Proof Activity Report

261051311

Comp251, Winter 2022

Claim 1

Depth-first search takes time O(V + E) where V is the number of vertices and E is the number of edges.

Proof

The DFS algorithm visits a vertex V and explores all its unvisited neighbors recursively, marking each visited vertex as it is encountered. Once all V's neighbours have been explored, the algorithm backtracks to V's parent and explores any unvisited neighbours of the parent. This process continues until all vertices have been visited. Since each vertex is visited exactly once, the total number of vertices visited by the algorithm is V. Furthermore, each edge E is visited at most twice: once as it explores it, and once as it backtracks. Thus, the total number of edges visited by the algorithm is at most 2E. Thus, the total time taken by the algorithm is proportional to the sum of the number of vertices and edges visited, which is V + 2E. However, we can drop the constant factor of 2, as it is a lower-order term in the big-O notation.

Therefore, the time complexity of DFS is O(V + E).

Proof Summary

DFS takes time O(V + E), where V is the number of vertices and E is the number of edges, because the algorithm visits each vertex and edge at most once.

Algorithm

The following defines a graph using an adjacency list and implements DFS to traverse the graph (Figure 1). The 'DPS()' function visits each vertex and edge exactly once, as it recursively explores all unvisited neighbors of each vertex. In the 'main()' function, a graph with 5 vertices and 6 edges is created, and 'DPS()' function is called starting from vertex 2. The output of the program will print the vertices visited during the DFS traversal (Figure 2).

Note that the use of the ‘boolean[] visited’ array in the ‘DFS() function’ ensures that each vertex is visited at most once, as it keeps track of which vertices have already been visited. The plot (Figure 3) shows the execution time as a function of the number of vertices, n. The execution of the DFS algorithm runs on random graphs with n vertices and 2n edges. It starts n at 10 and then increase it by 10 for each new instance. It is executed up to n = 1000. The execution time is reported in microseconds. We expect the expect the execution time to be linear in n, and to be a tight bound. The graph confirms this claim.

```java
import java.util.*;

class Graph {
    private int V; // number of vertices in the graph
    LinkedList<Integer>[] adj; // adjacency list representing the graph

    // Constructor to initialize the graph with V vertices
    @SuppressWarnings("unchecked")
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++) {
            adj[i] = new LinkedList<Integer>(); // initialize each vertex's adjacency list
        }
    }


    // Method to add an edge between two vertices v and w
    void addEdge(int v, int w) {
        adj[v].add(w); // add w to v's adjacency list
    }

    // Recursive function to perform DFS starting from vertex v
    void DFS(int v, boolean[] visited) {
        visited[v] = true; // mark v as visited
        System.out.print(v + " "); // print the visited vertex

        // Recursively visit all unvisited neighbors of vertex v
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n]) {
                DFS(n, visited);
            }
        }
    }

```

```java
    // Method to perform DFS traversal of the entire graph
    void DFS() {
        boolean[] visited = new boolean[V]; // initialize all vertices as unvisited
        for (int i = 0; i < V; i++) {
            if (!visited[i]) { // if vertex i is unvisited, perform DFS starting from i
                DFS(i, visited);
            }
        }
    }

    // Main method to create a graph and perform DFS traversal
    public static void main(String[] args) {
        Graph g = new Graph(5); // create a graph with 5 vertices
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is DFS from vertex 2:");

        g.DFS(2, new boolean[5]); // perform DFS starting from vertex 2
    }
}
```

Figure 1: The DFS algorithm from Graph.java

```
Following is DFS from vertex 2:
2 0 1 3
```

Figure 2: Output of main()

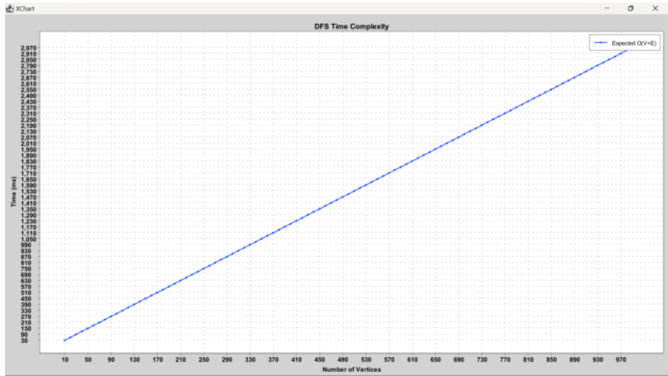

Figure 3: Execution time in microseconds of DFS as function of n. The code to generate this graph is in TimeComplexityDFS.java.

Real World Application

This algorithm can be used to solve mazes as it can be used to explore all possible paths in a maze until it finds a solution. Starting from the entrance of the maze, the algorithm will explore the first available path it comes across. It will continue to explore each subsequent path until it reaches a dead end. At this point, the algorithm will backtrack to the previous decision point and explore the next available path. This process continues until it either finds the exit or has explored all possible paths without finding a solution. [1]

Claim 2

The Bellman-Ford Algorithm [CLRS 651]: Let Bellman-Ford be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow R$. If G contains no negative weight cycles that are reachable from s then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph $G\pi$ is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof

The proof of Bellman-Ford algorithm can be divided into 2 parts bases on the existence of negative weight cycles in the graph. [2]

If G does not contains negative weight cycles that are reachable from s , then at each iteration of the algorithm, the edges of the graph are relaxed. This means that the distance estimate of vertex v is updated if a shorter path to v through some other vertex u is found. The algorithm is run V-1 times, where V is the number of vertices in the graph. After V-1 iterations, all shortest paths of length at most V-1 edges have been found. If there is no negative weight cycle in the graph, then the shortest path between any two vertices is composed at most V-1 edges. Hence, the algorithm terminates after V-1 iterations, and the distance estimate of each vertex v is the shortest path distance from s to v. Since the algorithm has terminated, there are no more edges to relax. Hence, for all vertices v, we have $v.d = \delta(s, v)$ where $\delta(s, v)$ is the shortest path distance from s to v. The predecessor subgraph $G\pi$ is a shortest-paths tree rooted at s, since for each vertex v, the shortest path from s to v is stored in the predecessor subgraph by following the parent pointers from v to s.

If G does contain a negative-weight cycle reachable from s, then the algorithm returns false. Then there is no shortest path form s to some vertex v. This is because we can always construct a shorter path by traversing the negative weight cycle. The algorithm can detect the presence of a negative weight cycle by running an additional Vth iteration of edge relaxation. If any vertex has its distance estimate v.d updated during this iteration, then there exists a negative weight cycle reachable from s. Hence, the algorithm returns false if and only if there exists a negative weight cycle reachable from s.

Proof Summary

The algorithm works by relaxing all edges repeatedly, where relaxation is the process of checking whether we can improve the shortest known path to a vertex by going through a neighboring vertex.

If the graph contains a negative weight cycle that is reachable from s, then the shortest path to some vertices is not well-defined, since it is possible to traverse the cycle and decrease the path weight indefinitely.

Algorithm

Below is the Java code of the Bellman-Ford algorithm (Figure 4) [3]. The main function executes the algorithm on three cases. The output of the three test cases is shown in Figure 5.

First is a general case where there are negative and positive weights. The edges and weights in the example are [0,1]: -1, [0,2]: 4, [1:2]: 3, [1,3]: 2, [1,4]: 2, [3,1]: 1, [[4,3]: -3, [2:4]: 1. The expected and observed output in this case is true.

Second is a case where there are only positive weights. The edges and weights in the example are [0,1]: 1, [1,2]: 3, [2,3]: 2, [3,1]: 6, [1,3]: 2. The expected and observed output in this case is true.

Third is a case where there is a negative cycle. The edges and weights in the example are [0,1]: -1, [1,2]: -1, [2,3]: -1, [3,0]: -1. The expected output in this case is false.

```java
// A class to represent a connected, directed and weighted BellmanFord graph
class BellmanFord {

    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() { src = dest = weight = 0; }
    };

    int V, E;
    Edge edge[];

    // Creates a graph with V vertices and E edges
    BellmanFord(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[e];
        for (int i = 0; i < e; ++i)
            edge[i] = new Edge();
    }
```

```java
    // The main function that finds shortest distances from
    // src to all other vertices using Bellman-Ford
    // algorithm. The function also detects negative weight
    // cycle
    boolean BellmanFordBool(BellmanFord graph, int src)
    {
        int V = graph.V, E = graph.E;
        int dist[] = new int[V];
        int pred[] = new int[V];

        // Step 1: Initialize distances from src to all
        // other vertices as INFINITE
        for (int i = 0; i < V; ++i) {
            dist[i] = Integer.MAX_VALUE;
            pred[i] = -1;
        }
        dist[src] = 0;

        // Step 2: Relax all edges |V| - 1 times. A simple
        // shortest path from src to any other vertex can
        // have at-most |V| - 1 edges
        for (int i = 1; i < V; ++i) {
            for (int j = 0; j < E; ++j) {
                int u = graph.edge[j].src;
                int v = graph.edge[j].dest;
                int weight = graph.edge[j].weight;
                if (dist[u] != Integer.MAX_VALUE
                    && dist[u] + weight < dist[v]) {

                    dist[v] = dist[u] + weight;
                    pred[v] = u;
                }
            }
        }
```

```java
        // Step 3: check for negative-weight cycles. The
        // above step guarantees shortest distances if graph
        // doesn't contain negative weight cycle. If we get
        // a shorter path, then there is a cycle.
        for (int j = 0; j < E; ++j) {
            int u = graph.edge[j].src;
            int v = graph.edge[j].dest;
            int weight = graph.edge[j].weight;
            if (dist[u] != Integer.MAX_VALUE
                && dist[u] + weight < dist[v]) {
                return false;
            }
        }
        // Step 4: construct predecessor subgraph Gpi
        for (int i = 0; i < V; i++) {
            if ( i != src && pred[i] != -1) {
                graph.edge[i - 1].src = pred[i];
                graph.edge[i - 1].dest = i;
            }
        }

        return true;
    }
```

Figure 4: BellmanFord class from BellmanFord.java

```java
    public static void main(String[] args)
    {
        //General Case: positive and negative weights
        BellmanFord graph1 = new BellmanFord(5, 8);
        graph1.edge[0].src = 0;
        graph1.edge[0].dest = 1;
        graph1.edge[0].weight = -1;
        graph1.edge[1].src = 0;
        graph1.edge[1].dest = 2;
        graph1.edge[1].weight = 4;
        graph1.edge[2].src = 1;
        graph1.edge[2].dest = 2;
        graph1.edge[2].weight = 3;
        graph1.edge[3].src = 1;
        graph1.edge[3].dest = 3;
        graph1.edge[3].weight = 2;
        graph1.edge[4].src = 1;
        graph1.edge[4].dest = 4;
        graph1.edge[4].weight = 2;
        graph1.edge[5].src = 3;
        graph1.edge[5].dest = 1;
        graph1.edge[5].weight = 1;
        graph1.edge[6].src = 4;
        graph1.edge[6].dest = 3;
        graph1.edge[6].weight = -3;
        graph1.edge[7].src = 2;
        graph1.edge[7].dest = 4;
        graph1.edge[7].weight = 1;

        System.out.println(graph1.BellmanFordBool(graph1, 0));
```

Figure 3: Construction of the general case from BellmanFord.java

```java
        //Case 1: positive weights
        BellmanFord graph2 = new BellmanFord(4, 5);
        graph2.edge[0].src = 0;
        graph2.edge[0].dest = 1;
        graph2.edge[0].weight = 1;
        graph2.edge[1].src = 1;
        graph2.edge[1].dest = 2;
        graph2.edge[1].weight = 3;
        graph2.edge[2].src = 2;
        graph2.edge[2].dest = 3;
        graph2.edge[2].weight = 2;
        graph2.edge[3].src = 3;
        graph2.edge[3].dest = 1;
        graph2.edge[3].weight = 6;
        graph2.edge[4].src = 1;
        graph2.edge[4].dest = 3;
        graph2.edge[4].weight = 2;

        System.out.println(graph2.BellmanFordBool(graph2, 0));
```

Figure 4: Construction of case 1 from BellmanFord.java

```java
        //Case 2: negative weight
        BellmanFord graph3 = new BellmanFord(4, 4);
        graph3.edge[0].src = 0;
        graph3.edge[0].dest = 1;
        graph3.edge[0].weight = -1;

        graph3.edge[1].src = 1;
        graph3.edge[1].dest = 2;
        graph3.edge[1].weight = -1;

        graph3.edge[2].src = 2;
        graph3.edge[2].dest = 3;
        graph3.edge[2].weight = -1;

        graph3.edge[3].src = 3;
        graph3.edge[3].dest = 0;
        graph3.edge[3].weight = -1;

        System.out.println(graph3.BellmanFordBool(graph3, 0));
```

Figure 5: Construction of case 2 from BellmanFord.java

```
true
true
false
```

Figure 6: Console output after running BellmanFordBool method on the 3 cases.

Real World Application

The Bellman-Ford algorithm can be used to find the shortest path between two points in a city by representing the city road network as a graph. The nodes would represent the intersections or junctions, and the edges represent the roads or streets connecting them. Each edge would be associated with a weight that represents the distance or travel time required to traverse the road. By applying this algorithm, the information can be used to optimize traffic flow by directing vehicles along the most efficient routes and minimizing travel times. The path would be valuable to emergency services in order to respond more quickly to emergencies and potentially save lives. It also does apply to public transportation routes by finding the shortest and most efficient paths between bus or train stops. This can reduce travel times for commuters and improve the overall efficiency of the transportation system. This example is inspired from *Brilliant Math & Science Wiki* [4].

References

[1] Depth-First Search (DFS). https://brilliant.org/wiki/depth-first-search-dfs/. Brilliant Math Science Wiki.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009

[3] Bellman-Ford Algorithm | DP-23. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/. Geeks For Geeks.

[4] Bellman-Ford Algorithm. https://brilliant.org/wiki/bellman-ford-algorithm/. Brilliant Math Science Wiki.