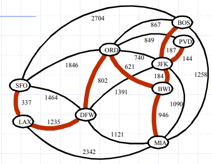
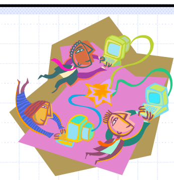
Minimum Spanning Trees



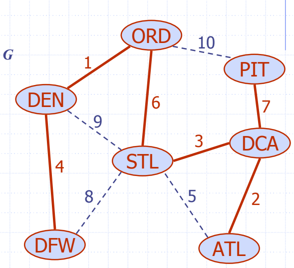
Outline and Reading



- Minimum Spanning Trees (§7.3)
  - Definitions
  - A crucial fact
- The Prim-Jarnik Algorithm (§7.3.2)
- Kruskal's Algorithm (§7.3.1)
- Baruvka's Algorithm (§7.3.3)

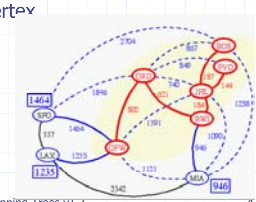
Minimum Spanning Tree

- 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

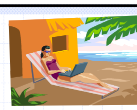


Prim-Jarnik's Algorithm

- Like Dijkstra's algorithm only simpler
- Grow the MST from arbitrary vertex s
- Greedily add vertices into cloud based on distance to any vertex in cloud
- At v, need to store $d(v)$ = minimum weight edge connecting v to a cloud vertex
- 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 (edge relaxation)

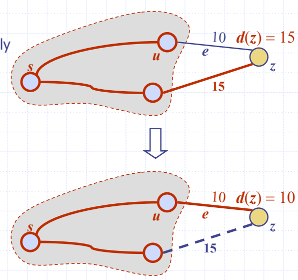


Prim's Edge Relaxation

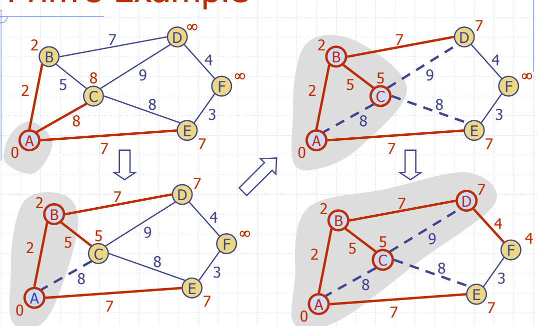


- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

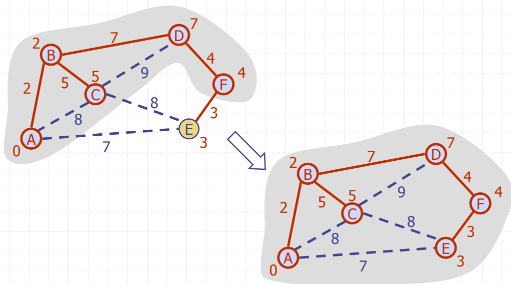
$d(z) \leftarrow \min\{d(z), e\}$



Prim's Example



Example (contd.)



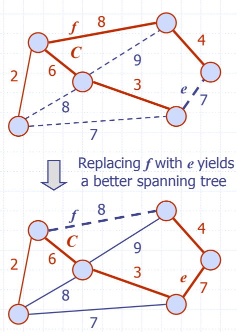
Cycle Property

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, $weight(f) \le weight(e)$

Proof:

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



Correctness of Prim's

Let T_k be tree produced by Prim's after kth iteration. Let G_k be the the subgraph of G induced by T_k. Then T_k is a MST of G_k.

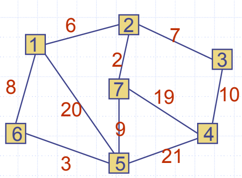
Prim-Jarnik's Algorithm (cont.)

- 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
  - Tree 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, ∞)
    setTreeEdge(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)
        setTreeEdge(z,e)
        Q.replaceKey(getLocator(z),r)
```


Example graph

Start at 1, run Prim's



Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the distance, tree and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex inserted and removed once taking $O(\log n)$ time each time for 2n times.
  - 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
- The running time is $O(m \log n)$ since the graph is connected
- What is running time for unsorted-sequence based priority queue?

Kruskal's MST algorithm

- Another greedy strategy for finding MST
- Gradually turn forest into tree as edges are added
- Add cheapest edge possible
  - Don't add edge if it forms cycle
- Overview:

```
kruskalMST (Graph G)
    Initalize F (forest) to empty.
    Place all edges in PQ according to cost
    For each edge (u,v) in PQ (in sorted order)
        if (u,v) does not make a cycle in F
            add (u,v) to F
    return F;
```

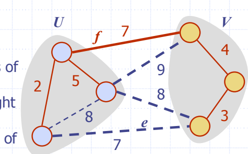

Partition Property

Partition Property:

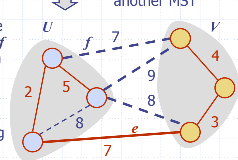
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- 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



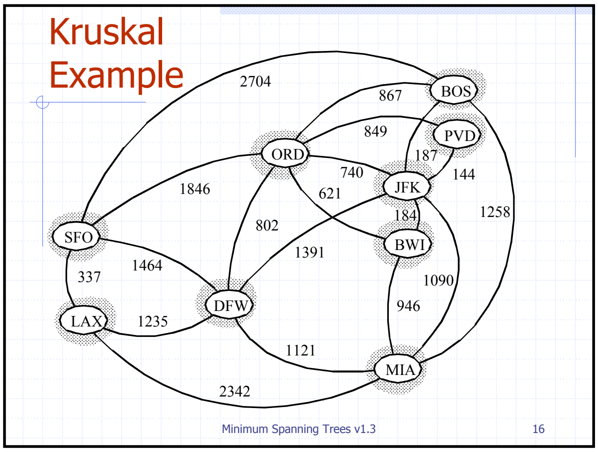
Replacing f with e yields another MST

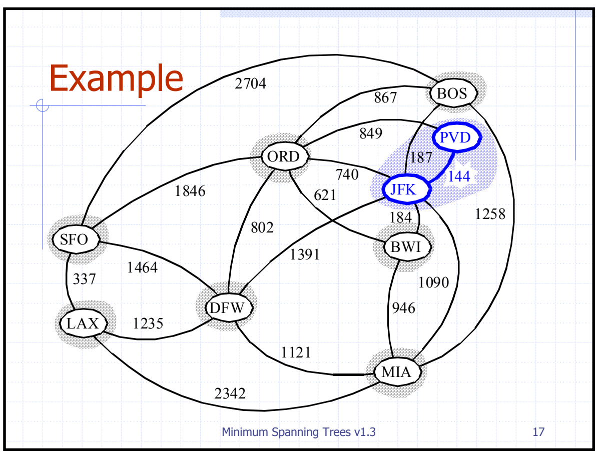


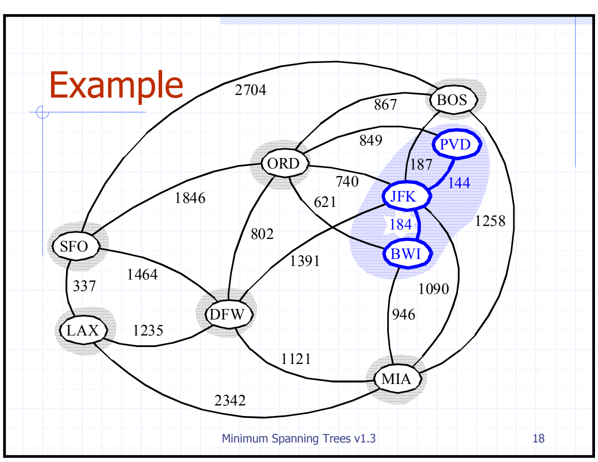
Kruskal's Algorithm

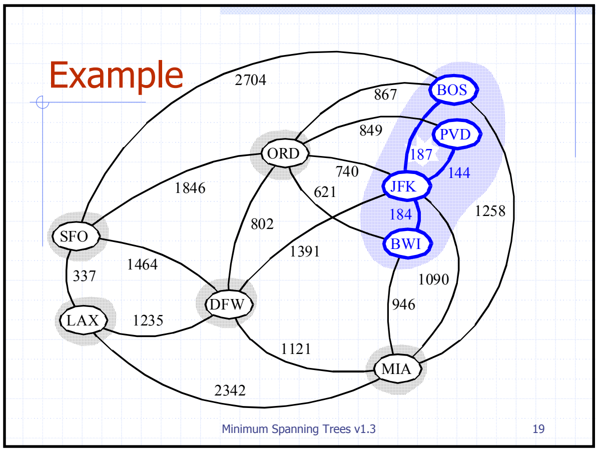
- Each vertex starts in its own cloud (a partition)
- Clouds merge together as edges are added
- A priority queue stores edges in weight order
  - Key: weight
  - Element: edge
- Only edges between clouds will not form cycles
  - add cheapest edge between clouds
- At end of algorithm:
  - All vertices in one cloud
  - Edges added form MST

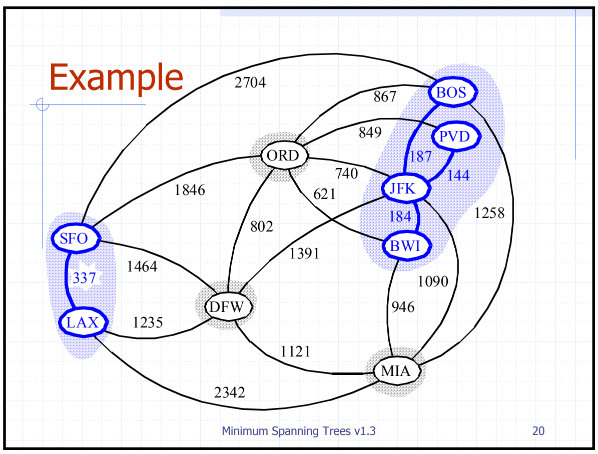
Algorithm *KruskalMST(G)*
  for each vertex V in G **do**
    define a ***Cloud(v)*** of $\leftarrow \{v\}$
  let Q be a priority queue.
  Insert all edges into Q using their weights as the key
  $T \leftarrow \varnothing$
  while T has fewer than n-1 edges **do**
    edge e = ***T.removeMin()***
  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

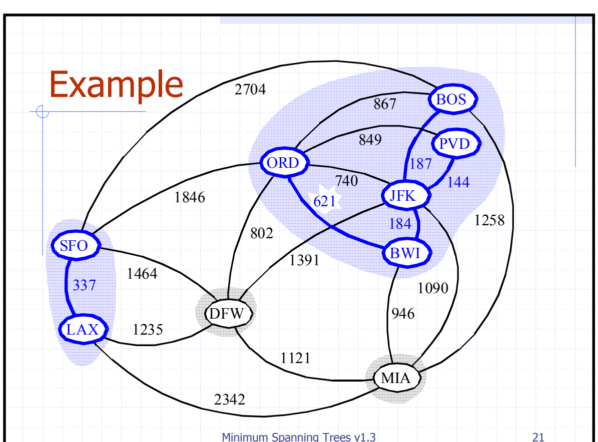

Kruskal Example
BOS
PVD
ORD
JFK
BWI
SFO
DFW
LAX
MIA
2704
867
849
187
144
740
621
1846
184
1258
802
1391
1464
337
1090
946
1235
1121
2342
Minimum Spanning Trees v1.3
16

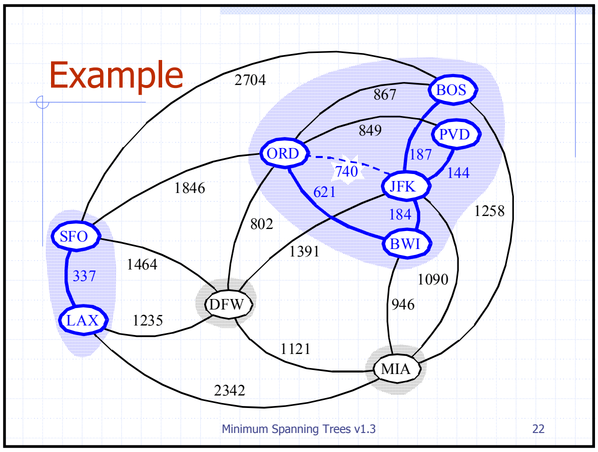

Example
BOS
PVD
ORD
JFK
BWI
SFO
DFW
LAX
MIA
2704
867
849
187
144
740
621
1846
184
1258
802
1391
1464
337
1090
946
1235
1121
2342
Minimum Spanning Trees v1.3
17

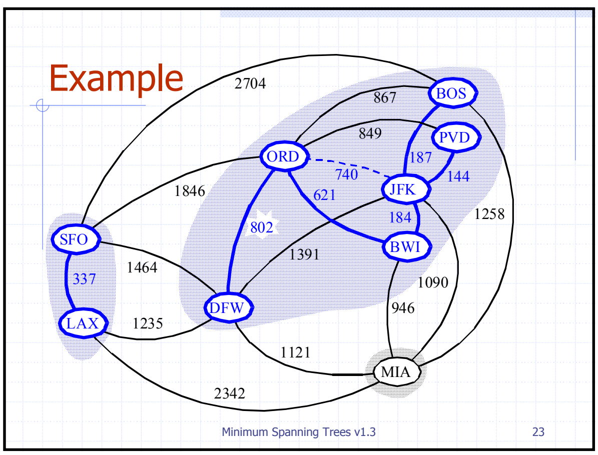

Example
BOS
PVD
ORD
JFK
BWI
SFO
DFW
LAX
MIA
2704
867
849
187
144
740
621
1846
184
1258
802
1391
1464
337
1090
946
1235
1121
2342
Minimum Spanning Trees v1.3
18

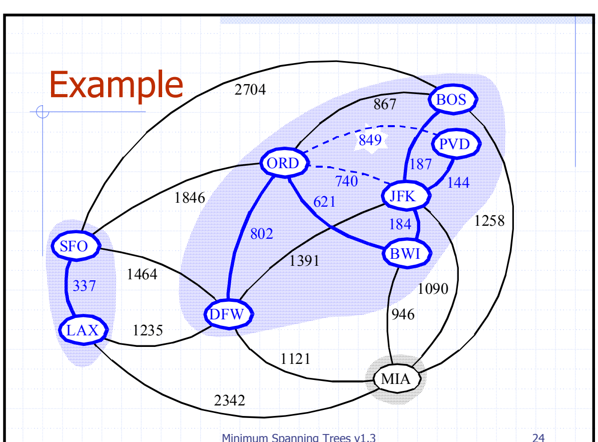

Example
2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
184
1258
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
19

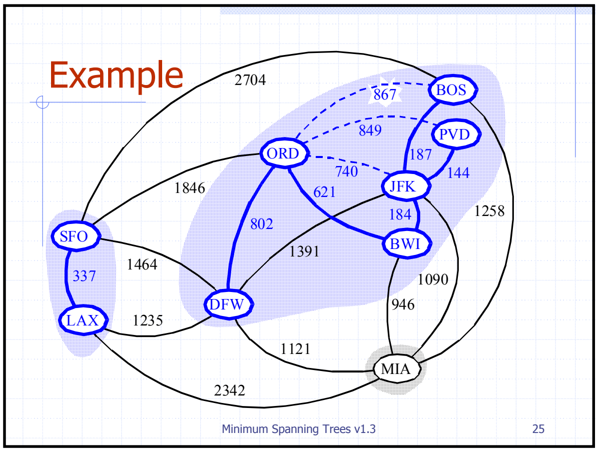

Example
2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
184
1258
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
20

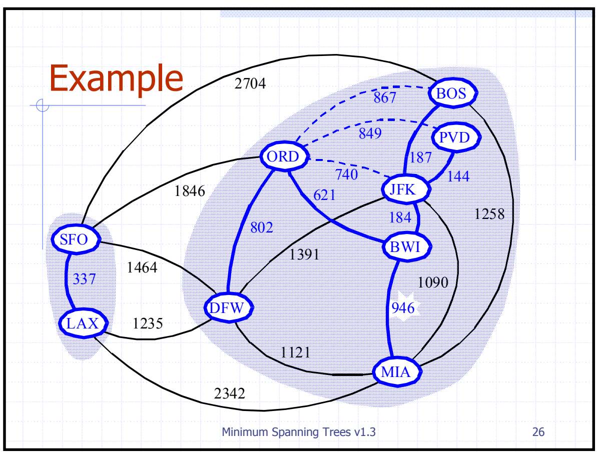

Example
2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
184
1258
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
21

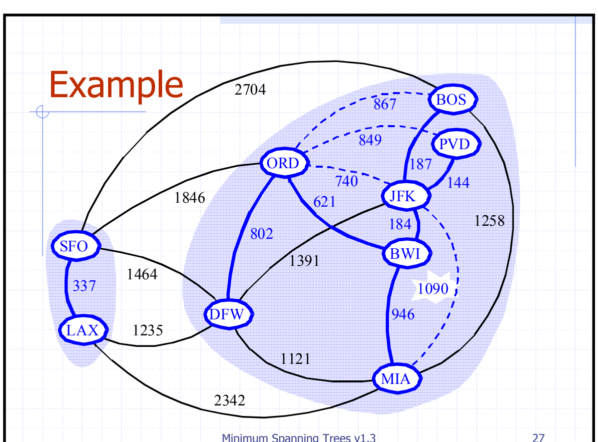

Example
2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
22


Example
2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
23

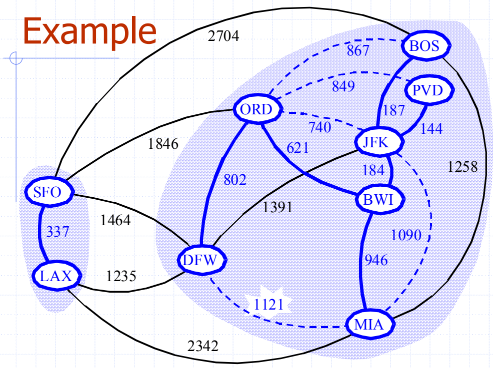

Example
2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
24


Example
2704
867
BOS
849
PVD
ORD
187
740
JFK
144
1846
621
1258
184
802
SFO
BWI
1391
1464
1090
337
946
DFW
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
25

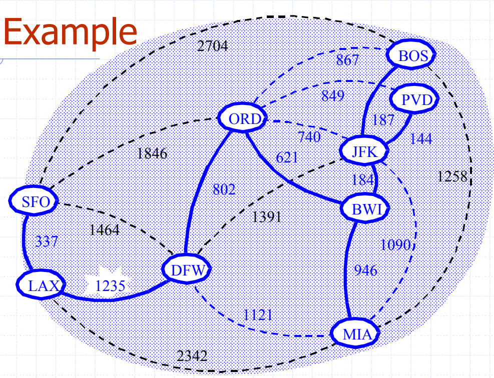

Example
2704
867
BOS
849
PVD
ORD
187
740
JFK
144
1846
621
1258
184
802
SFO
BWI
1391
1464
1090
337
946
DFW
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
26


Example
2704
867
BOS
849
PVD
ORD
187
740
JFK
144
1846
621
1258
184
802
SFO
BWI
1391
1464
1090
337
946
DFW
LAX
1235
1121
MIA
2342
Minimum Spanning Trees v1.3
27

Example



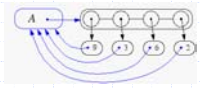
Example



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.
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, sorted by their weights
Let T be an initially-empty tree
while Q is not empty 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(m log n)

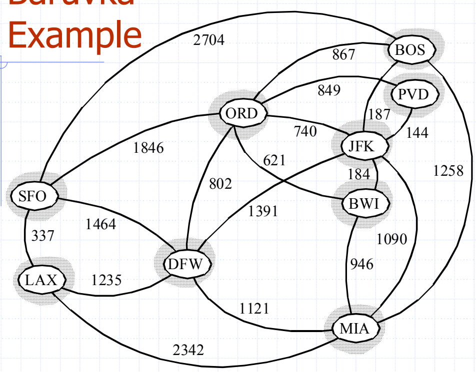
Baruvka's Algorithm

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

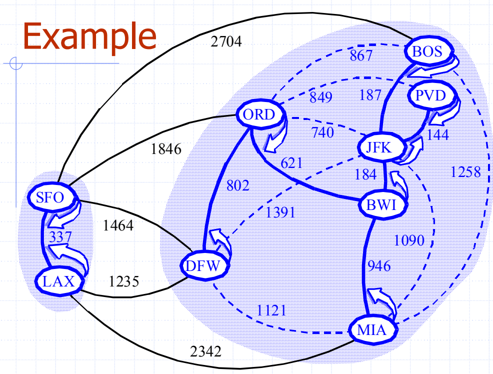
```
Algorithm BaruvkaMST(G)
    T ← V {just the vertices of G}
  while T has fewer than n-1 edges do
    for each connected component C in T do
      Let edge e be the smallest-weight edge from C to another component in T.
      if e is not already in T then
        Add edge e to T
  return T
```

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

Baruvka Example



Example



Example