Graphs

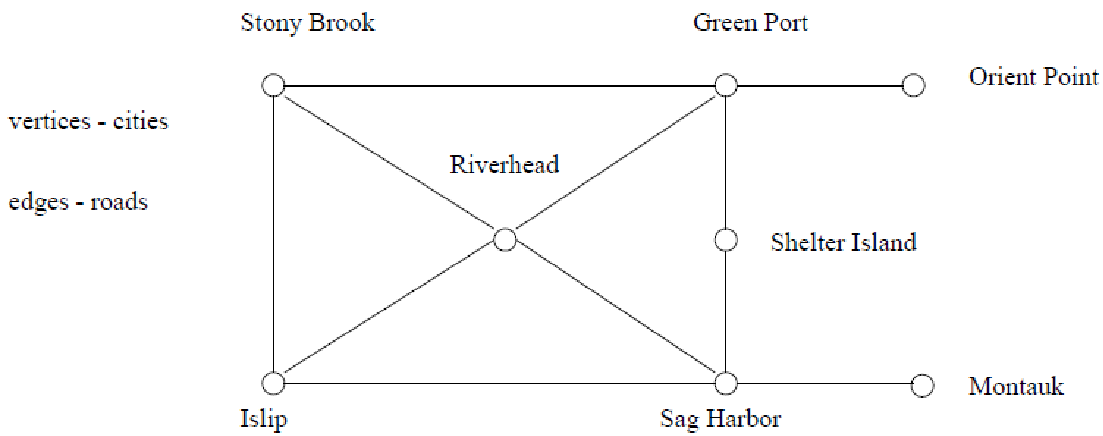
Initially prepared by Dr. İlyas Çiçekli; improved by various Bilkent CS202 instructors.

Graphs

- Graphs are one of the unifying themes of computer science.
- A graph G = (V, E) is defined by a set of *vertices V*, *and* a set of *edges E consisting of ordered or unordered pairs of* vertices from V .

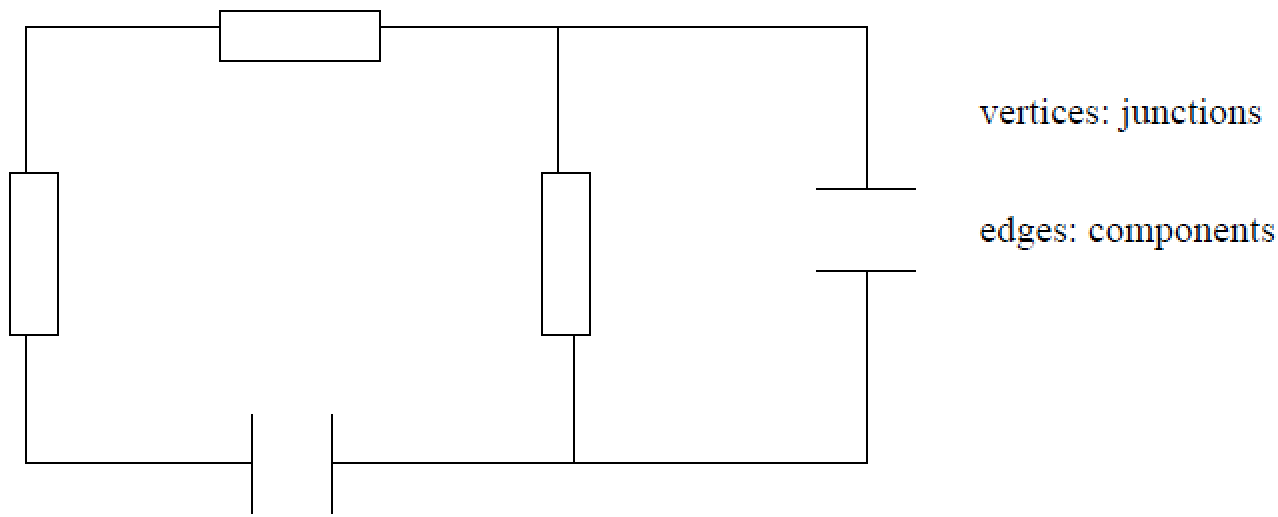
Road Networks

- In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.



Electronic Circuits

- In an electronic circuit, with junctions as vertices and components as edges.



Applications

- Social networks (facebook ...)
- Courses with prerequisites
- Computer networks
- Google maps
- Airline flight schedules
- Computer games
- WWW documents
- ... (so many to list!)

Graphs – Definitions

- A **graph** G = (V, E) consists of
  - a set of ***vertices***, V, and
  - a set of ***edges***, E, where each edge is a pair (v,w) s.t. v,w $\in$ V
- Vertices are sometimes called **nodes**, edges are sometimes called **arcs**.
- If the edge pair is ordered then the graph is called a **directed graph** **(**also called *digraphs)* .
- We also call a normal graph (which is not a directed graph) an **undirected graph**.
  - When we say graph we mean that it is an undirected graph.

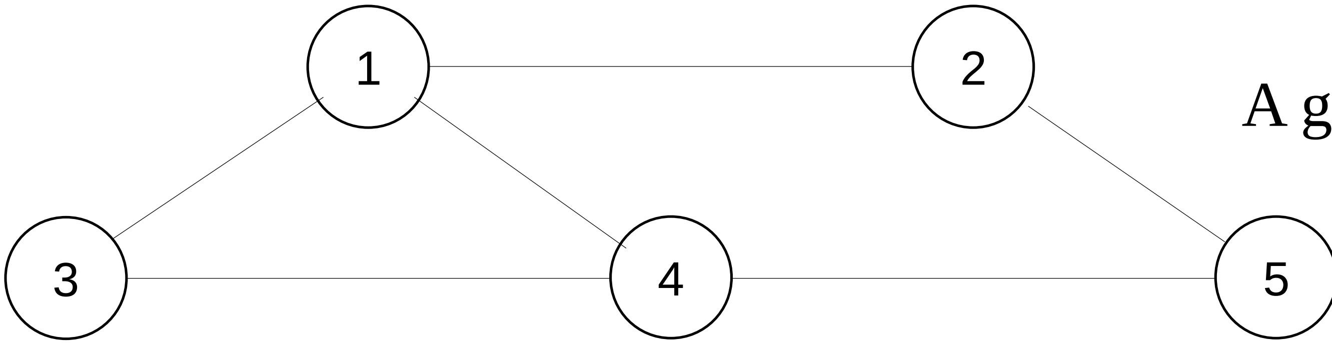
Graphs – Definitions

- Two vertices of a graph are **adjacent** if they are joined by an edge.
- Vertex w is **adjacent to** v iff $(v,w) \in E$.
  - In an undirected graph with edge (v, w) and hence (w,v) w is adjacent to v and v is adjacent to w.

- A **path** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex.
  - i.e. $w_1, w_2, \ldots, w_N$ is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq. N-1$
  - A **simple path** passes through a vertex only once.

- A **cycle** is a path that begins and ends at the same vertex.
  - A **simple cycle** is a cycle that does not pass through other vertices more than once.

Graphs – An Example



A graph G (undirected)

The graph G= (V,E) has 5 vertices and 6 edges:

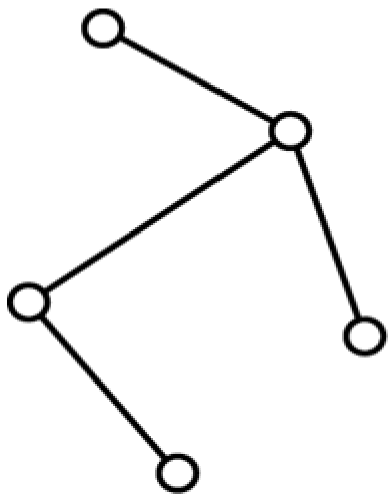
V = {1,2,3,4,5}

E = { (1,2),(1,3),(1,4),(2,5),(3,4),(4,5), (2,1),(3,1),(4,1),(5,2),(4,3),(5,4) }

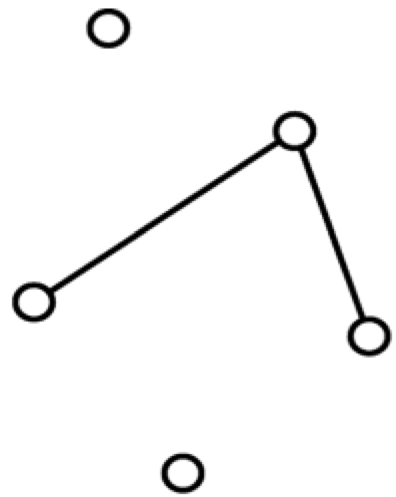
- *Adjacent:*
  1 and 2 are adjacent -- 1 is adjacent to 2 and 2 is adjacent to 1
- *Path:*
  1,2,5 (a simple path), 1,3,4,1,2,5 (a path but not a simple path)
- *Cycle:*
  1,3,4,1 (a simple cycle), 1,3,4,1,4,1 (cycle, but not simple cycle)

Graphs – Definitions

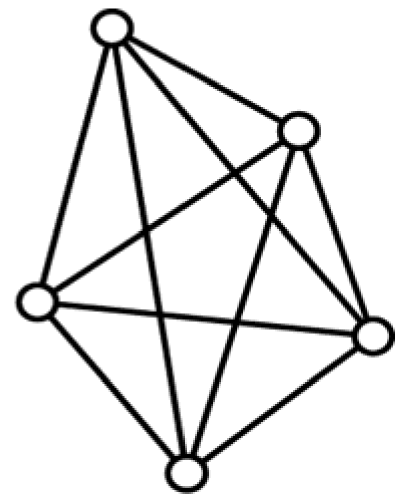
- A **connected graph** has a path between each pair of distinct vertices.
- A **complete graph** has an edge between each pair of distinct vertices.
  - A complete graph is also a connected graph. But a connected graph may not be a complete graph.



(a) **connected**



(b) **disconnected**



(c) **complete**

Directed Graphs

- If edge pair is ordered, then graph is called a **directed graph** **(**also called *digraphs)* .
  - Each edge in a directed graph has a direction. Called a **directed edge**.

- Definitions given for undirected graphs apply also to directed graphs, with changes that account for direction.

- Vertex w is ***adjacent to*** v iff $(v,w) \in E$.
  - i.e. There is a direct edge from v to w
  - w is **successor** of v
  - v is **predecessor** of w

- A **directed path** between two vertices is a sequence of directed edges that begins at one vertex and ends at another vertex.
  - i.e. $w_1, w_2, \ldots, w_N$ is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq. N-1$

Directed Graphs

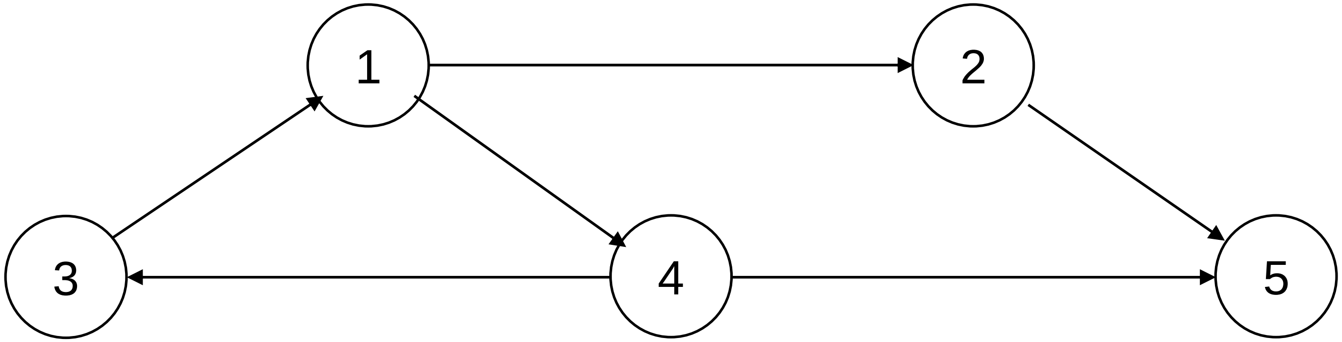
- A **cycle** in a directed graph is a path of length >= 1 such that $w_1 = w_N$.
  - This cycle is simple if the path is simple.
  - For undirected graphs, the edges must be distinct

- A **directed acyclic graph** (*DAG*) is a directed graph with no cycles.

- An undirected graph is **connected** if there is a path from every vertex to every other vertex.
  - A directed graph with this property is called **strongly connected**.
  - If a directed graph is not strongly connected, but the underlying graph (without direction to arcs) is connected then the graph is **weakly connected**.

total cycles --> traverse the graph with a node, if visit the same node, cycle + 1
continue with other nodes

Directed Graph – An Example



The graph G= (V,E) has 5 vertices and 6 edges:

V = {1,2,3,4,5}

E = { (1,2),(1,4),(2,5),(4,5),(3,1),(4,3) }

- *Adjacent:*

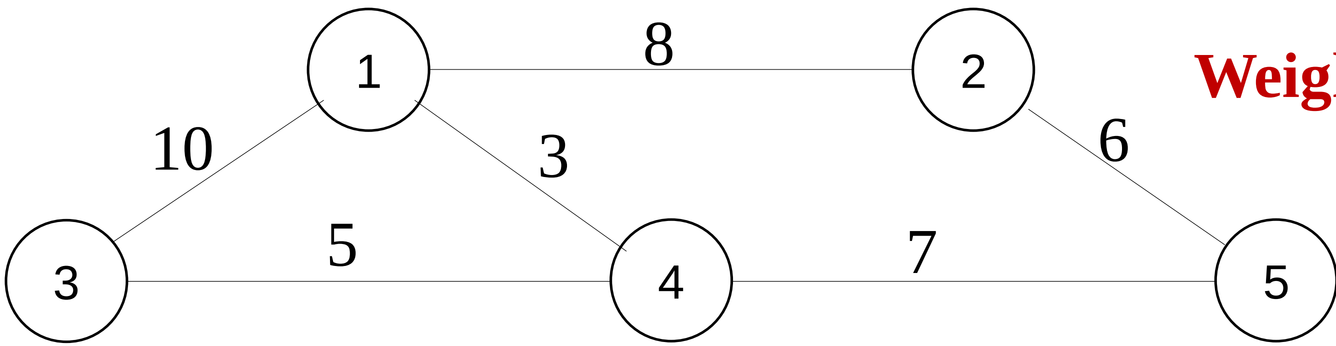
  2 is adjacent to 1, but 1 is NOT adjacent to 2
- *Path:*

  1,2,5 (a directed path),
- *Cycle:*

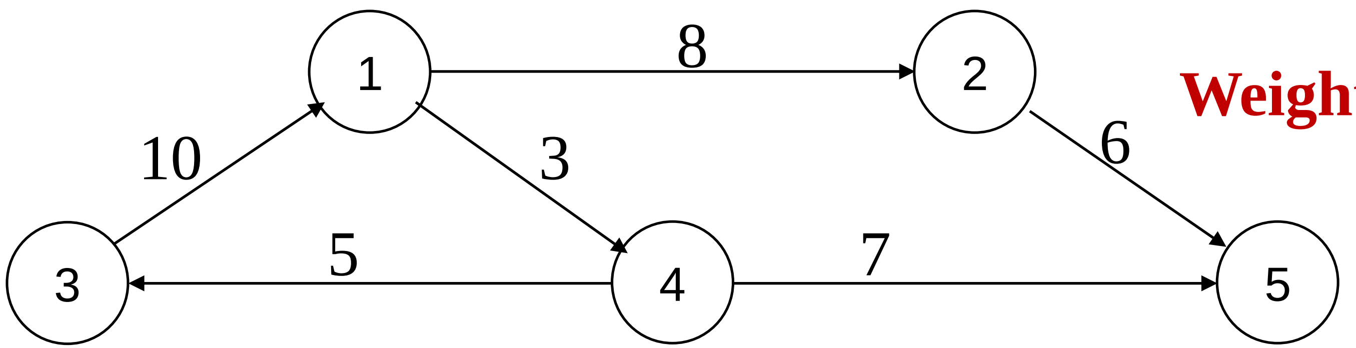
  1,4,3,1 (a directed cycle),

Weighted Graph

- We can label the edges of a graph with numeric values, the graph is called a **weighted graph**.



Weighted (Undirect) Graph



Weighted Directed Graph

Graph Implementations

- The two most common implementations of a graph are:

  – ***Adjacency Matrix***
    - A two dimensional array

  – ***Adjacency List***
    - For each vertex we keep a list of adjacent vertices

Adjacency Matrix

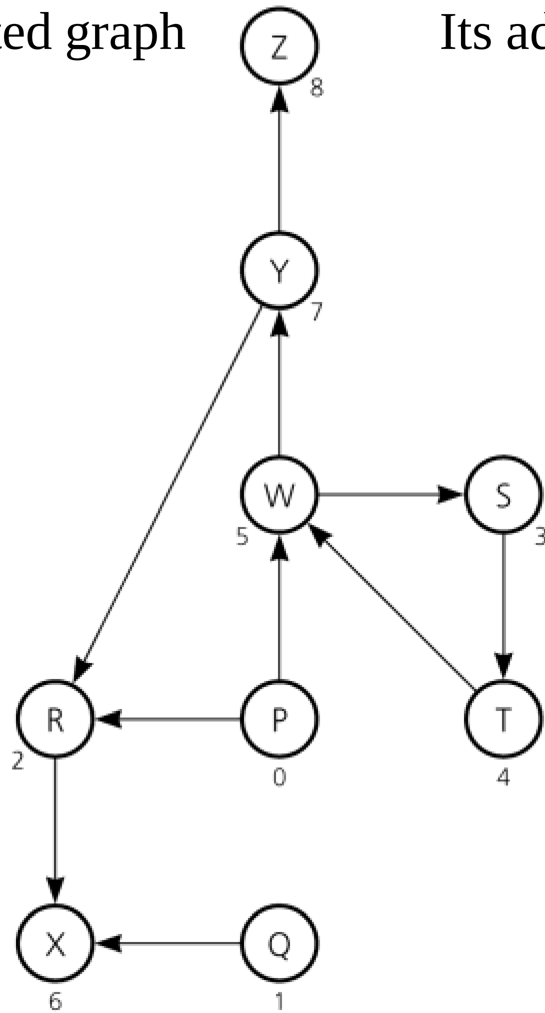
- An **adjacency matrix** for a graph with *n* vertices is an ***n* by *n* array *matrix*** such that
  - *matrix[i][j]* is 1 (true) if there is an edge from vertex *i* to vertex *j*
  - 0 (false) otherwise.
- When the graph is **weighted:**
  - *matrix[i][j]* is weight that labels edge from vertex *i* to vertex *j* instead of simply 1,
  - *matrix[i][j]* equals to ∞ instead of 0 when there is no edge from vertex *i* to *j*
- Adjacency matrix for an undirected graph is symmetrical.
  - i.e. *matrix[i][j]* is equal to *matrix[j][i]*

shortest path etc. can be confusing if 0 is used

- Space requirement **$O(|V|^2)$**
  - Acceptable if the graph is dense.

Adjacency Matrix – Example1

A directed graph

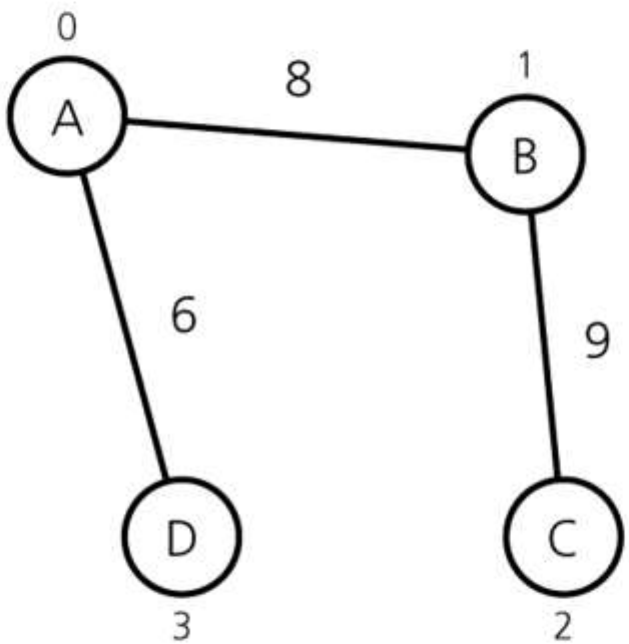


Its adjacency matrix

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Adjacency Matrix – Example2

An Undirected Weighted Graph



Its Adjacency Matrix

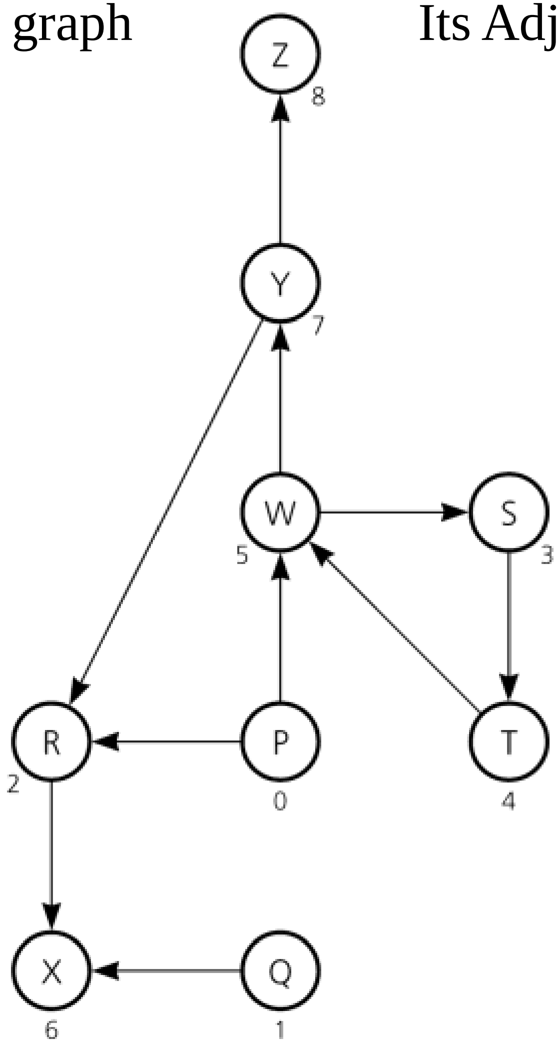
		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Adjacency List

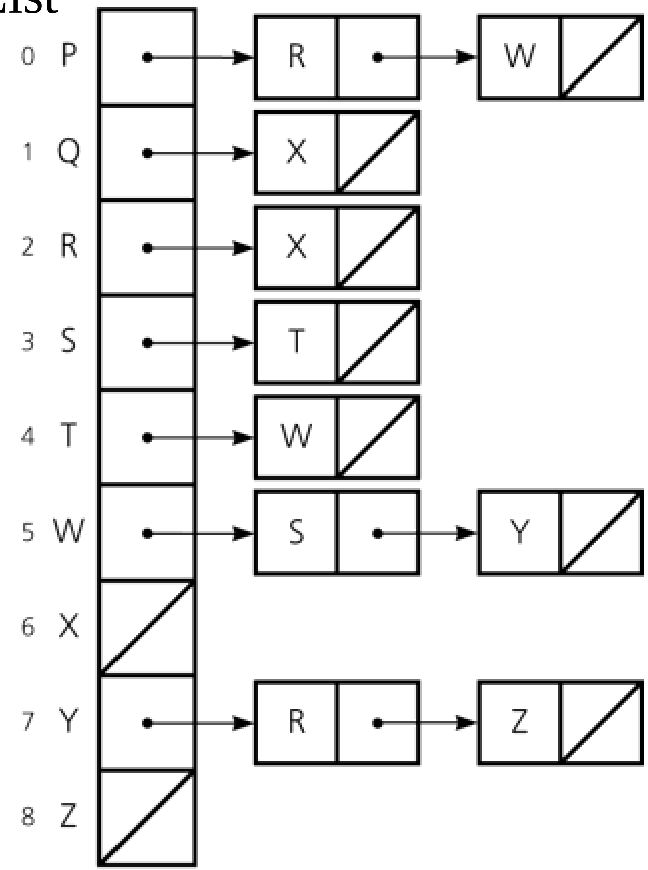
- An **adjacency list** for a graph with *n* vertices numbered 0,1,...,n-1 consists of *n* linked lists. The i^{th} linked list has a node for vertex *j* if and only if the graph contains an edge from vertex *i* to vertex *j*.
- Adjacency list is a better solution if the graph is sparse.
- Space requirement is **O(|E| + |V|),** which is linear in the size of the graph.
- In an undirected graph each edge (v, w) appears in two lists.
  - Space requirement is doubled.

Adjacency List – Example1

A directed graph

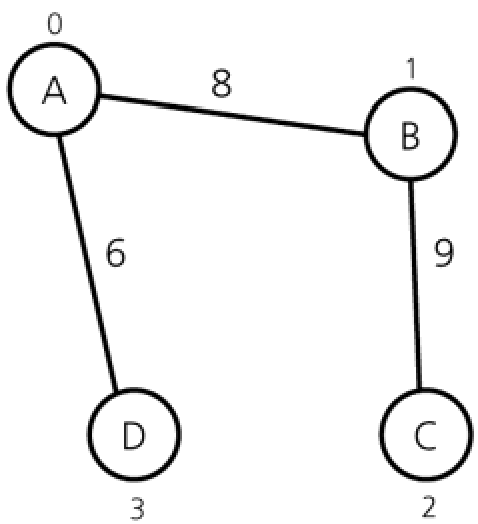


Its Adjacency List

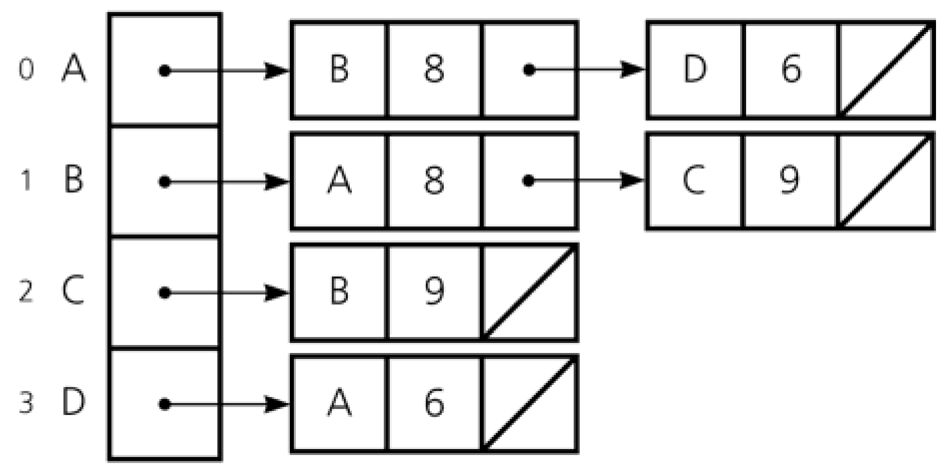


Adjacency List – Example2

An Undirected Weighted Graph



Its Adjacency List



Adjacency Matrix vs Adjacency List

- Two common graph operations:
  1. **Determine whether there is an edge** from vertex i to vertex j.
  2. **Find all vertices adjacent** to a given vertex i.

- An adjacency matrix supports operation 1 more efficiently.
- An adjacency list supports operation 2 more efficiently.

- An adjacency list often requires less space than an adjacency matrix.
  - Adjacency Matrix: Space requirement is $O(|V|^2)$
  - Adjacency List : Space requirement is $O(|E| + |V|)$, which is linear in the size of the graph.
  - Adjacency matrix is better if the graph is dense (too many edges)
  - Adjacency list is better if the graph is sparse (few edges)

Tradeoffs Between Adjacency Lists and Adjacency Matrices

- Faster to test if (x; y) exists?
- Faster to find vertex degree?
- Less memory on sparse graphs?
- Less memory on dense graphs?
- Edge insertion or deletion?
- Faster to traverse the graph?
- Better for most problems?

Tradeoffs Between Adjacency Lists and Adjacency Matrices

• Faster to test if (x; y) exists?	**matrices**
• Faster to find vertex degree?	**lists**
• Less memory on sparse graphs?	**lists – (m+ n) vs. (n^2)**
• Less memory on dense graphs?	**matrices (small win)**
• Edge insertion or deletion?	**matrices O(1)**
• Faster to traverse the graph?	**lists – (m + n) vs. (n^2)**
• Better for most problems?	**lists**

list -> O(V+E)
matrix -> O(V^2)

Graph Traversals

- A **graph-traversal** algorithm starts from a vertex v, **visits all of the vertices** that can be reachable from the vertex v.
  - A graph-traversal algorithm visits all vertices if and only if the graph is connected.
- A **connected component** is the subset of vertices visited during a traversal algorithm that begins at a given vertex.
- A graph-traversal algorithm must **mark each vertex** during a visit and must never visit a vertex more than once.
  - Thus, if a graph contains a cycle, the graph-traversal algorithm can avoid infinite loop.
- We look at two graph-traversal algorithms:
  - **Depth-First Traversal**
  - **Breadth-First Traversal**

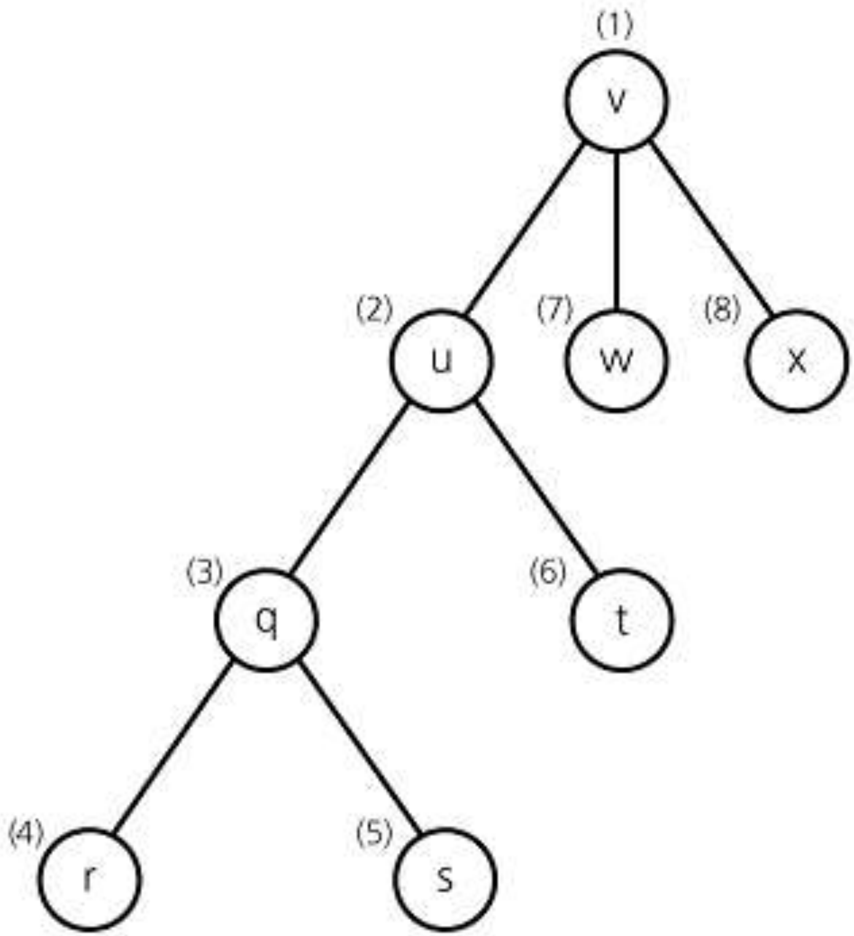
Depth-First Traversal

- For a given vertex v, **depth-first traversal** algorithm proceeds along a path from v as deeply into the graph as possible before backing up.

- That is, after visiting a vertex v, the algorithm visits (if possible) an unvisited adjacent vertex to vertex v.

- The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  - We may visit the vertices adjacent to v in sorted order.

Depth-First Traversal – Example



- A depth-first traversal of the graph starting from vertex v.

- Visit a vertex, then visit a vertex adjacent to that vertex.

- If there is no unvisited vertex adjacent to visited vertex, back up to the previous step.

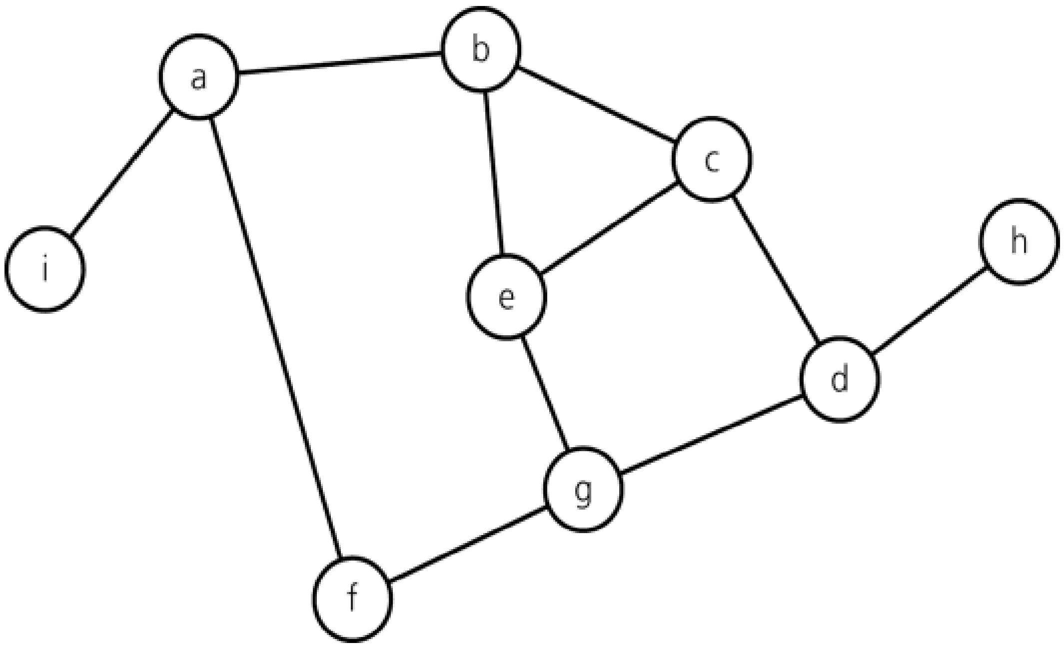
Recursive Depth-First Traversal Algorithm

```
dft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy
// Recursive Version
   Mark v as visited;
   for (each unvisited vertex u adjacent to v)
        dft(u)
}
```

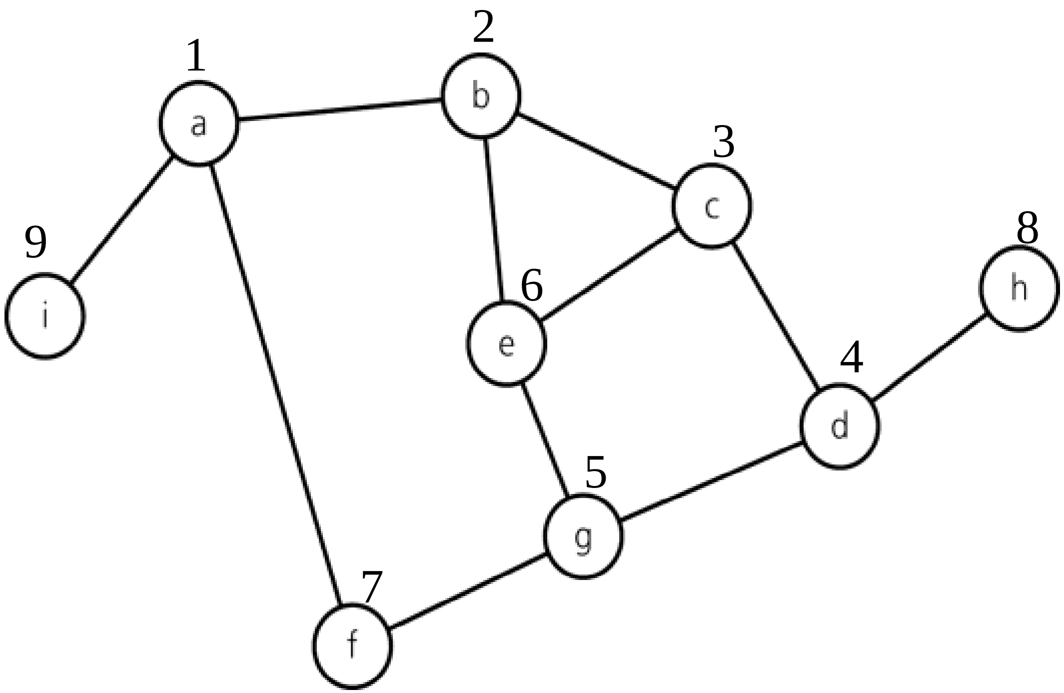
Iterative Depth-First Traversal Algorithm

```
dft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy: Iterative Version
   s.createStack();
   // push v into the stack and mark it
   s.push(v);
   Mark v as visited;
   while (!s.isEmpty()) {
      if (no unvisited vertices are adjacent to the vertex on
          the top of stack)
         s.pop();  // backtrack
      else {
         Select an unvisited vertex u adjacent to the vertex
            on the top of the stack;
         s.push(u);
         Mark u as visited;
      }
   }
}
```

Trace of Iterative DFT – starting from vertex a



Trace of Iterative DFT – starting from vertex a



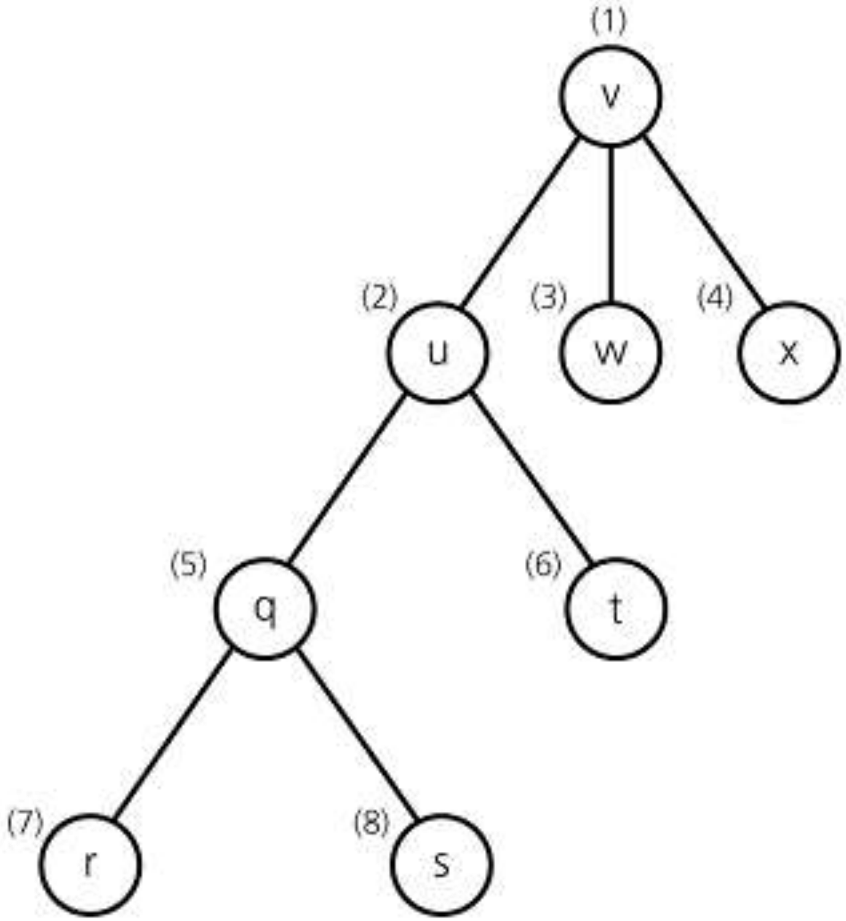
Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

Breadth-First Traversal

- After visiting a given vertex v, the **breadth-first traversal** algorithm visits every vertex adjacent to v that it can before visiting any other vertex.

- The breadth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  - We may visit the vertices adjacent to v in sorted order.

Breadth-First Traversal – Example



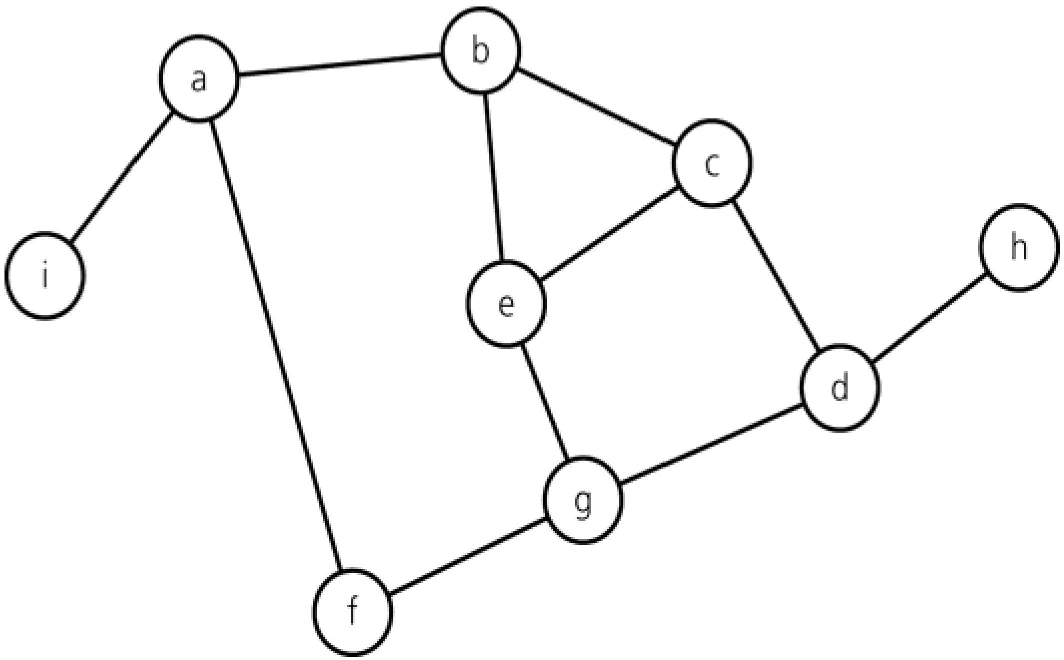
- A breadth-first traversal of the graph starting from vertex v.

- Visit a vertex, then visit all vertices adjacent to that vertex.

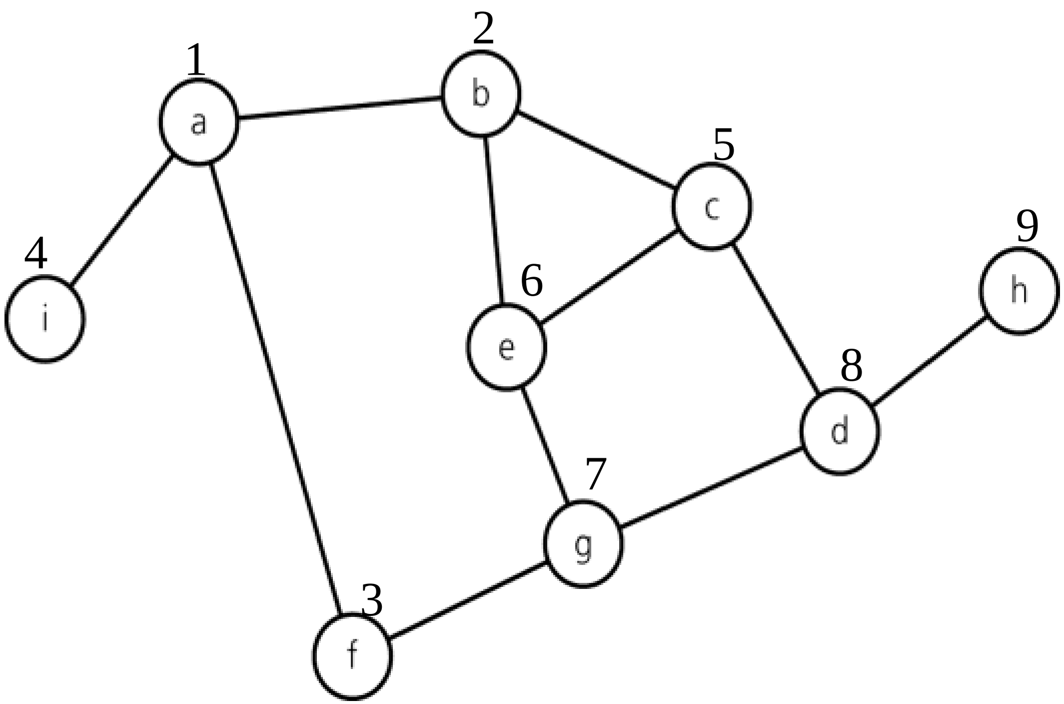
Iterative Breadth-First Traversal Algorithm

```
bft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using breadth-first strategy: Iterative Version
   q.createQueue();
   // add v to the queue and mark it
   Mark v as visited;
   q.enqueue(v);
   while (!q.isEmpty()) {
      q.dequeue(w);
      for (each unvisited vertex u adjacent to w) {
         Mark u as visited;
         q.enqueue(u);
      }
   }
}
```

Trace of Iterative BFT – starting from vertex a



Trace of Iterative BFT – starting from vertex a



Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

Topological Sorting

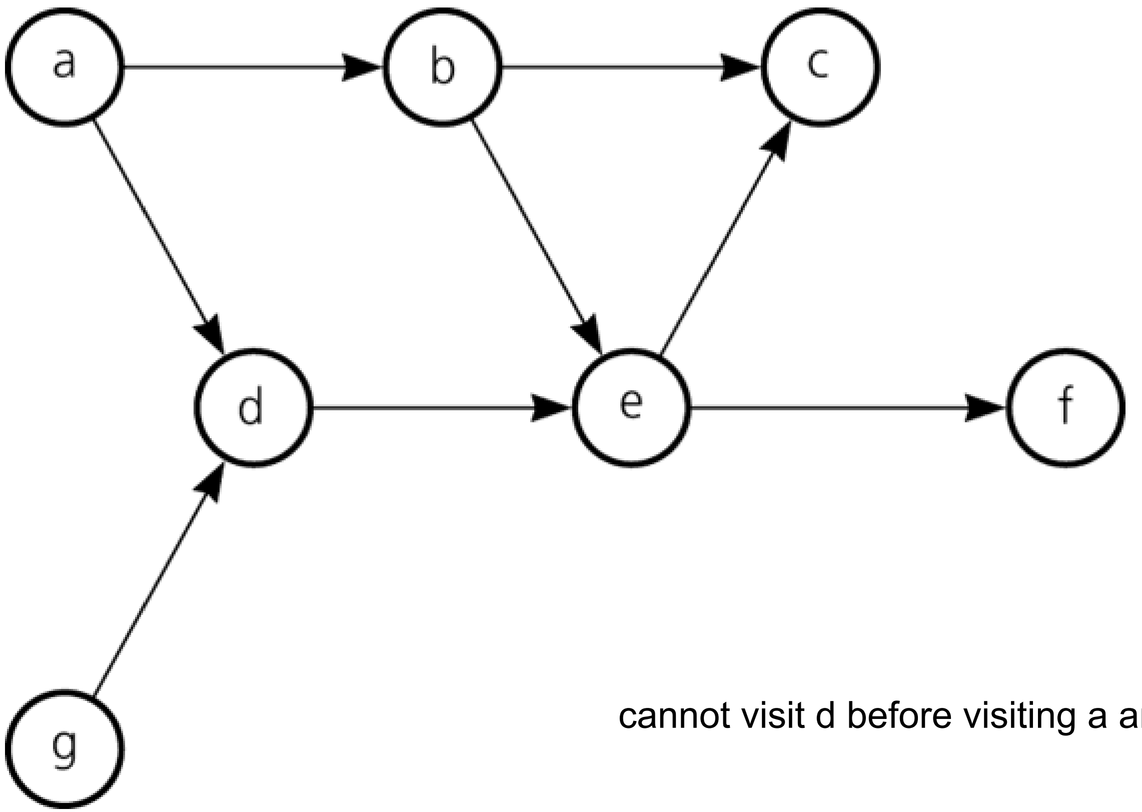
- A directed graph without cycles has a natural order.
  - That is, vertex a precedes vertex b, which precedes c
  - For example, the prerequisite structure for the courses.

- In which order we should visit the vertices of a directed graph without cycles so that we can visit vertex v after we visit its predecessors.
  - This is a linear order, and it is known as **topological order**.

- For a given graph, there may be more than one topological order.

- Arranging the vertices into a topological order is called **topological sorting**.

Topological Order – Example



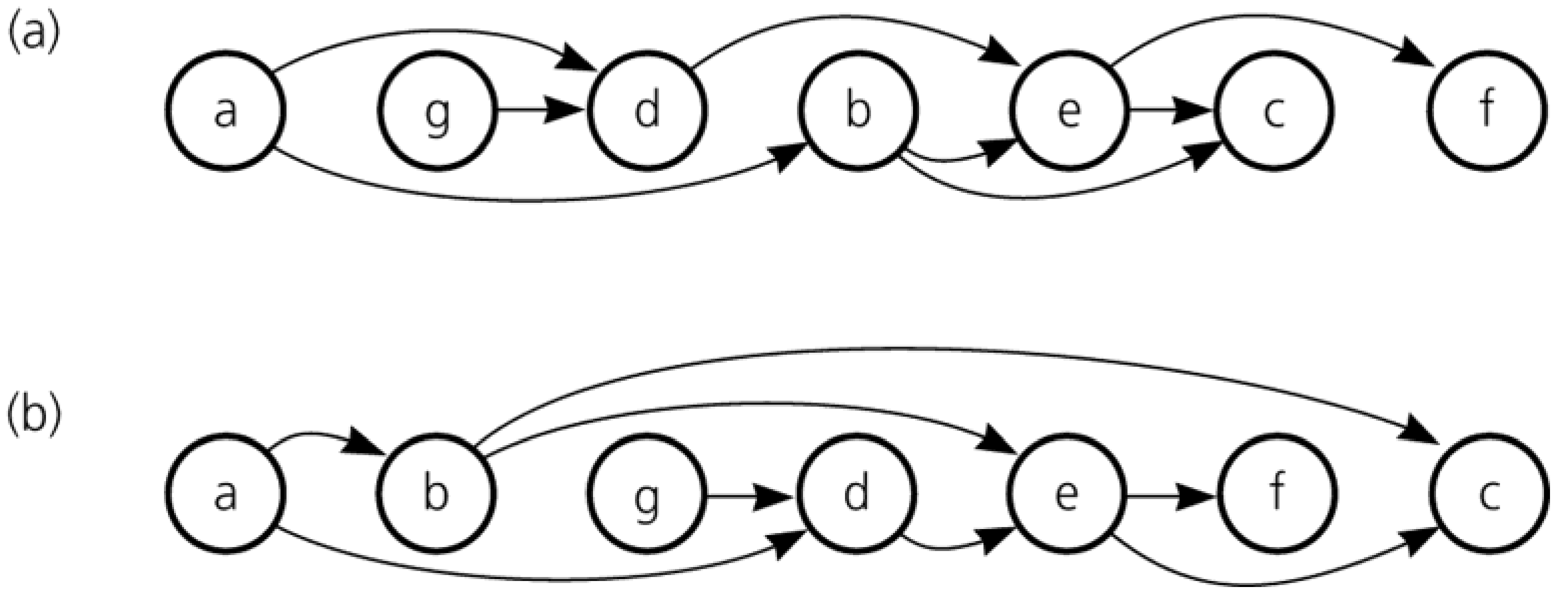
Some Topological Orders for this graph:

a, g ,d, b, e, c, f

a, b, g, d, e, f, c

cannot visit d before visiting a and g etc.

Topological Order – Example (cont.)



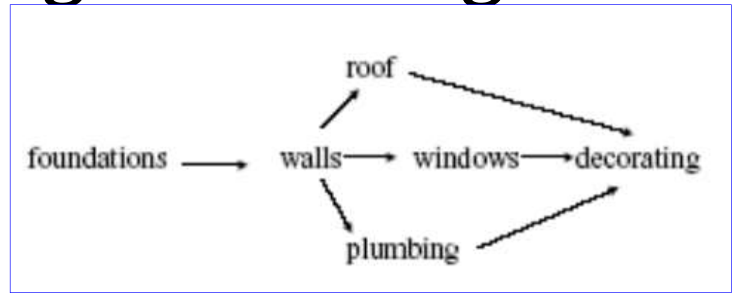
The graph arranged according to the topological orders

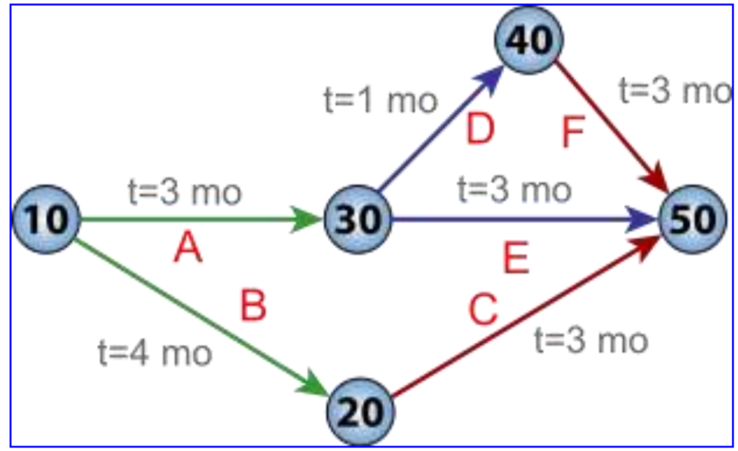
(a) a, g, d, b, e, c, f and

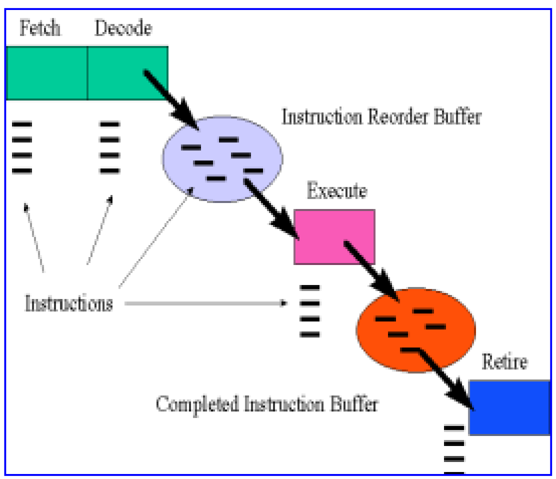
(b) a, b, g, d, e, f, c

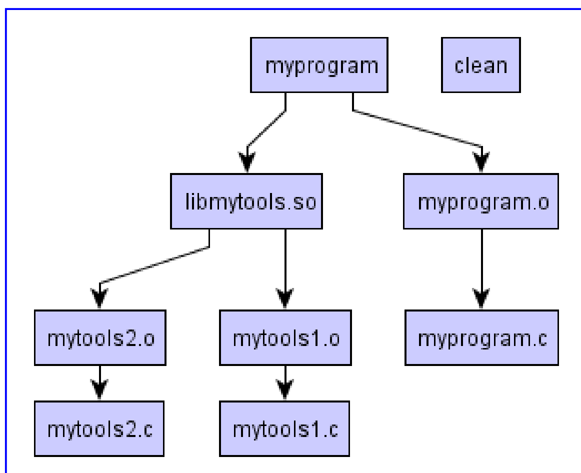
Applications of Topological Sorting

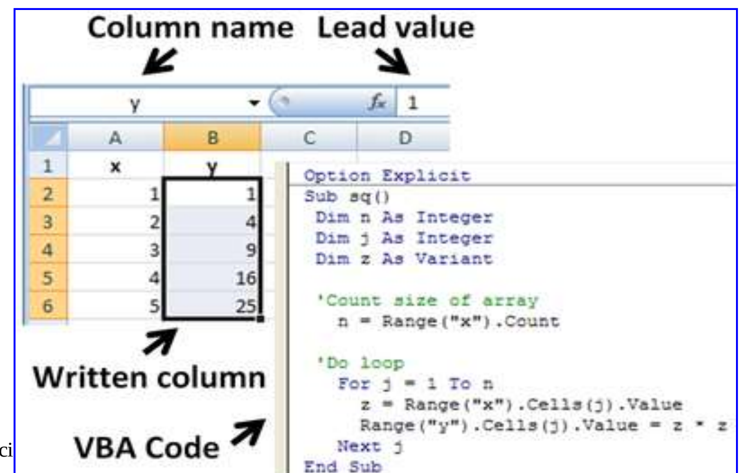
- Course prerequisites – which courses should you take next semester?
- Project scheduling (PERT)
- Processors: Instruction scheduling
- Spreadsheets
- Makefiles









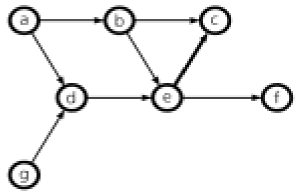


Simple Topological Sorting Algorithm1 – topSort1

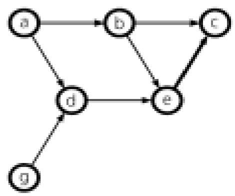
```
topSort1(in theGraph:Graph, out aList:List) {
// Arranges the vertices in graph theGraph into a
// toplogical order and places them in list aList
   n = number of vertices in theGraph;
   for (step=1 through n) {
      select a vertex v that has no successors;
      aList.insert(1,v);
      Delete from theGraph vertex v and its edges;
   }
}
```

starting from last vertex f

Graph `theGraph` | List `aList`

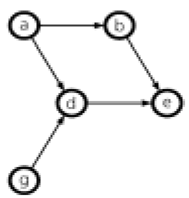


Remove f from `theGraph`;
add it to `aList`



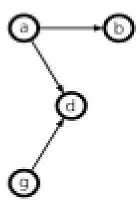
f

Remove c from `theGraph`;
add it to `aList`



c f

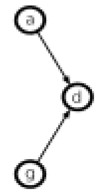
Remove e from `theGraph`;
add it to `aList`



e c f

Graph `theGraph` | List `aList`

Remove b from `theGraph`;
add it to `aList`



b e c f

Remove d from `theGraph`;
add it to `aList`

d b e c f

Remove g from `theGraph`;
add it to `aList`

g d b e c f

Remove a from `theGraph`;
add it to `aList`

a g d b e c f

DFS Topological Sorting Algorithm – topSort2

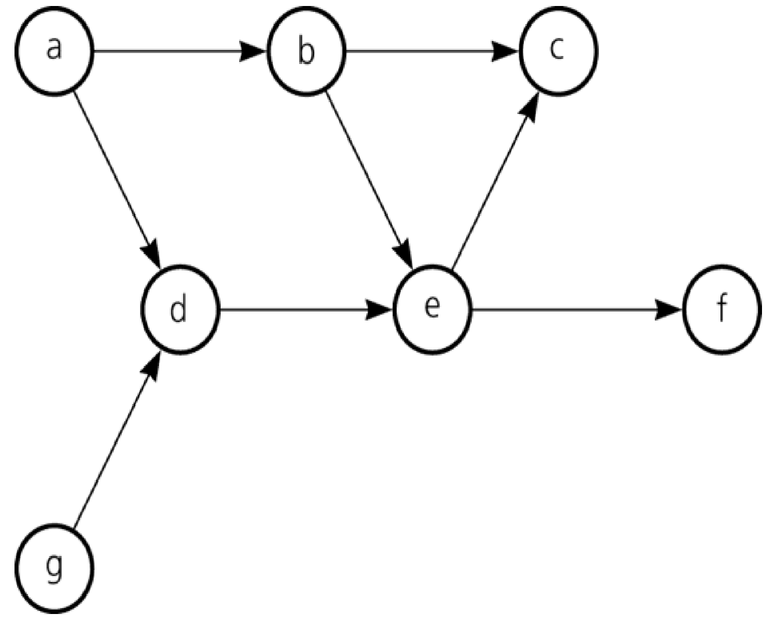
```
topSort2(in theGraph:Graph, out aList:List) {
// Arranges the vertices in graph theGraph into a topological order and
// places them in list aList
   s.createStack();
   for (all vertices v in the graph)
      if (v has no predecessors) {
         s.push(v);
         Mark v as visited;
      }
   while (!s.isEmpty()) {
      if (all vertices adjacent to the vertex on the top of stack
            have been visited) {
         s.pop(v);
         aList.insert(1,v);
      }
      else {
          Select an unvisited vertex u adjacent to the vertex on
             the top of the stack;
          s.push(u);
          Mark u as visited;
      }
   }
}
```

push a and g (initial)
for starting vertices

Remember: Iterative Depth-First Traversal Algorithm

```
dft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy: Iterative Version
   s.createStack();
   // push v into the stack and mark it
   s.push(v);
   Mark v as visited;
   while (!s.isEmpty()) {
      if (no unvisited vertices are adjacent to the vertex on
          the top of stack)
         s.pop();  // backtrack
      else {
         Select an unvisited vertex u adjacent to the vertex
            on the top of the stack;
         s.push(u);
         Mark u as visited;
      }
   }
}
```

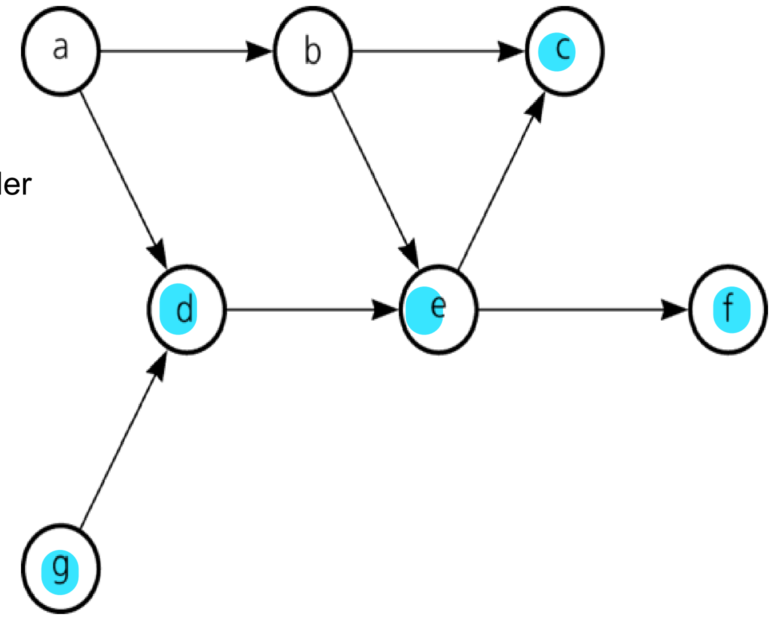
Trace of topSort2



Trace of topSort2

Action	Stack s (bottom to top)	List aList (beginning to end)
Push a	a	
Push g	a g	
Push d	a g d	
Push e	a g d e	
Push c	a g d e c	
Pop c, add c to **aList**	a g d e	c
Push f	a g d e f	c
Pop f, add f to **aList**	a g d e	f c
Pop e, add e to **aList**	a g d	e f c
Pop d, add d to **aList**	a g	d e f c
Pop g, add g to **aList**	a	g d e f c
Push b	a b	g d e f c
Pop b, add b to **aList**	a	b g d e f c
Pop a, add a to **aList**	*(empty)*	a b g d e f c

top = g
continue with unvisited vertices
adjacent to g, then d, then e etc.
c and f both adjacent to e, use sorted order

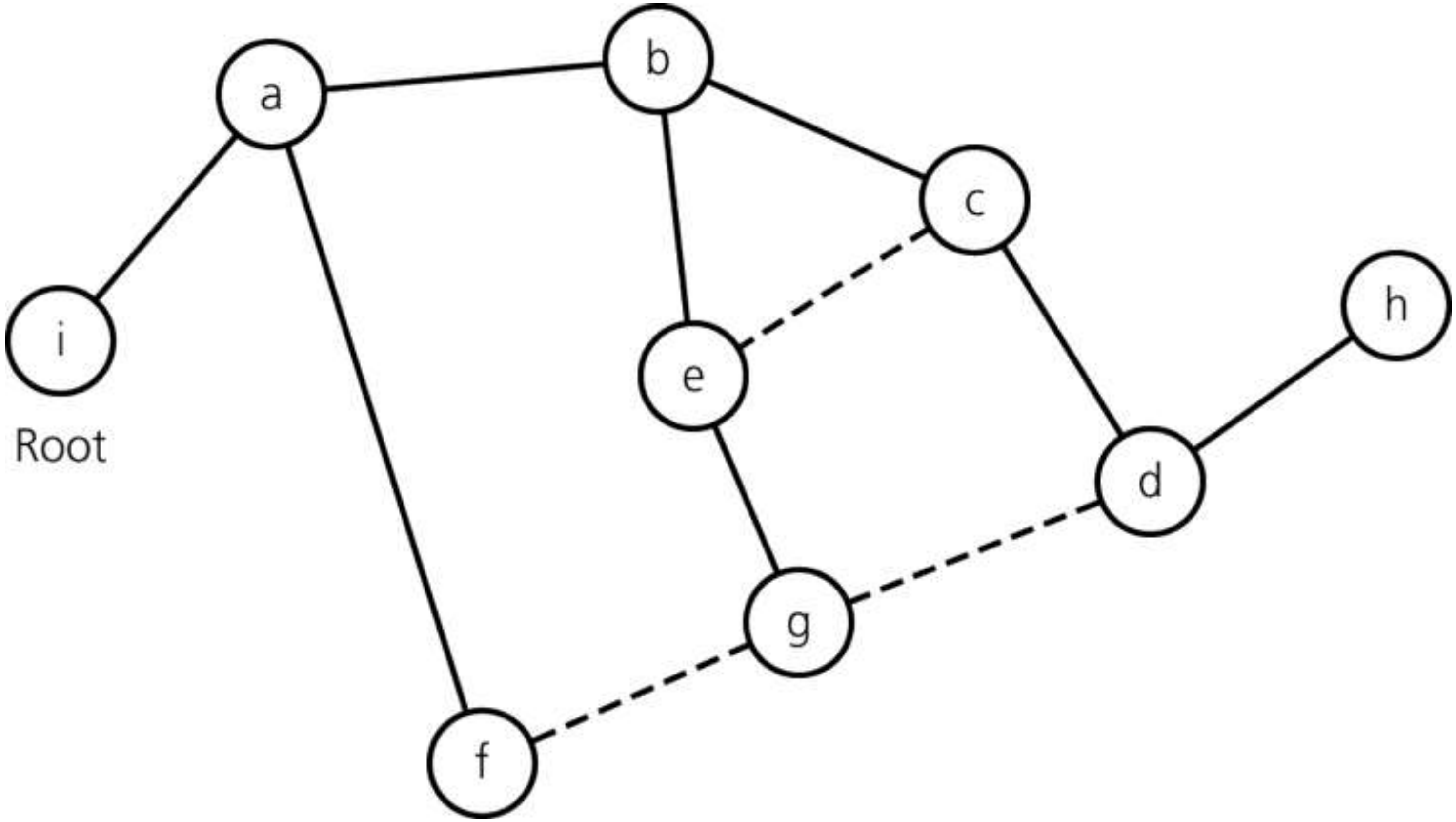


top = a, only adjacent unvisited is b, push b

Spanning Trees

- A tree is a special kind of undirected graph.
- That is, a **tree** is a connected undirected graph without cycles.
- All trees are graphs, not all graphs are trees. **Why?**
- A **spanning tree** of a connected undirected graph G is a sub-graph of G that contains all of G's vertices and enough of its edges to form a tree.
- There may be several spanning trees for a given graph.
- If we have a connected undirected graph with cycles, and we remove edges until there are no cycles to obtain a spanning tree.

A Spanning Tree



Remove dashed lines to obtain a spanning tree

Cycles?

Observations about graphs:

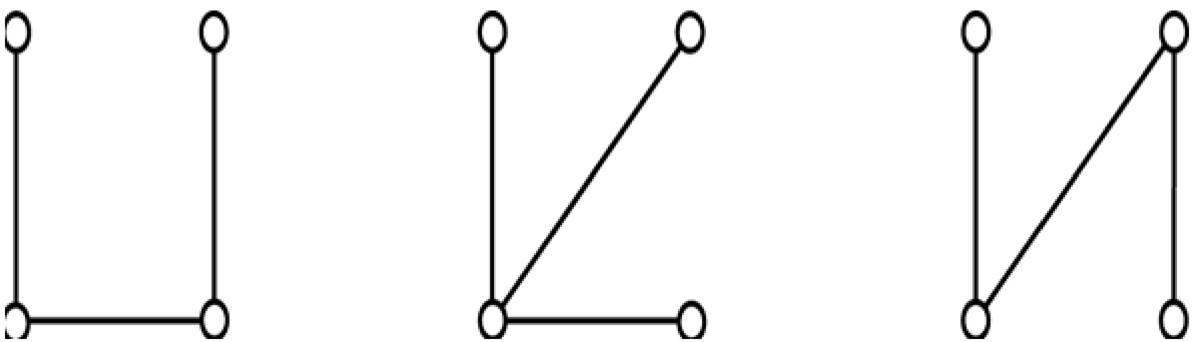
reverse is not true

1. A connected undirected graph that has n vertices must have at least n-1 edges.
2. A connected undirected graph that has n vertices and exactly n-1 edges cannot contain a cycle.

(tree)

3. A connected undirected graph that has n vertices and more than n-1 edges must contain a cycle.

cycle = minimum 2 edges



Connected graphs that each have four vertices and three edges

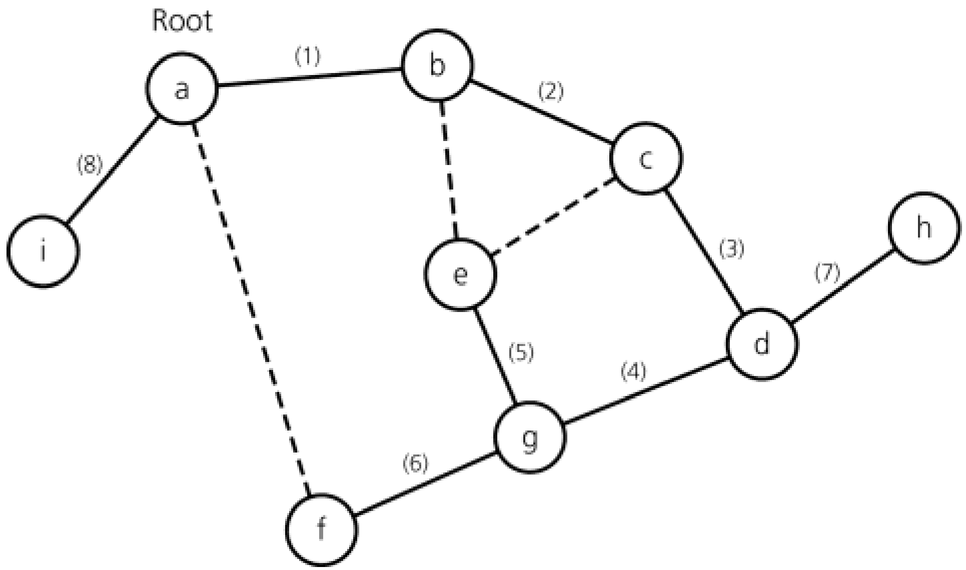
DFS Spanning Tree

```
dfsTree(in v:vertex) {
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using depth-first search;
// Recursive Version
   Mark v as visited;
   for (each unvisited vertex u adjacent to v) {
      Mark the edge from u tu v;
      dfsTree(u);
   }
}
```

Remember: Recursive Depth-First Traversal Algorithm

```
dft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy
// Recursive Version
   Mark v as visited;
   for (each unvisited vertex u adjacent to v)
        dft(u)
}
```

DFS Spanning Tree – Example



The DFS spanning tree algorithm visits vertices in this order: a, b, c, d, g, e, f, h, i. Numbers indicate the order in which the algorithm marks edges.

The DFS spanning tree rooted at vertex *a*

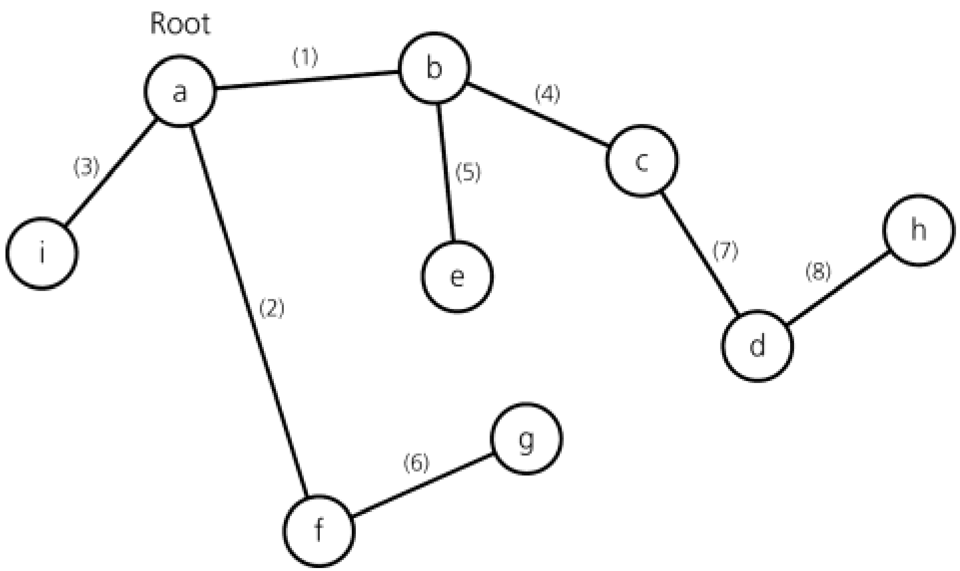
BFS Spanning tree

```
bfsTree(in v:vertex) {
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using breadth-first search;
// Iterative Version
   q.createQueue();
   q.enqueue(v);
   Mark v as visited;
   while (!q.isEmpty()) {
      q.dequeue(w);
      for (each unvisited vertex u adjacent to w) {
         Mark u as visited;
         Mark edge between w and u;
         q.enqueue(u);
      }
   }
}
```

Remember: Iterative Breadth-First Traversal Algorithm

```
bft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using breadth-first strategy: Iterative Version
   q.createQueue();
   // add v to the queue and mark it
   q.enqueue(v);
   Mark v as visited;
   while (!q.isEmpty()) {
      q.dequeue(w);
      for (each unvisited vertex u adjacent to w) {
         Mark u as visited;
         q.enqueue(u);
      }
   }
}
```

BFS Spanning tree – Example



The BFS spanning tree algorithm visits vertices in this order: a, b, f, i, c, e, g, d, h. Numbers indicate the order in which the algorithm marks edges.

The BFS spanning tree rooted at vertex *a*

Minimum Spanning Tree

- If we have a weighted connected undirected graph, the edges of each of its spanning tree will also be associated with costs.

- The *cost of a spanning tree* is the sum of the costs of edges in the spanning tree.

- A **minimum spanning tree** of a connected undirected graph has a minimal edge-weight sum.
  - A minimum spanning tree of a connected undirected may not be unique.
  - Although there may be several minimum spanning trees for a particular graph, their costs are equal.

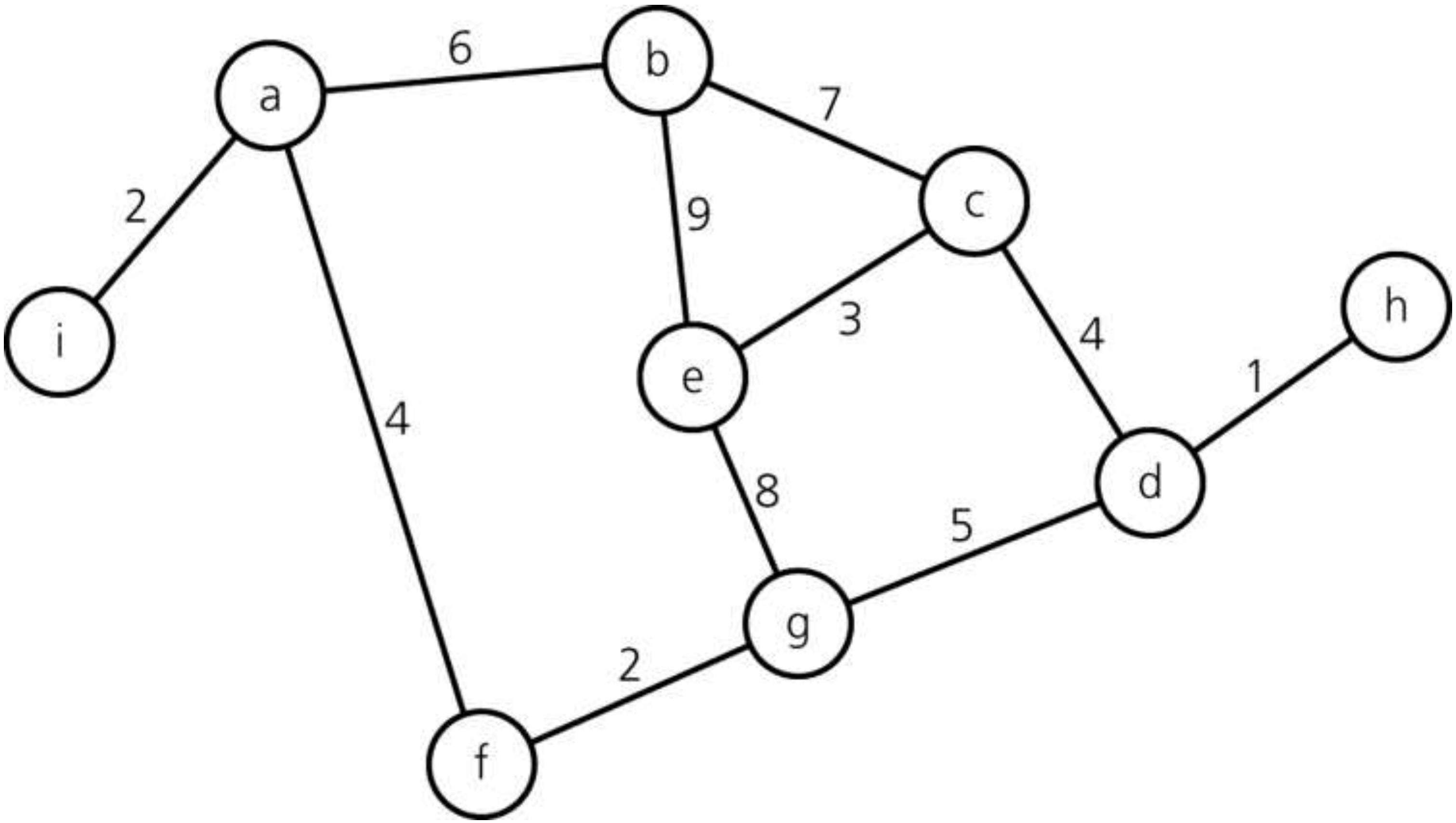
Prim’s Algorithm

- **Prim’s algorithm** finds a minimum spanning tree that begins any vertex.
- Initially, the tree contains only the starting vertex.
- At each stage, the algorithm selects a least-cost edge from among those that begin with a vertex in the tree and end with a vertex not in the tree.
- The selected vertex and least-cost edge are added to the tree.

Prim's Algorithm

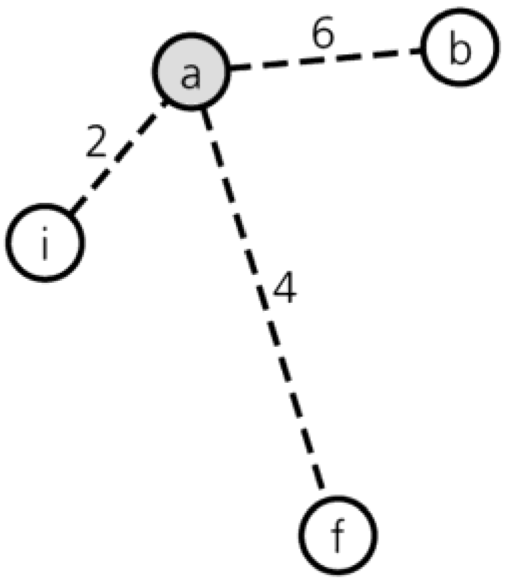
```
primsAlgorithm(in v:Vertex) {
// Determines a minimum spanning tree for a weighted,
// connected, undirected graph whose weights are
// nonnegative, beginning with any vertex.
   Mark vertex v as visited and include it in
      the minimum spanning tree;
   while (there are unvisited vertices) {
      Find the least-cost edge (v,u) from a visited vertex v
         to some unvisited vertex u;
      Mark u as visited;
      Add the vertex u and the edge (v,u) to the minimum
         spanning tree;
   }
}
```

Prim's Algorithm – Trace



A weighted, connected, undirected graph

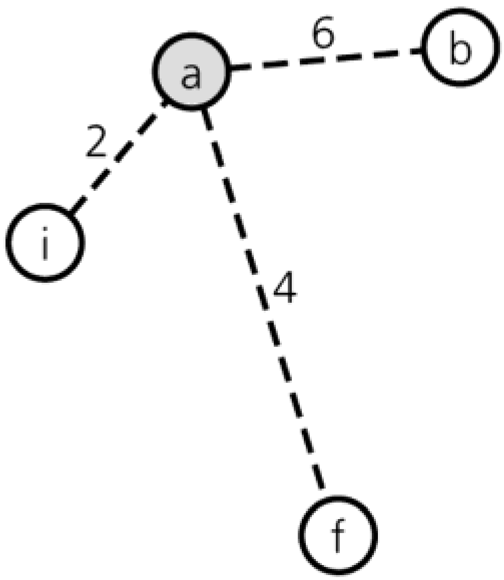
Prim's Algorithm – Trace (cont.)



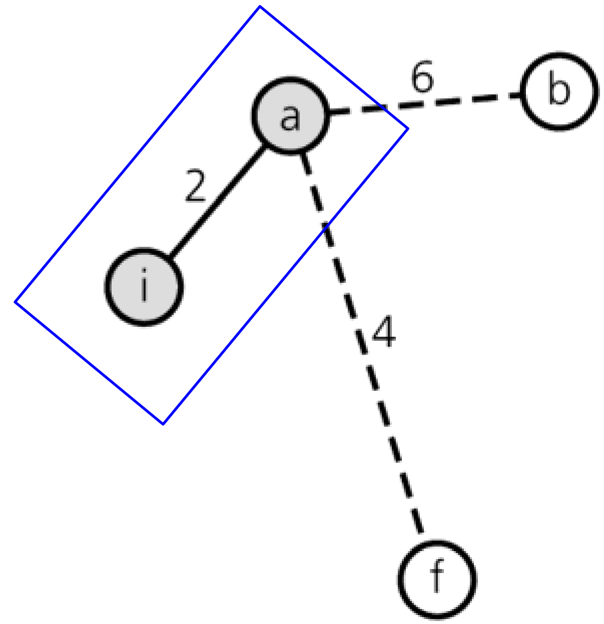
(a) Mark a, consider edges from a

beginning at vertex *a*

Prim’s Algorithm – Trace (cont.)



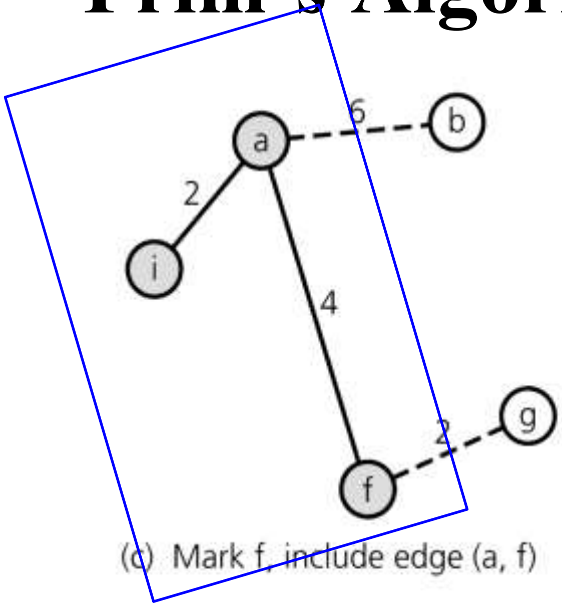
(a) Mark a, consider edges from a



(b) Mark i, include edge (a, i)

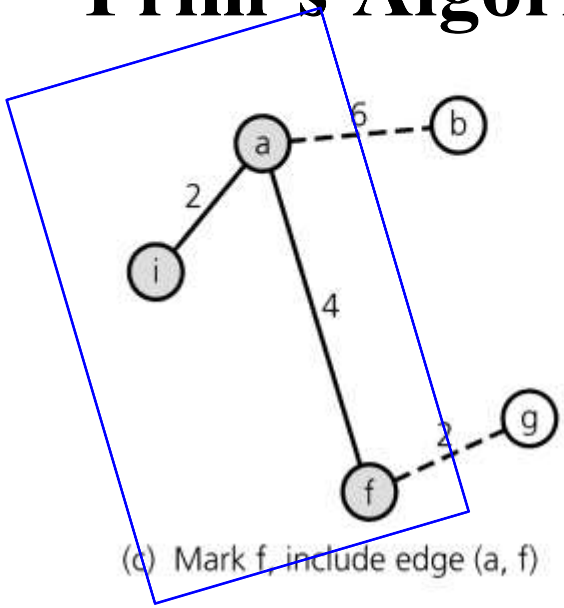
beginning at vertex *a*

Prim's Algorithm – Trace (cont.)

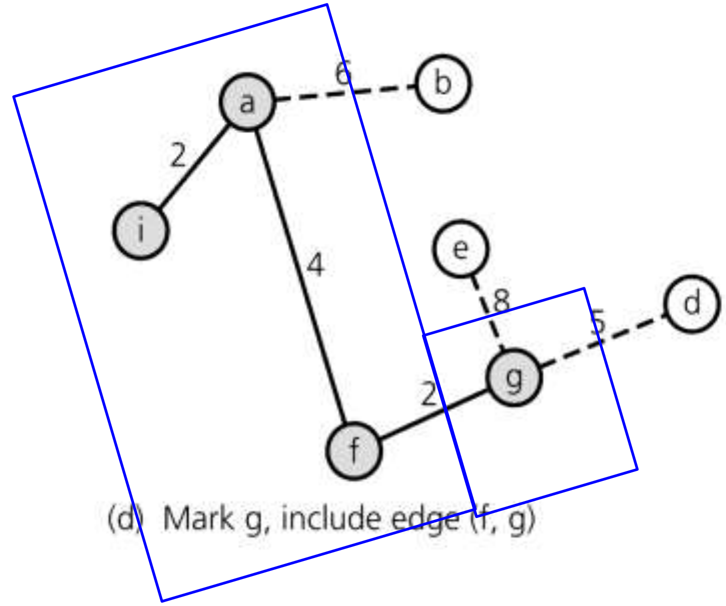


(c) Mark f, include edge (a, f)

Prim's Algorithm – Trace (cont.)

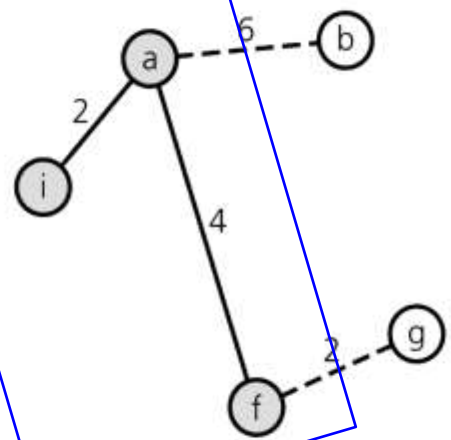


(c) Mark f, include edge (a, f)

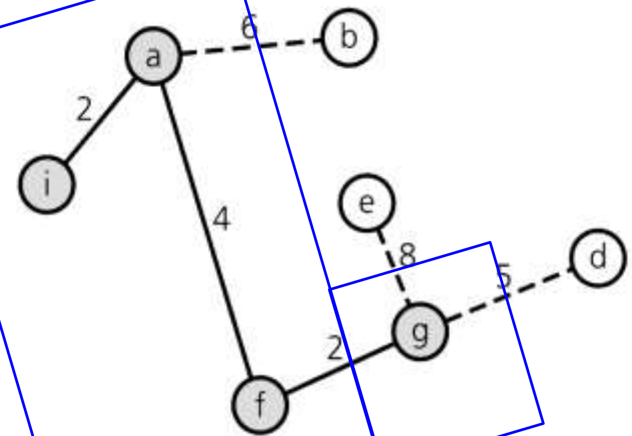


(d) Mark g, include edge (f, g)

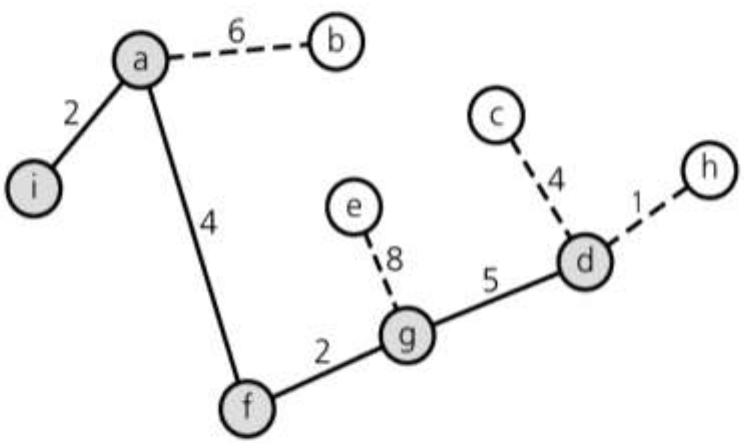
Prim’s Algorithm – Trace (cont.)



(c) Mark f, include edge (a, f)

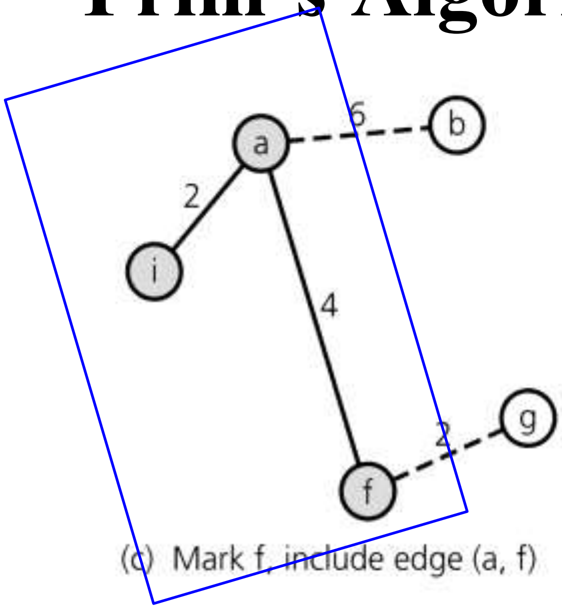


(d) Mark g, include edge (f, g)

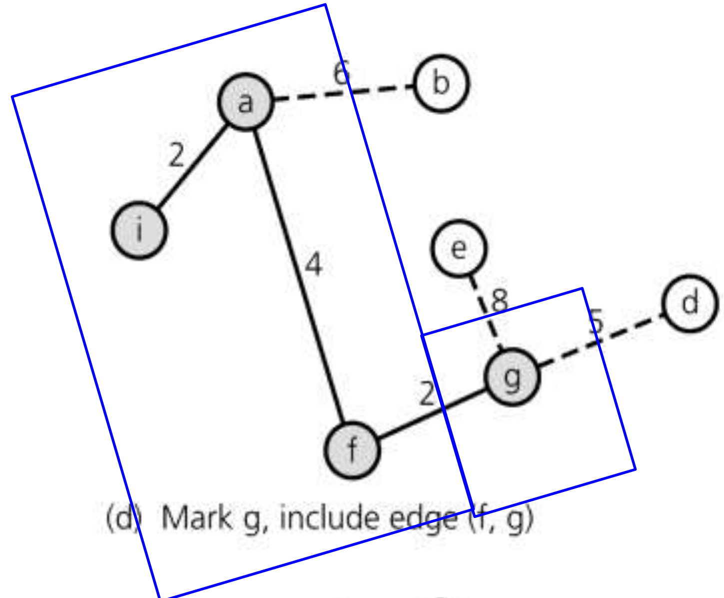


(e) Mark d, include edge (g, d)

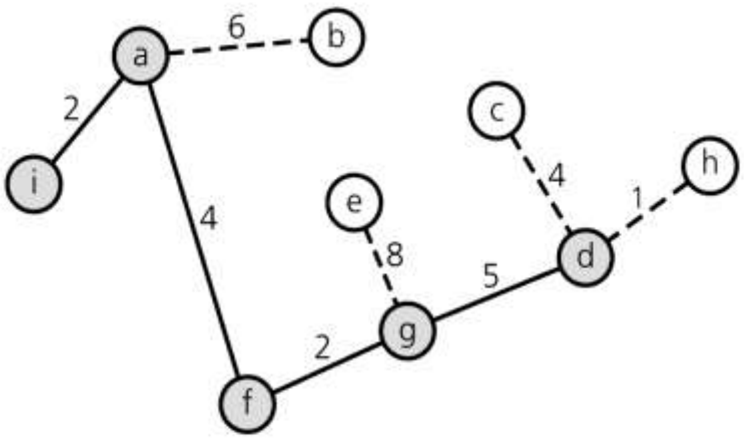
Prim’s Algorithm – Trace (cont.)



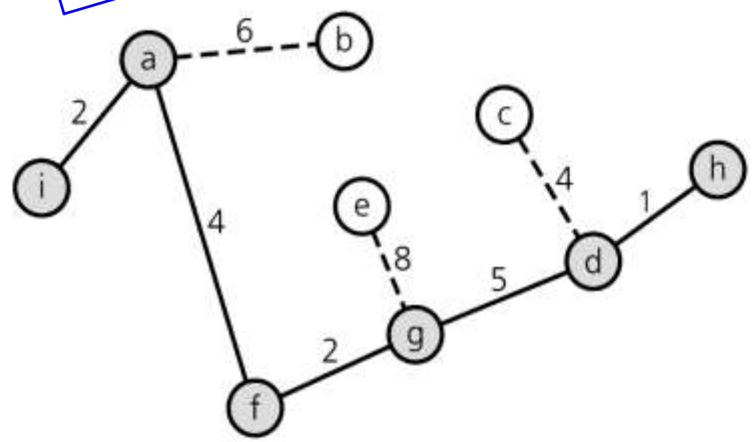
(c) Mark f, include edge (a, f)



(d) Mark g, include edge (f, g)

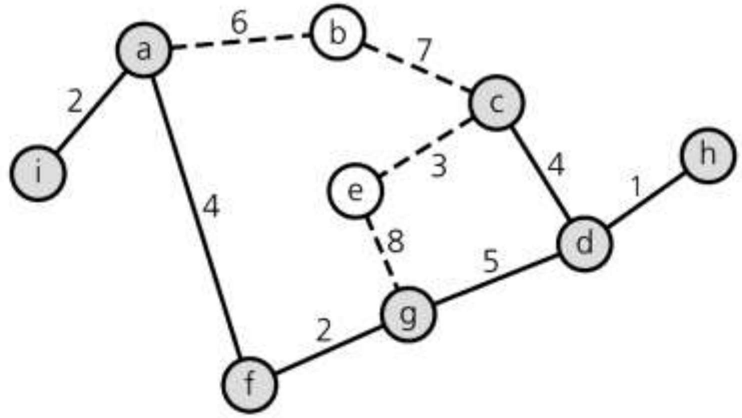


(e) Mark d, include edge (g, d)

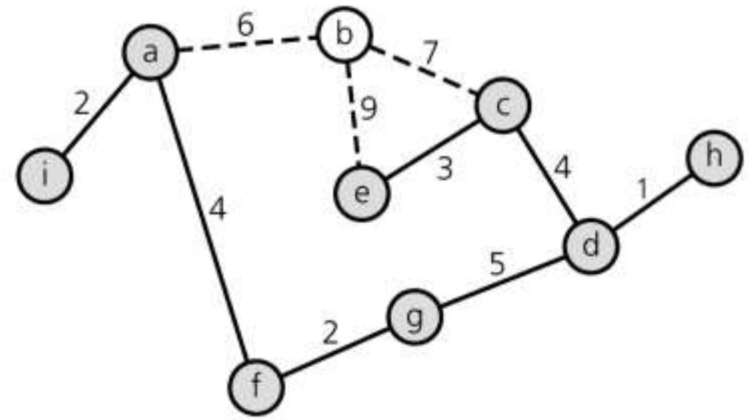


(f) Mark h, include edge (d, h)

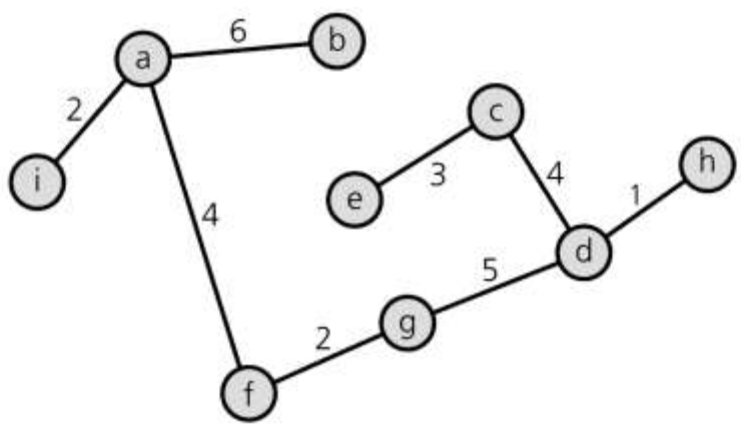
Prim’s Algorithm – Trace (cont.)



(g) Mark c, include edge (d, c)



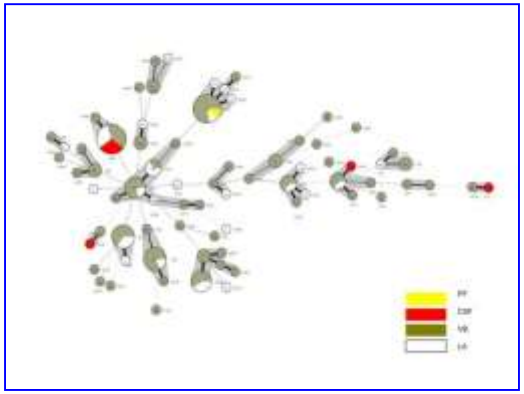
(h) Mark e, include edge (c, e)

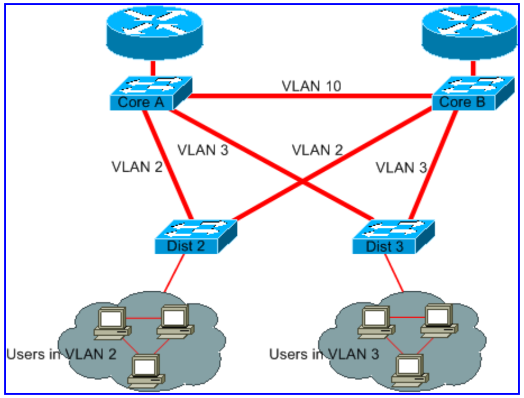


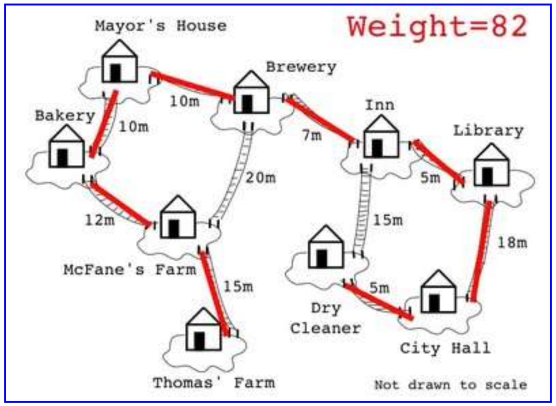
(i) Mark b, include edge (a, b)

Applications of Minimum Spanning Trees

- Building low-cost network
- Grouping objects into clusters, where each cluster is a set of similar objects.
- Minimum bottleneck







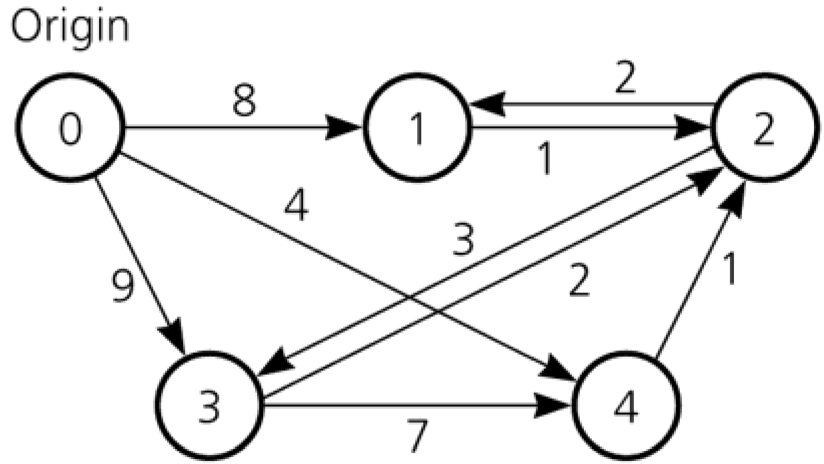
Shortest Paths

- The **shortest path** between two vertices in a weighted graph has the smallest edge-weight sum.

- **Dijkstra's shortest-path algorithm** finds the shortest paths between vertex 0 (a given vertex) and all other vertices.

Shortest Paths

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

A Weighted Directed Graph

Its Adjacency Matrix

Dijkstra's Shortest-Path Algorithm

```
shortestPath(in theGraph, in weight:WeightArray) {
// Finds the minimum-cost paths between an origin vertex (vertex 0)
// and all other vertices in a weighted directed graph theGraph;
// theGraph's weights are nonnegative
   Create a set vertexSet that contains only vertex 0;
   n = number of vertices in the Graph;
   // Step 1
   for (v=0 through n-1)
      weight[v] = matrix[0][v];
   // Steps 2 through n
   for (v=2 through n) {      //n-1 times v-->i
      Find the smallest weight[v] such that v is not in vertexSet;
      Add v to vertexSet;
      for (all vertices u adjacent to v but not in vertexSet)
         if (weight[u] > weight[v]+matrix[v][u])
            weigth[u] = weight[v]+matrix[v][u];
   }
}
```

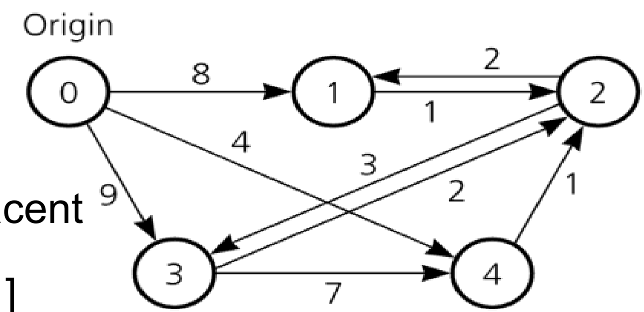
Dijkstra's Shortest-Path Algorithm – Trace

Step	v	vertexSet	weight [0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

min = 4

[2] is adjacent to
[4] check
inf > 4 + 1 update
[1] and [3] are adjacent
to [2] check
8 > 5 + 2 update [1]
9 > 5 + 3 update [3]
[2] is adjacent to [1]
check 7 + 1 > 5 dont update
etc.

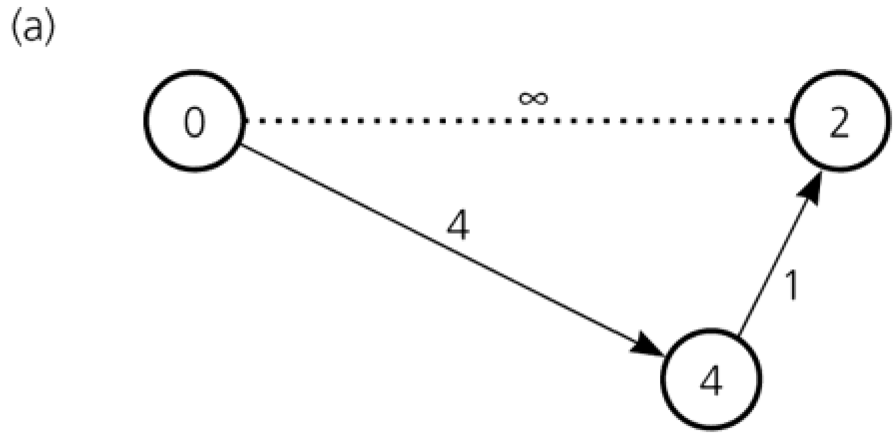
(a)



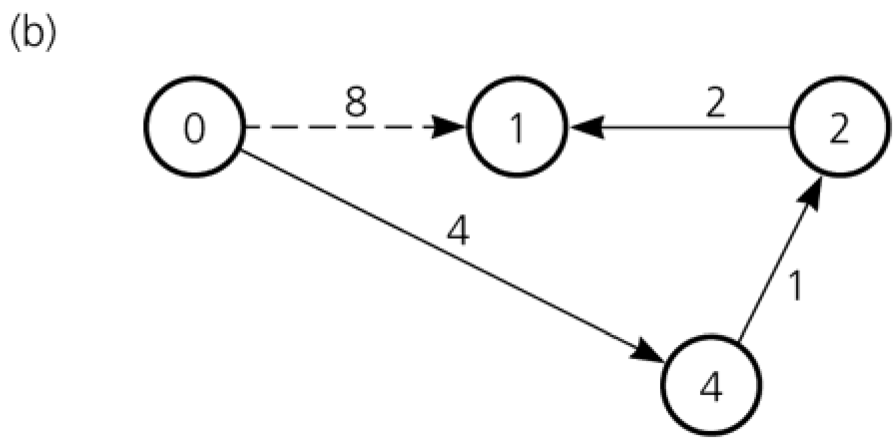
(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Dijkstra's Shortest-Path Algorithm – Trace (cont.)



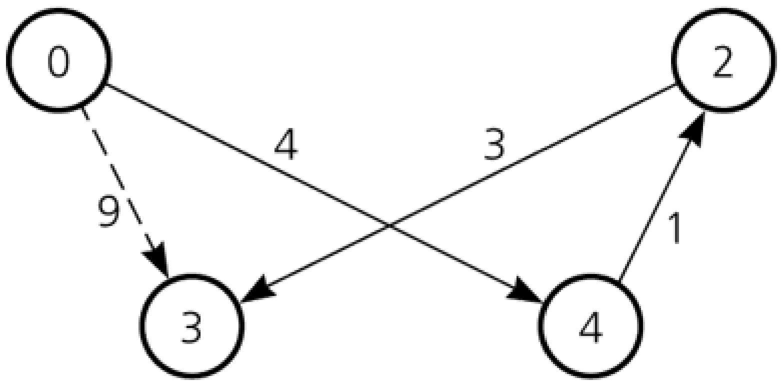
Step 2. The path 0–4–2 is shorter than 0–2



Step 3. The path 0–4–2–1 is shorter than 0–1

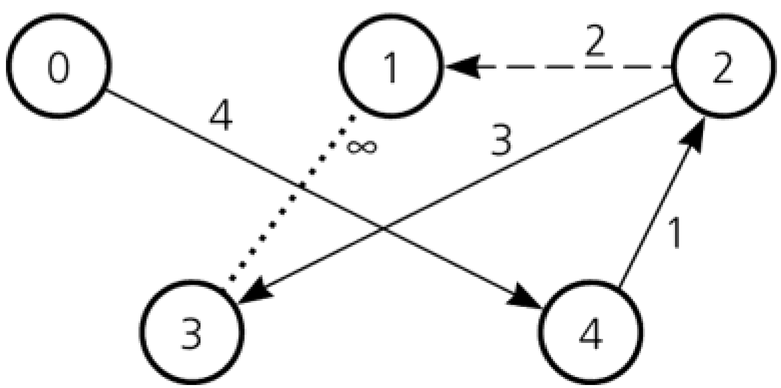
Dijkstra's Shortest-Path Algorithm – Trace (cont.)

(c)



Step 3 continued. The path 0–4–2–3 is shorter than 0–3

(d)



Step 4. The path 0–4–2–3 is shorter than 0–4–2–1–3