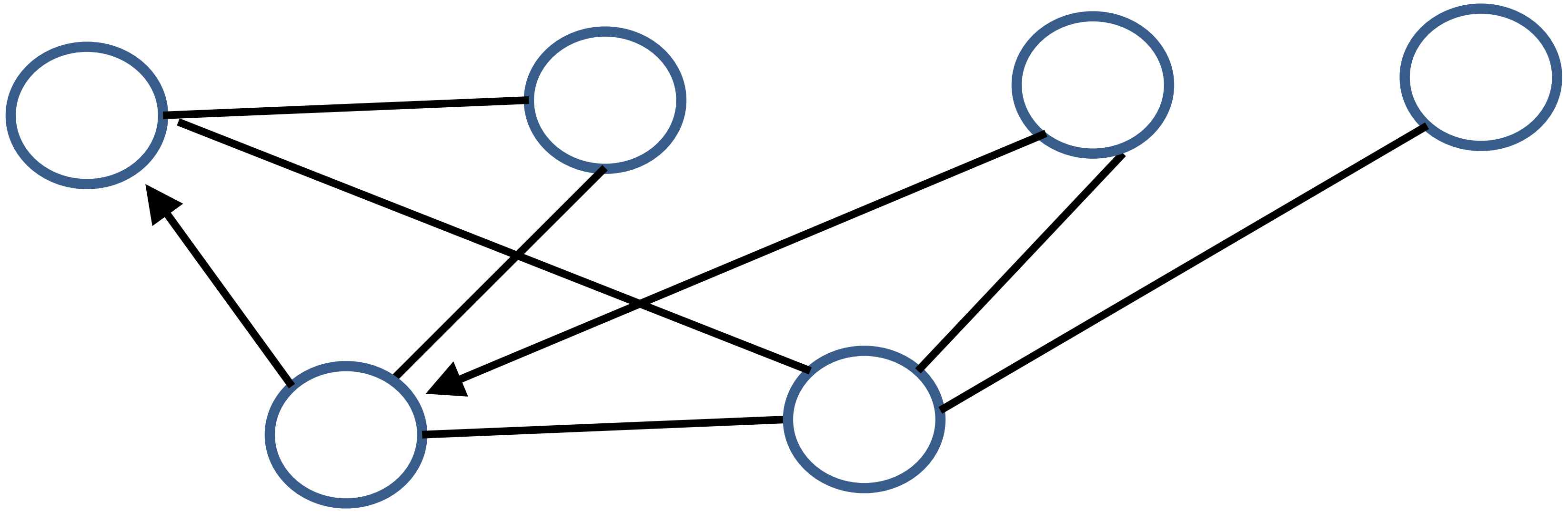
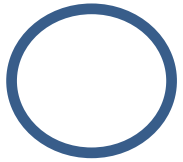
Graph Algorithms

Graphs

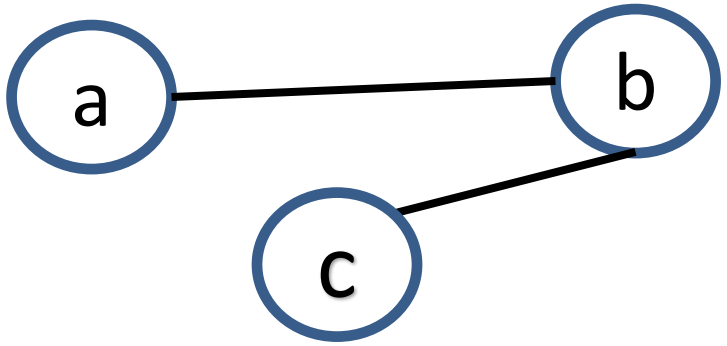


Nodes/vertexes: 

Edges: (undirected)

 (directed)

Representations of graph G with vertices V and edges E



- V x V adjacency-matrix A: $A_{u,v} = 1 \Leftrightarrow (u, v) \in E$

  Size: $|V|^2$

  Better for dense graphs, i.e., $|E| = \Omega(|V|^2)$

	a	**b**	**c**
a	0	1	0
b	1	0	1
c	0	1	0

- Adjaceny-list, e.g. (v_1, v_5), (v_1, v_{17}), (v_2, v_3) …

  Size: O(E)

  Better for sparse graphs, i.e., $|E| = O(|V|)$

Adj[a] = b

Adj[b] = a,c

Adj[c] = b

Next we see several algorithms to compute shortest distance

$\delta(u,v)$:= shortest distance from u to v
∞ if v is not reachable from u

Variants include weighted/unweighted, single-source/all-pairs

Algorithms will construct vector/matrix d; we want d = δ

Back pointers π can be computed to reconstruct path

Breadth-first search

Input:

Graph G= (V,E) as adjacency list, and $s \in V$.

Output:

Distance from s to any other vertex

- Discover vertices at distance k before those at distance k+1

Algorithm colors each vertex:

White : not discovered.
Gray : discovered but its neighbors may not be.
Black : discovered and all of its neighbors are too.

```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
  u := Dequeue(Q)        // a vertex with min distance d[u];
  for each v ∈ adj[u]    // checks neighbors
    if color[v] = white {
     color[v] := gray;
     d[v]:= d[u]+1;
     π[v]:=u;
     Enqueue(Q,v)
   }
   color[u]:=Black;
 }
```

BFS(G,s)

For each vertex u ∈ V[G] – {s}

color[u]:= White; d[u] := ∞; π[u] :=NIL;

Q:= empty Queue; color[s] := Gray; d[s] :=0; π[s] :=NIL;

Enqueue(Q,s)

While (|Q| > 0) {

u := Dequeue(Q)

for each v ∈ adj[u]

if color[v] = white {

color[v] := gray;

d[v]:= d[u]+1;

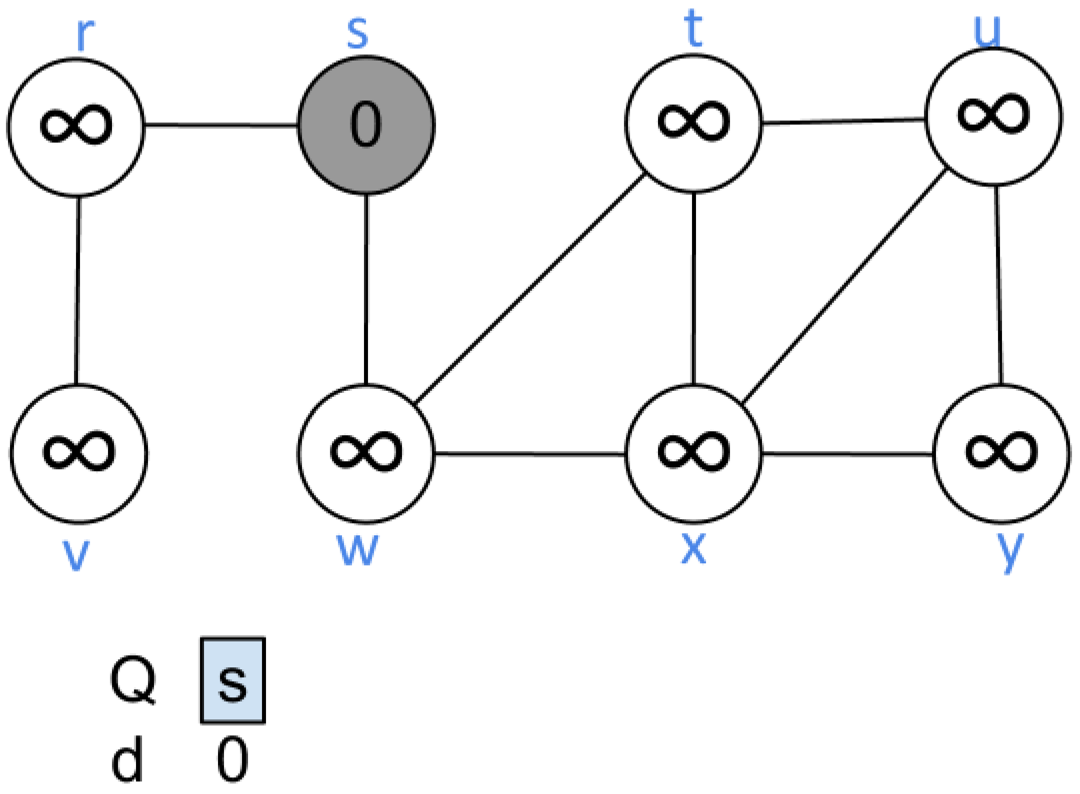
π[v]:=u;

Enqueue(Q,v)

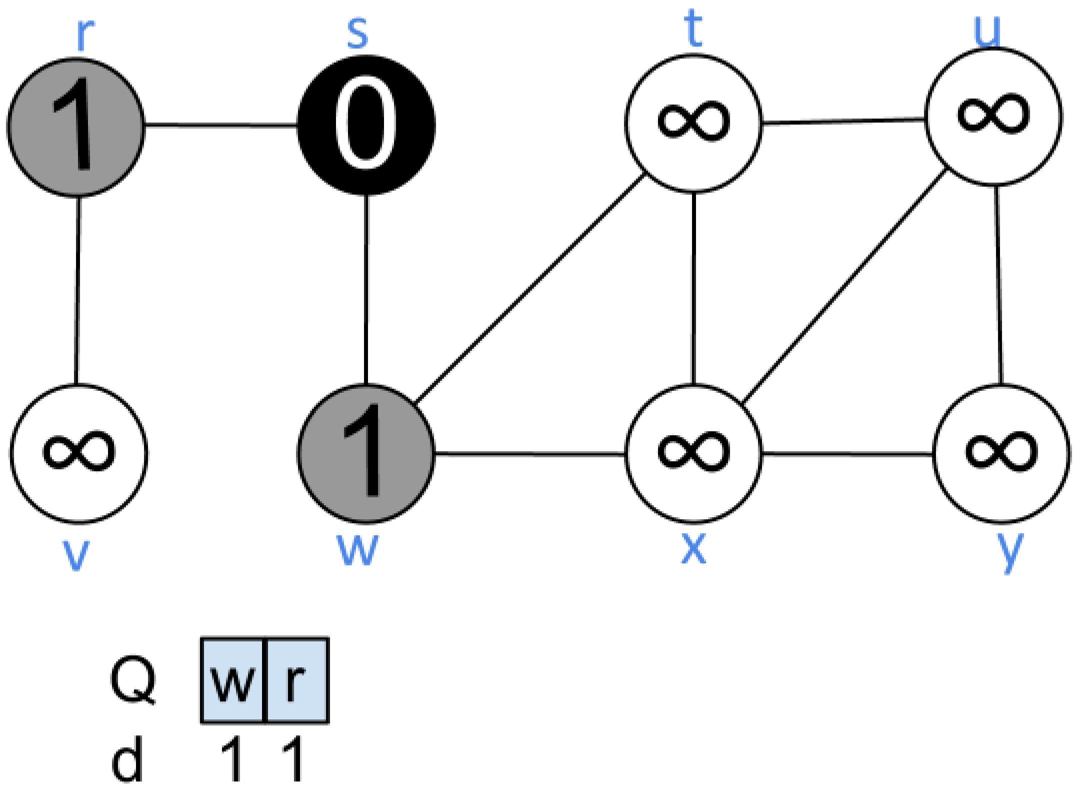
}

color[u]:=Black;

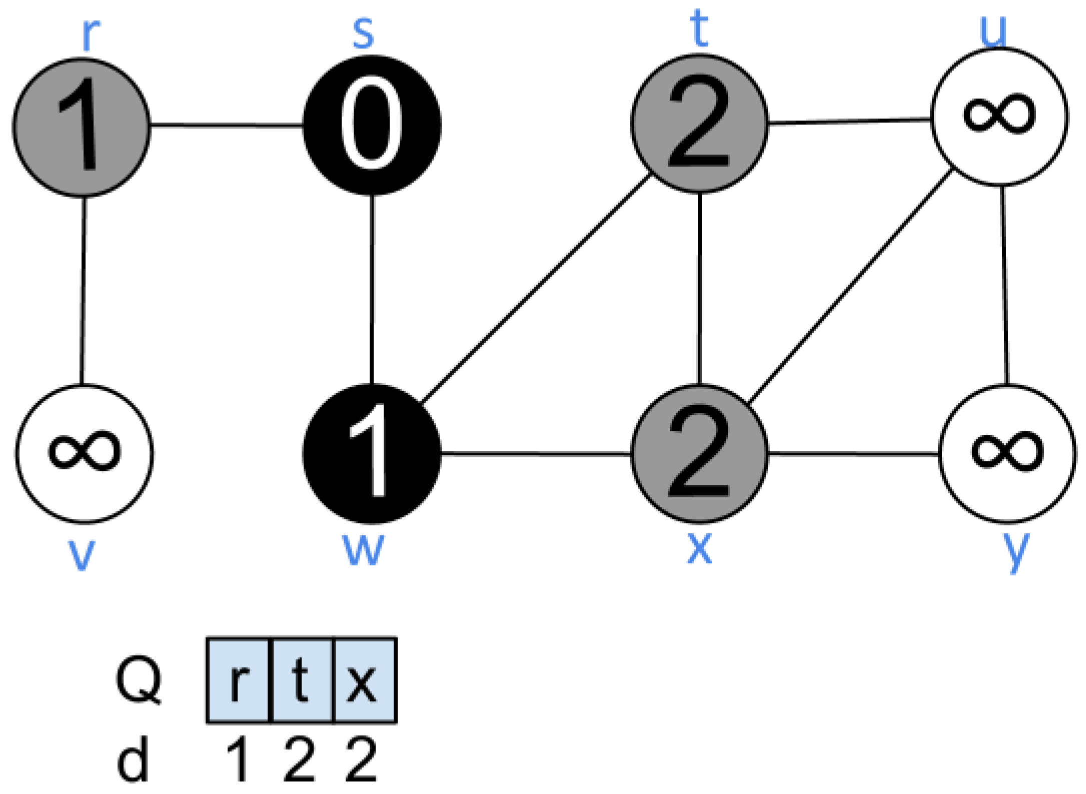
}



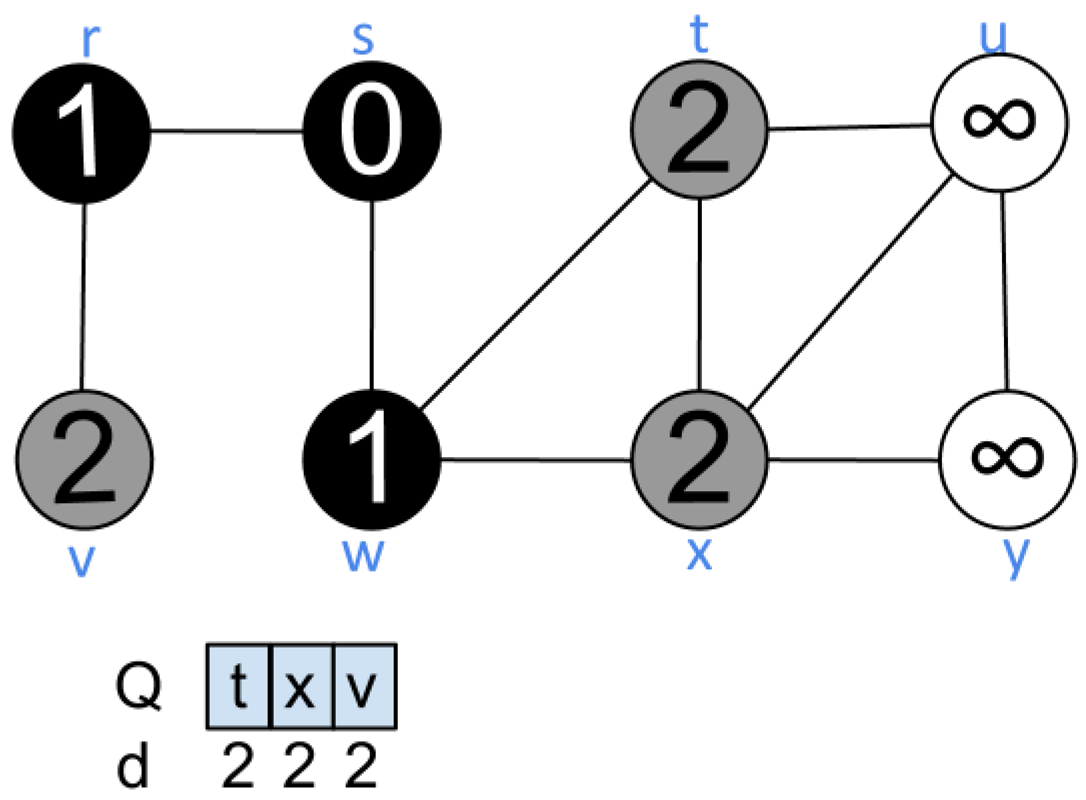
```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
   u := Dequeue(Q)
   for each v ∈ adj[u]
     if color[v] = white {
       color[v] := gray;
       d[v]:= d[u]+1;
       π[v]:=u;
       Enqueue(Q,v)
    }
   color[u]:=Black;
 }
```



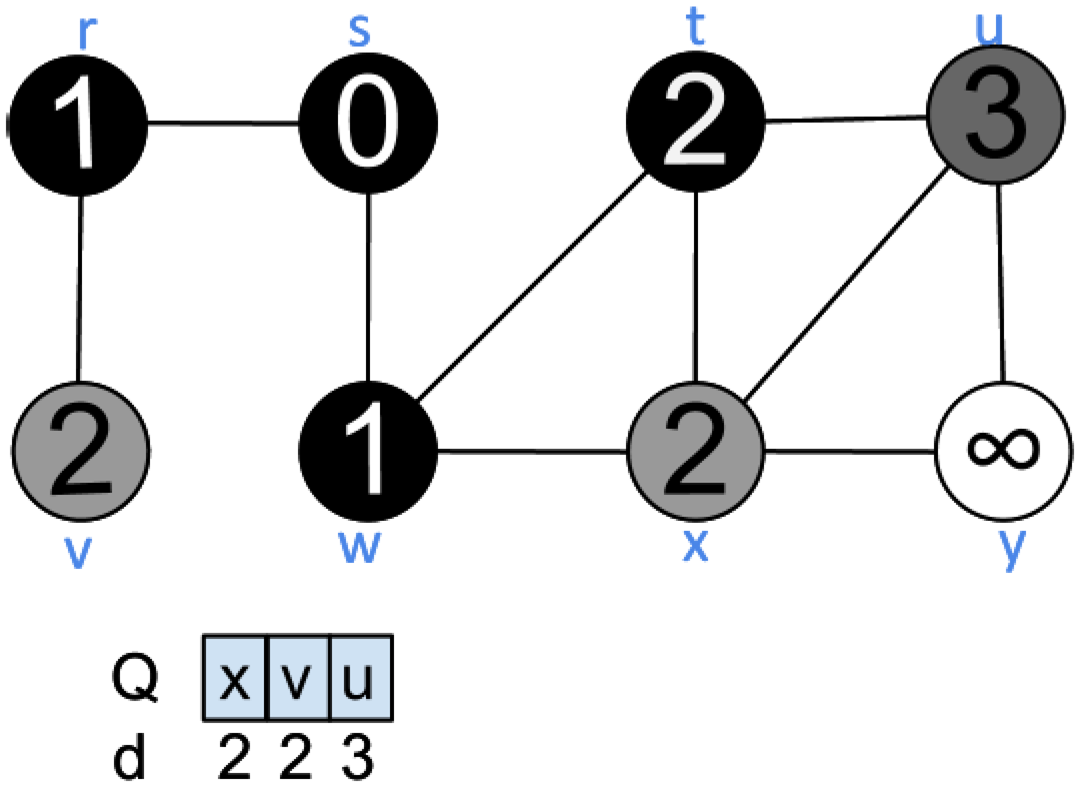
```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
   u := Dequeue(Q)
   for each v ∈ adj[u]
     if color[v] = white {
       color[v] := gray;
       d[v]:= d[u]+1;
       π[v]:=u;
       Enqueue(Q,v)
   }
   color[u]:=Black;
 }
```



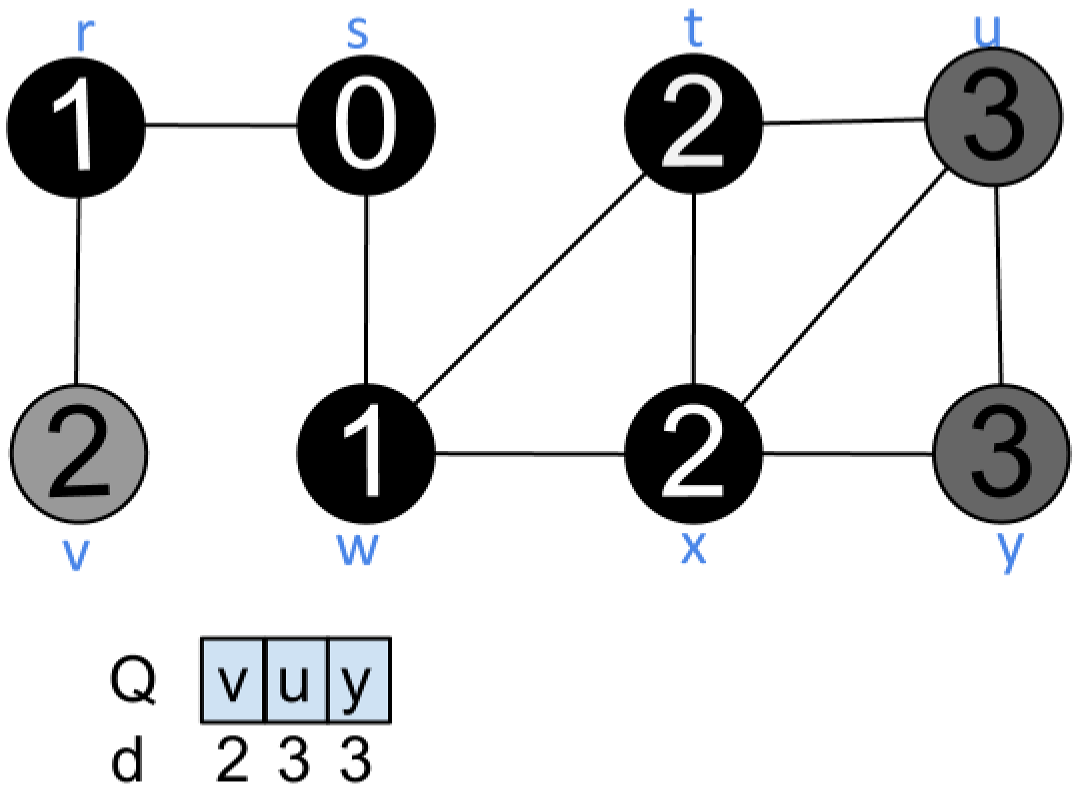
```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
   u := Dequeue(Q)
   for each v ∈ adj[u]
     if color[v] = white {
       color[v] := gray;
       d[v]:= d[u]+1;
       π[v]:=u;
       Enqueue(Q,v)
     }
   color[u]:=Black;
 }
```



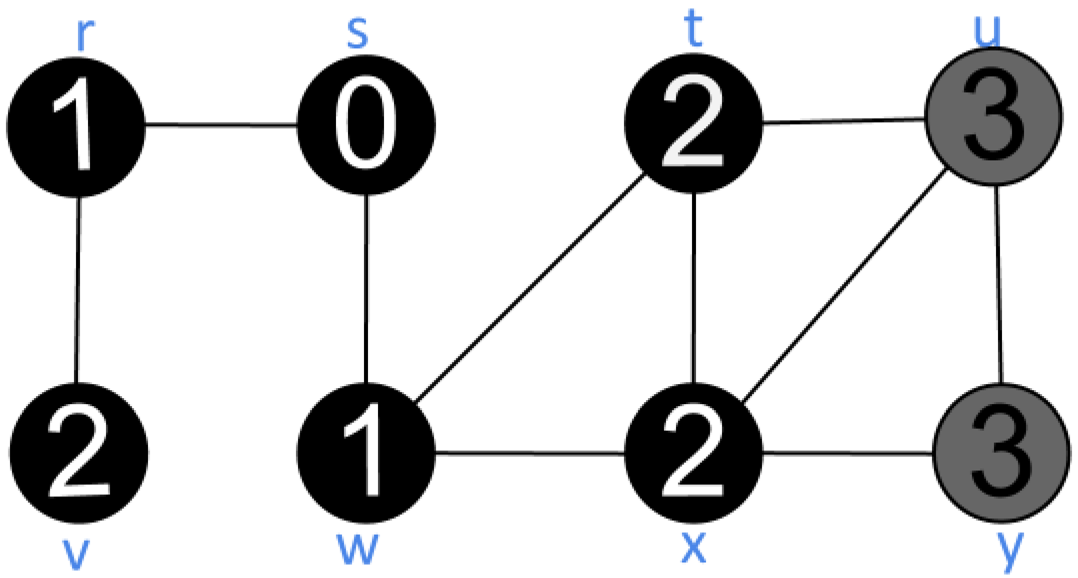
```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
   u := Dequeue(Q)
   for each v ∈ adj[u]
     if color[v] = white {
       color[v] := gray;
       d[v]:= d[u]+1;
       π[v]:=u;
       Enqueue(Q,v)
     }
   color[u]:=Black;
 }
```



```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
   u := Dequeue(Q)
   for each v ∈ adj[u]
     if color[v] = white {
       color[v] := gray;
       d[v]:= d[u]+1;
       π[v]:=u;
       Enqueue(Q,v)
   }
   color[u]:=Black;
 }
```

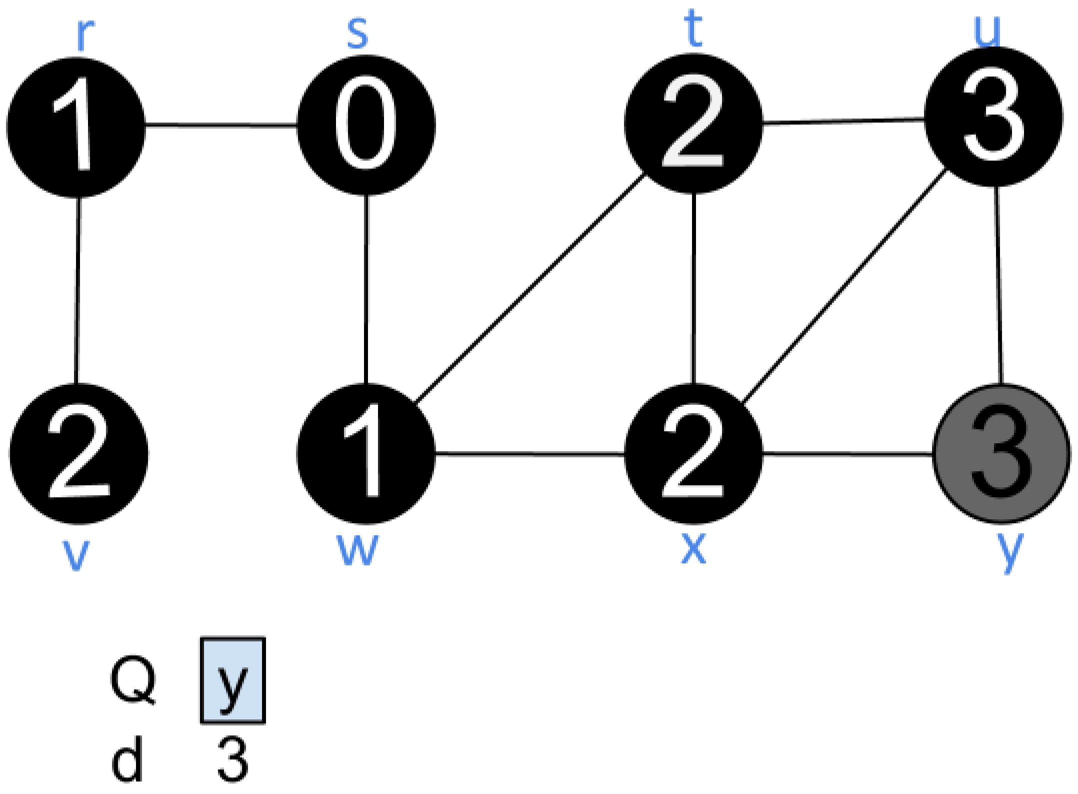


```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
   u := Dequeue(Q)
   for each v ∈ adj[u]
     if color[v] = white {
       color[v] := gray;
       d[v]:= d[u]+1;
       π[v]:=u;
       Enqueue(Q,v)
     }
   color[u]:=Black;
 }
```

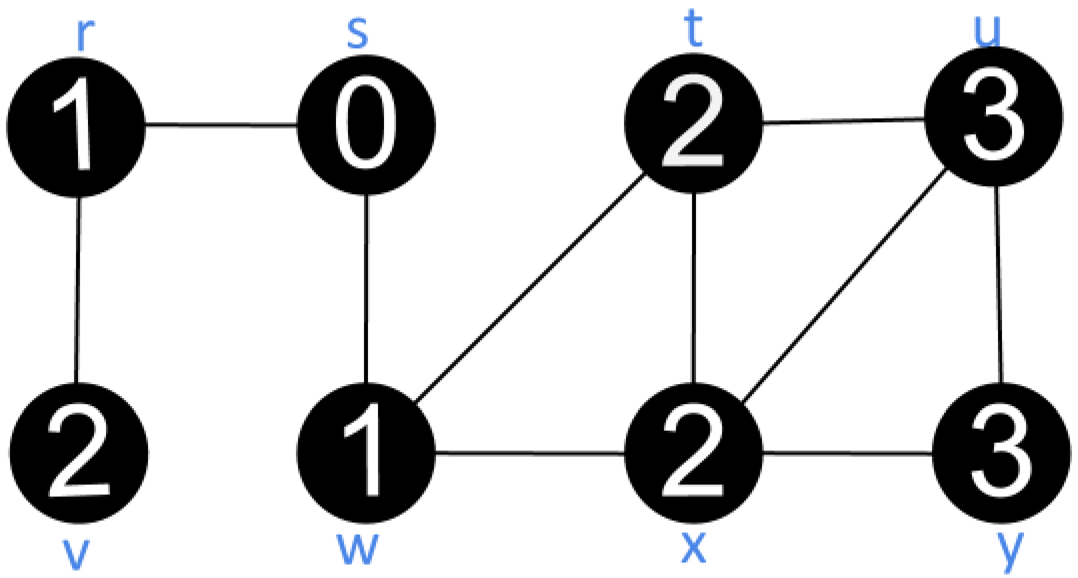


Q	u	y
d	3	3

```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
   u := Dequeue(Q)
   for each v ∈ adj[u]
     if color[v] = white {
       color[v] := gray;
       d[v]:= d[u]+1;
       π[v]:=u;
       Enqueue(Q,v)
     }
   color[u]:=Black;
 }
```



```
BFS(G,s)
 For each vertex u ∈ V[G] – {s}
   color[u]:= White; d[u] := ∞; π[u] :=NIL;

 Q:= empty Queue; color[s] := Gray;  d[s] :=0; π[s] :=NIL;

 Enqueue(Q,s)
 While (|Q| > 0) {
  u := Dequeue(Q)
  for each v ∈ adj[u]
    if color[v] = white {
     color[v] := gray;
     d[v]:= d[u]+1;
     π[v]:=u;
     Enqueue(Q,v)
   }
   color[u]:=Black;
 }
```



Q
d

Running time of BFS in adjacency-list representation

Recall Enqueue and Dequeue take time ?

Running time of BFS in adjacency-list representation

Recall Enqueue and Dequeue take time $O(1)$

Each edge visited $O(1)$ times.

Main loop costs $O(E)$.

Initialization step costs $O(V)$

Running time $O(V + E)$

What about space?

Space of BFS

$\Theta(V)$ to mark nodes

Optimal to compute all of d

What if we just want to know if u and v are connected?

Theorem: Given a graph with n nodes, can decide if two nodes are connected in space $O(\log^2 n)$

Proof:

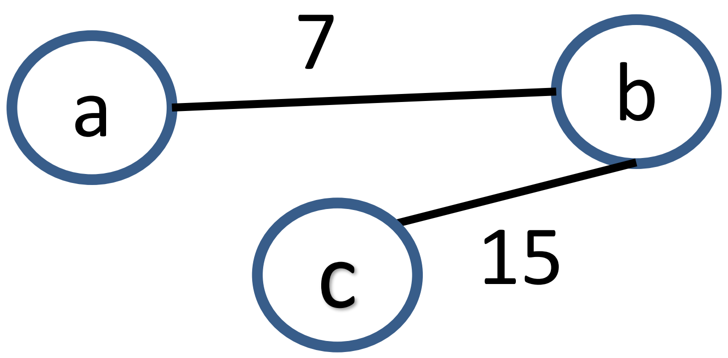
```
REACH(u, v, n) :=       \\ is v reachable from u in n steps?
  Enumerate all nodes w {
      If REACH(u, w, n/2) and REACH(w, v, n/2) return YES
  }
  Return NO
```

$S(n)$:= space for REACH(u, v, n).

$S(n) := O(\log n) + S(n/2)$. Reuse space for 2 calls to REACH.

$S(n) = O(\log^2 n)$ □

Next: weighted single-source shortest path



Input: Directed graph G= (V,E), s $\in$ V, w: E $\rightarrow$ Z

Output: Shortest paths from s to all the other vertces

- Note: Previous case was for w : E $\rightarrow$ {1}

- Note: if weights can be negative, shortest paths exist $\Leftrightarrow$ s cannot reach a cycle with negative weight

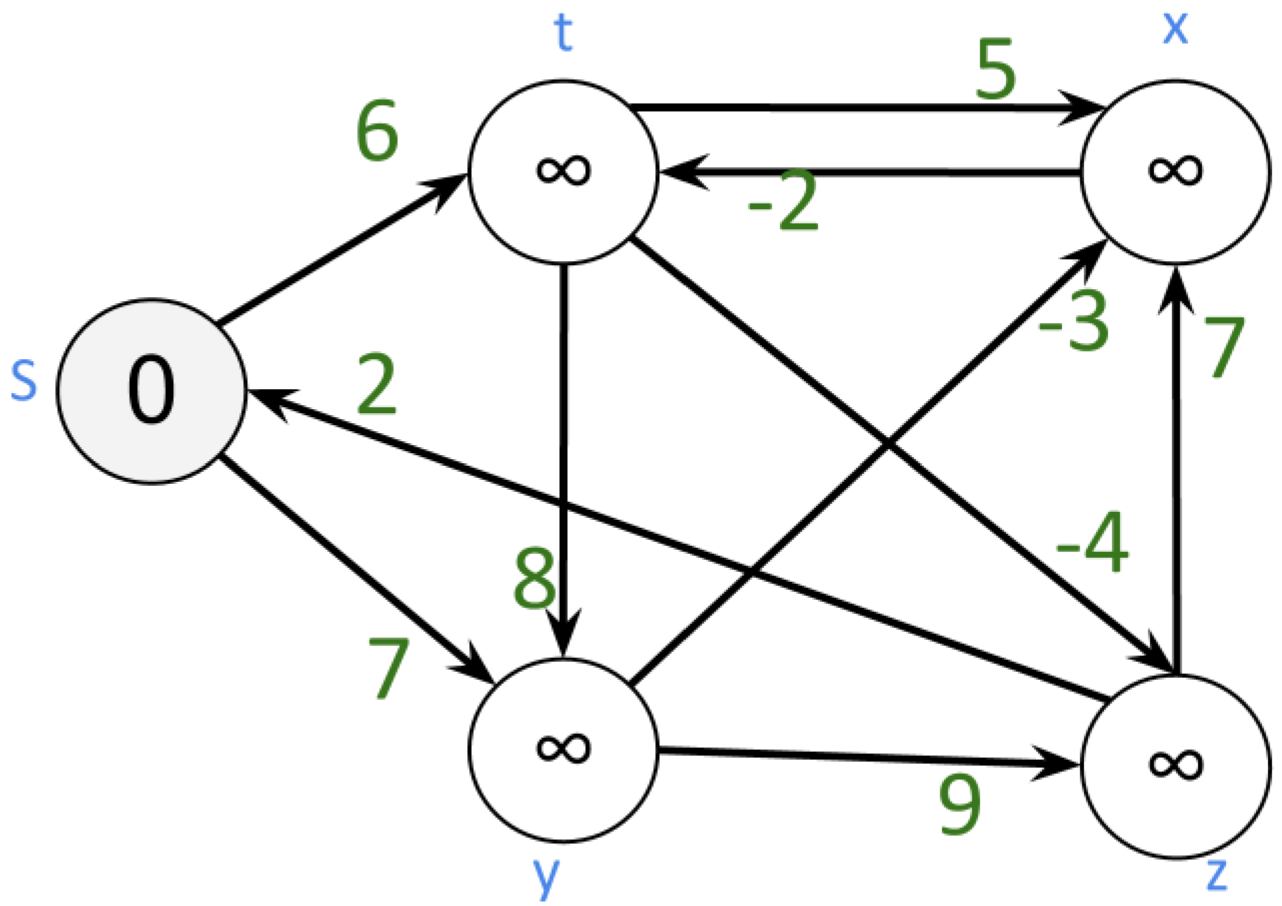
```
Bellman-Ford(G,w, s)
   d[s] :=0; Set the others to ∞

  Repeat |V| stages:
    for each edge (u,v) ∈ E[G]
        d[v] := min{ d[v], d[u]+w(u,v); }  //relax(u,v)
```

At the end of the algorithm, can detect negative cycles by:

```
for each edge (u,v) ∈ E[G]
   if d[v] > d[u]+w(u,v)
       Return Negative cycle

return No negative cycle
```

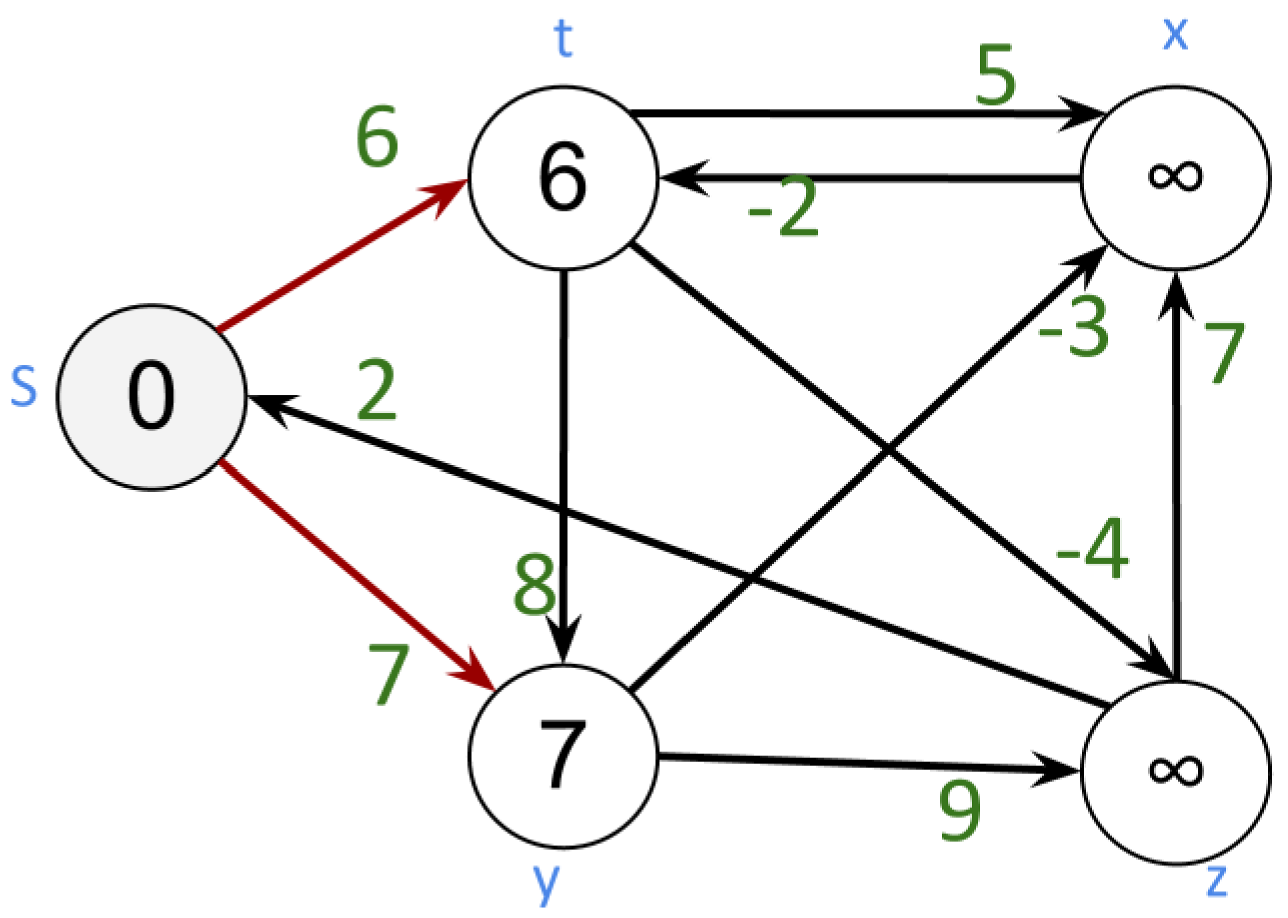
Bellman-Ford(G,w, s)
  d[s] :=0; Set the others to ∞

 Repeat |V| stages:
   for each edge $(u,v) \in E[G]$
     $d[v] := \min\{ d[v], d[u]+w(u,v); \}$ //relax(u,v)



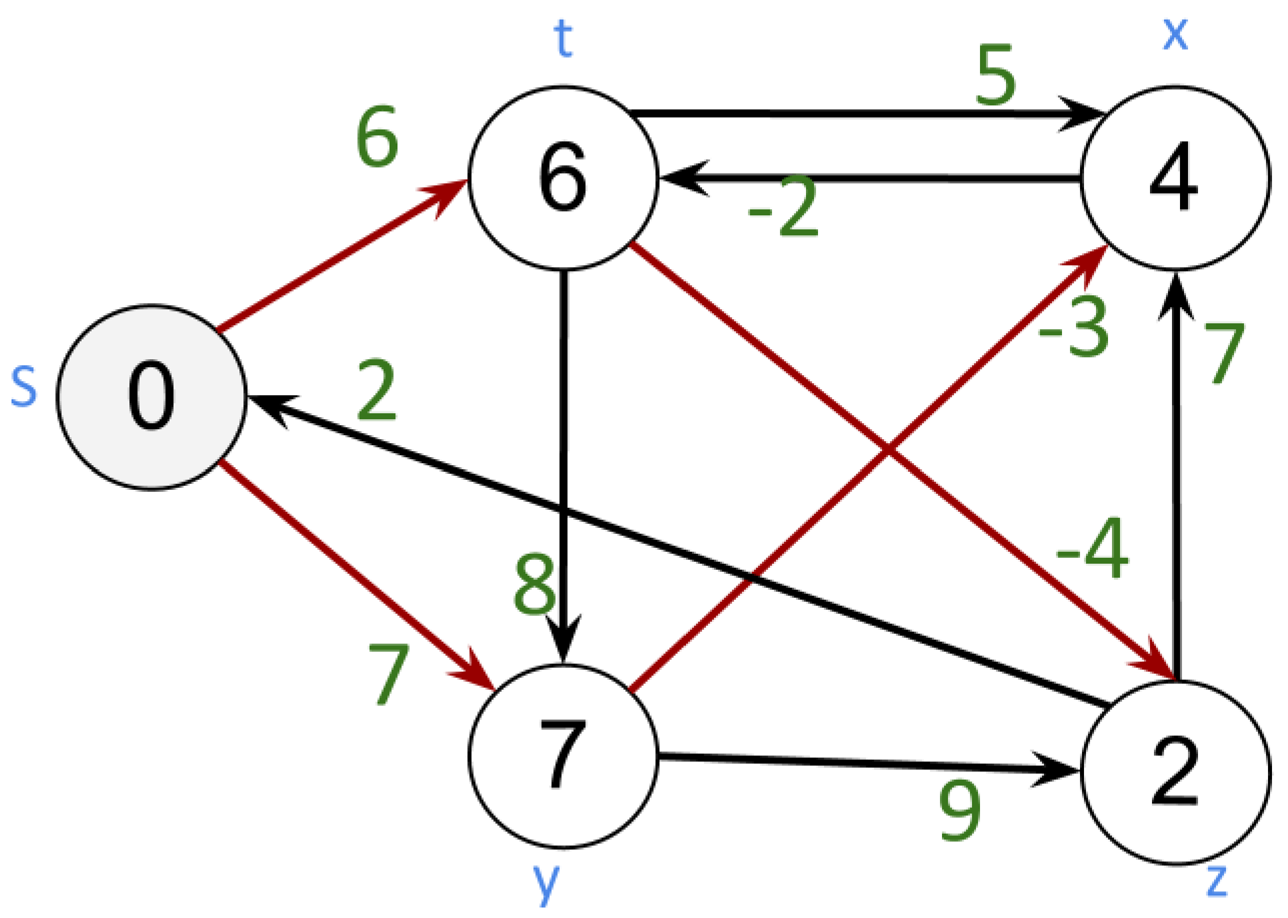
Bellman-Ford(G,w, s)
    d[s] :=0; Set the others to ∞

  Repeat |V| stages:
    for each edge (u,v) ∈ E[G]
        d[v] := min{ d[v], d[u]+w(u,v); } //relax(u,v)



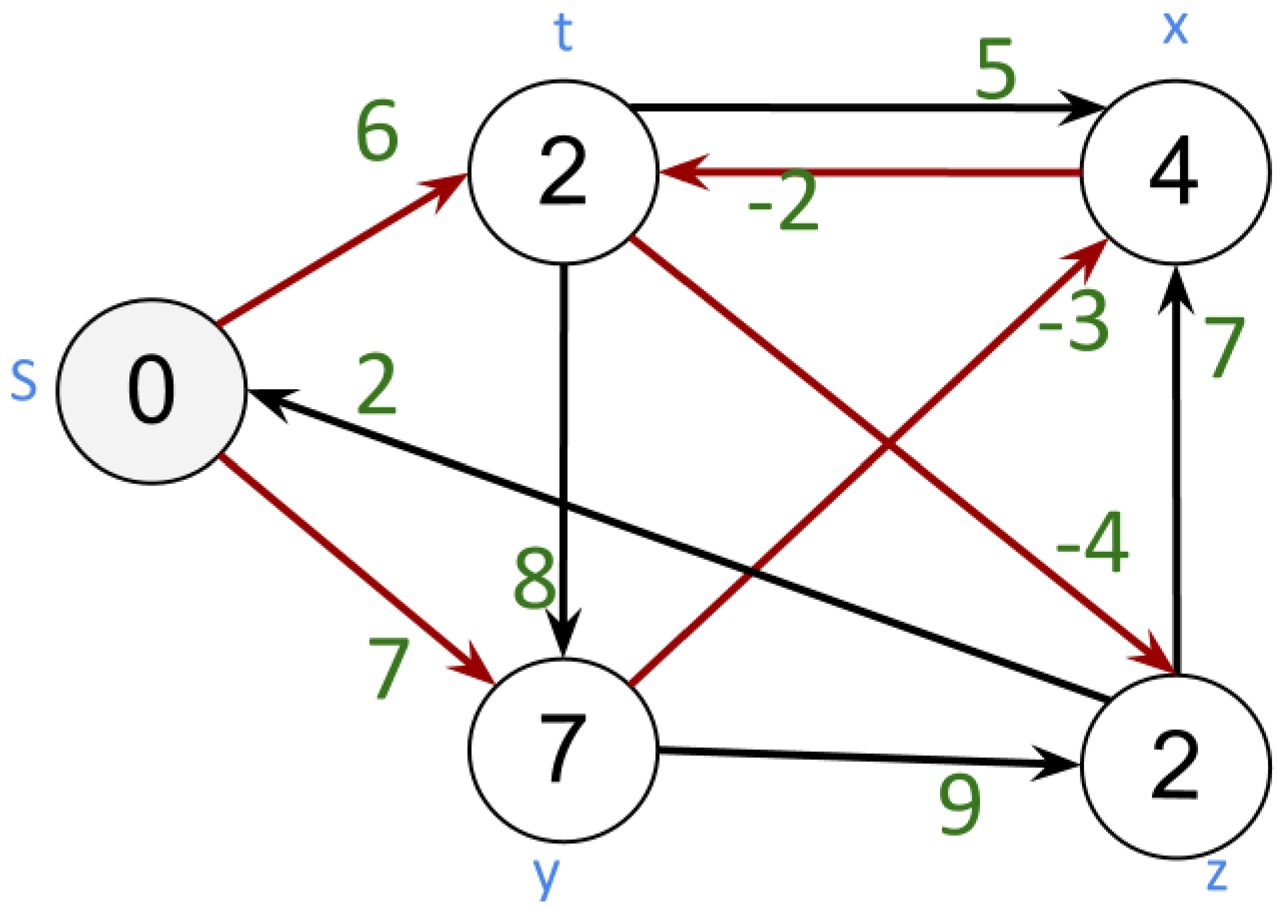
Bellman-Ford(G,w, s)
  d[s] :=0; Set the others to ∞

Repeat |V| stages:
  for each edge (u,v) ∈ E[G]
    d[v] := min{ d[v], d[u]+w(u,v); } //relax(u,v)



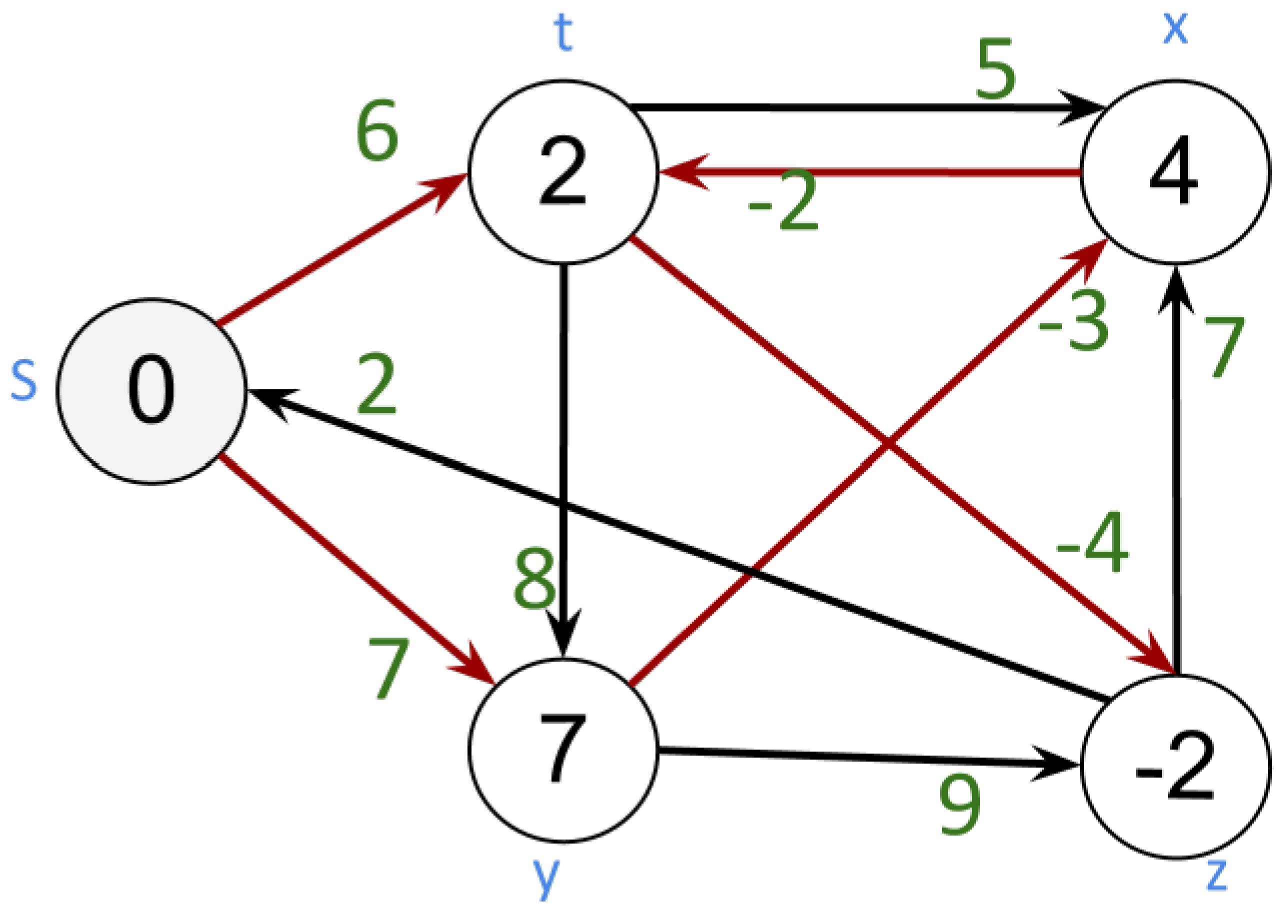
Bellman-Ford(G,w, s)
  d[s] :=0; Set the others to ∞

 Repeat |V| stages:
  for each edge (u,v) ∈ E[G]
   d[v] := min{ d[v], d[u]+w(u,v); } //relax(u,v)



Bellman-Ford(G,w, s)
d[s] :=0; Set the others to ∞

Repeat |V| stages:
for each edge (u,v) ∈ E[G]
d[v] := min{ d[v], d[u]+w(u,v); } //relax(u,v)



Running time of Bellman-Ford

```
Bellman-Ford(G,w, s)
    d[s] :=0; Set the others to ∞

  Repeat |V| stages:
     for each edge (u,v) ∈ E[G]
         d[v] := min{ d[v], d[u]+w(u,v); }  //relax(u,v)
```

Time = ??

Running time of Bellman-Ford

Bellman-Ford(G,w, s)
    d[s] :=0; Set the others to ∞

  Repeat |V| stages:
    for each edge $(u,v) \in E[G]$
        d[v] := min{ d[v], d[u]+w(u,v); } //relax(u,v)

Time = O(|V|.|E|)

Analysis of Bellman-Ford(G,w, s)
  d[s] :=0; Set the others to ∞

Repeat |V| stages:
  for each edge $(u,v) \in E[G]$
    d[v] := min{ d[v], d[u]+w(u,v); } //relax(u,v)

- Claim: $d = \delta$ if no negative-weight cycle exists.
- Proof: Consider a shortest path $s \to u_1 \to u_2 \to \ldots \to u_k$

  $k \leq n$ by assumtion.

We claim at stage $i = 1..|V|$, $d[u_i] = \delta(s, u_i)$

This holds by induction, because:

$d[u_i] = \delta(s, u_i)$ and relax $u_i \to u_{i+1}$ $\Leftrightarrow$ $d[u_{i+1}] = \delta(s, u_{i+1})$.

d is never increased

d is never set below δ □

Fact: Consider an algorithm that starts with
$d[s] = 0$ and ∞ otherwise, and only does edge relaxations.

Then $d \geq \delta$ throughout

Analysis of negative-cycle detection at the end of algorithm:

```
for each edge (u,v) ∈ E[G]
   if d[v] > d[u]+w(u,v)
      Return Negative cycle

return No negative cycle
```

- Proof of correctness:

If not ∃ neg-cycle, d = δ, tests pass (triangle inequality).

O.w. let $u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_k = u_1$ so that $\sum_{i<k} w(u_i, u_{i+1}) < 0$

We know $\forall\ i<k : d[u_{i+1}] \leq d[u_i] + w(u_i, u_{i+1})$

now what?

Analysis of negative-cycle detection at the end of algorithm:

```
for each edge (u,v) ∈ E[G]
  if d[v] > d[u]+w(u,v)
     Return Negative cycle

return No negative cycle
```

- Proof of correctness:

If not ∃ neg-cycle, $d = \delta$, tests pass (triangle inequality).

O.w. let $u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_k = u_1$ so that $\sum_{i < k} w(u_i , u_{i+1}) < 0$

We know $\forall\ i<k : d[u_{i+1}] \leq d[u_i] + w(u_i , u_{i+1})$

$\sum_{i<k} d[u_{i+1}] \leq \sum_{i<k} d[u_i] + \sum_{i<k} w(u_i , u_{i+1}) \Rightarrow 0 < 0$ □

Dijkstra's algorithm

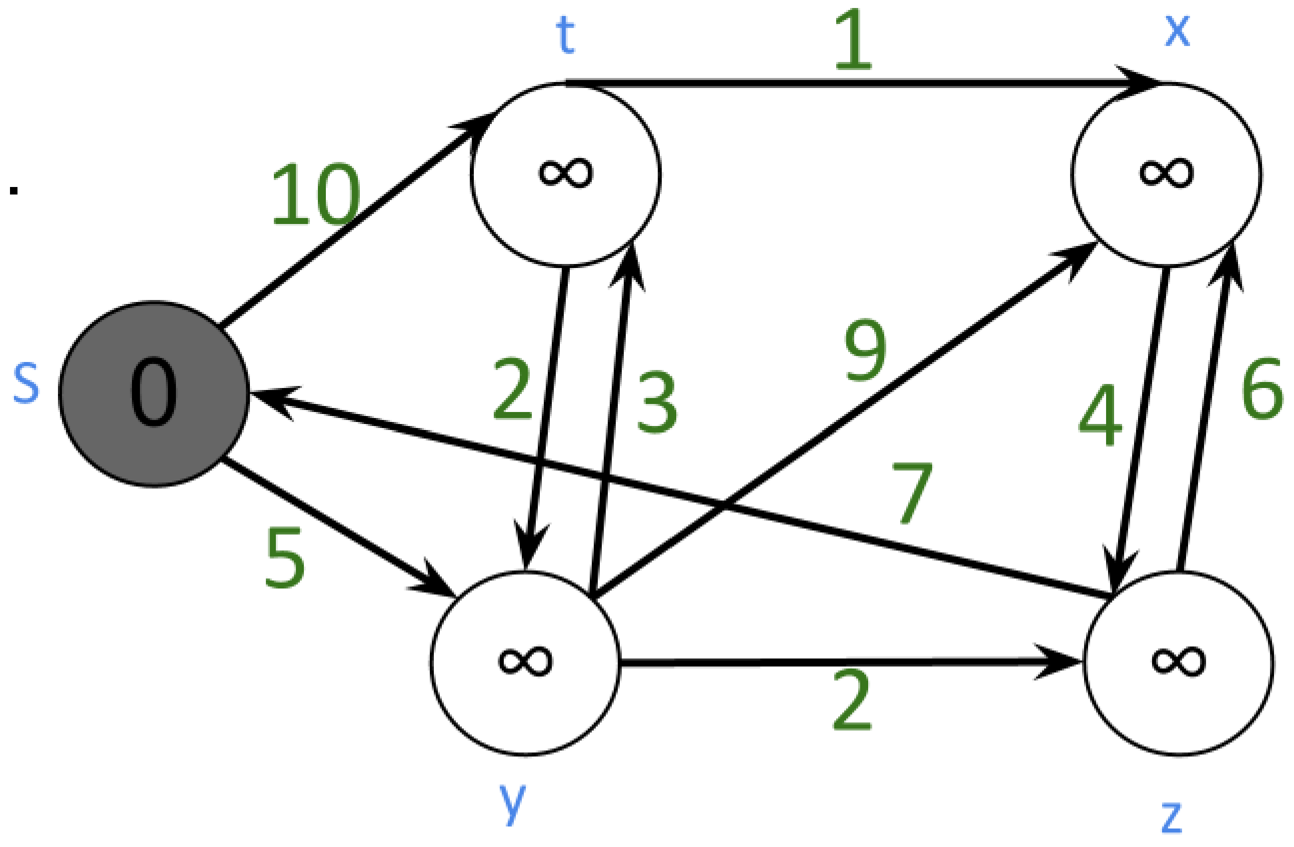
Input: Directed graph $G=(V,E)$, $s \in V$, non-negative $w: E \to N$

Output: Shortest paths from s to all the other vertices.

```
Dijkstra(G,w,s)
  d[s] :=0; Set others d to ∞; Q := V

  While (|Q|> 0) {
    u:= extract-remove-min(Q) // vertex with min distance d[u];
     for each v ∈ adj[u]
          d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
   }
```

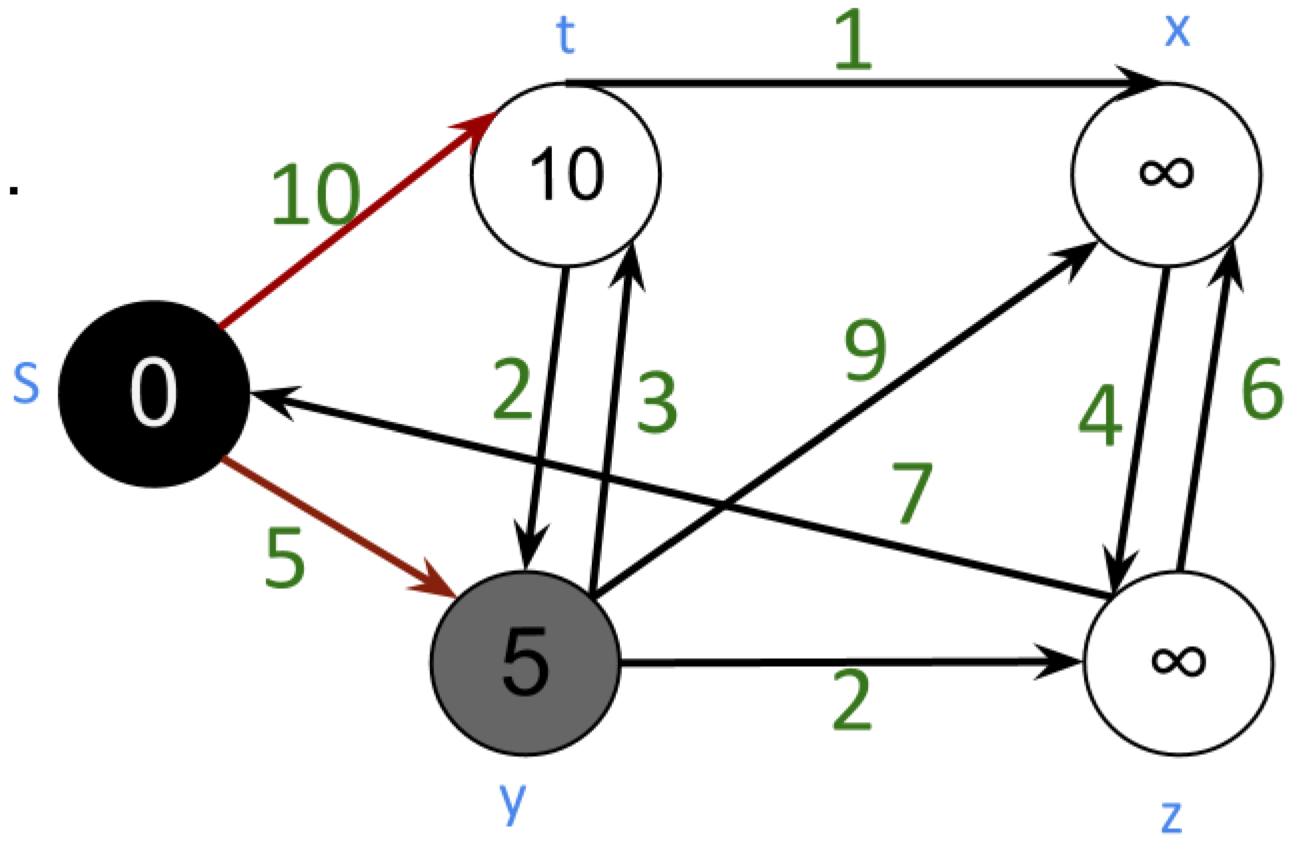
```
Dijkstra(G,w,s)
  d[s] :=0; Set others d to ∞; Q := V

  While (|Q|> 0) {
    u:= extract-remove-min(Q) // vertex with min distance d[u];
    for each v ∈ adj[u]
        d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
  }
```

Gray: the extracted u.
Black: not in Q



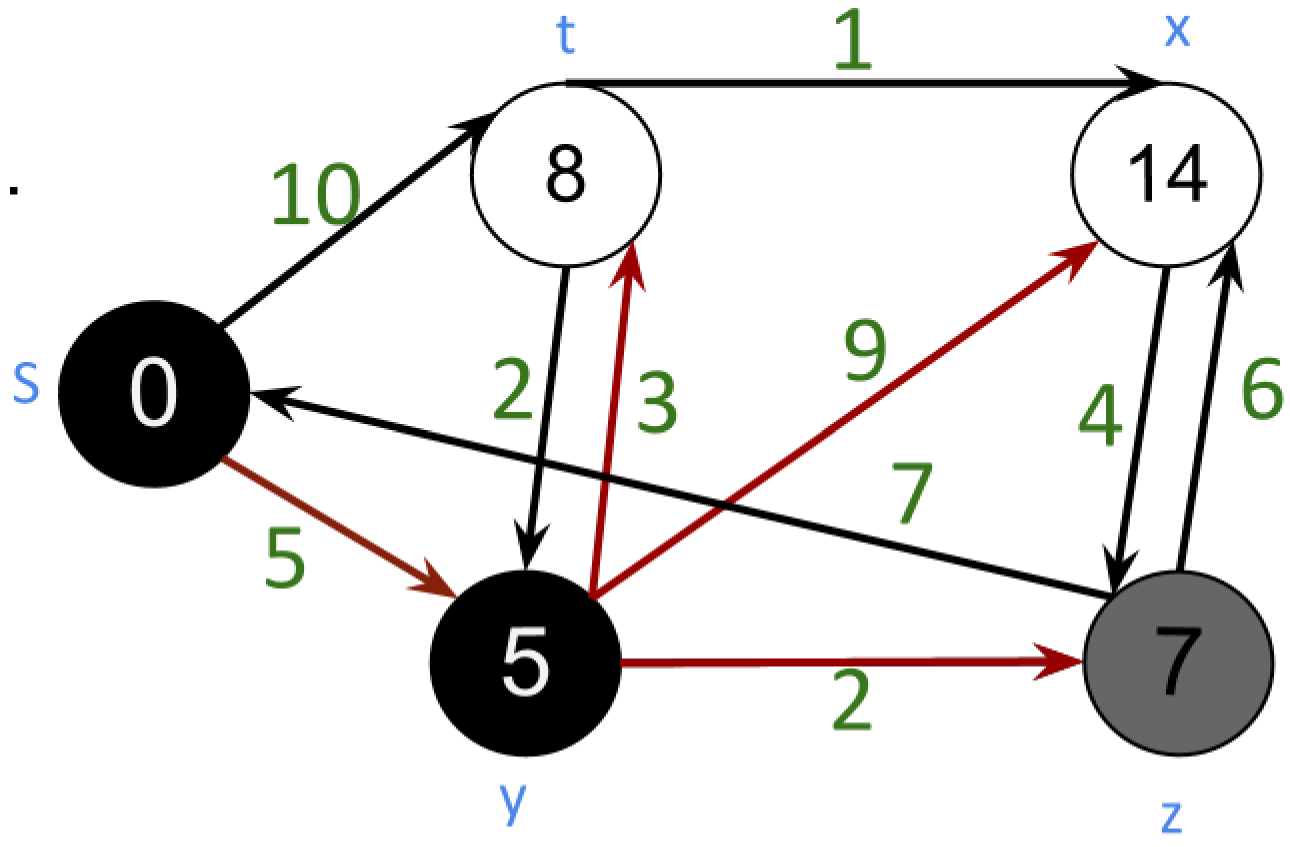
```
Dijkstra(G,w,s)
  d[s] :=0; Set others d to ∞; Q := V

  While (|Q|> 0) {
    u:= extract-remove-min(Q) // vertex with min distance d[u];
     for each v ∈ adj[u]
         d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
   }
```

Gray: the extracted u.
Black: not in Q



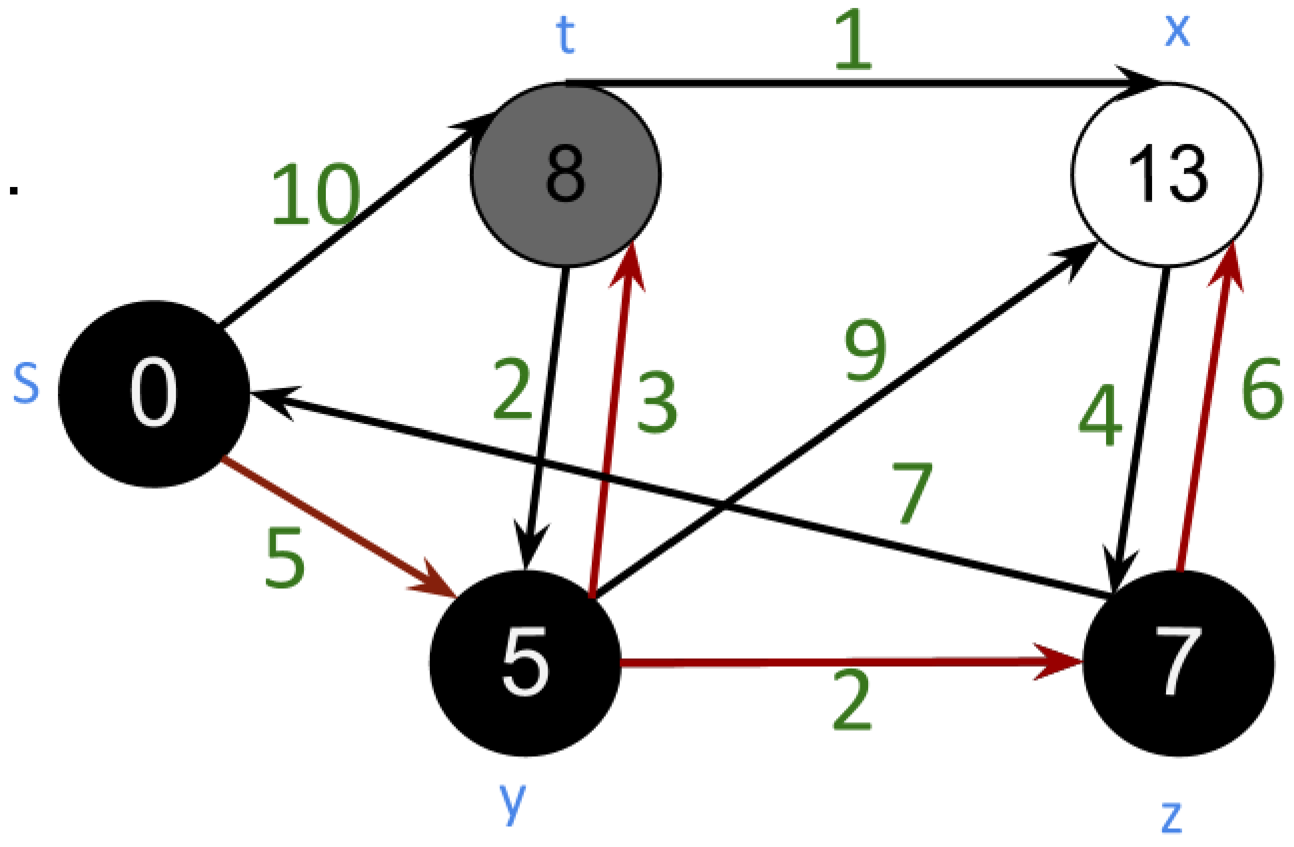
```
Dijkstra(G,w,s)
  d[s] :=0; Set others d to ∞; Q := V

  While (|Q|> 0) {
    u:= extract-remove-min(Q) // vertex with min distance d[u];
     for each v ∈ adj[u]
         d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
  }
```

Gray: the extracted u.
Black: not in Q



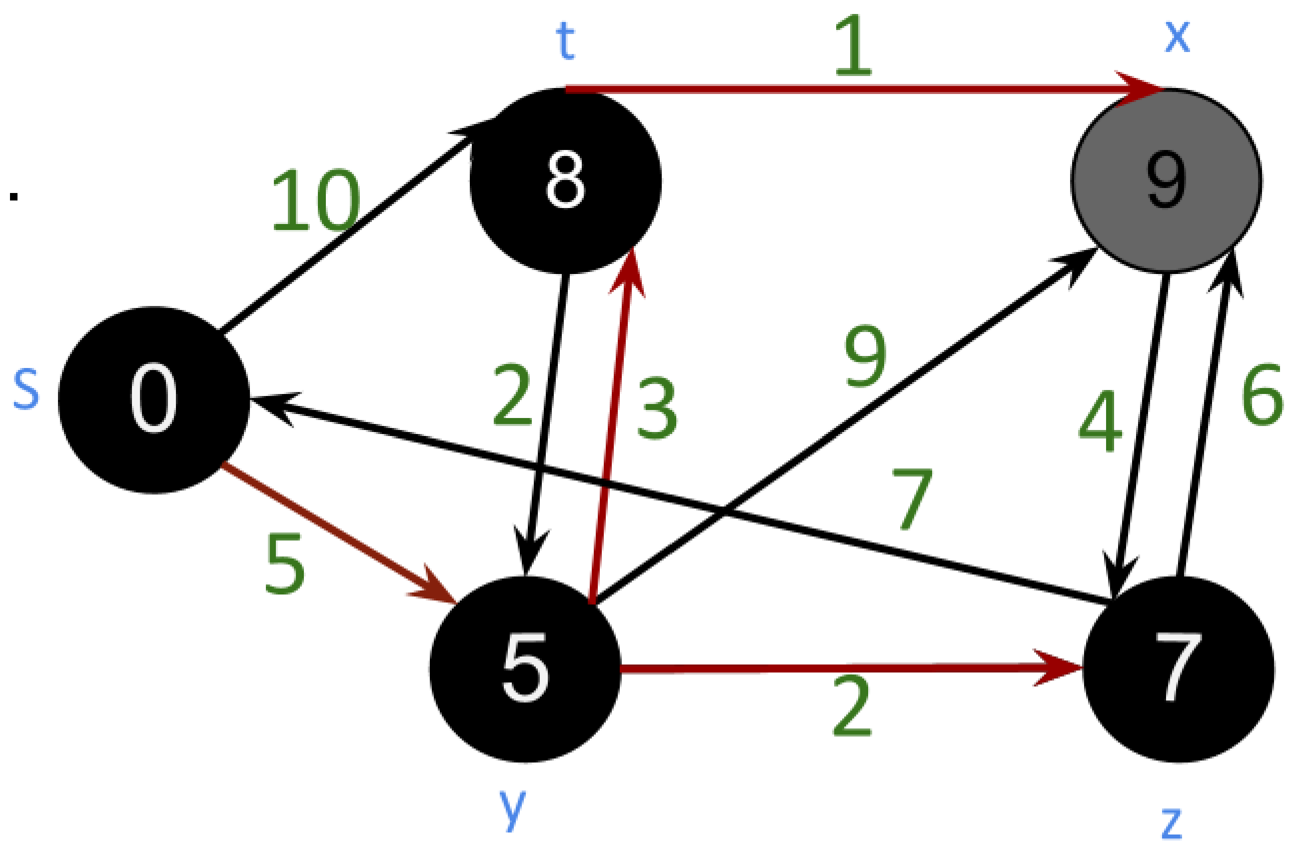
```
Dijkstra(G,w,s)
  d[s] :=0; Set others d to ∞; Q := V

  While (|Q|> 0) {
    u:= extract-remove-min(Q) // vertex with min distance d[u];
     for each v ∈ adj[u]
         d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
  }
```

Gray: the extracted u.
Black: not in Q



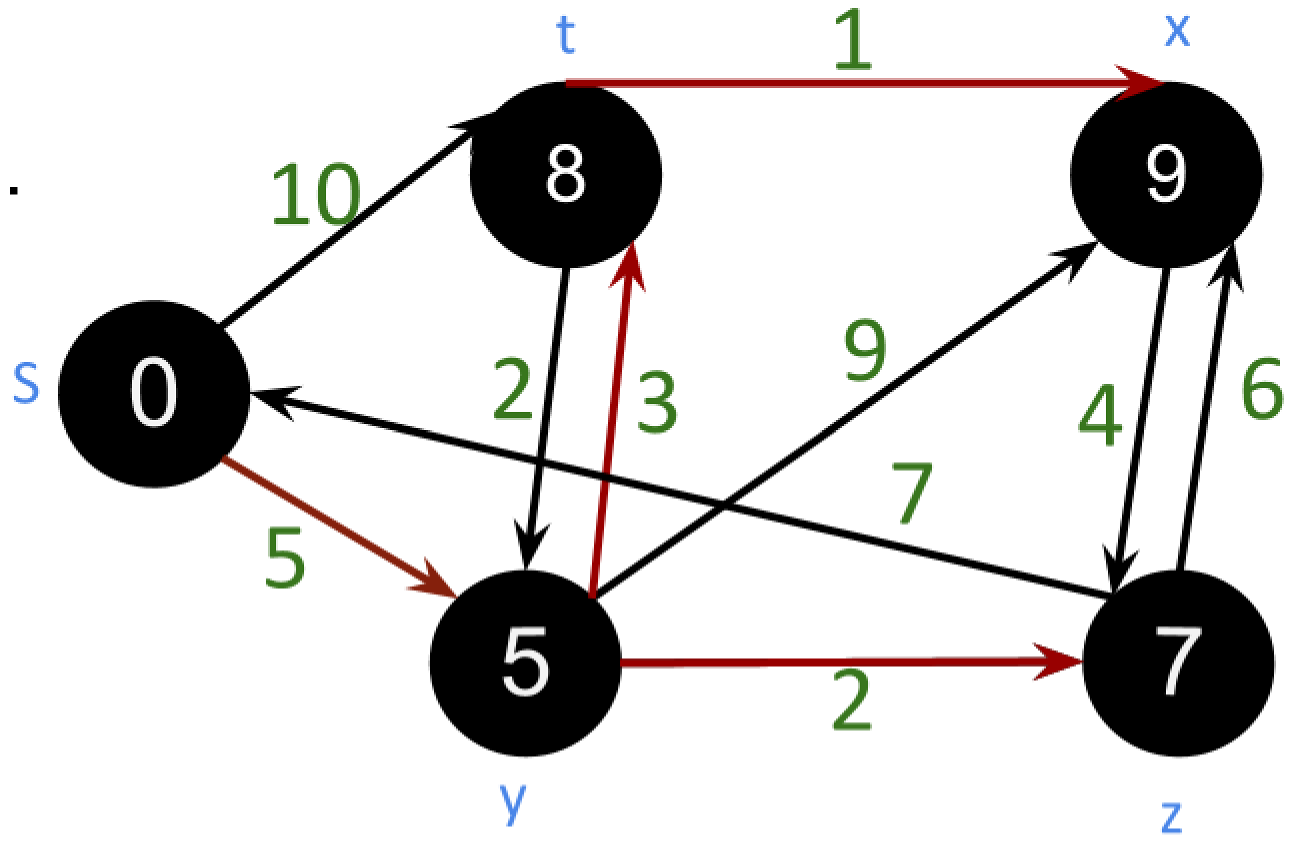
```
Dijkstra(G,w,s)
  d[s] :=0; Set others d to ∞; Q := V

  While (|Q|> 0) {
    u:= extract-remove-min(Q) // vertex with min distance d[u];
     for each v ∈ adj[u]
         d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
  }
```

Gray: the extracted u.
Black: not in Q



```
Dijkstra(G,w,s)
  d[s] :=0; Set others d to ∞; Q := V

  While (|Q|> 0) {
    u:= extract-remove-min(Q) // vertex with min distance d[u];
     for each v ∈ adj[u]
         d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
  }
```

Gray: the extracted u.
Black: not in Q



Running time of Dijkstra(G,w,s)

d[s] :=0; Set others d to ∞; Q := V

```
While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
  for each v ∈ adj[u]
      d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
 }
```

Running time depends on data structure for Q

Naive implementation, array: Extract-min in time ?

Running time of Dijkstra(G,w,s)

```
d[s] :=0; Set others d to ∞; Q := V

While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
  for each v ∈ adj[u]
      d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Running time depends on data structure for Q

Naive implementation, array: Extract-min in time $|V|$

$\Rightarrow$ running time = ?

Running time of Dijkstra(G,w,s)

```
d[s] :=0; Set others d to ∞; Q := V

While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
  for each v ∈ adj[u]
      d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Running time depends on data structure for Q

Naive implementation, array: Extract-min in time |V|

$\Rightarrow$ running time = $O(V^2 + E)$

Can we do better?

Running time of Dijkstra(G,w,s)

```
d[s] :=0; Set others d to ∞; Q := V

While (|Q|> 0) {
   u:= extract-remove-min(Q) // vertex with min distance d[u];
   for each v ∈ adj[u]
        d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Running time depends on data structure for Q

Naive implementation, array: Extract-min in time |V|

$\Rightarrow$ running time = $O(V^2 + E)$

Implement Q with min-heap. Extract-min in time ?

Running time of Dijkstra(G,w,s)

```
d[s] :=0; Set others d to ∞; Q := V

While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
  for each v ∈ adj[u]
      d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Running time depends on data structure for Q

Naive implementation, array: Extract-min in time |V|

$\Rightarrow$ running time = $O(V^2 + E)$

Implement Q with min-heap. Extract-min in time O(log V)

$\Rightarrow$ running time = ?

Running time of Dijkstra(G,w,s)

```
d[s] :=0; Set others d to ∞; Q := V

While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
   for each v ∈ adj[u]
        d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
 }
```

Running time depends on data structure for Q

Naive implementation, array: Extract-min in time |V|

$\Rightarrow$ running time = $O(V^2 + E)$

Implement Q with min-heap. Extract-min in time O(log V)
$\Rightarrow$ running time = O((V+E) log V)
Note: Can be improved to V log V + E

Analysis of Dijkstra

```
d[s] :=0; Set others d to ∞; Q := V
While (|Q|> 0) {
   u:= extract-remove-min(Q) // vertex with min distance d[u];
   for each v ∈ adj[u]
        d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Claim: When u is extracted, $d[u] = \delta(s,u)$

Analysis of Dijkstra

```
d[s] :=0; Set others d to ∞; Q := V
While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
   for each v ∈ adj[u]
       d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Claim: When u is extracted, $d[u] = \delta(s,u)$

Proof: Let $u \neq s$ be first violation, and Q right before extract

Let $s \to \ldots \to x \to y \to \ldots \to u$ be a shortest path,
where $s \notin Q$ and y is first $\in Q$

Note d[x] = ?

Analysis of Dijkstra

```
d[s] :=0; Set others d to ∞; Q := V
While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
   for each v ∈ adj[u]
       d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Claim: When u is extracted, $d[u] = \delta(s,u)$

Proof: Let $u \neq s$ be first violation, and Q right before extract

Let $s \rightarrow \ldots \rightarrow x \rightarrow y \rightarrow \ldots \rightarrow u$ be a shortest path,
where $s \notin Q$ and y is first $\in Q$

Note $d[x] = \delta(s,x)$ (since u is first violation)
$d[y] =$?

Analysis of Dijkstra

```
d[s] :=0; Set others d to ∞; Q := V
While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
   for each v ∈ adj[u]
        d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Claim: When u is extracted, $d[u] = \delta(s,u)$

Proof: Let $u \neq s$ be first violation, and Q right before extract

Let $s \to \ldots \to x \to y \to \ldots \to u$ be a shortest path,
where $s \notin Q$ and y is first $\in Q$

Note $d[x] = \delta(s,x)$ (since u is first violation)
$d[y] = \delta(s,y)$ (since $x \to y$ was relaxed)

Then d[u] ? d[y] How do they compare?

Analysis of Dijkstra

```
d[s] :=0; Set others d to ∞; Q := V
While (|Q|> 0) {
  u:= extract-remove-min(Q) // vertex with min distance d[u];
   for each v ∈ adj[u]
       d[v] := min{ d[v], d[u]+w(u,v)}    //relax(u,v)
}
```

Claim: When u is extracted, $d[u] = \delta(s,u)$

Proof: Let $u \neq s$ be first violation, and Q right before extract

Let $s \rightarrow \ldots \rightarrow x \rightarrow y \rightarrow \ldots \rightarrow u$ be a shortest path,
where $s \notin Q$ and y is first $\in Q$

Note $d[x] = \delta(s,x)$ (since u is first violation)
$d[y] = \delta(s,y)$ (since $x \rightarrow y$ was relaxed)

Then $d[u] \leq d[y]$ (because d[u] is minimum)
$= \delta(s,y) \leq \delta(s,u)$ □

All-pairs shortest paths

Input:

- Directed graph $G= (V,E)$, and $w: E \rightarrow R$

Output:

- The shortest paths between all pairs of vertices.

- Run Dijkstra $|V|$ times: $O(V^2 \log V + E\, V)$ if $w \geq 0$
- Run Bellman-Ford $|V|$ times: $O(V^2 E)$

- Next, simple algorithms achieving time about $|V|^3$ for any w

All-pairs shortest paths

Dynamic programming approach:
$d_{i,j}^{(m)}$ = shortest paths of lengths ≤ m

$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + w(k,j) \}$

(Includes k = j, w(j,j) = 0)

Compute |V| x |V| matrix $d^{(m)}$ from $d^{(m-1)}$ in time $|V|^3$.

$\Rightarrow d^{|V|}$ computables in time $|V|^4$

How to speed up?

All-pairs shortest paths

Note:

$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + w(k,j) \}$

Is just like matrix multiplication: $d^{(m)} = d^{(m-1)} W$,
except $+ \rightarrow \min$
$\quad\quad\quad x \rightarrow +$

Like matrix multiplication, this is associative. So, instead of doing $d^{|V|} = (...)W)W)W$ can do ?

All-pairs shortest paths

Note:

$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + w(k,j) \}$

Is just like matrix multiplication: $d^{(m)} = d^{(m-1)} W$,
except $+ \rightarrow \min$
$\qquad\quad x \rightarrow +$

Like matrix multiplication, this is associative. So,
instead of doing $d^{|V|} = (...)W)W)W$ can do repeated squaring:

Compute $d^{(2)} = W^2$
$\qquad\qquad d^{(4)} =$?

All-pairs shortest paths

Note:

$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + w(k,j) \}$

Is just like matrix multiplication: $d^{(m)} = d^{(m-1)} W$,
except $+ \rightarrow \min$
$\times \rightarrow +$

Like matrix multiplication, this is associative. So,
instead of doing $d^{|V|} = (...)W)W)W$ can do repeated squaring:

Compute $d^{(2)} = W^2$

$d^{(4)} = d^{(2)} \times d^{(2)} = W^2 \times W^2$

$d^{(8)} = ?$

All-pairs shortest paths

Note:

$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + w(k,j) \}$

Is just like matrix multiplication: $d^{(m)} = d^{(m-1)} W$,
except $+ \rightarrow \min$
$\quad x \rightarrow +$

Like matrix multiplication, this is associative. So,
instead of doing $d^{|V|} = (...)W)W)W$ can do repeated squaring:

Compute $d^{(2)} = W^2$

$d^{(4)} = d^{(2)} \times d^{(2)} = W^2 \times W^2$

$d^{(8)} = d^{(4)} \times d^{(4)}$

...

To get $d^{|V|}$ need ?

All-pairs shortest paths

Note:

$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + w(k,j) \}$

Is just like matrix multiplication: $d^{(m)} = d^{(m-1)} W$,

except $+ \rightarrow \min$

$x \rightarrow +$

Like matrix multiplication, this is associative. So,

instead of doing $d^{|V|} = (...)W)W)W$ can do repeated squaring:

Compute $d^{(2)} = W^2$

$d^{(4)} = d^{(2)} \times d^{(2)} = W^2 \times W^2$

$d^{(8)} = d^{(4)} \times d^{(4)}$

...

To get $d^{|V|}$ need log |V| multiplications only ⇒ ? time

All-pairs shortest paths

Note:

$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + w(k,j) \}$

Is just like matrix multiplication: $d^{(m)} = d^{(m-1)} W$,
except $+ \rightarrow \min$
$\quad\quad x \rightarrow +$

Like matrix multiplication, this is associative. So,
instead of doing $d^{|V|} = (...)W)W)W$ can do repeated squaring:

Compute $d^{(2)} = W^2$
$\quad d^{(4)} = d^{(2)} \times d^{(2)} = W^2 \times W^2$
$\quad d^{(8)} = d^{(4)} \times d^{(4)}$
$\quad ...$

To get $d^{|V|}$ need log |V| multiplications only $\Rightarrow$ $|V|^3 \log |V|$ time

The Floyd-Warshall algorithm

A more clever dynamic programming algorithm

Before, $d_{i,j}^{(m)}$ = shortest paths of lengths ≤ m

Next: $d_{i,j}^{(m)}$ = shortest paths from i to j such that
all INTERMEDIATE vertices are ≤ m

$d^{(0)} = W$

$d^{(m)} = ???$

The Floyd-Warshall algorithm

A more clever dynamic programming algorithm

Before, $d_{i,j}^{(m)}$ = shortest paths of lengths $\leq m$

Next: $d_{i,j}^{(m)}$ = shortest paths from i to j such that all INTERMEDIATE vertices are $\leq m$

$d^{(0)} = W$

$$d^{(m)}_{i,j} = \begin{cases} w(i,j) \\ \min\left(d^{(m-1)}_{i,j}\ ,\ d^{(m-1)}_{i,m} + d^{(m-1)}_{m,j}\right) & \text{if } m \geq 1. \end{cases}$$

Floyd-Warshall(W)

$D^{(0)} := W;$

for $m = 1$ to n

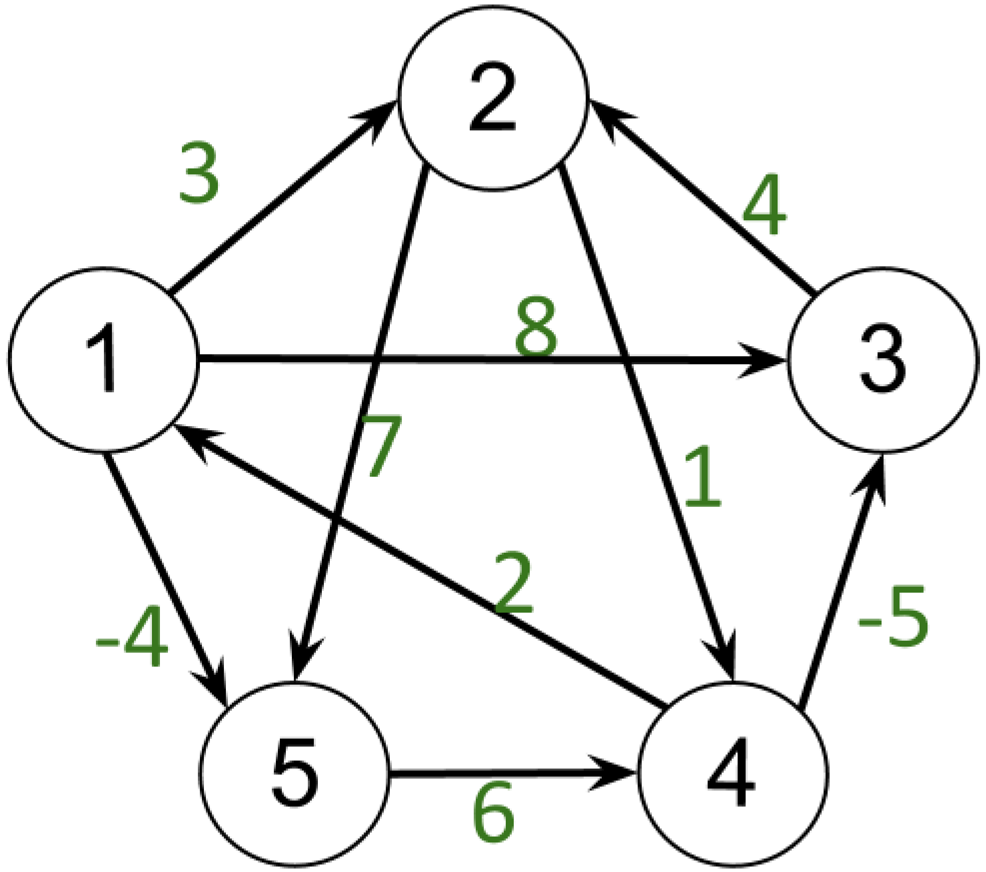
  for every i,j :

    $d^{(m)}_{i,j} = \min\left(d^{(m-1)}_{i,j}\ ,\ d^{(m-1)}_{i,m} + d^{(m-1)}_{m,j}\right)$

Return $D^{(n)}$

Time $\Theta(|V|^3)$

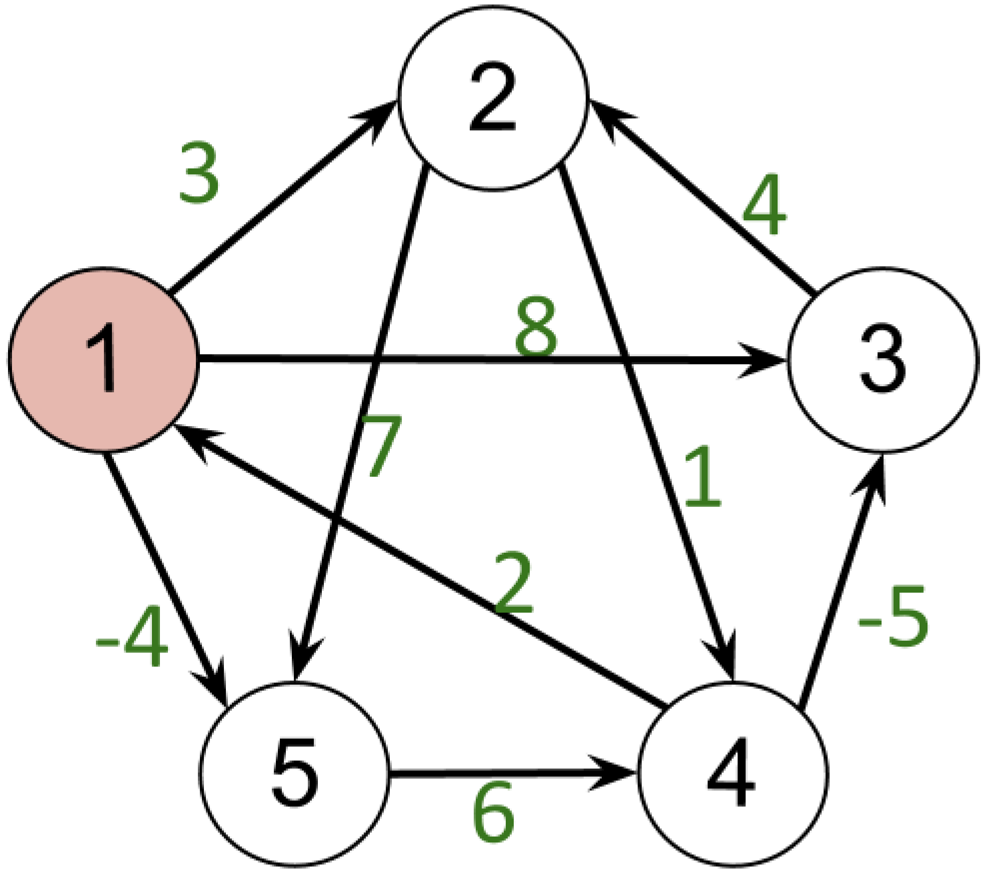
The Floyd-Warshall Example



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

$d^{(0)}$ = adjacency matrix with diagonal 0

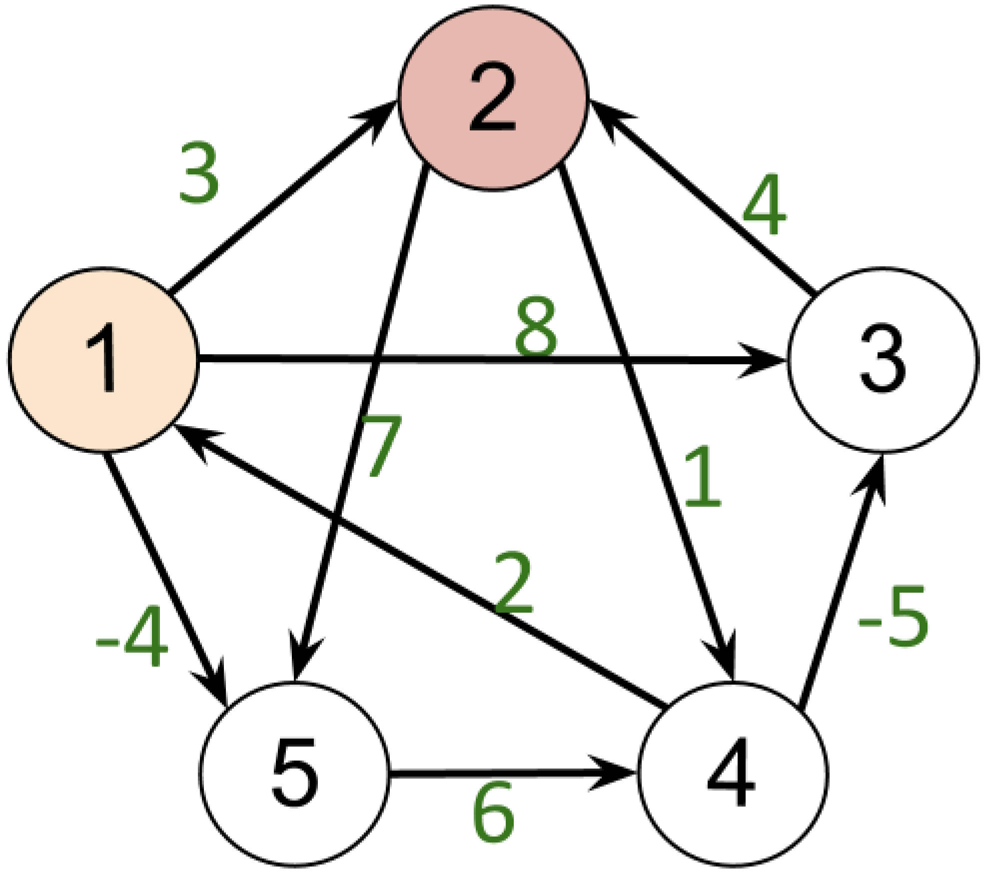
The Floyd-Warshall Example



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

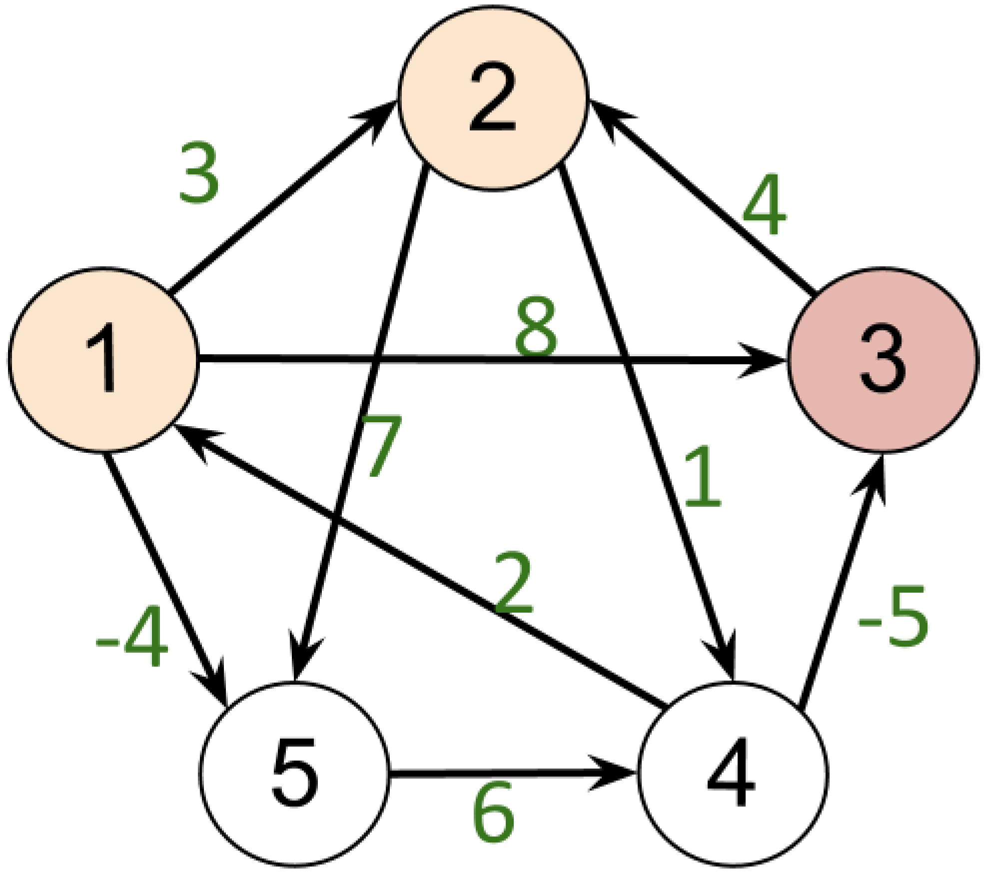
Entries $d_{(4,2)}$ and $d_{(4,5)}$ updated

The Floyd-Warshall Example



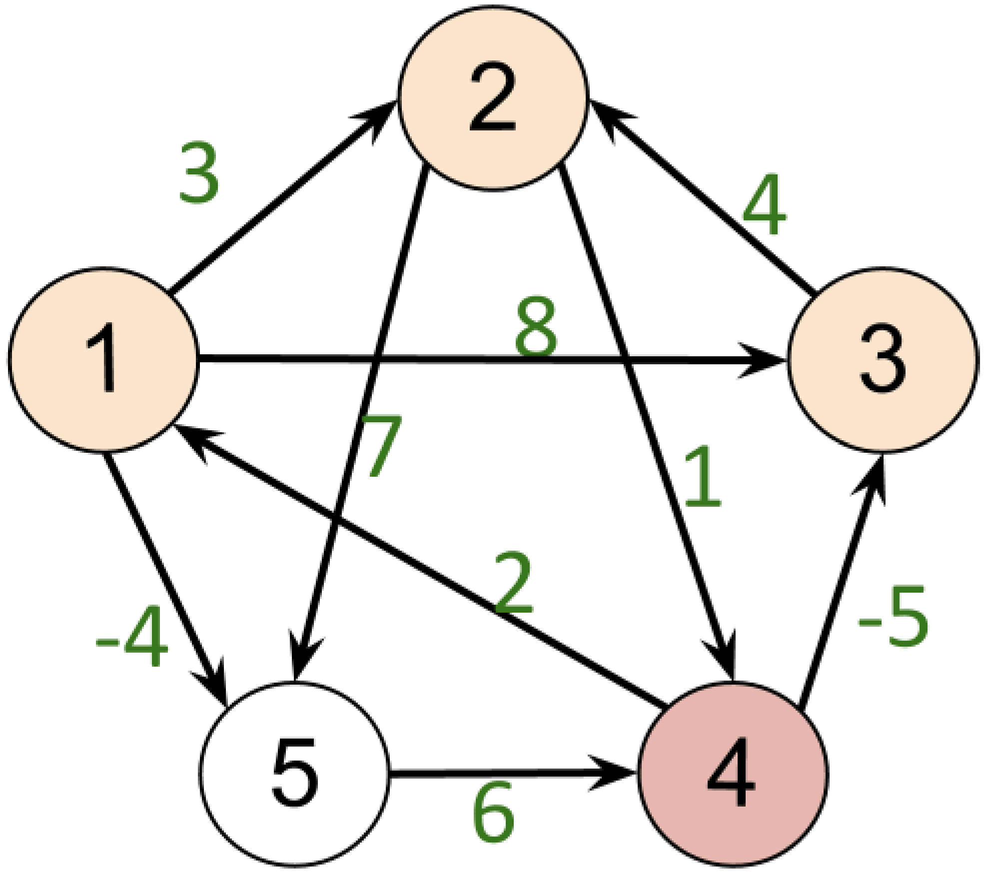
$$d^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

The Floyd-Warshall Example



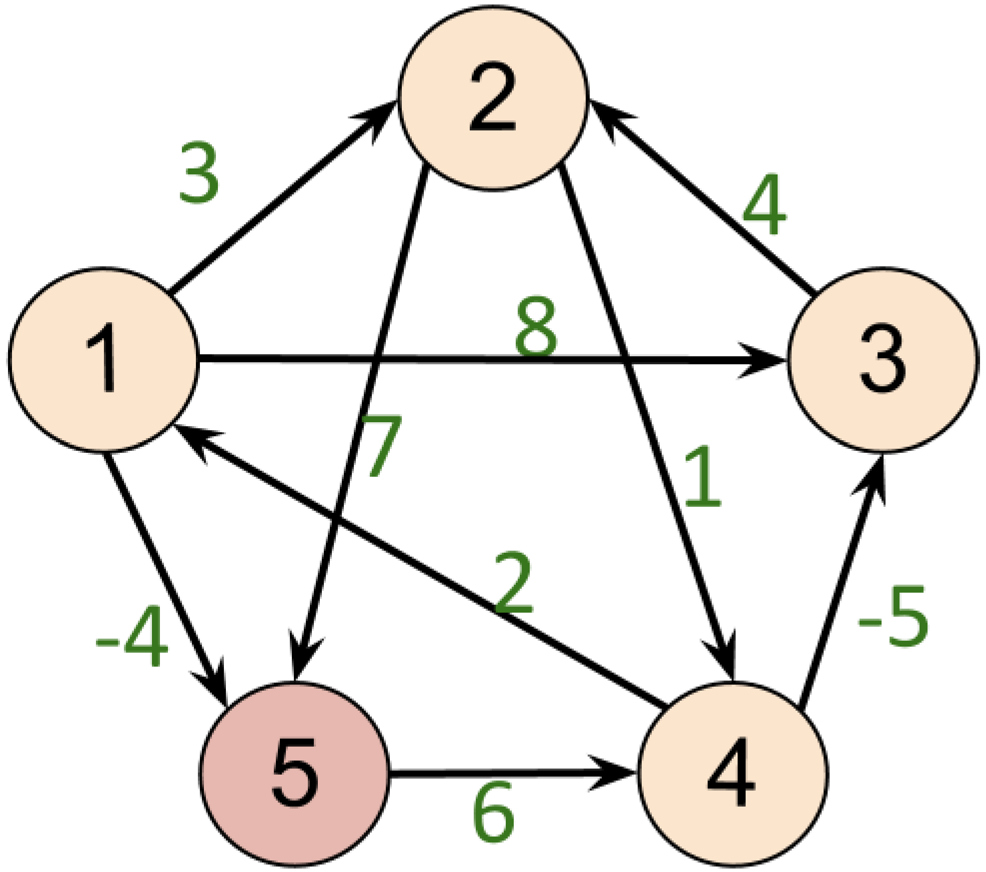
$$d^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

The Floyd-Warshall Example



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

The Floyd-Warshall Example



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

Note: Matrix multiplication/ Floyd Warshall allow for $w < 0$

If $w \geq 0$, can repeat Dijkstra. Time: $O(V^2 \log V + VE) = O(|V|^3)$

Floyd Warshall is easier and has better constants

Johnson algorithm matches Dijkstra but allows for $w < 0$.

Johnson:

Idea: Reweigh so that shortest paths don't change, but $w \geq 0$

Add new node s, with zero weight edges to all previous nodes

Run Bellman-Ford to get minimum distances from s (only)

Use Bellman-Ford distances bf(s,x) to reweigh:

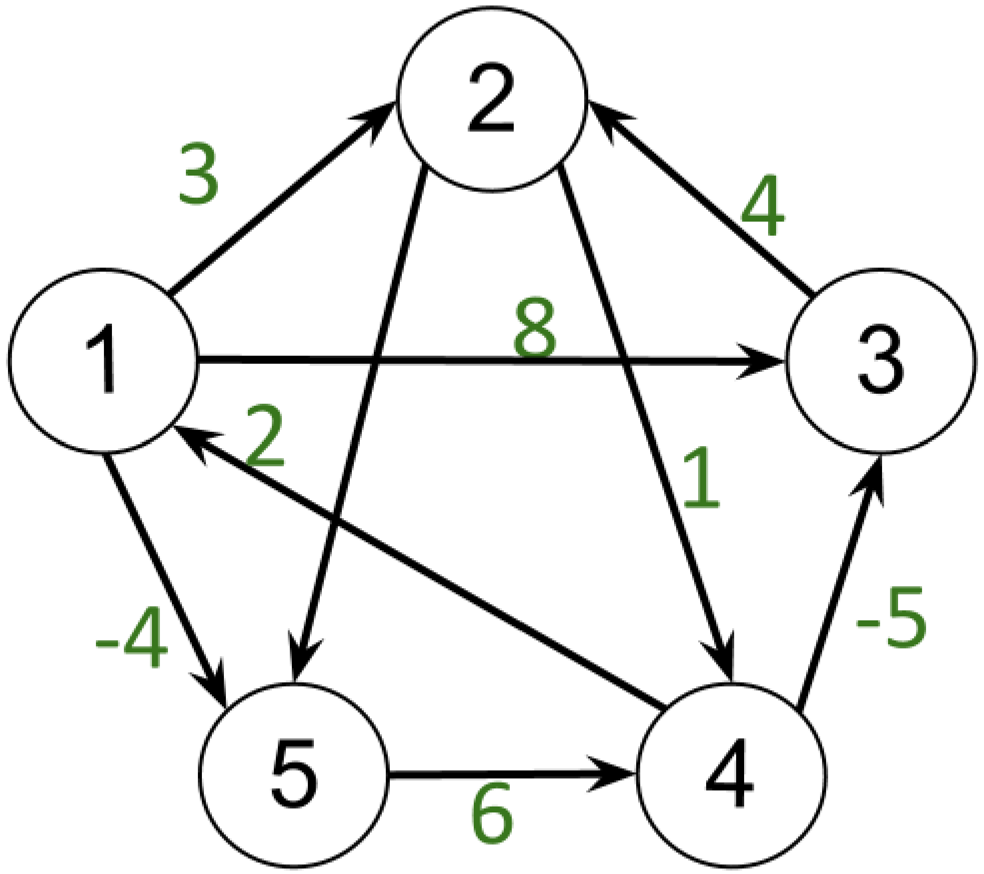
$$w'(u,v) := w(u,v) + bf(u) - bf(v)$$

(Can show this preserves shortest paths, and $w' \geq 0$)

Now run Dijkstra |V| times

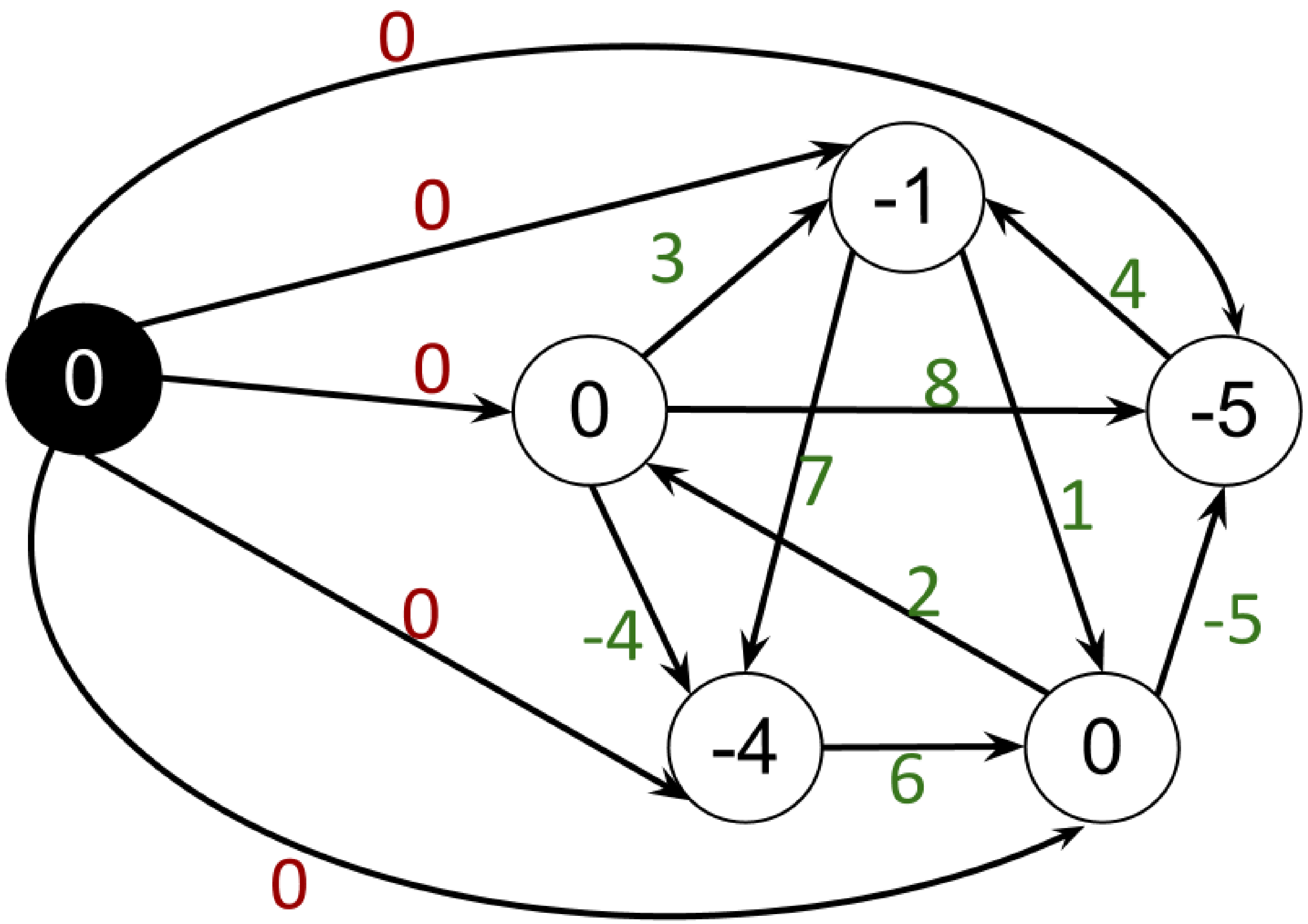
Time: $O(V E + V^2 \log V + VE) = O(V^2 \log V + VE)$.

Jonhson's Algorithm Example



Jonhson's Algorithm Example

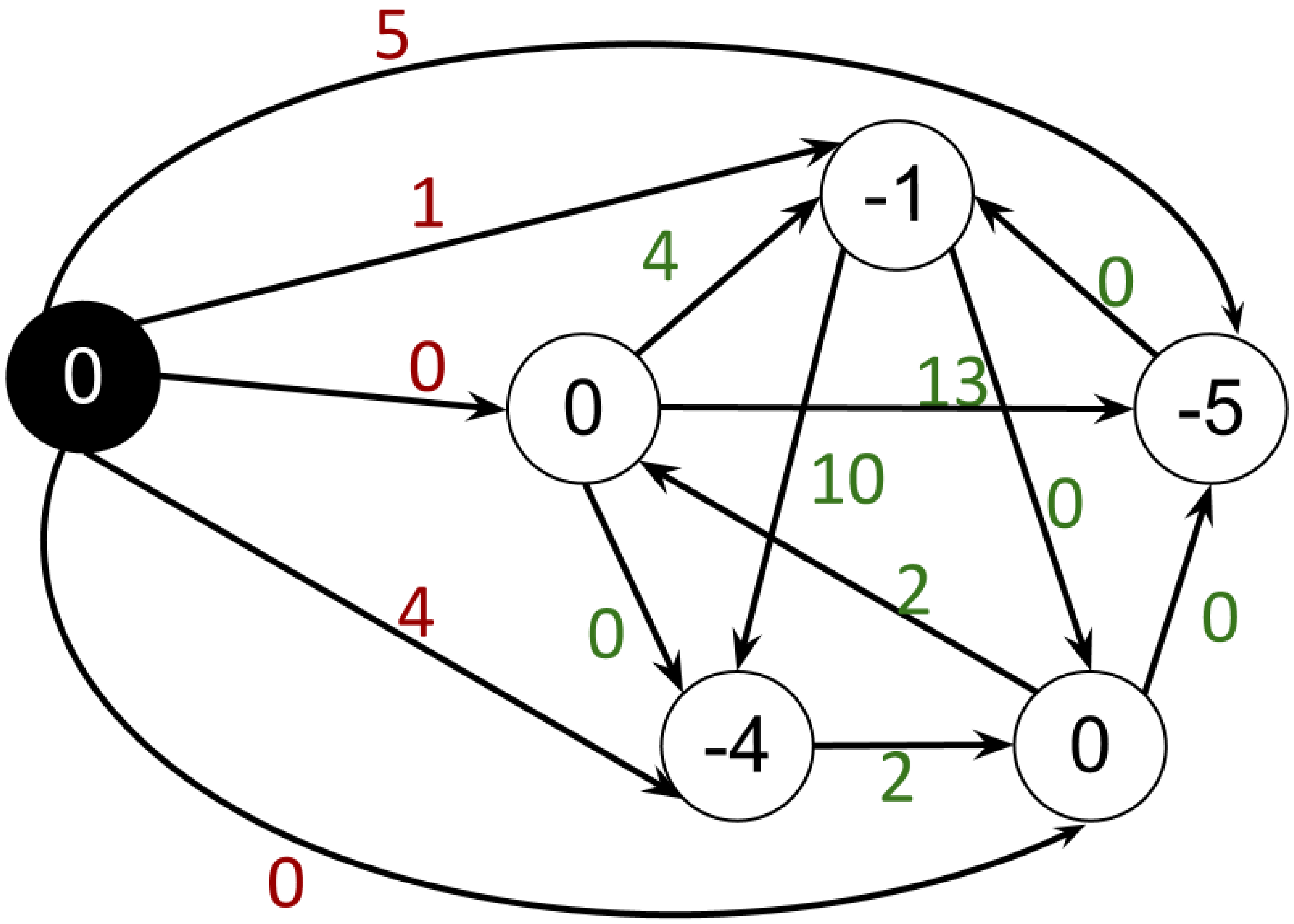
Add new node s, with weight-0 edges to all previous nodes.
Compute Bellman-Ford distance bf(s,x) from s to all nodes x
(distance shown inside the nodes)



Jonhson's Algorithm Example

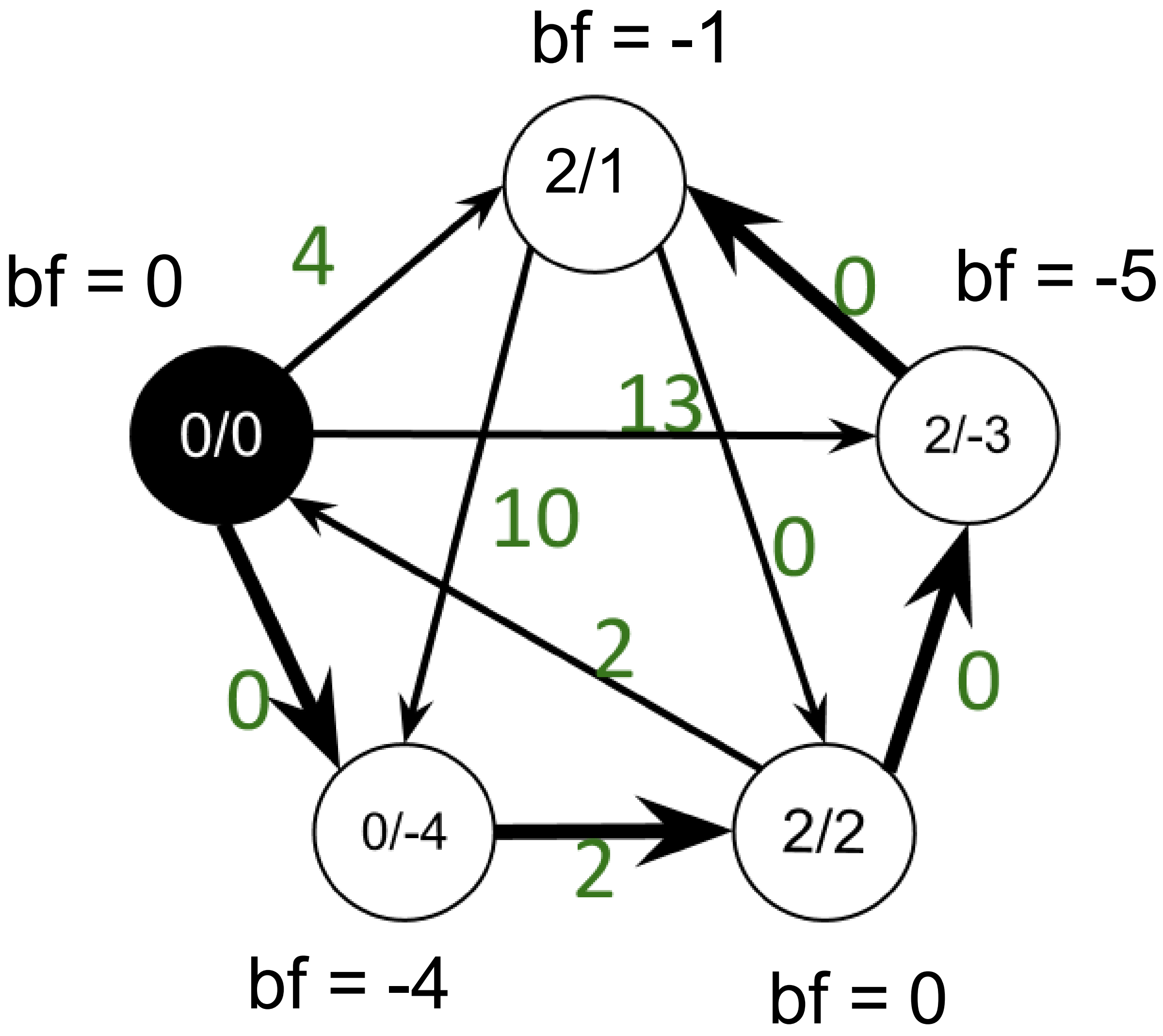
Use Bellman-Ford distances bf(s,x) to reweight:

w'(u,v) = w(u,v) + bf(u) – bf(v)



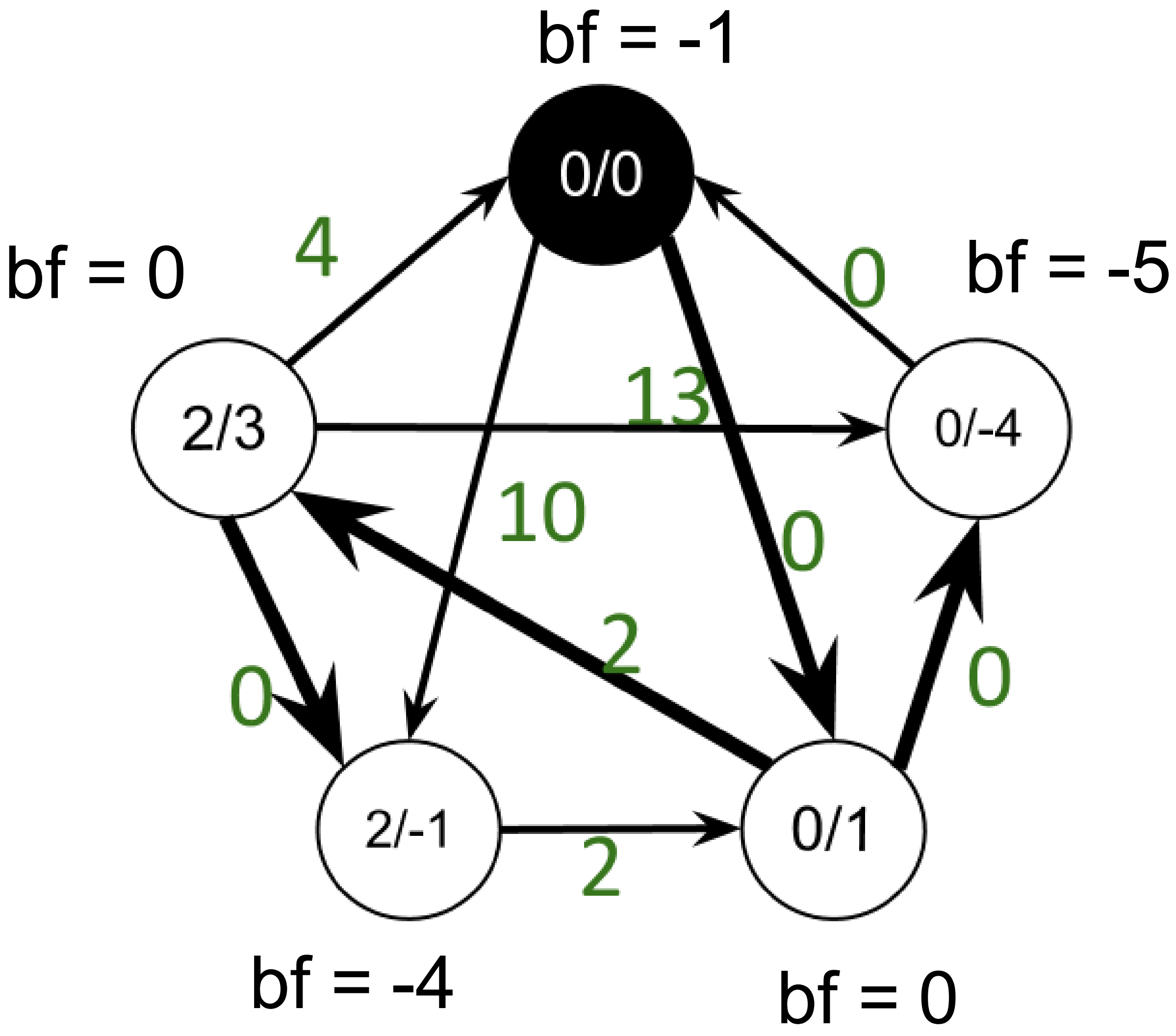
Jonhson's Algorithm Example

Run Dijkstra algorithm from each node.
Inside each node are minimum distances d'/d w.r.t. w' and w
$d(u,v) = d'(u,v) + bf(v) - bf(u) \geq 0$



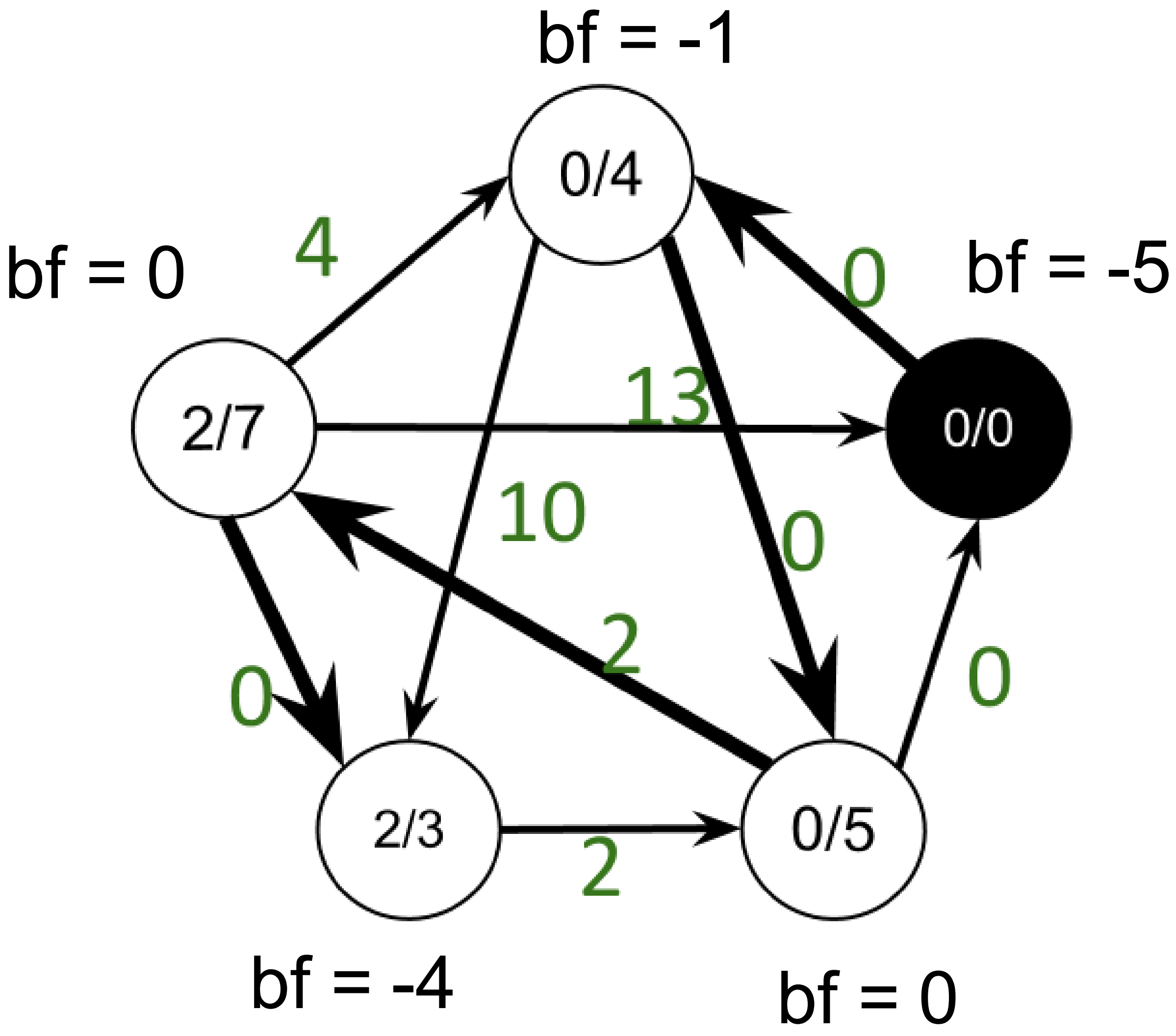
Jonhson's Algorithm Example

Run Dijkstra algorithm on each node.



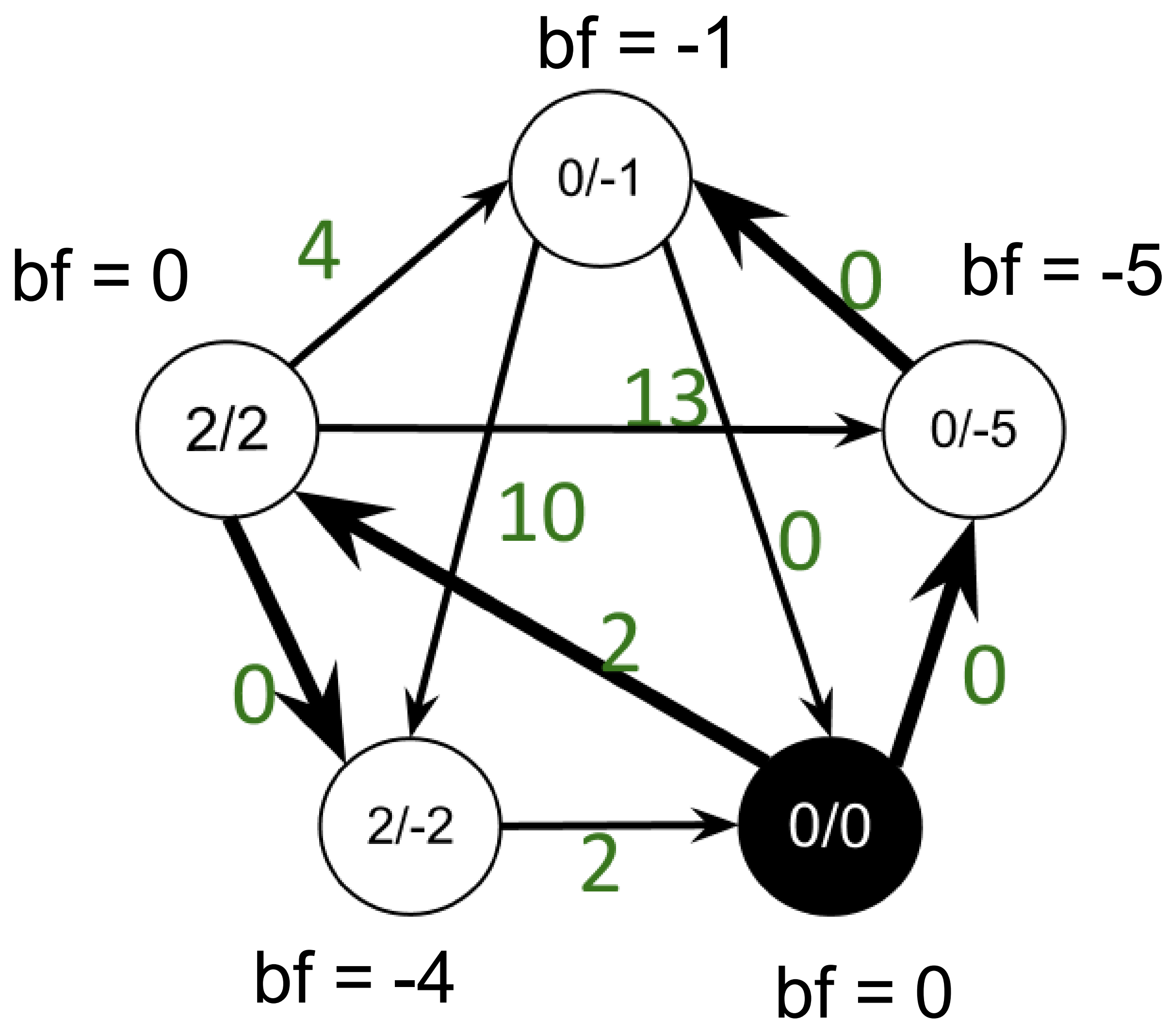
Jonhson's Algorithm Example

Run Dijkstra algorithm on each node.



Jonhson's Algorithm Example

Run Dijkstra algorithm on each node.



Jonhson's Algorithm Example

Run Dijkstra algorithm on each node.