Simple Graph Traversal

Julian Tu

1 Introduction

The two methods of graph traversal described below are hard to give concrete use cases for without describing specific problems. This is because of their generality. Thus, while learning their applications and implementations, it is also important to take note of the programming concepts that they utilise. DFS is a good introduction to recursion. Always consider the possible things that the function could return, outside of marking nodes in a seen array.

2 Depth First Search

2.1 Possible Uses

Finding all nodes reachable from a given node.
Detecting cycles in a graph.
Generating an ordering of the nodes in a graph.

2.2 Implementation

2.2.1 C++

```
bool seen[N] = {false};
vector<int> adj[N];
void dfs(int at) {
    seen[at] = true;
    for (int i = 0; i < adj[at].size(); i++) {
        if (!seen[adj[at][i]]) {
            dfs(adj[at][i]);
        }
    }
}
```

2.2.2 Python

```
seen = {i: False for i in graph}
def dfs(at):
    seen[at] = True
```

```
    for neighbour in adj[at]:
        if not seen[neighbour]:
            dfs(neighbour)
```

3 Breadth First Search

3.1 Possible Uses

Finding the shortest path from a single source to all other nodes (SSSP) in an unweighted graph.

3.2 Implementation

Not described below is a method of generating the shortest path from the source to a node. Currently, a trivial change only allows us to determine the distance between two nodes. The method of finding the path itself is left as an excercise to the reader.
Both implementations should initialise the queue with the source node.

3.2.1 C++

```
bool seen[N] = {false};
vector<int> adj[N];
while (!queue.empty()) {
    int at = queue.front();
    seen[at] = true;
    for (int i = 0; i < adj[at].size(); i++) {
        if (!seen[adj[at][i]]) {
            queue.push(adj[at][i]);
        }
    }
}
```

3.2.2 Python

Note that Python implementations will be a bit problematic as there is no built-in queue implementation. If the following does not satisfy the time requirements consider workarounds such as maintaining two stacks in parallel and alternating between pushing and popping between the two.

```
seen = {i: False for i in graph}
while queue:  # False when queue is empty
    at = queue.pop(0)
    seen[at] = True
    for neighbour in adj[at]:
        if not seen[neighbour]:
            queue.append(at)
```