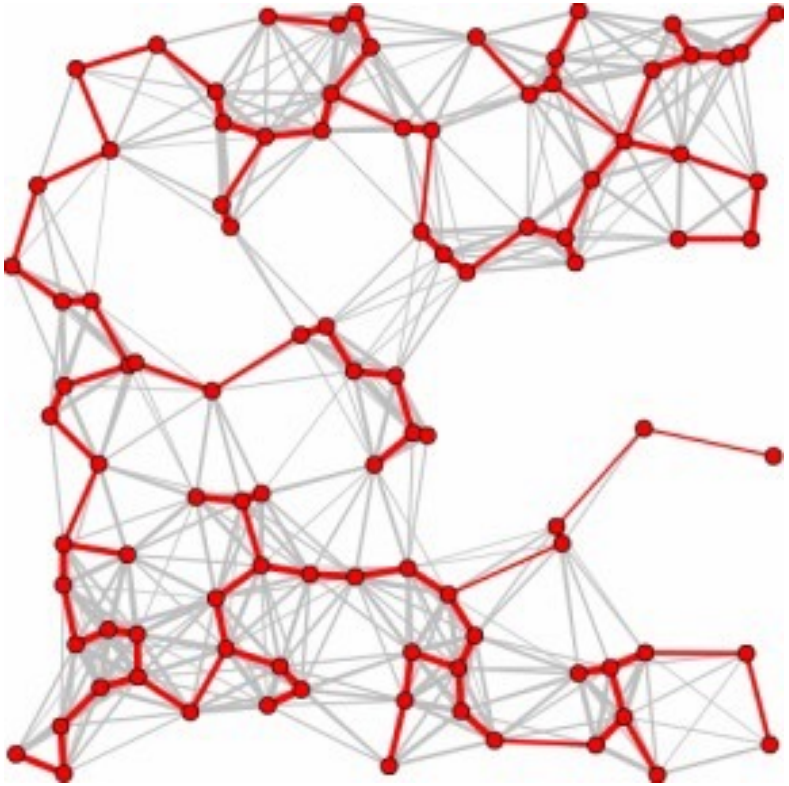
MST

CLRS 21

(+ some supplemental material)



Overview

Problem

- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that

  1. everyone stays connected: can reach every house from all other houses, and
  2. total repair cost is minimum.

Model as a graph:

- Undirected graph $G = (V, E)$.
- ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that

  1. T connects all vertices (T is a ***spanning tree***), and
  2. $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.

Minimum Spanning Tree

Spanning subgraph: subgraph of a graph G containing all the vertices of G

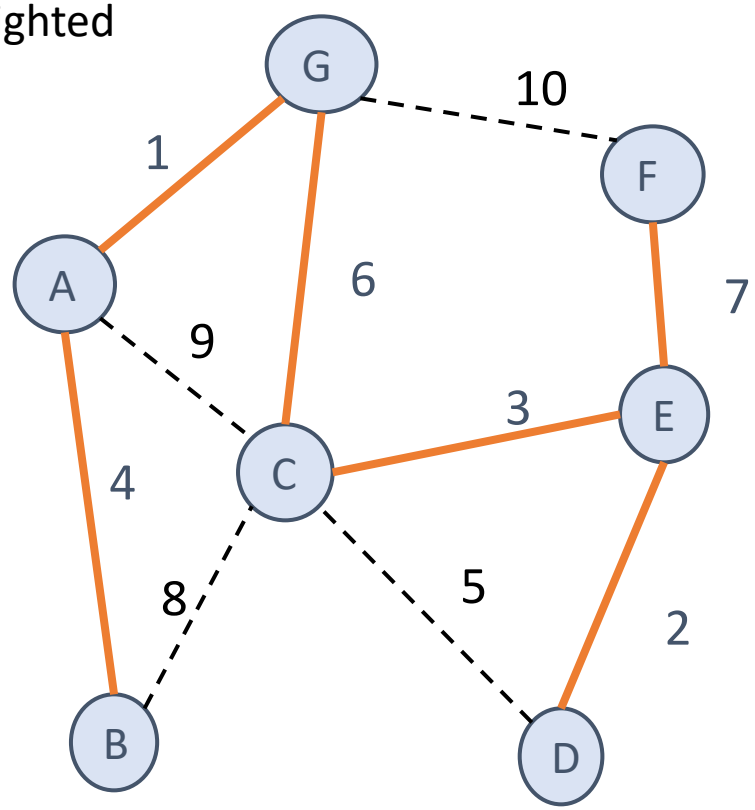
Spanning tree: spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST): spanning tree of a weighted graph with **minimum total edge weight**

- It has |V|-1 edges.
- It has no cycles.
- It might not be unique.

Applications

- Communications networks
- Transportation networks



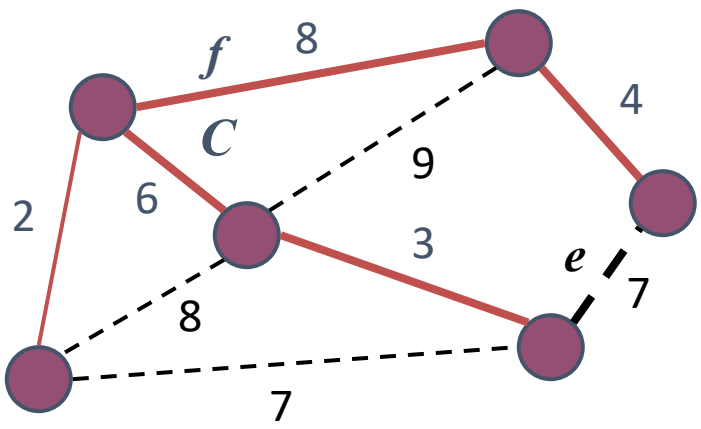
Cycle Property

Cycle Property:

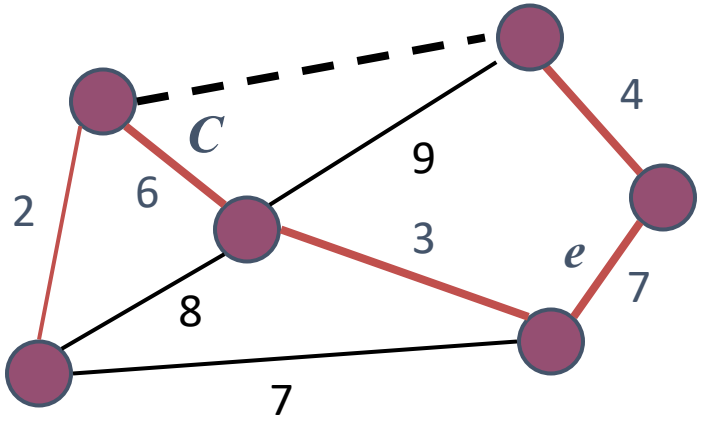
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:

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



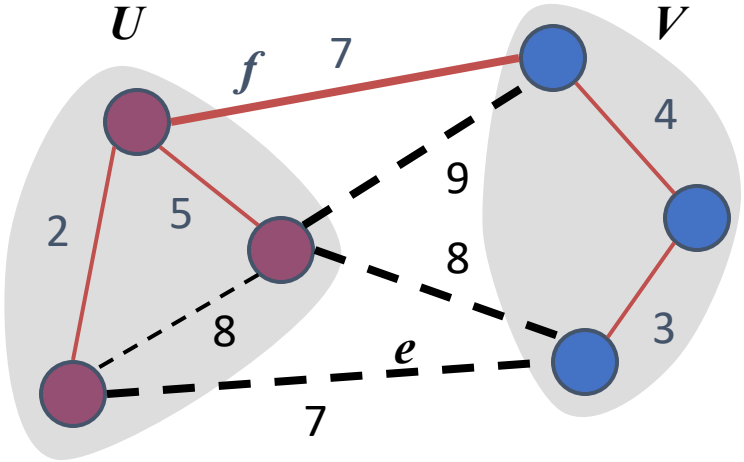
Replacing f with e yields a better spanning tree



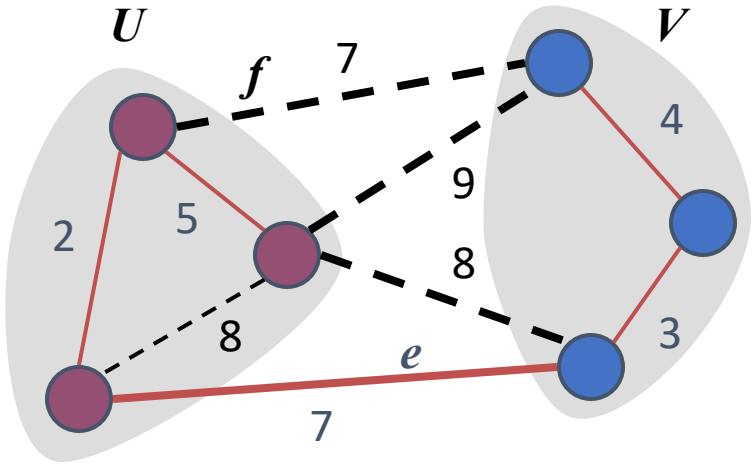
Partition Property

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



Replacing f with e yields another MST



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

A priority queue stores the edges outside the cloud

- Key: weight
- Element: edge

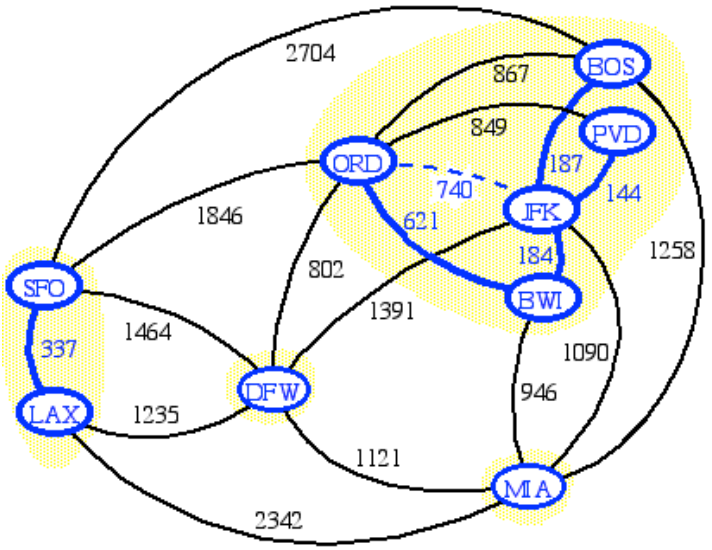
At the end of the algorithm

- We are left with one cloud that encompasses the MST
- A tree *T* which is our MST

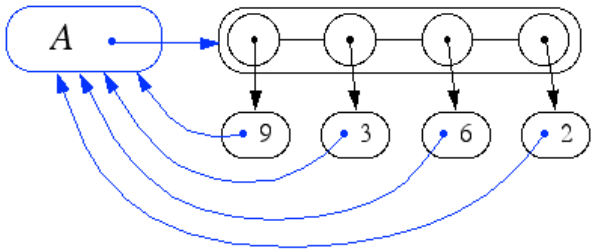
```
Algorithm KruskalMST(G)
    for each vertex V in G do
        define a Cloud(v) of ← {v}
    let Q be a priority queue
    Insert all edges into Q using their weights as the key
    T ← ∅
    while T has fewer than n-1 edges do
        edge e = Q.removeMin()
        Let u, v be the endpoints of e
        { check if edge is necessary to connect two clouds }
        if Cloud(v) ≠ 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

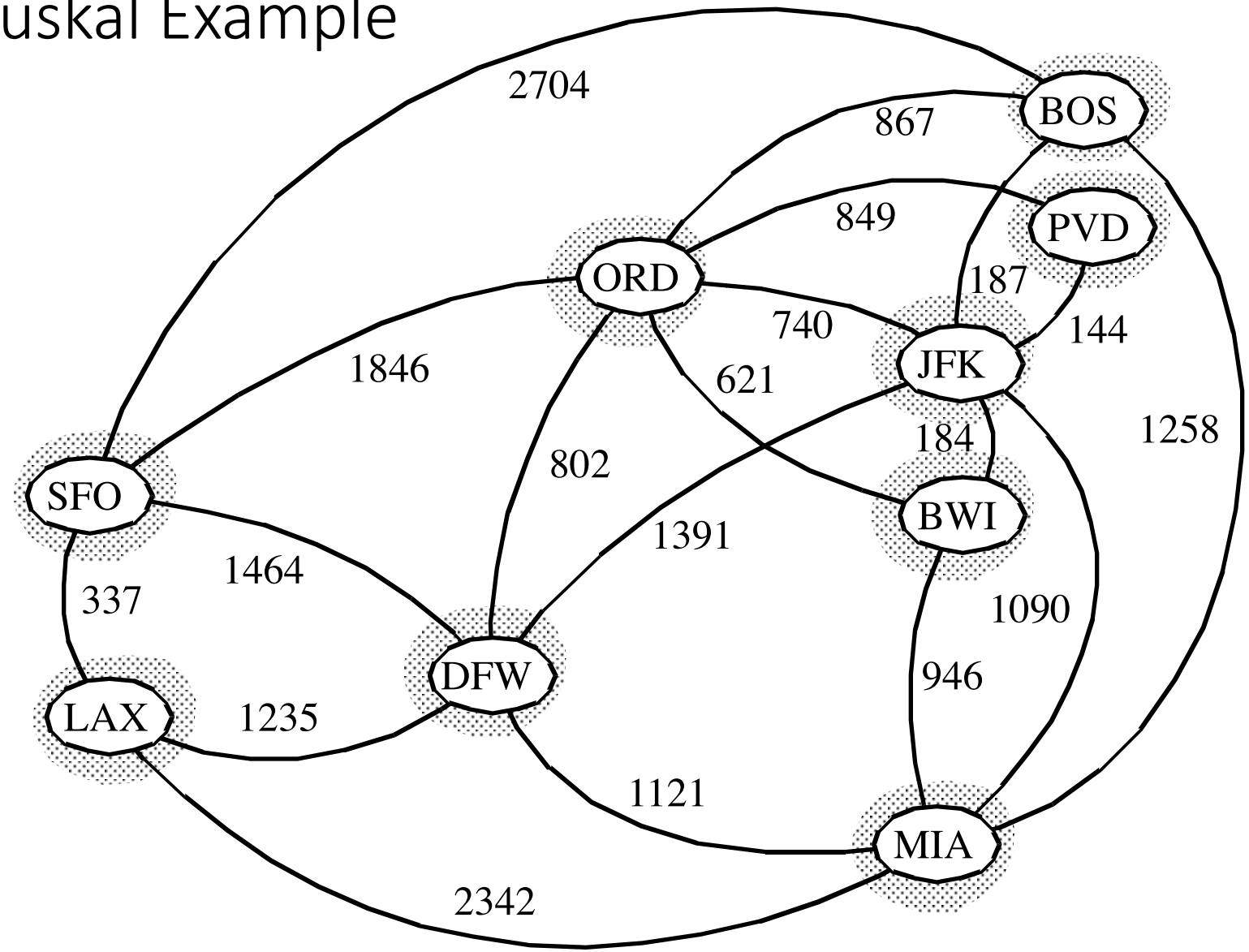
Partition-Based Implementation

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

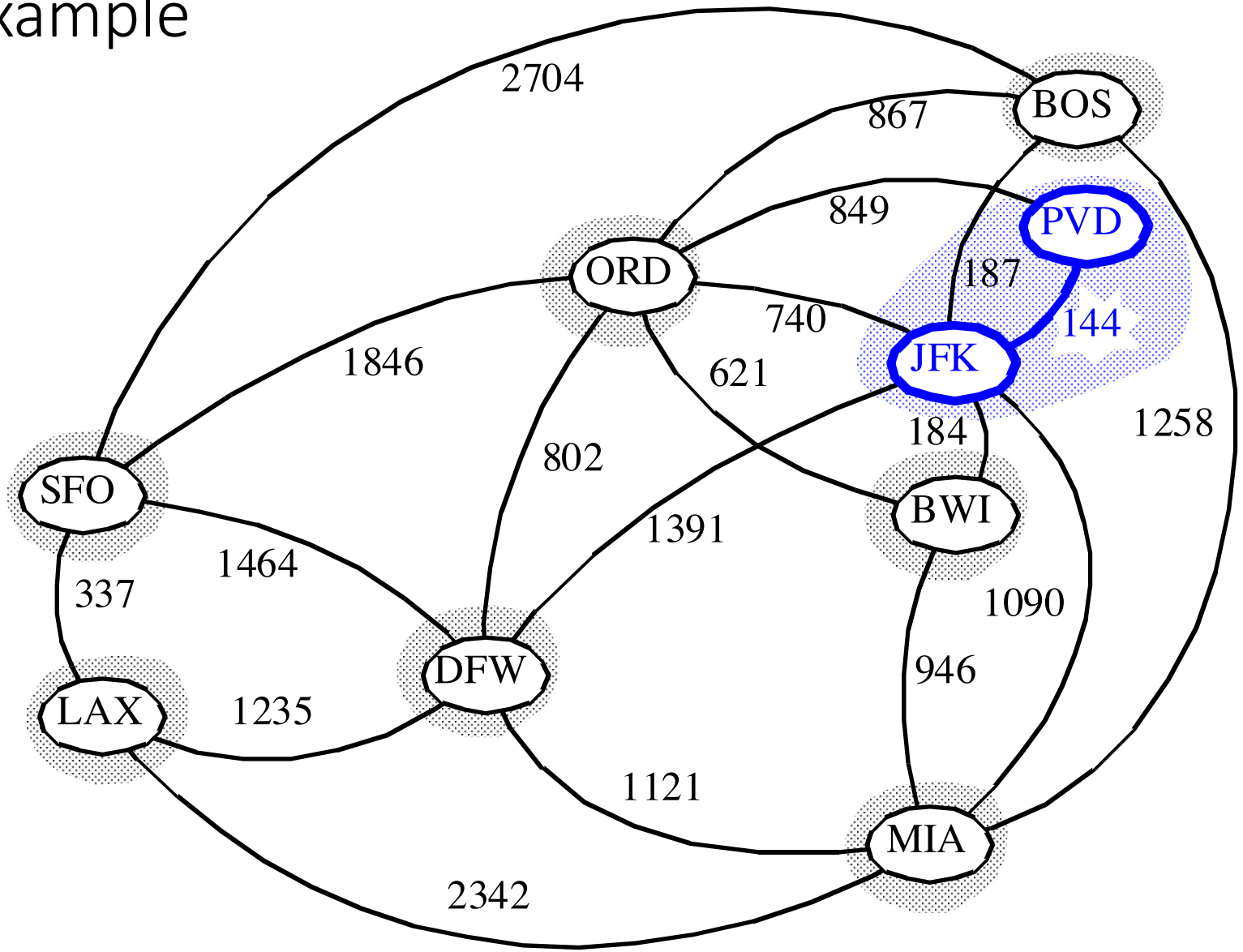
```
Algorithm Kruskal(G):
  Input: A weighted graph G.
  Output: An MST T for G.
1 Let P be a partition of the vertices of G, where each vertex forms a separate set.
2 Let Q be a priority queue storing the edges of G, sorted by their weights
3 Let T be an initially-empty tree
4 while Q is not empty do
5   (u,v) ← Q.removeMinElement()
6   if P.find(u) != P.find(v) then
7       Add (u,v) to T
8       P.union(u,v)
9 return T
```

Running time: O(m log n)

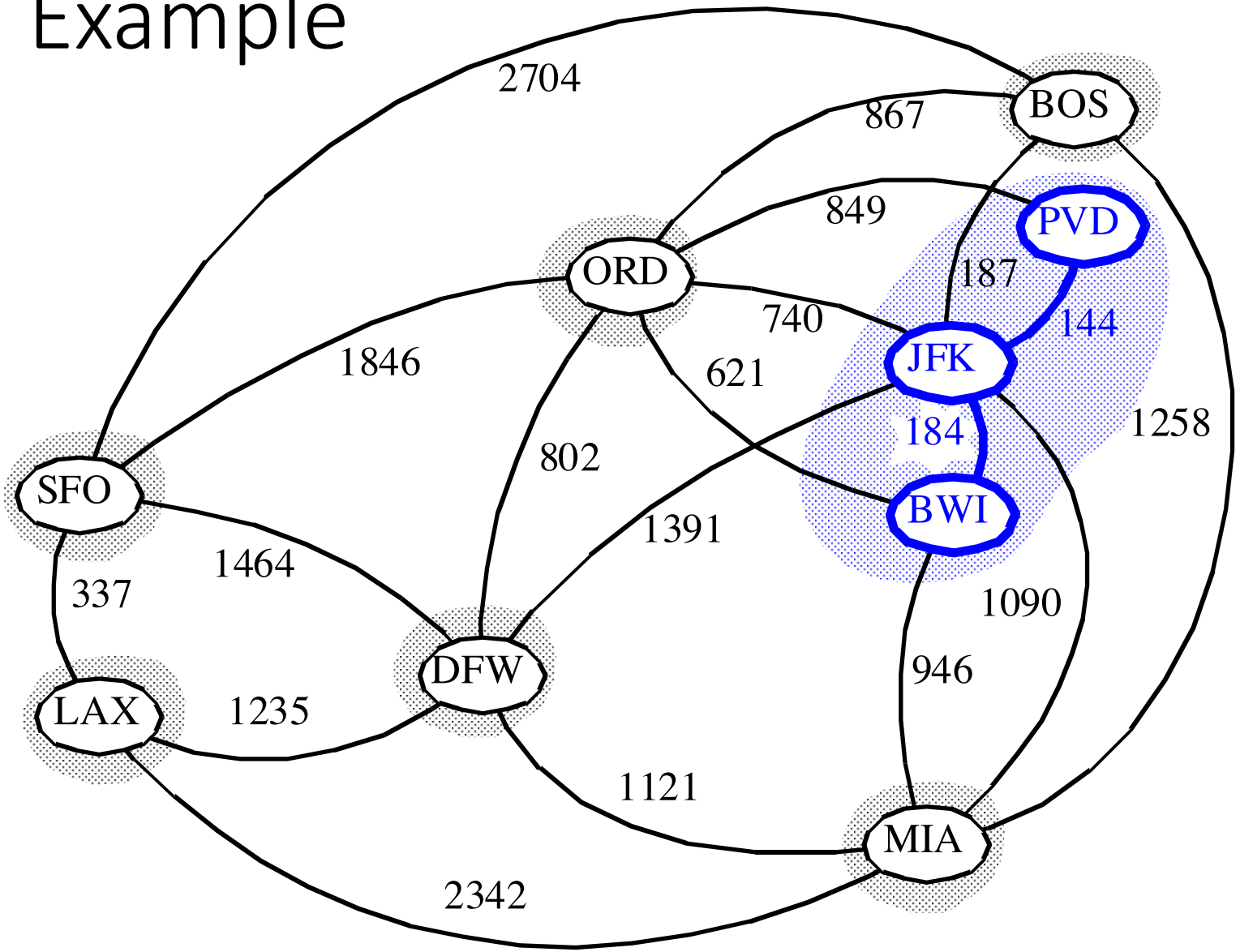
Kruskal Example



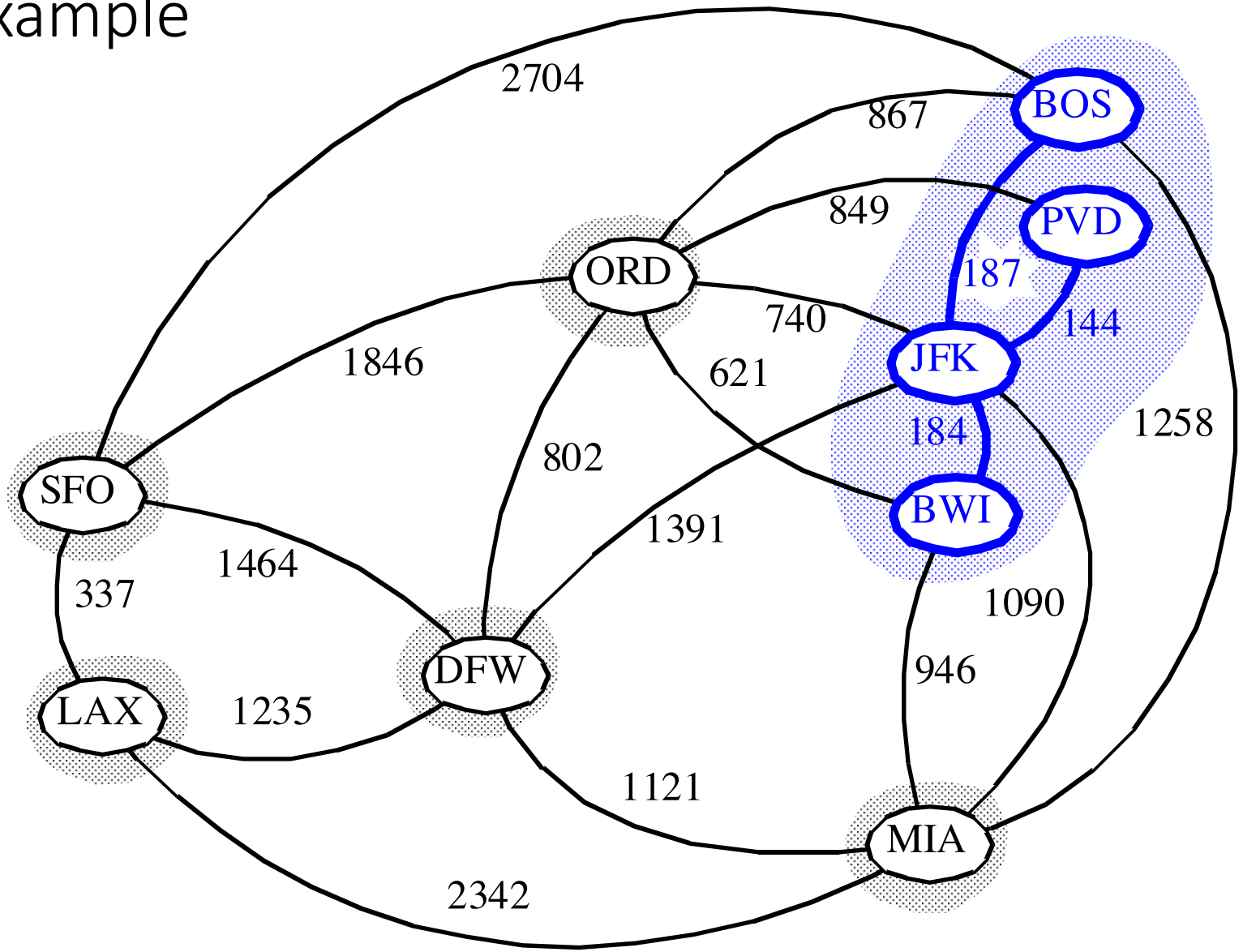
Example


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

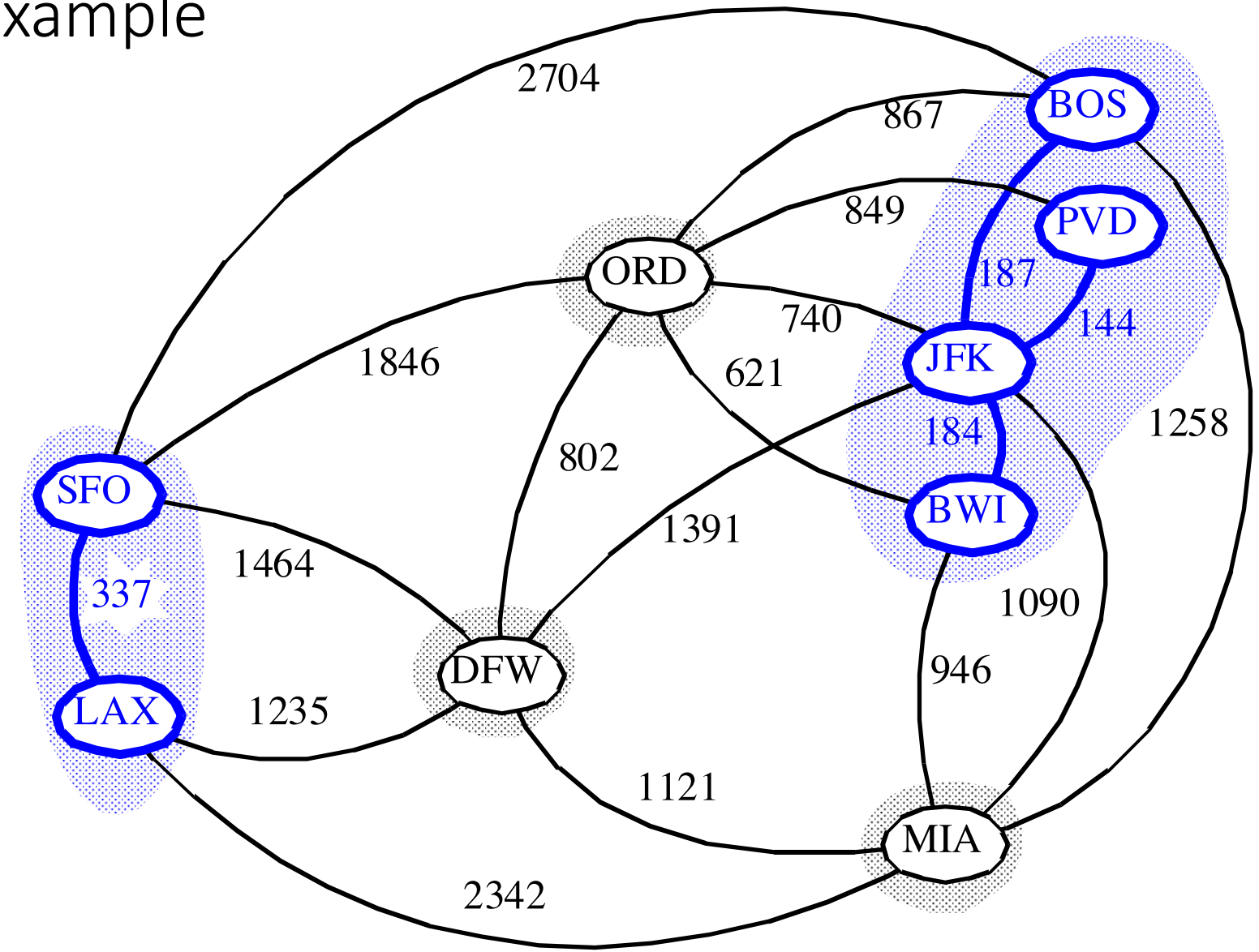
Example


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

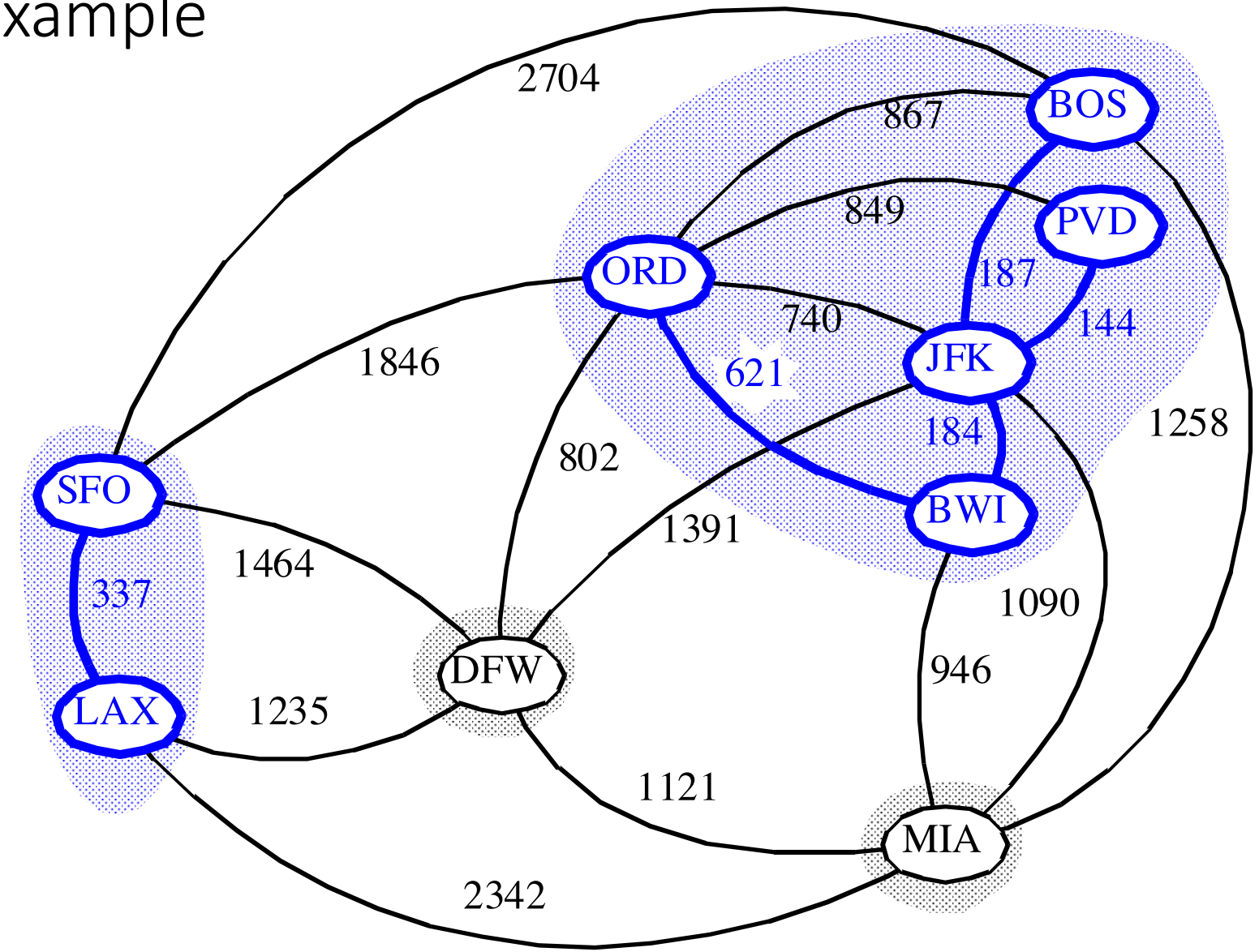
Example


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

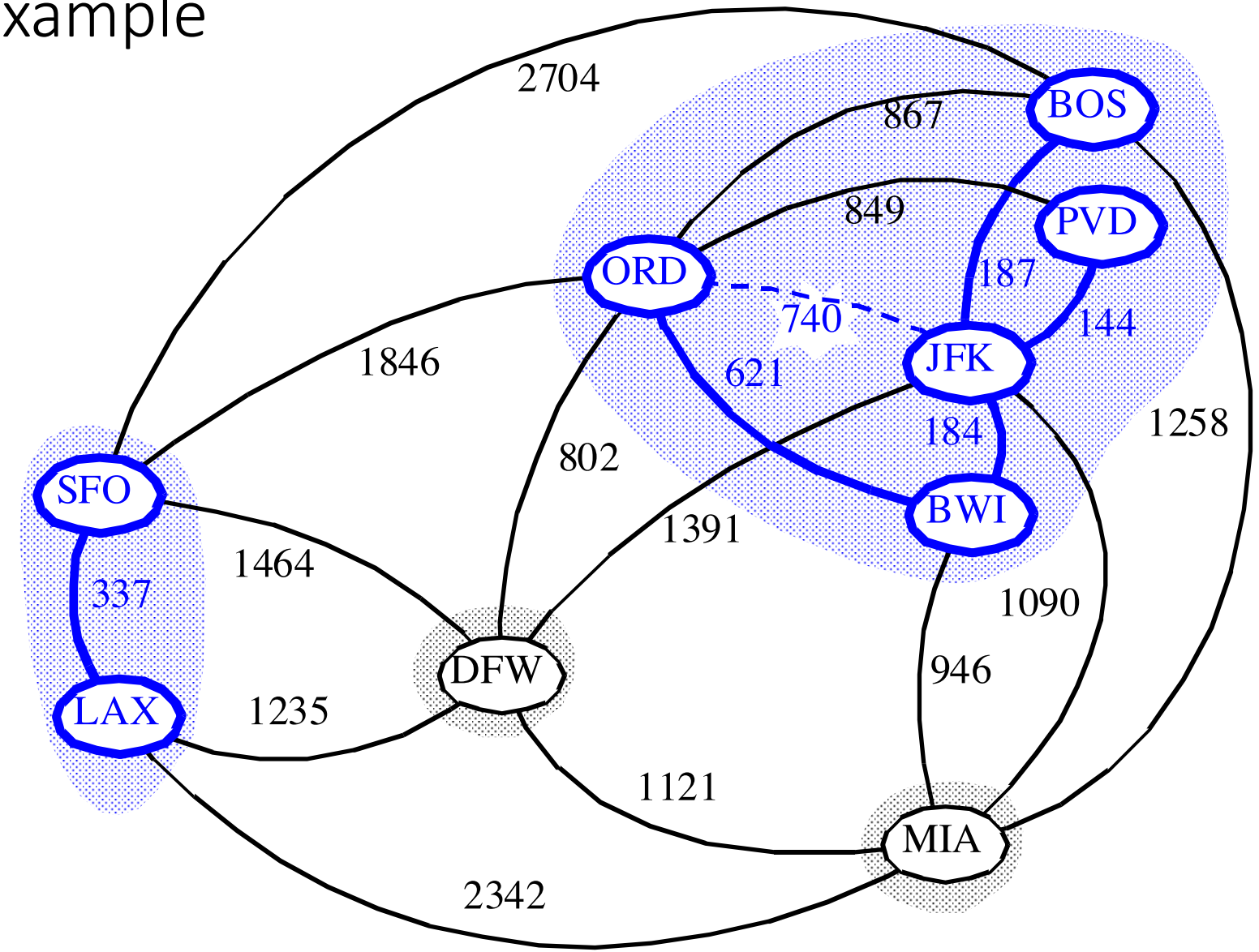
Example


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

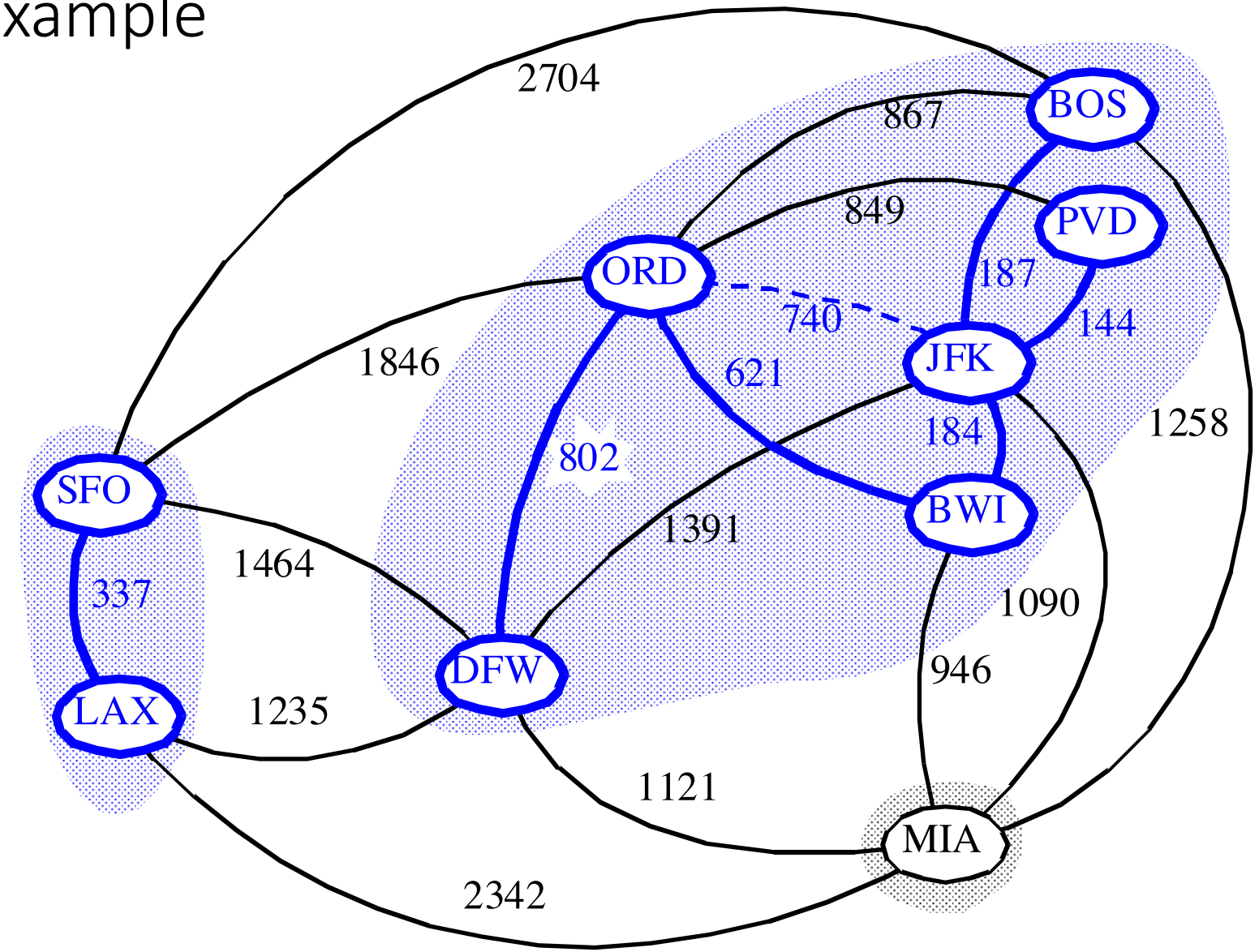
Example


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

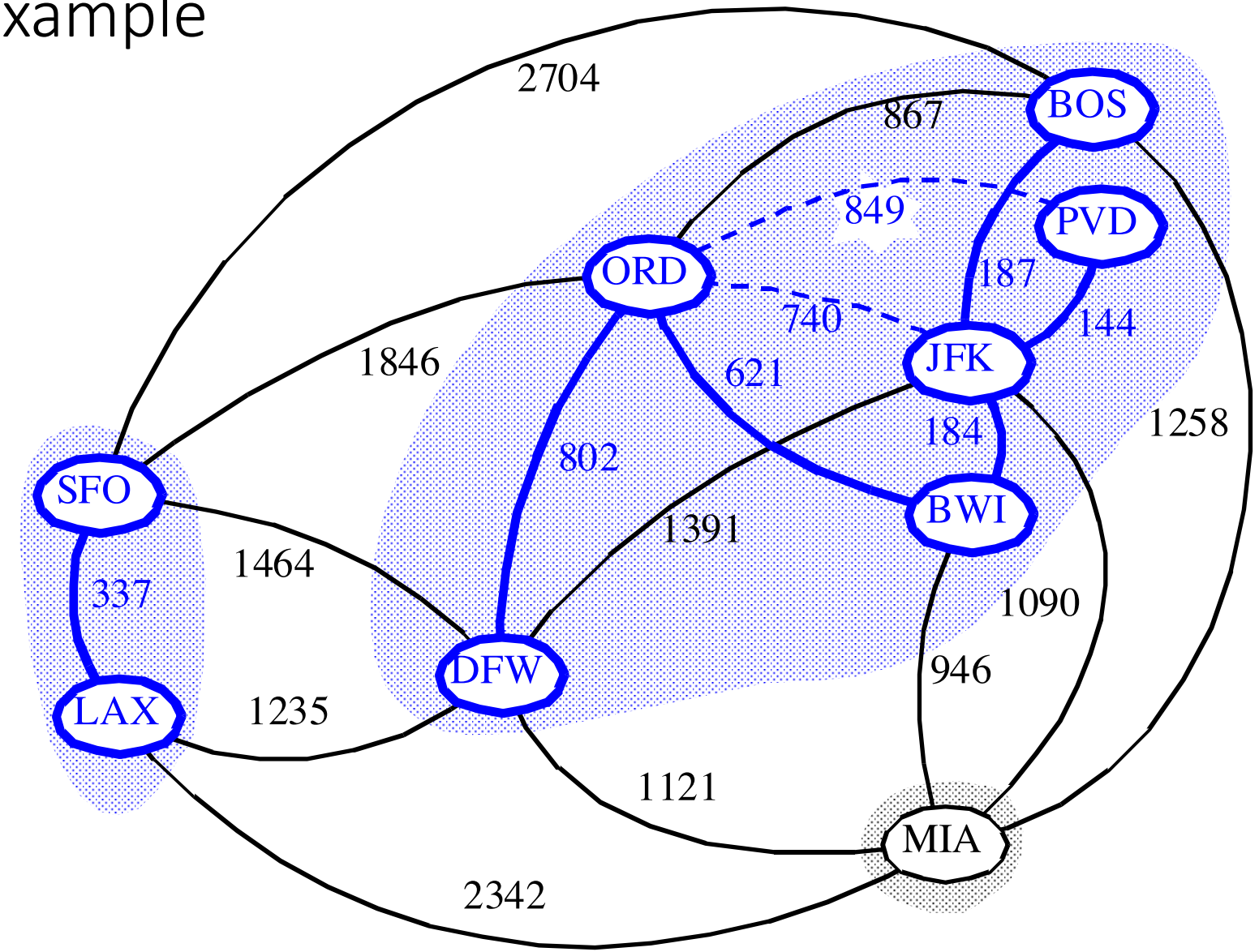
Example


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

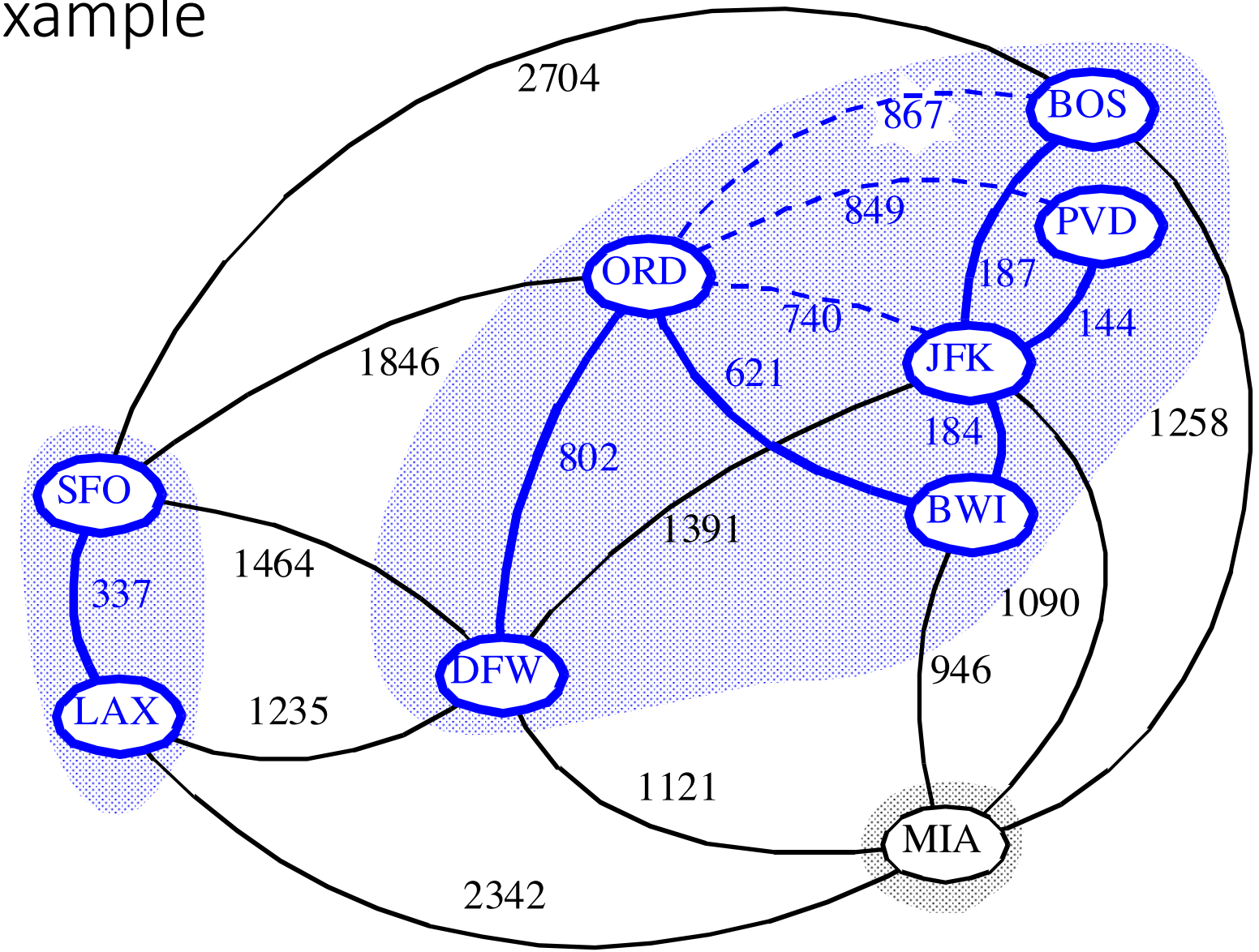
Example


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

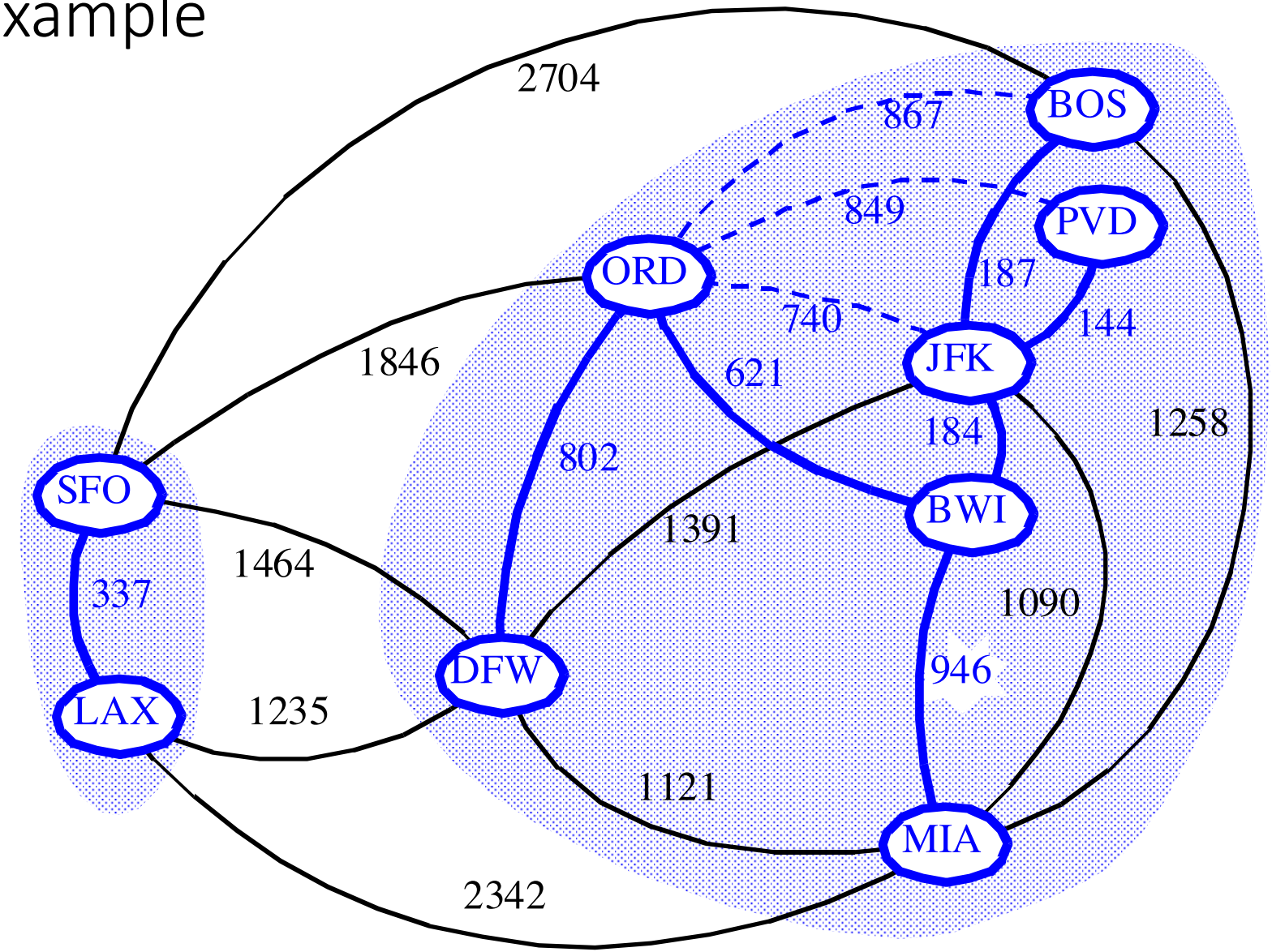
Example


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

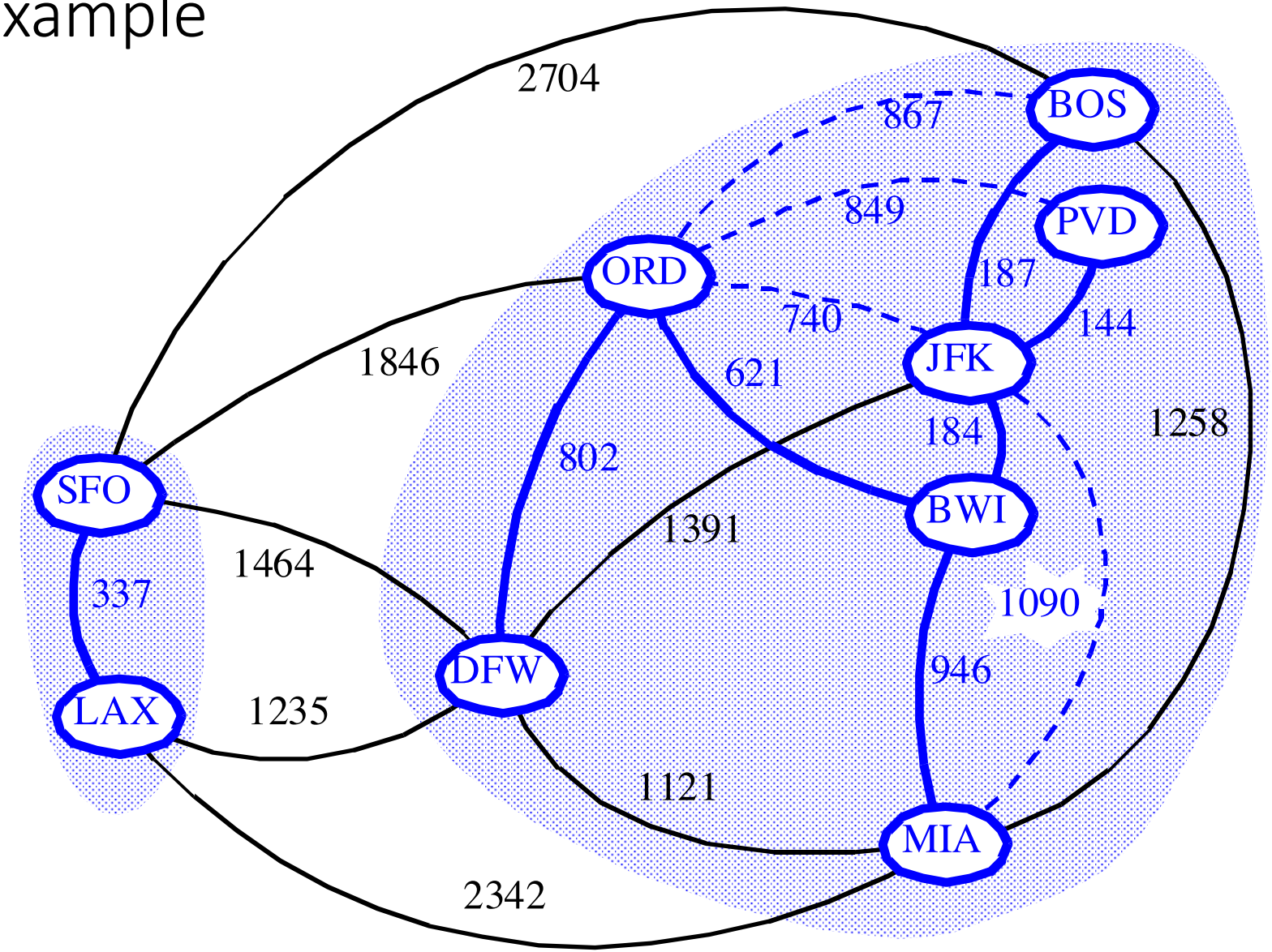
Example


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

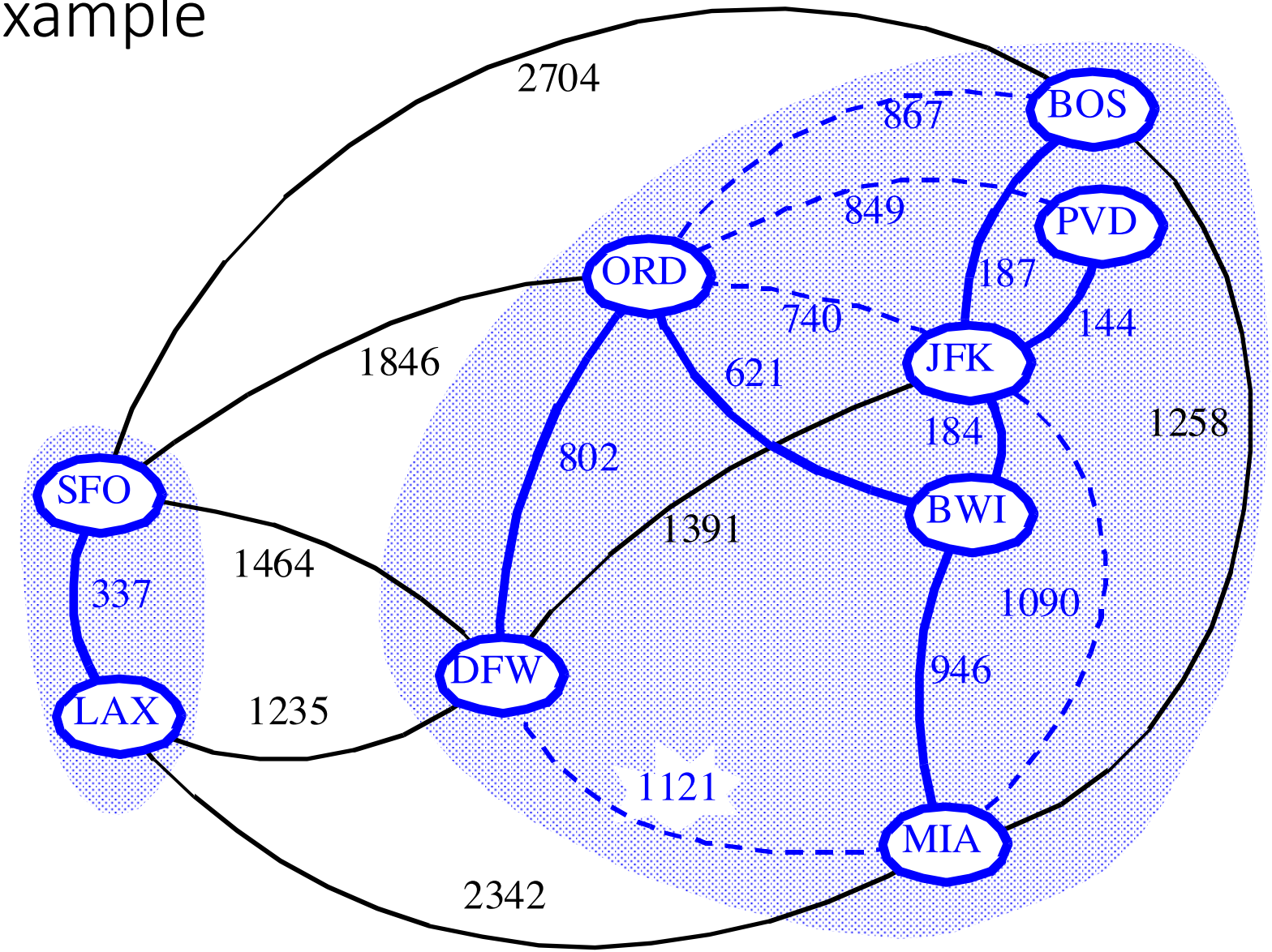
Example


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

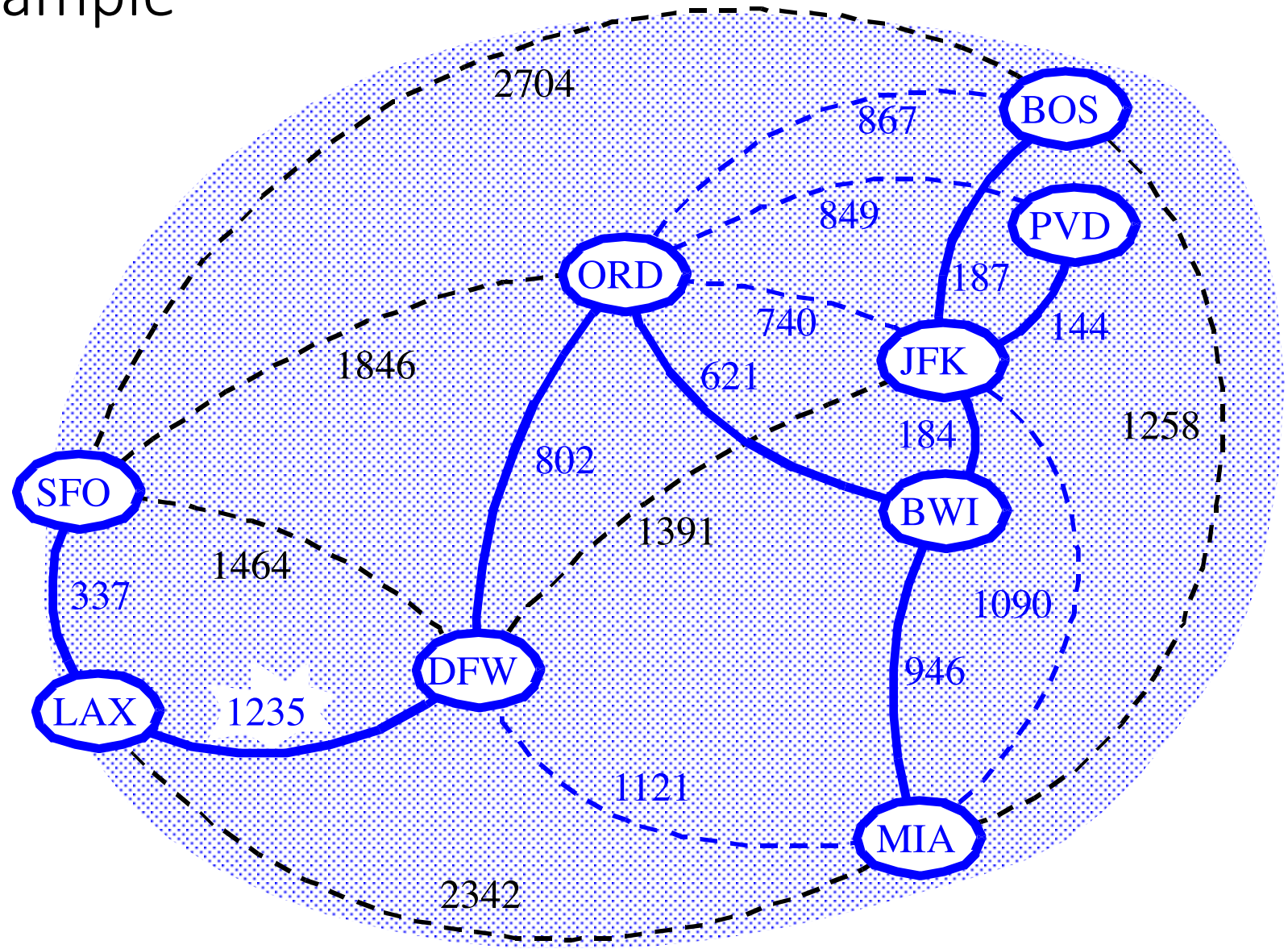
Example


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

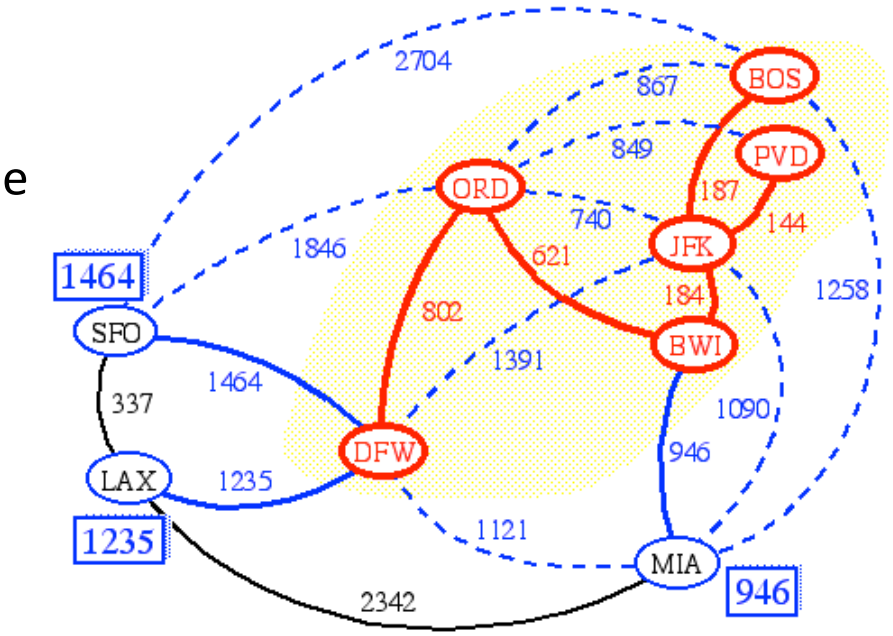
Prim-Jarnik's Algorithm

Idea:

- Builds one tree
- Pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- Store with each vertex v a label $d(v)$ = the smallest weight of an edge connecting v to a vertex in the cloud

At each step:

- Add to the cloud the vertex u outside the cloud with the smallest label
- Update the labels of the vertices adjacent to u



Prim-Jarnik's Algorithm (continued)

A priority queue stores the vertices outside the cloud

- Key: distance
- Element: vertex

Locator-based methods

- *insert*(*k*,*e*) returns a locator
- *replaceKey*(*l*,*k*) changes the key of an item

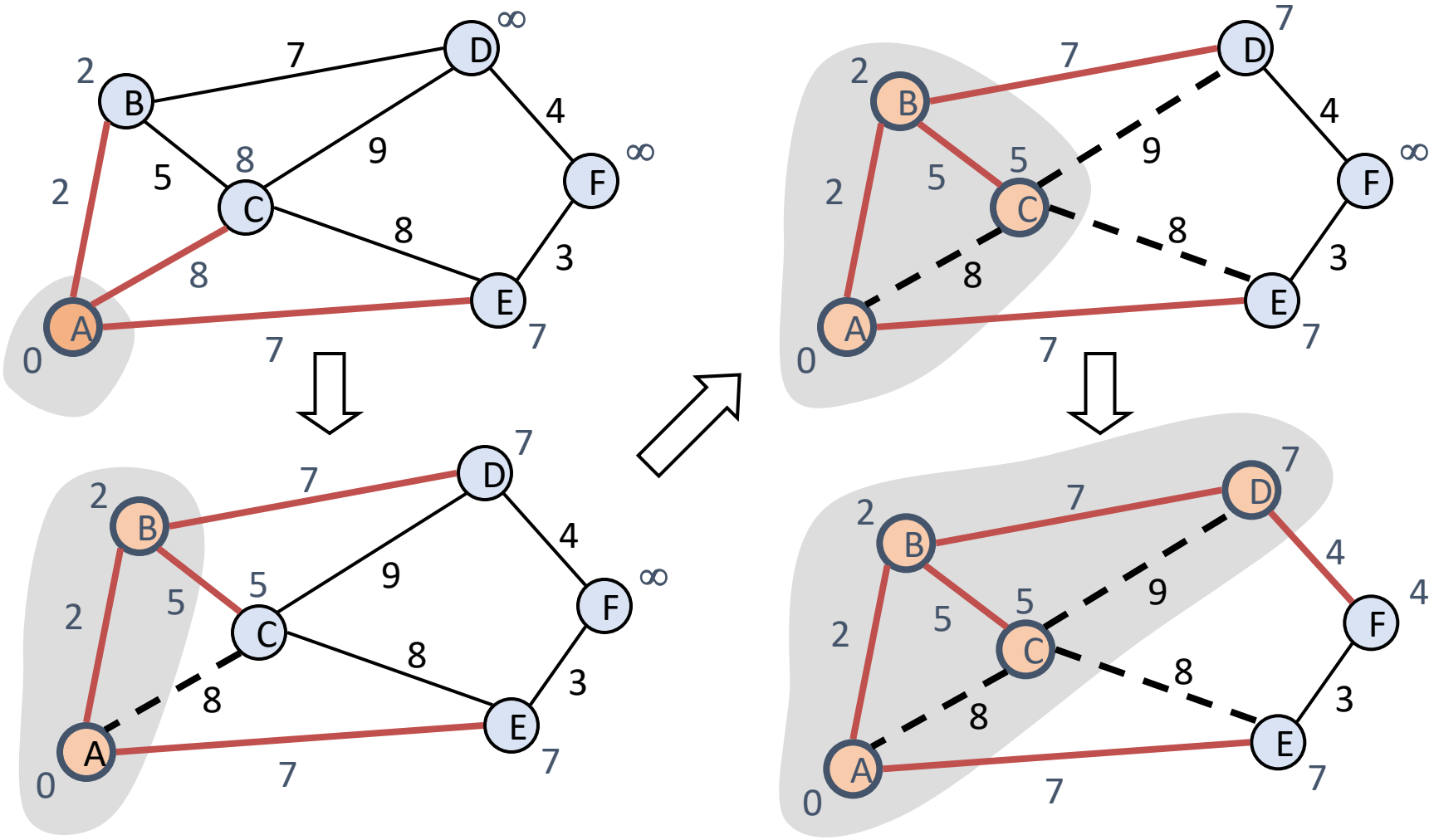
We store three labels with each vertex:

- Distance
- Parent edge in MST
- Locator in priority queue

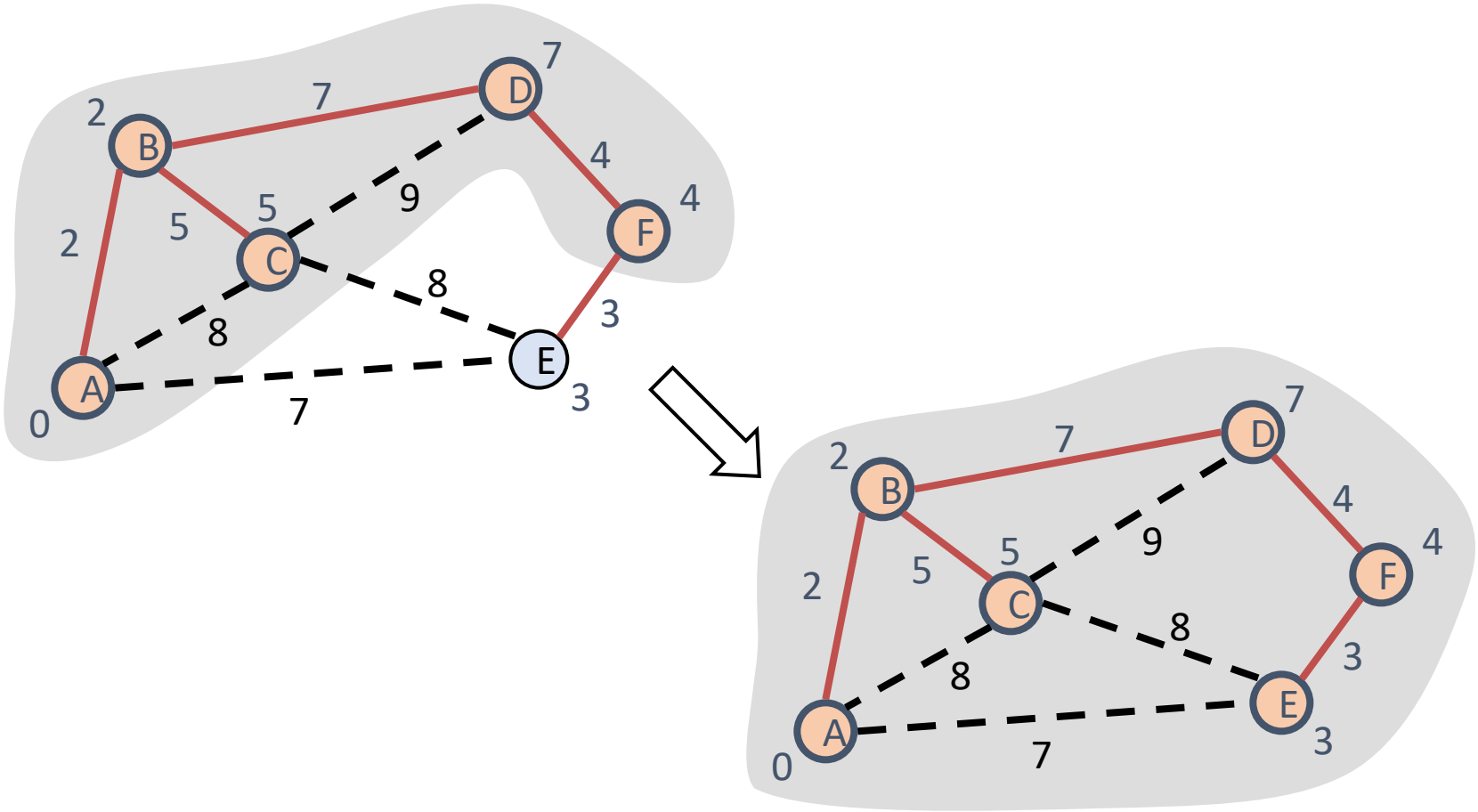
```
Algorithm PrimJarnikMST(G)
    Q ← new heap-based priority queue
    s ← a vertex of G
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        setParent(v, ∅)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    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)
```

Running time: O(*m* log *n*)

Example



Example (continued)



There's a long list of cities that Santa Claus needs to visit, and he only has from now until Christmas to figure out a good route to take. Use techniques you have learned in this course to give an efficient algorithm that will find a guaranteed short route starting from the North Pole that will visit every city and come back to the North Pole again. You don't need to find the (optimal) shortest route, but you can guarantee that you will always come *close* to the shortest route. This is known as an approximation algorithm. For example, a 10-approximation algorithm would produce a route whose length is no more than 10 times the length of the shortest route.

1. How does your algorithm work?
2. What is the running time of your algorithm?
3. How closely does your algorithm approximate the optimal route?