17-0: Computing Shortest Path

- Given a directed weighted graph $G$ (all weights non-negative) and two vertices $x$ and $y$, find the least-cost path from $x$ to $y$ in $G$.
  - Undirected graph is a special case of a directed graph, with symmetric edges
  - Least-cost path may not be the path containing the fewest edges
  - “shortest path” == “least cost path”
  - “path containing fewest edges” = “path containing fewest edges”

17-1: Shortest Path Example

- Shortest path \neq path containing fewest edges

![Graph Diagram]

- Shortest Path from A to E?

17-2: Shortest Path Example

- Shortest path \neq path containing fewest edges

![Graph Diagram]

- Shortest Path from A to E:
  - A, B, C, D, E

17-3: Single Source Shortest Path
• To find the shortest path from vertex $x$ to vertex $y$, we need (worst case) to find the shortest path from $x$ to all other vertices in the graph

• Why?

17-4: **Single Source Shortest Path**

• To find the shortest path from vertex $x$ to vertex $y$, we need (worst case) to find the shortest path from $x$ to all other vertices in the graph

  • To find the shortest path from $x$ to $y$, we need to find the shortest path from $x$ to all nodes on the path from $x$ to $y$

  • Worst case, all nodes will be on the path

17-5: **Single Source Shortest Path**

• If all edges have unit weight ...

17-6: **Single Source Shortest Path**

• If all edges have unit weight,

  • We can use Breadth First Search to compute the shortest path

  • BFS Spanning Tree contains shortest path to each node in the graph

    • Need to do some more work to create & save BFS spanning tree

    • When edges have differing weights, this obviously will not work

17-7: **Single Source Shortest Path**

• General Idea for finding Single Source Shortest Path

  • Start with the distance estimate to each node (except the source) as $\infty$

  • Repeatedly relax distance estimate until you can relax no more

  • To relax and edge $(u, v)$

    • $\text{dist}(v) > \text{dist}(u) + \text{cost}(u, v)$

    • Set $\text{dist}(v) \leftarrow \text{dist}(u) + \text{cost}(u, v)$

17-8: **Single Source Shortest Path**

• Dijkstra’s algorithm

  • Relax edges from source

  • *Remarkably* similar to Prim’s MST algorithm

    • Pretty neat – algorithms are doing different things, but code is almost identical

17-9: **Single Source Shortest Path**

• Divide the vertices into two sets:

  • Vertices whose shortest path from the initial vertex is known
- Vertices whose shortest path from the initial vertex is not known
- Initially, only the initial vertex is known
- Move vertices one at a time from the unknown set to the known set, until all vertices are known

17-10: Single Source Shortest Path

- Start with the vertex A

17-11: Single Source Shortest Path

- Known vertices are circled in red
- We can now extend the known set by 1 vertex

17-12: Single Source Shortest Path
• Why is it safe to add D, with cost 1?

17-13: **Single Source Shortest Path**

- Why is it safe to add D, with cost 1?
  - Could we do better with a more roundabout path?

17-14: **Single Source Shortest Path**

- Why is it safe to add D, with cost 1?
  - Could we do better with a more roundabout path?
    - No – to get to any other node will cost at least 1
    - No negative edge weights, can’t do better than 1

17-15: **Single Source Shortest Path**

- We can now add another vertex to our known list ...
17-16: Single Source Shortest Path

- How do we know that we could not get to B cheaper by going through D?

17-17: Single Source Shortest Path

- How do we know that we could not get to B cheaper by going through D?
  - Costs 1 to get to D
  - Costs at least 2 to get anywhere from D
    - Cost at least \((1+2 = 3)\) to get to B through D

17-18: Single Source Shortest Path

- Next node we can add ...
17-19: Single Source Shortest Path

- (We also could have added E for this step)
- Next vertex to add to Known ...

17-20: Single Source Shortest Path

- Cost to add F is 8 (through C)
- Cost to add G is 5 (through D)

17-21: Single Source Shortest Path

- Last node ...
17-22: **Single Source Shortest Path**

![Graph with distances]

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
</tr>
</tbody>
</table>

- We now know the length of the shortest path from $A$ to all other vertices in the graph

17-23: **Dijkstra’s Algorithm**

- Keep a table that contains, for each vertex
  - Is the distance to that vertex known?
  - What is the best distance we’ve found so far?
- Repeat:
  - Pick the smallest unknown distance
  - mark it as known
  - update the distance of all unknown neighbors of that node
- Until all vertices are known

17-24: **Dijkstra’s Algorithm Example**

![Graph with distances and table]

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>false</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>$\infty$</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>$\infty$</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>$\infty$</td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>$\infty$</td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

17-25: **Dijkstra’s Algorithm Example**
17-26: Dijkstra’s Algorithm Example

17-27: Dijkstra’s Algorithm Example

17-28: Dijkstra’s Algorithm Example
17-29: **Dijkstra’s Algorithm Example**

**Node** | **Known** | **Distance**
---|---|---
A | true | 0
B | false | 5
C | true | 4
D | false | 6
E | true | 3
F | true | 1

17-30: **Dijkstra’s Algorithm Example**

**Node** | **Known** | **Distance**
---|---|---
A | true | 0
B | true | 5
C | true | 4
D | false | 6
E | true | 3
F | true | 1

17-31: **Dijkstra’s Algorithm**

- After Dijkstra’s algorithm is complete:
• We know the length of the shortest path
• We do not know what the shortest path is

How can we modify Dijkstra’s algorithm to compute the path?

17-32: Dijkstra’s Algorithm

• After Dijkstra’s algorithm is complete:
  • We know the length of the shortest path
  • We do not know what the shortest path is

• How can we modify Dijkstra’s algorithm to compute the path?
  • Store not only the distance, but the immediate parent that led to this distance

17-33: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>false</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

17-34: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

17-35: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>
17-36: Dijkstra’s Algorithm Example

Node | Known | Dist | Path
--- | --- | --- | ---
A | true | 0 | 
B | false | 5 | A
C | true | 3 | A
D | true | 4 | C
E | false | 9 | D
F | false | 9 | D
G | false | 7 | D

17-37: Dijkstra’s Algorithm Example

Node | Known | Dist | Path
--- | --- | --- | ---
A | true | 0 | 
B | true | 5 | A
C | true | 3 | A
D | true | 4 | C
E | false | 9 | D
F | false | 9 | D
G | false | 7 | D

17-38: Dijkstra’s Algorithm Example

Node | Known | Dist | Path
--- | --- | --- | ---
A | true | 0 | 
B | true | 5 | A
C | true | 3 | A
D | true | 4 | C
E | false | 9 | D
F | false | 8 | G
G | true | 7 | D

17-39: Dijkstra’s Algorithm Example

Node | Known | Dist | Path
--- | --- | --- | ---
A | true | 0 | 
B | true | 5 | A
C | true | 3 | A
D | true | 4 | C
E | false | 9 | D
F | true | 8 | G
G | true | 7 | D

17-40: Dijkstra’s Algorithm Example
17-1: Dijkstra’s Algorithm

- Given the “path” field, we can construct the shortest path
  - Work backward from the end of the path
  - Follow the “path” pointers until the start node is reached
    - We can use a sentinel value in the “path” field of the initial node, so we know when to stop

17-2: Dijkstra Code

```java
void Dijkstra(Edge G[], int s, tableEntry T[]) {   
    int i, v;
    Edge e;
    for(i=0; i<G.length; i++) {   
        T[i].distance = Integer.MAX_VALUE;
        T[i].path = -1;
        T[i].known = false;
    }
    T[s].distance = 0;
    for (i=0; i < G.length; i++) {
        v = minUnknownVertex(T);
        T[v].known = true;
        for (e = G[v]; e != null; e = e.next) {
            if (T[e.neighbor].distance > T[v].distance + e.cost) {
                T[e.neighbor].distance = T[v].distance + e.cost;
                T[e.neighbor].path = v;
            }
        }
    }
}
```

17-3: Dijkstra Running Time

- If minUnknownVertex(T) is calculated by doing a linear search through the table:
  - Each minUnknownVertex call takes time Θ(|V|)
    - Called |V| times – total time for all calls to minUnknownVertex: Θ(|V|^2)
  - If statement is executed |E| times, each time takes time O(1)
    - Total time: O(|V|^2 + |E|) = O(|V|^2).

17-4: Dijkstra Running Time

- If minUnknownVertex(T) is calculated by inserting all vertices into a min-heap (using distances as key) updating the heap as the distances are changed
  - Each minUnknownVertex call takes time Θ(\lg |V|)
    - Called |V| times – total time for all calls to minUnknownVertex: Θ(|V| \lg |V|)
  - If statement is executed |E| times – each time takes time \( O(\lg |V|) \), since we need to update (decrement) keys in heap
Total time: \( O(|V| \log |V| + |E| \log |V|) \in O(|E| \log |V|) \)

17-45: **Dijkstra Running Time**

- If `minUnknownVertex(T)` is calculated by inserting all vertices into a Fibonacci heap (using distances as key) updating the heap as the distances are changed
  - Each `minUnknownVertex` call takes amortized time \( \Theta(|V|) \)
    - Called \(|V|\) times – total amortized time for all calls to `minUnknownVertex`: \( \Theta(|V| \log |V|) \)
  - If statement is executed \(|E|\) times – each time takes amortized time \( O(1) \), since decrementing keys takes time \( O(1) \).
- Total time: \( O(|V| \log |V| + |E|) \)

17-46: **Negative Edges**

- Does Dijkstra’s algorithm work when edge costs can be negative?
  - Give a counterexample!
- What happens if there is a negative-weight cycle in the graph?

17-47: **Bellman-Ford**

- Bellman-Ford allows us to calculate shortest paths in graphs with negative edge weights, as long as there are no negative-weight cycles
- As a bonus, we will also be able to detect negative-weight cycles

17-48: **Bellman-Ford**

- For each node \( v \), maintain:
  - A “distance estimate” from source to \( v \), \( d[v] \)
  - Parent of \( v \), \( \pi[v] \), that gives this distance estimate
- Start with \( d[v] = \infty \), \( \pi[v] = \text{nil} \) for all nodes
- Set \( d[\text{source}] = 0 \)
- Update estimates by “relaxing” edges

17-49: **Bellman-Ford**

- Relaxing an edge \((u, v)\)
  - See if we can get a better distance estimate for \( v \) by going through \( u \)

\[
\text{Relax}(u,v,w) \\
\text{if } d[v] > d[u] + w(u, v) \\
\quad d[v] \leftarrow d[u] + w(u, v) \\
\quad \pi[v] \leftarrow u
\]

17-50: **Bellman-Ford**
- Relax all edges in the graph (in any order)
- Repeat until relax steps cause no change
  - After first relaxing, all optimal paths from source of length 1 are computed
  - After second relaxing, all optimal paths from source of length 2 are computed
  - After |V| - 1 relaxing, all optimal paths of length |V| - 1 are computed
  - If some path of length |V| is cheaper than a path of length |V| - 1 that means ...

17-51: Bellman-Ford
- Relax all edges in the graph (in any order)
- Repeat until relax steps cause no change
  - After first relaxing, all optimal paths from source of length 1 are computed
  - After second relaxing, all optimal paths from source of length 2 are computed
  - After |V| - 1 relaxing, all optimal paths of length |V| - 1 are computed
  - If some path of length |V| is cheaper than a path of length |V| - 1 that means ...
    - Negative weight cycle

17-52: Bellman-Ford
BellmanFord(G, s)
  Initialize d[], π[]
  for i ← 1 to |V| - 1 do
    for each edge (u, v) ∈ G do
      if d[v] > d[u] + w(u, v)
        d[v] ← d[u] + w(u, v)
        π[v] ← u
    for each edge (u, v) ∈ G do
      if d[v] > d[u] + w(u, v)
        return false
  return true

17-53: Bellman-Ford
- Running time:
  - Each iteration requires us to relax all |E| edges
  - Each single relaxation takes time O(1)
  - |V| - 1 iterations (|V| if we are checking for negative weight cycles)
  - Total running time O(|V| * |E|)

17-54: Shortest Path/DAGs
- Finding Single Source Shorest path in a Directed, Acyclic graph
- Very easy! How can we do this quickly?

17-55: Shortest Path/DAGs
• Finding Single Source Shortest path in a Directed, Acyclic graph
• Very easy!
• How can we do this quickly?
  • Do a topological sort
  • Relax edges in topological order
  • We’re done!

17-56: **All-Source Shortest Path**

• What if we want to find the shortest path from all vertices to all other vertices?
• How can we do it?

17-57: **All-Source Shortest Path**

• What if we want to find the shortest path from all vertices to all other vertices?
• How can we do it?
  • Run Dijkstra’s Algorithm $V$ times
  • How long will this take?

17-58: **All-Source Shortest Path**

• What if we want to find the shortest path from all vertices to all other vertices?
• How can we do it?
  • Run Dijkstra’s Algorithm $V$ times
  • How long will this take?
  • $\Theta(V^2 \lg V + VE)$ (using Fibonacci heaps)
    • Doesn’t work if there are negative edges! Running Bellman-Ford $V$ times (which does work with negative edges) takes time $O(V^2 E)$ – which is $\Theta(V^4)$ for dense graphs

17-59: **Multi-Source Shortest Path**

• Let $L^{(m)}[i, j]$ (in text, $l_{i,j}^{(m)}$) be cost of the shortest path from $i$ to $j$ that contains at most $m$ edges
• If $m = 0$, there is a shortest path from $i$ to $j$ with no edges iff $i = j$

$$L^{(0)}[i, j] = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

• How can we calculate $L^{m}[i, j]$ recursively?

17-60: **Multi-Source Shortest Path**

• Let $L^{(m)}[i, j]$ (in text, $l_{i,j}^{(m)}$) be cost of the shortest path from $i$ to $j$ that contains at most $m$ edges

$$L^{(0)}[i, j] = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$
How can we calculate \( L^m[i, j] \) recursively?

\[
L^m[i, j] = \min \left( L^{m-1}[i, j], \min_{1 \leq k \leq n} (L^{m-1}[i, k] + w_{kj}) \right)
\]

17-61: **Multi-Source Shortest Path**

- Create \( L^{(m+1)} \) from \( L^{(m)} \):

```
Extend-Shortest-Paths(L, W)
```

```
n ← rows[L]
L′ ← new n × n matrix
for i ← 1 to n do
    for j ← 1 to n do
        L′[i, j] ← ∞
    for k ← 1 to n do
        L′[i, j] ← min(L′[i, j], L[i, k] + W[k, j])
return L′
```

17-62: **Multi-Source Shortest Path**

- Need to calculate \( L^{(n-1)} \)
  - Why \( L^{(n-1)} \), and not \( L^{(n)} \) or \( L^{(n+1)} \)?

```
All-Pairs-Shortest-Paths(W)
n ← rows[W]
L^{(1)} ← W
for m ← 2 to n - 1 do
    L^{(m)} ← Extend-Shortest-Path(L^{(m-1)}, W)
return L^{(n-1)}
```

17-63: **Multi-Source Shortest Path**

- We really don’t care about any of the \( L \) matrices except \( L^{(n-1)} \)
- We can save some time by not calculating all of the intermediate matrices \( L^{(1)} \ldots L^{(n-2)} \)
- Note that Extend-Shortest-Path looks a lot like matrix multiplication

17-64: **Multi-Source Shortest Path**

```
Square-Matrix-Multiply(A, B)
n ← rows[A]
C ← new n × n matrix
for i ← 1 to n do
    for j ← 1 to n do
        C[i, j] ← 0
    for k ← 1 to n do
return L′
```
• Replace min with +, + with *

17-65: Multi-Source Shortest Path

• Using our “Extend-Multiplication”
  • Replace + with min, * with +

\[
L^{(1)} = L^{(0)} \ast W = W \\
L^{(1)} = L^{(1)} \ast W = W^2 \\
L^{(2)} = L^{(2)} \ast W = W^3 \\
L^{(3)} = L^{(3)} \ast W = W^4 \\
\vdots \\
L^{(n-1)} = L^{(n-2)} \ast W = W^{n-1}
\]

17-66: Multi-Source Shortest Path

\[
L^{(1)} = W \\
L^{(2)} = W^2 = W \ast W \\
L^{(4)} = W^4 = W^2 \ast W^2 \\
L^{(8)} = W^8 = W^4 \ast W^4 \\
\vdots \\
L^{2^{[\log(n-1)]}} = L^{2^{[\log(n-1)]}} = L^{2^{[\log(n-1)]}-1} \ast L^{2^{[\log(n-1)]}-1}
\]

• Since \(L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots\), it doesn’t matter if \(n\) is an exact power of 2 – we just need to get to at least \(L^{(n-1)}\), not hit it exactly

17-67: Multi-Source Shortest Path

All-Pairs-Shortest-Paths(W)
\[
n \leftarrow \text{rows}[W] \\
L^{(1)} \leftarrow W \\
m \leftarrow 1 \\
\text{while } m < n - 1 \text{ do} \\
L^{(2m)} \leftarrow \text{Extend-Shortest-Path}(L^{(m)}, L^{(m)}) \\
m \rightarrow m \ast 2 \\
\text{return } L^{(m)}
\]

17-68: Multi-Source Shortest Path

• Each call to Extend-Shortest-Path takes time:

• # of calls to Extend-Shortest-Path:

• Total time:
17-69: **Multi-Source Shortest Path**

- Each call to Extend-Shortest-Path takes time Θ(|V|^3)
- # of calls to Extend-Shortest-Path: Θ(lg |V|)
- Total time: Θ(|V|^3 lg |V|)

17-70: **Floyd’s Algorithm**

- Alternate solution to all pairs shortest path
- Yields Θ(V^3) running time for all graphs

17-71: **Floyd’s Algorithm**

- Vertices numbered from 1..n
- k-path from vertex v to vertex u is a path whose intermediate vertices (other than v and u) contain only vertices numbered k or less
- 0-path is a direct link

17-72: **k-path Examples**

- Shortest 0-path from 1 to 5: 5
- Shortest 1-path from 1 to 5: 5
- Shortest 2-path from 1 to 5: 4
- Shortest 3-path from 1 to 5: 4
- Shortest 4-path from 1 to 5: 3

17-73: **k-path Examples**

- Shortest 0-path from 1 to 3: 7
• Shortest 1-path from 1 to 3: 7
• Shortest 2-path from 1 to 3: 6
• Shortest 3-path from 1 to 3: 6
• Shortest 4-path from 1 to 3: 6
• Shortest 5-path from 1 to 3: 4

17-74: **Floyd’s Algorithm**

• Shortest \( n \)-path = Shortest path
• Shortest 0-path:
  • \( \infty \) if there is no direct link
  • Cost of the direct link, otherwise

17-75: **Floyd’s Algorithm**

• Shortest \( n \)-path = Shortest path
• Shortest 0-path:
  • \( \infty \) if there is no direct link
  • Cost of the direct link, otherwise
  • If we could use the shortest \( k \)-path to find the shortest \( (k + 1) \) path, we would be set

17-76: **Floyd’s Algorithm**

• Shortest \( k \)-path from \( v \) to \( u \) either goes through vertex \( k \), or it does not
• If not:
  • Shortest \( k \)-path = shortest \( (k - 1) \)-path
• If so:
  • Shortest \( k \)-path = shortest \( k - 1 \) path from \( v \) to \( k \), followed by the shortest \( k - 1 \) path from \( k \) to \( w \)

17-77: **Floyd’s Algorithm**

• If we had the shortest \( k \)-path for all pairs \((v, w)\), we could obtain the shortest \( k + 1 \)-path for all pairs
  • For each pair \( v, w \), compare:
    • length of the \( k \)-path from \( v \) to \( w \)
    • length of the \( k \)-path from \( v \) to \( k \) appended to the \( k \)-path from \( k \) to \( w \)
    • Set the \( k + 1 \) path from \( v \) to \( w \) to be the minimum of the two paths above

17-78: **Floyd’s Algorithm**

• Let \( D_k[v, w] \) be the length of the shortest \( k \)-path from \( v \) to \( w \).
• \( D_0[v, w] = \) cost of arc from \( v \) to \( w \) (\( \infty \) if no direct link)
• $D_k[v, w] = \text{MIN}(D_{k-1}[v, w], D_{k-1}[v, k] + D_{k-1}[k, w])$

• Create $D_0$, use $D_0$ to create $D_1$, use $D_1$ to create $D_2$, and so on – until we have $D_n$

17-79: Floyd’s Algorithm

• Use a doubly-nested loop to create $D_k$ from $D_{k-1}$
  • Use the same array to store $D_{k-1}$ and $D_k$ – just overwrite with the new values
  • Embed this loop in a loop from 1..k

17-80: Floyd’s Algorithm

Floyd(Edge G[], int D[][]) {
  int i,j,k

  Initialize D, D[i][j] = cost from i to j

  for (k=0; k<G.length; k++)
    for (i=0; i<G.length; i++)
      for (j=0; j<G.length; j++)
        if ((D[i][k] != Integer.MAX_VALUE) &&
          (D[k][j] != Integer.MAX_VALUE) &&
          (D[i][j] > (D[i,k] + D[k,j])))
          D[i][j] = D[i][k] + D[k][j]
}

17-81: Floyd’s Algorithm

• We’ve only calculated the distance of the shortest path, not the path itself

• We can use a similar strategy to the PATH field for Dijkstra to store the path
  • We will need a 2-D array to store the paths: P[i][j] = last vertex on shortest path from i to j

17-82: Johnson’s Algorithm

• Yet another all-pairs shortest path algorithm

• Time $O(|V|^2 \log |V| + |V| * |E|)$
  • If graph is dense ($|E| \in \Theta(|V|^2)$), no better than Floyd
  • If graph is sparse, better than Floyd

• Basic Idea: Run Dijkstra $|V|$ times
  • Need to modify graph to remove negative edges

17-83: Johnson’s Algorithm

• Reweighing Graph
  • Create a new weight function $\hat{w}$, such that:
    • For all pairs of vertices $u, v \in V$, a path from $u$ to $v$ is a shortest path using $w$ if and only if it is also a shortest path using $\hat{w}$. 
For all edges \((u, v)\), \(\hat{w}(u, v)\) is non-negative.

17-84: **Johnson’s Algorithm**

- **Reweighing Graph**
  - **First Try:**
    - Smallest weight is \(-w\), for some positive \(w\)
    - Add \(w\) to each edge in the graph
    - Is this a valid reweighing?

17-85: **Johnson’s Algorithm**

- **Reweighing Graph**
  - **First Try:**
    - Smallest weight is \(-w\), for some positive \(w\)
    - Add \(w\) to each edge in the graph
    - Is this a valid reweighing?

```
A ---- 4 ---- D
|         |
|            |
B ---- 5 ---- C
```

17-86: **Johnson’s Algorithm**

- **Reweighing Graph**
  - **Second Try:**
    - Define some function on vertices \(h(v)\)
    - \(\hat{w}(u, v) = w(u, v) + h(u) - h(v)\)
    - Does this preserve shortest paths?

17-87: **Johnson’s Algorithm**

- Let \(p = v_0, v_1, v_2, \ldots, v_k\) be a path in \(G\)
- Cost of \(p\) under \(\hat{w}\):

\[
\hat{w}(p) = \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i)
= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i))
= \left( \sum_{i=1}^{k} (w(v_{i-1}, v_i)) + h(v_0) - h(v_k) \right)
= w(p) + h(v_0) - h(v_k)
\]
Thus, any shortest path under \( w \) will be a shortest path under \( \hat{w} \), and vice-versa

17-88: **Johnson’s Algorithm**

- So, if we can come up with a function \( h(V) \) such that \( w(u, v) + h(u) - h(v) \) is positive for all edges \((u, v)\) in the graph, we’re set
  - Use the function \( h \) to reweigh the graph
  - Run Dijkstra’s algorithm \(|V|\) times, starting from each vertex on the new graph, calculating shortest paths
  - Shortest path in new graph = shortest path in old graph

17-89: **Johnson’s Algorithm**

- Add a new vertex \( s \) to the graph
- Add an edge from \( s \) to every other vertex, with cost 0
- Find the shortest path from \( s \) to every other vertex in the graph
- \( h(v) = \delta(s, v) \), the cost of the shortest path from \( s \) to \( v \)
  - Using this \( h(V) \) function, all new weights are guaranteed to be non-negative

17-90: **Johnson’s Algorithm**

- \( h(v) = \delta(s, v) \), the cost of the shortest path from \( s \) to \( v \)

\[
\hat{w}(u, v) = w(u, v) + h(u) - h(v) = w(u, v) + \delta(s, u) - \delta(s, v)
\]

- Since \( \delta \) is a shortest path,

\[
\delta(s, v) \leq \delta(s, u) + w(u, v) \\
0 \leq w(u, v) + \delta(s, u) - \delta(s, v)
\]
17-92: Johnson’s Algorithm

17-93: Johnson’s Algorithm
17-94: Johnson’s Algorithm

17-95: Johnson’s Algorithm
17-96: **Johnson’s Algorithm**

**Johnson(G)**

- Add \( s \) to \( G \), with 0 weight edges to all vertices
- if Bellman-Ford\((G, s)\) = FALSE
  - There is a negative weight cycle, fail
- for each vertex \( v \in G \)
  - set \( h(v) \leftarrow \delta(s, v) \) from B-F
- for each edge \((u, v) \in G\)
  - \( \hat{w}(u, v) = w(u, v) + h(u) - h(v) \)
- for each vertex \( u \in G \)
  - run Dijkstra\((G, \hat{w}, u)\) to compute \( \hat{\delta}(u, v) \)
  - \( \delta(u, v) = \hat{\delta}(u, v) + h(v) - h(u) \)