EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

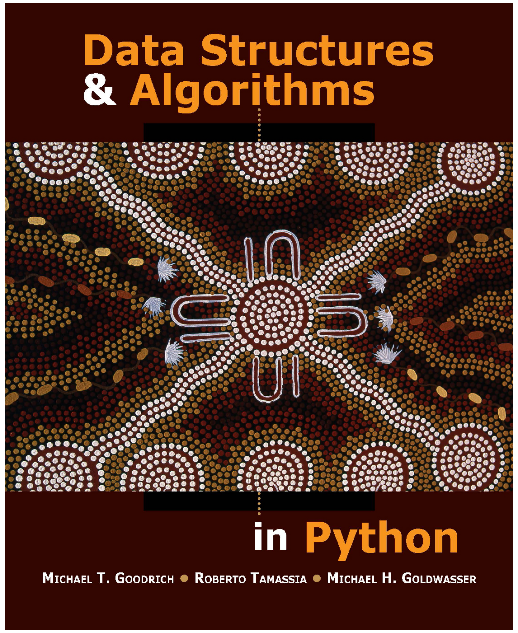
FACULTY OF
HUMANITIES
Department of General and Computational Linguistics

Minimum Spanning Trees

Data Structures and Algorithms for CL III, WS 2019-2020

Corina Dima
corina.dima@uni-tuebingen.de

Michael Goodrich
Roberto Tamassia
Michael Goldwasser



14.7 Minimum Spanning Trees

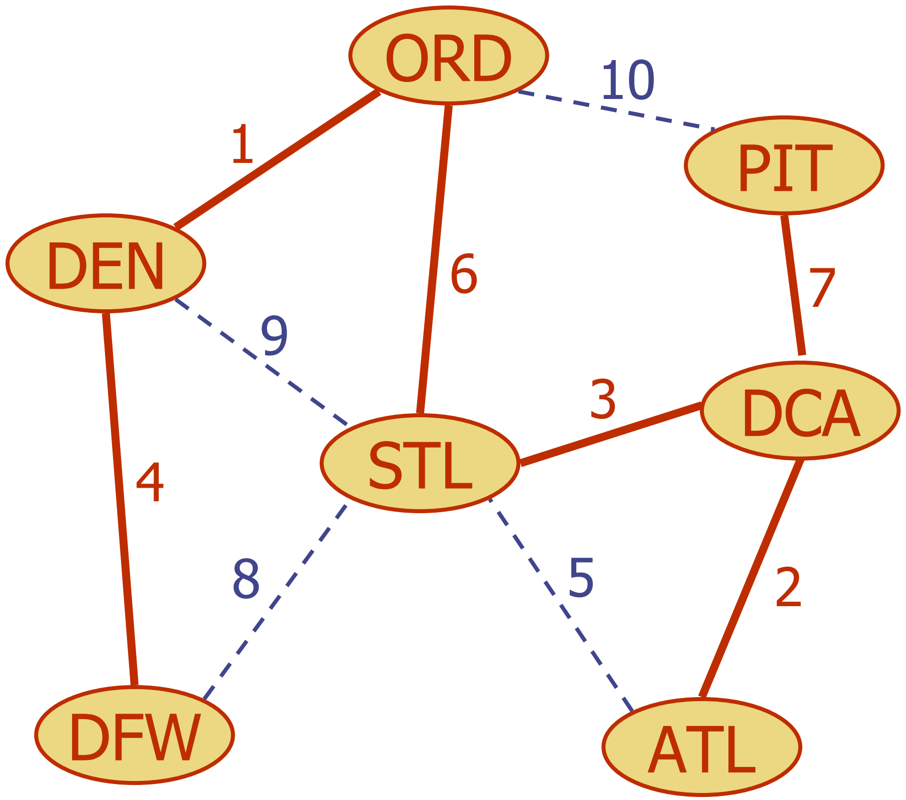
- ❖ Prim-Jarník Algorithm
- ❖ Kruskal's Algorithm
- ❖ Disjoint Partitions and Union-Find Structures

Minimum Spanning Tree – Sample Problem

- Suppose a company needs to connect all the computers in a new office building using the least amount of cable
- Model the problem using an undirected weighted graph G:
  - The vertices represent the computers
  - The edges represent all the possible pairs (u, v) of computers, where the weight $w(u, v)$ of the edge is the amount of cable needed to connect computers u and v
- Not interested in the shortest path between u and v – rather, in finding a tree T, containing all the vertices in G, with minimum weight (minimum sum of the edge weights) over all the possible trees

Minimum Spanning Tree - Terminology

- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is a tree (no cycles)
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight



Minimum Spanning Tree

- Given an undirected, weighted graph G, find a tree T that contains all the vertices of G and minimizes the sum

$$w(T) = \sum_{(u,v) \text{ in } T} w(u,v)$$

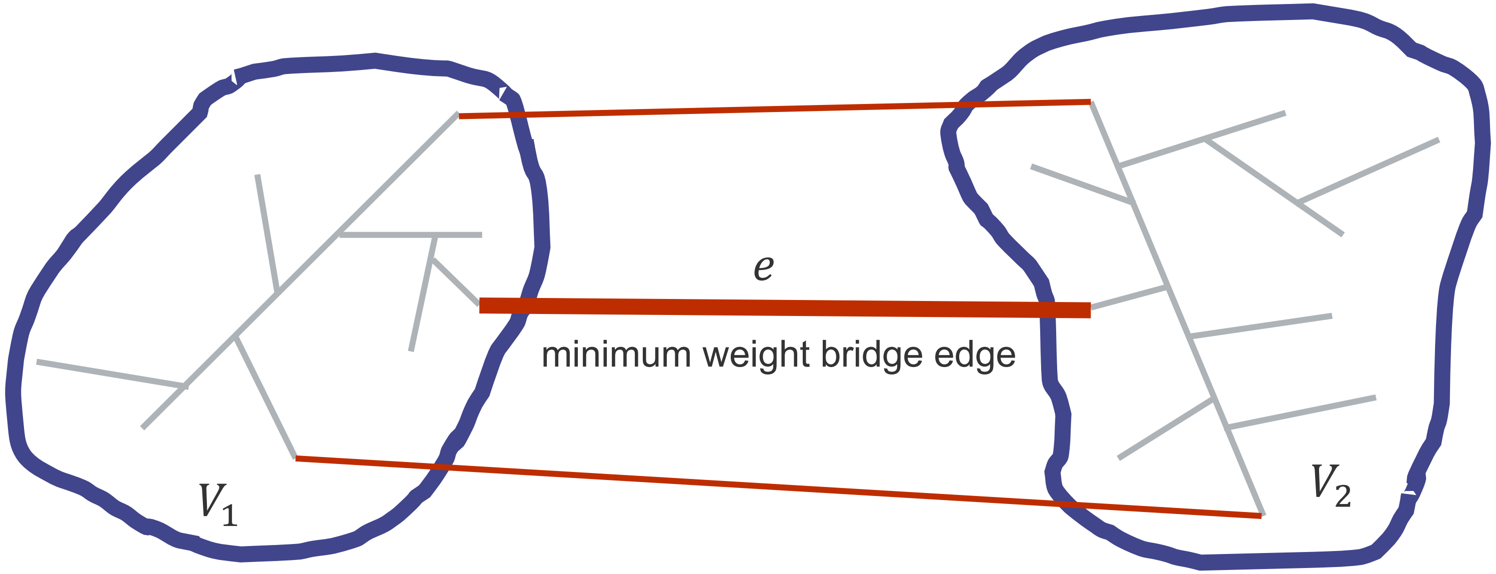
- Computing a spanning tree with the smallest total weight is known as the minimum spanning tree (MST) problem
- Two algorithms for computing the MST of a graph:
  - The Prim-Jarník algorithm, which “grows” the MST from a single root vertex (similar to Dijkstra’s algorithm)
  - Kruskal’s algorithm, which “grows” the MST in clusters by considering edges in nondecreasing order of their weights
  - Both greedy algorithms – the next edge to be added has to minimize the total cost

Minimum Spanning Tree - Prequel

- Simplifying assumptions:
  - The graph G is undirected
  - The graph G is simple (it has no self-loops or parallel edges)

Minimum Spanning Tree – Prequel (2)

- Proposition. Let G be a weighted connected graph, and V_1 and V_2 be a partition of the vertices of G into two disjoint, non-empty sets. Also, let e be an edge in G with minimum weight among those edges of G that have an endpoint in V_1 and another one in V_2. There is a minimum spanning tree T that has e as one of its edges.



Minimum Spanning Tree – Prequel (3)

- Justification.
  - Let T be a minimum spanning tree of G.
  - If T does not contain edge e, then the addition of e to T must create a cycle.
  - Therefore, there is an edge $f \neq e$ in this cycle with one endpoint in V_1 and another endpoint in V_2
  - $w(e) \leq w(f)$ – because e was chosen to be the minimum weight edge between those with an edge in V_1 and another edge in V_2
  - If f is removed from $T \cup \{e\}$, then the new minimum spanning tree obtained has a total weight that is not larger than the weight of T
  - Since T was a minimum spanning tree, the new tree must also be a minimum spanning tree.

Minimum Spanning Tree – Prequel (4)

- The proposition is valid even if G has negative weights or negative-weight cycles
- If the weights of the graph are distinct, then there is an unique minimum spanning tree
  - Otherwise G has multiple minimum spanning trees

Prim-Jarník Algorithm

Prim-Jarník Algorithm - Intuition

- Grow a minimum spanning tree from a single cluster, starting from a “root” vertex s
- Similar to Dijkstra’s algorithm:
  - Begin with a vertex s, which becomes the initial “cloud” of vertices C
  - At each iteration, choose a minimum-weight edge $e = (u, v)$, connecting a vertex u from the “cloud” C to a vertex v outside of C
  - The vertex v is brought into C – for each vertex we store the label $D[v]$ representing the smallest weight of an edge connecting v to a vertex in C
  - The iterative process is repeated until a spanning tree is formed
  - The validity of this approach rests on the property presented before - the vertices in the “cloud” and the vertices outside of it form the two sets of vertices, V_1 and V_2
  - Whenever we add a new edge of minimum weight, we are adding a valid edge to the minimum spanning tree

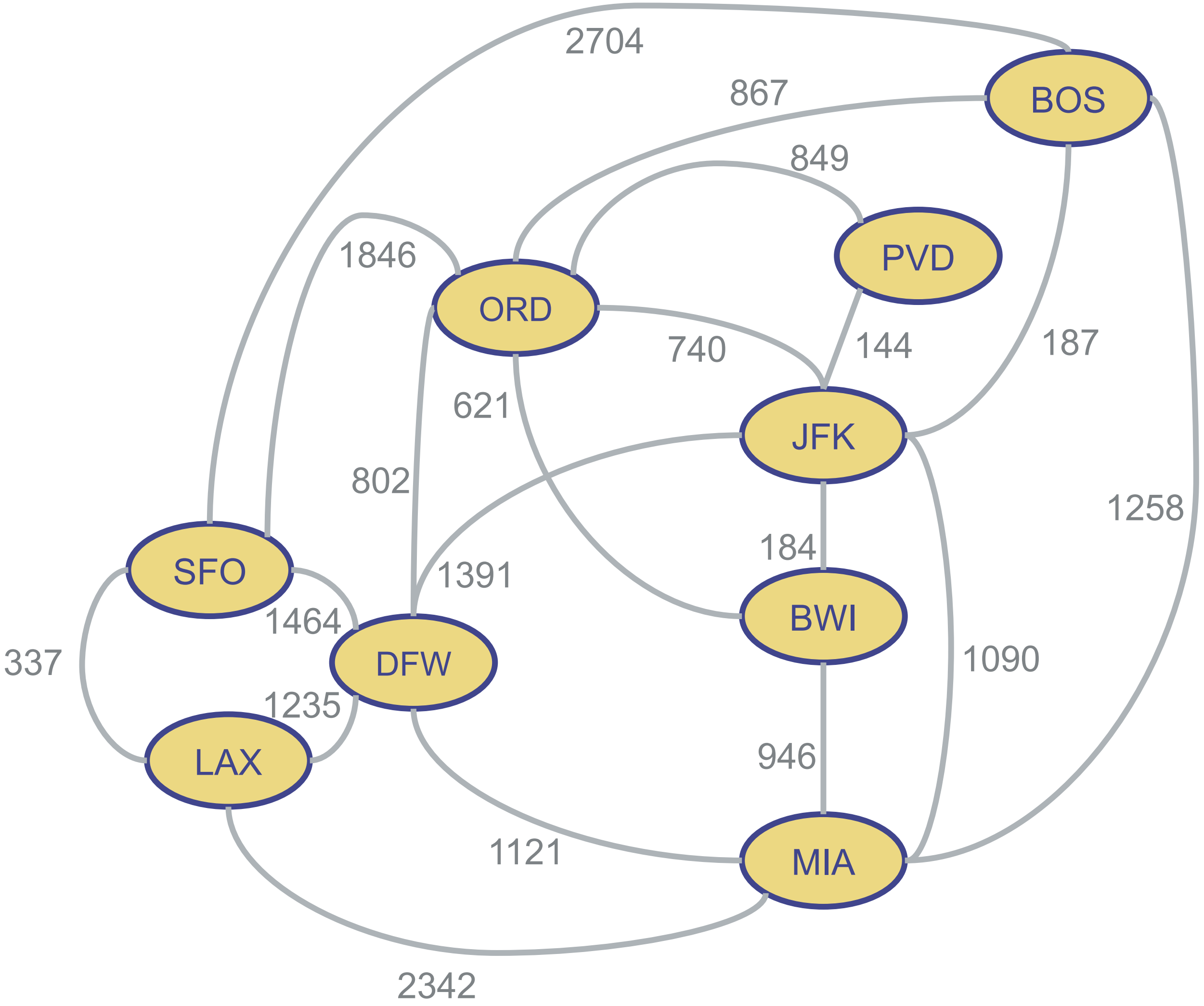
Prim-Jarník Algorithm - Pseudocode

Algorithm PrimJarnik(G):

Input: An undirected, weighted, connected graph G with n vertices and m edges
Output: A minimum spanning tree T for G

Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

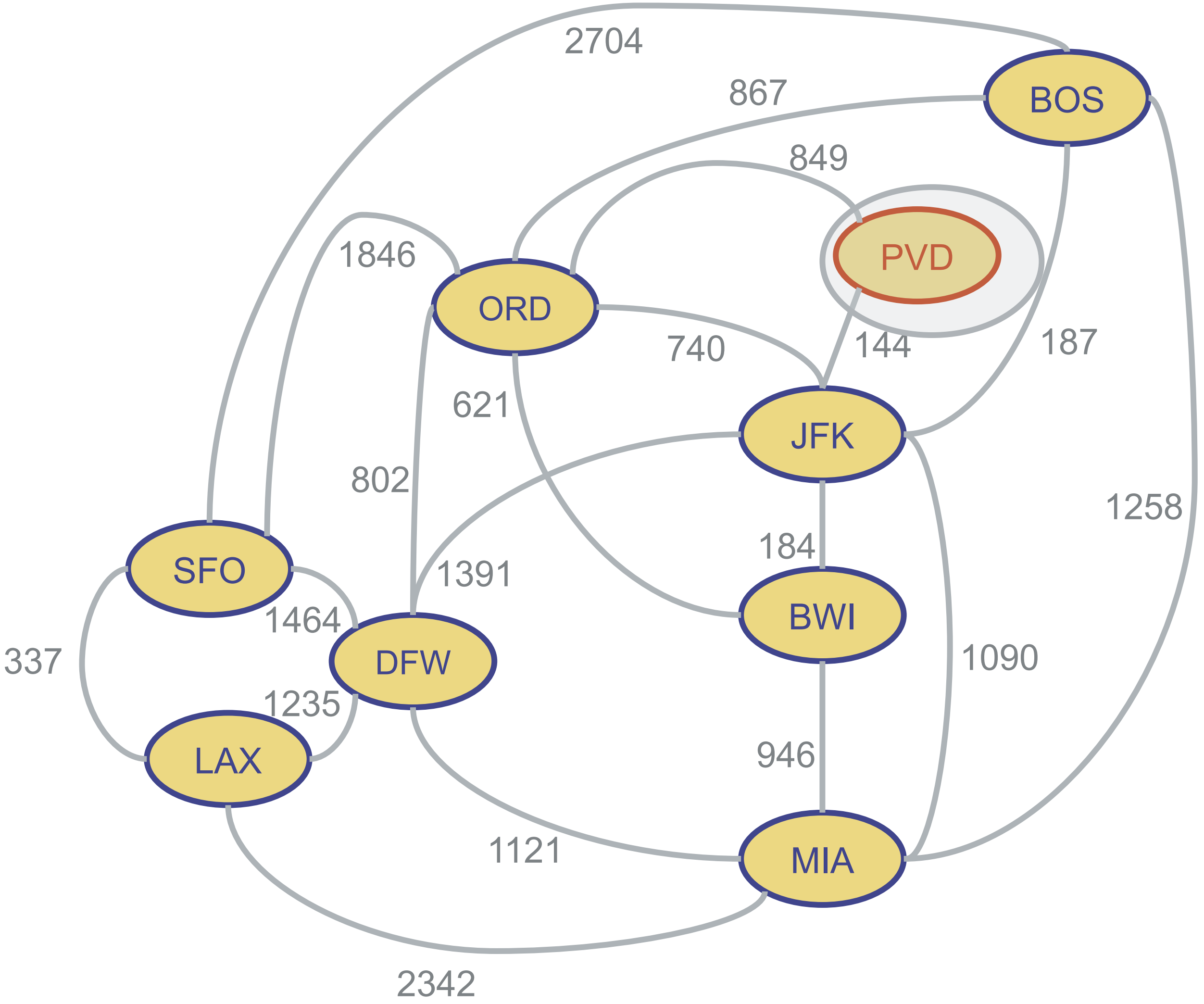
Prim-Jarník Algorithm - Example



PQ	Tree
∞	(BOS, None)
0	(PVD, None)
∞	(JFK, None)
∞	(BWI, None)
∞	(MIA, None)
∞	(ORD, None)
∞	(DFW, None)
∞	(SFO, None)
∞	(LAX, None)

- Start vertex is PVD, the only one with length 0

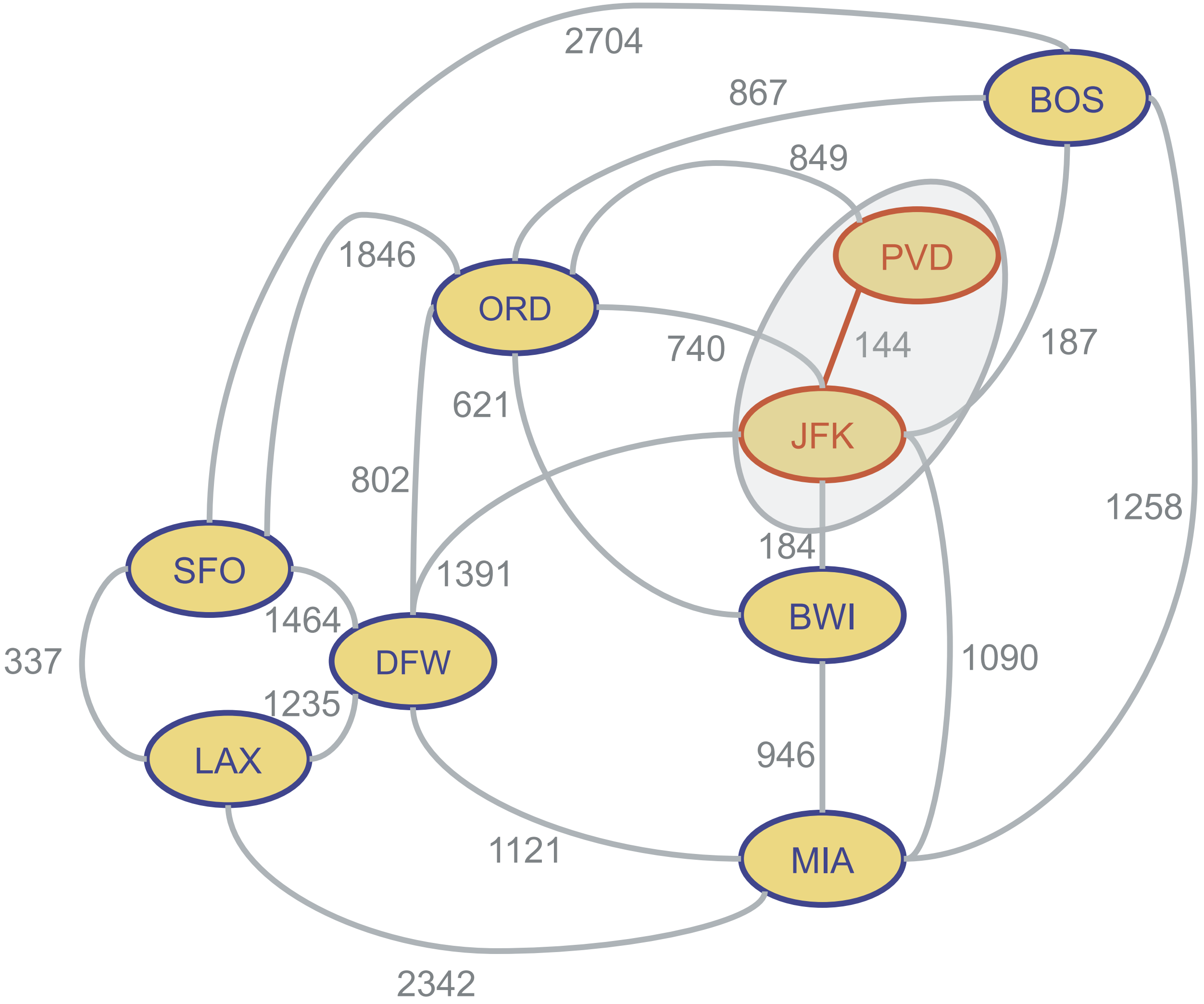
Prim-Jarník Algorithm - Example



PQ	Tree
∞	(BOS, None)
144	(JFK,(PVD, JFK))
∞	(BWI, None)
∞	(MIA, None)
849	(ORD,(PVD, ORD))
∞	(DFW, None)
∞	(SFO, None)
∞	(LAX, None)

- Remove vertex with minimum distance, PVD, from PQ
- Update the length of the paths from PVD to all adjacent vertices that are still in PQ
  - To ORD (was ∞, now 849)
  - To JFK (was ∞, now 144)

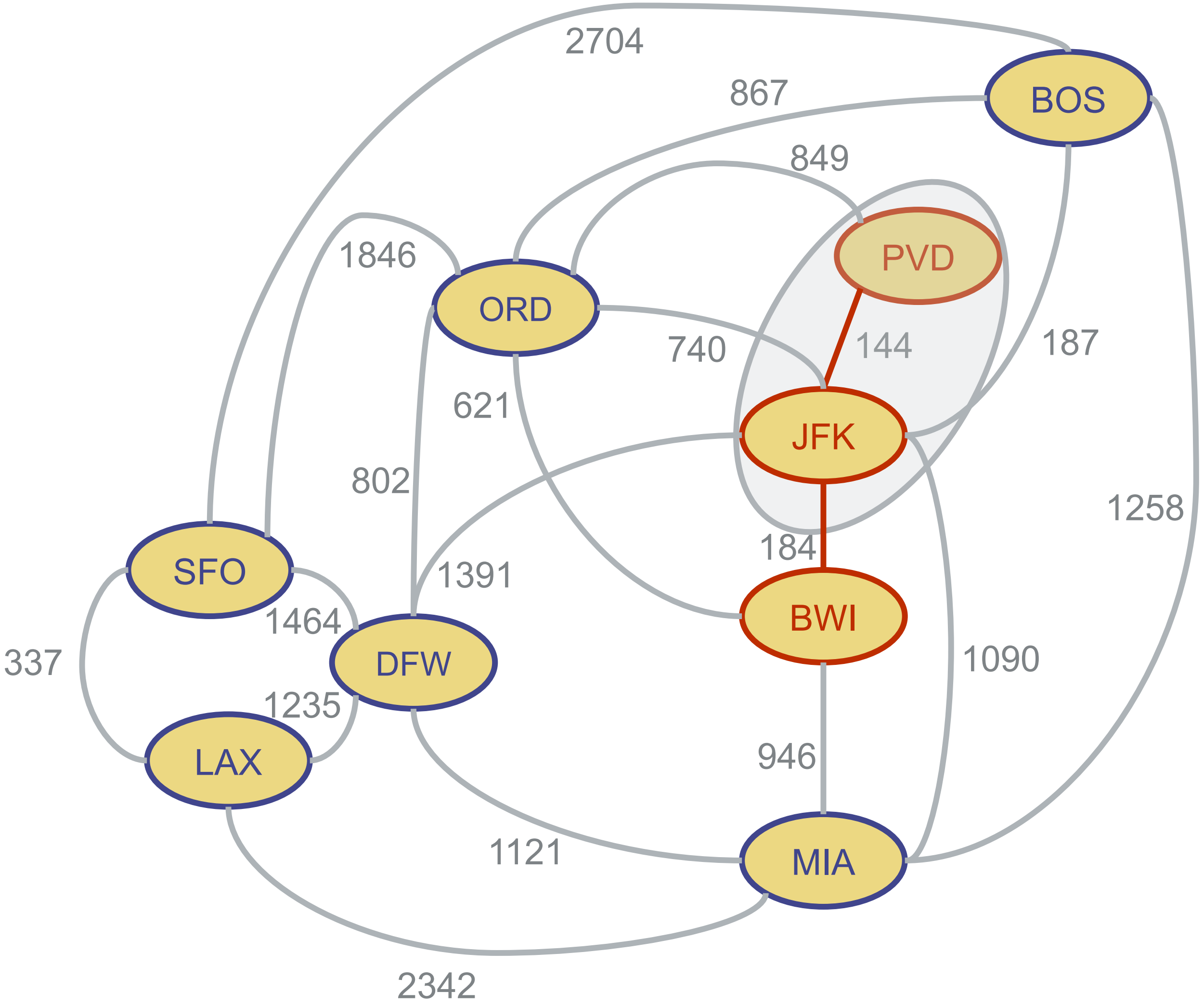
Prim-Jarník Algorithm - Example



	PQ	Tree
187	(BOS,(JFK,BOS))	(PVD, JFK)
184	(BWI,(JFK,BWI))	
1090	(MIA,(JFK, MIA))	
740	(ORD,(JFK, ORD))	
1391	(DFW,(JFK, DFW))	
∞	(SFO, None)	
∞	(LAX, None)	

- Remove vertex with minimum distance, JFK, from PQ
- Add min weight edge (PVD, JFK) to tree
- Update the length of the paths from JFK to all adjacent vertices that are still in PQ
  - To ORD (was 849, now 740)
  - To BOS (was ∞, now 187)
  - To MIA (was ∞, now 1090)
  - To DFW (was ∞, now 1391)
  - To BWI (was ∞, now 184)

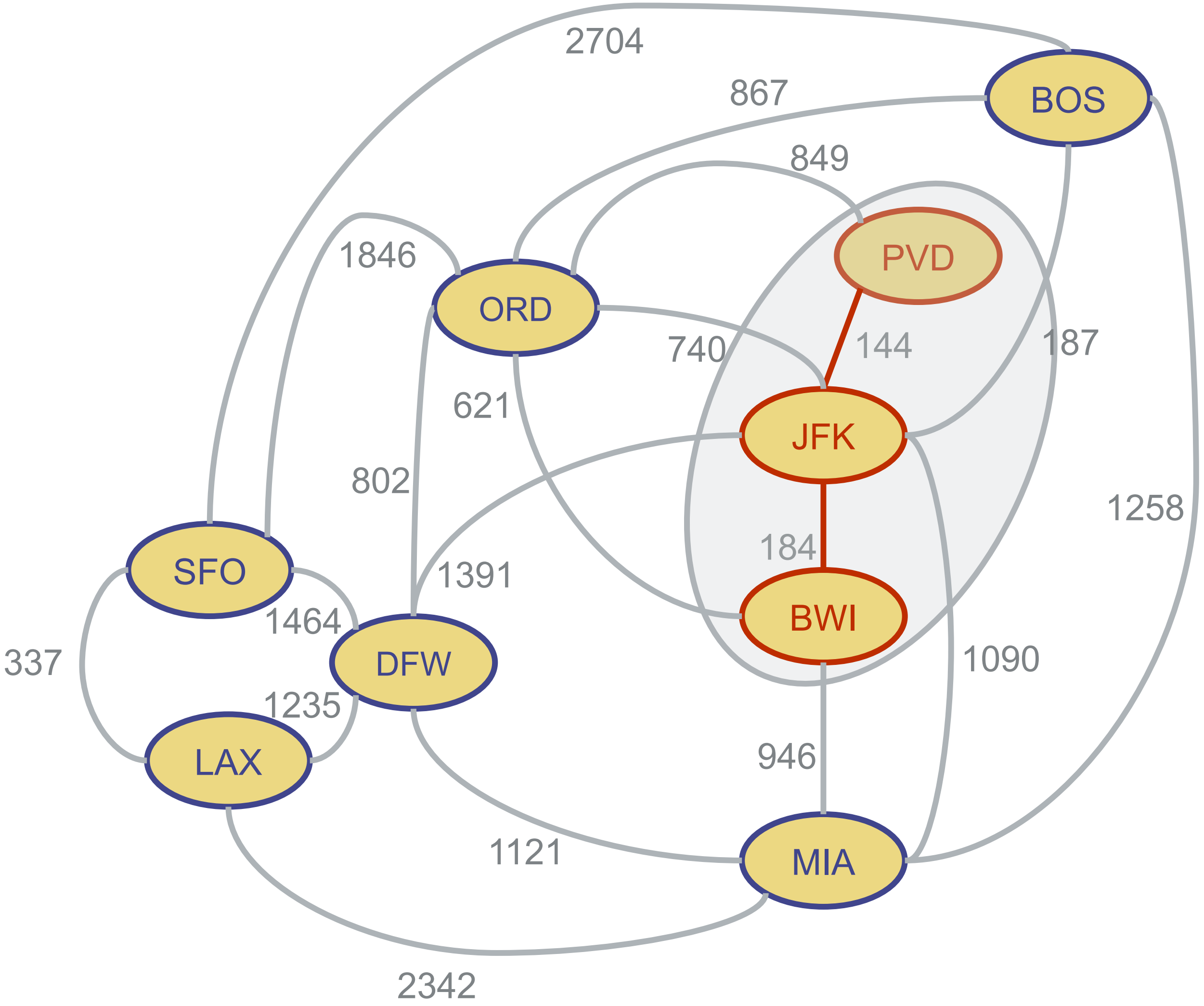
Prim-Jarník Algorithm - Example



	PQ	Tree
187	(BOS,(JFK,BOS))	(PVD, JFK)
1090	(MIA,(JFK, MIA))	(JFK, BWI)
740	(ORD,(JFK, ORD))	
1391	(DFW,(JFK, DFW))	
∞	(SFO, None)	
∞	(LAX, None)	

- Remove vertex with minimum distance, BWI, from PQ
- Add min weight edge (JFK, BWI) to tree

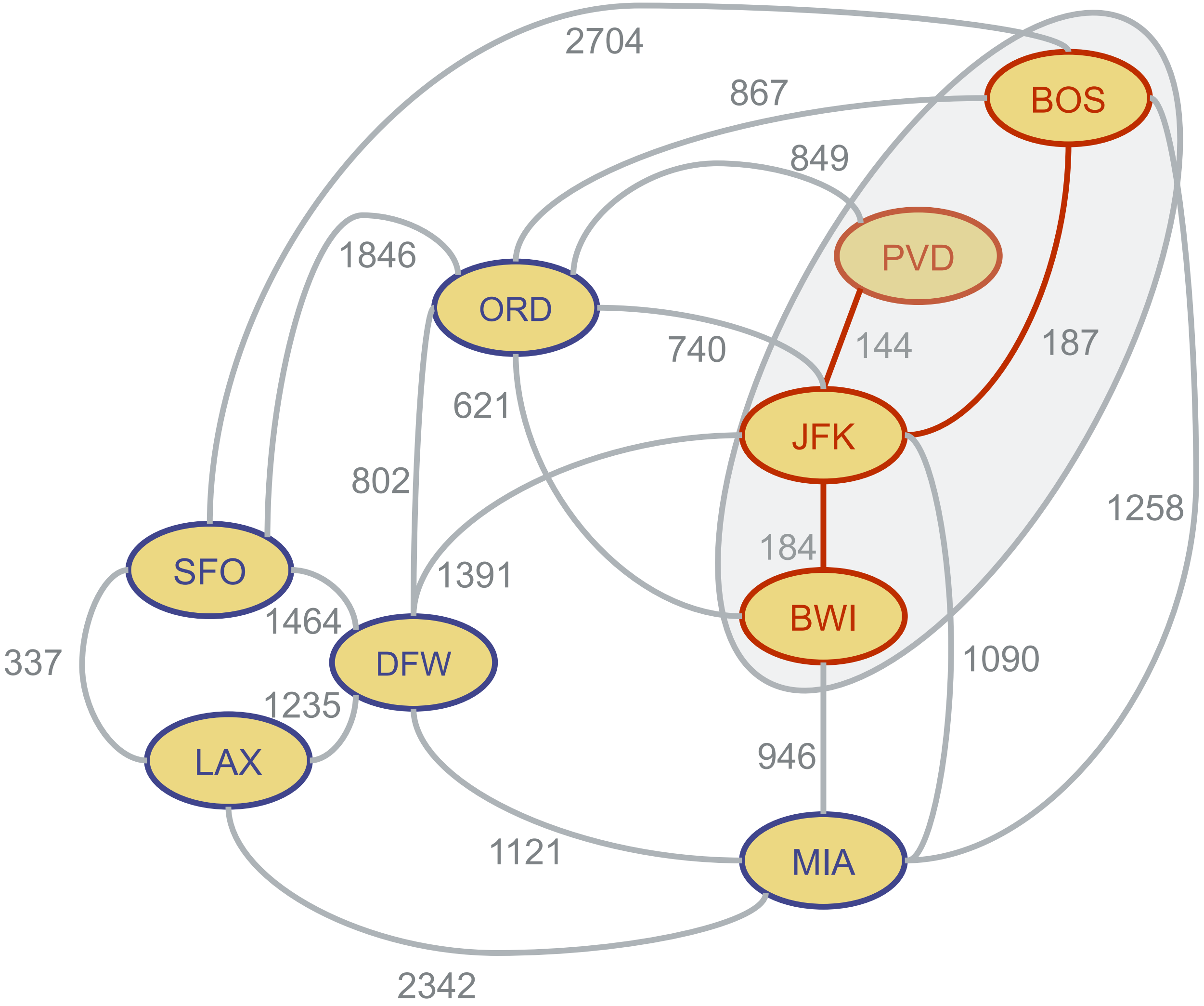
Prim-Jarník Algorithm - Example



	PQ	Tree
187	(BOS,(JFK,BOS))	(PVD, JFK)
946	(MIA,(BWI, MIA))	(JFK, BWI)
621	(ORD,(BWI, ORD))	
1391	(DFW,(JFK, DFW))	
∞	(SFO, None)	
∞	(LAX, None)	

- Update the length of the paths from BWI to all adjacent vertices that are still in PQ
  - To ORD (was 740, now 621)
  - To MIA (was 1090, now 946)

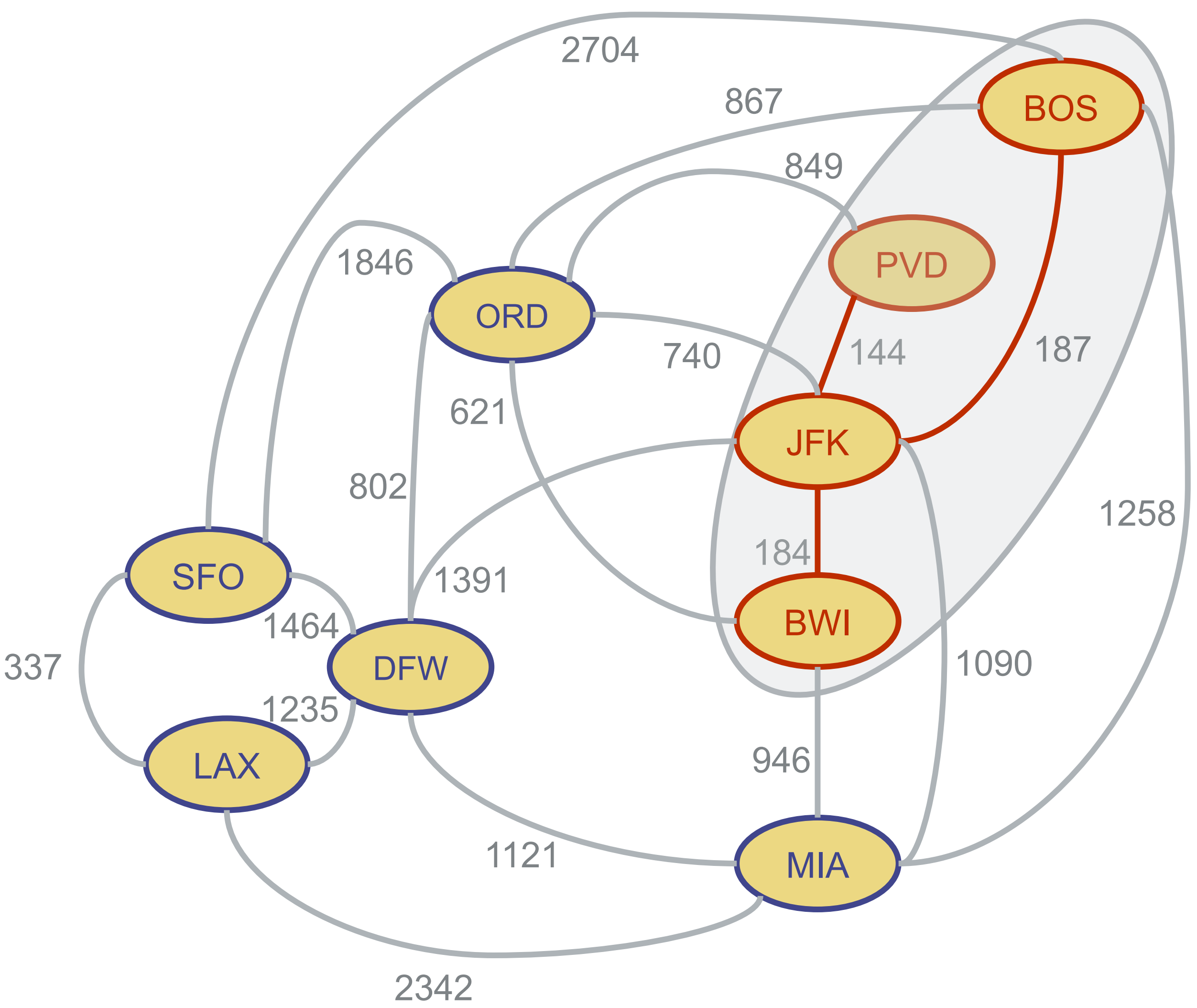
Prim-Jarník Algorithm - Example



	PQ	Tree
946	(MIA,(BWI, MIA))	(PVD, JFK)
621	(ORD,(BWI, ORD))	(JFK, BWI)
1391	(DFW,(JFK, DFW))	(JFK, BOS)
∞	(SFO, None)	
∞	(LAX, None)	

- Remove vertex with minimum distance, BOS, from PQ
- Add min weight edge (JFK, BOS) to tree

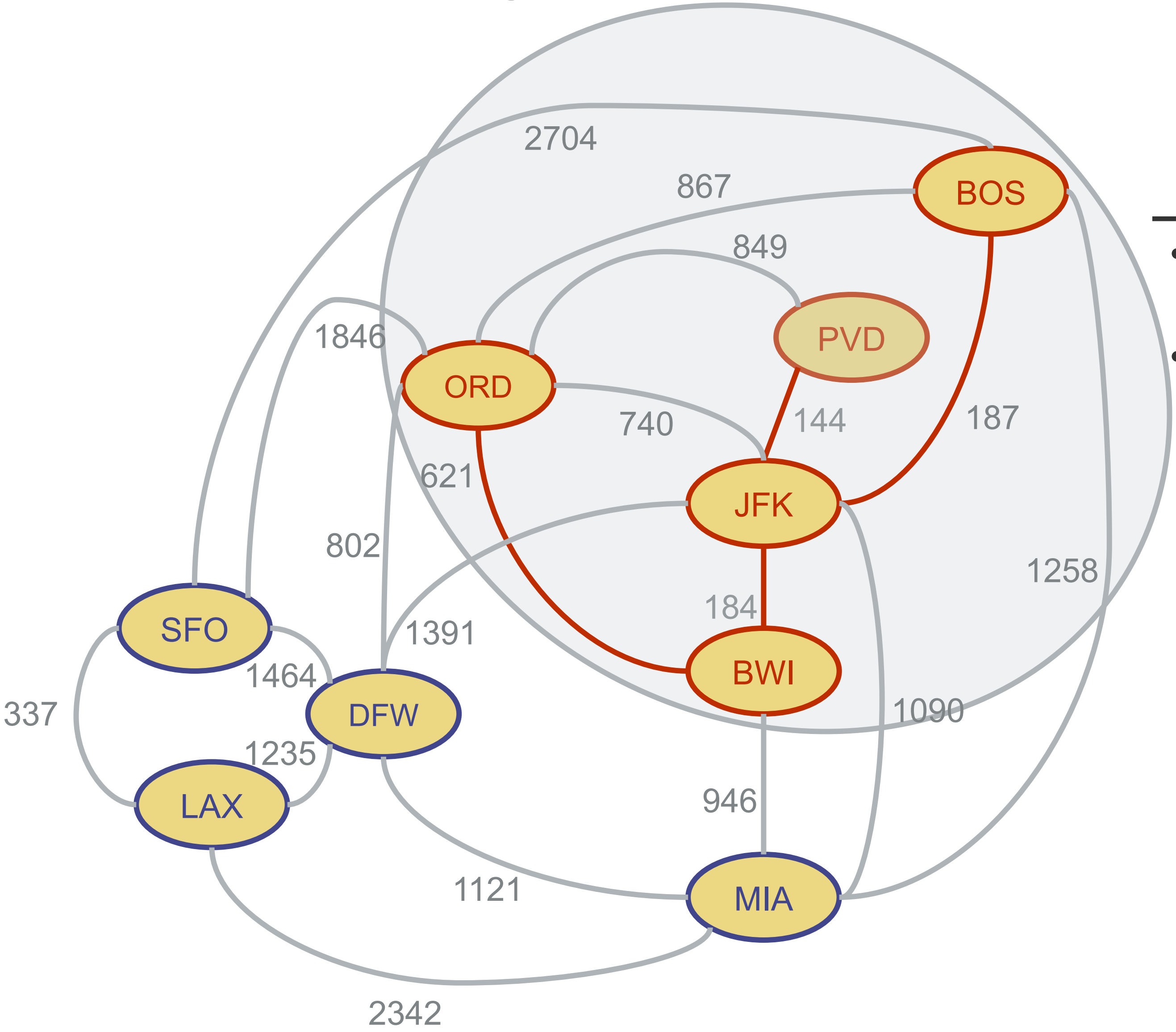
Prim-Jarník Algorithm - Example



	PQ	Tree
946	(MIA,(BWI, MIA))	(PVD, JFK)
621	(ORD,(BWI, ORD))	(JFK, BWI)
1391	(DFW,(JFK, DFW))	(JFK, BOS)
2704	(SFO,(BOS, SFO))	
∞	(LAX, None)	

- Update the length of the paths from BOS to all adjacent vertices that are still in PQ
  - To ORD (was 621, remains – 867 not better)
  - To MIA (was 946, remains – 1258 not better)
  - To SFO (was ∞, now 2704)

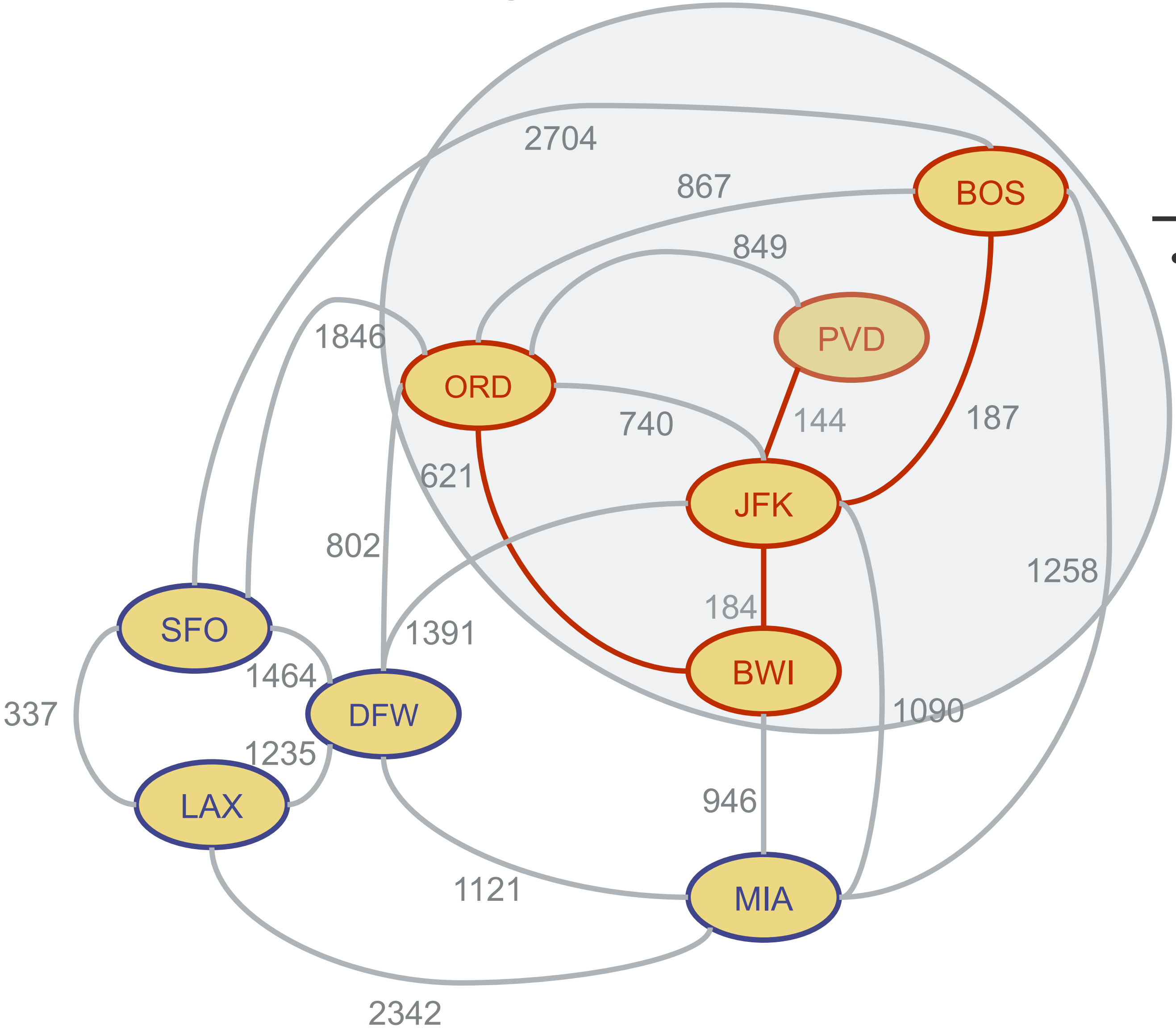
Prim-Jarník Algorithm - Example



	PQ	Tree
946	(MIA,(BWI, MIA))	(PVD, JFK)
1391	(DFW,(JFK, DFW))	(JFK, BWI)
2704	(SFO,(BOS, SFO))	(JFK, BOS)
∞	(LAX, None)	(BWI,ORD)

- Remove vertex with minimum distance, ORD, from PQ
- Add min weight edge (BWI,ORD) to tree

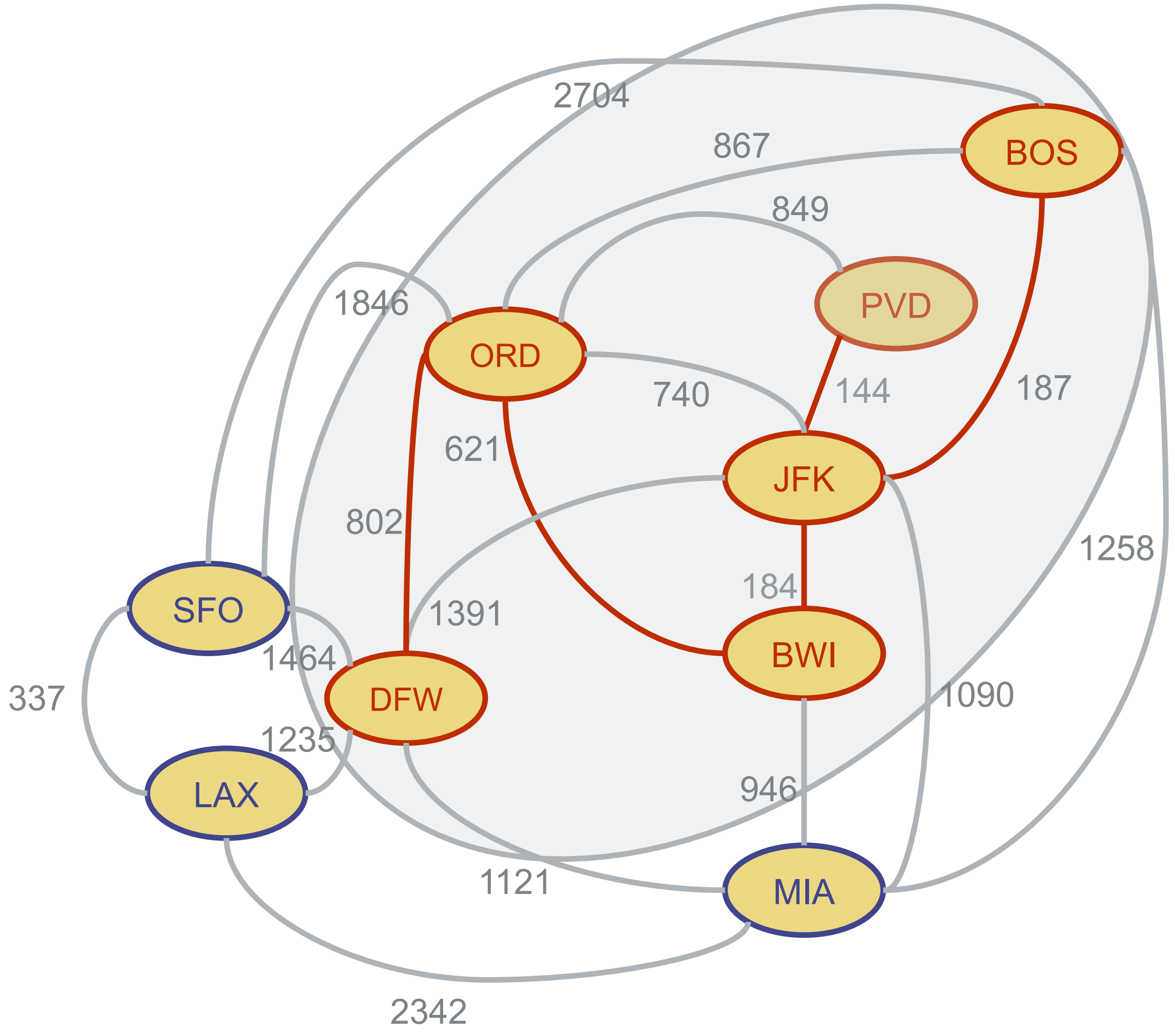
Prim-Jarník Algorithm - Example



	PQ	Tree
946	(MIA,(BWI, MIA))	(PVD, JFK)
802	(DFW,(ORD, DFW))	(JFK, BWI)
1846	(SFO,(BOS, SFO))	(JFK, BOS)
∞	(LAX, None)	(BWI,ORD)

- Update the length of the paths from ORD to all adjacent vertices that are still in PQ
  - To DFW (was 1391, now 802)
  - To SFO (was 2704, now 1846)

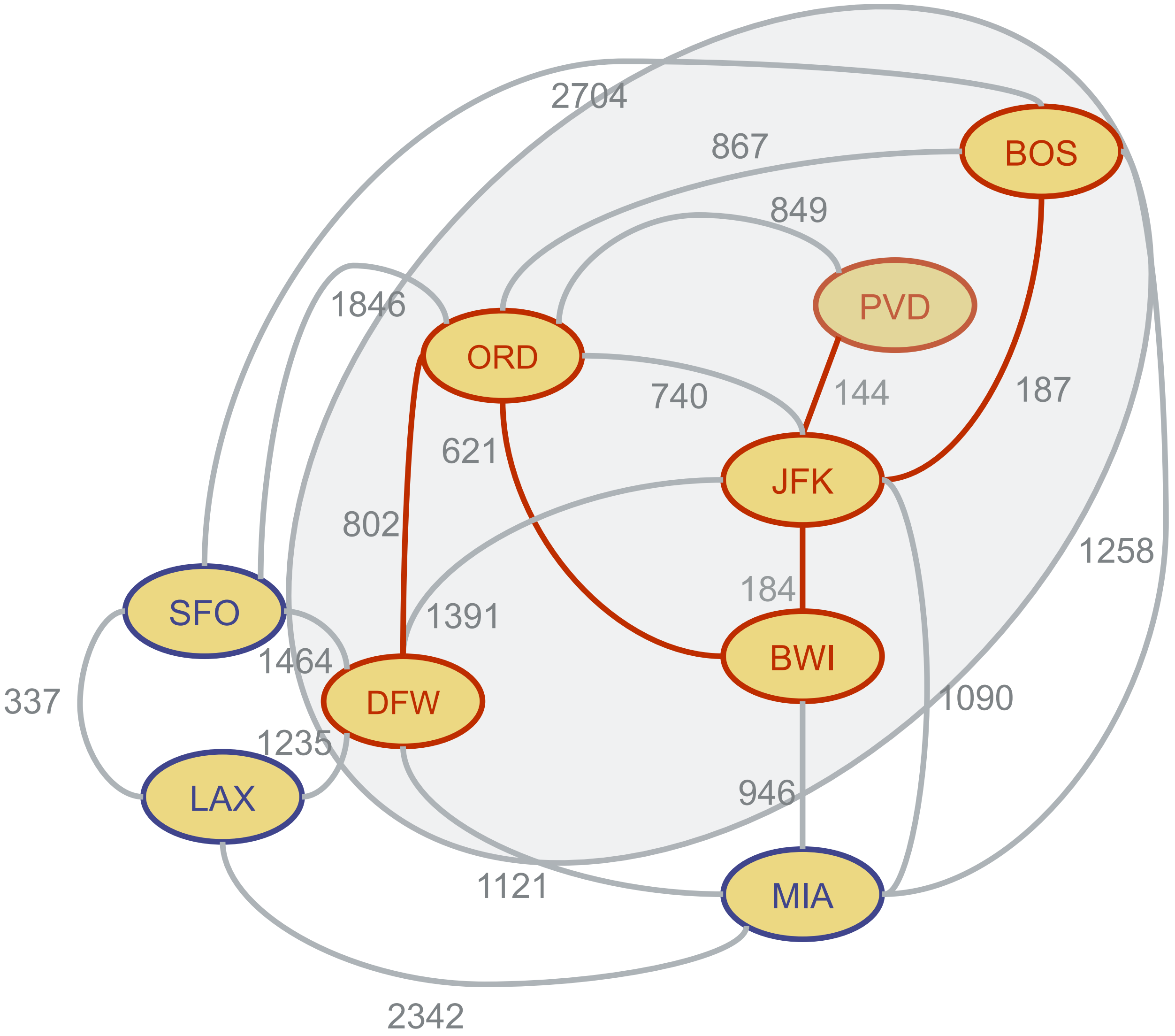
Prim-Jarník Algorithm - Example



	PQ	Tree
946	(MIA,(BWI, MIA))	(PVD, JFK)
1846	(SFO,(BOS, SFO))	(JFK, BWI)
∞	(LAX, None)	(JFK, BOS)
		(BWI,ORD)
		(ORD,DFW)

- Remove vertex with minimum distance, DFW, from PQ
- Add min weight edge (ORD, DFW) to tree

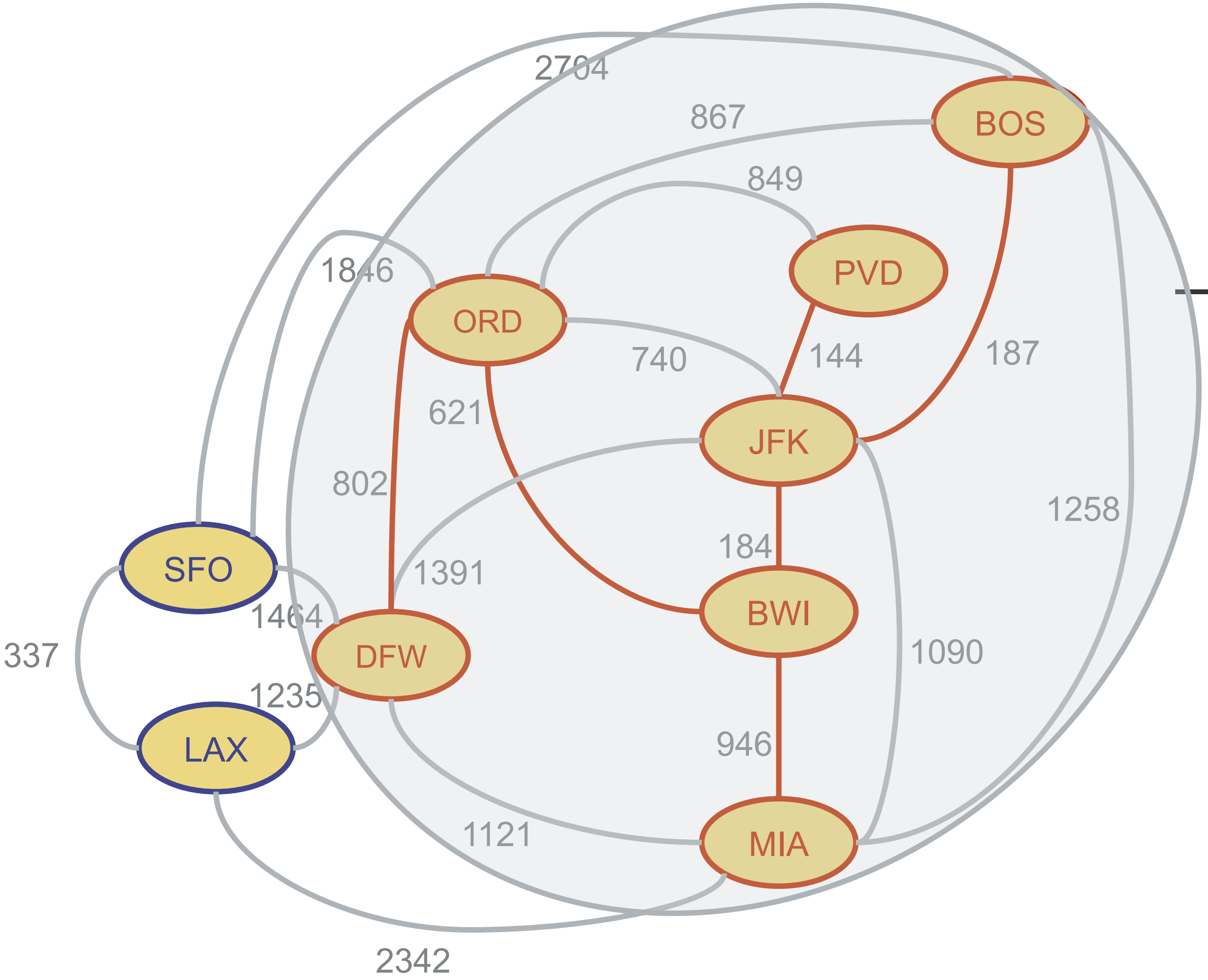
Prim-Jarník Algorithm - Example



	PQ	Tree
946	(MIA,(BWI, MIA))	(PVD, JFK)
1464	(SFO,(DFW, SFO))	(JFK, BWI)
1235	(LAX,(DFW, LAX))	(JFK, BOS)
		(BWI,ORD)
		(ORD,DFW)

- Update the length of the paths from DFW to all adjacent vertices that are still in PQ
  - To MIA (was 946, remains – 1121 not better)
  - To SFO (was 1846, now 1464)
  - To LAX (was ∞, now 1235)

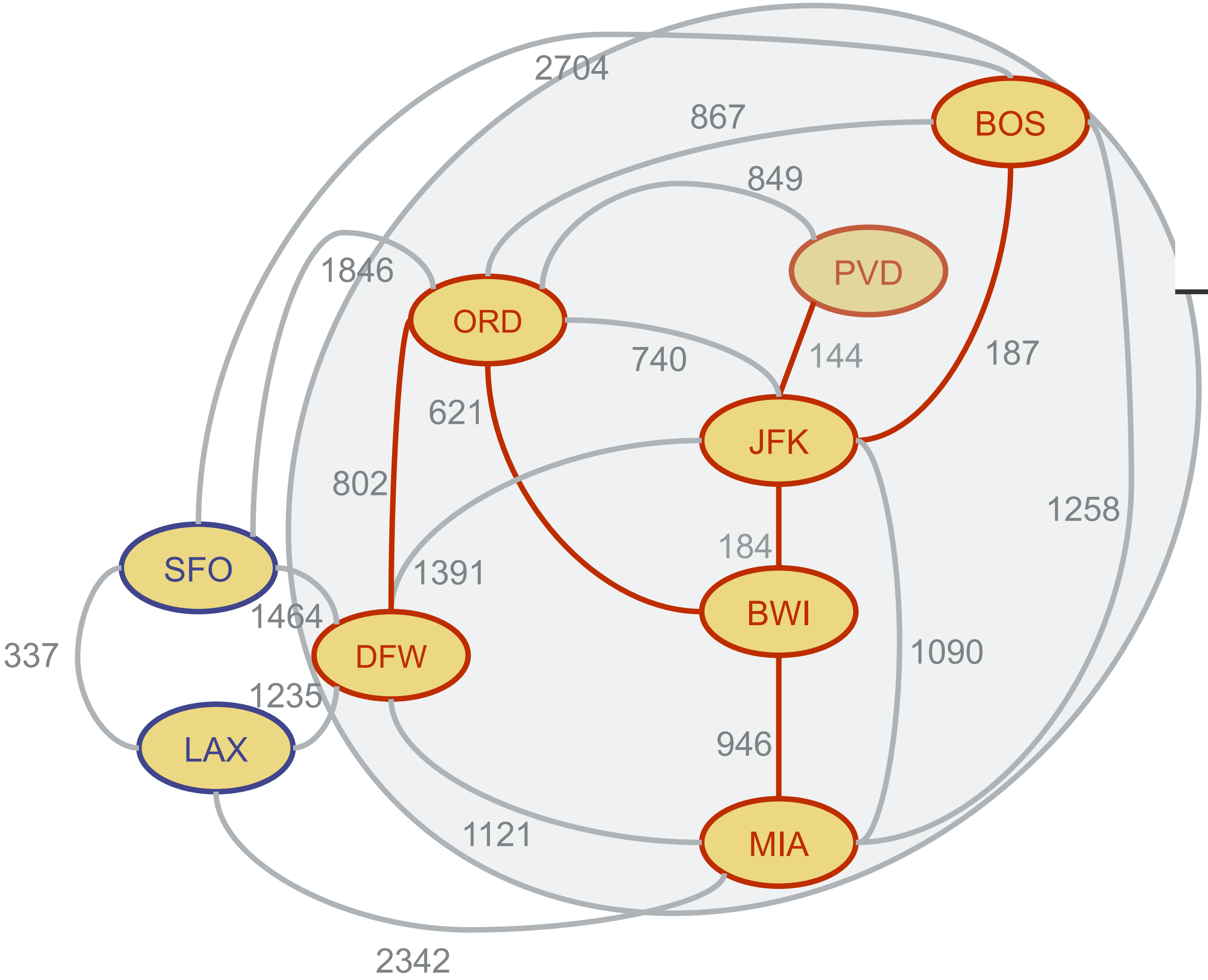
Prim-Jarník Algorithm - Example



	PQ	Tree
1464	(SFO,(DFW, SFO))	(PVD, JFK)
1235	(LAX,(DFW, LAX))	(JFK, BWI)
		(JFK, BOS)
		(BWI,ORD)
		(ORD,DFW)
		(BWI, MIA)

- Remove vertex with minimum distance, MIA, from PQ
- Add min weight edge (BWI, MIA) to tree

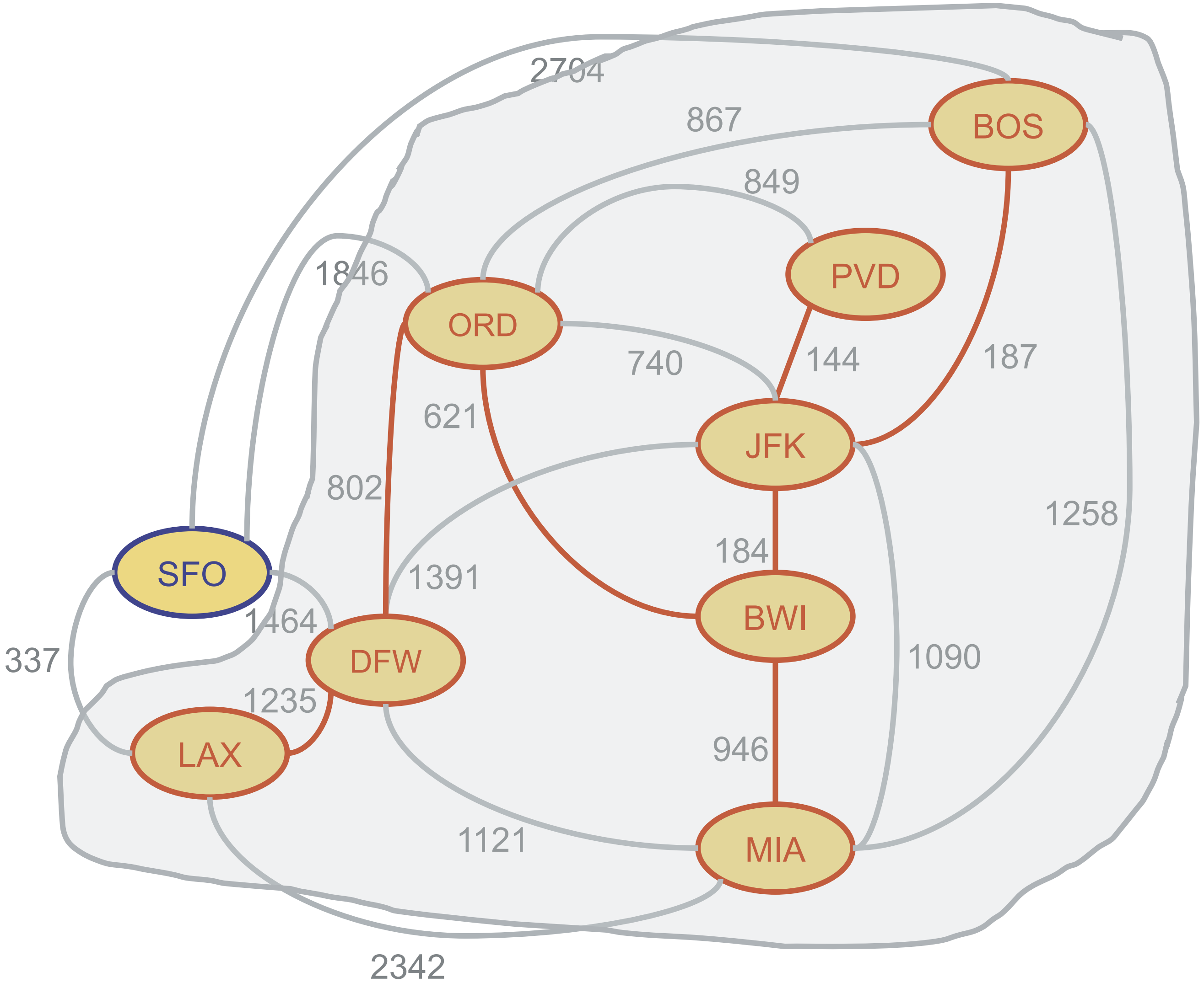
Prim-Jarník Algorithm - Example



	PQ	Tree
1464	(SFO,(DFW, SFO))	(PVD, JFK)
1235	(LAX,(DFW, LAX))	(JFK, BWI)
		(JFK, BOS)
		(BWI,ORD)
		(ORD,DFW)
		(BWI, MIA)

- Update the length of the paths from MIA to all adjacent vertices that are still in PQ
  - To LAX (was 1235 – remains, 2342 is greater)

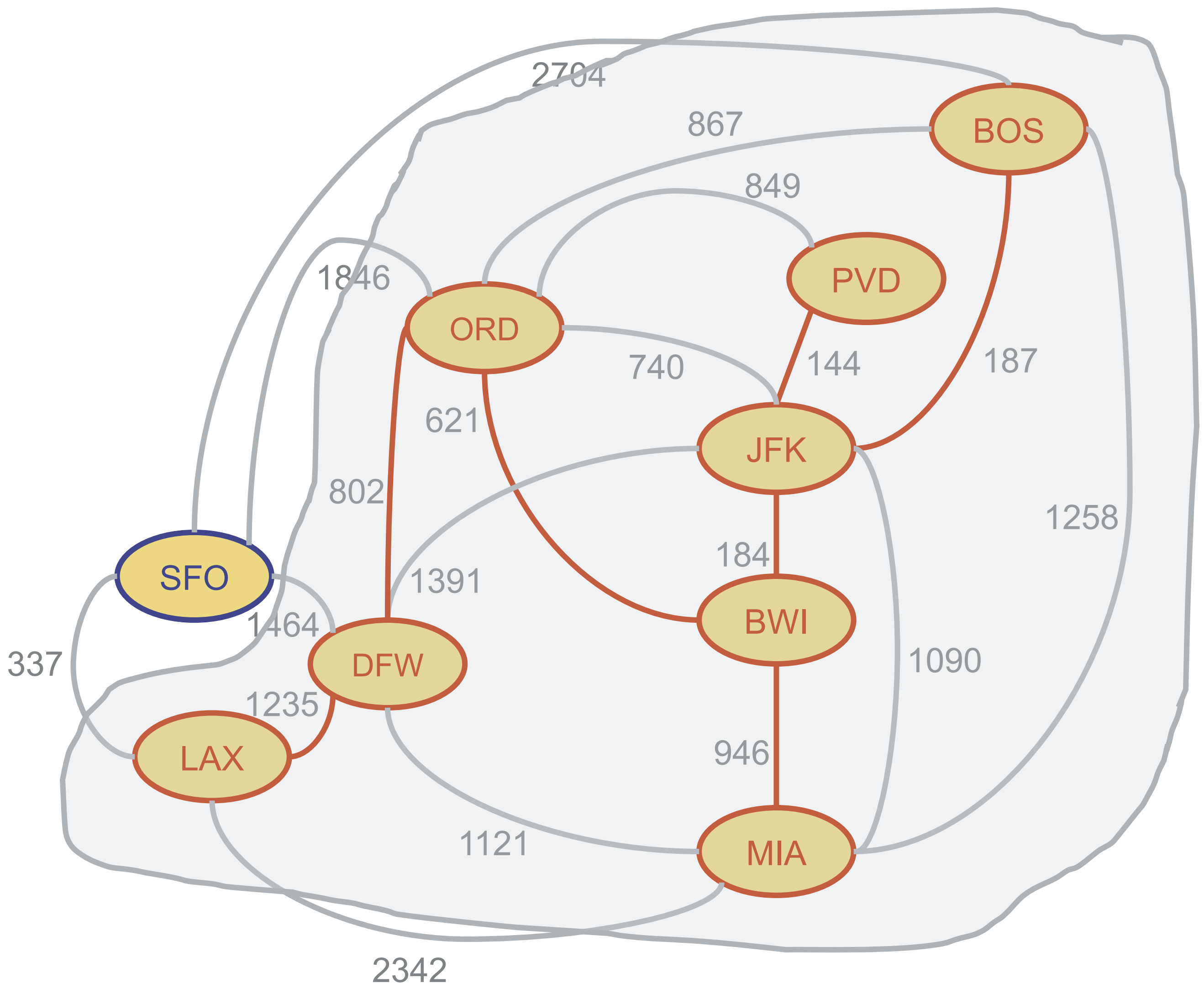
Prim-Jarník Algorithm - Example



	PQ	Tree
1464	(SFO,(DFW, SFO))	(PVD, JFK)
		(JFK, BWI)
		(JFK, BOS)
		(BWI,ORD)
		(ORD,DFW)
		(BWI, MIA)
		(DFW, LAX)

- Remove vertex with minimum distance, LAX, from PQ
- Add min weight edge (DFW, LAX) to tree

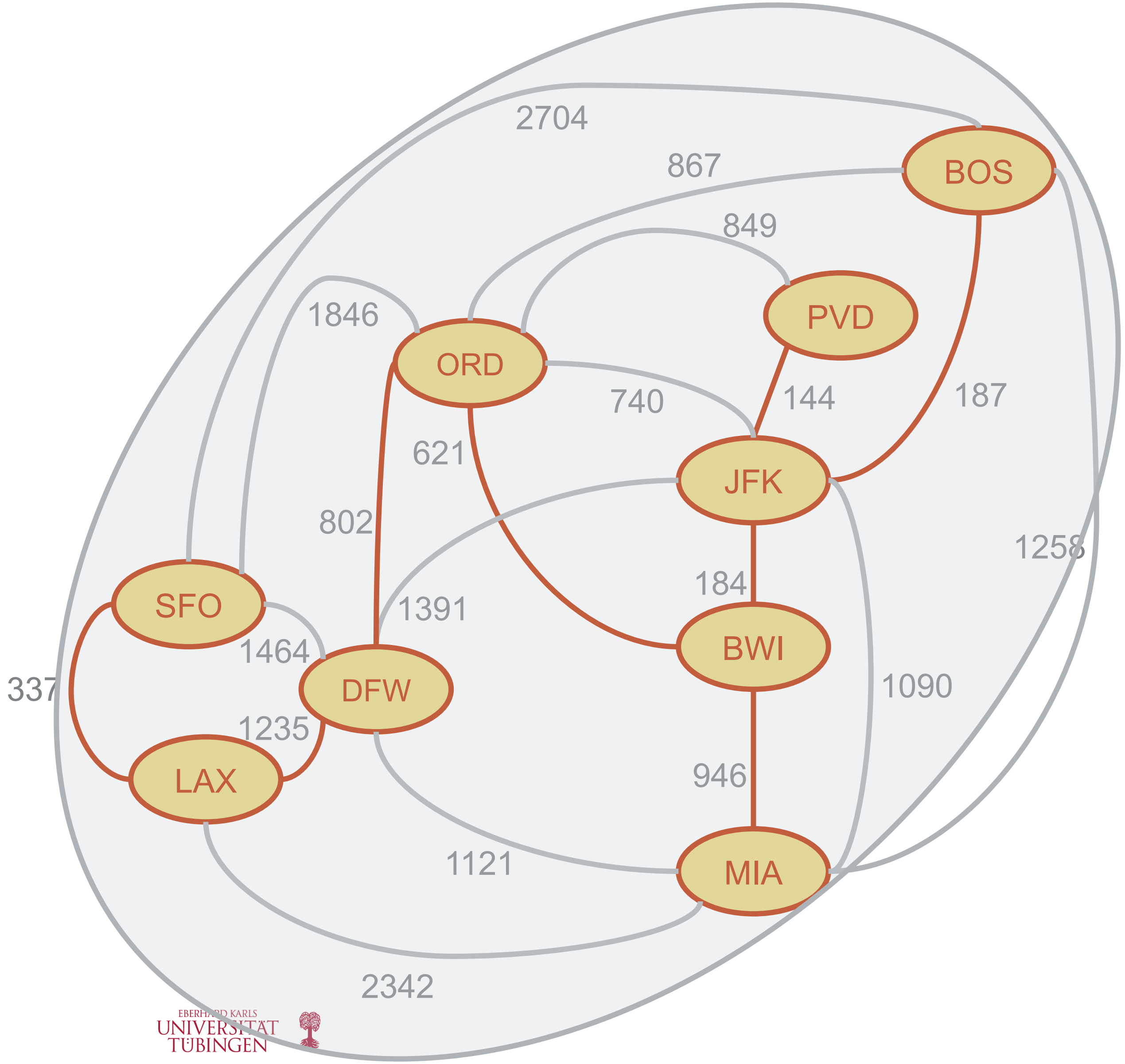
Prim-Jarník Algorithm - Example



	PQ	Tree
337	(SFO,(LAX, SFO))	(PVD, JFK)
		(JFK, BWI)
		(JFK, BOS)
		(BWI,ORD)
		(ORD,DFW)
		(BWI, MIA)
		(DFW, LAX)

- Update the length of the paths from DFW to all adjacent vertices that are still in PQ
  - To SFO (was 1464, now 337)

Prim-Jarník Algorithm - Example



PQ	Tree
	(PVD, JFK)
	(JFK, BWI)
	(JFK, BOS)
	(BWI,ORD)
	(ORD,DFW)
	(BWI, MIA)
	(DFW, LAX)
	(LAX, SFO)

- Remove vertex with minimum distance, SFO, from PQ
- Add min weight edge (LAX, SFO) to tree
- No more edges in the PQ, STOP.

Prim-Jarník Algorithm – Running Time Analysis

- The implementation of the algorithm relies, just like Dijkstra's algorithm, on the adaptable priority queue
- Initially, all n vertices are added to the PQ - n PQ insertions
- Each vertex is removed from the PQ via a remove_min operation - n PQ remove_min
- Throughout the algorithm, at most m PQ update operations are performed
- With a heap-based PQ, the insert, remove_min and update operations need $O(\log n)$ time
- The overall running time is $O((n+m)\log n)$
- Using an unsorted list implementation of a priority queue the algorithm achieves an $O(n^2)$ running time

Prim-Jarník Algorithm – Python Implementation

```
def MST_PrimJarnik(g):
  """Compute a minimum spanning tree of weighted graph g.

  Return a list of edges that comprise the MST (in arbitrary order).
  """
  d = { }                             # d[v] is bound on distance to tree
  tree = [ ]                          # list of edges in spanning tree
  pq = AdaptableHeapPriorityQueue( )  # d[v] maps to value (v, e=(u,v))
  pqlocator = { }                     # map from vertex to its pq locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices( ):
    if len(d) == 0:                   # this is the first node
      d[v] = 0                        # make it the root
    else:
      d[v] = float('inf')             # positive infinity
    pqlocator[v] = pq.add(d[v], (v,None))

  while not pq.is_empty( ):
    key,value = pq.remove_min( )
    u,edge = value                    # unpack tuple from pq
    del pqlocator[u]                  # u is no longer in pq
    if edge is not None:
      tree.append(edge)               # add edge to tree
    for link in g.incident_edges(u):
      v = link.opposite(u)
      if v in pqlocator:              # thus v not yet in tree
        # see if edge (u,v) better connects v to the growing tree
        wgt = link.element( )
        if wgt < d[v]:                # better edge to v?
          d[v] = wgt                  # update the distance
          pq.update(pqlocator[v], d[v], (v, link))   # update the pq entry
  return tree
```

Kruskal's Algorithm

Kruskal's Algorithm - Intuition

- In contrast to the Prim-Jarník algorithm, which grows an MST from a single starting vertex, Kruskal's algorithm maintains a forest of clusters – repeatedly merges pairs of clusters until a single cluster spans the graph
- Initially, each vertex is by itself in a cluster
- For each edge, edges considered in order of increasing weight:
  - If an edge connects two clusters, then add e to the set of edges of the MST and merge the clusters
  - If e connects two vertices from the same cluster, discard e
- The algorithm terminates when it has found enough edges to form a MST
- For a graph with n vertices, $n - 1$ edges are needed to form a MST

Kruskal's Algorithm - Pseudocode

Algorithm Kruskal(G):

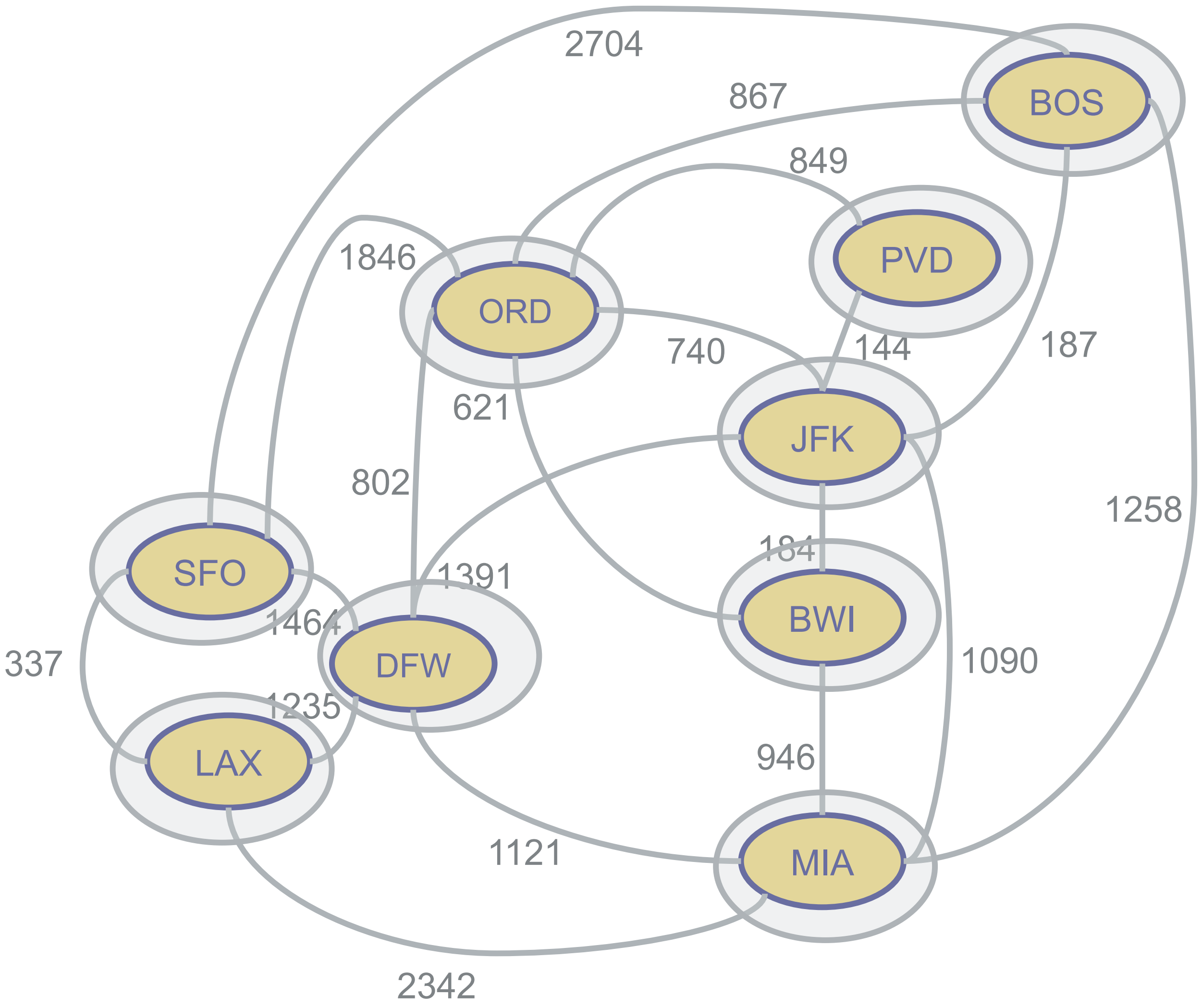
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G

for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n-1$ edges **do**
  (u,v) = value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u,v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

Kruskal's Algorithm – Why It Works

- The correctness of Kruskal's algorithm is based, again, on the proposition from the introduction
- Each time an edge $e = (u, v)$ is added to the MST, a partitioning of the vertices in V can be constructed having the cluster containing v on one side (V_1), and a cluster containing the rest of the vertices in V on the other side (V_2)
- This defines a disjoint partitioning of the vertices of V
- Since edges are considered in increasing weight order, an edge e with an endpoint in V_1 and another endpoint in V_2 must be a minimum-weight edge – thus Kruskal's algorithm will always add a valid edge to the MST

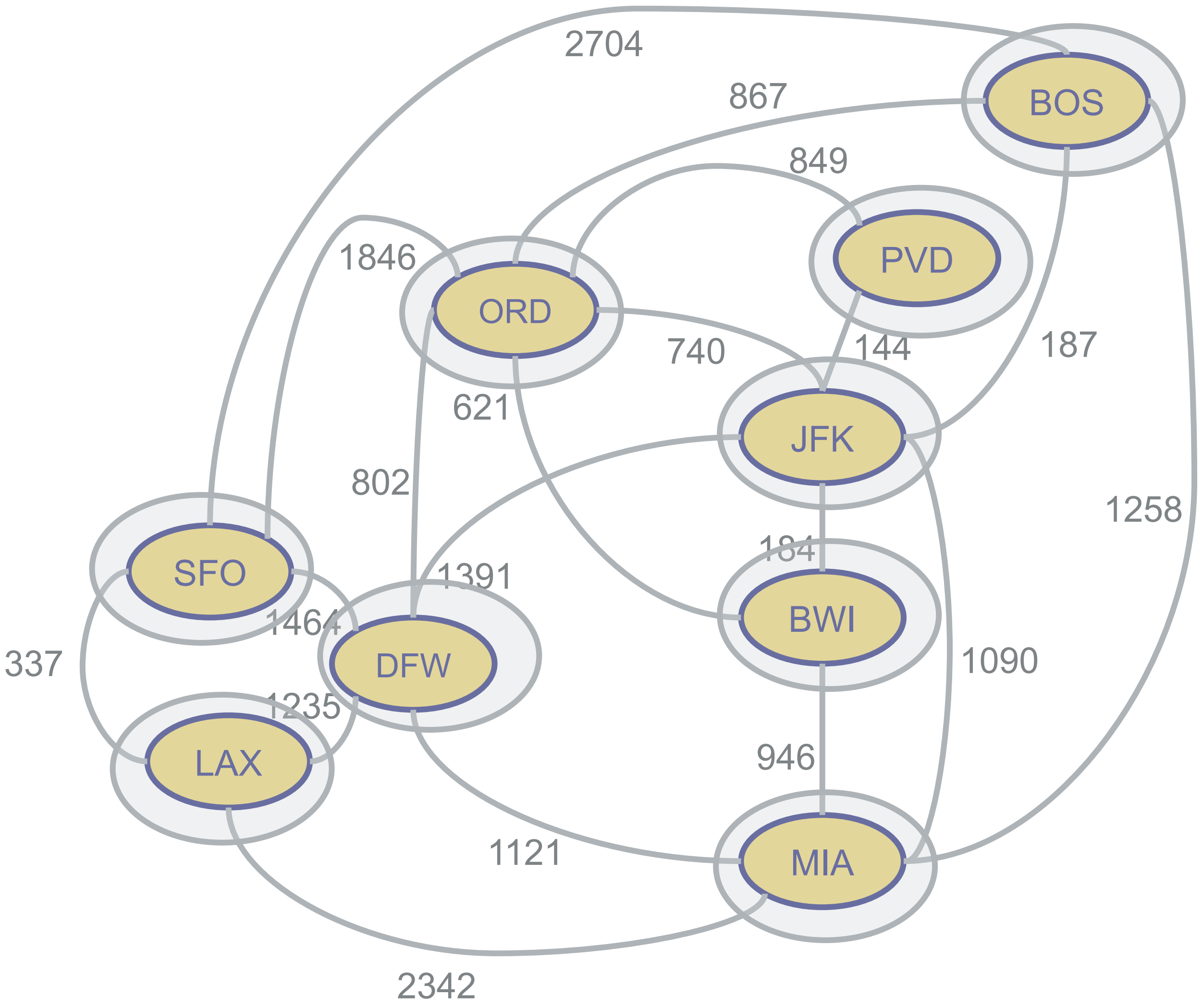
Kruskal's Algorithm – Example



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

- Initially, every node is in its own cluster

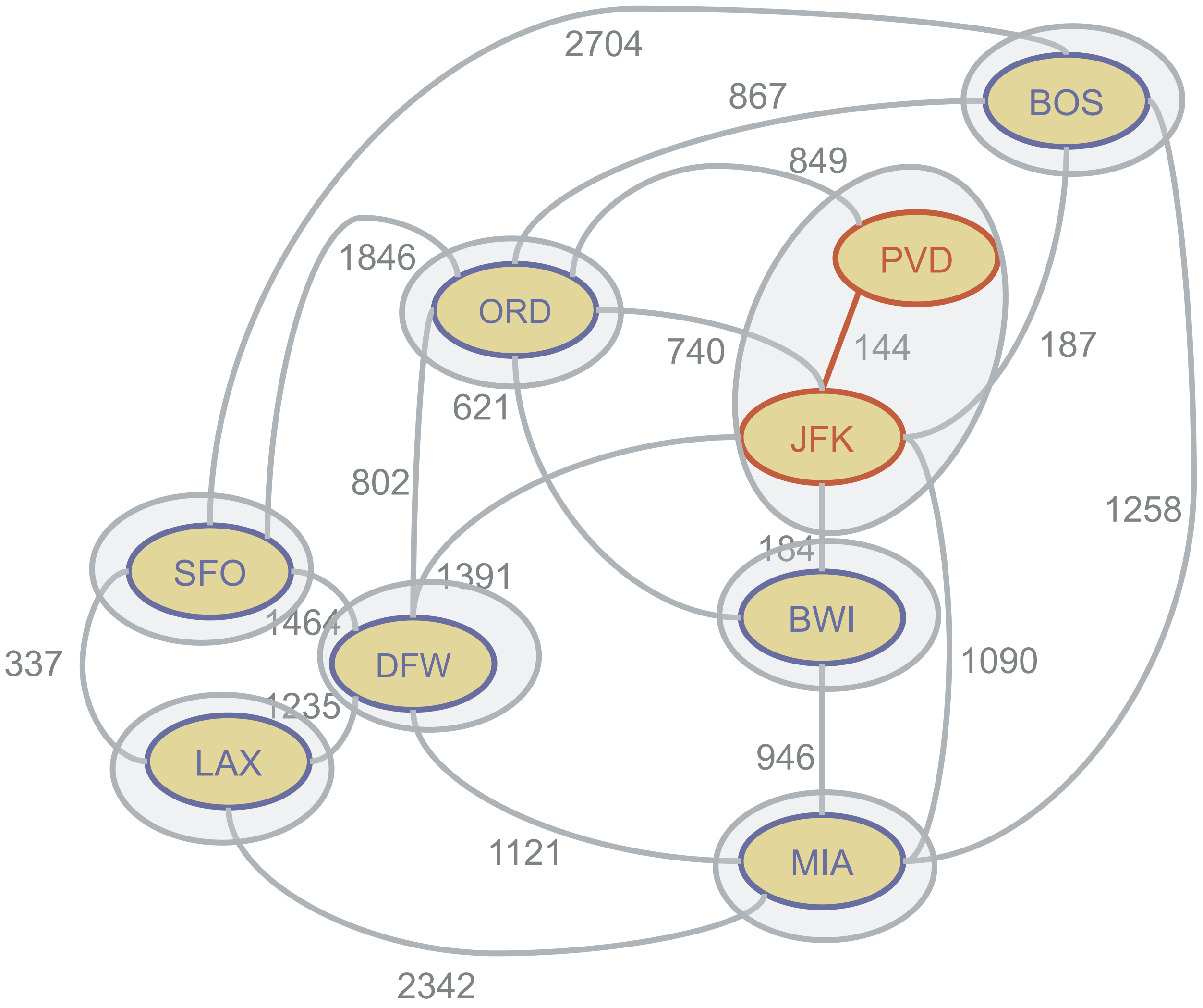
Kruskal’s Algorithm – Example



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

- Remove the minimum weight edge, (JFK, PVD), from PQ, add it to the tree, join clusters

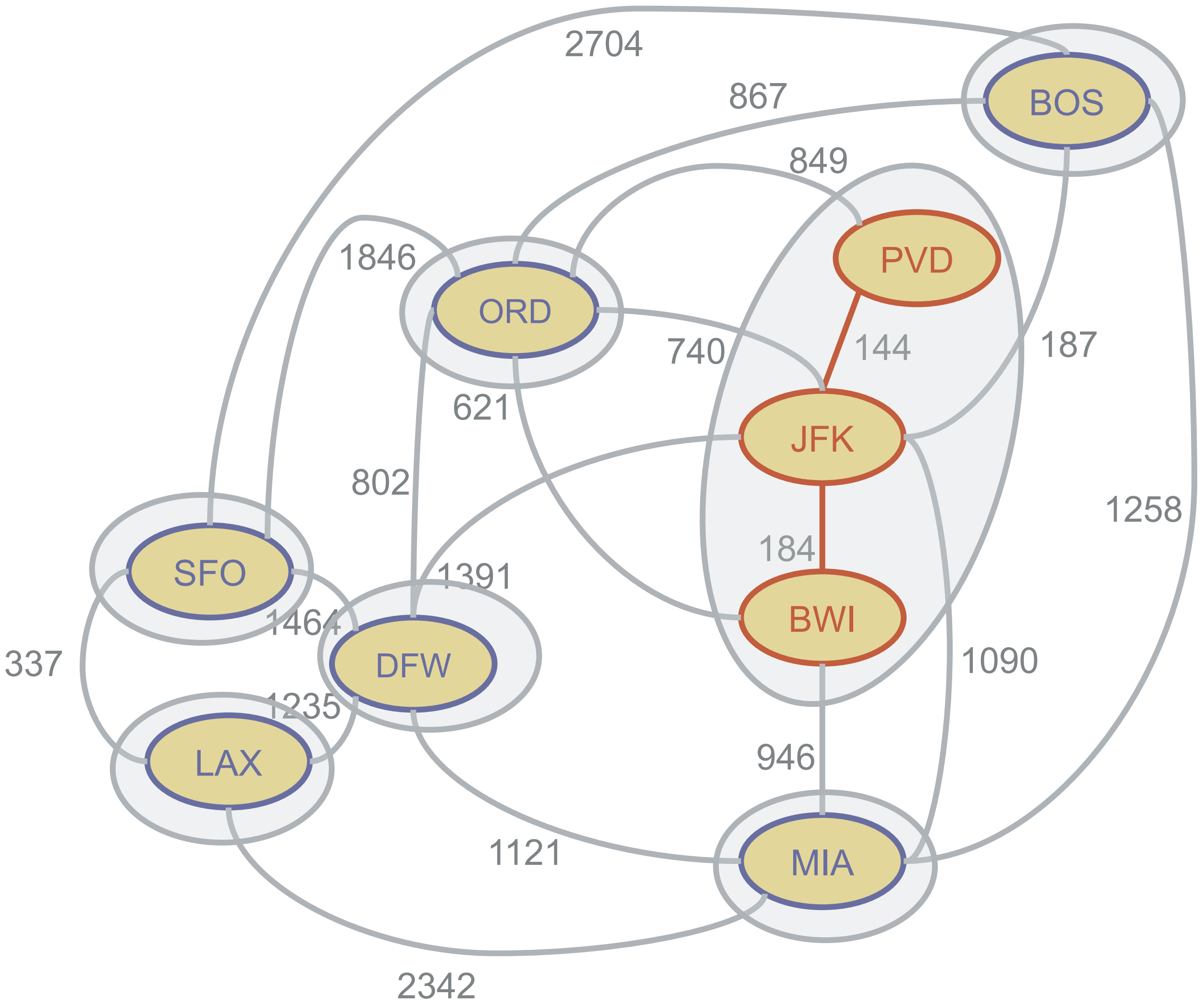
Kruskal's Algorithm – Example



	PQ	Tree
184	(BWI, JFK)	(JFK, PVD)
187	(JFK, BOS)	
337	(SFO, LAX)	
621	(BWI, ORD)	
740	(ORD, JFK)	
802	(DFW, ORD)	
849	(ORD, PWD)	
867	(ORD, BOS)	
946	(MIA, BWI)	
1090	(MIA, JFK)	
1121	(DFW, MIA)	
1235	(LAX, DFW)	
1258	(MIA, BOS)	
1391	(DFW, JFK)	
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (JFK, PVD), from PQ, add it to the tree, join clusters

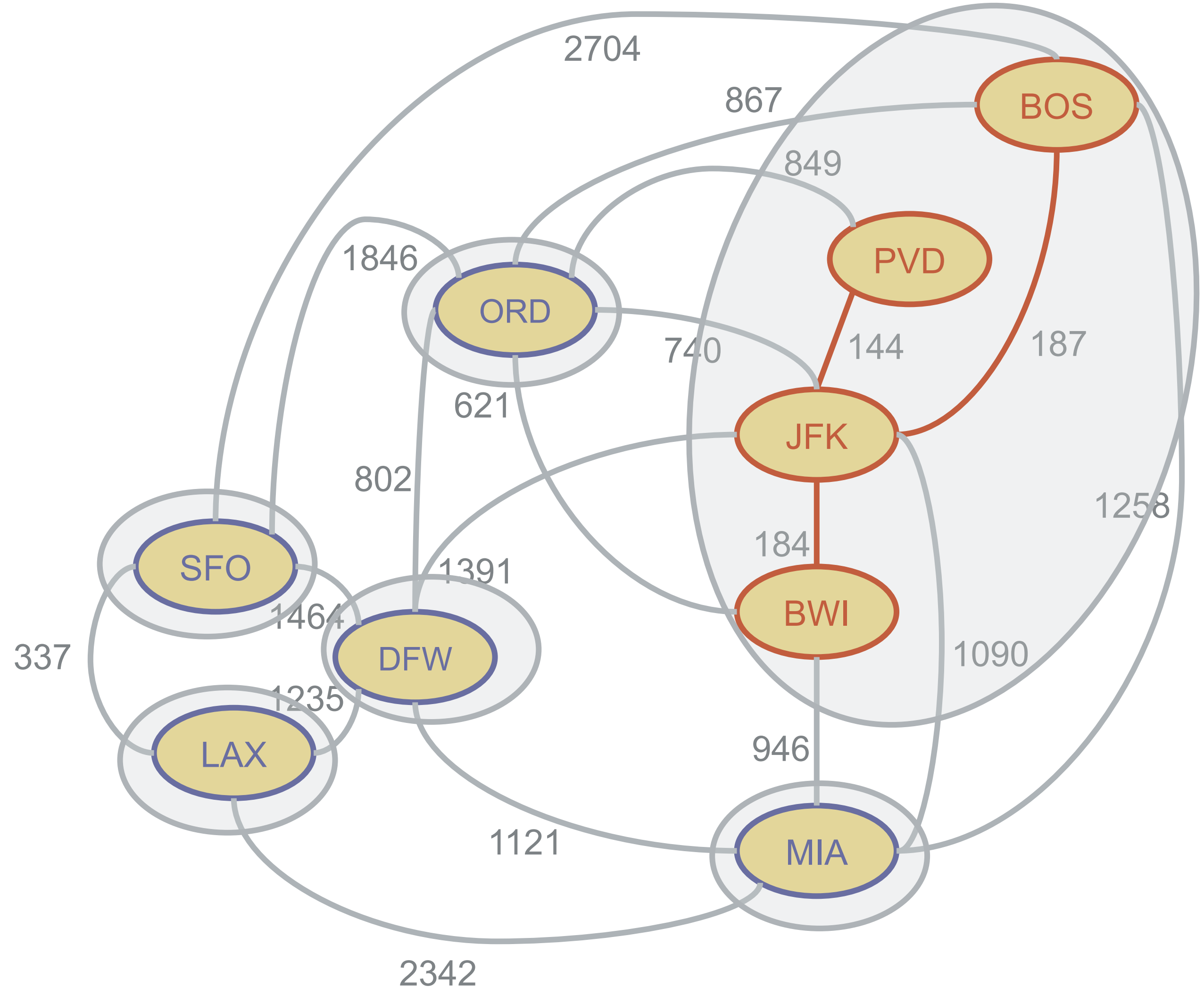
Kruskal's Algorithm – Example



	PQ	Tree
187	(JFK, BOS)	(JFK, PVD)
337	(SFO, LAX)	(BWI, JFK)
621	(BWI, ORD)	
740	(ORD, JFK)	
802	(DFW, ORD)	
849	(ORD, PWD)	
867	(ORD, BOS)	
946	(MIA, BWI)	
1090	(MIA, JFK)	
1121	(DFW, MIA)	
1235	(LAX, DFW)	
1258	(MIA, BOS)	
1391	(DFW, JFK)	
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (BWI, JFK), from PQ, add it to the tree, join clusters

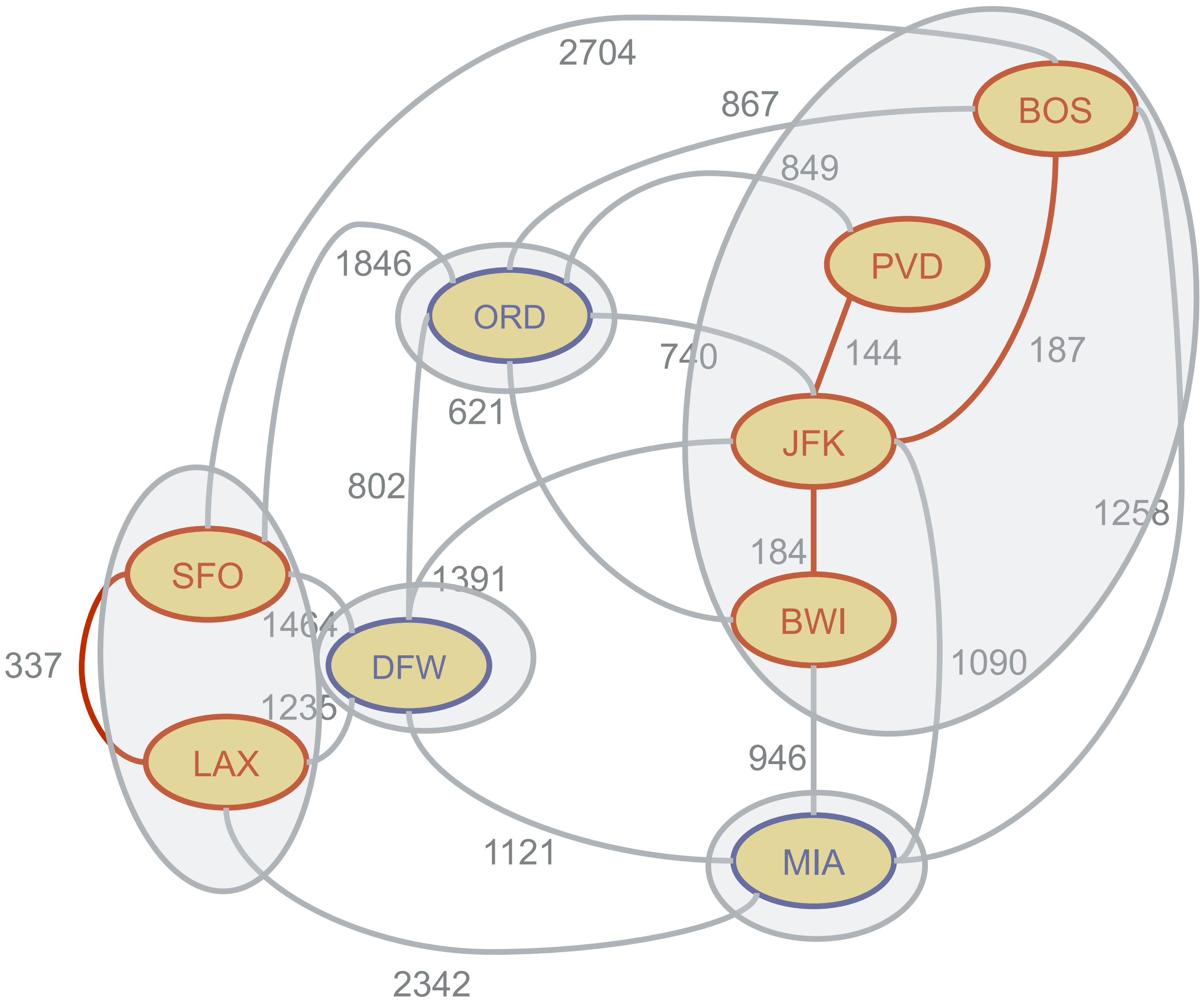
Kruskal's Algorithm – Example



	PQ	Tree
337	(SFO, LAX)	(JFK, PVD)
621	(BWI, ORD)	(BWI, JFK)
740	(ORD, JFK)	(JFK, BOS)
802	(DFW, ORD)	
849	(ORD, PWD)	
867	(ORD, BOS)	
946	(MIA, BWI)	
1090	(MIA, JFK)	
1121	(DFW, MIA)	
1235	(LAX, DFW)	
1258	(MIA, BOS)	
1391	(DFW, JFK)	
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (JFK,BOS), from PQ, add it to the tree, join clusters

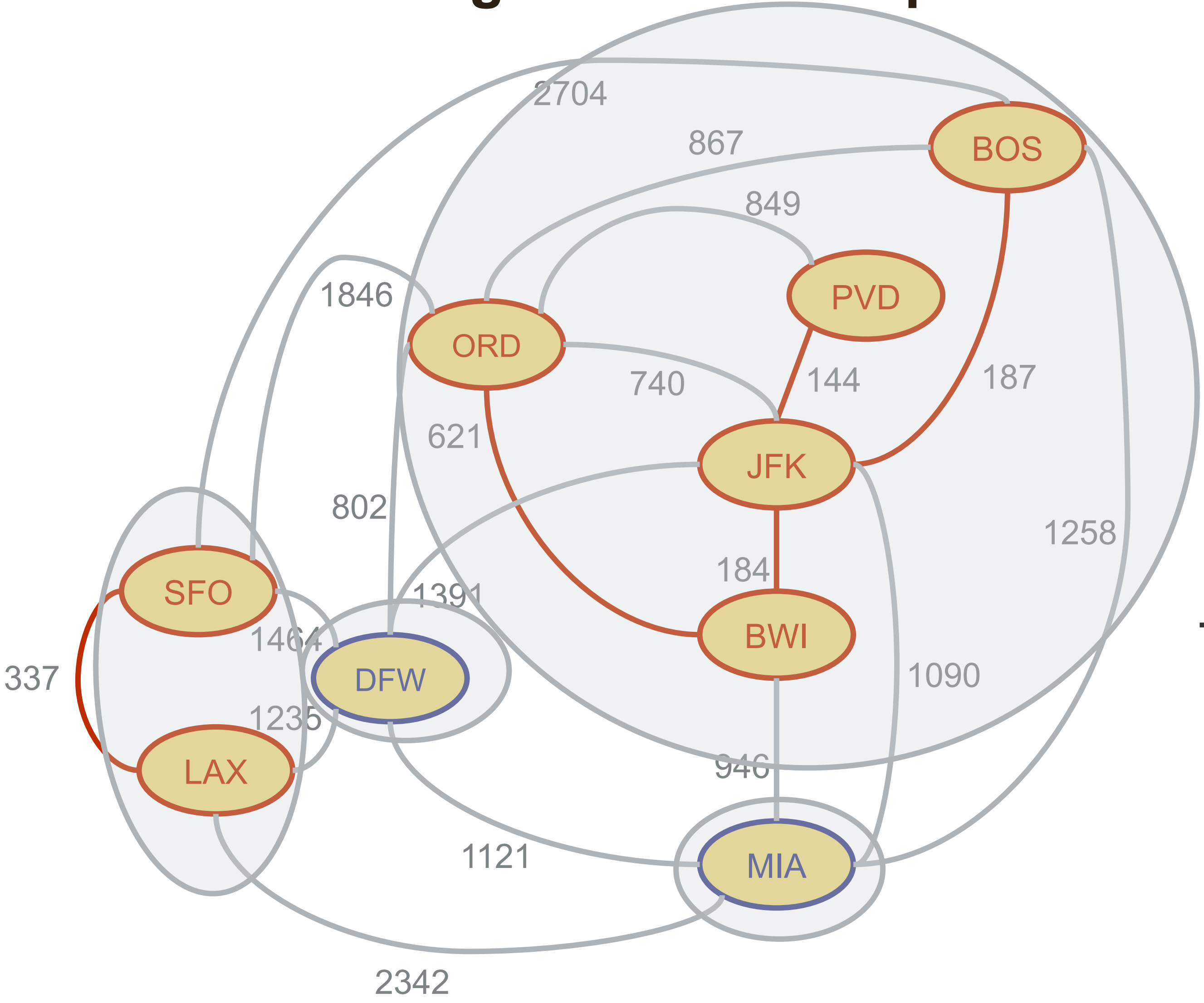
Kruskal's Algorithm – Example



	PQ	Tree
621	(BWI, ORD)	(JFK, PVD)
740	(ORD, JFK)	(BWI, JFK)
802	(DFW, ORD)	(JFK, BOS)
849	(ORD, PWD)	(SFO, LAX)
867	(ORD, BOS)	
946	(MIA, BWI)	
1090	(MIA, JFK)	
1121	(DFW, MIA)	
1235	(LAX, DFW)	
1258	(MIA, BOS)	
1391	(DFW, JFK)	
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (SFO,LAX), from PQ, add it to the tree, join clusters

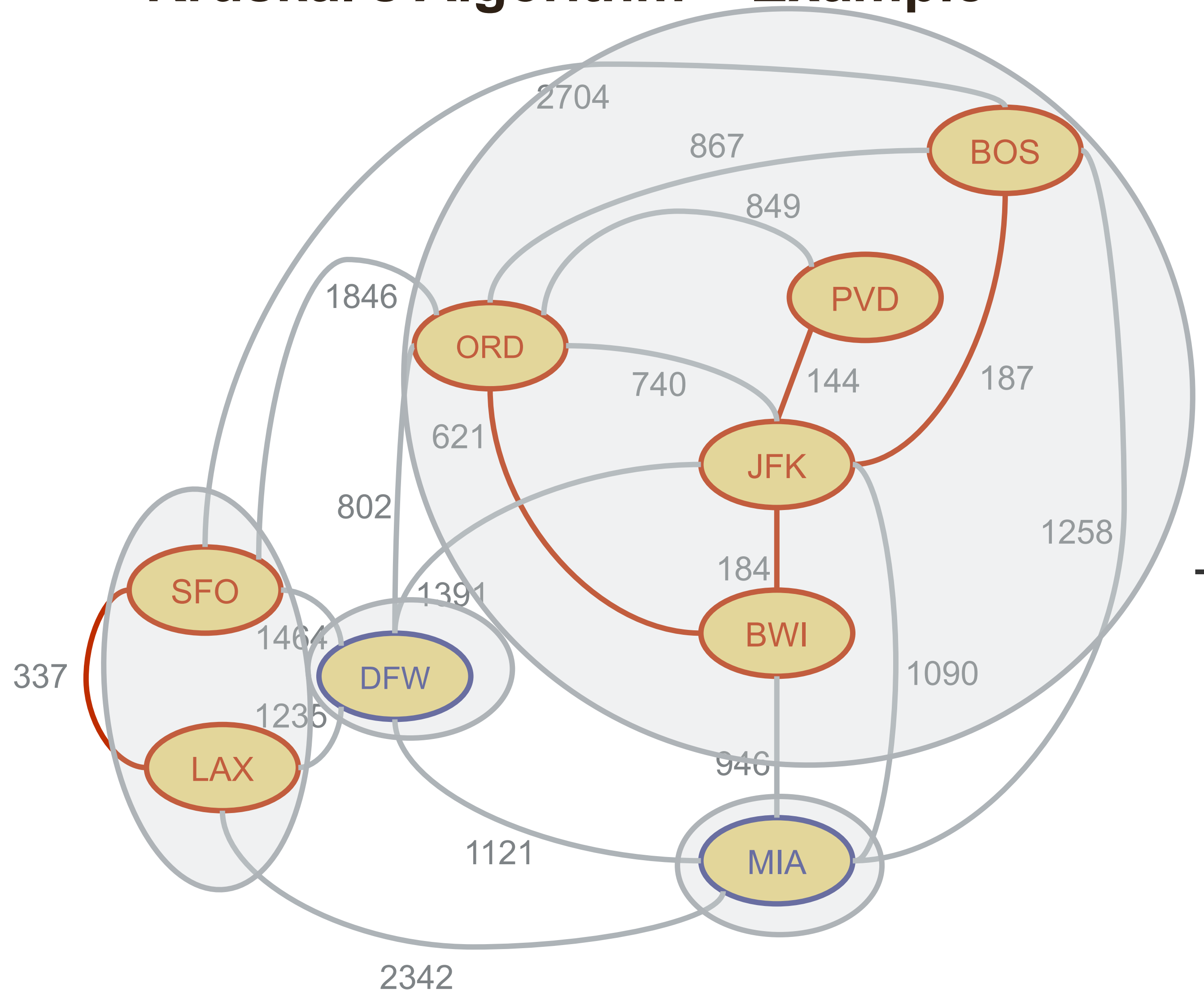
Kruskal's Algorithm – Example



	PQ	Tree
740	(ORD, JFK)	(JFK, PVD)
802	(DFW, ORD)	(BWI, JFK)
849	(ORD, PWD)	(JFK, BOS)
867	(ORD, BOS)	(SFO, LAX)
946	(MIA, BWI)	(BWI, ORD)
1090	(MIA, JFK)	
1121	(DFW, MIA)	
1235	(LAX, DFW)	
1258	(MIA, BOS)	
1391	(DFW, JFK)	
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (BWI, ORD), from PQ, add it to the tree, join clusters

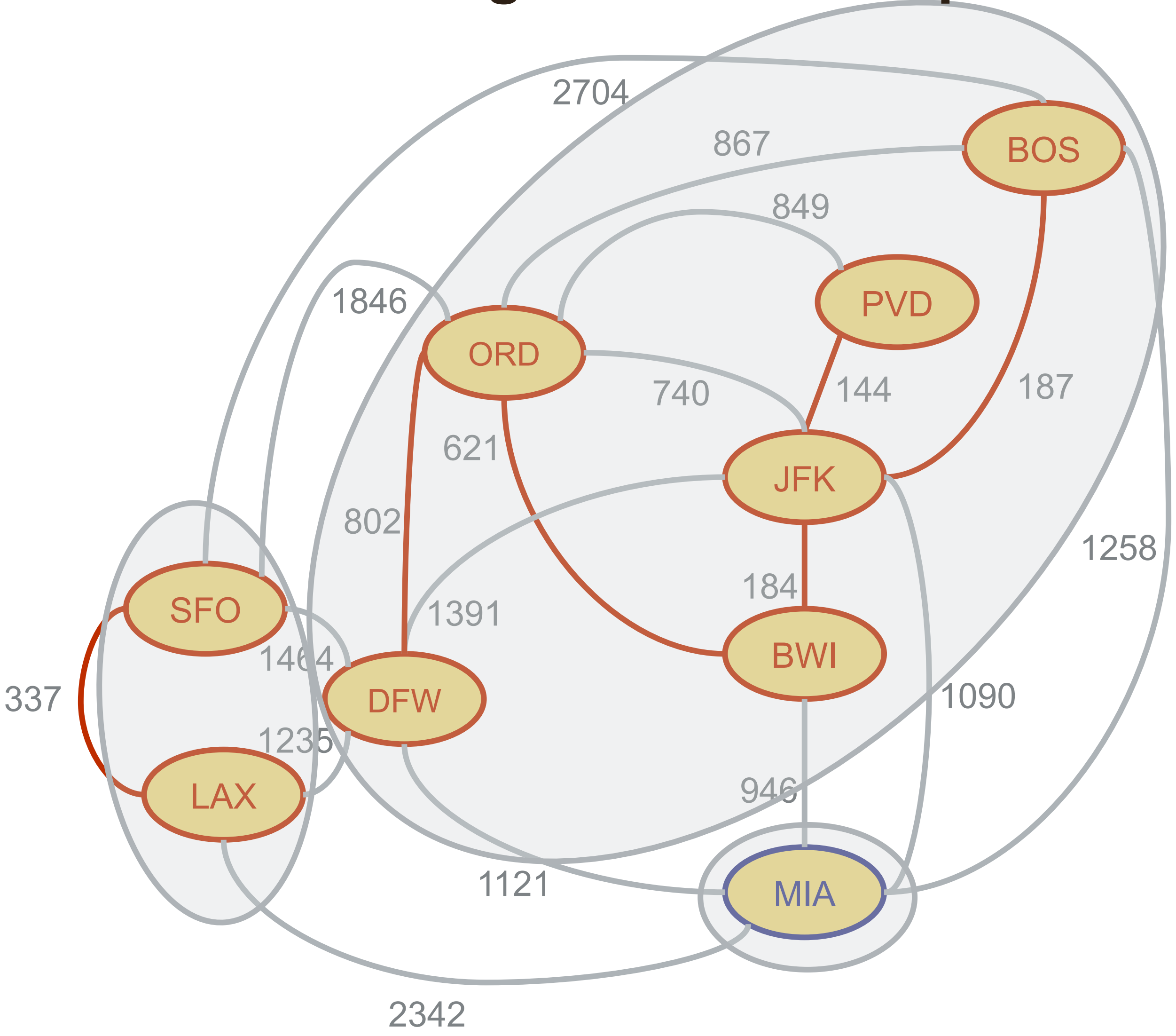
Kruskal's Algorithm – Example



	PQ	Tree
802	(DFW, ORD)	(JFK, PVD)
849	(ORD, PWD)	(BWI, JFK)
867	(ORD, BOS)	(JFK, BOS)
946	(MIA, BWI)	(SFO, LAX)
1090	(MIA, JFK)	(BWI, ORD)
1121	(DFW, MIA)	
1235	(LAX, DFW)	
1258	(MIA, BOS)	
1391	(DFW, JFK)	
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (ORD, JFK), from PQ
- Ignore it, both endpoints are in the same cluster

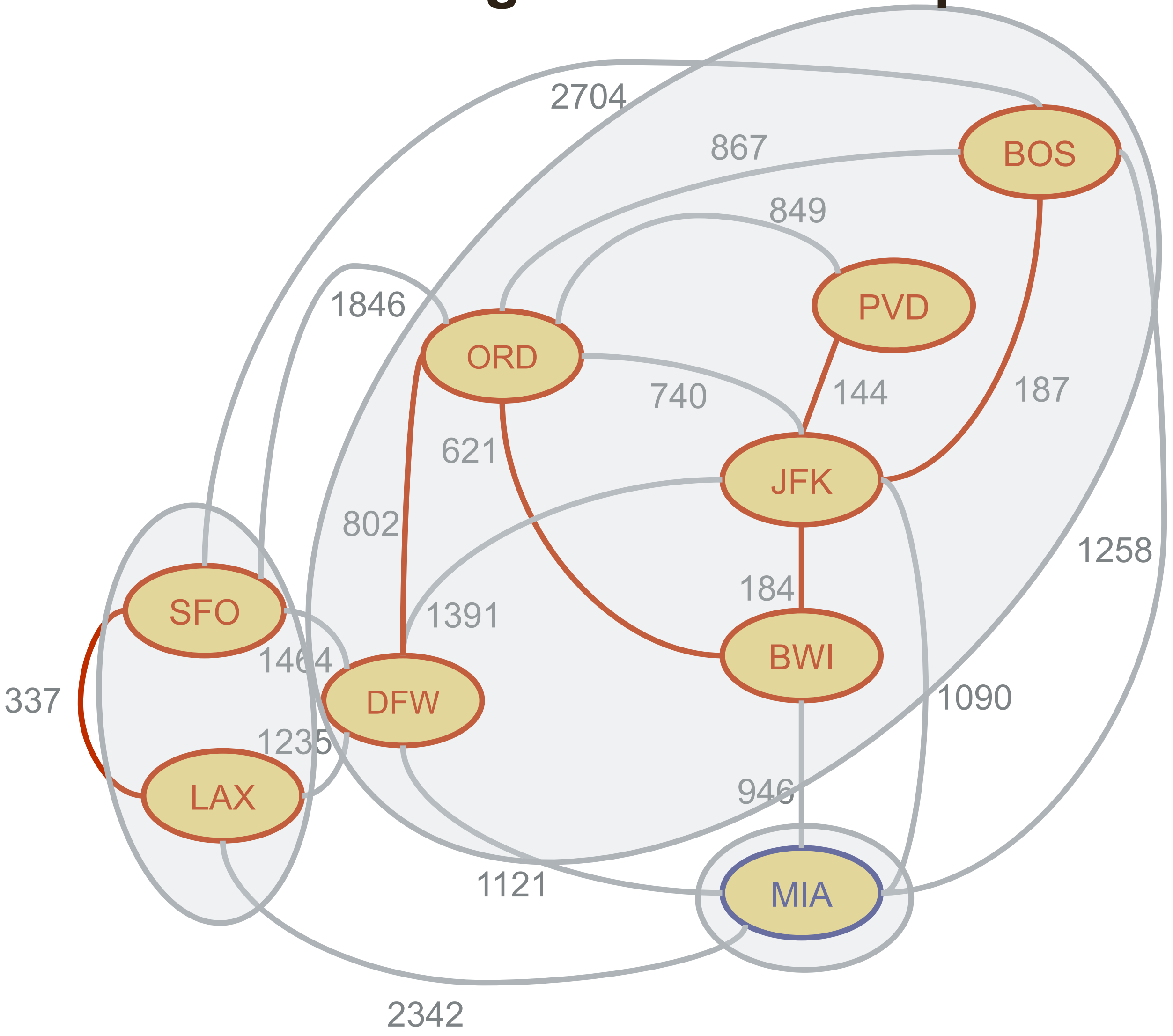
Kruskal's Algorithm – Example



PQ		Tree
849	(ORD, PWD)	(JFK, PVD)
867	(ORD, BOS)	(BWI, JFK)
946	(MIA, BWI)	(JFK, BOS)
1090	(MIA, JFK)	(SFO, LAX)
1121	(DFW, MIA)	(BWI, ORD)
1235	(LAX, DFW)	(DFW, ORD)
1258	(MIA, BOS)	
1391	(DFW, JFK)	
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (DFW, ORD), from PQ, add it to the tree, join clusters

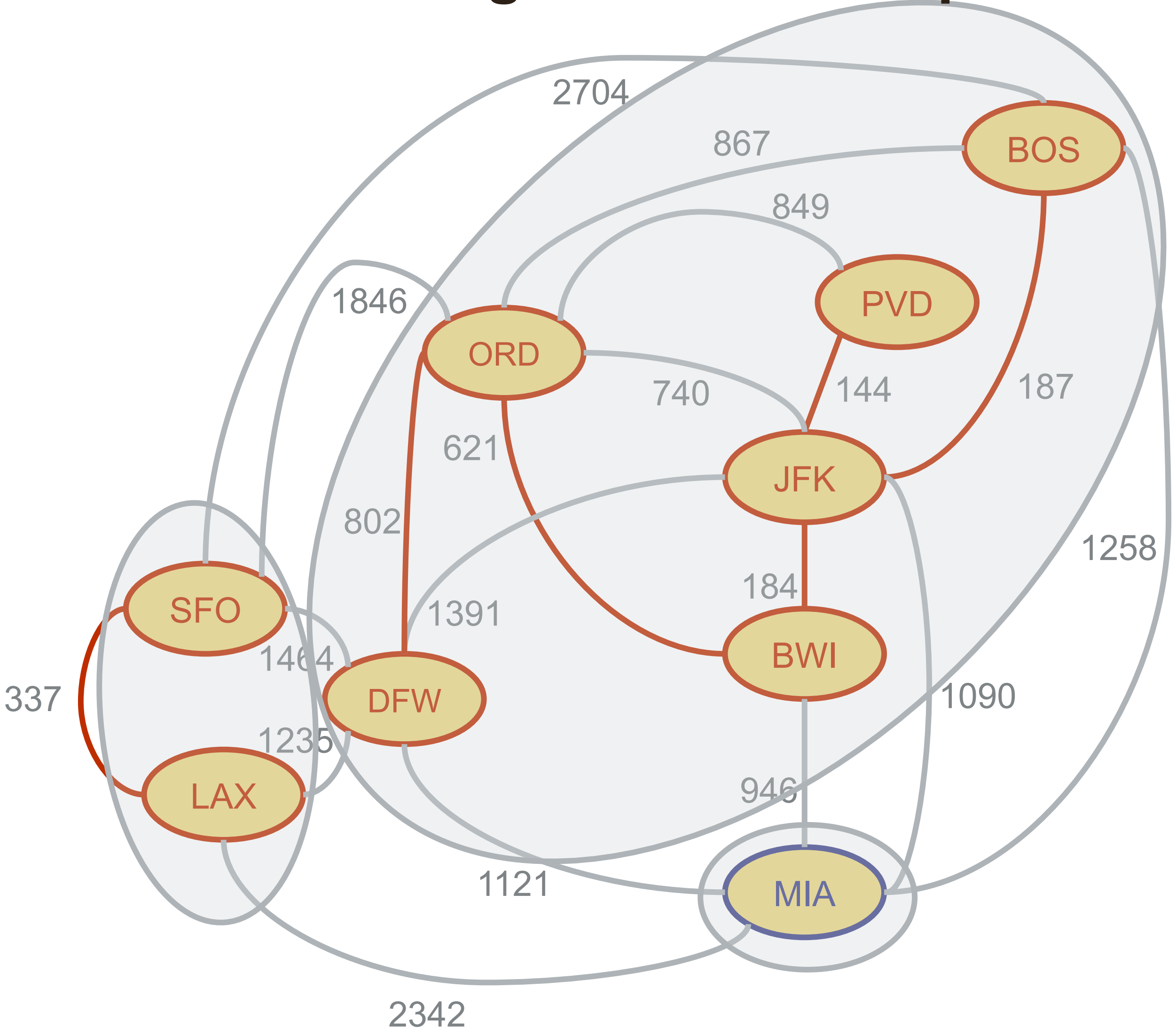
Kruskal's Algorithm – Example



	PQ	Tree
867	(ORD, BOS)	(JFK, PVD)
946	(MIA, BWI)	(BWI, JFK)
1090	(MIA, JFK)	(JFK, BOS)
1121	(DFW, MIA)	(SFO, LAX)
1235	(LAX, DFW)	(BWI, ORD)
1258	(MIA, BOS)	(DFW, ORD)
1391	(DFW, JFK)	
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (ORD, PVD), from PQ
- Ignore it, both endpoints are in the same cluster

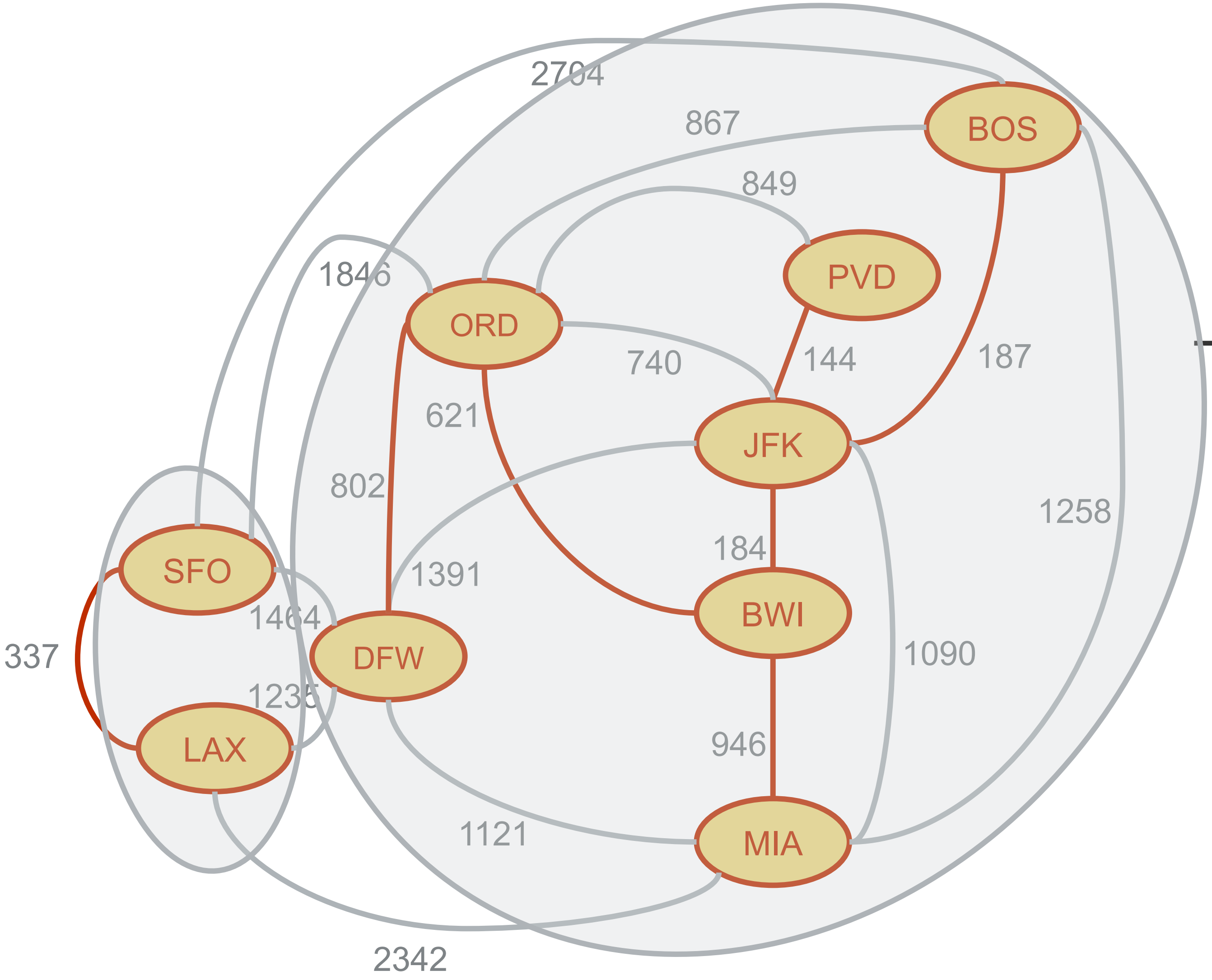
Kruskal's Algorithm – Example



	PQ	Tree
946	(MIA, BWI)	(JFK, PVD)
1090	(MIA, JFK)	(BWI, JFK)
1121	(DFW, MIA)	(JFK, BOS)
1235	(LAX, DFW)	(SFO, LAX)
1258	(MIA, BOS)	(BWI, ORD)
1391	(DFW, JFK)	(DFW, ORD)
1464	(SFO, DFW)	
1846	(SFO, ORD)	
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (ORD, BOS), from PQ
- Ignore it, both endpoints are in the same cluster

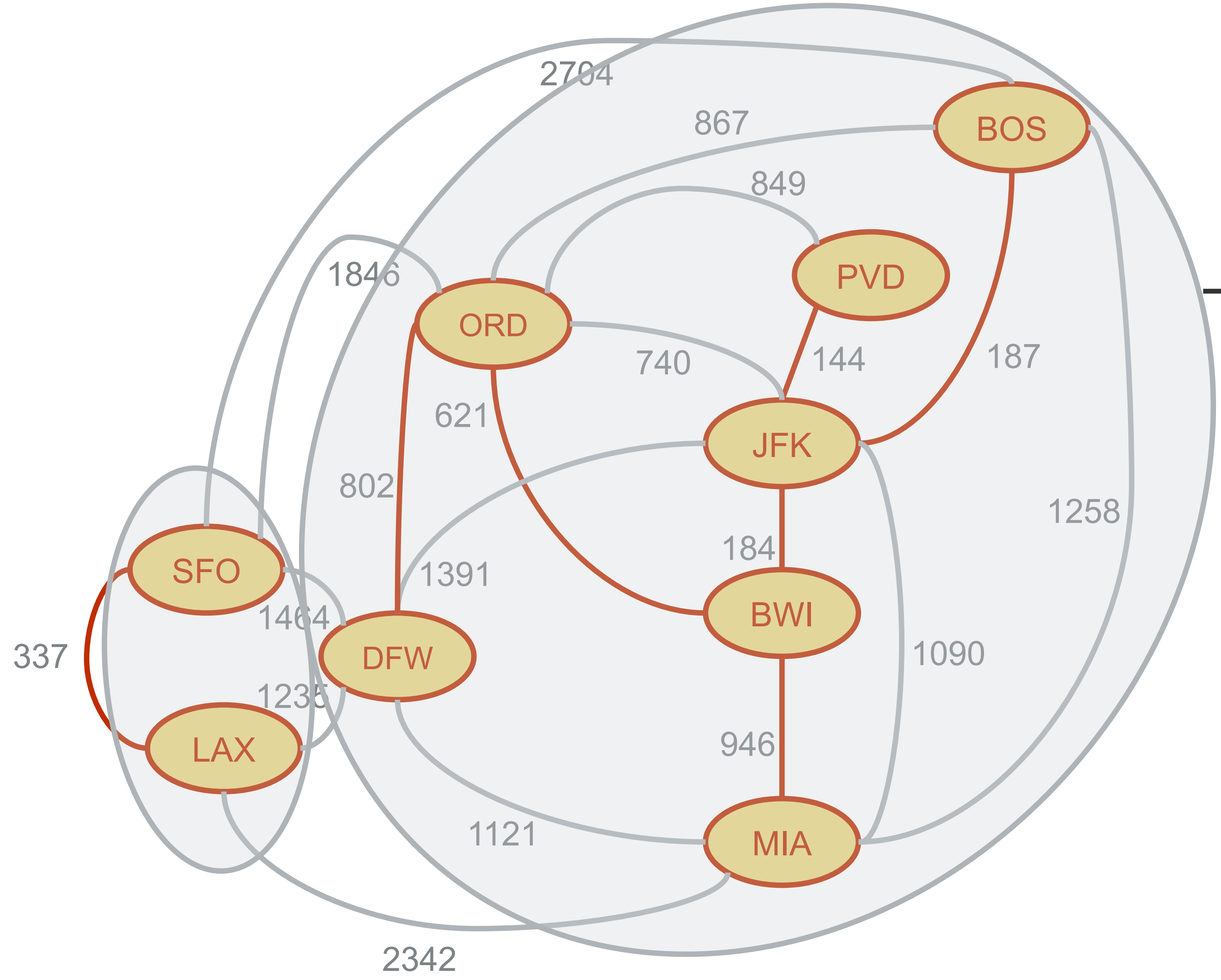
Kruskal's Algorithm – Example



	PQ	Tree
1090	(MIA, JFK)	(JFK, PVD)
1121	(DFW, MIA)	(BWI, JFK)
1235	(LAX, DFW)	(JFK, BOS)
1258	(MIA, BOS)	(SFO, LAX)
1391	(DFW, JFK)	(BWI, ORD)
1464	(SFO, DFW)	(DFW, ORD)
1846	(SFO, ORD)	(MIA, BWI)
2704	(SFO, BOS)	
2342	(LAX, MIA)	

- Remove the minimum weight edge, (MIA, BWI), from PQ, add it to the tree, join clusters

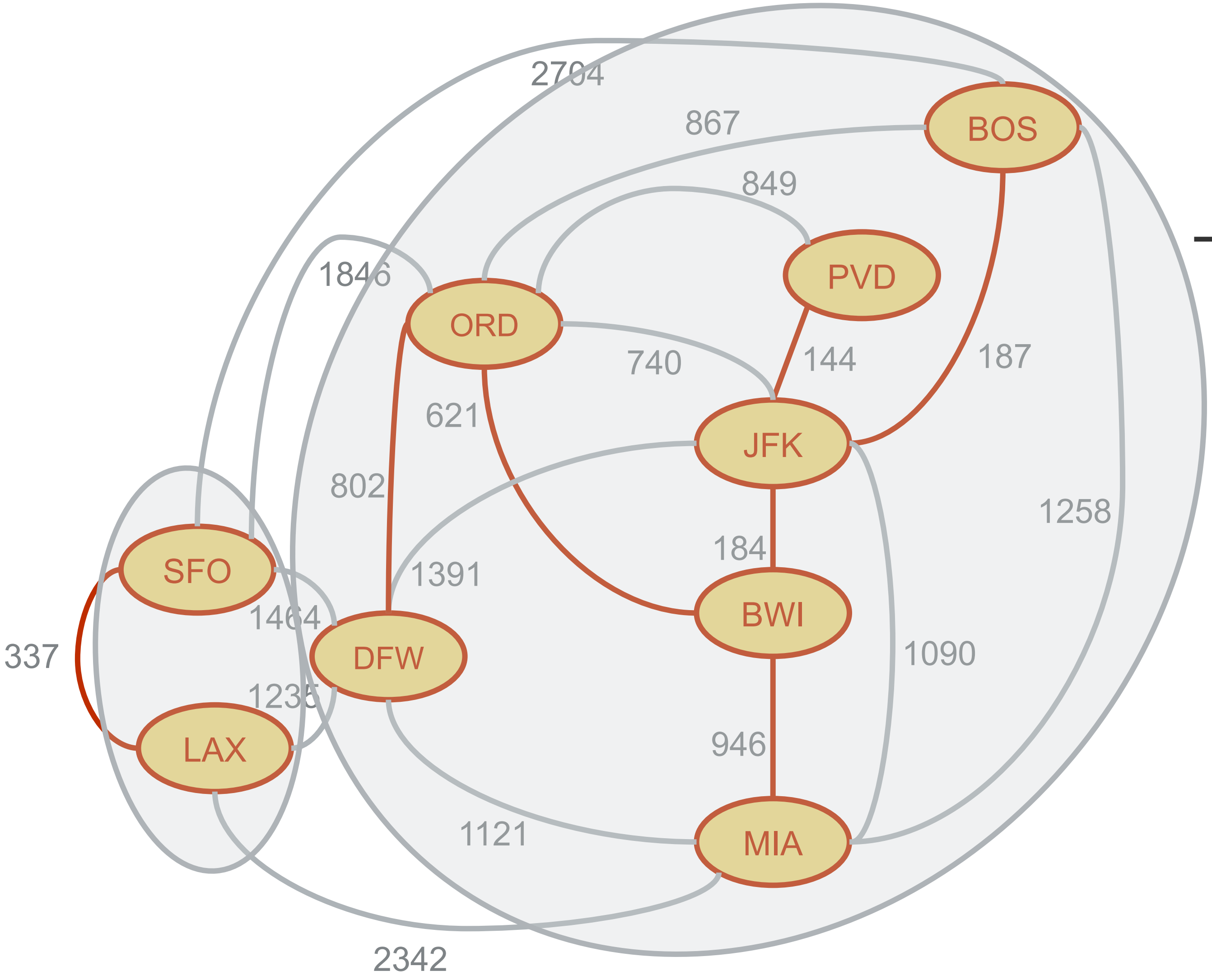
Kruskal's Algorithm – Example



	PQ	Tree
1121	(DFW, MIA)	(JFK, PVD)
1235	(LAX, DFW)	(BWI, JFK)
1258	(MIA, BOS)	(JFK, BOS)
1391	(DFW, JFK)	(SFO, LAX)
1464	(SFO, DFW)	(BWI, ORD)
1846	(SFO, ORD)	(DFW, ORD)
2704	(SFO, BOS)	(MIA, BWI)
2342	(LAX, MIA)	

- Remove the minimum weight edge, (MIA, JFK), from PQ
- Ignore it, both endpoints are in the same cluster

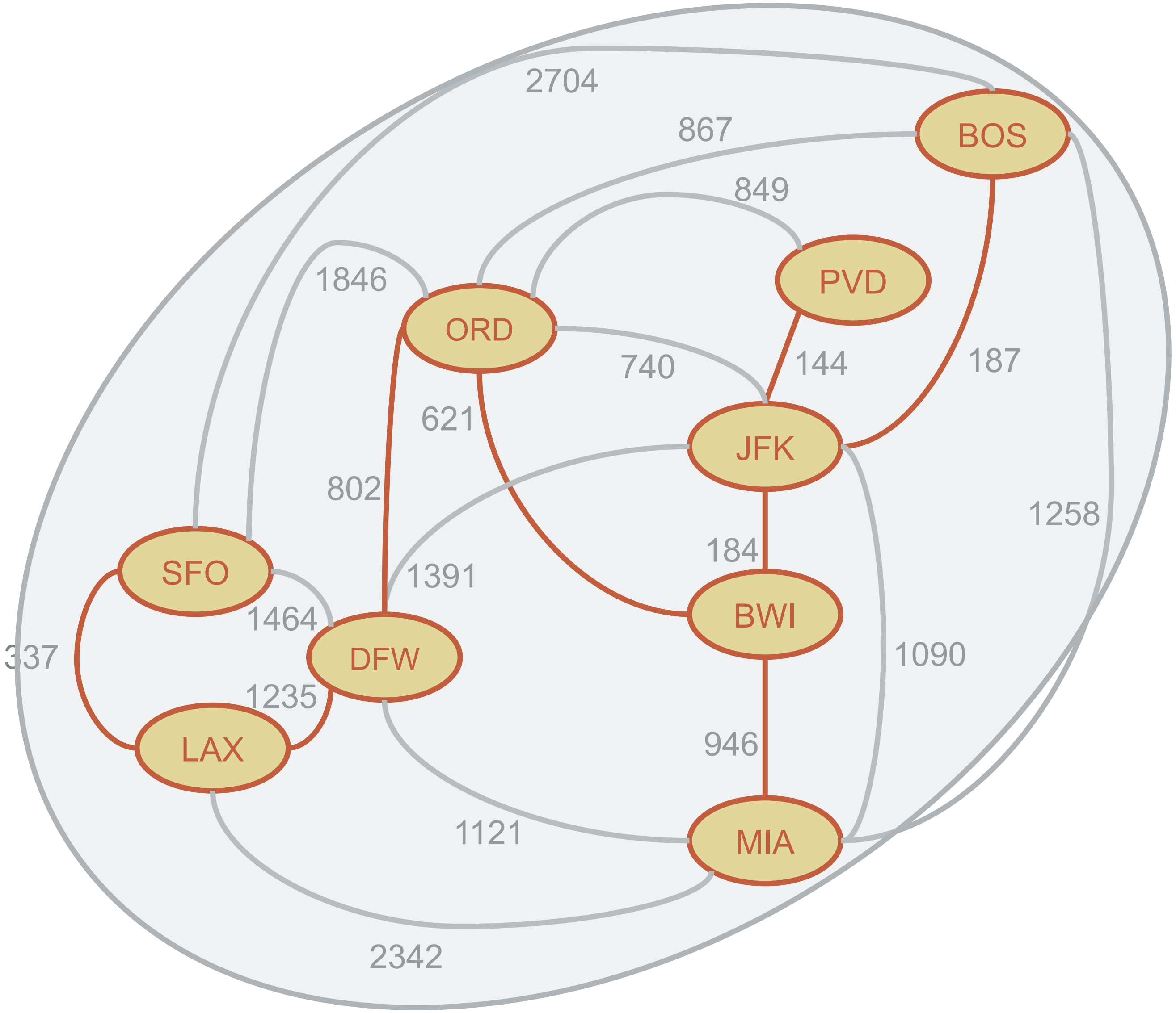
Kruskal's Algorithm – Example



	PQ	Tree
1235	(LAX, DFW)	(JFK, PVD)
1258	(MIA, BOS)	(BWI, JFK)
1391	(DFW, JFK)	(JFK, BOS)
1464	(SFO, DFW)	(SFO, LAX)
1846	(SFO, ORD)	(BWI, ORD)
2704	(SFO, BOS)	(DFW, ORD)
2342	(LAX, MIA)	(MIA, BWI)

- Remove the minimum weight edge, (DFW, MIA), from PQ
- Ignore it, both endpoints are in the same cluster

Kruskal's Algorithm – Example



	PQ	Tree
1258	(MIA, BOS)	(JFK, PVD)
1391	(DFW, JFK)	(BWI, JFK)
1464	(SFO, DFW)	(JFK, BOS)
1846	(SFO, ORD)	(SFO, LAX)
2704	(SFO, BOS)	(BWI, ORD)
2342	(LAX, MIA)	(DFW, ORD)
		(MIA, BWI)
		(LAX, DFW)

- Remove the minimum weight edge, (LAX, DFW), from PQ, add it to the tree, join clusters
- The graph contains 9 nodes
- The tree now contains 8 edges, so it is a MST – STOP.

Kruskal's Algorithm – Running Time Analysis

- If the graph has n vertices and m edges
- Part I: ordering the edges
  - Ordering the edges by weight takes $O(m \log m)$ time – either using a sorting algorithm directly, or a heap-based priority queue
  - If using a heap-based priority queue, initialization takes $O(m \log m)$ – repeated insertions or $O(m)$ – bottom-up heap construction
  - Each remove_min call takes $O(\log m)$ time
  - In a simple graph, m is $O(n^2)$– so $O(\log m)$ is the same as $O(\log n)$
  - So the time needed for ordering m edges is $O(m \log n)$

Kruskal's Algorithm – Running Time Analysis (cont'd)

- Part II: managing the clusters. To implement Kruskal's algorithm, we need to be able to:
  - Find the clusters for vertices u and v, the endpoints of edge e
  - Test whether the two clusters are distinct
  - Merge two clusters into one
  - We perform at most $2m$ find operations, and at most $n - 1$ union operations
- We need an efficient data structure for managing disjoint partitions – union-find
- Using the union-find structure the cluster operations in Kruskal's algorithm require $O(m + n \log n)$ time
- Thus the total running time of the algorithm is $O(m \log n)$

Kruskal's Algorithm – Python Implementation

```
def MST_Kruskal(g):
  """Compute a minimum spanning tree of a graph using Kruskal's algorithm.

  Return a list of edges that comprise the MST.

  The elements of the graph's edges are assumed to be weights.
  """
  tree = [ ]                    # list of edges in spanning tree
  pq = HeapPriorityQueue( )     # entries are edges in G, with weights as key
  forest = Partition( )         # keeps track of forest clusters
  position = { }                # map each node to its Partition entry

  for v in g.vertices( ):
    position[v] = forest.make_group(v)

  for e in g.edges( ):
    pq.add(e.element( ), e)     # edge's element is assumed to be its weight

  size = g.vertex_count( )
  while len(tree) != size − 1 and not pq.is_empty( ):
    # tree not spanning and unprocessed edges remain
    weight,edge = pq.remove_min( )
    u,v = edge.endpoints( )
    a = forest.find(position[u])
    b = forest.find(position[v])
    if a != b:
      tree.append(edge)
      forest.union(a,b)

  return tree
```

Disjoint Partitions and Union-Find Structures

The Partition Data Structure

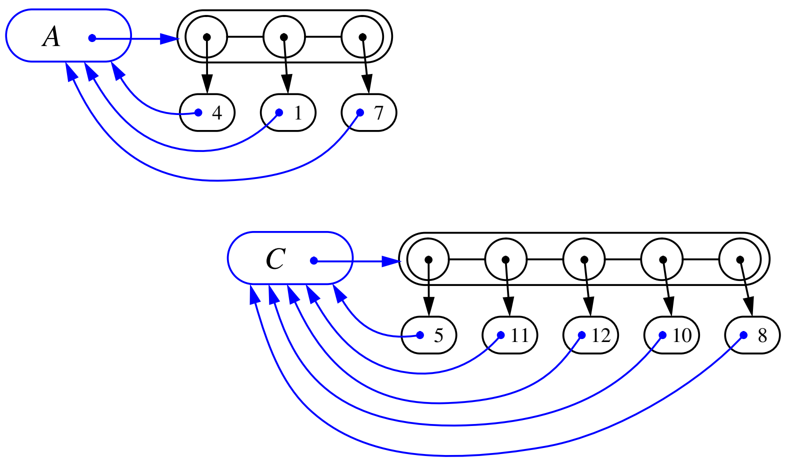
- A Partition data structure manages a collection of elements organized into disjoint sets
- Each element can belong to one and only one of the sets in the partition
- We don't want to iterate through the elements of a partition, or be able to test if a given set includes a given element
- Rather, we want to be able to create sets containing certain elements, be able to merge them and also be able to find the group containing a particular element
- To avoid confusion, refer to the clusters of the partitions as groups
- The groups don't need an explicit internal structure
- To differentiate between groups, assume that each group has a designated entry called the leader of the group

Partition ADT

- We define the following methods for the Partition ADT:
  - make_group(x): Create a singleton group containing a new element x and return the position storing x
  - union(p,q): Merge the groups containing positions p and q
  - find(p): Return the position of the leader of the group containing position p

Partition ADT – Sequence Implementation

- Implement a partition with a total of n elements using a collection of sequences, one for each group
- The sequence for group A stores element positions
- Each Position object stores:
  - A variable element which references its associated element x and allows the execution of an element() method in $O(1)$ time
  - A variable group which that references the sequence storing p



sequence-based implementation of a partition consisting of two groups:
A={1,4,7} and
C={5,8,10,11,12}

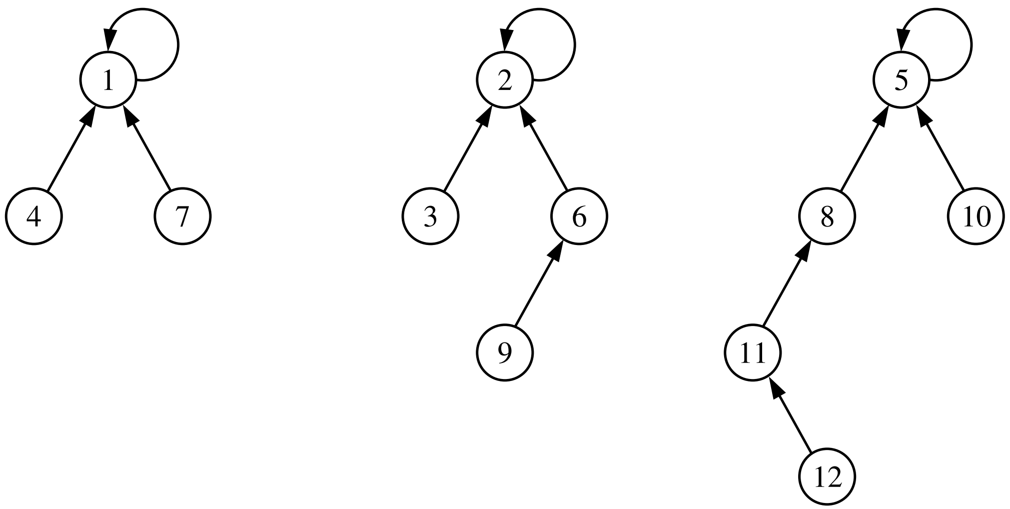
Partition ADT – Running Time for Sequence Implementation

operation	running time
make_group(x)	$O(1)$
find(p)	$O(1)$
union(p,q)	$O(n)$

- The make_group(x) and find(p) operations can be implemented in constant time, if the first position of a sequence is used as the leader
- The union(p,q) operation requires two sequences to be joined into one; plus, the group references in one of the sequences have to be updated
- The time for the union(p,q) operation is $\min(|A|, |B|)$ where A and B are the groups containing positions p and q - $O(n)$ running time if there are n elements in the whole partition

Partition ADT – Tree-Based Implementation

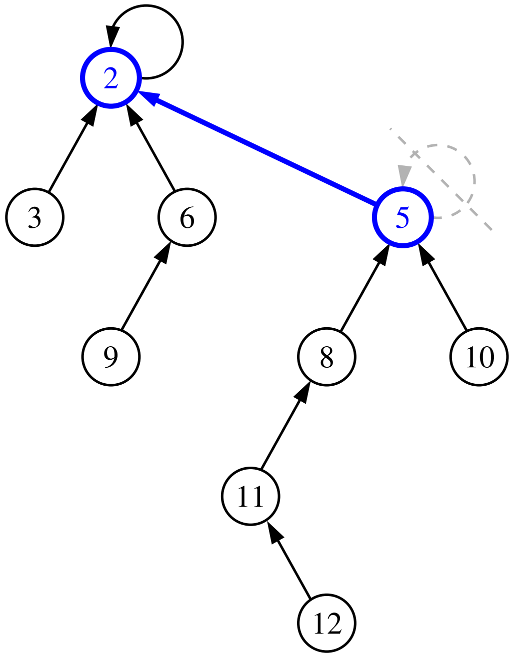
- Use a collection of trees to store the n elements of a partition, where each tree is associated with a different group
- Each position p is a node having
  - An instance variable element referring to its element x
  - An instance variable parent referring to its parent node
- By convention, if p is the root of its tree, then its parent reference is set to itself

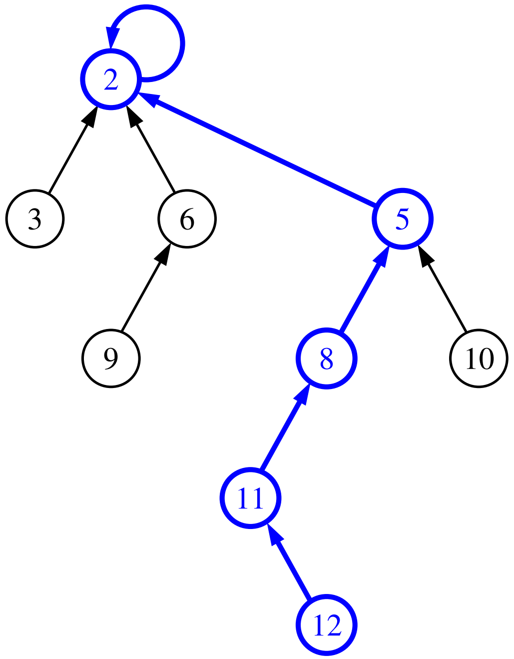


Partition ADT – Tree-Based Implementation (cont'd)

- Using the tree-based implementation the find(p) operation is performed by walking up from position p to the root of its tree - $O(n)$ worst case time
- The union(p,q) operation is implemented by making one of the trees a subtree of the other: first locate the two roots, then set the parent reference of one root to point to the other

union(2,5) operation



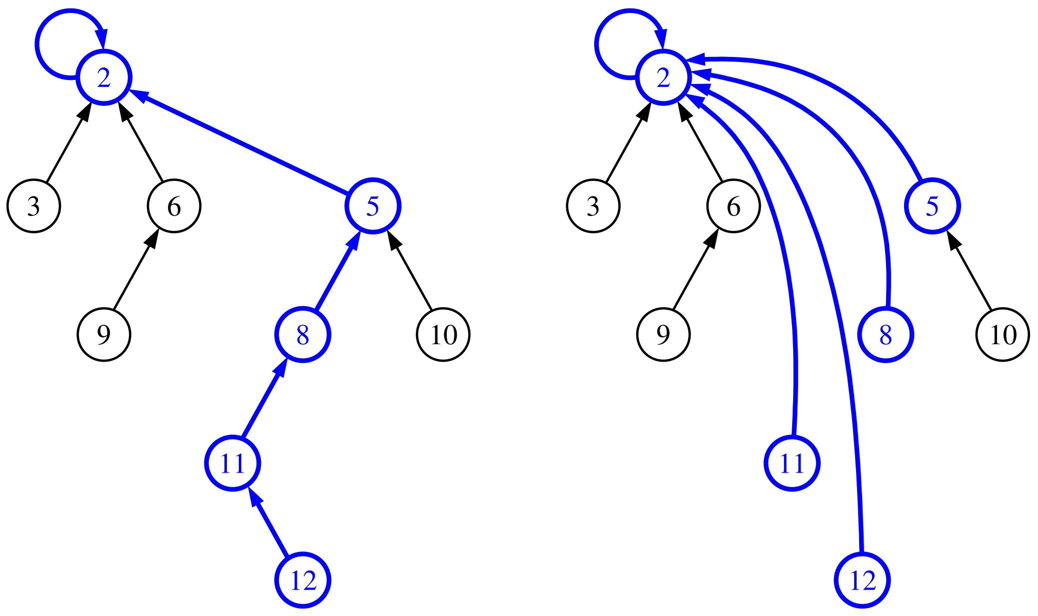


find(12) operation

Partition ADT – Tree-Based Implementation (cont'd)

- Problem: finding the root might still take $O(n)$ time if the tree is made of a long chain of nodes
- Solution 1: union-by-size
  - with each position p, also store the number of elements in the subtree rooted at p
  - In a union operation, make the root of the smaller group become a child of the other root, and update the size field of the larger root
- Solution 2: path compression
  - In a find operation, for each position q that find visits, reset the parent of q to the root

Path compression



Partition – Python Implementation

```
class Partition:
  """Union-find structure for maintaining disjoint sets."""

  #------------------------ nested Position class ------------------------
  class Position:
    __slots__ = '_container', '_element', '_size', '_parent'

    def __init__(self, container, e):
      """Create a new position that is the leader of its own group."""
      self._container = container          # reference to Partition instance
      self._element = e
      self._size = 1
      self._parent = self                  # convention for a group leader

    def element(self):
      """Return element stored at this position."""
      return self._element

```

Partition – Python Implementation (cont'd)

```
#------------------------- public Partition methods -------------------------
def make_group(self, e):
  """Makes a new group containing element e, and returns its Position."""
  return self.Position(self, e)

def find(self, p):
  """Finds the group containging p and return the position of its leader."""
  if p._parent != p:
    p._parent = self.find(p._parent)      # overwrite p._parent after recursion
  return p._parent

def union(self, p, q):
  """Merges the groups containg elements p and q (if distinct)."""
  a = self.find(p)
  b = self.find(q)
  if a is not b:                          # only merge if different groups
    if a._size > b._size:
      b._parent = a
      a._size += b._size
    else:
      a._parent = b
      b._size += a._size
```

Partition ADT – Running Time for Tree-Based Implementation

- Proposition. When using a tree-based partition representation with both union-by-size and path compression, performing a series of k make_group, union and find operations on an initially empty partition involving at most n elements takes $O(k \log^* n)$ time.

- $\log^* n$ - log star function

minimum n	2	$2^2 = 4$	$2^{2^2} = 16$	$2^{2^{2^2}} = 65,536$	$2^{2^{2^{2^2}}} = 2^{65,536}$
$\log^* n$	1	2	3	4	5

- A linear running time in practice, although it is theoretically not linear

Thank you.