COMPSCI 330: Design and Analysis of Algorithms 2/2/2016 and 2/4/2016

Dynamic Programming

Lecturer: Debmalya Panigrahi *Scribe: Tianqi Song, Tianyu Wang*

1 Overview

In this lecture we introduce dynamic programming. Dynamic programming is method to quickly solve large problems by first solving intermediate problems, then using these intermediate problems to solve the large problem. We will illustrate the idea of dynamic programming via examples.[1]

2 Longest Increasing Subsequence

We starts with an application of dynamic programming to finding a longest increasing subsequence.

Definition 1. *A subsequence of sequence $x_1, \dots, x_n$ is some sequence $x_{\phi(1)}, \dots, x_{\phi(h)}$ such that for all k, $1 \le k \le h$, we have $1 \le \phi(k) \le n$; and for any x_j in the subsequence, all x_i preceding x_j in the subsequence satisfy $i < j$. An increasing subsequence is a subsequence such that for any x_j in the subsequence, all x_i preceding x_j in the subsequence satisfy $x_i < x_j$. A largest increasing subsequence is a subsequence of maximum length.*

Note that the longest increasing subsequence need not be unique. For example, consider the following subsequence.

$$11 \quad 14 \quad 13 \quad 7 \quad 8 \quad 15 \tag{1}$$

The following is a subsequence.

$$14 \quad 8 \quad 15$$

A longest increasing subsequence of the sequence given in 1 is

$$11 \quad 13 \quad 15$$

In this case, there are also two other longest increasing subsequences:

$$7 \quad 8 \quad 15$$
$$11 \quad 14 \quad 15$$

The problem we will solve is to find a longest increasing subsequence. What kind of subproblem will help with this? Let the input sequence be denoted $v_1, \dots, v_n$. We have the following two options:

Option 1 v_n is in the subsequence.

Option 2 v_n is not in the subsequence.

[1]Some of the material in this note is from a previous note by Samuel Haney for this class in Fall 2014.

Option 2 is easy, we just need to solve the same problem on a smaller sequence, so we can recurse. However, to solve Option 1, we need to recurse on a slightly stronger problem: we would like $LIS(k)$ to be the longest increasing subsequence that ends at v_k. Formally, we have the following expression:

$$LIS(k) = \max_{\substack{j<k \\ v_j<v_k}} \{LIS(j)\} + 1 \tag{2}$$

To finally solve our original problem, we find

$$LIS = \max_{k} \{LIS(k)\}.$$

Again, implementing this naively using recursion is slow. Instead, we want to use dynamic programming. That is, we want to start with $k = 1$ and then increase k, instead of starting with $k = n$ and recursing. We define this formally in Algorithm 1.

Algorithm 1 Longest Increasing Subsequence

```
1: function LIS($v_1, \ldots, v_n$)
2:     for $k \leftarrow 1$ to $n$ do
3:         $length[k] \leftarrow 1$
4:         for $j \leftarrow k+1$ to $n$ do
5:             if $v_j > v_k$ then
6:                 $length[j] \leftarrow \max\{length[j], length[k]+1\}$
7:     return $\max_{1\le i\le n}\{length[i]\}$
```

The runtime of Algorithm 1 is $O(n^2)$ because the nested loop is $O(n^2)$.

3 Knapsack Problem (with integer weights)

We now move to another problem: knapsack with integer weights. An instance of the knapsack problem is a set of n items, denoted I. Each item has a value and a weight; the value and weight of the ith item are denoted v_i and w_i respectively. We are given some budget W, and the goal is to select some subset of items, $I' \subseteq I$, such that

$$\sum_{i\in I'} w_i \le W,$$
$$\sum_{i\in I'} v_i \quad \text{is maximized.}$$

Unlike our previous discussion of this problem, we will not allow selecting fractions of an item. Only whole items may be selected. Again, let's try to break down the problem:

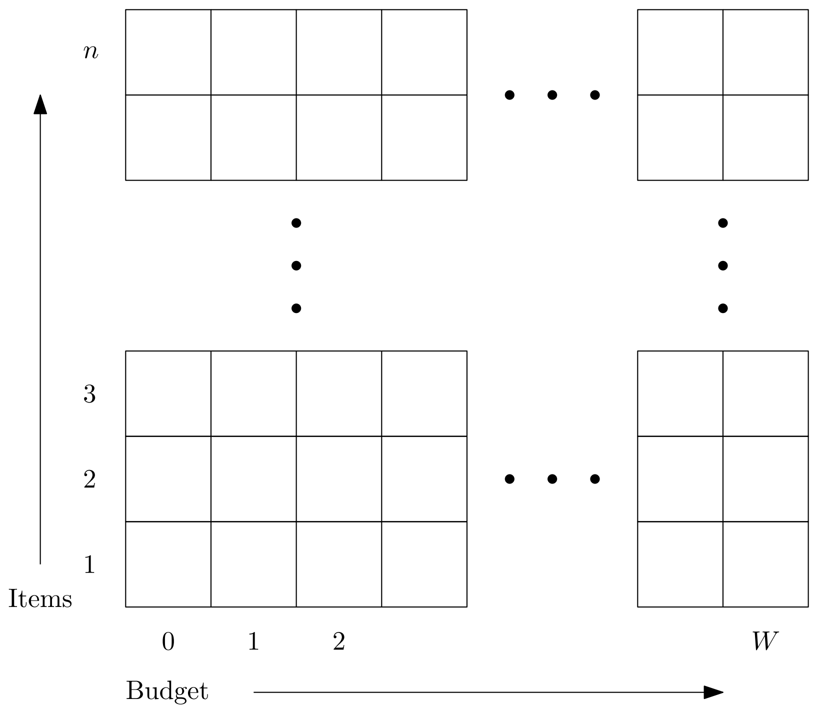
Option 1 $v_n \in OPT$.

Option 2 $v_n \notin OPT$.

For Option 2, we again recurse on the smaller problem. For Option 2, we recurse with a new budget of $W - w_n$. Therefore, our dynamic program needs to solve the knapsack problem for all smaller budgets.

$$KS(W,n) = \max\{KS(W,n-1), KS(W-w_n,n-1)+v_n\} \tag{3}$$

Unlike our previous examples, this recursion has two parameters, so we will need to fill in a two dimensional table.



From the recursion, we know that each entry in the table depends on two values in the row below it. Therefore, we should fill in the bottom row first, then continue filling in rows bottom to top. Each row can be filled in any order. Calculating the value of each entry takes constant time. Therefore, the running time is proportional to the size of the table, $O(nW)$. This running time is polynomial in the value of one of the inputs, W, and is therefore a pseudo-polynomial time algorithm.

4 Maximal Matching on Trees

We present one more application of dynamic programming – maximal matching on trees. On general graphs, maximal matching is NP-hard. As we will see later in the class, it is even hard to approximate. However, the problem becomes much easier when restricted to trees.

Definition 2. *Let $G = (V, E)$ be a graph. A matching is a set of edges without common vertices.*

Definition 3. *A maximal matching on a graph $G = (V, E)$ is a matching with the largest number of edges.*

In addition, we also define trees.

Definition 4. *A tree is an acyclic, connected graph.*

Let $M(v)$ be the size of the maximal matching in the subtree rooted at v. Like before, we have two options.

Option 1 v incidents with an edge in the maximal matching of the subtree rooted at v.

Option 2 v does not incident with any edge in the maximal matching of the subtree rooted at v.

Let $c(v)$ denote the children of vertex v. For Option 1,

$$M(v) = 1 + \max_{u \in c(v)} \{ \sum_{i \in c(v), i \neq u} M(i) + \sum_{j \in c(i)} M(j) \}$$

For Option 2,

$$M(v) = \sum_{i \in c(v)} M(i)$$

From here we can write out the recurrence relation.

$$M(v) = \max \left\{ 1 + \max_{u \in c(v)} \{ \sum_{i \in c(v), i \neq u} M(i) + \sum_{j \in c(i)} M(j) \}, \sum_{i \in c(v)} M(i) \right\}. \tag{4}$$

Our dynamic program should run from the leaves of the up to the root. The running time will be $O(n)$, because every node has at most one parent and one grandparent.

5 Maximal Independent Set on Trees

Maximal independent set is similar to maximal matching. They both are packing problem.

Definition 5. *Let $G = (V, E)$ be a graph. An independent set is a set of vertices $\{v_1, \ldots, v_k\} \subseteq V$ such that for all i, j, with $1 \leq i \leq k$ and $1 \leq j \leq k$, $(v_i, v_j) \notin E$.*

Let $IS(v)$ be the size of the largest independent set in the subtree rooted at v. Like before, we have two options.

Option 1 v is in the largest independent set of the subtree rooted at v.

Option 2 v is not in the largest independent set of the subtree rooted at v.

Note that v can only appear in the independent set if none of its children are in the independent set.

$$IS(v) = \max \left\{ 1 + \sum_{w \in \text{granchildren}(v)} IS(w), \sum_{w \in \text{children}(v)} IS(w) \right\}. \tag{5}$$

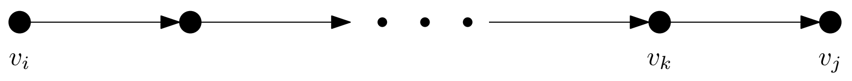
Our dynamic program should run from the leaves of the up to the root. The running time will be $O(n)$, because every node has at most one parent and one grandparent.

6 Shortest Path in a DAG

Next, we try to find the shortest path in a directed acyclic graph (DAG). Recall that a DAG has directed edges and contains no cycles. Recall the definition of a topological sort:

Definition 6. *Let $G = (V, E)$ be a graph. Let $v_1, \ldots, v_n$ be an ordering of the vertices in V. $v_1, \ldots, v_n$ are in topologically sorted order if for all edges $(v_i, v_j) \in E$, $i < j$.*

If G is a DAG, then it is always possible to find a topological sorting of the vertices. This ordering is not necessarily distinct. Consider some shortest path on the DAG:



Note that the shortest path to v_j is the shortest path to v_k, plus the edge (v_k, v_j). This is a property we have used before. Additionally, we know that $k < j$ since we have assumed that the vertices are in topologically sorted order. How does this help us? We have the following:

$$SP(v_i, v_j) = \min_{v_k \in V} \left\{ SP(v_i, v_k) + \ell(v_k, v_j) \right\}, \tag{6}$$

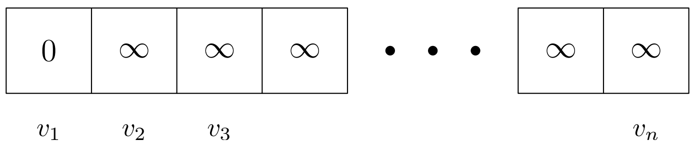
where $\ell(v_k, v_j)$ is the length of the edge (v_k, v_j). Can we use Equation 6 to write a recursion? This is not clear. A recursion must make progress, and this one does not. Therefore, this recursion will not necessarily terminate. Fortunately, an additional property of the topological sort fixes this problem:

$$SP(v_i, v_j) = \min_{\substack{v_k \in V \\ k<j}} \left\{ SP(s, v_k) + \ell(v_k, v_j) \right\}. \tag{7}$$

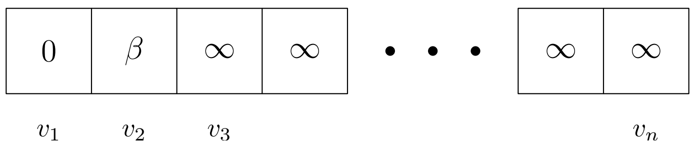
Now our recursion is well-defined. Is a recursive algorithm based on this recursion efficient? For each vertex v_j that we visit, we will potentially need to make a recursive call for each v_k where $k < j$ (this is the case if every vertex preceding v_j has an edge to v_j). Therefore, our recurrence relation is

$$\begin{aligned} T(n) &= \sum_{i<n} T(i) + O(n) \\ &\approx O(n^n). \end{aligned} \tag{8}$$

This is extremely slow! To fix this, we will solve the subproblems bottom-up instead of top-down. This will prevent us from needlessly solving the same subproblems multiple times, which is causing the slow runtime. We want to fill in the following table. Initially, v_1 is zero, and the rest of the values in the table are ∞.



We have everything that we need to fill in v_2. If we suppose there is an edge (v_1, v_2) of length β, we get



Now, we have everything we need to fill in the value of v_3! In general, when solving for v_j, we consider all vertices v_i such that there is an edge (v_i, v_j). The value of v_i plus $\ell(v_i, v_j)$ is a potential value for v_j. To find the best value, we take the minimum of this expression over all such vertices v_i (this precisely what is asserted by Equation 7). This process is described formally in Algorithm 2.

Algorithm 2 Shortest Path in a DAG

```
function SP($V, E, s$)
    $\{v_1, \cdots, v_n\} \leftarrow$ TopSort(V)
Assume: $v_i = s$
    $d[v_i] \leftarrow 0$
    for $v_j \neq v_i$ do
        $d[v_j] \leftarrow \infty$
    for $j \leftarrow 1$ to $n$ do
        for $k < j$ do
            if $d[j] > d[k] + \ell(v_k, v_j)$ then
                $d[j] \leftarrow d[k] + \ell(v_k, v_j)$
```

The result of this algorithm will be an array of values where each value is the shortest path in the DAG from s to the vertex corresponding to that index in the array. To calculate the value in location i, this algorithm takes $O(i)$ time. Summed over all locations in the array, the running time is $O(n^2)$.

In general, we solve dynamic programs in the following two steps:

1. Come up with a table.
2. Move in the table so that we solve a problem whose required subproblems have all been solved already.

7 Longest Decreasing Continuous Subsequence

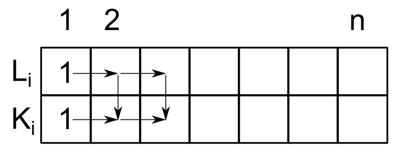
Problem description: Given a sequence of numbers $a_1, a_2, ...a_n$, find the longest subsequence $a_{k1}, a_{k2}, ..a_{kw}$ such that $a_{ki} > a_{kj}$ and $kj - ki = 1$ for all $j = i + 1$.
Solution: We solve this problem by dynamic programming. Define K_i as the longest such sequence in the first i numbers, and L_i as the longest such sequence ending at the ith number. We have:

$$K_{i+1} = max(L_{i+1}, K_i) \tag{9}$$

$$L_{i+1} = 1 \quad if \quad a_{i+1} \geq a_i \tag{10}$$

$$L_{i+1} = L_i + 1 \quad if \quad a_{i+1} < a_i \tag{11}$$

The dynamic programming table is like:



It takes constant time to fill one item in the table and then the running time of this dynamic programming algorithm is $O(n)$.

8 Minimum Edit Distance

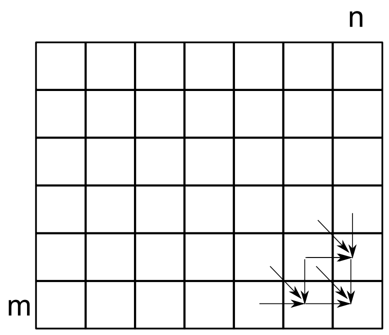
Problem description: Given two strings $a_1a_2...a_n$ and $b_1b_2...b_m$, find their minimum edit distance. The edit distance is the number of mismatches in an alignment, for example, the edit distance between the two strings *SUNNY* and *SNOWY* in the following alignment is 3:

$$\begin{array}{cccccc} S & \boxed{U} & N & \boxed{N} & \boxed{\_} & Y \\ S & \boxed{\_} & N & \boxed{O} & \boxed{W} & Y \end{array}$$

Solution: Define $E[k,l]$ as the minimum edit distance between $a_1a_2...a_k$ and $b_1b_2...b_l$. There can be three kinds of alignments:

$$(1)\ \begin{array}{cc} \ldots\ldots & - \\ \ldots\ldots & b_l \end{array} \qquad (2)\ \begin{array}{cc} \ldots\ldots & a_k \\ \ldots\ldots & - \end{array} \qquad (3)\ \begin{array}{cc} \ldots\ldots & a_k \\ \ldots\ldots & b_l \end{array}$$

Therefore, we have $E[k,l] = minimum(1+E[k,l-1]),1+E[k-1,l],W)$, where $W = E[k-1,l-1]$ if $a_k = b_l$, and $W = E[k-1,l-1]+1$ if $a_k \neq b_l$. The dynamic programming table is like:



It takes constant time to fill one item in the table and then the running time of this dynamic programming algorithm is $O(nm)$.