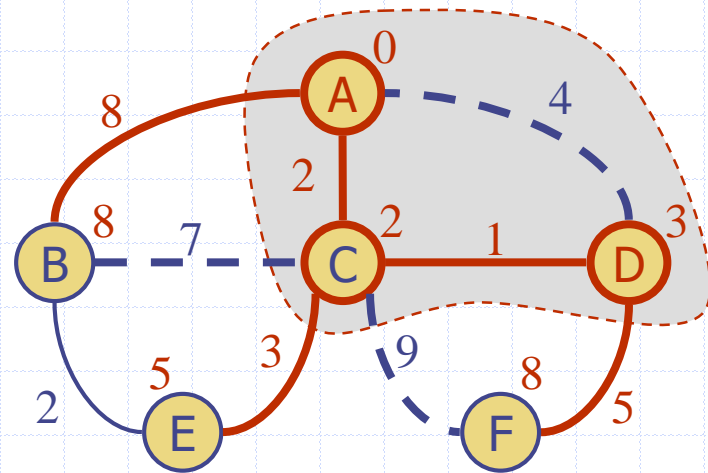
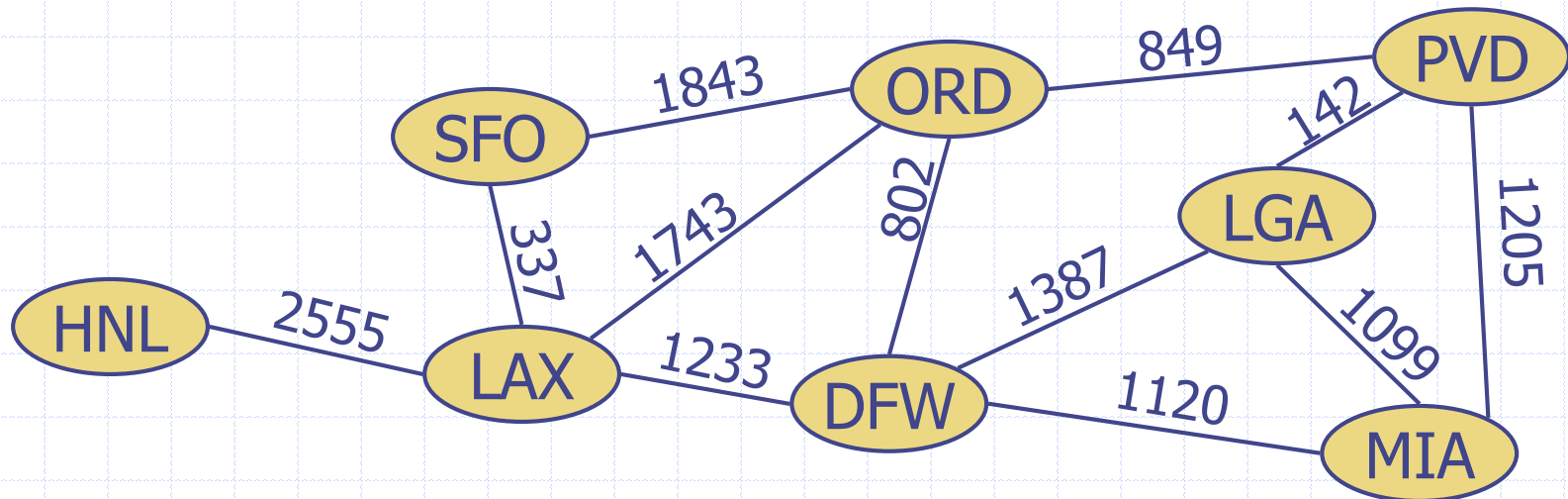


Shortest Paths



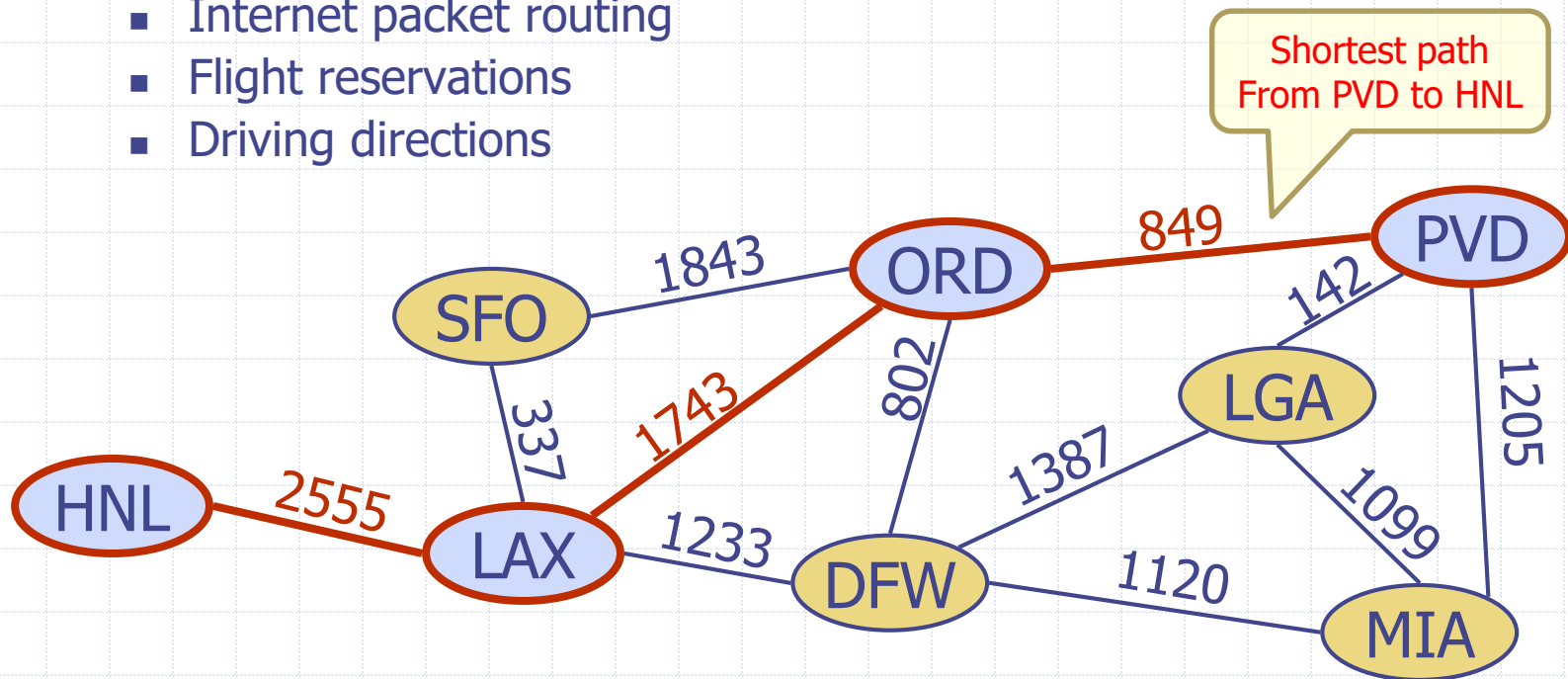
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
 - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices u and v , we want to find a path of minimum total weight between u and v .
- Example:
 - Shortest path between Providence and Honolulu
- Applications
 - Internet packet routing
 - Flight reservations
 - Driving directions



Shortest Path Properties

Property 1:

A subpath of a shortest path is itself a shortest path

DP!

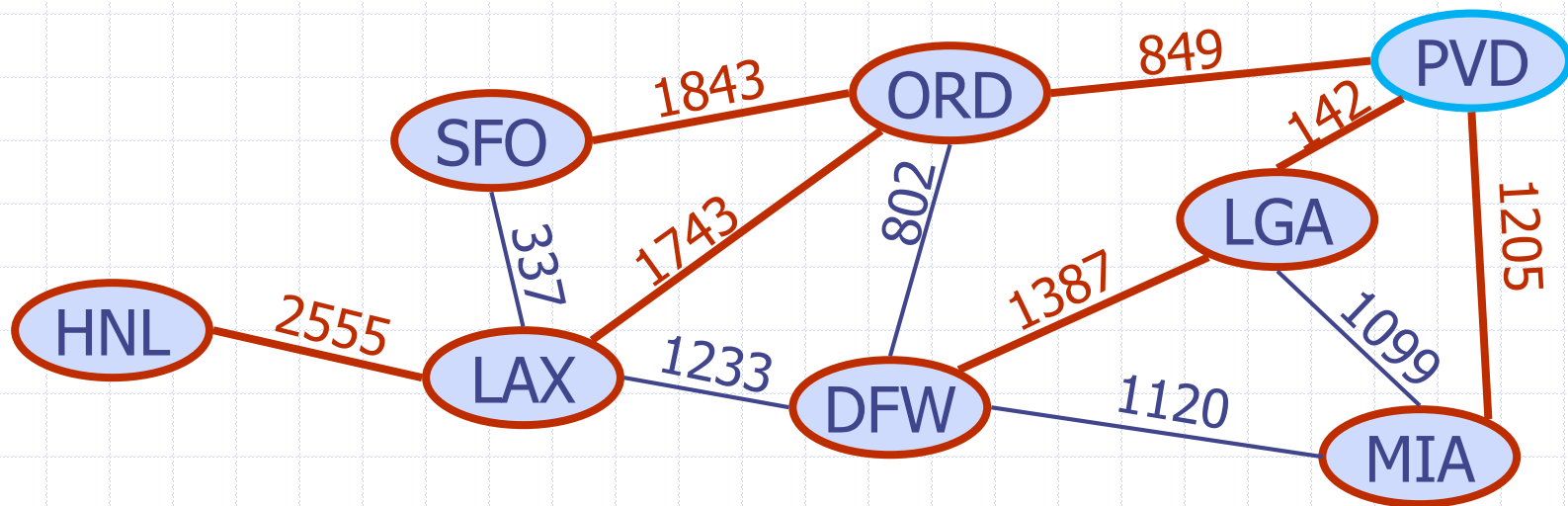
Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Connected & no cycles

Example:

Tree of shortest paths from Providence



Dijkstra's Algorithm

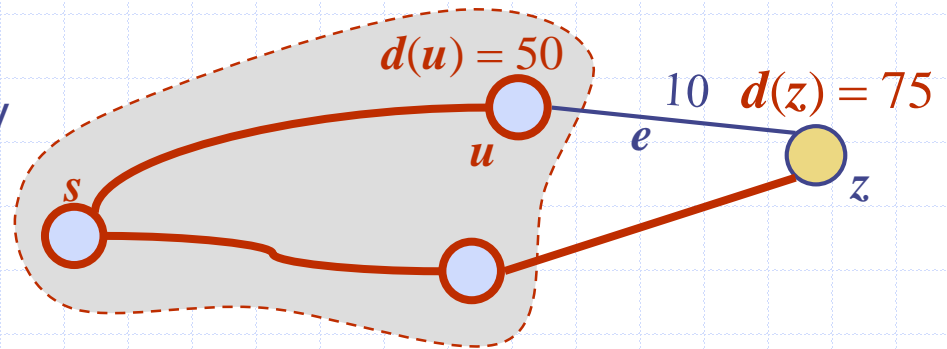
It was designed in 20 min
without pencil and paper!
([source](#))



- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s → Single-source all-destination
- Assumption:
 - the edge weights are nonnegative
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
 - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
 - We update the labels of the vertices adjacent to u

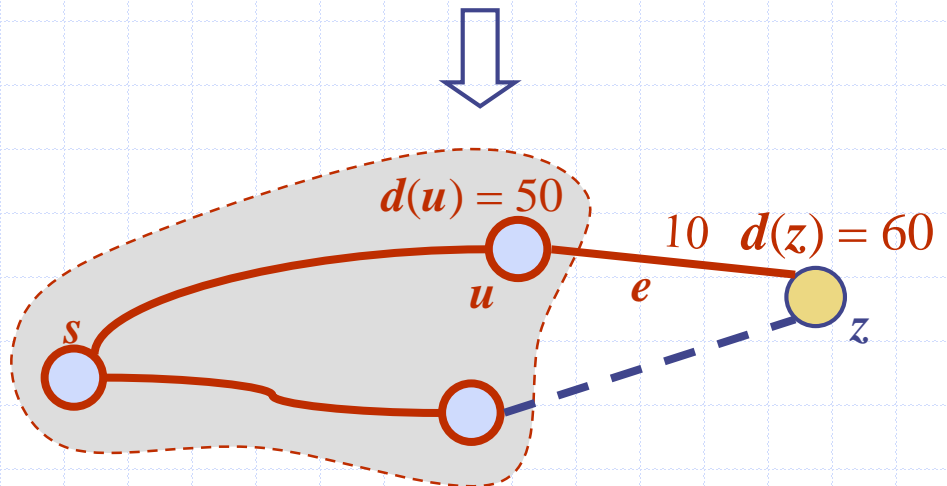
Edge Relaxation

- Consider an edge $e = (u, z)$ such that
 - u is the vertex most recently added to the cloud
 - z is not in the cloud

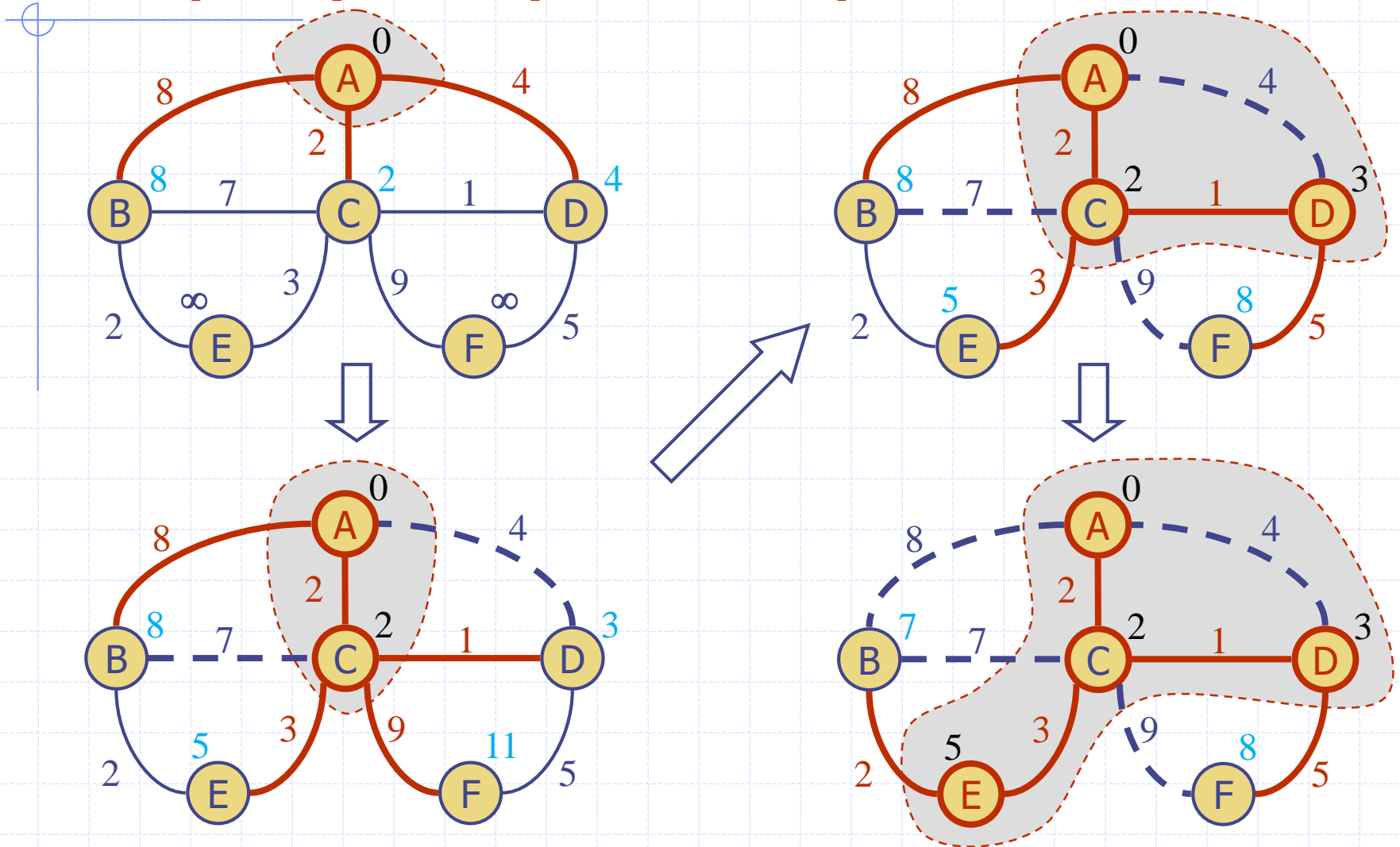


- The relaxation of edge e updates distance $d(z)$ as follows:

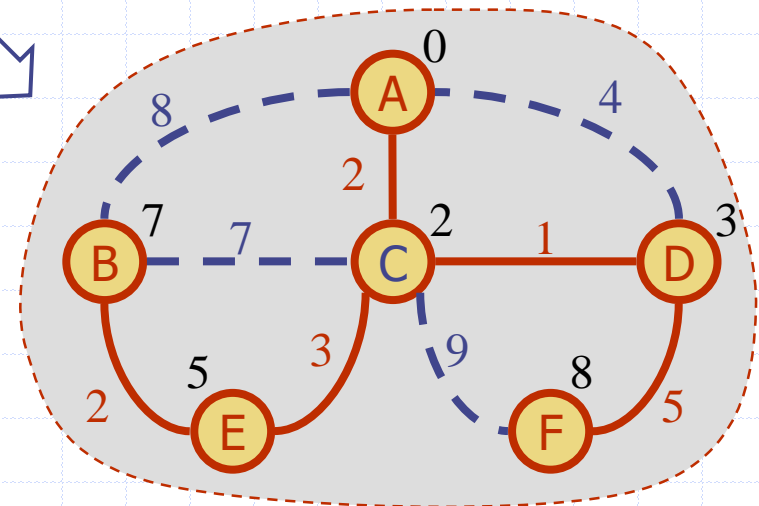
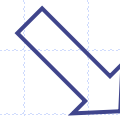
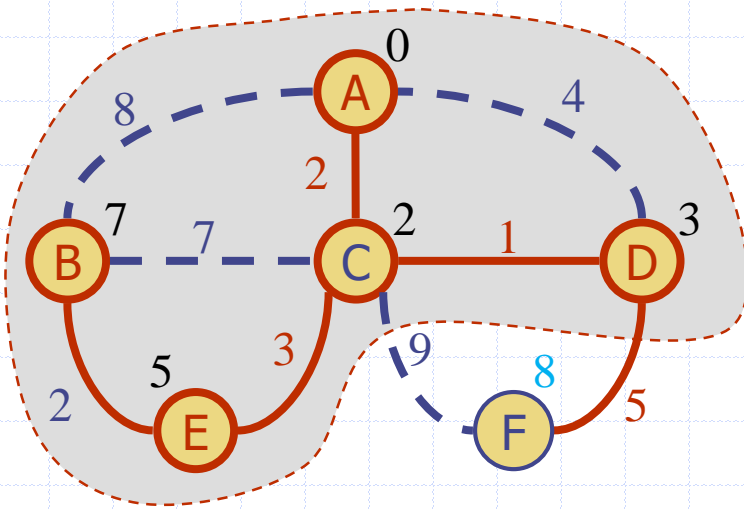
$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



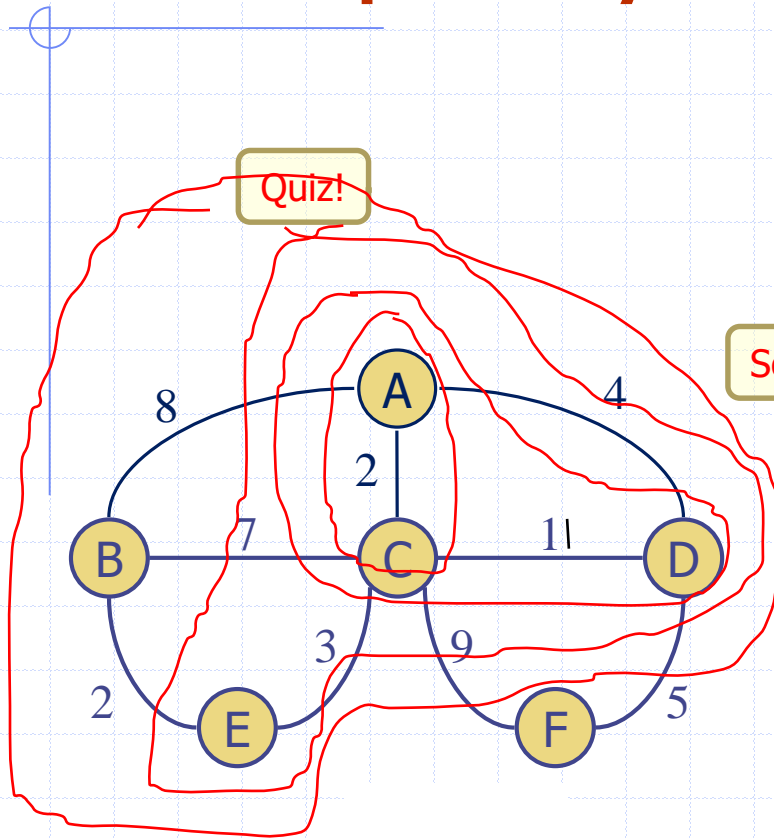
Step-by-step Example



Example (cont.)



Example by Table Filling (1/2)

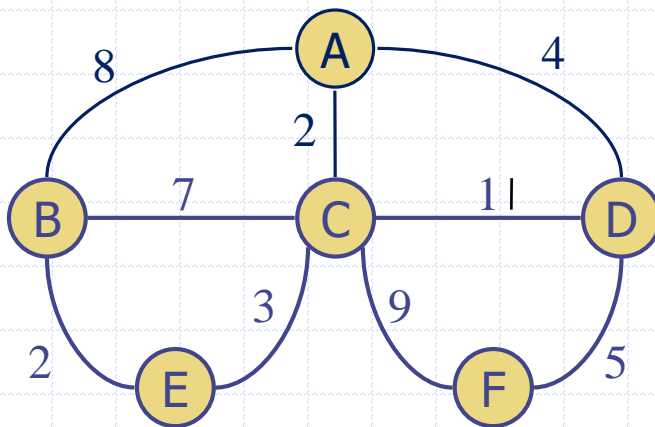


Source node

	B	C	D	E	F
A	8/A	2/A	4/A	∞	∞
	8/A		3/C	5/C	11/C
	8/A			5/C	8/D
	7/E				8/D
					8/D

Example by Table Filling (1/2)

Quiz!



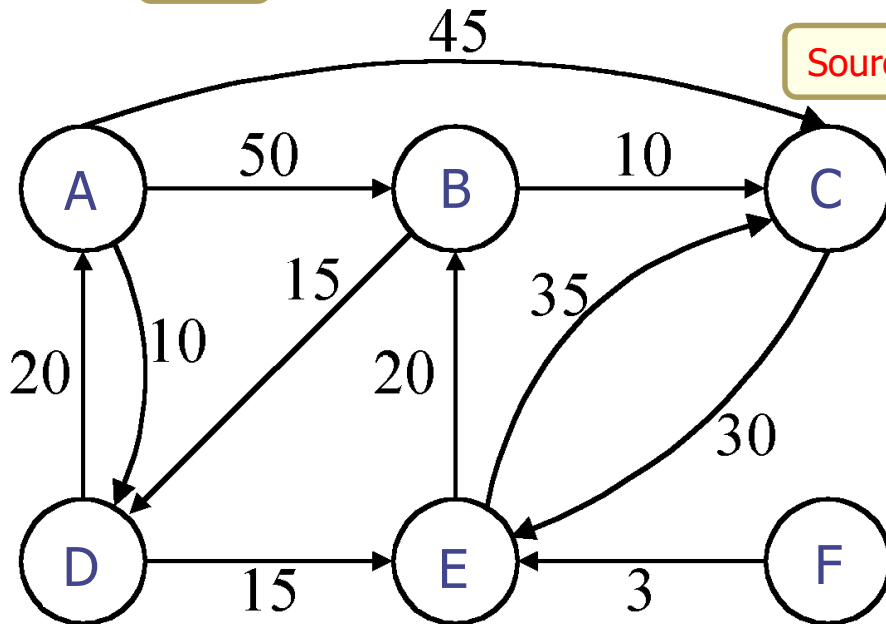
Source node

	B	C	D	E	F
A	8 _A	<u>2</u> _A	4 _A	∞	∞
C			<u>3</u> _C	5 _C	11 _C
D				—	8 _D
E	<u>7</u> _E				
B					—
F					

Selected nodes

Example by Table Filling (2/2)

Quiz!



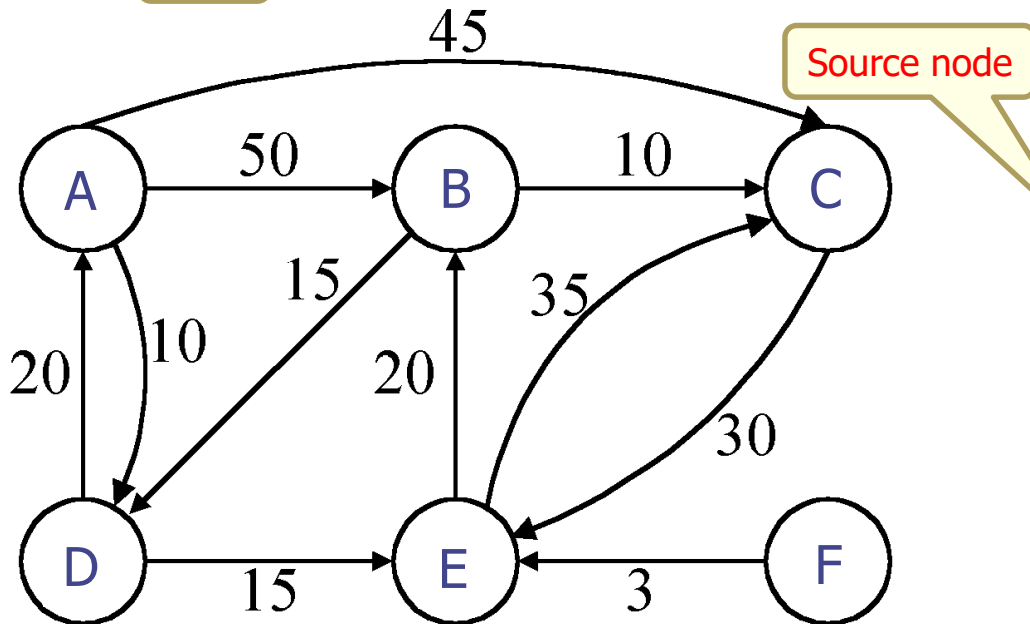
Source node

	B	C	D	E	F
A	50/A	45/A	10/A	∞	∞
D	50/A	45/A		25/D	∞
E	45/E	--			--
B		--			--
C		--			--
F					

Animation

Example by Table Filling (2/2)

Quiz!



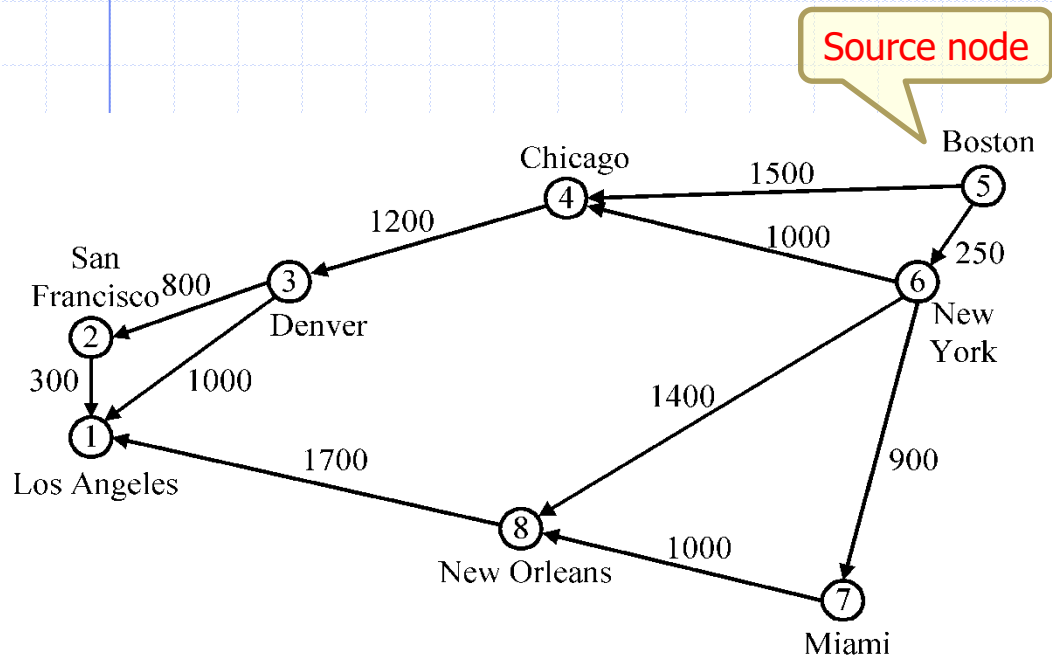
	B	C	D	E	F
A	50 _A	45 _A	<u>10_A</u>	∞	∞
D				<u>25_D</u>	
E	<u>45_E</u>				
B		—			
C					—
F					

Animation

Selected nodes

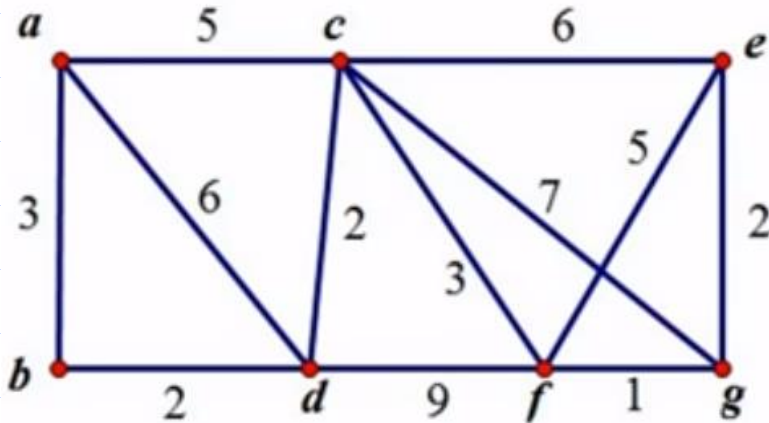
Exercise 1

Quiz!



Exercise 2

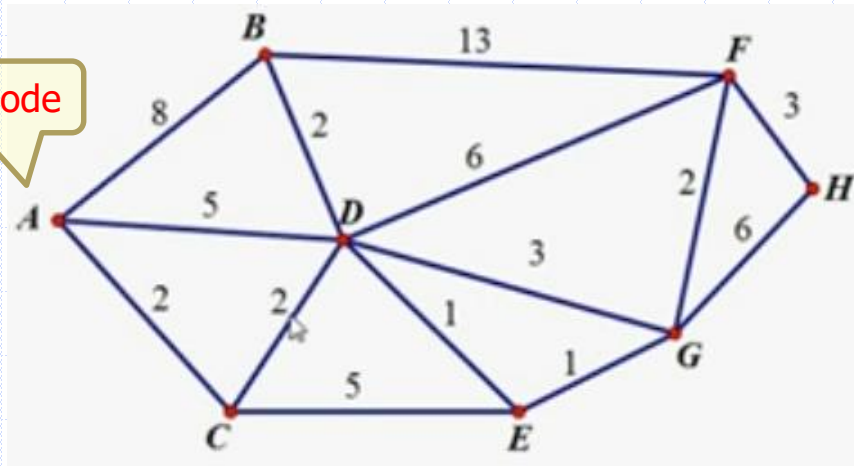
Source node



[Solution!](#)

Exercise 3

Source node



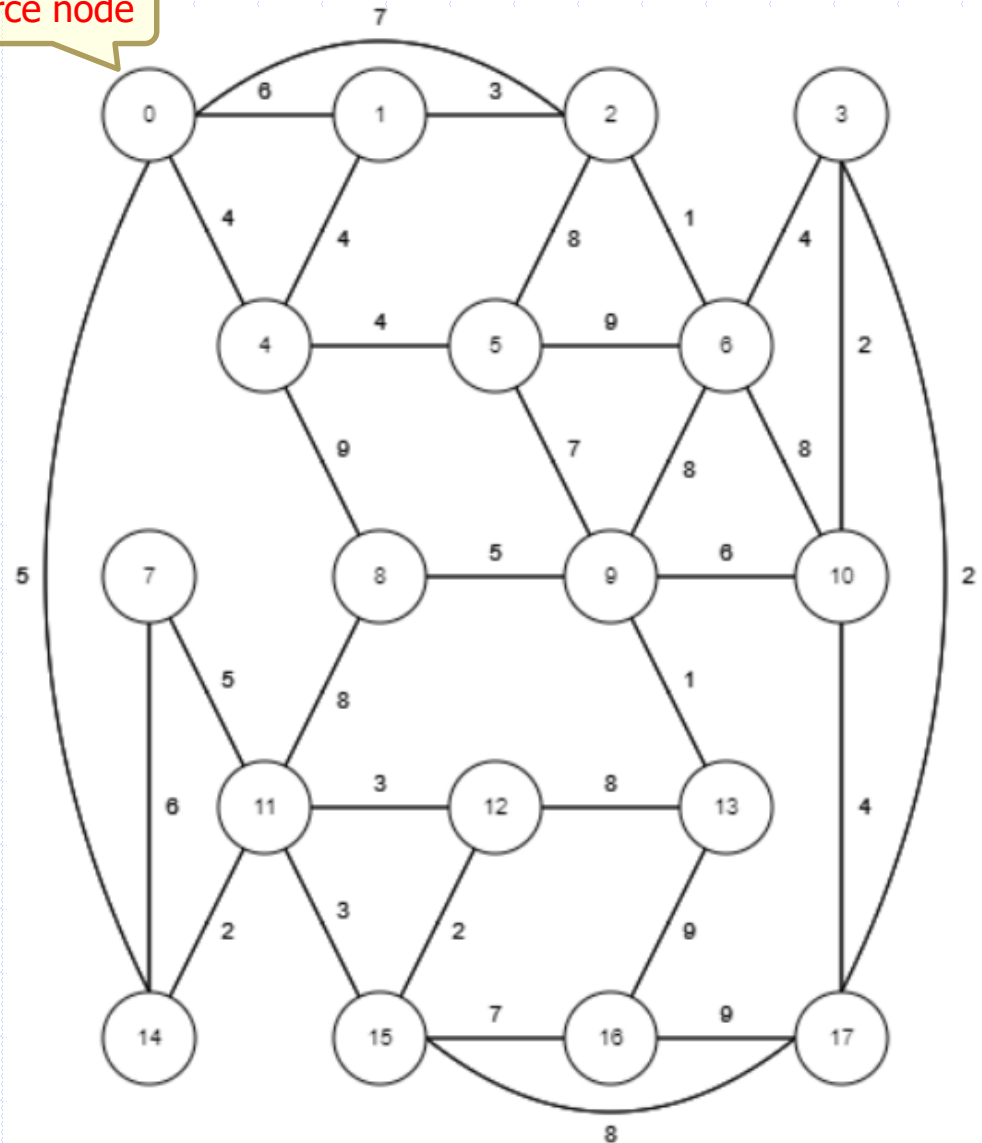
[Solution!](#)

Exercise 4

Source node

Vertex	Known	Cost	Path
0	T	0	-1
1	T	6	0
2	T	7	0
3	T	12	6
4	T	4	0
5	T	8	4
6	T	8	2
7	T	11	14
8	T	13	4
9	T	15	5
10	T	14	3
11	T	7	14
12	T	10	11
13	T	16	9
14	T	5	0
15	T	10	11
16	T	17	15
17	T	14	3

0
 0 1
 0 2
 0 2 6 3
 0 4
 0 4 5
 0 2 6
 0 14 7
 0 4 8
 0 4 5 9
 0 2 6 3 10
 0 14 11
 0 14 11 12
 0 4 5 9 13
 0 14
 0 14 11 15
 0 14 11 15 16
 0 2 6 3 17



Dijkstra's Algorithm

- A heap-based adaptable **priority queue** with location-aware entries stores the vertices outside the cloud
 - Key: distance
 - Value: vertex
 - Recall that method *replaceKey(l,k)* changes the key of entry *l*
- We store **two labels** with each vertex:
 - Distance
 - Entry in priority queue

Algorithm *DijkstraDistances*(*G*, *s*)

```
Q ← new heap-based priority queue
for all v ∈ G.vertices()
    if v = s
        v.setDistance(0)
    else
        v.setDistance(∞)
    l ← Q.insert(v.getDistance(), v)
    v.setEntry(l)
while ¬Q.empty()
    l ← Q.removeMin()
    u ← l.getValue()
    for all e ∈ u.incidentEdges() { relax e }
        z ← e.opposite(u)
        r ← u.getDistance() + e.weight()
        if r < z.getDistance()
            z.setDistance(r)
            Q.replaceKey(z.getEntry(), r)
```

Analysis of Dijkstra's Algorithm

- Graph operations
 - Method `incidentEdges` is called once for each vertex
- Label operations
 - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
 - Setting/getting a label takes $O(1)$ time
- Priority queue operations
 - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
 - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
 - Recall that $\sum_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

Shortest Paths Tree

- Using the template method pattern, we can extend Dijkstra's algorithm to return a **tree of shortest paths** from the start vertex to all other vertices
- We store with each vertex a third label:
 - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label

Easy for backtracking

Algorithm *DijkstraShortestPathsTree*(G, s)

...

for all $v \in G.vertices()$

...

v.setParent(\emptyset)

...

for all $e \in u.incidentEdges()$

{ relax edge e }

$z \leftarrow e.opposite(u)$

$r \leftarrow u.getDistance() + e.weight()$

if $r < z.getDistance()$

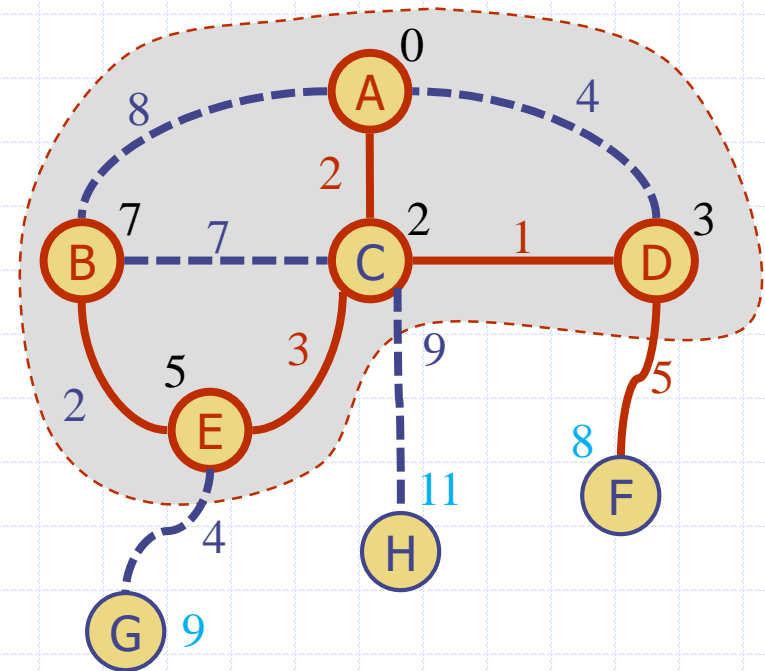
$z.setDistance(r)$

z.setParent(e)

$Q.replaceKey(z.getEntry(), r)$

Why Dijkstra's Algorithm Works

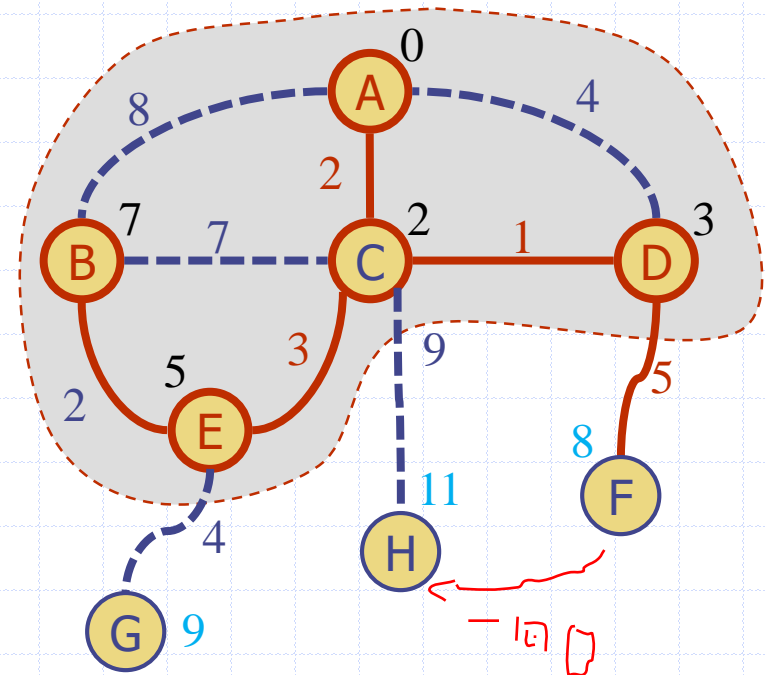
- Dijkstra's algorithm is based on DP. It adds vertices by **increasing distance**.
 - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
 - When the previous node, D, on the true shortest path was considered, its distance was correct
 - But the edge (D,F) was **relaxed** at that time!
 - Thus, so long as $d(F) \geq d(D)$, F's distance cannot be wrong. That is, there is no wrong vertex



Why It Doesn't Work for Negative-Weight Edges

- ◆ Dijkstra's algorithm is based on the DP. It adds vertices **by increasing distance**, which does not hold if we have negative-weight edges.

Quiz!



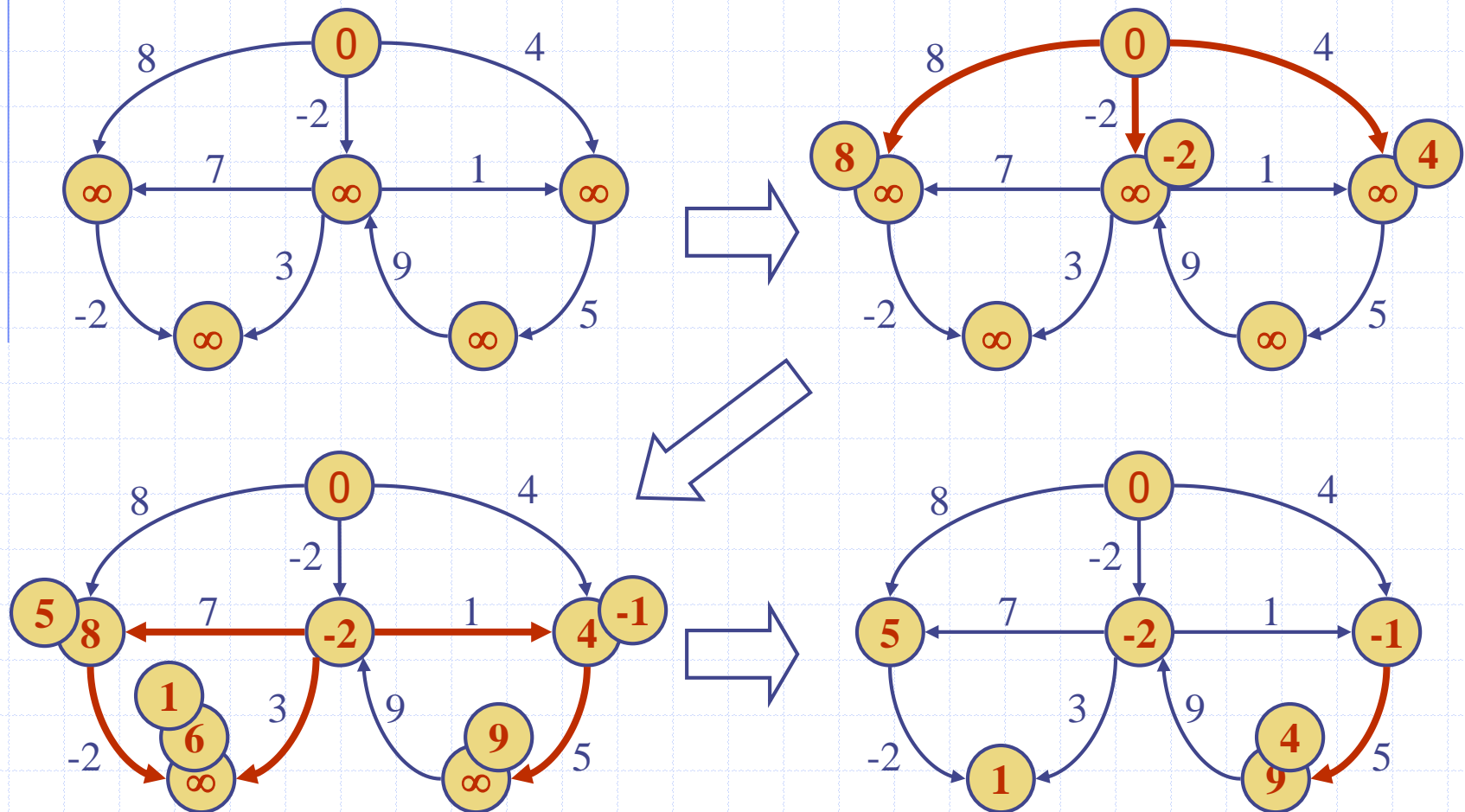
Bellman-Ford Algorithm (not in book)

- ❑ Works even with negative-weight edges
- ❑ Must assume directed edges (for otherwise we would have negative-weight cycles)
- ❑ Iteration i finds all shortest paths that use i edges.
- ❑ Running time: $O(nm)$.
- ❑ Can be extended to detect a negative-weight cycle if it exists
 - How?

```
Algorithm BellmanFord( $G, s$ )  
  for all  $v \in G.vertices()$   
    if  $v = s$   
       $v.setDistance(0)$   
    else  
       $v.setDistance(\infty)$   
  for  $i \leftarrow 1$  to  $n - 1$  do  
    for each  $e \in G.edges()$   
      { relax edge  $e$  }  
       $u \leftarrow e.origin()$   
       $z \leftarrow e.opposite(u)$   
       $r \leftarrow u.getDistance() + e.weight()$   
      if  $r < z.getDistance()$   
         $z.setDistance(r)$ 
```

Bellman-Ford Example

Nodes are labeled with their $d(v)$ values



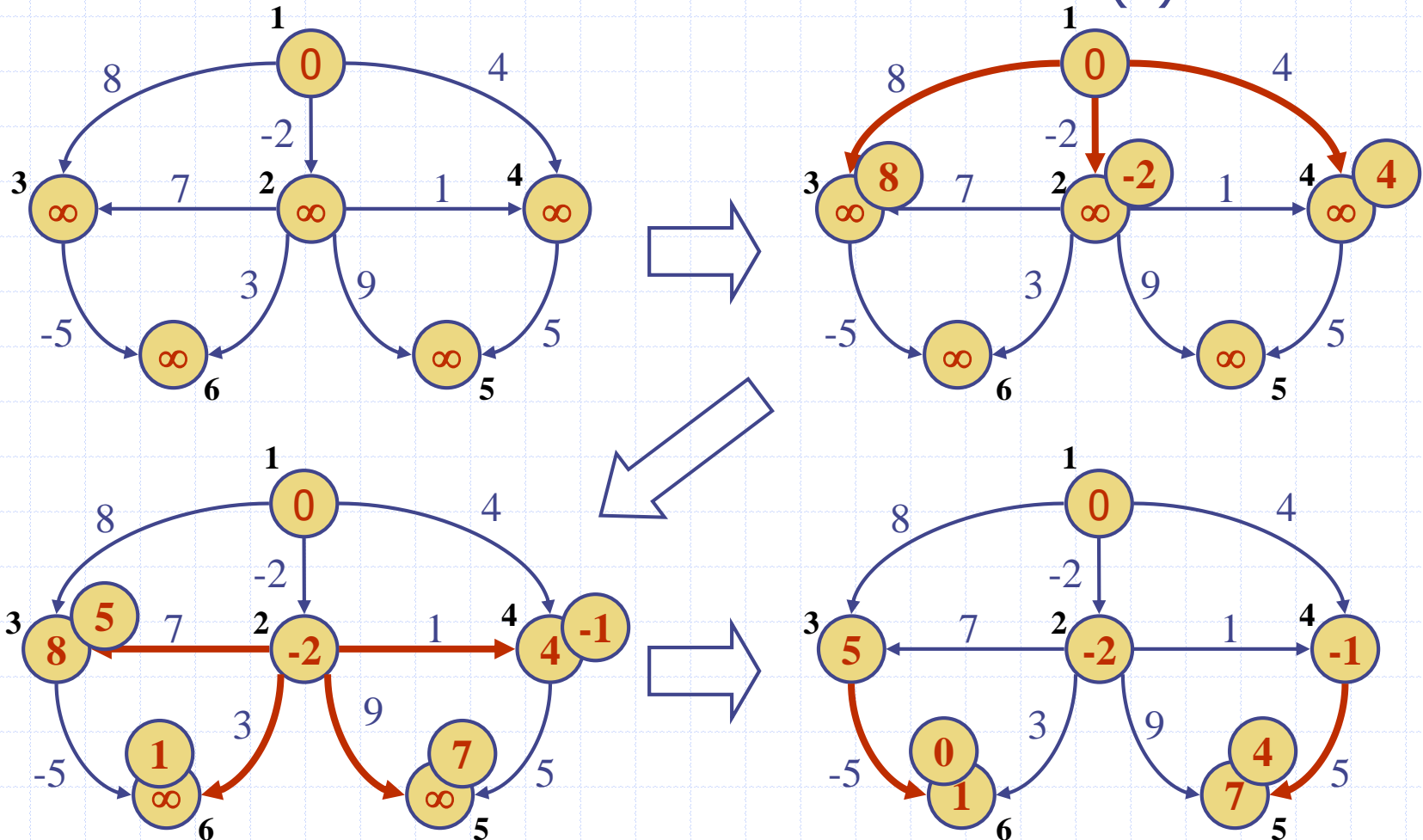
DAG-based Algorithm (not in book)

- ❑ Works even with negative-weight edges
- ❑ Uses topological order
- ❑ Doesn't use any fancy data structures
- ❑ Is much faster than Dijkstra's algorithm
- ❑ Running time: $O(n+m)$.

```
Algorithm DagDistances(G, s)  
  for all v  $\in$  G.vertices()  
    if v = s  
      v.setDistance(0)  
    else  
      v.setDistance( $\infty$ )  
  { Perform a topological sort of the vertices }  
  for u  $\leftarrow$  1 to n do { in topological order }  
    for each e  $\in$  u.outEdges()  
      { relax edge e }  
      z  $\leftarrow$  e.opposite(u)  
      r  $\leftarrow$  u.getDistance() + e.weight()  
      if r < z.getDistance()  
        z.setDistance(r)
```


DAG Example

Nodes are labeled with their $d(v)$ values



Shortest Paths

(two steps)

Resources

□ Youtube tutorials

- [Dijkstra's algorithm](#): Single source all destination
- [Bellman-Ford algorithm](#): Single source all destination
- [Floyd-Warshall algorithm](#): All pairs shortest path

□ Animation

- [Step-by-step animation](#)