12 All-Pair Shortest Paths (October 24)

12.1 The Problem

In the last lecture, we saw algorithms to find the shortest path from a source vertex s to a target vertex t in a directed graph. As it turns out, the best algorithms for this problem actually find the shortest path from s to every possible target (or from every possible source to t) by constructing a shortest path tree. The shortest path tree specifies two pieces of information for each node v in the graph

- $\text{dist}(v)$ is the length of the shortest path (if any) from s to v.
- $\text{pred}(v)$ is the second-to-last vertex (if any) the shortest path (if any) from s to v.

In this lecture, we want to generalize the shortest path problem even further. In the *all pairs shortest path* problem, we want to find the shortest path from *every* possible source to *every* possible destination. Specifically, for every pair of vertices u and v, we need to compute the following information:

- $\text{dist}(u, v)$ is the length of the shortest path (if any) from u to v.
- $\text{pred}(u, v)$ is the second-to-last vertex (if any) on the shortest path (if any) from u to v.

For example, for any vertex v, we have $\text{dist}(v, v) = 0$ and $\text{pred}(v, v) = \text{NULL}$. If the shortest path from u to v is only one edge long, then $\text{dist}(u, v) = w(u \to v)$ and $\text{pred}(u, v) = u$. If there is *no* shortest path from u to v—either because there's no path at all, or because there's a negative cycle—then $\text{dist}(u, v) = \infty$ and $\text{pred}(v, v) = \text{NULL}$.

The output of our shortest path algorithms will be a pair of $V \times V$ arrays encoding all V^2 distances and predecessors. Many maps include a distance matrix—to find the distance from (say) Champaign to (say) Columbus, you would look in the row labeled 'Champaign' and the column labeled 'Columbus'. In these notes, I'll focus almost exclusively on computing the distance array. The predecessor array, from which you would compute the actual shortest paths, can be computed with only minor additions to the algorithms I'll describe (hint, hint).

12.2 Lots of Single Sources

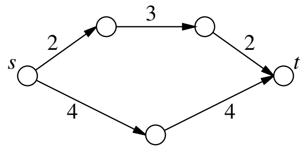
The most obvious solution to the all pairs shortest path problem is just to run a single-source shortest path algorithm V times, once for every possible source vertex! Specifically, to fill in the one-dimensional subarray $\text{dist}[s][\,]$, we invoke either Dijkstra's or Moore's algorithm starting at the source vertex s.

```
OBVIOUSAPSP(V, E, w):
  for every vertex s
      dist[s][] ← SSSP(V, E, w, s)
```

The running time of this algorithm depends on which single source algorithm we use. If we use Moore's algorithm, the overall running time is $\Theta(V^2E) = O(V^4)$. If all the edge weights are positive, we can use Dijkstra's algorithm instead, which decreases the running time to $\Theta(VE + V^2 \log V) = O(V^3)$. For graphs with negative edge weights, Dijkstra's algorithm can take exponential time, so we can't get this improvement directly.

12.3 Reweighting

One idea that occurs to most people is increasing the weights of all the edges by the same amount so that all the weights become positive, and then applying Dijkstra's algorithm. Unfortunately, this simple idea doesn't work. Different paths change by different amounts, which means the shortest paths in the reweighted graph may not be the same as in the original graph.



Increasing all the edge weights by 2 changes the shortest path s to t.

However, there is a more complicated method for reweighting the edges in a graph. Suppose each vertex v has some associated *cost* $c(v)$, which might be positive, negative, or zero. We can define a new weight function w' as follows:

$$w'(u \to v) = c(u) + w(u \to v) - c(v)$$

To give some intuition, imagine that when we leave vertex u, we have to pay an exit tax of $c(u)$, and when we enter v, we get $c(v)$ as an entrance gift.

Now it's not too hard to show that the shortest paths with the new weight function w' are exactly the same as the shortest paths with the original weight function w. In fact, for *any* path $u \rightsquigarrow v$ from one vertex u to another vertex v, we have

$$w'(u \rightsquigarrow v) = c(u) + w(u \rightsquigarrow v) - c(v).$$

We pay $c(u)$ in exit fees, plus the original weight of of the path, minus the $c(v)$ entrance gift. At every intermediate vertex x on the path, we get $c(x)$ as an entrance gift, but then immediately pay it back as an exit tax!

12.4 Johnson's Algorithm

Johnson's all-pairs shortest path algorithm finds a cost $c(v)$ for each vertex, so that when the graph is reweighted, every edge has non-negative weight.

Suppose the graph has a vertex s that has a path to every other vertex. Johnson's algorithm computes the shortest paths from s to every other vertex, using Moore's algorithm (which doesn't care if the edge weights are negative), and then sets

$$c(v) = \operatorname{dist}(s, v),$$

so the new weight of every edge is

$$w'(u \to v) = \operatorname{dist}(s, u) + w(u \to v) - \operatorname{dist}(s, v).$$

Why are all these new weights non-negative? Because otherwise, Moore's algorithm wouldn't be finished! Recall that an edge $u \to v$ is *tense* if $\operatorname{dist}(s, u) + w(u \to v) < \operatorname{dist}(s, v)$, and that single-source shortest path algorithms eliminate all tense edges. The only exception is if the graph has a negative cycle, but then shortest paths aren't defined, and Johnson's algorithm simply aborts.

But what if the graph *doesn't* have a vertex s that can reach everything? Then no matter where we start Moore's algorithm, some of those vertex costs will be infinite. Johnson's algorithm avoids this problem by adding a new vertex s to the graph, with zero-weight edges going from s to every other vertex, but *no* edges going back into s. This addition doesn't change the shortest paths between any other pair of vertices, because there are no paths into s.

So here's Johnson's algorithm in all its glory.

```
JohnsonAPSP(V, E, w) :
  create a new vertex s
  for every vertex v ∈ V
      w(s → v) ← 0;  w(v → s) ← ∞
  dist[s][] ← Moore(V, E, w, s)
  abort if Moore found a negative cycle

  for every edge (u, v) ∈ E
      w′(u → v) ← dist[s][u] + w(u → v) − dist[v][s]
  for every vertex v ∈ V
      dist[v][] ← Dijkstra(V, E, w′, v)
```

The algorithm spends $\Theta(V)$ time adding the artificial start vertex s, $\Theta(VE)$ time running Moore, $O(E)$ time reweighting the graph, and then $\Theta(VE + V^2 \log V)$ running V passes of Dijkstra's algorithm. The overall running time is $\boxed{\Theta(VE + V^2 \log V)}$.

12.5 Dynamic Programming

There's a completely different solution to the all-pairs shortest path problem that uses dynamic programming instead of a single-source algorithm. For *dense* graphs where $E = \Omega(V^2)$, the dynamic programming approach gives the same running time as Johnson's algorithm, but with a much simpler algorithm. In particular, the new algorithm avoids Dijkstra's algorithm, which gets its efficiency from Fibonacci heaps, which are rather easy to screw up in the implementation.

To get a dynamic programming algorithm, we first need to come up with a recursive formulation of the problem. If we try to recursively define $\text{dist}(u, v)$, we might get something like this:

$$\text{dist}(u,v) = \begin{cases} 0 & \text{if } u = v \\ \min_x \big(\text{dist}(u,x) + w(x \to v)\big) & \text{otherwise} \end{cases}$$

In other words, to find the shortest path from u to v, try all possible predecessors x, compute the shortest path from u to x, and then add the last edge $u \to v$. **Unfortunately, this recurrence doesn't work!** In order to compute $\text{dist}(u, v)$, we first have to compute $\text{dist}(u, x)$ for every other vertex x, but to compute any $\text{dist}(u, x)$, we first need to compute $\text{dist}(u, v)$. We're stuck in an infinite loop!

To avoid this circular dependency, we need some additional parameter that decreases at each recursion, eventually reaching zero at the base case. One possibility is to include the number of edges in the shortest path as this third magic parameter. So let's define $\text{dist}(u, v, k)$ to be the length of the shortest path from u to v that uses *at most* k edges. Since we know that the shortest path between any two vertices has at most $V - 1$ vertices, what we're really trying to compute is $\text{dist}(u, v, V - 1)$.

After a little thought, we get the following recurrence.

$$\text{dist}(u,v,k) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } k = 0 \text{ and } u \neq v \\ \min_x \big(\text{dist}(u,x,k-1) + w(x \to v)\big) & \text{otherwise} \end{cases}$$

Just like last time, the recurrence tries all possible predecessors of v in the shortest path, but now the recursion actually bottoms out when $k = 0$.

Now it's not difficult to turn this recurrence into a dynamic programming algorithm. Even before we write down the algorithm, though, we can tell that its running time will be $\boxed{\Theta(V^4)}$ simply because recurrence has four variables—u, v, k, and x—each of which can take on V different values. Except for the base cases, the algorithm itself is just four nested for loops. To make the algorithm a little shorter, let's assume that $w(v \to v) = 0$ for every vertex v.

```
DynamicProgrammingAPSP(V, E, w):
  for all vertices u ∈ V
      for all vertices v ∈ V
          if u = v
              dist[u][v][0] ← 0
          else
              dist[u][v][0] ← ∞
  for k ← 1 to V − 1
      for all vertices u ∈ V
          for all vertices v ∈ V
              dist[u][v][k] ← ∞
              for all vertices x ∈ V
                  if dist[u][v][k] > dist[u][x][k − 1] + w(x → v)
                      dist[u][v][k] ← dist[u][x][k − 1] + w(x → v)
```

The last four lines actually evaluate the recurrence.

In fact, this algorithm is almost exactly the same as running Moore's algorithm once for every source vertex. The only difference is the innermost loop, which in Moore's algorithm would read "for all edges $x \to v$". This simple change improves the running time to $\Theta(V^2E)$, assuming the graph is stored in an adjacency list.

12.6 Divide and Conquer

But we can make a more significant improvement. The recurrence we just used broke the shortest path into a slightly shorter path and a single edge, by considering all predecessors. Instead, let's break it into two shorter paths at the middle vertex on the path. This idea gives us a different recurrence for $\text{dist}(u, v, k)$. Once again, to simplify things, let's assume $w(v \to v) = 0$.

$$\text{dist}(u, v, k) = \begin{cases} w(u \to v) & \text{if } k = 1 \\ \min_x \big(\text{dist}(u, x, k/2) + \text{dist}(x, v, k/2)\big) & \text{otherwise} \end{cases}$$

This recurrence only works when k is a power of two, since otherwise we might try to find the shortest path with a fractional number of edges! But that's not really a problem, since $\text{dist}(u, v, 2^{\lceil \lg V \rceil})$ gives us the overall shortest distance from u to v. Notice that we use the base case $k = 1$ instead of $k = 0$, since we can't use half an edge.

Once again, a dynamic programming solution is straightforward. Even before we write down the algorithm, we can tell the running time is $\boxed{\Theta(V^3 \log V)}$—we consider V possible values of u, v, and x, but only $\lceil \lg V \rceil$ possible values of k.

```
FastDynamicProgrammingAPSP(V, E, w):
  for all vertices u ∈ V
      for all vertices v ∈ V
          dist[u][v][0] ← w(u → v)
  for i ← 1 to ⌈lg V⌉            ⟨⟨k = 2^i⟩⟩
      for all vertices u ∈ V
          for all vertices v ∈ V
              dist[u][v][i] ← ∞
              for all vertices x ∈ V
                  if dist[u][v][i] > dist[u][x][i − 1] + dist[x][v][i − 1]
                      dist[u][v][i] ← dist[u][x][i − 1] + dist[x][v][i − 1]
```

12.7 Aside: 'Funny' Matrix Multiplication

There is a very close connection between computing shortest paths in a directed graph and computing powers of a square matrix. Compare the following algorithm for multiplying two $n \times n$ matrices A and B with the inner loop of our first dynamic programming algorithm. (I've changed the variable names in the second algorithm slightly to make the similarity clearer.)

```
MatrixMultiply(A, B):
  for i ← 1 to n
      for j ← 1 to n
          C[i][j] ← 0
          for k ← 1 to n
              C[i][j] ← C[i][j] + A[i][k] · B[k][j]
```

```
APSPInnerLoop:
  for all vertices u
      for all vertices v
          D′[u][v] ← ∞
          for all vertices x
              D′[u][v] ← min{D′[u][v], D[u][x] + w[x][v]}
```

The *only* difference between these two algorithms is that we use addition instead of multiplication and minimization instead of addition. For this reason, the shortest path inner loop is often referred to as 'funny' matrix multiplication.

DynamicProgrammingAPSP is the standard iterative algorithm for computing the $(V-1)$th 'funny power' of the weight matrix w. The first set of for loops sets up the 'funny identity matrix', with zeros on the main diagonal and infinity everywhere else. Then each iteration of the second main for loop computes the next 'funny power'. FastDynamicProgrammingAPSP replaces this iterative method for computing powers with repeated squaring, exactly like we saw at the beginning of the semester. The fast algorithm is simplified slightly by the fact that unless there are negative cycles, every 'funny power' after the Vth is the same.

There are faster methods for multiplying matrices, similar to Karatsuba's divide-and-conquer algorithm for multiplying integers. (See 'Strassen's algorithm' in CLR.) Unfortunately, these algorithms us subtraction, and there's no 'funny' equivalent of subtraction. (What's the inverse operation for min?) So at least for general graphs, there seems to be no way to speed up the inner loop of our dynamic programming algorithms.

Fortunately, this isn't true. There is a beautiful randomized algorithm, due to Noga Alon, Zvi Galil, Oded Margalit*, and Moni Noar,[1] that computes all-pairs shortest paths in undirected graphs in $O(M(V)\log^2 V)$ expected time, where $M(V)$ is the time to multiply two $V \times V$ integer matrices. A simplified version of this algorithm for *unweighted* graphs, due to Raimund Seidel[2], appears in the current homework.

12.8 Floyd and Warshall's Algorithm

Our fast dynamic programming algorithm is still a factor of $O(\log V)$ slower than Johnson's algorithm. A different formulation due to Floyd and Warshall removes this logarithmic factor. Their insight was to use a different third parameter in the recurrence.

Number the vertices arbitrarily from 1 to V, and define $\mathrm{dist}(u, v, r)$ to be the length of the shortest path from u to v, where all the *intermediate* vertices (if any) are numbered r or less. If $r = 0$, we aren't allowed to use any intermediate vertices, so the shortest legal path from u to v is just the edge (if any) from u to v. If $r > 0$, then either the shortest legal path from u to v goes through vertex r or it doesn't. We get the following recurrence:

$$\mathrm{dist}(u, v, r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min\left\{\mathrm{dist}(u, v, r-1),\ \mathrm{dist}(u, r, r-1) + \mathrm{dist}(r, v, r-1)\right\} & \text{otherwise} \end{cases}$$

We need to compute the shortest path distance from u to v with no restrictions, which is just $\mathrm{dist}(u, v, V)$.

Once again, we should immediately see that a dynamic programming algorithm that implements this recurrence will run in $\boxed{\Theta(V^3)}$ time: three variables appear in the recurrence (u, v, and r), each of which can take on V possible values. Here's one way to do it:

```
FloydWarshall(V, E, w):
  for u ← 1 to V
      for v ← 1 to V
          dist[u][v][0] ← w(u → v)
  for r ← 1 to V
      for u ← 1 to V
          for v ← 1 to V
              if dist[u][v][r − 1] < dist[u][r][r − 1] + dist[r][v][r − 1]
                  dist[u][v][r] ← dist[u][v][r − 1]
              else
                  dist[u][v][r] ← dist[u][r][r − 1] + dist[r][v][r − 1]
```

[1] N. Alon, Z. Galil, O. Margalit*, and M. Naor. Witnesses for Boolean matrix multiplication and for shortest paths. *Proc. 33rd FOCS* 417-426, 1992. See also N. Alon, Z. Galil, O. Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

[2] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995. This is one of the few algorithms papers where (in the conference version at least) the algorithm is completely described and analyzed *in the abstract* of the paper.