Shortest paths: Floyd-Warshall

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

October 15, 2017

Shortest paths: All pairs, negative weights

General all-pairs shortest paths problem

- Saw: Dijkstra, solving:
  - Compute shortest paths from single source to all destinations
  - Directed graphs, non-negative weights
- Complications today:
  1. Negative weights
  2. Computing all distances (from all, to all)
- Algorithm: Floyd-Warshall
  - Mystical procedure
  - Solution category: Dynamic programming

General all-pairs shortest paths problem

- Saw: Dijkstra, solving:
  - Compute shortest paths from single source to all destinations
  - Directed graphs, non-negative weights
- Complications today:
  1. Negative weights
  2. Computing all distances (from all, to all)
- Algorithm: Floyd-Warshall
  - Mystical procedure
  - Solution category: Dynamic programming

General all-pairs shortest paths problem

- Saw: Dijkstra, solving:
  - Compute shortest paths from single source to all destinations
  - Directed graphs, non-negative weights
- Complications today:
  1. Negative weights
  2. Computing all distances (from all, to all)
- Algorithm: Floyd-Warshall
  - Mystical procedure
  - Solution category: Dynamic programming

General all-pairs shortest paths problem

- Saw: Dijkstra, solving:
  - Compute shortest paths from single source to all destinations
  - Directed graphs, non-negative weights
- Complications today:
  1. Negative weights
  2. Computing all distances (from all, to all)
- Algorithm: Floyd-Warshall
  - Mystical procedure
  - Solution category: Dynamic programming

Negative weights in graph distances

Negative weights

- Negative weights make shortest paths question more complex
- Beneficial detours
  - If an arc has negative weight, we may gain by taking a detour to include it
- “Improving” (negative-weight) cycles
  - If going around a cycle $v_1 v_2 \dots v_n v_1$ has negative total weight, there is no sensible “shortest” solution
  - We may always take one more pass around the cycle for an even cheaper passage
  - Any path that can go through negative cycle treated as $-\infty$ cost (can be made as low value as you want)
- Can still sensibly compute shortest paths when no such cycle exists

Negative weights

- Negative weights make shortest paths question more complex
- Beneficial detours
  - If an arc has negative weight, we may gain by taking a detour to include it
- "Improving" (negative-weight) cycles
  - If going around a cycle $v_1 v_2 \ldots v_n v_1$ has negative total weight, there is no sensible "shortest" solution
  - We may always take one more pass around the cycle for an even cheaper passage
  - Any path that can go through negative cycle treated as $-\infty$ cost (can be made as low value as you want)
- Can still sensibly compute shortest paths when no such cycle exists

Negative weights

- Negative weights make shortest paths question more complex
- Beneficial detours
  - If an arc has negative weight, we may gain by taking a detour to include it
- "Improving" (negative-weight) cycles
  - If going around a cycle $v_1 v_2 \ldots v_n v_1$ has negative total weight, there is no sensible "shortest" solution
  - We may always take one more pass around the cycle for an even cheaper passage
  - Any path that can go through negative cycle treated as $-\infty$ cost (can be made as low value as you want)
- Can still sensibly compute shortest paths when no such cycle exists

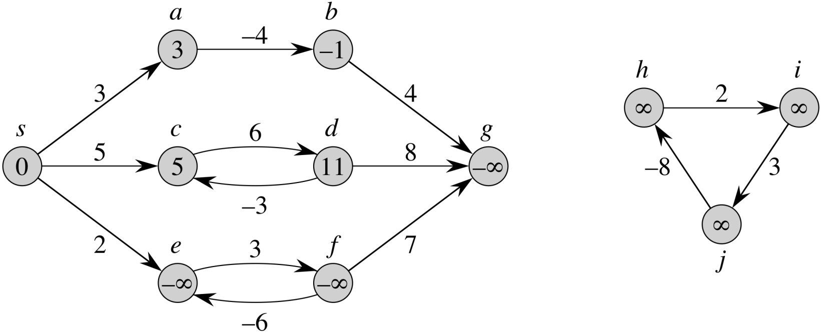
Negative weights

- Negative weights make shortest paths question more complex
- Beneficial detours
  - If an arc has negative weight, we may gain by taking a detour to include it
- "Improving" (negative-weight) cycles
  - If going around a cycle $v_1 v_2 \ldots v_n v_1$ has negative total weight, there is no sensible "shortest" solution
  - We may always take one more pass around the cycle for an even cheaper passage
  - Any path that can go through negative cycle treated as $-\infty$ cost (can be made as low value as you want)
- Can still sensibly compute shortest paths when no such cycle exists

Negative weights

- Negative weights make shortest paths question more complex
- Beneficial detours
  - If an arc has negative weight, we may gain by taking a detour to include it
- “Improving” (negative-weight) cycles
  - If going around a cycle $v_1 v_2 \dots v_n v_1$ has negative total weight, there is no sensible “shortest” solution
  - We may always take one more pass around the cycle for an even cheaper passage
  - Any path that can go through negative cycle treated as $-\infty$ cost (can be made as low value as you want)
- Can still sensibly compute shortest paths when no such cycle exists

Illustration: Single-source, negative weights



Cycle efe is negative cycle, gets distance $-\infty$
Cycle cdc has negative arc, but positive weight
Vertex g inherits distance $-\infty$ from f

Negative weights: Situations

1. Graph has no negative cycle at all (but negative weights)
   - Shortest paths computed by reasonable algorithms
2. Graph has negative cycle, but not reachable from u or to v
   - We can still hope to compute a sensible path from u to v
3. There is a negative cycle on u or v,
   or passable on the way from u to v
   - Shortest path must be given as $-\infty$
4. Will focus on graphs with no negative cycle (or on the negative cycle detection problem)

Negative weights: Situations

1. Graph has no negative cycle at all (but negative weights)
   - Shortest paths computed by reasonable algorithms
2. Graph has negative cycle, but not reachable from u or to v
   - We can still hope to compute a sensible path from u to v
3. There is a negative cycle on u or v,
   or passable on the way from u to v
   - Shortest path must be given as $-\infty$
4. Will focus on graphs with no negative cycle (or on the negative cycle detection problem)

Negative weights: Situations

1. Graph has no negative cycle at all (but negative weights)
   - Shortest paths computed by reasonable algorithms
2. Graph has negative cycle, but not reachable from u or to v
   - We can still hope to compute a sensible path from u to v
3. There is a negative cycle on u or v,
   or passable on the way from u to v
   - Shortest path must be given as $-\infty$
4. Will focus on graphs with no negative cycle (or on the negative cycle detection problem)

Negative weights: Situations

1. Graph has no negative cycle at all (but negative weights)
   - Shortest paths computed by reasonable algorithms
2. Graph has negative cycle, but not reachable from u or to v
   - We can still hope to compute a sensible path from u to v
3. There is a negative cycle on u or v,
   or passable on the way from u to v
   - Shortest path must be given as $-\infty$
4. Will focus on graphs with no negative cycle (or on the negative cycle detection problem)

Observations

1. Negative weights are a real obstacle – can't handle by alternative bookkeeping
   - For example, adding weight to arcs until all weights are positive warps and destroys the shortest path situation
2. Dijkstra's algorithm is essentially unpatchable
   - A central notion is marked vertices – we are done with a vertex as soon as we have visited it (no more edge updates can occur)
   - With negative weights, an improving update to v (finding a shorter path) can occur by going through vertices that are further away than v
   - Would need to keep updating until it stabilises
3. The problems we investigate:
   - Compute negative cycle in graph
   - Compute shortest paths where there are no negative cycles

Observations

1. Negative weights are a real obstacle – can't handle by alternative bookkeeping
   - For example, adding weight to arcs until all weights are positive warps and destroys the shortest path situation
2. Dijkstra's algorithm is essentially unpatchable
   - A central notion is marked vertices – we are done with a vertex as soon as we have visited it (no more edge updates can occur)
   - With negative weights, an improving update to v (finding a shorter path) can occur by going through vertices that are further away than v
   - Would need to keep updating until it stabilises
3. The problems we investigate:
   - Compute negative cycle in graph
   - Compute shortest paths where there are no negative cycles

Observations

1. Negative weights are a real obstacle – can't handle by alternative bookkeeping
   - For example, adding weight to arcs until all weights are positive warps and destroys the shortest path situation
2. Dijkstra's algorithm is essentially unpatchable
   - A central notion is marked vertices – we are done with a vertex as soon as we have visited it (no more edge updates can occur)
   - With negative weights, an improving update to v (finding a shorter path) can occur by going through vertices that are further away than v
   - Would need to keep updating until it stabilises
3. The problems we investigate:
   - Compute negative cycle in graph
   - Compute shortest paths where there are no negative cycles

Observations

1. Negative weights are a real obstacle – can't handle by alternative bookkeeping
    - For example, adding weight to arcs until all weights are positive warps and destroys the shortest path situation
2. Dijkstra's algorithm is essentially unpatchable
    - A central notion is marked vertices – we are done with a vertex as soon as we have visited it (no more edge updates can occur)
    - With negative weights, an improving update to v (finding a shorter path) can occur by going through vertices that are further away than v
    - Would need to keep updating until it stabilises
3. The problems we investigate:
    - Compute negative cycle in graph
    - Compute shortest paths where there are no negative cycles

All-pairs shortest paths problem

All-pairs shortest paths: Representation

- Want to know all distances between pairs of vertices
- Representation (e.g.) array distance[u][v]
- Note – $\Omega(n^2)$ data
  - Unavoidable: Consider complete graph with edge weights
  - Takes $\Theta(n^2)$ numbers to store
- To recreate the paths, would also need parentOf data
  - Example: array previousNode[u][w]=v records that on the shortest path from u to w, the last node before w is v
  - Like parentOf – basically, previousNode[u][*] will encode a single-source tree rooted in u
- Will focus on distances, not to overload with complications

All-pairs shortest paths: Representation

- Want to know all distances between pairs of vertices
- Representation (e.g.) array distance[u][v]
- Note – $\Omega(n^2)$ data
  - Unavoidable: Consider complete graph with edge weights
  - Takes $\Theta(n^2)$ numbers to store
- To recreate the paths, would also need parentOf data
  - Example: array previousNode[u][w]=v records that on the shortest path from u to w, the last node before w is v
  - Like parentOf – basically, previousNode[u][*] will encode a single-source tree rooted in u
- Will focus on distances, not to overload with complications

All-pairs shortest paths: Representation

- Want to know all distances between pairs of vertices
- Representation (e.g.) array distance[u][v]
- Note – $\Omega(n^2)$ data
  - Unavoidable: Consider complete graph with edge weights
  - Takes $\Theta(n^2)$ numbers to store
- To recreate the paths, would also need parentOf data
  - Example: array previousNode[u][w]=v records that on the shortest path from u to w, the last node before w is v
  - Like parentOf – basically, previousNode[u][*] will encode a single-source tree rooted in u
- Will focus on distances, not to overload with complications

All-pairs shortest paths: Representation

- Want to know all distances between pairs of vertices
- Representation (e.g.) array distance[u][v]
- Note – $\Omega(n^2)$ data
  - Unavoidable: Consider complete graph with edge weights
  - Takes $\Theta(n^2)$ numbers to store
- To recreate the paths, would also need parentOf data
  - Example: array previousNode[u][w]=v records that on the shortest path from u to w, the last node before w is v
  - Like parentOf – basically, previousNode[u][*] will encode a single-source tree rooted in u
- Will focus on distances, not to overload with complications

Computing all-pairs shortest paths

- Simple solution: Compute shortest paths from v, for every vertex v
  - Non-negative weights: May use Dijkstra, time $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$
  - Negative weights single-source paths: More expensive to compute, but algorithm exists (total time $\mathcal{O}(|V|^2 \cdot |E|)$)
- Will see faster algorithms, making gains by computing all distances in parallel
  1. Simple, slower mock-up algorithm
  2. Floyd-Warshall

Preparation: A simple mock-up suggestion

- Let's fill out a large table distance(u,v,d), storing the shortest path from u to v with at most d steps
- Fill out by induction: First steps easy
  1. When d=0: distance(u,u,0)=0, otherwise distance(u,v,0)=∞, u $\neq$ v
  2. When d=1: distance(u,v,1)=weight(uv) if u $\neq$ v and the arc uv exists
- Future steps build on past steps
  - distance(u,v,d+1) = min(distance(u,v,d), min distance(u,w,d)+weight(wv)), over all arcs wv
- One iteration: There are n^2 pairs to fill in, each pair requires looking at $\mathcal{O}(n)$ further values
- Total time $\Theta(n^3)$ to complete one iteration

Preparation: A simple mock-up suggestion

- Let's fill out a large table distance(u,v,d), storing the shortest path from u to v with at most d steps
- Fill out by induction: First steps easy
  1. When d=0: distance(u,u,0)=0, otherwise distance(u,v,0)=∞, u $\neq$ v
  2. When d=1: distance(u,v,1)=weight(uv) if u $\neq$ v and the arc uv exists
- Future steps build on past steps
  - distance(u,v,d+1) = min(distance(u,v,d), min distance(u,w,d)+weight(wv)), over all arcs wv
- One iteration: There are n^2 pairs to fill in, each pair requires looking at $\mathcal{O}(n)$ further values
- Total time $\Theta(n^3)$ to complete one iteration

Preparation: A simple mock-up suggestion

- Let's fill out a large table distance(u,v,d), storing the shortest path from u to v with at most d steps
- Fill out by induction: First steps easy
  1. When d=0: distance(u,u,0)=0, otherwise distance(u,v,0)=∞, u $\neq$ v
  2. When d=1: distance(u,v,1)=weight(uv) if u $\neq$ v and the arc uv exists
- Future steps build on past steps
  - distance(u,v,d+1) = min(distance(u,v,d),
    min distance(u,w,d)+weight(wv)), over all arcs *wv*
- One iteration: There are n^2 pairs to fill in, each pair requires looking at $\mathcal{O}(n)$ further values
- Total time $\Theta(n^3)$ to complete one iteration

Preparation: A simple mock-up suggestion

- Let's fill out a large table distance(u,v,d), storing the shortest path from u to v with at most d steps
- Fill out by induction: First steps easy
  1. When d=0: distance(u,u,0)=0, otherwise distance(u,v,0)=∞, u $\neq$ v
  2. When d=1: distance(u,v,1)=weight(uv) if u $\neq$ v and the arc uv exists
- Future steps build on past steps
  - distance(u,v,d+1) = min(distance(u,v,d),
    min distance(u,w,d)+weight(wv)), over all arcs *wv*
- One iteration: There are n^2 pairs to fill in, each pair requires looking at $\mathcal{O}(n)$ further values
- Total time $\Theta(n^3)$ to complete one iteration

Completing and using the distance info

- Observe: Every sensible path has length at most n
- Therefore
  - After n iterations, time $\mathcal{O}(n^4)$, all sensible paths have been found
  - The only paths still improving after $> n$ steps must contain a cycle
- Detects negative cycles:
  - distance(u,u,d)< 0 for some $d \leq n$, vertex u if and only if there is a negative cycle
- If there are no negative cycles,

$$\text{distance(u,v,n)}$$

contains the shortest path data for all pairs of vertices u, v.

Completing and using the distance info

- Observe: Every sensible path has length at most n
- Therefore
  - After n iterations, time $\mathcal{O}(n^4)$, all sensible paths have been found
  - The only paths still improving after $> n$ steps must contain a cycle
- Detects negative cycles:
  - distance(u,u,d)< 0 for some $d \leq n$, vertex u if and only if there is a negative cycle
- If there are no negative cycles,

distance(u,v,n)

contains the shortest path data for all pairs of vertices u, v.

Completing and using the distance info

- Observe: Every sensible path has length at most n
- Therefore
  - After n iterations, time $\mathcal{O}(n^4)$, all sensible paths have been found
  - The only paths still improving after $> n$ steps must contain a cycle
- Detects negative cycles:
  - distance(u,u,d)< 0 for some $d \leq n$, vertex u if and only if there is a negative cycle
- If there are no negative cycles,

$$\text{distance(u,v,n)}$$

contains the shortest path data for all pairs of vertices u, v.

Dynamic programming strategy

- Dynamic programming is an advanced algorithm design principle (not fully covered in this course)
- Rough principle: Add extra memory use to speed up repetitive or complex computations
- In the mock-up:
  - Wanted the table distance(u,v,n) as final result
  - Used $n-1$ temporary tables distance(u,v,d) to produce it
  - Interpretation distance(u,v,d) stores path of at most d steps
- Floyd-Warshall:
  - Tables $\delta^t(i,j)$ storing shortest paths using only certain vertices
  - Build towards $\delta^n(i,j)$ which just stores shortest paths

Dynamic programming strategy

- Dynamic programming is an advanced algorithm design principle (not fully covered in this course)
- Rough principle: Add extra memory use to speed up repetitive or complex computations
- In the mock-up:
  - Wanted the table distance(u,v,n) as final result
  - Used $n - 1$ temporary tables distance(u,v,d) to produce it
  - Interpretation distance(u,v,d) stores path of at most d steps
- Floyd-Warshall:
  - Tables $\delta^t(i,j)$ storing shortest paths using only certain vertices
  - Build towards $\delta^n(i,j)$ which just stores shortest paths

Dynamic programming strategy

- Dynamic programming is an advanced algorithm design principle (not fully covered in this course)
- Rough principle: Add extra memory use to speed up repetitive or complex computations
- In the mock-up:
  - Wanted the table distance(u,v,n) as final result
  - Used $n-1$ temporary tables distance(u,v,d) to produce it
  - Interpretation distance(u,v,d) stores path of at most d steps
- Floyd-Warshall:
  - Tables $\delta^t(i,j)$ storing shortest paths using only certain vertices
  - Build towards $\delta^n(i,j)$ which just stores shortest paths

Dynamic programming strategy

- Dynamic programming is an advanced algorithm design principle (not fully covered in this course)
- Rough principle: Add extra memory use to speed up repetitive or complex computations
- In the mock-up:
  - Wanted the table distance(u,v,n) as final result
  - Used $n - 1$ temporary tables distance(u,v,d) to produce it
  - Interpretation distance(u,v,d) stores path of at most d steps
- Floyd-Warshall:
  - Tables $\delta^t(i, j)$ storing shortest paths using only certain vertices
  - Build towards $\delta^n(i, j)$ which just stores shortest paths

Floyd-Warshall

- Graph $G = (V, E)$, rename vertices $V = \{1, 2, \dots, n\}$
- Write weight(ij) for weight of arc $ij \in E$, if exists, otherwise weight(ij) $:= \infty$
- Temporary tables δ_{ij}^t, meaning

  Shortest path from i to j, if all intermediate vertices (i.e., other than i or j) have index at most t.

- As with the mock-up example:
  1. Can compute base case easily
  2. Can use tables $\delta^t(i,j)$ to compute $\delta^{t+1}(i,j)$
  3. Once we know $\delta^n(i,j)$, we are done
- Base case

$$\delta^0(i,j) = \text{weight}(ij)$$

with $\delta^0(i,i) = 0$.

Floyd-Warshall

- Graph $G = (V, E)$, rename vertices $V = \{1, 2, \ldots, n\}$
- Write weight(ij) for weight of arc $ij \in E$, if exists, otherwise weight(ij) $:= \infty$
- Temporary tables δ_{ij}^t, meaning

  Shortest path from i to j, if all intermediate vertices (i.e., other than i or j) have index at most t.
- As with the mock-up example:
  1. Can compute base case easily
  2. Can use tables $\delta^t(i,j)$ to compute $\delta^{t+1}(i,j)$
  3. Once we know $\delta^n(i,j)$, we are done
- Base case

$$\delta^0(i,j) = \mathrm{weight}(ij)$$

with $\delta^0(i,i) = 0$.

Floyd-Warshall

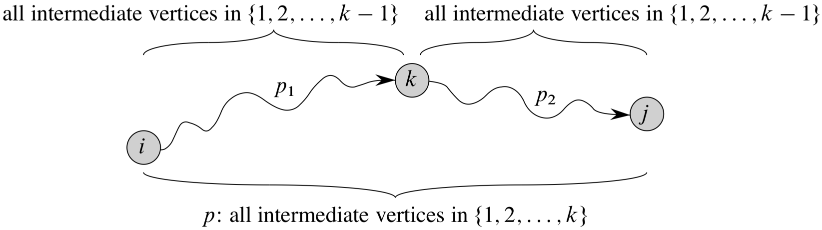
- Graph $G = (V, E)$, rename vertices $V = \{1, 2, \ldots, n\}$
- Write weight(ij) for weight of arc $ij \in E$, if exists, otherwise weight(ij) $:= \infty$
- Temporary tables δ_{ij}^t, meaning

  Shortest path from i to j, if all intermediate vertices (i.e., other than i or j) have index at most t.
- As with the mock-up example:
  1. Can compute base case easily
  2. Can use tables $\delta^t(i, j)$ to compute $\delta^{t+1}(i, j)$
  3. Once we know $\delta^n(i, j)$, we are done
- Base case

$$\delta^0(i, j) = \text{weight}(ij)$$

  with $\delta^0(i, i) = 0$.

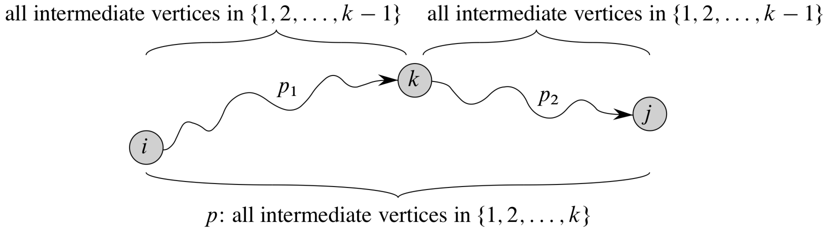
Floyd-Warshall: Recursive case



- Values $\delta^{k-1}(i,j)$ store information about paths whose internal vertices come from the set $\{1, \ldots, k-1\}$
- To compute $\delta^k(i,j)$, we need to add information about paths where also k may be internal
- Two varieties:
  1. New path does not use k: $\delta^k(i,j) = \delta^{k-1}(i,j)$
  2. New path uses k: Break new path into before k and after k (see figure), getting

$$\delta^k(i,j) = \delta^{k-1}(i,k) + \delta^{k-1}(k,j)$$

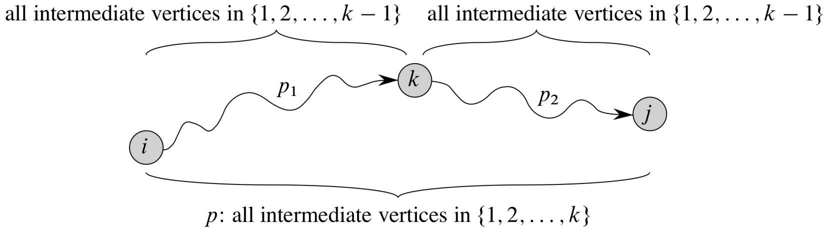
Floyd-Warshall: Recursive case



- Values $\delta^{k-1}(i,j)$ store information about paths whose internal vertices come from the set $\{1, \ldots, k-1\}$
- To compute $\delta^k(i,j)$, we need to add information about paths where also k may be internal
- Two varieties:
  1. New path does not use k: $\delta^k(i,j) = \delta^{k-1}(i,j)$
  2. New path uses k: Break new path into before k and after k (see figure), getting

$$\delta^k(i,j) = \delta^{k-1}(i,k) + \delta^{k-1}(k,j)$$

Floyd-Warshall: Recursive case



- Values $\delta^{k-1}(i,j)$ store information about paths whose internal vertices come from the set $\{1, \ldots, k-1\}$
- To compute $\delta^k(i,j)$, we need to add information about paths where also k may be internal
- Two varieties:
  1. New path does not use k: $\delta^k(i,j) = \delta^{k-1}(i,j)$
  2. New path uses k: Break new path into before k and after k (see figure), getting

$$\delta^k(i,j) = \delta^{k-1}(i,k) + \delta^{k-1}(k,j)$$

Algorithm: Floyd-Warshall

1. Initialise:
    1.1 $\delta^0(i,i) = 0$
    1.2 $\delta^0(i,j) =$weight(ij), otherwise
2. For k=1 to n:
    2.1 Create the table $\delta^k(i,j)$
    2.2 For every pair $i,j \in V$, compute

$$\delta^k(i,j) = \min(\delta^{k-1}(i,j), \quad \delta^{k-1}(i,k) + \delta^{k-1}(k,j))$$

3. Return the values $\delta^n(i,j)$ as final distances

Graph contains negative cycle if and only if $\delta^k(i,i) < 0$ at some point.

Time: Obviously $\Theta(n^3)$.

Algorithm: Floyd-Warshall

1. Initialise:
   1.1 $\delta^0(i,i) = 0$
   1.2 $\delta^0(i,j) =$weight(ij), otherwise
2. For k=1 to n:
   2.1 Create the table $\delta^k(i,j)$
   2.2 For every pair $i,j \in V$, compute

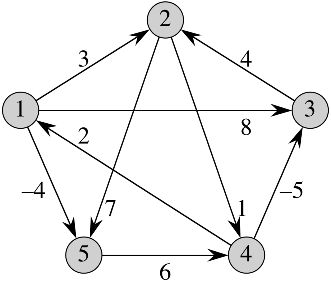
$$\delta^k(i,j) = \min(\delta^{k-1}(i,j), \quad \delta^{k-1}(i,k) + \delta^{k-1}(k,j))$$

3. Return the values $\delta^n(i,j)$ as final distances

Graph contains negative cycle if and only if $\delta^k(i,i) < 0$ at some point.

Time: Obviously $\Theta(n^3)$.

Example for Floyd-Warshall



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$\Pi^{(k)}$ entries $\pi_{ij}^{(k)}$ are predecessors of j on the current best path from i

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Using all $\Pi^{(k)}$ one can get the shortest path from i to j

Shortest paths: Summary

- Single source, unit weights:
- Single source, non-negative weights:
- Single source, arbitrary weights: Omitted (see below)
- All pairs, arbitrary weights but no negative cycles:
- Omitted:
  - Directed Acyclic Graphs, special-purpose algorithms
  - Bellman-Ford: Single source, arbitrary weights

Shortest paths: Summary

- Single source, unit weights:
  - BFS runs in $\mathcal{O}(|E|)$ time
- Single source, non-negative weights:
- Single source, arbitrary weights: Omitted (see below)
- All pairs, arbitrary weights but no negative cycles:
- Omitted:
  - Directed Acyclic Graphs, special-purpose algorithms
  - Bellman-Ford: Single source, arbitrary weights

Shortest paths: Summary

- Single source, unit weights:
  - BFS runs in $\mathcal{O}(|E|)$ time
- Single source, non-negative weights:
  - Dijkstra, in $\mathcal{O}(|E| \log |V|)$ or $\mathcal{O}(|E| + |V| \log |V|)$ or $\mathcal{O}(|V|^2)$ time (basic priority queue, Fibonacci heap, no priority queue)
- Single source, arbitrary weights: Omitted (see below)
- All pairs, arbitrary weights but no negative cycles:
- Omitted:
  - Directed Acyclic Graphs, special-purpose algorithms
  - Bellman-Ford: Single source, arbitrary weights

Shortest paths: Summary

- Single source, unit weights:
  - BFS runs in $\mathcal{O}(|E|)$ time
- Single source, non-negative weights:
  - Dijkstra, in $\mathcal{O}(|E|\log|V|)$ or $\mathcal{O}(|E|+|V|\log|V|)$ or $\mathcal{O}(|V|^2)$ time (basic priority queue, Fibonacci heap, no priority queue)
- Single source, arbitrary weights: Omitted (see below)
- All pairs, arbitrary weights but no negative cycles:
- Omitted:
  - Directed Acyclic Graphs, special-purpose algorithms
  - Bellman-Ford: Single source, arbitrary weights

Shortest paths: Summary

- Single source, unit weights:
  - BFS runs in $\mathcal{O}(|E|)$ time
- Single source, non-negative weights:
  - Dijkstra, in $\mathcal{O}(|E| \log |V|)$ or $\mathcal{O}(|E| + |V| \log |V|)$ or $\mathcal{O}(|V|^2)$ time (basic priority queue, Fibonacci heap, no priority queue)
- Single source, arbitrary weights: Omitted (see below)
- All pairs, arbitrary weights but no negative cycles:
  - Floyd-Warshall, in $\mathcal{O}(|V|^3)$ time
  - Repeated single-source algorithm, in special cases
- Omitted:
  - Directed Acyclic Graphs, special-purpose algorithms
  - Bellman-Ford: Single source, arbitrary weights

Shortest paths: Summary

- Single source, unit weights:
  - BFS runs in $\mathcal{O}(|E|)$ time
- Single source, non-negative weights:
  - Dijkstra, in $\mathcal{O}(|E| \log |V|)$ or $\mathcal{O}(|E| + |V| \log |V|)$ or $\mathcal{O}(|V|^2)$ time (basic priority queue, Fibonacci heap, no priority queue)
- Single source, arbitrary weights: Omitted (see below)
- All pairs, arbitrary weights but no negative cycles:
  - Floyd-Warshall, in $\mathcal{O}(|V|^3)$ time
  - Repeated single-source algorithm, in special cases
- Omitted:
  - Directed Acyclic Graphs, special-purpose algorithms
  - Bellman-Ford: Single source, arbitrary weights

Shortest paths: Summary

- Single source, unit weights:
  - BFS runs in $\mathcal{O}(|E|)$ time
- Single source, non-negative weights:
  - Dijkstra, in $\mathcal{O}(|E| \log |V|)$ or $\mathcal{O}(|E| + |V| \log |V|)$ or $\mathcal{O}(|V|^2)$ time (basic priority queue, Fibonacci heap, no priority queue)
- Single source, arbitrary weights: Omitted (see below)
- All pairs, arbitrary weights but no negative cycles:
  - Floyd-Warshall, in $\mathcal{O}(|V|^3)$ time
  - Repeated single-source algorithm, in special cases
- Omitted:
  - Directed Acyclic Graphs, special-purpose algorithms
  - Bellman-Ford: Single source, arbitrary weights

Shortest paths: Summary

- Single source, unit weights:
  - BFS runs in $\mathcal{O}(|E|)$ time
- Single source, non-negative weights:
  - Dijkstra, in $\mathcal{O}(|E| \log |V|)$ or $\mathcal{O}(|E| + |V| \log |V|)$ or $\mathcal{O}(|V|^2)$ time (basic priority queue, Fibonacci heap, no priority queue)
- Single source, arbitrary weights: Omitted (see below)
- All pairs, arbitrary weights but no negative cycles:
  - Floyd-Warshall, in $\mathcal{O}(|V|^3)$ time
  - Repeated single-source algorithm, in special cases
- Omitted:
  - Directed Acyclic Graphs, special-purpose algorithms
  - Bellman-Ford: Single source, arbitrary weights

Negative edge weights: Application

Illustration: Currency exchange

- Have: Exchange rates offered by different agences
  - 1 GBP $\rightarrow$ 1.4 USD
  - 1 USD $\rightarrow$ 0.7 GBP
  - 1 Mexican Peso $\rightarrow$ 0.056 USD
  - 1 GBP $\rightarrow$ 9.4 Chinese Yuan
  - ...
- Seek most profitable exchange path
  - Example: 1 Peso $\rightarrow$ 0.056 USD $\rightarrow$ $0.056 \cdot 0.7$ GBP $\rightarrow$ $0.056 \cdot 0.7 \cdot 9.4$ Yuan
  - Makes path from 1 Peso to 0.368 Yuan
- Arbitrage options
  - What if a black market dealer offers 3 Peso per Yuan?
  - $0.368 \cdot 3 = 1.104 > 1$ Peso per Peso $\Rightarrow$ profit!

Illustration: Currency exchange

- Have: Exchange rates offered by different agences
  - 1 GBP $\rightarrow$ 1.4 USD
  - 1 USD $\rightarrow$ 0.7 GBP
  - 1 Mexican Peso $\rightarrow$ 0.056 USD
  - 1 GBP $\rightarrow$ 9.4 Chinese Yuan
  - ...
- Seek most profitable exchange path
  - Example: 1 Peso $\rightarrow$ 0.056 USD $\rightarrow$ $0.056 \cdot 0.7$ GBP $\rightarrow$ $0.056 \cdot 0.7 \cdot 9.4$ Yuan
  - Makes path from 1 Peso to 0.368 Yuan
- Arbitrage options
  - What if a black market dealer offers 3 Peso per Yuan?
  - $0.368 \cdot 3 = 1.104 > 1$ Peso per Peso $\Rightarrow$ profit!

Illustration: Currency exchange

- Have: Exchange rates offered by different agences
  - 1 GBP $\to$ 1.4 USD
  - 1 USD $\to$ 0.7 GBP
  - 1 Mexican Peso $\to$ 0.056 USD
  - 1 GBP $\to$ 9.4 Chinese Yuan
  - ...
- Seek most profitable exchange path
  - Example: 1 Peso $\to$ 0.056 USD $\to$ $0.056 \cdot 0.7$ GBP $\to$ $0.056 \cdot 0.7 \cdot 9.4$ Yuan
  - Makes path from 1 Peso to 0.368 Yuan
- Arbitrage options
  - What if a black market dealer offers 3 Peso per Yuan?
  - $0.368 \cdot 3 = 1.104 > 1$ Peso per Peso $\Rightarrow$ profit!

Currency exchange/arbitrage, continued

- Casting the currency exchange situation as a graph problem
  1. Instead of GBP $\rightarrow$ USD multiplier of 1.4...
  2. Create arc from GBP to USD, of weight $-\log 1.4$
  3. Negative weight $\Leftrightarrow$ less valuable target currency
  4. Rate 1.0 gets weight 0, rate 2.0 weight -1
- Exchange rate multiplication becomes addition of logarithms
  - Two-step conversion from USD to Yuan $0.7 \cdot 9.4$
  - $-\log(0.7 \cdot 9.4) = (-\log 0.7) + (-\log 9.4)$
  - $-\log 0.7 = 0.51$ and $-\log 9.4 = -3.23$
- Now we have:
  1. Best exchange rate A $\rightarrow$ B found via shortest path from A to B
  2. Arbitrage option (gaining cycle) found if and only if negative cycle – cycle where the edge weights sum to less than 0

Currency exchange/arbitrage, continued

- Casting the currency exchange situation as a graph problem
  1. Instead of GBP $\rightarrow$ USD multiplier of 1.4...
  2. Create arc from GBP to USD, of weight $-\log 1.4$
  3. Negative weight $\Leftrightarrow$ less valuable target currency
  4. Rate 1.0 gets weight 0, rate 2.0 weight -1
- Exchange rate multiplication becomes addition of logarithms
  - Two-step conversion from USD to Yuan $0.7 \cdot 9.4$
  - $-\log(0.7 \cdot 9.4) = (-\log 0.7) + (-\log 9.4)$
  - $-\log 0.7 = 0.51$ and $-\log 9.4 = -3.23$
- Now we have:
  1. Best exchange rate A $\rightarrow$ B found via shortest path from A to B
  2. Arbitrage option (gaining cycle) found if and only if negative cycle – cycle where the edge weights sum to less than 0

Currency exchange/arbitrage, continued

- Casting the currency exchange situation as a graph problem
  1. Instead of GBP $\rightarrow$ USD multiplier of 1.4...
  2. Create arc from GBP to USD, of weight $-\log 1.4$
  3. Negative weight $\Leftrightarrow$ less valuable target currency
  4. Rate 1.0 gets weight 0, rate 2.0 weight -1
- Exchange rate multiplication becomes addition of logarithms
  - Two-step conversion from USD to Yuan $0.7 \cdot 9.4$
  - $-\log(0.7 \cdot 9.4) = (-\log 0.7) + (-\log 9.4)$
  - $-\log 0.7 = 0.51$ and $-\log 9.4 = -3.23$
- Now we have:
  1. Best exchange rate A $\rightarrow$ B found via shortest path from A to B
  2. Arbitrage option (gaining cycle) found if and only if negative cycle – cycle where the edge weights sum to less than 0

Currency exchange/arbitrage, continued

- Casting the currency exchange situation as a graph problem
  1. Instead of GBP $\rightarrow$ USD multiplier of 1.4...
  2. Create arc from GBP to USD, of weight $-\log 1.4$
  3. Negative weight $\Leftrightarrow$ less valuable target currency
  4. Rate 1.0 gets weight 0, rate 2.0 weight -1
- Exchange rate multiplication becomes addition of logarithms
  - Two-step conversion from USD to Yuan $0.7 \cdot 9.4$
  - $-\log(0.7 \cdot 9.4) = (-\log 0.7) + (-\log 9.4)$
  - $-\log 0.7 = 0.51$ and $-\log 9.4 = -3.23$
- Now we have:
  1. Best exchange rate A $\rightarrow$ B found via shortest path from A to B
  2. Arbitrage option (gaining cycle) found if and only if negative cycle – cycle where the edge weights sum to less than 0