



A **graph** is a data structure which encodes pairwise relationships among a set of objects.

A graph $G = (V, E)$ is a collection of vertices V and a collection of edges $E \subseteq V \times V$. Each edge represents a relationship between the corresponding vertices in V .

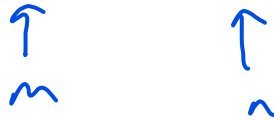
In some settings, the relationship that an edge represents is symmetric, and thus an edge is just a subset of two vertices $\{u, v\}$ for some $u, v \in V$ (we call u and v neighbors). Other times the relationship is asymmetric, and thus an edge is an ordered pair (u, v) .

Symmetric: 

asymmetric: 

If the edges represent asymmetric relationships, then we call the graph a directed graph (or digraph). If a graph is not specified to be directed, then generally we view the edges as representing symmetric relationships. Sometimes these graphs are called *undirected graphs*.

In either case, we have $|E| = O(|V|^2)$.



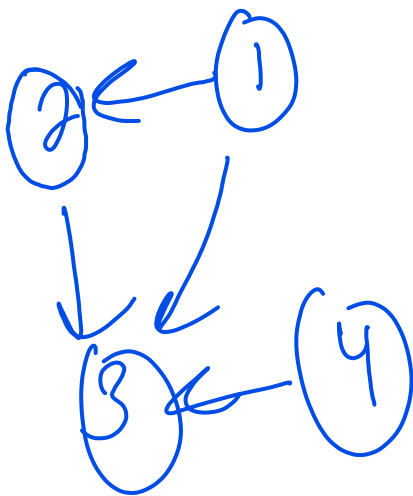
In an undirected graph, the degree of v is the number of neighbors of v . For digraphs, the out degree is the number of “outgoing” edges there are from v and the in degree is defined similarly.

There are many applications in which a graph may be a useful data structure:

- Transportation networks
 - Vertices represent airports and edges represent the existence of a flight between the corresponding airports.
- Communication networks
 - Vertices represent computers on a network and edges exist between computers with a direct physical link connecting them.
- Information networks
 - Vertices represent web pages and (directed) edges represent a link from one web page to another.

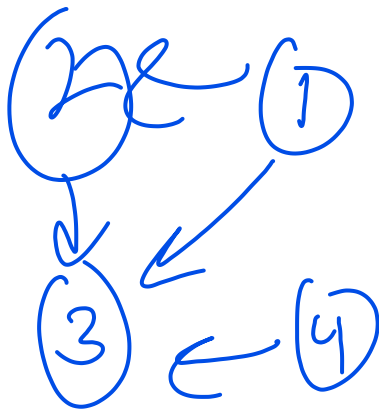
One way of representing a graph is an adjacency matrix. Let $|V| = n$. The adjacency matrix A of a graph G is the $n \times n$ matrix such that:

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{o/w} \end{cases}$$



	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Another way of representing a graph is an adjacency list representation. For each $v \in V$, its adjacency list $Adj[v]$ is the list of vertices adjacent to v .



$$Adj[1] = \{2, 3\}$$

$$Adj[2] = \{3\}$$

$$Adj[3] = \{\}$$

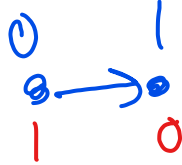
$$Adj[4] = \{3\}$$

In an undirected graph, $|Adj[v]| = \underline{degree(v)}$. For digraphs, $|Adj[v]| = \underline{out-degree(v)}$.



Handshaking Lemma:

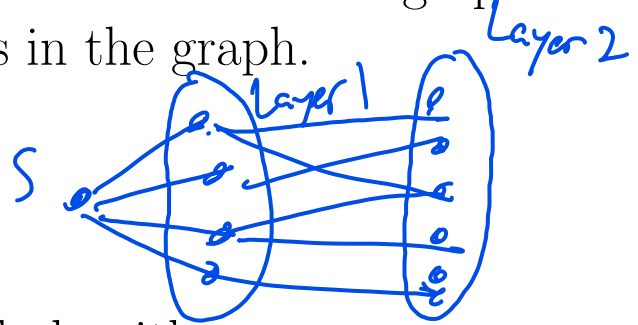
- Undirected graphs: $\sum_{v \in V} \text{degree}(v) = 2|E|$
- Digraphs: $\sum_{v \in V} \text{in-degree}(v) + \sum_{v \in V} \text{out-degree}(v) = 2|E|$



This implies that adjacency lists use $\Theta(|V| + |E|)$ storage. Contrast this with adjacency matrices use $O(|V|^2)$ storage. Adjacency lists use less storage when the graphs are “sparse” (sub-quadratic number of edges).

We will assume we are using the adjacency list representation unless stated otherwise.

A common task one might want to do with a graph is to traverse each of the nodes in the graph.

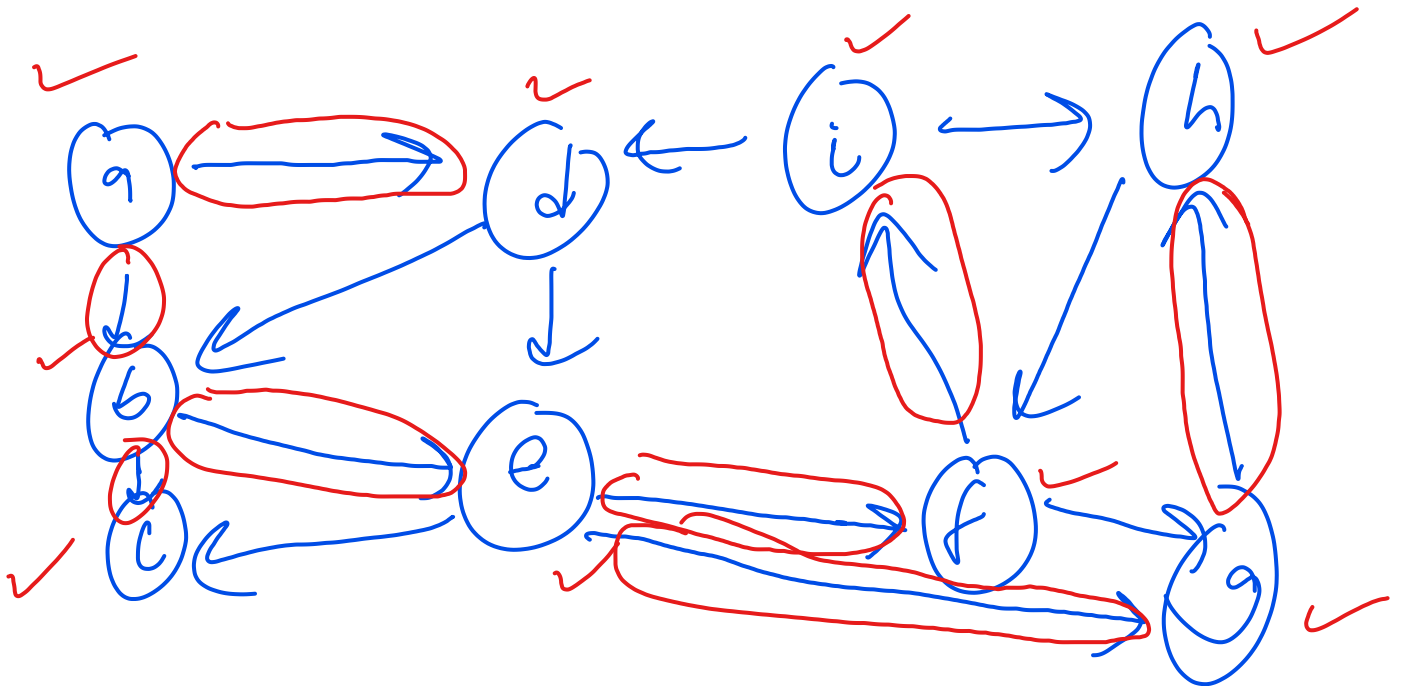


Two popular graph traversal algorithms:

- Breadth-first search (BFS): Start from an arbitrary vertex v and visit the remaining vertices in “layers”. We first visit all of v ’s neighbors (the first layer), and then we visit all of the neighbors of the first layer (which we have not already visited), etc.
- Depth-first search (DFS): Start from an arbitrary vertex v , and then visit one neighbor of v . Then visit a new neighbor from this vertex. Keep visiting an unvisited neighbor until we get “stuck”, then backtrack and try again.

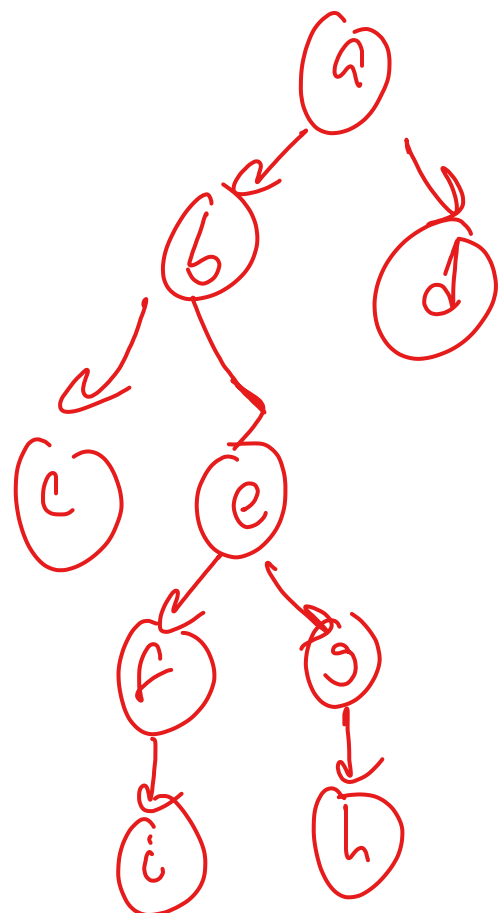
When discussing these algorithms, we assume $|V| = n$ and $|E| = m$ (common notation when discussing graphs).

Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.



Q : a b d c e f g i h
 x x x x x x x x x

BFS tree:

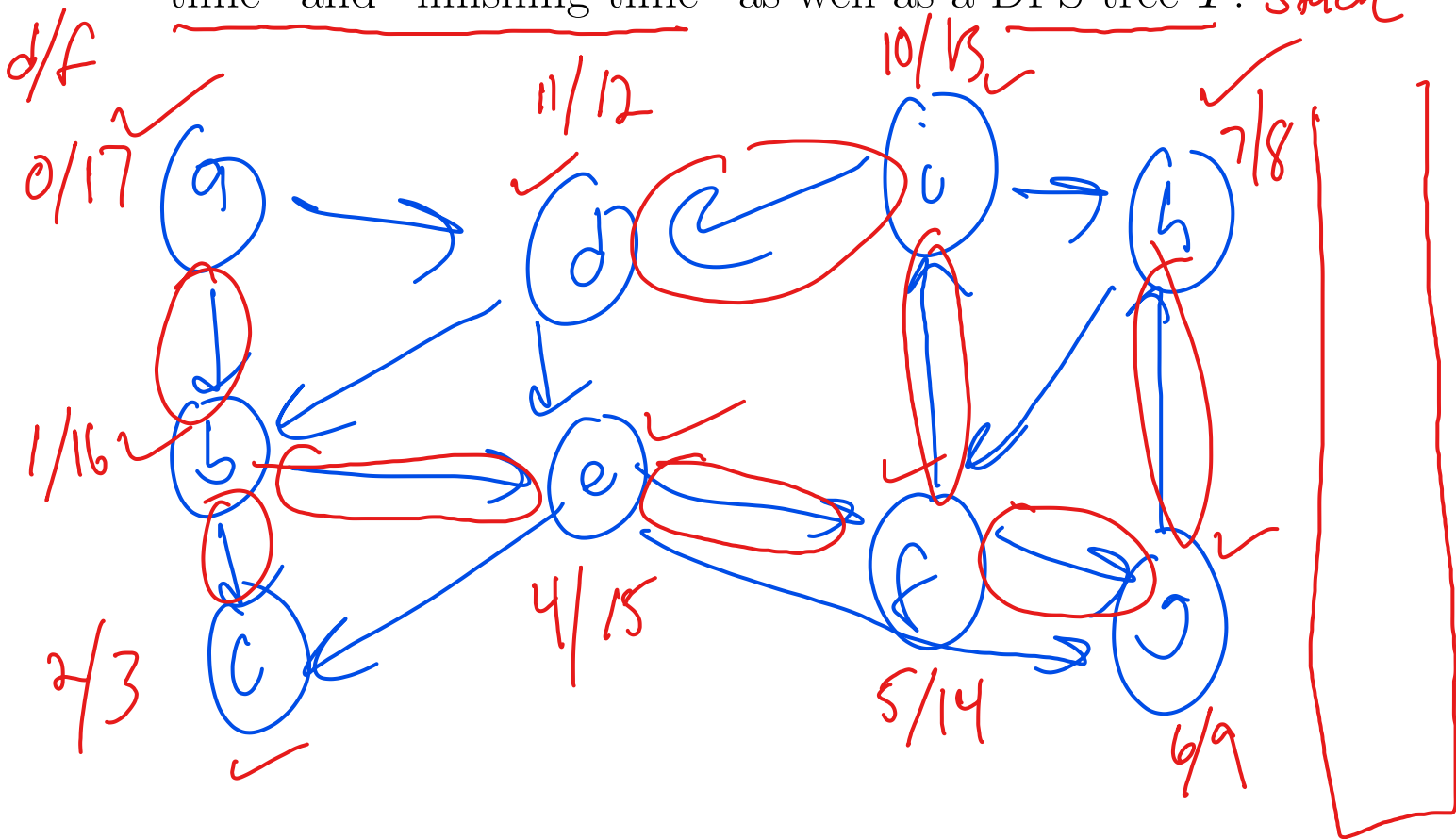


Running time of BFS:

- Each vertex is marked as visited at most once and enqueued/dequeued at most once ($O(n)$ time).
- Each edge is “checked” to see if the neighbor has been already visited twice ($O(m)$ time).

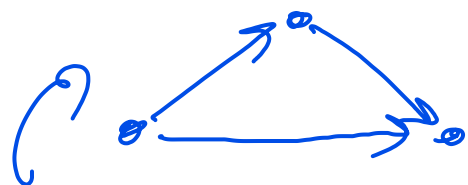
Therefore the running time is $O(n+m) = O(|V|+|E|)$.

Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a “discovery time” and “finishing time” as well as a DFS tree T . *Stack*



Given a
Graph $G=(V,E)$:

- A path is sequence of vertices v_1, \dots, v_k such that $(v_i, v_{i+1}) \in E$ for each $i \in \{1, \dots, k-1\}$.
- A path is simple if all vertices are distinct.
- A path is a cycle if $v_1 = v_k$.
- A graph with no cycles is acyclic.
 - Undirected graphs with no cycles are called forests.
 - Directed acyclic graphs are called DAGs.



this is a DAG

DAGs commonly occur in some applications. For example if the graph is representing precedence between certain objects (e.g. course prerequisites).

Motivated by this, we may be interested in computing a linear ordering of the vertices so that all of the edges “go to the right” (courses left to right that can be taken in order without violating prerequisites).

Such an ordering is called a topological sort.

One way of obtaining a topological sort is to perform a DFS and then sort the nodes in decreasing order according to finishing time.

Another topological sort algorithm:

Store all nodes of in-degree 0 in a queue Q .

While Q is not empty:

Dequeue a vertex v and append it to the sort.

Decrease in-degree for all vertices adjacent to v .

Enqueue all vertices whose in-degree is now 0.

$O(n+m)$ time.

