# CH13. GRAPH ALGORITHMS

CSED233 Data Structure
Prof. Hwanjo Yu
POSTECH

# Graphs

- A graph G is a set V of **vertices (or nodes)** and a collection E of pairs of vertices from V, called **edge (or arcs)**.
- Edges in a graph are either **directed** or **undirected**.
- Two vertices u and v are **adjacent** if there is an edge whose end vertices are u and v.
- An edge is **incident** on a vertex if the vertex is one of the edge's endpoints.
- The **degree** of a vertex v is the number of incident edges of v.
- The **in-degree** and **out-degree** of a vertex v are the number of the incoming and outgoing edges of v.
- A graph is **simple** if it does not have parallel edges or self-loops.
- If G is a graph with m edges, $\sum_{v\ in\ G} \deg(v) = 2m$
- If G is a directed graph with m edges, $\sum_{v\ in\ G} \mathrm{indeg}(v) = \sum_{v\ in\ G} \mathrm{outdeg}(v) = m$
- A simple graph with n vertices has at most $O(n^2)$ edges.
- A graph is **connected** if, for any two vertices, there is a path between them.
- A **spanning tree** of a graph is a connected graph without cycles that contains all the vertices of the graph.

# Graph ADT

- Vertex u
  - *operator*()*
  - *incidentEdges()*
  - *isAdjacentTo(v)*
- Edge e
  - *operator*()*
  - *endVertices()*
  - *opposite(v)*
  - *isAdjacentTo(f)*
  - *isIncidentOn(v)*
- Graph
  - *vertices()*
  - *edges()*
  - *insertVertex(x)*
  - *insertEdge(v,w,x)*
  - *eraseVertex(v)*
  - *eraseEdge(e)*

# Data Structures for Graphs: Edge List

- The Edge List Structure
- Space: O(n+m)
- O(1)
  - *endVertices()*
  - *opposite(v)*
  - *isIncidentOn(v)*
- O(m)
  - *incidentEdges()*
  - *isAdjacentTo(v)*

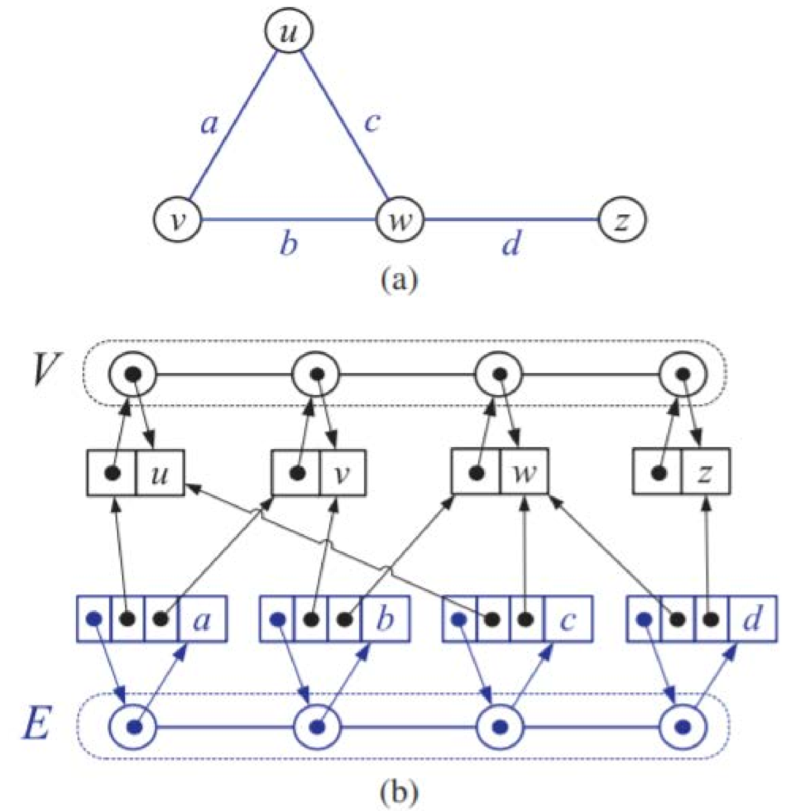


**Figure 13.3:** (a) A graph $G$. (b) Schematic representation of the edge list structure for $G$. We visualize the elements stored in the vertex and edge objects with the element names, instead of with actual references to the element objects.

# Data Structures for Graphs: Adjacency List

- The Adjacency List Structure
- Provide direct access from the edges to the vertices and from the vertices to their incident edges
- Space: O(n+m)
- O(1)
  - *endVertices()*
  - *opposite(v)*
  - *isIncidentOn(v)*
- O(deg(v))
  - ***v.incidentEdges()***
- O(min(deg(v),deg(w)))
  - ***v.isAdjacentTo(w)***

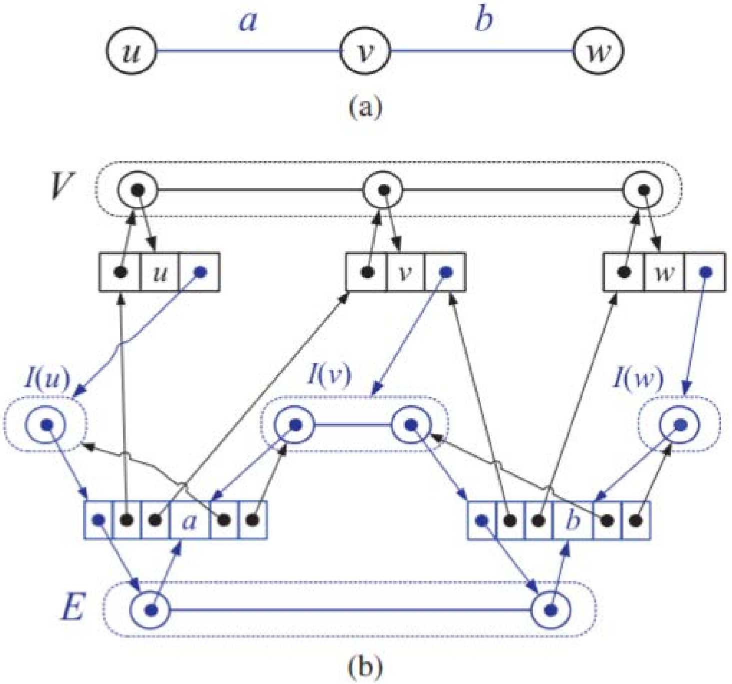


**Figure 13.4:** (a) A graph $G$. (b) Schematic representation of the adjacency list structure of $G$. As in Figure 13.3, we visualize the elements of collections with names.

# Data Structures for Graphs: Adjacency Matrix

- The Adjacency Matrix Structure
- Space: $O(n^2)$
- O(1)
  - *endVertices()*
  - *opposite(v)*
  - *isIncidentOn(v)*
  - *isAdjacentTo(v)*
- O(n)
  - *incidentEdges()*

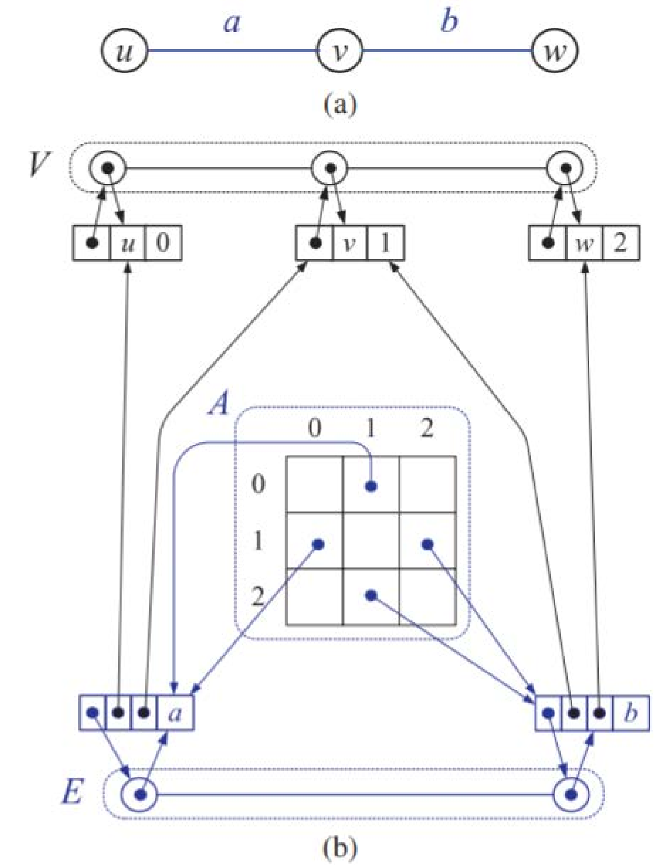


**Figure 13.5:** (a) A graph $G$ without parallel edges. (b) Schematic representation of the simplified adjacency matrix structure for $G$.

# Graph Traversals: Depth-First Search (DFS)

**Algorithm** DFS($G,v$):
  ***Input:*** A graph $G$ and a vertex $v$ of $G$
  ***Output:*** A labeling of the edges in the connected component of $v$ as discovery edges and back edges

```
label v as visited
for all edges e in v.incidentEdges() do
  if edge e is unvisited then
    w ← e.opposite(v)
    if vertex w is unexplored then
      label e as a discovery edge
      recursively call DFS(G,w)
    else
      label e as a back edge
```

**Code Fragment 13.1:** The DFS algorithm.

- Use Stack (implicitly by recursive function)
- Discovery edges form a spanning tree
- DFS: O(n+m)
  - *Test whether G is connected*
  - *Compute a spanning tree*
  - *Compute a path if exists*
  - *Compute a cycle in G*

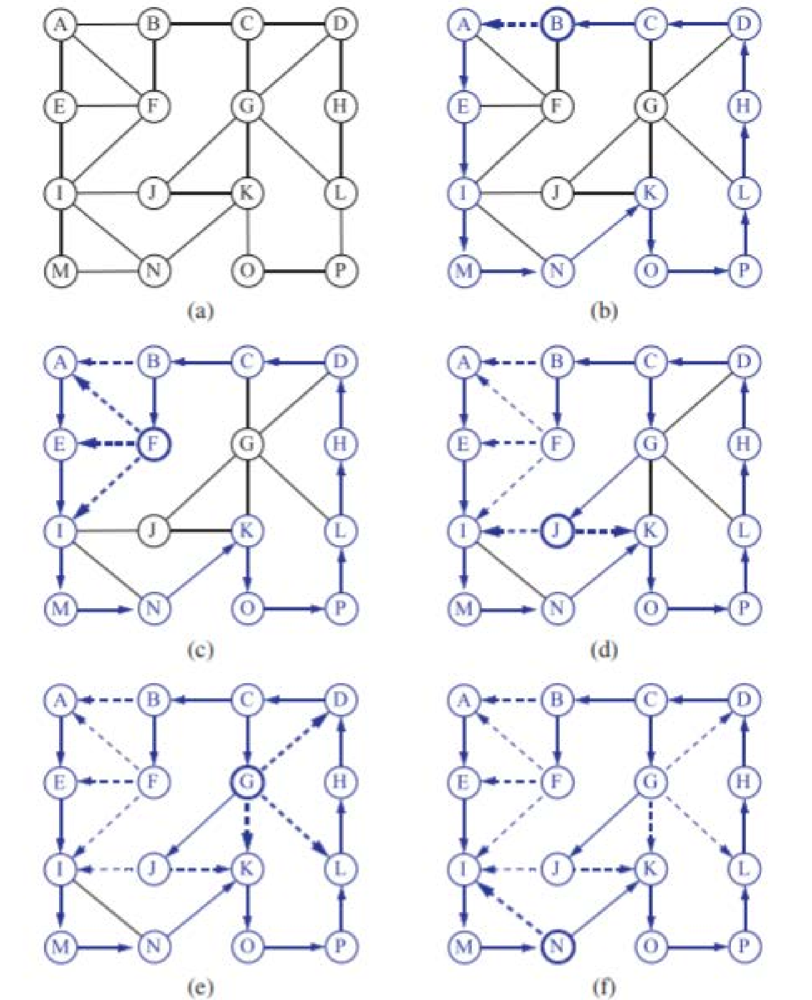


**Figure 13.6:** Example of depth-first search traversal on a graph starting at vertex $A$. Discovery edges are shown with solid lines and back edges are shown with dashed lines: (a) input graph; (b) path of discovery edges traced from A until back edge (B,A) is hit; (c) reaching F, which is a dead end; (d) after backtracking to C, resuming with edge (C,G), and hitting another dead end, J; (e) after backtracking to G; (f) after backtracking to N.

# Graph Traversals: Breadth-First Search (BFS)

```
Algorithm BFS(s):
    initialize collection L_0 to contain vertex s
    i ← 0
    while L_i is not empty do
        create collection L_{i+1} to initially be empty
        for all vertices v in L_i do
            for all edges e in v.incidentEdges() do
                if edge e is unexplored then
                    w ← e.opposite(v)
                    if vertex w is unexplored then
                        label e as a discovery edge
                        insert w into L_{i+1}
                    else
                        label e as a cross edge
        i ← i+1
```

**Code Fragment 13.20:** The BFS algorithm.

- Use Queue (explicitly) to implement $L\_i$

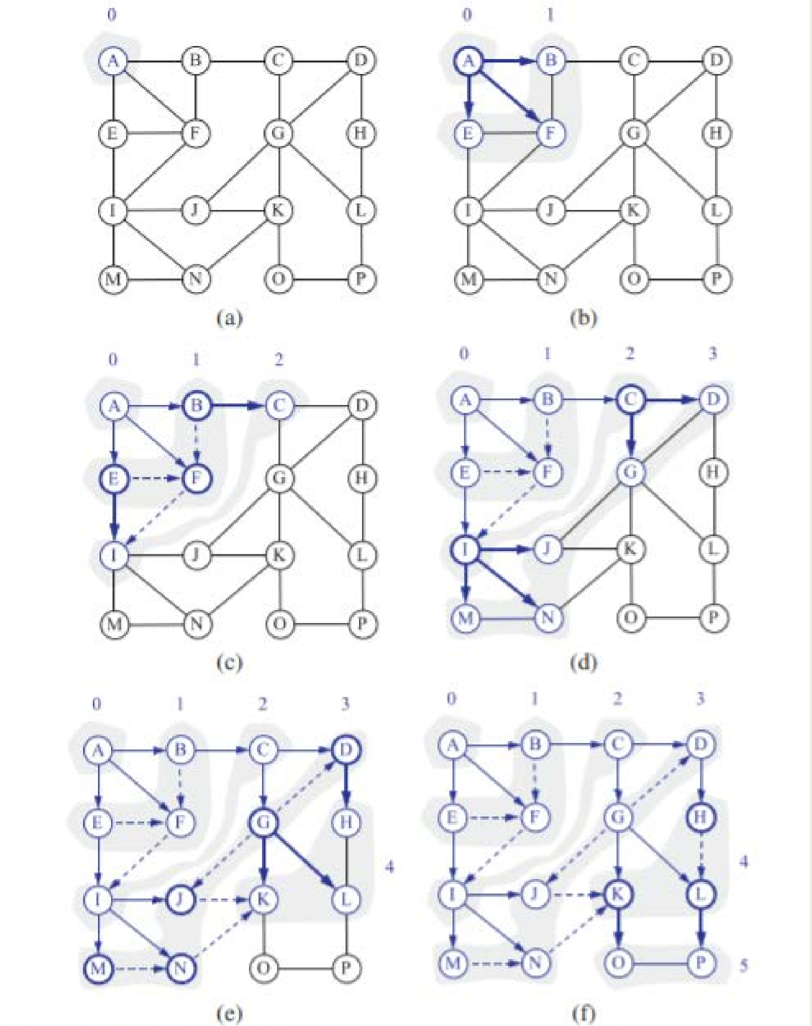


**Figure 13.7:** Example of breadth-first search traversal, where the edges incident on a vertex are explored by the alphabetical order of the adjacent vertices. The discovery edges are shown with solid lines and the cross edges are shown with dashed lines: (a) graph before the traversal; (b) discovery of level 1; (c) discovery of level 2; (d) discovery of level 3; (e) discovery of level 4; (f) discovery of level 5.

# Directed Graphs, Directed Acyclic Graphs (DAG)

- A directed graph G is **strongly connected** if, for any two vertices u and v, u reaches v and v reaches u.
- DFS or BFS on G starting at a vertex s visits all the vertices of G that are reachable from s.
- How to test whether G is strongly connected?
  - *Perform DFS starting at an arbitrary vertex s.*
  - *If there is any vertex not reachable, not strongly connected*
  - *Reverse all the edges and perform another DFS starting at s*
  - *If visit every vertex, strongly connected*
- Directed Acyclic Graphs (DAG)
  - *C++ class inheritance*
  - *Course prerequisites*
  - *Task scheduling*
- A topological ordering of G is an ordering of vertices such that for every edge $(v_i, v_j)$ of G, $i < j$
  - *TopologicalSort(G): Refer to Code Frag. 13.23*

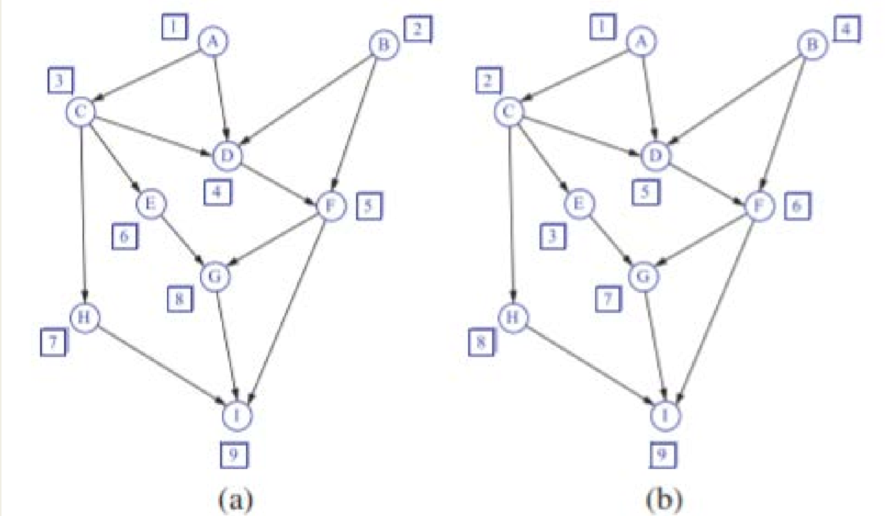


**Figure 13.11:** Two topological orderings of the same acyclic digraph.

# Shortest Paths (Dijkstra's algorithm, Uniform Cost Search)

- Weighted Graphs: each edge has a weight w(e)
- A path $P = ((v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k))$
- The weight of $P$: $w(P) = \sum_{i=0}^{k-1} w((v_i, v_{i+1}))$
- If all weights are the same non-negative value, BFS computes the shortest paths from one to many (single-source problem).
- Dijkstra's Algorithm
  - *A kind of "weighted" BFS*
  - *Whenever a vertex u is pulled into the cloud, D[u] is the length of the shortest path from v to u.*
  - *Proof?*
  - $O((n+m)\log n)$

**Algorithm** ShortestPath($G, v$):
  ***Input:*** A simple undirected weighted graph $G$ with nonnegative edge weights and a distinguished vertex $v$ of $G$
  ***Output:*** A label $D[u]$, for each vertex $u$ of $G$, such that $D[u]$ is the length of a shortest path from $v$ to $u$ in $G$
  Initialize $D[v] \leftarrow 0$ and $D[u] \leftarrow +\infty$ for each vertex $u \neq v$.
  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}
    $u \leftarrow Q$.removeMin()
    **for** each vertex $z$ adjacent to $u$ such that $z$ is in $Q$ **do**
      {perform the ***relaxation*** procedure on edge $(u, z)$}
      **if** $D[u] + w((u,z)) < D[z]$ **then**
        $D[z] \leftarrow D[u] + w((u,z))$
        Change to $D[z]$ the key of vertex $z$ in $Q$.
  **return** the label $D[u]$ of each vertex $u$

**Code Fragment 13.24:** Dijkstra's algorithm for the single-source, shortest-path problem.

# Shortest Paths

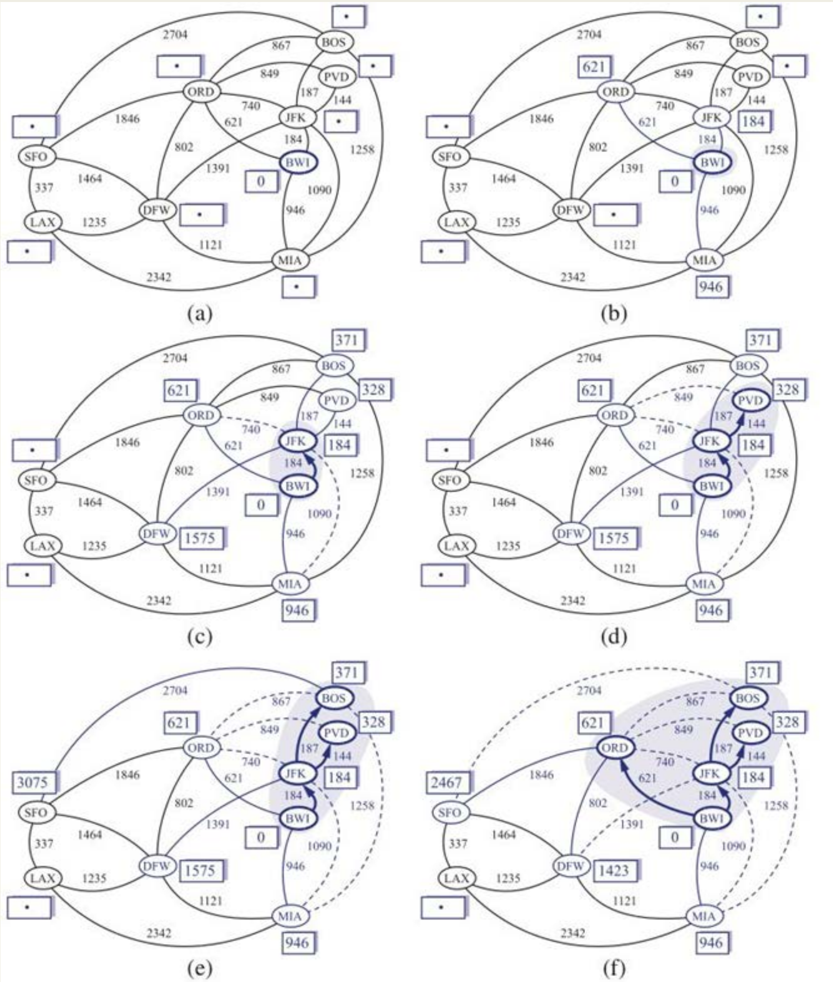


**Figure 13.14:** An execution of Dijkstra's algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex $v$ stores the label $D[v]$. The symbol $\bullet$ is used instead of $+\infty$. The edges of the shortest-path tree are drawn as thick blue arrows and, for each vertex $u$ outside the "cloud," we show the current best edge for pulling in $u$ with a solid blue line. (Continues in Figure 13.15.)

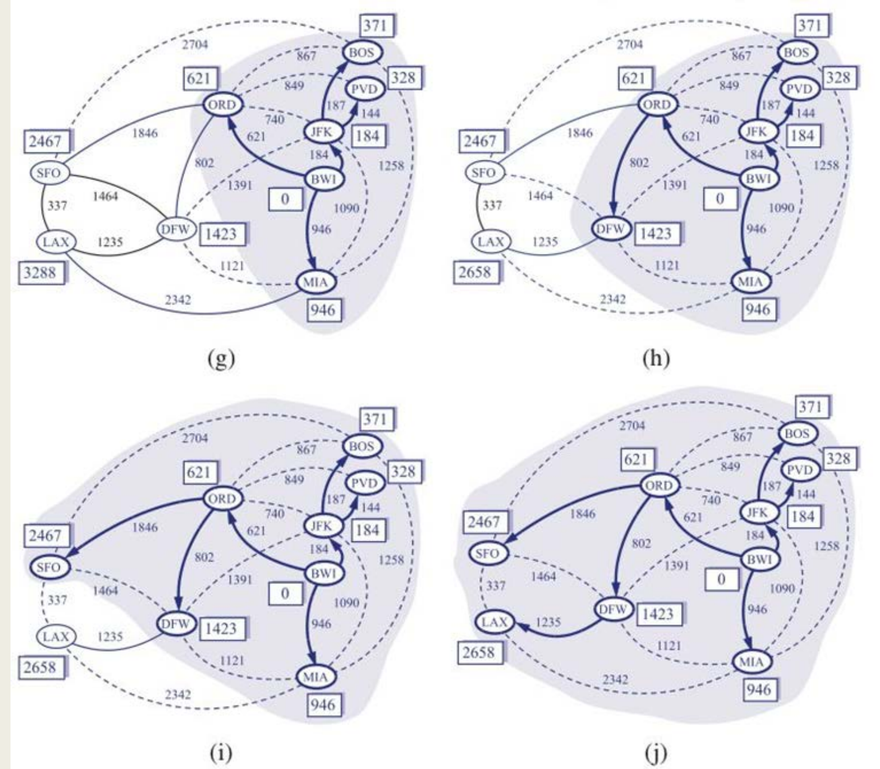


**Figure 13.15:** An example execution of Dijkstra's algorithm. (Continued from Figure 13.14.)

# Minimum Spanning Trees

- Want to connect all computers with the least amount of cable.
- $w(T) = \sum_{(v,u)\ in\ T} w((v,u))$
- $T = arg\min_{T} w(T)$
- A ***spanning tree*** is a tree containing every vertex.
- ***Minimum spanning tree (MST)*** problem is to find a spanning tree with smallest total weight.

**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 $C(v) \leftarrow \{v\}$.

Initialize a priority queue $Q$ to contain all edges in $G$, using the weights as keys.

$T \leftarrow \emptyset$     $\{T$ will ultimately contain the edges of the MST$\}$

**while** $T$ has fewer than $n-1$ edges **do**

  $(u,v) \leftarrow Q$.removeMin()

  Let $C(v)$ be the cluster containing $v$, and let $C(u)$ be the cluster containing $u$.

  **if** $C(v) \neq C(u)$ **then**

    Add edge $(v,u)$ to $T$.

    Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$.

**return** tree $T$

**Code Fragment 13.25:** Kruskal's algorithm for the MST problem.

- Proposition 13.24: Let *G* be a weighted connected graph, and let *V*1 and *V*2 be a partition of the vertices of *G* into two disjoint nonempty sets. Furthermore, let e be an edge in *G* with minimum weight from among those with one endpoint in *V*1 and the other in *V*2. There is a minimum spanning tree *T* that has *e* as one of its edges.
- Justification: Let *T* be a minimum spanning tree of *G*. If *T* does not contain edge *e*, the addition of *e* to *T* must create a cycle. Therefore, there is some edge *f* of this cycle that has one endpoint in *V*1 and the other in *V*2. Moreover, by the choice of *e*, *w*(*e*) ≤ *w*( *f* ). If we remove *f* from *T* ∪{e}, we obtain a spanning tree whose total weight is no more than before. Since *T* is a minimum spanning tree, this new tree must also be a minimum spanning tree
- Kruskal algorithm: $O((m+n)\log m)$

# Minimum Spanning Trees: Kruskal algorithm

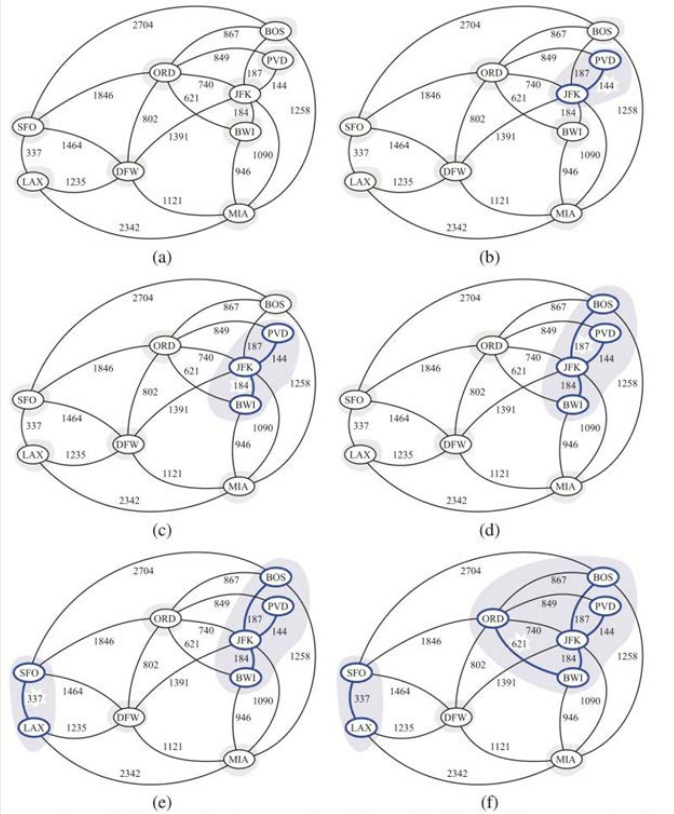

**Figure 13.18:** Example of an execution of Kruskal's MST algorithm on a graph with integer weights. We show the clusters as shaded regions and we highlight the edge being considered in each iteration. (Continues in Figure 13.19.)

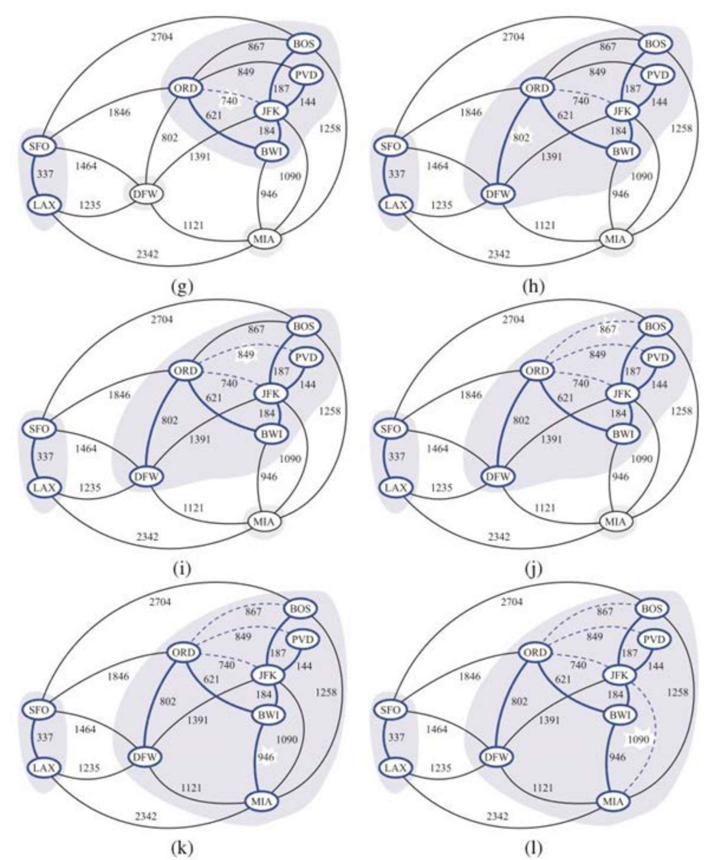

**Figure 13.19:** Example of an execution of Kruskal's MST algorithm. Rejected edges are shown dashed. (Continues in Figure 13.20.)

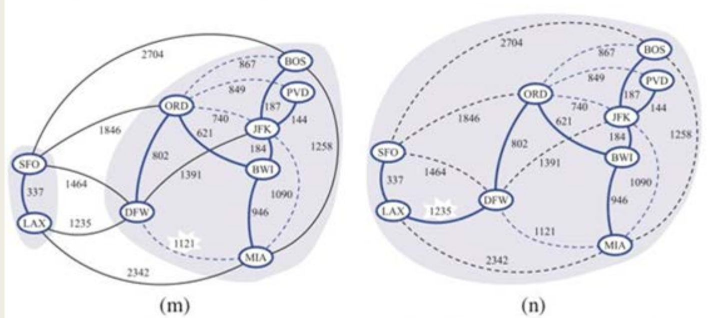

**Figure 13.20:** Example of an execution of Kruskal's MST algorithm. The edge considered in (n) merges the last two clusters, which concludes this execution of Kruskal's algorithm. (Continued from Figure 13.19.)

# Minimum Spanning Trees: Prim-Jarnik algorithm

- Prim-Jarnik algorithm: $O((m+n)\log n)$

**Algorithm** PrimJarnik($G$):
  ***Input:*** A weighted connected graph $G$ with $n$ vertices and $m$ edges
  ***Output:*** A minimum spanning tree $T$ for $G$

Pick any vertex $v$ of $G$
$D[v] \leftarrow 0$
**for** each vertex $u \neq v$ **do**
  $D[u] \leftarrow +\infty$
Initialize $T \leftarrow \emptyset$.
Initialize a priority queue $Q$ with an entry $((u, null), D[u])$ for each vertex $u$, where $(u, null)$ is the element and $D[u])$ is the key.
**while** $Q$ is not empty **do**
  $(u,e) \leftarrow Q$.removeMin()
  Add vertex $u$ and edge $e$ to $T$.
  **for** each vertex $z$ adjacent to $u$ such that $z$ is in $Q$ **do**
    {perform the relaxation procedure on edge $(u,z)$}
    **if** $w((u,z)) < D[z]$ **then**
      $D[z] \leftarrow w((u,z))$
      Change to $(z,(u,z))$ the element of vertex $z$ in $Q$.
      Change to $D[z]$ the key of vertex $z$ in $Q$.
**return** the tree $T$

**Code Fragment 13.26:** The Prim-Jarník algorithm for the MST problem.

# Minimum Spanning Trees: Prim-Jarnik Algorithm

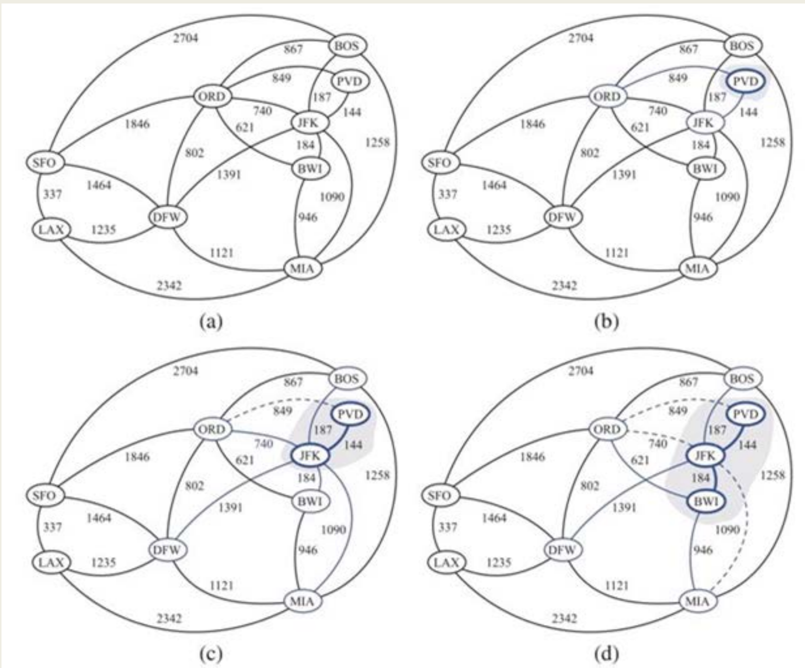


Figure 13.21: The Prim-Jarník MST algorithm. (Continues in Figure 13.22.)

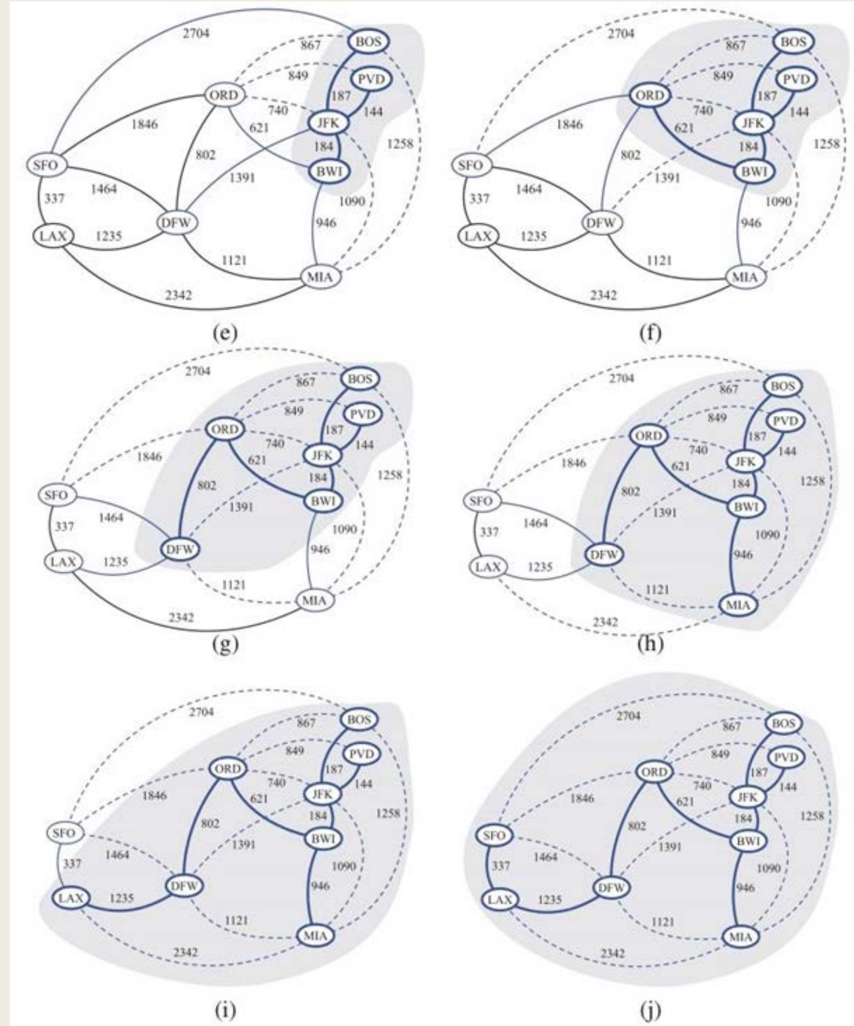


Figure 13.22: The Prim-Jarník MST algorithm. (Continued from Figure 13.21.)