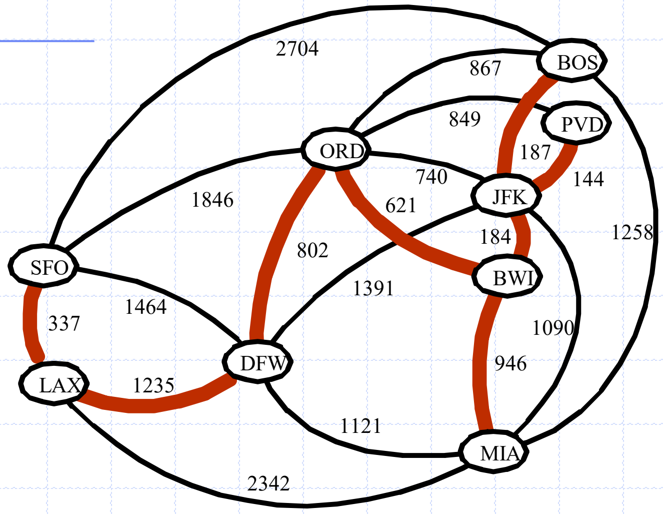
Minimum Spanning Trees



Minimum Spanning Trees

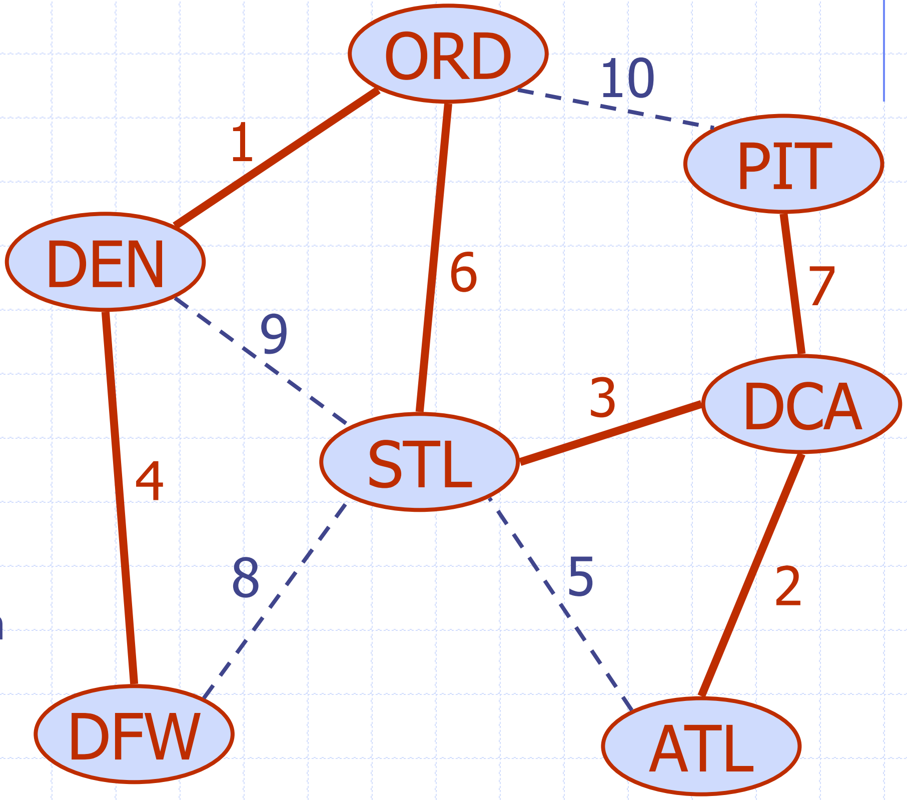
We will consider undirected weighted graphs.

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 (ελάχιστο επικαλύπτον δένδρο, MST)
- Spanning tree of a weighted graph with minimum total edge weight.

- Applications
  - Communications networks
  - Transportation networks



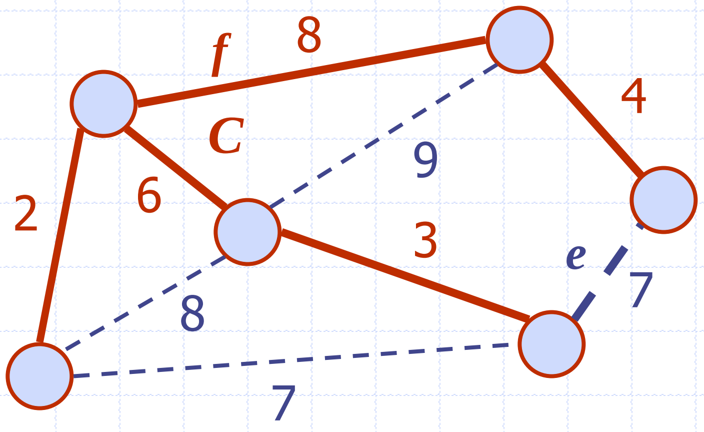
Cycle Property

Cycle Property:

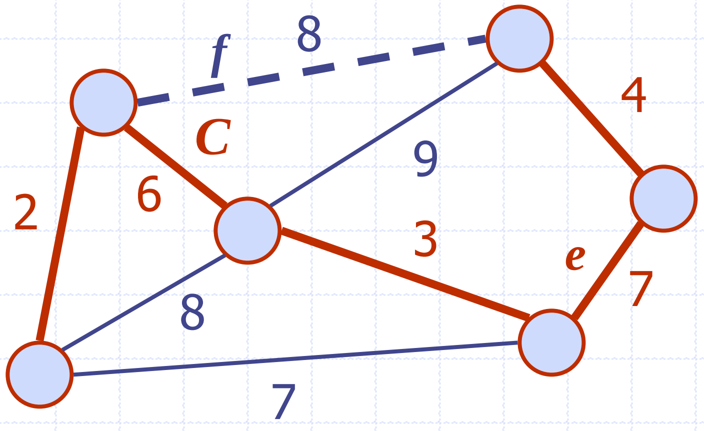
- Let T be a minimum spanning tree of a weighted graph G.
- Let e be an edge of G that is not in T and let C be the cycle formed by e with T.
- For every edge f of C, $weight(f) \leq weight(e)$.

Proof:

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



Replacing f with e yields a better spanning tree



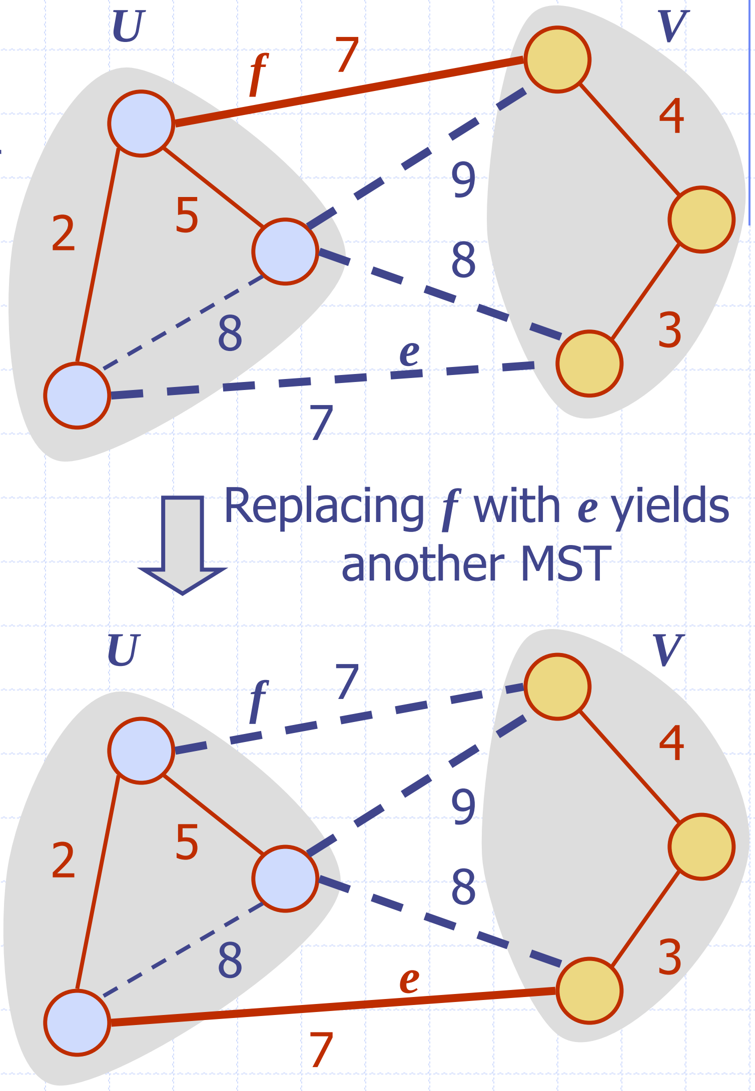
Partition Property

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.



Kruskal's Algorithm

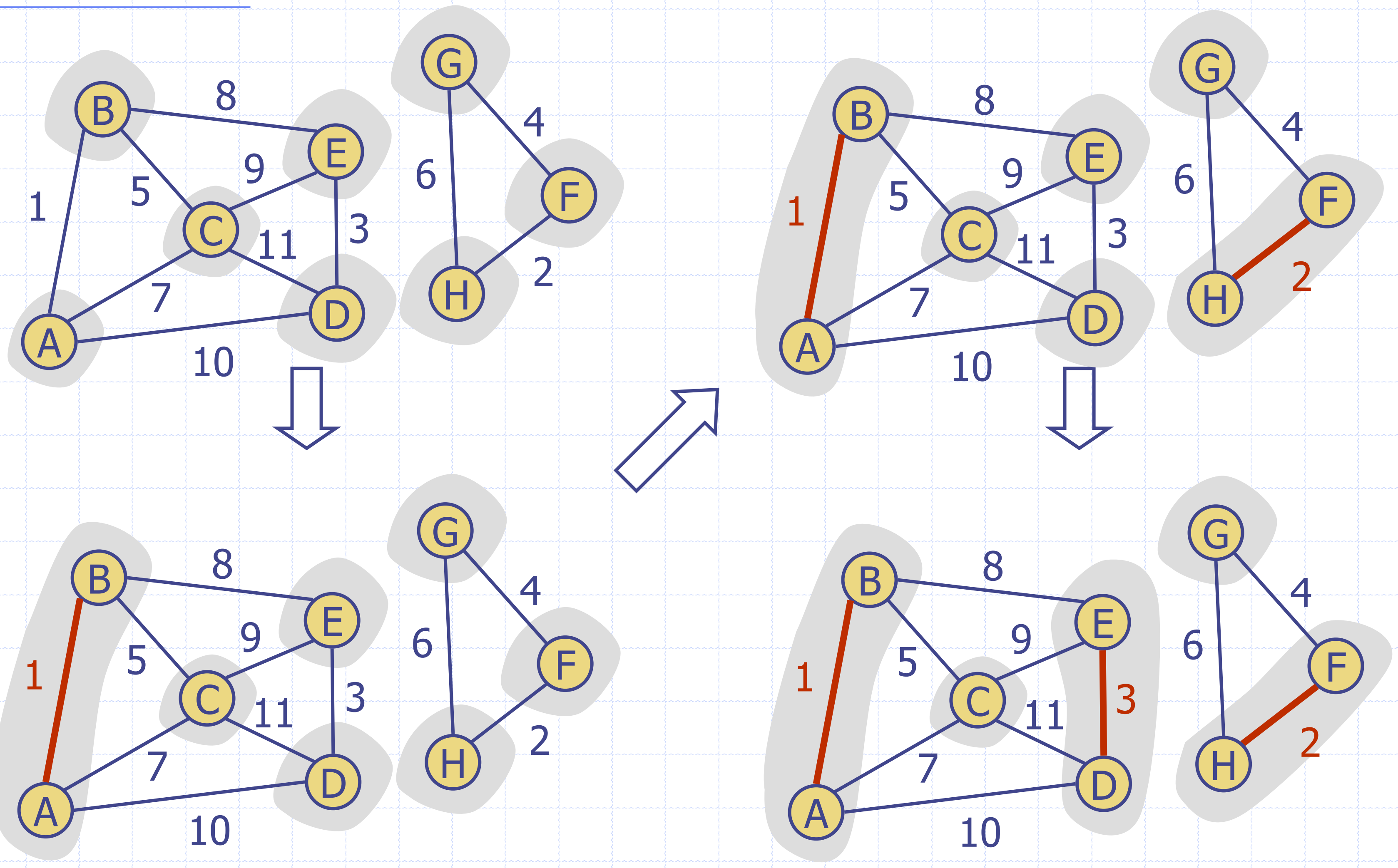
- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters.
  - Keep an MST for each cluster.
  - Merge "closest" clusters and their MSTs.
- A priority queue stores the edges outside clusters
  - Key: weight
  - Value: edge
- At the end of the algorithm
  - One cluster 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
    $T \leftarrow \varnothing$
    {T is the union of the MSTs of the clusters}
    while T has fewer than $n - 1$ edges do
        $e \leftarrow$ Q.removeMin().getValue()
        $[u, v] \leftarrow$ G.endVertices(e)
        $A \leftarrow$ getCluster(u)
        $B \leftarrow$ getCluster(v)
        if $A \neq B$ then
            Add edge e to T
            mergeClusters(A, B)
    return T
```

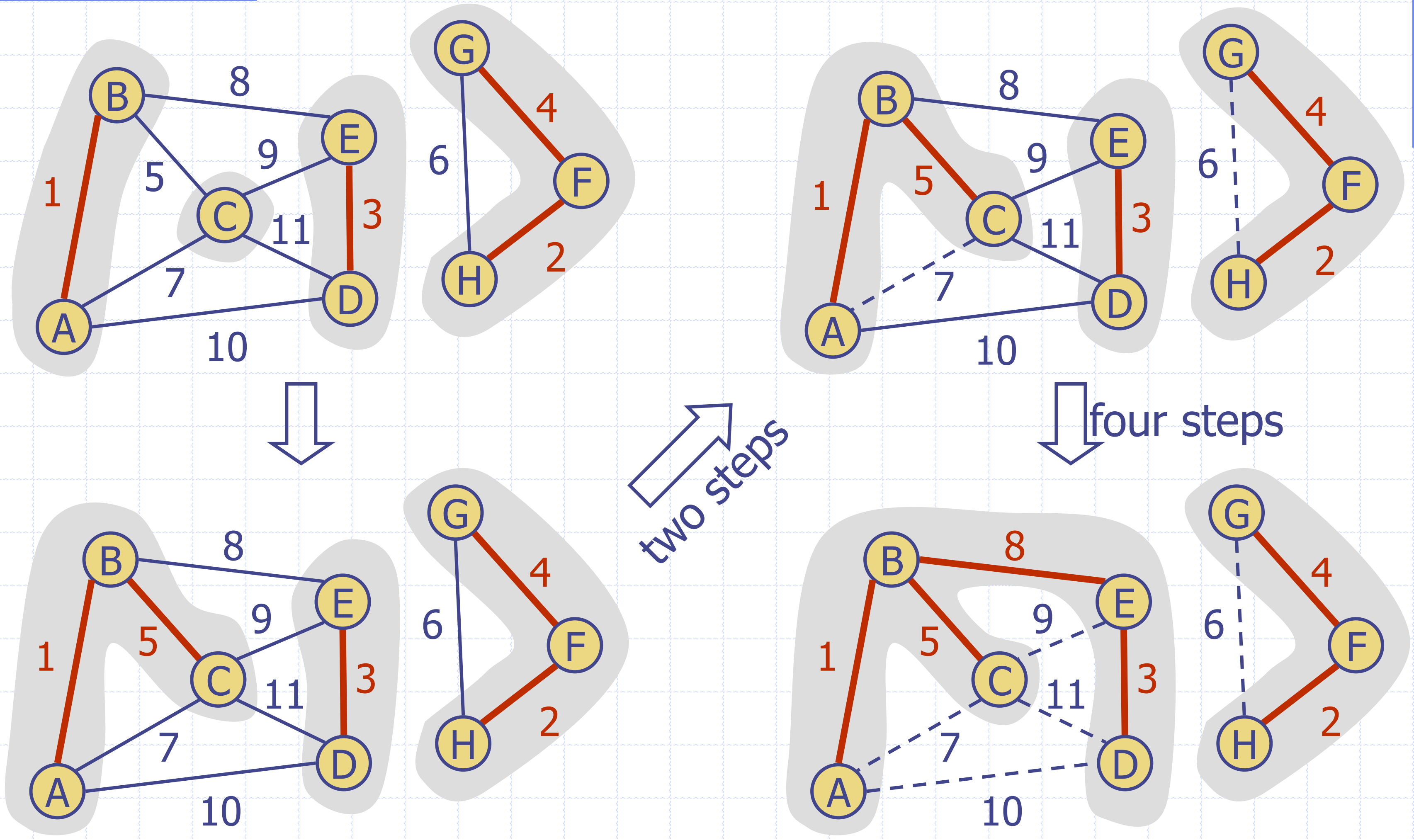
Kruskal’s Algorithm (cont’d)

- The notation ***object.function1.function2*** in the code comes from object-oriented programming, and it means apply ***function2*** to the result of applying ***function1*** to object ***object***. In other words, ***Q.removeMin().getValue()*** means get the value of the minimum element in the priority queue. Similarly, for other statements.
- The edge e selected inside the **while** loop is the edge in the priority queue Q with the minimum weight.
- u and v are the endpoints of this edge.
- The partition property guarantees that each time the body of the **while** loop is executed, the edge added is part of an MST.

Example



Example (cont'd)



Example (cont'd)

- Red edges are edges of the MST.
- Dashed blue edges are edges that were considered by the algorithm but were discarded because their endpoints were already in the same cluster.

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 if it 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
  - findSet(u): return the set storing u
  - union(A, B): replace sets A and B with their union

Data Structures for Kruskal’s Algorithm (cont’d)

- For the disjoint sets data structure we will use the implementation using disjoint forests and the algorithm weighted quick-union with path compression by halving.

Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
  - Cluster merges as unions
  - Cluster locations as findSets

```
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
    $T \leftarrow \emptyset$
    {T is the union of the MSTs of the clusters}
    while T has fewer than $n - 1$ edges do
    $e \leftarrow$ Q.removeMin().getValue()
       $[u, v] \leftarrow$ G.endVertices(e)
       $A \leftarrow$ P.findSet(u)
       $B \leftarrow$ P.findSet(v)
       if $A \neq B$ 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.
- The priority queue can be initialized in $O(m \log m)$ time with repeated insertions or with the bottom up construction algorithm we have presented in the lecture on priority queues in $O(m)$ time.
- The removal operations from the priority queue will take $O(m \log m)$ time.

- Alternative: Sort the edges in increasing order of weight and then scan them. This can also be done in $O(m \log m)$ time with algorithms mergesort of heapsort we will study.

Complexity Analysis (cont'd)

- We will also need n-1 calls to union and at most m calls to findSet. These operations will take $\boldsymbol{O(n \log m)}$ time for the chosen disjoint-set data structure and algorithms.
- Therefore, the running time of Kruskal's algorithm is $\boldsymbol{O((n + m) \log n)}$

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** $D(u)$**.**
  - We **update the labels** $D(z)$ of the vertices z adjacent to u.

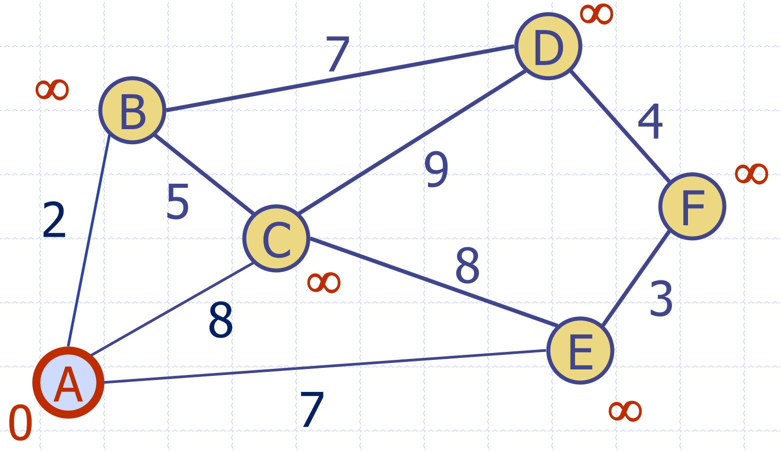
Prim-Jarnik's Algorithm (cont'd)

- 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.)

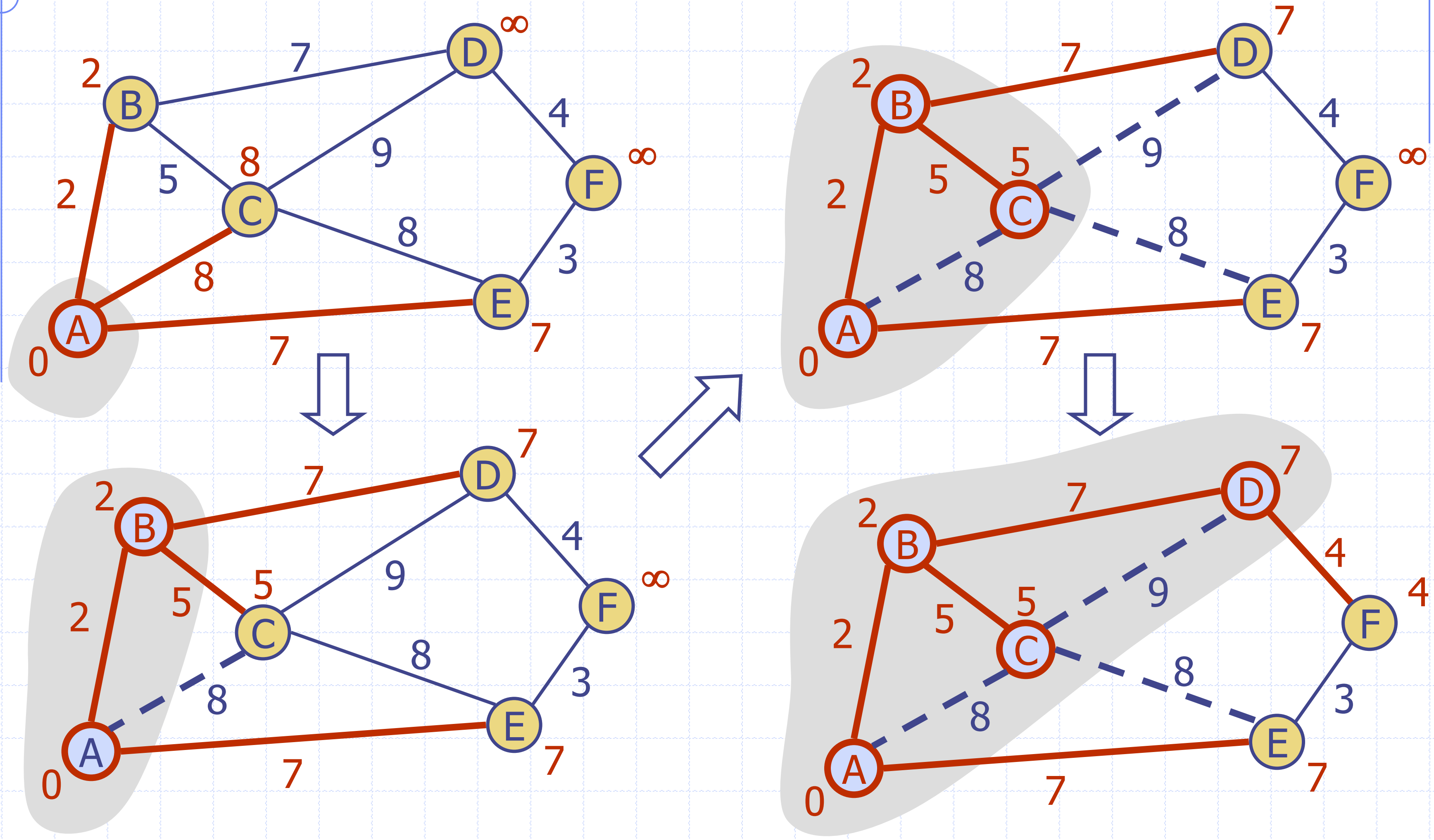
```
Algorithm PrimJarnikMST(G)
    Pick any vertex $v$ of $G$
    $D[v] \leftarrow 0$
    for each vertex $u \neq v$ do
        $D[u] \leftarrow +\infty$
    Initialize $T \leftarrow \emptyset$.
    Initialize a priority queue $Q$ with an entry $((u,null),D[u])$ for each vertex $u$,
    where $(u,null)$ is the value and $D[u]$ is the key.
    while $Q$ is not empty do
        $(u,e) \leftarrow Q.removeMin()$
        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
                $D[z] \leftarrow weight((u,z))$
                Change to $(z,(u,z))$ the value of vertex $z$ in $Q$
                Change to $D[z]$ the key of vertex $z$ in $Q$
    return the tree $T$
```

Example

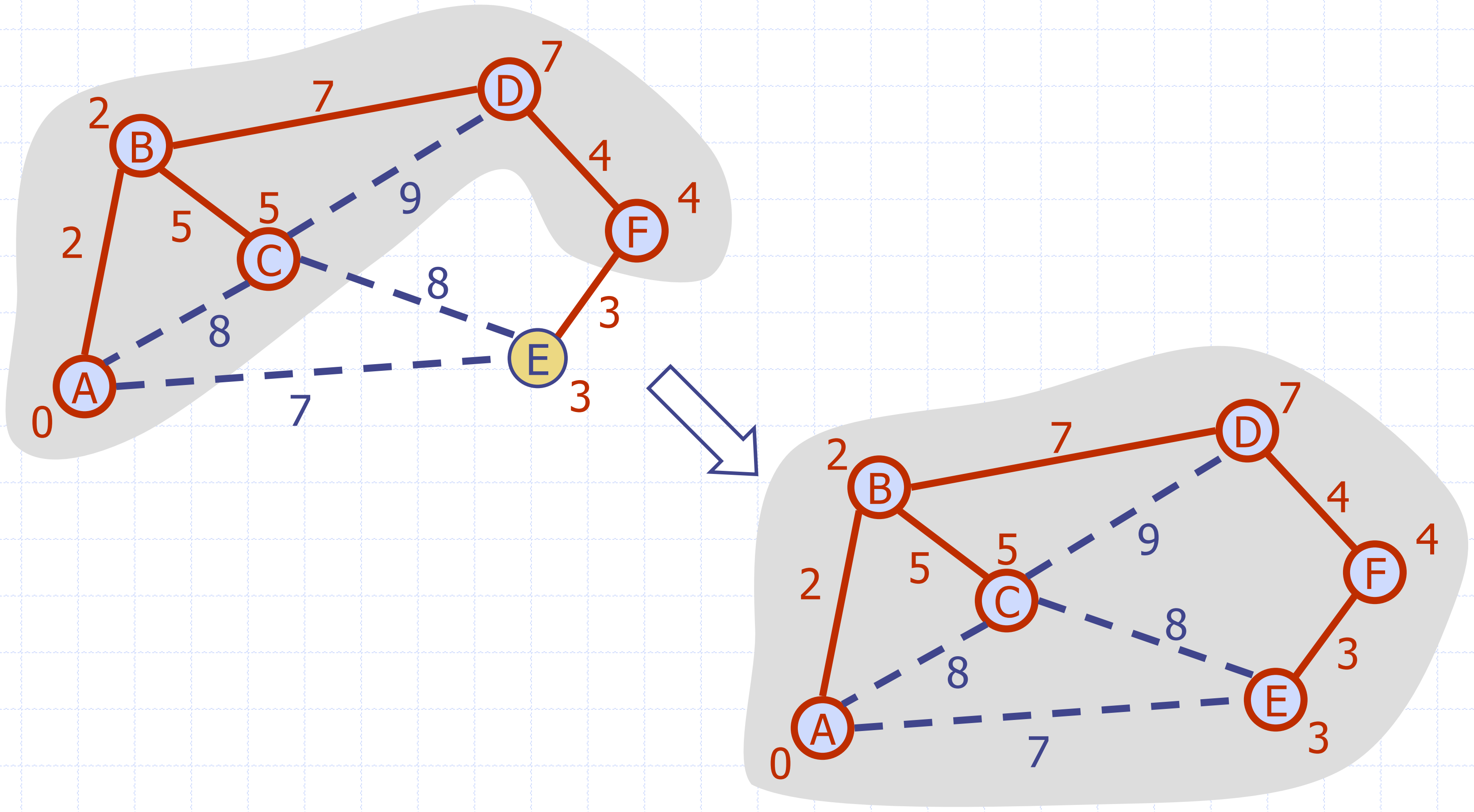


This is the graph before the **while** loop is executed. The values of *D[v]* are shown in red near the vertices. In the first iteration of the **while** loop, the vertex A will be selected and then the algorithm will proceed as shown in the following slide.

Example



Example (contd.)



Example (cont'd)

- In an implementation, we can use a very big positive integer in the place of ∞. All the weights are then assumed to be smaller than this integer.
- In the previous figures, red edges denote MST edges when they are inside the cloud.
- Red edges also denote edges of minimum weight connecting vertices in the MST (cloud) to vertices in Q.
- Dashed blue edges denote edges that have been discarded, inside the **if** statement of the algorithm, in favour of (red) edges with minimum weight.
- Upon completion of the algorithms, red edges form the MST.

Complexity Analysis

- Let n and m denote the number of vertices and edges of the input graph respectively.
- The **for** loop takes $O(n)$ time.
- Since the priority queue is implemented as a min heap, we can initialize it in $O(n \log n)$ time with repeated insertions or in $O(n)$ using the bottom up construction algorithm we have presented in the lectures for heaps.
- We can extract the vertex u from the priority queue in $O(\log n)$ time. So the complexity for extracting all vertices is $O(n \log n)$.

Complexity Analysis (cont'd)

- We can do the two change operations inside the **if** statement 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) so the total updates can be done in $O(m \log n)$.
- 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
- M. T. Goodrich and R. Tamassia. Δομές Δεδομένων και Αλγόριθμοι σε Java. 5η έκδοση. Εκδόσεις Δίαυλος. 2013.
    - Chapter 13
- R. Sedgewick. Algorithms in C. 3rd edition. Part 5. Graph Algorithms.
    - Chapter 20.