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 graphs, and explore how to speed up dynamic programming implementations with clever tricks like data structures and matrices!

Objectives of this lecture

In this lecture, we will:

- Learn about *subset DP* via the traveling salesperson problem
- 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 DPs with matrices via counting paths in a graph

1 Traveling Salesperson Problem (TSP)

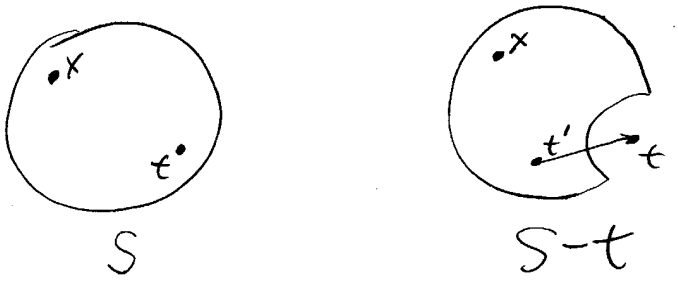
The NP-hard *Traveling Salesperson Problem* (TSP) asks to find the shortest route that visits *all* vertices in a graph exactly once and returns to the start.[1] We assume that the graph is complete (there is a directed edge between every pair of vertices in both directions) and that the weight of the edge (u, v) is denoted by $w(u, v)$. This is convenient since it means a solution is really just a permutation.

Since the problem is NP-hard, we don't expect to get a polynomial-time algorithm, but perhaps dynamic programming can still help get something better than brute force. Specifically, the naive algorithm for the TSP is just to run brute-force over all $n!$ permutations of the n vertices and to compute the cost of each, choosing the shortest. We're going to use Dynamic Programming to reduce this to $O(n^2 2^n)$. This is still exponential time, but it's not as brutish as the naive algorithm. As usual, let's first just worry about computing the *cost* of the optimal solution, and then we'll later be able to recover the path.

[1] Note that under this definition, it doesn't matter which vertex we select as the start. The *Traveling Salesperson Path Problem* is the same thing but does not require returning to the start. Both problems are NP-hard.

Step 1: Find some optimal substructure This one harder than the previous examples, so we might have to try a couple of times to get it right. Suppose we want to make a tour of some subset of nodes S. Can we relate the cost of an optimal tour to a smaller version of the problem? Its not clear that we can. In particular, suppose we call out a particular vertex t, and then ask whether it is possible to relate the cost of the optimal tour of $S-\{t\}$ and S. It doesn't seem so, because its not clear how we would splice t into the tour formed by $S-\{t\}$ without additional information.

So, lets be more specific and add some additional information. Adding a vertex into a cycle seems difficult to do, but adding a vertex onto a path seems like something we could work with. So lets fix an arbitrary starting vertex x, and now consider the cheapest path that starts at x, visits all of the vertices in S and ends at a specific vertex t. Can we find any substructure in this much more specific object? Well, yes, we know that the optimal path from x to t must have some second-last vertex t', and the path from x to t' must be an optimal such path using the vertices $S-\{t\}$. Lets use this for our subproblems.



Step 2: Define our subproblems Based on the above, lets make our subproblems

$C(S,t) =$ The minimum-cost path starting at x and ending at t going through all vertices in S

What is the solution to our original problem? Is it one of the subproblems? Actually the answer is no this time. But we can figure it out by combining a handful of the subproblems. Specifically, we want to form a tour (a cycle) using a path that starts at x. So we can just try every other vertex in the graph t, and make a path from x to t then back to x again to complete the cycle. So the answer, once we solve the DP will be

$$\text{answer} = \min_{t \in (V-\{x\})} (C(V,t) + w(t,x))$$

Step 3: Deriving a recurrence Using the substructure we described above, the idea that will power the recurrence is that to get a path that goes from x to t, we want a path that goes from x to t' plus an edge from t' to t. Which t' will be the best? We can't know for sure, so we should just try all of them and take the best one. We also need a base case. We can't use an empty path as our base case since we assume a starting vertex x and an ending vertex t for every subproblem, so lets use a path of two vertices as our base case. This gives us the recurrence

Algorithm: Dynamic programming recurrence for TSP

$$C(S,t) = \begin{cases} w(x,t) & \text{if } S = \{x,t\}, \\ \min\limits_{\substack{t' \in S \\ t' \notin \{x,t\}}} C(S - \{t\}, t') + w(t',t) & \text{otherwise.} \end{cases}$$

Step 4: Analysis The parameter space for $C(S,t)$ is 2^{n-1} (the number of subsets S considered) times n (the number of choices for t). For each recursive call we do $O(n)$ work inside the call to try all previous vertices t', for a total of $O(n^2 2^n)$ time. This is assuming we can lookup a set (e.g, $S - \{t\}$) in constant time.

Remark: Efficiently representing the set

Since this algorithm is too slow for large values of n, we usually assume that n is small, and so a highly efficient way to store the set S is as a single integer, where you use the individual bits of the integer to indicate which vertices are in or not in the set. This makes it easy to store and lookup the subproblem solutions because an integer is a much better key for an array or hashtable than a set!

This technique is sometimes called "Subset DP". These ideas apply in many cases to reduce a factorial running to time to a regular exponential running time.

2 All-pairs Shortest Paths

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. 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 $O(n^3)$.

Step 1: Find some optimal substructure In the first lecture, we used shortest paths as an example of substructure. A shortest path is always made of combining two smaller shortest paths.

Step 2: Define our subproblems Although we can see the substructure, its not clear how to actually break up the paths, because there are lots of options. One way would be to do so by *length*, i.e., consider paths of length k in terms of paths of length $k-1$ plus another edge, but this would turn out to be quite inefficient. Instead, a more efficient idea is to consider *which vertices we are allowed to use*. The idea is that instead of increasing the number of edges in the path, we'll increase the set of vertices we allow as intermediate nodes in the path. So, lets try the subproblems

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

Step 3: 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

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

Step 4: Analysis We have $O(n^3)$ subproblems and each of them takes $O(1)$ time to evaluate, so this takes $O(n^3)$ time.

2.1 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 is one straightforward way to reduce it, and then a more subtle way to reduce it even further. 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. I claim that 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!

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.

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.

Step 1: 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.

Step 2: 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

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

Step 3: 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

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

Step 4: 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-book 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 $1 \leq j < i$, 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. Here's an idea. Let's store the values of LIS in a different order! Rather than storing LIS[1], LIS[2], ..., as usual, we can store them in order of the values of a_i, so that we can actually do a range query over all values less thana_i. To do this, we first sort $(a_i)_i$ which takes $O(n \log n)$ with any fast sorting algorithm, then use the rank of a_i as the position of LIS$[i]$. In pseudocode, the optimized version might therefore look something like:

```
LIS(a : list⟨int⟩) -> int {
  n := size(a)
  b := sorted(a)
  LIS : SegTree = (array⟨int⟩(n+1, 0)) // SegTree is endowed with the RangeMax operation
  for i in 0 to n - 1 do {
    rank = binary_search(b, a[i])      // the rank is a[i]'s position in the sorted list
    LIS.Assign(rank, 1 + LIS.RangeMax(0, rank)) // Note: assumes ranks are 1-based
  }
  return LIS.RangeMax(0, n+1)
}
```

This optimized algorithm performs one binary search and 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.

4 Counting Paths in a Graph

Optional content — Not required knowledge for the exams

For this next problem lets consider a directed unweighted graph on n vertices. Our goal is to compute, for every pair of vertices u, v, the number of directed walks[2] of length k that start at u and end at v. A walk is a path between u and v that might repeat vertices or edges.

Step 1: Find some optimal substructure A walk of length k between u and v is just a walk of length $k-1$ from u to some neighbor of v with one additional edge.

Step 2: Define our subproblems Lets define our subproblems to be

$$W[u][v][k] = \text{the number of walks of length } k \text{ between } u \text{ and } v$$

Step 3: Deriving a recurrence To find the number of walks of length k between u and v, we have to look at walks of length $k-1$ and then add another edge. This means we are interested in vertices that are one away from u or v. To avoid overcounting, we should just pick one of them, so lets consider vertices adjacent to v. Our goal is then to sum over all such vertices.

$$W[u][v][k] = \begin{cases} 1_{u=v} & \text{if } k = 0, \\ \displaystyle\sum_{v \in \text{adj}[x]} W[u][x][k-1] & \text{otherwise} \end{cases}$$

Here, I have used the notation $1_{u=v}$ as an indicator function, i.e. it is equal to 1 if $u = v$, otherwise it is equal to zero, and adj$[x]$ to mean the set of vertices adjacent to x.

Step 4: Analysis Computing this recurrence as written, we have $O(n^2 k)$ subproblems and each takes $O(n)$ time to evaluate, so the total runtime is $O(n^3 k)$. Can we do better?

4.1 Optimizing the runtime: matrices

Although it might not look it from the way it is written, the recurrence above is actually of a useful form. Lets rewrite it a bit to make it clearer. I'm going to move the k's to subscripts, and rewrite the sum using an indicator function. The reason will hopefully become clear soon.

$$W_k[u][v] = \sum_{x \in V} W_{k-1}[u][x] A[x][v], \qquad W_0 = I$$

Here, A is the adjacency matrix of the graph (it contains 1 at $[u][v]$ if there is an edge from u to v, otherwise it contains zero), and I is the identity matrix ($n \times n$ in this case). Does this formula look familiar? Its actually matrix multiplication!! Specifically, it says that

$$W_k = W_{k-1} A,$$

[2] If k is large, then the number of such paths could end up being quite a big number, which could mess with the analysis. Lets just assume that we can still do arithmetic in constant time. One way to achieve this is to compute everything mod p

where A is the adjacency matrix of the graph, and $W_0 = I$. Which means by extension that

$$W_k = A^k.$$

So we can compute W_k by computing a power of a matrix. Using exponentiation by squaring, we can therefore compute W_k in $O(n^3 \log k)$ time, since matrix multiplication takes $O(n^3)$ time, and we have to square $O(\log k)$ times.

Key Idea: Speeding up DP with matrices

If your dynamic programming recurrence is a low-order linear recurrence relation, you can sometimes speed it up by writing it as a matrix equation and using matrix exponentiation!

4.2 Making the matrix method even faster

There are a couple of ways to make the matrix method even faster.

Option 1: faster multiplication Above, we saw how to compute the number of walks in a graph in $O(n^3 \log k)$ time using matrix multiplication. This was assuming that matrix multiplication takes $O(n^3)$ time, but there are actually faster algorithms out there (asymptotically at least). Currently, matrix multiplication takes $O(n^\omega)$ where $\omega = 2.37...$. It is not yet known whether there exists an $O(n^{2+o(1)})$-time algorithm! So, just plugging in a "faster" matrix multiplication, we get a runtime of

$$O(n^\omega \log k).$$

Option 2: diagonalization Another way to do better, if we remember our linear algebra class, is by *diagonalizing* the matrix into the form $A = PDP^{-1}$, where D is a diagonal matrix. Then our equation becomes

$$W_k = PD^kP^{-1},$$

which means we only have to compute powers of the diagonal elements of the matrix, rather than powers of an entire $n \times n$ matrix, which is much cheaper! Since it just takes $\log k$ time to compute each power, and there are n elements on the diagonal of D, this takes $O(n \log k)$ time plus the time required to compute the diagonalization, which is $O(n^3)$, for a runtime of

$$O(n^3 + n \log k).$$

Exercises: Dynamic Programming II

Problem 1. Given the results of $C(S,t)$ for a TSP problem, explain how to find the actual sequence of vertices that make up the tour.

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