Programming Team Lecture: DP Algorithm for Traveling Salesman Problem

One version of the traveling salesman problem is as follows:

Given a graph of n vertices, determine the minimum cost path to start at a given vertex and travel to each other vertex exactly once, returning to the starting vertex.

In some versions, the starting and ending points are different and fixed, and all other points have to be visited exactly once from start to end.

A standard way to solve these problems is to try all possible orders of visiting the n points, which results in a runtime of O(n!). (If evaluation of a path takes O(n) time, then it would be O(n x n!).) this limits the input size to n = 10 on current (2010ish) machines.

We can solve this problems for a slightly larger input size by realizing that some of the paths we try are "redundant". For example, consider the two paths specified below:

1, 5, 2, 8, 4, 6, 3
1, 2, 8, 4, 5, 6, 3

You'll notice that both paths end in the edge 6 → 3 but visit the vertices 2, 4, 5 and 8 in between 1 and 6 in a different order. If 1 → 5 → 2 → 8 → 4 → 6 is less costly than 1 → 2 → 8 → 4 → 5 → 6, then we only want to consider building off the first path and have no need to evaluate the second path.

In particular, for each subset of vertices to visit and a fixed ending vertex, we only need to store the minimum cost for visiting that subset, ending at the fixed end vertex.

Let f(S, k) represent the minimum cost for visiting all the vertices in subset S, ending at vertex k. Using this definition, we could build answers to arbitrary queries of this form. Consider calculating, where we assume 1 is always the start vertex:

f({1, 2, 3, 4, 5, 6, 8}, 3)

We know we need to end in vertex three, so that our previous subset visited must be:

{1, 2, 4, 5, 6, 8}

Now, with this subset, we could have five possible ending vertices: 2, 4, 5, 6, or 8. Thus, if we want to find the answer to our query, we must simply take the minimum answer over trying each of these five possibilities. Formally, we have:

f({1, 2, 3, 4, 5, 6, 8}, 3) = minimum (f({1, 2, 4, 5, 6, 8}, 2) + edge(2 → 3) ,
f({1, 2, 4, 5, 6, 8}, 4) + edge(4 → 3) ,
f({1, 2, 4, 5, 6, 8}, 5) + edge(5 → 3) ,
f({1, 2, 4, 5, 6, 8}, 6) + edge(6 → 3) ,
f({1, 2, 4, 5, 6, 8}, 8) + edge(8 → 3))

Note that all of the function evaluations are smaller problems of the same nature.

Thus, in code, we can either write this recursively with memoization or with dynamic programming. Either way, we need to declare a data structure to store all possible answers to recursive queries. We'll need an array with two dimensions. One dimension tells us the subset we are referring to while the other dimension tells us the last vertex.

The best way to store an arbitrary subset of n vertices is a bitmask of n bits. Thus, our array that stores all of our answers might look like:

```
int[][] dp = new int[1 << n][n];
```

where n stores the number of vertices in our problem. Note that some elements in the array go unused, since they are non-sensical. (You can't visit the subset {1, 3, 5} ending at vertex 4, for example.)

More specifically, for a given integer, if the i^{th} bit is 1, then element i is in the subset, and if the i^{th} bit is 0, then element i is not in the subset. For example, the value $25 = 11001_2$ would represent a subset of elements 0, 3 and 4. (Typically we store subsets of 0-based sets.)

We can check if element k is in the bitmask represented by the integer item in these two ways (and there are probably more):

```
if (((item >> k) & 1) == 0) …
if ((item & (1 << k)) > 0) …
```

We can check if the set a is a subset of the set b as follows:

```
if ((a & b) == a) …
```

We can check if two sets a and b are disjoint as follows:

```
if ((a & b) == 0) …
```

If we know that item k is in the set subset, then we can remove it from the set as follows:

```
int itemsLeft = subset - (1 << k);
```

In short, once we know how to use our bitwise operators, we can easily make manipulations to sets as necessary for algorithms where we are building answers from various subsets.

Note that when building an answer for a particular subset, you only need to look up answers for subsets of that subset. Also, for any bitmask, all of its subsets are strictly less than it. Thus, if we simply write our dynamic programming algorithm to cycle through each subset in numerical order of bitmask, all of our necessary subcases will be previously solved.

On the following page we'll have the rough structure of code to solve a traveling salesman like problem using the bit mask dynamic programming technique. For each specific problem, how the array is initialize may change as well as other small items.

Pseudocode structure of the bitmask dynamic programming solution to Traveling Salesman

```
int[][] dp = new int[1 << n][n];

// Some initialization of dp, possibly.

for (int mask = 1; mask <= (1<<n); mask++) {
  for (int last = 0; last<n; last++) {

    if (((mask >> last) & 1) == 0) continue;

    int prev = mask - (1 << last);
    dp[mask][last] = Integer.MAX_VALUE;

    // v is the last item visited in prev.
    for (int v=0; v<n; v++) {

      if (((prev >> v) & 1) == 0) continue;

      int curScore   = dp[prev][v] + cost(v, last);
      dp[mask][last] = Math.min(dp[mask][last], curScore);
   }


  }
}
```

Explanation of Pseudocode

1. The first two loops go through the total search space.

2. The first if statement screens out impossible sub-cases, where the last vertex isn't in the subset specified by mask.

3. prev represents the subset of vertices visited BEFORE arriving at vertex last.

4. v represents the possible last locations we could visit in visiting each vertex in perv.

5. The inner most if does the same task as the previous if, screening out impossible cases.

6. The relevant score for this path (curScore) is the sum of the best score of visiting everything in prev, ending in vertex v, added to the cost/edge from vertex v to vertex last, which is where we are trying to end up. I denote the edge weight simply by a cost function that the programmer is free to define. (In some questions, an edge isn't given and some cost has to be calculated from one location to the next.)

7. The last line in the inner-most loop simply updates our answer to be the best of all possibilities where we visit all the vertices in prev followed by traveling to vertex last.