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.

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 *subset DP* and use it to solve the *traveling salesperson problem*

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

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

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

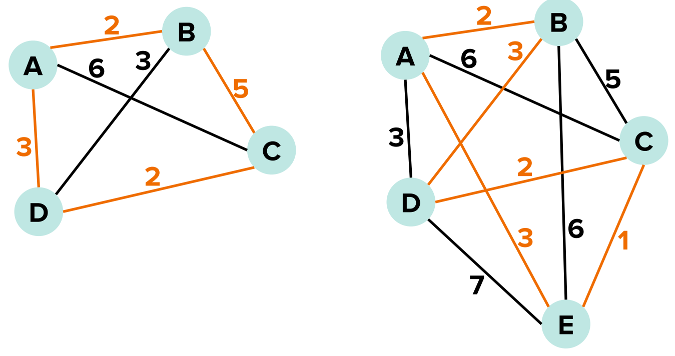
Sometimes our subproblems might not all be necessary to the solve the problem, so if we can eliminate many of them, we will either speed up the algorithm or reduce the amount of space it requires.

2 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 of the vertices.*

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.

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 $S - \{t\}$. In fact, we can even show an example which demonstrates that the optimal tour of $S - \{t\}$ may not actually have all that much in common with the optimal tour of S:

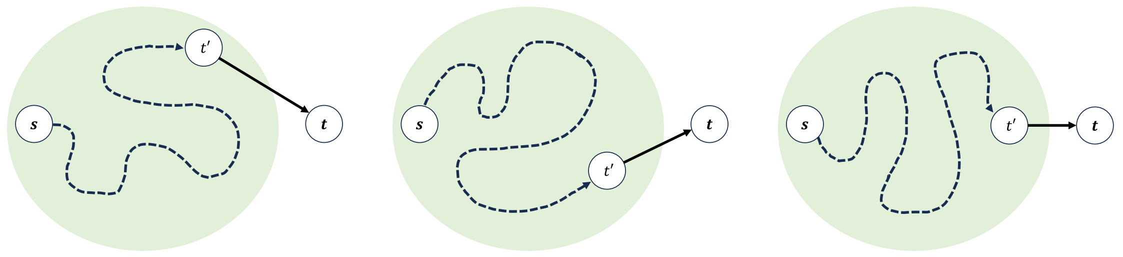


[1]Note that under this definition, it doesn't matter which vertex we select as the start. The *Traveling Salesperson Path Problem* is the same thing but does not require returning to the start. Both problems are NP-hard.

On the left is an optimal tour of a graph with four vertices. After adding a fifth vertex, we can see that the new optimal tour does not actually contain the old one as a subset, so there does not appear to be any *optimal substructure* here and we can not use these as our subproblems!

When faced with such an issue, it can quite often be resolved by adding more information to the subproblems. Adding a vertex into a cycle seems difficult to do because we don't know where to put it and how it might affect everything after it, but adding a vertex *onto the end of a path* sounds simpler, so lets try that. We will 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**. The last part is essentially the "additional information" that we added to make the subproblems more specific. If we just considered any paths consisting of the vertices in S but without fixing the final vertex, we would run into the same problem as before.

Can we find any substructure in this much more specific object? The path now has a final vertex, which means it also has a second-last vertex! Specifically, 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 $S - \{t\}$ over all possible choices of 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

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

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 (wait, is that reasonable?).

Remark: Efficiently representing the set

How would we actually represent the sets used in the dynamic programming subproblems? Since each set has up to n elements, does each set require $\Theta(n)$ space to store and lookup? That would make the runtime worse by a factor of $\Theta(n)$.

Luckily, there's a trick to work around this! Since our algorithm has to store exponentially many such subsets, specifically 2^{n-1} of them, we have to assume that we have enough space to actually store them. In the word RAM model, we assume that we can index memory up to 2^w words (i.e., we have to be able to write the indices in w bits), which means that to even store this many subproblems, we are forced to assume that $2^{n-1} \leq 2^w$, or $n - 1 \leq w$. This means that n can not be very large, and that we can actually write all of the integers from 0 to 2^{n-1} in a **single word each**. This enables us to represent the subsets as **bitmasks**. Specifically, we represent each subset as single word-size integer such that the i^{th} bit of the integer is 1 if and only if the i^{th} vertex is in the subset. This lets us represent each subset in $\Theta(1)$ space!

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.

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.