Template for CS161 Graphs, DFS and Topological Sort
Scribes: Juliana Cook (2015), Virginia **Date:** April 29, 2015

1 Graphs

A **graph** is a set of **vertices** and **edges** connecting those vertices. Formally, we define a graph G as $G = (V, E)$ where $E \subseteq V \times V$. For ease of analysis, the variables n and m typically stand for the number of vertices and edges, respectively. Graphs can come in two flavors, **directed** or **undirected**. If a graph is undirected, it must satisfy the property that $(i, j) \in E$ iff $(j, i) \in E$ (aka all edges are bidirectional).

1.1 Representation

A common issue is the topic of how to represent a graph's edges in memory. There are two standard methods for this task.

An **adjacency matrix** uses an arbitrary ordering of the vertices from 1 to $|V|$. The matrix consists of an $n \times n$ binary matrix such that the $(i, j)^{th}$ element is 1 if (i, j) is an edge in the graph, 0 otherwise.

An **adjacency list** consists of an array A of $|V|$ lists, such that $A[u]$ contains a linked list of vertices v such that $(u, v) \in E$ (the neighbors of u). In the case of a directed graph, it's also helpful to distinguish between outgoing and ingoing edges by storing two different lists at $A[u]$: a list of v such that $(u, v) \in E$ (the out-neighbors of u) as well as a list of v such that $(v, u) \in E$ (the inneighbors of u).

What are the tradeoffs between these two methods? To help our analysis, let $deg(v)$ denote the **degree** of v, or the number of vertices connected to v. In a directed graph, we can distinguish between out-degree and in-degree, which respectively count the number of outgoing and incoming edges.

- The adjacency matrix can check if (i, j) is an edge in G in constant time, whereas the adjacency list representation must iterate through up to $deg(i)$ list entries.
- The adjacency matrix takes $\Theta(n^2)$ space, whereas the adjacency list takes $\Theta(m + n)$ space.
- The adjacency matrix takes $\Theta(n)$ operations to enumerate the neighbors of a vertex v since it must iterate across an entire row of the matrix. The adjacency list takes $\deg(v)$ time.

What's a good rule of thumb for picking the implementation? One useful property is the sparsity of the graph's edges. If the graph is **sparse**, and the number of edges is considerably less than the max ($m \ll n^2$), then the adjacency list is a good idea. If the graph is **dense** and the number of edges is nearly n^2, then the matrix representation makes sense because it speeds up lookups without too much space overhead. Of course, some applications will have lots of space to spare, making the matrix feasible no matter the structure of the graphs. Other applications may prefer adjacency lists even for dense graphs. Choosing the appropriate structure is a balancing act of requirements and priorities.

2 Depth First Search (DFS)

Given a starting vertex, it's desirable to find all vertices reachable from the start. There are many algorithms to do this, the simplest of which is depth-first search. As the name implies, DFS enumerates the deepest paths, only backtracking when it hits a dead end or an already-explored section of the graph. DFS by itself is fairly simple, so we introduce some augmentations to the basic algorithm.

- To prevent loops, DFS keeps track of a "color" attribute for each vertex. Unvisited vertices are white by default. Vertices that have been visited but still may be backtracked to are colored gray. Vertices which are completely processed are colored black. The algorithm can then prevent loops by skipping non-white vertices.

- Instead of just marking visited vertices, the algorithm also keeps track of the tree generated by the depth-first traversal. It does so by marking the “parent” of each visited vertex, aka the vertex that DFS visited immediately prior to visiting the child.
- The augmented DFS also marks two auto-incrementing timestamps d and f to indicate when a node was first discovered and finished.

The algorithm takes as input a start vertex s and a starting timestamp t, and returns the timestamp at which the algorithm finishes. Let $N(s)$ denote the neighbors of s.

Algorithm 1: init(G)

```
foreach $v \in G$ do
    color($v$) $\leftarrow$ white
    d($v$), f($v$) $\leftarrow \infty$
    p($v$) $\leftarrow$ nil
```

Algorithm 2: DFS(s, t) $s \in V$. t = time, s is white

```
color($s$) $\leftarrow$ grey
\\ discovery time
d($s$) $\leftarrow t$
$t$++;
\\ $N_{out}(s)$ for directed $G$
foreach $v \in N(s)$ do
    if color(v) = white then
        p($v$) $\leftarrow s$
        \\ finish time of DFS
        $t \leftarrow$ DFS($v, t$)
        $t$++
\\ finish time
f($s$) $\leftarrow t$
\\ $s$ is finished
color($s$) $\leftarrow$ black
return f($s$)
```

There are multiple ways we can search using DFS. One way is to search from some source node s, which will give us a set of black nodes reachable from s and white nodes unreachable from s.

Algorithm 3: DFS(s) \\ DFS from a source node s

```
init($G$)
DFS($S$, 1)
```

Another way to use DFS is to search over the entire graph, choosing with some white node and finding everything we can reach from that node, and repeating until we have no white nodes remaining. In an undirected graph this will give us all of the connected components.

2.1 Runtime of DFS

We will now look at the runtime for the standard DFS algorithm (Algorithm 2).

Algorithm 4: DFS(G) \\ DFS on an entire graph G

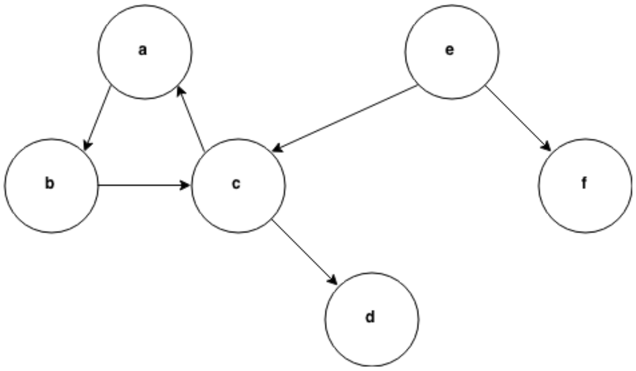
```
init($G$);
$t \leftarrow 1$
foreach $v \in G$ do
    if $color(v) = white$ then
        $t \leftarrow$ DFS($v$, $t$)
        $t$++
```

Everything above the loop runs in $O(1)$ time per node visit. Excluding the recursive call, everything inside of the for loop takes $O(1)$ time every time an edge is scanned. Everything after the for loop also runs in $O(1)$ time per node visit.

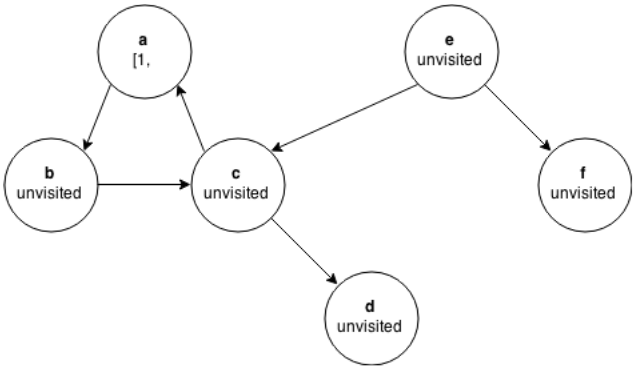
We can express the runtime of DFS as $O(\#$ of node visits $+ \#$ of edge scans$)$. Assume we have a graph with n nodes and m edges. We know that the # of node visits is $\leq$ n, since we only visit white nodes and whenever we visit a node we change its color from white to grey and never change it back to white again. We also know that an edge (u, v) is scanned only when u or v is visited. Since every node is visited at most once, we know that an edge (u, v) is scanned at most twice (or only once for directed graphs). Thus, # of edges scanned is $O(m)$, and the overall runtime of DFS is $O(m + n)$.

2.2 DFS Example

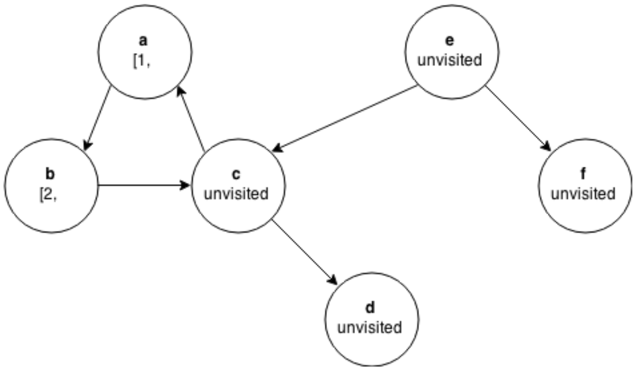
We will now try running DFS on the example graph below.



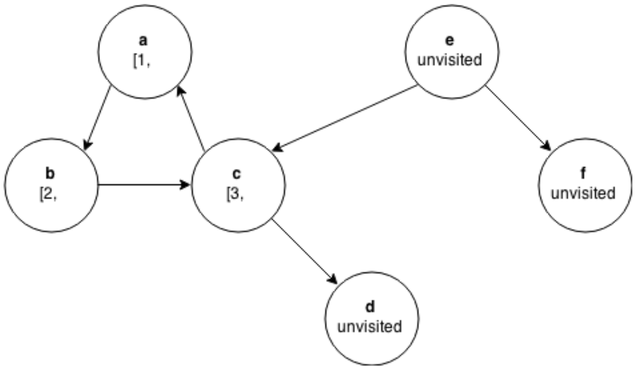
We mark all of the nodes as unvisited and start at a white node, in our case node a.



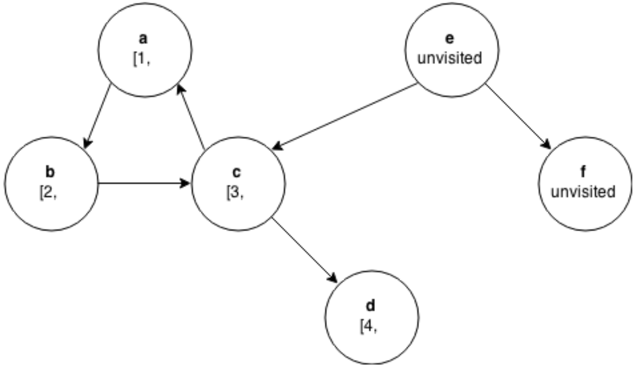
From node a we will visit all of a's children, namely node b.



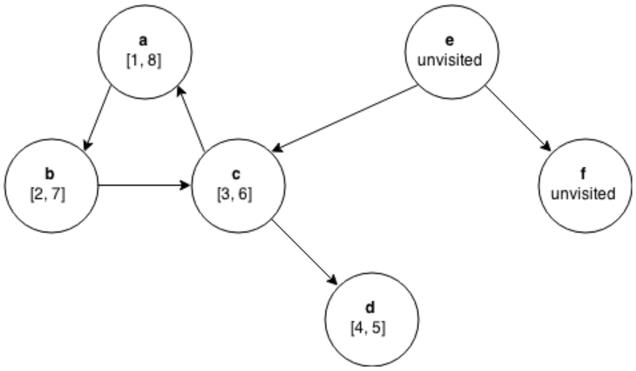
We now visit b's child, node c.



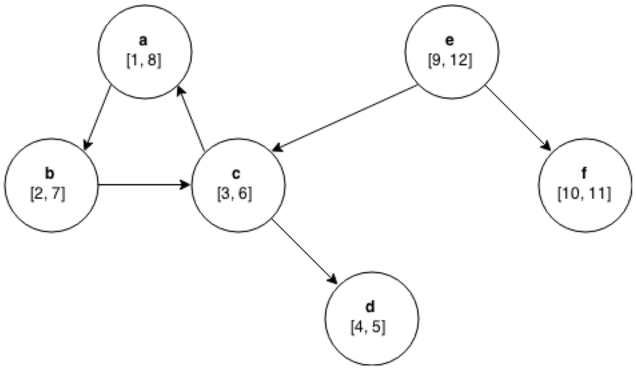
Node c has two children that we must visit. When we try to visit node a we find that node a has already been visited (and would be colored grey, as we are in the process of searching a's children), so we do not continue searching down that path. We will next search c's second child, node d.



Since node d has no children, we return back to its parent node, c, and continue to go back up the path we took, marking nodes with a finish time when we have searched all of their children.



Once we reach our first source node a we find that we have searched all of its children, so we look in the graph to see if there are any unvisited nodes remaining. For our example, we start with a new source node e and run DFS to completion.



2.3 DFS Trees and Time Interval Properties

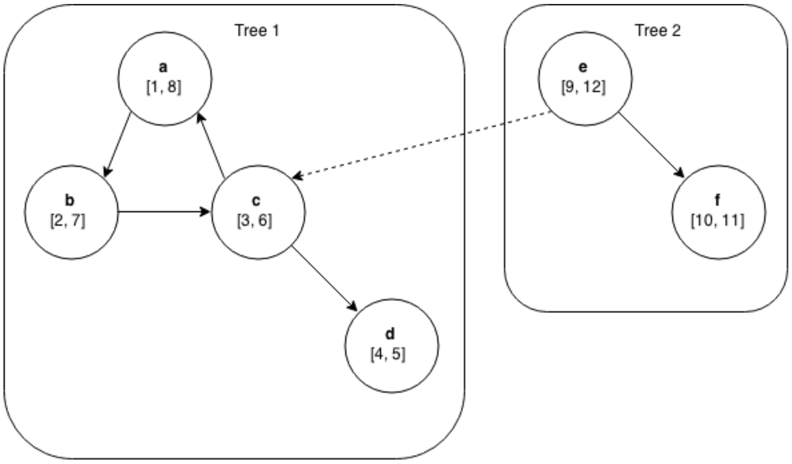
Last lecture we said that a DFS computes two things: a time interval consisting of a discovery time and a finish time that defines when the node was active (when its color is grey), and a DFS tree.

2.3.1 DFS Trees

DFS trees are defined by the edges created from a parent node to the visited child node.

DFS(s, t) computes a tree rooted at s defined by the edges $(p(x), x)$ for x reachable from S (i.e. nodes discovered while S was gray).

Running DFS on an entire graph G will create a DFS forest. Our earlier example of DFS on a graph resulted in the following DFS forest with two trees.



Note that even if some node w is reachable from v in a directed graph, v may not be connected to w in the DFS forest. We can see an instance of this above with nodes e and c. Even though node e is connected to node c, the two are not in the same DFS tree because of the source node we started with.

2.3.2 Time Interval Properties

We will now explore some of the properties of time intervals created by running DFS on a graph.

1. If u is a descendant of v in the DFS tree rooted at v, then $[d(u), f(u)] \subset [d(v), f(v)]$.
   This means that for every descendant of a node in a DFS tree we have that the child's time interval is properly contained in the parent's time interval. This makes sense, as we only "finish" a node after we have explored all of its children.

2. If neither u is a descendant of v nor v is a descendant of u, then $[d(u), f(u)] \bigcap [d(v), f(v)] = \emptyset$.
   This means that if two nodes are not connected in a DFS tree, then their intervals don't overlap.

2.3.3 White Path Theorem

Theorem 2.1 (White Path Theorem)**.** *Node u is a descendant of v in the DFS tree rooted at v if and only if when DFS(v, t) was called there was a path in G from v to u using only white nodes.*

Proof of White Path Theorem. We must prove both directions.
($\implies$) If u is a descendant of v, then the path from v to u in the DFS tree must have only had white nodes since we only visit white nodes. ($\impliedby$) Suppose G has a path $v = x_1, x_2, ...x_k = u$ such that all x_i were white when DFS(v) was called. We show that all x_i are visited while v is gray, so all x_i are descendants of v. We prove this claim by induction on i.
Base case: $i = 1$. $x_1 = v$ so x_1 is visited by DFS(v, t).
Suppose x_i is visited. We know that (x_i, x_{i+1}) is scanned. If x_{i+1} is white then DFS(x_{i+1}, t) was called so x_{i+1} visited.
If x_{i+1} was not white, then its color was changed from white to grey within DFS(v, t) (and its recursive calls). We thus know that x_{i+1} is visited and all x_j are visited, so u is visited and is a descendant of v. □

From this theorem, we know

1. DFS(s) colors all nodes reachable from s black.

2. DFS(G) computes the connected components of any undirected graph G.

2.4 Topological Order

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

Definition. The topological order of a DAG G is a permutation π of the vertices such that for every edge (u, v), $\pi(u) < \pi(v)$. In other words, one can use π to order the vertices from left to right such that all edges go from left to right. (Note that a DAG might have many valid topoogical orders.)

Claim 1. *If we run DFS(G) on a DAG G, for all (u, v) we find that $f(v) < f(u)$, so a reverse sorted order by f is a topological order.*

Hence, to compute a topological order of G, run DFS(G) and whenever a node is finished, add it to the front of a running list L.

Proof of Claim 1. There are three cases:

1. (v is a descendent of u in u's DFS tree) By time interval property (1), we get that in this case $f(v) < f(u)$.
2. (v is not a descendant of u and u is not a descendent of v) In this case, v must have been visited before u (otherwise v would have been white when (u, v) was scanned and v would have become a descendent of u, but v is not a descendent of u). Hence, $f(v) < d(u) < f(u)$ (by property 2).
3. (u is a descendent of v) This cannot occur because it would cause a cycle: take the tree path from v to u, followed by edge (u, v). However, our graph is a DAG, which cannot contain a cycle.

□