1 Strongly Connected Components

The issue with our algorithm is that we recompute the postorder for every SCC we need to find. However, we don't need to do this; rather, after removing some strongly connected components to get G', the largest postorder number of vertices in G' is still in a sink component of G'.

1.1 Better Algorithm

```
SCCs(G)
    Run DFS(G^R), record postorders
    Mark all vertices as unvisited
    For v in V in reverse postorder
        If v not in a component yet
            explore(v) on G-components found,
            marking new component
```

So, really, this is just 2 DFSs, so the runtime is $O(|V| + |E|)$.

2 Paths in Graphs

DFS and `explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find a path. However, this is often not an efficient path.

2.1 Goal

Given a graph G with two vertices s and t in the same connected component, find the *best* path from s to t. What do we mean by the best?

- Least expensive.
- Best scenery.
- Shortest.

For now, we want the fewest edges.

2.2 Observation

If there is a path from s to v with length at most d, then there is some w adjacent to v with a length at most $\leq (d-1)$ for a path from s to w.

2.3 Algorithm Idea

For each d, create a list of all vertices at distance d from s.

- For $d = 0$, this is just $\{s\}$.
- For larger d, we want all new vertices adjacent to vertices at distance $d - 1$.

```
ShortestPaths(G, s)
    Initialize Array A
    A[0] = {s}
    dist(s) = 0
    For d = 0 to n
        For u in A[d]
            For (u, v) in E
                if dist(v) undefined
```

```
9                         dist(v) = d + 1
10                        add v to A[d + 1]
```

How can we improve this?

- What if `dist(v)` undefined at end? We can set the distances of all vertices to undefined.
- The algorithm goes through `A[0]`, `A[1]`, in order. We can just use a queue.

```
1    ShortestPaths(G, s)
2        Initialize Queue Q
+        Q.enqueue(s)
4        dist(s) = 0
+        While Q not empty
+        u = front(Q)
7            For (u, v) in E
8                if dist(v) = infinity
9                    dist(v) = dist(u) + 1
10                   Q.enqueue(v)
```

- What if we want to keep track of the paths?

```
1    ShortestPaths(G, s)
2        Initialize Queue Q
3        Q.enqueue(s)
4        dist(s) = 0
5        While Q not empty
6        u = front(Q)
7            For (u, v) in E
8                if dist(v) = infinity
9                    dist(v) = dist(u) + 1
10                   Q.enqueue(v)
+                    v.prev = u
```

2.4 Breadth First Search

In our last change above, we note that we simply have BFS.

```
BFS(G, s)
    For v in V, dist(v) = infinity
    Initialize Queue Q
    Q.enqueue(s)
    dist(s) = 0
    While Q is not empty
        u = front(Q)
        For (u, v) in E
            If dist(v) = infinity
                dist(v) = dist(u) + 1
                Q.enqueue(v)
                v.prev = u
```

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

2.5 DFS vs. BFS

- Similarities:
  - The way both algorithms process vertices is the same (`visited` vs. `dist < infinity`).
  - For each vertex, process all unprocessed neighbors.
- Differences:
  - DFS uses a stack to store vertices waiting to be processed.
  - BFS uses a queue.
- Big Effect:
  - DFS goes depth-first: very long path. Get a very "skinny" tree.
  - BFS is breadth first: visits all side paths. Get a very shallow tree since we process all of the neighbors.

2.6 Edge Length

The number of edges in a path is not always the right measure of distance. Sometimes, taking several shorter steps is preferably to taking a few longer ones.

We can assign each edge (u, v) a non-negative length $\ell(u, v)$. The length of a path is the sum of the lengths of its edges.

2.7 Problem: Shortest Path

Coming soon!