COMPSCI 330: Design and Analysis of Algorithms 9/23/2014

Lecture #9

Lecturer: Debmalya Panigrahi *Scribe: Samuel Haney*

1 Overview

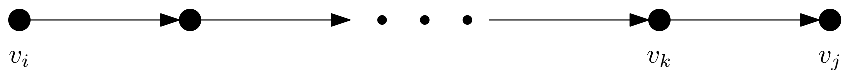
In this lecture we introduce dynamic programming. Dynamic programming is method to quickly solve large problems by first solving intermediate problems, then using these intermediate problems to solve the large problem. Note that we have already seen algorithms like this (e.g. Dijkstra's algorithm).

2 Shortest Path in a DAG

We begin with trying 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 1. *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:

$$SP(v_i, v_j) = \min_{v_k \in V} \left\{ SP(v_i, v_k) + \ell(v_k, v_j) \right\}, \tag{1}$$

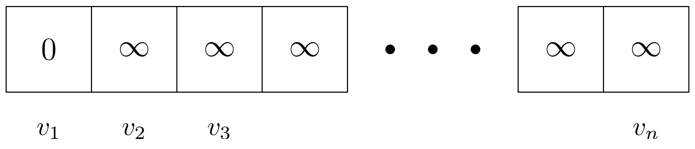
where $\ell(v_k, v_j)$ is the length of the edge (v_k, v_j). Can we use Equation 1 to write a recursion? This is not clear. A recursion must make progress, and this one does not. Therefore, this recursion will not necessarily terminate. Fortunately, an additional property of the topological sort fixes this problem:

$$SP(v_i, v_j) = \min_{\substack{v_k \in V \\ k < j}} \left\{ SP(s, v_k) + \ell(v_k, v_j) \right\}. \tag{2}$$

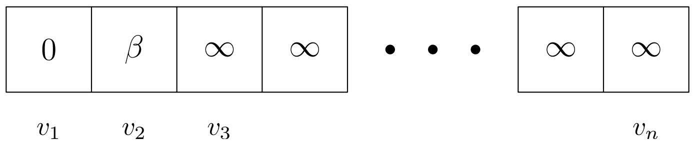
Now our recursion is well-defined. Is a recursive algorithm based on this recursion efficient? For each vertex v_j that we visit, we will potentially need to make a recursive call for each v_k where $k < j$ (this is the case if every vertex preceding v_j has an edge to v_j). Therefore, our recurrence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

$$\begin{aligned} T(n) &= \sum_{i<n} T(i) + O(n) \\ &\approx O(n^n). \end{aligned} \tag{3}$$

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 2). This process is described formally in Algorithm 1.

Algorithm 1 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.

3 Largest Increasing Subsequence

In this section, we give another application of dynamic programming.

Definition 2. *A subsequence of sequence $x_1, \ldots, x_n$ is some sequence $x_{\phi(1)}, \ldots, x_{\phi(h)}$ 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 largest increasing subsequence need not be unique. For example, consider the following subsequence.

$$11 \quad 14 \quad 13 \quad 7 \quad 8 \quad 15 \tag{4}$$

The following is a subsequence.

$$14 \quad 8 \quad 15$$

A largest increasing subsequence of the sequence given in 4 is

$$11 \quad 13 \quad 15$$

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

$$\begin{array}{ccc} 7 & 8 & 15 \\ 11 & 14 & 15 \end{array}$$

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

Option 1 v_n is in the subsequence.

Option 2 v_n is not in the subsequence.

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:

$$LIS(k) = \max_{\substack{j<k \\ v_j < v_k}} \{LIS(j)\} + 1 \tag{5}$$

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 2.

Algorithm 2 Largest Increasing Subsequence

```
function LIS($v_1, \ldots, v_n$)
    $lis[1] \leftarrow 1$
    for $k \leftarrow 2$ to $n$ do
        $lis[k] \leftarrow 0$
        if $lis[j] + 1 > lis[k]$ then
            $lis[k] \leftarrow lis[j] + 1$
    $lis \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        if $lis[k] > lis$ then
            $lis \leftarrow lis[k]$
    return $lis$
```

The runtime of Algorithm 2 is $O(n^2)$ by same argument as we used for Algorithm 1.

4 Knapsack Problem (with integer weights)

We now move to a more difficult 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

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

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:

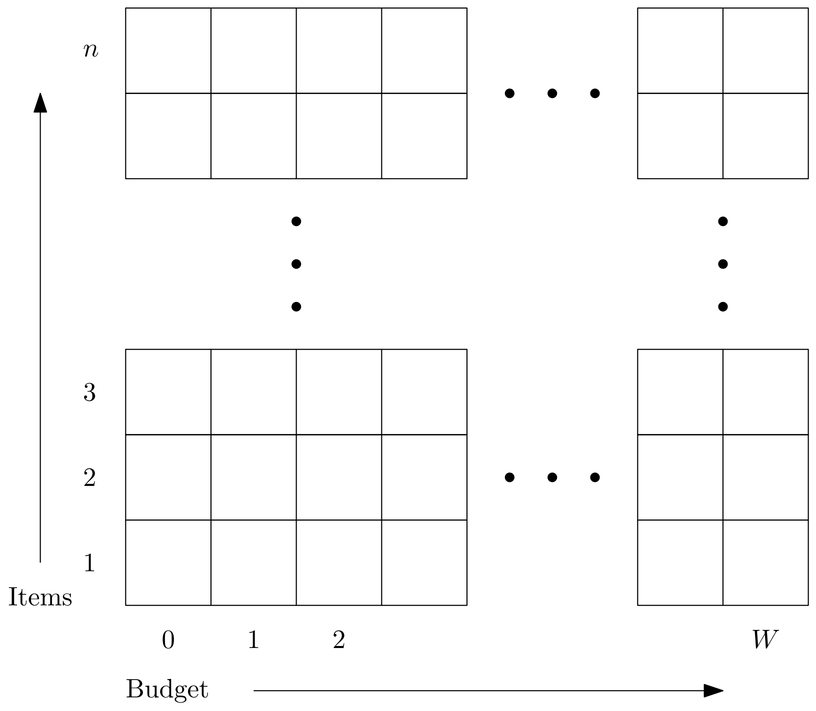
Option 1 $v_n \in OPT$.

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.

$$KS(W,n) = \max\{KS(W,n-1), KS(W - w_n, n-1) + v_n\} \tag{6}$$

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.

5 Independent Set on Trees

We present one last application of dynamic programming – independent set on trees. On general graphs, independent set 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 3. *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 $1 \leq j \leq k$, $(v_i,v_j) \notin E$.*

For completeness, we also define trees.

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

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.

$$IS(v) = \max\left\{1 + \sum_{w \in \text{granchildren}(v)} IS(w), \sum_{w \in \text{children}(v)} IS(w)\right\}. \tag{7}$$

Our dynamic program should run from the leaves of the up to the root. The running time will be $O(n)$. We leave the proof of this running time as an exercise.