Graph Traversal Algorithms – Overview

Visually a graph traversal is typically drawn as a decision tree.

When traversing all the nodes through a graph, regardless of the algorithm used, we can face the following issues:

1. There may be cases in which the graph is not connected, therefore not all nodes can be reached.
2. In the case of cyclic (starts and ends at the same node) graphs, we should make sure that cycles do not cause the algorithm to go into an infinite loop.
3. We may need to visit some nodes more than once, since we do not know if a node has already been seen, before transitioning to it.

Graph traversal algorithms can solve the second and third problems by flagging vertices as visited when appropriate: (1) at first, no node is flagged as visited; (2) when the node is visited, we flag it as visited during the traversal; (3) a flagged node is not visited a second time. This keeps the program from going into an infinite loop when it encounters a cycle.

Traversing a graph is, without a doubt, one of the most useful processes when dealing with graphs. In this blog we will describe the two most frequent methods when traversing a graph:

- Depth first search (DFS)
- Breadth first search (BFS)

Depth First Search (DFS)

In this algorithm, we follow one path as far as it will go. The algorithm starts in an arbitrary node, called root node and explores as far as possible along each branch before backtracking.

The algorithm can also be applied to directed graphs. DFS is implemented with a recursive algorithm and its temporal complexity is O(E+V).

HOW DOES IT WORK CONCEPTUALLY?

The DFS algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored.

Breadth First Search (BFS)

In the BFS algorithm, rather than proceeding recursively, we pull out the first element from the queue, check if it has a path, check if it is the destination node we are interested in and if not add all the children nodes to it. We look at all the nodes adjacent to one before moving on to the next level.

The algorithm can also be applied to directed graphs. The algorithm's time complexity is O(E+V).

HOW DOES IT WORK CONCEPTUALLY?

The BFS algorithm starts by searching a start node, followed by its adjacent nodes, then all nodes that can be reached by a path from the start node containing two edges, three edges, and so on.

DFS vs. BFS

As discussed here, BFS is used to search nodes that are closer to the given source; while DFS is used instead in cases where the solution is away from the source.

When coming to practical examples, BFS is typically used to find the shortest distance between two nodes, such as routing for GPS navigation; while DFS is more suitable for e.g. game/puzzle problems: we make a decision, then explore all paths through this decision, and if this decision leads to a win situation, then we stop.

Traversal algorithms are also a fundamental component for numerous additionally complex graph algorithms. For instance, the DFS and BFS traversal algorithms often show up bundled into more advanced graph algorithms, such as:

- Shortest path in an unweighted graph

- Topological sorting
- Strongly connected component
- 2 Satisfaction (2-SAT)
- Lowest Common ancestor (LCA)
- Max Flow / Min Cut

```
#BFS
def bfs(graph,node):
    visited=[]
    queue=[]
    visited.append(node)
    queue.append(node)

    while queue:
        s=queue.pop(0)

        for x in graph[s]:
            if x not in visited:
                visited.append(x)
                queue.append(x)
    return visited#DFS
def dfs(graph,node):
    visited=[]
    queue=[]

    queue.append(node)
    visited.append(node)

    while queue:
        s=queue.pop()
        print(s)        for x in graph[s][::-1]:
            if x not in visited:
                visited.append(x)
                queue.append(x)
```