2019

Shortest Path assignment

NAME:MAHMOUD HAMED SHARSHAR
ID:46

Requirement specification:

- Implementation of methods to get shortests paths in graph.
- Implementation of Dijkstra algorithm.
- Implementation of Bellman-Ford Algorithm.

Data Structure used:

- Array: to store distances of each node from source node.
- Priority Queue: used in Dijkstra algorithm to store nodes with their distances as a priority and remove minimum in each iteration and put it Known region (Array List).

Algorithms Description:

- 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) in case of use array to find minimum 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 paths 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.

Pseudocode:

- ## Dijkstra Algorithm:
  - for all u ∈ V:
    - dist[u] ←∞
  - dist[S] ← 0
  - Q← MakeQueue(V) {dist-values as keys}
  - while Q is not empty:
    - u ← ExtractMin(Q)
    - for all (u, v) ∈ E:

        - if dist[v] > dist[u] +w(u, v):
            - dist[v] ← dist[u] +w(u, v) ChangePriority(Q, v, dist[v])

- Bellman-Ford Algorithm:
    - for all u ∈ V:
        - dist[u] ←∞
    - dist[S] ← 0
    - for I =0 to |V|:
        - for all (u, v) ∈ E:
            - if dist[v] > dist[u] +w(u, v):
                - dist[v] ← dist[u] +w(u, v)
                - If I == v:
                    - There is a negative cycle in the graph.

Code snapshots:

- ## Read Graph:

```java
@Override
public void readGraph(File file) {
    try {
        @SuppressWarnings("resource")
        BufferedReader fileReader = new BufferedReader(new FileReader(file));
        String line = fileReader.readLine();
        int iteration = -1;
        while (line != null) {
            String[] parts = line.split(" ");
            if (iteration == -1) {
                this.numOfVertices = Integer.parseInt(parts[0]);
                this.numOfEdges = Integer.parseInt(parts[1]);
                for (int i = 0; i < this.numOfVertices; i++)
                    this.graph.add(new ArrayList<>());
            } else
                graph.get(Integer.parseInt(parts[0]))
                        .add(new Pair<Integer, Integer>(Integer.parseInt(parts[1]), Integer.parseInt(parts[2])));
            iteration++;
            line = fileReader.readLine();
        }
        if (iteration != this.numOfEdges)
            throw new RuntimeException();
    } catch (FileNotFoundException e) {
        throw new RuntimeException();
    } catch (IOException e) {
        throw new RuntimeException();
    }
}
```

- ## Bellman-Ford Algorithm:

```java
@Override
public boolean runBellmanFord(int src, int[] distances) {
    for (int i = 0; i < distances.length; i++) {
        distances[i] = Integer.MAX_VALUE / 2;
    }
    distances[src] = 0;
    boolean cycleFree = true;
    for (int i = 0; i < distances.length; i++) {
        boolean flag = false;
        for (int j = 0; j < distances.length; j++) {
            ArrayList<Pair<Integer, Integer>> neighbours = graph.get(j);
            for (int k = 0; k < graph.get(j).size(); k++) {
                Pair<Integer, Integer> child = neighbours.get(k);
                if (distances[child.getKey()] > (long) distances[j] + (long) child.getValue()) {
                    if (i == distances.length - 1)// test negative cycle
                        cycleFree = false;
                    distances[child.getKey()] = distances[j] + child.getValue();
                    flag = true;
                }
            }
        }
        if (!flag)
            break;
    }
    return cycleFree;
}
```

- Dijkstra Algorithm:

```
@Override
public void runDijkstra(int src, int[] distances) {
    // initialize all distances to infinity
    int length = distances.length;
    for (int i = 0; i < length; i++)
        distances[i] = Integer.MAX_VALUE / 2;
    distances[src] = 0;
    // make custom comparator priority queue
    PriorityQueue<Integer> Q = new PriorityQueue<>(new Comparator<Integer>() {

        @Override
        public int compare(Integer node1, Integer node2) {
            if (distances[node1] > distances[node2])
                return 1;
            else if (distances[node1] < distances[node2])
                return -1;
            else
                return 0;
        }
    });
    // add vertices to priority queue
    Q.add(src);
    for (int i = 0; i < length; i++) {
        if (i != src) {
            Q.add(i);
        }
    }

    while (!Q.isEmpty()) {
        int min = Q.remove();
        DijkstraProcessedOrder.add(min);
        ArrayList<Pair<Integer, Integer>> adj = this.graph.get(min);
        for (int i = 0; i < adj.size(); i++) {
            Pair<Integer, Integer> child = adj.get(i);
            if (distances[child.getKey()] > (long) distances[min] + (long) child.getValue()) {
                distances[child.getKey()] = distances[min] + child.getValue();
                // change priority
                Q.remove(child.getKey());
                Q.add(child.getKey());
            }
        }
    }
}
```