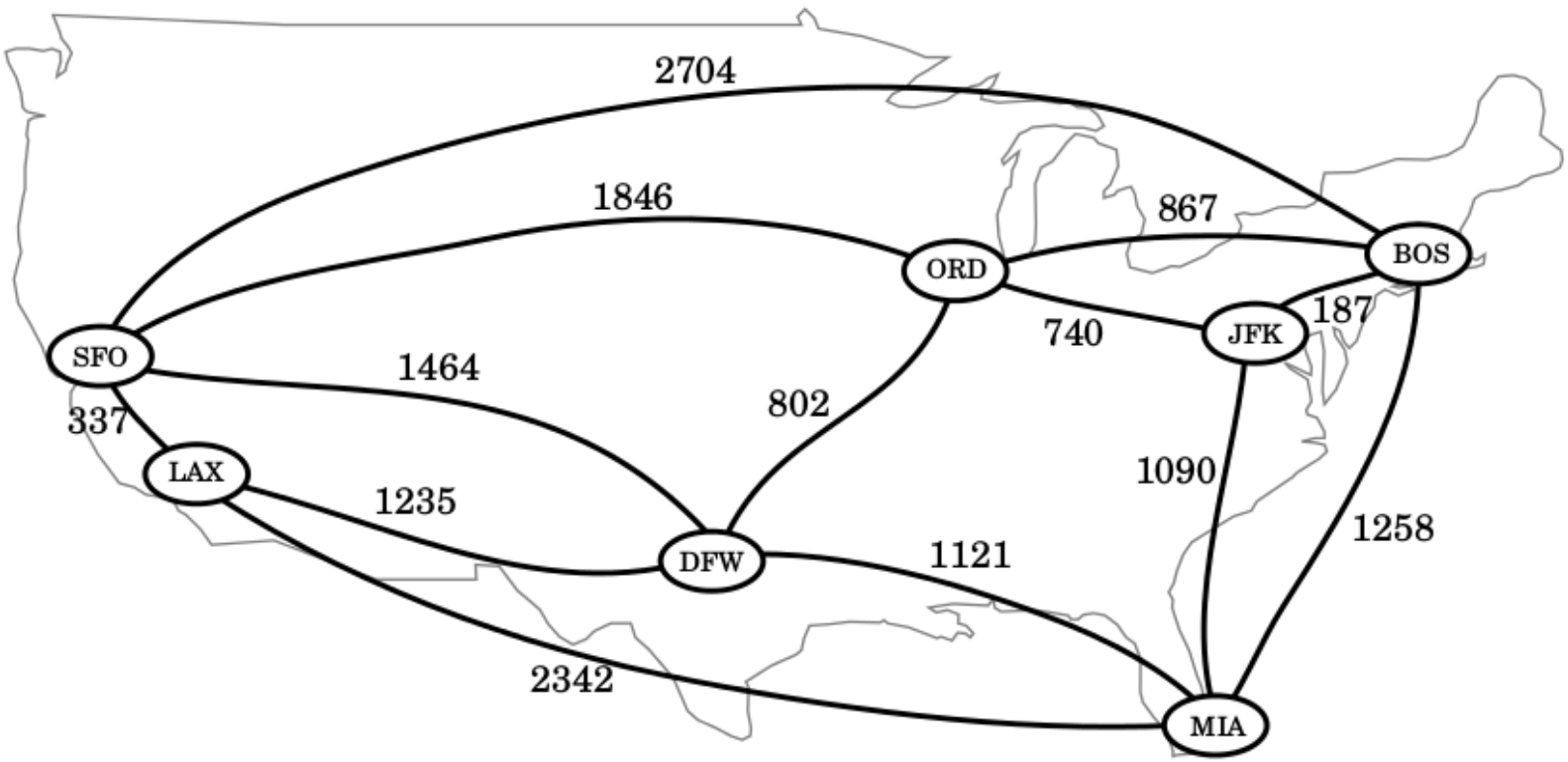
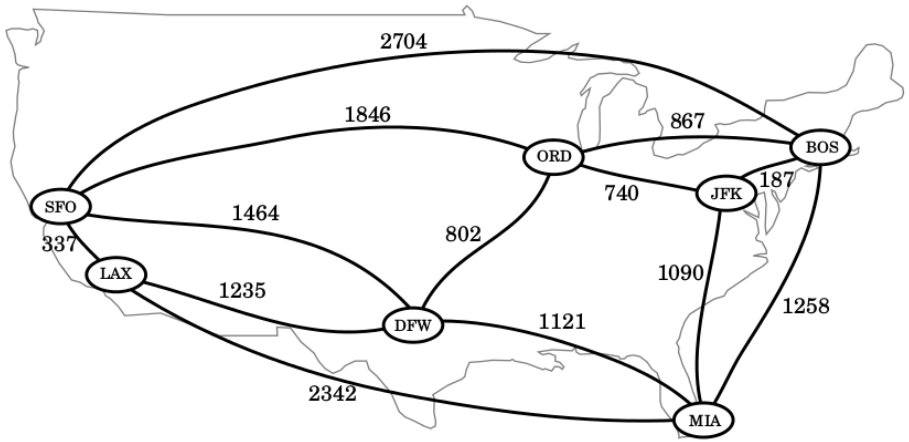
Shortest Paths

- The BFS search strategy can be used to find a shortest path from some starting vertex v to every other vertex in a connected graph
- This approach is useful when each edge is as good as any other, in particular it fails when a **weighted graph** has to be taken into account
- A **weighted graph** is a graph that has a numeric label $w(e$ associated with each edge e (the weight of that edge)
- For $e = (u, v)$ we have $w(u, v) = w(e)$

Shortest Paths



Shortest Paths



Possible paths from JFK to LAX:

Path	**Weights**	**Total**
JFK - ORD - SFO - LAX	$740 + 1846 + 337$	2923
JFK - ORD - DFW - LAX	$740 + 802 + 1235$	2777
JFK - MIA - LAX	$1090 + 2342$	3432
JFK - MIA - DFW - LAX	$1090 + 1121 + 1235$	3446
JFK - BOS - ORD - SFO - LAX	$187 + 867 + 1846 + 337$	3237
JFK - BOS - ORD - DFW - LAX	$187 + 867 + 802 + 1235$	3091

Shortest Paths

The weight (or length) of a path P in a weighted graph G is the sum of the weights of the edges of P.

If $P = ((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$ then

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

The **distance** from a vertex u to a vertex v in G, denoted $d(u, v)$ is the length of a minimum-length path (**shortest path**) from u to v (if such a path exists)

- A class of algorithms solve the problem of finding a shortest path from some vertex s to each other vertex in a weighted graph G
- One of these, **Dijkstra's algorithm**, applies the **greedy method**: a given problem is solved by repeatedly selecting the best choice from among those available at each iteration
- The main idea is to perform a weighted BFS search starting from the source vertex s and creating a "cloud" of vertices, each entering the cloud in order of their distance from s
- Then, in each iteration, the next vertex chosen is the vertex outside the cloud closest to s
- The algorithm terminates when no more vertices are outside the cloud (or when those outside the cloud are not connected to those within the cloud)

- Let us define a label $D[v]$ for each vertex v in V
- These labels will always store the length of the best path obtained **so far** from s to v
- Initially, $D[s] = 0$ and $D[v] = \infty$ for each $v \neq s$
- We define the set C (the "cloud" of vertices) as the empty set
- At each iteration, a vertex u is selected not in C with smallest $D[u]$ label, and u is put in C

- Once a new vertex u is pulled into C, the label $D[v]$ of each vertex adjacent to u and outside of C is updated, to reflect the fact that may be a new and better way to reach v via u
- This update is known as **relaxation**, because it takes an old estimate and checks if it can be improved to get closer to its value

More precisely:

```
if D[u] + w(u,v) < D[v] then
   D[v] = D[u] + w(u,v)
```

```
Algorithm ShortestPath(G,s):
  D[s] = 0
  D[v] = infinite for each vertex v != s

  Let a priority queue Q contain all the vertices
  of G using the D labels as keys

  while Q is not empty do
    u = value returned by Q.remove_min()
    for each vertex v adjacent to u such that v is in Q
        do
      if D[u] + w(u,v) < D[v] then
        D[v] = D[u] + w(u,v)
        Change to D[v] the key of vertex v in Q
  return the label D[v] of each vertex v
```

```
. . .
  Let a priority queue Q contain all the vertices
  of G using the D labels as keys

  while Q is not empty do
    u = value returned by Q.remove_min()
    for each vertex v adjacent to u such that v is in Q
        do
      if D[u] + w(u,v) < D[v] then
        D[v] = D[u] + w(u,v)
        Change to D[v] the key of vertex v in Q
  return the label D[v] of each vertex v
```

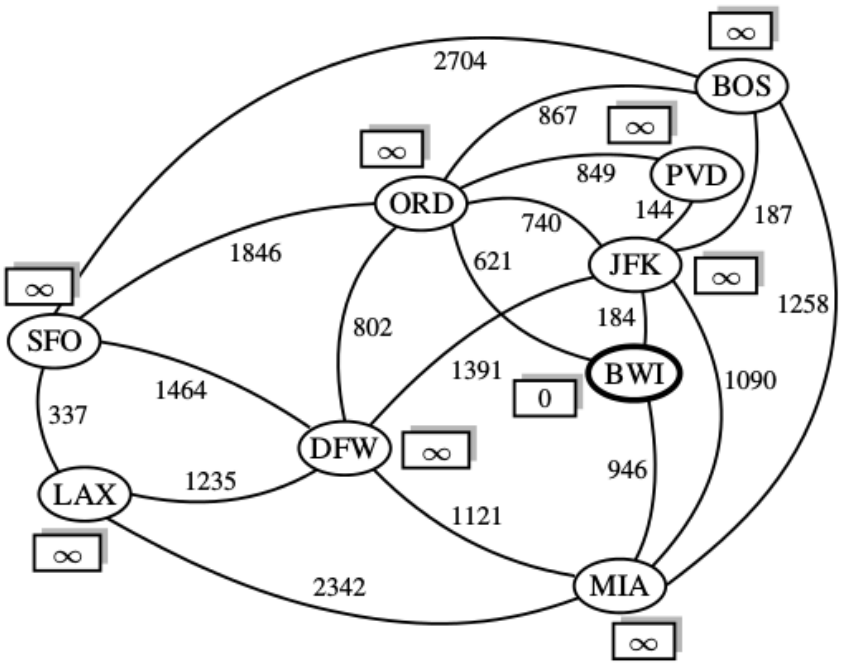
Shortest Paths

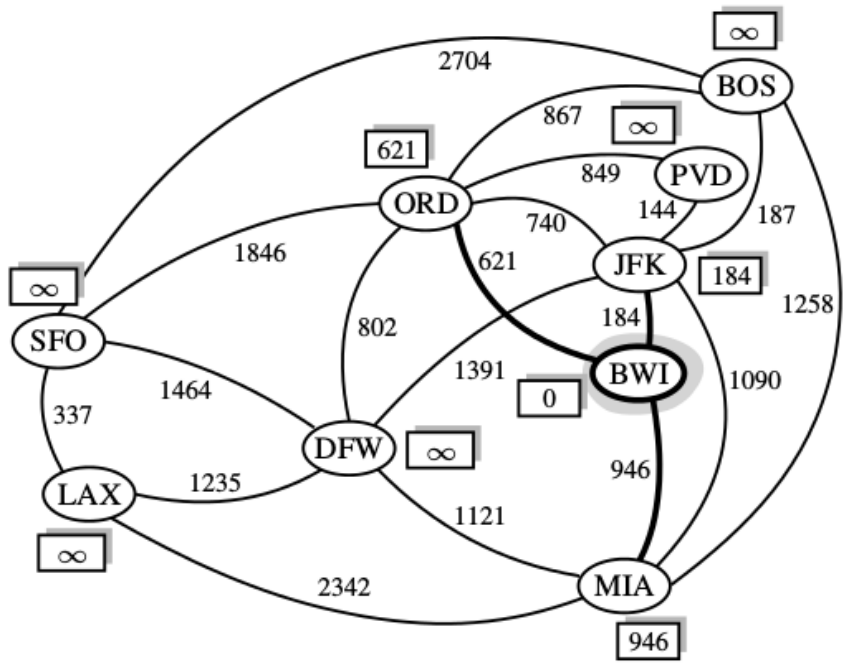
Computational Cost

- The nested **for** loop runs in $O(m)$ time
- The outer **while** loop executes in $O(n)$ time
- The update of Q requires $O(m)$ time
- If Q is a priority queue, each of the above operations run in $O(logn)$, therefore the overall running time is $O((n+m)logn)$, that is (approx) $O(n^2 logn)$

Shortest Paths

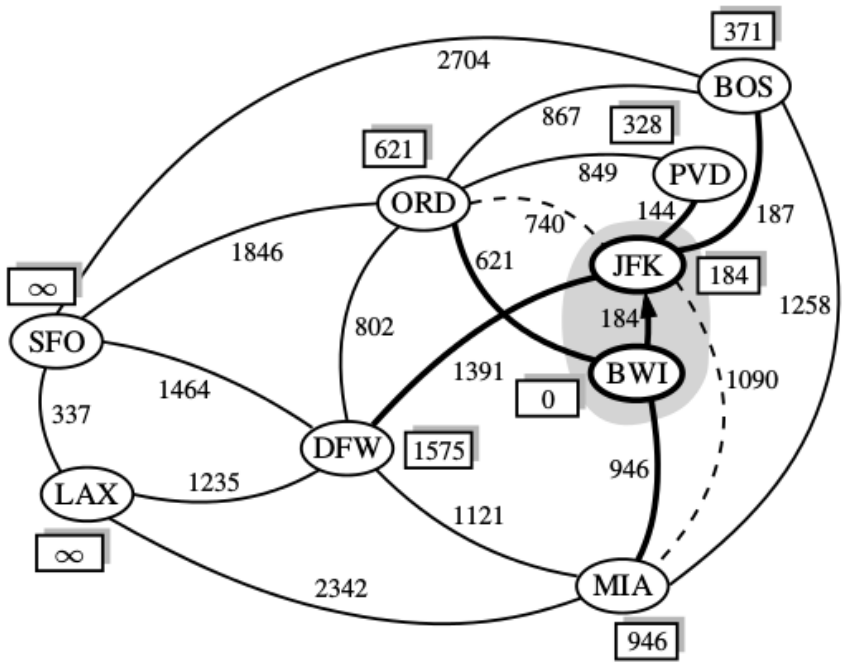
Dijkstra's Algorithm

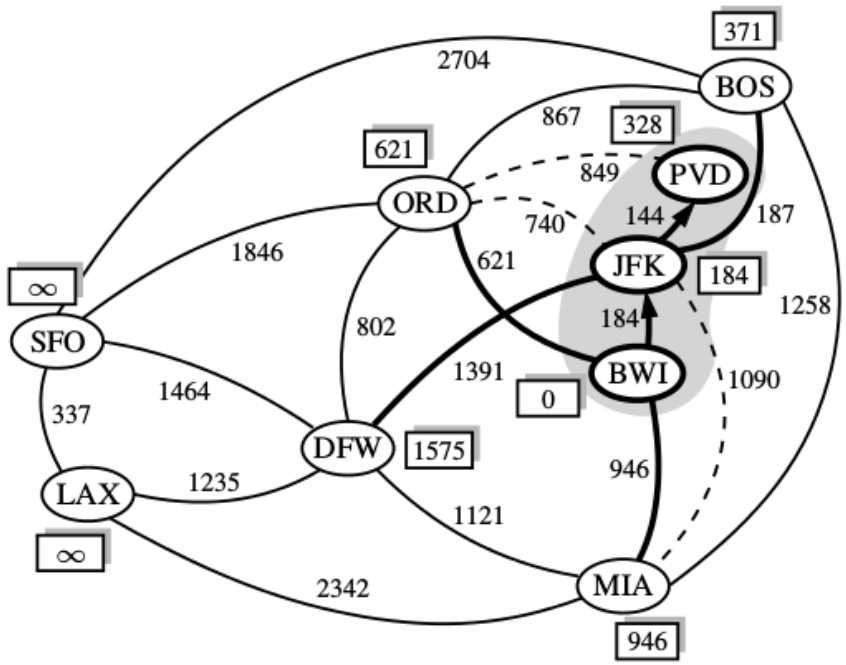




Shortest Paths

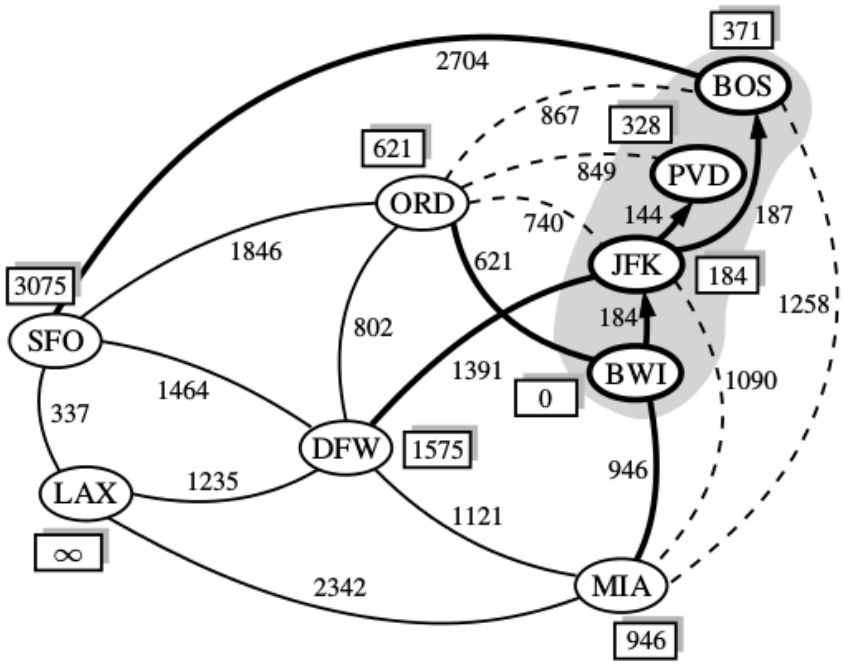
Dijkstra's Algorithm

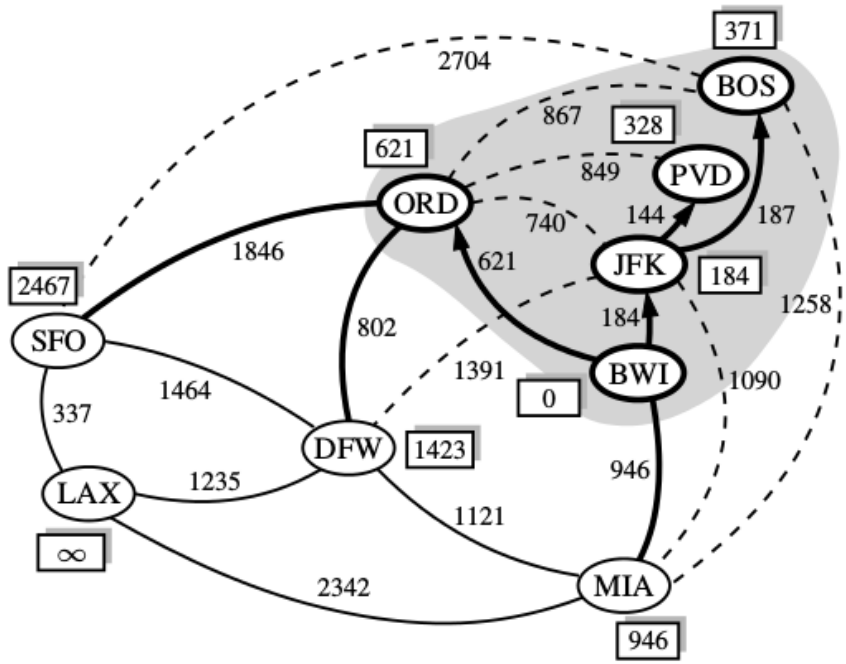




Shortest Paths

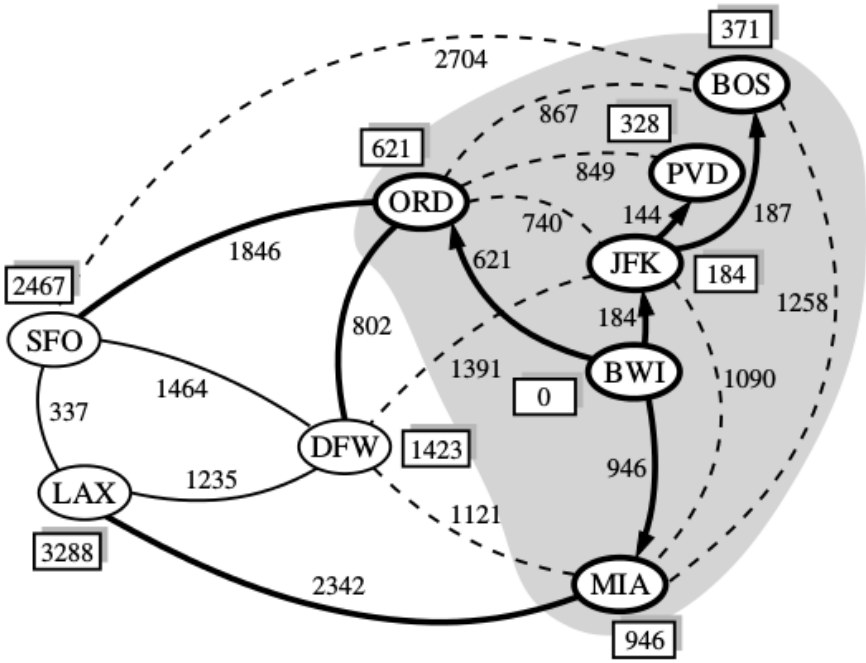
Dijkstra's Algorithm

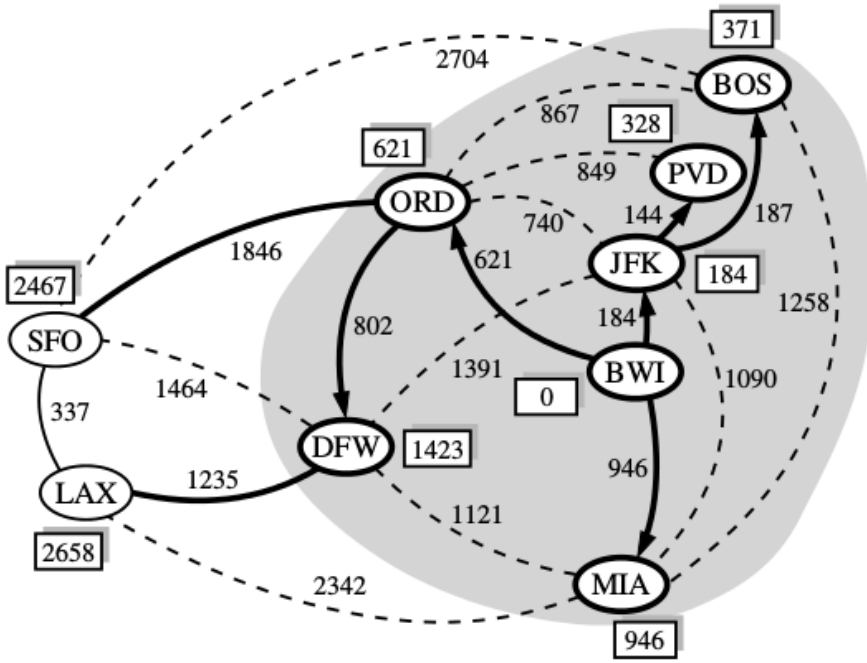




Shortest Paths

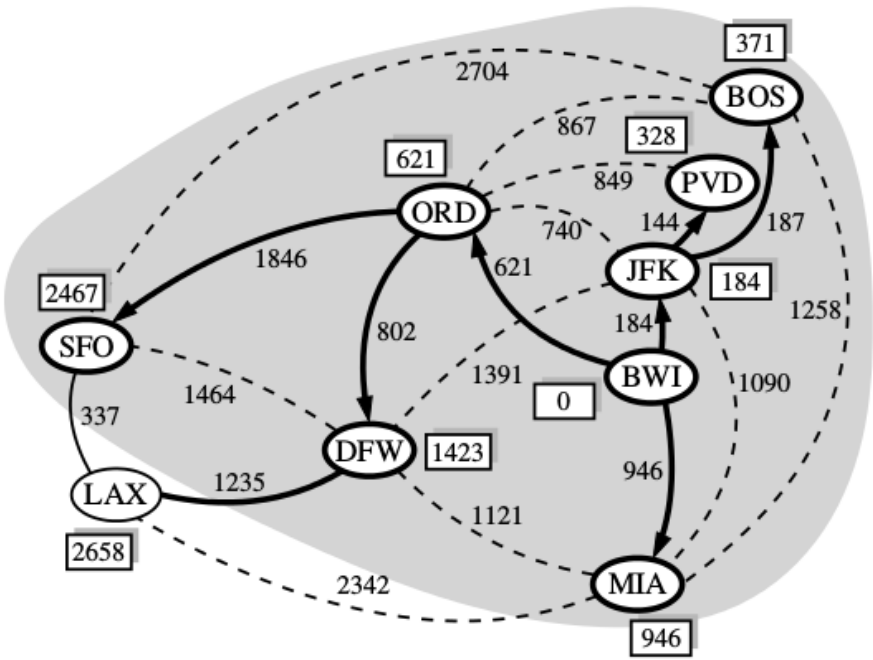
Dijkstra's Algorithm

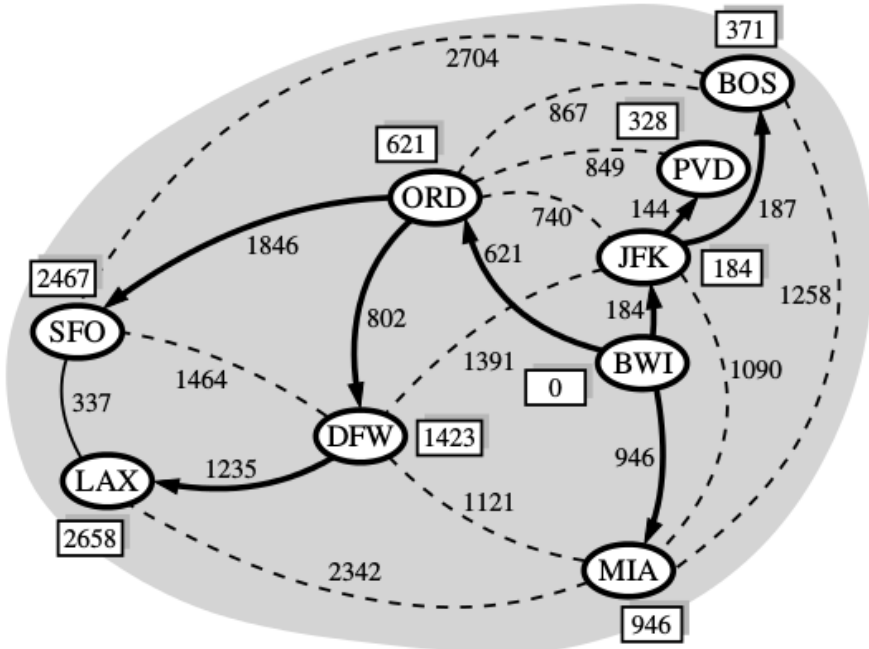




Shortest Paths

Dijkstra's Algorithm





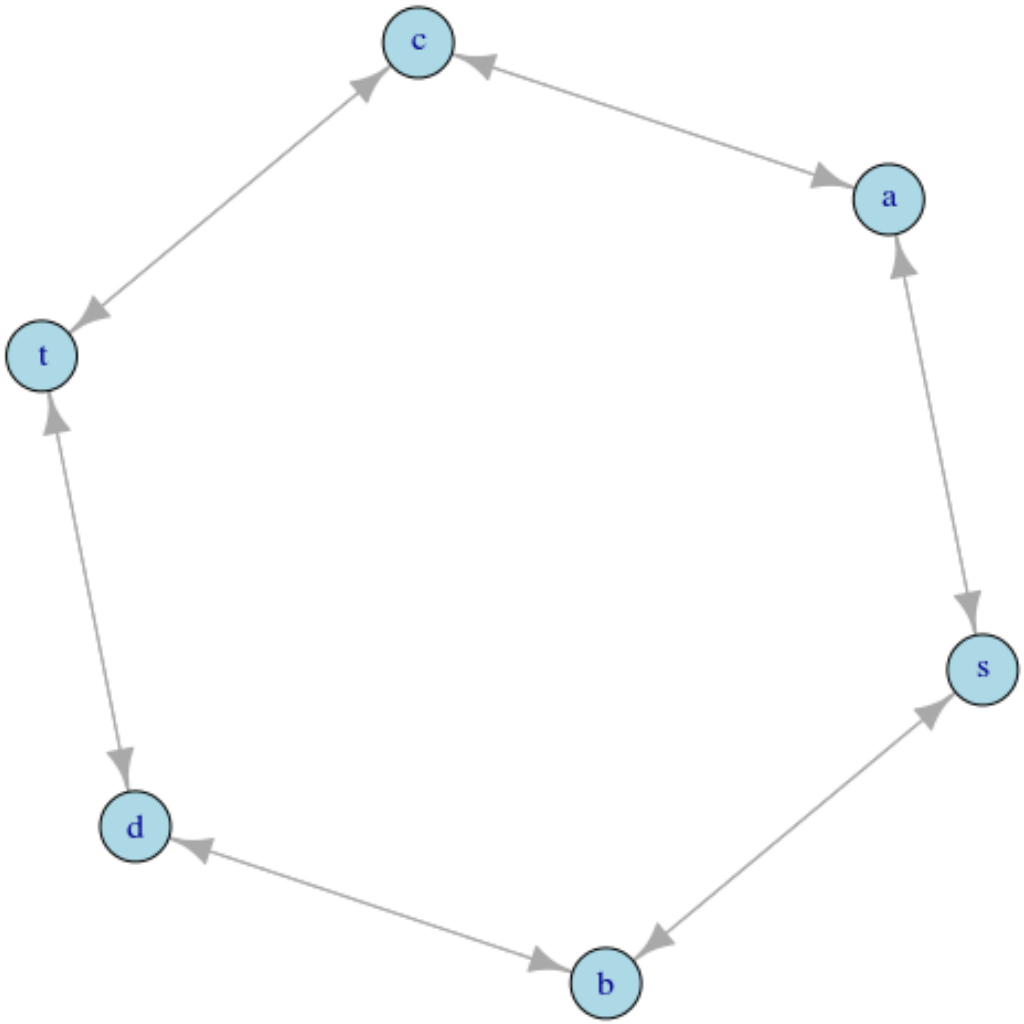
Dijkstra's Algorithm

Shortest Path Tree

```
def shortest_path_tree(g, s, d):

  tree = {}
  for v in d:
    if v is not s:
      for e in g.incident_edges(v, False):
        u = e.opposite(v)
        wgt = e.element()
        if d[v] == d[u] + wgt:
          tree[v] = e
  return tree
```

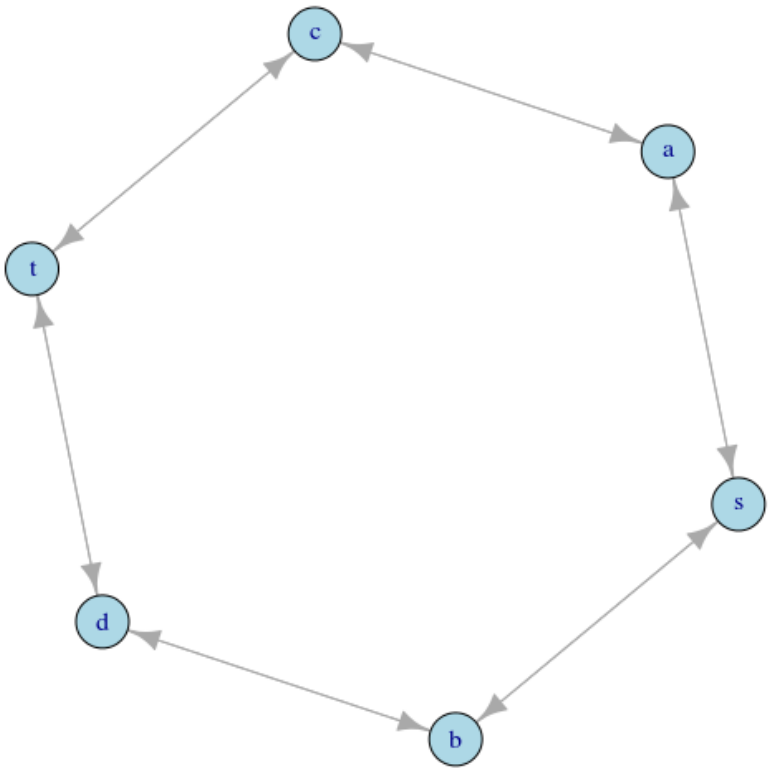
Shortest Paths

Dijkstra's Algorithm



Shortest Paths

Dijkstra's Algorithm




```
graph = {'s': {'a': 2, 'b': 1},
         'a': {'s': 3, 'b': 4, 'c'
             :8},
         'b': {'s': 4, 'a': 2, 'd':
             2},
         'c': {'a': 2, 'd': 7, 't':
             4},
         'd': {'b': 1, 'c': 11, 't':
              5},
         't': {'c': 3, 'd': 5}}
```

Shortest Paths

Dijkstra's Algorithm

```
def dijkstra(graph,src,dest,visited=[],distances={},
    predecessors={}):
    if src not in graph:
        raise TypeError('the root of the shortest path
            tree cannot be found in the graph')
    if dest not in graph:
        raise TypeError('the target of the shortest path
             cannot be found in the graph')
    if src == dest:
        path=[]
        pred=dest
        while pred != None:
            path.append(pred)
            pred=predecessors.get(pred,None)
        print('shortest path: '+str(path)+" cost="+str(
            distances[dest]))
    else :
        if not visited:
            distances[src]=0
```

```
. . .

for neighbor in graph[src] :
  if neighbor not in visited:
  new_distance = distances[src] + graph[src][neighbor]
  if new_distance < distances.get(neighbor,float('inf'))
      :
    distances[neighbor] = new_distance
    predecessors[neighbor] = src

visited.append(src)
unvisited={}

for k in graph:
  if k not in visited:
    unvisited[k] = distances.get(k,float('inf'))
x=min(unvisited, key=unvisited.get)
dijkstra(graph,x,dest,visited,distances,predecessors)
```

Minimum Spanning Trees

- All the computers in an office must be connected using the least amount of cable (wireless is not admitted)
- This problem can be modeled using an undirected weighted graph G whose vertices represent the computers, and whose edges represent all the possible pairs (u, v) of computers
- The weight $w(u, v)$ of the edge (u, v) is equal to the amount of cable needed to connect computer u to computer v
- The problem is finding a tree that contains all the vertices of G and has the minimum total weight over all such trees

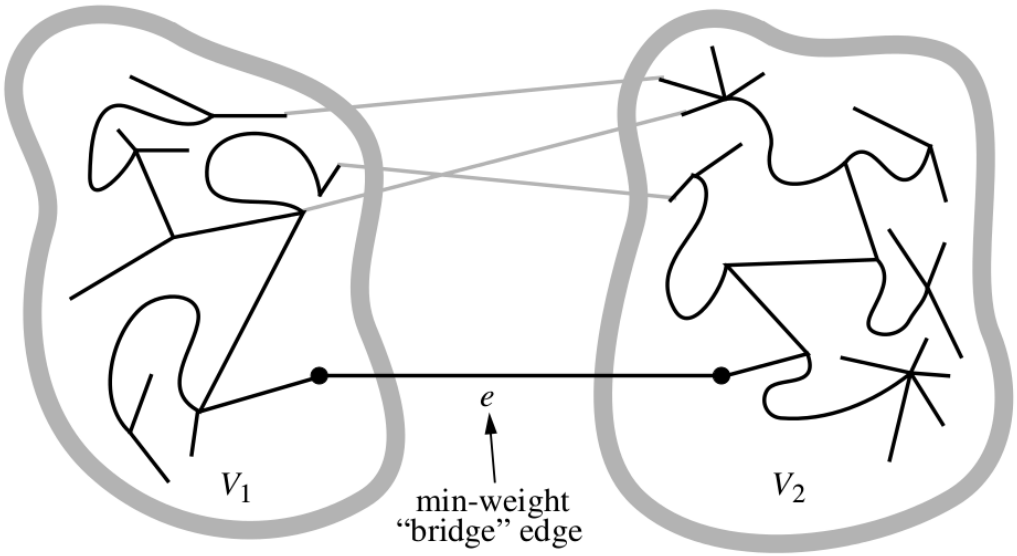
Minimum Spanning Trees

Given an undirected weighted graph G, we are interested in finding a tree T that contains all the vertices in G and minimizes the sum

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

Such a tree that contains every vertex of a connected graph G is said to be a **spanning tree**, and the problem of computing a spanning tree T with smallest total weight is known as the **minimum spanning tree (MST)** problem

Minimum Spanning Trees



Proposition

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

Minimum Spanning Trees

Prim-Jarnik Algorithm

- In this algorithm, a minimum spanning tree is grown starting from a single cluster starting from some "root" vertex s
- The main idea is similar to that of Djkstra's algorithm: a cloud of vertices is defined that grows at each iteration
- At each iteration, a minimum-weight edge is chosen $e = (u, v)$ which connects a vertex u in the cloud C to a vertex v outside the cloud C
- The vertex v is brought into the cloud C and the process is repeated until a spanning tree is formed
- As in Dijkstra'a algorithm, a label $D[v]$ is maintained for each vertex outside the cloud C, so that $D[v]$ stores the weight of the minimum observed edge for joining v to the cloud C

Minimum Spanning Trees

Prim-Jarnik Algorithm

```
Algorithm Prim_Jarnik(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 != s do
    D[v] = infinite
  Initialize T = 0
  Initialize a priority queue Q with an entry (D[v], (v,
       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
      if w(u,v) < D[v] do
```

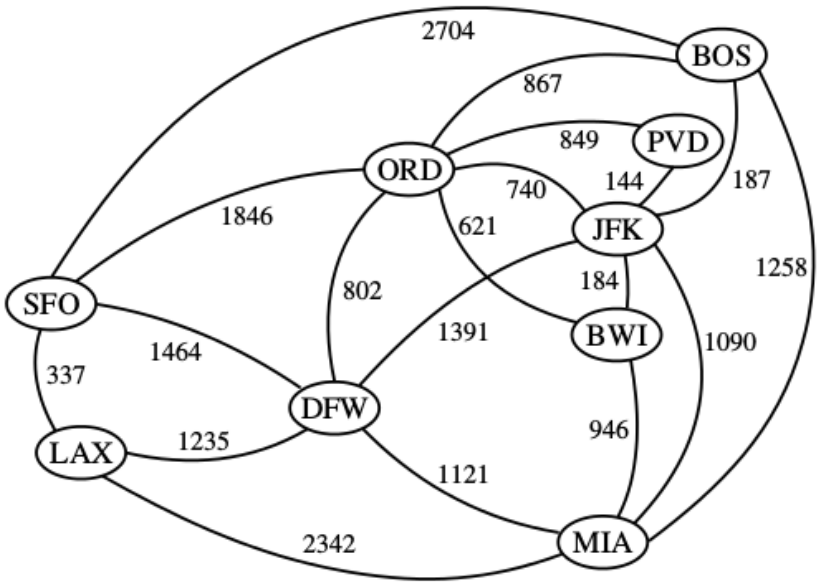
Minimum Spanning Trees

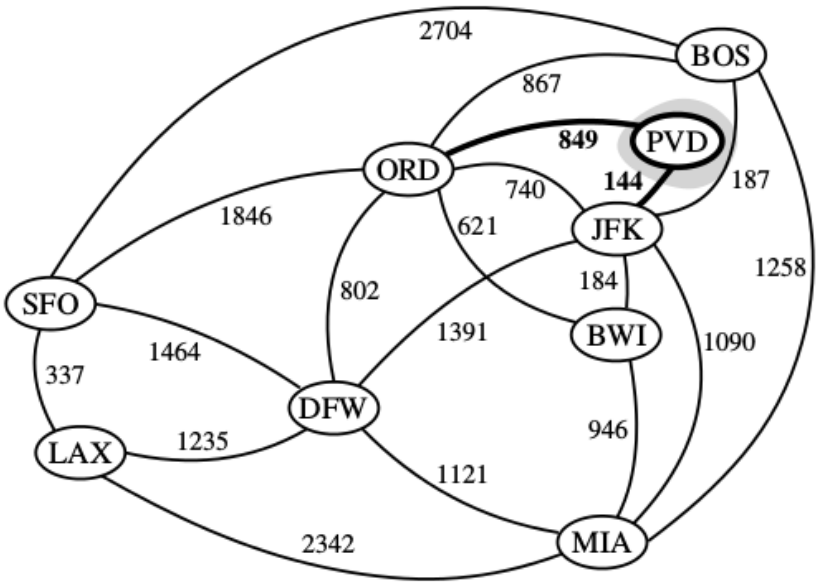
Analysis of the Prim-Jarnik Algorithm

- n insertions are performed into Q
- Later, n extract-min are performed
- A total of m priorities are updates
- With a priority queue, each operation runs in $O(logn)$, and the overall time for the algorithms is $O((n+m)logn)$, which is $O(mlogn)$ for a connected graph
- By using an unsorted list, the running time will be $O(n^2)$

Minimum Spanning Trees

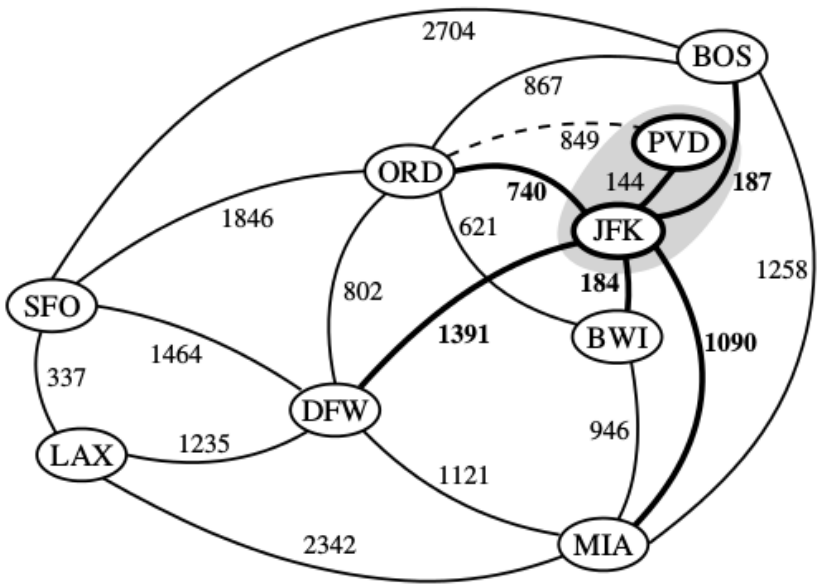
Prim-Jarnik Algorithm

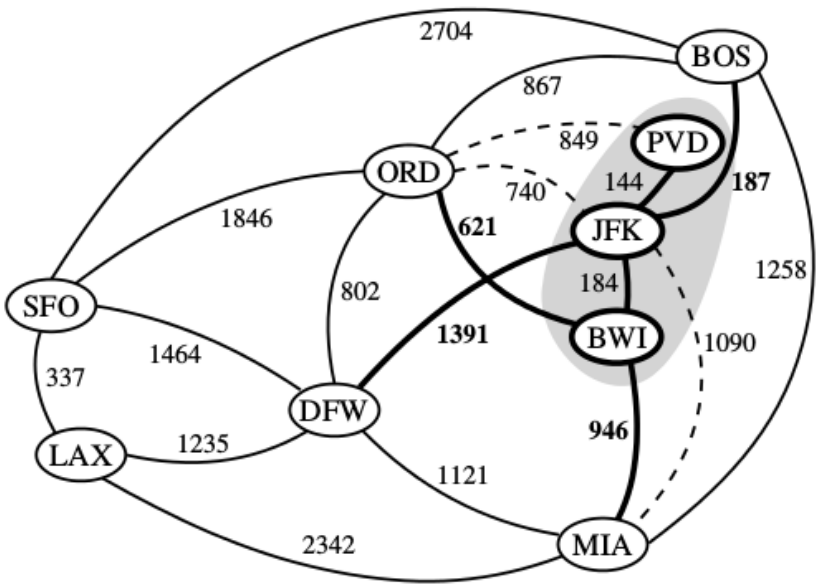




Minimum Spanning Trees

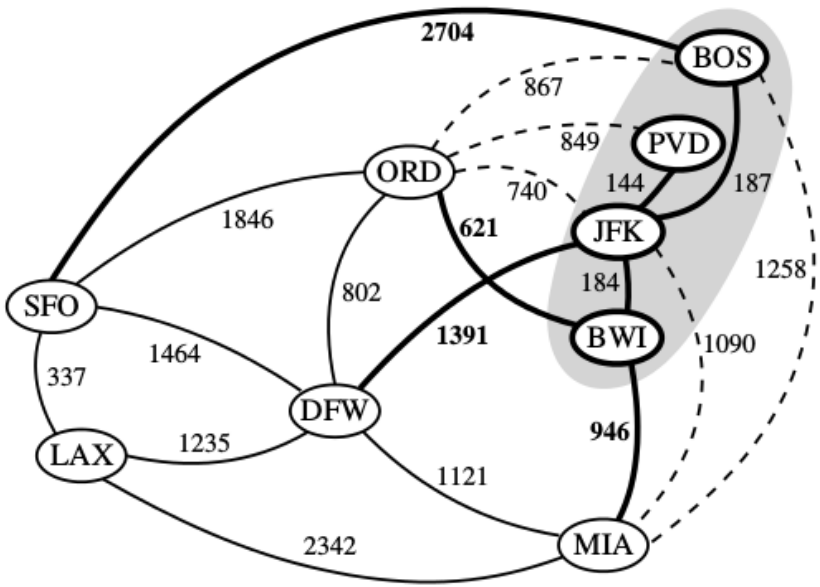
Prim-Jarnik Algorithm

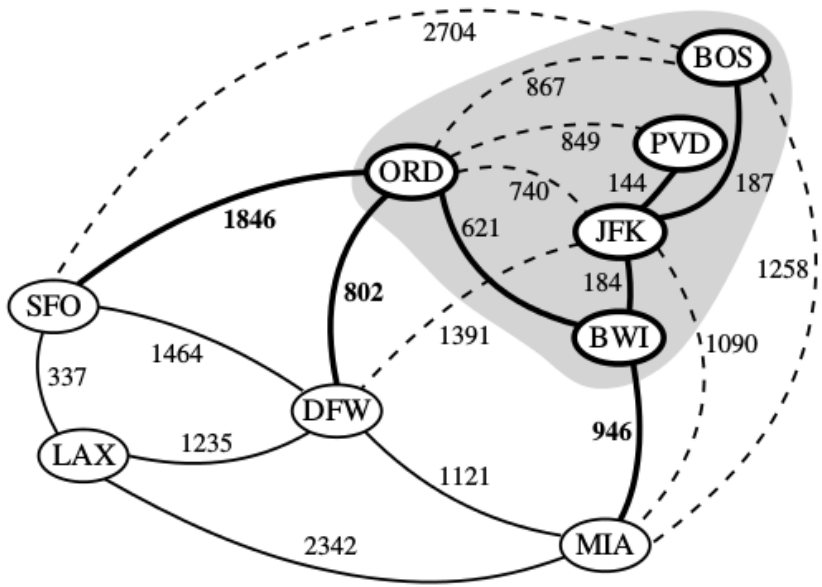




Minimum Spanning Trees

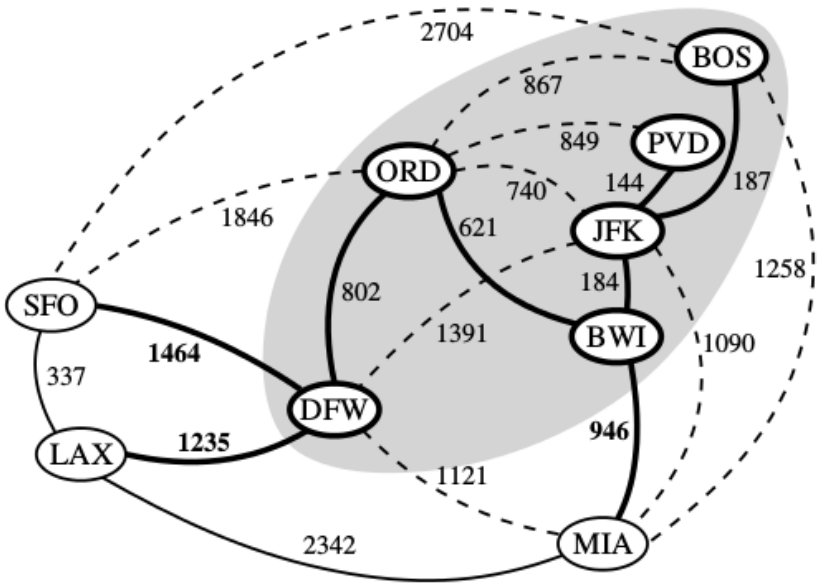
Prim-Jarnik Algorithm

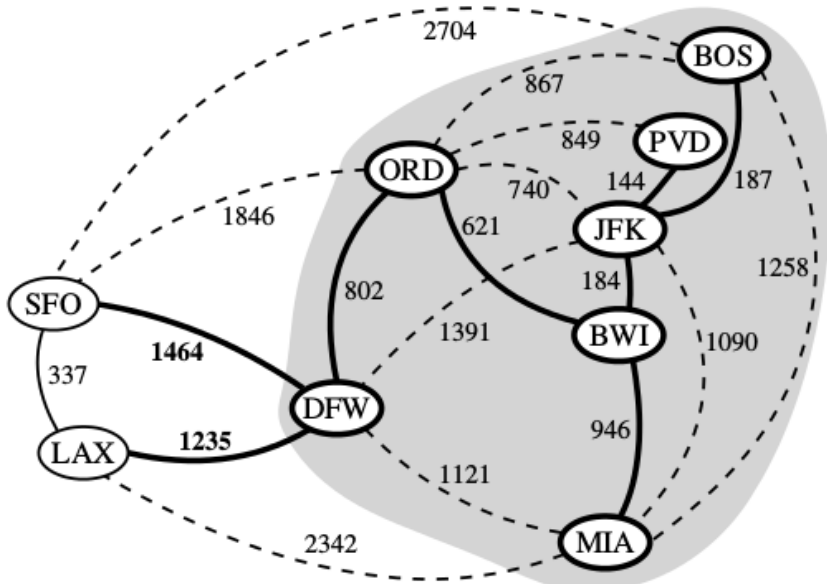




Minimum Spanning Trees

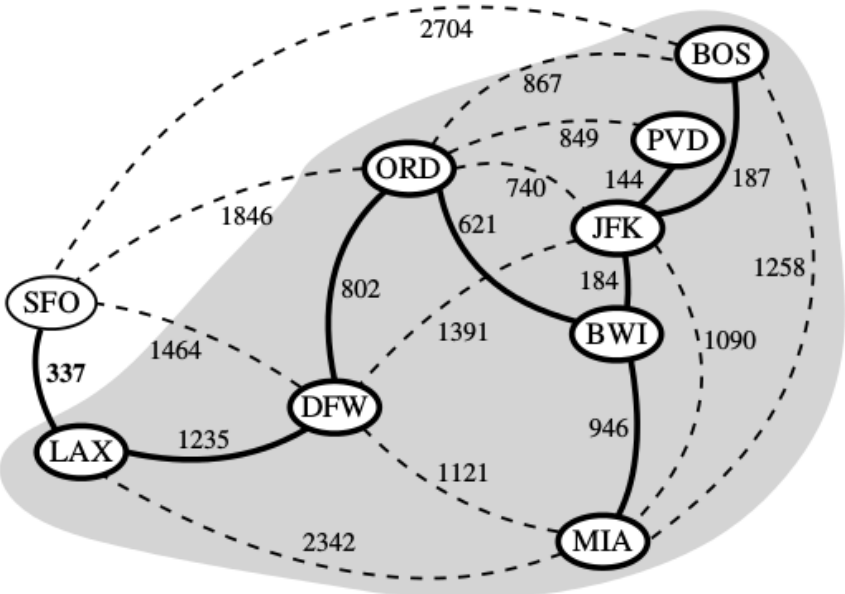
Prim-Jarnik Algorithm

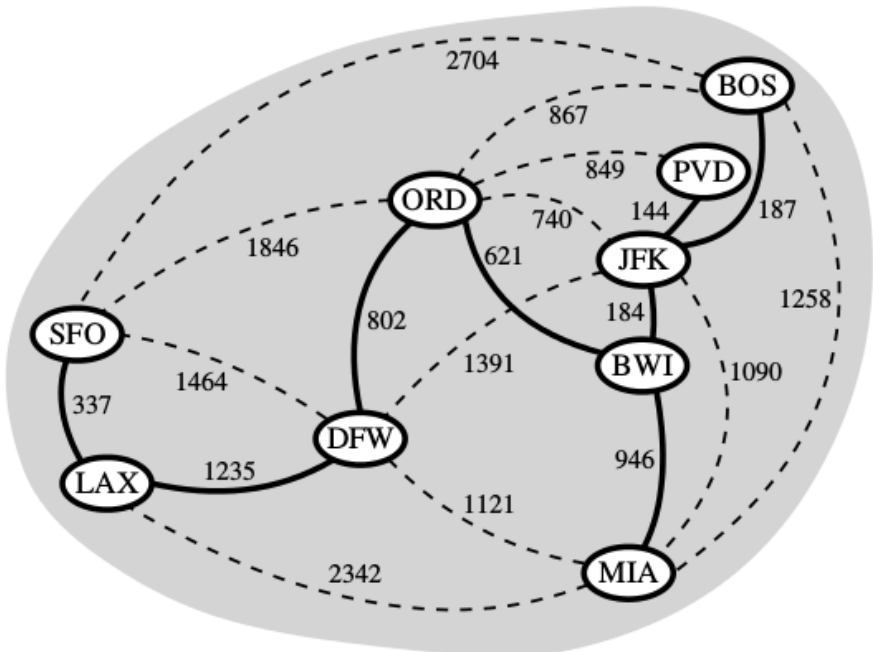




Minimum Spanning Trees

Prim-Jarnik Algorithm





Minimum Spanning Trees

Prim-Jarnik Algorithm

```
def MST_PrimJarnik(g):
  d = {}
  tree = []
  pq = AdaptableHeapPriorityQueue()
  pqlocator = {}

  for v in g.vertices():
    if len(d) == 0:
      d[v] = 0
    else:
      d[v] = float('inf')
    pqlocator[v] = pq.add(d[v], (v,None))

  . . .

(to be continued)
```

Minimum Spanning Trees

Prim-Jarnik Algorithm

```
. . .

  while not pq.is_empty():
    key,value = pq.remove_min()
    u,edge = value
    del pqlocator[u]
    if edge is not None:
      tree.append(edge)
    for link in g.incident_edges(u):
      v = link.opposite(u)
      if v in pqlocator:
        wgt = link.element()
        if wgt < d[v]:
          d[v] = wgt
          pq.update(pqlocator[v], d[v], (v, link))

  return tree
```

Minimum Spanning Trees

Kruskal's Algorithm

- Kruskal's algorithm maintains a **forest** of clusters, repeatedly merging pairs of clusters until a single cluster spans the graph
- Initially, each vertex is by itself a singleton cluster
- The algorithm considers each edge in turn, ordered by increasing weight
- If an edge e connects two different clusters, then e is added to the set of edges of the MST, and the clusters connected by e are merged into a single cluster
- On the contrary, if e connects two vertices that are already in the same cluster, then e is discarded
- The algorithms ends when enough edges have been added to form a spanning tree

Minimum Spanning Trees

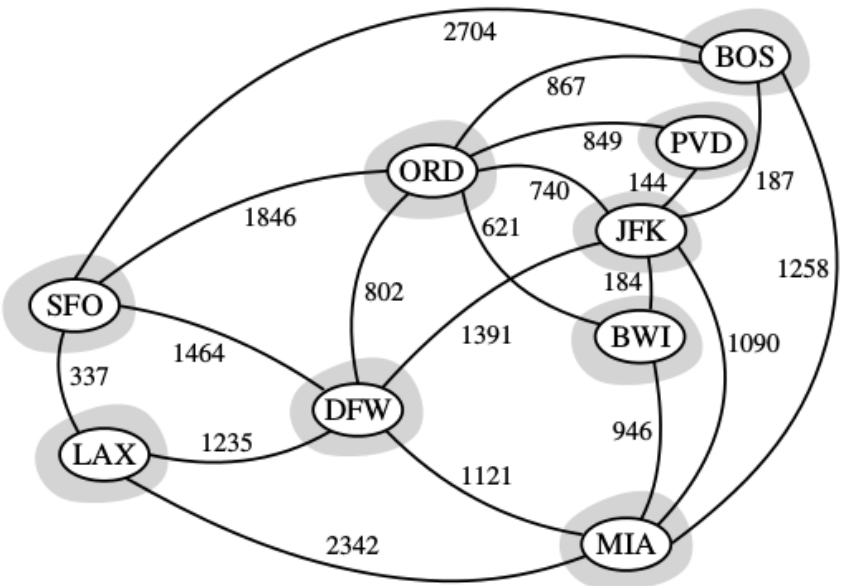
Kruskal's Algorithm

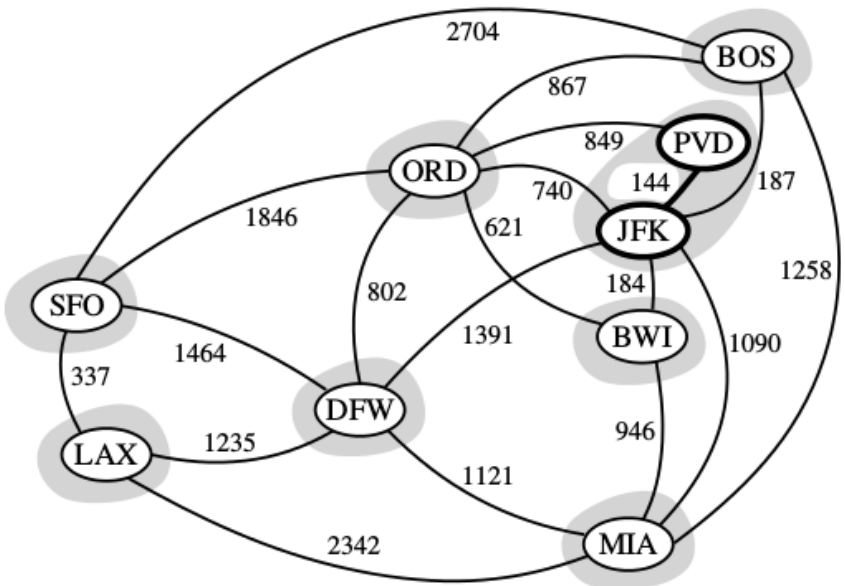
```
Algorithm Kruskal(G):
  Input: A simple connected weighted graph G
         with n vertices and m edges
  Output: A minimum spanning tree T for G

  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 = 0
    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 clustre containing v
      if C(u) != C(v) then
        Add edge (u,v) to T
        Merge C(u) and C(v) into one cluster
    return tree T
```

- The correctness of Kruskal's algorithm is based upon the crucial fact about minimum spanning trees from Proposition
- Each time that Kruskal's algorithm adds an edge (u, v) to the MST T, we can define a partitioning of the set of vertices V by letting V_1 be the cluster containing v and letting V_2 contain the rest of the vertices in V
- This defines a disjoint partitioning of the vertices of V
- Moreover, since we are extracting edges from Q in order by their weights, e must be a minimum-weight edge with one vertex in V_1 and the other in V_2.
- Therefore, Kruskal's algorithm always adds a valid MST edge

Minimum Spanning Trees

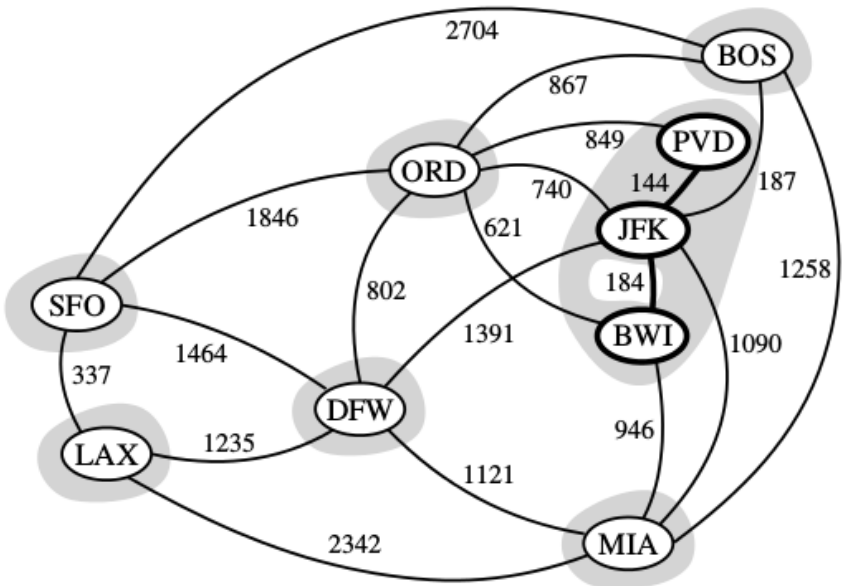
Kruskal's Algorithm

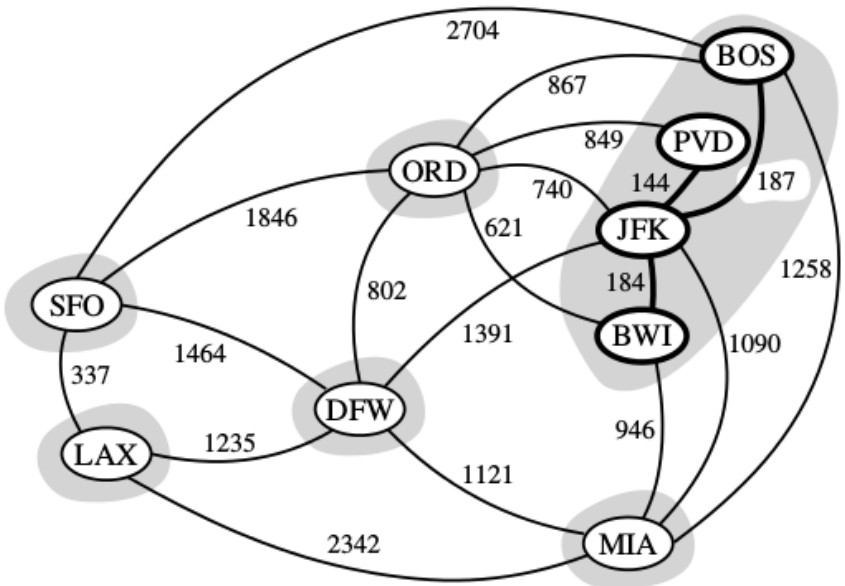




Minimum Spanning Trees

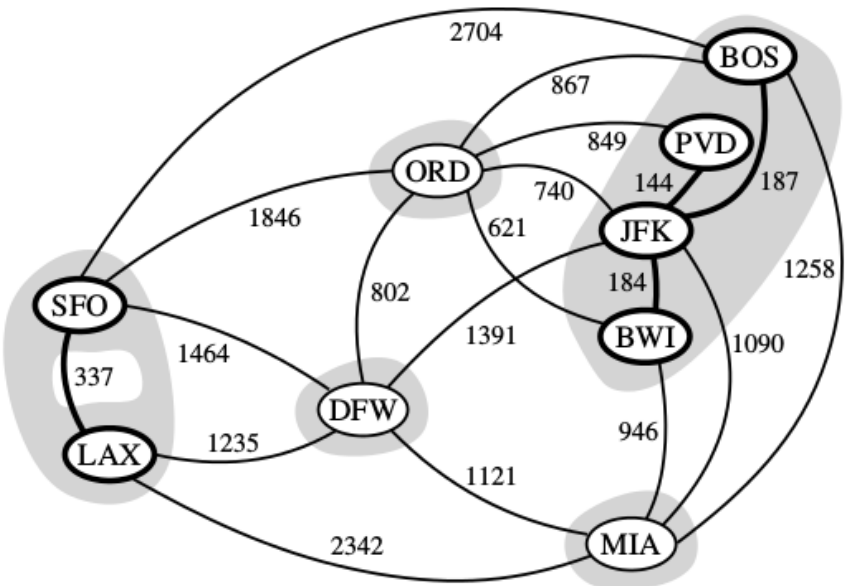
Kruskal's Algorithm

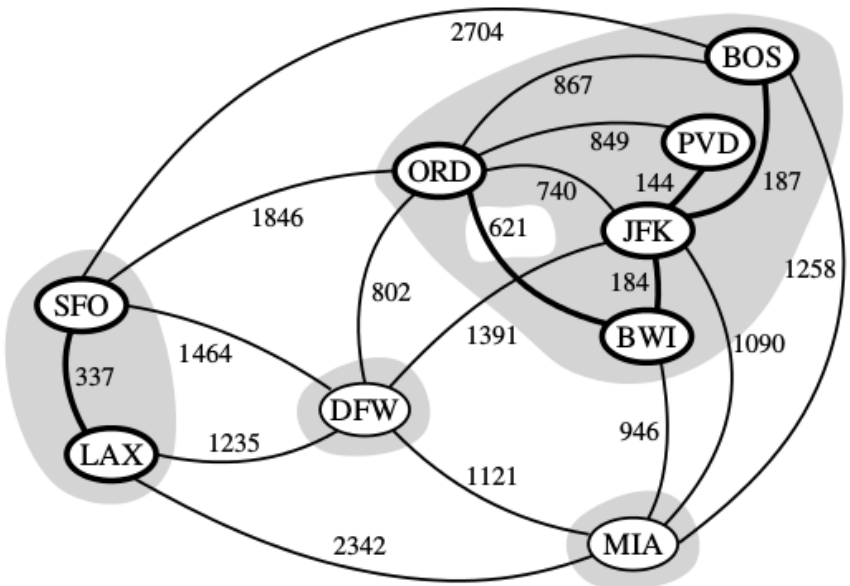




Minimum Spanning Trees

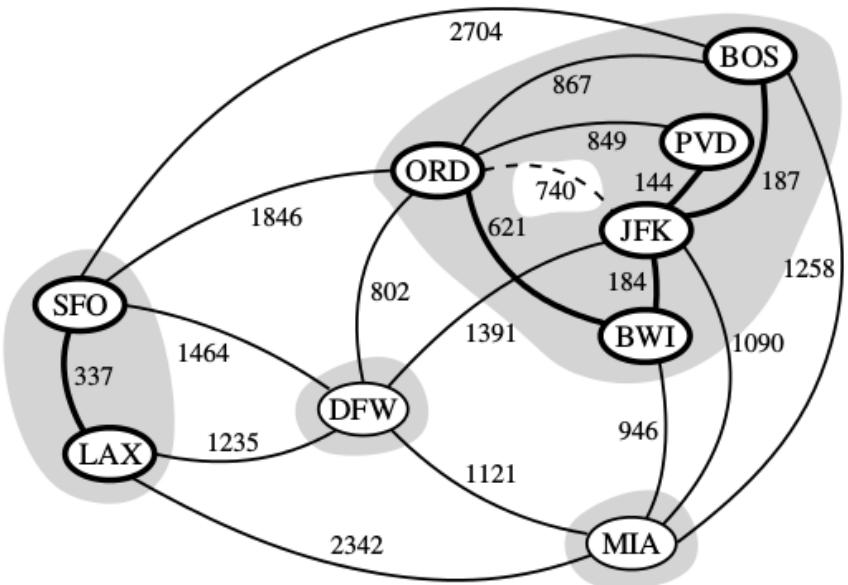
Kruskal's Algorithm

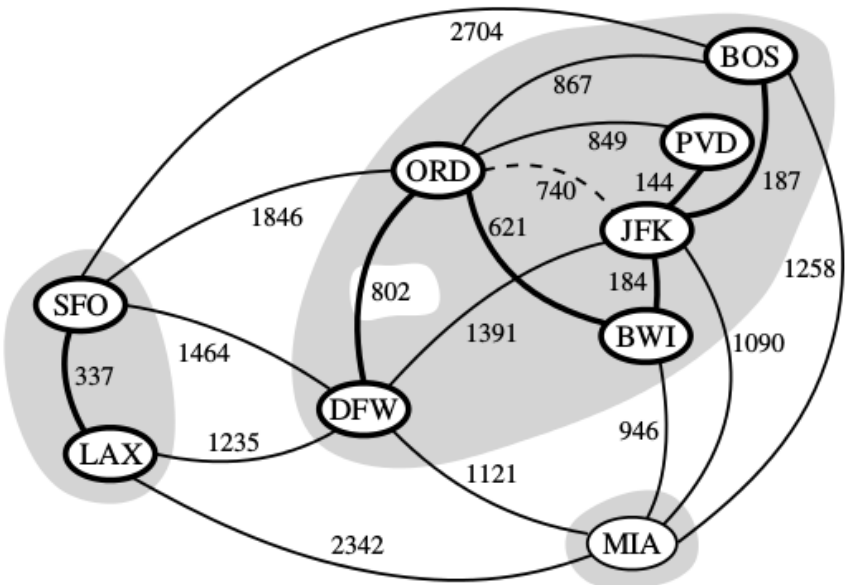




Minimum Spanning Trees

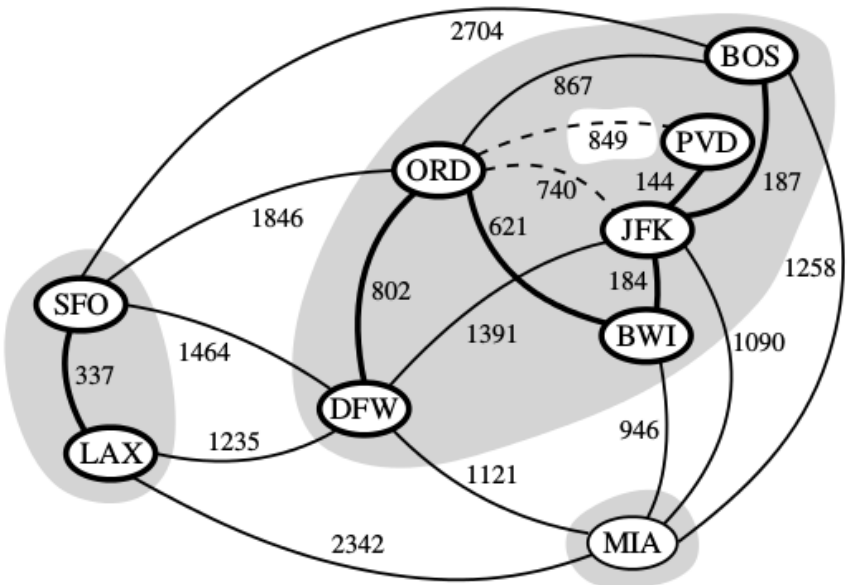
Kruskal's Algorithm

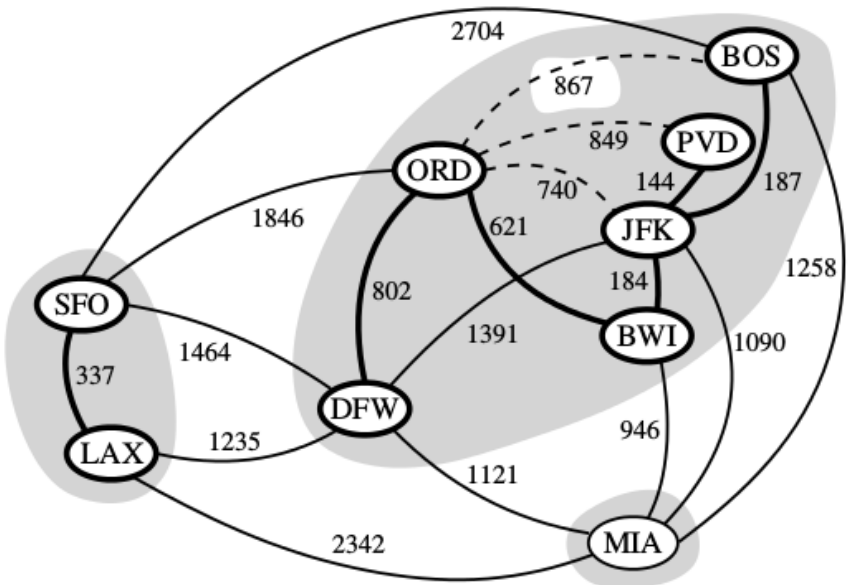




Minimum Spanning Trees

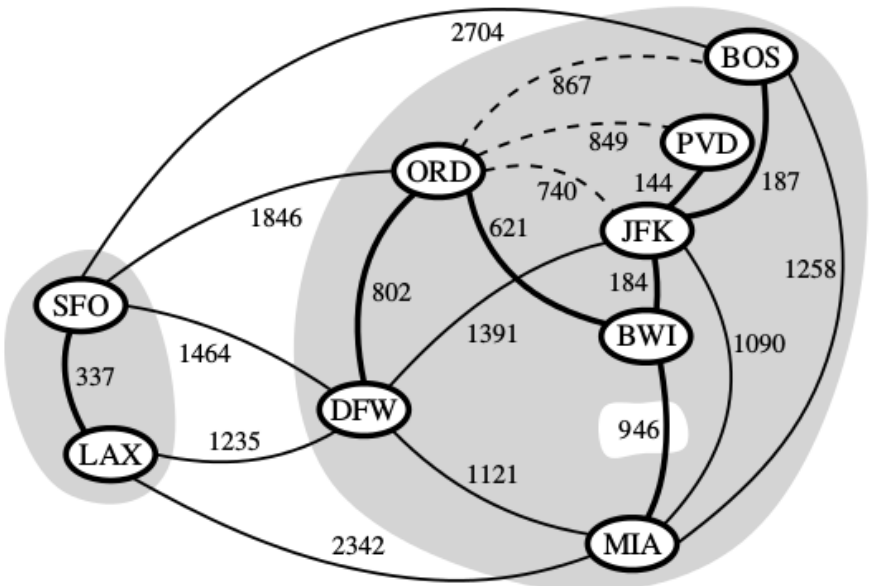
Kruskal's Algorithm

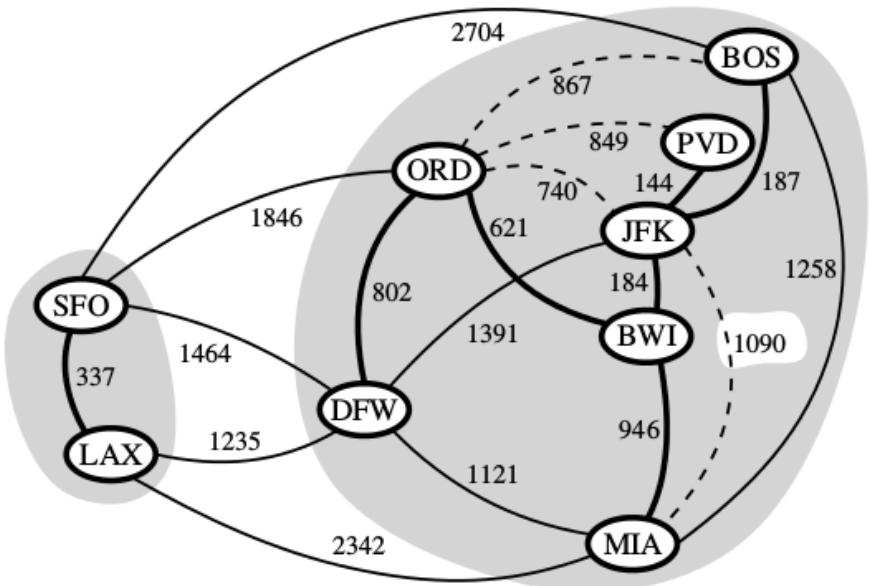




Minimum Spanning Trees

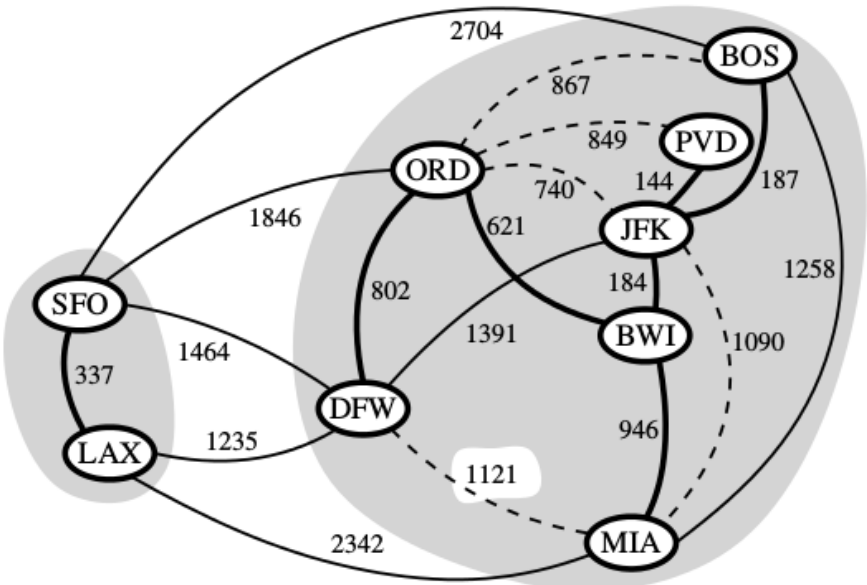
Kruskal's Algorithm

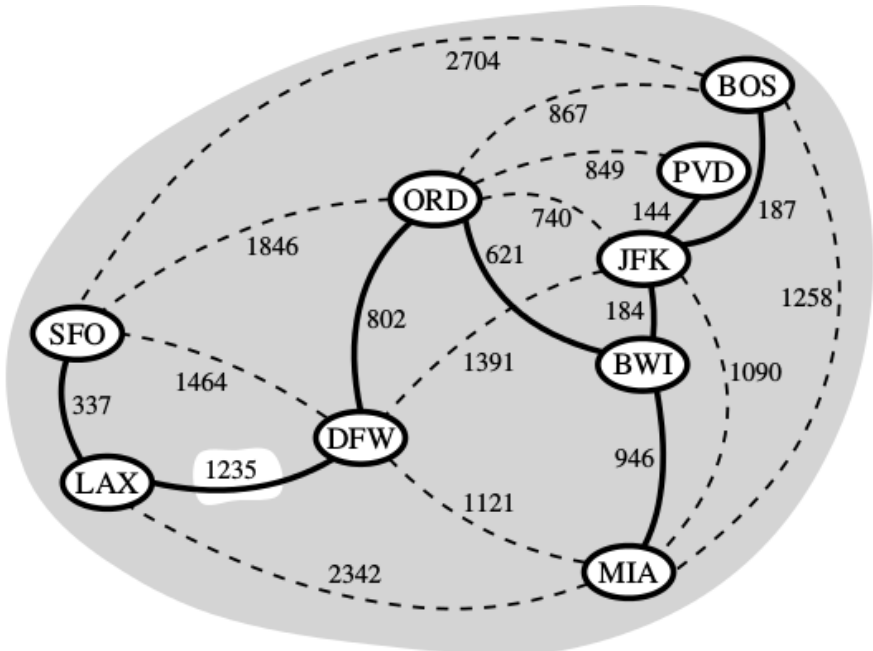




Minimum Spanning Trees

Kruskal's Algorithm





Minimum Spanning Trees

Analysis of Kruskal's Algorithm

- The ordering of edges can be implemented in $O(mlogn)$ (by using a priority queue)
- To implement Kruskal's algorithm, we must be able to find the cluster for vertices u and v that are endpoints of an edge e, to test whether those two clusters are distinct, and if so, to merge those two cluster into one
- The management of **disjoint partitions** can be performed in $O(m + nlogn)$
- For a connected graph we have $m \geq n - 1$ and therefore the dominant term is $O(mlogn)$, which is the running time of Kruskal's algorithm

Minimum Spanning Trees

Kruskal's Algorithm

```
def MST_Kruskal(g):
  tree = []
  pq = HeapPriorityQueue()
  forest = Partition()
  position = {}

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

  for e in g.edges():
    pq.add(e.element(), e)

  size = g.vertex_count()
  while len(tree) != size - 1 and not pq.is_empty():
    weight,edge = pq.remove_min()
    u,v = edge.endpoints()
    a = forest.find(position[u])
    b = forest.find(position[v])
    if a != b:
```

- A partition data structure manages an universe of elements that are organized into disjoint sets (an element belongs to one and only one of these sets)
- For clarity reason, the clusters of a partition are referred to as **groups**
- To differentiate between one group and another, we assume that at any point in time, each group has a designated entry called **leader** of the group

Method	Functionality
`make_group(x)`	Create a singleton group containing 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