EEL 4837
Programming for Electrical Engineers II

Ivan Ruchkin
Assistant Professor
Department of Electrical and Computer Engineering
University of Florida at Gainesville
iruchkin@ece.ufl.edu
http://ivan.ece.ufl.edu

Dijkstra’s Algorithm

Readings:
- Weiss 9.3.2
- Cormen 24.1–24.3

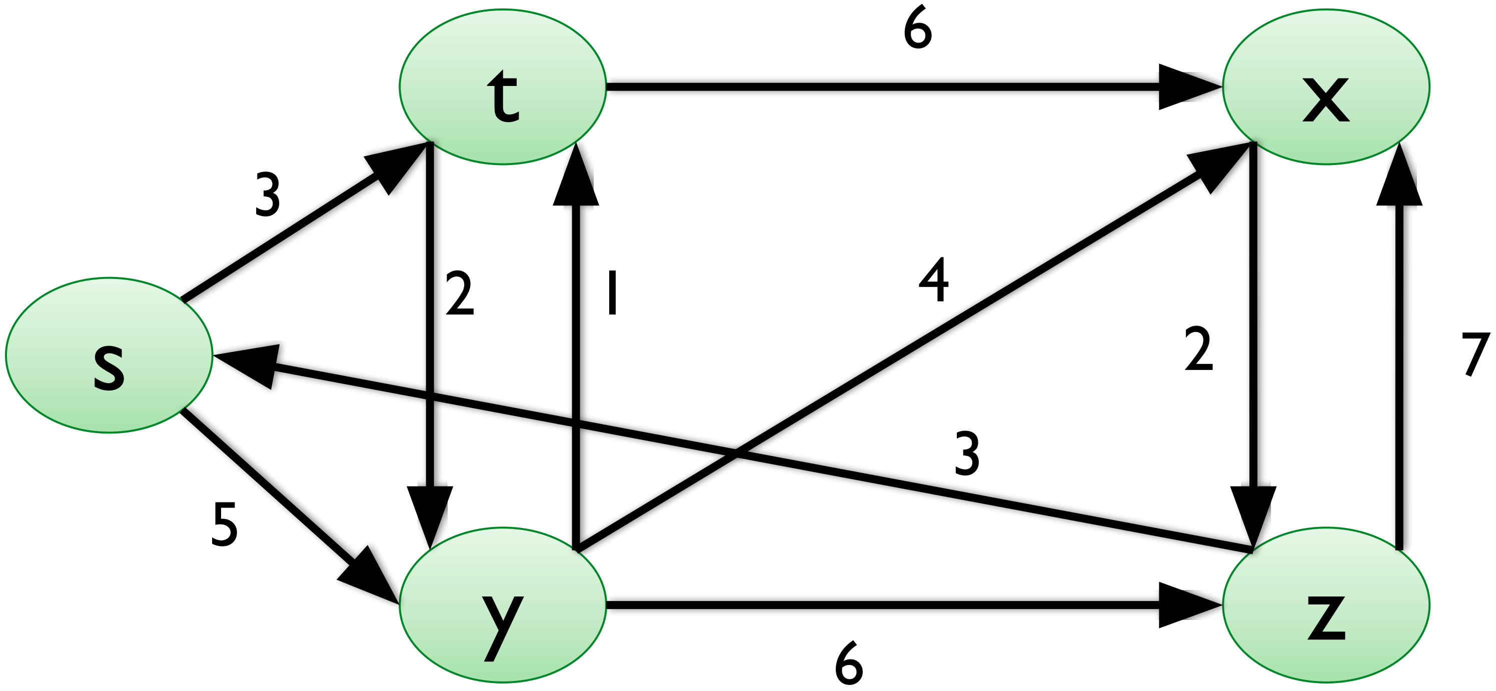
Shortest Path in a Weighted Graph

Given:

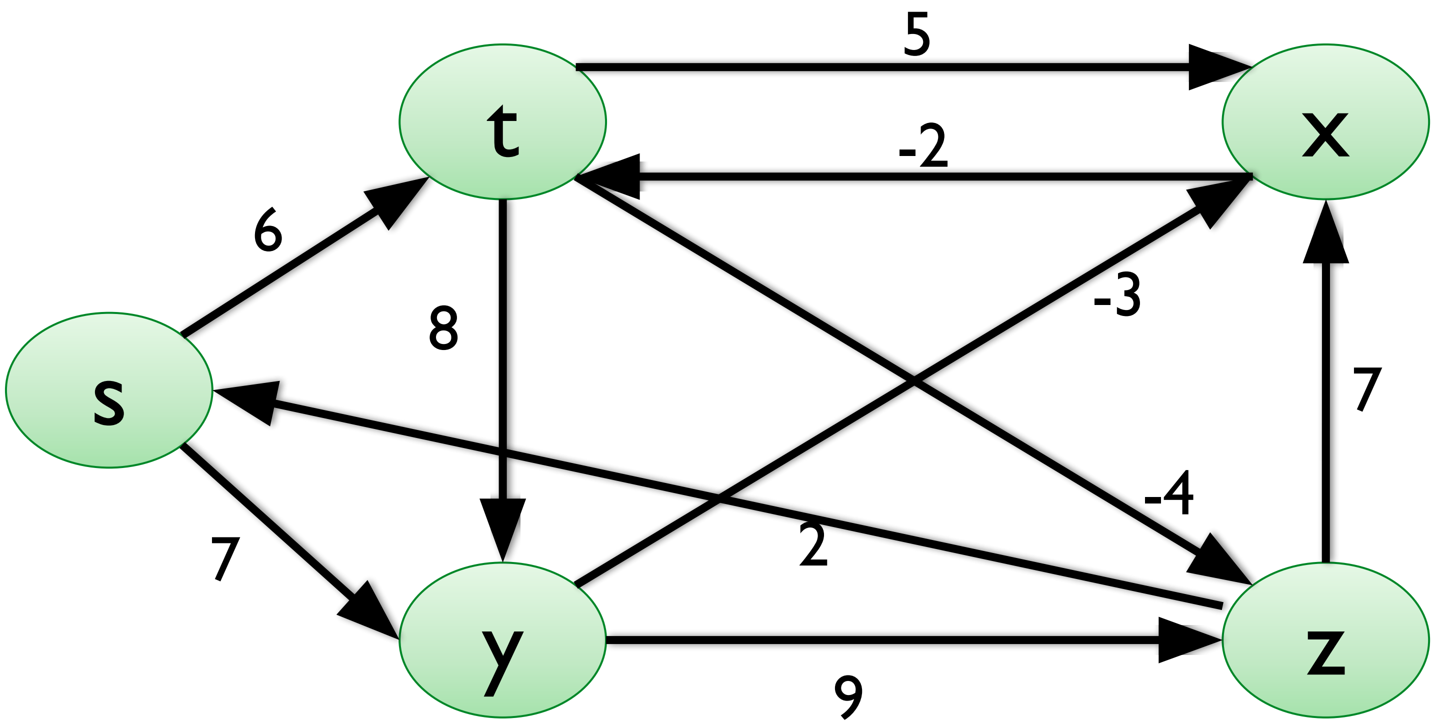
- A **weighted** directed graph
- Two nodes s and t in the graph

Find: the **shortest path** from s to t and its **total weight**

Example: Positive Weights



Example: Negative Weights



Shortest Path Problem

- Given a weighted, directed graph $G = (V, E)$, with weight function $w: E \to \mathbb{R}$. The **weight** $w(p)$ of a path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges
- $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- **Shortest-path weight** $\delta(u, v)$ from u to v is
- $\delta(u,v) = \begin{cases} \min\left\{w(p): u \xrightarrow{p} v\right\} & \text{if there is a path from } u \text{ to } v \\ \infty & otherwise \end{cases}$
- The **shortest path** is any path with shortest path weight

Problem Variants

- Single-source single-destination shortest path
- Single-source all-destinations shortest paths
- All-sources single-destination shortest paths
- All-pairs shortest paths

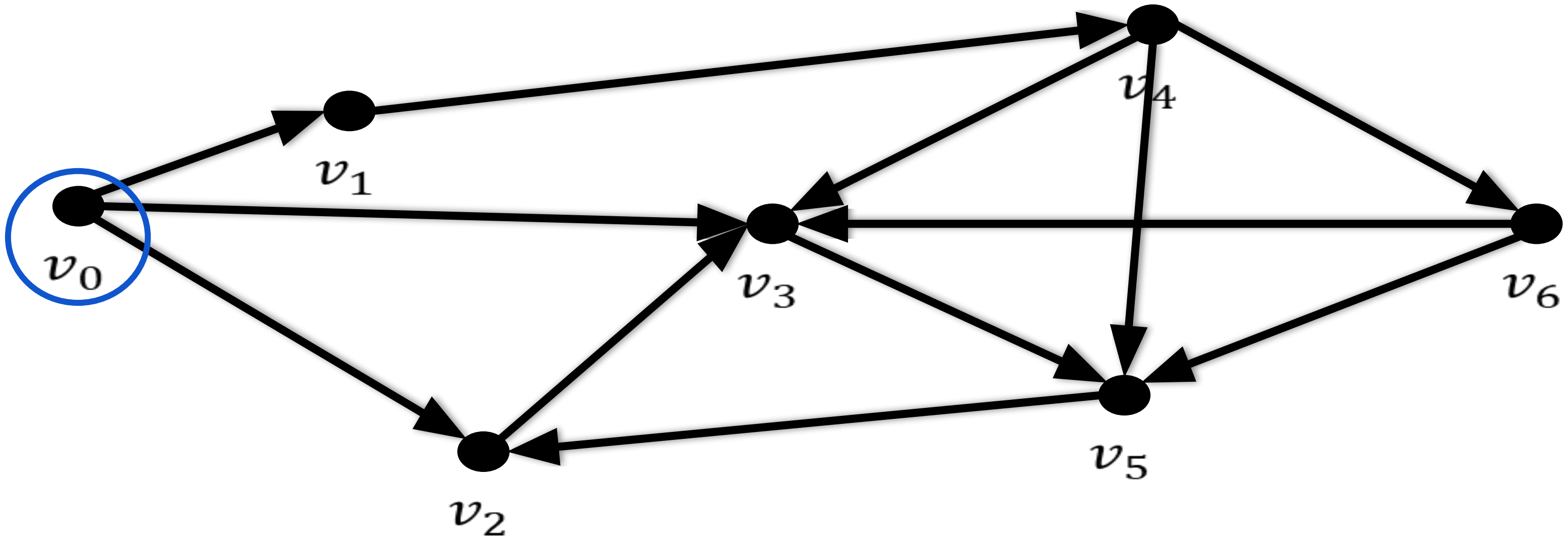
Cycles in Shortest Path Problems

- *Negative weight cycles?*
  - The shortest path is **undefined**
- *Positive weight cycles?*
  - Can be **removed** to produce a shorter path
- *Zero-weight cycles?*
  - Can be **removed** without changing the shortest path
- **Conclusion:** we can disregard cycles in our solutions

Shortest Path Representation

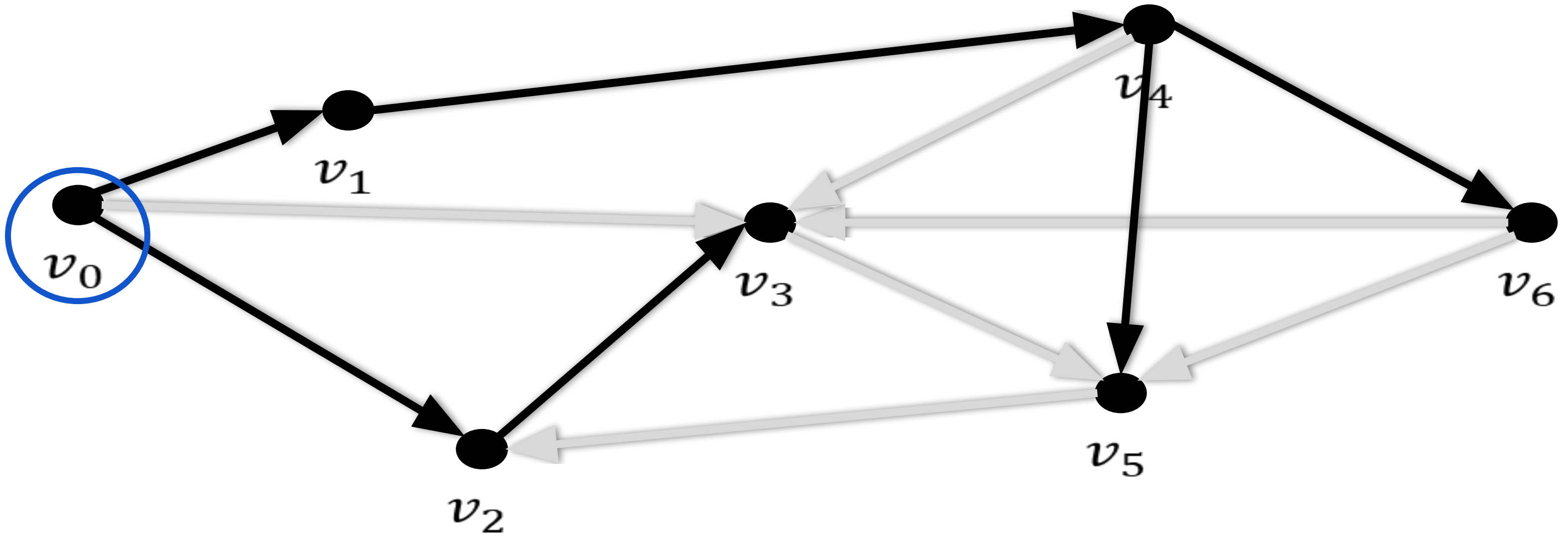
- For single-source all-destinations shortest path

Issue: storing all paths explicitly is *expensive*!



Shortest Path Representation

- For single-source all-destinations shortest path



- For each vertex v_i, we store its predecessor $v_i.\pi$ in the shortest-path tree and the cost of the shortest path $v_i.d = \delta(s, v_i)$

The best direction to arrive from, when starting in v_0

- Stores |V| shortest paths in O(|V|) memory

Reminder: BFS

Problem: single-source all-destinations **unweighted** shortest path

```
BFS(G,s)
for u ∈ G.V do
   u.color := white
   u.dist := ∞
   u.pred := NULL
s.color := gray
s.dist := 0
Q := new Queue()    // FIFO queue
Q.enqueue(s)
while not Q.isEmpty() do
   u := Q.dequeue()
   for v ∈ u.adj do
      if v.color = white
        then v.color := gray
             v.dist := u.dist + 1
             v.pred := u
             Q.enqueue(v)
```

Initialize all vertices

Initialize BFS with *s*

Handle all of *u*'s children before handling children of children

- A vertex is white if it is undiscovered
- A vertex is gray if it has been discovered but not all of its edges have been explored

Dijkstra's Algorithm: Intuition

Problem: single-source all-destinations **weighted** shortest path

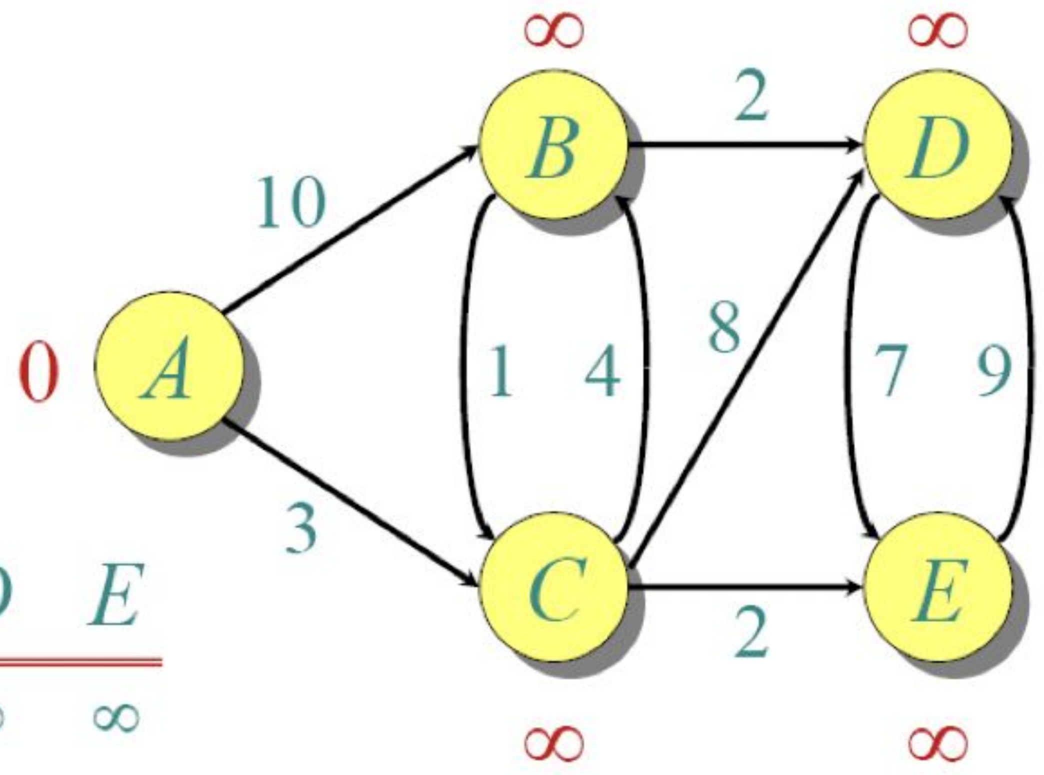
- All edge weights have to be *non-negative*

Ideas:

- Build on the BFS for unweighted shortest path
  - Keep track of the nodes to visit in queue Q
  - Keep track of which nodes have been visited in set S
- Save the best path length to each node so far with `dist[v]`
- Always visit the lowest-weight unvisited node next
  - Makes Q a **priority queue**, with edge weight as a priority
  - When we visit the node, we are *guaranteed to know* its shortest path length

Dijkstra's Algorithm: Example

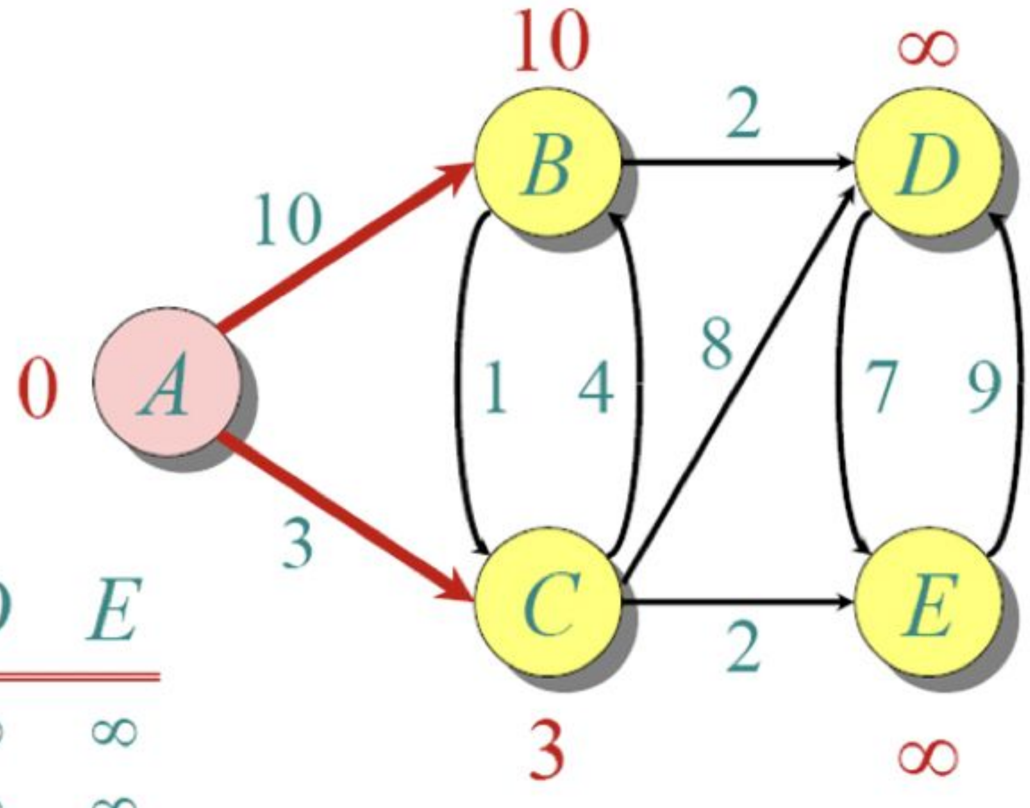
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

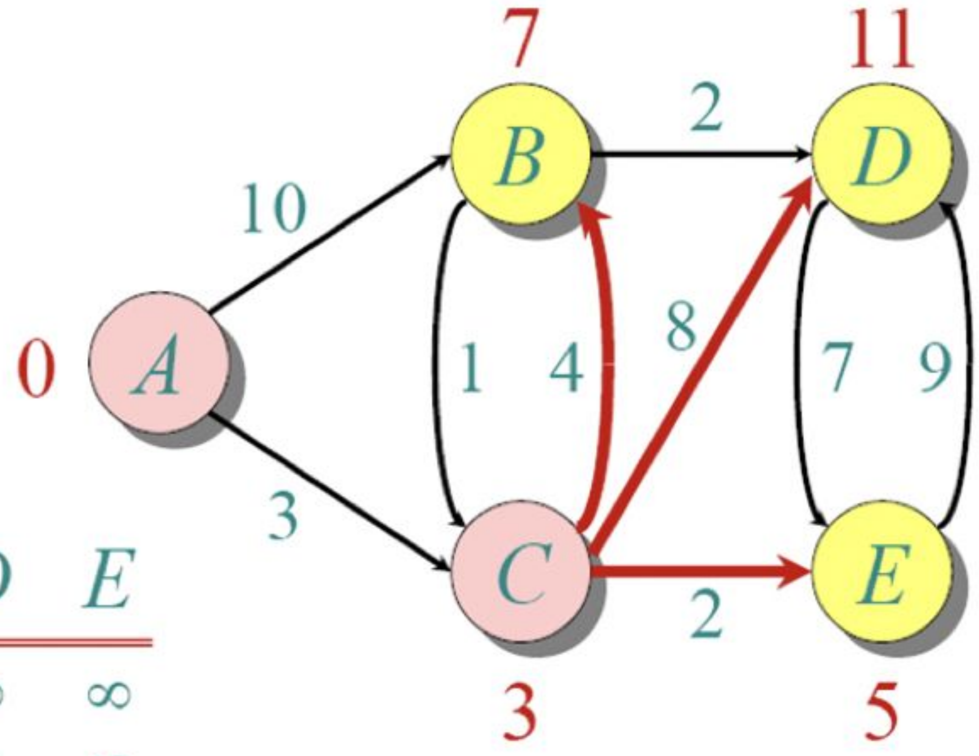
Dijkstra’s Algorithm: Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

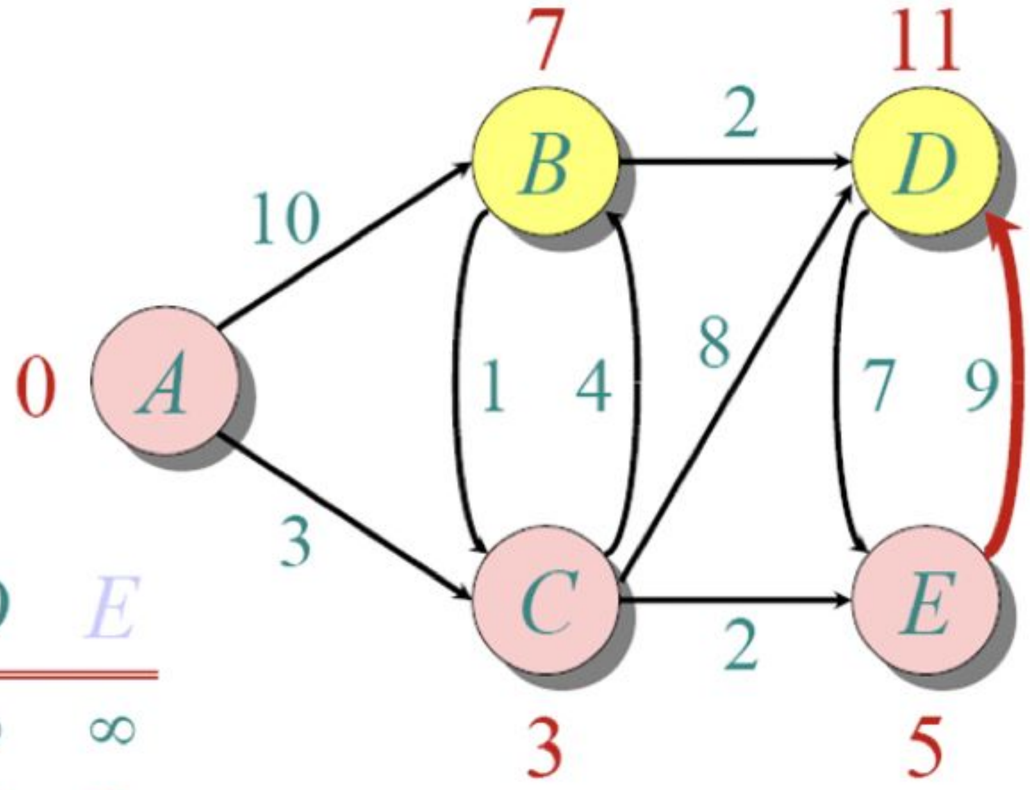
Dijkstra’s Algorithm: Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }

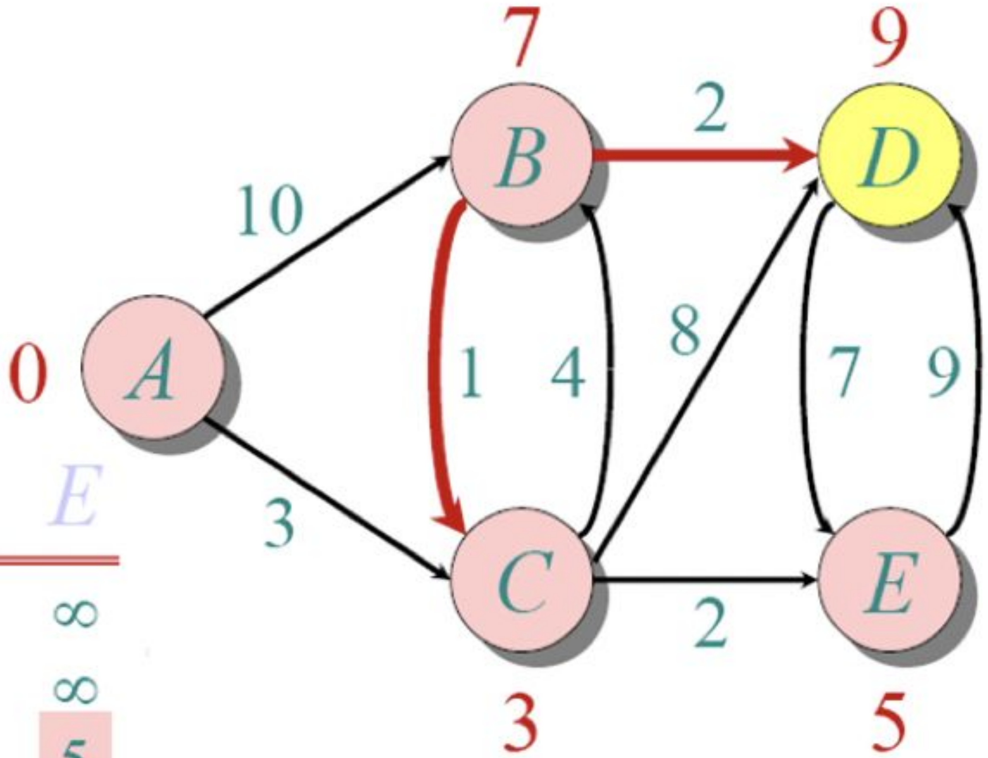
Dijkstra’s Algorithm: Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

Dijkstra's Algorithm: Example

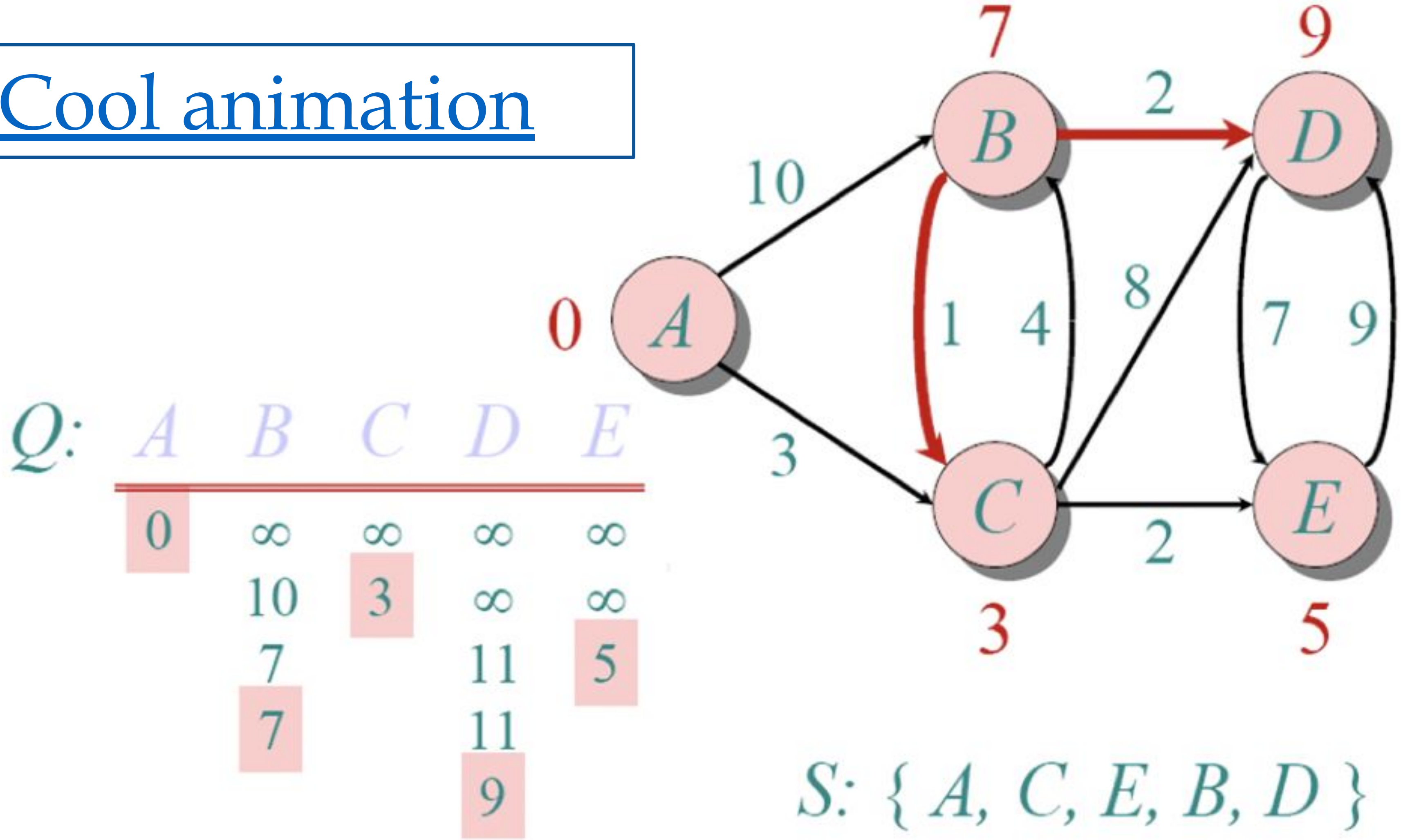


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B* }

Dijkstra's Algorithm: Example

Cool animation



Dijkstra's Algorithm

dist[s] ←0	(distance to source vertex is zero)
for all v ∈ V–{s}	
do dist[v] ←∞	(set all other distances to infinity)
S←∅	(S, the set of visited vertices is initially empty)
Q←V	(Q, the queue initially contains all vertices)
while Q ≠∅	(while the queue is not empty)
do u ← mindistance(Q,dist)	(select the element of Q with the min. distance)
S←S∪{u}	(add u to list of visited vertices)
for all v ∈ neighbors[u]	
do if dist[v] > dist[u] + w(u, v)	(if new shortest path found)
then d[v] ←d[u] + w(u, v)	(set new value of shortest path)
(if desired, add traceback code)	//pred[v] = u
return dist	

Question: what if we didn't pick the **min-distance node** from Q?
(e.g., going to a random node from Q instead)

Dijkstra's Algorithm: Analysis

```
dist[s] ←0                                      (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ←∞                          (set all other distances to infinity)
S←∅                                             (S, the set of visited vertices is initially empty)
Q←V                                             (Q, the queue initially contains all vertices)
while Q ≠∅                                      (while the queue is not empty)
do   u ← mindistance(Q,dist)                    (select the element of Q with the min. distance)
      S←S∪{u}                                   (add u to list of visited vertices)
       for all v ∈ neighbors[u]
              do  if   dist[v] > dist[u] + w(u, v)           (if new shortest path found)
                        then      d[v] ←d[u] + w(u, v)       (set new value of shortest path)
                          (if desired, add traceback code)  //pred[v] = u
return dist
```

Complexity analysis: Dijkstra does |V| insertions, |V| removals, and |E| key-updates

- How is the *priority queue* implemented?
- **Unsorted array:** O(1) insertion, O(1) key-update, O(|V|) removal -> O(V^2)
- **Binary heap:** O(log|V|) insertion, O(log|V|) key-update, and O(log|V|) removal
  -> O((|V|+|E|)*log|V|)

Greedy Algorithms: Design Technique

- Dijkstra's algorithm is an example of a **greedy algorithm**
  - It goes "greedily" towards the low-cost path
- **Greedy choice:** in each step, make a decision that appears the best
  - Disregard long-term consequences
  - Often, it is a good heuristic: *local optimum* leads to *global optimum*
- **Example problem:** fewest bills/coins to represent some amount of $
  - E.g., $1.58 = $1 + 2x$0.25 + 3x$0.01
  - **Greedy heuristic:** pick max number of the largest denomination; move on to a lower denomination. This always works for the USD denominations.
  - Sometimes greedy algorithms do **not** deliver optimal results
  - **Example:** count 15 cents if 12-cent coins existed: 12+1+1+1 is *worse* than 10+5