Analysis and Design of Algorithms

Graphs
Part II: Finding Shortest Paths

Instructor: **Morteza Zakeri**

Shortest Path Problems

- How can we find the shortest route between two points on a road map?
- Model the problem as a graph problem:
  - Road map is a weighted graph:

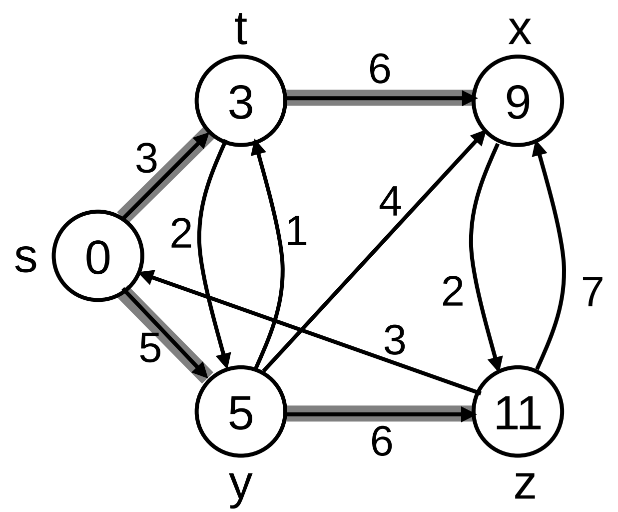
    vertices = cities

    edges = road segments between cities

    edge weights = road distances
  - Goal: find a shortest path between two vertices (cities)

Shortest Path Problem

- **Input:**
  - Directed graph G = (V, E)
  - Weight function w : E → **R**



- **Weight of path** p = $\langle v_0, v_1, \ldots, v_k \rangle$

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

- **Shortest-path weight** from u to v:

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

- **Note:** there might be multiple shortest paths from u to v

Variants of Shortest Path

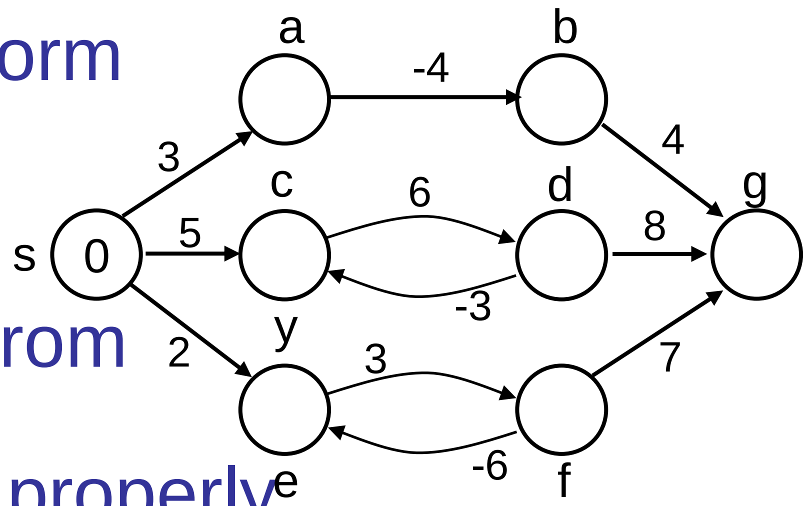
- **Single-pair shortest path**
  - Find a shortest path from u to v for given vertices u and v
- **Single-source shortest paths**
  - G = (V, E) $\Rightarrow$ Find a shortest path from a given source vertex *s* to each vertex v $\in$ V
  - **Dijkstra** and **Bellman-Ford** algorithm algorithms
- **Single-destination shortest paths**
  - Find a shortest path to a given destination vertex **t** from each vertex v
  - Reversing the direction of each edge $\Rightarrow$ single-source

Variants of Shortest Paths (cont'd)

- **All-pairs shortest-paths**
  - Find a shortest path from u to v for every pair of vertices u and v
  - **Floyd-Warshall** algorithm
    - $O(V^3)$

Negative-Weight Edges

- Negative-weight edges may form negative-weight cycles
- If such cycles are reachable from the source, then δ(s, v) is not properly defined!
  - Keep going around the cycle, and get w(s, v) = - ∞ for all v on the cycle



Negative-Weight Edges

- s → a: only one path

  δ(s, a) = w(s, a) = 3

- s → b: only one path

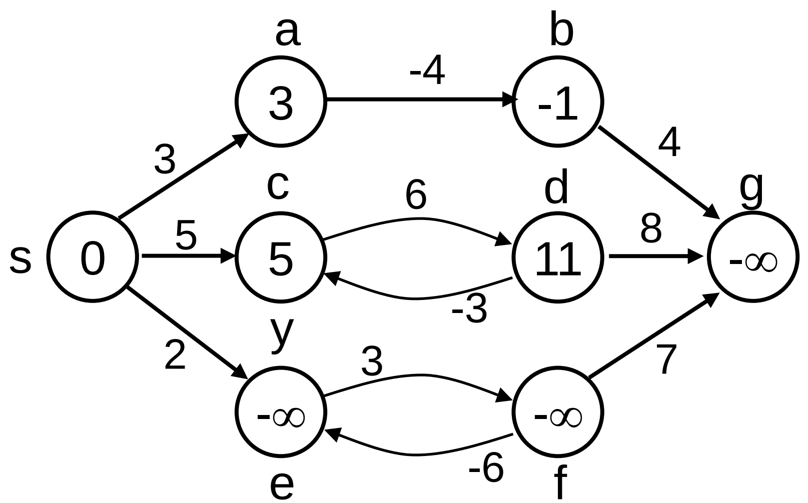
  δ(s, b) = w(s, a) + w(a, b) = -1

- s → c: infinitely many paths

  ⟨s, c⟩, ⟨s, c, d, c⟩, ⟨s, c, d, c, d, c⟩

  cycle has positive weight (6 - 3 = 3)

  ⟨s, c⟩ is shortest path with weight δ(s, b) = w(s, c) = 5



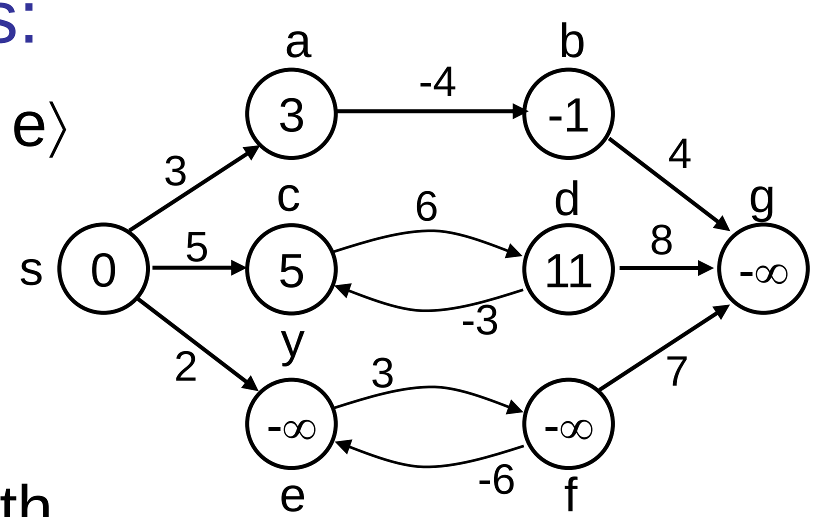
Negative-Weight Edges

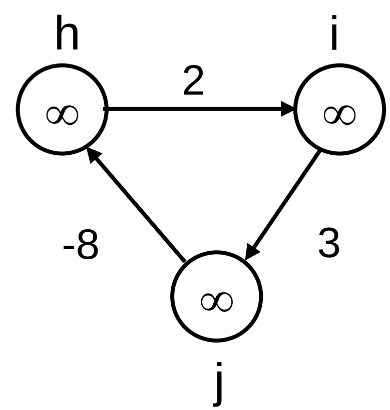
- $s \rightarrow e$: infinitely many paths:
  - $\langle s, e\rangle, \langle s, e, f, e\rangle, \langle s, e, f, e, f, e\rangle$
  - cycle $\langle e, f, e\rangle$ has negative weight:

    $$3 + (-6) = -3$$

  - Can find paths from *s* to *e* with arbitrarily large negative weights
  - $\delta(s, e) = -\infty \Rightarrow$ no shortest path exists between *s* and *e*
  - Similarly: $\delta(s, f) = -\infty$,

    $\delta(s, g) = -\infty$





h, i, j not reachable from s

$\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$

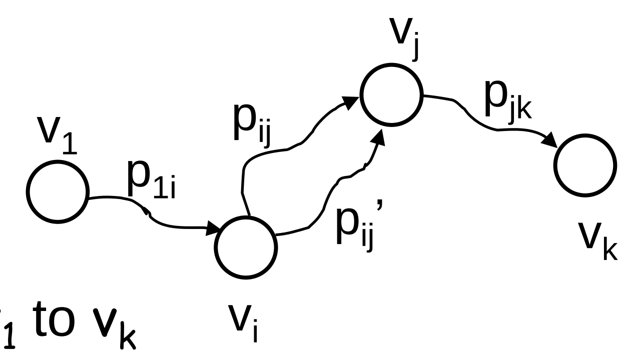
Cycles

- Can shortest paths contain cycles?
- Negative-weight cycles No!
  - Shortest path is not well defined
- Positive-weight cycles: No!
  - By removing the cycle, we can get a shorter path
- Zero-weight cycles
  - No reason to use them
  - Can remove them to obtain a path with same weight

Optimal Substructure Theorem

Given:

- A weighted, directed graph $G = (V, E)$
- A weight function $w: E \rightarrow \mathbf{R}$,
- A shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ from v_1 to v_k
- A subpath of p: $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$, with $1 \le i \le j \le k$



Then: p_{ij} is a shortest path from v_i to v_j

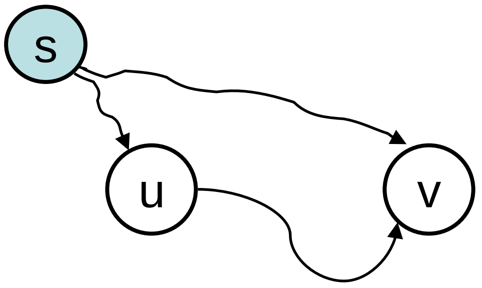
Proof: $p = v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$

$$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$$

Assume $\exists$ p_{ij}' from v_i to v_j with $w(p_{ij}') < w(p_{ij})$

$\Rightarrow w(p') = w(p_{1i}) + w(p_{ij}') + w(p_{jk}) < w(p)$ contradiction!

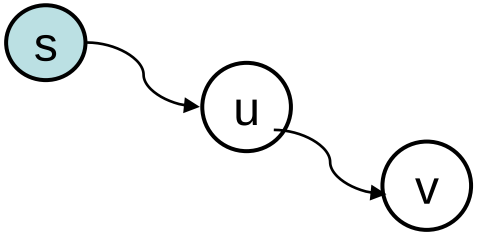
Triangle Inequality



For all (u, v) $\in$ E, we have:

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v)$$

- If u is on the shortest path to v we have the equality sign



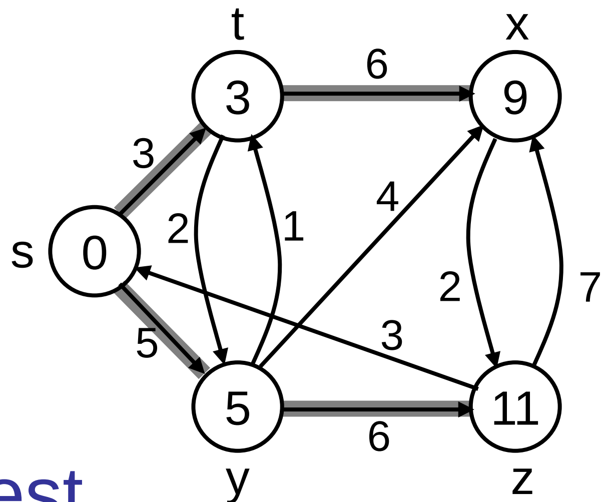
Single-Source Shortest Paths Algorithms

- Bellman-Ford algorithm
  - Negative weights are allowed
  - Negative cycles reachable from the source are not allowed.
- Dijkstra's algorithm
  - Negative weights are not allowed
- Operations common in both algorithms:
  - Initialization
  - Relaxation

Shortest-Paths Notation

For each vertex v ∈ V:

- δ(s, v): **shortest-path weight**
- d[v]: shortest-path weight **estimate**
  - Initially, d[v]=∞
  - d[v]→δ(s,v) as algorithm progresses
- π[v] = **predecessor** of v on a shortest path from *s*
  - If no predecessor, π[v] = NIL
  - π induces a tree—**shortest-path tree**



Initialization

Alg.: INITIALIZE-SINGLE-SOURCE(V, s)

1. **for** each $v \in V$

2. **do** $d[v] \leftarrow \infty$

3. $\pi[v] \leftarrow$ NIL

4. $d[s] \leftarrow 0$

- All the shortest-paths algorithms start with INITIALIZE-SINGLE-SOURCE

Relaxation Step

- **Relaxing** an edge (u, v) = testing whether we can improve the shortest path to v found so far by going through u

    If $d[v] > d[u] + w(u, v)$

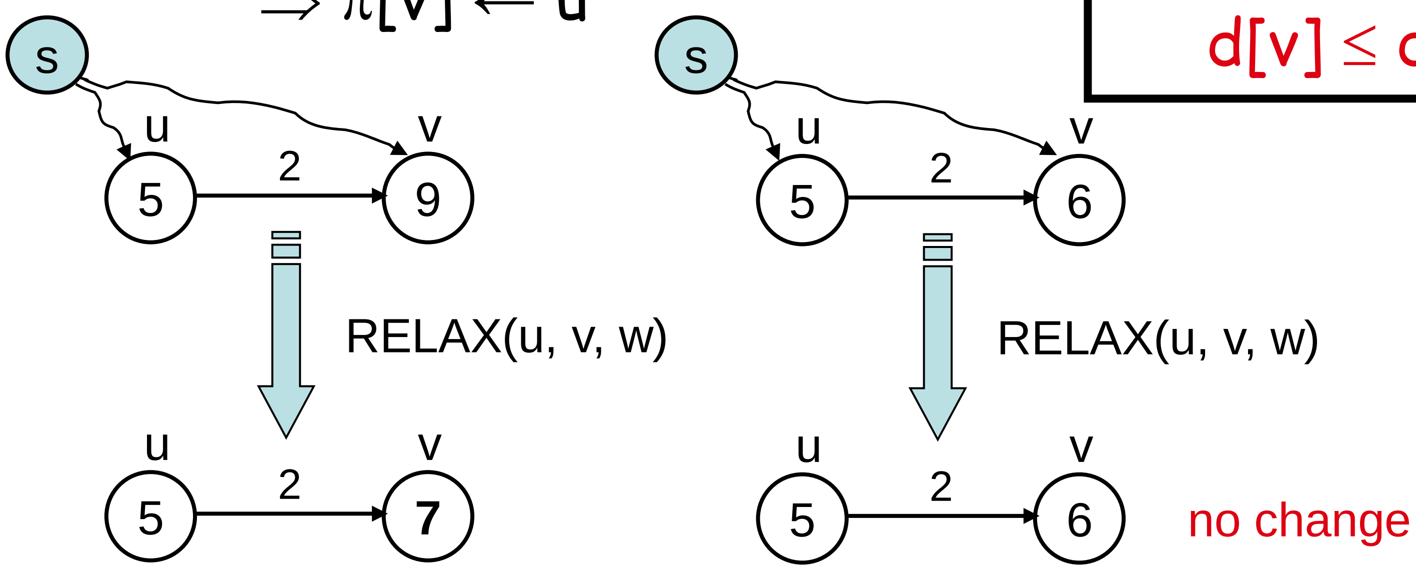
    we can improve the shortest path to v

    $\Rightarrow d[v]=d[u]+w(u,v)$

    $\Rightarrow \pi[v] \leftarrow u$

After relaxation:

$d[v] \leq d[u] + w(u, v)$



Bellman-Ford Algorithm

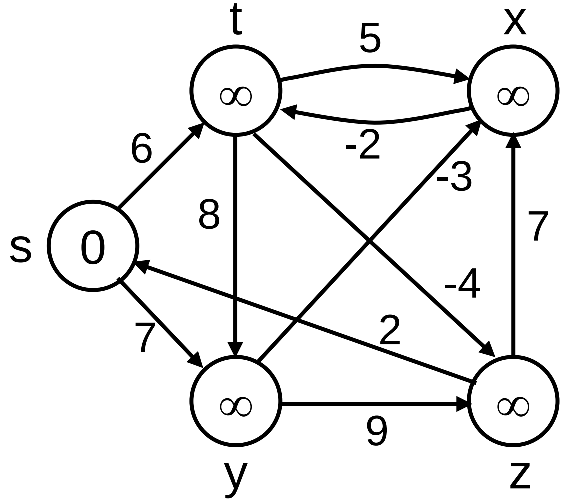
- Single-source shortest path problem
  - Computes $\delta(s, v)$ and $\pi[v]$ for all $v \in V$

- Allows negative edge weights - can detect negative cycles.
  - Returns TRUE if no negative-weight cycles are reachable from the source s
  - Returns FALSE otherwise $\Rightarrow$ no solution exists

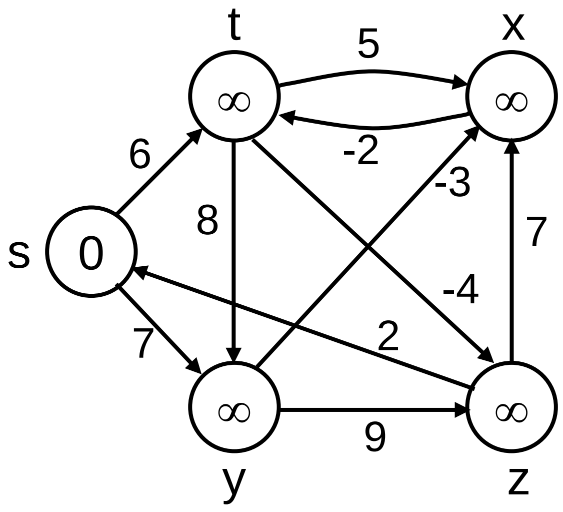
Bellman-Ford Algorithm (cont'd)

- Idea:
  - Each edge is relaxed |V–1| times by making |V-1| passes over the whole edge set.
  - To make sure that each edge is relaxed exactly |V – 1| times, it puts the edges in an unordered list and goes over the list |V – 1| times.

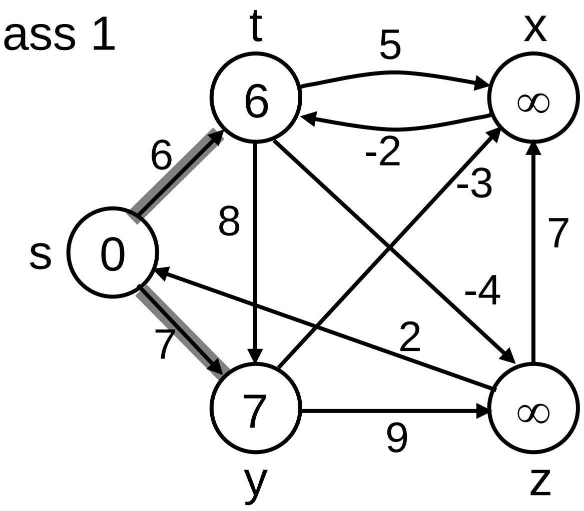
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)



BELLMAN-FORD(V, E, w, s)



Pass 1

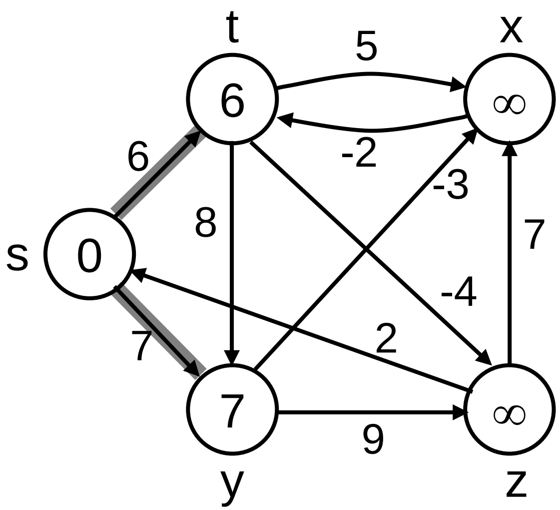


E: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

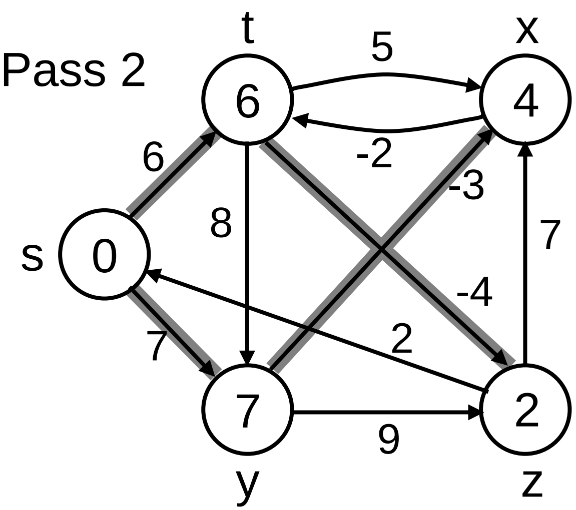
Example

(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

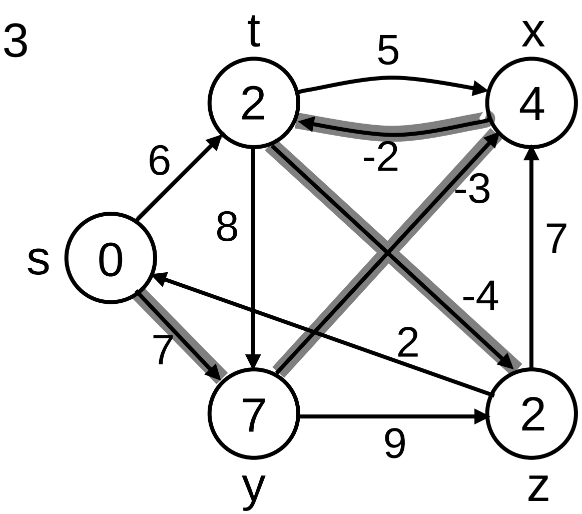
Pass 1 (from previous slide)



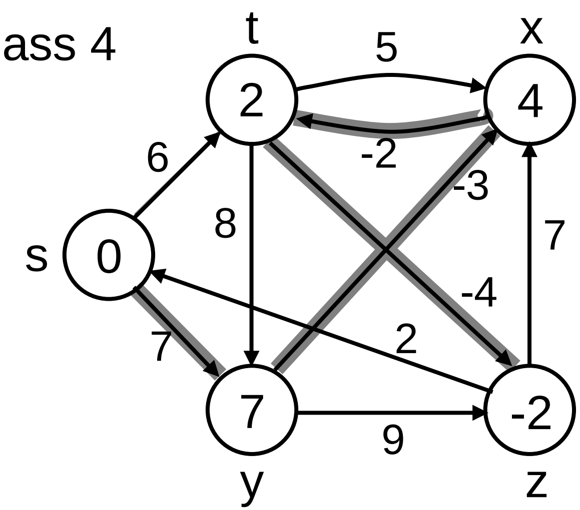
Pass 2



Pass 3

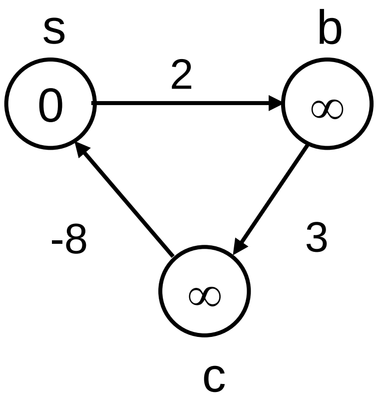


Pass 4

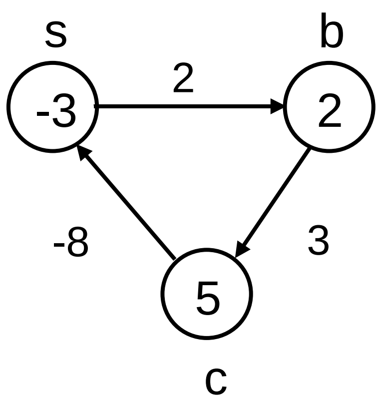


Detecting Negative Cycles (perform extra test after V-1 iterations)

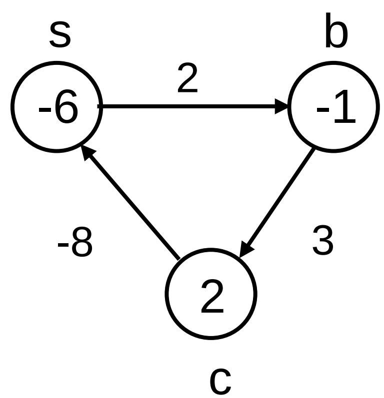
- **for** each edge $(u, v) \in E$
- **do if** $d[v] > d[u] + w(u, v)$
- **then return** FALSE
- **return** TRUE



1st pass



2nd pass



(s,b) (b,c) (c,s)

Look at edge (s, b):

$d[b] = -1$

$d[s] + w(s, b) = -4$

$\Rightarrow d[b] > d[s] + w(s, b)$

BELLMAN-FORD(V, E, w, s)

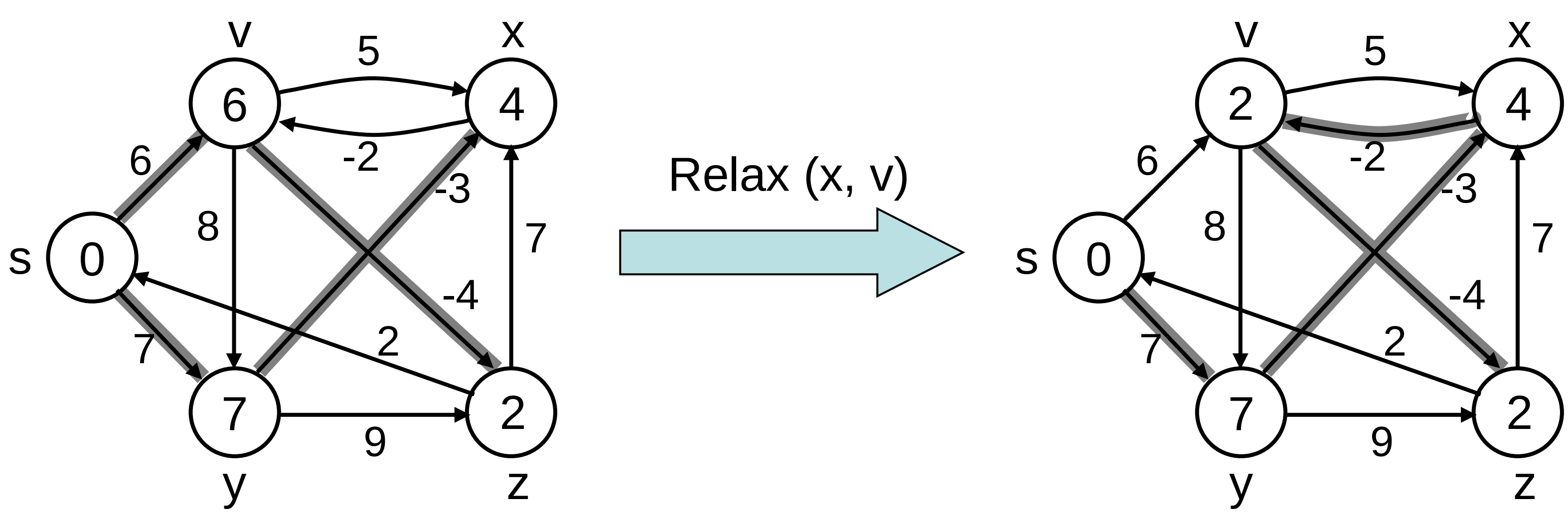
1. INITIALIZE-SINGLE-SOURCE(V, s) $\longleftarrow$ $\Theta(V)$
2. **for** i ← 1 to |V| - 1 $\longleftarrow$ O(V)
3. **do for** each edge (u, v) ∈ E $\longleftarrow$ O(E)
4. **do** RELAX(u, v, w)
5. **for** each edge (u, v) ∈ E $\longleftarrow$ O(E)
6. **do if** d[v] > d[u] + w(u, v)
7. **then return** FALSE
8. **return** TRUE

O(VE)

Running time: O(V+VE+E)=O(VE)

Shortest Path Properties

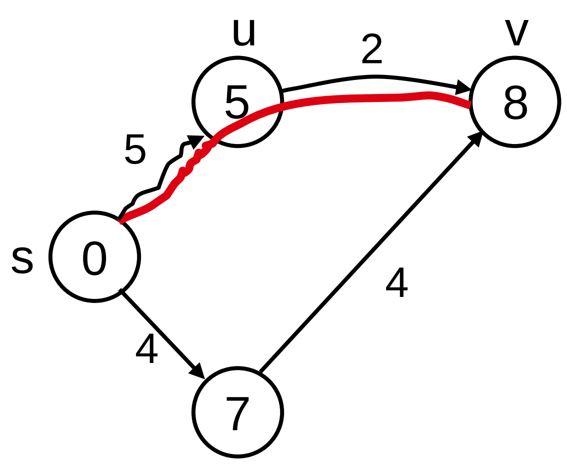
- **Upper-bound property**
  - We always have $d[v] \geq \delta(s, v)$ for all v.
  - The estimate never goes up – relaxation only lowers the estimate



Shortest Path Properties

- **Convergence property**

If $s \rightsquigarrow u \rightarrow v$ is a shortest path, 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 after relaxing (u, v).



- If $d[v] > \delta(s, v) \Rightarrow$ after relaxation:

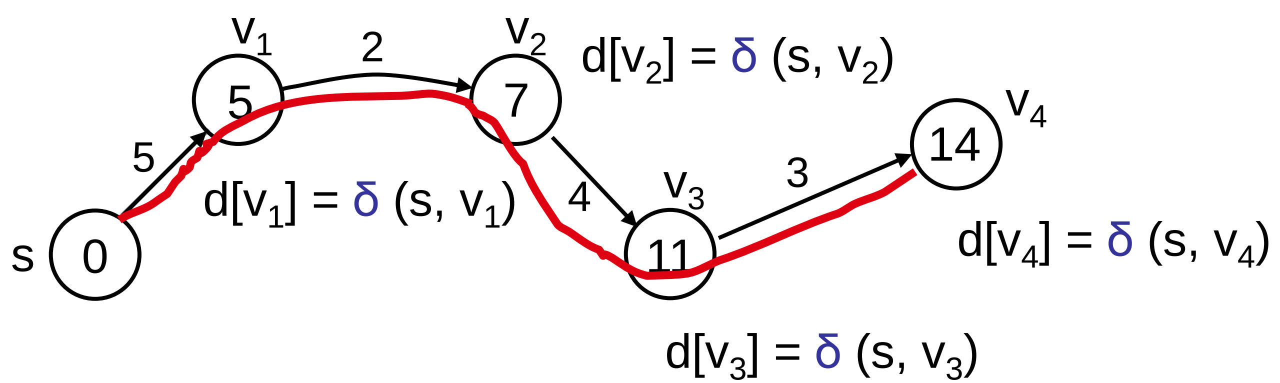
  $d[v] = d[u] + w(u, v)$

  $d[v] = 5 + 2 = 7$
- Otherwise, the value remains unchanged, because it must have been the shortest path value

Shortest Path Properties

- **Path relaxation property**

  Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If we relax, in order, (v_0, v_1), (v_1, v_2), . . . , (v_{k-1}, v_k), even intermixed with other relaxations, then $d[v_k] = \delta(s, v_k)$.



Correctness of Belman-Ford Algorithm

- **Theorem:** Show that $d[v]= \delta (s, v)$, for every v, after |V-1| passes.

Case 1: G does not contain negative cycles which are reachable from s

– Assume that the shortest path from s to v is
$p = \langle v_0, v_1, \ldots, v_k \rangle$, where $s=v_0$ and $v=v_k$, $k \leq |V-1|$

– Use mathematical induction on the number of passes i to show that:

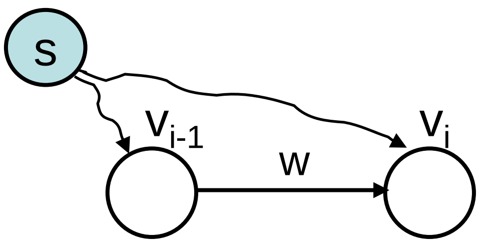
$$d[v_i]= \delta (s, v_i) , i=0,1,\ldots,k$$

Correctness of Belman-Ford Algorithm (cont.)

Base Case: $i=0$ $d[v_0]= \delta(s, v_0)= \delta(s, s)= 0$

Inductive Hypothesis: $d[v_{i-1}]= \delta(s, v_{i-1})$

Inductive Step: $d[v_i]= \delta(s, v_i)$



$d[v_{i-1}]= \delta(s, v_{i-1})$

After relaxing (v_{i-1}, v_i):

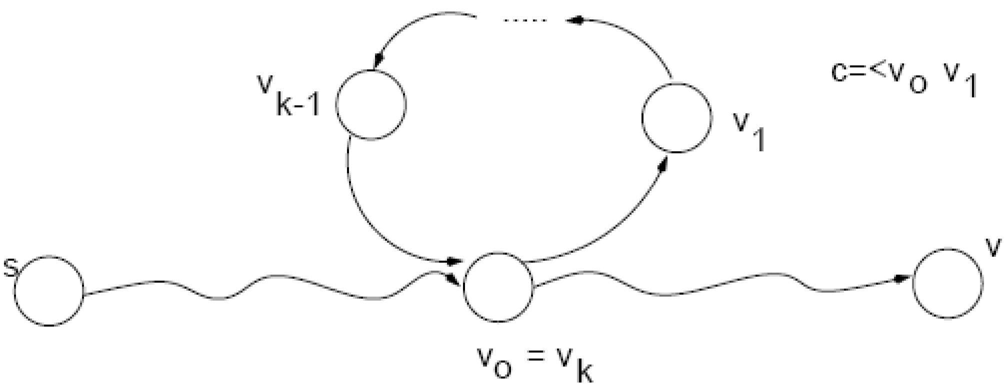
$d[v_i] \le d[v_{i-1}]+w= \delta(s, v_{i-1})+w= \delta(s, v_i)$

From the upper bound property: $d[v_i] \ge \delta(s, v_i)$

Therefore, $d[v_i]=\delta(s, v_i)$

Correctness of Belman-Ford Algorithm (cont.)

- <u>Case 2:</u> G contains a negative cycle which is reachable from s



$c = \langle v_0\ v_1\ \ldots\ v_k \rangle$ is a negative cycle

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

Proof by Contradiction: suppose the algorithm returns a solution

After relaxing (v_{i-1}, v_i): $\mathbf{d}[v_i] \le \mathbf{d}[v_{i-1}] + w(v_{i-1}, v_i)$

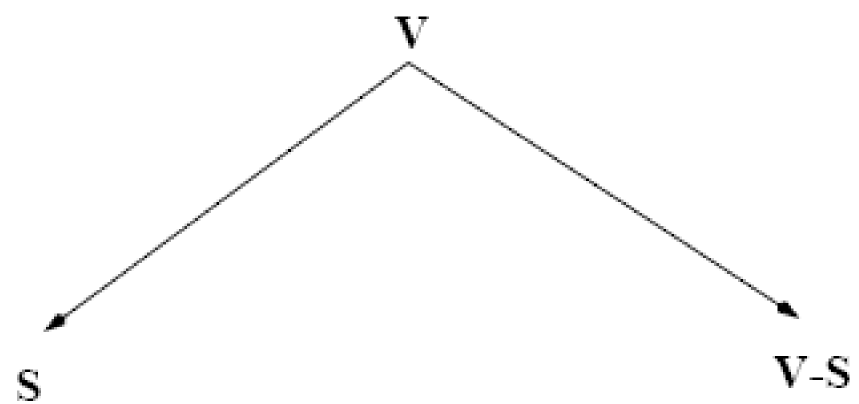
$$\text{or } \sum_{i=1}^{k} \mathbf{d}[v_i] \le \sum_{i=1}^{k} \mathbf{d}[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$\text{or } \sum_{i=1}^{k} w(v_{i-1}, v_i) \ge 0 \ \left(\sum_{i=1}^{k} \mathbf{d}[v_i] = \sum_{i=1}^{k} \mathbf{d}[v_{i-1}]\right)$$

Contradiction!

Dijkstra's Algorithm

- Single-source shortest path problem:
  - No negative-weight edges: $w(u, v) > 0, \ \forall (u, v) \in E$
- Each edge is relaxed **only once!**
- Maintains two sets of vertices:



$d[v]=\delta(s, v)$

$d[v]>\delta(s, v)$

Dijkstra's Algorithm (cont.)

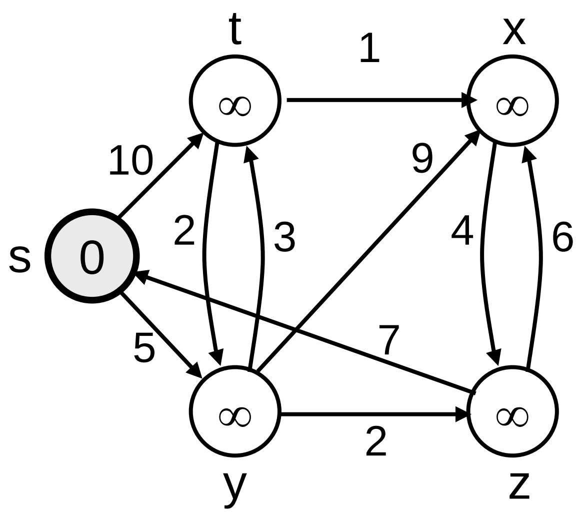
- Vertices in V – S reside in a min-priority queue
  - Keys in Q are estimates of shortest-path weights d[u]
- Repeatedly select a vertex u $\in$ V – S, with the minimum shortest-path estimate d[u]
- Relax all edges leaving u

- **Steps**

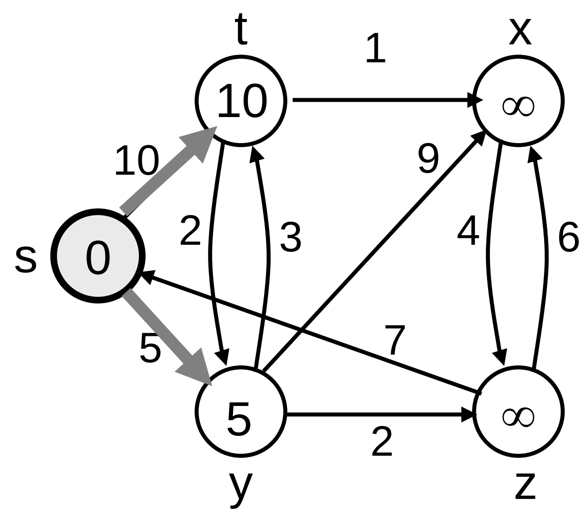
  1) Extract a vertex u from Q (i.e., u has the highest priority)
  2) Insert u to S
  3) Relax all edges leaving u
  4) Update Q

Dijkstra (G, w, s)

S=<> Q=<s,t,x,z,y>

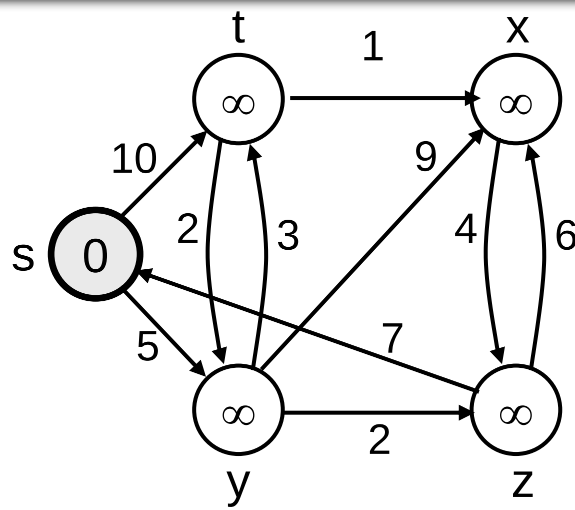


S=<s> Q=<y,t,x,z>

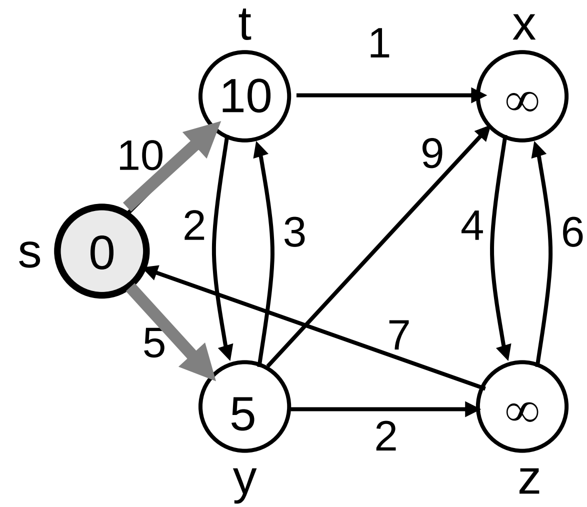


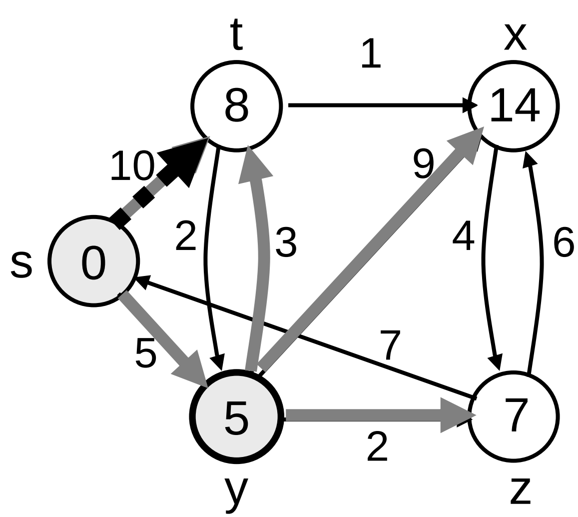
Example (cont.)

S=<> Q=<s,t,x,z,y>

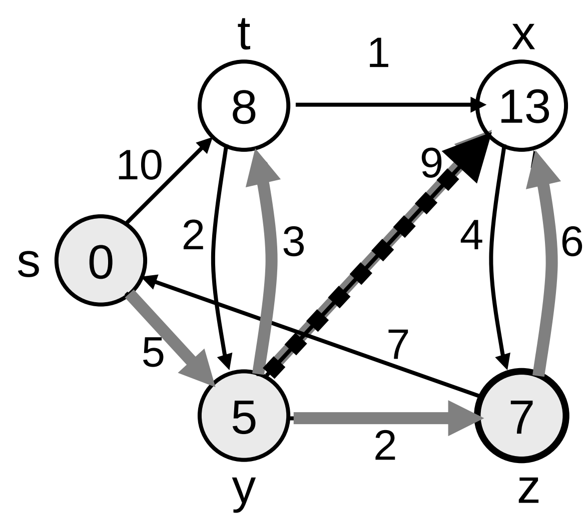


S=<s> Q=<y,t,x,z>





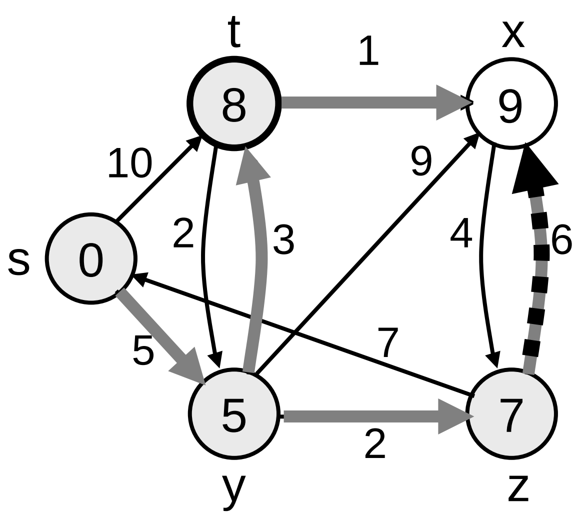
S=<s,y> Q=<z,t,x>



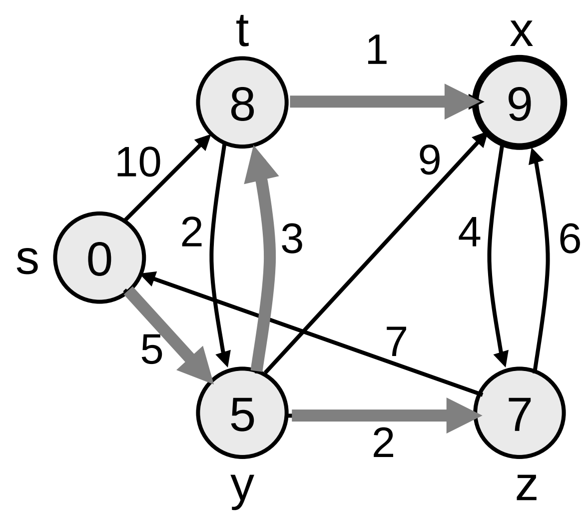
S=<s,y,z> Q=<t,x>

Example (cont.)

S=<s,y,z,t> Q=<x>



S=<s,y,z,t,x> Q=<>



Dijkstra (G, w, s)

```
INITIALIZE-SINGLE-SOURCE(V, s)          ← Θ(V)
S ← ∅
Q ← V[G]                                ← O(V) build min-heap
while Q ≠ ∅                             ← Executed O(V) times
    do u ← EXTRACT-MIN(Q)               ← O(lgV)
        S ← S ∪ {u}
        for each vertex v ∈ Adj[u]      ← O(E) times (total)
            do RELAX(u, v, w)
            Update Q (DECREASE_KEY)     ← O(lgV)
```

O(VlgV)

O(ElgV)

Running time: $O(V\lg V + E\lg V) = O(E\lg V)$

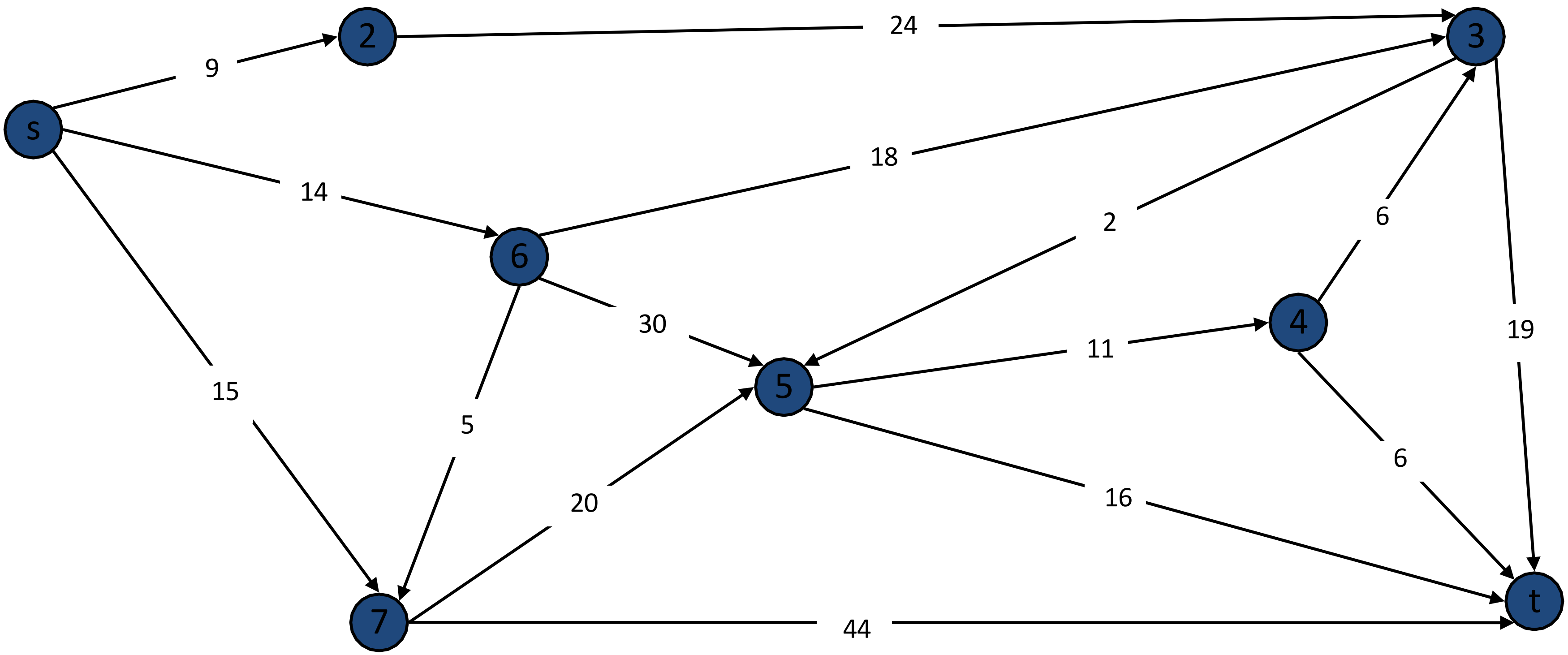
Binary Heap vs Fibonacci Heap

Running time depends on the implementation of the heap

	EXTRACT-MIN	DECREASE-KEY	Total
binary heap	O(lgV)	O(lgV)	O(ElgV)
Fibonacci heap	O(lgV)	O(1)	O(VlgV + E)

Dijkstra's Shortest Path Algorithm

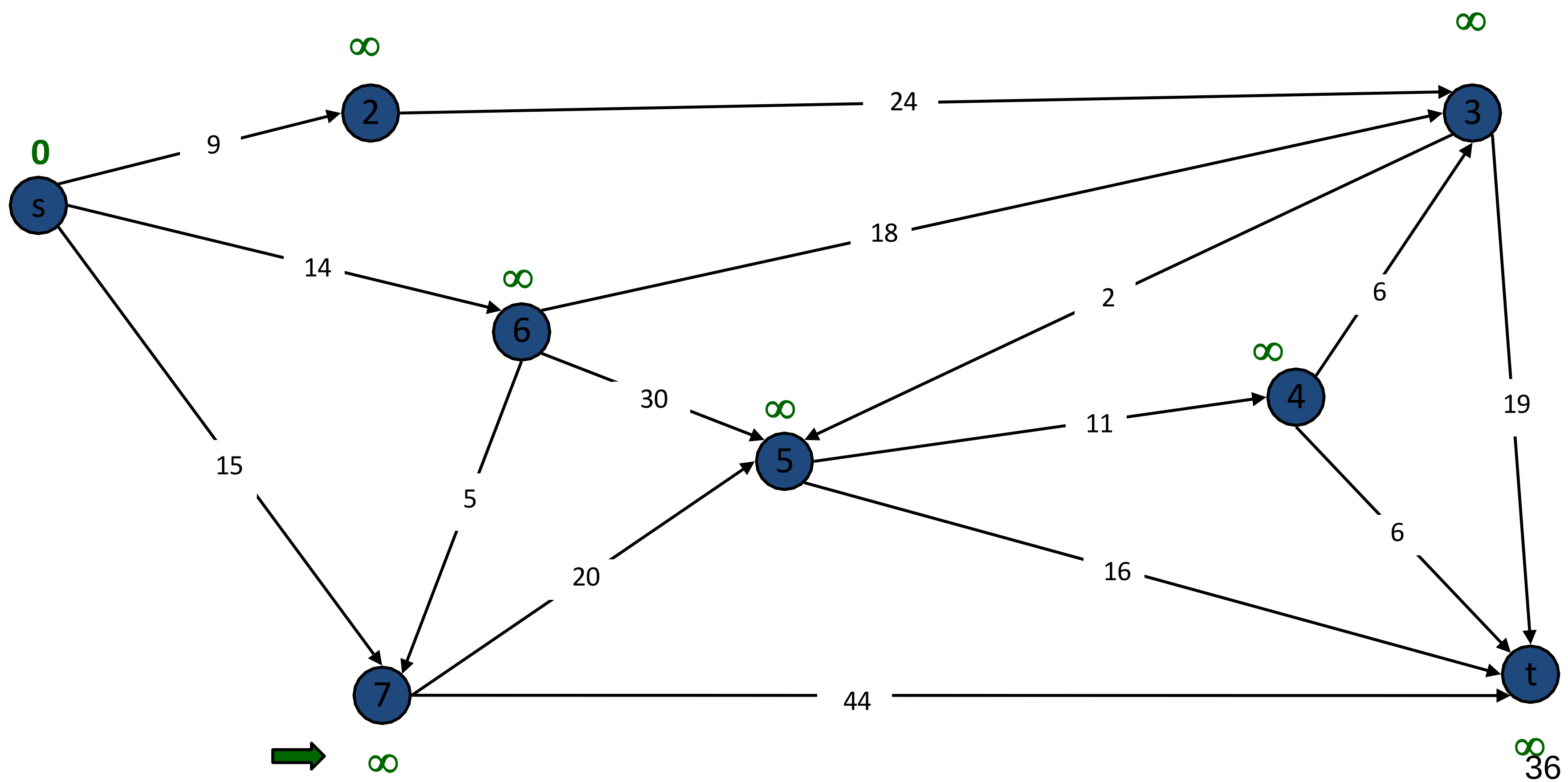
- Find shortest path from s to t.



Dijkstra's Shortest Path Algorithm

S = { }

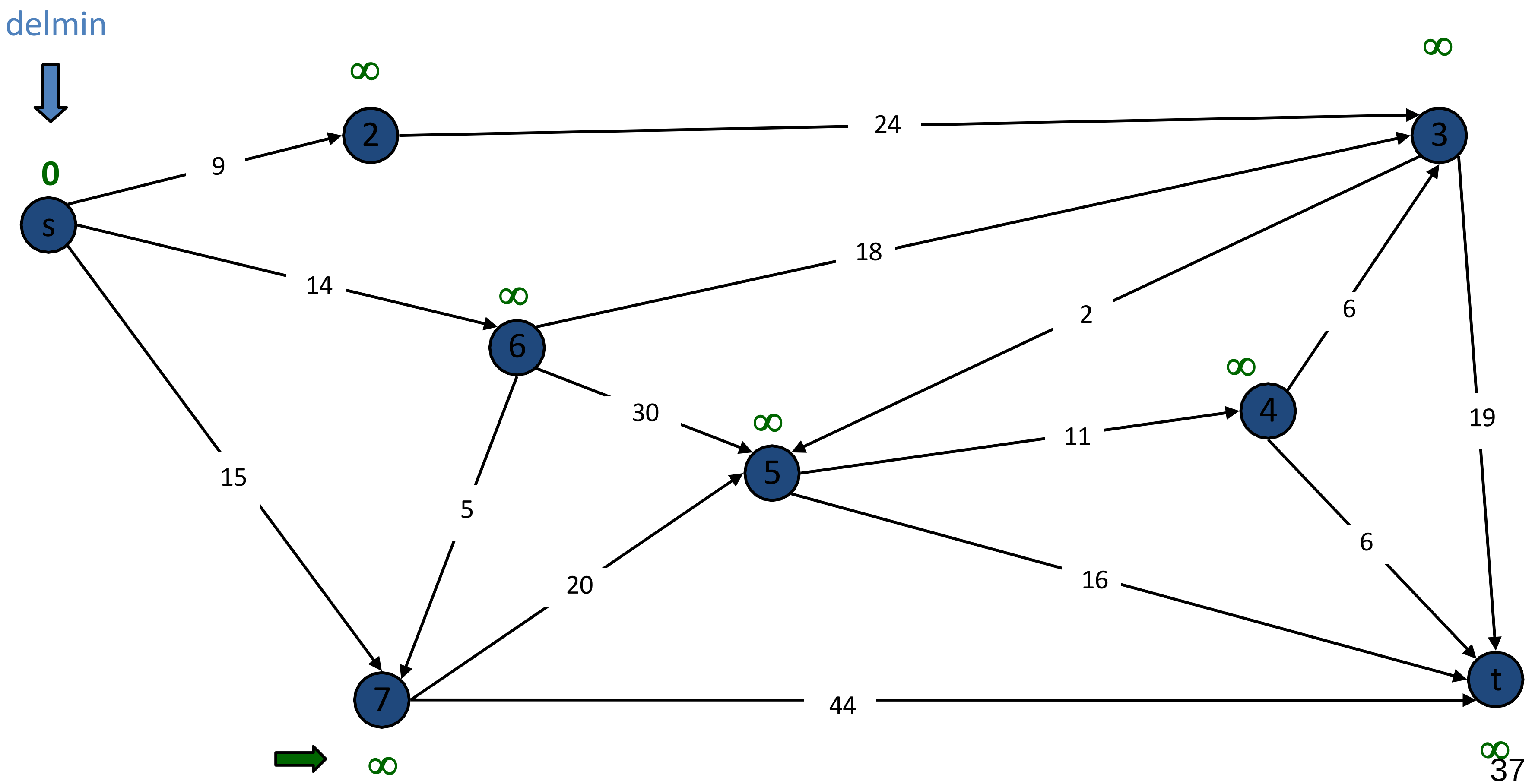
PQ = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { }

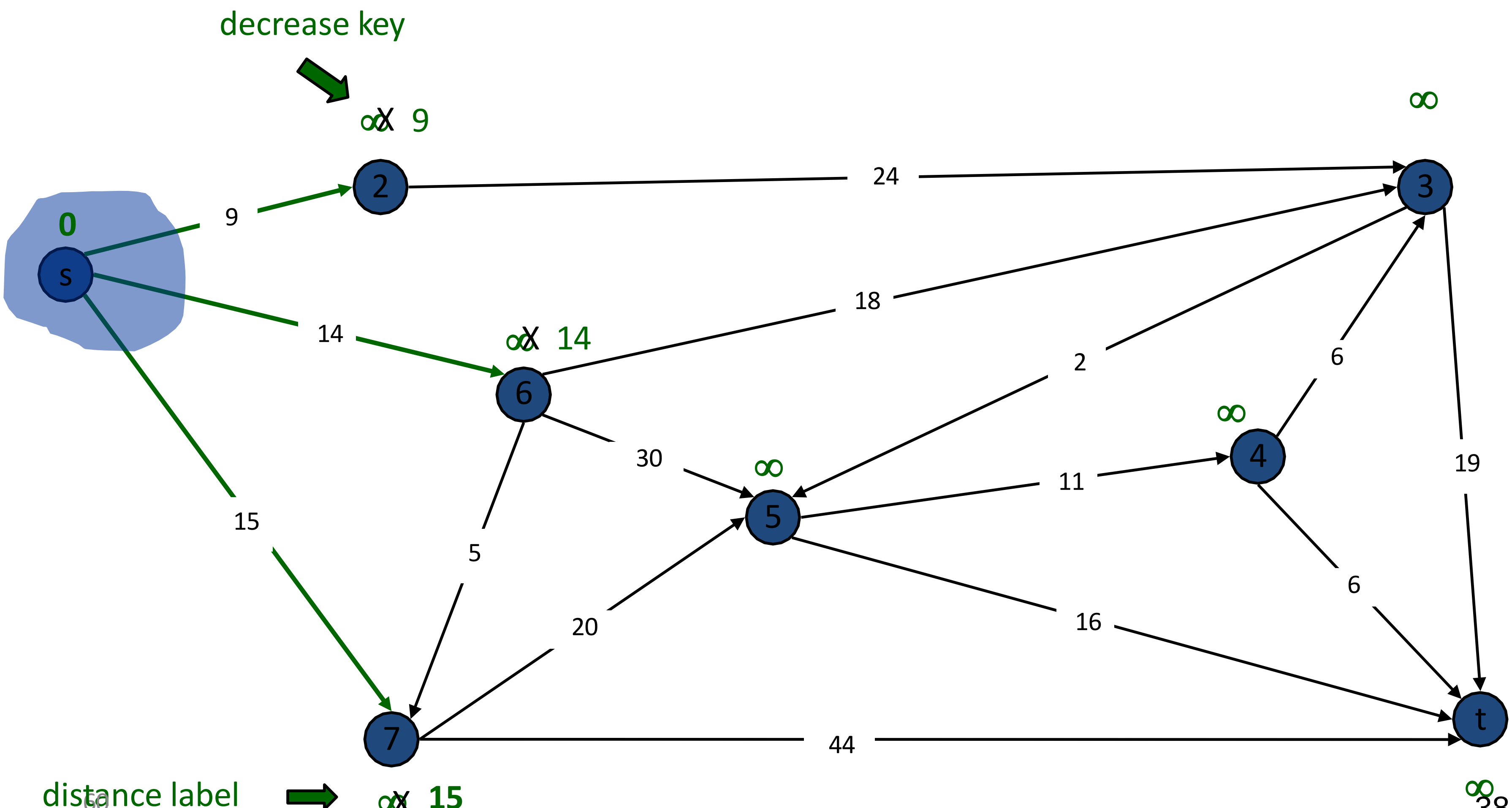
PQ = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s }

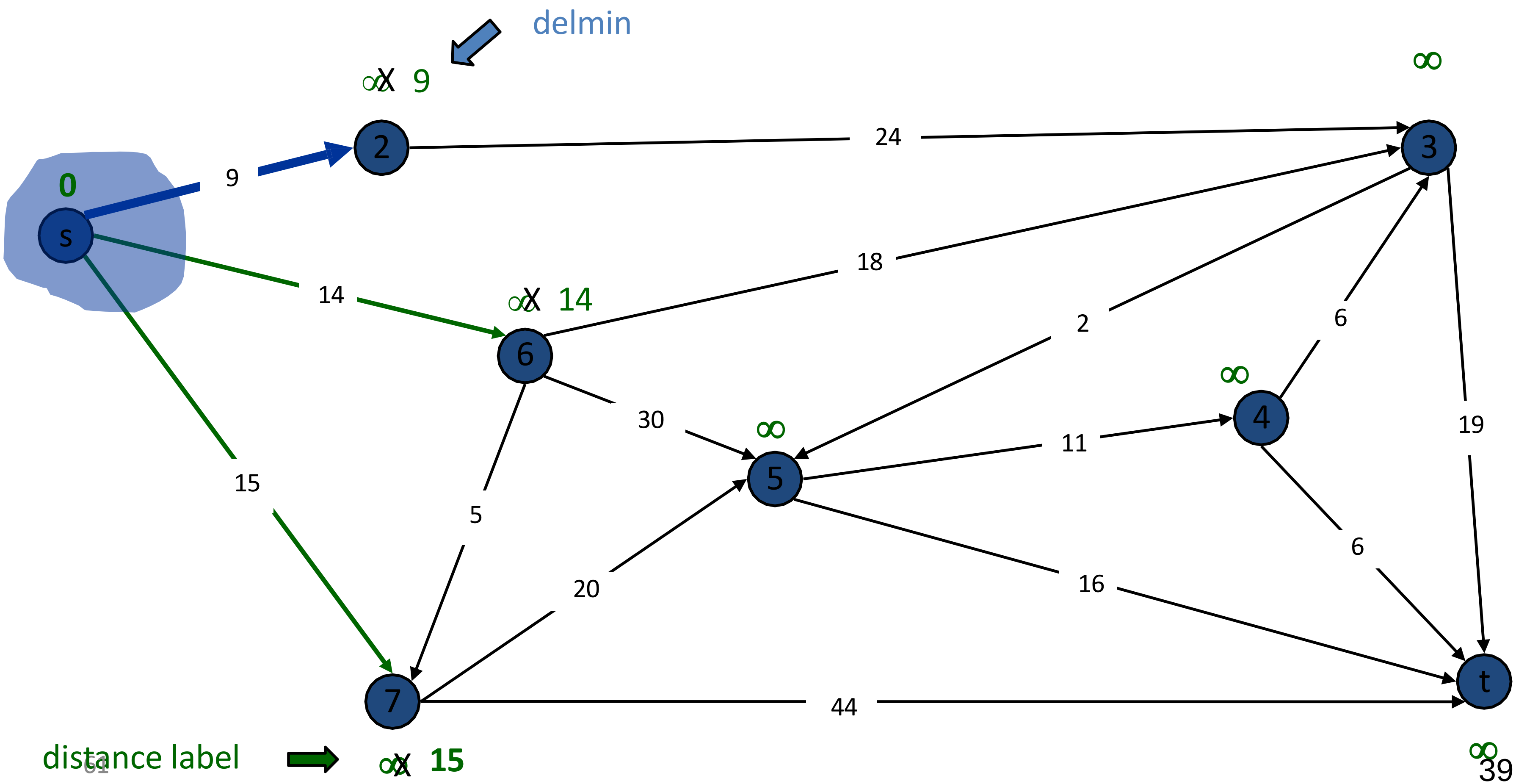
PQ = { 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s }

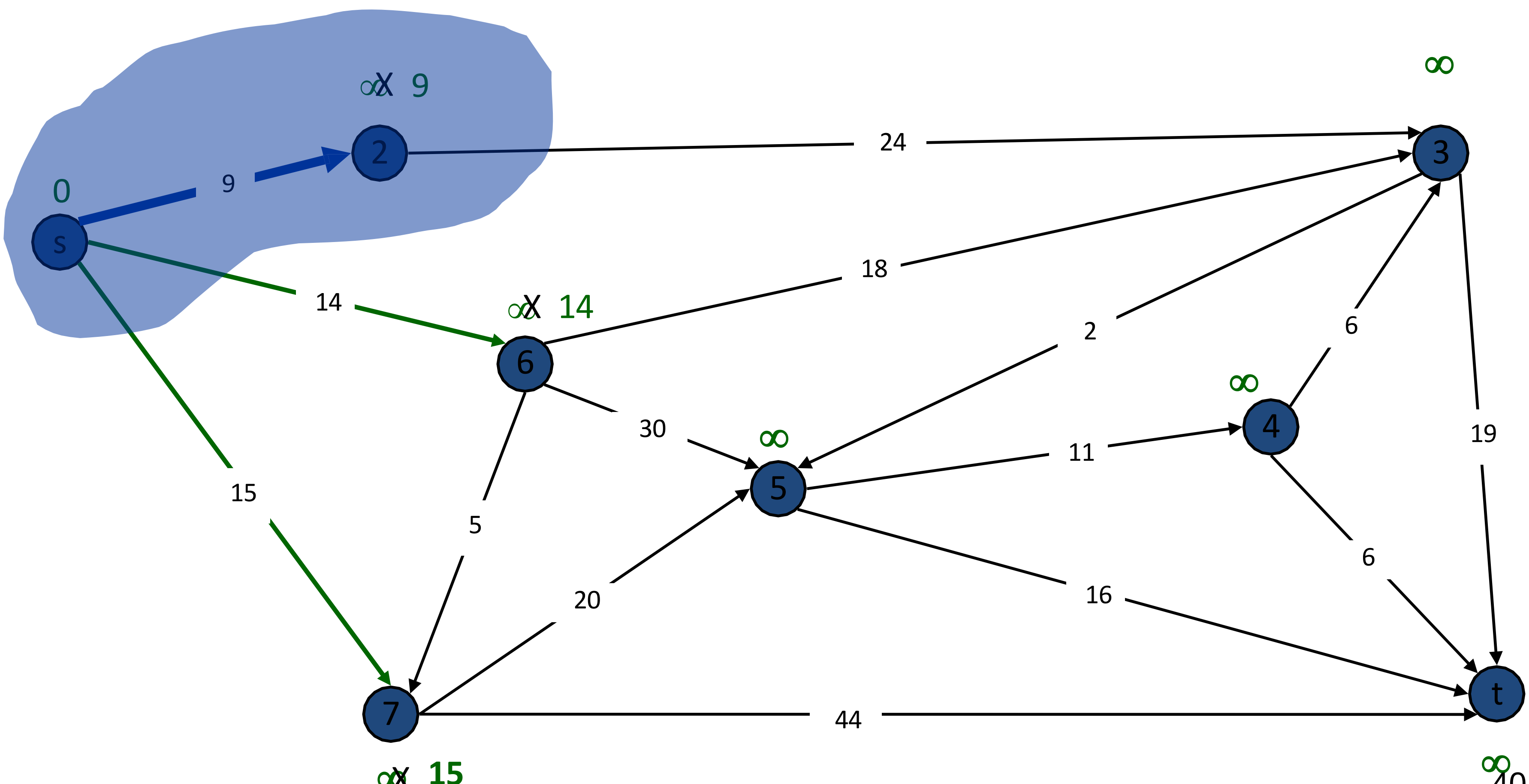
PQ = { 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

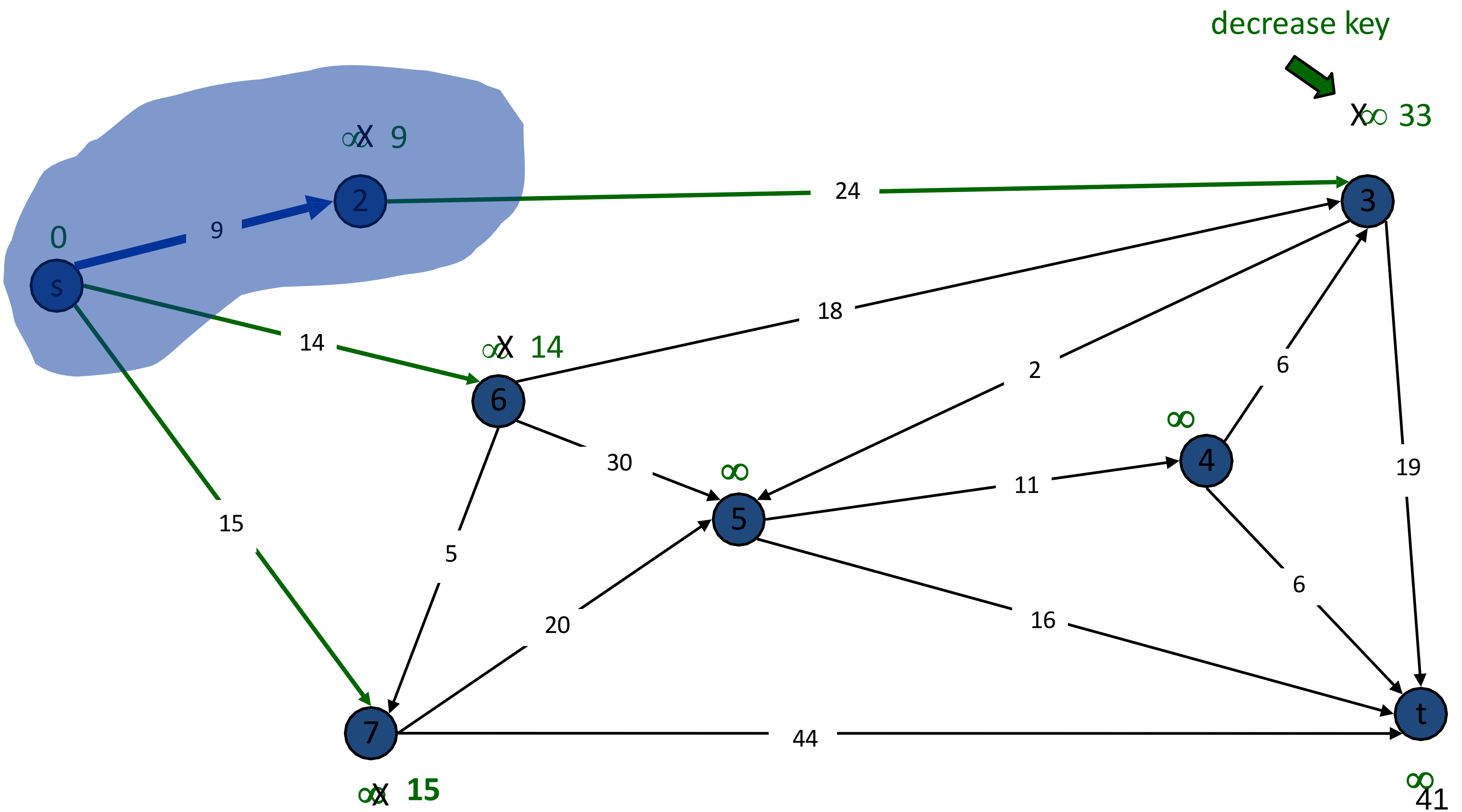
PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

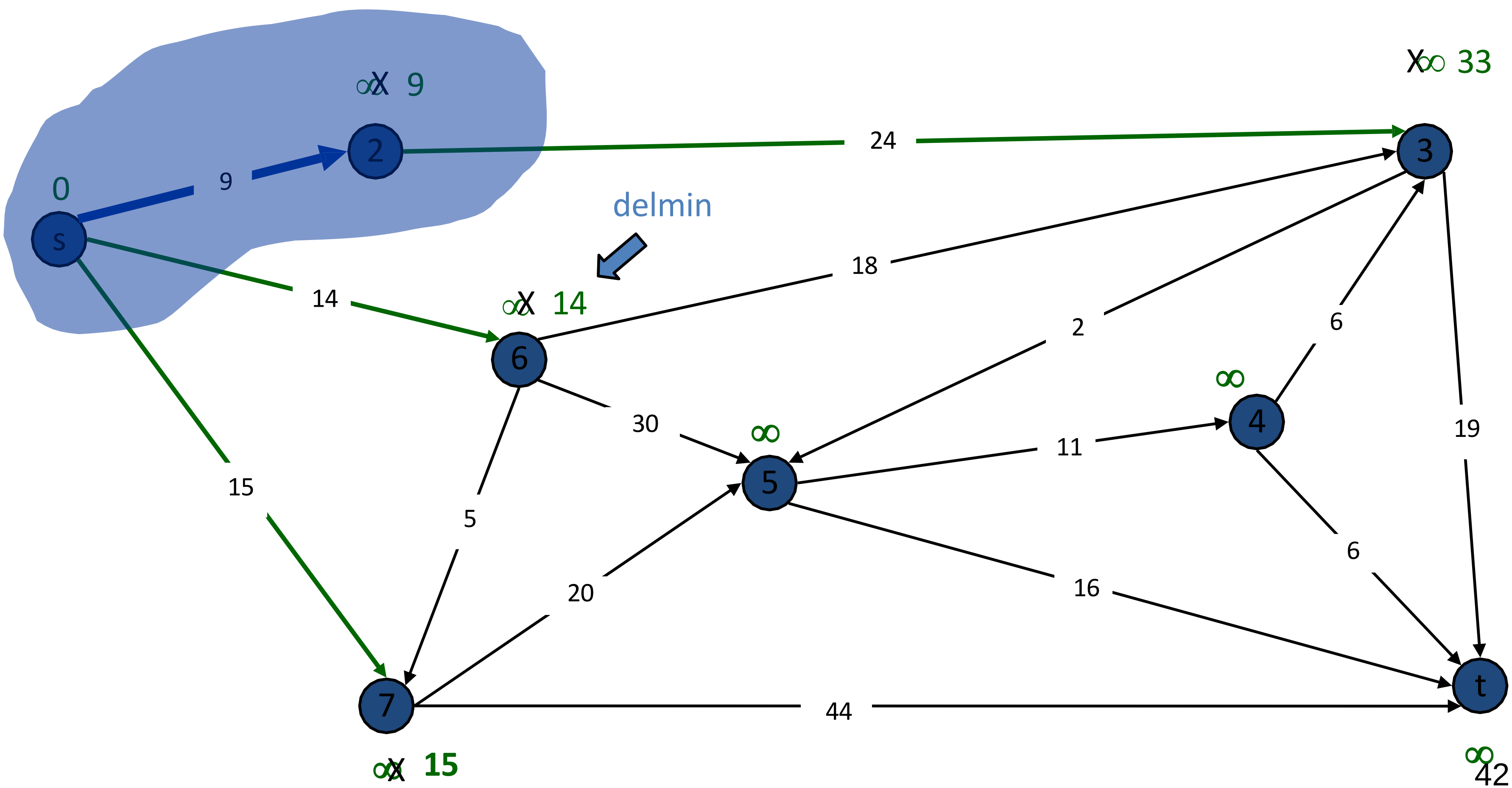
PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

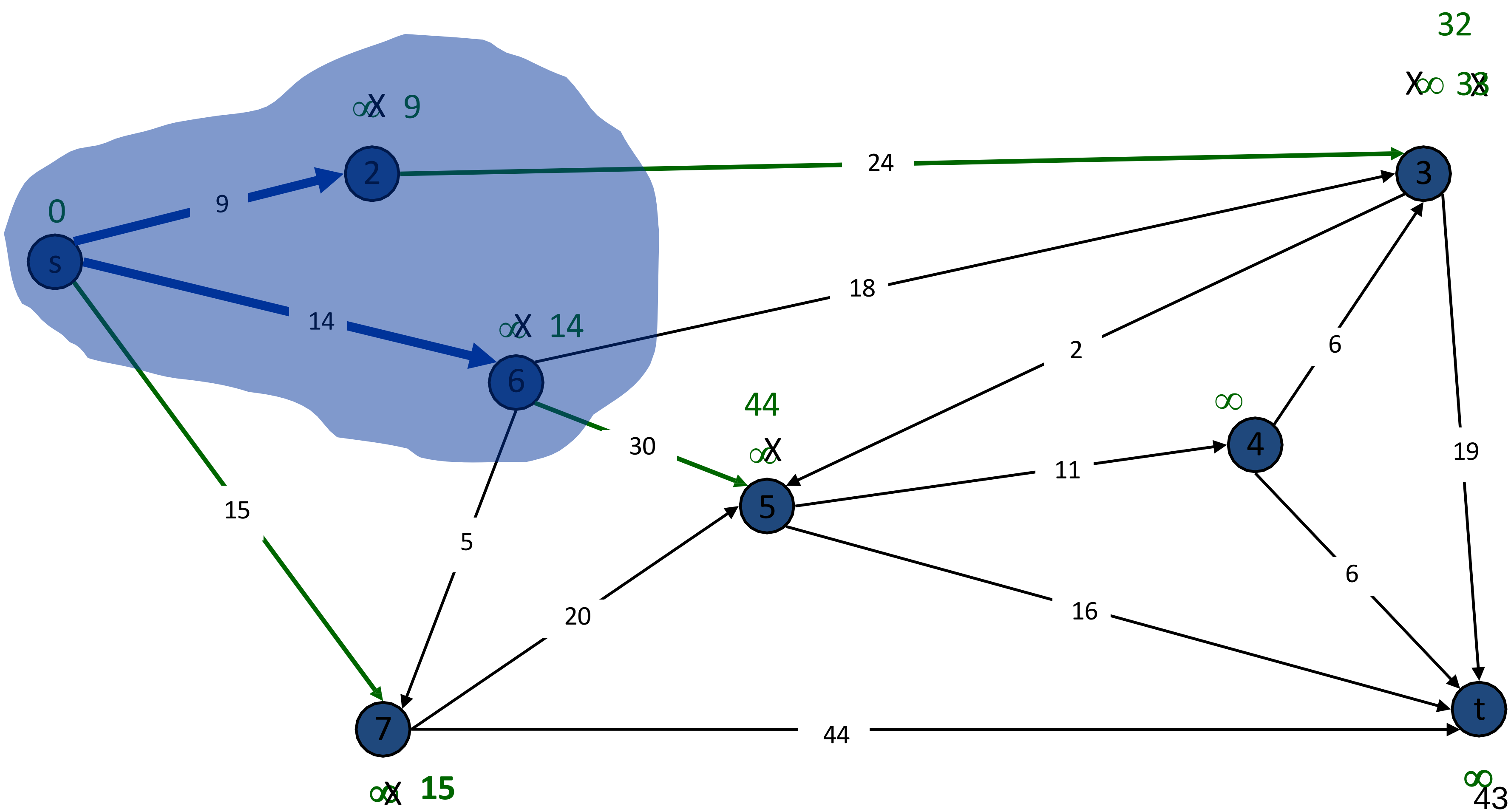
PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }

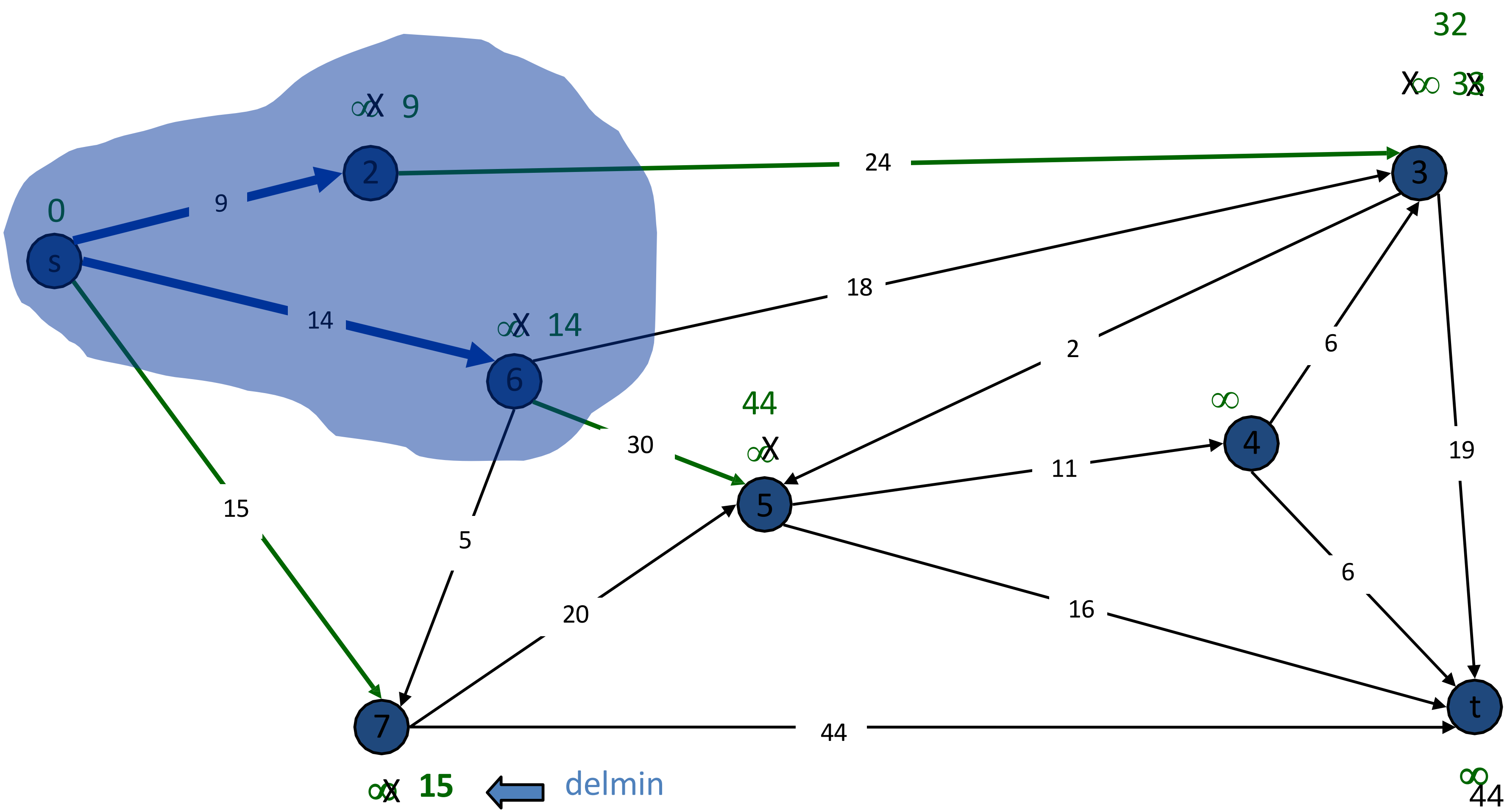
PQ = { 3, 4, 5, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }

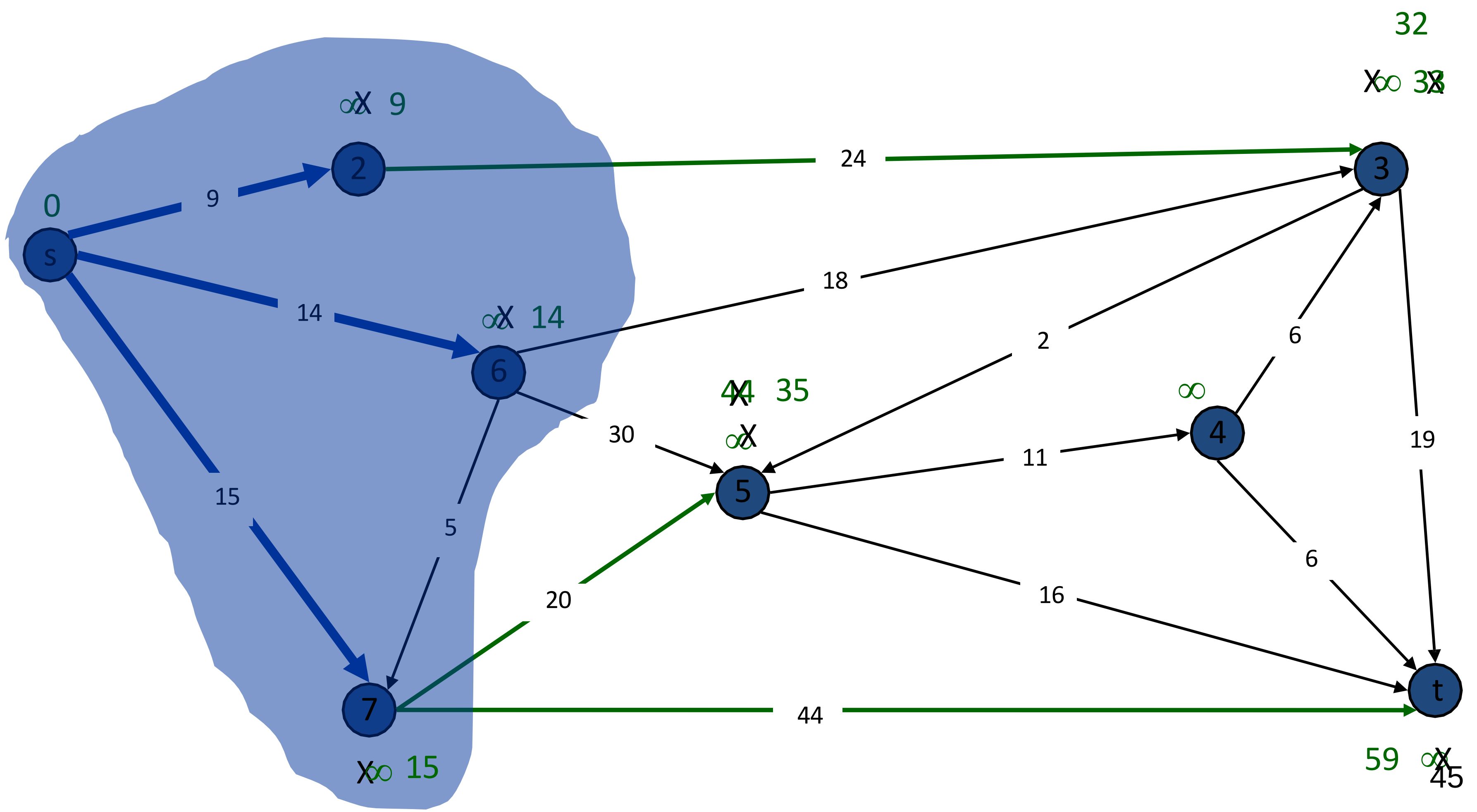
PQ = { 3, 4, 5, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }

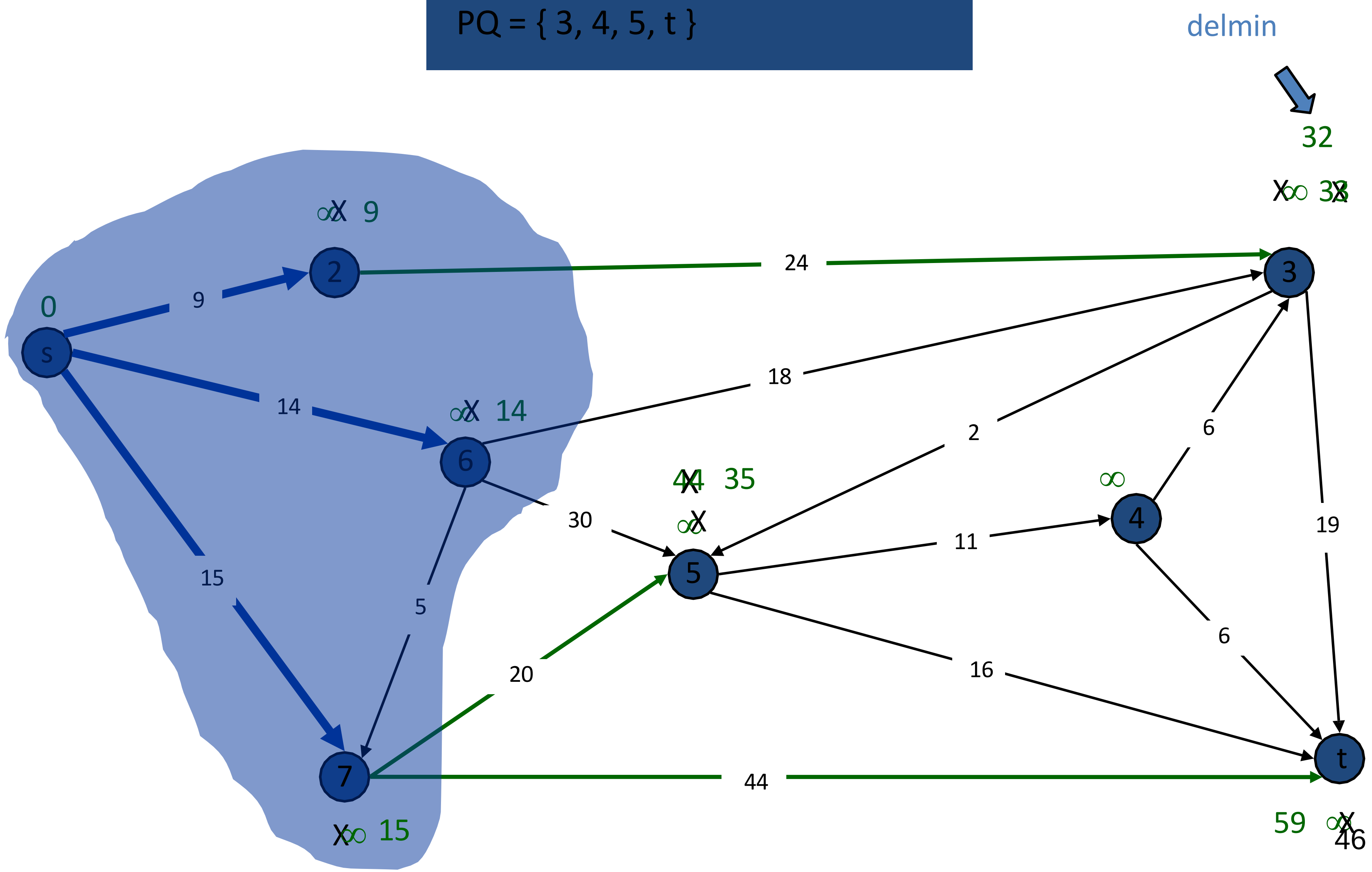
PQ = { 3, 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }

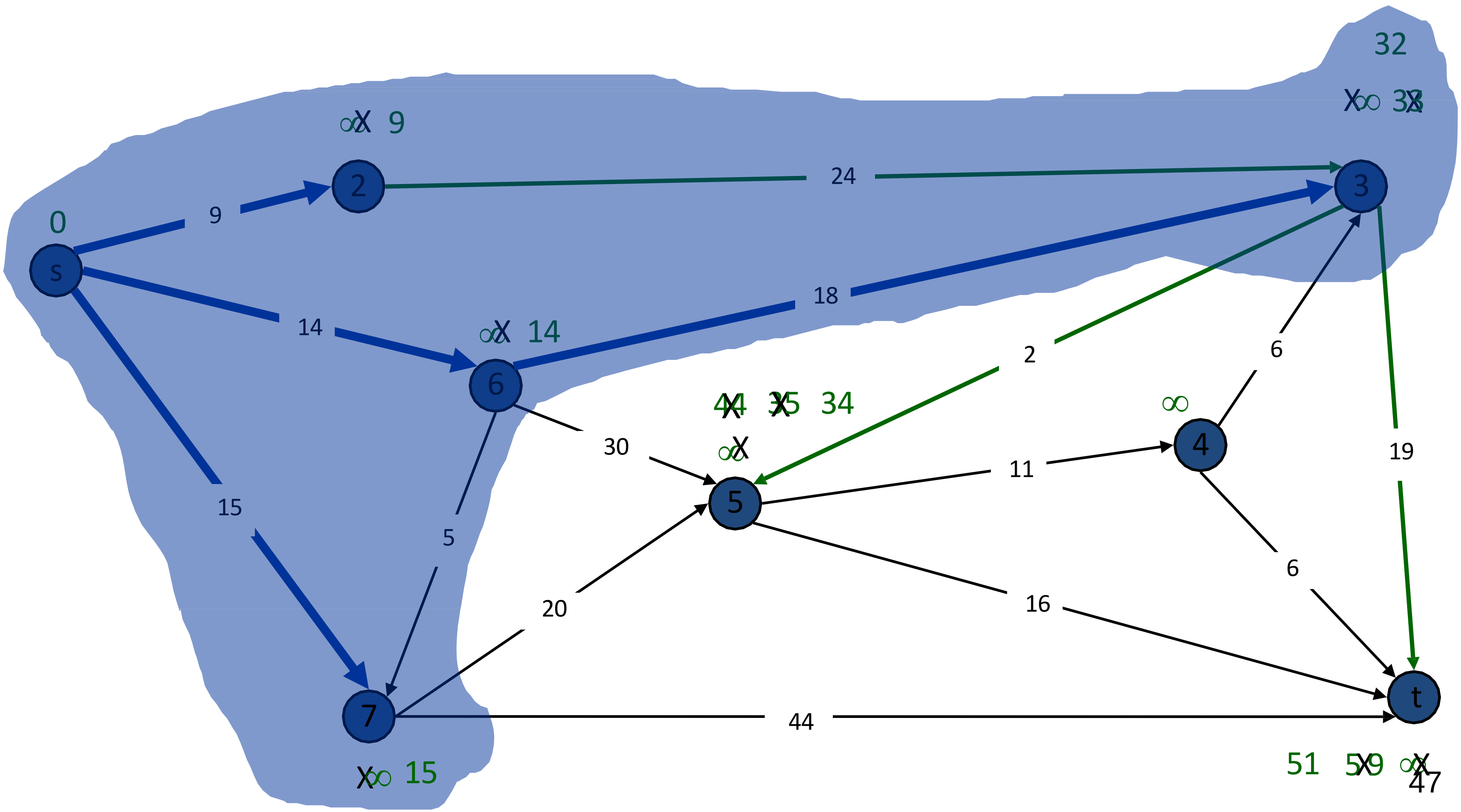
PQ = { 3, 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }

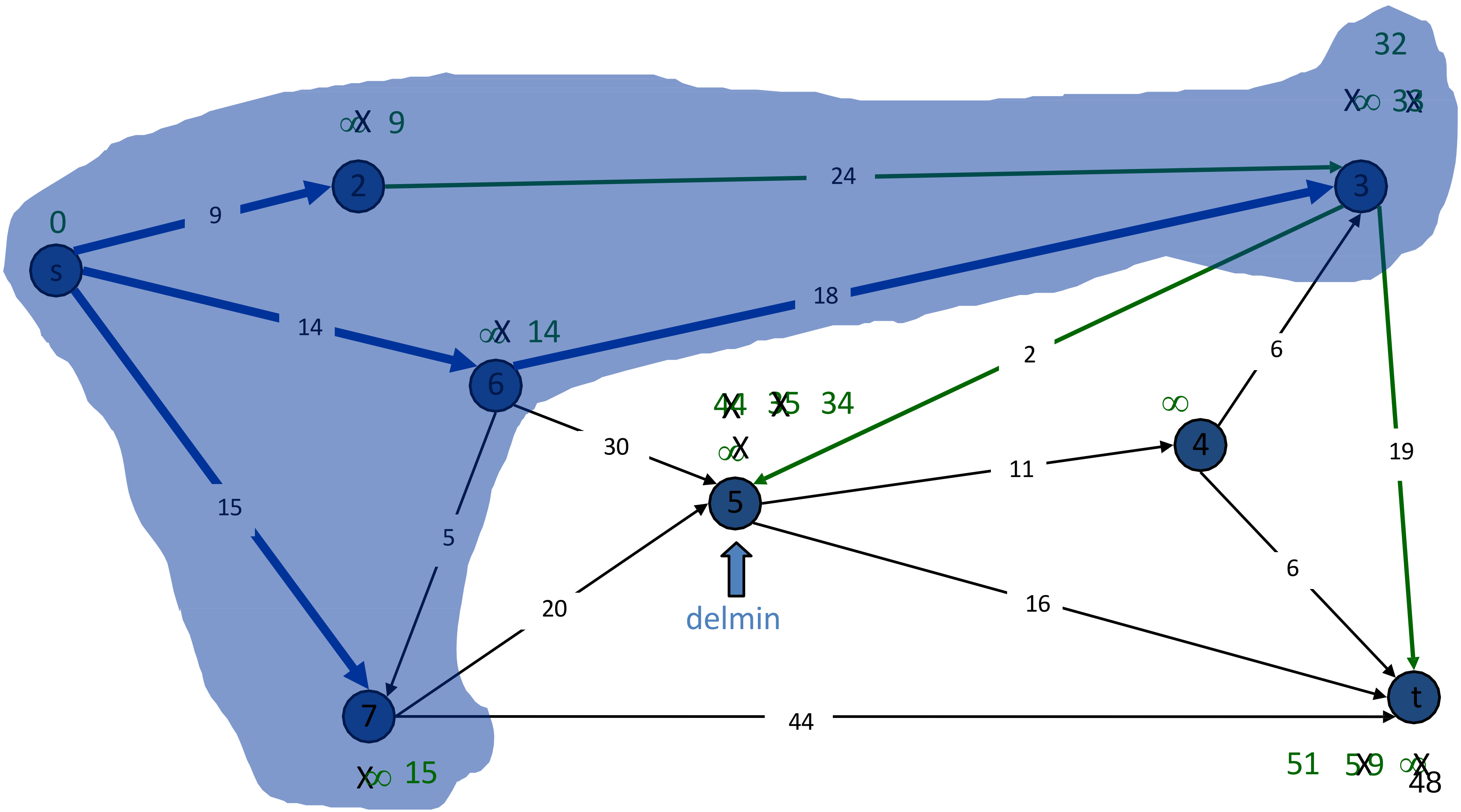
PQ = { 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }

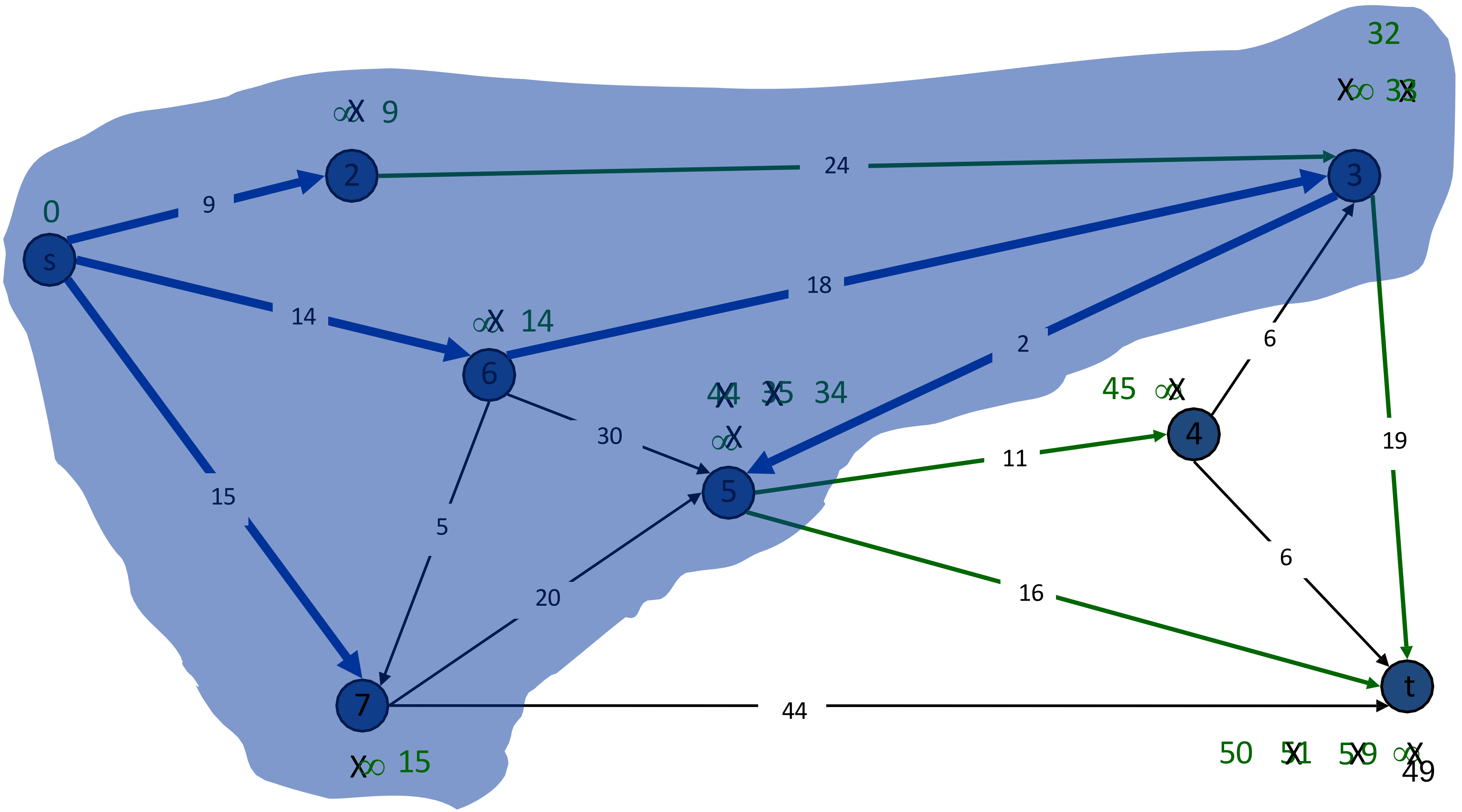
PQ = { 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }

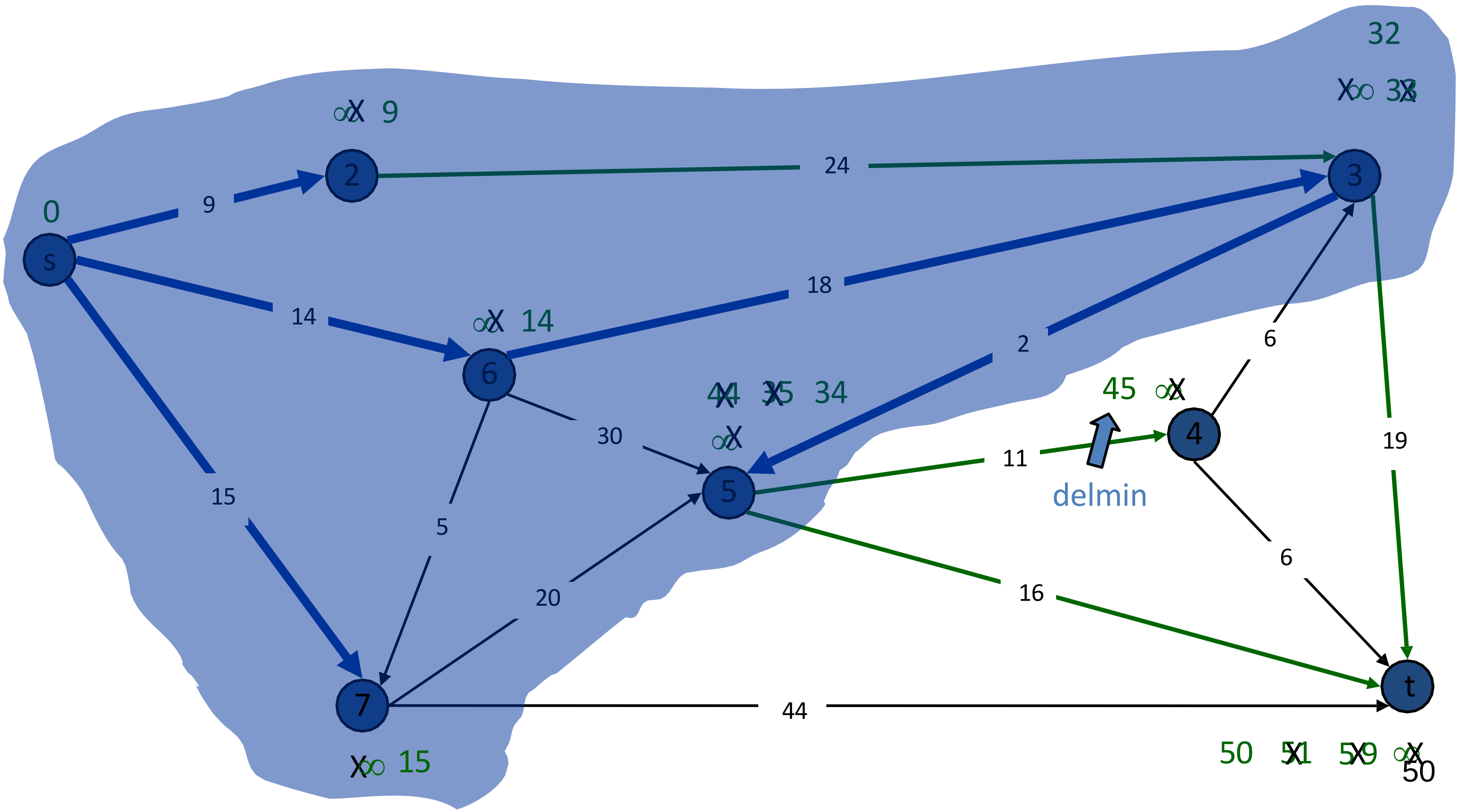
PQ = { 4, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }

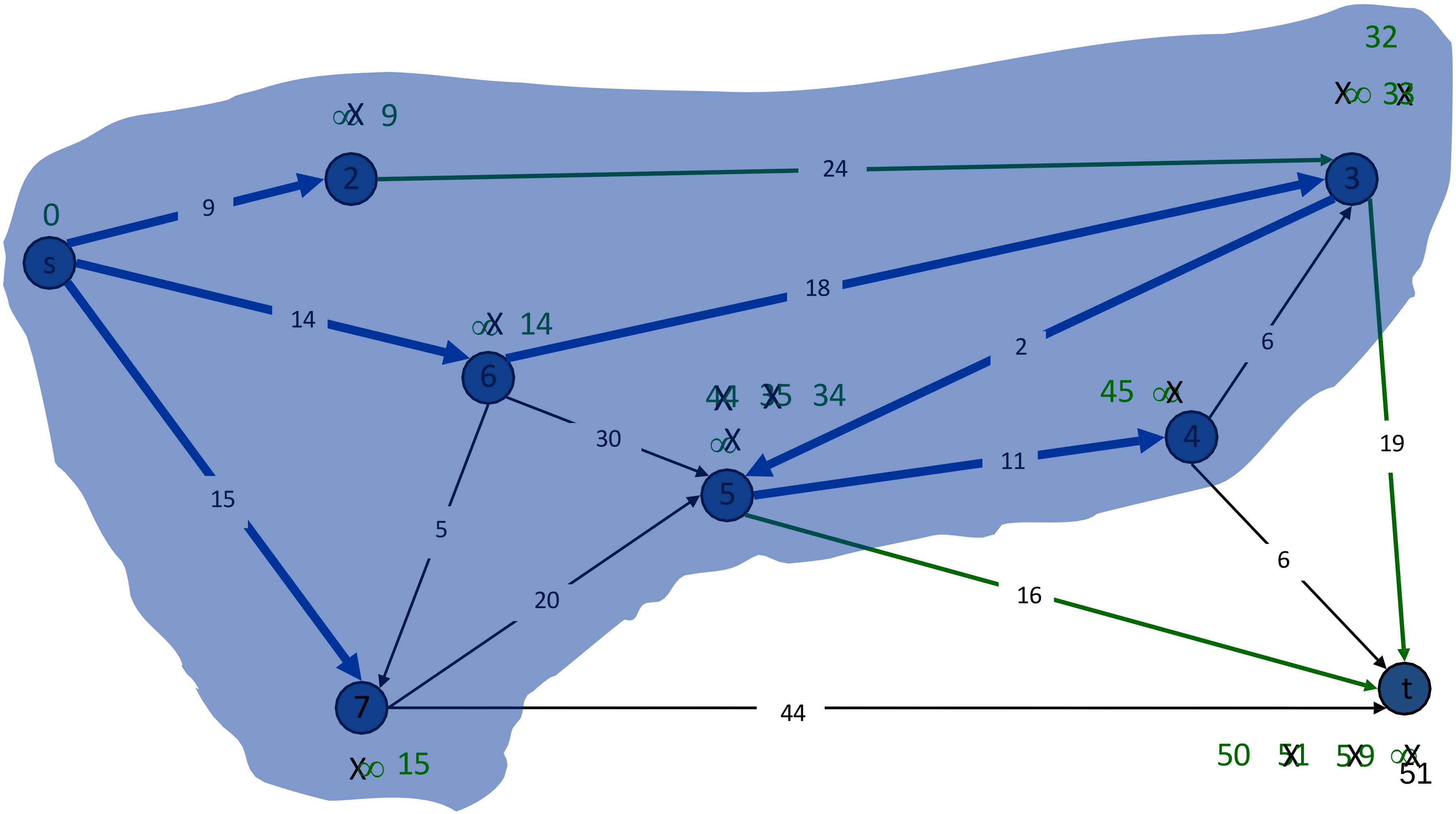
PQ = { 4, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }

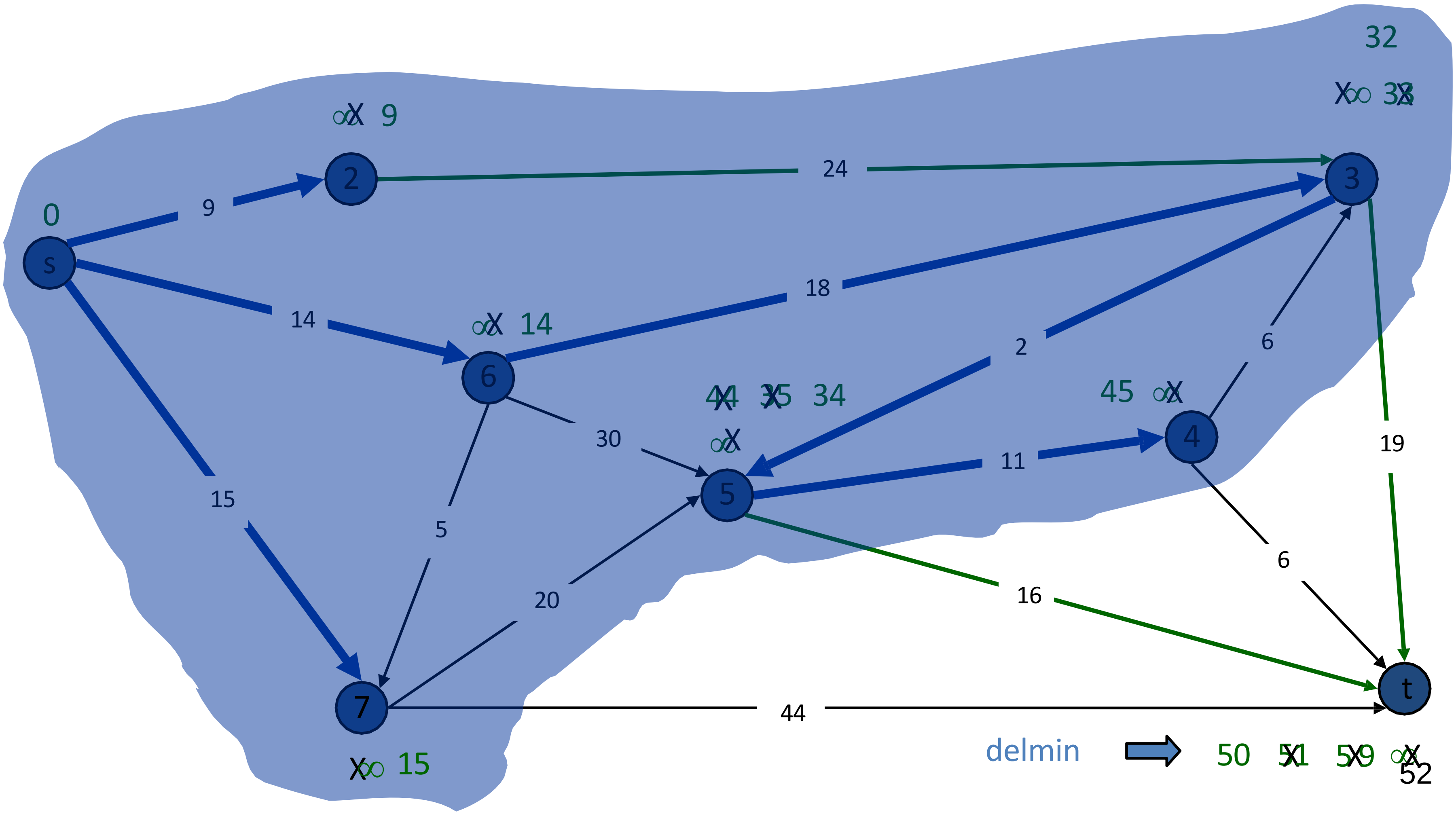
PQ = { t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }

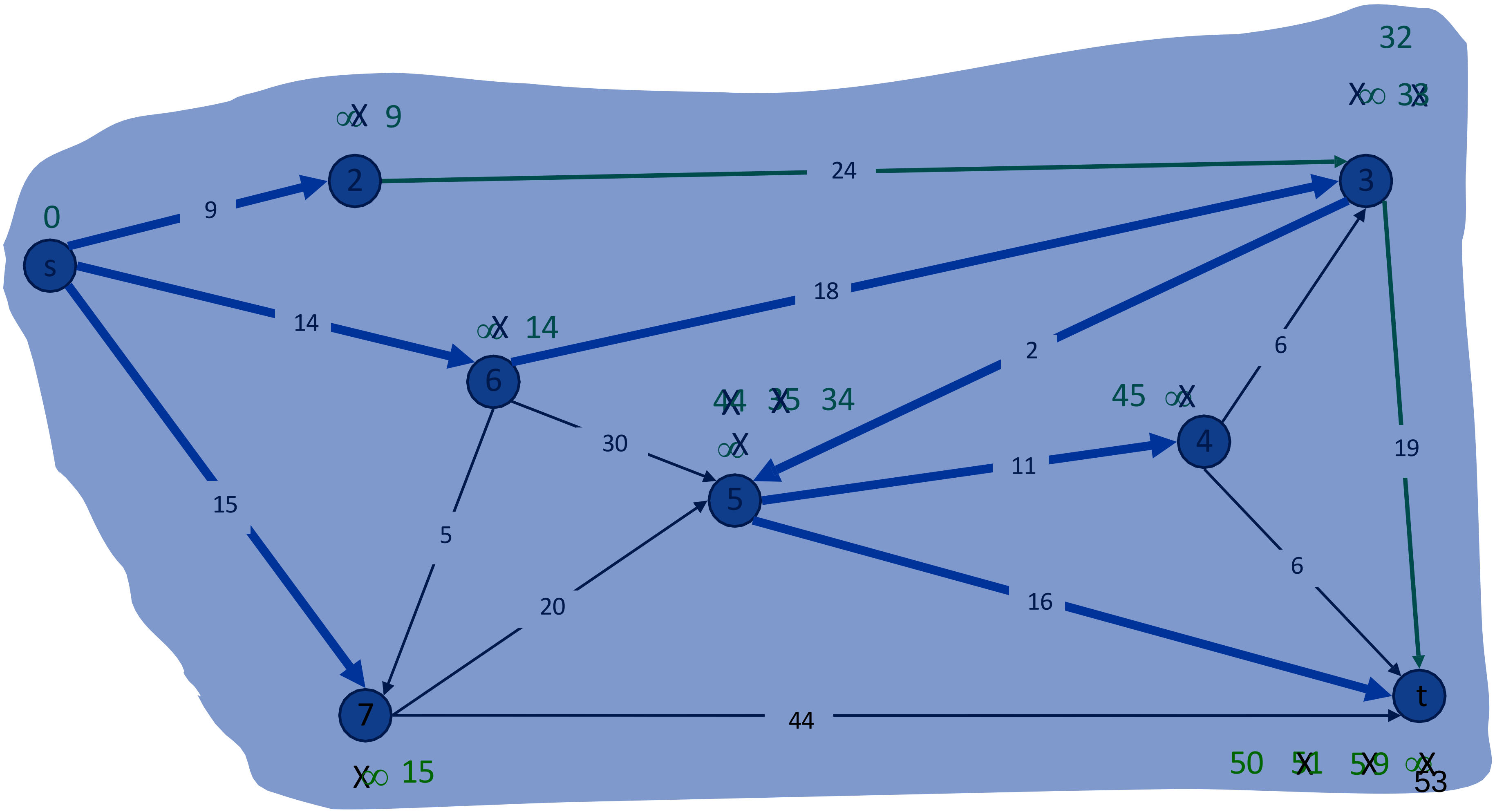
PQ = { t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }

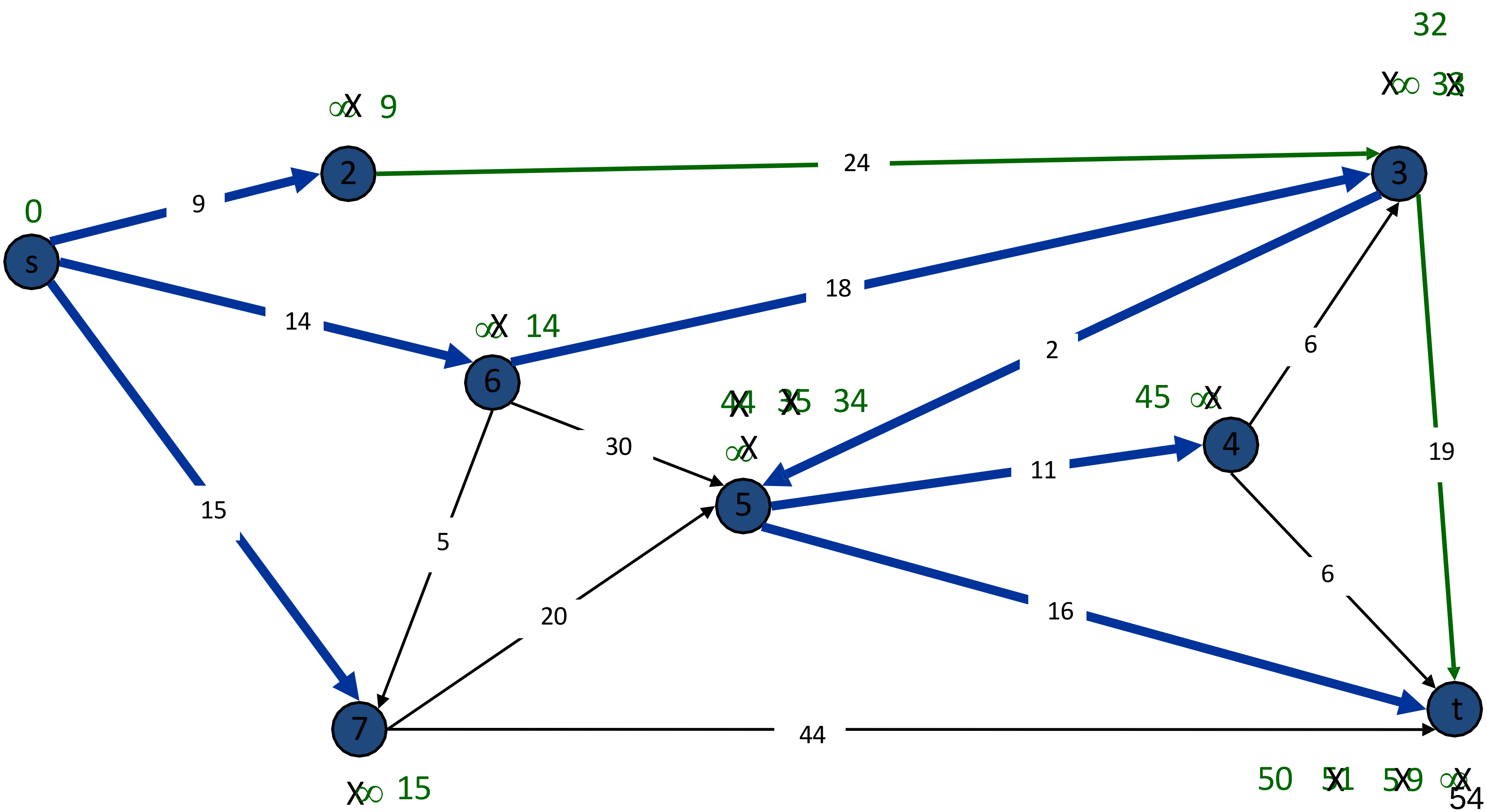
PQ = { }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }

PQ = { }



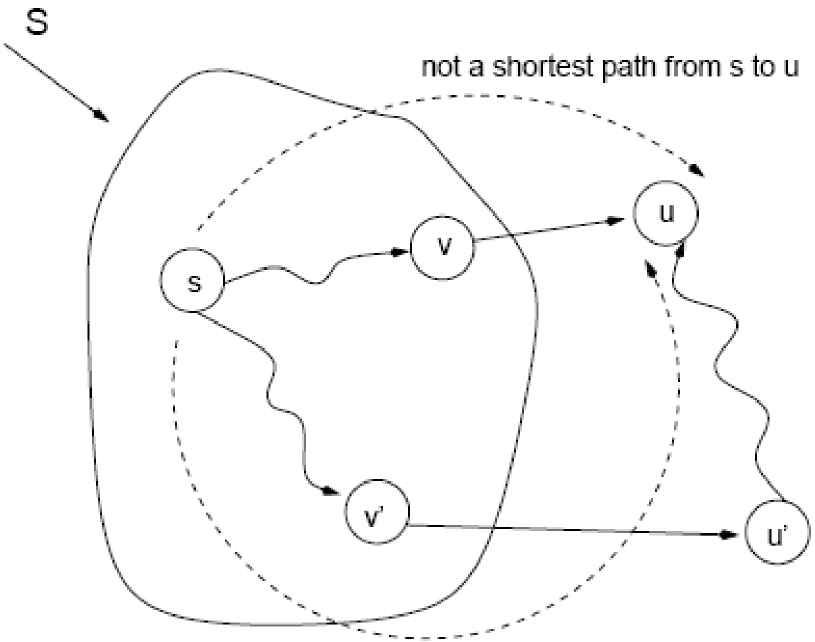
Correctness of Dijskstra's Algorithm

- For each vertex $u \in V$, we have $d[u] = \delta(s, u)$ at the time when u is added to S.

Proof:

- Let u be the first vertex for which $d[u] \neq \delta(s, u)$ when added to S
- Let's look at a true shortest path p from s to u:

Correctness of Dijskstra's Algorithm



What is the value of d[u]?

$d[u] \leq d[v] + w(v,u) = \delta(s,v) + w(v,u)$

What is the value of d[u']?

$d[u'] \leq d[v'] + w(v',u') = \delta(s,v') + w(v',u')$

Since u' is in the shortest path of u: $d[u'] < \delta(s,u)$

Using the upper bound property: $d[u] > \delta(s,u)$

$d[u'] < d[u]$

Contradiction!

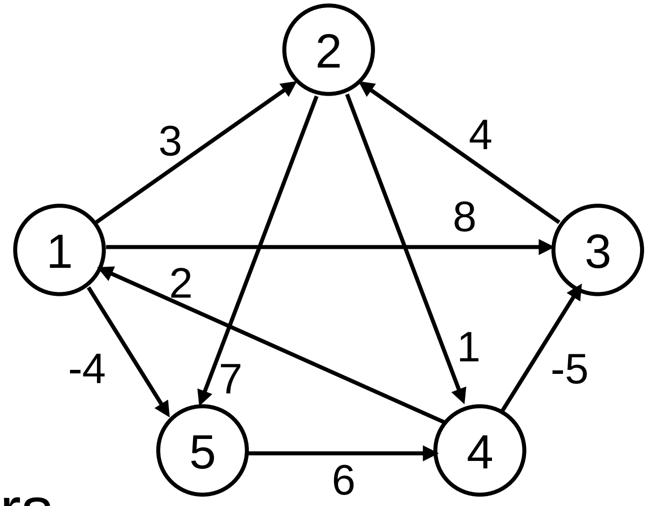
Priority Queue Q: <u, …, u', ….> (i.e., $d[u] < \ldots < d[u'] < \ldots$)

Dijskstra's Algorithm Summary

- Given a weighted directed graph, we can find the shortest distance between two vertices by:
  - starting with a trivial path containing the initial vertex
  - growing this path by always going to the next vertex which has the shortest current path

All-Pairs Shortest Paths

- **Given:**
  - Directed graph $G = (V, E)$
  - Weight function $w : E \rightarrow \mathbf{R}$
- **Compute:**
  - The shortest paths between all pairs of vertices in a graph
  - Result: an $n \times n$ matrix of shortest-path distances $\delta(u, v)$



All-Pairs Shortest Paths - Solutions

- Run **BELLMAN-FORD** once from each vertex:
  - $O(V^2E)$, which is $O(V^4)$ if the graph is dense ($E = \Theta(V^2)$)
- If no negative-weight edges, could run **Dijkstra's** algorithm once from each vertex:
  - $O(VE\lg V)$ with binary heap, $O(V^3\lg V)$ if the graph is dense
- We can solve the problem in $O(V^3)$, with no elaborate data structures

Floyd’s Algorithm

All pairs shortest path

All pairs shortest path

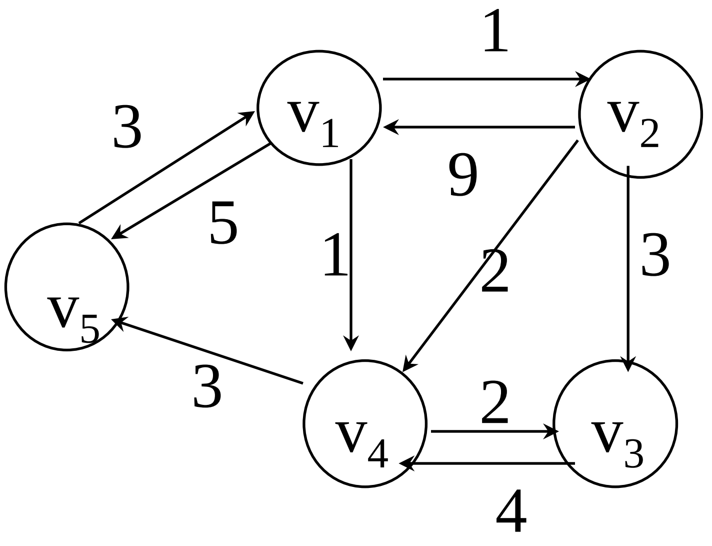
- *The problem:* find the shortest path between **every pair** of vertices of a graph

- *The graph*: **may contain negative edges** but no negative cycles.

- *A representation*: a weight matrix where
  $W(i,j)=0$ if $i=j$.
  $W(i,j)=\infty$ if there is no edge between i and j.
  $W(i,j)=$"weight of edge"
- Note: we have shown **principle of optimality** applies to shortest path problems

The weight matrix and the graph

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0



The subproblems

- How can we define the shortest distance $d_{i,j}$ in terms of "smaller" problems?

- One way is to restrict the paths to only include vertices from a restricted subset.
-
- Initially, the subset is empty.
- Then, it is incrementally increased until it includes all the vertices.

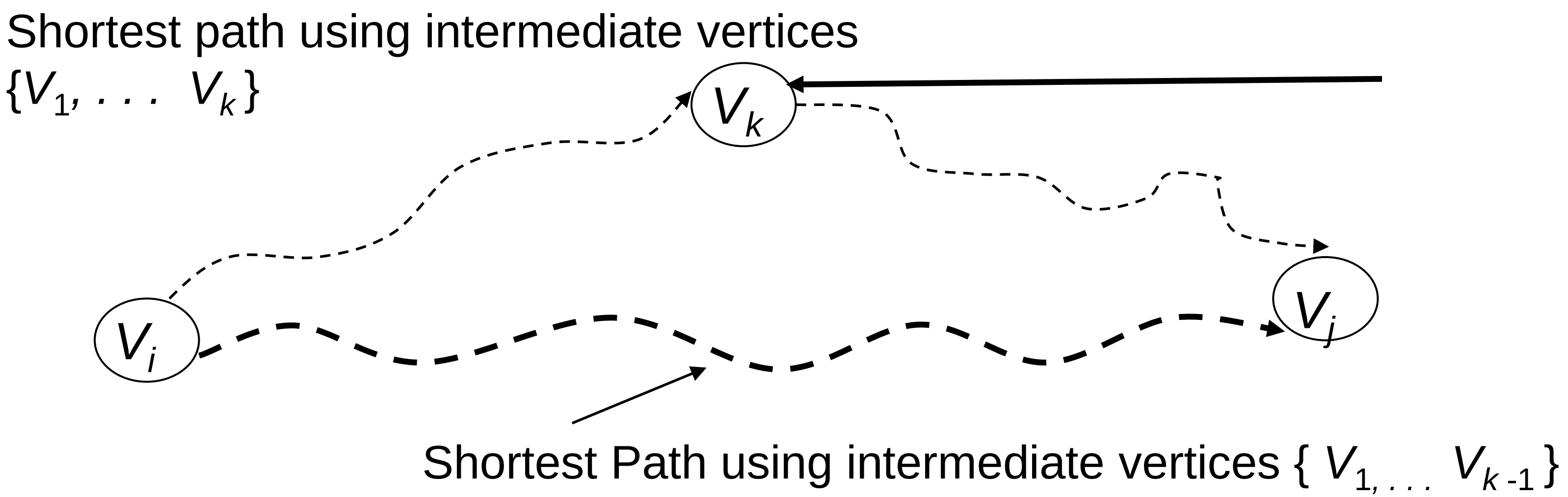
The subproblems

- Let $D^{(k)}[i,j]$=weight of a shortest path from v_i to v_j using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices in the path
  - $D^{(0)}=W$
  - $D^{(n)}=D$ which is the goal matrix
- How do we compute $D^{(k)}$ from $D^{(k-1)}$?

The Recursive Definition:

Case 1: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices does not use v_k. Then $D^{(k)}[i,j]= D^{(k-1)}[i,j]$.

Case 2: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices does use v_k. Then $D^{(k)}[i,j]= D^{(k-1)}[i,k]+ D^{(k-1)}[k,j]$.



The recursive definition

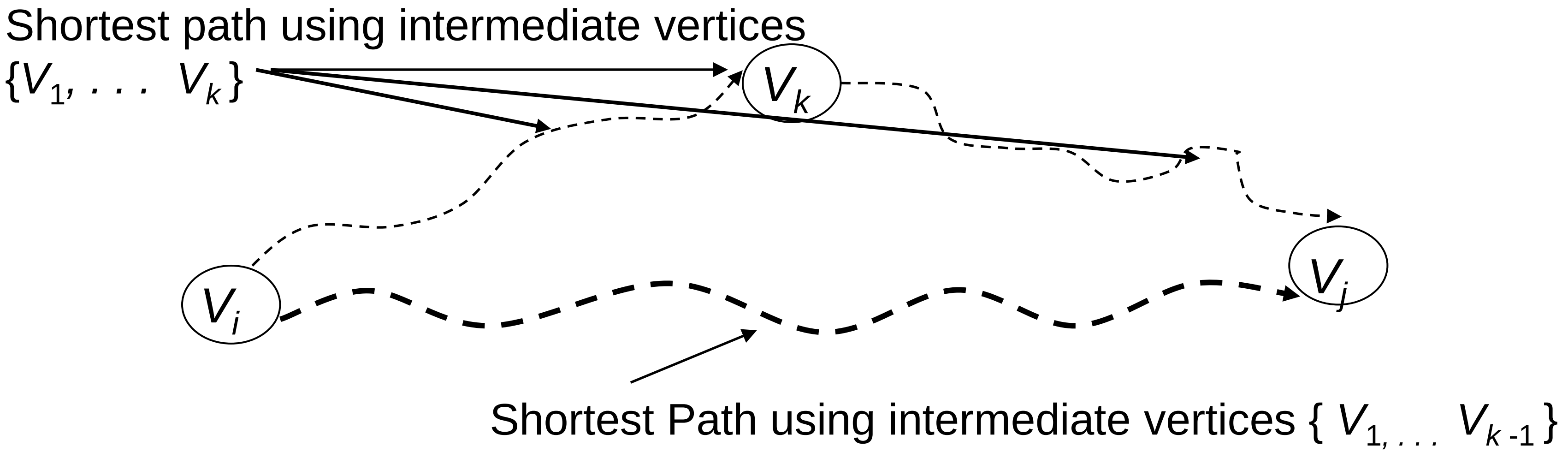
- Since

  $D^{(k)}[i,j] = D^{(k-1)}[i,j]$ or

  $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$.

  We conclude:

  $D^{(k)}[i,j] = \min\{ D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \}$.



The pointer array P

- Used to enable finding a shortest path
- Initially the array contains 0

- Each time that a shorter path from *i* to *j* is found the *k* that provided the minimum is saved (highest index node on the path from *i* to *j*)

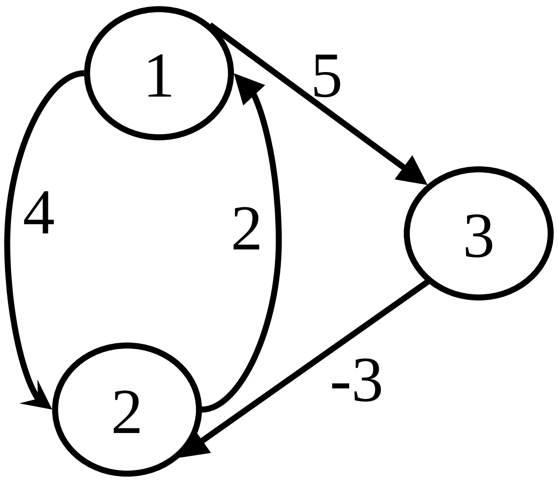
- To print the intermediate nodes on the shortest path a recursive procedure that print the shortest paths from *i* and *k*, and from *k* to *j* can be used

Floyd's Algorithm Using n+1 *D* matrices

Floyd//Computes shortest distance between all pairs of
//nodes, and saves P to enable finding shortest paths

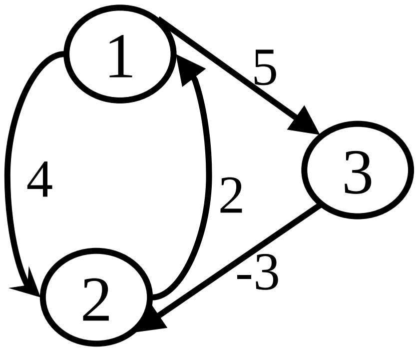
1. $D^0 \leftarrow W$ // initialize *D* array to *W* [ ]
2. $P \leftarrow 0$ // initialize P array to [0]
3. **for** $k \leftarrow 1$ **to** n
4. **do for** $i \leftarrow 1$ **to** n
5. **do for** $j \leftarrow 1$ **to** n
6. **if** $(D^{k-1}[i, j] > D^{k-1}[i, k] + D^{k-1}[k, j])$
7. **then** $D^k[i, j] \leftarrow D^{k-1}[i, k] + D^{k-1}[k, j]$
8. $P[i, j] \leftarrow k$;
9. **else** $D^k[i, j] \leftarrow D^{k-1}[i, j]$

Example



$$W = D^0 = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & \infty \\ 3 & \infty & -3 & 0 \\ \hline \end{array}$$

$$P = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ \hline \end{array}$$



$$D^0 = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & \infty \\ 3 & \infty & -3 & 0 \\ \hline \end{array}$$

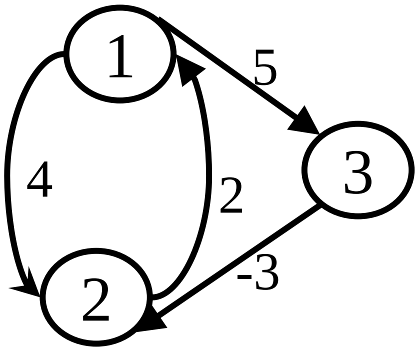
k = 1
Vertex 1 can be intermediate node

$$D^1 = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & \infty & -3 & 0 \\ \hline \end{array}$$

$D^1[2,3] = \min(D^0[2,3], D^0[2,1]+D^0[1,3])$
$= \min (\propto, 7)$
$= 7$

$$P = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 \\ \hline \end{array}$$

$D^1[3,2] = \min(D^0[3,2], D^0[3,1]+D^0[1,2])$
$= \min (-3,\propto)$
$= -3$



$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

k = 2
Vertices 1, 2 can be intermediate

$D^2 =$

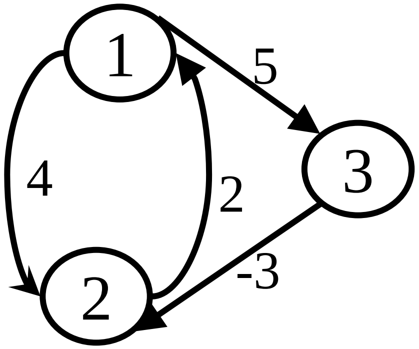
	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

$D^2[1,3] = \min(D^1[1,3], D^1[1,2]+D^1[2,3])$
$= \min (5, 4+7)$
$= 5$

P =

	1	2	3
1	0	0	0
2	0	0	1
3	2	0	0

$D^2[3,1] = \min(D^1[3,1], D^1[3,2]+D^1[2,1])$
$= \min (\propto, -3+2)$
$= -1$



$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

k = 3
Vertices 1, 2, 3 can be intermediate

$D^3 =$

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

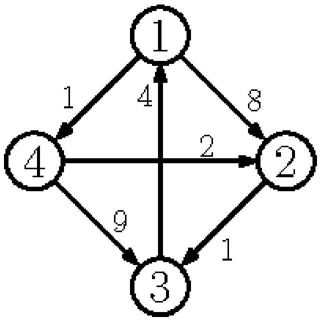
$D^3[1,2] = \min(D^2[1,2], D^2[1,3]+D^2[3,2])$
$= \min(4, 5+(-3))$
$= 2$

P =

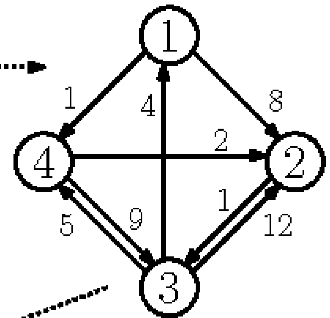
	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

$D^3[2,1] = \min(D^2[2,1], D^2[2,3]+D^2[3,1])$
$= \min(2, 7+(-1))$
$= 2$

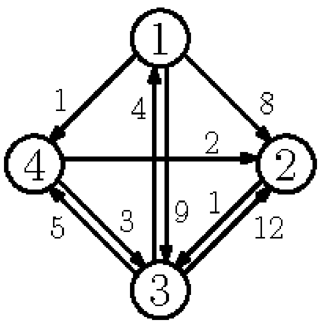
Floyd algorithm example



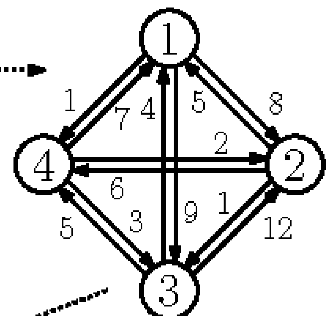
$$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



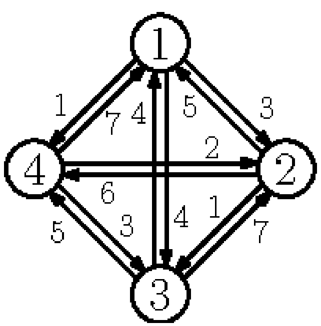
$$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



$$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$



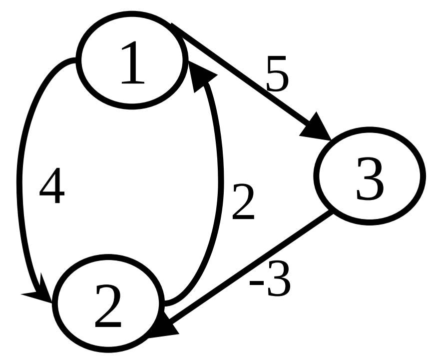
$$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

$$\text{final} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Printing intermediate nodes on shortest path from q to r

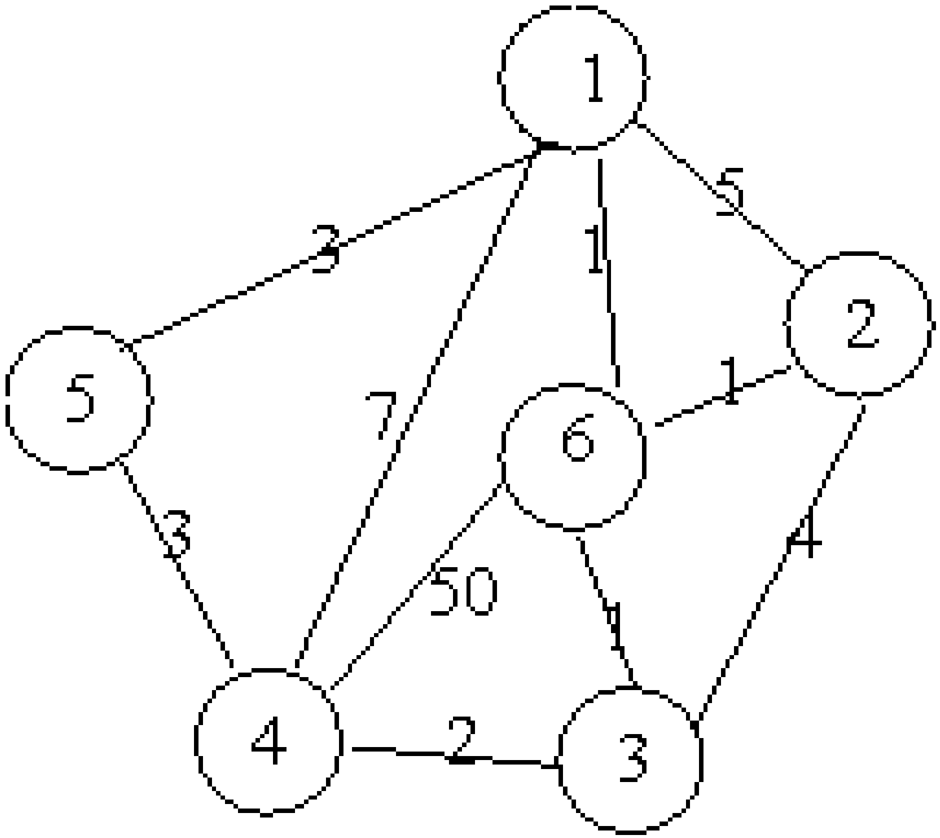
```
path(index q, r)
    if (P[ q, r ]!=0)
          path(q, P[q, r])
          println( "v"+ P[q, r])
          path(P[q, r], r)
         return;
    //no intermediate nodes
    else return
```

$$P = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 3 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline 3 & 2 & 0 & 0 \\ \hline \end{array}$$



Before calling path check D[q, r] < ∞, and print node q, after the call to path print node r

Example

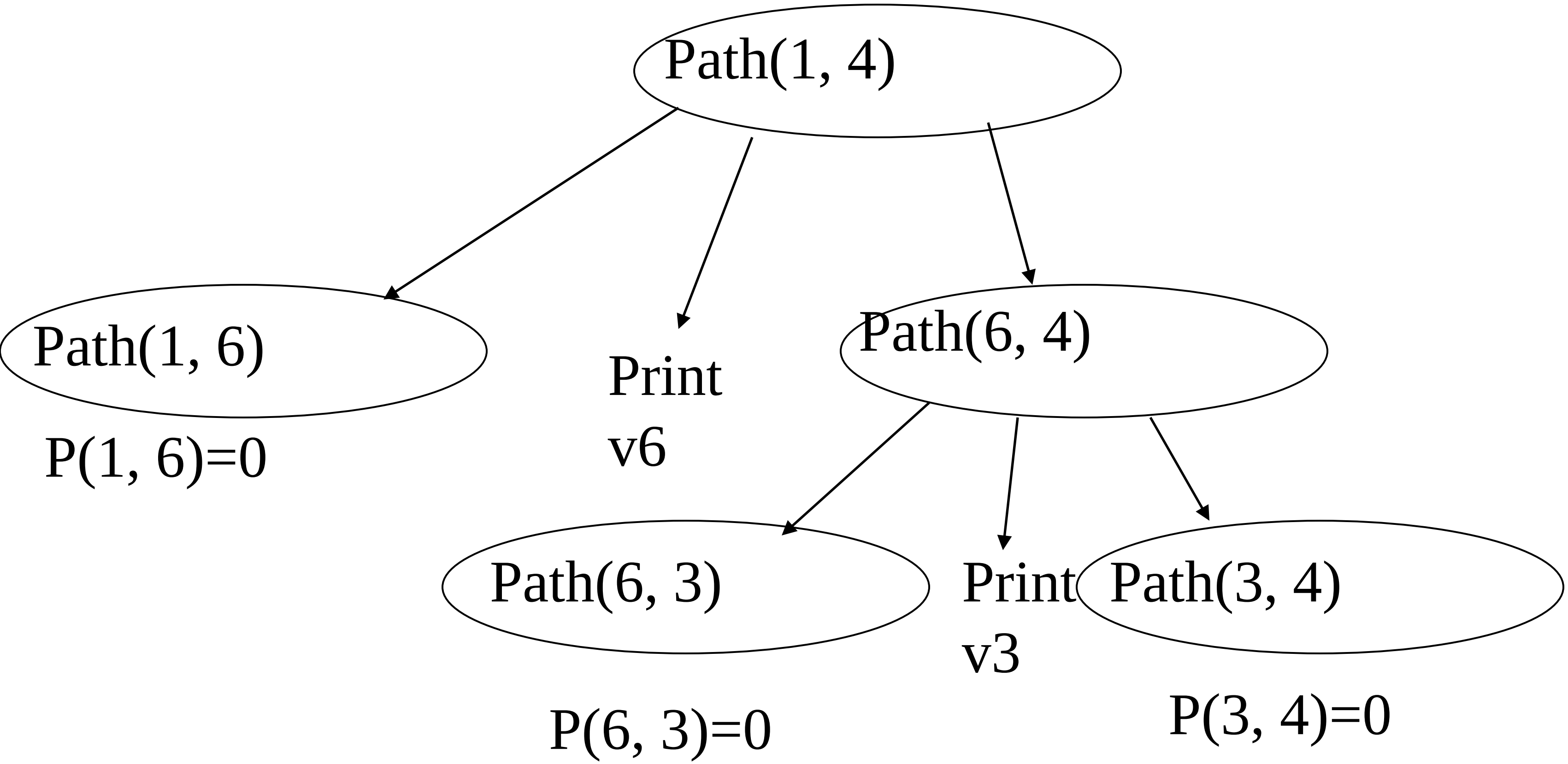


The final distance matrix and P

$$D^6 = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 2(6) & 2(6) & 4(6) & 3 & 1 \\ 2 & 2(6) & 0 & 2(6) & 4(6) & 5(6) & 1 \\ 3 & 2(6) & 2(6) & 0 & 2 & 5(4) & 1 \\ 4 & 4(6) & 4(6) & 2 & 0 & 3 & 3(3) \\ 5 & 3 & 5(6) & 5(4) & 3 & 0 & 4(1) \\ 6 & 1 & 1 & 1 & 3(3) & 4(1) & 0 \end{array}$$

The values in parenthesis are the non zero P values.

The call tree for Path(1, 4)



The intermediate nodes on the shortest path from 1 to 4 are v6, v3.
The shortest path is v1, v6, v3, v4.

Floyd's Algorithm: Using Tow D matrices

Floyd

1. ***D*** $\leftarrow$ ***W*** // initialize *D* array to *W* [ ]

2. *P* $\leftarrow$ 0 // initialize P array to [0]

3. **for *k*** $\leftarrow$ **1 to *n***

// Computing D' from D

4. do for *i* $\leftarrow$ **1 to *n***

5. do for *j* $\leftarrow$ **1 to *n***

6. if (*D*[*i*, *j*] > *D*[*i*, *k*] + *D*[*k*, *j*])

7. then *D'*[*i*, *j*] $\leftarrow$ ***D*[*i*, *k*] + *D*[*k*, *j*]**

***P*[*i*, *j*]** $\leftarrow$ ***k*;**

9. else *D'*[*i*, *j*] $\leftarrow$ ***D*[*i*, *j*]**

10. *Move D' to D.*

Can we use only one D matrix?

- *D*[*i*, *j*] depends only on elements in the *k*th column and row of the distance matrix.
- We will show that the *k*th row and the *k*th column of the distance matrix are unchanged when D^k is computed
- This means *D* can be calculated *in-place*

The main diagonal values

- **Before we show that *k*th row and column of *D* remain unchanged we show that the main diagonal remains 0**

- $D^{(k)}[\,j,j\,] = \min\{\,D^{(k-1)}[\,j,j\,],\ \ D^{(k-1)}[\,j,k\,] + D^{(k-1)}[\,k,j\,]\,\}$

  $= \min\{\,0,\ \ D^{(k-1)}[\,j,k\,] + D^{(k-1)}[\,k,j\,]\,\}$

  $= 0$

- **Based on which assumption?**

The *k*th column

- ***k*th column of D^k is equal to the *k*th column of D^{k-1}**

- ***Intuitively true* - a path from i to k will not become shorter by adding k to the allowed subset of intermediate vertices**

- **For all i, $D^{(k)}[i,k] =$**

$$= \min\{ D^{(k-1)}[i,k],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,k] \}$$
$$= \min\{ D^{(k-1)}[i,k], D^{(k-1)}[i,k] + 0 \}$$
$$= D^{(k-1)}[i,k]$$

The *k*th row

- *k*th row of D^k is equal to the *k*th row of D^{k-1}

For all j, $D^{(k)}[k,j] =$

$$= \min\{ D^{(k-1)}[k,j], \mathrm{D}^{(k-1)}[k,k] + D^{(k-1)}[k,j] \}$$

$$= \min\{ D^{(k-1)}[k,j], 0 + D^{(k-1)}[k,j] \}$$

$$= D^{(k-1)}[k,j]$$

Floyd's Algorithm using a single *D*

Floyd

1. ***D*** $\leftarrow$ ***W*** // initialize *D* array to *W* [ ]

2. *P* $\leftarrow$ 0 // initialize P array to [0]

3. **for *k*** $\leftarrow$ **1 to *n***

4. do for *i* $\leftarrow$ **1 to *n***

5. do for *j* $\leftarrow$ **1 to *n***

6. if (*D*[*i*, *j*] > *D*[*i*, *k*] + *D*[*k*, *j*])

7. then *D*[*i*, *j*] $\leftarrow$ ***D*[*i*, *k*] + *D*[*k*, *j*]**

8. *P*[*i*, *j*] $\leftarrow$ ***k*;**

Application: Feasibility Problem

- **Linear Programming**

$$\max c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \text{ (objective function)}$$

$$\text{subject to } Ax \le b \text{ (constraints)}$$

  - *Simplex* is a common approach used to solve the above problem

- **Feasibility problem**

  - Find x such that $Ax \le b$

Application: Feasibility Problem (cont.)

- **Special case of fesibility problem**

  - All constraints have the form $x_j - x_i \le b_k$

$x_1 - x_2 \le 3$

$$x_2 - x_3 \le -2 \quad \text{or} \quad \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \le \begin{bmatrix} 3 \\ -2 \\ 2 \end{bmatrix}$$

$x_1 - x_3 \le 2$

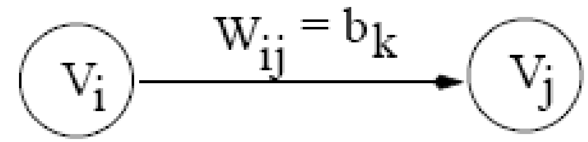
Application: Feasibility Problem (cont.)

- **Constraint graph**

  - Assign one vertex per variable

  - Assign one edge per constraint with weight b_k

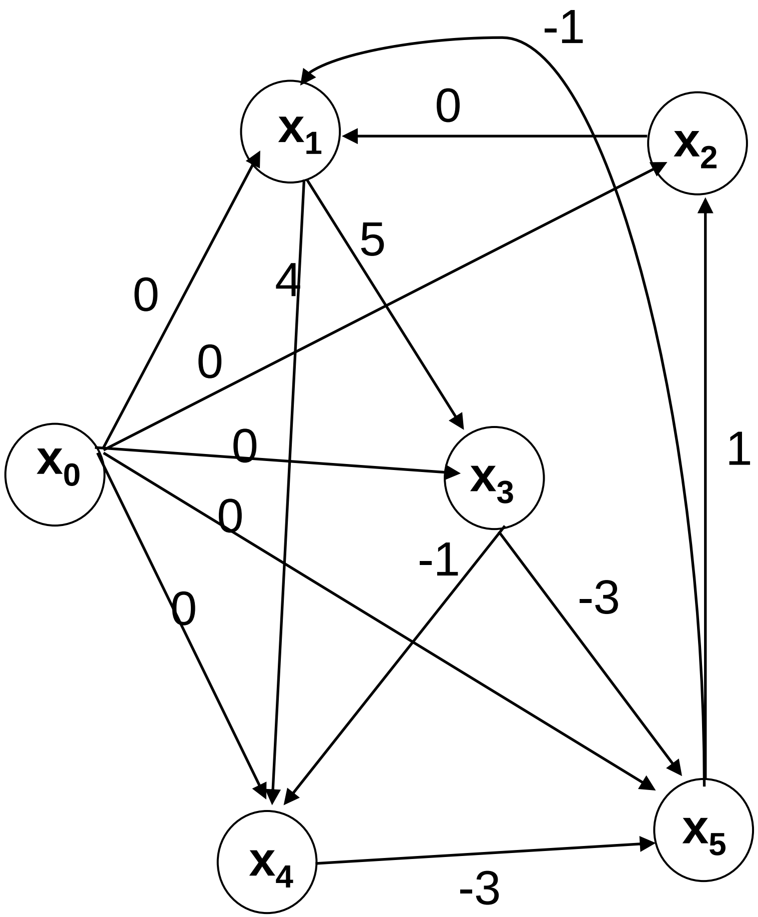
If $X_j - X_i \le b_k$ then



- Include an extra vertex and edges from this vertex to every other vertex

- Set the weights of the extra edges to zero

Application: Feasibility Problem (cont.)

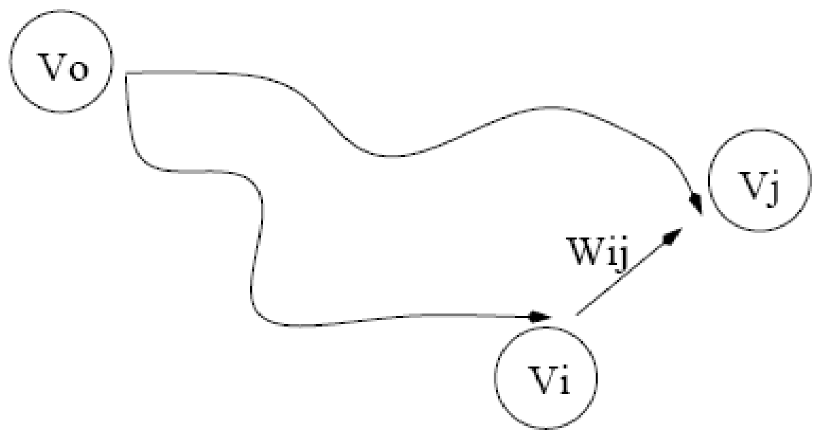


$$
\begin{aligned}
x_1 - x_2 &\le 0 \\
x_1 - x_5 &\le -1 \\
x_2 - x_5 &\le 1 \\
x_3 - x_1 &\le 5 \\
x_4 - x_1 &\le 4 \\
x_4 - x_3 &\le -1 \\
x_5 - x_3 &\le -3 \\
x_5 - x_4 &\le -3
\end{aligned}
$$

(feasible solution: -5, -3, 0, -1, -4)

Application: Feasibility Problem (cont.)

Theorem: If G contains no negative cycles, then $(\delta(v_0,v_1), \delta(v_0,v_2),\ldots, \delta(v_0,v_n))$ is a feasible solution.



For every (v_i, v_j): $\delta(v_0,v_j) \leq \delta(v_0,v_i)+w(v_i,v_j)$

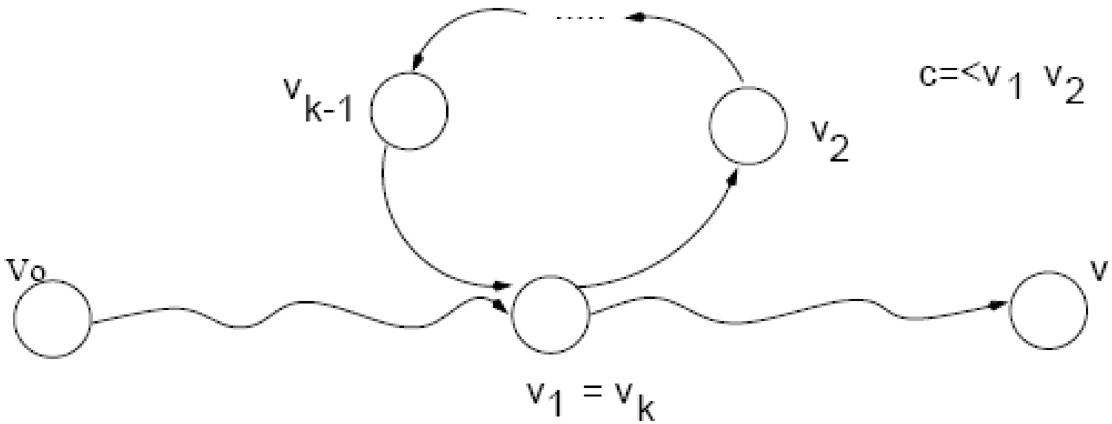
or $\delta(v_0,v_j) - \delta(v_0,v_i) \leq w(v_i,v_j)$

Setting $x_i= \delta(v_0,v_i)$ and $x_j= \delta(v_0,v_j)$, we have

$x_j-x_i \leq w(v_i,v_j)$

Application: Feasibility Problem (cont.)

- **Theorem:** If G contains a negative cycle, then there is no feasible solution.



$c = \langle v_1 \; v_2 \; \cdots \; v_k \rangle$ is a negative cycle

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

Proof by contradiction: suppose there exist a solution, then:

$$\begin{aligned} x_2 - x_1 &\le w(v_1, v_2) \\ x_3 - x_2 &\le w(v_2, v_3) \\ &\cdots\cdots\cdots\cdots \\ x_k - x_{k-1} &\le w(v_{k-1}, v_k) \\ x_1 - x_k &\le w(v_k, v_1) \end{aligned}$$

- Add them up:

$$0 \le \sum_{i=1}^{k-1} w(v_i, v_{i+1}) \quad \textbf{Contradiction !!}$$

Application: Feasibility Problem (cont.)

- **Size of the constraint graph**

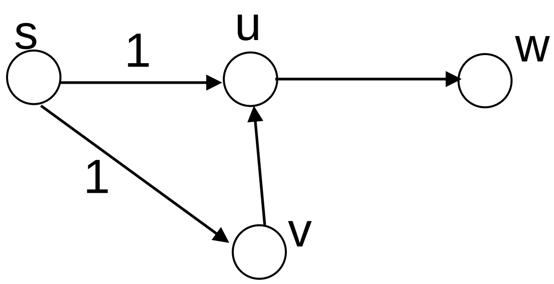
  - If we have m constraints with n unknowns ($Ax \leq b$, A is m x n)

  $V = n + 1$ and $E = m + n$

  - Running time: $O(VE) = O((n+1)(m+n)) = O(n^2 + nm)$

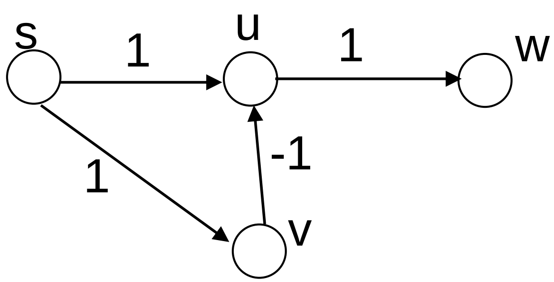
Problem 1

Write down weights for the edges of the following graph, so that Dijkstra's algorithm would not find the correct shortest path from *s* to *t*.



Problem 1

Write down weights for the edges of the following graph, so that Dijkstra's algorithm would not find the correct shortest path from *s* to *t*.



1st iteration

d[s]=0
d[u]=1
d[v]=1

S={s} Q={u,v,w}

2nd iteration

d[w]=2

S={s,u} Q={v,w}

3rd iteration

d[u]=0

S={s,u,v} Q={w}

4th iteration

S={s,u,v,w}
Q={}

- d[w] is not correct!
- d[u] should have converged when u was included in S!

Problem 2

- **(Exercise 24.3-4, page 600)** We are given a directed graph $G=(V,E)$ on which each edge (u,v) has an associated value $r(u,v)$, which is a real number in the range $0 \le r(u,v) \le 1$ that represents the reliability of a communication channel from vertex u to vertex v.
- We interpret $r(u,v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent.
- Give an efficient algorithm to find the most reliable path between two given vertices.

Problem 2 (cont.)

- Solution 1: modify Dijkstra's algorithm

  - Perform relaxation as follows:

    if d[v] < d[u] w(u,v) then

    d[v] = d[u] w(u,v)

  - Use "EXTRACT_MAX" instead of "EXTRACT_MIN"

Problem 2 (cont.)

- Solution 2: use Dijkstra's algorithm without any modifications!
  - r(u,v)=Pr(channel from u to v will not fail)
  - Assuming that the probabilities are independent, the reliability of a path p=<v_1,v_2,…,v_k> is:

    $r(v_1,v_2)r(v_2,v_3) \ldots r(v_{k-1},v_k)$

  - We want to find the channel with the highest reliability, i.e.,

$$\max_p \prod_{(u,v)\in p} r(u,v)$$

Problem 2 (cont.)

- But Dijkstra's algorithm computes

$$\min_p \sum_{(u,v)\in p} w(u,v)$$

- Take the *lg*

$$\lg(\max_p \prod_{(u,v)\in p} r(u,v)) = \max_p \sum_{(u,v)\in p} \lg(r(u,v))$$

Problem 2 (cont.)

- Turn this into a minimization problem by taking the negative:

$$-\min_p \sum_{(u,v)\in p} \lg(r(u,v)) = \min_p \sum_{(u,v)\in p} -\lg(r(u,v))$$

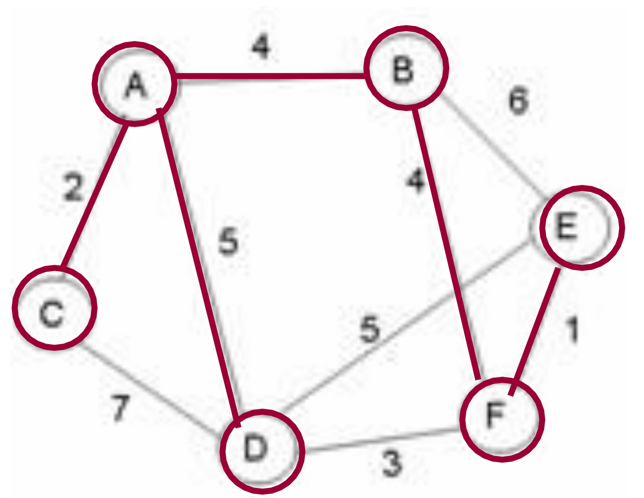
- Run Dijkstra’s algorithm using

$$w(u,v) = -\lg(r(u,v))$$

Problem 3

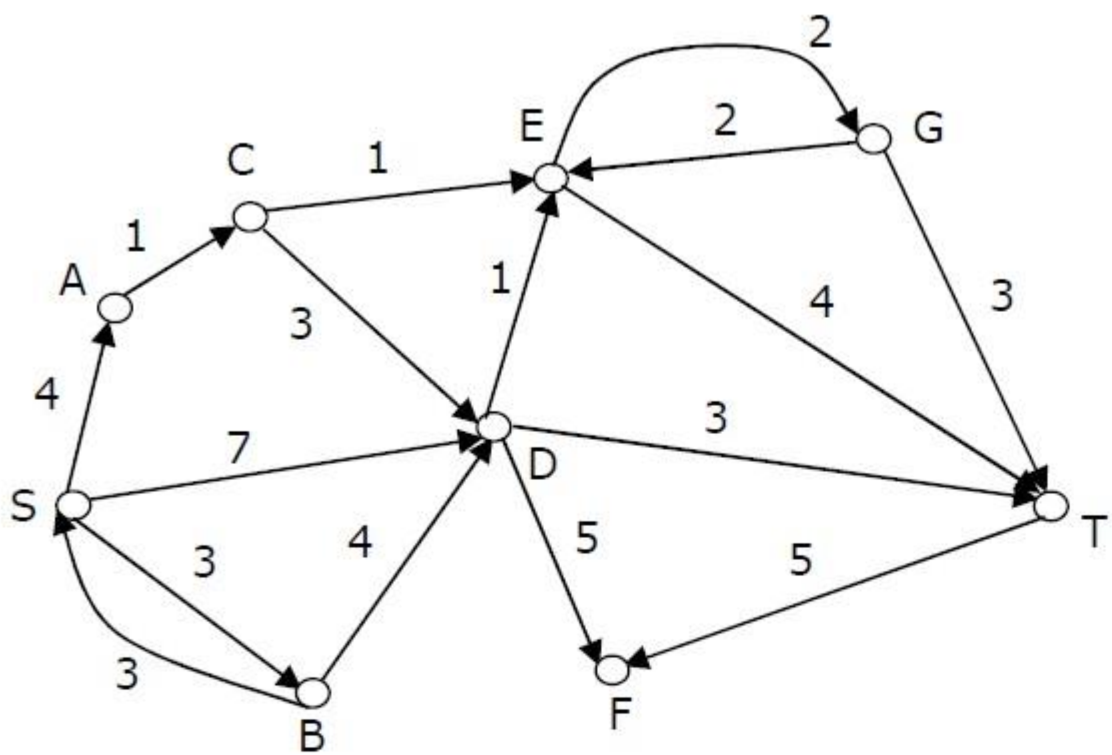
Node	Included	Distance	Path
A	t	-	-
B	~~f~~ t	4	A
C	~~f~~ t	2	A
D	~~f~~ t	5	A
E	~~f~~ t	~~∞~~ ~~10~~ 9	~~-~~ ~~B~~ F
F	~~f~~ t	~~∞~~ 8	~~-~~ B

- Give the shortest path tree for node A for this graph using Dijkstra's shortest path algorithm. Show your work with the 3 arrays given and draw the resultant shortest path tree with edge weights included.



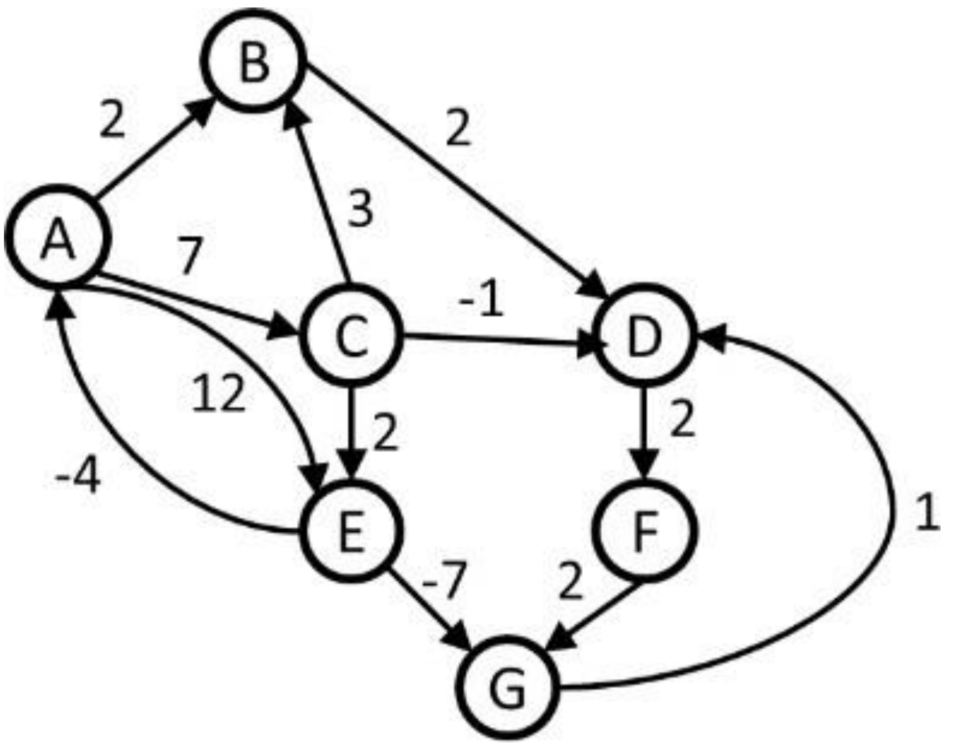
Quiz 1

- Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will l



Quiz 2

- Calculate shortest paths from A to every other vertex using dijkstra algorithm



WAKE UP IT'S QUIZ TIME

Quiz 3

- Show the result of Dijkstra's algorithm from F to D