Lecture 5

Graphs

[Part 1]

Lecture Content

1. Graph Basics

1.1 Definitions and Terminologies

1.2 Data Structures Used to Store Graphs

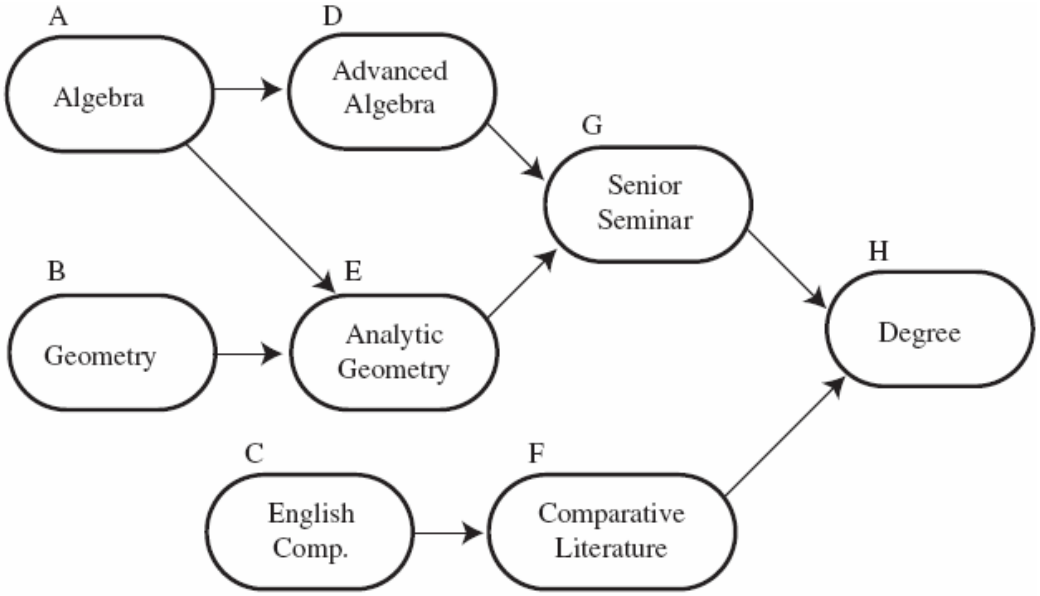
2. Graph Traversal

2.1 Depth-First Search (DFS)

2.2. Breadth-First Search (BFS)

Lecture Content

3. Topological Sorting

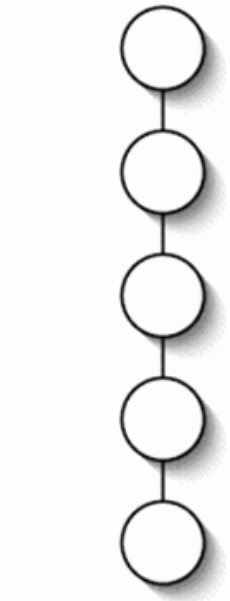


Course prerequisites

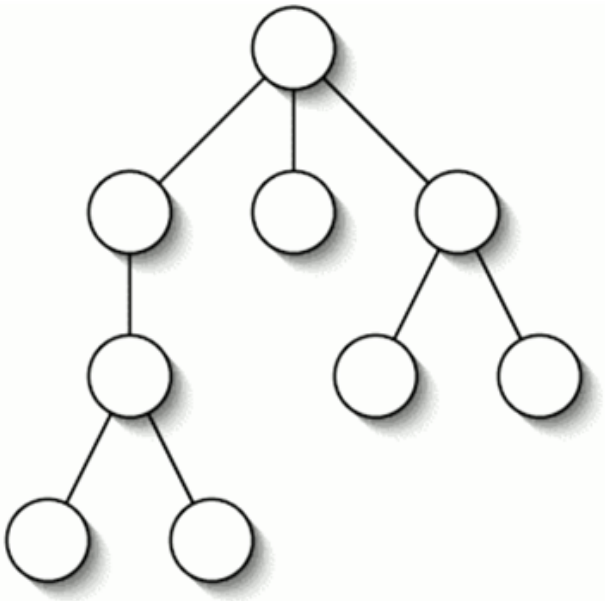
Topologically sorted order: C F B A E D G H

1. Graph Basics

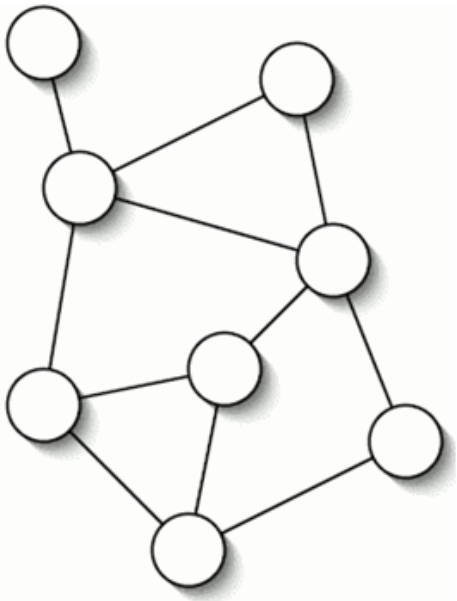
- Tree generalizes linear structures (i.e., singly linked list), graph generalizes tree.



Linear structure



Tree



Graph

1. Graph Basics

• The key difference between tree and graph is that, in a graph, there may be more than one path between two nodes.

• Many real-world problems can be modeled by using graphs. For example,

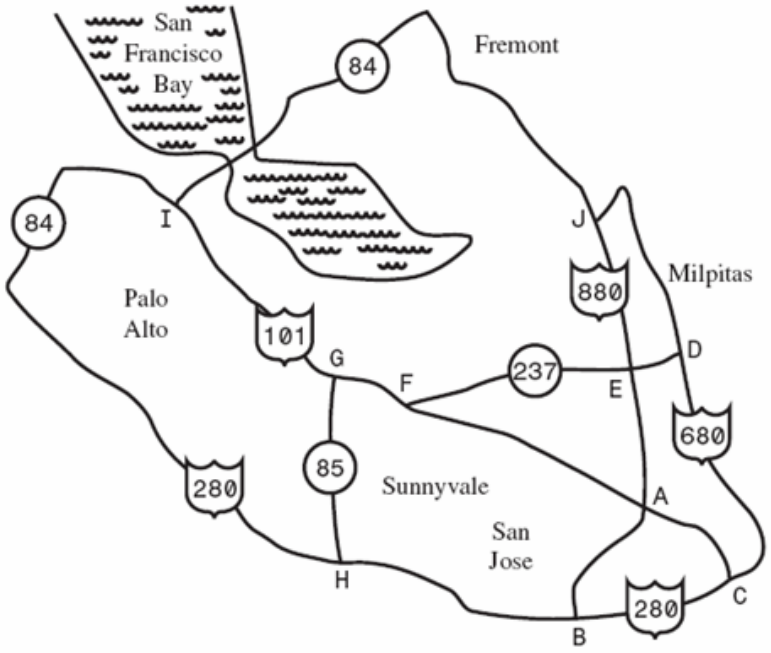
- finding the fastest routes for a mass transportation system (answers to the question: what is the shortest driving route from city *A* to city *B*),

1. Graph Basics

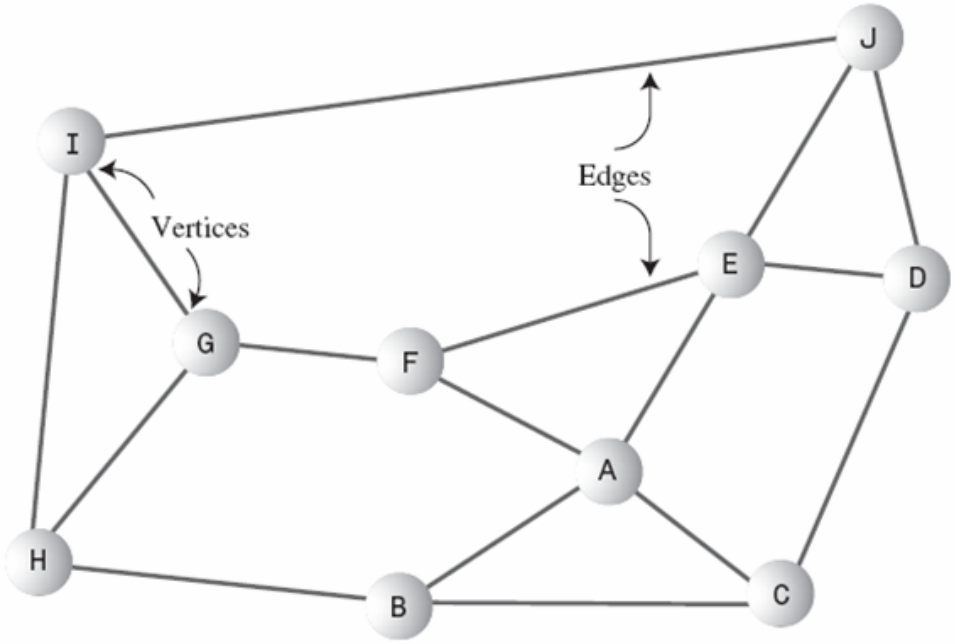
- finding a minimum spanning tree (answers to the question: how can computers be connected with the least amount of cable)

- routing electronic email through a computer network.

1. Graph Basics



Roadmap



Corresponding graph

1.1 Definitions and Terminologies

• A graph G consists of a set of **vertices** (also called nodes) V and a set of **edges** (also called arcs) E that connect the vertices.

• That is, $G = (V, E)$, where V is a set of n vertices $\{v_0, v_1, \ldots, v_{n-1}\}$ and E is a set of m edges $\{e_0, e_1, \ldots, e_{m-1}\}$.

• Each edge $e \in E$ is a pair (u, v), where $u, v \in V$ (i.e., $e = (u, v)$).

1.1 Definitions and Terminologies

• The number of vertices and edges of a graph G is denoted as $|V|$ and $|E|$, respectively (i.e., $|V| = n$ and $|E| = m$).

• If each edge $e = (u, v)$ in G is ordered (i.e., $(u, v) \neq (v, u)$), the graph is called **directed graph** (also called **digraph**). Otherwise, the graph is called **undirected graph**.

1.1 Definitions and Terminologies

- If a graph is directed, the **in-degree of a vertex** is the number of edges entering it.
- The **out-degree of a vertex** is the number of edges leaving it.
- The **degree of a vertex** is the sum of its in-degree and out-degree.

1.1 Definitions and Terminologies

• If each edge $e = (u, v)$ in G has a **weight** w (also called cost or length), the graph is called **weighted graph**.

• Vertex v is **adjacent** (also called **neighbor**) to vertex u if there is an edge from u to v.

• A **path** in a graph is a sequence of vertices connected by edges.

1.1 Definitions and Terminologies

• In **unweighted graphs**, a **path length** is the number of edges on the path.

• The **distance** between two vertices is the length of the shortest path between them.

• A **weighted path length** is the sum of weights (costs or lengths) on the path.

1.1 Definitions and Terminologies

- If $|E| = \Theta(|V|^2)$, then the graph G is called **dense graph**.

- If $|E| = \Theta(|V|)$, then the graph G is called **sparse graph**.

- A **cycle** is a path from a vertex back to itself.

1.1 Definitions and Terminologies

- A graph with no cycle is called **acyclic** graph. A directed acyclic graph is called a **DAG**.
- A graph in which every pair of vertices is connected by a path is said to be **connected**.
- Let G be a simple graph (i.e., no parallel edges and no self-loop/cycle) with n vertices and m edges. If G is undirected, then $m \leq n(n - 1)/2$. If G is directed, then $m \leq n(n - 1)$.

1.2 Data Structures Used to Store Graphs

• A graph can be stored by using an **adjacency matrix** (i.e., two-dimensional array, also called **neighbor matrix**) or an **adjacency list** (i.e., singly linked list).

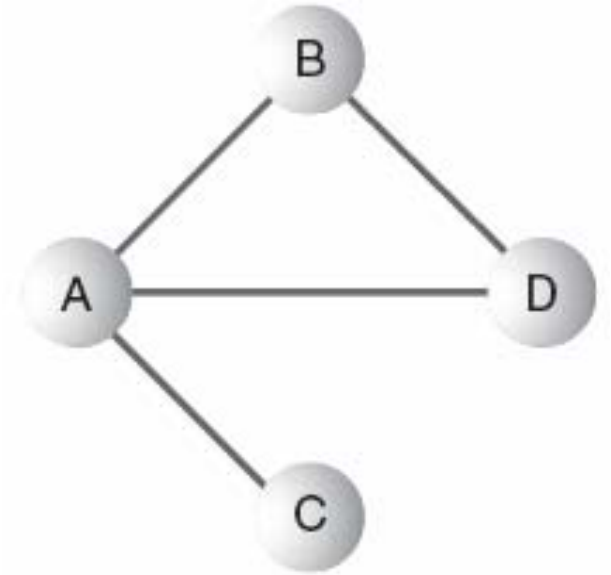
• Normally, dense and sparse graphs are represented by using adjacency matrix and an adjacency list, respectively.

Adjacency Matrix

- An adjacency matrix (e.g., `int adjMat[][]`) is a two-dimensional array in which the elements indicate whether an edge is present between two vertices. If a graph has n vertices, an $n \times n$ adjacency matrix is needed to store the graph.

Adjacency Matrix

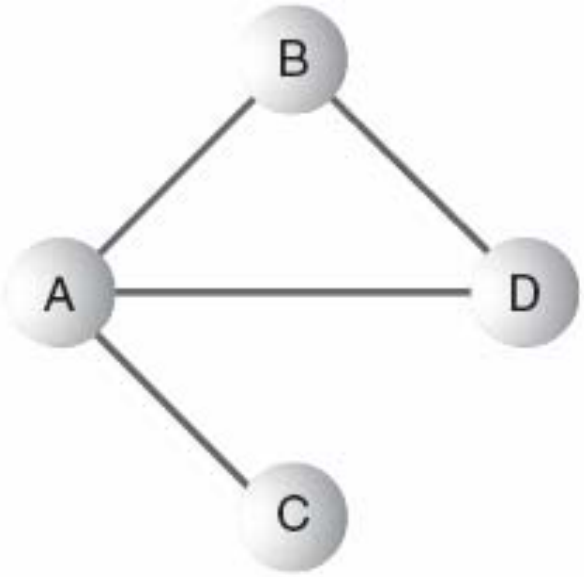
- **Example**: Consider the following graph



Adjacency Matrix

- The adjacency matrix for the graph is as follows.

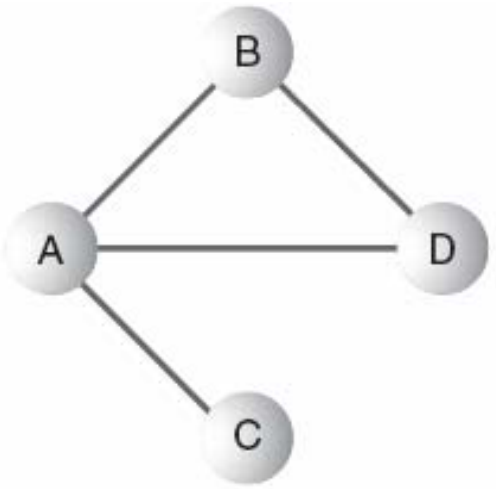
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0



Adjacency List

- An adjacency list is an array of singly linked lists. Each individual list shows **what vertices a given vertex is adjacent to**.
- The adjacency lists for the graph is given next.

A	B→C→D
B	A→D
C	A
D	A → B



Lecture Content

1. Graph Basics

   1.1 Definitions and Terminologies

   1.2 Data Structures Used to Store Graphs

2. Graph Traversal

   2.1 Depth-First Search (DFS)

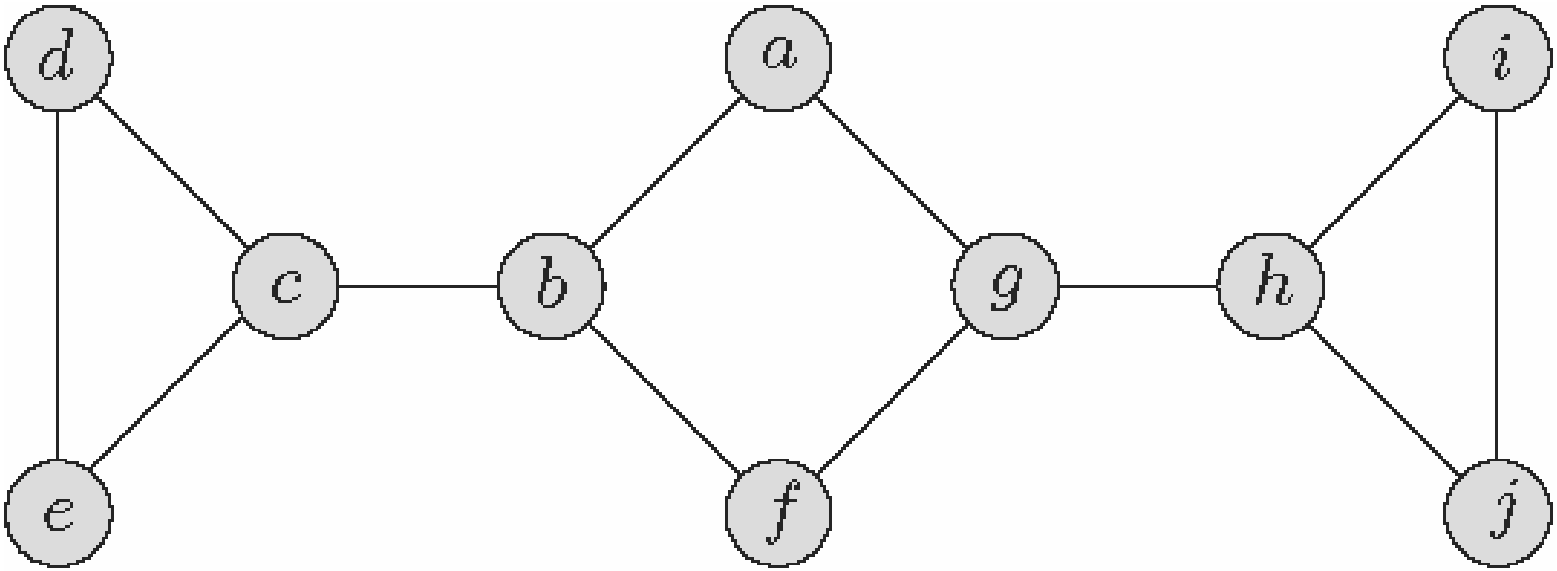
   2.2. Breadth-First Search (BFS)

2. Graph Traversal

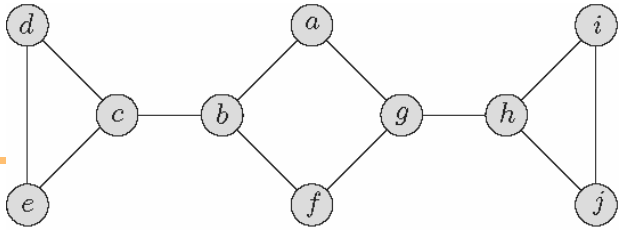
- Three traversals of a tree are preorder, inorder, and postorder. Tree traversal is always starts at the root of the tree.
- Two traversals of a graph are depth-first search (DFS) and breadth-first search (BFS). Since a graph has no root, we must specify a vertex at which to begin a traversal.
- Depth-first search is essentially a generalization of the preorder traversal of a rooted tree.

2.1 Depth-First Search (DFS)

• **Example**: List the order in which the nodes of the undirected graph shown in the figure below are visited by a *depth-first traversal* that starts from vertex *a*. Assume that we choose to visit adjacent vertices in alphabetical order.



2.1 Depth-First Search (DFS)



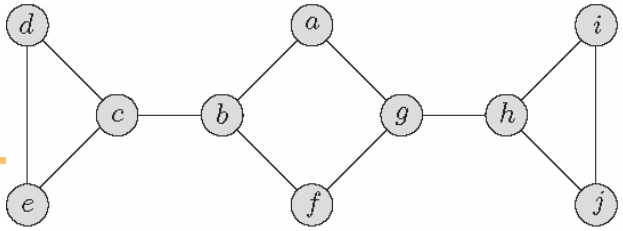
Algorithm DFS // M.H. Alsuwaiyel

Input: A directed or undirected graph $G = (V, E)$.

Output: Numbering of the vertices in depth-first search order.

1. *predfn* ← 1; *postdfn* ← 1
2. **for** each vertex $v \in V$
3. mark *v unvisited*
4. **end for**
5. **for** each vertex $v \in V$
6. **if** *v* is marked *unvisited* **then** *dfs*(*v*) // starting vertex
7. **end for**

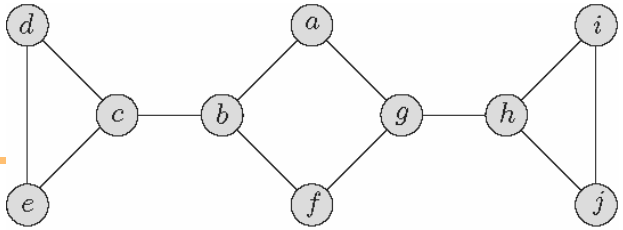
2.1 Depth-First Search (DFS)



Procedure *dfs*(*v*) // *v* is starting vertex, using stack

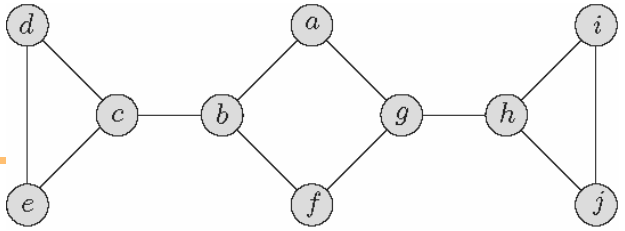
1. $S \leftarrow \{v\}$ // insert *v* into stack
2. mark *v visited*
3. **while** $S \neq \{\}$
4. $v \leftarrow$ Peek(S) // *v* is current vertex
5. find an unvisited neighbor *w* of *v*

2.1 Depth-First Search (DFS)




```
6.    if w exists then
7.        Push(w, S)
8.        mark w visited
9.        predfn ← predfn + 1
10.   else
11.       Pop(S); postdfn ← postdfn + 1
12.   end if
13. end while
```

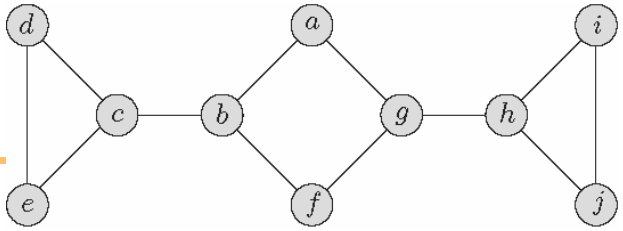
2.1 Depth-First Search (DFS)



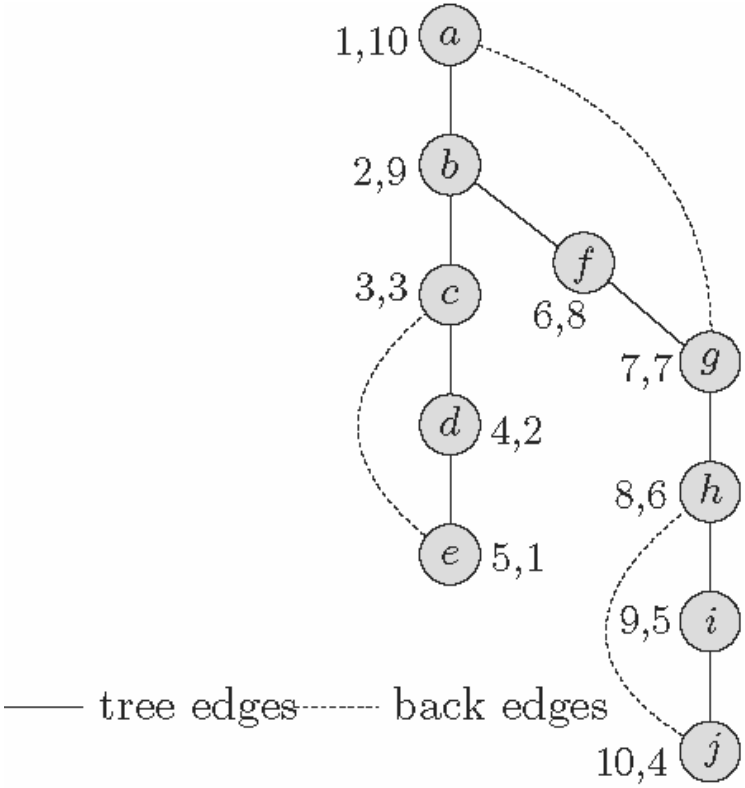
- The stack contents during DFS are given below.

Event	Stack Contents	Current vertex
Visit *a*	*a*	*a*
Visit *b*	*a*, *b*	*b*
Visit *c*	*a*, *b*, *c*	*c*
Visit *d*	*a*, *b*, *c*, *d*	*d*
Visit *e*	*a*, *b*, *c*, *d*, *e*	*e*
Pop *e*	*a*, *b*, *c*, *d*	*d*
Pop *d*	*a*, *b*, *c*	*c*
Pop *c*	*a*, *b*	*b*
Visit *f*	*a*, *b*, *f*	*f*
Visit *g*	*a*, *b*, *f*, *g*	*g*
Visit *h*	*a*, *b*, *f*, *g*, *h*	*h*
Visit *i*	*a*, *b*, *f*, *g*, *h*, *i*	*i*
Visit *j*	*a*, *b*, *f*, *g*, *h*, *i*, *j*	*j*
Pop *j*	*a*, *b*, *f*, *g*, *h*, *i*	*i*
Pop *i*	*a*, *b*, *f*, *g*, *h*	*h*
Pop *h*	*a*, *b*, *f*, *g*	*g*
Pop *g*	*a*, *b*, *f*	*f*
Pop *f*	*a*, *b*	*b*
Pop *b*	*a*	*a*
Pop *a*	Empty	

2.1 Depth-First Search (DFS)



- The order in which the nodes are visited by a DFS that starts from vertex *a* is *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*.
- The resulting tree (i.e., the *depth-first search tree*) is



2.1 Depth-First Search (DFS)

- Tree edges: edges in the depth-first search tree. An edge (v, w) is a tree edge if w was first visited when exploring the edge (v, w).
- Back edges: All other edges.

2.1 Depth-First Search (DFS)

In depth-first search traversal of directed graphs, however, the edges of G are classified into four types:

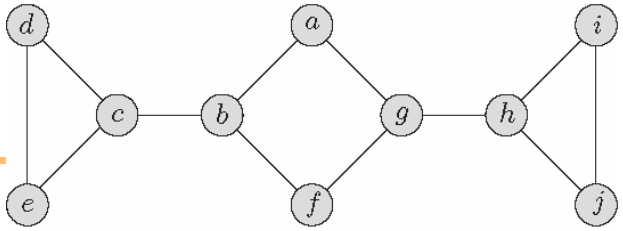
- Tree edges: edges in the depth-first search tree. An edge (v, w) is a tree edge if w was first visited when exploring the edge (v, w).
- Back edges: edges of the form (v, w) such that w is an ancestor of v in the depth-first search tree (constructed so far) and vertex w was marked visited when (v, w) was explored.

2.1 Depth-First Search (DFS)

• Forward edges: edges of the form (v, w) such that w is a descendant of v in the depth-first search tree (constructed so far) and vertex w was marked visited when (v, w) was explored.

• Cross edges: All other edges.

2.1 Depth-First Search (DFS)



Procedure *dfs*(*v*) // *v* is starting vertex, using recursion

1. mark *v* visited
2. *predfn* ← *predfn* + 1
3. **for** each edge (*v*, *w*) ∈ *E*
4. **if** *w* is marked *unvisited* **then** *dfs*(*w*)
5. **end for**
6. *postdfn* ← *postdfn* + 1

2.2 Breadth-First Search (BFS)

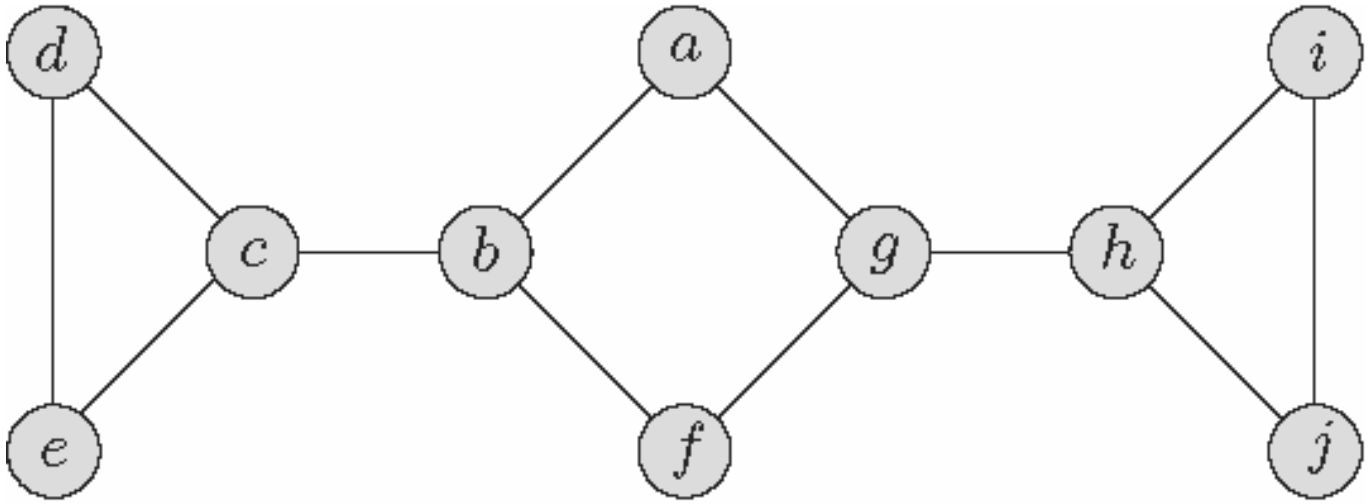
• Depth-first search algorithm gets as far away from the starting point as quickly as possible. DFS is implemented using a stack.

• In contrast, breadth-first search algorithm stays as close as possible to the starting point. BFS visits all the vertices adjacent to the starting vertex, and then goes further afield. BFS is implemented using a queue.

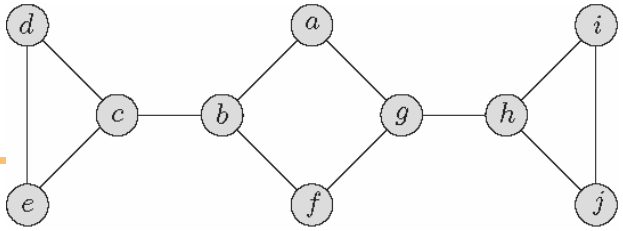
• The level-order traversal of a tree is an example of the breadth-first traversal.

2.2 Breadth-First Search (BFS)

• **Example**: List the order in which the nodes of the undirected graph shown in the figure below are visited by a *breadth-first traversal* that starts from vertex *a*. Assume that we choose to visit adjacent vertices in alphabetical order.



2.2 Breadth-First Search (BFS)



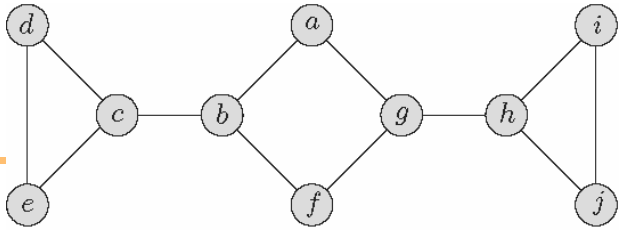
Algorithm BFS // M.H. Alsuwaiyel

Input: A directed or undirected graph $G = (V, E)$.

Output: Numbering of the vertices in BFS order.

1. $bfn \leftarrow 1$
2. **for** each vertex $v \in V$
3. mark v *unvisited*
4. **end for**
5. **for** each vertex $v \in V$
6. **if** v is marked *unvisited* **then** *bfs*(v) // starting vertex
7. **end for**

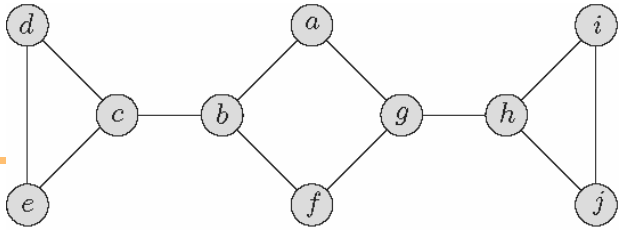
2.2 Breadth-First Search (BFS)



Procedure *bfs*(*v*) // *v* is starting vertex, using queue

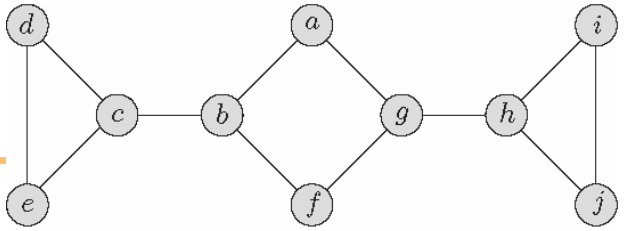
1. $Q \leftarrow \{v\}$ // insert *v* into queue
2. mark *v visited*
3. **while** $Q \neq \{\}$
4. $\quad v \leftarrow$ dequeue(Q) // *v* is current vertex
5. $\quad$ **for** each edge $(v, w) \in E$
6. $\quad\quad$ **if** *w* is marked *unvisited* **then**
7. $\quad\quad\quad$ enqueue(w, Q)
8. $\quad\quad\quad$ mark *w visited*
9. $\quad\quad\quad$ *bfn* $\leftarrow$ *bfn* + 1
10. $\quad\quad$ **end if**

2.2 Breadth-First Search (BFS)



11. **end for**
12. **end while**

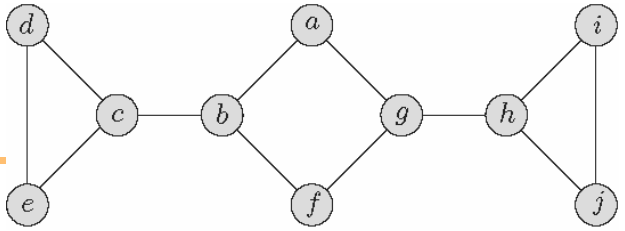
2.2 Breadth-First Search (BFS)



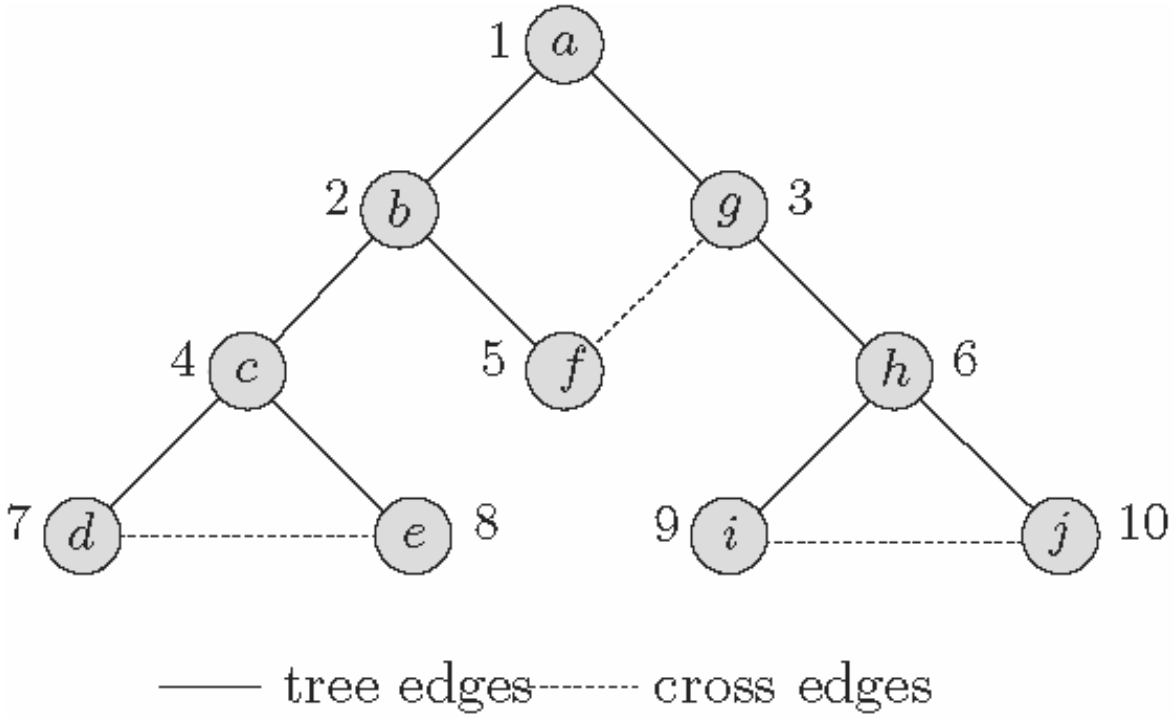
- The queue contents during BFS are given below.

Event	Queue Contents	Current vertex
Visit *a*	*a*	*a*
Visit *b*	*b*, *g*	*b*
Visit *g*	*g*, *c*, *f*	*g*
Visit *c*	*c*, *f*, *h*	*c*
Visit *f*	*f*, *h*, *d*, *e*	*f*
Visit *h*	*h*, *d*, *e*	*h*
Visit *d*	*d*, *e*, *i*, *j*	*d*
Visit *e*	*e*, *i*, *j*	*e*
Visit *i*	*i*, *j*	*i*
Visit *j*	*j*	*j*
Done	Empty	

2.2 Breadth-First Search (BFS)



- The order in which the nodes are visited by a BFS that starts from vertex *a* is *a*, *b*, *g*, *c*, *f*, *h*, *d*, *e*, *i*, *j*.
- The resulting tree (i.e., the *breadth-first search tree*) is

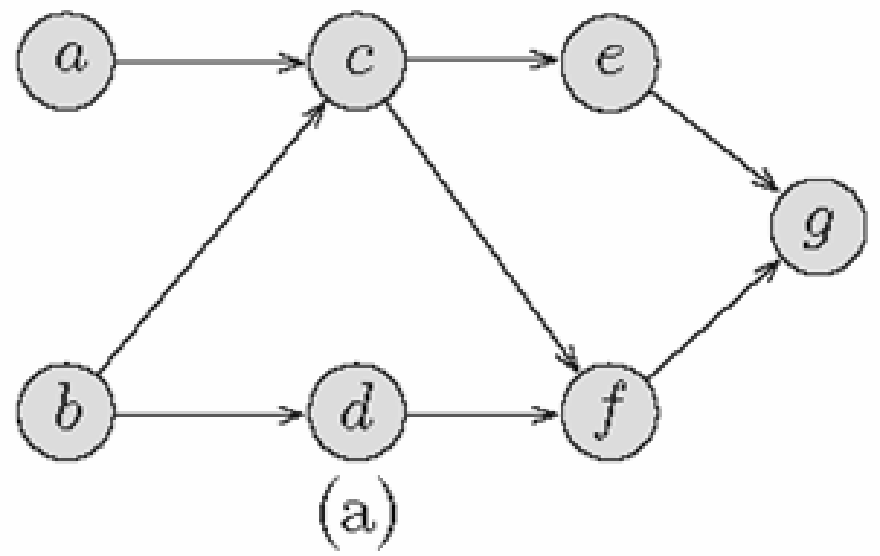


2.2 Breadth-First Search (BFS)

- Tree edges: edges in the breadth-first search tree. An edge (v, w) is a tree edge if w was first visited when exploring the edge (v, w).
- Cross edges: All other edges.

3. Topological Sorting

• Given a directed acyclic graph (dag for short) $G = (V, E)$, the problem of **topological sorting** is to find a linear ordering of its vertices in such a way that if $(v, w) \in E$, then v appears before w in the ordering.

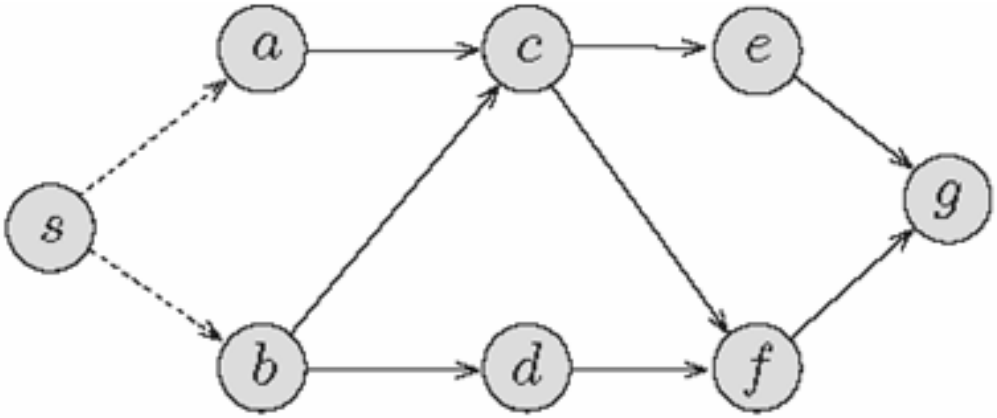


(a)

3. Topological Sorting

• For example, one possible topological sorting of the vertices in the dag shown in figure (a) above is b, d, a, c, f, e, g (or a, b, d, c, e, f, g)

• We will assume that the dag has only one vertex, say s, of indegree 0. If not, we may simply add a new vertex s and edges from s to all vertices of indegree 0.



(b)

3. Topological Sorting

- Next, we simply carry out a depth-first search on *G* starting at vertex *s*.
- When the traversal is complete, the values of the counter *postdfn* define a reverse topological ordering of the vertices in the dag.
- Thus, to obtain the ordering, we may add an output step to Algorithm DFS just after the counter *postdfn* is incremented. The resulting output is reversed to obtain the desired topological ordering.

Exercises

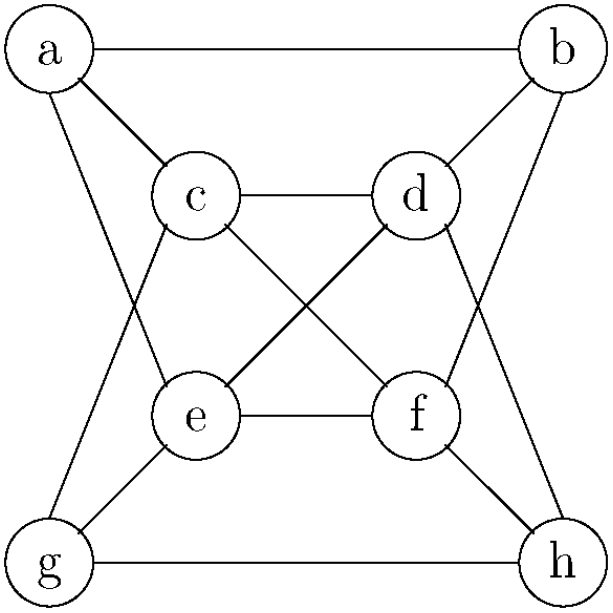
1. Write a complete program to implement the DFS.

2. Write a complete program to implement the BFS.

3. Modify the DFS program to find the topologically sorted order of a given dag.

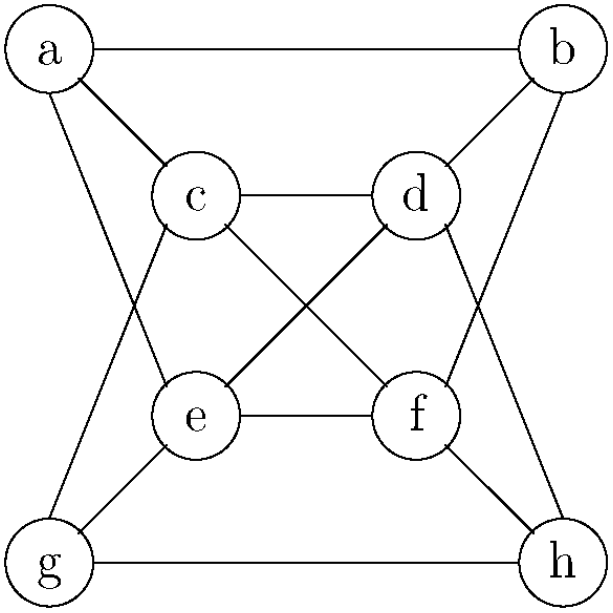
Exercises

4. List the order in which the nodes of the undirected graph shown in the figure below are visited by a depth-first traversal that starts from vertex *a*. Repeat this exercise for a depth-first traversal starting from vertex *d*.



Exercises

5. List the order in which the nodes of the undirected graph shown in the figure below are visited by a breadth-first traversal that starts from vertex *a*. Repeat this exercise for a breadth-first traversal starting from vertex *d*.



References

1. Noel Kalicharan. 2008. *Data Structures in Java*. CreateSpace. ISBN: 143827517X. Chapter 5

2. Noel Kalicharan. 2008. *Data Structures in C*. Createspace Press. ISBN: 1438253273. Chapter 7

3. Robert Lafore. 2002. *Data Structures and Algorithms in Java*. 2nd Ed, SAMS. ISBN: 0672324539.

References

4. M.H. Alsuwaiyel. 1999. *Algorithms Design Techniques and Analysis*. World Scientific Publishing. ISBN: 9810237405. Chapters 1, 4, 6

5. Anany V. Levitin. 2011. *Introduction to the Design and Analysis of Algorithms*. 3Ed. Addison-Wesley. ISBN: 0132316811.