Lecture 7: Dynamic programming (contd.)

Plan/outline

We have seen a few examples of dynamic programming. The overall strategy has been to (a) first write down a recursive formulation, (b) identify the sub-problems, "parametrize" them appropriately, and (c) store solutions to the sub-problems, so as to avoid possible recomputation.

Today we will see a more involved example, in which parametrizing the sub-problems turns out to be somewhat more tricky.

The traveling salesman problem (TSP)

The "TSP" or traveling salesman problem is one of the most well-studied problems in the area of combinatorial optimization (and related fields like operations research). The problem is simple to state:

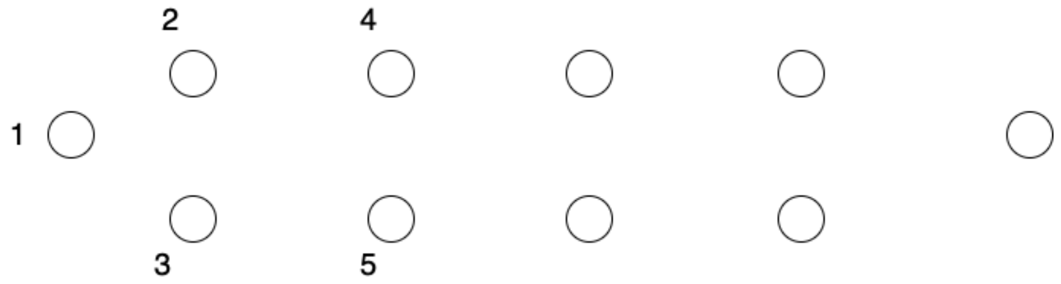
Problem. Suppose we have n cities, and suppose that the distance between cities i and j is denoted by d_{ij}. Consider a salesman who starts at city 1, and whose goal is to visit every city exactly once and return to his home (city 1). The problem is now to minimize the total distance traveled by the salesman.

"Baseline" approaches

Brute force. naive approach is to consider every possible order of visiting the cities. Of course, the time complexity will be large -- something like $O(n!)$. [Because there are $(n-1)!$ possible orders, and for each one, we add up the costs -- which takes n steps.]

Greedy solution. another natural idea is to use the greedy algorithm: suppose that at every step, the salesman visits the closest city that is still unvisited, and returns home once all the cities have been visited. This algorithm is polynomial time (the basic implementation takes quadratic time, in fact). The problem with the greedy solution is that it is not always optimal (this is typical of greedy algorithms, as we will see).

A simple example that illustrates this (on the plane) is as follows.



The vertices on the top are numbered 2, 4, 6, ..., 2ℓ, and the ones at the bottom are numbered $3, 5, \dots, 2\ell+1$ and the last vertex is $2\ell+2$.

The salesman will travel in the order: 1-2-3-5-4-6-... The optimal ordering is clearly 1-3-5-...-$(2\ell+1)$-(2\ell)- ... 2-1. The greedy tour is strictly worse because it involves a lot of up-and-down movement.

Solution via dynamic programming

How can we write a recursive formulation of the problem?

If one were to think of writing down a recursive procedure, what should the input parameters to the procedure be? Imagine a procedure that checks all paths.

It should certainly know which vertices have been visited and which have not (so far). Also, it needs to know the current location. Intuitively, it also needs to know where to end up eventually, but we know that the salesman has to return to vertex 1.

Thus, let us attempt to write a procedure that takes as input two quantities: a start vertex u, and a subset of unvisited vertices S.

```
procedure Tour(start vertex u, unvisited set S):
  if S is empty, return d(u, 1) // no unvisited vertices; tour needs to return to vertex 1
  for all v in S, do:
    candidate_tour_length = d(u, v) + Tour(v, S \ {v})
  return the least value of candidate_tour_lengths found in the process
```

We finally call the procedure Tour with start vertex 1 and $S = \{2, 3, \dots, n\}$. The procedure always terminates, because the size of the S in the recursive call is one less than the size of S at the start.

The correctness of the procedure is easy to see. If we unwind the recursion, we visit every vertex precisely once (because once visited, it is removed from the set S). Further, we will consider every possible ordering of the vertices, and thus the output value will correspond to the optimal solution.

Storing answers. In this case, the sub-problems are defined by the pair (u, S). We thus maintain solutions in an array of size $n \times 2^n$ (or a hashmap that takes an integer and string). Then, in the procedure above, each time we make a recursive call, we check the array to see if the solution has already been computed. If so, we simply look up the answer, and otherwise we make the recursive call. At the end of the call, we store the answer into the array.

Running time. As we have seen with dynamic programs, the run time can be bounded as: (number of distinct sub-problems) * (time taken per sub-problem). The first term is basically the number of pairs (u, S). We have n possible values of u and 2^n possible values for S (because there are 2^n subsets of the set of n vertices).

The time taken per sub-problem is $|S| \le n$, because we iterate over all the elements of S and compute one quantity for each. Thus the overall running time is $O(n^2 2^n)$.

Comparing with the baseline. For starters, it is not obvious why $n^2 2^n$ is a reasonable running time. After all, it is exponential in n. Also, the space usage is $n \cdot 2^n$ --- also exponential. But the key point is that these times are still much smaller than $n!$, when n is large. For example, when $n = 25$, a running time of $n^2 2^n$ is is feasible on most modern computers (in a few minutes), but $n!$ would take well over a lifetime.

The other thing to compare is the memory. The naive algorithm had only a polynomial (linear) memory footprint, as we only needed to store the permutation itself, while the DP solution requires 2^n memory. This is the price one needs to pay for the improved running time.

Understanding such *space vs time tradeoffs* is one of the fundamental directions in computational complexity theory. The last few years have seen much significant progress on proving that such tradeoffs are inherent (cannot be overcome via more clever algorithms), under certain complexity theoretic assumptions (the interested reader can look for papers with keywords "SETH based hardness dynamic programming").