# Worksheet 4 Graph traversal algorithms

**Task 1**

1. The algorithm for a depth-first traversal is given below, together with the definition of the graph that has just been traversed in the PowerPoint presentation, held in an adjacency list implemented as a dictionary data structure. The main program calls the subroutine , passing the first node and the empty list of visited nodes as parameters

GRAPH = { "A":["B","D","E"], "B":["A","C","D"], "C":["B","G"], "D":["A","B","E","F"], "E":["A","D"] , "F":["D"], "G":["C"]}

visitedList = [] #an empty list of visited nodes

SUB dfs(graph, currentVertex, visited)

 append currentVertex to list of visited nodes

 #check neighbours of currentVertex

 FOR vertex in graph[currentVertex]

 IF vertex NOT IN visited THEN

 dfs(graph, vertex, visited)

#system stack will automatically store return address, parameters and local variables

 ENDIF

 ENDFOR

 RETURN visited

ENDSUB

#main program

traversal = dfs(GRAPH, "A", visitedList)

OUTPUT "Nodes visited in this order: ", traversal

(a) Which vertices are referred to in the FOR statement the first time it is executed?

(b) The subroutine is recursive. What are the values of **vertex** and **visited** the first time the recursive call is made in the subroutine?

(c) Why is there no reference to pushing and popping items on and off the stack in this algorithm?

(d) What order are the nodes visited in using this traversal algorithm?

2. A recursive routine can be used to perform a depth-first search of the graph that represents a maze to test if there is a route from the entrance 1 to the exit 5.



 The routine is shown below. It has two parameters, v (the current vertex) and endV (the exit vertex)

 PROCEDURE DFS(v, endV)

 discovered[v] 🡨 True

 IF v = endV THEN found 🡨 True

 FOR each neighbour u of v

 IF discovered[u] = False THEN DFS(u, endV)

 ENDFOR

 completelyExplored[v] 🡨 True

 END PROCEDURE

 Complete the trace table below to show how the discovered and completelyExplored flag arrays and the variable found are updated by the algorithm when it is called using DFS.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **discovered** | **completelyExplored** |  |
|  | Call | V | U | endV | [1] | [2] | [3] | [4] | [5] | [1] | [2] | [3] | [4] | [5] | Found |
| 1 |  | - | - | 5 | F | F | F | F | F | F | F | F | F | F | F |
| 2 | DFS(1,5) | 1 | 2 | 5 | **T** | F | F | F | F | F | F | F | F | F | F |
| 3 | DFS(2,5) | 2 | 1 | 5 | T | **T** | F | F | F | F | F | F | F | F | F |
| 4 |  |  | 3 | 5 | T | T | F | F | F | F | F | F | F | F | F |
| 5 | DFS(3,5) | 3 | 2 | 5 | T | T | **T** | F | F | F | F | **T** | F | F | F |
| 6 | DFS(2,5) | 2 | 4 | 5 | T | T | T | F | F | F | F | T | F | F | F |
| 7 | DFS(4,5) | 4 | 2 | 5 |  |  |  |  |  |  |  |  |  |  |  |
| 8 | DFS(4,5) |  | 5 | 5 |  |  |  |  |  |  |  |  |  |  |  |
| 9 | DFS(5,5) | 5 | 4 | 5 |  |  |  |  |  |  |  |  |  |  |  |
| 10 | DFS(4,5) | 4 | - | 5 |  |  |  |  |  |  |  |  |  |  |  |
| 11 | DFS(2,5) | 2 | - | 5 |  |  |  |  |  |  |  |  |  |  |  |
| 12 | DFS(1,5) | 1 | - | 5 |  |  |  |  |  |  |  |  |  |  |  |

**Additional question**

3. Here is another graph.



Using the same algorithm as above, complete the trace table below to show how the discovered and completelyExplored flag arrays and the variable found are updated by the algorithm when it is called using DFS.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **discovered** | **completelyExplored** |  |
| Call | v | u | endV | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | found |
|  | - | - | 7 | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| DFS(1,7) | 1 | 2 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(2,7) | 2 | 1 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 3 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(3,7) | 3 | 2 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(2,7) | 2 | 4 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(4,7) | 4 | 2 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(2,7) | 2 | - | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(1,7) | 1 | 5 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(5,7) | 5 | 1 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 5 | 6 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(6,7) | 6 | 5 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(5,7) | 5 | 7 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(7,7) | 7 | 5 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(5,7) | 5 | - | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DFS(1,7) | 1 | - | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

 **Task 2**

4. The algorithm for the breadth-first search is given below, together with the adjacency list implemented as a dictionary data structure which defines the graph.

GRAPH = {

 "A": {"colour": "White", "neighbours": ["B", "D", "E"]},

 "B": {"colour": "White", "neighbours": ["A", "D", "C"]},

 "C": {"colour": "White", "neighbours": ["B", "G"]},

 "D": {"colour": "White", "neighbours": ["A", "B", "E", "F"]},

 "E": {"colour": "White", "neighbours": ["A", "D"]},

 "F": {"colour": "White", "neighbours": ["D"]},

 "G": {"colour": "White", "neighbours": ["C"]}

 }

SUB bfs(graph, vertex)

 queue 🡨 [] #an empty queue

 visited 🡨 [] #an empty list of visited nodes

 enqueue vertex

 WHILE queue NOT empty

 dequeue item and put in currentNode

 set colour of currentNode to "Black"

 append currentNode to visited

 FOR each neighbour of currentNode

 IF colour of neighbour = "White" THEN

 enqueue neighbour

 set colour of neighbour to "Grey"

 ENDIF

 ENDFOR

 ENDWHILE

 RETURN visited

ENDSUB

#main

visited 🡨 bfs(GRAPH, "A")

OUTPUT "List of nodes visited: ", visited

(a) Draw the graph. In what order are the vertices visited in this traversal?

(b) Is this algorithm iterative or recursive?

(c) What is the state of the queue before the WHILE loop is entered for the first time?

(d) What does the colour of a particular node signify?