Graph Traversal

1 BFS and DFS

Keep track of visited list and return list separately.

Breadth-First Search (BFS):

```
enqueue start, adding visited set

while queue not empty:
    v = queue.dequeue
    add v to return list

    for each neighbor u of v:
        if u not in visited:
            enqueue u
            add u to visited
```

Depth-First Search (DFS) iteratively:

```
push start onto stack

while stack not empty:
    v = stack.pop
    if v not visited:
        add v to return list
        add v to visited set

        for each neighbor u of v:
            if u not in visited:
            push u onto stack
            # do not add to visited after pushing!
```

DFS recursively:

```
#Needs a public wrapper to instantiate the list
private function dfs(start, list, visited):
    if v not in visited:
        add v to visited
        add v to list

        for each neighbor u of v:
            dfs(u, list, visited)
```

Time complexity: $O(|V| + |E|)$, where V and E are the amounts of vertices and edges, respectively.

2 Dijkstra's Algorithm

- Weighted edges
- Finds shortest path
- Structures: visited set, priority queue (VDP), start, map of distances

Algorithm:

```
instantiate map
for all vertices: #(keys)
    initialize value to infinity

priorityqueue.add(start, 0)
#or map.put(start, 0) and add neighbors to priorityqueue

while priorityqueue not empty and visited.size < number of vertices:
    v = priorityqueue.remove
    if v is not visited:
        mark v as visited
        update value for v in map
        for all neighboring VDPs: #obtained from graph.getAdjList().get(v.vertex)
            newPath = v.distance + u.distance
            if newPath to u's vertex < map.get(u.vertex):
                add VDP(u.vertex, newPath)
```

To find the neighbors of a vertex v: graph.getAdjList().get(v)

Time complexity: $O[(|E| + |V|) \cdot \log |V|]$

3 Kruskal's Algorithm

Minimum Spanning Tree algorithm.