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'.

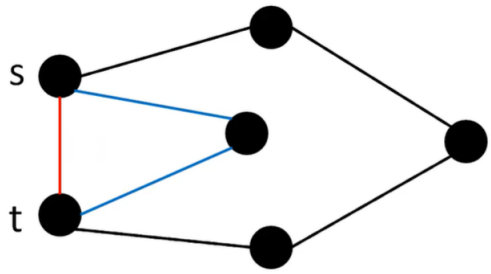
1.1 Better Algorithm

```
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     // if v is not visited
            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. This is because DFS picks one path and tries to go to the end of that path before trying a different path. If that path happens to be a not-so-ideal path, then we've taken a much longer path than necessary. For example, consider the following graph:



Suppose we wanted to get from s to t. Using DFS, we might take the black path; the one with 4 edges. How do we guarantee that we take the shortest path, i.e. the red path?

2.1 Goal

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

- Least expensive.
- Best scenery.
- Shortest.

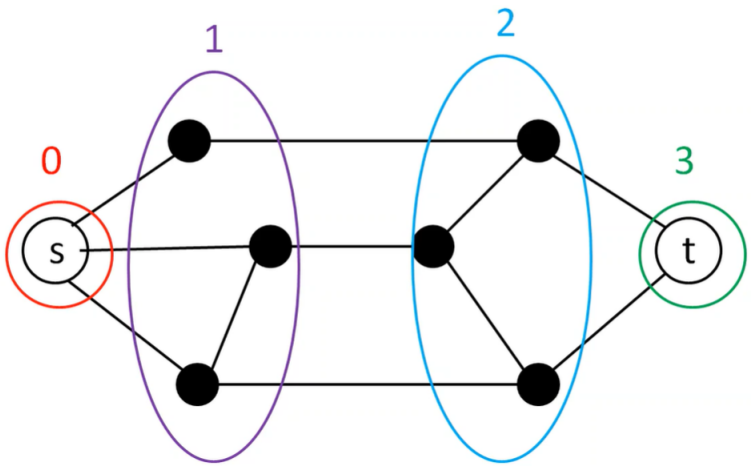
For now, we want the fewest edges.

2.2 Observation

Proposition. *If there is a path from vertices s to v with length at most d, then there is some w adjacent to v where there is a path a length at most $(d-1)$ from vertices s to w.*

Proof. If w is the next-to-last vertex on the path, then there are d edges to get you from s to v and that means the number of edges you have to take from s to w is only $d-1$. This means that if we know all of the vertices with distance at most $d-1$, then we can find all of the vertices with distance at most d. □

For example, consider the following (annotated) graph:



Here, suppose we start at s.

- The only place we can get with a path of length 0 is s, or the one vertex in the red circle labeled `0`.
- If we want to travel one edge, the only places we can get with a path of length one are the vertices in the purple circle labeled `1`.
- If we want to travel two edges, we can get to some vertex that is adjacent to one of the vertices in the purple circle. For each of the purple vertices, if we list all of the adjacent vertices, the only new vertices we can reach are the vertices in the cyan circle labeled `2`.
- Finally, if we want to travel three edges, we can get to something that is adjacent to the vertices at distance 2, which means we can get to t, or the vertex in the green circle labeled `3`.

By the way we constructed these sets (the circles), it's not possible to get to t at distance 2 because, for that to happen, it has to be adjacent to the vertices at distance 1, which it isn't.

2.3 Algorithm Idea

This observation gives us a pretty reasonable way to find an algorithm. In particular, if we want to find the best path length between s and t, we really need to find the best path length between s and every other vertex in the graph.

So, the idea is, 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 $d-1$.

2.3.1 Algorithm

The algorithm is as follows:

```
1    ShortestPaths(G, s)
2        Initialize Array A
3        A[0] = {s}
4        dist(s) = 0
5        For d = 0 to n
6            For u in A[d]
7                For (u, v) in E
8                    if dist(v) undefined
9                        dist(v) = d + 1
10                       add v to A[d + 1]
```

2.3.2 Improving the Algorithm

How can we improve this?

- What if `dist(v)` undefined at end? We can set the distances of all vertices to infinity.[1]

```
1    ShortestPaths(G, s)
2+       For each v in V, dist(v) = infinity
3        Initialize Array A
4        A[0] = {s}
5        dist(s) = 0
6        For d = 0 to n
7            For u in A[d]
8                For (u, v) in E
9c                   if dist(v) = infinity
10                       dist(v) = d + 1
11                       add v to A[d + 1]
```

  Here, at the end of the algorithm, any vertices that could not be found will be assigned a distance of infinity to indicate that it could not be reached.

- The algorithm goes through `A[0]`, `A[1]`, and so on in order. We can just use a queue.

```
1    ShortestPaths(G, s)
2        For each v in V, dist(v) = infinity
3c       Initialize Queue Q
5        dist(s) = 0
6c       While Q not empty
7c           u = front(Q)
8            For (u, v) in E
9                if dist(v) = infinity
10c                  dist(v) = dist(u) + 1
11c                  Q.enqueue(v)
```

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

```
1    ShortestPaths(G, s)
2        For each v in V, dist(v) = infinity
3        Initialize Queue Q
4        dist(s) = 0
5        While Q not empty
```

[1] Note that `+` next to a line number means added code and `c` next to a line number means changed line.

```
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)
11+                 v.prev = u     // Keep track of path
```

With this change, we can simply follow the chain of `previous` vertices, which is the path that was taken.

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
```

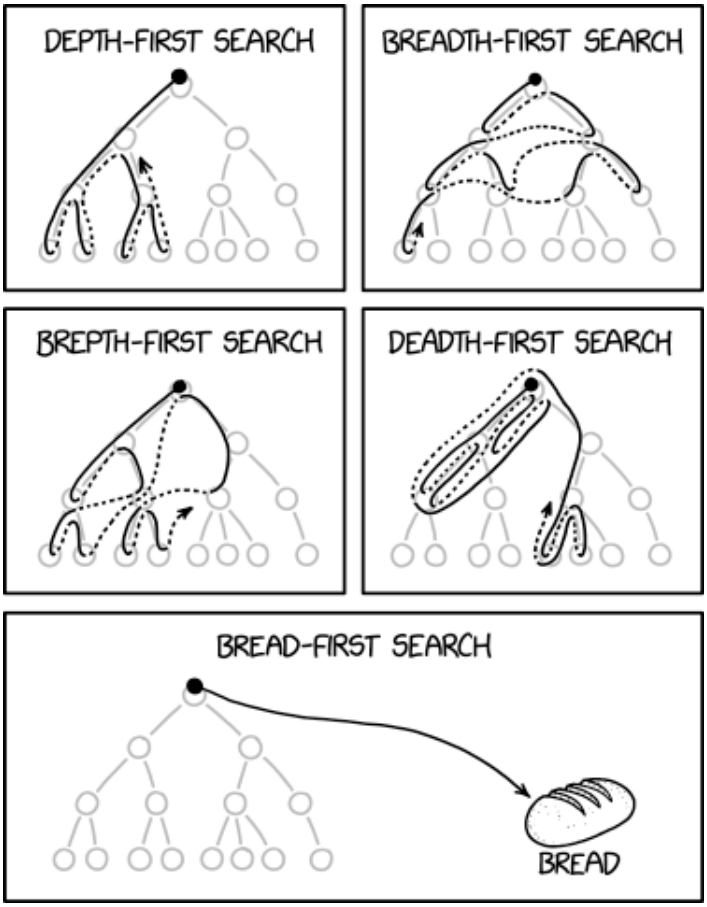
The total runtime is $O(|V| + |E|)$.

- In the first few lines, we have $O(|V|)$ time.
- In the while loop and the `front(Q)` lines, we have $O(|V|)$ iterations.
- In the edge iteration, we have $O(|E|)$ iterations.

2.5 DFS vs. BFS

- Similarities:
  - The way both algorithms process vertices is the same (`visited` for DFS vs. `dist < infinity` for BFS).
  - 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.

2.5.1 Relevant XKCD



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!