0 Introduction

So far, we've only worked with lists of data. For the next few notes, we'll consider algorithms on graphs, which are useful for encapsulating relationships in the data. Before we go into the algorithms, we'll first review graphs broadly in mathematics and computing. Then, we'll discuss graph search using depth first search (DFS). We will define pre/post order numbers, and edge types based on the tree formed by a DFS to use as a useful framework to help us understand how DFS allows us to decompose the structural relationships in graphs, namely for cycle detection, DAG linearization, and finding (strongly) connected components.

1 Graph representation

Recall the structure of a graph from a discrete mathematics course. A graph G consists of a set of vertices (often called 'nodes'), and a set of edges between pairs of vertices, denoted V and E, respectively. The edges of a graph may be either directed or undirected. An undirected edge is denoted $e = \{u, v\}$, while an edge in a directed graph is denoted $e = (u, v)$, where u and v are the two incident vertices.

Also, a tree is a minimally connected graph, where the removal of any edge disconnects the graph. Equivalently, it can be defined as a connected graph with no cycles. As such, $|E| = |V| - 1$ for any tree.

Recall the graph abstract data type from a data structures course. Graphs should support the operations of:

1. Retrieving the set of neighbors of a given vertex
2. Testing if two vertices are adjacent to each other
3. Addition and removal of vertices and edges

The two most common data structures to efficiently support this are the adjacency matrix and adjacency list.

Below, we'll describe the two data structures for directed graphs, as undirected edges can be expressed as two directed edges (i.e $\{v_i, v_j\}$ is equivalent to (v_i, v_j) and (v_j, v_i)).

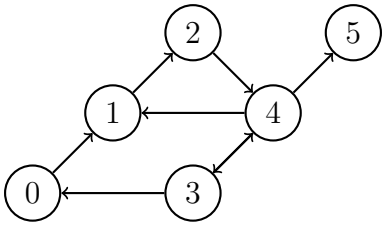
1.1 Adjacency Matrix

In an adjacency matrix, we represent a graph as a $|V| \times |V|$ matrix. The entry at index (i, j) of the matrix is an indicator (1 if True, 0 if False) indicating if there exists a directed edge from v_i to v_j. For simplicity, we'll assume our adjacency matrix is implemented as a list of lists[1].

[1] list as in Python list, not the list abstract data type.

A cute observation is that this implies that for an undirected graph, its corresponding adjacency matrix is symmetric.

Example:



The adjacency matrix for the previous graph is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

1.2 Adjacency List

In an adjacency list, we represent a graph as a list of length $|V|$. The entry at index i in the list is a collection containing all the vertices that v_i has an outgoing edge pointing to it. For simplicity, we'll assume that our adjacency list is implemented as a list of lists.[2]

Similarly, this implies for an undirected graph, if v_j exists in v_i's list, then v_i exists in v_j's list.

Example: The adjacency list for the previous graph is:

Vertex	Neighbors
0	1
1	2
2	4
3	0, 4
4	1, 3, 5
5	

[2]Why not use a list of hashsets or binary trees, as that would allow us to find an edge in $O(1)$ or $O(\log(\text{outdeg}(v_i)))$ time, respectively? In addition to overhead costs unexpressed in asymptotic notation, ultimately, they are unnecessary for most algorithms, which usually just involve iterating over the edges.

1.3 Tradeoffs

For both of our data structures, we've assumed they are implemented as a list of lists. But they act differently. The inner lists of the adjacency matrix will always be length $|V|$, and we can identify a desired vertex by its corresponding index. On the other hand, the inner lists of the adjacency list only include a vertex if the corresponding edge exists in the graph, rendering its indices meaningless, and each has a length of only outdeg(v_i) entries.

Below is a table demonstrating the tradeoffs:

Implementation	Space	Find Neighbors	Test Adjacency
Adj Matrix	$O(\|V\|^2)$	$O(\|V\|)$	$O(1)$
Adj List	$O(\|V\| + \|E\|)$	$O(\deg(v_i))$	$O(\deg(v_i))$

Here, we have not considered putting weights on edges, though we can easily modify our implementations to accommodate it. In this note, edge weights are irrelevant, though it will be necessary in the next note.

Generally, we prefer to use adjacency lists, as most graphs in practice are very large and sparse, meaning that there aren't many edges, adjacency lists are much more memory efficient and have a faster find neighbors operation. Thus, for the rest of this note, we will assume our graph is given as an adjacency list for any runtime calculations.

2 DFS

2.1 Inspiration

Suppose you are exploring a maze. Without any tools, you may end up walking in cycles and never explore the entire maze. The age-old method of systematically exploring a maze is to use a ball of string and a piece of chalk. The process can be summarized as going as far as you can until you see a dead-end while unraveling the ball of string and marking every junction in your path. At the dead-end, backtrack using your string until you reach a point where there exists an unmarked junction. Then repeat if there are still unexplored junctions. This guarantees that you reach every junction reachable from your start point.

Translating this into computer science, a junction is a vertex and a corridor between two junctions is an edge. The chalk is a length $|V|$ list of booleans, denoted 'visited', where each ith entry corresponds to whether or not we've visited v_i yet in our algorithm. The ball of string is a stack, where each entry is a vertex to begin or continue visiting. Though for clarity, we'll create the stack implicitly using recursion. Let's call this algorithm Explore(v).

2.2 Explore/DFS Algorithm

To summarize, given a vertex v, EXPLORE(v) marks the vertex as visited, then recursively calls EXPLORE on each of its neighbors. By the nature of how recursive algorithms execute on a computer, this gives us the "depth-first" behavior. This holds for both undirected and directed graphs. Below is the pseudocode. We'll explain PREVISIT(v) and POSTVISIT(v) in the next section. [3]

Algorithm 1 Explore Algorithm

```
1: function EXPLORE(G, v)
2:     visited[v] ← True
3:     PREVISIT(v)
4:     for (v, w) ∈ E do
5:         if not visited[u] then
6:             EXPLORE(G,w)
7:     POSTVISIT(v)
```

EXPLORE finishes once it visits all the vertices reachable from the start vertex. This isn't necessarily the entire graph, as there can exist many vertices unreachable from the vertex. DFS solves this issue by just repeatedly restarting EXPLORE at a new, unvisited part of the graph.

Explicitly, DFS simply runs EXPLORE as a subroutine for all unvisited vertices. Below is the pseudocode for DFS. EXPLORE is also re-written for reference.

[3] Implementation details vary. In our implementation, for each neighbor, we only explore it if it hasn't been visited yet. In other implementations, we instead explore all its neighbors, and include a base case at the top of the function to immediately return if the node has already been visited. Other implementations may also opt to save space by bookkeeping if a node has been visited through modifying the graph in some way instead of explicitly tracking a visited list or set of vertices.

Algorithm 2 DFS

```
1: function DFS(G)
2:     visited ← [False] * |V|
3:     for v ∈ V do
4:         if not visited[v] then
5:             EXPLORE(G, v)
6: function EXPLORE(G, v)
7:     visited[v] ← True
8:     PREVISIT(v)
9:     for (v, u) ∈ E do
10:        if not visited[u] then
11:            EXPLORE(G,u)
12:    POSTVISIT(v)
```

Note that Explore and DFS are essentially the same algorithm. We make a distinction between them to stay consistent with DPV, though many other sources will use DFS and explore synonymously.

Runtime: First, constructing the visited list takes $O(|V|)$ time. Second, for each vertex, Explore looks through $O(\deg(v_i))$ edges. Since the sum of the degrees of a graph is $|E|$ (or, $2|E|$ for undirected graphs), this step yields $O(|V| + |E|)$. Thus in total, we have $O(|V| + |E|)$.

In other words, we are using an amortized analysis for the second part of our analysis - throughout the entire algorithm in total, each edge will only be considered twice for undirected graphs or once for directed graphs, yielding $O(|V| + |E|)$.

This runtime, $O(|V| + |E|)$, is often called linear time, where linear refers to "linear in terms of the vertices and edges".

2.3 Preorder/Postorder Numbers

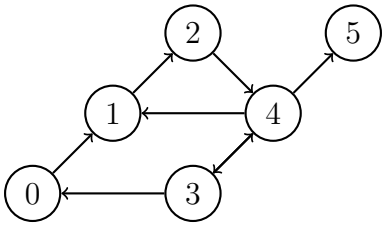
Now, let's return to previsit and postvisit. These are optional procedures performed at the first time you arrive at the vertex and the last time you leave the vertex, respectively. The naming is rather confusing as they are called during a visit (not before or after as the name suggests) - perhaps it'd be clearer as pre-(explore neighbors) and post-(explore neighbors).

In any case, as an example, you could define previsit(v) to just print v. Here, running DFS will just print the vertices in the order you first reach each vertex.

To keep track of the order our DFS traverses each vertex, we'll keep track of a global 'clock' that we'll use to mark for each vertex when the DFS first arrives at it and last

leaves from it. These are called the preorder and postorder numbers. So, the interval $[\text{pre}[v], \text{post}[v]]$ is exactly the range of time that v was on the stack.

Example: The preorder and postorder numbers of DFS starting at vertex 0, assuming we break ties in favor of vertices with a smaller label:



Vertex	**preorder**$[v]$	**postorder**$[v]$
0	0	11
1	1	10
2	2	9
3	4	5
4	3	8
5	6	7

preorder$[v]$ - The first time we arrive at v.

postorder$[v]$ - The last time we leave from v.

Explicitly, we could implement this by defining in the main body of the DFS function a variable c (for 'clock') and two length V lists called pre and post that will contain the preorder and postorder number of v_i at the ith index for all i. In the body of the previsit or postvisit procedures, we increment c, then assign relevant index of the pre or post lists to c.

Note that these numbers are specific to a particular DFS traversal on a given start vertex. It doesn't make sense to refer to the preorder/postorder numbers of a graph, as there is a numbering for each start vertex and each of its possible tiebreaking schemes.

So, the pre/post order numbers tells us when each vertex in visited in the context of a DFS traversal. This information will be useful for understanding the intuition the algorithms later in this note.

2.4 DFS Trees

By nature of DFS never revisiting any vertices, the path that the DFS actually traverses through is acyclic and thus defines a tree, called a DFS tree. The start vertex of the DFS is known as the "root" of the tree.

Note that every time we finish exploring what is reachable, when we restart the DFS at a new section of the graph, this starts a new tree. So, in total we have a DFS forest.

We can classify all the edges of a graph with respect to a DFS tree.

For undirected graphs:

- *Tree edges* are the edges that we actually traverse in the DFS (i.e the edges in the DFS tree).
- *Non-tree edges* are all the other edges in the graph.

For directed graphs, we can further break down non-tree edges based on ancestry in the DFS tree. Informally, an ancestor of a vertex is defined as a parent, grandparent, great-grandparent, etc. A descendant of a vertex is defined as a child, grandchild, great-grandchild, etc. The root is the ancestor of every vertex:

- *Tree edges* are the edges that we actually traverse in the DFS (i.e the edges in the DFS tree).
- *Forward edges* are edges that go from a vertex to one of its descendants in the DFS tree to a descendant.
- *Back edges* are edges that go from a vertex to one of its ancestors in the DFS tree.
- *Cross edges* are edges that go from a vertex to neither ancestor nor descendant.

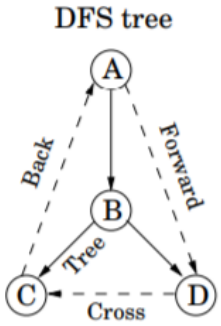


Figure 1: Types of DFS edges. Source: DPV, pg. 95

Note that both tree edges and forward edges go from an ancestor to its descendant. The difference is that a tree edge goes directly from a parent to a child.

Regarding cross edges, they could define any indirect familial relationship (e.g sibling, cousin, aunt/uncle, nephew/niece, etc.), or they could be from a completely different family (i.e another DFS tree in the DFS forest). For example, imagine adding a new vertex, E to the example above. Any edge from E to any other vertex would be a cross edge. Try experimenting with adding other vertices/edges to see the other familial relationships!

How can we compute these edge classifications directly in the DFS? For undirected graphs, during our DFS, any edge to an unvisited vertex is a tree edge, and the rest (i.e edges to a visited vertex) are non-tree edges.

For directed graphs, given two vertices u and v, what does it mean if u is an ancestor of v? u is an ancestor of v iff u is discovered first and v is discovered during Explore(u) (either immediately or after a some recursive calls). Remember, we have the relative times of when vertices are being explored (i.e on the stack) from the pre/post numbers! Here, $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$. So, this identifies tree/forward edges. We can also use this to identifies back edges by symmetry - u is a descendant of v if v is an ancestor of u, so we can just swap u and v above. Finally, as cross edges mean there's no direct lineage, u and v should not be on the stack at the same time - v should have already finished.

Here is a precise summary in interval notation. '[' denotes pre and ']' denotes post:

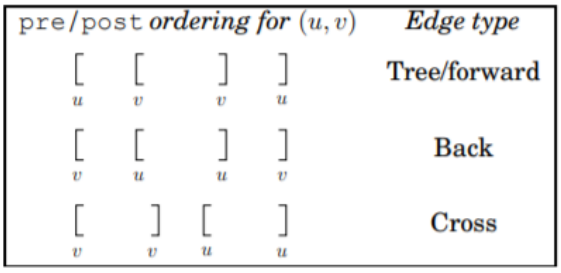


Figure 2: Types of DFS pre/postorder intervals. Source: DPV, pg. 95

Note that this is comprehensive, that is, no other orderings can possibly exist (i.e if u is placed on the stack before v, it must finish after v due to LIFO of a stack. The other case would be u completely finishing before v, but that contradicts how DFS works, as it should visit v before completing).

3 Cycle Detection

Now, with the framework given by the pre/post-order numbers and DFS tree edges, we'll move onto the applications of DFS. Cycle detection follows immediately from the previous discussion on DFS edge classification. Recall that a cycle occurs when there are two distinct paths between two vertices.

For undirected graphs, the moment we see a non-tree edge, our graph is therefore not a tree. Explicitly, when we loop over the neighbors in Explore, if any neighbor is already visited, we can immediately return that there is a cycle.

For directed graphs, it is not as simple, as a non-tree edge can be either a forward, back, or cross edge (i.e just because we see an already visited neighbor, unlike undirected

graphs, it doesn't mean that neighbor can reach us!). We are only interested in back edges.

There exists a cycle iff there exists a back edge. This should make sense, since a back edge implies there is a path from an ancestor to a descendant, but also another path (by following the back edge) from a descendant to an ancestor. And given a cycle, running DFS from an arbitrary root node in the cycle yields the last edge a back edge, since it points to the root. So, we can just run DFS, keeping track of pre/post numbers, then scan the edges to see if there exists a pre/post ordering corresponding with a back edge.

There are other methods to detect cycles as well - check out the exercises.

4 Topological Sort/Linearization of a DAG

When a directed graph has no cycles at all, it is called a directed acyclic graph (DAG). DAGs are really interesting due to their hierarchical substructure - for example, treating each vertex as a task and each edge as a dependency, a DAG means there is an order you can do all your tasks (whereas if there is a cycle, there is no solution)[4]

Any vertex that doesn't have any incoming edges is called a source vertex, and any vertex that doesn't have any outgoing edges is a sink vertex. There can be multiple sources or sinks, and every DAG must have at least one source and at least one sink (it must start and end somewhere!). If you remove a sink, then informally, "the second to last" vertex becomes the new sink, and similarly for a source. If the graph consists of just a single vertex, it is both a source and a sink.

Given a DAG, we want to topologically sort it (also called linearization), which means to return an ordering of the vertices such that for every edge (u, v) in the graph, u comes before v in the ordering (as such, the first vertex in the topological ordering is a source, and the last vertex in the ordering is a sink). Thus, all dependency constraints are satisfied.

Example:

Note that the topological ordering is not unique (e.g if you have two sources, either one can come first).

To topological sort a DAG, we can simply just run a DFS starting at any vertex. Imagine running DFS on a DAG. You'll quickly see that a vertex cannot finish until its descendants finish. In other words, for all edges (u, v), post$[u] >$ post$[v]$. We can see this as given two vertices u and v with an edge (u, v), try running DFS from v first and running DFS from u first. In the former case, Explore(v) will finish before we

[4]In fact, the substructure given by a DAG forms the basis of all dynamic programming algorithms, which we'll see in a future note.

even start exploring u. And in the latter case, u calls EXPLORE(v), which will then finish before backtracking to u.

Alternatively, we can just see this is true from figure 2 in the DFS tree section, as in the two cases besides back edges (which can't exist in a DAG as they are acyclic), post[u] > post[v].

Note the highest postorder number is taken by a source vertex, and the lowest postorder number is taken by a sink. We can just return the vertices in order of decreasing postorder number.

5 Connectivity

5.1 Undirected Graphs

Recall two vertices u and v are connected to each other if there is a path between them (i.e u can reach v and v can reach u). In an undirected graph, if u can reach v, this implies v can reach u, by following the same path backwards.

An undirected graph can be decomposed into its connected components, that is, maximal disjoint subsets of the graph where every pair of vertices in a component can reach each other.

DFS directly yields us our connected components, as EXPLORE(v) visits everything reachable from v. Every time we get stuck, we've concluded a connected component. When we restart, we create a new connected component.

Explicitly, we can just define a list denoted 'cc' and an index 'i' in the main body of the DFS. Every time we call EXPLORE on a new vertex in the outer loop, we increment the index. In the EXPLORE(v) function, we can assign append v to the current index of cc.

Note that each DFS tree is a *spanning tree* of a connected component.[5]

5.2 Directed Graphs - Kosaraju's Algorithm)

In directed graphs, connectivity is defined the same way (i.e u can reach v and v can reach u), but is now called strong connectivity. A directed graph is thus built up of strongly connected components (SCCs).

However, unlike undirected graphs, in a directed graph, if u can reach v, it does not necessarily mean v can reach u, which makes the strongly connected components trickier to compute. A naive algorithm would be to run DFS from all vertices, keeping

[5]Spanning trees turn out to be very important in computer science. We'll cover spanning trees in detail in a future note.

track of the set of vertices reachable from each vertex, then merging them into groups to retrieve strongly connected components. This would be $O(|V| * (|V| + |E|))$, which isn't great.

One important property of SCCs is that if we treat each SCC as a 'meta-node', and an edge between SCCs as a 'meta-edge', our resulting 'meta-graph' forms a DAG. This should make sense, as suppose it wasn't a DAG (i.e has a cycle). In that case, the cycle of meta-nodes should merge into a single meta-node, as the nodes in the cycle are strongly connected to each other.

As the metagraph is a DAG, it now has at least one source SCCs and at least one sink SCCs.

Example: SCC's

Below, we'll derive an intuitive algorithm that can be done in linear time, using DFS twice. Though feel free to skip to the algorithm.

5.2.1 Derivation

Try just running DFS on a graph at various start points. Ideally, DFS would traverse in a way such that the vertices belong to the same SCC, so we can isolate and extract it. However, DFS may very likely travel across to another SCC without fully exploring an SCC. How can we guarantee that we visit only nodes within a SCC?

Observation 1: Exploring sink SCC Suppose we already had our SCC metagraph (we don't actually yet, that's the goal of the algorithm. But it's useful for intuition). You may have noticed that if we start DFS from a vertex belonging to a sink SCC, then it can't accidentally leave the SCC without fully exploring the component, as there aren't any outgoing edges. So, for a vertex v in a sink SCC, Explore(v) yields precisely the sink connected component. Then we can effectively "remove" this sink SCC, yielding a new sink SCC, and we can iterate from there.

However, we have no way of finding a vertex in a sink SCC. Unlike a normal DAG, iterating the adjacency list won't suffice, as the degrees of the vertices don't tell us anything here.

Observation 2: Generalizing a DAG property Recall this property in a DAG: for all edges (u, v), post$[u] >$ post$[v]$. We can generalize it with SCCs: Say we have two SCCs C and D. For all edges between two SCCs, the highest postorder number in C is greater than the highest postorder in D. We can justify this similarly - either DFS starts from a vertex in D or a vertex in C. In the former case, everything in D will finish before anything in C, thus every postorder number in C will be higher than anything in D. In the latter case, C cannot be finished exploring without travelling to D and

then backtracking to C (otherwise it would contradict reachability in Explore). So, the highest postorder in C is greater than all the postorders in D.

Thus, the highest postorder number overall, by transitivity, must be in a source SCC.

However, note that the lowest postorder number is not necessarily in a sink vertex, as if we start DFS from a vertex that connects to a lower SCC, but it ends up travelling other vertices in the source first, the other vertices will get stuck and finish before ever travelling to the lower SCC (Exercise). So that doesn't work.

But we're looking for a sink SCC... luckily sources and sinks are quite similar - they are special being the first and last members of a DAG.

Observation 3: Reverse Graph If we reverse all the edges, then sinks become sources and sources become sinks. So, we use the reverse graph as a proxy - we can find a source in G^R by selecting the vertex with the highest postorder number, which gives us a sink in G that we desired for observation 1: running Explore in a sink yields us exactly the sink connected component.

Furthermore, suppose we finish exploring this component. Recall from DAGs that removing a source transforms the second place vertex into a source. Referring back to our postorders again, the unvisited vertex with next highest postorder is the new source, as the relative ordering of the postorders remains fixed from observation 2. And it corresponds to a new sink in G. We can repeat this process to retrieve all the SCCs in reverse topological order of G.

5.2.2 Kosaraju's Algorithm

Thus, we have our algorithm:

1. Create a copy of G with the edges reversed, called G^R.
2. Run DFS on G^R, storing the postorder numbers
3. Run DFS on G, tiebreaking in decreasing order of the postorder numbers from G^R.

For the last step, just like in the undirected graph algorithm, we can just store the SCCs in a list '*scc*'. Every time we jump to a new unvisited vertex in the outer loop, that is a new SCC.

Notably, not only does this give us the strongly connected components, but it gets us them in reverse topological order! This should make sense, as we essentially generalized the topological sorting algorithm, with a few additional modifications.

6 Exercises

1. Write the preorder and postorder numbers when running DFS(A) on the following graph, tie-breaking alphabetically.
2. How many times do we 'visit' u if u is of degree d in an undirected graph in a DFS traversal, using our pseudocode above? Let's define 'visit' intuitively how you'd expect; explicitly, we 'visit' u every independent instance of executing some amount of code within the explore(u) function.
3. Given a directed graph G and an edge (u, v) in G, find an efficient algorithm to determine if a cycle exists with (u, v) in it.
4. Give an efficient algorithm to determine whether there exists a vertex s in a directed graph G such that every vertex in G is reachable from s.
5. https://leetcode.com/problems/number-of-islands/
6. How else can you find a backedge
7. https://leetcode.com/problems/course-schedule/

7 Solutions

1. XX
2. d times for $d \geq 1$, or 1 for $d = 0$. For $d \geq 1$, we visit u once when we first arrive at the u, and for the $d-1$ other neighbors, we visit u again after finishing exploring the neighbor. For $d = 0$, the outer code in DFS will allow us to visit u once in its entirety.
3. Let the edge be (u, v). There exists a cycle with the edge in it if and only if u is reachable from v. This is achievable using DFS, and our algorithm will take linear time with respect to the number of edges and vertices.
4. We claim such a vertex s exists if and only if there is exactly one source strongly connected component. If there exists one source component, then pick any vertex in that component to be s; clearly s will be able to reach all other vertices in the entire graph. If there is more than one source component, then no matter how s is picked at least one of those components will be entirely unreachable.

   Thus, our algorithm consists of using the SCC-DAG algorithm to find the strongly connected components of G, then verifying that only one source exists. This takes linear time with respect to the vertices and edges.

Feedback form: https://forms.gle/cM1io6eXyH65ytMP8

If time permits, in a future date, we'll add extra sections on a stack implementation of DFS, alternate algorithms for DAG topological sort (Kahn's algorithm) and finding SCCs (Tarjan's algorithm).

Contributors:

- Kevin Zhu
- Axel Li