CSE 548: Analysis of Algorithms

Prerequisites Review 7
(More Graph Algorithms: Basic and Beyond)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2019

Breadth-First Search (BFS)

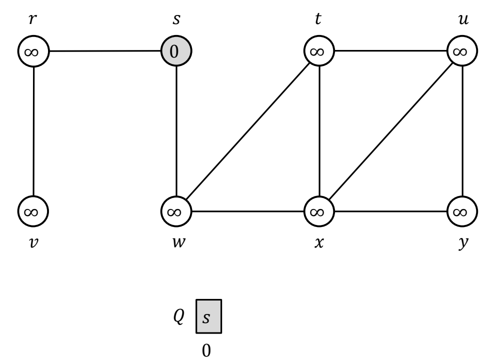
Input: Unweighted directed or undirected graph $G = (V, E)$ with vertex set V and edge set E, and a source vertex $s \in G.V$. For each $v \in V$, the adjacency list of v is $G.Adj[v]$.

Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance (in terms of the number of edges) from s to v. Also, $v.\pi$ pointers form a breadth-first tree rooted at s that contains all vertices reachable from s.

```
BFS ( G, s )
    for each vertex u ∈ G.V \ {s} do
        u.color ← WHITE, u.d ← ∞, u.π ← NIL
    s.color ← GRAY, s.d ← 0, s.π ← NIL
    Queue Q ← ∅
    ENQUEUE( Q, s )
    while Q ≠ ∅ do
        u ← DEQUEUE( Q )
        for each v ∈ G.Adj[u] do
            if v.color = WHITE then
                v.color ← GRAY, v.d ← u.d + 1, v.π ← u
                ENQUEUE( Q, v )
        u.color ← BLACK
```


Breadth-First Search (BFS)

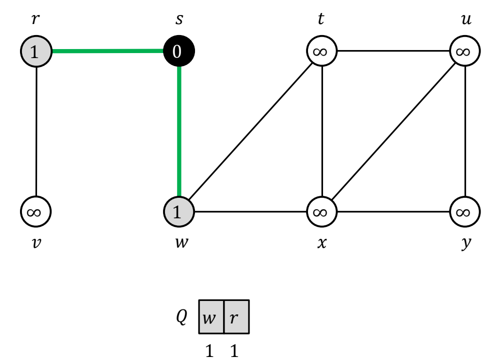
ENQUEUE (Q, s)

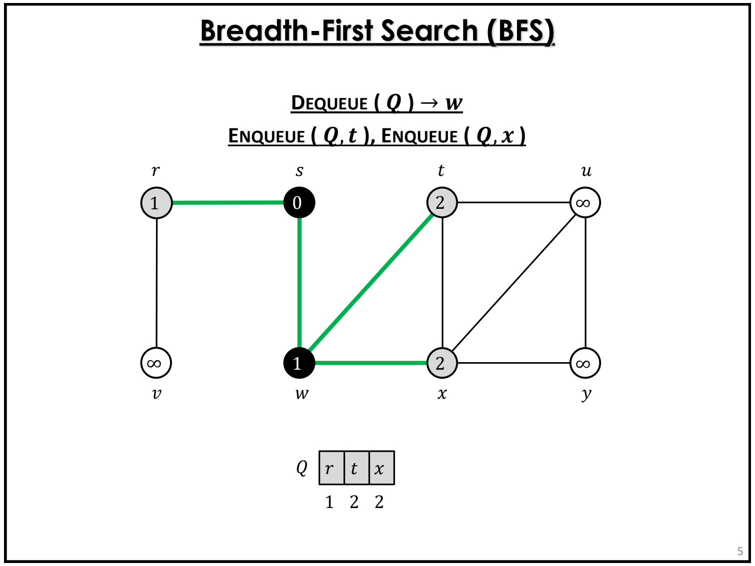


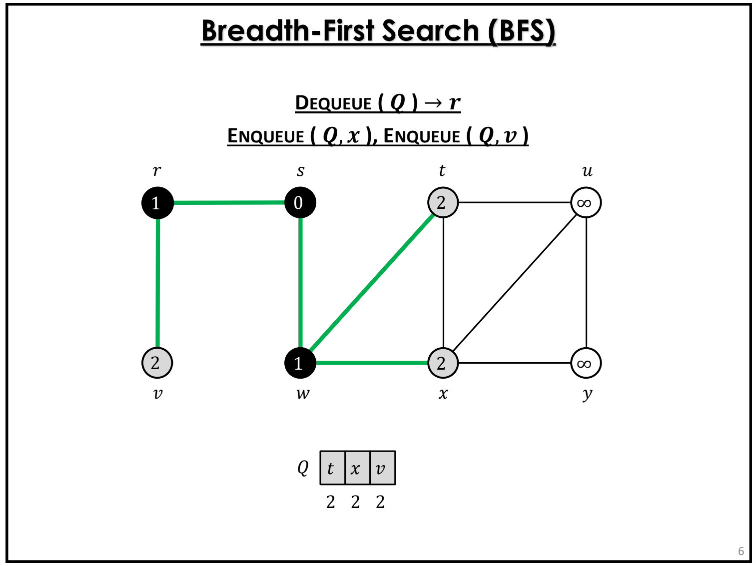
Breadth-First Search (BFS)

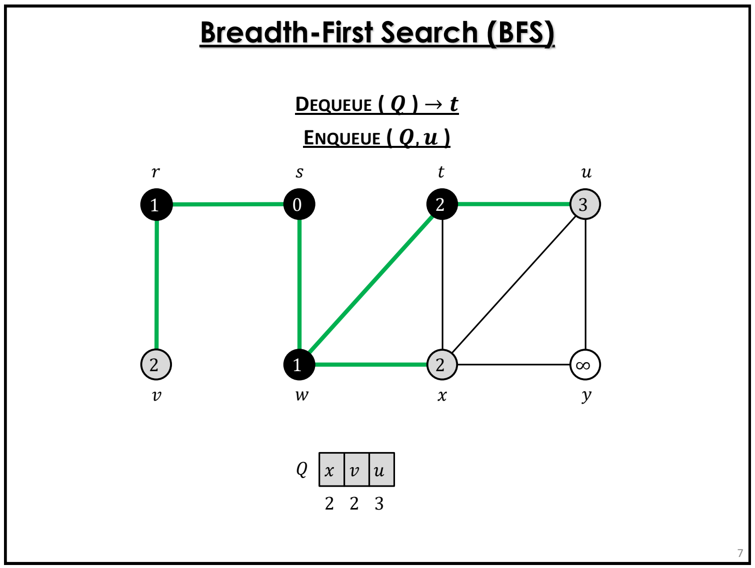
DEQUEUE (Q) → s

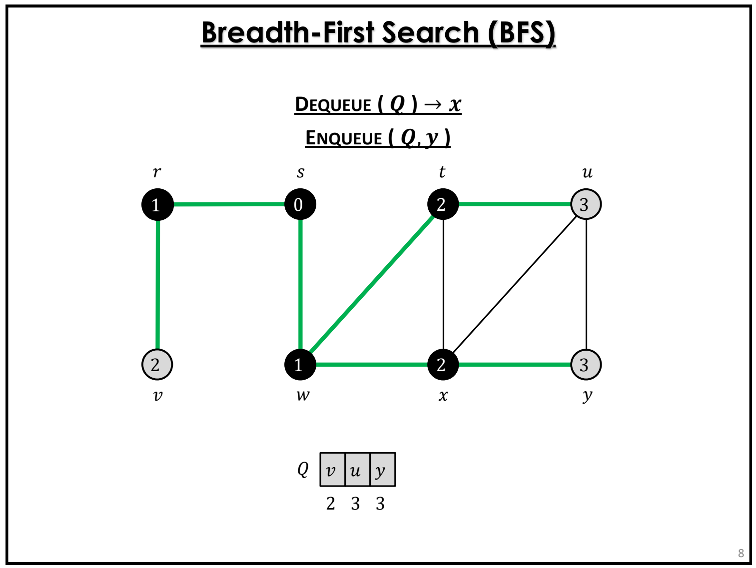
ENQUEUE (Q, w), ENQUEUE (Q, r)




Breadth-First Search (BFS)
DEQUEUE (Q) → w
ENQUEUE (Q, t), ENQUEUE (Q, x)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q r t x
1 2 2
5

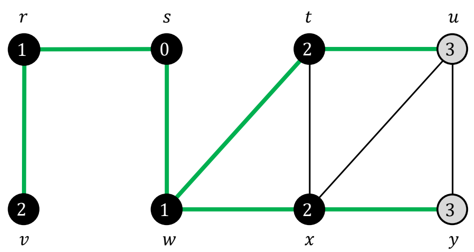

Breadth-First Search (BFS)
DEQUEUE (Q) → r
ENQUEUE (Q, x), ENQUEUE (Q, v)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q t x v
2 2 2
6


Breadth-First Search (BFS)
DEQUEUE (Q) → t
ENQUEUE (Q, u)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3
7


Breadth-First Search (BFS)
DEQUEUE (Q) → x
ENQUEUE (Q, y)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3
8

Breadth-First Search (BFS)

DEQUEUE (Q) → v

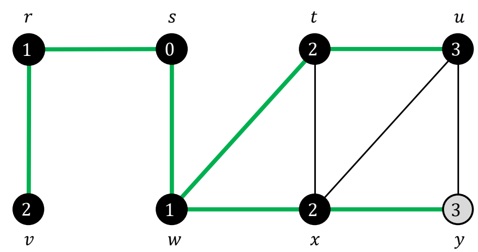


Q | u | y |
3 3

9

Breadth-First Search (BFS)

DEQUEUE (Q) → u

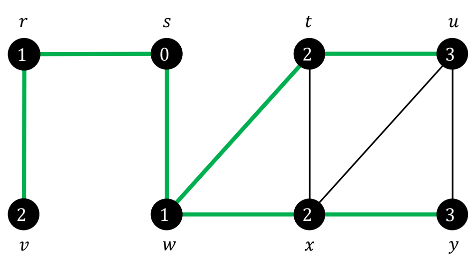


Q | y |
3

10

Breadth-First Search (BFS)

DEQUEUE (Q) → y



Q $\emptyset$

11

Breadth-First Search (BFS)

```
BFS ( G, s )
    for each vertex u ∈ G.V \ {s} do
        u.color ← WHITE,  u.d ← ∞,  u.π ← NIL
    s.color ← GRAY,  s.d ← 0,  s.π ← NIL
    Queue Q ← ∅
    ENQUEUE( Q, s )
    while Q ≠ ∅ do
        u ← DEQUEUE( Q )
        for each v ∈ G.Adj[u] do
            if v.color = WHITE then
                v.color ← GRAY,  v.d ← u.d + 1,  v.π ← u
                ENQUEUE( Q, v )
        u.color ← BLACK
```

Let $n = |G.V|$ and $m = |G.E|$

Time spent

- initializing $= \Theta(n)$
- enqueuing / dequeuing $= \Theta(n)$
- scanning the adjacency lists $= \Theta\left(\sum_{v \in G.V} |G.Adj[v]|\right) = \Theta(m)$

∴ Total cost $= \Theta(m + n)$

12

Depth-First Search (DFS)

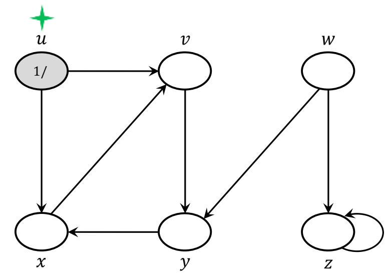
Input: Unweighted directed or undirected graph $G = (V, E)$ with vertex set V and edge set E. For each $v \in V$, the adjacency list of v is $G.Adj[v]$.

Output: For each $v \in G[V]$, $v.d$ is set to the time when v was first discovered and $v.f$ is set to the time when v's adjacency list has been examined completely. Also, $v.\pi$ pointers form a breadth-first tree rooted at s that contains all vertices reachable from s.

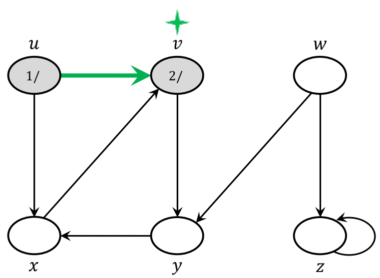
```
DFS ( G )
    for each vertex u ∈ G.V do
        u.color ← WHITE,  u.π ← NIL
    time ← 0
    for each u ∈ G.V do
        if u.color = WHITE then
            DFS-VISIT( G, u )
```

```
DFS-VISIT ( G, u )
    time ← time + 1
    u.d ← time
    u.color ← GRAY
    for each v ∈ G.Adj[u] do
        if v.color = WHITE then
            v.π ← u
            DFS-VISIT( G, v )
    u.color ← BLACK
    time ← time + 1
    u.f ← time
```


Depth-First Search (DFS)

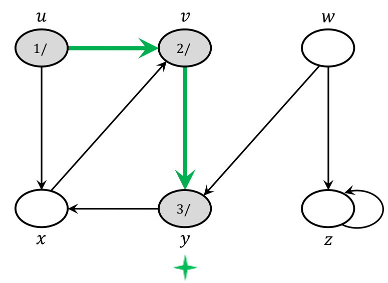


Depth-First Search (DFS)

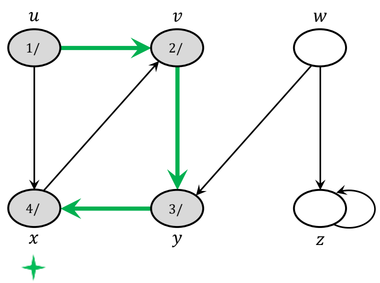


Tree Edge (T): These are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring that edge. In the example above, we will make all tree edges green and thick.

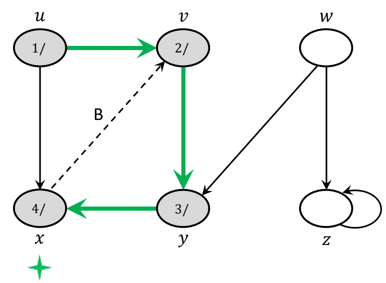
Depth-First Search (DFS)



Depth-First Search (DFS)

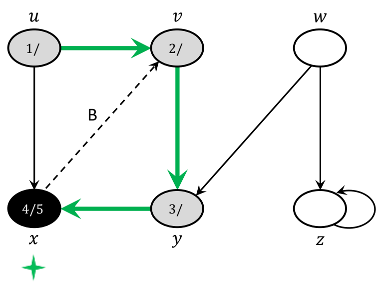


Depth-First Search (DFS)

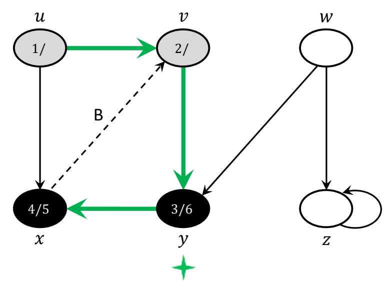


Back Edge (B): A back edge goes from a vertex to its ancestor in a depth-first tree. Self-loops are also considered back edges.

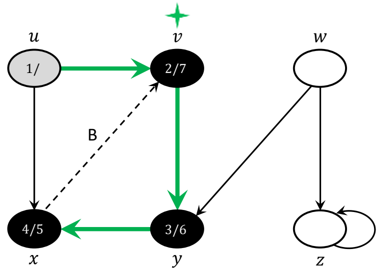
Depth-First Search (DFS)



Depth-First Search (DFS)

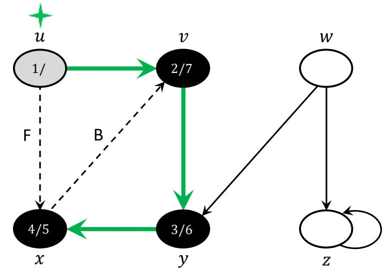


Depth-First Search (DFS)



21

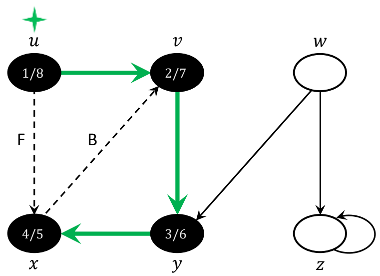
Depth-First Search (DFS)



Forward Edge (F): A forward edge is a nontree edge that connects a vertex to a descendant in a depth-first tree.

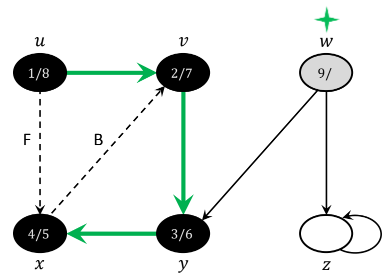
22

Depth-First Search (DFS)



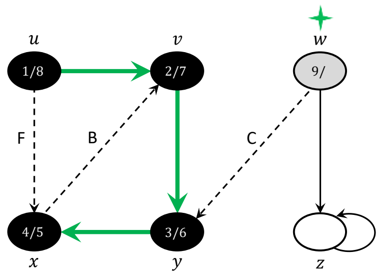
23

Depth-First Search (DFS)



24

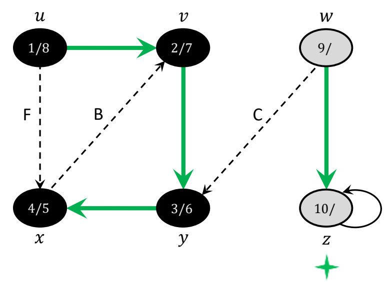
Depth-First Search (DFS)



Cross Edge (C): If a non-tree edge is neither a back edge nor a forward edge then it's a cross edge. Cross edges can go between vertices in the same depth-first tree or in different depth-first trees.

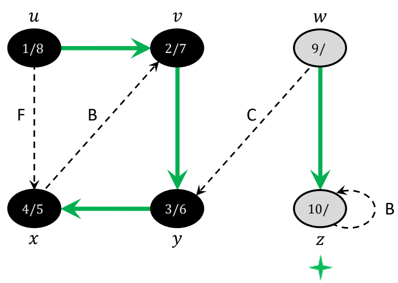
25

Depth-First Search (DFS)



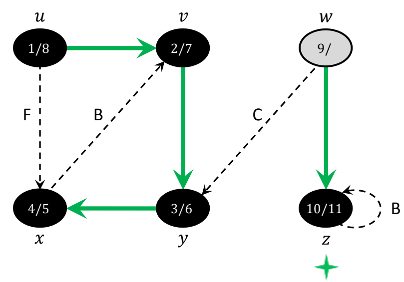
26

Depth-First Search (DFS)



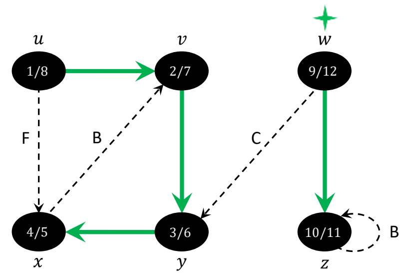
27

Depth-First Search (DFS)



28

Depth-First Search (DFS)



Depth-First Search (DFS)

```
DFS ( G )
    for each vertex u ∈ G.V do
        u.color ← WHITE,  u.π ← NIL
    time ← 0
    for each u ∈ G.V do
        if u.color = WHITE then
            DFS-VISIT( G, u )
```

```
DFS-VISIT ( G, u )
    time ← time + 1
    u.d ← time
    u.color ← GRAY
    for each v ∈ G.Adj[u] do
        if v.color = WHITE then
            v.π ← u
            DFS-VISIT( G, v )
    u.color ← BLACK
    time ← time + 1
    u.f ← time
```

Let $n = |G.V|$ and $m = |G.E|$

Time spent

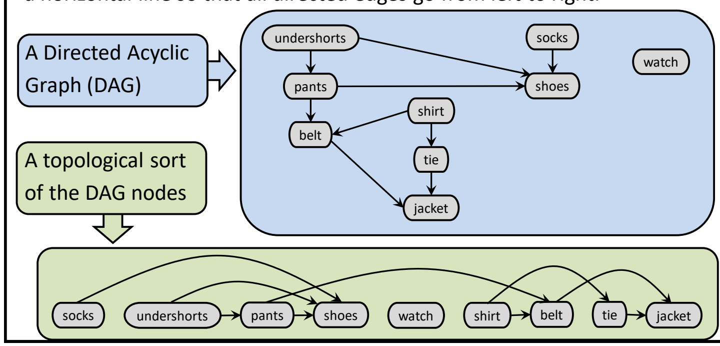
- in *DFS* (exclusive of calls to *DFS-VISIT*) $= \Theta(n)$
- in *DFS-VISIT* scanning the adjacency lists $= \Theta\left(\sum_{v \in G.V} |G.Adj[v]|\right)$ $= \Theta(m)$

$\therefore$ Total cost $= \Theta(m+n)$

Topological Sort

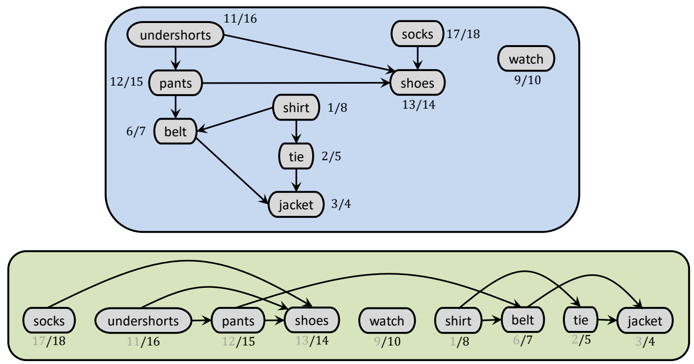
A ***topological sort*** of a DAG (i.e., directed acyclic graph) $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.



Topological Sort

```
TOPOLOGICAL-SORT ( G )
    call DFS ( G ) to compute the finish times v.f for each vertex v ∈ G.V
    as each vertex is finished, insert it into the front of a linked list
    return the linked list of vertices
```

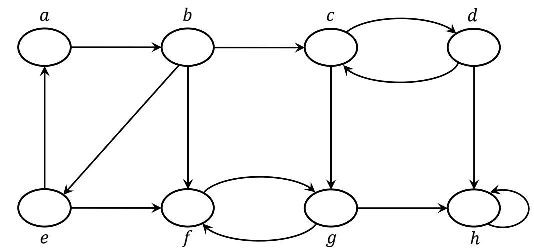


Strongly Connected Components

A ***strongly connected component*** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other.

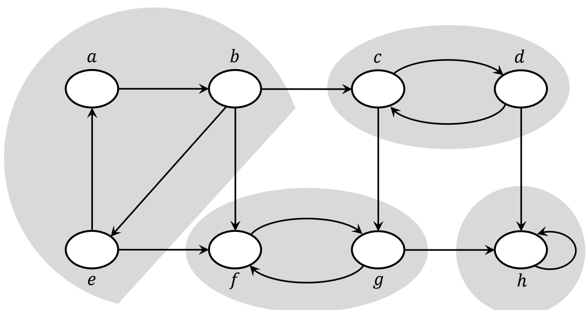
Strongly Connected Components

A ***strongly connected component*** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other.



Strongly Connected Components

A ***strongly connected component*** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other.



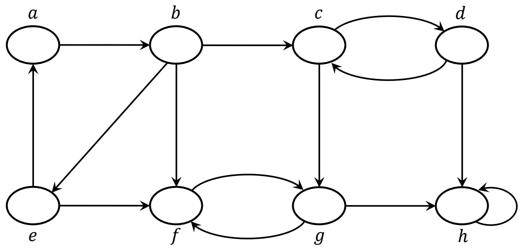
Strongly Connected Components

```
STRONGLY-CONNECTED-COMPONENTS ( G )
1.  call DFS ( G ) to compute the finish times v.f for each vertex v ∈ G.V
2.  compute G^T
3.  call DFS ( G^T ), but in the main loop of DFS, consider the vertices in order
        of decreasing v.f (as computed in line 1)
4.  output the vertices of each tree in the depth-first forest formed in line 3 as
        a separate strongly connected component
```

Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS (G)

1. call *DFS* (G) to compute the finish times $v.f$ for each vertex $v \in G.V$
2. compute G^T
3. call *DFS* (G^T), but in the main loop of *DFS*, consider the vertices in order of decreasing $v.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

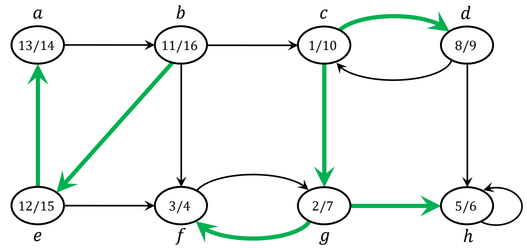


37

Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS (G)

1. call *DFS* (G) to compute the finish times $v.f$ for each vertex $v \in G.V$
2. compute G^T
3. call *DFS* (G^T), but in the main loop of *DFS*, consider the vertices in order of decreasing $v.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

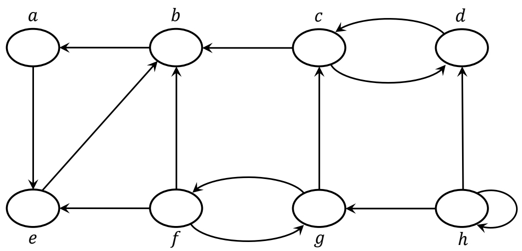


38

Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS (G)

1. call *DFS* (G) to compute the finish times $v.f$ for each vertex $v \in G.V$
2. compute G^T
3. call *DFS* (G^T), but in the main loop of *DFS*, consider the vertices in order of decreasing $v.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

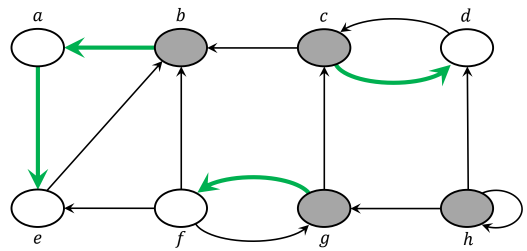


39

Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS (G)

1. call *DFS* (G) to compute the finish times $v.f$ for each vertex $v \in G.V$
2. compute G^T
3. call *DFS* (G^T), but in the main loop of *DFS*, consider the vertices in order of decreasing $v.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

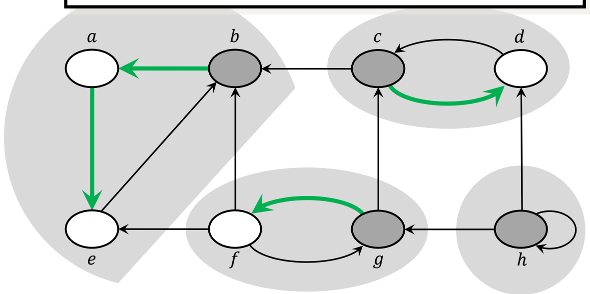


40

Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS (G)

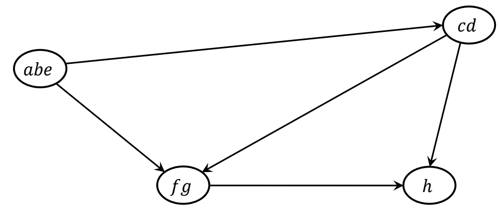
1. call *DFS* (G) to compute the finish times $v.f$ for each vertex $v \in G.V$
2. compute G^T
3. call *DFS* (G^T), but in the main loop of *DFS*, consider the vertices in order of decreasing $v.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component



Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS (G)

1. call *DFS* (G) to compute the finish times $v.f$ for each vertex $v \in G.V$
2. compute G^T
3. call *DFS* (G^T), but in the main loop of *DFS*, consider the vertices in order of decreasing $v.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component



The Single-Source Shortest Paths (SSSP) Problem

We are given a weighted, directed graph $G = (V, E)$ with vertex set V and edge set E, and a weight function w such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

We are also given a source vertex $s \in V$.

Our goal is to find a shortest path (i.e., a path of the smallest total edge weight) from s to each vertex $v \in V$.

SSSP: Relxation

INITIALIZE-SINGLE-SOURCE ($G = (V, E)$, s)

1. *for* each vertex $v \in G.V$ *do*
2. $\quad v.d \leftarrow \infty$
3. $\quad v.\pi \leftarrow NIL$
4. $s.d \leftarrow 0$

RELAX (u, v, w)

1. *if* $u.d + w(u, v) < v.d$ *then*
2. $\quad v.d \leftarrow u.d + w(u, v)$
3. $\quad v.\pi \leftarrow u$

(Prerequisites Review 7): More Graph Algorithms: Basic and Beyond

SSSP: Properties of Shortest Paths and Relxation

The ***weight*** $w(p)$ of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the ***shortest-path weight*** $\delta(u, v)$ from u to v by

$$\delta(u,v) = \begin{cases} \min\{w(p) : p \text{ is } u \sim v\}, & \text{if there is a path from } u \text{ to } v, \\ \infty, & \text{otherwise.} \end{cases}$$

A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

45

SSSP: Properties of Shortest Paths and Relxation

Triangle inequality (Lemma 24.10 of CLRS)

For any edge $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Upper-bound inequality (Lemma 24.11 of CLRS)

We always have $v.d \ge \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(u, v)$, it never changes.

No-path property (Corollary 24.12 of CLRS)

If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14 of CLRS)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

46

SSSP: Properties of Shortest Paths and Relxation

Path-relaxation property (Lemma 24.15 of CLRS)

If $p = \langle v_0, v_1, \dots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations on the edges of p.

Predecessor-subgraph property (Lemma 24.17 of CLRS)

Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

47

Dijkstra's SSSP Algorithm with a Min-Heap
(SSSP: Single-Source Shortest Paths)

Since we already discussed Dijkstra's SSSP algorithm when we talked about greedy algorithms, we will skip over it in this lecture.

Dijkstra's SSSP Algorithm with a Min-Heap
(SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set V and edge set E, a non-negative weight function w, and a source vertex $s \in G[V]$.

Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance from s to v.

```
Dijkstra-SSSP ( G = (V, E), w, s )
    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    s.d ← 0
    Min-Heap Q ← ∅
    for each vertex v ∈ G.V do
        Insert( Q, v )
    while Q ≠ ∅ do
        u ← Extract-Min( Q )
        for each (u, v) ∈ G.E do
            if u.d + w(u, v) < v.d then
                v.d ← u.d + w(u, v)
                v.π ← u
                Decrease-Key( Q, v, u.d + w(u, v) )
```

Let $n = |G[V]|$ and $m = |G[E]|$

Worst-case running time:

Using a binary min-heap $= O((m + n) \log n)$

Using a Fibonacci heap $= O(m + n \log n)$

48

Dijkstra's SSSP Algorithm with a Min-Heap
(SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set V and edge set E, a non-negative weight function w, and a source vertex $s \in G[V]$.

Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance from s to v.

```
Dijkstra-SSSP ( G = (V,E), w, s )
    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    s.d ← 0
    Min-Heap Q ← ∅
    for each vertex v ∈ G.V do
        INSERT( Q, v )
    while Q ≠ ∅ do
        u ← EXTRACT-MIN( Q )
        for each (u,v) ∈ G.E do
            if u.d + w(u,v) < v.d then
                v.d ← u.d + w(u,v)
                v.π ← u
                DECREASE-KEY( Q, v, u.d + w(u,v) )
```

Let $n = |G[V]|$ and $m = |G[E]|$

Worst-case running time:

Using a binary min-heap $= O((m + n)\log n)$

Using a Fibonacci heap $= O(m + n\log n)$

49

The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set V and edge set E, a weight function w, and a source vertex $s \in G[V]$. Negative-weight edges are allowed (unlike Dijkstra's SSSP algorithm).

Output: Returns FALSE if a negative-weight cycle is reachable from s, otherwise returns TRUE and for all $v \in G[V]$, sets $v.d$ to the shortest distance from s to v.

```
INITIALIZE-SINGLE-SOURCE ( G = (V,E), s )
    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    s.d ← 0
```

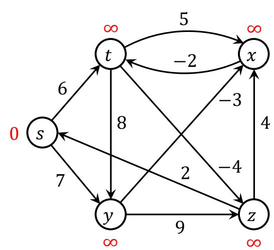
```
RELAX ( u, v, w )
    if u.d + w(u,v) < v.d then
        v.d ← u.d + w(u,v)
        v.π ← u
```

```
BELLMAN-FORD ( G = (V,E), w, s )
    INITIALIZE-SINGLE-SOURCE( G, s )
    for i ← 1 to |G.V| − 1 do
        for each (u,v) ∈ G.E do
            RELAX( u, v, w )
    for each (u,v) ∈ G.E do
        if u.d + w(u,v) < v.d then
            return FALSE
    return TRUE
```

50

The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

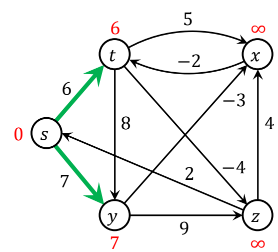
Initial State (with initial tentative distances)



51

The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

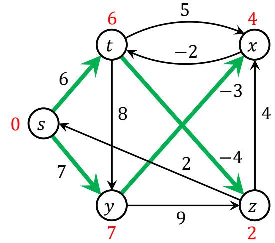
Iteration 1



52

The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

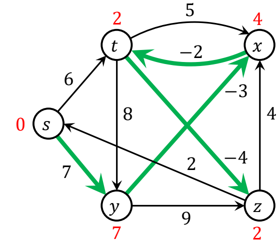
Iteration 2



53

The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

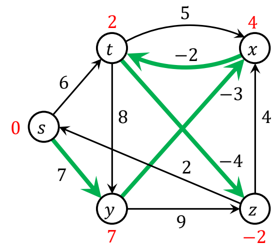
Iteration 3



54

The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

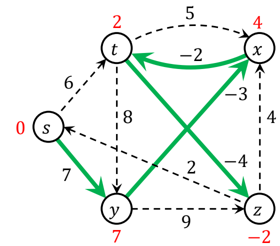
Iteration 4



55

The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

Done!



56

The Bellman-Ford (SSSP) Algorithm
(SSSP: Single-Source Shortest Paths)

```
INITIALIZE-SINGLE-SOURCE ( G = (V,E), s )
1.  for each vertex v ∈ G.V do
2.      v.d ← ∞
3.      v.π ← NIL
4.  s.d ← 0
```

```
RELAX ( u, v, w )
1.  if u.d + w(u,v) < v.d then
2.      v.d ← u.d + w(u,v)
3.      v.π ← u
```

```
BELLMAN-FORD ( G = (V,E), w, s )
1.  INITIALIZE-SINGLE-SOURCE( G,s )
2.  for i ← 1 to |G.V| − 1 do
3.      for each (u,v) ∈ G.E do
4.          RELAX( u,v,w )
5.  for each (u,v) ∈ G.E do
6.      if u.d + w(u,v) < v.d then
7.          return FALSE
8.  return TRUE
```

Let $n = |V|$ and $m = |E|$

Time taken by: Line 1: $\Theta(n)$
Lines $2-4$: $\Theta(mn)$
Lines $5-7$: $\Theta(m)$

Total time: $\Theta(mn)$

Correctness of the Bellman-Ford Algorithm

LEMMA 24.2 (CLRS): Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w: E \rightarrow \mathbb{R}$, and suppose G contains no negative-weight cycles reachable from s. Then, after the $|V| - 1$ iterations of the for loop of lines 2– 4 of BELLMAN-FORD, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

PROOF: The proof is based on the ***path-relaxation property***. Consider any $v \in G.V$ reachable from s, and let $p = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the for loop of lines 2– 4 relaxes all $|E|$ edges. Among the edges relaxed in the i^{th} iteration, for $i = 1, 2, \dots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$.

Correctness of the Bellman-Ford Algorithm

COROLLARY 24.3 (CLRS): Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w: E \rightarrow \mathbb{R}$, and suppose G contains no negative-weight cycles reachable from s. Then, for each $v \in V$, there is a path from s to v if and only if BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Correctness of the Bellman-Ford Algorithm

THEOREM 24.4 (CLRS): Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w: E \rightarrow \mathbb{R}$. If G contains no negative-weight cycles reachable from s, then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Correctness of the Bellman-Ford Algorithm

Proof of Theorem 24.4: Two cases:

G contains no negative-weight cycles reachable from s:

If $v \in G.V$ is reachable from s then according to Lemma 24.2 we have $v.d = \delta(s,v)$ at termination. Otherwise, $v.d = \delta(s,v) = \infty$ follows from the ***no-path property***.

The ***predecessor-subgraph property***, along with $v.d = \delta(s,v)$, implies that G_π is a shortest-paths tree.

Now, since at termination, for all edges $(u,v) \in G.E$, we have, $v.d = \delta(s,v)$ and $u.d = \delta(s,u)$, then by ***triangle inequality***:

$$v.d = \delta(s,v) \le \delta(s,u) + w(u,v) = u.d + w(u,v).$$

So, none of the tests in line 6 causes *Bellman-Ford* to return False. Therefore, it returns True.

61

Correctness of the Bellman-Ford Algorithm

Proof of Theorem 24.4 (Continued):

G contains a negative-weight cycle reachable from s:

Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ be the cycle, where $v_0 = v_k$. Then

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$

Assume for the sake of contradiction that *Bellman-Ford* returns True. Then $v_i.d \le v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1,2,\ldots,k$. Thus,

$$\sum_{i=1}^{k} v_i.d \le \sum_{i=1}^{k} \left(v_{i-1}.d + w(v_{i-1}, v_i)\right) = \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

But $\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$, and by Corollary 24.3, each $v_i.d$ is finite.

Thus, $\sum_{i=1}^{k} w(v_{i-1}, v_i) \ge 0$, which contradicts our initial assumption that $c = \langle v_0, v_1, \ldots, v_k \rangle$ is a negative-weight cycle.

62

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

Input: Weighted DAG $G = (V, E)$ with vertex set V and edge set E, a weight function w, and a source vertex $s \in G[V]$. Negative-weight edges are allowed (unlike Dijkstra's SSSP algorithm).

Output: For all $v \in G[V]$, sets $v.d$ to the shortest distance from s to v.

```
Initialize-Single-Source ( G = (V,E), s )
    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    s.d ← 0
```

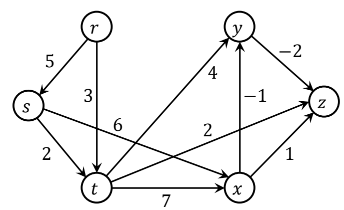
```
Relax ( u, v, w )
    if u.d + w(u,v) < v.d then
        v.d ← u.d + w(u,v)
        v.π ← u
```

```
DAG-Shortest-Paths ( G = (V,E), w, s )
    topologically sort the vertices of G
    Initialize-Single-Source( G, s )
    for each v ∈ V.G taken in topologically sorted order do
        for each (u,v) ∈ G.E do
            Relax( u, v, w )
```

63

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

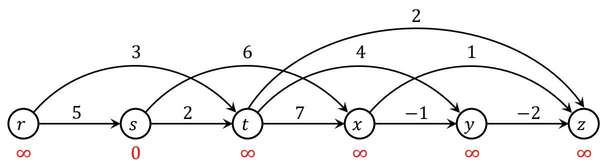
Given DAG



64

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

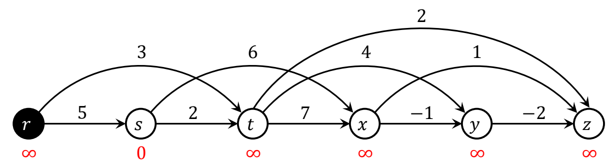
After Topological Sorting (with initial tentative distances)



65

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

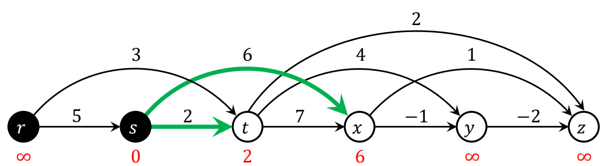
After Iteration 1



66

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

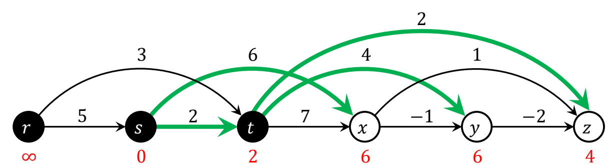
After Iteration 2



67

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

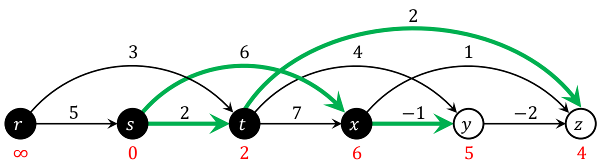
After Iteration 3



68

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

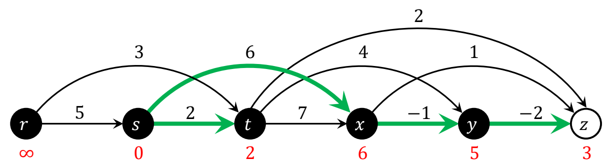
After Iteration 4



69

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

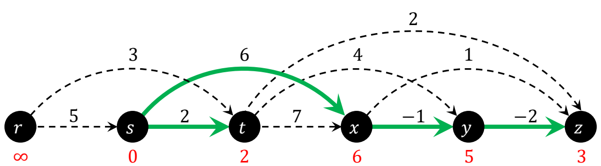
After Iteration 5



70

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

Done!



71

SSSP in Directed Acyclic Graphs (DAGs)
(SSSP: Single-Source Shortest Paths)

INITIALIZE-SINGLE-SOURCE ($G = (V, E)$, s)

1. *for* each vertex $v \in G.V$ *do*
2. $\quad v.d \leftarrow \infty$
3. $\quad v.\pi \leftarrow NIL$
4. $s.d \leftarrow 0$

RELAX (u, v, w)

1. *if* $u.d + w(u,v) < v.d$ *then*
2. $\quad v.d \leftarrow u.d + w(u,v)$
3. $\quad v.\pi \leftarrow u$

DAG-SHORTEST-PATHS ($G = (V, E)$, w, s)

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G, s)
3. *for* each $v \in V.G$ taken in topologically sorted order *do*
4. $\quad$ *for* each $(u, v) \in G.E$ *do*
5. $\quad\quad$ RELAX(u, v, w)

Let $n = |V|$ and
$m = |E|$

Time taken by: Line 1: $\Theta(n + m)$
Line 2: $\Theta(n)$
Lines $3 - 5$: $\Theta(m)$

Total time: $\Theta(n + m)$

72

Correctness of DAG-Shortest-Paths

Theorem 24.5 (CLRS): If a weighted, directed graph $G = (V, E)$ has a source vertex s and no cycles, then at the termination of the *DAG-Shortest-Paths* procedure, $v.d = \delta(s, v)$ for all vertices $v \in G.V$, and the predecessor subgraph G_π is a shortest-paths tree.

Proof: Consider any $v \in G.V$.

If v is not reachable from s then $v.d = \delta(s, v) = \infty$ follows from the ***no-path property***.

If v is reachable from s, and let $p = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Since we process the vertices in topological order, we relax the edges on p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. The ***path-relaxation property*** implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 1, 2, \dots, k$.

By the ***predecessor-subgraph property***, G_π is a shortest-paths tree.

73

Correctness of DAG-Shortest-Paths

Theorem 24.5 (CLRS): If a weighted, directed graph $G = (V, E)$ has a source vertex s and no cycles, then at the termination of the *DAG-Shortest-Paths* procedure, $v.d = \delta(s, v)$ for all vertices $v \in G.V$, and the predecessor subgraph G_π is a shortest-paths tree.

Proof: Consider any $v \in G.V$.

If v is not reachable from s then $v.d = \delta(s, v) = \infty$ follows from the ***no-path property***.

If v is reachable from s, and let $p = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Since we process the vertices in topological order, we relax the edges on p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. The ***path-relaxation property*** implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 1, 2, \dots, k$.

By the ***predecessor-subgraph property***, G_π is a shortest-paths tree.

74

Correctness of DAG-Shortest-Paths

Theorem 24.5 (CLRS): If a weighted, directed graph $G = (V, E)$ has a source vertex s and no cycles, then at the termination of the *DAG-Shortest-Paths* procedure, $v.d = \delta(s, v)$ for all vertices $v \in G.V$, and the predecessor subgraph G_π is a shortest-paths tree.

Proof: Consider any $v \in G.V$.

If v is not reachable from s then $v.d = \delta(s, v) = \infty$ follows from the ***no-path property***.

If v is reachable from s, and let $p = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Since we process the vertices in topological order, we relax the edges on p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. The ***path-relaxation property*** implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 1, 2, \dots, k$.

By the ***predecessor-subgraph property***, G_π is a shortest-paths tree.

75

The All-Pairs Shortest Paths (APSP) Problem

We are given a weighted, directed graph $G = (V, E)$ with vertex set V and edge set E, and a weight function w such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

Our goal is to find, for every pair of vertices $u, v \in G.V$, a shortest path (i.e., a path of the smallest total edge weight) from u to v.

76

The All-Pairs Shortest Paths (APSP) Problem

One can solve the APSP problem by running an SSSP algorithm $n = |G.V|$ times, once for each vertex as the source.

If all edge weights are nonnegative, one can use ***Dijkstra's SSSP algorithm***. Using a binary min-heap as the priority queue, one can solve the problem in $O(n(m+n)\log n)$ time, where $m = |G.E|$. Using a Fibonacci heap as the priority queue yields a running time of $O\left(n^2 \log n + mn\right)$.

If G has negative-weight edges, then one can use the slower ***Bellman-Ford SSSP algorithm*** resulting in a running time of $O\left(mn^2\right)$ which is $O\left(n^4\right)$ for dense graphs.

77

The All-Pairs Shortest Paths (APSP) Problem

We assume that the edge-weights are given as an $n \times n$ adjacency matrix $W = \left(w_{ij}\right)$, where

$$w_{ij} = \begin{cases} 0, & if\ i = j, \\ weight\ of\ directed\ edge\ (i,j) & if\ i \neq j\ and\ (i,j) \in E, \\ \infty & if\ i \neq j\ and\ (i,j) \notin E. \end{cases}$$

We allow negative-weight edges, but we assume for the time being that G contains no negative-weight cycles.

78

APSP: Extending SPs by One Edge at a Time

Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges. Then

$$l_{ij}^{(m)} = \begin{cases} 0, & if\ m = 0\ and\ i = j, \\ \infty & if\ m = 0\ and\ i \neq j, \\ \min\limits_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}, & otherwise\ (i.e., m > 0). \end{cases}$$

If G has no negative-weight cycles, then for every pair of vertices i and j for which $\delta(i,j) < \infty$, there is a shortest path from i to j that is simple and thus contains at most $n-1$ edges. A path from vertex i to vertex j with more than $n-1$ edges cannot have lower weight than a shortest path from i to j. Hence,

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots.$$

79

APSP: Extending SPs by One Edge at a Time

Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges. Then

$$l_{ij}^{(m)} = \begin{cases} 0, & if\ m = 0\ and\ i = j, \\ \infty & if\ m = 0\ and\ i \neq j, \\ \min\limits_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}, & otherwise\ (i.e., m > 0). \end{cases}$$

If G has no negative-weight cycles, then for every pair of vertices i and j for which $\delta(i,j) < \infty$, there is a shortest path from i to j that is simple and thus contains at most $n-1$ edges. A path from vertex i to vertex j with more than $n-1$ edges cannot have lower weight than a shortest path from i to j. Hence,

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots.$$

80

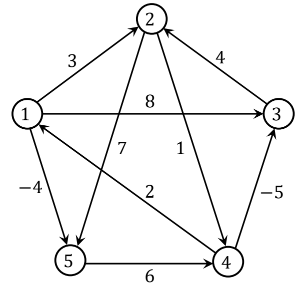
APSP: Extending SPs by One Edge at a Time

```
EXTEND-SHORTEST-PATHS ( L, W )
n ← L.rows
let L' = (l'_ij) be a new n × n matrix
for i ← 1 to n do
    for j ← 1 to n do
        l'_ij ← ∞
        for k ← 1 to n do
            l'_ij ← min( l'_ij, l'_ik + w_kj )
return L'
```

```
SLOW-ALL-PAIRS-SHORTEST-PATHS ( W )
n ← W.rows
L^(1) ← W
for m ← 2 to n − 1 do
    let L^(m) be a new n × n matrix
    L^(m) ← EXTEND-SHORTEST-PATHS( L^(m−1), W )
return L^(n−1)
```

81

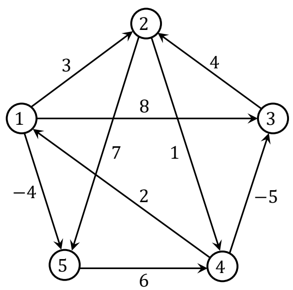
APSP: Extending SPs by One Edge at a Time



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

82

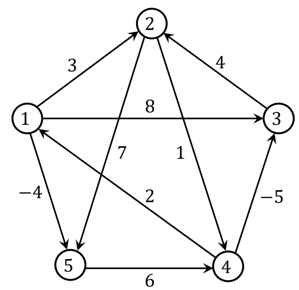
APSP: Extending SPs by One Edge at a Time



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

83

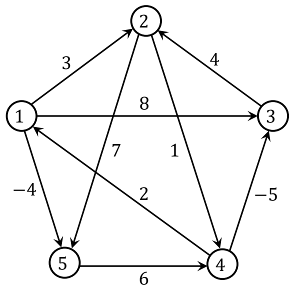
APSP: Extending SPs by One Edge at a Time



$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix} \qquad L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

84

APSP: Extending SPs by One Edge at a Time



$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

85

APSP: Extending SPs by One Edge at a Time

Note the similarity between *EXTEND-SHORTEST-PATHS* and *SQUARE-MATRIX-MULTIPLY*:

```
EXTEND-SHORTEST-PATHS ( L, W )
    n ← L.rows
    let L' = (l'_ij) be a new n × n matrix
    for i ← 1 to n do
        for j ← 1 to n do
            l'_ij ← ∞
            for k ← 1 to n do
                l'_ij ← min( l'_ij, l_ik + w_kj )
    return L'
```

```
SQUARE-MATRIX-MULTIPLY ( A, B )
    n ← A.rows
    let C = (c_ij) be a new n × n matrix
    for i ← 1 to n do
        for j ← 1 to n do
            c_ij ← 0
            for k ← 1 to n do
                c_ij ← c_ij + a_ik · b_kj
    return C
```

Both have the same $\Theta(n^3)$ running time.

86

APSP: Extending SPs by One Edge at a Time

```
EXTEND-SHORTEST-PATHS ( L, W )
    n ← L.rows
    let L' = (l'_ij) be a new n × n matrix
    for i ← 1 to n do
        for j ← 1 to n do
            l'_ij ← ∞
            for k ← 1 to n do
                l'_ij ← min( l'_ij, l_ik + w_kj )
    return L'
```

Running time $= \Theta(n^3)$

```
SLOW-ALL-PAIRS-SHORTEST-PATHS ( W )
    n ← W.rows
    L^(1) ← W
    for m ← 2 to n − 1 do
        let L^(m) be a new n × n matrix
        L^(m) ← EXTEND-SHORTEST-PATHS( L^(m−1), W )
    return L^(n−1)
```

Running time $= n \times \Theta(n^3) = \Theta(n^4)$

87

APSP: Extending SPs by Repeated Squaring

```
EXTEND-SHORTEST-PATHS ( L, W )
    n ← L.rows
    let L' = (l'_ij) be a new n × n matrix
    for i ← 1 to n do
        for j ← 1 to n do
            l'_ij ← ∞
            for k ← 1 to n do
                l'_ij ← min( l'_ij, l_ik + w_kj )
    return L'
```

```
FASTER-ALL-PAIRS-SHORTEST-PATHS ( W )
    n ← W.rows
    L^(1) ← W
    m ← 1
    while m < n − 1 do
        let L^(2m) be a new n × n matrix
        L^(2m) ← EXTEND-SHORTEST-PATHS( L^(m), L^(m) )
        m ← 2m
    return L^(m)
```

88

APSP: Extending SPs by Repeated Squaring

```
EXTEND-SHORTEST-PATHS ( L, W )
n ← L.rows
let L′ = (l′_ij) be a new n × n matrix
for i ← 1 to n do
    for j ← 1 to n do
        l′_ij ← ∞
        for k ← 1 to n do
            l′_ij ← min(l′_ij, l′_ik + w_kj)
return L′
```

Running time $= \Theta(n^3)$

```
FASTER-ALL-PAIRS-SHORTEST-PATHS ( W )
n ← W.rows
L^(1) ← W
m ← 1
while m < n − 1 do
    let L^(2m) be a new n × n matrix
    L^(2m) ← EXTEND-SHORTEST-PATHS( L^(m), L^(m) )
    m ← 2m
return L^(m)
```

Running time $= \lceil \log_2(n-1) \rceil \times \Theta(n^3) = \Theta(n^3 \log n)$

APSP: Floyd-Warshall's Algorithm

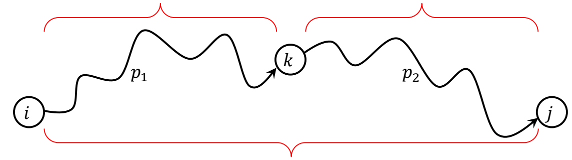
Let $d_{ij}^{(k)}$ be the minimum weight of any path from vertex i to vertex j for which all intermediate vertices are in $\{1,2,\dots,k\}$. Then

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{if } k = 0, \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{if } k \geq 1. \end{cases}$$

Then $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives: $d_{ij}^{(n)} = \delta(i,j)$ for all $i, j \in G.V$.

p_1: *all intermediate vertices in* $\{1,2,\dots,k-1\}$

p_1: *all intermediate vertices in* $\{1,2,\dots,k-1\}$



p: *all intermediate vertices in* $\{1,2,\dots,k\}$

APSP: Floyd-Warshall's Algorithm

```
FLOYD-WARSHALL ( W )
n ← W.rows
D^(0) ← W
for k ← 1 to n do
    let D^(k) = (d^(k)_ij) be a new n × n matrix
    for i ← 1 to n do
        for j ← 1 to n do
            d^(k)_ij ← min(d^(k−1)_ij, d^(k−1)_ik + d^(k−1)_kj)
return D^(n)
```

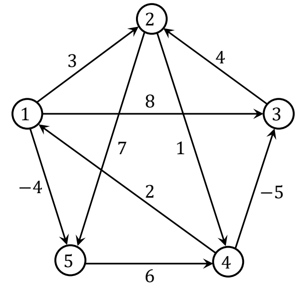
APSP: Floyd-Warshall with Predecessor Matrix

```
FLOYD-WARSHALL ( W )
n ← W.rows
D^(0) ← W
let Π^(0) = (π^(0)_ij) be a new n × n matrix
for i ← 1 to n do
    for j ← 1 to n do
        if i = j or w_ij = ∞ then π^(0)_ij ← NIL
        else π^(0)_ij ← i
for k ← 1 to n do
    let D^(k) = (d^(k)_ij) and Π^(k) = (π^(k)_ij) be new n × n matrices
    for i ← 1 to n do
        for j ← 1 to n do
            if d^(k−1)_ij ≤ d^(k−1)_ik + d^(k−1)_kj then π^(k)_ij ← π^(k−1)_ij
            else π^(k)_ij ← π^(k−1)_kj
            d^(k)_ij ← min(d^(k−1)_ij, d^(k−1)_ik + d^(k−1)_kj)
return D^(n) and Π^(n)
```

APSP: Floyd-Warshall with Predecessor Matrix

```
PRINT-ALL-PAIRS-SHORTEST-PATH ( Π, i, j )
    if i = j then
        print i
    elseif π_ij = NIL then
        print "no path from" i "to" j "exists"
    else PRINT-ALL-PAIRS-SHORTEST-PATH ( Π, i, π_ij )
        print j
```

APSP: Floyd-Warshall with Predecessor Matrix



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

APSP: Floyd-Warshall with Predecessor Matrix

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

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

APSP: Floyd-Warshall with Predecessor Matrix

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

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

APSP: Floyd-Warshall with Predecessor Matrix

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

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

APSP: Floyd-Warshall with Predecessor Matrix

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

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

APSP: Floyd-Warshall with Predecessor Matrix

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

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

APSP: Floyd-Warshall's Algorithm

FLOYD-WARSHALL (W)

1. $n \leftarrow W.rows$
2. $D^{(0)} \leftarrow W$
3. *for* $k \leftarrow 1$ *to* n *do*
4. let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
5. *for* $i \leftarrow 1$ *to* n *do*
6. *for* $j \leftarrow 1$ *to* n *do*
7. $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8. *return* $D^{(n)}$

Running Time $= \Theta(n^3)$

Space Complexity $= \Theta(n^3)$

APSP: Floyd-Warshall's Algorithm

But $D^{(k)}$ depends only on $D^{(k-1)}$.

```
FLOYD-WARSHALL-QUADRATIC-SPACE ( W )
n ← W.rows
let D^(0) = ( d_ij^(0) ) and D^(1) = ( d_ij^(1) ) be new n × n matrices
D^(0) ← W
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            d_ij^(1) ← min( d_ij^(0), d_ik^(0) + d_kj^(0) )
    D^(0) ← D^(1)
return D^(0)
```

Running Time $= \Theta(n^3)$

Space Complexity $= \Theta(n^2)$

APSP: Floyd-Warshall's Algorithm

Can be solved in-place!

```
FLOYD-WARSHALL-IN-PLACE ( W )
n ← W.rows
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            w_ij ← min( w_ij, w_ik + w_kj )
return W
```

Running Time $= \Theta(n^3)$

Space Complexity $= \Theta(n^2)$

103

104