Data Structures

Lab Report 01

Ganesh Nathan
2021BCS0104

Con tents

1 Problem Statemen t

Lab-1: Detecting cycles in graphs

(1) Implement the algorithm for detecting cycles in an undirected graphs using DFS algorithm
(2) Extend the implementation for directed graphs
(3) Analyze the complexity of the algorithm (with formal justification) Note use Latex for writing the theory and algorithms.

2 Algorithm

The Algorithm to detect cycles in directed or undirected graphs is as follows :

1. We pick a source node and start DFS
2. Whenever a node is popped from the DFS stack:
    (a) We traverse all the adjacent nodes of the popped node and check if any of the adjacent nodes which are not already visited, if the node is in the stack , then the loop is exited and return true or else return false and the node is added to the stack

The Algorithm to detect cycles in directed or directed graphs is as follows :

1. We pick a source node and start DFS
2. Whenever a node is popped from the DFS stack:
    (a) We traverse all the adjacent nodes of the popped node and check if any of the adjacent nodes are already visited , if it is visited , then the loop is exited and return true else return false and the node is added to the stack

3 Explaination

The code uses a depth-first search traversal of the graph to check for cycles. It starts by initializing an empty stack with the starting node (in your case, node 1). It also initializes an empty visited list to keep track of the nodes visited during the traversal.

Then, while the stack is not empty, the code pops a node from the stack, marks it as visited, and checks its neighbors. For each neighbor that has not been visited yet, the code adds it to the stack. If a neighbor has already been visited and is not the parent of the current node (for an undirected graph) or is already in the stack (for a directed graph), the code prints a message indicating that a cycle has been detected and exits the loop.

4 Complexit y Analysis

Time Complexity: The time complexity of the algorithm is O(V+E), where V is the number of vertices/nodes and E is the number of edges in the graph. This is because the algorithm visits each node in the graph at most once, and for each node, it visits all its adjacent edges. Since each edge is visited exactly twice (once for each endpoint), the total number of edge traversals is 2E. Therefore, the total number of operations performed by the algorithm is V+2E, which is O(V+E).

Space Complexity: The space complexity of the algorithm is O(V+E), because it uses two data structures to keep track of the visited nodes and the nodes in the stack. The visited nodes are stored in a list that can hold up to V nodes. The stack can hold up to V nodes at any point in time, but since the maximum depth of the stack is the number of nodes in the longest path in the graph, it can also be upper-bounded by E. Therefore, the total space used by the algorithm is O(V+E).

Overall, this algorithm is a simple and efiicient way to check for cycles in an undirected or directed graph using depth-first search. The time and space complexity are both linear in the size of the graph, making it a practical solution for large graphs.

5 The Co de

Listing 1: Detecting Cycle in a Graph

```
def              _                    , undirected ):
    detect cycle ( dit stack = [ ]
   # sample data
      # adj _ l i st = {1 :         [2] ,
    #                       2 :    [1 ,    3 ,4] ,
    #                        3:[2 ,4] ,
    #                       4 :     [2 ,3 ,5] ,
    #   5 : [4] } adj _list = dit
    stack . append(1) visited = [ ]

    while stack :
        i = stack .pop() i f i not in
        visited :
            visited . append( i )
            for j in adj _list [ i ] : i f j in visited
                :
                    print (” cycle ␣ detected” )
                    break i f j not in
                   stack :
                    stack . append( j )
                else : print (” cycle ␣ detected” ) break


def main ():
    num = int (input (” Enter the no of nodes : ” ))   ␣        ␣
    lines = [ input () for in range(num)] dit = {} for i in lines :
        line = l i s t (map(int , i . split ())) dit [ line [ 0] ] = line
        [ 1: ]
    undirected = 1 i f input (” Press 1 if the graph is undirected” ) else 0 print ( dit ) detect _cycle ( dit ,
    undirected )
```

```
if __name__ == '__main__':
    main()
```