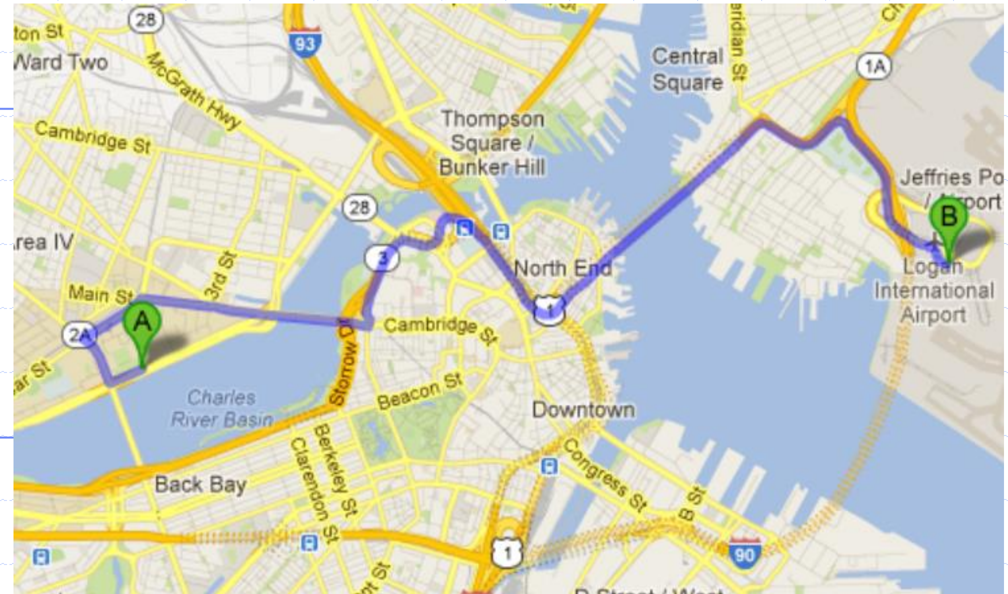
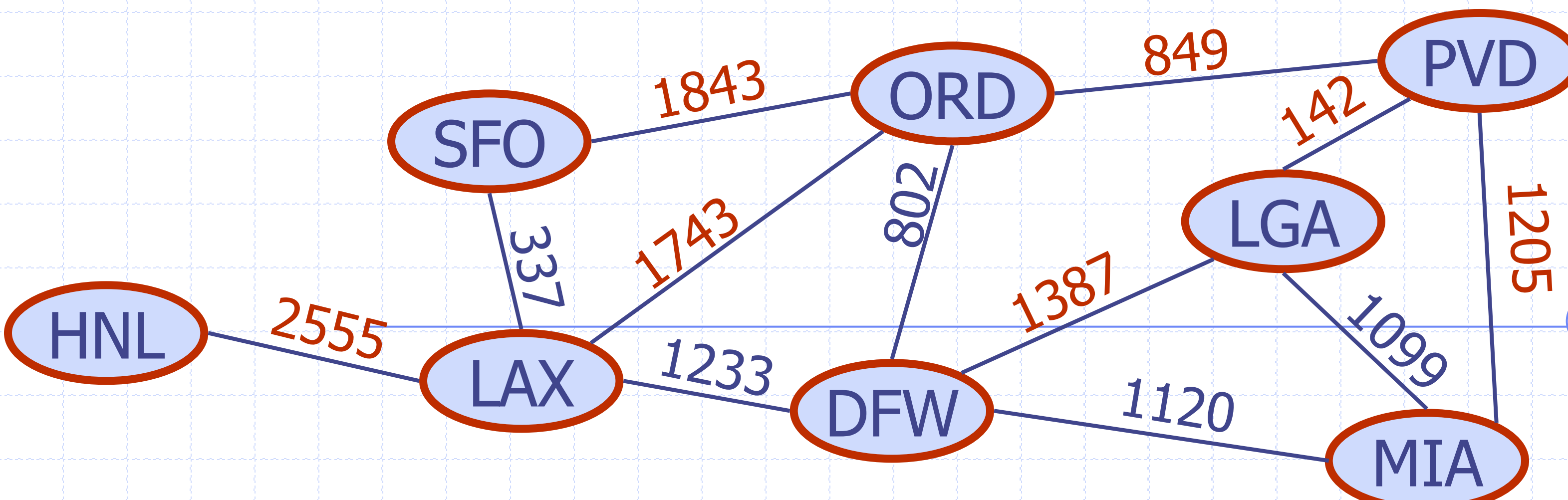
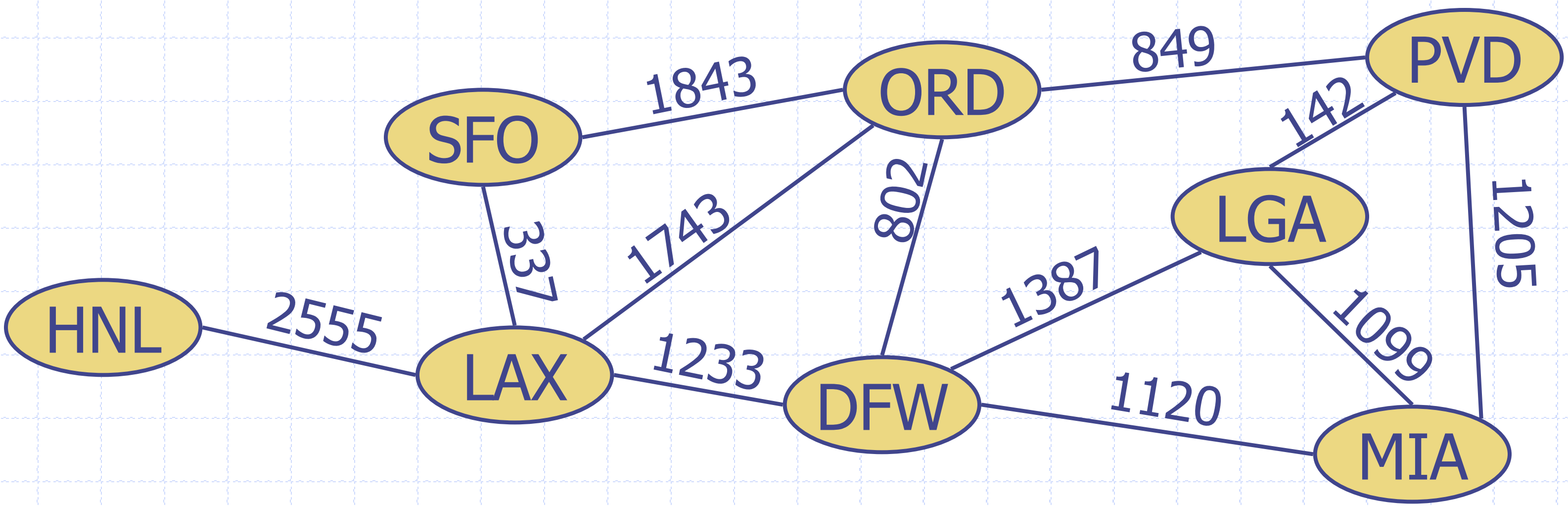
Shortest Paths





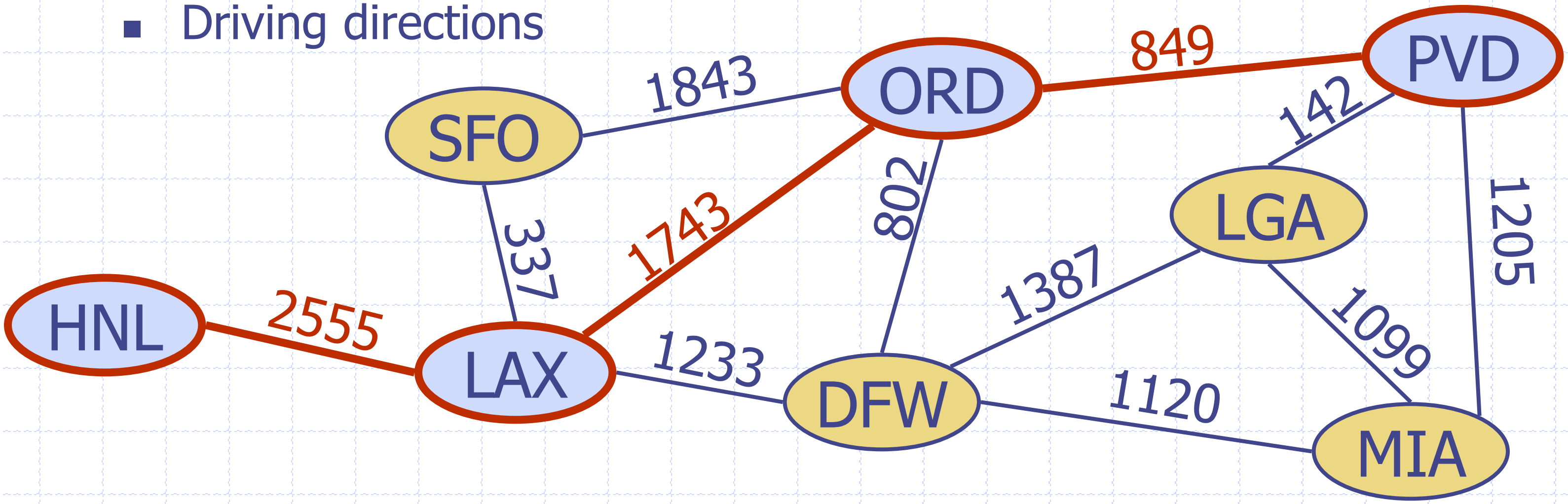
Weighted Graphs

- 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
  - What is the shortest path from HNL to PVD ?



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.***
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Shortest Path Properties

Property 1:

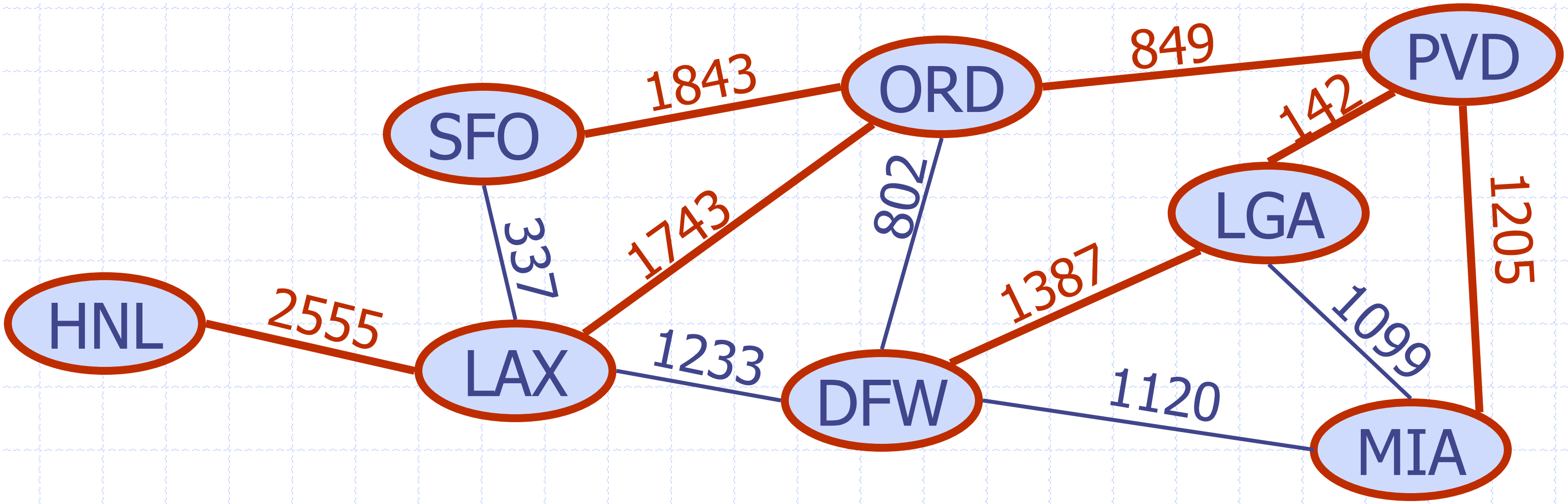
A subpath of a shortest path is itself a shortest path

Property 2:

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

Example:

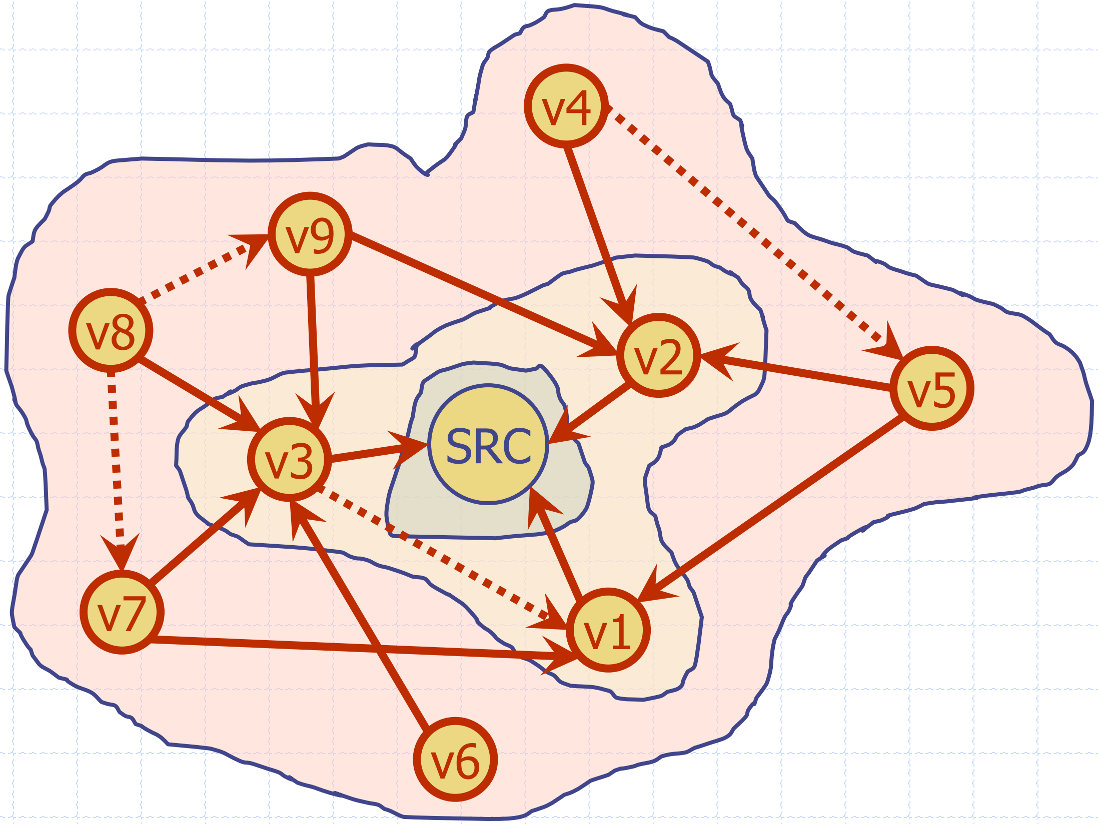
Tree of shortest paths from Providence



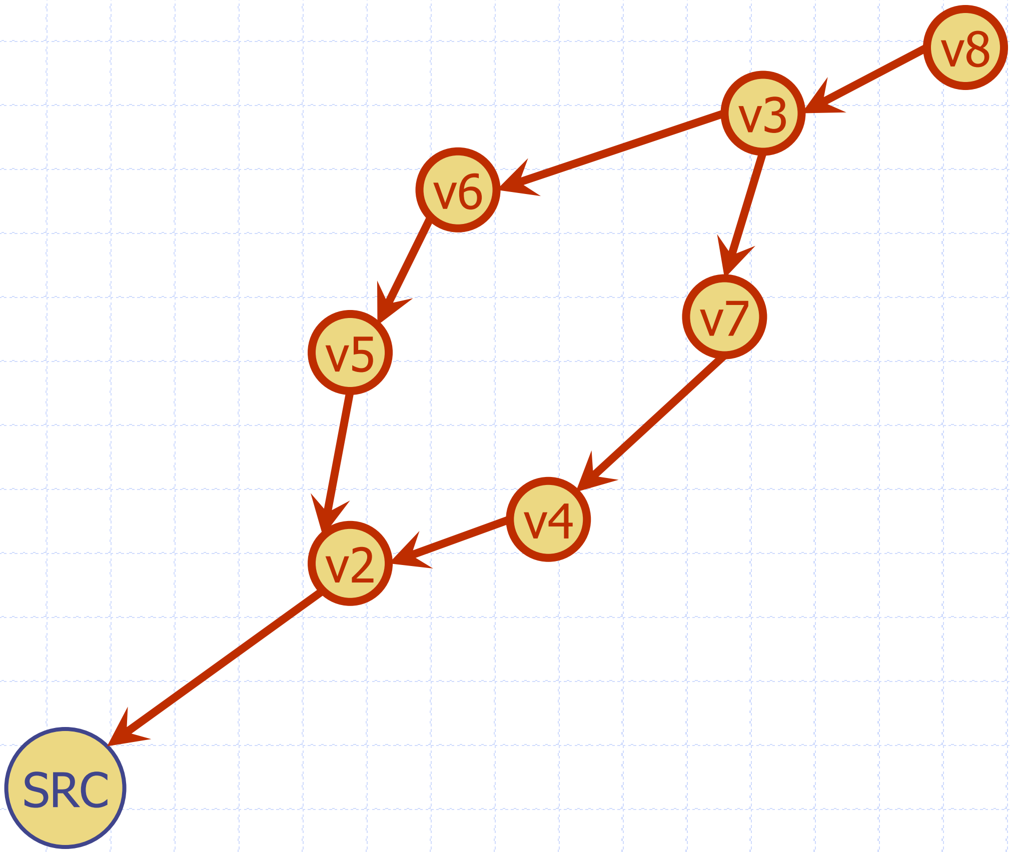
Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are directed
  - 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

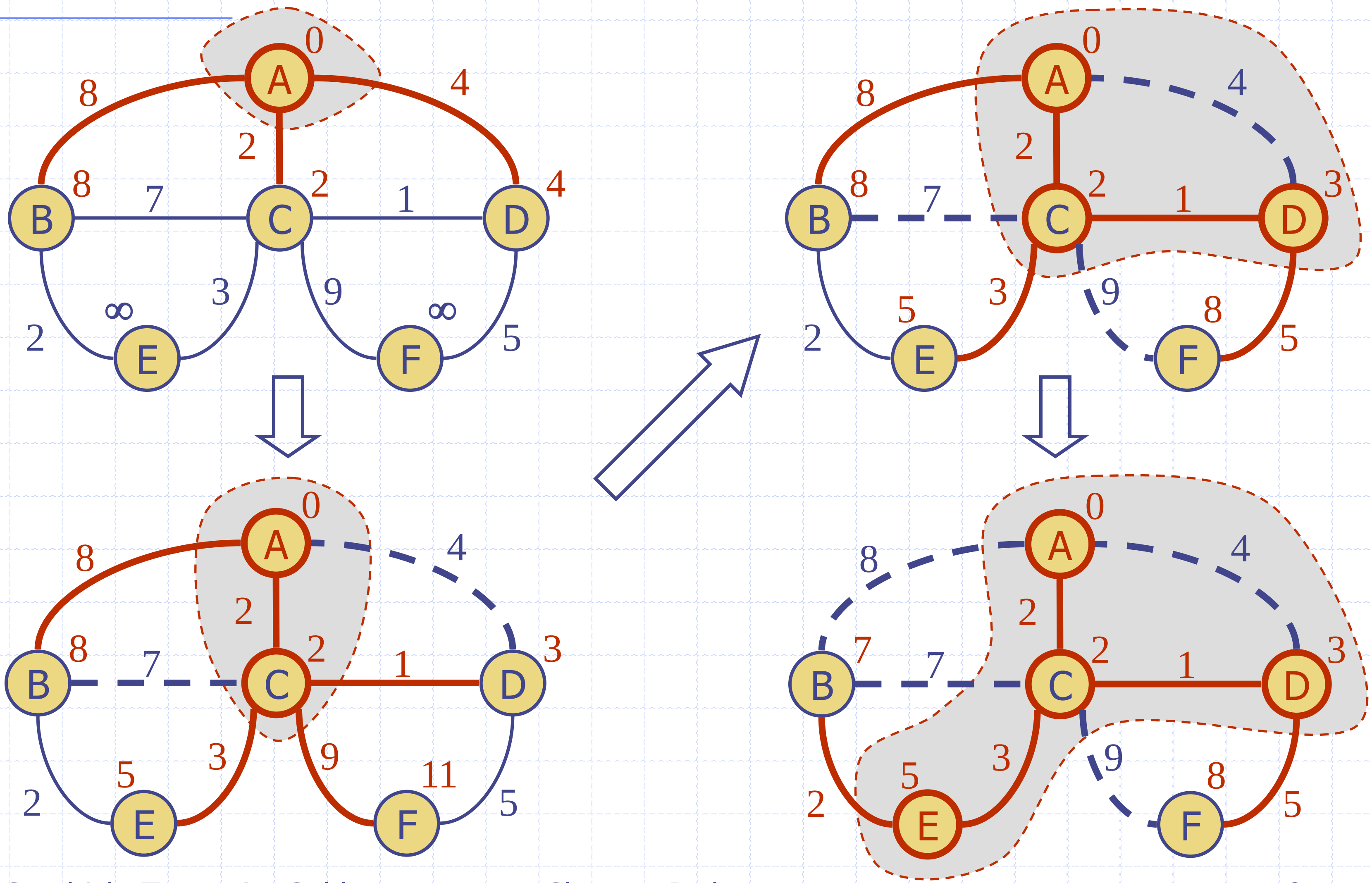
Cloud Progresssion



Correctness Proof

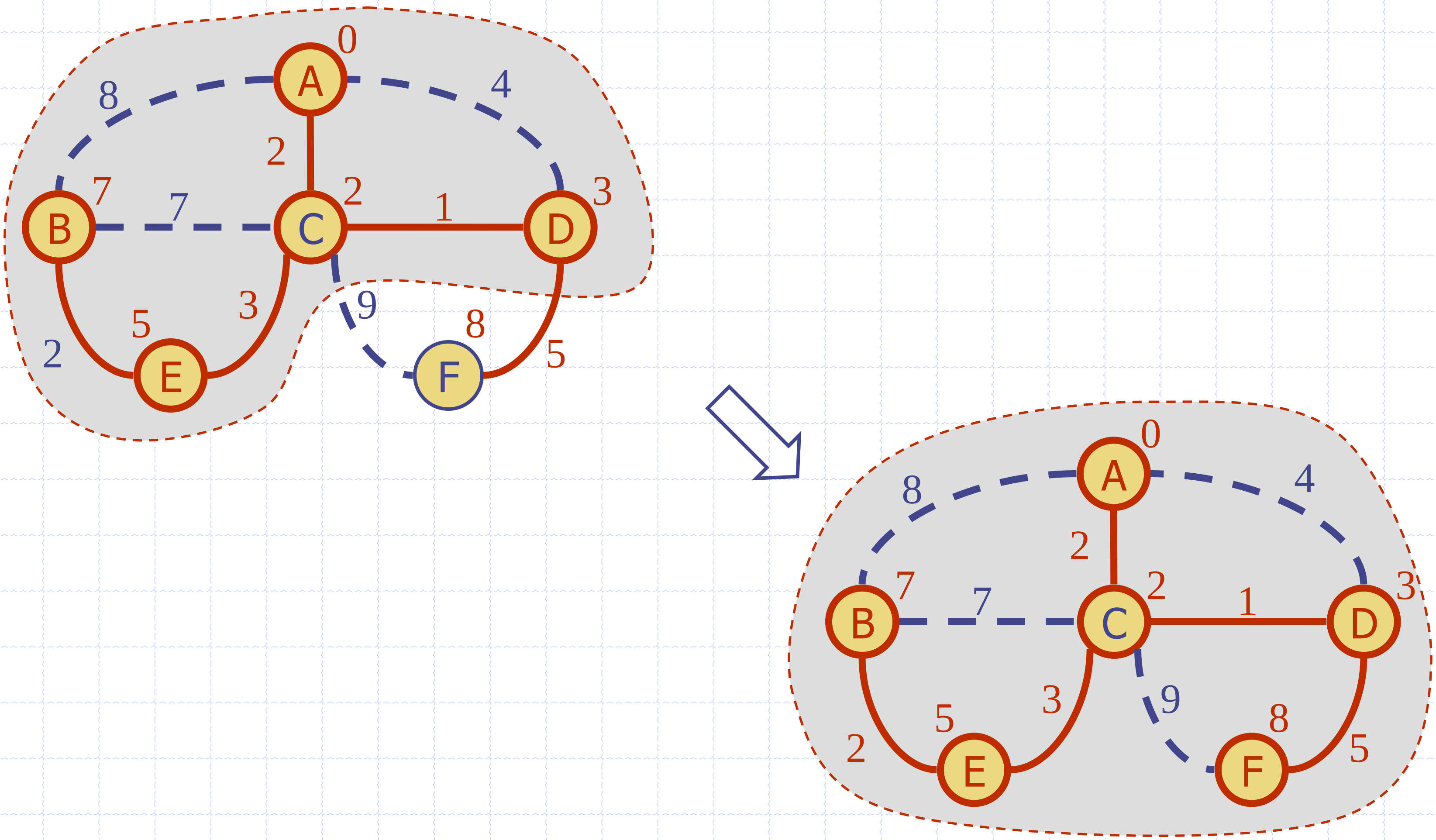


Example



0
A
8
4
2
8
7
2
1
4
B
C
D
3
9
∞
∞
2
5
E
F
0
A
8
4
2
8
7
2
1
3
B
C
D
5
3
9
11
2
5
E
F
0
A
8
4
2
8
7
2
1
3
B
C
D
5
3
9
8
2
5
E
F
0
A
8
4
2
7
7
2
1
3
B
C
D
5
3
9
8
2
5
E
F

Example (cont.)

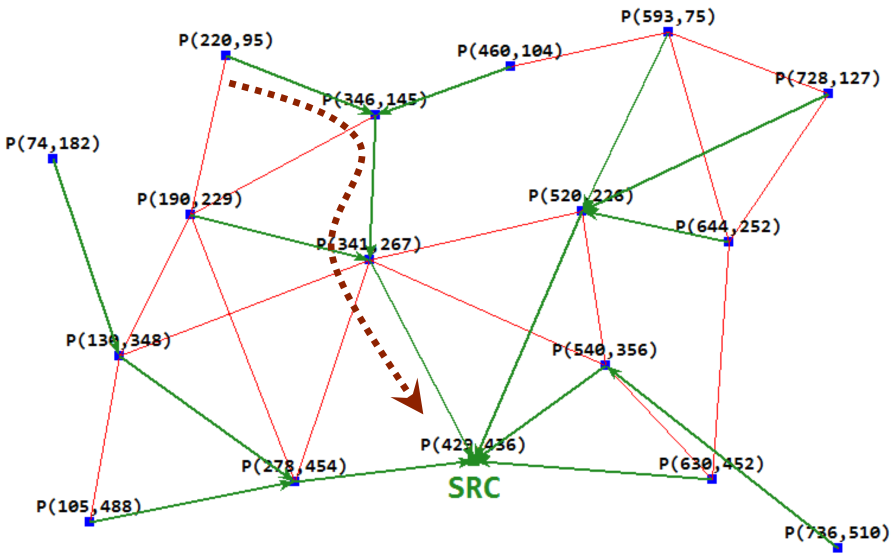


Dijkstra’s Algorithm

```
def dijkstra(g, src):
    cloud = {src: 0}       # cloud of visited vertices/edges and their distance from src
    gps = {}               # gps dictionary maps a vertex to edge toward source src
    distance = {}          # distance dictionary: distance[u] = min distance from u to src
    vertices = set(g.vertices())
    vertices.remove(src)   # src is the single element currently in cloud
    distance[src] = 0      # distance from src to itself is 0
    for u in vertices:     # distance of any other vertex to source is infinity
        distance[u] = float('Infinity')

    while True:
        # Construct the next ring
        ring = []
        for v in cloud:
            for edge in g.incident_edges(v, False):   # incoming edges to v
                u = edge.opposite(v)
                du = distance[v] + edge.element()
                if du < distance[u]:
                    distance[u] = du
                    gps[u] = edge
                if u not in cloud:
                    ring.append(u)
 if not ring:
            break

        for u in ring:
            cloud[u] = distance[u]

    return cloud, gps
```

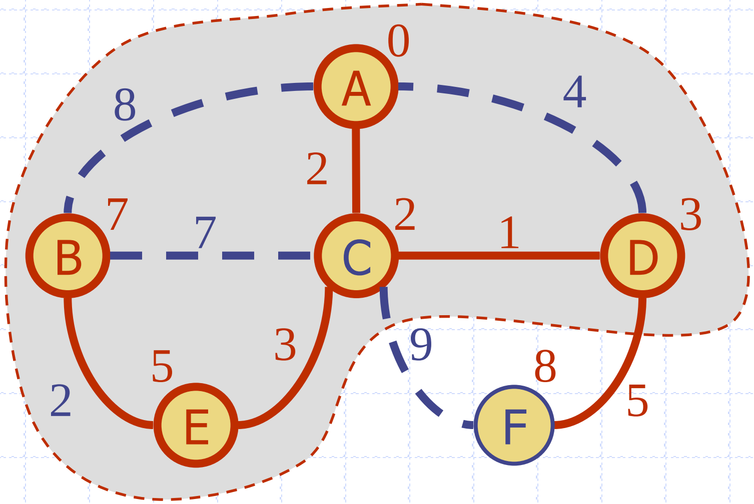
Shortest Path

```
# Given a graph g, a cloud tree as above
# we can easily compute a path from source to destination

def shortest_path(g, tree, source, destination):
    path = []
    v = destination
    while True:
        if not v in tree:
            break
        e = tree[v]
        path.append((v,e))
        v = e.opposite(v)

    return path
```



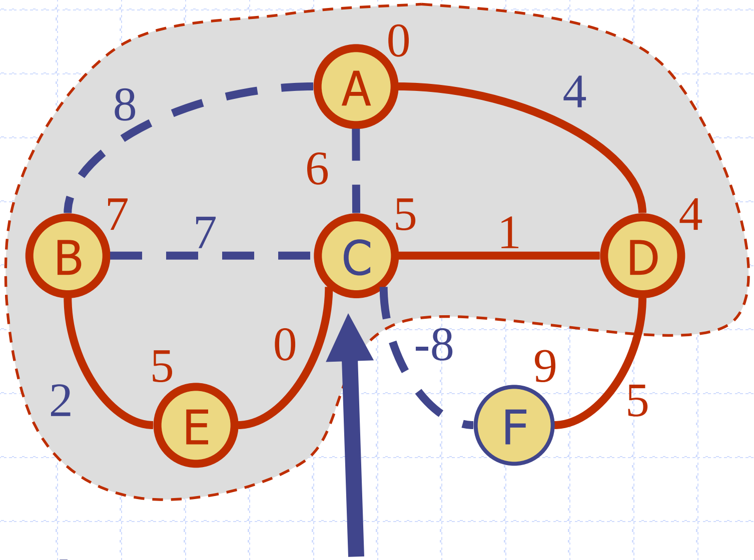
Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method. 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 greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C’s true distance is 1, but it is already in the cloud with d(C)=5!

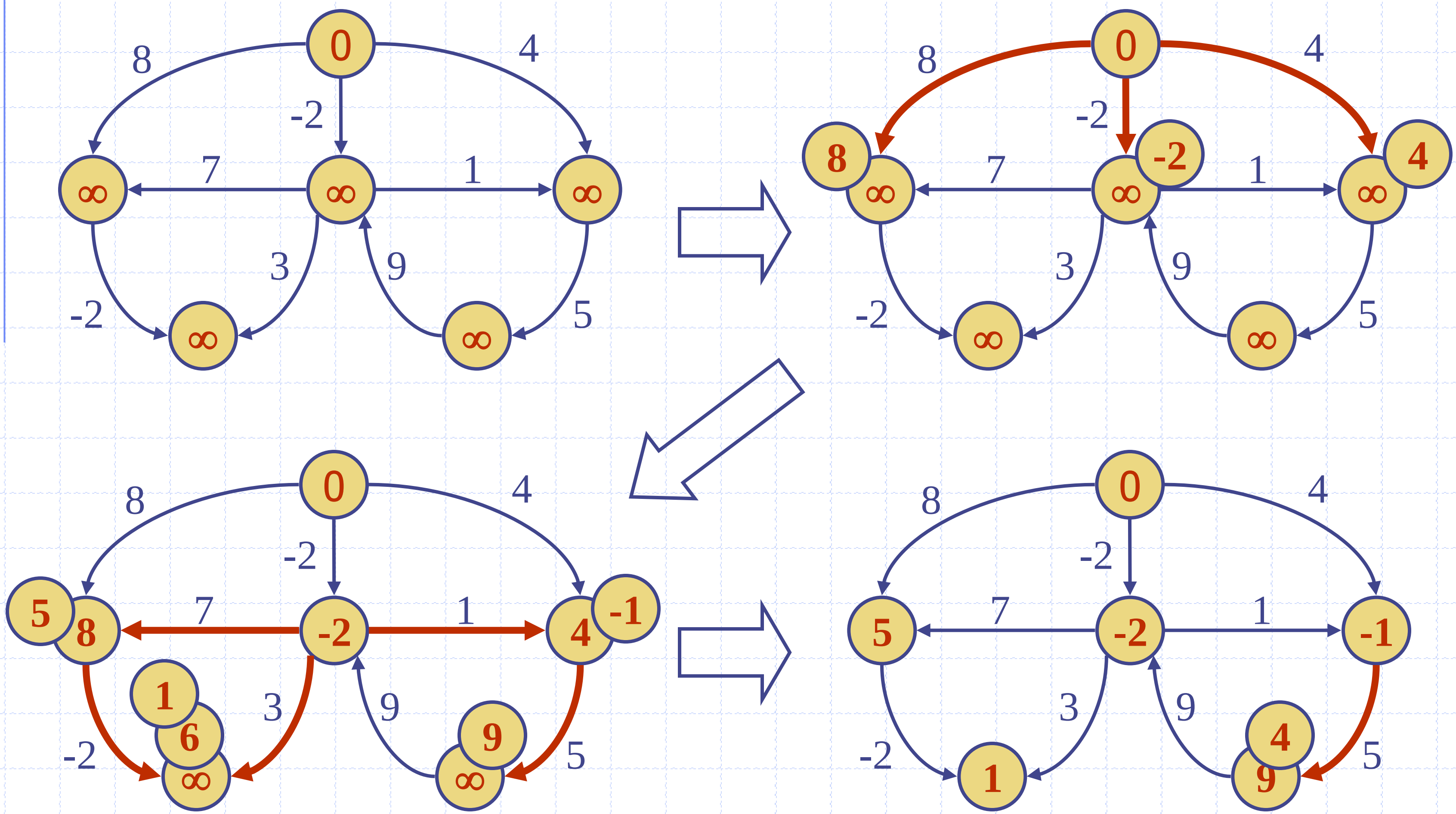
Bellman-Ford Algorithm (not in book)

- 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 ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  for i ← 1 to n – 1 do
    for each e ∈ G.edges()
      { relax edge e }
      u ← G.origin(e)
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
```

Bellman-Ford Example

Nodes are labeled with their d(v) values



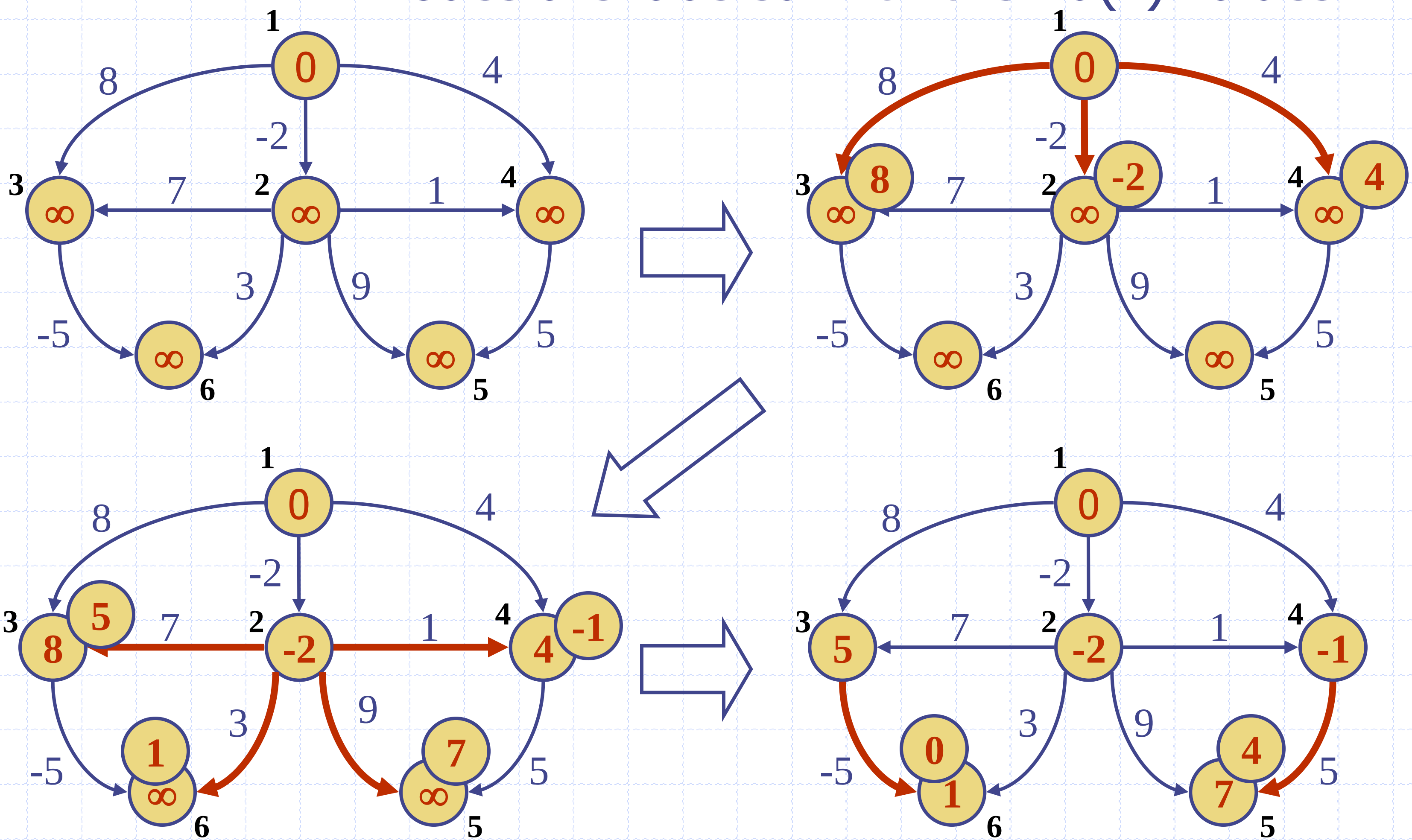
DAG-based Algorithm (not in book)

- 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 ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  { Perform a topological sort of the vertices }
  for u ← 1 to n do    {in topological order}
    for each e ∈ G.outEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
```

DAG Example

Nodes are labeled with their d(v) values



(two steps)