Lecture 3: DAGs and Dynamic Programming

1 Philosophy

A major goal in this lecture will be to make explicit the following point about dynamic programs. They can be used to solve problems when the "dependency graph" of states has no cycles. By dependency graph, we mean that we draw an arrow/edge from a state S to a state T if state T requires the value of state S in its formula. We give several varied examples of this.

Objectives of this lecture

In this lecture we will discuss a bit more of knapsack dynamic programming, and start discussing other kinds of dynamic programming, specifically those on directed acyclic graphs, which naturally induce a DP state. In some sense, the existence of a DP means that the problem itself has a DAG structure hidden somewhere. The problems we will consider are:

- Shortest paths with negative edge weights in a DAG.
- Subset dynamic programming.
- The edit distance and longest common subsequence problem.
- Interesting ways to pick states for a DP.

2 Example Problems

2.1 Shortest Paths in a DAG

You have seen the shortest path problem several times before. Today we consider a more challenging version where the edges are allowed to have negative weights. However, we will only consider the special case where the graph is guaranteed to be a directed acyclic graph (DAG), a directed graph with no cycles.

Problem: Shortest Paths in a DAG

The input consists of a a directed acyclic graph (DAG) G whose edges have lengths (not necessarily positive), and a vertex s in the graph. For every other vertex t, report the shortest path distance from s to t (or report that t is not reachable from s).

We give an algorithm to solve this problem in $O(m)$ time, where m is the number of edges in the graph. Note that the standard Dijkstra's algorithm does not work, because that algorithm assumes positive edge lengths. In 15-210, you may have seen the *Bellman-Ford* algorithm, which does work for negative lengths, but its runtime is $O(mn)$. You will not need to know any aspects of that algorithm to understand the algorithm we present today.

Finding a *topological sort* of the graph. Every DAG on n vertices admits a topological sort: an ordering of the vertices where all edges are increasing with respect to this ordering. Formally, a function $\pi : V(G) \to \{1, \ldots, n\}$ such that for every edge $e = (i, j)$, it holds that $\pi(i) < \pi(j)$. The first step is to process the graph G and find a topological ordering. We leave this as an exercise. Equivalently, u must come before v if there is a edge (or path) from u to v. **Hint:** There is always a vertex with 0 incoming edges in a DAG. Find it, put it first in the topological ordering, delete it, and repeat.

Processing vertices in the order of the topological ordering. The algorithm is very simple. Go through the vertices in the order of the topological ordering. Let's say we are processing the vertex v. A shortest path to v must go through some edge of the form (u, v). The shortest path to v then consists of the shortest path to u (which we have already computed, since u is before v in the topological ordering, **this is a critical point**), plus the length of the edge (u, v), which we'll denote as $\ell(u, v)$.

Below we give the formula. We assume that we can preprocess the graph so that for all vertices v, we can find all edges $e = (u, v)$ that go into v. This can be done in $O(m)$ time.

Initially, set $\text{dist}[v] = +\infty$ for all $v \in V$, except $\text{dist}[s] = 0$. Go through vertices v in the order of the topological sort. For a vertex $v \neq s$, update

$$\text{dist}[v] = \min_{\text{edges } (u,v)} (\text{dist}[u] + \ell(u, v)).$$

2.2 Subset Dynamic Programming

In this section we will describe exponential time dynamic programs which still provide a runtime speedup in some cases.

Problem: Counting topological orderings

Let G be a DAG with n vertices. Count the number of topological orderings of G.

Problem: Traveling Salesperson / Hamiltonian Path

Let G be a directed graph on n vertices whose edges have lengths. Determine if there is a path in G that visits each vertex exactly once, and if so find its minimum possible length.

One thing you can notice about both problems is there is a very natural $n! = n \times (n-1) \times \cdots \times 1$ time algorithm. For both: check all possible permutations of the vertices separately. It turns out that by using dynamic programming, you can improve the runtime of both problems to approximate $O(2^n \times n^2)$. The "main term" here is the 2^n, which is much less than $n!$ for moderate values of n, for example $n = 25$.

For $n = 25$, the number of operations for each algorithm is approximate $25! \approx 1.5 \cdot 10^{25}$, which $2^{25} \times 25^2 \approx 2 \cdot 10^{10}$. The former would take forever to run, but the latter can comfortably run on your personal laptop.

We cover the former problem since we are already talking about DAGs, and leave the latter for you to do as an exercise (the algorithm is nearly identical).

Intuition for DP state. Let's imagine that we are building a topological sort of the vertices in the graph G. Let's say that so far, it consists of vertices $v_1, \ldots, v_k$. What information about $V_1, \ldots, v_k$ do we need to decide if a new vertex v is allowed to come next?

Remember the definition of topological sort: we cannot put v after $v_1, \ldots, v_k$ if there is an edge from v to any of $v_1, \ldots, v_k$. Thus, to decide whether v can come after $v_1, \ldots, v_k$ we only need to know the set $\{v_1, \ldots, v_k\}$, not the ordering itself! This naturally motivates using subsets of vertices as states.

Formal definition of DP state and transitions. For a subset $S \subseteq V$ (where V is the set of n vertices of G), define $\text{dp}[S]$ to be the number of ways to order vertices in S such that they respect the rules of topological ordering: if $u, v \in S$ and $u \to v$ is an edge, then u must come before v in the ordering.

The transition is simple now. To compute $\text{dp}[S]$, test if $v \in S$ can be the last element. Formally,

$$\text{dp}[S] = \sum_{\substack{v \in S \\ \text{no edges from } v \text{ to } S \setminus \{v\}}} \text{dp}[S \setminus \{v\}].$$

The base case is $\text{dp}[\emptyset] = 1$ ($\emptyset$ is the empty set), and it equals 1 because there is 1 way to order 0 elements.

The answer is of course $\text{dp}[V]$.

Runtime. There are 2^n states, one for each subset of n elements. The transition sums over n terms. For each term, we have to decide whether there is an edge from v to $S \setminus \{v\}$ which takes n times. So the overall complexity is $O(2^n \cdot n^2)$.

Implementation details. You might be wondering how to store subsets in an actual implementation. This is simple: each subset corresponds to a binary string, where 0 means to exclude to element and 1 means to include. Every binary string corresponds to an integer via the binary (base 2) expansion. So we can encode each subset as an integer, and removing a vertex v corresponds to subtracting off some power of 2. This is known as a *bitmask DP*.

Faster algorithms? There is strong evidence that both these problems require exponential time algorithms, i.e., runtime at least c^n for some $c > 1$. There has been work on improving 2^n to eg. 1.99^n, but we're unlikely to ever get a polynomial time or truly fast exact algorithm for these problems that work in all cases.

2.3 Edit Distance

Problem: T

he input consists of two strings S and T of length n, costs $c_1, \dots, c_n > 0$, and an integer $x > 0$. We are allowed to do the following operations to S.

- Delete $S[i]$ at cost $c[i]$.
- Insert an arbitrary character anywhere in S at cost x.

Find the smallest possible cost required to transform S into T in time $O(n^2)$.

In the lecture we will primarily discuss the Longest Common Subsequence problem. The DP for edit distance is basically the same – $\text{dp}[i, j]$ represents the minimal cost required to transform the first i characters of S into the first j characters of T using the allowed operations. The answer is then $\text{dp}[n][n]$. Transitions can be done in $O(1)$ time, for an $O(n^2)$ algorithm.

2.4 Longest Common Subsequence

Longest common subsequence is a very classical problem with a dynamic programming solution. It asks for the longest common subsequence shared between two strings/sequences.

Problem: Longest Common Subsequence (LCS)

Given as input two sequences $A = (a_1, \dots, a_n)$ and $B = (b_1, \dots, b_m)$, find the length of the longest sequence which is a subsequence (not necessarily contiguous) of both A and B.

We will present a dynamic programming algorithm for this problem that runs in $O(mn)$ time. Our states will be the following: $\text{LCS}[i, j]$ (for $1 \le i \le n$ and $1 \le j \le m$) will denote the LCS between the prefixes

$$(a_1, \dots, a_i) \quad \text{and} \quad (b_1, \dots, b_j).$$

From these we can easily update the DP formula. If both $a_i = b_j$ and we choose for both to be in the subsequence, then we reduce to the $\text{LCS}[i-1, j-1]$ case. Otherwise, one of a_i and b_j must be omitted from the LCS, and we reduce to the larger of the cases $\mathrm{LCS}[i, j-1]$ and $\mathrm{LCS}[i-1, j]$. We write the formulas explicitly below.

$$\mathrm{LCS}[i,j] = \begin{cases} \max\left\{\mathrm{LCS}[i,j-1], \mathrm{LCS}[i-1,j]\right\} & \text{if } a_i \neq b_j, \\ \max\left\{\mathrm{LCS}[i,j-1], \mathrm{LCS}[i-1,j], 1+\mathrm{LCS}[i-1,j-1]\right\} & \text{if } a_i = b_j. \end{cases}$$

Notes. Interesting, Edit Distance and LCS are two problems where it's conjectured that the DP algorithm is the best possible (and we have "evidence" for this belief).

Pictorial representation. Here is a picture of the DP states, and arrows denoting which states are involved in the DP transitions for which other states.

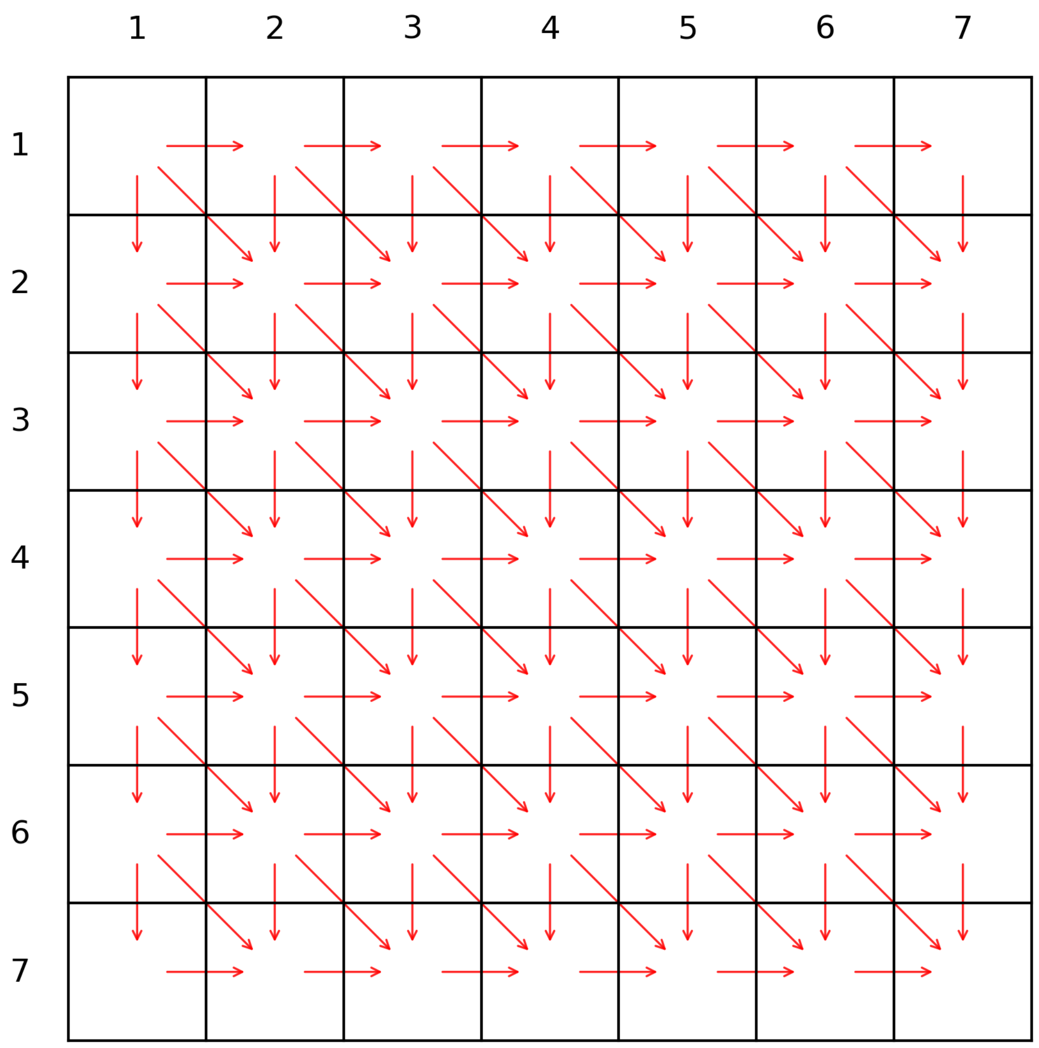

Figure 1: DP states and transitions for the LCS and edit distance problems.

2.4.1 Faster algorithms when m is small.

When $m = n$, there are complexity-theoretic reasons to believe that $O(n^2)$ is about the best runtime an algorithm can achieve for the LCS problem. However, when m is much less than n, better algorithm are known. More generally, if the length L of the LCS is known to be much less than m, n, then better algorithms are known. For simplicity, we present an algorithm whose runtime is $O((n+m^2)\log n)$. In your mind, you should think that $m \approx \sqrt{n}$.

Definition of states. The challenge here is that you cannot afford to even define $O(mn)$ DP states: this is already too large. Instead, we need to heavily use the fact that because the sequence B is length m, then the LCS is also length at most m. This suggests a natural m^2 size DP table: have an entry for each prefix of B and for each LCS length. For $\ell \leq j \leq m$ define $\mathrm{DP}[j,\ell]$ as the smallest index $i \in \{1,\dots,n\}$ such that $\mathrm{LCS}[i,j]=\ell$, or $+\infty$ if no such i exists.

Let's see how to update the DP formula. To find $\mathrm{DP}[j,\ell]$ from previous values, there are two cases to consider. The first case is if b_j is not part of the LCS. Then this gives the state $\mathrm{DP}[j-1,\ell]$. Otherwise, j is part of the LCS. In this case, the length of the prefix of A we need can be calculated as follows. First go to index $\mathrm{DP}[j-1,\ell-1]$ (we have to generate an LCS of length $\ell-1$ using only the first $j-1$ entries in B). Then, we find the first entry in A past this point which equals b_j. Formally,

$$\mathrm{DP}[j,\ell]=\min\left\{\mathrm{DP}[j-1,\ell],\mathrm{first}\big(b_j,\mathrm{DP}[j-1,\ell-1]\big)\right\},$$

where

$$\mathrm{first}(x,i)=\min\left\{i'>i : a_{i'}=x\right\}.$$

We want to fill in this DP table efficiently. Thus, we need to be able to evaluate the function first(x,i) efficiently. In the next paragraph we will show how to preprocess A in $O(n)$ time so that each query to the value of first(x,i) can be answered in $O(\log n)$ time using a binary search. Thus, the overall runtime is $O((n+m^2)\log n)$.

Preprocessing *A*. For each distinct element in A, build a sorted list of its indices in A. This can be done in $O(n)$ total time. When computing first(x,i), go to the list corresponding to x, and binary search to find the first index larger than i. This is the value of first(x,i) by definition. Thus, preprocessing is $O(n)$ time and each query is $O(\log n)$ time.

Pictorial representation. We describe how this relates to the above picture / DP table. In that case, the grid is only $m \times n$ (let's say m rows and n columns). Each row of the DP table is *increasing* and only contains numbers from 1 to m. Thus, each row looks like a bunch of 1's, 2's, ..., m's. The compressed DP state we described above is simply for each row j, the first position in the that row where ℓ appears.