Lecture 17: Shortest Paths

Shortest Path Problem

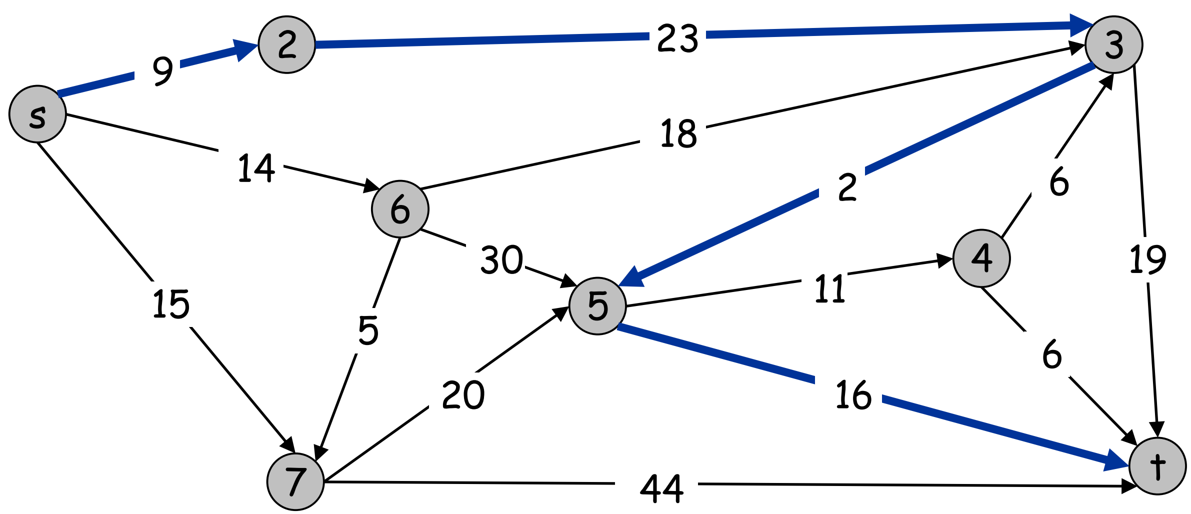
Input:

- Directed graph $G = (V, E)$.
  - An undirected edge is considered as two directed edges.
- Source s, destination t.
- Weight $w(e)$ = length of edge e.

Shortest path problem: Find the shortest path from s to t.

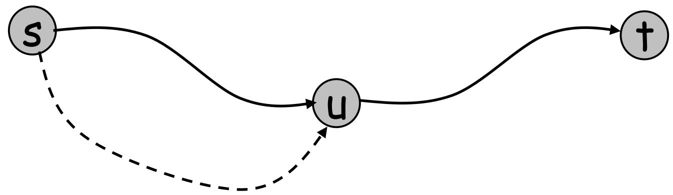
Single-source shortest path: Find the shortest path from s to every node.

Def: The distance from u to v is the length of the shortest path from u to v, denoted as $\delta(u, v)$



$\delta(s, t) = 9 + 23 + 2 + 16$
$= 50.$

Key property of shortest path: Subpath optimality



Lemma: Let $P = (s, \ldots, u, .., t)$ be the shortest path from s to t. Then the subpaths $(s, \ldots, u)$ and $(u, .. t)$ must also be shortest paths from s to u and from u to t, respectively.

Pf: (by contradiction)

- Suppose the subpath $(s, \ldots, u)$ is not the shortest, and there is another path P' from s to u that is shorter.
- Then we can replace the subpath from s to u with P', which will make the whole path from s to t shorter.
- This contradicts with the fact that the original path from s to t is the shortest.
- Same proof works for the subpath from u to t.

Note: This holds for any subpath.

Two easy variants

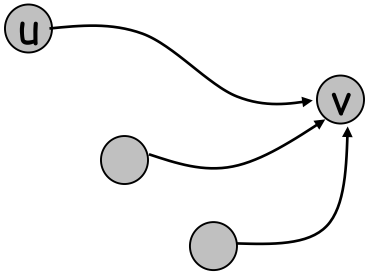
If all weights are 1 (or all weights are equal):

- Can be solved by BFS in $\Theta(V+E)$ time.

If the graph is a DAG:

- Can use dynamic programming
- By subpath optimality, we have

$$\delta(s,v) = \min_{u,(u,v)\in E}\{\delta(s,u) + w(u,v)\}$$

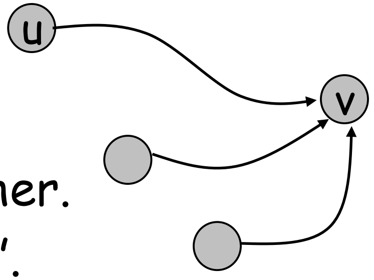


- We can compute all the $\delta(s,v)$'s in the topological order of nodes.

Shortest path in a DAG

```
DAG-Shortest-Path(G,s)
topologically sort the vertices of G
reverse every edge of G
for each vertex v ∈ V
      v.d ← ∞
      v.p ← nil
s.d ← 0
for each vertex v in topological order
      for each vertex u ∈ Adj[v]
            if v.d > u.d + w(u,v) then      ┐
                  v.d ← u.d + w(u,v)         ├ Relax(u,v)
                  v.p ← u                    ┘
```

A nice trick to avoid reversing all edges:



- $v.d$ starts with ∞.
- Incoming edges do not have to be evaluated together.
- We are OK as long as all edges have been "relaxed".
  - Here "relax" means this edge no longer needs to be considered.

Shortest path in a DAG: Final algorithm

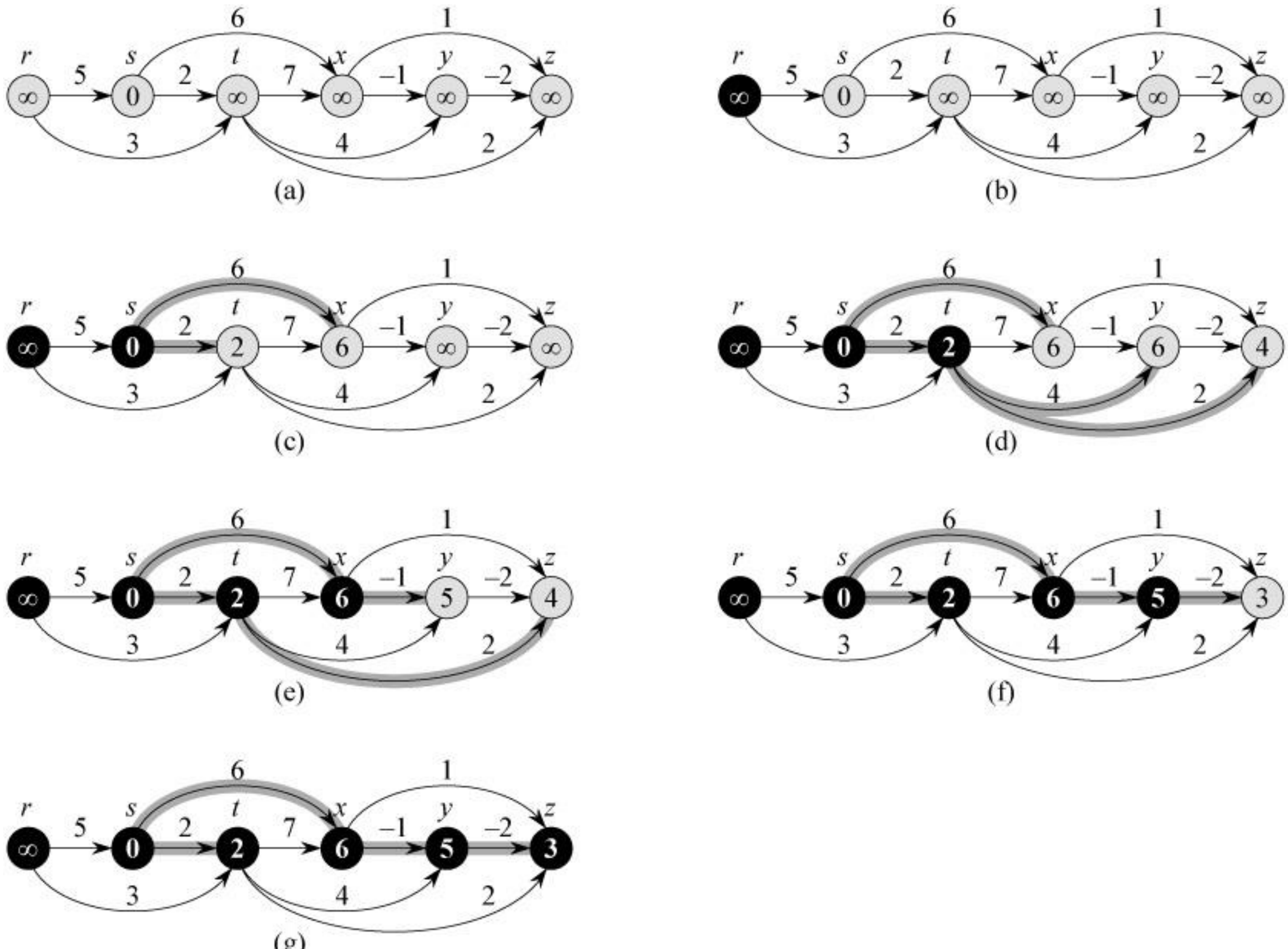
```
DAG-Shortest-Path(G, s)
topologically sort the vertices of G
for each vertex v ∈ V
    v.d ← ∞
    v.p ← nil
s.d ← 0
for each vertex u in topological order
    for each vertex v ∈ Adj[u]
        if v.d > u.d + w(u, v) then      ┐
            v.d ← u.d + w(u, v)          ├ Relax(u, v)
            v.p ← u                      ┘
```

Running time: $\Theta(V + E)$

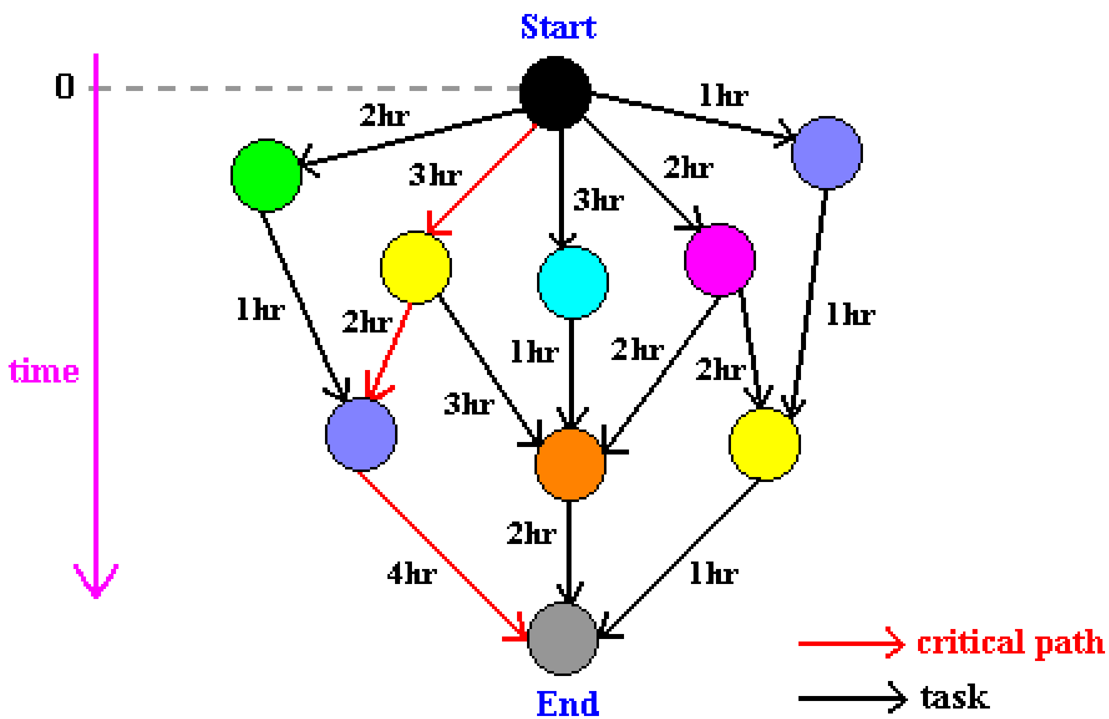
Note:

- Can find the actual shortest path by tracing the parent pointers.
- If we just want to find the shortest path from s to t, can stop the algorithm when $u = t$. But this does not reduce the running time asymptotically.

Shortest path in a DAG: Example



Longest Paths in a DAG



Modified recurrence:

- Let $d(s, v)$ be the longest distance from s to v.

$$d(s, v) = \max_{u,(u,v) \in E} \{d(s, u) + w(u, v)\}$$

Q: What if we use nodes to model tasks, and edges to model the dependencies?

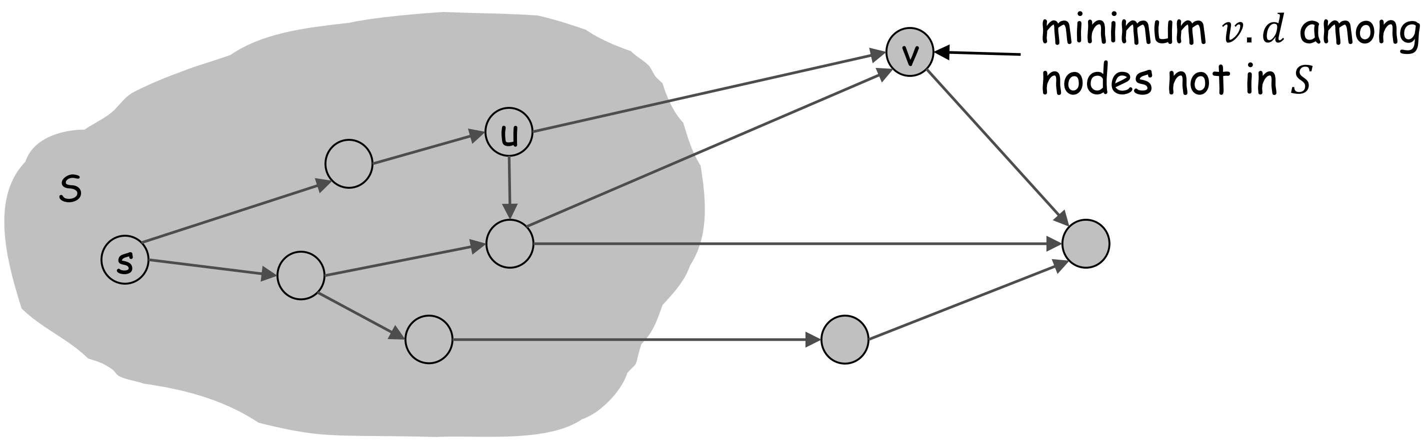
Shortest paths in a graph with cycles and nonnegative weights

Def: $\delta(s, v)$ = minimum distance from s to v.

Challenge: The same recurrence holds, but there is no order to compute the recurrence if the graph has cycles.

Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have $u.d = \delta(s, u)$. Initialize $S = \{s\}, s.d = 0, v.d = \infty$
- Key lemma: If all edges leaving S are relaxed, then $v.d = \delta(s, v)$, where v is the vertex in $V - S$ with the minimum $v.d$.
  - So this v can be added to S, and we then repeat.



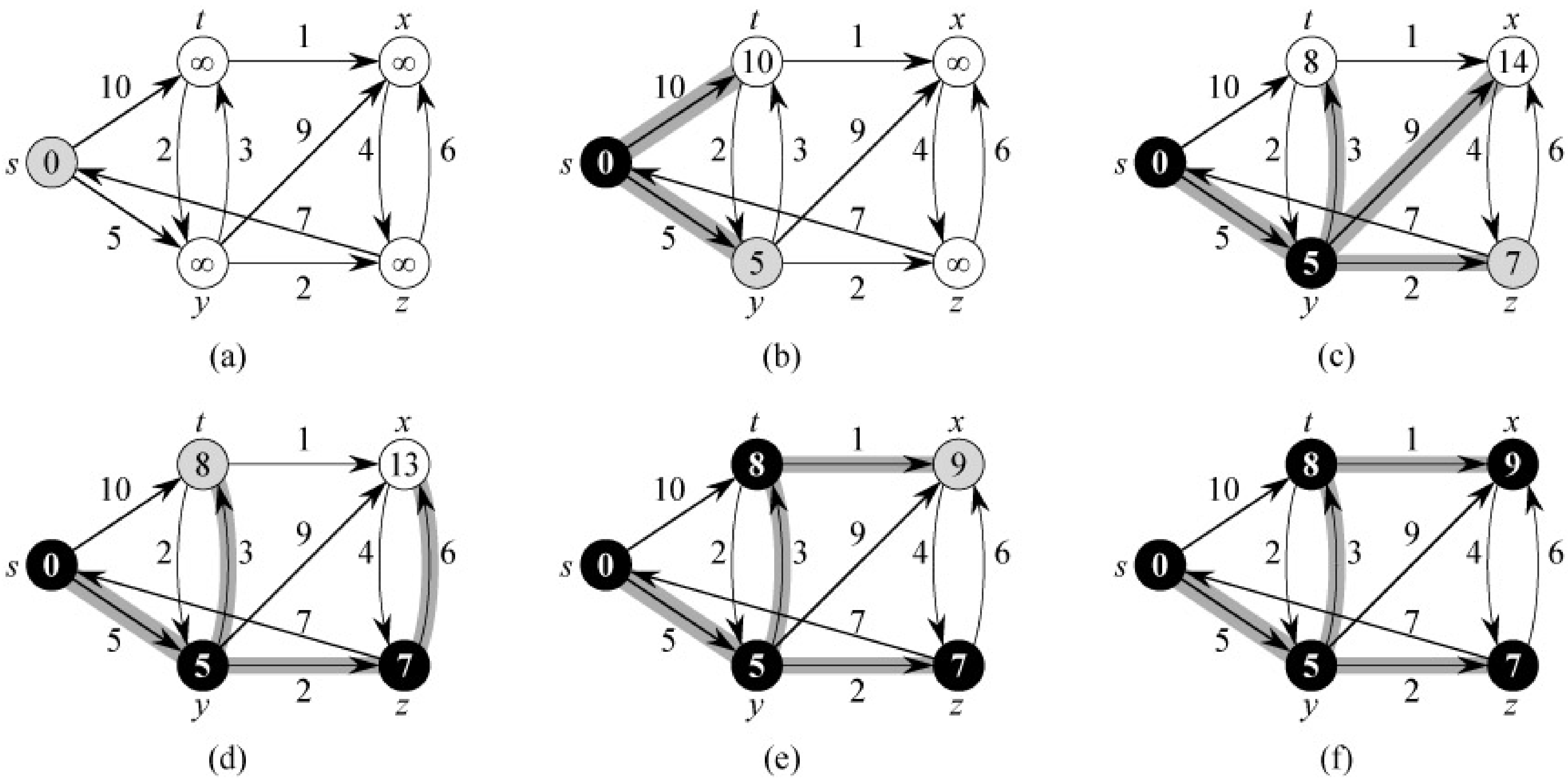
Dijkstra's Algorithm

```
Dijkstra(G,s):
for each v ∈ V do
      v.d ← ∞, v.p ← nil, v.color ← white
s.d ← 0
create a min priority queue Q on V with d as key
while Q ≠ ∅
      u ← Extract-Min(Q)
      u.color ← black
      for each v ∈ Adj[u] do
            if v.color = white and u.d + w(u,v) < v.d then
                  v.p ← u
                  v.d ← u.d + w(u,v)
                  Decrease-Key(Q,v,v.d)
```

Running time: $O(E \log V)$

- Very similar to Prim's algorithm with only one key difference
- Try to run both algorithms on the same graph to see the difference.

Dijkstra's Algorithm: Example



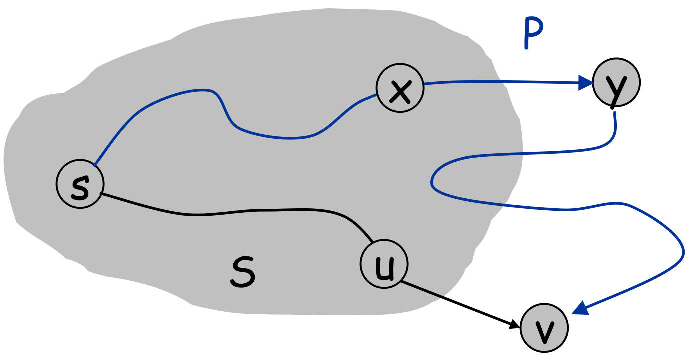
Note: All the shortest paths found by Dijkstra's algorithm form a tree (shortest-path tree).

Dijkstra's Algorithm: Correctness

Lemma. If $u.d = \delta(s,u)$ for all $u \in S$, and all edges leaving S are relaxed, we have $v.d = \delta(s,v)$, where v is the vertex with the minimum $v.d$ in $V - S$.

Pf. (by contradiction)

- Suppose $v.d \neq \delta(s,v)$
  - Since $v.d$ starts with ∞, and whenever it's updated, we must have found a path with distance $v.d$. So we always have $v.d \geq \delta(s,v)$.
  - Thus it can only be $v.d > \delta(s,v)$.



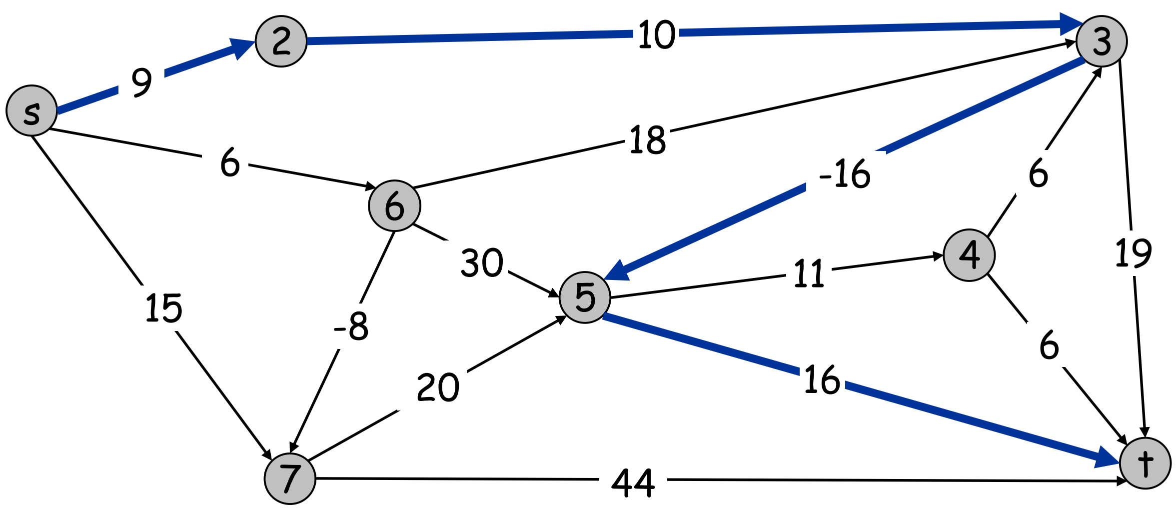
- Consider the shortest path P from s to v.
  - Suppose $x \to y$ is the first edge on P that takes P out of S.
  - Since $x \in S$, we have $x.d = \delta(s,x)$.
  - The edge $x \to y$ has been relaxed, so $y.d \leq x.d + w(x,y)$.
  - P is shortest path, its subpath $(s, \dots, x, y)$ must also be shortest, so $x.d + w(x,y) = \delta(s,y)$.
  - $\delta(s,y) \leq \delta(s,v)$, assuming nonnegative weights.
  - Thus, $v.d > \delta(s,v) \geq \delta(s,y) = x.d + w(x,y) \geq y.d$, contradicting with the fact that $v.d$ is the smallest in $V - S$.

Shortest paths on graphs with negative-weight edges

Shortest path problem. Given a directed graph $G = (V, E)$, with edge weights that may be both positive and negative, find shortest path from node s to every other node.

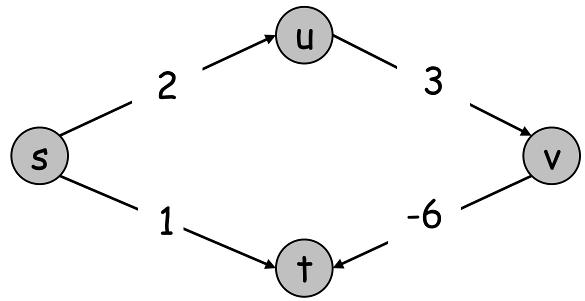
Applications.

- Road network: scenic roads
- Financial transactions: edges may be have positive or negative costs (profit)

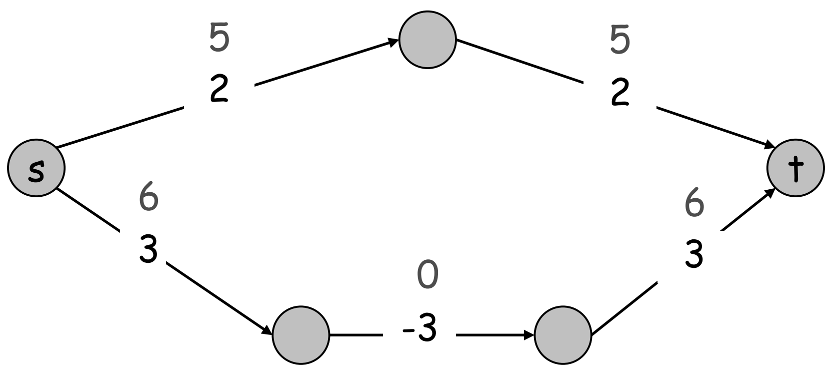


Shortest Paths with Negative Weights: Failed Attempts

Dijkstra. Can fail if negative edge costs.

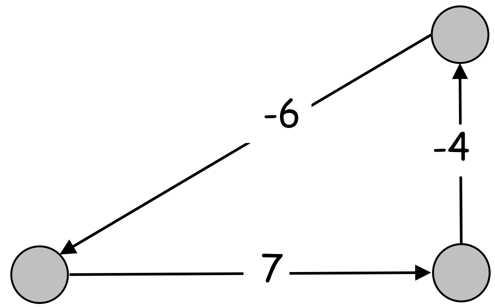


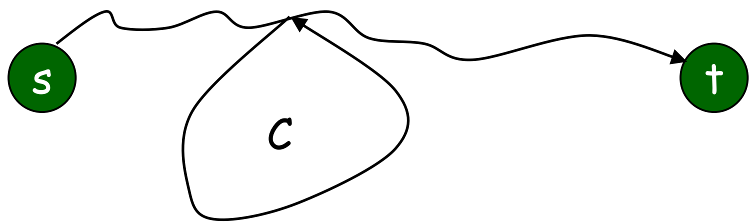
Re-weighting. Adding a constant to every edge weight can fail.



Shortest Paths: Negative Weight Cycles

Negative weight cycle.





$w(C) < 0$

Note. The shortest path problem is not be well defined if there are negative-weight cycles in the graph. So will assume no negative cycles.

Dynamic Programming

Def. $v.d[i]$ = length of shortest path from s to v using up to i edges.

Recurrence:

- Suppose (u, v) is the last edge of the shortest path from s to v. The subpath from s to u must also be shortest, which consists of at most $i - 1$ edges, followed by (u, v).

$$v.d[i] = \min_{u,(u,v)\in E} \{v.d[i-1] + w(u, v)\}$$
$$v.d[0] = \infty$$
$$s.d[i] = 0, \text{for all } i$$

Remark. $v.d[n-1]$ = length of the shortest path from s to v, since no shortest path can have n edges or more.

Dynamic Programming: Implementation

```
Shortest-Path(G, s):
for each node v ∈ V do
    for i ← 0 to n − 1 do
        v.d[i] ← ∞
for i ← 0 to n − 1 do
    s.d[i] ← 0
for i ← 1 to n − 1
    for each edge (u, v) ∈ E
        if u.d[i − 1] + w(u, v) < v.d[i] then
            v.d[i] ← u.d[i − 1] + w(u, v)
            v.p[i] ← u
```

Analysis. $\Theta(VE)$ time, $\Theta(V^2)$ space.

Improvements and simplifications

Improvements.

- Use only one $v.d$ instead of $v.d[i]$
  - After the i-th iteration, $v.d \leq v.d[i]$
  - This may make things even better (faster convergence).
- Use only one $v.p$ instead of $v.p[i]$
  - $v.p$ is always the last stop to v on the shortest path found so far.
- No need to check edges of the form (u, v) unless $u.d$ changed in previous iteration.
- If no $v.d$ has changed in an iteration, terminate the algorithm.

Bellman-Ford: Efficient Implementation

```
Bellman-Ford(G,s):
for each node v ∈ V
    v.d ← ∞, v.p ← nil
s.d ← 0
for i ← 1 to n − 1
    for each node u ∈ V
        if u.d is changed in previous iteration then
            for each v ∈ Adj[u]
                if u.d + w(u,v) < v.d then
                    v.d ← u.d + w(u,v)
                    v.p ← u
    if no v.d changed in this iteration then terminate
```

Analysis.

- $O(VE)$ time in the worst case, but can be much faster in practice
- $O(V)$ space.

Remark:

- Can be run in parallel.
- Used on massive graphs (even if no negative edges).
- Can also detect whether there is a negative cycle (see textbook).

All-Pairs Shortest Paths

Input:

- Directed graph $G = (V, E)$.
- Weight $w(e)$ = length of edge e.

Output:

- $\delta(u, v)$, for all pairs of nodes u, v.
- A data structure from which the shortest path from u to v can be extracted efficiently, for any pair of nodes u, v
  - Note: Storing all shortest paths explicitly for all pairs requires $O(V^3)$ space.

Graph representation

- Assume adjacency matrix
  - $w(u, v)$ can be extracted in $O(1)$ time.
  - $w(u, u) = 0$, $w(u, v) = \infty$ if there is no edge from u to v.
- If the graph is stored in adjacency lists format, can convert to adjacency matrix in $O(V^2)$ time.

Using previous algorithms

When there are no negative cost edges

- Apply Dijkstra's algorithm to each vertex (as the source).
- Recall that Dijkstra algorithm runs in $O(E \log V)$
- This gives an $O(VE \log V)$-time algorithm
- If the graph is dense, this is $O(n^3 \log n)$.

When negative-weight edges are present

- Run the Bellman-Ford algorithm from each vertex.
- $O(V^2E)$ time, which is $O(n^4)$ for dense graphs.

Dynamic Programming: Solution 1

Def: $d_{ij}^{(m)}$ = length of the shortest path from i to j that contains at most m edges.

- Use $D^{(m)}$ to denote the matrix $\left[d_{ij}^{(m)}\right]$.

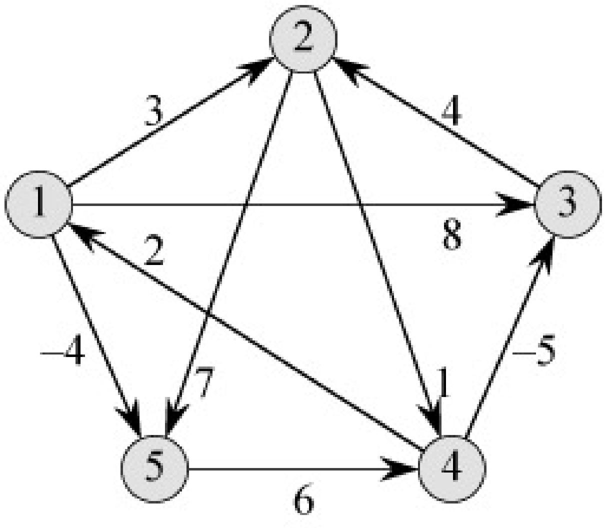
Recurrence: (Essentially the same as in Bellman-Ford)

$$d_{ij}^{(m)} = \min_{1 \le k \le n} \{d_{ik}^{(m-1)} + w(k,j)\}$$
$$d_{ij}^{(1)} = w(i,j)$$

Goal: $D^{(n-1)}$, since no shortest path can have n edges or more.

```
Slow-All-Pairs-Shortest-Paths(G):
d_ij^(1) = w(i,j) for all 1 ≤ i,j ≤ n
for m ← 2 to n − 1
    let D^(m) be a new n × n matrix
    for i ← 1 to n
        for j ← 1 to n
            d_ij^(m) ← ∞
            for k ← 1 to n
                if d_ik^(m−1) + w(k,j) < d_ij^(m) then d_ij^(m) ← d_ik^(m−1) + w(k,j)
return D^(n−1)
```

Example



Analysis:

- $O(n^4)$ time
- $O(n^3)$ space, can be improved to $O(n^2)$

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

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

Dynamic Programming: Solution 2

Observation:

- To compute $d_{ij}^{(m)}$, instead of looking at the last stop before j, we look at the middle point.
- This can cut down the problem size by half.

New recurrence:

$$d_{ij}^{(2s)} = \min_{1 \le k \le n} \{d_{ik}^{(s)} + d_{kj}^{(s)}\}$$

Algorithm:

- We can calculate $D^{(1)}, D^{(2)}, D^{(4)}, D^{(8)}, \ldots$
- Each matrix takes $O(n^3)$ time, total time $O(n^3 \log n)$.

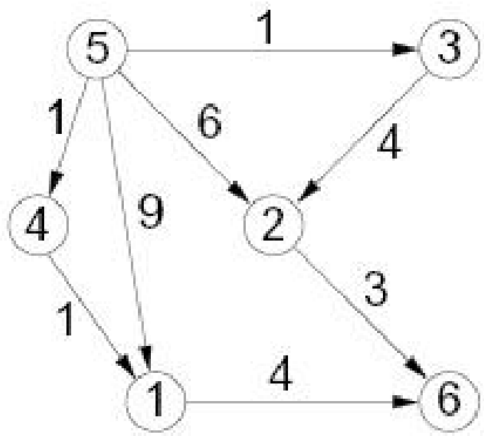
Q: We will overshoot $D^{(n-1)}$?

A: It's OK. Since $D^{(n')}, n' > n-1$ has the shortest paths with up to n' edges, it will not miss any shortest path with up to $n-1$ edges.

- Actually, $D^{(n')} = D^{(n-1)}$ for any $n' > n-1$, since no shortest path has more than $n-1$ edges.

Dynamic Programming: Solution 3 (Floyd-Warshall)

Def: $d_{ij}^{(k)}$ = length of the shortest path from i to j that such that all intermediate vertices on the path (if any) are in the set $\{1,2,\ldots,k\}$.



$d_{5,6}^{(0)}$ = INF (no path)

$d_{5,6}^{(1)} = 13$ (5,1,6)

$d_{5,6}^{(2)} = 9$ (5,2,6)

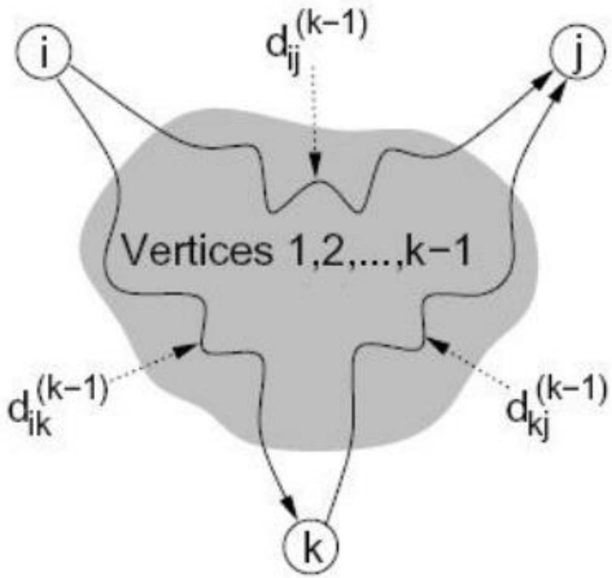
$d_{5,6}^{(3)} = 8$ (5,3,2,6)

$d_{5,6}^{(4)} = 6$ (5,4,1,6)

Initially: $d_{ij}^{(0)} = w(i,j)$

Goal: $D^{(n)}$

Recurrence



$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

To compute $d_{ij}^{(k)}$, there are two cases:

- Case 1: k is not a vertex on the shortest path from i to j, then the path uses only vertices in $\{1,2,\dots,k-1\}$.
- Case 2: k is an intermediate node on the shortest path from i to j, then the path can be divided into a subpath from i to k and a subpath from k to j. Both subpaths use only vertices in $\{1,2,\dots,k-1\}$

The Floyd-Warshall Algorithm

```
Floyd-Warshall(G):
d_ij^(0) = w(i,j) for all 1 ≤ i,j ≤ n
for k ← 1 to n
    let D^(k) be a new n × n matrix
    for i ← 1 to n
        for j ← 1 to n
            if d_ik^(k-1) + d_kj^(k-1) < d_ij^(k-1) then
                d_ij^(k) ← d_ik^(k-1) + d_kj^(k-1)
            else
                d_ij^(k) ← d_ij^(k-1)
return D^(n)
```

Analysis:

- $O(n^3)$ time
- $O(n^3)$ space, but can be improved to $O(n^2)$

Surprising discovery: If we just drop all the superscripts, i.e., the algorithm just uses one $n \times n$ array D, the algorithm still works! (why?)

The Floyd-Warshall Algorithm: Final Version

```
Floyd-Warshall(G):
d_ij = w(i,j) and intermed[i,j] ← 0 for all 1 ≤ i,j ≤ n
for k ← 1 to n
    for i ← 1 to n
        for j ← 1 to n
            if d_ik + d_kj < d_ij then
                d_ij ← d_ik + d_kj
                intermed[i,j] ← k
return D
```

Analysis:

- $O(n^3)$ time
- $O(n^2)$ space

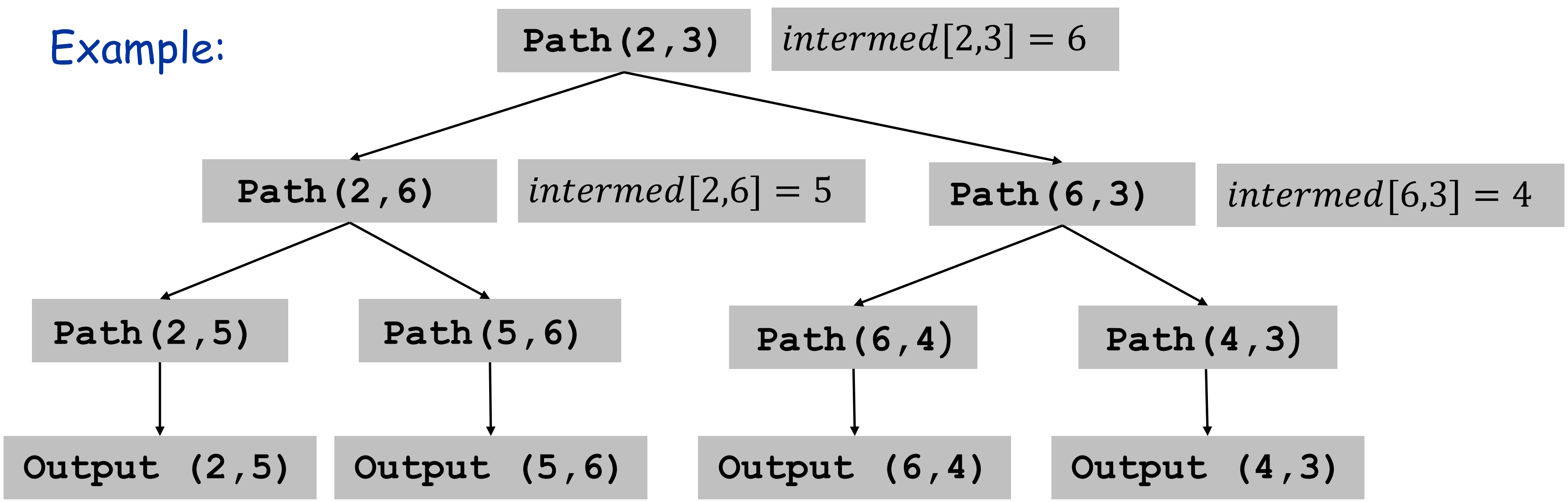
The $intermed[i, j]$ array records one intermediate node on the shortest path from i to j.

- It is nil if the shortest path does not pass any intermediate nodes.

Extracting Shortest Paths

Path(i,j)**:**
if $intermed[i,j] = nil$ **then**
    output (i,j)
else
    Path$(i, intermed[i,j])$
    Path$(intermed[i,j], j)$

Example:



Running time: O(length of the shortest path)