Shortest Paths

March 19\textsuperscript{th}/21\textsuperscript{st}
Outline For Today

1. SSSP in DAGs: DP Algorithm
2. SSSP without Negative Edges: Dijkstra’s Greedy Algorithm
3. SSSP with Negative Edges: Bellman Ford DP Algorithm
4. All pairs Shortest Paths: Floyd Warshall DP Algorithm
Shortest Paths Problems

- Input is $G(V, E)$ with edge weights
- Shortest Paths from a single source $s$ to all/one destination in DAGs (DP Solution)
- Shortest Paths from a single source $s$ to all/one destination in general graphs with no negative edge weights (Dijkstra: Greedy)
- Shortest Paths from all sources to all dests (Floyd Warshall: DP).
Outline For Today

1. **SSSP in DAGs**: DP Algorithm

2. **SSSP without Negative Edges**: Dijkstra’s Greedy Algorithm

3. **SSSP with Negative Edges**: Bellman Ford DP Algorithm

4. **All pairs Shortest Paths**: Floyd Warshall DP Algorithm
Shortest Paths In DAGs

- Input: weighted DAG $G(V, E)$ with arbitrary edge weights and source $s$
  - Note edge weights can be negative
- Output: shortest paths from $s$ to all vertices
Q: Shortest path (distance) from s to e?

A: 6: s->b->e

Let's think of a DP solution.
Defining Subproblems

◆ Recall Linear IS:
  ▪ Line graph was naturally ordered from left to right.
  ▪ Subproblems could be defined as prefix graphs.

◆ Recall Sequence Alignment:
  ▪ X, Y strands were naturally ordered strings.
  ▪ Subproblems could be defined as prefix strings.

◆ Trick: Use the Topological ordering of G and solve shortest paths for “prefix graphs” again.
Defining Subproblems

Let’s solve a larger subproblem in terms of smaller subproblems. For example distance to vertex e:

SD(e) = \min \begin{cases} 
SD(d) + w(d, e) \\
SD(c) + w(c, e) \\
SD(b) + w(b, e)
\end{cases}

Idea: think of the last edge in path s \sim e
In General:

\[ SD(v) = \min_{(u,v) \in E} \left\{ SD(u) + w(u, v) \right\} \]

\( B/c \ G \) is a DAG, we can find shortest paths from left to right!
**SSSP DAG DP**

**procedure** \( \text{ssspDAG}(\text{DAG } G(V, E)) : \)**

1. **topologically sort** \( G \) \( \quad \) \( O(m + n) \)
2. \( \text{let } SD[s] = 0; \ SD[v] = +\infty \)
3. **for** \( v \in G \) **in** topologically sorted order:
   \( SD[v] = \min_{(u,v) \in E} \ SD[u] + w(u, v) \)
4. **return** \( D \)

**Total Runtime:** \( O(n + m) \)

\[ \sum_{u \in V} \text{deg}(u) = m \]

\( O(n + m) : \)

We loop over each vertex exactly once.

We look at each edge exactly once.
Outline For Today

1. SSSP in DAGs: DP Algorithm

2. **SSSP without Negative Edges: Dijkstra’s Greedy Algorithm**

3. SSSP with Negative Edges: Bellman Ford DP Algorithm

4. All pairs Shortest Paths: Floyd Warshall DP Algorithm
SSSP In General Graphs Without Neg. Edges

**Input:** A directed/undirected graph $G(V, E)$:
- $n$ nodes (one is the source), $m$ edges $(u,v)$ and costs $c_{u,v}$

**Output:** For each node $v$ in the graph, shortest $s$-$v$ path.

**Assumption 1:** Graph is connected (all $s$-$v$ paths exist)

**Assumption 2:** Edge costs are non-negative, i.e., $w(u, v) \geq 0$
Shortest Path Example

<table>
<thead>
<tr>
<th>Dst</th>
<th>Path</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>S-&gt;X</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>S-&gt;X-&gt;Y</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>S-&gt;B-&gt;A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>S-&gt;B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>S-&gt;B-&gt;A-&gt;C</td>
<td>4</td>
</tr>
</tbody>
</table>
Shortest Path Example

<table>
<thead>
<tr>
<th>Dst</th>
<th>Path</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>S-&gt;X</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>S-&gt;X-&gt;Y</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>S-&gt;B-&gt;A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>S-&gt;B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>S-&gt;B-&gt;A-&gt;C</td>
<td>4</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm
Dijkstra’s Algorithm
Dijkstra’s Algorithm
Dijkstra’s Algorithm
Dijkstra’s Algorithm
Dijkstra’s Algorithm
Dijkstra’s Algorithm
Dijkstra’s Algorithm
Dijkstra’s Algorithm

A

B

C

D

E

F

G

H

0

1

2

3

4

5

6

7

8

9

10

11
Dijkstra’s Algorithm

procedure dijkstra(G(V,E),s, weights w(u, v)):
L = {s}; R=V-{s}
shortestDistL is an array initialized to null
parent is an array initialized to null
distSoFarR = priority queue of size n
distSoFarR[s] = 0; distSoFarR[v] = +∞ for other v
for i = 1 to n-1:
  let v* = extract-min from distSoFarR
  remove v* from R and add to L
  shortestDistL[v*] = distSoFarR[v*]
  for each (v*, w) s.t. w∈R:
    decrement distSoFarR[w] =
      min{distSoFarR[w], shortestDistL[v*] + w(v*, w)}
    if distSoFarR[w] decreased: set parent[w] = v*
return shortestDist

Run time: O(mlog(n))
Dijkstra’s Correctness (1)

Induction on the # of iterations

Inductive Claim: at each iteration i:

\( \forall v \in L, \text{shortestDist}[v] \text{ is correct (same for parent}[v]) \)

\( \forall v \in R, \text{shortestDist}[v] \text{ is shortest (s, v) path contained in L} \)

(except last edge)

Base Case: L only contains s and shortestDist[s] is 0 and true.

IH: Assume both claims hold for first k v’s in L (i.e., for iteration k)

Let v* be the picked vertex from R in iteration k + 1.

(i.e. distSoFar[v*] was the minimum over all vertices in R)

And let (u*, v*) be the edge that minimized v*’s distSoFar.

So Dijkstra’s path P is:

Claim: P is the shortest path from s to v*!
Consider any other path $P'$. 

Only in $L$, $P'$ crosses to $R$ an arbitrary path to $v^*$.

\[
\text{cost}(P') = (\geq \text{shortestDist}[l_1]) + w(l_1,r_1) + (\geq 0 \text{ cost})
\]

\[\geq \text{distSoFar}[r_1] \geq \text{distSoFar}[v^*] = \text{cost}(P)\]  

Q.E.D
Outline For Today

1. SSSP in DAGs: DP Algorithm
2. SSSP without Negative Edges: Dijkstra’s Greedy Algorithm
3. SSSP with Negative Edges: Bellman Ford DP Algorithm
4. All pairs Shortest Paths: Floyd Warshall DP Algorithm
Pros/Cons of Dijkstra’s Algorithm

Pros: $O(m \log n)$ super fast & simple algorithm

Cons:

1. Works only if $c_e \geq 0$
   - Sometimes need negative weights, e.g. (finance)

2. Not parallelizable:
   - Looks very “serial”

Bellman-Ford addresses both of these drawbacks
Preliminary: Negative Weight Cycles

Question: How to define shortest paths when G has negative weights cycles?
Possible Shortest $s \rightarrow v$ Path Definition 1

Shortest path from $s$ to $v$ with cycles allowed.

Problem: Can loop forever in a negative cycle.

So there is no "shortest path"!
Shortest path from $s$ to $v$, cycles NOT allowed.

Q: Now, can shortest paths contain cycles?

A: No. Assume P is shortest path from s to v, with a cycle.

P = s \rightarrow x \rightarrow x \rightarrow v. Then since x \rightarrow x is a cycle, and we assumed no negative weight cycles, we could get P` = s \rightarrow x \rightarrow v, and get a shorter path.
Upshot: Bellman-Ford’s Properties

Note: Bellman-Ford will be able to detect if there is a negative weight cycle!

Both outputs computed in reasonable amount of time.
Challenge of A DP Approach

Need to identify some sub-problems.

- **Linear IS:**
  - Line graph was naturally ordered from left to right.
  - Subproblems could be defined as prefix graphs.

- **Sequence Alignment:**
  - X, Y strands were naturally ordered strings.
  - Subproblems could be defined as prefix strings.

**Shortest Paths’ Input G Has No Natural Ordering**
High-level Idea Of Subproblems

But the Output Is Paths & Paths Are Sequential!

Trick: Impose an Ordering Not On G but on Paths.

Larger Paths Will Be Derived By Appending New Edges To The Ends Of Smaller (Shorter) Paths.
Subproblems

Input: G(V, E) no negative cycles, s, c_e arbitrary weights.
Output: ∀v, global shortest paths from s to v.

Q: Max possible # hops (or # edges) on the shortest paths?

A: n-1 (**since there are no negative cycles**)

\[ P_{(v, i)} = \text{Shortest path from } s \text{ to } v \text{ with at most } i \text{ edges (\& no cycles)}. \]
Let $P_{(v, i)}$ be the shortest $s$-$v$ path with $\leq i$ edges.

Q: $P_{(t, 1)}$?

A: Does not exist (assume such paths have $\infty$ weights.)
Let $P_{(v, i)}$ be the shortest $s$-$v$ path with $\leq i$ edges.

**Q:** $P_{(t, 2)}$?

**A:** $s \rightarrow x \rightarrow t$ with weight 2.
Let $P_{(v, i)}$ be the shortest s-v path with $\leq i$ edges.

Q: $P_{(t, 3)}$?

A: $s\to w\to z\to t$ with weight -1.
Let $P_{(v, i)}$ be the shortest $s$-$v$ path with $\leq i$ edges.

Q: $P_{(t,4)}$?

A: $s$-$\rightarrow$ $w$-$\rightarrow$ $z$-$\rightarrow$ $t$ with weight $-1$. 
Let $P = P_{(v, i)}$ be the shortest $s-v$ path with $\leq i$ edges.

Note: For some $v$, an $s-v$ path with $\leq i$ edges may not exist. Assume $v$ has such a path.

A Claim that does not require a proof:

$|P| \leq i-1$ OR $|P| = i$
Case 1: $|P=P_{(v, i)}| \leq i-1$

Q: What can we assert about $P_{(v, i-1)}$?

A: $P_{(v, i-1)} = P_{(v, i)}$ (by contradiction)

($P_{(v, i)}$ is also the shortest $s \leadsto v$ path with at most $i-1$ edges)
Case 2: $|P = P_{(v, i)}| = i$

Q: What can we assert about $P`$?

Claim: $P` = P_{(u, i-1)}$

($P`$ is shortest $s \leadsto u$ path in with $\leq i-1$ edges)
Proof that $P` = P_{(u, i-1)}$

Assume $\exists$ a better $s \Rightarrow u$ path $Q$ with $\leq i-1$ edges

$Q$ had $\leq i-1$ edges, then $Q \cup (u,v)$ has $\leq i$ edges.

cost($Q$) $< \text{cost}(P`)$, cost($Q \cup (u,v)$) $< \text{cost}(P)$.

Q.E.D
Summary of the 2 Cases

Case 1: $|P_{(v, i)}| \leq i-1 \Rightarrow P_{(v, i-1)} = P_{(v, i)}$

Case 2: $|P_{(v, i)}| = i \Rightarrow P^` = P_{(u, i-1)}$

$P^` \Rightarrow |s \sim u| = i-1$
∀ v, and for i=\{1, \ldots, n\}

\( P_{(v, i)} \): shortest \( s \leadsto v \) path with \( \leq i \) edges (or null)

\( L_{(v, i)} \): \( w(P_{(v, i)}) \) (and \( +\infty \) for null paths)

\[
L_{(v, i)} = \min \left\{ L_{(v, i-1)} \right. \\
\left. \min_{u: \exists (u,v) \in E} : L_{(u, i-1)} + c_{(u,v)} \right\}
\]
Bellman-Ford Algorithm

$L_{(v, i)}: w(P_{(v, i)})$

Let A be an nxn 2D array.

$A[i][v] = \text{shortest path to vertex } v \text{ with } \leq i \text{ edges.}$

**procedure** Bellman-Ford(G(V,E), weights C):
Base Cases: $A[0][s] =$
Bellman-Ford Algorithm

$L_{(v, i)}: w(P_{(v, i)})$

Let A be an nxn 2D array.

$A[i][v] = \text{shortest path to vertex } v \text{ with } \leq i \text{ edges.}$

**procedure** Bellman-Ford(G(V,E), weights C):

Base Cases: $A[0][s] = 0$

$A[0][j] =$
Bellman-Ford Algorithm

\[ L_{(v, i)}: w(P_{(v, i)}) \]

Let A be an nxn 2D array.

\[ A[i][v] = \text{shortest path to vertex } v \text{ with } \leq i \text{ edges.} \]

**procedure** Bellman-Ford(G(V,E), weights C):

Base Cases: \( A[0][s] = 0 \)
\[
A[0][j] = \infty \text{ where } j \neq s
\]

for \( i = 1, \ldots, n-1: \)

for \( v \in V: \)

\[
A[i][v] = \min \{ A[i-1][v] \}
\]
\[
\min_{(u, v) \in E} A[i-1][u] + c_{(u, v)}
\]
Correctness of BF

By induction on $i$ and correctness of the recurrence for $L_{(v, i)}$ (exercise)
# entries in $A$ is $n^2$.

**Q**: How much time for computing each $A[i][v]$?

**A**: in-deg($v$)

For each $i$, total work for all $A[i][v]$ entries is: $\sum_{v \in V} \text{in-deg}(v)$

**Total Runtime**: $O(nm)$
Suppose for some $i \leq n$:

$$A[i][v] = A[i-1][v] \quad \forall \ v.$$ 

Q: What does this mean?

A: Values will not change in any later iteration

=>$\text{ We can stop!}$

... 

for $i = 1, \ldots, n-1$: 

for $v \in V$:

$$A[i][v] = \min \{ A[i-1][v] \min_{(u,v) \in E} A[i-1][u] + c(u,v) \}$$ 

Values only depend on the previous iteration!
Consider any graph $G(V, E)$ with arbitrary edge weights. 

$\Rightarrow$ There may be negative cycles.

Claim: If BF stabilizes at some iteration $i > 0$, then

$G$ has no negative cycles.

(i.e., negative cycles implies BF never stabilizes!)
How To Check For Cycles If Claim is True

Run BF just one extra iteration!
If so, no negative cycles, o.w. there is a negative cycle.

Running $n$ iterations is the general form of BF:

$G(V, E)$, weights $\rightarrow$ Bellman-Ford

$\exists$ negative-weight cycle

Shortest Paths
Proof of Claim:
BF Stabilizes => G has no negative cycles

Assume BF has stabilized in iteration $i$.


$$A[i][v] = \min \{ A[i-1][v], \min_{(u,v) \in E} A[i-1][u] + c_{(u,v)} \}$$

$$d(v) = \min \{ d(v), \min_{(u,v) \in E} d(u) + c_{(u,v)} \}$$

$$d(v) \leq d(u) + c_{(u,v)}$$

Let’s argue that every cycle $C$ has non-negative weight…
Proof of Claim (continued)

\[ d(v) \leq d(u) + c_{(u,v)} \]

Fix a cycle \( C \):

\[ y \rightarrow z \rightarrow t \leftarrow x \]
Proof of Claim (continued)

\[ d(v) \leq d(u) + c_{(u,v)} \Rightarrow d(v) - d(u) \leq c_{(u,v)} \]

Fix a cycle C:
Proof of Claim (continued)

\[ d(v) \leq d(u) + c_{(u,v)} \implies d(v) - d(u) \leq c_{(u,v)} \]

Fix a cycle \( C \):

\[ d(x) - d(t) \leq c_{(t,x)} \]
Proof of Claim (continued)

d(v) \leq d(u) + c_{(u,v)} \Rightarrow d(v) - d(u) \leq c_{(u,v)}

Fix a cycle C:

\[ d(x) - d(t) \leq c_{(t,x)} \]
\[ d(y) - d(x) \leq c_{(x,y)} \]
Proof of Claim (continued)

\[ d(v) \leq d(u) + c_{(u,v)} \Rightarrow d(v) - d(u) \leq c_{(u,v)} \]

Fix a cycle C:

\[
\begin{align*}
    d(x) - d(t) &\leq c_{(t,x)} \\
    d(y) - d(x) &\leq c_{(x,y)} \\
    d(z) - d(y) &\leq c_{(y,z)}
\end{align*}
\]
Proof of Claim (continued)

\[ d(v) \leq d(u) + c_{(u,v)} \Rightarrow d(v) - d(u) \leq c_{(u,v)} \]

Fix a cycle \( C \):

\[
\begin{align*}
    d(x) - d(t) & \leq c_{(t,x)} \\
    d(y) - d(x) & \leq c_{(x,y)} \\
    d(z) - d(y) & \leq c_{(y,z)} \\
    d(t) - d(z) & \leq c_{(z,t)} \\
\end{align*}
\]

Same algebra and result for any cycle (exercise).

Q.E.D.
For $i = 1, \ldots, n-1$:

$$A[i][v] = \min \{ A[i-1][v], \min_{(u,v) \in E} A[i-1][u] + c(u,v) \}$$

...
Space Optimization (1)

Fix: Each \( v \) stores a predecessor pointer (initially null)

Whenever \( A[i][v] \) is updated to \( A[i-1][u]+c_{(u,v)} \), we set the \( \text{Pred}[v] \) to \( u \).

Claim: At termination, tracing pointers back from \( v \) yields the shortest \( s-v \) path.

(Details in the book, by induction on \( i \))

\[
\text{for } i = 1, ..., n-1:\n\text{for } v \in V:\nA[i][v] = \min \{ A[i-1][v], \min_{(u,v) \in E} A[i-1][u]+c_{(u,v)} \}\]
Summary of BF

Runtime: $O(nm)$, not as fast as Dijkstra’s $O(m \log n)$.  
But works with negative weight edges. 
And is distributable/parallelizable. 

*Might see its distributed version last lecture of class.*
Outline For Today

1. SSSP in DAGs: DP Algorithm
2. SSSP without Negative Edges: Dijkstra’s Greedy Algorithm
3. SSSP with Negative Edges: Bellman Ford DP Algorithm
4. All pairs Shortest Paths: Floyd Warshall DP Algorithm
All-Pairs Shortest Paths (APSP)

Input: Directed $G(V, E)$, arbitrary edge weights.

Output: $\forall u, v, d(u, v)$: shortest $(u, v)$ path in $G$.

(no fixed source $s$)

Q: What’s a lower-bound to solve APSP?

A: $O(n^2)$ b/c there are $O(n^2)$ outputs
Note: Floyd-Warshall will be able to detect if there is a negative weight cycle!

G(V, E), weights → Floyd-Warshall → ∃ negative-weight cycle

Both outputs computed in asymptotically the same amount of time.
Floyd-Warshall Idea

Linear IS: input graph naturally ordered sequentially

Seq. Alignment: strings naturally ordered sequentially

SSSP in DAGs: topological ordering

FW imposes sequentiality on the vertices

⇒ order vertices from 1 to n

⇒ only use the first i vertices in each subproblem

(Same idea works for SSSP, but not very efficient)
Floyd-Warshall Subproblems

\[ V = \{1, \ldots, n\}, \text{ordered completely arbitrarily} \]
\[ V^k = \{1, \ldots, k\} \]

Original Problem: \( \forall (u, v) \) shortest \( u, v \) path.

We need to define the subproblems.

Subproblem \( P_{(i, j, k)} \) = shortest \( i, j \) path that uses only \( V^k \) as intermediate nodes (excluding \( i \) and \( j \)).
Floyd-Warshall Subproblems

Q: $P_{(6,4,1)}$?

A: null (weight of $+\infty$)
Q: $P_{(2,4,1)}$?
A: 2->1->4 (weight of 2)
Q: $P_{(2,6,0)}$?

A: 2-6 (weight of 2)

(no intermediate nodes needed)
Floyd-Warshall Subproblems

Q: $P_{(2,6,1)}$?
A: 2-6 (still weight of 2) (without intermediate nodes)
Floyd-Warshall Subproblems

Q: $P_{(2,6,2)}$?

A: 2-6 (still weight of 2) (without intermediate nodes)
Floyd-Warshall Subproblems

Q: \( P_{(2,6,3)} \)?

A: 2->3->6 (weight of 0) (now with intermediate node 3)
Floyd-Warshall Subproblems

Final shortest \(i \sim j\) path is \(P_{(i, j, n)}\) when we’re allowed to use any vertices as intermediate nodes.
Fix source $i$, and destination $j$. Consider $P_{(i, j, k)}$:

$$k \notin P_{(i, j, k)} \text{ OR } k \in P_{(i, j, k)}$$

(either one of the intermediate vertices is $k$ or it’s not)
Case 1: $k \notin P_{(i, j, k)}$

Then all internal nodes are from 1,...,k-1.

Q: What can we assert about $P_{(i, j, k)}$?

A: $P_{(i, j, k)} = P_{(i, j, k-1)}$

(proof by contradiction)
Case 2: $k \in P_{(i, j, k)}$

Q: What can we assert about $P_1$ and $P_2$?

one (and only one) of the int. nodes is $k$. (why only one?)

A1: $P_1 \& P_2$ only contain int. nodes $1, \ldots, k-1$

A2: $P_1 = P_{(i,k,k-1)} \& P_2 = P_{(k,j,k-1)}$

(proof by contradiction)
Summary of the 2 Cases

Case 1: $k \notin P_{(i, j, k)} \Rightarrow P_{(i, j, k)} = P_{(i, j, k-1)}$

Case 2: $k \in P_{(i, j, k)} \Rightarrow P_1 = P_{(i,k,k-1)} \& P_2 = P_{(k,j,k-1)}$
∀ i, j, k and where i, j, k = \{1, ..., n\}

\( P_{(i, j, k)} \): shortest \( i \sim j \) path with all intermediate
nodes from \( V^k = \{1, ..., k\} \) (or null)

\( L_{(i, j, k)} \): \( w(P_{(i, j, k)}) \) (and \( +\infty \) for null paths)

\[ L_{(i, j, k)} = \min \left\{ L_{(i, j, k-1)}, L_{(i, k, k-1)} + L_{(k, j, k-1)} \right\} \]

With appropriate
base cases.
Floyd-Warshall Algorithm

Let A be an nxnxn 3D array.

\[ A[i][j][k] = \text{shortest } i \rightarrow j \text{ path with } V^k \text{ as intermediate nodes} \]

**procedure** Floyd-Warshall(G(V,E), weights C):
Base Cases: \( A[i][i][\emptyset] \)
Floyd-Warshall Algorithm

Let $A$ be an $nxnxn$ 3D array.

$A[i][j][k] = \text{shortest } i \rightarrow j \text{ path with } V^k \text{ as intermediate nodes}$

**procedure** Floyd-Warshall($G(V,E)$, weights $C$):

Base Cases: $A[i][i][0] = 0$

$A[i][j][0] = \text{previous value}$
Floyd-Warshall Algorithm

Let $A$ be an $n \times n \times n$ 3D array.

$A[i][j][k] = \text{shortest } i \rightarrow j \text{ path with } V^k \text{ as intermediate nodes}$

**procedure** Floyd-Warshall($G(V,E)$, weights $C$):

Base Cases: $A[i][i][0] = 0$

$A[i][j][0] = C_{i,j} \text{ if } (i,j) \in E$

$+\infty \text{ if } (i,j) \notin E$

for $k = 1, \ldots, n$:

for $i = 1, \ldots, n$:

for $j = 1, \ldots, n$:

$A[i][j][k] = \min \{A[i][j][k-1], A[i][k][k-1] + A[k][j][k-1]\}$
Correctness & Runtime

Correctness: induction on i,j,k & correctness of recurrence
Runtime: $O(n^3)$ (b/c $n^3$ subproblems, $O(1)$ for each one)

procedure Floyd-Warshall(G(V,E), weights C):
  Base Cases: $A[i][i][0] = 0$
  $A[i][j][0] = C_{i,j}$ if $(i,j) \in E$
  $+\infty$ if $(i,j) \notin E$

  for $k = 1, \ldots, n$:
    for $i = 1, \ldots, n$:
      for $j = 1, \ldots, n$:
        $A[i][j][k] = \min \{A[i][j][k-1],$
        $A[i][k][k-1] + A[k][j][k-1]\}$
Detecting Negative Cycles

*Just check the $A[i][i][n]$ for each $i$!* 

Let $C$ be a negative cycle with $l$ the largest ID vertex on $C$ 

$\Rightarrow$ for any vertex $j$ on $C$, $A[j][j][l] \leq 0$ 

$\Rightarrow$ therefore $A[j][j][n]$ will be negative
As Promised

\[ G(V, E), \text{ weights} \rightarrow \text{Floyd-Warshall} \]

\[ \exists \text{ negative-weight cycle} \]

\[ \text{All Pairs} \]
\[ \text{Shortest Paths} \]
Path Reconstruction

Keep successors for each $i \ j$ path in an array $S[i][j]$. Initially, $S[i][j] = \text{null}$ or $j$ if $(i,j)$ exists.


E.g: Suppose at termination $S[i][j] = w$. Then we look at $S[w][j] = z$ Then we look at $S[z][j] \ldots$ until we hit $j$. 
## SSSP DAG, Dijkstra, FW

<table>
<thead>
<tr>
<th></th>
<th>SSSP DAG</th>
<th>Dijkstra</th>
<th>Bellman-Ford</th>
<th>FW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single-Source / All Pairs</strong></td>
<td>Single-Source</td>
<td>Single-Source</td>
<td>Single Source</td>
<td>All Pairs</td>
</tr>
<tr>
<td><strong>Run-time</strong></td>
<td>(O(n + m))</td>
<td>(O(m \log(n)))</td>
<td>(O(mn))</td>
<td>(O(n^3))</td>
</tr>
<tr>
<td><strong>Negative Edges</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Negative Cycles</strong></td>
<td>No</td>
<td>No</td>
<td>No, but can detect</td>
<td>No, but can detect</td>
</tr>
</tbody>
</table>
Next Week: Intractability, P vs NP &
What to Do for NP-hard Problems?
Especially don’t miss the first lecture!