Ruth Tilahun
CS337
Lab 7

Dijkstra's Shortest Path Algorithm

Introduction

Dijkstra's algorithm is a vital algorithm for solving various real-world problems. One of its most common applications is in telecommunication networks. For instance, Dijkstra's algorithm calculates the shortest paths between network nodes based on bandwidth. Dijkstra's algorithm is mainly used to find the shortest path between two vertices of a weighted graph where the edge weights represent distance or travel time. For example, if a graph has vertices that represent cities and edges that represent routes between cities, the weight on an edge or route would be the distance or travel time between two cities. In this lab, we will implement Dijkstra's algorithm to find the shortest path between vertices.

Graph ADT

Before we get into our implementation of Dijkstra's algorithm, we will look at how we designed our Graph ADT.

```
// Define Edge structure
Edge:
    start: integer
    end: integer
    weight: Double

    Edge(start, end, weight):
        this.start = start
        this.end = end
        this.weight = weight

// Define Graph structure
Graph<V, E>:
    n: integer
    adjList: map(integer, list(Edge))

    Graph(n):
        this.n = n
        adjList = new HashMap<>()
        for i from 0 to n-1:
            adjList.put(i, new LinkedList<Edge>())

    addEdge(v, u, weight):
        edge = new Edge(v, u, weight)
        adjList.get(v).add(edge)
```

Figure 1: Pseudocode for Graph structure

For the sake of simplicity, we designed our graph to be compatible with the data files we will be working with in this Lab. Each data file that will be represented in a graph provides the number of vertices, *n,* numbered from 0 to n -1 and a list of directed edges in the form $<X, Y, W_{XY}>$ where vertex X goes to vertex Y with a weight of W_{XY}.

In Figure 1, the *Edge* class constructs a directed edge from vertex *start* to vertex *end* with a weight/distance of *weight.* We didn't create a *Vertex* class to construct vertices since each vertex is simply an integer from 0 to n-1. We employ the adjacency list representation to represent the graph where we index an *n* element array by the vertices and the array entry for each vertex *v* is the list of all the vertices adjacent to *v*. To that end, we use Java's HashMap with vertices of the graph as keys and a list of *Edges* associated with the vertex, as values. To construct a graph with *n* vertices, we initialize our HashMap with vertices from 0 to n – 1 as keys and empty lists of *Edges* as values the vertices. Then, whenever we need to add an edge to the graph, the *addEdge* method will first create a new *Edge* with the provided data and add it to the list *Edges* of the appropriate vertex in the map.

Dijkstra's Algorithm

Given a certain vertex *v*, Dijkstra's algorithm calculates the shortest paths from *v* to all other vertices.

```
Procedure DIJKSTRA(G, s)
Inputs: G as described above,
         s a starting vertex.
Results: For each non-source vertex v in V, shortest[v] is
the weight of the shortest path from s to v and pred[v] is
the vertex preceding v on some shortest path.
Let:
    shortest[v]: an array of length n, with one entry for
each vertex.
    pred[v]: an array of length n. pred[v] contains the
vertex that is the predecessor of v on the shortest path
from s to v.
    Q: A min priority queue of nodes. The key is shortest
1. for each vertex v in V:
    shortest[v] = ∞
    pred[v] = null
shortest[s] = 0, pred[s] = null

2. for each vertex v in V:
    insert v in Q
3. while Q is not empty:
    u = remove from Q the vertex with smallest value of
shortest
    for each vertex v adjacent to u:
        RELAX(u, v)
```

```
Procedure RELAX(u, v)
Inputs: u, v are vertices in V such that there is an edge
between them
Result: Updates the values in shortest[v] and pred[v] if
possible
if shortest[u]+weight(u, v) < shortest[v]
   shortest[v] = shortest[u]+weight(u, v)
   pred[v] = u
   update Q as needed for the new value of shortest[v]
```

Figure 2: Dijkstra's algorithm and helper method RELAX

In Figure 2, we take in a graph and a starting vertex and output a collection containing the shortest distances from the starting vertex to all other vertices and another collection containing the predecessor vertex of each vertex on the shortest path from the starting vertex. For example, let's say we input graph *G* and starting vertex $s = 0$ and our output is *shortest* = [0.0, 1.5, 0.2, 0.9, 0.3] and *pred* = [-1, 4, 0, 2, 3]. We think of the indices of both arrays as vertices of *G.* Thus, it means that the weight of the shortest path from vertex 0 to vertex 0 is 0.0 and it has no predecessors since pred[(vertex) 0] = -1 is an invalid vertex. Similarly, the weight of the shortest path from vertex 0 to vertex 1 is 1.5 and vertex 1 has 4 as its predecessor. In other words, we must go through vertex 4 to get to vertex 1 from vertex 0. Now since vertex 4 has a predecessor vertex 3, it means that we must go through vertex 3 and vertex 4 to get to vertex 1, and so on.

First, we initialize every vertex's weight of the shortest path in *shortest* to infinity, except for the starting vertex *s* which we set to 0, because we know that the distance between *s* and *s* is 0. We will also set the *pred* value of each vertex to -1. We will the values of *shortest* and *pred* as we analyze each vertex. The min priority queue *Q* is then initialized with all the vertices where each vertex *v* is assigned a priority based on the value of shortest[v] which is v's weight on the shortest path from s. Therefore, since *shortest[s]* = 0 is less than all other values in shortest, s will be given a priority in the beginning. We use Java's PriorityQueue for this purpose because the head of the queue is the least element with respect to the specified ordering and it can easily be removed from the queue using the *poll()* method. At each iteration of the outer loop in step 3, a vertex *u* that is given priority is removed/polled from the priority queue and each of its adjacent vertex *v* is examined through the RELAX procedure. The relax procedure updates shortest[v] if the weight of the path from u to v is smaller than the current weight of the path to v. Since we always remove the vertex with lowest *shortest* value from the priority queue, we know that shortest[u] is always going to be the weight of the shortest path to u. We repeat this process until there are no vertices left in *Q.*

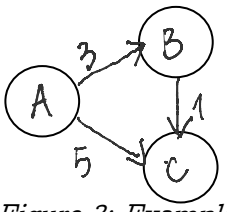

Figure 3: Example graph

In Figure 3, let's say our starting vertex is A. Before the first iteration of step 3, we have *shortest = [0, ∞, ∞]*, *pred = [-1, -1, -1]* and *Q = [A, B, C].* As we enter the loop in step 3, we remove A from Q because it has the lowest *shortest* value. and analyze its neighbors. When we analyze vertex B, we find that the path from A to B has weight 3 which is smaller than ∞, meaning *shortest[A]* = 0 and *weight (A, B)* = 3, so 0+3=3 is less than *shortest[B]* = ∞. Thus, we update *shortest[B]* to 3, pred[B] to A and Q[B] with the new *B* that has the updated *shortest* value. Similarly analyzing vertex *C* results in *shortest[C] = 5, pred[C] = A* and *Q[C]* with the updated value of *shortest[C*]. At this point, *shortest = [0, 3, 5]* and *pred[-1, A, A].* The next iteration of the outer loop of step 3 removes *B* from *Q* because *shortest[B] < shortest[A].* B has one neighbor, C. The path from *B* to *C* costs *shortest[B] + weight(B,C)* = 3+1 =4. This new path costs less than the current cost of getting to *C*, *shortest[C] = 5*. Accordingly, we update *shortest[C]* to 4 and the predecessor of *C* to *B.* Since *C* is the last vertex in *Q*, we remove it but find that it has no neighbors, so the loop terminates. The final product is *shortest = [0, 3, 4]* and *pred[-1, A, B].*

The running time of this algorithm is Θ(V) for step 1 where V is the number of vertices. In our implementation, we use Java's PriorityQueue as *Q* to put the n vertices into, which provides Θ(log(V)) running time to add and remove from the queue. (JavaDoc). Thus, step 2 takes Θ(Vlog(V)) time. The outer loop of step 3 iterates V times, so it takes Θ(V) time. Removing the lowest *shortest* value V times takes Θ(Vlog(V)) time. The inner loop examines each vertex and edge exactly once, thus it has a running time of Θ(E). Finally, RELAX takes Θ(Elog(V)) time because we need to update *Q.* The overall complexity of the algorithm is Θ(Elog(V)).(Cormen, 2013)

Finding Highest Cost (Task 2)

To find the highest, shortest path cost from a source node to a reachable node, we go through the array *shortest* from Dijkstra's algorithm and find the maximum value. The destination of the highest cost path is the node which has the highest *shortest* value. The time complexity of this algorithm is worst-case Θ(V) if the source node has a path to every vertex.

The highest cost from the shortest path from node 3310 in the huge graph is approximately 1.7977 and it goes to node 17.

Conclusion

Dijkstra's algorithm is designed to find the shortest path between a single source node and all other nodes in a weighted graph. Its time complexity is $\Theta(E\log(V))$, where V is number of vertices and E is number of edges. In this Lab, we implemented the algorithm and used it to find the shortest path between two nodes and to get the vertex that has the highest cost to be reached.

References

Cormen, Thomas H. *Algorithms Unlocked*. MIT Press, 2013.

PriorityQueue (Java Platform SE 8). (n.d.). Retrieved March 31, 2024, from https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

Appendix A: Program

```
class Edge {
    int start;
    int end;
    double weight;

    Edge(int start, int end, double weight) {
        this.start = start;
        this.end = end;
        this.weight = weight;
    }

    public int getEnd() {
        return end;
    }

    public double getWeight() {
        return weight;
    }

    public String toString() {
        return "(" + start + ", " + end + ", " + weight + ")";
    }
}

class Graph<V, E> {
    private int numVertices;
    private Map<Integer, List<Edge>> adjList;

    Graph(int numVertices) {
        this.numVertices = numVertices;
        adjList = new HashMap<>();
        for (int i = 0; i < numVertices; i++) {
            adjList.put(i, new LinkedList<Edge>());
        }
    }

    public void addEdge(int v, int u, double weight) {
        Edge edge = new Edge(v, u, weight);
        adjList.get(v).add(edge);
    }

    public String toString() {
        StringBuilder result = new StringBuilder();
```

```
        for (Map.Entry<Integer, List<Edge>> entry :
adjList.entrySet()) {
            result.append(entry.getKey()).append(":
").append(entry.getValue()).append("\n");
        }
        return result.toString();
    }

    public void dijkstra(Graph<V, E> G, int sourceV, int destV) {
        double[] shortest = new double[numVertices];
        int[] pred = new int[numVertices];
        PriorityQueue<Integer> Q = new
PriorityQueue<>(Comparator.comparingDouble(vertex ->
shortest[vertex]));

        for (int i = 0; i < numVertices; i++) {
            shortest[i] = Double.MAX_VALUE;
            pred[i] = -1;
        }

        shortest[sourceV] = 0;
        for (int i = 0; i < numVertices; i++) {
            Q.add(i);
        }

        while (!Q.isEmpty()) {
            int u = Q.poll();
            for (Edge edge : G.adjList.get(u)) {
                int v = edge.getEnd();
                double weight = edge.weight;
                if (shortest[u] + weight < shortest[v]) {
                    shortest[v] = shortest[u] + weight;
                    pred[v] = u;
                    Q.remove(v);
                    Q.add(v);
                }

            }

        }
        for (int i = 0; i < numVertices; i++) {
            System.out.println("Shortest path from " + sourceV + "
to " + i + " is " + shortest[i]);
        }
        highestCost(sourceV, shortest);
```

```
    }
    void printSolution(int source, double[] shortest, int[] pred) {
        System.out.println("Shortest distances from source " +
source + ":");
        for (int i = 0; i < adjList.size(); i++) {
            System.out.println("Vertex " + i + ": " + shortest[i]);
            printPath(source, i, pred);
        }
    }
    void printPath(int source, int vertex, int[] pred) {
        if (vertex == source) {
            System.out.print(source + " ");
            return;
        }
        if (pred[vertex] == -1) {
            System.out.print("No path");
            return;
        }
        printPath(source, pred[vertex], pred);
        System.out.print(vertex + " ");
    }
public class Main {
    public static void main(String[] args) {
        try (Scanner input = new Scanner(new File("hugeEWD.txt"))) {
            int numVertices = Integer.parseInt(input.nextLine());
            int numEdges = Integer.parseInt(input.nextLine());
            Graph<Integer, Edge> graph = new Graph<>(numVertices);

            while (input.hasNextLine()) {
                String w = input.nextLine();
                int start = Integer.parseInt(input.next());
                int end = Integer.parseInt(input.next());
                double weight = Double.parseDouble(input.next());
                graph.addEdge(start, end, weight);

            }

            int source;
            int destination;
            String in;
            Scanner sc = new Scanner(System.in);
            while (true) {
                System.out.print("Enter a source vertex(RETURN to
exit): ");
```

```
                in = sc.nextLine();
                if (in.equals("")) {
                    break;
                }
                source = Integer.parseInt(in);
                System.out.print("Enter a destination vertex(RETURN
to exit): ");
                in = sc.nextLine();
                if (in.equals("")) {
                    break;
                }
                destination = Integer.parseInt(in);
                graph.dijkstra(graph, source, destination);
            }

        } catch (Exception ee) {
            System.err.println(ee);

        }

    }
}
```

Appendix B: Task

Task 2

Most Expensive to get to is node 17 with a cost of 1.7976931348623157E308