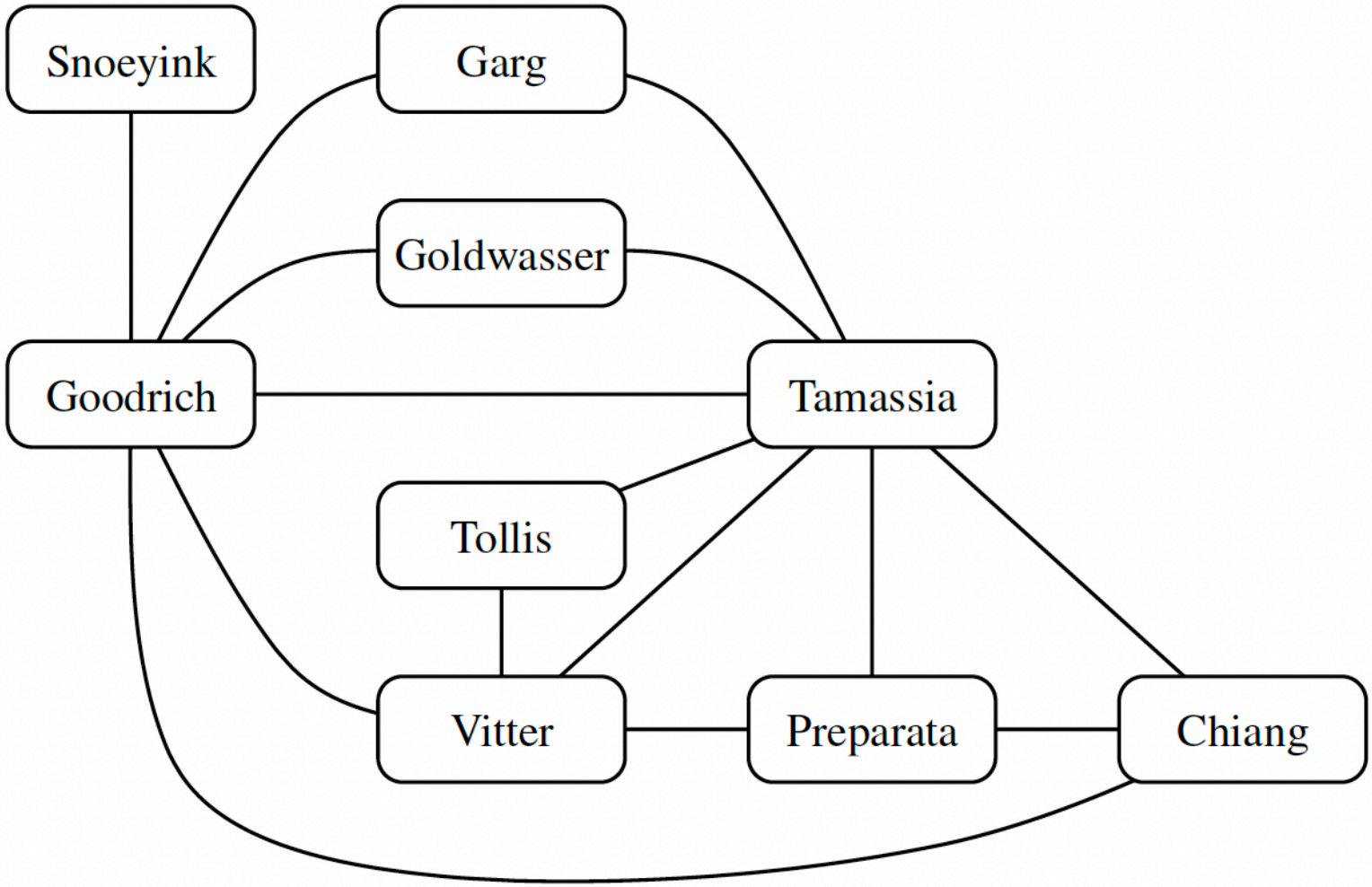
Graph Algorithms

Applied Algorithms

Graph

- A graph is a structure to represent relationships that exist between pairs of objects.
- A graph consists of vertices and edges (directed or undirected edges).
- The vertices are visualized as ovals/rectangles and the edges are as segments/curves connecting pairs of ovals/ rectangles.
- Some examples (domains): Transportation, computer networks, and electrical engineering.
- A graph is not a bar chart or function plot.

Example: An Undirected Graph

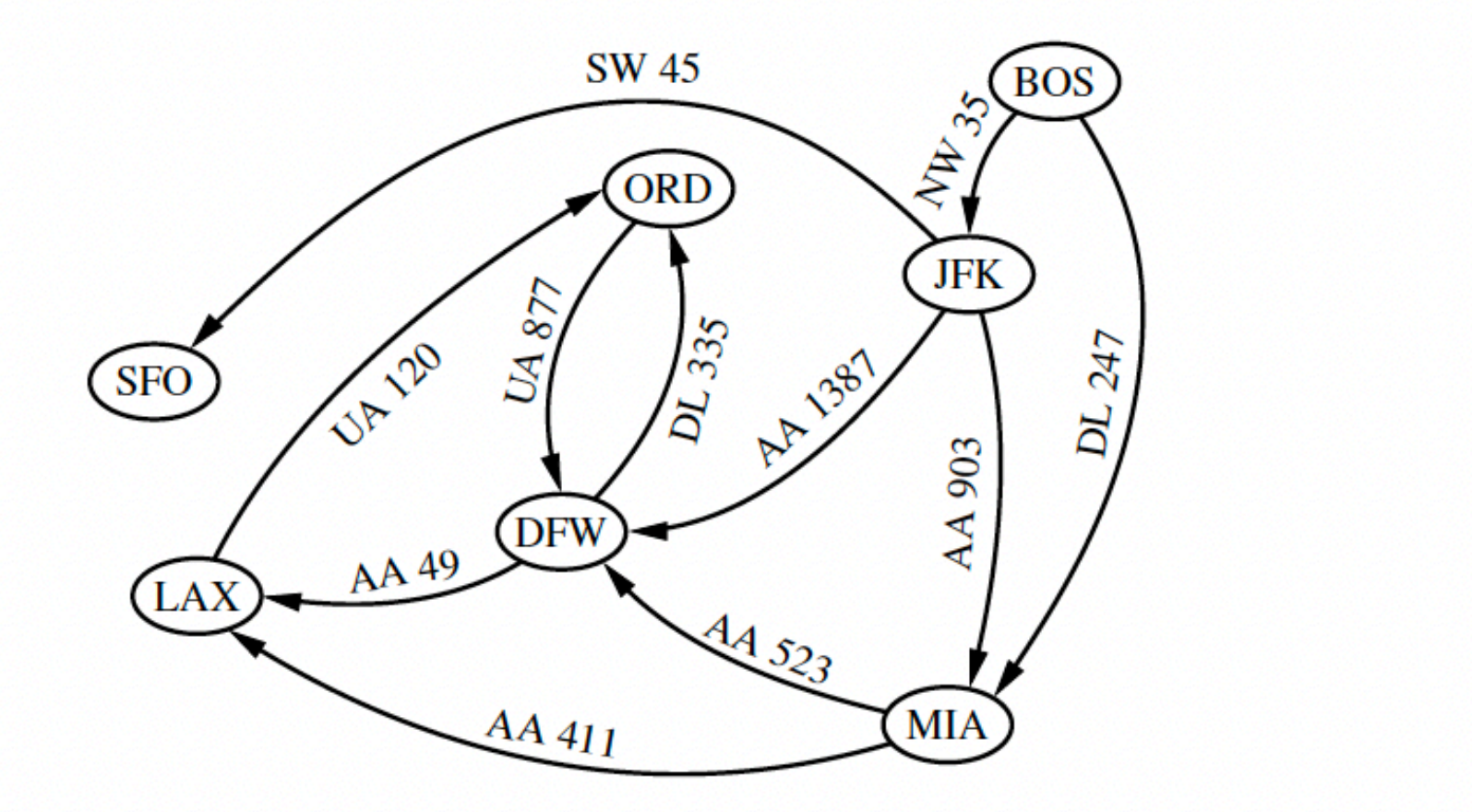


Co-authorship among some authors.

Graph

- Undirected graph: all edges are undirected.
- Directed graph (Digraph): all edges are directed
- Mixed graph: All edges consist of both directed and undirected edges.
- Two vertices u and v are **adjacent** if there is an edge whose end vertices are u and v.
- The **degree** of a vertex: The number of incident edges of v.
- The **in-degree** and **out-degree** of a vertex: The number of incoming and outgoing edges, respectively.

Example: A Directed Graph



A flight network.

Graph

- The group of edges is a **collection,** not a set.
- **Parallel edges (multiple edges):** two undirected edges which have the same end vertices, or two directed edges which have the same origin and the same destination.
- **Self-loop:** An edge is a self-loop if its two endpoints coincide.
- **Simple graph:** no parallel edges or self-loops.
- **Cyle:** a path that starts and ends at the same vertex, and that includes at least one edge.
- A path is **simple** if each vertex in the path is distinct, and a **cycle** is simple if each vertex in the cycle is distinct, except for the first and last one.

Graph

- A directed graph is **acyclic** if it has no directed cycles.
- Given vertices u and v of a (directed) graph G, we say that u **reaches** v, and that v is **reachable** from u, if G has a (directed) path from u to v.
- A graph is **connected** if, for any two vertices, there is a path between them.
- A directed graph G is **strongly connected** if for any two vertices u and v of G, u reaches v and v reaches u.

Graph

- A **subgraph** of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G, respectively.
- **A spanning** subgraph of G is a subgraph of G that contains all the vertices of the graph G.
- **A forest** is a graph without cycles.
- A **tree i**s a connected forest—a connected graph without cycles.
- A **spanning tree** of a graph is a spanning subgraph that is a tree

Data Structures for Graphs

- Edge list:
- Adjacency list:
- Adjacency map
- Adjacency matrix

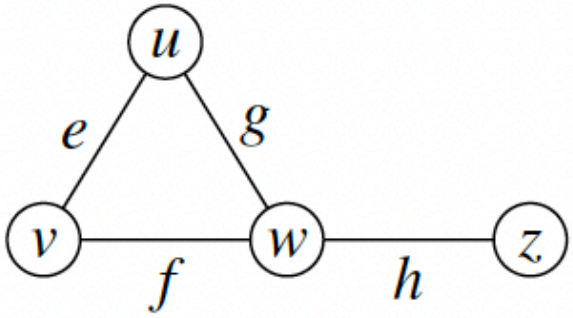
Data Structures for Graphs

Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

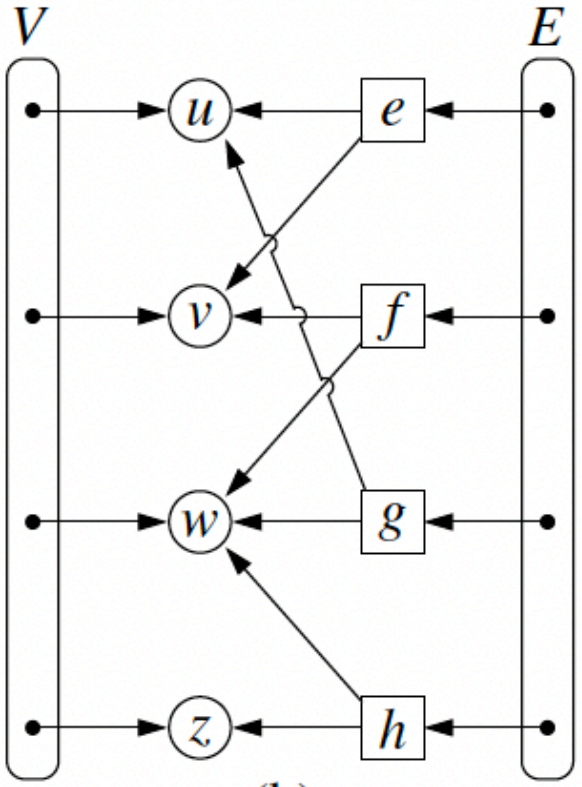
n: the number of vertices, m: he number of edges, and d_v: the degree of vertex v. The adjacency matrix uses O(n^2)space, while all other structures use O(n+m) space.

Edge List Structure

- V is an unordered list and includes all vertex objects and E is another unordered list which stores all edges.



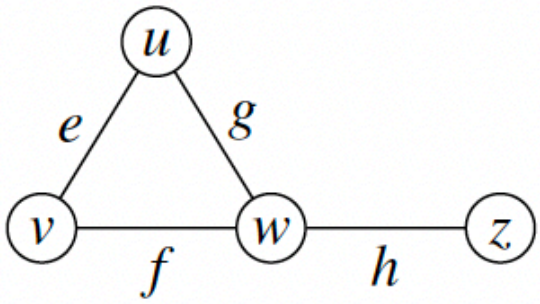
(a)



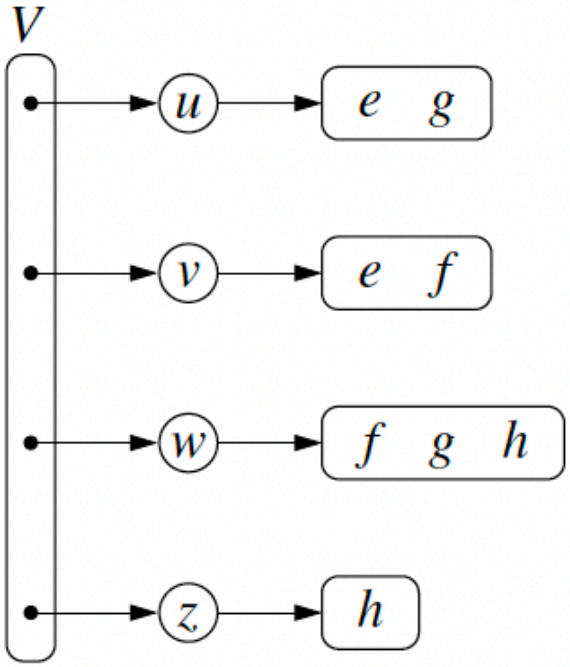
(b)

Adjacency List Structure

- The adjacency list structure stores edges in smaller, secondary containers which are associated with each individual vertex.



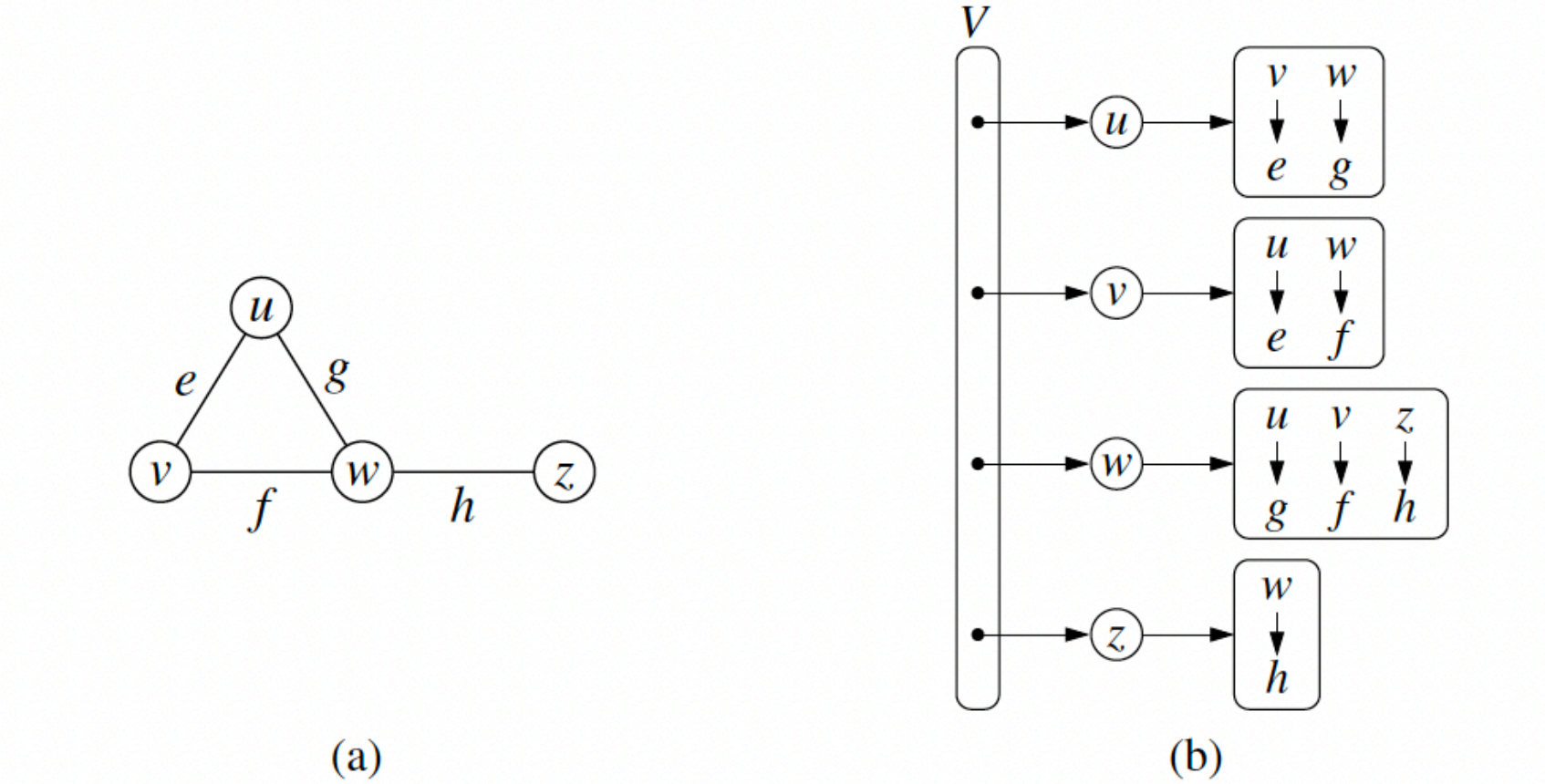
(a)



(b)

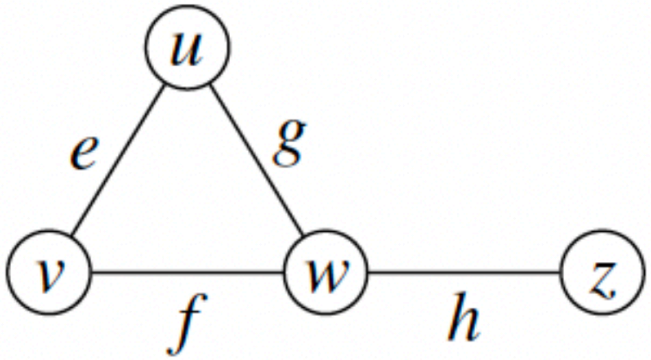
Edge Map Structure

- The secondary incidence collections are implemented as unordered linked lists



Edge Matrix Structure

- Allows us to locate an edge between a given pair of vertices in worst-case constant time.



(a)

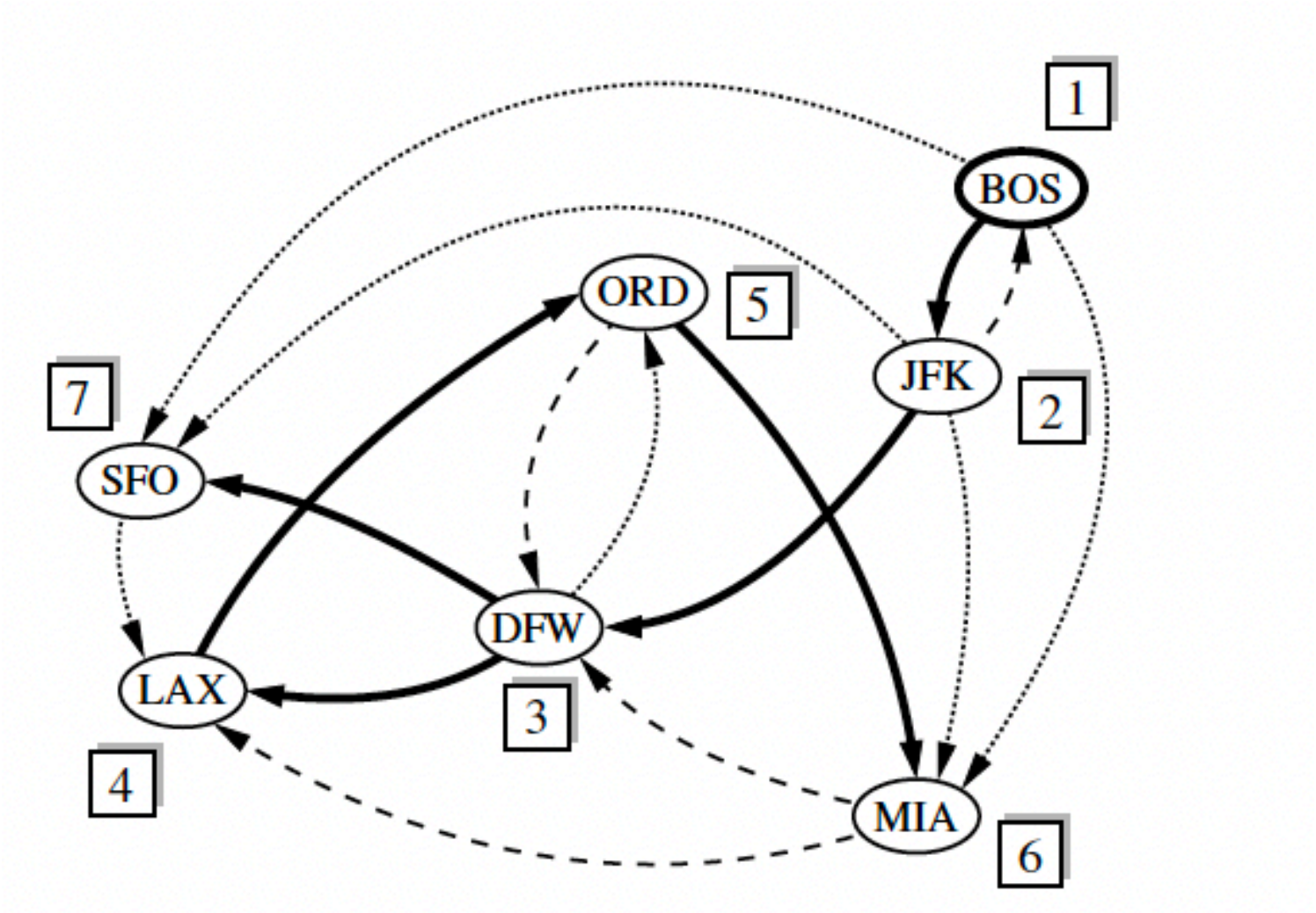
		0	1	2	3
u	0		e	g	
v	1	e		f	
w	2	g	f		h
z	3			h	

(b)

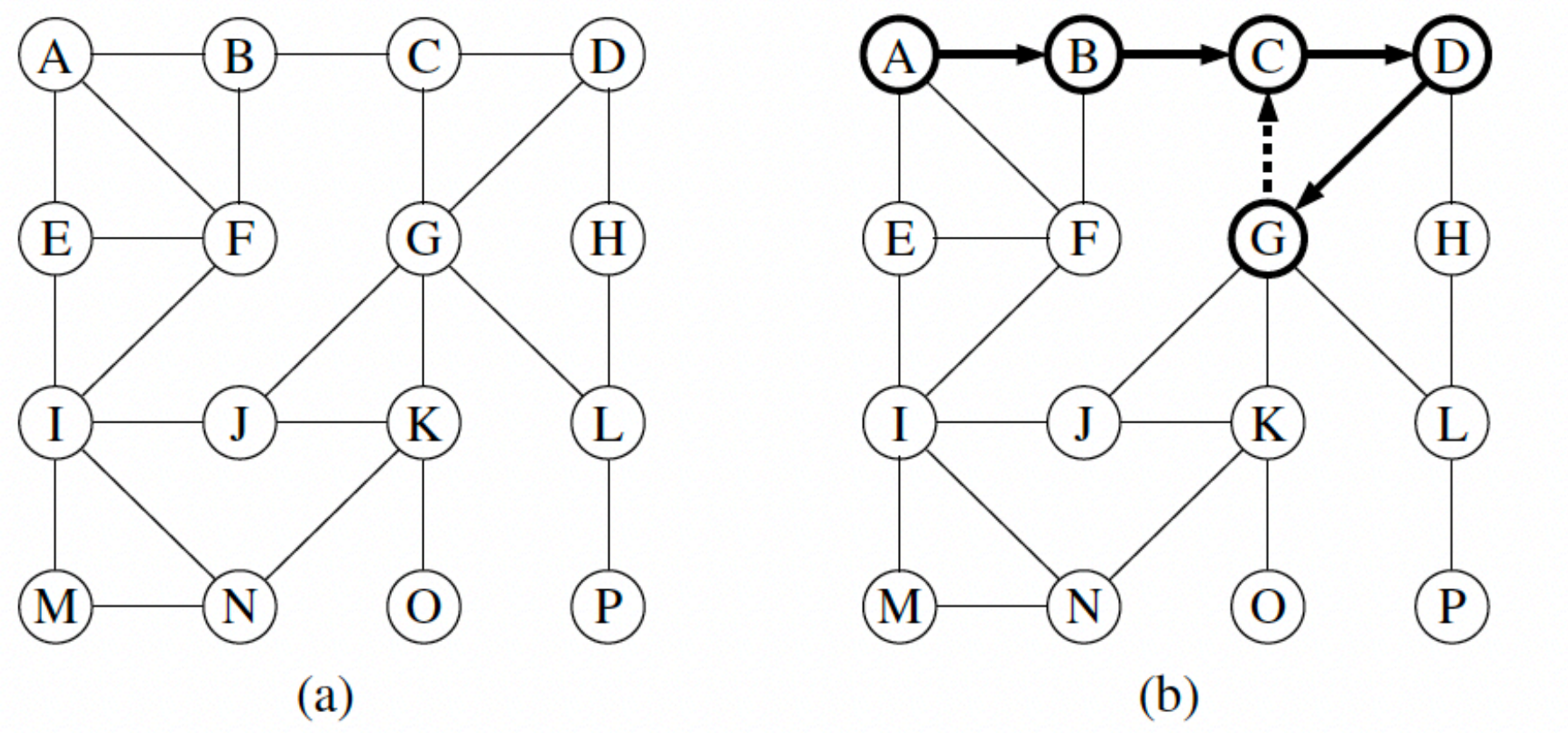
Graph Traversals

- A systematic procedure for exploring a graph by examining all of its vertices and edges.
- A traversal is efficient if it visits all the vertices and edges in time proportional to their number (linear time).
- Depth-first search (DFS) and breadth-first search (BFS).
- Video: https://www.youtube.com/watch?v=62IcXF_OF3k

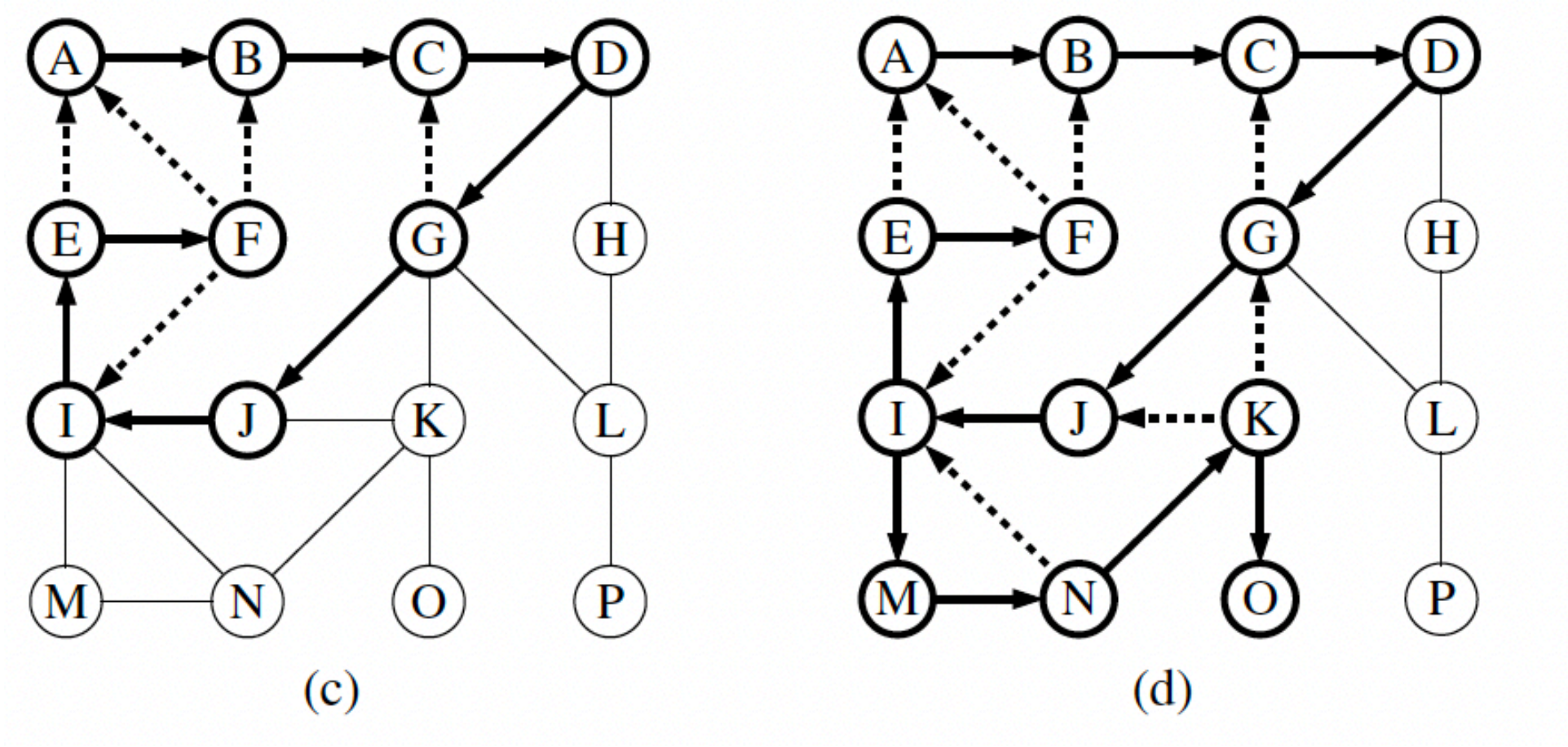
Example: Depth-First Search (DFS)



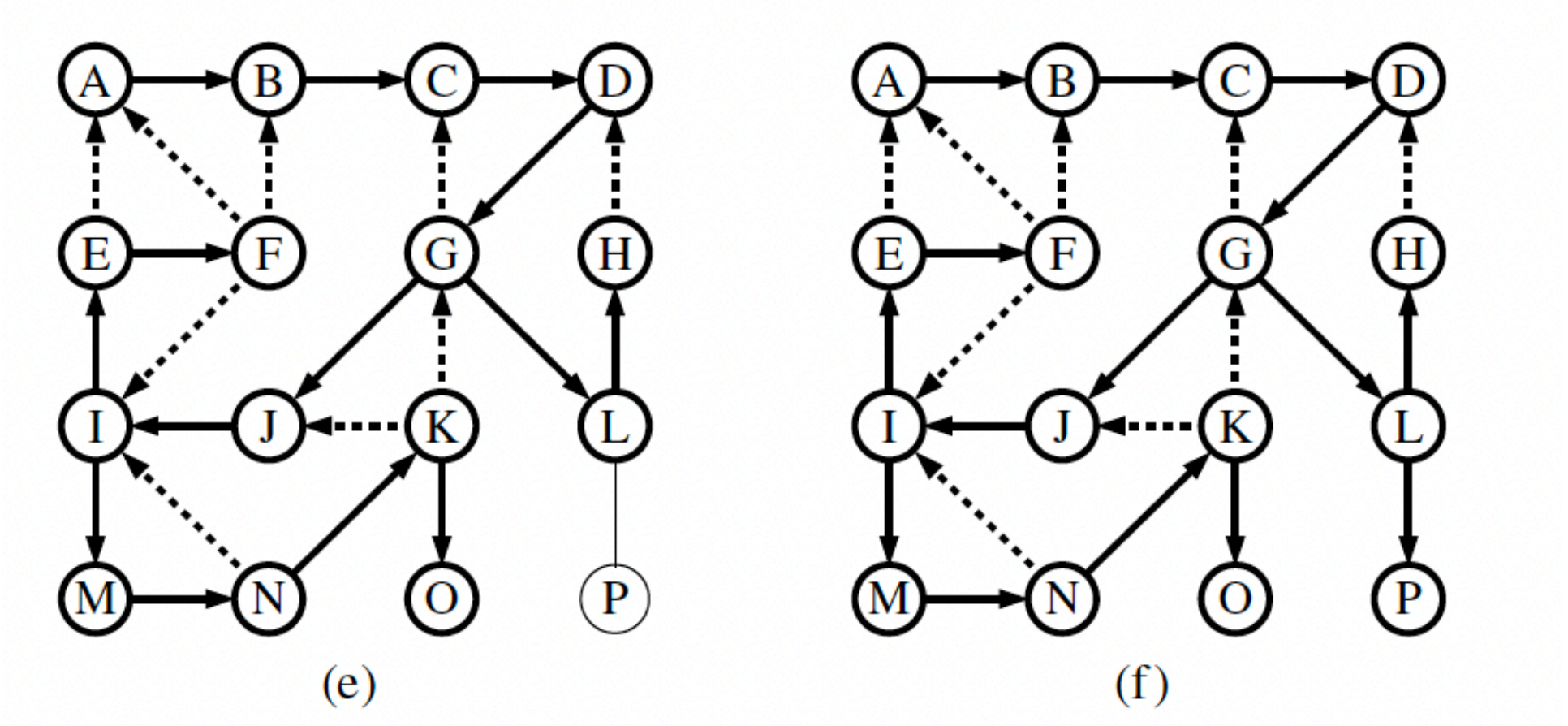
Example: Depth-First Search (DFS)



Example: Depth-First Search (DFS)



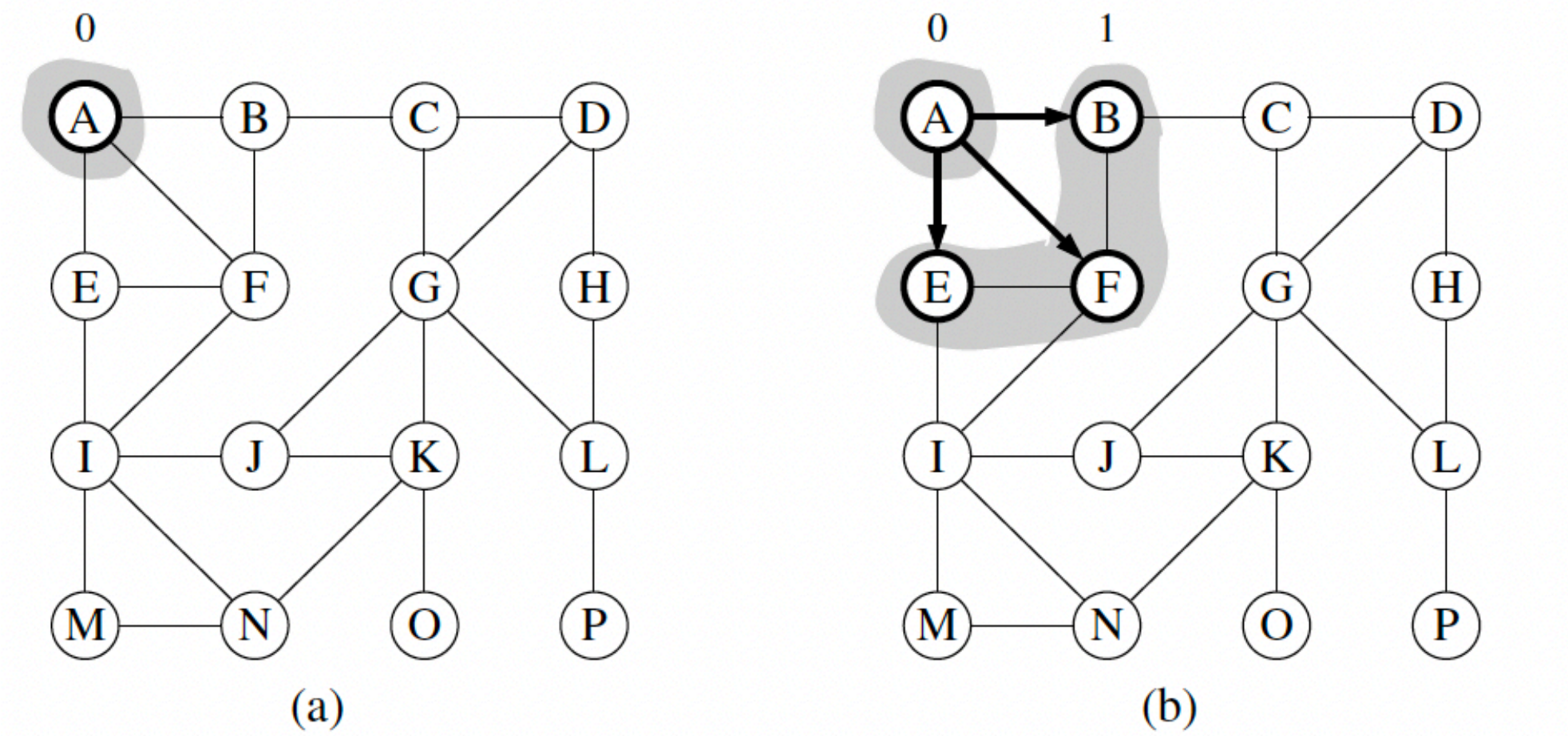
Example: Depth-First Search (DFS)



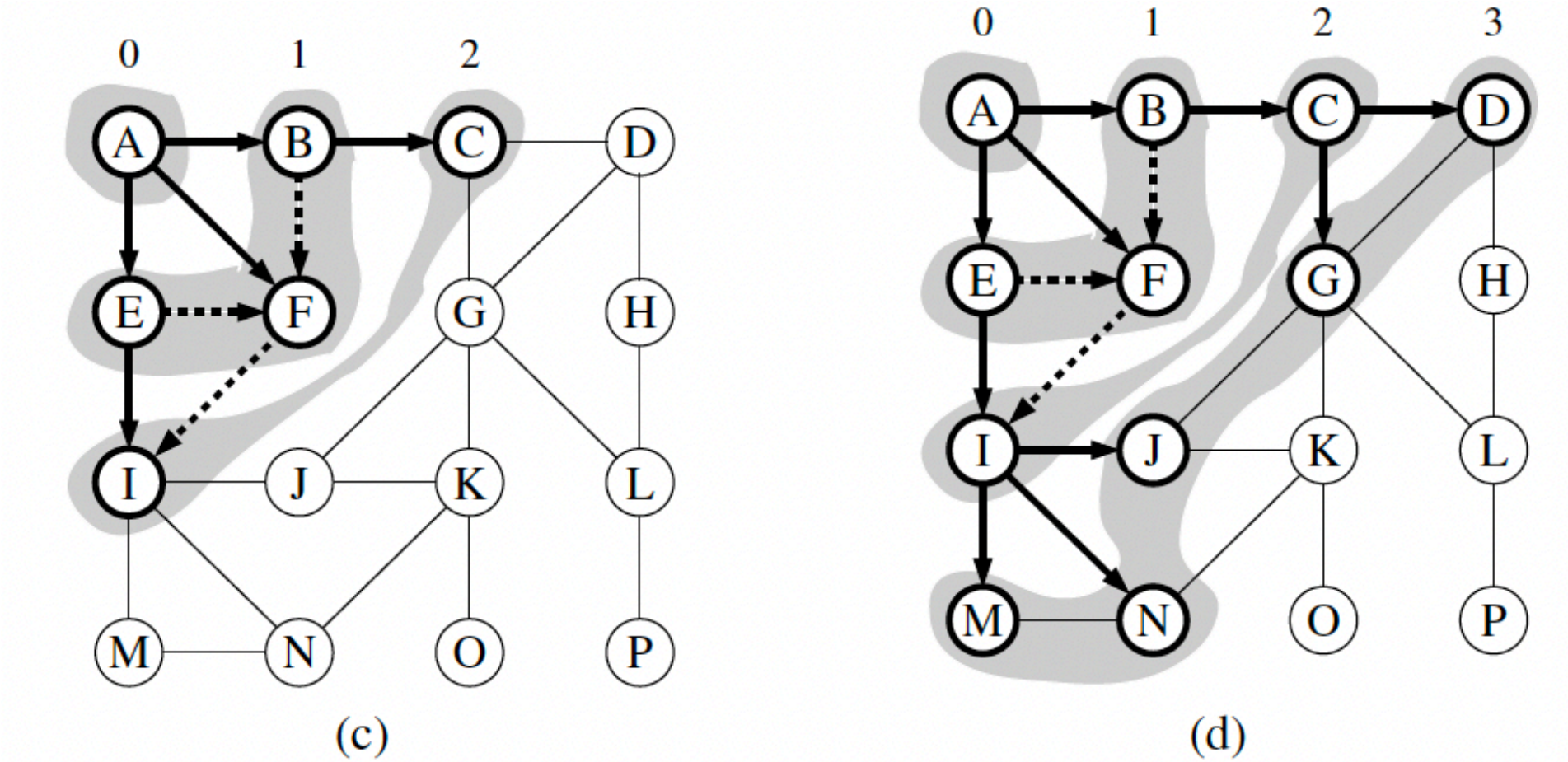
Depth-First Search (DFS)

```
1  def DFS(g, u, discovered):
2    """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.
3
4    discovered is a dictionary mapping each vertex to the edge that was used to
5    discover it during the DFS. (u should be "discovered" prior to the call.)
6    Newly discovered vertices will be added to the dictionary as a result.
7    """
8    for e in g.incident_edges(u):          # for every outgoing edge from u
9      v = e.opposite(u)
10     if v not in discovered:              # v is an unvisited vertex
11       discovered[v] = e                  # e is the tree edge that discovered v
12       DFS(g, v, discovered)              # recursively explore from v
```

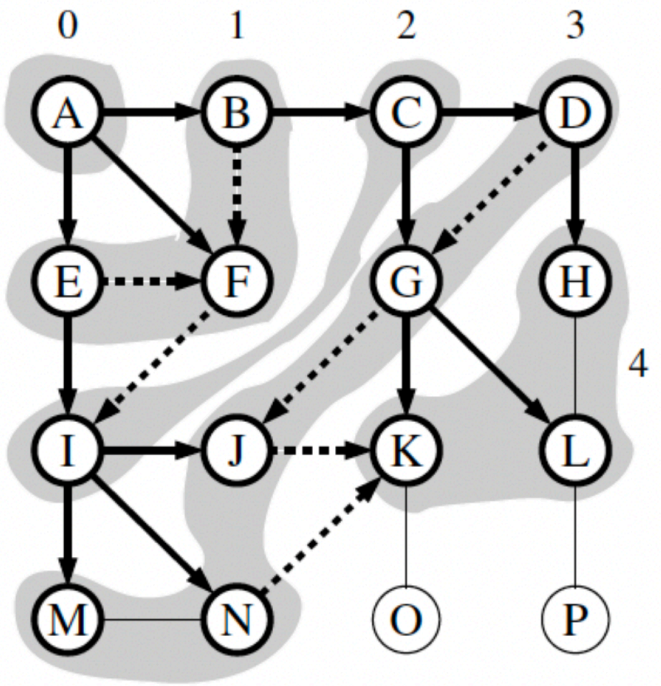
Breadth-First Search(BFS)



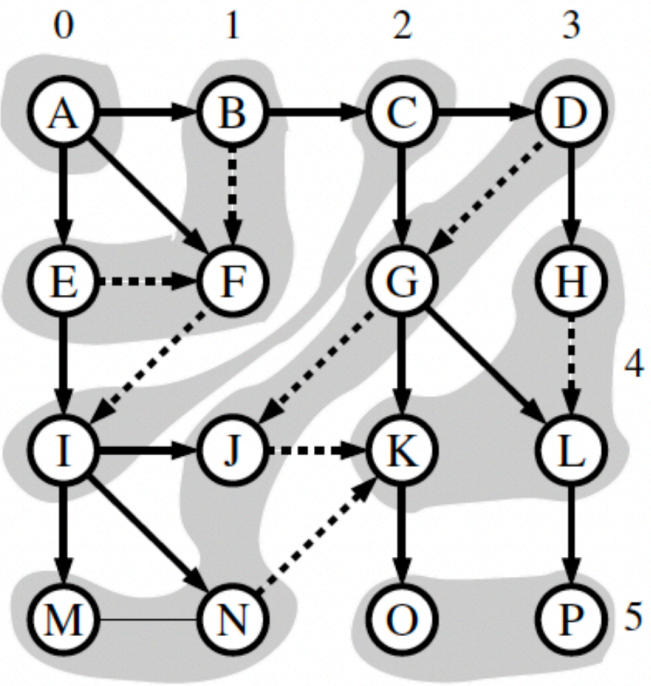
Breadth-First Search(BFS)



Breadth-First Search(BFS)



(e)

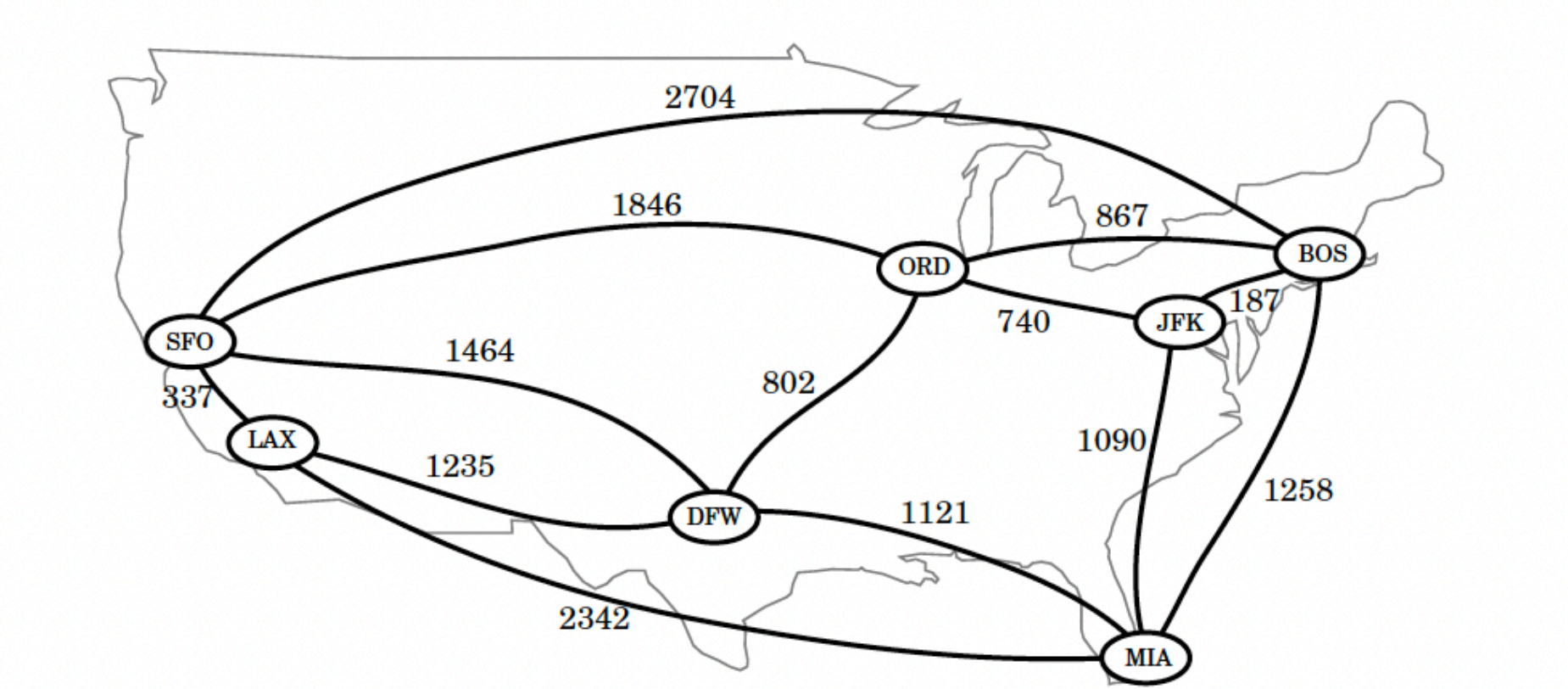


(f)

Breadth-First Search(BFS)

```
1  def BFS(g, s, discovered):
2    """Perform BFS of the undiscovered portion of Graph g starting at Vertex s.
3
4    discovered is a dictionary mapping each vertex to the edge that was used to
5    discover it during the BFS (s should be mapped to None prior to the call).
6    Newly discovered vertices will be added to the dictionary as a result.
7    """
8    level = [s]                          # first level includes only s
9    while len(level) > 0:
10     next_level = [ ]                   # prepare to gather newly found vertices
11     for u in level:
12       for e in g.incident_edges(u):    # for every outgoing edge from u
13         v = e.opposite(u)
14         if v not in discovered:        # v is an unvisited vertex
15           discovered[v] = e            # e is the tree edge that discovered v
16           next_level.append(v)         # v will be further considered in next pass
17     level = next_level                 # relabel 'next' level to become current
```

Weighted Graphs



- A weighted graph: Vertices are U.S. airports and edges are distances in miles.

Shortest Paths

- BFS can be used to find a shortest path from some starting vertex to every other vertex in a connected graph when each edge is as good as any other.
- When edges are weighted, this approach is not appropriate. For example, finding the fastest way to travel cross-country.
- One pair, single source, all pairs.

Defining Shortest Paths in a Weighted Graph

Let G be a weighted graph. The ***length*** (or weight) of a path is the sum of the weights of the edges of P. That is, if $P = ((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$, then the length of P, denoted $w(P)$, is defined as

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

The ***distance*** from a vertex u to a vertex v in G, denoted $d(u, v)$, is the length of a minimum-length path (also called ***shortest path***) from u to v, if such a path exists.

Dijkstra's Algorithm

Algorithm ShortestPath(G, s):

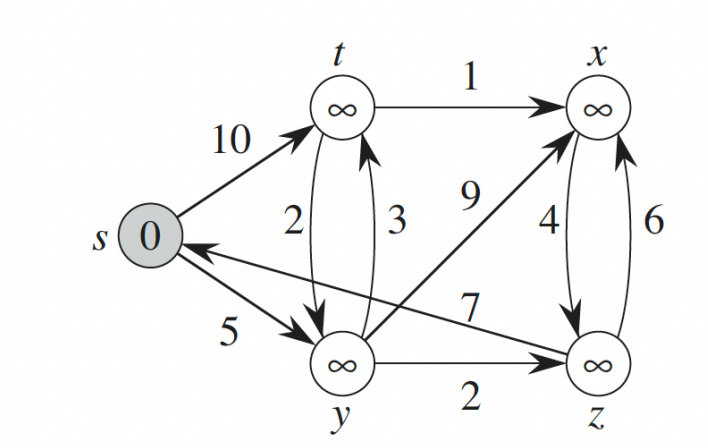
Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

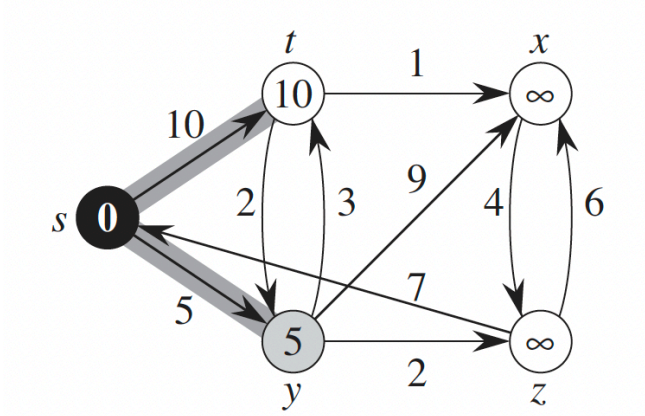
Output: The length of a shortest path from s to v for each vertex v of G.

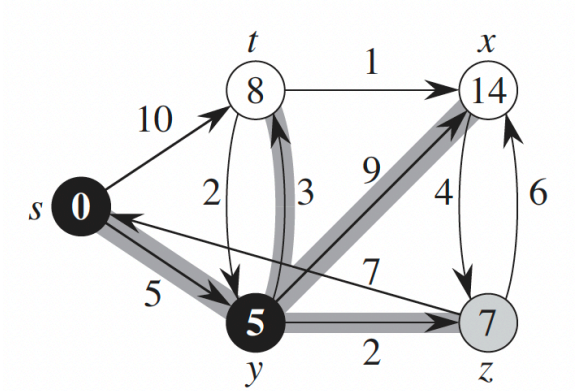
Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  u = value returned by Q.remove_min()
  for each vertex v adjacent to u such that 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

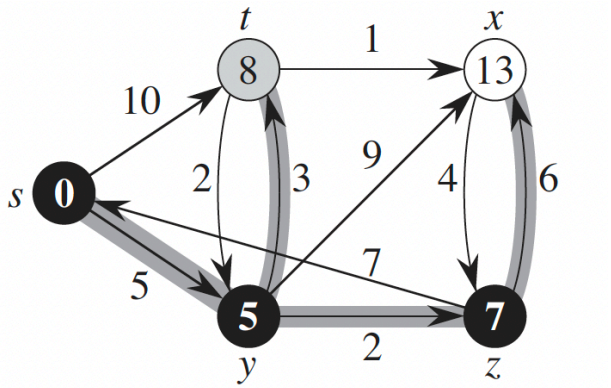
- Video: https://www.youtube.com/watch?v=pVfj6mxhdMw

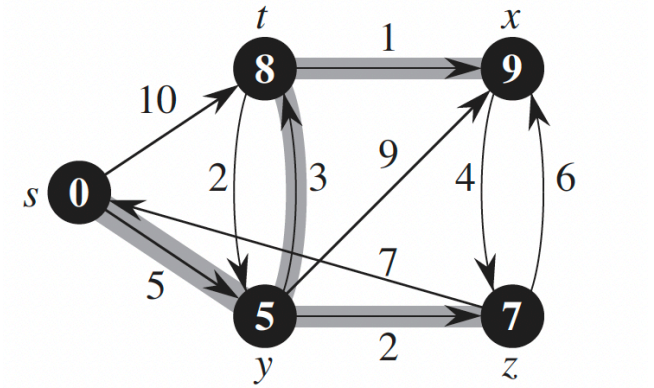
Class Exercise: Dijkstra's Algorithm

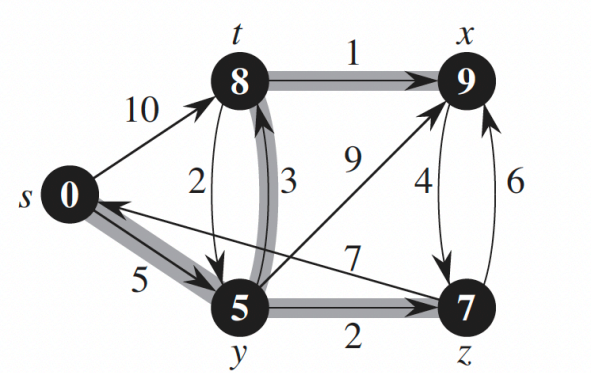






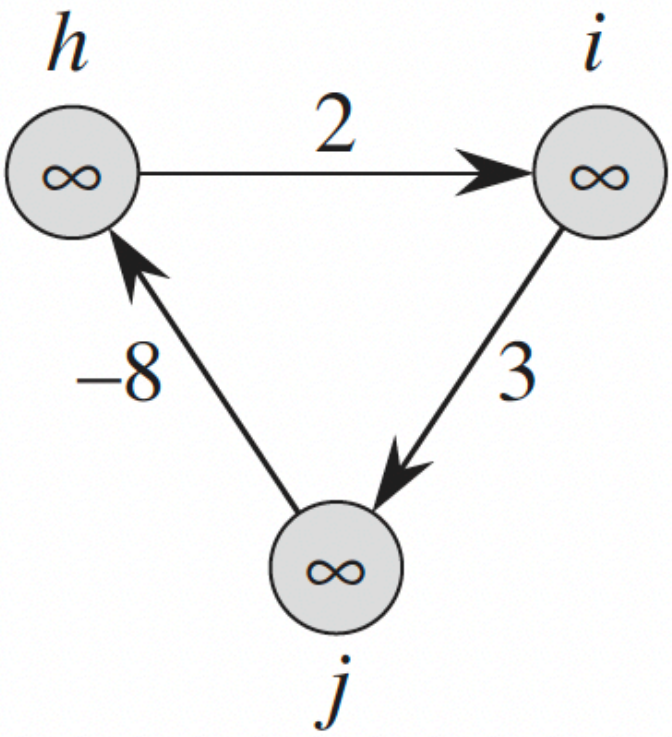






The Bellman-Ford Algorithm

- Negative-weight cycle and relaxation.



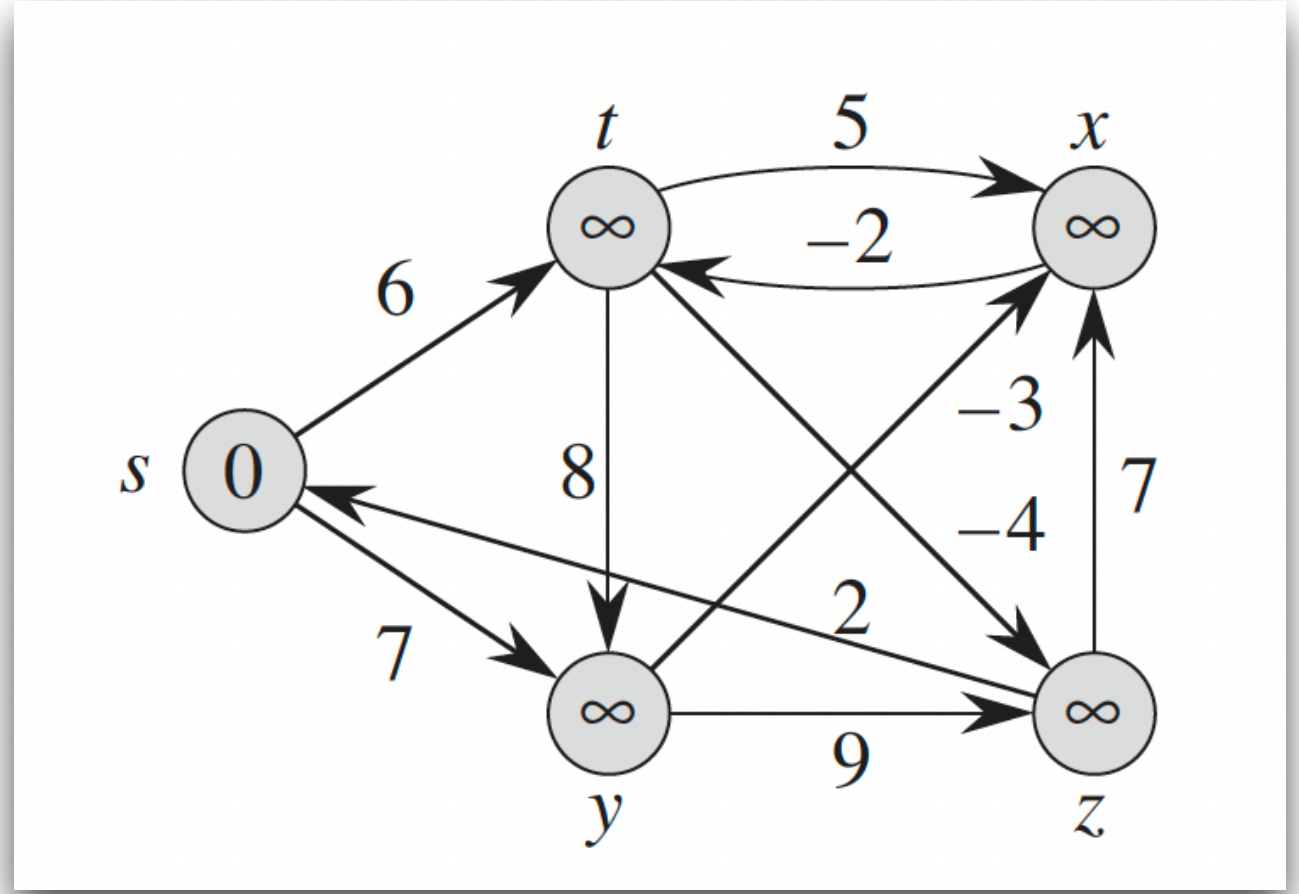
The Bellman-Ford algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

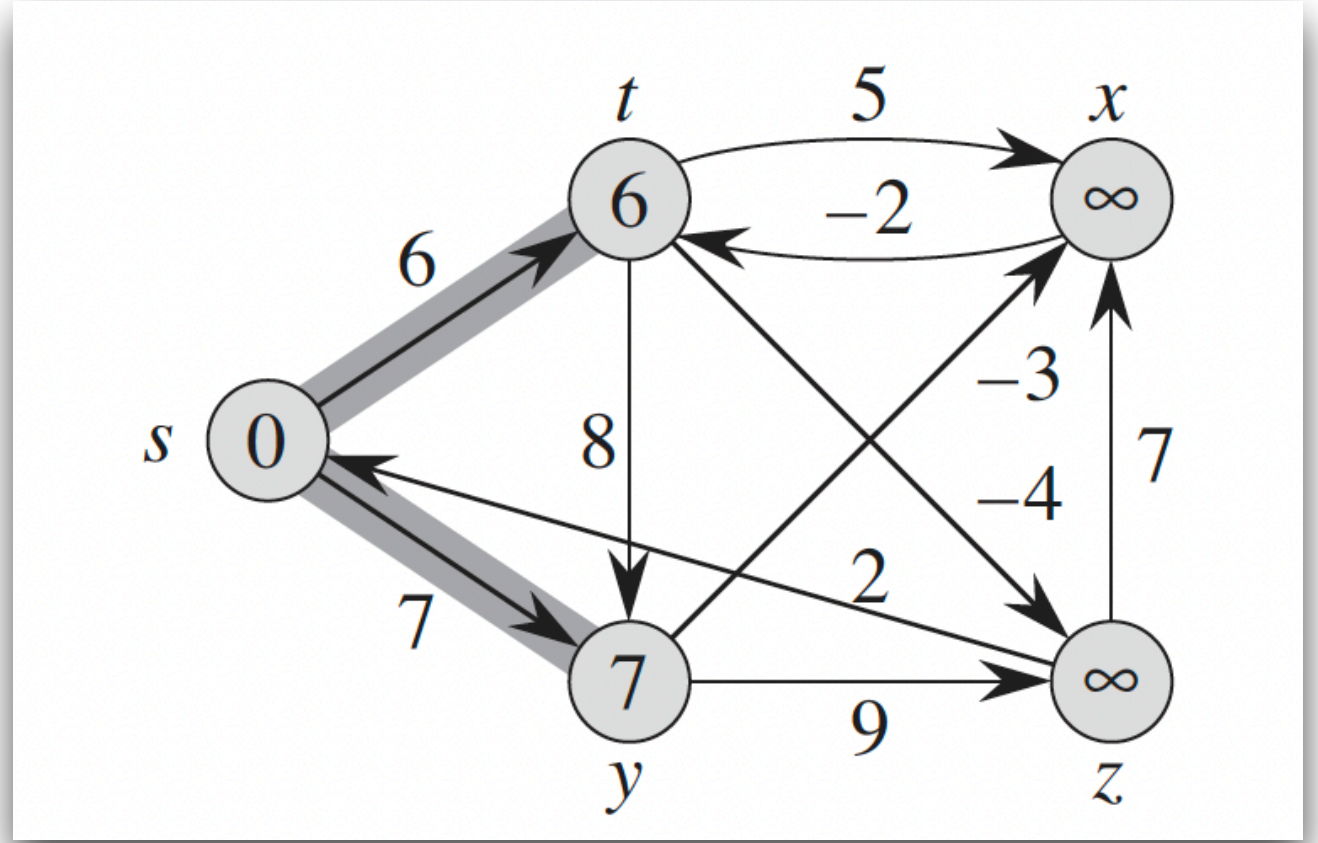
The Bellman-Ford Algorithm: Run-time

- The Bellman-Ford algorithm runs in time O(EV)
- The initialization takesO(V) time, each of the |V-1| passes over the edges $\Theta(E)$ time, and the for loop of takes O(E) time.
- https://www.youtube.com/watch?v=obWXjtg0L64

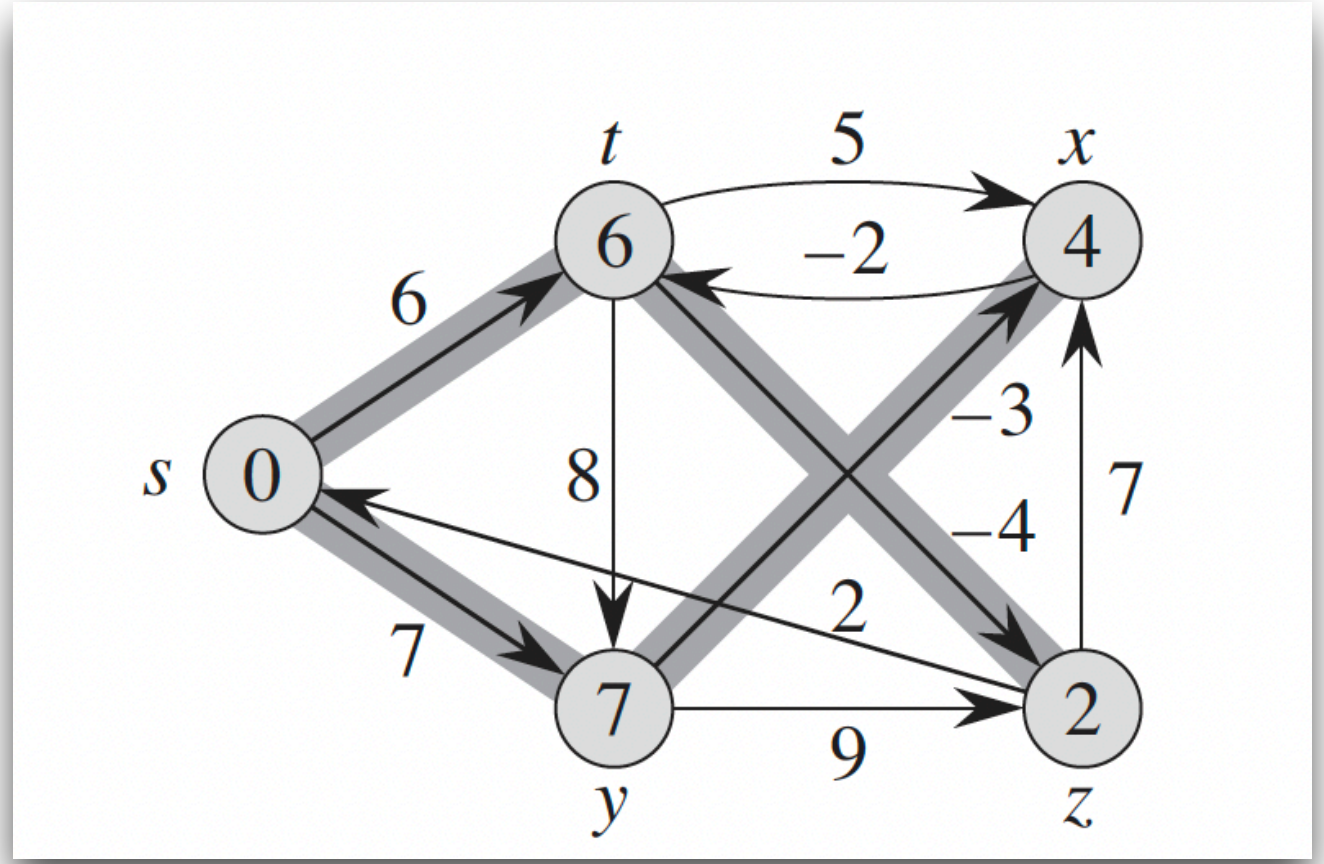
Class Exercise:
The Bellman-Ford algorithm



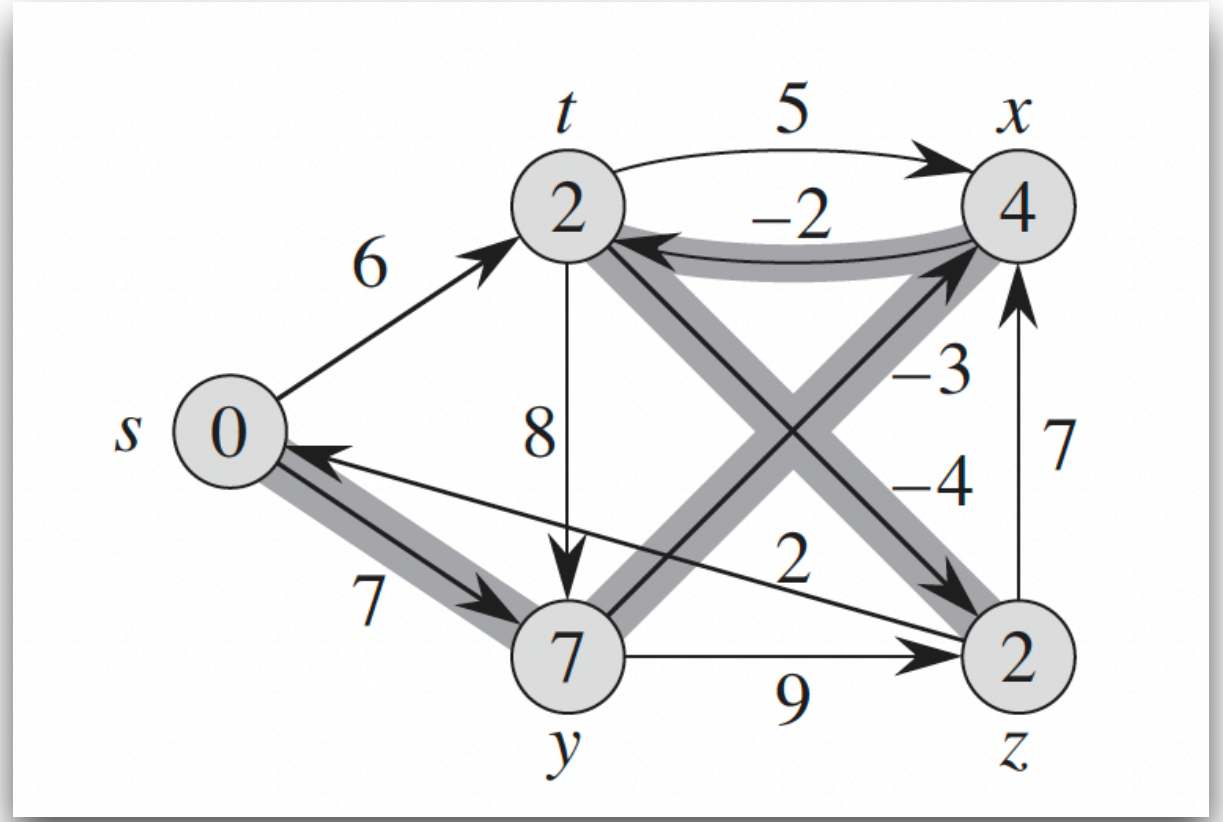
Class Exercise:
The Bellman-Ford algorithm



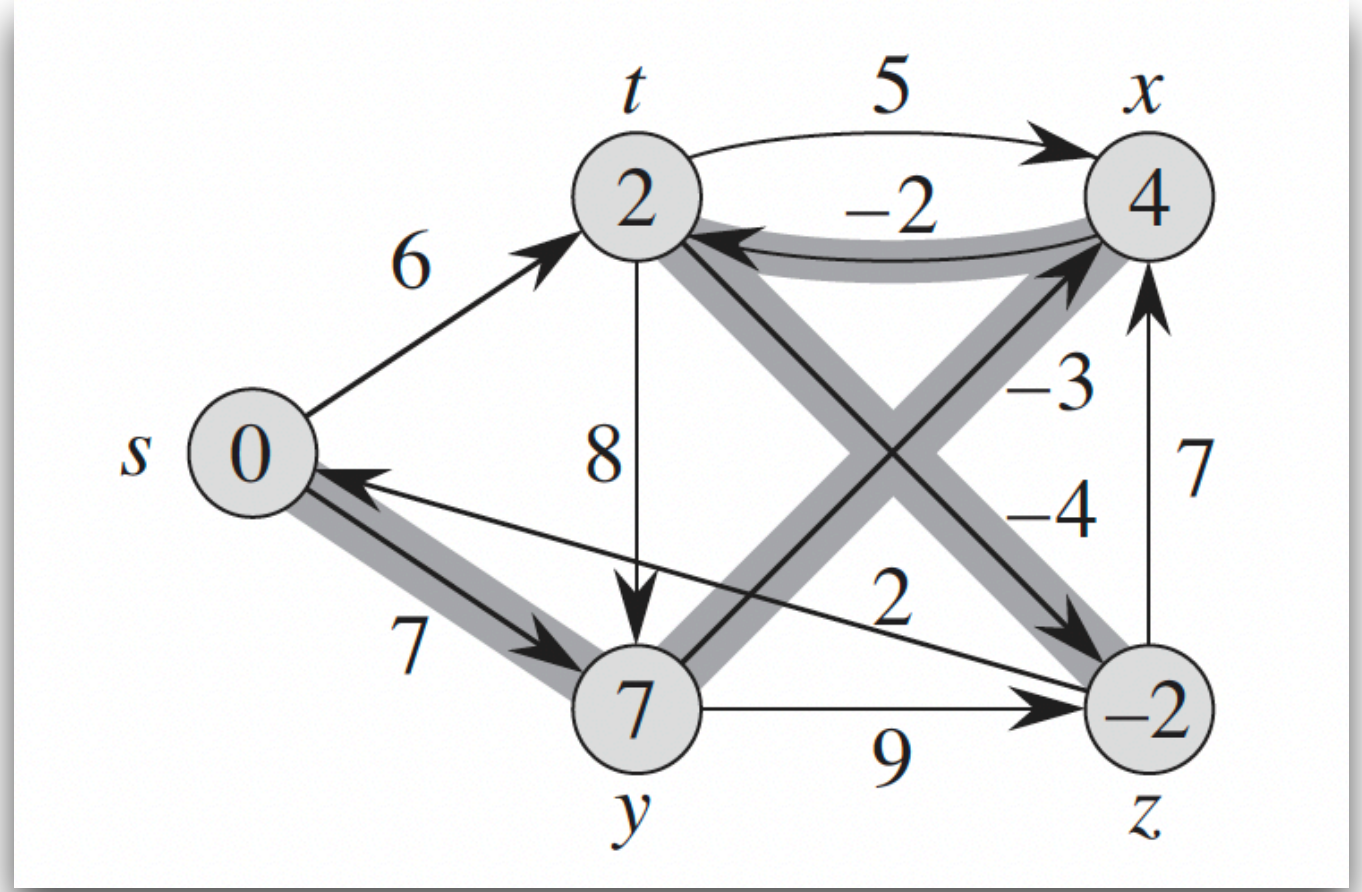
Class Exercise:
The Bellman-Ford algorithm



Class Exercise:
The Bellman-Ford algorithm



Class Exercise:
The Bellman-Ford algorithm



Minimum Spanning Trees

- Example: Connect all the computers in this building using the least amount of cable.
- Vertices are the computers, and edges represent all the possible pairs (u,v) of computers
- Weight w(u,v) of an edge: the amount of cable we need to connect computer u and v.
- This time, we are interested instead in finding a tree that contains all the vertices of G and has the minimum total weight over all such trees.

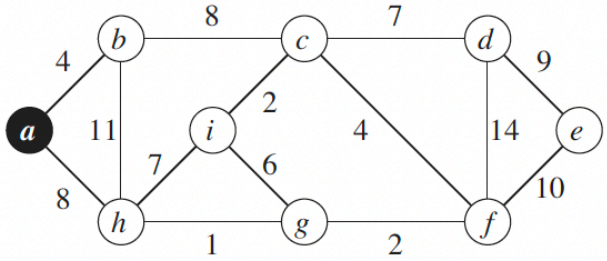
Minimum Spanning Trees

- Two classic algorithms for solving the MST problem: Prim's and Kruskal's MST algorithms. Both of them are greedy.
- Prim's algorithm: Grow the MST from a single root vertex. (similar to Dijkstra's shortest-path algorithm).
- Kruskal's algorithm: Grow the MST in clusters by considering edges in nondecreasing order of their weights.
- Both run $O(E \times \lg V)$ in time using ordinary binary heaps.
- Prim's algorithm runs $O(E + V \times \lg V)$ in time using Fibonacci heaps (if $|V|$ is much smaller than $|E|$, this is more efficient).

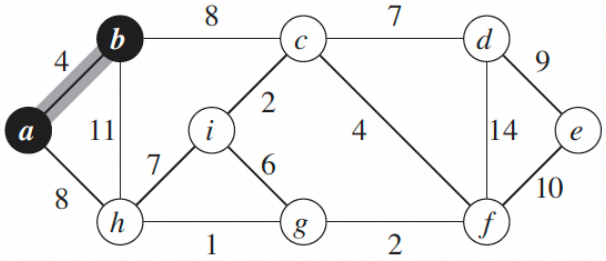
Prim-Jarnik Algorithm

- Grow a MST from a single cluster starting from some vertex.
- In each iteration, we choose a minimum-weight edge e = (u,v), connecting a vertex u in the cloud C to a vertex v outside of C. The vertex v is then brought into the cloud C and the process is repeated until a spanning tree is formed.
- Video: https://www.youtube.com/watch?v=Uj47dxYPow8

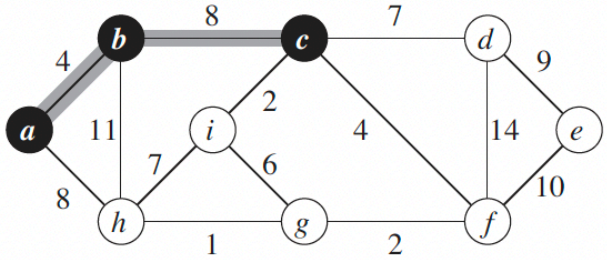
(a)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

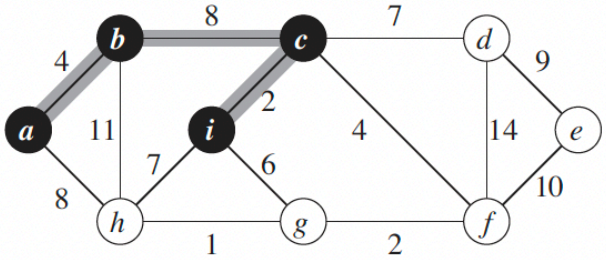
(b)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

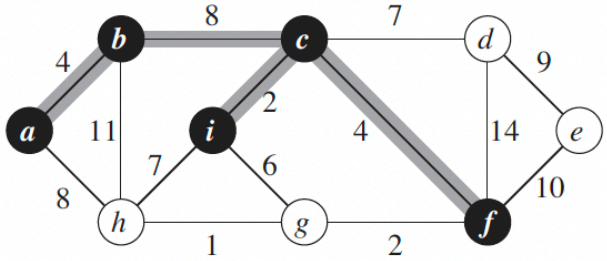
(c)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

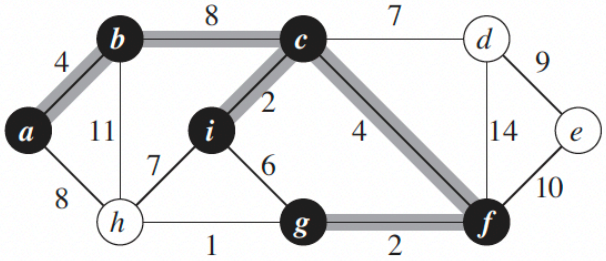
(d)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

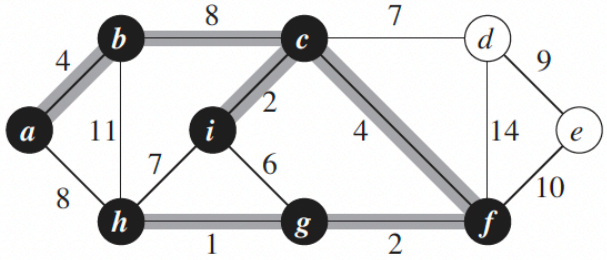
(e)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

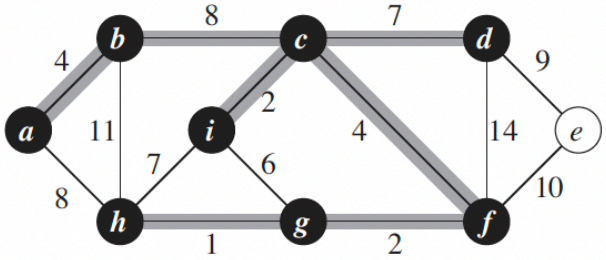
(f)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

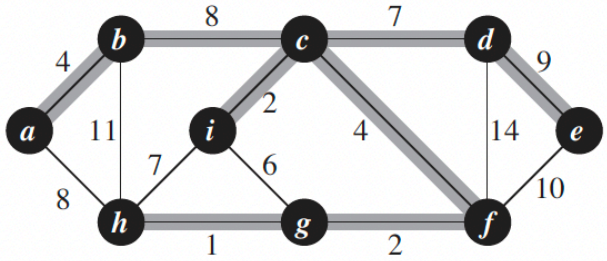
(g)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

(h)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

(i)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

Prim-Jarnik Algorithm

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').

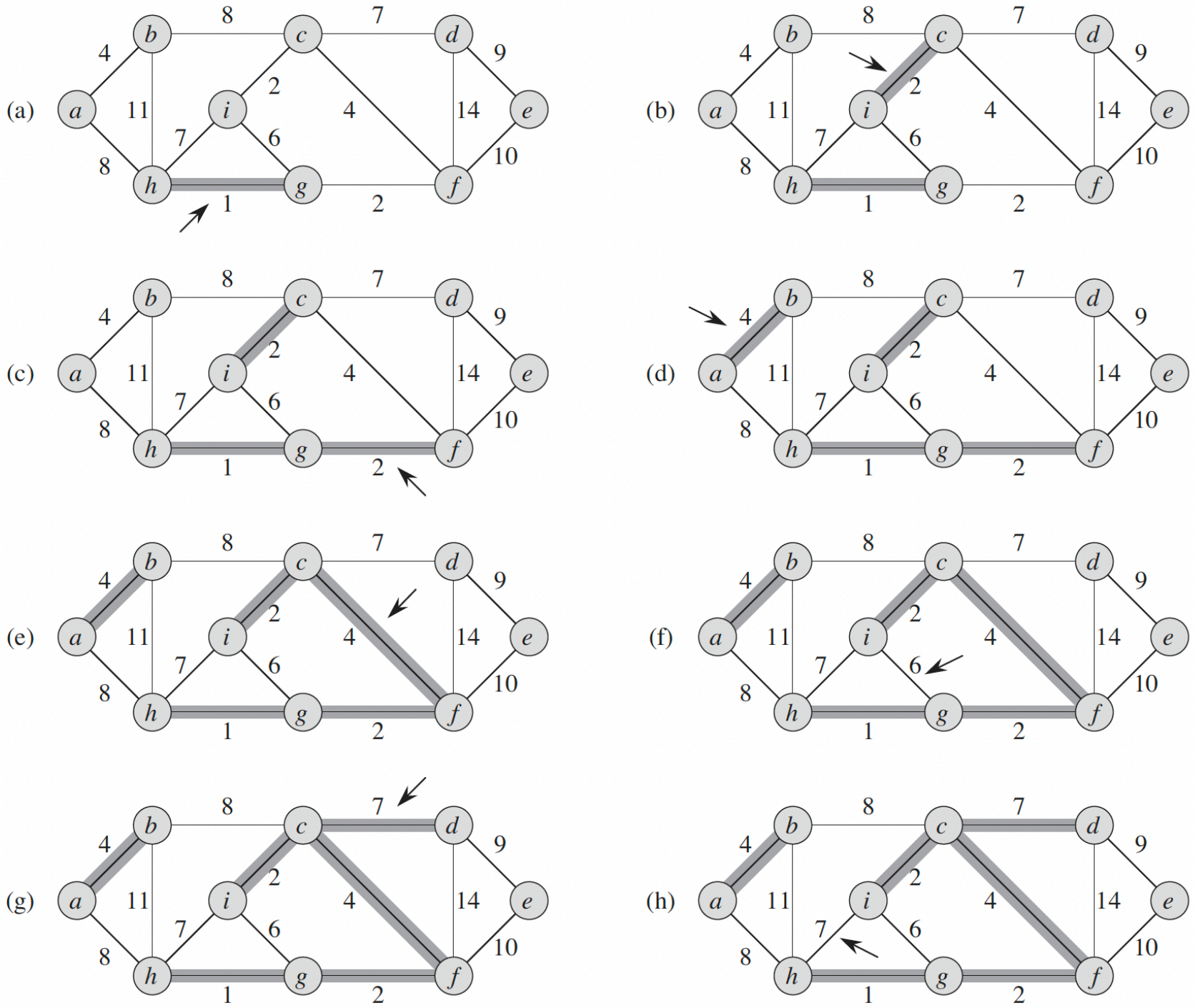
return the tree T

```
1  def MST_PrimJarnik(g):
2    """Compute a minimum spanning tree of weighted graph g.
3
4    Return a list of edges that comprise the MST (in arbitrary order).
5    """
6    d = { }                                  # d[v] is bound on distance to tree
7    tree = [ ]                               # list of edges in spanning tree
8    pq = AdaptableHeapPriorityQueue( )       # d[v] maps to value (v, e=(u,v))
9    pqlocator = { }                          # map from vertex to its pq locator
10
11   # for each vertex v of the graph, add an entry to the priority queue, with
12   # the source having distance 0 and all others having infinite distance
13   for v in g.vertices( ):
14     if len(d) == 0:                                  # this is the first node
15       d[v] = 0                                       # make it the root
16     else:
17       d[v] = float('inf')                            # positive infinity
18     pqlocator[v] = pq.add(d[v], (v,None))
19
20   while not pq.is_empty( ):
21     key,value = pq.remove_min( )
22     u,edge = value                                   # unpack tuple from pq
23     del pqlocator[u]                                 # u is no longer in pq
24     if edge is not None:
25       tree.append(edge)                              # add edge to tree
26     for link in g.incident_edges(u):
27       v = link.opposite(u)
28       if v in pqlocator:                             # thus v not yet in tree
29         # see if edge (u,v) better connects v to the growing tree
30         wgt = link.element( )
31         if wgt < d[v]:                               # better edge to v?
32           d[v] = wgt                                 # update the distance
33           pq.update(pqlocator[v], d[v], (v, link))   # update the pq entry
34   return tree
```

Kruskal's algorithm

- This algorithm maintains a forest of clusters and iteratively merges pairs of clusters until a single cluster spans the graph.
- Each vertex is a cluster in the first iteration.
- The algorithm first orders edges by increasing weight.
- When an edge e connects two different clusters, then e is added to the set of edges of the minimum spanning tree, and the two clusters connected by e are merged into a single cluster.
- If e connects two vertices that are already in the same cluster, then e is discarded.
- Once the algorithm has added enough edges to form a spanning tree, it terminates. The output is a minimum spanning tree.

Kruskal's algorithm



Kruskal's algorithm

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 algorithm

```
1  def MST_Kruskal(g):
2    """Compute a minimum spanning tree of a graph using Kruskal's algorithm.
3
4    Return a list of edges that comprise the MST.
5
6    The elements of the graph's edges are assumed to be weights.
7    """
8    tree = [ ]                          # list of edges in spanning tree
9    pq = HeapPriorityQueue( )           # entries are edges in G, with weights as key
10   forest = Partition( )               # keeps track of forest clusters
11   position = { }                      # map each node to its Partition entry
12
13   for v in g.vertices( ):
14     position[v] = forest.make_group(v)
15
16   for e in g.edges( ):
17     pq.add(e.element( ), e)           # edge's element is assumed to be its weight
18
19   size = g.vertex_count( )
20   while len(tree) != size - 1 and not pq.is_empty( ):
21     # tree not spanning and unprocessed edges remain
22     weight,edge = pq.remove_min( )
23     u,v = edge.endpoints( )
24     a = forest.find(position[u])
25     b = forest.find(position[v])
26     if a != b:
27       tree.append(edge)
28       forest.union(a,b)
29
30   return tree
```

References

1. Hetland, M. L. (2014). Python Algorithms: mastering basic algorithms in the Python Language. Apress.
2. Lee, K. D., Lee, K. D., & Steve Hubbard, S. H. (2015). *Data Structures and Algorithms with Python*. Springer.
3. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. Hoboken: Wiley
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.
5. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-wesley professional.