Lesson 4: Graphs

CSC325 – ADVANCED DATA STRUCTURES & ALGORITHMS | SPRING 2022

DR. ANDREY TIMOFEYEV

OUTLINE

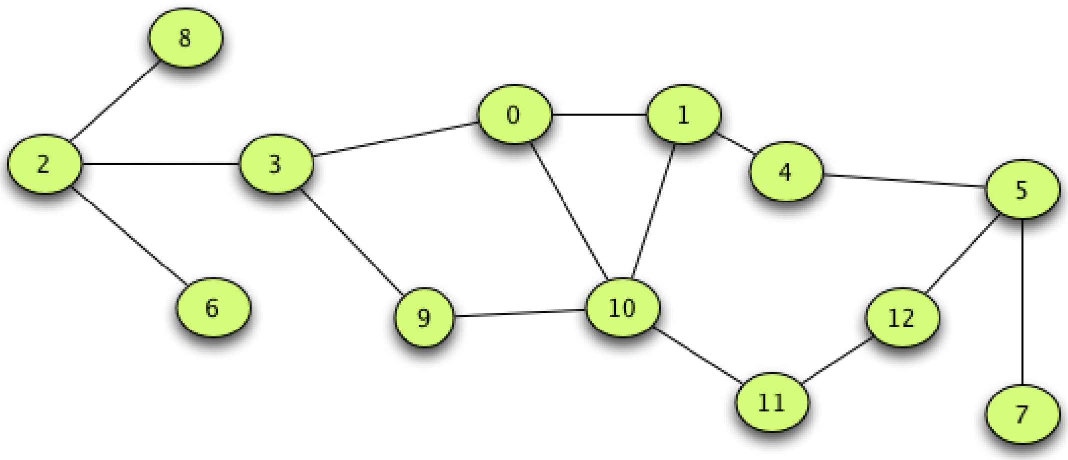
- Introduction.
- Graphs types.
- Graph representation.
- Minimum spanning trees.
- Kruskal’s algorithm.
- Shortest paths.
- Dijkstra's algorithm.

INTRODUCTION

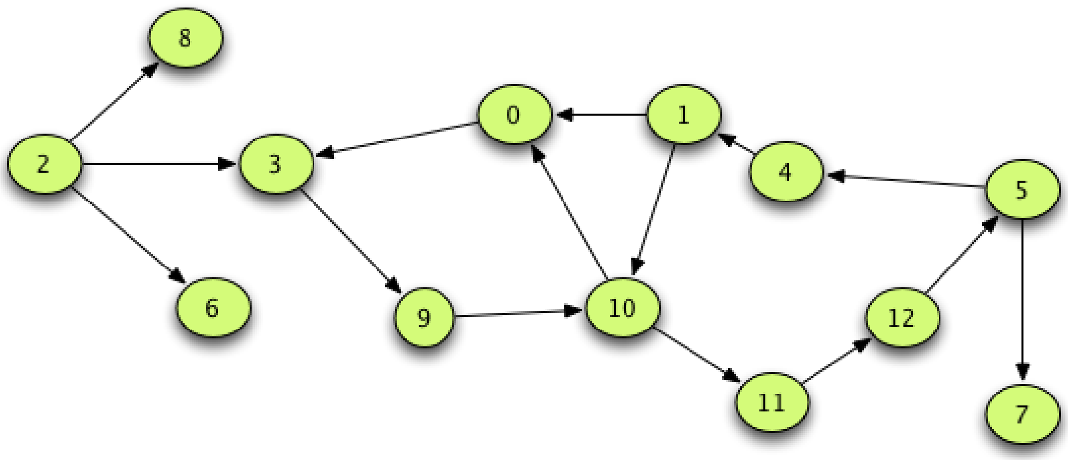
- **Graph** – complex non-linear data structure defined by set of **vertices** connected by **edges**.
  - *G = (V, E)*, where V - set of vertices/nodes, E - set of edges/links/arcs between the vertices/nodes.
- **Vertex/node.**
  - Represents an entity in a graph.
  - Has a name/key and can carry additional information – payload.
- **Edge/link/arc.**
  - Connects two vertices and denotes relationship between them.
    - Represented by a pair of vertices it connects.
  - One-way (directed) or two-way (undirected).
  - Can have a value associated with it (weight).
    - Cost to go from one vertex to another.

GRAPH TYPES AND NOTATIONS (1)

- **Graph types:**
  - **Direction-based.**
    - Undirected graphs $E(v_i, v_j) = E(v_j, v_i)$
      - All edges are bi-directional and can be traversed both ways.
    - Directed graphs $E(v_i, v_j) \neq E(v_j, v_i)$
      - Edges are uni-directional and can only be traversed one way.



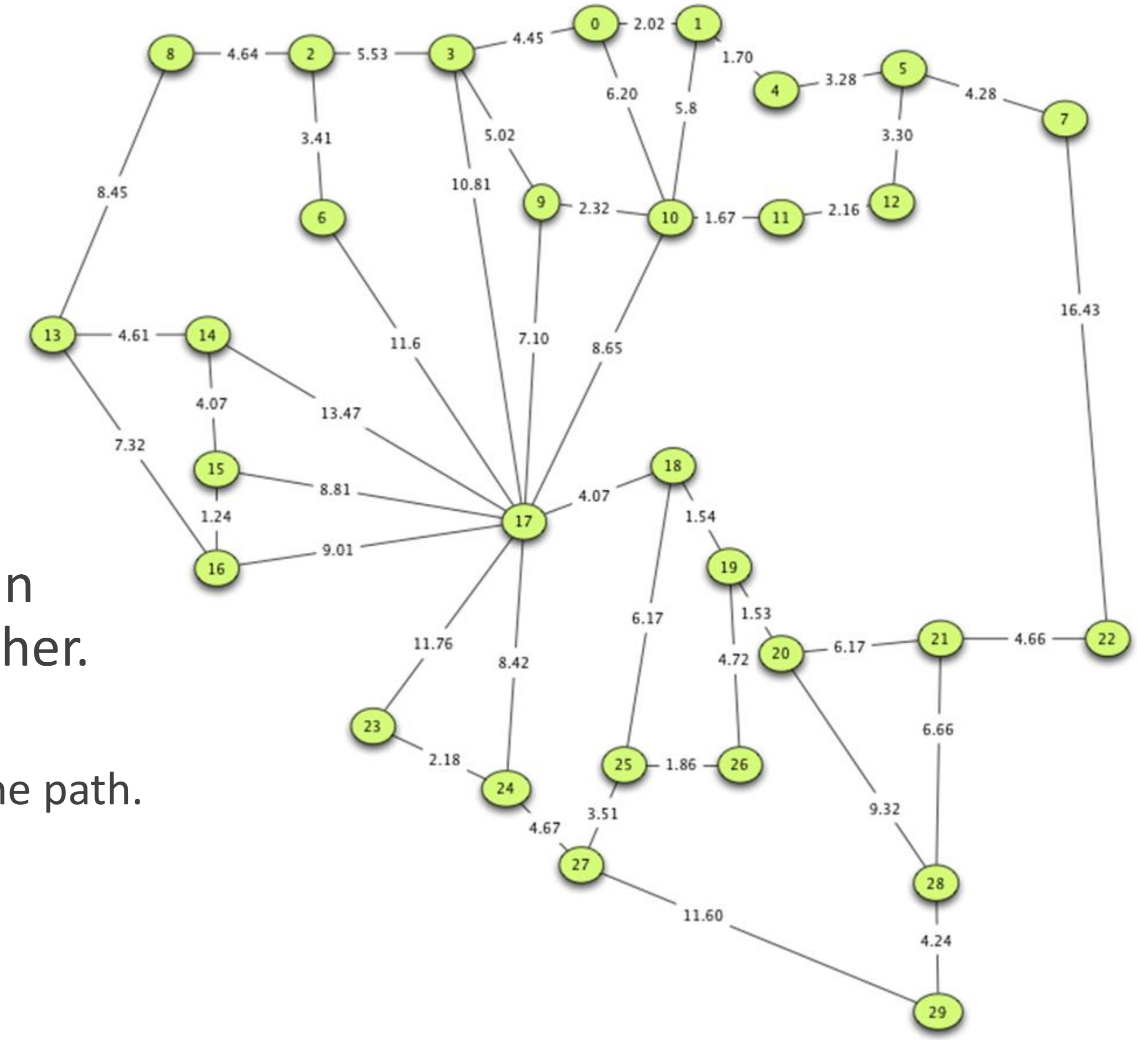
Undirected graph



Directed graph

GRAPH TYPES AND NOTATIONS (2)

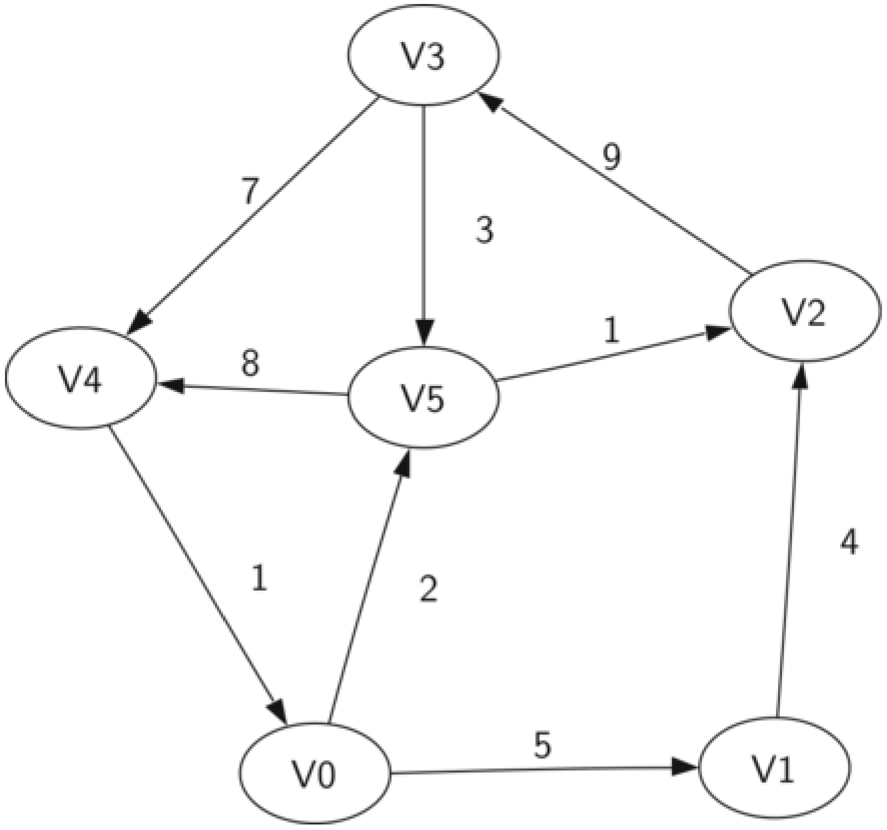
- **Graph types (cont.):**
  - **Weight-based.**
    - <u>Weighted graphs</u> *$E(v_i, v_j, w)$.*
      - Every edge have a weight *w* associated with it.
      - Weight function maps edges to real numbers.
    - <u>Unweighted graphs.</u>
      - No value associated with an edge.
  - **Path** is a series of graph edges (none repeated) that can be traversed in order to travel from one vertex to another.
    - Path length in unweighted graph = number of edges in the path.
    - Path length in weighted graph = sum of weights of all edges in the path.



Weighted undirected graph

GRAPH TYPES AND NOTATIONS (2)

- **Graph types (cont.):**
  - **Weight-based.**
    - Weighted graphs *E(v_i, v_j, w).*
      - Every edge have a weight *w* associated with it.
      - Weight function maps edges to real numbers.
    - Unweighted graphs.
      - No value associated with an edge.
  - **Path** is a series of graph edges (none repeated) that can be traversed in order to travel from one vertex to another.
    - Path length in unweighted graph = number of edges in the path.
    - Path length in weighted graph = sum of weights of all edges in the path.



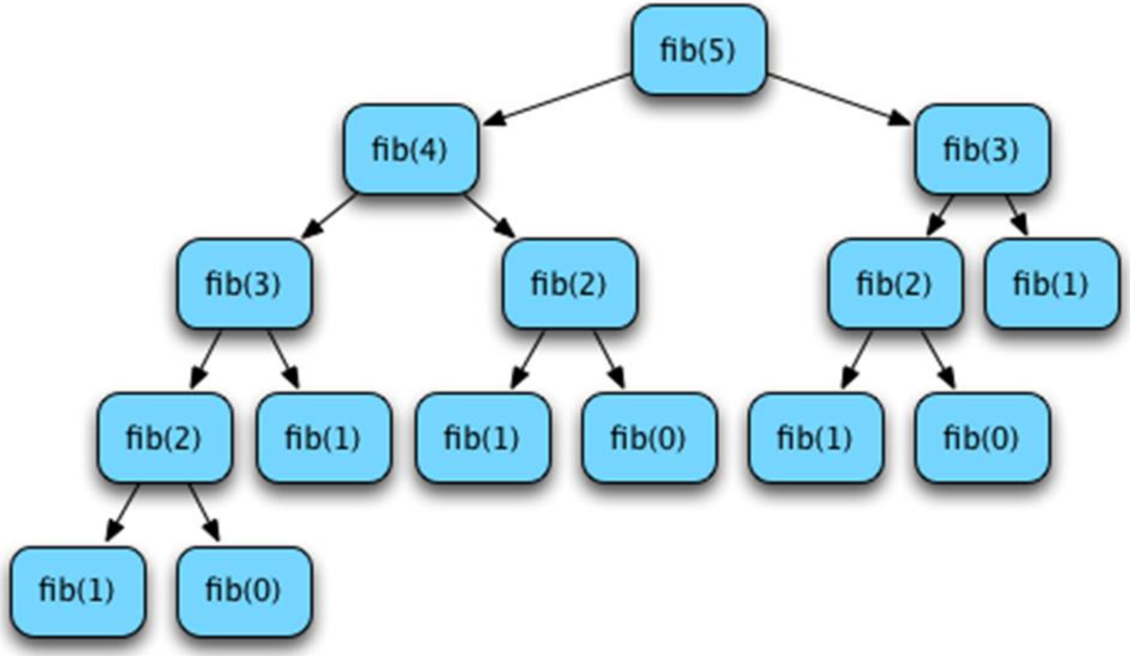
$$V = \{V0, V1, V2, V3, V4, V5\}$$

$$E = \left\{ \begin{array}{l} (v0, v1, 5), (v1, v2, 4), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1), \\ (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1) \end{array} \right\}$$

Weighted directed graph

GRAPH TYPES AND NOTATIONS (3)

- **Graph types (cont.):**
  - **Cycle-based.**
    - Cyclic graphs.
      - Contains cycles.
    - Acyclic graphs.
      - Does not have any cycles.
  - **Cycle** is a path that starts and ends at the same vertex.

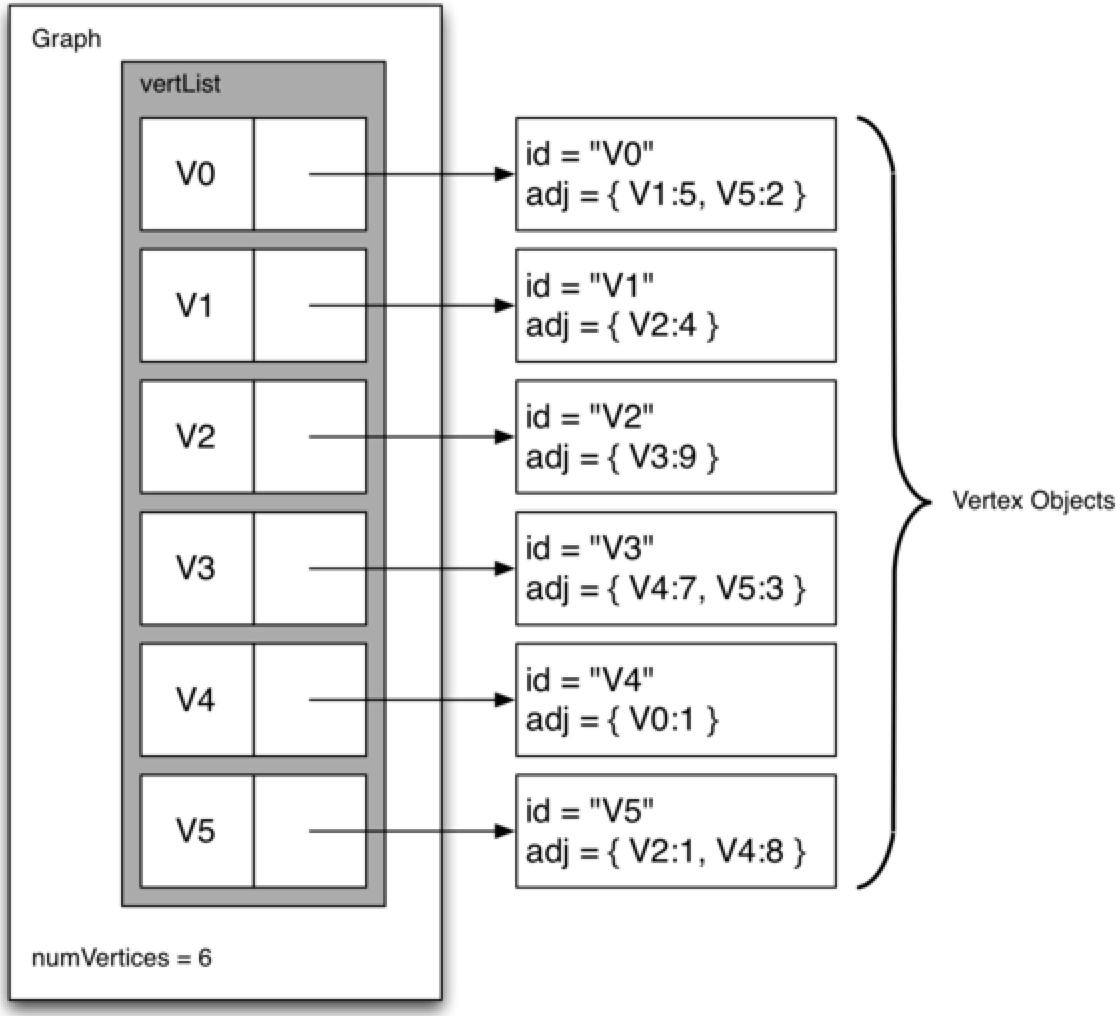


Directed acyclic graph (DAG)

GRAPH REPRESENTATION (1)

- **Graph can be represented as:**
  - **Adjacency list.**
  - Adjacency matrix.
  - Incidence matrix.

- **Adjacency list.**
  - Graph = **master list** of all vertices + **list** of all **adjacent vertices** in each vertex.
  - Pros:
    - Space-efficient for sparse graphs.
    - Iterating over the edges is efficient.
  - Cons:
    - Not efficient edge weight lookup.



Adjacency list

GRAPH REPRESENTATION (2)

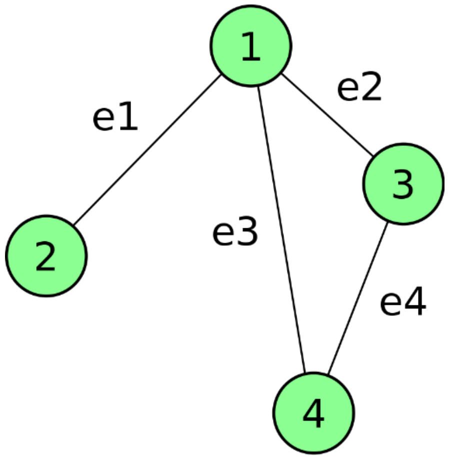
- **Graph can be represented as:**
  - Adjacency list.
  - **Adjacency matrix.**
  - Incidence matrix.
- **Adjacency matrix.**
  - Graph = **two-dimensional matrix**, where each **rows** & **columns** are **vertices** and **cells** are **edge**.
  - **Cell value** = **weight** or **connection** (unweighted graph).
  - Pros:
    - Space-efficient for dense graph representation.
    - The time complexity of getting an edge weight is O(1).
  - Cons:
    - Requires more space.
    - Iterating through the edges has high complexity.

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

Adjacency matrix

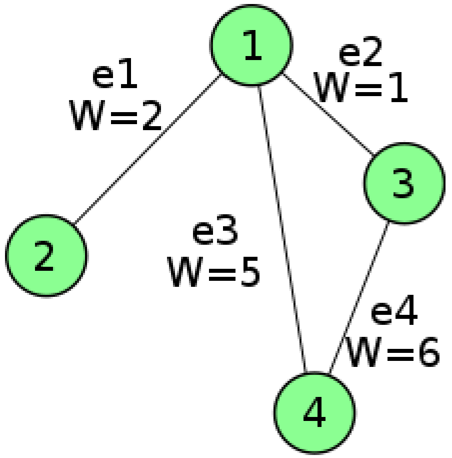
GRAPH REPRESENTATION (3)

- Graph can be represented as:
  - Adjacency list.
  - Adjacency matrix.
  - **Incidence matrix.**
- **Incidence matrix.**
  - Graph = **two-dimensional matrix**, where **row** is a **vertex**, **column** is an **edge** and **cell** is an **incidence relation** between two.
  - Not frequently used in practice.



	e_1	e_2	e_3	e_4
1	1	1	1	0
2	1	0	0	0
3	0	1	0	1
4	0	0	1	1

$$= \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$



	e_1	e_2	e_3	e_4
1	2	1	5	0
2	2	0	0	0
3	0	1	0	6
4	0	0	5	6

$$= \begin{bmatrix} 2 & 1 & 5 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 5 & 6 \end{bmatrix}$$

GRAPH REPRESENTATION (4)

- Graph **operations complexities** based on **representation**:

	Adjacency list	Adjacency matrix	Incidence matrix
Store graph	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \times \|E\|)$
Add vertex	$O(1)$	$O(\|V\|^2)$	$O(\|V\| \times \|E\|)$
Add edge	$O(1)$	$O(1)$	$O(\|V\| \times \|E\|)$
Remove vertex	$O(\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \times \|E\|)$
Remove edge	$O(\|V\|)$	$O(1)$	$O(\|V\| \times \|E\|)$
Check vertex adjacency	$O(\|V\|)$	$O(1)$	$O(\|E\|)$
Remarks	Slow to remove vertices & edges - needs to find all vertices or edges.	Slow to add or remove vertices - matrix must be resized/copied.	Slow to add or remove vertices & edges - matrix must be resized/copied.

Graph operations complexities

GRAPH TRAVERSAL (1)

- **Graph traversal (search)** – process of visiting **each** node in the graph.
- **Two types of graph traversal algorithms:**
  - **Depth first search.**
    - Checks "**children**" vertices **before** "**sibling**" vertices.
    - **<u>Process:</u>**
      - Start at "current" vertex, mark as "visited".
      - Consider arbitrary edge of "current" vertex.
        - If connected to already "visited" vertex, ignore edge.
        - If connected to "unvisited" vertex, go to vertex, consider "current", & mark "visited".
        - Repeat.
      - If all edges in "current" lead to "visited" – backtrack.
        - Go back to the previous "current" and check another "unvisited" vertex.
      - Terminate when backtracked to starting vertex and all edges lead to "visited" vertices.
  - **Breadth first search.**
    - Checks "**sibling**" vertices **before** "**child**" vertices.
    - **<u>Process:</u>**
      - Start at vertex at level 0.
      - Mark "visited" all vertices adjacent to start vertex.
        - One edge away from start vertex – level 1.
      - Go two levels away from starting vertex.
      - Place vertices adjacent to level 1 and not "visited" to level 2 & mark "visited".
      - Repeat until no new vertices found on a level.

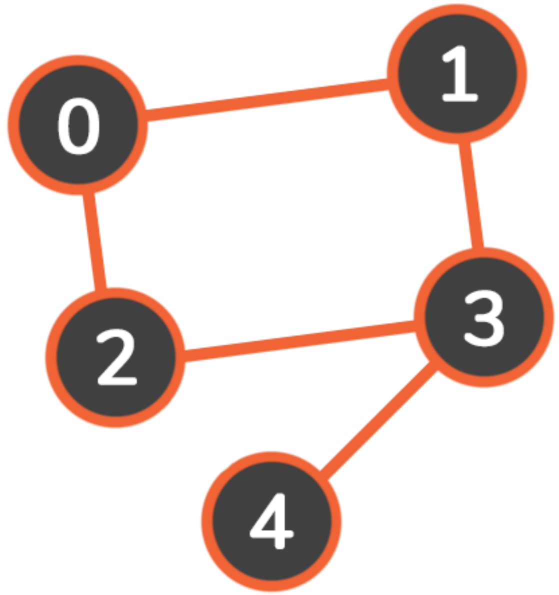
GRAPH TRAVERSAL (2)

- **Depth (breadth) first search traversal pseudocode:**

```
DFS-iterative(G, s)
      S = stack
      S.push(s)
      mark s as visited
      while (S is not empty):
            v = S.pop()
            for all neighbors w of v in Graph G:
                  if w is not visited:
                        S.push(w)
                        mark w as visited
```

```
DFS-recursive(G, s)
      mark s as visited
      for all neighbors v of s in Graph G:
            if v is not visited
                  DFS-recursive(G, v)
```

- **Depth first search traversal peculiarities:**
  - Easy to implement **recursively**.
  - **Cycles** are **avoided** by marking "visited" vertices.
  - *O(|V| + |E|)* time complexity.



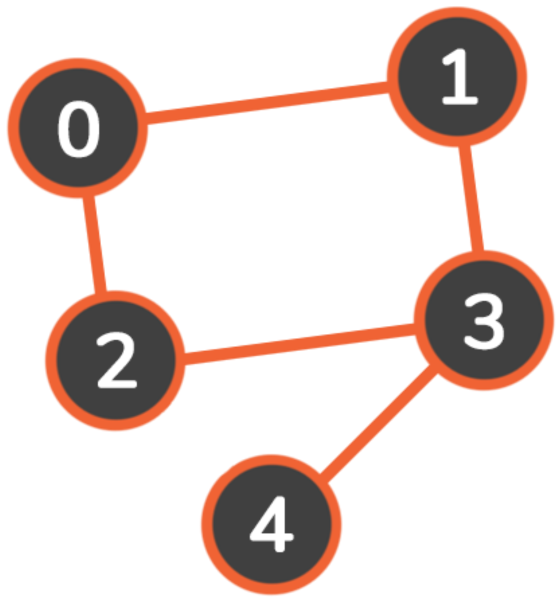
DFS example

GRAPH TRAVERSAL (2)

- **Depth first search traversal pseudocode (fixed):**

```
DFS-iterative(G, s)
      S = stack
      S.push(s)
      while (S is not empty):
            v = S.pop()
            for all neighbors w of v in Graph G:
                  if w is not visited:
                        S.push(w)
            mark v as visited
```

```
DFS-recursive(G, s)
      mark s as visited
      for all neighbors v of s in Graph G:
            if v is not visited
                  DFS-recursive(G, v)
```



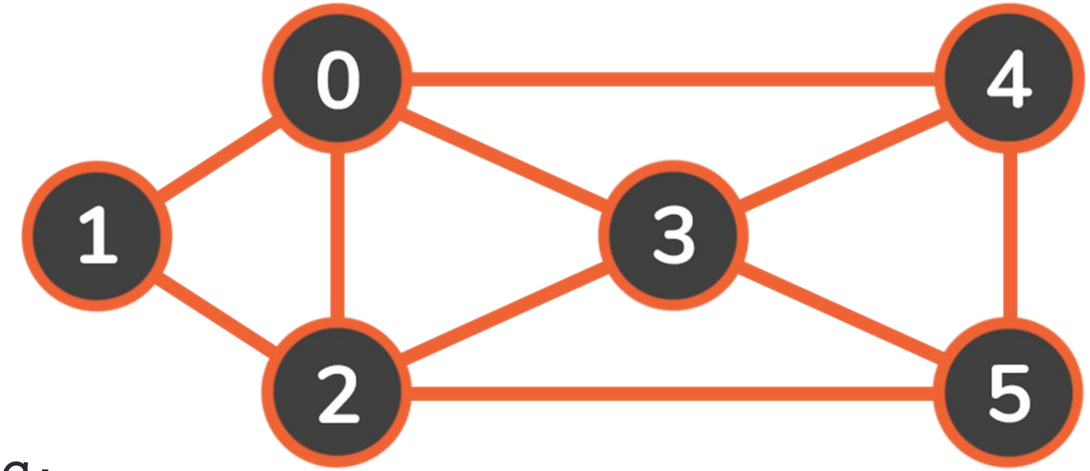
DFS example

- **Depth first search traversal peculiarities:**
  - Easy to implement **recursively**.
  - **Cycles** are **avoided** by marking "visited" vertices.
  - $O(|V| + |E|)$ time complexity.

GRAPH TRAVERSAL (3)

- **Breadth first search traversal pseudocode:**

```
BFS(G, s)
    Q = queue
    Q.enq(s)
    mark s as visited
    while (Q is not empty):
        v = Q.deq()
        for all neighbors w of v in Graph G:
            if w is not visited:
                Q.enq(w)
                mark w as visited
```



BFS example

- **Breadth first search traversal peculiarities:**
  - **Cycles** are **avoided** by marking “visited” vertices.
  - Traverses graph in “**rounds**” and by “**layers**”.
  - Forms a **BFS tree** while traversing.
  - $O(|V| + |E|)$ time complexity.

MINIMUM SPANNING TREES

- **Minimum spanning tree (MST)** problem definition:
  - In **undirected weighted** graph *G*, find a tree *T* that contains **all vertices** in G and **minimizes** sum:
    - $\boldsymbol{w(T) = \sum_{(u,v) in\ T} w(u,v)}$
      - *w(T)* – total weight of tree T.
      - *(u, v)* – edge between vertices u and v.
      - *w(u, v)* – weight of an edge between vertices u and v.
  - Tree T = **spanning tree**.
  - Finding T with **min total weight** = minimum spanning tree (**MST**) problem.
- MST problem is solved by **greedy methods**.
  - Choose **objects** to join a **growing collection** by iteratively **picking** an object that **minimizes** the cost of some **function**.

MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (1)

- **Kruskal's algorithm for constructing MST.**
  - **Grows** the MST in **clusters** by considering edges in **increasing** order of their **weights**.
  - **Maintains** a **forest** of **clusters**, repeatedly merging pairs of clusters until a **single cluster** spans the **graph**.

- **Algorithm process:**
  - Initially, treat **each vertex** as a **singleton cluster**.
  - Then consider **each edge** in turn, ordered by the **weight**.
    - If edge **connects** two **vertices** in **different clusters**:
      - **Add edge** to the list of edges of the **MST**.
      - **Merged two clusters** connected by edge into a **single cluster**.
    - If edge **connects** two **vertices** that are in the **same cluster**:
      - **Discard** edge.
  - **Terminate** & **output the MST** once **enough edges** added to the **cluster** to **span** the **whole graph**.

MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (2)

- **Kruskal's algorithm pseudocode.**

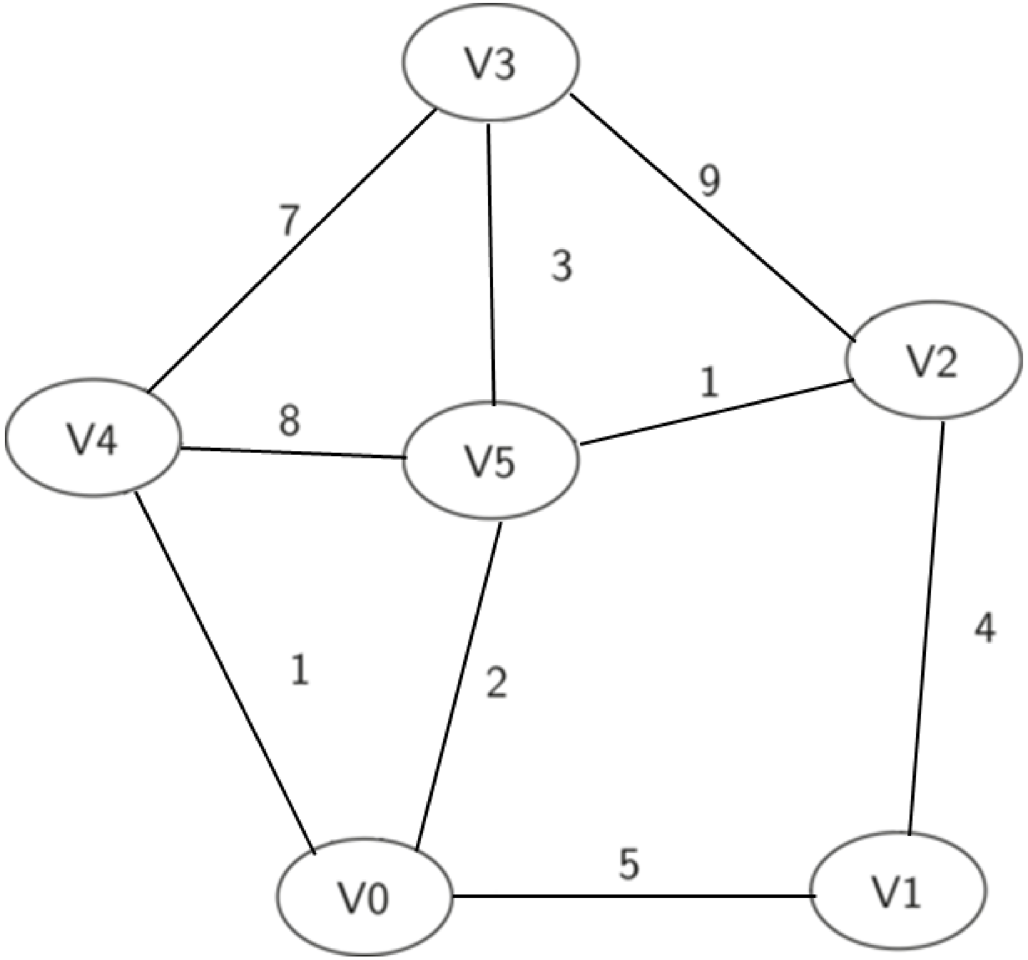
```
Kruskal(G):
        Define set S(v) = {v} for each vertex v in G

        Initialize dict D that contains all edges in G with edges as keys and weights as values
        Sort edges by their weights in ascending order

        Create empty set T that will contain edges and weights of the MST

        For each edge (u, v) in dict D
                Let S(u) be the set containing u, and S(v) be the set containing v
                If S(u) != S(v) then
                        Add edge (u, v) and weight w to T
                        Merge S(u) and S(v) into one set
                        Delete S(u) and S(v)

        Return tree T
```

MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (3)

- **Kruskal's algorithm complexity depends on:**
  - **Sorting edges by their weights.**
    - Complexity of O(nlogn), where n - # of edges.
  - **Choosing correct edges & forming unions of sets.**
    - Find clusters for vertices *u* and *v* (edge endpoints) -> *O(1)* by index lookup.
    - Check if clusters are distinct (edge connects two different clusters) -> *O(1)* by reference compare.
    - Merge two clusters into one -> $O(n^2)$.
      - Forming new set from other two.
      - Performed *n-1* times (for each vertex added to MST).
      - Complexity of $O(n^2)$, where n - # of vertices.
        - First time 1 vertex added, second time 2 vertices added, etc.
- **Overall algorithm complexity – $O(n^2)$.**
  - Can be improved by using **partition** or **union-find** data structures.
    - **O(E log V)**

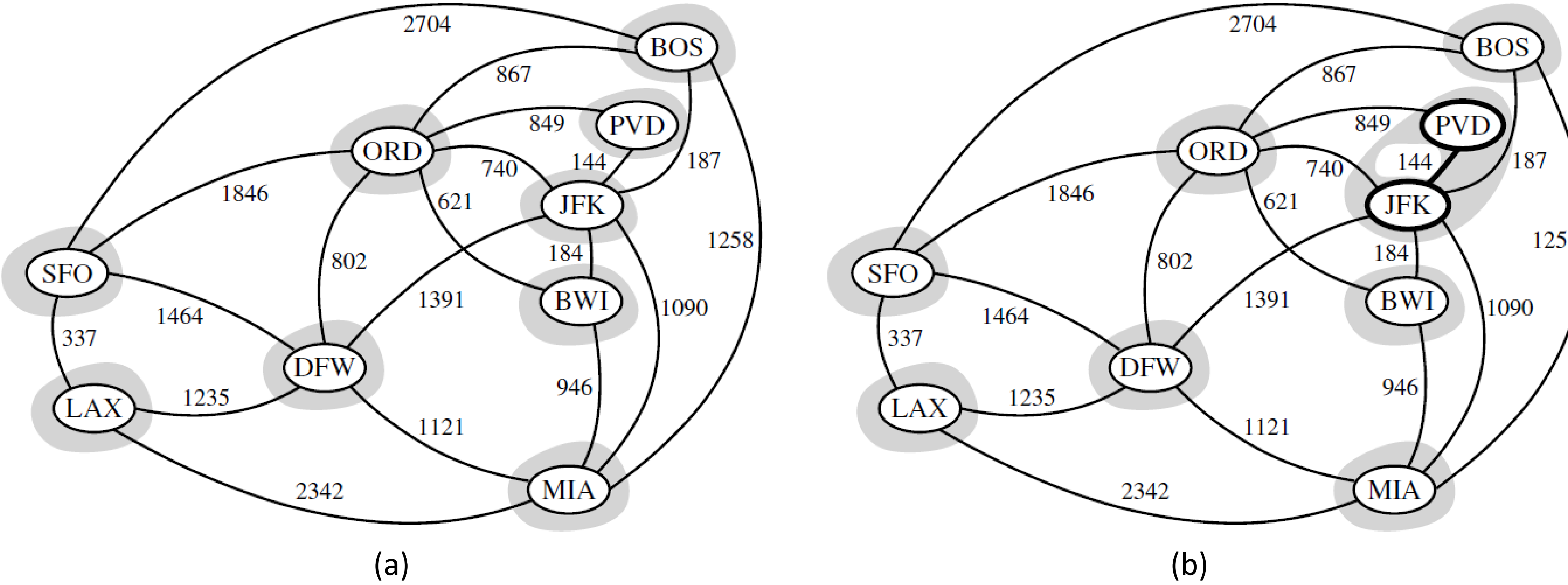
MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (4)

- **Kruskal's algorithm example (1).**



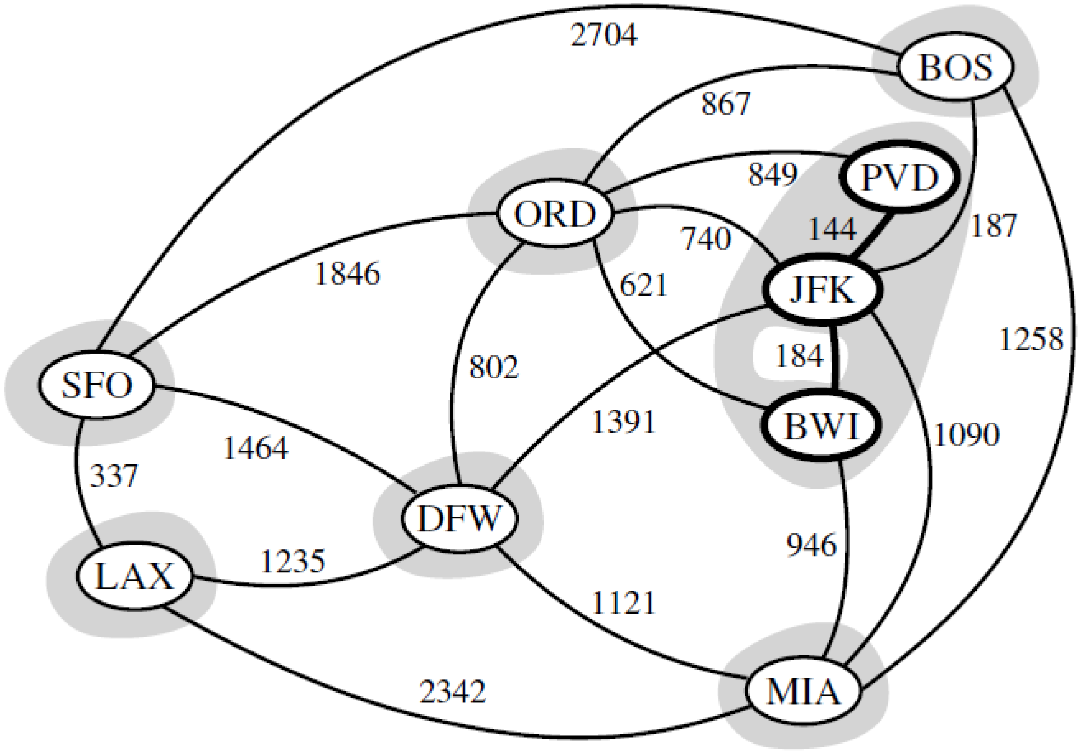
MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (5)

- **Kruskal's algorithm example (2).**

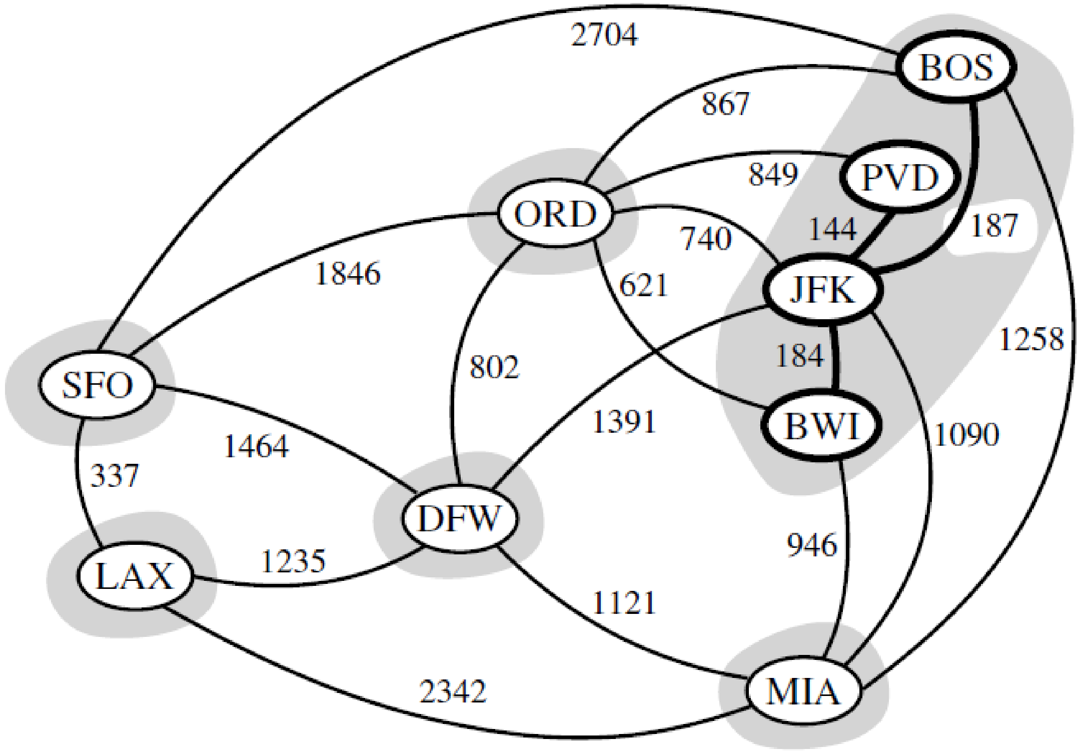


MINIMUM SPANNING TREES: KRUSKAL’S ALGORITHM (5)

- **Kruskal’s algorithm example (2).**



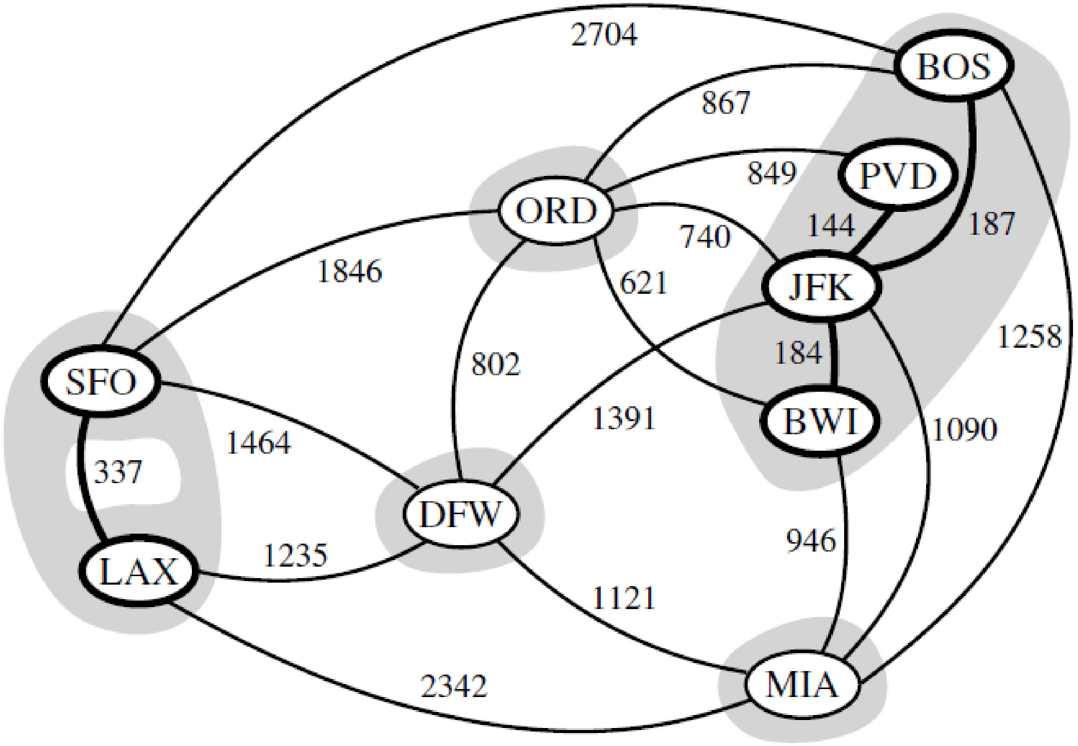
(c)



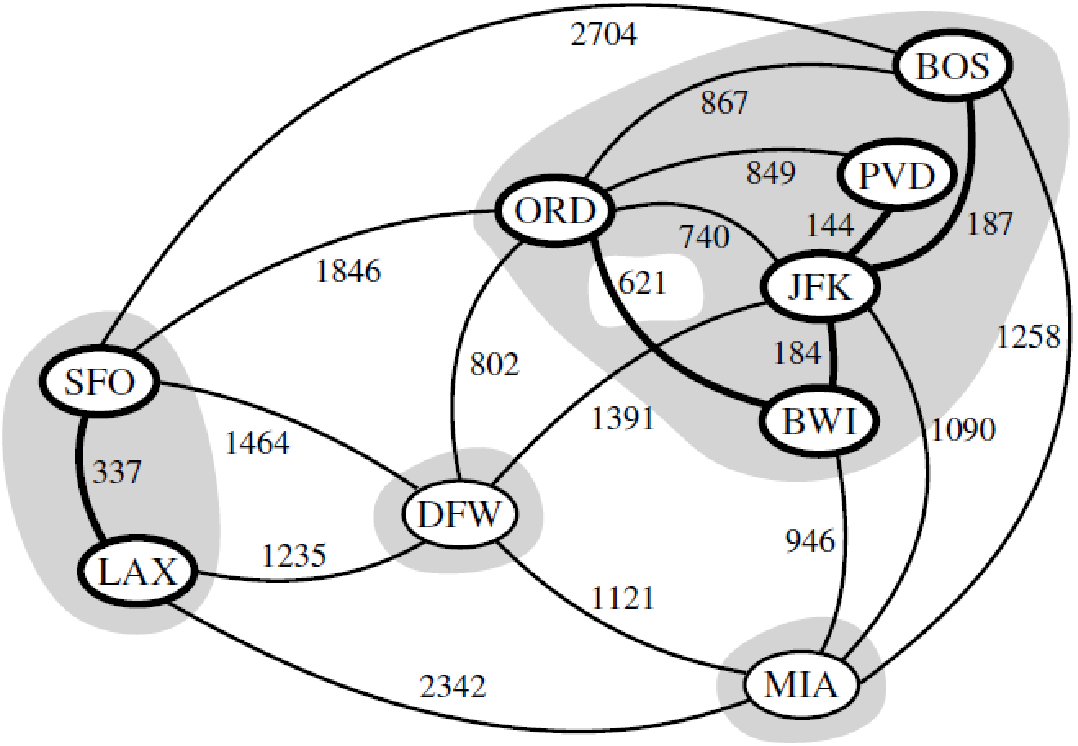
(d)

MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (5)

- **Kruskal's algorithm example (2).**



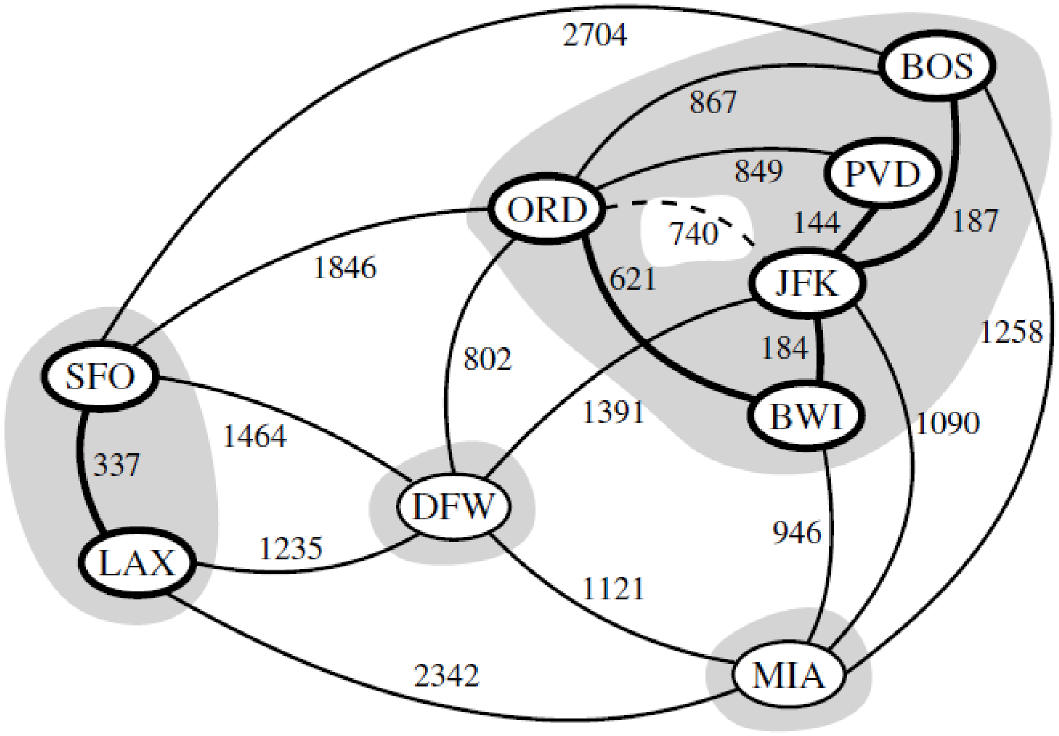
(e)



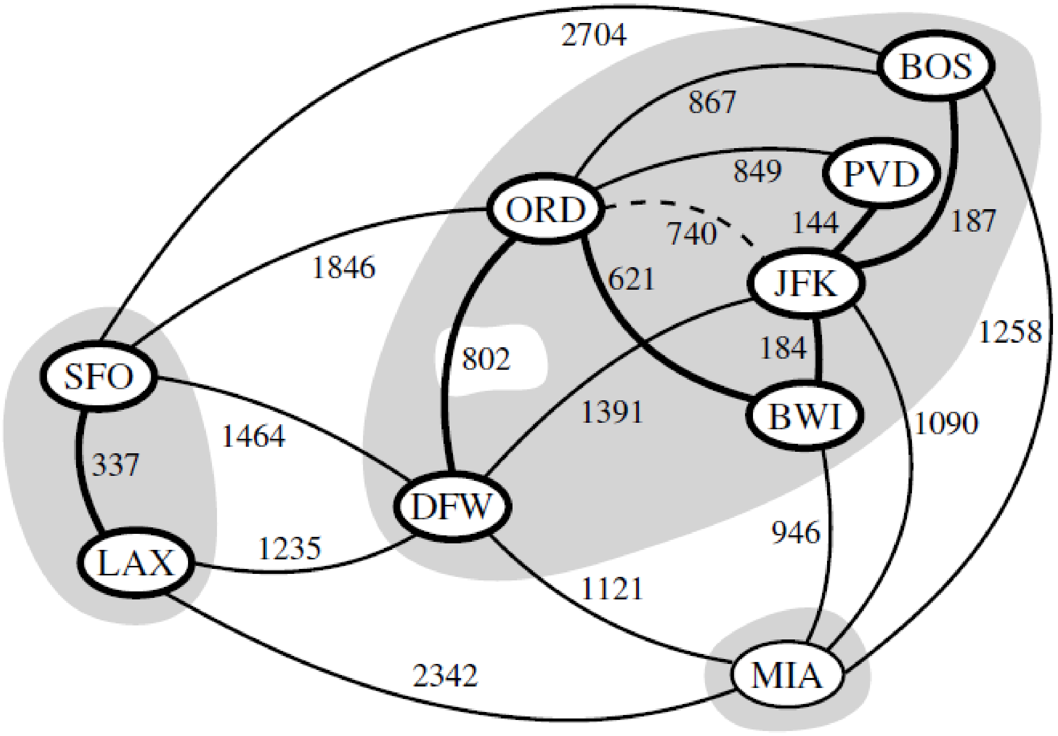
(f)

MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (5)

- **Kruskal's algorithm example (2).**



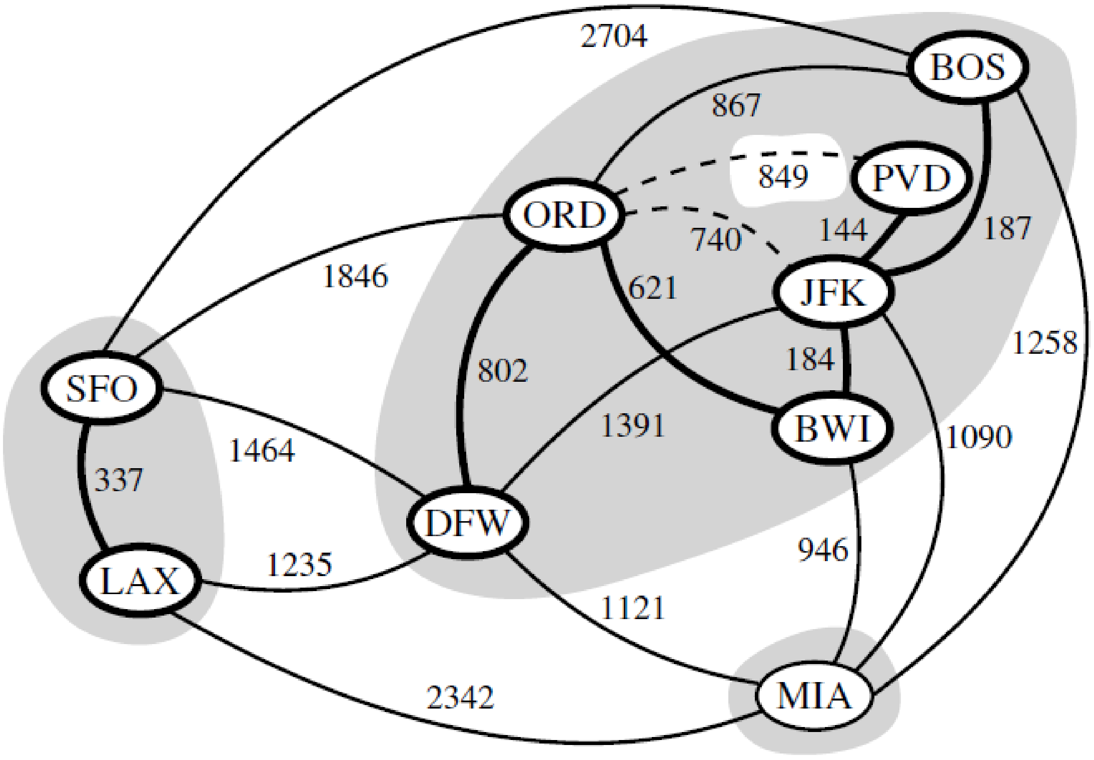
(g)



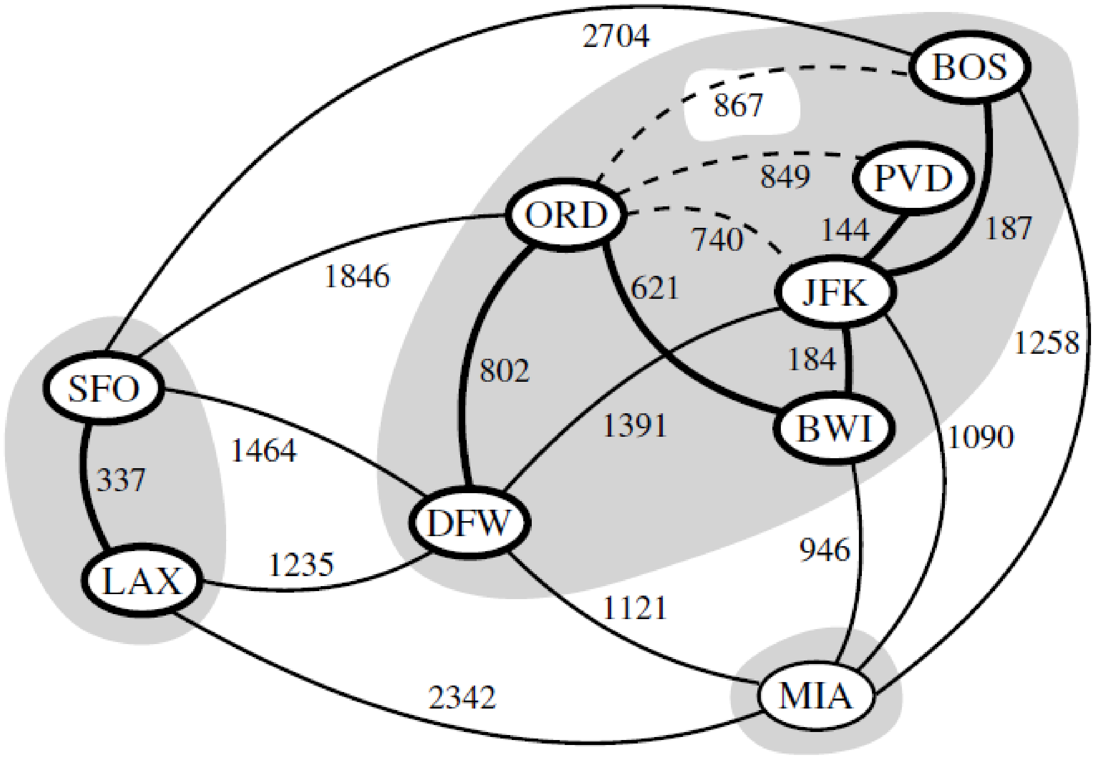
(h)

MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (5)

- **Kruskal's algorithm example (2).**



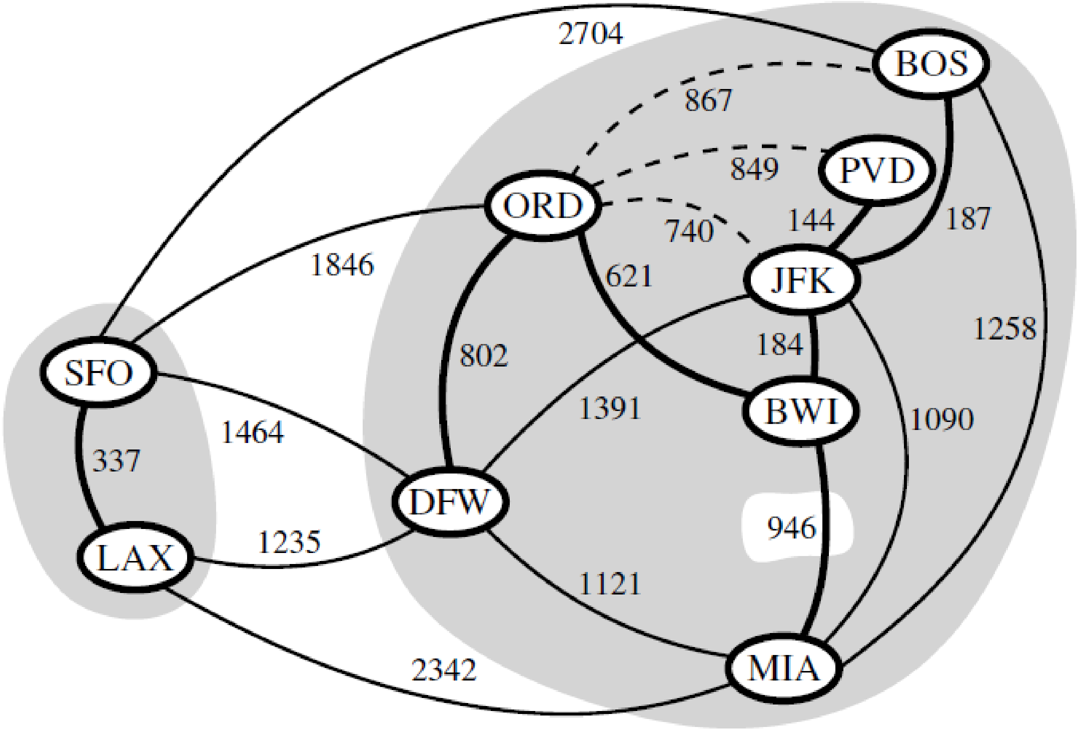
(i)



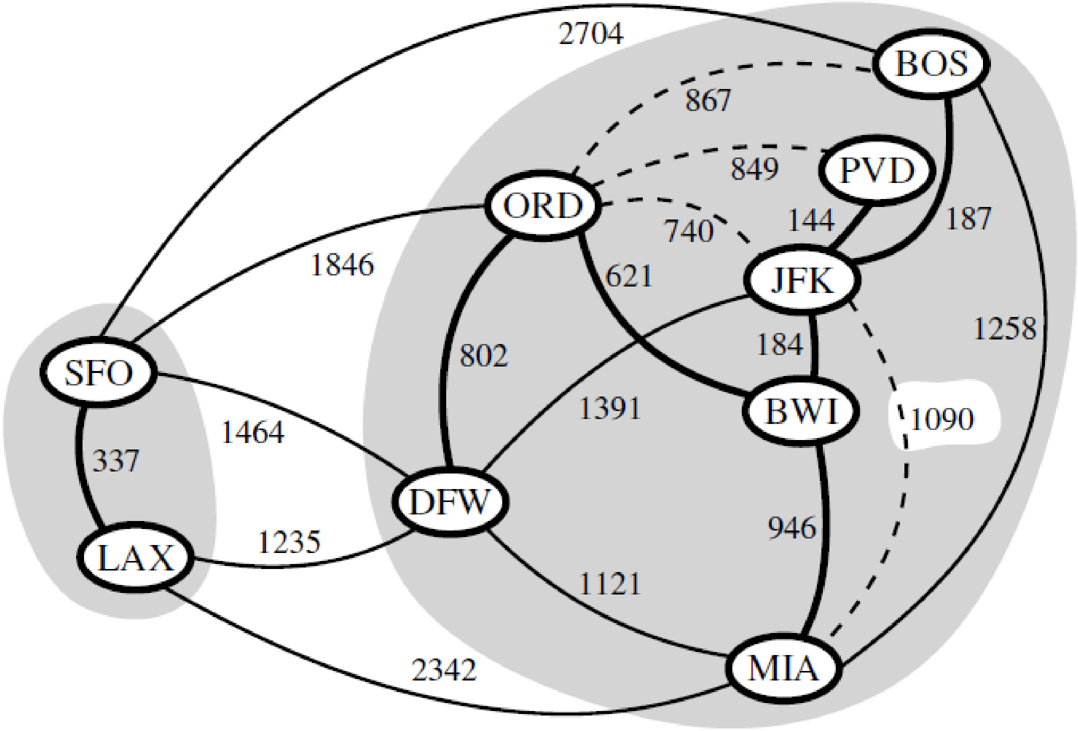
(j)

MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (5)

- **Kruskal's algorithm example (2).**



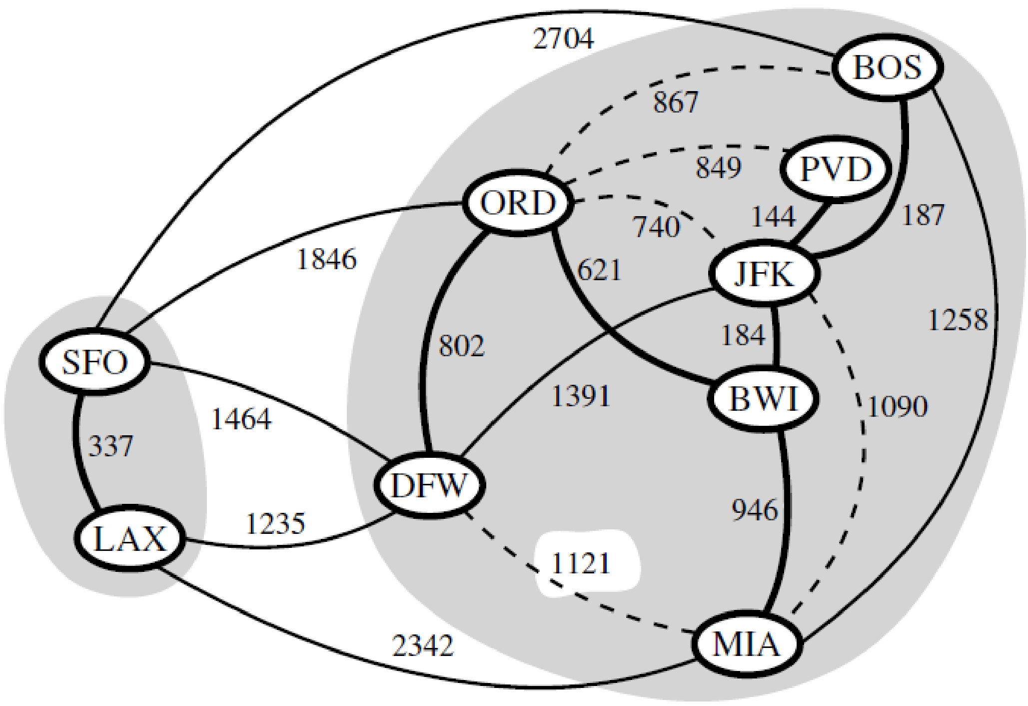
(k)



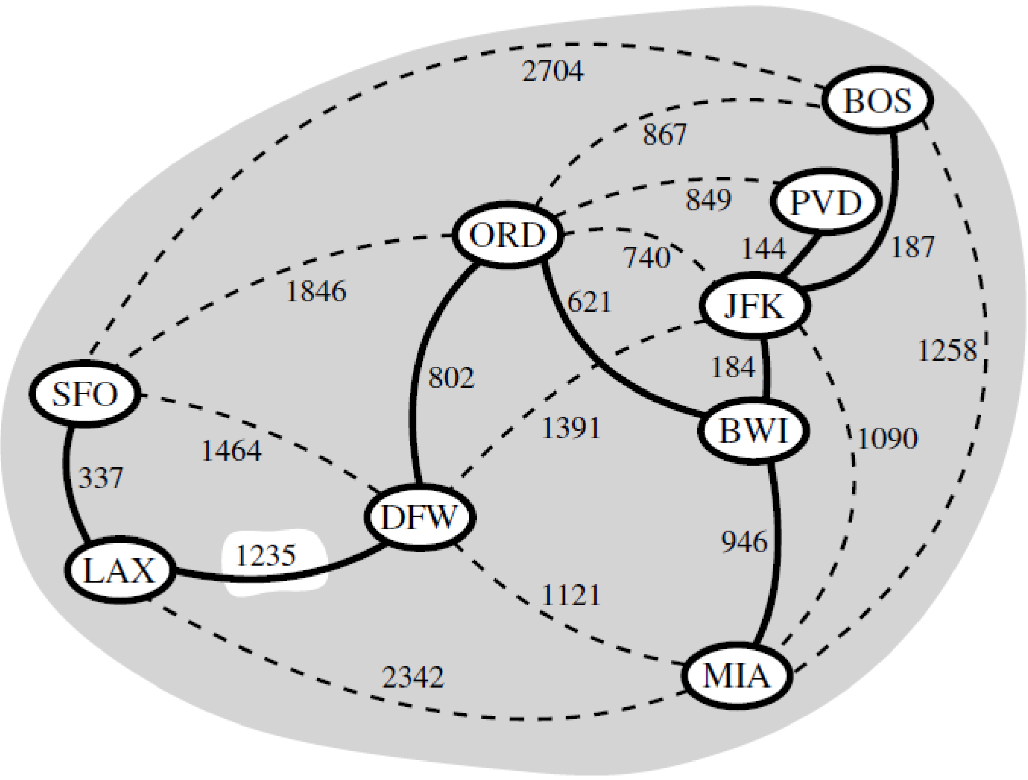
(l)

MINIMUM SPANNING TREES: KRUSKAL'S ALGORITHM (5)

- **Kruskal's algorithm example (2).**



(m)



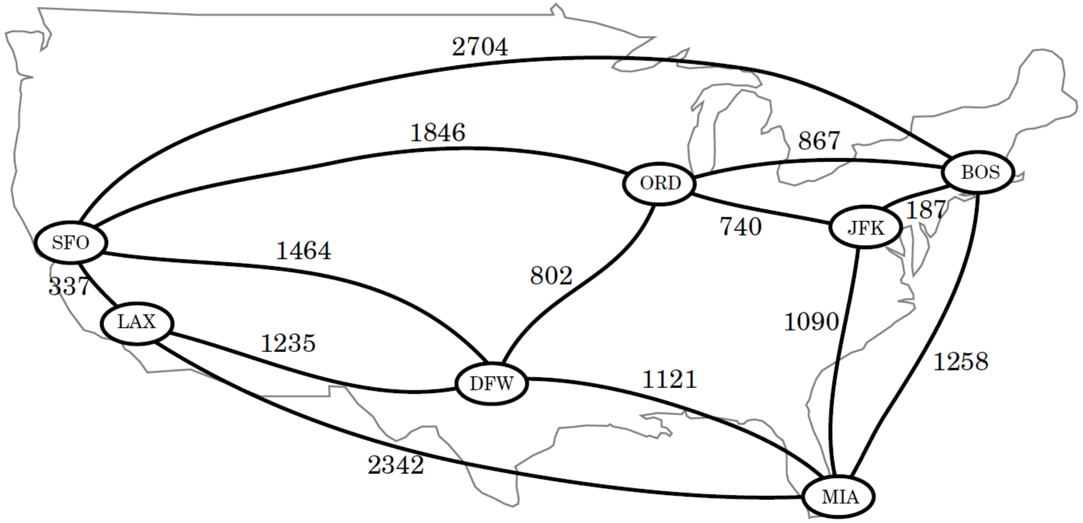
(n)

SHORTEST PATHS

- **DFS/BFS** algorithms can be used to find **shortest path** from one vertex to every other vertex.
  - **DFS** only works on **trees** / **BFS** only works for **unweighted** graphs.

- **Finding shortest path in weighted graph problem statement:**
  - Given **graph G**, find a **shortest path** from some **vertex s** to each other **vertex in G**, considering the **weights** on the edges as **distances**.

- **Notations:**
  - Path $\mathbf{P} = ((\mathrm{v}_0, \mathrm{v}_1), (\mathrm{v}_1, \mathrm{v}_2), \ldots, (\mathrm{vk}_{-1,} \mathrm{v}_k))$
  - Length/weight of path $w(P) = \sum_{i=0}^{k-1} w(vi, vi_{+} 1)$
  - Distance form vertex *u* to vertex *v* is $d(u, v)$.
    - Length of the minimum-length (**shortest**) path if it exists.
    - If no path between u and v, then $d(u, v) = \infty$



- Shortest path problem can be solved by **greedy methods**.
  - Repeatedly selecting the best choice from among those available in each iteration.

SHORTEST PATHS: DIJKSTRA'S ALGORITHM (1)

- **Dijkstra's algorithm for finding shortest paths in graph.**
  - **Greedy method** to solve **single-source shortest-path** problem.
  - Finds **shortest paths** from a given **source vertex** to all other **vertices** in the graph.
- **Algorithm process.**
  - **Iteratively** grows a **cluster** of vertices out of source **vertex s**.
  - Vertices **entering** the cluster in **order** of their **distances** from **vertex s**.
    - In each iteration, the next vertex chosen is the vertex **outside** the cluster that is **closest** to vertex s.
  - **Terminates** when no more vertices are **outside** the cluster.
    - Derived shortest path from source vertex s to every vertex of graph G that is reachable from s.
- Dijkstra's algorithm is based on the **edge relaxation** approach.

SHORTEST PATHS: DIJKSTRA'S ALGORITHM (2)

- **Dijkstra's algorithm edge relaxation process.**
  - Each vertex v in graph G is defined by distance $\mathbf{D[v]}$.
    - Approximate distance from source vertex s to vertex v.
    - The length of the best (**<u>so far</u>**) path from s to v.
  - Initially, $\mathbf{D[s]} = 0$, $\mathbf{D[v] = \infty}$, for each $\mathbf{v} \mathrel{!=} \mathbf{s}$, and cluster $\mathbf{C = \{\,\}}$.
  - At each iteration, select vertex u, not in C, with min $\mathbf{D[u]}$ and add it to cluster C.
  - Once vertex u is added to cluster C, perform **relaxation** procedure.
    - Update $\mathbf{D[v]}$ of each vertex v that is adjacent to vertex u and not in cluster C.
      - Checks if $\mathbf{D[v]}$ estimate can be improved to get closer to its true value.
  - Edge relaxation operation:
    - If $\mathbf{D[u] + w(u,v) < D[v]}$, then $\mathbf{D[v] = D[u] + w(u, v)}$.

SHORTEST PATHS: DIJKSTRA'S ALGORITHM (3)

- **Dijkstra's algorithm pseudocode.**

```
Dijkstra(G, s):
     Initialize D[s] = 0 and D[v] = ∞ for each vertex v != s

     Define set Q containing all vertices with their distances

     While Q is not empty do
            Remove vertex u with min D[u] from Q
            For each vertex v adjacent to u and still in Q do
                   If D[u] + w(u,v) < D[v] then
                          D[v] = D[u] + w(u,v)

     Return D[v] of each vertex v
```

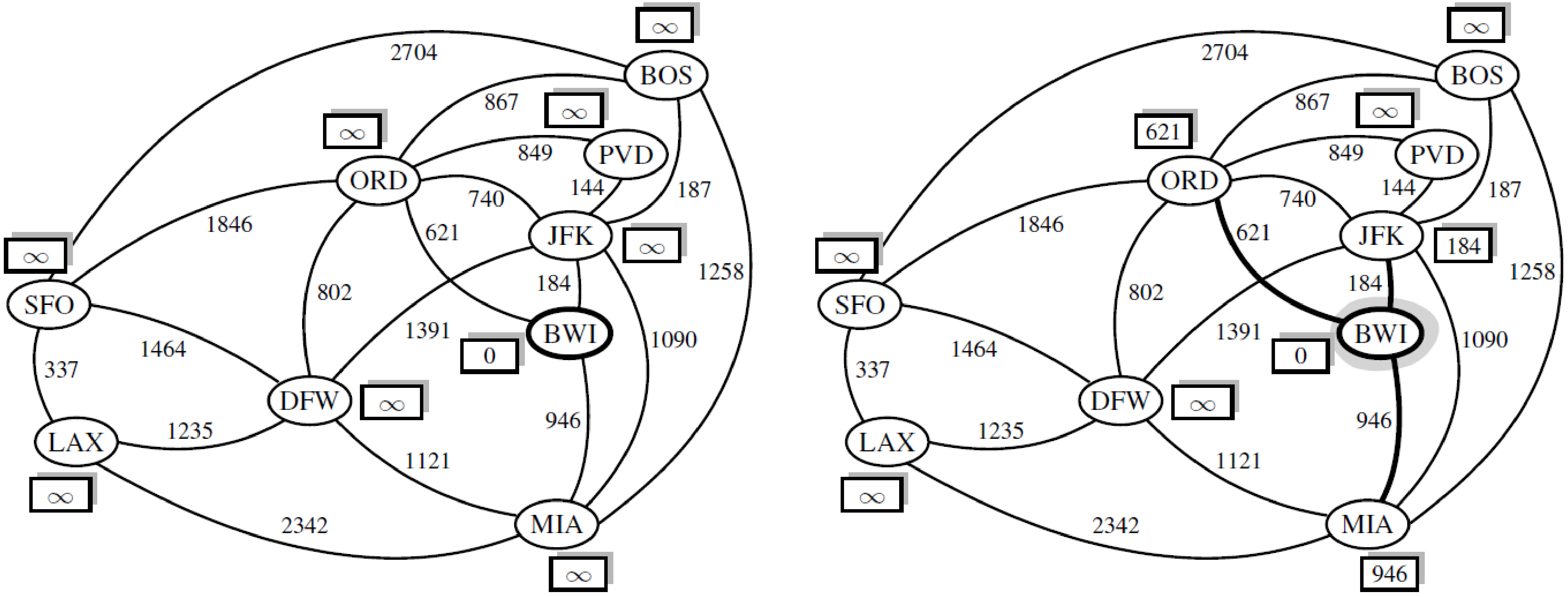
SHORTEST PATHS: DIJKSTRA'S ALGORITHM (4)

- **Dijkstra's algorithm complexity depends on:**
  - **Inserting** n vertices to the set Q.
  - **Finding** vertex with minimum distance & **removing** it from set Q.
    - Performed n times.
  - **Updating** the distances of vertices adjacent to "current" vertex.
    - Performed m times, where $m < n$.

- Overall, the complexity depends on the **data structure** for **set Q**.
  - **Implemented as heap: $O((n + m)\log n)$.**
    - Each operation runs in $O(\log n)$ time.
  - **Implemented as unsorted sequence: $O(n^2)$.**
    - $O(n)$ for storing and $O(n)$ for extracting vertices.

SHORTEST PATHS: DIJKSTRA'S ALGORITHM (5)

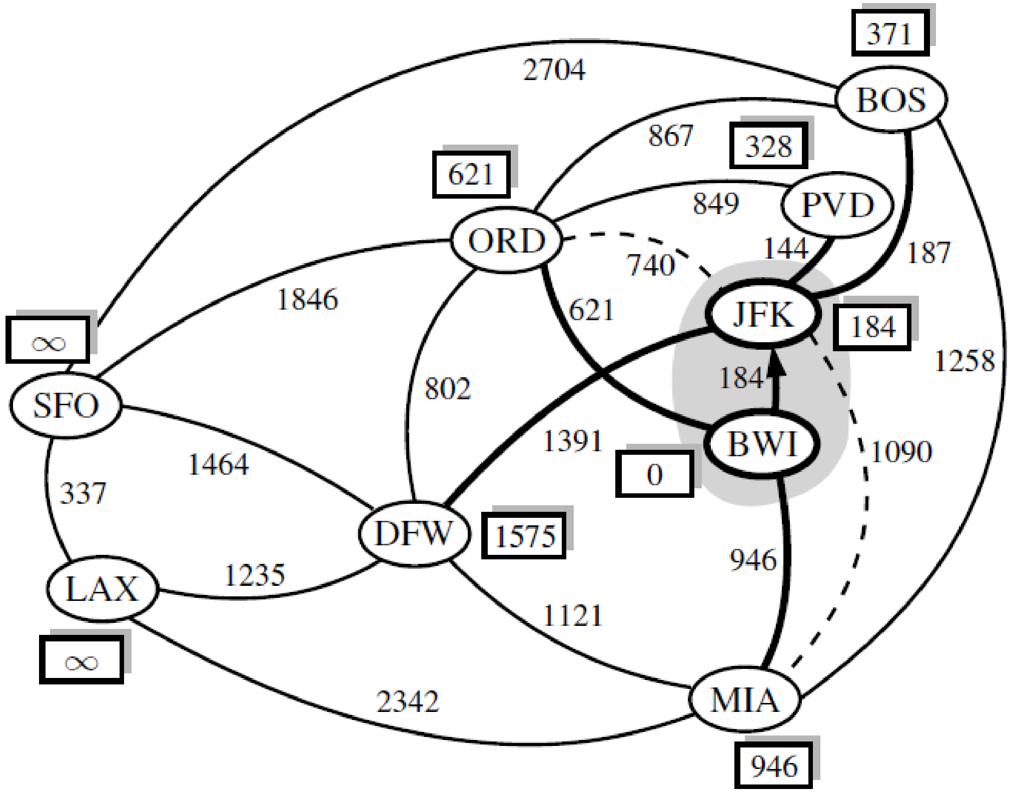
- **Dijkstra's algorithm example.**



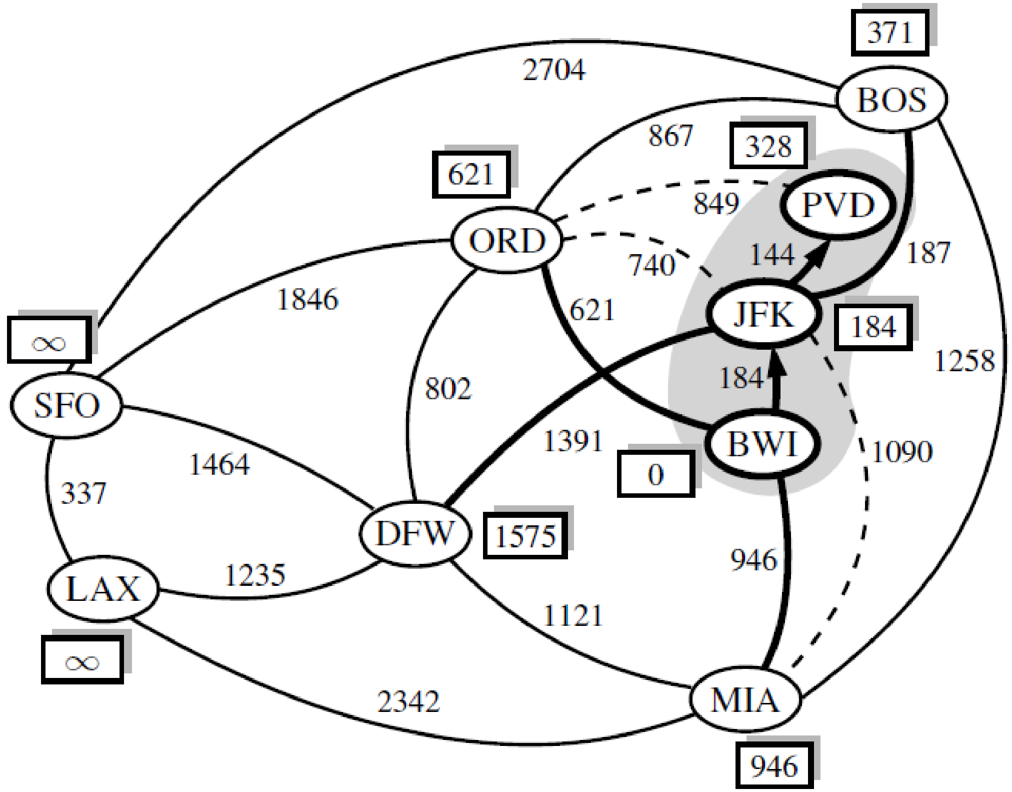
(a) (b)

SHORTEST PATHS: DIJKSTRA'S ALGORITHM (5)

- **Dijkstra's algorithm example.**



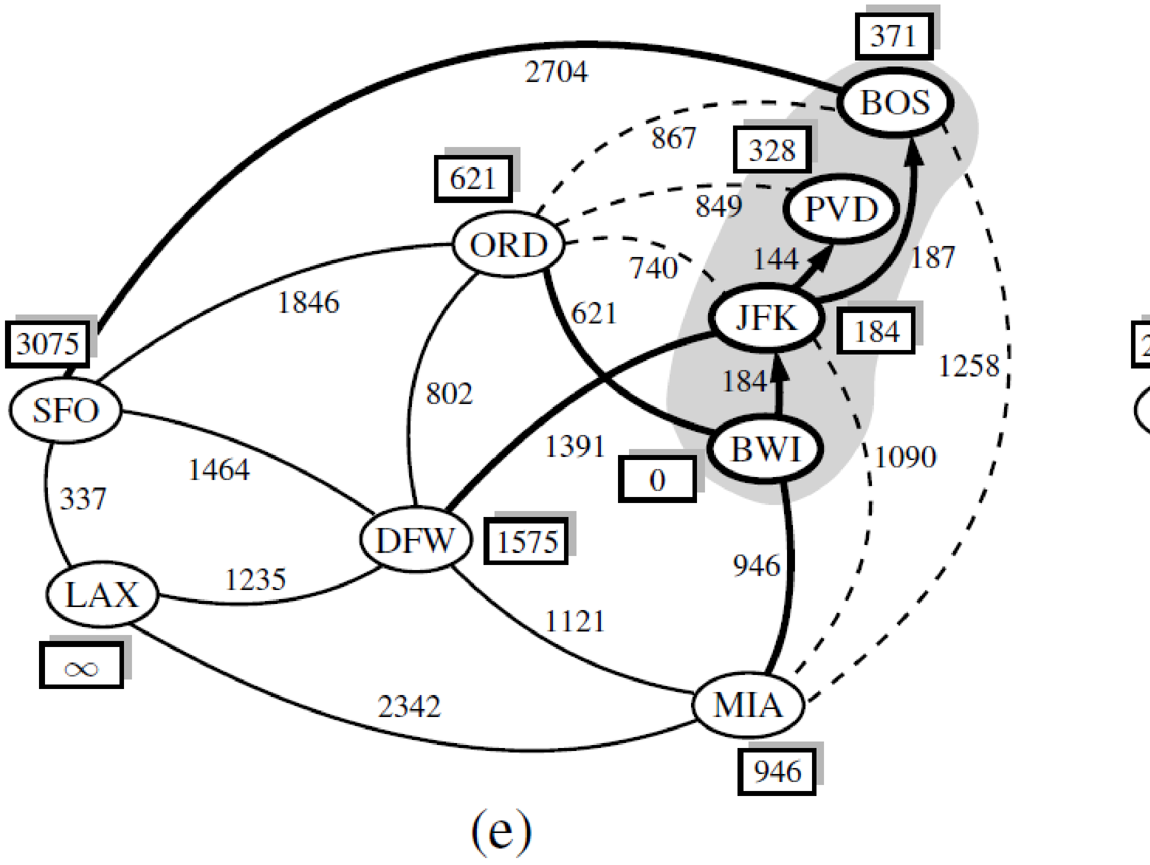
(c)



(d)

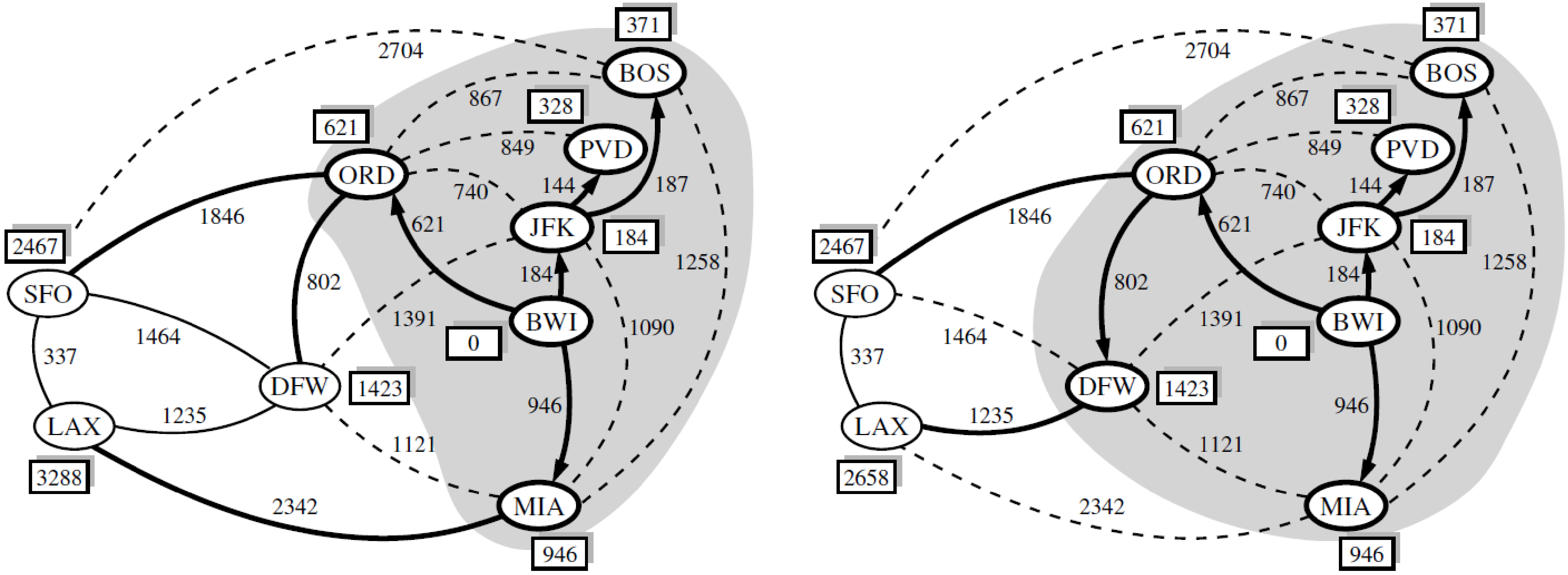
SHORTEST PATHS: DIJKSTRA'S ALGORITHM (5)

- **Dijkstra's algorithm example.**



SHORTEST PATHS: DIJKSTRA'S ALGORITHM (5)

- **Dijkstra's algorithm example.**



(g) (h)

SHORTEST PATHS: DIJKSTRA'S ALGORITHM (5)

- **Dijkstra's algorithm example.**