Australian National University
Research School of Computer Science

COMP3600/COMP6466 in 2016 –Solutions to Assignment Two

Due: 23:55 Monday, October 24
Late Penalty: 5% per working day, cutting off date October 30

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) sum of edge weights 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 both Prim's and Kruskal's algorithms when $n = 10, 100, 500, 1,000$ respectively. **(12 points)**

Answer to question 1(i):

```
/******************************************************************************
 *  Compilation:  javac MST.java
 *  Execution:    java MST
 *
 *  Compute a minimum spanning tree using Prim's algorithm on an undirected
 *  complete graph
 *  By Edward Xu
 ******************************************************************************

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

public class MST {

  /*****************************************************************
```

```
 * Inner classes : (1) MinHeap builds the min-heap and maintains the
 * min-heap-property (2) Each instance of Node is used to represent a node
 * in the min-heap.
 ************************************************************************/

private static class MinHeap {

  // build a min-heap on an array of comparable objects
  public static void build(Comparable[] array, int size) {
    for (int i = (size - 1) / 2; i >= 0; i--)
      min_heapify(array, i, size);
  }

  /************************************************************************
   * Helper functions to restore the heap invariant.
   ************************************************************************/

  // min-heapify which maintains the min-heapify-property
  public static void min_heapify(Comparable[] array, int i, int size) {

    int l = left(i);
    int r = right(i);

    int smallest = -1;
    if (l < size && less(array[l], array[i]))
      smallest = l;
    else
      smallest = i;

    if (r < size && less(array[r], array[smallest]))
      smallest = r;

    if (smallest != i) {
      exch(array, i, smallest);
      min_heapify(array, smallest, size);
    }

  }

  /************************************************************************
   * Helper functions for comparisons and swaps.
   ************************************************************************/
```

```
  private static int parent(int i) {
    return (i - 1) / 2;
  }

  private static int left(int i) {
    return 2 * i + 1;
  }

  private static int right(int i) {
    return 2 * i + 2;
  }

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

  private static void exch(Object[] arr, int i, int j) {
    Object swap = arr[i];
    arr[i] = arr[j];
    arr[j] = swap;
  }

  // returns -1 if obj does not exist, else return the index of obj.
  public static int contains(Comparable[] arr, Object obj, int heapSize) {

    for (int i = 0; i < heapSize; i++) {
      if (arr[i].equals(obj))
        return i;
    }
    return -1;
  }

}

// node in the min-heap
private class Node implements Comparable {

  public int id;
  public int pi;
  public double key;

  public Node(int id, int pi, double key) {
    this.id = id;
```

```
  this.pi = pi;
  this.key = key;
}

@Override
public int compareTo(Object anotherNode) throws ClassCastException {

  final int SMALL = -1;
    final int EQUAL = 0;
    final int LARGE = 1;

  if (!(anotherNode instanceof Node))
    throw new ClassCastException("A Node object expected!");

  if ( this == anotherNode ) return EQUAL;

  double anotherNodeKey = ((Node) anotherNode).key;

  if (this.key < anotherNodeKey) return SMALL;
  if (this.key > anotherNodeKey) return LARGE;

  return EQUAL;

}

@Override
public boolean equals(Object another) {
  // check for self-comparison
  if (this == another)
    return true;

  if (!(another instanceof Node))
    return false;

  // cast to native object is now safe
  Node that = (Node) another;

  // now a proper field-by-field evaluation can be made
  if (this.id == that.id)
    return true;
  else
    return false;
}
```

```
  @Override
  public String toString(){
    return this.id +"";
  }
}

/*************************************************************************
 * Class members of MST
 *************************************************************************/

// We use a hash table to map the nodes in the original graph and the nodes
// in the min-heap.
private Hashtable<Integer, Node> idToNode = new Hashtable<Integer, Node>();
// the weight of a minimum spanning tree
private double mstWeight = 0;

// the minimum spanning tree (you may not need this if you
// do not want to see the tree structure.)
private double[][] mst;

// find the adjacent nodes of the node with ID = id
private Object[] adjacents(double[][] graph, int id) {

  ArrayList<Node> adjs = new ArrayList<Node>();

  for (int i = 0; i < graph.length; i++) {
    if ((i != id) && (-1 != graph[id][i]))
      adjs.add(idToNode.get(i));
  }
  return adjs.toArray();
}

// extract the root of the min-heap (the node with minimum key)
private Node extract_min(Node[] array, int heapSize) {

  // exchange the root with the last element;

  MinHeap.exch(array, 0, heapSize - 1);

  Node minKeyNode = array[heapSize - 1];
  // delete the minimum node
  array[heapSize - 1] = null;
```

```
  // return the node with minimum key
  return minKeyNode;

}

public double prim(double[][] graph) {

  int graphSize = graph.length;

  // use an equal size matrix to represent the minimum spanning tree
  // and initialise all the elements to -1;
  this.mst = new double[graphSize][graphSize];
  for (int i = 0; i < graphSize; i++)
    for (int j = 0; j < graphSize; j++)
      this.mst[i][j] = -1;

  Object[] array = new Node[graphSize];
  for (int i = 0; i < graphSize; i++) {
    if (0 == i)
      array[i] = new Node(i, -1, 0);
    else
      array[i] = new Node(i, -1, Double.MAX_VALUE);

    idToNode.put(i, (Node) array[i]);
  }

  int heapSize = graphSize;

  // build a min-heap
  MinHeap.build((Node[]) array, heapSize);

  while ((heapSize = size(array)) > 0) {

    Node minKeyNode = extract_min((Node[]) array, heapSize);
    --heapSize;

    if (0 != minKeyNode.id) {// not the root node
      this.mst[minKeyNode.pi][minKeyNode.id] = minKeyNode.key;
      this.mst[minKeyNode.id][minKeyNode.pi] = minKeyNode.key;
    }

    this.mstWeight += minKeyNode.key;
```

```
    Object[] adjs = adjacents(graph, minKeyNode.id);

    for (Object nd : adjs) {
      Node node = (Node) nd;
      int nodeIndex = MinHeap.contains((Node[]) array, nd, heapSize);
      if ((-1 != nodeIndex)
          && (graph[minKeyNode.id][node.id] < node.key)) {

        node.key = graph[minKeyNode.id][node.id];
        node.pi = minKeyNode.id;
      }
    }
    MinHeap.build((Node[]) array, heapSize);
  }

  return this.mstWeight;
}

// 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, 50, 100, 150, 200 };
  //int sizes[] = { 50, 100, 150, 200, 250 };

  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
      MST.generate(graph);
      MST mst = new MST();

      long startTime = System.nanoTime();
      averageWeight[i] += mst.prim(graph) / 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]);
  }

}

/**************************************************************************
 * Helper functions for get sizes and print graphs.
 *************************************************************************/
// return the size of the min-heap
private int size(Object[] array) {
  int size = 0;

  for (int i = 0; i < array.length; i++) {
    if (null != array[i])
      size++;
  }

  return size;
}

// 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]) + ", ");
      }
    }
  }
}
```

```
/******************************************************************************
 *  Compilation:  javac KruskalMST.java
 *  Execution:    java KruskalMST
 *
 *  Compute a minimum spanning tree using Kruskal's algorithm on an undirected
 *  complete graph
 * .
 *  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..
// 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 sen

        // 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 posit
    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
    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
 * @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
        if (w < 0) throw new IndexOutOfBoundsException("Vertex name must be a
        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 vert
     * (unless the edge represents a self-loop in which case it returns the s
     */
    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) Observe the changing trend of the value of $L(n)$ with the growth of n, and explain why this happens. **(5 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.

(iii) The running times of both Prim's and Kruskal's algorithms when $n = 10, 100, 500, 1,000$ respectively. **(4 points)**

We run both algorithms on a machine with a 4 GHz Intel i7 Quad-core CPU and 32 GiB RAM.

The running time of Prim's algorithm when $n = 10, 100, 500, 1,000$.

```
Size : 10; Average Run Time : 52054 ns ; Average Weight : 1.1368859096614992
Size : 100; Average Run Time : 887823 ns ; Average Weight : 1.2081159088499633
Size : 150; Average Run Time : 1512130 ns ; Average Weight : 1.234243808703075
Size : 200; Average Run Time : 2874311 ns ; Average Weight : 1.1977441071731987
```

The running times of of Kruskal's algorithm when $n = 10, 100, 500, 1,000$.

```
Size : 10; Average Run Time : 84022 ns ; Average Weight : 1.1119954851870812
Size : 50; Average Run Time : 560706 ns ; Average Weight : 1.1887011658383622
Size : 100; Average Run Time : 1160104 ns ; Average Weight : 1.2077029857509862
Size : 150; Average Run Time : 2367202 ns ; Average Weight : 1.1936974701780239
Size : 200; Average Run Time : 2671743 ns ; Average Weight : 1.214796126027724
```

(iv) With the growth on the number of vertices n, observe the running time trends of both Prim's and Kruskal's algorithms, whether they grow at the same speed, or one is growing much faster than the other one, and explain why that happens (**4 points**)

It clearly shows from the experimental results that the running time of Prim's algorithm is much faster than that of Kruskal's algorithm, as the dominant time in Kruskal's algorithm is sorting, and there are $O(n^2)$ elements (edges) to be sorted, while Prim's algorithm does not need sorting.

Your program should have the following properties.

1. Do not read any input. Write one line of output for each n, and at most 5 extra lines of explanatory output.
2. Store the graph as a symmetric matrix containing the edge weights.
3. Implement Kruskal's algorithm, using subroutines of Quick Sort and Directed Forest for disjoint sets, you need to implement both of these two subroutines, rather than calling them from the program language library.
4. For measuring running time, refer to its usage in the exercise of Lab01.
5. (a) If you use C, write the program as a single file `mst.c` which will compile on Linux with the command

```
gcc -o mst mst.c -lm
```

("**-lm**" selects the maths library; you might not need it). Use real random numbers (type **double**) generated using the library function **rand()** whose use is illustrated as follows.

```
/* This program generates 10 random double numbers in (0,1).
 * Note that rand() returns an integer in the range [0,RAND_MAX],
 * so you need performing some conversion to double and scaling.*/
```

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    double x;
    int i;

    for (i = 0; i < 10; ++i)
    {
        x =  (rand()+0.5)/(RAND_MAX+1.0);
        printf("%11.8f\n",x);
    }
    exit(0);
}
```

(b) If you use Java, write the program as a single file `mst.java` which will compile on Linux with the command

```
javac mst.java
```

Use real random numbers (type **double**) generated using the library function **Math.random()** whose use is illustrated as follows.

```
// This program generates 10 random double numbers in (0,1).
//

class dran
{
   public static void main(String[] args)
   {
      double x;

      for (int i = 0; i < 10; i++)
      {
          x = Math.random();
          System.out.println(x);
      }
   }
}
```

What to submit:

Upload your file `mst.c` or `mst.java` containing the program to Wattle. Don't submit both these files, just one of them.

You also need to submit a report in *the PDF format* that contains the source code and its outputs, and the answers to part (i), part (ii), part (iii), and part (iv) of the question.

Question 2 (**10 points**).

Answer any **two** of the three questions Q2.(a), Q2(b), and Q2.(c). Should you answer all of them, the lower mark ones will be counted as your received mark for this question.

Q2.(a) Suppose a CS curriculum consists of n courses, all of them mandatory. The prerequisite graph G has a node for each course, and an edge from course u to course v if and only if u is a prerequisite for v.

Devise *a linear running time* algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum (assume that a student can take any number of courses in one semester). (**5 points**)

We first construct a directed graph $G = (V, E)$, where each course corresponds to a node v in V, there is a directed edge from node $u \in V$ to node $v \in V$ if and only if u is a prerequisite for v, and each edge is assigned a weight of one. It can be seen that G is directed acyclic graph (DAG), otherwise, we can derive that course u is a prerequisite of itself.

We then construct another DAG based on G. That is, add a virtual source node s and a virtual destination node t to G, and add a directed edge from s to every node with an incoming degree of zero and from each node with an outgoing degree of zero to node t, and assign each of such edges with weight 0. Let G' be the result graph.

The proposed algorithm is detailed in Algorithm 1.

Then, the value of $t.d$ is the minimum number of semesters needed. The length $\Delta(s, t)$ $(= u.t)$ of the longest path in G' from s to t is the minimum number of semesters to enroll in order to graduate.

The time complexity of the algorithm is analyzed as follows.

The construction of G' takes $O(|E| + |V|)$ time, the computation of topological rank of each node in G' takes $O(|E|+|V|)$ time as well due to the fact that performing DFS on G' needs $O(|E|+|V|)$ time. The relaxation of all edges according to the node topological ranking takes $O(|E| + |V|)$ time. Thus, the algorithm takes $O(|E| + |V|)$ time to find the minimum number of semesters to enrol for the graduation.

Q2.(b) Given a telecommunication network represented by a directed graph $G = (V, E)$,

Algorithm 1 compute the minimum number of semesters to finish a degree

Input: An acyclic directed graph $G = (V, E)$;

Output: a sequence of courses to be chosen in each semester of the minimum number of semesters to finish the degree.

1: Construct $G' = (V \cup \{s, t\}, E \cup \{\langle s, v\rangle \mid v \in V$ and the incoming degree of v in G is zero $\} \cup \{\langle v, t\rangle \mid v \in V$ and the outgoing degree of v in G is zero $\})$;
2: Compute the topological rank of each node in G' by performing DFS traversal on G' starting in from node s;
3: List the nodes in increasing oder of their topological ranks. Without loss of generality, let $s, v_1, v_2, \ldots, v_n, t$ be the sorted node sequence, and $v_i \in V$.
4: /* compute a longest path in G' from s to t */
5: $s.d \leftarrow 0$; /* the estimate of the longest distance from s to itself */
6: **for** every $v \in V \setminus \{s\}$ **do**
7: $v.d \leftarrow \infty$; /* the longest distance from s to v */
8: **end for**;
9: **for** each i with $1 \leq i \leq n$ **do**
10: **for** $u \in Adj(v_i)$ **do**
11: $Relax(v_i, u, w)$ /* perform the relaxation if the distance of $u.d$ increases */;
12: **if** node u is relaxed **then**
13: $\pi(u) \leftarrow v_i$ /* v_i becomes the predecessor of u in the longest path*/;
14: **end if**;
15: **end for**;
16: **end for**

each link $(u, v) \in E$ has an associated value $p(u, v)$, which is a real number in the range $0 \leq p(u, v) \leq 1$ 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. (**5 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 $-\log p_j \geq 0$ as $0 \leq p_j \leq 1$ for all j with $1 \leq j \leq k$. 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.

Q2.(c) Given an undirected connected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, assume that there are only three distinct weights on its edges, i.e., for each edge $e \in E$, its weight is one of the values in $\{w_1, w_2, w_3\}$, where $w_j > 0$ with $1 \leq j \leq 3$.

Devise an $O(n+m)$ time algorithm to find a minimum spanning tree in G. (**5 points**)

Let $n = |V|$ and $m = |E|$. As a minimum spanning tree contains $n - 1$ edges, the cost of an MST is $n_1 \cdot w_1 + n_2 \cdot w_2 + n_3 w_3 = \sum_{i=1}^{3} n_i \cdot w_i$ and $\sum_{i=1}^{3} n_i = n - 1$, where

n_i is the number of edges with weight w_i. Without loss of generality, we assume that $w_1 < w_2 < w_3$. We aim to minimize the cost of a spanning tree, that is, we aim to minimize $\sum_{i=1}^{3} n_i \cdot w_i$ 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 $E_1 = \{e \mid e \in E \text{ and } w(e) = w_1\}$. Note that G_1 may not be connected. Find a spanning subtree for each connected component in G_1, using the DFS or BFS technique. 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 (i.e., a graph with loops or possibly more than one edge between two vertices), delete multiple edges by keeping only one of them 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 ith phase, apply DFS or BFS to the 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 multiple edges by keeping only one of them to obtain a simple graph. Again, remember the original endpoints of the edges that 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, $1 \leq i \leq 3$.

Continue this process until the obtained forest F_i is a tree. This occurs at most three phases. It follows from the correctness of Kruskal's algorithm that the edges chosen to be the tree edges by our DFS or BFS algorithms will form an MST in G.

The total running time is $O(3(|V|+|E|))$, including the running times of the DFS/BFS algorithms ($O(3(|V|+|E|)) = O(|V|+|E|)$ in total) and the constructions of the graphs G_i ($O(3(|V| + |E|))$ time in total). The algorithm thus takes $O(|V| + |E|)$ time.

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

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) A mission-critical production system has n stages that have to be performed sequentially; stage i is performed by machine M_i. Each machine m_i has a probability r_i of functioning reliably and a probability $1 - r_i$ of failing (and the failures are independent).

Therefore, if we implement each stage with a single machine, the probability that the while system works is $r_1 \cdot r_2 \cdot \ldots r_n$. To improve this probability, we add redundancy by having m_i (≥ 1) copies of the machine M_i that perform in stage i. The probability that all m_i copies fail simultaneously is only $(1-r_i)^{m_i}$, so the probability that stage i is completed correctly is $1-(1-r_i)^{m_i}$ and the probability that the whole system works is $\Pi_{i=1}^{n}(1-(1-r_i)^{m_i})$. Each machine M_i has a cost c_i, and there is a total budget B to buy machines, assuming that both B and c_i are positive integers.

Given the probabilities $r_1, r_2, \ldots, r_n$, the costs $c_1, c_2, \ldots, c_n$, and the budget B, find the redundancies $m_1, m_2, \ldots, m_n$ that are within the available budget and that maximize the probability that the system works correctly.

- Devise an efficient algorithm to determine the values of $m_1, m_2, \ldots, m_n$ under the given budget B such that the probability that the system works correctly is maximized. (**12 points for COMP3600 and 8 points for COMP6466/Honours**)

  Notice that the maximum success probability would be the minimum failed probability, and each machine should have at least one copy, i.e., $m_i \geq 1$ for all $1 \leq i \leq n$. Let $R(b, j)$ be the maximum reliability for constructing the system through stages $1, \ldots, j$ with budget $b \geq 0$. Note that if $b < \sum_{i=1}^{j} c_i$, we have $R(b, j) = 0$ for all j with $1 \leq j \leq n$. Following this, we have the following recurrence

$$R(b,j) = \begin{cases} 1 & \text{if } j = 0, \\ \max_{(\sum_{i=1}^{j-1} c_i) \leq b' \leq B - c_j} \{R(b', j-1) * (1-(1-r_j)^{\lfloor \frac{b-b'}{c_j} \rfloor})\} & \text{otherwise} \end{cases}$$

  where b' is the residual budget after having $\lceil \frac{b-b'}{c_j} \rceil$ copies of machines in the j-th stage. Note that by convention, $\max \emptyset = 0$, thus if there does not exist a b' that meets the constraints, $R(b, j) = \max \emptyset = 0$.

  The calculation of $R(B, n)$ is given by Algorithm 2 as follows.

  The final result is $R(B, n)$ which is the maximum reliability of the system and m_i is the number of machine M_i for all i with $1 \leq i \leq n$.

- Analyze the running time of your algorithm. (**3 points for COMP3600 and 2 points for COMP6466/Honours**)

  Following Algorithm 2, The dominant running time of the algorithm is the nested for loops which is $O(nB)$.

Q3.(b) A certain string-processing language offers a primitive operation which splits a string into two pieces. Since this operation involves copying the original string, it takes n units of time for a string of length n, regardless of the location of the cut. Suppose, now that you want to break a string into many pieces. The order in which the breaks are

Algorithm 2 compute the most reliable system given a budget B

Input: A set of n machines where each machine M_i has a reliability r_i with cost c_i; and the total budget B

Output: The number of copies m_i (≥ 1) of each machine M_i such that the system reliability is maximized

1: **for** $j \leftarrow 1$ to n **do**
2: **for** $b \leftarrow 0$ to $(\sum_{i=1}^{j} c_i) - 1$ **do**
3: $R(b, j) \leftarrow 0$; /* initialization */
4: **end for**;
5: $R(\sum_{i=1}^{j} c_i, j) \leftarrow \Pi_{i=1}^{j} r_i$; /* initial value*/
6: $m_j \leftarrow 1$; /* numbers of machine M_i */
7: **end for**;
8: **if** $B < \sum_{i=1}^{n} c_i$ **then**
9: no solution; EXIT;
10: **end if**;
11: **for** $j \leftarrow 2$ to n **do**
12: $R(B, j) \leftarrow \Pi_{i=1}^{j} r_i$; /* initial value*/
13: **for** $b' \leftarrow (\sum_{i=1}^{j-1} c_i) + 1$ to $B - c_j$ **do**
14: $temp \leftarrow R(b', j-1) * (1 - (1 - r_j))^{\lfloor \frac{B-b'}{c_j} \rfloor}$;
15: **if** $temp > R(B, j)$ **then**
16: $R(B, j) \leftarrow temp$;
17: $m_j \leftarrow \lfloor \frac{B-b'}{c_j} \rfloor$;
18: **end if**;
19: **end for**;
20: **end for**

made can affect the total running time. For example, if you want to cut a 20-character string at position 3 and 10, then making the first cut at position 3 incurs a total cost 20+17=37, while doing position 10 first has a better cost of 20+10=30.

- Devise a dynamic programming algorithm that, given the locations of m cuts in a string of length n ($m < n$), find the minimum cost of breaking the string into $m+1$ pieces. (**12 points for COMP3600 and 8 points for COMP6466/Honours**)

  Following the design paradigm of DP, we characterize the optimal substructure of the string-splitting problem. We index the pieces of the string from left to right as 1, 2, . . . , m + 1, where each neighboring pair of pieces is separated by one of the supplied cuts.

  Consider a solution which initially cuts pieces i through j just after piece k with $i \leq k < j$. This will result in two sub-pieces that consist of the original pieces i through k and $k+1$ through j. The cost of the optimal solution then can defined recursively. Let $c(i, j)$ be the cost of an optimal solution for partitioning pieces i through j with $1 \leq i \leq j \leq m+1$, and let I be the index array of cut locations, where $I[i] = k$ implies that the ith cut location is just after the character indexed by k (assuming characters in the original string are indexed from 1 to n). For the sake of convenience, we may set $I[0] = 0$ and $I[m+1] = n$. We then have

$$c(i,j) = \min_{i \leq k < j} \begin{cases} 0 & \text{if } i = j \\ I[j] - I[i] + \{c(i,k) + c(k,j)\} & \text{otherwise,} \end{cases}$$

  where $I[j]$- $I[i]$ is the length of pieces from i to j, as given by their corresponding cut locations. The optimal solution for the string cutting problem thus will be $c(1, m+1)$, which is the minimum cost of dividing pieces 1 through $m+1$, using the given cut locations.

  We then devise an algorithm to compute $c(1, m+1)$. The detailed algorithm, Algorithm 3, is described as follows.

- Analyze the running time of your algorithm. (**3 points for COMP3600 and 2 points for COMP6466/Honours**)

  The initialization takes $O(m^2)$ time (the dominant step 3). The amount time for the nested loops from step 4 to step 15 is calculated as follows.
  $\sum_{i=m+1}^{1} \sum_{j=i}^{m+1} O(j-i) = \sum_{i=m+1}^{1} O(m+1-i) = \sum_{i=m+1}^{1} O(m^2) = O(m^3)$.
  The time complexity of the proposed algorithm thus is $O(m^3)$.

Question 4 (**5 points for COMP6466/Honours only**)

A $d-$dimensional box with dimensions $(x_1, x_2, \ldots, x_d)$ *fits inside* another box with dimensions $(y_1, y_2, \ldots, y_d)$ if there exists a permutation π on $\{1, 2, \ldots, d\}$ such that $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

- Describe an efficient method to determine whether or not one d-dimensional box fits inside another. (**2 points**)

  Given two boxes B_i and B_j, we first sort their dimensions in increasing order, let $(x_1, x_2, \ldots, x_d)$ for B_i and $(y_1, y_2, \ldots, y_d)$ for B_j, then check whether their corresponding dimension $x_l < y_l$ for all l with $1 \leq l \leq d$, if yes, B_i fits inside B_j.

- Suppose that you are given a set of n $d-$dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Describe an efficient algorithm to determine the longest sequence $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$ of boxes such that B_{i_j} fits inside $B_{i_{j+1}}$ for $j = 1, 2, \ldots, k-1$ and $1 \leq i_j \leq n$. Express the running time of your algorithm in terms of n and d. (**3 points**)

  For each box B_i, there is a node $v_i \in V$, if B_i fits inside B_j, there is a directed edge in E from v_i to v_j and the weight $w(v_i, v_j) = 1$. Thus, a directed acyclic graph $G = (V, E)$ is constructed. Then, the longest sequence of boxes in the original problem now becomes to find a longest path in G.

  Let $L(j)$ be the length of the box fitting inside sequence where box j is the last box in this sequence. Then,

  $L(j) = \max_{(v_i, v_j) \in E}\{L(i) + 1\}$.

  The length of the longest box fitting inside sequence is $L(j_0) = \max\{L(j) \mid 1 \leq j \leq n\}$.

  The running time is analyzed as follows. Given two boxes, checking their fitting inside relationship takes $O(d \log d)$ time. The construction of the DAG G takes $O(n^2 d \log d)$ time. Finding all $L(j)$ and the maximum value among $L(j)$ takes $O(n^2)$ time. Thus, the algorithm takes $O(d \log dn^2)$ time in total.

  The above solution is a dynamic programming solution. Alternatively, in the following we propose a greedy algorithm for the problem.

  We construct a directed acyclic graph $G' = (V \cup \{v_0\}, E \cup \{\langle v_0, v_i \rangle \mid v_i \in V\})$ and assign each edge from v_0 to v_i with weight 0. Let $d(i)$ be the longest path from v_0 to $v_i \in V$. Apply Dijkstra's algorithm for finding a single-source (at v_0) longest path tree including all node in V. We only need to modify the Dijkstra algorithm a bit. The Dijkstra algorithm consists of number of iterations, Within each iteration, it picks a node v with the minimum distance estimate $d(v)$, adds the node to the shortest path tree, and updates the minimum distance estimate of every other node u that has a directed edge from v to it. What we do here is to modify this step. That is, it picks a node v with the maximum distance estimate $d(v)$, adds the node to the longest path tree, and updates the maximum distance estimate of every other node u that has a directed edge from v to it.

The time complexity analysis is as follows. The construction of G' takes $O(d \log dn^2)$ time, the Dijkstra algorithm takes $O(m + n \log n) = O(n^2 + n \log n)$ time. Thus, the algorithm takes $O(d \log dn^2)$ time.

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 3-edge connected, weighted undirected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, we say that two edges $e_1 \in E$ and $e_2 \in E$ are *most vulnerable* if their removal will result in the maximum increase on the weighted sum of the edges in a minimum spanning tree of graph $G' = (V, E - \{e_1, e_2\})$.

Devise an $O(n^3)$ time algorithm to identify these two most vulnerable edges e_1 and e_2 in G. (A graph is 3-edge connected if it is still connected after the removal of any two edges from it.) (**3 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 3-edge connected, there must have a non-tree edge $e' \in E \setminus E(T)$, and if there are multiple such non-tree edges e', choose one with the minimum cost. Clearly, $T' = T \setminus \{e\} \cup \{e'\}$ is an MST of graph $G(V, E \setminus \{e\})$.

Let F_1 and F_2 be the minimum spanning forests in graphs $G_1 = (V, E \setminus E(T))$ and $G_2 = (V, E \setminus (E(T) \cup E(F_1)))$, respectively, where $E(T)$ and $E(F_1)$ are the sets of tree edges in T and F_1, respectively, it is obvious that $|E(F_i)| \leq n - 1$ with $i = 1, 2$. It can also be easily shown $e' \in \cup_{i=1}^{2} E(F_i)$ by contradiction. Thus, for each pair of two tree edges e_1 and e_2 with $e_i \in E(T)$ and $i = 1, 2$, it takes $O(|\cup_{i=1}^{2} E(F_i)|) = O(n)$ time to find e'_1 and e'_2. In other words, removing e_1 and e_2 from T results in three connected components, finding two edges e'_1 and e'_2 by connecting these three connected components into a minimum spanning tree takes $O(|\cup_{i=1}^{2} E(F_i)|) = O(n)$ time. There are $O(n^2)$ pairs of tree edges to be considered. Thus, the total amount of time to identify the two most vulnerable edges in T is $O(n^2 \cdot n + m + n \log n)) = O(n^3)$ since there are $(n - 1)(n - 2)$ pairs of edges to be checked.

BQ.2: Given a directed weighted graph $G = (V, E)$, assume that the shortest path

between any two vertices in V contains no more than k edges. Devise an $O(k|E|)$ time algorithm to find a shortest path between a pair of vertices $u \in V$ and $v \in V$. (**2 points**)

Modify the Bellman-Ford algorithm with source u to perform k phases only.

That is, assign 0 to u and ∞ to every other vertex initially, then perform k phases, where each phase consists of relaxing all the edges once. Clearly, this algorithm takes $O(k|E|)$ time.

To show that it is correct, consider a shortest path $u = v_0 \to v_1 \to v_2 \to \ldots \to v_{k-1} \to v_k = v$ from u to v with no more than k edges. Since a subpath of a shortest path is also a shortest path, $u \to v_1$ is a shortest path from u to v_1. Thus, after the first phase, the value $v_1.d$ assigned to v_1 (upper bound on the length of a shortest path from u) is equal to $\delta(u, v_1)$ (length of a shortest path from u to v_1). Suppose that after i phases ($1 \leq i < k$), vertex v_i is assigned $\delta(u, v_i)$. Then, as $u = v_0 \to v_1 \to \ldots \to v_i \to v_{i+1}$ is a shortest path from u to v_{i+1} (it is a subpath of a shortest path), after the $(i + 1)$th phase, v_{i+1} is assigned $\delta(u, v_{i+1})$. Thus, by induction, v_k is assigned $\delta(u, v_k)$ (the length of a shortest path from u to $v_k = v$) after k phases. That is, we have proved that after k phases, we have found the length of a shortest path from u to v.

We also have to show that the parent pointers provide us with a shortest path from u to v. Since the above induction can be applied to any vertex w of G (not only to v), after k phases, every vertex w is assigned $w.d = \delta(u, w)$. Using this fact, we can prove that the edges $\{(w.\pi; w) \ : \ w \in V \setminus \{u\}\}$ form a shortest-path tree rooted at u (we leave this as an exercise; see Lemma 24.17, page 676, of our textbook). The unique path from u to v in this shortest path tree is a shortest path from u to v in G (by the definition of a shortest-path tree), which completes the proof.

Algorithm 3 compute the optimal string cuts

Input: A string S of length n; m cutting locations in an increasing order
Output: The minimum cost of breaking the strings into $m+1$ pieces

```
1: I[0] ← 0;
2: I[m + 1] ← 0;
3: c[1, . . . , m + 1; 1, . . . , m + 1] ← ∞;
4: for i ← m + 1 downto 1 do
5:    for j ← i to m + 1 do
6:       if i = j then
7:          c[i, j] ← 0;
8:       else
9:          w ← I[j] − I[i];
10:         for k ← i to j − 1 do
11:            c[i, j] ← min{c[i, j], w + c[i, k] + c[k + 1, j]};
12:         end for
13:      end if
14:   end for
15: end for
16: return c[1, m + 1];
```