Computer Algorithms

Segment 6

Shortest Path

SINGLE-SOURCE SHORTEST PATHS

Generalization of BFS to handle weighted graphs

- Direct Graph ***G*** = (***V***, ***E***), edge weight function; w : ***E*** → ***R***
- In BFS w(e)=1 for all e î E

Weight of path p = $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is $$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

Shortest Path = Path of minimum weight

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

Shortest-Path Variants

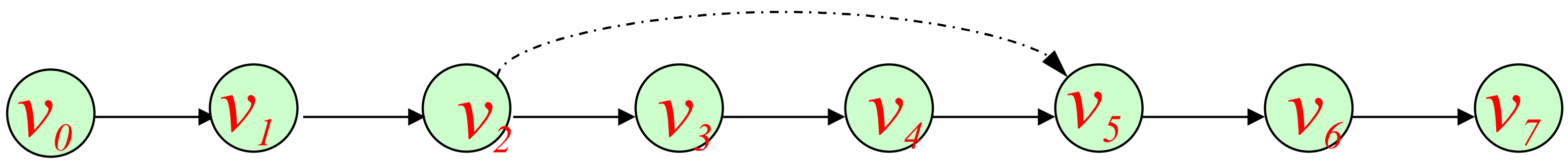
- Shortest-Path problems
  - Single-source shortest-paths problem: Find the shortest path from s to each vertex v. (e.g. BFS)
  - Single-destination shortest-paths problem: Find a shortest path to a given *destination* vertex t from each vertex v.
  - Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v.
  - All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.

Optimal Substructure Property

Theorem: Subpaths of shortest paths are also shortest paths

- Let $P_{1k} = \langle v_1, \ldots, v_k \rangle$ be a shortest path from v_1 to v_k
- Let $P_{ij} = \langle v_i, \ldots, v_j \rangle$ be subpath of P_{1k} from v_i to v_j for any i, j
- Then P_{ij} is a shortest path from v_i to v_j

Proof: By cut and paste



- If some subpath *were not* a shortest path
- We could substitute a shorter subpath to create a *shorter total path*
- Hence, the original path would not be shortest path

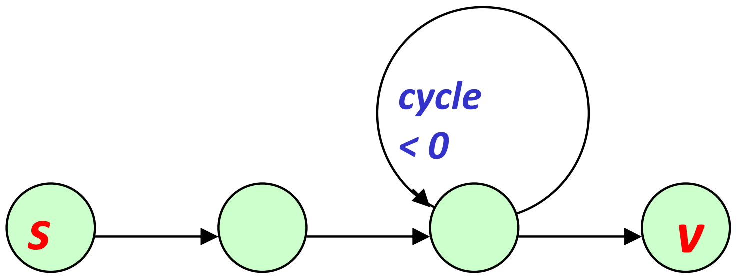
Optimal Substructure Property

Definition:

- $\delta(u,v)$ = weight of the shortest path(s) from u to v

Well Definedness:

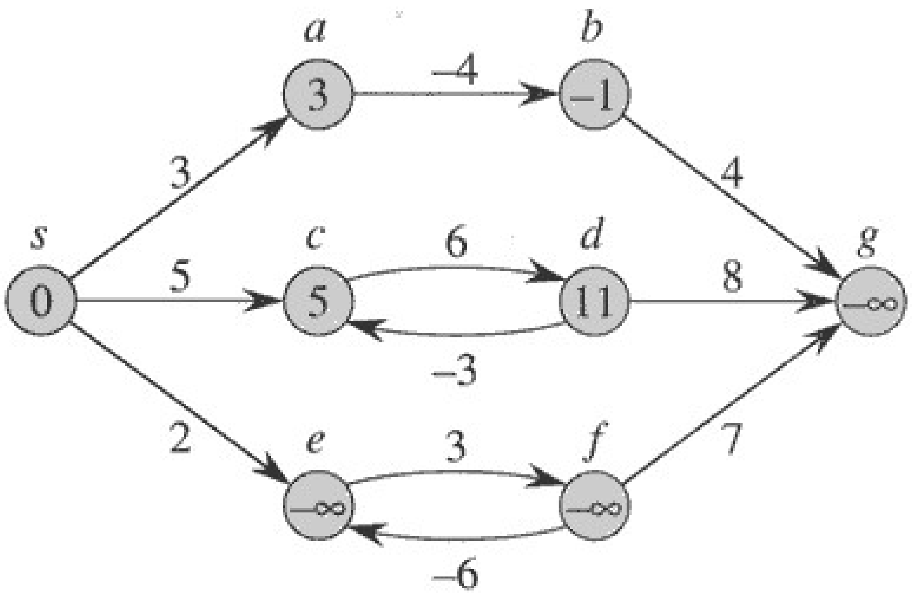
- *negative-weight cycle in graph*: Some shortest paths may not be defined
- *argument:* can always get a shorter path by going around the cycle again

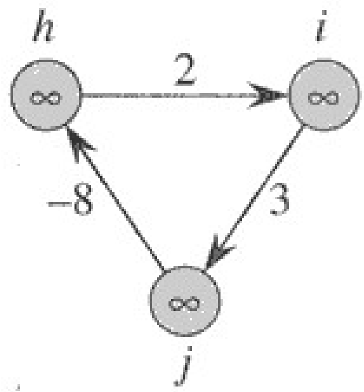


Negative-weight edges

- Negative weight edges can be allowed as long as there are no negative weight cycles
- If there are negative weight cycles, then there cannot be a shortest path from s to any node t (why?)
- If we disallow negative weight cycles, then there always is a shortest path that contains no cycles
- No problem, as long as no negative-weight cycles are reachable from the source
- Otherwise, we can just keep going around it, and

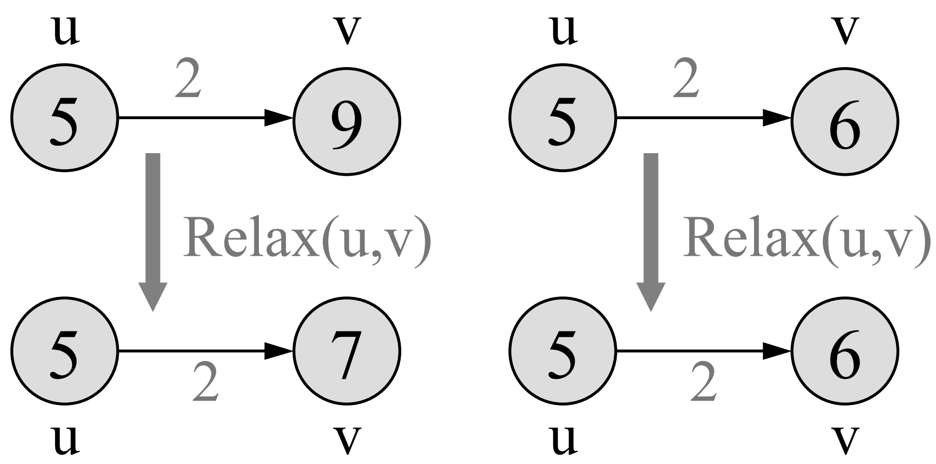
get $w(s, v) = -\infty$ for all v on the cycle.





Relaxation technique

- For each vertex v, we maintain an upper bound d[v] on the weight of shortest path from s to v
- d[v] initialized to infinity
- Relaxing an edge (u,v)
  - Can we improve the shortest path to v by going through u?
  - If d[v] > d[u] + w(u,v), d[v] = d[u] + w(u,v)
  - This can be done in O(1) time



Relaxation

Algorithms keep track of $d[v]$, $\pi[v]$. **Initialized** as follows:

```
Initialize(G, s)
    for each v ∈ V[G] do
        d[v] := ∞;
        π[v] := NIL
    d[s] := 0
```

These values are changed when an edge (u, v) is **relaxed**:

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
```


//Reference: Introduction to algorithm-CORMEN, page 585, 586

Properties of Relaxation

- Triangle inequality

  for a given vertex s Î V *and for every edge* (u,v) Î E,

  we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$
- Upper-bound property

  We always have $d[v] \geq \delta(s, v)$ for all vertices v ÎV, and once $d[v]$ achieves the value $\delta(s, v)$, it never changes.
- No-path property

  If there is no path from s to v, then $d[v] = \delta(s, v) = \infty$.
- Convergence property

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

Bellman-Ford Algorithm

- Can have negative-weight edges. Will "detect" reachable negative-weight cycles.
- Given a weighted, directed graph G=(V,E) with source *s* and weight function $w : E \rightarrow \mathbf{R}$, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.

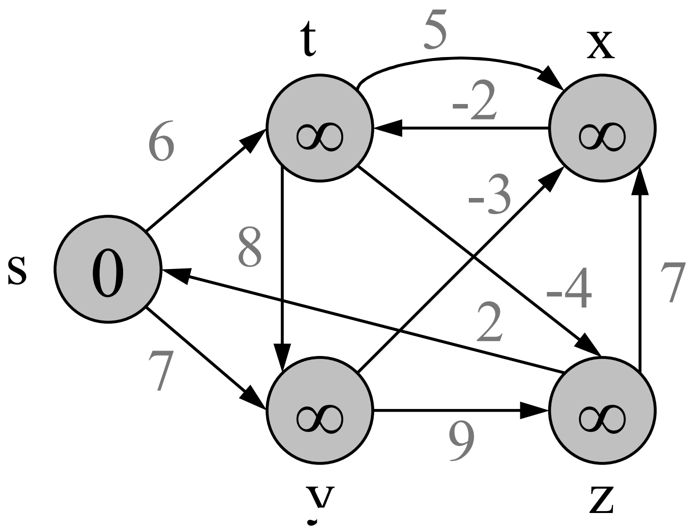
```
Bellman-Ford(G, w, s)
1.  Initialize(G, s);
2.  for i := 1 to |V[G]| –1 do
3.     for each (u, v) in E[G] do
4.           Relax(u, v, w)
5.  for each (u, v) in E[G] do
6.     if d[v] > d[u] + w(u, v) then
7.       return false
8.  return true
```

Time Complexity is O(VE).

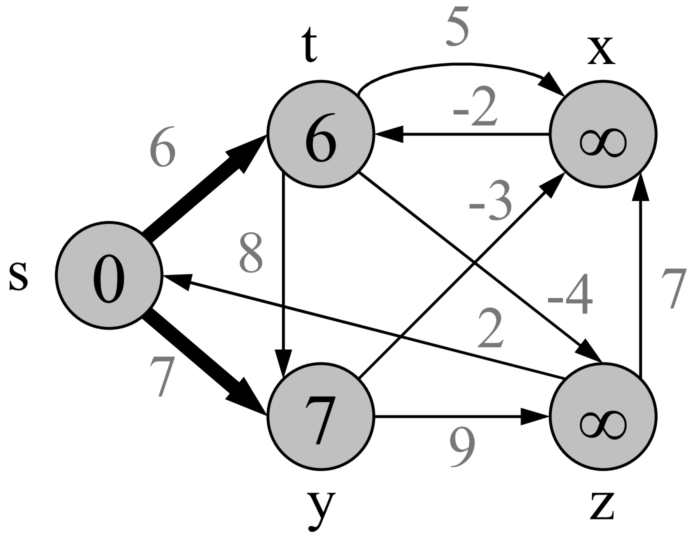
d and π values are the final values

//Reference: Introduction to algorithm-CORMEN, page 588, 586

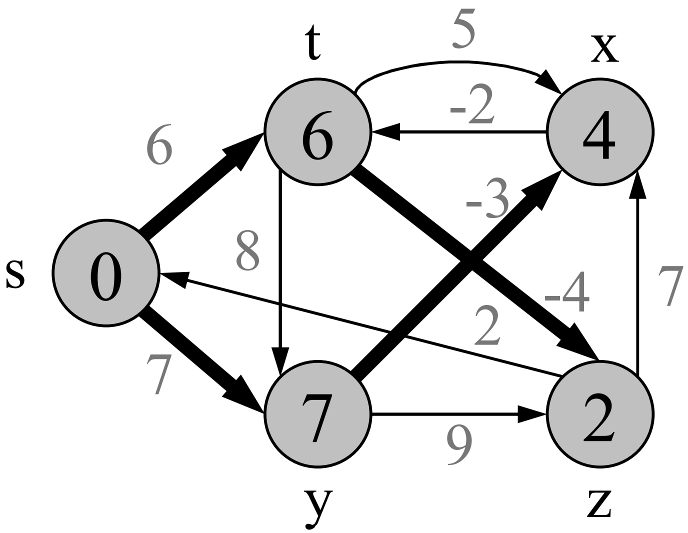
Bellman-Ford Algorithm (cont.)



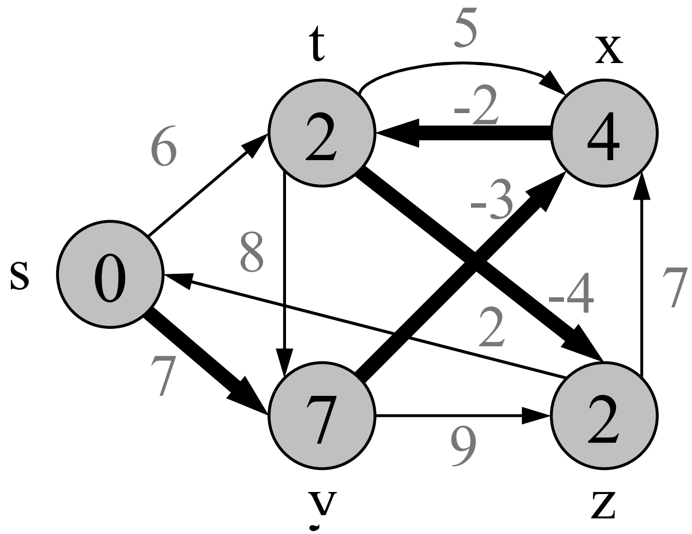
(a)



(b)

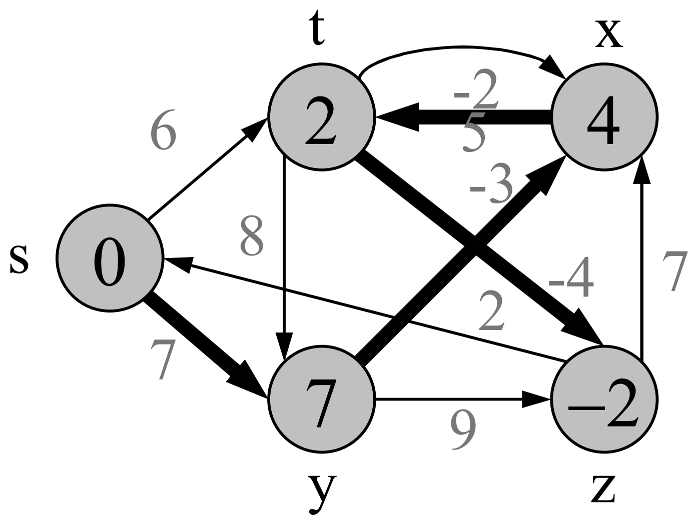


(c)



(d)

Bellman-Ford Algorithm (cont.)



(e)

d and π values in (e) are the final values

- Bellman-Ford running time:
  - $(|V|-1)|E| + |E| = \Theta(VE)$

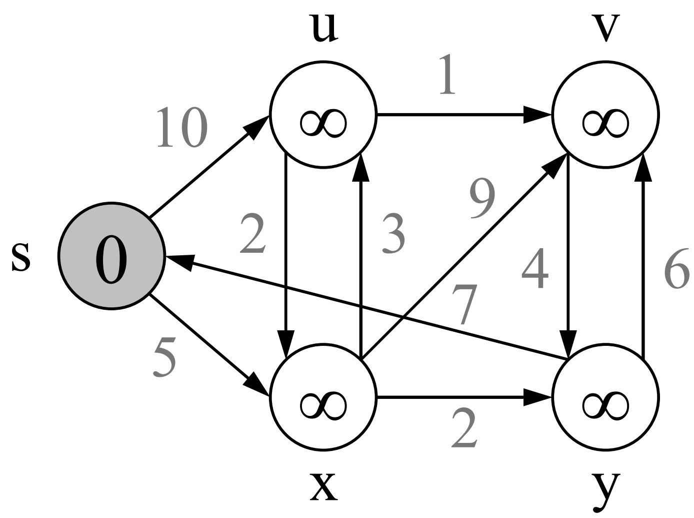
Dijkstra's Algorithm For Shortest Paths

- Non-negative edge weight
- Like BFS: If all edge weights are equal, then use BFS, otherwise use this algorithm
- Use Q = priority queue keyed on d[v] values (note: BFS uses FIFO)

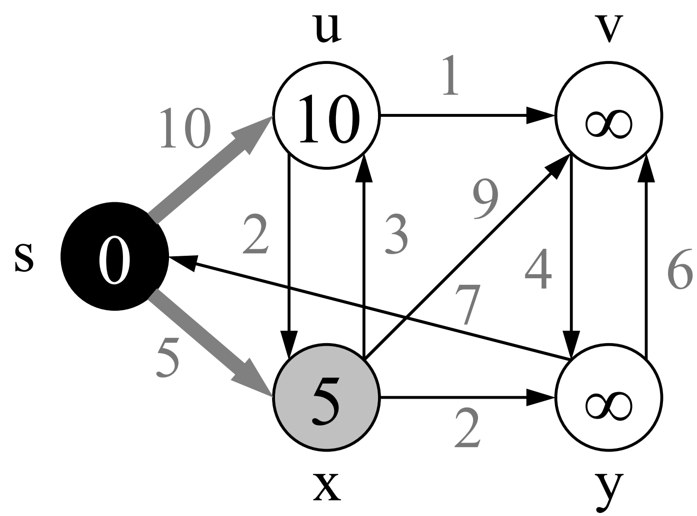
Dijkstra(G,w,s)
1. Initialize(G, s);
2. $S := \varnothing$;
3. $Q := V[G]$;
4. **while** $Q \neq \varnothing$ **do**
5. $\quad u :=$ Extract-Min(Q);
6. $\quad S := S \cup \{u\}$;
7. $\quad$ **for** each $v \in$ Adj[u] **do**
8. $\quad\quad$ Relax(u, v, w)

//Reference: Introduction to algorithm-CORMEN, page 595

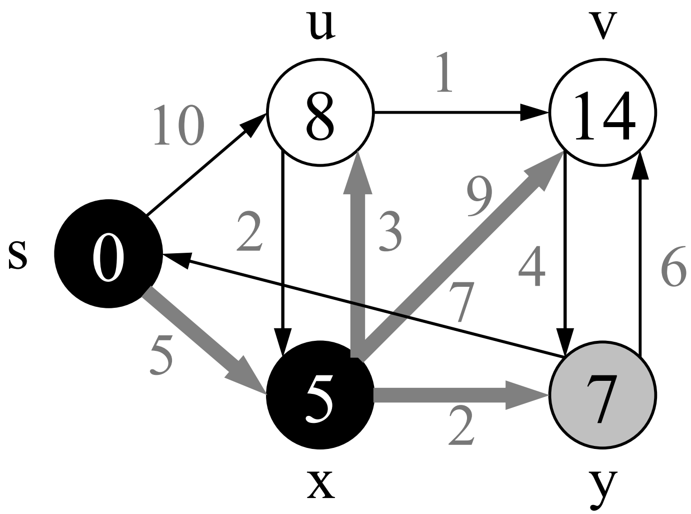
Dijkstra’s Algorithm For Shortest Paths



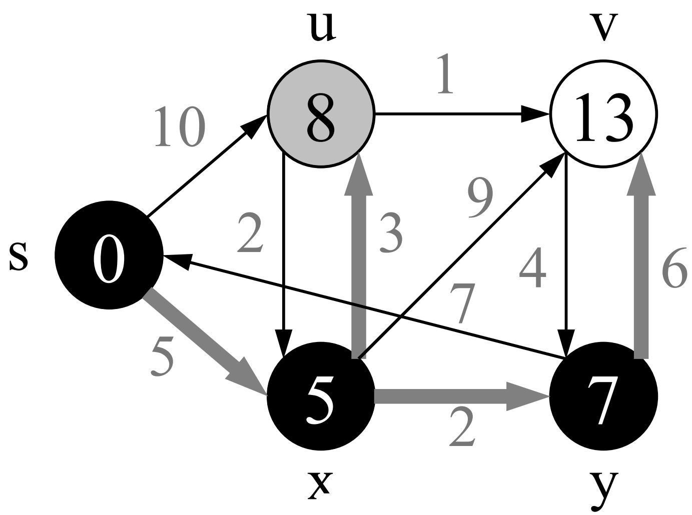
(a)



(b)

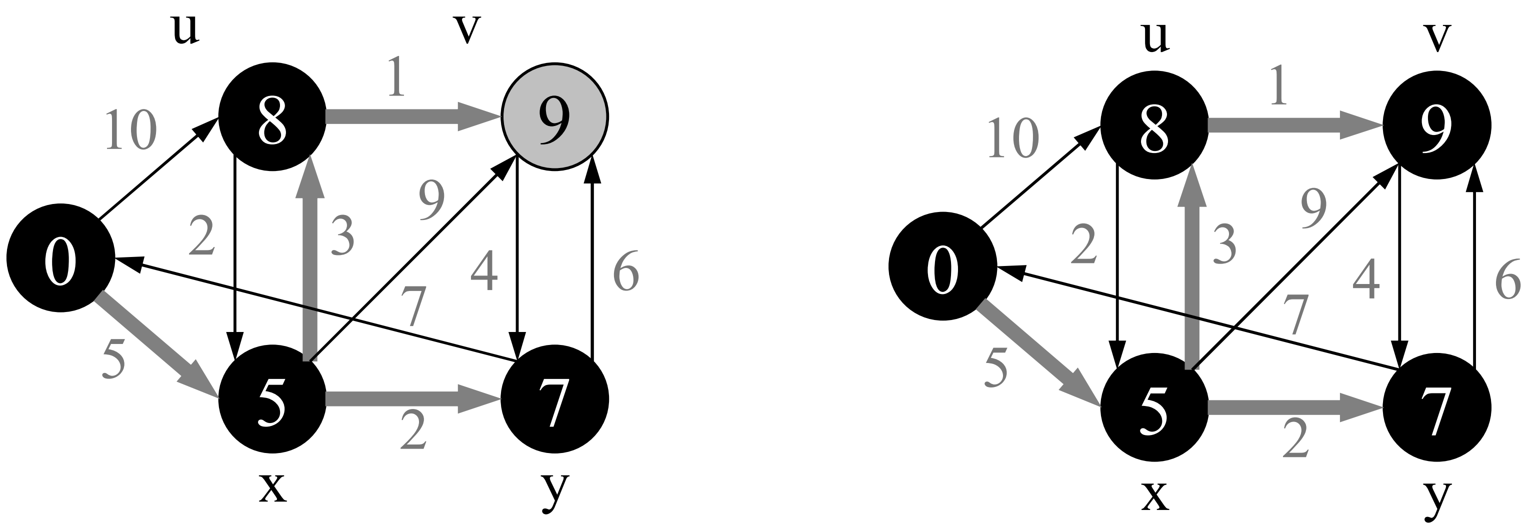


(c)



(d)

Dijkstra's Algorithm For Shortest Paths



(e) (f)

d and π values in (f) are the final values

Running Time Analysis cont'd

- O(E) edge relaxations
- Priority Queue operations
  - O(E) decrease key operations
  - O(V) extract-min operations
- Three implementations of priority queues
  - Array: $O(V^2)$ time
    - decrease-key is O(1) and extract-min is O(V)
  - Binary heap: O(E log V) time assuming E >= V
    - decrease-key and extract-min are O(log V)
  - Fibonacci heap: O(V log V + E) time
    - decrease-key is O(1) amortized time and extract-min is O(log V)

- Running time of Dijkstra's algorithm is lower than that of Bellman-Ford algorithm

All-Pairs Shortest Paths

- We now want to compute a table giving the length of the shortest path between any two vertices. (We also would like to get the shortest paths themselves.)
- Assume input graph is given by an adjacency matrix.

  W = (w_{ij}) where

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

- Let $d_{ij}^{(m)}$ = minimum weight of any path from vertex i to vertex j, containing at most m edges.

Dynamic programming algorithms for all-pairs shortest path

- We will study a new technique—dynamic programming algorithms (typically for optimization problems)
- **Ideas:**
  - Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution in a bottom-up fashion (using matrix to compute)
  - Backtracking to construct an optimal solution from computed information.

Floyd-Warshall algorithm for shortest path:

- Use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G=(V,E)$.
- The resulting algorithm, known as the Floyd-Warshall algorithm, runs in $O(V^3)$ time.
  - negative-weight edges may be present,
  - but we shall assume that there are no negative-weight cycles.

The structure of a shortest path:

- We use a different characterization of the structure of a shortest path than we used in the matrix-multiplication-based all-pairs algorithms.
- The algorithm considers the “intermediate” vertices of a shortest path, where an intermediate vertex of a simple path $p=\langle v_1,v_2,\ldots,v_l\rangle$ is any vertex in p other than v_1 or v_l, that is, any vertex in the set $\{v_2,v_3,\ldots,v_{l-1}\}$

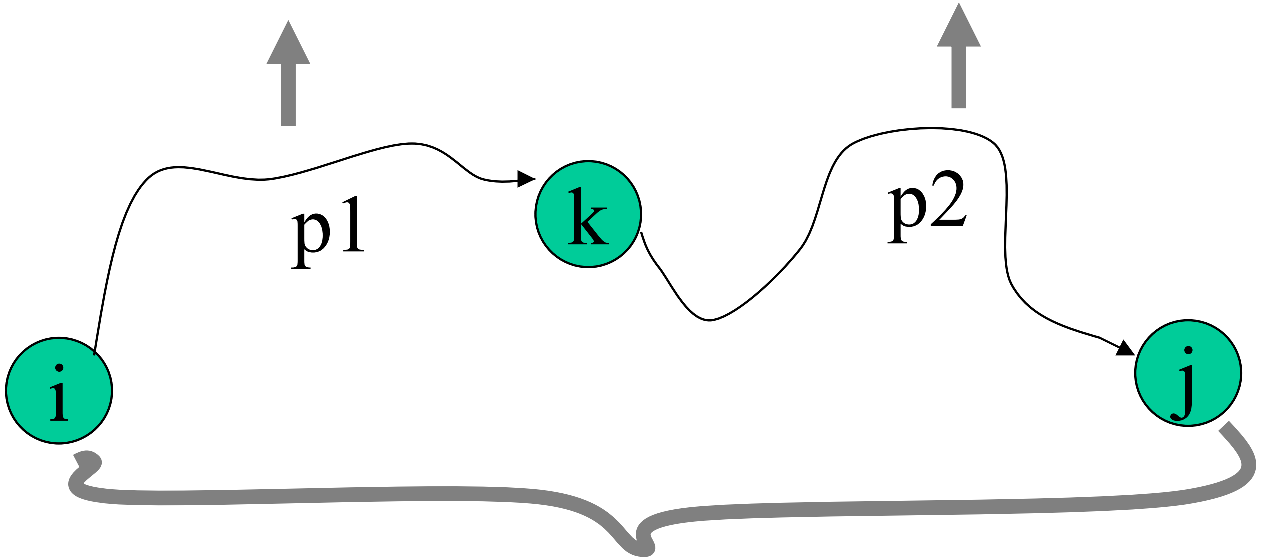
Continue:

- Let the vertices of G be V={1,2,…,n}, and consider a subset {1,2,…,k} of vertices for some k.
- For any pair of vertices $i,j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from {1,2,…,k},and let p be a minimum-weight path from among them.
- The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1,2,…,k-1}.

Relationship:

- The relationship depends on whether or not k is an intermediate vertex of path p.
- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1,2,…,k-1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1,2,…,k-1} is also a shortest path from i to j with all intermediate vertices in the set {1,2,…,k}.
- If k is an intermediate vertex of path p,then we break p down into $i \xrightarrow{p1} k \xrightarrow{p2} j$ as shown Figure 2.p1 is a shortest path from i to k with all intermediate vertices in the set {1,2,…,k-1}, so as p2.

All intermediate vertices in {1,2,…,k-1}



P:all intermediate vertices in {1,2,…,k}

Figure2. Path p is a shortest path from vertex i to vertex j,and k is the highest-numbered intermediate vertex of p. Path p1, the portion of path p from vertex i to vertex k,has all intermediate vertices in the set {1,2,…,k-1}.The same holds for path p2 from vertex k to vertex j.

A recursive solution to the all-pairs shortest paths problem:

- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set {1,2,…,k}. A recursive definition is given by
- $$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

- The matrix $D^{(n)}=(d_{ij}^{(n)})$ gives the final answer-$d_{ij}^{(n)}=\delta(i,j)$ for all $i,j \in V$-because all intermediate vertices are in the set {1,2,…,n}.

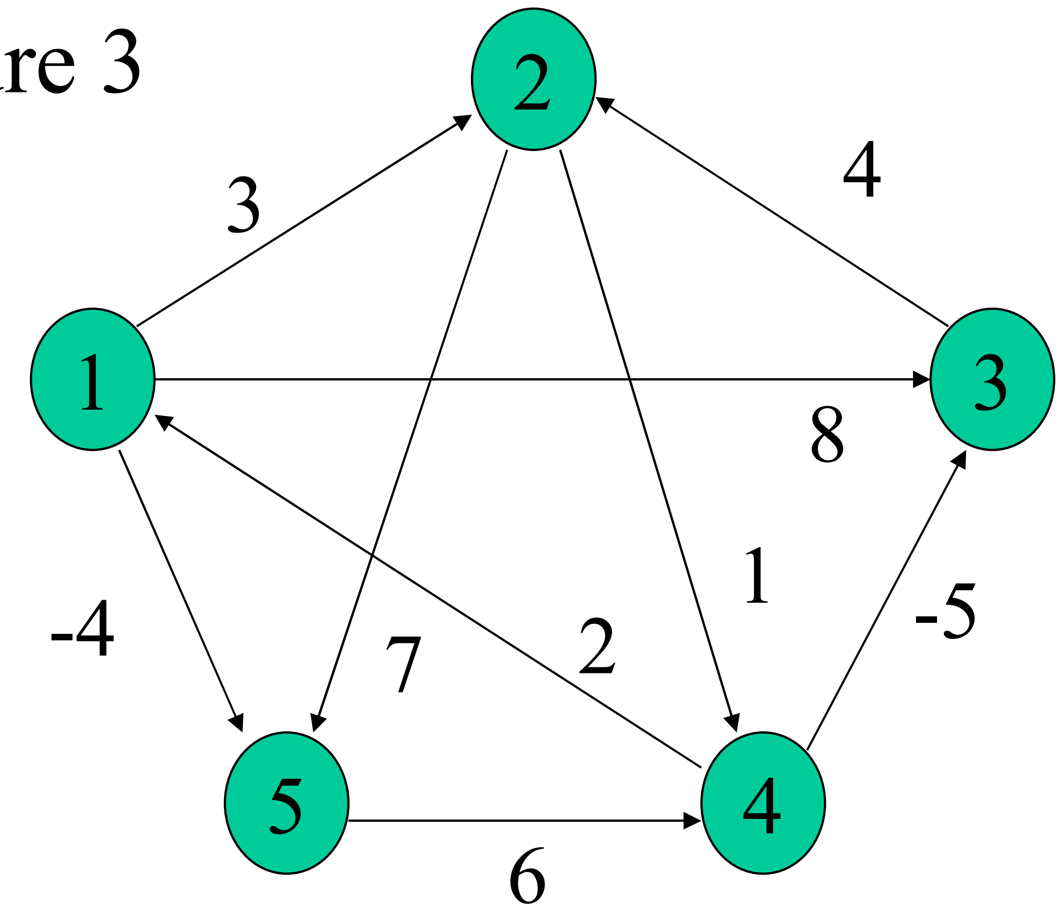
Computing the shortest-path weights bottom up:

- FLOYD-WARSHALL(W)
- n $\leftarrow$ rows[W]
- $D^{(0)} \leftarrow W$
- for k $\leftarrow$ 1 to n
-     do for i $\leftarrow$ 1 to n
-         do for j $\leftarrow$ 1 to n
-             $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

- return $D^{(n)}$

Example:

- Figure 3



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

$$\prod(0)=\begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

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

$$\prod(1)=\begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

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

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

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

$$\prod(4)=\begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

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

$$\prod(5)=\begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Constructing a shortest path:

$\pi_{ij}^{(k)}$: the predecessor of vertex j on a shortest path from i with all intermediate vertices in the set {1, 2, …, k}

$$\pi_{ij}^{(0)} = \begin{cases} NIL & if\ i = j\ or\ w_{ij} = \infty \\ i & if\ i \neq j\ and\ w_{ij} < \infty \end{cases} \quad \text{........(25.6)}$$

For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$, then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & if\ d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & if\ d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} \quad \text{......25.7}$$

Print a shortest path from vertex i to vertex j

Print-All-Pairs-Shortest-Path(Π, i,j)

1. **if** i=j

2. then print i

3. **else if** $\pi_{ij} = NIL$

4. **then** print "no path from" i "to" j "exists"

5. **else** Print-All-Pairs-Shortest-Path(Π, i, π_{ij})

6. print j