TSP

The NP-hard *Traveling Salesman Problem* (TSP) asks to find the shortest route that visits *all* vertices in the graph. To be precise, the TSP is the shortest tour that visits all vertices and returns back to the start.[2] Since the problem is NP-hard, we don't expect that Dynamic Programming will give us a polynomial-time algorithm, but perhaps it can still help.

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 can reduce this to $(n-1)!$ permutations by always using the same start vertex, but we still pay $\Theta(n)$ to compute the cost of each permutation, so the overall running time is $O(n!)$.) We're going to use Dynamic Programming to reduce this to "only" $O(n^2 2^n)$.

Any ideas? As usual, let's first just worry about computing the *cost* of the optimal solution, and then we'll later be able to add in some hooks to recover the path. Also, let's work with the shortest-path metric where we've already computed all-pairs-shortest paths (so we can view our graph as a complete graph with weights between any two vertices representing the shortest path between them). This is conveninent since it means a solution is really just a permutation. Finally, let's fix some start vertex s.

Now, here is one fact we can use. Suppose someone told you what the initial part of the solution should look like and we want to use this to figure out the rest. Then really all we need to know about it for the purpose of completing it into a tour is the *set* of vertices visited in this initial segment and the *last* vertex t visited in the set. We don't really need the whole ordering of the initial segment. This means there are "only" $n2^n$ subproblems (one for every set of vertices and ending vertex t in the set). Furthermore, we can compute the optimal solution to a subproblem in time $O(n)$ given solutions to smaller subproblems (just look at all possible vertices t' in the set we could have been at right before going to t and take the one that minimizes the cost so far (stored in our lookup table) plus the distance from t' to t).

Here is a top-down way of thinking about it: if we were writing a recursive piece of code to solve this problem, then we would have a bit-vector saying which vertices have been visited so far and a variable saying where we are now, and then we would mark the current location as visited and recursively call on all possible (i.e., not yet visited) next places we might go to. Naively this would take time $\Omega(n!)$. However, by storing the results of our computations we can use the fact that there are "only" $n2^n$ possible (bit-vector, current-location) pairs, and so only that many *different* recursive calls made. For each recursive call we do $O(n)$ work inside the call, for a total of $O(n^2 2^n)$ time.

The last thing is we just need to recompute the paths, but this is easy to do from the computations stored in the same way as we did for shortest paths.

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