ITCS 6114

Project 2

Graph Algorithms and Related Data Structures

Team Members

Meghana Palaparthi
Tarun Ravada

Algorithms

Single Source Shortest Path

The task is to find the shortest path tree in a given weighted graph. The graphs could be weighted or unweighted, with no negative edges.

Dijkstra's Algorithm

Given a graph and a source vertex in the graph, Dijkstra's algorithm calculates the shortest path from the source vertex to all other vertices in the graph.

The algorithm is iterative and works by growing a cloud of vertices starting from the source vertex. At each iteration, the next vertex adjacent to the cloud, with the least distance from the source node is added to the cloud.

For Dijkstra's algorithm to work the graph must be a connected graph, with no negative weight edges. This is a greedy algorithm as it selects the vertex with the least distance at each iteration.

Minimum Spanning Tree

The task is to find a spanning tree that connects all the vertices of a graph, with the least total weight. The graph is connected undirected and weighted.

Kruskal's Algorithm

Given an undirected, weighted and connected graph Kruskal's algorithm finds a spanning tree of the graph.

The algorithm starts with a forest in which each tree corresponds to one vertex of the graph. At each iteration the algorithm finds the least weight edge that connects any two trees and adds that to the MST. Once all iterations are completed, we have a tree that connects all the vertices of the graph with the least possible edge weights.

The algorithm is a greedy algorithm as it selects the edge with the least weight at each iteration.

Data Structures used

Dijkstra's Algorithm

The data structures used for this algorithm are List and Graph.
The graph data structure is implemented using List.

Lists are used for the following:

- Store weights of edges.
- Store and update distance between source and destination vertices at each step.
- Maintain record of intermediate nodes leading to shortest distance between source and destination.
- Store the list of nodes added to the cloud at each step.

List operations performed are:

- Access via index
- Modify via index

List Access via index and modify via index run in constant time.

Kruskal's Algorithm

The data structures used for this algorithm are List, Set, Dictionary, and UndirectedGraph.
The UndirectedGraph data structure is implemented using List and Set.

- Set is used to maintain a record of all the vertices of the graph.
- Dictionaries are used to track the trees and their parents in Kruskal's algorithm.
- Lists are used to represent the edges of the graph.

List operations performed are:

- Access via index
- Append to list

List access via index and append run in constant time

Set operations performed are:

- Add element
- Access element

Set Add element and Access element run in constant time

Dictionary operations performed are:

- Add element
- Access element

Dictionary Add element and Access element run in constant time

Runtime Analysis

Dijkstra's Algorithm

Pseudocode

```
Dijkstra(graph, source):

    distance = List()           # size v
    parent = List()             # size v
    shortPathTree = List()      # size v

    # Initializing takes v time, where v is number of vertices
    distance = ∞
    parent = -1
    shortPathTree = False

    distance[source] = 0
    vertices = length(graph)

    index = 0

    # Runs max v times
    while index < vertices do

        # get node with minimum weight
        # Runs in v time
        u = minDistance(distance, shortPathTree)

        # add the min weight node to cloud of accessed nodes
        shortPathTree[u] = True

        # Runs v times
        for each vertex v ∈ graph.V

            # Graph[u][v] specifies weight of edge u-v
            if graph[u][v] > 0 and shortPathTree[v] == False and
            Distance[v] > distance[u] + graph[u][v]:

                distance[v] = distance[u] + graph[u][v]
                parent[v] = u
```

```
        return distance, parent

    minDistance(distance, shortPathTree):
        min = ∞
        # Runs v times
        for v in 1..vertices
            if distance[v] < min and shortPathTree[v] == False:
                min = distance[v]
                min_index = v
        return v
```

Runtime

minDistance():	O(**v**)
minDistance() is called v times:	O($\mathbf{v^2}$)
In Dijkstra(), the outer loop runs v times and the inner loop runs log v times:	O($\mathbf{v^2}$)

Overall Runtime = O($\mathbf{v^2}$)

Where **v** is the no. of vertices in the graph.

Kruskal's Algorithm

Pseudocode

```
    kruskalMST(graph):

        mst = UndirectedGraph()
        vertices =Set()
        edges = List()
        roots = Dictionary()
        ranks = Dictionary()

        # Runs in e Log(e) time, where e is no.of edges of graph
        vertices, edges = sortEdges(graph)

        # Runs in v time, where v is no.of vertices of graph
        roots, ranks = initForest(vertices)

        i = 0
        j = 0
```

```
# Runs max e times
while j < (vertices.length - 1) do
        u, v, w = edges[ i ]      # constant time
        i = i + 1
        x = getRoot(u, roots)   # log(v) time, using union by rank
        y = getRoot(v, roots)   # aka Find set

        if x != y:          # runs max v-1 times
                j = j + 1   # Added new edge to MST
                mst.addEdge(u, v, w)          # constant time
                union(x, y, roots, ranks)        # log(v) time, union by rank

return mst
```

Runtime

Sorting the edges:	O(**e** log(**e**))
Initializing Spanning forest:	O(**v**)
Find set of a vertex:	O(**e** log(**v**))
Union of two sets:	O((**v-1**) log(**v**))

Overall runtime = **O(e log(e))**

Where **e** is no.of edges and **v** is no.of vertices.

Sample Input and Output

Single Source Shortest Path

Input File	Output
9 21 D 1 2 8 1 6 13 1 3 3 2 3 2 2 4 1 3 2 3 3 4 9 3 5 2 4 5 4	Distance Path 0 1 6 1-3-2 3 1-3 7 1-3-2-4 5 1-3-5 10 1-3-5-6 11 1-3-5-6-7 9 1-3-2-4-8 9 1-3-5-9

4 7 6 4 8 2 5 1 5 5 4 6 5 6 5 5 9 4 6 7 1 6 9 7 7 5 3 7 8 4 8 9 1 9 7 5 1	

Minimum Spanning Tree

Input File	Output
12 16 U A D 7 A I 9 A L 2 B D 3 B E -1 C F 6 C G -1 D K -1 D J 4 E K -2 F H 8 G K 1 G L -2 G H 7 H L 4 H K 8	Edges E -- K == -2 G -- L == -2 B -- E == -1 C -- G == -1 D -- K == -1 G -- K == 1 A -- L == 2 D -- J == 4 H -- L == 4 C -- F == 6 A -- I == 9 Total Weight 19

Submission Details

Files

- Data
- RunMST.ipynb
- MST_utils.py
- RunShortestPath.ipynb
- ShortestPath_utils.py

Note: Tests on four different graphs for both algorithms can be seen in RunMST.ipynb and RunShortestPath.ipynb files.

To test Shortest path

- Run the RunShortestPath.ipynb notebook.
- Details to use different input files are included in the notebook.
    - The variable *path* is set to 'data/'. If input files are placed elsewhere, change this variable to point to the relative path.
    - Choose among input1, input2, input3, input4 to run the program for various inputs and pass it to runShortestPath() function.
- ShortestPath_utils.py contains the implementation of Dijkstra's algorithm and the Graph data structure.
- **Note**: The algorithm requires a specific input format. Input vertices must be numbers only. Please refer to the provided input files (SP1) for example.
- Sample input format:

```
9  21  D
1  2  8
1  6  13
```

To test Minimum Spanning Tree

- Run the RunMST.ipynb notebook.
- Details to use different input files are included in the notebook.
    - The variable *path* is set to 'data/'. If input files are placed elsewhere, change this variable to point to the relative path.
    - Choose among input1, input2, input3, input4 to run the program for various inputs and pass it to runMST() function.
- MST_utils.py contains the implementation of Kruskal's algorithm and the Undirected graph data structure.