

Algorithms and Data Structures

Weighted Graphs & Shortest Paths

Dr. Bernhard Anrig

HS 2012/13

Outline

Definitions and Problems

Dijkstra's Algorithm

Bellman-Ford Algorithm

Minimal Spanning Tree

Prim-Jarnik Algorithm

Kruskals Algorithm

Outline

Definitions and Problems

Dijkstra's Algorithm

Bellman-Ford Algorithm

Minimal Spanning Tree

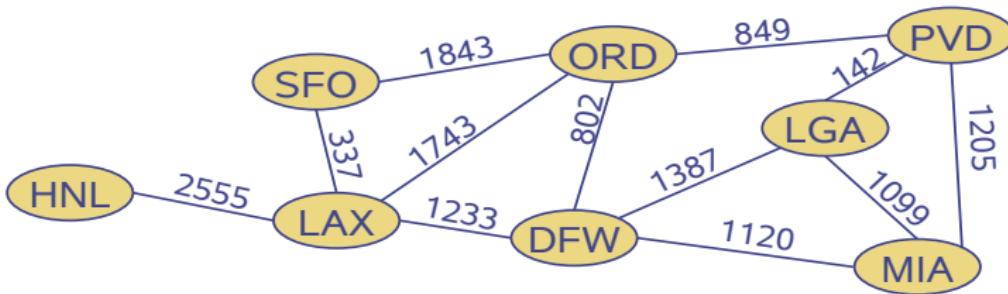
Prim-Jarnik Algorithm

Kruskals Algorithm



Weighted Graphs

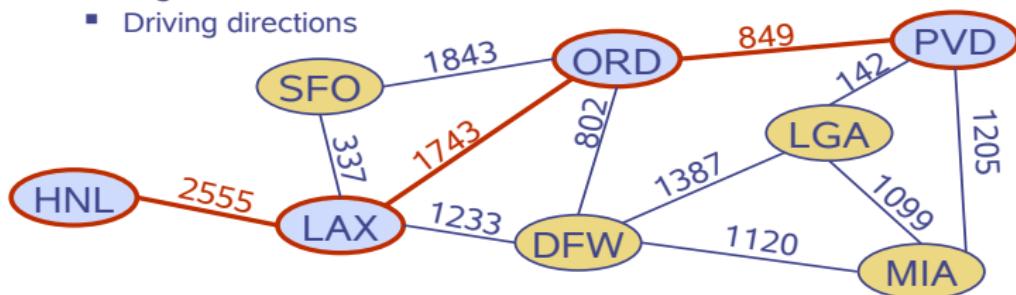
- ◆ In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- ◆ Edge weights may represent, distances, costs, etc.
- ◆ Example:
 - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports





Shortest Path Problem

- ◆ Given a weighted graph and two vertices u and v , we want to find a path of minimum total weight between u and v .
 - Length of a path is the sum of the weights of its edges.
- ◆ Example:
 - Shortest path between Providence and Honolulu
- ◆ Applications
 - Internet packet routing
 - Flight reservations
 - Driving directions



Shortest Path Properties



Property 1:

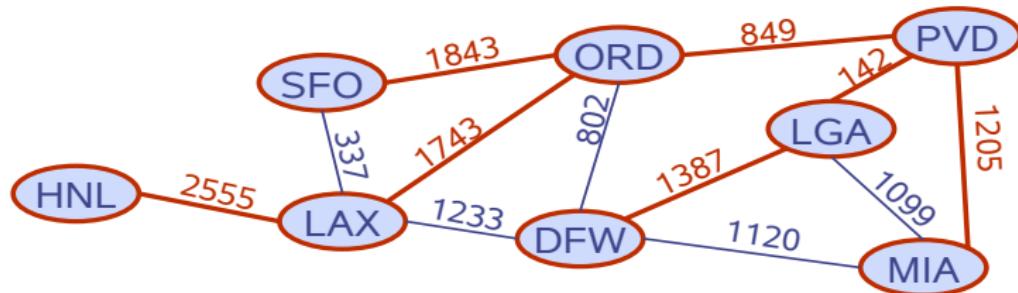
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



Outline

Definitions and Problems

Dijkstra's Algorithm

Bellman-Ford Algorithm

Minimal Spanning Tree

Prim-Jarnik Algorithm

Kruskals Algorithm

Dijkstra's Algorithm



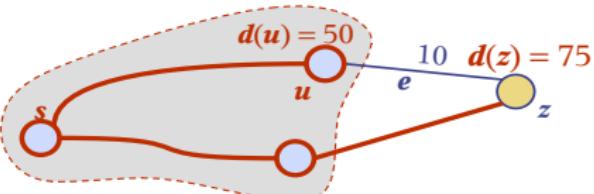
- ◆ The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- ◆ Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- ◆ Assumptions:
 - the graph is connected
 - the edges are undirected
 - the edge weights are nonnegative
- ◆ We grow a “**cloud**” of vertices, beginning with s and eventually covering all the vertices
- ◆ We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- ◆ At each step
 - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
 - We update the labels of the vertices adjacent to u

Edge Relaxation



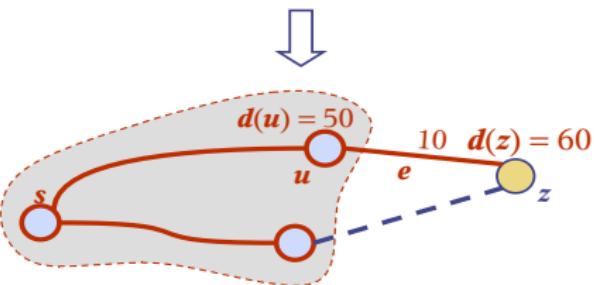
◆ Consider an edge $e = (u, z)$ such that

- u is the vertex most recently added to the cloud
- z is not in the cloud

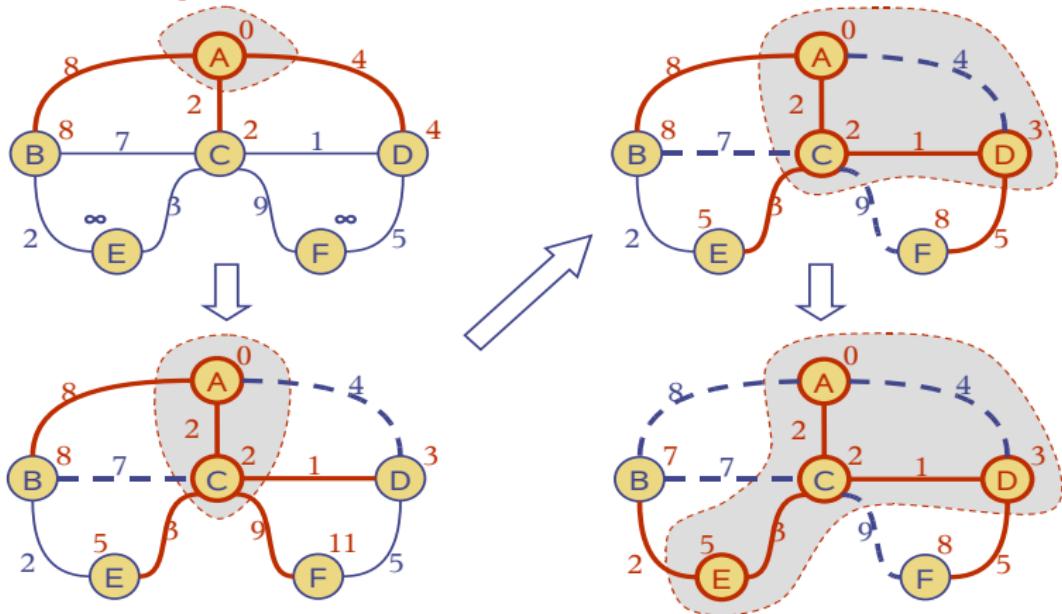


◆ The relaxation of edge e updates distance $d(z)$ as follows:

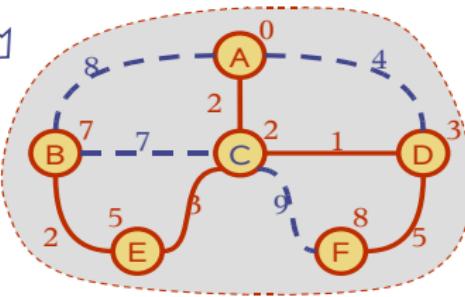
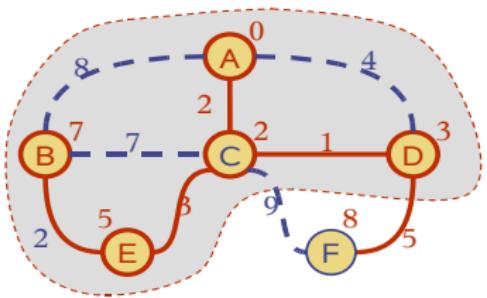
$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



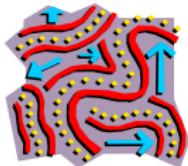
Example



Example (cont.)



Dijkstra's Algorithm



- ◆ A priority queue stores the vertices outside the cloud
 - Key: distance
 - Element: vertex
- ◆ Locator-based methods
 - *insert(l,e)* returns a locator
 - *replaceKey(l,k)* changes the key of an item
- ◆ We store two labels with each vertex:
 - Distance ($d(v)$ label)
 - locator in priority queue

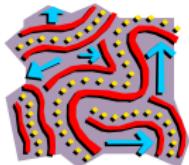
Algorithm *DijkstraDistances(G, s)*

```

 $Q \leftarrow$  new heap-based priority queue
for all  $v \in G.vertices()$ 
  if  $v = s$ 
    setDistance( $v, 0$ )
  else
    setDistance( $v, \infty$ )
   $l \leftarrow Q.insert(getDistance(v), v)$ 
  setLocator( $v, l$ )
while  $\neg Q.isEmpty()$ 
   $u \leftarrow Q.removeMin()$ 
  for all  $e \in G.incidentEdges(u)$ 
    { relax edge  $e$  }
     $z \leftarrow G.opposite(u, e)$ 
     $r \leftarrow getDistance(u) + weight(e)$ 
    if  $r < getDistance(z)$ 
      setDistance( $z, r$ )
      Q.replaceKey(getLocator( $z$ ),  $r$ )

```

Analysis



- ◆ Graph operations
 - Method `incidentEdges` is called once for each vertex
- ◆ Label operations
 - We set/get the distance and locator 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 in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- ◆ Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
 - Recall that $\sum_v \deg(v) = 2m$
- ◆ The running time can also be expressed as $O(m \log n)$ since the graph is connected

Extension



- ◆ Using the template method pattern, we can extend Dijkstra's algorithm to return a tree of shortest paths from the start vertex to all other vertices
- ◆ We store with each vertex a third label:
 - parent edge in the shortest path tree
- ◆ In the edge relaxation step, we update the parent label

Algorithm *DijkstraShortestPathsTree(G, s)*

...

for all $v \in G.vertices()$

...

setParent(v, Ø)

...

for all $e \in G.incidentEdges(u)$

{ relax edge e }

$z \leftarrow G.opposite(u,e)$

$r \leftarrow getDistance(u) + weight(e)$

if $r < getDistance(z)$

$setDistance(z,r)$

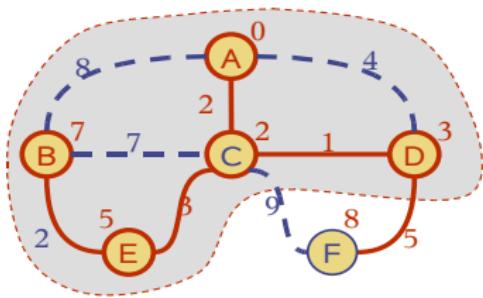
setParent(z,e)

Q.replaceKey(getLocator(z),r)

Why Dijkstra's Algorithm Works



- ◆ Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
 - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
 - When the previous node, D, on the true shortest path was considered, its distance was correct.
 - But the edge (D,F) was **relaxed** at that time!
 - Thus, so long as $d(F) \geq d(D)$, F's distance cannot be wrong. That is, there is no wrong vertex.

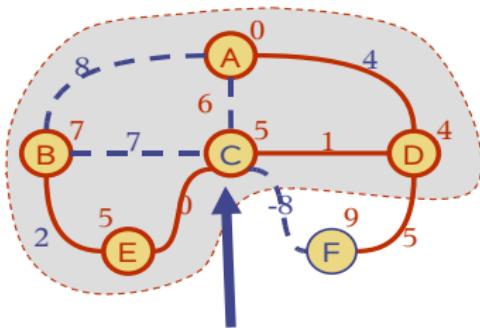


Why It Doesn't Work for Negative-Weight Edges



- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with $d(C)=5$!

Outline

Definitions and Problems

Dijkstra's Algorithm

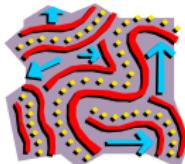
Bellman-Ford Algorithm

Minimal Spanning Tree

Prim-Jarnik Algorithm

Kruskals Algorithm

Bellman-Ford Algorithm



- ◆ Works even with negative-weight edges
- ◆ Must assume directed edges (for otherwise we would have negative-weight cycles)
- ◆ Iteration i finds all shortest paths that use i edges.
- ◆ Running time: $O(nm)$.
- ◆ Can be extended to detect a negative-weight cycle if it exists
 - How?

Algorithm *BellmanFord(G, s)*

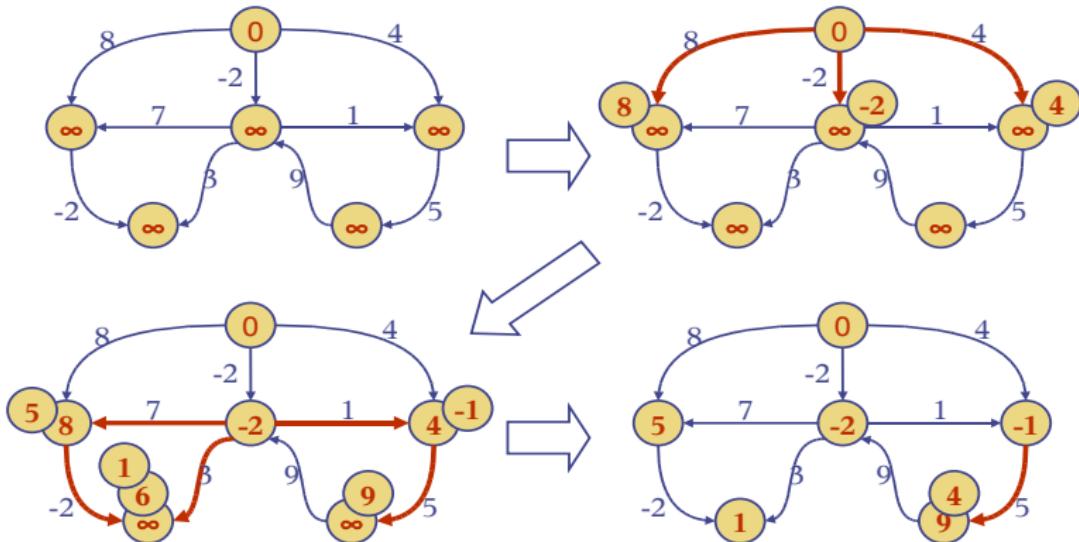
```

for all  $v \in G.vertices()$ 
  if  $v = s$ 
    setDistance(v, 0)
  else
    setDistance(v, infinity)
for  $i \leftarrow 1$  to  $n-1$  do
  for each  $e \in G.edges()$ 
    { relax edge  $e$  }
     $u \leftarrow G.origin(e)$ 
     $z \leftarrow G.opposite(u,e)$ 
     $r \leftarrow getDistance(u) + weight(e)$ 
    if  $r < getDistance(z)$ 
      setDistance(z,r)

```

Bellman-Ford Example

Nodes are labeled with their $d(v)$ values



Outline

Definitions and Problems

Dijkstra's Algorithm

Bellman-Ford Algorithm

Minimal Spanning Tree

Prim-Jarnik Algorithm

Kruskals Algorithm

Minimum Spanning Tree

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

Spanning tree

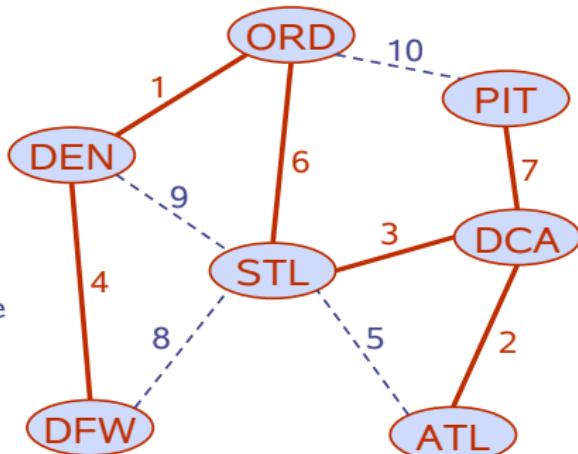
- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)

- Spanning tree of a weighted graph with minimum total edge weight

◆ Applications

- Communications networks
- Transportation networks



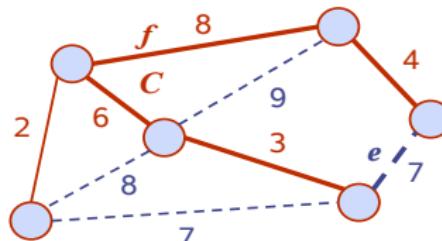
Cycle Property

Cycle Property:

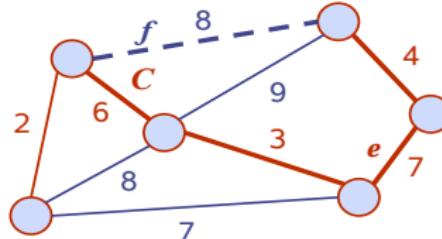
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C ,
 $\text{weight}(f) \leq \text{weight}(e)$

Proof:

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



Replacing f with e yields
a better spanning tree



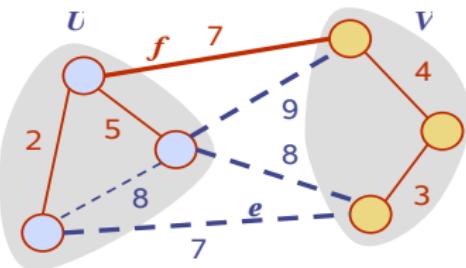
Partition Property

Partition Property:

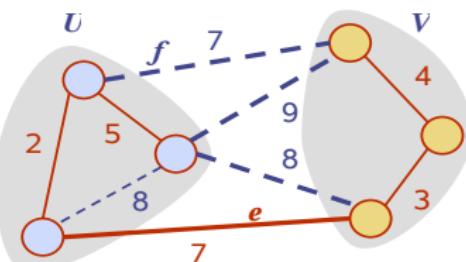
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

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,
 $\text{weight}(f) \leq \text{weight}(e)$
- Thus, $\text{weight}(f) = \text{weight}(e)$
- We obtain another MST by replacing f with e



Replacing f with e yields another MST



Outline

Definitions and Problems

Dijkstra's Algorithm

Bellman-Ford Algorithm

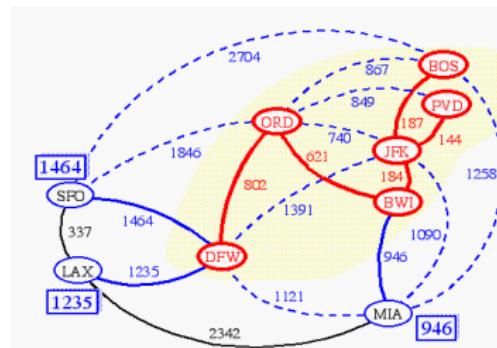
Minimal Spanning Tree

Prim-Jarnik Algorithm

Kruskals Algorithm

Prim-Jarnik's Algorithm

- ◆ Similar to Dijkstra's algorithm (for a connected graph)
- ◆ 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 $d(v)$ = the smallest weight of an edge connecting v to a vertex in the cloud
- ◆ 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



Prim-Jarnik's Algorithm (cont.)

- ◆ A priority queue stores the vertices outside the cloud
 - Key: distance
 - Element: vertex
- ◆ Locator-based methods
 - $\text{insert}(k,e)$ returns a locator
 - $\text{replaceKey}(l,k)$ changes the key of an item
- ◆ We store three labels with each vertex:
 - Distance
 - Parent edge in MST
 - Locator in priority queue

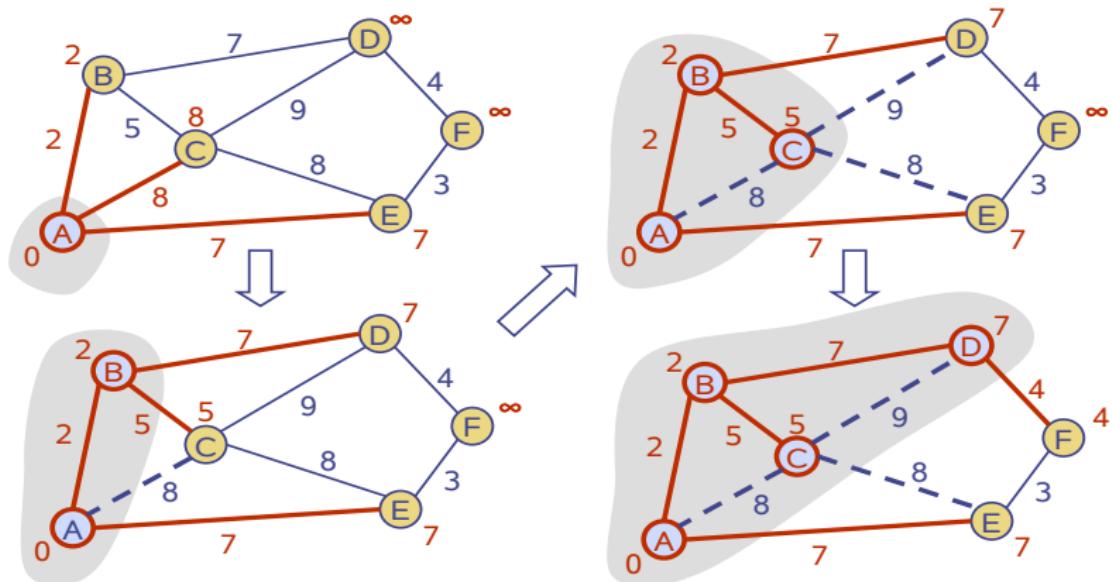
Algorithm *PrimJarnikMST(G)*

```

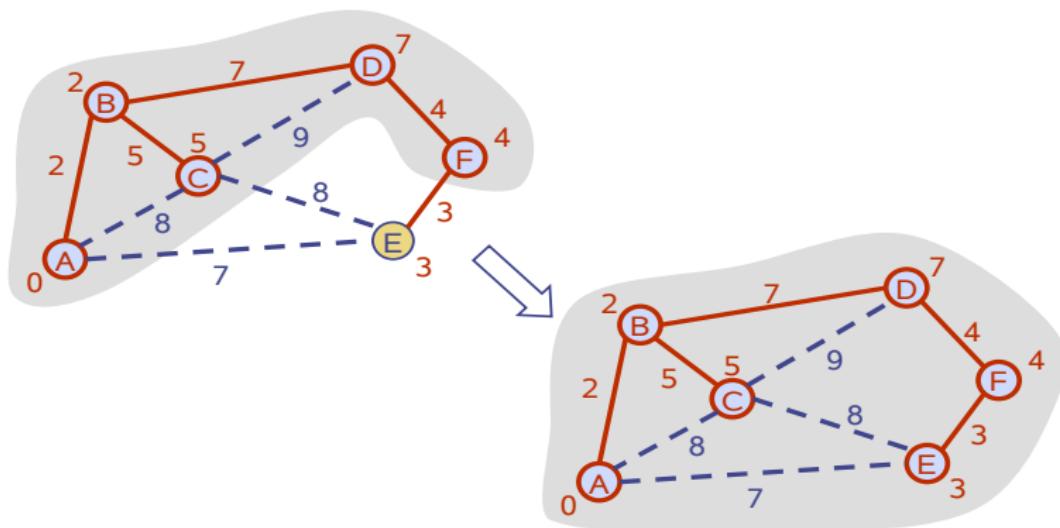
 $Q \leftarrow$  new heap-based priority queue
 $s \leftarrow$  a vertex of  $G$ 
for all  $v \in G.\text{vertices}()$ 
  if  $v = s$ 
     $\text{setDistance}(v, 0)$ 
  else
     $\text{setDistance}(v, \infty)$ 
     $\text{setParent}(v, \emptyset)$ 
     $l \leftarrow Q.\text{insert}(\text{getDistance}(v), v)$ 
     $\text{setLocator}(v,l)$ 
while  $\neg Q.\text{isEmpty}()$ 
   $u \leftarrow Q.\text{removeMin}()$ 
  for all  $e \in G.\text{incidentEdges}(u)$ 
     $z \leftarrow G.\text{opposite}(u,e)$ 
     $r \leftarrow \text{weight}(e)$ 
    if  $r < \text{getDistance}(z)$ 
       $\text{setDistance}(z,r)$ 
       $\text{setParent}(z,e)$ 
       $Q.\text{replaceKey}(\text{getLocator}(z),r)$ 

```

Example



Example (contd.)



Analysis

- ◆ Graph operations
 - Method `incidentEdges` is called once for each vertex
- ◆ Label operations
 - We set/get the distance, parent and locator 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 $\sum_v \deg(v) = 2m$
- ◆ The running time is $O(m \log n)$ since the graph is connected

Outline

Definitions and Problems

Dijkstra's Algorithm

Bellman-Ford Algorithm

Minimal Spanning Tree

Prim-Jarnik Algorithm

Kruskals Algorithm

Kruskal's Algorithm

Algorithm KruskalMST(G)

Input: A digraph G

Output: A tree with the MST

for each vertex v of G define a $Cloud(v) \leftarrow \{v\}$

let Q be a priority queue

insert all edges into Q using their weights as key

$T \leftarrow \emptyset$

while T has fewer than $n - 1$ edges do

 edge $e = T.\text{removeMin}()$

 let u, v be the endpoints of e

 if $Cloud(u) \neq Cloud(v)$ then

 add edge e to T

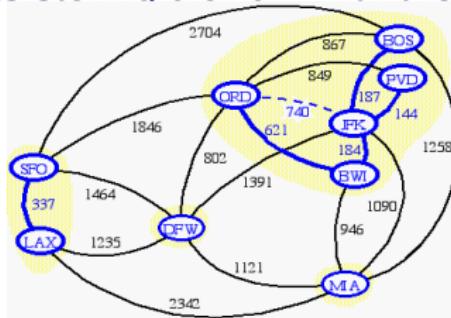
 merge $Cloud(u)$ and $Cloud(v)$

return T

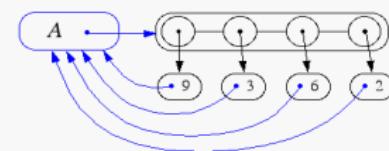


Data Structure for Kruskal Algorithm

- ◆ The algorithm maintains a forest of trees
- ◆ An edge is accepted if it 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**

$(u,v) \leftarrow Q.\text{removeMinElement}()$

if $P.\text{find}(u) \neq P.\text{find}(v)$ **then**

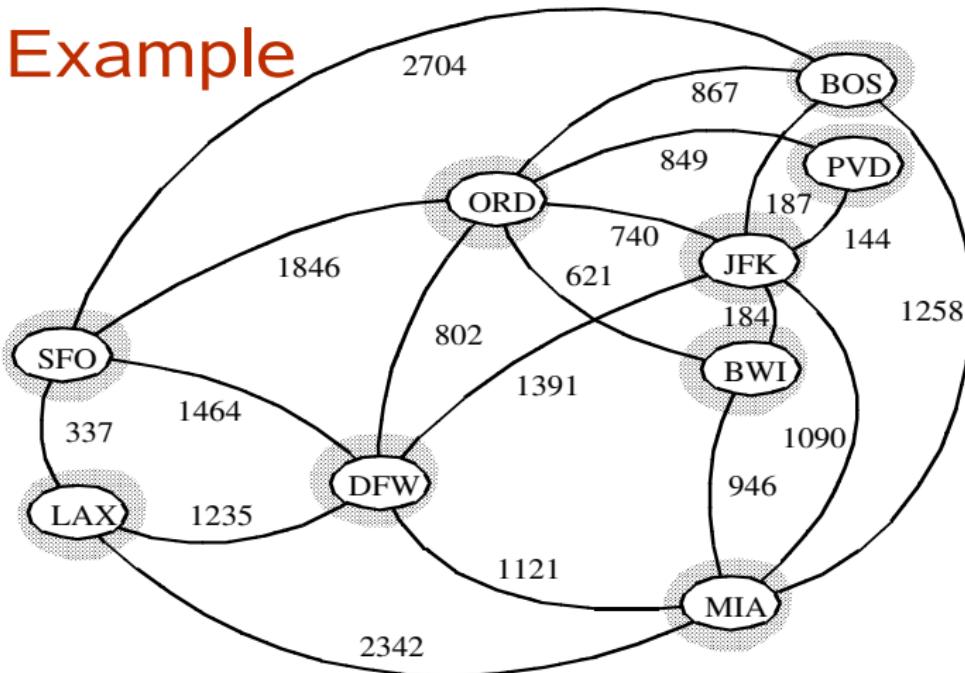
 Add (u,v) to T

$P.\text{union}(u,v)$

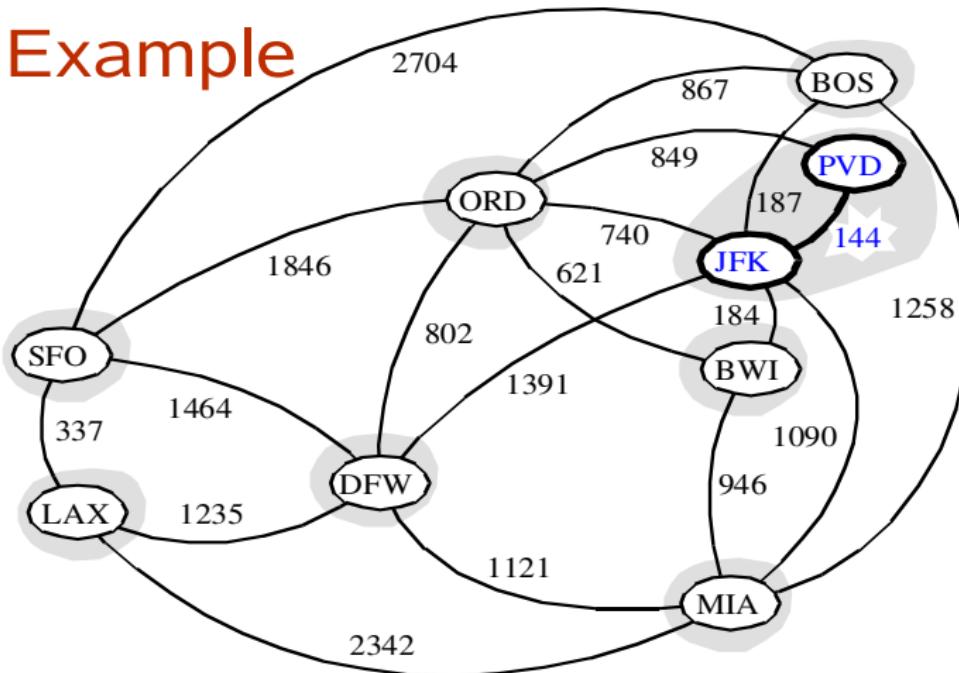
return T

Running time:
 $O((n+m)\log n)$

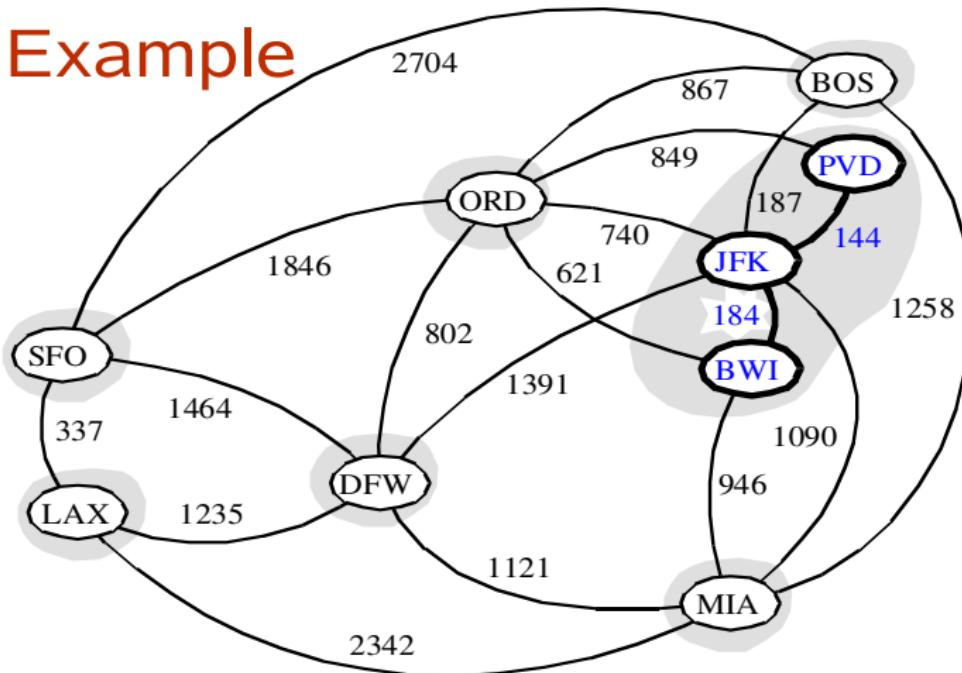
Kruskal Example



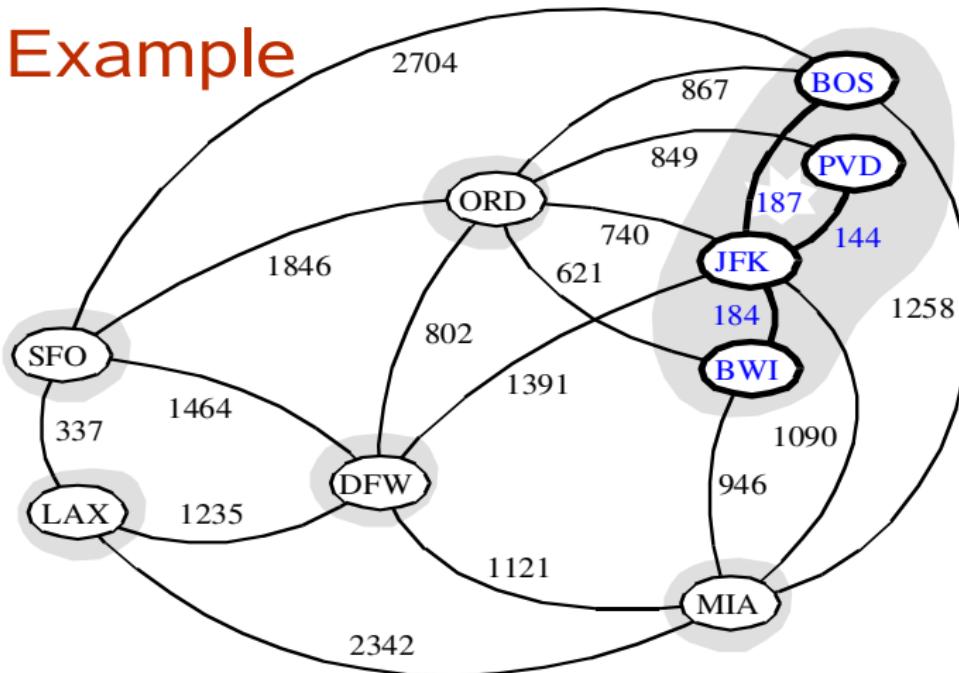
Example



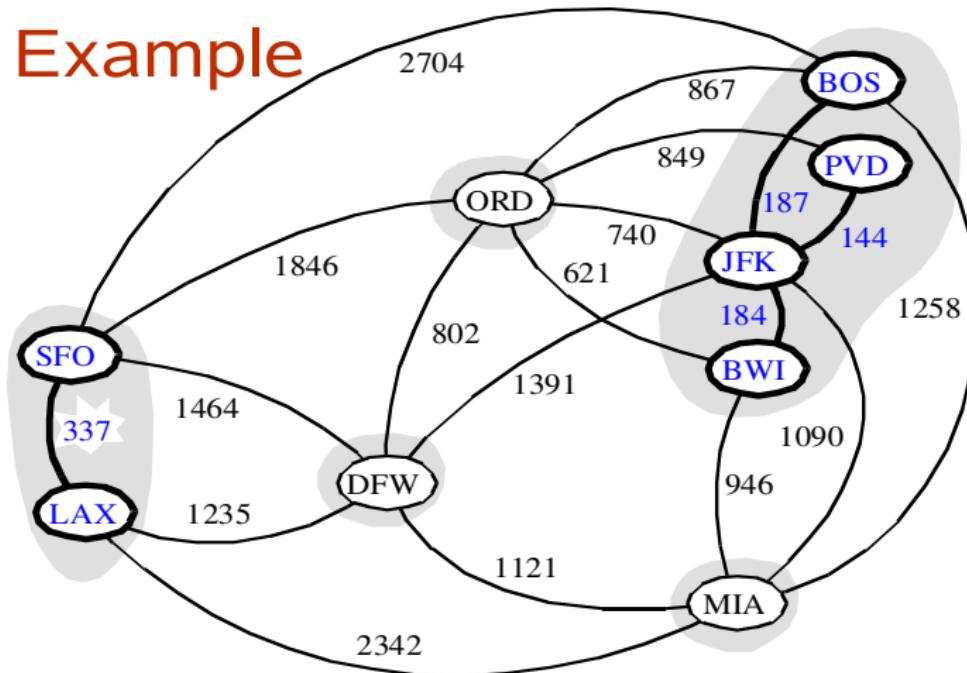
Example



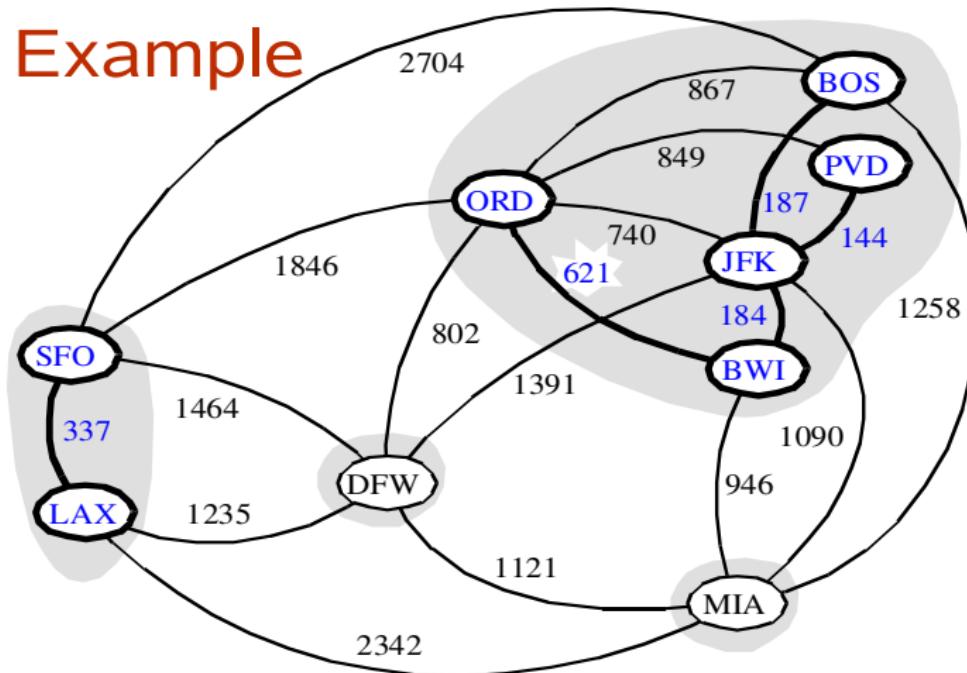
Example



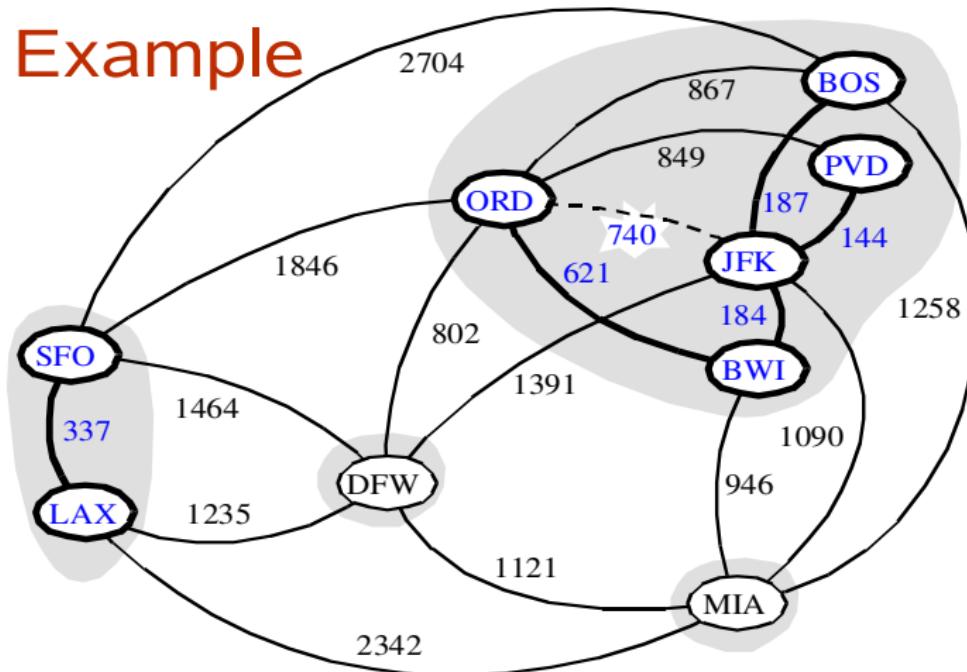
Example



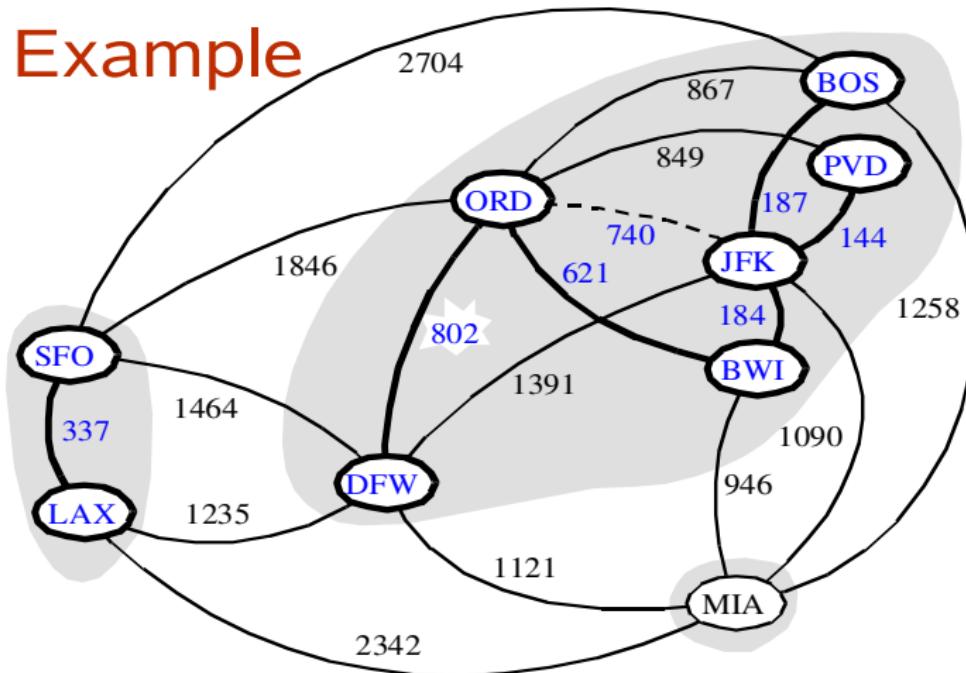
Example



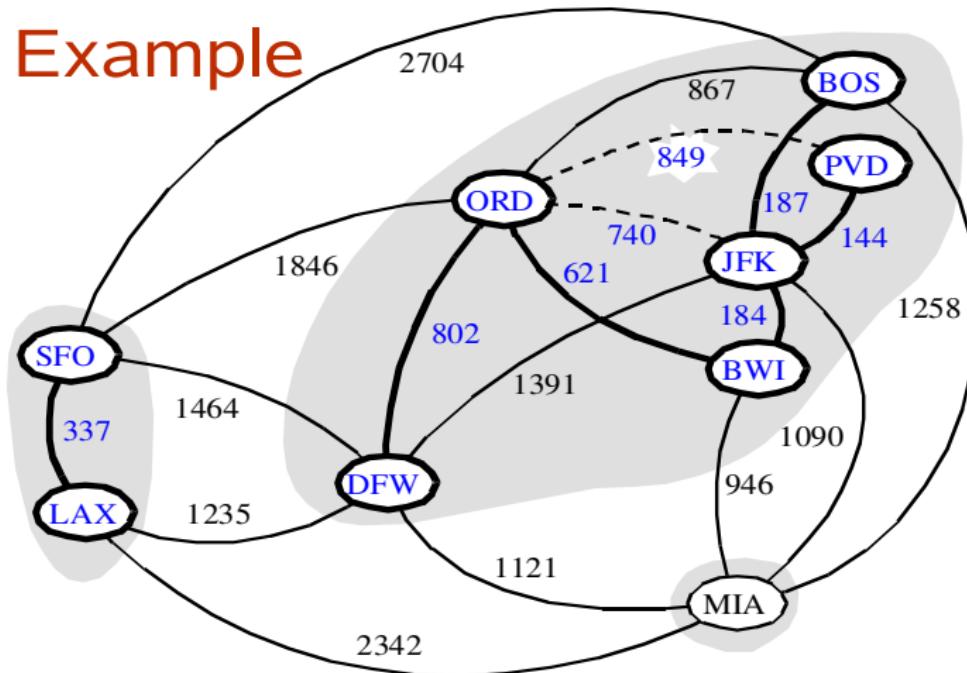
Example



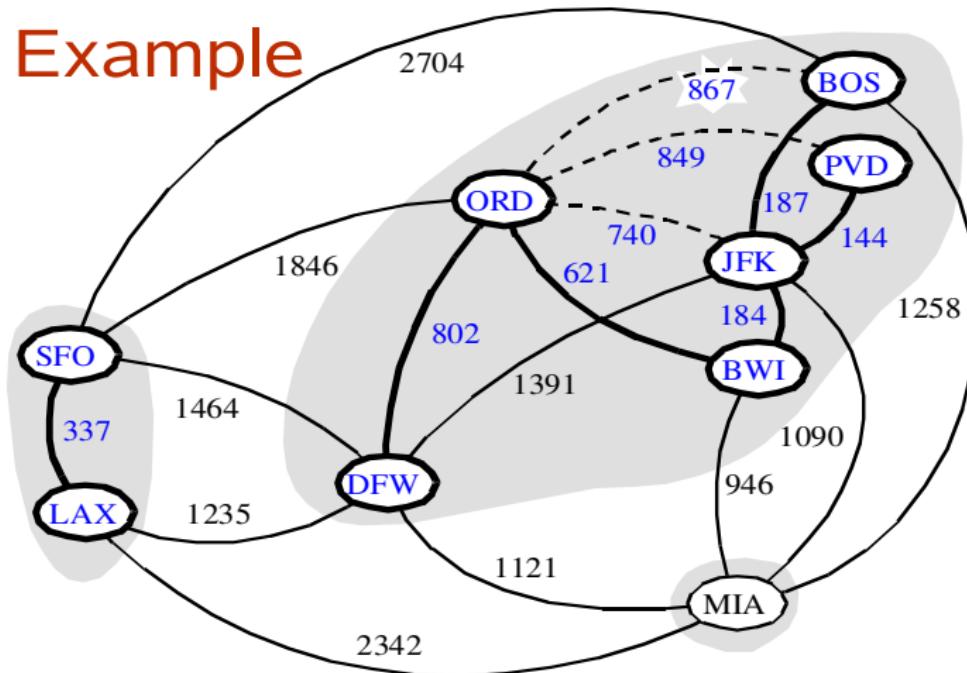
Example



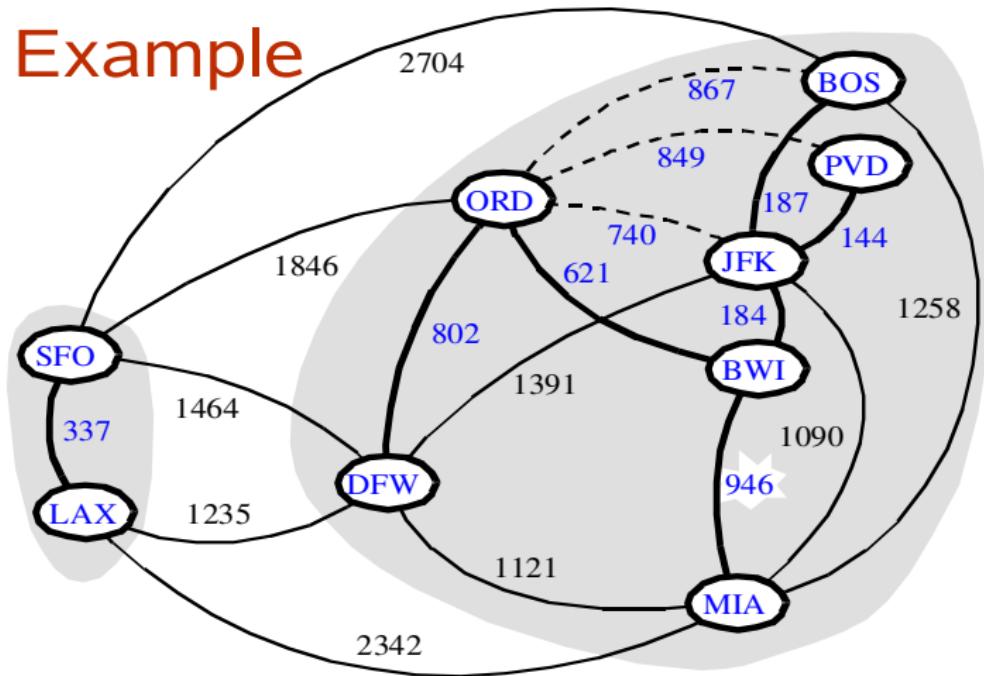
Example



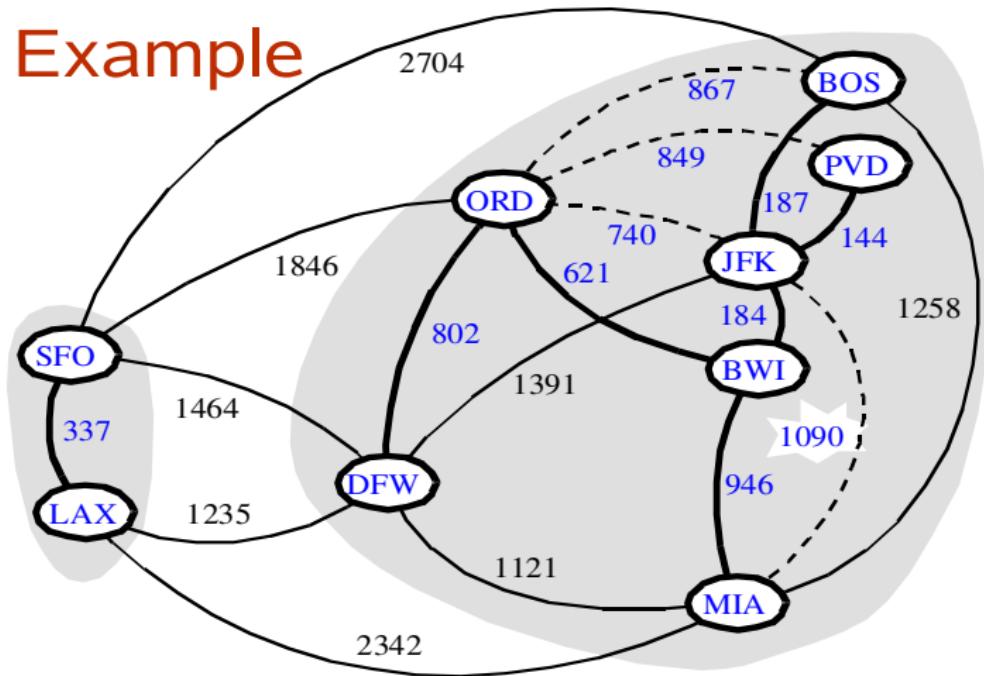
Example



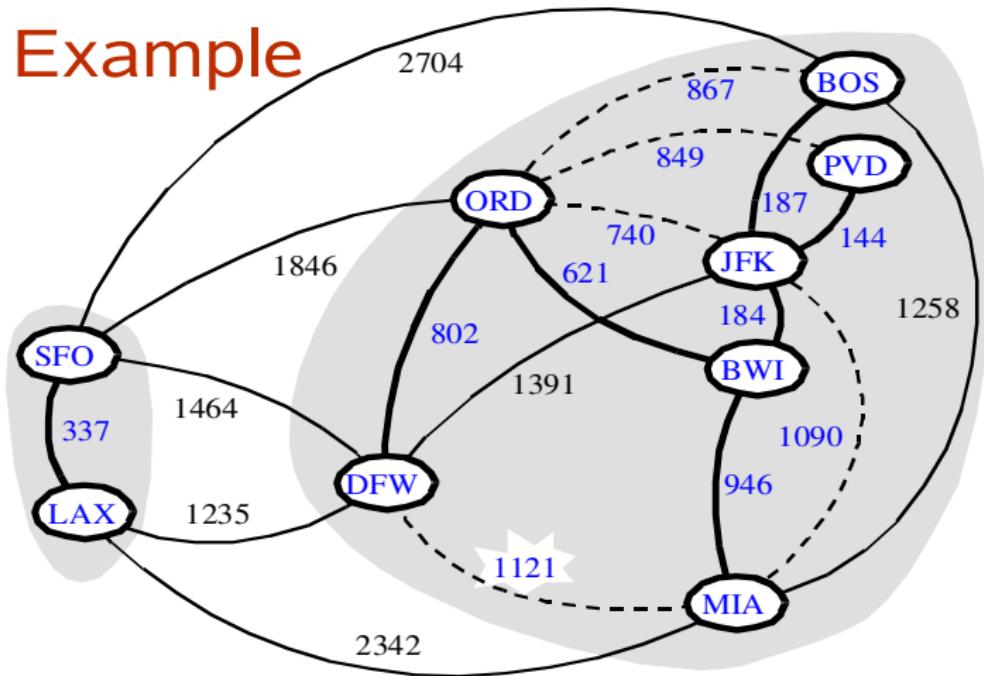
Example



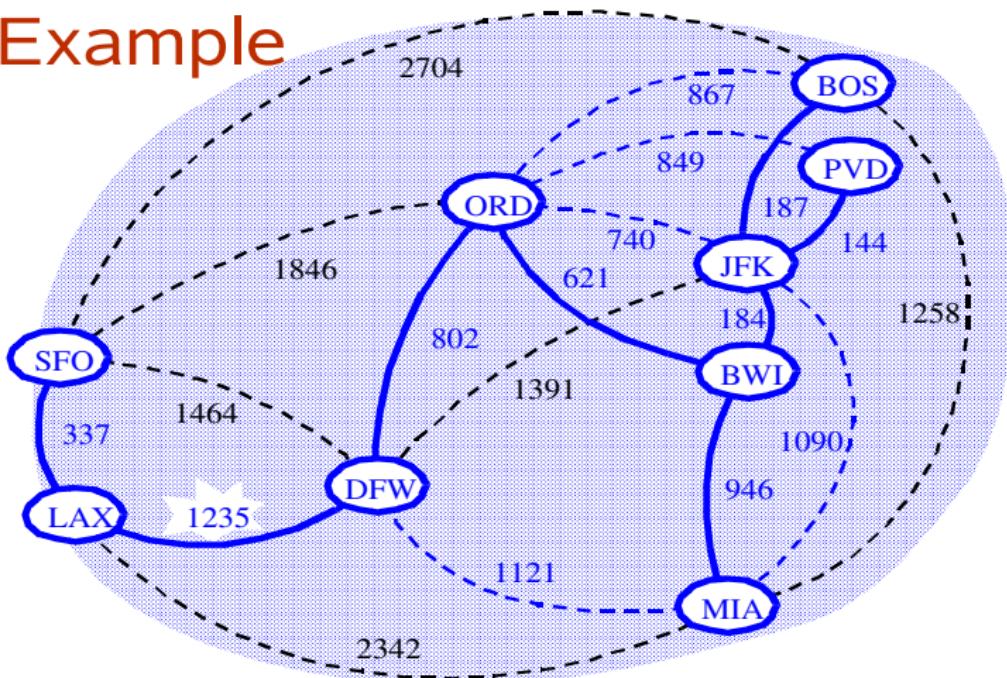
Example



Example



Example



Baruvka's Algorithm

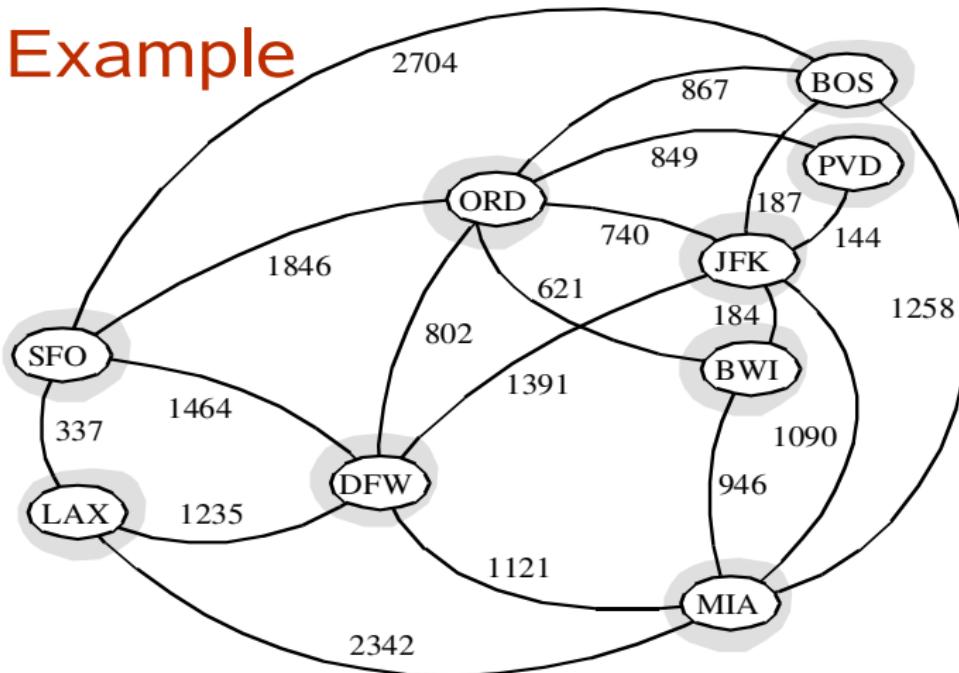
- Like Kruskal's Algorithm, Baruvka's algorithm grows many "clouds" at once.

Algorithm *BaruvkaMST(G)*

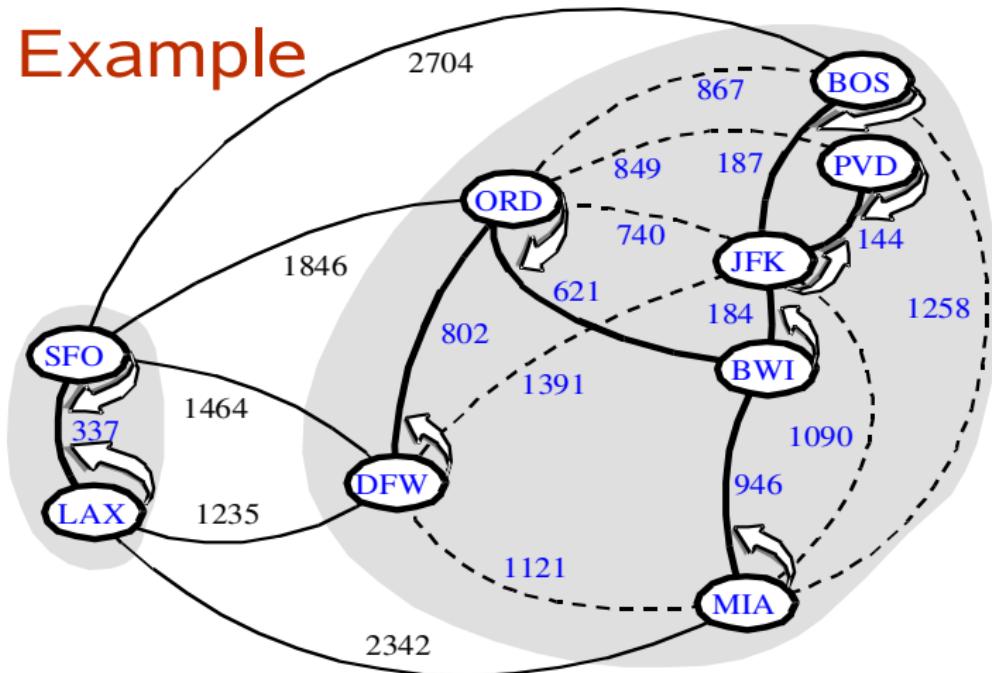
```
 $T \leftarrow V$  {just the vertices of  $G$ }  
while  $T$  has fewer than  $n-1$  edges do  
  for each connected component  $C$  in  $T$  do  
    Let edge  $e$  be the smallest-weight edge from  $C$  to another component in  $T$ .  
    if  $e$  is not already in  $T$  then  
      Add edge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

