Dynamic Programming II

In the previous lecture, we reviewed Dynamic Programming and saw how it can be applied to problems about sequences and trees. Today, we will extend our understanding of DP by applying it to other classes of problems, like shortest paths, and explore how to speed up dynamic programming implementations with clever tricks like fancier data structures!

Objectives of this lecture

In this lecture, we will:

- Learn about optimizing DPs by **eliminating redundancies** via the all-pairs shortest path problem
- Learn about optimizing DPs **with data structures** via the longest increasing subsequence problem
- Learn about optimizing tree DPs **by adding child information one at a time**

1 All-pairs Shortest Paths (APSP)

Say we want to compute the length of the shortest path between *every* pair of vertices in a weighted graph. This is called the **all-pairs** shortest path problem (APSP). If we use the Bellman-Ford algorithm (recall 15-210), which takes $O(nm)$ time, for all n possible destinations t, this would take time $O(mn^2)$. We will now see a Dynamic-Programming algorithm that runs in time $O(n^3)$, but let's warm up with a simpler but worse algorithm.

1.1 A first attempt in $O(n^4)$

We dealt with the Traveling Salesperson Problem (TSP) just last lecture, which we also solved in terms of substructure over paths. It therefore seems natural to try the same thing for APSP. In our DP solution for TSP, we created paths by extending them by one edge at a time. We also had to remember the current subset of vertices since the goal was to visit every vertex once and only once. In this problem, we no longer have that restriction, so it won't be necessary to store all of the subsets.

Instead, to build a path out of k vertices, all we need is a path of $k-1$ vertices! We don't actually care *which* vertices they are, so we can just parameterize the subproblems by an integer (the number of vertices in the path) rather than a subset of vertices.

Defining the subproblems Based on the above observation that we can build a path from a shorter path plus a new edge, we can define our subproblems as

$$D[u][v][k] = \text{the length of the shortest path from } u \text{ to } v \text{ using } k \text{ vertices}$$

Deriving a recurrence To build a path from u to v with k vertices, we just need to build a path from u to some other vertex, then add v. We can try every possible intermediate vertex to ensure that we definitely get the right one, which gives us a recurrence that looks like

$$D[u][v][k] = \min_{v' \in V} \big(D[u][v'][k-1] + w(v', v) \big)$$

For simplicity, assume that if $(v', v) \notin E$, then $w(v', v) = \infty$. Our base case will be when there is just a single vertex v in the path, in which case the distance from v to itself is zero.

$$D[v][v][1] = 0.$$

Otherwise for $u \neq v$, we want $D[u][v][1] = \infty$ (it is impossible to have a path with one vertex that goes from u to v when $u \neq v$). After evaluating D for all u, v, k where $1 \leq k \leq n$, the length of the shortest path from u to v is given by the minimum of $D[u][v][k]$ for all $1 \leq k \leq n$.

Analysis We have $O(n^3)$ subproblems and each of them takes $O(n)$ time to evaluate, so this takes $O(n^4)$ time. Actually, we can be a little bit more clever in a similar way to which we analyze tree DP algorithms. Note that we only have to check $v' \in V$ for v' that have an outgoing edge pointing to v, i.e., such that $(v', v) \in E$, so the total amount of work spent evaluating the minimum over all v for any fixed value of u and k is $O(m)$, so the total work is at most $O(n^2 m)$, which matches the strategy of repeatedly running Bellman-Ford.

If we think about it carefully, perhaps this isn't so surprising since Bellman-Ford also builds paths by repeatedly adding one edge at a time starting from a single source node, so this algorithm is kind of doing the same thing as running Bellman-Ford simultaneously from every possible start vertex! (We just reinvented Bellman-Ford, oops).

1.2 An improved version in $O(n^3)$

The nice thing about paths is that there are lots of ways of breaking them up into pieces. The previous strategy was to just add a single edge at a time, but that seems inefficient because every time we want to add an edge, we have to look at every possible second-last vertex.

Another way to break paths up is to chop them into two paths! A path from u to v can be broken at any point along the path k into the two paths u to k and k to v. How exactly does this let us decompose the problem into *smaller problems* though? We can not just define our subproblems as the shortest path between u and v and then solve them by trying all intermediate vertices k, because this would assume that we already had all the shortest paths between all u and all possible k to begin with, i.e., the problems aren't guaranteed to be smaller problems.

To make the problems smaller, we need one crucial observation: In a valid shortest path, there is no reason to use the same vertex twice! So, when we decide to break a shortest path u to v into two paths U to k and k to v, we don't need k to occur inside either of those paths! Let's therefore try adding new intermediate vertices one at a time.

Define our subproblems The idea is that we want to build paths out of increasingly larger sets of intermediate vertices. When vertex k is introduced, we can stitch together a path from u to k and k to v to build a path from u to v. Those smaller paths do not need to contain k since there is no point in using k more than once. So, we will use the subproblems

$$D[u][v][k] = \text{length of the shortest path from } u \text{ to } v \text{ using intermediate vertices } \{1,2,\ldots,k\}$$

Deriving a recurrence We need to consider two cases. For the pair u, v, either the shortest path using the intermediate vertices $\{1,2,\ldots,k\}$ goes through k or it does not. If it does not, then the answer is the same as it was before k became an option. If k now gets used, we can *break the path at* k and use the optimal substructure to glue together the two shortest paths divided at k to get a new shortest path. Writing the recurrence using this idea looks like this.

$$D[u][v][k] = \min\{D[u][v][k-1], D[u][k][k-1] + D[k][v][k-1]\}.$$

Our base case will just be

$$D[u][v][0] = \begin{cases} 0 & \text{if } u = v, \\ w(u,v) & \text{if } (u,v) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

Analysis We have $O(n^3)$ subproblems and each of them takes $O(1)$ time to evaluate, so this takes $O(n^3)$ time! Notice that the key improvement here over the previous algorithm was that for each subproblem, we only needed to make a single binary decision: use k or don't use k, which is much more efficient than our earlier algorithm which had to try *every vertex*.

1.3 Optimizing the space: eliminating redundancies

One downside of the algorithm is that it uses a lot of space, $O(n^3)$, which is a factor n larger than the input graph. This is bad if the graph is large. Can we reduce this? There are two ways. First, notice that the subproblems for parameter k only depend on the subproblems with parameter $k-1$. So, we don't actually need to store all $O(n^3)$ subproblems, we can just keep the last set of subproblems and compute bottom-up in increasing order of k.

Here's an even simpler but more subtle way to optimize the algorithm. We can just write:

```
// After each iteration of the outside loop, D[u][v] = length of the
// shortest u->v path that's allowed to use vertices in the set 1..k
for k = 1 to n do
  for u = 1 to n do
    for v = 1 to n do
      D[u][v] = min( D[u][v], D[u][k] + D[k][v] );
```

So what happened here, it looks like we just forgot the k parameter of the DP, right? It turns out that this algorithm is still correct, but now it only uses $O(n^2)$ space because it just keeps a single 2D array of distance estimates. Why does this work? Well, compared to the by-the-book implementation, all this does is allow the possibility that $D[u][k]$ or $D[k][v]$ accounts for vertex k already, but a shortest path won't use vertex k twice, so this doesn't affect the answer! This algorithm is known as the *Floyd-Warshall* algorithm.

Key Idea: Optimize DP by eliminating redundant subproblems

Sometimes our subproblems might not all be necessary to the solve the problem, so if we can eliminate many of them, we will either speed up the algorithm or reduce the amount of space it requires.

2 Advanced Tree DP

In the previous lecture we saw the bread and butter of tree DP, which involves solving the problem for every possible rooted subtree of a tree and then combining the answers to the child problems to form the answer for the root problem. Sometimes this is not quite enough and is not efficient, so let's see a trick to make it even better. The problem arises when we want to solve problems on trees with high degree, but the problem involves making decisions about how much of a quantity to split between the children. Here's an example problem:

Problem: Counting subtrees

Given a rooted tree with n nodes and root r, count the number of distinct subtrees rooted at r with exactly K nodes.

To get the main idea we will first solve the problem with the additional assumption that the tree is a *binary tree*, then we will see how to remove this assumption.

2.1 Solution for binary trees

Under the assumption that the tree is binary, counting the number of possible subtrees rooted at a particular vertex v with exactly k nodes just consists in deciding how to divide the $k-1$ nodes other than v between the two children.

Defining the subproblems For simplicity we will assume that the tree is a full binary tree, i.e., every node is either a leaf or has two children. We will use the subproblems:

$$S(v,k) = \text{the number of subtrees rooted at } v \text{ containing } k \text{ nodes}$$

The solution to the problem is $S(r,K)$.

Deriving a recurrence To solve the subproblems, we just sum over all of the possible ways to divide the $k-1$ remaining nodes between the two children:

$$S(v,k) = \begin{cases} 1 & \text{if } k=0 \text{ or } k=1, \\ 0 & \text{if } k>1 \text{ and if } v \text{ is a leaf} \\ \displaystyle\sum_{k'=0}^{k-1} S(L(v),k') \cdot S(R(v),(k-1)-k') & \text{otherwise} \end{cases}$$

The first base cases are when $k \in \{0,1\}$ since we can make exactly one subtree out of one node (just itself) and one subtree using zero nodes (the empty subtree). Otherwise if $k > 1$ and v is a leaf then we can not possible make a subtree with k nodes since all we have is a single leaf!

Analysis We have $O(nK)$ subproblems and each takes $O(K)$ time, so the cost is $O(nK^2)$.

2.2 Generalization to high-degree trees

How generalizable is this solution to high-degree trees? Well, it works, but its going to be slow. For a node with two children, there were $O(K)$ ways to divide the subtree nodes between them. For a node with three children, there would be $O(K^2)$ ways to do it, for four children, $O(K^3)$ and so on... This is exponential in K, so trying all possibilities is off the table.

However, there is a lot of redundancy in these possibilities. If I decide to put k' out of k nodes on the first child, then there are $k-1-k'$ remaining nodes to distribute between the remaining children. The fact that I put k' nodes on the first child does not affect the others at all. So we can eliminate the redundancy by further parameterizing the subproblems by which children we are considering, and just adding in one child at a time!

Defining the subproblems For simplicity we will assume that the tree is a full binary tree, i.e., every node is either a leaf or has two children. We will use the subproblems:

$$S(v,k,i) = \begin{cases} \text{the number of subtrees rooted at } v \text{ containing } k \\ \text{nodes using only the } i^{\text{th}} \text{ child and after of } v \end{cases}$$

The solution to the problem is $S(r,K,0)$.

Deriving a recurrence To solve the subproblems, we sum over all of the possible ways to divide the k nodes between the current child and the root with the rest of the children:

$$S(v,k,i) = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = 1, \\ 0 & \text{if } k > 1 \text{ and if } i = \deg(v) \\ \displaystyle\sum_{k'=0}^{k-1} S(C(v,i),k',0) \cdot S(v,k-k',i+1) & \text{otherwise} \end{cases}$$

For notation, we have used $C(v,i)$ to mean the i^{th} child of v, and $\deg(v)$ to be the degree (number of children) of v. The base cases are similar to the binary case, we can make only one tree with a single node or no nodes. Note that $S(v,k,\deg(v))$ refers to a subproblem in which we have ran out of children of v to consider, i.e., we are considering v alone. If v is a leaf then $\deg(v) = 0$ and all the subproblems for v will fall into this or the first case.

Note the subtle difference between this and the binary version where the recursion divides the k nodes into k' and $k-k'$ instead of $(k-1)-k'$. This is because in the binary case, we used that one on v and then split the remaining $k-1$ nodes between the children, but in this recurrence, the second subproblem $S(v,\ldots)$ *still includes the root*, so we do not lose one node anymore.

Analysis How many subproblems do we have now? There are n choices for v, and n choices for i, and K choices for K, so our new cost should be $O(n^2K^2)$, right? No! This is tree DP of course, and there aren't very many possible values of i for every single v! In particular, note that each (v,i) **pair** maps one-to-one with a specific child vertex, so there at most n of them in total. This means that the number of subproblems is actually still at most nK and hence the cost is still $O(nK^2)$, so we just got to generalize to non-binary trees *for free*! It does not actually cost any more than the binary case.

3 Longest Increasing Subsequence

Our next problem is the "longest increasing subsequence" (LIS) problem, which has an $O(n^2)$ solution, but can then be improved with some clever optimizations!

Problem: Longest Increasing Subsequence

Given a sequence of comparable elements $a_1, a_2, \ldots, a_n$, an increasing subsequence is a subsequence $a_{i_1}, a_{i_2}, ..., a_{i_{k-1}}, a_{i_k}$ $(i_1 < i_2 < \ldots i_k)$ such that

$$a_{i_1} < a_{i_2} < \ldots < a_{i_{k-1}} < a_{i_k}.$$

A longest increasing subsequence is an increasing subsequence such that no other increasing subsequence is longer.

Find some optimal substructure Given a sequence $a_1, \ldots, a_n$ and its LIS $a_{i_1}, \ldots, a_{i_{k-1}}, a_{i_k}$, what can we say about $a_{i_1}, \ldots, a_{i_k-1}$? Since $a_{i_1}, \ldots, a_{i_k}$ is an LIS, it must be the case that $a_{i_1}, \ldots, a_{i_{k-1}}$ is an LIS of $a_1, \ldots a_{i_k}$ such that $a_{i_{k-1}} < a_{i_k}$. Alternatively, it is also an LIS that ends at (and contains) $a_{i_{k-1}}$. This suggests a set of subproblems.

Define our subproblems Lets define our subproblems to be

$$\text{LIS}[i] = \text{the length of a longest increasing subsequence of } a_1, \ldots a_i \text{ that contains } a_i$$

Note that the answer to the original problem **is not** necessarily LIS$[n]$ since the answer might not contain a_n, so the actual answer is

$$\text{answer} = \max_{1 \le i \le n} \text{LIS}[i]$$

Deriving a recurrence Since LIS$[i]$ ends a subsequence with element i, the previous element must be anything a_j before i such that $a_j < a_i$, so we can try all possibilities and take the best one

$$\text{LIS}[i] = \begin{cases} 0 & \text{if } i = 0, \\ 1 + \max\limits_{\substack{0 \le j < i \\ a_j < a_i}} LIS[j] & \text{otherwise.} \end{cases}$$

Analysis We have $O(n)$ subproblems and each one takes $O(n)$ time to evaluate, so we can evaluate this DP in $O(n^2)$ time. Is this a good solution or can we do better?

3.1 Optimizing the runtime: better data structures

The by-the-definition implementation of the recurrence for LIS gives an $O(n^2)$ algorithm, but sometimes we can speed up DP algorithms by solving the recurrence more cleverly. Specifically in this case, the recurrence is computing a **minimum over a range**, which sounds like something we know how to do faster than $O(n)$...

How about we try to apply a range query data structure (a SegTree) to this problem! Initially, its not clear why this would work, because although we are doing a range query over $1 \le j < i$, we have to account for the the constraint that $a_j < a_i$, so we can not simply do a range query over the values of LIS$[1 \ldots (i-1)]$ or this might include larger elements.

So here's an idea... Let's solve the subproblems *in order* by the value of the final element, instead of just left-to-right by order of i. That way, when we solve a particular subproblem corresponding to a particular final element, we will have only processed the subproblems corresponding to all smaller elements, which are all legal to append the next larger element to!

```
function LIS(a : list⟨int⟩) -> int = {
  sortedByVal := sorted list of (value, index) pairs
  // SegTree is endowed with the RangeMax operation
  LIS := SegTree(array⟨int⟩(size(a), 0))
  for val, index in sortedByVal do {
    answer := LIS.RangeMax(0, index) + 1   // Solve the subproblem
    LIS.Assign(index, answer)              // Store the subproblem
  }
  return LIS.RangeMax(0, size(a))
}
```

This optimized algorithm performs two SegTree operations per iteration, and it has to sort the input, so in total it takes $O(n \log n)$ time.

Key Idea: Speed up DP with data structures

If your DP recurrence involves computing a minimum, or a sum, or searching for something specific, you can sometimes speed it up by storing the results in a data structure other than a plain array (e.g., a SegTree or BST).

Exercises: Dynamic Programming II

Problem 1. Provide a formal proof by induction that the Floyd-Warshall algorithm gives a correct answer. The comment in the pseudocode is the inductive hypothesis on which you should use induction on k.

Problem 2. Can you find a greedy algorithm that matches the $O(n \log n)$ performance of the LIS algorithm above?

Problem 3. Write a different implementation of the LIS algorithm that still uses a SegTree but loops over i and uses range queries on j instead (the opposite of the solution above).