25 All-Pairs Shortest Paths

25.1 Shortest paths and matrix multiplication

25.1-1

> Run $\text{SLOW-ALL-PAIRS-SHORTEST-PATHS}$ on the weighted, directed graph of Figure 25.2, showing the matrices that result for each iteration of the loop. Then do the same for $\text{FASTER-ALL-PAIRS-SHORTEST-PATHS}$.

- Initial:

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

- Slow:

  $m = 2$:

$$\begin{pmatrix} 0 & 6 & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 3 & -3 & 0 & 4 & \infty & -8 \\ -4 & 10 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & \infty & 0 \end{pmatrix}$$

  $m = 3$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -2 & -3 & 0 & -1 & 2 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

  $m = 4$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -3 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

m = 5:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

- Fast:

  m = 2:

$$\begin{pmatrix} 0 & 6 & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 3 & -3 & 0 & 4 & \infty & -8 \\ -4 & 10 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & \infty & 0 \end{pmatrix}$$

  m = 4:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -3 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

  m = 8:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

25.1-2

> Why do we require that $w_{ii} = 0$ for all $1 \leq i \leq n$?

This is consistent with the fact that the shortest path from a vertex to itself is the empty path of weight 0. If there were another path of weight less than 0 then it must be a negative-weight cycle, since it starts and ends at v_i.

If $w_{ii} \neq 0$, then $L^{(1)}$ produced after the first run of $\text{EXTEND-SHORTEST-PATHS}$ would not contain the minimum weight of any path from i to its neighbours. If $w_{ii} = 0$, then in line 7 of $\text{EXTEND-SHORTEST-PATHS}$, the second argument to $\min$ would not equal the weight of the edge going from i to its neighbours.

25.1-3

> What does the matrix
>
> $$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$
>
> used in the shortest-paths algorithms correspond to in regular matrix multiplication?

The identity matrix.

25.1-4

> Show that matrix multiplication defined by $\text{EXTEND-SHORTEST-PATHS}$ is associative.

To verify associativity, we need to check that $(W^iW^j)W^p = W^i(W^jW^p)$ for all i, j and p, where we use the matrix multiplication defined by the $\text{EXTEND-SHORTEST-PATHS}$ procedure. Consider entry (a, b) of the left hand side. This is:

$$\begin{aligned} \min_{1 \le k \le n} [W^iW^j]_{a,k} + W^p_{k,b} & = \min_{1 \le k \le n} \min_{1 \le q \le n} W^i_{a,q} + W^j_{q,k} + W^p_{k,b} \\ & = \min_{1 \le q \le n} W^i_{a,q} + \min_{1 \le k \le n} W^j_{q,k} + W^p_{k,b} \\ & = \min_{1 \le q \le n} W^i_{a,q} + [W^jW^p]_{q,b}, \end{aligned}$$

which is precisely entry (a, b) of the right hand side.

25.1-5

> Show how to express the single-source shortest-paths problem as a product of matrices and a vector. Describe how evaluating this product corresponds to a Bellman-Ford-like algorithm (see Section 24.1).

(Removed)

25.1-6

> Suppose we also wish to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix $\prod$ from the completed matrix L of shortest-path weights in $O(n^3)$ time.

For each source vertex v_i we need to compute the shortest-paths tree for v_i. To do this, we need to compute the predecessor for each $j \neq i$. For fixed i and j, this is the value of k such that $L_{i,k} + w(k, j) = L[i, j]$. Since there are n vertices whose trees need computing, n vertices for each such tree whose predecessors need computing, and it takes $O(n)$ to compute this for each one (checking each possible k), the total time is $O(n^3)$.

25.1-7

> We can also compute the vertices on shortest paths as we compute the shortestpath weights. Define $\pi_{ij}^{(m)}$ as the predecessor of vertex j on any minimum-weight path from i to j that contains at most m edges. Modify the $\text{EXTEND-SHORTESTPATHS}$ and $\text{SLOW-ALL-PAIRS-SHORTEST-PATHS}$ procedures to compute the matrices$\prod^{(1)}, \prod^{(2)}, \ldots, \prod^{(n-1)}$ as the matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ are computed.

To have the procedure compute the predecessor along the shortest path, see the modified procedures, $\text{EXTEND-SHORTEST-PATH-MOD}$ and $\text{SLOW-ALL-PAIRS-SHORTEST-PATHS-MOD}$

```
EXTEND-SHORTEST-PATH-MOD(∏, L, W)
    n = L.row
    let L' = l'[i, j] be a new n × n matirx
    ∏' = π'[i, j] is a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            l'[i, j] = ∞
            π'[i, j] = NIL
            for k = 1 to n
                if l[i, k] + l[k, j] < l[i, j]
                    l[i, j] = l[i, k] + l[k, j]
                    if k != j
                        π'[i, j] = k
                    else
                        π'[i, j] = π[i, j]
    return (∏', L')
```

```
SLOW-ALL-PAIRS-SHORTEST-PATHS-MOD(W)
    n = W.rows
    L(1) = W
    ∏(1) = π[i, j](1) where π[i, j](1) = i if there is an edge from i to j,
and NIL otherwise
    for m = 2 to n - 1
        ∏(m), L(m) = EXTEND-SHORTEST-PATH-MOD(∏(m - 1), L(m - 1), W)
    return (∏(n - 1), L(n - 1))
```

25.1-8

> The $\text{FASTER-ALL-PAIRS-SHORTEST-PATHS}$ procedure, as written, requires us to store $\lceil \lg(n-1) \rceil$ matrices, each with n^2 elements, for a total space requirement of $\Theta(n^2 \lg n)$. Modify the procedure to require only $\Theta(n^2)$ space by using only two $n \times n$ matrices.

We can overwrite matrices as we go. Let $A \star B$ denote multiplication defined by the $\text{EXTEND-SHORTEST-PATHS}$ procedure. Then we modify $\text{FASTER-ALL-EXTEND-SHORTEST-PATHS}(W)$. We initially create an n by n matrix L. Delete line 5 of the algorithm, and change line 6 to $L = W \star W$, followed by $W = L$.

25.1-9

> Modify $\text{FASTER-ALL-PAIRS-SHORTEST-PATHS}$ so that it can determine whether the graph contains a negative-weight cycle.

For the modification, keep computing for one step more than the original, that is, we compute all the way up to $L^{(2k+1)}$ where $2^k > n - 1$. Then, if there aren't any negative weight cycles, then, we will have that the two matrices should be equal since having no negative weight cycles means that between any two vertices, there is a path that is tied for shortest and contains at most $n - 1$ edges.

However, if there is a cycle of negative total weight, we know that it's length is at most n, so, since we are allowing paths to be larger by $2k \ge n$ between these two matrices, we have that we would need to have all of the vertices on the cycle have their distance reduce by at least the negative weight of the cycle. Since we can detect exactly when there is a negative cycle, based on when these two matrices are different. This algorithm works. It also only takes time equal to a single matrix multiplication which is littlee oh of the unmodified algorithm.

25.1-10

> Give an efficient algorithm to find the length (number of edges) of a minimum-length negative-weight cycle in a graph.

(Removed)

25.2 The Floyd-Warshall algorithm

25.2-1

> Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 25.2. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

$k = 1:$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

$k = 2:$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$k = 3:$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$k = 4:$

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$k = 5:$

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$k = 6$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

25.2-2

> Show how to compute the transitive closure using the technique of Section 25.1.

We set $w_{ij} = 1$ if (i, j) is an edge, and $w_{ij} = 0$ otherwise. Then we replace line 7 of $\text{EXTEND-SHORTEST-PATHS}(L, W)$ by $l''_{ij} = l''_{ij} \lor (l_{ik} \land w_{kj})$. Then run the $\text{SLOW-ALL-PAIRS-SHORTEST-PATHS}$ algorithm.

25.2-3

> Modify the FLOYD-WARSHALL procedure to compute the $\prod^{(k)}$ matrices according to equations (25.6) and (25.7). Prove rigorously that for all $i \in V$, the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root i. (Hint: To show that $G_{\pi,i}$ is acyclic, first show that $\pi_{ij}^{(k)} = l$ implies $d_{ij}^{(k)} \ge d_{il}^{(k)} + w_{lj}$, according to the definition of $\pi_{ij}^{(k)}$. Then, adapt the proof of Lemma 23.16.)

```
MOD-FLOYD-WARSHALL(W)
    n = W.rows
    D(0) = W
    let π(0) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            if i != j and D[i, j](0) < ∞
                π[i, j](0) = i
    for k = 1 to n
        let D(k) be a new n × n matrix
        let π(k) be a new n × n matrix
        for i = 1 to n
            for j = 1 to n
                if d[i, j](k - 1) ≤ d[i, k](k - 1) + d[k, j](k - 1)
                    d[i, j](k) = d[i, j](k - 1)
                    π[i, j](k) = π[i, j](k - 1)
                else
                    d[i, j](k) = d[i, k](k - 1) + d[k, j](k - 1)
                    π[i, j](k) = π[k, j](k - 1)
```

In order to have that $\pi_{ij}^{(k)} = l$, we need that $d_{ij}^{(k)} \ge d_{il}^{(k)} + w_{lj}$. To see this fact, we will note that having $\pi_{ij}^{(k)} = l$ means that a shortest path from i to j last goes through l. A path that last goes through l corresponds to taking a chepest path from i to l and then following the single edge from l to j. However, This means that $d_{il} \le d_{ij} - w_{ij}$, which we can rearrange to get the desired inequality. We can just continue following this inequality around, and if we ever get some cycle, $i_1, i_2, \ldots, i_c$, then we would have that $d_{ii_1} \le d_{ii_1} + w_{i_1 i_2} + w_{i_2 i_3} + \cdots + w_{i_c i_1}$. So, if we subtract the common term sfrom both sides, we get that $0 \le w_{i_c i_1} + \sum_{q=1}^{c-1} w_{i_q i_{q+1}}$. So, we have that we would only have a cycle in the precedessor graph if we ahvt that there is a zero weight cycle in the original graph. However, we would never have to go around the weight zero cycle since the constructed path of shortest weight favors ones with a fewer number of edges because of the way that we handle the equality case in equation (25.7).

25.2-4

> As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.
>
> ```
> FLOYD-WARSHALL'(W)
>     n = W.rows
>     D = W
>     for k = 1 to n
>         for i = 1 to n
>             for j = 1 to n
>                 d[i, j] = min(d[i, j], d[i, k] + d[k, j])
>     return D
> ```

(Removed)

25.2-5

> Suppose that we modify the way in which equation (25.7) handles equality:
>
> $$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \ge d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$
>
> Is this alternative definition of the predecessor matrix $\prod$ correct?

If we change the way that we handle the equality case, we will still be generating a the correct values for the π matrix. This is because updating the π values to make paths that are longer but still tied for the lowest weight. Making $\pi_{ij} = \pi_{kj}$ means that we are making the shortest path from i to j passes through k at some point. This has the same cost as just going from i to j, since $d_{ij} = d_{ik} + d_{kj}$.

25.2-6

> How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle?

(Removed)

25.2-7

> Another way to reconstruct shortest paths in the Floyd-Warshall algorithm uses values $\phi_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$, where $\phi_{ij}^{(k)}$ is the highest-numbered intermediate vertex of a shortest path from i to j in which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. Give a recursive formulation for $\phi_{ij}^{(k)}$, modify the FLOYD-WARSHALL procedure to compute the $\phi_{ij}^{(k)}$ values, and rewrite the $\text{PRINT-ALLPAIRS-SHORTEST-PATH}$ procedure to take the matrix $\Phi = \left(\phi_{ij}^{(n)}\right)$ as an input. How is the matrix Φ like the s table in the matrix-chain multiplication problem of Section 15.2?

We can recursively compute the values of $\phi_{ij}^{(k)}$ by, letting it be $\phi_{ij}^{(k-1)}$ if $d_{ik}^{(k)} + d_{kj}^{(k)} \ge d_{ij}^{(k-1)}$, and otherwise, let it be k. This works correctly because it perfectly captures whether we decided to use vertex k when we were repeatedly allowing ourselves use of each vertex one at a time. To modify Floyd-Warshall to compute this, we would just need to stick within the innermost for loop, something that computes $\phi(k)$ by this recursive rule, this would only be a constant amount of work in this innermost for loop, and so would not cause the asymptotic runtime to increase. It is similar to the s table in matrix-chain multiplication because it is computed by a similar recurrence.

If we already have the n^3 values in $\phi_{ij}^{(k)}$ provided, then we can reconstruct the shortest path from i to j because we know that the largest vertex in the path from i to j is $\phi_{ij}^{(n)}$, call it a_1. Then, we know that the largest vertex in the path before a_1 will be $\phi_{ia_1}^{(a_1-1)}$ and the largest after a_1 will be $\phi_{a_1 j}^{(a_1-1)}$. By continuing to recurse until we get that the largest element showing up at some point is NIL, we will be able to continue subdividing the path until it is entirely constructed.

25.2-8

> Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.

We can determine the vertices reachable from a particular vertex in $O(V + E)$ time using any basic graph searching algorithm. Thus we can compute the transitive closure in $O(VE + V^2)$ time by searching the graph with each vertex as the source. If $|V| = O(E)$, we're done as VE is now the dominating term in the running time bound. If not, we preprocess the graph and mark all degree-0 vertices in $O(V + E)$ time. The rows representing these vertices in the transitive closure are all 0s, which means that the algorithm remains correct if we ignore these vertices when searching. After preprocessing, $|V| = O(E)$ as $|E| \geq |V|/2$. Therefore searching can be done in $O(VE)$ time.

25.2-9

> Suppose that we can compute the transitive closure of a directed acyclic graph in $f(|V|, |E|)$ time, where f is a monotonically increasing function of $|V|$ and $|E|$. Show that the time to compute the transitive closure $G^* = (V, E^*)$ of a general directed graph $G = (V, E)$ is then $f(|V|, |E|) + O(V + E^*)$.

First, compute the strongly connected components of the directed graph, and look at it's component graph. This component graph is going to be acyclic and have at most as many vertices and at most as many edges as the original graph. Since it is acyclic, we can run our transitive closure algorithm on it. Then, for every edge (S_1, S_2) that shows up in the transitive closure of the component graph, we add an edge from each vertex in S_1 to a vertex in S_2. This takes time equal to $O(V + E')$. So, the total time required is $\leq f(|V|, |E|) + O(V + E)$.

25.3 Johnson's algorithm for sparse graphs

25.3-1

> Use Johnson's algorithm to find the shortest paths between all pairs of vertices in the graph of Figure 25.2. Show the values of h and $\hat{w}$ computed by the algorithm.

v	$h(v)$
1	-5
2	-3
3	0
4	-1
5	-6
6	-8

u	v	$\hat{w}(u, v)$	u	v	$\hat{w}(u, v)$
1	2	NIL	4	1	0
1	3	NIL	4	2	NIL
1	4	NIL	4	3	NIL
1	5	0	4	5	8
1	6	NIL	4	6	NIL
2	1	3	5	1	NIL
2	3	NIL	5	2	4
2	4	0	5	3	NIL
2	5	NIL	5	4	NIL
2	6	NIL	5	6	NIL
3	1	NIL	6	1	NIL
3	2	5	6	2	0
3	4	NIL	6	3	2
3	5	NIL	6	4	NIL
3	6	0	6	5	NIL

So, the d_{ij} values that we get are

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}.$$

25.3-2

> What is the purpose of adding the new vertex s to V', yielding V'?

This is only important when there are negative-weight cycles in the graph. Using a dummy vertex gets us around the problem of trying to compute $-\infty + \infty$ to find $\hat{w}$. Moreover, if we had instead used a vertex v in the graph instead of the new vertex s, then we run into trouble if a vertex fails to be reachable from v.

25.3-3

> Suppose that $w(u, v) \ge 0$ for all edges $(u, v) \in E$. What is the relationship between the weight functions w and $\hat{w}$?

If all the edge weights are nonnegative, then the values computed as the shortest distances when running Bellman-Ford will be all zero. This is because when constructing G' on the first line of Johnson's algorithm, we place an edge of weight zero from s to every other vertex. Since any path within the graph has no negative edges, its cost cannot be negative, and so, cannot beat the trivial path that goes straight from s to any given vertex. Since we have that $h(u) = h(v)$ for every u and v, the reweighting that occurs only adds and subtracts 0, and so we have that $w(u, v) = \hat{w}(u, v)$

25.3-4

> Professor Greenstreet claims that there is a simpler way to reweight edges than the method used in Johnson's algorithm. Letting $w^* = \min_{(u,v)\in E}\{w(u, v)\}$, just define $\hat{w}(u, v) = w(u, v) - w^*$ for all edges $(u, v) \in E$. What is wrong with the professor's method of reweighting?

(Removed)

25.3-5

> Suppose that we run Johnson's algorithm on a directed graph G with weight function w. Show that if G contains a 0-weight cycle c, then $\hat{w}(u, v) = 0$ for every edge (u, v) in c.

If $\delta(s, v) - \delta(s, u) \le w(u, v)$, we have

$$\delta(s, u) \le \delta(s, v) + (0 - w(u, v)) < \delta(s, u) + w(u, v) - w(u, v) = \delta(s, u),$$

which is impossible, thus $\delta(s, v) - \delta(s, u) = w(u, v)$, $\hat{w}(u, v) = w(u, v) + \delta(s, u) - \delta(s, v) = 0$.

25.3-6

> Professor Michener claims that there is no need to create a new source vertex in line 1 of JOHNSON. He claims that instead we can just use $G' = G$ and let s be any vertex. Give an example of a weighted, directed graph G for which incorporating the professor's idea into JOHNSON causes incorrect answers. Then show that if G is strongly connected (every vertex is reachable from every other vertex), the results returned by JOHNSON with the professor's modification are correct.

(Removed)

Problem 25-1 Transitive closure of a dynamic graph

> Suppose that we wish to maintain the transitive closure of a directed graph $G = (V, E)$ as we insert edges into E. That is, after each edge has been inserted, we want to update the transitive closure of the edges inserted so far. Assume that the graph G has no edges initially and that we represent the transitive closure as a boolean matrix.
>
> **a.** Show how to update the transitive closure $G^* = (V, E^*)$ of a graph $G = (V, E)$ in $O(V^2)$ time when a new edge is added to G.
>
> **b.** Give an example of a graph G and an edge e such that $\Omega(V^2)$ time is required to update the transitive closure after the insertion of e into G, no matter what algorithm is used.
>
> **c.** Describe an efficient algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of n insertions, your algorithm should run in total time $\sum_{i=1}^{n} t_i = O(V^3)$,

> where t_i is the time to update the transitive closure upon inserting the ith edge. Prove that your algorithm attains this time bound.

a. We can update the transitive closure in time $O(V^2)$ as follows. Suppose that we add the edge (x_1, x_2). Then, we will consider every pair of vertices (u, v). In order to of created a path between them, we would need some part of that path that goes from u to x_1 and some second part of that path that goes from x_2 to v. This means that we add the edge (u, v) to the transitive closure if and only if the transitive closure contains the edges (u, x_1) and (x_2, v). Since we only had to consider every pair of vertices once, the runtime of this update is only $O(V^2)$.

b. Suppose that we currently have two strongly connected components, each of size $|V|/2$ with no edges between them. Then their transitive closures computed so far will consist of two complete directed graphs on $|V|/2$ vertices each. So, there will be a total of $|V|^2/2$ edges adding the number of edges in each together. Then, we add a single edge from one component to the other. This will mean that every vertex in the component the edge is coming from will have an edge going to every vertex in the component that the edge is going to. So, the total number of edges after this operation will be $|V|/2 + |V|/4$ So, the number of edges increased by $|V|/4$. Since each time we add an edge, we need to use at least constant time, since there is no cheap way to add many edges at once, the total amount of time needed is $\Omega(|V|^2)$.

c. We will have each vertex maintain a tree of vertices that have a path to it and a tree of vertices that it has a path to. The second of which is the transitive closure at each step. Then, upon inserting an edge, (u, v), we will look at successive ancestors of u, and add v to their successor tree, just past u. If we ever don't insert an edge when doing this, we can stop exploring that branch of the ancestor tree. Similarly, we keep doing this for all of the ancestors of v. Since we are able to short circuit if we ever notice that we have already added an edge, we know that we will only ever reconsider the same edge at most n times, and, since the number of edges is $O(n^2)$, the total runtime is $O(n^3)$.

Problem 25-2 Shortest paths in epsilon-dense graphs

> A graph $G = (V, E)$ is ϵ-***dense*** if $|E| = \Theta(V^{1+\epsilon})$ for some constant ϵ in the range $0 < \epsilon \le 1$. By using d-ary min-heaps (see Problem 6-2) in shortest-paths algorithms on ϵ-dense graphs, we can match the running times of Fibonacci-heap-based algorithms without using as complicated a data structure.
>
> **a.** What are the asymptotic running times for INSERT, EXTRACT-MIN, and DECREASE-KEY, as a function of d and the number n of elements in a d-ary min-heap? What are these running times if we choose $d = \Theta(n^\alpha)$ for some constant $0 < \alpha \le 1$? Compare these running times to the amortized costs of these operations for a Fibonacci heap.
>
> **b.** Show how to compute shortest paths from a single source on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(E)$ time. (Hint: Pick d as a function of ϵ.)
>
> **c.** Show how to solve the all-pairs shortest-paths problem on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(VE)$ time.

> **d.** Show how to solve the all-pairs shortest-paths problem in $O(VE)$ time on an ϵ-dense directed graph $G = (V, E)$ that may have negative-weight edges but has no negative-weight cycles.

a.

- $\text{INSERT}: \Theta(\log_d n) = \Theta(1/\alpha)$.
- $\text{EXTRACT-MIN}: \Theta(d \log_d n) = \Theta(n^\alpha / \alpha)$.
- $\text{DECREASE-KEY}: \Theta(\log_d n) = \Theta(1/\alpha)$.

b. Dijkstra, $O(d \log_d V \cdot V + \log_d V \cdot E)$, if $d = V^\epsilon$, then

$$\begin{aligned} O(d \log_d V \cdot V + \log_d V \cdot E) &= O(V^\epsilon \cdot V/\epsilon + E/\epsilon) \\ &= O((V^{1+\epsilon} + E)/\epsilon) \\ &= O((E + E)/\epsilon) \\ &= O(E). \end{aligned}$$

c. Run $|V|$ times Dijkstra, since the algorithm is $O(E)$ based on (b), the total time is $O(VE)$.

d. Johnson's reweight is $O(VE)$.