Traveling Salesperson

Hello, and welcome to competitive programming!

In this video we will talk about the traveling salesperson problem and show how to solve it using the bitmask technique and dynamic programming.

Objectives

You will want to know how to use the bitmask technique to set up a dynamic programming solution to the TSP problem. There is a brute force technique also that may be useful since it is easier to program. For this reason you should know the limit of each technique.

TSP

The Problem

The problem is a famous one; you are given a list of cities you need to visit, each of them one time, and when you are done you must return to the start city. There is a cost for the path between each city.

This solution is a Hamiltonian cycle but with one extra constraint: we want to minimize the cost of the cycle.

It turns out that this is NP hard. One thing you could do is fix one city as the canonical start city, and then check all the permutations of 2 through (n) to find the

minimum one. This will be order n-factorial. In a contest setting, where you may have one second or so, (n=11) is about the best you can hope for.

The TSP algorithm

Fortunately, we can use dynamic programming, and reduce the time complexity to ({\cal O}(2^n)). This is still horrible, but it's better than n-factorial, and lets us check up to (n=16).

We will use a bitmask to keep track of which cities we have already visited. The DP array will be (2^16) bits long. The TSP function takes a cost array and a variable mx as references. The variable `mx` has a bit set for each city; this enables us to check quickly if we are done.

Finally we have input `cur` for our current city, and `state` for the bitmask.

We always start from the first city, so we loop from bit 2 to n to find cities we haven't tried yet. For each unvisited city, we try that direction and then pick the minimum cost as the cost for the current state.

Finally we return the result and memoize it for future calls.

Setup

Here's the code to set things up.

We first read (n), the number of cities, and allocate a 2D array `adj`. We use a double loop to read in the costs.

Finally, we compute `mx` by shifting a 1 (n) steps to the left and then subtracting 1.