20.1 Dynamic Programming

In the previous lecture, we saw two applications of the dynamic programming technique. In this lecture we will see two more applications of dynamic programming.

20.1.1 Application 3: Optimal Subtree Problem

Given: A tree $T = (V, E)$ rooted at $r \in V$, $p : V \to \mathbb{R}$ (could be negative)

Goal: Find a subtree of $T$ rooted at $r$ with maximum weight.

Example: See Figure 20.1.

Exercise. Formulate an IP for this problem.

States: Let $g(v) :=$ optimal value of a subtree rooted at $v$. We need to find $g(r)$.

Recursion: The recursion is typical for DP algorithm on trees. Compute $g(v)$ bottom up starting from the leaves. If $v$ is a leaf, then $g(v) = \max \{0, p(v)\}$. Suppose $v$ has children $v_1, \ldots, v_k$. Let $T'$ be an optimal subtree rooted at $v$. We have two possibilities:

1. Either $T'$ is empty,
2. or $T'$ is composed of subtrees rooted at $v_i$s in which case any subtree rooted at $v_i$ included in $T'$ should be an optimal subtree rooted at $v_i$ (principle of optimality).
Therefore, \( g(v) = \max \left\{ 0, p(v) + \sum_{i=1}^{k} g(v_i) \right\} \).

**Theorem 1.** Optimal subtree problem can be solved in \( O(|V|) \) time.

**Proof.** Use the recursion to compute \( g(v) \) bottom up (from the leaves to the root). Time to compute \( g(v) \) is \( O(|\text{children}(v)|) \). Total time is \( O(\sum_{v \in V} |\text{children}(v)|) = O(|V|) \). \( \square \)

### 20.1.2 Application 4: Optimal subtree problem with edge costs

**Given:** Tree \( T = (V, E) \) rooted at \( r \in V \), \( p : v \rightarrow \mathbb{R}, c : E \rightarrow \mathbb{R}_+ \)

**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.

**States:** Let \( h(v) := \text{optimal profit from subtree rooted at } v \). We need to find \( h(r) \).

**Recursion:** If \( v \) is a leaf, then \( h(v) = \max \{0, p(v)\} \). Suppose \( v \) has children \( v_1, \ldots, v_k \). Let \( T' \) be an optimal subtree rooted at \( v \). Note that the subtree routed at \( v_i \) in \( T' \) can be non-empty only if the edge \( vv_i \in E(T') \). Therefore,

\[
h(v) = \max \left\{ 0, p(v) + \max_{x \in \{0,1\}^k} \left\{ \sum_{i=1}^{k} (h(v_i) - c(vv_i)) x_i \right\} \right\}
\]

where \( x_i = \begin{cases} 1 & \text{if optimal subtree routed at } v_i \text{ is included in } T' \\ 0 & \text{otherwise.} \end{cases} \)

We also observe that

\[
\max_{x \in \{0,1\}^k} \sum_{i=1}^{k} (h(v_i) - c(vv_i)) x_i
\]

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

\[
x^*_i := \begin{cases} 1 & \text{if } h(v_i) - c(vv_i) > 0 \\ 0 & \text{otherwise.} \end{cases}
\]

Therefore, \( h(v) \) can be computed in \( O(k) \) time.

**Theorem 2.** Optimal Subtree Problem with edge costs can be solved in \( O(|V|) \) time.
20.2 Unstructured IPs

Next, we will see solving techniques for unstructured IPs max\{c^T x : Ax \leq b, x \in \mathbb{Z}^n\}. These solving techniques may or may not be polynomial-time.

20.2.1 Pre-Processing

Given an unstructured IP, we first pre-process the IP. This involves three steps, namely tightening bounds, removing redundant constraints and fixing variables. These steps are also performed with LPs. Let us illustrate these steps with an example.

Example:

max \ 2x_1 + x_2 - x_3
\ 5x_1 - 2x_2 + 8x_3 \leq 15
\ 8x_1 + 3x_2 - x_3 \geq 9
\ x_1 + x_2 + x_3 \leq 6
\ 0 \leq x_1 \leq 3
\ 0 \leq x_2 \leq 1
\ 1 \leq x_3
\ x_1, x_2, x_3 \in \mathbb{Z}

---

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.
20.2.2 Solving Unstructured IPs: Branch and Bound (B&B) Technique

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

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

We break it into subproblems. Let \( S = S_1 \cup S_2 \cup \cdots \cup S_k \) be a decomposition of \( S \) and let

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

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 a 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 20.2 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 \( 2^n \) nodes 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 \( 2^n \) nodes.

![Figure 20.2: Example of an enumeration tree](image)

**Example 2:** Say we want to find a min cost TSP tour on four 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 nodes in this enumeration tree is \( (n - 1)! \) (\( \approx n^n \approx 2^{n \log n} \)). Note that this enumeration tree has much fewer nodes 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. Note that larger enumeration tree leads to more computation time. So, complete enumeration would take too much time. We will avoid complete enumeration by cleverly pruning the enumeration tree. We will cut off unnecessary parts of the tree. To prune, we will exploit relaxation and bounding techniques.