In this lecture, we did a review of single source shortest paths and all pair shortest paths. This included reviews of Dijkstra's algorithm, the Bellman-Ford algorithm, Johnson's algorithm, and the Floyd-Marshall algorithm. We also discussed the possibility of using min-sum products to find all pairs shortest path solutions. Lastly, we did an in depth discussion of Seidel's algorithm, which finds the all pairs shortest path solution for an undirected graph with all edge weights equal to 1.

1 Single Source Shortest Path Algorithms

For a weighted directed graph, the shortest path problem finds the path with the lowest total weight between two vertices. The single source shortest path problem (SSSP) finds the shortest path from a single vertex to every other vertex in the graph.

1.1 Dijkstra's Algorithm

The best running time for SSSP comes Dijkstra's algorithm, a greedy algorithm. In Dijkstra's algorithm, we start with an initial guess of ∞ for the distance to each node. Then in each step, we look at the vertex in our priority queue with the shortest distance from the source. We see if we can go from that node to each of its neighbors and get a shorter path than that neighbor's current estimate. If so, we "relax" that estimate; that's to say, we update it with the new lower value. We also add the vertex to our priority. We repeat until the priority queue is empty.

Dijkstra's algorithm has a few advantages. Being a greedy algorithm, it's fairly intuitive. It's also fast. According to the 210 lecture notes, it takes $O(m \log n)$ time when implemented with a binary heap, but it improvess to $O(m + n \log n)$ time with a Fibonacci heap. The downside of Dijkstra's algorithm is that it doesn't work with negative edge weights. This reason for this we assume that once every vertex in the priority queue has a bigger distance than our current vertex, we have found the shortest path to our current vertex.

For full pseudocode of Dijkstra's algorithm, I suggest looking at the myriad web sources. In particular Wikipedia has some really nifty animations that explain it well.

Modern scholars have actually found a slightly better implementation of Dijkstra's algorithm that runs in $O(m + n \log \log n)$ time.

1.2 Bellman-Ford Algorithm

The Bellman-Ford algorithm is a common alternative to Dijkstra's algorithm with the benefit of allowing for negative weight edges.

The algorithm starts with an overestimate of the shortest path to each vertex, just like Dijkstra's algorithm. At each step of the algorithm, each vertex looks at whether its current shortest path length plus its edge weight results in a lower estimate for its neighbor (it does this once for each adjacent edge). If so, we relax that edge's estimate. We repeat n times.

Wikipedia gives a nice lemma that makes the correctness clear.

Lemma 4.1. *Lemma. After i repetitions of for loop:*

If Distance(u) is not infinity, it is equal to the length of some path from s to u;

If there is a path from s to u with at most i edges, then Distance(u) is at most the length of the shortest path from s to u with at most i edges.

No path can be more than n in length, so n repetitions is sufficient for all estimates to be correct.

The caveat for this is that Bellman-Ford does not give the correct answer if the graph contains a negative cycle. In fact, there is no well defined shortest path if there is a negative cycle; any path can be made shorter by adding more times around the cycle. However, Bellman Ford can detect these negative cycles. After n iterations of the algorithm, if we do an additional iteration and one of the estimates changes, then there must be a negative cycle.

Each step of Bellman-Ford looks at each edge once, and there are n steps, so the runtime is $O(m*n)$, which is worse than Dijkstras algorithm.

Again, this is a well-known algorithm with a myriad of demonstrations online. You can also find detailed pseudocode on the Wikipedia page.

2 All Pairs Shortest Paths (APSP)

Like the name implies, APSP is the problem of finding the shortest path between any combination of two pairs you choose. The most obvious way to do this is to run SSSP on every vertex in the graph. The runtime will be n multiplied by the runtime of your $SSSP$ algorithm. Thus for Bellman-Ford it is $O(mn^2)$, and for Dijkstra it is the slightly better $O(mn + n^2\ log\ log\ n)$. However, we must remember the limitations of Dijkstra's algorithm: no negative edge weights allowed. Fortunately, Johnson's algorithm provides a clever trick to bypass this issue and still get $O(mn + n^2\ log\ log\ n)$.

2.1 Johnson's Algorithm

Johnson's algorithm consists of two steps:

1) Do Bellman-Ford on a single vertex to reweight the edge weights so that they're nonnegative (taking care to not change what the shortest path is for any pair).

2) Run n instances of Dijkstra's SSSP algorithm.

The first step is the trickier of the two. To perform the reweighting, we give each vertex a "feasible potential." We find the feasible potentials by introducing a new vertex, connecting it to every other vertex by a zero-weight edge, and running Bellman-Ford from the new vertex. Note that this has the convenient side effect of finding any negative weight cycles that might exist. When Bellman-Ford is finished, we set the feasible potential of each vertex to be the shortest path length we found for that vertex; call it $C(v)$.

Now we do the reweighting. Specifically we use the following formula:

$$w_{new}(x, y) = w(x, y) + C(y) - C(x)$$

This reweighting scheme has two advantages. The first is that all edge weights are now nonnegative (we did not prove this fact in detail in class). The second advantage is that it preserves what the shortest path is. This is because any path from x to y must have its length increased by $C(y)-C(x)$. The potentials on the intermediate vertices are added to one edge and subtracted from the next such that they have no net effect. If every path is increased by the same amount, the optimal path can't change.

With no negative potentials, running Dijkstra's algorithm at each vertex gives the APSP. The total runtime will be $O(mn + n^2\ log\ log\ n)$.

2.2 Floyd-Marshall's Algorithm

The Floyd-Marshall algorithm is perhaps best understood through its strikingly simple pseudocode

for all vertices z

  for all vertices x,y

    $d(x, y) \leftarrow min\{d(x, y), d(x, z) + d(z, y)\}$

This has a runtime of $O(n^3)$...definitely worse than Johnson's algorithm. But it does have a few advantages. It is simple, and it is quick to implement with minimal errors. The most common error is nesting the for-loops in reverse. That yields the wrong answer, so watch out there. Another advantage is that Floyd-Marshall is also parellelizable. Lastly, it is very cache efficient.

3 Min-Sum Products and APSPs

Let us define the *min-sum product* of two matrices to be:

$$(A\ \circ\ B)_{ij} = min_{k=1}^{n}(A_{ik} + B_{kj})$$

Compare and contrast it with matrix multiplication:

$$(A * B)_{ij} = \sum_{k=1}^{n} A_{ik} + B_{kj}$$

Let us now define the adjacency matrix A for a given graph to be:

$$A_{ij} = \begin{cases} w_{ij}, & \text{if } i, j \in E \\ \infty, & \text{if not} i, j \in E \\ 0, & \text{if } i = j \end{cases}$$

The min-sum product of an adjacency matrix with itself can provide insight into shortest paths. For example $(A\ \circ\ A)_{ij}$ equals the length of the shortest 1 or 2 hop path from vertex i to vertex j.

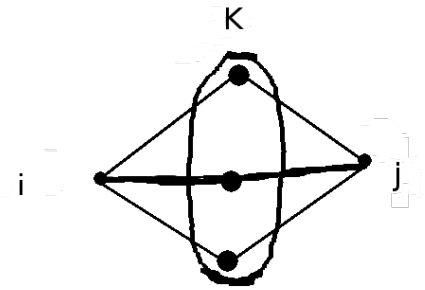

Figure 4.1: $(A\ \circ\ A)_{ij}$ looks at all of the 2-hop paths from i to j

Inductively, we can then see that $((A \circ A) \circ A)$ will be a matrix of shortest paths of 3 hops or fewer. We can continue up to $A \circ A \circ A... \circ A$ with n multiplications (call this $A\circ_n$), which gives the shortest n-hop path between any two vertices. No shortest path can be more than n hops, so we can use n min-sum products on the adjacency matrix to get the APSP.

How long will this take? At first it may look like like $A\circ_n$ will take n min-sum product calculations, but that is not true. By noting $A\circ_n = A\circ_{n/2} \circ A\circ_{n/2}$, we can get $A\circ_n$ recursively in $log\ n$ min-sum product steps. So our runtime is $O(log\ n \times MSP(n))$, where $MSP(n)$ is the time it takes to take the min-sum product of two $n \times n$ matrices.

How long is $MSP(n)$? Naively, it takes n^3 time, just like matrix multiplication. But matrix multiplication can be sped up. Straussen's algorithm can do it in $O(n^{log_2(7)})$. And we don't know the exact exponent for the optimal matrix multiplication solution, but we write it as $O(n^\omega)$. Can we do min-sum product in $O(n^\omega)$ time? Unfortunately scholars do not know. In fact, we don't even know if it can be done in $O(n^{3-\epsilon})$ time.

We have however made some improvement over the $O(n^3)$ time. Fredman showed an algorithm for doing the min-sum product in $O(\frac{n^3}{log\ n})$. Then in 1994 a CMU alumni, Williams, improved this to $O(\frac{n^3}{2^{\sqrt{log\ n}}})$. To make that more clear, recall $2^{c\ log\ log\ n} = (log\ n)^c < 2^{c\sqrt{log\ n}} < n^c = 2^{c\ log\ n}$. We'll discuss thes algoirthms in lecture 5.

All of these APSP algorithms discussed so far work in general. However, we'll now segue to Seidel's algorithm, which does APSP in $O(n^\omega\ log\ n)$ but only for a special graph.

4 Seidel's Algorithm

Seidel's Algorithm is an APSP algorithm, but it only works on undirected graphs where each edge has weight one.

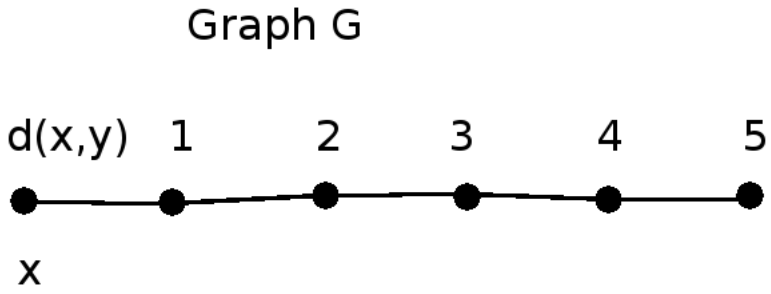

Figure 4.2: The vertices are connected by $w = 1$ undirected edges. Distances from vertex x are shown.

This algorithm works by recursively shrinking the given graph G to a graph H such that H has 2 useful properties: H's shortest pairs are about half as long as G's, and the APSP of H can be used to quickly find the APSP of G.

This shrink operation is very simple: if the distance between two vertices was 2 in G, connect those vertices with an edge in H. This creates many triangles. Let d(x,y) be the shortest path length from x to y in G and let $D(x, y)$ be the shortest path length from x to y in H.

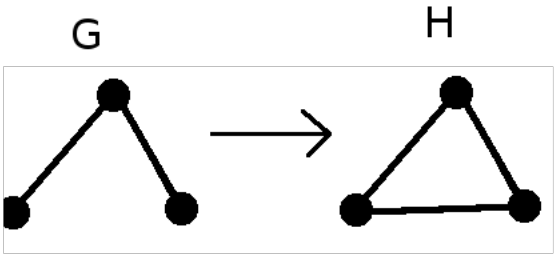

Figure 4.3: 2-Edge lines in G become triangles in H

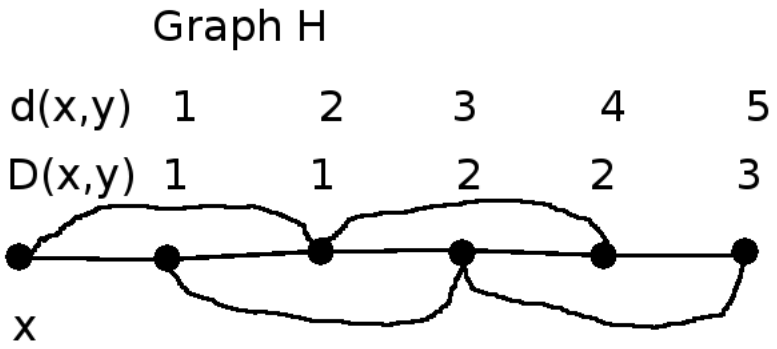

Figure 4.4: Put an edge between every edge (x, y) where $d(x, y) \leq 2$. Old and new distances are shown.

4.1 Matrix representation of Graphs

We'll prove some useful claims about the relationship between d and D, but first we need to take note of how we algebraically determine H from G. We let G be represented by the adjacency matrix MG. In this matrix we define each element as follows:

$$MG_{ij} = \begin{cases} 1, & \text{if } i,j \in G_E \\ 0, & \text{if not} i,j \in E \text{ or if } i = j \end{cases}$$

We can then represent H by a different adjacency matrix MH as follows:

$$MH_{ij} = \begin{cases} 1, & \text{if } MG_{ij} = 1 \text{ or if there exists } k \text{ s.t. } MG_{ik} = MG_{jk} = 1 \\ 0, & \text{otherwise} \end{cases}$$

We can calculate MH by noting $MH = MG + MG^2$.

We return to the task of relating d to D.

4.2 Relating the Distances of the Two Graphs

Lemma 4.2. $D(x,y) = \lceil d(x,y)/2 \rceil$.

Proof: We'll prove this equality by proving to nonstrict inequalities.

$D(x,y) \leq \lceil d(x,y)/2 \rceil$ because for any even d, we can get there in half as many jumps, and for any odd d, when can get to the previous vertex (an even distance) in half as many jumps, then just use one more jump to get to the target vertex.

$D(x,y) \geq \lceil d(x,y)/2 \rceil$ because every edge in H must be one or two edges in G so we can't hope to do better than cut our path in half.

Corallary: $d(x,y) \in \{2 * D(xy), 2 * D(x,y) - 1\}$

Proof: This is how ceiling is defined.

Lemma 4.3. *Suppose $d(x,y) = 2D(x,y)$. In this case, all neighbors z of y in G have have $D(x,z) \geq D(x,y)$*

Proof: Assume the inverse is true: y has a neighbor z in G such that $D(x,z) < D(x,y)$. Now we take stock of some inequalities and derive a contradiction.

We can first rewrite our assumption as $D(x,z)+1 \leq D(x,y)$. Then we can double that and subtract one to get $2D(x,z) + 1 \leq 2D(x,y) - 1$

We know from Lemma 4.2 that $D(x,z) = \lceil d(x,z)/2 \rceil$. This implies $d(x,z) + 1 \leq 2 * D(x,z) + 1$

We know $d(x,y) \leq d(x,z) + 1$ because they're neighbors.

We can string these inequalities together as $d(x,y) \leq d(x,z) + 1 \leq 2D(x,z) + 1 \leq 2D(x,y) - 1 < 2D(x,y)$.

We pull out the left and right ends of the inequality to get $d(x,y) < 2D(x,y)$. But our initial supposition was that $d(x,y) = 2D(x,y)$. So our assumption that there exists a neighbor z in G such that $D(x,z) < D(x,y)$ was wrong, and we've proved the claim via contradiction.

Lemma 4.4. *If $d(x,y) = 2D(x,y)$, $\sum_{neighbors\ z\ of\ y} D(x,z) \geq deg(y) * D(x,y)$*

Proof: We didn't do this proof in detail in class, but it's only a little bit of algebra on top of Lemma 4.3

Lemma 4.5. *If $d(x,y) = 2D(x,y) - 1$, then $D(x,z) \leq D(x,y)$ for all neighbors z of y in G*

Proof:

Again, because z and y are adjacent in G, we can say $d(x,z) \leq d(x,y) + 1 \leq (2D(x,y) - 1) + 1 = 2D(x,y)$

Dividing by 2, then adding ceilings doesn't change the inequality: $\lceil d(x,z)/2 \rceil \leq \lceil 2D(x,y)/2 \rceil$

Now recall $D(x,z) = \lceil d(x,z)/2 \rceil$

We can combine the two statements to get $D(x,z) \leq \lceil 2D(x,y)/2 \rceil = D(x,y)$

Lemma 4.6. *If $d(x,y) = 2D(x,y) - 1$, y has a neighbor z^* in G such that $D(x,z^*) < D(x,y)$*

Proof:

Let z^* be the immediate predecessor to y on the shortest path to y. Then $d(x,z) = d(x,y) - 1 = 2D(x,y) - 2$.

Now recall $\lceil d(x,z^*)/2 \rceil = D(x,z^*)$

$D(x,z^*) = \lceil (2D(x,y) - 2)/2 \rceil = D(x,y) - 1 < D(x,y)$

$D(x,z^*) < D(x,y)$

Lemma 4.7. *If $d(x,y) = 2D(x,y) - 1$, then $\sum_{neighbors\ z\ of\ y} D(x,z) \leq deg(y) * D(x,y)) - 1$*

Proof: Again, this wasn't explicitly proven in class, but it's just an algebraic manipulation of Lemma 4.5 and Lemma 4.6

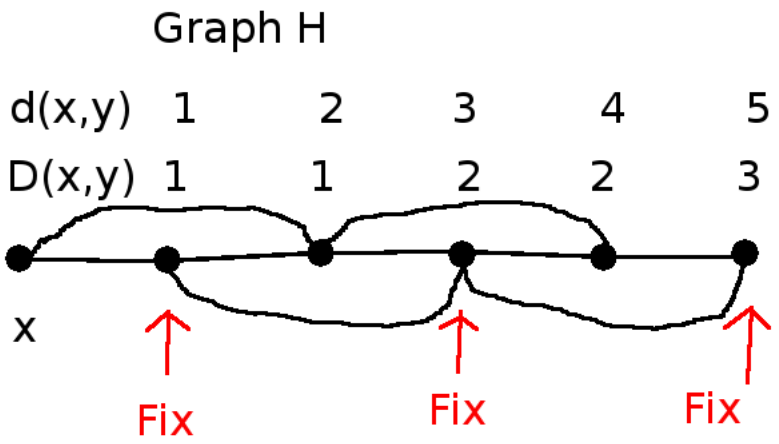

Figure 4.5: We assume every distance gets cut in half. Then, depending on whether they have a neighbor with smaller $D(x,z)$, we decide whether to subtract one.

With these claims out of the way, we present the psuedocode for the algorithm.

The for loop is a likely source of confusion here. Basically, for every shortest path we find in H, we assume the corresponding path length in G in twice as long, then "fix" the esimate by subtracting one if certain conditions are met. Specifically we test this statement:

$$2(D * MG)_{ij} < D_{ij} * degree(j)$$

Algorithm 1 Seidel's algorithm

```
1: procedure SEIDEL(MG)
2:     //Get H from G
3:     MH ← MG + MG^2
4:     //Recursively solve problem on H
5:     D ← Seidel(MH)
6:     //We assume everything is off by a factor of two and fix if necessary
7:     for i, j do
8:         d(i, j) ← 2D(i, j) − 1 ∗ ((D ∗ MG)_ij < D_ij ∗ degree(j)
9:     Return d
```

The left half is equivalent to $\sum_z D(x,z) * MG_{zy}$; that's to say it is the sum of the path lengths to the neighbors of Y. Lemmas 4.4 and 4.7 show how this lets us see how to relationship from distances in H to G.

This algorithm makes $log\ n$ recursive calls. Why? Each call lowers the length of the longest path by a factor of two. Once everything is connected by a path of length 1 (which we can quickly check), we are finished.

Each recursive call takes $O(n^\omega)$ time since it involves a matrix multiplication. The total time is thus $O(n^\omega log\ n)$.