1 Results from UVA

17146660	544 Heavy Cargo	Accepted	JAVA	0.353	2016-04-04 20:08:37

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 1 Set-Up

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

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

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

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

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 2 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 1.

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:

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

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

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:

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

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 2. *The space complexity of this algorithm is* $\boldsymbol{O(N^2)}$

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 $\implies 2 \cdot N^2 + 2 \cdot N$

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

□

Proposition 3. *The* <u>*time complexity*</u> *of this algorithm is* ***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.

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

□

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)