**ST.FRANCIS INSTITUTE OF TECHNOLOGY**

Mount Poinsur, S.V.P. Road, Borivli (West), Mumbai - 400103

**Computer Engineering Department**

**Academic Year:** 2021-2022 **Class/Branch**: BE CMPN

**Subject:** CSC 703 Artificial Intelligence & Soft Computing **Semester**: VII

**Experiment No. 3**

**BFS and DFS Algorithm**

**Aim:** To learn the working of BFS and DFS Algorithms and apply it in state space search to achieve the solution goals of well defined problems.

**Theory:**

The aim of the BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let’s see how BFS traversal works with respect to the following graph:

If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. “A B C D E F”. The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

**Algorithmic Steps**

1. Step 1: Push the root node in the Queue.

2. Step 2: Loop until the queue is empty.

3. Step 3: Remove the node from the Queue.

4. Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

The aim of the DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search. Let’s see how depth first search works with respect to the following graph:

As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be “A B E F C D”. DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

**Algorithmic Steps**

Step 1: Push the root node in the Stack.

Step 2: Loop until the stack is empty.

Step 3: Peek the node of the stack.

Step 4: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.

Step 5: If the node does not have any unvisited child nodes, pop the node from the

stack.

**Experiment Exercise** :

1. Implement BFS and DFS using any programming language of your choice.

BFS

Code:

|  |
| --- |
| graph = {  '5' : ['3','7'],  '3' : ['2', '4'],  '7' : ['8'],  '2' : [],  '4' : ['8'],  '8' : []  }  visited = [] *# List for visited nodes.*  queue = [] *#Initialize a queue*  **def** bfs(visited, graph, node): *#function for BFS*  visited.append(node)  queue.append(node)  **while** queue: *# Creating loop to visit each node*  m = queue.pop(0)  **print** (m, end = " ")  **for** neighbour **in** graph[m]:  **if** neighbour **not** **in** visited:  visited.append(neighbour)  queue.append(neighbour)  *# Driver Code*  **print**("Following is the Breadth-First Search")  bfs(visited, graph, '5') *# function calling* |

Output:

|  |
| --- |
| Following is the Breadth-First Search  5 3 7 2 4 8 |

DFS

Code:

|  |
| --- |
| # Using a Python dictionary to act as an adjacency list  graph = {  '5' : ['3','7'],  '3' : ['2', '4'],  '7' : ['8'],  '2' : [],  '4' : ['8'],  '8' : []  }  visited = set() # Set to keep track of visited nodes of graph.  **def** **dfs**(visited, graph, node): #function for dfs  **if** node **not** **in** visited:  **print** (node)  visited.add(node)  **for** neighbour **in** graph[node]:  dfs(visited, graph, neighbour)  # Driver Code  **print**("Following is the Depth-First Search")  dfs(visited, graph, '5') |

Output:

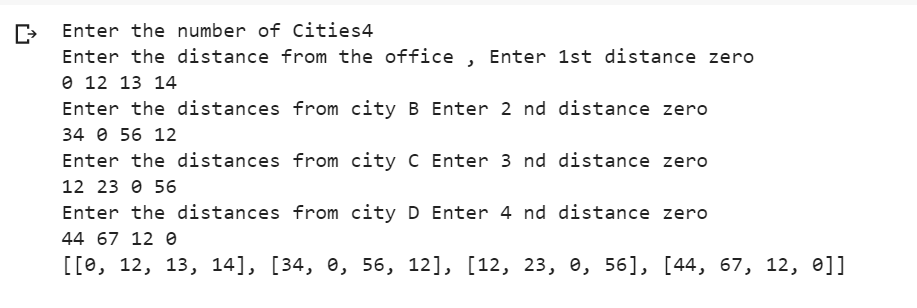
|  |
| --- |
| Following is the Depth-First Search  5 3 2 4 8 7 |

**Post Experiment Exercise:**

1. Identify a problem (e.g. Chess Knight Problem) that can be solved using BFS/DFS or both to resolve the problem. Implement it using any programming language.

|  |
| --- |
| #initialize list  edges , temp = ([],)\*2  n=int(input("Enter the number of Cities"))  for  j in range(n):    if j==0:      print("Enter the distance from the office , Enter 1st distance zero")      temp=input().split()      temp = [int(i) for i in temp]      edges.append( temp )    else:      print("Enter the distances from city" ,chr(65+j) , "Enter", j+1 , "nd distance zero" )      temp=input().split()      temp = [int(i) for i in temp]      edges.append( temp )  print(edges) |

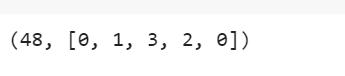
Output:



**DFS**

|  |
| --- |
| def TSP\_dfs(node, edges, visited, cost, path):      cnt = 0      path.append(node)      visited[node] = True      for i in range(4):          if visited[i] == False:              cnt += 1      if cnt == 0:          path.append(0)          return (cost + edges[node][0]), path      mini = 10000      A = []      for i in range(4):          if visited[i] == False:              tmp = [False]\*5              for j in range(4):                  tmp[j] = visited[j]              P = []              for j in range(len(path)):                  P.append(path[j])              t, l = TSP\_dfs(i, edges, tmp, cost + edges[node][i], P)              if mini > t:                  mini = t                  A = l      return mini, A  visited = [False]\*5  path = []  print(TSP\_dfs(0, edges, visited, 0, path)) |

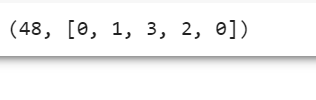
Output:



**BFS**

|  |
| --- |
| def TSP\_bfs(edges):      q = []      path = []      visited = [False]\*5      p = [0]      q.append((0, 0, visited, p))      while len(q) != 0:          cnt = 0          curr = q.pop(0)          curr[2][curr[0]] = True          for i in range(4):              if curr[2][i] == False:                  cnt += 1          if cnt == 0:              P = curr[3]              P.append(0)              path.append((curr[1] + edges[curr[0]][0], P))          for i in range(4):              if curr[2][i] == False:                  tmp = [False]\*5                  for j in range(4):                      tmp[j] = curr[2][j]                  P = []                  for j in range(len(curr[3])):  mini = 1000      P = []      for i in range(len(path)):          if mini > path[i][0]:              mini = path[i][0]              P = path[i][1]      return mini, P  print(TSP\_bfs(edges))                      P.append(curr[3][j])                  P.append(i)                  q.append((i, curr[1] + edges[curr[0]][i], tmp, P)) |

Output:



**Conclusion:** Problem Solving using BFS/DFS has been explored and implemented using Python.