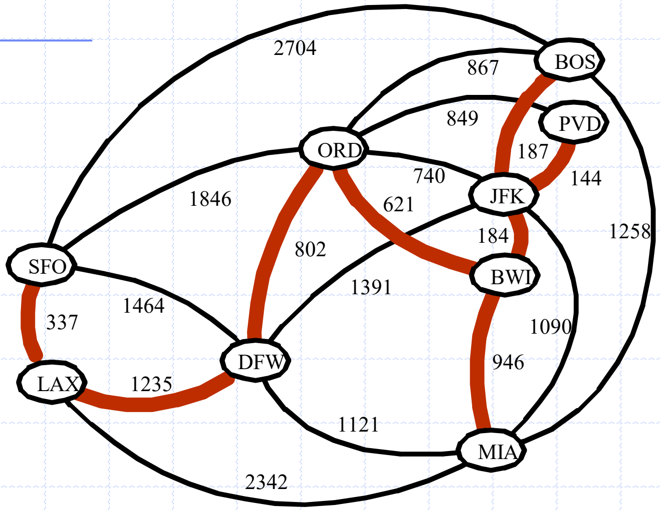
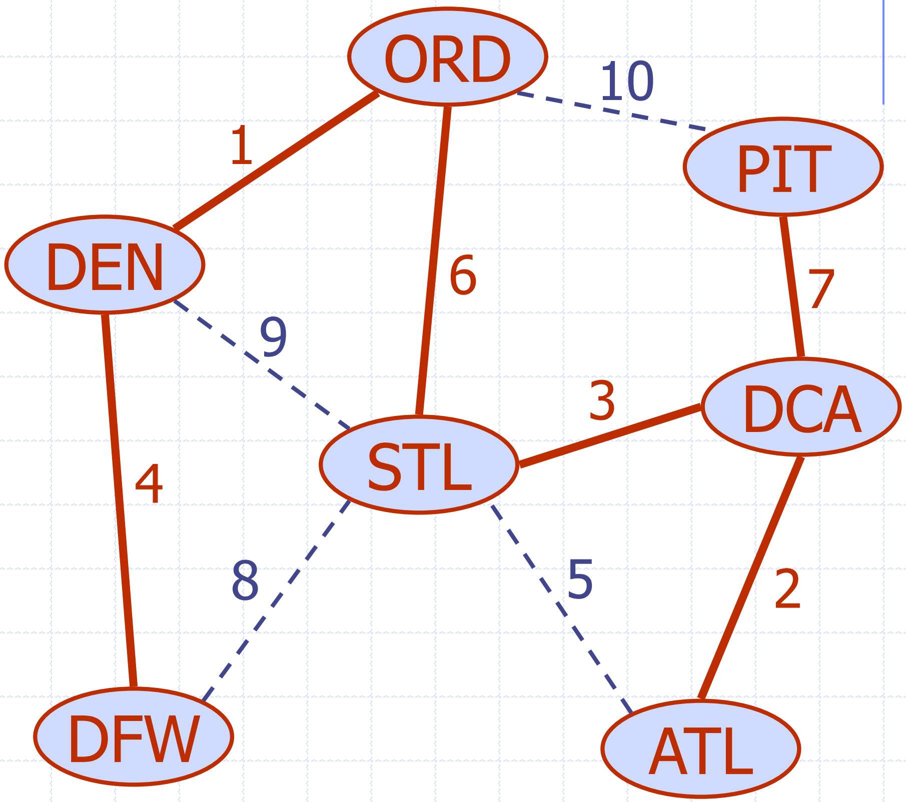
Minimum Spanning Trees



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



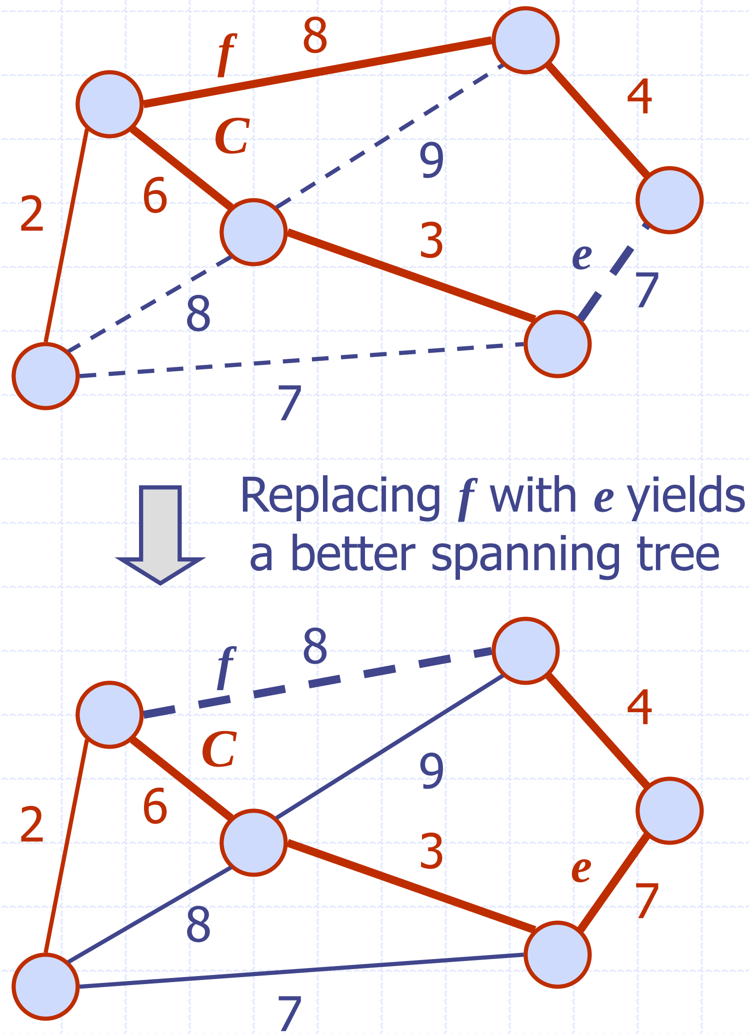
Cycle Property

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 let C 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 f with e



Partition Property

Partition Property:

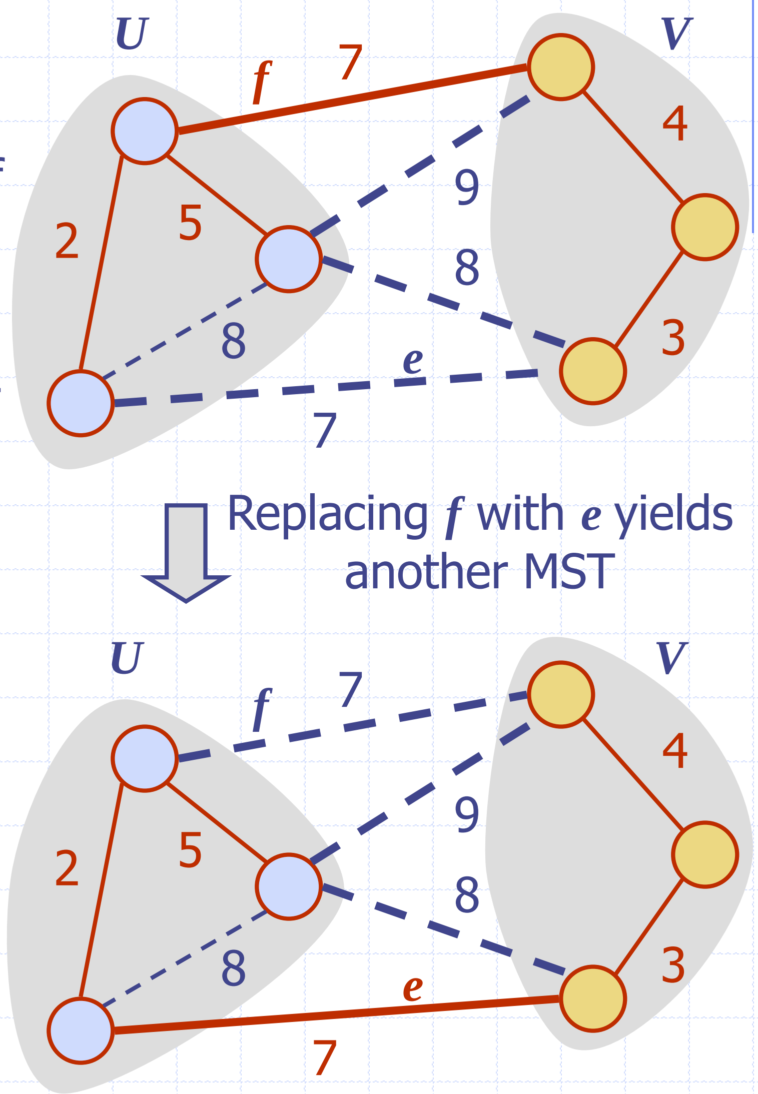
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

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

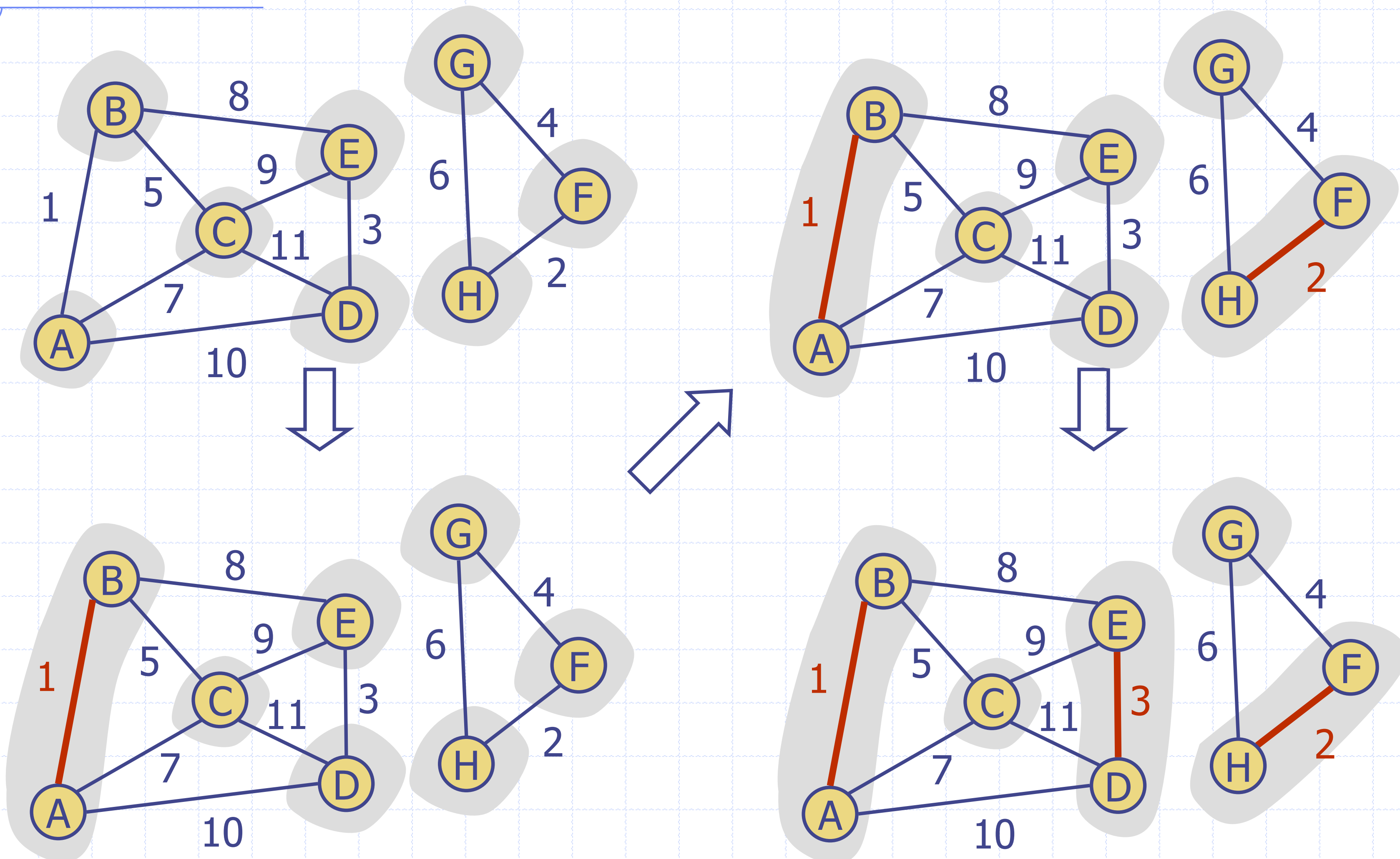


Kruskal's Algorithm

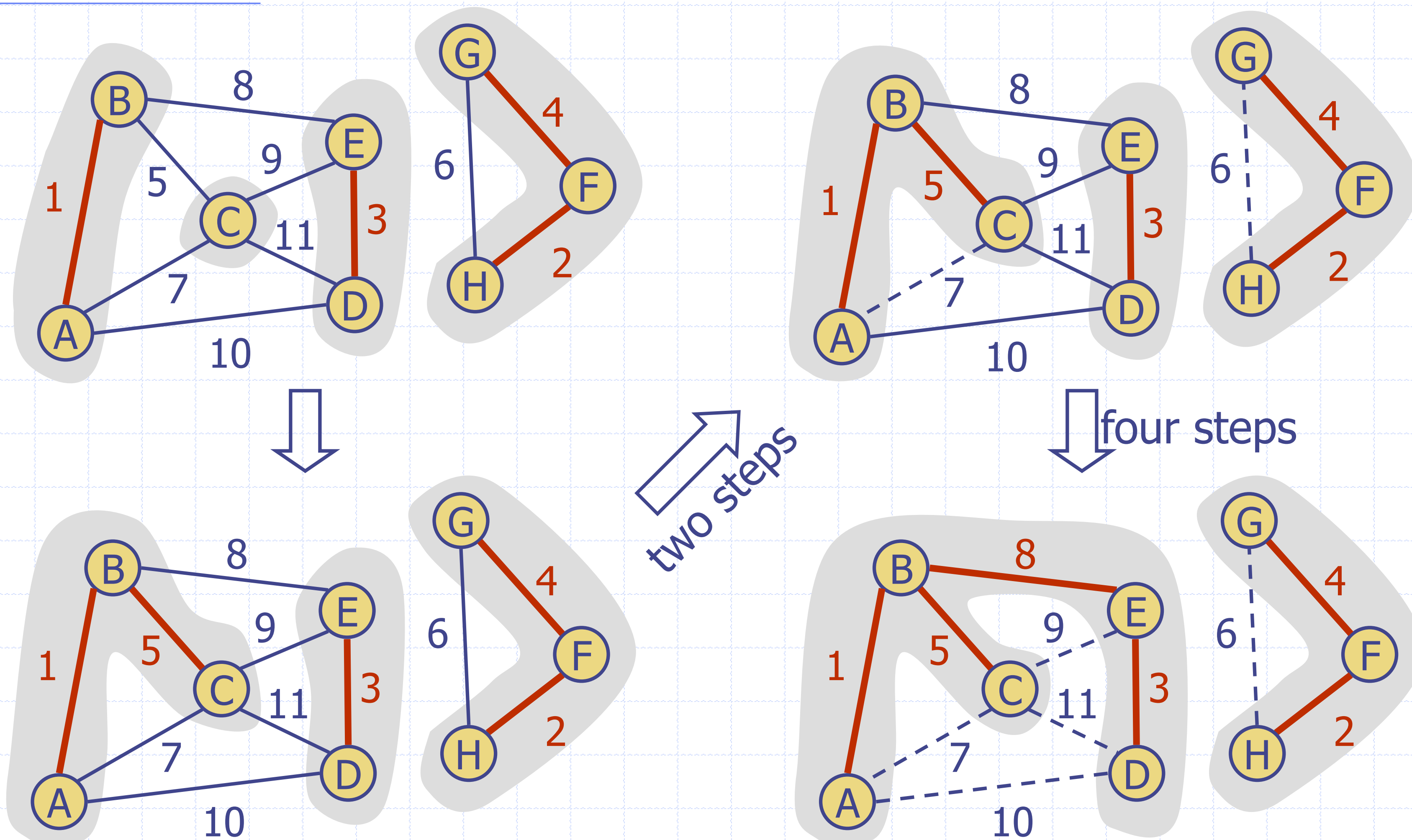
- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge "closest" clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster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

```
Algorithm KruskalMST(G)
    for each vertex v in G do
        Create a cluster consisting of v
    let Q be a priority queue.
    Insert all edges into Q
    T ← ∅
    {T is the union of the MSTs of the clusters}
    while T has fewer than n – 1 edges do
        e ← Q.removeMin().getValue()
        [u, v] ← G.endVertices(e)
        A ← getCluster(u)
        B ← getCluster(v)
        if A ≠ B then
            Add edge e to T
            mergeClusters(A, B)
    return T
```

Example



Example (contd.)



Data Structures for Kruskal's Algorithm

- The graph will be implemented using adjacency lists.
- The algorithm maintains a forest of trees.
- A priority queue extracts the edges by increasing weight. The priority queue is implemented as a min heap.
- 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

Recall of Data Structures for Disjoint Sets

- Each set may be stored as a linked list represented by its first member.
- Each list element (set member) has a reference to the set representative.
- 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 end of the list of the larger set and update their references to the set representative.
  - 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

```
Algorithm KruskalMST(G)
    Initialize a partition P
    for each vertex v in G do
        P.makeSet(v)
    let Q be a priority queue.
    Insert all edges into Q
    T ← ∅
    {T is the union of the MSTs of the clusters}
    while T has fewer than n – 1 edges do
    e ← Q.removeMin().getValue()
       [u, v] ← G.endVertices(e)
       A ← P.find(u)
       B ← P.find(v)
       if A ≠ B then
           Add edge e to T
           P.union(A, B)
    return T
```

Complexity Analysis

- Let n and m denote the number of vertices and edges of the input graph respectively
- PQ operations $O(m \log m) = O(m \log n)$
- UF operations $O(n \log n)$
- Therefore, the running time of the algorithm is $O((n + m) \log n)$

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

Prim-Jarnik's Algorithm (cont.)

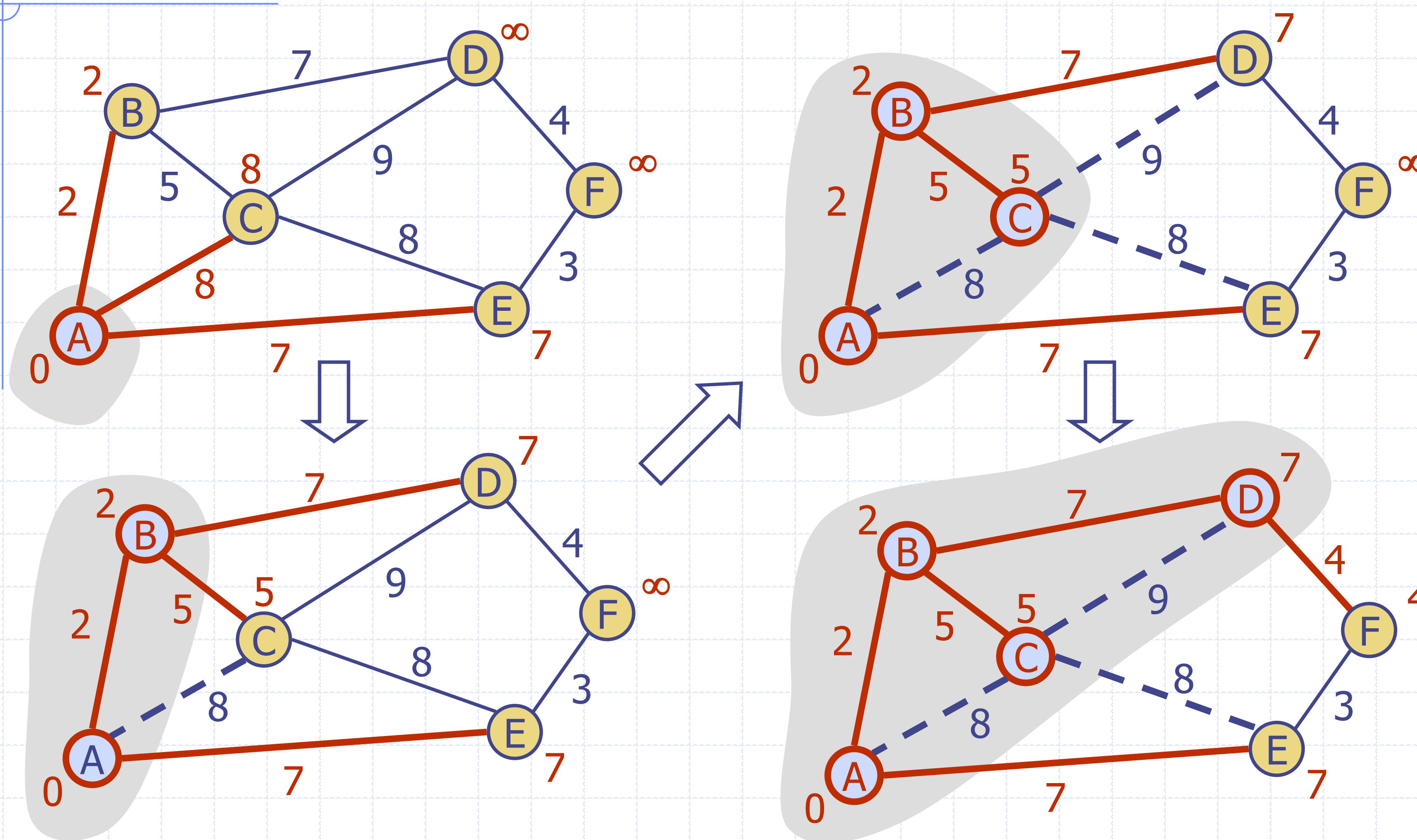
- We will use adjacency lists for the representation of the input graph.
- We will use a priority queue to store, for each vertex v, the pair (v,e) with key $D(v)$ where e is the edge with the smallest weight connecting v to the cloud and $D(v)$ is that weight.
- The priority queue will be implemented as a min heap.

Prim-Jarnik's Algorithm (cont.)

Algorithm *PrimJarnikMST*(*G*)

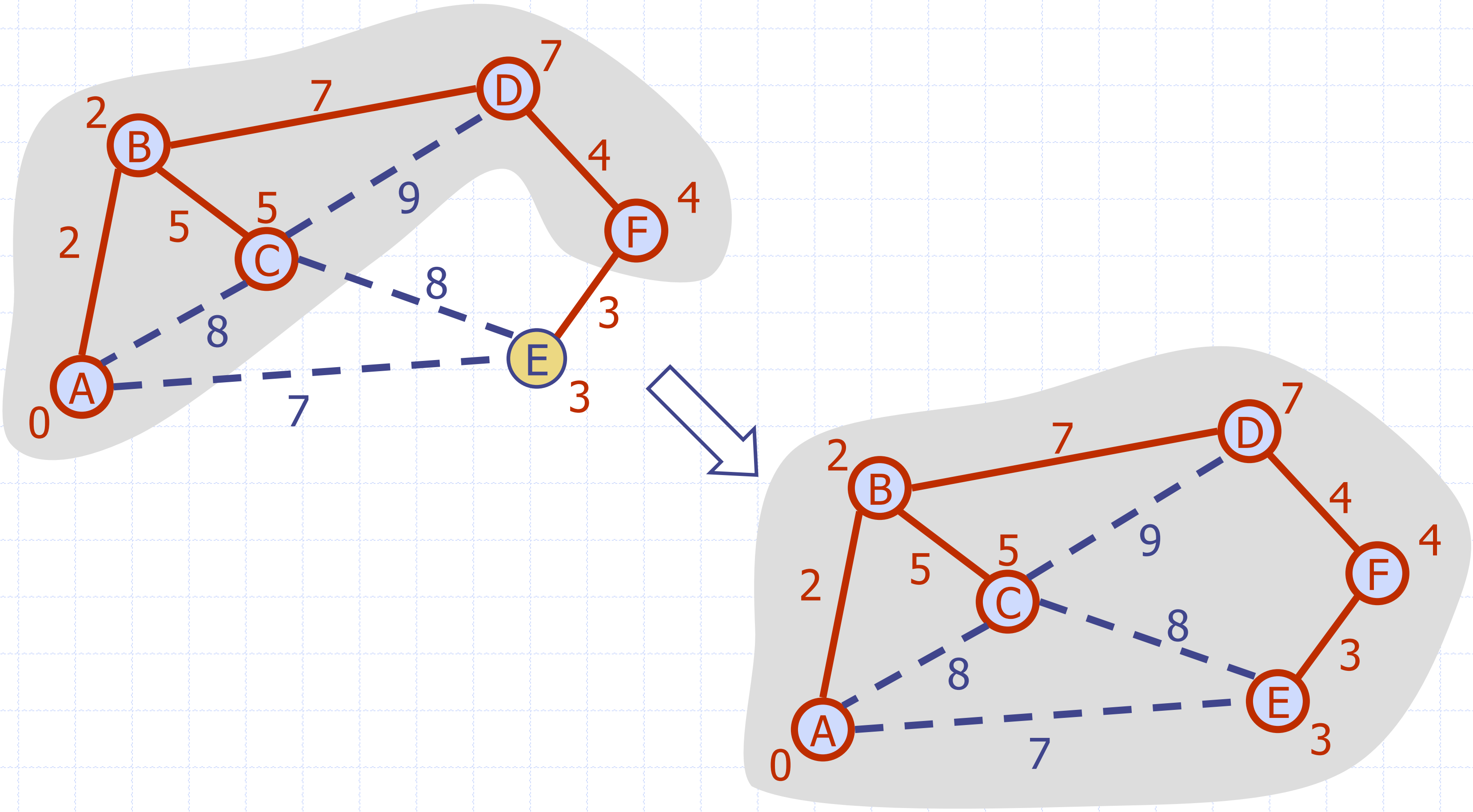
```
Pick any vertex v of G
D[v] ← 0
for each vertex u ≠ v do
    D[u] ← +∞
Initialize T ← ∅.
Initialize a priority queue Q with an entry ((u,null),D[u]) for each vertex u,
where (u,null) is the element and D[u] is the key.
while Q is not empty do
    (u,e) ← Q.removeMin()
    Add vertex u and edge e to T.
    for each vertex z adjacent to u such that z is in Q do
        if weight((u,z)) < D[z] then
            D[z] ← weight((u,z))
            Change to (z,(u,z)) the element of vertex z in Q
            Change to D[z] the key of vertex z in Q
Return the tree T
```

Example



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

Example (contd.)



Complexity Analysis

- Let n and m denote the number of vertices and edges of the input graph respectively.
- Since the priority queue is implemented as a heap, we can extract the vertex u in $O(\log n)$ time.
- We can update each $D[z]$ value in $O(\log n)$ time as well (how can we augment the priority queue implementation to achieve this bound?). This update is done at most once for each edge (u,z).
- Hence, Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time.

Readings

- M. T. Goodrich, R. Tamassia and D. Mount. Data Structures and Algorithms in C++. 2nd edition. John Wiley. 2011.
  - Chapter 13