DFS

Idea. DFS starts at the root and explores as far as possible along each branch before backtracking.

Complexity. $O(|E| + |V|) = O(|V|^2)$.

Used used to find...

- **SCCs**: Here is how:
  - Find a vertex that is guaranteed to be in a sink SCC
  - After identifying a sink SCC, somehow continue

```
run DFS on G reversed
for v in V ordered by post num. desc.
  if not visited[v]
    run DFS on G from v
    output vertices seen as a SCC
```

- **Topological sort**: given a DAG, produce a linear order consistent with it by running DFS and sorting by descending post number. (can't do it if it's not a DAG)
- **Shortest paths in DAGs**: topological sort the vertices and for each vertex u, taken in the order of the topological sort, relax every edge $u \to v$.

BFS

Idea. BFS begins at the root vertex and explores all the neighbors. Then for each of those nearest vertices, it explores their unexplored neighbors, and so on, until it finds the goal.

BFS produces a tree such that all vertices at depth d in the tree are exactly d hops away from the root.

Complexity. $O(|E| + |V|) = O(|V|^2)$.

DFS vs BFS

They are both using basically the same idea:

```
Main loop:
  v = EXTRACT-NEXT(X)
  mark v as explored
  for each u where (u, v) in E:
    if v not explored:
      INSERT(u, X)
```

They are only using different data structures:

- For DFS, `X` is a stack (first in, first out), this is why DFS explores the children of the current vertex before exploring the other vertices at the same level.
- For BFS, `X` is a queue (first in, last out), this is why BFS explores first the other vertices at the same level before the children of the current vertex.

Dijkstra

For a given source vertex, Dijkstra finds the path with lowest cost between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined.

Complexity: $O(|E|) = O(|V|^2)$.

Bellman-Ford

Idea. Bellman-Ford relaxes all the edges, and does this $|V| - 1$ times. The repetitions allow minimum distances to accurately propagate throughout the graph, since, in the absence of negative cycles, the shortest path can only visit each vertex at most once.

```
repeat |V|-1 rounds:
  for all vertices u:
    if d(s,u) + weight(u,v) < d(s,v):
      update distance to v
```

Complexity. $O(|E| \cdot |V|) = O(|V|^3)$.

Bellman-Ford vs Dijkstra

Bellman-Ford relaxes all edges $|V| - 1$ times, while Dijkstra's algorithm greedily selects the not-yet-visited minimum-weight vertex.

They are both used to find the shortest path: Dijkstra does not work with negative edges and has a complexity of $O(|V|^2)$ whereas Bellman-Ford does work with negative edges (without negative cycle) but is slower: $O(|V|^3)$.

Prim

Idea. Prim starts from an arbitrary vertex and at every step, finds the cheapest edge that extends the current tree.

Complexity: $O(|E|) = O(|V|^2)$

Prim vs Dijkstra

Both are building trees, in similar ways.

- Prim is building a Minimum Spanning Tree.

```
Main loop:
  Iteratively add edge (u,v) to T,
  such that u in T, v not in T,
  and d(u, v) is minimum
```

- Dijkstra is building a Shortest Path Tree.

```
Main loop:
  Iteratively add edge (u,v) to T,
  such that u in T, v not in T,
  and l(u, v) + d(u, v) is minimum
```

How to find the shortest path?

It depends on the graph:

- If the graph is a DAG: topological sort the vertices with **DFS** and for each vertex u, taken in the order of the topological sort, relax every edge $u \to v$.
- If the graph has no negative edges, run **Dijkstra**.
- If the graph has some negative edges but no negative cycles, run **Bellman-Ford**.
- If the graph has negative cycles, we don't even know what is a shortest path.