CMSC 451: Lecture 2
Graph Basics and Depth-First Search

Graphs and Digraphs: A graph $G = (V, E)$ is a structure that represents a discrete set V objects, called *vertices* or *nodes*, and a set of pairwise relations E between these objects, called *edges*. Edges may be *directed* from one vertex to another or may be *undirected*. The term "graph" means an undirected graph, and directed graphs are often called *digraphs* (see Fig. 1). Graphs and digraphs provide a flexible mathematical model for numerous application problems involving binary relationships between a discrete collection of object. Examples of graph applications include *communication* and *transportation networks*, *social networks*, *logic circuits*, *surface meshes* used for shape description in computer-aided design and geographic information systems, *precedence constraints* in scheduling systems.

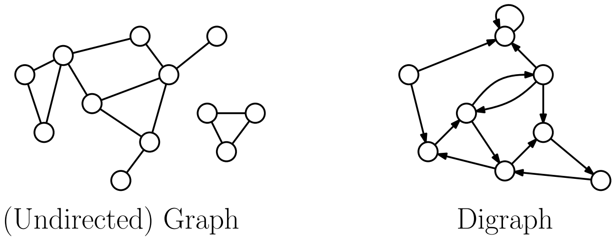


Fig. 1: Graphs and digraphs.

Definition: An *undirected graph* (or simply *graph*) $G = (V, E)$ consists of a finite set V and a set E of *unordered pairs* of distinct vertices.

Definition: A *directed graph* (or *digraph*) $G = (V, E)$ consists of a finite set V and a set E of *ordered pairs* of vertices.

Observe that multiple edges between the same two vertices are not allowed, but in a directed graph, it is possible to have two oppositely directed edges between the same pair of vertices. For undirected graphs, *self-loop* edges are not allowed, but they are allowed for directed graphs. Directed graphs and undirected graphs are different objects mathematically. Certain notions (such as path) are defined for both, but other notions (such as connectivity and spanning trees) may be defined only for one.

Graph and Digraph Terminology: Given an edge $e = (u, v)$ in a digraph, we say that u is the *origin* of e and v is the *destination* of e. Given an edge $e = \{u, v\}$ in an undirected graph, u and v are called the *endpoints* of e. The edge e is *incident* on (meaning that it touches) both u and v. Given two vertices in a graph or digraph, we say that vertex v is *adjacent* to vertex u if there is an edge $\{u, v\}$ (for graphs) or (u, v) (for digraphs).

In a digraph, the number of edges coming out of v is called its *out-degree*, denoted out-deg(v), and the number of edges coming in is called its *in-degree*, denoted in-deg(v). In an undirected graph we just talk about the *degree* of a vertex as the number of incident edges, denoted deg(v).

When discussing the size of a graph, we typically consider both the number of vertices and the number of edges. The number of vertices is typically written as n, and the number of edges is written as m. (**Beware:** There are many different conventions. The number of vertices may be expressed as n, v, V, or $|V|$, and the number of edges may be expressed as m, e, E, or $|E|$.

Here are some basic combinatorial facts about graphs and digraphs. We will leave the proofs to you. Given a graph with n vertices and m edges then:

In a graph:

Number of edges: $0 \leq m \leq \binom{n}{2} = n(n-1)/2 = O(n^2)$.

Sum of degrees: $\sum_{v \in V} \deg(v) = 2m$.

In a digraph:

Number of edges: $0 \leq m \leq n^2$.

Sum of degrees: $\sum_{v \in V} \text{in-deg}(v) = \sum_{v \in V} \text{out-deg}(v) = m$.

Notice that generally the number of edges in a graph may be as large as quadratic in the number of vertices. However, the large graphs that arise in practice typically have much fewer edges. A graph is said to be *sparse* if m is $O(n)$, and *dense*, otherwise. When giving the running times of algorithms, we will usually express it as a function of both n and m, so that the performance on sparse and dense graphs will be apparent.

Paths and Cycles: A *path* in a graph or digraph is a sequence of vertices $\langle v_0, \ldots, v_k \rangle$ such that (v_{i-1}, v_i) is an edge for $i = 1, \ldots, k$. The *length* of the path is the number of edges, k. A path is *simple* if all vertices and all the edges are distinct. A *cycle* is a path containing at least one edge and for which $v_0 = v_k$. A cycle is *simple* if its vertices (except v_0 and v_k) are distinct, and all its edges are distinct.

A graph or digraph is said to be *acyclic* if it contains no simple cycles. An acyclic connected graph is called a *free tree* or simply *tree* for short (see Fig. 2). (The term "free" is intended to emphasize the fact that the tree has no root, in contrast to a *rooted tree*, as is usually seen in data structures.) An acyclic undirected graph (which need not be connected) is a collection of free trees, and is called a *forest*. An acyclic digraph is called a *directed acyclic graph*, or *DAG* for short (see Fig. 2).

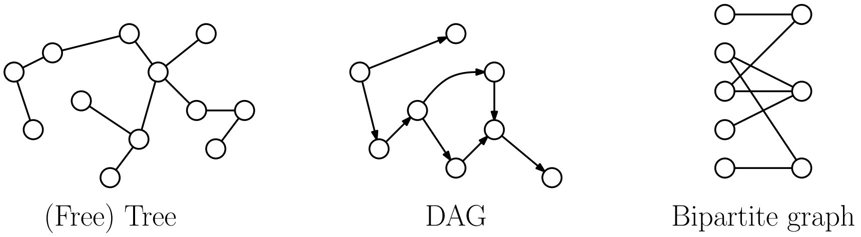


Fig. 2: Illustration of common graph terms.

A *bipartite graph* is one in which the vertices of a graph can be partitioned into two disjoint subsets, denoted V_1 and V_2, such that all the edges have one endpoint in V_1 and one in V_2 (see Fig. 2). Note that every cycle in a bipartite graph contains an even number of edges.

We say that w is *reachable* from u if there is a path from u to w. Note that every vertex is reachable from itself by a trivial path that uses zero edges. An undirected graph is *connected* if every vertex can reach every other vertex. (Connectivity is a bit messier for digraphs, and we will define it later.) The subsets of mutually reachable vertices partition the vertices of the graph into disjoint subsets, called the *connected components* of the graph. In digraphs the notion of reachability is a bit different, because it is possible for u to reach w but not vice versa. A digraph is said to be *strongly connected* if for each u and w, there is a path from u to w and a path from w to u.

Representations of Graphs and Digraphs: There are two common ways of representing graphs and digraphs. First we show how to represent digraphs. Let $G = (V, E)$ be a digraph with $n = |V|$ and let $m = |E|$. We will assume that the vertices of G are indexed $\{1, 2, \ldots, n\}$.

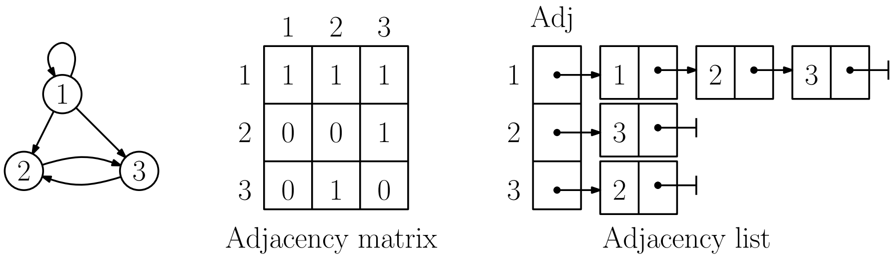
Adjacency Matrix: An $n \times n$ matrix defined for $1 \leq v, w \leq n$.

$$A[v, w] = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise.} \end{cases}$$

(See Fig. 3.) If the digraph has weights we can store the weights in the matrix. For example if $(v, w) \in E$ then $A[v, w] = W(v, w)$ (the weight on edge (v, w)). If $(v, w) \notin E$ then generally $W(v, w)$ need not be defined, but often we set it to some "special" value, e.g. $A(v, w) = -1$, or ∞. (By ∞ we mean some number which is larger than any allowable weight.)

It might come as a surprise, but there are a number of interesting relationships between the use of matrices to represent graphs and the matrices that arise in linear algebra to represent linear transformations. For example, the eigenvalues of the adjacency matrix of a graph provide a lot of information about the structure of the graph.

Adjacency List: An array $Adj[1 \ldots n]$ of pointers where for $1 \leq v \leq n$, $Adj[v]$ points to a list (e.g., a singly or doubly linked list) containing the vertices that are adjacent to v (i.e., the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements (see Fig. 3).



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

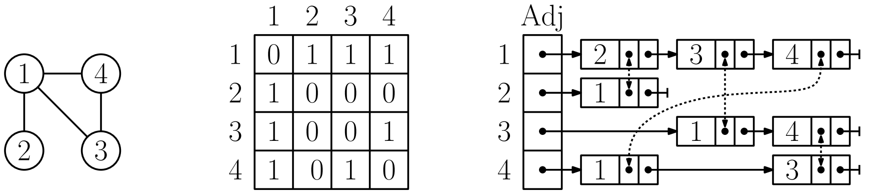
Adjacency matrix

Adjacency list

Fig. 3: Adjacency matrix and adjacency list for digraphs.

We can represent undirected graphs using exactly the same representation, but we will store each edge twice. In particular, we representing the undirected edge $\{v, w\}$ by the two oppositely directed edges (v, w) and (w, v) (see Fig. 4). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

This can cause some complications. For example, suppose you write an algorithm that operates by marking edges of a graph. You need to be careful when you mark edge (v, w) in the representation that you also mark (w, v), since they are both the same edge in reality. When dealing with adjacency lists, it may not be convenient to walk down the entire linked list, so it is common to include *cross links* between corresponding edges.



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

Adjacency matrix

Adjacency list (with crosslinks)

Fig. 4: Adjacency matrix and adjacency list for graphs.

An adjacency matrix requires $\Theta(n^2)$ storage, and an adjacency list requires $\Theta(n+m)$ storage. The n arises because there is one entry for each vertex in *Adj*. Since each list has out-deg(v) entries, when this is summed over all vertices, the total number of adjacency list records is $\Theta(m)$. For most applications, the adjacency list representation is standard.

Depth-First Search: One of the most important basic operations on a graph is to systematically visit all its vertices. These traversals natually impose a type of tree structure (or generally a forest) on the graph, and trees are usually much easier to reason about than general graphs.

We are given a graph $G = (V, E)$, which may be directed or undirected. We employ four auxiliary arrays. To avoid revisiting the same vertex, we maintain a mark for each vertex: undiscovered, discovered, finished. Additional information can be stored as part of the traversal process:

Discovery time: $d[u]$ indicates the time when vertex u was discovered, which coincides with the moment that the DFS process is started at this vertex.

Finish time: $f[u]$ indicates the time when vertex u is finished processing. At this point, all of u's neighboring nodes have been visited, and indeed, everything reachable from u has been discovered and possibly finished.

Predecessor pointer: $p[u]$ indicates the vertex that discovered u. Each edge of the form $(p[u], u)$ is a tree edge in the DFS recursion tree.

DFS induces a tree structure. In order to handle instances where not all vertices are reachable from the starting vertex, we include a main program that invokes DFS whenever an undiscovered vertex is encountered. The main program is shown in code block below and the recursive DFSvisit function is shown in the next code block. (Fig. 5 illustrates the execution on an undirected graph, and Fig. 6 shows an example on a directed graph.)

Depth-First Search (Main Program)

```
DFS(G) {                                        // main program
    time = 0
    for each (u in V)                           // initialization
        mark[u] = undiscovered

    for each (u in V)
        if (mark[u] == undiscovered)            // undiscovered vertex?
            DFSVisit(u)                         // ...start a new search here
}
```

DFS Visit (Process a single node)

```
DFSVisit(u) {                                   // perform a DFS search at u
    mark[u] = discovered                        // u has been discovered
    d[u] = ++time
    for each (v in Adj(u)) {
        if (mark[v] == undiscovered) {          // undiscovered neighbor?
            pred[v] = u
            DFSVisit(v)                         // ...visit it
        }
    }
    mark[u] = finished                          // we're done with u
    f[u] = ++time
}
```

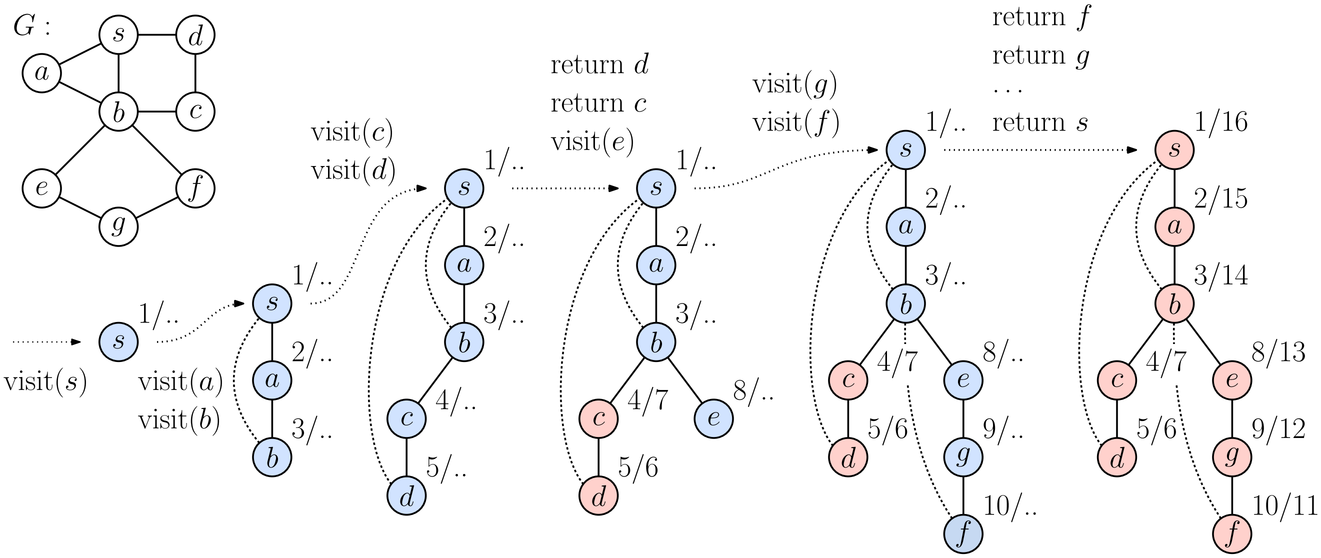


Fig. 5: Depth-first search on an undirected graph. (Blue nodes are discovered, and pink nodes are finished. Each node u is labeled with the values $d[u]/f[u]$.)

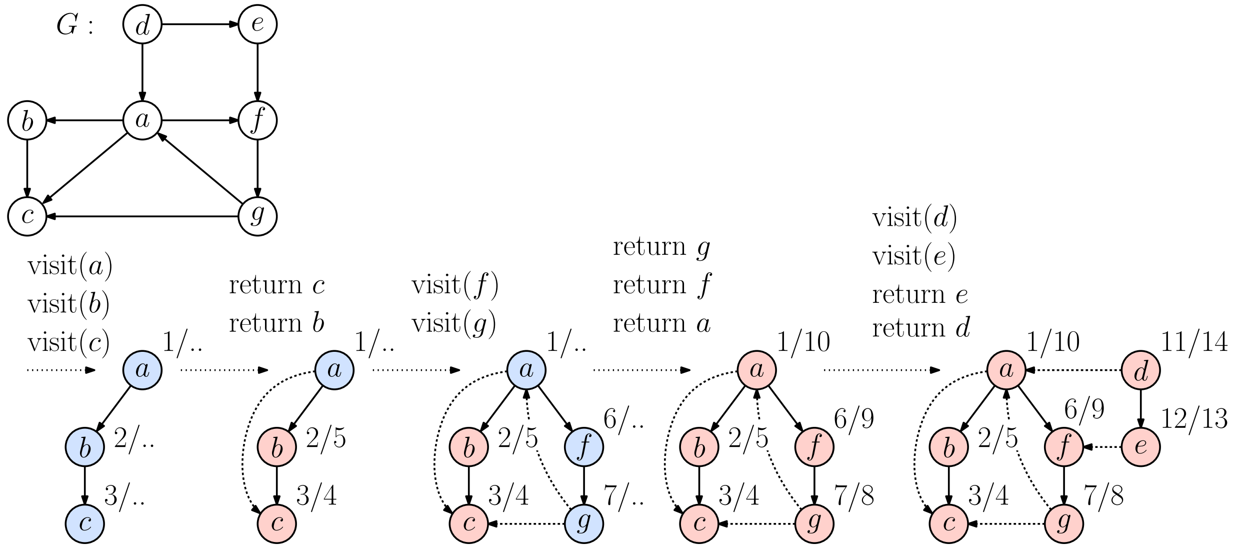


Fig. 6: Depth-first search on a directed graph. (Blue nodes are discovered, and pink nodes are finished. Each node u is labeled with the values $d[u]/f[u]$.)

Analysis: The running time of DFS is $O(n + m)$. We'll do the analysis for undirected graphs. First observe that if we ignore the time spent in the recursive calls, the main DFS procedure runs in $O(n)$ time. Each vertex is visited exactly once in the search, and hence the call `DFSVisit()` is made exactly once for each vertex. We can just analyze each one individually and add up their running times. Ignoring the time spent in the recursive calls, we can see that each vertex u can be processed in $O(1 + \deg(u))$ time (the "+1" is needed in case the degree is 0). Thus the total time used in the procedure is

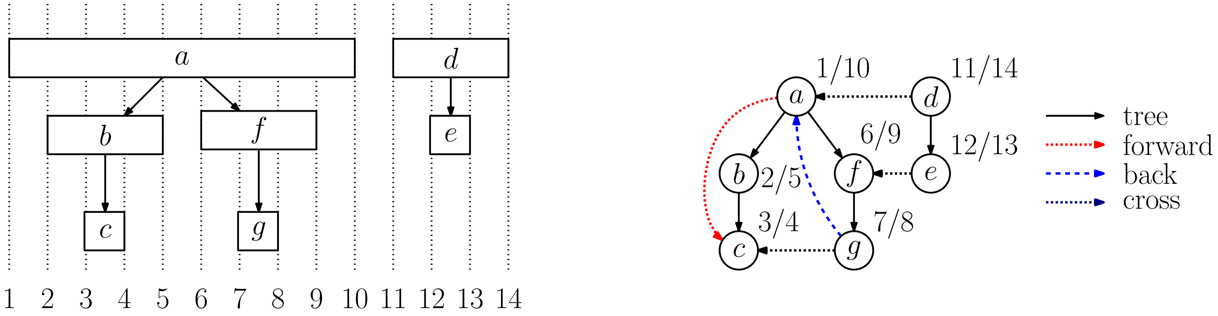
$$T(n) \;=\; n + \sum_{u \in V} (1 + \deg(u)) \;=\; n + \left(\sum_{u \in V} \deg(u) \right) + n \;=\; 2n + m \;=\; O(n + m).$$

A similar analysis holds if we consider DFS for digraphs.

Parenthesis Lemma and Edge Types: DFS naturally imposes a tree structure (actually a collection of trees, or a forest) on the structure of the graph. This is just the recursion tree, where the edge (u, v) arises when processing vertex u we call `DFSVisit(v)` for some neighbor v. The hierarchical structure naturally imposes a nesting structure on the discovery-finish time intervals. This is described in the following lemma (and illustrated in Fig. 7(a)).

Lemma: (Parenthesis Lemma) Given a graph $G = (V, E)$ (directed or undirected), and any DFS tree for G and any two vertices $u, v \in V$:

- u is a descendant of v iff $[d[u], f[u]] \subseteq [d[v], f[v]]$.
- u is an ancestor of v iff $[d[u], f[u]] \supseteq [d[v], f[v]]$.
- u and v are unrelated (in terms of ancestor/descendant) iff $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.



(a) (b)

Fig. 7: (a) the Parenthesis Lemma and (b) the DFS edge types.

The structure of the remaining (non-tree) edges of the graph depend on the type of graph involved. For **undirected graphs**, the remaining edges are called *back edges*. An important observation is that for each back edge (u, v), u is either a proper ancestor or a proper descendant of v. To see why, consider any non-tree edge (u, v). Since the graph is undirected, we may assume without loss of generality that u was discovered before v. By the parenthesis lemma, this means either that u is an ancestor of v (and we are done) or that their discovery-finish intervals are disjoint. If they are disjoint, u must finish before v is discovered. However, this is impossible, because as we are processing u, we will see the edge (u, v) and thus discover v.

For **directed graphs** the non-tree edges of the graph can be classified as follows (See Fig. 7(b)):

Back edges: (u, v) where v is a (not necessarily proper) ancestor of u in the tree. (Thus, a self-loop edge is considered to be a back edge.)

Forward edges: (u, v) where v is a non-child, proper descendant of u in the tree.

Cross edges: (u, v) where u and v are not ancestors or descendants of one another (in fact, the edge may go between different trees of the forest).

It is not difficult to classify the edges of a DFS tree on-the-fly by analyzing the vertex status (undiscovered, discovered, finished) and/or considering the time stamps. (This is left as an exercise.)[1]

[1] Be careful, however. Remember that in an undirected graph, every edge is represented twice. When classifying back edges, you should be sure that you are not seeing the other half of a tree edge.