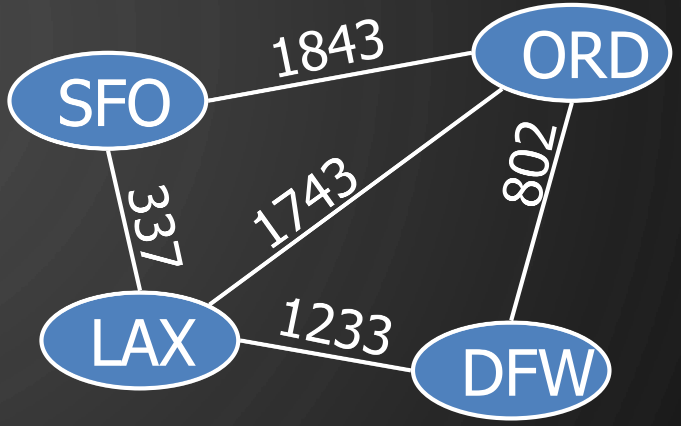
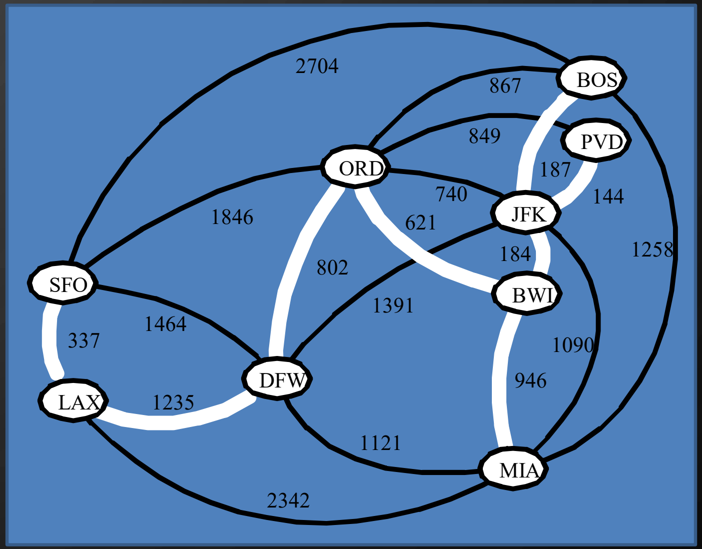
CHAPTER 14
GRAPH ALGORITHMS



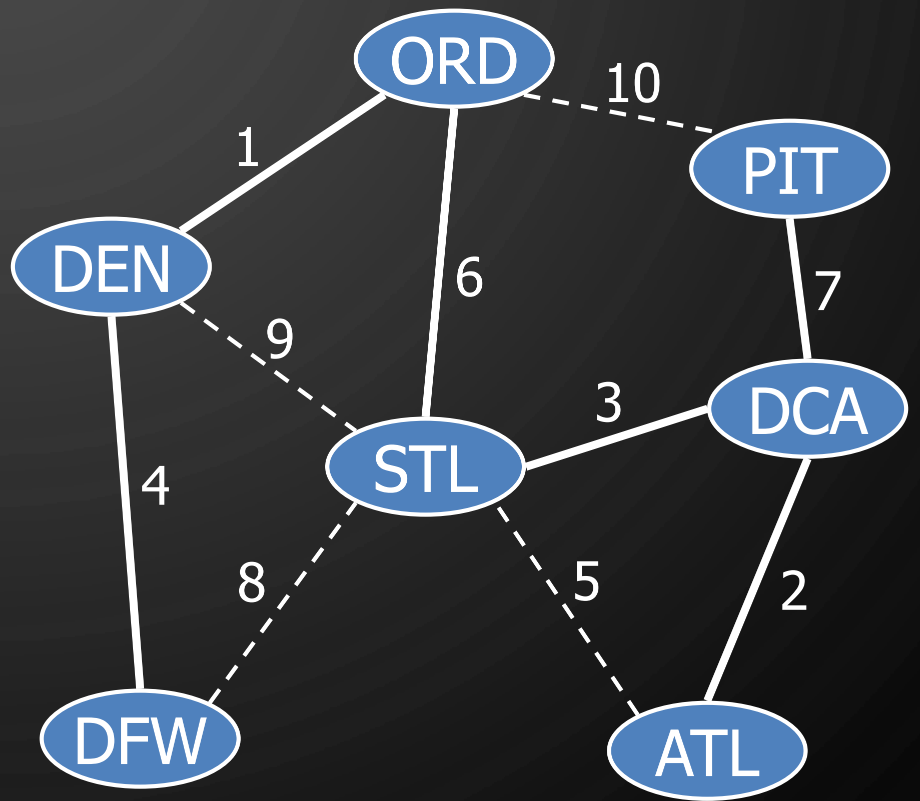
ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

MINIMUM SPANNING TREES



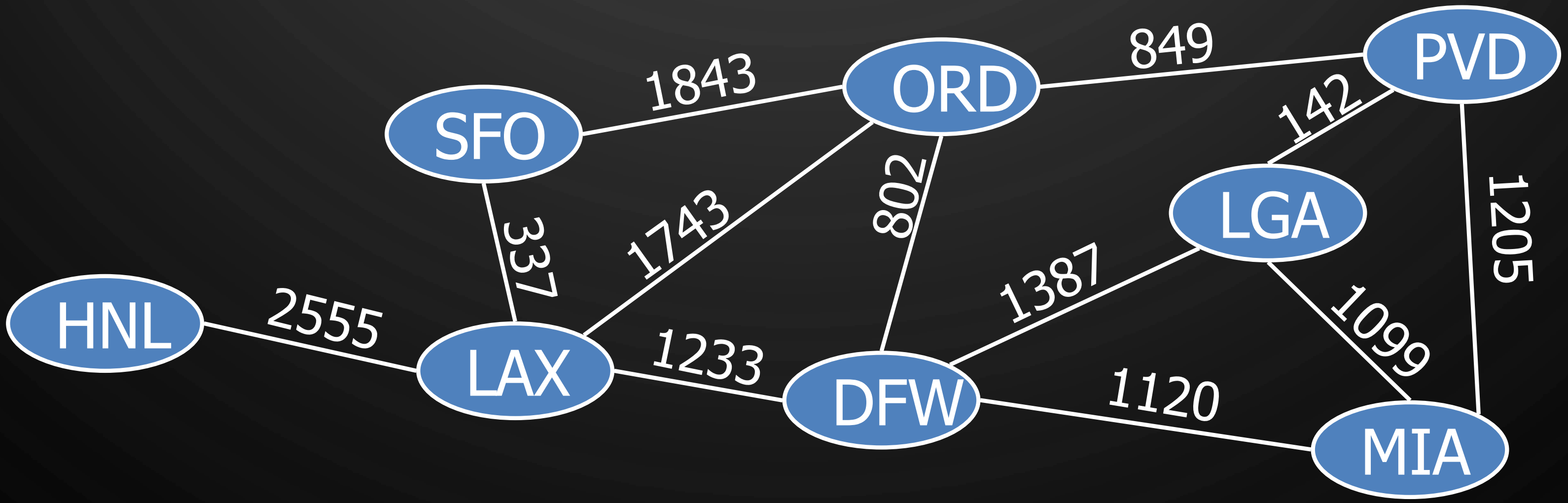
MINIMUM SPANNING TREE

- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation networks



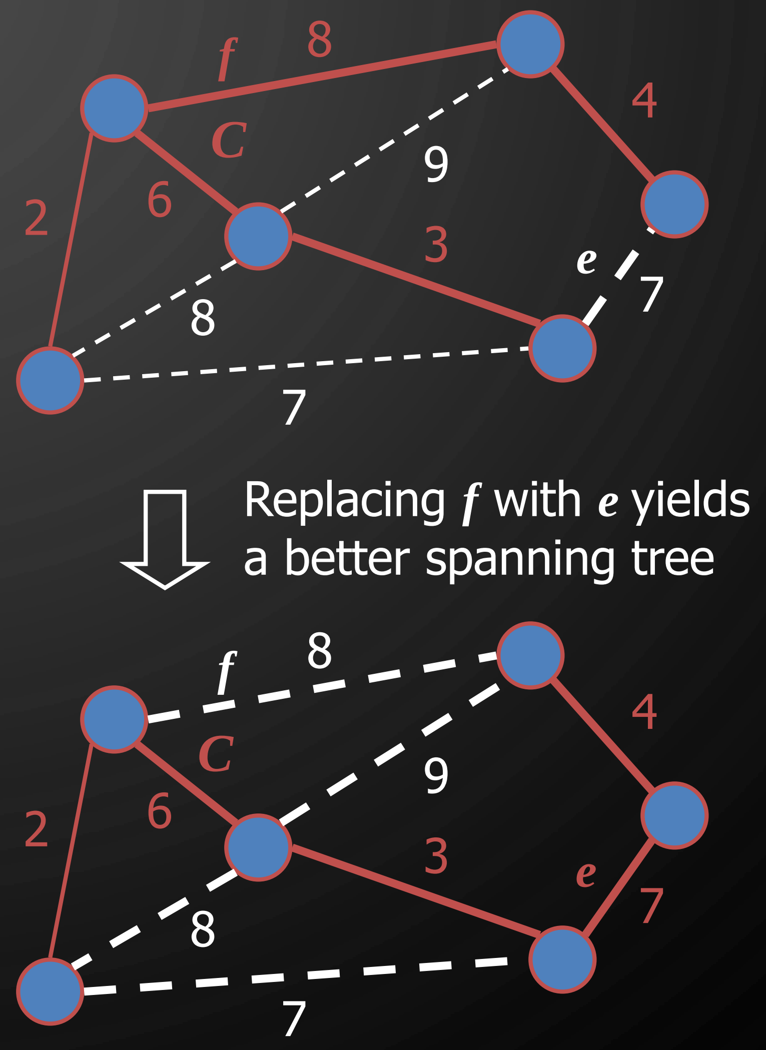
EXERCISE
MST

- Show an MST of the following graph.



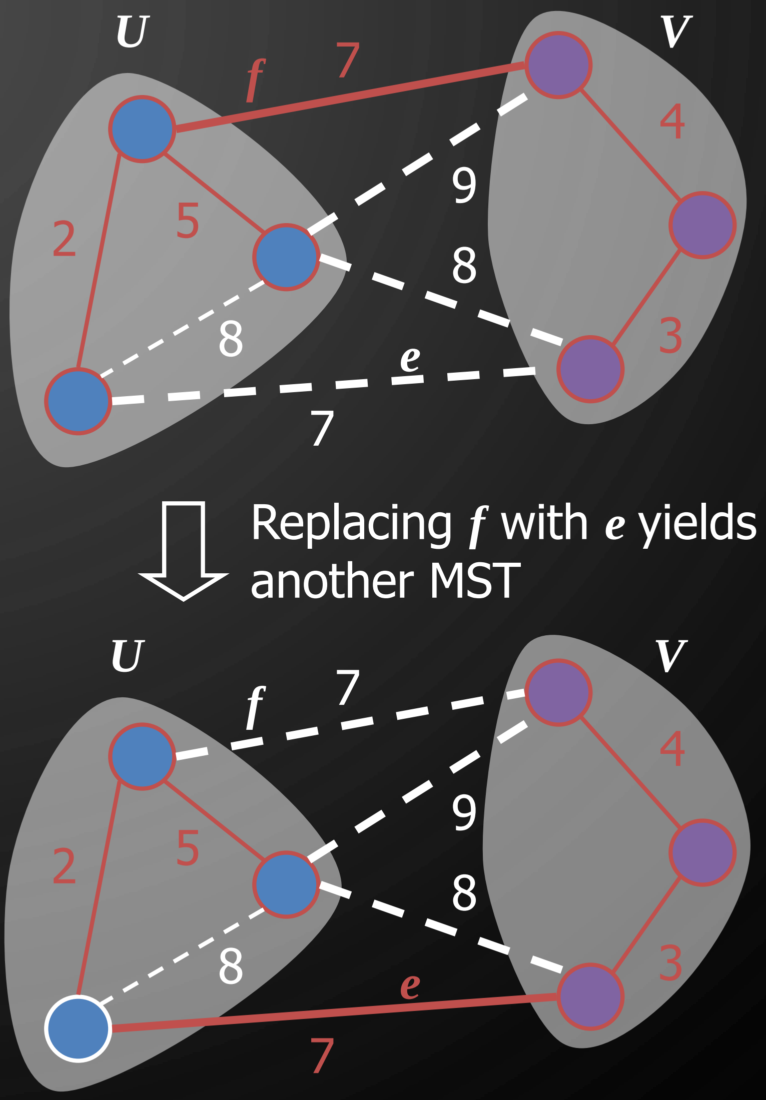
CYCLE PROPERTY

- **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



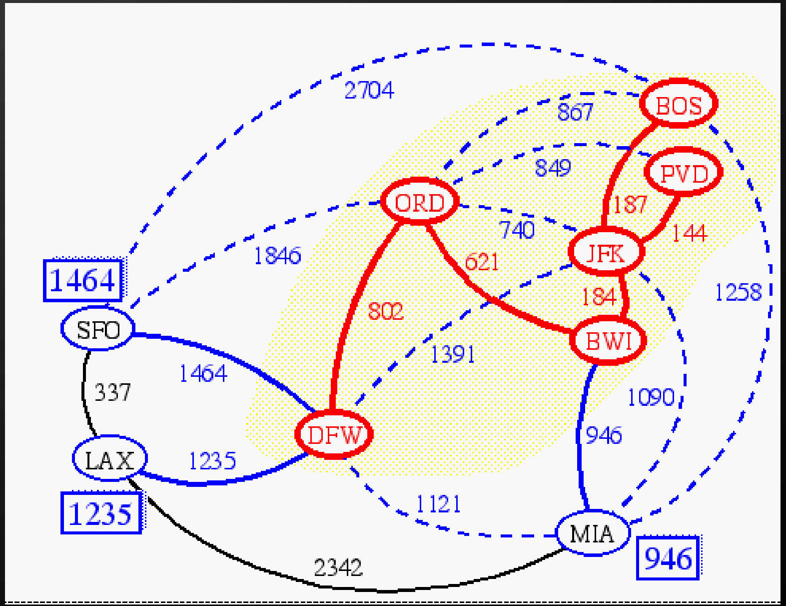
PARTITION PROPERTY

- **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 by contradition:
  - 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

- 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 a 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

- An adaptable priority queue stores the vertices outside the cloud
  - Key: distance, $D[v]$
  - Element: vertex v
  - $Q.replace(i, k)$ changes the key of an item
- We store three labels with each vertex v:
  - Distance $D[v]$
  - Parent edge in MST $P[v]$
  - Locator in priority queue

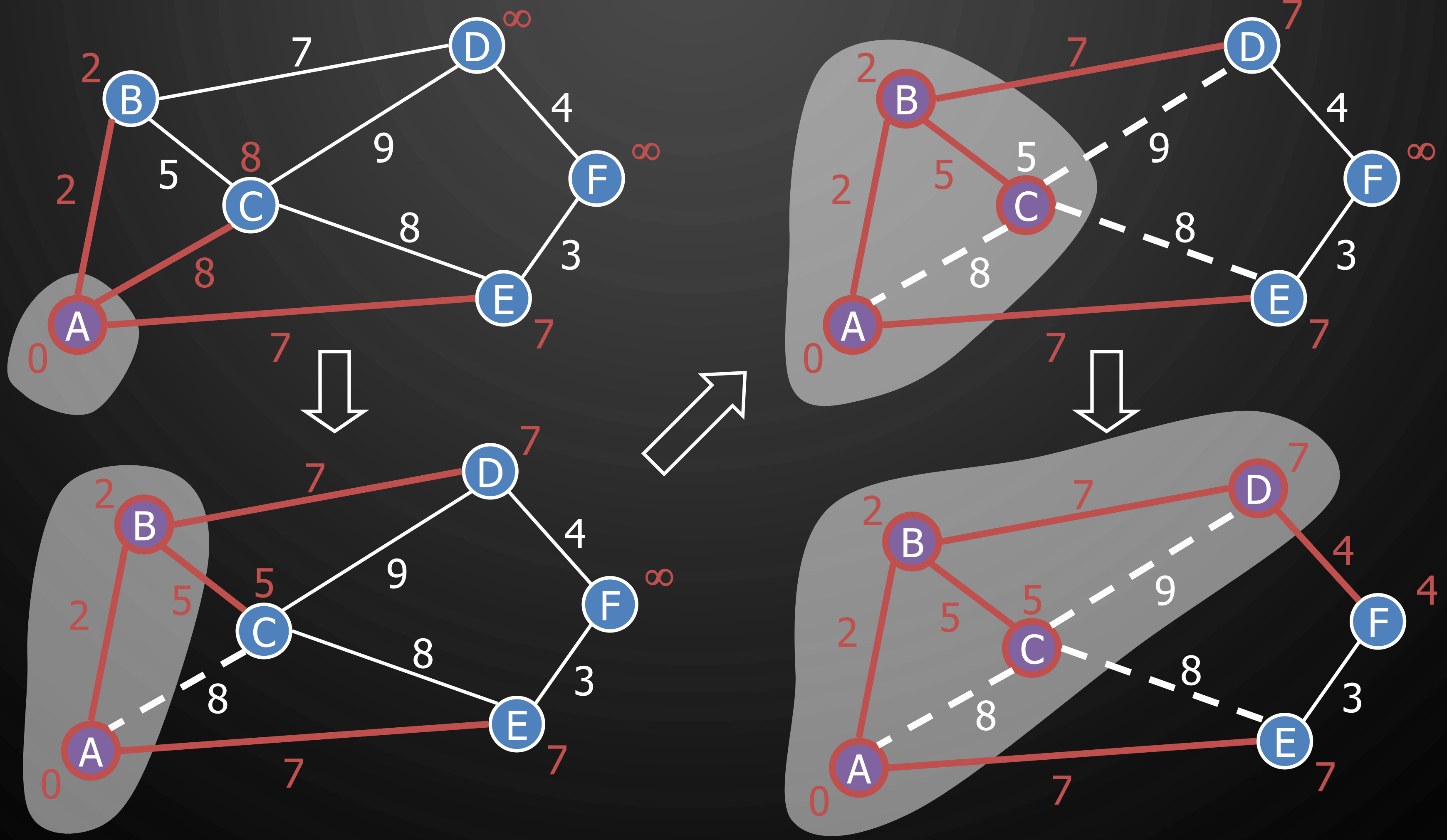
Algorithm PrimJarnikMST(G)

Input: A weighted connected graph G

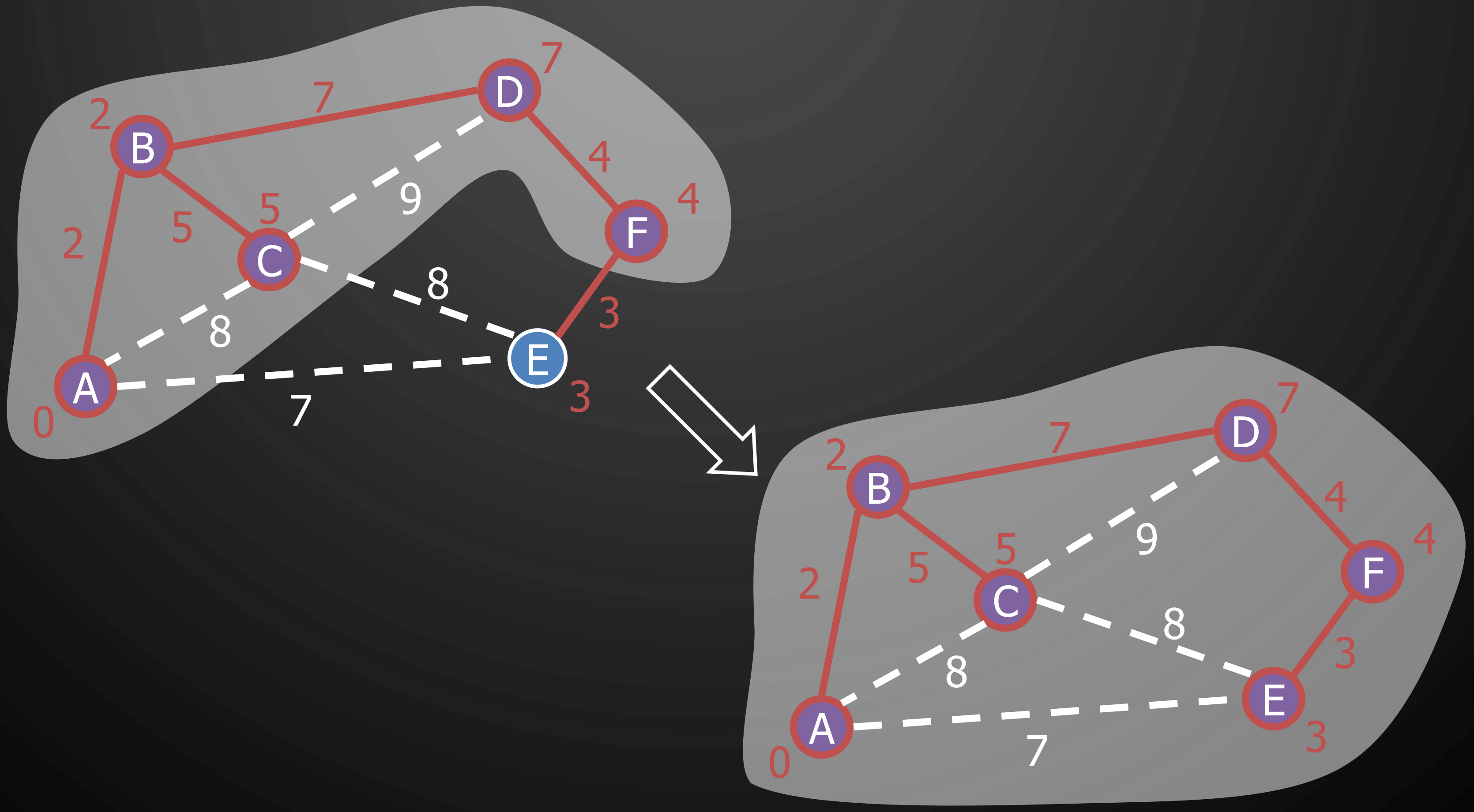
Output: A minimum spanning tree T of G

```
Pick any vertex s of G
D[s] ← 0;  P[s] ← ∅
for each vertex v ≠ s do
  D[v] ← ∞;  P[v] ← ∅
T ← ∅
Priority queue Q of vertices with
D[v] as the key
while ¬Q.isEmpty() do
  u ← Q.removeMin()
  Add vertex u and edge P[u] to T
  for each e ∈ u.outgoingEdges() do
    v ← G.opposite(u, e)
    if e.weight() < D[v]
      D[v] ← e.weight();  P[v] ← e
      Q.replace(v, D[v])
return T
```

EXAMPLE

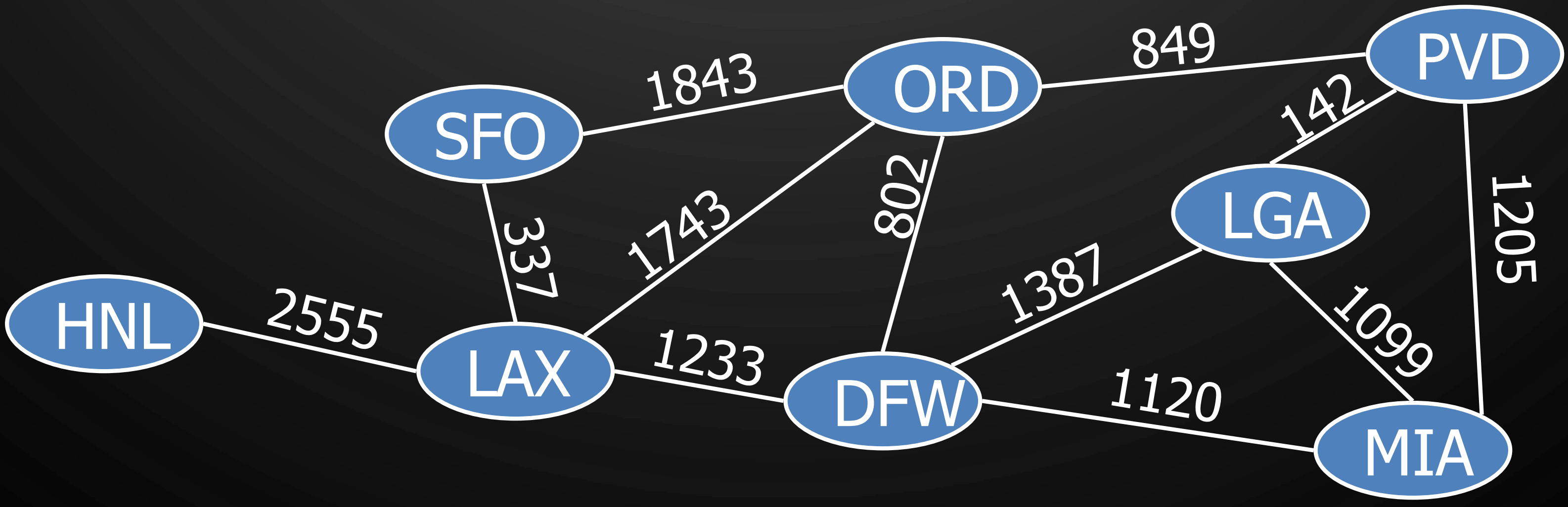


EXAMPLE



EXERCISE
PRIM'S MST ALGORITHM

- Show how Prim's MST algorithm works on the following graph, assuming you start with SFO
  - Show how the MST evolves in each iteration (a separate figure for each iteration).



ANALYSIS

- Graph operations
  - Method incidentEdges is called 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 $O\big((n+m)\log n\big)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- If the graph is connected the running time is $O(m \log n)$

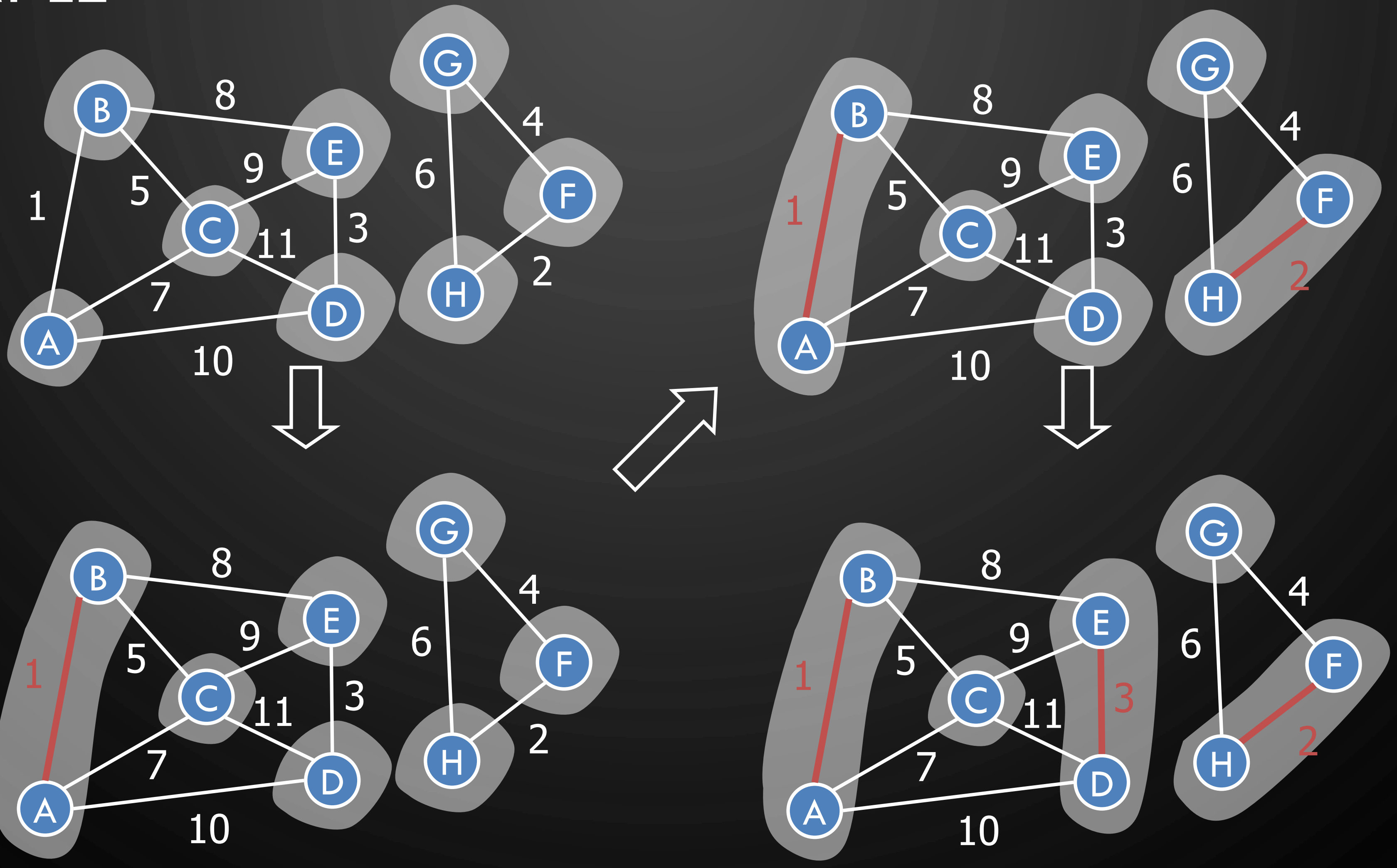
KRUSKAL'S ALGORITHM

- 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

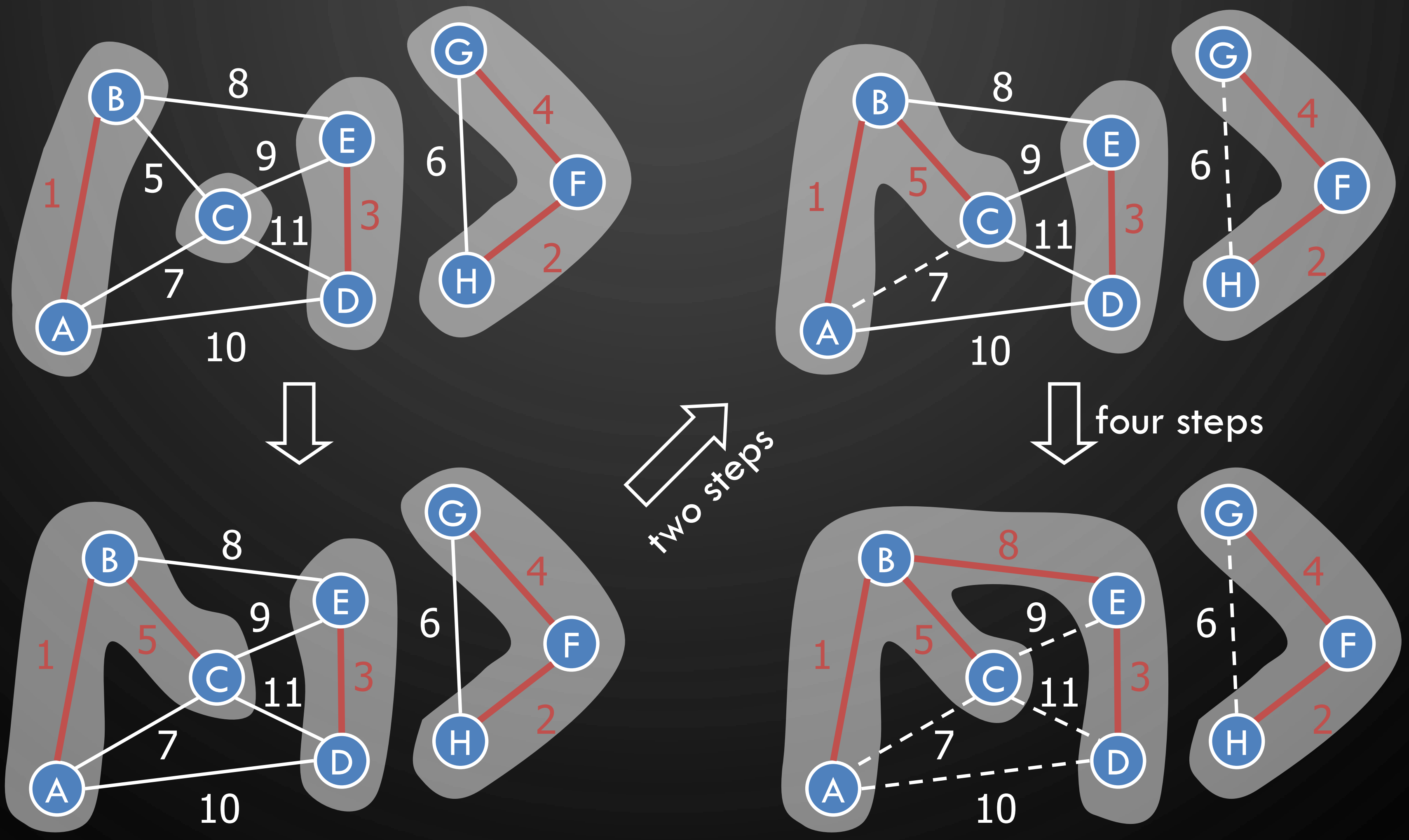
Algorithm KruskalMST(G)

1. **for each** vertex $v \in G$.vertices() **do**
2. Define a cluster $C(v) \leftarrow \{v\}$
3. Initialize a priority queue Q of edges using the weights as keys
4. $T \leftarrow \emptyset$
5. **while** T has fewer than $n-1$ edges **do**
6. $(u, v) \leftarrow Q$.removeMin()
7. **if** $C(u) \neq C(v)$ **then**
8. Add (u, v) to T
9. Merge $C(u)$ and $C(v)$
10. **return** T

EXAMPLE

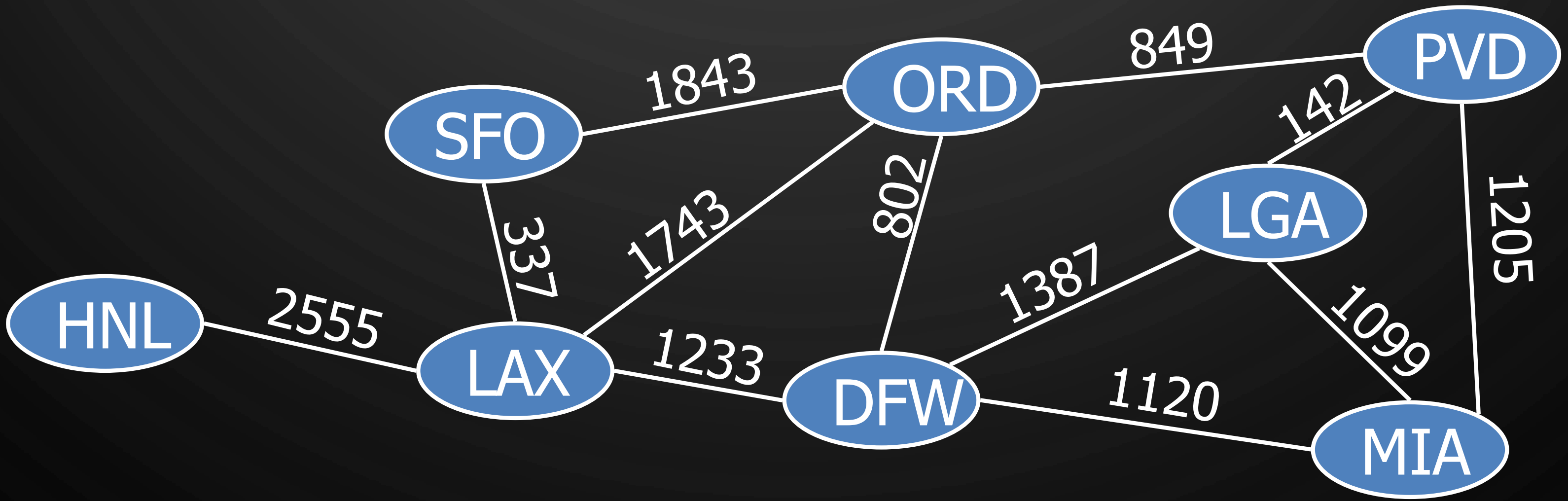


EXAMPLE

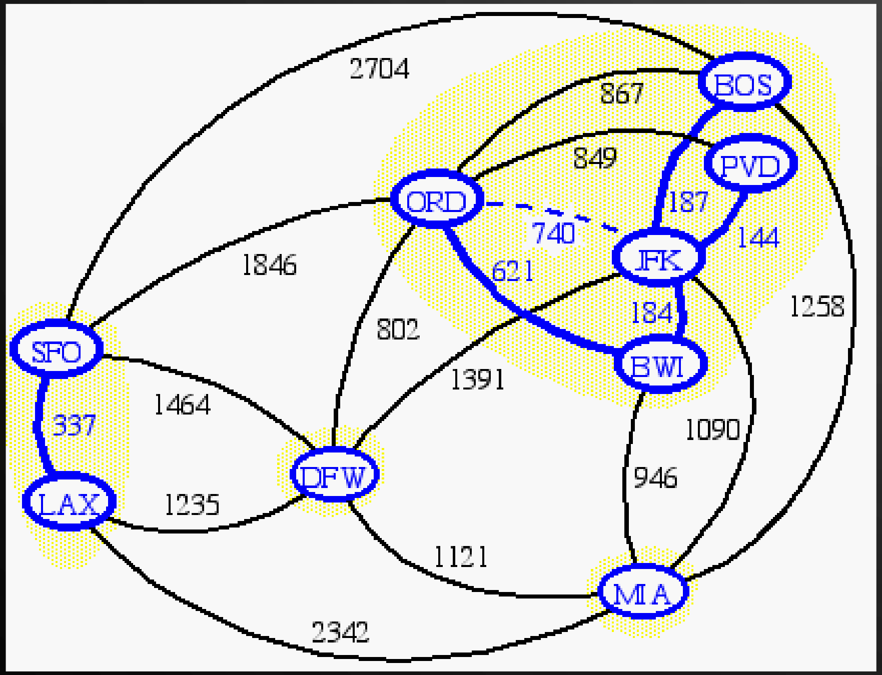


EXERCISE
KRUSKAL'S MST ALGORITHM

- Show how Kruskal's MST algorithm works on the following graph.
  - Show how the MST evolves in each iteration (a separate figure for each iteration).



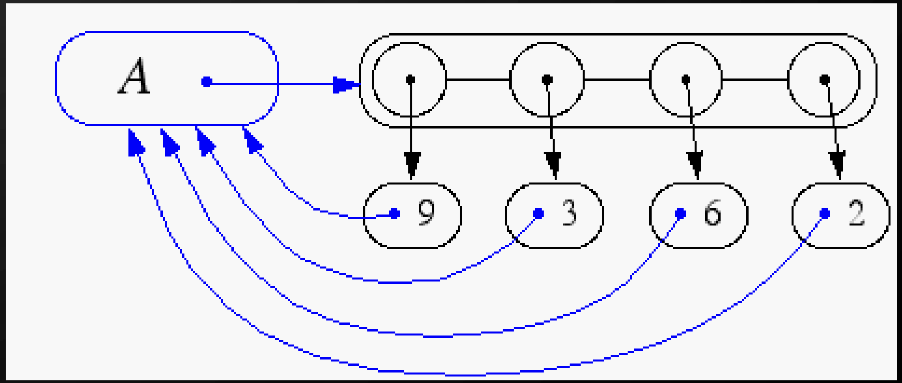
DATA STRUCTURE FOR KRUSKAL'S ALGORITHM



- The algorithm maintains a forest of trees
- 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 the operations:
  - $\mathrm{find}(u)$: return the set storing u
  - $\mathrm{union}(A, B)$: replace the sets A and B with their union

LIST-BASED PARTITION

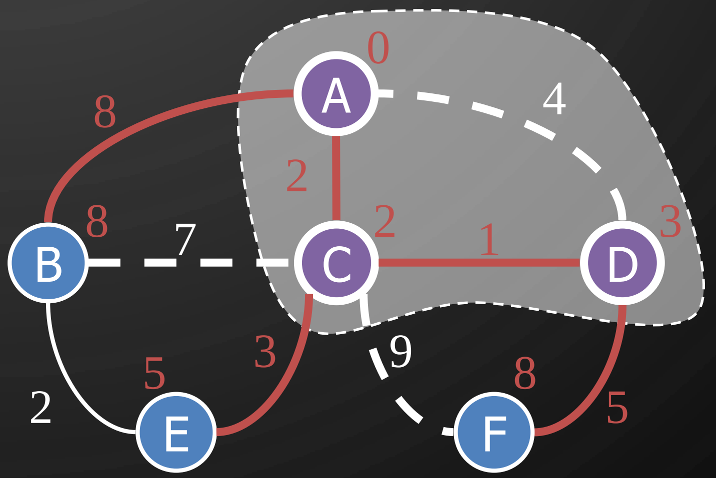
- Each set is stored in a List
- Each element has a reference back to the set
  - Operation $\text{find}(u)$ takes $O(1)$ time, and returns the set of which u is a member.
  - In operation $\text{union}(A, B)$, we move the elements of the smaller set to the sequence of the larger set and update their references
  - The time for operation $union(A, B)$ is $O(\min(|A|, |B|))$
- 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



ANALYSIS

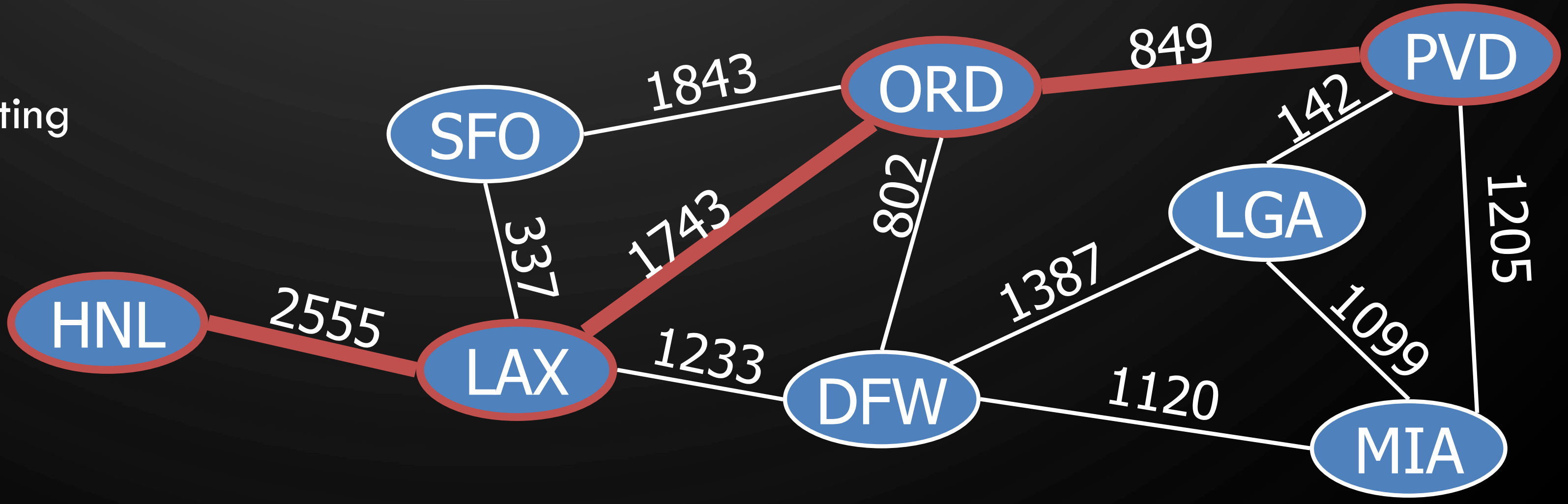
- A partition-based version of Kruskal's Algorithm
  - Cluster merges as unions
  - Cluster locations as finds
- Complexity – $O\big((n+m)\log n\big)$ time
  - At most m removals from the priority queue - $O(m \log n)$
  - Each vertex can be merged at most $\log n$ times, as the clouds tend to "double" in size - $O(n \log n)$

SHORTEST PATHS



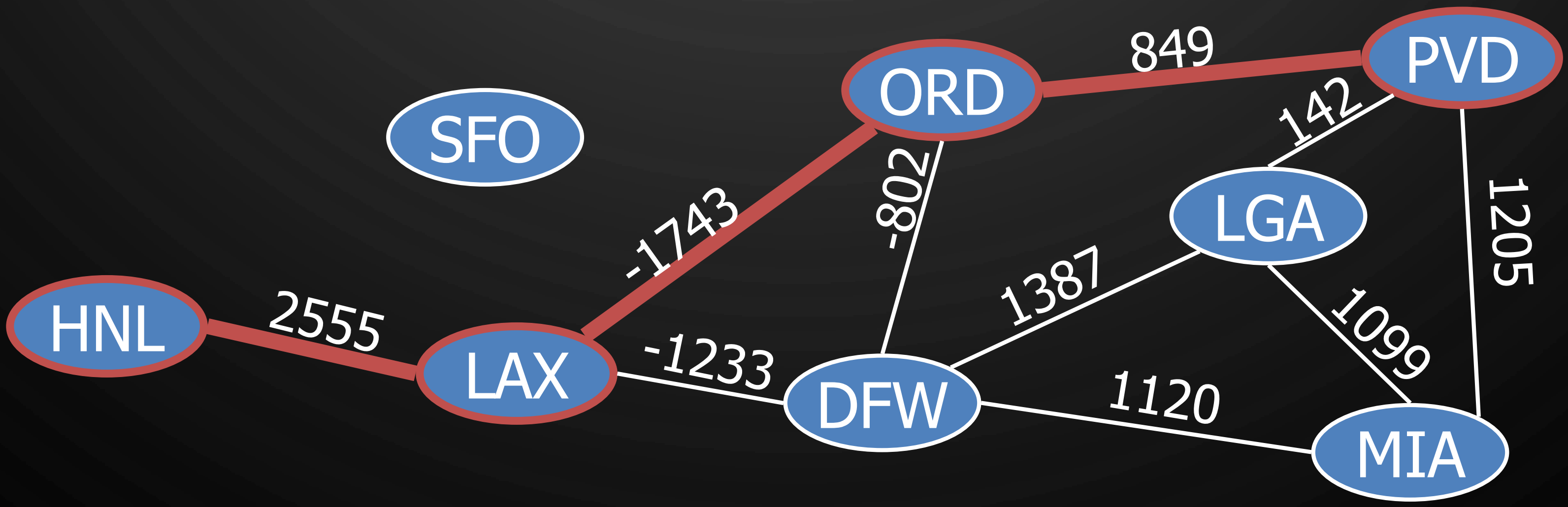
SHORTEST PATH PROBLEM

- 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



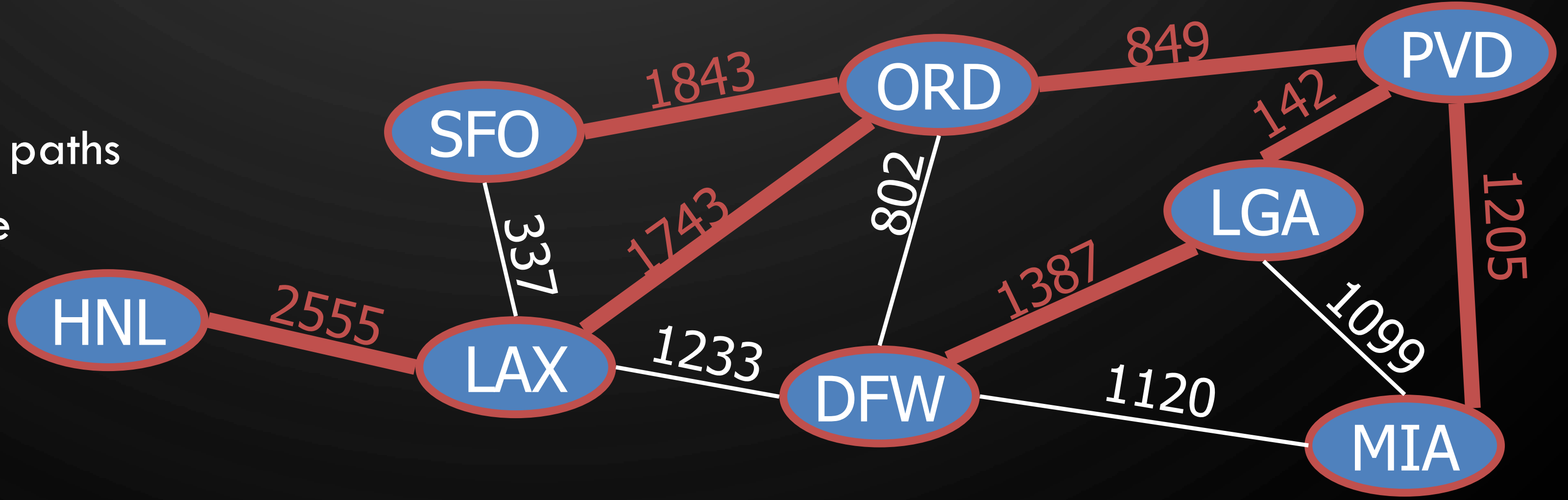
SHORTEST PATH PROBLEM

- If there is no path from v to u, we denote the distance between them by $d(v,u) = \infty$
- What if there is a negative-weight cycle in the graph?



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

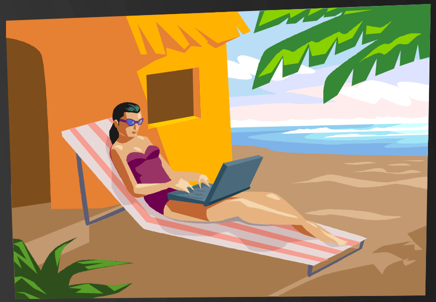


DIJKSTRA'S ALGORITHM

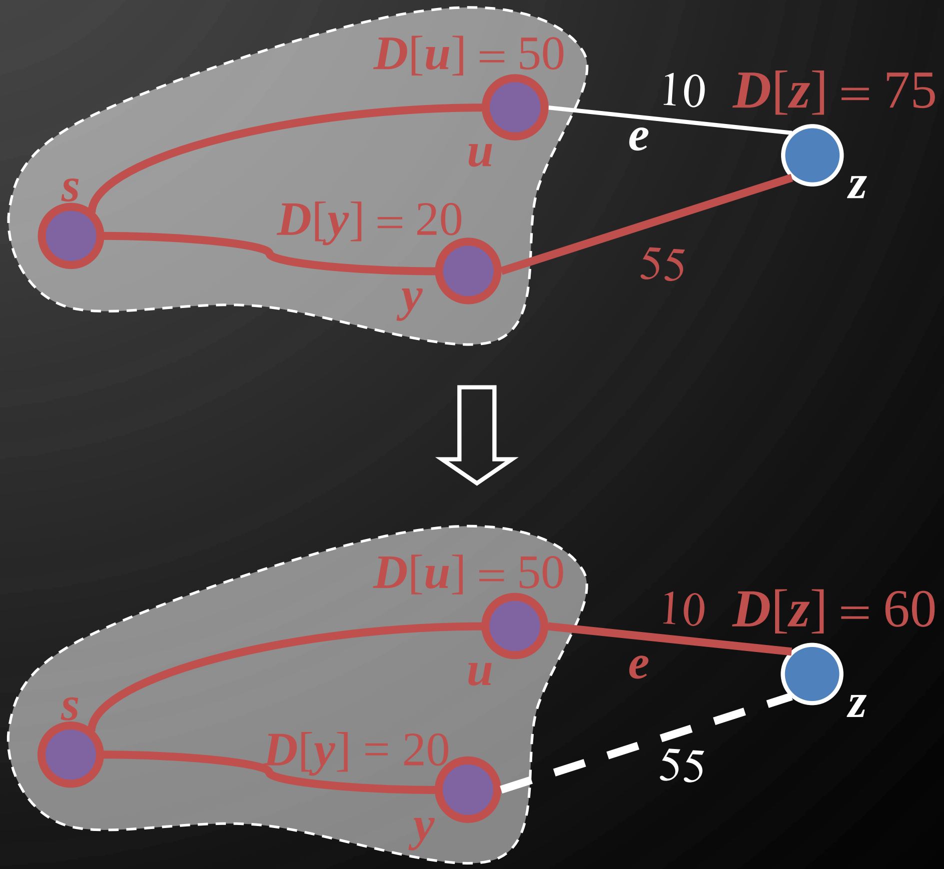
- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s (single-source shortest paths)
- Assumptions:
  - The graph is connected
  - The edges are undirected
  - The edge weights are **nonnegative**
- Extremely similar to Prim-Jarnik's MST Algorithm

- 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
- The label $D[v]$ is initialized to positive infinity
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $D[v]$
  - We update the labels of the vertices adjacent to u, in a process called edge relaxation

EDGE RELAXATION

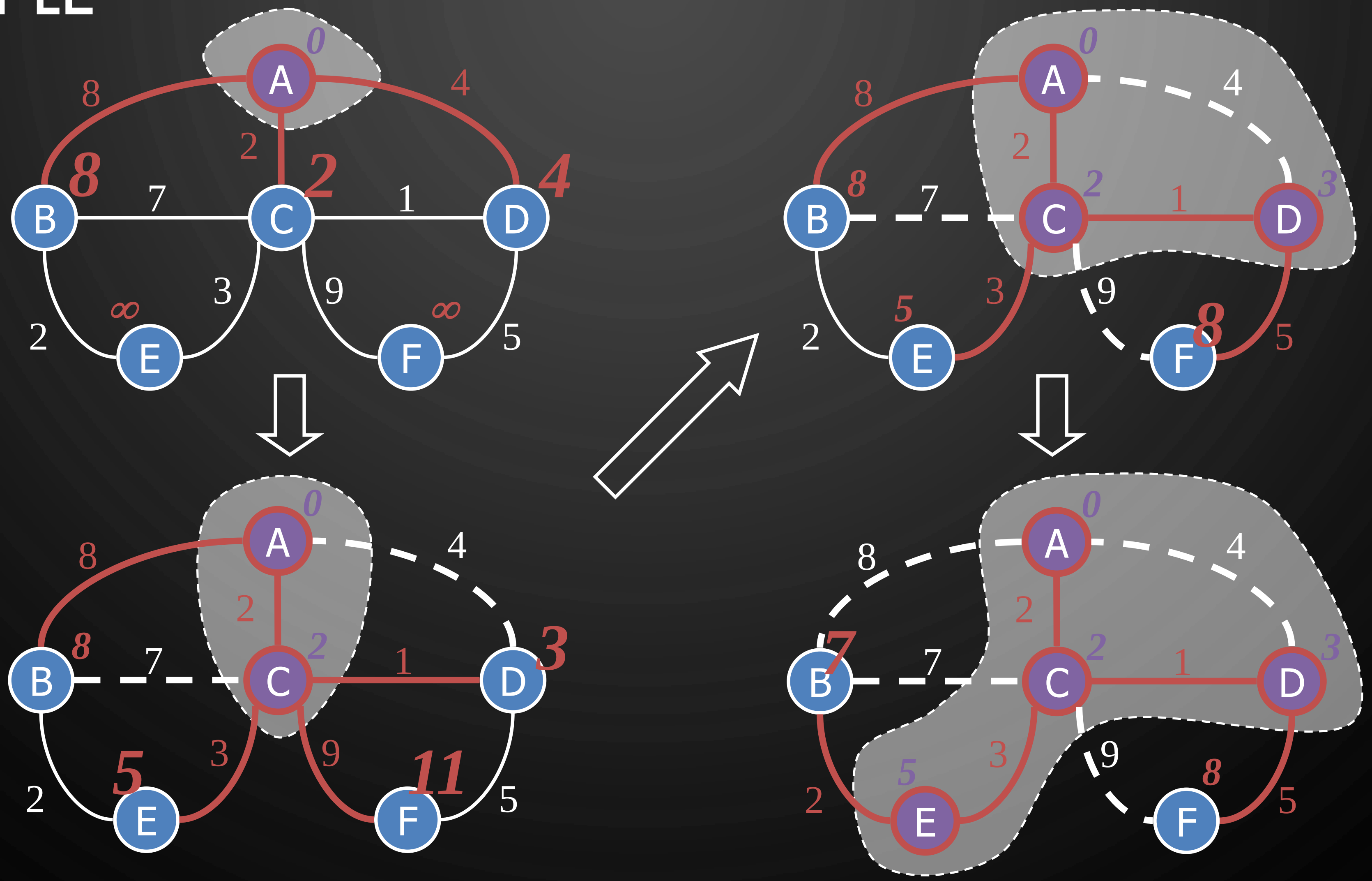


- Consider an edge $e = (u, z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $D[z]$ as follows:
- $D[z] \leftarrow \min\big(D[z], D[u] + e.weight()\big)$

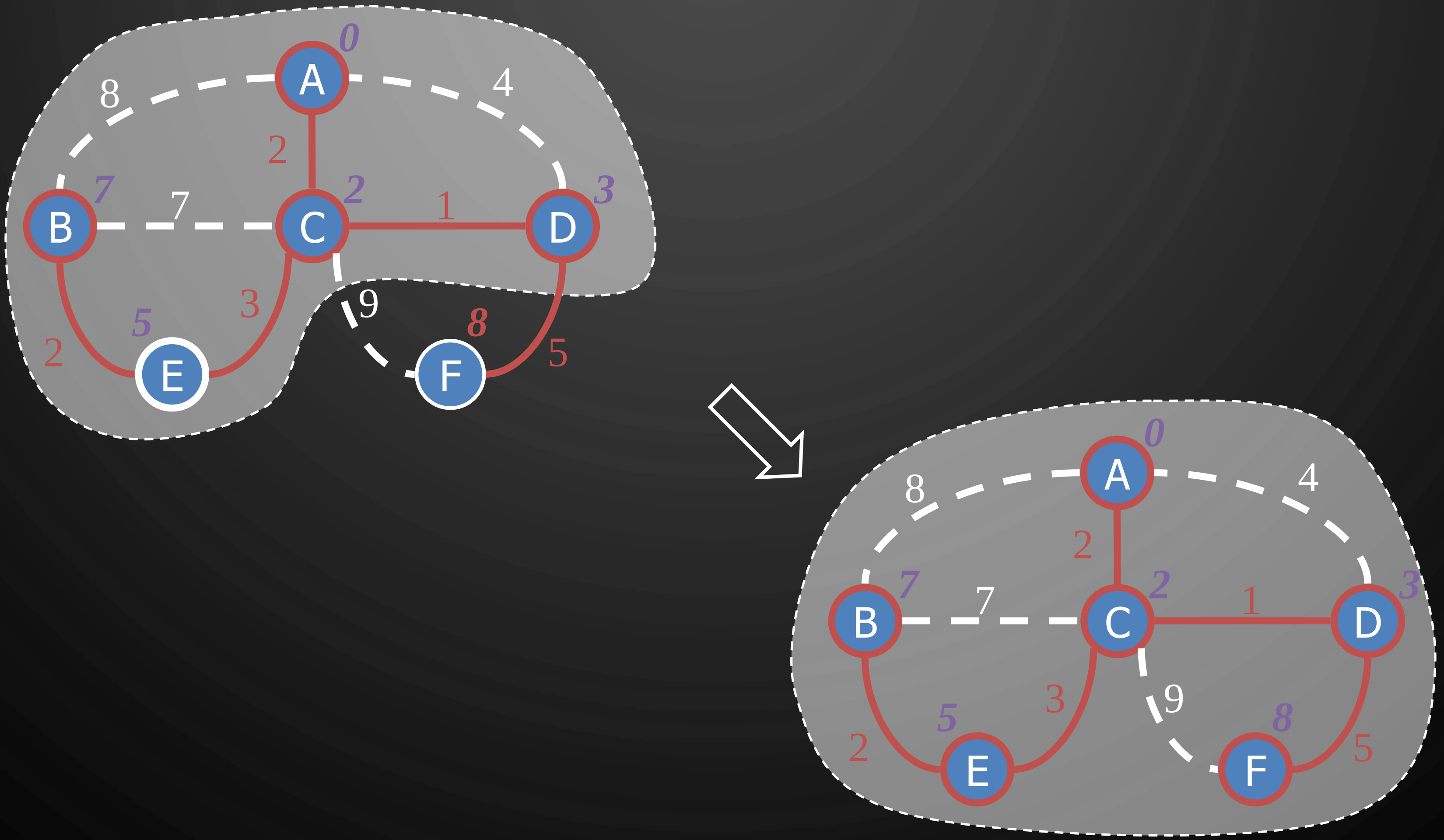


EXAMPLE

1. Pull in one of the vertices with **red labels**
2. The relaxation of edges updates the labels of **LARGER font size**

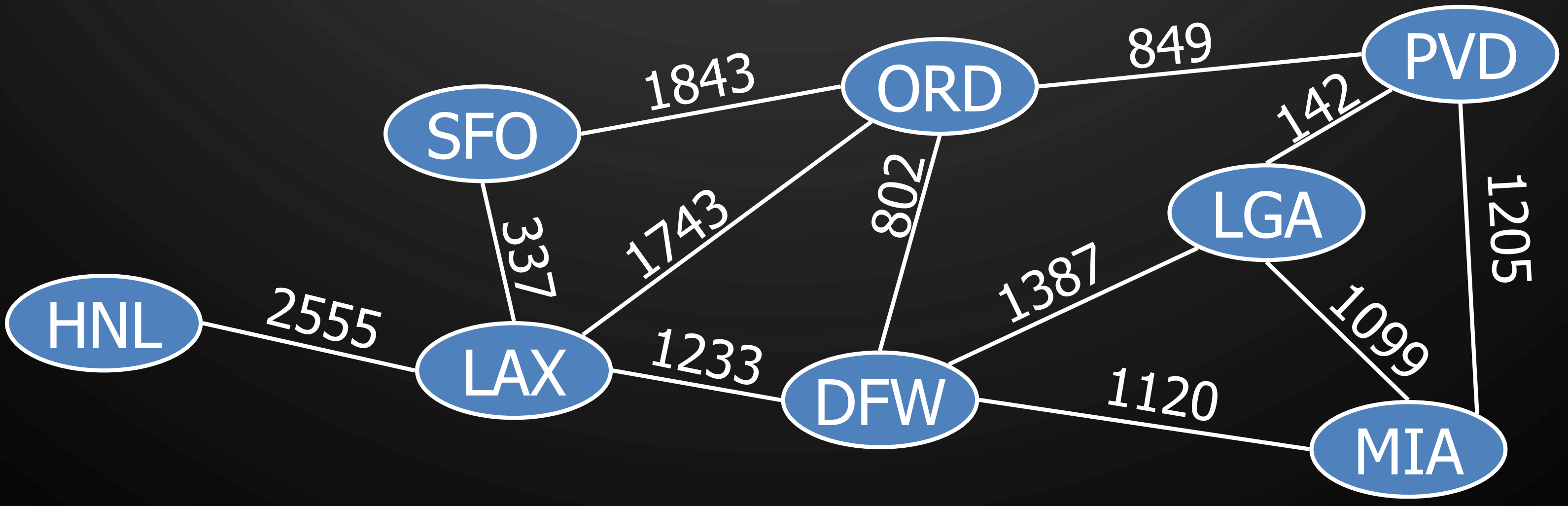


EXAMPLE

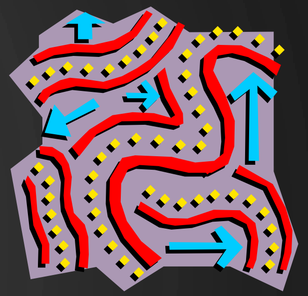


EXERCISE
DIJKSTRA'S ALGORITHM

- Show how Dijkstra's algorithm works on the following graph, assuming you start with SFO, i.e., s =SFO.
  - Show how the labels are updated in each iteration (a separate figure for each iteration).



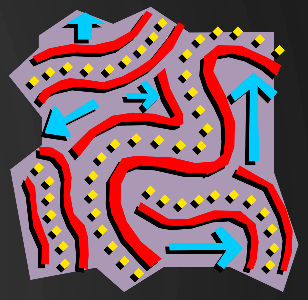
DIJKSTRA'S ALGORITHM



- An adaptable priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- We store with each vertex:
  - distance $D[v]$ label
  - locator in priority queue

```
Algorithm  Dijkstras_sssp(G,s)
Input: A simple undirected weighted graph G with
       nonnegative edge weights and a source vertex s
Output: A label D[v] for each vertex v of G,
        such that D[u] is the length of the
        shorted path from s to v
D[s] ← 0;  D[v] ← ∞ for each vertex v ≠ s
Let priority queue Q contain all the vertices of G
using D[v] as the key
while ¬Q.isEmpty() do //O(n) iterations
  //pull a new vertex u in the cloud
  u ← Q.removeMin() //O(log n)
  for each edge e ∈ G.outgoingEdges(u) do //O(deg(u)) iterations
    //relax edge e
    v ← G.opposite(u,e)
    if D[u] + e.weight() < D[v] then
      D[v] ← D[u] + e.weight()
      Q.replace(v,D[v]) //O(log n)
```

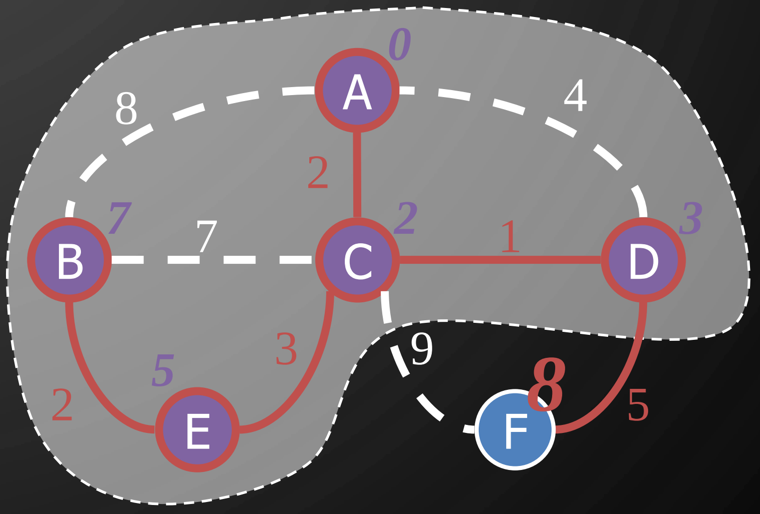
ANALYSIS



- Graph operations
  - We find 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 $O\big((n+m)\log n\big)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg v = 2m$
- The running time can also be expressed as $O(m \log n)$ if the graph is connected

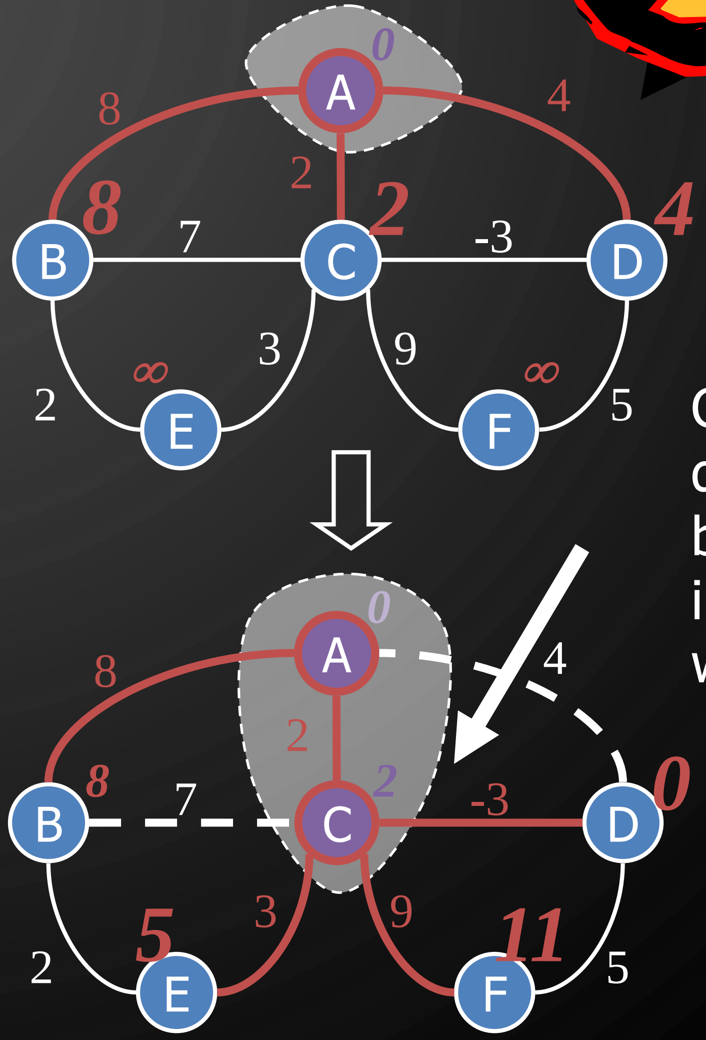
WHY DIJKSTRA'S ALGORITHM WORKS

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
- Proof by contradiction
  - 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 $D[C] = 2$!

BELLMAN-FORD ALGORITHM

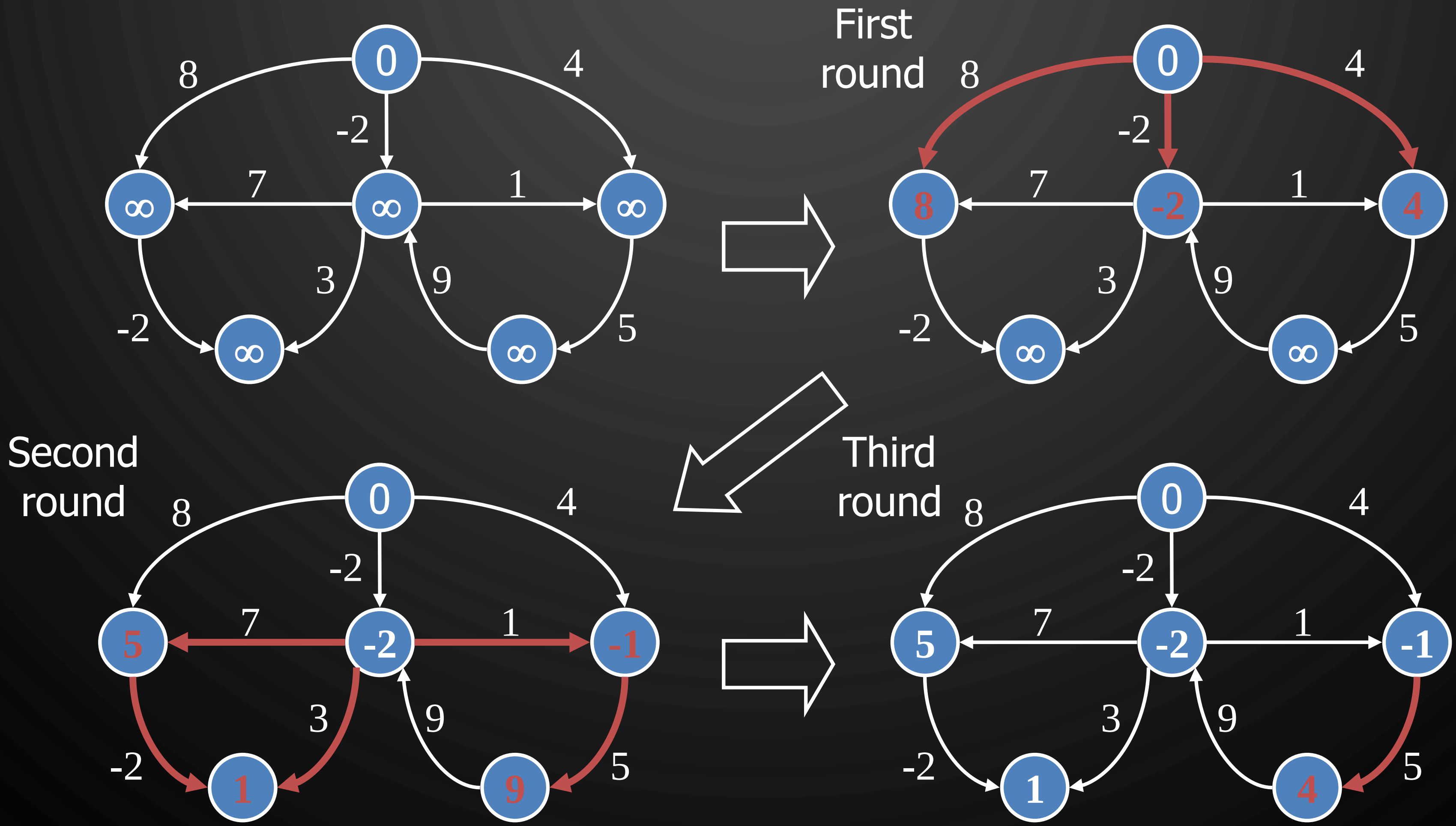
- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: $O(nm)$
- Can be extended to detect a negative-weight cycle if it exists
  - How?

Algorithm BellmanFord(G, s)

1. Initialize $D[s] \leftarrow 0$ and
   $D[v] \leftarrow \infty$ for all vertices $v \neq s$
2. **for** $i \leftarrow 1 \ldots n-1$ **do**
3.   **for each** $e \in G$.edges() **do**
4.   //relax edge e
5. $u \leftarrow G$.origin(e)
6. $z \leftarrow G$.opposite(u, e)
7.   **if** $D[u] + e$.weight() $< D[z]$ **then**
8. $D[z] \leftarrow D[u] + e$.weight()

- Nodes are labeled with their $D[v]$ values

BELLMAN-FORD EXAMPLE



EXERCISE
BELLMAN-FORD'S ALGORITHM

- Show how Bellman-Ford's algorithm works on the following graph, assuming you start with the top node
  - Show how the labels are updated in each iteration (a separate figure for each iteration).