# COP 4516 Spring 2026 Week 11 Individual Contest #4
# Weighte Graphs - Solution Sketches

*Underground Cables*

When reading the problem, it looks as if a minimum spanning tree is requested except for one issue: edges that physically intersect/cross are not allowed to be selected. In the typical algorithm it doesn't seem as if there's a method to prevent these edges from being chosen. The key realization is that when the usual algorithm is run on edges with distances generated between vertices on the x-y plane, crossing edges will never be chosen. Consider a picture with crossing edges of a potential minimum spanning tree (Figure 1) that is shown below. In all cases, we can redraw the picture without crossing edges where the alternate edges added are of shorter total length (Figure 2):

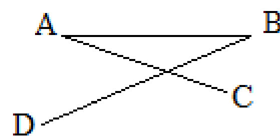


Figure 1

Figure 2

Thus, it follows that we can just create all possible edges between all pairs of vertices (points) and then run either Kruskal's or Prim's. The key point to note is that since the edge weights are doubles, we must take care with our compareTo method. A common bug in a compareTo method would be something of this nature:

```
public int compareTo(edge other) {
    return (int)(this.w - other.w);
}
```

The problem with this implementation is that edges with weights that are within 1 of each other are compared as equal with this method. If two edges are equal, there's no guarantee in which order they will get sorted. Thus, either Prim's or Kruskal's may end up considering an edge with length 2.5 before an edge with length 2.2, for example. In doing so, the algorithm may choose the 2.5 weighted edge OVER the 2.2 weighted edge (if the latter ends up causing a cycle with the former).

Thus, the key is just to compare the edge weights as usual with <, >, etc. For this problem, no tolerance checking is necessary, but it's usually better to give some tolerance when comparing doubles.

*Come Rain or Shine*

We can use either Bellman-Ford or Dijkstra's to determine the shortest path. However, these algorithms don't determine the longest path in a graph. One little trick that works is to negate all of the edge weights in the original graph G, creating a new graph G'. All the distances in G' will be negative (except source to source). If we find the "shortest" distance in this graph, it will correspond to the "most negative" number. If we think about what this means for our original graph G, this corresponds to the "most positive" path length, or longest path length in the negative graph. So one possible solution is as follows: Create G' by negating all of the edges in G. Run Bellman-Ford's on it (since this works with negative edge weights and Dijkstra's doesn't), get the shortest distance to the final destination (a negative number). Then, just negate this to get the longest possible path length in G from source to destination.

*Relatives*

The maximum degree of separation in a connected graph is the longest distance between any two pairs of vertices. This is also known as the diameter of a graph. Since this graph is so small, an algorithm with a run-time of $O(V^3)$ runs plenty fast, since $V \leq 50$. Thus, Floyd-Warshall's can be executed in time to find the shortest distance between all pairs of vertices. Then, we can scan all of these shortest distances to find the largest one. We can handle the disconnected case by either running a single BFS/DFS or simply creating an edge of length 100 between all pairs of vertices that aren't connected. Any shortest distance of 100 represents no path (or a disconnected graph).

*Triangular Sums*

This is the easiest problem in the set. A definition for W(n) is given to you and you have to calculate W(n) for any input value from 1 to 300. One key observation is that $W(n+1) = W(n) + (n+1)T(n+2)$. Thus, we can simply build up the results, one by one. The triangular numbers themselves have a formula, $T(n) = n(n+1)/2$. (Even if one didn't know that formula, one can create an extra variable to store the sum of the first i integers and add to it.)

Substituting, we have $W(n+1) = W(n) + (n+1)(n+2)(n+3)/2$ and $W(0) = 0$. From there we can just run a single for loop building up these values and storing them in an array. Then, process the input answering each query immediately.