CS494 -- Lab 4 -- Our Friend Floyd

- CS494
- James S. Plank
- This file: **http://web.eecs.utk.edu/~jplank/plank/classes/cs494/494/labs/Lab-4-Floyd/index.html**
- Lab Directory: **/home/plank/cs494/labs/Lab-4-Floyd**

Your job is to do the 600 point problem from Topcoder SRM 626, D1. Here is the Problem Statement.. You don't have to do it topcoder-style (although you can do that to help you test). Instead, you need to read each set of parameters on standard input, each on its own line.

Since Topcoder's servers sometimes go down, here's a description of the problem tailored to how the lab is done:

- You are given a directed, weighted graph with **N** nodes and **E** edges.
- The nodes are numbered from 1 to **N**.
- There may be up do 2500 edges. There may be multi-edges and self-loops.
- The graph is given to you by 5 parameters, each on a separate line of standard input.
- Parameter #1 is **N**, which is a number between 1 and 50.
- Parameter #2 is a vector with **E** node numbers. It is called **from** .
- Parameter #3 is a vector with **E** node numbers. It is called **to** .
- Parameter #4 is a vector with **E** edge weights. It is called **weights**.
- The weights are numbers between 0 and 100,000.
- Parameter #5 is a number between 0 and 1,000,000,000, called **charges**.
- It should be pretty clear that edge *i* goes from node **from[i]** to node **to[i]** with weight **weight[i]**.
- You want to travel from node 1 to node **N** by traveling along edges.
- The cost of your journey is the sum of the edge weights that you travel along, and you want to minimize the cost of your journey.
- Your travels can involve cycles if you want -- you can visit nodes and edges as many times as you'd like.
- When you are traveling, you may use a "charge", which negates the edge's weight when you travel along it.
- You can use a total of **charges** charges.
- Return the minimum cost of your journey. It should be a **long long**.

I have the examples in text files in the lab directory. You can see their answers below. Comments on the examples:

- Example 0: The optimal path is [1,2,3], using the charge on the edge from 2 to 3.
- Example 1: The graph may contain self-loops, so you use the loop 100000 times, applying the charge each time.
- Example 2: When you have two edges between a pair of nodes, you may want to use one of them without a charge and one of them with a charge.
- Example 3: When the graph doesn't have cycles, you may not be able to use all of the charges.

So you can see the format of the input, here's example 0:

```
UNIX> cat Example-0.txt
3
1 1 2  2 3 3
```

```
2 3 1  3 1 2
1 5 1 10 1 1
1
UNIX> NegativeGraphDiv1 < Example-0.txt
-9
UNIX>
```

I have the rest of the topcoder examples in the lab directory as well:

```
UNIX> NegativeGraphDiv1 < Example-1.txt
-10000000
UNIX> NegativeGraphDiv1 < Example-2.txt
-9
UNIX> NegativeGraphDiv1 < Example-3.txt
-98765
UNIX> NegativeGraphDiv1 < Example-4.txt
-15328623718914
UNIX> time NegativeGraphDiv1 < Example-4.txt
-15328623718914
0.396u 0.004s 0:00.40 97.5%     0+0k 0+0io 0pf+0w
UNIX>
```

Example 4 is kind of a big one, but it doesn't max out the constraints. I'll have some in the gradescripts that do max out the constraints.

```
UNIX> head -n 1 Example-4.txt
40
UNIX> sed -n 4p Example-4.txt | wc
      1     442    2593
UNIX> tail -n 1 Example-4.txt
160743953
UNIX>
```

The gradescript *will* time your program, and you need to come in under two seconds of user time when compiled with -O2.

I'm not testing your error checking.

This problem has three components to it, and they all have a Floyd-Warshall flavor to them.

Part 1

This is the actual Floyd Warshall component. Make an adjacency matrix out of the smallest weight edges from each node to each other node (with zero-weight self-edges). Run Floyd-Warshall on that to create a matrix of shortest paths from every node to every other node. Call this matrix S. If **charges** equals 0, you can use this matrix to return the answer.

While you're at it, create an adjacency matrix that has the largest weight edges from every node to every other node (zero if there is no edge). Call that matrix L.

To help you, you can give an optional command line argument to my program (your program does not have to do this). If the argument is "P0", then the program will print out the original adjacency matrix, and if it's "P1", then the program will print out S:

```
UNIX> mv a.out NegativeGraphDiv1
UNIX> NegativeGraphDiv1 P0 < Example-0.txt
   6000000         1         5
```

```
         1   6000000        10
         1         1   6000000
-9
UNIX> NegativeGraphDiv1 P1 < Example-0.txt
         0         1         5
         1         0         6
         1         1         0
-9
UNIX>
```

Part 2

Now create a new matrix, which we'll call A. This is going to be a matrix of shortest path lengths from each node to each other node, where there is *exactly* one negative edge used.

You make this by enumerating the following:

- For every starting node f.
- For every ending node t;
- For every intemediate starting node i;
- For every intemediate ending node j:

Consider the path f->i, (i,j), j->t, where the length of f->i and j->t come from S, and (i,j) is the longest edge from i to j (in L), multiplied by -1 to make it negative. Set A[f][t] to be the shortest of these for all i and j.

You can give my executable the argument "P2" to see my A matrix:

```
UNIX> NegativeGraphDiv1 P2 < Example-0.txt
        -8        -8        -9
        -9        -9       -10
        -8        -8        -9
-9
UNIX>
```

Part 3

Now, you want to create a matrix $A_{charges}$ for an arbitrary value of *charges*. $A_{charges}$ is going to be the matrix of shortest paths from *f* to *t* that have exactly *charges* negative edges. If you have A_i and A_j, and *x* equals *i*+*j*, then you can make A_x from A_i and A_j. You do that by considering all paths from *f* to *z* (using A_i) and *z* to *t* (using A_j) for every possible intermediate node *z*.

Given that, you should be able to make any arbitrary A_x in *O(log x)* time.

If you give my executable "P3", it will print the final $A_{charges}$ matrix. I won't show this on Example 0, since *charges* equals 1 there. Here's example 2:

```
UNIX> NegativeGraphDiv1 P0 < Example-2.txt
   6000000         1
         4   6000000
-9
UNIX> NegativeGraphDiv1 P1 < Example-2.txt
         0         1
         4         0
```

```
-9
UNIX> NegativeGraphDiv1 P2 < Example-2.txt
        -3        -6
        -4        -3
-9
UNIX> NegativeGraphDiv1 P3 < Example-2.txt
       -10        -9
        -7       -10
-9
UNIX>
```

I don't know if you have analyzed running times yet, but Part 1 is *$O(N^3)$*, Part 2 is *$O(N^4)$*, and Part 3 is *$O(N^3 log(charges))$*.