Algorithm Analysis and Data Structures

Assignment 4

Name: Vignesh Viswanathan
UTD ID: 2021502907
Net ID: VXV190028

Problem Statement:

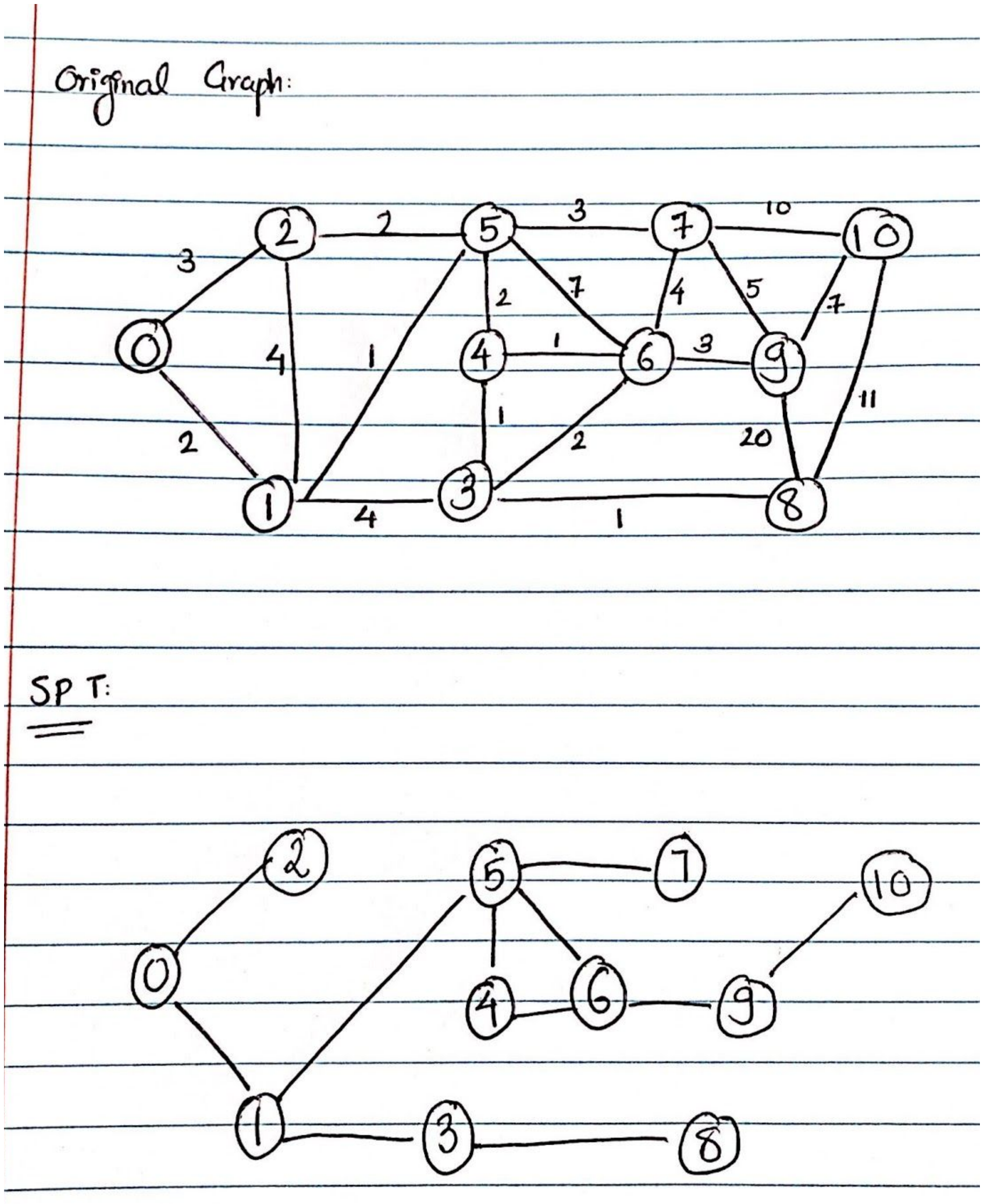
a) Implement Dijkstra's Algorithm for finding the shortest path from a vertex to all other vertices.

b) The graph must contain at least 10 vertices and 20 edges.

c) Print the Shortest Path Tree.

d) Use the Min Heap or the Priority Queue to implement the algorithm.

Input Graph:



Code in C++:

// Just for reference. Cpp file also submitted

```cpp
// C++ code to implement Djikstra Algorithm For Shortest Path


#include <iostream>
```

```
// Using the vector instead of an array, as the vector is dynamic

#include <vector>

// Using the library queue to implement the min heap in the form of a priority queue

#include <queue>

using namespace std;

// Array to store the parents of the resultant Shortest Path Tree
int parent[11] = {-1};

// Array to store the minimum distances
int dis[11] = {0};
// Each adjList[i] holds all the adjacent of node i.

// The first int is the vertex of the adjacent node, the second int is the weight of the path.

vector<vector<pair<int, int> > > FormAdjList()
{
  // Our adjacency list. by the name adjList()

  vector<vector<pair<int, int> > > adjList;

  // We have 10 vertices, so we set n = 11, to iterate through 0 - 10.

  const int n = 11;

  for (int i = 0; i < n; i++)
  {
    // Create a vector to represent a row, and add it to the adjList.
```

```
  vector<pair<int, int> > row;
  adjList.push_back(row);
}

// Adding our actual edges into the adjacency list. The index indicates the vertex
// Push_Back adds the adjacent vertices and weights to each edge

// Eg: Make_pair(1, 2) indicates an edge from 0 -> 1 with a weight of 2

adjList[0].push_back(make_pair(1, 2));
adjList[0].push_back(make_pair(2, 3));

adjList[1].push_back(make_pair(0, 2));
adjList[1].push_back(make_pair(1, 9));
adjList[1].push_back(make_pair(2, 4));
adjList[1].push_back(make_pair(3, 4));
adjList[1].push_back(make_pair(5, 1));

adjList[2].push_back(make_pair(0, 3));
adjList[2].push_back(make_pair(1, 4));
adjList[2].push_back(make_pair(5, 2));

adjList[3].push_back(make_pair(1, 4));
adjList[3].push_back(make_pair(4, 1));
adjList[3].push_back(make_pair(6, 2));
adjList[3].push_back(make_pair(8, 1));

adjList[4].push_back(make_pair(3, 1));
adjList[4].push_back(make_pair(5, 2));
adjList[4].push_back(make_pair(6, 1));

adjList[5].push_back(make_pair(1, 1));
adjList[5].push_back(make_pair(2, 2));
adjList[5].push_back(make_pair(4, 2));
```

```
  adjList[5].push_back(make_pair(6, 7));
  adjList[5].push_back(make_pair(7, 3));

  adjList[6].push_back(make_pair(3, 2));
  adjList[6].push_back(make_pair(4, 1));
  adjList[6].push_back(make_pair(5, 7));
  adjList[6].push_back(make_pair(7, 4));
  adjList[6].push_back(make_pair(9, 3));

  adjList[7].push_back(make_pair(5, 3));
  adjList[7].push_back(make_pair(6, 4));
  adjList[7].push_back(make_pair(9, 5));
  adjList[7].push_back(make_pair(10, 10));


  adjList[8].push_back(make_pair(3, 1));
  adjList[8].push_back(make_pair(7, 9));
  adjList[8].push_back(make_pair(9, 20));
  adjList[8].push_back(make_pair(10, 11));

  adjList[9].push_back(make_pair(6, 3));
  adjList[9].push_back(make_pair(7, 5));
  adjList[9].push_back(make_pair(8, 20));
  adjList[9].push_back(make_pair(10, 7));

  adjList[10].push_back(make_pair(7, 10));
  adjList[10].push_back(make_pair(8, 11));
  adjList[10].push_back(make_pair(9, 7));

  // The graph is represented as an adjacency list which is in the form of a vector.

  return adjList;
}
```

```
void printPath(int parent[], int j)
{
  // Base Case : If j is source
  if (parent[j] == -1)
    return;

  printPath(parent, parent[j]);

  cout << " ->" << j;
}

// A utility function to print the constructed distance
// array
int printSPT(int dist[], int n, int parent[])
{
  int src = 0;
  cout << "Vertex\t  Distance  \t\tPath";
  for (int i = 0; i < n; i++)
  {
    cout << "\n"
         << src << "-> " << i << "\t\t " << dist[i] << "\t\t " << src;
    printPath(parent, i);
  }
}

// Given an Adjacency List, find all shortest paths from "source" vertex to all other vertices.

vector<int> Dijkstra(vector<vector<pair<int, int> > > &adjList, int &source)
{
  cout << "\nGetting the shortest path from " << source << " to all other nodes.\n";

  vector<int> dist;

  // Initialize all source->vertex distances as infinite.
```

```
int n = adjList.size();

for (int i = 0; i < n; i++)
{
  // Define "infinity" as a certain big number

  dist.push_back(10000000);
}

// Create a Min Heap, in the form of a priority queue. The Priority queue is similar to the heap
// In the sense that, elements are popped from the top and the distances to the nodes are updated acc to
each pop

// We call the priority queue pq;

priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > pq;

// Add source to pq, where distance from source->source is 0.
pq.push(make_pair(source, 0));
dist[source] = 0;

// Now, implementing Djikstra's; While pq is not empty

while (pq.empty() == false)
{
  // Get min distance vertex from pq. We store this in (u)

  // As pq is a priority queue, the root of heap is represented by the element at the top of the queue
  int u = pq.top().first;
  cout << "U = " << u << endl;

  pq.pop();
```

```cpp
  // Visit all of u's adjacent vertices. We get the adjacent vertex and store it in V
  cout << "-----------------------" << endl;
  // Iterate through all adjacent vertices V
  for (int i = 0; i < adjList[u].size(); i++)
  {

    int v = adjList[u][i].first;
    int weight = adjList[u][i].second;

    cout << "The Nodes adjacent to " << u << " are: \t";
    cout << "\t Node: " << v << "\t With Weight:" << weight << endl;

    // cout<<"With Weight:"<<weight<<endl;
    // If the distance to v is shorter by going through u

    if (dist[v] > dist[u] + weight)
    {
      // Update the distance of v.
      parent[v] = u;
      dist[v] = dist[u] + weight;

      // Insert v into the pq.

      pq.push(make_pair(v, dist[v]));
      dis[v] = dist[v];
    }
    else
    {
      dis[v] = dist[v];
    }
  }
}
printSPT(dis, n, parent);
return dist;
```

```
}

// Function to print the Shortest Path Tree

void PrintShortestPathTree(vector<int> &dist, int &source)
{

  cout << "\nPrinting the shortest path tree from node " << source << ".\n";
  for (int i = 0; i < dist.size(); i++)
  {
    cout << "The distance from node " << source << " to node " << i << " is: " << dist[i] << endl;
  }
}

int main()
{

  // Constructing the adjacency list that represents the graph.

  vector<vector<pair<int, int> > > adjList = FormAdjList();

  // Get a list of shortest path distances for node 0.

  int node = 0;
  vector<int> dist = Dijkstra(adjList, node);

  // Printing the SPT.
  PrintShortestPathTree(dist, node);

  return 0;
}
```

OutPut:

```
Getting the shortest path from 0 to all other nodes.
U = 0
-----------------------
The Nodes adjacent to 0 are:             Node: 1         With Weight:2
The Nodes adjacent to 0 are:             Node: 2         With Weight:3
U = 1
-----------------------
The Nodes adjacent to 1 are:             Node: 0         With Weight:2
The Nodes adjacent to 1 are:             Node: 1         With Weight:9
The Nodes adjacent to 1 are:             Node: 2         With Weight:4
The Nodes adjacent to 1 are:             Node: 3         With Weight:4
The Nodes adjacent to 1 are:             Node: 5         With Weight:1
U = 2
-----------------------
The Nodes adjacent to 2 are:             Node: 0         With Weight:3
The Nodes adjacent to 2 are:             Node: 1         With Weight:4
The Nodes adjacent to 2 are:             Node: 5         With Weight:2
U = 3
-----------------------
The Nodes adjacent to 3 are:             Node: 1         With Weight:4
The Nodes adjacent to 3 are:             Node: 4         With Weight:1
The Nodes adjacent to 3 are:             Node: 6         With Weight:2
The Nodes adjacent to 3 are:             Node: 8         With Weight:1
U = 4
-----------------------
The Nodes adjacent to 4 are:             Node: 3         With Weight:1
The Nodes adjacent to 4 are:             Node: 5         With Weight:2
The Nodes adjacent to 4 are:             Node: 6         With Weight:1
U = 5
-----------------------
The Nodes adjacent to 5 are:             Node: 1         With Weight:1
The Nodes adjacent to 5 are:             Node: 2         With Weight:2
The Nodes adjacent to 5 are:             Node: 4         With Weight:2
The Nodes adjacent to 5 are:             Node: 6         With Weight:7
The Nodes adjacent to 5 are:             Node: 7         With Weight:3
U = 4
-----------------------
The Nodes adjacent to 4 are:             Node: 3         With Weight:1
The Nodes adjacent to 4 are:             Node: 5         With Weight:2
The Nodes adjacent to 4 are:             Node: 6         With Weight:1
```

```
U = 6
-----------------------
The Nodes adjacent to 6 are:            Node: 3        With Weight:2
The Nodes adjacent to 6 are:            Node: 4        With Weight:1
The Nodes adjacent to 6 are:            Node: 5        With Weight:7
The Nodes adjacent to 6 are:            Node: 7        With Weight:4
The Nodes adjacent to 6 are:            Node: 9        With Weight:3
U = 6
-----------------------
The Nodes adjacent to 6 are:            Node: 3        With Weight:2
The Nodes adjacent to 6 are:            Node: 4        With Weight:1
The Nodes adjacent to 6 are:            Node: 5        With Weight:7
The Nodes adjacent to 6 are:            Node: 7        With Weight:4
The Nodes adjacent to 6 are:            Node: 9        With Weight:3
U = 7
-----------------------
The Nodes adjacent to 7 are:            Node: 5        With Weight:3
The Nodes adjacent to 7 are:            Node: 6        With Weight:4
The Nodes adjacent to 7 are:            Node: 9        With Weight:5
U = 8
-----------------------
The Nodes adjacent to 8 are:            Node: 3        With Weight:1
The Nodes adjacent to 8 are:            Node: 7        With Weight:9
The Nodes adjacent to 8 are:            Node: 9        With Weight:20
The Nodes adjacent to 8 are:            Node: 10       With Weight:11
U = 9
-----------------------
The Nodes adjacent to 9 are:            Node: 6        With Weight:3
The Nodes adjacent to 9 are:            Node: 7        With Weight:5
The Nodes adjacent to 9 are:            Node: 8        With Weight:20
The Nodes adjacent to 9 are:            Node: 10       With Weight:7
U = 10
-----------------------
The Nodes adjacent to 10 are:           Node: 8        With Weight:11
The Nodes adjacent to 10 are:           Node: 9        With Weight:7
U = 10
-----------------------
The Nodes adjacent to 10 are:           Node: 8        With Weight:11
The Nodes adjacent to 10 are:           Node: 9        With Weight:7
```

```
Printing the shortest path tree from node 0.
The distance from node 0 to node 0 is: 0
The distance from node 0 to node 1 is: 2
The distance from node 0 to node 2 is: 3
The distance from node 0 to node 3 is: 6
The distance from node 0 to node 4 is: 5
The distance from node 0 to node 5 is: 3
The distance from node 0 to node 6 is: 6
The distance from node 0 to node 7 is: 6
The distance from node 0 to node 8 is: 7
The distance from node 0 to node 9 is: 9
The distance from node 0 to node 10 is: 16
```

```
Vertex    Distance            Path
0-> 0          0               0
0-> 1          2               0 ->1
0-> 2          3               0 ->2
0-> 3          6               0 ->1 ->3
0-> 4          5               0 ->1 ->5 ->4
0-> 5          3               0 ->1 ->5
0-> 6          6               0 ->1 ->5 ->4 ->6
0-> 7          6               0 ->1 ->5 ->7
0-> 8          7               0 ->1 ->3 ->8
0-> 9          9               0 ->1 ->5 ->4 ->6 ->9
0-> 10         16              0 ->1 ->5 ->4 ->6 ->9 ->10
```

Conclusion:

Hence, the Dijkstra algorithm for finding the shortest path was implemented and the Shortest Path tree was printed.