1 Results from UVA

17217712	10937 Blackbeard the Pirate	Accepted	JAVA	0.160	2016-04-17 16:43:13

2 UVA Problem 10937: Blackbeard

Background

Blackbeard the Pirate has stashed up to 10 treasures on a tropical island, and now he wishes to retrieve them. He is being chased by several authorities however, and so would like to retrieve his treasure as quickly as possible. Blackbeard is no fool; when he hid the treasures, he carefully drew a map of the island which contains the position of each treasure and positions of all obstacles and hostile natives that are present on the island.

Given a map of an island and the point where he comes ashore, help Blackbeard determine the least amount of time necessary for him to collect his treasure.

Input

Input consists of a number of test cases. The rst line of each test case contains two integers h and w giving the height and width of the map, respectively, in miles. For simplicity, each map is divided into grid points that are a mile square. The next h lines contain w characters, each describing one square on the map. Each point on the map is one of the following:

- @ The landing point where Blackbeard comes ashore.
- $\sim$ Water. Blackbeard cannot travel over water while on the island.
- # A large group of palm trees; these are too dense for Blackbeard to travel through
- · Sand, which he can easily travel over.
- $*$ A camp of angry natives. Blackbeard must stay at least one square away or risk being captured by them which will terminate his quest. Note, this is one square in any of eight directions, including diagonals.
- ! A treasure. Blackbeard is a stubborn pirate and will not leave unless he collects all of them.

Blackbeard can only travel in the four cardinal directions; that is, he cannot travel diagonally. Blackbeard travels at a nice slow pace of one mile (or square) per hour, but he sure can dig fast, because digging up a treasure incurs no time penalty whatsoever.

The maximum dimension of the map is 50 by 50. The input ends with a case where both h and w are 0. This case should not be processed.

Output

For each test case, simply print the least number of hours Blackbeard needs to collect all his treasure and return to the landing point. If it is impossible to reach all the treasures, print out ''-1'

2.1 Mathematical Formulation

Given an input of the above described map with n treasures and a starting position we will determine whether or not all treasures are reachable, if so the distances between all of them and the length of the shortest path to start at the starting point, visit every treasure, and return to the start point.

2.2 Solution

Important Confusing Data Structures:

- int **R, C, numTreasures** : Stores (respectively), the max number rows and columns in the given map and the total number of treasures.
- char[R][C] **map** : Stores the map provided from the input, then altered so that any un-crossable peice of land is represented as the symbol '#' in *replaceCamps*()
- int[numTreasures+1][numTreasures+1] **dist** : Stores the distnaces between two treasures (or the source). Built through calls to $bfs()$.
- int[numTreasures+1][2] **tresLoc** : Stores the row and column coordinates of each treasure, $t_1 \rightarrow (x_1, y_1), t_2 \rightarrow (x_2, y_2), ..., t_n \rightarrow (x_n, y_n)$ at indexes $1..n$. Index 0 corresponds with the coordinates of the source.
- HashMap<Integer><Integer> **index** : Keeps track of the index of each treasure based off their [row][col] coordinates. The keys were (row$\cdot$**R**+col) and the value as the treasures index in the **dist** array. This is initialized and built in $locateTreasure()$ and used in $bfs()$.
- int[] **dr, dc** : These arrays are **dr** $\leftarrow$ {-1, 0, 0, 1, 1, -1, -1, 1} and **dr** $\leftarrow$ { 0, -1, 1, 0, 1, -1, 1, -1} which when put together will give you left, down, up, right, up/right, down/left, up/left, down/right. The whole arrays are used in the *replaceCamps*() whereas, only the first 4 entries are used for the $bfs()$ since Blackbeard can only move in the 4 cardinal directions (not diagonally).
- HashMap<String><Integer> **mtsp** : This is the hashmap which stores the intermediate values for the tsp $\therefore$ only seen in *tsp*(). Given n total treasures

each key will be some string i.e. n = 6, string = 10001003. We can break this up such that the first n+1 characters are either 0 or 1, (1 is the presence of that indexed treasure and 0 is the absence) and the first character (string.charAt(0)) is always 1 signifying we always have the source. Then the final character is a number s, $1 \leq s \leq n$, symbolizing that this number is the index of the treasure which we want to be last. Therefore putting all of them together gives us a subset where the source is always present followed by either 1's or 0's (if any intermeditate treasures) and finally the index of the last treasure in the subset.

The main functionality of this algorithm is to process the given map and put more constraints on it, then construct a distance adjacency list using a breadth first search, essentially building a fully connected graph, then finally running the Traveling Salesman Problem solution on this.

Algorithm 1 Main

```
procedure MAIN( )
    while true do
        R, C ← number of Rows and Columns respectively
        if R == C == 0 then break;
        Fill map from input and count number of Treasures
        numTreasures ← number of Treasures on map
        if numTreasures == 0 then
            PRINT(-1); continue;
        REPLACECAMPS( ) // see Algorithm 2
        if !locateTreasures() then // see Algorithm 3
            PRINT(-1); continue;
        else
            booleanreachable ← true
            dist[][] ← init
            for each treasure, t do
                if !bfs(t_x,t_y) then // see Algorithm 4
                    reachable ← false; break;
            if !reachable then
                PRINT(-1); continue;
```

When we are altering the map[ ][ ], what we want to do is to replace all instances of water and angry native camps with the same symbol for trees. In addition, we are also ensuring that we change the surrounding area of the camps as described in the problem description.

Algorithm 2 Alter the Map

```
procedure replaceCamps( )
    for row ∈ 0..R, col ∈ 0..C do
        charc ← map[row][col]
        if c == '∼' then map[row][col] ← '#';
        else if c == '∗' then
            map[row][col] ← '#';
            for i ∈ 0..dc.length do
                int rPrime, cPrime ← row + dr[i], col + dc[i]
                if (rPrime and cPrime in bounds) and (not a camp) then
                    map[rPrime][cPrime] ← '#'
```

Once we have Altered this we need to initialize index and tresLoc and ensure that we still have all of our treasures and start point (these could have been erased in the *replaceCamps*() method). We will return false if either of these have been altered.

Algorithm 3 Build Support for dist array and ensure treasures preserved

```
procedure locateTreasures( )
    index, tresLoc ← init, boolean start = false, count = 0
    for row ∈ 0..R, col ∈ 0..C do
        if map[row][col] == '@' then
            start ← true
            tresLoc[0][0, 1] ← row, col
        else if map[row][col] == '!' then
            index.put(row*R+col, ++count)
            tresLoc[count][0, 1] ← row, col
    return start && (count==numTreasures)
```

Once we have built our adjacency distance array, we can perform the Travelling Salesman Problem solution. We do this using our HashMap mtsp (see explanation of data structure above for details).

Algorithm 4 Traveling Salesman Problem, determine cost of minimum path

procedure TSP( )
  $mtsp \leftarrow$ init; int length = (2 <<(dist.length-2));
  // build up the dp hashmap
  for $i \in [length, (length{<<}1)]$ **do**
    $char[]\ bitRep \leftarrow$ Binary representation of i
    for $c \in [1, BitRep.length]$ **do**
      if bitRep[c] == '1' **then**
        $bestVal \leftarrow$ MAX_VALUE;
        for $k \in [1, bitRep.length]$ **do**
          if bitRep[k] == '1' **then**
            $val \leftarrow$ mtsp.get(cToString(bitRep,k)) + dist[k][c]
            if val <bestVal **then** bestVal = val
        mtsp.put(cToString(bitRep, c), bestVal);
  // determine the best option on how to return to the source
  $lastLevel \leftarrow$ full bit representation (all 1's); $best \leftarrow$ MAX_VALUE
  for $i \in [1, lastLevel.length]$ **do**
    $val \leftarrow$ mtsp.get(cToString(lastLevel, i)) + dist[i][0];
    if val <best **then** $best \leftarrow$ val
  Print best;

2.3 Correctness

Proposition 1.
This is clearly a TSP problem for which we need to construct a fully connected graph G to run it on. Therefore this algorithm will construct such a graph (if one exists) that the TSP solution can be run on.

Proof.
We build this graph G such that each node is either a treasure or the source. We determine if this is possible by first recording how many treasures there are. Then we clean up the map such that the native camps and all of their surrounding aread are marked as unreachable. Then we ensure that all of the treasures are still there; if they are not, we know that we cannot create G.

Next we begin to create the connections between treasures by running a bfs with the source and each treasure as the starting point and only marking tiles where Blackbeard can travel. i.e. we check to see if all treasures/source are reachable from eachother and how far each one is from one another. These distances are recorded as the the edge weights between each point. Therefore, after fully running each n+1 bfs's we have created a fully connected graph G. □

2.4 Analysis

For the following analysis, we will say that the map that we are given is of dimensions RxC which contains N treasures and 1 source. It should be noted that N is (in the typical case) much smaller than $R \cdot C$.

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

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

- map: Stores the map $\implies R \cdot C$
- tresLoc: Stores the location of each treasure on the map $\implies 2 \cdot (N+1)$
- dist: The adjacency matrix for all treasures and the source $\implies (N+1)^2$
- index: Indicies of the treasures in the distance array based off of their coordinates in the map $\implies N$
- mtsp: Stores all of the sub-problem solutions to tsp $\implies N \cdot 2^N$

Summing these all together we get $(R \cdot C) + (2 \cdot (N+1)^3) + (N) + (N \cdot 2^N)$

Giving us a space complexity of $\mathbf{O(R{\cdot}C{+}N{\cdot}2^{N})}$

□

Proposition 3. *The time complexity of this algorithm is* $\boldsymbol{O(N{\cdot}R{\cdot}C{+}N{\cdot}2^N)}$

Proof. This is the case because our algorithm performs the following actions: build the map $(R \cdot C)$; alter the map $(R \cdot C)$; locate the treasures $(R \cdot C)$; perform a bfs witch each treasure as the source $(N \cdot R \cdot C)$; perform traveling salesman problem using all of the treasures as nodes $(N \cdot 2^N)$. When we sum this together we get $2 \cdot (R \cdot C) + (N \cdot R \cdot C) + (N \cdot 2^N)$

Giving us a time complexity of $\mathbf{O(N{\cdot}R{\cdot}C{+}N{\cdot}2^{N})}$

□

2.5 An Example

We read in the input in our map[5][5] and then convert it to our uniform form:

$$map = \begin{bmatrix} . & ! & . & \# & \sim \\ \sim & . & . & . & \sim \\ * & . & \# & . & @ \\ \sim & \# & \# & . & \sim \\ \sim & \sim & \sim & ! & \sim \end{bmatrix} \to \begin{bmatrix} . & ! & . & \# & \# \\ \# & \# & . & . & \# \\ \# & \# & \# & . & @ \\ \# & \# & \# & . & \# \\ \# & \# & \# & ! & \# \end{bmatrix}$$

We see that both treasures and the source are still there so we proceed with computing a bfs with each source and two treasures as the source producing:

$$source = \begin{bmatrix} 6 & \mathbf{5} & 4 & \# & \# \\ \# & \# & 3 & 2 & \# \\ \# & \# & \# & 1 & \mathbf{0} \\ \# & \# & \# & 2 & \# \\ \# & \# & \# & \mathbf{3} & \# \end{bmatrix}, t_1 = \begin{bmatrix} 1 & \mathbf{0} & 1 & \# & \# \\ \# & \# & 2 & 3 & \# \\ \# & \# & \# & 4 & \mathbf{5} \\ \# & \# & \# & 5 & \# \\ \# & \# & \# & \mathbf{6} & \# \end{bmatrix}, t_2 = \begin{bmatrix} 7 & \mathbf{6} & 5 & \# & \# \\ \# & \# & 4 & 3 & \# \\ \# & \# & \# & 2 & \mathbf{3} \\ \# & \# & \# & 1 & \# \\ \# & \# & \# & \mathbf{0} & \# \end{bmatrix}$$

Then from these we get the distance array:

$$dist = \begin{bmatrix} 0 & 5 & 3 \\ 5 & 0 & 6 \\ 3 & 6 & 0 \end{bmatrix}$$

Therefore we can now begin our Traveling salesman problem, we start with 101 and can therefore start building up our DP solution.

$$101 \implies \{s, t_2\} = \begin{cases} 1002 \end{cases} = dist[s][t_2] = dist[0][2] = 3$$

$$110 \implies \{s, t_1\} = \begin{cases} 1001 \end{cases} = dist[s][t_1] = dist[0][1] = 5$$

$$111 \implies \{s, t_2, t_1\} = \begin{cases} 1011 \\ 1102 \end{cases} = \begin{cases} 101 + dist[t_2][t_1] = 1002 + dist[1][2] = 3 + 6 = 9 \\ 110 + dist[t_1][t_2] = 1001 + dist[2][1] = 5 + 6 = 11 \end{cases}$$

$$best \implies \{s, t_2, t_1, s\} = \begin{cases} 1011 + d[t_1][0] = 9 + 5 = 14 \\ 1102 + d[t_2][0] = 11 + 3 = 14 \end{cases}$$

Therefore we have the least cost as 14.