Heaven's Light is Our Guide

RAJSHAHI UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Dept. of Computer Science & Engineering

Course Code : *CSE 2202*
Course Title : *Sessional based on CSE 2201*
Experiment NO : *04*
Experiment Name : *Implementation and Complexity analysis of Depth-first search, Breadth-first search and Topological Sort.*

Date of Experiment : *31/10/2022*
Date of Submission : *07/11/2022*

MACHINE CONFIGURATION:
Intel(R) Core(TM) i5-10210U CPU
@ 1.60GHz 2.11 GHz, 20.0 GB of RAM 1 TB of Hard disk

Submitted By

Name: Md. Golam All Gaffar Tasin
Roll : 1903114
Section: B
Year: 2nd Year(Even)
Dept. of CSE

Submitted to

Rizoan Toufiq
Assistant Professor, Dept. of CSE
Rajshahi University Of Engineering And Technology.

Name of the Experiment: Implementation and Complexity analysis of Depth-first search, Breadth-first search and Topological Sort.

Objective: Getting knowledge about Depth-first search, Breadth-first search and Topological Sort. Also getting knowledge about complexity of these algorithms.

Theory:

Depth-first search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored.

In each call DFS-VISIT(G,u), vertex u is initially white. Line 1 increments the global variable time, line 2 records the new value of time as the discovery time u:d, and line 3 paints u gray. Lines 4–7 examine each vertex adjacent to u and recursively visit if it is white. As each vertex v € Adj[u] is considered in line 4, we say that edge (u,v) is explored by the depth-first search. Finally, after every edge leaving u has been explored, lines 8–10 paint u black, increment time, and record the finishing time in u.f . Note that the results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices and upon the order in which line 4 of DFS-VISIT visits the neighbors of a vertex. The procedure DFS-VISIT is called exactly once for each vertex v € V, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G,u), the loop on lines 4–7 executes | Adj[v] | times. Since

$$\sum | Adj[v] | = \theta(E)$$

the total cost of executing lines 4–7 of DFS-VISIT is θ(E). The running time of DFS is therefore θ(V+E).

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.

The operations of enqueuing and dequeuing take O(1) time, and so the total time devoted to queue operations is O(V). Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is θ(E), the total time spent in scanning adjacency lists is O(E). The overhead for initialization is O(V), and thus the total running time of the BFS procedure is O(V + E). Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

A topological sort is a graph traversal in which each node v is only visited after all of its dependencies have been visited. If the graph contains no directed cycles, then it is a directed acyclic graph. Any DAG has at least one topological ordering, and there exist techniques for building topological orderings in linear time for any DAG.

We can perform a topological sort in time θ(V + E), since depth-first search takes θ(V + E) time and it takes O(1) time to insert each of the |V | vertices onto the front of the linked list.

Algorithm:

DFS(G)

1. for each vertex u € G.V

2. u.color = WHITE

3. u.∏ = NIL

4. time = 0

5. for each vertex u € G.V

6. if u.color == WHITE

7. DFS-VISIT (G,u)

DFS-VISIT (G,u)

1. time = time + 1 // white vertex u has just been discovered

2. u.d = time

3. u.color = GRAY

4. for each vertex u € G.Adj[u] // explore edge (u,v)

5. if v.color == WHITE

6. v.∏ = u

7. DFS-VISIT(G,v)

8. u.color = BLACK // blacken u; it is finished

9. time = time + 1

10. u.f = time

BFS(G,s)

1.for each vertex u € G.V – {s}

```
2       u.color = WHITE
3       u.d = ∞
4.      u.∏ = NIL
5. s.color = GRAY
6. s.d = 0
7. s.∏ = NIL
8. Q = ø
9. ENQUEUE(Q, s)
10. while Q ≠ ø
11      u = DEQUEUE(Q)
12.     for each v € G:Adj[u]
13.       if v.color == WHITE
14.             v.color = GRAY
15.             v.d = u.d + 1
16.             v.∏ = u
17.              ENQUEUE(Q,v)
18        u.color = BLACK
```

TOPOLOGICAL-SORT(G)

1. call DFS(G) to compute finishing times v.f for each vertex v

2. As each vertex is finished, insert it onto the front of a linked list

3. return the linked list of vertices

Code:

DFS:

```
1. #include<iostream>
2. #include<list>
3. using namespace std;
4.
5. class graph{
6.     int numVertices;
7.     list<int>* adjLists;
8.     bool* visited;
9. public:
10.     graph(int vertices);
11.     void addEdge(int src,int destination);
12.     void DFS(int vertex);
13. };
14.
15. graph::graph(int vertices){
16.     numVertices = vertices;
17.     adjLists = new list<int>[vertices];
18.     visited = new bool[vertices];
19. }
20.
21. void graph::addEdge(int src , int destination){
22.     adjLists[src].push_front(destination);
23.     adjLists[destination].push_back(src);
24. }
25.
26. void graph::DFS(int vertex){
27.     visited[vertex] = true;
28.     list<int> adjList = adjLists[vertex];
29.     cout<<vertex<<" ";
30.
31.     list<int>::iterator i ;
32.
33.     for(i=adjList.begin() ; i!= adjList.end();i++){
34.         if(!visited[*i]){
35.             DFS(*i);
36.         }
37.     }
38. }
39.
40. int main()
41. {
```

```
    graph g(6);
    g.addEdge(1,2);
    g.addEdge(1,5);
    g.addEdge(2,3);
    g.addEdge(2,4);
    g.addEdge(2,5);
    g.addEdge(3,4);
    g.addEdge(4,5);
    g.DFS(2);
}
```

BFS:

```
#include<iostream>
#include<list>
using namespace std;

class graph{
    int numVertices;
    list<int>* adjLists;
    bool* visited;
public:
    graph(int vertices);
    void addEdge(int src,int destination);
    void BFS(int startVertex);
};

graph::graph(int vertices){
    numVertices = vertices;
    adjLists = new list<int>[vertices];
}

void graph::addEdge(int src, int destination){
    adjLists[src].push_back(destination);
    adjLists[destination].push_back(src);
}

void graph::BFS(int startVertex){
    visited = new bool[numVertices];
    for(int i=0;i<numVertices;i++){
        visited[i] = false;
    }
    list<int> queue;

    visited[startVertex] = true;
```

```
33.      queue.push_back(startVertex);
34.
35.      list<int>::iterator i;
36.
37.      while(!queue.empty()){
38.          int currVertex = queue.front();
39.          cout<<"Visited "<<currVertex<<" ";
40.          queue.pop_front();
41.
42.          for(i=adjLists[currVertex].begin();
   i!=adjLists[currVertex].end();i++){
43.              int adjVertex = *i;
44.              if(!visited[adjVertex]){
45.                  visited[adjVertex] = true;
46.                  queue.push_back(adjVertex);
47.              }
48.          }
49.          cout<<endl;
50.      }
51.
52.  }
53.
54.  int main()
55.  {
56.      graph g(6);
57.
58.      g.addEdge(1, 2);
59.      g.addEdge(1, 5);
60.      g.addEdge(2, 5);
61.      g.addEdge(2, 4);
62.      g.addEdge(2,3);
63.      g.addEdge(3, 4);
64.      g.addEdge(4, 5);
65.
66.
67.      g.BFS(2);
68.      return 0;
69.  }
70.
71.
```

Topological sort:

```
#include<iostream>
#include<list>
using namespace std;

int v=1;

class graph{
    int numVertices;
    list<int>* adjLists;
     bool* visited;
     int time;
public:
     graph(int vertices);
     void addEdge(int src,int destination);
     void DFS(int vertex);
     void setTime(int t);
     int showTime();
};

graph::graph(int vertices){
     numVertices = vertices;
     adjLists = new list<int>[vertices];
     visited = new bool[vertices];
}

void graph::addEdge(int src , int destination){
     adjLists[src].push_front(destination);
     adjLists[destination].push_back(src);
}

void graph::setTime(int t){
     time = t;
}

int graph::showTime(){
     cout<<time<<endl;
     return time;
}

void graph::DFS(int vertex){
     visited[vertex] = true;
     setTime(v);
     v++;
```

```
44.      showTime();
45.      list<int> adjList = adjLists[vertex];
46.      cout<<vertex<<" ";
47.
48.      list<int>::iterator i ;
49.
50.      for(i=adjList.begin() ; i!= adjList.end();i++){
51.          if(!visited[*i]){
52.              DFS(*i);
53.          }
54.      }
55.  }
56.
57.
58.  int main()
59.  {
60.     graph g(3);
61.     g.addEdge(2,0);
62.     g.addEdge(2,1);
63.     g.DFS(2);
64.  }
```

Discussion:

BFS uses Queue data structure for finding the shortest path. It builds the tree level by level. It is more suitable for searching vertices closer to the given source.The Time complexity of BFS is O(V + E) when Adjacency List is used and $O(V^2)$ when Adjacency Matrix is used, where V stands for vertices and E stands for edges.BFS is slow as compared to DFS. DFS uses Stack data structure. It builds the tree sub-tree by sub-tree. It is more suitable when there are solutions away from source.DFS algorithm is a recursive algorithm that uses the idea of backtracking. DFS is not optimal for finding the shortest path compared to BFS. If a topological sort has the property that all pairs of consecutive vertices in the sorted order are connected by edges, then these edges form a directed Hamiltonian path in the DAG. If a Hamiltonian path exists, the topological sort order is unique; no other order respects the edges of the path. Conversely, if a topological sort does not form a Hamiltonian path, the DAG will have two or more valid topological orderings, for in this case it is always possible to form a second valid ordering by swapping two consecutive vertices that are not connected by an edge to each other.