Shortest Path

Dr. Animesh Chaturvedi

Assistant Professor at Data Science and Artificial Intelligence Department,

IIIT Dharwad

Post Doctorate: King's College London & The Alan Turing Institute

PhD: IIT Indore MTech: IIITDM Jabalpur

Shortest Path

- Given a road map of the India on which the distance between each pair of adjacent intersections is marked between route from Delhi to Mumbai, how can you find the shortest possible route?
- To examine an enormous number of possibilities, most of which are simply not worth considering!
- Model the road map as a graph vertices represent intersections, edges represent road segments between intersections, and edge weights represent road distances.
- For other examples weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest Path

- In a ***shortest-paths problem***, given a weighted directed graph G = (V, E) with edges mapped to real-valued weights.
- The ***weight*** w(p) of path p = <v_0, v_1, …., v_k> is the sum of the weights of its constituent edges:

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

We define the ***shortest-path weight*** $\delta(u, v)$ from u to v by

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

A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest Path Properties

- *Paths are directed.* A shortest path must respect the direction of its edges.
- *The weights are not necessarily distances.* Geometric intuition can be helpful, but the edge weights might represent time or cost.
- *Not all vertices need be reachable.* If t is not reachable from s, there is no path at all, and therefore there is no shortest path from s to t.
- *Negative weights introduce complications.*
- *Shortest paths are normally simple.*
- *Shortest paths are not necessarily unique.* There may be multiple paths of the lowest weight from one vertex to another; we are content to find any one of them.
- *Parallel edges and self-loops may be present.*

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Shortest Path: Variant

- **Single-destination shortest-paths problem:**
  - Find a shortest path to a given ***destination*** vertex *t* from each vertex *v*.
  - Reversing the direction of each edge → reduce this problem to a single-source problem.
- **Single-pair shortest-path problem:**
  - Find a shortest path from u to v for given vertices u and v, where source vertex is u
- **All-pairs shortest-paths problem:**
  - Find a shortest path from u to v for every pair of vertices u and v.
  - Solve this problem by running a single source algorithm once from each vertex.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest Paths Tree

- A ***shortest-paths tree*** rooted at s is a directed subgraph $G' = (V', E')$,

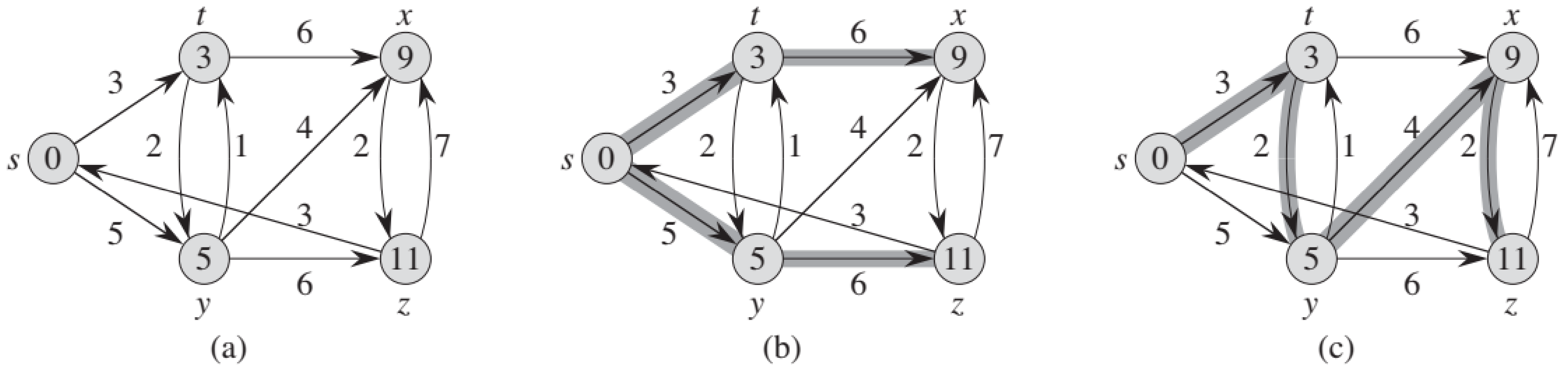
  where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest Paths Tree

- Shortest paths are not necessarily unique, and neither are shortest-paths trees.
- A weighted, directed graph and two shortest-paths trees with the same root.



Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest Path 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

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest Path Algorithms Initialization

- For each vertex v ∈ V, we maintain an attribute v.d, which is an upper bound on the weight of a shortest path from source s to v.
- We call v.d a shortest-path estimate.
- We initialize the shortest-path estimates and predecessors by the following O(V) time procedure:

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$

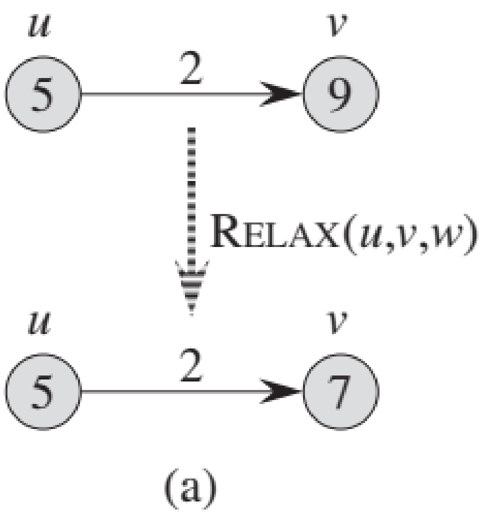
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest Path Algorithms Relaxation

Relaxing an edge (u, v) with weight w(u, v) = 2. The shortest-path estimate of each vertex appears within the vertex.

(a) Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases.

(b) Here, $v.d \le u.d + w(u, v)$ before relaxing the edge, and so the relaxation step leaves $v.d$ unchanged.



Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest Path Algorithms Relaxation

- Relaxation is the only means by which shortest path estimates and predecessors change.
- Algorithms differ in how many times they relax each edge and the order in which they relax edges.
- Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs relax each edge exactly once.
- The Bellman-Ford algorithm relaxes each edge $|V|-1$ times.

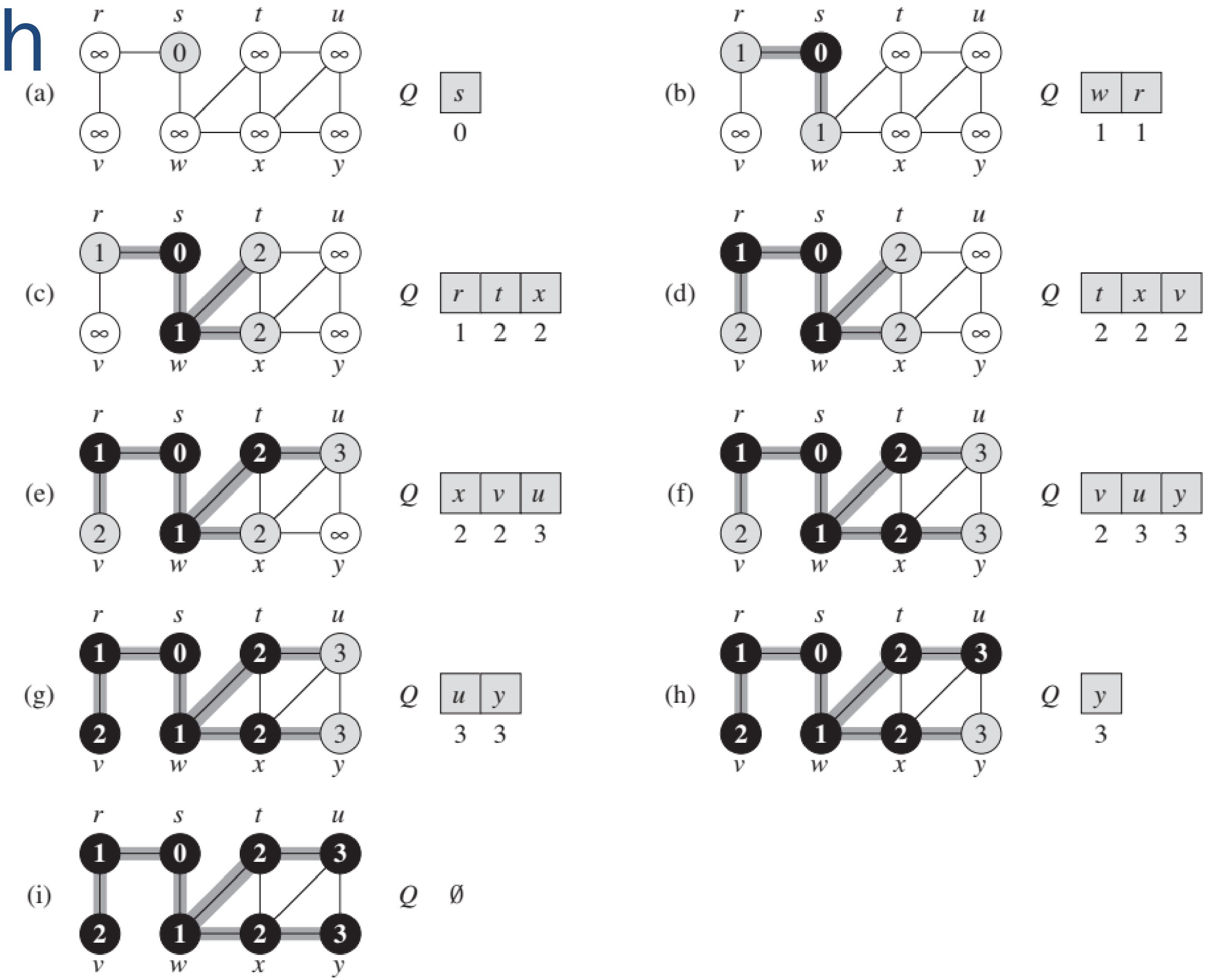
```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Breadth First Search

BFS algorithm is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge has unit weight.

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```



Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Bellman-Ford Algorithm

Bellman-Ford Algorithm

- Single-source shortest path problem
  - Computes d(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

- After initializing the d and π values of all vertices in line 1,
- The algorithm makes |V|–1 passes over the edges of the graph. Each pass is one iteration of the **for** loop of lines 2–4 and consists of relaxing each edge of the graph once.
- After making |V|–1 passes, lines 5–8 check for a negative-weight cycle and return the appropriate Boolean value.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Bellman-Ford Algorithm

- Runs in time O(VE),
- Initialization takes O(V) time,
- Each |V|−1 passes over the edges takes O(V) time
  - For loop takes O(E) time
    - Relax will take O(VE) times
- Running time:
  - O(V+VE+E) = O(VE)

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)          ← O(V)
for i = 1 to |G.V| − 1                  ← O(V)
    for each edge (u, v) ∈ G.E          ← O(E)
        RELAX(u, v, w)                  ← O(VE)
for each edge (u, v) ∈ G.E              ← O(E)
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Bellman-Ford Algorithm example

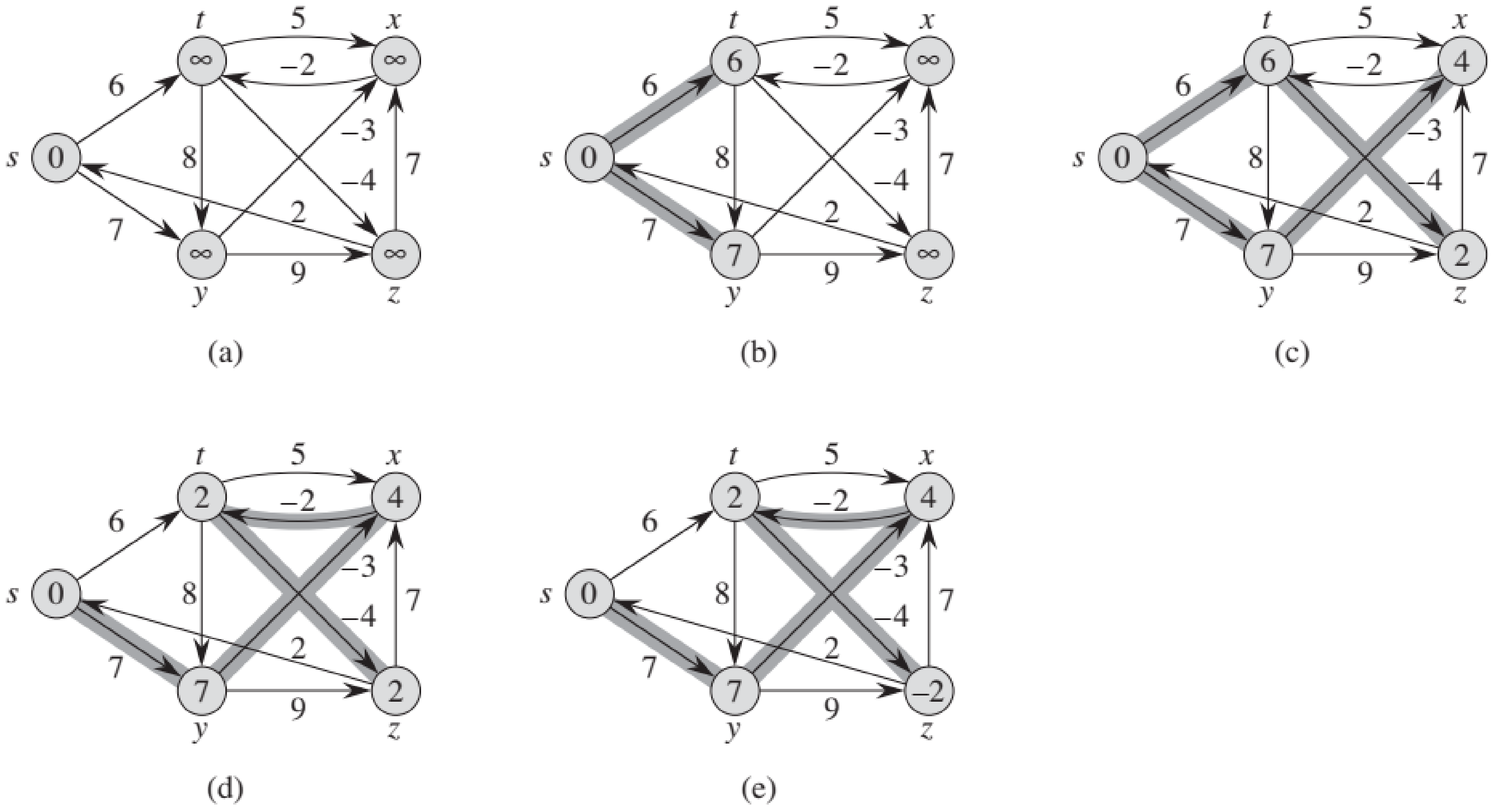
- Source is vertex s
- shaded edges indicate predecessor values: if edge (u, v) is shaded, then
  - π .v = u
- Each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

(a) Initialization

(b)–(e) Each successive pass |V|-1 = 5-1 = 4 over edges

(e) The d and π values are the final values

The Bellman-Ford algorithm returns TRUE

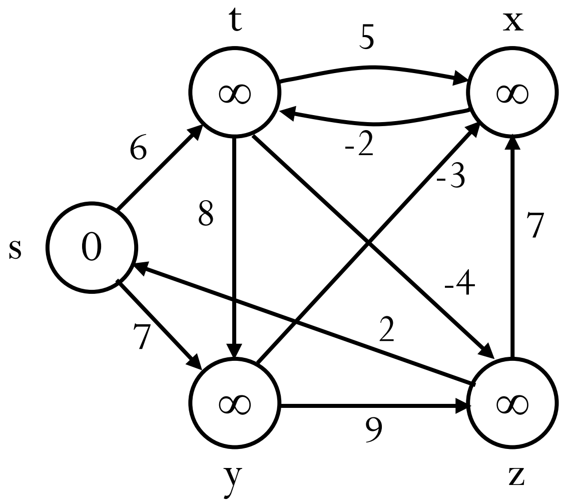


Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

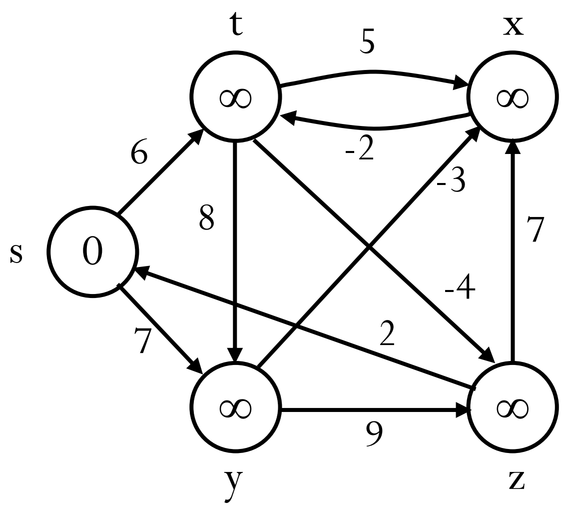
Bellman-Ford Algorithm

- 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.

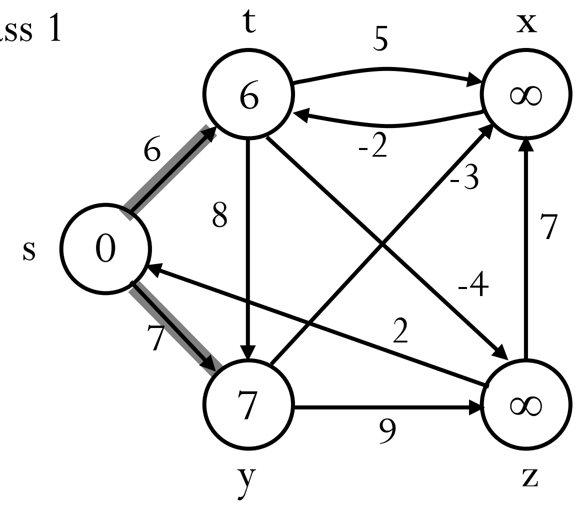
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)



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



Pass 1

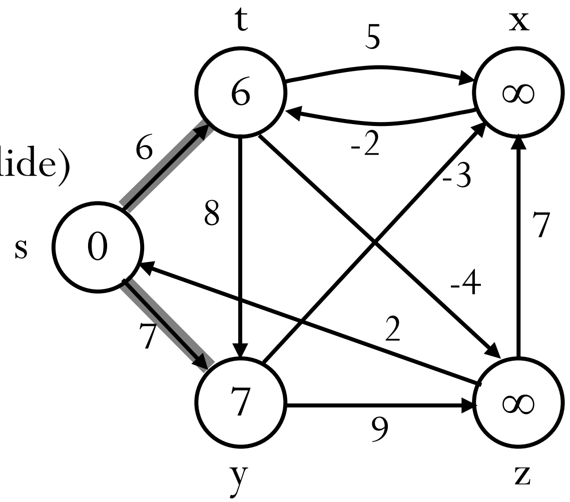


E: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

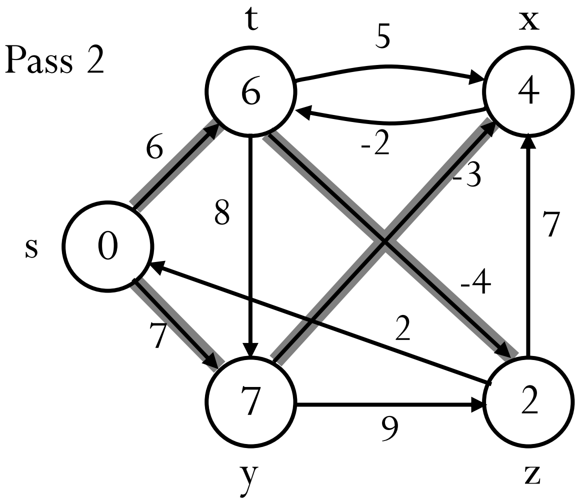
Example

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

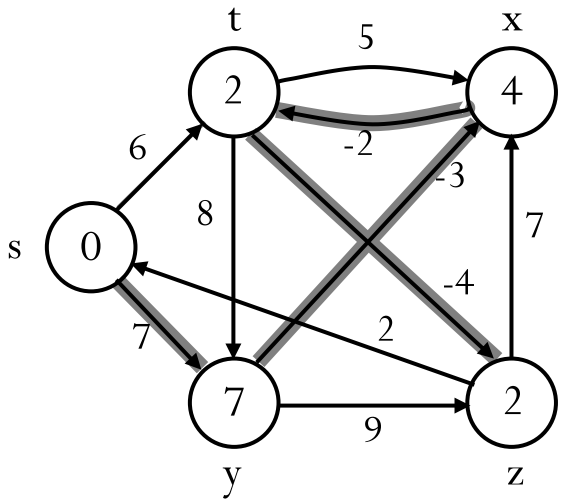
Pass 1
(from previous slide)



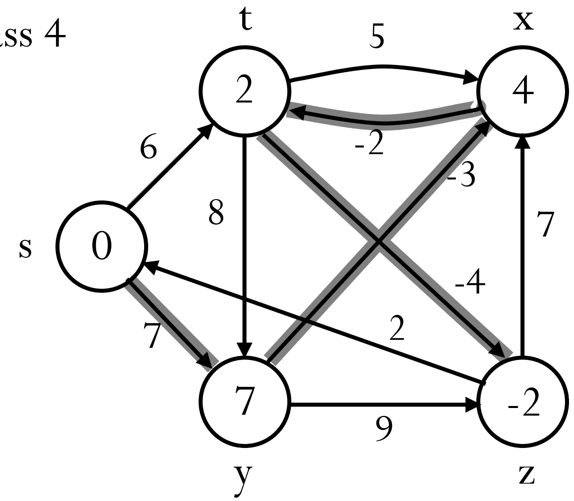
Pass 2



Pass 3



Pass 4



George Bebis

Detecting Negative Cycles

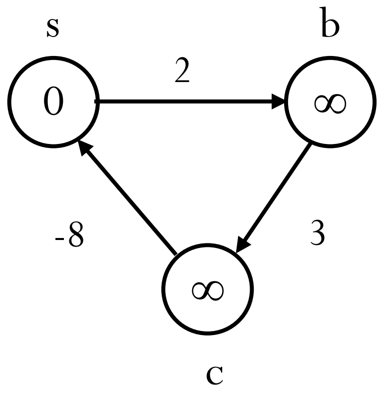
- (perform extra test after V-1 iterations)

  for each edge $(u, v) \in E$

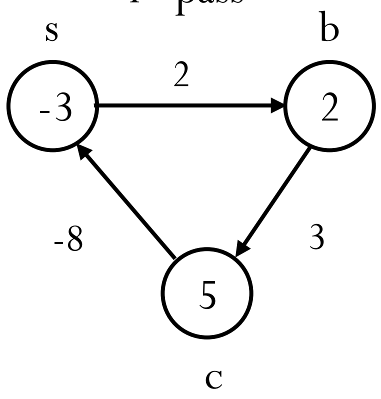
      do if $d[v] > d[u] + w(u, v)$

          then return FALSE

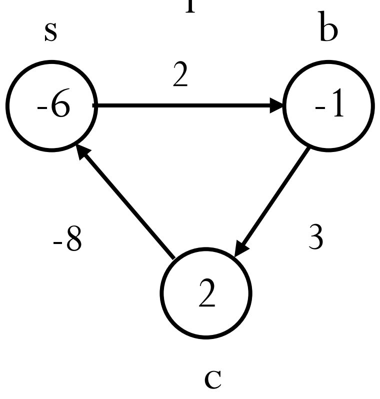
  return TRUE



1st pass



2nd pass



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

Look at edge (s, b):

$d[b] = -1$

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

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

Cycles

Can shortest paths contain cycles?

- Negative-weight cycles: NO
  - Shortest path is not well defined, because each iteration result in reduced shortest path
- Positive-weight cycles: NO
  - Path is a tree, property of tree that tree does not have cycle
  - By removing the cycle, we can get a shorter path, like we did in minimum spanning tree
- Zero-weight cycles
  - No reason to use them
  - Can remove them to obtain a path with same weight

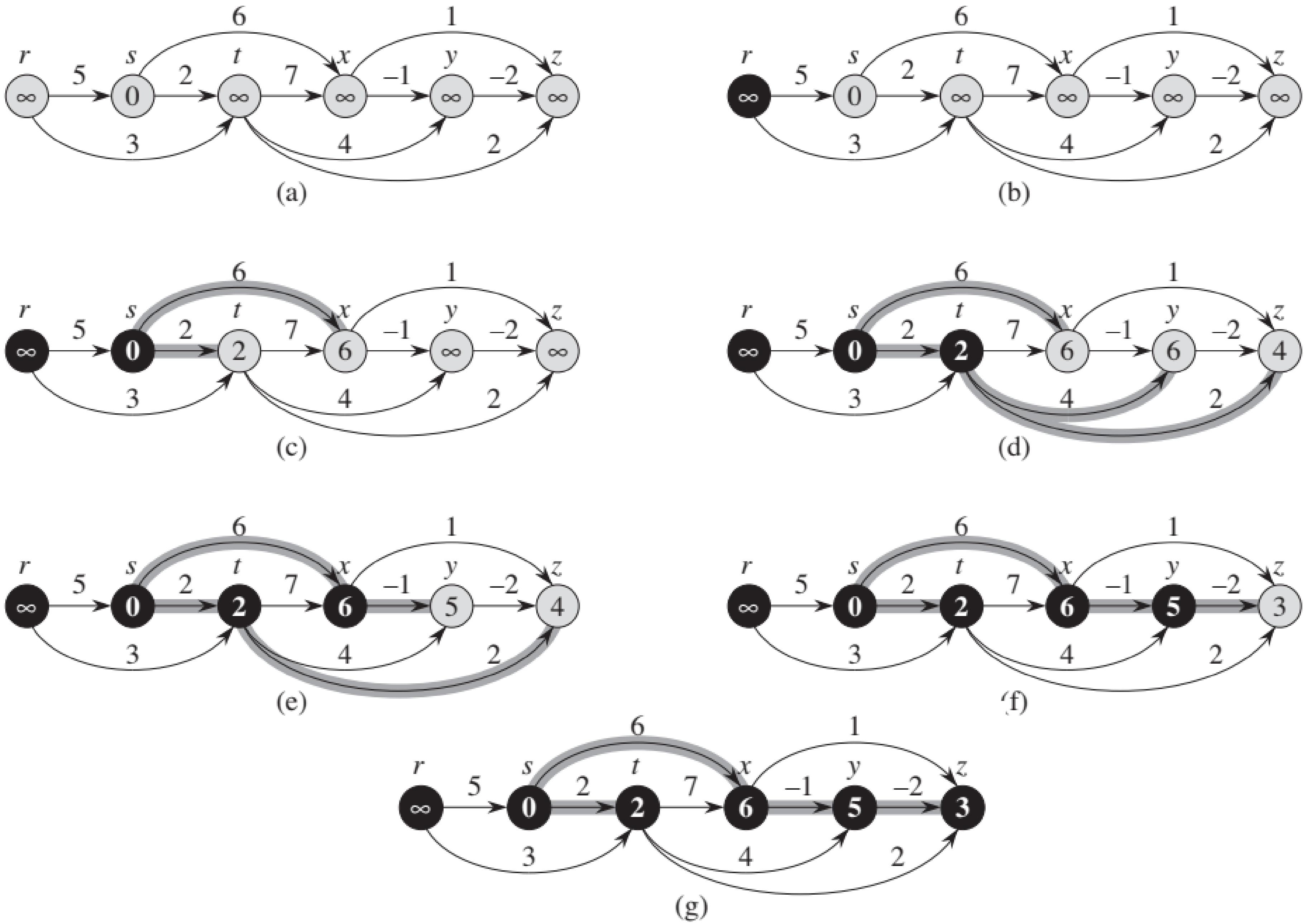
Shortest paths in Directed Acyclic Graphs (DAG)

- Single-source Shortest paths in DAG
- Topological sort of line 1 takes O(V + E) time
- INITIALIZE line 2 takes O(V) time
- **for** loop of lines 3–5 makes one iteration per vertex
- **for** loop of lines 4–5 relaxes each edge exactly once.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G          ← O(V+E)
2  INITIALIZE-SINGLE-SOURCE(G, s)                ← O(V)
3  for each vertex u, taken in topologically sorted order   ← O(V)
4      for each vertex v ∈ G.Adj[u]              ← O(E)
5          RELAX(u, v, w)                        ← O(E)
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Shortest paths in Directed Acyclic Graphs (DAG)

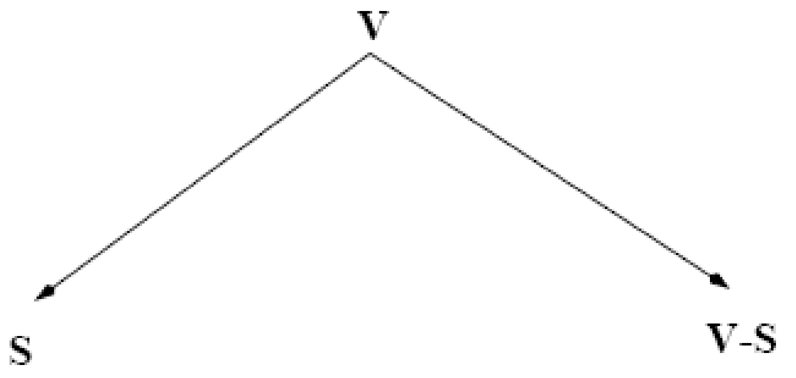


Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Dijkstra’s Algorithm

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:
- Similar to Prim's algorithm, which finds Minimum Spanning Tree (MST)



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

Dijkstra's Algorithm

- Line 1 initializes the d and π values in the usual way,
- Line 2 initializes the set S to the empty set.
- Line 3 initializes the min-priority queue Q to contain all the vertices in V;

DIJKSTRA(G, w, s)

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

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Dijkstra’s Algorithm

- While loop of lines 4–8, line 5 extracts a vertex u from Q and
- Line 6 adds it to set S, thereby maintaining the invariant. Vertex u, therefore, has the smallest shortest-path estimate of any vertex in V - S.
- Then, lines 7–8 relax each edge, thus updating the estimate v.d and the predecessor v.π

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

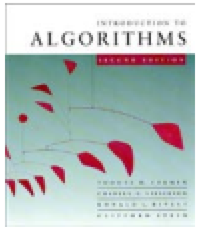
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Dijkstra's Algorithm

- While loop of lines 4–8, line 5 extracts a vertex u from Q and
- Line 6 adds it to set S, thereby maintaining the invariant. Vertex u, therefore, has the smallest shortest-path estimate of any vertex in V - S.
- Then, lines 7–8 relax each edge, thus updating the estimate v.d and the predecessor v.π

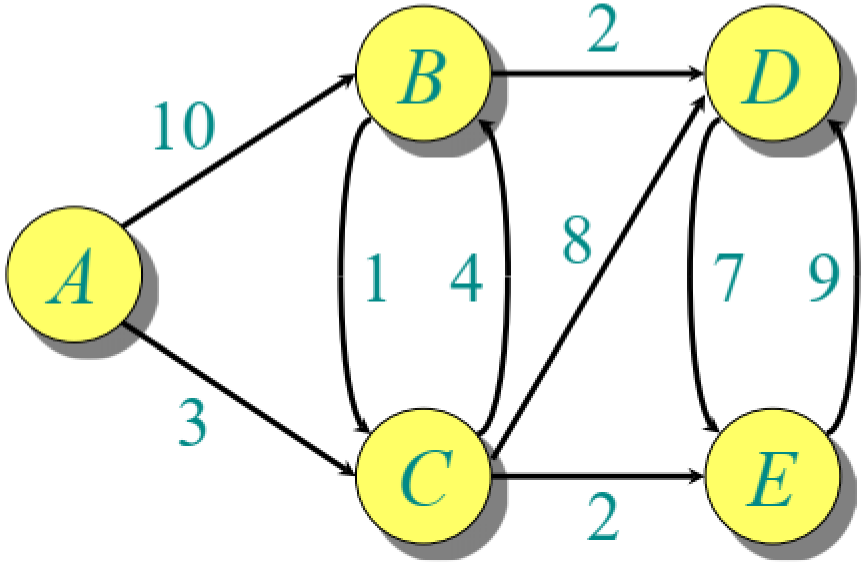
```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

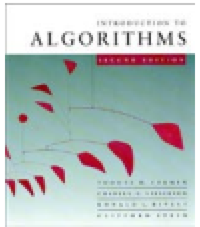
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.



Example of Dijkstra's algorithm

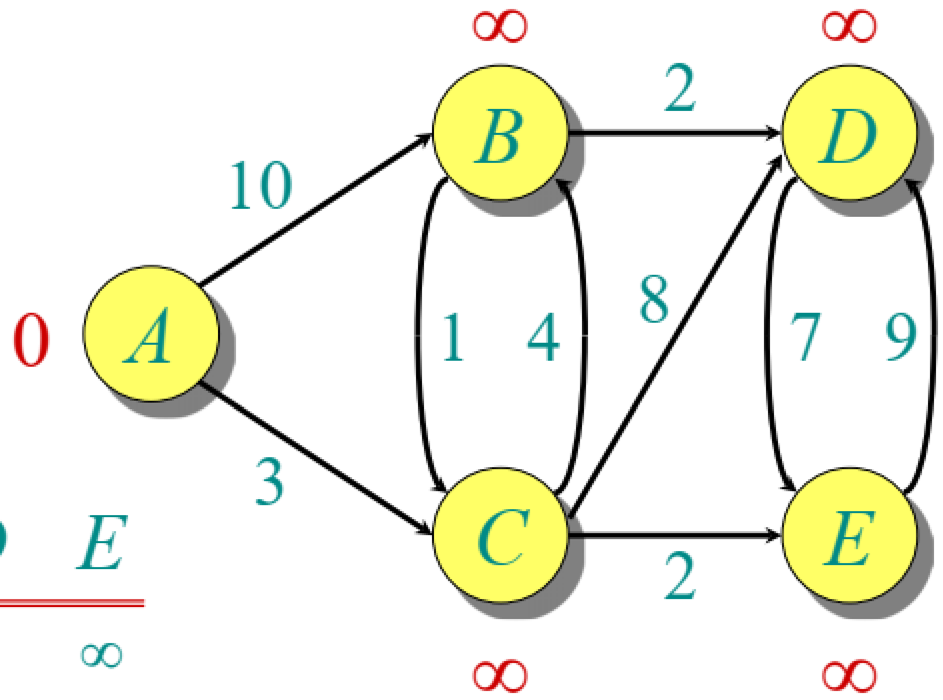
Graph with nonnegative edge weights:





Example of Dijkstra's algorithm

Initialize:

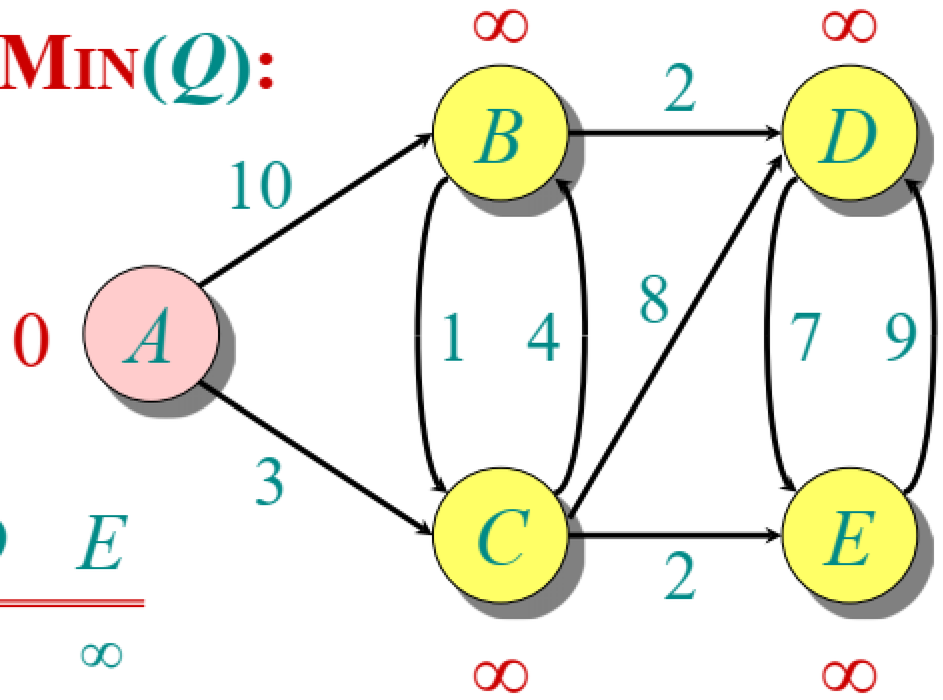


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Example of Dijkstra's algorithm

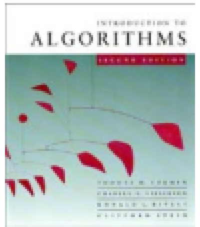
"A" ← EXTRACT-MIN(Q):



Q: A B C D E

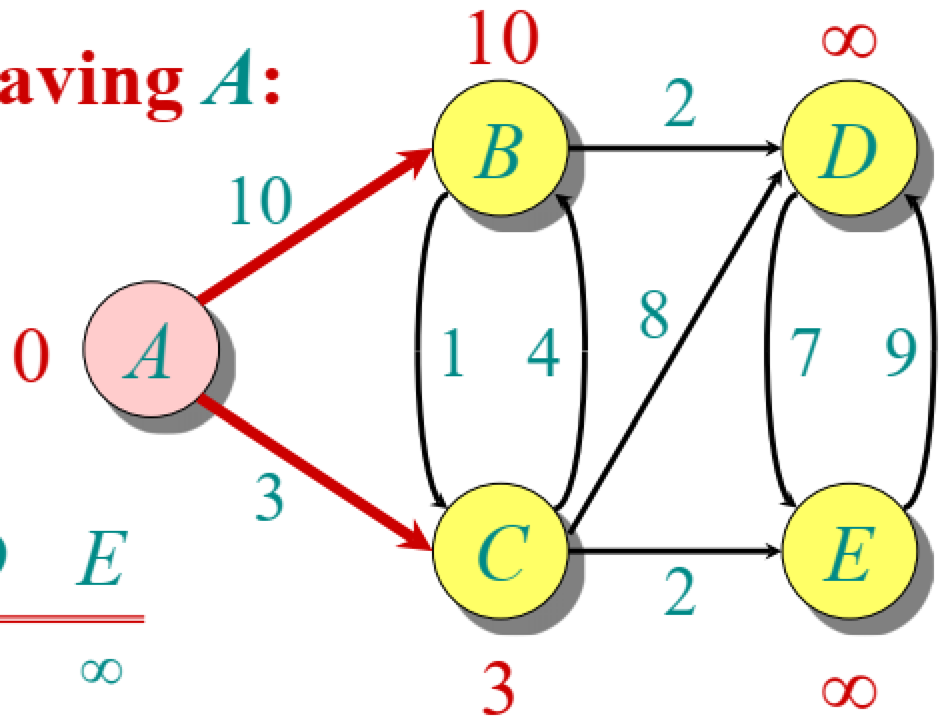
A	B	C	D	E
0	∞	∞	∞	∞

S: { A }



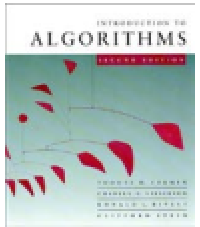
Example of Dijkstra's algorithm

Relax all edges leaving *A*:



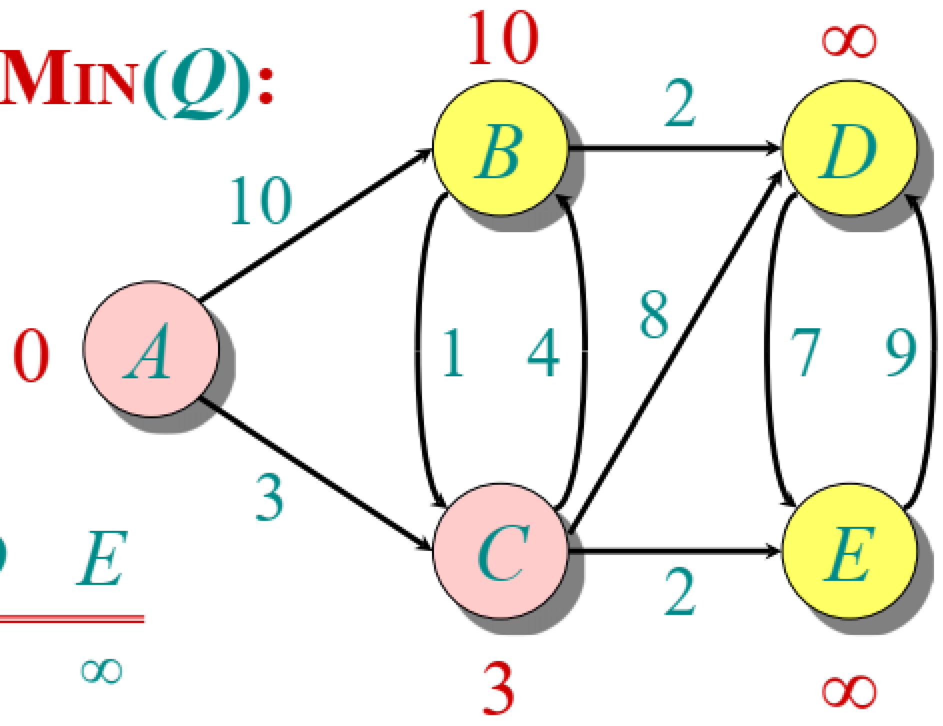
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A* }



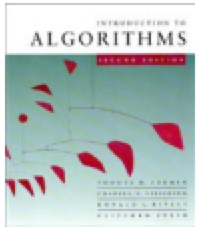
Example of Dijkstra's algorithm

"*C*" ← EXTRACT-MIN(*Q*):



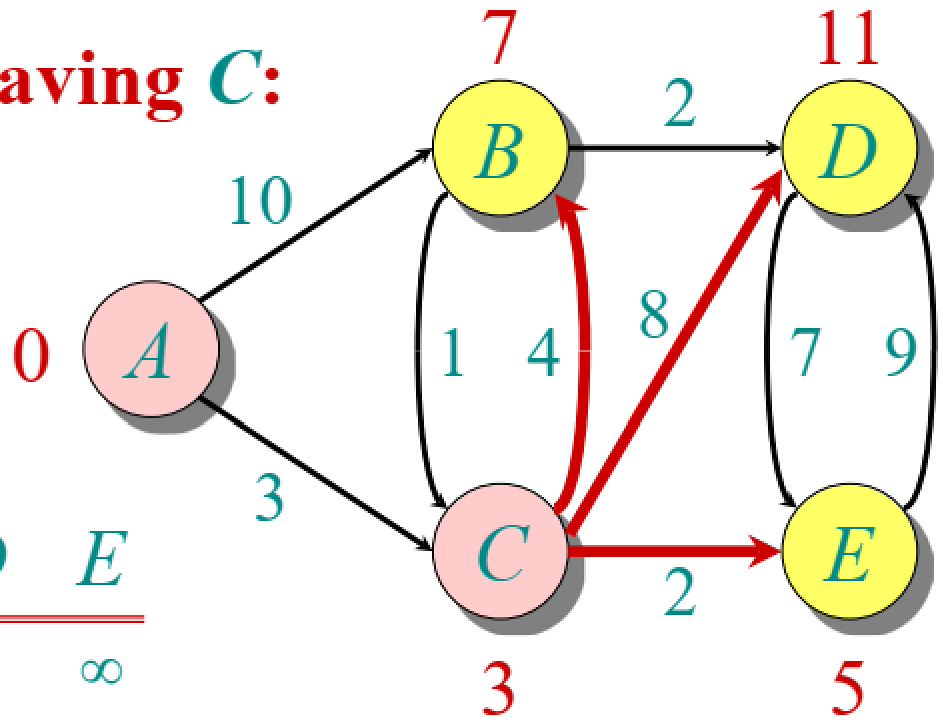
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A*, *C* }



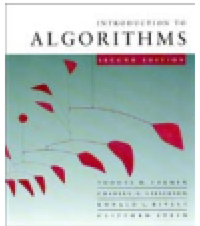
Example of Dijkstra's algorithm

Relax all edges leaving *C*:



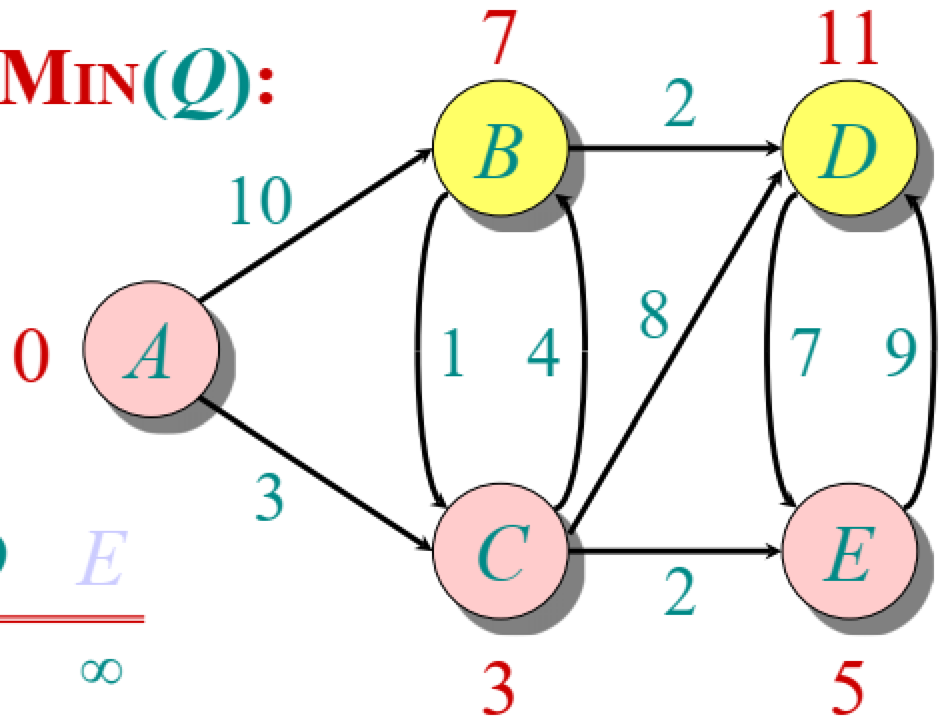
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }



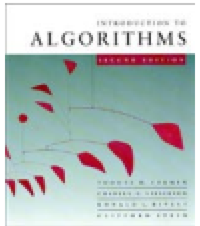
Example of Dijkstra's algorithm

"E" $\leftarrow$ EXTRACT-MIN(Q):



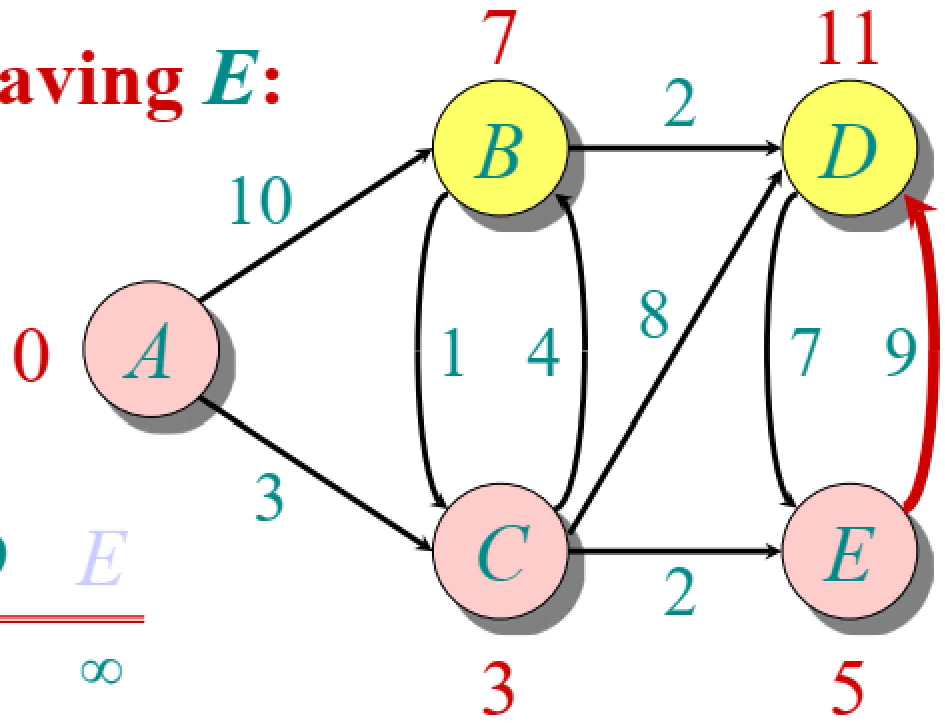
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }



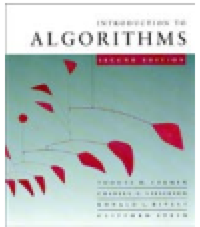
Example of Dijkstra's algorithm

Relax all edges leaving *E*:

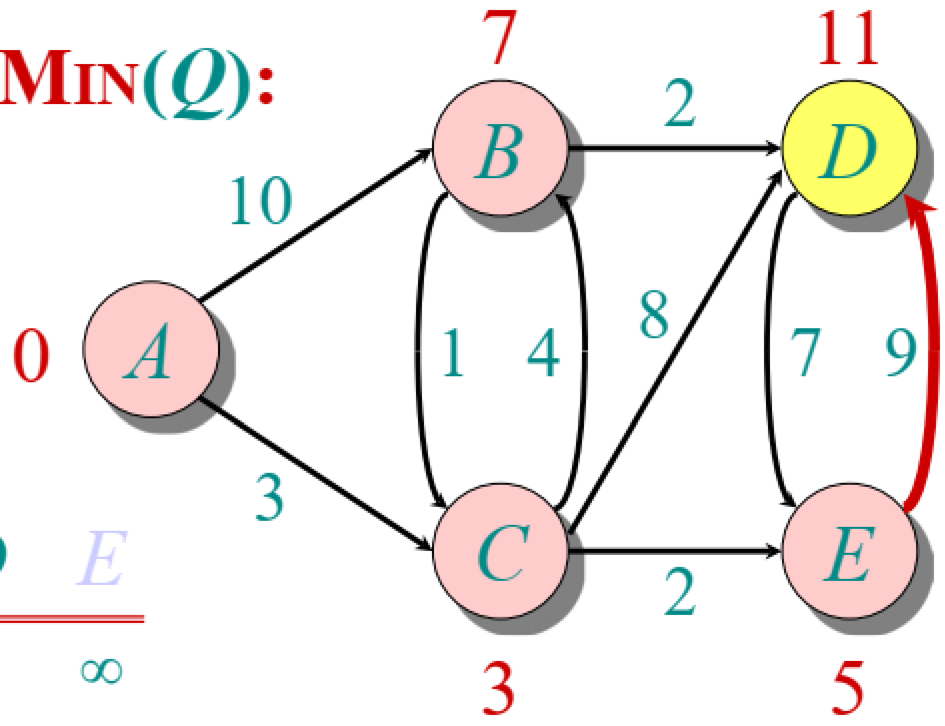


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

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



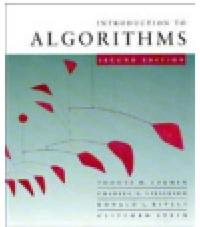
Example of Dijkstra's algorithm



"B" $\leftarrow$ EXTRACT-MIN(Q):

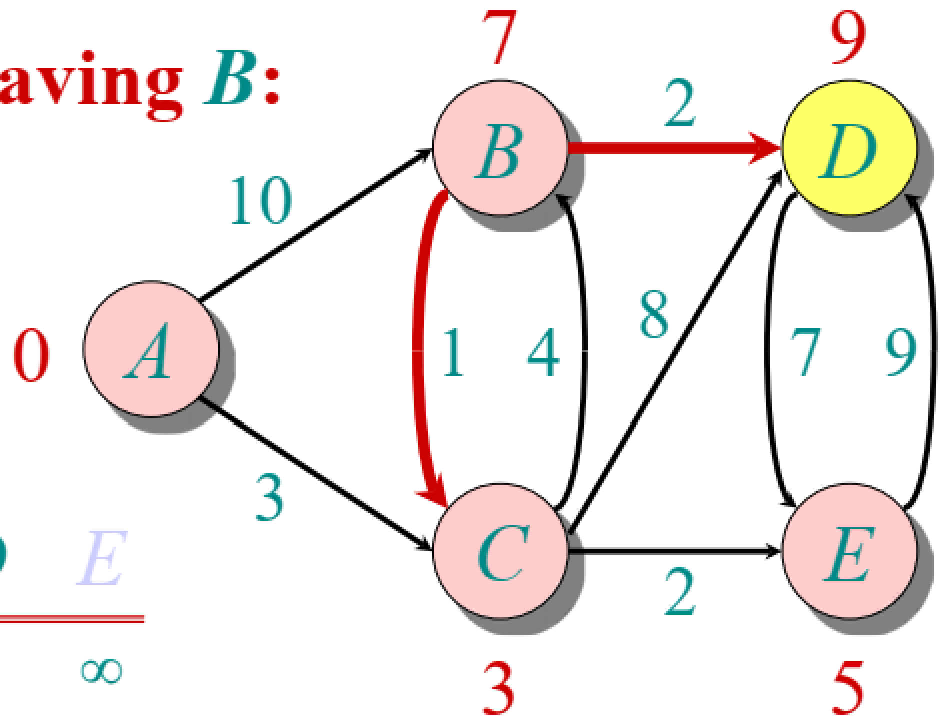
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }



Example of Dijkstra's algorithm

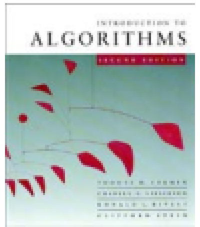
Relax all edges leaving *B*:



Q:

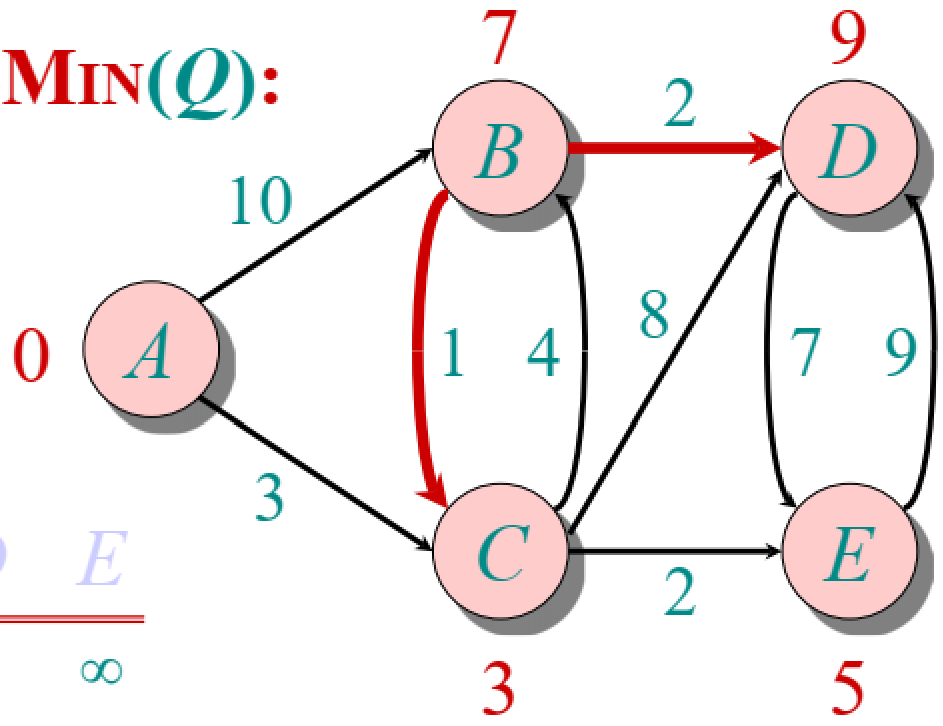
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	

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



Example of Dijkstra's algorithm

"D" ← EXTRACT-MIN(Q):

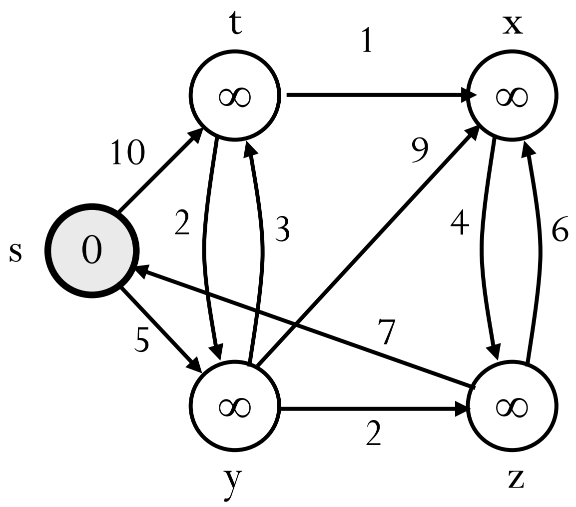


Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	

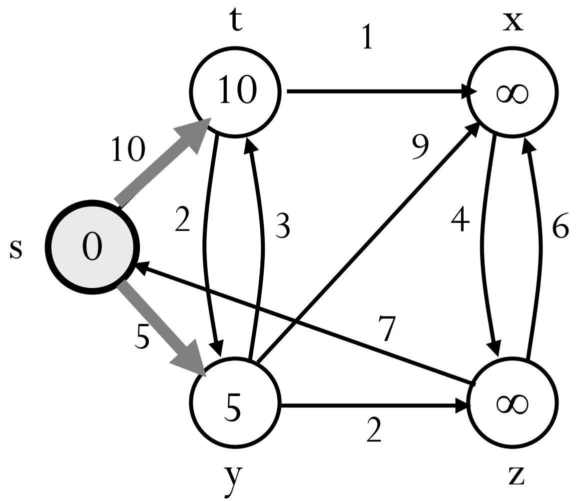
S: { A, C, E, B, D }

Dijkstra (G, w, s)

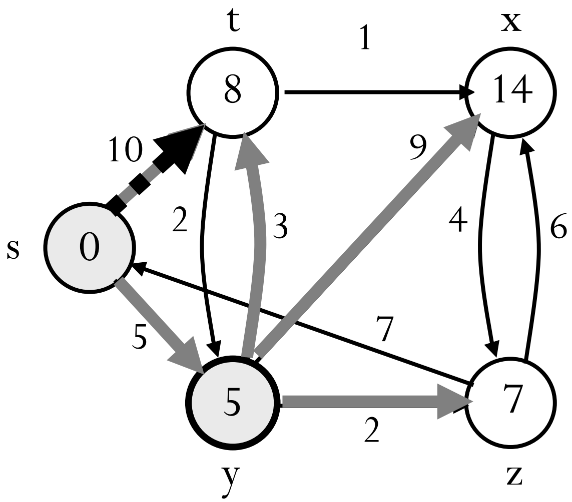
S=<> Q=<s,t,x,z,y>



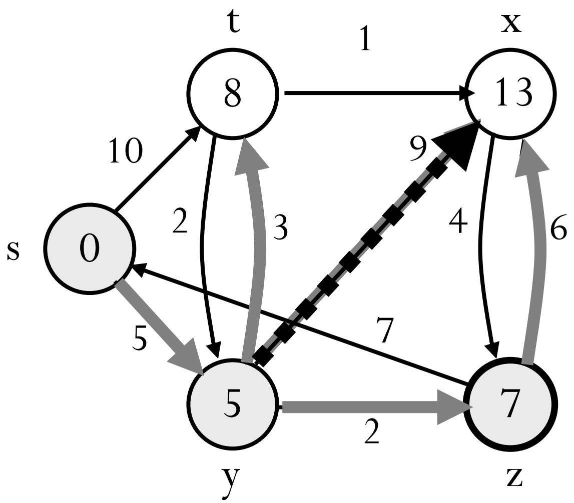
S=<s> Q=<y,t,x,z>



Example (cont.)



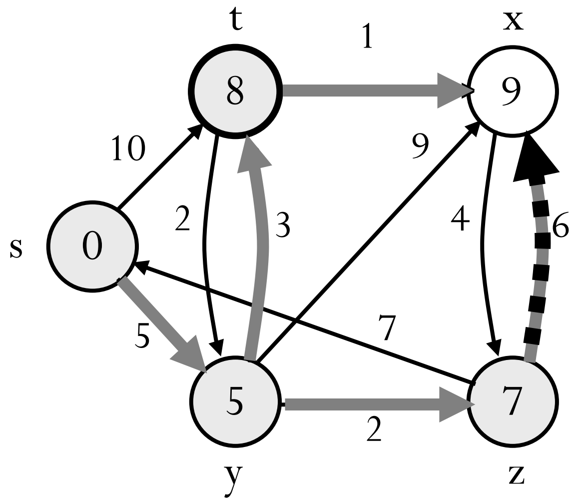
S=<s,y> Q=<z,t,x>



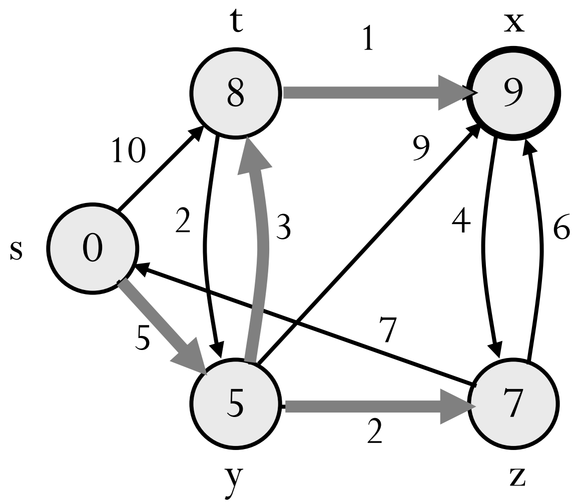
S=<s,y,z> Q=<t,x>

Example (cont.)

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



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



Dijkstra's Algorithm

- Running time: O(VlgV + **ElgV**) = **O(ElgV)**

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s) ← Θ(V)
$S = \emptyset$
$Q = G.V$ ← O(V) build min-heap
while $Q \neq \emptyset$ ← Executed O(V) times
  $u =$ EXTRACT-MIN(Q) ← O(lgV)
  $S = S \cup \{u\}$ ← **O(VlgV)**
  for each vertex $v \in G.Adj[u]$ ← O(E) times
    RELAX(u, v, w) ← **O(ElgV)**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Dijkstra’s Algorithm

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	1 †	$\log V$ †	1 †	$E + V \log V$

† amortized

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Single Source Shortest-Paths Implementation

algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	$E + V$	$E + V$	V
Dijkstra (binary heap)	no negative weights	$E \log V$	$E \log V$	V
Bellman-Ford	no negative cycles	$E V$	$E V$	V
Bellman-Ford (queue-based)		$E + V$	$E V$	V

Remark 1. Directed cycles make the problem harder.

Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Java Code Implementation for Graph and Shortest Path

Java Code: Weighted Directed Edge

```
public class DirectedEdge
{
   private final int v, w;
   private final double weight;

   public DirectedEdge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int from()
   {  return v;  }

   public int to()
   {  return w;  }

   public int weight()
   {  return weight; }
}
```

from() and to() replace either() and other()

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Java Code: Edge-Weighted Digraph

```
public class EdgeWeightedDigraph
{
   private final int V;
   private final Bag<DirectedEdge>[] adj;

   public EdgeWeightedDigraph(int V)
   {
      this.V = V;
      adj = (Bag<DirectedEdge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<DirectedEdge>();
   }

   public void addEdge(DirectedEdge e)
   {
      int v = e.from();
      adj[v].add(e);
   }

   public Iterable<DirectedEdge> adj(int v)
   {  return adj[v];  }
}
```

add edge e = v→w to only v's adjacency list

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Java Code: Relaxation

- Relax(u, v, w) → relax(DirectedEdge e)
- u became v u = v = e.from()
- v became w v = w = e.to()
- w(u, v) became e.weight()

RELAX(u, v, w)

1 **if** $v.d > u.d + w(u, v)$
2 $v.d = u.d + w(u, v)$
3 $v.\pi = u$

```java
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
   }
}
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Java Code: Bellman-Ford algorithm

```java
private static final double EPSILON = 1E-14;

private double[] distTo;               // distTo[v] = distance  of shortest s->v path
private DirectedEdge[] edgeTo;         // edgeTo[v] = last edge on shortest s->v path
private boolean[] onQueue;             // onQueue[v] = is v currently on the queue?
private Queue<Integer> queue;          // queue of vertices to relax
private int cost;                      // number of calls to relax()
private Iterable<DirectedEdge> cycle;  // negative cycle (or null if no such cycle)
```


Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Java Code: Bellman-Ford algorithm

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

```java
public BellmanFordSP(EdgeWeightedDigraph G, int s) {
    distTo  = new double[G.V()];
    edgeTo  = new DirectedEdge[G.V()];
    onQueue = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;

    // Bellman-Ford algorithm
    queue = new Queue<Integer>();
    queue.enqueue(s);
    onQueue[s] = true;
    while (!queue.isEmpty() && !hasNegativeCycle()) {
        int v = queue.dequeue();
        onQueue[v] = false;
        relax(G, v);
    }

    assert check(G, s);
}
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Java Code: Bellman-Ford algorithm

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

```java
// relax vertex v and put other endpoints on queue if changed
private void relax(EdgeWeightedDigraph G, int v) {
    for (DirectedEdge e : G.adj(v)) {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight() + EPSILON) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (!onQueue[w]) {
                queue.enqueue(w);
                onQueue[w] = true;
            }
        }
        if (++cost % G.V() == 0) {
            findNegativeCycle();
            if (hasNegativeCycle()) return;  // found a negative cycle
        }
    }
}
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Java Code: Shortest paths in DAG

Directed Acyclic Graphs (DAG)

DAG-SHORTEST-PATHS(G, w, s)

1 topologically sort the vertices of G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **for** each vertex u, taken in topologically sorted order
4 **for** each vertex $v \in G.Adj[u]$
5 RELAX(u, v, w)

```
public class AcyclicSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;

   public AcyclicSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      Topological topological = new Topological(G);
      for (int v : topological.order())
         for (DirectedEdge e : G.adj(v))
            relax(e);
   }
}
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Java Code: Dijkstra's algo

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX(u, v, w)
```

```java
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
      {
          int v = pq.delMin();
          for (DirectedEdge e : G.adj(v))
             relax(e);
      }
   }
}
```

relax vertices in order of distance from s

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.
ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Dijkstra's algorithm: Relaxation

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
      if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
      else                pq.insert     (w, distTo[w]);
   }
}
```

update PQ

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Applications of Shortest Path

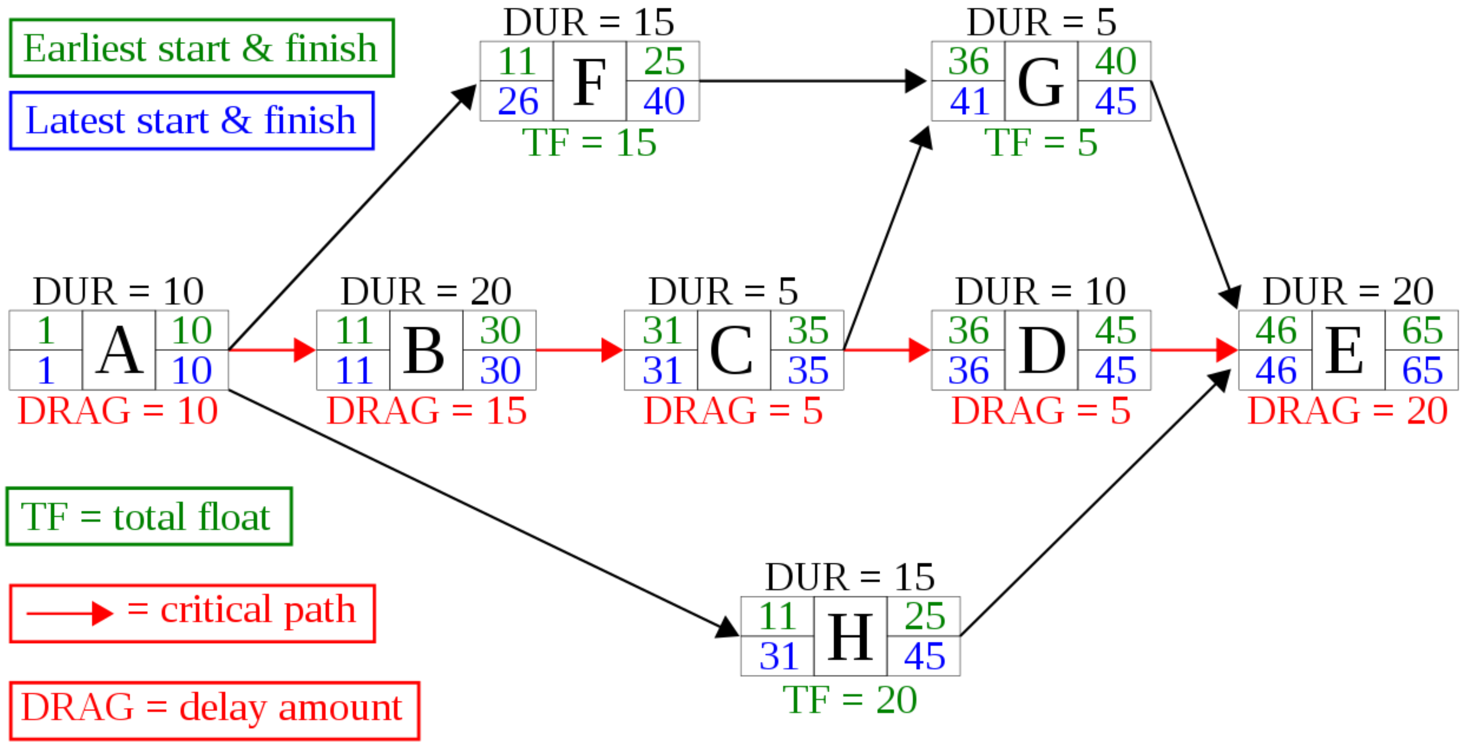
PERT Chart Analysis: Critical Path

- Edges represent jobs to be performed, and edge weights represent the times required to perform particular jobs.
- If edge (u, v) enters vertex v and edge (v, x) leaves v, then job (u, v) must be performed before job (v, x).
- A path through this dag represents a sequence of jobs that must be performed in a particular order. A ***critical path*** is a *longest* path through the dag, corresponding to the longest time to perform any sequence of jobs. Thus, the weight of a critical path provides a lower bound on the total time to perform all the jobs. We can find a critical path by either

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

PERT Chart Analysis: Critical path

- The longest stretch of dependent activities and measuring the time required to complete them from start to finish.
  - Activities A, B, C, D, and E comprise the critical path,



https://en.wikipedia.org/wiki/Critical_path_method

Longest path: Critical Path

- Negating the edge weights and running DAG-SHORTEST-PATHS,

or

- running DAG SHORTEST-PATHS, with the modification that we replace "∞" by "$-\infty$" in line 2 of INITIALIZE-SINGLE-SOURCE and ">" by "<" in the RELAX procedure.

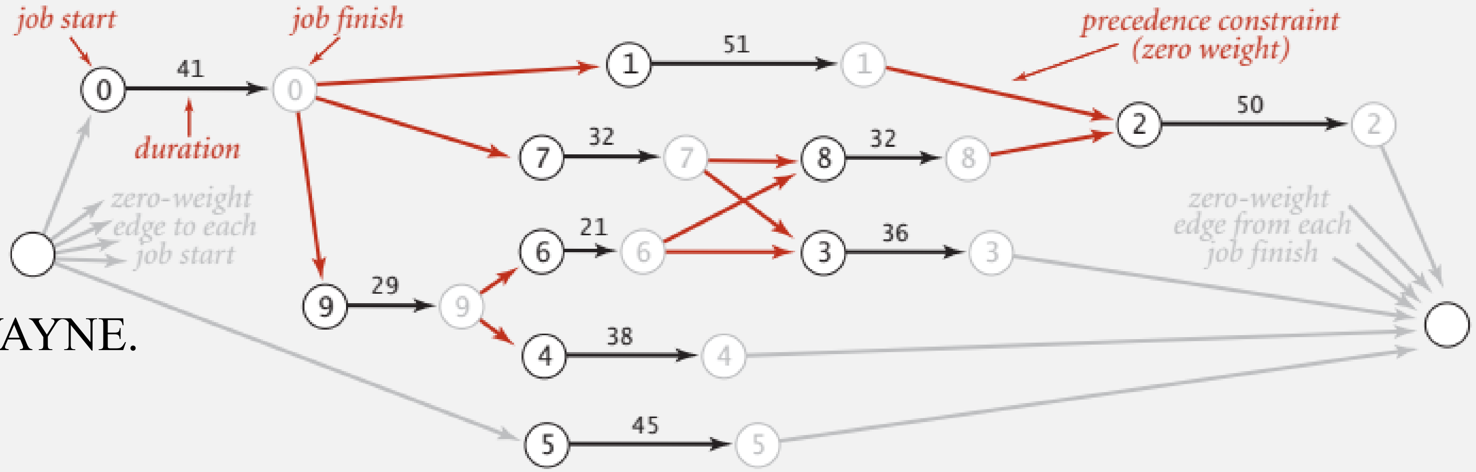
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

Critical path method

CPM. To solve a parallel job-scheduling problem, create edge-weighted DAG:

- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
  - begin to end (weighted by duration)
  - source to begin (0 weight)
  - end to sink (0 weight)
- One edge for each precedence constraint (0 weight).

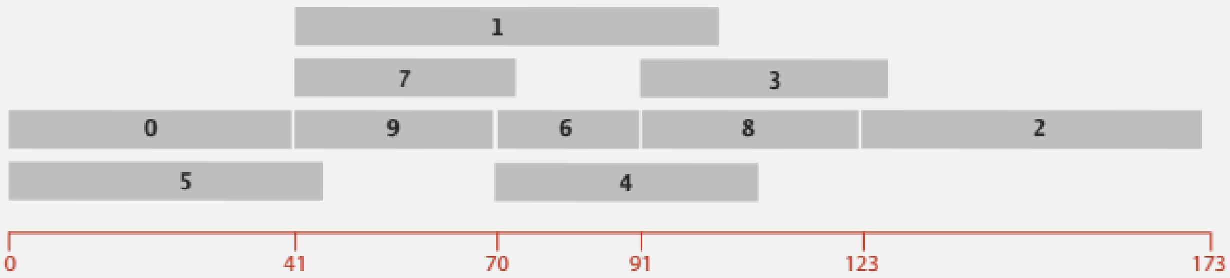
job	duration	must complete before
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6



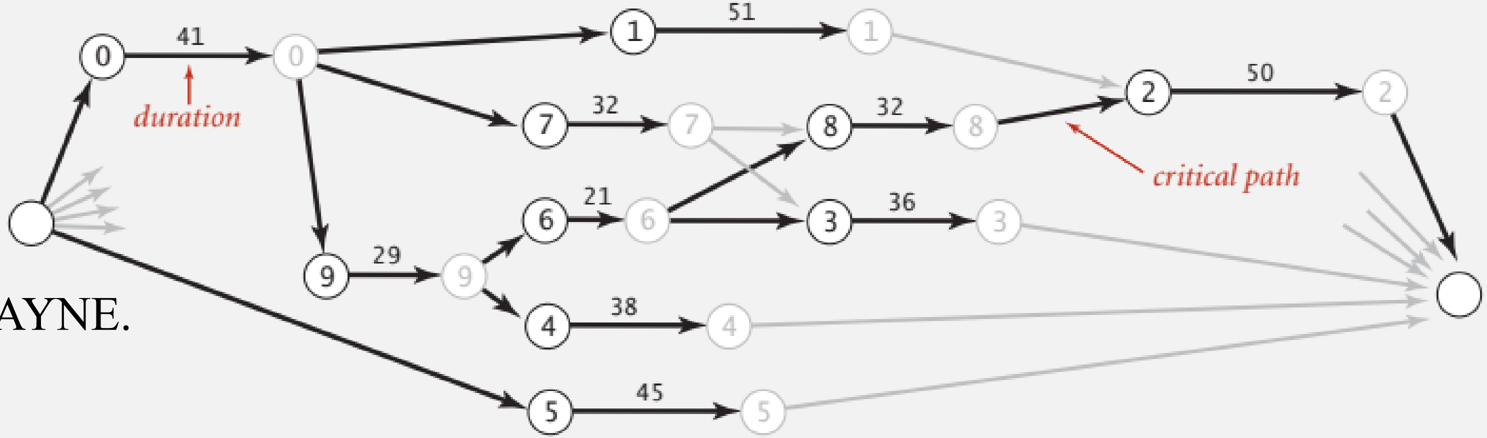
ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Critical path method

CPM. Use longest path from the source to schedule each job.



Parallel job scheduling solution



ROBERT SEDGEWICK | KEVIN WAYNE.
Algorithms 4th Edition,
https://algs4.cs.princeton.edu/44sp/

Negative cycle application: arbitrage detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.350	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.620	1	0.953
CAD	0.995	0.732	0.650	1.049	1

Ex. \$1,000 $\Rightarrow$ 741 Euros $\Rightarrow$ 1,012.206 Canadian dollars $\Rightarrow$ \$1,007.14497.

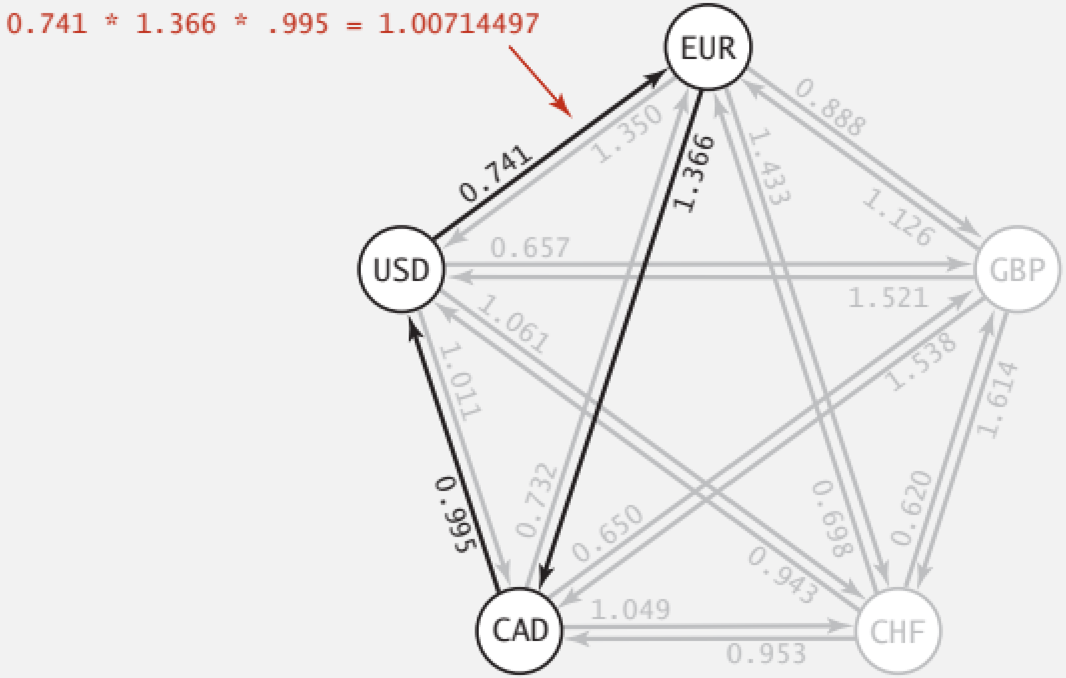
$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Negative cycle application: arbitrage detection

Currency exchange graph.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1.



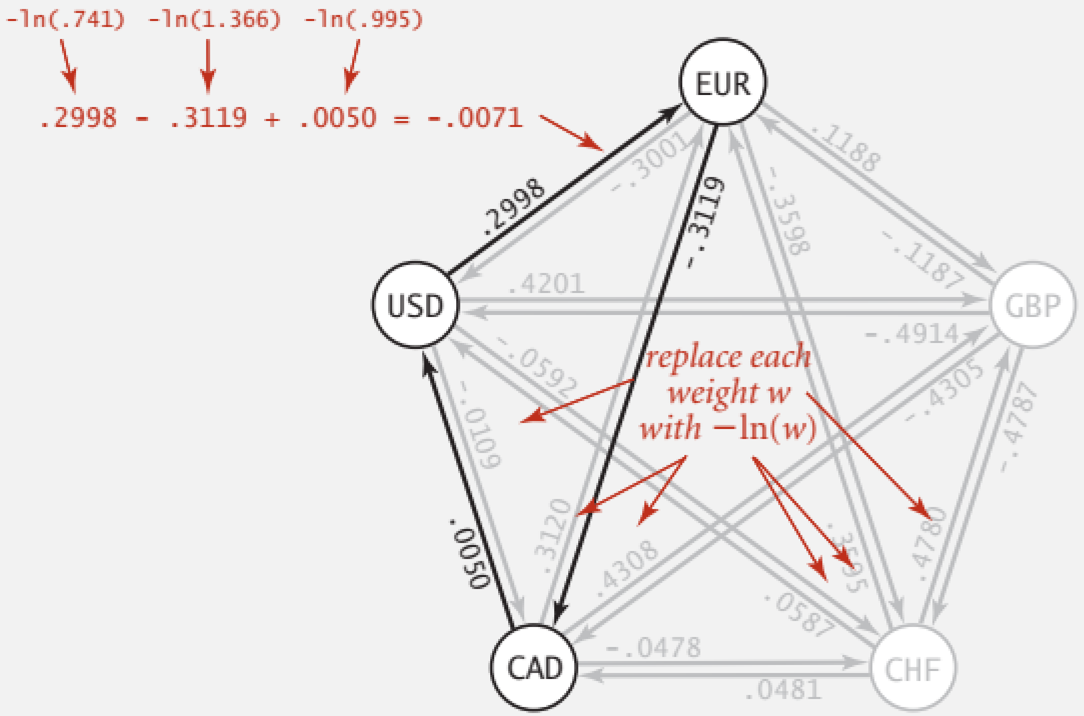
Challenge. Express as a negative cycle detection problem.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \rightarrow w$ be $-ln$ (exchange rate from currency v to w).
- Multiplication turns to addition; > 1 turns to < 0.
- Find a directed cycle whose sum of edge weights is < 0 (negative cycle).



Remark. Fastest algorithm is extraordinarily valuable!

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

Shortest paths summary

Nonnegative weights.

- Arises in many application.
- Dijkstra's algorithm is nearly linear-time.

Acyclic edge-weighted digraphs.

- Arise in some applications.
- Topological sort algorithm is linear time.
- Edge weights can be negative.

Negative weights and negative cycles.

- Arise in some applications.
- If no negative cycles, can find shortest paths via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

Shortest-paths is a broadly useful problem-solving model.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition, https://algs4.cs.princeton.edu/44sp/

ขอบคุณ
Thai

Grazie
Italian

תודה רבה
Hebrew

ধন্যবাদ
Bangla

धन्यवादः
Sanskrit

ಧನ್ಯವಾದಗಳು
Kannada

Ευχαριστώ
Greek

Thank You
English

Gracias
Spanish

Спасибо
Russian

Obrigado
Portuguese

شكراً
Arabic

https://sites.google.com/site/animeshchaturvedi07

Merci
French

多謝
Traditional
Chinese

धन्यवाद
Hindi

Danke
German

多谢
Simplified
Chinese

நன்றி
Tamil
Tamil

ありがとうございました
Japanese

감사합니다
Korean