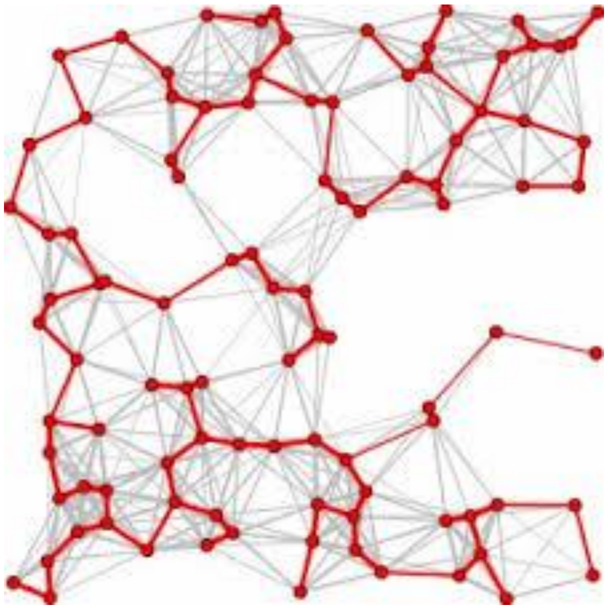
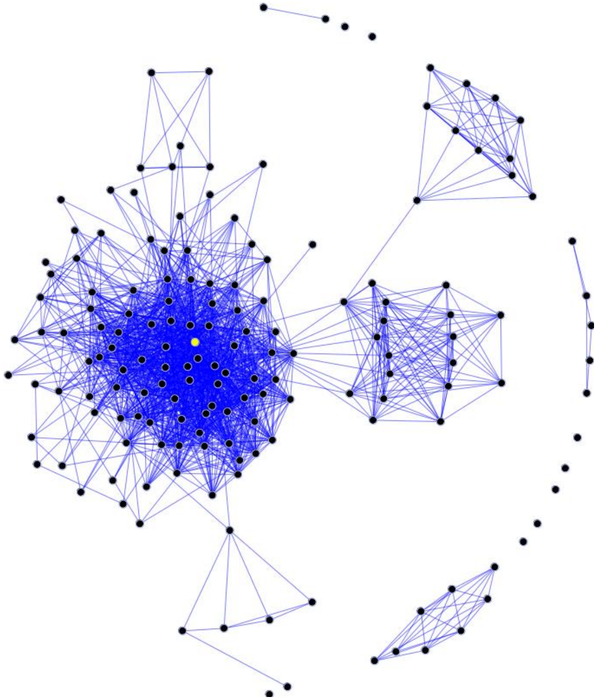
Unit 4
Graph Algorithms





A. Niño. Universidad de Castilla-La Mancha

References

UNIT 4: GRAPH ALGORITHMS

Basic references

- A. Levitin, “Introduction to the Design and Analysis of Algorithms”, Third Edition, Pearson (2012). Chapter 9.
- CLRS: T. Cormen, C. Leiserson, R. Rivest, and C. Stein, ”Introduction to Algorithms”, 3rd edition, MIT Press (2009). Chapters 23 and 24.
- R. Sedgewick, K. Wayne, “Algorithms”, 4th edition, Addison-Wesley Professional (2011). Chapter 4.

Other references

- Graph algorithms: http://en.wikipedia.org/wiki/Book:Graph_Algorithms

Outline

UNIT 4: GRAPH ALGORITHMS

1. Introduction

- **Graphs can be used to model many types of relations and processes in physical, biological, social and information systems. Many practical problems can be represented by graphs**

- **Therefore, a huge amount of algorithms exist to solve different problems in graph theory**

- **Here, we review some of them under the optics of our course**

2. Simple problems

- **There are some typical and simple graphs problems that appear again and again**

- **Usually, they are associated or related to graph traversals**

- **In this point of the unit, we are going to consider some of them**

- **They all involve traversals in one way or another**

2.1. Traversals (I)

- **Remember:** There are two different ways to traverse a graph (with its associated algorithms):
  - Depth first search (DFS)
  - Breath first search (BFS)

- These basic operations are essential for more sophisticated applications

- Let us reconsider here briefly both, DFS and BFS in its basic form (dealing only with vertices)

2.1. Traversals (II)

- **DFS (simplest version)**

Algorithm ***DFS (G, v)***

    Input: *A graph G with all its vertices unvisited and a start vertex v*

    PreVisit(v)   *// Perform some action at node v*

    label v as visited

    for *each node u adjacent to v*

        if *u is not visited* **then**

            set v as node parent of u *// If needed*

            DFS(G, u)

        end_if

    end_for

    PostVisit(v) *// Perform some action at node v*

2.1. Traversals (III)

- **BFS (simplest version)**

```
Algorithm BFS (G, v)
    Input: A graph G with all its vertices unvisited and a start vertex v
        Create an empty queue Q
        Enqueue v in Q
        set v as visited
        while Q not empty do
            u ← Apply a dequeue to Q
            Visit u    // Do whatever you need to do in vertex u (if needed)
            for each vertex z adjacent to u
                if z is unvisited then
                    set z as visited
                    set u as node parent of z    // If needed
                    Enqueue z in Q
                end_if
            end_for
        end_while
```

2.1. Traversals (IV)

- **DFS/BFS analysis**
  - **Setting/getting a vertex label takes O(1) time**
  - **Using adjacency lists, at each vertex we process only its adjacent edges. (Remember, Σv deg(v) = 2m). Therefore:**

$$\sum_{i=1}^{n}(1+k_i) = n + \sum_{i=1}^{n} k_i = n + 2m \in O(n) + O(m) = O(n+m)$$

  - **Using adjacency matrices, we process all the edges for each vertex. Therefore,**

$$\sum_{i=1}^{n}(1+n) = n + n * n \in O\left(n^2\right)$$

2.2. Connected components (I)

- **Remember:** In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

- Connected component identification (also known as connected component labeling or analysis, blob extraction, region labeling, blob discovery, or region extraction) is important in computer vision (for instance, in OCR or in the analysis of astrophysical images).

- We can determine the number of connected components easily using graph traversals.

2.2. Connected components (II)

- **Analysis: We traverse the vertices and edges of each connected component. Assuming the complexity of the traversal of each component i is $O(m_i+n_i)$:**

$$\sum_{i=1}^{nc} O(m_i + n_i) = O(m + n)$$

- **Is linear in m+n**

Algorithm ***connectedComponents (G)***

Input: graph G = (V, E). m = |E|, n=|V|

Output: nc, the number of connected components

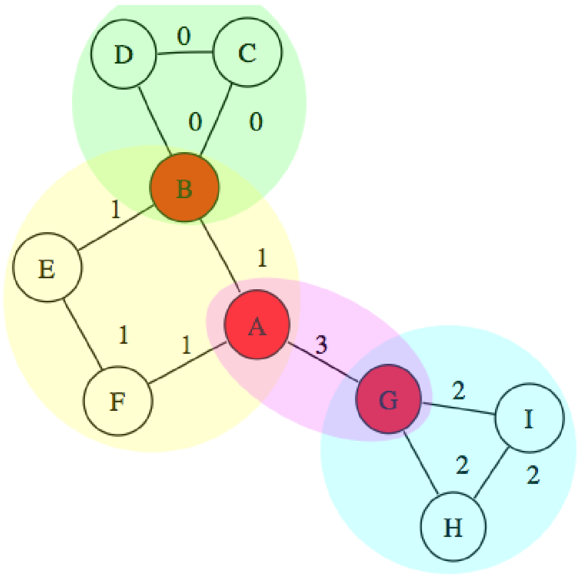
```
nc ← 0
for each vertex v in G
    if v not visited then
        traverse G from v marking
        each vertex reached as visited
        nc ← nc+1
    end_if
end_for
return nc
```

2.3. Articulation points (I)

- A connected, undirected graph is **biconnected** if the graph is still connected after removing any one vertex (*i.e.*, when a "node" fails, there is always an alternative route).

- If a graph is not biconnected, the disconnecting vertices are called **articulation points** or, equivalently, **cut vertices** (see examples in the figure).

Examples of where articulation points are important are airline hubs, electric circuits, network wires, protein bonds, traffic routers, and numerous other industrial applications. Also: law enforcement, intelligence and the military (disruption of criminal or terrorist networks and enemy targets)



2.3. Articulation points (II)

- **We can determine if a vertex is an articulation point very easily.**

- **Analysis: Similar to the connected components case:**

$$\sum_{i=1}^{nc} O(m_i + n_i) = O(m + n)$$

- **Is linear in m+n.**

Algorithm ***articulationPoint (G, v)***

Input: graph G = (V, E) and a vertex *v*. m = |E|, n=|V|

Output: if *v* is an articulation point

```
Set v as visited
nc ← 0
for each vertex u in G
    if u not visited then
        traverse G from u marking
        each vertex reached as visited
        nc ← nc+1
    end_if
end_for
return nc>connectedComponents(G)
```

3. The single source shortest path problem (SSSP)

- **In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.**

- **The related single source shortest path problem is the problem of finding the shortest path from one vertex (or node) to every other such that the sum of the weights of its constituent edges is minimized.**

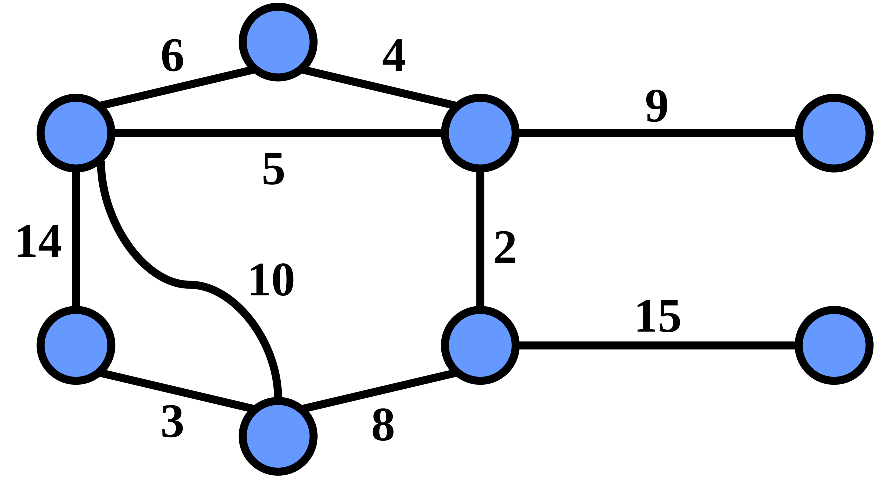
- **Here we focus on the latter.**

3.1. Presentation of the problem (I)

- Given: a connected graph G with non-negative weights on the edges and a source vertex s in G.

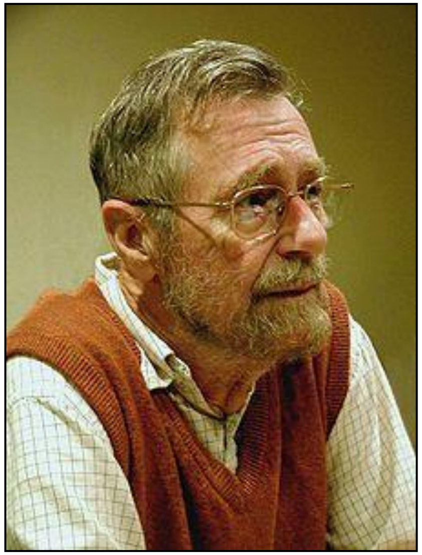
- Goal: determine the shortest paths from s to all other vertices in G.

- Useful in many applications (e.g., road map applications).



3.1. Presentation of the problem (II)

- **Applying the greedy pattern to the shortest path problem results in Dijkstra's algorithm.**



Edsger Wybe Dijkstra

- **Edsger Wybe Dijkstra: One of the fathers of modern computer science.**

- **Some Dijkstra's quotes:**
  - *Computer Science is no more about computers than astronomy is about telescopes*
  - *Object-oriented programming is an exceptionally bad idea which could only have originated in California*
  - *It is not the task of the University to offer what society asks for, but to give what society needs*

3.2. Dijkstra’s algorithm (I)

- Goal: To compute 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 ← Very important, ensures optimality

- Strategy:
  - 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 the beginning $d(s)=0$ and $d(u \neq s)=\infty$

3.2. Dijkstra's algorithm (II)

- **Greedy choice:**
  - 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 and iterate

- **The process of updating distance labels is called relaxation.**

- **How do we relax an edge?**

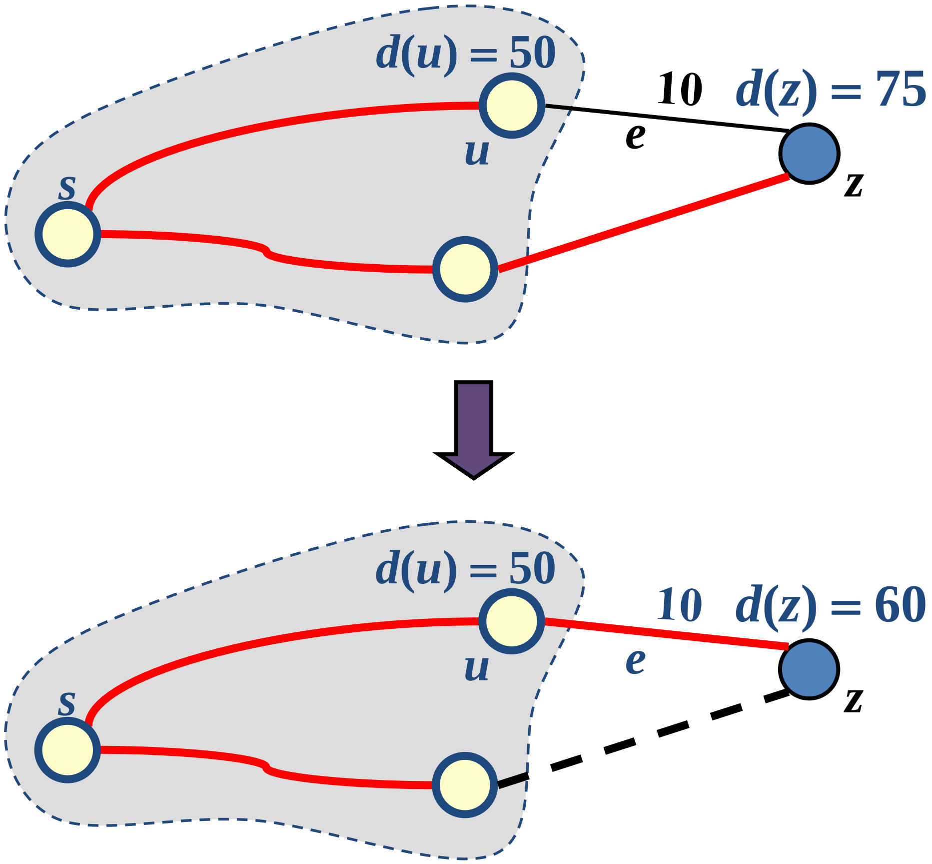
3.2. Dijkstra's algorithm (III)

- 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* is the process of updating distance *d*(*z*) as follows:

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

La diferencia que tiene con el problema de salesman problem es que Dijkstra busca la relajación de los arcos

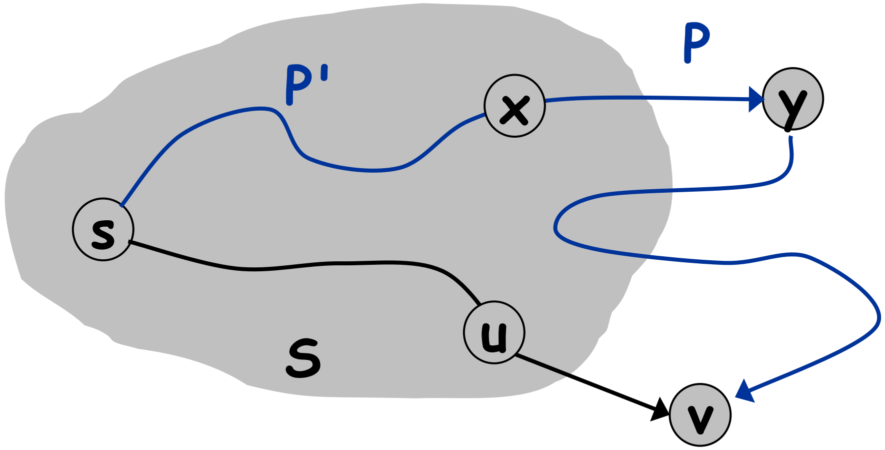


3.2. Dijkstra's algorithm (IV)

- **Correctness:**
  - **Theorem: Given a graph G and one of its vertex, s, Dijkstra's algorithm determines the shortest paths from s to all other vertices in G**
  - **Proof. (by induction)**
    - We maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to any v.
    - Base case: s = v, is trivial.
    - Inductive hypothesis: Assume the theorem is true for $|S| = k \geq 1$.

3.2. Dijkstra's algorithm (V)

- Let v be the k node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\ell(P) \underset{\uparrow}{\geq} \ell(P') + \ell(x,y) \underset{\uparrow}{\geq} d(x) + \ell(x,y) \underset{\uparrow}{\geq} \pi(y) \underset{\uparrow}{\geq} \pi(v)$$

nonnegative weights | inductive hypothesis | defn of $\pi(y)$ | Because Dijkstra chosed v instead of y

3.2. Dijkstra's algorithm (VI)

Algorithm ***DijkstrasSSSP* (*G, s*)**

Input: graph G = (V, E) including costs, weights, of edges and starting vertex s

Output: array D with path lenghts

Initialize *D[s]* ← 0 and *D[u≠s]* ← ∞

Q ← new priority queue containing all vertices and using the D labels as keys

Cola prioritaria:

while *Q is not Empty* **do**

u ← *Q.removeMin*( )

for *all vertex z adjacent to u such that z is in Q*

if *D[u] +w(u, z)*< *D[z]* **then** // Relax edge (u, z)

D[z] ←*D[u] +w((u, z))*

Change to D[z] the key of vertex z in Q

end_if

end_for

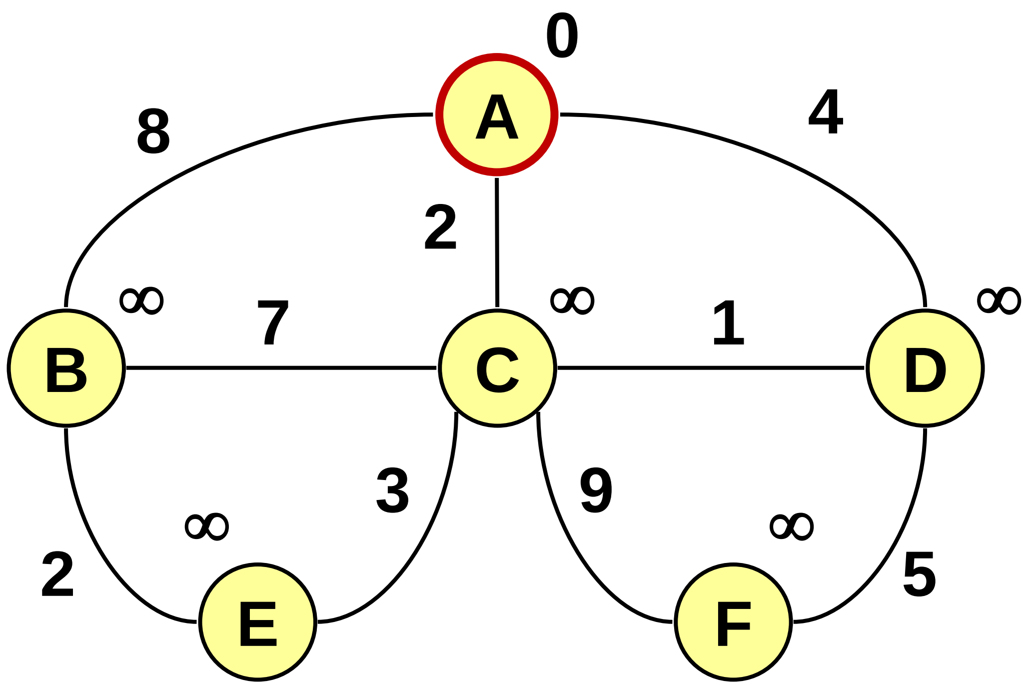
end_while

return *the label D[u] of each vertex u*

- **Fast version with heap**

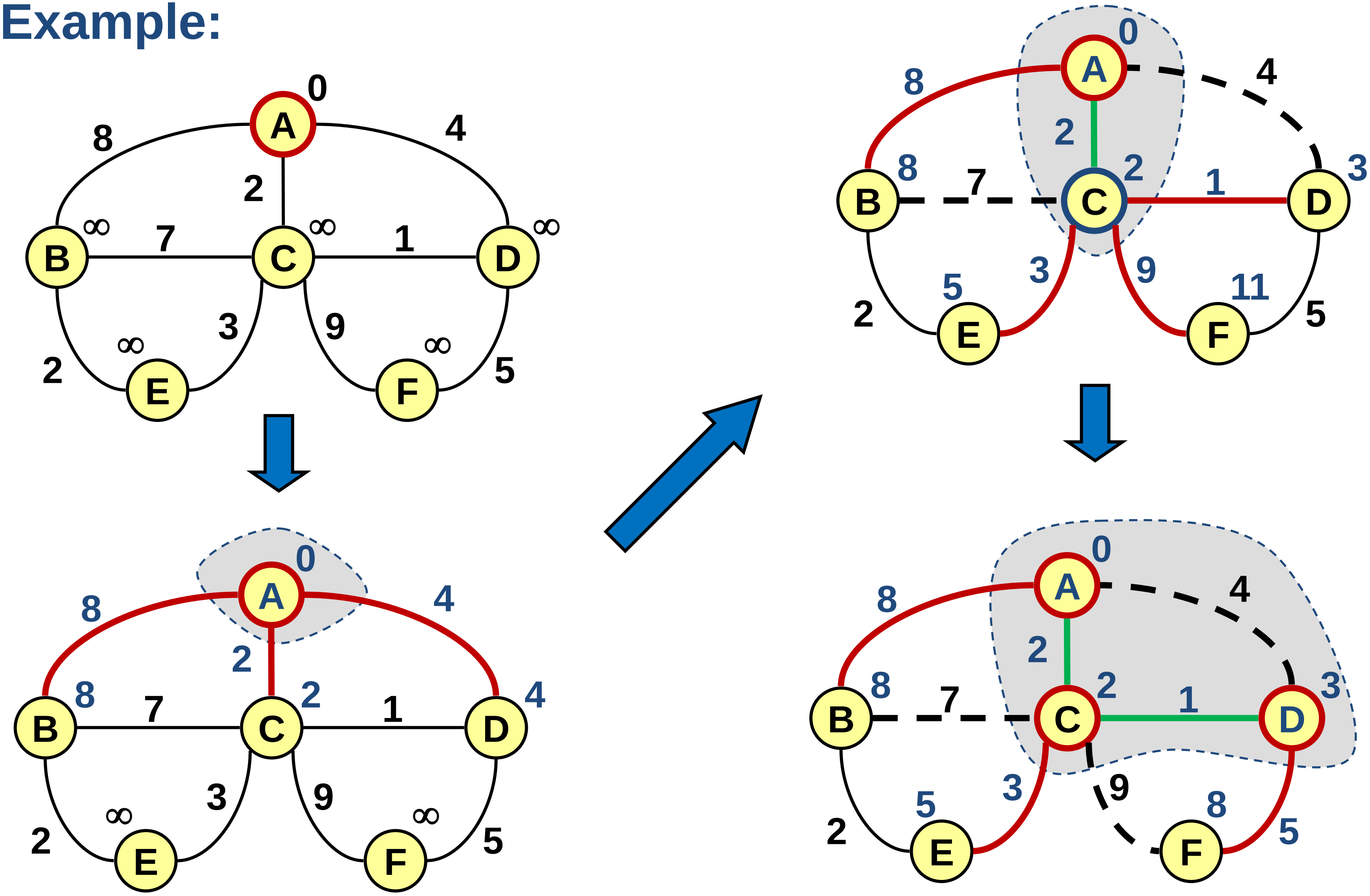
Dijkstra’s example (I)

- **Apply Dijkstra’s algorithm to the following graph starting at vertex A:**



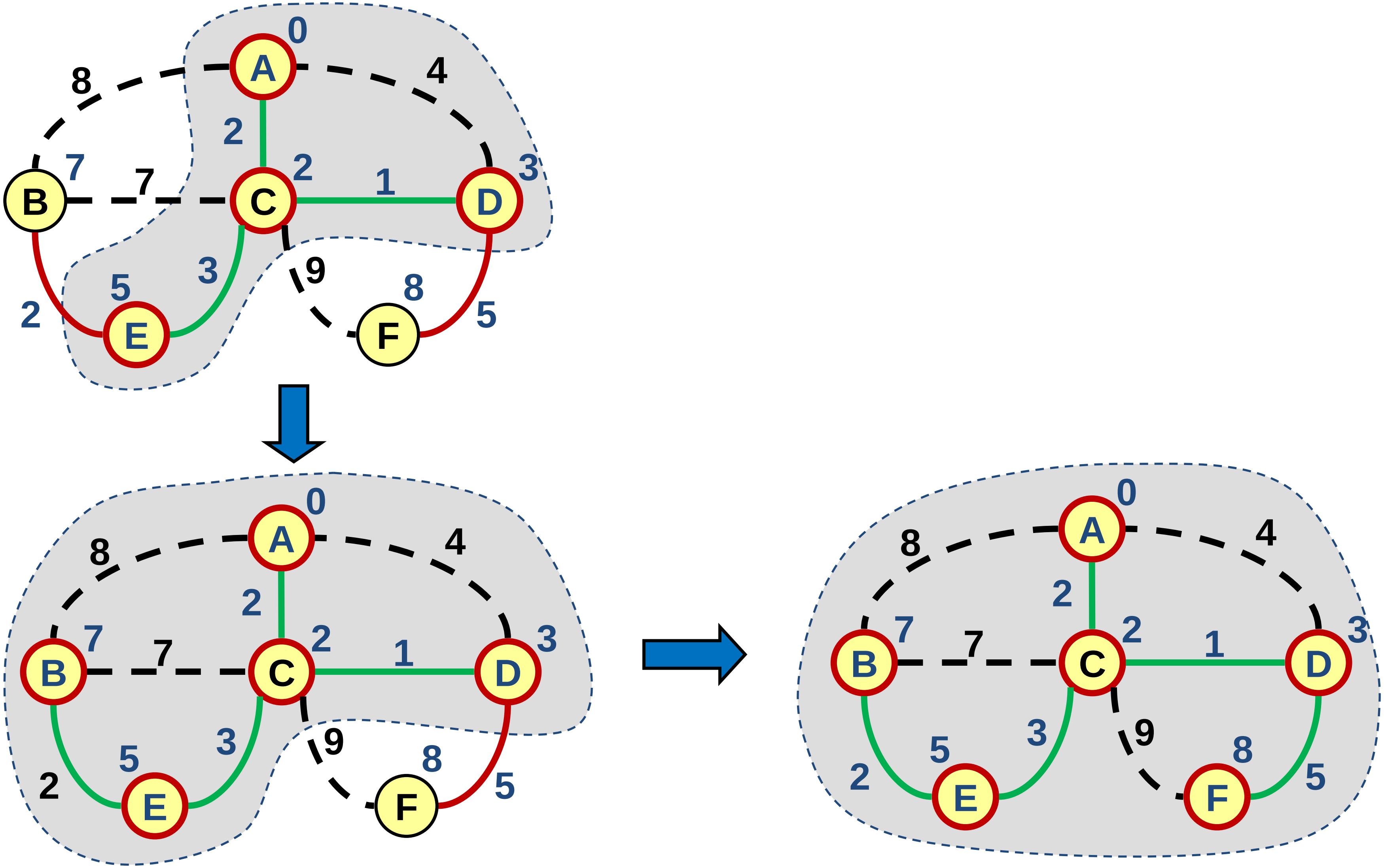
Dijkstra's example (II)

- **Example:**



Dijkstra’s example (III)

- **Example:**



3.2. Dijkstra's algorithm (IX)

- **Running time of Dijkstra's algorithm. We assume:**
  - **We represent graph G using an adjacency list.**
  - **For the priority queue Q we use a heap $\Rightarrow$ extracting vertex u with smallest D[u] (removeMin method) in O(log n) a total of n times.**
  - **For updating the key of z in the priority queue we use an adaptable priority queue $\Rightarrow$ queue updated in O(log n) a total of m times.**

- **The algorithm runs in O((n+m) log n)**

- **When negative weights do exist, we can use the Bellman-Ford Algorithm**

4. The minimum spanning tree problem

- In graph theory, a spanning tree of an undirected graph, G, is a subgraph that is a tree which includes all of the vertices of G, with minimum possible number of edges. In general, a graph may have several spanning trees.

- A minimum spanning tree (MST) of a connected, edge-weighted undirected graph is a subset of the edges that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

- In other words, it is a spanning tree whose sum of edge weights is as small as possible.

4.1. Presentation of the problem (I)

- **Informal goal: Connect a bunch of points together as cheaply as possible**
- **We use a weighted graph G (V, E) with m edges and n vertices**
- **We are interested in finding a tree, T, containing all the vertices of G with the minimum total weight**
- **We seek to minimize:** $$w(T) = \sum_{(v,u)\ in\ T} w((v, u))$$
- **One typical and very useful task in graph theory. Applications: Clustering, networking, and many more**

4.1. Presentation of the problem (II)

- **Blazingly Fast Greedy Algorithms available:**
  - **Prim's Algorithm [1957, also Dijkstra 1959, Jarnik 1930]**
  - **Kruskal's algorithm [1956]**

- **They work in O(m log n) time (using suitable data structures)**

- **They are almost linear in the number of edges ⇒ About the same time than reading the graph**

4.2. Prims's (Prim-Jarnik) algorithm (I)

- **Problem definition:**
  - **Input: Undirected and connected graph G = (V, E) and a cost w_e for each edge e $\in$ E.**
    - › Assume adjacency list representation
    - › OK if edge costs are negative
  - **Output: minimum cost tree T $\in$ E that spans all vertices:**

- **In the present problem we have,**
  - **Objective: minimize:** $\boldsymbol{w(T) = \sum_{(u,v)\in T} w_e\,((u,v))}$
  - **Constraint: T must be a spanning tree**

- **The algorithm resembles Dijkstra's**

4.2. Prims's (Prim-Jarnik) algorithm (II)

- **Greedy choice: Select the edge with smaller cost connecting some vertex of the tree with a vertex not yet in the tree**
- **Correctness: Using the cut property and proving that the algorithm generates a spanning tree**
- **CUT PROPERTY: Consider an edge e of G. Suppose there is a cut (A, B) such that e is the cheapest edge of G that crosses it. Then e belongs to the MST of G.**

Algorithm ***PrimsMST*** **(*G*)**

Input: graph G = (V, E)
including costs, weights, of edges
Output: spanning tree T
select arbitrarily vertex s
$X = \{s\}$
$T = \varnothing$
while $X \neq V$ **do**
    select e = (u, v), the cheapest edge of G with $u \in X$, $v \notin X$
    Add e to T
    Add v to X
end_while
return *T*

4.2. Prims's (Prim-Jarnik) algorithm (III)

- Time complexity of the straightforward implementation:
  - O(n) iterations [where n = # of vertices]
  - O(m) time per iteration (brute-force linear search of edges) [m = # of edges]
  - Total: $O(m \cdot n)$
- Our eternal mantra: **CAN WE DO BETTER?**
- Yes, using an appropriate data structure
- We are going to use a priority queue (heap) with logarithmic insertion and removal time

4.2. Prims's (Prim-Jarnik) algorithm (IV)

- 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.
- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: 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 still in the queue
- See pseudocode in next slide

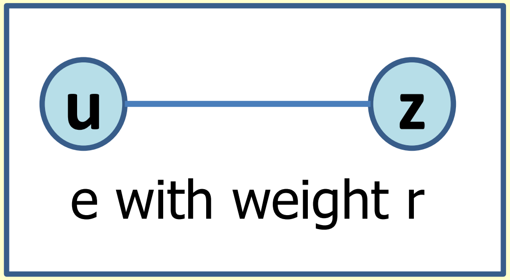
4.2. Prims's (Prim-Jarnik) algorithm (V)

Algorithm *PrimsMST_PriorityQueue* (*G*)

Input: graph **G** = (V, E) with weights on edges
Output: spanning tree (described as distances and parents)

s ← a vertex of ***G***

Initialize *D[s]* ← 0, *D[s≠v]* ← ∞ and parent [∀vertex ∈ **G**] ← ∅
Q ← priority queue with all vertices and the D distances as keys
while *Q not empty*
  u ← get and remove min of Q
  for all *e ∈ incident edges of u*
    z ← opposite vertex to u over e
    if *z in queue*
      r ← weight of e
      if *r < D[z]*
        D[z] ← r
        parent[z] ← u
        replace (update) key of z with r in Q
return distances and parents



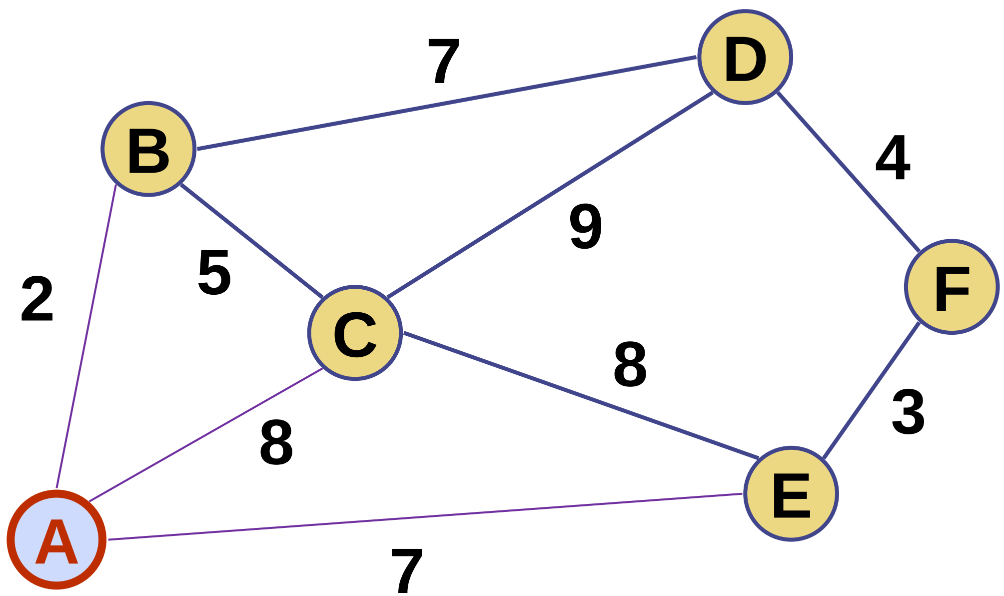
4.2. Prims's (Prim-Jarnik) algorithm (VI)

- **Time complexity of the clever heap based implementation (storing vertices in the heap):**
  - **O(m) preprocessing of edges to fill the heap of vertices**
  - **O(n) iterations**
  - **O(log n) time per iteration (search in the heap)**
  - **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 (Recall that $\sum_v \deg(v) = 2m$)**
  - **Total: O((m+n) log n), which is O(m log n)**

    (in a connected graph m≥ n-1)

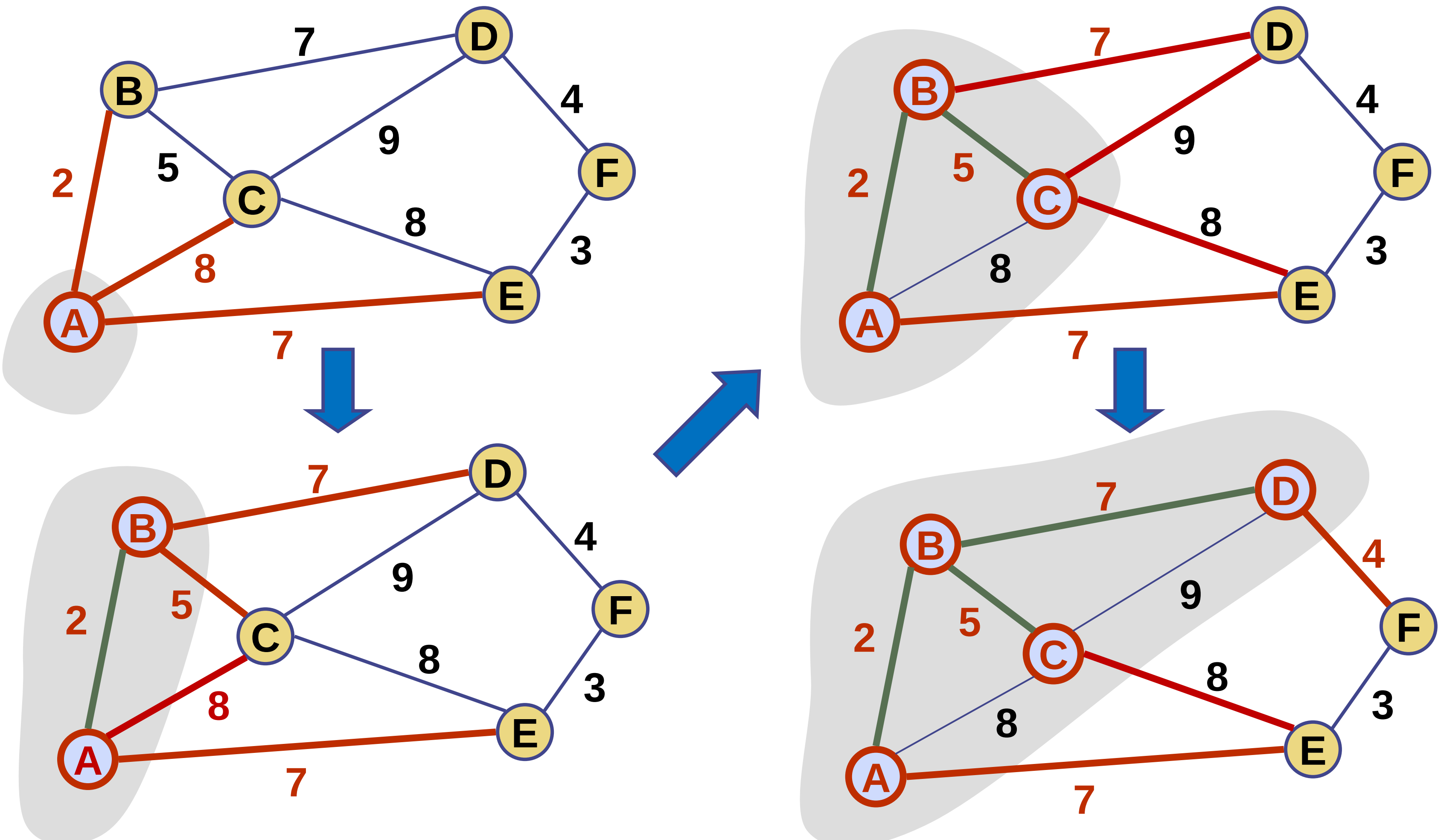
Prim-Jarnik example (I)

- **Apply Prim-Jarnick algorithm (heap based implementation with edges in the heap) to the following graph starting at vertex A:**



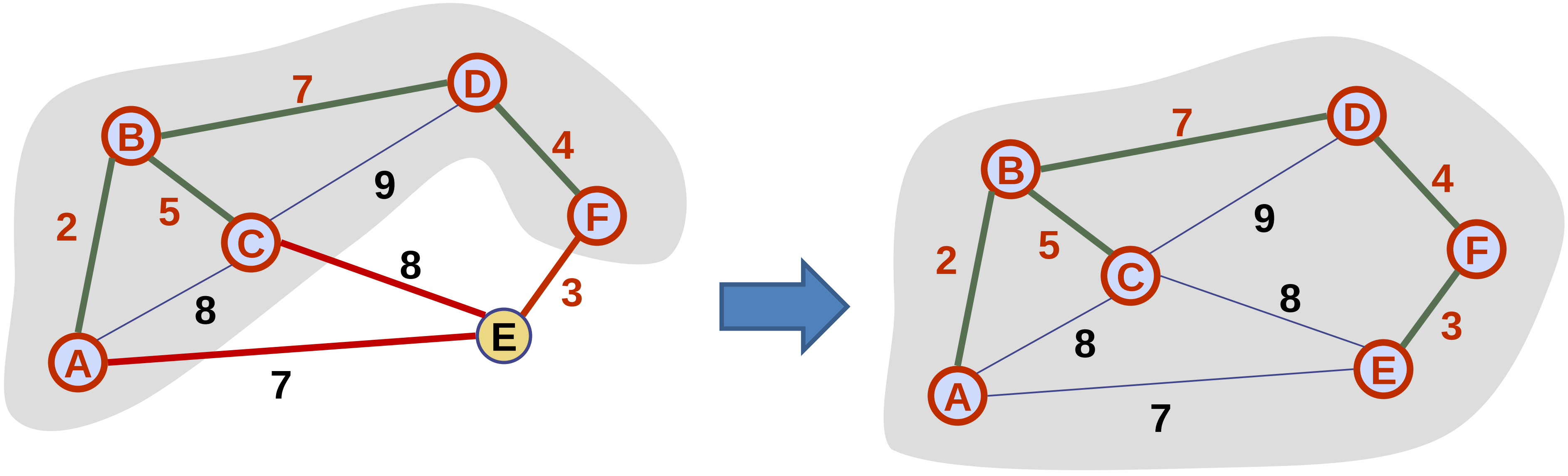
Prim-Jarnik example (II)

- **Example:**



Prim-Jarnik example (III)

- **Example:**



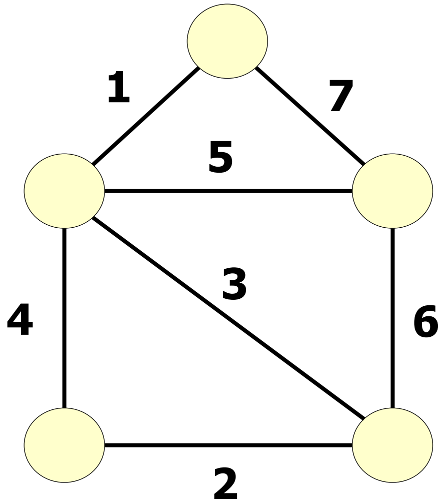
4.3. Kruskal's algorithm (I)

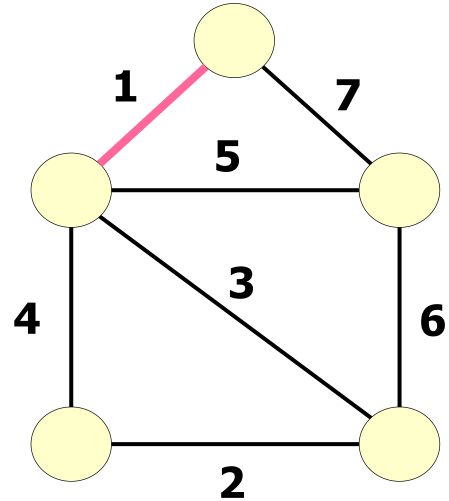
- This is another greedy approach for building a MST

- Here, we focus on the cost of edges selecting the edge with minimum cost among those not already in the tree that leads to no cycle.

- In Kruskal's we don´t care if the growing tree is a connected subgraph (the tree is connected at the end)

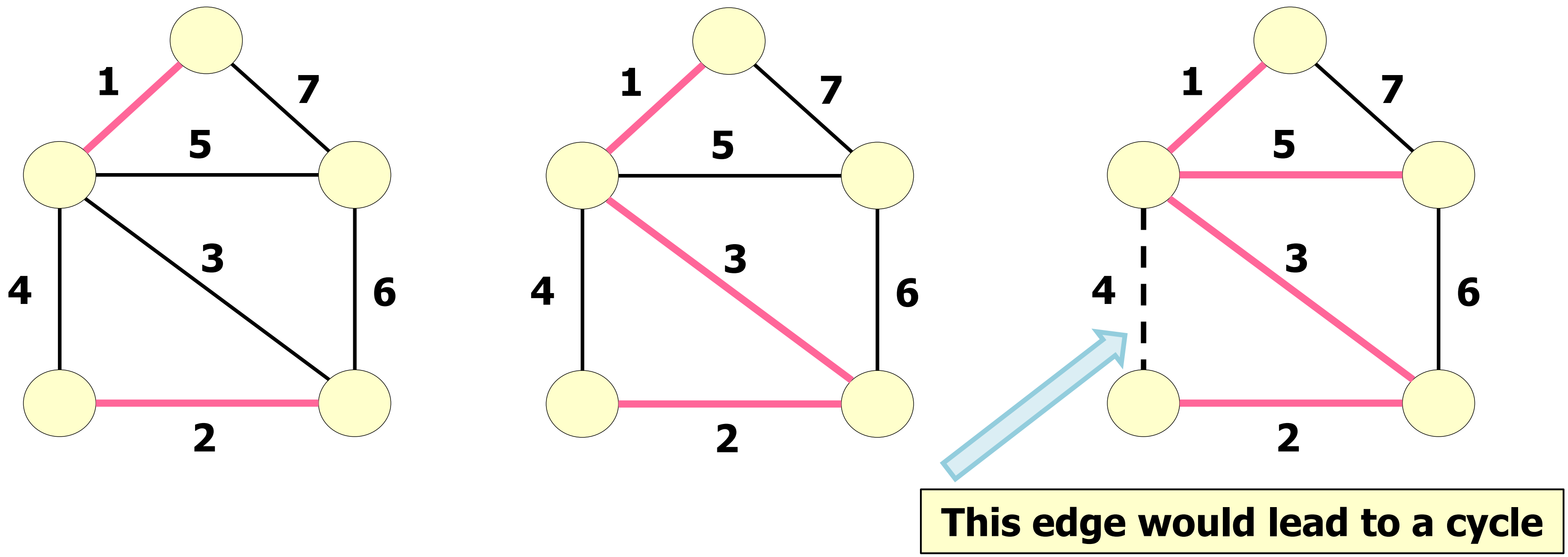
- Example:





4.3. Kruskal's algorithm (II)

- **Example (cont.):**



- **Kruskal's algorithm is linked to clustering (community detection) algorithms**

4.3. Kruskal's algorithm (III)

- **The algorithm can be presented in a similar way to Prim`s**

- **Problem definition:**
  - **Input: Undirected and connected graph G = (V, E) and a cost w_e for each edge e ∈ E.**
    - › Assume adjacency list representation
    - › OK if edge costs are negative
  - **Output: minimum cost tree T ∈ E that spans all vertices:**

- **In the present problem we have,**
  - **Objective: minimize:** $\boldsymbol{w(T) = \sum_{(u,v)\in T} w_e\,((u,v))}$
  - **Constraint: T must be a spanning tree**

4.3. Kruskal's algorithm (IV)

- **Greedy choice: Select the edge with smallest cost out of the tree that leads to no cycle in the tree.**
- **Correctness: Using the cut property and proving that the algorithm generates a spanning tree.**
- **CUT PROPERTY: Consider an edge e of G. Suppose there is a cut (A, B) such that e is the cheapest edge of G that crosses it. Then e belongs to the MST of G.**

Algorithm ***KruskalsMST*** **(*G*)**

Input: graph G = (V, E)
including costs, weights, of edges
Output: spanning tree T
select arbitrarily vertex s
Sort edges in order of increasing cost
$T = \varnothing$
for $i \leftarrow 0$ *to* $m\text{-}1$ *and* $|T| \neq n\text{-}1$ **do**
    select e, the cheapest edge of G such that $e \not\subseteq T$ *and* $T \cup e$ *has no cycles*
    Add e to T
end_for
return *T*

4.3. Kruskal's algorithm (V)

- **Time complexity of direct Kruskal's:**
  - **Sorting edges: O (m log m) ≡ O(m log n) since at most m = O(n^2) and O(m log n^2) =O(m 2 log n) = O(m log n)**
  - **for loop: O(m)**
  - **Determine if T∪ e has no cycles (using, DFS or BFS on T, which contains ≤ n -1 edges): O(n)**
  - **Total: O(m log n + m·n) = O(m·n)**

- **Not very good, so (our eternal mantra):**

CAN WE DO BETTER?

- **Yes, using an appropriate data structure**

- **We are going to use the Union-Find data structure**

4.3. Kruskal's algorithm (VI)

- **Version with Union-Find**
- **Complexity**
  - Sorting edges O (m log n)
  - for loop: O(m) times
  - Union, find: log*n = O(1)
  - Total:

    O(m log n + m · 1) =

    O(m log n)
- **As good as Prim's!!!!**
- **Sorting dominates the algorithm**

Algorithm ***KruskalsMST_UF* (*G*)**

Input: graph G = (V, E) including costs, weights, of edges

Output: spanning tree T

select arbitrarily vertex s

Sort edges in order of increasing cost

$T = \varnothing$

for $i \leftarrow 0$ *to* m-1 *and* $|T| \neq n$-1 **do**

select e=(u, v), the cheapest edge of G

if (find(u) $\neq$ *find (v))*

union (u, v)

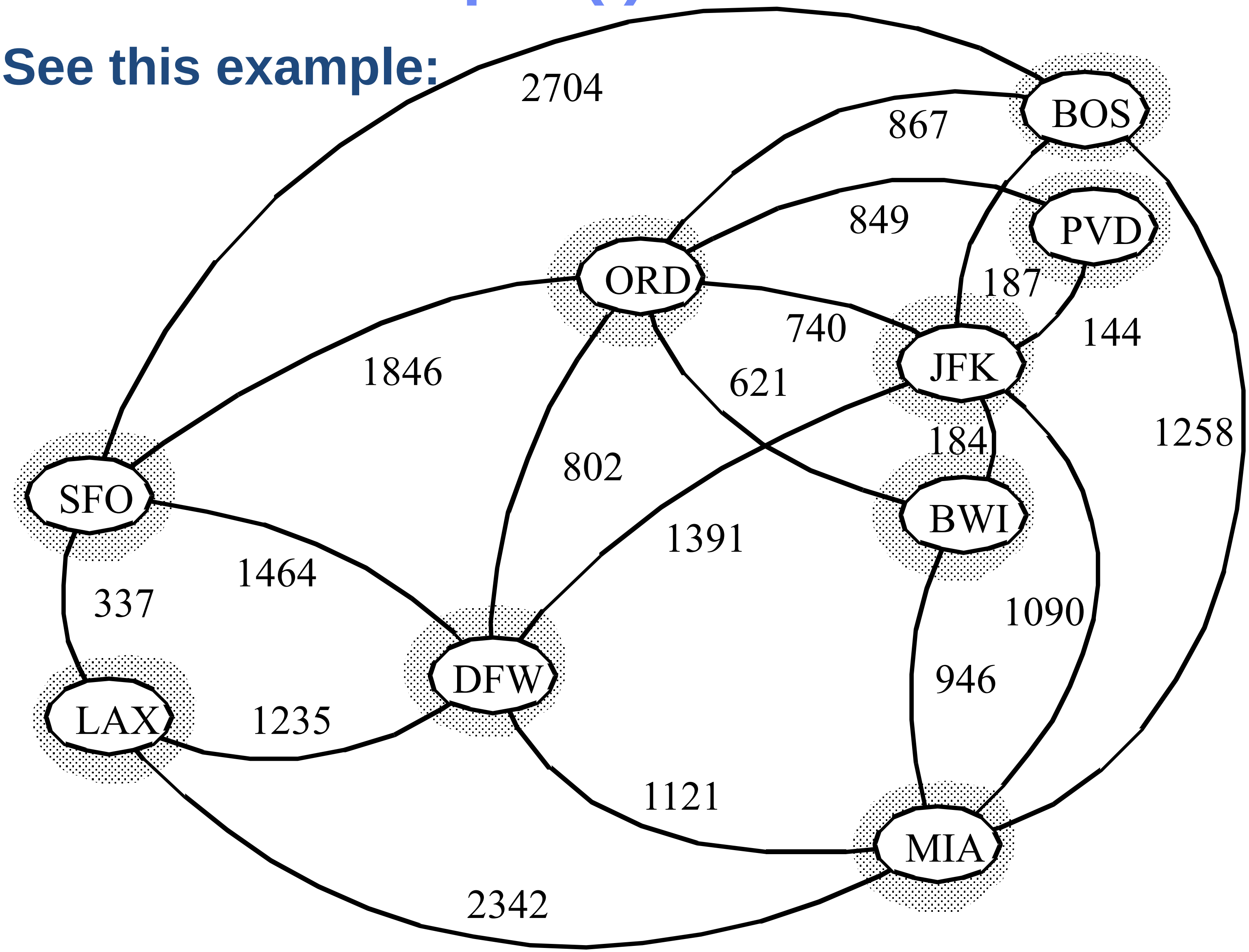
Add e to T

end_for

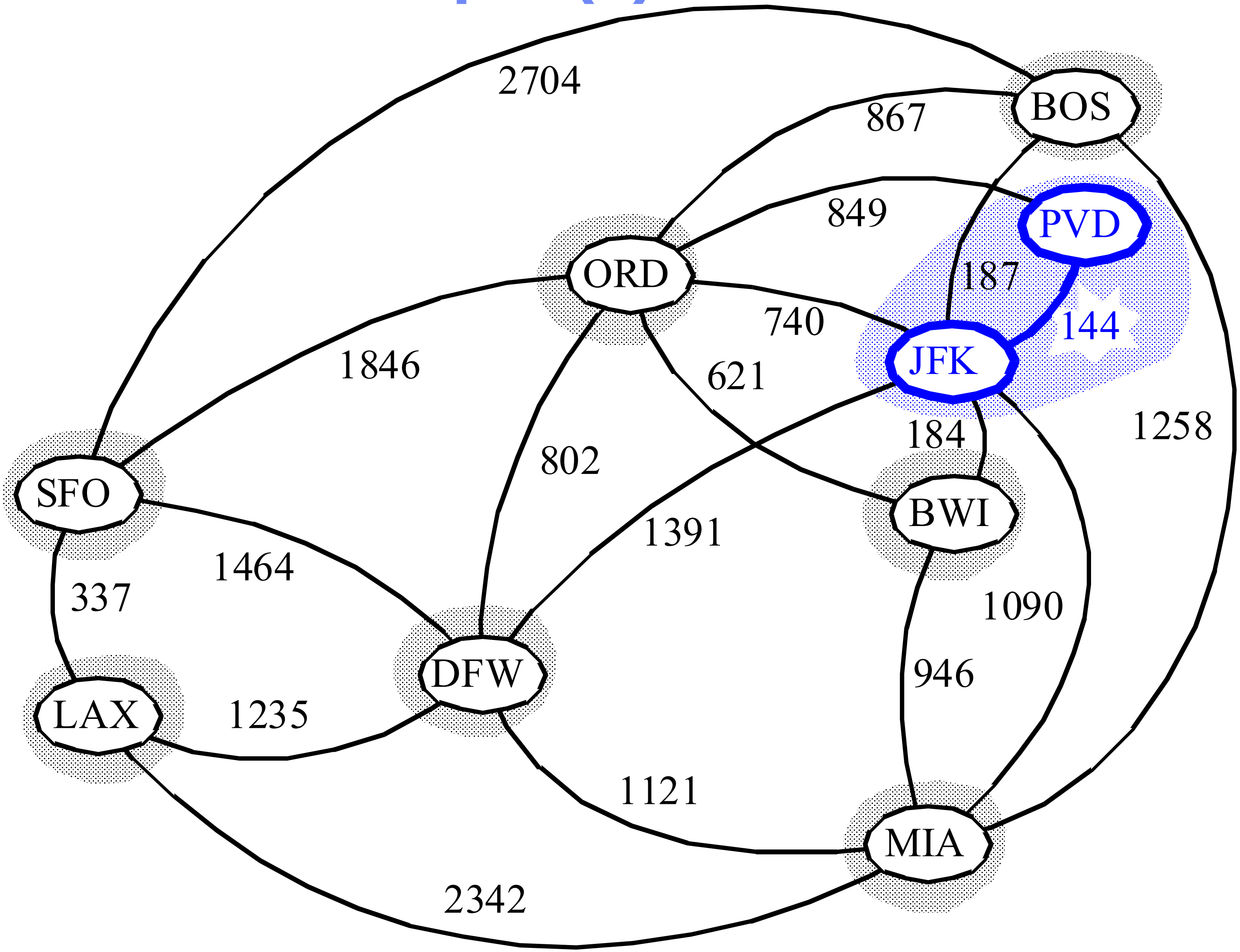
return *T*

Kruskal's example (I)

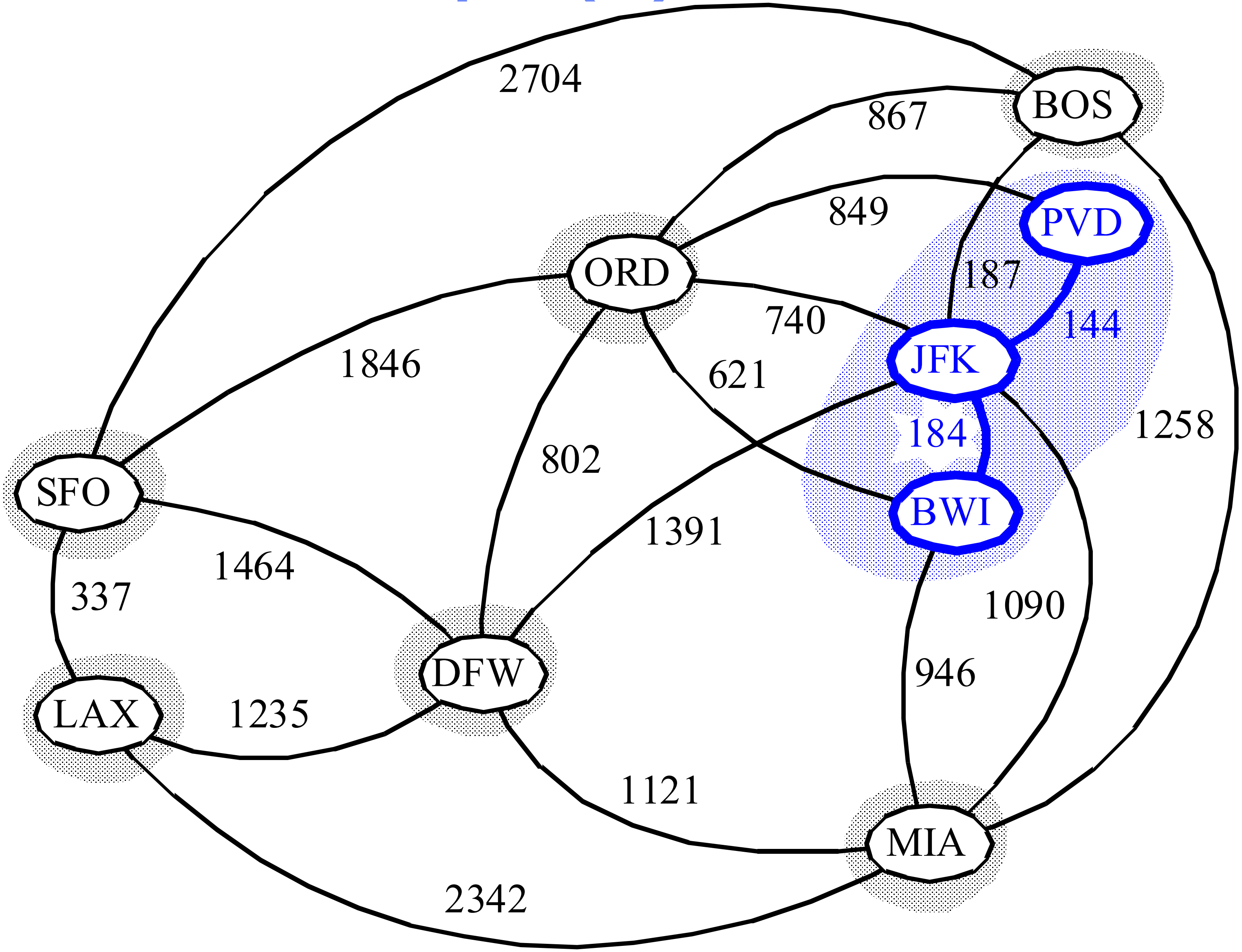
- **See this example:**



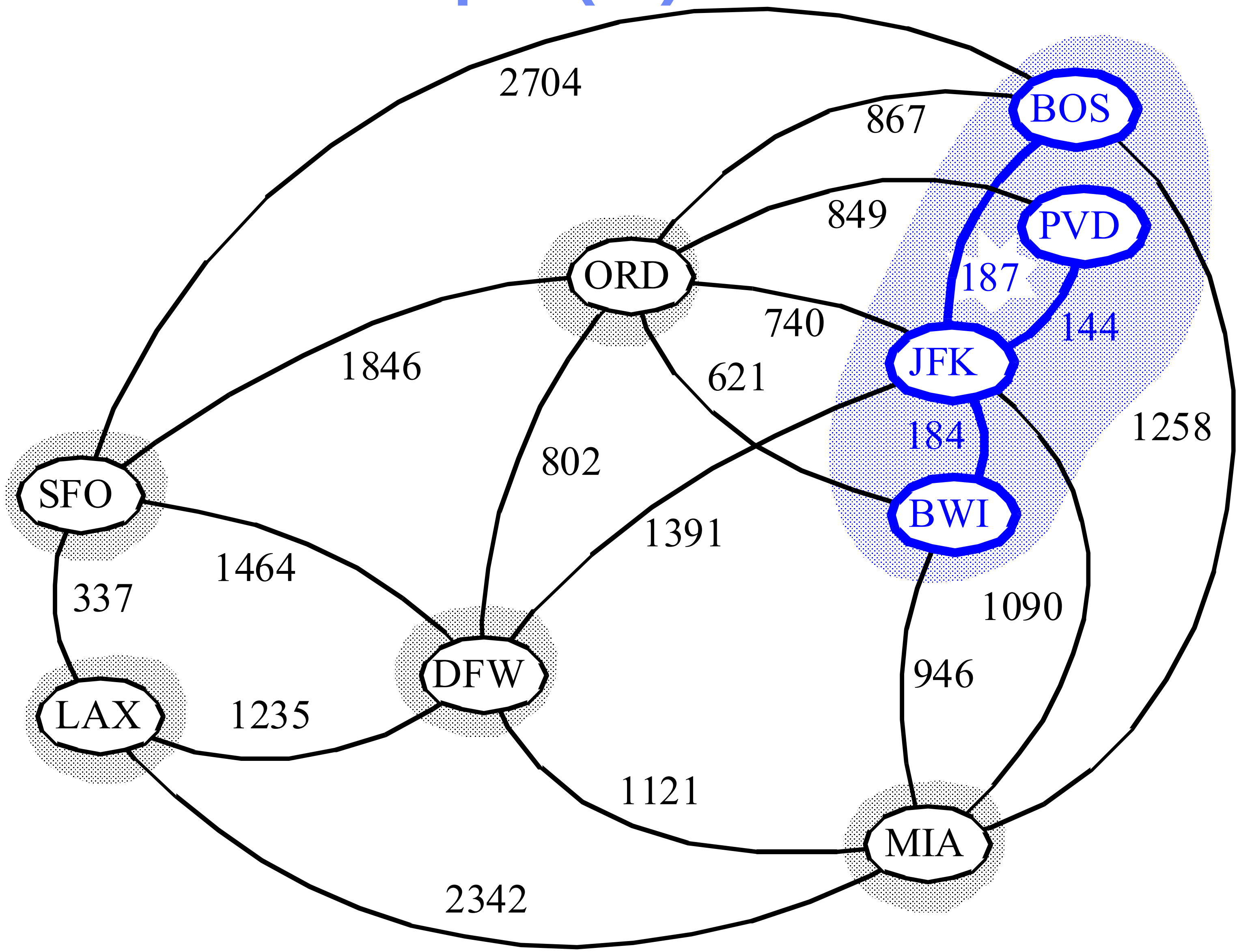
Kruskal’s example (II)



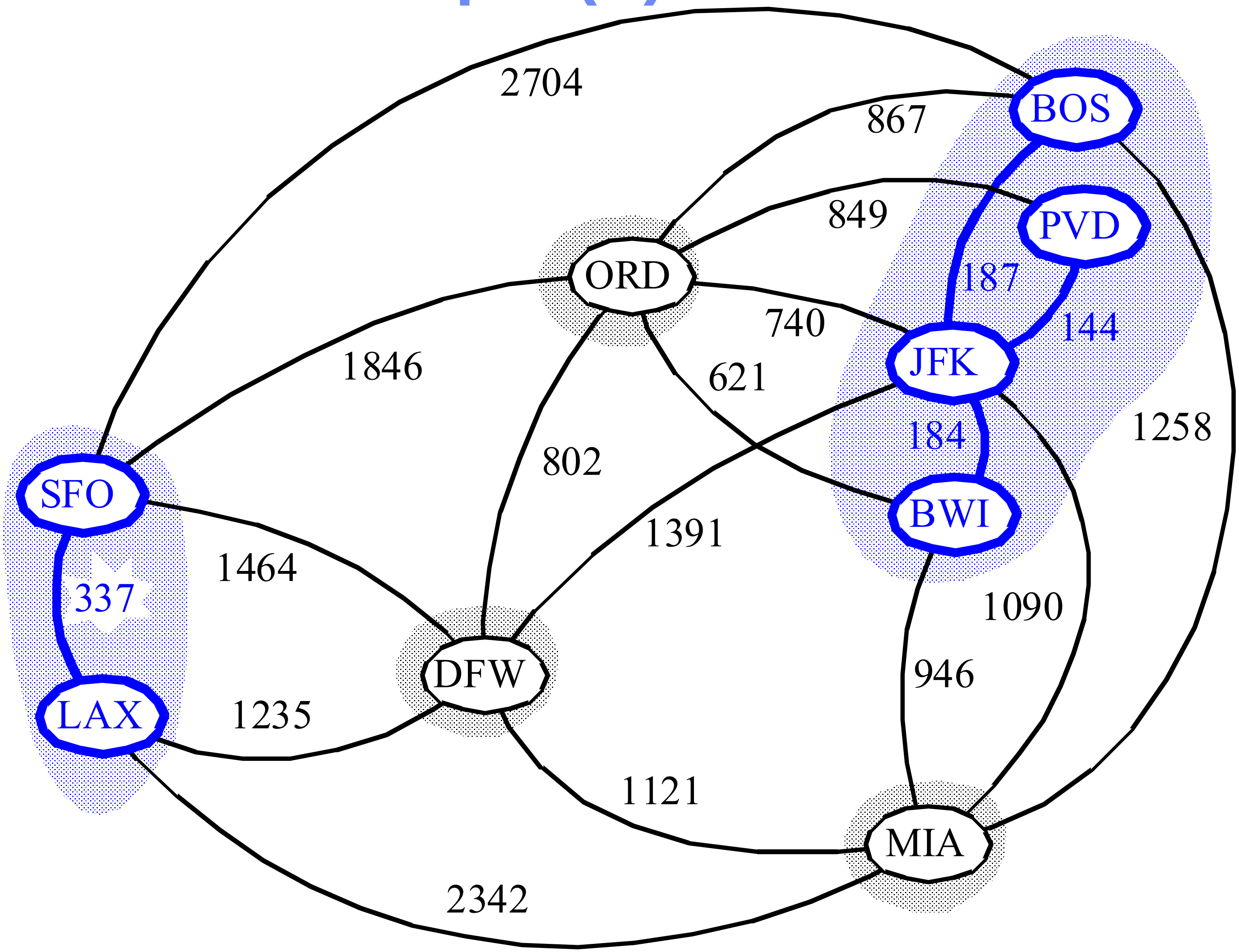
Kruskal’s example (III)



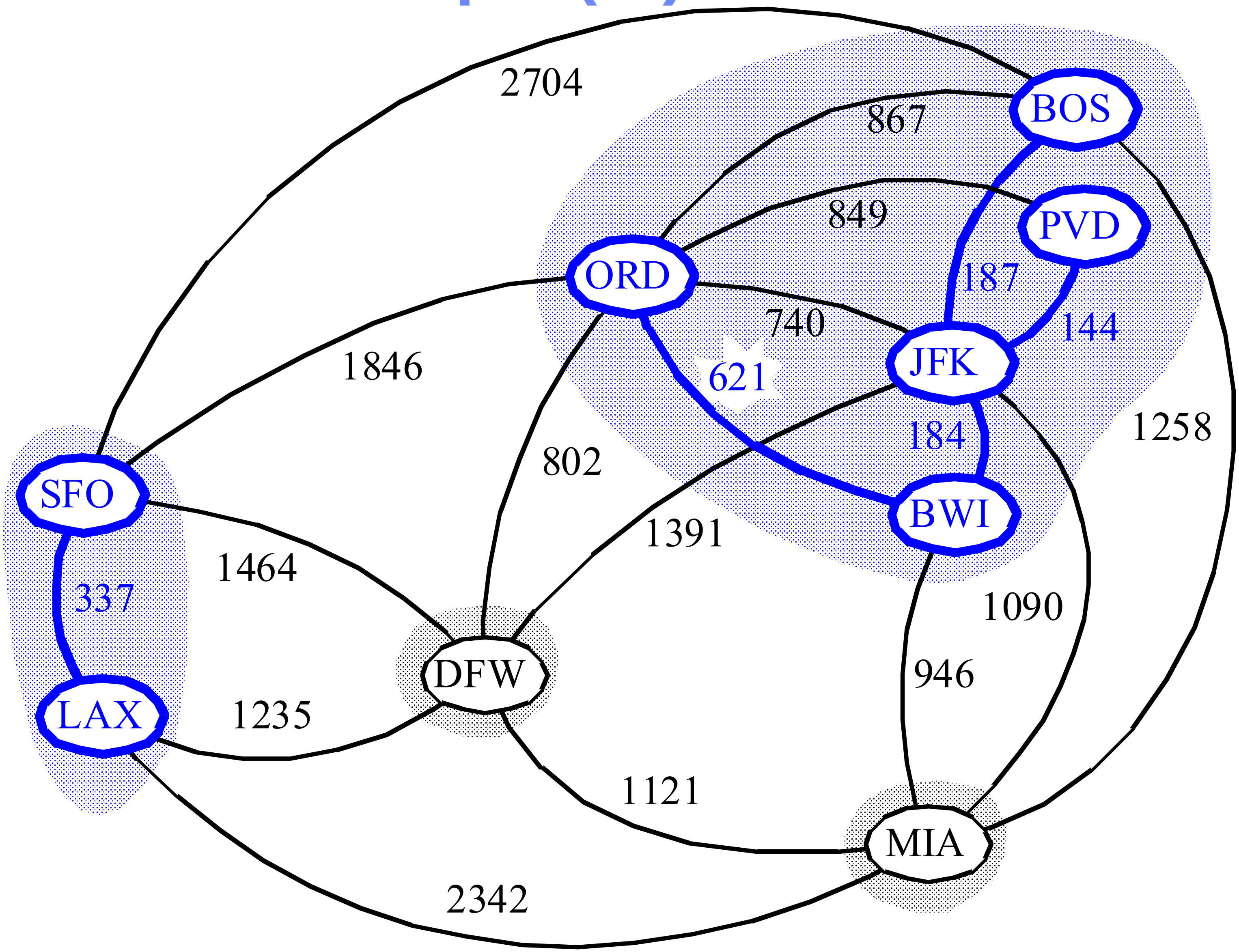
Kruskal's example (IV)



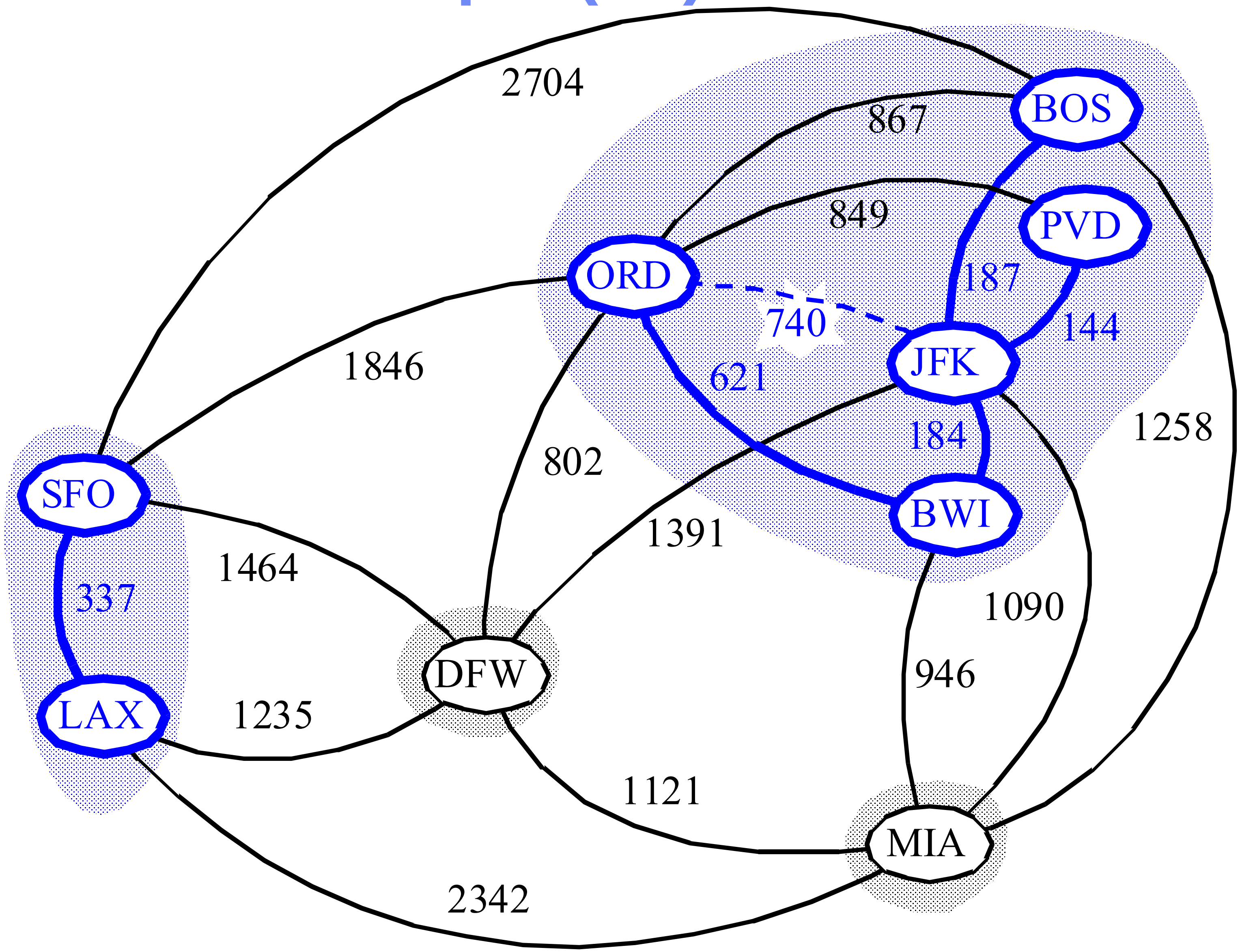
Kruskal’s example (V)



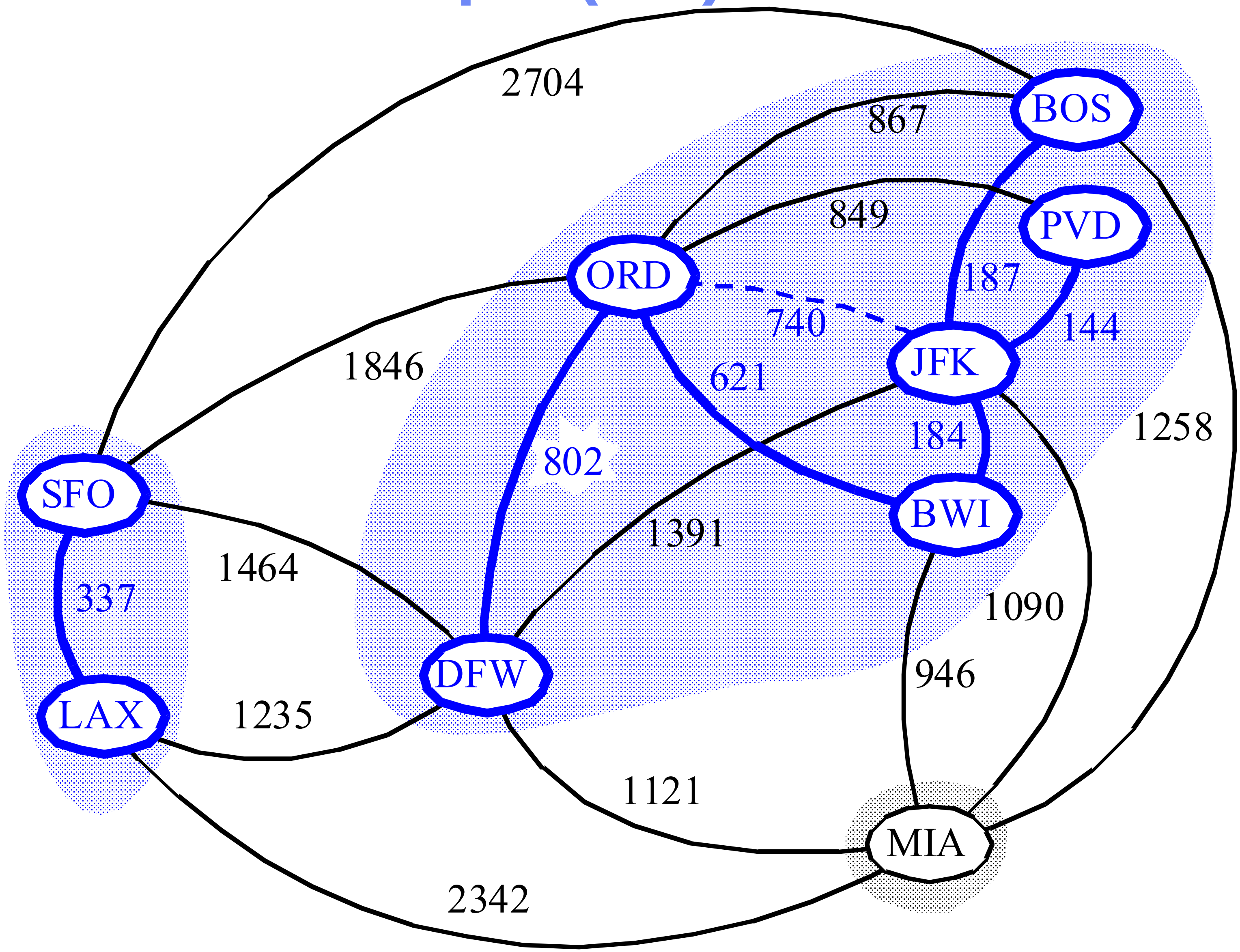
Kruskal’s example (VI)



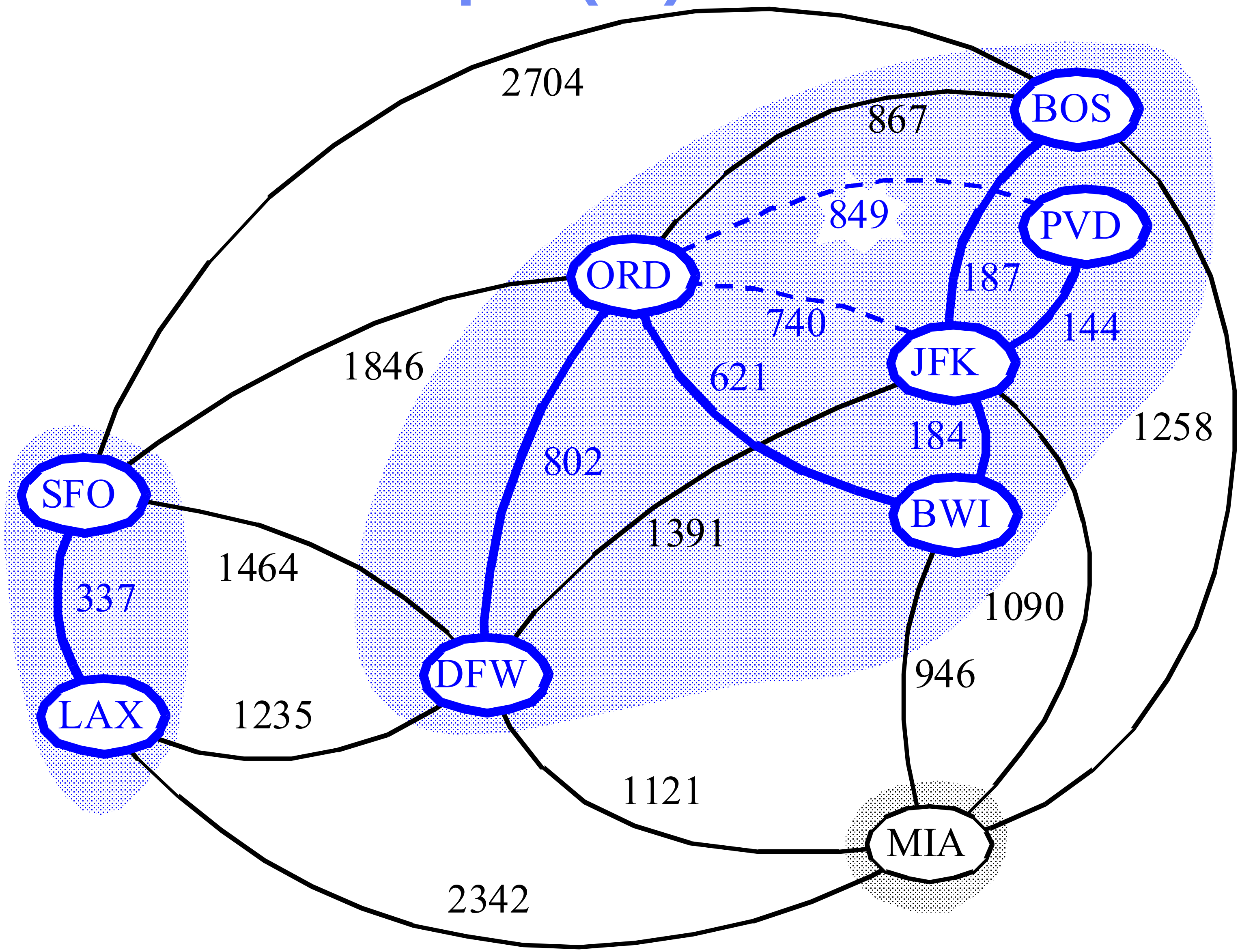
Kruskal’s example (VII)



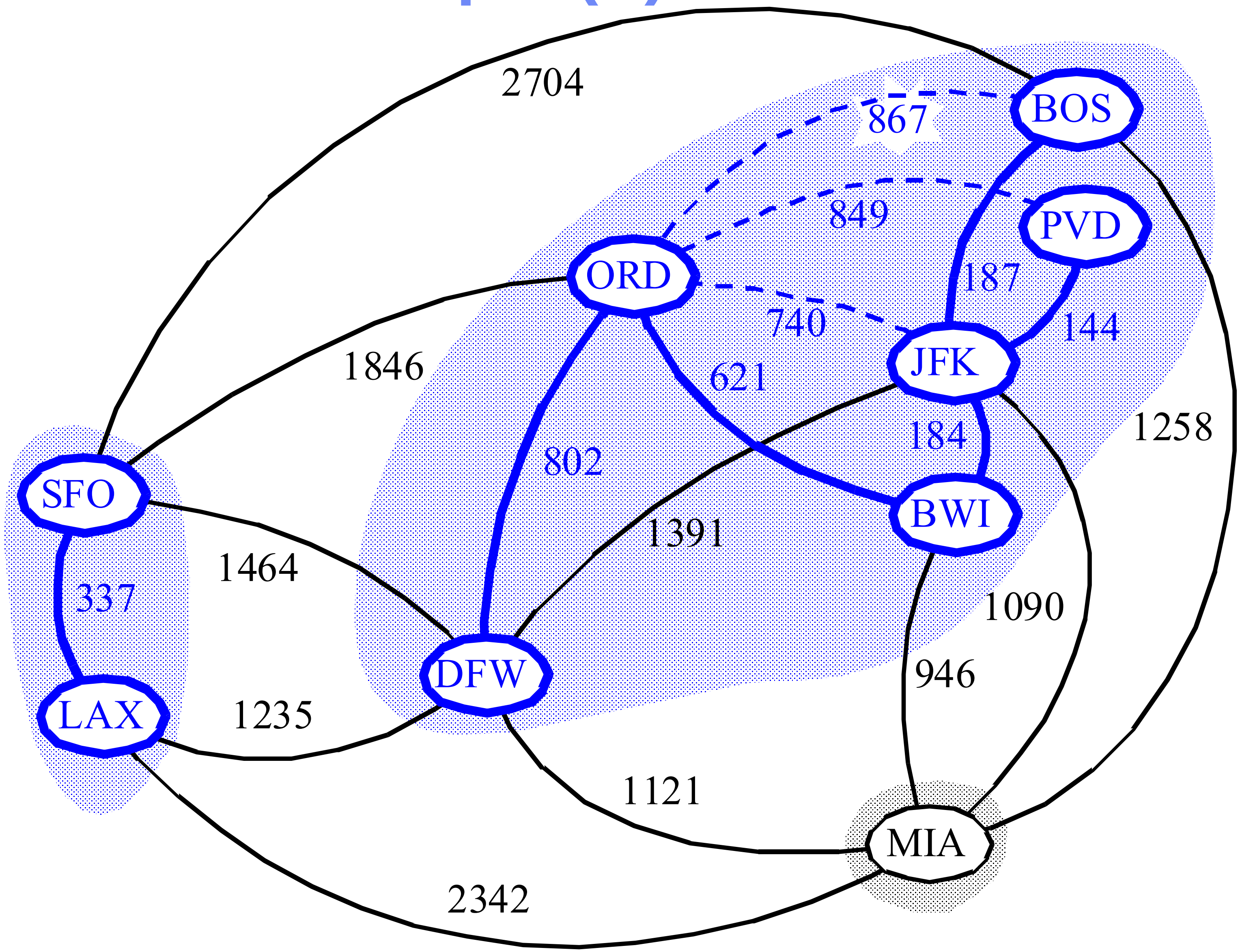
Kruskal’s example (VIII)



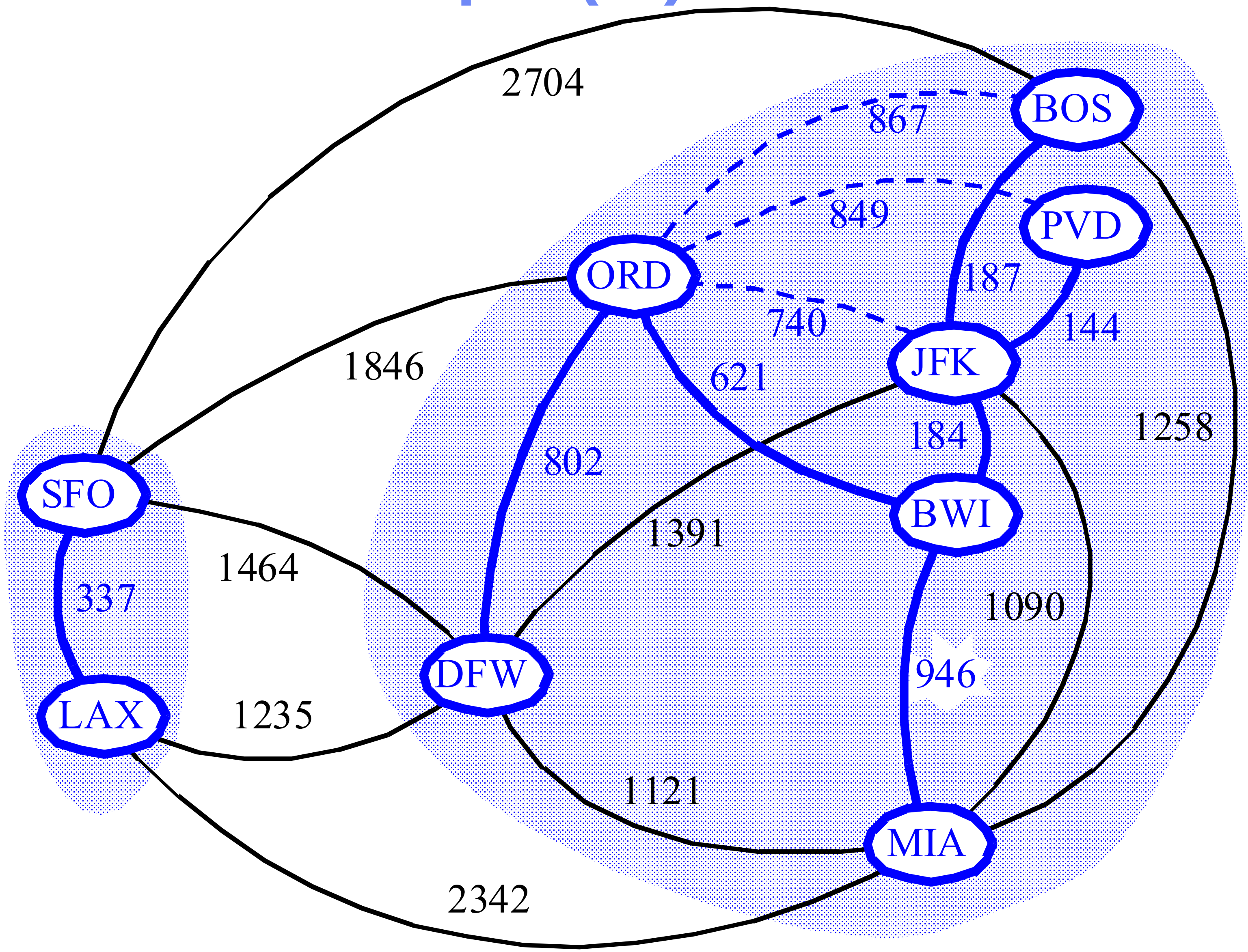
Kruskal’s example (IX)



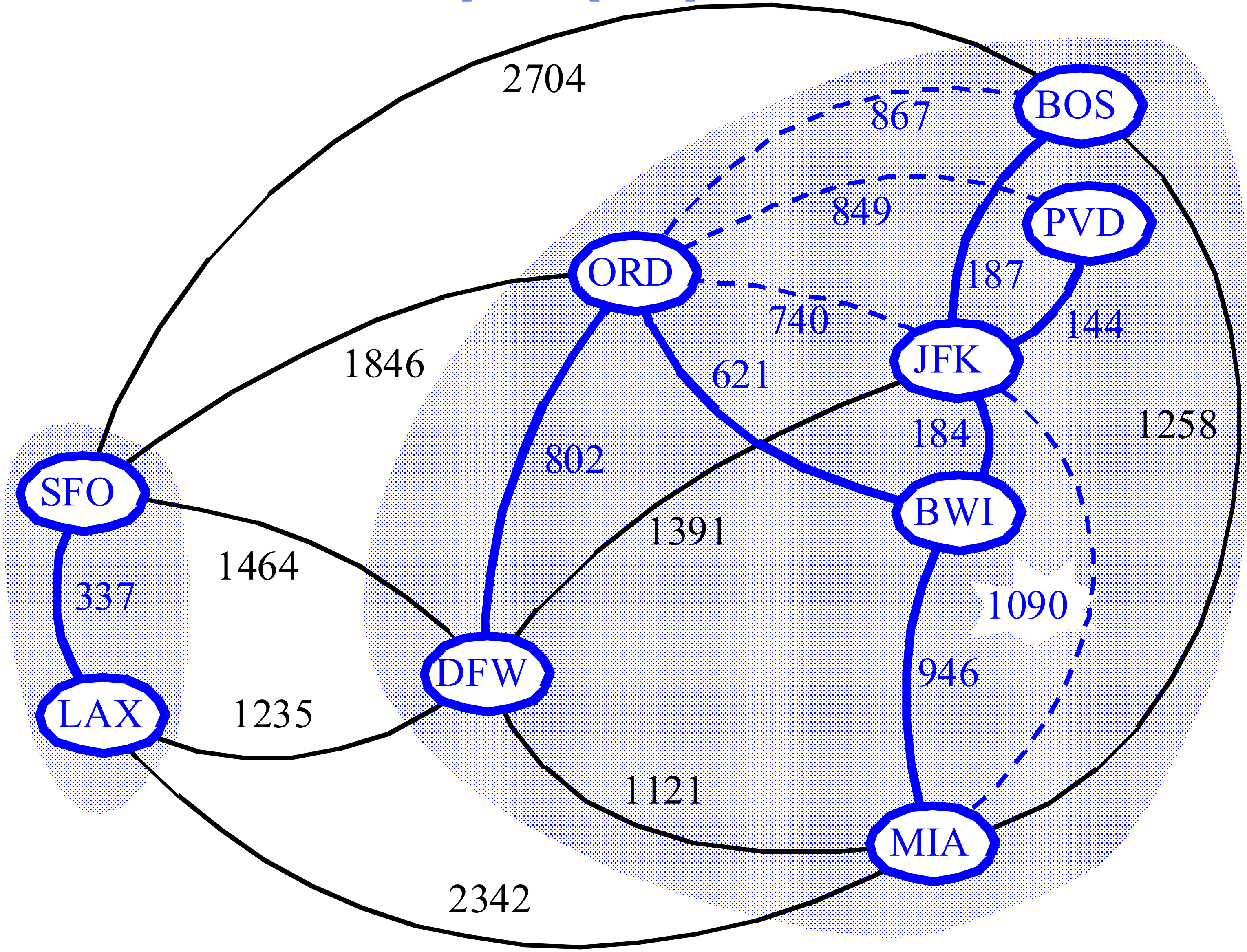
Kruskal's example (X)



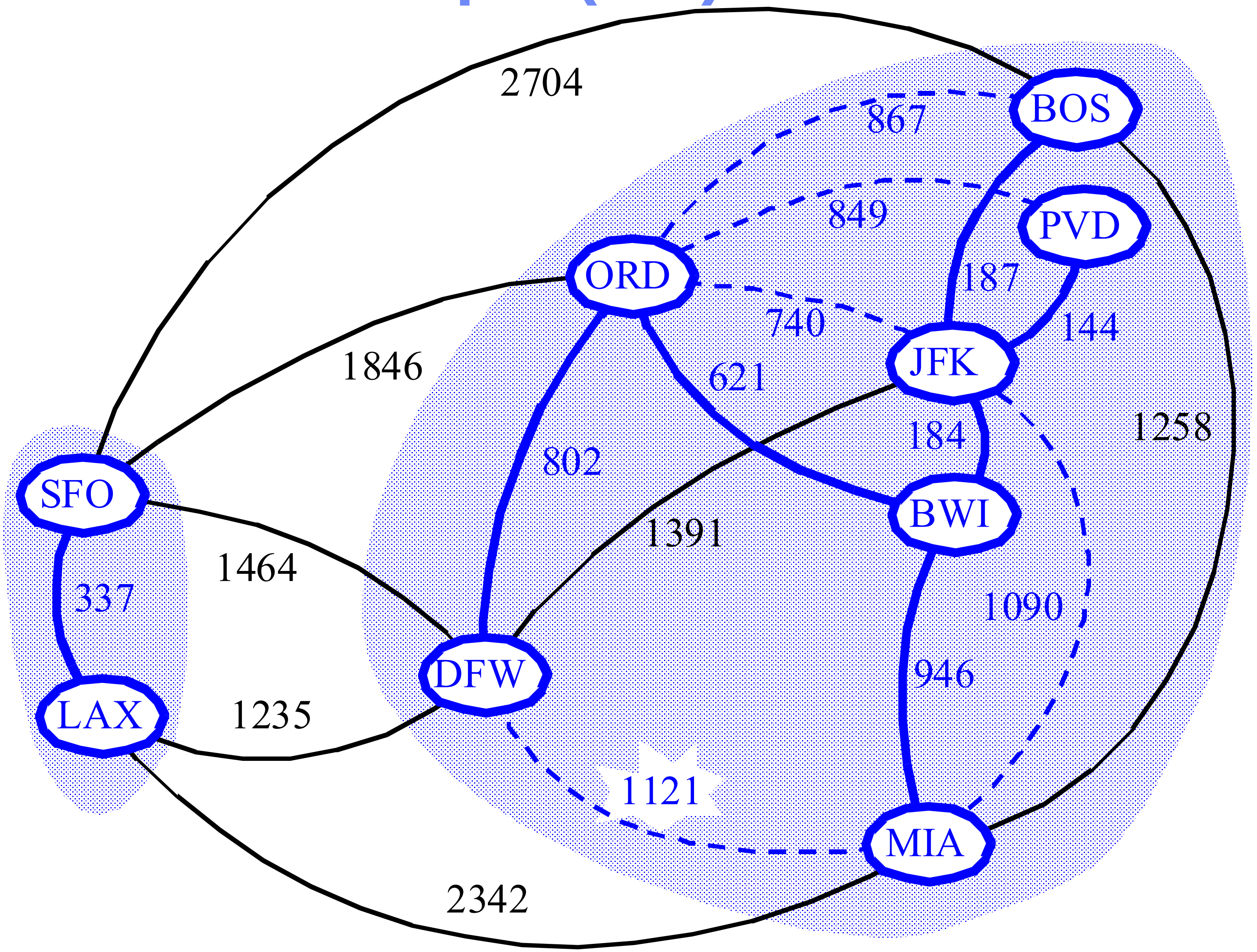
Kruskal's example (XI)



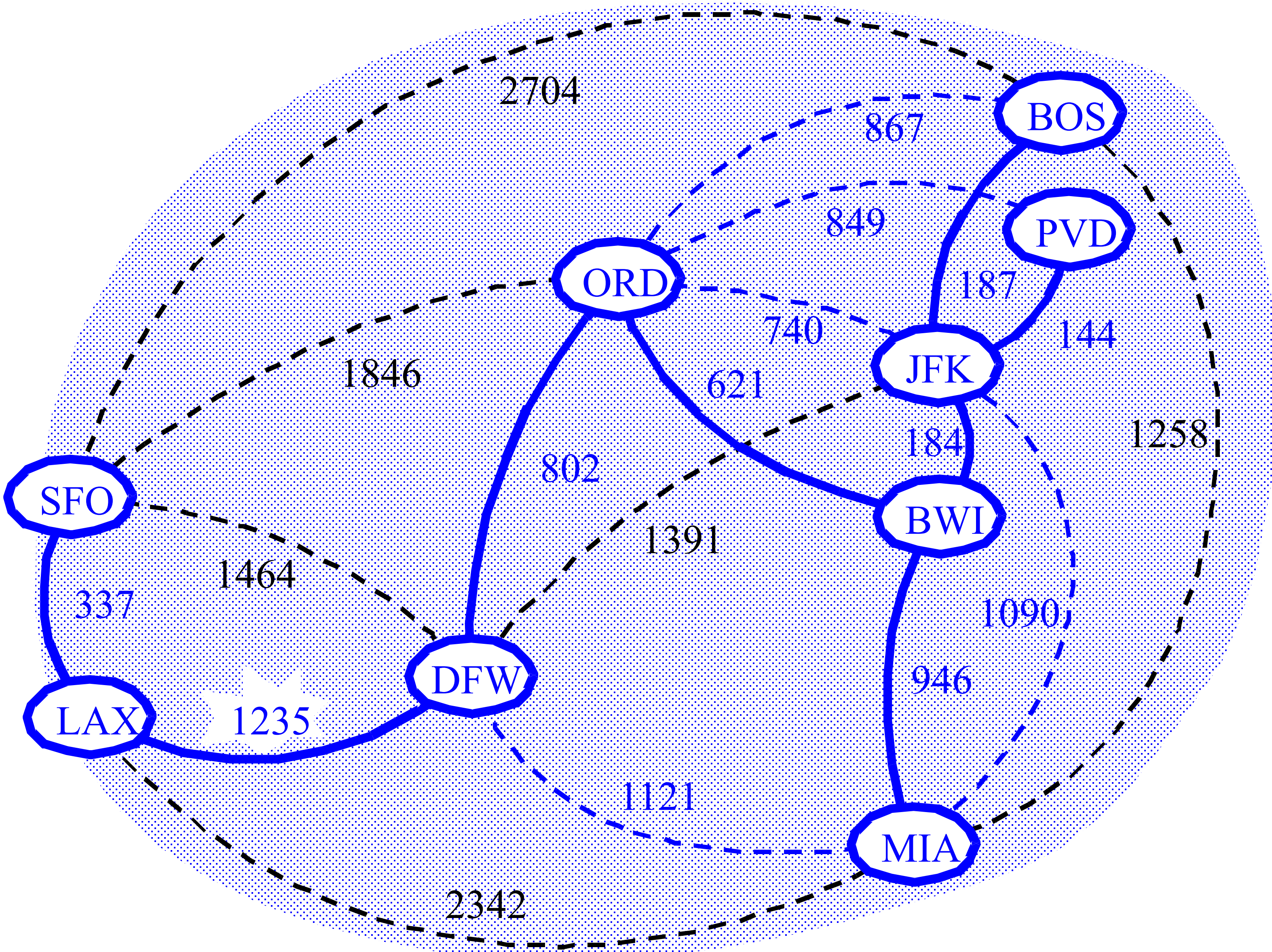
Kruskal's example (XII)



Kruskal’s example (XIII)



Kruskal's example (XIV)



Recomended activities

UNIT 4: GRAPH ALGORITHMS

Recommended readings

- Algorithm for finding all the articulation points of a graph: http://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/