2. Breadth First Search

01/08/2019

INTRODUCTION

BFS is a traversing algorithm where we start traversing from a selected node (source or starting node) and traverse the graph layer-wise thus exploring the neighbour nodes (nodes which are directly connected to source node). We then move towards the next-level neighbour nodes. Formally, the BFS algorithm visits all vertices in a graph G that are k edges away from the source vertex s before visiting any vertex k+1 edges away. This is done until no more vertices are reachable from s.

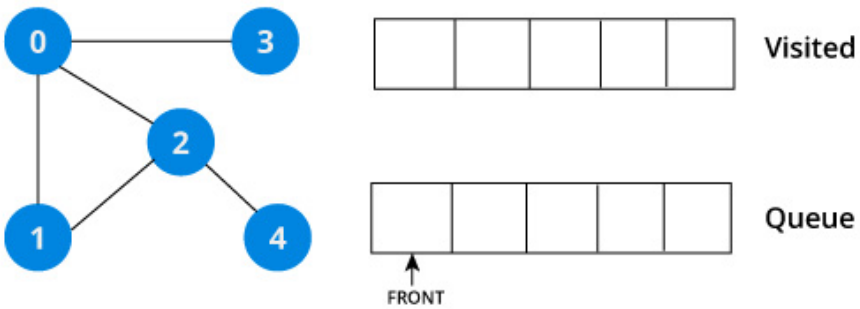
For a graph G = (V, E) and a source vertex v, breadth-first search traverses the edges of G to find all reachable vertices from v. It also computes the shortest distance to any reachable vertex. Any path between two points in a breadth-first search tree corresponds to the shortest path from the root v to any other node s.

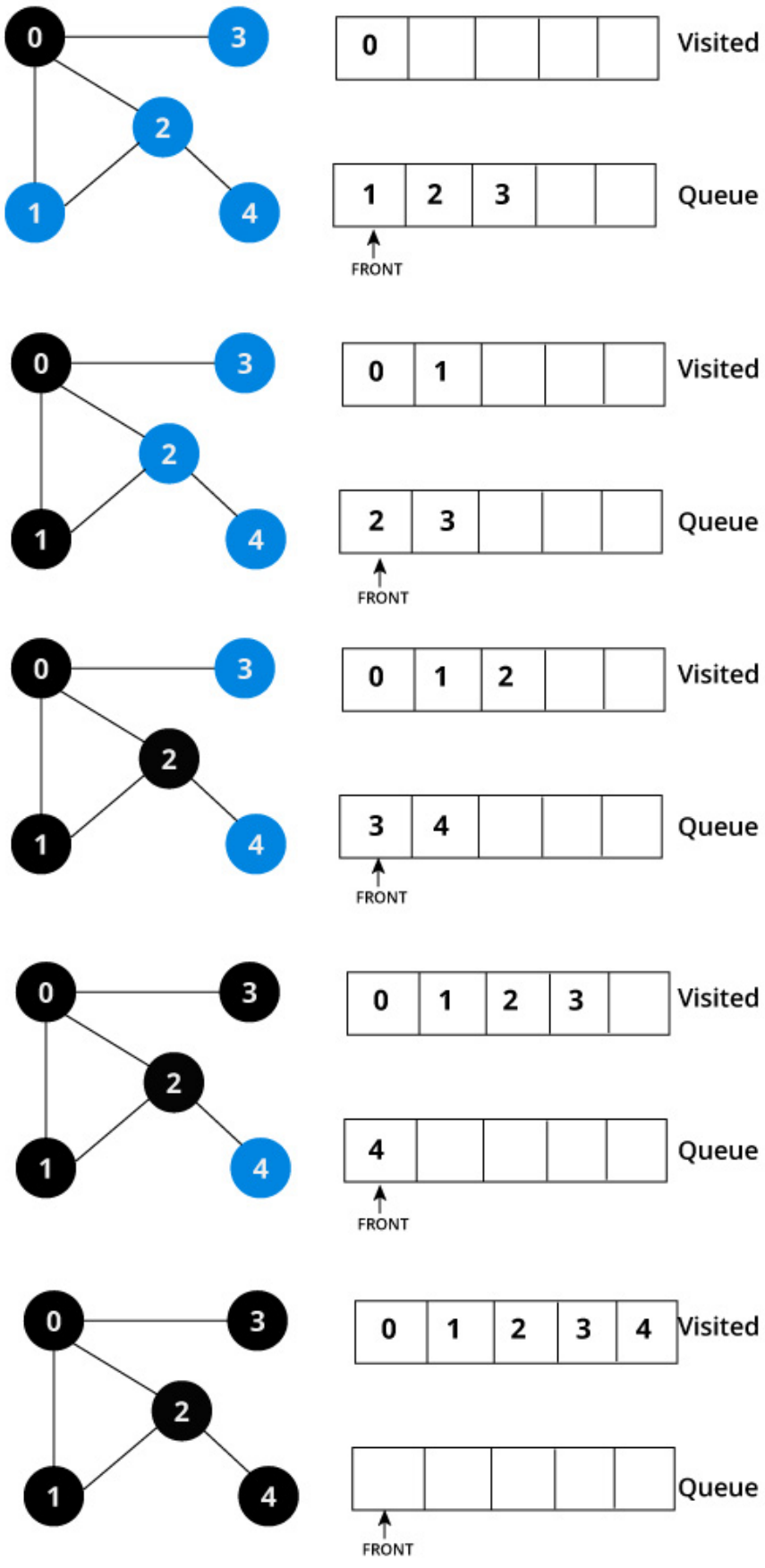
ALGORITHM OF BFS

```
1  procedure BFS(G,start_v):
2      let Q be a queue
3      label start_v as discovered
4      Q.enqueue(start_v)
5      while Q is not empty
6          v = Q.dequeue()
7          if v is the goal:
8              return v
9          for all edges from v to w in G.adjacentEdges(v) do
10             if w is not labeled as discovered:
11                 label w as discovered
12                 w.parent = v
13                 Q.enqueue(w)
```

IMPLEMENTATION USING DATA STRUCTURE

BFS uses **Queue** data structure.

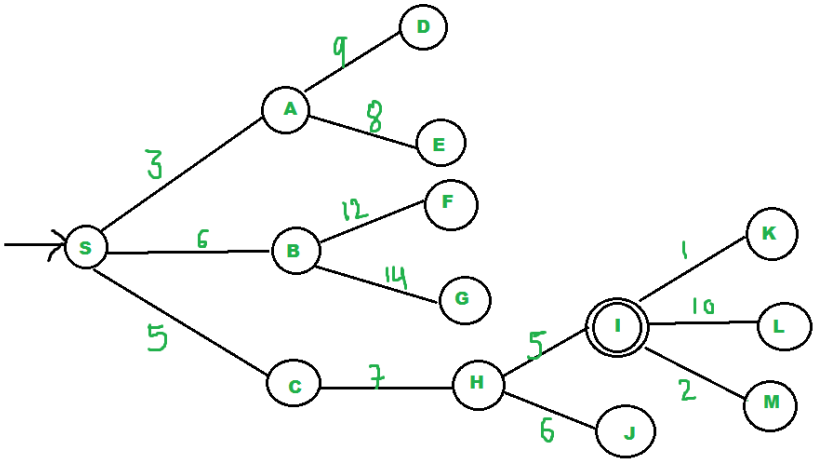




Since the queue is empty, we have completed the Breadth First Search.

PYTHON IMPLEMENTATION OF BFS

```
class Graph:

    def __init__(self, num):
        self.graph = [[] for i in range(num)]

    def addEdge(self, u, v, cost):
        self.graph[u].append((v, cost))
        self.graph[v].append((u, cost))

    def bfs(self, start, end):
        visited = [False for i in self.graph]
        queue = [start, ]
        visited[start] = True
        path = []

        while True:
            if len(queue) != 0:
                curr_node = queue.pop(0)
            else:
                break
            path.append(curr_node)
            if curr_node == end:
                print("Path found from %d to %d (%s)" % (
                    start, end, ' -> '.join(map(str, path))
                ))
                break
            for node, _ in self.graph[curr_node]:
                if not visited[node]:
                    visited[node] = True
                    queue.append(node)

if __name__ == "__main__":
    graph = Graph(14)
    graph.addEdge(0, 1, 3)
    graph.addEdge(0, 2, 6)
    graph.addEdge(0, 3, 5)
    graph.addEdge(1, 4, 9)
    graph.addEdge(1, 5, 8)
    graph.addEdge(2, 6, 12)
    graph.addEdge(2, 7, 14)
    graph.addEdge(3, 8, 7)
    graph.addEdge(8, 9, 5)
    graph.addEdge(8, 10, 6)
    graph.addEdge(9, 11, 1)
    graph.addEdge(9, 12, 10)
    graph.addEdge(9, 13, 2)

    graph.bfs(0, 9)
```



Output:

Path found from 0 to 9 (0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9)

APPLICATIONS OF BFS

1. **Shortest Path and Minimum Spanning Tree for Unweighted Graph:** In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
2. **Peer to Peer Networks:** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbour nodes.
3. **Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same.
4. **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
5. **Cycle detection in undirected graph:** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.
6. **To test if a graph is Bipartite:** We can either use Breadth First or Depth First Traversal.
7. **Path Finding:** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
8. **Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

ADVANTAGES AND DISADVANTAGES OF BFS

Advantages of BFS:

1. Solution will definitely found out by BFS If there are some solution.
2. BFS will never get trapped in blind alley , means unwanted nodes.
3. If there are more than one solution, then it will find solution with minimal steps.

Disadvantages Of BFS

1. There are some memory constraints as it stores all the nodes of present level to go for next level.
2. If solution is far away, then it consumes time.

COMPLEXITY OF BFS

The time complexity can be expressed as, since every vertex and every edge will be explored in the worst case is the number of vertices and is the number of edges in the graph. Note that may vary between and, depending on how sparse the input graph is.

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as , where is the cardinality of the set of vertices. This is in addition to the space required for the graph itself, which may vary depending on the graph representation used by an implementation of the algorithm.

CONCLUSION

Breadth-first search is less space-efficient than depth-first search because BFS keeps a priority queue of the entire frontier while DFS maintains a few pointers at each level.

If it is known that an answer will likely be found far into a tree, DFS is a better option than BFS. BFS is good to use when the depth of the tree can vary or if a single answer is needed—for example, the shortest path in a tree. If the entire tree should be traversed, DFS is a better option.

BFS always returns an optimal answer, but this is not guaranteed for DFS.