Lecture: Analysis of Algorithms (CS583 - 002)

Amarda Shehu

Fall 2017

Some material adapted from Kevin Wayne’s Algorithm Class © Princeton
1 Single-source Shortest Paths
   - Dijkstra’s Algorithm
   - Bellman-Ford’s Algorithm

2 All-pairs Shortest Paths
   - Dynamic Programming Approach
   - Floyd-Warshall
Shortest Paths Problems

- Single-source/destination shortest path: Shortest path from a given vertex to any other vertex in $V$
  - Dijkstra’s on graphs with nonnegative weights
  - Bellman-Ford’s on graphs with nonnegative cycles

- All-pairs shortest path: Shortest-paths between any pair
  - Can run single-source shortest path problem on every vertex
  - Solved faster with Floyd-Warshall’s algorithm
Let $G = (V, E)$ be an unweighted graph

- The shortest path between two vertices in $G$ is a path with the shortest **length** (least number of edges). This length is also referred to as the **link-distance**.
- BFS yields shortest (link-distance) paths from a given source vertex to all other vertices.
- BFS processes vertices in increasing order of their distance from the root vertex (given source).
- BFS has running time $O(|V| + |E|)$. 

Finding Shortest Paths in Weighted Graphs

Given a weighted directed graph and a start vertex $u = v_1$:

- The **weight of a path** $p = (v_1, v_2, \ldots, v_k)$ is the sum of the weights of the corresponding edges: $w(p) = \sum_{i=2}^{k} w(v_{i-1}, v_i)$

- The **shortest path weight** from a vertex $u$ to a vertex $v$ is:
  $\delta(u, v) = \begin{cases} 
  \min \{w(p) : p = (u, \ldots, v)\} & \text{if } p \text{ exists} \\
  \infty & \text{else}
  \end{cases}$

- A **shortest path** from $u$ to $v$ is any path $p$ with weight $\delta(u, v)$

Question: How can we find the shortest weight paths from a given source vertex $s$ to all other vertices?
Finding Shortest Paths in Graphs

Can we employ a template similar to BFS, essentially process vertices in increasing distance (shortest path weight) from the source?

Given source $s$ and two vertices $u, v$, s. t. $\delta(s, u) \leq \delta(s, v)$, the algorithm should first process $u$ and then $v$.

Graph:

- $w(\langle a, b, c \rangle) = 3$
- $\delta(a, c) = 3$
- $w(\langle a, b, c, e \rangle) = 6$
- $\delta(a, e) = 6$
Finding Shortest Paths in $G$ with Nonnegative Weights

**Important Observation:** Any subpath of a shortest path must also be a shortest path. Why?

Example: $\langle a, b, c, e \rangle$ is a shortest path. The subpath $\langle a, b, c \rangle$ is also a shortest path.

Finding shortest paths from a source vertex has the greedy choice property. Hence, a greedy algorithm can be employed.

**Observation:** Extending this idea, we observe the existence of a shortest path tree, in which the distance from source $s$ to a vertex is weight of shortest path from $s$ to vertex in $G$. 
The Tree of Shortest Paths

- BFS is efficient in finding all shortest paths from a root because it constructs the tree of shortest paths one edge at a time.

- The tree of shortest paths is a spanning tree of $G = (V, E)$, where the path from its root, the source vertex $s$, to any vertex $u \in V$ is the shortest path $s \rightsquigarrow u$ in $G$.

Template: Grow tree from $s$ outwards one vertex at a time. The vertex $v$ added to the tree is added with the edge that yields the shortest path $s \rightsquigarrow v$. 
Intuition Behind Dijkstra’s Algorithm

- Report the vertices (add them to shortest path tree) in increasing order of their distance from the source vertex.
- Construct the shortest path tree edge by edge, at each step adding one new edge that corresponds to construction of shortest path to the current new vertex.
Dijkstra’s algorithm grows the shortest path tree similarly to how Prim grows the MST of a graph.

In both, the tree grows from $S$ to $V - S$, where $S$ is the set of explored vertices. $S$ keeps vertices that are currently part of the shortest path tree.

In Dijkstra, a vertex $v$ is added to $S$ (extracted from $V - S$) when the shortest path from $s$ to $v$ is found.

Prim maintains an attachment cost of a vertex $v$, which is just the weight of the lightest edge that connects $v$ to $S$.

Dijkstra needs to do the same, but its attachment cost needs to be an estimate of the shortest path $s$ to $v$. 
Essence of Dijkstra’s Algorithm

- Let $d[v]$ denote the current shortest path weight from $s$ to $v$. $d[v]$ is an upper bound on (overestimates) $\delta(s, v)$.
- Lower $d[v]$ values may be found, as new vertices added to $S$ may provide alternative shorter paths to $v$.
- For vertices $u \in S$ (in shortest path tree), $d[u] = \delta(s, u)$.
- For vertices $v \in V - S$ (not yet in shortest path tree), $d[v] \geq \delta(s, v)$.
- $d[v]$ needs to be regularly updated for every vertex $v \in V - S$ discovered as a neighbor of a vertex in $S$.
- The vertex with smallest attachment cost of all other vertices in $V - S$ can be added to $S$. Why?
Proof of Correctness of Dijkstra’s Algorithm

- $u \in S$ means that $d[u] = \delta(s, u)$ (invariant of Dijkstra’s).
- When $u$ was extracted from $V - S$ and added to $S$, its neighboring vertices were analyzed. If $d[u] + w(u, v) < d[v]$, then $d[v]$ gets updated. A shorter path $s \leadsto v$ has been found that passes through $u$.
- All attachments costs of vertices in $V - S$ are compared. If $d[v]$ is smallest, $v$ is extracted and added to $S$.
- Claim: Shortest path $s \leadsto v$ has been found.
- Claim: No other vertex $y \in V - S$ can result in a shorter path to $v$. 
Proof of Correctness of Dijkstra’s Algorithm

**Invariant:** \( \forall u \in S, \ d[u] \) is length of shortest \( s \leadsto u \) path.

**Proof:** (by induction on \(|S|\))

**Base case:** \(|S| = 1\) (trivial).

**Inductive Hypothesis:** Assume invariant holds for \(|S| = k \geq 1\).

- Let \( v \) be next vertex added to \( S \), and let \((u, v)\) be chosen edge.
- Goal is to show that any other \( s \leadsto v \) path is no shorter than \( d[v] \).
- \( P \) is already too long as soon as it leaves \( S \).
- Dijkstra chose \( v \) over \( y \) to extract from \( V - S \), meaning that \( d[v] \leq d[y] \).

\[
\begin{align*}
    w(P) &\geq w(P') + w(x, y) & \text{nonnegative weights} \\
    &\geq d[x] + w(x, y) & \text{inductive hypothesis} \\
    &\geq d[y] & \text{definition of } d[y] \\
    &\geq d[v] & \text{Dijkstra chose } v \text{ over } y
\end{align*}
\]
Towards an Implementation of Dijkstra’s Algorithm

- Maintain a set of explored nodes $S$ for which we have determined the shortest path distance from the source vertex
- Initialize: $S \leftarrow \{s\}$, $d[s] = 0$
- Repeat until $V - S = \emptyset$:
  - choose unexplored vertex $v$ with smallest $d[v]$: 
    $$d[v] = \min_{u \in S} \{d[u] + w(u, v)\}$$
  - update: $S \leftarrow S \cup \{v\}$ and set $\pi[v] \leftarrow u$

**Implementation:**
- Keep $d[v]$ ∀ unexplored vertex $v$
- Priority queue keeps unexplored vertices, prioritized by $d[v]$
- Next vertex to explore (add to $S$) is the one with minimum $d[v]$
Dijkstra’s Algorithm in Pseudocode

**Dijkstra(G, s, w)**

1. $Q \leftarrow V$, $S \leftarrow \emptyset$
2. $d[v] \leftarrow \infty$ for all $v \in V$
3. $d[s] \leftarrow 0$
4. while $Q \neq \emptyset$ do
   5. $u \leftarrow \text{Extract-Min}(Q)$
   6. $S \leftarrow S \cup \{u\}$
   7. for each $v \in \text{Adj}(u)$ do
      8. Relax($u$, $v$, $w$)

**Relax($u$, $v$, $w$)**

1. if $d[v] > d[u] + w(u, v)$ then
2. $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$
Relaxation and Dijkstra’s Algorithm

- For each vertex \( v \), \( d[v] \) maintains an upper bound estimate on the weight of the shortest path from source vertex \( s \) to \( v \).
- \( d[v] \) is referred to as a shortest-path estimate on \( v \).
- Relaxation updates this estimate when coming across new edges \((u, v)\).
- The process of relaxing tests whether one can improve the shortest-path estimate \( d[v] \) by going through the vertex \( u \) in the shortest path from \( s \) to \( v \).
- If \( d[u] + w(u, v) < d[v] \), then \( u \) replaces the predecessor of \( v \).
Dijsktra’s Algorithm in Action

Figure: Graph \( G = (V, E) \)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Initial ( d )</th>
<th>( \pi )</th>
<th>Pass1 ( d )</th>
<th>( \pi )</th>
<th>Pass2 ( d )</th>
<th>( \pi )</th>
<th>Pass3 ( d )</th>
<th>( \pi )</th>
<th>Pass4 ( d )</th>
<th>( \pi )</th>
<th>Pass5 ( d )</th>
<th>( \pi )</th>
<th>Pass6 ( d )</th>
<th>( \pi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>( \infty )</td>
<td></td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>( \infty )</td>
<td>5</td>
<td>B</td>
<td>4</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>( \infty )</td>
<td></td>
<td>( \infty )</td>
<td></td>
<td>6</td>
<td>C</td>
<td>6</td>
<td>C</td>
<td>6</td>
<td>C</td>
<td>6</td>
<td>C</td>
<td>6</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>( \infty )</td>
<td></td>
<td>( \infty )</td>
<td></td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>( \infty )</td>
<td></td>
<td>( \infty )</td>
<td></td>
<td>( \infty )</td>
<td></td>
<td>11</td>
<td>D</td>
<td>9</td>
<td>E</td>
<td>9</td>
<td>E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: Shortest paths from B

Dijkstra’s Algorithm in Action
Class Exercise

Outline of Today’s Class
- Single-source Shortest Paths
- All-pairs Shortest Paths

Dijkstra’s Algorithm
Bellman-Ford’s Algorithm

Class Exercise

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Initial</th>
<th>Pass1</th>
<th>Pass2</th>
<th>Pass3</th>
<th>Pass4</th>
<th>Pass5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
<td>π</td>
<td>d</td>
<td>π</td>
<td>d</td>
<td>π</td>
</tr>
<tr>
<td>b</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- Graph with vertices a, b, c, d, e and edges with weights (a to b: 2, a to d: 5, b to c: 1, c to d: 8, c to e: 3, d to e: 4).

Amarda Shehu
Lecture: Analysis of Algorithms (CS583 - 002)
Analysis of Dijkstra’s Algorithm

**Dijkstra(G, s, w)**

1: \( Q \leftarrow V, S \leftarrow \emptyset \)
2: \( d[v] \leftarrow \infty \) for all \( v \in V \)
3: \( d[s] \leftarrow 0 \)
4: while \( Q \neq \emptyset \) do
5: \( u \leftarrow \text{Extract-Min}(Q) \)
6: \( S \leftarrow S \cup \{u\} \)
7: for each \( v \in \text{Adj}(u) \) do
8: \( \text{Relax}(u, v, w) \)

**Relax(u, v, w)**

1: if \( d[v] > d[u] + w(u, v) \)
2: then
3: \( d[v] \leftarrow d[u] + w(u, v) \)
4: \( \pi[v] \leftarrow u \)

What is the asymptotic behavior of Dijkstra’s algorithm?
What are the costs of Extract-Min and Relax as \( f(|V|, |E|) \)?
Analysis of Dijkstra’s Algorithm

- Updating the heap takes at most $O(lg(|V|))$ time
- The number of updates equals the total number of edges
- So, the total running time is $O(|E| \cdot lg(|V|))$
- As in Prim’s, the running time can be improved depending on the actual implementation of the priority queue
- For example, using a Fibonacci heap yields a total running time of $O(V \cdot lg(|V|) + |E|)$
Analysis of Dijkstra’s Algorithm

Time = \theta(V) \cdot T(\text{Extract} - \text{Min}) + \theta(E) \cdot T(\text{Decrease} - \text{Key})

<table>
<thead>
<tr>
<th>Q</th>
<th>T(\text{Extr.-Min})</th>
<th>T(\text{Decr.-Key})</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>O(</td>
<td>V</td>
<td>)</td>
</tr>
<tr>
<td>Binary heap</td>
<td>O(1)</td>
<td>O(lg</td>
<td>V</td>
</tr>
<tr>
<td>Fib. heap</td>
<td>O(lg</td>
<td>V</td>
<td>)</td>
</tr>
</tbody>
</table>

How does this compare with BFS?
How does BFS get away from a \(lg(|V|)\) factor?
Edsger Dijkstra: 1930-2002

Some Quotes

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture.

In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.
Insights and Shortcomings

**Insight:** All covered graph-search algorithms follow similar template:
- Maintain a set of explored vertices \( S \)
- Grow \( S \) by exploring edges with exactly one endpoint in \( S \)

- **DFS:** Take edge from vertex discovered most recently
- **BFS:** Take edge from vertex discovered least recently
- **Prim:** Take edge of minimum weight
- **Dijkstra:** Take edge to vertex closest to source vertex \( s \)

**Implication:** identifying similar template allows to write reusable code

**Shortcomings:** Negative weights
- Arise in applications
- Make problem intractable in presence of negative cycles
- Easy solution using old algorithms otherwise
What is the Big Deal with Negative Weights?

**Dijkstra:** can fail if there are negative weights.

Dijkstra selects vertex 3 immediately after 0. But shortest path from 0 to 3 is $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$.

**Re-weighting:** Adding a constant to every edge weight can also fail.

Adding 9 to each edge changes the shortest path because it adds 9 to each edge; wrong thing to do for paths with many edges.
What is the Big Deal with Negative Cycles?

Negative weight cycle: sum of weights in cycle is negative.

Observation: If some path from $s$ to $t$ contains a negative weight cycle, then there is no shortest $s \leadsto t$ path. Shortest path can be made arbitrarily shorter by spinning around cycle.
Essence of Bellman-Ford’s Algorithm

- Bellman-Ford solves the single-source shortest path problem in the general case when the graph may have negative weights.
- Gives up trying to find shortest paths when negative cycles are detected.
- Unlike greedy Dijkstra, which depends on the structural assumptions derived from nonnegative weights, an iterative approach is employed that uses Dynamic Programming and so extends to the general case.
- Instead of greedily selecting the minimum-key vertex from the priority queue (not yet processed) to relax, the algorithm relaxes all edges for a certain number of iterations.
- The iterations allow the shortest-path estimates to accurately propagate throughout the graph.
A greedy approach would not work, given negative weights
Bellman-Ford pursues a Dynamics Programming approach
Key realization is that in the absence of negative cycles, any shortest path cannot have more than \(|V| - 1\) edges in it
Let’s introduce a variable that keeps track of the shortest path from a source vertex \(s\) to any vertex \(v\)
\(L(i, v) = \text{length of shortest } s \leadsto v \text{ path that has } \leq i \text{ edges}\)
How does \(L(i, v)\) depend on \(i\)?
**Essence of Bellman-Ford: Dynamic Programming**

**Definition:** \( \mathcal{L}(i, v) = \) length of shortest \( s \leadsto v \) path \( P \) that has \( \leq i \) edges

- Case 1: \( P \) uses at most \( i - 1 \) edges
  - Then \( \mathcal{L}(i, v) = \mathcal{L}(i - 1, v) \)

- Case 2: \( P \) uses exactly \( i \) edges
  - Let \((u, v)\) be last edge in \( P \). Then, *given that the last edge is* \((u, v)\), \( \mathcal{L} \) has to select \( s \leadsto u \) shortest path using at most \( i - 1 \) edges followed by \((u, v)\)

\[
\mathcal{L}(i, v) = \begin{cases} 
0 & \text{if } i = 0 \\
\min_{e=(u, v)} \{ \mathcal{L}(i - 1, v), \mathcal{L}(i - 1, u) + w(e) \} & \text{otherwise}
\end{cases}
\]

**Termin.:** When \( \nexists \) negative cycles, \( \mathcal{L}(n - 1, v) = \) length of shortest \( s \leadsto v \) path
**How are Negative Cycles Detected?**

**Lemma** If $L(n, v) = L(n - 1, v)$ for all vertices $v$, then there are no negative weight cycles in the graph.

**Lemma** If $L(n, v) < L(n - 1, v)$ for some vertex $v$, then any shortest path from $s$ to $v$ contains a negative weight cycle $C$.

**Proof:**

- Since $L(n, v) < L(n - 1, v)$, we know that $P$ has exactly $n$ edges.
- By the pigeonhole principle, $P$ must contain a directed cycle $C$.
- Deleting $C$ yields $s \rightsquigarrow v$ path with $< n$ edges. So, $w(C) < 0$.

**Note:** Any $(\ast, v)$ edge can be that last edge in the shortest path $s \rightsquigarrow v$. That is why the recursion is over the length of the path.
Pseudocode DP Approach

**DP-Approach**\((G, s, \text{weight})\)

1. for each node \(v \in V\) do
2. \(\mathcal{L}[0, v] \leftarrow \infty\)
3. \(\mathcal{L}[0, s] \leftarrow 0\)
4. for \(i = 1\) to \(n - 1\) do
5. for each node \(v \in V\) do
6. \(\mathcal{L}[i, v] \leftarrow \mathcal{L}[i - 1, v]\)
7. for each edge \((u, v) \in E\) do
8. \(\mathcal{L}[i, v] \leftarrow \min\{\mathcal{L}[i - 1, v], \mathcal{L}[i - 1, u] + \text{weight}(u, v)\}\)

**Analysis:** Running Time? Space?
Single-source Shortest Path: Bellman-Ford’s Algorithm

Bellman-Ford(G, s, w)

1: \( d[v] \leftarrow \infty \) for all \( v \in V \)
2: \( d[s] \leftarrow 0 \)
3: for \( i \leftarrow 1 \) to \(|V| - 1\) do
4:   for each edge \((u, v) \in E\) do
5:     Relax\((u, v, w)\)
6: for each edge \((u, v) \in E\) do
7:   if \( d[v] > d[u] + w(u, v) \) then
8:     Negative cycles detected: return FALSE
9: return TRUE

Efficient Implementation No need to maintain the matrix \( L(i, v) \). Maintain only row/iteration with changes. So, use an array \( d[v] \).
Why Bellman-Ford Implements DP Approach?

- Bellman-Ford performs $|V| - 1$ iterations, because the highest number of edges in any shortest path is $|V| - 1$.
- After each iteration $i$, $d[v]$ is no smaller than $L(i, v)$.
- The relaxation updates $d[v]$ for each vertex $v$.
- Essentially, after each iteration $i$, Bellman-Ford answers the question: if shortest path $s \leadsto v$ contains $i$ edges, what would be its weight?
- After $|V| - 1$ iterations, the shortest paths have been found.
- If after one more iteration, the weights can be made shorter, a negative cycle exists.
Bellman-Ford Trace

Graph $G = (V, E)$

Initialization
Bellman-Ford Trace

Order of edge relaxation

Begin of pass 1
Bellman-Ford Trace

Pass 1 continued

Amarda Shehu  Lecture: Analysis of Algorithms (CS583 - 002)
Bellman-Ford Trace

Pass 1 continued

![Bellman-Ford Trace Diagram](image-url)
Bellman-Ford Trace

Pass 1 continued

Pass 1 continued
Bellman-Ford Trace

Pass 1 ends

Pass 2 begins
Bellman-Ford Trace

Pass 2 continued

Pass 2 continued
Outline of Today’s Class
- Single-source Shortest Paths
- All-pairs Shortest Paths
- Dijkstra’s Algorithm
- Bellman-Ford’s Algorithm

Bellman-Ford Trace

Pass 2 continued

Graph showing the Bellman-Ford algorithm in action with a network of nodes and edges, illustrating the shortest path calculations.
Bellman-Ford Trace

Pass 2 continued

Pass 2 continued
Bellman-Ford Trace

Pass 2 continued

Pass 2, 3, 4 end
All-pairs Shortest Paths

- **Problem:** Find shortest paths between all pairs.
- **Assumption:** There are no cycles of 0 or negative weight.
- **Approach:** Dynamic programming, similar to Bellman-Ford, keeps track of intermediate vertices in a path.
Definition: The vertices $v_2, v_3, \ldots, v_{l-1}$ are the intermediate vertices of the path $p = \langle v_1, v_2, \ldots, v_l \rangle$.

- Let $d_{i,j}^{(k)}$ be the weight of shortest path $i \rightsquigarrow j$ s.t. all intermediate vertices (if any) are in set $\{1, 2, \ldots, k\}$.

- Let $D^{(k)}$ be $n \times n$ matrix $[d_{i,j}^{(k)}]$.

- What is $d_{i,j}^{(0)}$?

- Claim: $d_{i,j}^n$ is weight of shortest weight path from $i$ to $j$. Our goal is to compute $D^n$. 
Observation 1: A shortest path does not contain the same vertex twice. Why?

Observation 2: For a shortest path from $i$ to $j$ s.t. that any intermediate vertices on the path are chosen from the set \{1, 2, \ldots, k\}, there are only two possibilities:

- $k$ is not a vertex on the path: Shortest path has weight $d_{i,j}^{(k-1)}$.
- $k$ is a vertex on the path: Shortest path has weight $d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$. 
Structure of Shortest Paths

Consider a shortest path from $i$ to $j$ containing the vertex $k$. It consists of a subpath from $i$ to $k$ and a subpath from $k$ to $j$. Each subpath can only contain intermediate vertices in \{1, \ldots, k - 1\}, and must be as short as possible.

Putting it all together:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\} & k \geq 1 \end{cases}$$
**Floyd-Warshall: The Algorithm**

```
Floyd-Warshall(G, w, n)
1: for i = 1 to n do
2:    for j = 1 to n do
3:       d[i, j] ← w[i, j]
4:       pred[i, j] = NIL
5: for k = 1 to n do
6:    for i = 1 to n do
7:       for j = 1 to n do
8:          if d[i, k] + d[k, j] < d[i, j] then
9:             d[i, j] ← d[i, k] + d[k, j]
10:        pred[i, j] = k
```

- Keep $d_{i,j}^{(k)}$ values in matrix $D^{(k)}$
- Compute $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$
- $D^{(k)} \leftarrow D^{(k-1)}$
- $pred$ can be used to extract paths (how?)
- Storage: $\theta(|V|^2)$
- Time: $\theta(|V|^3)$
Floyd-Warshall: Time and Space Complexity

- **Storage:** $\theta(|V|^2)$
- **Time:** $\theta(|V|^3)$
  
  - Filling each matrix cell requires computing a minimum value by iterating over all $v_k \in V$: $|V|$ iterations are needed to fill each cell
  - Matrix has $|V| \times |V|$ cells
  - Hence, $|V| \times |V| \times |V| = |V|^3$ operations

- Johnson’s algorithm improves time: $O(|V|^2 \cdot \lg(|V|) + |V| \cdot |E|)$
  
  - best on sparse graphs: adjacency list representation
  - uses Bellman-Ford: smart reweighting of negative-weight edges
  - uses Dijkstra: after reweighting, from each vertex
Johnson’s Algorithm

- Add fake source $s$, connected by zero weights to all nodes in graph
- Run Bellman-Ford to get shortest paths from $s$ to any other vertex in $G$.
- Let $h(v)$ be the length of the shortest path from $s$ to $v \in V$
- Reweight all edges of $G$: $w(u, v) \leftarrow w(u, v) + h(u) - h(v)$
- Now run Dijkstra’s from each vertex to find all shortest paths

Figure: Why are edges reweighted this way?