[Date]

Data Structure

Shortest Paths Algorithms

Abdelmonem Badr Elsawy. 26
Mahmoud Fathy Aboeleneen. 47

Abstract:

In this lab we implemented two shortest paths algorithms which are Dijkstra and Bellman-Ford.

Dijkstra Algorithm

This algorithm finds shortest paths from the source to all other nodes in the graph, producing a shortest path tree. Its time complexity is O (V^2) but can reach less than that when using priority queue. Dijkstra algorithm can't handle negative weights. But, it is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

Bellman-Ford Algorithm

The Bellman-Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is capable of handling graphs in which some of the edge weights are negative numbers. It works in O (V E) time and O (V) space complexities where V is the number of vertices and E is the number of edges in the graph.

Algorithms Pseudo codes:

1. Dijkstra:

   Algorithm

   1) create a Boolean array called checked to keep track the vertices which included in the shortest path tree and assign all elements to false.

   2) Assign the distance of all vertices to INFINITE and assign the distance of source vertex to zero.

   3) for j from 0 to number of vertices:

```
min = INFINITE
index = -1

For i from 0 to number of vertices

      If !checked[i] && distances[i] < min:

            min = distances[i]

            index = i

      end if.

End for

Checked[index] = true

For i from 0 to number of adjacent lists of vertex (index)

      edge = adj.get(i)

      f = the ith adjacent vertex of (index)
```

```
                If distances[index] + edge.getWeight() < distances[f] &&!ckecked[f]:
                    distances[f] = distances[index] + edge.getWeight.
                end if.
            End for.
        End for.
        End dijkstra.
```

Code snapshots:

Read graph.

```java
public void readGraph(File file) {
    try {
        BufferedReader br = new BufferedReader(new FileReader(file));
        String line ;
        line = br.readLine();
        Pattern first = Pattern.compile("([0-9]+)[ ]+([0-9]+)");
        Matcher m = first.matcher(line);
        m.find();
        size = Integer.parseInt(m.group(1));
        numEdges = Integer.parseInt(m.group(2));
        adjList.clear();
        for(int i=0;i<size;i++) {
            adjList.add(new ArrayList<Edge>());
        }
        Pattern second = Pattern.compile("([0-9]+)[ ]+([0-9]+)[ ]+(-?[0-9]+)");
        while((line = br.readLine())!=null) {
            m=second.matcher(line);
            m.find();
            int start = Integer.parseInt(m.group(1));
            int finish = Integer.parseInt(m.group(2));
            int weight = Integer.parseInt(m.group(3));
            edges.add(new Edge(start,finish,weight));
            adjList.get(start).add(new Edge(start,finish,weight));
        }
        // check inValid graph
        if(edges.size()!=numEdges) {
            throw new RuntimeErrorException(new Error());
        }
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        throw new RuntimeErrorException(new Error());
    } catch (IOException e) {
        // TODO Auto-generated catch block
        throw new RuntimeErrorException(new Error());
    }
}
```

Dijkstra.

```java
@Override
public void runDijkstra(int src, int[] distances) {
    Arrays.fill(distances, OO);
    distances[src] = 0;
    boolean[] checked = new boolean[distances.length];
    Arrays.fill(checked, false);
    for (int j = 0; j < distances.length; j++) {
        int index = -1;
        int min = INF;
        for (int i = 0; i < checked.length; i++) {
            if (!checked[i] && distances[i] < min) {
                min = distances[i];
                index = i;
            }
        }
        if (index == -1) {
            return ;
        }
        checked[index] = true;
        dijkstra.add(index);
        ArrayList<Edge> adj = adjList.get(index);
        for(int i = 0; i < adj.size(); i++) {
            Edge edge = adj.get(i);
            int f = edge.getFinish();
            if(distances[index] + edge.getWeight() < distances[f] && !checked[f]) {
                distances[f] = distances[index] + edge.getWeight();
            }
        }
    }
}

@Override
```

Bellman Ford.

```java
@Override
public boolean runBellmanFord(int src, int[] distances) {
    // TODO Auto-generated method stub
    Arrays.fill(distances, 00);
    distances[src] = 0;
    boolean noNegativeCycles = true;
    for (int i = 0; i < size; i++) {
        boolean noChange = true;
        for (int j = 0; j < edges.size(); j++) {
            int start = edges.get(j).getStart();
            int finish = edges.get(j).getFinish();
            int weight = edges.get(j).getWeight();
            // relaxation step
            if (distances[start] + weight < distances[finish]) {
                distances[finish] = distances[start] + weight;
                noChange = false;
                if (i == size - 1) {
                    noNegativeCycles = false;
                }
            }

        }
        // if no relaxation then we get shortest paths
        if (noChange) {
            return true;
        }
    }
    // check if there is no negative cycles
    return noNegativeCycles;
}
```

GitHub Link:

https://github.com/Aboeleneen/datastructure2/tree/master/Graph