1 Graph Algorithm Runtimes

Recall the following algorithms:

```
explore(v)
    v.visited = true
    For each edge (v, w)
        If not w.visited
            explore(w)

DepthFirstSearch(G)
    Mark all v in G as unvisited
    For v in G
        If not v.visited
            explore(v)
```

Although $O(|V|+|E|)$ is linear time, in reality, graph algorithm runtimes depend on both $|V|$ and $|E|$. What algorithm is better may depend on the relative sizes of these parameters.

	Sparse Graphs	**Dense Graphs**
Runtime	$\|E\|$ small ($\approx V$)	$\|E\|$ large ($\approx V^2$)
Examples	Internet, Road Maps	Flight Maps, Wireless Networks

2 Graph Representation

How do you store a graph in a computer?

- **Adjacency Matrix:** Store list of vertices and an array `A[i, j] = 1` if edge between v_i and v_j.
  - Slow space for dense graphs.
  - Slow for most operations.
- **Edge List:** List of all vertices with list of all edges.
  - More space-efficient for a sparse graph.
  - Hard to determine edges outof the single vertex.
- **Adjacency List:** For each vertex, store a list of neighbors.
  - Needed for DFS to be efficient.
  - We will usually assume this representation.

3 Connected Components

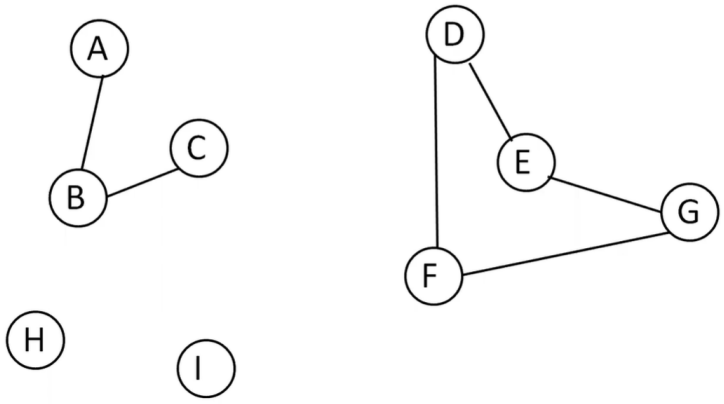
We want to understand which vertices are reachable from which others in a graph. We've already kind of done this using `explore(v)` to find which vertices are reachable from a given vertex.

For a more theoretical answer, consider the theorem:

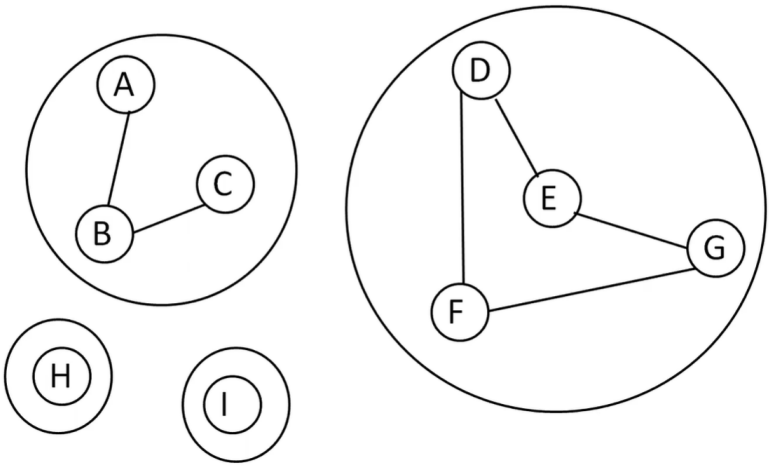
Theorem 3.1

The vertices of a graph G can be partitioned into connected components so that v is is reachable from w if and only if they are in the same connected component.

For instance, consider the following graph:



We can split this graph into four components:



Here, if there are two points in a component, then we can reach the first point from the second and vice versa. However, if there are two points in two different components, then this is not possible.

3.1 Computing Connected Components

Given a graph G, how do we compute its connected components?

- Easy Way: For each v, run `explore(v)` to find the vertices reachable from it. Group together into components. The runtime will be $O(|V|(|V| + |E|))$.
- Better Way: Run `explore(v)` to find the components of v. Then, repeat this on the unclassified vertices.

3.2 Applying Depth-First Search (DFS)

We can use depth-first search to simulate this. In fact, we can use the same algorithm above but with a bit of additional "information" to get our answer:

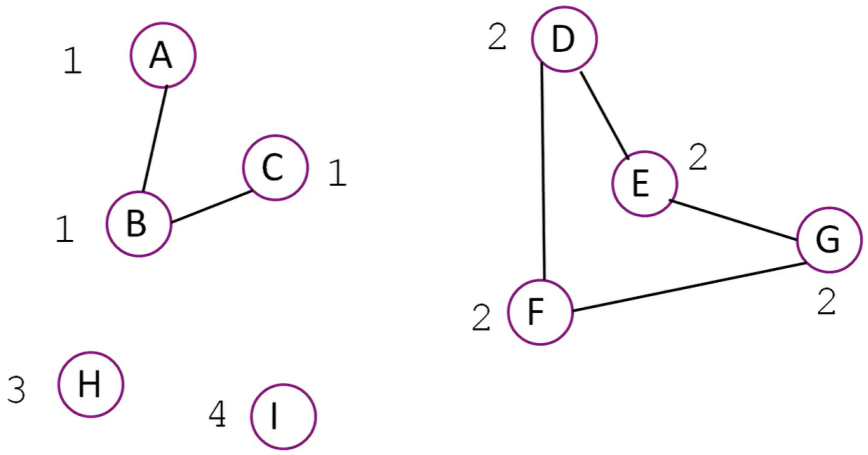
```
explore(v, CCNum):
    v.visited = true
    // CC is connected components
    v.CC = CCNum
    for each edge (v, w):
```

```
            if not w.visited:
                explore(w)

    ConnectedComponents(G):
        CCNum = 0
        for each v in G:
            v.visited = false
        for each v in G:
            if not v.visited:
                CCNum++
                explore(v, CCNum)
```

The runtime is $O(|V| + |E|)$.

If we recall the graph above, running this algorithm would yield something like:



Where `CCNum` is `4`, or 4 connected components.

3.3 DFS: A Discussion

What does DFS actually do?

- No output.
- Marks all vertices as visited.
- Easier ways to do it.

However, DFS is a useful way to *explore the graph*. In particular, by augmenting the algorithm a bit, like we did with the connected components algorithm, we can learn useful things. In other words, DFS by itself is quite useless, but adding a bit of additional information will make it more useful.

3.4 Pre- and Post- Order

How can we augment this algorithm?

- Keep track of what algorithm does and in what order.
- Have a "clock" and note time whenever:
  - Algorithm visits a new vertex for the first time.
  - Algorithm finishes processing a vertex.
- Finally, we can record values as `v.pre` and `v.post`.

3.5 Computing Pre- and Post- Order

Consider the algorithm, which is the same thing as above but with some changes to make it useful:

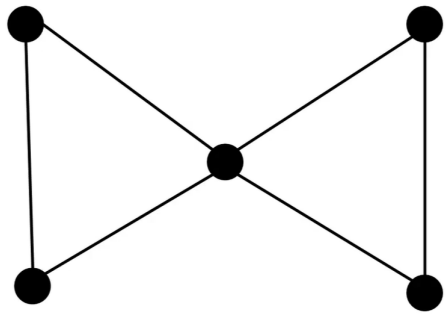
```
explore(v)
    v.visited = true
    v.pre = clock
    clock++
    For each edge (v, w)
        If not w.visited
            explore(w)
    v.post = clock
    clock++

DepthFirstSearch(G)
    clock = 1
    Mark all v in G as unvisited
    For v in G
        If not v.visited
            explore(v)
```

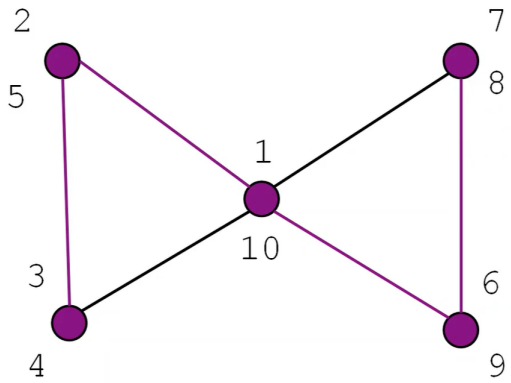
The runtime is still $O(|V| + |E|)$.

3.6 Example of Pre- and Post- Order

Consider the following graph:



After running the above algorithm, we get:



How did we get this?

- We assign `1` to the center vertex. `1` has an unexplored neighbor, which is the top-left vertex.

- We assign `2` to the top-left vertex. `2` has an unexplored neighbor, which is the bottom-left vertex.
- We assign `3` to the bottom-left vertex. `3` doesn't have any other unexplored neighbors.
- Since we can't go anywhere from `3`, we assign it a post-order value of `4`.
- Now that we're back at `2`, note that there aren't any other unexplored edges, so we give it a post-order value of `5`.
- We're now back at the center vertex. However, we still have more to explore.
- We assign `6` to the bottom-right vertex.
- We assign `7` to the top-right vertex.
- Since we can't go anywhere from `7`, we assign it a post-order value of `8`.
- Now that we're back at `6`, we assign it a post-order value of `9` since we can't explore anything else.
- Now that we're back at `1`, we assign it a post-order value of `10` since we can't explore anything else.

If the graph is connected, the first vertex will always be the last vertex. However, if the graph is disconnected, the first vertex may not necessarily be the last vertex.

3.7 Application of Orders

Proposition. *For vertices v, w, we can consider the intervals `[v.pre, v.post]` and `[w.pre, w.post]`. These intervals:*

1. *contain each other if v is an ancestor/descendant of w in the DFS tree.*
2. *are disjoint if v and w are cousins in the DFS tree.*
3. *never interleave (`v.pre < w.pre < v.post < w.post`).*

Proof. Assume that the algorithm finds v before w (`v.pre < w.pre`). If the algorithm discovers w after fully processing v, then:

- `v.post < w.pre`
- Intervals are disjoint
- v and w are cousins.

If the algorithm discovers w before fully processing v:

- The algorithm finishes processing w before it finishes v.
- `v.pre < w.pre < w.post < v.post`
- Represents nested intervals.
- v is an ancestor of w.

And so we are done. □

4 Directed Graphs

Often, an edge makes sense both ways. But, sometimes, streets are one directional.

Definition 4.1

A **directed graph** is a graph where each edge has a direction. We say that it goes from v to w.

Often, we draw arrows on the edges to denote direction.

4.1 DFS on Directed Graphs

We can use DFS on a directed graph. However, we only follow directed edges from v to w. The runtime is still $O(|V| + |E|)$, and `explore(v)` discovers all vertices reachable from v following only directed edges.