

# **Chapter 24, 25**

## **Shortest Paths**

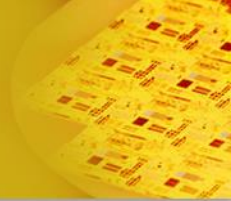
Algorithm Analysis

School of CSEE

# Shortest path

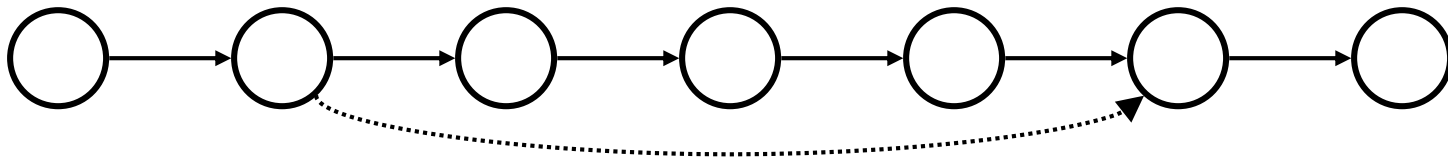
- Input : directed graph  $G = (V, E)$  with weight function  $w : E \rightarrow R$
- Determine the path  $p$  with minimum weight from source to destination.
- Weight  $w(p)$  of path  $p$  : sum of edge weights on path  $p$
- *shortest-path weight  $u$  to  $v$  :*  
$$\delta(u, v) = \begin{cases} \min \{ w(p) : \text{path } p \text{ from } u \text{ to } v \} & \text{if there exists a path from } u \text{ to } v. \\ \infty, & \text{otherwise.} \end{cases}$$

- **Single-source shortest path** : find shortest paths from a given source vertex  $s \in V$  to every vertex  $v \in V$
- Single-destinations : find shortest paths to a given destination vertex. By reversing the direction of the each edge in the graph, the problem is reduced to single-source problem.
- Single-pair : find shortest path from  $u$  to  $v$ . No easier than single-source problem.
- **All-pairs shortest-paths** : find shortest path from  $u$  to  $v$  for all  $u, v \in V$



- Single-source shortest path
  - Bellman-Ford algorithm
  - In DAG
  - Dijkstra's algorithm
- All-pairs shortest-paths
  - Floyd-Warshall algorithm

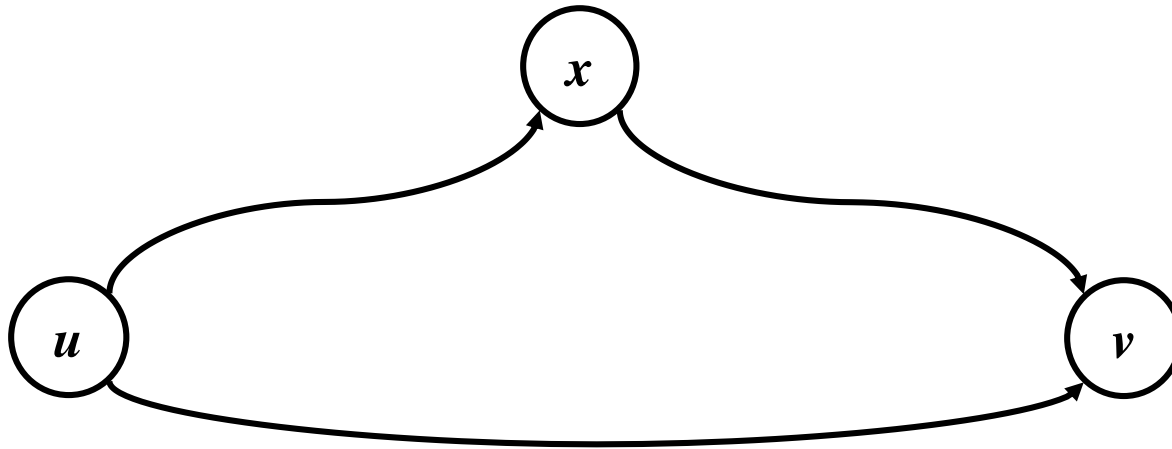
- Again, we have *optimal substructure*: the shortest path consists of shortest subpaths:



- Proof: suppose some subpath is not a shortest path.
  - There must then exist a shorter subpath.
  - We could substitute the shorter subpath for a shortest path.
  - But then overall path is not shortest path. Contradiction!!

# Shortest Path Properties

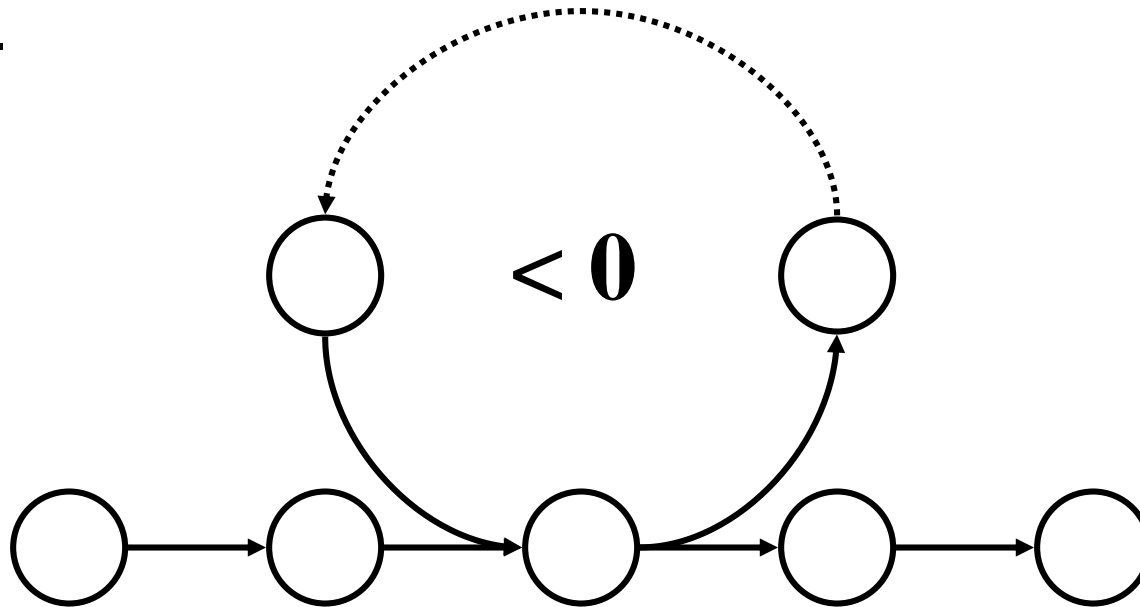
- Define  $\delta(u, v)$  to be the weight of the shortest path from  $u$  to  $v$ . Shortest paths satisfy the *triangle inequality*:  $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$



*This path is no longer than any other path*

# Negative-weight edges

- OK, as long as no negative-weight cycle are reachable from the source.
- If we reach a negative-weight cycle, we can just keep going around it and get  $w(s, v) = -\infty$  for all  $v$  in the cycle.
- Some algorithms work only if there are no negative-weight edges.

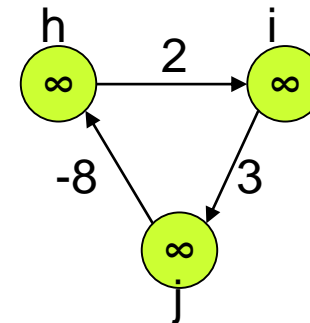
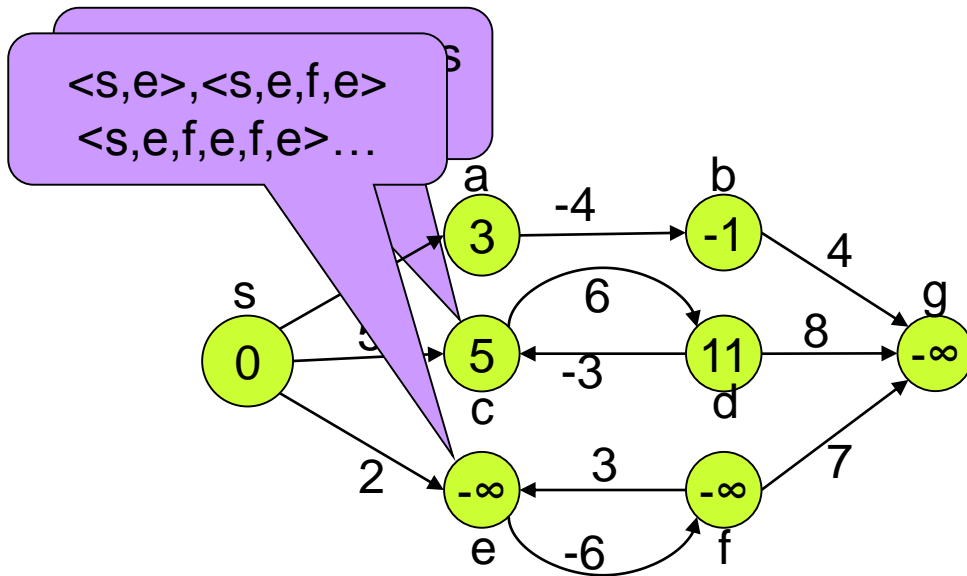


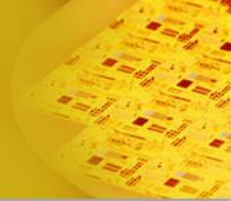
- Shortest paths can't contain cycles.
  - We assumed that there is no negative-weight cycles.
  - Positive-weight cycle : Just omit it to get a shorter path.
  - Zero-weight cycle : There is no reason to use them. Assume that our solutions won't use them.



# Cycles

- If there is a cycle reachable from  $s$ , shortest-path weights are not well defined.
- $\delta(s, v) = -\infty$  ( $v \in V$ )





# Single Source Shortest Path

# SS shortest-path algorithm

- Find shortest paths from a given source vertex  $s \in V$  to every vertex  $v \in V$ .
- Output for each vertex  $v \in V$ :
  - $d[v] = \delta(s, v)$ 
    - Initially,  $d[v] = \infty$
    - Reduces  $d[v]$  as algorithm progress.  
But always maintain  $d[v] \geq \delta(s, v)$
    - $d[v]$  : *shortest-path estimate*
  - $\pi[v]$  = predecessor of  $v$  on a shortest path from  $s$ .
    - If no predecessor,  $\pi[v] = \text{NIL}$
    - $\pi$  induces a tree – *shortest-path tree*

INIT-SINGLE-SOURCE( $V, s$ )

for each  $v \in V$

do  $d[v] = \infty$

$\pi[v] = \text{NIL}$

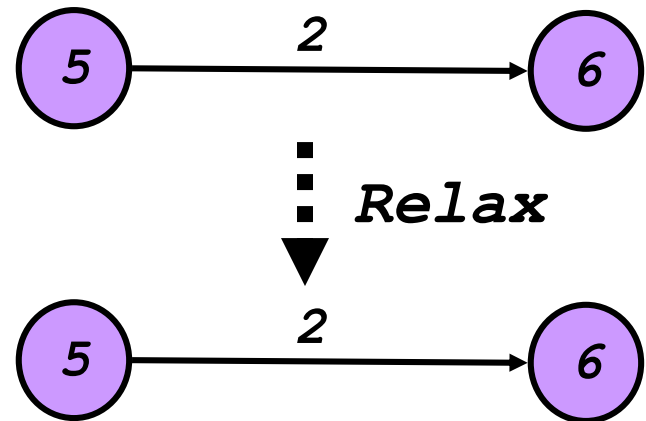
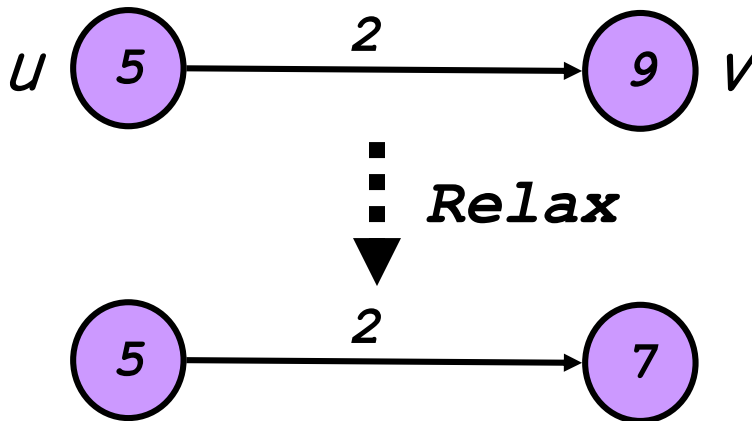
$d[s] = 0$

# Relaxation

- A key technique in shortest path algorithms is *relaxation*.
  - Idea: for all  $v$ , maintain upper bound  $d[v]$  on  $\delta(s,v)$

```

Relax( $u, v, w$ ) {
  if ( $d[v] > d[u] + w$ ) /*  $w = w(u, v)$  */
    then  $d[v] = d[u] + w$ ;
     $\pi[v] = u$ 
}
```



# [1] Bellman-Ford algorithm

- Solves the single-source shortest-paths problem in general case in which **edge weights may be negative**.
- Allow negative-weight edges and produce a correct answer as long as no negative-weight cycles are reachable from the source.
- Returns a boolean value
  - negative-weight cycle – FALSE
  - No such cycle – TRUE

# Pseudocode

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

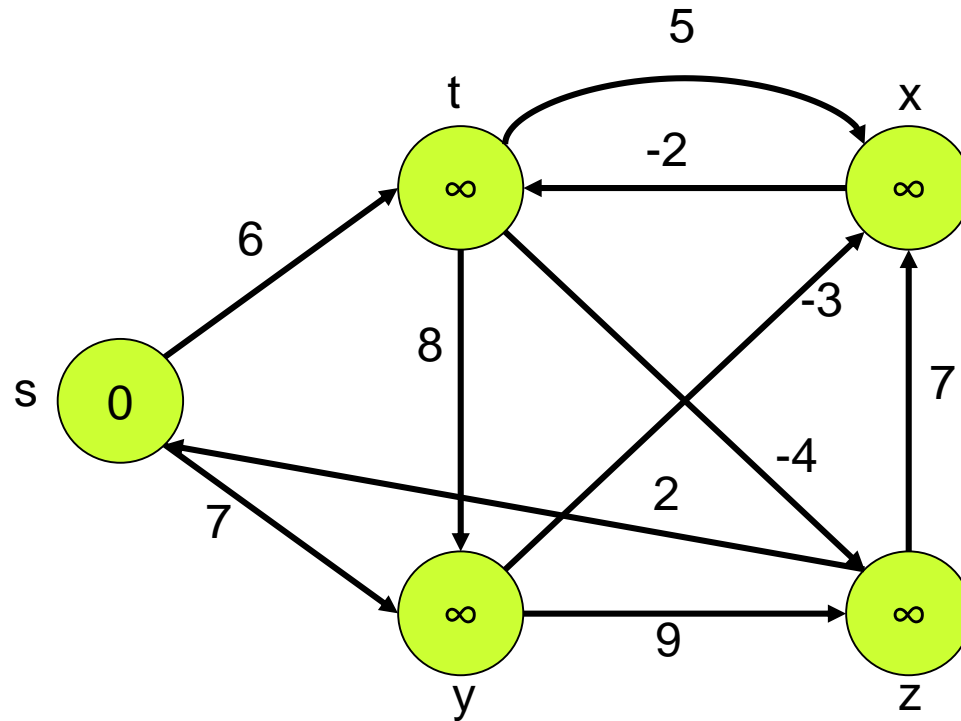
- |   |  |   |
|---|--|---|
| <ol style="list-style-type: none"> <li>1. INIT-SINGLE-SOURCE(<math>V, s</math>)</li> <li>2. for <math>i = 1</math> to <math> V  - 1</math></li> <li>3.     do for each edge <math>(u, v) \in E</math></li> <li>4.         do RELAX(<math>u, v, w</math>)</li> <li>5. for each edge <math>(u, v) \in E</math></li> <li>6.     do if <math>d[v] &gt; d[u] + w(u, v)</math></li> <li>7.         then return FALSE</li> <li>8. return TRUE</li> </ol> | <div style="font-size: 2em;">→</div> <div style="font-size: 4em;">}</div> <div style="font-size: 4em;">}</div> | <p><i>Initialize <math>d[]</math>, which will converge to shortest-path value <math>\delta</math></i></p> <p><i>Relaxation:</i><br/> <i>Make <math> V  - 1</math> passes, relaxing each edge</i></p> <p><i>Test for solution</i><br/> <i>Under what condition do we get a solution?</i></p> |
|---|--|---|

The first for loop relaxes all edges  $|V| - 1$  times.

Time :  $\Theta(VE)$

# Example

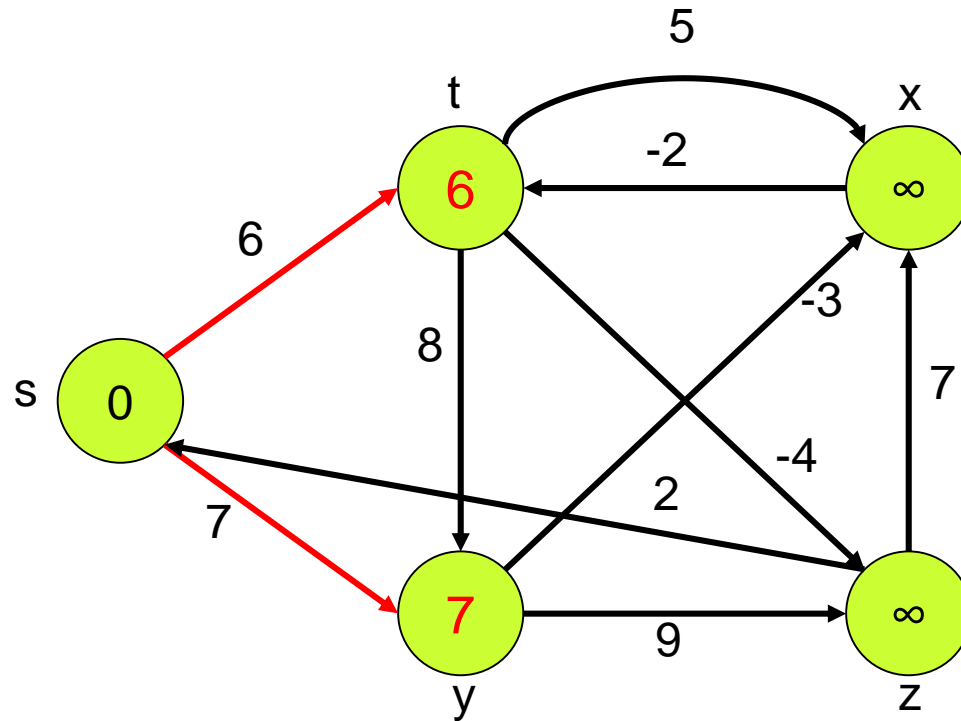
- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- After INIT-SINGLE-SOURCE(V, s)





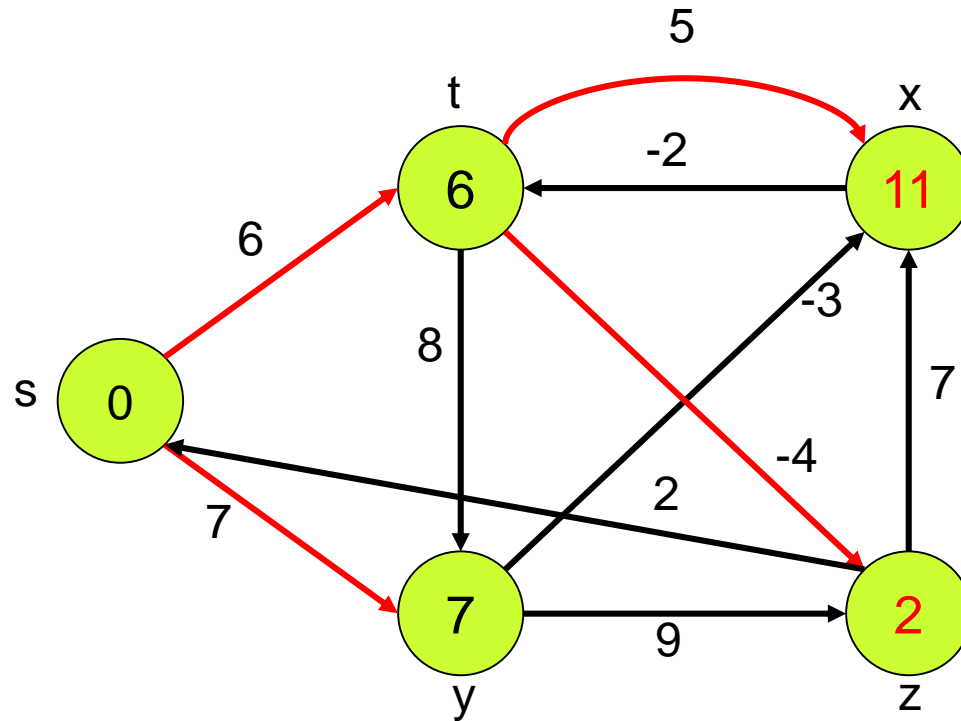
# Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), **(s,t)**, **(s,y)**
- At first pass



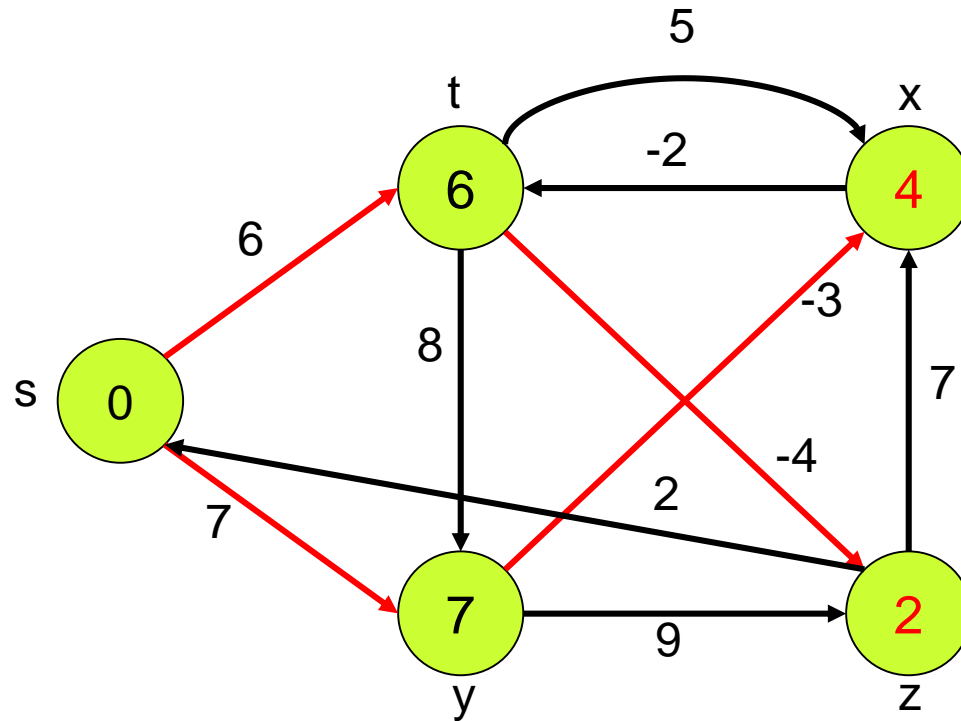
# Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At second pass



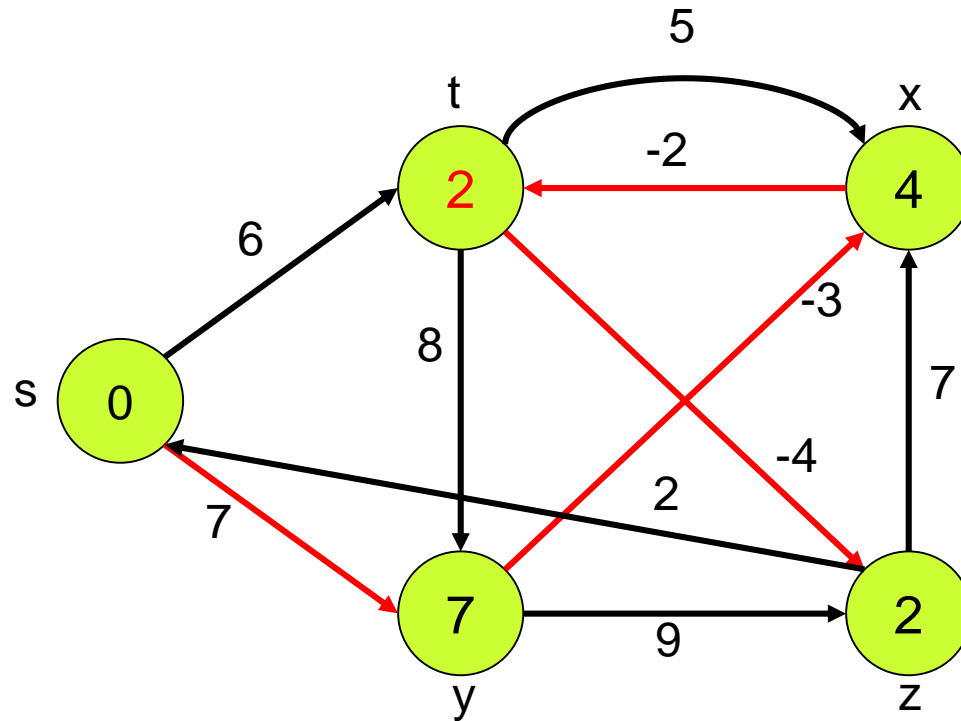
# Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At second pass



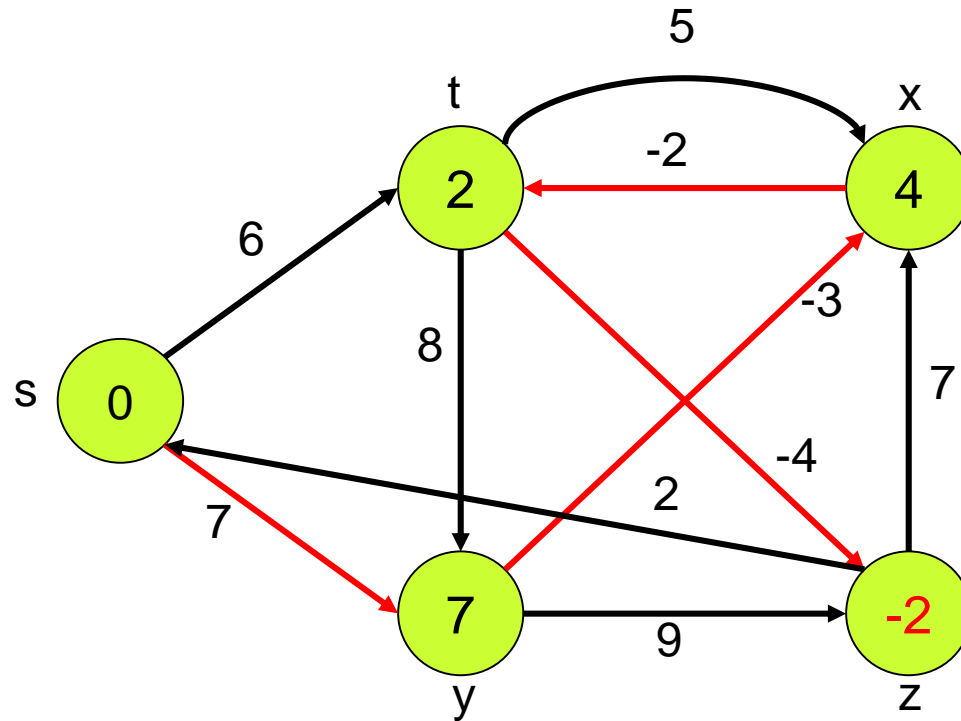
# Example

- Order : (t,x), (t,y), (t,z), **(x,t)**, (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At third pass



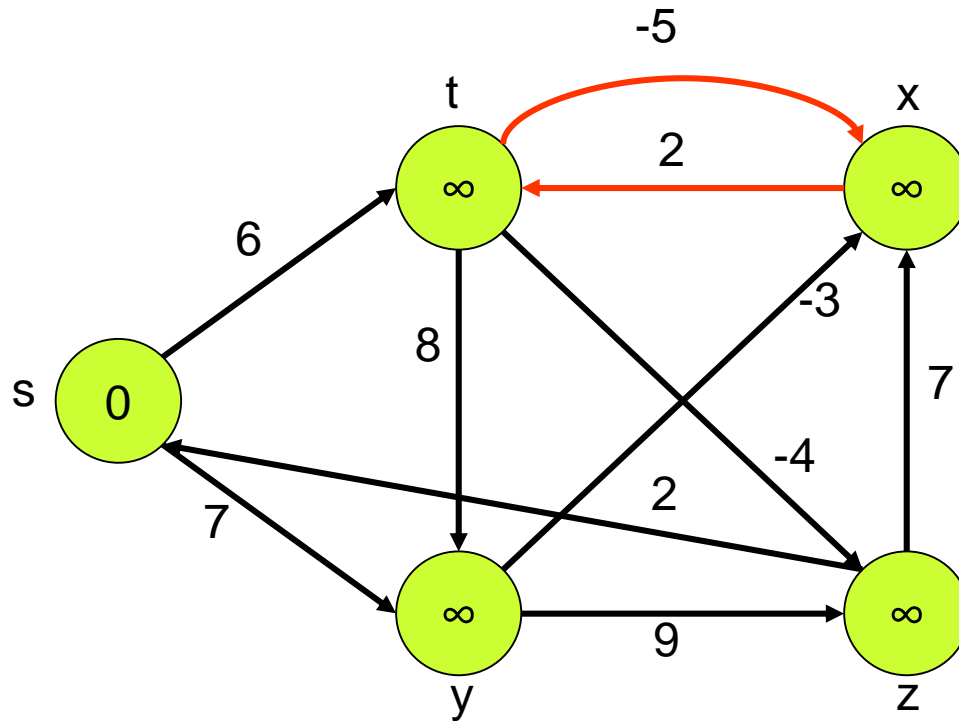
# Example

- Order : (t,x), (t,y), **(t,z)**, (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At fourth pass



# Neg. Weight Cycle Example

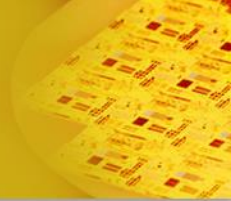
- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- After INIT-SINGLE-SOURCE(V, s)



- Values you get on each pass and how quickly it converges depends on order of relaxation.
- But guaranteed to converge after  $|V|-1$  passes, assuming no negative-weight cycles.
- Proof : use path-relaxation property.

## Theorem 24.4

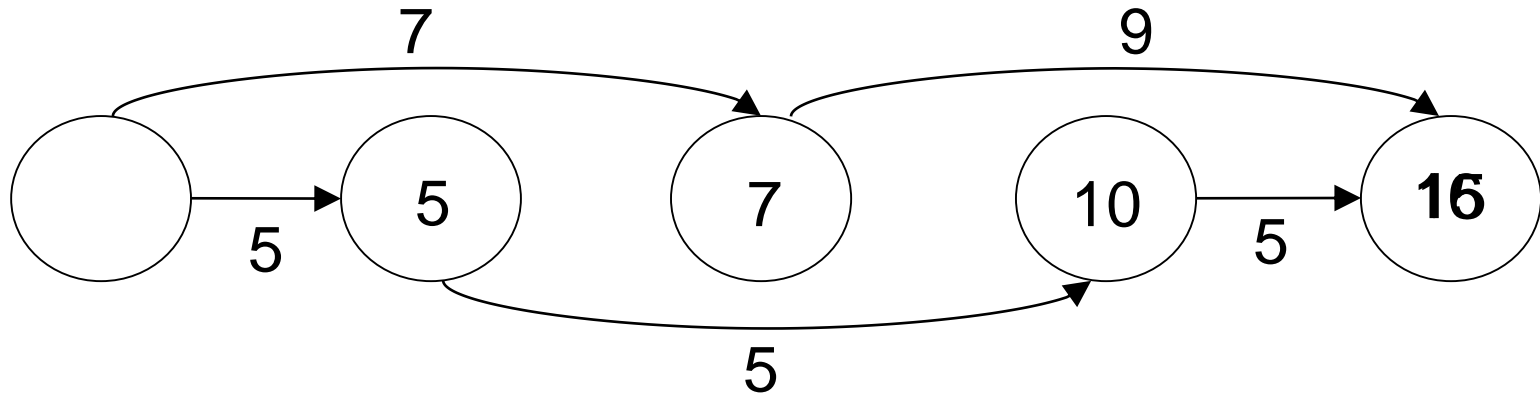
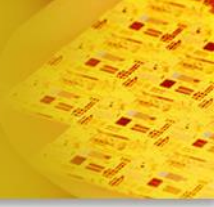
## [2] SSP in DAG

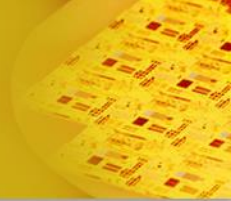


- Problem: finding shortest paths in DAG
  - Bellman-Ford takes  $O(VE)$  time.
  - *How can we do better?*
  - Idea: use topological sort
    - Since it is a DAG, there are no cycles.
    - Every path in a DAG is subsequence of topologically sorted, so processes vertices on each shortest path from left to right, then it would be done in one pass.
    - *What will be the running time?*



# Getting Dressed Example





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

topologically sort the vertices

INIT-SINGLE-SOURCE( $V, s$ )

for each vertex  $u$ , take in topologically sorted order

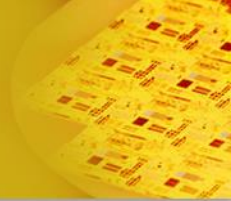
do for each vertex  $v \in Adj[u]$

do RELAX( $u, v, w$ )

Time :  $\Theta(V+E)$

# [3] Dijkstra's Algorithm

- If there are no negative edge weights, we can beat Bellman-Ford.
- Similar to breadth-first search : weighted version of breadth-first search.
  - Grow a tree gradually, advancing from vertices taken from a queue.
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weights ( $d[v]$ ).



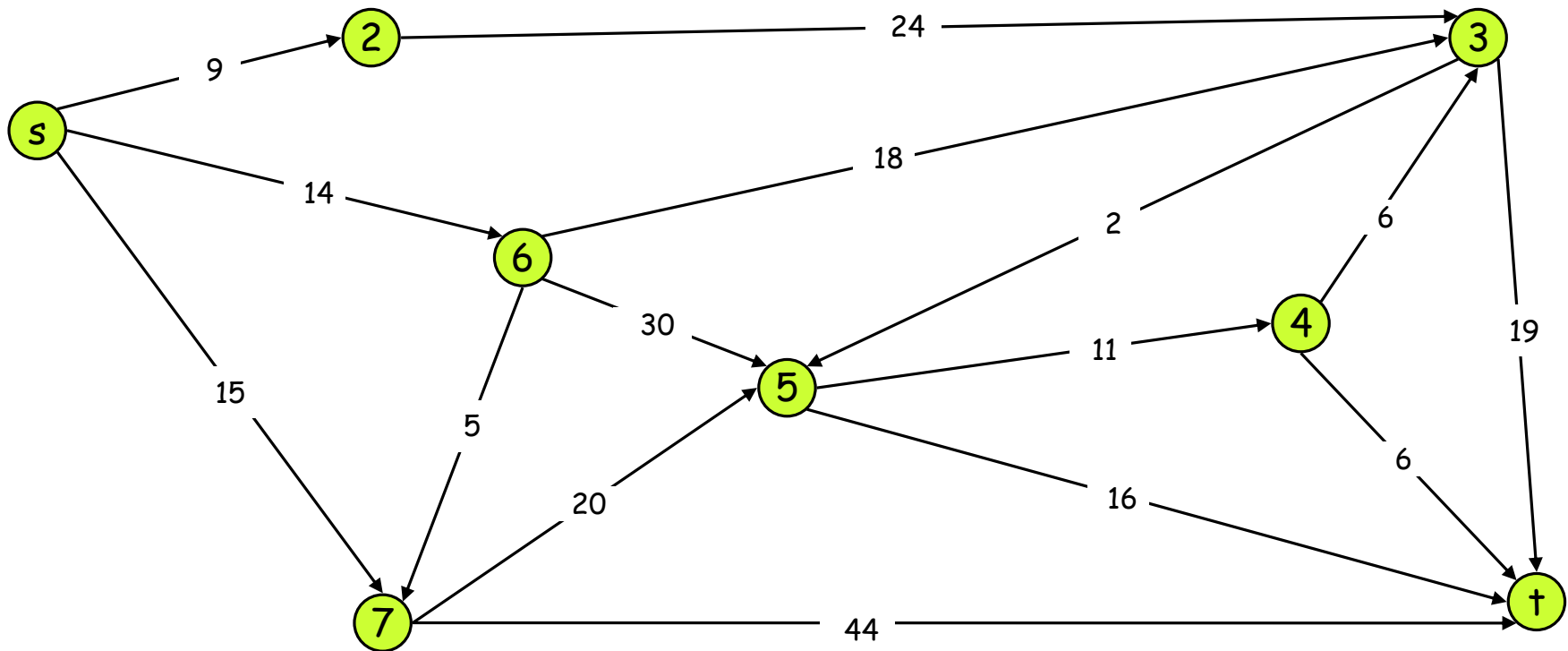
- Have two sets of vertices :
  - $S$  = vertices whose final shortest-path weights are determined.
  - $Q$  = priority queue =  $V - S$ .
- For the graph  $G=(V,E)$ , maintains a set  $S$  of vertices for which the shortest paths are known.
- Repeatedly selects the vertex  $u$  ( $u \in V-S$ ), with the minimum shortest-path estimated, adds  $u$  to  $S$ , and relaxes all edges leaving  $u$ .

DIJKSTRA( $G, w, s$ )

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 while  $Q \neq \emptyset$ 
5     do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6          $S \leftarrow S \cup \{u\}$ 
7         for each vertex  $v \in \text{Adj}[u]$ 
8             do RELAX( $u, v, w$ )
```

# Example

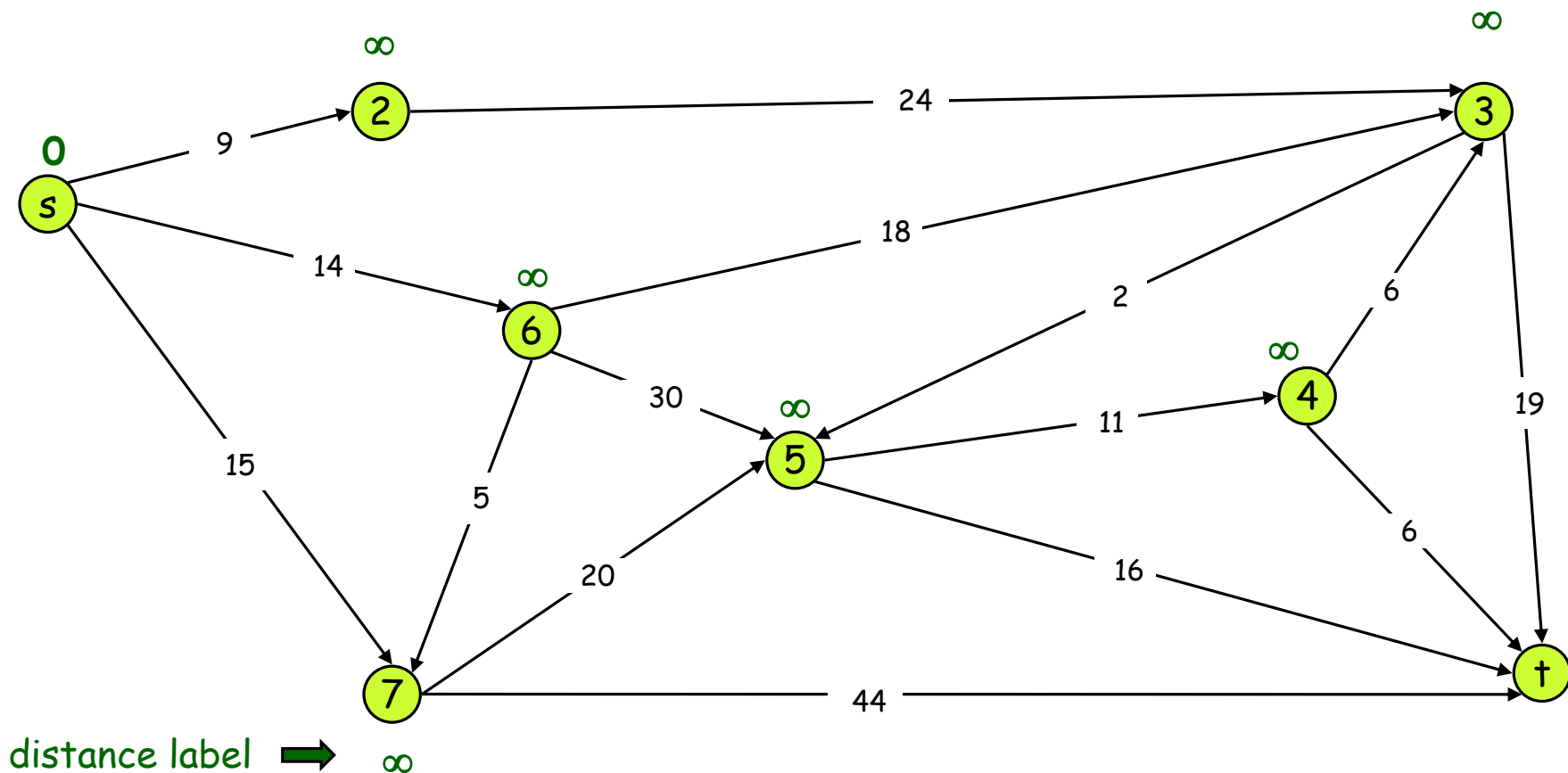
- Find shortest path from  $s$  to  $t$ .



# Example

$S = \{ \}$   
 $Q = \{ s, 2, 3, 4, 5, 6, 7, \dagger \}$

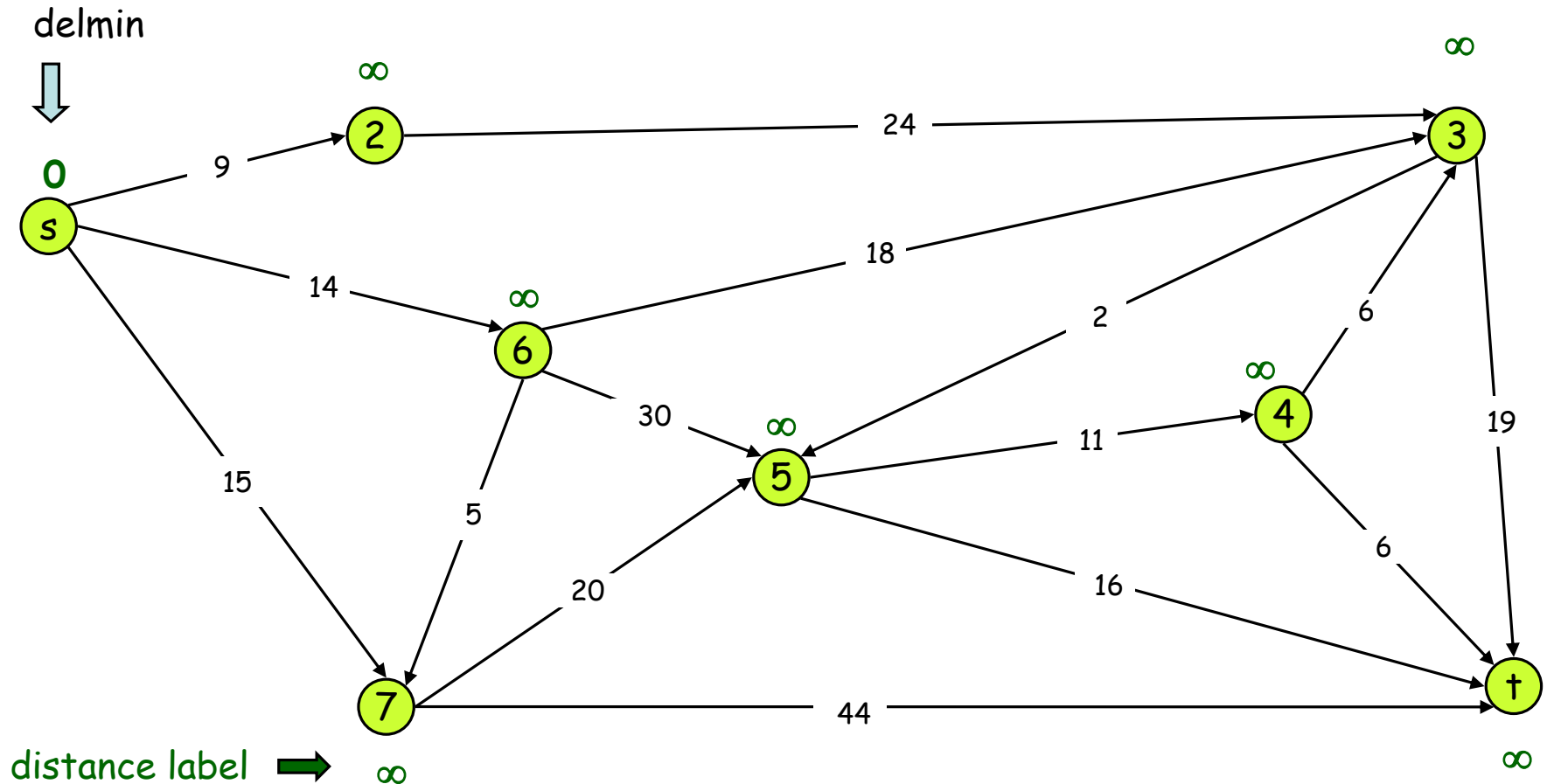
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
 $S \leftarrow \emptyset, Q \leftarrow V[G]$



# Example

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

**While**  $Q \neq \emptyset$   
**do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$

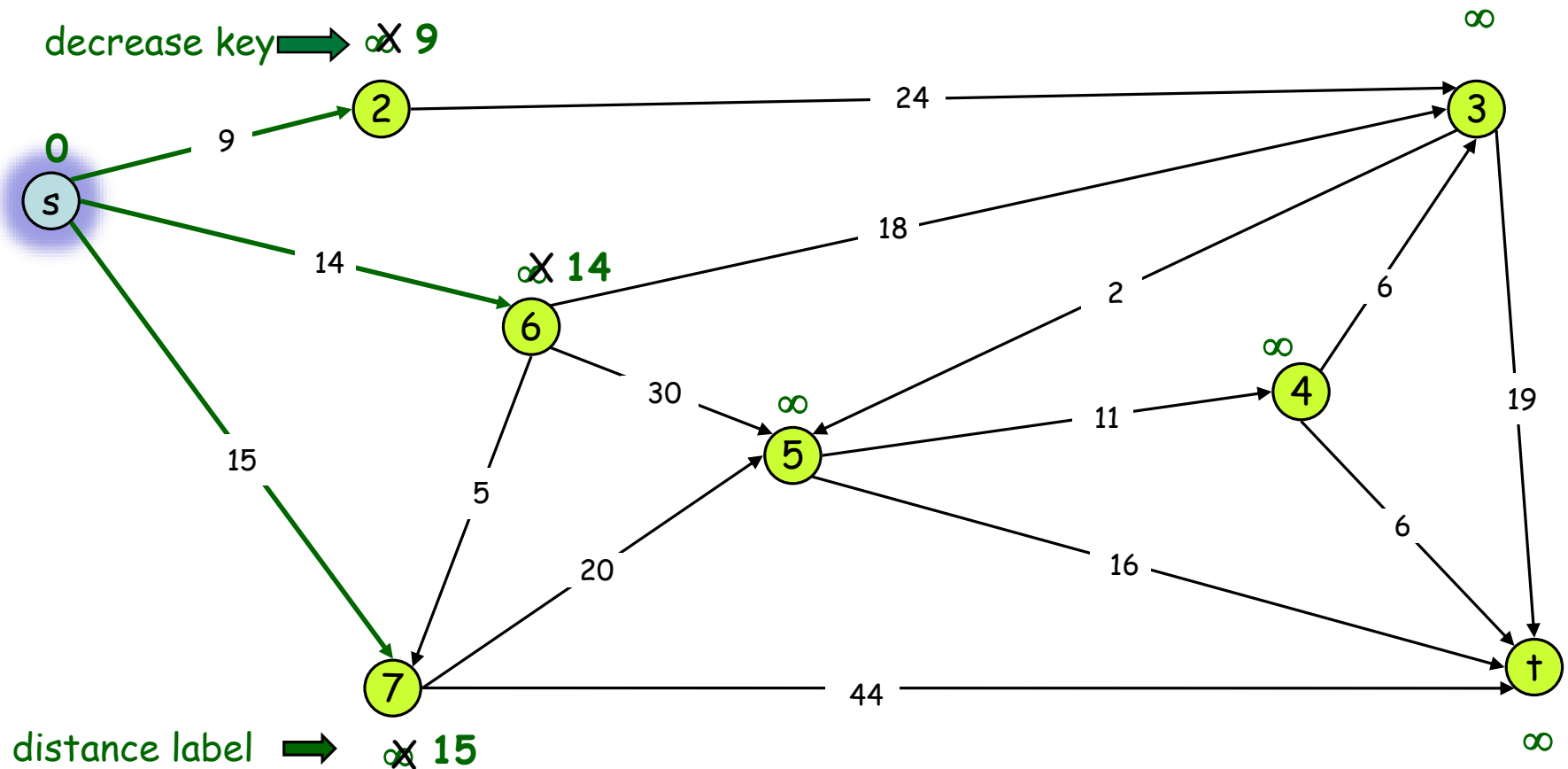




# Example

$S = \{s\}$   
 $Q = \{2, 6, 7, 3, 4, 5, \dagger\}$

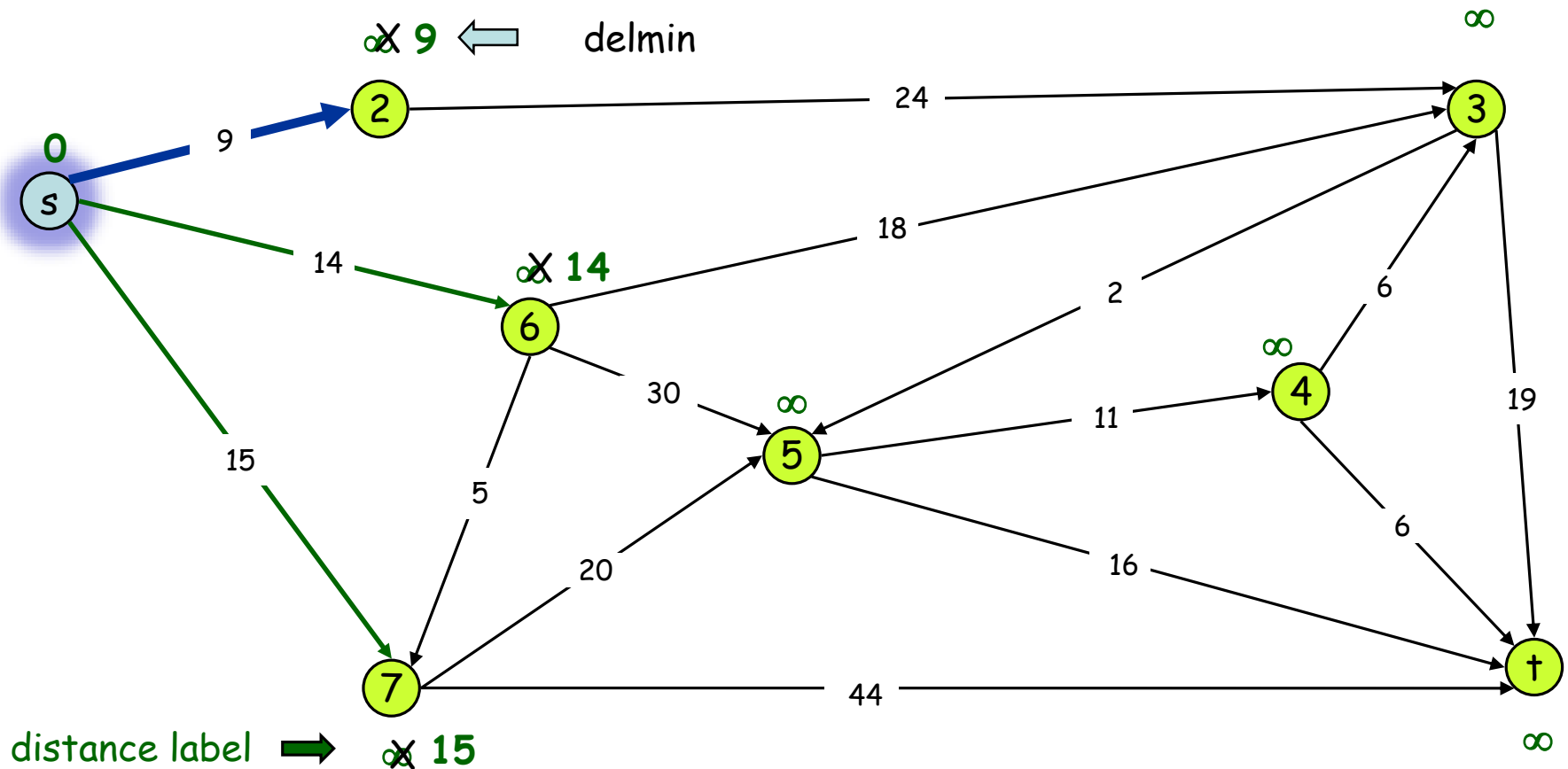
$S \leftarrow S \cup \{u\}$   
 for each vertex  $v \in \text{Adj}[u]$   
 do RELAX( $u, v, w$ )



# Example

$S = \{s\}$   
 $Q = \{2, 6, 7, 3, 4, 5, \dagger\}$

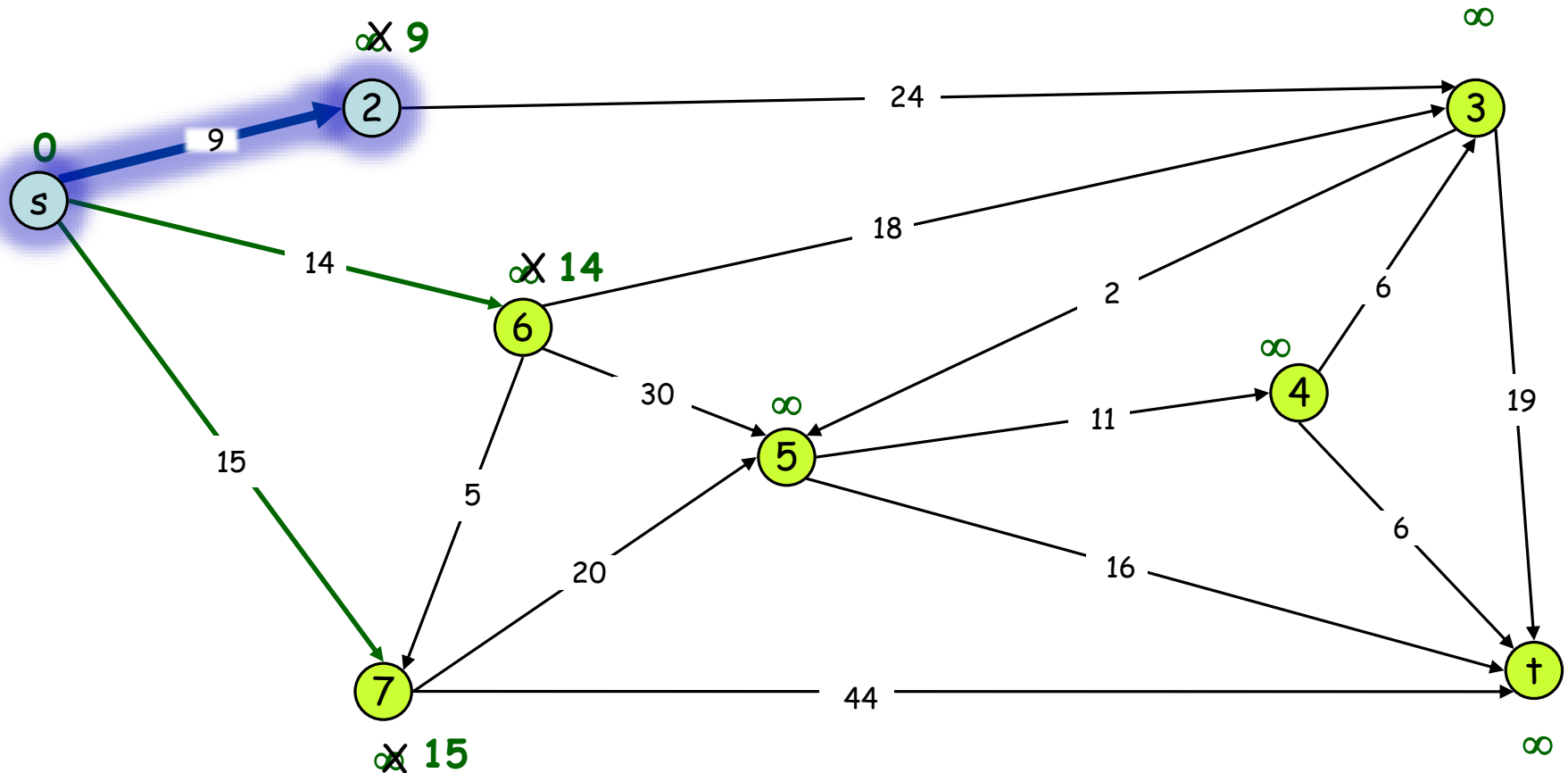
do  $u \leftarrow \text{EXTRACT-MIN}(Q)$



# Example

$S = \{s, 2\}$   
 $Q = \{6, 7, 3, 4, 5, \dagger\}$

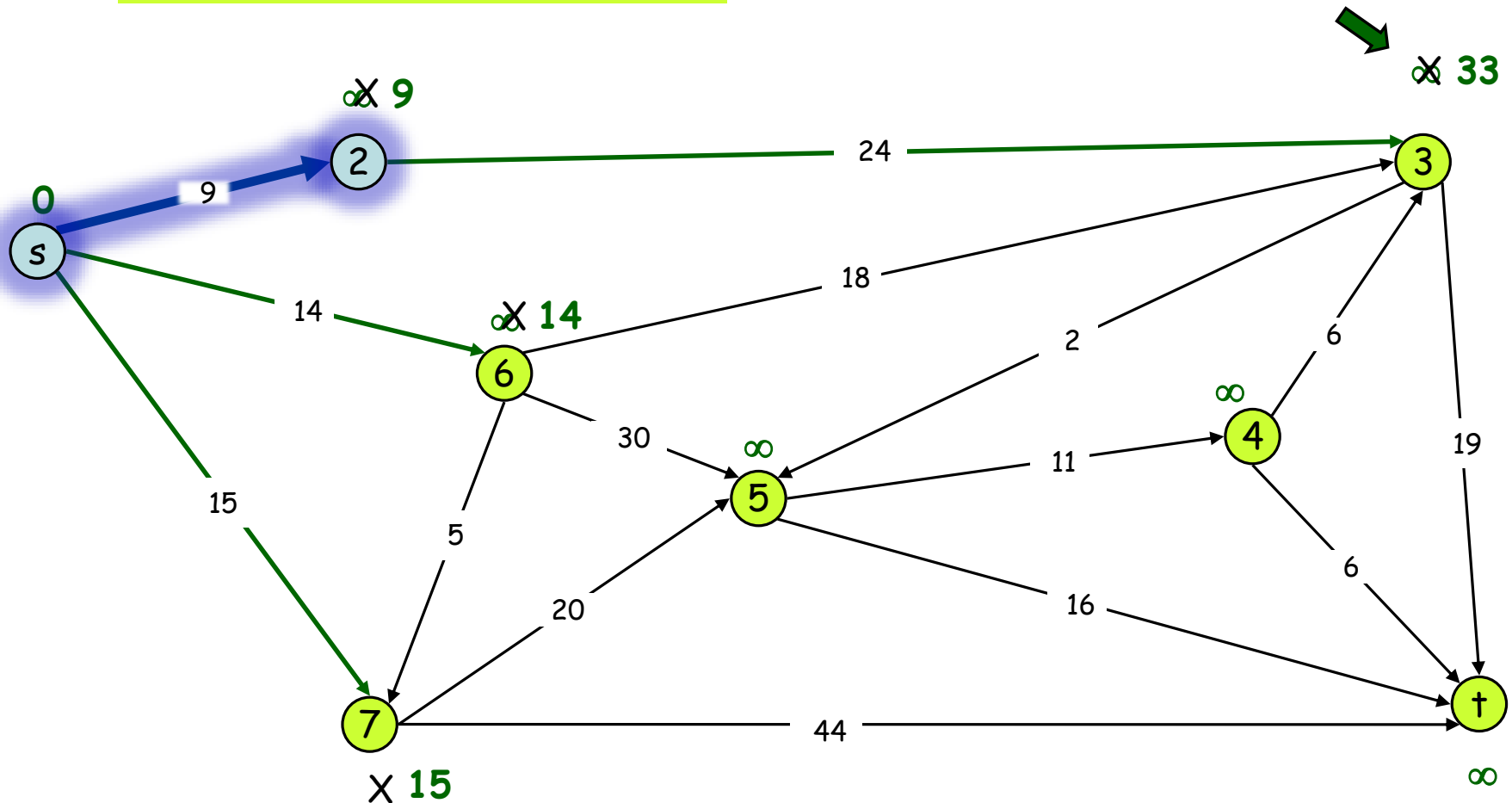
$S \leftarrow S \cup \{u\}$



# Example

$S = \{s, 2\}$   
 $Q = \{6, 7, 3, 4, 5, \dagger\}$

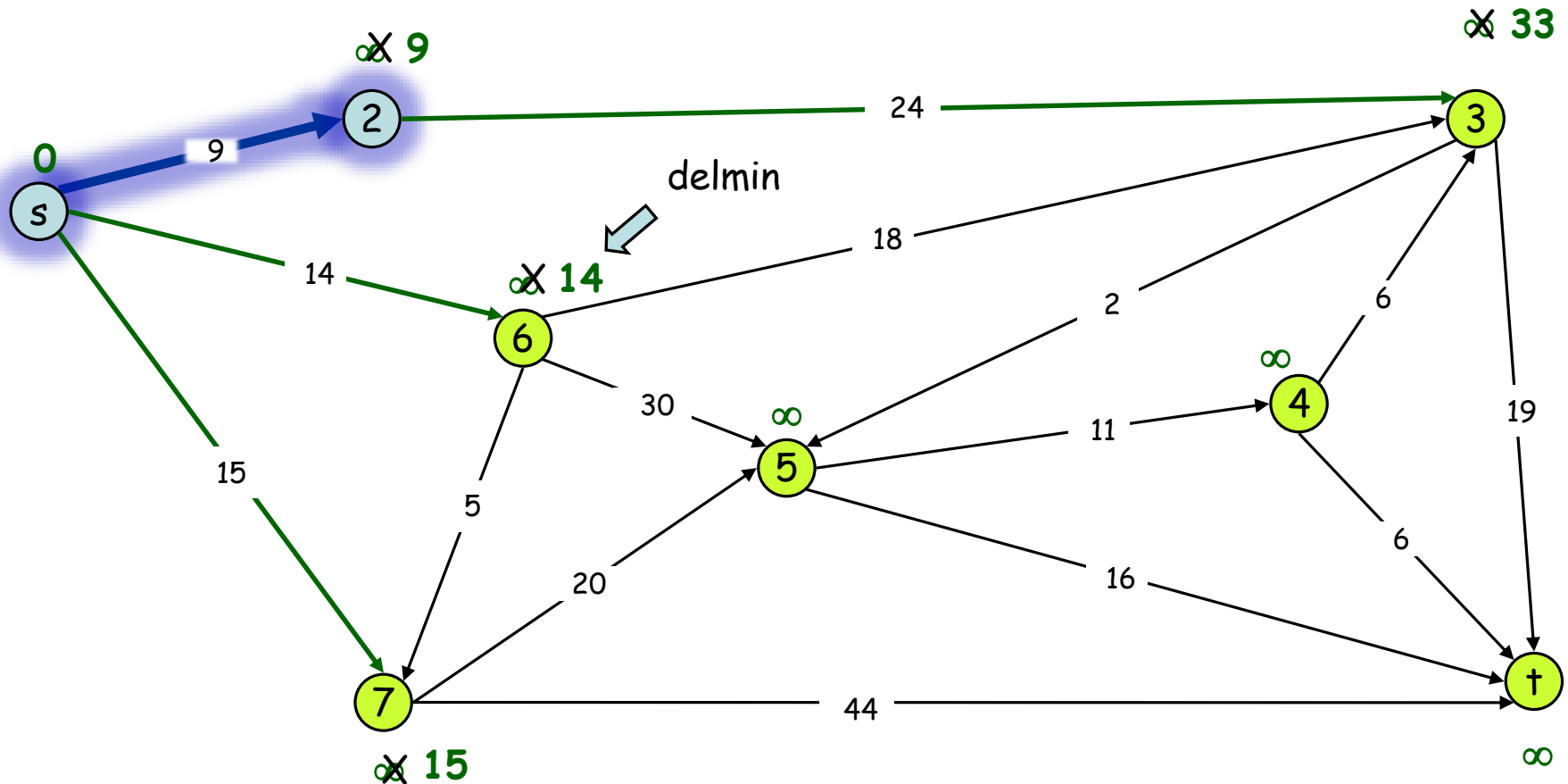
for each vertex  $v \in \text{Adj}[u]$   
 do RELAX( $u, v, w$ ) decrease key



# Example

$S = \{s, 2\}$   
 $Q = \{6, 7, 3, 4, 5, \dagger\}$

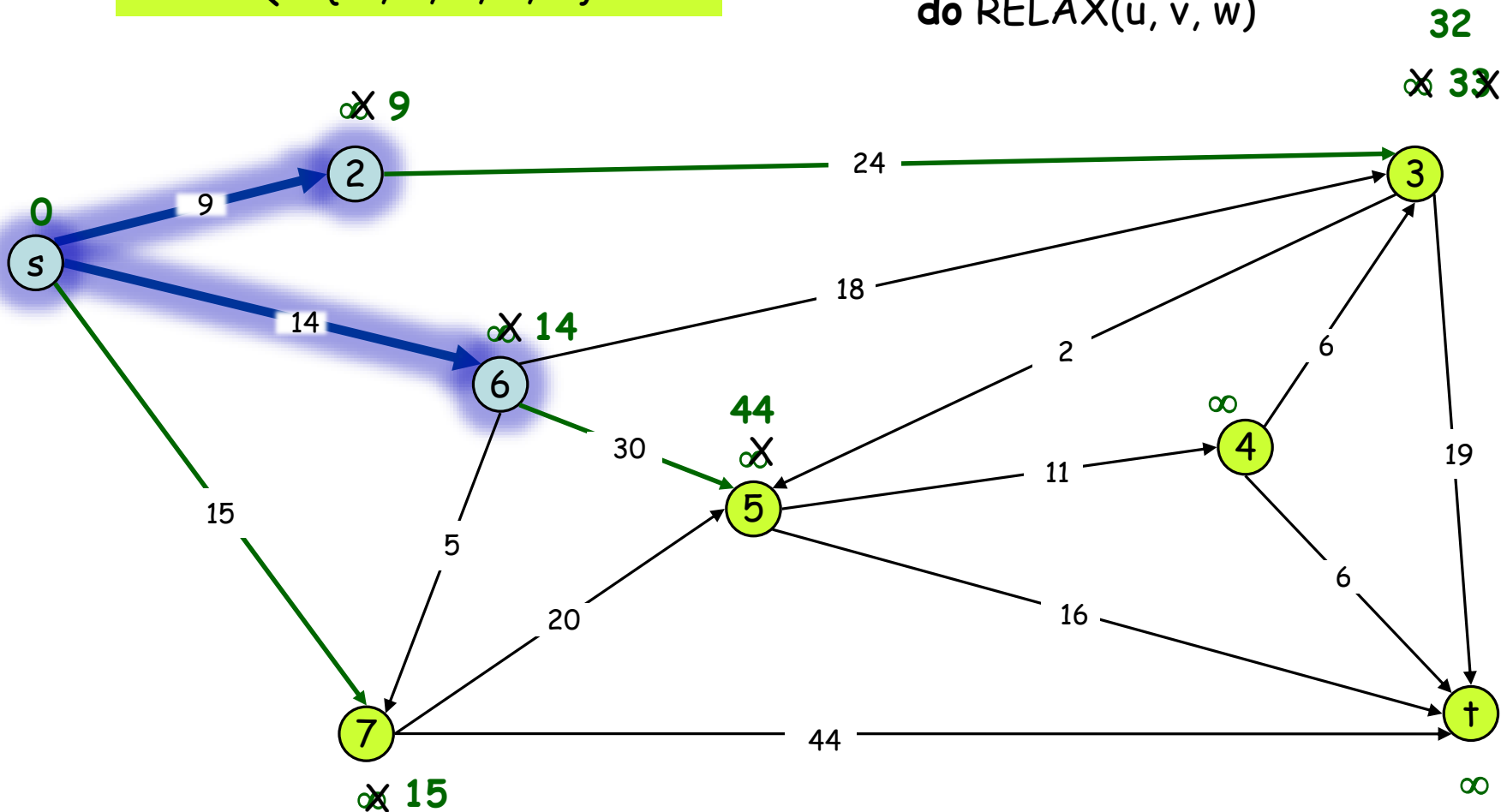
do  $u \leftarrow \text{EXTRACT-MIN}(Q)$



# Example

$S = \{s, 2, 6\}$   
 $Q = \{7, 3, 5, 4, \dagger\}$

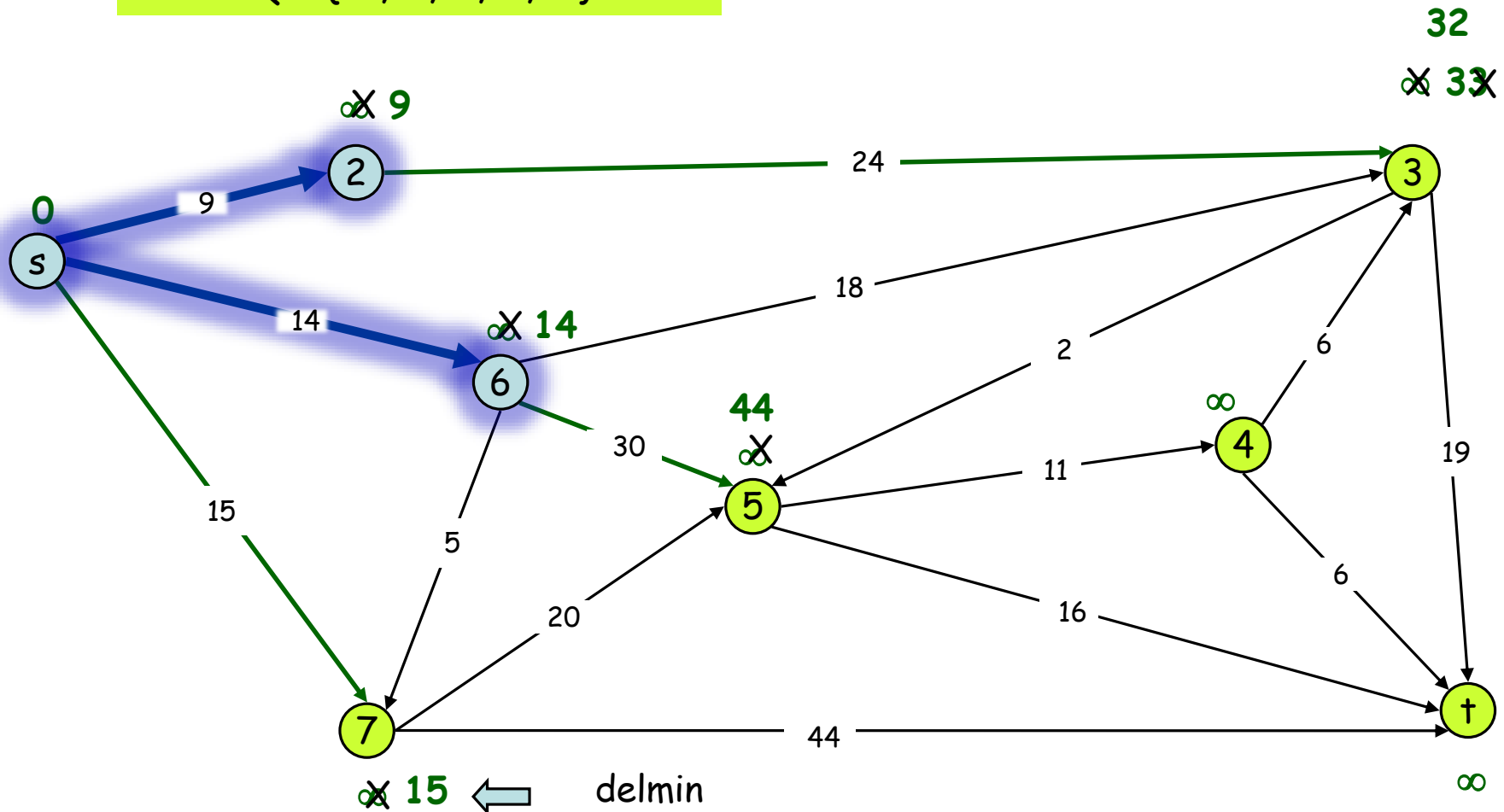
$S \leftarrow S \cup \{u\}$   
 for each vertex  $v \in \text{Adj}[u]$   
 do RELAX( $u, v, w$ )



# Example

$S = \{s, 2, 6\}$   
 $Q = \{7, 3, 5, 4, \dagger\}$

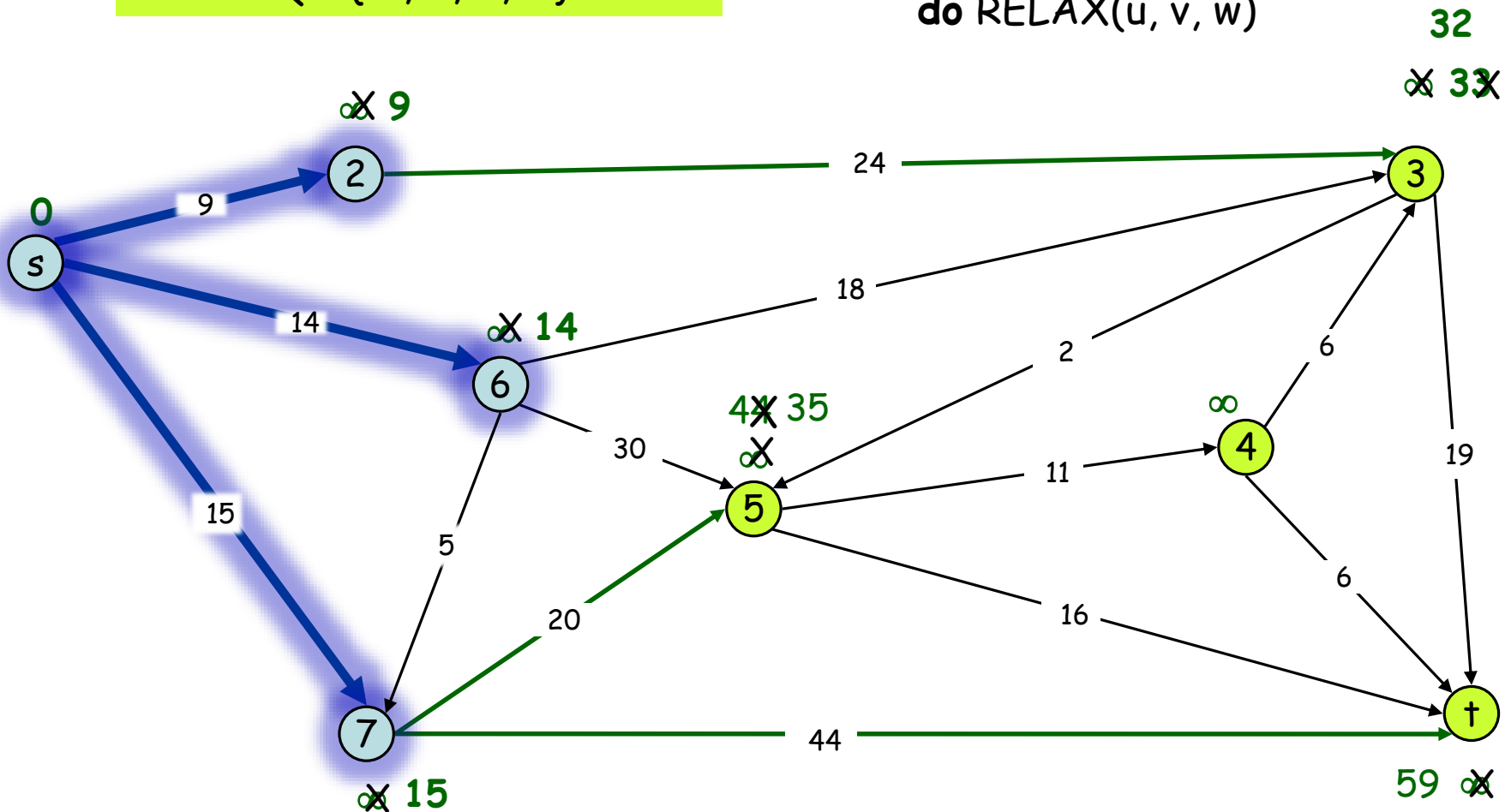
do  $u \leftarrow \text{EXTRACT-MIN}(Q)$



# Example

$S = \{s, 2, 6, 7\}$   
 $Q = \{3, 5, 4, \dagger\}$

$S \leftarrow S \cup \{u\}$   
 for each vertex  $v \in \text{Adj}[u]$   
 do RELAX( $u, v, w$ )

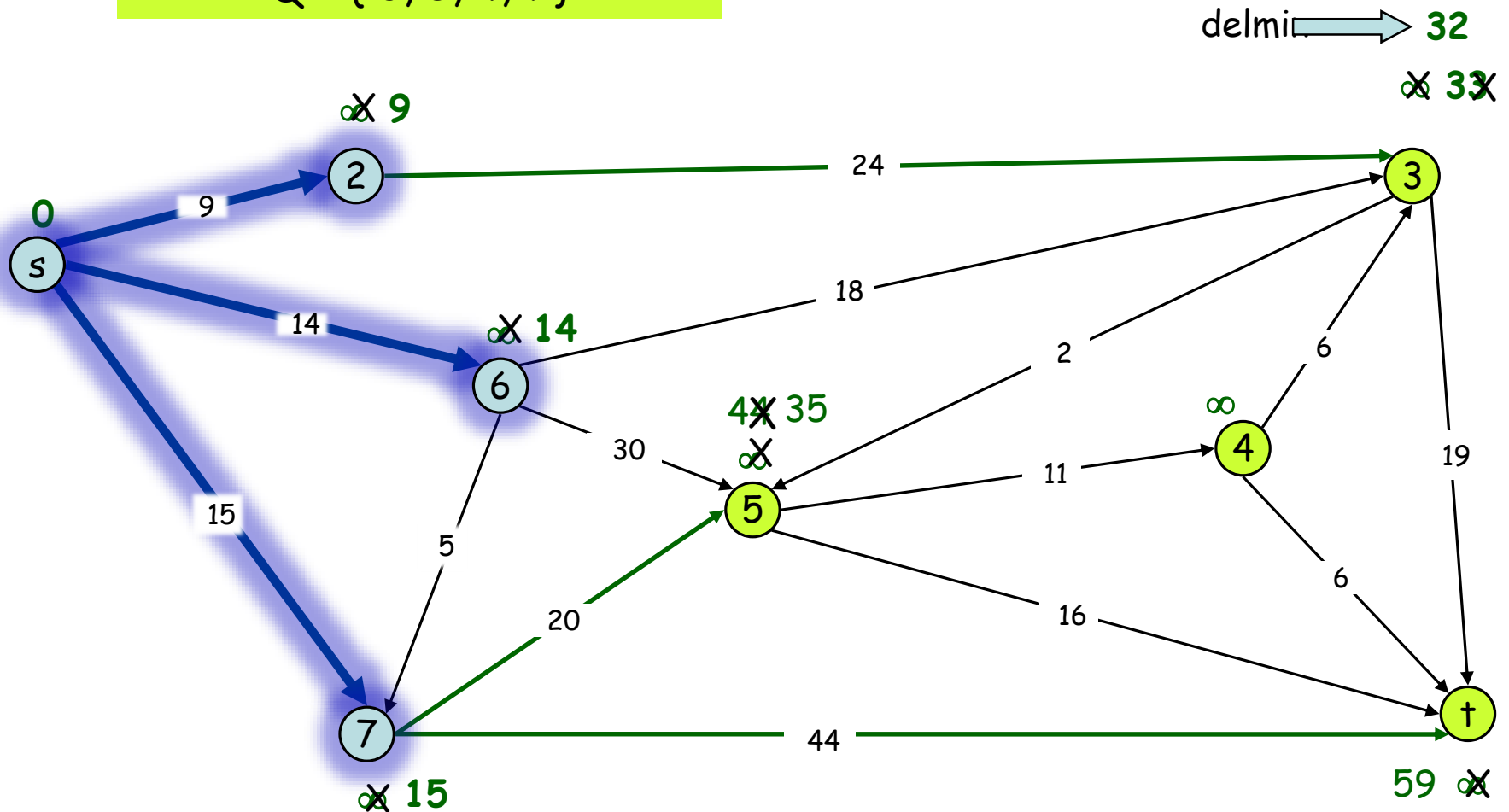




# Example

$S = \{s, 2, 6, 7\}$   
 $Q = \{3, 5, 4, \dagger\}$

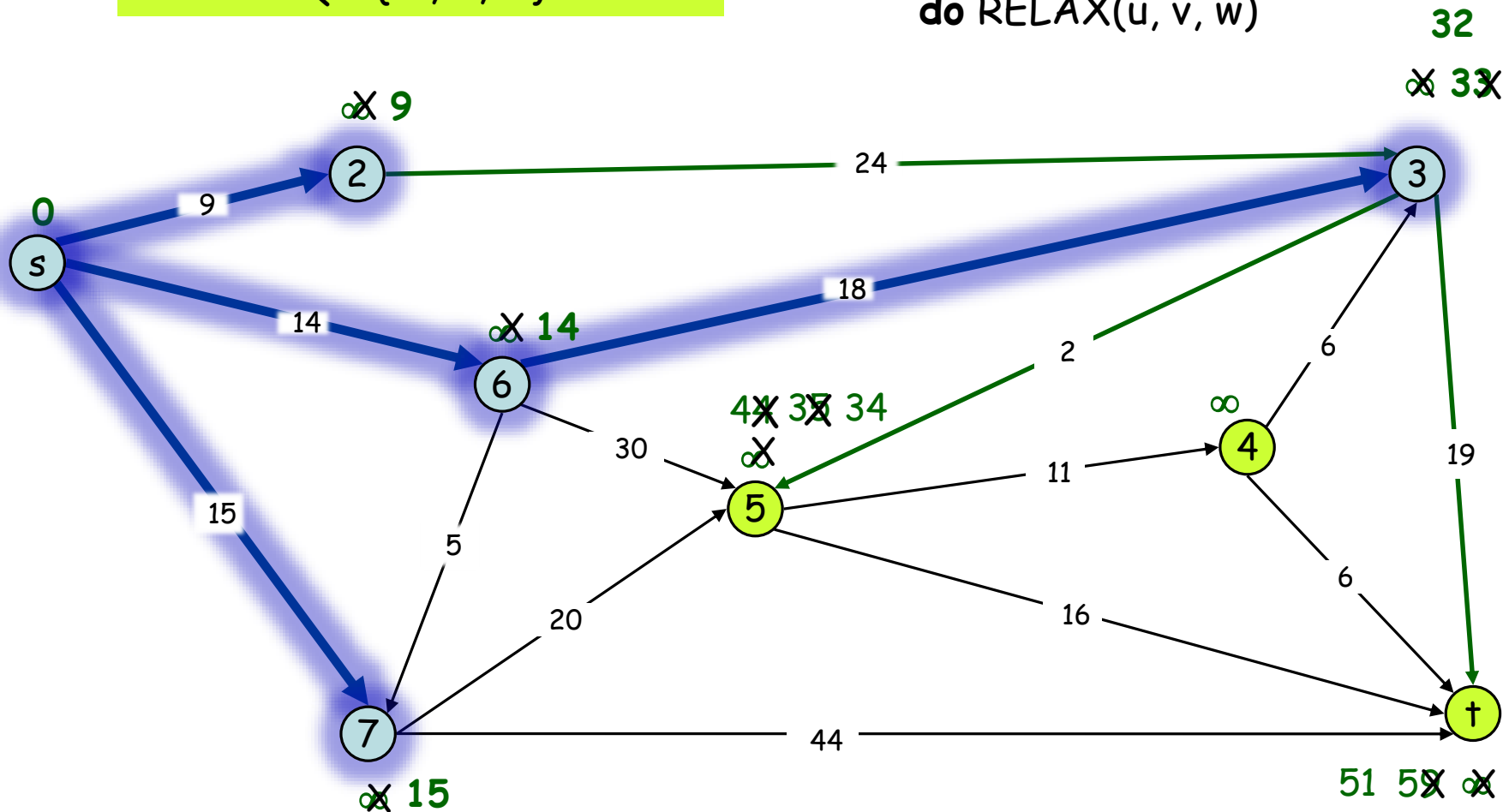
do  $u \leftarrow \text{EXTRACT-MIN}(Q)$



# Example

$S = \{s, 2, 3, 6, 7\}$   
 $Q = \{5, 4, \dagger\}$

$S \leftarrow S \cup \{u\}$   
 for each vertex  $v \in \text{Adj}[u]$   
 do RELAX( $u, v, w$ )

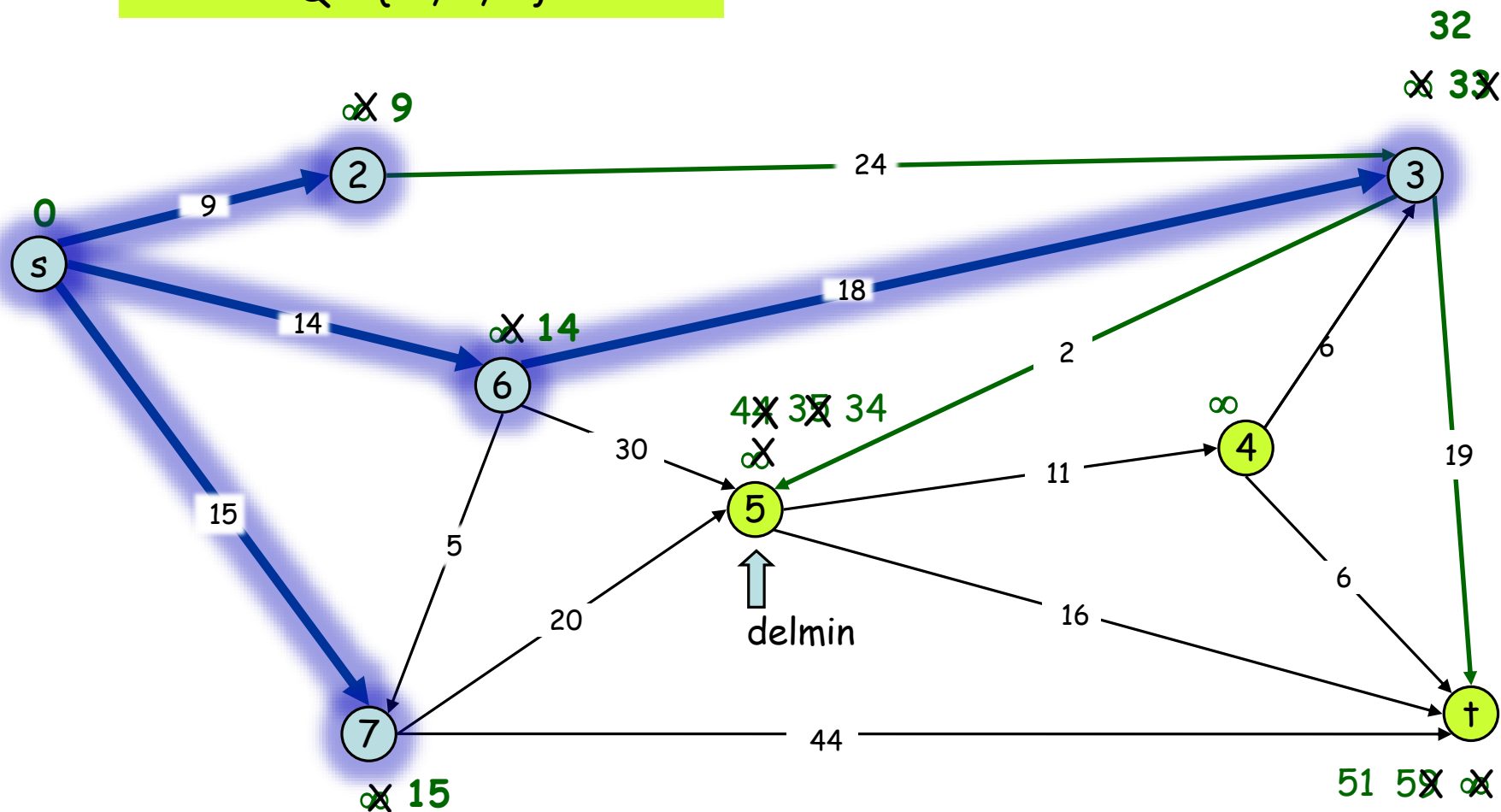


# Example

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

$Q = \{5, 4, \dagger\}$

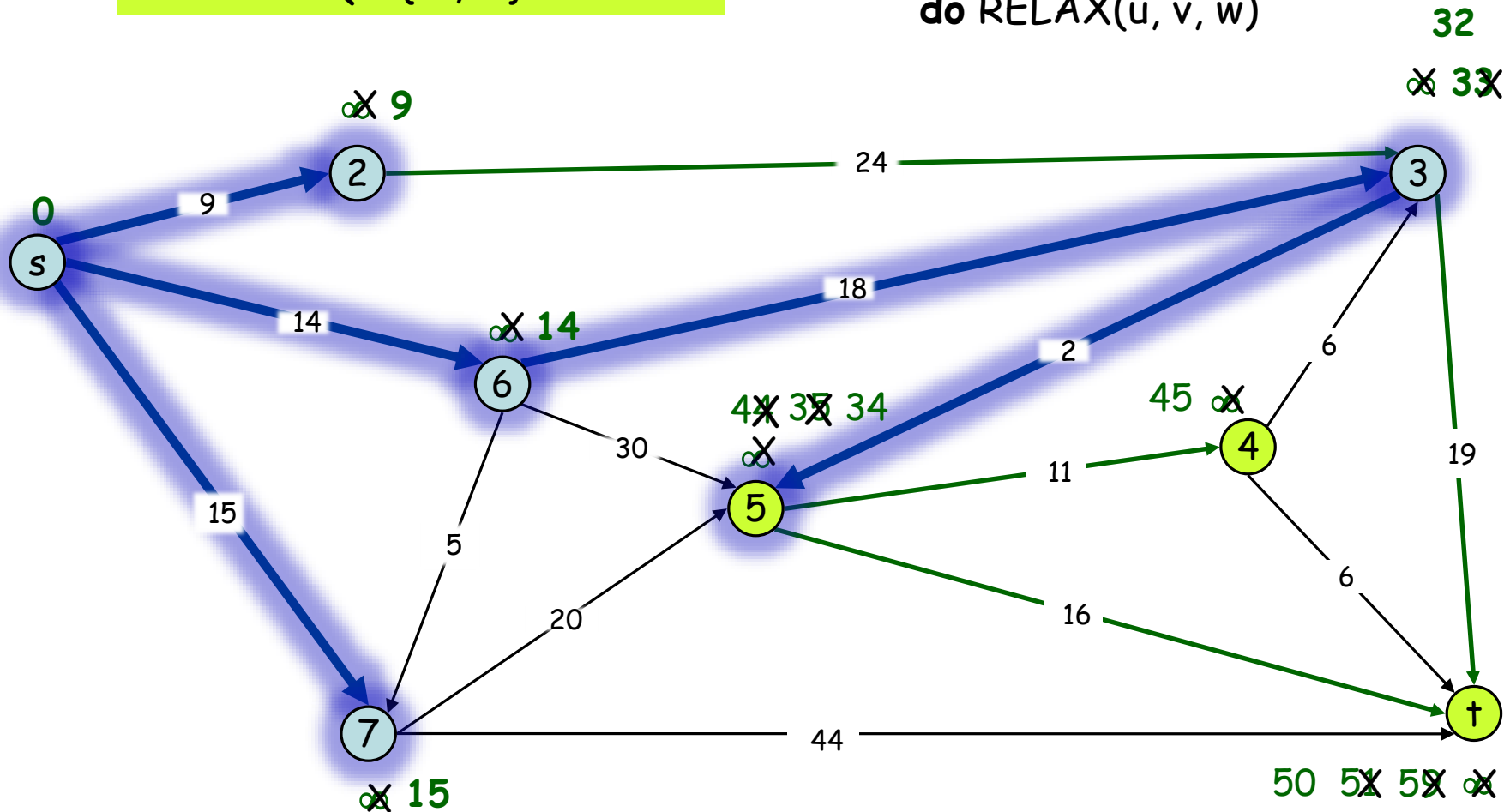
do  $u \leftarrow \text{EXTRACT-MIN}(Q)$



# Example

$S = \{s, 2, 3, 5, 6, 7\}$   
 $Q = \{4, \dagger\}$

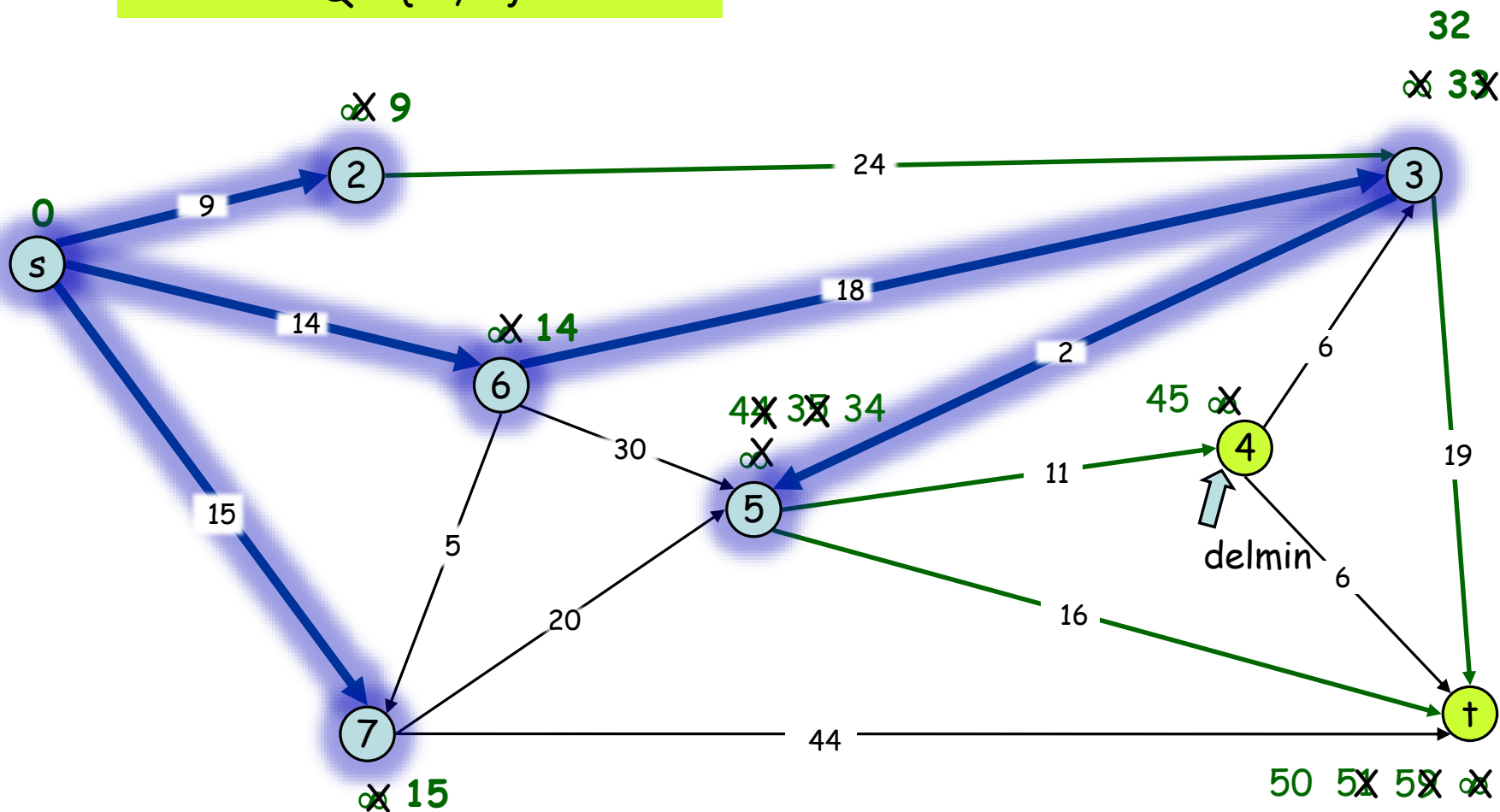
$S \leftarrow S \cup \{u\}$   
 for each vertex  $v \in \text{Adj}[u]$   
 do RELAX( $u, v, w$ )



# Example

$S = \{s, 2, 3, 5, 6, 7\}$   
 $Q = \{4, \dagger\}$

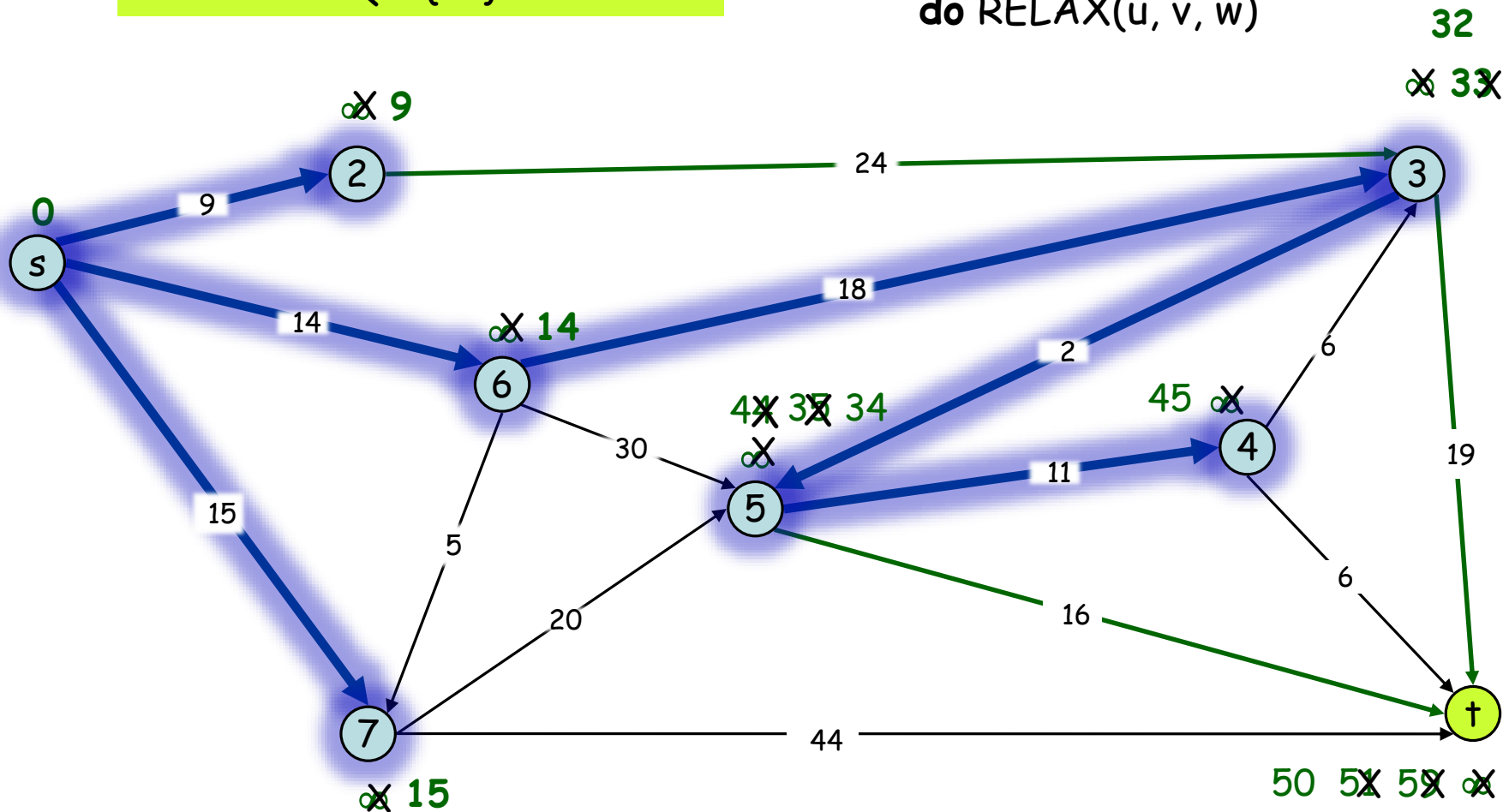
do  $u \leftarrow \text{EXTRACT-MIN}(Q)$



# Example

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

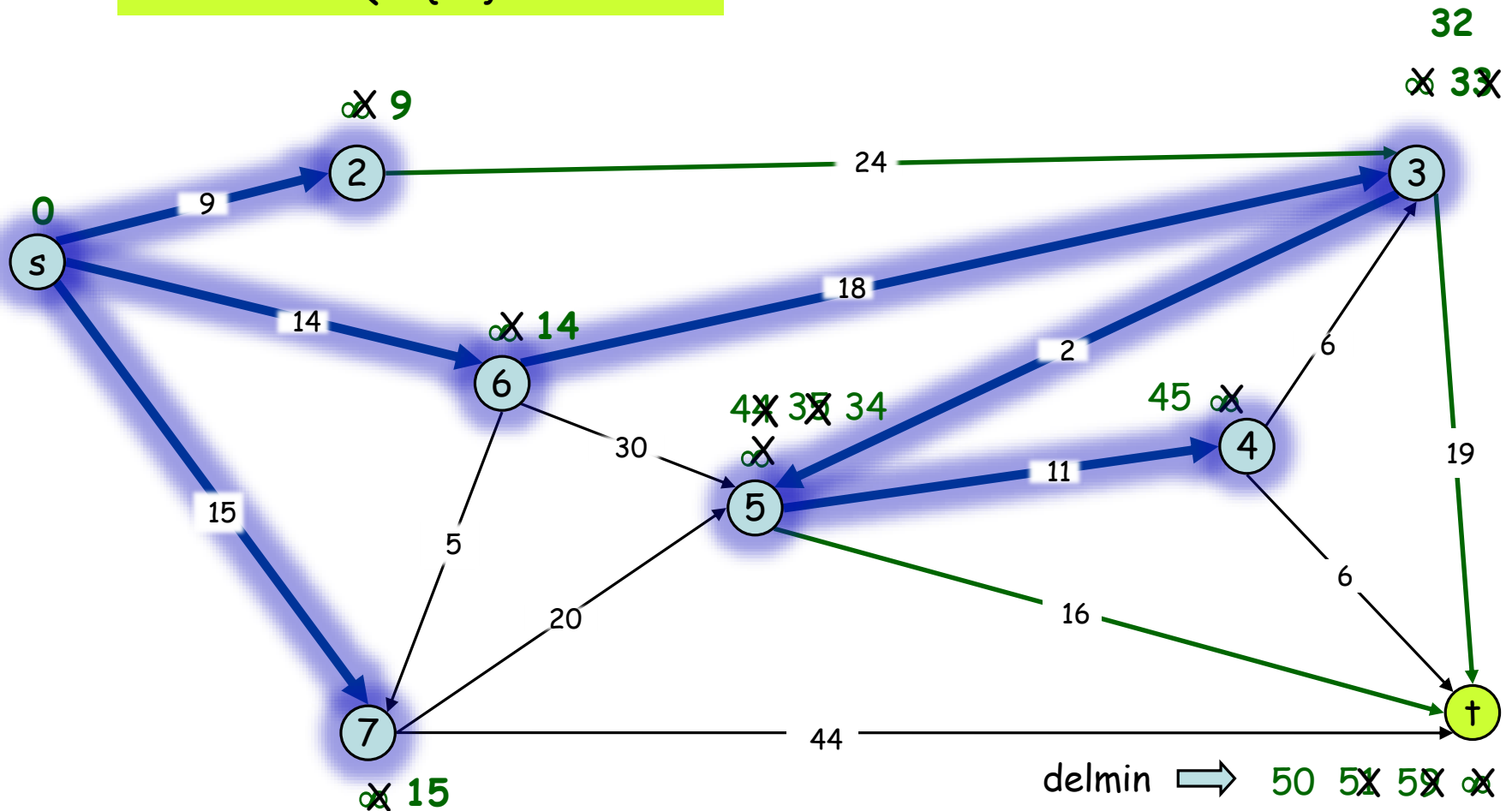
$S \leftarrow S \cup \{u\}$   
 for each vertex  $v \in \text{Adj}[u]$   
 do RELAX( $u, v, w$ )



# Example

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

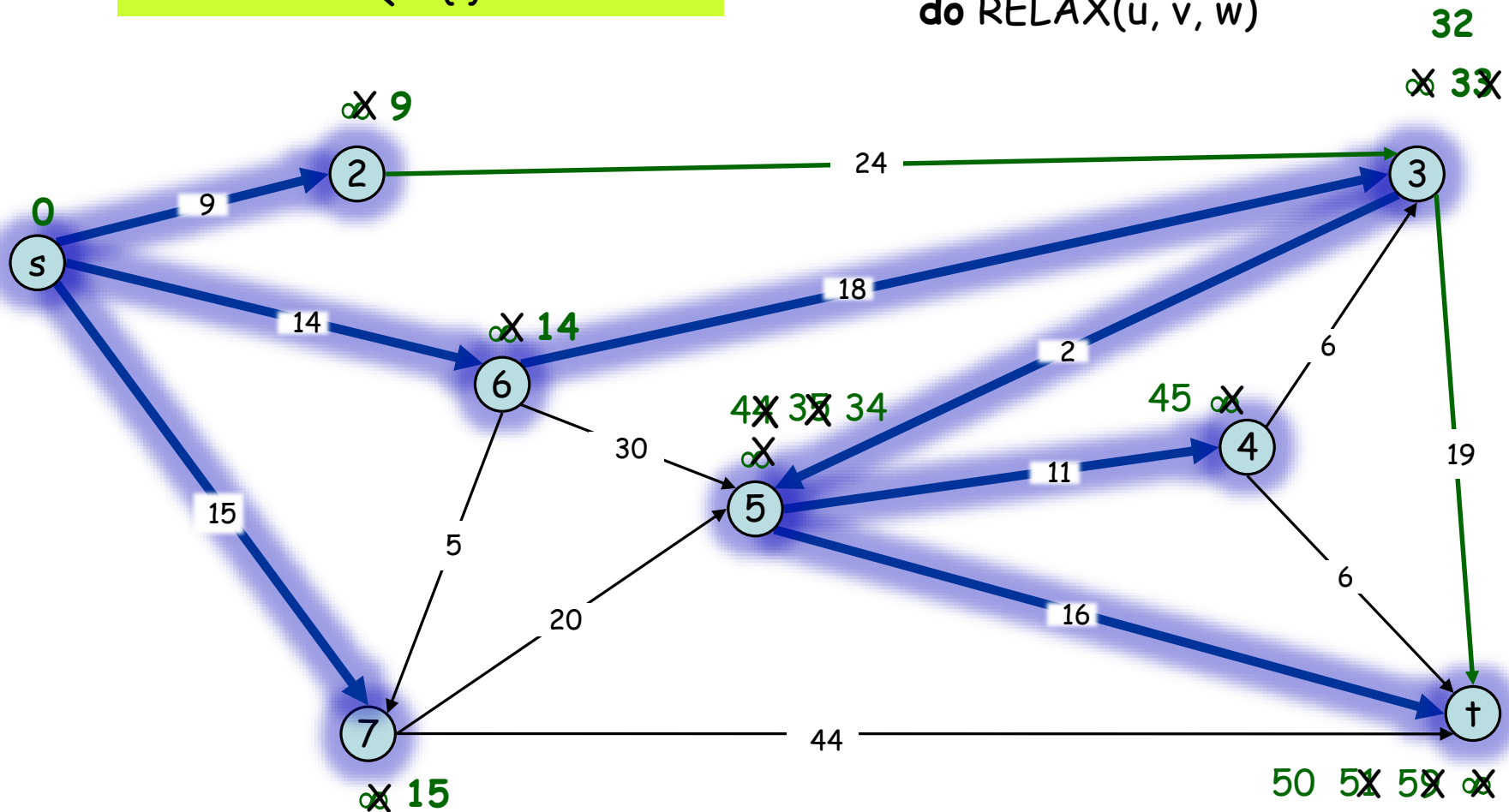
do  $u \leftarrow \text{EXTRACT-MIN}(Q)$



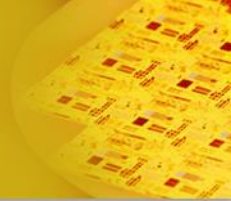
# Example

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

$S \leftarrow S \cup \{u\}$   
 for each vertex  $v \in \text{Adj}[u]$   
 do RELAX( $u, v, w$ )



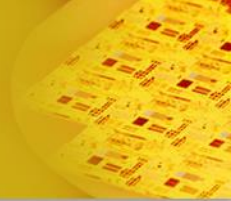




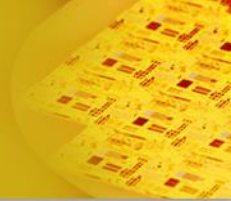
## Theorem 24.6

loop invariant : At the start of each iteration of the  
while loop of lines 4-8,  $d[v] = \delta(s, v)$   
for each vertex  $v \in S$ .

Proof at page 660 ~ 661



- Like Prim's algorithm, performance depends on implementation of priority queue.
  - Binary heap :
    - Each operation takes  $O(\lg V)$  time  
→  $O(E \lg V)$
  - Fibonacci heap :
    - $O(V \lg V + E)$  time.



# All Pairs Shortest Path

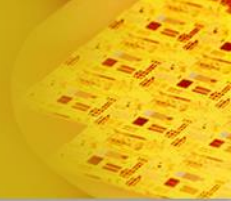
# Floyd-Warshall Algorithm

- The easiest way!
    - Iterate Dijkstra's and Bellman-Ford  $|V|$  times!
  - Dijkstra:
    - $O(V \lg V + E)$   $\rightarrow$   $O(V^2 \lg V + VE)$
  - Bellman-Ford:
    - $O(VE)$   $\rightarrow$   $O(V^2 E)$
- $O(V^2 \lg V + VE)$   
 $O(V^2 E)$

$\xrightarrow{\text{On dense graph}}$

$O(V^3)$   
 $O(V^4)$
- Faster-All-Pairs-Shortest-Paths (Ch 25.1):
    - $O(V^3 \lg V)$   $\rightarrow$  better than Dijkstra and Bellman-Ford ?
  - Any other faster algorithms?
    - Floyd-Warshall Algorithm

# Floyd-Warshall Algorithm



- Negative edges are allowed
- Assume that no negative-weight cycle
- Dynamic Programming Solution
  - Optimal substructure

# The structure of a shortest path

- Intermediate vertex
  - In simple path  $p = \langle v_1, \dots, v_L \rangle$ , any vertex of  $p$  other than  $v_1$  and  $v_L$ , i.e., any vertex in the set  $\{v_2, \dots, v_{L-1}\}$ .
- Key Observation
  - For any pair of vertices  $i, j$  in  $V$ .
  - Let  $p$  be a **minimum-weight path** of all paths from  $i$  to  $j$  whose **intermediate vertices are all from  $\{1, 2, \dots, k\}$** .
  - Assume that we have all shortest paths from  $i$  to  $j$  whose **intermediate vertices are from  $\{1, 2, \dots, k-1\}$** .
  - Observe relationship between path  $p$  and above shortest paths.

# Key Observation

- A shortest path does not contain the same vertex twice.
  - Proof: A path containing the same vertex twice contains a cycle. Removing cycle give a shorter path.

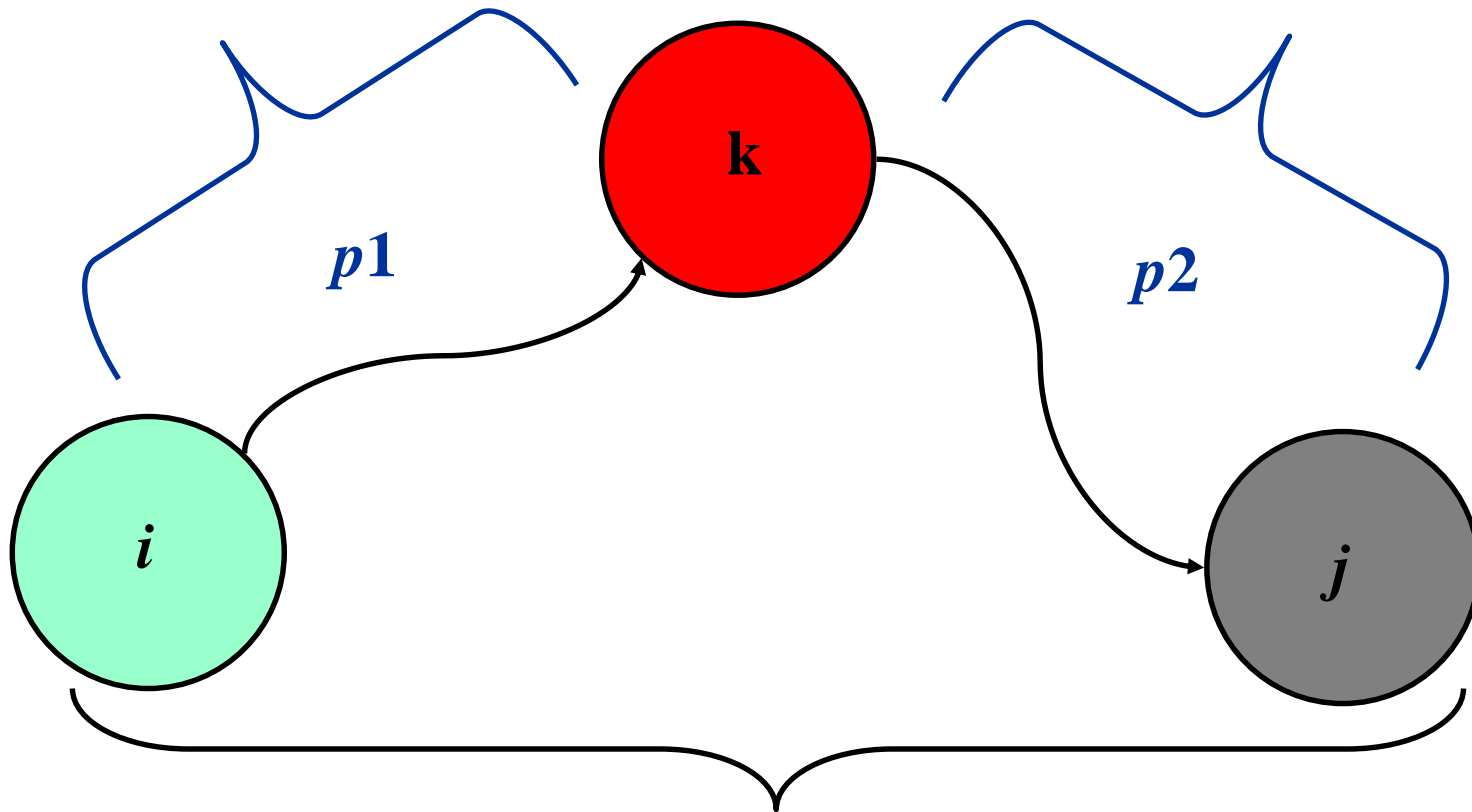
# Key Observation

- $p$  is determined by the shortest paths whose intermediate vertices from  $\{1, \dots, k-1\}$ .
- Case1: If  $k$  is not an intermediate vertex of  $p$ .
  - Path  $p$  is the shortest path from  $i$  to  $j$  with intermediates from  $\{1, \dots, k-1\}$ .
- Case2: If  $k$  is an intermediate vertex of path  $p$ .
  - Path  $p$  can be broken down into  $i \xrightarrow{p_1} k \xrightarrow{p_2} j$ .
  - $p_1$  is the shortest path from  $i$  to  $k$  with all intermediate vertices in the set  $\{1, 2, \dots, k\}$  or  $\{1, 2, \dots, k-1\}$ .
  - $p_2$  is the shortest path from  $k$  to  $j$  with  $\{1, 2, \dots, k\}$  or  $\{1, 2, \dots, k-1\}$ .



# Key Observation

$p1$ : All intermediate vertices in  $\{1, 2, \dots, k-1\}$    
  $p2$ : All intermediate vertices in  $\{1, 2, \dots, k-1\}$



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

# A recursive solution

- Let  $d_{ij}^{(k)}$  be the **length of the shortest path** from  $i$  to  $j$  such that all intermediate vertices on the path are in set  $\{1, 2, \dots, k\}$ .
- Let  $D^{(k)}$  be the  $n \times n$  matrix  $[d_{ij}^{(k)}]$ .
- $d_{ij}^{(0)}$  is set to be  $w_{ij}$  (no intermediate vertex).
- $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \quad (k \geq 1)$
- $D^{(n)} = (d_{ij}^{(n)})$  gives the final answer, for all intermediate vertices are in the set  $\{1, 2, \dots, n\}$ .

# A recursive solution

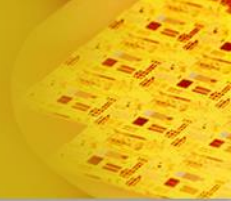
- $$d_{ij}^{(k)} = \begin{cases} w_{ij} & (\text{if } k=0) \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & (\text{if } k \geq 1) \end{cases}$$

- The Matrix  $D^{(n)} = (d_{ij}^{(n)})$  gives the final answer:  
 $d_{ij}^{(n)} = \delta(i,j)$  for all  $i, j \in V$ .

# Extracting the Shortest Paths

- The predecessor pointers  $pred[i,j]$  can be used.
- Initially all  $pred[i,j] = \text{nil}$
- Whenever the shortest path from  $i$  to  $j$  passing through an intermediate vertex  $k$  is discovered, we set  $pred[i,j] = k$

# Extracting the Shortest Paths



- Observation:
  - If  $pred[i,j] = \text{nil}$ , shortest path does not exist.
  - If there exists shortest path and the shortest path does not pass through any intermediate vertex, then  $pred[i,j] = i$ .
  - If  $pred[i,j] = k$ , vertex  $k$  is an intermediate vertex on shortest path from  $i$  to  $j$

# Extracting the Shortest Paths

- How to find?
  - If  $pred[i,j] = i$ , the shortest path is edge  $(i,j)$
  - Otherwise, recursively compute  $(i, pred[i,j])$  and  $(pred[i,j], j)$

- The Floyd-Warshall Algorithm: Version 1

**Floyd-Warshall( $w, n$ )**

```
{ for  $i = 1$  to  $n$  do
```

**initialize**

See figure 25.4.

```
    for  $j = 1$  to  $n$  do
```

```
        {  $D^0[i, j] = w[i, j]$  ;
```

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

Can you find a shortest path from vertex 1 to vertex 2 from the  $\pi$  matrix?

```
    for  $k = 1$  to  $n$  do
```

**dynamic programming**

```
        for  $i = 1$  to  $n$  do
```

```
            for  $j = 1$  to  $n$  do
```

```
                if ( $d^{(k-1)}[i, k] + d^{(k-1)}[k, j] < d^{(k-1)}[i, j]$ )
```

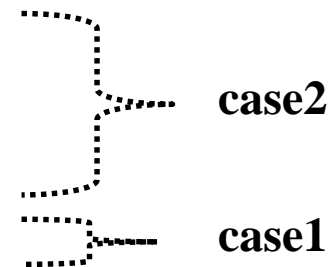
```
                    {  $d^{(k)}[i, j] = d^{(k-1)}[i, k] + d^{(k-1)}[k, j]$ ;
```

```
                    pred[i, j] = k;
```

```
                    else  $d^{(k)}[i, j] = d^{(k-1)}[i, j]$ ;
```

```
    return  $d^{(n)}[1..n, 1..n]$ ;
```

```
}
```



# Analysis

- Running time is clearly  $\Theta(?)$
- $\Theta(n^3) \rightarrow \Theta(|V|^3)$
- Faster than previous algorithms.  
 $O(|V|^4), O(|V|^3 \lg |V|)$
- Problem: Space Complexity  $\Theta(|V|^3)$ .
- It is possible to reduce this down to  $\Theta(|V|^2)$  by keeping only one matrix instead of  $n$ .



# Transitive Closure

- Given directed graph  $G = (V, E)$
- Compute  $G^* = (V, E^*)$
- $E^* = \{(i, j) : \text{there is path from } i \text{ to } j \text{ in } G\}$
- Could assign weight of 1 to each edge, then run FLOYD-WARSHALL
- If  $d_{ij} < n$ , then there is a path from  $i$  to  $j$ .
- Otherwise,  $d_{ij} = \infty$  and there is no path.

# Transitive Closure – Warshall

- Using logical operations  $\vee$  (OR),  $\wedge$  (AND)
- Assign weight of 1 to each edge, then run FLOYD-WARSHALL with this weights.
- Instead of  $D^{(k)}$ , we have  $T^{(k)} = (t_{ij}^{(k)})$ 
  - $t_{ij}^{(0)} = \begin{cases} 0 & (\text{if } i \neq j \text{ and } (i, j) \notin E) \\ 1 & (\text{if } i = j \text{ or } (i, j) \in E) \end{cases}$
  - $t_{ij}^{(k)} = \begin{cases} 1 & (\text{if there is a path from } i \text{ to } j \text{ with all intermediate vertices in } \{1, 2, \dots, k\}) \\ (t_{ij}^{(k-1)} \text{ is } 1) \text{ or } (t_{ik}^{(k-1)} \text{ is } 1 \text{ and } t_{kj}^{(k-1)} \text{ is } 1) \\ 0 & (\text{otherwise}) \end{cases}$

# Transitive Closure

TRANSITIVE-CLOSURE( $E, n$ )

for  $i = 1$  to  $n$

do for  $j = 1$  to  $n$

do if  $i=j$  or  $(i, j) \in E$

then  $t_{ij}^{(0)} = 1$

else  $t_{ij}^{(0)} = 0$

for  $k = 1$  to  $n$

do for  $i = 1$  to  $n$

do for  $j = 1$  to  $n$

do  $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee ( t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} )$

return  $T^{(n)}$