Lab Assignment #9
Graph DFS

Full Name: Shreyas Srinivasa
BlazerId: SSRINIVA

CS 303 Algorithms and Data Structures

Spring 2023

1. Problem Specification

The purpose of this project was to implement the depth first search algorithm and test it with the graph datasets provided and consecutively use the same algorithm for performing a topological sort on the vertices. The programmatic representation used is the adjacency list representation, same as the previous lab. The DFS algorithm was tested by printing the contents and paths between vertices of the graph formed from the mediumG.txt dataset. The topological sort is performed with the same algorithm, so the vertices are inserted into a list in the desired order during DFS. Since the topological sort can be performed only on directed graphs, tingDG.txt is used to obtain the final sorted output.

2. Program Design

```
public Node get(int i) {
    return vertices[i];
}

public void insert(int v1, int v2, boolean directed) {

    if (vertices[v1] == null)
        vertices[v1] = new Node((long) v1);
    if (vertices[v2] == null)
        vertices[v2] = new Node((long) v2);

    graph.get(v1).add(vertices[v2]);
    if(!directed) graph.get(v2).add(vertices[v1]);

}
```

The functions for retrieving a single vertex from the list of vertices and inserting a new vertex by specifying its neighbor. This function has been modified to work with directed graphs.

```java
public void print() {
    for (int i = 0; i < graph.size(); i++) {
        System.out.format("%d -> ", i);
        for (Node v : graph.get(i)) {
            System.out.format("%d -> ", v.getValue());
        }
        System.out.print("/ \n");
    }
}

public void dfs() {
    for (Node v : vertices) {
        v.setColor("WHITE");
        v.setP(null);
    }
    time = 0l;
    for(Node v: vertices) {
        if(v.getColor().equals("WHITE")) {
            dfsVisit(v);
        }
    }
}
```

The implementation of the print function to display the contents of the graph, which is the same as used in the BFS experiment. The dfs() function is part of the implementation of the DFS algorithm, and serves as a starting point for the algorithm.

```
public void dfsVisit(Node u) {
    time += 1;
    u.setD(time);
    u.setColor("GRAY");
    for(Node v: graph.get(Math.toIntExact(u.getValue()))) {
        if(v.getColor().equals("WHITE")) {
            v.setP(u);
            dfsVisit(v);
        }
    }
    u.setColor("BLACK");
    time += 1;
    u.setF(time);
    sortedVertices.add(0, u);
}

public void printPath(Node s, Node v) {
    if (v.equals(s)) {
        System.out.print("\n" + s.getValue());
    } else if (v.getP() == null) {
        System.out.format("\n%d X %d", s.getValue(), v.getValue());
    } else {
        printPath(s, v.getP());
        System.out.format(" , %d", v.getValue());
    }
}
```

The dfsVisit() function which actually does all the work of the algorithm. It recursively digs deep into the tree and follows every path till its end until all the vertices have been traversed. The print path function uses the parent pointer stored to print paths between a chosen source vertex and every other vertex.

```
public void print() {
    for (int i = 0; i < graph.size(); i++) {
        System.out.format("%d -> ", i);
        for (Node v : graph.get(i)) {
            System.out.format("%d -> ", v.getValue());
        }
        System.out.print("/ \n");
    }
}
```

The function to print the entire adjacency list that holds the graph information

```java
public static Graph getGraphFromFile(String filename) {
    Graph g = null;
    try (Scanner fileScanner = new Scanner(Paths.get(filename))) {
        Integer vertices = fileScanner.nextInt();
        g = new Graph(vertices);
        Integer edges = fileScanner.nextInt();
        fileScanner.nextLine();
        while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            String elements[] = line.split(" ");
            g.insert(Integer.valueOf(elements[0]), Integer.valueOf(elements[1]));
        }
        fileScanner.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
        System.exit(1);
    }
    return g;
}
```

The function which reads all the graph data from the passed filename and return a Graph object

```java
public static void driver1(String filename) {
    Instant start, finish;
    long timeElapsed;

    Graph g = getGraphFromFile(filename, false);
    g.print();


    start = Instant.now();
    g.dfs();
    finish = Instant.now();
    timeElapsed = Duration.between(start, finish).toNanos();

    System.out.format("Time taken for BFS:= %d\n", timeElapsed);

    for(Node v: g.getAllVertices()) {
        System.out.format("\nPath from Source Vertex %d to Vertex %d", g.get(0).getValue(), v.getValue());
        g.printPath(g.get(0), v);
    }

    for(Node v: g.getAllVertices()) {
        System.out.println(v);
    }
}
```

The function to perform the DFS and print all the paths between the source vertex 0 and any other vertex.

```java
public static void driver2(String filename) {
    Instant start, finish;
    long timeElapsed;

    Graph g = getGraphFromFile(filename, true);
    g.print();


    start = Instant.now();
    g.dfs();
    finish = Instant.now();
    timeElapsed = Duration.between(start, finish).toNanos();

    System.out.format("Time taken for BFS:= %d\n", timeElapsed);

    for(Node v: g.getAllVertices()) {
        g.printPath(g.get(0), v);
    }

    System.out.println("Topological Sort");
    for(Node v: g.getSortedVertices()) {
        System.out.print(v.getValue() + ", ");
    }

    for(Node v: g.getAllVertices()) {
        System.out.println(v);
    }
}
```

Similar driver function to print the contents of the topologically sorted array

3. **Output**

```
0 -> 5 -> 1 -> 2 -> 6 -> /
1 -> 0 -> /
2 -> 0 -> /
3 -> 4 -> 5 -> /
4 -> 3 -> 6 -> 5 -> /
5 -> 0 -> 4 -> 3 -> /
6 -> 4 -> 0 -> /
7 -> 8 -> /
8 -> 7 -> /
9 -> 12 -> 10 -> 11 -> /
10 -> 9 -> /
11 -> 12 -> 9 -> /
12 -> 9 -> 11 -> /
Time taken for DFS:= 108629

Path from Source Vertex 0 to Vertex 0
0
Path from Source Vertex 0 to Vertex 1
0 , 1
Path from Source Vertex 0 to Vertex 2
0 , 2
Path from Source Vertex 0 to Vertex 3
0 , 5 , 4 , 3
Path from Source Vertex 0 to Vertex 4
0 , 5 , 4
Path from Source Vertex 0 to Vertex 5
0 , 5
Path from Source Vertex 0 to Vertex 6
0 , 5 , 4 , 6
Path from Source Vertex 0 to Vertex 7
 no path from 0 to 7 exists
Path from Source Vertex 0 to Vertex 8
 no path from 0 to 7 exists , 8
Path from Source Vertex 0 to Vertex 9
 no path from 0 to 9 exists
```

Output 1 The adjacency list graph representation printed out on the console and the output of the printPaths function

```
0 , 1
Path from Source Vertex 0 to Vertex 2
0 , 2
Path from Source Vertex 0 to Vertex 3
0 , 5 , 4 , 3
Path from Source Vertex 0 to Vertex 4
0 , 5 , 4
Path from Source Vertex 0 to Vertex 5
0 , 5
Path from Source Vertex 0 to Vertex 6
0 , 5 , 4 , 6
Path from Source Vertex 0 to Vertex 7
 no path from 0 to 7 exists
Path from Source Vertex 0 to Vertex 8
 no path from 0 to 7 exists , 8
Path from Source Vertex 0 to Vertex 9
 no path from 0 to 9 exists
Path from Source Vertex 0 to Vertex 10
 no path from 0 to 9 exists , 10
Path from Source Vertex 0 to Vertex 11
 no path from 0 to 9 exists , 12 , 11
Path from Source Vertex 0 to Vertex 12
 no path from 0 to 9 exists , 12

0 -> 4 -> 2 -> /
1 -> 3 -> /
2 -> 7 -> /
3 -> 6 -> /
4 -> 5 -> 7 -> /
5 -> 4 -> 7 -> 1 -> /
6 -> 2 -> 0 -> 4 -> /
7 -> 5 -> 3 -> /
Time taken for DFS:= 38201
Topological Sort
0, 4, 5, 1, 7, 3, 6, 2,
```

Output 2. The output of the driver2 function, which displays the results of the sort

4. Analysis and Conclusions

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.

So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

Time Complexity: O(V+E)
Where V is the number of vertices and E is the number of edges in the graph.

Space Complexity: O(V+E)
Where V is the collected size invested in tracking visited vertices and E is the stack size used up in recursion.

The extra dfs() function we are using that serves an an entry point for the algorithm contains the code for handling disconnected graphs.

Advantages of Depth First Search:

- A linear relationship between the memory needed and the search graph exists. Contrast this with breadth-first search, which takes up more room. The method only needs to store a stack of nodes on the route from the root to the current node, which explains why.
- Given that a depth-first search produces the same set of nodes as a breadth-first search, but in a different order, the time complexity of a depth-first search to depth d and branching factor b (the number of children at each node, the outdegree) is O(bd). Hence, depth-first search is actually time-limited as opposed to space-limited.
- It is possible that DFS may find a solution without much exploration in very less time.

Disadvantages of Breadth First Search:

- The disadvantage of Depth-First Search is that there is a possibility that it may down the left-most path forever. Using a cutoff depth d leads to undesirable consequences if d is not selected accurately enough. Deviation from the ideal value for the problem may lead to failure in detecting a solution or additional cost in the form of execution time.
- DFS does not guarantee a solution
- DFS does not guarantee the best available solution

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Topological Sorting vs Depth First Traversal (DFS):
In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices.

For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting.

Time Complexity: O(V+E). The above algorithm is simply DFS with an extra list or stack. So time complexity is the same as DFS.

Space Complexity: O(V+E). Extra space is needed for the list or stack containing the sorted vertices.

A non-recursive approach by using a working stack is also possible.

Applications of Depth First Search

- Detecting cycle in a graph
- Path finding
- Topological Sorting
- Bipartite testing

- Solving puzzles with only one solution
- Web Crawlers
- Maze Generation
- Model Checking
- Backtracking

Applications of Topological Sort

- The major application of topological sorting is to schedule tasks based on the known dependencies between tasks.
- Instruction scheduling
- Ordering of formula cell evaluation when recomputing formula values in spreadsheets
- Logic synthesis
- Determining the order of compilation tasks to perform in make files
- Data serialization
- Resolving symbol dependencies in linkers

Observations based on the lab experiment

- We tested our DFS algorithm using the tinyG.txt dataset. This graph turned out to be a disconnected graph, as some paths between the chosen source vertex and other vertices did not exist. The traversal of disconnected graphs is handled by adding an extra loop in an entry point function that iterates over all the vertices of the graph and moving the core logic to a util function which calls itself recursively.
- For implementing the topological sort, we did not implement a different function. We simply added logic to insert the vertices into a list once exploration of its subtree is completed. We push the vertices only after its entire subtree is already in the list. We store this list as a property of the graph object which is easily accessible by using a getter method after triggering the DFS. The insertion method was modified to support the insertion of directed graph edges.

5. References

- https://brilliant.org/wiki/depth-first-search-dfs/
- https://cp-algorithms.com/graph/depth-first-search.html
- https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
- https://www.geeksforgeeks.org/topological-sorting/
- https://www.geeksforgeeks.org/applications-of-depth-first-search/