Mahidol University
Faculty of Information and Communication Technology

ITDS 122 Data Structures and Algorithm Analysis Lecture 14: Graph Algorithms

Asst. Prof. Dr. Ananta Srisuphab

2/2024

Teaching Lecture Hour : 30 Lab Hour : 30

- Recalling Python Coding
- Object-Oriented Programming with Python
- Algorithm Analysis
- Recursion
- Array-Based Sequences
- Stacks, Queues, Deques
- Linked Lists
- Review & checkpoint [Midterm]
- Trees and Search Trees *
- Maps and Hash Tables
- Graph Algorithms *
- Sorting
- Review & checkpoint [Final]

Graph Algorithms

- Transitive Closure
- Directed Acyclic Graphs (DAGs)
- Shortest Paths
- Minimum Spanning Tree (MST)

Transitive Closure

Introduction to Transitive Closure

- Goal: Determine reachability in a directed graph
- If vertex u can reach vertex v, then v is in the transitive closure of u
- Applications:
  - Routing & network analysis
  - Scheduling systems
  - Analyzing dependencies

Definition of Transitive Closure

- Transitive closure G* of a directed graph G:
  - Same vertex set as G
  - Edge (u, v) exists in G* iff there is a path from u to v in G
- Can include self-loops (path from u to u)

Motivation for Precomputing

- DFS/BFS answers reachability for a single source
- Inefficient for multiple queries
- Precompute transitive closure for constant-time reachability checks
- Efficient for dense graphs or matrix-based representations

Floyd-Warshall Algorithm

- Iteratively add paths by checking intermediate vertices
- Let vertices be v1, v2, …, vn
- For each k, add (i, j) if i → k and k → j exist
- Key idea: path i → k → j implies i → j

Pseudocode

Algorithm FloydWarshall($\vec{G}$):

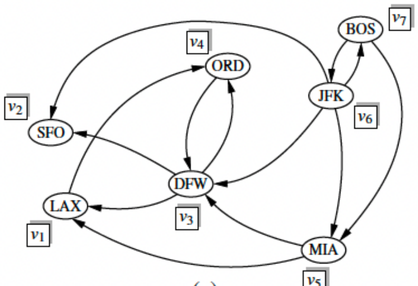
Input: A directed graph $\vec{G}$ with n vertices

Output: The transitive closure $\vec{G}^*$ of $\vec{G}$

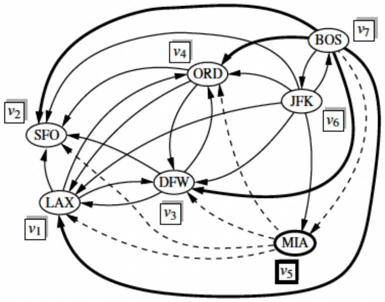
let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$
$\vec{G}_0 = \vec{G}$
for $k = 1$ to n **do**
    $\vec{G}_k = \vec{G}_{k-1}$
    for all i, j in $\{1, \ldots, n\}$ with $i \neq j$ and $i, j \neq k$ **do**
        if both edges (v_i, v_k) and (v_k, v_j) are in $\vec{G}_{k-1}$ **then**
            add edge (v_i, v_j) to $\vec{G}_k$ (if it is not already present)
return $\vec{G}_n$

Python Implementation

```python
class Graph:
    def __init__(self):
        self.adj = {}

    def add_vertex(self, v):
        if v not in self.adj:
            self.adj[v] = set()

    def add_edge(self, u, v):
        self.add_vertex(u)
        self.add_vertex(v)
        self.adj[u].add(v)

    def vertices(self):
        return list(self.adj.keys())

    def get_edge(self, u, v):
        return v if v in self.adj.get(u, set()) else None

    def insert_edge(self, u, v):
        self.adj[u].add(v)

    def __str__(self):
        return "\n".join(f"{u} -> {sorted(vs)}" for u, vs in self.adj.items())

```



(a)



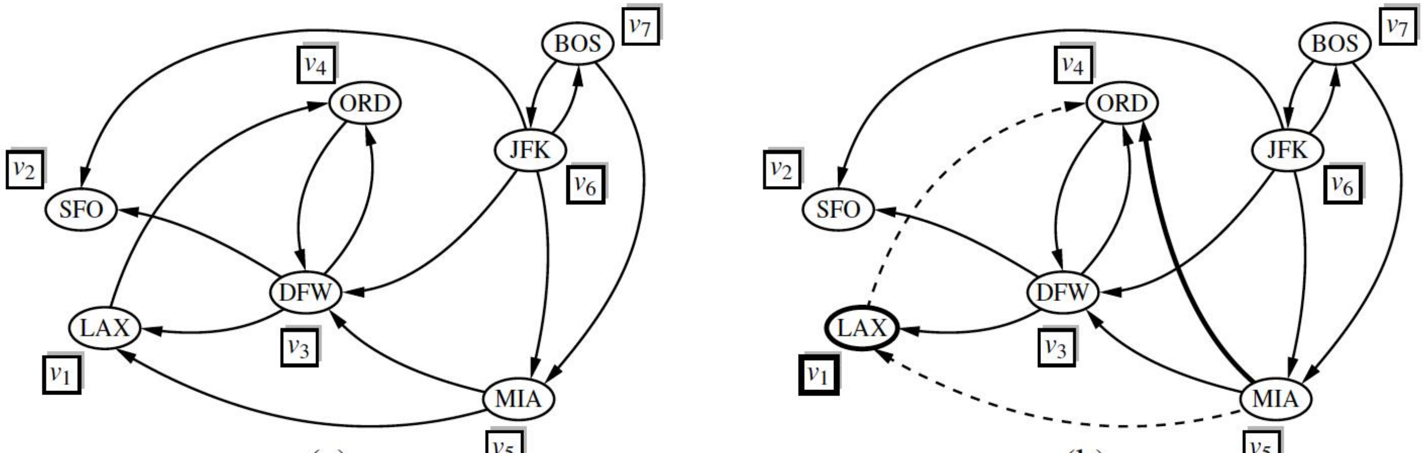
(f)

```python
def floyd_warshall(g):
    """Return a new graph that is the transitive closure of g."""
    closure = deepcopy(g)
    verts = list(closure.vertices())
    n = len(verts)

    for k in range(n):
        for i in range(n):
            if i != k and closure.get_edge(verts[i], verts[k]) is not None:
                for j in range(n):
                    if i != j != k and closure.get_edge(verts[k], verts[j]) is not None:
                        if closure.get_edge(verts[i], verts[j]) is None:
                            closure.insert_edge(verts[i], verts[j])

    return closure

```

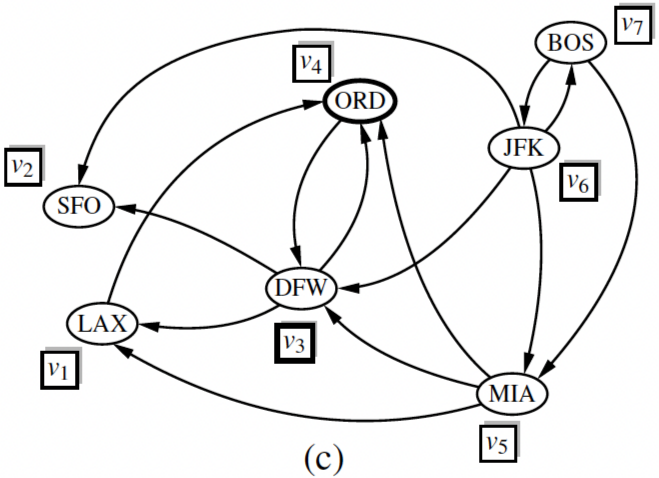
```python
#
g = Graph()
g.add_edge("LAX", "ORD")
g.add_edge("DFW", "LAX")
g.add_edge("DFW", "SFO")
g.add_edge("DFW", "ORD")
g.add_edge("ORD", "DFW")
g.add_edge("MIA", "LAX")
g.add_edge("MIA", "DFW")
g.add_edge("JFK", "DFW")
g.add_edge("JFK", "MIA")
g.add_edge("JFK", "BOS")
g.add_edge("BOS", "JFK")



if __name__ == "__main__":
    print("Original Graph:")
    print(g)

    closure = floyd_warshall(g)
    print("\nTransitive Closure:")
    print(closure)

```

```
Original Graph:
LAX -> ['ORD']
ORD -> ['DFW']
DFW -> ['LAX', 'ORD', 'SFO']
SFO -> []
MIA -> ['DFW', 'LAX']
JFK -> ['BOS', 'DFW', 'MIA']
BOS -> ['JFK']

Transitive Closure:
LAX -> ['DFW', 'ORD', 'SFO']
ORD -> ['DFW', 'LAX', 'SFO']
DFW -> ['LAX', 'ORD', 'SFO']
SFO -> []
MIA -> ['DFW', 'LAX', 'ORD', 'SFO']
JFK -> ['BOS', 'DFW', 'LAX', 'MIA', 'ORD', 'SFO']
BOS -> ['DFW', 'JFK', 'LAX', 'MIA', 'ORD', 'SFO']
```

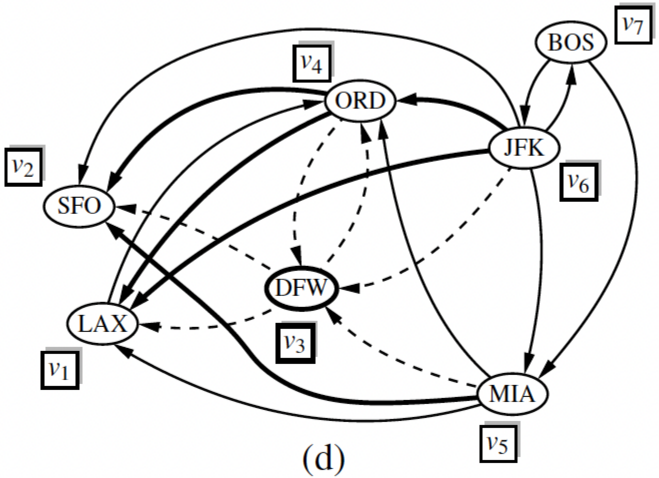
Visual Walkthrough



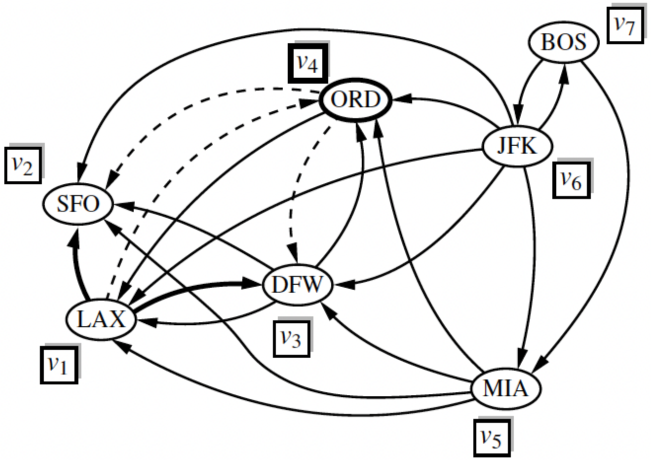
(a) (b)

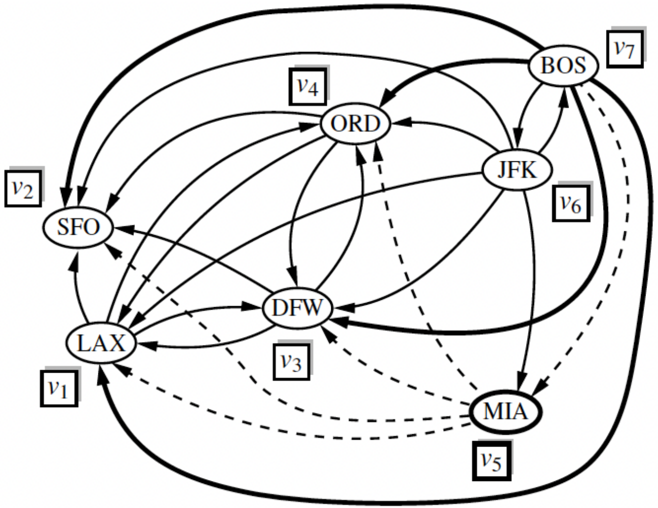

BOS
ORD
JFK
SFO
DFW
LAX
MIA
v_1
v_2
v_3
v_4
v_5
v_6
v_7

(c)


BOS
ORD
JFK
SFO
DFW
LAX
MIA
v_1
v_2
v_3
v_4
v_5
v_6
v_7

(d)


BOS
v_7
v_4
ORD
JFK
v_2
v_6
SFO
DFW
LAX
v_3
v_1
MIA
v_5
(e)


v_7
BOS
v_4
ORD
JFK
v_2
v_6
SFO
DFW
LAX
v_3
v_1
MIA
v_5
(f)

Summary

- Transitive closure = precomputed reachability
- Floyd-Warshall is conceptually simple and easy to implement
- Use for dense graphs or matrix-based representations
- Alternatives: DFS/BFS for sparse graphs (lower time but repeated calls)

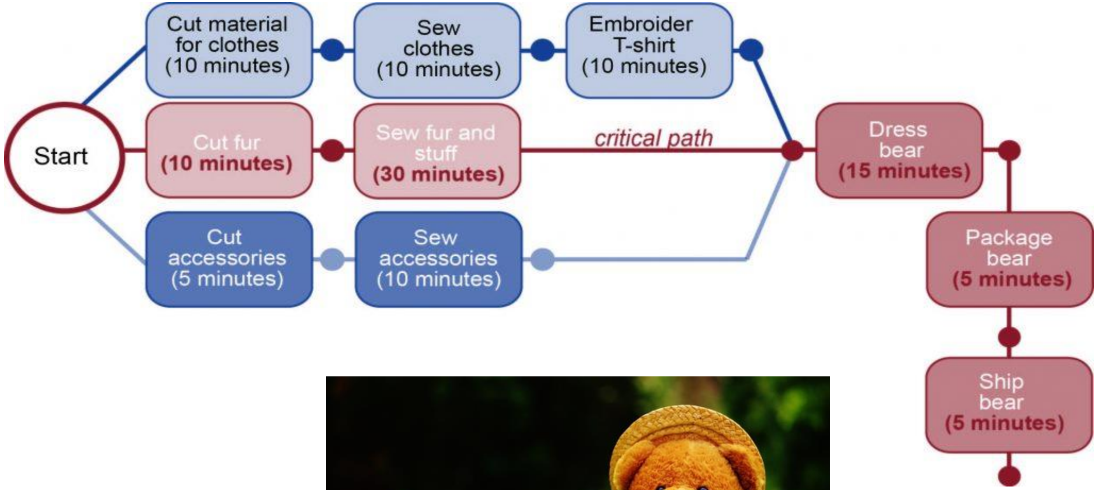
Directed Acyclic Graphs (DAGs)

Introduction to Directed Acyclic Graphs (DAGs)

- A Directed Acyclic Graph (DAG) is a directed graph with no cycles.
- Common applications:
  - Course prerequisites
  - Task scheduling
  - Inheritance hierarchies in OOP
- Important property: DAGs can be topologically ordered.

Motivation Example

- To manage a large project, break it into smaller tasks.
- Some tasks depend on others (e.g., foundation before walls).
- Represent tasks as vertices, dependencies as directed edges.
- If feasible, the graph must be a DAG.

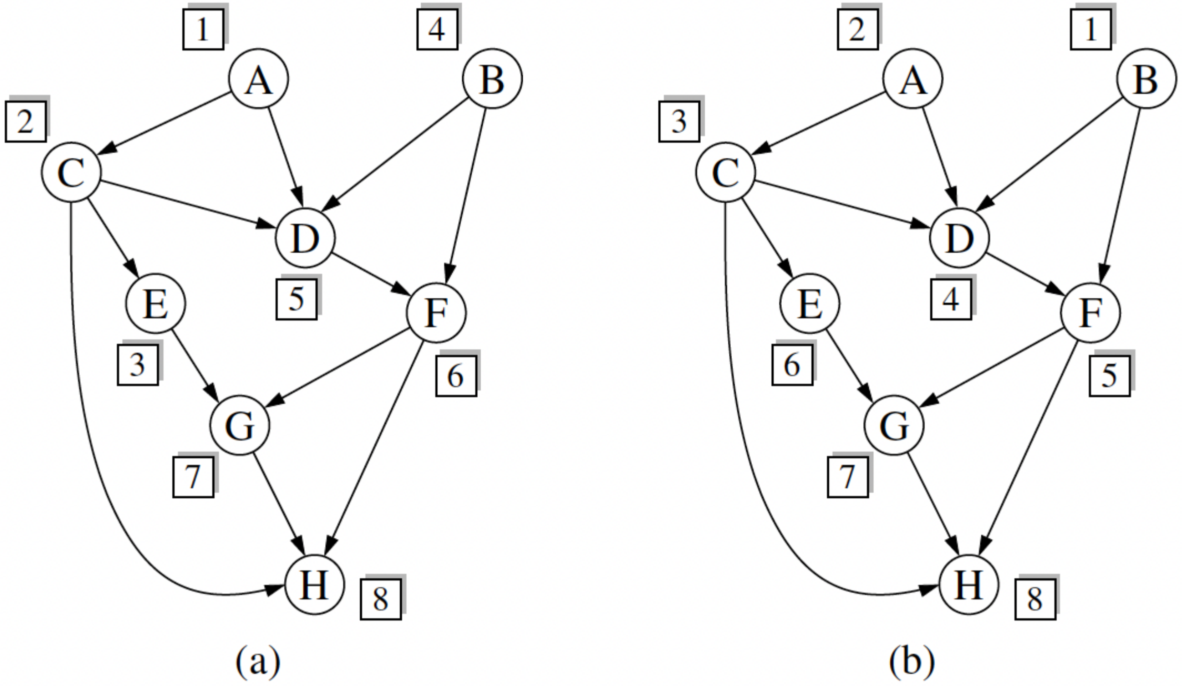


Topological Ordering

- A topological ordering of a DAG:
  - A sequence of vertices such that for every edge (u, v), u comes before v.
- Proposition: A graph has a topological order iff it is acyclic.

Visual Example

- Two possible topological orders for the same DAG:
  - Different valid sequences based on constraint satisfaction.
- Key point: multiple valid topological sorts may exist.



Topological Sorting Algorithm : Kahn's Algorithm

1. Initialize:
   - Count incoming edges (in-degree)
   - Add nodes with 0 in-degree to a ready list
2. While ready is not empty:
   - Remove node u from ready
   - Append u to topo
   - Decrease in-degree of u's neighbors
   - Add new nodes with 0 in-degree to ready

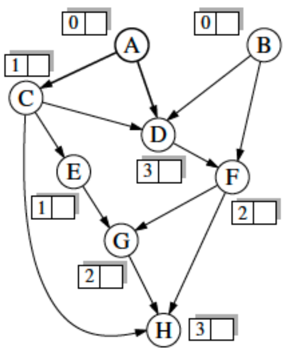
Python Implementation

```python
def topological_sort(g):
    """
    Return a list of vertices of directed acyclic graph g in topological order.
    If graph g has a cycle, the result will be incomplete.
    """
    topo = []                       # list of vertices placed in topological order
    ready = []                      # list of vertices that have no remaining constraints
    incount = {}                    # keep track of in-degree for each vertex

    for u in g:
        incount[u] = 0

    for u in g:
        for v in g[u]:
            incount[v] += 1

    for u in g:
        if incount[u] == 0:
            ready.append(u)

    while ready:
        u = ready.pop()
        topo.append(u)
        for v in g[u]:
            incount[v] -= 1
            if incount[v] == 0:
                ready.append(v)

    return topo
```

```python
graph = {
    'A': ['C','D'],
    'B': ['D','F'],
    'C': ['D', 'E','H'],
    'D': ['E', 'F'],
    'E': ['G'],
    'F': ['G', 'H'],
    'G': ['H'],
    'H': []
}

if __name__ == "__main__":
    ordering = topological_sort(graph)
    print("Topological Ordering:", ordering)
```

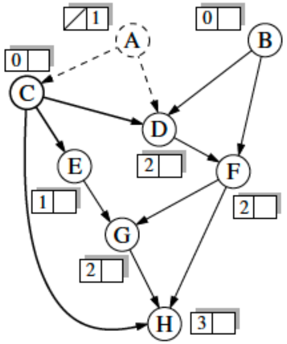
Example Walkthrough

- Highlight the step-by-step updates:
  - Track incount changes
  - Highlight which node is selected next
  - Show dashed edges as processed
- Visualize how nodes are added to final order.



(a)

🛠 **Ready List**: [A, B]
📄 **Topological Order**: []



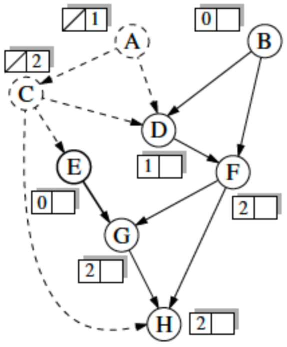
(b)

🛠 **Ready List**: [B, C]
📄 **Topological Order**: [A]

✅ **Selected Node:** A (first in the ready list)
📄 **Topological Order:** [A]
🛠 **Ready List:** [B]

📉 **Update in-degrees for neighbors of A:**
- A → C → incount[C] -= 1 → 1 → 0 ✅
- A → D → incount[D] -= 1 → 3 → 2

➕ **Add C to ready** (since incount[C] == 0)



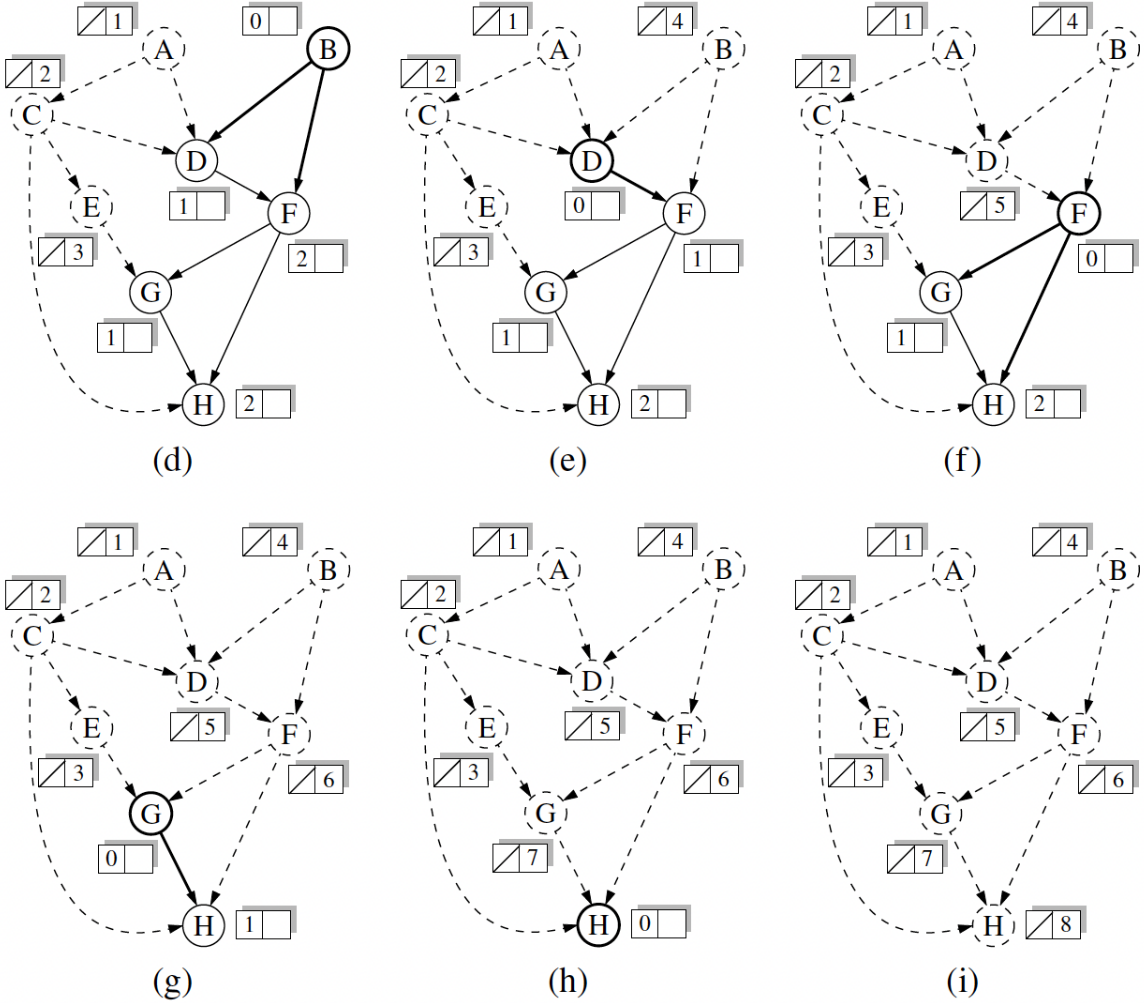
(c)

🛠 **Ready List**: [B, E]
📄 **Topological Order**: [A, C]

✅ **Selected Node:** C
🛠 **Ready List:** [B]

📉 **Update in-degrees for C's neighbors:**
- C → D → incount[D] -= 1 → 2 → 1
- C → E → incount[E] -= 1 → 1 → 0 ✅
- C → H → incount[H] -= 1 → 3 → 2

➕ **Add E to ready** (because incount[E] == 0)



1
0
A
B
2
C
D
E
1
F
3
2
G
1
H
2
(d)
1
4
A
B
2
C
D
E
0
F
3
1
G
1
H
2
(e)
1
4
A
B
2
C
D
E
5
F
3
0
G
1
H
2
(f)
1
4
A
B
2
C
D
E
5
F
3
6
G
0
H
1
(g)
1
4
A
B
2
C
D
E
5
F
3
6
G
7
H
0
(h)
1
4
A
B
2
C
D
E
5
F
3
6
G
7
H
8
(i)

Performance Analysis

- Let n = number of vertices, m = number of edges
- Time complexity: $O(n + m)$
- Space complexity: $O(n)$
- Efficient for large sparse graphs

Detecting Cycles

- If topo does not include all nodes:
  - The graph contains a cycle
  - Some nodes remain with non-zero in-degree

Summary

- DAGs are fundamental in scheduling and dependency resolution
- Topological sorting is a key algorithm on DAGs
- Python implementation uses Kahn's algorithm efficiently
- Always verify acyclicity to ensure a valid ordering

Shortest Paths

Introduction to Shortest Paths

- Goal: Find the shortest (minimum-weight) path between vertices
- Applications:
  - Navigation systems
  - Routing in computer networks
  - Task scheduling
- Breadth-First Search (BFS) only works when all edge weights are equal

Weighted Graphs

- A weighted graph assigns a numeric weight $w(e)$ to each edge e
- For edge (u, v), use notation: $w(u, v) = w(e)$
- Weights can represent:
  - Distance
  - Cost
  - Time

Path and Distance Definitions

- A path from u to v is a sequence of edges
- The length of a path = sum of its edge weights
- The shortest path from u to v is a path with minimum total weight
- If no path exists: $d(u, v) = \infty$

Valid Weights for Shortest Path Algorithms

- Must use nonnegative edge weights for Dijkstra
- Negative-weight edges allow cycles that reduce cost arbitrarily
- Algorithms must avoid revisiting same nodes with lower cost in cycles

Dijkstra's Algorithm Overview

- Greedy algorithm for single-source shortest paths
- Always picks vertex with smallest tentative distance (D[v])
- Expands a "cloud" of known shortest paths from the source
- Relaxes edges from each newly added vertex

Edge Relaxation Explained

- For each edge (u, v):

```
if D[v] > D[u] + w(u, v):
    D[v] = D[u] + w(u, v)
```

- Ensures shortest known distance is improved
- D[v] values always shrink or stay the same

Dijkstra Pseudocode

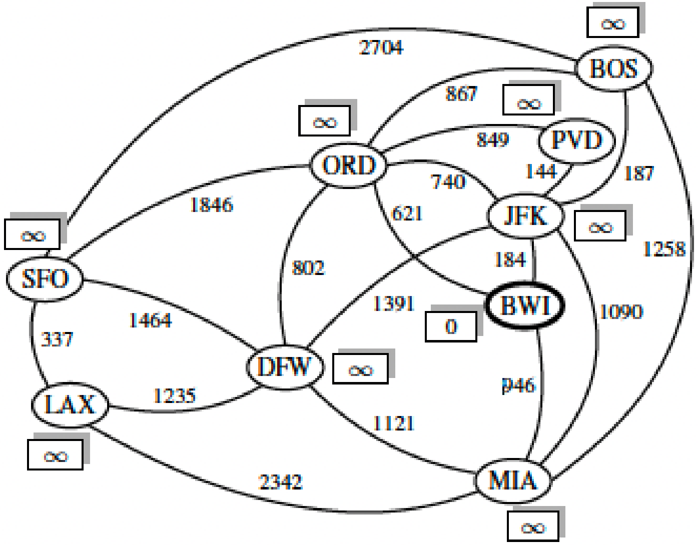
Algorithm ShortestPath(G, s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

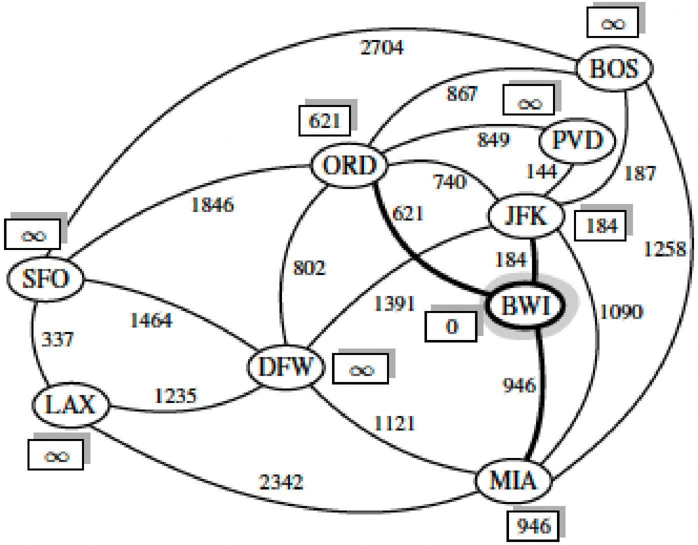
Output: The length of a shortest path from s to v for each vertex v of G.

```
Initialize D[s] = 0 and D[v] = ∞ for each vertex v ≠ s.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty do
   {pull a new vertex u into the cloud}
   u = value returned by Q.remove_min()
   for each vertex v adjacent to u such that v is in Q do
      {perform the relaxation procedure on edge (u,v)}
      if D[u] + w(u,v) < D[v] then
         D[v] = D[u] + w(u,v)
         Change to D[v] the key of vertex v in Q.
return the label D[v] of each vertex v
```

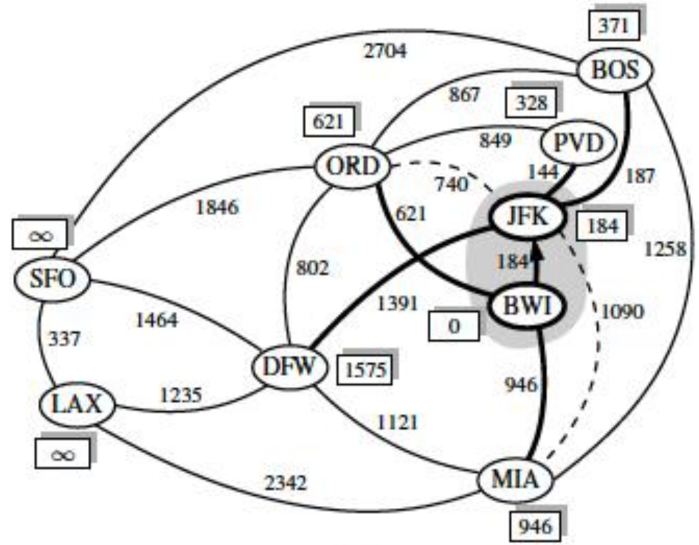
Example Execution

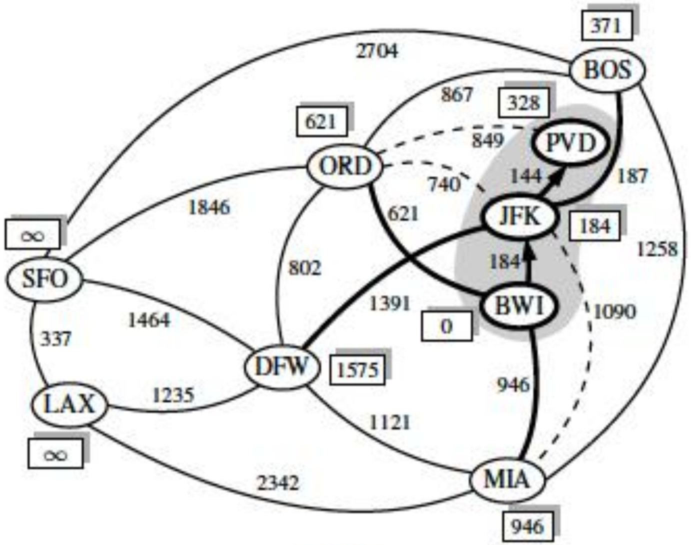


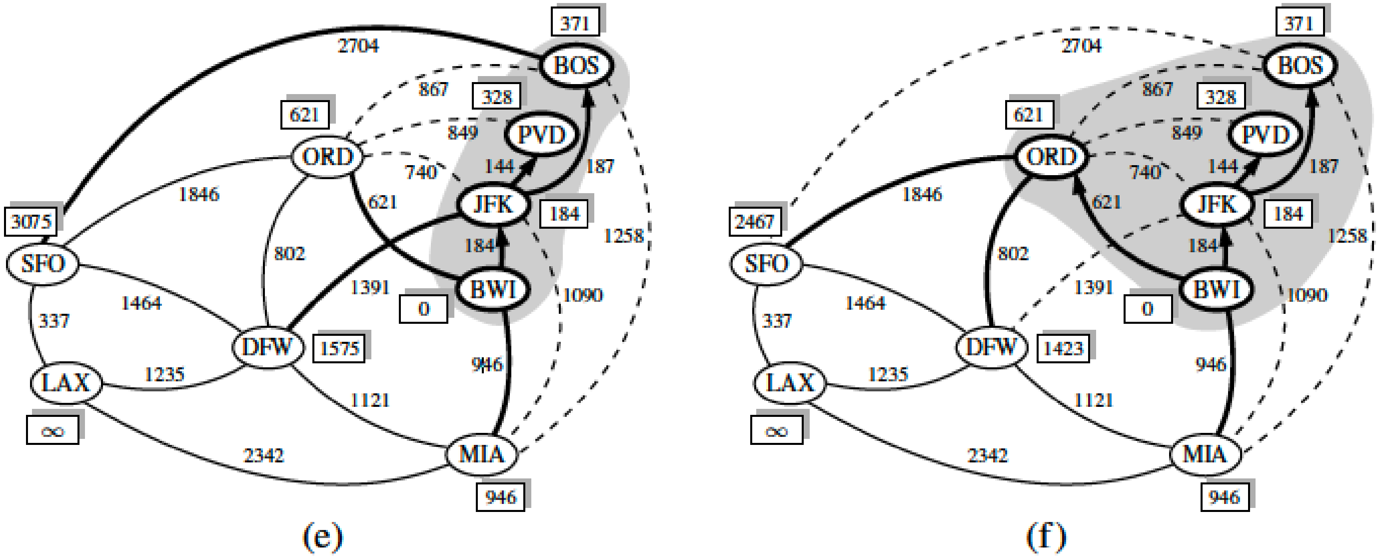
(a)



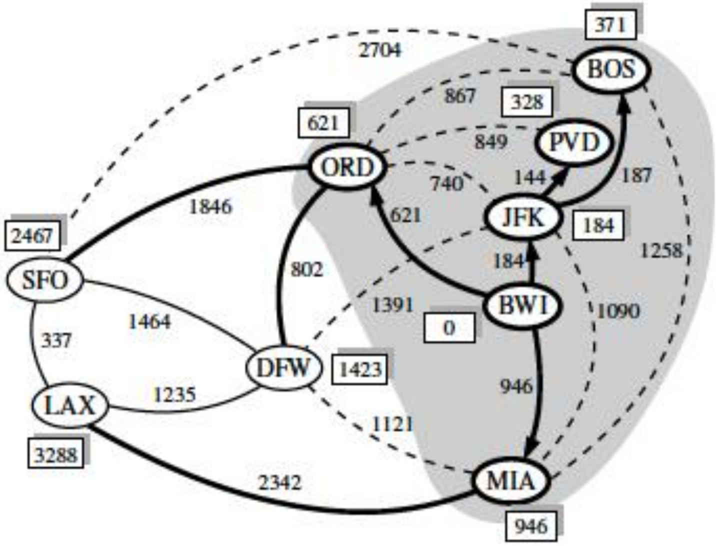
(b)


371
2704
BOS
867
328
621
849
PVD
ORD
144
740
187
1846
621
JFK
184
∞
1258
SFO
802
184
BWI
1391
1090
1464
0
337
DFW
1575
946
LAX
1235
1121
∞
MIA
2342
946
(c)


371
2704
BOS
867
328
621
849
PVD
ORD
144
740
187
1846
621
JFK
184
∞
1258
SFO
802
184
BWI
1391
1090
1464
0
337
DFW
1575
946
LAX
1235
1121
∞
MIA
2342
946
(d)

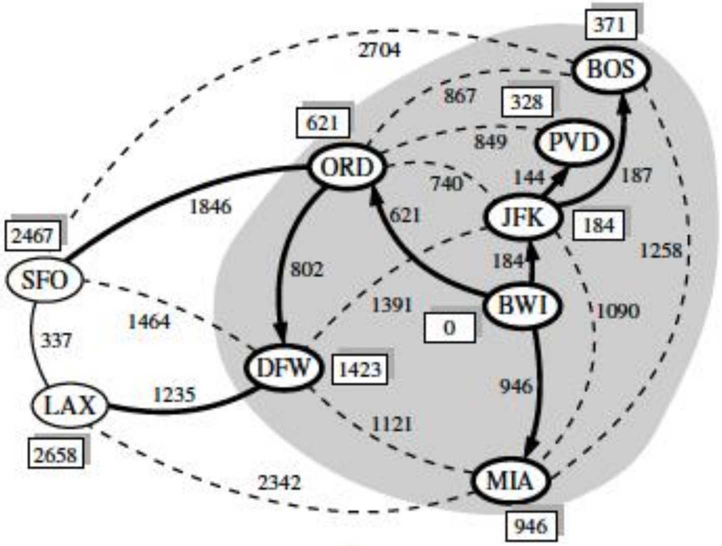


371
BOS
2704
867
328
PVD
621
ORD
849
740
144
187
3075
1846
JFK
184
621
1258
SFO
802
184
BWI
1391
0
1090
337
1464
DFW
1575
LAX
1235
946
1121
∞
2342
MIA
946
(e)
371
BOS
2704
867
328
PVD
621
ORD
849
740
144
187
2467
1846
JFK
184
621
1258
SFO
802
184
BWI
1391
0
1090
337
1464
DFW
1423
LAX
1235
946
1121
∞
2342
MIA
946
(f)



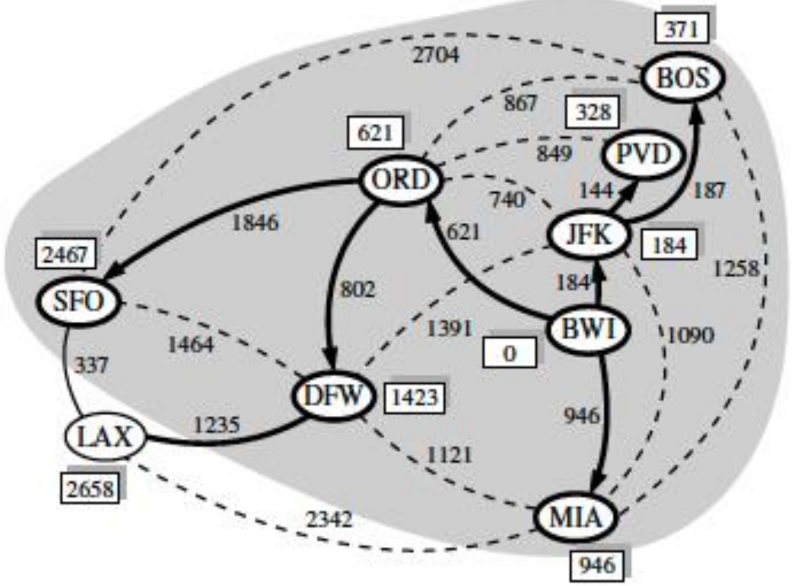
371
BOS
2704
867
328
621
PVD
849
ORD
740
144
187
1846
621
JFK
184
2467
1258
184
SFO
802
1391
BWI
1090
1464
0
337
DFW
1423
946
LAX
1235
1121
3288
2342
MIA
946

(g)

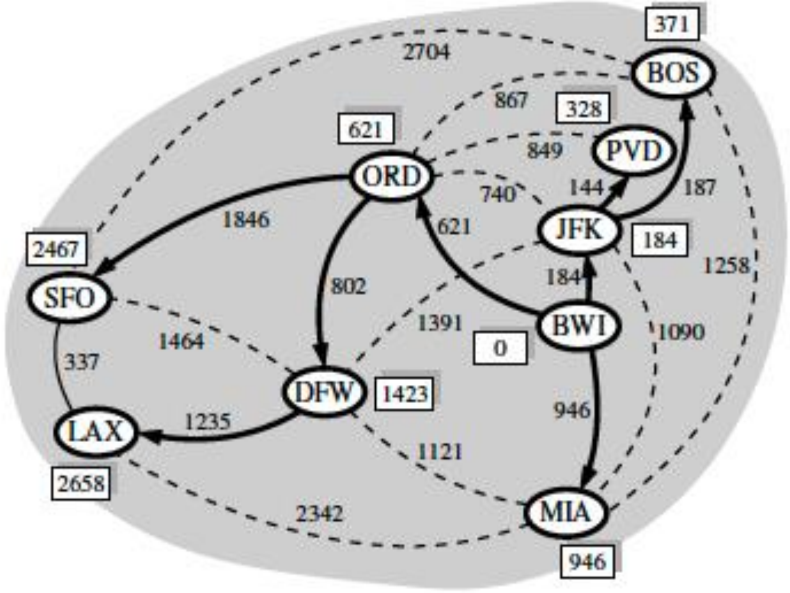


371
BOS
2704
867
328
621
PVD
849
ORD
740
144
187
1846
621
JFK
184
2467
1258
184
SFO
802
1391
BWI
1090
1464
0
337
DFW
1423
946
LAX
1235
1121
2658
2342
MIA
946

(h)


371
BOS
2704
867
328
PVD
621
ORD
849
740
144
187
JFK
184
621
1846
2467
SFO
802
184
1258
1391
BWI
0
1090
1464
337
DFW
1423
LAX
1235
946
1121
2658
2342
MIA
946

(i)


371
BOS
2704
867
328
PVD
621
ORD
849
740
144
187
JFK
184
621
1846
2467
SFO
802
184
1258
1391
BWI
0
1090
1464
337
DFW
1423
LAX
1235
946
1121
2658
2342
MIA
946

(j)

Reconstructing Shortest Paths

- Dijkstra gives shortest distance, but not actual path
- To reconstruct:
  - For each $v \neq s$, find edge (u, v) such that $D[u] + w(u, v) = D[v]$
- Builds the shortest-path tree

Summary

- Dijkstra’s algorithm solves the single-source shortest path problem
- Assumes nonnegative weights
- Builds both distance map and shortest-path tree
- Efficient and correct with proper data structures

Minimum Spanning Tree (MST)

Minimum Spanning Tree (MST)

- Connect all vertices of a weighted, undirected graph
- Use minimum total edge weight
- Real-world: Lay out cable to connect offices with minimal cost

Problem Definition

- Given undirected graph G = (V, E) with weights w(e)
- Find tree T ⊆ E that spans all V with minimal:

```
w(T) = Σ w(u, v)   for all (u,v) in T
```

- T is a Minimum Spanning Tree (MST)

Prim-Jarník Algorithm - Concept

- Grow MST from an arbitrary start vertex
- Use a greedy strategy: always add lightest edge to expand MST
- Very similar structure to Dijkstra's Algorithm

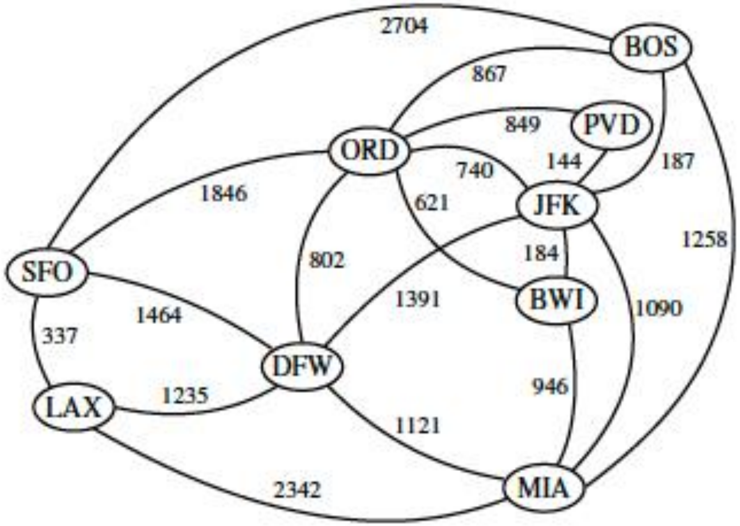
Prim-Jarník Pseudocode

Algorithm PrimJarnik(G):

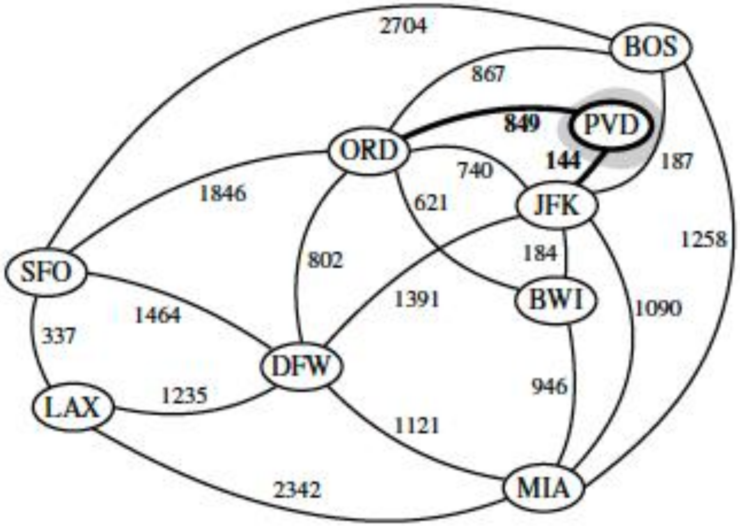
  Input: An undirected, weighted, connected graph G with n vertices and m edges
  Output: A minimum spanning tree T for G

Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  (u, e) = value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

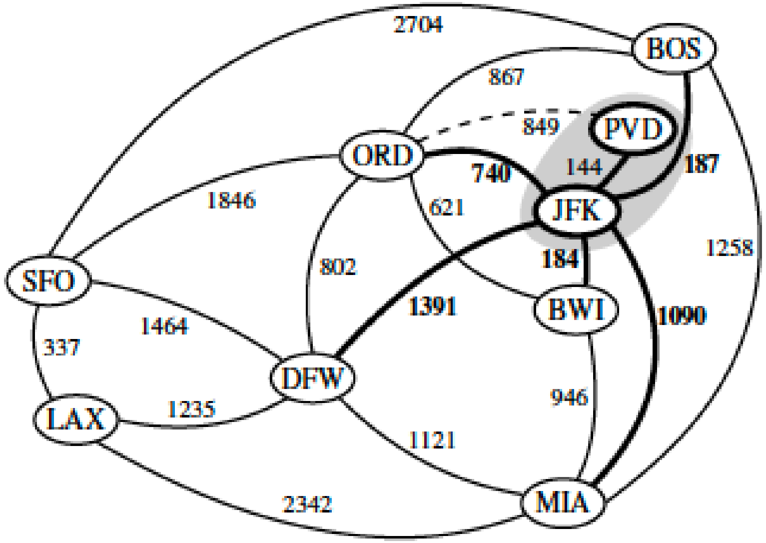
MST Illustration



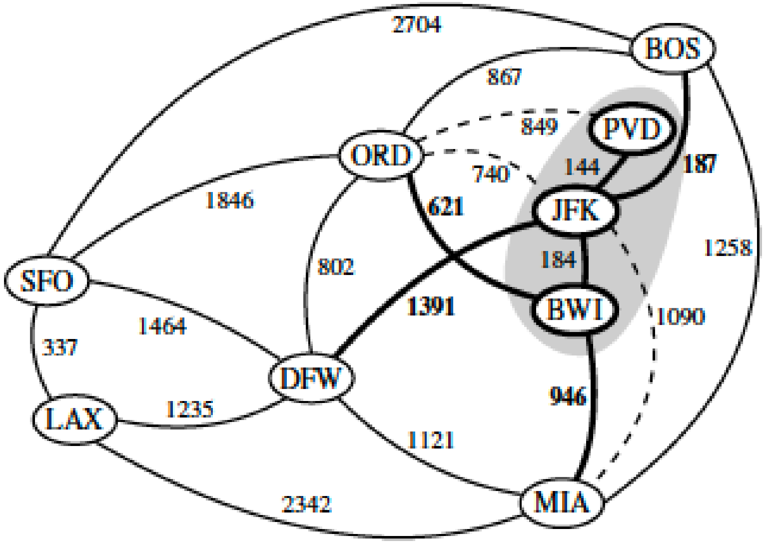
(a)



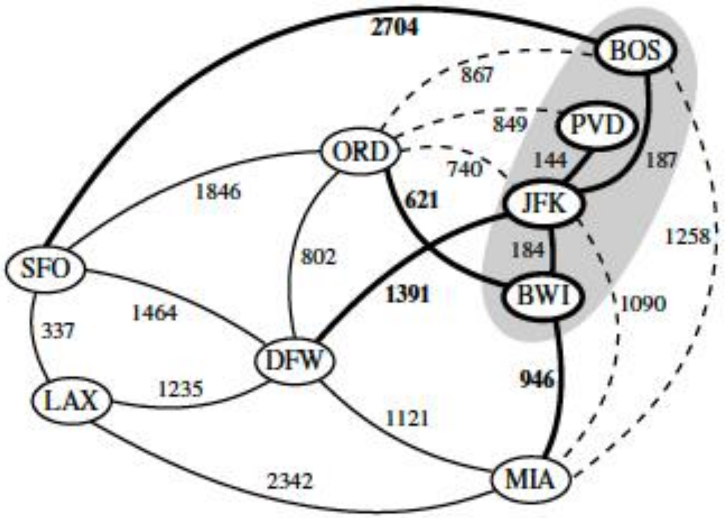
(b)


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

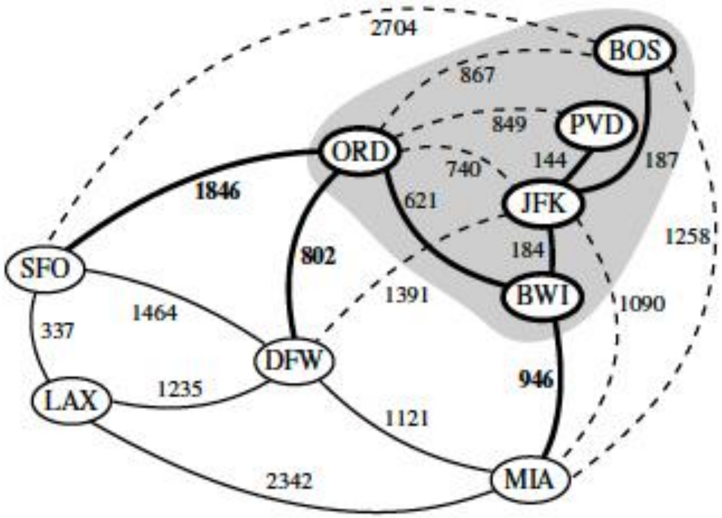
(c)


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

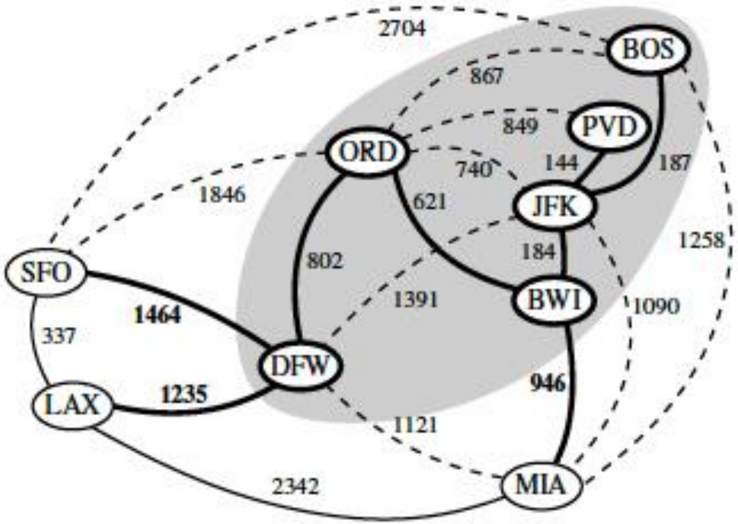
(d)


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

(e)

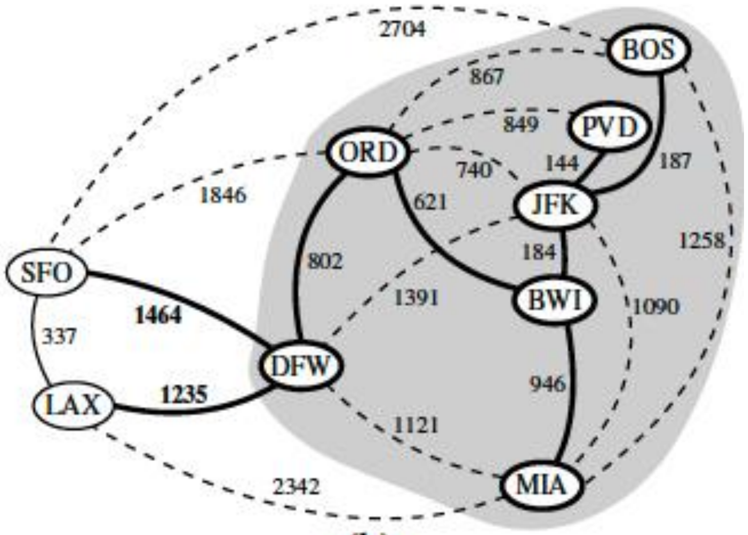

BOS
PVD
ORD
JFK
BWI
SFO
DFW
LAX
MIA
2704
867
849
144
187
740
1846
621
1258
184
802
1391
1090
1464
337
946
1235
1121
2342

(f)



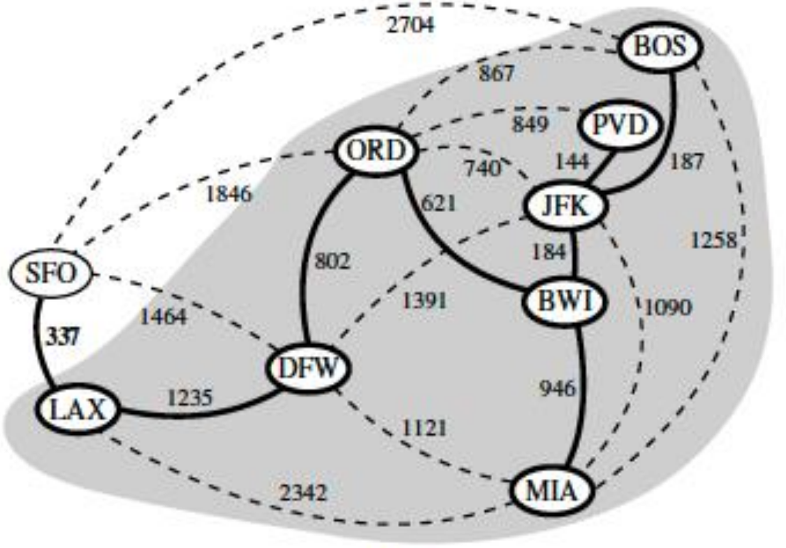
2704
BOS
867
PVD
849
ORD
740
144
187
1846
621
JFK
1258
SFO
802
184
1391
BWI
1090
1464
337
DFW
946
LAX
1235
1121
2342
MIA

(g)

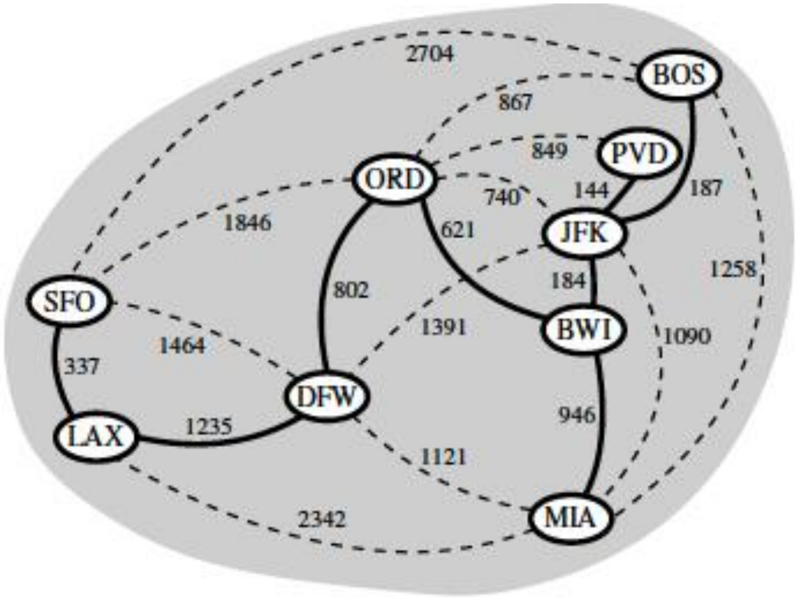


2704
BOS
867
PVD
849
ORD
740
144
187
1846
621
JFK
1258
SFO
802
184
1391
BWI
1090
1464
337
DFW
946
LAX
1235
1121
2342
MIA

(h)


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

(i)


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

(j)

Kruskal's Algorithm for MST

- Greedy algorithm for Minimum Spanning Tree (MST)
- Builds MST by adding lightest edges one-by-one
- Keeps growing a forest (set of trees), until connected

Kruskal's Algorithm Concept

- Start: All vertices are individual clusters (singleton sets)
- Sort edges by weight (min → max)
- For each edge (u, v):
  - If u and v are in different clusters → add edge to MST
  - Merge their clusters
  - If u and v are already connected → skip (to avoid cycle)

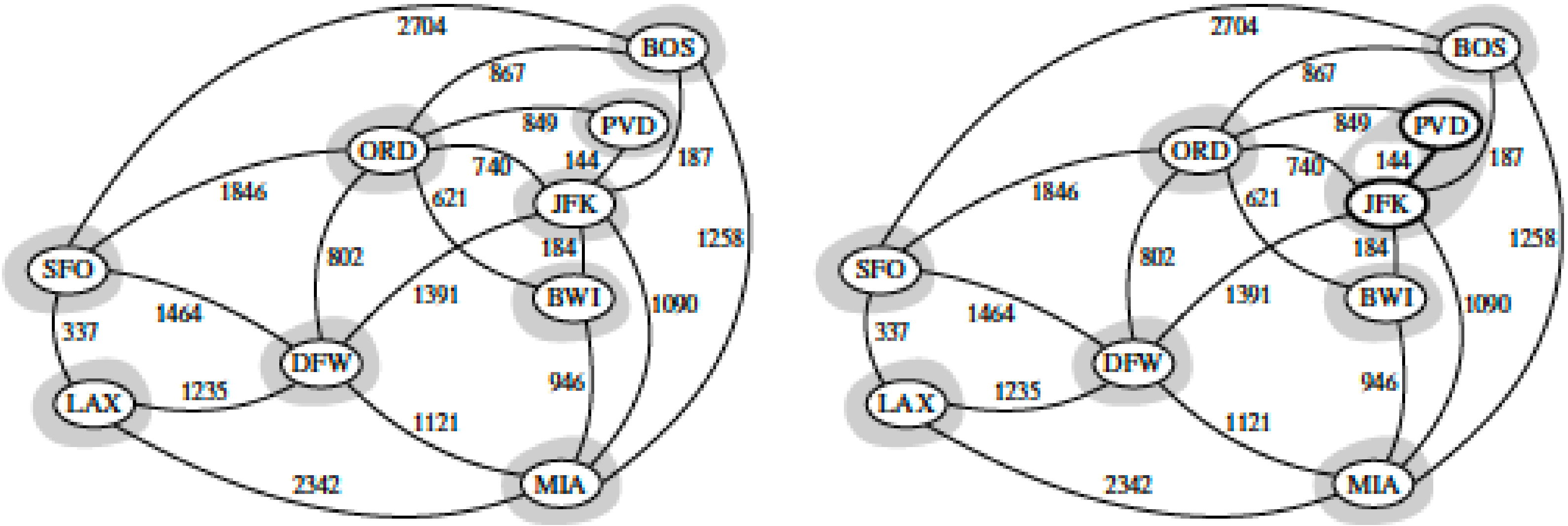
Kruskal's Pseudocode

Algorithm Kruskal(G):

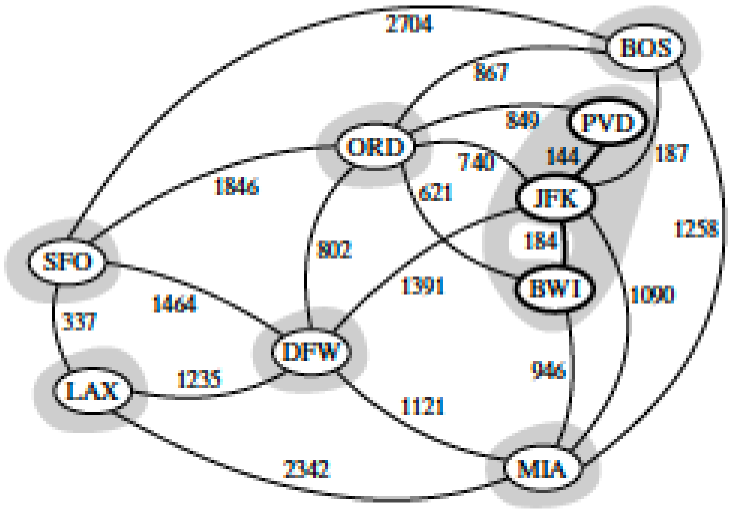
  Input: A simple connected weighted graph G with n vertices and m edges
  Output: A minimum spanning tree T for G

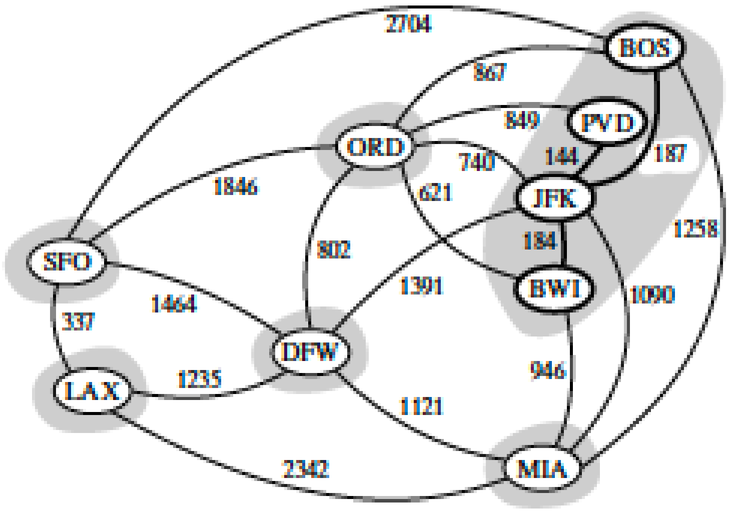
for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ $\{T$ will ultimately contain the edges of the MST$\}$
while T has fewer than $n-1$ edges **do**
  (u,v) = value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u,v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

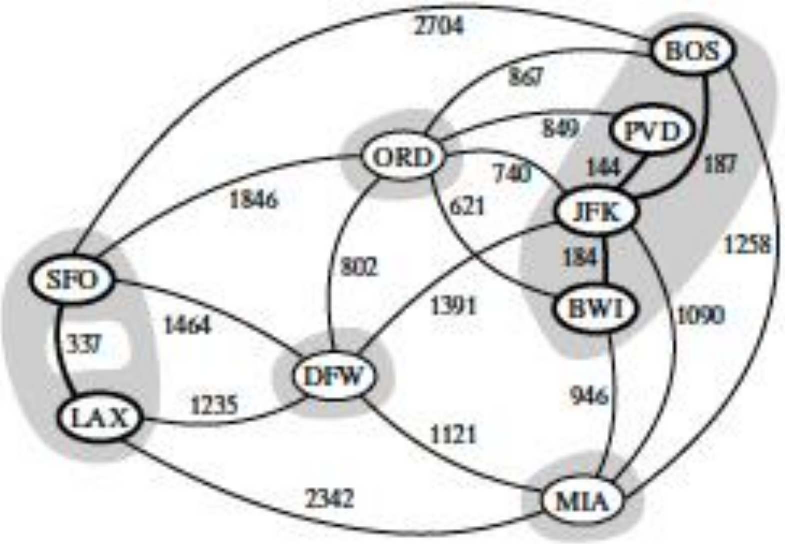
Kruskal’s MST algorithm Illustration



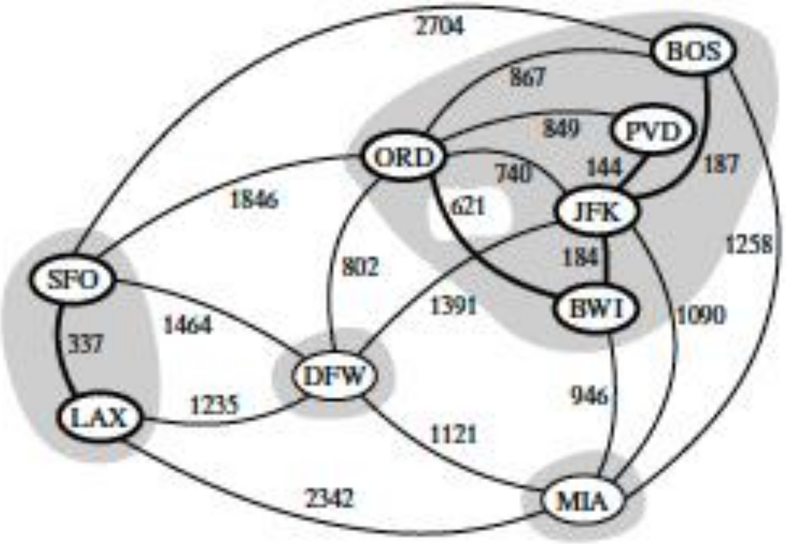
(a) (b)


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

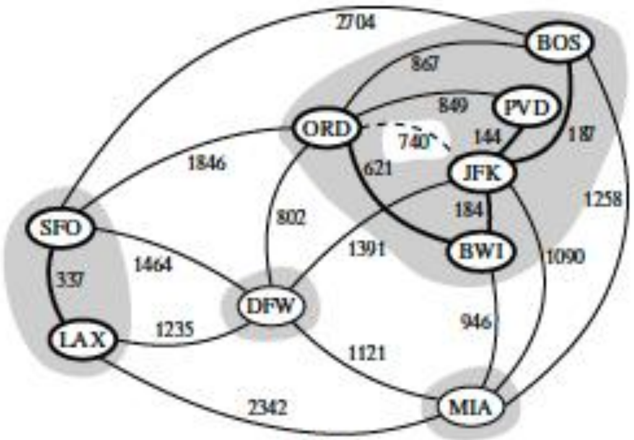

BOS
PVD
ORD
JFK
SFO
BWI
DFW
LAX
MIA
2704
867
849
740
144
187
1846
621
184
1258
802
1391
1090
1464
337
946
1235
1121
2342
(d)


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

(e)

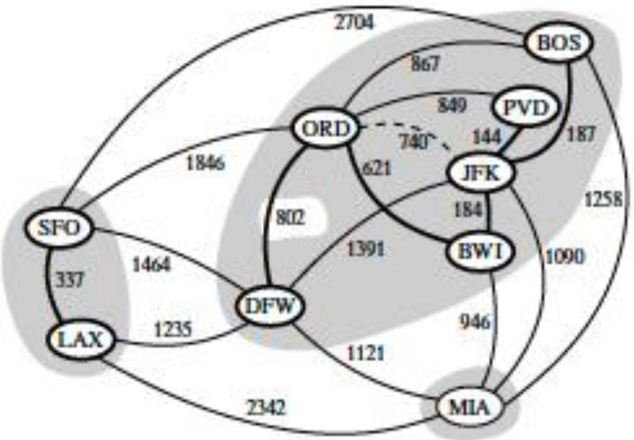

2704
BOS
867
849
PVD
ORD
740
144
187
1846
621
JFK
1258
802
184
SFO
BWI
1391
1090
1464
337
DFW
946
1235
LAX
1121
2342
MIA

(f)



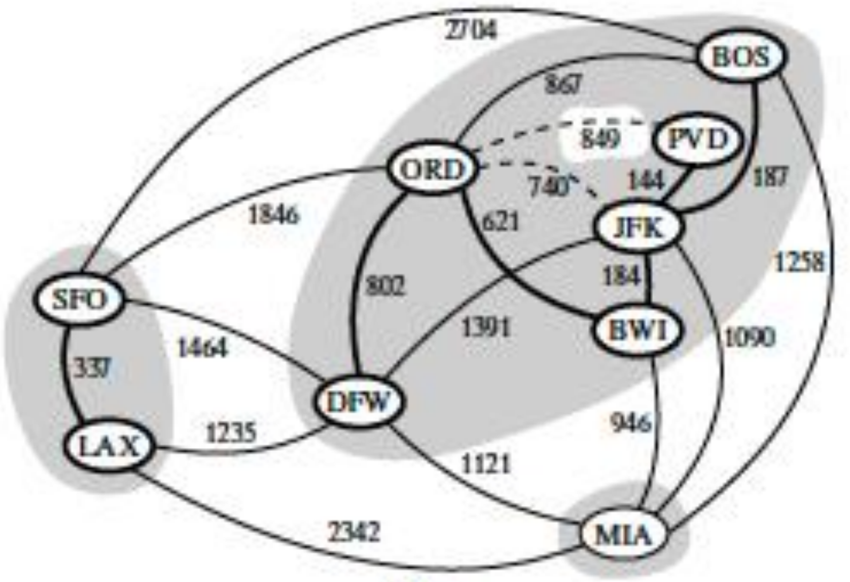
2704
BOS
867
849
PVD
ORD
740
144
187
1846
621
JFK
1258
184
SFO
802
1391
BWI
1090
1464
337
DFW
946
LAX
1235
1121
2342
MIA

(g)

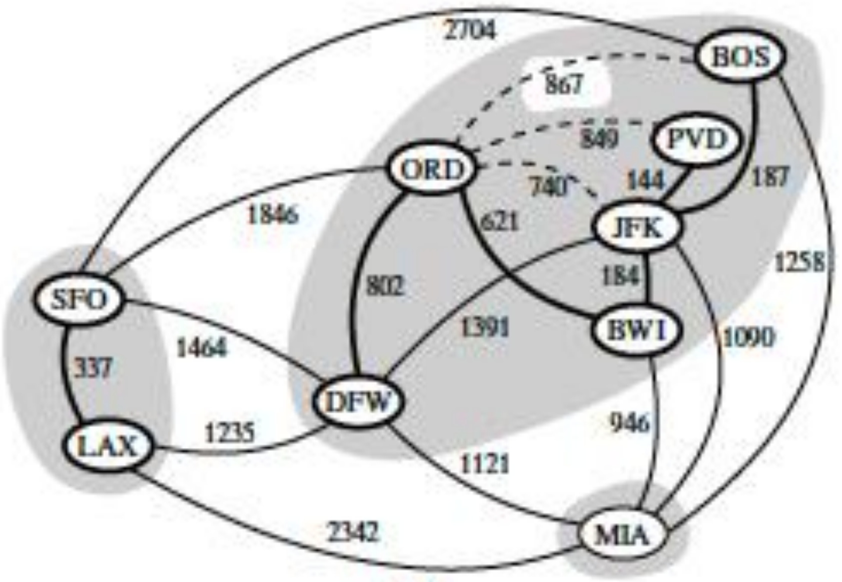


2704
BOS
867
849
PVD
ORD
740
144
187
1846
621
JFK
1258
184
SFO
802
1391
BWI
1090
1464
337
DFW
946
LAX
1235
1121
2342
MIA

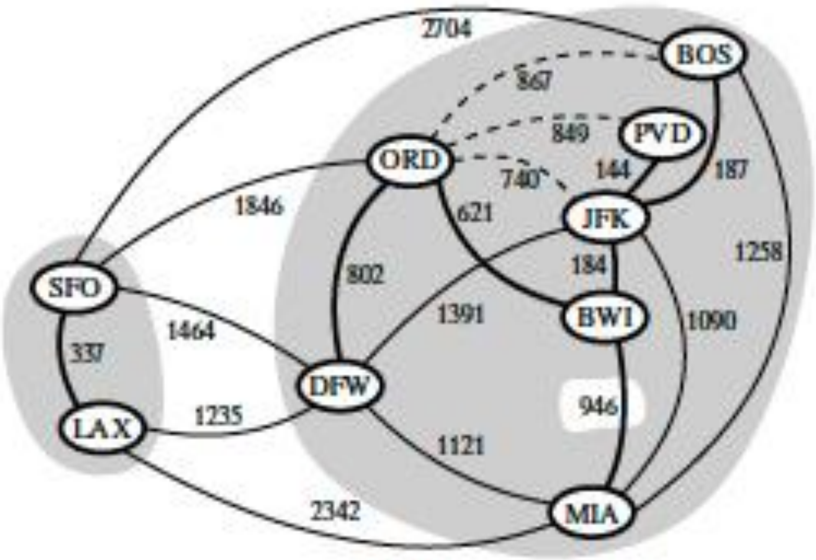
(h)


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

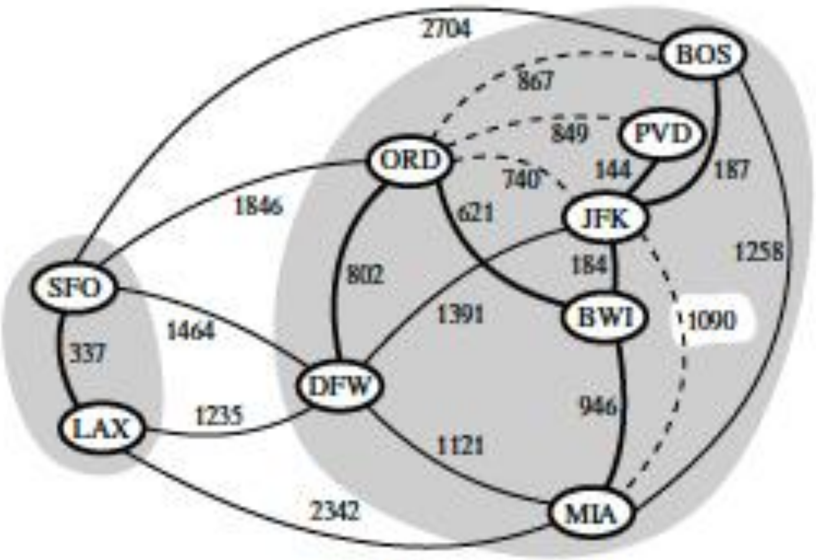
(i)


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

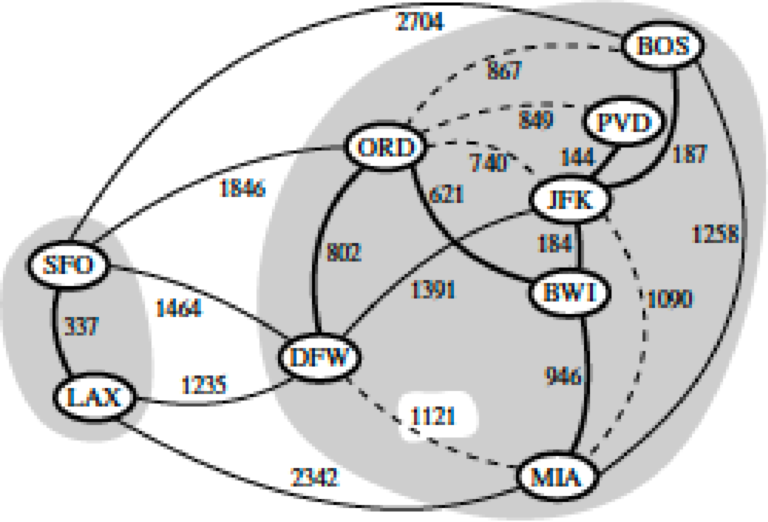
(j)

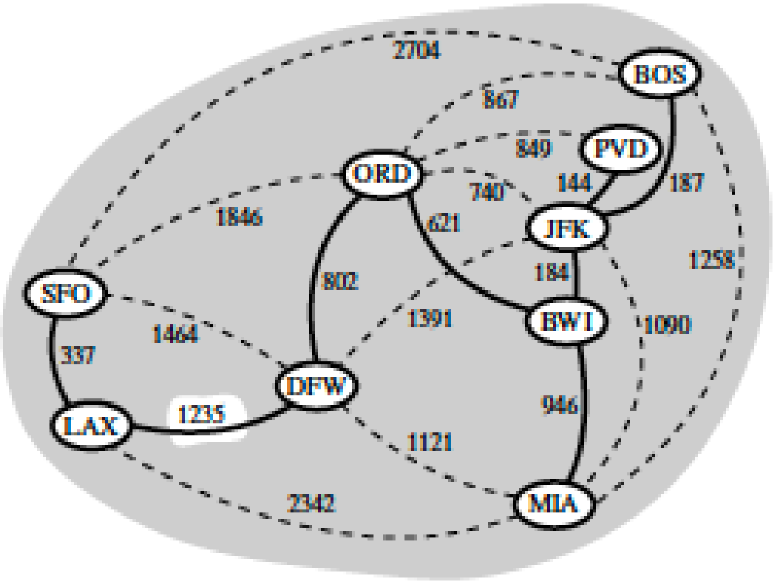

BOS
PVD
ORD
JFK
SFO
BWI
DFW
LAX
MIA
2704
867
849
740
144
187
1846
621
1258
184
802
1391
1090
1464
337
946
1235
1121
2342

(k)


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

(l)


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


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