

Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. Applications include:

- Printing out the contents of each edge and vertex.
- Counting the number of edges.
- Identifying connected components of a graph.

For *correctness*, we must do the traversal in a systematic way so that we don't miss anything.

For *efficiency*, we must make sure we visit each edge at most twice.

Marking Vertices

The idea in graph traversal is that we mark each vertex when we first visit it, and keep track of what is not yet completely explored.

For each vertex, we maintain two flags:

- *discovered* - have we encountered this vertex before?
- *explored* - have we finished exploring this vertex?

We must maintain a structure containing all the vertices we have discovered but not yet completely explored.

Initially, only a single start vertex is set to be discovered.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

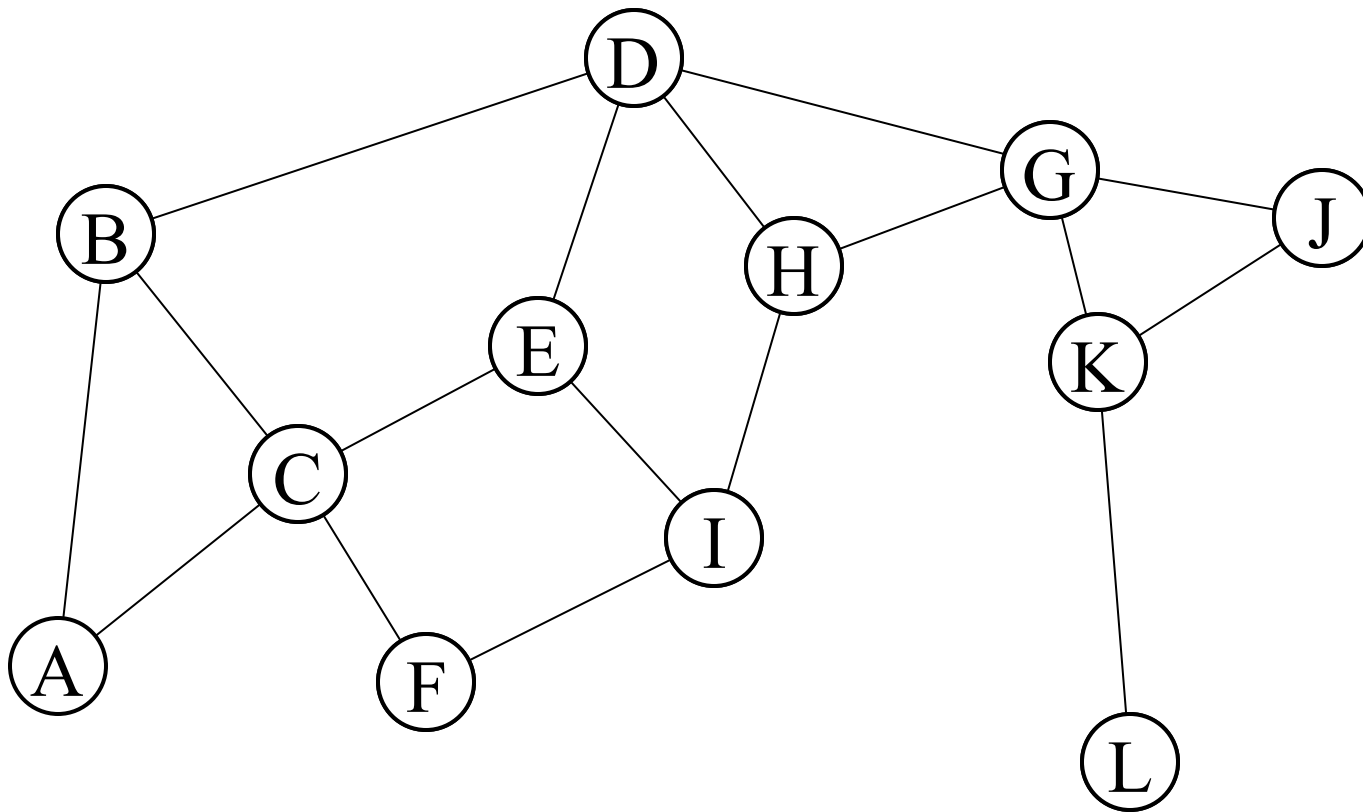
Suppose not, i.e. there exists a vertex which was unvisited whose neighbor *was* visited. This neighbor will eventually be explored so we *would* visit it....

Traversal Orders

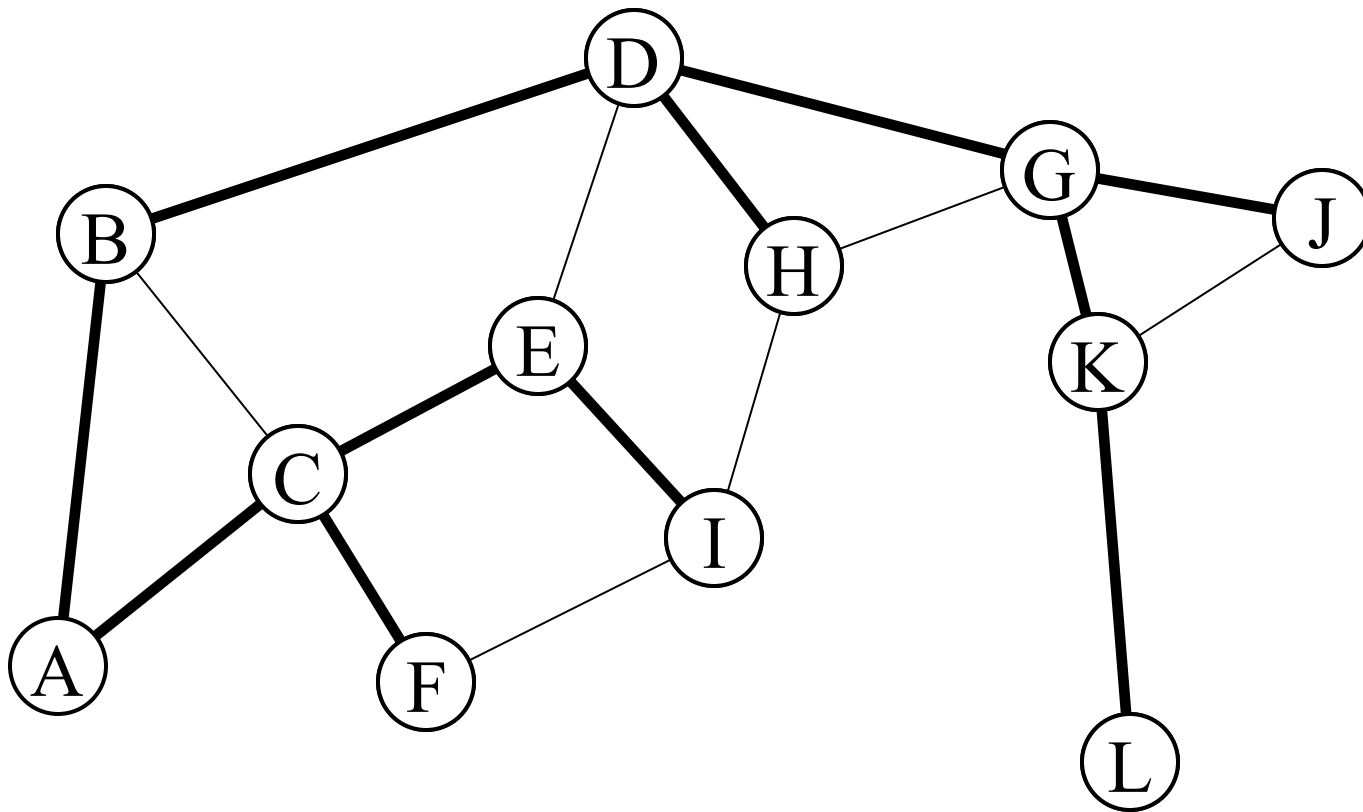
The order we explore the vertices depends upon the data structure used to hold the discovered vertices yet to be fully explored:

- ***Queue*** - by storing the vertices in a first-in, first out (FIFO) queue, we explore the oldest unexplored vertices first. Thus we radiate out slowly from the starting vertex, defining a so-called *breadth-first search*.
- ***Stack*** - by storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by lurching along a path, constantly visiting a new neighbor if one is available, and backing up only when surrounded by previously discovered vertices. Thus our explorations speed away from our starting point, defining a so-called *depth-first search*.

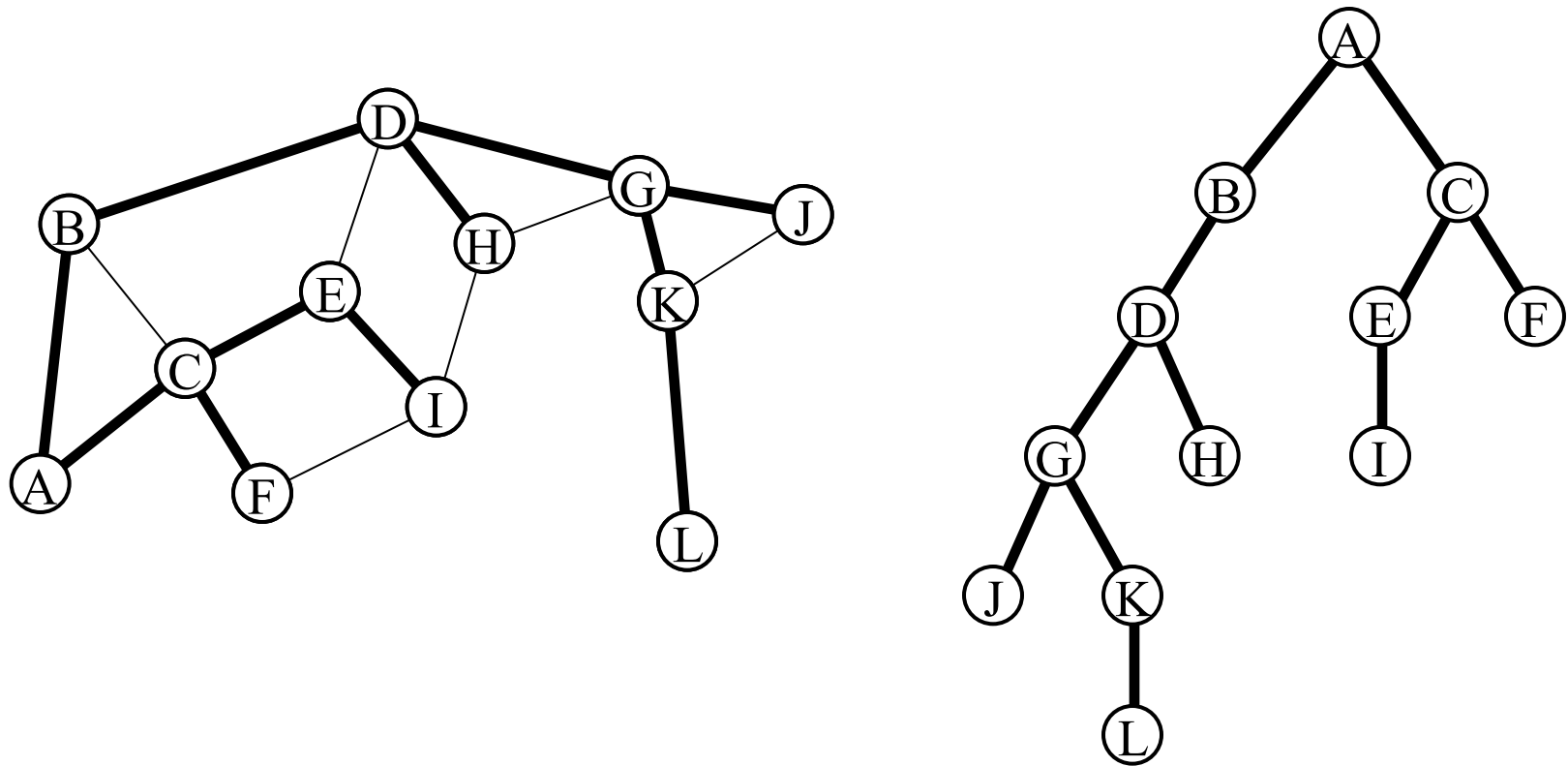
Example Graph



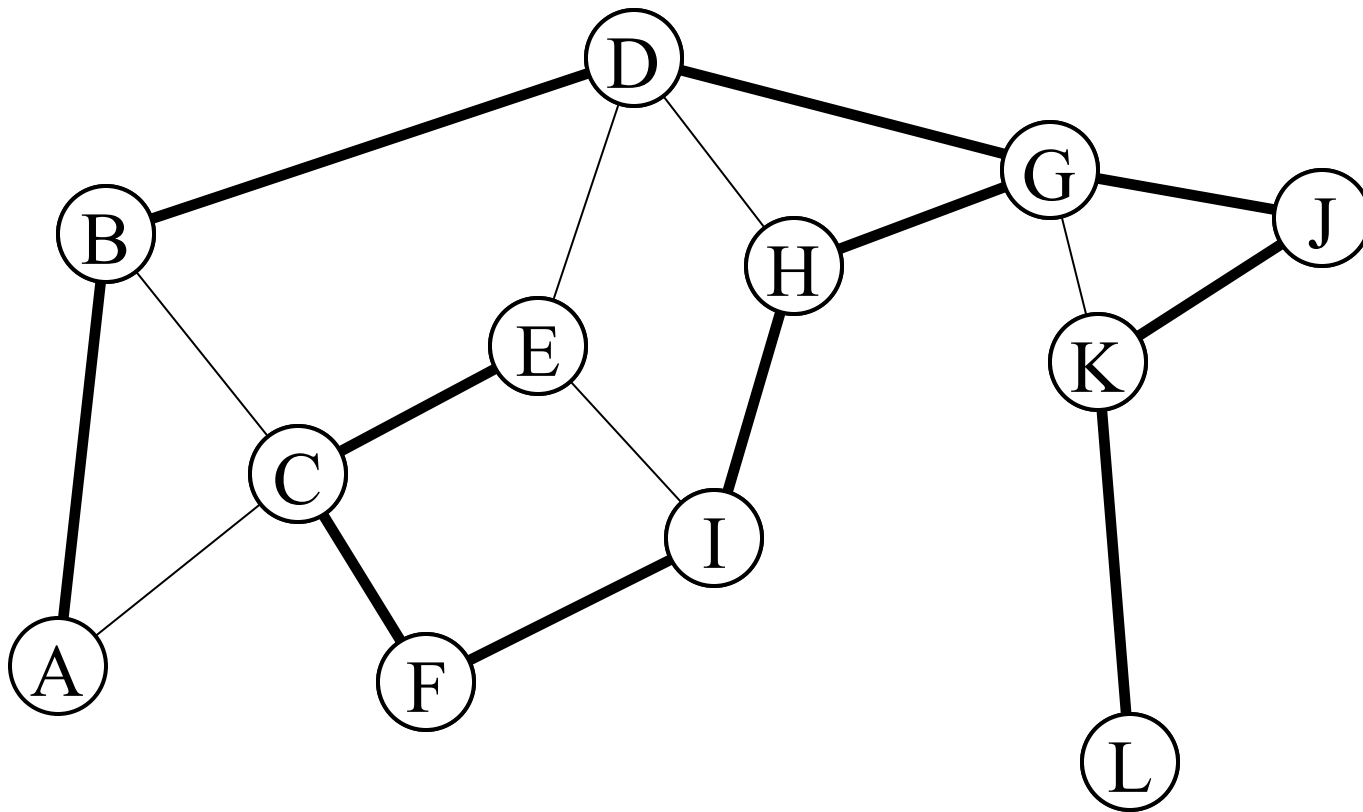
Breadth-First Search Tree



Breadth-First Search Tree

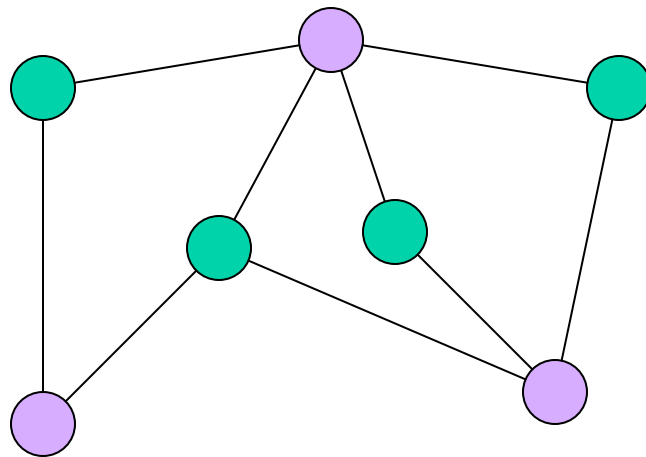


Depth-First Search Tree



Question

Give an efficient algorithm to determine if a graph is bipartite. (Bipartite means that the graph can be colored with 2 colors such that all edges connect vertices of different colors.)



Question

Give an $O(n)$ algorithm to determine if an undirected graph contains a cycle.

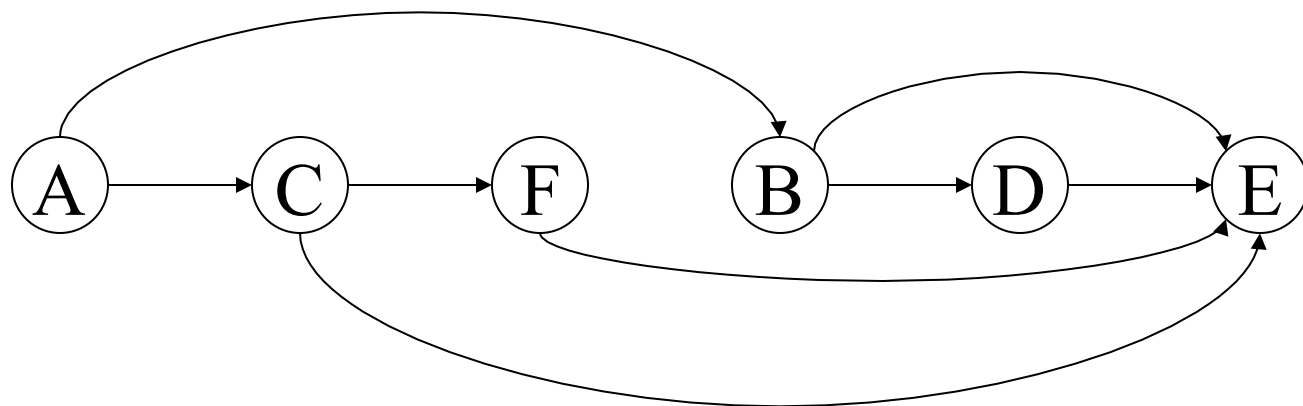
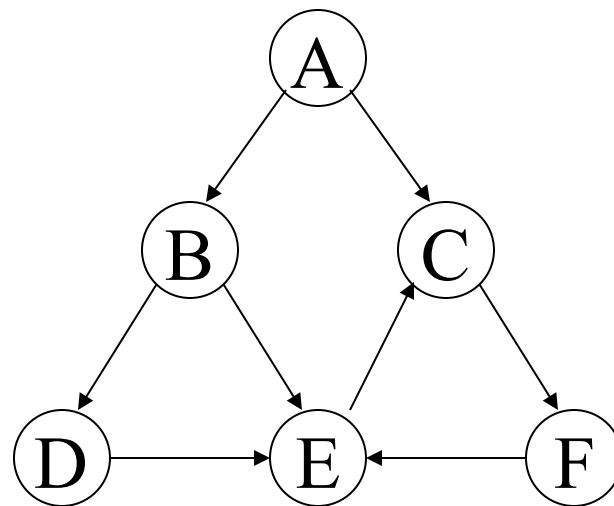
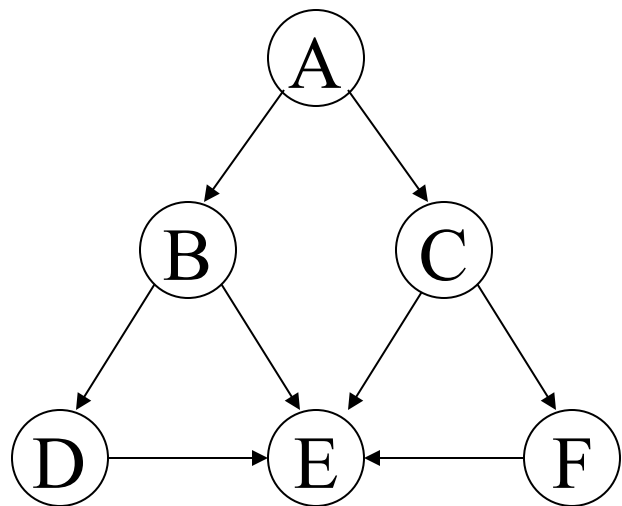
Note: Traversal by BFS and DFS both take $O(n+m)$ time.... What can we do?

Topological Sorting

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

A topological sort is an ordering of vertices such that all edges go from left to right. Only an acyclic graph can have a topological sort because eventually all cycles must return to their starting point.

How can we find a topological sort?



Topological Sorting

Consider doing a depth-first-search on the graph - what properties does the resulting tree have? Can we use this to find a topological sorting?

Since a DAG may have multiple sources in it (vertices with outgoing edges but none incoming), it is not always possible to construct a depth-first-search tree.

What if we do a DFS from all of the sources? How can we combine the results together?