COP 3530: Computer Science III
Summer 2005

Graphs and Graph Algorithms – Part 6

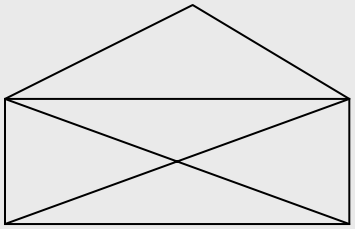
Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790

http://www.cs.ucf.edu/courses/cop3530/summer05

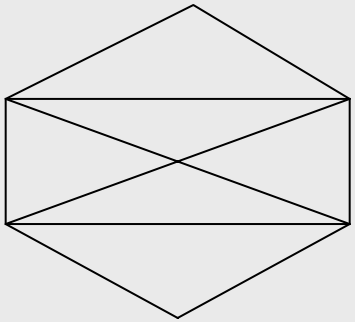
School of Computer Science
University of Central Florida

Euler Paths and Circuits

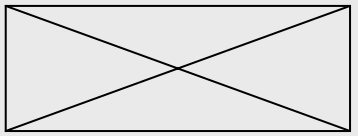
- Consider the three figures (a) – (c) shown below. A puzzle for you to solve is to reconstruct these three figures using a pencil and paper drawing each line exactly once without lifting the pencil from the paper while drawing the figure.
- To make the puzzle even harder, see if you can draw the figure following the rules above but have the pencil finish at the same point you originally started the drawing. Try to do this before you read any further in the notes.



(a)



(b)

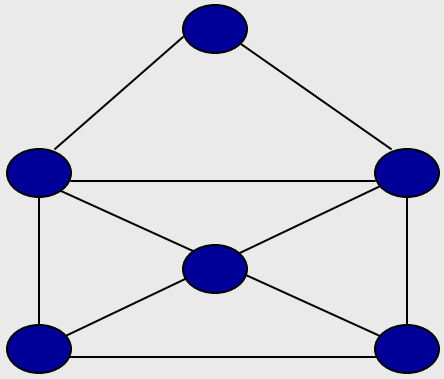


(c)

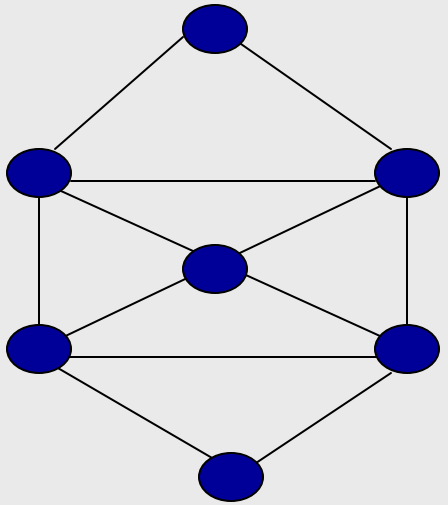
Euler Paths and Circuits (cont.)

- It turns out that these puzzles have a fairly simple solution.
- Figure (a) can only be drawn within the specified rules, if the starting point is the lower-left or lower-right hand corner, and it is not possible to finish at the starting point.
- Figure (b) is easily drawn with the finishing point being the same as the starting point (see the page XX for one possible solution).
- Figure (c) cannot be drawn at all within the specified rules, even though it appears to be the simplest of the drawings!
- These puzzles are converted into graph theory problems by assigning a vertex to each intersection. Then the edges are assigned in the natural manner. The corresponding graphs are shown on the next page.

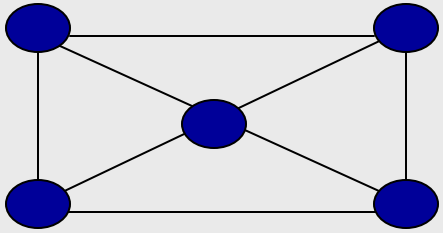
Euler Paths and Circuits



(a)



(b)



(c)

Euler Paths and Circuits (cont.)

- To find an Euler path, once the puzzle has been converted into the graphs as shown on the previous page, our problem becomes one of finding a path that visits every edge exactly once.
- If the extra challenge is to be solved, then a cycle must be found that visits every edge exactly once. This is an Euler circuit.
  - This problem was solved in 1736 by the mathematician Euler and is commonly regarded as the beginning of graph theory.
- The Euler path and Euler circuit problems, although slightly different problems, have the same basic solution and we will focus only on the Euler circuit problem.

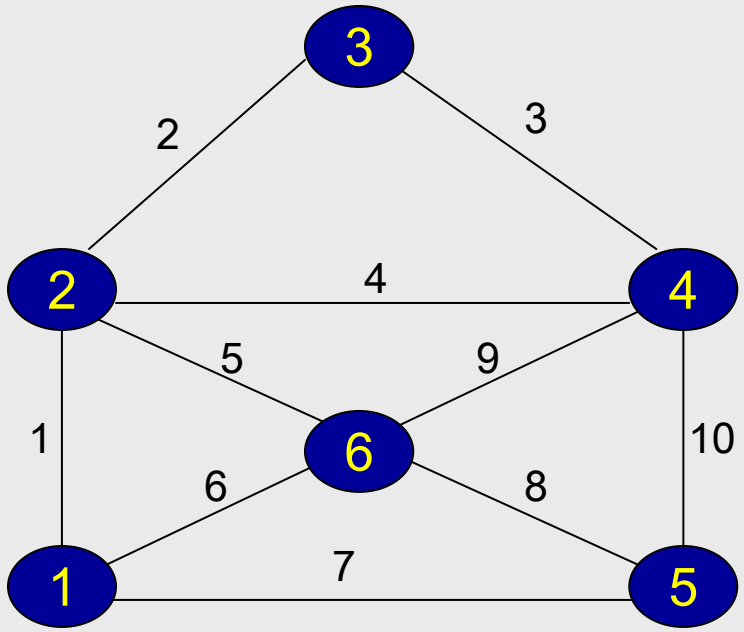
Euler Paths and Circuits (cont.)

- For a given graph to have an Euler circuit certain properties must hold in the graph.
- Namely, since an Euler circuit must begin and end on the same vertex, such a circuit is only possible if:
  - (1) the graph is connected and,
  - (2) each vertex in the graph has an even degree.
- If any vertex were to have an odd degree, then eventually you would reach the point where only one edge into that vertex is "unvisited", and taking that edge into that vertex would strand you at that vertex.

Euler Paths and Circuits (cont.)

- If exactly two vertices have an odd degree, then a Euler path is still possible (since you are not required to begin and end on the same vertex in an Euler path) if the path begins on one of the odd degree vertices and ends on the other odd degree vertex.
- If more than two vertices have an odd degree, then an Euler path is not possible.
- Applying this knowledge to the earlier graphs we see that:
  - Graph (a) has only an Euler path beginning at either the lower left or lower right corners which are the two vertices with an odd degree. All other vertices in this graph have an even degree of either 2 or 4.
  - Graph (c) has neither an Euler circuit nor an Euler path since there are four vertices in this graph which have an odd degree.
  - Graph (b) however has no vertices of odd degree and thus does have an Euler circuit (as well as an Euler path).

Euler Path For Graph (A)



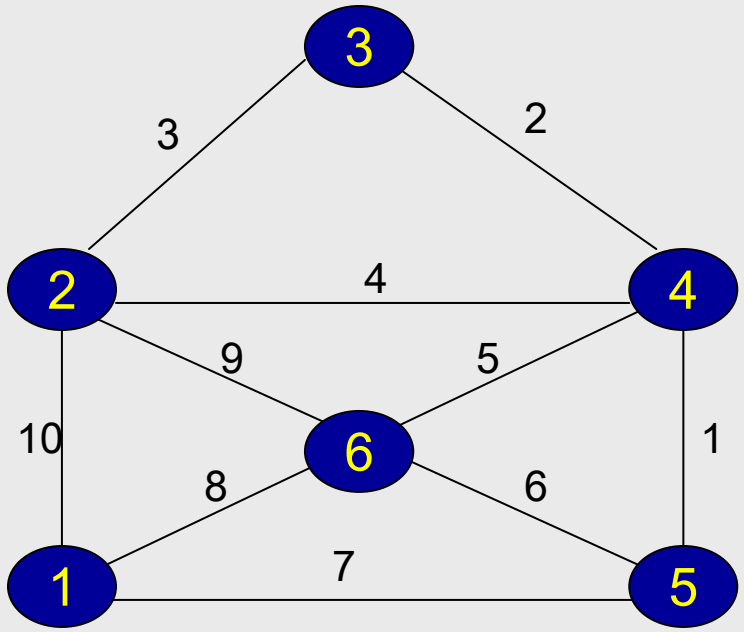
Order of traversal of edges for Euler path

1 – 2
2 – 3
3 – 4
4 – 2
2 – 6
6 – 1
1 – 5
5 – 6
6 – 4
4 – 5

Starting vertex

Ending vertex

Another Euler Path For Graph (A)



Order of traversal of edges for Euler path

5 – 4
4 – 3
3 – 2
2 – 4
4 – 6
6 – 5
5 – 1
1 – 6
6 –2
2 - 1

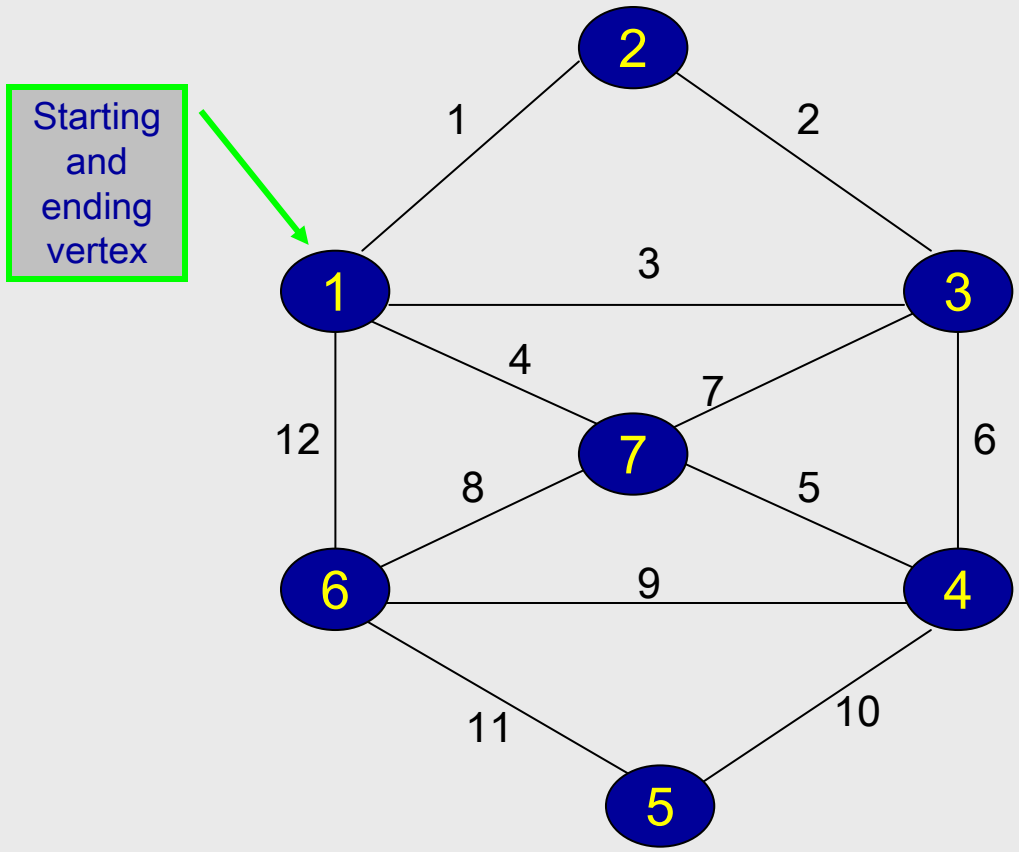
Ending vertex

Starting vertex

Euler Paths and Circuits (cont.)

- The necessary and sufficient condition for a graph to have an Euler circuit turns out to be exactly the conditions we have just described.
- Thus, *any* connected graph in which all the vertices have even degree, must have an Euler circuit.
- It also turns out that an Euler circuit can be found in linear time!
- The algorithm which is capable of performing this operation is a depth-first search.
- The basic problem that must be overcome by such an algorithm is that only a portion of the graph may have been visited before you return to the original starting vertex.
- If all the edges coming out of the start vertex have been traversed. then part of the graph will be un-traversed. The easiest way to fix this problem is to find the first vertex on the path which has an un-traversed edge, and perform another depth-first search from this node. This will give another circuit, which can be spliced into the original. This process is continued until all edges have been traversed.

Euler Circuit For Graph (B)

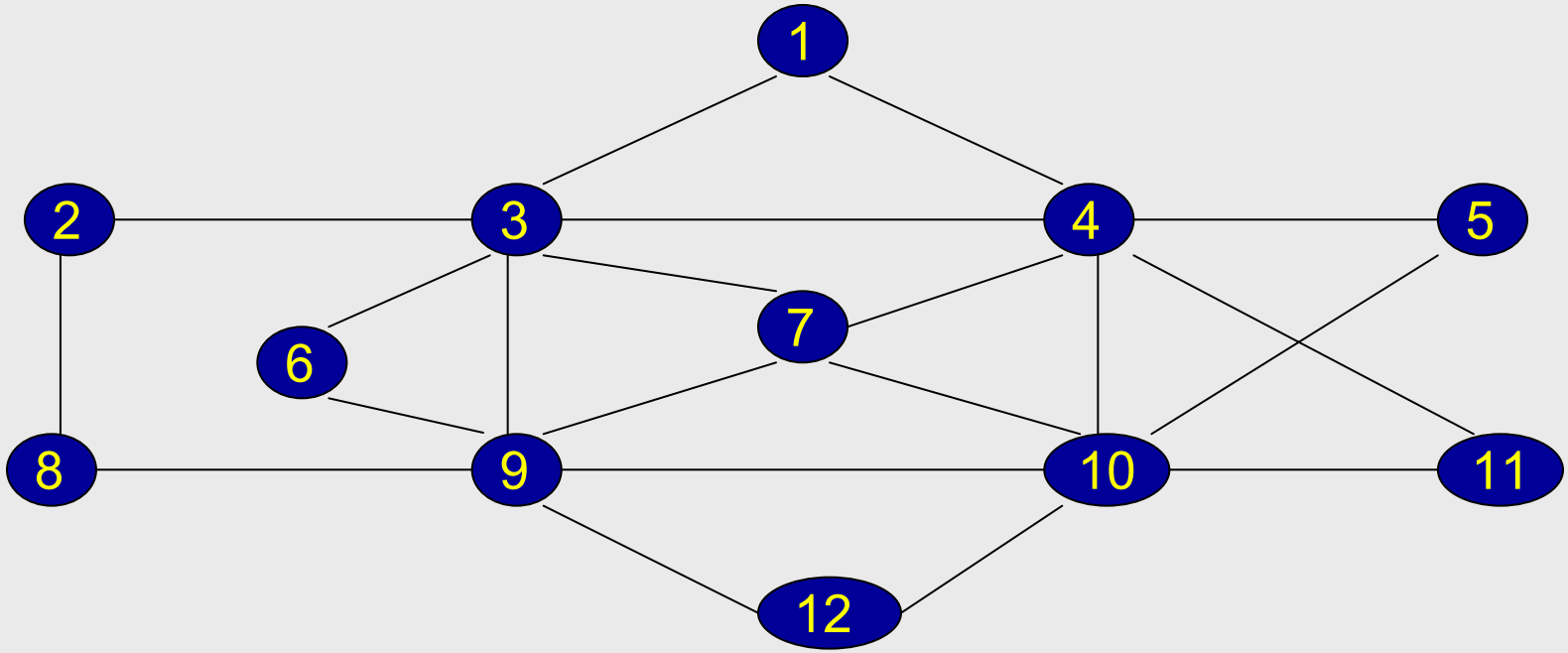


Order of traversal of edges for Euler circuit
1 – 2
2 – 3
3 – 1
1 – 7
7 – 4
4 – 3
3 – 7
7 – 6
6 – 4
4 – 5
5 – 6
6 – 1

Euler Circuit Example

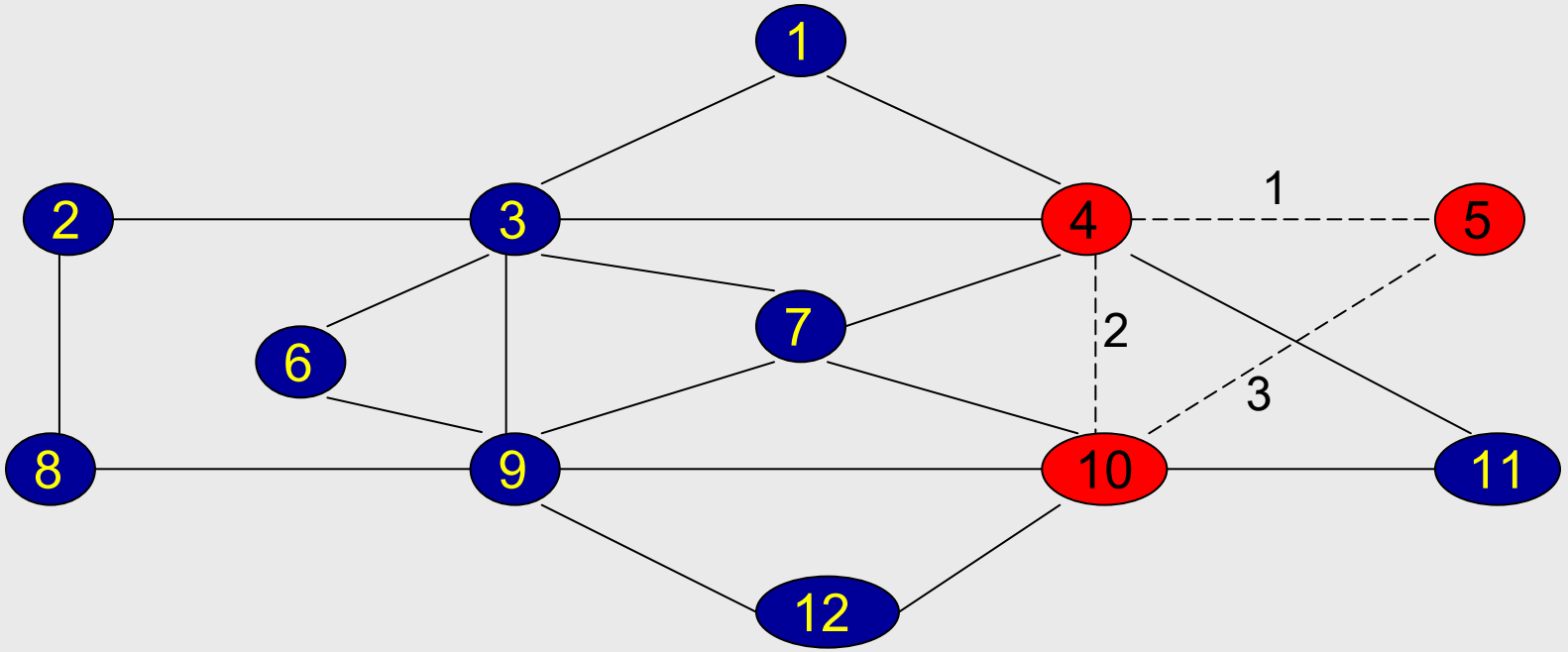
- Does the graph below have an Euler circuit?

  Yes – every vertex has even degree.



Euler Circuit Example (cont.)

A depth-first search beginning at vertex 5 produces the circuit 5 – 4 – 10 – 5.

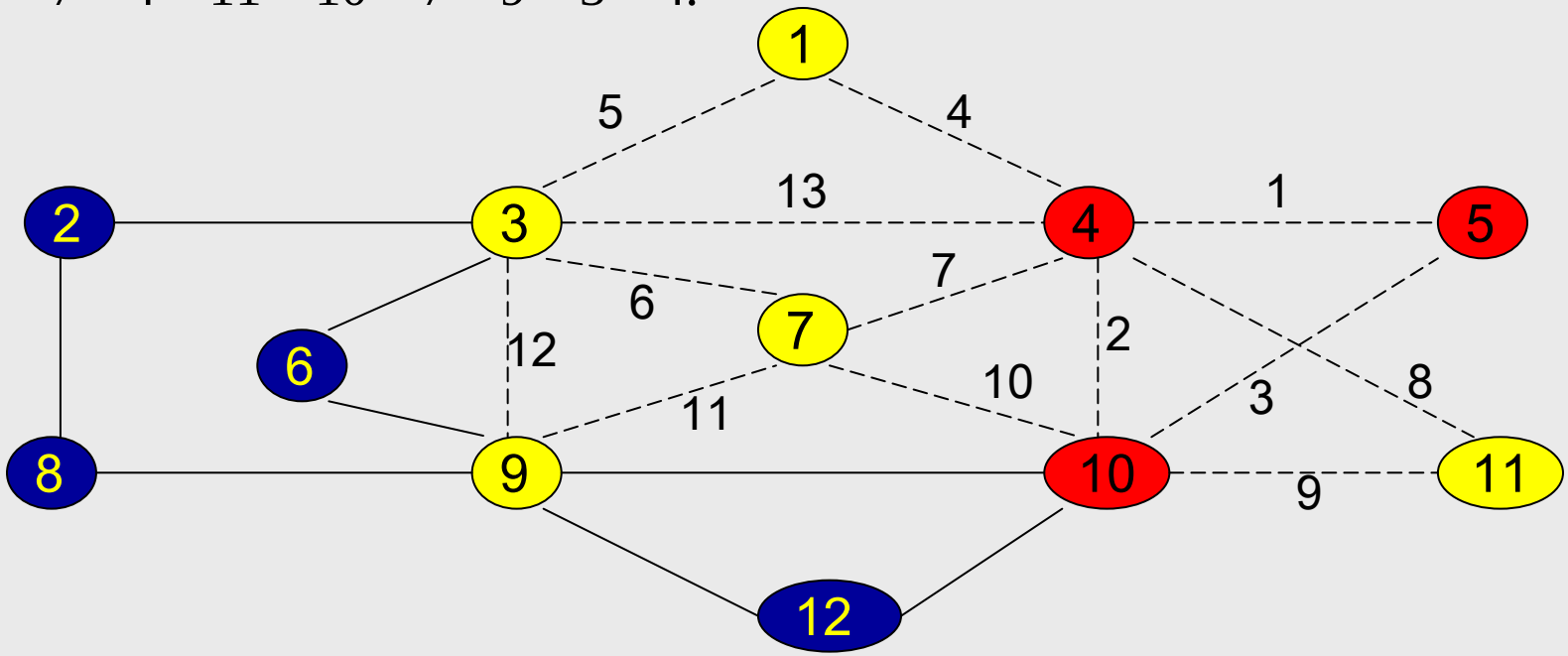


Notice that we are now stuck as there are no un-traversed edges out of the start vertex – yet most of the graph is still un-traversed.

Euler Circuit Example (cont.)

We continue from vertex 4 (the next vertex in the circuit) which still has un-traversed edges.

One possible depth-first search from vertex 4 would produce the circuit: 4 – 1 – 3 – 7 – 4 – 11 – 10 – 7 – 9 – 3 – 4.



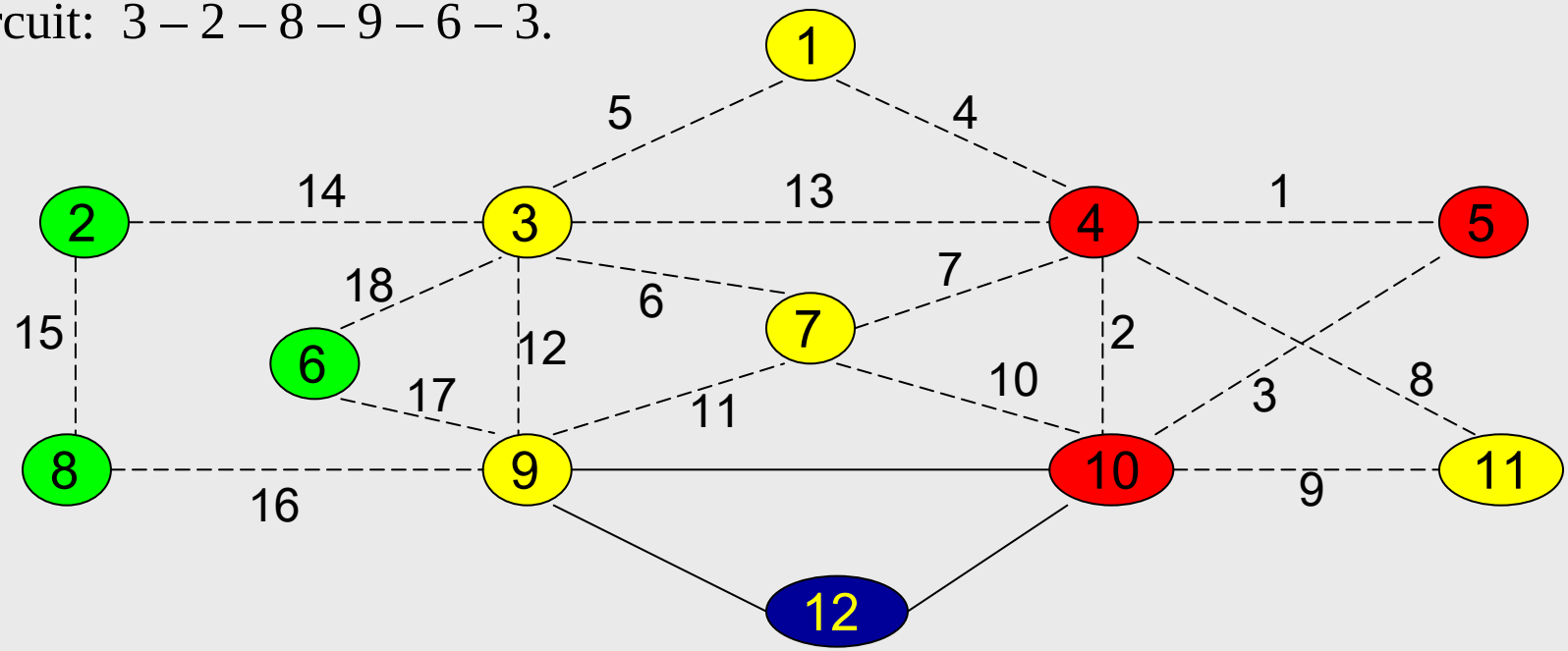
This new circuit is "spliced" into the existing circuit to produce the circuit:

5 – 4 – 1 – 3 – 7 – 4 – 11 – 10 – 7 – 9 – 3 – 4 – 10 – 5.

Euler Circuit Example (cont.)

The current circuit is: 5 – 4 – 1 – 3 – 7 – 4 – 11 – 10 – 7 – 9 – 3 – 4 – 10 – 5.

All edges from vertices 5, 4, and 1 have been traversed. Vertex 3 is the next vertex which still has un-traversed edges and is thus selected as the next vertex to begin a new depth-first search. This search might produce the following circuit: 3 – 2 – 8 – 9 – 6 – 3.



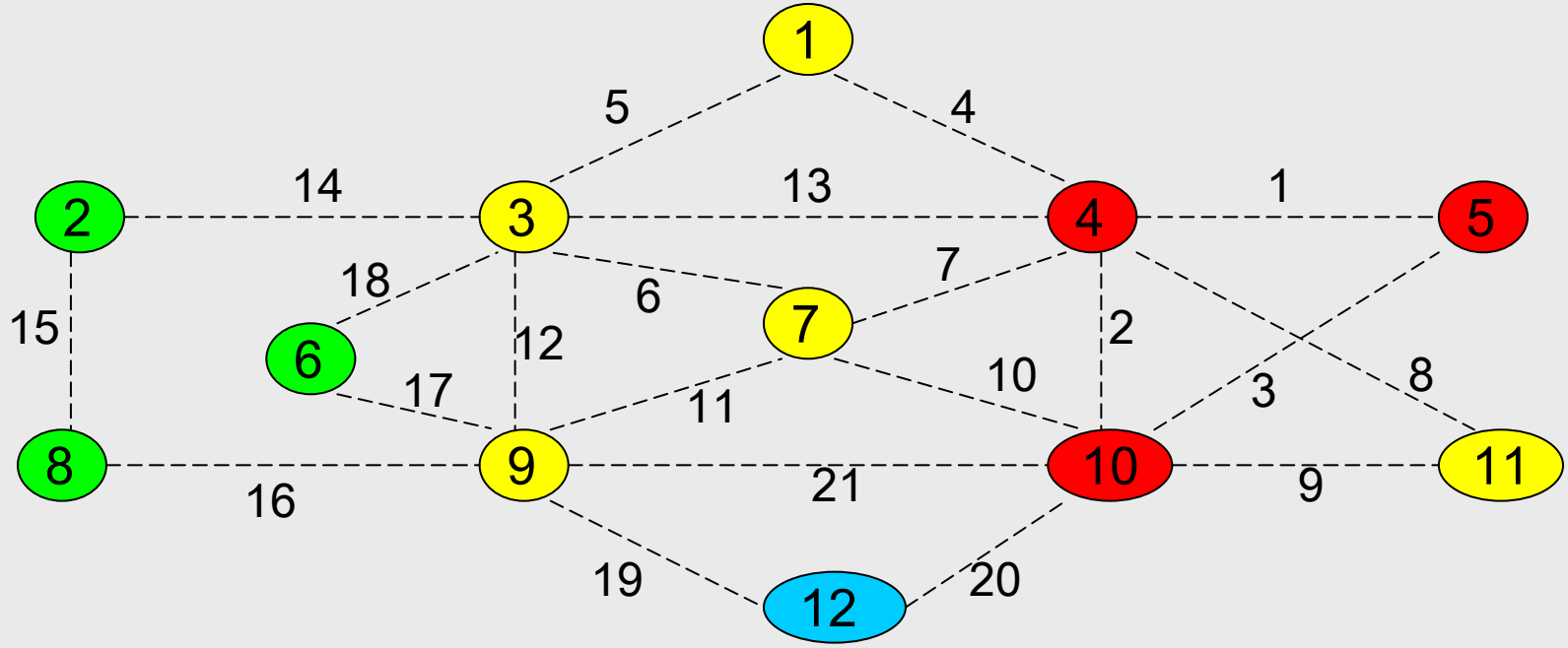
This new circuit is "spliced" into the existing circuit to produce the circuit:

5 – 4 – 1 – 3 – 2 – 8 – 9 – 6 – 3 – 7 – 4 – 11 – 10 – 7 – 9 – 3 – 4 – 10 – 5.

Euler Circuit Example (cont.)

The current circuit is: 5 – 4 – 1 – 3 – 2 – 8 – 9 – 6 – 3 – 7 – 4 – 11 – 10 – 7 – 9 – 3 – 4 – 10 – 5.

The next vertex along the circuit which still has un-traversed edges is vertex 9. A depth-first search at vertex 9 might produce the following circuit: 9 – 12 – 10 – 9.



This new circuit is "spliced" into the existing circuit to produce the final circuit:

5 – 4 – 1 – 3 – 2 – 8 – 9 – 12 – 10 – 9 – 6 – 3 – 7 – 4 – 11 – 10 – 7 – 9 – 3 – 4 – 10 – 5

Efficiency of Euler Circuit Producing Algorithms

- The implementation issues that concern any algorithm which determines an Euler circuit are concerned mainly with the efficiency of the circuit splicing operation.
- To do this efficiently requires that the circuit being constructed be maintained as a linked list so that new sub-circuits can be easily added to the middle of an existing circuit as we did in the previous example.
- To avoid repetitious scanning of the adjacency lists which define the graph it is best to maintain (for each list) a record of the last edge traversed. When a path is spliced in, the search for a new vertex from which to perform the next depth-first search must begin at the start of the splice point. This will guarantee that the total work performed on the vertex search phase is $O(|E|)$ during the entire lifetime of the algorithm.
- With the appropriate data structures in place, the running time of an algorithm to determine the Euler circuit will be $O(|E| + |V|)$.

Minimum Spanning Tree

Spanning subgraph

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

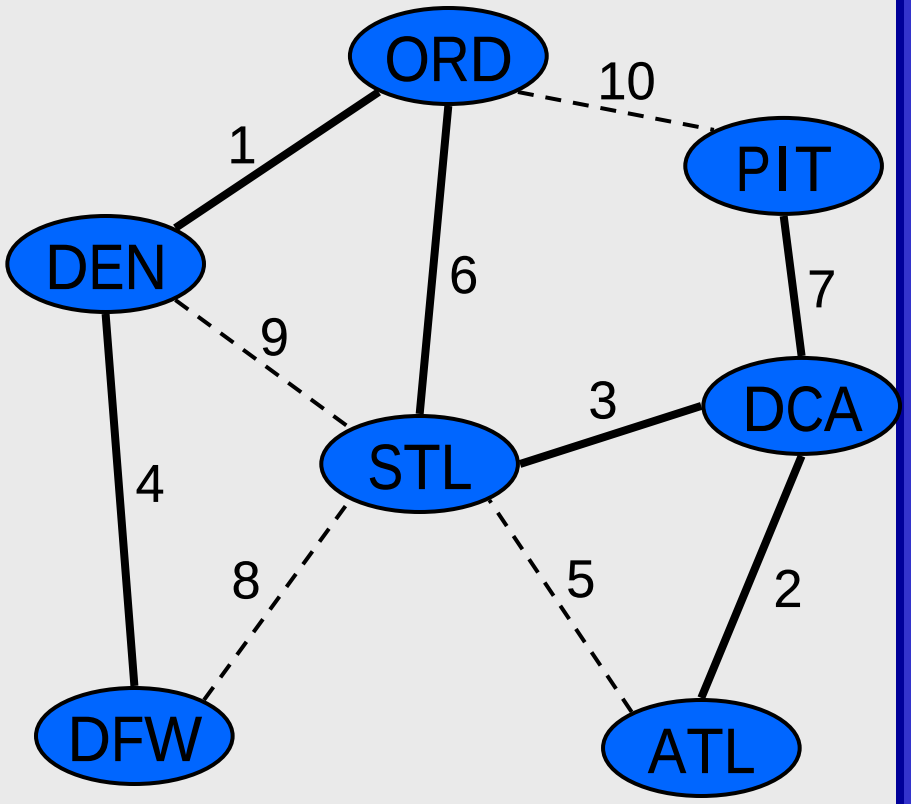
Spanning tree

- 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



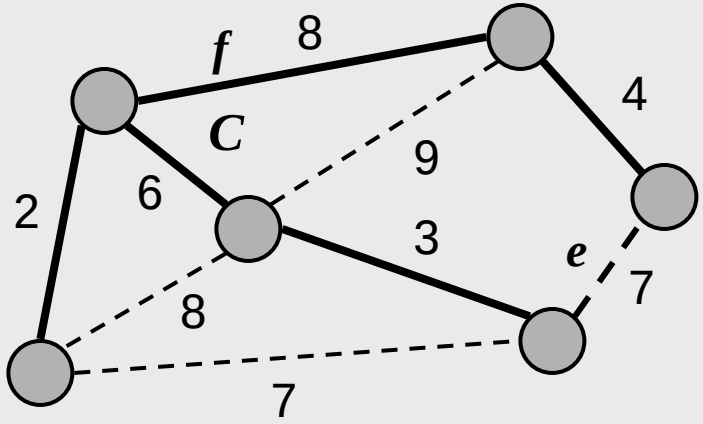
Cycle Property

For any spanning tree *T*, if an edge *e* that is not in *T* is added, a cycle will be created.

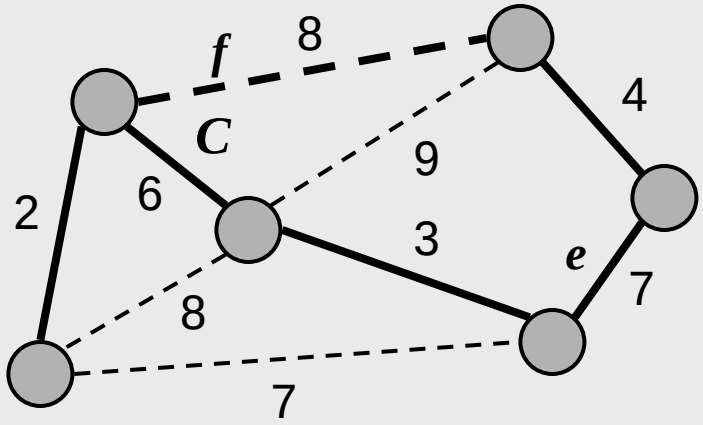
The removal of any edge on the cycle will reinstate the spanning tree property.

The cost of the spanning tree is lowered if *e* has a lower cost than the edge that was removed.

If, as a spanning tree is created, the edge that is added is the one with the minimum cost, the creation of the cycle will be avoided and the cost associated with the tree cannot be improved because any replacement edge would have an associated cost of at least as much as the edge already included in the spanning tree.

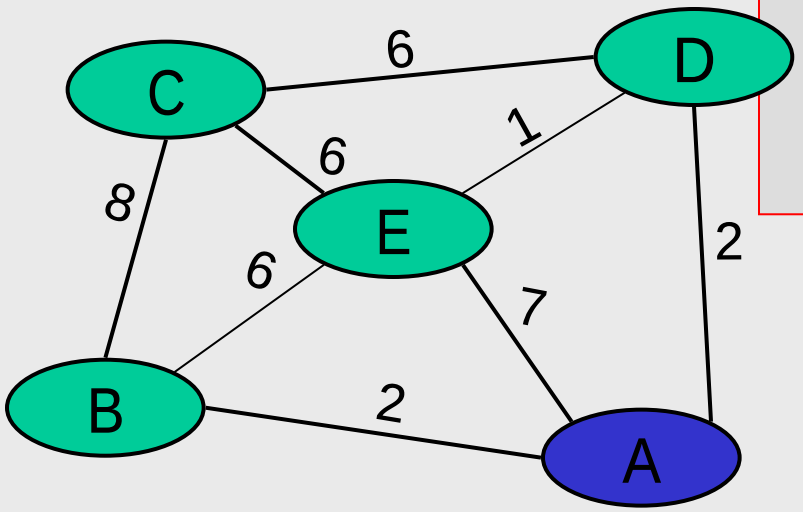


Replacing ***f*** with ***e*** yields a better spanning tree



Minimum Spanning Tree

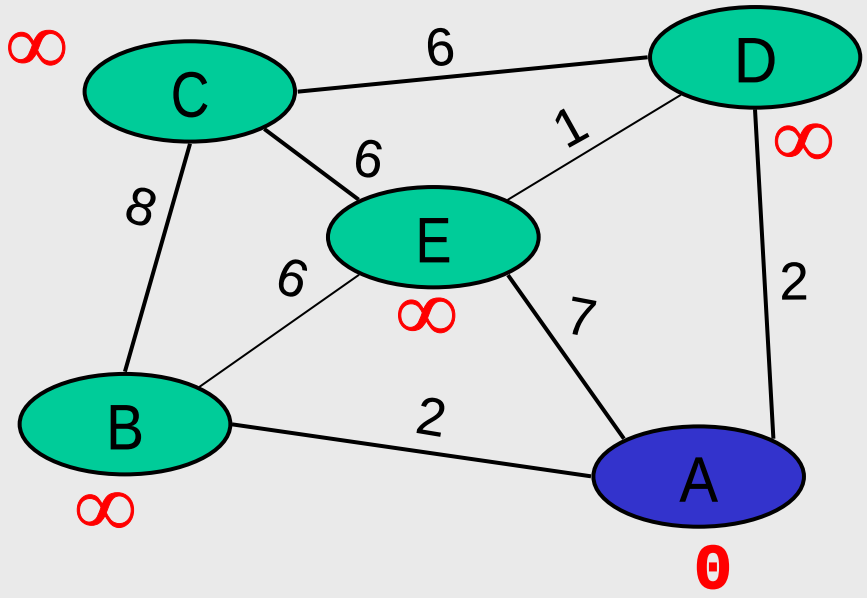
- **Prim's Algorithm (Prim-Jarnik Algorithm)**
  - **Label cost of each vertex as** ∞ (or 0 for the start vertex)
  - **Loop while there is a vertex**
    - **Remove a vertex that will extend the tree with minimum additional cost**
    - **Check and if required update the path length of its adjacent neighbors (Update rule different from Dijkstra's algorithm)**
  - **end loop**



Update rule:

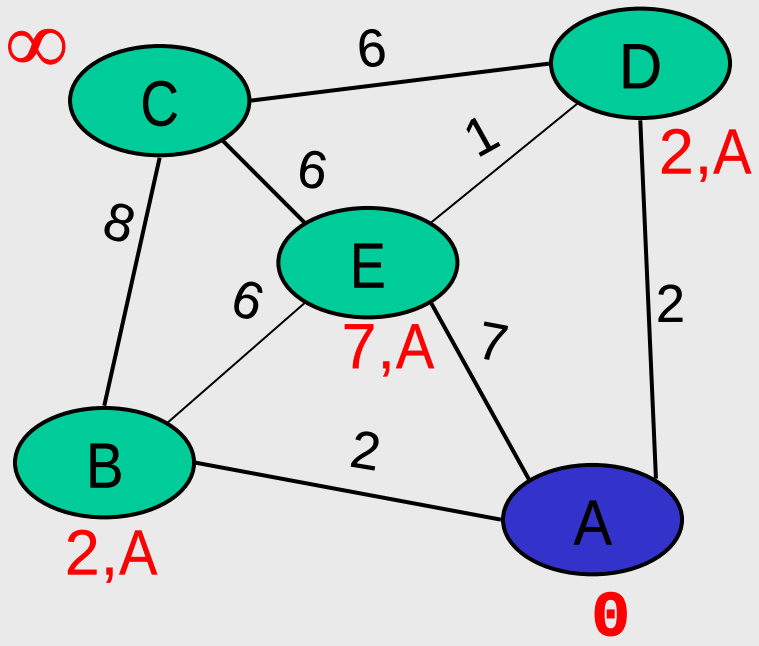
Let 'a' be the vertex removed and 'b' be its adjacent vertex
Let 'e' be the edge connecting a to b.
if (e.weight < b.cost)
    b.cost←e.weight
    b.parent ← a

MST: Prim-Jarnik's Algorithm



vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	F	∞	0
C	F	∞	0
D	F	∞	0
E	F	∞	0

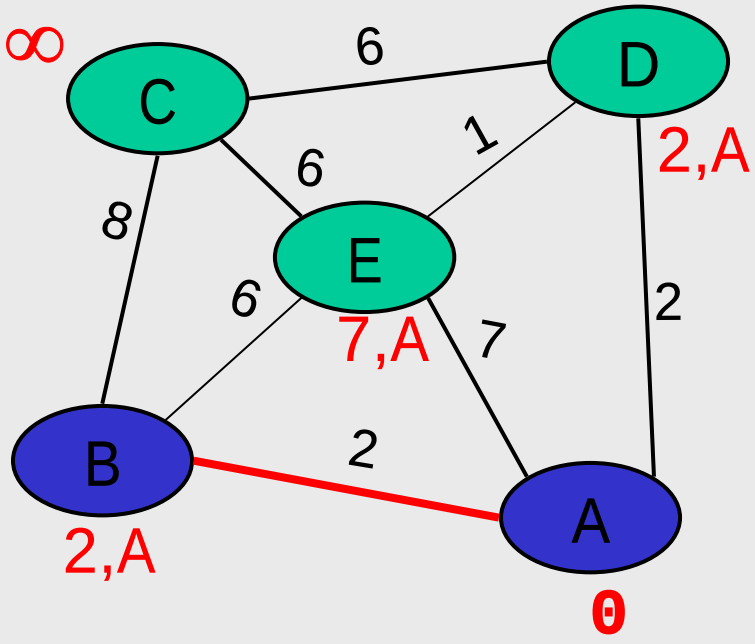
MST: Prim-Jarnik's Algorithm



vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	F	2	A
C	F	∞	0
D	F	2	A
E	F	7	A

Set distance to all vertices adjacent to vertex A.

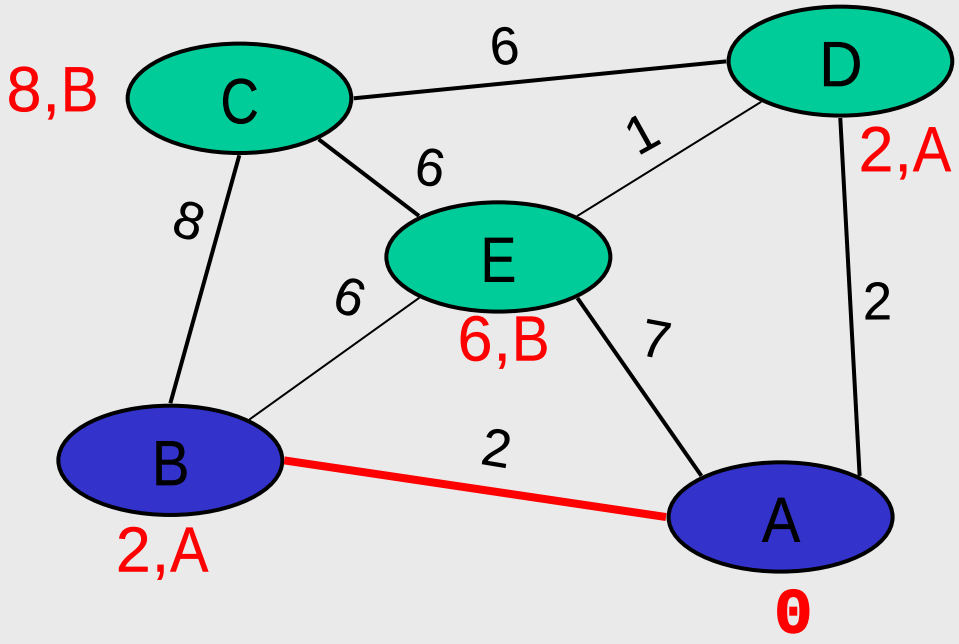
MST: Prim-Jarnik's Algorithm



vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	F	2	A
C	F	∞	0
D	F	2	A
E	F	7	A

Use greedy approach to select next vertex in MST – in this example either B or D could be chosen.

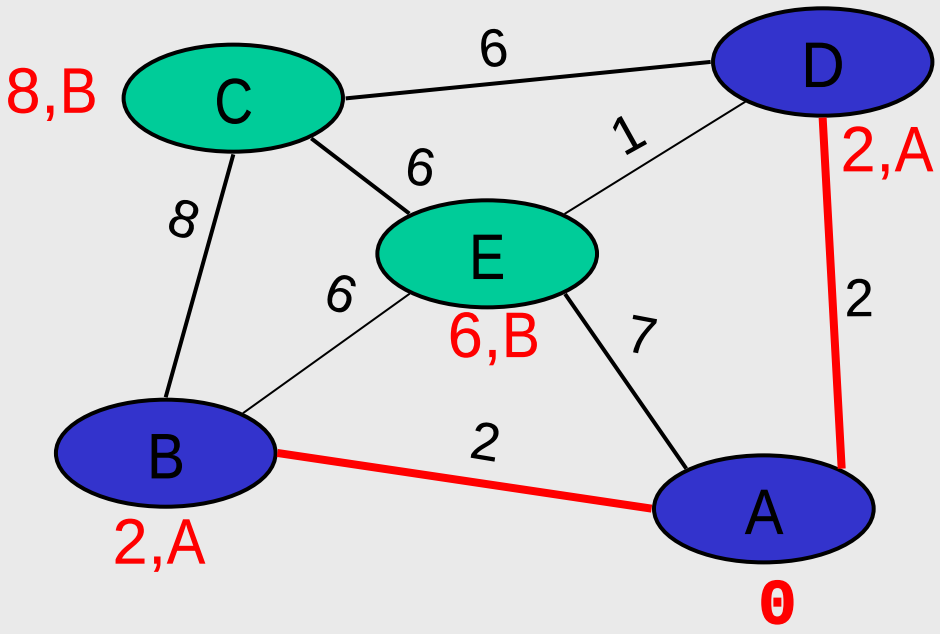
MST: Prim-Jarnik's Algorithm



vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	T	2	A
C	F	8	B
D	F	2	A
E	F	6	B

Set vertex B as visited and adjust distances to vertices adjacent to B.

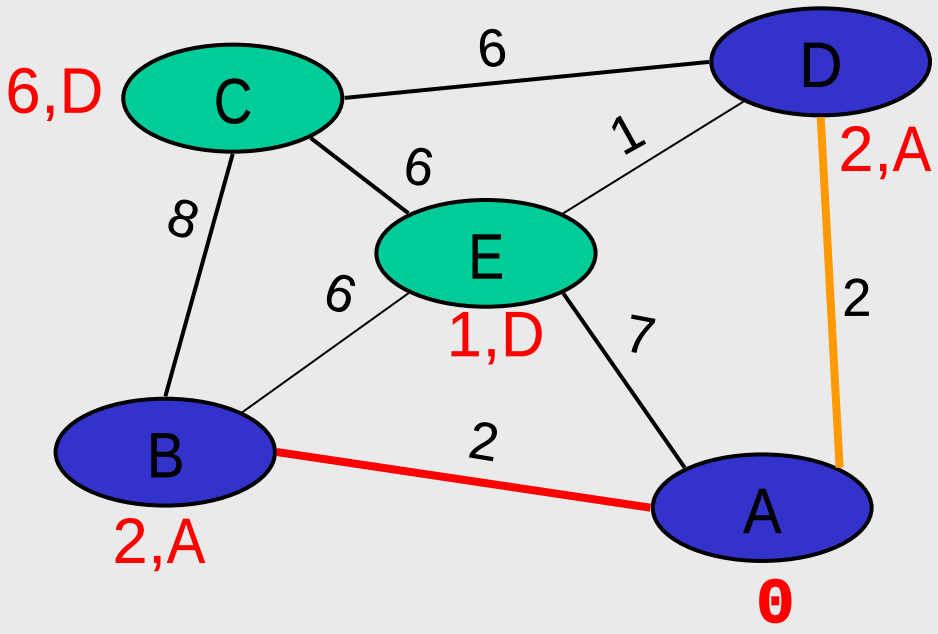
MST: Prim-Jarnik's Algorithm



Greed approach selects D as the next vertex – so mark as visited.

vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	T	2	A
C	F	8	B
D	T	2	A
E	F	6	B

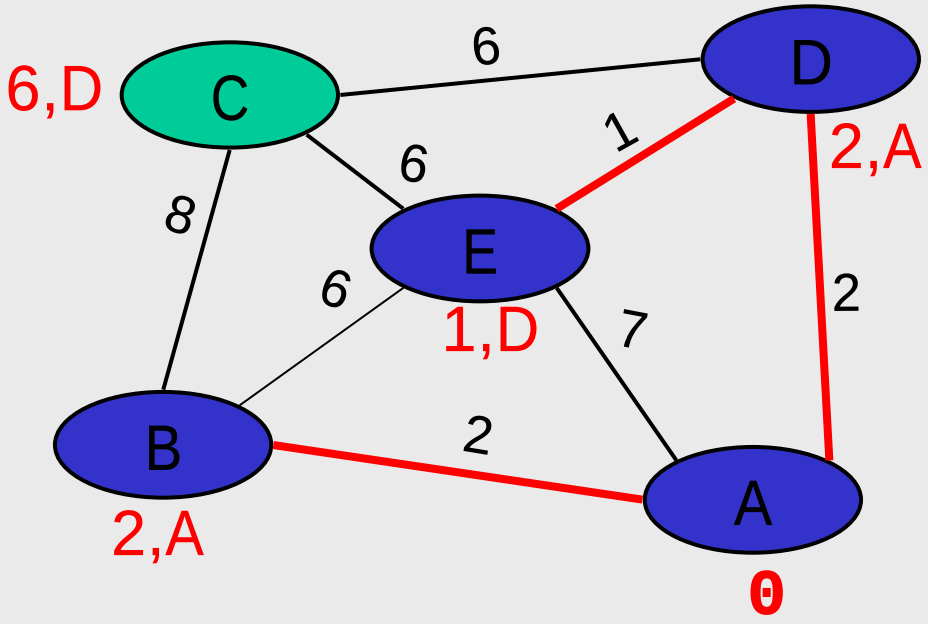
MST: Prim-Jarnik's Algorithm



vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	T	2	A
C	F	6	D
D	T	2	A
E	F	1	D

Reset distances to vertices adjacent to vertex D.

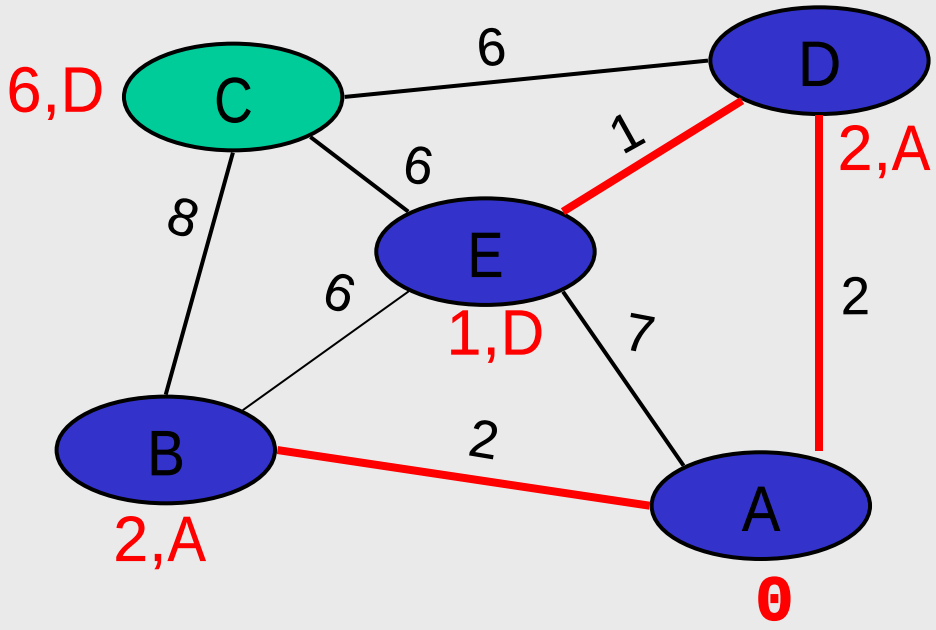
MST: Prim-Jarnik's Algorithm



Greedy approach selects vertex E next. Mark as visited in the table.

vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	T	2	A
C	F	6	D
D	T	2	A
E	T	1	D

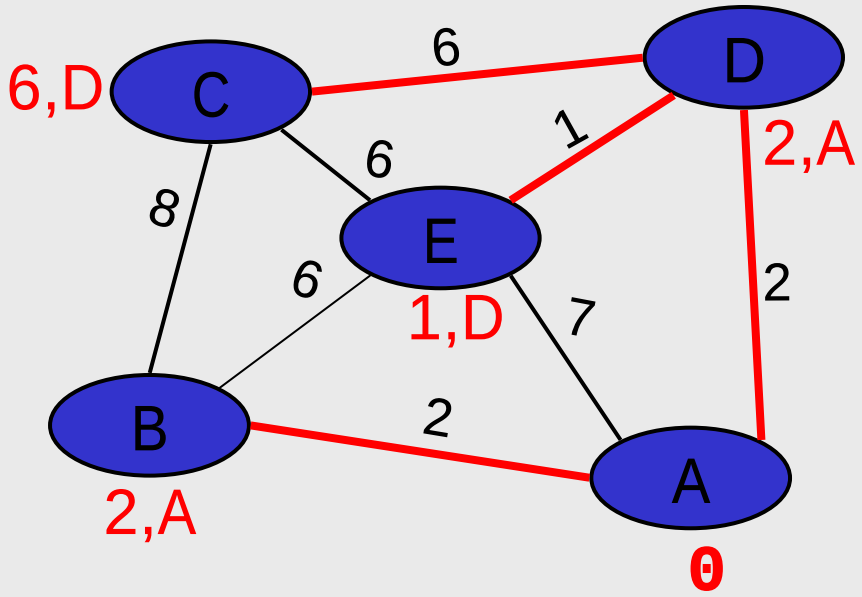
MST: Prim-Jarnik's Algorithm



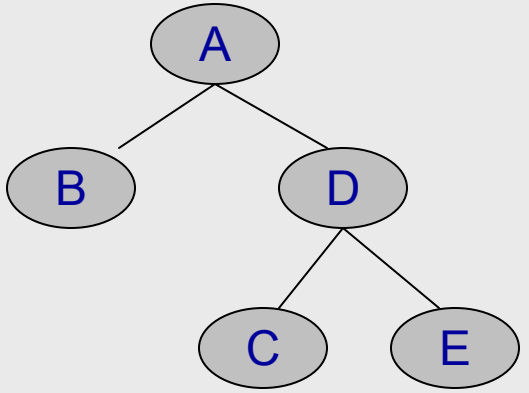
vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	T	2	A
C	F	6	D
D	T	2	A
E	T	1	D

In this case – no distances are decreased to vertices adjacent to E.

MST: Prim-Jarnik's Algorithm



vertex	visited	Minimum weight	vertex causing change to min weight
A	T	0	0
B	T	2	A
C	T	6	D
D	T	2	A
E	T	1	D

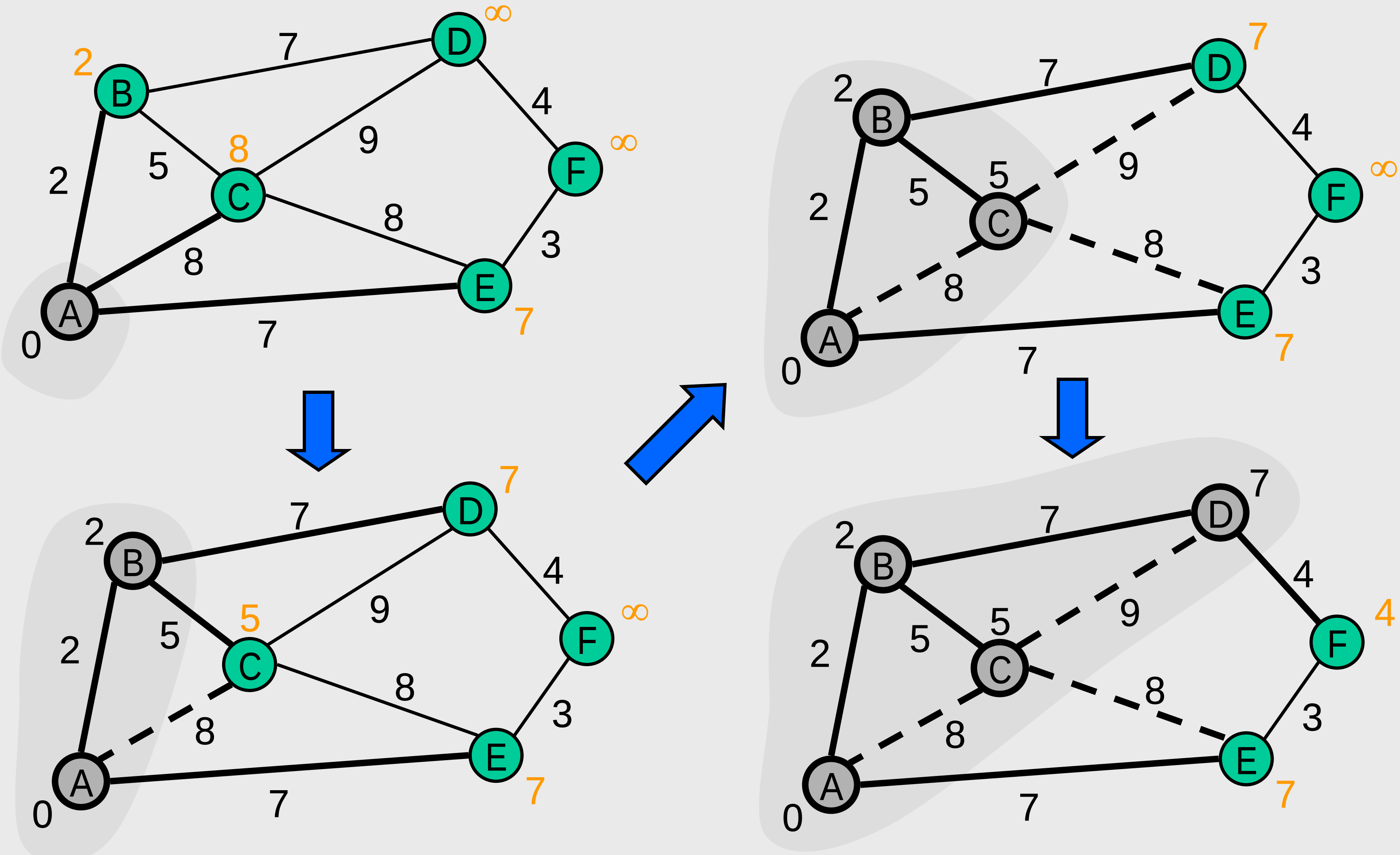


Final table has all vertices visited and minimum edge weights identified. The MST is also identified in the table. A is set as the root of the MST, B and D are children of A while C and E are children of D in the MST.

Prim-Jarnik Algorithm

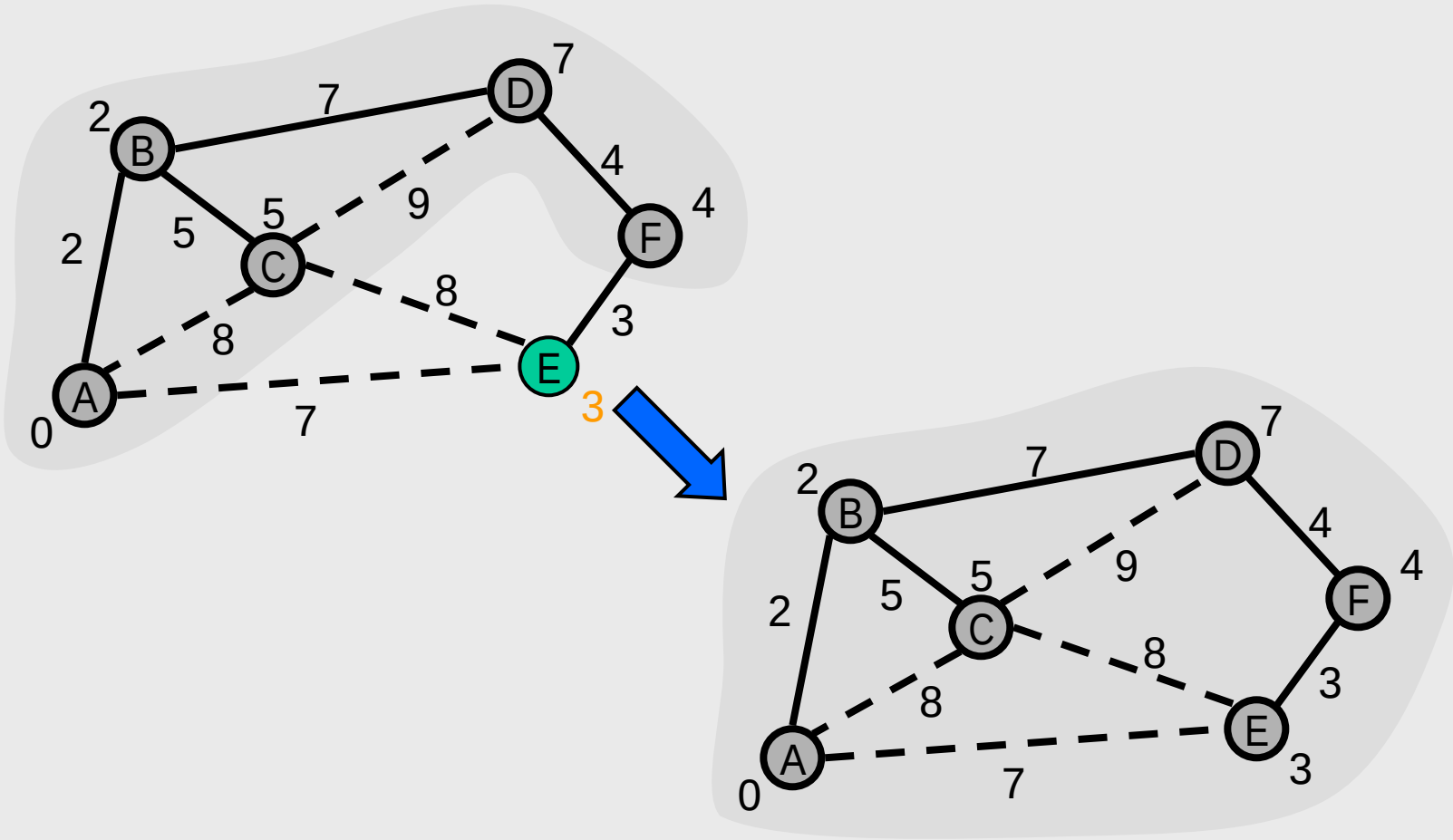
```
Algorithm MST (G)
   Q ← new priority queue
   Let s be any vertex
   for all  v ∈ G.vertices()                    O(n)
      if v = s
         v.cost ← 0
      else v.cost ← ∞
         v. parent ← null
      Q.enQueue(v.cost, v)                      O(n log n)
   while  ¬Q.isEmpty()
      v ← Q .removeMin()
       v.pathKnown ← true
      for all  e ∈ G.incidentEdges(v)           O((n+m) log n)
            w ← opposite(v,e)
            if ¬w.pathKnown
               if  weight(e) < w.cost
                  w.cost ← weight(e)
                  w. parent ← v
                  update key of w in Q
```


Example



2
B
7
D
∞
4
2
5
8
C
9
F
∞
8
3
8
A
E
7
0
7
2
B
7
D
7
5
C
8
9
4
2
8
F
∞
3
A
E
7
0
7
2
B
7
D
7
4
2
5
5
C
9
F
∞
8
8
3
A
E
7
0
7
2
B
7
D
7
4
2
5
5
C
9
F
4
8
8
3
A
E
7
0
7

Example (cont.)



Partition Property

Partition Property:

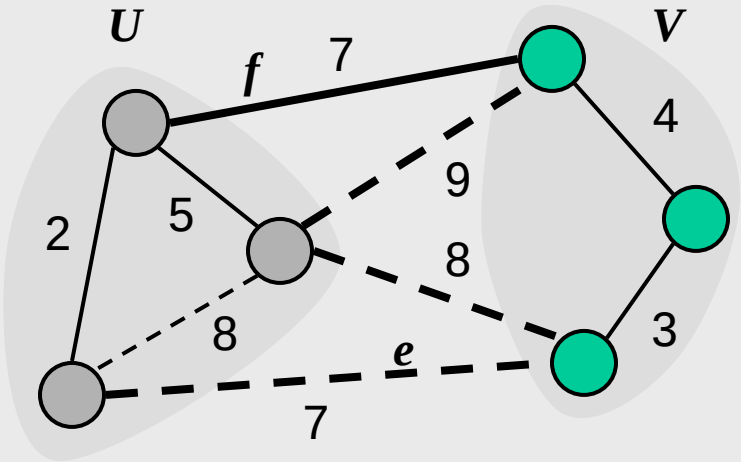
- Consider a partition of the vertices of ***G*** into subsets ***U*** and ***V***
- Let ***e*** be an edge of minimum weight across the partition
- There is a minimum spanning tree of ***G*** containing edge ***e***

Proof:

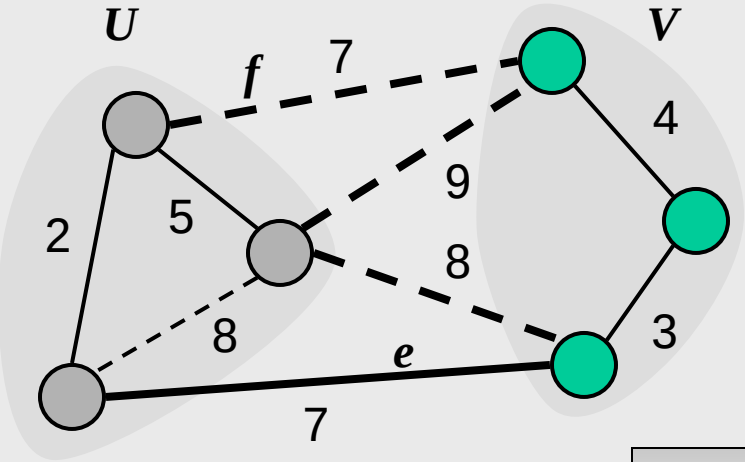
- Let ***T*** be an MST of ***G***
- If ***T*** does not contain ***e,*** consider the cycle ***C*** formed by ***e*** with ***T*** and let ***f*** be an edge of ***C*** across the partition
- By the cycle property,

$$weight(f) \leq weight(e)$$

- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing ***f*** with ***e***

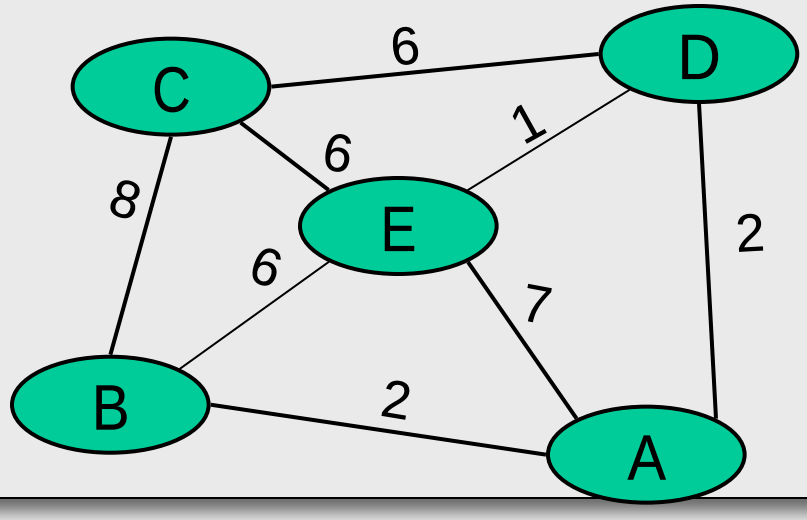


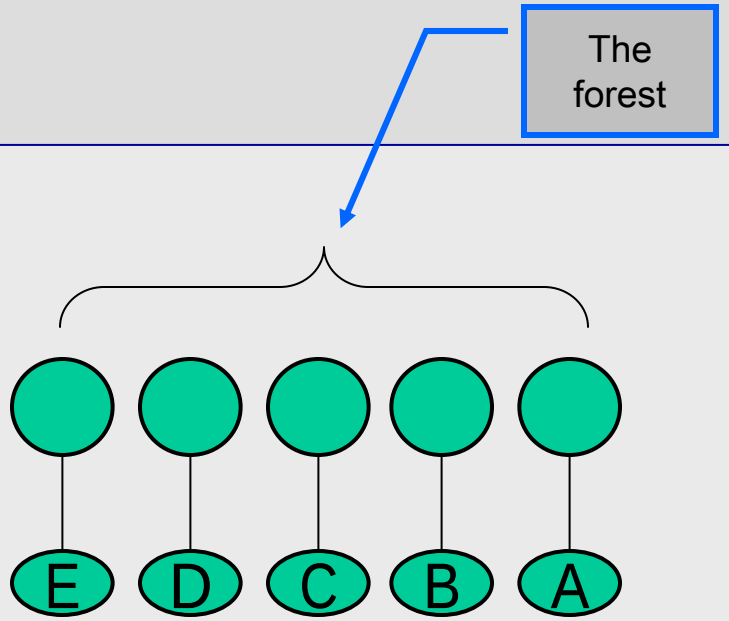
Replacing ***f*** with ***e*** yields another MST



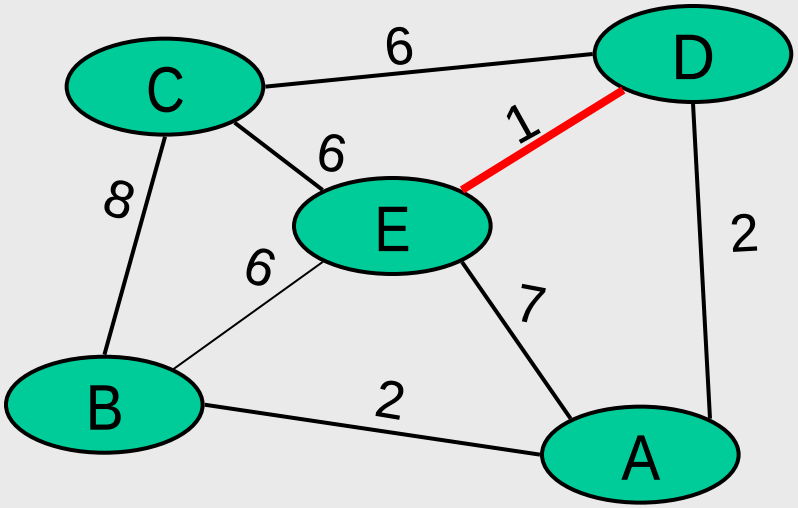
Minimum Spanning Tree

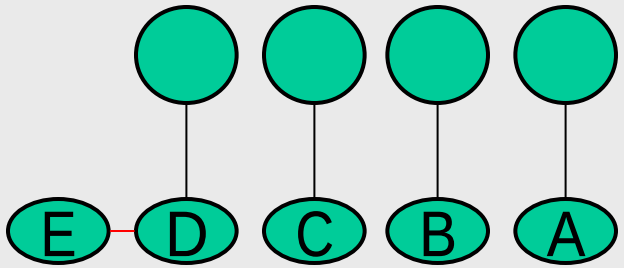
- **Kruskal's Algorithm**
  - **Create a forest of n trees**
  - **Loop while (there is > 1 tree in the forest)**
    - **Remove an edge with minimum weight**
    - **Accept the edge only if it connects 2 trees from the forest in to one.**
  - **end loop**



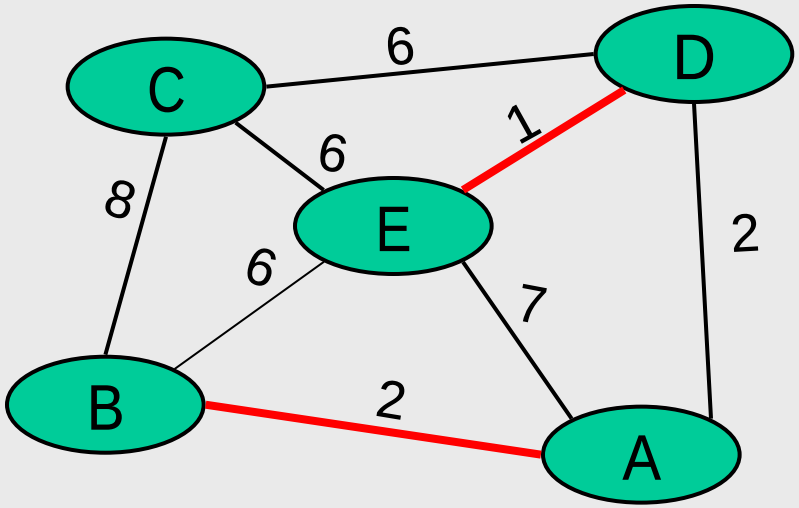


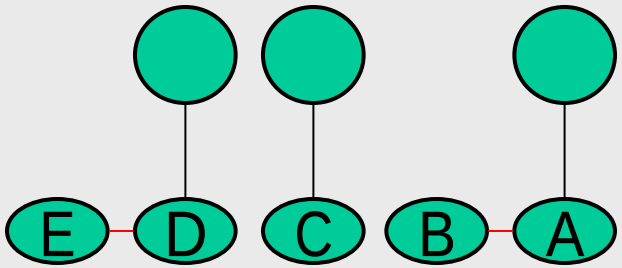
MST: Kruskal's Algorithm



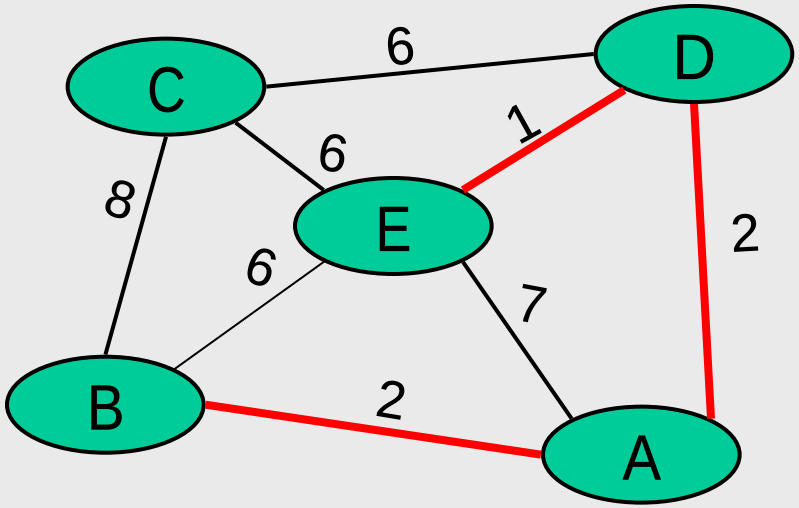


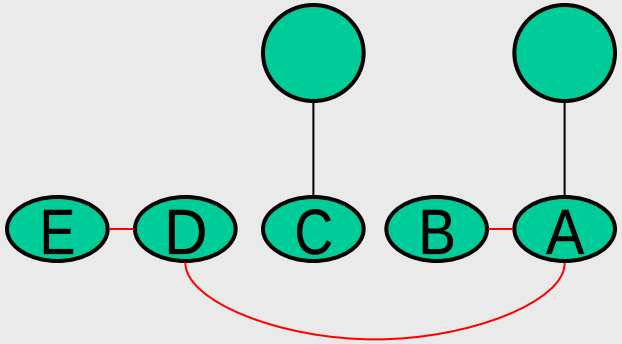
MST: Kruskal's Algorithm



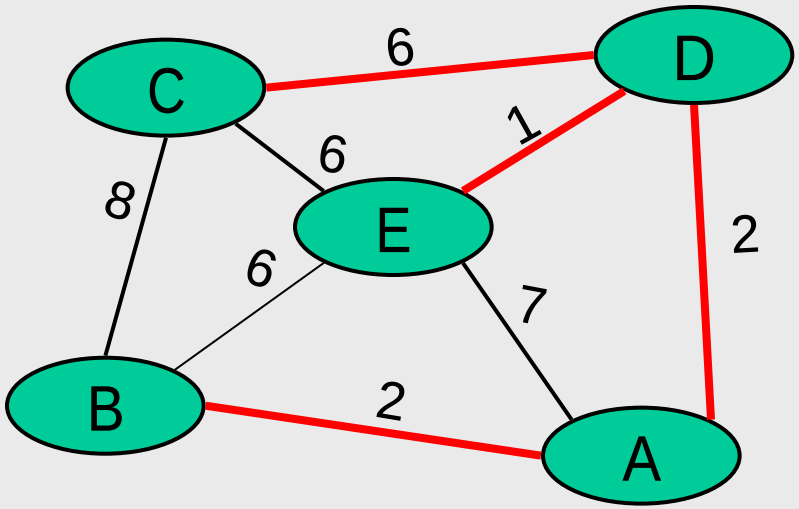


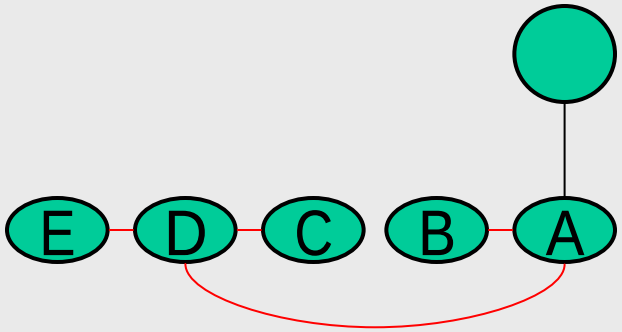
MST: Kruskal's Algorithm

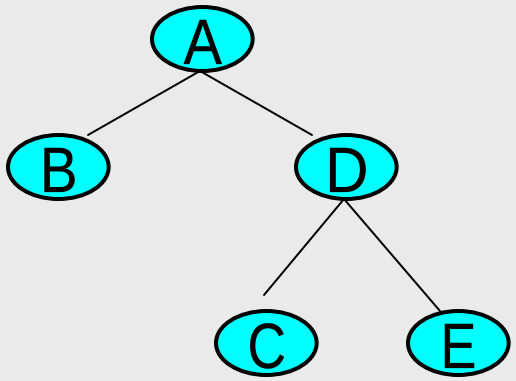




MST: Kruskal's Algorithm



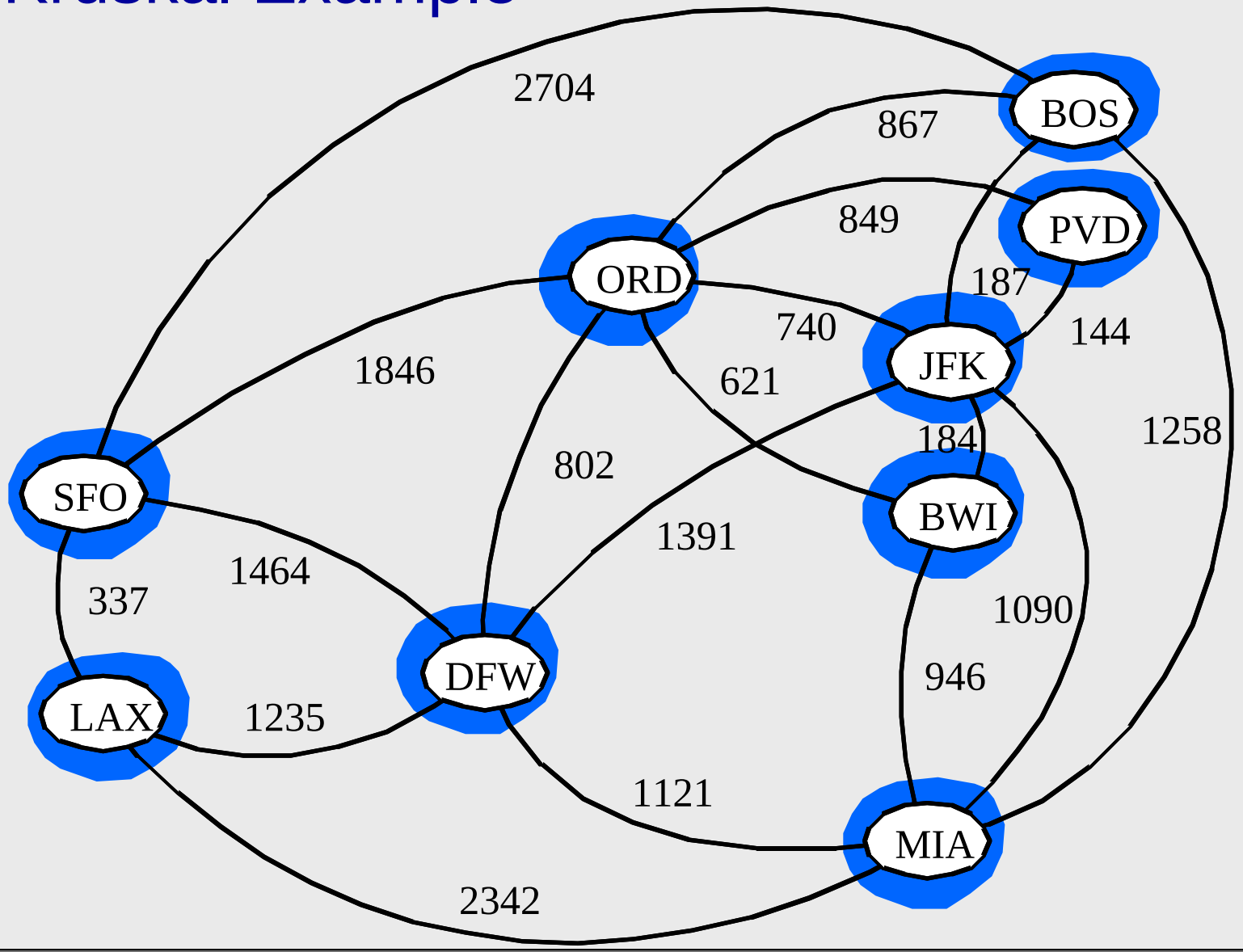




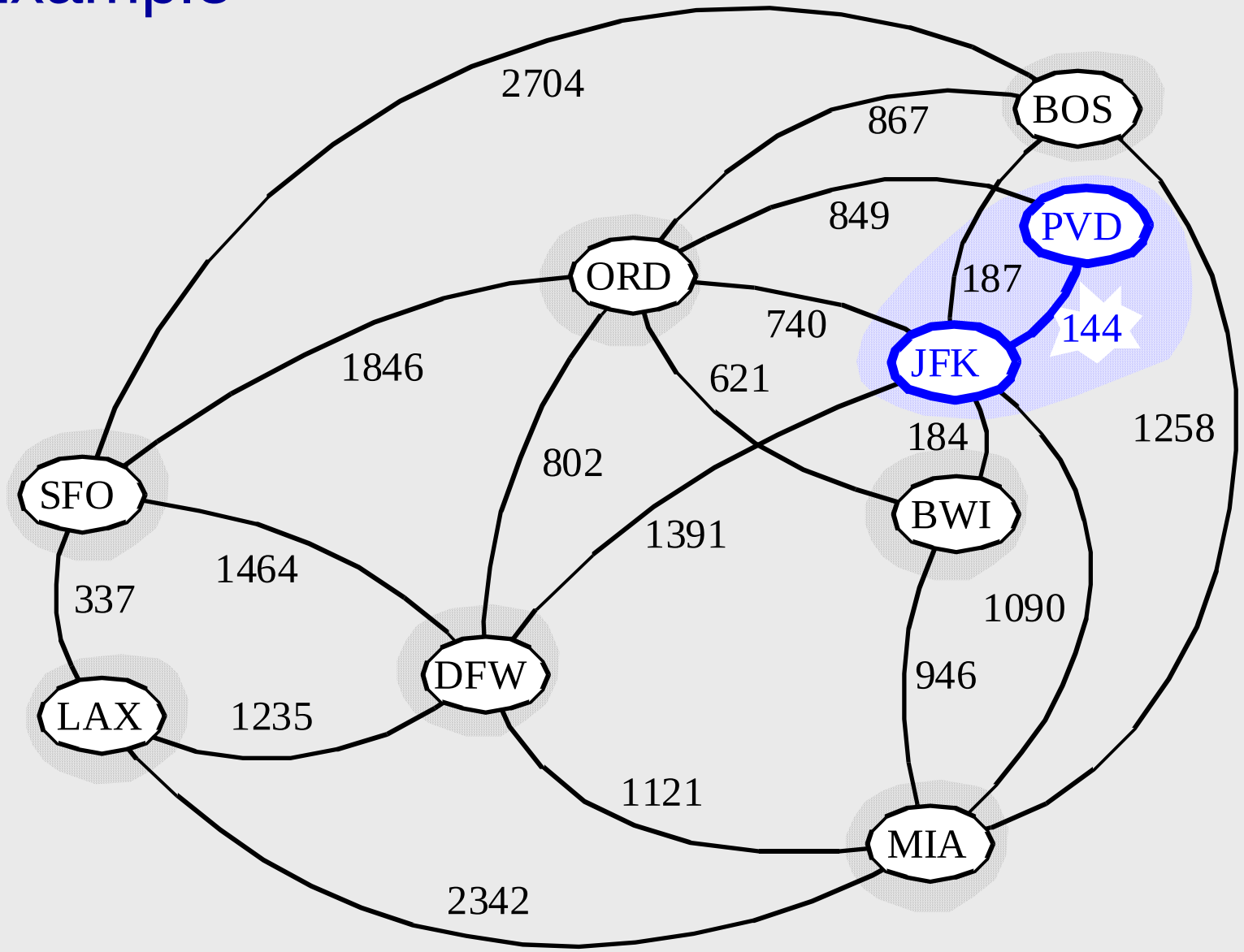
Kruskal's Algorithm

```
Algorithm KruskalMST(G)
   let Q be a priority queue.
   Insert all edges into Q using their
         weights as the key
   Create a forest of n trees                          O(m log m)
        where each vertex is a tree
   numberOfTrees ← n
   while numberOfTrees > 1 do
     edge e ← Q.removeMin()
     Let u, v be the endpoints of e
     if Tree(v) ≠ Tree(u) then
        Combine Tree(v) and Tree(u)                    O(m log m)
              using edge e
        decrement numberOfTrees
   return T
```

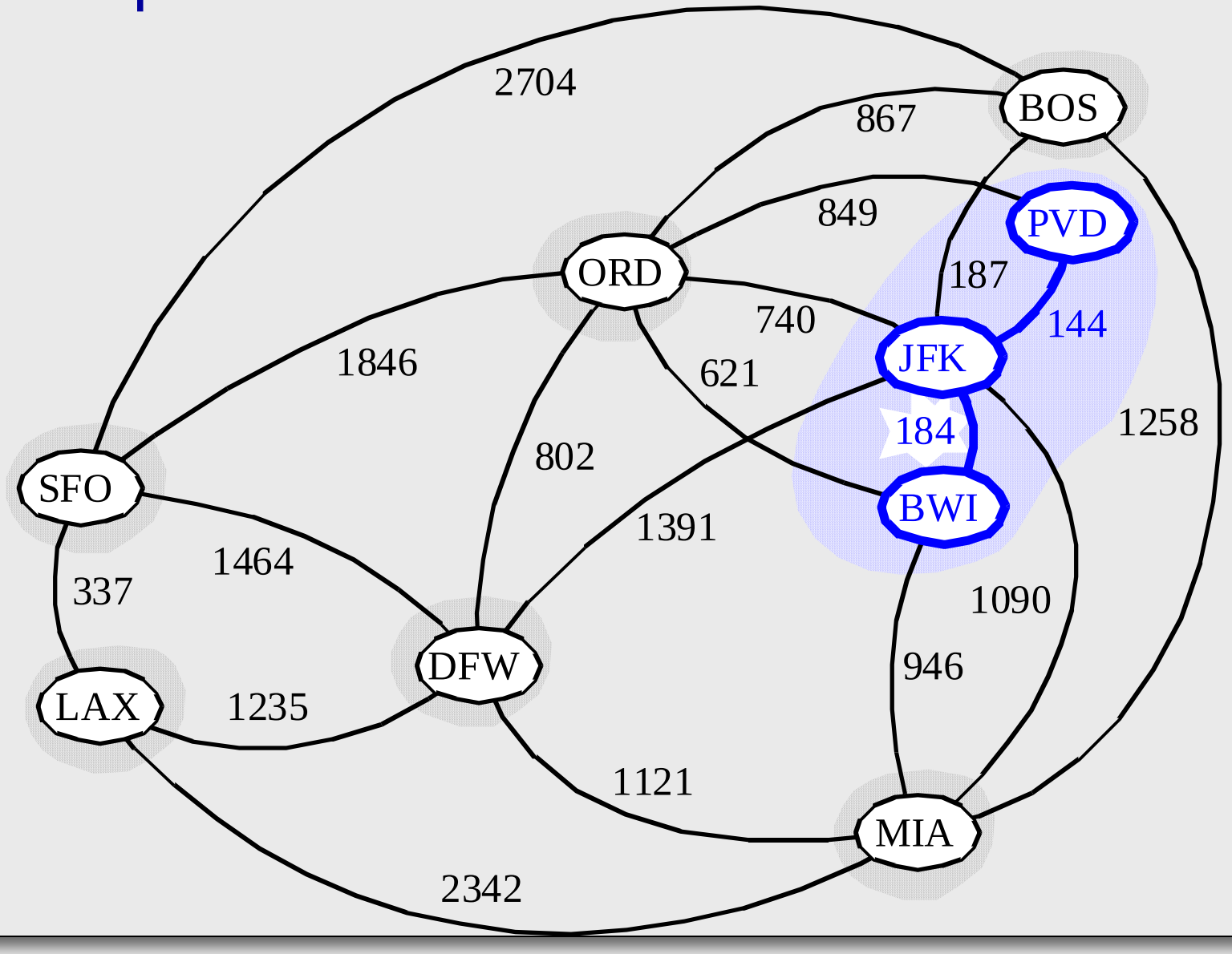

Kruskal Example



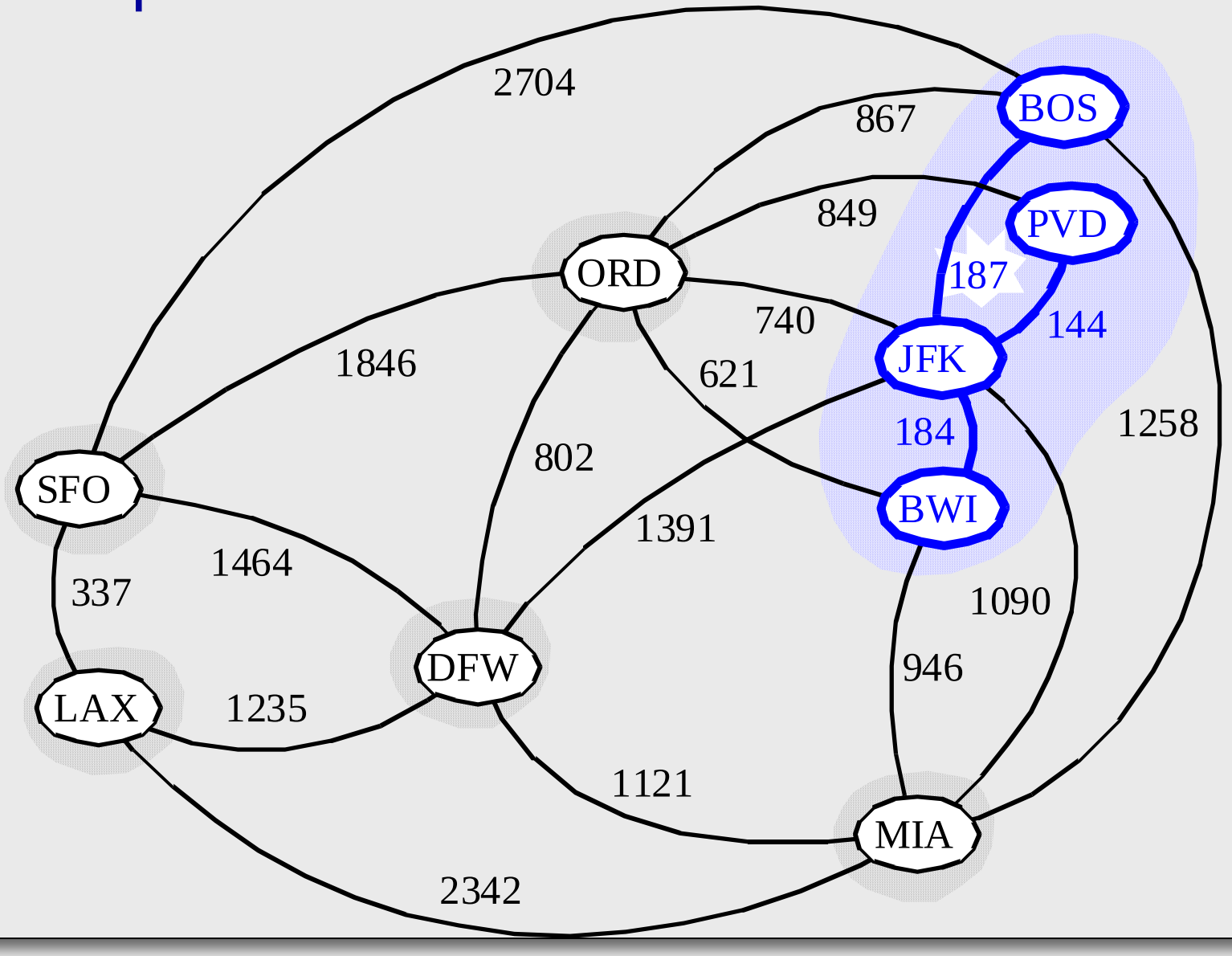
Example


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

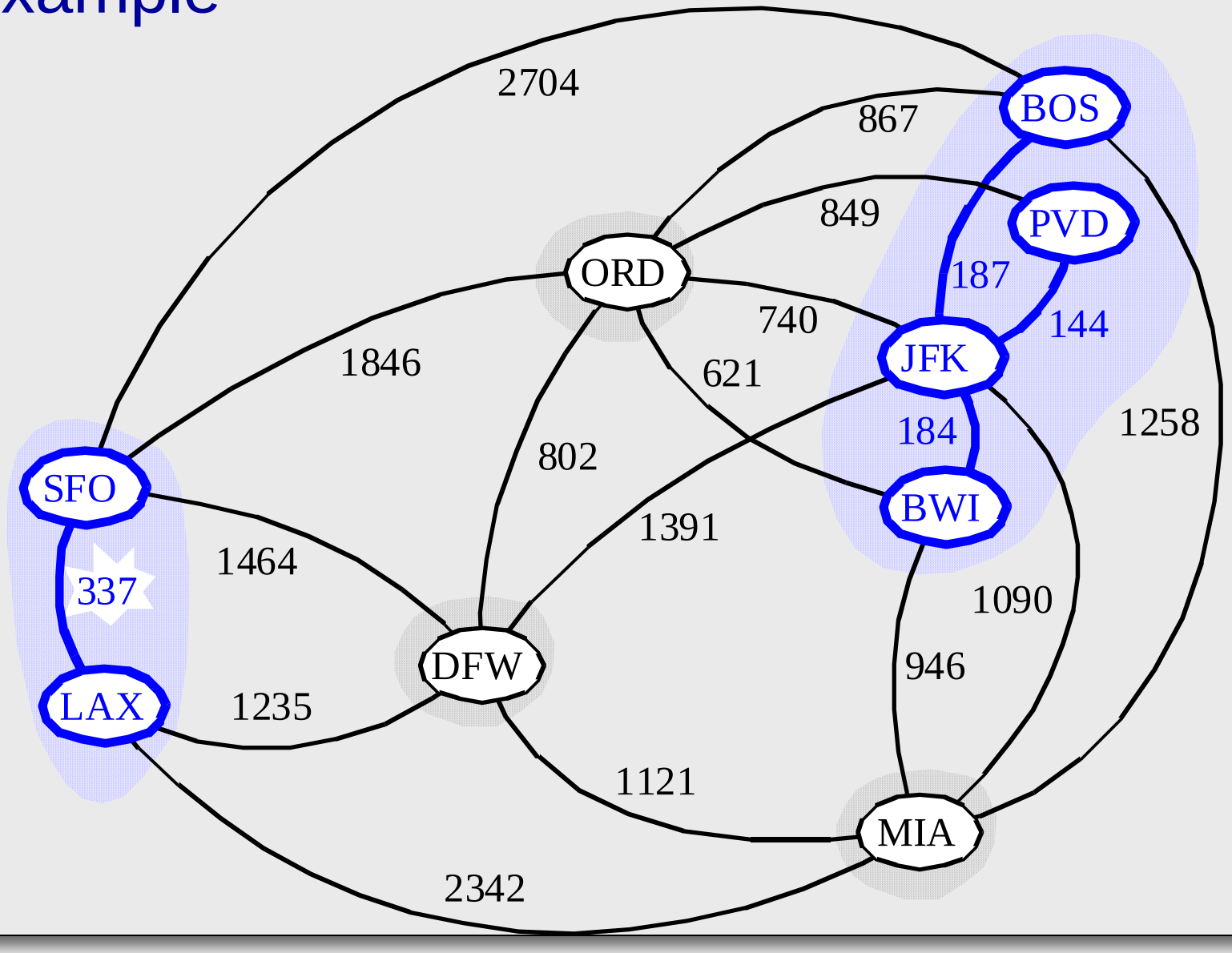
Example


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

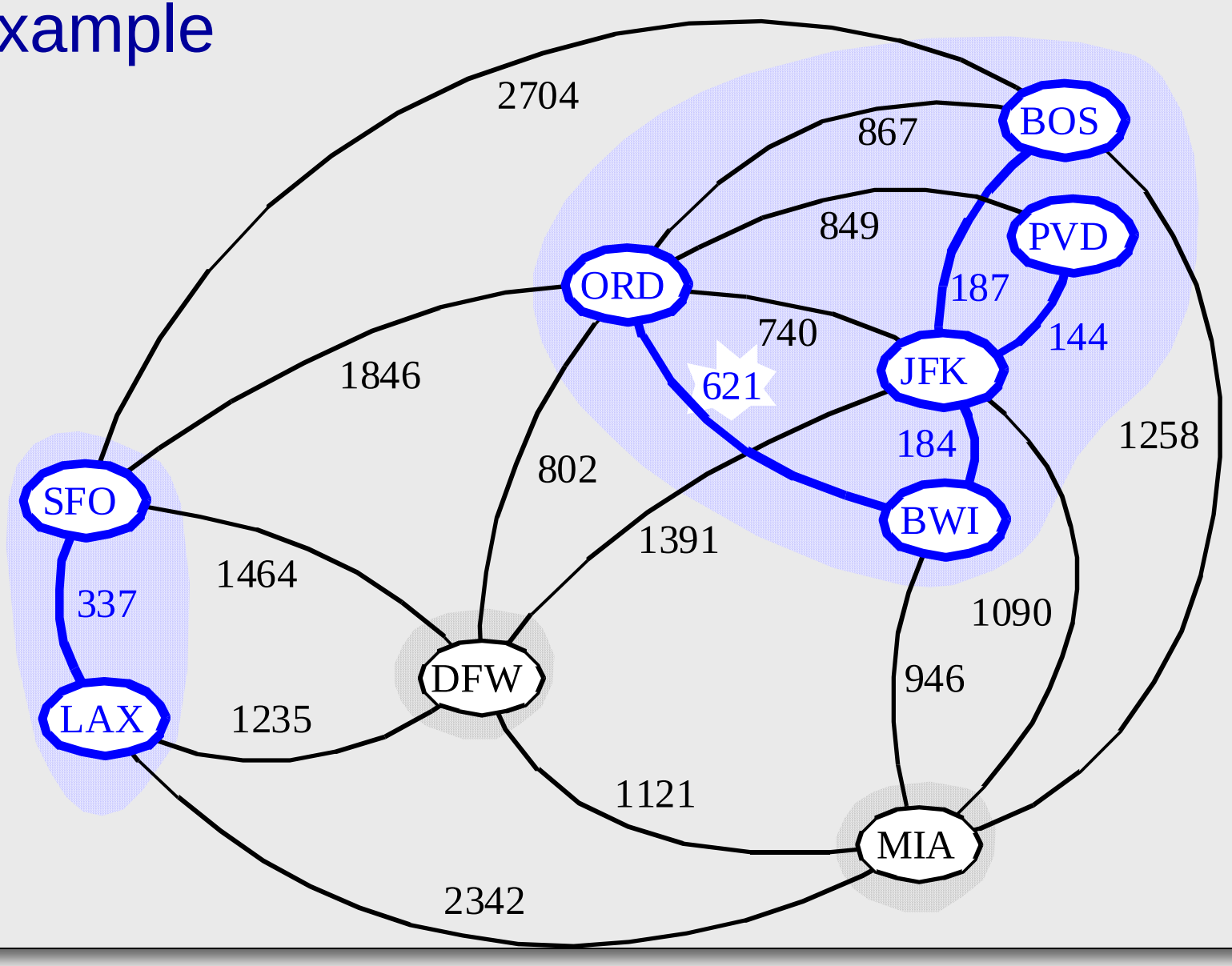
Example


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

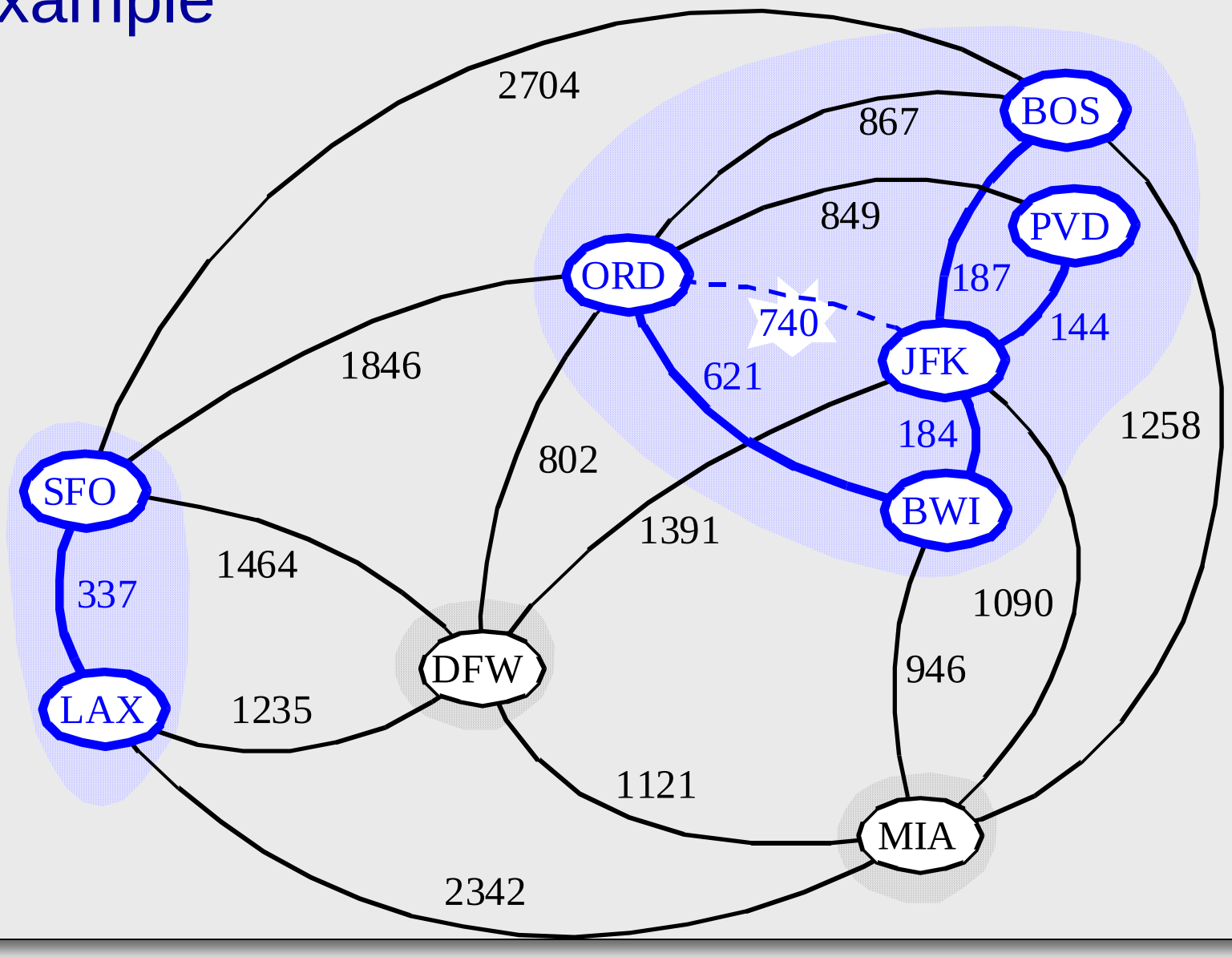
Example


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

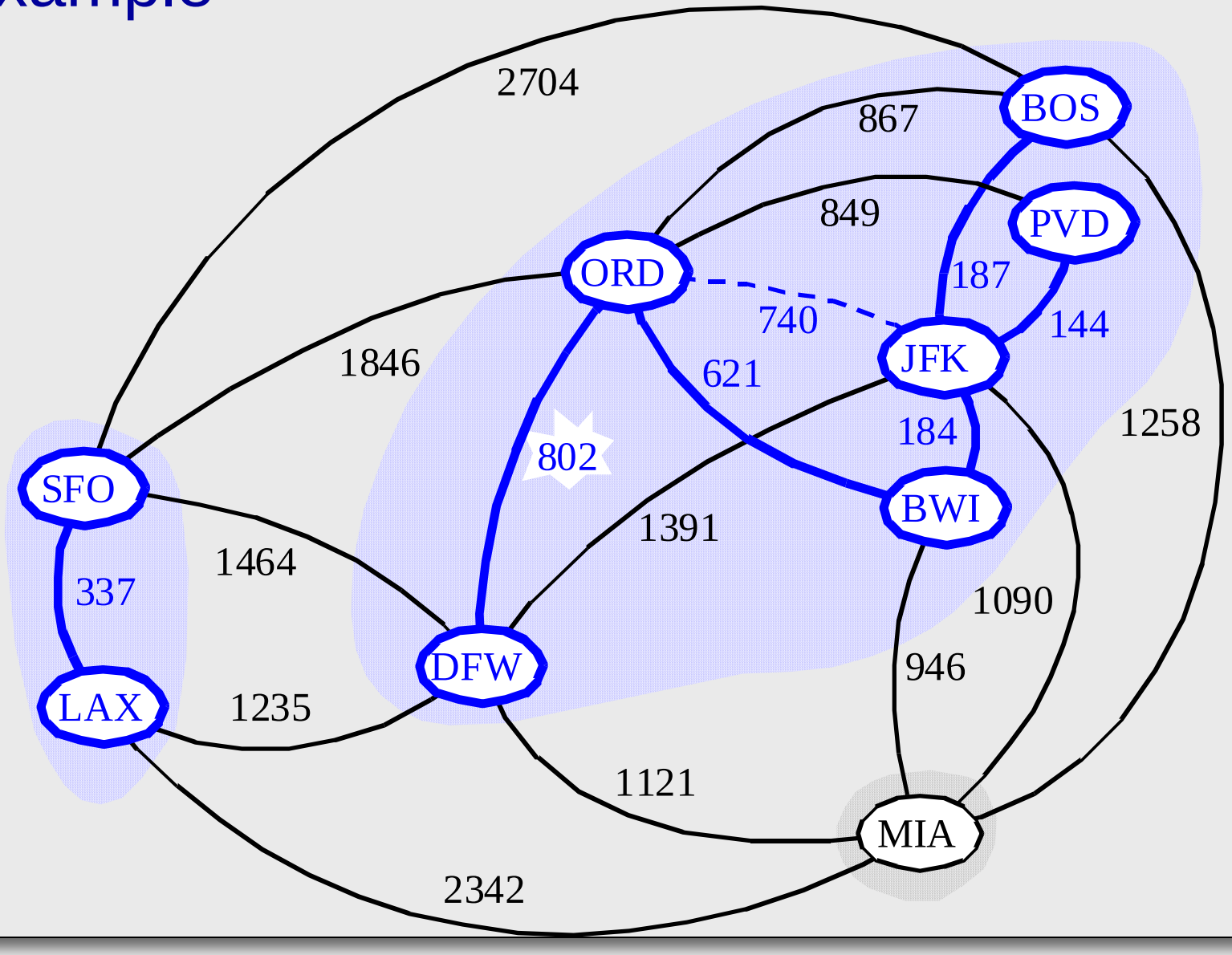
Example


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

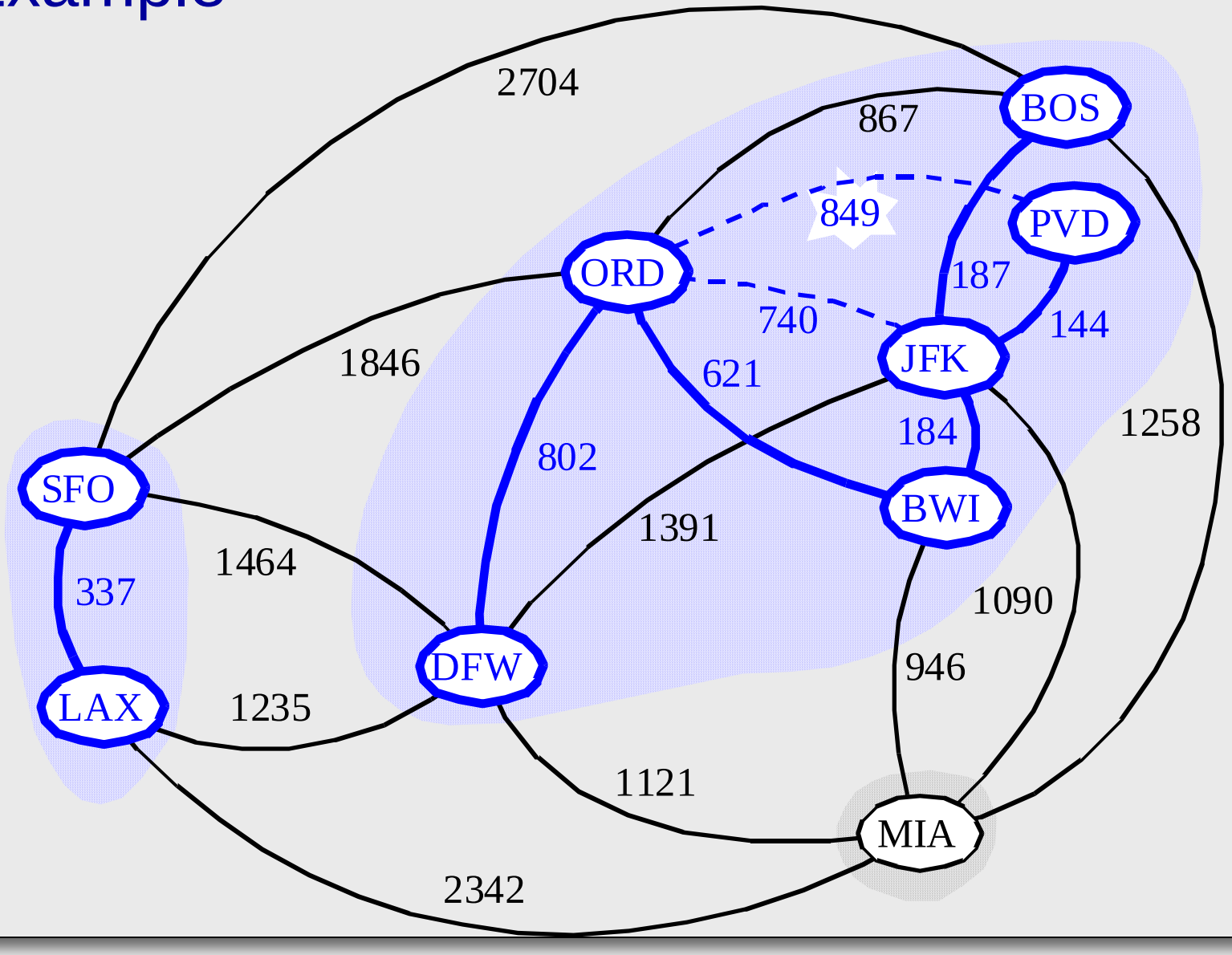
Example


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

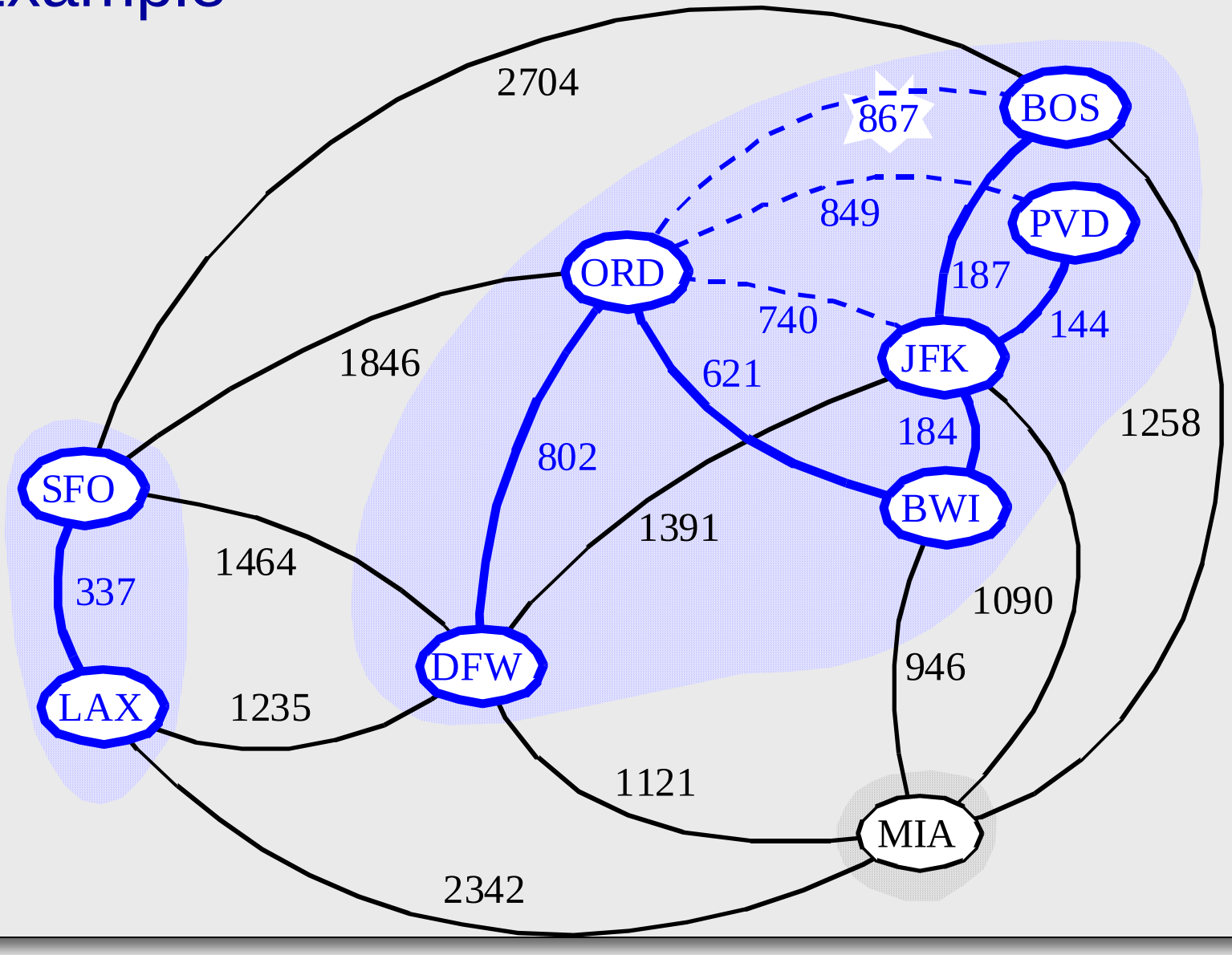
Example


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

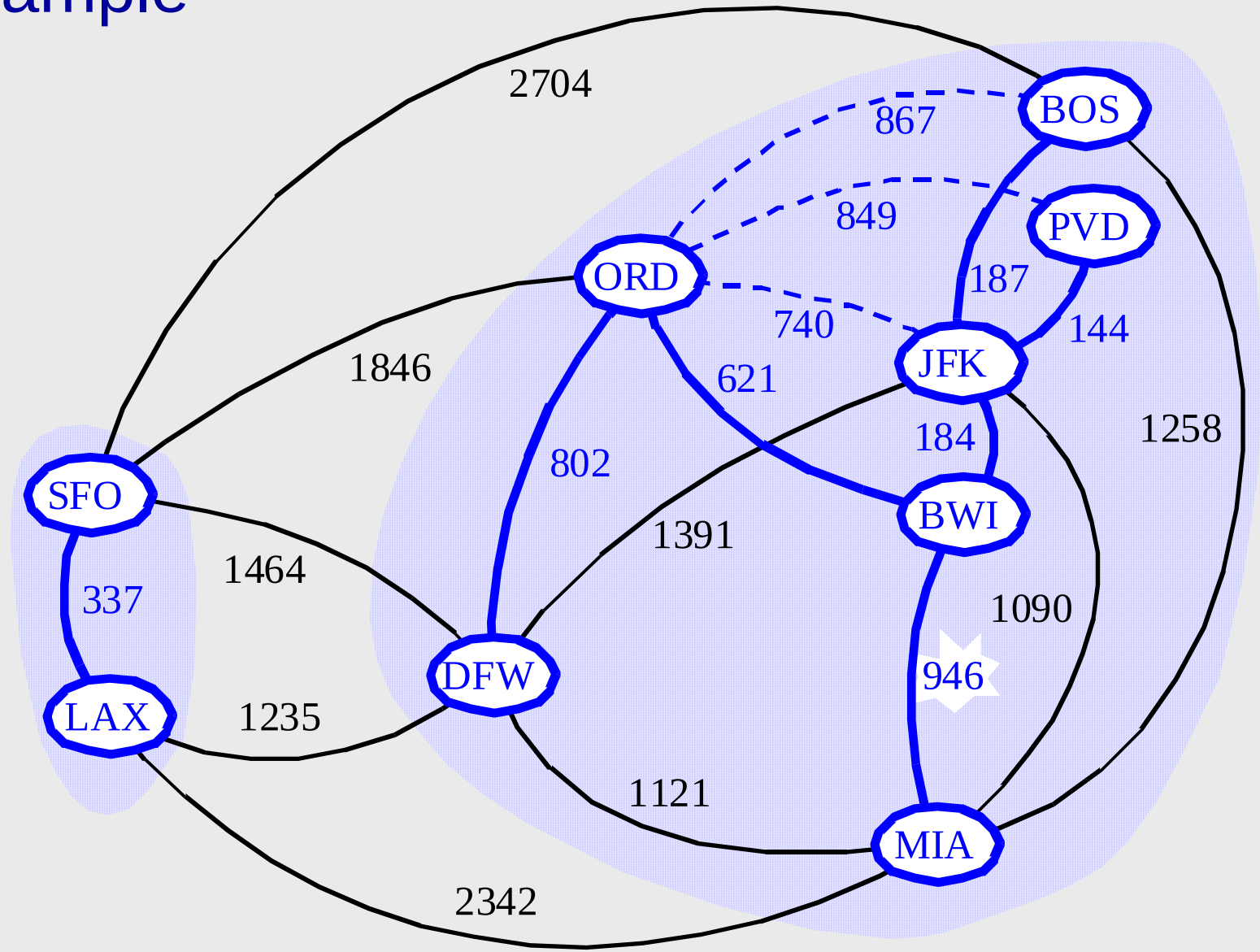
Example


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

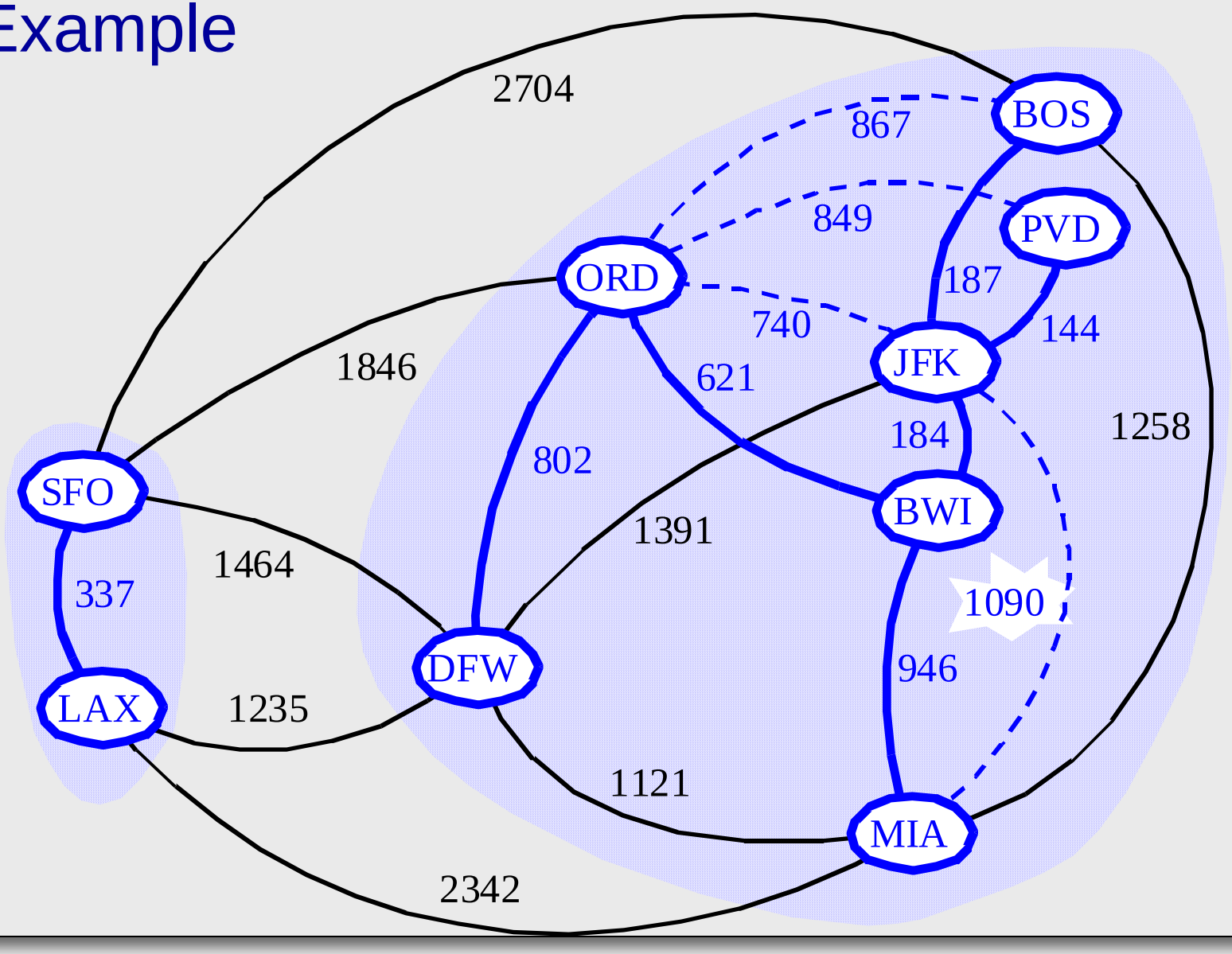
Example


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

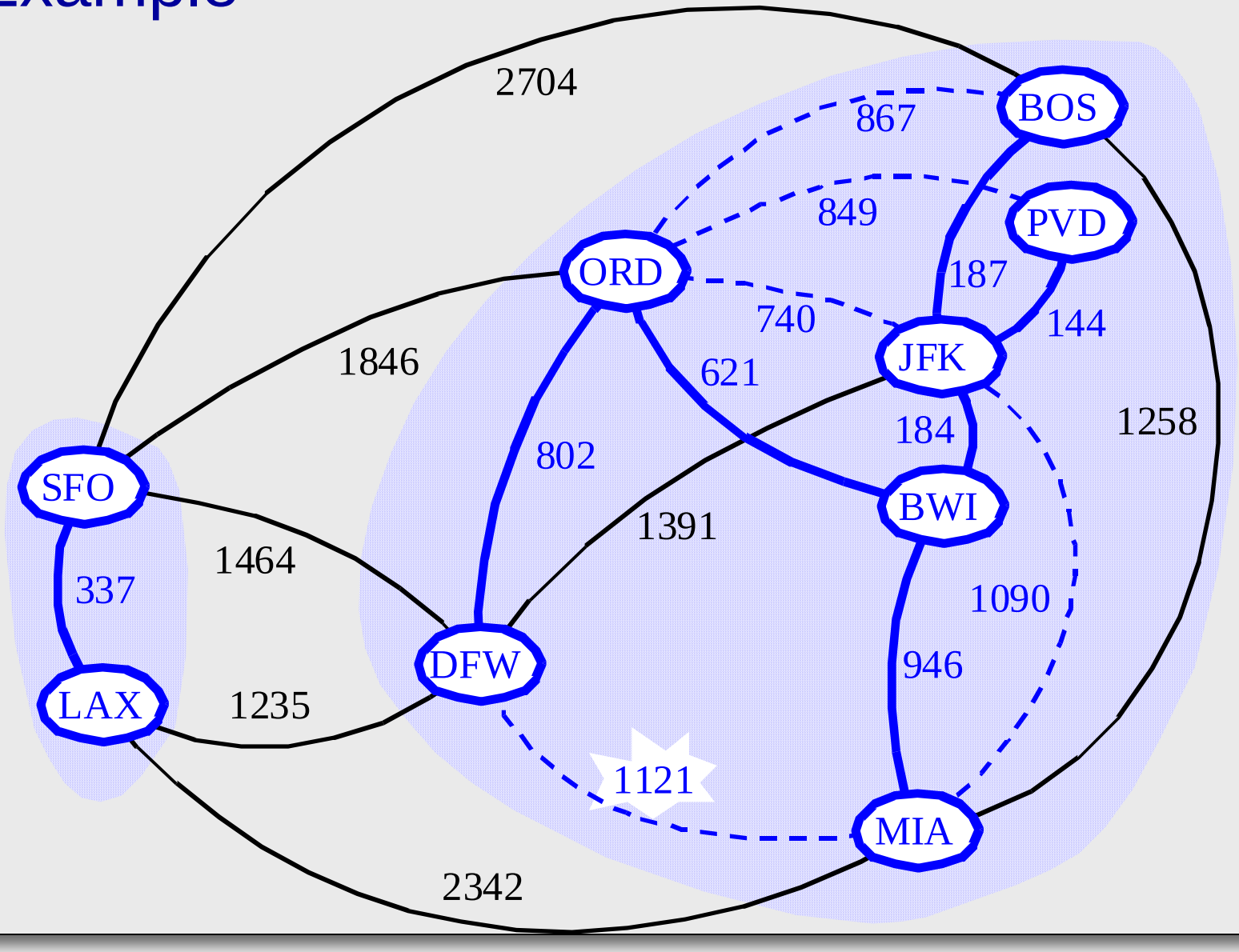
Example


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

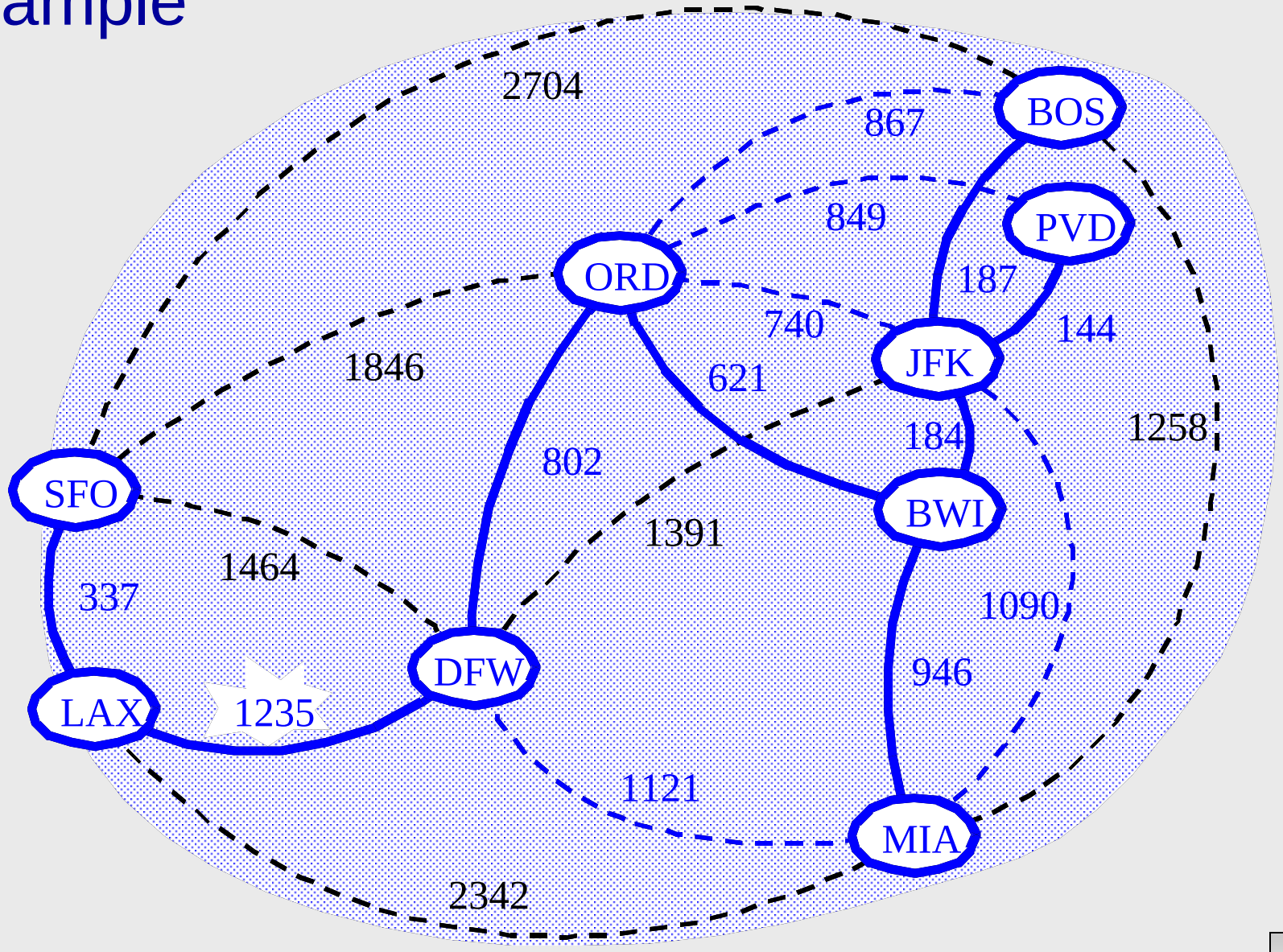
Example


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

Example

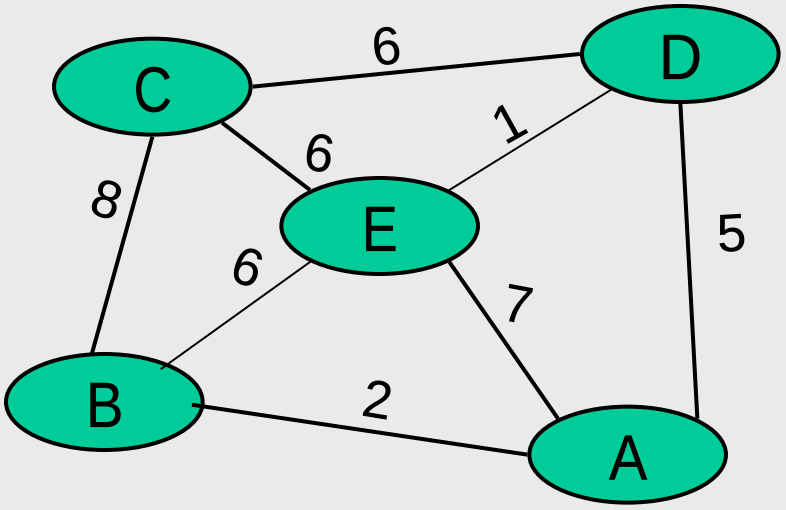

2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

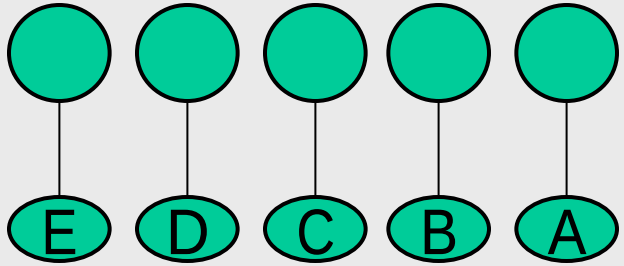
Example


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

Minimum Spanning Tree

- **Baruvka's Algorithm**
  - **Create a forest of n trees**
  - **Loop while (there is > 1 tree in the forest)**
    - **For each tree T_i in the forest**
      - **Find the smallest edge e = (u,v), in the edge list with u in T_i and v in $T_j \neq T_i$**
      - **connects 2 trees from the forest in to one.**
  - **end loop**





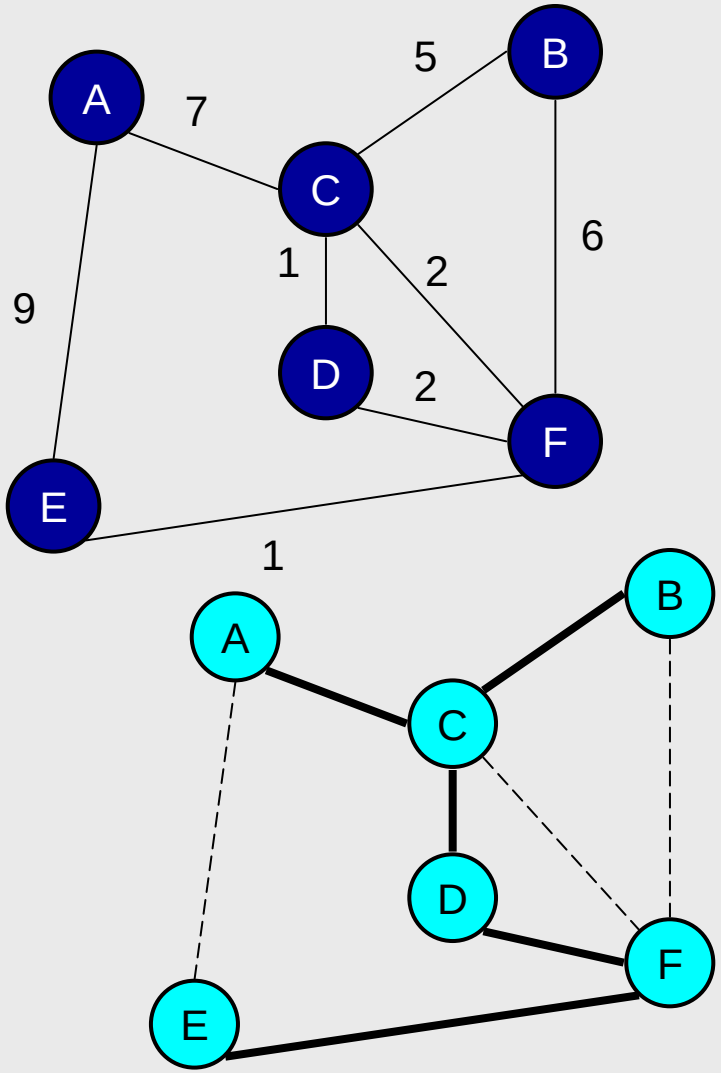
Baruvka's Algorithm

- 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's MST Algorithm - Example



A: edges a-c = 7, a-e = 9

select edge a-c, since c not in tree

B: edges b-c = 5, b-f = 6

select edge b-c, since c not in tree

C: edges c-a = 7, c-b = 5, c-d = 1, c-f = 2

can't select c-a since a is in tree
can't select c-b since b is in tree
select c-d since d not in tree

D: edges d-c = 1, d-f = 2

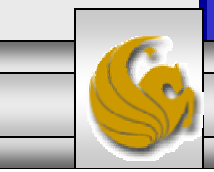
can't select d-c since c is in tree
select d-f since f not in tree

E: edges e-a = 9, e-f = 1

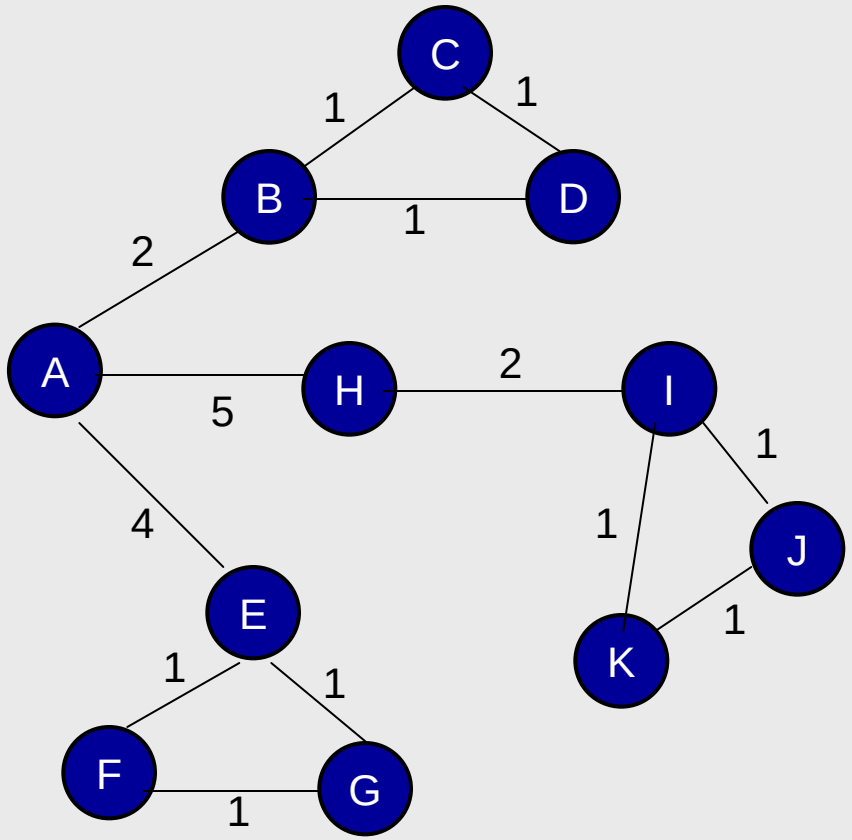
select e-f since f not in tree

F: edges f-b = 6, f-c = 2, f-d = 2, f-e = 1

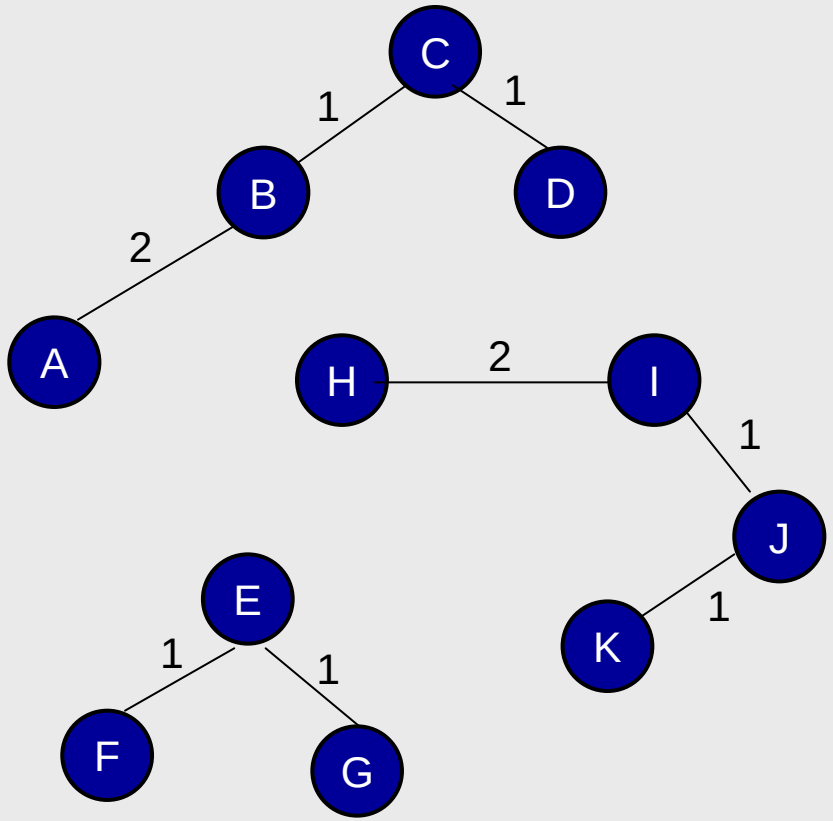
f already included in the tree



Baruvka's MST Algorithm – Example 2



Baruvka's MST Algorithm – Example 2



PASS #1

A: edge a-b = 2

B: edge b-c = 1

C: edge c-d = 1

D: none

E: edge e-f = 1

F: edge f-g = 1

G: none

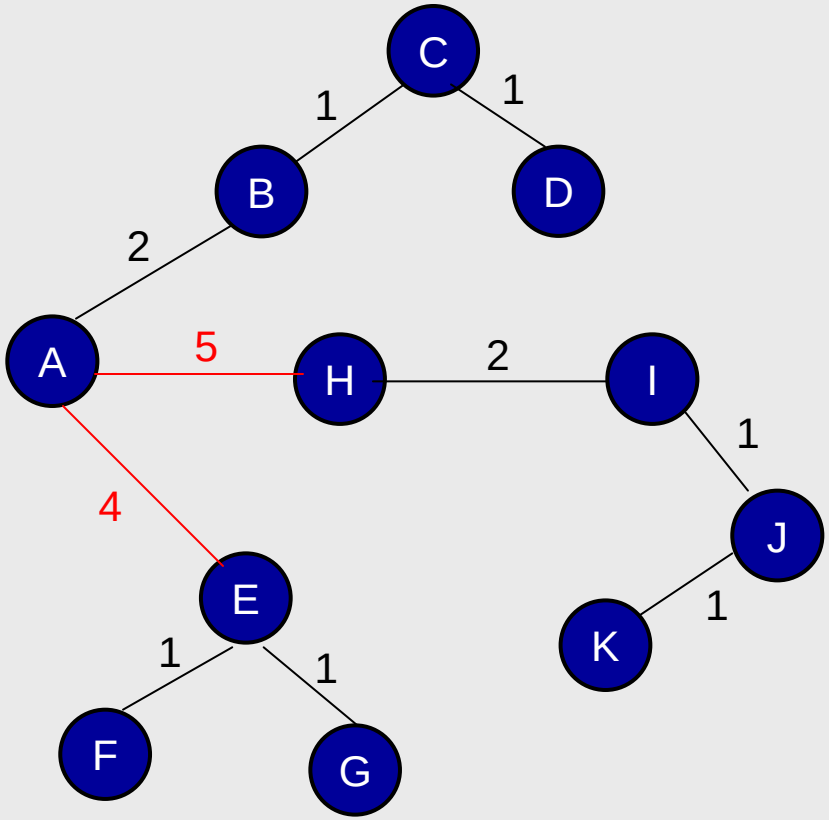
H: edge h-i = 2

I: edge i-j = 1

J: edge j-k = 1

K: none

Baruvka's MST Algorithm – Example 2



PASS #2

A: edge a-e = 4

B: none

C: none

D: none

E: none

F: none

G: none

H: edge a-h = 5

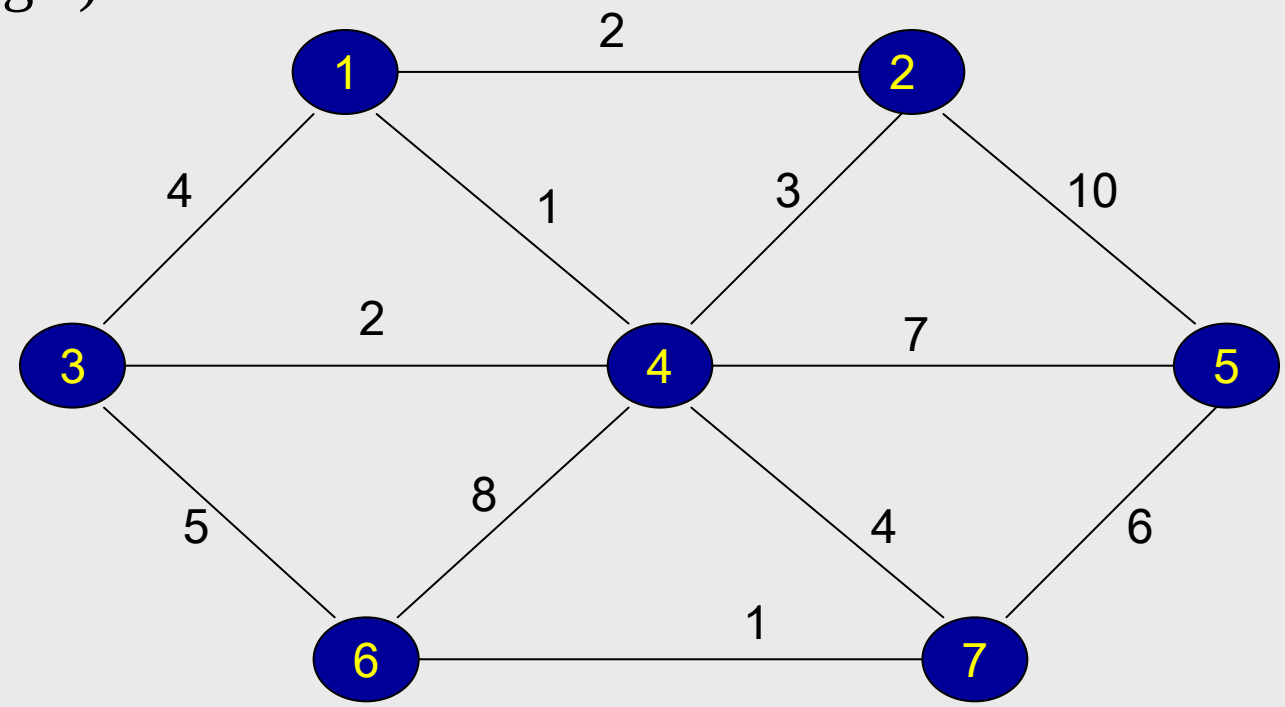
I: none

J: none

K: none

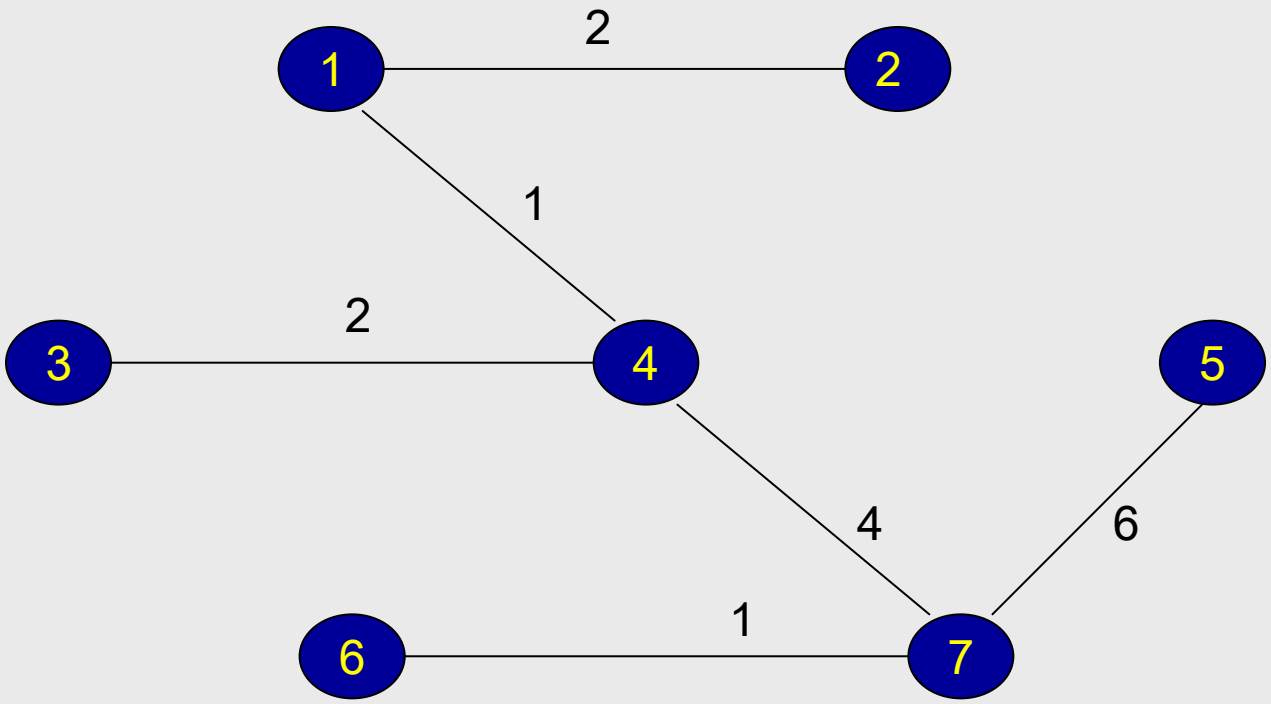
Minimum Spanning Tree – Practice Problem

Generate the minimum spanning tree for the graph shown below using Prim's, Kruskal's, and Baruvka's algorithms. (Answer on next page.)



Minimum Spanning Tree – Practice Problem

Each of the algorithms generates the same MST. Why?



Minimum Spanning Tree – Practice Problem

Table from Prim's algorithm.

vertex	visited	Minimum weight	vertex causing change to min weight
1	T	0	0
2	T	2	1
3	T	2	4
4	T	1	1
5	T	6	7
6	T	1	7
7	T	4	4