CSE 202 NOTES FOR OCTOBER 31, 2002

Dynamic Programming Part II

Recall that in dynamic programming, we follow these steps:

Find backtracking/recursive solution: Typically the simpler the recursive algorithm you start with, the simpler (more likely you are to find) the dynamic programming algorithm.

Identify and characterize the subproblems: Generally this involves looking at how the recursion works out from a decision tree point of view, and then parameterizing the subproblems.

Rewrite recursion in terms of renaming: This generally removes recursion from the problem.

Identify bottom-up order on parameters: This usually consists of properly initializing whatever data structure you may be using with the appropriate base-cases.

Rewrite the recursive algorithm: Initialize any data structures with base cases
for every subproblem, in bottom-up order **do**
*d*o rewritten recursion
end for
return *main problem*

All-Pairs Shortest Paths

Consider a new problem: given a weighted directed graph $G = (V, E, w)$, find the shortest path between each pair of vertices in V (here, "length" means the sum of weights of the edges in the path). There are two variations to this problem: one where all weights in w are positive, and the other where weights in w may be negative. We will solve here only the $w_i > 0$ case, since that constraint has the effect that only simple loops can be considered optimal.

I will abbreviate *Shortest Path Length* as SPL. We could say that $SPL(s,t) = \min\{w[s,t] : \min\{w(s,s') + SPL(s',t)\}, s' \in N(s)\}$, but this is problematic because it will get caught in loops. To take care of this problem, we could keep track of every node that gets visited on a path from s to t and check each time we visit a node that it hasn't been visited before on this path, but that leads to an exponential bookkeeping problem. Instead, we can just keep track of the number of steps in the path: it will never be larger than the number of edges in the graph, because all weights are positive and the paths must be simple.

So let's redefine our function $SPL(s,t)$ and augment it with a length l, which is an upper bound on the number of edges we will allow in our path:

$$SPL(s,t,l) = \begin{cases} w[s,t] \text{ if } l = 1, (s,t) \in E \\ \infty \text{ if } l = 1, (s,t) \notin E \\ \min\left\{SPL(s,t,l-1), \min_{s' \in N(s)}\{SPL(s',t',l-1) + w[s,s']\}\right\} \end{cases}$$

How many subproblems are in this new algorithm? We have l that varies from 1 to n, and we have $s \in V$ and $t \in V$, so n^3 possible subproblems (here n is the size of V). (Note that this $O(n^3)$ is not the amount of work that our algorithm will do, but the number of subproblems there are to solve.)

What is the bottom-up order? Well, in the top-down order, l decreases, so in the bottom-up order, l increases.

The new algorithm written in the form of the template above is:

for $s \in V$ **do**
  for $t \in V$ **do**
    if $(s,t) \in E$ **then**
      $SPL[s][t][1] \leftarrow w[s,t]$
    else
      $SPL[s][t][1] \leftarrow \infty$
    end if
  end for
end for
for $l = 2$ upto n **do**
  for $s \in V$ **do**
    for $t \in V$ **do**

$$SPL[s][t][l] = \begin{cases} w[s,t] \text{ if } l = 1, (s,t) \in E \\ \infty \text{ if } l = 1, (s,t) \notin E \\ \min\left\{SPL[s][t][l-1], \min_{s' \in N(s)} \{SPL[s'][t'][l-1] + w[s,s']\}\right\} \{*\} \end{cases}$$

    end for
  end for
end for

We're really doing $O(n)$ operations on all the neighbors in the innermost loop, which isn't $O(n^2)$ but $O(m)$ ($m = |E|$). Of course, on dense graphs, this is the same thing, but it's more accurate to say $O(m)$. Thus, our time is $O(n^2 m)$.

This is the Bellman-Ford algorithm. Notice that for dense graphs this is quite bad: $O(n^4)$ (as $m \to n^2$).

A Dense Graph Algorithm. We just described a dynamic programming algorithm for the APSP problem, and we arrived at it through the usual approach of finding a backtracking solution and then working out the repetitive subproblem structure. However, just because we did this with one backtracking algorithm doesn't mean we couldn't do something similar with other backtracking algorithms. The decision points in the Bellman-Ford algorithm are nodes, and the options are what the next node in the shortest path will be. We could instead consider the edges $e \in E$ and an option could be whether or not e is in a shortest path, and if so, how it fits in. If e is in the path, then there must be an edge to the left of it and an edge to the right of it, unless it is one of the terminal edges.

We can write the backtracking version of this algorithm, also known as the *Floyd-Warhsall* algorithm, as follows. We use $N_?(x)$ to denote the neighborhood of x, that is, all vertices connected by edges to x. N_i is the set of inbound neighbors, and N_o is the set of outbound neighbors. As usual, we only concern ourselves with the length of the path and leave out the details that return the path itself.

BTFW($G = (V,E)$, $x \in V$, $y \in V$):

$v_n \leftarrow$ some node in G that is not x or y.
$s \leftarrow \text{BTFW}(G - \{v_n\}, x, y)$
for $u \in N_i(v_n)$ **do**
  for $v \in N_o(v_n)$ **do**
    temp $\leftarrow \text{BTFW}(G-\{v_n\}, x, u)+\text{BTFW}(G-\{v_n\}, v, y)+w(u, v_n)+w(v_n, v)$.
    if temp $\leq s$ **then**
      $s \leftarrow$ temp {We know that v_n is in the path, then.}
    end if
  end for
end for
return s

Clearly BTFW is an exponential algorithm. However, we can consider the subproblem structure as we did for the Belman-Ford algorithm. Each invocation of the backtracking algorithm is on a graph G' and two nodes a and b. If we establish an ordering of the vertices (it doesn't really matter how), then we define an ordering on the calls for the graphs G', as long as "some node in G" in the backtracking algorithm is replaced with "the last node in G that is not x or y". In this case, our graph G' is a list of vertices $v_1, v_2, \ldots, v_i$ at iteration i and edges on only these vertices. We can construct a datastructure $D[a, b, i]$ that holds the minimum distance of a path from a to b whose interior nodes come from G_i. In this context, G_i represents the set of allowable vertices for a path to pass through. The dynamic programming version of the Floyd-Warshall algorithm is shown below.

FW[G]
for $u \in V$ **do**
  for $v \in V$ **do**
    $D[u, v, 1] \leftarrow \min\{w(u, v), w(u, v_1) + w(v_1, v)\}$
  end for
end for
for $i = 2, 3 \ldots, N$ **do**
  for $u \in V$ **do**
    for $v \in V$ **do**
      $D[u, v, i] \leftarrow \min\{D[u, v, i-1], D[u, v_i, i-1] + D[v_i, v, i-1]\}$.
    end for
  end for
end for

This is clearly $O(n^3)$. Thus, for dense graphs, Floyd-Warshall APSP is an order of n faster.