COMP3600 Assignment 2

Name: Shiqin Huo
ID: u5949730

Question 1

```java
package MST;// Java program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
// weighted graph

import java.util.*;
import java.lang.*;

class MST {

    // Total number of vertices
    int N;
    // V-> no. of vertices & E->no.of edges
    int E;
    // collection of all edges
    Edge[] edges;

    /** Constructor for MST ----------------------------------------------------
     */
    MST(int n, int e) {
        N = n;  // total number of vertices
        E = e; // all edges
        edges = new Edge[E];
        for (int i = 0; i < e; i++) {
            edges[i] = new Edge(0,0,0);
        }
    }

    /** Inner Class for Vertices -----------------------------------------------
     */
    private static class Vertices{
        ArrayList<Position> pos = new ArrayList<>();
        class Position{
            float x;
            float y;
            int n;
            Position(int n, float x, float y){
                this.n = n;
                this.x = x;
                this.y = y;
            }
        }
        void addPos(int n){
            Random rand = new Random();
            float x = rand.nextFloat();
            float y = rand.nextFloat();
            Position p = new Position(n, x, y);
            pos.add(p);
        }
        public float getPosX(int index){
            return pos.get(index).x;
        }
        public float getPosY(int index){
            return pos.get(index).y;
        }
    }

    /** Inner Class for Edge ---------------------------------------------------
```

```java
 */
static class Edge implements Comparable<Edge> {
    int src, dest;
    double weight;

    public Edge(int src, int dest, double weight){
        this.dest = dest;
        this.src = src;
        this.weight = weight;
    }

    @Override
    public int compareTo(Edge b)
    {
        return this.weight > b.weight ? 1 : -1;
    }

    @Override
    public String toString(){
        return "( from  " + src  + " to " + dest + ", weight " + weight +" )";
    }

};

/** Inner Class for Union_Find ------------------------------------------------
 */
static class Disjoint_Set{
    private int[] root, rank;
    int num;

    Disjoint_Set(int num){
        rank = new int[num];
        root = new int[num];
        this.num = num;
        // initialization for this data structure
        init_set(num);
    }

    void init_set(int num){
        for (int i = 0; i<num; i++){
            // initialize the root to itself
            root[i] = i;
            rank[i] = 0;
        }
    }

    /** Path Compression technique --------------------------------------------
     */
    public int find_set(int a){
        if(root[a]!=a){
            root[a] = find_set(root[a]);
        }
        return root[a];
    }

    /** Union by Rank ---------------------------------------------------------
     */
    public void Union(int x,int y){
        int xroot = find_set(x);
        int yroot = find_set(y);
        if(xroot == yroot){
            return;
        }
        if (rank[xroot] < rank[yroot])
            root[xroot] = yroot;
            else if(rank[xroot] > rank[yroot])
                root[yroot] = xroot;

            else{
                root[yroot] = xroot;
                rank[xroot]++;
            }
        }
    }

    /** Generate MST by Kruskal's algorithm, return the average weight ------------
     */
    double KruskalMST()
    {
        // This will store the resulting MST
        Edge[] mst= new Edge[N];

        for (int i = 0; i< N-1; i++)
            // to store the final mst
            mst[i] = new Edge(0,0,0);

        // sorting the edges, num == N(N-1)/2 for complete graph
        Arrays.sort(edges);

        // Object for Disjoint_Set ------------------------------------------------
        Disjoint_Set sets = new Disjoint_Set(N);

        int i = 0;
        int e = 0;

        while (i < N-1) {

            Edge current_edge = edges[e];

            int xroot = sets.find_set(current_edge.src);
            int yroot = sets.find_set(current_edge.dest);

                // Cycle Detection ------------------------------------------------
            if (xroot != yroot) {
                // not cycle -> pick
                mst[i] = current_edge;
                sets.Union(xroot, yroot);
                i++;
            }
            e++;
            }


        double total_W = 0;
        for (int m = 0; m < i; m++) {
            total_W += mst[m].weight;}
        return total_W;
    }


    // Construct MST by Prim's algorithm, return the average weight

    double PrimMST(LinkedList<Integer>[] AdjList, double[][] w_matrix){
        // initial total weight
        double total_W = 0;

        // initialize a boolean array to record the traversal
        boolean[] visited = new boolean[N];

        // initialize the data structure to store possible edges for the current
step
```

```
        PriorityQueue<Edge> Q = new PriorityQueue();

        // mark the starting node as visited, assuming always start from the 0
vertex
        visited[0] = true;

        // traversal for neighbours of source 0 -----------------------------------
        for(int i : AdjList[0]){
            double w = w_matrix[0][i];
            Edge e = new Edge(0,0,0);
            e.dest = i;
            e.weight = w;
            // all adjacent edges added
            Q.add(e);
        }

        while( !Q.isEmpty()){
            // peek() to find the head of min_heap
            Edge e = Q.peek();

            if (visited[e.dest] && visited[e.src]){
                Q.poll();
                continue;
            }
            // else pick this edge e
            total_W += e.weight;

            Q.poll(); // remove the smallest edge

            if (!visited[e.dest]){
                visited[e.dest] = true;

                for(int nbrs_dest :AdjList[e.dest]){
                    int src = Math.min(nbrs_dest,e.dest);
                    int dest = Math.max(nbrs_dest,e.dest);
                    double w = w_matrix[src][dest];

                    if ((!visited[src] && visited[dest])||
(visited[src]&&!visited[dest])){
                        Edge e1 = new Edge(0,0,0);
                        e1.src = src;
                        e1.dest = dest;
                        e1.weight = w;

                        // all adjacent edges added to Queue
                        Q.add(e1);

                        visited[e.src] = true;
                        visited[e.dest] = true;
                    }
                }
            }
            else{
                visited[e.src] = true;

                for(int nbrs_dest :AdjList[e.src]){
                    int src = Math.min(nbrs_dest,e.src);
                    int dest = Math.max(nbrs_dest,e.src);
                    double w = w_matrix[src][dest];

                    if ((!visited[src] && visited[dest])||
(visited[src]&&!visited[dest])){
                        Edge e1 = new Edge(0,0,0);
                        e1.src = src;
                        e1.dest = dest;
                        e1.weight = w;
                    // all adjacent edges added
                    Q.add(e1);
                    visited[e.src] = true;
                    visited[e.dest] = true;
                }
            }
        }
    }

    // check whether all vertices are visited ---------------------------------
    for (int i = 1; i < N; i++){
        if (!visited[i]){
            return -1;
        }
    }

    return total_W;
}


/** Main Program ---------------------------------------------------------------
 */
public static void main (String[] args)
{
    double[] averageWeight_Q1a = new double[5];
    long[] averageRunTime_Q1a = new long[5];

    double[] K_averWeight_Q1c = new double[5];
    long[] K_averRunTime_Q1c = new long[5];

    double[] P_averWeight_Q1c = new double[5];
    long[] P_averRunTime_Q1c = new long[5];

    int sizes[] = {100,500,1000,5000};

    /** Q1_a Implementation begins---------------------------------------------
     */
    for(int s =0; s<sizes.length;s++){

        double testWeight_Q1a = 0.0;
        long testTime_Q1a = (long) 0.0;


        for (int d=0; d<50;d++) {
            int N = sizes[s]; // Input number of vertices
            int E = N * (N - 1) / 2; // Number of edges in complete graph
            MST graph = new MST(N, E);

            // Initialize the ArrayList of Points Objects
            Vertices V = new Vertices();

            // Assign the position for all vertices
            for (int k = 0; k < N; k++) {
                V.addPos(k);}

            int t = 0; // t_th edge
            double[][] w_matrix = new double[N][N]; // matrix to store

            // Q1_a Complete Graph Initialization
            // Fill the matrix to represent the graph
            for (int i = 0; i < N; i++) { // src = i
                for (int j = i; j < N; j++) { // dest = j
                    if (i == j) {
                        w_matrix[i][j] = 999;}
                    else {
                            graph.edges[t].src = i;
                            graph.edges[t].dest = j;
                            double w = Math.sqrt(Math.pow(V.getPosX(i) -
V.getPosX(j), 2) + Math.pow(V.getPosY(i) - V.getPosY(j), 2));
                            graph.edges[t].weight = w;
                            w_matrix[i][j] = w; // w_matrix[src][dest] = weight
                            t++;}
                        }
                    }

                    // Timing for Q1_a
                    long startTime_Q1a = System.nanoTime();
                    testWeight_Q1a+=graph.KruskalMST();
                    long endTime_Q1a = System.nanoTime();
                    testTime_Q1a +=(endTime_Q1a - startTime_Q1a);

                }

                averageWeight_Q1a[s] = testWeight_Q1a/50;
                averageRunTime_Q1a[s] =  testTime_Q1a/50;
                System.out.println("Complete Graph Size: "+ sizes[s] + " with an
average Weight: "+averageWeight_Q1a[s]);
                System.out.println("Complete Graph Size: "+ sizes[s] + " with an
average RunTime: "+averageRunTime_Q1a[s]);
            }

            /** Q1_c Implementation begins---------------------------------------------
            */

            for(int s = 0; s < sizes.length; s++){

                double K_testWeight_Q1c = 0.0;
                long K_testTime_Q1c = (long) 0.0;

                double P_testWeight_Q1c = 0.0;
                long P_testTime_Q1c = (long) 0.0;

                for (int d=0; d < 50; d++) { // repeated exp

                    int N = sizes[s];// int E = N * (N - 1) / 2; // Upper boundary of
the number of edges

                    // Initialize the ArrayList of Points Objects
                    Vertices V = new Vertices();

                    // Assign the position for all vertices
                    for (int k = 0; k < N; k++) {
                        V.addPos(k);
                    }

                    /** Q1_c Random Connected Graph Initialization---------------------
                     */

                    // Adjacency list representation of graph -------------------------
                    LinkedList<Integer>[] AdjList = new LinkedList[N];
                    ArrayList<Edge> pickedEdges = new ArrayList<Edge>();

                    for (int i = 0; i < N; i++) {
                        AdjList[i] = new LinkedList<>();
                    }

                    // Initialize flag boolean ----------------------------------------
                    Boolean unconnected = true;
                    // Initialize disjoint sets ---------------------------------------
                    Disjoint_Set sets = new Disjoint_Set(N);
                    while (unconnected) {
                        Random rand = new Random();
                        int node1_index = rand.nextInt(N);
                        int k = rand.nextInt(N);
                        while (k == node1_index) {
                            k = rand.nextInt(N); // random form  0 - (N-1)
                        }
                        int node2_index = k;

                        /** check the unique membership node2 in AdjList[node1] -------
                         */
                        boolean unique = true;
                        if (AdjList[node1_index].contains(node2_index)){
                            unique = false;
                        }

                        if (unique) { // if this edge didn't cover, new edge to process
                            AdjList[node1_index].add(node2_index);
                            AdjList[node2_index].add(node1_index);

                            sets.Union(node1_index, node2_index);

                            int min = Math.min(node1_index, node2_index);
                            int max = Math.max(node1_index, node2_index);
                            double w = Math.sqrt(Math.pow(V.getPosX(min) -
    V.getPosX(max), 2) + Math.pow(V.getPosY(min) - V.getPosY(max), 2));

                            Edge e = new Edge(min,max,w);
                            pickedEdges.add(e);

                            int first_root = sets.find_set(0);
                            int unconnected_src = 0;

                            /** traversal to check whether this graph is connected
                             */
                            for (int m = 1; m < AdjList.length; m++) {
                                // if there is one group root is not the first root -->
    unconnected --> repeated while loop
                                if (sets.find_set(m) != first_root) {
                                    unconnected_src++;
                                    // continue;
                                    // the graph is still unconnected
                                }
                            }
                            HashSet set = new HashSet();

                            for (int num = 0; num < AdjList.length; num++){
                                for (int t : AdjList[num]){
                                    set.add(t);
                                }
                            }
                            int numVisited = set.size();

                            if (unconnected_src == 0 && numVisited == N) {
                                // all src connected
                                unconnected = false; // break;
                            }
                        }
                        // if not unique, then it's duplicated random edge --> continue
    to next loop
                        else continue;
                    }

                    // Generating Random Connected Graph COMPLETED --------------------
                    /**
                     * Below gives the initialization of graph_c as a object of MST
    with size as pickedEdges, the ArrayList
                     * storing all edges.
                     */

                    double[][] w_matrix = new double[N][N]; // matrix to store
                    int E = pickedEdges.size();
                    MST graph_c = new MST(N, E);

                    for (int i = 0; i < N; i++) { // src = i
                        for (int j = i; j < N; j++) { // dest = j
                                w_matrix[i][j] = 999;
                        }
                    }
                    for(int e = 0; e < E; e++){
                        graph_c.edges[e].src = pickedEdges.get(e).src;
                        graph_c.edges[e].dest = pickedEdges.get(e).dest;
                        graph_c.edges[e].weight = pickedEdges.get(e).weight;
                        w_matrix[pickedEdges.get(e).src][pickedEdges.get(e).dest] =
    pickedEdges.get(e).weight;
                    }

                    // Timing for Q1_c (Kruskal's algorithm)
                    long K_startTime_Q1c = System.nanoTime();
                    K_testWeight_Q1c += graph_c.KruskalMST();
                    long K_endTime_Q1c = System.nanoTime();
                    K_testTime_Q1c += (K_endTime_Q1c - K_startTime_Q1c);

                    // Timing for Q1_c (Prim's algorithm)
                    long P_startTime_Q1c = System.nanoTime();
                    P_testWeight_Q1c += graph_c.PrimMST(AdjList, w_matrix);
                    long P_endTime_Q1c = System.nanoTime();
                    P_testTime_Q1c += (P_endTime_Q1c - P_startTime_Q1c);
            }

                K_averWeight_Q1c[s] = K_testWeight_Q1c/50;
                K_averRunTime_Q1c[s] =  K_testTime_Q1c/50;

                P_averWeight_Q1c[s] = P_testWeight_Q1c/50;
                P_averRunTime_Q1c[s] =  P_testTime_Q1c/50;

                System.out.println("Random Connected Graph Size: "+ sizes[s] + "
    Kruskal's average RunTime: "+ K_averRunTime_Q1c[s]);
                System.out.println("Random Connected Graph Size: "+ sizes[s] + "
    Kruskal's average Weight: "+ K_averWeight_Q1c[s]);

                System.out.println("Random Connected Graph Size: "+ sizes[s] + " Prim's
    average RunTime: "+ P_averRunTime_Q1c[s]);
                System.out.println("Random Connected Graph Size: "+ sizes[s] + " Prim's
    average Weight: "+ P_averWeight_Q1c[s]);
            }

        }
    }
```

(ii) The changing trend of the value of $\overline{L(n)}$ with the growth of n.

Running test for Q1_a:

```
Complete Graph Size: 100 with an average Weight: 6.766328812417876
Complete Graph Size: 100 with an average RunTime: 3044968 ns
Complete Graph Size: 500 with an average Weight: 14.745542598210426
Complete Graph Size: 500 with an average RunTime: 20506516 ns
Complete Graph Size: 1000 with an average Weight: 20.80189264506525
Complete Graph Size: 1000 with an average RunTime: 97888969 ns
Complete Graph Size: 5000 with an average Weight: 46.124602784258386
Complete Graph Size: 5000 with an average RunTime: 4409706667 ns
```

After observation and manual test, we found the formula below

$\overline{\mathrm{L(n)}} = 0.66\sqrt{\mathrm{n}}$

The table for input 4 points is shown below:

	x	y	Calculated y	Error
1.	100	6.766328812417876	6.6	-0.166
2.	500	14.745542598210426	14.75804	0.0125
3.	1000	20.80189264506525	20.8710325	0.0692
4.	5000	46.124602784258386	46.6690476	0.5444

It means the formula we get is very accurate, and the changing trend of the value of $\overline{L(n)}$ with the growth of n can be represented as: $\overline{L(n)} = 0.66\sqrt{n}$

General analysis:

http://www.cs.toronto.edu/~avner/papers/emst-full.pdf

We could consider this question from a statistical perspective. I did some related research online and found a related paper which indicates that the weight of such a Euclidean MST can be estimated with high confidence to use only $\tilde{O}(\sqrt{\mathrm{n}} \cdot \mathrm{poly}(1/\ \mathcal{E}))$ queries within 1+ $\mathcal{E}$. The proof relates to a lot of advanced discrete mathematics which I cannot fully digest, but it is obvious that the conclusion for this paper is consistent with the formula we found. The $\overline{\mathrm{L(n)}}$ generally has a positive correlation with the $\sqrt{\mathrm{n}}$. Rigorous mathematical proof refers to the paper that I attached a link above.

(iii) the running time trends of both algorithms with the growth of the value of n.

Running test for Q1_c

```
Random Connected Graph Size: 100 Kruskal's average RunTime: 136527 ns
Random Connected Graph Size: 100 Kruskal's average Weight: 31.013871506746117
Random Connected Graph Size: 100 Prim's average RunTime: 195843  ns
Random Connected Graph Size: 100 Prim's average Weight: 31.013871506746113
Random Connected Graph Size: 500 Kruskal's average RunTime: 355820 ns
Random Connected Graph Size: 500 Kruskal's average Weight: 135.05442260795914
Random Connected Graph Size: 500 Prim's average RunTime: 259204  ns
Random Connected Graph Size: 500 Prim's average Weight: 135.05442260795914
Random Connected Graph Size: 1000 Kruskal's average RunTime: 487349 ns
Random Connected Graph Size: 1000 Kruskal's average Weight: 245.53723172776728
Random Connected Graph Size: 1000 Prim's average RunTime: 587833 ns
Random Connected Graph Size: 1000 Prim's average Weight: 245.53723172776733
Random Connected Graph Size: 5000 Kruskal's average RunTime: 3177596 ns
Random Connected Graph Size: 5000 Kruskal's average Weight: 1101.2749453455524
Random Connected Graph Size: 5000 Prim's average RunTime: 6879204 ns
Random Connected Graph Size: 5000 Prim's average Weight: 1101.2749453455522
```

We calculate the Kruskal's and Prim's running time respectively:

By ***Online Polynomial Regression*** powered by http://www.xuru.org/rt/PR.asp#CopyPaste.

Kruskal's input:

100 136527

500 355820

1000 487349

5000 3177596

Result: $\overline{T(n)_k}$ **= $8.323701814\cdot10^{-5}$ n^3 - $4.500397846\cdot10^{-1}$ n^2 + 792.4528951 n + 61698.87132**

Residual Sum of Squares: **rss = 0** and Coefficient of Determination: **R^2 = 1**

Prim's input:

100 195843

500 259204

1000 587833

5000 6879204

Result: $\overline{T(n)_p}$ **= $-7.159604308\cdot10^{-5}$ n^3 + $6.688375578\cdot10^{-1}$ n^2 - 220.7052613 n + 211296.7466**

Residual Sum of Squares: **rss = 0** and Coefficient of Determination: **R^2 = 1**

Personal analysis:

The time complexity for Prim's algorithm is $O(|E|\lg|V|)$ while for Kruskal's algorithm is $O(E|\lg|E|)$ and as the graph is randomly generated, so it is more likely to be a sparse graph rather than the complete graph. But we could also find the order of magnitude is same since $O(|E|\lg|V|)$ = is $O(E|\lg|E|)$ since E is always less than V*(V-1)/2. However, in the real implementation, I adopt more complicated data structure to store those weight and neighbours for each edge and node. Then it takes more traversal to convert from one data structure to another one. For example, the AdjList is an array of LinkedList to store the neighbours information for each vertex. If we would like to check each adjacent edges, we should first detect the AdjList and according to its neighbours to locate the valid adjacent edges. Also, in Prim's we need to check the valid adjacent edges with one endpoint reached and another one unvisited. That's another reason why in my real impletation, there exist more iterations/traversal/loops. Therefore, it gives a possible excuse for the linear increase in runtime when compared to Kruskal's, but they are still in the same order of magnitude as they both have a $O(|E|\lg|V|)$ running time in general. Also, I applied adjacency matrix to record the weight, and it will increase the complexity of Prim's to O(|V|^2).

Question 2

(a) **Step 1: Characterize the structure of an optimal solution (the key observation)**

We denote the n characters string as S, then we use X[j] as the Boolean flag to represent whether the substring S [1…j] is a valid string of words. Then the question requires us to determine whether the X[n] is True.

$$\mathrm{X[j]} = \begin{cases} True, & if\ S[1\ldots j]\ is\ a\ valid\ word\ sequence \\ \qquad False, & \text{otherwise} \end{cases}$$

Step 2: Recurrence of the optimal solution

According to the procedures of dynamic programming, we should define the recurrence of X[j] with previous substring X[i], where there is i < j. We consider that if the X[j] is True which means that S[1…j] is a sequence of valid words and it requires that both S[1…i] and S[i+1…j] are valid word sequences.

Based on above reasoning we could construct the recurrence below:

$$\mathrm{X[j] = True\ iff\ there\ is\ i < k\ such\ that\ X[i]}\ is\ True\ and\ \mathrm{S[i+1..j]}\ is\ a\ valid\ word$$

Step 3: Algorithm for calculating the defined recurrence

```
Valid_Word_Sequence (s,n)
1.  define X[0…n-1] as an array of Booleans
2.  for j from 0 to n-1
3.      X[j] = false // initialization for all values in X[0…n-1]
4.      for i from 0 to j
5.          if (i==0 or X[i-1]) and b(s[i…j])
6.              X[j] = True
7.  return X[-1]
```

Step 4: Implementation for the above algorithm:

```python
def wordBreak(self, s, wordDict):
    """
    :type s: str
    :type wordDict: List[str] Note: wordDict works like "b(s)" for valid_word check
    :rtype: bool
    """
    b = [False for _ in range(len(s))]  # initialization
    for j in range(len(s)):
        for k in range(j + 1):
            if (k == 0 or b[k - 1]) and (s[k:j + 1] in wordDict):
                b[j] = True
    return b[-1]
```

We could simply add a stack to store the valid words in the string and if we need to know the broken words, we can reverse the stack and it will be the output.

Time Complexity: There only exists a nested loop: the outer one is from 0 to n-1 with n iterations, the inner one is from 0 to j with constraints to the outer loop which is definitely less than n iterations. Overall, the total time complexity is O(n^2)

(b) Firstly, we assume that w1 < w2 < ... < wk and n = |V|, e = |E|. Then an MST has n-1 edges with only k distinct weight. The total weight of MST is $\sum_{i=1}^{k} n_i \cdot w_i$ with the constraint that $\sum_{i=1}^{k} n_i = n - 1$. Denote n1 as the number of w1-weighted edges, n2 as the number of w2-weighted edges, similarly for all edges. We need to minimize the total weight of the generated MST. This question is equivalent to minimize $\sum_{i=1}^{k} n_i \cdot w_i$. Applying the idea of greedy algorithm, we would like to choose as many w1- weighted edges as possible, and then the second least weight. Meanwhile, we should avoid cycles due to the requirement of Tree.

Step 1: We consider the subgraph G1 = (V, E1), where E1 is a set of w1-weighted edges. We can easily find the G1 may not be connected, so we have to consider each connected component in it. We could simply apply DFS to generate a M1 set for all resulting spanning trees, served like a forest that M1 = {T1, T2, T3……TK}, where k is the number of connected components in G1 as each component can result in a spanning tree.

Step 2: We would regard k connected components in G1 as big vertices to construct a new graph G2 = (V2, E2) where E2 is a set of w2-weighted edges. Then we have to consider the loops and parallel edges with violates the definition of MST. Simply delete the cycles and parallel edges from G2 to generate a simple graph. We should have a memory to record each endpoints of the MST since we would backtrack to give an output of MST trace. Let M2 = { T1', T2', T3'……TK'}, where k' is the number of connected components in G2 as each component can result in a spanning tree.

Step 3: In each step, apply DFS to Gi = (Vi, Ei), and Vi are the connected components of G(i-1), Ei are the wi-weighted edges in G. Remove the redundant edges and store the original endpoints to backtrack and give out the resulting forest Mi = { T1i, T2i, T3i……TKi}

Step 4: Repeat above steps until the resulting forest is a single tree. The upper boundary for this procedure is at most k times. By Kruskal's algorithm, MST is generated as edges are selected by DFS traversal, and we could return the MST trace by the assistance of the memory.

Time Complexity: The running time for DFS algorithm is O(kn+m), and the initialization of Gi is O(k(n+m)) in total. Therefore, the total running time of the proposed algorithm is O(k(n+m)). Since k is a constant, we could conclude that this is a linear algorithm with O(n+m) running time.

Question 3

(a) **Proposed algorithm:**

Step 1: Firstly, we remove all edges with a 0 trustiness since it means they are blocked for these edges.

Step 2: Consider a source node s and the destination node t. We usually would like to divide the whole problem into several subparts. Assume P is the most trustable path from s to t consisting of (v0, v1), (v1, v2), (v2,v3), (v3,v4) … (v(k-1), vk) and we note that v0 = s and vk = t. Consider the intermediate node pi = p(v(i-1),vi), we should maximize the trust value trust(P) = p1*p2*p3….*pk, which means its reciprocal $\frac{1}{trust(P)}$ should be minimized. We apply logarithm function to the $\frac{1}{trust(P)}$

$$\log\left(\frac{1}{trust(P)}\right) = -\log(p_1) - \log(p_2) - \log(p_3) - \cdots - \log\,(p_k)$$

As we know, $-\log(p_i) \geq 0$ since $0 \leq p_i \leq 1$for all p_i as it represent a probability. Therefore, we could assign a weight w (u, v) = - log p(u, v) to each edge. Finally, we convert the original problem into a shortest path problem from s to t.

Step 3: We adopt Dijkstra's algorithm we learned from the lectures to find the shortest path from s to t. Also, we only need to find a shortest path to node t so we do not have to find all shortest paths to all others.

Step 4: Calculate the shortest path and the trust value. Assume we find L is the shortest length from s to t. If we found the L is infinity, it means there is no path from s to t. If L is finite value, the trust value can be calculated by the formula: $\mathbf{2^{-L}}$

(b) **Time Complexity**: As we can apply priority queue in this algorithm, specifically, the Min-Heap, the total complexity can be calculated by O(|E|log|V|+|V|log|V|), which gives a total O(|E|log|V|) running time.