Notes 4 for CS 170

1 Depth-First Search

We will start by studying two fundamental algorithms for searching a graph: *depth-first search* and *breadth-first search.* To better understand the need for these algorithms, let us imagine the computer's view of a graph that has been input into it: it can examine each of the edges adjacent to a vertex in turn, by traversing its adjacency list; it can also mark vertices as "visited." This corresponds to exploring a dark maze with a flashlight and a piece of chalk. You are allowed to illuminate any corridor of the maze emanating from the current room, and you are also allowed to use the chalk to mark the current room as having been "visited." Clearly, it would not be easy to find your way around without the use of any additional data structures.

Mythology tells us that the right data structure for exploring a maze is a ball of string. Depth-first search is a technique for exploring a graph using as a basic data structure a *stack*—the cyberanalog of a ball of string. It is not hard to visualize why a stack is the right way to implement a ball of string in a computer. A ball of string allows two primitive steps: *unwind* to get into a new room (the stack analog is *push the new room*) and *rewind* to return to the previous room—the stack analog is *pop*.

Actually, we shall present depth-first search not as an algorithm explicitly manipulating a stack, but as a *recursive* procedure, using the stack of activation records provided by the programming language. We start by defining a recursive procedure `explore`. When `explore` is invoked on a vertex v, it explores all previously unexplored vertices that are reachable from v.

```
procedure explore(v:  vertex)
visited(v) := true
previsit(v)
for each edge (v,w) out of v do
  {if not visited(w) then explore(w)}
postvisit(v)

algorithm dfs(G = (V,E): graph)
for each v in V do {visited(v) := false}
for each v in V do
   {if not visited(v) then explore(v)}}
```

`previsit(v)` and `postvisit(v)` are two routines that perform at most some constant number of operations on the vertex v. As we shall see, by varying these routines depth-first search can be tuned to accomplish a wide range of important and sophisticated tasks.

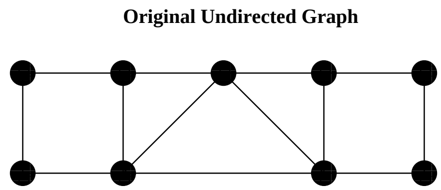
Depth-first search takes $O(n+e)$ steps on a graph with n vertices and e edges, because the algorithm performs *at most some constant number of steps per each vertex and edge of* *the graph.* To see this clearly, let us go through the exercise of "charging" each operation of the algorithm to a particular vertex or edge of the graph. The procedure `explore` (lines 1 through 6) is called at most once—in fact, *exactly* once—for each vertex; the boolean variable `visited` makes sure each vertex is visited only once. The finite amount of work required to perform the call `explore(v)` (such as adding a new activation record on the stack and popping it in the end) is charged to the vertex v. Similarly, the finite amount of work in the routines `previsit(v)` and `postvisit(v)` (lines 3 and 6) is charged to the vertex v. Finally, the work done for each outgoing edge (v, w) (looking it up in the adjacency list of v and checking whether v has been visited, lines 4 and 5) is charged to that edge. Each edge (v, w) is processed only once, because its tail (also called source) v is visited only once—in the case of an undirected graph, each edge will be visited twice, once in each direction. Hence, every element (vertex or edge) of the graph will end up being charged at most some constant number of elementary operations. Since this accounts for all the work performed by the algorithm, it follows that depth-first search takes $O(n + e)$ work—it is a *linear-time algorithm.*

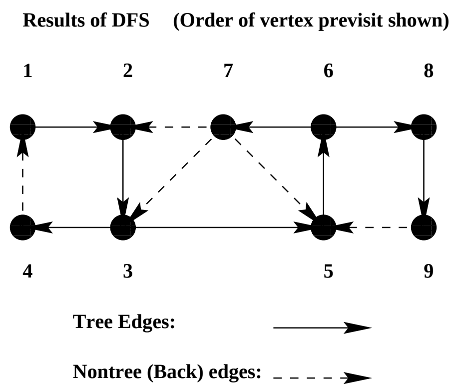
This is the fastest possible running time of any algorithm solving a nontrivial problem, since any decent problem will require that each data item be inspected at least once.[1] It means that a graph can be searched in time comparable to the time it takes to read it into the memory of a computer!

Here is an example of DFS on an undirected graph, with vertices labeled by the order in which they are first visited by the DFS algorithm. DFS defines a tree in a natural way: each time a new vertex, say w, is discovered, we can incorporate w into the tree by connecting w to the vertex v it was discovered from via the edge (v, w). If the graph is disconnected, and thus depth-first search has to be restarted, a separate tree is created for each restart. The edges of the depth-first search tree(s) are called *tree edges;* they are the edges through which the control of the algorithm flows. The remaining edges, shown as broken lines, go from a vertex of the tree to an *ancestor* of the vertex, and are therefore called *back edges.* If there is more than one tree, the collection of all of them is called a *forest.*

---

[1]There is an exception to this, namely *database queries*, which can often be answered without examining the whole database (for example, by binary search). Such procedures, however, require data that have been preprocessed and structured in a particular way, and so cannot be fairly compared with algorithms, such as depth-first search, which must work on unpreprocessed inputs.





By modifying the procedures `previsit` and `postvisit`, we can use depth-first search to compute and store useful information and solve a number of important problems. The simplest such modification is to record the "time" each vertex is first seen by the algorithm, i.e., produce the labels on the vertices in the above picture. We do this by keeping a counter (or *clock*), and assigning to each vertex the "time" `previsit` was executed (`postvisit` does nothing). This would correspond to the following code:

```
procedure previsit(v: vertex)
pre(v) := clock++
```

Naturally, `clock` will have to be initialized to zero in the beginning of the main algorithm. If we think of depth-first search as using an explicit stack, pushing the vertex being visited next on top of the stack, then `pre(v)` is the order in which vertex v is first pushed on the stack. The contents of the stack at any time yield a path from the root to some vertex in the depth first search tree. Thus, the tree summarizes the history of the stack during the algorithm.

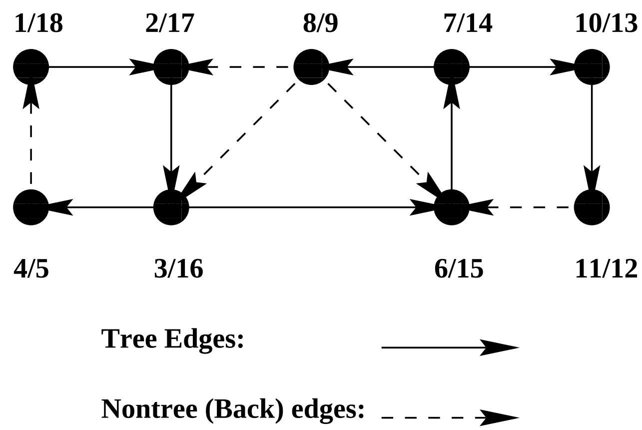
More generally, we can modify both `previsit` and `postvisit` to increment `clock` and record the time vertex v was first visited, or pushed on the stack (`pre(v)`), and the time vertex v was last visited, or popped from the stack (`post(v)`):

```
procedure previsit(v: vertex)
pre(v) := clock++

procedure postvisit(v: vertex)
post(v) := clock++
```

Let us rerun depth-first search on the undirected graph shown above. Next to each vertex v we showing the numbers `pre(v)/post(v)` computed as in the last code segment, next to each vertex v.

Results of DFS (pre(v)/post(v) shown next to each vertex x)



How would the nodes be numbered if `previsit()` did not do anything?

Notice the following important property of the numbers `pre(v)` and `post(v)` (when both are computed by `previsit` and `postvisit`: Since `[pre[v],post[v]]` is essentially the time interval during which v stayed on the stack, it is always the case that

- *two intervals* `[pre[u],post[u]]` *and* `[pre[v],post[v]]` *are either disjoint, or one contains another.*

  In other words, any two such intervals cannot partially overlap. We shall see many more useful properties of these numbers.

One of the simplest feats that can accomplished by appropriately modifying depth-first search is to subdivide an undirected graph to its *connected components*, defined next. A *path* in a graph $G = (V, E)$ is a sequence $(v_0, v_1, \ldots, v_n)$ of vertices, such that $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, n$. An undirected graph is said to be *connected* if there is a path between any two vertices.[2] If an undirected graph is disconnected, then its vertices can be partitioned into *connected components*. We can use depth-first search to tell if a graph is connected, and, if not, to assign to each vertex v an integer, `ccnum[v]`, identifying the connected component of the graph to which v belongs. This can be accomplished by adding a line to `previsit`.

```
procedure previsit(v:  vertex)
pre(v) := clock++
ccnum(v) := cc
```

Here `cc` is an integer used to identify the various connected components of the graph; it is initialized to zero just after line 7 of dfs, and increased by one just before each call of `explore` at line 10 of dfs.

[2]As we shall soon see, connectivity in directed graphs is a much more subtle concept.

2 Applications of Depth-First Searching

We can represent the tasks in a business or other organization by a *directed graph*: each vertex u represents a task, like manufacturing a widget, and each directed edge (u, v) means that task u must be completed before task v can be done (such as shipping a widget to a customer). The simplest question one can ask about such a graph is: *in what orders can I complete the tasks, satisfying the precedence order defined by the edges, in order to run the business?* We say "orders" rather than "order" because there is not necessarily a unique order (can you give an example?) Furthermore it may be that no order is legal, in which case you have to rethink the tasks in your business (can you give an example?). In the language of graph theory, answering these questions involves *computing strongly connected components* and *topological sorting.*

Here is a slightly more difficult question that we will also answer eventually: suppose each vertex is labeled by a positive number, which we interpret as the time to execute the task it represents. Then what is the minimum time to complete all tasks, assuming that independent tasks can be executed simultaneously?

The tool to answer these questions is DFS.

3 Depth-first Search in Directed Graphs

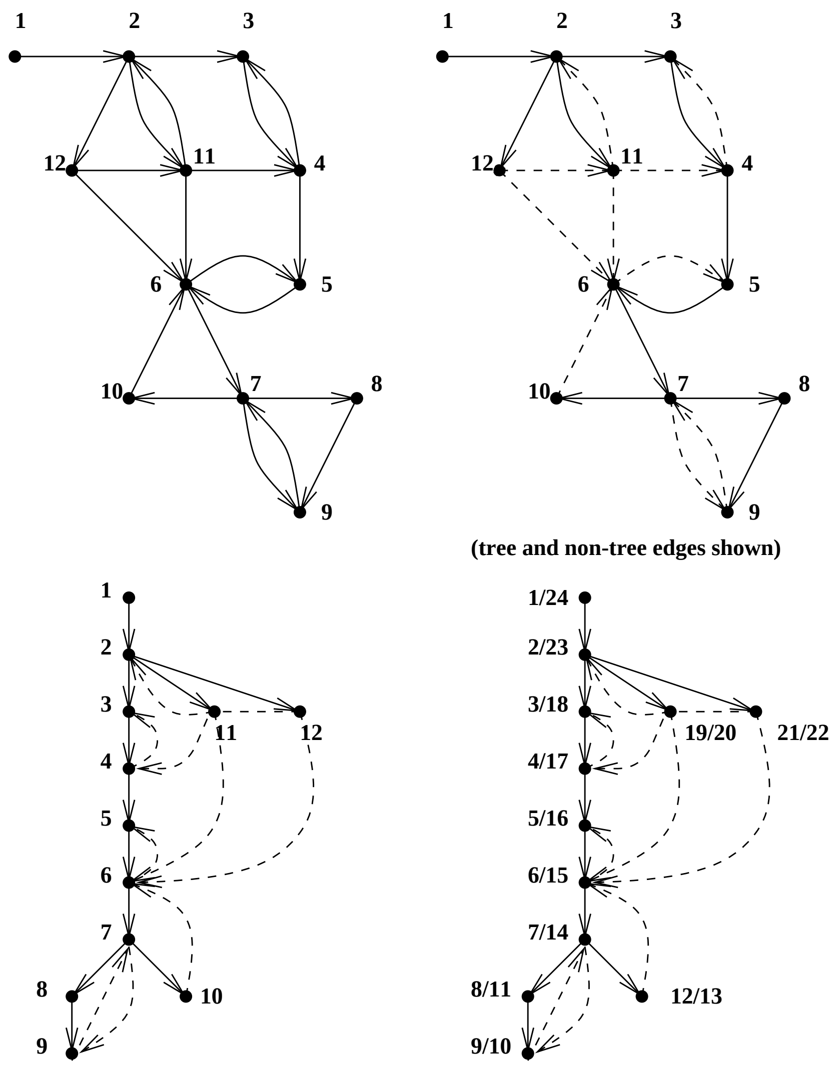
We can also run depth-first search on a directed graph, such as the one shown below.

The resulting `pre[v]/post[v]` numbers and depth-first search tree are shown. Notice, however, that not all non-tree edges of the graph are now back edges, going from a vertex in the tree to an ancestor. The non-tree edges of the graph can be classified into three types:

- Forward edges: these go from a vertex to a descendant (other than child) in the depth-first search tree. You can tell such an edge (v, w) because `pre[v]` $<$ `pre[w]`.
- Back edges: these go from a vertex to an ancestor in the depth-first search tree. You can tell such an edge (v, w) because, at the time it is traversed, `pre[v]` $>$ `pre[w]`, and `post[w]` is undefined.
- Cross edges: these go from "right to left," from a newly discovered vertex to a vertex that lies in a part of the tree whose processing has been concluded. You can tell such an edge (v, w), because, at the time it is traversed, `pre[v]` $>$ `pre[w]`, and `post[w]` is defined. Can there be cross edges in an undirected graph?

Why do all edges fall into one of these four categories (tree, forward, back, cross)?

DFS of a Directed Graph



(tree and non-tree edges shown)

(tree and non-tree edges shown) **(pre/post numbers shown)**

4 Directed Acyclic Graphs

A *cycle* in a directed graph is a path $(v_0, v_1, \ldots, v_n)$ such that (v_n, v_0) is also an edge. A directed graph is *acyclic*, or a *dag*, if it has no cycles. See CLR page 88 (CLRS Appendix B.4) for a more careful definition.

CLAIM 1
A directed graph is acyclic if and only if depth-first search on it discovers no backedges.

PROOF: If (u, v) is a backedge, then (u, v) together with the path from v to u in the depth-first search tree form a cycle.

Conversely, suppose that the graph has a cycle, and consider the vertex v on the cycle assigned the lowest `pre[v]` number, i.e., v is the first vertex on the cycle that is visited.

Then all the other vertices on the cycle must be descendants of v, because there is path from v to each of them. Let (w, v) be the edge on the cycle pointing to v; it must be a backedge. □

It is therefore very easy to use depth-first search to see if a graph is acyclic: Just check that no backedges are discovered. But we often want more information about a dag: We may want to *topologically sort it.* This means to order the vertices of the graph from top to bottom so that all edges point downward. (*Note:* This is possible if and only if the graph is acyclic. Can you prove it? One direction is easy; and the topological sorting algorithm described next provides a proof of the other.) This is interesting when the vertices of the dag are tasks that must be scheduled, and an edge from u to v says that task u must be completed before v can be started. The problem of topological sorting asks: in what order should the tasks be scheduled so that all the precedence constraints are satisfied. The reason having a cycle means that topological sorting is impossible is that having u and v on a cycle means that task u must be completed before task v, and task v must be completed before task u, which is impossible.

To topologically sort a dag, we simply do a depth-first search, and then arrange the vertices of the dag in decreasing `post[v]`. That this simple method correctly topologically sorts the dag is a consequence of the following simple property of depth-first search:

Lemma 2
For each edge (u, v) *of* G, `post(u)` $<$ `post(v)` *if and only if* (u, v) *is a back edge.*
Proof: Edge (u, v) must either be a tree edge (in which case descendent v is popped before ancestor u, and so `post(u)`$>$`post(v)`), or a forward edge (same story), or a cross edge (in which case u is in a subtree explored after the subtree containing v is explored, so v is both pushed and popped before u is pushed and popped, i.e., `pre(v)` and `post(v)` are both less `pre(u)` and `post(u)`), or a back edge (u is pushed and popped after v is pushed and before v is popped, i.e., `pre(v)`$<$`pre(u)`$<$`post(u)`$<$`post(v)`). □

This property proves that our topological sorting method correct. Because take any edge (u, v) of the dag; since this is a dag, it is not a back edge; hence `post[u]` $>$ `post[v]`. Therefore, our method will list u before v, as it should. We conclude that, using depth-first search we can determine in linear time whether a directed graph is acyclic, and, if it is, to topologically sort its vertices, *also in linear time.*

Dags are an important subclass of directed graphs, useful for modeling hierarchies and causality. Dags are more general than rooted trees and more specialized than directed graphs. It is easy to see that every dag has a *sink* (a vertex with no outgoing edges). Here is why: Suppose that a dag has no sink; pick any vertex v_1 in this dag. Since it is not a sink, there is an outgoing edge, say (v_1, v_2). Consider now v_2, it has an outgoing edge (v_2, v_3). And so on. Since the vertices of the dag are finite, this cannot go on forever, vertices must somehow repeat—and we have discovered a cycle! Symmetrically, every dag also has a *source*, a vertex with no incoming edges. (But the existence of a source and a sink does not of course guarantee the graph is a dag!)

The existence of a source suggests another algorithm for outputting the vertices of a dag in topological order:

- `Find a source, output it, and delete it from the graph. Repeat until the graph is empty.`

Can you see why this correctly topologically sorts any dag? What will happen if we run it on a graph that has cycles? How would you implement it in linear time?