CS 311: Algorithm Design and Analysis

Lecture 8

Last Lecture we have

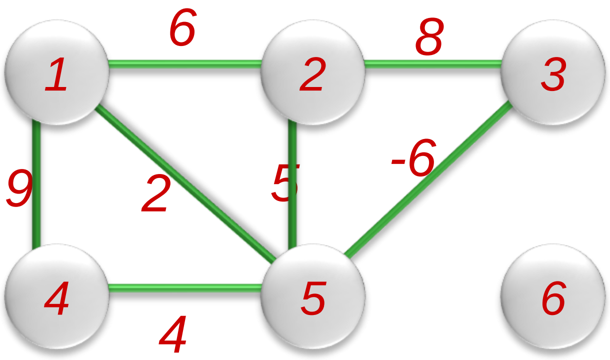
- Graph
- MST

This Lecture we have

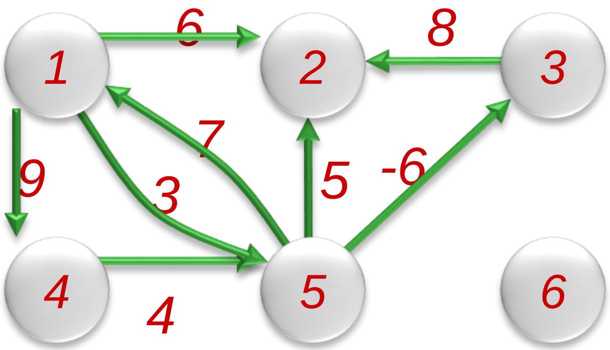
- MST
- Shortest path

Weighted Adjacency Matrix

$$A[i, j] = \begin{cases} w(i, j) & \text{if } (i, j) \in E(G) \\ 0 & \text{if } i = j, (i, j) \notin E(G) \\ \infty & \text{otherwise} \end{cases}, \quad \text{for } i, j \in V(G).$$



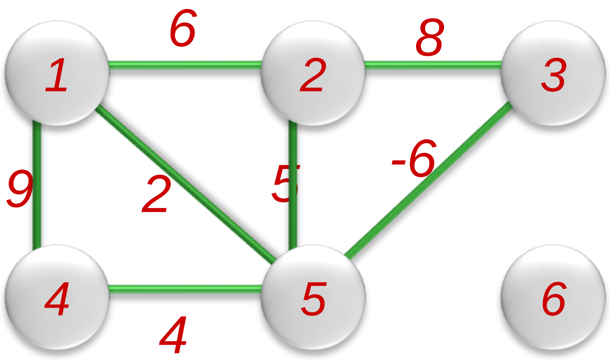
$$A = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 6 & \infty & 9 & 2 & \infty \\ 2 & 6 & 0 & 8 & \infty & 5 & \infty \\ 3 & \infty & 8 & 0 & \infty & -6 & \infty \\ 4 & 9 & \infty & \infty & 0 & 4 & \infty \\ 5 & 2 & 5 & -6 & 4 & 0 & \infty \\ 6 & \infty & \infty & \infty & \infty & \infty & 0 \end{array}$$

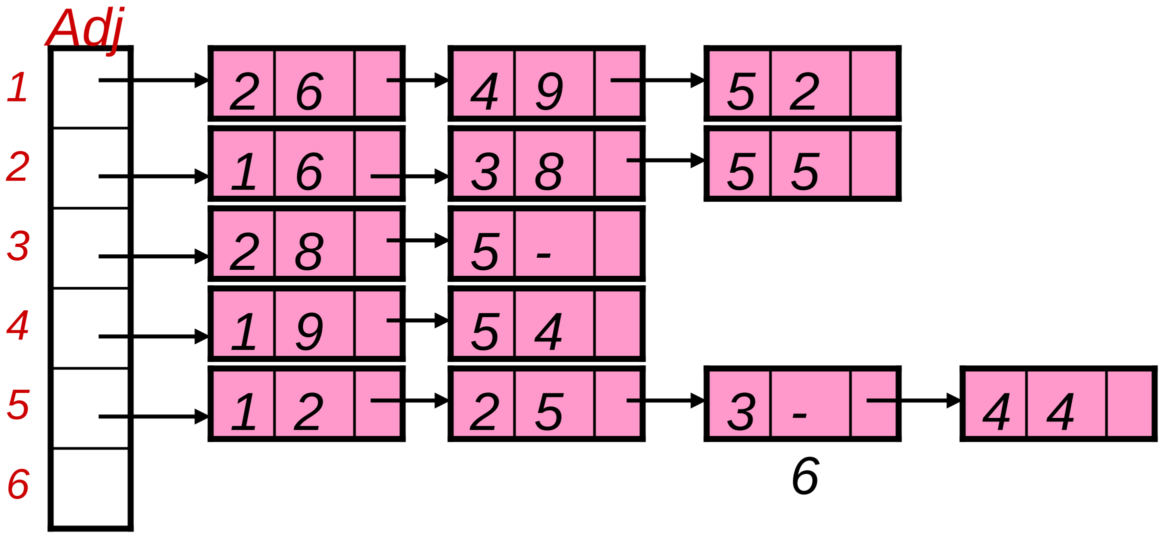


$$A = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 6 & \infty & 9 & 3 & \infty \\ 2 & \infty & 0 & \infty & \infty & \infty & \infty \\ 3 & \infty & 8 & 0 & \infty & \infty & \infty \\ 4 & \infty & \infty & \infty & 0 & 4 & \infty \\ 5 & 7 & 5 & -6 & \infty & 0 & \infty \\ 6 & \infty & \infty & \infty & \infty & \infty & 0 \end{array}$$

(Weighted) Adjacency List Structure

$$\mathrm{Adj}[i] = \{\, \langle j, w(i,j)\rangle \mid (i,j) \in E(G) \,\}, \quad \text{for } i \in V(G).$$




Adj

1 → 2 6 → 4 9 → 5 2

2 → 1 6 → 3 8 → 5 5

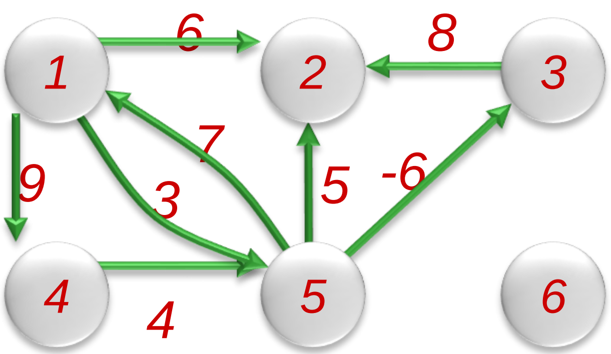
3 → 2 8 → 5 -

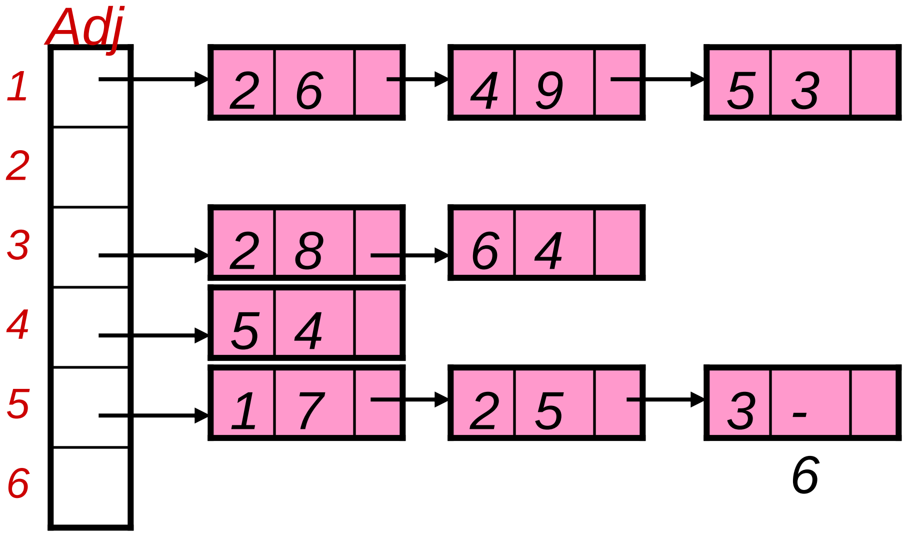
4 → 1 9 → 5 4

5 → 1 2 → 2 5 → 3 - → 4 4

6

6




Adj

1 → 2 6 → 4 9 → 5 3

2

3 → 2 8 → 6 4

4 → 5 4

5 → 1 7 → 2 5 → 3 -

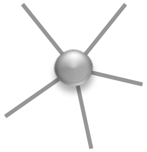
6

6

The Hand-Shaking Lemma

Vertex v ∈V(G): *degree (or valance) , in-degree, out-degree*

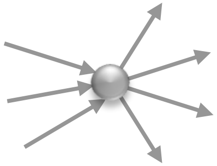
Undirected G: *deg(v) = | { u | (v,u) ∈ E(G) } | = | Adj[v] |*



Digraph G: *outdeg(v) = | { u | (v,u) ∈ E(G) } | = | Adj[v] |*

indeg(v) = | { u | (u,v) ∈ E(G) } |

deg(v) = outdeg(v) + indeg(v)



The Hand-Shaking Lemma:

For any graph (directed or undirected) we have:

$$\sum_{v \in V(G)} \deg(v) = 2\,|\,E\,|\,.$$

For any directed graph we also have:

$$\sum_{v \in V(G)} \operatorname{indeg}(v) = \sum_{v \in V(G)} \operatorname{outdeg}(v) = |\,E\,|\,.$$

Finding a MST

- Principal greedy methods: algorithms by Prim and Kruskal

- Prim
  - Grow a single tree by repeatedly adding the least cost edge that connects a vertex in the existing tree to a vertex not in the existing tree
    - Intermediary solution is a subtree

- Kruskal
  - Grow a tree by repeatedly adding the least cost edge that does not introduce a cycle among the edges included so far
    - Intermediary solution is a spanning forest

MST Applications

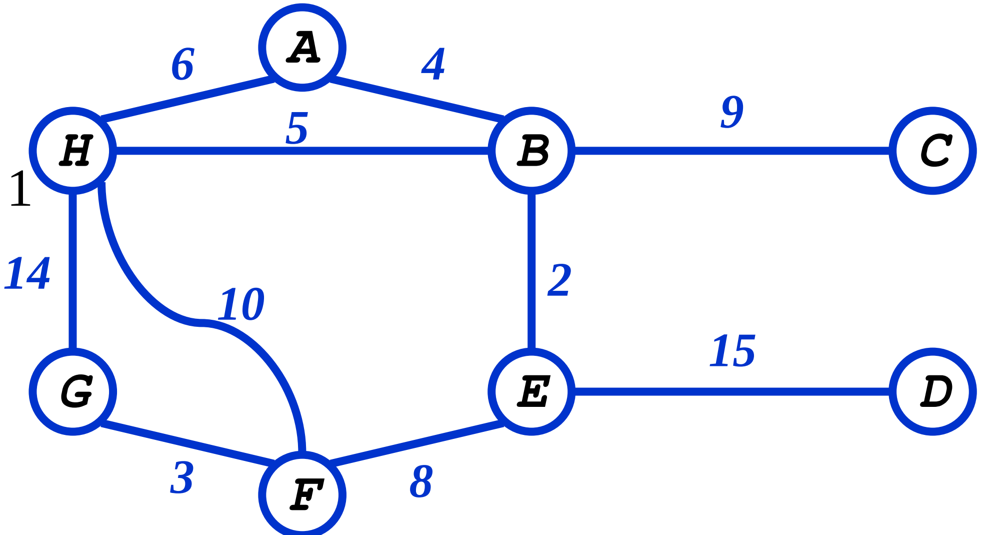
- Network design
  - telephone, electrical power, hydraulic, TV cable, computer, road

- MST gives a minimum-cost network connecting all sites

- MST: the most economical construction of a network

Kruskal's Algorithm
(High-Level Pseudocode)

```
Kruskal(G)
//Input: A weighted connected graph G = <V, E>
//Output: $E_T$ --- the set of edges composing MST of G
Sort E in nondecreasing order of the edge weight
$E_T = \varnothing$; encounter = 0          //initialize the set of tree edges and its size
k = 0                                       //initialize the number of processed edges
while encounter < |V| - 1
        k = k+1
        if $E_T \cup \{e_k\}$ is acyclic
           $E_T = E_T \cup \{e_k\}$;
           encounter = encounter + 1
return $E_T$
```



Kruskal’s Algorithm

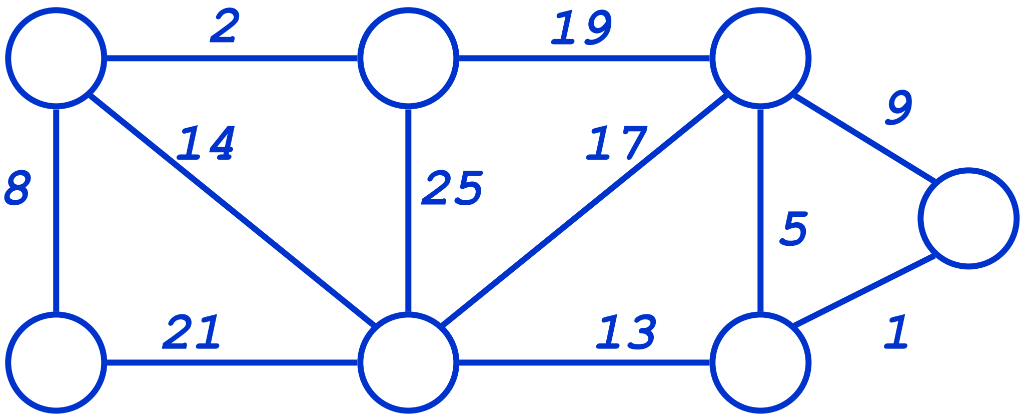
Grow a tree by repeatedly adding the least cost edge that does not introduce a cycle among the edges included so far

Intermediary solution is a spanning forest

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

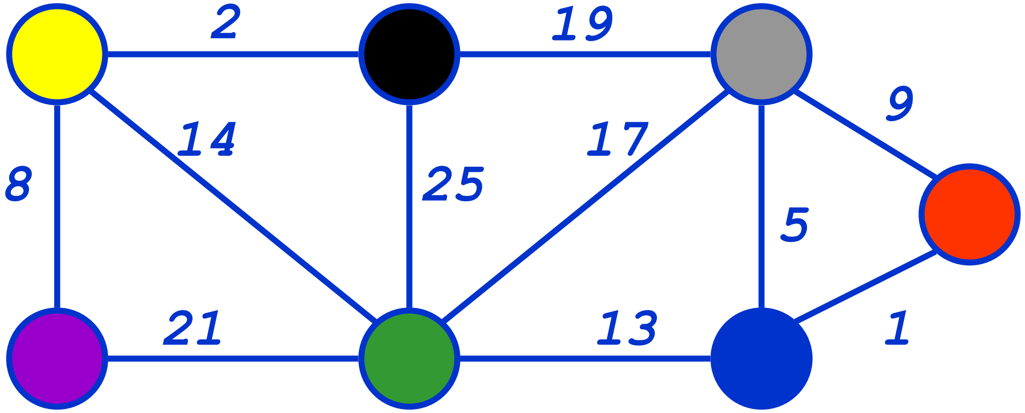
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

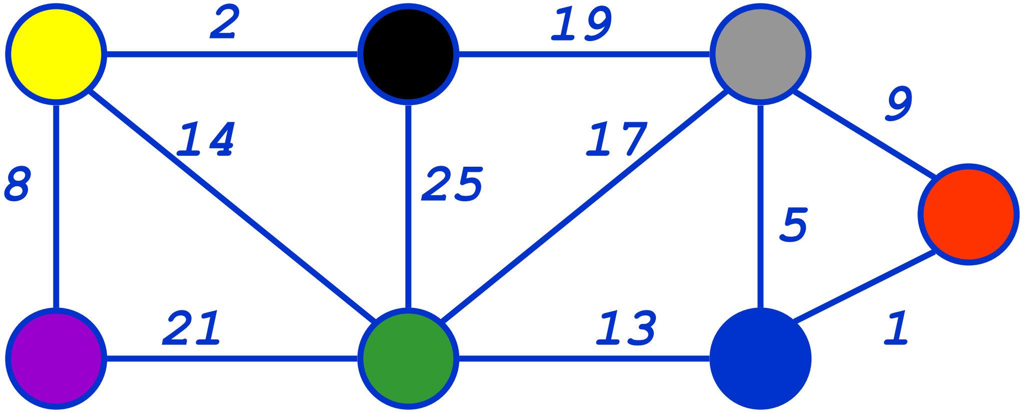
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

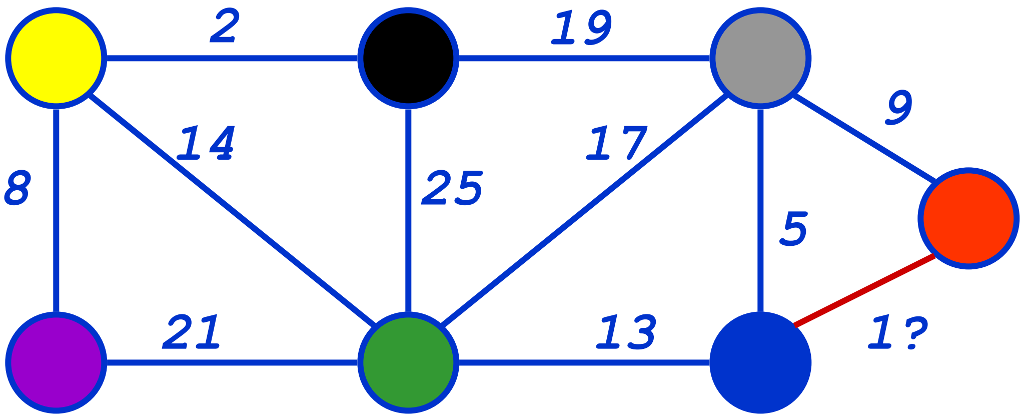
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

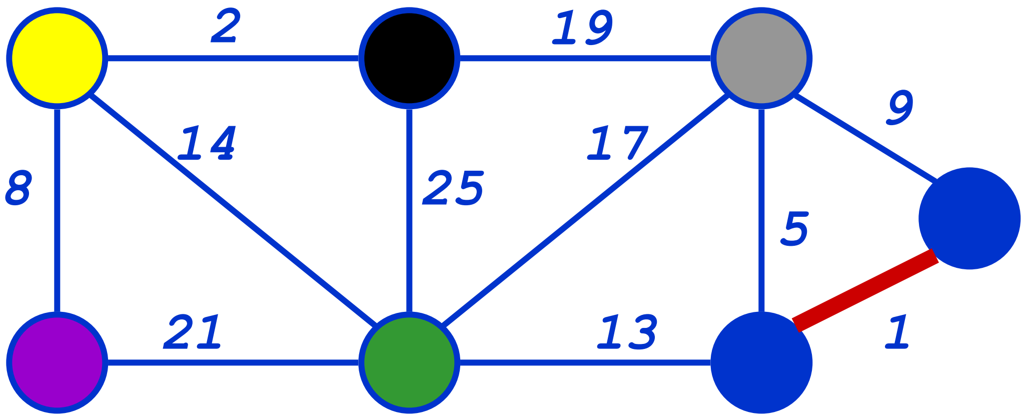
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

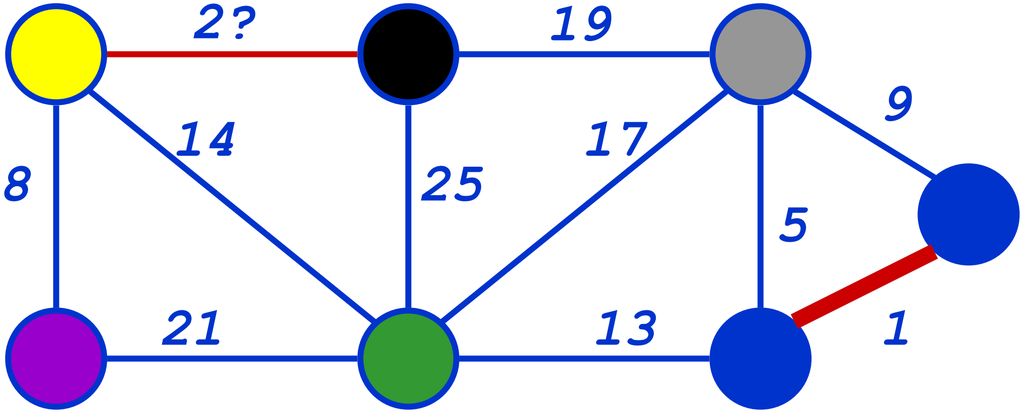
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

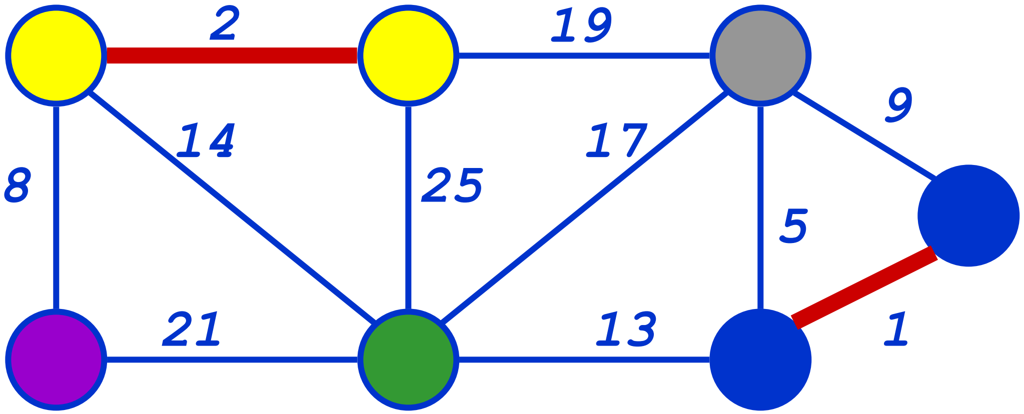
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
       MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
       if FindSet(u) ≠ FindSet(v)
          T = T U {{u,v}};
          Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

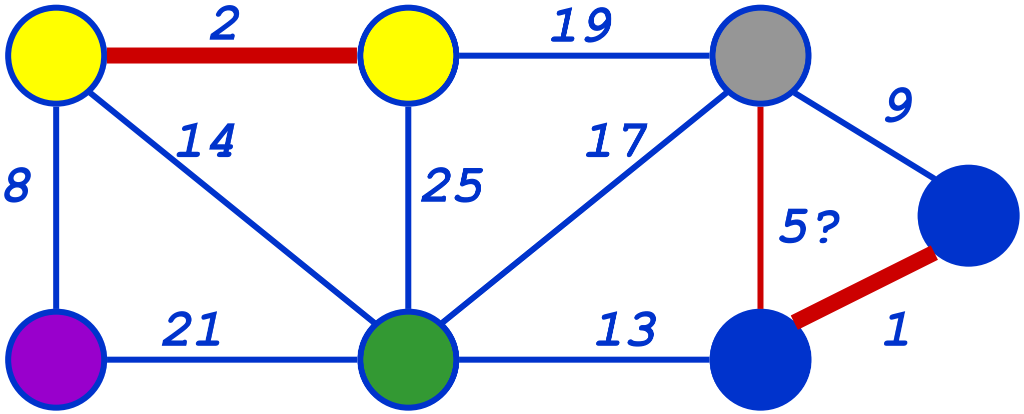
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

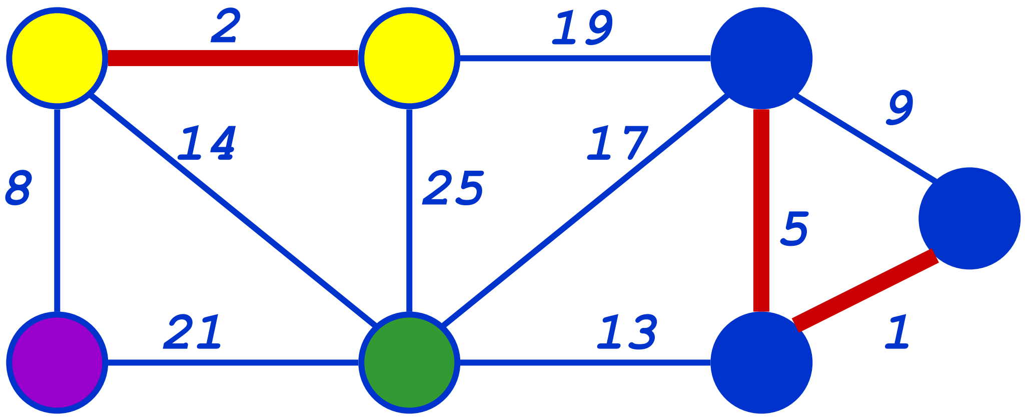
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

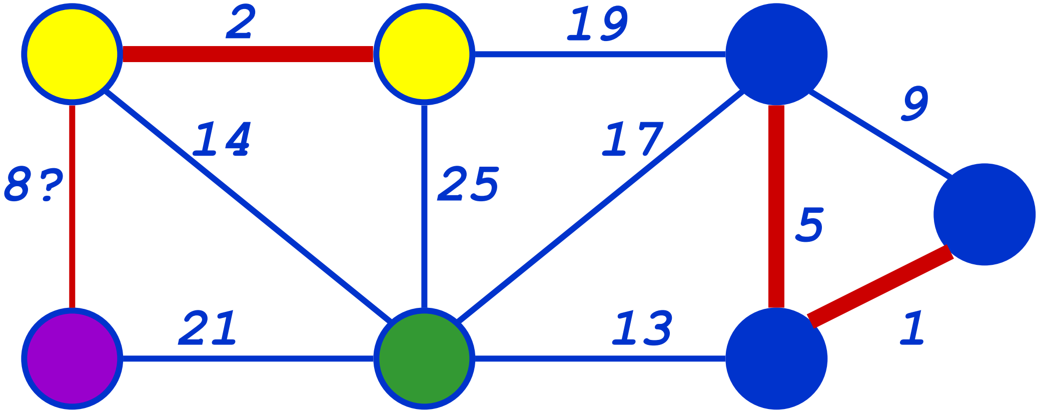
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

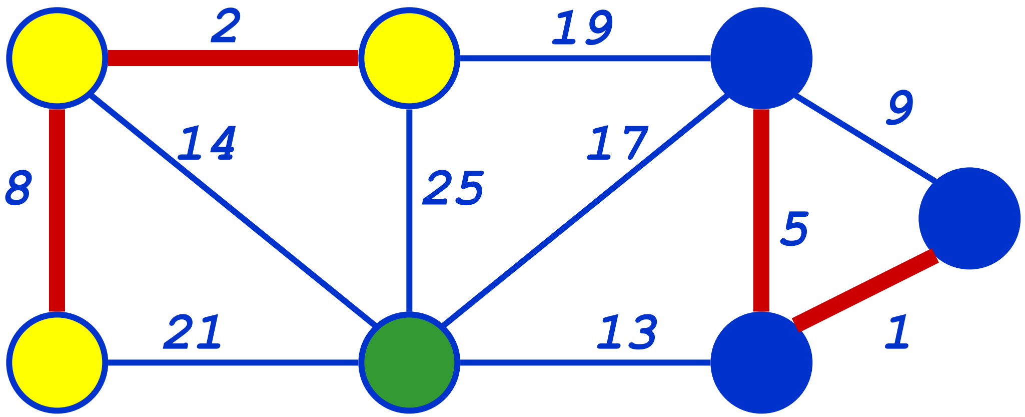


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

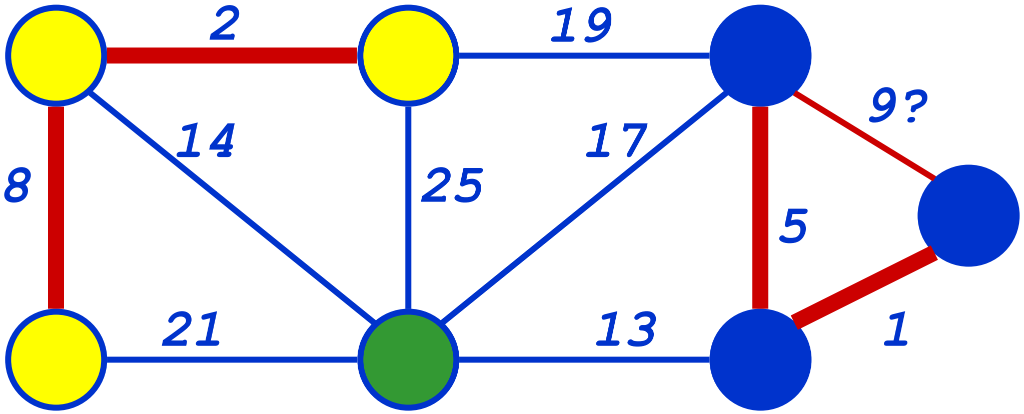
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
       MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
       if FindSet(u) ≠ FindSet(v)
          T = T U {{u,v}};
          Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

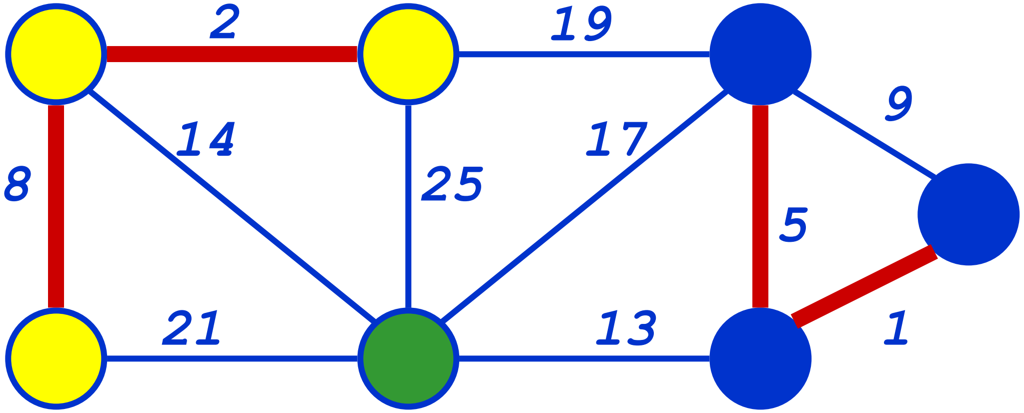
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

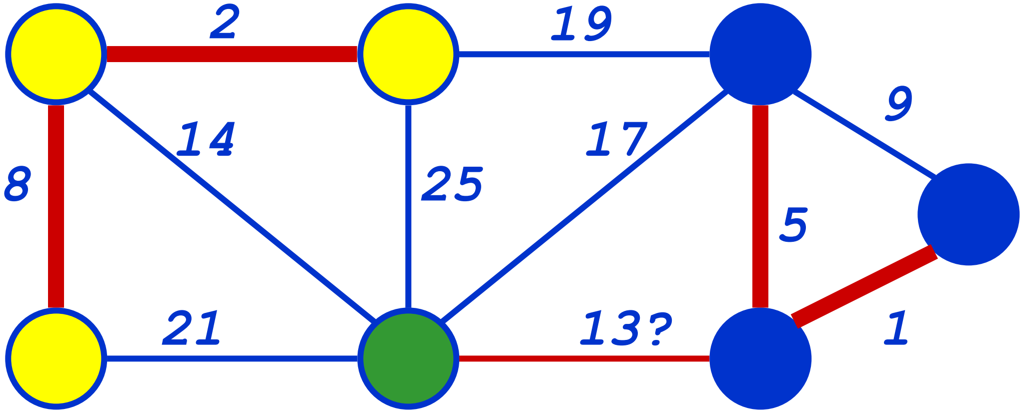
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

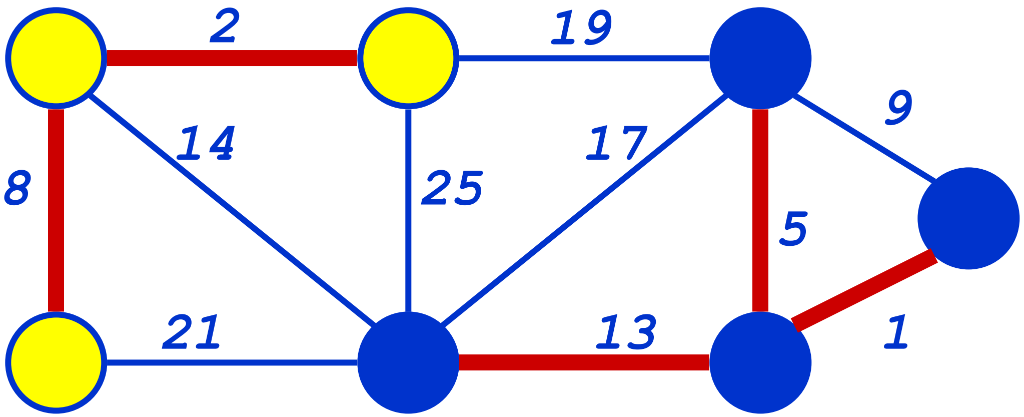
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

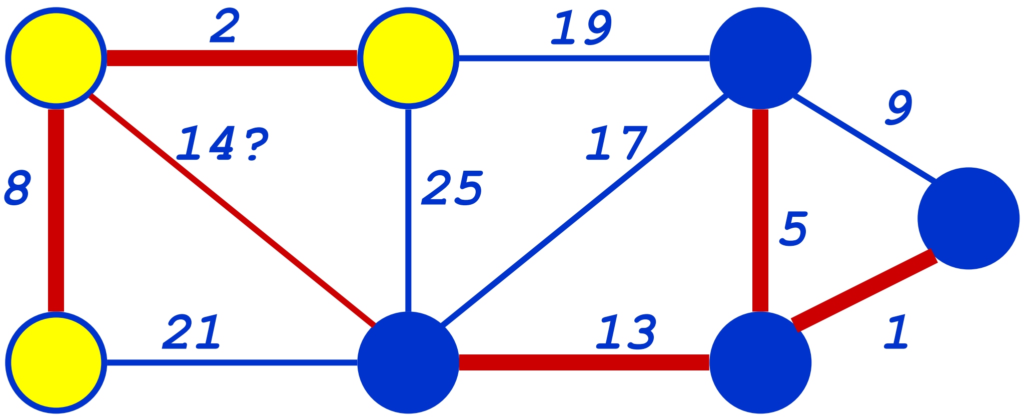


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
       MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
       if FindSet(u) ≠ FindSet(v)
          T = T U {{u,v}};
          Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

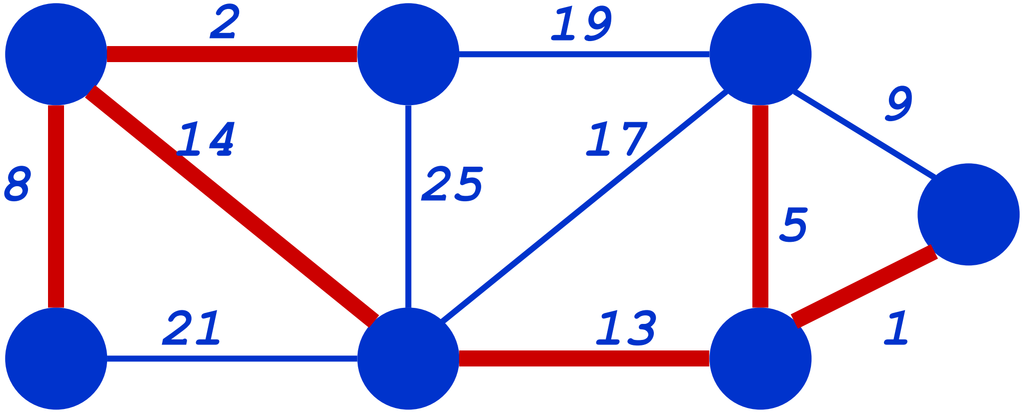
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

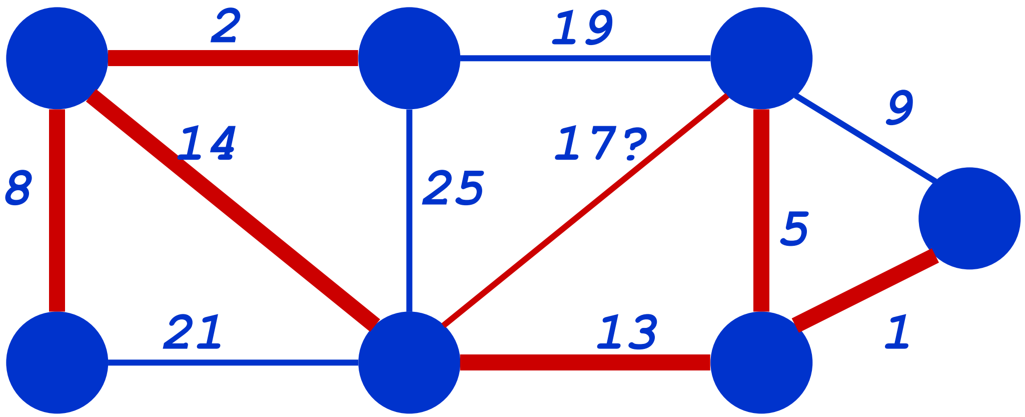
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
       MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
       if FindSet(u) ≠ FindSet(v)
          T = T U {{u,v}};
          Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

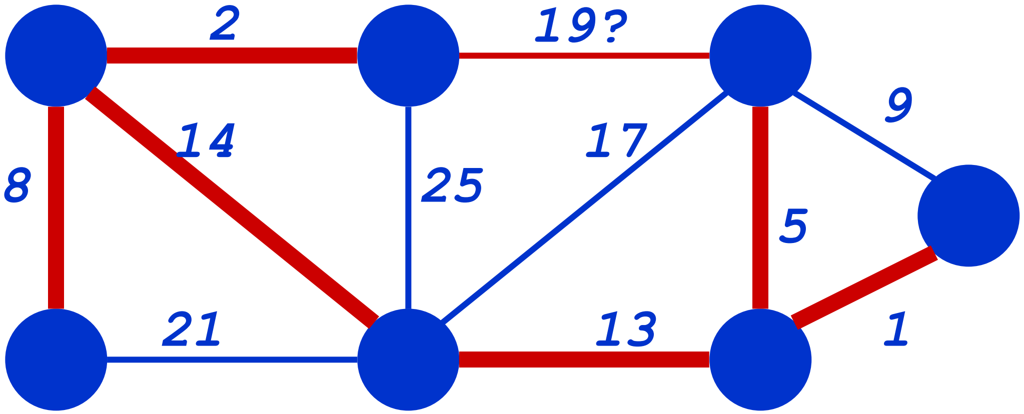
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

Run the algorithm:

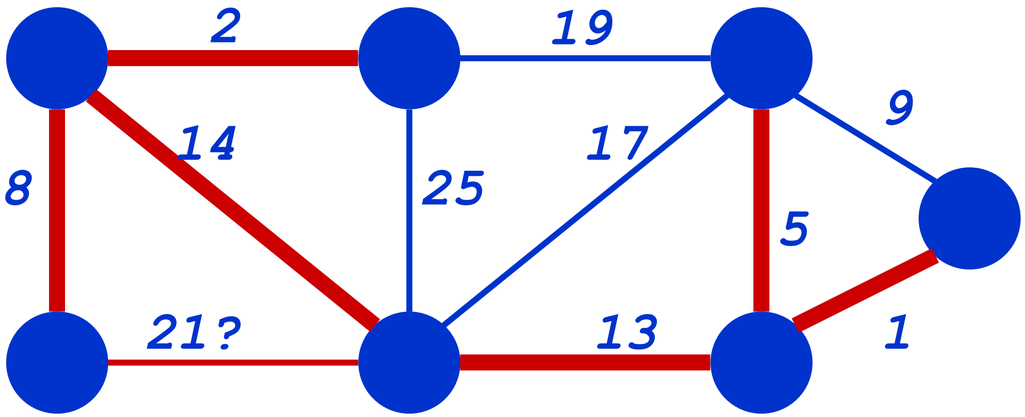


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

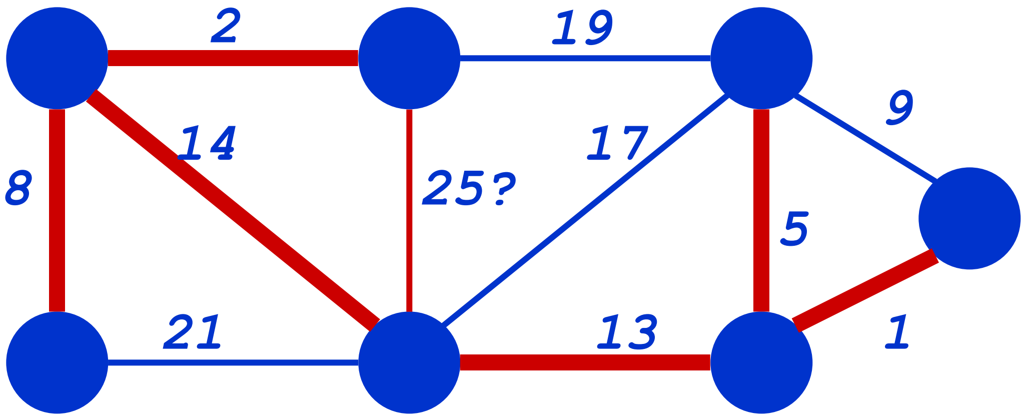
```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal’s Algorithm

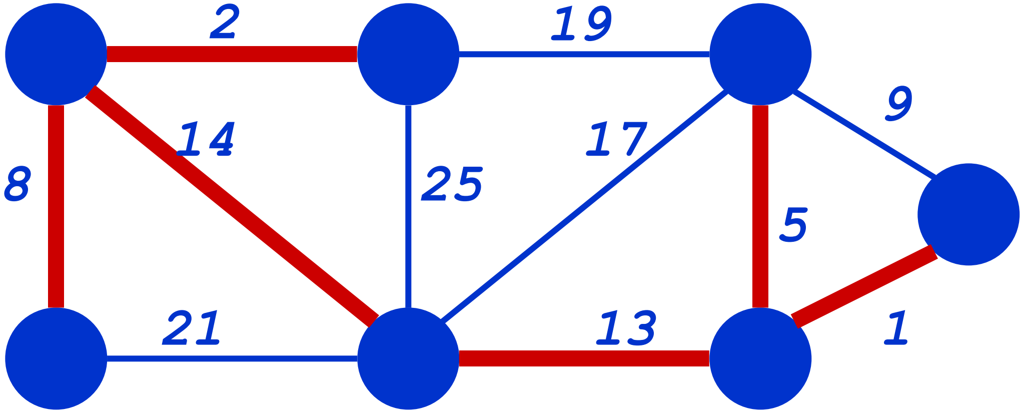
Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

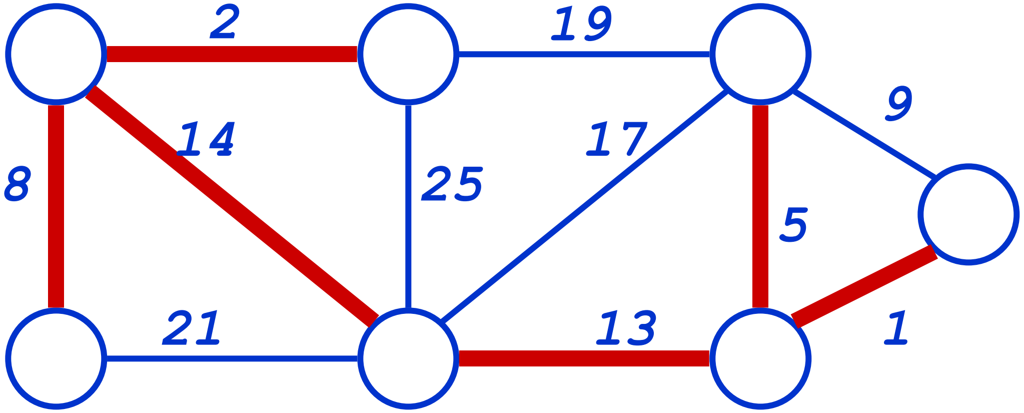
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

Run the algorithm:



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?

Let n=|V| and m=|E|

O(n) MakeSet() calls

1 Sort

O(m) FindSet() calls

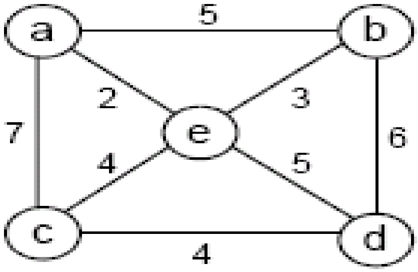
O(n) Union() calls

Kruskal’s Algorithm: Running Time

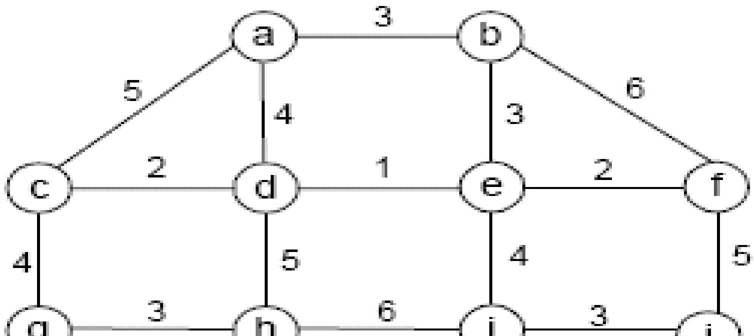
- To summarize:
  - Sort edges: O(m lg m)
  - O(n) MakeSet()’s
  - O(m) FindSet()’s
  - O(n) Union()’s
- Upshot:
  - Best disjoint subsets union-find algorithm makes above 3 operations only slightly worse than linear
  - Refer to the textbook for a discussion on the algorithms
  - Overall thus O(m lg m)

In-Class Exercise

7. a. Apply Prim's algorithm to the following graph. Include in the priority queue all the vertices not already in the tree.



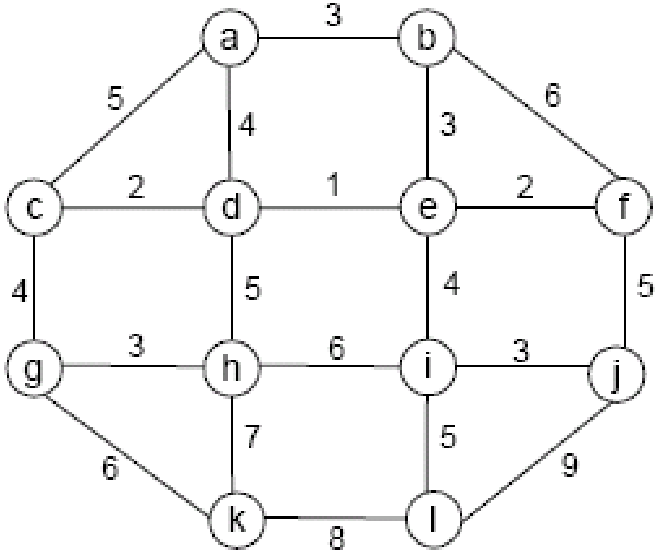
b. Apply Prim's algorithm to the following graph. Include in the priority queue only the fringe vertices (the vertices not in the current tree which are adjacent to at least one tree vertex).



In-Class Exercise

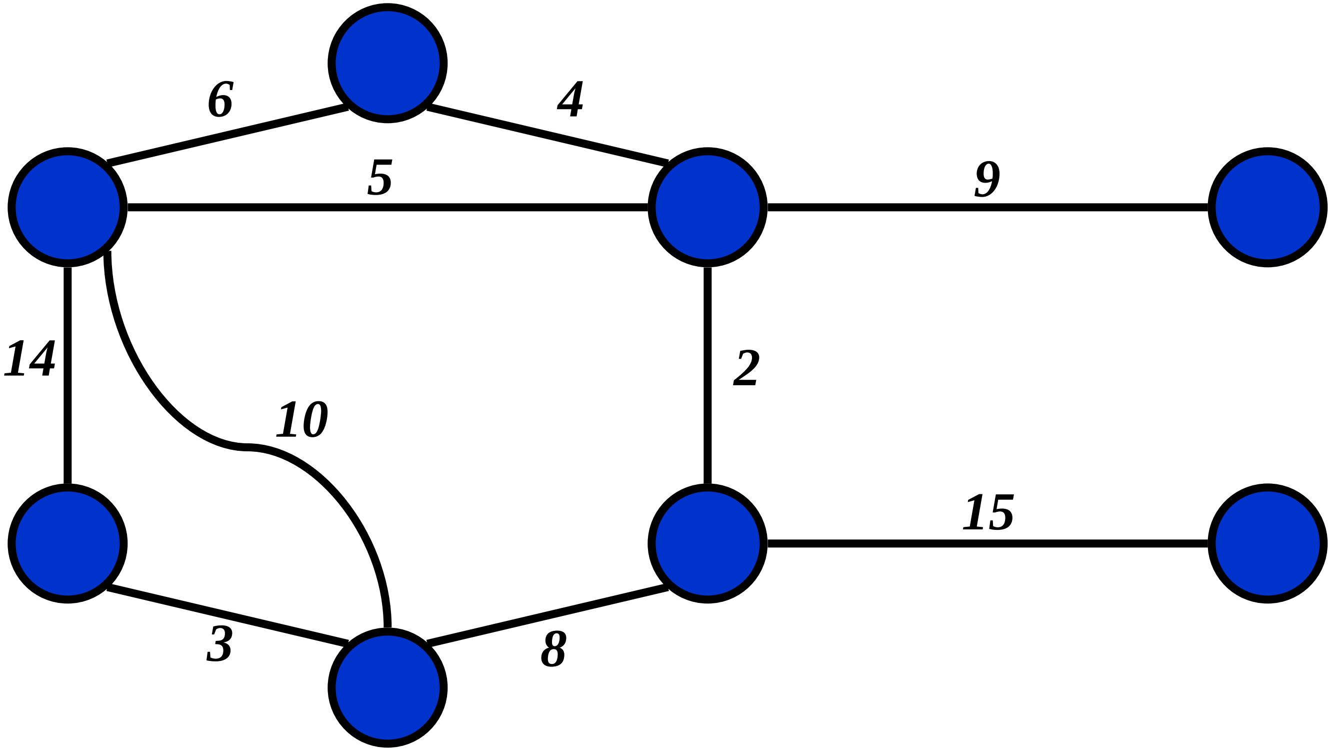
- Applying Kruskal’s algorithm

b.



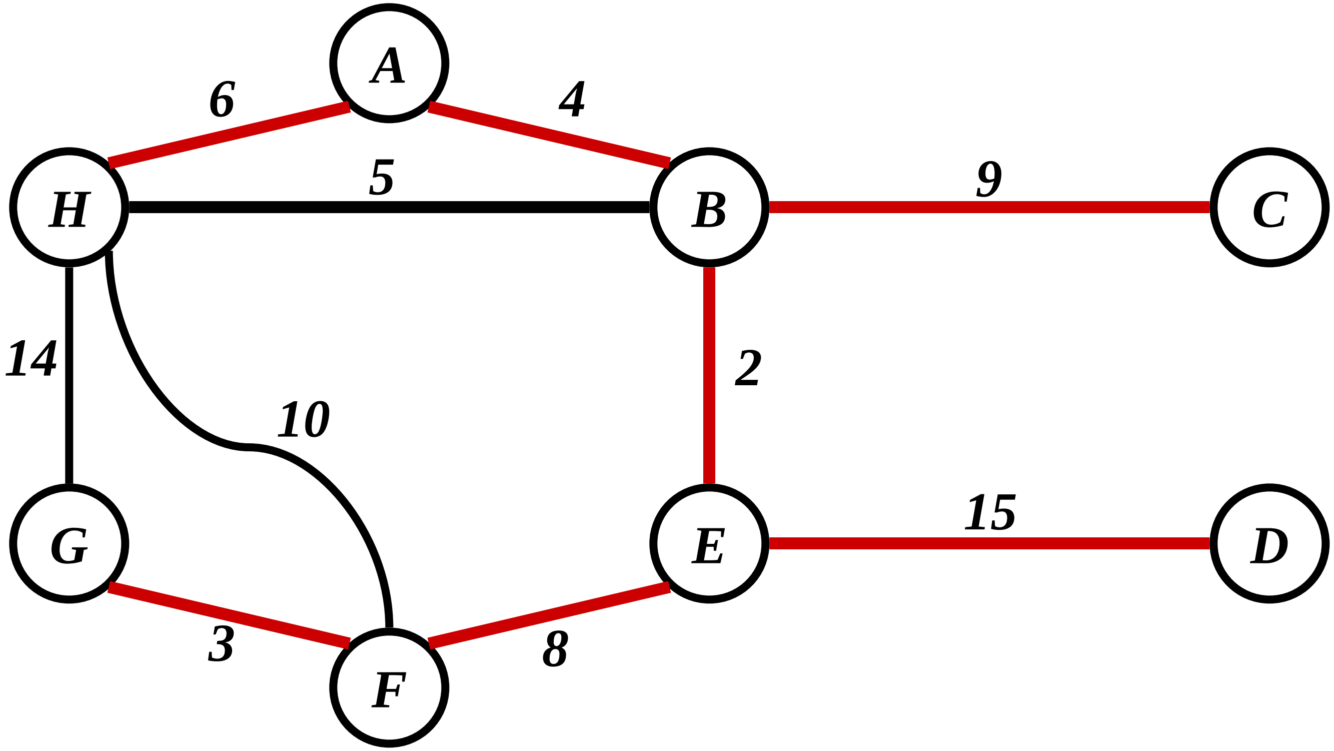
Single-Source Shortest-Paths Problem

- Problem: given a connected graph G with non-negative weights on the edges and a source vertex s in G, determine the shortest paths from s to all other vertices in G.
- Useful in many applications (e.g., road map applications)



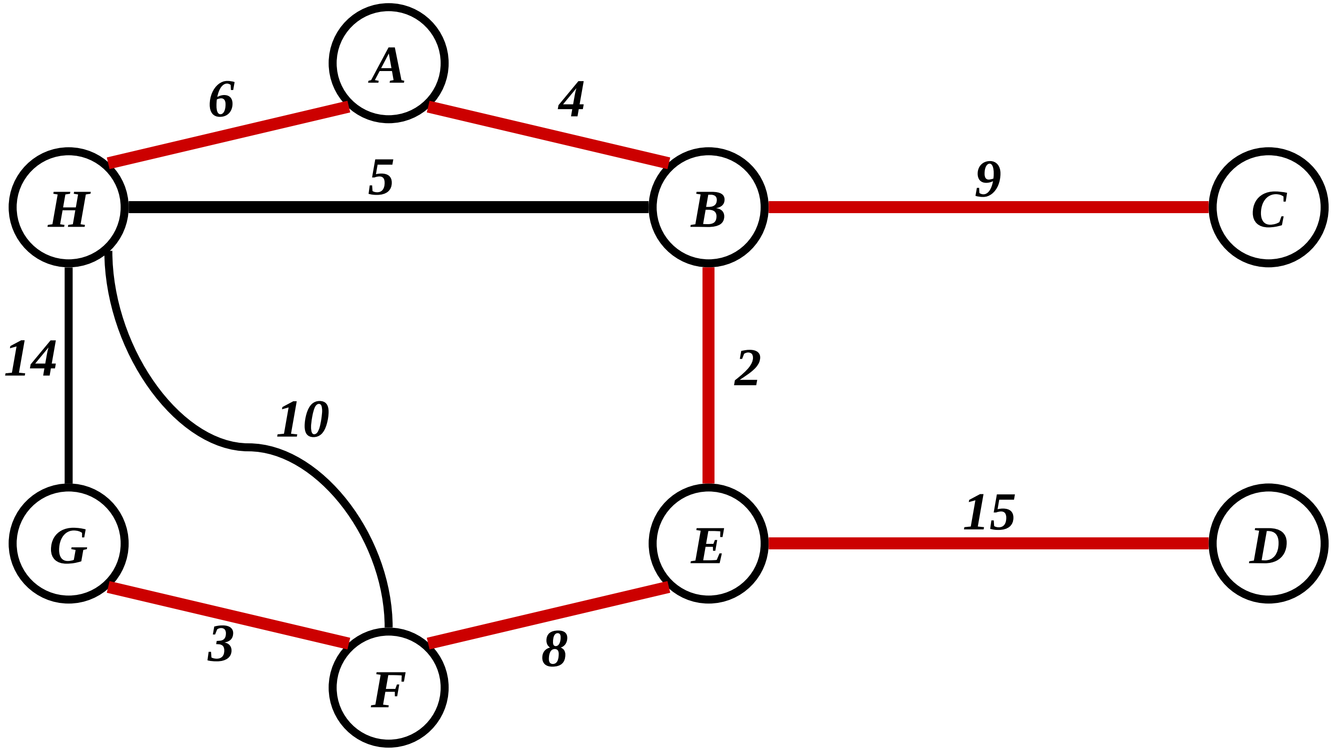
Single-Source Shortest-Paths

- For instance, the following edges form the shortest paths from node A



Single-Source Shortest-Paths

- ***Hint: does this problem looks similar?***
- ***Can we modify Prim’s algorithm to solve this problem?***

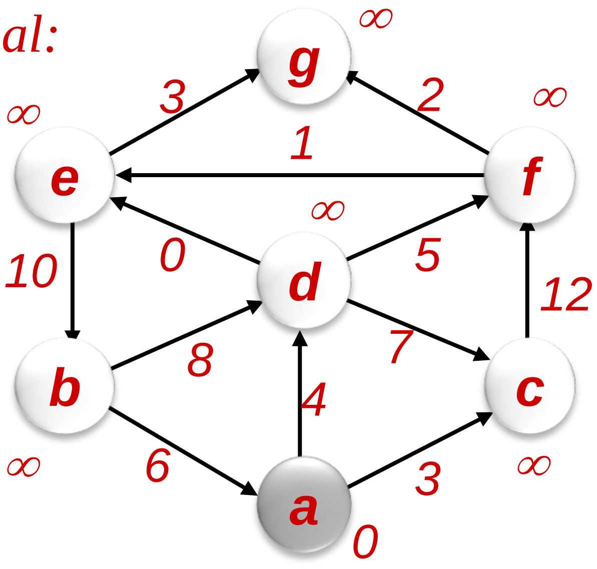


Dijkstra’s Algorithm

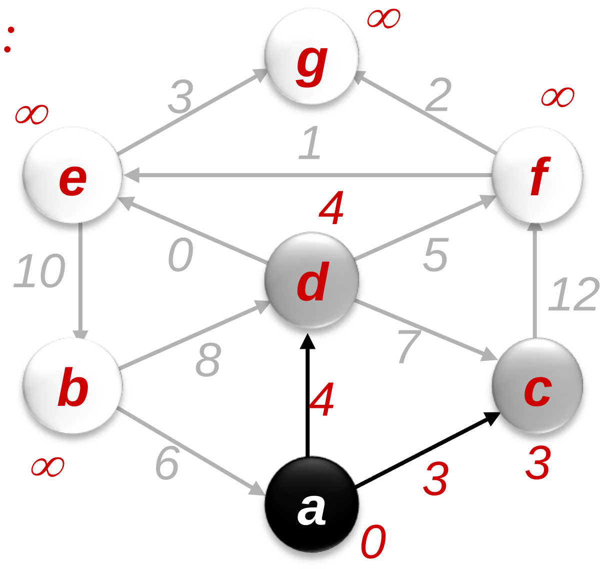
- Solves the single-source shortest paths problem
- Involves keeping a table of current shortest path lengths from source vertex (initialize to infinity for all vertices except s, which has length 0)
- Repeatedly select the vertex u with shortest path length, and update other lengths by considering the path that passes through that vertex
- Stop when all vertices have been selected
- Could use a priority queue to facilitate selection of shortest lengths
  - Here, priority is defined differently from that of the Prim’s algorithm
  - Like Prim’s algorithm, it needs to refine data structure so that the update of key-values in priority queue is allowed
  - Like Prim’s algorithm, time complexity is O((n + m) log n)

Example

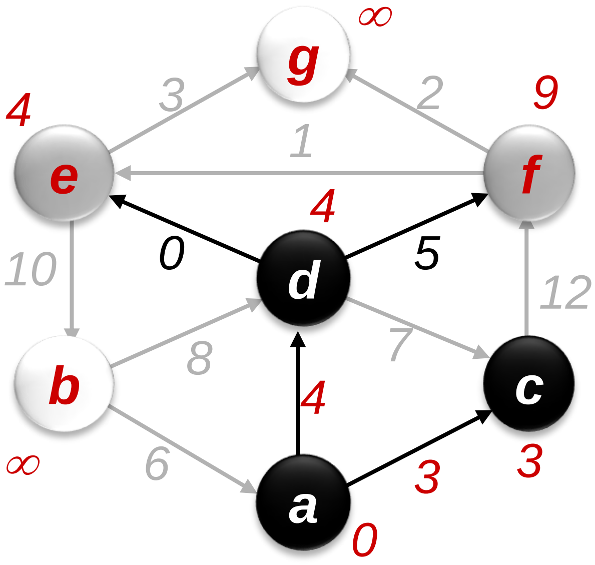
0) Initial:



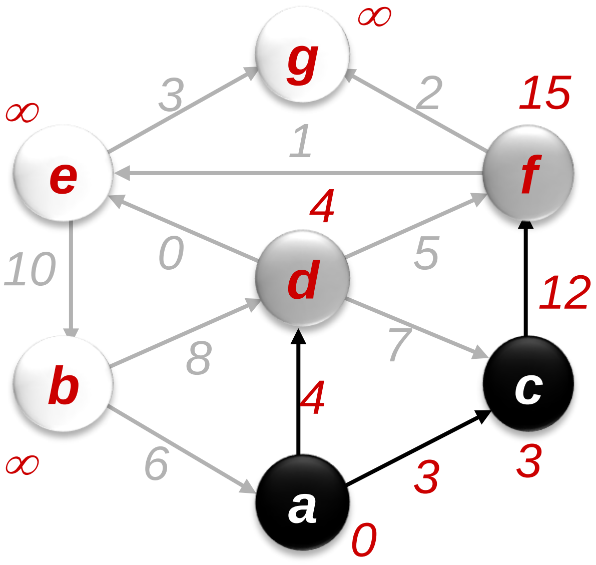
1) Scan a:



3) Scan d:

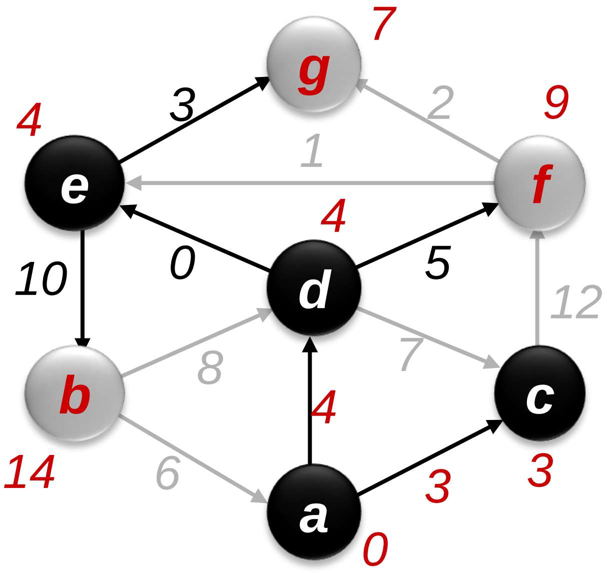


2) Scan c:

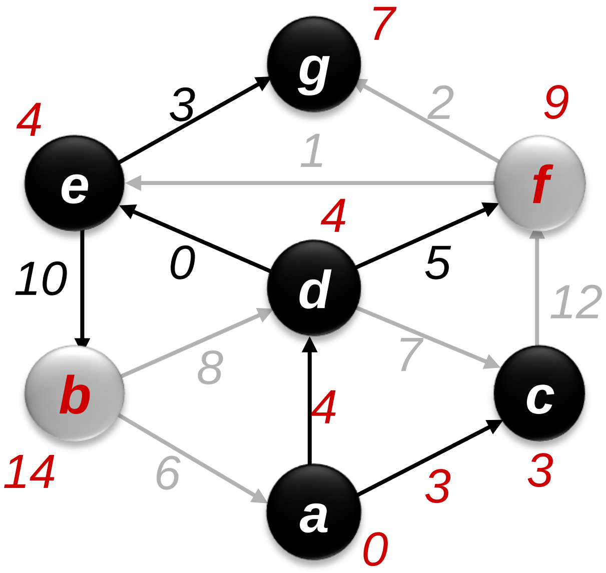


Example

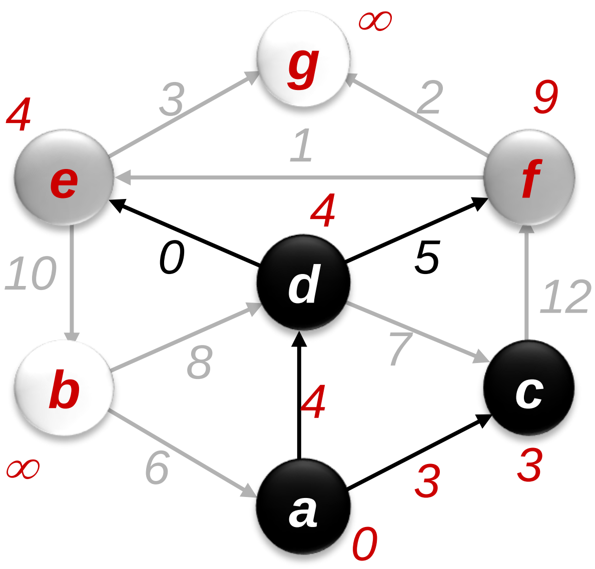
4) Scan e:



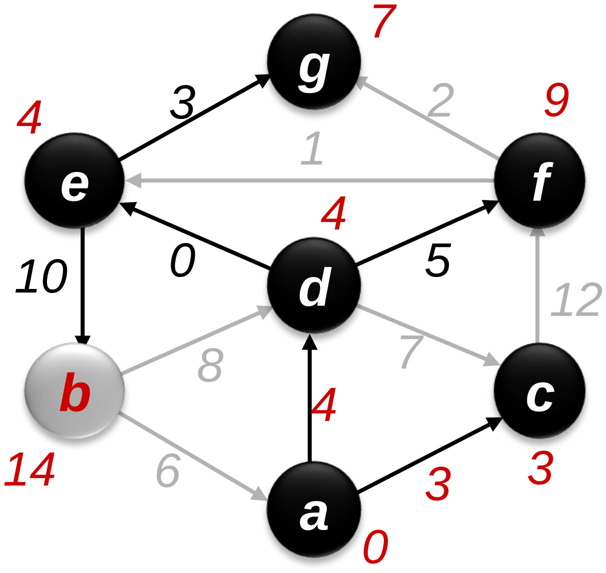
5) Scan g:



3) Scan d:

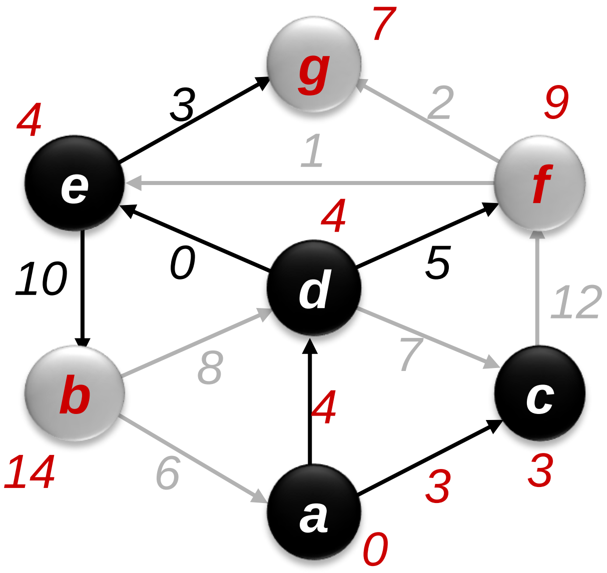


6) Scan f:

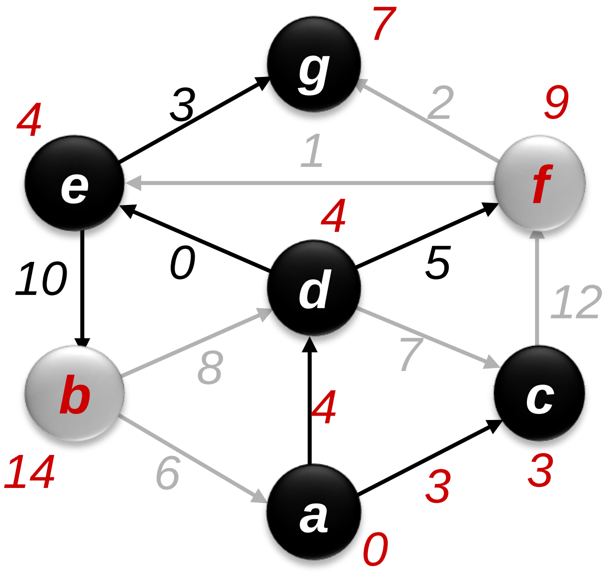


Example

4) Scan e:

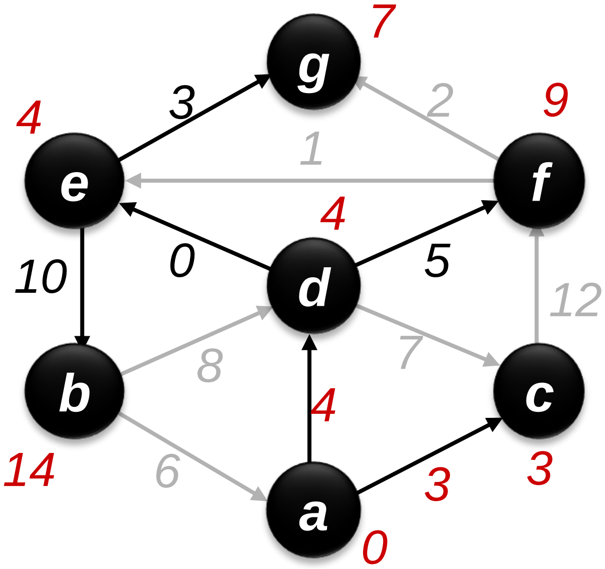


5) Scan g:

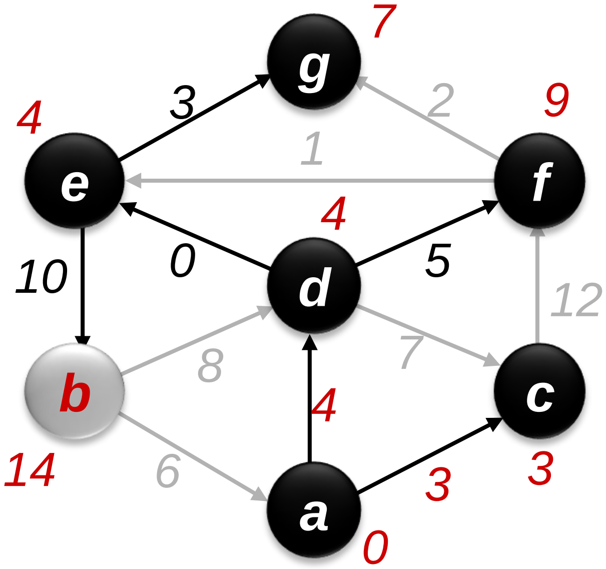


7) Scan b:

DONE



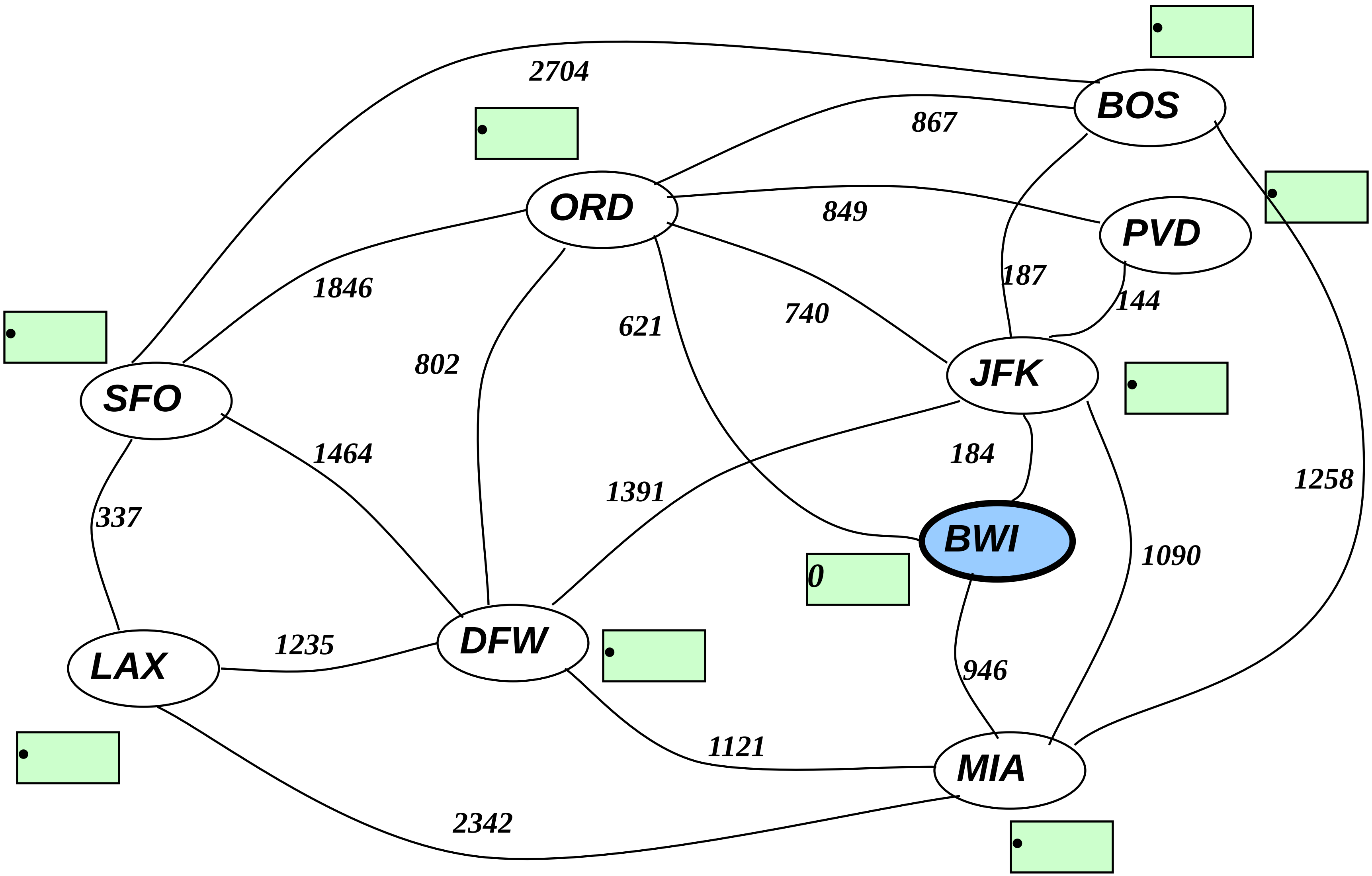
6) Scan f:



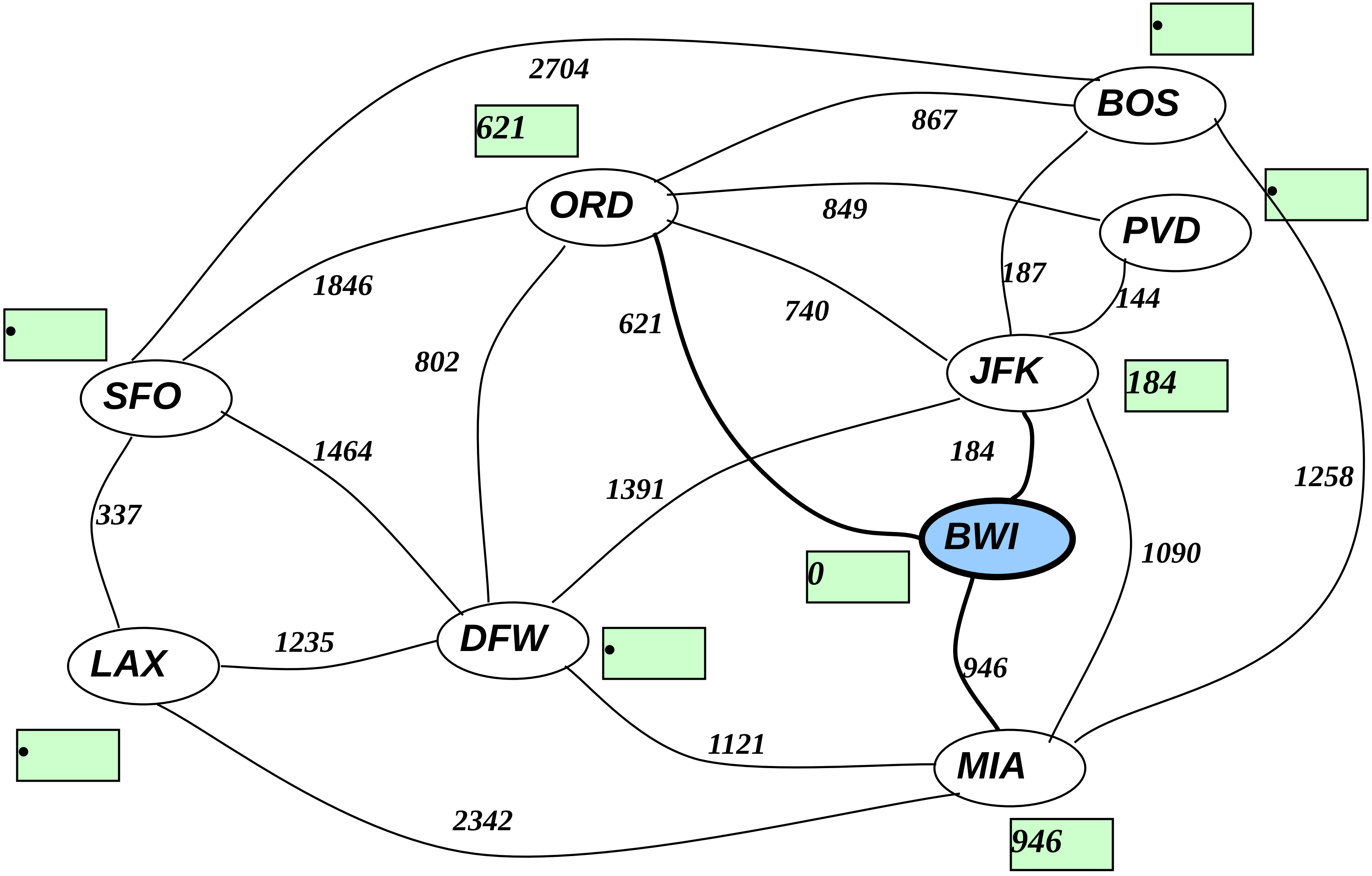
```
Algorithm Dijkstra ( $G$ , $s \in V(G)$ )
for each vertex $u \in V(G)$ do $dist[u] \leftarrow +\infty$
$dist[s] \leftarrow 0$ ;   $\pi[s] \leftarrow nil$                 § first node to be scanned
$Q \leftarrow ConstructMinHeap(V(G), dist)$                § dist[u] = priority(u)
while $Q \neq \emptyset$ do                                § |V| iterations
    $u \leftarrow DeleteMin(Q)$                             § O(log V) time
    for each $v \in Adj[u]$ do
            if  $dist[v] > dist[u] + w(u,v)$  then do
                $dist[v] \leftarrow dist[u] + w(u,v)$
                $\pi[v] \leftarrow u$                          § O( |Adj[u]| log V) time
                $UpHeap(v,Q)$
           end-if
    end-for
end-while
end
```

Total time = $O(E \log V)$ with standard binary heap

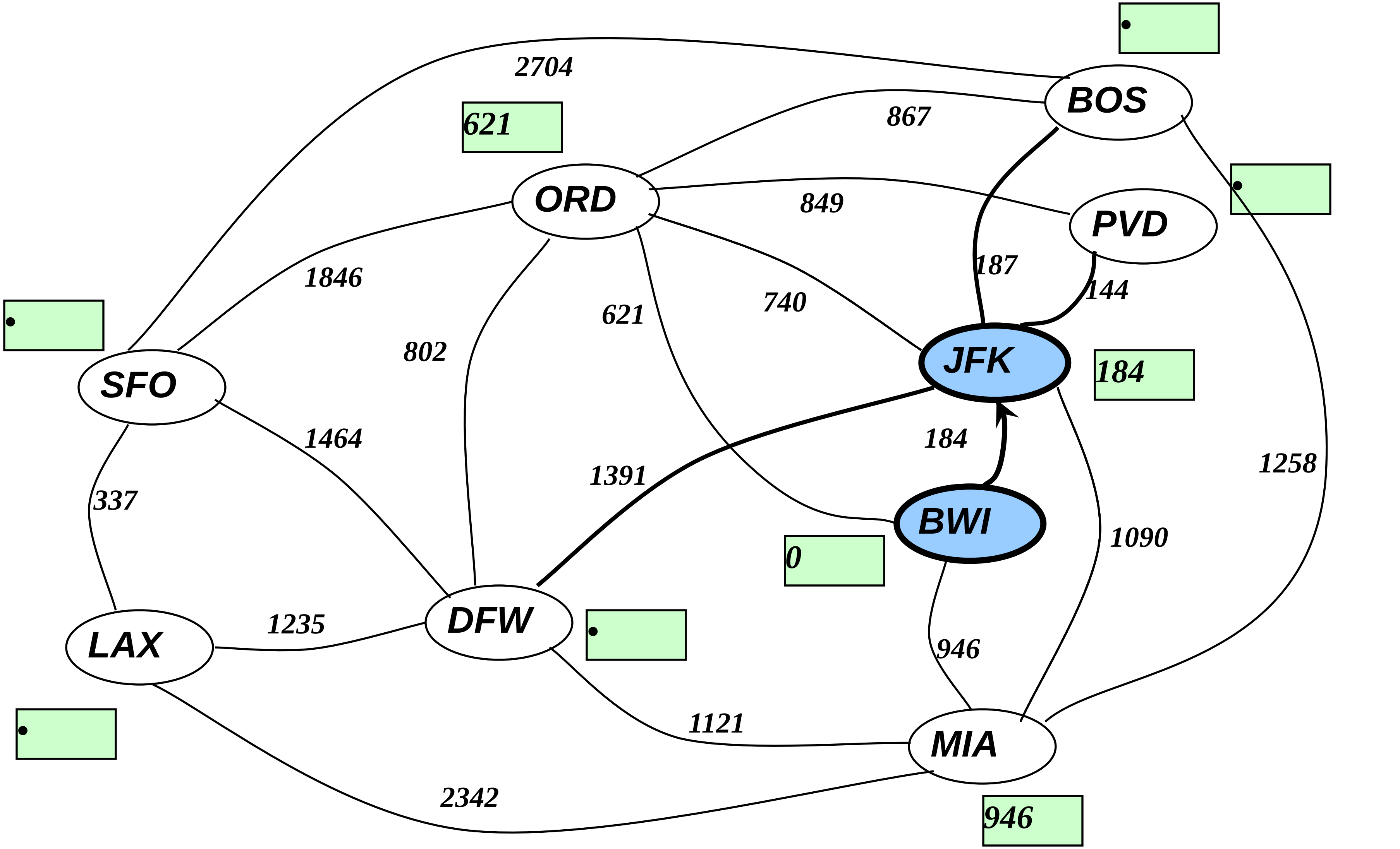
Dijkstra’s algorithm



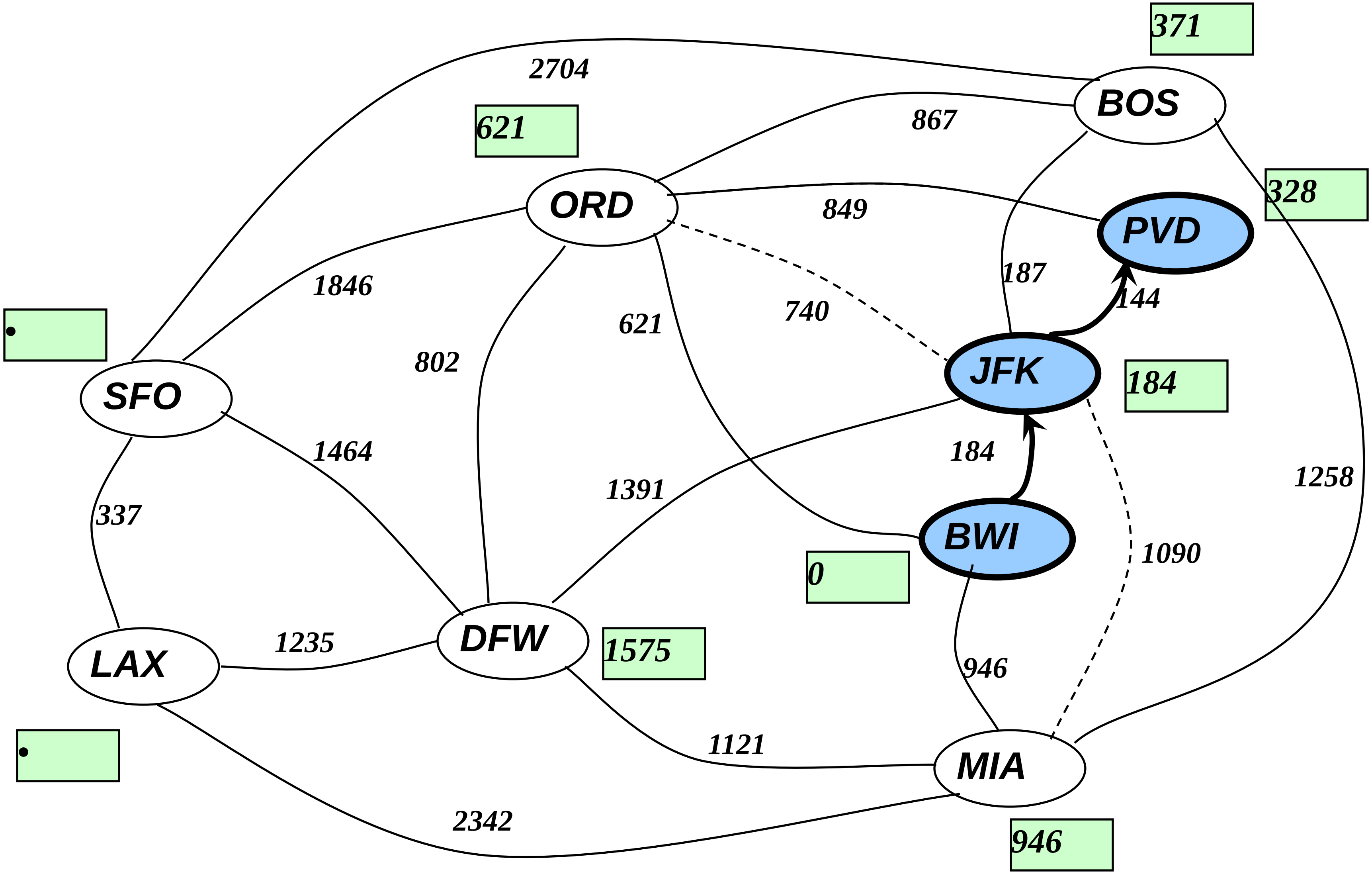
Dijkstra’s algorithm



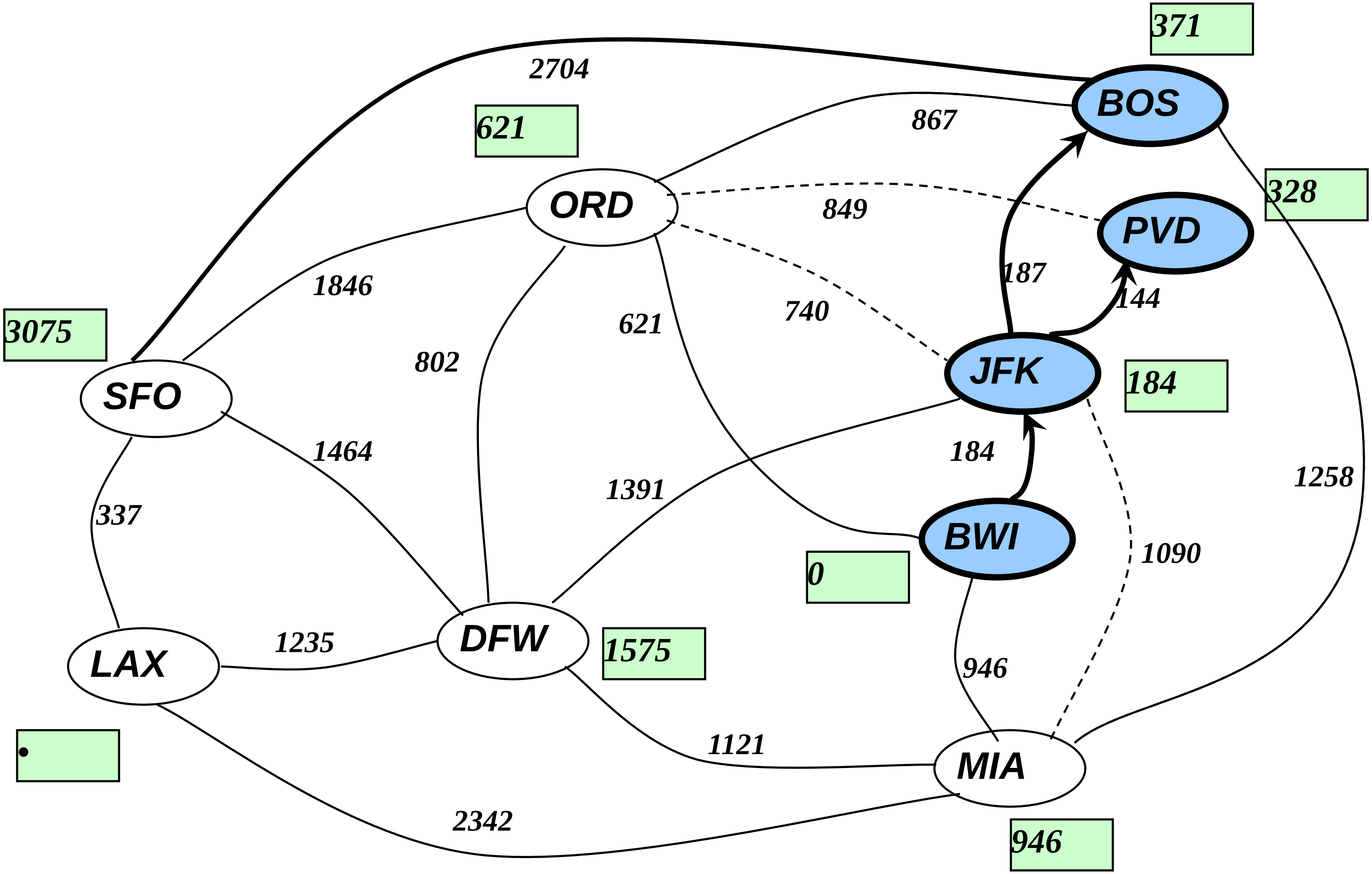
Dijkstra’s algorithm



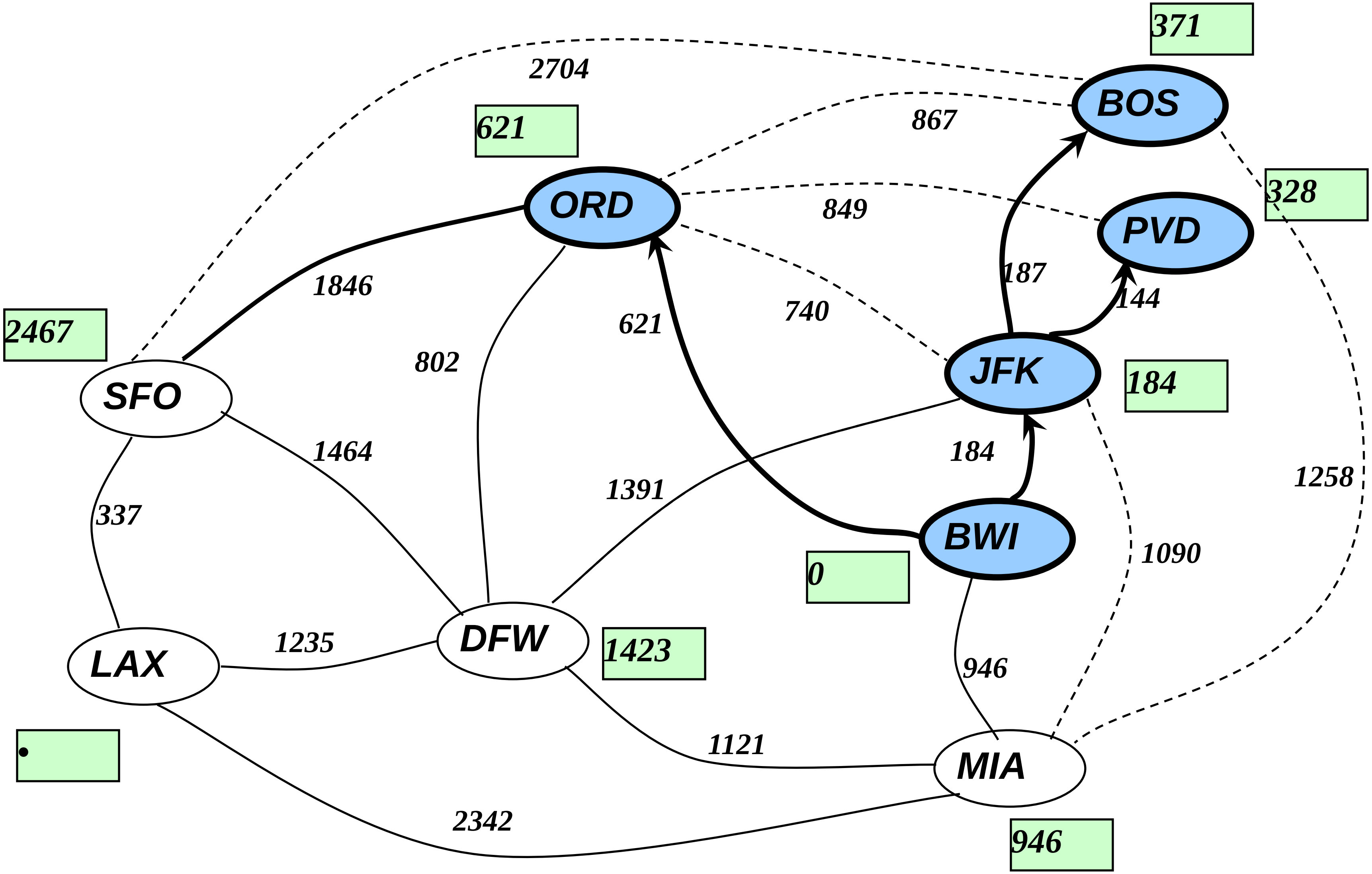
Dijkstra’s algorithm



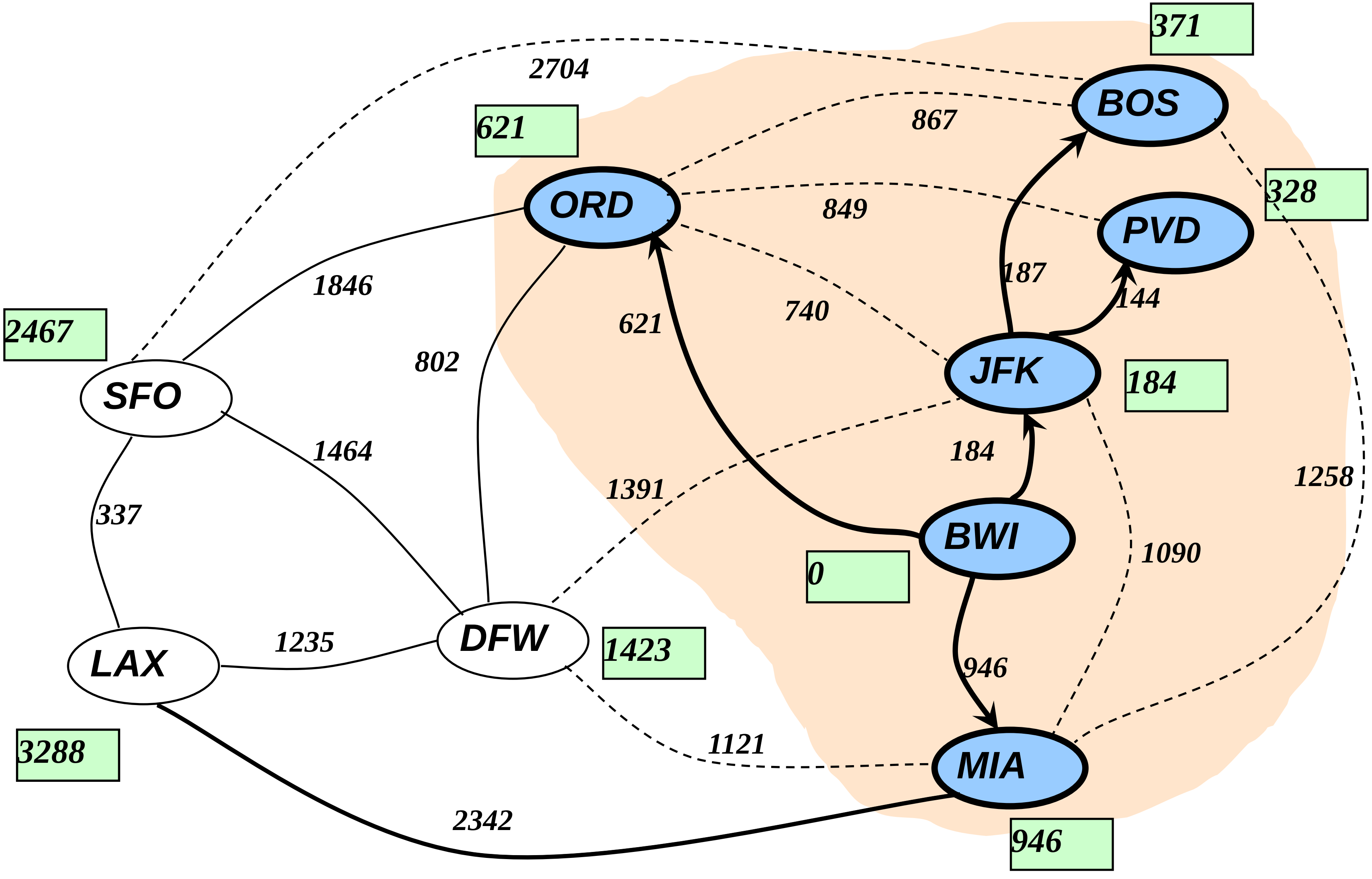
Dijkstra’s algorithm



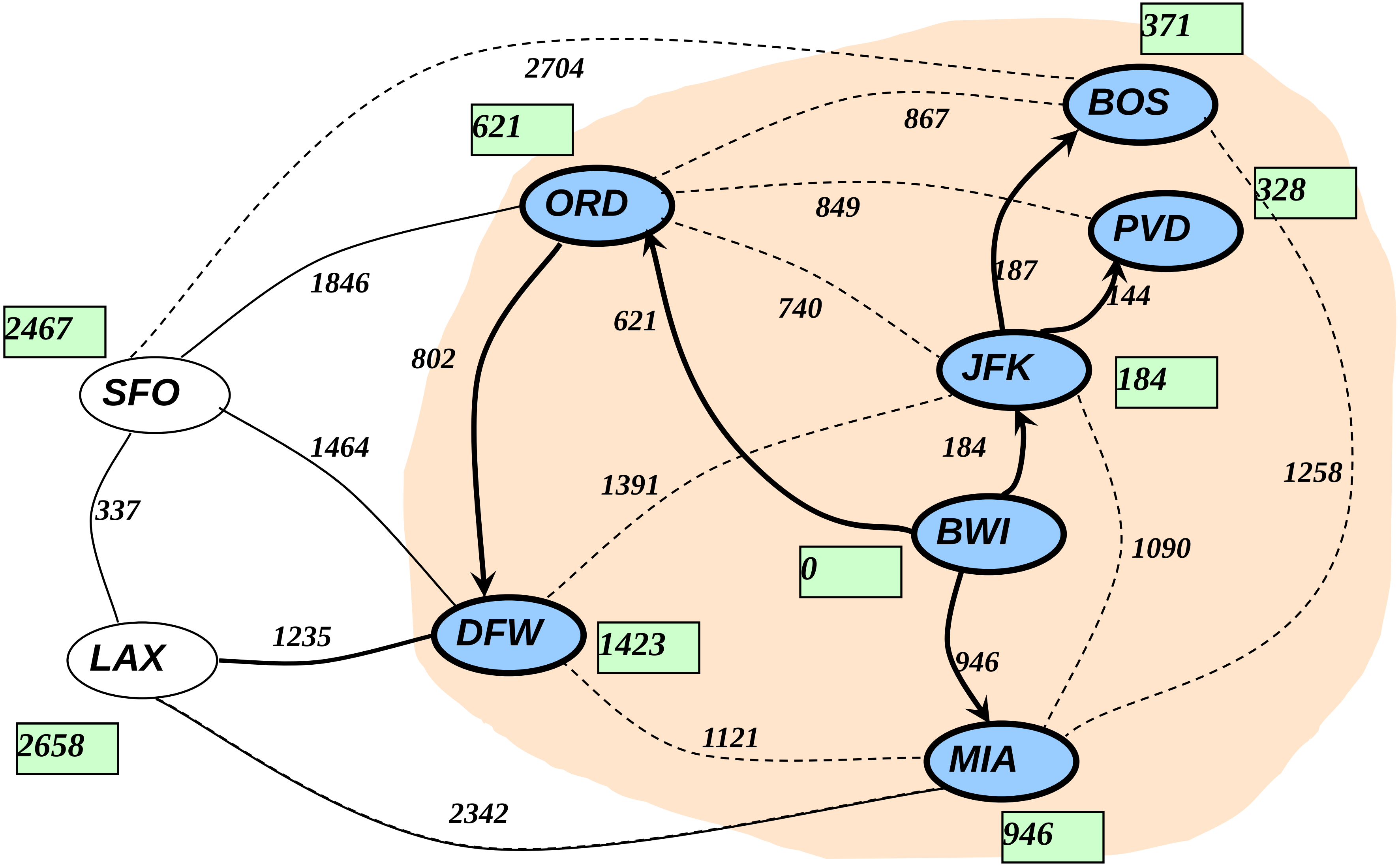
Dijkstra’s algorithm



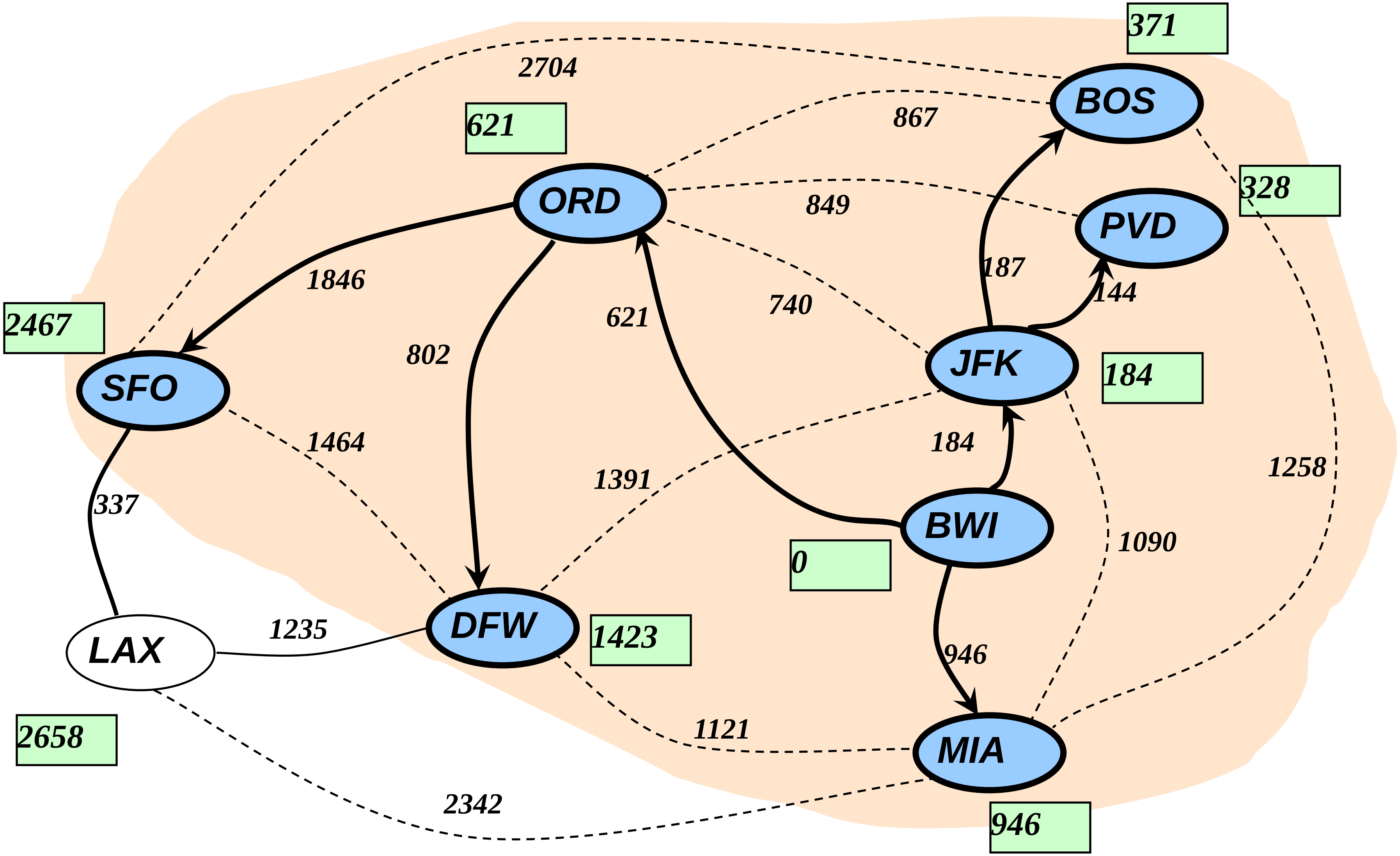
Dijkstra’s algorithm (cont)



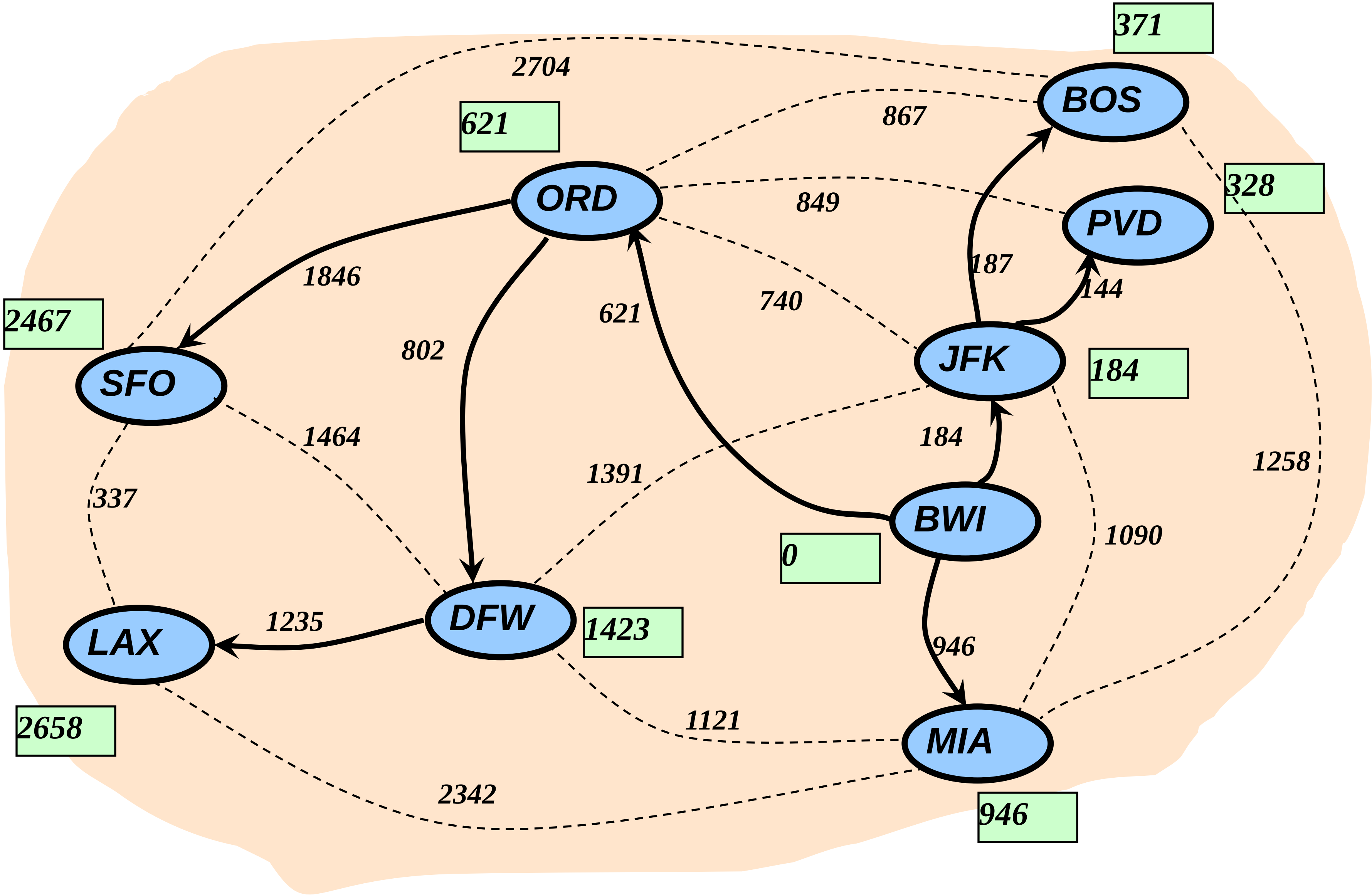
Dijkstra’s algorithm (cont)



Dijkstra’s algorithm (cont)



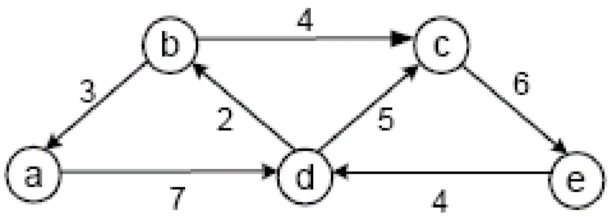
Dijkstra’ s algorithm (cont)



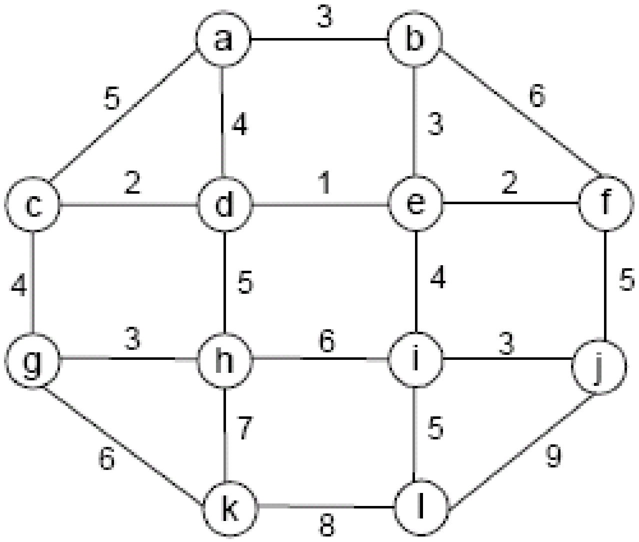
In-Class Exercises

2. Solve the following instances of the single-source shortest-paths problem with vertex a as the source:

a.



b.



The End