Results from UVA

17180540	11506	Angry Programmer	Runtime error	JAVA	0.000	2016-04-10 19:58:48
17174431	544	Heavy Cargo	Accepted	JAVA	0.290	2016-04-09 18:50:28

1 UVA Problem 11506: Angry Programmer

Background

You, a programmer of an important software house, have been fired because you didn't solve an important problem that was assigned to you. You are very furious and want to take revenge on your boss, breaking the communication between his computer and the central server.

The computer of your boss and the central server are in the same network, which is composed of many machines (computers) and wires linking pairs of those machines. There is at most one wire between any pair of machines and there can be pairs of machines without a wire between them.

To accomplish your objective, you can destroy machines and wires, but you can't destroy neither the computer of your boss nor the central server, because those machines are monitored by security cameras. You have estimated the cost of blowing up each machine and the cost of cutting each wire in the network.

You want to determine the minimum cost of interrupting the communication between your boss' computer and the central server. Two computers A and B can communicate if there is a sequence of undestroyed machines $x_1, ..., x_n$ such that $x_1 = A, x_n = B$ and x_i is linked with x_{i+1} with an uncut wire (for each $1 \leq i \leq n-1$).

Input

The input consists of several test cases. Each test case is represented as follows:

- A line with two integers M and W ($2 \leq M \leq 50, 0 \leq W \leq 1000$), representing (respectively) the number of machines and the number of wires in the network.
- **M-2** lines, one per machine (different from the boss' machine and the central server), containing the following information seperated by spaces:
  - An integer i ($2 \leq i \leq M-1$) with the identifier of the machine. Assume that the boss' machine has id 1 and that the central server has id M.
  - An integer c ($0 \leq c \leq 100000$) specifying the cost of destroying the machine.
- **W** lines, one per wire, containing the following information separated by spaces:
  - Two integers j and k ($1 \leq j < k \leq M$) specifying the identifiers of the machines linked by the wire. remember that the wire is bidirectional.
  - An integer $d (0 \leq d \leq 100000)$ specifying the cost of cutting the wire.

The end of the input is specified by a line with the string "0 0".
Suppose that the machines have distinct identifiers.

1.1 Mathematical Formulation

Given an input of M machines, $M-2$ of which are potential intermediate nodes between the boss' computer and the server, and W wires connecting computers m_i and m_j where $0 \leq i<j \leq M$, determine the minimum cost to destroy (m + w) machines and wires where $(0 \leq m \leq M-2, 0 \leq w \leq W)$.

1.2 Solution

Important Confusing Data Structures:

- LinkedList<FlowEdge>[ ] **adj** : An adjacency list of all FlowEdges where the specified node is the originating node.

The main functionality of this algorithm is to implement the Ford Fulkerson min cut algoritm by first building a flow network and then executingthe algorithm on it. We represent the network as each machine (index) has its own linked list of Flow Edges. Each computer has 2 nodes, one as its input and the other as its output. The only edge that flows out of the input and into the output is the edge connecting the two such that the input points to the output with edgeweight equivalent to the cost of destroying that computer; this is the MAX_INTEGER for the boss' computer and the server since we cannot destroy these. Since we are given the boss' computer as "1" and the server as "M" we just let the input for every computer be index = i-1 and their output as index = (i-1)+M for $i \in 1..M$ as described below.

Algorithm 1 Determine index of Computers

```
procedure detIndex(int index, boolean out)
    if out then return (index - 1) + m
    return (index - 1)
```

Once we do this we have to ensure that the bi-directional rule is accounted for. So, for each wire connecting computers A and B with cost C, we will connect the output for A to the input for B with weight C and the output of B to the input of A with cost C. Once we have built up the entirety of our network we will simply use the Ford Fulkerson algorithm on the built network and print that out. (As outlined below)

Algorithm 2 Build the Flow Network from input

```
procedure MAIN(Scanner input)
    while true do
        m, w ← from input
        if m == 0 && w == 0 then break;
        if w == 0 then
            PRINT(0)
            for m-2 times do input.NEXTLINE( )
        intnumV, serverIn ← 2*m, m-1 respectively
        adj[numV] ← initialized
        // create computers
        adj[0].ADD(FlowEdge(0, m, Integer.MAX_VALUE))
        adj[serverIn].ADD(FlowEdge(serverIn, numV-1, Integer.MAX_VALUE))
        for length of m-2 do
            inti, c ← index of machine and cost to destroy
            FlowEdgee ← new FlowEdge(i, i+m, c)
            adj[i].ADD(e)
        // connect computers to eachother
        for length of W do
            inta, b, c ← comp1, comp2, cost to cut wire
            outA, inA ← detIndex(a, true), detIndex(a, false)
            outB, inB ← detIndex(b, true), detIndex(b, false)
            e1, e2 ← FlowEdge(outA, inB, c), FlowEdge(outB, inB)
        PRINT(calcMinCut())
```

1.3 Correctness

Proposition 1.
This is the correct flow Network to build which will yeild the min cut.

Proof.
This is correct because we are using the "flow" of each edge to be the cost of destroying either a wire or a computer. We account for the fact that if we destroy a computer, then we will also sever any connection that used this computer as an intermediate node. We have done this by seperating each computer into an input and an output node, between which ∃ only one connection which has a flow of the cost of destroying a computer. The bi-directional aspects has us connect the network such that, for each wire connecting computers A and B with cost C, we will connect the output for A to the input for B with weight C and the output of B to the input of A with cost C. Therefore if we sever inA to outA (destroying computer A) B would no longer be able to use this as an intermedite node since inA no longer has any out edges and outA is unreachable since its only in edge was just severed. □

1.4 Analysis

For the following analysis, we will say that we have built a flow network graph with $V = 2 \cdot M$ and $E = M + 2 \cdot W$. Our Time and Space analysis will be done with these variables but can be converted to V ≈ M and E ≈ (M+W)

Proposition 2. *The space complexity of this algorithm is* ***O(V+E)***

Proof.

This is due to the fact that all of our data is stored in data structures:

- adj[V]: Our Flow NetWork $\implies V + E$
- edgeTo[V]: Used in Ford Fulkerson implementation

Giving us a space complexity of **O(V+E)**

□

Proposition 3. *The time complexity of this algorithm is* ***O(V+E+MaxFlowValue·E)***

Proof.

This is the case because our algorithm first builds our flow network which takes V + E time and then implements the Ford Fulkerson Min Cut algorithm which is synonomous to the Max Flow. This implementation takes MinCutValue·E time.

Giving us a time complexity of **O(V+E+MaxFlowValue·E)**

□

1.5 An Example

input

4 4
3 5
2 2
1 2 3
1 3 3
2 4 1
3 4 3
0 0

Now when we run this through our Algorithm we will build first the nodes for the computers and connect their "ins" to their "outs"

$$\begin{aligned}(\text{boss}) &\implies 0 \to 4 \text{ with cap} = 2.147483647 \cdot 10^9 \\ (\text{server}) &\implies 3 \to 7 \text{ with cap} = 2.147483647 \cdot 10^9 \\ (3\ 5) &\implies 2 \to 6 \text{ with cap} = 5.0 \\ (2\ 2) &\implies 1 \to 5 \text{ with cap} = 2.0\end{aligned}$$

Next we build the wire connections:

$$\begin{aligned}(1\ 2\ 3) &\implies 4 \to 1 \text{ with cap} = 3.0\\ \text{convex} &\implies 5 \to 0 \text{ with cap} = 3.0\end{aligned}$$

$$\begin{aligned}(1\ 3\ 3) &\implies 4 \to 2 \text{ with cap} = 3.0\\ \text{convex} &\implies 6 \to 0 \text{ with cap} = 3.0\end{aligned}$$

$$\begin{aligned}(2\ 4\ 1) &\implies 5 \to 3 \text{ with cap} = 1.0\\ \text{convex} &\implies 7 \to 1 \text{ with cap} = 1.0\end{aligned}$$

$$\begin{aligned}(3\ 4\ 3) &\implies 6 \to 3 \text{ with cap} = 3.0\\ \text{convex} &\implies 7 \to 2 \text{ with cap} = 3.0\end{aligned}$$

Then when we calculate the mincut value we get 4!

2 UVA Problem 544: Heavy Cargo

Background

Big Johnsson Trucks Inc. is a company specialized in manufacturing big trucks. Their latest model, the Godzilla V12 , is so big that the amount of cargo you can transport with it is never limited by the truck itself. It is only limited by the weight restrictions that apply for the roads along the path you want to drive.

Given start and destination city, your job is to determine the maximum load of the Godzilla V12 so that there still exists a path between the two specified cities.

Input

The input file will contain one or more test cases. The first line of each test case will contain two integers: the number of cities n ($2 \leq n \leq 200$) and the number of road segments r ($1 \leq r \leq 19900$) making up the street network.

Then r lines will follow, each one describing one road segment by naming the two cities connected by the segment and giving the weight limit for trucks that use this segment. Names are not longer than 30 characters and do not contain white-space characters. Weight limits are integers $\in (0, 10000)$. Roads can always be travelled in both directions.

The last line of the test case contains two city names: start and destination.

Input will be terminated by two values of 0 for n and r.

Output

For each test case, print three lines:

- a line saying "Scenario #x" where x is the number of the test case
- a line saying "y tons" where y is the maximum possible load
- a blank line

2.1 Mathematical Formulation

Given an input of n cities, $c_1, c_2, ..., c_n$, and r roads with loads $l_1, l_2, ..., l_r$ which connect cities c_i *and* c_j, $1 \leq i, j \leq r$, we will determine the maximum load that can be transported from the specified cities (a to b).

2.2 Solution

Important and potentially confusing data structures:

- HashMap<String, Integer> **cityIndex** : keeps track of the index associated with each city.
- int[n][n] **load** : each element load[i][j] stores the highest constraining load that each road connecting city i and j.
- int[n] **curSet** : stores the current row/column that we will add together for the Floyd-Warshall implementation.
- int[n][n] **l_i** : corresponding row/column sum that is calculated in each sub situation during the Floyd-Warshall algorithm.

The main functionality of this algorithm is an adapted Floyd-Warshall implementation where we first build up our cityIndex HashMap and our load[ ][ ] which stores the max load that each road can take between city "a" and city "b".

Algorithm 3 Set-Up

```
procedure BUILD(Scanner in)
    intsenario ← 0
    while true do
        n, r ← number of cities and number of roads from in
        if n == 0 and r == 0 then break;
        load[n][n]andcityIndex ← initialized
        intlastIndex ← 0
        for i ∈ 0..(r − 1) do
            If not in hashmap put in.
            inta, b ← index of the two connected cities from cityIndex
            intcurLoad ← the load specified by the file
            load[a][b], load[b][a] ← curLoad;
        FLYODWARSHALL(load) // see Algorithm 2
        inta, b ← index of the two cities want to connect from cityIndex
        PRINT(load[a][b])
```

The main difference comes from our rules, instead of using the rule:

$$load(i,j,k) = \textbf{min} \begin{cases} load(i,j,k-1) \\ load(i,k,k-1) + load(k,j,k-1) \end{cases}$$

where i, j are the the correspoding cities i and j and i = j $\implies$ load(i,j,k) = 0 $\forall$ k, we have

$$load(i,j,k) = \textbf{max} \begin{cases} load(i,j,k-1) \\ load(i,k,k-1) + load(k,j,k-1) \end{cases}$$

One thing to note here is that we will be representing the distances seen so far in our 2-D array *load* array. When implementing this the things to keep in mind are that elements load[a][b] = load[b][a] always.

Algorithm 4 Floyd-Warshall Implementation

```
procedure floydWarshall(int[ ][ ] load)
    curSet, l_i ← initialized
    for i ∈ 0..(n − 1) do
        for k ∈ 0..(n − 1) do
            curSet[k] ← load[i][k]
        // build l_i
        for row ∈ 0..(n − 1) do
            for col ∈ 0..(n − 1) do
                if row == col then continue;
                l_i[row][col] ← Math.min(curSet[row], curSet[col]);
        // do dynamic step
        for row ∈ 0..(n − 1) do
            for col ∈ 0..(n − 1) do
                if row == col then continue;
                load[row][col] ← Math.max(load[row][col], l_i[row][col]);
```

2.3 Correctness

Proposition 4.
Our adapted Floyd-Warshall Shortest Path Distance algorithm to a Floyd-Warshall Highest Load algorithm sufficiently solves the problem.

Proof.
In class we prooved that the Floyd-Warshall Shortest Path Distance algorithm will determine the shortest path between two nodes given a "graph" with the edge weights being the distance between two nodes. Now the dynamic algorithm rule that

we use to build this solution is expressed as:

$$dist(i,j,k) = \textbf{min} \begin{cases} dist(i,j,k-1) \\ dist(i,k,k-1) + dist(k,j,k-1) \end{cases}$$

$$where,\ dist(i,j,0) = \begin{cases} w_{i,j}\ if\ (i,j) \in E \\ \infty\ if\ (i,j) \notin E \\ 0\ if\ i = j \end{cases}$$

Such that $w_{i,j}$ is the weight between nodes i and j and E is the set of all edges. Now we adapt this such that E is all roads, $w_{i,j}$ is the load of each road between cities i and j. Therefore we are no longer looking for distances, but loads $\implies dist \to load$ and instead of finding the min we are looking for the max. Therefore, we have the new formulation:

$$load(i,j,k) = \textbf{max} \begin{cases} load(i,j,k-1) \\ load(i,k,k-1) + load(k,j,k-1) \end{cases}$$

Which is the algorithm which we have implemented. Since this is the only difference with the original, we can say that this is sufficient to solve the given problem. □

2.4 Analysis

For the following analysis, we will say that N is the number of cities that are given to us. We note here that all of the analysis is dependent on the number of cities and not the number of roads. This is due to the fact that the roads are being represented through our load[ ][ ] which is dependent on N already.

Proposition 5. *The space complexity of this algorithm is* ***O(N²)***

Proof.

This is due to the fact that all of our data is stored in data structures:

- HashMap<String, Integer> **cityIndex** : stores indexes of all cities $\implies N$
- int[n][n] **load** : stores the distances from cities a to b $\implies N^2$
- int[n] **curSet** : stores the row/column distances that will be used to calculate l_i $\implies N$
- int[n][n] **l_i** : stores the calculated distances from cities a to b (N^2) from $curSet \implies N^2$
- cause: reason $\implies$ *complexity*

At any point in time, there will exist at most one of each of these data structures $\implies 2 \cdot N^2 + 2 \cdot N$

$\therefore$ Giving us a space complexity of $\mathbf{O(N^2)}$

□

Proposition 6. *The time complexity of this algorithm is* $\boldsymbol{O(N^3)}$

Proof. This is the case because our algorithm is just the adjusted Floyd-Warshall Algorithm which builds N, NxN matricies. At each step, n *s.t.* $1 \leq n \leq N$ we are computing the NxN matrix in linear time (N^2) and somparing each element to an existing NxN graph which is also done in linear time. Therefore we are going through N^2 operations N times.

$\therefore$ Giving us a time complexity of $\mathbf{O(N^3)}$

□

2.5 An Example

Given the input:
4 3
K S 100
S U 80
U M 120
K M
We build our initial matrix load[4][4]:

$$load = \begin{bmatrix} \mathbf{0} & \mathbf{100} & \mathbf{0} & \mathbf{0} \\ \mathbf{100} & 0 & 80 & 0 \\ \mathbf{0} & 80 & 0 & 120 \\ \mathbf{0} & 0 & 120 & 0 \end{bmatrix}$$

And we begin our Floyd-Warshall algorithm by computing l_i with the rows bolded above to produce (the altered load elements have been underlined):

$$l\_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \implies load = \begin{bmatrix} 0 & \mathbf{100} & 0 & 0 \\ \mathbf{100} & \mathbf{0} & \mathbf{80} & \mathbf{0} \\ 0 & \mathbf{80} & 0 & 120 \\ 0 & \mathbf{0} & 120 & 0 \end{bmatrix}$$

continuing computing l_i with the rows bolded above to produce:

$$l\_2 = \begin{bmatrix} 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 0 \\ 80 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \implies load = \begin{bmatrix} 0 & 100 & \underline{\mathbf{80}} & 0 \\ 100 & 0 & \mathbf{80} & 0 \\ \underline{\mathbf{80}} & \mathbf{80} & \mathbf{0} & \mathbf{120} \\ 0 & 0 & \mathbf{120} & 0 \end{bmatrix}$$

continuing computing l_i with the rows bolded above to produce:

$$l\_3 = \begin{bmatrix} 0 & 80 & 0 & 80 \\ 80 & 0 & 0 & 80 \\ 0 & 0 & 0 & 0 \\ 80 & 80 & 0 & 0 \end{bmatrix} \implies load = \begin{bmatrix} 0 & 100 & 80 & \underline{\mathbf{80}} \\ 100 & 0 & 80 & \underline{\mathbf{80}} \\ 80 & 80 & 0 & \mathbf{120} \\ \underline{\mathbf{80}} & \underline{\mathbf{80}} & \mathbf{120} & \mathbf{0} \end{bmatrix}$$

and computing the final l_i with the rows bolded above to produce:

$$l\_3 = \begin{bmatrix} 0 & 80 & 80 & 0 \\ 80 & 0 & 80 & 0 \\ 80 & 80 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \implies load = \begin{bmatrix} 0 & 100 & 80 & 80 \\ 100 & 0 & 80 & 80 \\ \mathbf{80} & 80 & 0 & 120 \\ 80 & 80 & 120 & 0 \end{bmatrix}$$

Therefore since K correlates with index 0 and M with index 3, the maximum load is then load[0][3] = 80 (as bolded above)

3 Book 6.16: ROTC

Background

There are many sunny days in Ithaca, New York; but this year, as it happens, the spring ROTC picnic at Cornell has fallen on a rainy day. The ranking officer decides to postpone the picnic and must notify everyone by phone. Here is the mechanism she uses to do this.

Each ROTC person on campus except the ranking officer reports to a unique *superior officer*. Thus the reporting hierarchy can be described by a tree T, rooted at the ranking officer, in which each other node v has a parent node u equal to his or her superior officer. Conversely, we will call v a *direct subordinate* of u.

To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time. The process continues this way until everyone has been notified. Note that each person in this process can only call direct subordinates on the phone.

We can picture this process as being divided into rounds. In one round, each person who has already learned of the postponement can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds.

3.1 Mathematical Formulation

Given an input of the tree T with n nodes in total and depth of d. Let the most number of direct subordinates (children) any one officer (parent) has be expressed as c. We will determine least number of calls needed to reach every node of the tree as well as the order in which to acheive this.

3.2 Solution

This algorithm begin by doing a by-level bottom up traversal. As it traverses, it will assign values to each node based off its children. The way it does this is first sorting all of its children from highest value to lowest and such that $v_0 \geq v_1 \geq ... \geq v_{n-1}$ for a node with n children, then saying:

$$value = \begin{cases} 0 \; if \; no \; children \\ max_{i \in 0..(n-1)} \; (child_i value + 1 + i) \; else \end{cases}$$

With this we will develope our values and when we reach the top level (n), its value will be the least number of calls needed to be made. Our path to take then is to call the open value node from each seen node.

Algorithm 5 Fill out Tree and get num Calls

```
procedure getNumCalls(Tree T)
    for level ∈ 1..d do
        for node ∈ T.level(level) do
            int value = 0
            if node.hasChildren then
                children.inverseSort
                int i = 0
                for child ∈ children do
                    childV = child.value + 1 + (i++)
                    value = max(value, childV)
            node.setValue(value)
    print(root.value) // least num calls
    traverse(T) // T has all values filled out and inverse sorted
```

3.3 Correctness

Proposition 7.
This algorithm will determine the correct minimum number of calls.

Proof.
We do this by construction. We observe that doing a bottom up traversal we can see that as long as the children have the correct value, then the parent will be able to calculate the correct value. When I say value of a node I am referring to the number of calls required to fully reach the subtree rooted at this node, assuming the node has already been reached. $\implies$ if the node is a leaf, its value = 0. Therefore when we have a node with a single leaf as a child, $\implies$ will take the number of calls that the leaf will make, 0, + 1 because it has to call the leaf. Now if there are multiple children, k of them, and all of them are leaves we will note that we cannot call multiple leaves at the same time, therefore the number of calls that this node will have to make is n and its value will also be n since each of the nodes must make 0 calls.

Now lets change it such that each of the children have thier own value $v_1, ..., v_n$ we note that it will always be most efficient to call the children with more calls to make first, therefore let us inverse-sort them based off of their values s.t. $v_1 \geq ... \geq v_n$ Now we can create a *value* variable and set it equal to cost of calling c_1 which is $v_1 + 1$ beacause we have to call the node itself. We will then check to see if the cost of calling c_2 exceeds that of c_1 as in is cost of calling c_2 second > cost of calling c_1 first? i.e. $value = max(value, v_2 + 1 + 1)$ since it cost 0 to call first, 1 to call second , 2 to call third, etc. We will then continue with this letting $value = max_{i \in 1..(n)}\ (child_i value + 1 + (i - 1))$

After we have done this for every node (building from bottom $\rightarrow$ up), we just ask for the value of the root which will give us the minimum number of calls we have to make to reach everyone. □

3.4 Analysis

Proposition 8. *The space complexity of this algorithm is* ***O(size(T))***

Proof.
This is due to the fact that all of our data is stored in the tree itself: □

Proposition 9. *The time complexity of this algorithm is* $\boldsymbol{O(N \cdot n^* \cdot log(n^*))}$

Proof.
Let us define the average number of children every node has as n^* $\implies$ the cost to sort will be $n^* \cdot log(n^*)$ now since we have to sort for each node, N times $\implies$ $N \cdot n^* \cdot log(n^*)$ Note here though that n^* is likely to be a low number when compared to N. □