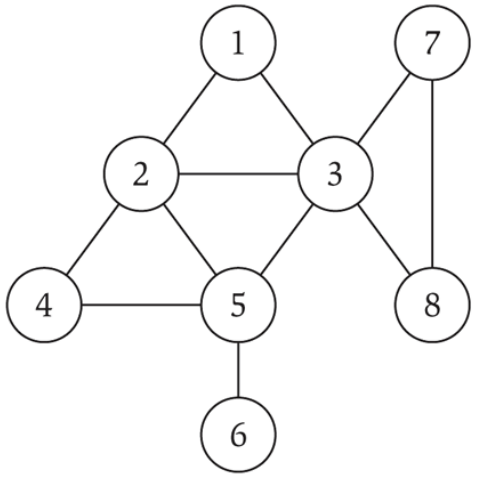
Graphs

Undirected Graphs

- Undirected graph. G = (V, E)
  - V = nodes.
  - E = edges between pairs of nodes.
  - Captures pairwise relationship between objects.
  - Graph size parameters: n = |V|, m = |E|.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

m = 11

Graphs

- **Definition (Graph):**

A graph is a tuple G(V,E) where V is a nonempty set of vertices (or nodes) and E is a set of edges. Each edge has either one or two vertices as endpoints, that is, each edge is either a one or two element subset of V .

The vertex set V of a graph G may be infinite.

Graphs

- For the graph G = (V ,E) and v $\in$ V , the edge e = {v} is called a self-loop .

- The number of vertices of a graph (| V |) is called its **order**, and the number of its edges (| E|) is called its **size.**

- **Definition (Vertex Adjacency)** Let G(V,E) be a graph. Two vertices v_1 and v_2 are said to be adjacent if there exists an edge e $\in$ E that connects them so that e = {v1, v2}.

Graphs

- **Definition (Edge Adjacency)** For a graph G(V,E), two edges e_1 and e_2 are said to be adjacent if there exists a vertex v that is incident to (connects) both edges.

- **Definition (Neighborhood)** Given G(V,E), the neighborhood of a vertex $v \in V$ is the set of vertices that are adjacent to v.
  - Formally, $N(v) = \{ u \in V : e(u, v) \in E \}$.

- **N(v)** is usually called the open neighborhood of v, whereas **N[v] = N(v)** $\cup$ **{v}** is called the closed neighborhood of v.

Graphs

- **Definition (Degree)** The degree of $v \in V$, deg(v), is the number of edges plus twice the number of self-loop edges incident to v.

- The maximum degree of a graph is denoted by $\Delta(G)$, and the minimum degree by $\delta(G)$.

- **Definition (Directed Graph)** A directed graph (digraph) G(V,E) consists of a nonempty set of vertices V and a set of directed edges E where each $e \in E$ is associated with an ordered set of vertices.

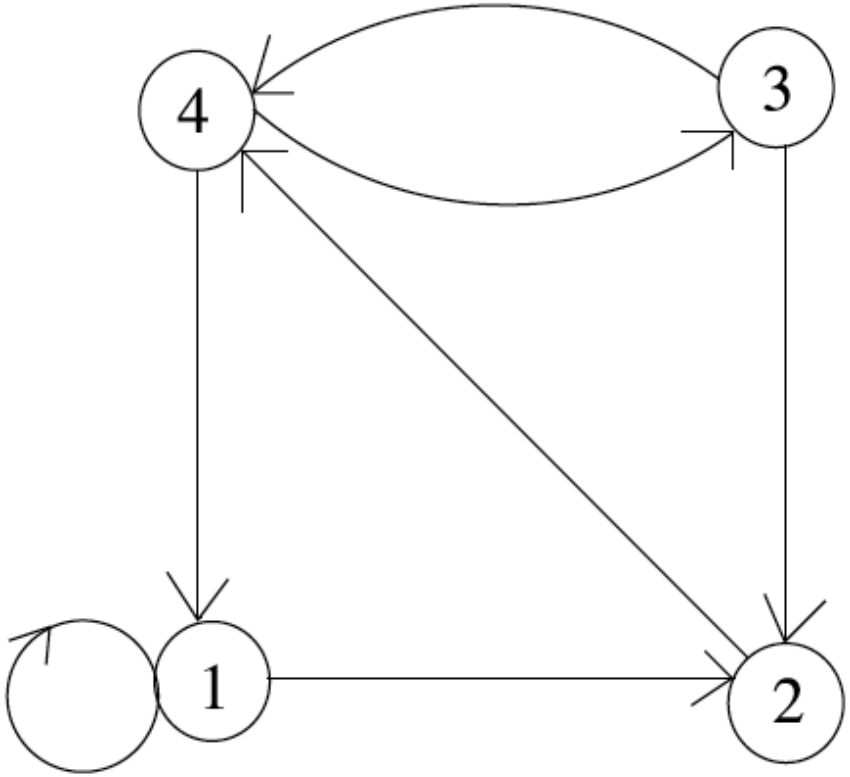


Figure shows a digraph with V = {1 , 2 , 3 , 4} and
E = {(1 , 1) , (1 , 2) , (2 , 4) , (3 , 2) , (3 , 4) , (4 , 3) , (4 , 1)}.

Graphs

- **Definition (In-Degree, Out-Degree)** The in-degree of a vertex v in a digraph G is the total number of edges in E that end at v. The out-degree of v is the total number of edges in E that start from v.

  - We will denote the in-degree of v by $\mathbf{deg_{in}(v)}$ and the out-degree by $\mathbf{deg_{out}(v)}$.

Special Graphs

- Complete graph,
- Bipartite graph,
- The complement of a graph,

Special Graphs

- **Definition (Complete Graph)**

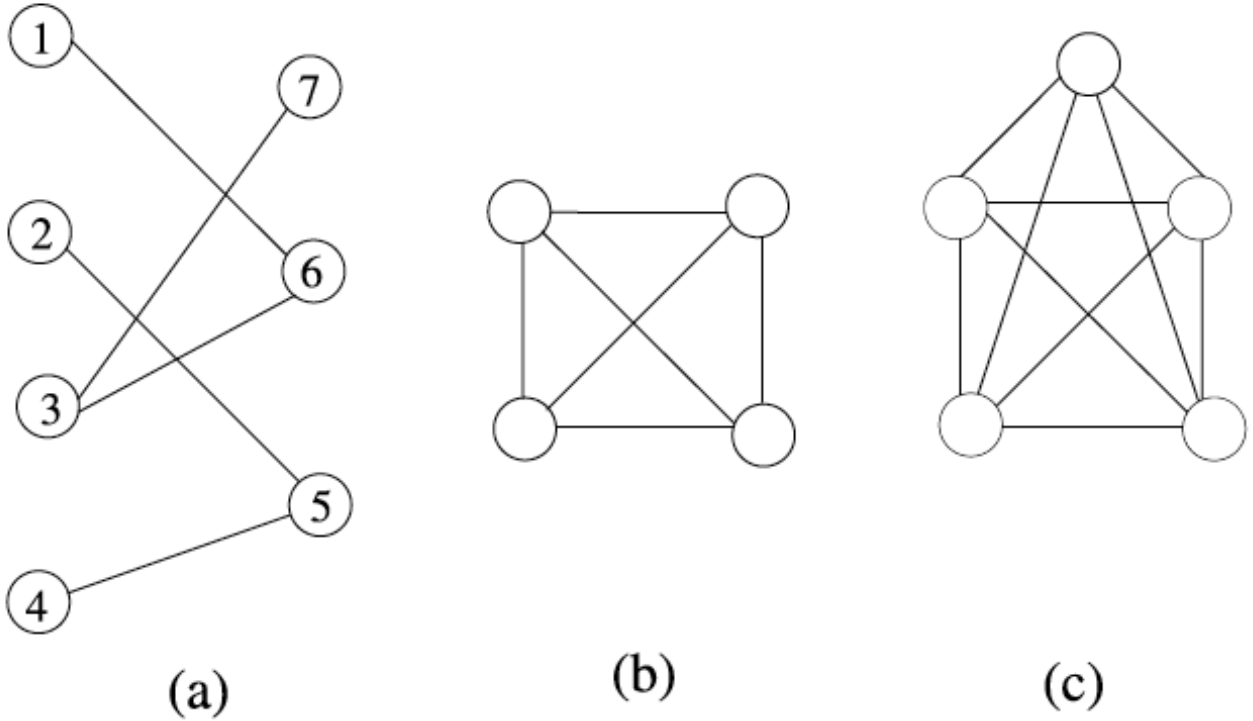
  For the graph G(V,E), if $\forall v \in V$, N(v) = V \ {v}, that is, if every vertex is connected to all other vertices of G, then G is called a complete graph.

  For a graph G with n vertices, the complete graph is denoted by K_n.

  For $K_n(V ,E)$, $|E| = n(n -1)/2$.

- **Definition (Bipartite Graphs)**
- A graph G(V,E) is called bipartite if V can be partitioned into two disjoint sets V_1 and V_2 such that every edge of G joins a vertex in V_1 to a vertex in V_2.

Special Graphs

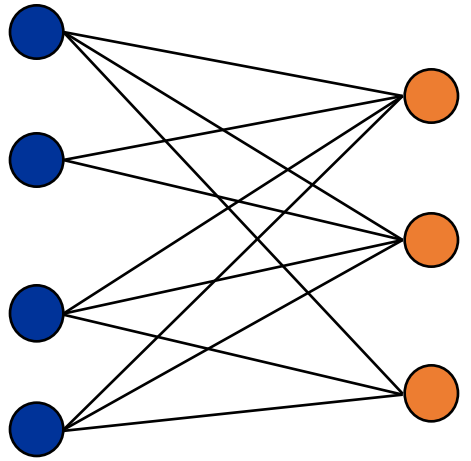


A bipartite graph with $V_1 = \{ 1, 2, 3, 4\}$ and $V_2 = \{ 5, 6, 7\}$ is shown in Fig. (a), and K_4 and K_5 are shown in Fig. (b) and (c).

Bipartite Graphs

- **Def.** An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

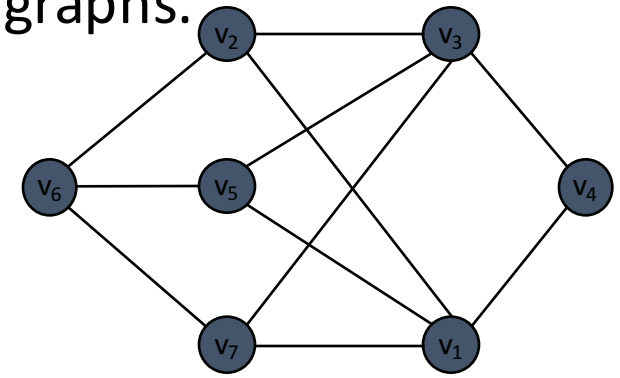
- Applications.
  - Stable marriage: men = red, women = blue.
  - Scheduling: machines = red, jobs = blue.



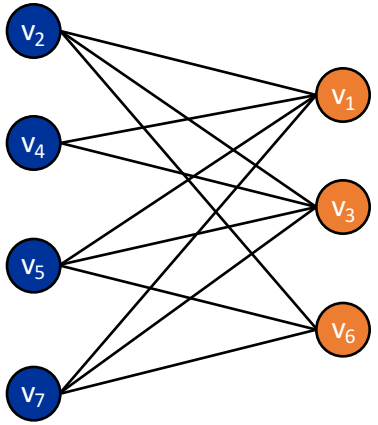
a bipartite graph

Testing Bipartiteness

- Testing bipartiteness. Given a graph G, is it bipartite?
  - Many graph problems become:
    - easier if the underlying graph is bipartite (matching)
    - tractable if the underlying graph is bipartite (independent set)
  - Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

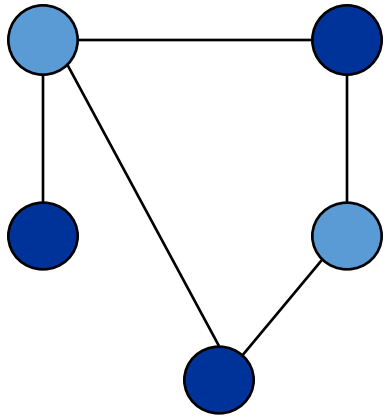


another drawing of G

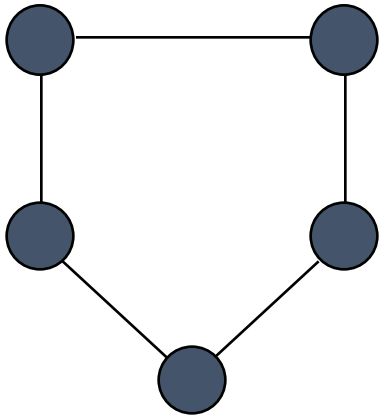
An Obstruction to Bipartiteness

- Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

- Pf. Not possible to 2-color the odd cycle, let alone G.



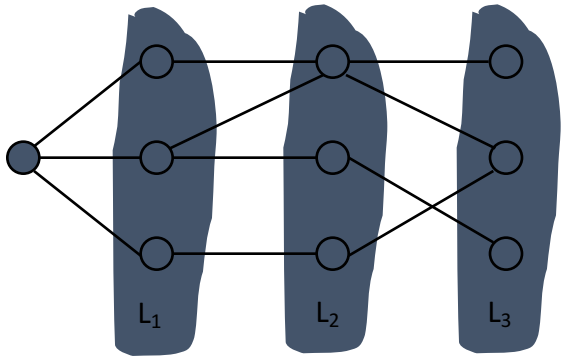
bipartite
(2-colorable)



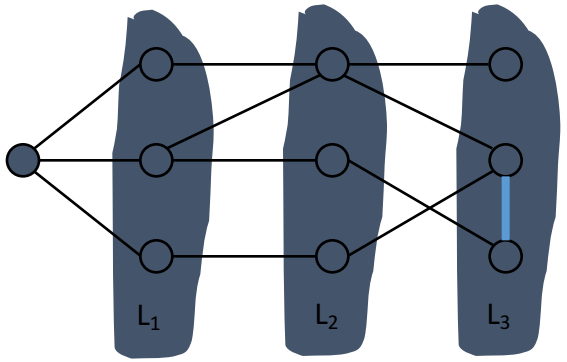
not bipartite
(not 2-colorable)

Bipartite Graphs

- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.
  - (i) No edge of G joins two nodes of the same layer, and G is bipartite.
  - (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



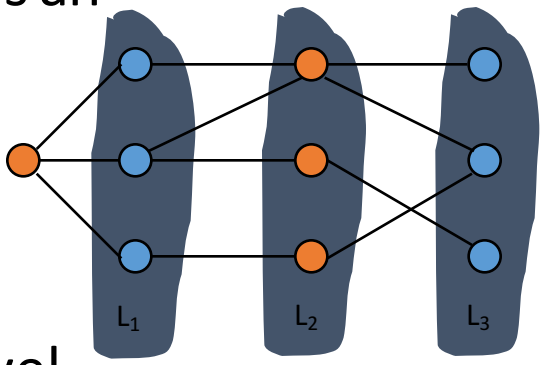
Case (i)



Case (ii)

Bipartite Graphs

- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.
  (i) No edge of G joins two nodes of the same layer, and G is bipartite.
  (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



- Pf. (i)
  - Suppose no edge joins two nodes in the same layer.
  - By previous lemma, this implies all edges join nodes on same level.
  - Bipartition: red = nodes on odd levels, blue = nodes on even levels.

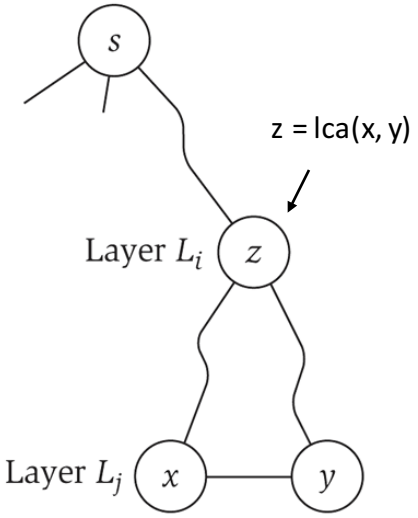
Case (i)

Bipartite Graphs

- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.
  (i) No edge of G joins two nodes of the same layer, and G is bipartite.
  (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

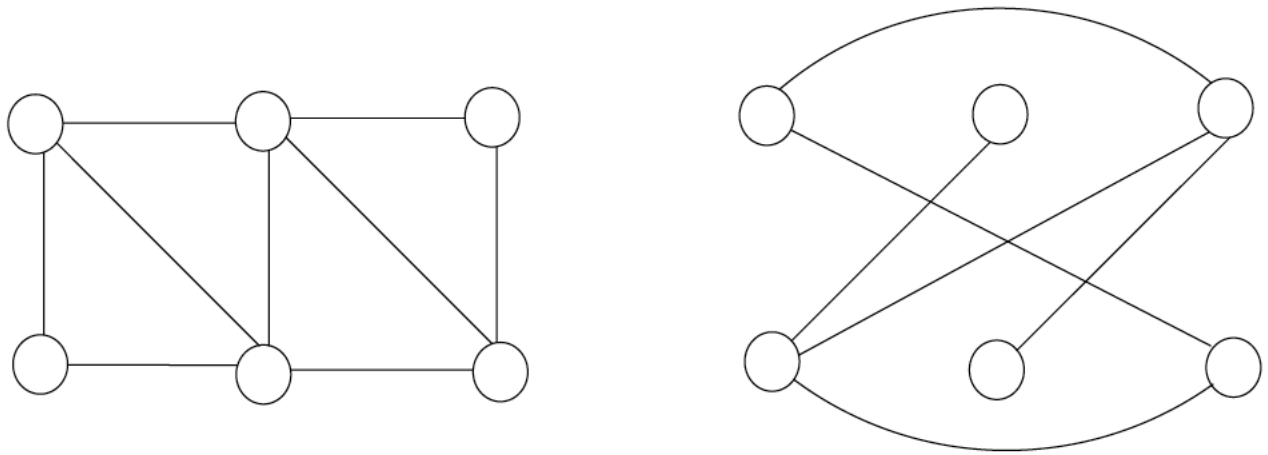
- Pf. (ii)
  - Suppose (x, y) is an edge with x, y in same level L_j.
  - Let z = lca(x, y) = lowest common ancestor.
  - Let L_i be level containing z.
  - Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
  - Its length is 1 + (j-i) + (j-i), which is odd. ▪

(x, y) path from y to z path from z to x



Special Graphs

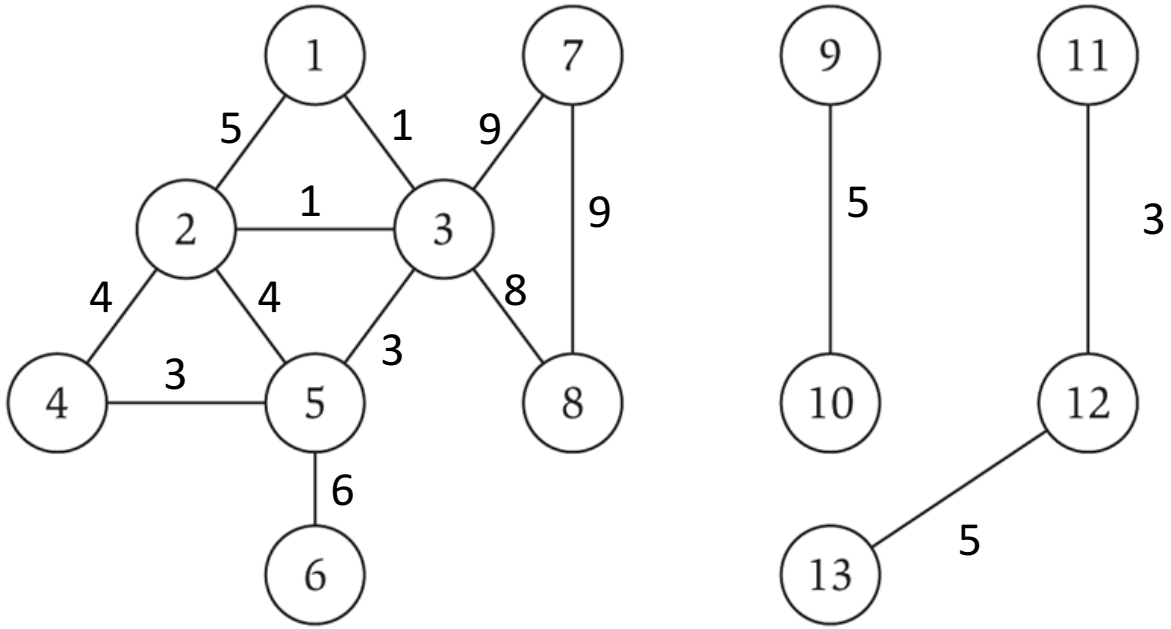
- **Definition (Complement of a Graph)** The complement of a graph G(V,E) is the graph H(V,E') such that e = {v1, v2} ∈ E'
- if and only if e = $\{v_1, v_2\} \notin E$.
- The complement of G is denoted G' or $\overline{G}$.



(a) (b)

Graphs

- **Definition (Weighted Graphs) .** A weighted graph G(V,E,w) is a graph that has weights associated with edges, that is, $w : E \rightarrow R$.



Graph Representations

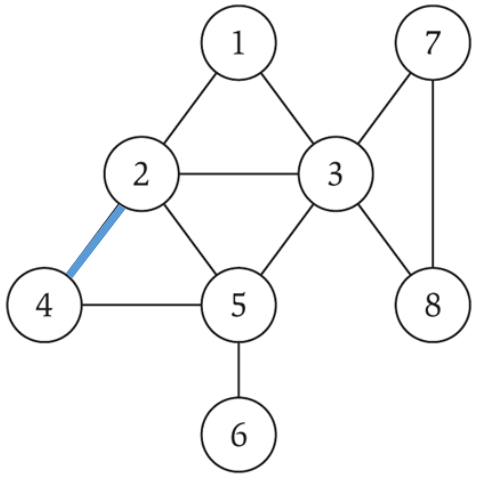
- **Definition (Adjacency Matrix)** The adjacency matrix of a graph G(V,E) with n vertices is an n x n matrix which has entry 1 at element (i, j) if there is an edge connecting vertex i to vertex j and 0 otherwise.

- **Definition (Incidence Matrix)** The incidence matrix of a graph G(V,E) with n vertices and m edges is an n x m matrix which has entry 1 at element (i, j) if vertex i is incident to edge j and 0 otherwise.

- **Definition (Adjacency List)** The adjacency list of a graph G(V,E) with n vertices is a list of n elements where each element consists of a vertex $v \in V$ and its neighbors connected using linked lists.

Graph Representation: Adjacency Matrix

- Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.
  - Two representations of each edge.
  - Space proportional to n^2.
  - Checking if (u, v) is an edge takes $\Theta(1)$ time.
  - Identifying all edges takes $\Theta(n^2)$ time.

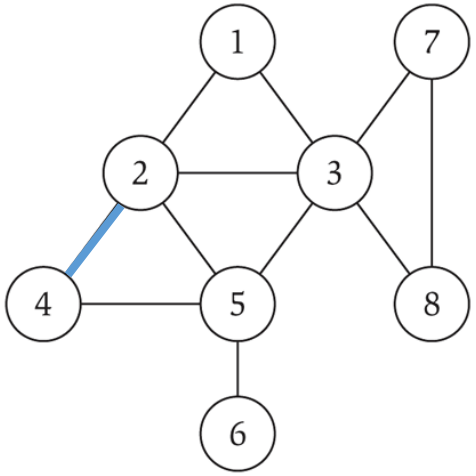


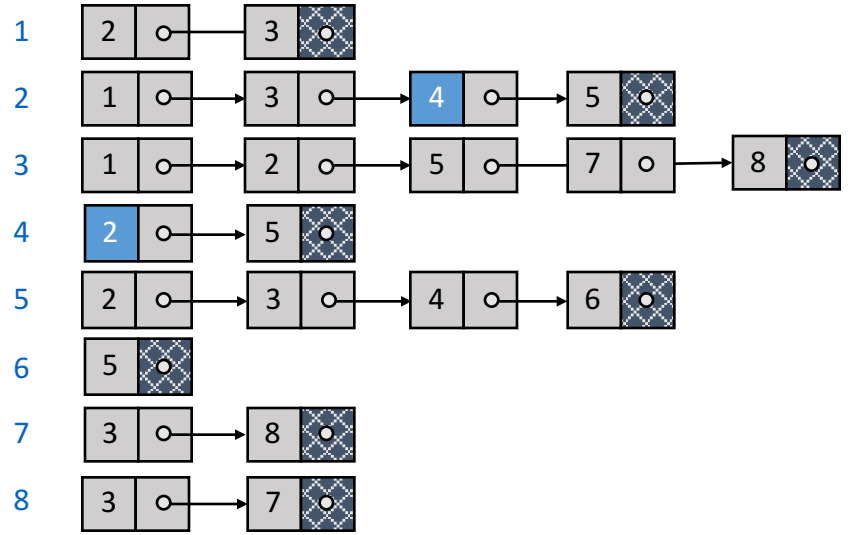
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Representation: Adjacency List

- Adjacency list. Node indexed array of lists.
  - Two representations of each edge.
  - Space proportional to m + n.
  - Checking if (u, v) is an edge takes O(deg(u)) time.
  - Identifying all edges takes $\Theta(m + n)$ time.

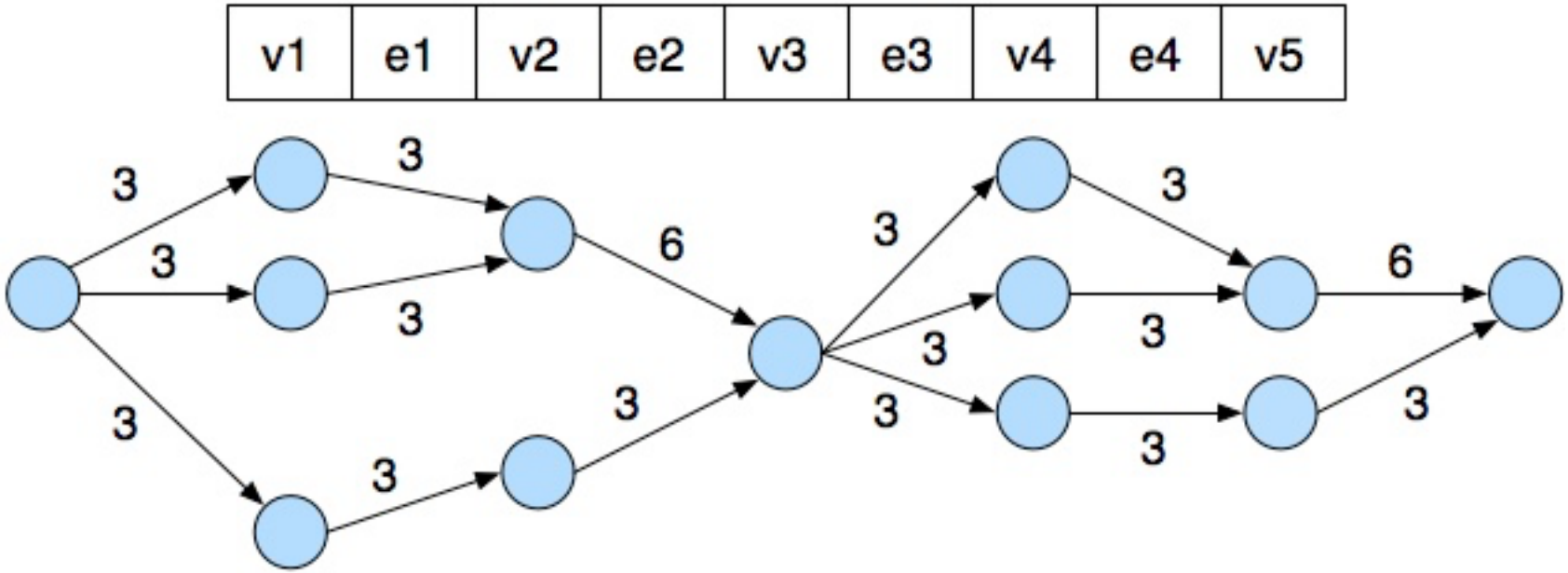
degree = number of neighbors of u





Walks, Paths, Cycles

- **Definition (Walk)** A walk $w = (v_1, e_1, v_2, e_2, \ldots, v_x, e_x, v_{x+1})$ in G is an alternating sequence of vertices and edges in V and E, respectively, such that for all $i = 1, \ldots, n$, $\{v_i, v_{i+1}\} = e_i$.
- A walk is called closed if $v_1 = v_{n+1}$ and open otherwise.



Walks, Paths, Cycles

- **Definition (Trail, Tour)** A trail in G is a walk in G where no edge is repeated and, **a tour is a closed trail**. An **Eulerian trail** is a trail that contains exactly one copy of each edge in E, and an **Eulerian tour** is a closed trail (tour) that contains exactly one copy of each edge.

- **Definition (Path)** A path p from a vertex u to vertex v in graph G is a sequence of edges e_1, . . . , ex such that each consecutive edge is incident to consecutive vertices along the path. The length of p is the number of edges it contains. When G is simple, a path can be represented by the set of vertices v_1, . . . , v_x that it passes through (traverses).

Walks, Paths, Cycles

- The path is called a circuit if it starts and ends at the same vertex.
- A **Hamiltonian Path** is a path that contains each vertex in V once.

- **Definition (Cycle)** A cycle is a circuit of length of at least 3 and with no repeated edges except the first and last vertices. A Hamiltonian cycle is a cycle in a graph containing every vertex.

- **Definition (Hamiltonian/Eulerian Graph)** A graph G = (V ,E) is said to be Hamiltonian if it contains a Hamiltonian cycle and Eulerian if it contains an Eulerian tour.

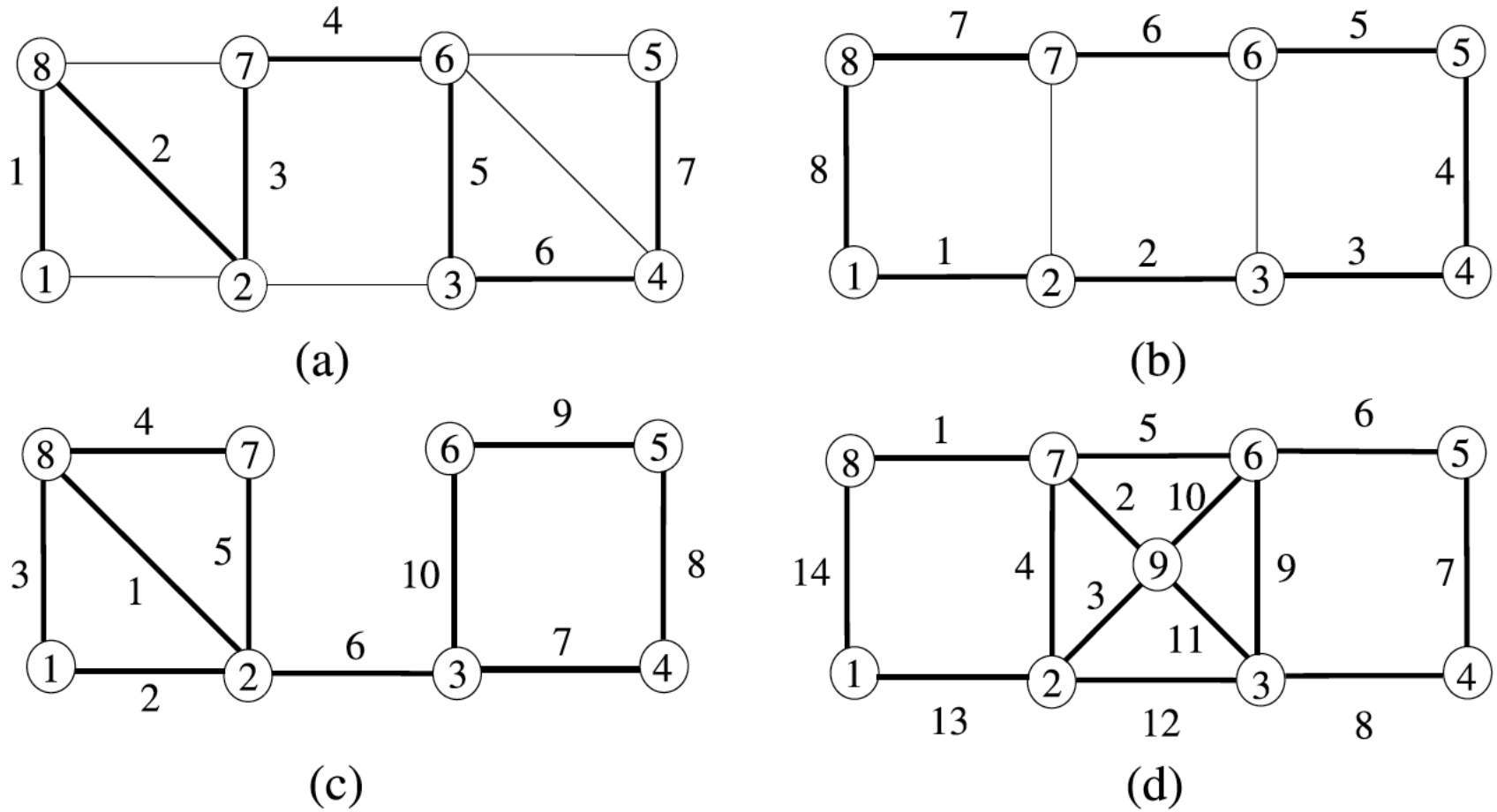


Fig. **(a)** A Hamiltonian trail through vertices 1, 8, 2, 7, 6, 3, 4, 5. **(b)** A Hamiltonian tour through vertices 1, 2, 3, 4, 5, 6, 7, 8, 1. **(c)** An Eulerian trail through vertices 8, 2, 1, 8, 7, 2, 3, 4, 5, 6, 3. **(d)** An Eulerian tour through vertices 8, 7, 9, 2, 7, 6, 5, 4, 3, 6, 9, 3, 2, 1, 8, all shown by bold lines and each edge labeled in sequence

Diameter, Radius, Circumference, and Girth

- **Definition (Distance)** For a graph G(V,E), the distance between the two vertices v_1 and v_2 in V is the length of the shortest path beginning at v_1 and ending at v_2, provided that such a walk exists. We will write $d_G(v_1, v_2)$ to denote the distance between v_1 and v_2 in G.

- **Definition (Diameter, Eccentricity, Radius)** The diameter of G (diam(G)) is the length of the greatest distance in G. The eccentricity of v_1 is the maximum distance from v_1 to any other vertex v_2 in V. The radius of G is the minimum eccentricity of vertices of G .

Diameter, Radius, Circumference, and Girth

- **Definition (Girth)** For a graph G(V,E), the girth of G is the length of the shortest cycle, provided that there is a cycle. When G does not have any cycle, the girth is defined as 0.

- **Definition (Circumference)** For a graph G(V,E), the circumference of G is the length of the longest cycle, provided that there is a cycle in G.When G does not have any cycle, the circumference is defined as ∞.

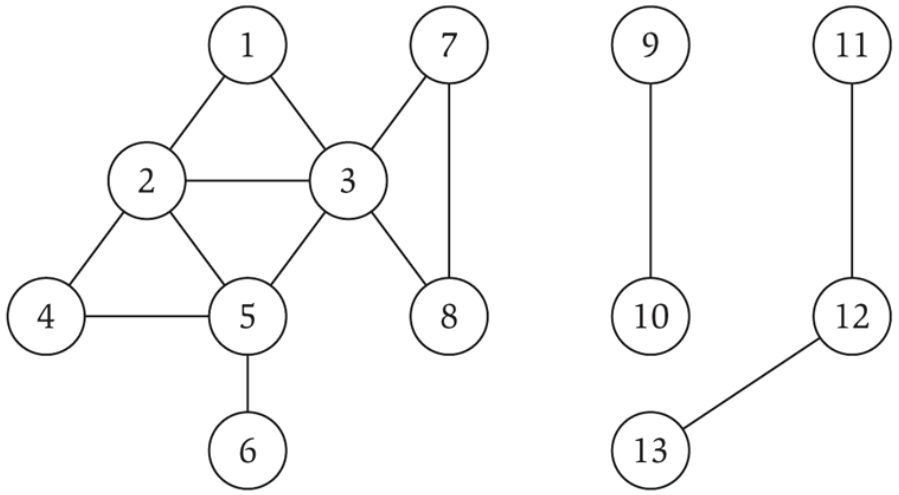
Connectivity

- **Definition (Connectedness)** A graph G(V,E) is connected if there is a walk between any pair of vertices v_1 and v_2. A digraph G is strongly connected if for every walk from every vertex $v_1 \in V$ to any vertex $v_2 \in V$, there is also a walk from v_2 to v_1.

- **Definition (Component)** A component of a graph G(V,E) is a subgraph G of G where any pair of vertices in G is connected. A connected graph G has only one component which is itself.

Connected Component

- Connected component. Find all nodes reachable from s.



- Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

- **Definition (Edge Deletion Graph)** For a graph G(V,E) and E’$\subset$ E, the graph G formed after deleting the edges in E from G is the subgraph induced by the edge set E \ E’, which is denoted G’ = G− E’

- **Definition (Vertex Deletion Graph)** For the graph G(V,E) and V’ $\subset$ V , the graph G formed after deleting the vertices in V from G is the subgraph induced by the vertex set V \ V’, which is denoted G = G−V’.

- **Definition (Cutpoint)** For a graph G(V,E), a vertex v $\in$ V is a cutpoint of G if G−v has more components than G has. If G is connected, G−v is disconnected.

- **Definition (Bridge, Cutset)** For a graph G(V,E), a bridge is an edge e $\in$ E deletion of which increases the number of components of G. A minimal set of edges whose deletion disconnects G is called a cutset in G.

- **Definition** A block of a graph G is its maximal subgraph that is connected and contains no cutpoints.

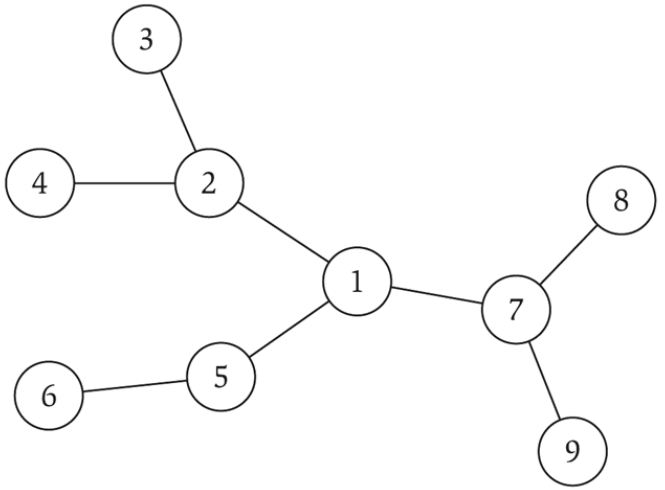
- **Definition (Connectivity)** The vertex connectivity (or just the connectivity) $\mathcal{K}$ of a graph G is the minimum number of vertices whose removal from G results in either a disconnected graph or a single vertex. The edge connectivity E(G) is defined as the minimum number of edges whose removal disconnects G.

Trees

- **Def.** An undirected graph is a tree if it is connected and does not contain a cycle.

- **Theorem**. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  - G is connected.
  - G does not contain a cycle.
  - G has n-1 edges.
  - G is connected, and each edge is a bridge;
  - Any two vertices of G are connected

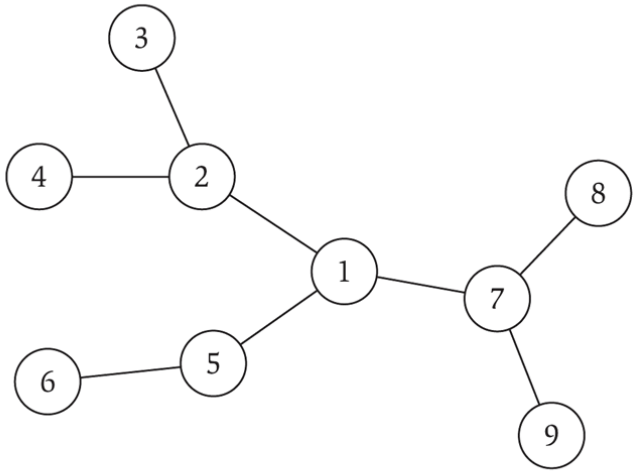
    by exactly one path;



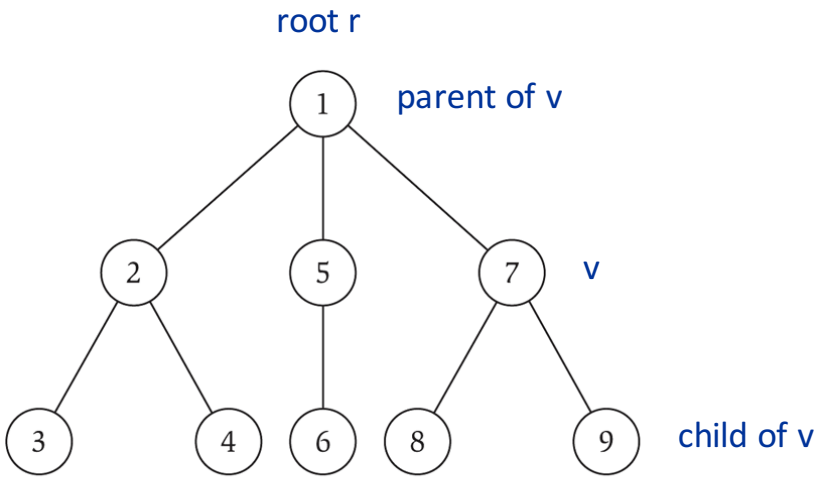
Rooted Trees

- Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.

- Importa



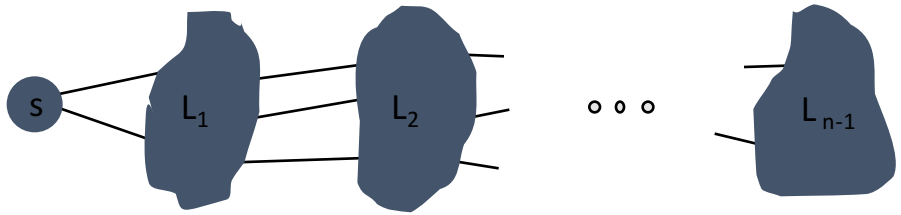
a tree



the same tree, rooted at 1

Breadth First Search

- BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



- BFS algorithm.
  - $L_0 = \{ s \}$.
  - L_1 = all neighbors of L_0.
  - L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
  - L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

- Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

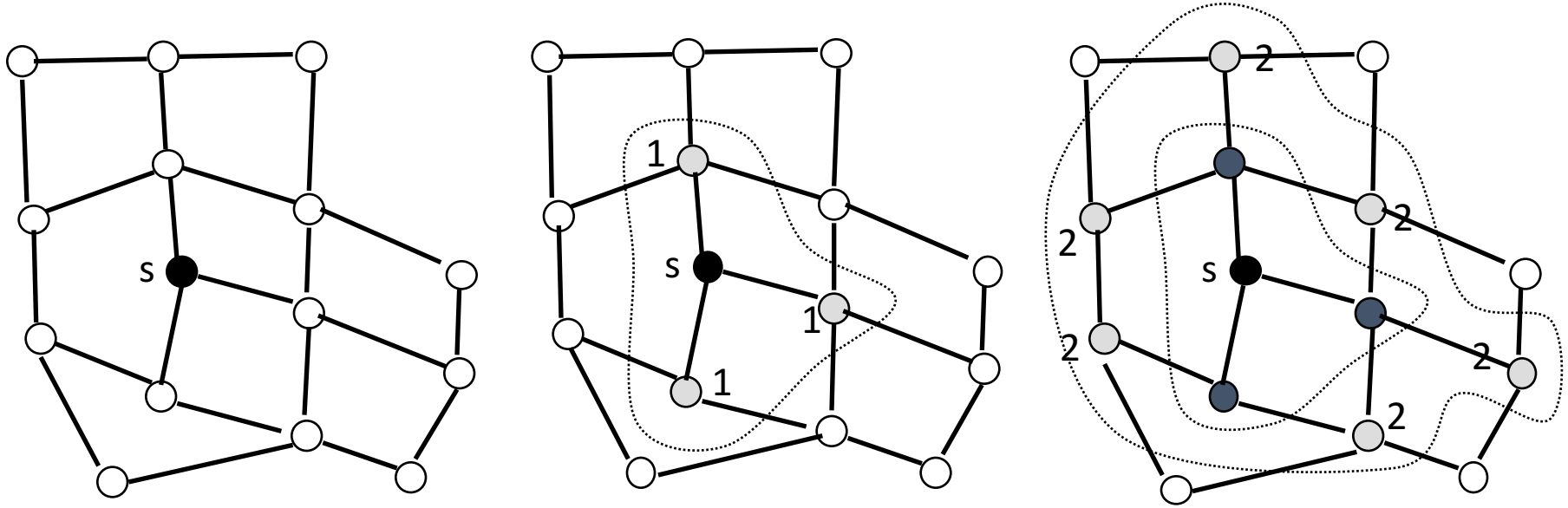
Breadth First Search Algorithm

```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = ∅
  While L[i] is not empty
    Initialize an empty list L[i + 1]
    For each node u ∈ L[i]
      Consider each edge (u, v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Add v to the list L[i + 1]
      Endif
    Endfor
    Increment the layer counter i by one
  Endwhile
```

BFS - Analysis

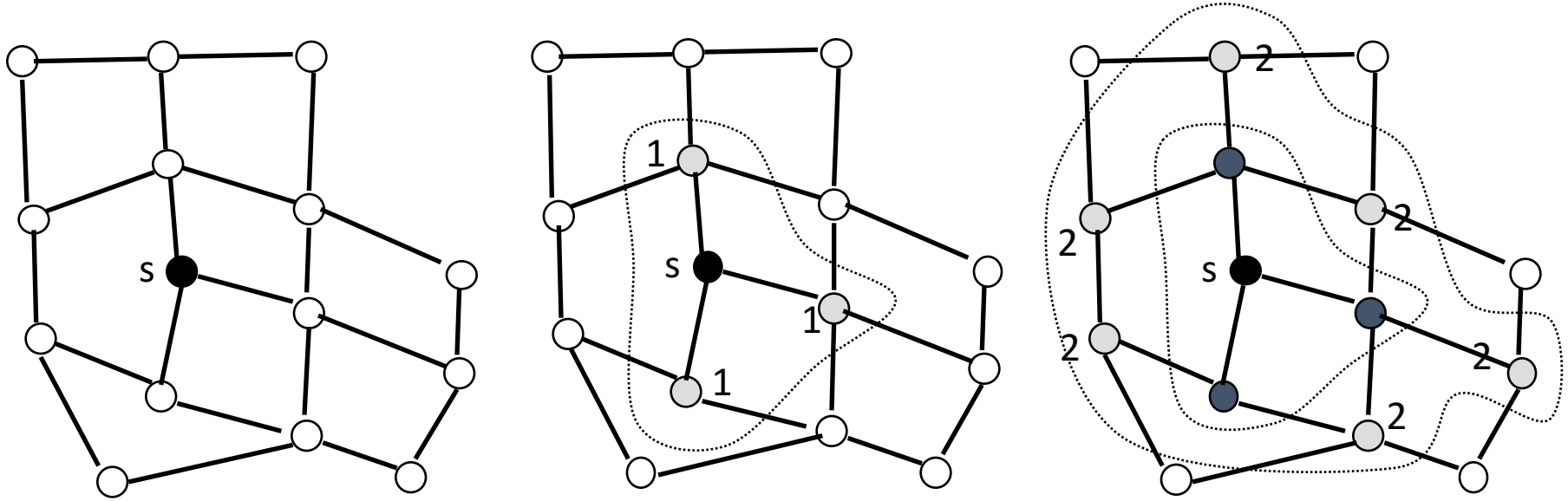
- Let n = |V| and e = |E|
- Observe that the initial portion requires $\theta(n)$
- The real meat is through the traversal loop
- Since we never visit a vertex twice, the number of times we go through the loop os at most n (exactly n assuming each vertex is reachable from the source)
- The number of iterations through the inner loop is proportional to deg(u)
- Summing up over all vertices we have
  - $T(n) = n + \text{Sum}_{v \varepsilon V} \deg[v] = n + 2e = \theta(n+e)$

Breadth-First Search - Idea



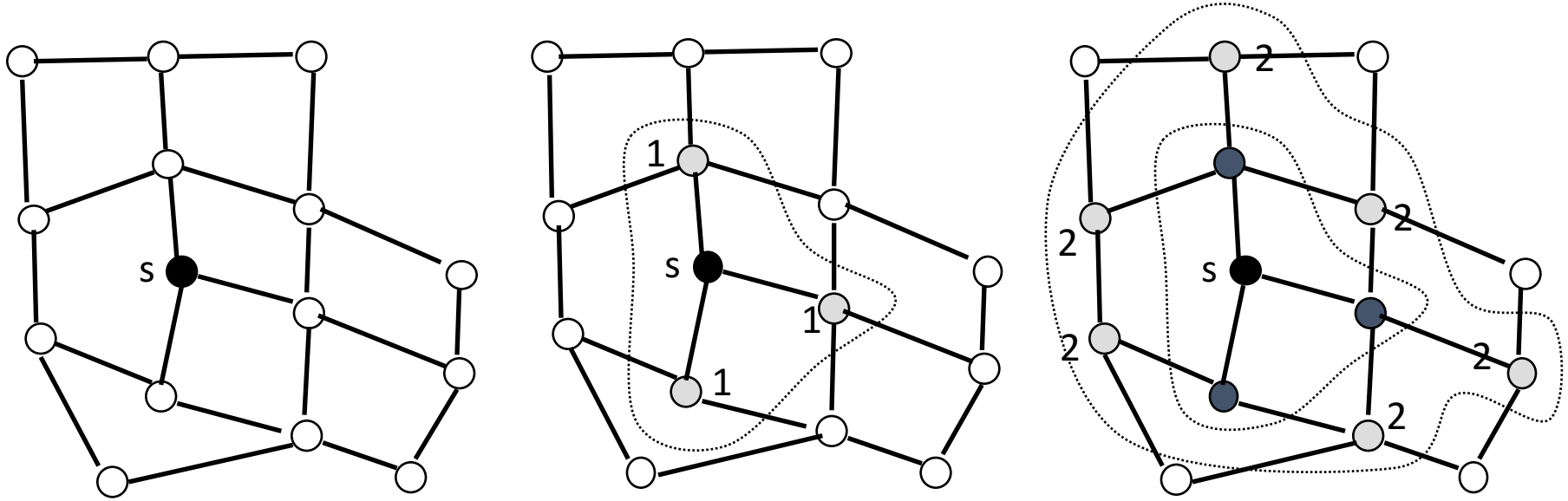
- Given a graph G = (V, E), start at the source vertex "s" and discover which vertices are reachable from s
  - At any time there is a "frontier" of vertices that have been discovered, but not yet processed
- Next pick the nodes in the frontier in sequence and discover their neighbors, forming a new "frontier"
  - Breadth-first search is so named because it visits vertices across the entire breadth of this frontier before moving on

BFS – Implementation



- Initially all vertices (except the source) is colored white, meaning they have not been discovered just yet
- When a vertex is first discovered, it is colored gray (and is part of the frontier)
- When a gray vertex is processed, it becomes black

BFS - Implementation



- The search makes use of a FIFO queue, Q
- We also maintain arrays
  - color[u], which holds the color of vertex u
    - either white, gray, black
  - pred[u], which points to the predecessor of u
    - The vertex that discovered u
  - d[u], the distance from s to u

BFS –Another way of Implemenattion

```
BFS(G, s){
  for each u in V- {s} {              // Initialization
     color[u] = white;
     d[u] = INFINITY;
     pred[u] = NULL;
   } //end-for

  color[s] = GRAY;                   // initialize source s
  d[s] = 0;
  pred[s] = NULL;
  Q = {s};                           // Put s in the queue
  while (Q is nonempty){
    u = Dequeue(Q);                  // u is the next vertex to visit

    for each v in Adj[u] {
      if (color[v] == white){        // if neighbor v undiscovered
          color[v] = gray;           // … mark is discovered
          d[v] = d[u] + 1;           // … set its distance
          pred[v] = u;               // … set its predecessor
          Enqueue(v);                //…  put it in the queue
      } //end-if
    } //end-for

    color[u] = black;                // we are done with u
  } //end-while
} //end-BFS
```

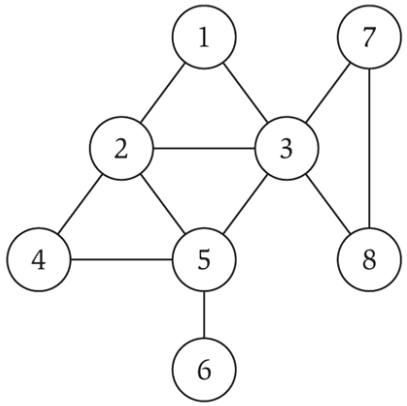
Depth First Search

- Consider searching the way out from a maze.
  - As you enter a location of the maze, paint some graffiti on the wall to remind yourself that you were there
  - Successively travel from room to room as long as you come to a place you have not already been
  - When you return to the same room, try a different door (assuming it goes somewhere you have not been before)
  - When all doors have been tried in a room, backtrack

  - Same idea in trying to find a door out of a puzzle

DFS Algorithm Recursive Version 1

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```



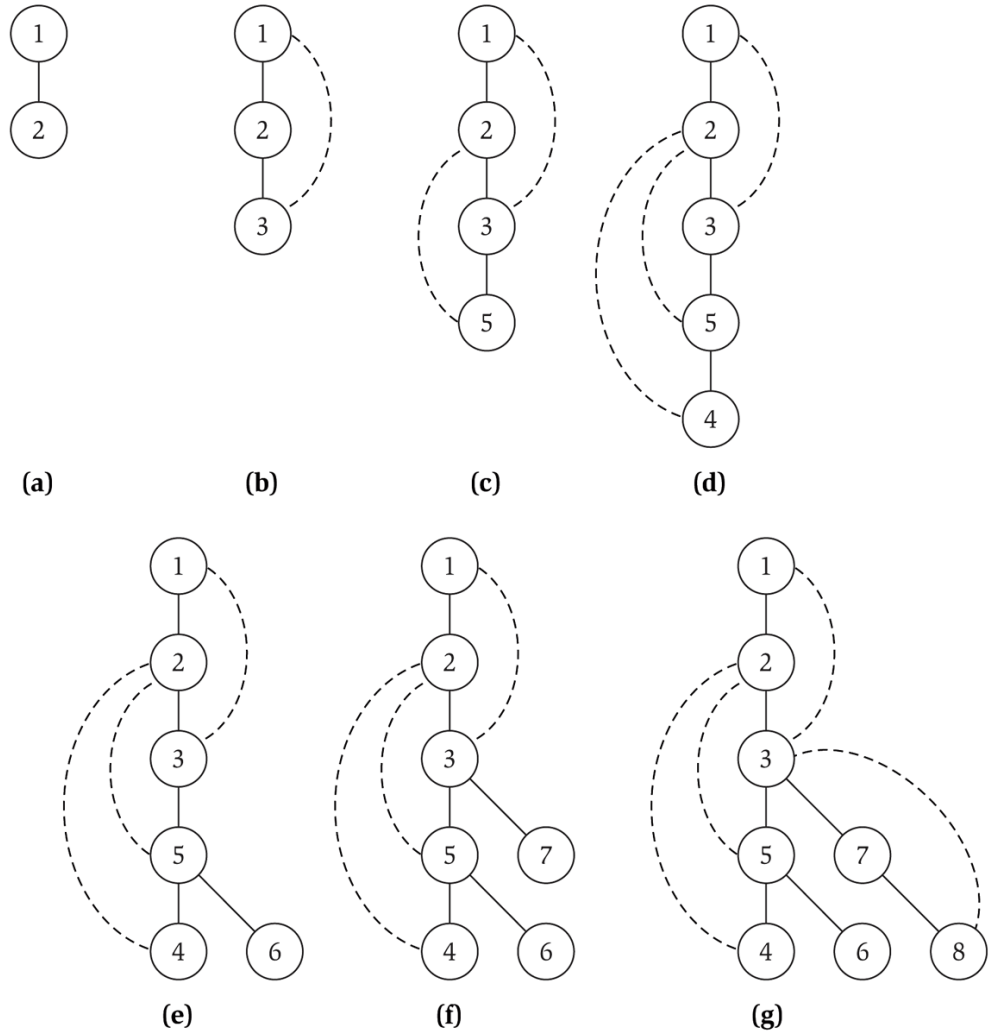
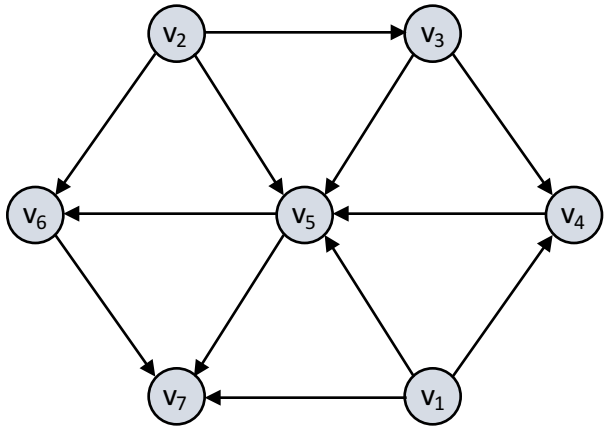


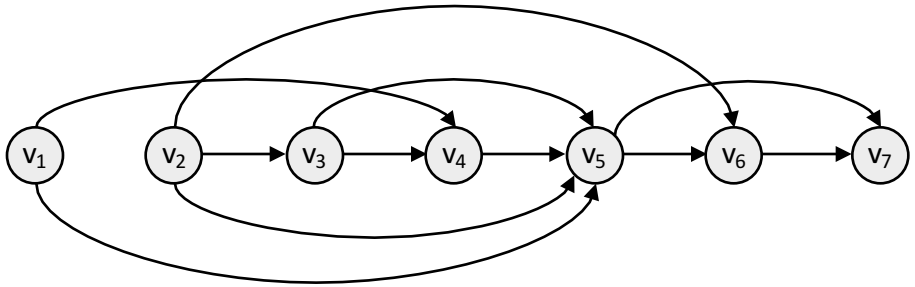
Figure 3.5 The construction of a depth-first search tree T for the graph in Figure 3.2, with (a) through (g) depicting the nodes as they are discovered in sequence. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T.

Directed Acyclic Graphs

- Def. An DAG is a directed graph that contains no directed cycles.
- Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.
- Def. A topological order of a directed graph G = (V, E) is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG

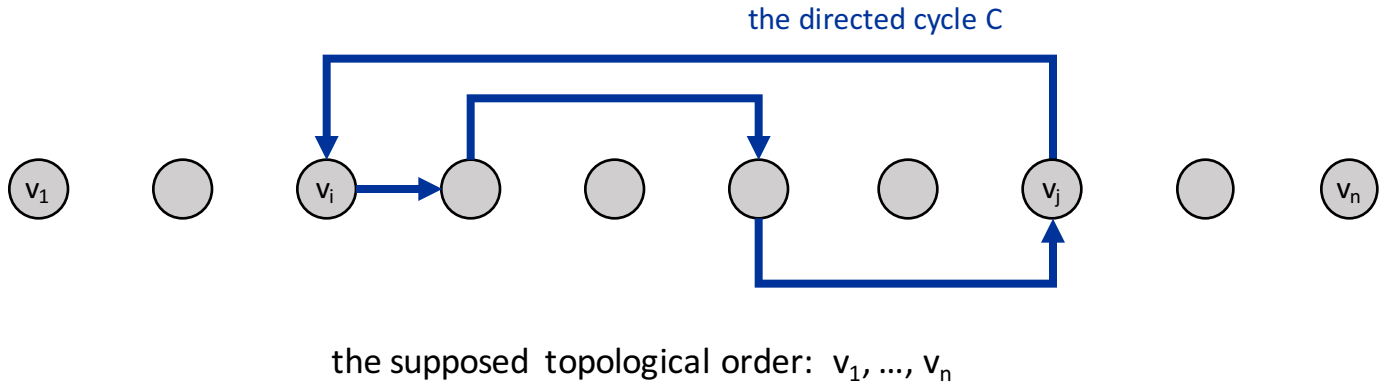


a topological ordering

Directed Acyclic Graphs

- Lemma. If G has a topological order, then G is a DAG.

- Pf. (by contradiction)
  - Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C. Let's see what happens.
  - Let v_i be the lower-indexed node in C, and let v_i be the node just before v_j;
  - By our choice of i, we have i < j.
  - On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have j < i, a contradiction. ▪



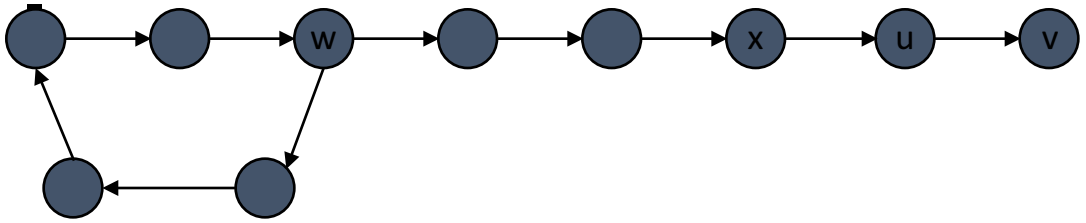
Directed Acyclic Graphs

- Lemma. If G has a topological order, then G is a DAG.
- Q. Does every DAG have a topological ordering?
- Q. If so, how do we compute one?

Directed Acyclic Graphs

- Lemma. If G is a DAG, then G has a node with no incoming edges.

- Pf. (by contradiction)
  - Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
  - Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
  - Then, since u has at least one incoming edge (x, u), we can walk backward to x.
  - Repeat until we visit a node, say w, twice.
  - Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. ∎



Directed Acyclic Graphs

- Lemma. If G is a DAG, then G has a topological ordering.

- Pf. (by induction on n)
  - Base case: true if n = 1.
  - Given DAG on n > 1 nodes, find a node v with no incoming edges.
  - G - { v } is a DAG, since deleting v cannot create cycles.
  - By inductive hypothesis, G - { v } has a topological ordering.
  - Place v first in topological ordering; then append nodes of G - { v }
  - in topological order. This is valid since v has no incoming edges. ▪

Topological Sorting Algorithm: Running Time

- Theorem. Algorithm finds a topological order in O(m + n) time.

- Pf.
  - Maintain the following information:
    - `count[w]` = remaining number of incoming edges
    - S = set of remaining nodes with no incoming edges
  - Initialization: O(m + n) via single scan through graph.
  - Update: to delete v
    - remove v from S
    - decrement `count[w]` for all edges from v to w, and add w to S if c `count[w]` hits 0
    - this is O(1) per edge ▪

Topological Ordering Example

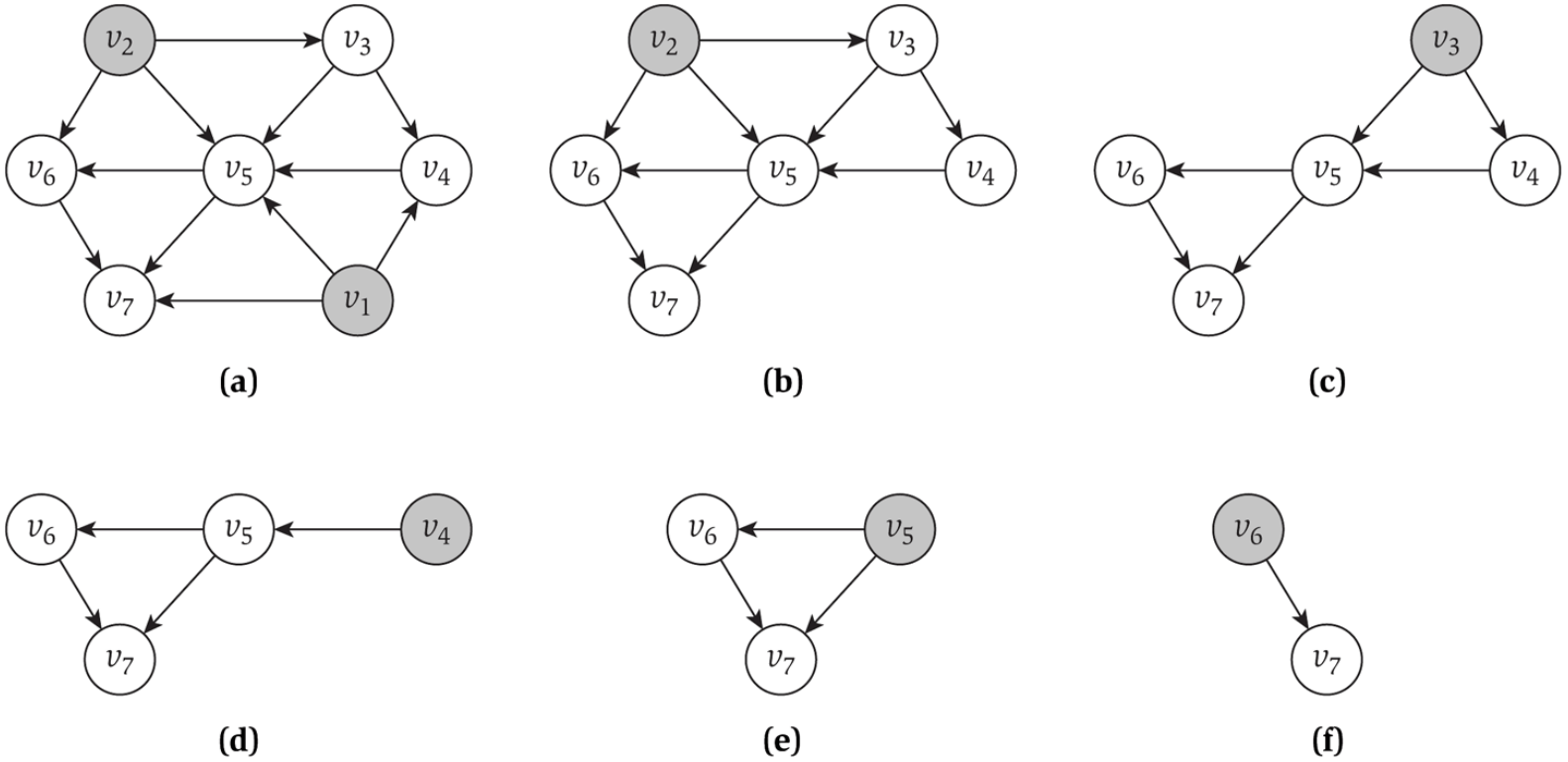
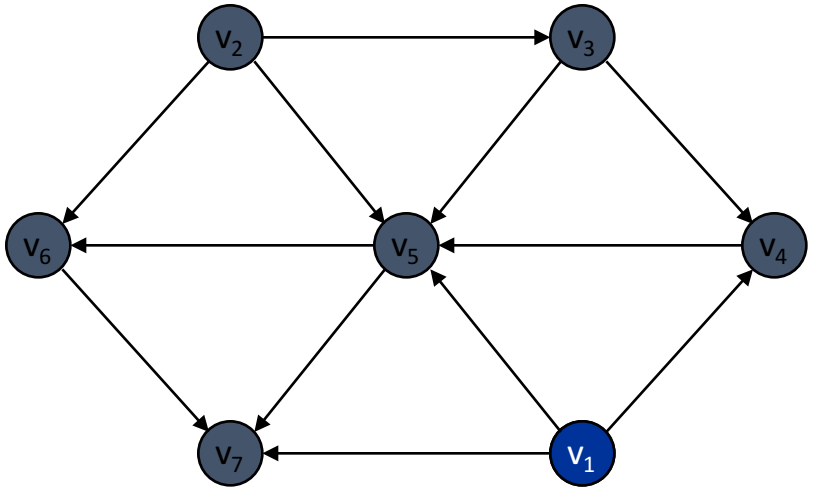


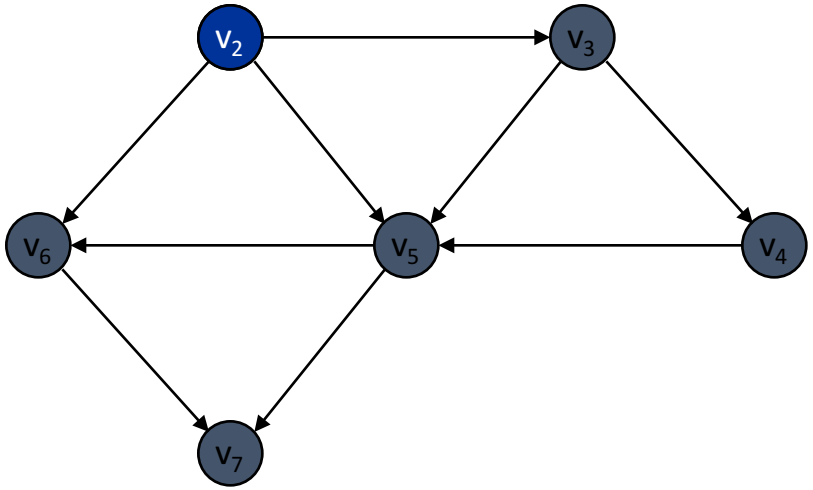
Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm's execution.

Topological Ordering Algorithm: Example



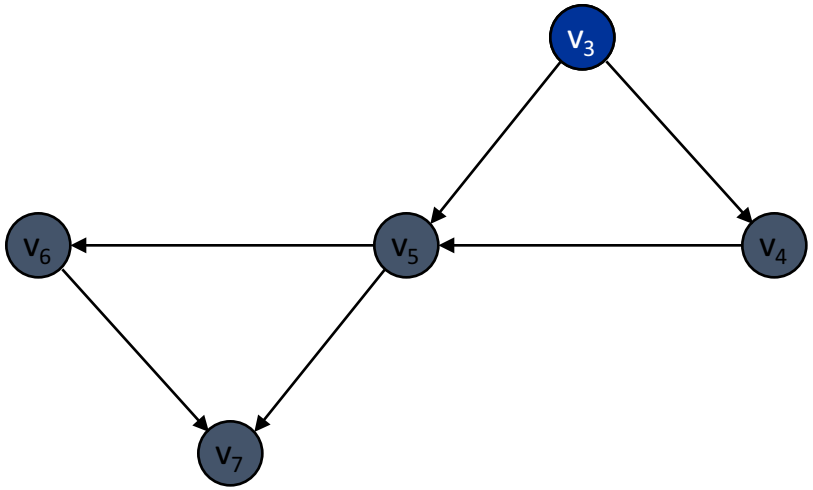
Topological order:

Topological Ordering Algorithm: Example



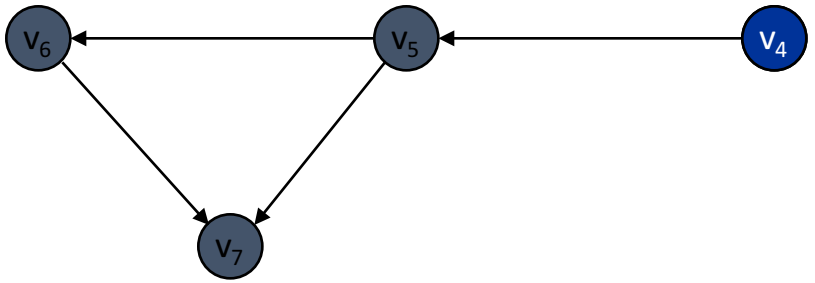
Topological order: v_1

Topological Ordering Algorithm: Example



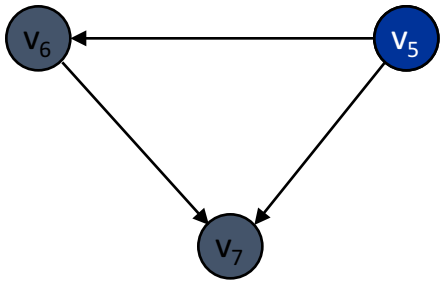
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



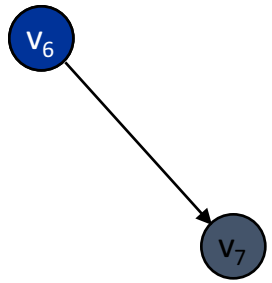
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

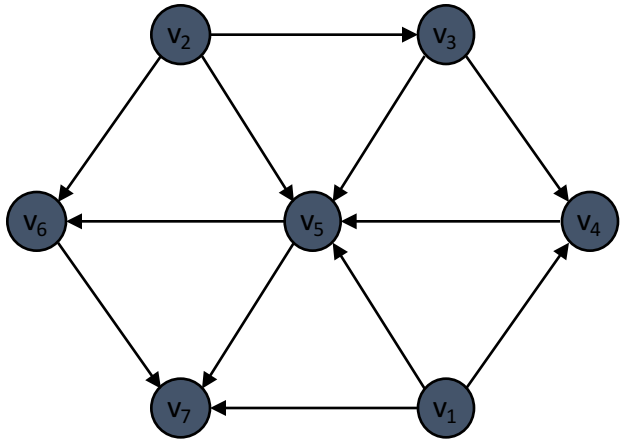


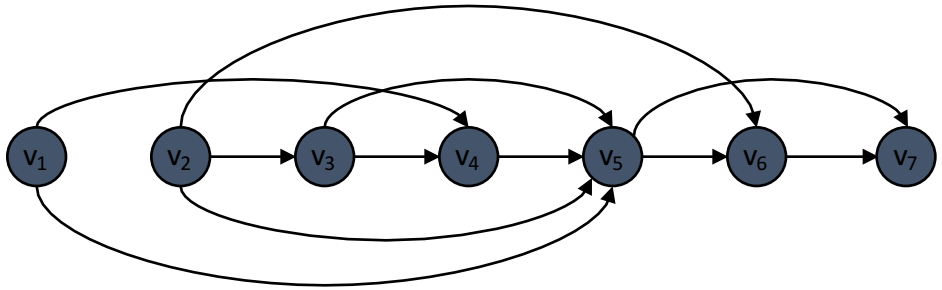
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

Topological Ordering Algorithm: Example





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.