Lab Assignment #11

Djikstra’s Algorithm

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 Implement Dijkstra's Algorithm to find the single source shortest paths for a directed graph. This algorithm uses the same Graph implementation which was used in the previous lab. It was further tested with small datasets to verify its correctness. The algorithm was further executed on large datasets to measure real time machine specific performance and perform our time complexity analysis.

2. Program Design

```
public Graph(int size) {
    this.graph = new ArrayList<>(size);
    this.weights = new ArrayList<>(size);
    for (int i = 0; i < size; i++) {
        graph.add(new LinkedList<>());
        weights.add(new HashMap<>());
    }
    this.vertices = new Node[size];
}

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

public void insert(int v1, int v2, double weight, 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]);
    weights.get(v1).put((long) v2, weight);
    weights.get(v2).put((long) v1, weight);

}
```

The functions for retrieving a single vertex from the list of vertices and inserting a new vertex by specifying its neighbor. The insertion function has works with directed weighted graphs. A list of map data structures is used to store weights for all the edges associated to each vertex.

```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, %f), ", v.getValue().intValue(),
                    weights.get(i).get(v.getValue()));
        }
        System.out.print("/ \n");
    }
}

public Double w(Node u, Node v) {
    return weights.get(u.getValue().intValue()).get(v.getValue());
}

public void initializeSingleSource(Node s) {
    for (Node v : vertices) {
        v.setD(Double.MAX_VALUE);
        v.setP(p:null);
    }
    s.setD(d:0D);
}
```

The implementation of the print function to display the contents of the graph, which is the same as used in the previous experiment. The w() function returns the weight of the edge between the two supplied vertices. The initializeSingleSource() function initializes all the vertices of the graph with the required initial values in preparation for the execution of the algorithm. The source node gets a weight of 0 to start off. It signifies that the distance to itself is 0.

```
public void relax(Node u, Node v) {
    // System.out.println(u + " " + v);
    // System.out.println(v.getD() + " " + u.getD() + w(u, v));
    if (v.getD() > u.getD() + w(u, v)) {
        v.setD(u.getD() + w(u, v));
        v.setP(u);
        // System.out.println(u + " " + v);
    }
}

public void dijkstra(Node s) {
    initializeSingleSource(s);
    // System.out.println(Arrays.toString(vertices));
    Set<Node> S = new HashSet<>();
    PriorityQueue<Node> Q = new PriorityQueue<>(Arrays.asList(vertices));
    while (Q.size() > 0) {
        Node u = Q.poll();
        S.add(u);
        for (Node v : graph.get(u.getValue().intValue())) {
            relax(u, v);
            if (Q.contains(v)) {
                Q.remove(v);
                Q.add(v);
            }
        }
    }
}
```

The relax() is the cornerstone of Djikstra's algorithm. D represents the distance value from the source. If the vertex v has a greater distance value than the distance value of the adjacent parent node u added to the weight of the edge between v and u, v gets assigned a new distance value. This gets executed for all the vertices in the graph which have outbound edges to neighboring vertices. The djikstra() function contains the core logic of the algorithm. First it initializes the vertices with the proper initial values for the distance from the source and parent vertex. A set S is maintained to store all the vertices that are included in the shortest path tree formed by the algorithm. The queue Q is a priority queue that helps us select the vertex with the smallest distance value at each level as we proceed away from the source vertex with every iteration. Every adjacent vertex is relaxed with respect to its parent vertex and the algorithm completes after relaxing the neighbors of all the vertices in the graph.

```
public String printShortestPath(Node s, Node v) {
    if (v.equals(s)) {
        return String.format("\n" + s.getValue());
    } else if (v.getP() == null) {
        return String.format("\n no path from %d to %d exists", s.getValue(), v.getValue())
    } else {
        return printShortestPath(s, v.getP()) + String.format("-> %d", v.getValue());

    }
}

public Node[] getAllVertices() {
    return vertices;
}
```

The function used to return a string that represents the shortest path from the provided source vertex using which the algorithm was initiated and any other vertex from the graph.

```
public static Graph getGraphFromFile(String filename, boolean directed) {
    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().strip();
            String elements[] = line.split("\\s+");
            g.insert(Integer.valueOf(elements[0]), Integer.valueOf(elements[1]), Double.valueOf(elements[2]),
                    directed);
        }
        fileScanner.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
        System.exit(1);
    }
    return g;
}
```

The function used to build the graph from the provided datasets.

```
public static void writeShortestPathOutputToFile(Node s, Graph g, String filename) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename, true))) {
        for (Node v : g.getAllVertices()) {
            writer.write(String.format("The minimum distance between %d to %d = %f", s.getValue(), v.getValue(), v.getD()));
            writer.write(g.printShortestPath(s, v));
            writer.write("\n");
        }
        writer.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
        System.exit(1);
    }
}
```

The function used to write the output of the shortest paths of large graphs to a file

```java
public static void driver1(String filename, boolean directed) {

    Graph g = getGraphFromFile(filename, directed);
    // g.print();

    Node s = g.get(i:0);

    g.dijkstra(s);

    writeShortestPathOutputToFile(s, g, String.format("%s-Output", filename));

}

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

    Graph g = getGraphFromFile(filename, directed);
    Node s = g.get(i:0);

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

    writeShortestPathOutputToFile(s, g, String.format("%s-Output", filename));

    System.out.format("%s: Time taken for completion:= %d ns\n", filename, timeElapsed);

}
```

The functions used to test the algorithm, obtain the desired output and calculate the execution times with a provided dataset.

3. **Output**

```
tinyDG.txt: Time taken for completion:= 1294247 ns
mediumDG.txt: Time taken for completion:= 14790494 ns
largeDG.txt: Time taken for completion:= 51260677 ns
```

Output 1 The execution times in nanoseconds recorded for all the datasets provided

4**. Analysis and Conclusions**

Dijkstra's algorithm and Prim's approach for the minimum spanning tree are extremely similar. Create an SPT (shortest path tree) using the specified source as the root, similar to Prim's MST. Maintain two sets, one of which contains vertices that are already included in the shortest-path tree and the other of which contains vertices that are not yet included. Find a vertex that is in the other set (set not yet included in the algorithm) that is closest to the source at each stage.

- *Dijkstra’s algorithm works only for connected graphs.*
- *It works only for graphs that don’t contain any edges with a negative weight.*
- *It only provides the value or cost of the shortest paths.*
- *The algorithm works for directed and undirected graphs.*

Dijkstra's algorithm makes use of breadth-first search (BFS) to solve a single source problem. However, unlike the original BFS, it uses a priority queue instead of a normal first-in-first-out queue. Each item's priority is the cost of reaching it from the source.

Working of *Dijkstra's* Algorithm

Initial Values

- d property – Every node object has a d property which contains the minimum distance from the source node to each node in the graph. At the beginning, the d value for the source node is 0 and is infinity for all other nodes. The value is recalculated and finalized when the shortest distance to every node is found.
- Q – a priority queue of all nodes in the graph. The algorithm completes when Q becomes empty.
- S – a set to indicate which nodes have been visited by the algorithm. At the end. S will contain all the nodes in the graph.

Procedure

- Pop the node v with the smallest dist(v) from Q. In the first run, source node s will be chosen because dist(s) = 0 in the initialization.
- Add node v to S, to indicate that v has been visited.
- Update D value for each adjacent node of the current node v as follow:
  - If v.d + weight(v, u) < u.d so update u.d to the new minimal distance value,
  - Otherwise no updates are made to the u.d.
- The progress will stop when Q is empty, or in other words, when S contains all nodes, which means every node has been visited.

Time Complexity Analysis

- It takes **O(|V|)** time to construct the initial priority queue of |V| vertices.
- With adjacency list representation, all vertices of the graph can be traversed using BFS. Therefore, iterating over all vertices' neighbors and updating their dist values over the course of a run of the algorithm takes **O(|E|)** time.
- The time taken for each iteration of the loop is **O(|V|)**, as one vertex is removed from Q per loop.
- The binary heap data structure allows us to extract-min (remove the node with minimal dist) and update an element (recalculate u.d) in **O(log|V|)** time.
- Therefore, the time complexity becomes **O(|V|) + O(|E| * log|V|) + O(|V| * log|V|)**, which is **O((|E|+|V|) * log|V|) = O(|E| * log|V|)**, since |E| >= |V| - 1 as G is a connected graph.

Applications of Dijkstra's Algorithm

- *Digital Mapping Services like Google Maps:* Suppose yo want to travel from on city to another city. You use Google maps to find the shortest route. How will it work? Assume the city you are in to be the source vertex and your destination to be another vertex. There will still be many cities between your destination and starting point. Assume those cities to be intermediate vertices. The distance and traffic between any two cities can be

assumed to be the weight between each pair of vertices. Google maps will apply Dijkstra's algorithm on the city graph and find the shortest path.

- *Designate Servers:* In a network, Dijkstra's Algorithm can be used to find the shortest path to a server for transmitting files between the nodes on a network.

- *IP Routing:* Dijkstra's Algorithm can be used by link state routing protocols to find the best path to route data between routers.

Observations based on the lab experiment

- The execution time increased greatly with the increase in the number of edges. The usage of the PriorityQueue enabled us to obtain the most efficient implementation of the algorithms which use an adjacency list :- the one which uses a binary heap.

5. References

- https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html
- https://www.baeldung.com/java-write-to-file
- https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/
- https://www.scaler.com/topics/data-structures/dijkstra-algorithm/
- https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/
- https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
- https://www.baeldung.com/cs/dijkstra-time-complexity#:~:text=2.2.-,Dijkstra's%20Algorithm,reaching%20it%20from%20the%20source.