CS787: Advanced Algorithms	
Scribe: Evan Driscoll and David Malec	**Lecturer:** Shuchi Chawla
Topic: Dynamic Programming II	**Date:** September 10, 2007

Today's lecture covered two more dynamic programming problems. The first is Sequence Alignment, which attempts to find the distance between two strings. The second is Shortest Path in a graph. We discussed three algorithms for this; one that solves the single-source shortest path, and two that solve the all-pairs shortest path.

3.1 Sequence Alignment

The Sequence Alignment problem is motivated in part by computational biology. One of the goals of scientists in this field is to determine an evolutionary tree of species by examining how close the DNA is between two potential evolutionary relatives. Doing so involves answering questions of the form "how hard would it be for a species with DNA 'AATCAGCTT' to mutate into a species with DNA 'ATCTGCCAT'?"

Formally stated, the Sequence Alignment problem is as follows:

Given: two sequences over some alphabet and costs for addition of a new element to a sequence, deletion of an element, or exchanging one element for another.

Goal: find the minimum cost to transform one sequence into the other.

(There is a disclaimer here. We assume that only one transformation is done at each position. In other words, we assume that 'A' doesn't mutate to 'C' before mutating again to 'T'. This can be assured either by explicit edict or by arranging the costs so that the transformation from 'A' to 'T' is cheaper than the above chain; this is a triangle inequality for the cost function.)

As an example, we will use the following sequences:

$$\begin{aligned} S &= \text{ATCAGCT} \\ T &= \text{TCTGCCA} \end{aligned}$$

Our goal is to find how to reduce this problem into a simpler version of itself.

The main insight into how to produce an algorithm comes when we notice that we can start with the first letter of S (in our example, 'A'). There are three things that we might do:

- We could remove 'A' from the sequence. In this case, we would need to transform 'TCAGCT' into 'TCTGCCA'.
- We could assume that the 'T' at the beginning of sequence T is in some sense "new", so we add a 'T'. We would need to transform 'ATCAGCT' into 'CTGCCA', then prepend the 'T'.
- We could just assume that the 'A' mutated *into* the 'T' that starts the second string, and exchange 'A' for 'T'. We would then need to transform 'TCAGCT' into 'CTGCCA'.

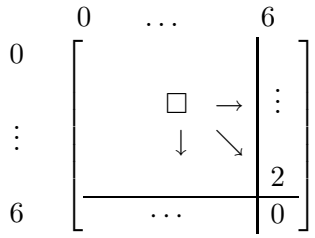
Once we have the above, it becomes a simple matter to produce a recurrence that describes the solution. Given two sequences S and T, we can define a function $D(i_s, j_s, i_t, j_t)$ that describes the distance between the subsequences $S[i_s \dots j_s]$ and $T[i_t \dots j_t]$.

$$D(i_s, j_s, i_t, j_t) = \min \begin{cases} D(i_s + 1, j_s, i_t, j_t) + c(\text{delete } S[i_s]) \\ D(i_s, j_s, i_t + 1, j_t) + c(\text{add } T[i_t]) \\ D(i_s + 1, j_s, i_t + 1, j_t) + c(\text{exchange } S[i_s] \text{ to } T[i_t]) \end{cases}$$

There are two ways to implement the above using dynamic programming. The first approach, top-down, creates a recursive function as above but adds memoization so that if a call is made with the same arguments as a previous invocation, the function merely looks up the previous result instead of recomputing it. The second approach, bottom-up, creates a four dimensional array corresponding to the values of D. This array is then populated in a particular traversal, looking up the recursive values in the table.

However, we can do better at the bottom-up approach than $O(n^4)$. Note that only the starting indices (i_s and i_t) are changed. In other words, we only ever compute $D(\dots)$ with $j_s = |S|$ and $j_t = |T|$. Thus we don't need to store those values in the table, and we only need a two-dimensional array with coordinates i_s, i_t. First, let $m = |S|$ and $n = |T|$ be the length of the strings.

For our example above, this is an 7x7 matrix:



Each element D_{ij} in this matrix denotes the minimum cost of transforming $S[i \dots n]$ to $T[j \dots m]$. Thus the final solution to the problem will be in D_{00}.

When the algorithm is computing the boxed element of the matrix, it need only look at the squares adjacent to it in each of the three directions marked with arrows, taking the minimum of each of the three as adjusted for the cost of the transformation.

The matrix can be traversed in any fashion that ensures that the three squares the box is dependent on are filled in before the boxed square is. (Keeping in mind whether the array is stored in column- or row-major format could help cache performance should the whole array not fit.)

Each square takes constant time to compute, and there are $|S| \cdot |T|$ squares, so the overall running time of the algorithm is $O(|S| \cdot |T|)$.

3.2 Shortest Path and the Bellman-Ford Algorithm

There are two forms of the shortest path algorithm that we will discuss here: single-source multi-target and all-pairs.

We will first look at the single-source problem. The algorithm we will develop to solve this is due to Bellman and Ford. The single-source multi-target shortest path problem is as follows:

Given: Weighted graph $G = (V, E)$ with cost function $c : E \to \mathbb{R}$, and a distinguished vertex $s \in V$.

Goal: Find the shortest path from a source node s to all other vertices.

G must not have cycles of negative weight (or a shortest path wouldn't exist, as it would traverse the cycle an infinite number of times), though it may have edges of negative weight. Section 3.5 discusses this matter further.

Aside: There is a greedy algorithm, Dijkstra's algorithm, that solves single-source shortest path. It expands outward from s, having computed the shortest path to each node within a boundary. However, Dijkstra's algorithm will only produce correct results for graphs that do not have edges with negative weight; our problem statement does not wish to make such an exclusion.

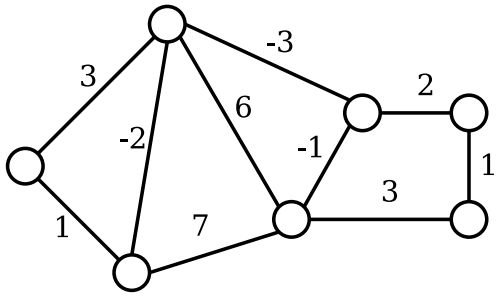
There are two possible approaches to solving this problem with dynamic programming:

- Create a smaller problem by restricting paths to a set number of edges
- Create a smaller problem by restricting the set of nodes that may be visited on intermediate paths

We will start with the former approach, which leads to a single-source solution as well as an all-pairs solution; the latter leads to a solution to the all-pairs problem, and will be discussed later.

The key observation for the first approach comes as a result of noticing that any shortest path in G will have at most $|V| - 1$ edges. Thus if we can answer the question "what's the shortest path from s to each other vertex t that uses at most $|V| - 1$ edges," we will have solved the problem. We can answer this question if we know the neighbors of t and the shortest path to each node that uses at most $|V| - 2$ edges, and so on.

For a concrete illustration, consider the following graph:



Computing the shortest path from s to t with at most 1 edge is easy: if there is an edge from s to

t, that's the shortest path; otherwise, there isn't one. (In the recurrence below, such weights are recorded as ∞.)

Computing the shortest path from s to t with at most 2 edges is not much harder. If u is a neighbor of s and t, there is a path from s to t with two edges; all we have to do is take the minimum sum. If we consider missing edges to have weight ∞ (which we are), then we needn't figure out exactly what us are neighbors of s and can just take the minimum over all nodes adjacent to t. (Or even over all of V.) We must also consider the possibility that the shortest path is still just of length one.

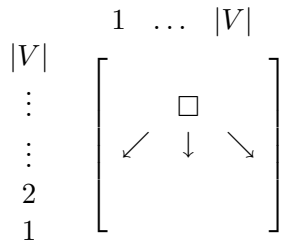
A similar process is repeated for 3 edges; we look at each neighbor u of t, add the weight of going from s to u with at most 2 hops to the weight of going from u to t, then taking the minimum such weight. Again, we must consider the possibility that the best path doesn't change when adding the 3rd edge.

Formally, we define $A(i,k)$ to be the length of the shortest path from the source node to node i that uses at most k edges, or ∞ if such a path does not exist. Then the following recurrence will compute the weights of the shortest paths, where $N(i)$ is the set of nodes adjacent to i and min returns ∞ if $N(i)$ is empty:

$$A(i,k) = \min \begin{cases} \min_{j \in N(i)} \left(A(j,k-1) + c(j,i) \right) \\ A(i,k-1) \end{cases}$$

The first option corresponds to the case where the path described by $A(i,k)$ actually involves all k edges; the second option is if allowing a kth edge doesn't change the solution. The second option can be removed if we consider an implicit, zero-weight loop at each node (that is, $c(v,v) = 0$ for all v).

The bottom-up dynamic programming approach to this problem uses an $|V| \times |V|$ matrix.



Rows represent the maximum number of edges that are allowed for a given instance; the rows are ordered so that the final answer (with a maximum of $|V|$ edges) is at the top of the graph. Columns represent vertices in the graph.

When computing $A(i,k)$, represented by the boxed square, the algorithm looks at all of the boxes in the next row down for nodes that are adjacent to i, as well as i itself. If the degree of the nodes in the graph are not bounded other than by $|V|$, then each square takes $O(|V|)$ time. Since there are $|V|^2$ squares, this immediately gives us an upper bound of $O(|V|^3)$ for the running time. However, we can do better.

Consider a traversal of one entire row in the above matrix. During this pass, each edge will be considered twice: once from node s to node t, and once from t to s. (To see this, note that the

diagonal arrows in the schematic above correspond to edges.) Thus the work done in each *row* is $O(|E|)$. There are $|V|$ rows, so this gives a total bound of $O(|V| \cdot |E|)$ for the algorithm.

This presentation of these shortest-path algorithms only explains how to determine the weight of the shortest path, not the path itself, but it is easy to adapt the algorithm to find the path as well. This is left as an exercise for the reader.

3.3 All Pairs Shortest Path

The all pairs shortest path problem constitutes a natural extension of the single source shortest path problem. Unsurprisingly, the only change required is that rather than fixing a particular start node and looking for shortest paths to all possible end nodes, we instead look for shortest paths between all possible pairs of a start node and end node. Formally, we may define the problem as follows:

Given: A graph $G = (V, E)$. A cost function $c : E \to \mathbb{R}$.

Goal: For every possible pair of nodes $s, t \in V$, find the shortest path from s to t.

Just as the problem itself can be phrased as a simple extension of the single source shortest paths problem, we may construct a solution to it with a simple extension to the Bellman-Ford algorithm we used for that problem. Specifically, we need to add another dimension (corresponding to the start node) to our table. This will change the definition of our recurrence to

$$A[i, j, k] = \min \begin{cases} A[i, j, k-1] \\ \min\limits_{l}\{A[i, l, k-1] + c(l, j)\} \end{cases} .$$

If we take $c(l, l) = 0$ for all $l \in V$, we may write this even more simply as

$$A[i, j, k] = \min_{l}\{A[i, l, k-1] + c(l, j)\}.$$

It is easy to see how this modification will impact the running time of our algorithm — since we have added a dimension of size $|V|$ to our table, our running time will increase from being $O(|V| \cdot |E|)$ to being $O(|V|^2 \cdot |E|)$.

3.4 Time Bound Improvement

If we want to gain some intuition as to how we might go about improving the time bound of our algorithm when applied to dense graphs, it is helpful to alter how we think of the process. Rather than considering a 3-dimensional table A, we will consider a sequence of 2-dimensional tables A_k. Our original table had dimensions for the start node, the end node, and the number of hops; we will use the number of hops to index the sequence A_k, leaving each table in the sequence with

dimensions for the start node and the end node. We may then produce these tables sequentially as

$$\begin{aligned} A_1(i,j) &= c(i,j) \\ A_2(i,j) &= \min_l\{A_1(i,l) + c(l,j)\} \\ A_3(i,j) &= \min_l\{A_2(i,l) + c(l,j)\} \\ &\vdots \end{aligned}$$

If we consider how we define A_1 in the above sequence, we can see that we may write general A_k as

$$A_k(i,j) = \min_l\{A_{k-1}(i,l) + A_1(l,j)\}.$$

Of special interest to us is the close resemblance this definition holds to matrix multiplication, when we write that operation in the form $\sum_l[A_{k-1}(i,l) \times A_1(l,j)]$. If we define an operator on matrices $(\cdot)$ which has the same form as matrix multiplication, but substituting the minimum function for summation and addition for the cross product, then we may rewrite our sequence of matrices as

$$\begin{aligned} A_1(i,j) &= c(i,j) \\ A_2 &= A_1 \cdot A_1 \\ A_3 &= A_2 \cdot A_1 \\ A_4 &= A_3 \cdot A_1 \\ &\vdots \end{aligned}$$

While the change in the underlying operations means that we can not employ efficient matrix multiplication algorithms to speed up our own algorithm, if we look at the definition of A_4 we can see that the similarities to multiplication will still provide an opportunity for improvement. Specifically, expanding out the definition of A_4 gives us

$$A_4 = A_3 \cdot A_1 = A_2 \cdot A_1 \cdot A_1 = A_2 \cdot A_2,$$

which demonstrates how we may speed up our algorithm by mimicking the process which allows exponentiation by a power of 2 to be sped up by replacing it with repeated squaring.

Phrased in terms of our original statement of the problem, where before we would have split a path of length k into a path of length $k-1$ and a path of length 1, we now split such a path into two paths of length $k/2$. Essentially, we are leveraging the fact that while accommodating any possible start for our path increases the complexity of our algorithm, it also gives us more information to work with at each step. Previously, at step k, only paths of length less than k starting at s and paths of length 1 were inexpensive to check; now, at step k, we have that any path of length less than k is inexpensive to check. So, rather than splitting our new paths one step away from the destination, we may split them halfway between the start and the destination. Our recurrence relation becomes

$$A[i,j,k] = \min_l\{A[i,l,k/2] + A[l,j,k/2]\}, \text{ where } A[i,j,1] = c(i,j).$$

The running time for this version of the algorithm may be seen to be $O(|V|^3 \log |V|)$. In each iteration, in order to compute each of the $|V|^2$ elements in our array, we need to check each possible midpoint, which leads to each iteration taking $O(|V|^3)$ time. Since we double the value of k at each step, we need only $\lceil \log |V| \rceil$ iterations to reach our final array. Thus, it is clear that the overall running time must be $O(|V|^3 \log |V|)$, a definite improvement over our previous running time of $O(|V|^2 \cdot |E|)$ when run upon a dense graph.

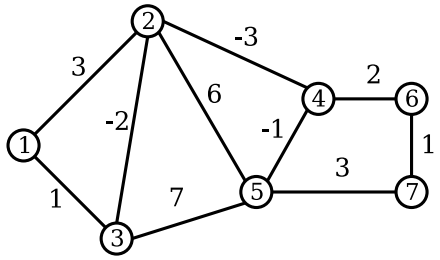
3.5 Negative Cycles

Throughout the preceding discussion on how to solve the shortest path problem — whether the single source variant or the all pairs variant — we have assumed that the graph we are given will not include any negative cost cycles. This assumption is critical to the problem itself, since the notion of a shortest path becomes ill-defined when a negative cost cycle is introduced into the graph. After all, once we have a negative cost cycle in the graph, we may better any proposed minimum cost for traveling between a given start node and end node simply by including enough trips around the negative cost cycle in the path we choose. While the shortest path problem is only well-defined in the absence of negative cost cycles, it would still be desirable for our algorithm to remain robust in the face of such issues.

In fact, the Bellman-Ford algorithm does behave predictably when given a graph containing one or more negative cycles as input. Once we have run the algorithm for long enough to determine the the best cost for all paths of length less than or equal $|V| - 1$, running it for one more iteration will tell us whether or not the graph has a negative cost cycle: our table will change if and only if a negative cost cycle is present in our input graph. In fact, we only need to look at the costs for a particular source to determine whether or not we have a negative cycle, and hence may make use of this test in the single source version of the algorithm as well. If we wish to find such a cycle after determining that one exists, we need only consider any pair of a start node and an end node which saw a decrease in minimum cost when paths of length $|V|$ were allowed. If we look at the first duplicate appearance of a node, the portion of the path from its first occurrence to its second occurrence will constitute a negative cost cycle.

3.6 Floyd-Warshall

A second way of decomposing a shortest path problem into subproblems is to reduce along the intermediate nodes used in the path. In order to do so, we must first augment the graph we are given by labeling its nodes from 1 to n; for example, we might change the graph we used as an example in section 3.2 to



When we reduce the shortest paths problem in this fashion, we end up with the table

$$A[i,j,k] = \text{ shortest path from } i \text{ to } j \text{ using only intermediate nodes from } 1,\ldots,k,$$

which is governed by the recurrence relation

$$A[i,j,k] = \min \left\{ \begin{array}{l} A[i,j,k-1] \\ A[i,k,k-1] + A[k,j,k-1] \end{array} \right. , \text{ where } A[i,j,0] = c(i,j).$$

It is reasonably straightforward to see that this will indeed yield a correct solution to the all pairs shortest path problem. Assuming once again that our graph is free of negative cost cycles, we can see that a minimum path using the first k nodes will use node k either 0 or 1 times; these cases correspond directly to the elements we take the minimum over in our definition of $A[i,j,k]$. An inductive proof of the correctness of the Floyd-Warshall algorithm follows directly from this observation.

If we wish to know the running time of this algorithm, we may once again consider the size of the table used, and the time required to compute each entry in it. In this case, we need a table that is of size $|V|^3$, and each entry in the table can be computed in $O(1)$ time; hence, we can see that the Floyd-Warshall algorithm will require $O(|V|^3)$ time in total.

Given that this improves upon the bound we found for the running time of the all pairs variant of the Bellman-Ford algorithm in (3.4), a natural question is why we might still wish to use the Bellman-Ford algorithm. First, within the context of the single source version of the problem, the Bellman-Ford algorithm actually provides the more attractive time bound, since the Floyd-Warshall algorithm doesn't scale down to this problem nicely. Second, considered within the context of the all pairs version of the problem, the main advantage it offers us is the way it handles negative cycles. While the Bellman-Ford algorithm allows us to test for the presence of negative cycles simply by continuing for one additional iteration, the Floyd-Warshall algorithm provides no such mechanism. Thus, unless we can guarantee that our graph will be free of negative cycles, we may actually prefer to use the Bellman-Ford algorithm, despite its less attractive time bound.