Graph Search

1 Generic Search

```
def GenericSearch(G, start_vertex):
    mark all G unexplored, S explored
    while unexplored exist:
        choose edge (u, v) with u explored, v unexplored
        mark v explored
```

The difference in how the two most significant graph search algorithms, **breadth-first search** and **depth-first search** work is how each chooses the next node to explore.

BFS explores nodes in "layers", exploring all of a vertex's neighbors before moving another level away. DFS explores aggressively, going as deep as possible before backtracking if necessary. BFS is often implemented with a queue (FIFO) structure, while DFS is implemented with a stack (FILO) or recursively. Both algorithms are $O(m + n)$.

2 BFS

```
def BFS(G, start):
    mark start explored
    enqueue(start)

    while !empty(queue):
        v = dequeue()
        mark v explored
        for (v, w) in G:
            if w unexplored:
                enqueue(w)
```

Runtime is actually $O(m_s + n_s)$, where s means the edges and nodes reachable from the starting vertex.

2.1 Shortest Path Problem

Given an undirected graph with equally-weighted edges, it is simple to adapt BFS to compute the shortest possible path between two vertices. Initialize all vertices with distance ∞ except the starting vertex with distance 0, and conduct BFS except if `w` is unexplored, then `dist(w) = dist(v) + 1`.

2.2 Undirected Connectivity

A connected component is an undirected subgraph where any two vertices are connected. They are the "pieces" of an undirected grpah, and there does not exist a path between two connected components. BFS can be adapted to find these subgraphs.

```
def FindConnections(G):
   for V in G:
      if (V unexplored):
         BFS(G, V)   //Finds the connected vertices
```

3 DFS

```
def DFS(G, start):
   start explored
   for (s, v) in G:
      if v unexplored:
         mark v explored
         DFS(G, v)
```

It is also possible to implement DFS with a stack if the recursive version busts the call stack.

3.1 Topological Sort

Topological ordering of a directed graph G is a labeling function f of G's nodes such that

$$\forall (u, v) \in G, f(u) < f(v) \tag{1}$$

Note that this is only possible if there are no cycles present in G, as topological sorting turns G into a **directed acyclic graph**, or DAG.

```
def DFS-Loop(G):
   label = n
   for v in G:
      if v unexplored:
         DFS_Top(G, v)
```

```
def DFS_Top(G, v):
   v explored
   for (v, w) in G:
      if w unexplored:
         DFS_Topology(G, w)
   f(start) = label--
```

If we order the graph by increasing $f(v)$, then G is clearly a DAG. This algorithm runs in $O(m + n)$ time if there are no cycles in G.