SE274 Data Structure

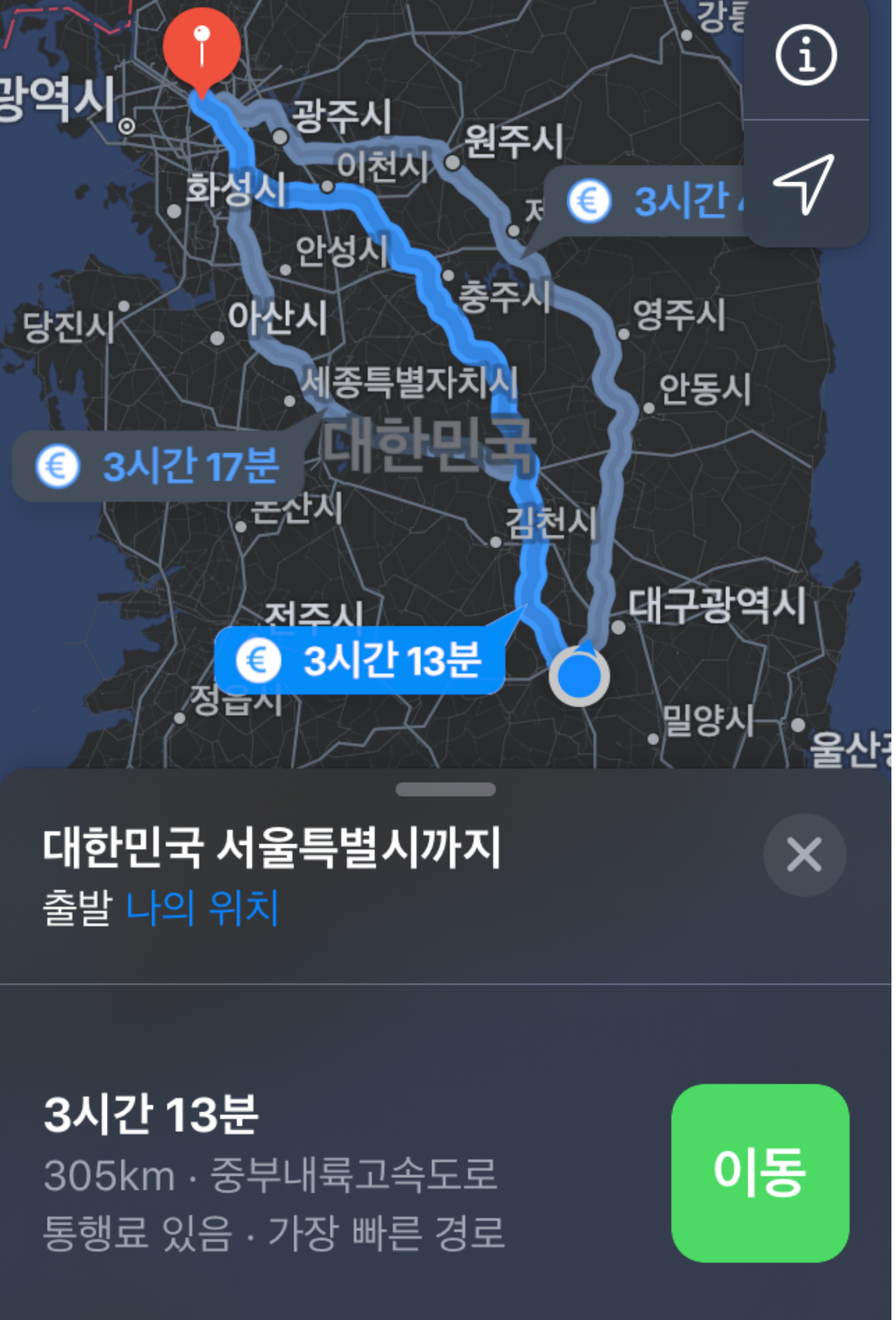
Lecture 9: Graphs

(textbook: Chapter 14)

May *18*, 2020

Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

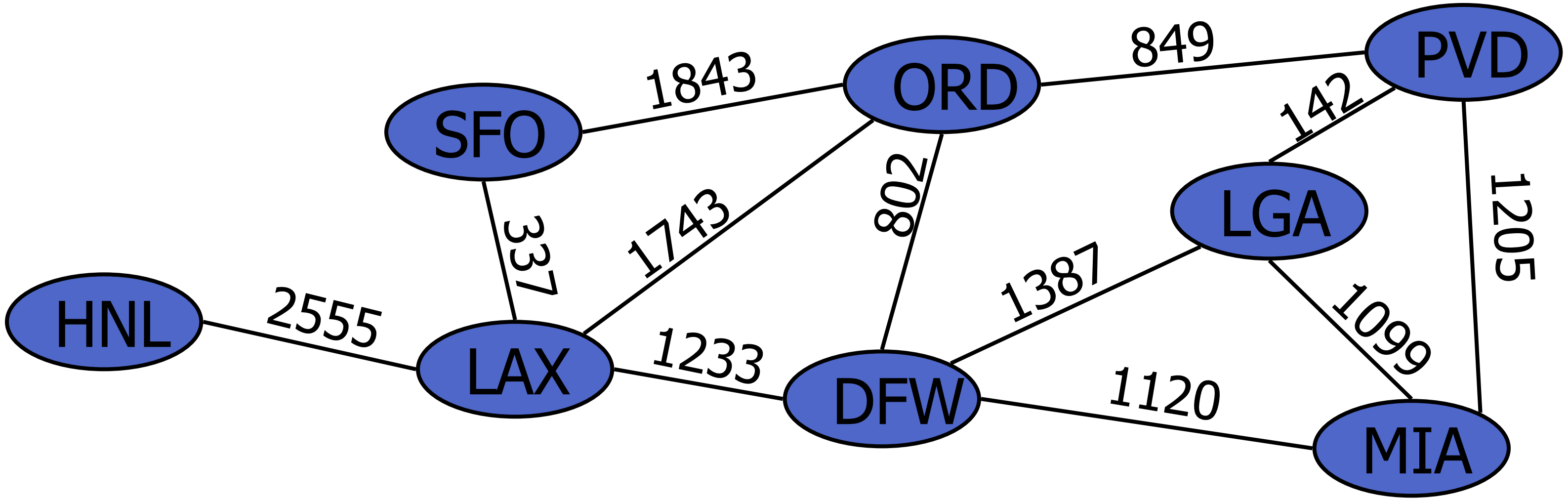


Problem: shortest path

- Finding the fastest/shortest/cheapest route in a map application

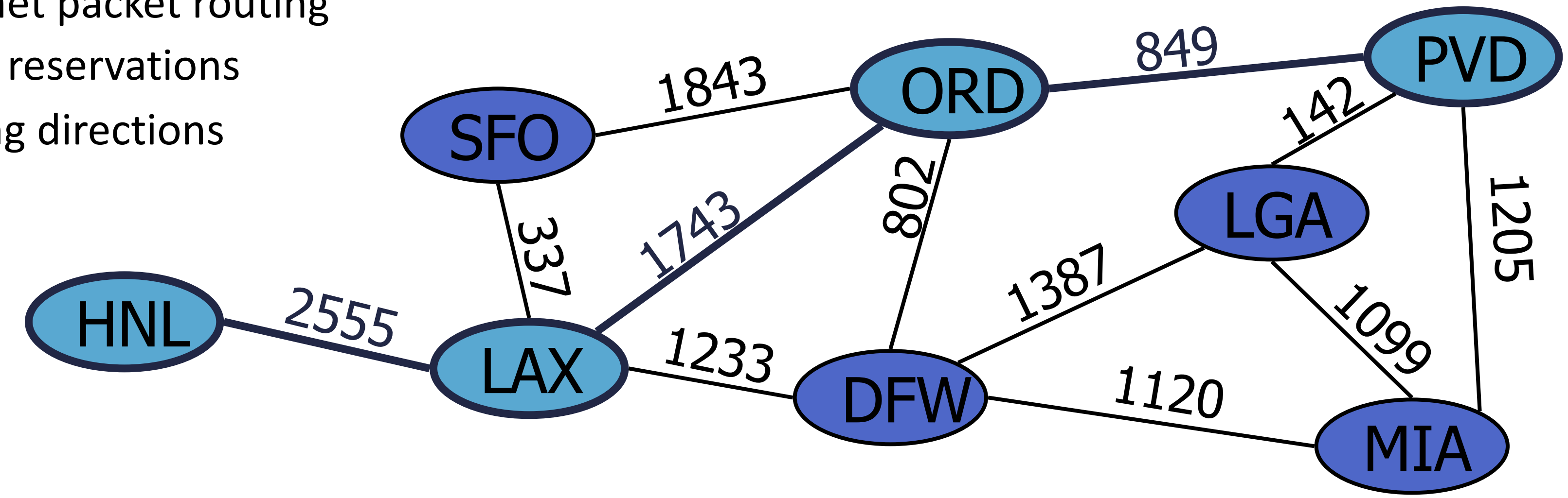
Weighted Graphs

- 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.
  - 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:

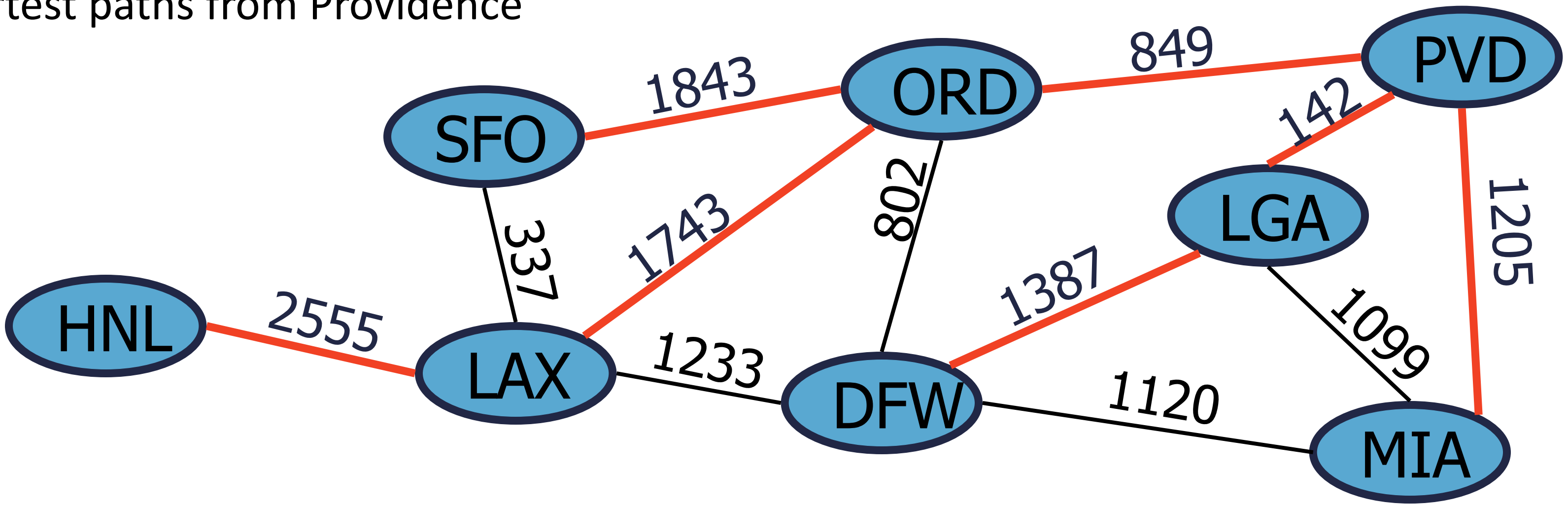
A subpath of a shortest path is itself a shortest path

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 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

Main idea

- Weighted greedy BFS

- Greedy Algorithm:
  - Take a best option at current stage
  - Work best when the optimal solution for a subproblem is also included in the optimal solution of the whole problem.
    - ***REMEMBER: Property 1) A subpath of a shortest path is itself a shortest path***

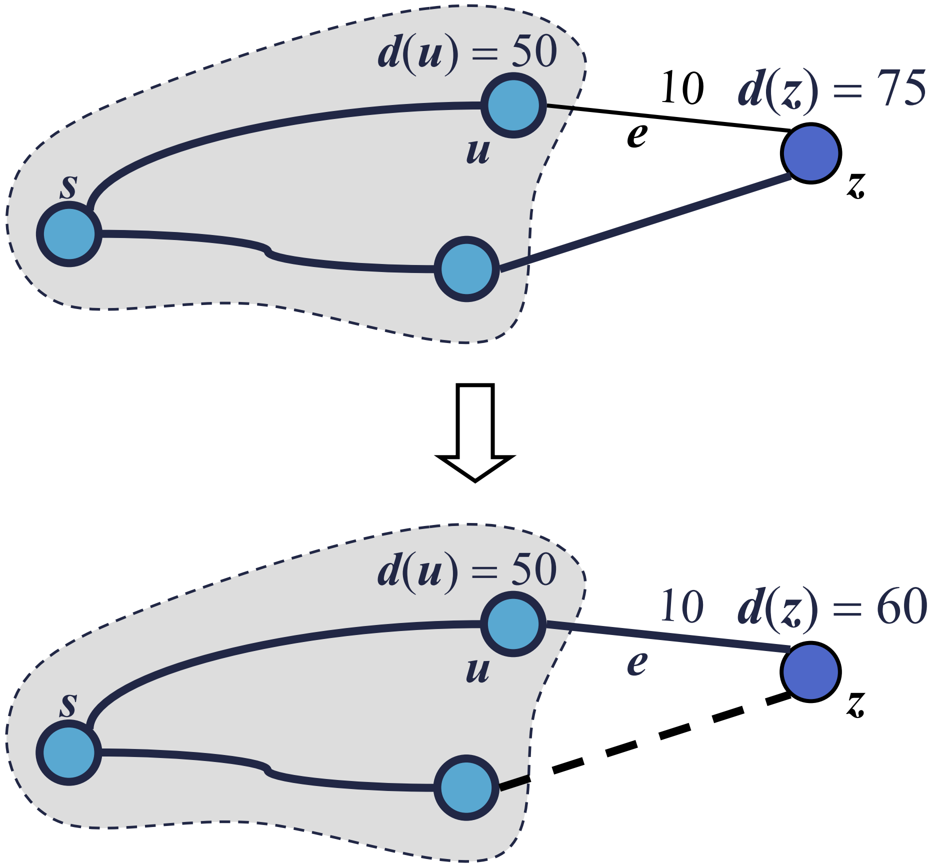
- Maintina a “cloud” of “optimal” vertices (=subproblem), and do greedy expansion at each iteration (=edge relaxation).

Edge Relaxation

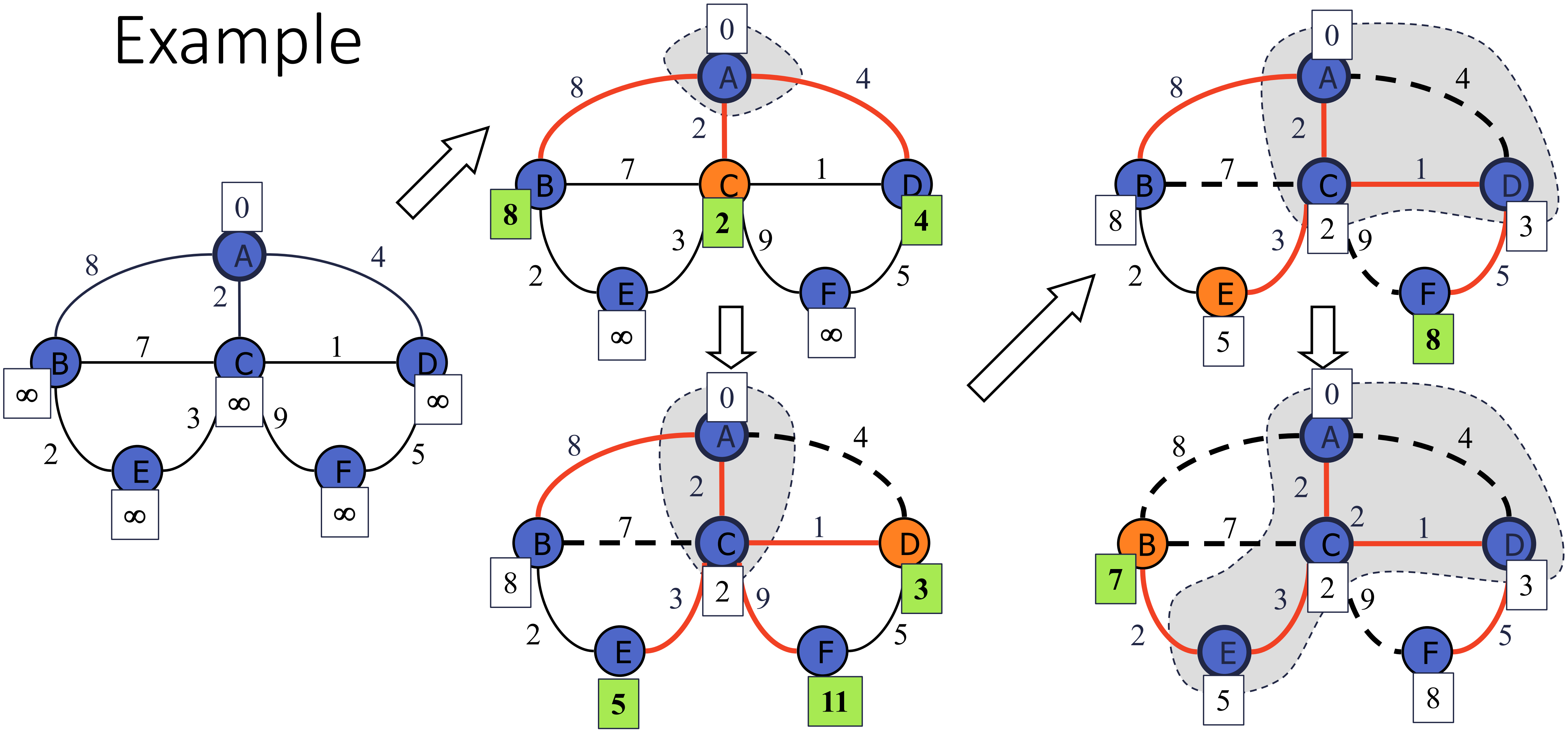
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud

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

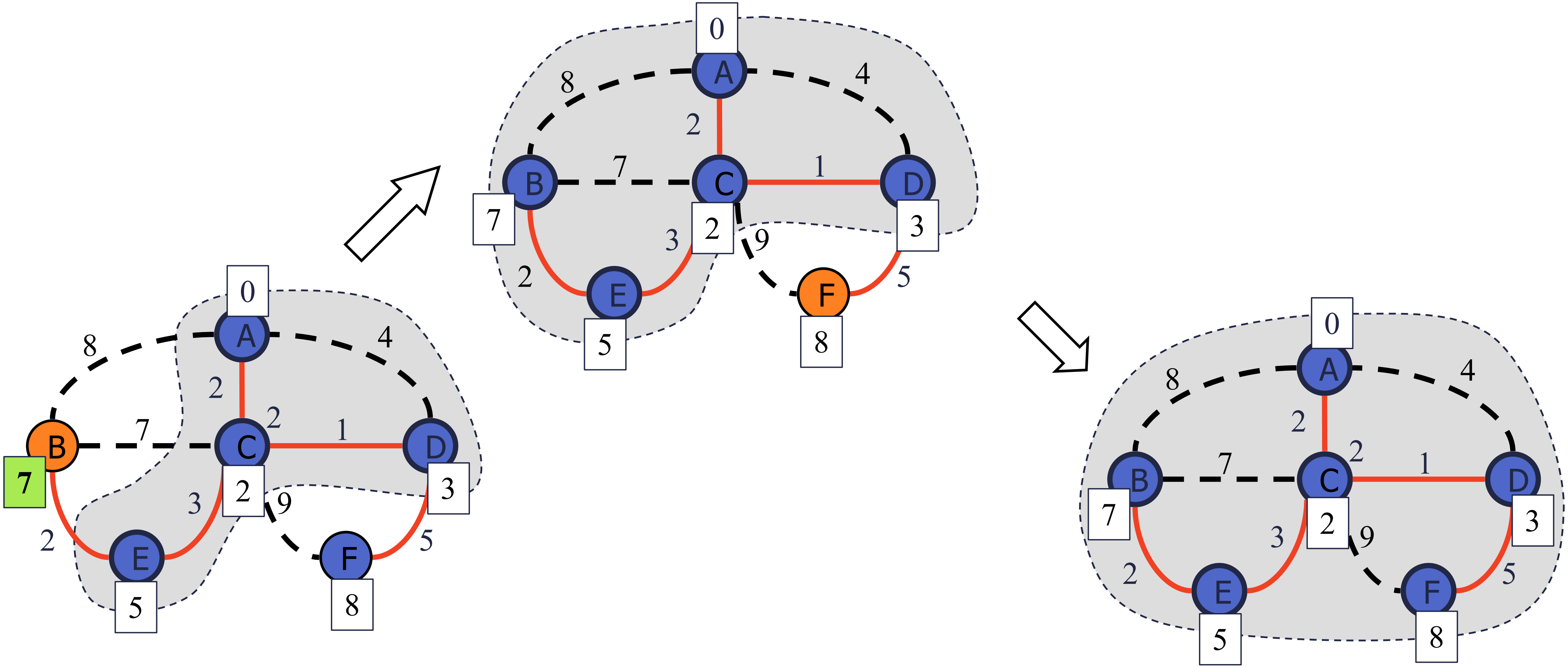
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



Example



Example (cont.)



Implmentation (using a table)

https://www.youtube.com/watch?v=pVfj6mxhdMw

Implementation (using a priority queue)

Algorithm ShortestPath(G, s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

Output: The length of a shortest path from s to v for each vertex v of G.

```
Initialize D[s] = 0 and D[v] = ∞ for each vertex v ≠ s.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty do
  {pull a new vertex u into the cloud}
  u = value returned by Q.remove_min()
  for each vertex v adjacent to u such that v is in Q  do
    {perform the relaxation procedure on edge (u,v)}
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change to D[v] the key of vertex v in Q.
return the label D[v] of each vertex v
```

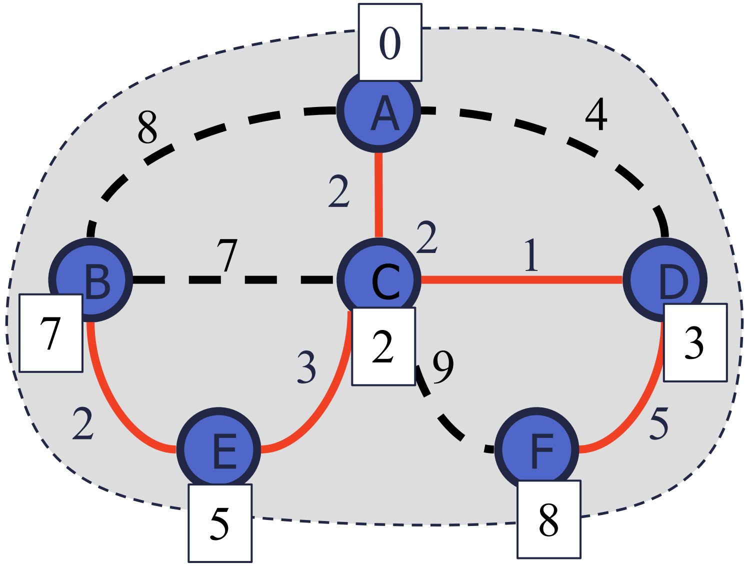
Reconstructing the shortest-path tree

Maintaining a separate table

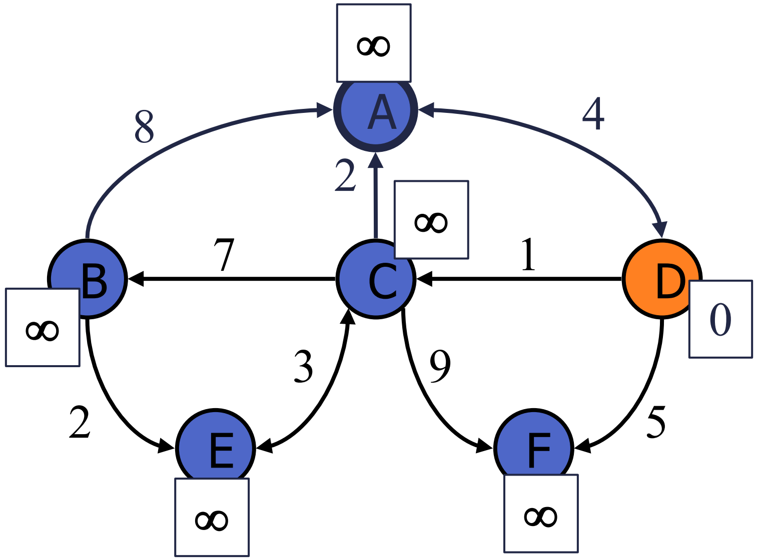
Vertex	From
A	-
B	E
C	A
D	C
E	C
F	D

Reconstring from a property:

- $d[u]+w(u,v) = d[v]$



Exercise



Analysis of Dijkstra's Algorithm

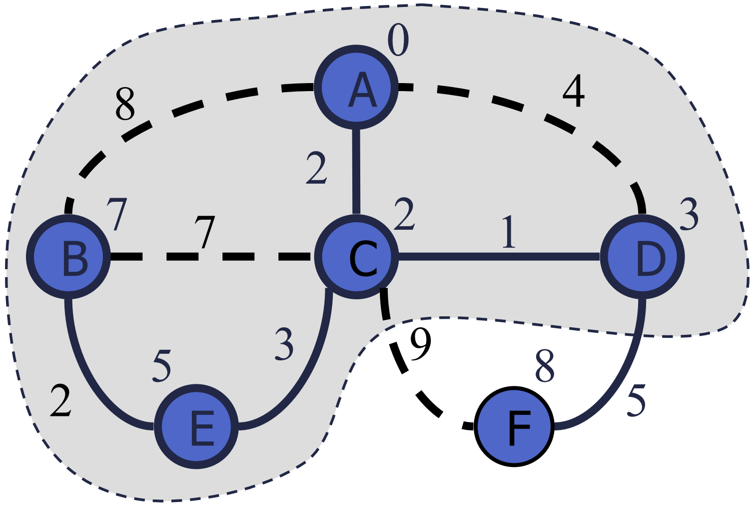
- Graph operations
  - We find all the incident edges once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once 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/map structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

Python Implementation

```
def shortest_path_lengths(g, src):
  """Compute shortest-path distances from src to reachable vertices of g.

  Graph g can be undirected or directed, but must be weighted such that
  e.element() returns a numeric weight for each edge e.

  Return dictionary mapping each reachable vertex to its distance from src.
  """
  d = { }                               # d[v] is upper bound from s to v
  cloud = { }                           # map reachable v to its d[v] value
  pq = AdaptableHeapPriorityQueue( )    # vertex v will have key d[v]
  pqlocator = { }                       # map from vertex to its pq locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices( ):
    if v is src:
      d[v] = 0
    else:
      d[v] = float('inf')               # syntax for positive infinity
    pqlocator[v] = pq.add(d[v], v)      # save locator for future updates

  while not pq.is_empty( ):
    key, u = pq.remove_min( )
    cloud[u] = key                      # its correct d[u] value
    del pqlocator[u]                    # u is no longer in pq
    for e in g.incident_edges(u):       # outgoing edges (u,v)
      v = e.opposite(u)
      if v not in cloud:
        # perform relaxation step on edge (u,v)
        wgt = e.element( )
        if d[u] + wgt < d[v]:                   # better path to v?
          d[v] = d[u] + wgt                     # update the distance
          pq.update(pqlocator[v], d[v], v)      # update the pq entry

  return cloud                          # only includes reachable vertices
```

Why Dijkstra's Algorithm Works

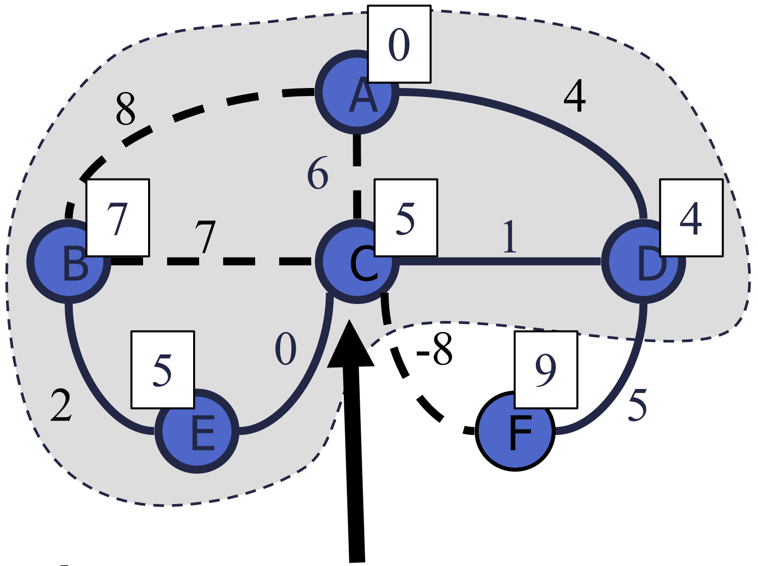
- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - 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!

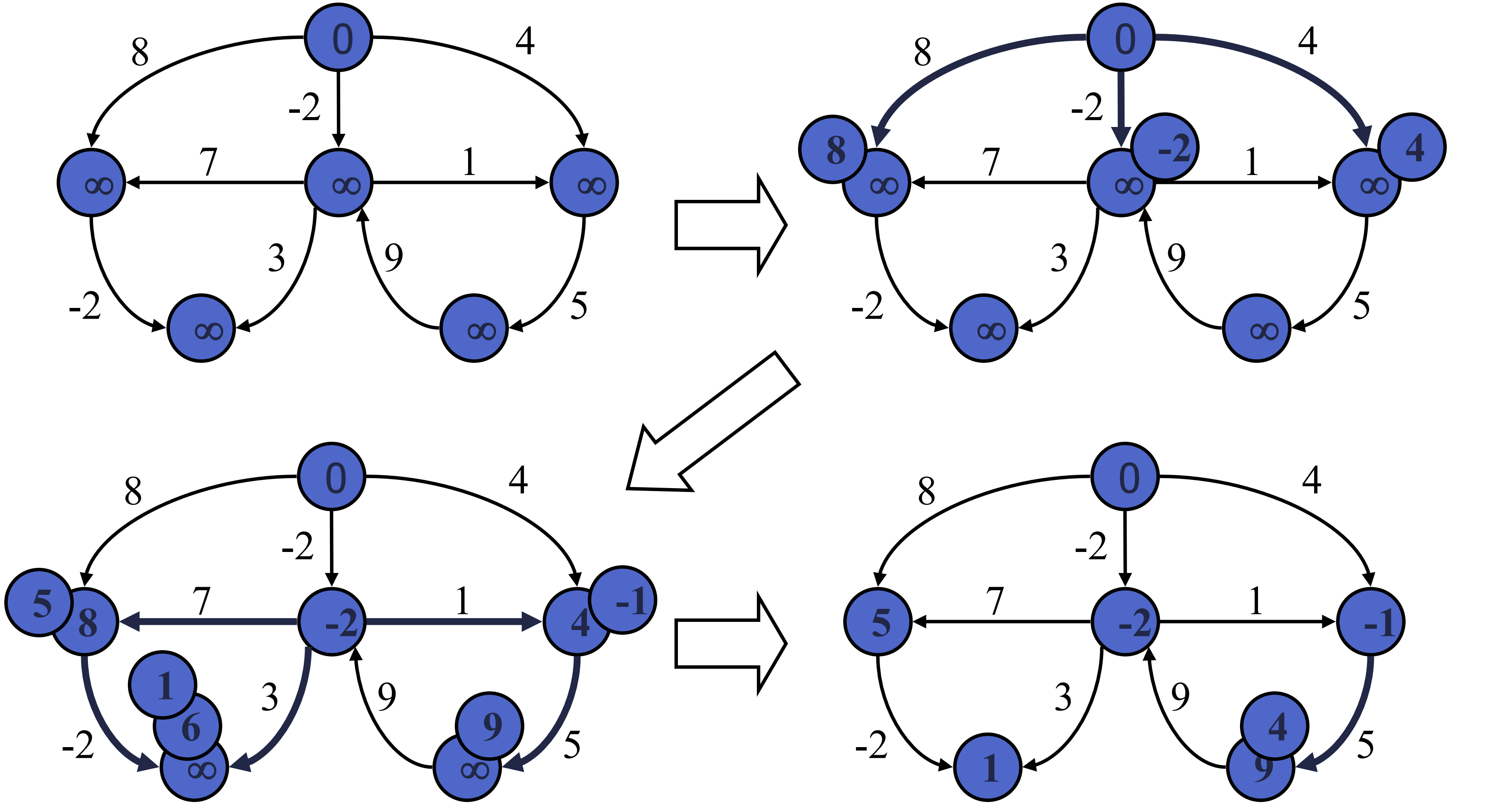
Bellman-Ford Algorithm (not in book)

- 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



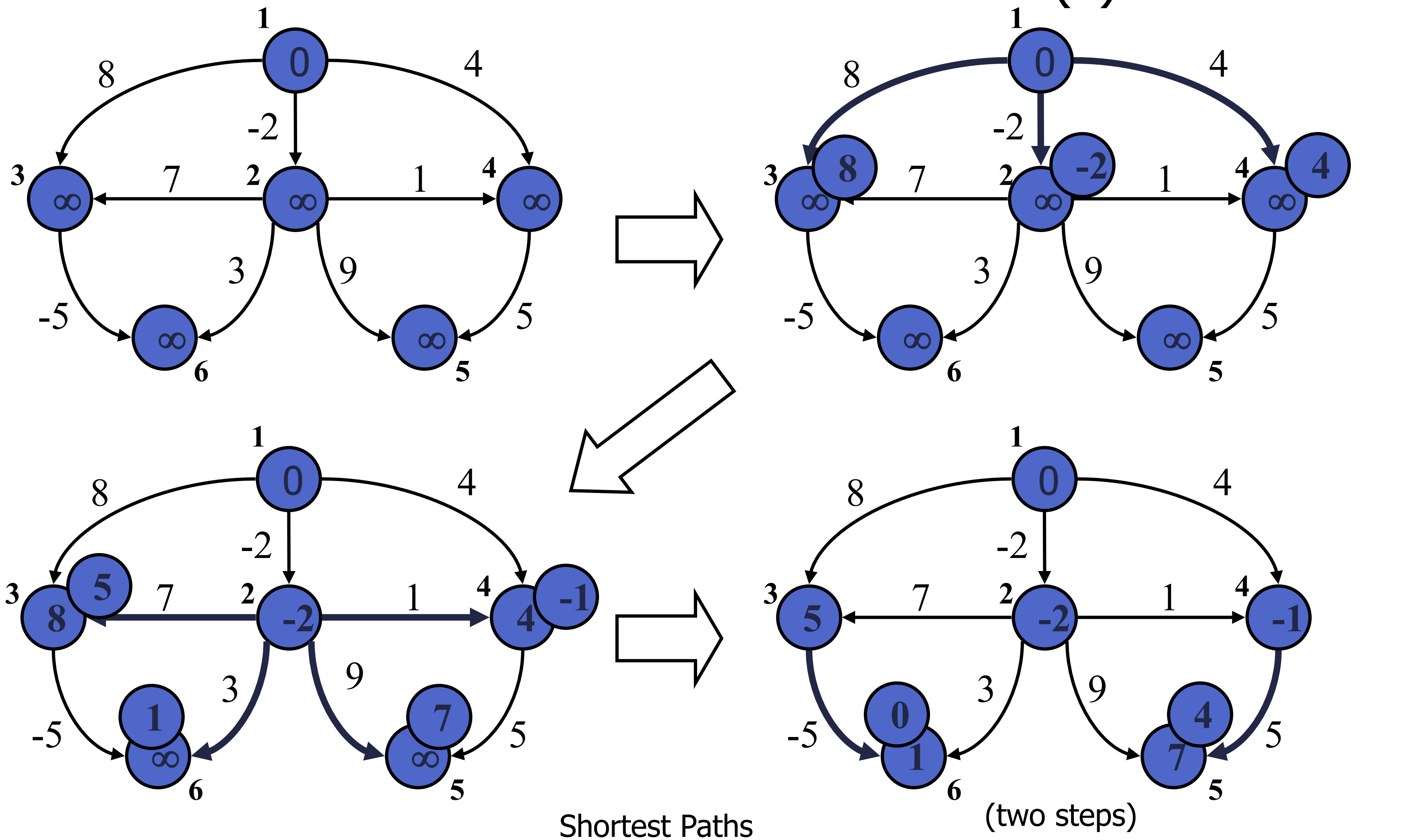
DAG-based Algorithm (not in book)

- 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).

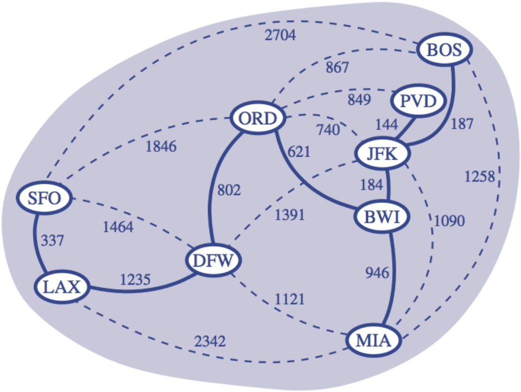
```
Algorithm DagDistances(G, s)
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    { Perform a topological sort of the vertices }
    for u ← 1 to n do    {in topological order}
        for each e ∈ G.outEdges(u)
            { relax 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



Minimum Spanning Trees



Minimum Spanning Trees

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

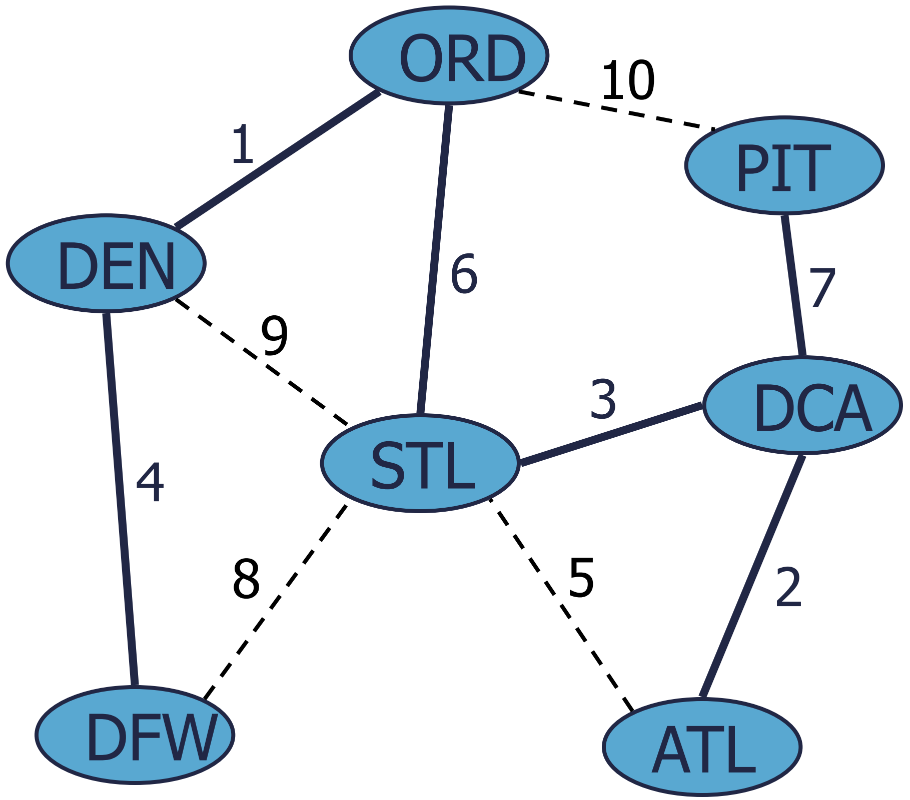
Spanning tree

- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)

- Spanning tree of a weighted graph with minimum total edge weight

- Applications
  - Communications networks
  - Transportation networks



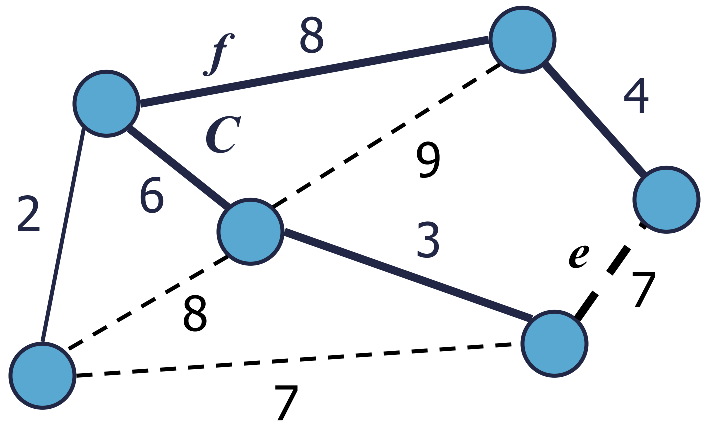
Cycle Property

Cycle Property:

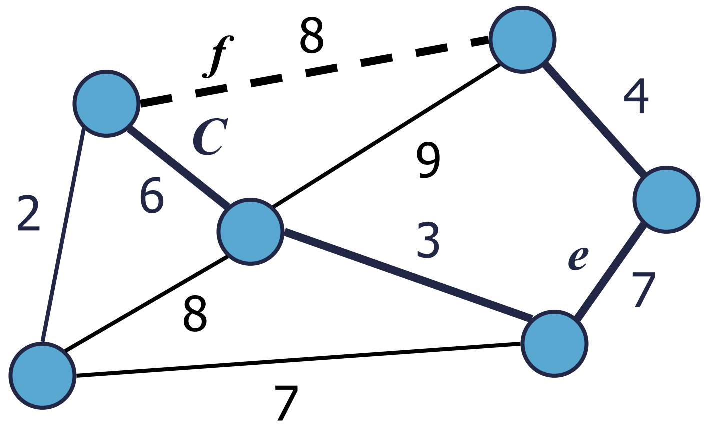
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:

- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f



Replacing f with e yields a better spanning tree



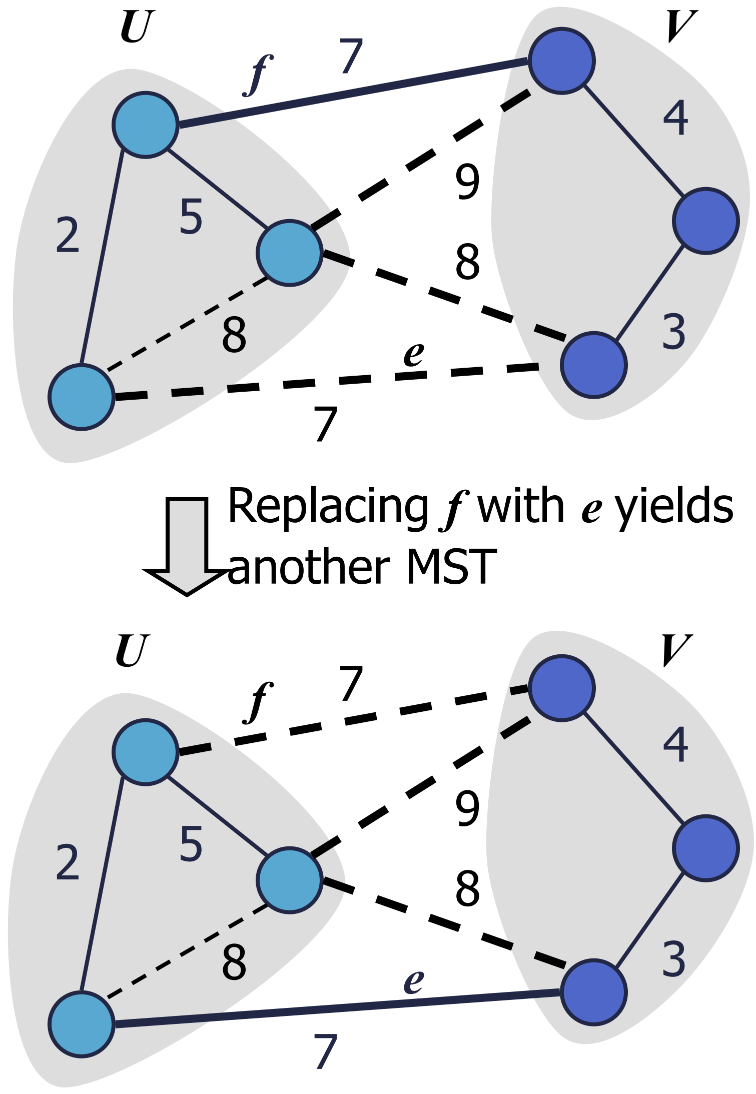
Partition Property

Partition Property:

- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

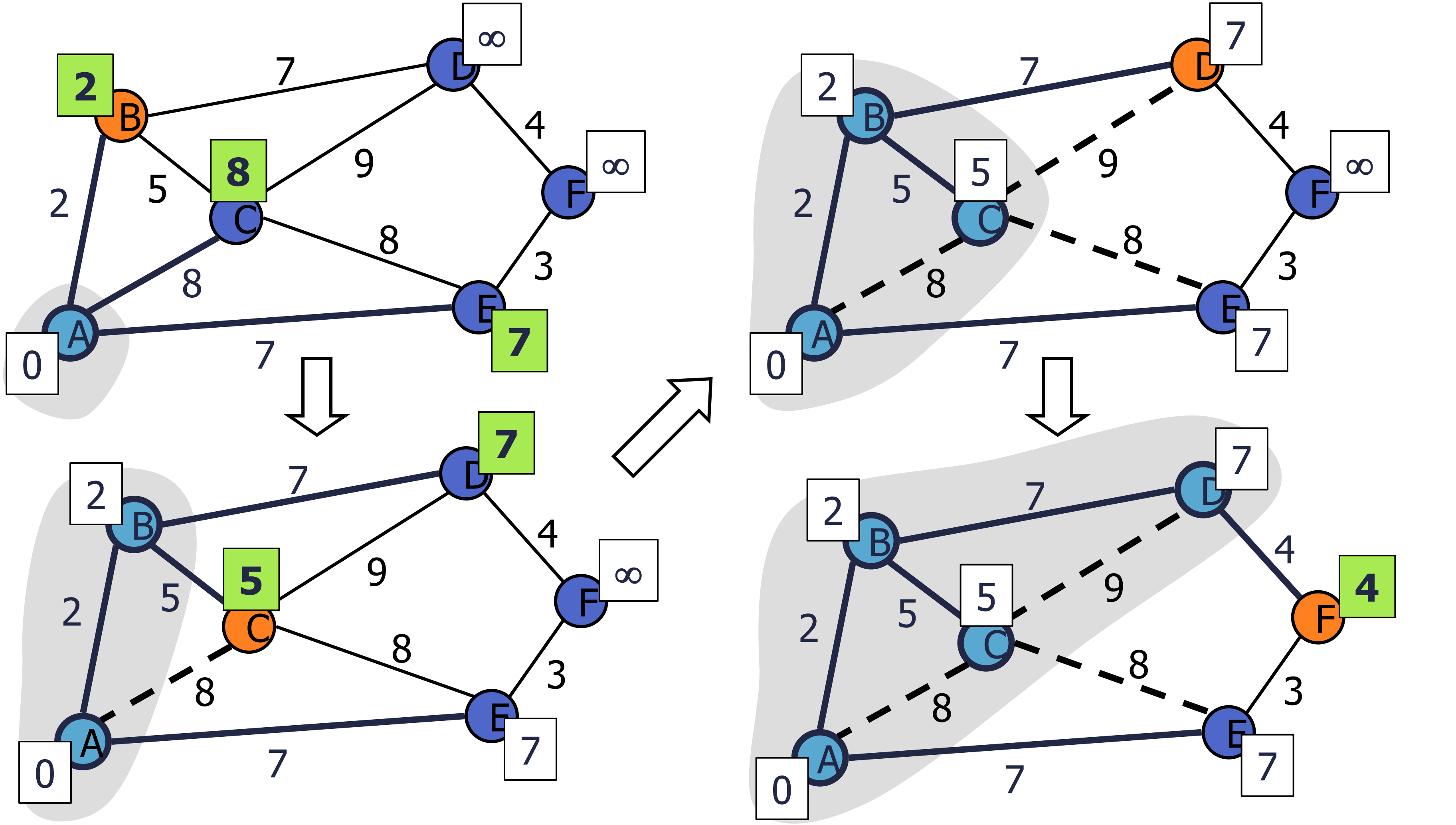
- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property,
$$weight(f) \leq weight(e)$$
- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e



Prim-Jarnik's Algorithm

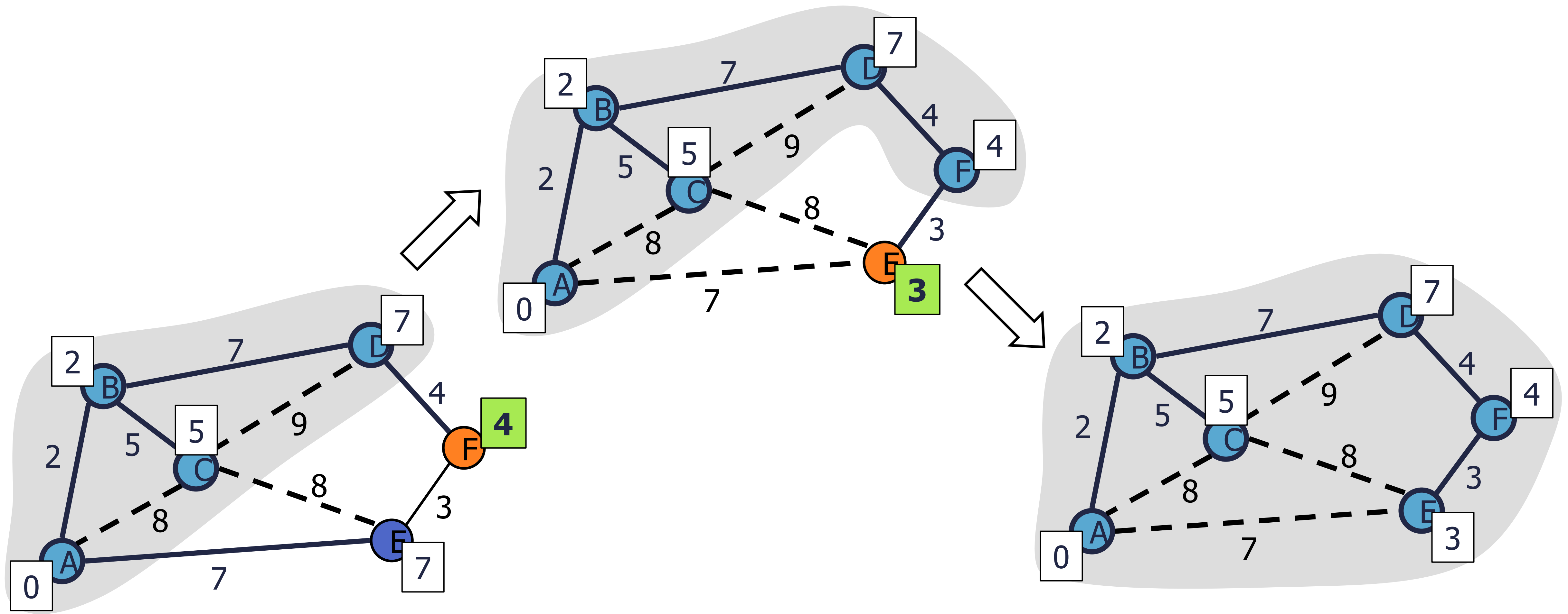
- Similar to Dijkstra's algorithm
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

Example



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

Example (contd.)



Prim-Jarnik Pseudo-code

Algorithm PrimJarnik(G):

Input: An undirected, weighted, connected graph G with n vertices and m edges

Output: A minimum spanning tree T for G

Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

Analysis

- Graph operations
  - We cycle through the incident edges once for each vertex
- Label operations
  - We set/get the distance, parent 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 w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

Python Implementation

```
def MST_PrimJarnik(g):
  """Compute a minimum spanning tree of weighted graph g.

  Return a list of edges that comprise the MST (in arbitrary order).
  """
  d = { }                              # d[v] is bound on distance to tree
  tree = [ ]                           # list of edges in spanning tree
  pq = AdaptableHeapPriorityQueue( )   # d[v] maps to value (v, e=(u,v))
  pqlocator = { }                      # map from vertex to its pq locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices( ):
    if len(d) == 0:                          # this is the first node
      d[v] = 0                               # make it the root
    else:
      d[v] = float('inf')                    # positive infinity
    pqlocator[v] = pq.add(d[v], (v,None))

  while not pq.is_empty( ):
    key,value = pq.remove_min( )
    u,edge = value                           # unpack tuple from pq
    del pqlocator[u]                         # u is no longer in pq
    if edge is not None:
      tree.append(edge)                      # add edge to tree
    for link in g.incident_edges(u):
      v = link.opposite(u)
      if v in pqlocator:                     # thus v not yet in tree
        # see if edge (u,v) better connects v to the growing tree
        wgt = link.element( )
        if wgt < d[v]:                       # better edge to v?
          d[v] = wgt                         # update the distance
          pq.update(pqlocator[v], d[v], (v, link))   # update the pq entry
  return tree
```

Kruskal’s Approach

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge “closest” clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster and one MST

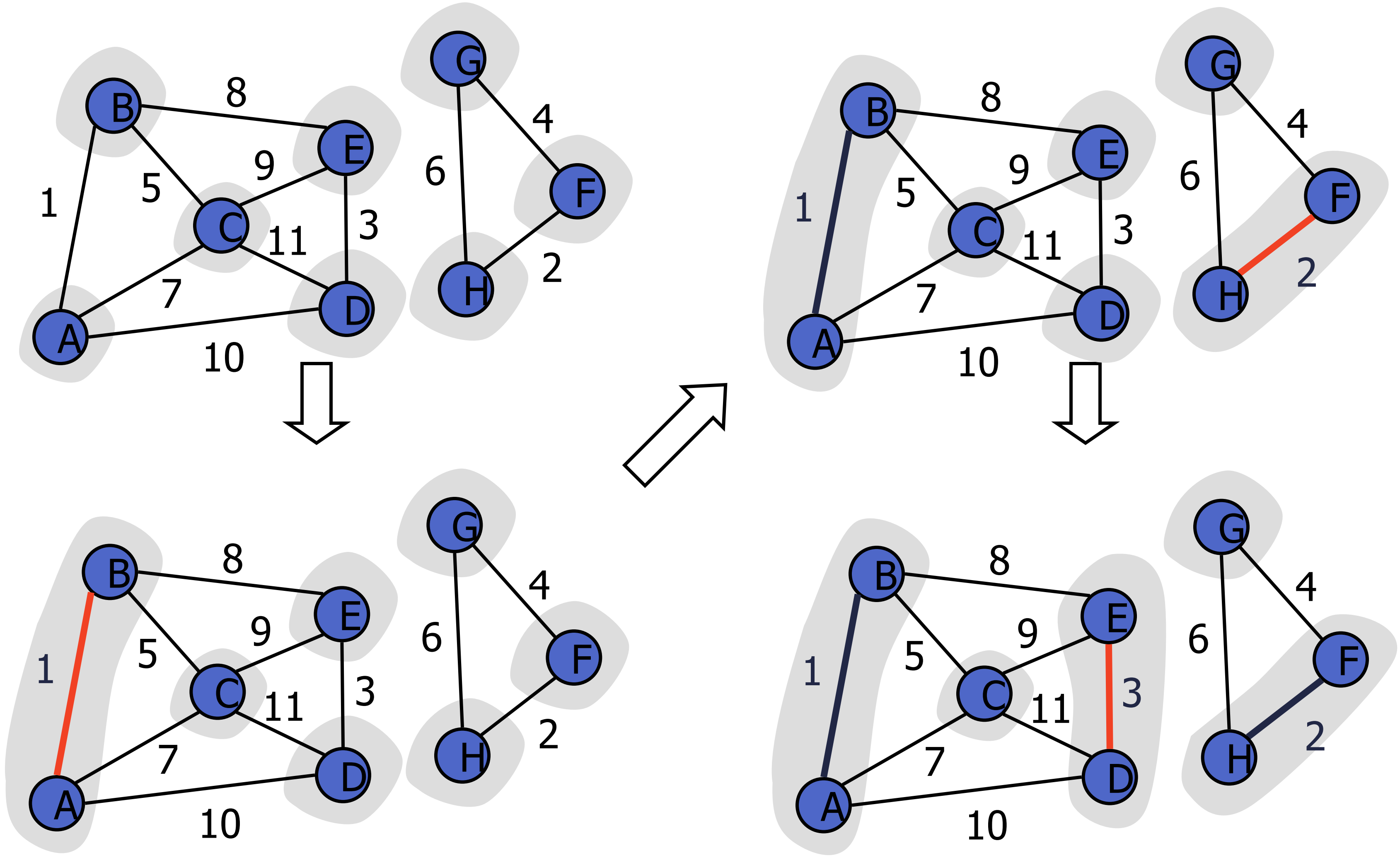
Kruskal's Algorithm

Algorithm Kruskal(G):

Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G

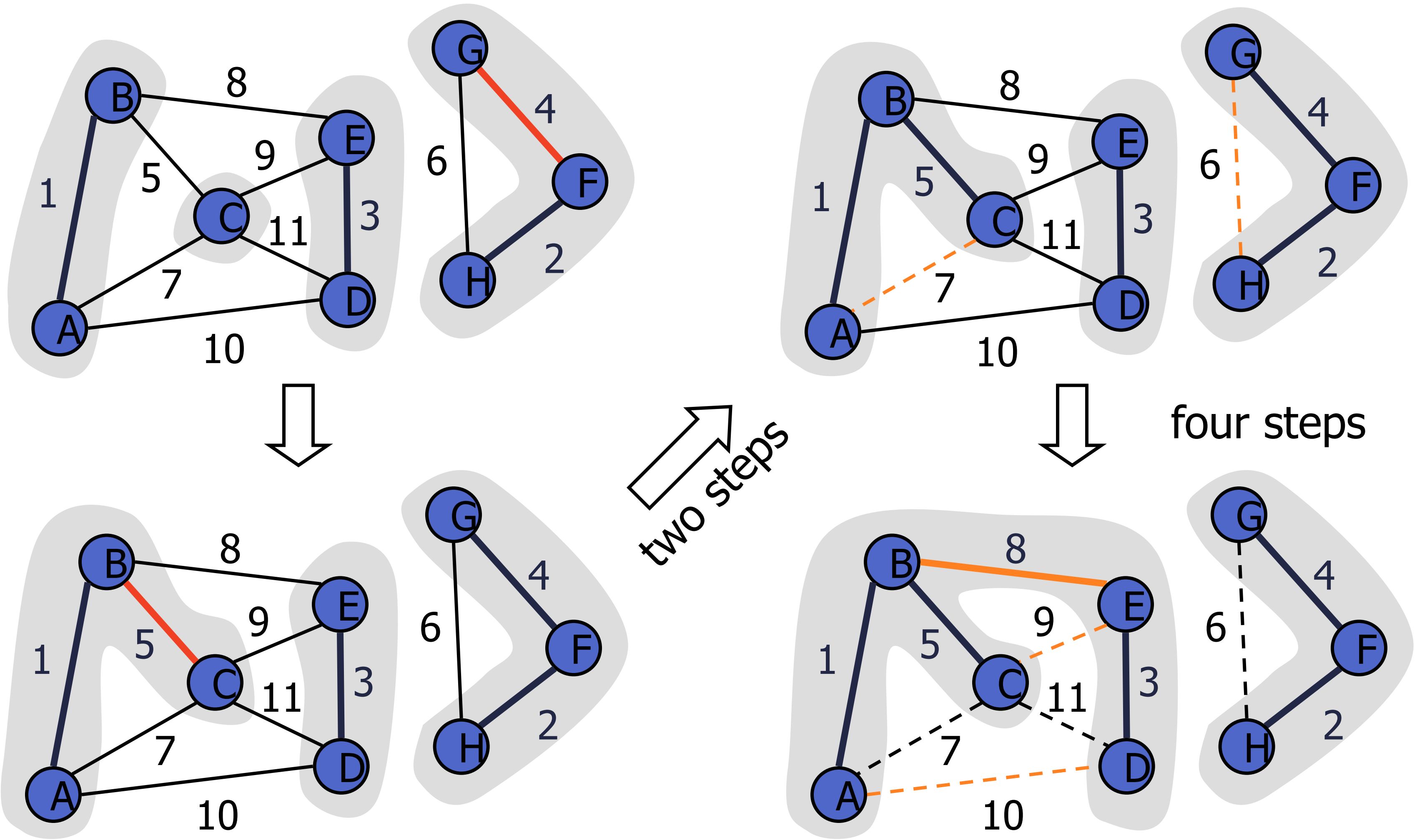
for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n-1$ edges **do**
  (u,v) = value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u,v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

Example



Campus Tour

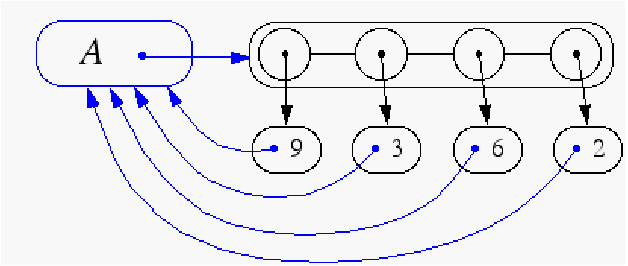
Example (contd.)



Data Structure for Kruskal’s Algorithm

- The algorithm maintains a forest of trees
- A priority queue extracts the edges by increasing weight
- An edge is accepted it if connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with operations:
  - makeSet(u): create a set consisting of u
  - find(u): return the set storing u
  - union(A, B): replace sets A and B with their union

List-based Partition



- Each set is stored in a sequence
- Each element has a reference back to the set
  - operation find(u) takes O(1) time, and returns the set of which u is a member.
  - in operation union(A,B), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation union(A,B) is min(|A|, |B|)
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most log n times

Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
  - Cluster merges as unions
  - Cluster locations as finds
- Running time $O((n + m) \log n)$
  - Priority Queue operations: $O(m \log n)$
  - Union-Find operations: $O(n \log n)$

Python Implementation

```
def MST_Kruskal(g):
  """Compute a minimum spanning tree of a graph using Kruskal's algorithm.

  Return a list of edges that comprise the MST.

  The elements of the graph's edges are assumed to be weights.
  """
  tree = [ ]                      # list of edges in spanning tree
  pq = HeapPriorityQueue( )       # entries are edges in G, with weights as key
  forest = Partition( )           # keeps track of forest clusters
  position = { }                  # map each node to its Partition entry

  for v in g.vertices( ):
    position[v] = forest.make_group(v)

  for e in g.edges( ):
    pq.add(e.element( ), e)       # edge's element is assumed to be its weight

  size = g.vertex_count( )
  while len(tree) != size − 1 and not pq.is_empty( ):
    # tree not spanning and unprocessed edges remain
    weight,edge = pq.remove_min( )
    u,v = edge.endpoints( )
    a = forest.find(position[u])
    b = forest.find(position[v])
    if a != b:
      tree.append(edge)
      forest.union(a,b)

  return tree
```

Baruvka's Algorithm (Exercise)

- Like Kruskal's Algorithm, Baruvka's algorithm grows many clusters at once and maintains a forest T
- Each iteration of the while loop halves the number of connected components in forest T
- The running time is $O(m \log n)$

```
Algorithm BaruvkaMST(G)
    T ← V  {just the vertices of G}
  while T has fewer than n − 1 edges do
    for each connected component C in T do
      Let edge e be the smallest-weight edge from C to another component in T
      if e is not already in T then
        Add edge e to T
  return T
```