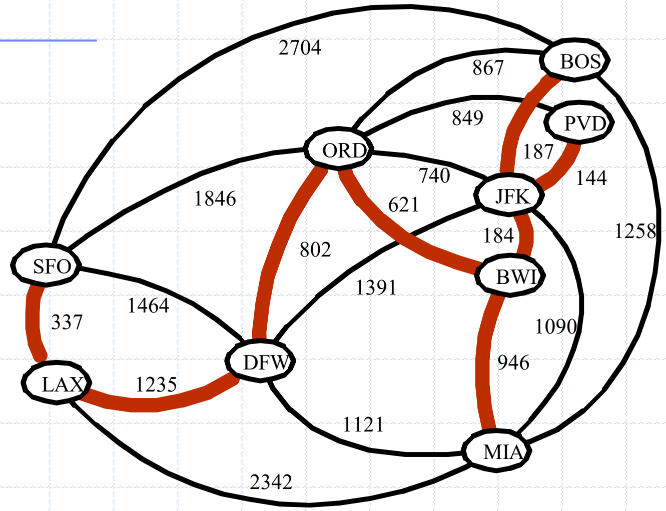
Lecture 14: Minimum Spanning Trees

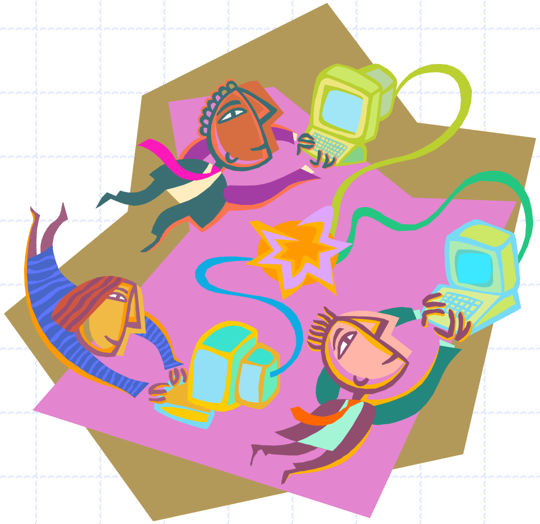
Infinite Correlation



Wholeness Statement

A minimum spanning tree is a spanning tree subgraph with minimum total edge weight. Efficient greedy algorithms have been developed to compute MST both with and without special data structures. Pure creative intelligence is the source of all creative algorithms. Regular practice of TM improves our ability to make use of our own innate creative potential.

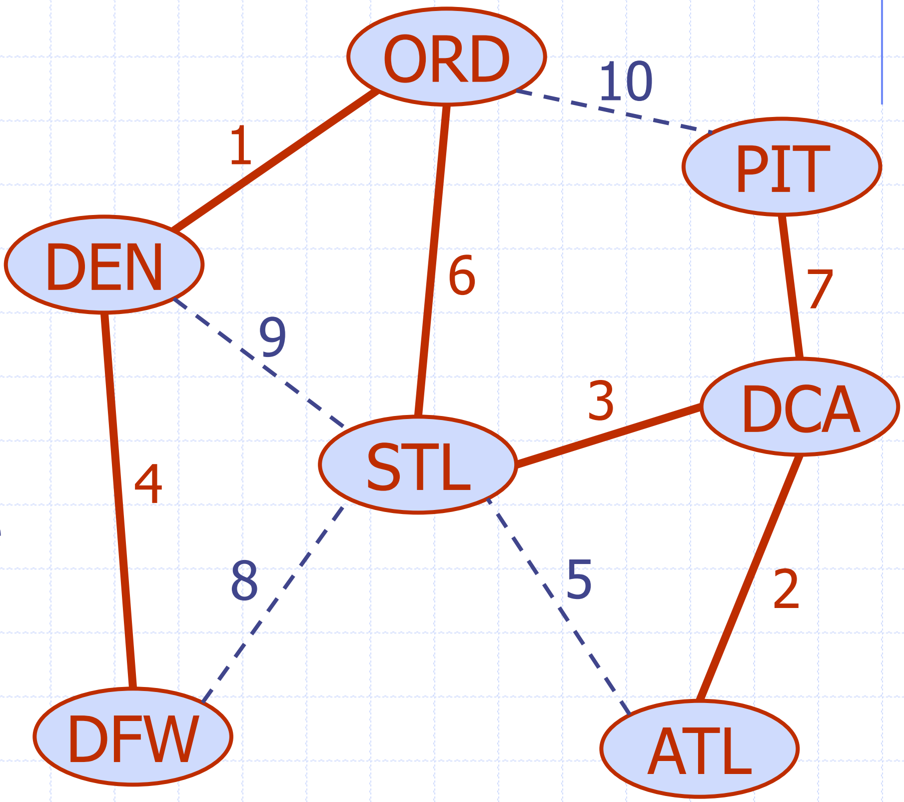
Outline and Reading



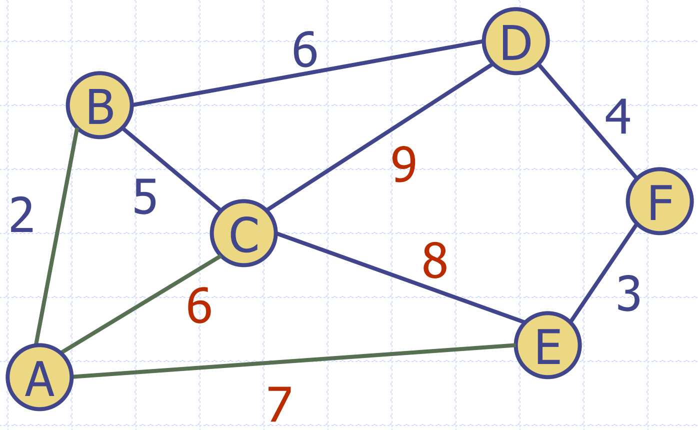
- Minimum Spanning Trees (§7.3)
  - Definitions
  - Cycle Property
  - Partition Property
- The Prim-Jarnik Algorithm (1957, 1930) (§7.3.2)
- Kruskal's Algorithm (1956) (§7.3.1)
- Baruvka's Algorithm (1926) (§7.3.3)

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

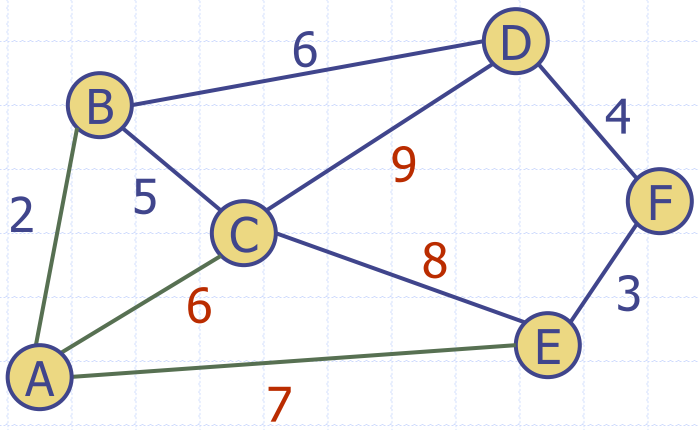


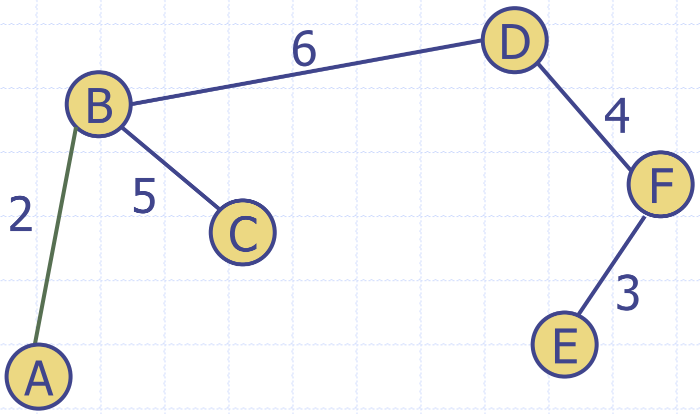
Cycle Property



If the weight of an edge ***e*** of a cycle C is larger than the weights of other edges of C, then this edge cannot belong to a MST.

Cycle Property





If the weight of an edge ***e*** of a cycle C is larger than the weights of other edges of C, then this edge cannot belong to a MST.

Cycle Property

- **Cycle Property**: For any cycle C in a graph, if the weight of an edge ***e*** of C is larger than the weights of other edges of C, then this edge cannot belong to a MST.

- **Proof**: Assume the contrary, i.e. that e belongs to an MST ***T1***; then deleting ***e*** will break ***T1*** into two subtrees with the two ends of ***e*** in different subtrees. The remainder of C reconnects the subtrees, hence there is an edge ***f*** of C with ends in different subtrees, i.e., it reconnects the subtrees into a tree ***T2*** with weight less than that of ***T1***, because the weight of ***f*** is less than the weight of ***e***; thus ***T1*** cannot be a MST.

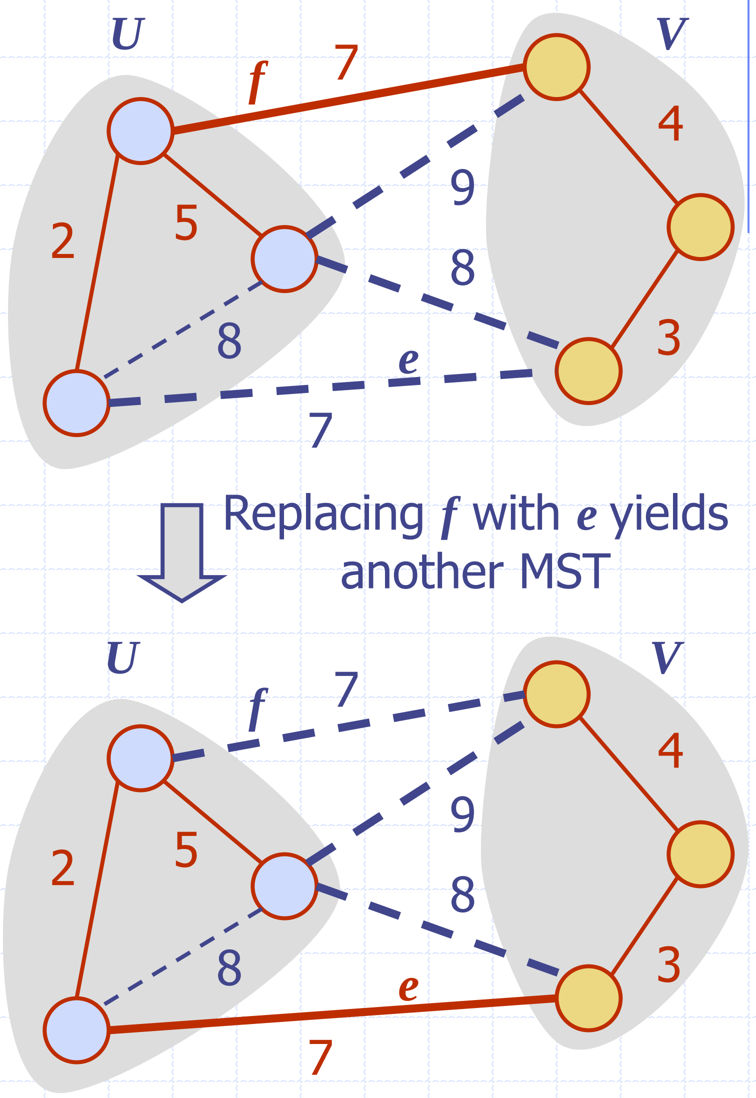
Partition Property-
A Crucial Fact

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



Generic MST Algorithm

Algorithm ***GenericMST(G)***

$T \leftarrow$ a tree with all vertices in G, but no edges

while ***T does not form a spanning tree*** **do**

$(u, v) \leftarrow$ a safe edge of G

$T \leftarrow (u, v) \cup T$

return T

A *safe edge* is one that when added to T forms a subgraph of a MST

Main Point

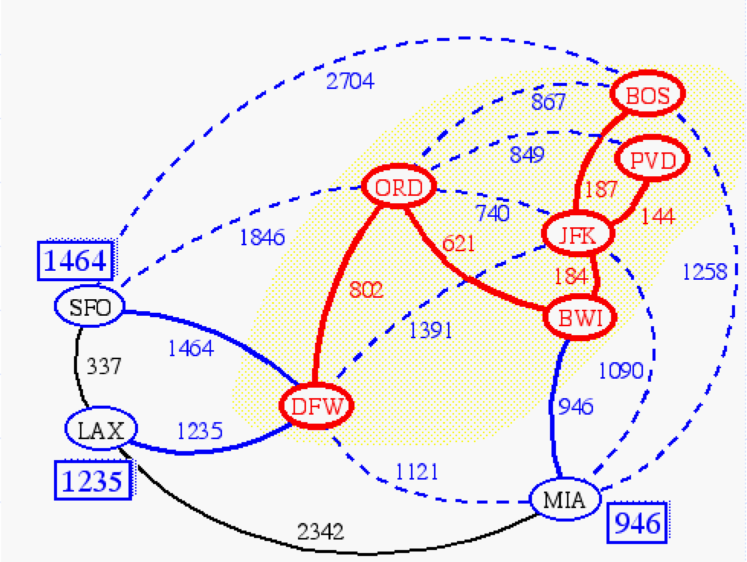
1. A minimum spanning tree algorithm gradually grows a (sub-solution) tree by adding a “safe edge” that connects a vertex in the tree to a vertex not yet in the tree.
   Science of Consciousness: The nature of life is to grow and progress to the state of enlightenment, fulfillment.

Prim(1957)-Jarnik(1930) Algorithm

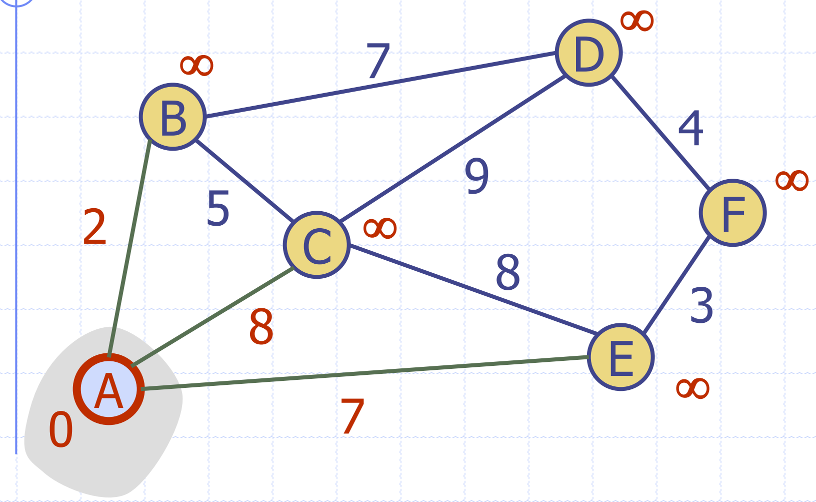
AKA Dijkstra-Prim Algorithm

Prim-Jarnik's Algorithm

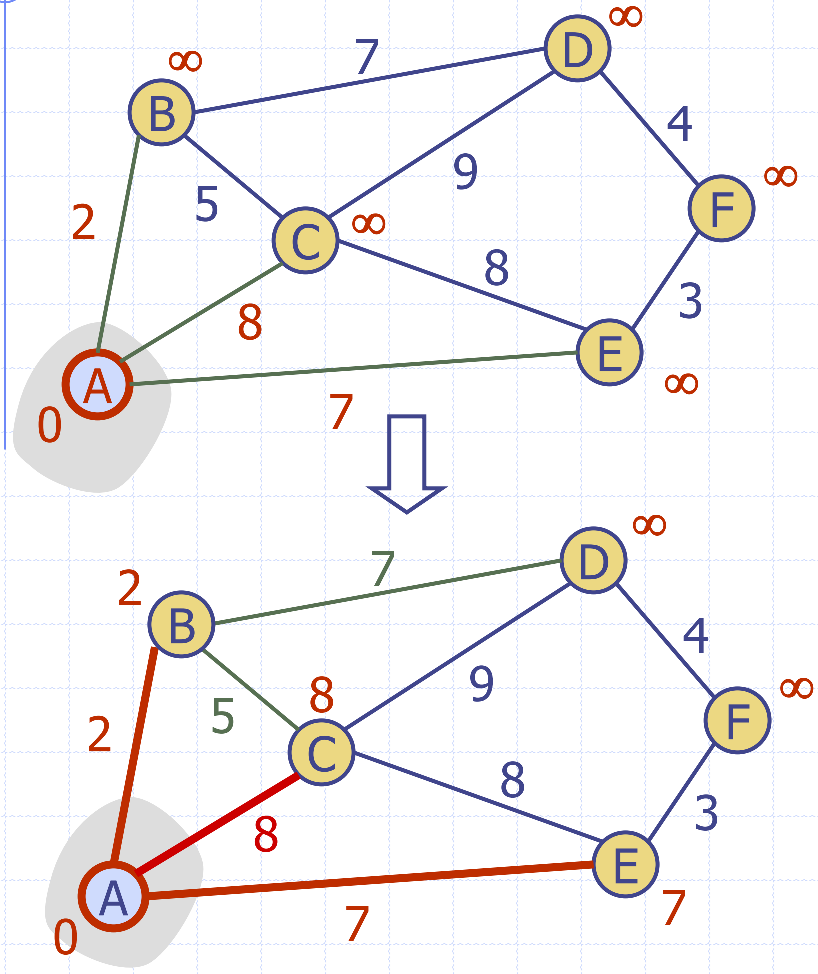
- Similar to Dijkstra's shortest path algorithm (for a connected graph)
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v a label $d(v)$ = the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We 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



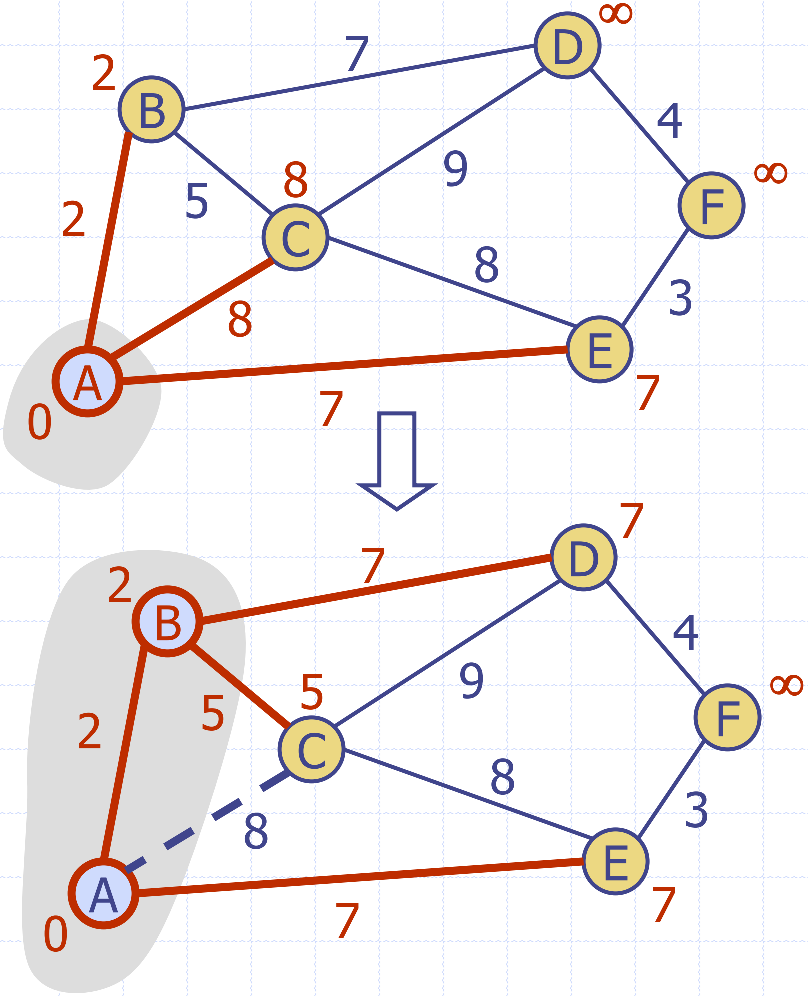
Example



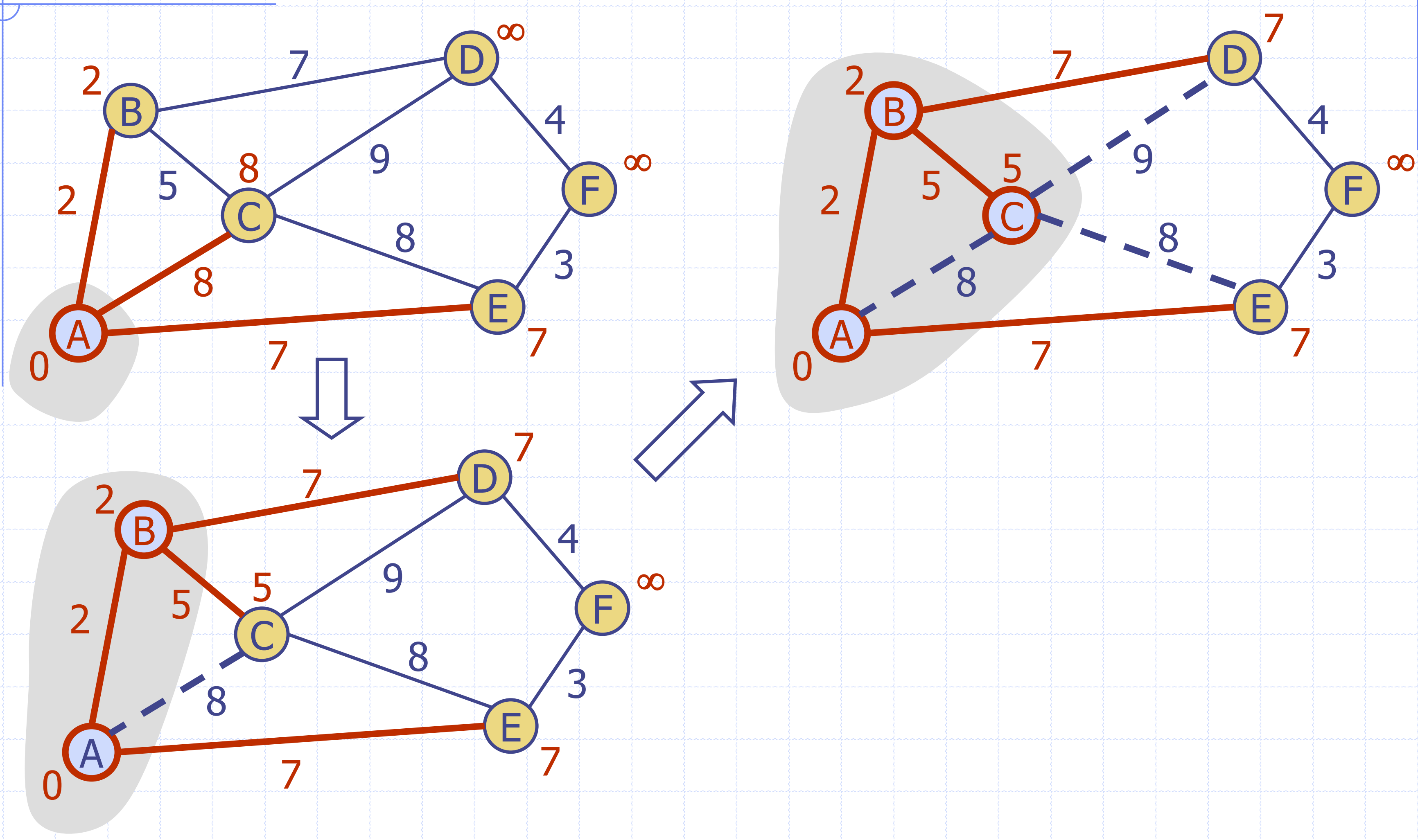
Example



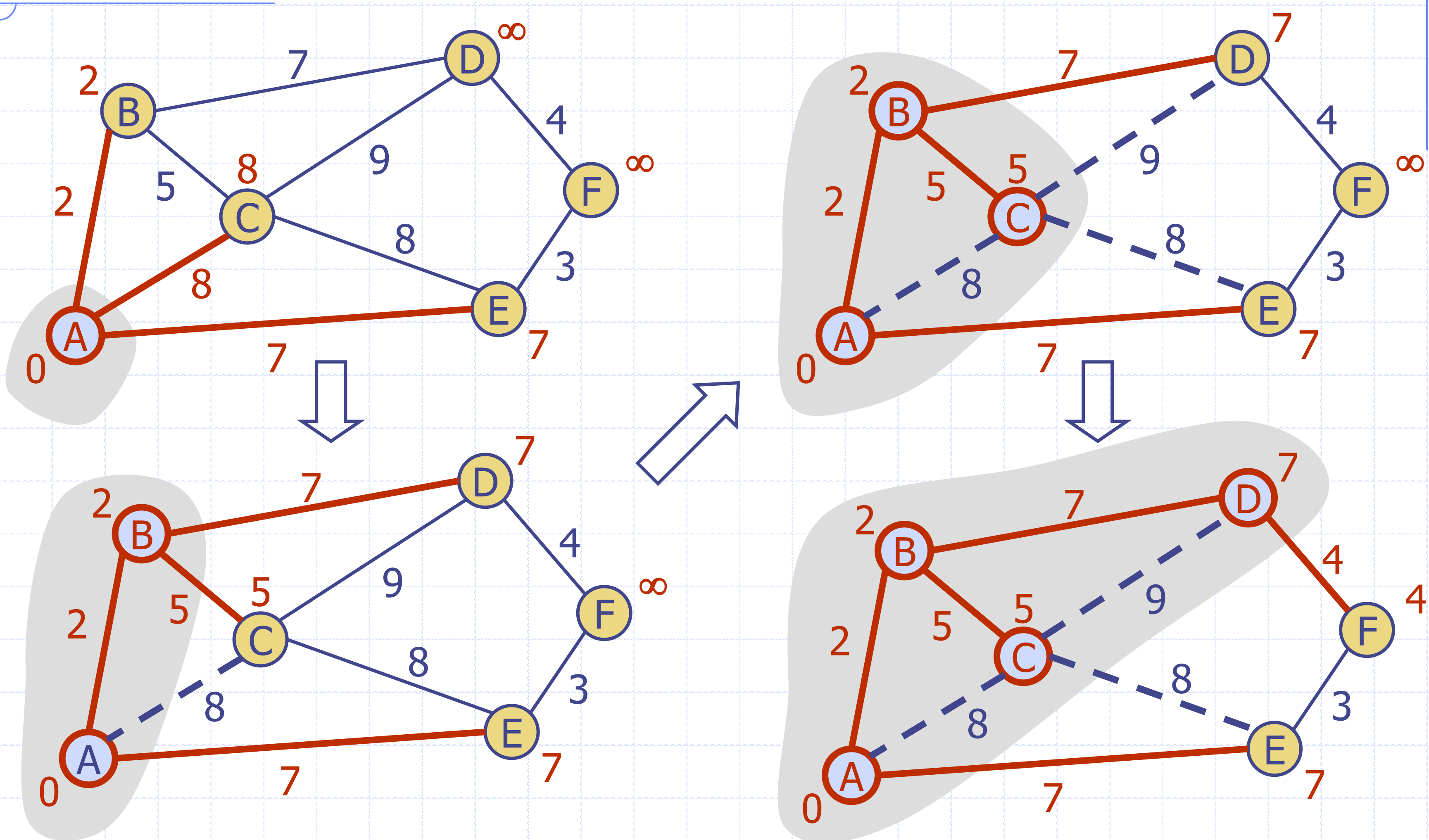
Example



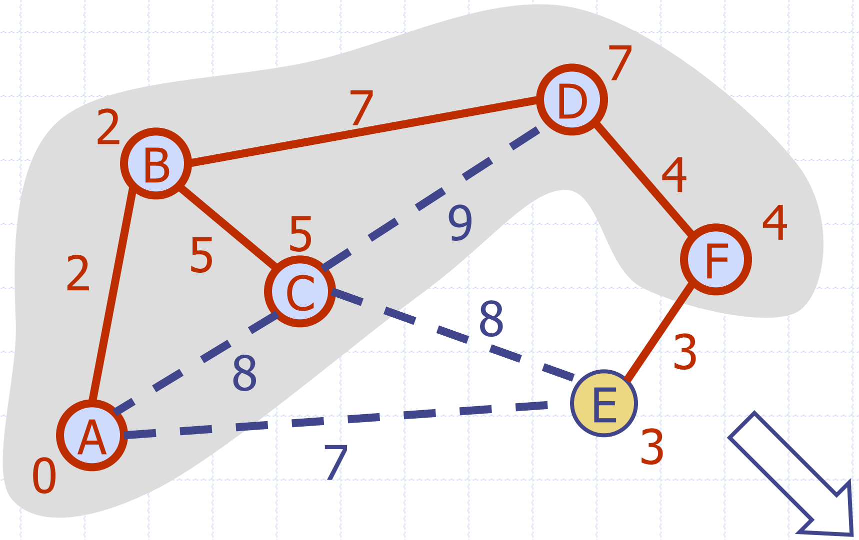
Example

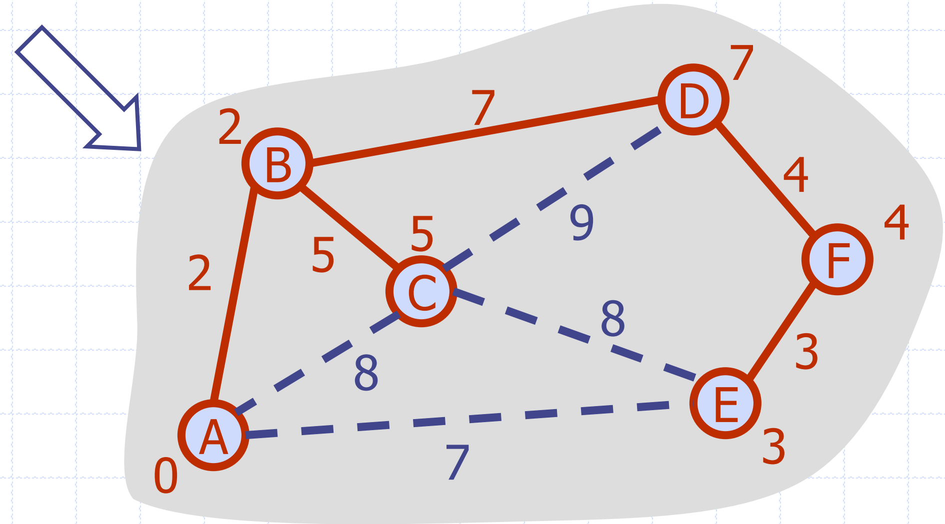


Example



Example (contd.)





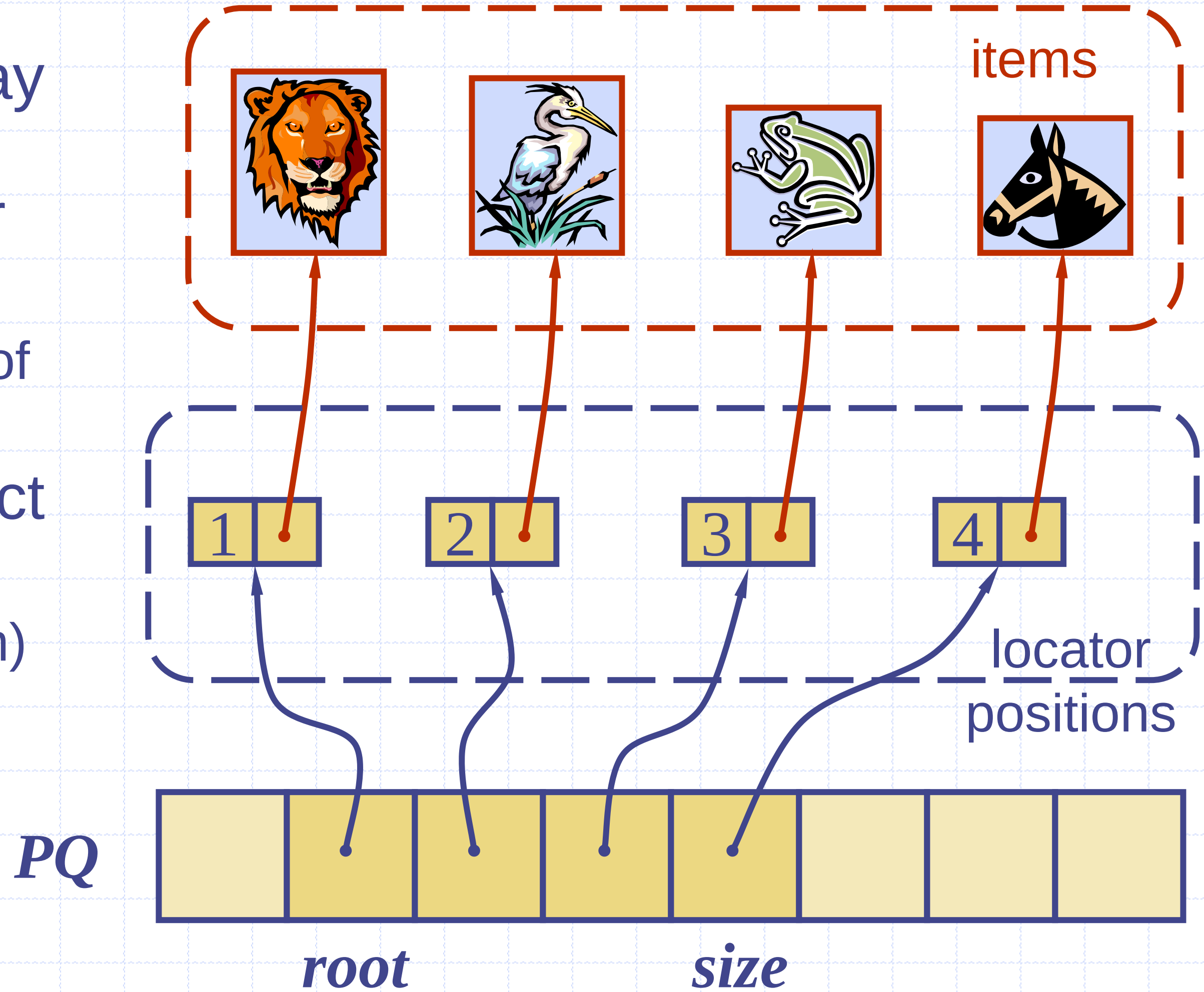
Prim-Jarnik's Algorithm (cont.)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(*k*, *e*) returns a locator position
  - *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
- Correction in red inspired by Bereket Chalew (May 2014)

```
Algorithm PrimJarnikMST(G)
   s ← G.aVertex ()
   Q ← new heap-based priority queue
   for all v ∈ G.vertices() do
     if v = s then
          setDistance(v, 0)
     else
          setDistance(v, ∞)
      setParent(v, ∅)
      l ← Q.insertItem(getDistance(v), v)
      setLocator(v, l)
   while ! Q.isEmpty() do
      u ← Q.removeMin()
      setLocator(u, ∅)   {u is now in MST}
      for all e ∈ G.incidentEdges(u) do
         z ← G.opposite(u, e)
         r ← weight(e)   {diff. from ShortestPath}
         if getLocator(z) ≠ ∅  {z not yet in MST}
               ∧ r < getDistance(z) then
            setDistance(z, r)
            setParent(z, e)
            Q.replaceKey(getLocator(z), r)
```

Array-based Implementation

- We use an array storing locator-positions in our Priority Queue
  - Another level of indirection
- A position object stores:
  - Item (key,elem)
  - Index



Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
  - Recall that $\Sigma_v \deg(v) = 2m$
- 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
- The running time is $O(m \log n)$ since the graph is connected

Main Point

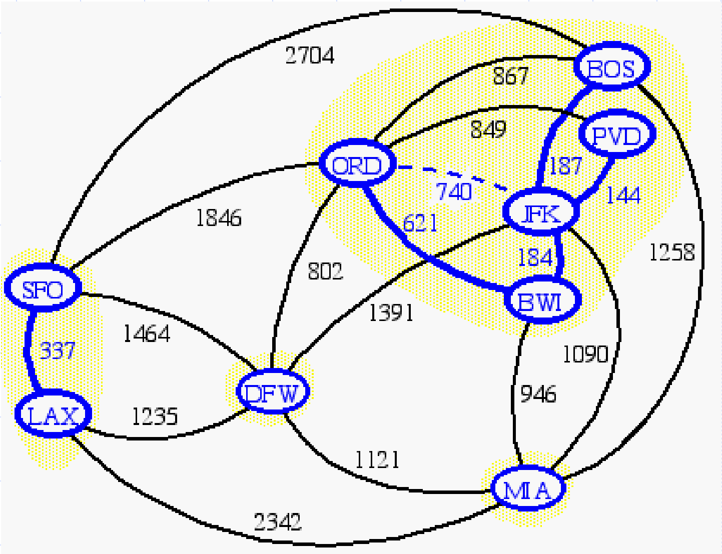
2. A defining feature of the Minimum Spanning Tree (and shortest path) greedy algorithms is that once a vertex becomes in-tree (or “inside the cloud”), the resulting subtree is optimal and nothing can change this state.
   Science of Consciousness: A defining feature of enlightenment is that once this state is reached, one’s consciousness is optimal and nothing can change this state.

Kruskal's Algorithm

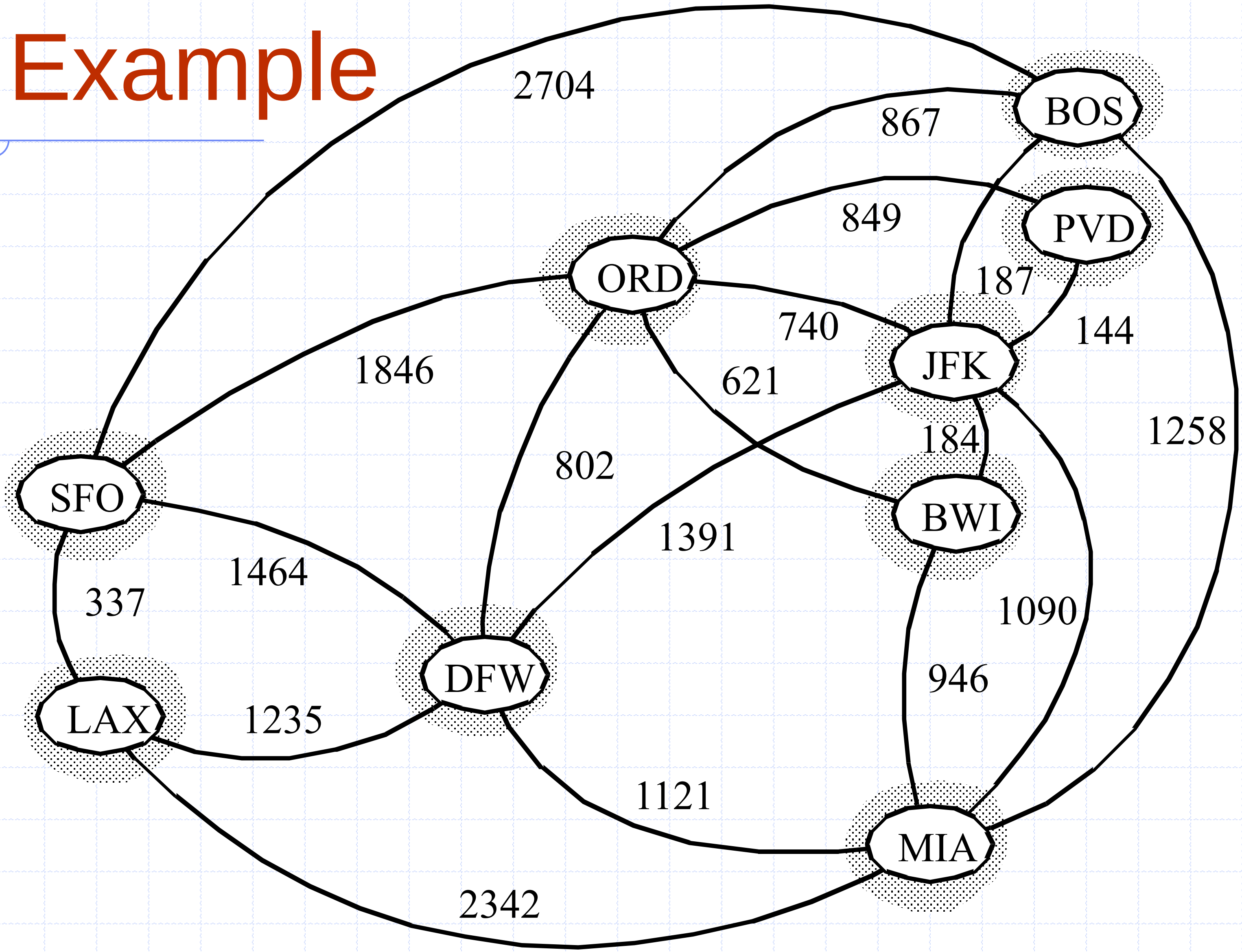
Based on the Partition Property

Basic Idea of the Kruskal Algorithm

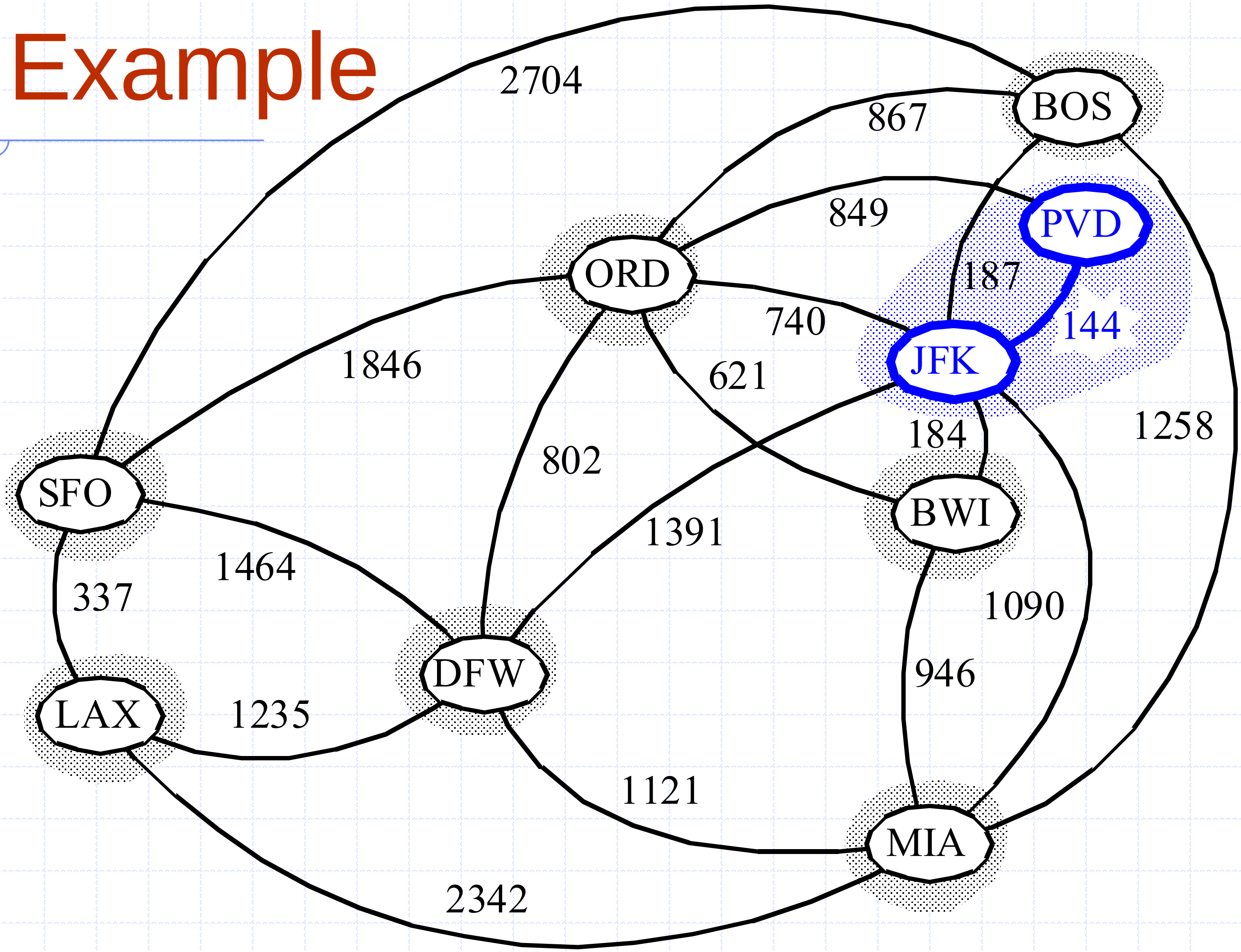
- The algorithm maintains a forest of trees
- Select the edge with the smallest weight
  - The edge is accepted if it connects distinct trees
- Keep adding edges until the tree has n-1 edges



Kruskal Example

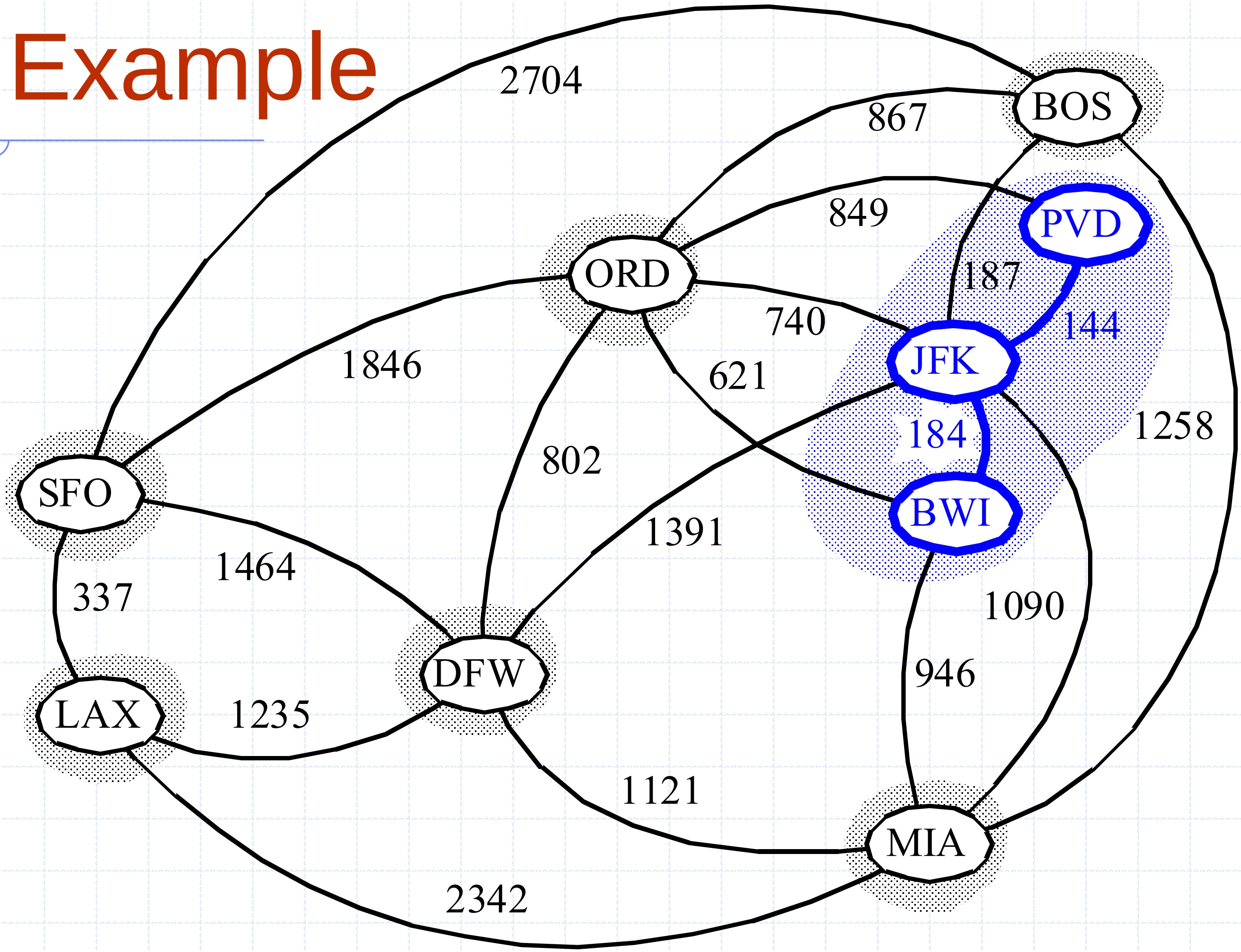


Example



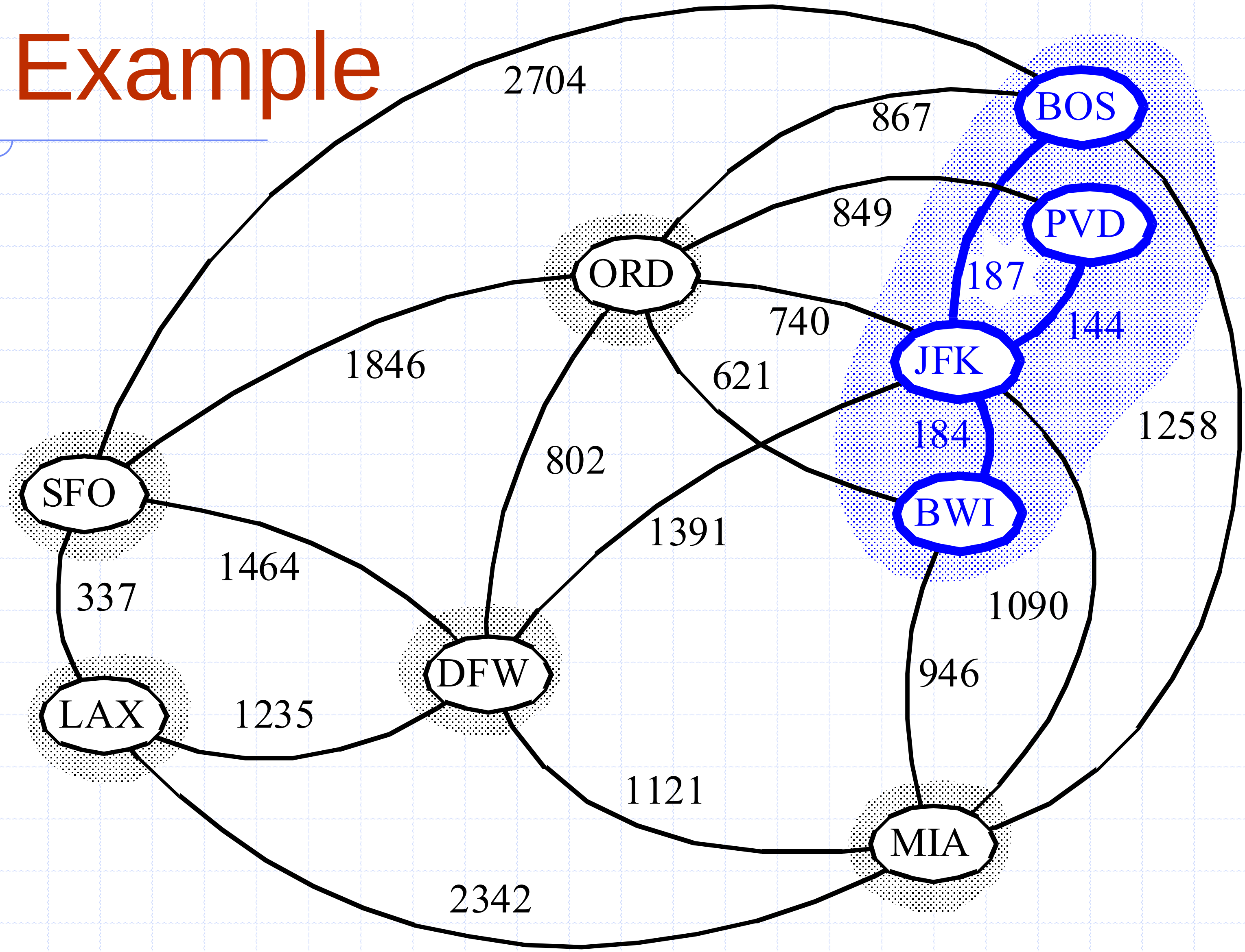
2704
BOS
867
849
PVD
ORD
187
740
144
1846
621
JFK
SFO
1258
184
802
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example

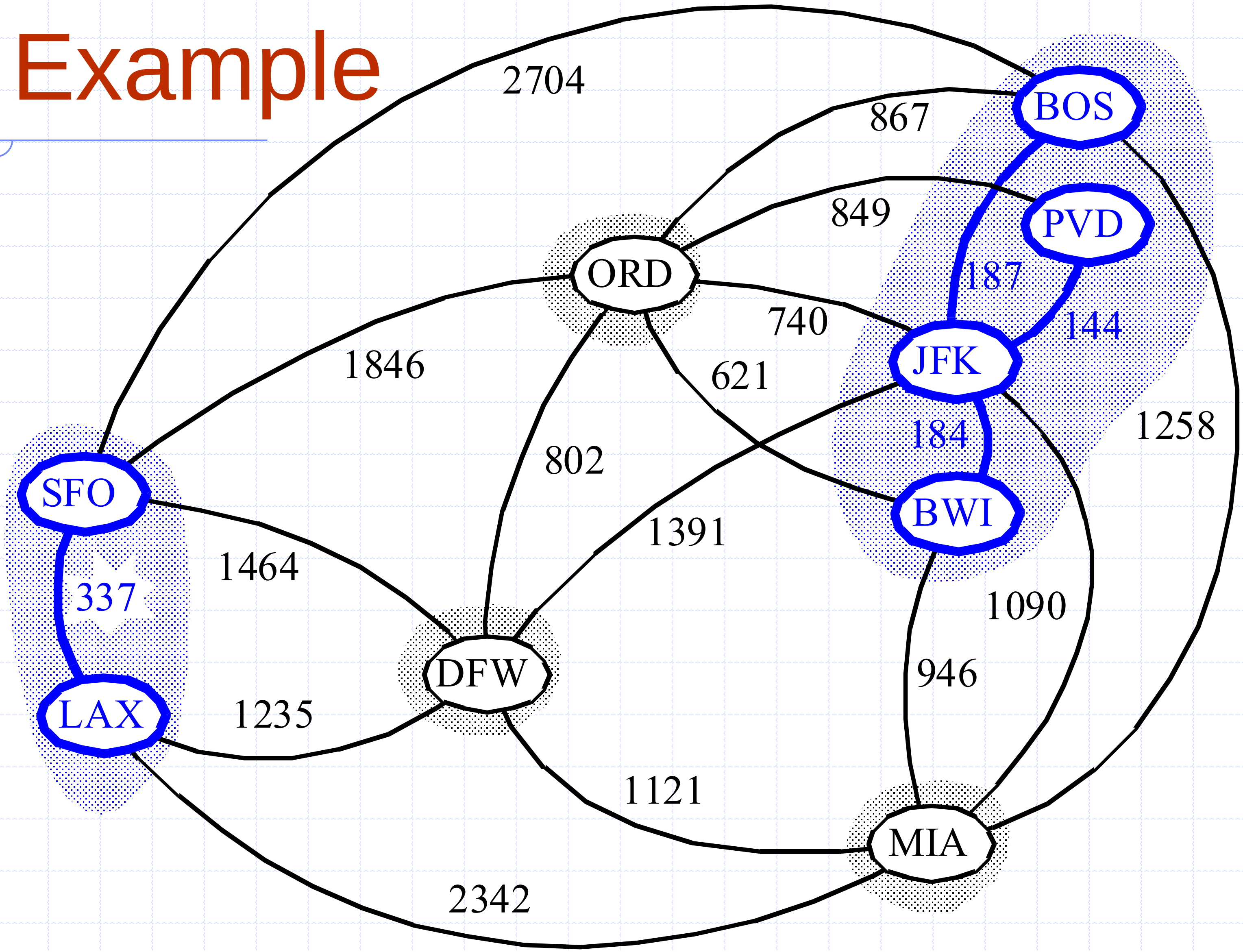


2704
BOS
867
849
PVD
ORD
187
740
144
JFK
1846
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example

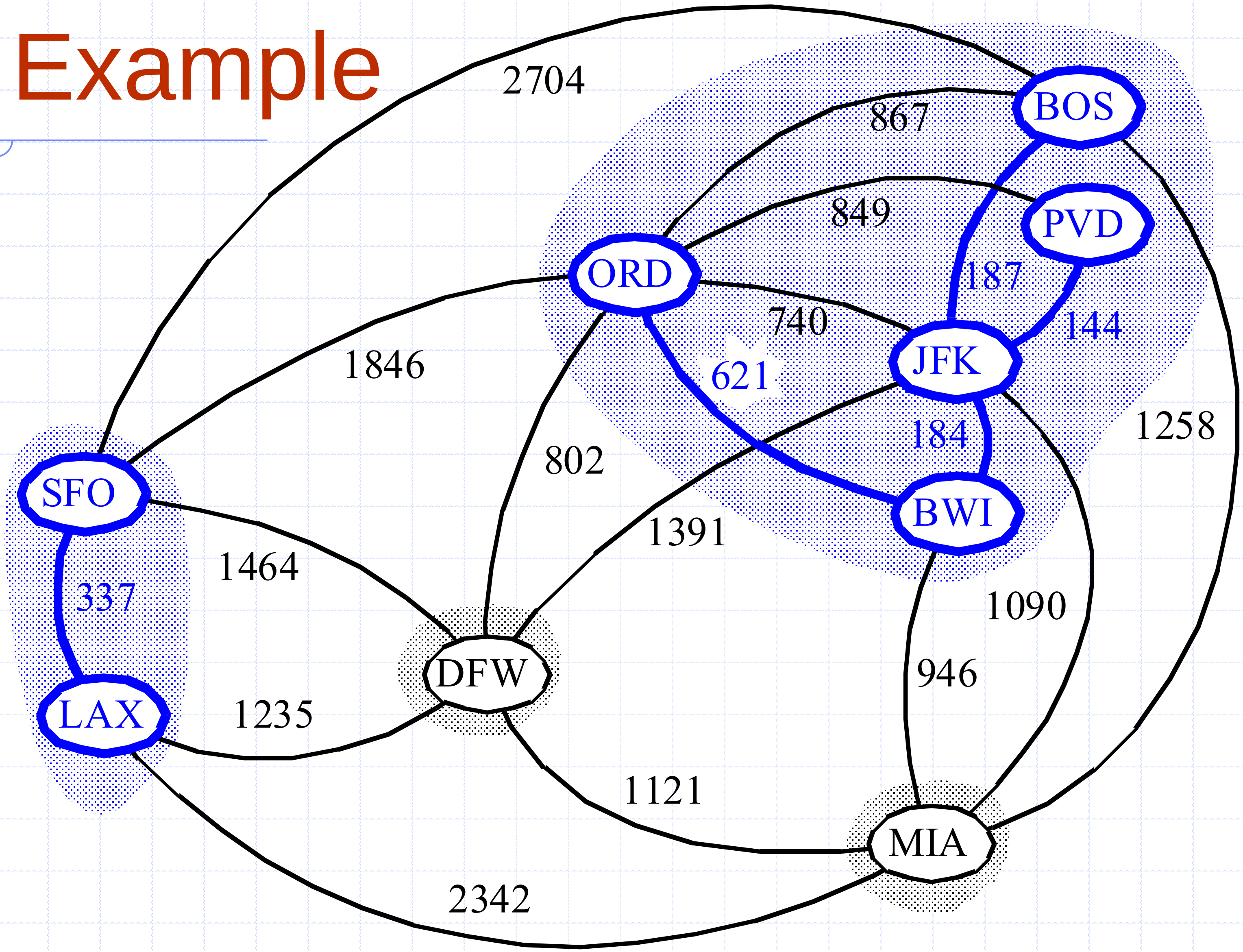

BOS
PVD
ORD
JFK
BWI
SFO
LAX
DFW
MIA
2704
867
849
187
144
740
621
1846
802
184
1258
1391
1464
337
1090
946
1235
1121
2342

Example



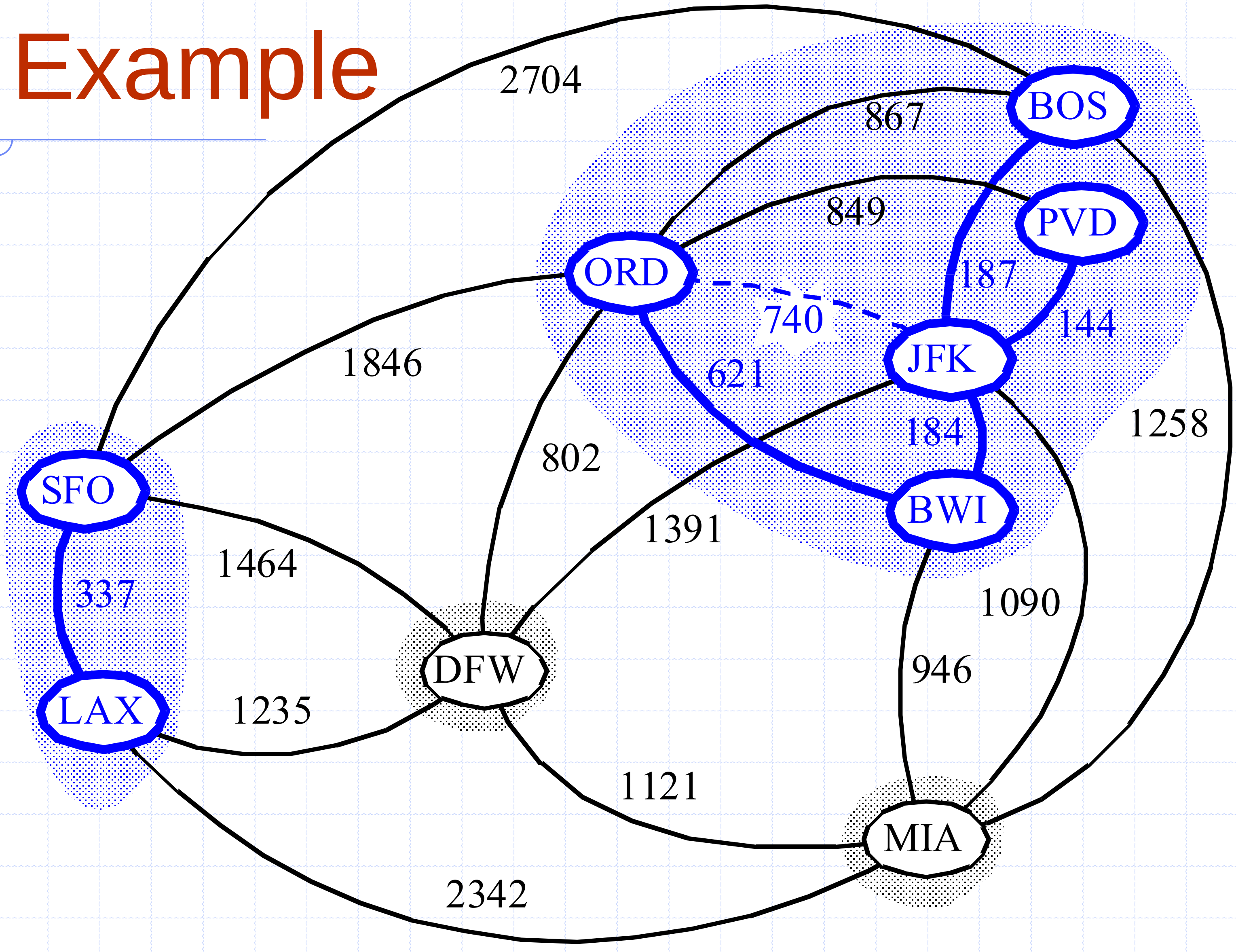
2704
867
849
BOS
PVD
ORD
187
144
740
1846
621
JFK
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



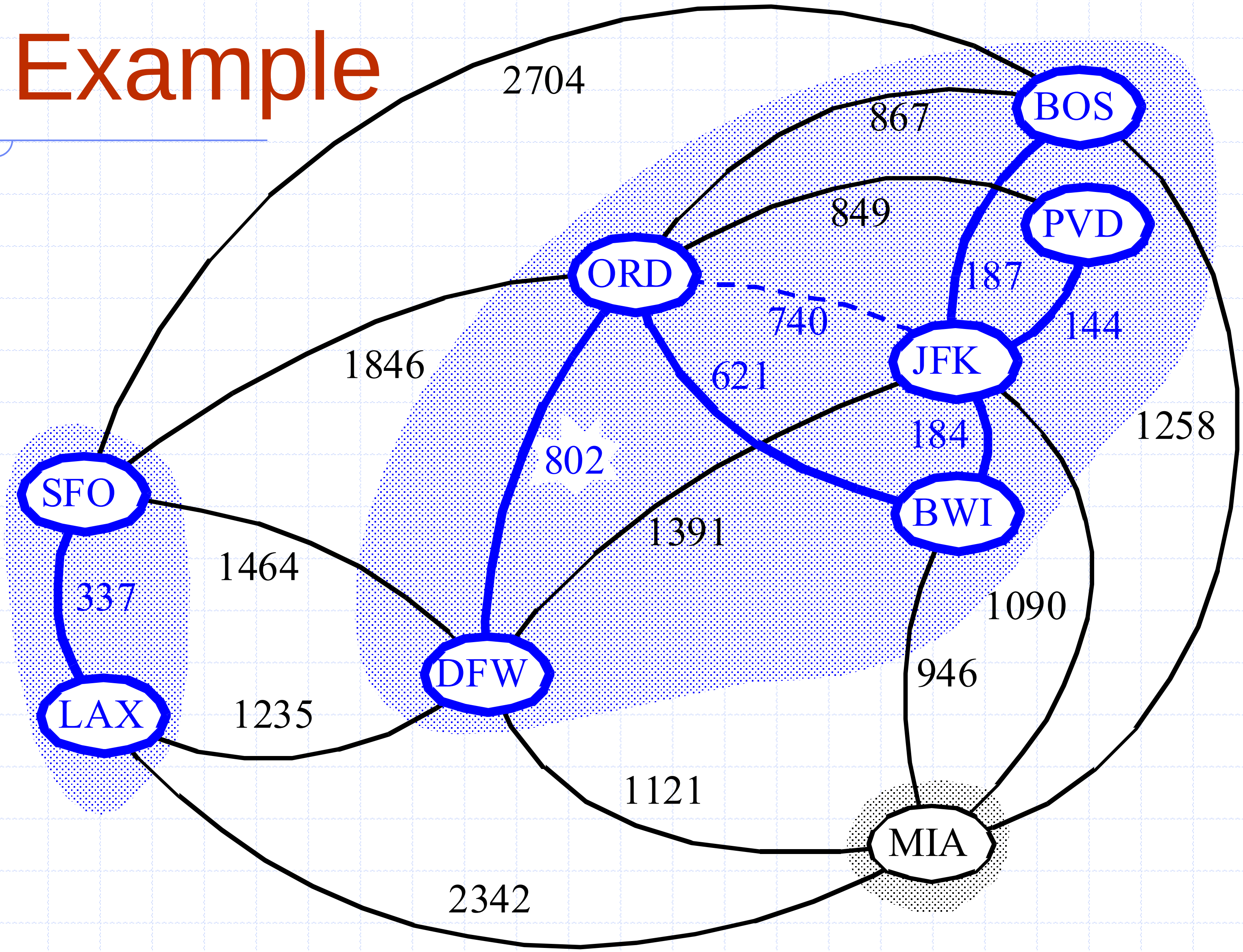
BOS
PVD
ORD
JFK
BWI
SFO
LAX
DFW
MIA
2704
867
849
187
144
740
621
1846
184
802
1391
1258
1464
337
1090
946
1235
1121
2342

Example

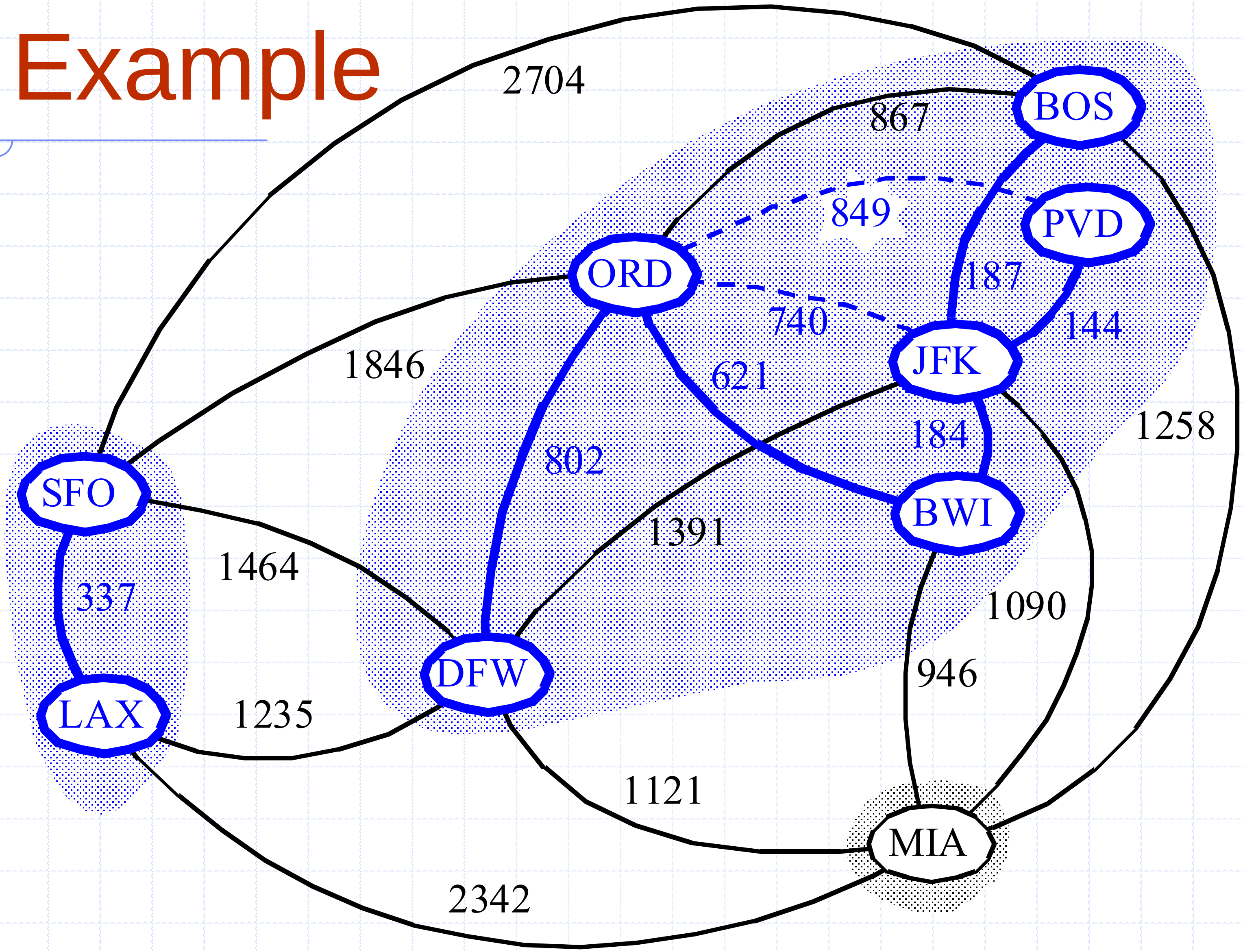


2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
184
1258
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example

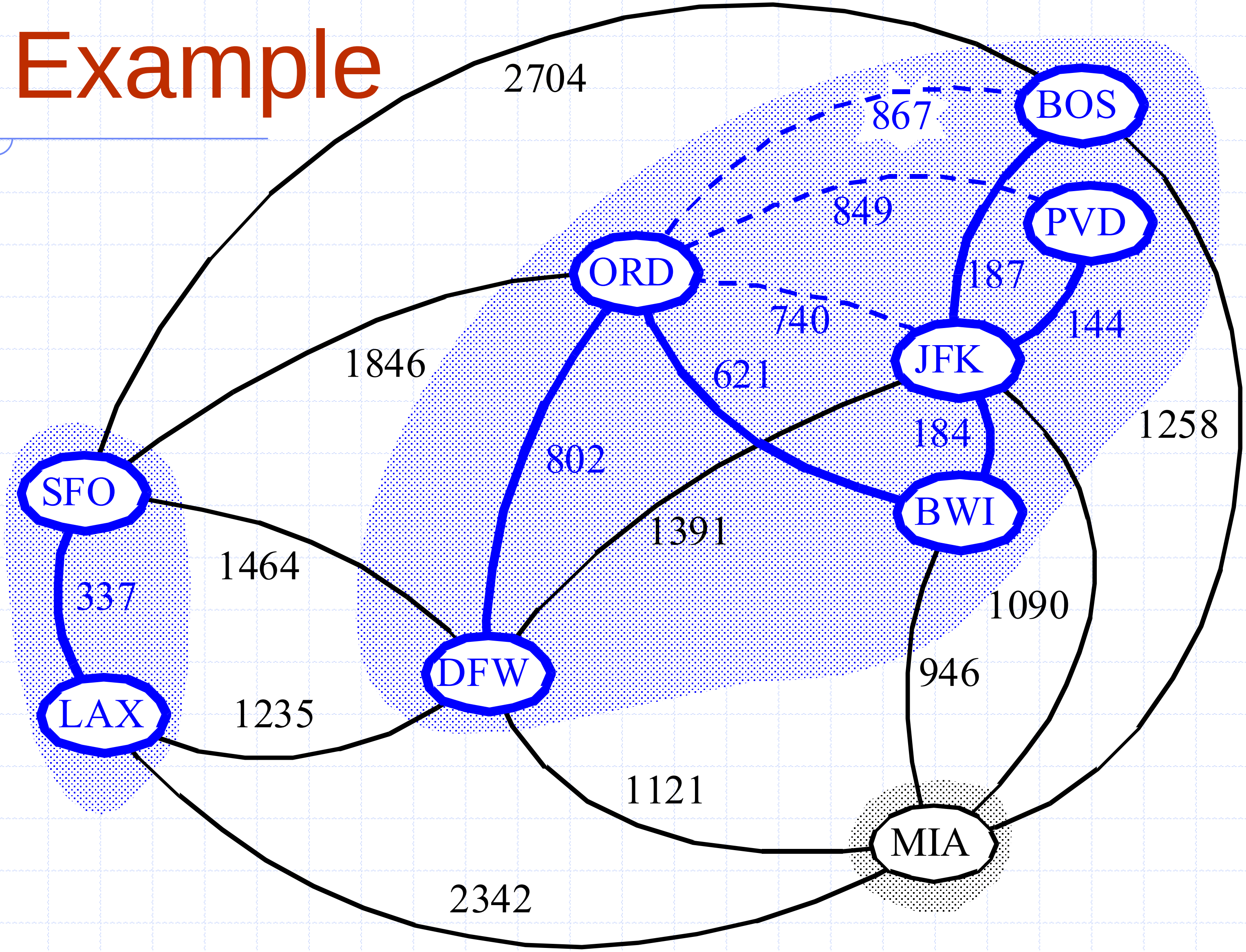

BOS
PVD
ORD
JFK
BWI
SFO
LAX
DFW
MIA
2704
867
849
187
740
144
1846
621
184
1258
802
1391
1464
337
1090
946
1235
1121
2342

Example



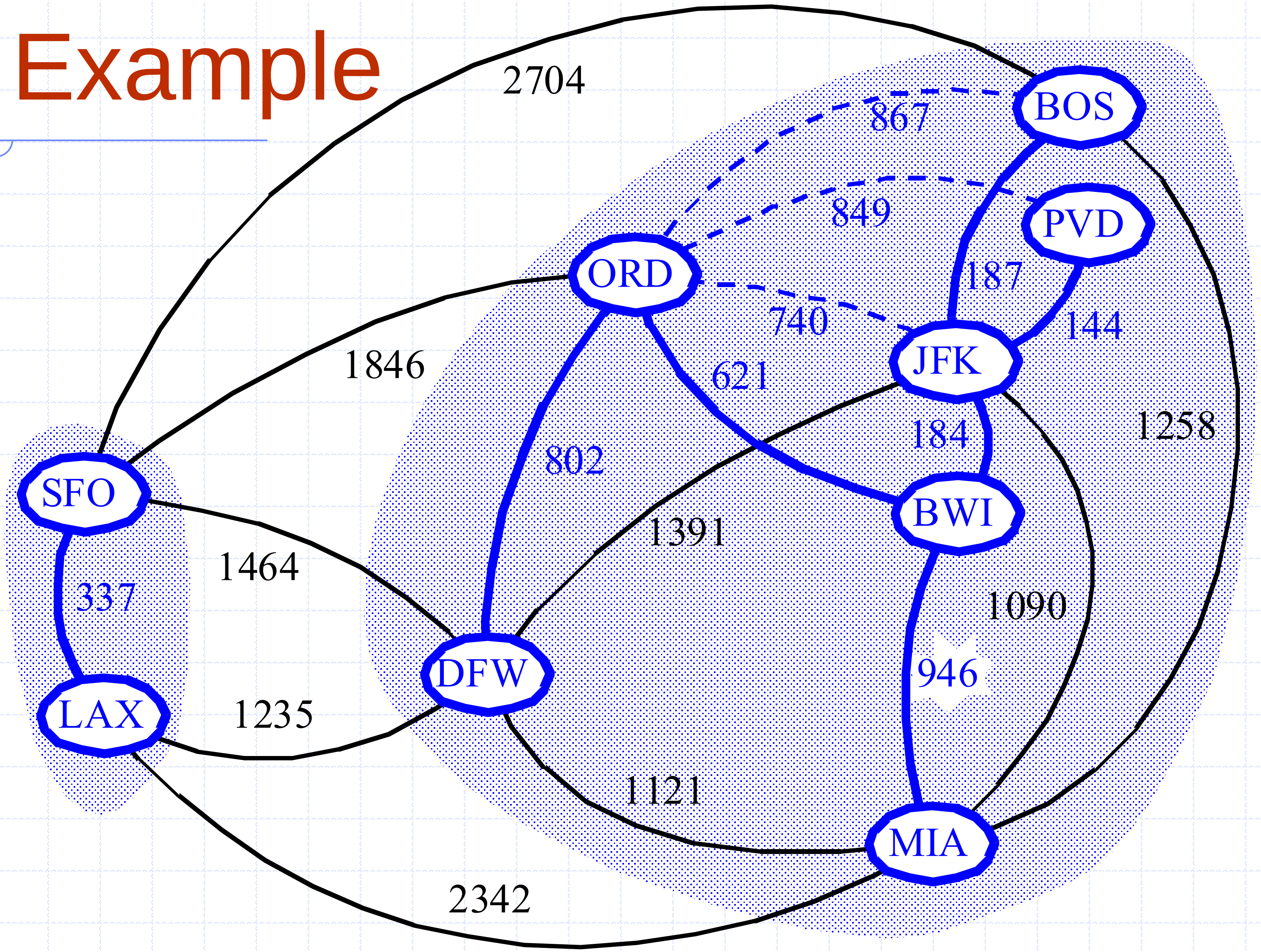
2704
BOS
867
849
PVD
ORD
187
740
144
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



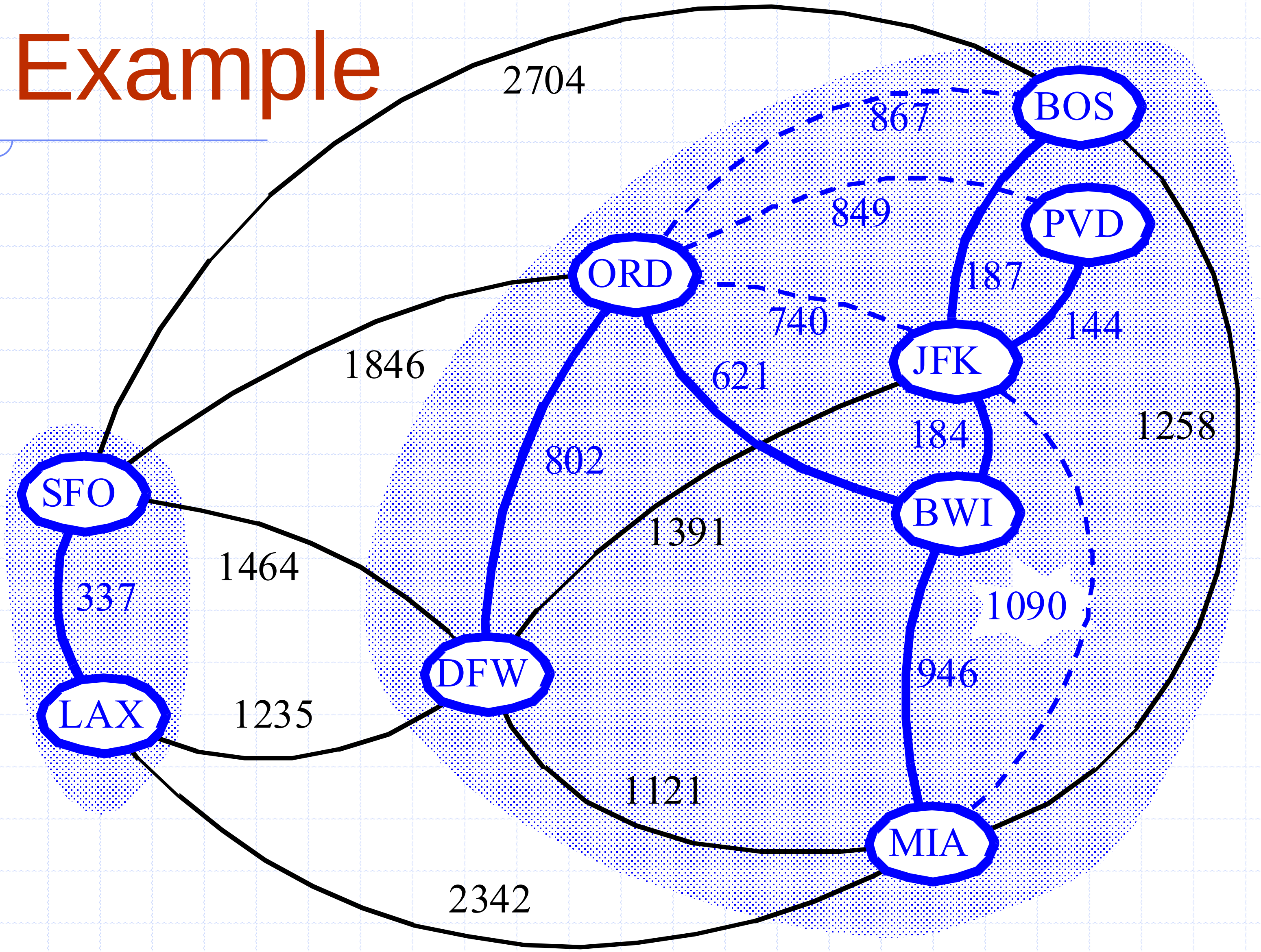
2704
867
BOS
849
PVD
ORD
187
740
144
1846
JFK
621
1258
802
184
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



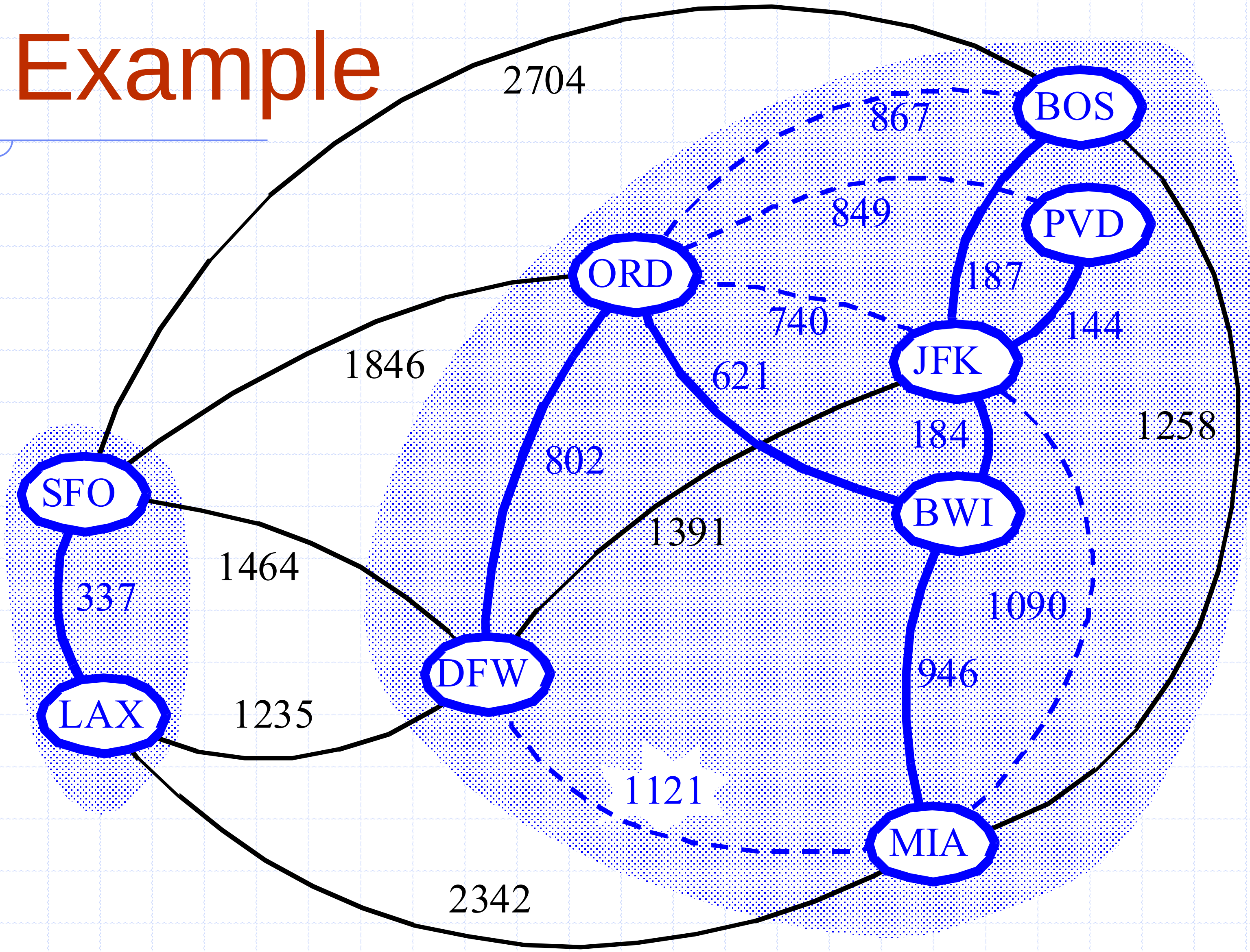
2704
867
BOS
849
PVD
ORD
187
740
144
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



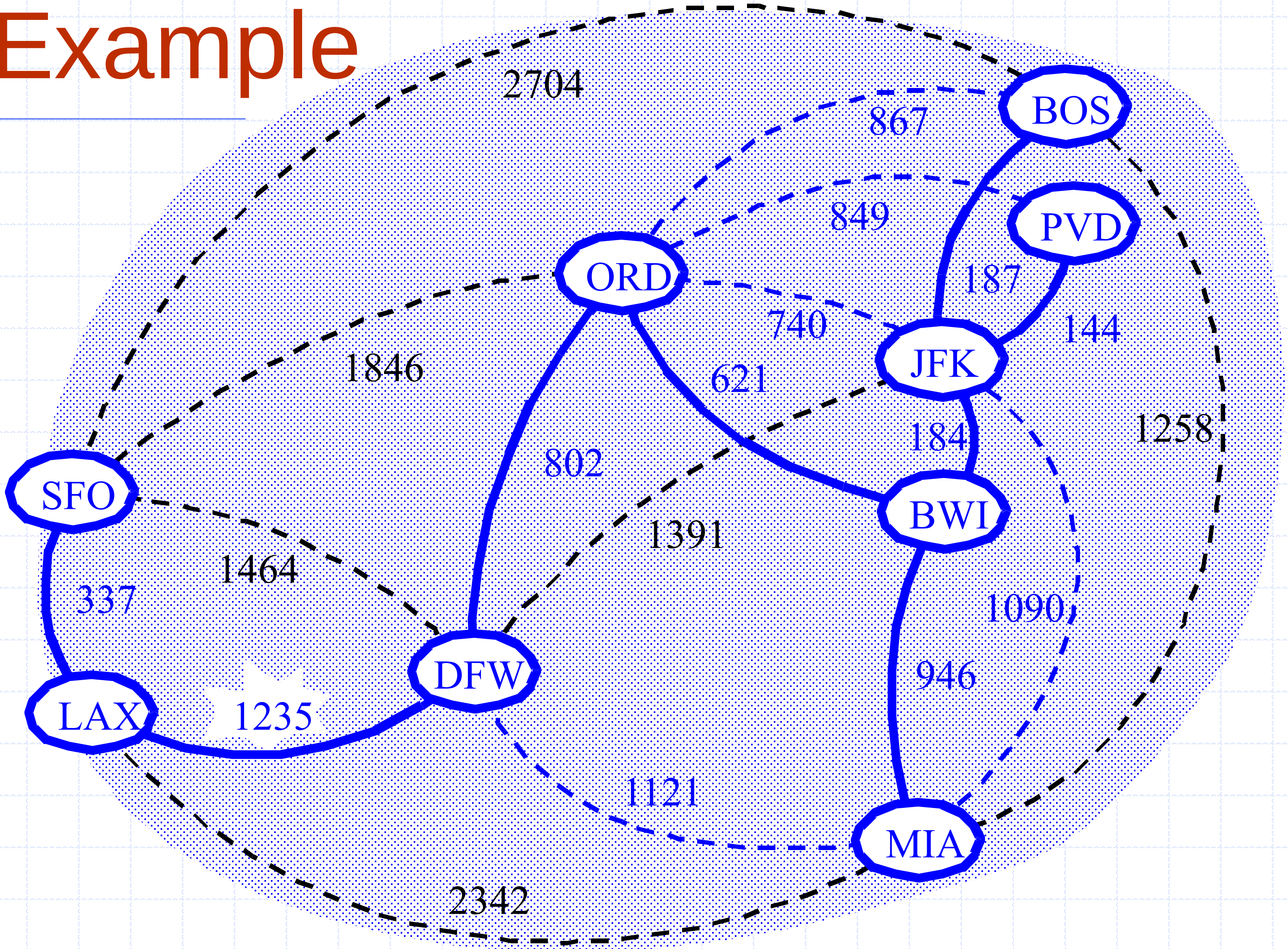
BOS
PVD
ORD
JFK
BWI
SFO
LAX
DFW
MIA
2704
867
849
187
740
144
1846
621
1258
184
802
1391
1464
337
1090
946
1235
1121
2342

Example



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

Example



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

Kruskal's Algorithm (1956) (High Level)

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree *T* which is our MST

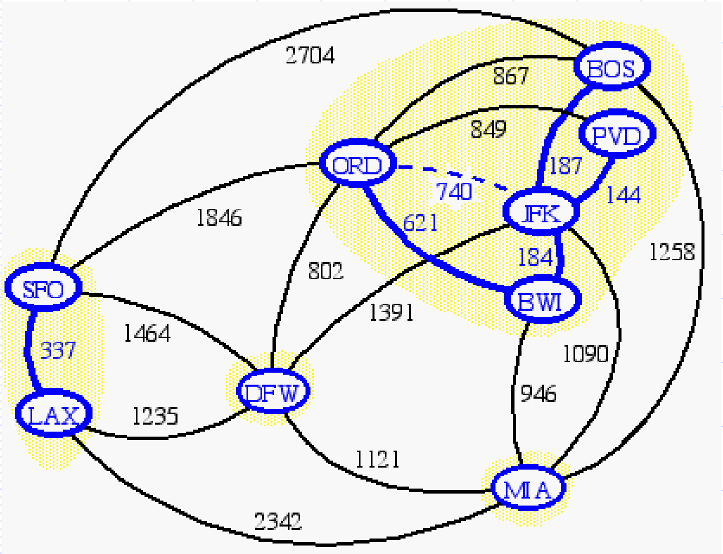
Algorithm *KruskalMST*(*G*)
  for each vertex *v* in *G* **do**
    define a *Cloud(v)* ← {*v*}
  Q ← new heap-based priority queue.
  for all $e \in G.edges()$
    Q.insert(*weight*(*e*), *e*)
  $T \leftarrow \varnothing$
  while *T* has fewer than *n*-1 edges **do**
    e ← *Q.removeMin()*
    (*u*, *v*) ← *G.endVertices(e)*
    if $Cloud(v) \neq Cloud(u)$ **then**
      Add edge *e* to *T*
      Merge *Cloud(v)* and *Cloud(u)*
  return *T*

Data Structure for Kruskal Algorithm

- The algorithm maintains a forest of trees/clouds
- An edge is accepted if it connects distinct trees/clouds
- We need a data structure that maintains **clouds**, e.g., an array of disjoint sets of vertices (clouds), with the operations:

  -**getCloud**(u): returns the index of the cloud containing u (stored at the vertex as an integer attribute)

  -**mergeClouds**(u,v): merge the smaller cloud into the larger cloud to minimize time complexity



Representation of a Cloud

- Each cloud/partition is stored in a sequence
- Each vertex has a reference back to the cloud containing it, which is an integer index into the Clouds array
  - operation getCloud(u) takes O(1) time, and returns the index of the cloud containing u.
  - in operation mergeClouds(u,v), we move the elements of the smaller cloud to the larger cloud and update the cloud attribute of vertices in the smaller cloud
  - the time for operation mergeClouds(u,v) is $MIN(n_u,n_v)$, where n_u and n_v are the sizes of the sequences storing u and v
- For each edge we do two getCloud operations
- We insert n-1 edges into the MST, so we do at most O(n) merges
- Each element is processed (copied into a different cloud) at most log n times (Why?)
  - Because whenever a vertex is processed in a union, it goes into a set of size at least double (Why?)
- O(log m) is O(log n) (Why?)
  - Because $m \leq n(n-1)/2 < n^2$

Partition-Based Implementation

Uses an array to keep track of clouds and the cloud # is the index into the array of clouds.

```
Algorithm Kruskal(G):
  Input: A weighted, simple, connected graph G.
  Output: A MST T for G (T is a sequence containing the edges in the MST).
   Clouds ← new array of size n
   c ← 0
  for all v ∈ G.vertices() do
        setCloudNum(v, c)      // all start out in separate clouds
        Clouds[c] ← {v}        // sequence containing v is stored in Clouds array
        c ← c + 1
    Q ← new heap-based priority queue
    for all e ∈ G.edges() do
       Q.insert(weight(e), e)                 // O(m log m) which is O(m log n)
    T ← new Sequence           // edges in the MST
    while T.size() < n-1 do
       e ← Q.removeMin ()
       (u,v) ← G.endVertices(e)
       if getCloudNum(u) != getCloudNum(v) then
          T.insertLast( e )
          mergeClouds(u,v, Clouds)  {O(n log n) since a vertex is merged O(log n) times}
    return T
```

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

Merge Two Clouds in time O(size of smaller cloud)

```
Algorithm mergeClouds(u, v, Clouds)
 Input: Vertices u and v and array Clouds containing all clouds that still exist
     (or are non-empty).
 Output: merges the two clouds containing u and v
     cu ← getCloudNum(u)
     cv ← getCloudNum(v)
     if Clouds[cu].size() ≥ Clouds[cv].size()  then
           larger ← Clouds[cu];  smaller ← Clouds[cv]
           newCloud ← cu        // cloud # of larger
     else
           larger ← Clouds[cv];  smaller ← Clouds[cu]
           newCloud ← cv        // cloud # of larger
     while smaller.size() > 0  do   // move vertices from smaller to larger cloud
           p ← smaller.first()
           v ← smaller.remove(p)           // smaller cloud will be empty
           setCloudNum(v, newCloud)    // set to cloud # of larger cloud
           larger.insertLast(v)
```

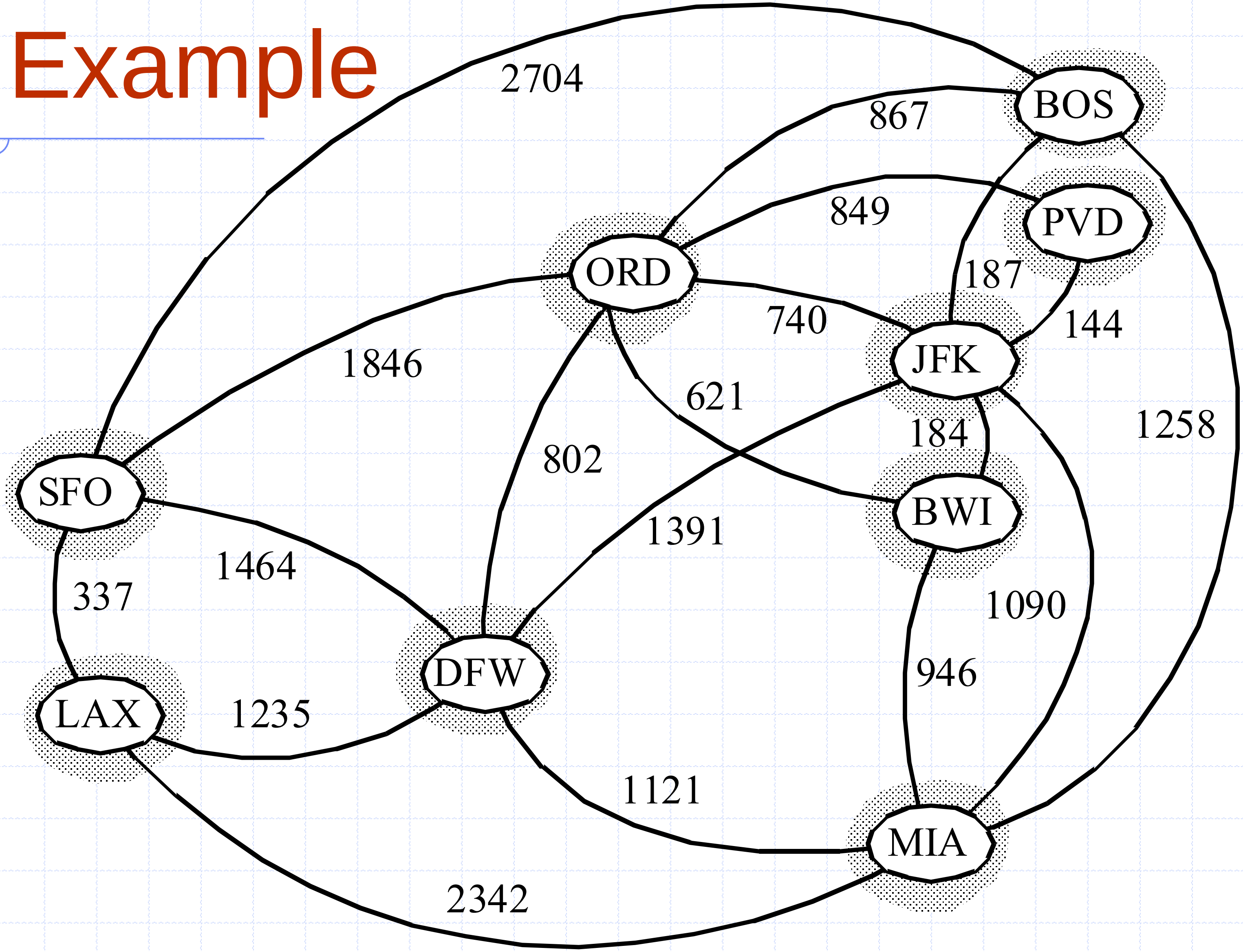
Main Point

3. Kruskal’s minimum spanning tree algorithm first finds the shortest edge, then tests it for safety; if it passes, it becomes part of the proposed solution.

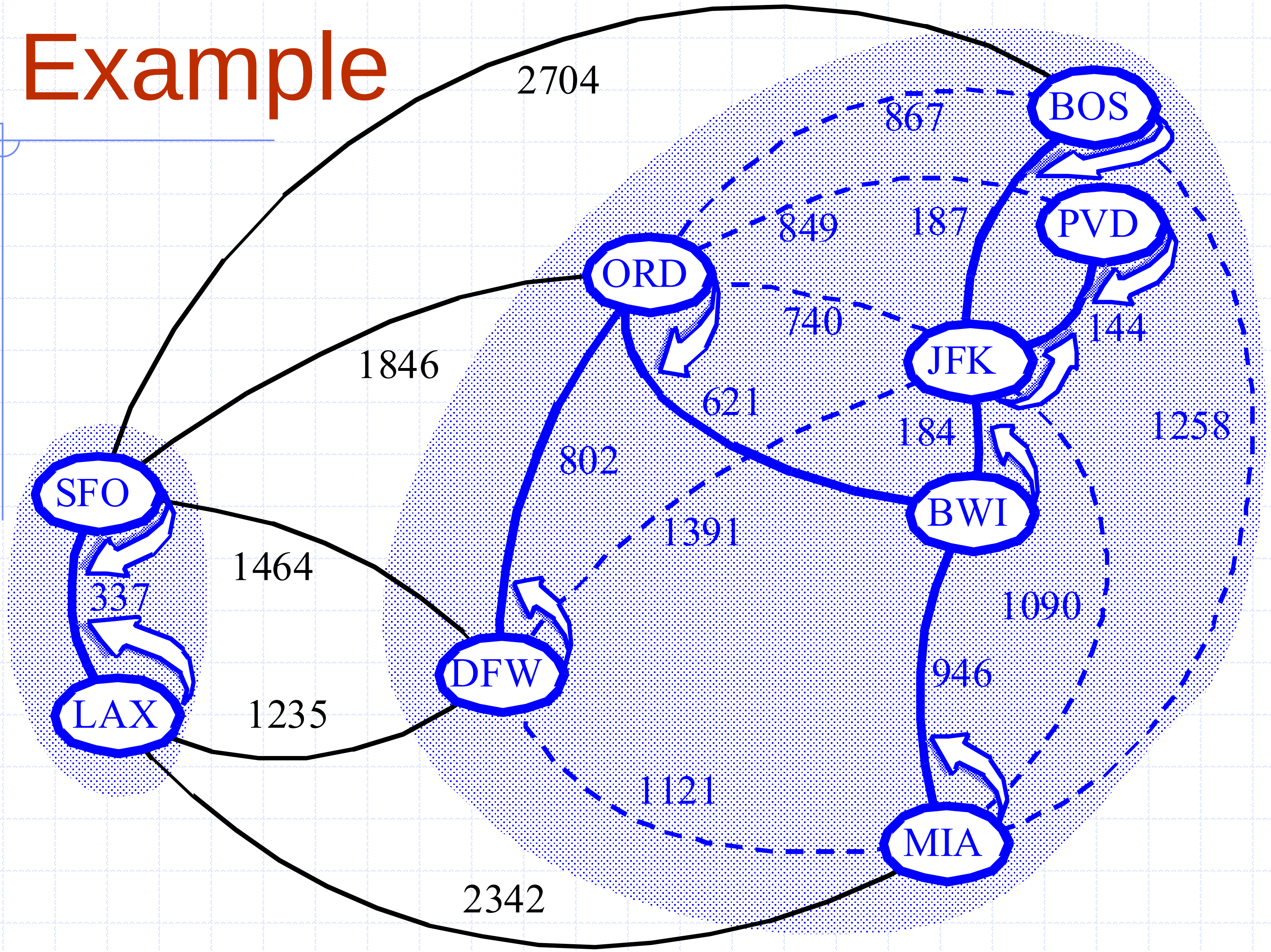
   The fruits of an action cannot be determined precisely on the level of waking consciousness; it can only be determined from the level of infinite correlation where everything is interconnected. Thus only by establishing our awareness in pure awareness is right action possible.

Baruvka's Algorithm (1926)

Baruvka Example

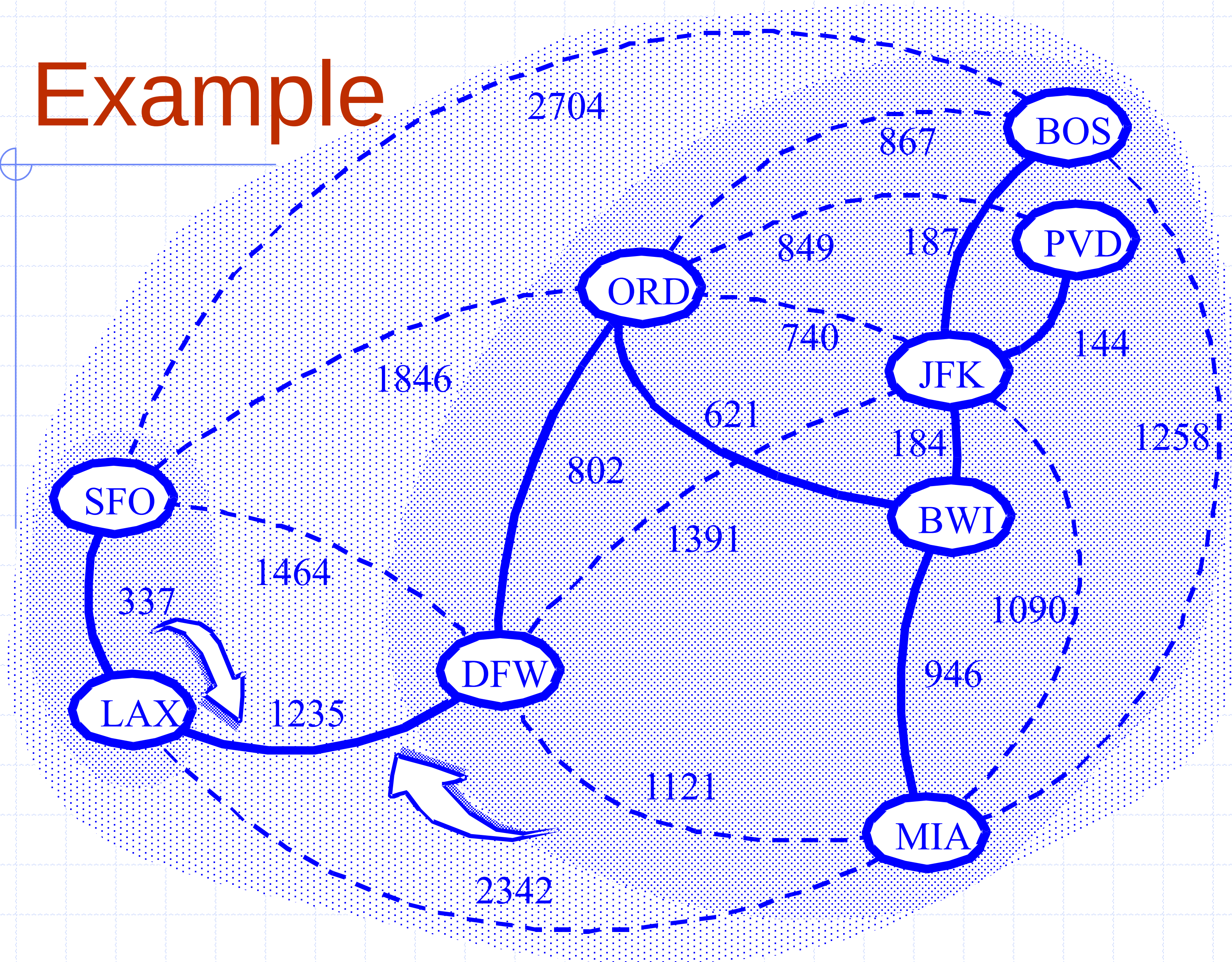


Example



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

Example



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

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, no edges, n connected components}
  while T has fewer than n-1 edges **do** {T is not yet an MST}
    for each connected component C in T **do**
      Find edge e with 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.

Baruvka's Algorithm (more details)

Algorithm ***BaruvkaMST***(***G***)
  for each e ∈***G.edges()*** **do** {label edges *NOT_IN_MST*}
    *setMSTLabel(e, **NOT_IN_MST**)* {no edges in MST}
  numEdges ← *0*
  while *numEdges* < ***n***-1 **do**
    labelVerticesOfEachComponent(G) {BFS}
    insertSmallest-WeightEdgeOutOfComponents(G)
  return ***G***

Required functionality

- Does not use a priority queue or locators!
- Does not use union-find data structures!
- Maintains a forest T subject to edge insertion
  - Can be supported in O(1) time using labels on edges in MST
- Step 1: Mark vertices with number of the component to which they belong
  - Traverse forest T to identify connected components
    - O(1) time to label each vertex
    - Requires extra instance variable for each vertex
  - Takes O(n+m) time using a DFS or a BFS each time through the while-loop
- Step 2: Find a smallest-weight edge in E incident on each component C in T (insert into MST)
  - Scan edges to find the minimum weight edge incident on one of the vertices in C and incident on another vertex not in C
  - Takes O(m) each time through the for-loop

Analysis of Baruvka's Algorithm

- While-loop: each iteration (at worst) halves the number of connected components in T
  - Thus executed log n times
- Identifying connected components (in for-loop)
  - Vertices are labelled with component number
  - DFS or BFS of T runs in O(m+n) time (Skip edges of G that are not in T)
- For each component C find smallest edge connecting C to a different component of T
  - Scan edges in G
  - O(m) time
- The running time is O(m log n).

Output of the three MST algorithms is different

- Prim-Jarnik Algorithm (1957, 1930)
  - The edges in the MST are stored at the vertices by setParent(e)
- Kruskal's Algorithm (1956)
  - The MST edges are returned in a Sequence T
- Baruvka's Algorithm (1926)
  - The MST edges are labelled IN_MST

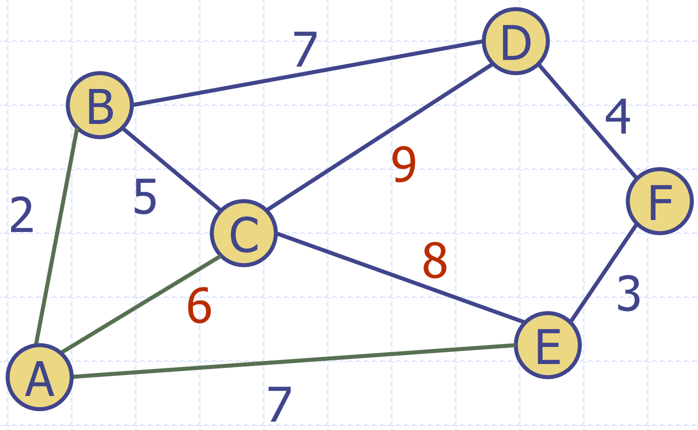
After labeling each vertex with its component number (HW13)

```
Algorithm DFS(G)
  Input graph G
  Output the edges of G are labeled as
         discovery edges and back edges

    initResult(G)
    for all  u ∈ G.vertices()
        setLabel(u, UNEXPLORED)
        postVertexInit(G, u)
    for all  e ∈ G.edges()
        setLabel(e, UNEXPLORED)
        postEdgeInit(G, e)
    for all  v ∈ G.vertices()
        if  getLabel(v) = UNEXPLORED
            preComponentVisit(G, v)
            DFScomponent(G, v)
            postComponentVisit(G, v)

    result(G)
```

```
Algorithm DFScomponent(G, v)
  setLabel(v, VISITED)
  startVertexVisit(G, v)
  for all  e ∈ G.incidentEdges(v)
    preEdgeVisit(G, v, e, w)
    if  getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      edgeVisit(G, v, e, w)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        preDiscoveryVisit(G, v, e, w)
        DFScomponent(G, w)
        postDiscoveryVisit(G, v, e, w)
      else
        setLabel(e, BACK)
        backEdgeVisit(G, v, e, w)
  finishVertexVisit(G, v)
```

HW14: Insert into T the smallest-weight edge going out from each component

Cycle Property



By the Cycle Property, AC, CD, and CE cannot be in a MST.
What about AE and BD?

Let's run the algorithms to verify this.

Lower Bound on MST Computation

- There are **randomized algorithms** that compute MST's in **expected linear time**
- Linear time seems to be the lower bound
- Unknown whether there is a deterministic algorithm that runs in linear time (open question)

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Finding the minimum spanning tree can be done by an exhaustive search of all possible spanning trees, then choosing the one with minimum weight (but that takes at least exponential time).

2. To devise a greedy strategy, we identify a set of candidate choices, determine a selection procedure, and consider whether there is a feasibility problem. Then we have to prove that the strategy works.

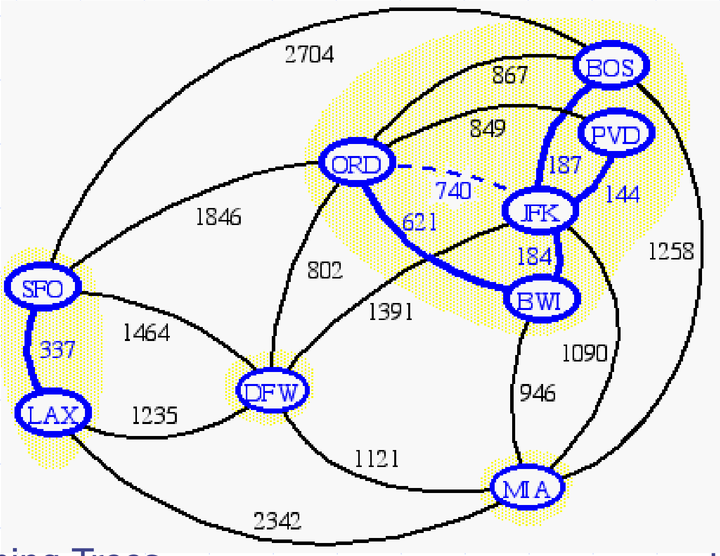
3. **Transcendental Consciousness** is the home of all the laws of nature, the source of all algorithms.
4. **Impulses within Transcendental Consciousness**: The natural laws within this unbounded field are the algorithms of nature governing all the activities of the universe.
5. **Wholeness moving within itself:** In Unity Consciousness, we perceive the spanning tree of natural law and appreciate the unity of all creation.

Union-Find Data Structure for Kruskal Algorithm

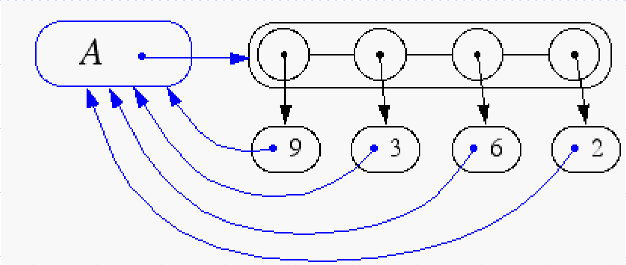
- The algorithm maintains a forest of trees
- An edge is accepted if it connects distinct trees
- We need a data structure that maintains a **partition**, i.e., a collection of disjoint sets, with the operations:

  -**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
- For each edge we do two finds
- We insert n-1 edges into the MST, so we do at most O(n) unions
- Each element is processed (copied into a different cloud) at most log n times (Why?)
  - Because whenever a vertex is processed in a union, it goes into a set of size at least double (Why?)
- O(log m) is O(log n) (Why?)
  - Because $m \leq n(n-1)/2 < n^2$

Partition-Based Implementation

Performs cloud merges as unions and tests as finds.

```
Algorithm Kruskal(G):
  Input: A weighted, simple, connected graph G.
  Output: An MST T for G.
    for all v ∈ G.vertices() do
        insert vertex v into T
        define a Cloud(v) ← {v}
    Q ← new heap-based priority queue
    for all e ∈ G.edges() do
        Q.insert(weight(e), e)                {O(m log m) which is O(m log n)}
    T ← new empty tree
    while T.numEdges() < n-1 do
        e ← Q.removeMin ()
        (u,v) ← G.endVertices(e)
        if P.find(u) != P.find(v) then
            insert edge e into T
            P.union(u,v)           {O(n log n) since a vertex is merged O(log n) times}
    return T
```

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