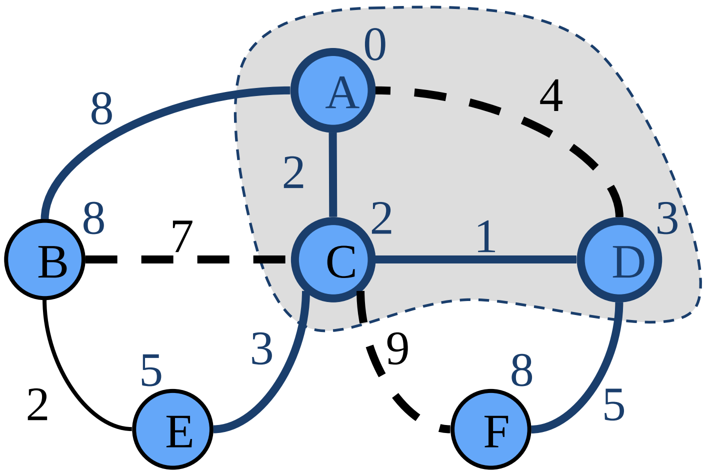
Shortest Paths

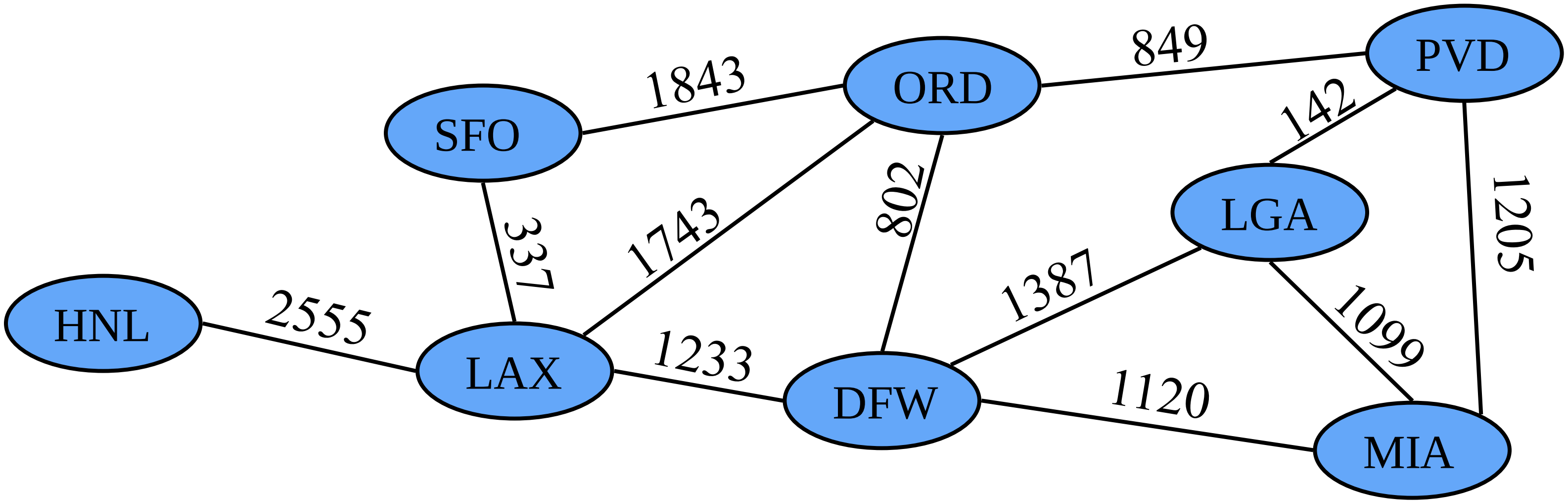


Outline and Reading

- Weighted graphs (7.1)
  - Shortest path problem
  - Shortest path properties
- Dijkstra’s algorithm (7.1.1)
  - Algorithm
  - Edge relaxation
- The Bellman-Ford algorithm (7.1.2)
- Shortest paths in DAGs (7.1.3)
- All-pairs shortest paths (7.2.1)

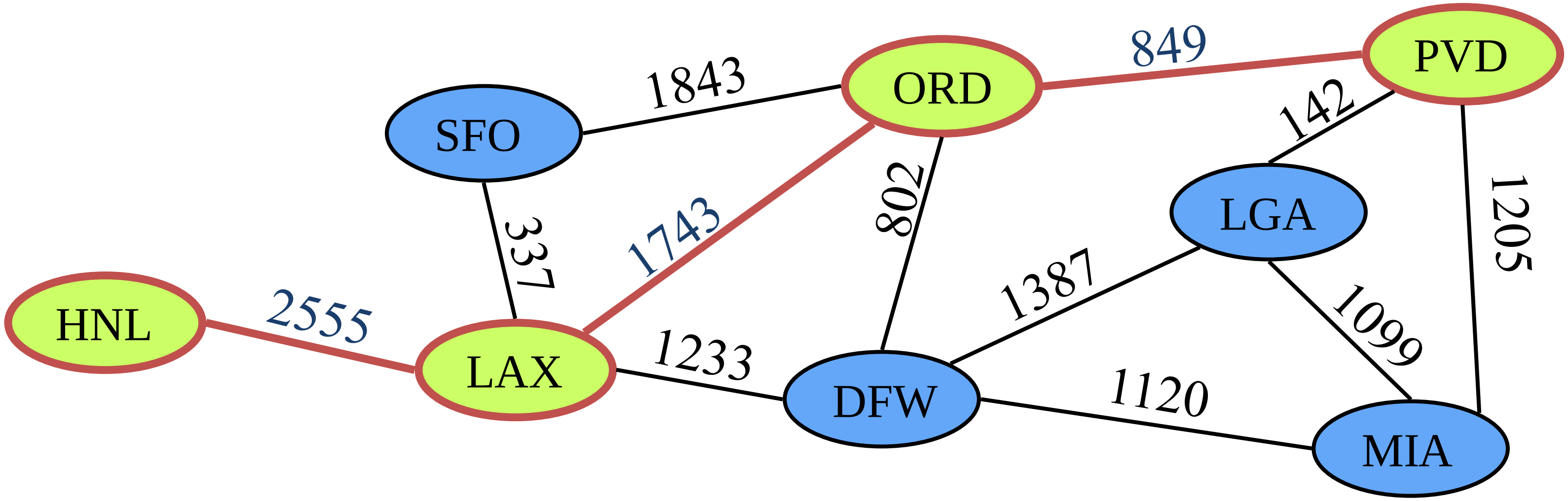
Weighted Graphs

- 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 Path Problem

- 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

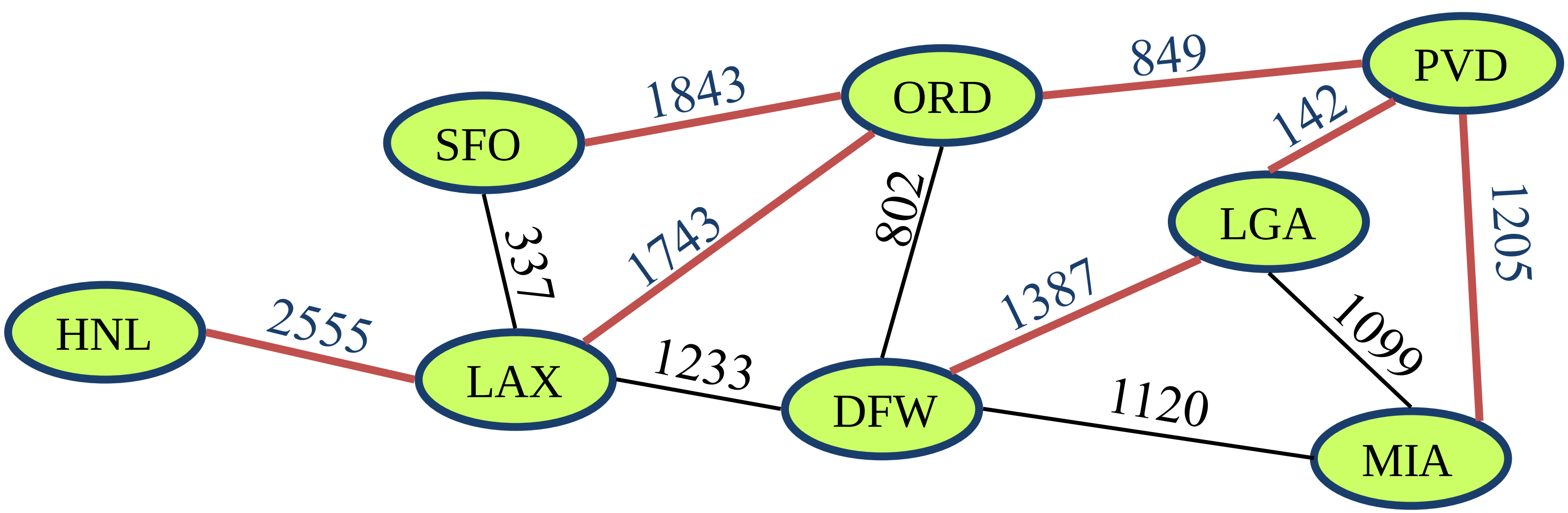


Shortest Path Problem

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 other vertices.

- Example: tree of shortest paths from Providence



Dijkstra's Algorithm

The distance of vertex ***v*** from ***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 undirected
  - the edge weights are nonnegative

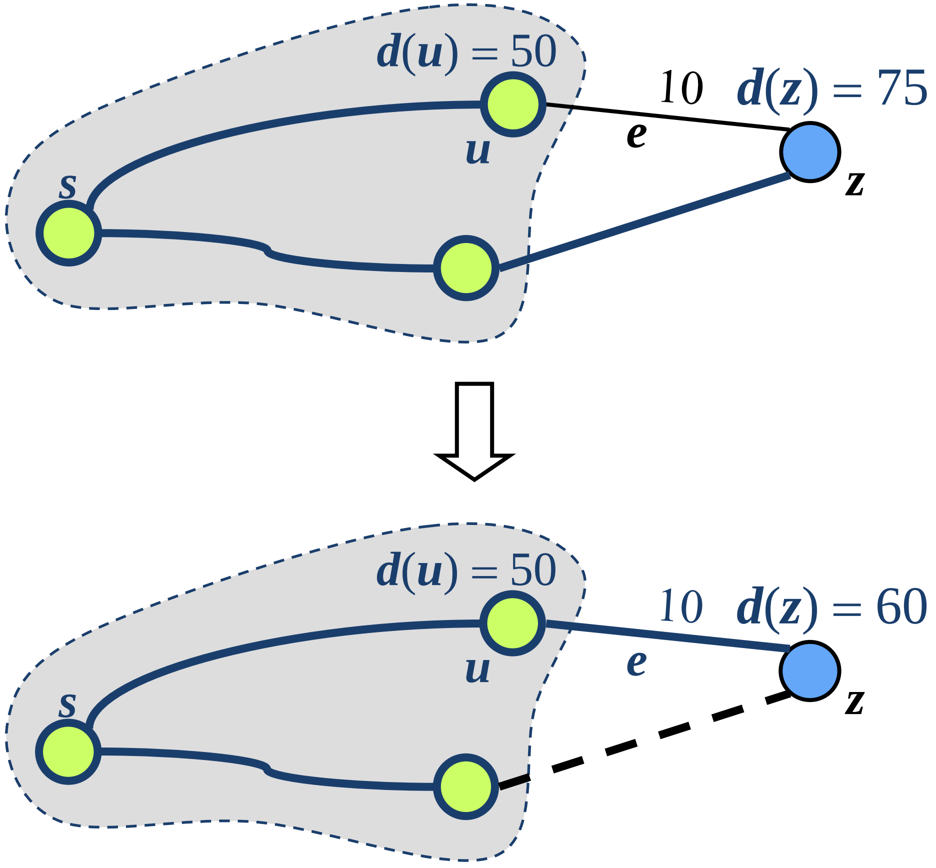
Idea:

- Grow a "cloud" of vertices, beginning with ***s*** and eventually covering all vertices
- 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
  - Add to the cloud the vertex ***u*** outside the cloud with the smallest distance label, ***d***(***u***)
  - Update the labels of the vertices adjacent to ***u***

Edge Relaxation

Consider an edge $e = (u,z)$ such that

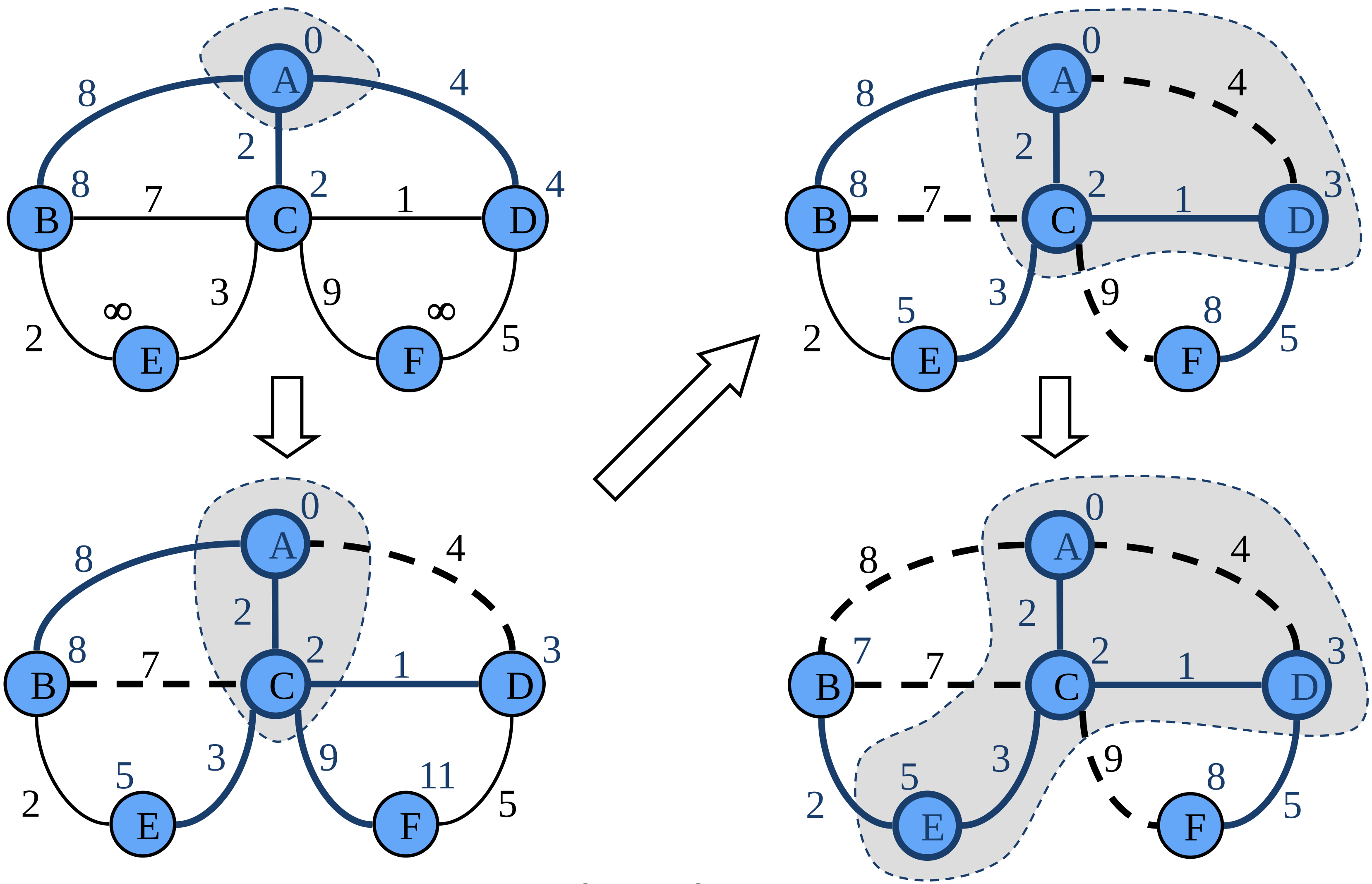
- u is the vertex most recently added to the cloud
- z is not in the cloud



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

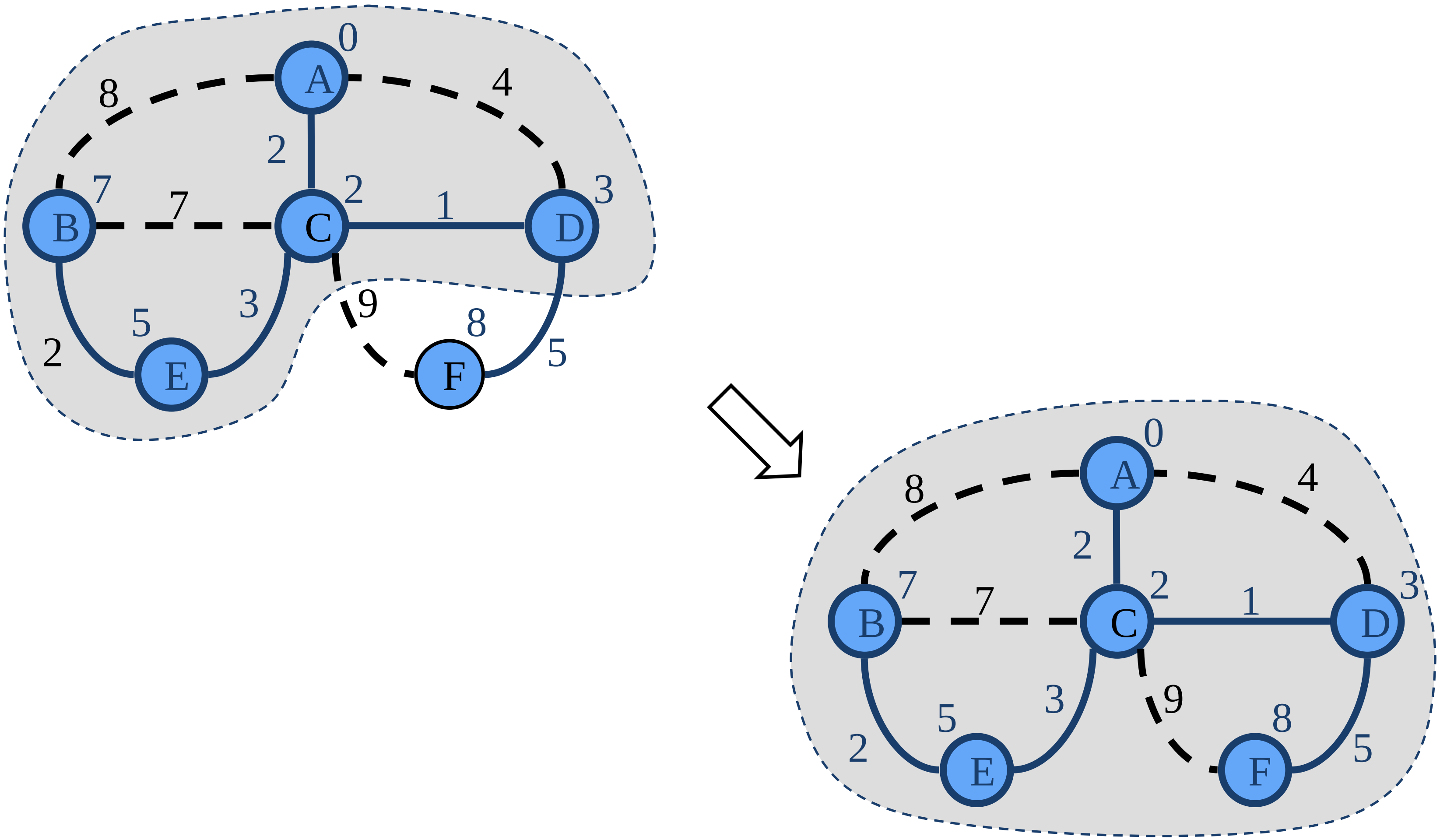
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$

Example



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

Example (cont.)



Dijkstra's Algorithm

A priority queue stores the vertices outside the cloud

- Key: distance
- Element: vertex

Locator-based methods

- *insert*(*k*,*e*) returns a locator
- *replaceKey*(*l*,*k*) changes the key of an item

We store two labels with each vertex:

- Distance ($d(v)$ label)
- locator in priority queue

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

Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex

- Label operations
  - We set/get the distance and locator labels of vertex $\boldsymbol{z}$ $\boldsymbol{O}(\deg(\boldsymbol{z}))$ times
  - Setting/getting a label takes $\boldsymbol{O}(1)$ time

- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $\boldsymbol{O}(\log \boldsymbol{n})$ time
  - The key of a vertex in the priority queue is modified at most $\deg(\boldsymbol{w})$ times, where each key change takes $\boldsymbol{O}(\log \boldsymbol{n})$ time

- Dijkstra's algorithm runs in $\boldsymbol{O}((\boldsymbol{n} + \boldsymbol{m}) \log \boldsymbol{n})$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2\boldsymbol{m}$

- The running time can also be expressed as $\boldsymbol{O}(\boldsymbol{m} \log \boldsymbol{n})$ since the graph is connected.

Extension

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

- Store with each vertex a third label:
  - parent edge in the shortest path tree
- In the edge relaxation step, update the parent label

Algorithm *DijkstraShortestPathsTree*(*G*, *s*)

    …

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

    …

    setParent(v, $\emptyset$)

    …

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

      { relax edge ***e*** }

      $z \leftarrow G.opposite(u,e)$

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

      if $r < getDistance(z)$

        $setDistance(z,r)$

        $setParent(z,e)$

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

Why Dijkstra's Algorithm Works

Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

Claim: Whenever a vertex u is pulled into the cloud, $D[u] = d(v, u)$.

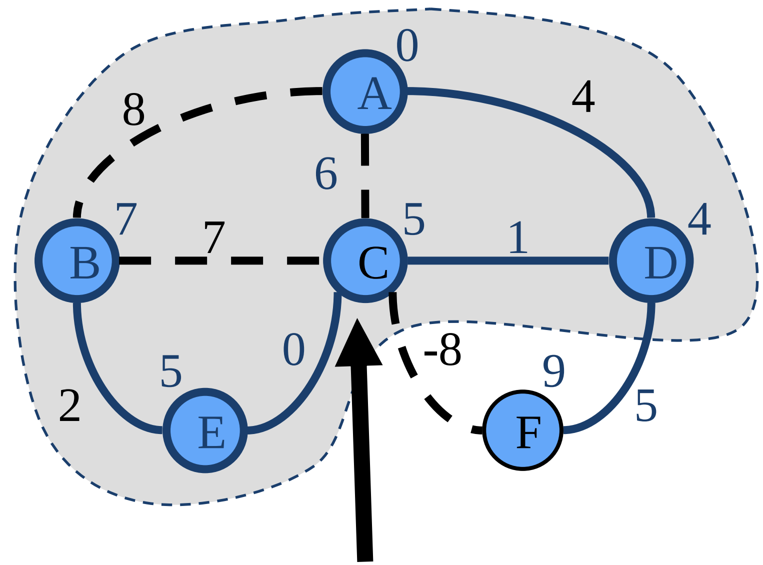
Outline of Proof (by contradiction):

- Suppose $\boldsymbol{u}$ is the **first** vertex such that $D[u] > d(v, u)$.
- Let $\boldsymbol{z}$ be the first vertex on the shortest v-u path P which hasn't been pulled into the cloud yet, and let $\boldsymbol{y}$ be the vertex before z on P.
    - Then, $D[z] = d(v, z)$.
    - Since z is on shortest v-u path, $d(v, z) + d(z, u) = d(v, u)$.
    - Since u is processed before z, $D[u] \leq D[z]$.
- $D[u] \leq D[z] = d(v, z) \leq d(v, z) + d(z, u) = d(v, u)$, a contradiction.

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.
- This violates the greedy property.



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

Bellman-Ford Algorithm

- Works even with negative-weight edges
- Must assume directed edges (otherwise we would have negative-weight cycles)
- Iteration ***i*** finds all shortest paths that use *i* edges beginning at ***s***

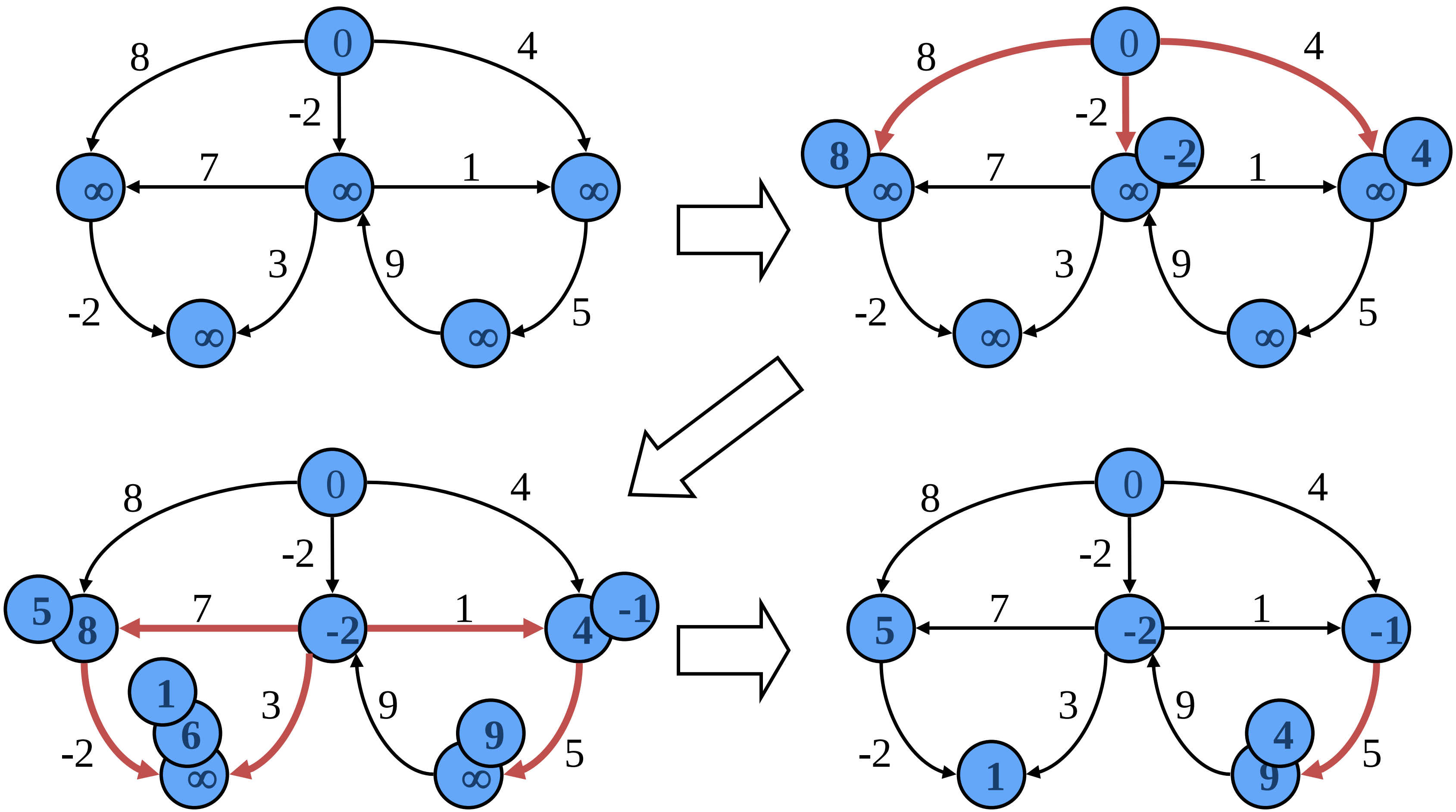
- Running time: $O(nm)$.

- Can be extended to detect a negative-weight cycle if it exists
  - How?

Algorithm ***BellmanFord(G, s)***
    for all $v \in$ ***G.vertices()***
        if $v = s$
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    for $i \leftarrow 1$ **to** $n-1$ **do**
        for each (directed) **edge** $e=(u,z) \in$ ***G.edges()***
            { relax edge ***e*** }
            $r \leftarrow$ ***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

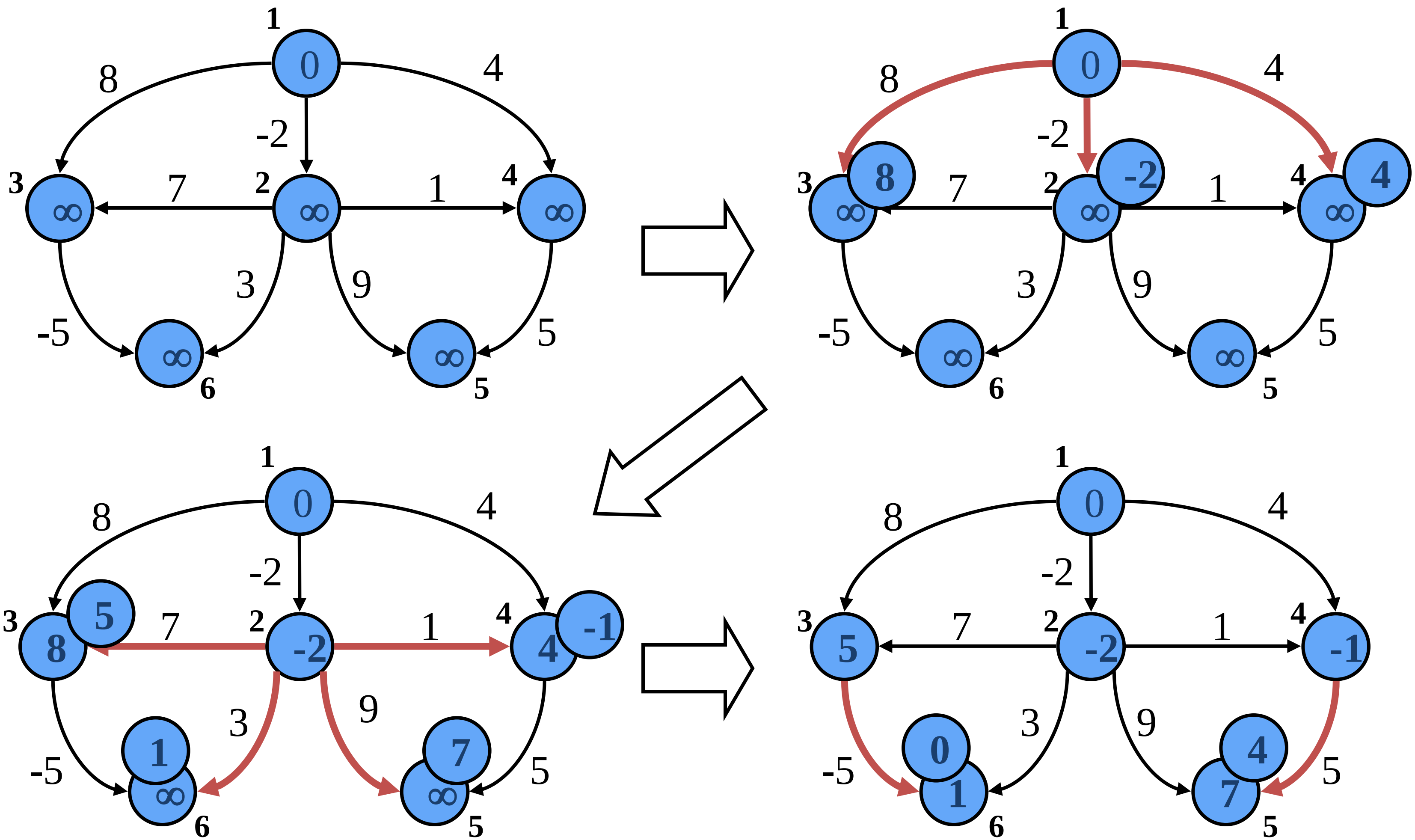
- Assumes *G* is a DAG
- Works even with negative-weight edges
- Uses topological order
- Much faster than Dijkstra's algorithm

- 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 edge e=(u,z) ∈ G.edges()
      { relax edge 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)

All-Pairs Shortest Paths

Find the distance between every pair of vertices in a weighted directed graph G.

- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes $O(nm\log n)$ time.
- Likewise, n calls to Bellman-Ford would take $O(n^2m)$ time.

We can achieve $O(n^3)$ time using the Floyd-Warshall dynamic programming algorithm.

Algorithm ***AllPair***(G) {assumes vertices $1,\ldots,n$}
 for all *vertex pairs (i,j)*
   if $i = j$
     $D_0[i,i] \leftarrow 0$
   else if (i,j) *is an edge in G*
     $D_0[i,j] \leftarrow$ *weight of edge* (i,j)
   else
     $D_0[i,j] \leftarrow +\infty$
 for $k \leftarrow 1$ **to** n **do**
   for $i \leftarrow 1$ **to** n **do**
     for $j \leftarrow 1$ **to** n **do**
       $D_k[i,j] \leftarrow \min\{ D_{k-1}[i,j],\ D_{k-1}[i,k]+D_{k-1}[k,j] \}$
 return D_n