

Artificial Intelligence Lab

Exercise 4: Depth First Search and Breadth-First Search

AIM: To implement a depth-first search and breadth-first search

INTRODUCTION:

What is depth-first search (DFS)?

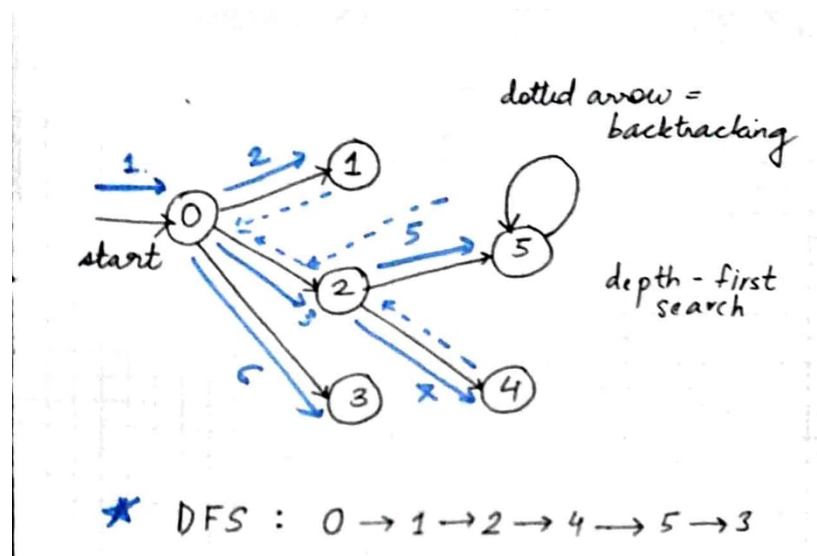
Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

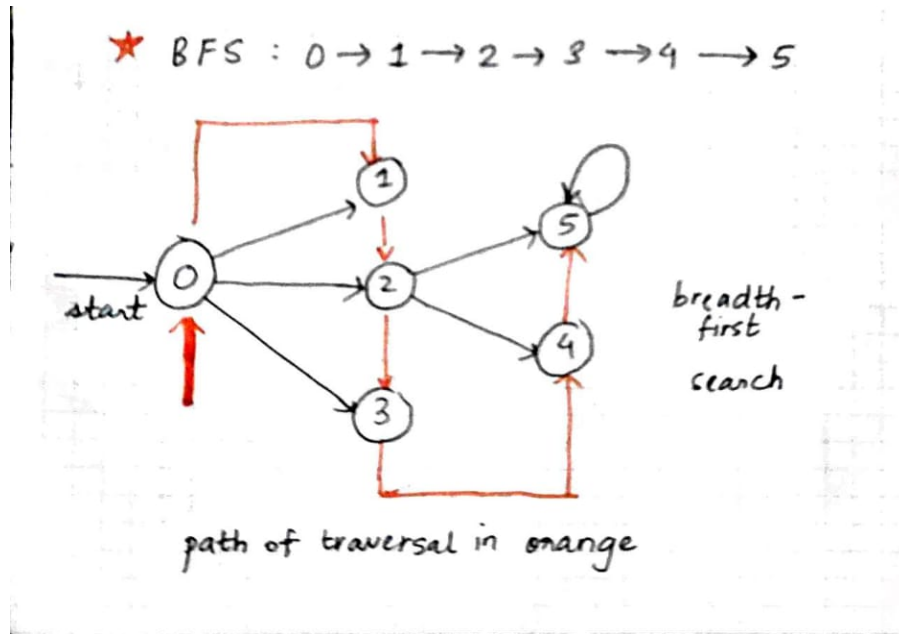
What is breadth-first search (BFS)?

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

DEMONSTRATION:

Both of the above algorithms can be executed by the example of a simple directed graph.





ALGORITHM FOR DFS:

1. Take an input as a graph or tree.
2. Create a recursive function to mark the nodes.
3. Select an arbitrary node in the graph as the root node.
4. Move to an adjacent unmarked node.
5. Mark it as visited and add to an array.
6. Continue 4 and 5 till there are no unmarked nodes.
7. Then backtrack and call the recursive function on the next unmarked node.
8. Repeat 7 till all nodes are marked.
9. Print the final array as the path.

PROGRAM FOR DFS:

```
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:
    def __init__(self):
        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
```

```
        self.graph[u].append(v)

# A function used by DFS
def DFSUtil(self, v, visited):

    # Mark the current node as visited and print it
    visited.add(v)
    print(v, end=' ')

    # Recur for all the vertices adjacent to this vertex
    for neighbour in self.graph[v]:
        if neighbour not in visited:
            self.DFSUtil(neighbour, visited)

# The function to do DFS traversal. It uses recursive DFSUtil()
def DFS(self, v):
    # Create a set to store visited vertices
    visited = set()

    # Call the recursive helper function
    # to print DFS traversal
    self.DFSUtil(v, visited)

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(0, 3)
g.addEdge(2, 4)
g.addEdge(2, 5)
g.addEdge(5, 5)

print("Following is DFS from (starting from vertex 0)")
g.DFS(0)
```

OUTPUT FOR DFS:

The screenshot shows a code editor with two tabs: 'bfs.py' and 'dfs.py'. The 'dfs.py' tab is active, displaying a Python script for Depth-First Search (DFS) on a graph. The script defines a 'Graph' class with an 'addEdge' method and a 'DFS' method. It then creates a graph with 6 vertices and 7 edges, and prints the DFS traversal starting from vertex 0. Below the code, a terminal window shows the command 'python3 dfs.py' being executed, resulting in the output: 'Following is DFS from (starting from vertex 0): 0->1->2->4->5->3->'.

```

44 # in the above diagram
45 g = Graph()
46 g.addEdge(0, 1)
47 g.addEdge(0, 2)
48 g.addEdge(0, 3)
49 g.addEdge(2, 4)
50 g.addEdge(2, 5)
51 g.addEdge(5, 5)
52
53 print("Following is DFS from (starting from vertex 0): ", end="")
54 g.DFS(0)
55 print()
56
bash - "ip-172-31-14-219" x Immediate (Javascript (br x
file dfs.py, line 57, in <module>
RA1911026010006:~/environment/RA1911026010014/Week4 $ python3 dfs.py
Following is DFS from (starting from vertex 0): 0->1->2->4->5->3->
RA1911026010006:~/environment/RA1911026010014/Week4 $

```

ALGORITHM FOR BFS:

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.

PROGRAM FOR BFS:

```

from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):

```

```

        self.graph[u].append(v)

# Function to print a BFS of graph
def BFS(self, s):
    # Mark all the vertices as not visited
    visited = [False] * (max(self.graph) + 1)
    queue = [] # Create a queue for BFS

    # Mark the source node as visited and enqueue it
    queue.append(s)
    visited[s] = True

    while queue:
        # Dequeue a vertex from queue and print it
        s = queue.pop(0)
        print (s, end = " ")

        # Get all adjacent vertices of the
        # dequeued vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and enqueue it
        for i in self.graph[s]:
            if visited[i] == False:
                queue.append(i)
                visited[i] = True

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(0, 3)
g.addEdge(2, 4)
g.addEdge(2, 5)
g.addEdge(5, 5)

print("Following is Breadth First Traversal"
      (starting from vertex 0): ")

g.BFS(0)

```

OUTPUT FOR BFS:

The screenshot shows a code editor with two tabs: 'bfs.py' and 'dfs.py'. The 'bfs.py' tab is active, displaying the following Python code:

```

48 g.addEdge(0, 1)
49 g.addEdge(0, 2)
50 g.addEdge(0, 3)
51 g.addEdge(2, 4)
52 g.addEdge(2, 5)
53 g.addEdge(5, 5)
54
55 print ("Following is Breadth First Traversal"
56       " (starting from vertex 0): ")
57 g.BFS(0)
58 print()
59

```

Below the code editor, a terminal window shows the execution of the program:

```

bash - "ip-172-31-14-219" x Immediate (Javascript (br x +)
2
RA1911026010006:~/environment/RA1911026010014/Week4 $ python3 bfs.py
Following is Breadth First Traversal (starting from vertex 0):
0->1->2->3->4->5->
RA1911026010006:~/environment/RA1911026010014/Week4 $

```

OBSERVATION:

From the above algorithms, we can see the different types of search techniques or traversals that can be applied to a tree or graph.

RESULT:

Depth-first search and breadth-first search algorithms were implemented successfully.