1 Graphs

Formal definition of undirected graphs:

- Called $G(V, E)$.
- has a set V of vertices, and E of edges.
- Edges are denoted as a pair of vertices $\{v_i, v_j\}$, and undirected mean that each pair of $\{v_i, v_j\}$ is unique.
  - e.g. Facebook friendship graphs, where the friendship must be mutual. (Has nearly 3 billion nodes, and each node has 330 vertices)
- Directed graph is denoted in the same way, except that $\{v_i, v_j\}$ means specifically that $v_i \longrightarrow v_j$.

1.1 Size of Graphs

- Generally denoted as the number of vertices (denoted by n) and the number of edges (denoted by m).
- m can be at most n^2, in the case that every vertex is connected to every other vertex (in which case we have $n(n-1)$ edges)
- Degree: the number of edges that a vertex has. With directed graphs, we can specify in-degrees and out-degrees.

1.2 Representing Graphs

- Represented either as an adjacency matrix or an adjacency list.
  - Adjacency matrix: have the vertices listed out on both row and column, put a 1 if $\{v_i, v_j\}$ are connected on our graph.
  - Adjacency list: List of length n, and each cell is a linked list that points to all the neighbours of v_i.
- There are tradeoffs for both ways:

	Adjacency Matrix	**Adjacency List**
Storage Size	$O(n^2)$	$O(n+m)$
Checking whether $(u, v) \in E$	$O(1)$	$O(\deg(u))$
Enumerate all of u's neighbours	$O(n)$	$O(\deg(u))$

- We will work with adjacency lists, since the storage size for adjacency matrices get too large too quickly.

1.3 Graph Connectedness

- We first have to solve the problem of graph traversal. Our approach will use "string and chalk," so we mark when we've reached a dead end and the "string" will allow us to backtrack.
- Algorithm description:

```
def explore():
        visited[u] = true

        For all edges of $u$: if visited[v] = false then run explore(v)
```

- Explore guarantees that all the vertices that are visited by explore have a path from u to that vertex, and vice versa.
  - We prove the other direction: if there's a path, then that node is visited.

    Assume that this is false: assume $\exists v$ that hasn't been explored but there is a path from u to v. Instead of looking at u to v, we look at the path from u to v_k, the first node along the path from u to v that is unexplored. This means that the algorithm reached v_{k-1}, then failed to explore v.

    This is a contradiction, since explore must have been called on v_{k-1}, but failed to recurse down v_k.

1.4 Depth First Search (DFS)

Essentially calling explore, except it does it recursively on all the remaining nodes that haven't been visited yet.

- We can also use DFS to find the connected components of a graph! When we explore with DFS, we are implicitly exploring connected components, this uses the transitive fact of the `explore()` function.
- The edges that are visited by DFS are categorized into tree edges and back edges. Tree edges are edges that are used when the algorithm runs, and back edges are ones that exist within the original graph but aren't used.
- There's a third class called a *cross edges*, but they cannot exist for an undirected graph.
  - The proof is by contradiction: imagine that it did exist, then DFS would have called explore on that ancestor; but it can't possibly have since u and v do not exist in the same branch.
  - They *can* exist in a directed graph, since we only traverse through out edges (so there can be an in-edge that never gets traversed).

1.4.1 DFS Runtime

`Explore()` is called only once per node, and the runtime for each node for explore is proportional to $\deg(u)$, so the total is:

$$\sum_{u \in V} O(1 + \deg(u)) = O(n + m)$$

1.5 DFS for Directed Graphs

It's the same principle, our explore still only runs recursively on unvisited neighbours, but we need to also keep track of the amount of time at which we start and finish processing that node.

- Every time we enter a node, we stamp it with the time of the start clock. When we come out, we stamp again with the end clock. Every time we progress the clock, we increment it by 1.
- The point of the clock will be revealed later on in future lectures.
- The edges we keep track of here are forward, back and cross edges. Forward means we go down the tree from ancestor to descendant (not its immediate child), back means we go backwards (can be immediate) and cross edges are defined exactly as before.

The edges (pre, post, cross) are useful in determining properties of the graph G.

- Cross edges can only go from a later point in one branch to earlier than the other branch and not the other way around, due to the way that DFS executes depth-first.
- Suppose $(u, v) \in E$ is a tree edge. Then, we know that $\text{pre}(u) < \text{pre}(v)$ (since u is hit first), and $\text{post}(u) < \text{post}(v)$.
- Suppose that $(u, v) \in E$ is a back edge. Then, we have $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$.

2 Strongly Connected Graphs

Recall that we saw earlier that DFS basically just calls explore repeatedly, to find all the vertices reachable from a vertex u. We also introduced two clocks, where when we first visit a node we stamp it with the start clock, and that vertex will have a pre(u) quantity that stores its value, then increments clock. When we leave the vertex, we stamp again with a post(u).

- For cross edges (the only thing we didn't finish last time), we have pre(v) < post(v) < pre(u) < post(u) (we get to v and finish exploring v before we ever touch u).
- As a recap:

Edge type	Relation
Tree edge	pre(u) < pre(v) < post(v) < post(u)
Back edge	pre(v) < pre(u) < post(u) < post(v)
Cross edge	pre(v) < post(v) < pre(u) < post(u)

2.1 Topological Sort

- The process of finding an ordering of vertices so that no edges go backwards. This is used in software package loading, to make sure that things aren't being and that are being downloaded in the chronological order.
- Mathematically, if u comes before v in the ordering, then there is no edge (v, u).
- Two types of special nodes: **sources** and **sinks**
  - **Source:** A node that has no incoming edges and has only outgoing edges.
  - **Sink:** A node that has no outgoing edges and only has incoming edges
- Note that a node with no edge at all is considered both a source and a sink.

2.2 DAGs

- Called a directed acyclic graph, or basically just a graph without any directed cycles. If we have a cycle, then we can't toplogically sort.
- We will find that if we run DFS on a graph, it is a DAG iff it has no back edges. We prove that if we have a back edge, then it cannot be a DAG: since they go from something that's already been visited to something that's been visited earlier, then we have already found a cycle.

  Now we prove that if we don't have a DAG (i,e, has a cycle) , then it has a back edge. Since DFS visits every vertex in a graph, it will eventually enter our cycle at some v_i. Then, it will traverse through the cycle until it visits all the nodes and eventually gets to v_k, the node right before it comes back to v_i. Then, $v_k \rightarrow v_i$ will become a back edge, since it was visited earlier in the graph.
- Back edges are really special! Recall that we only have a back edge when post(u) < post(v), since the other edges have post(v) < post(u). We then conclude that if we have a DAG, then it has the property that post(v) < post(u) (since it can't have back edges).

  Does the logic work the other way around? Does this mean that if post(u) < post(v) then we have a DAG?

  Not necessarily. Just because post(v) > post(u) doesn't necessarily imply that a back edge exists.
- Our algorithm for topological sort: do a DFS on graph G, then enumerate all $v \in V$ in the decreasing order of post(v).

2.3 Strongly Connected Components

- To find DAGs on undirected graphs we can run DFS on them, but what about directed graphs?
- **Definition:** Vertices u and v are strongly connected if there is a path from u to v and there is also a path from v to u.
- A graph is *strongly connected* when all of its vertices are strongly connected.
- Generally, we partition a graph into *strongly connected components*, since it's very rare that the entire graph is strongly connected.
- Strong connectivity is an example of an equivalence relationship, since it satisfies all the properties: reflexive, symmetric and transitive.
  - Every vertex is strongly connected to itself
  - If A is strongly connected to B, then B is strongly connected to A.
  - If A is strongly connected to B and B is strongly connected to C, then A is strongly connected to C.
- If we flip all the edge (i.e. reverse the direction), the strongly connected components don't change at all!
- We can then divide this into a *meta graph*, where we group the graph by strongly connected components (try to be as general as possible when you do this).
  - The meta graph can't have cycles! Had a cycle existed, then it would imply that vertices from one strongly connected component can reach vertices of another, and vice versa!
  - Therefore, the meta graph *must* be a DAG.
- Why care about SCC? It is useful in many different fields, since SCCs naturally imply a strong equivalence of objects in a graph.

2.4 Finding SCCs

- **Attempt 1:** Consider all possible decompositions and check (BAD!)
- **Attempt 2:** Consider pairs of nodes, then run explore to see if they reach the other. (ALSO BAD!)
- There exists an algorithm that runs this in $O(n + m)$ time, where we have to run DFS (smartly) only twice!

2.4.1 More Properties of SCCs

- It actually matters where we start our DFS, since sometimes we can't exit the particular connected component. Specifically, if the node is part of a source in the meta graph, then it will visit many nodes, whereas if the node is a sink then we never exit that particular connected component.
- The "right" place to start the DFS is in the **sink of the meta graph!** The key thing is that we shouldn't exit that connected component, so running `explore()` on any vertex within this CC will give us the whole SCC.
- **Idea:** Do the topological sort on the meta graph, then run DFS on the sinks of the meta graph.
- Suppose we run DFS on a graph G. Let C and C' be two connected components such that $C \longrightarrow C'$ in the meta graph. Then, $\max(\text{post}(v \in C)) > \max(\text{post}(u \in C'))$.

  Proof: Split into cases, based on where in the graph we started:

Case 1: Suppose DFS visited C first ($u \in C$ is the first node visited by DFS. Then, DFS will explore C' first before finishing its exploration of C. This means that DFS finishes C' before finishing C, meaning that max(post($v \in C$)) > max(post($u \in C'$)).

Case 2: Suppose DFS visited C' first. Then, we will never visit C since there is no directed edge between $C' \longrightarrow C$. Therefore, the post of C' stops before DFS even reaches C, Hence, we have max(post($u \in C'$)) < max(post($v \in C$)).

- Immediate corollary of this: Suppose we ran DFS on a graph G. The highest post(v) belongs to a node v that is in the source SCC of the meta graph!
  - Imagine max(post(C)) is not the largest post(v) value. Then, this means that there exists another C'' whose post(C'') is larger than that of C!
- So now we have a way of finding the source of the vertices in C, but we want to start from sinks, so we **flip the edges, then run DFS!** The only difference betweeh G and G^R (the reversed graph) is the direction of the edges; the connected components remain the same.

  Instead of flipping edges, why not run the algorithm from the minimum post value instead?

  Because the minimum isn't guaranteed to be a SCC in the same way the max is.

2.5 The Algorithm

- Compute G^R.
- Run DFS on G^R.
- Store the post numbers of this DFS in array called post-r
- Run DFS on G, but explore unvisited nodes in *decreasing* order of post-r. For every DFS we run, we find a new SCC.

Compute the reverse of G, call it G^R.