Graph Traversal Algorithms

before start in graph traversal we want to show a quick definition of graph below:
Graph: A graph G(V, E) is a non-linear data structure that consists of node and edge pairs of objects connected by links,that is used specially for searching and exploring.

1. **searching Algorithm:**
    a) **Blind search Algorithm (**Also called an uninformed search**)**:
    is a search that has no information about its domain or nature problem.
    Example:
        1. Breadth First Search (BFS)
        2. Depth - First Search (DFS)

    They are the two most frequent methods when traversing a graph and we will talk deeply about it at the rest of this article.

    b) Heuristic search Algorithm (Also called an informed or directed search**)**:
    It has further information about the cost of the path between any state in search space and the goal state.
    Example:
        1. Best - First Search
        2. A - Star (A*)
        3. Tabu search

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.

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.

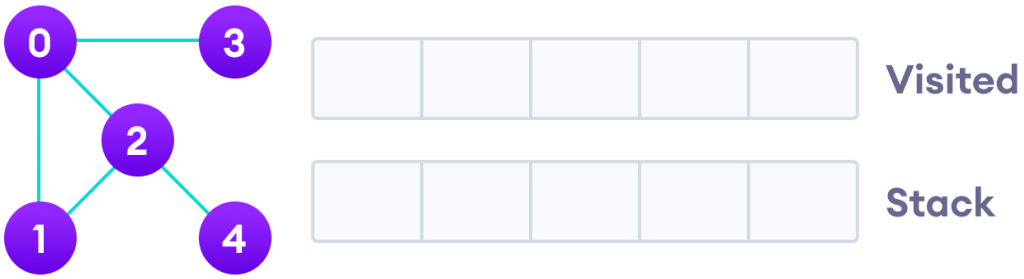
The algorithm can also be applied to directed graphs. DFS is implemented with a recursive algorithm and its temporal complexity is O(E+V).

HOW DOES IT WORK CONCEPTUALLY?

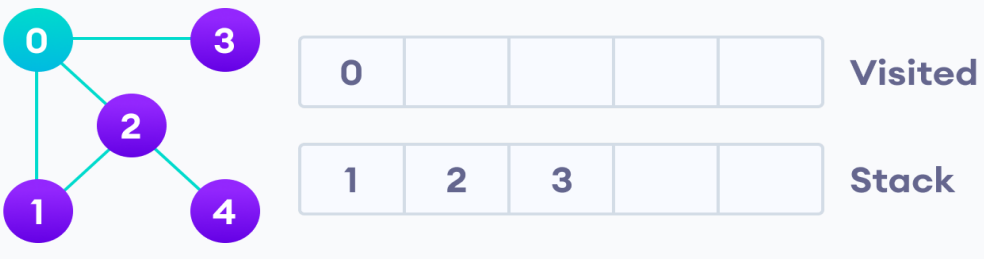
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Depth First Search Example

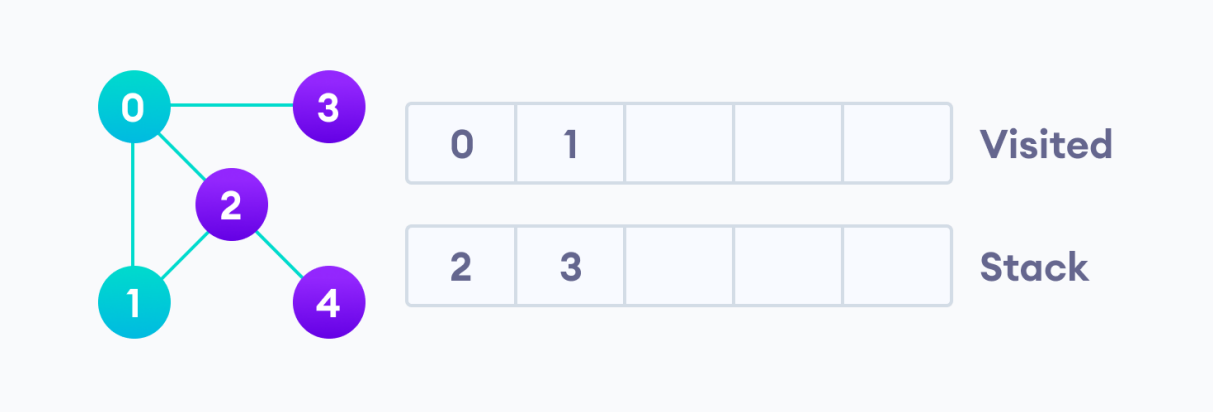
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



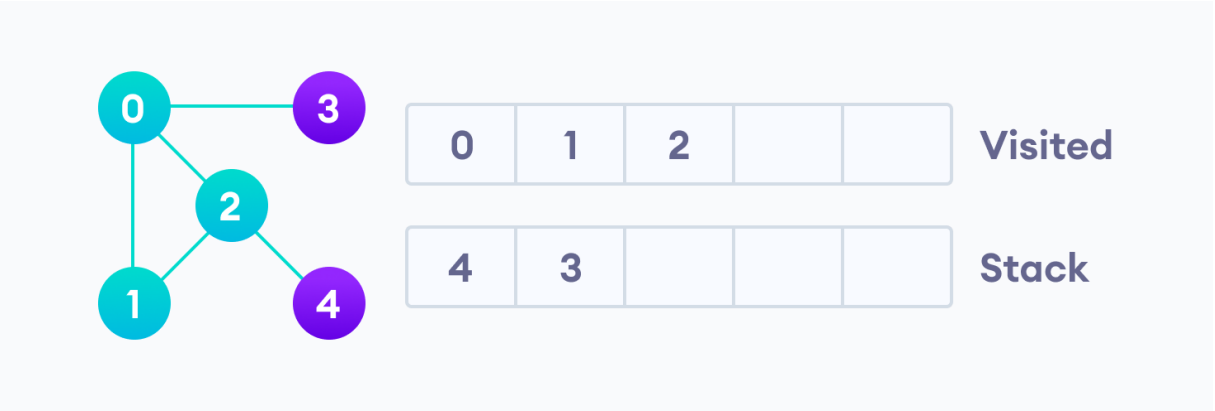
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

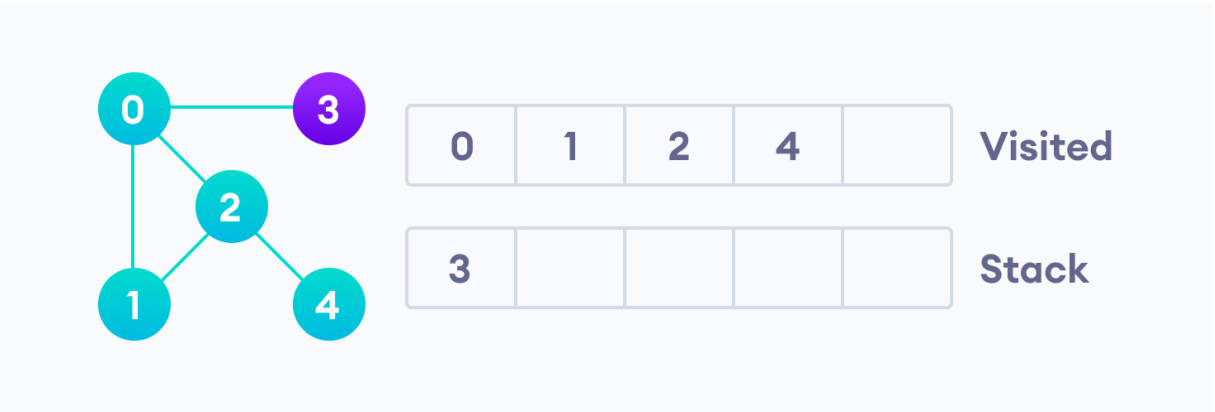


Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

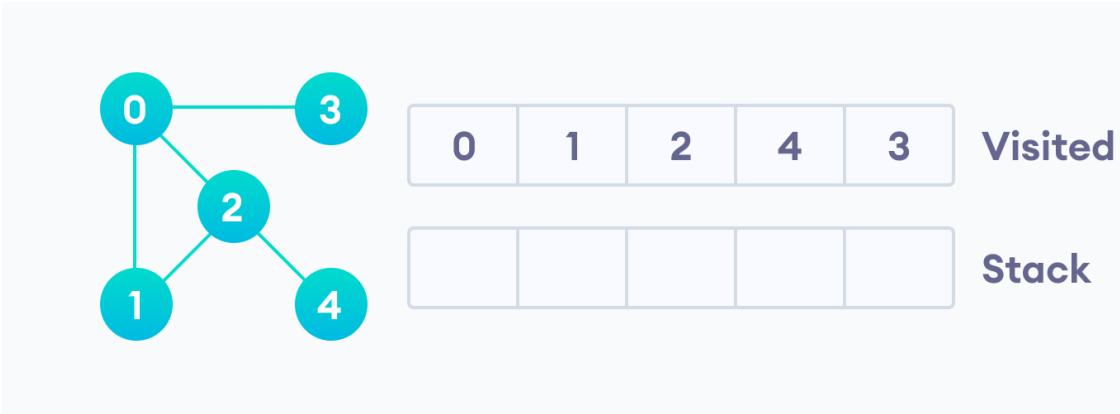


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.





After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



DFS Implementation in Python:

The code for the Depth First Search Algorithm with an example is shown below. The code has been simplified so that we can focus on the algorithm rather than other details.

```
# DFS algorithm in Python

# DFS algorithm

def dfs(graph, start, visited=None):

    if visited is None:

        visited = set()

    visited.add(start)

    print(start)

    for next in graph[start] - visited:

        dfs(graph, next, visited)

    return visited

graph = {'0': set(['1', '2']),

         '1': set(['0', '3', '4']),

         '2': set(['0']),

         '3': set(['1']),

         '4': set(['2', '3'])}

dfs(graph, '0')
```

Complexity of Depth First Search:

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Application of DFS Algorithm:

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

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.

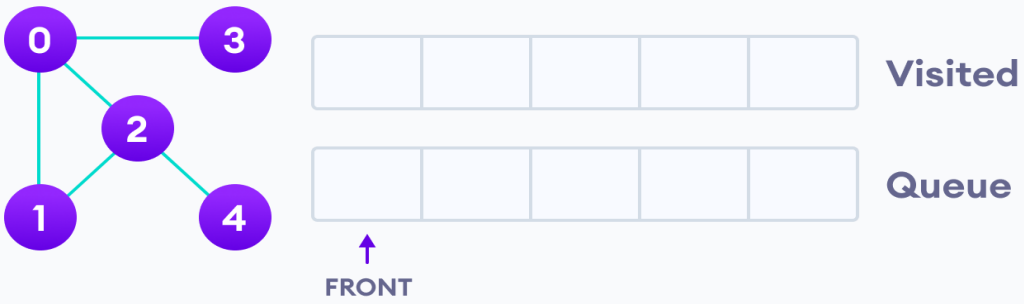
The algorithm can also be applied to directed graphs. The algorithm's time complexity is O(E+V).

HOW DOES IT WORK CONCEPTUALLY?

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

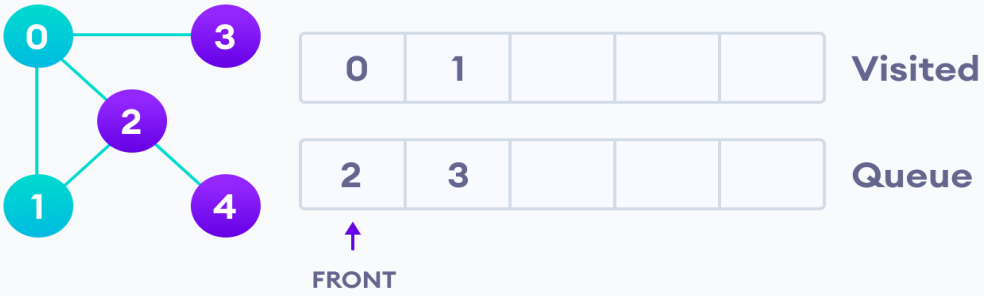
BFS example

Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.

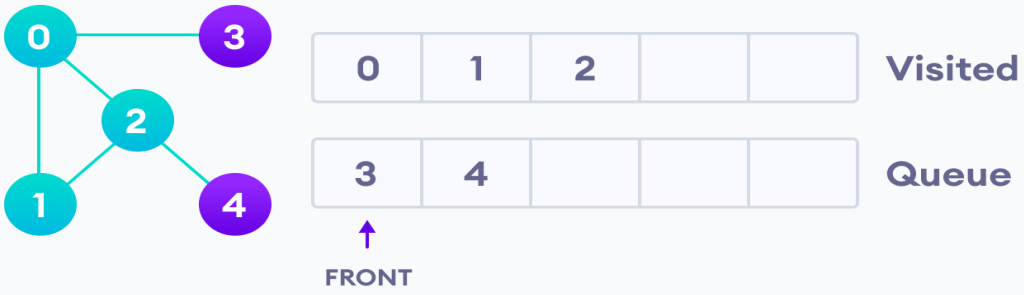


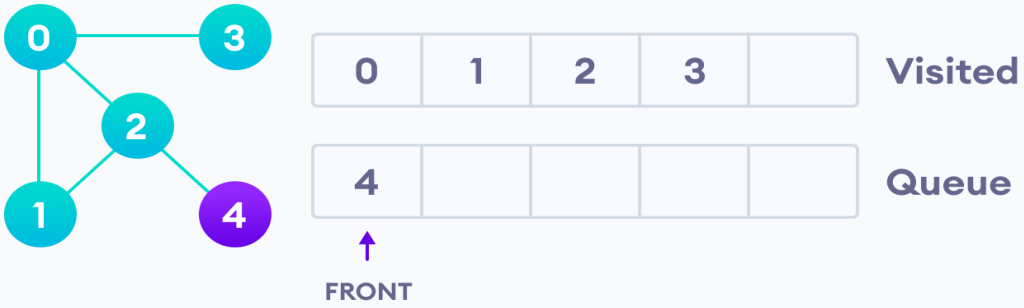
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.





Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited,So we visit it.

Since the queue is empty, we have completed the Breadth First Traversal of the graph.

BFS Implementation in Python:

The code for the Breadth First Search Algorithm with an example is shown below. The code has been simplified so that we can focus on the algorithm rather than other details.

```
# BFS algorithm in Python

import collections

# BFS algorithm

def bfs(graph, root):

    visited, queue = set(), collections.deque([root])

    visited.add(root)

    while queue:

        # Dequeue a vertex from queue

        vertex = queue.popleft()

        print(str(vertex) + " ", end="")

        # If not visited, mark it as visited, and

        # enqueue it

        for neighbor in graph[vertex]:

            if neighbor not in visited:

                visited.add(neighbor)

                queue.append(neighbor)

if __name__ == '__main__':

    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}

    print("Following is Breadth First Traversal: ")

    bfs(graph, 0)
```

BFS Algorithm Complexity:

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
The space complexity of the algorithm is O(V).

BFS Algorithm Applications:

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In minimum spanning tree

DFS vs. BFS

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