds of the sub-problem allow us to prune S_1 as we can provably say that the optimal solution is not coming from that sub-problem.

In Figure 20.3, the branch rooted at S_1 does not contain an optimal solution to the original problem because $u^1 \leq l$ (any solution under S_1 has value $\leq 20 = u^1$ and we already have a solution of value $\geq 21 = l$). So, prune the branch S_1 and update the bounds.

These observations lead to the following general proposition:

Proposition 4 (Pruning by bound)**.** *If $u^i < l$ for some i, then the optimal solution for $z^i = \max\{c^T x : x \in S_i\}$ can never be an optimal solution to the original problem, i.e., prune S_i.*

We have one more reason to prune:

Proposition 5 (Pruning by infeasibility)**.** *If $S_i = \emptyset$, then prune S_i.*

In spite of pruning rules, we may not be able to prune sometimes. In such scenarios, we might have to explore the branches further to obtain better bounds, then percolate these bounds upward, and verify if further pruning is possible.

Example 3: See Figure 20.4.

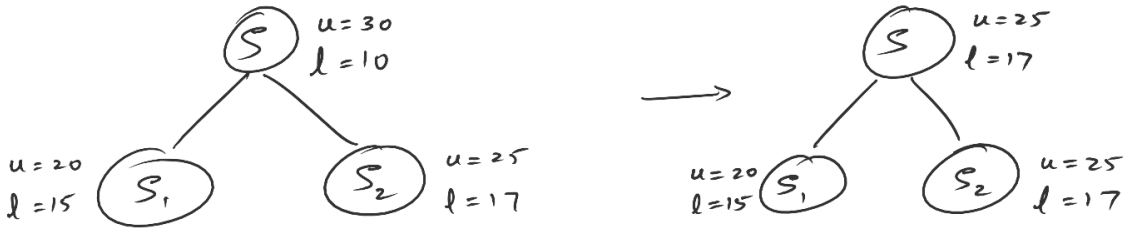


Figure 20.4: The bound on S_1 and S_2 lead to better bounds for the problem but we still need to explore both subtrees.