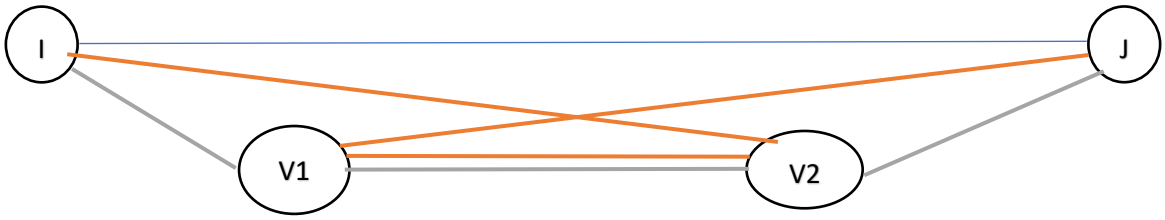
My algorithm models the problem by turning each provided segment into an edge in an UndirectedEdgeWeightedGraph. To find the shortest total duration, one just needs to add up the shortest paths from each vertex to all of the others, something sometimes referred to as the all pairs shortest path problem. To find the best edge to add from the list of possibilities, one just needs to simulate adding the edge and rerunning the shortest path calculation for each of the segment possibilities.

To solve this problem by brute force must simply create a graph for each possibility (the original segments + a possibility for each), solve the all pairs shortest path problem, and see which one has the smallest total. There are two ways to do this. One, using a modification of Warshal's matrix algorithm has a run time of p*V^3, where p is the number of possible edges to add. The other, requires running Dijkstra from every edge and adding the total of each shortest path found. This has a runtime of p*VELogV. This solution is faster, as long as E<=V^1.8 (see the graph at the end) so I went with this option as we were told that the graphs would be sparse.

To optimize this algorithm, I realized that I don't need to recalculate all the shortest paths every time I try out a new possibility. I can run the all pairs shortest path once on the original edges and store the shortest path distances in a matrix. Then, all I must do for each possibility is see if the original distance from each vertex to every other can be reduced by traversing the new edge, represented by v1,v2, and duration. There is a very simple way to do that.



As depicted above, the distance from one point (i) to another (j) (shown above in blue) will only be shortened by a new edge (v1 to v2) if either the distance from i to v1 + v1 to v2+ v2 to j (shown above in gray), or the distance from i to v2 + v1 to v2+ v1 to j (shown above in orange) is shorter than the original. Thus, I can run through each box in my matrix (v^2 time) which represents the shortest distance from i to j, and see if either distance[i][v1]+distance[v2][j]+duration, or distance[i][v2]+[v1][j]+duration is shorter than the original distance[i][j] and replace the original distance with the new shortest distance, storing the amount I was able to decrease it by in a sum. See the pseudocode below for a view of the loop.

For each row i:

    For each column j:

        double currentDistance=distances[i][j]

        if(distances[i][v1]+distances[v2][j]+length<currentDistance):

            totalShortenedByAddingThisEdge=difference between old distance and new

            currentDistance=new shortest distance I to j

```
if(distances[i][v2]+distances[v1][j]+length<currentDistance):
    myTotalShortened+=currentDistance-(distances[i][v2]+distances[v1][j]+length
```

With this modification, instead of running Djikstra from each vertex for each possibility, I only have to run it once. Thus, the big-O is still VELogV, but the constant goes from p to 1. To do that, I must pay a price of pV^2, but since E is certainly greater than V (as the requirements state that it must be one connected component which necessitates two edges from each vertex), my price of pV^2 will certainly be less than the previous cost that I am saving of (p-1)*VELogV.