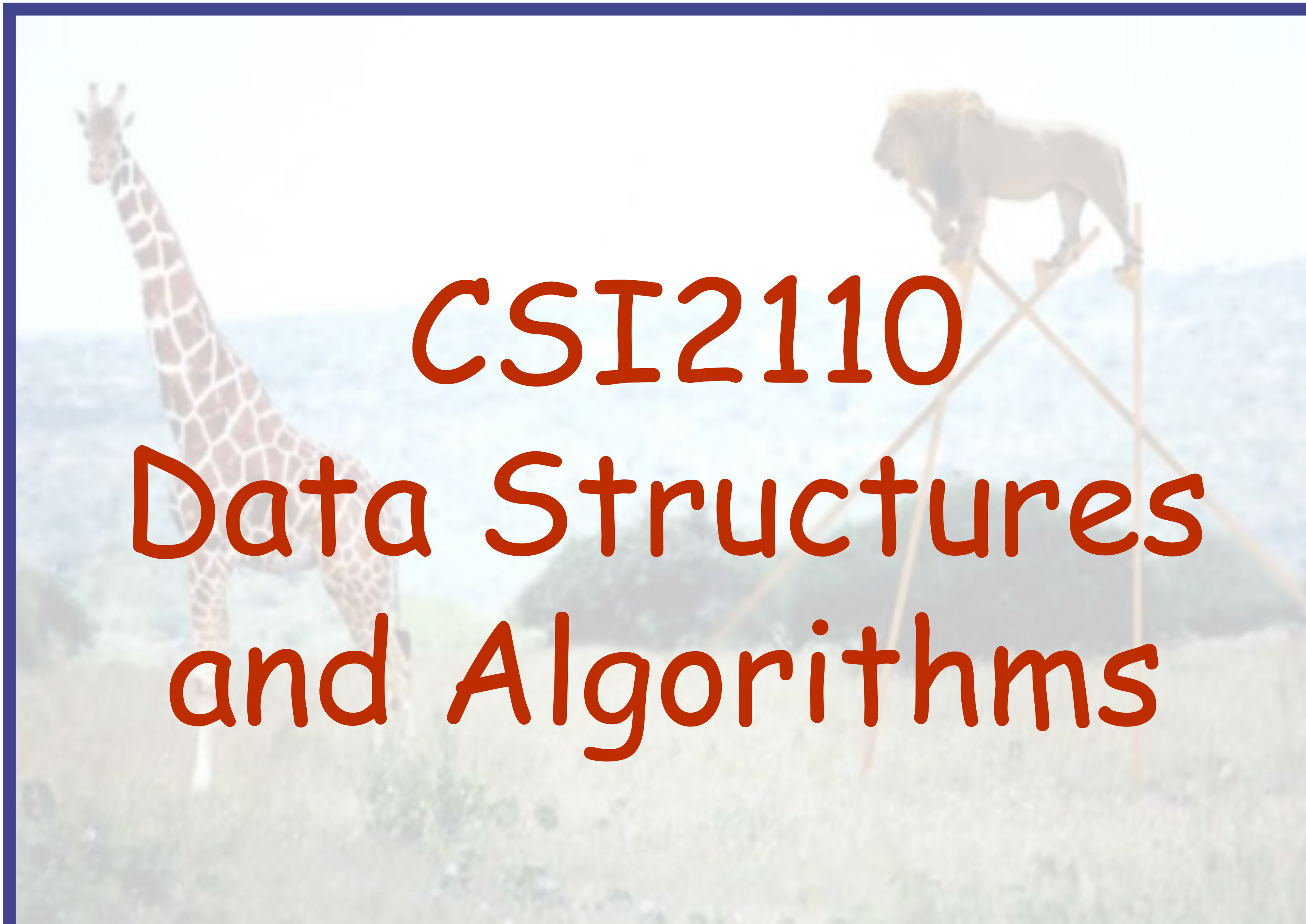
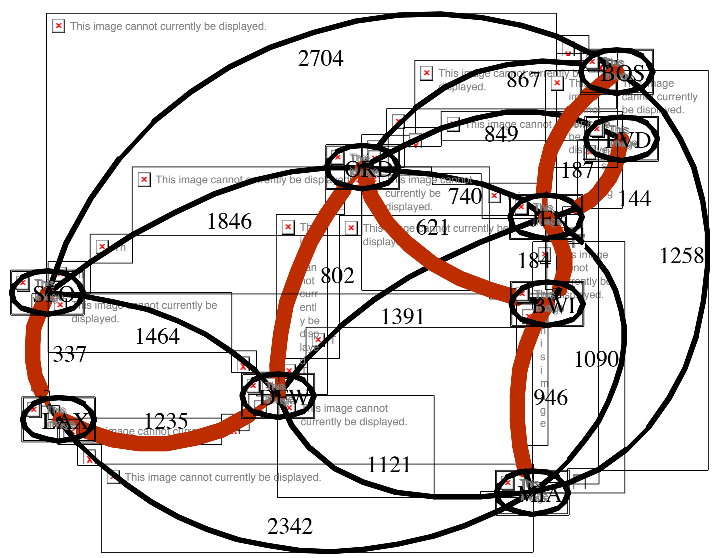


CSI2110 Data Structures and Algorithms

Minimum Spanning Tree

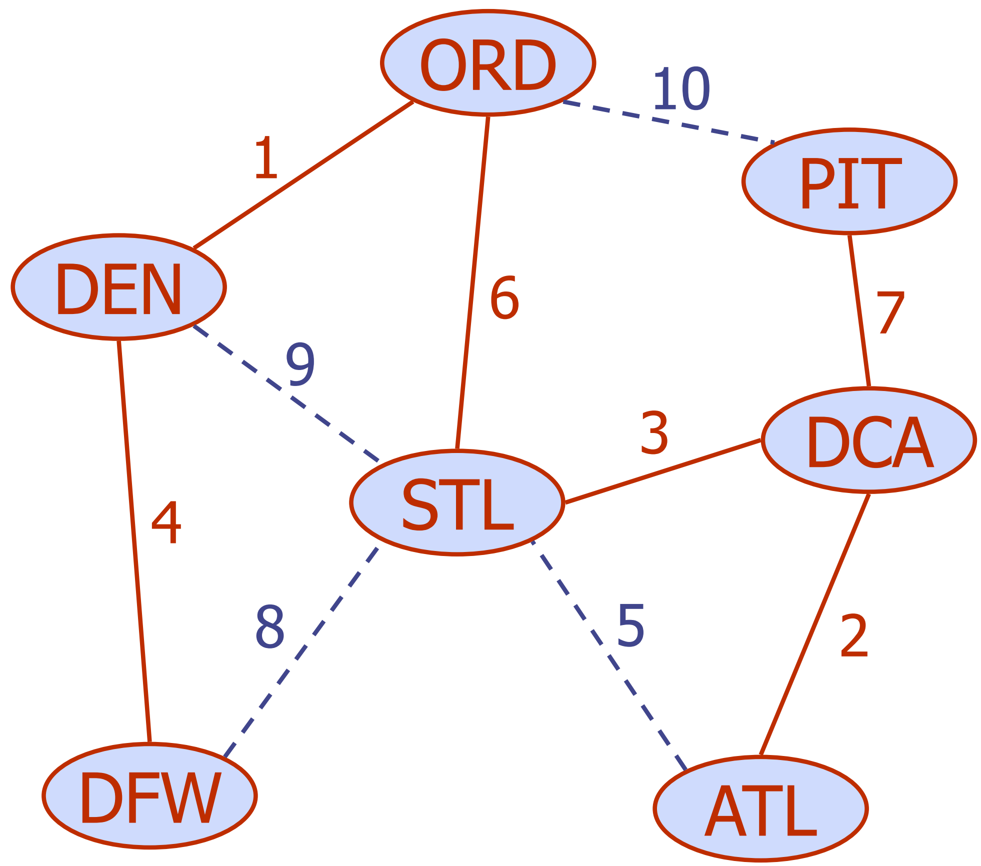


Outline and Reading

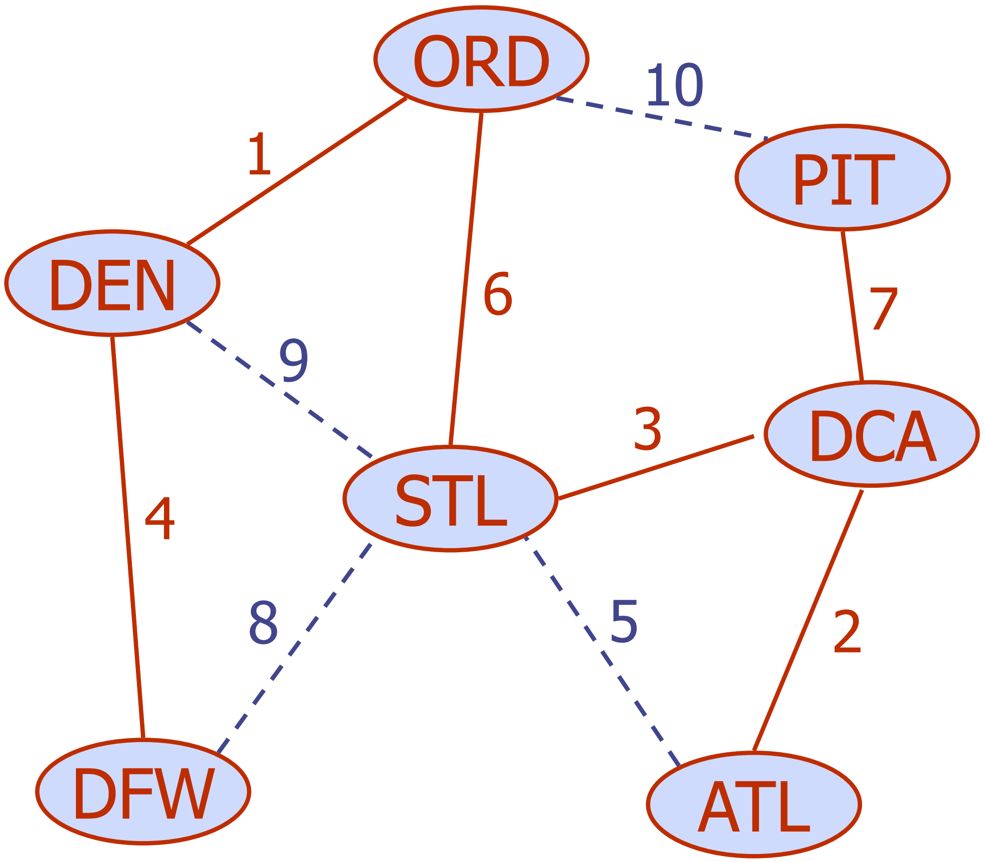
- Minimum Spanning Trees
  - Definitions
  - A crucial fact
- Prim-Jarnik's Algorithm
- Kruskal's Algorithm

Minimum Spanning Tree

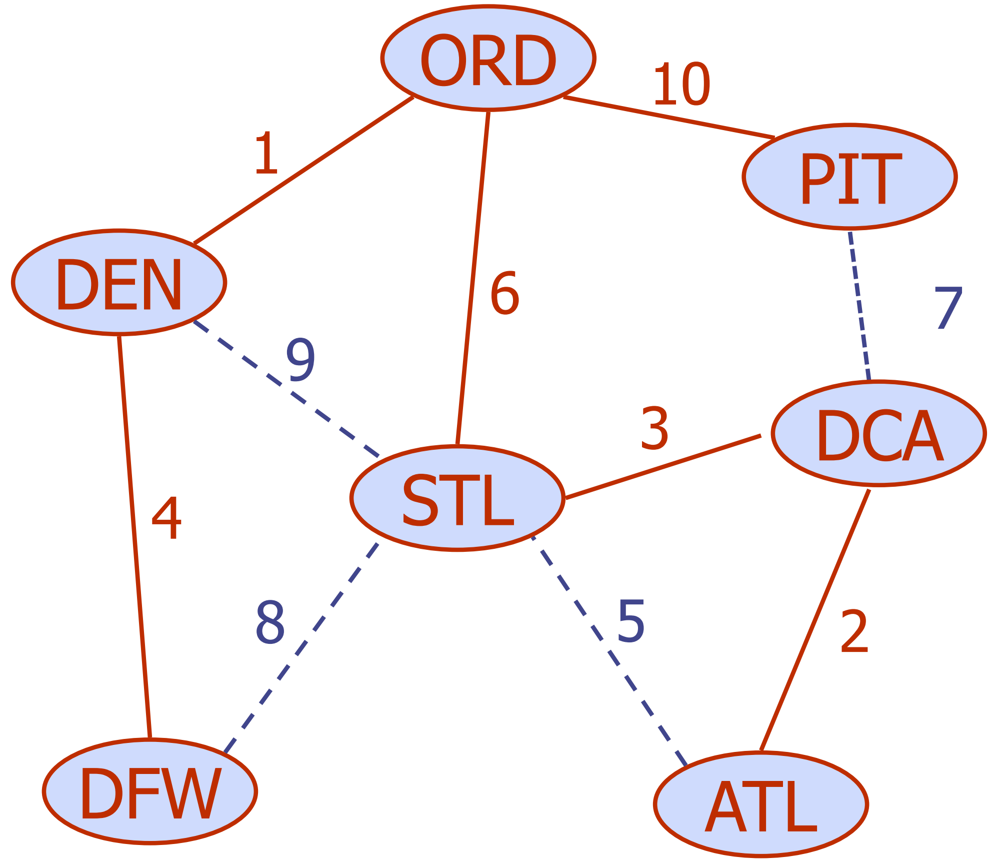
- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is itself a (free) tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation networks



Notice the difference



Minimum Spanning Tree



Shortest Path
Spanning Tree

Cycle Property

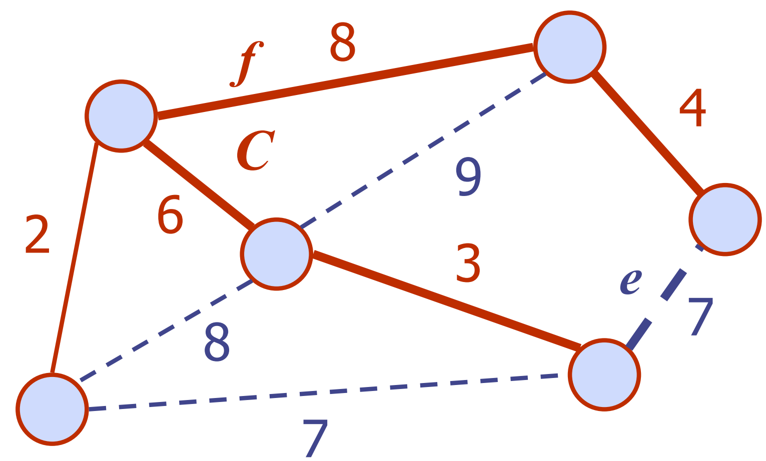
Cycle Property:

- 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 adding e to T
- For every edge f of C, $weight(f) \leq weight(e)$

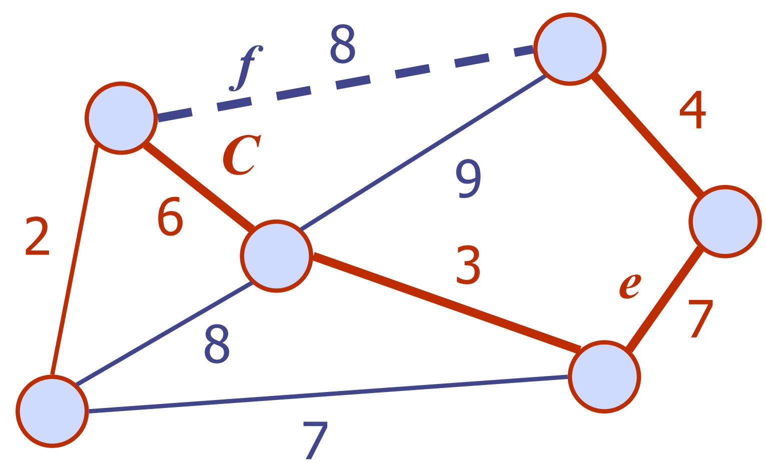
Proof:

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

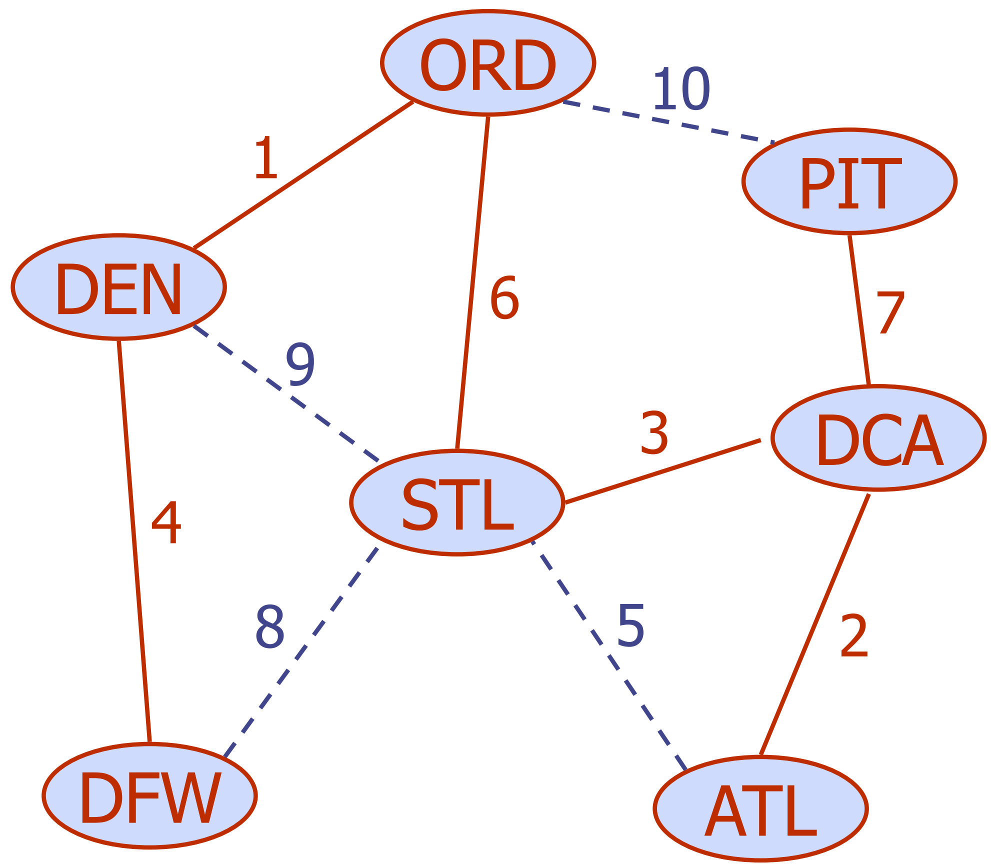


Cycle Property

In other words:

take a MST

in any cycle of the graph the non-spanning tree edge (dotted line) has max weight.



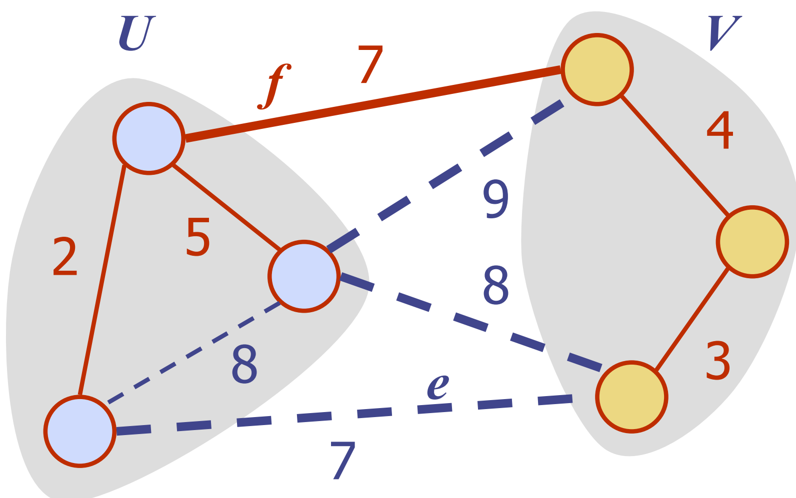
Partition Property

Partition Property:

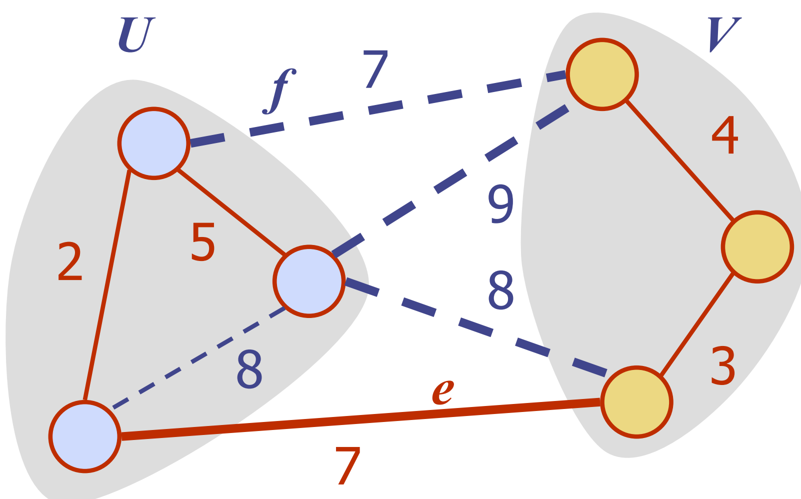
Consider a partition of the vertices of G into subsets U and V. Let e be an edge of minimum weight across the partition. There is a minimum spanning tree of G containing edge e

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



Replacing f with e yields another MST

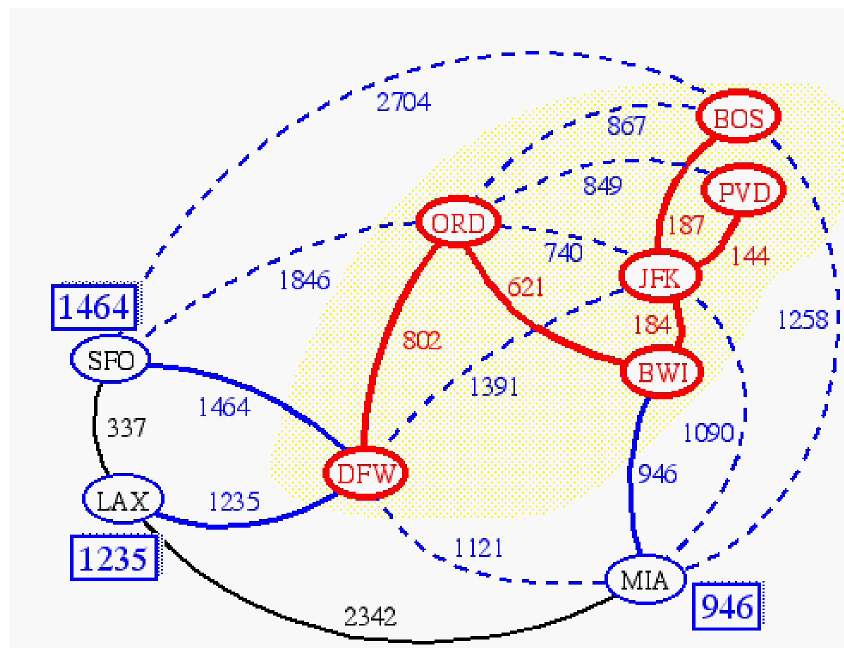


Prim-Jarnik's Algorithm

- Prim-Jarnik's algorithm for computing an MST is similar to Dijkstra's algorithm
- We assume that the graph is connected
- 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 any vertex in the cloud (as opposed to the total sum of edge weights on a path from the start vertex to u).

Prim-Jarnik's Algorithm

- 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



- Use a priority queue Q whose keys are D labels, and whose **elements are vertex-edge pairs**.
  - Key: distance
  - Element: vertex
- Any vertex v can be the starting vertex.
- We still initialize all the D[u] values to INFINITE, but we also **initialize E[u] (the edge associated with u) to null.**
- **Return** the minimum-spanning tree T.
- *We can reuse code from Dijkstra's, and we only have to change a few things. Let's look at the pseudocode....*

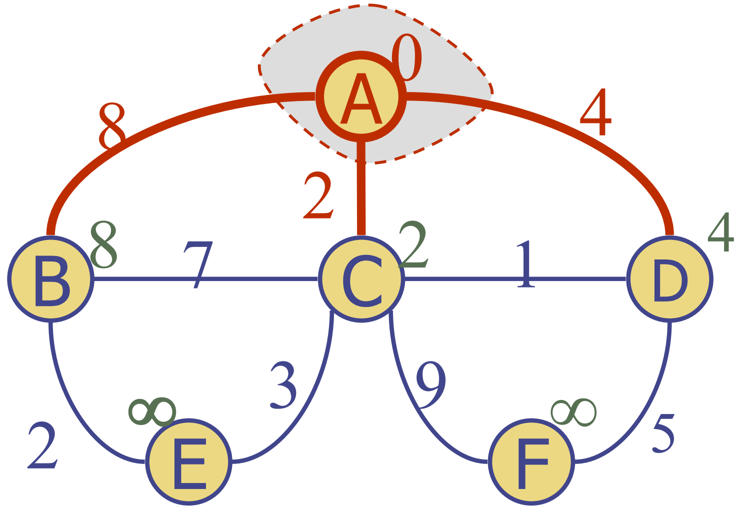
Algorithm PrimJarnik(G):

Input: A weighted graph G.

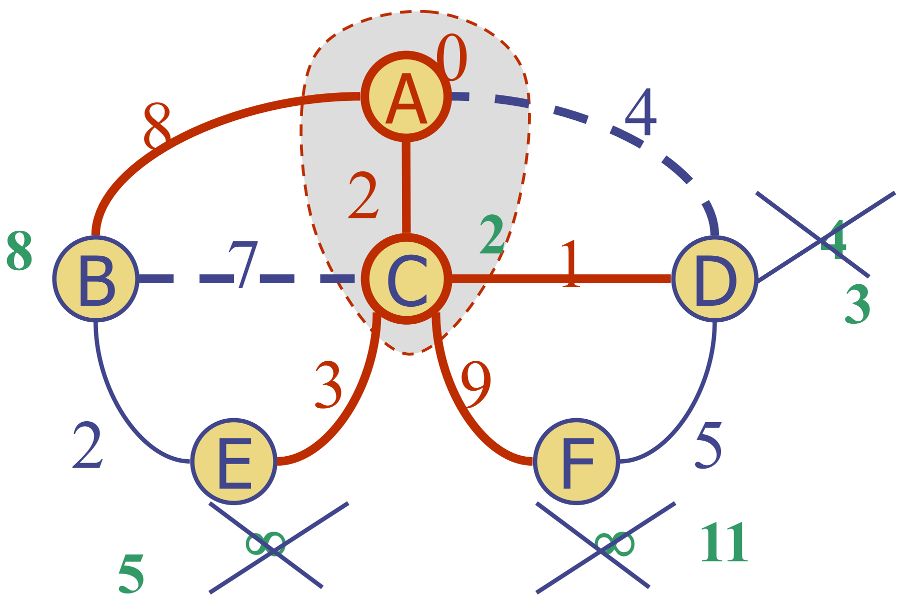
Output: A minimum spanning tree T for G.

```
pick any vertex v of G
{grow the tree starting with vertex v}
 T ← {v}
 D[u] ← 0
 E[u] ← ∅
 for each vertex u ≠ v do
   D[u] ← ∞
 let Q be a priority queue that contains vertices, using the D labels as keys
 while Q ≠ ∅ do  {pull u into the cloud C}
    u ← Q.removeMinElement()
    add vertex u and edge E[u] to T
   for each vertex z adjacent to u do  if z is in Q
    {perform the relaxation operation on edge (u, z) }
          if weight(u, z) < D[z] then
            D[z] ←weight(u, z)
            E[z] ← (u, z) change the key of z in Q to D[z]
 return tree T
```

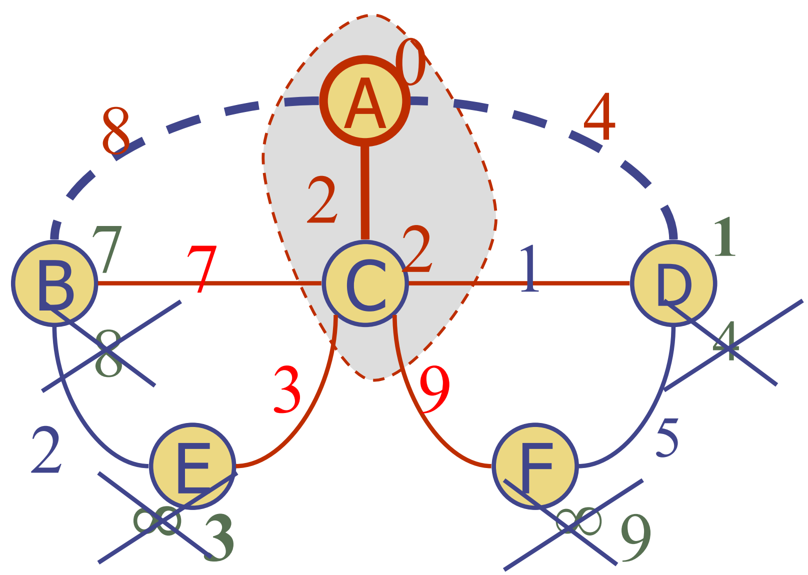
Notice
the difference

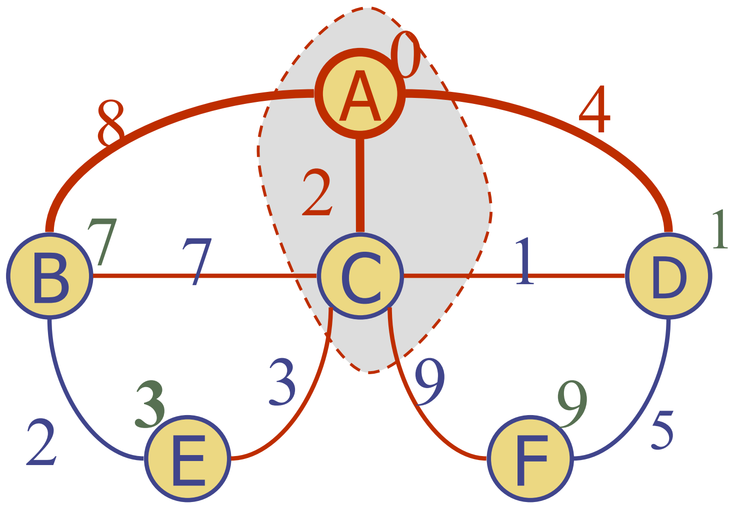


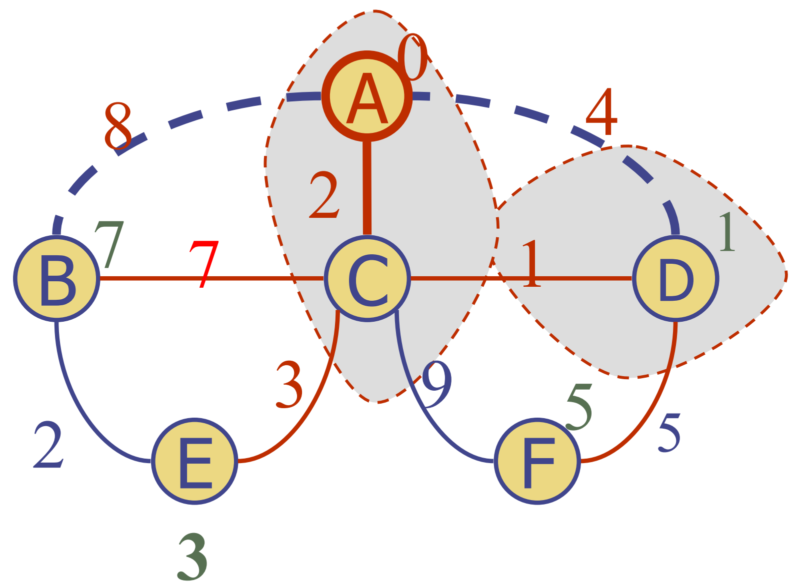
Djkstra
(Shortest Path Spanning Tree)

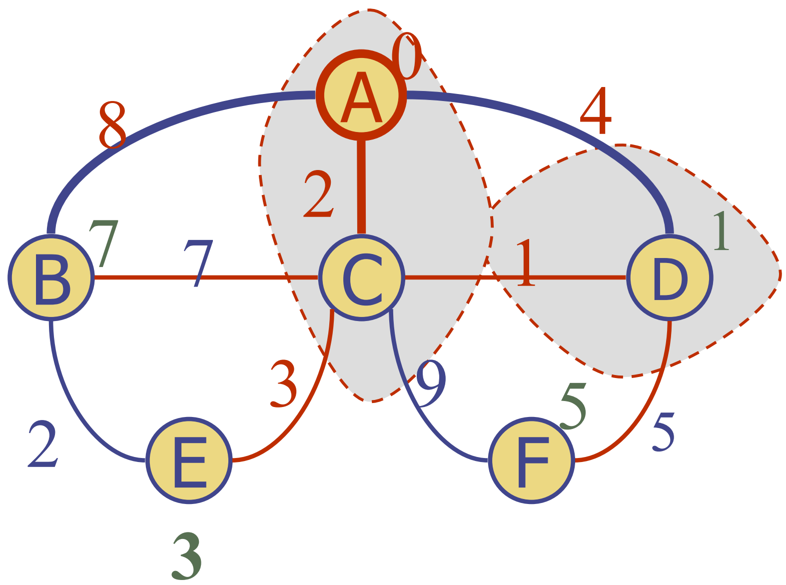


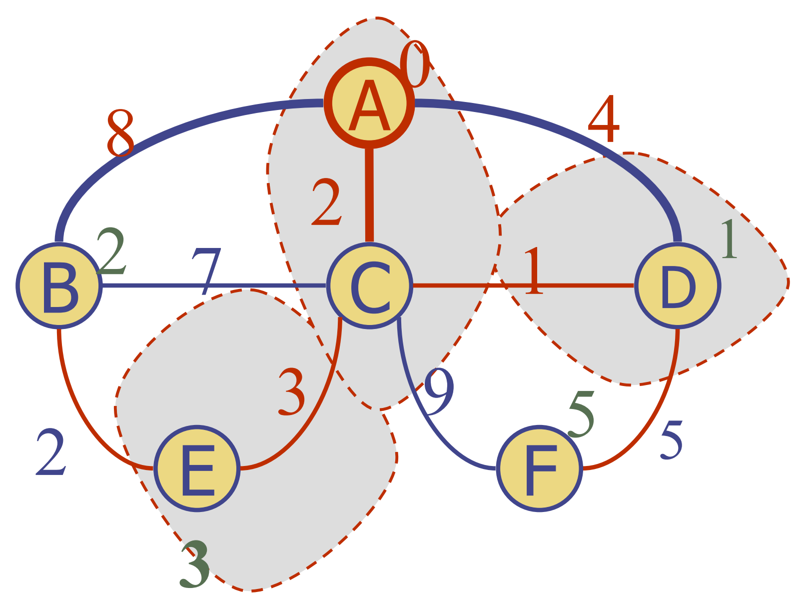
PRIM
(Minimum Spanning Tree)

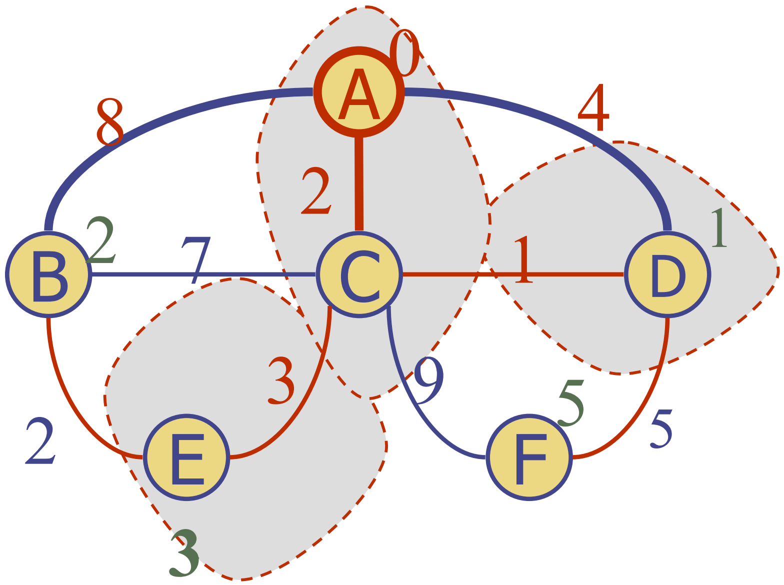


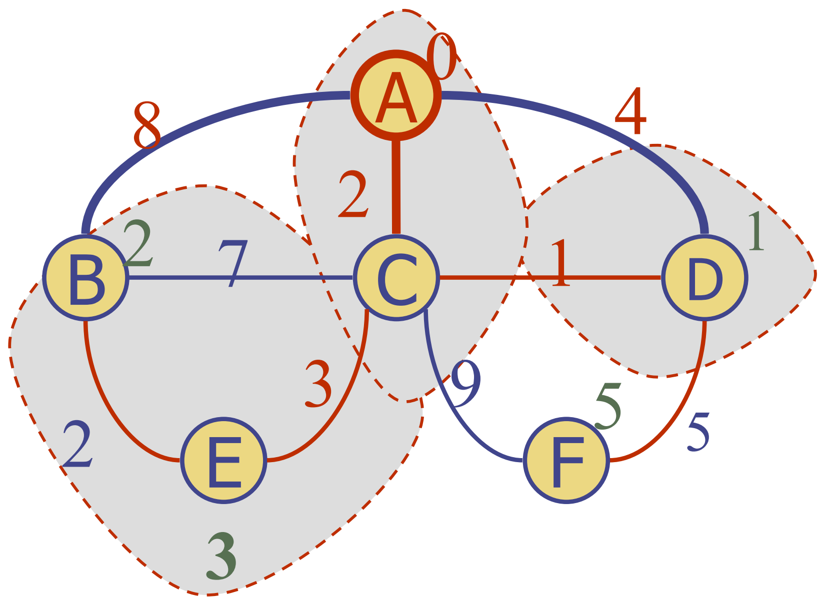

A
B
C
D
E
F
0
8
4
2
7
7
1
1
2
3
3
9
9
5

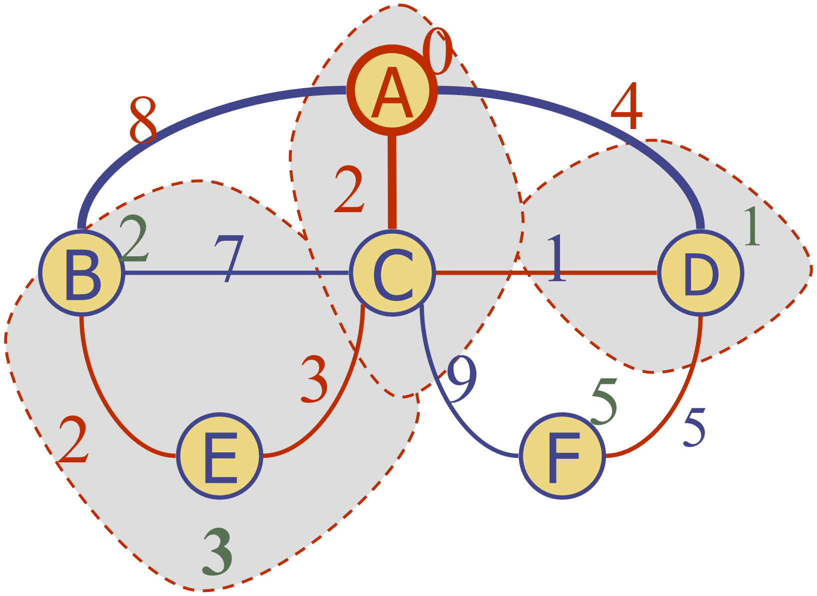

A
B
C
D
E
F
0
8
4
2
7
7
1
1
2
3
9
5
5
3

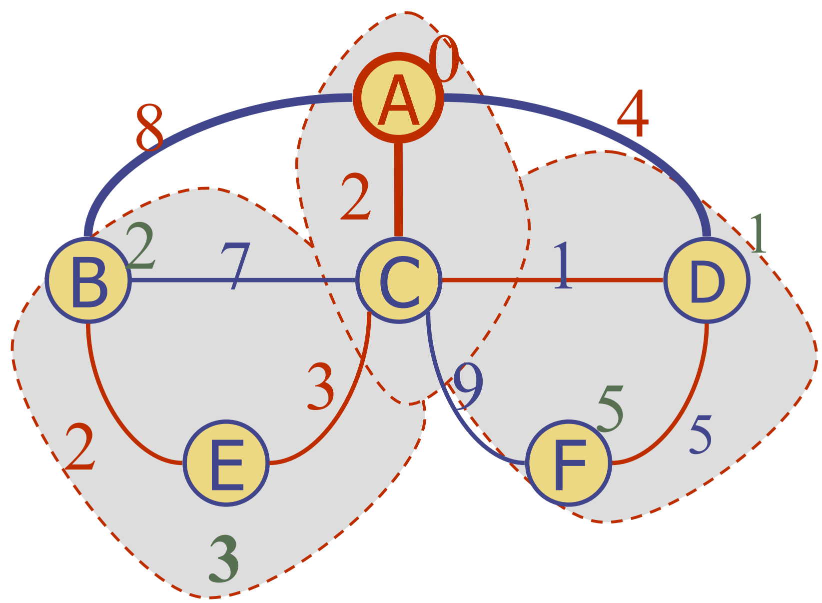

A
0
8
4
2
B
7
7
C
1
D
1
3
9
5
2
E
F
5
3


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

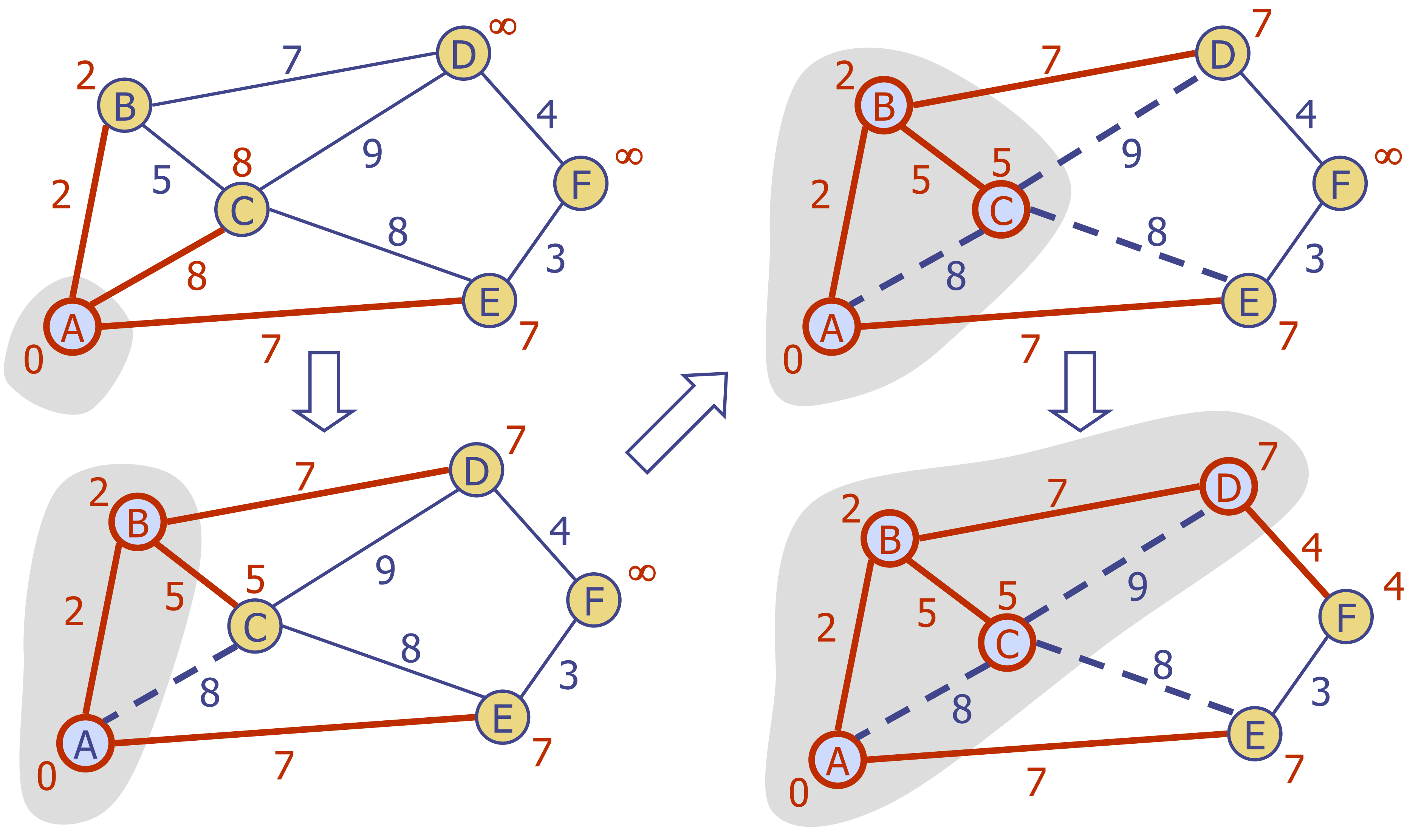

A
0
8
4
2
B
2
7
C
1
D
1
3
9
5
2
E
F
5
3


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

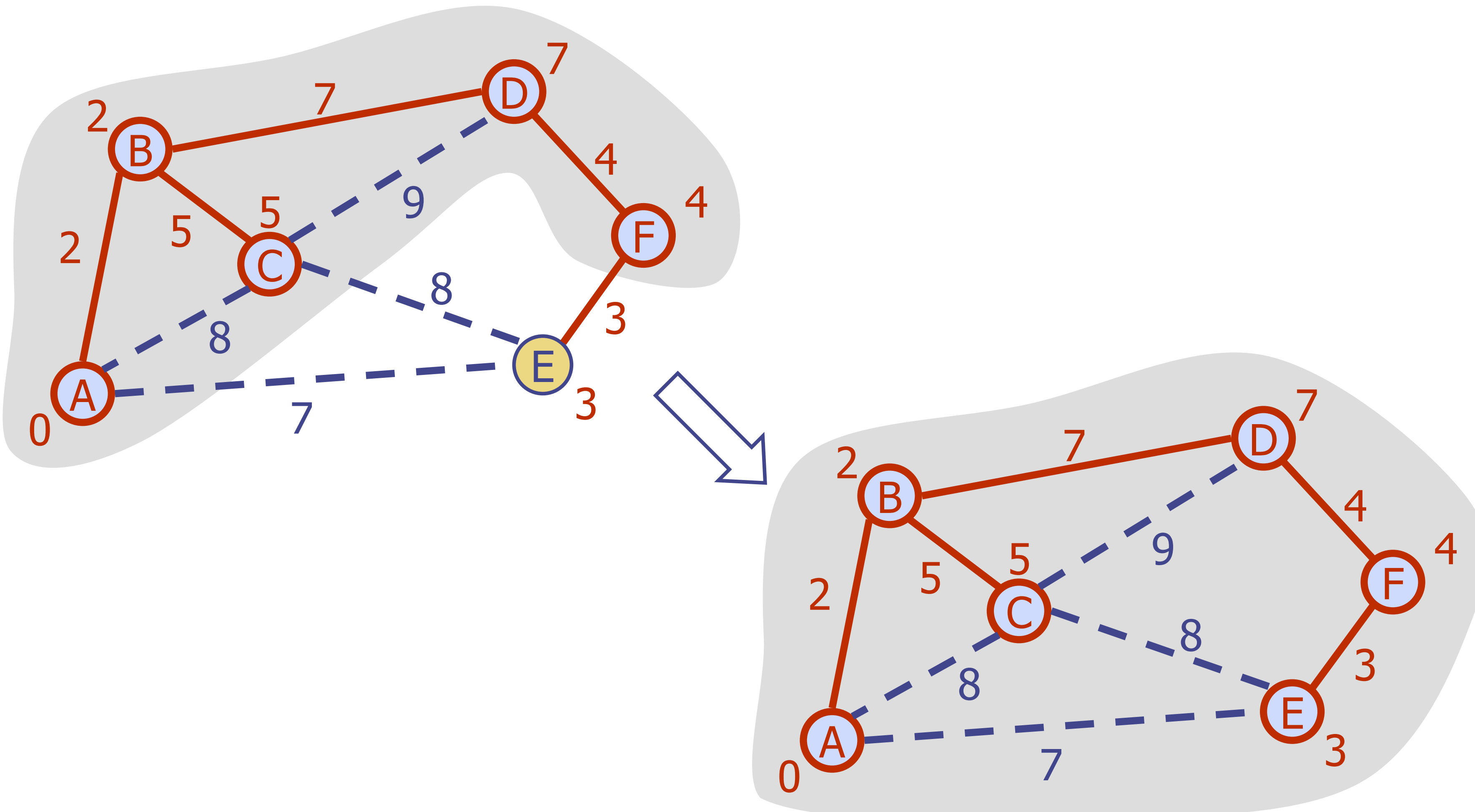

A
0
8
4
2
B
2
7
C
1
D
1
3
9
5
2
E
F
5
3


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

Example



Example (contd.)



Prim-Jarnik… Why It Works

- This is an application of the Cycle Property!
- Let the minimum edge at some iteration be (u,v). If there is an MST not containing (u,v), then (u,v) completes a cycle. Since (u,v) was considered before some other edge connecting v to the cluster, it must have weight equal to or lower than that other edge. A new MST can be formed by swapping.

Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the 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

Dijkstra vs. Prim-Jarnik

```
Algorithm DijkstraShortestPaths(G, s)
    Q ← new heap-based priority queue

    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        setParent(v, ∅)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
                setParent(z,e)
                Q.replaceKey(getLocator(z),r)
```

```
Algorithm PrimJarnikMST(G)
    Q ← new heap-based priority queue
    s ← a vertex of G
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        setParent(v, ∅)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
            z ← G.opposite(u,e)
            r ← weight(e)
            if r < getDistance(z)
                setDistance(z,r)
                setParent(z,e)
                Q.replaceKey(getLocator(z),r)
```

Kruskal's Algorithm

Kruskal's Algorithm

- Each vertex is initially stored as its own cluster.
- At each iteration, the minimum weight edge is added to the spanning tree if it joins 2 distinct clusters.
- The algorithm ends when all the vertices are in the same cluster.

Kruskal's Algorithm… Why It Works

- This is an application of the Partition Property!
- If the minimum edge at some iteration is (u,v), then if we consider a partition of G with u in one cluster and v in the other, then the partition property says that there must be an MST containing (u,v).

Kruskal's Algorithm

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree *T* which is our MST

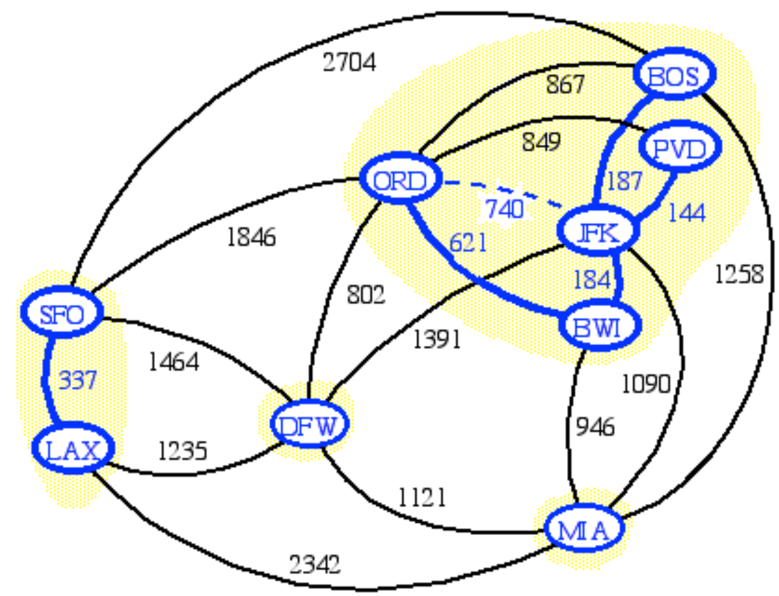
Algorithm ***KruskalMST(G)***
    for each vertex *V* in ***G*** **do**
        define a ***Cloud(v)*** of $\leftarrow$ {*v*}
    let ***Q*** be a priority queue.
    Insert all edges into ***Q*** using their weights as the key
    T $\leftarrow \varnothing$
    while ***T*** has fewer than ***n***-1 edges **do**
    edge ***e*** = ***Q.removeMin()***
        Let ***u***, ***v*** be the endpoints of *e*
        if ***Cloud(v)*** $\neq$ ***Cloud(u)*** **then**
            Add edge ***e*** to ***T***
            Merge ***Cloud(v)*** and ***Cloud(u)***
    return ***T***

Data Structure for Kruskal Algortihm

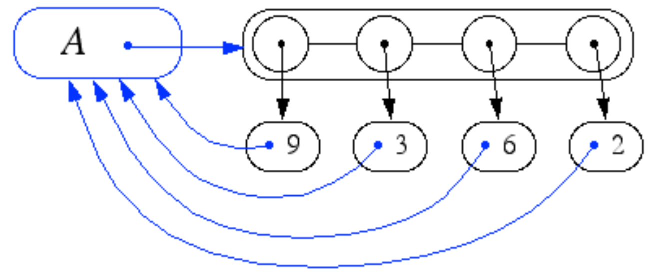
- The algorithm maintains a forest of trees
- An edge is accepted it if connects distinct trees
- We need a data structure that maintains a **partition**, i.e., a collection of disjoint sets, with the operations:

  -**find**(u): return the set storing u

  -**union**(u,v): replace the sets storing u and v with their union



Representation of a 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(u,v), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation union(u,v) is $\min(n_u,n_v)$, where n_u and n_v are the sizes of the sets storing u and v
- 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

- A oretion-based version of Kruskal's Algorithm performs cloud merges as unions and tests as finds.

Algorithm **Kruskal(*G*)**:

Input: A weighted graph ***G***.

Output: a set of edges T of a MST for ***G***, if ***G*** is connected;

a set of edges T of a MS forest ***T*** for ***G***, if ***G*** is not connected;

Let ***P*** be a partition of the vertices of ***G***, where each vertex forms a separate set.

Let ***Q*** be a priority queue storing the edges of ***G***, organized by their weights

Let ***T*** be an initially-empty set of edges

while (***Q*** is not empty) and (T.size() < n-1) **do**

(***u***,***v***) ← ***Q***.removeMinElement()

if ***P***.find(***u***) != ***P***.find(***v***) **then**

Add (***u***,***v***) to ***T***

P.union(***u***,***v***)

return ***T***

Running time:

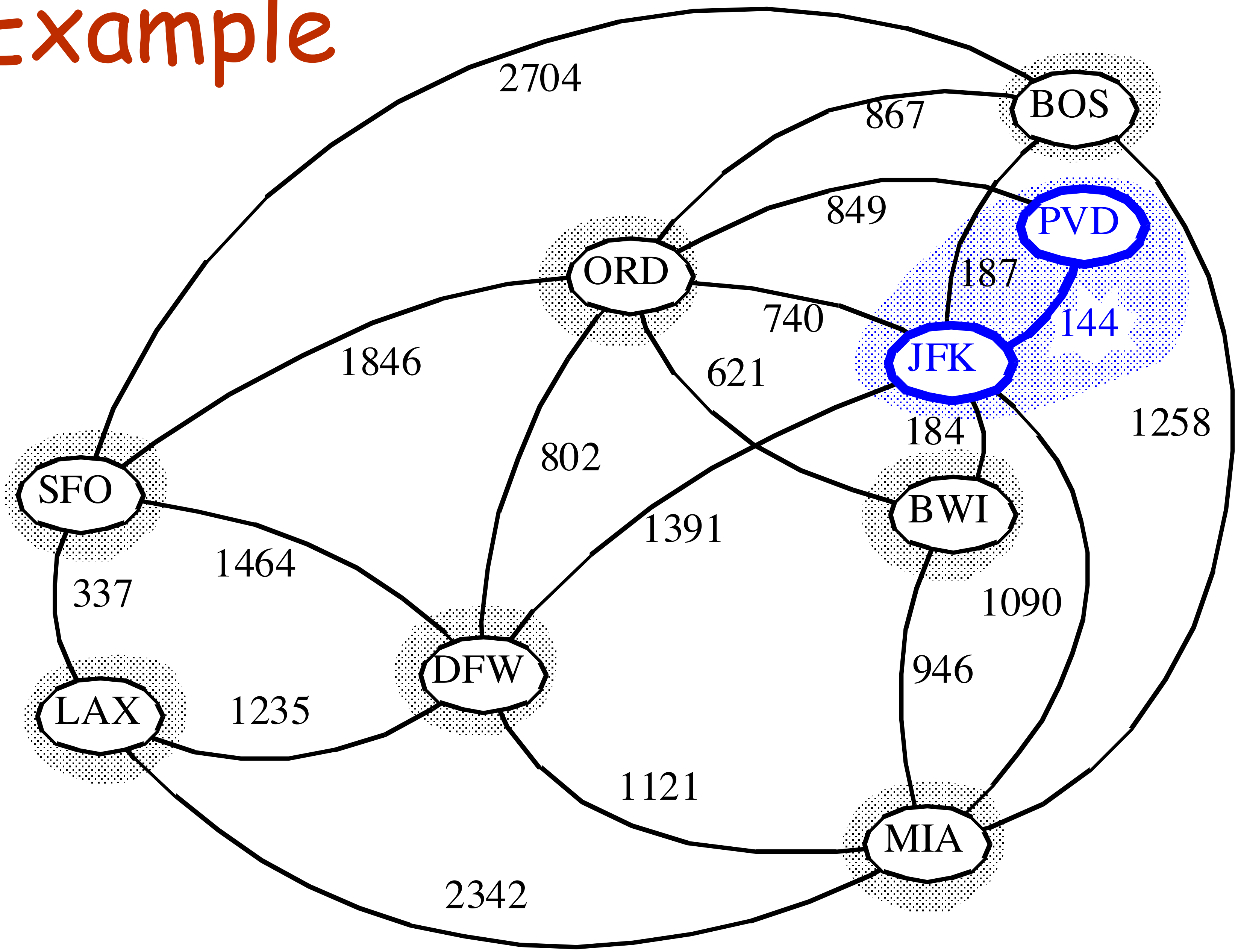
O((n+m)log n)

More about Kruskal's running time

- Building a priority queue Q with m elements takes O(m log m) or O(m) using buildheap.
- Operation ***Q***.removeMinElement() takes time O(log m) and is repeated up to m times for a total of O(m log m)
- Operation ***P***.find(x) takes O(1) and is repeated at most 2m times.
- Operation ***P***.union(***u***,***v***) takes time proportional to the smallest of the two parts and is repeated at most n-1 times. As discussed before each of the n elements gets moved to another list at most log n times over all unions. So the total cost for all operations ***P***.union(***u***,***v***) is O(n log n).

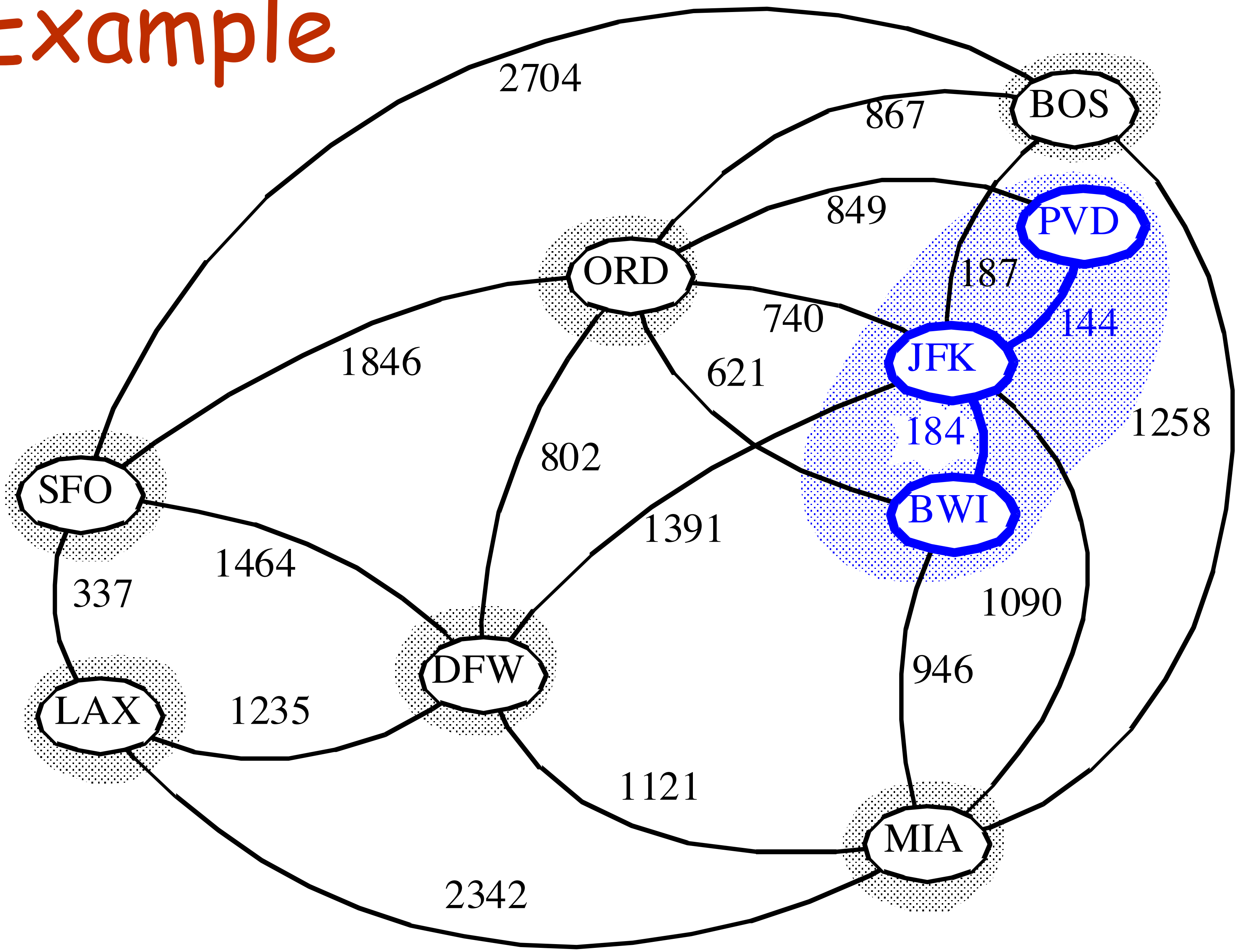
In summary, all above considered we get O(n log n + m log m); but since m=O(n^2), O(log m) is O(2 log n)=O(log n). So the running time can be written as O((m+n) log n).

Example



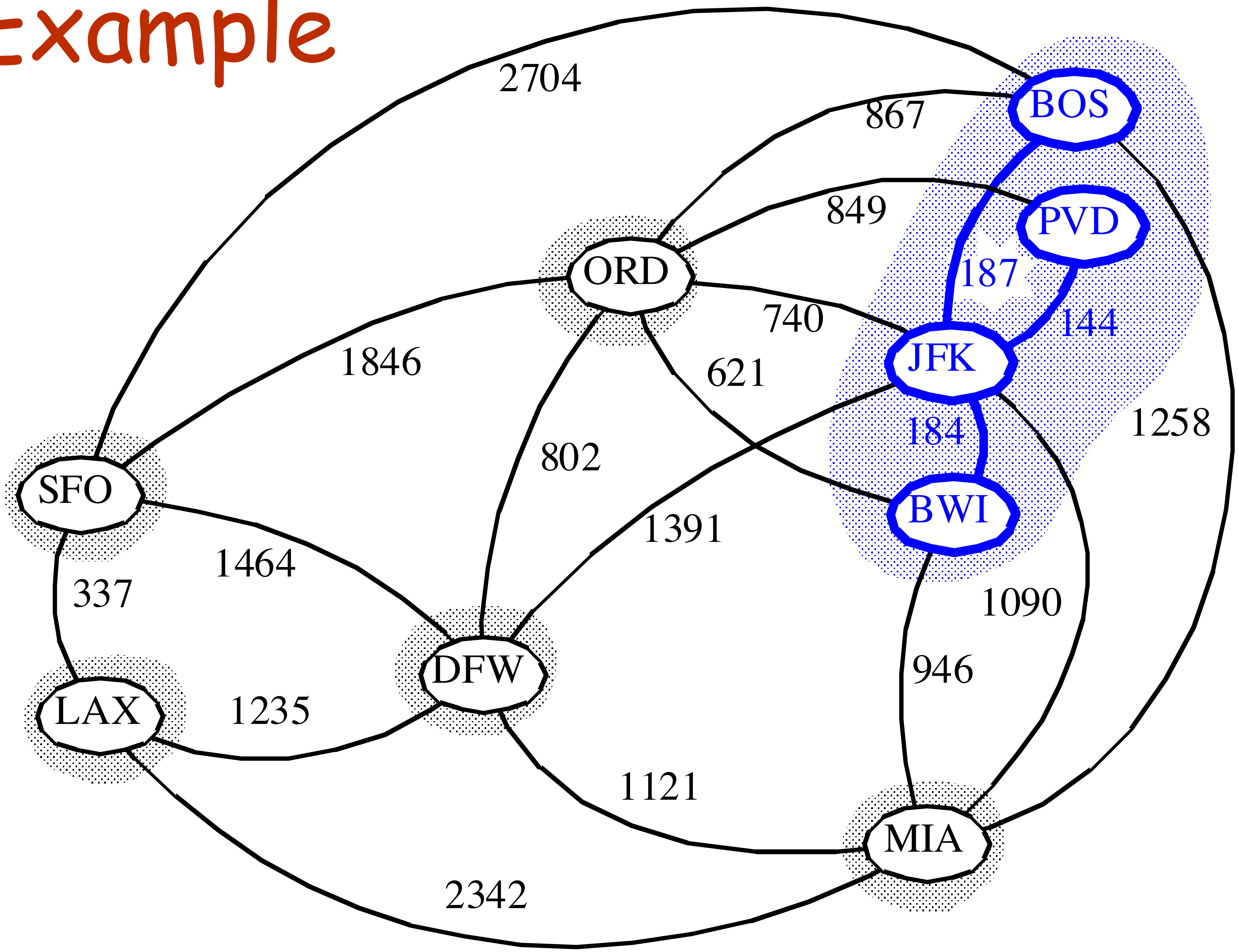
2704
BOS
867
849
PVD
ORD
187
740
144
JFK
1846
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



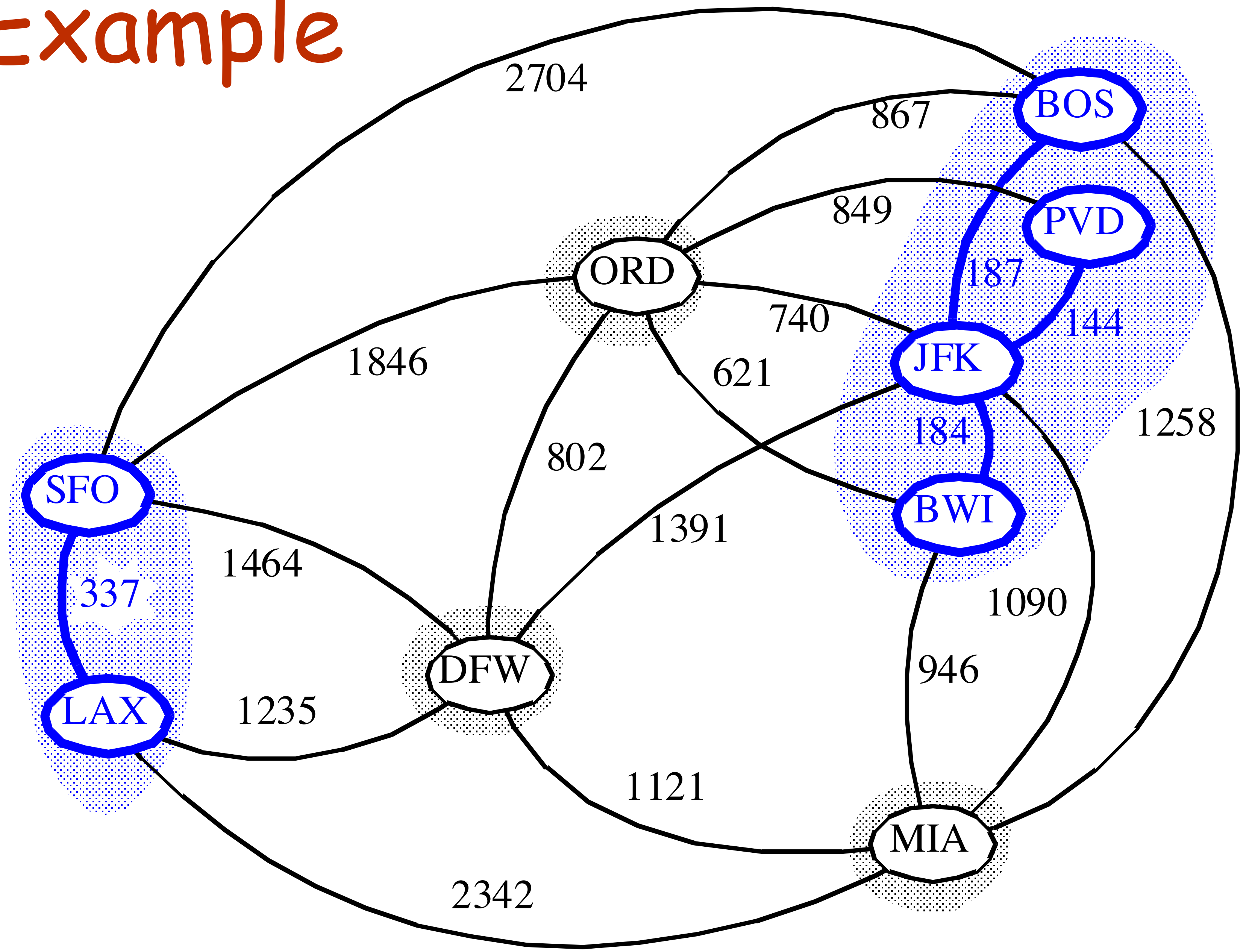
BOS
PVD
ORD
JFK
BWI
SFO
LAX
DFW
MIA
2704
867
849
187
144
740
621
1846
184
1258
802
1391
1464
337
1090
946
1235
1121
2342

Example



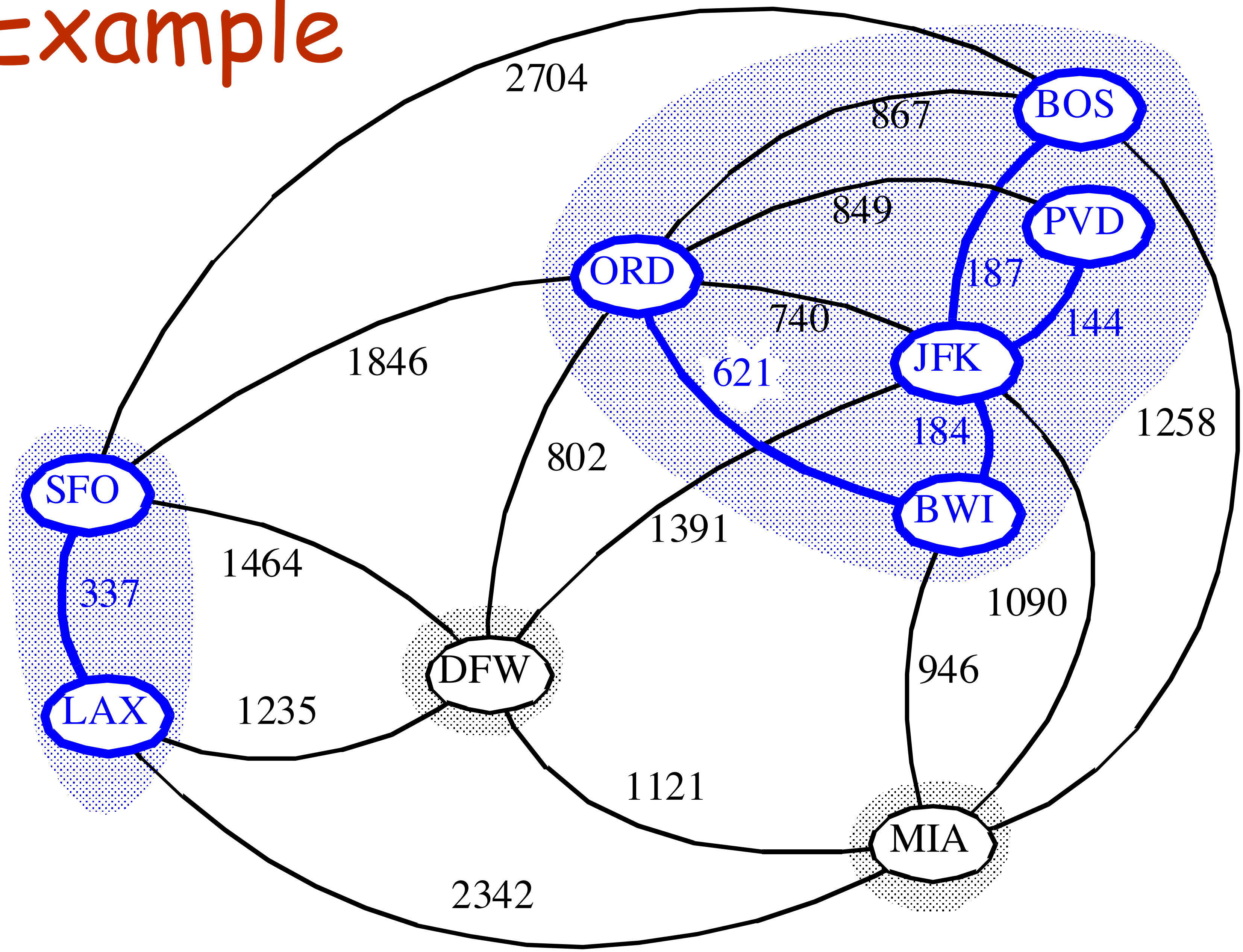
2704
867
BOS
849
PVD
187
ORD
144
740
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

Example



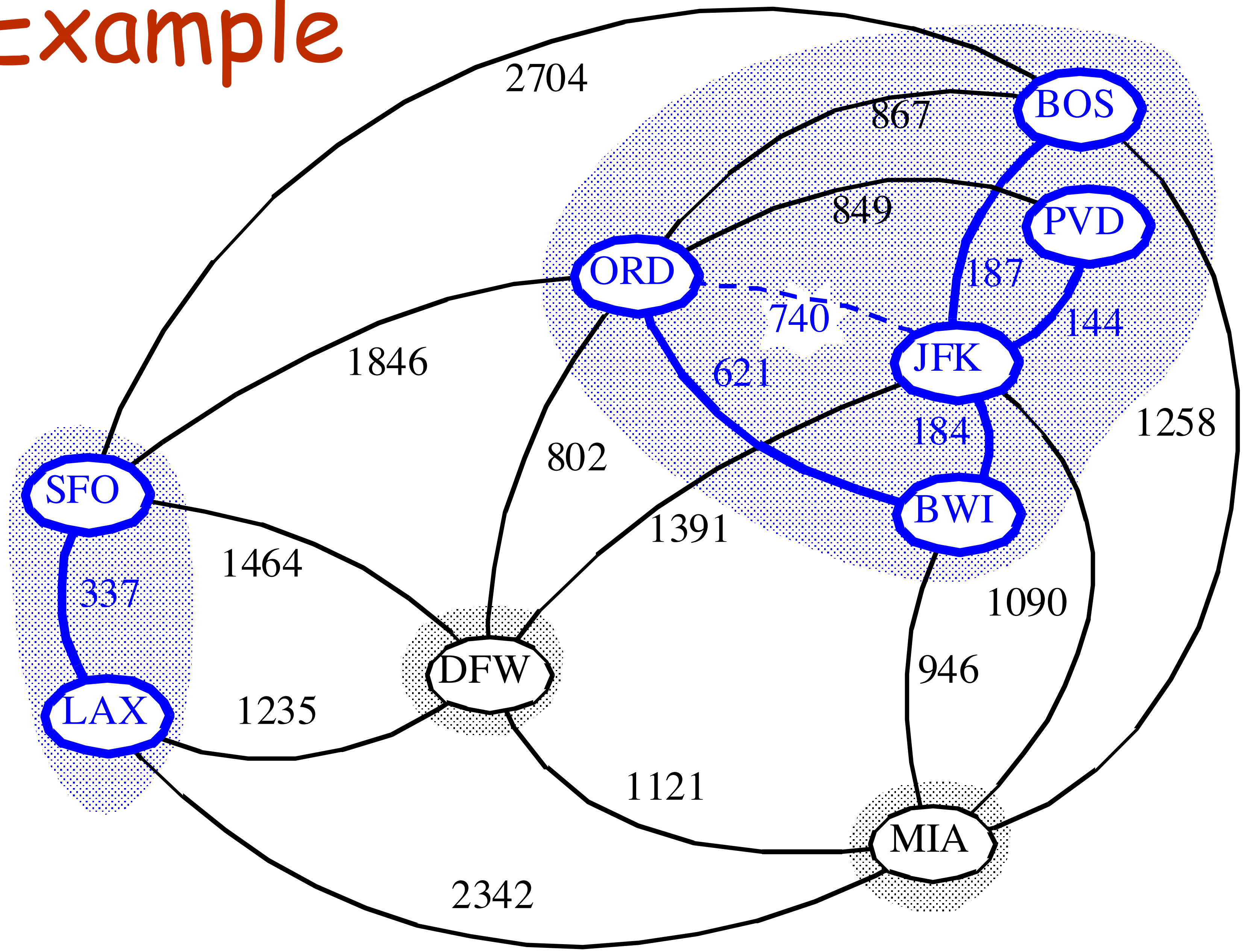
2704
BOS
867
PVD
849
ORD
187
144
740
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



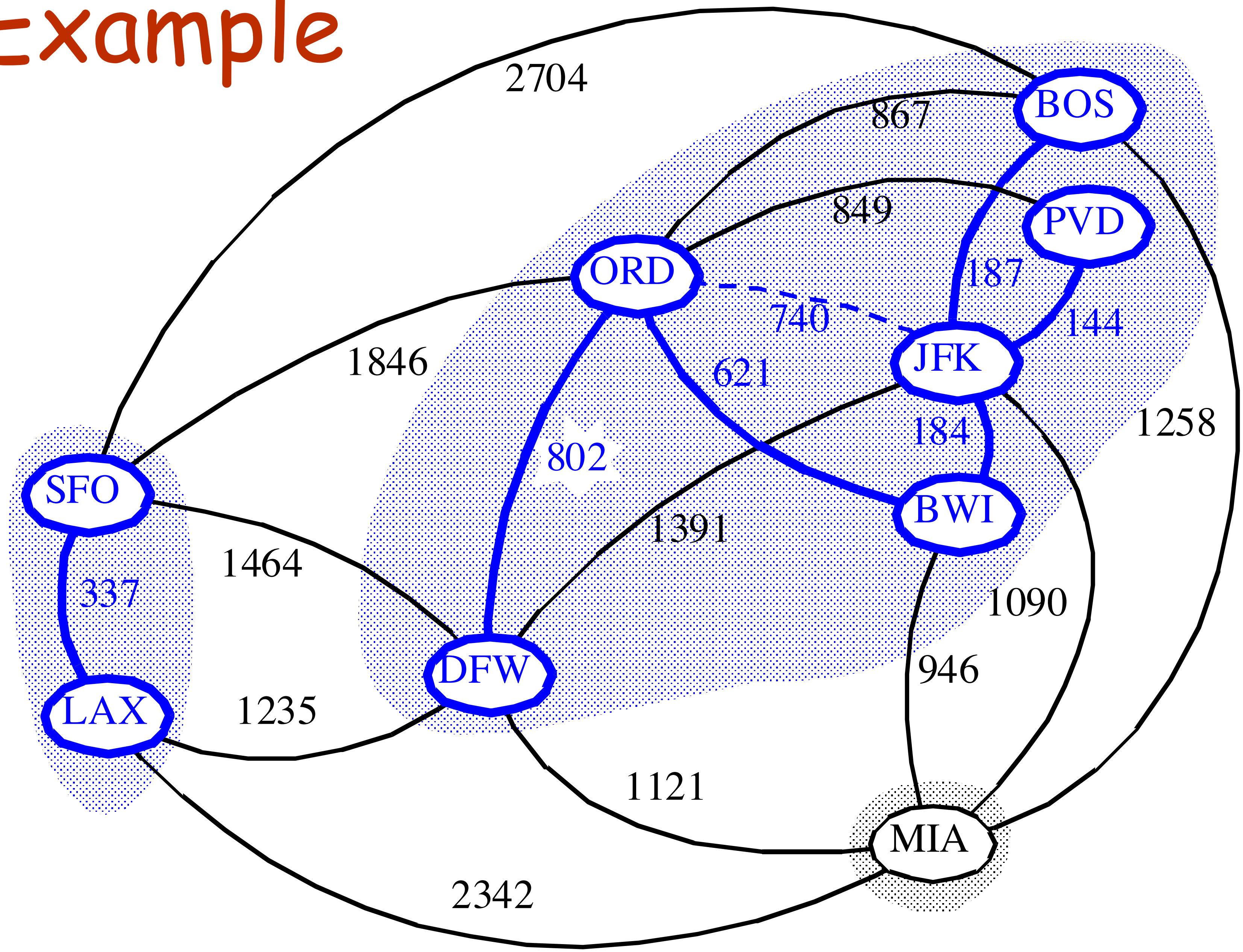
2704
BOS
867
849
PVD
ORD
187
144
740
JFK
1846
621
184
1258
SFO
802
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



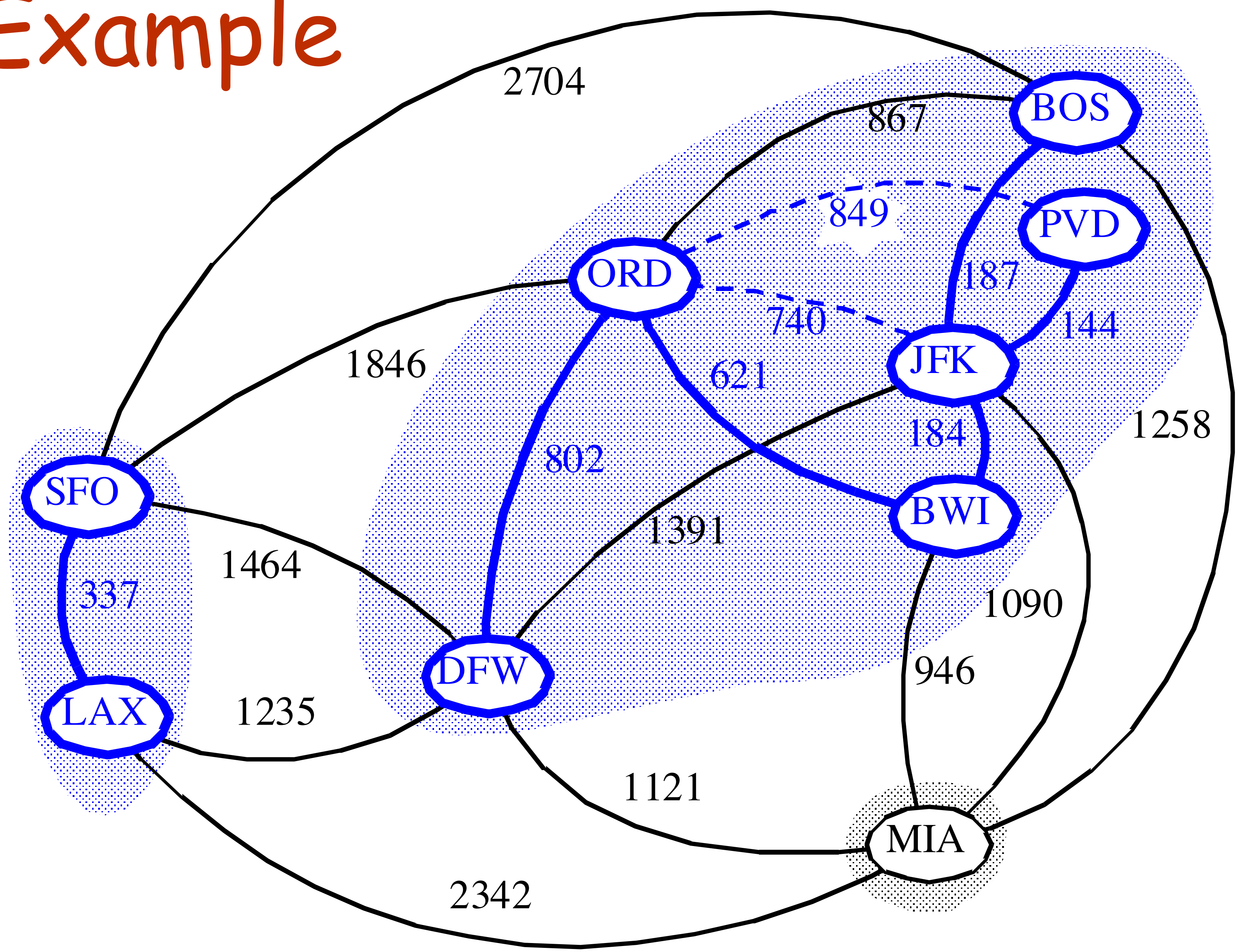
2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example

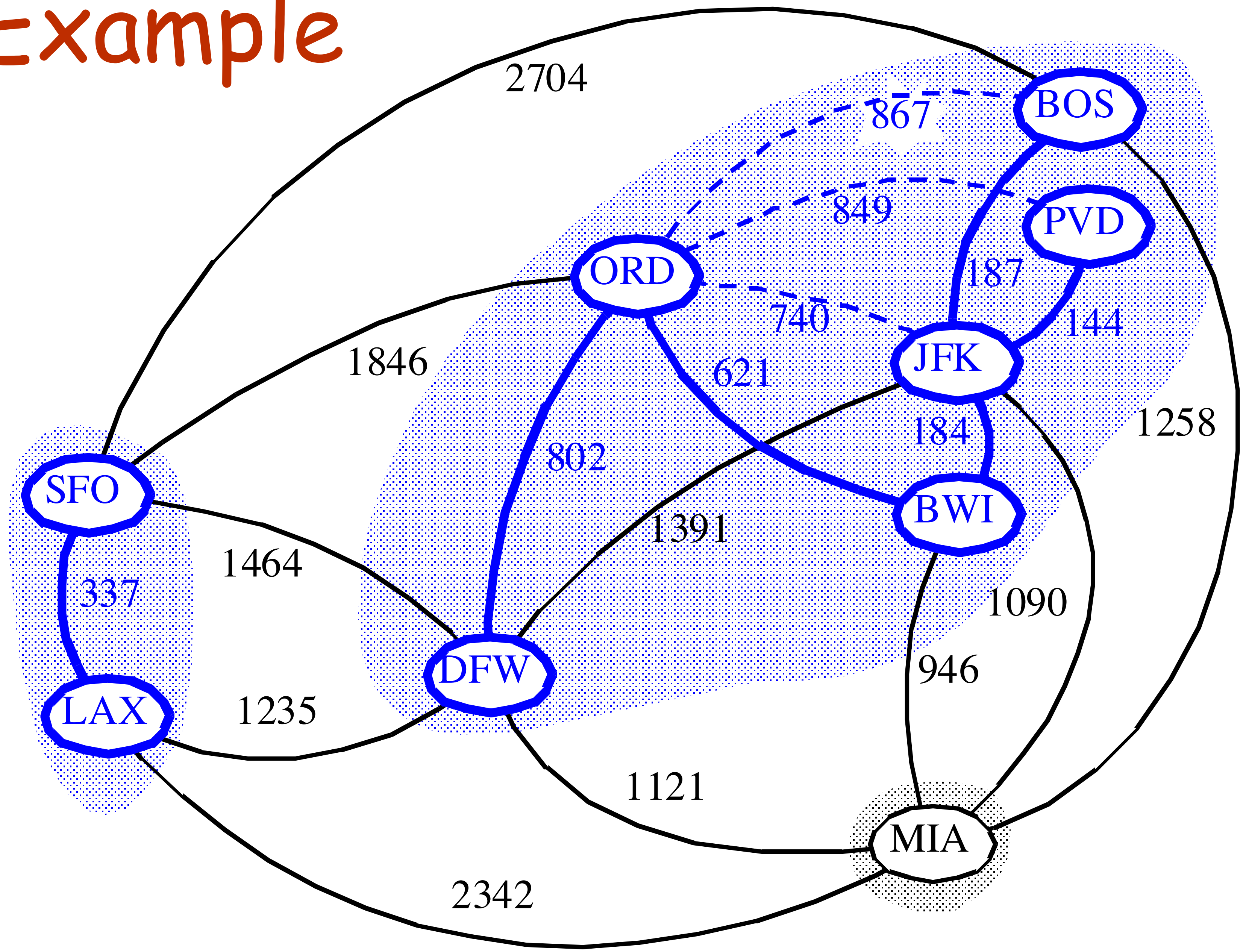


2704
BOS
867
PVD
849
ORD
187
740
144
JFK
1846
621
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

Example

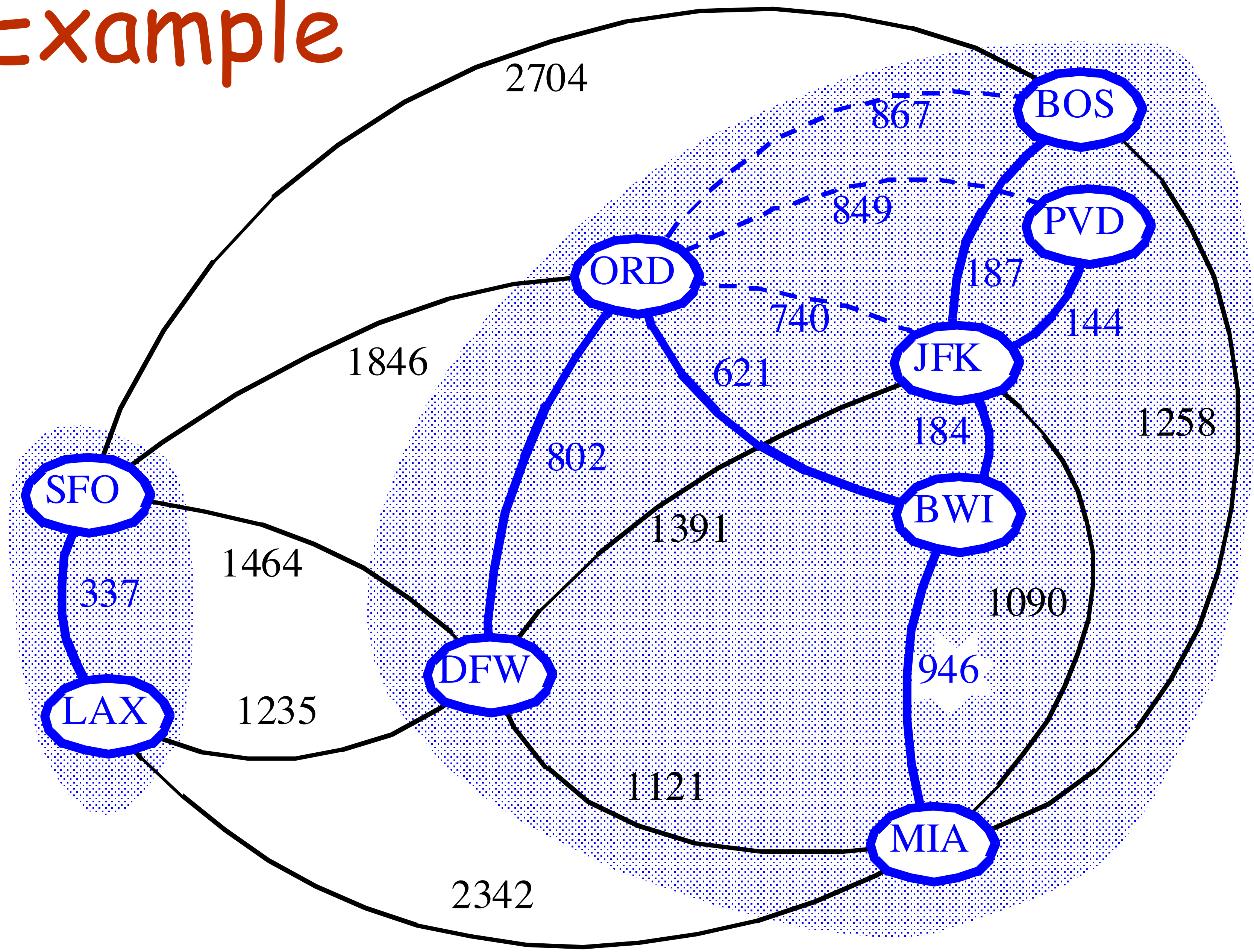

2704
867
BOS
849
PVD
ORD
187
740
144
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



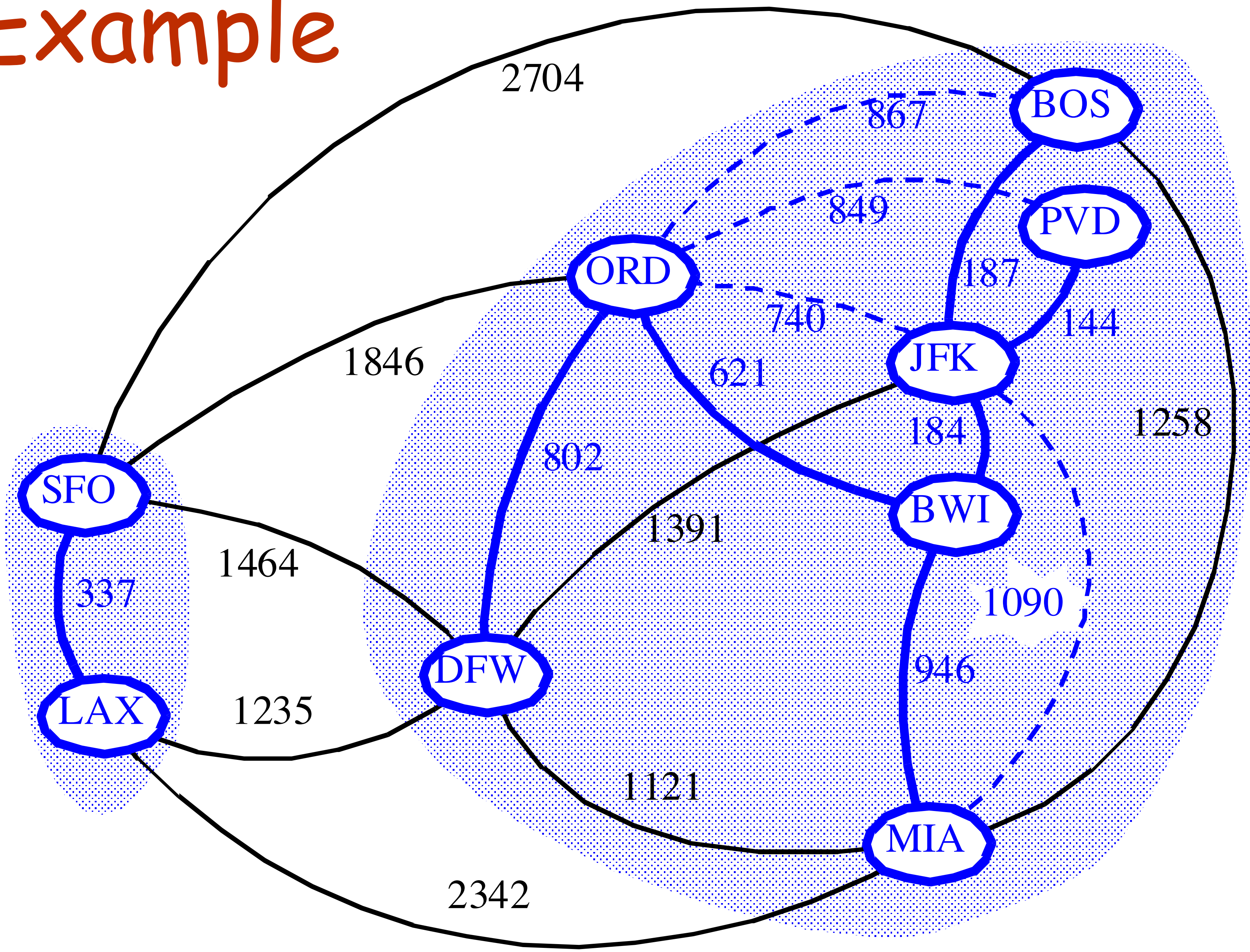
2704
867
BOS
849
PVD
ORD
187
1846
740
144
JFK
621
1258
184
SFO
802
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

Example



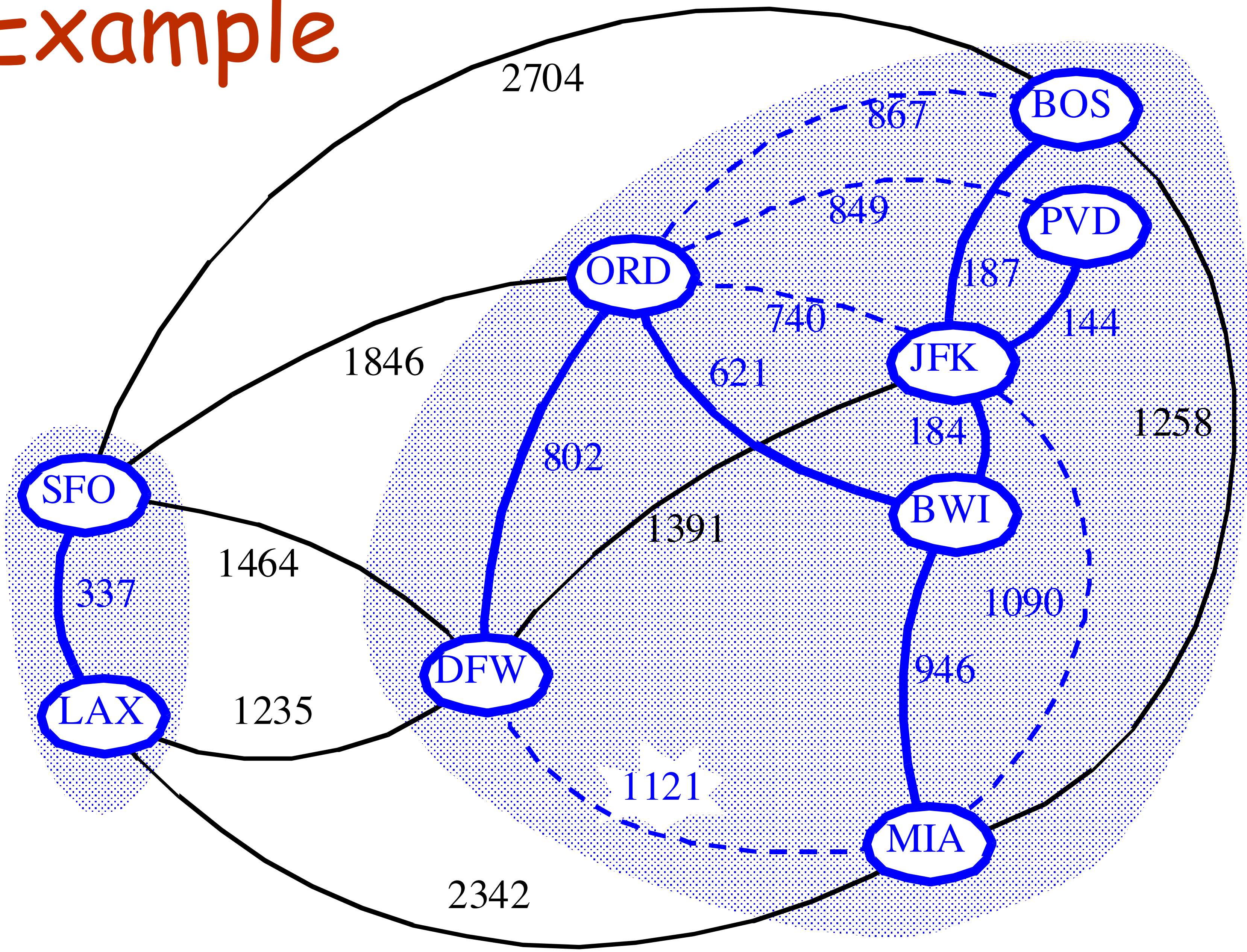
2704
867
BOS
849
PVD
ORD
187
740
144
1846
JFK
621
1258
184
SFO
802
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



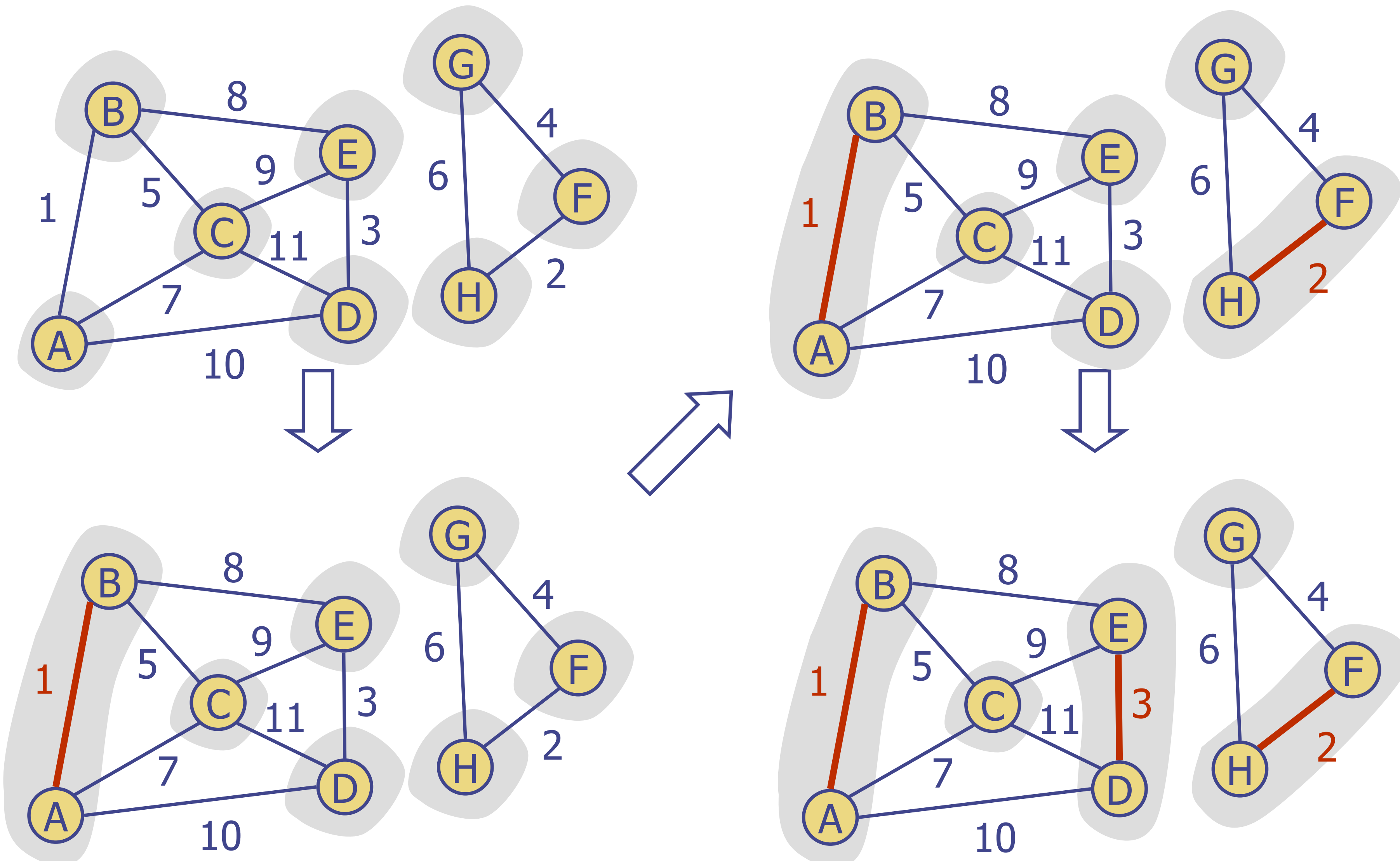
2704
BOS
867
849
PVD
ORD
187
740
144
1846
JFK
621
1258
184
SFO
802
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example

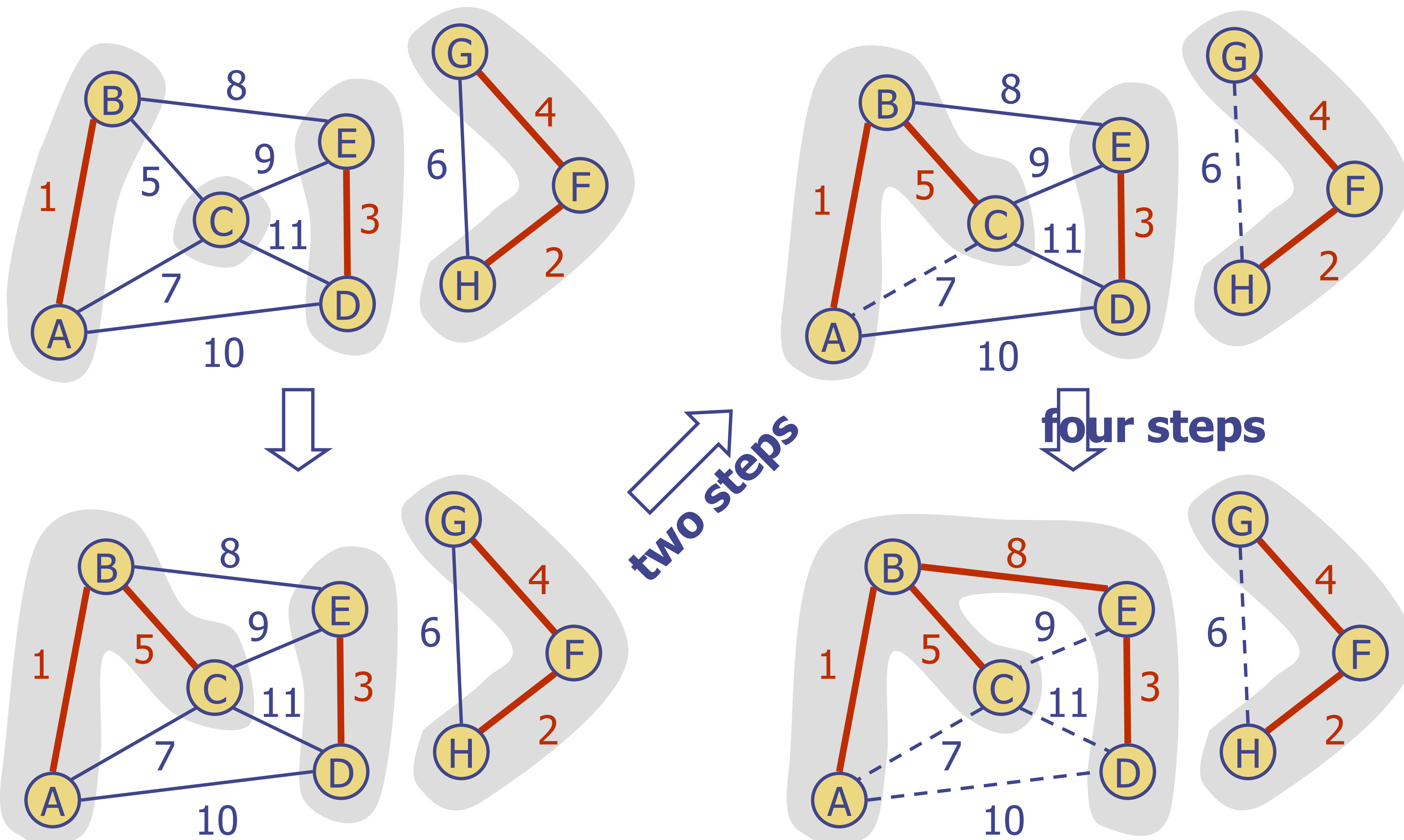


2704
BOS
867
PVD
849
ORD
187
740
144
1846
JFK
621
1258
184
SFO
802
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

Example



Example (contd.)



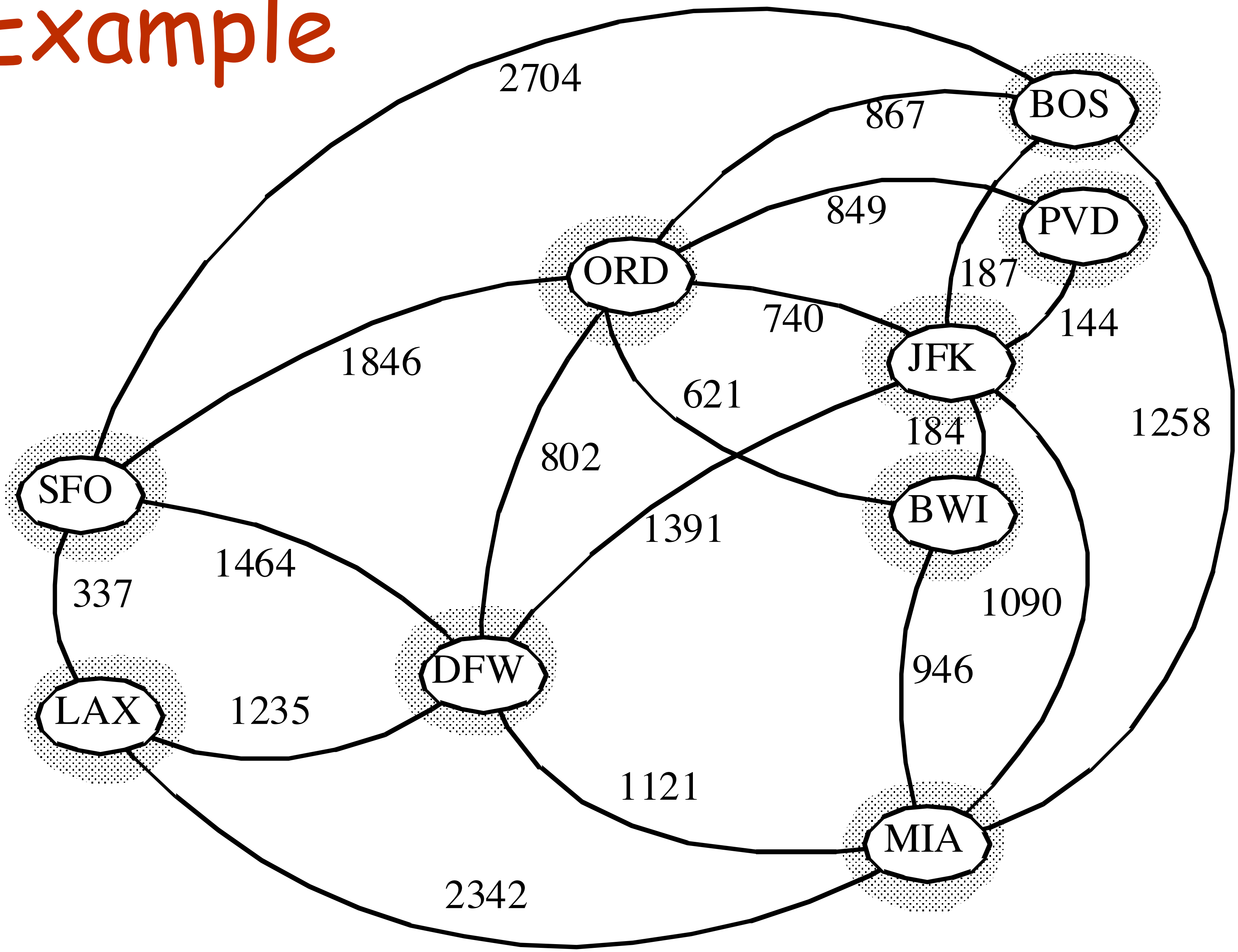
Baruvka's Algorithm

- Like Kruskal's Algorithm, Baruvka's algorithm grows many "clouds" at once.

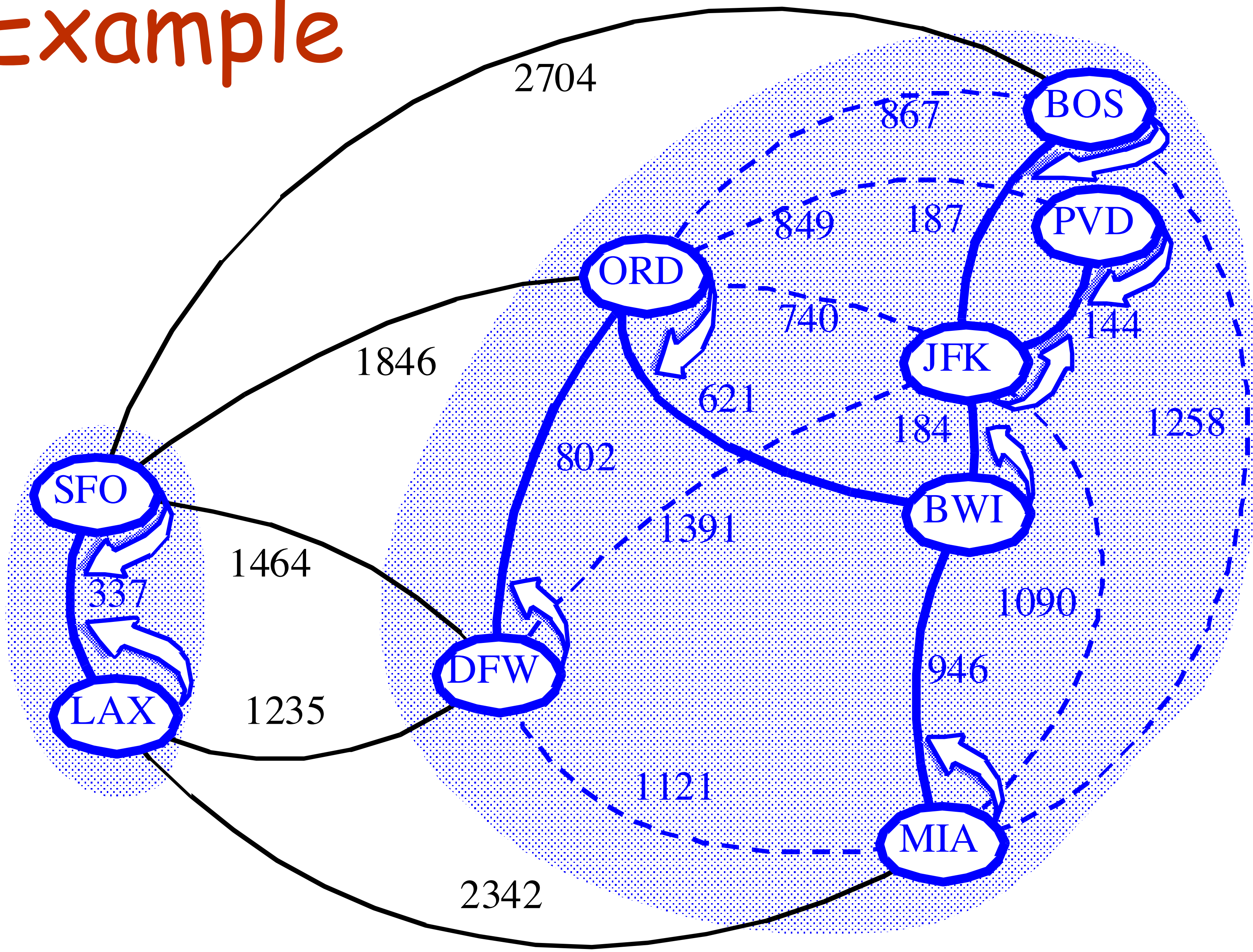
```
Algorithm BaruvkaMST(G)
    T ← V  {just the vertices of G}
  while T has fewer than n-1 edges do
    for each connected component C in T do
      Let edge e be the smallest-weight edge from C to another component in T.
      if e is not already in T then
        Add edge e to T
  return T
```

- Each iteration of the while-loop halves the number of connected compontents in T.
  - The running time is $O(m \log n)$.

Baruvka Example



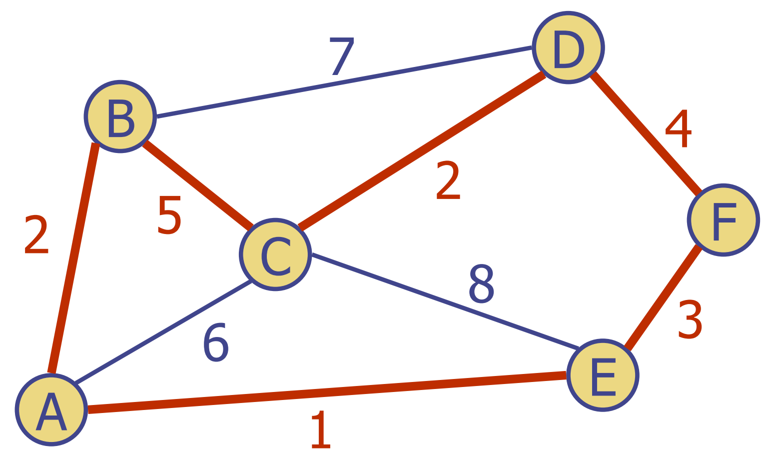
Example



2704
BOS
867
849
187
PVD
ORD
1846
740
144
JFK
621
1258
184
SFO
802
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Traveling Salesperson Problem

- A tour of a graph is a spanning cycle (e.g., a cycle that goes through all the vertices)
- A traveling salesperson tour of a weighted graph is a tour that is simple (i.e., no repeated vertices or edges) and has has minimum weight
- No polynomial-time algorithms are known for computing traveling salesperson tours
- The traveling salesperson problem (TSP) is a major open problem in computer science
  - Find a polynomial-time algorithm computing a traveling salesperson tour or prove that none exists



Example of traveling salesperson tour (with weight 17)

TSP Approximation

- We can approximate a TSP tour with a tour of at most twice the weight for the case of Euclidean graphs
  - Vertices are points in the plane
  - Every pair of vertices is connected by an edge
  - The weight of an edge is the length of the segment joining the points
- Approximation algorithm
  - Compute a minimum spanning tree
  - Form an Eulerian circuit around the MST
  - Transform the circuit into a tour