Australian National University
Research School of Computer Science

COMP3600/COMP6466 in 2015 – Assignment Two

Due: 23:55 Monday, October 26
Late Penalty: 5% per working day

Submit your work electronically through Wattle. The total mark is 50. *Note that the mark you received from each question is proportional to the quality of your solution.*

Question 1 (25 points).

A complete graph is an undirected graph with an edge between each pair of vertices. A randomly weighted complete graph is a complete graph, in which each edge is assigned a weight that is a random real number uniformly distributed between 0 and 1. Let $L(n)$ be the expected (average) weighted sum of the edges in a minimum spanning tree (MST) of a randomly weighted complete graph G with n vertices. Your tasks are to

(i) Calculate $L(n)$, using Kruskal's algorithm when $n = 10, 100, 200, 500, 1,000$, respectively. **(10 points)**

Answer part (i):

```
/**************************************************************************
 *  Compilation:  javac KruskalMST.java
 *  Execution:    java KruskalMST
 *
 *  Compute a minimum spanning tree using Kruskal's algorithm on an undirected
 *  complete graph (Quick sort and disjoint set implementation come from
 *  http://algs4.cs.princeton.edu/code/).
 *  By Edward Xu
 **************************************************************************

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Random;

public class KruskalMST {

private double weight;  // weight of MST
    private ArrayList<Edge> mst = new ArrayList<Edge>();  // edges in MST
    /**
     * Compute a minimum spanning tree of an edge-weighted graph.
     * @param G the edge-weighted graph
     */
    public KruskalMST(double [][] G) {
        // more efficient to build heap by passing array of edges
     int edgeC = G.length * (G.length - 1)/2; // number of edges of a complete
     Comparable [] edges = new Comparable[edgeC];

     int k = 0;
     for (int i = 0; i < G.length; i ++){
     for (int j = i; j < G.length; j ++){
     if (i == j)
continue;
else {
Edge edge = new Edge(i, j, G[i][j]);
edges[k++] = edge;
}
     }
     }

     QuickSort.sort(edges);
        // run greedy algorithm
     DisjointSet ds = new DisjointSet(G.length);
     int k_ = edges.length; //number of unconsidered edges in edges.
        while ( (k_ > 0) && mst.size() < G.length - 1) {
         Edge e = (Edge) edges[k - k_]; k_ --;
            int v = e.either();
            int w = e.other(v);
            if (!ds.connected(v, w)) { // v-w does not create a cycle
             ds.union(v, w);  // merge v and w components
                mst.add(e);  // add edge e to mst
                weight += e.weight();
            }
        }
    }

    /**
     * Returns the edges in a minimum spanning tree.
     */
    public ArrayList<Edge> edges() {
        return mst;
    }

    /**
     * Returns the sum of the edge weights in a minimum spanning tree.
     */
    public double weight() {
        return weight;
    }

    /************************************************************************
  * Class members of MST
  *************************************************************************/
// randomly generate the undirected complete graph using a symmetric matrix
public static void generate(double [][] graph) {

int size = graph.length;

for (int i = 0; i < size; i++) {
for (int j = i ; j < size; j++) {
if (i == j)
graph[i][j] = 0;
else {
graph[i][j] = Math.random();
graph[j][i] = graph[i][j];
}
}
}
}

public static void main(String[] args) {

int sizes[] = { 10, 100, 150, 200 };

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

double[] averageWeight = new double[5];
long[] averageRunTime = new long[5];

int size = sizes[i];

for (int j = 0; j < 100; j++) {
double[][] graph = new double[size][size];
// randomly generate graph
KruskalMST.generate(graph);
//print(graph);
long startTime = System.nanoTime();
KruskalMST kruskalMST = new KruskalMST(graph);
averageWeight[i] += kruskalMST.weight() / 100;
long endTime = System.nanoTime();
averageRunTime[i] += ((endTime - startTime) / 100);
}
System.out.println("Size : " + size + "; Average Run Time : "
+ averageRunTime[i] + " ns ; Average Weight : "
+ averageWeight[i]);
}
}

// print a graph
public static void print(double [][] graph) {

int r = graph.length;
int c = graph[0].length;

DecimalFormat df = new DecimalFormat();
df.setMaximumFractionDigits(2);
df.setMinimumFractionDigits(2);

for (int i = 0; i < r; i++) {
for (int j = 0; j < c; j++) {
if ((c - 1) == j)
System.out.print(df.format(graph[i][j]) + "\n");
else
System.out.print(df.format(graph[i][j]) + ", ");
}
}
}


/*************************************************************************
 * Inner classes : (1) Quicksort (2) Disjoint set (3) Edge
 *************************************************************************/

private static class QuickSort {

private static Random random;    // pseudo-random number generator
    private static long seed;        // pseudo-random number generator seed
    // static initializer
    static {
        // this is how the seed was set in Java 1.4
        seed = System.currentTimeMillis();
        random = new Random(seed);
    }

    // This class should not be instantiated.
    private QuickSort() { }

    /**
     * Rearranges the array in ascending order, using the natural order.
     * @param a the array to be sorted
     */
    public static void sort(Comparable[] a) {
        shuffle(a);
        sort(a, 0, a.length - 1);
    }

    // quicksort the subarray from a[lo] to a[hi]
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }

    // partition the subarray a[lo..hi] so that a[lo..j-1] <= a[j] <= a[j+1..hi
    // and return the index j.
    private static int partition(Comparable[] a, int lo, int hi) {
        int i = lo;
        int j = hi + 1;
        Comparable v = a[lo];
        while (true) {

            // find item on lo to swap
            while (less(a[++i], v))
                if (i == hi) break;

            // find item on hi to swap
            while (less(v, a[--j]))
                if (j == lo) break;      // redundant since a[lo] acts as senti
            // check if pointers cross
            if (i >= j) break;

            exch(a, i, j);
        }

        // put partitioning item v at a[j]
        exch(a, lo, j);

        // now, a[lo .. j-1] <= a[j] <= a[j+1 .. hi]
        return j;
    }

   /***********************************************************************
    *  Helper sorting functions
    ***********************************************************************/

    // is v < w ?
    private static boolean less(Comparable v, Comparable w) {
        return (v.compareTo(w) < 0);
    }

    // exchange a[i] and a[j]
    private static void exch(Object[] a, int i, int j) {
        Object swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }

    /**
     * Rearrange the elements of an array in random order.
     */
    private static void shuffle(Object[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int r = i + uniform(N-i);     // between i and N-1
            Object temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }
```

```
/**
 * Returns an integer uniformly between 0 (inclusive) and N (exclusive).
 *
 * @throws IllegalArgumentException
 *             if <tt>N <= 0</tt>
 */
public static int uniform(int N) {
if (N <= 0) throw new IllegalArgumentException("Parameter N must be positive");
return random.nextInt(N);
}
}


private static class DisjointSet{

private int[] id;     // id[i] = parent of i
    private byte[] rank;  // rank[i] = rank of subtree rooted at i (cannot be m
    private int count;    // number of components

    /**
     * Initializes an empty union-find data structure with
     * isolated components 0 through N-1
     * @throws java.lang.IllegalArgumentException if N < 0
     * @param N the number of sites
     */
    public DisjointSet(int N) {
        if (N < 0) throw new IllegalArgumentException();
        count = N;
        id = new int[N];
        rank = new byte[N];
        for (int i = 0; i < N; i++) {
            id[i] = i;
            rank[i] = 0;
        }
    }

    /**
     * Returns the component identifier for the component containing site.
     * @param p the integer representing one object
     * @return the component identifier for the component containing site
     * @throws java.lang.IndexOutOfBoundsException unless 0 <= p < N
     */
    public int find(int p) {
        if (p < 0 || p >= id.length) throw new IndexOutOfBoundsException();
        while (p != id[p]) {
            id[p] = id[id[p]];    // path compression
            p = id[p];
        }
        return p;
    }

    /**
     * Returns the number of components.
     * @return the number of components (between 1 and N)
     */
    public int count() {
        return count;
    }

    /**
     * Are the two sites p and q in the same component?
     * @param p the integer representing one site
     * @param q the integer representing the other site
     * @return true if the two sites p and q are in the same component; false o
     * @throws java.lang.IndexOutOfBoundsException unless
     *      both 0 <= p < N and 0 <= q < N.
     */
    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    /**
     * Merges the component containing site p with the
     * the component containing site q.
     * @param p the integer representing one site
     * @param q the integer representing the other site
     * @throws java.lang.IndexOutOfBoundsException unless
     *      both 0 <= p < N and 0 <= q < N.
     */
    public void union(int p, int q) {
        int i = find(p);
        int j = find(q);
        if (i == j) return;

        // make root of smaller rank point to root of larger rank
        if      (rank[i] < rank[j]) id[i] = j;
```

```
            else if (rank[i] > rank[j]) id[j] = i;
            else {
                id[j] = i;
                rank[i]++;
            }
            count--;
        }
    }

    private class Edge implements Comparable<Edge> {
        private final int v;
        private final int w;
        private final double weight;

        public Edge(int v, int w, double weight) {
            if (v < 0) throw new IndexOutOfBoundsException("Vertex name must be a n
            if (w < 0) throw new IndexOutOfBoundsException("Vertex name must be a n
            this.v = v;
            this.w = w;
            this.weight = weight;
        }


        public double weight() {
            return weight;
        }

        /**
         * Returns either endpoint of the edge.
         */
        public int either() {
            return v;
        }

        /**
         * Returns the endpoint of the edge that is different from the given vertex
         * (unless the edge represents a self-loop in which case it returns the sam
         */
        public int other(int vertex) {
            if      (vertex == v) return w;
            else if (vertex == w) return v;
            else throw new IllegalArgumentException("Illegal endpoint");
        }
        /**
         * Compares two edges by weight.
         */
        public int compareTo(Edge that) {
            if      (this.weight() < that.weight()) return -1;
            else if (this.weight() > that.weight()) return +1;
            else                                    return  0;
        }

        /**
         * Returns a string representation of the edge.
         */
        public String toString() {
            return String.format("%d-%d %.5f", v, w, weight);
        }
    }
    }
```

(ii) The running time of of Kruskal's algorithm when $n = 10, 100, 200, 500, 1,000$, respectively. **(3 points)**

```
Size : 10; Average Run Time : 194680 ns ; Average Weight: 1.1287322698499318
Size : 100; Average Run Time : 1240392 ns ; Average Weight: 1.1831670933909968
Size : 200; Average Run Time : 4160379 ns ; Average Weight: 1.2082695445194218
Size : 500; Average Run Time : 46409460 ns ; Average Weight: 1.1980165214479213
Size : 1000; Average Run Time : 313305851 ns ; Average Weight: 1.200325578245647
```

(iii) Observe the changing trend of the value of $L(n)$ with the growth of n, and justify why this happens **(4 points)**.

Intuitively, the value of $L(n)$ should be roughly equal to $p(n-1)$ where p is the average weight of edges, i.e., $p = 1/2$. However, the actual value of $L(n)$ does not grow proportionally with the problem size. The reason behind this is that for the edges in an MST, their weights are less than that of non-tree edges. It turns out that the growth of $L(n)$ is very slow. In all experiments, we can observe that $L(n) \leq 1.3$.

There are a number of research papers on this topic. If you are interested to read about it, have a look at the following paper:

A. M. Frieze, *On the value of a random minimum spanning tree problem*, Discrete Applied Mathematics 10 (1985), pp. 47-56.

(iv) If the weight of each edge in the randomly weighted complete graph is a random real number uniformly distributed between 0 to 0.5 (instead of the original one between

0 and 1), calculate the expected weight $L'(n)$ of an MST in the random complete graph, using Kruskal's algorithm. Observe whether $L'(n)$ and $L(n)$ are significantly different for each $n \in \{10, 100, 200, 500, 1,000\}$, explain why this happens. **(8 points)**

```
Size : 10; Average Run Time : 192384 ns ; Average Weight: 0.5492284931275271
Size : 100; Average Run Time : 1241843 ns ; Average Weight: 0.6018658976089695
Size : 200; Average Run Time : 4151473 ns ; Average Weight: 0.5935923696723033
Size : 500; Average Run Time : 46087009 ns ; Average Weight: 0.6030187396770953
Size : 1000; Average Run Time : 315056440 ns ; Average Weight: 0.6010329728525269
```

The value range of $L'(n)$ is around 0.6, no more than 0.7. The explanation for this case is similar to the one for part (iii), that is, with the growth of n, there are more edges with weights less than the mean $\frac{0+0.5}{2} = 0.25$, while the edges in an MST is always the edges with smaller weight edges. The value difference between $L(n)$ and $L'(n)$ is the former is about twice the latter, the difference fluctuation is insignificant.

Question 2 (10 points).

Answer only **one** of the two questions Q2.(a) and Q2.(b), not both of them.

Q2.(a) Given an undirected connected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, assume that there are only constant numbers of distinct weights on its edges, i.e., for each edge $e \in E$, its weight is one of the values in $\{w_1, w_2, \ldots, w_k\}$, where $w_j > 0$ with $1 \leq j \leq k$ and k is a given integer. Devise an $O(n + m)$ time algorithm to find an MST in G. (10 points)

Let $n = |V|$ and $m = |E|$. As a minimum spanning tree contains $n - 1$ edges, the cost of an MST is $n_1 \cdot w_1 + n_2 \cdot w_2 + \ldots n_k w_k = \sum_{i=1}^{k} n_i \cdot w_i$ and $\sum_{i=1}^{k} n_i = n - 1$, where n_i is the number of edges with weight w_i. Assume that $w_1 < w_2 < \ldots < w_k$. We aim to minimize the cost of a spanning tree, that is, we aim to minimize $\sum_{i=1}^{k} n_i \cdot w_i$ over all spanning trees. We know from Kruskal's algorithm that the minimum cost occurs if we maximize n_1 by choosing as many edges as possible of weight w_1 to avoid cycles, then we maximize n_2, and so on. Here, we use the fact that Kruskal's algorithm correctly finds an MST, but we do not use Kruskal's algorithm itself as it takes more than $O(|V| + |E|)$ time.

Instead, we devise the following algorithm for this problem:

Construct a spanning subgraph $G_1 = (V, E_1)$ of $G = (V, E)$, where $E_1 = \{e \mid e \in E$ and $w(e) = w_1\}$. Note that G_1 may not be connected. Find a spanning subtree for each connected component in G_1, using DFS or BFS. Let $F_1 = \{T_1^{(1)}, T_2^{(1)}, \ldots, T_{N(1)}^{(1)}\}$ be the resulting forest, assuming that there are $N(1)$ connected components in G_1.

Next, apply DFS or BFS to the graph $G_2 = (V_2, E_2)$ whose vertices are the connected components of G_1 and whose edges are the edges of G with weight w_2. If this is a multigraph, that is, a graph with loops or possibly more than one edge between two vertices, delete edges to obtain a simple graph. Also, remember the original endpoints of the edges that are kept as they will be needed to reconstruct the output MST at the end of the algorithm. Let $F_2 = \{T_1^{(2)}, T_2^{(2)}, \ldots, T_{N(2)}^{(2)}\}$ be the resulting forest, assuming that there are $N(2)$ connected components in G_2.

In the i-th phase, apply DFS or BFS to graph $G_i = (V_i, E_i)$ whose vertices are the connected components of G_{i-1} and whose edges are the edges of G with weight w_i. If this is a multigraph, delete duplicate edges to obtain a simple graph. Again, remember that the original endpoints of the edges are kept. Let $F_i = \{T_1^{(i)}, T_2^{(i)}, \ldots, T_{N(i)}^{(i)}\}$ be the resulting forest, assuming that there are $N(i)$ connected components in G_i.

Continue this process until the obtained forest F_j contains only one tree. This occurs after at most k phases. It follows from the correctness of Kruskal's algorithm that the edges that were chosen to be tree edges by the DFS or BFS algorithm form an MST in G.

The total running time of the proposed algorithm is $O(k(n + m))$, including the running times of the DFS/BFS algorithms ($O(kn+m)$ in total) and the constructions of graphs G_i ($O(k(n+m))$ time in total). As k is a constant, the algorithm takes $O(n+m)$ time.

Q2.(b) Given a telecommunication network represented by a directed graph $G = (V, E)$, each link $(u, v) \in E$ has an associated value $p(u, v)$, which is a real number in the range $0 \le p(u, v) \le 1$ that represents the reliability of a communication channel from node u to node v. We interpret $p(u, v)$ as the probability that channel from u to v will not fail, and we assume that these probabilities are independent. Devise an efficient algorithm to find the most reliable path between two given vertices. Notice that the reliability of a path from node s to node t is usually not equal to the reliability of another path from node t to node s, assuming that both nodes s and t are in V. (10 points)

First of all, the algorithm we devise removes all edges whose reliability probability is 0 as such edges fail certainly.

For a given pair of nodes s and t, let P be a most reliable path from s to t which consists of edges $\langle v_0, v_1\rangle, \langle v_1, v_2\rangle, \ldots, \langle v_{i-1}, v_i\rangle, \ldots, \langle v_{k-1}, v_k\rangle$ where $v_0 = s$ and $v_k = t$. Let $p_i = p(v_{i-1}, v_i)$. Then, the reliability of P, $r(P) = p_1 \cdot p_2 \cdot \ldots \cdot p_k$, is maximized. In other words, the value of $\frac{1}{r(P)}$ is minimized. Equivalently, we can say that

$$\log \frac{1}{r(P)} = -\log p_1 - \log p_2 - \ldots - \log p_k$$

is minimized.

Note that $-\log p_j \ge 0$ as $0 \le p_j \le 1$ for all j with $1 \le j \le k$. Thus, given the network $G = (V, E)$, we assign each edge $\langle u, v\rangle$ an edge weight $w(u, v) = -\log p(u, v)$. Then, finding a most reliable path from s to t is equivalent to finding a shortest path from s to t in this graph with edge weight function w. A shortest path from s to t is found by applying Dijkstra's algorithm (note that we can terminate once we have found a shortest path to t as we are not interested in finding shortest paths to the other vertices). Let L be the length of a shortest path from s to t. If $L = \infty$, there is no directed path from s to t at all. If L is finite, then the reliability of a most reliable path from s to t is 2^{-L}.

The time complexity of this algorithm is $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$ if the minimum priority queue (MIN-HEAP) data structure is used.

Question 3 (COMP3600 15 points and COMP6466/Honours 10 points)

Answer only **one** of the following two questions, not all of them. Should you answer both of them, the lower mark will be counted as your received mark for this question.

Q3.(a) You are given a rectangular piece of cloth with dimensions $L \times W$, where L and W are positive integers, and a list of n products that can be made using the cloth. For each product i with $1 \leq i \leq n$, you know that a rectangle of cloth of dimensions $x_i \times y_i$ is needed and that the final selling price of the product is c_i. Assume that the x_i, y_i and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically.

Design an algorithm that determines the best return on the $L \times W$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired.

We start with a $X \cdot Y$ rectangle with $1 \leq X \leq L$ and $1 \leq Y \leq W$, the optimal solution involves making a vertical (or horizontal) cut, then we will have two new rectangles with dimensions $X \cdot Y_1$ and $X \cdot Y_2$ with $Y = Y_1 + Y_2$. As we want to maximize the profit, we need to maximize the profit $p(X, Y)$ on these new rectangles (optimal substructure). The recursion then is represented as follows.

$p(X, Y) = \max\{p(X, Y_1) + p(X, Y_2), p(X_1, Y) + p(X_2, Y)\}$ for all possible values of X_1 and X_2 for horizontal cut and Y_1 and Y_2 for vertical cut. Now the question is when do I actually decide to make a product? We can decide to make a product when one of its dimensions equals one of the dimensions of the current rectangle (why? Because if this doesn't hold, and the optimal solution includes making this product, then sooner or later we will need to make a vertical (or horizontal) cut and this case is already handled in the initial recursion), so we make the appropriate cut and we have a new rectangle $X \cdot Y_1$ (or $X_1 \cdot Y$), depending on the cut we made to obtain the product), in this case the recursion becomes $p(X, Y) = thecostofproduct + p(X_1, Y)$.

The solution of the original problem is $p(L, W)$. The running time of this DP solution would be O(L * W * (L + W + number of available products)): we have L * W possible rectangles, for each of these we try every possible cut (L + W) and we make one of the available products out of this rectangle.

Q3.(b) Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 46.4 Indian rupees, 1 Indian rupee buys 2.5 Japanese yen, and 1 Japanese yen buy 0.0091 U.S. dollars. Thus, by converting currencies, a trader can start with 1 U.S. dollar and buy $46.5 \times 2.5 \times 0.0091 = 1.0556$ U.S. dollars, thus turning a profit of 5.56 percent.

Suppose that we are given n different currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j, $1 \le i, j \le n$.

- Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that $R[i_1, i_2] \cdot R[i_2, i_3] \cdot \ldots \cdot R[i_k, i_1] > 1$ where $1 \le i_j \le n$ and $1 \le j \le k$. Analyze the running time of your algorithm.

We first construct a directed weighted graph $G = (V, E)$, each currency corresponds to a node in V, there is a directed edge from node $v_i \in V$ to node $v_j \in V$ which represents converting 1 unit of currency c_i to another currency c_j, the weight of this edge is $-\log R[i, j]$. Notice that the weight values of some edges in G may be negative (i.e., if $R[i, j] > 1$, then $-\log R[i, j] < 0$; otherwise, $-\log R[i, j] \ge 0$).

For a given pair of nodes v_a and v_b, let P be the most profitable conversion sequence from v_a to v_b which consists of edges $\langle v_0, v_1 \rangle, \langle v_1, v_2 \rangle, \ldots, \langle v_{i-1}, v_i \rangle, \ldots, \langle v_{k-1}, v_k \rangle$ where $a = 0$ and $b = k$. Let $r_i = R[v_{i-1}, v_i]$. Then, the amount $c(P)$ of currency c_j converted from one unit of currency c_i is $c(P) = r_1 \cdot r_2 \cdot \ldots \cdot r_k$. Maximizing this value is equivalently to minimizing the value of $\frac{1}{c(P)}$, or equivalently, minimizing the value of

$$\log \frac{1}{c(P)} = -\log r_1 - \log r_2 - \ldots - \log r_k.$$

Note that the value of $-\log r_j$ may be larger or smaller than zero for some j with $1 \le j \le k$. Thus, given the network $G(V, E)$, we assign each edge $\langle v_i, v_j \rangle$ from v_i to v_j a weight $w(v_i, v_j) = -\log R[i, j]$. Then, find all pairs of shortest paths in G, using Floyd-Warshall's algorithm.

Given two currencies c_{i_1} and c_{i_k}, let $l(P)$ be the length of the shortest path from v_{i_1} to v_{i_k}. To check whether there is a currency exchange sequence from c_{i_1} to c_{i_k} such that $R[i_1, i_2] \cdot R[i_2, i_3] \cdot \ldots \cdot R[i_k, i_1] > 1$. What we need is to check whether $\frac{1}{2^{l(P)}} \cdot R[i_k, i_1] > 1$, i.e., $R[i_1, i_2] \cdot R[i_2, i_3] \cdot \ldots \cdot R[i_k, i_1] > 1$.

The time complexity of this algorithm is $O(n^3)$ using the Floyd-Warshall's algorithm.

- Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

When calculating shortest paths, use a parent pointer to track back the parent of a node in the path. Then, you can print out the found sequence easily. Start with a node on the sequence and follow parent vertices going backwards until you arrive back to the same vertex. The time complexity for this is $O(n)$ as we visit every vertex of the sequence once.

Question 4 (COMP6466/Honours only, 5 points)

Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(|V| + |E|)$. (5 points)

Let $G = (V, E)$ be a DAG. To implement the proposed algorithm within $O(|V|+|E|)$ time, we maintain an array to store the in-degree of each node. We also have a stack to store in-degree zero nodes. We always remove the top node (e.g., r) from the stack until the stack becomes empty. Each time we update the in-degree d_u of node u by decreasing its value by 1 if (r, u) is an outgoing edge from r and r is the node just removed from the top of the stack (we refer to this operation as a relaxation operation on node r). If $d_u = 0$, add node u to the stack. This procedure continues until the stack becomes empty. As every node is removed from the stack exactly once, we perform such relaxations from each node once (with in-degree zero when performing the relaxation from it), and its time is linearly proportional to the number of outgoing edges of the node. Thus, the total number of edge and node relaxations for topology sort is $O(|E| + |V|)$, and so the total running time is also $O(|E| + |V|)$.

Bonus Questions (5 points)

`Warning:` *the following questions are designed for people who are capable of doing some extra research-related work. Only if you have finished all basic questions* **correctly** *and are willing to challenge yourself, you can proceed the questions.*

BQ.1: Given a 2-edge connected, weighted undirected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, we say that an edge $e \in E$ is *most vulnerable* if its removal will result in the maximum increase on the weighted sum of the edges in a minimum spanning tree of graph $G' = (V, E - \{e\})$. Devise an $O(n^2)$ time algorithm to identify a most vulnerable edge e in G. (A graph is 2-edge connected if it is still connected after the removal of any edge from it.) (2 points)

Let T be an MST of G, then, it is easy to show that a vulnerable edge in G must be in T. Thus, for each edge $e \in E(T)$, we examine the cost of the MST of the graph $G(V, E - \{e\})$, and identify one with the maximum cost. Then, the corresponding tree edge is the most vulnerable edge. The MST of graph $G(V, E - \{e\})$ can be easily built by removing the edge e from T, then there are two connected components derived by the edges in T which contain the two endpoints of e respectively. As G is 2-edge connected, there must have a non-tree edge $e' \in E - E(T)$, and if there are multiple such non-tree edges e', choose one with the minimum cost. Clearly $T' = T - \{e\} \cup \{e'\}$ is an MST of graph $G(V, E - \{e\})$.

Let F be a minimum spanning forest in graph $G = (V, E - E(T))$, where $E(F)$ is the set of tree edges in F, it is obvious that $|E(F)| \leq n - 1$. It can also be easily shown $e' \in E(F)$ by contradiction. Thus, for each $e \in E(T)$, it takes $O(|E(F)|) = O(n)$ time to find e'. The total amount of time to identify the vulnerable edge in T thus is $O((n-1) \cdot n + m + n \log n) = O(n^2)$.

BQ.2: Given a communication network represented by an undirected, weighted graph $G(V, E)$ and a non-negative integer delay bound $L > 0$, assume that for each edge $e \in E$, the communication cost on edge e is $c(e)$ and the communication delay incurred on it is $d(e)$, respectively, the delay-constrained shortest path problem in G is to find a routing path P from a source node $s \in V$ to a destination node $t \in V$ such that the total cost of links on path P is minimized (i.e., the value of $\sum_{e \in P} c(e)$ is minimized), while the accumulative delay of links on P is bounded by L (i.e., $\sum_{e \in P} d(e) \leq L$), for a pair of given nodes s and t. Devise an efficient algorithm in terms of running time for this delay-constrained shortest path problem, and analyze the time complexity of your algorithm. (2 points)

This is a constrained-shortest path problem, we can solve this problem by dynamic programming. Let $V = \{v_1, v_2, \ldots, v_n\}$ and let $p(v, l)$ be the cost of a shortest path from node s to node v with the total delay no more than l, then the problem can be formulated into a dynamic programming as follows.

$p(s, l) = 0$ for any integer l with $1 \leq l \leq L$,
$p(v, d(s, v)) = c(s, v)$ if $(s, v) \in E$ and $d(s, v) \leq L$; otherwise $p(v, l) = \infty$ for all l with $1 \leq l \leq L$ but $l \neq d(s, v)$;
$p(v, l) = \min_{u \in V}\{p(u, l') + c(u, v) \mid l' + d(u, v) \leq l \ \&\ (u, v) \in E\}$

Having the above recurrence, the problem is to compute $p(t, L)$, which can be calculated in $O(n^2 L)$ time.

BQ.3: Given a set of nodes V in a 2-dimensional plane, each node v_i has coordinates (x_i, y_i) for all i with $1 \leq i \leq |V|$. Let $U \subset V$ be a subset of V. Show that the cost of the minimum spanning tree of the subgraph $G[U]$ induced by the nodes in U is no more than twice the cost of the minimum spanning tree of graph $G[V]$, where $G[X]$ is a complete graph induced by the nodes in X, in which the weight of each edge (v_i, v_j) is the Euclidean distance between nodes v_i and v_j, i.e., $w(v_i, v_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, where $v_i \in X$ and $v_j \in X$. (2 points)

Let T be an MST of $G[V]$, choose any degree-1 node r as the tree root, perform tree traversal on T starting the leaf nodes, layer by layer from bottom up to top, let u be the node that currently is considered which is not in U. We need to remove u from the tree and modify the tree without the node u, assuming all the descendants of u in the tree are in U. If u is a leaf node, just remove it from the tree; otherwise, u is an internal node, let v be the parent of u and $w_{1,2}, \dots, w_k$ the children of u, clearly all w_js are in U by our assumption. We modify the tree as follows. We set the parent of w_1 as $p(u)$, and set the parent w_{i+1} as w_i for all i with $2 \le i \le k$. Thus, u is removed from the tree. Let T' be the new tree derived from a tree T, Then, the cost difference between T and T' is

$$\begin{aligned} C(T') - C(T) &= c(w_1, p(u)) + \sum_{i=1}^{k-1} c(w_i, w_{i+1}) + c(w_1, p(u)) - [c(u, p(u)) + \sum_{i=1}^{k} c(w_i, u)] \\ &\le [c(w_1, u) + c(u, p(u))] + \sum_{i=1}^{k-1} [(c(w_i, u) + c(w_{i+1}, u)] \\ &\le 2\sum_{i=1}^{k-1} c(w_i, u). \end{aligned} \tag{1}$$

In the end the resulting tree cost is at most twice the cost of the original tree. As the MST of $G[U]$ is the minimum cost tree, thus, its cost is no greater than the tree generated by the above steps. Thus, the MST of $G[U]$ is no greater than twice the cost of the MST of $G[V]$.

Approach two: Let T be an MST in G spanning the nodes in V, we traverse T by pre-order traversal and start from its root. Then, a cycle consisting of each tree edge twice is found. Denote by $c[V]$ the cost of the cycle, then

$$c[V] = 2 * c(T),$$

i.e., the cost of the cycle is twice the cost of the minimum spanning tree T.

Let $(u_1, \dots, u_2)$,$(u_2, \dots, u_3)$,..., $(u_{|U|-1}, \dots, u_{|U|})$, $(u_{|U|}, \dots, u_1)$ be the segments of the cycle with the first appearance of each $u_i \in U$ in the cycle starting from a node $u_1 \in U$, then the cycle is divided into $|U|$ segments, we now create another cycle consisting of only the nodes in U by connecting u_i and u_{i+1} for all i with $1 \le i \le |U| - 1$, and $u_{|U|}$ and u_1. Let $c[U]$ be the cost of this newly constructed cycle, by the triangle inequality, clearly

$$c[U] \le c[V] = 2 * c(T).$$

As the MST, $T[U]$, of $G[U]$ is the minimum cost tree connecting all nodes in U together, denote by $c(T[U])$ the cost of the MST, then $c(T[U]) < c[U] \le c[V] = 2 * c(T)$.