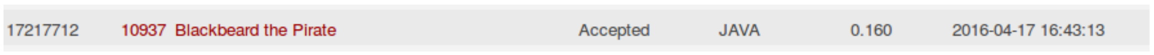
Results from UVA



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

1 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
- $\cdot$ 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’

1.1 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.

1.2 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·**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** ← {-1, 0, 0, 1, 1, -1, -1, 1} and **dr** ← { 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
        $R, C \leftarrow$ number of Rows and Columns respectively
        if R == C == 0 then break;
        Fill map from input and count number of Treasures
        $numTreasures \leftarrow$ 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
            $booleanreachable \leftarrow$ true
            $dist[][] \leftarrow$ init
            for each treasure, t do
                if !bfs($t_x$,$t_y$) then // see Algorithm 4
                    $reachable \leftarrow$ 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
        $charc \leftarrow$ map[row][col]
        if c == '∼' then $map[row][col] \leftarrow$ '#';
        else if c == '∗' then
            $map[row][col] \leftarrow$ '#';
            for i ∈ 0..dc.length do
                $int\ rPrime, cPrime \leftarrow$ row + dr[i], col + dc[i]
                if (rPrime and cPrime in bounds) and (not a camp) then
                    $map[rPrime][cPrime] \leftarrow$ '#'
```

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( )
    $index, tresLoc \leftarrow$ init, boolean start = false, count = 0
    for row ∈ 0..R, col ∈ 0..C do
        if map[row][col] == '@' then
            $start \leftarrow$ true
            $tresLoc[0][0,1] \leftarrow$ row, col
        else if map[row][col] == '!' then
            index.PUT(row*R+col, ++count)
            $tresLoc[count][0,1] \leftarrow$ 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;
```

1.3 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. □

1.4 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 $\mathbf{O(N{\cdot}R{\cdot}C+N{\cdot}2}^N\mathbf{)}$

□

1.5 An Example

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

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

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.

2 Book 7.28: TA Scheduling

2.1 Mathematical Formulation

(a) Given an input of T TA's, S sessions, and parameters a, b, and c where the number of sessions each TA can hold a week is t_i, $a \leq t_i \leq b, \forall i \in 1..T$ and the total number of seesions that can be held a week must be $\leq c$. Determine which TAs cover which sessions if all requirements are met.

(b) Given the above information, add in a density d_i representing the minimum number of sessions that must be held that day of the week. Again determine which TAs cover which sessions if all requirements are met.

2.2 Solution

In both cases we will begin by determining if the number of TA's T can acceed the minimum capacity c by checking $c \geq T \cdot a$. We then construct a circulation digraph max flow representation with $2 + T + S + 2$ nodes: 1 source, 1 virtual source, T TA, S session, 1 virtual sink, and 1 sink node. To ensure that the minimum capacity for each TA is met, we connect the source to each TA node with capacity a, then, to account for this we will connect the source to the virtual source with with a capacity $c - T \cdot a$ and the virtual source will connect to each of the TA nodes with a capacity of $b - a$ so that the total sum of in-capacities for each TA node is $a + b - a = b$. Therefore our total out-degree from the source node will be $T \cdot a + (c - T \cdot a) = c$. Now, according to availablility of the TA's, we connect them to the corresponding session nodes with a capacity of 1. This is the same build for both parts (a) and (b).

For part **(a)**, we will assign edges from each of the sesssion nodes to the virtual sink with capacity 1 to represent that each session can only have 1 TA. Then to ensure that the cycle is complete and that the maximum number of sessions is $\leq c$, we connect the virtual sink with a capacity of c to the sink.

For part **(b)**, we will add in 7 extra nodes, each representing a day with a correspoding value of $d_i, i \in 1..7$. We continue our connections by connecting each seesion, to their correspoding day with a capacity of 1 to represent that each session can only have 1 TA. The for each day, i, we connect it directly to the sink with a capacity of d_i to represent the minimum density that must be met. Next, we connect it to the virtual sink with a capacity of ∞ since they have met the minimum requitrement of d_i. Now we just connect the virtual sink to the sink with a capacity of $c - \sum_{i=1}^{7} d_i$ such that the total in-capacity of the sink is $\sum_{i=1}^{7} d_i + c - \sum_{i=1}^{7} d_i = c$.

Now that we are fully connected we run Ford Fulkerson and if the max flow is $< T \cdot a$ then we have not fufilled the required minimum flow. Otherwise we go through each TA node and check its edges, if the edge has a flow value, then print the TA and the Session information.

For Purposes of simplicity we will show the algorithm for **(b)** as it is part (a) with an additional constaint.

Algorithm 5 Build Network (part b)

```
procedure CONNECT(out, in, capacity)
procedure MAIN(T, S, a, b, c)
    if $c \geq T \cdot a$ then Not Possible To Compute
    $G \leftarrow$ initialize nodes
    for each TA node $t \in 1..T$ do
        G.CONNECT(source, t, a)
        G.CONNECT(cSource, t, (b-a))
        for each session, $s \in 1..S$, that $t$ can lead do
            G.CONNECT(t, s, 1)
    CONNECT(source, vSource, $c - T \cdot a$)
    for each session, $s \in 1..S$ do
        $d \leftarrow$ day that $s$ occurs on
        G.CONNECT(s, d, 1)
    for each day, $i$ do
        $d_i \leftarrow$ minimum sessions day $i$ must have
        G.CONNECT(i, sink, $d_i$)
        G.CONNECT(i, vSink, $\infty$)
    G.CONNECT(vSink, sink, $c - \sum_{i=1}^{7} d_i$)
    G.MAXFLOW( )
    for edge, $e$, from source to not vSource do
        if e.flowValue() != a then Not Possible To Compute
    for each day, $i$ do
        $d_i \leftarrow$ minimum sessions day $i$ must have
        edge, $e$, from $i$ to sink
        if e.flowValue() != $d_i$ then Not Possible To Compute
    for each TA node $t \in 1..T$ do
        for edge, $e$, from t do
            if e.flowValue() == 1 then
                session $s \leftarrow$ e.to()
                Print $t$ and $s$
```

2.3 Correctness

Proposition 4.

This algorithm will determine (if one exists) a valid schedule for office hours, specifying which TA will cover which time slots. If one does not exist, then it will report so.

Proof.

Above we have described how to construct the flow network in detail. As so we show now that this supports the constraints set forth by the question. The first constraint is that each TA must have a minimum of a office hours and a maximum of b each week. It is simple to upper bound each TA by b by ensuring that the sum of the maximum capacity of all edges coming into each TA is equal to b. Therefore the only thing we need to constrain is that we have at least a for each TA. We acheive this by making an edge of maximum capacity a between the source and each TA, which will be filled out first. Similarly we will have a "whatever we have left" node which we call the virtual source. The connections from the virtual source, we connect to each TA node with edges with max capacity of $b - a$ to ensure that each node has and input of b.

Now to account for the second constraint that at most c office hours may occur each week. We do this by making the maximum capacity of all edges leaving the source and entering the sink to be c. Therefore (since $T \cdot a$ has already left the source going to each TA), we connect the source to the virtual source with a value of $c - T \cdot a$.

The last constraint of the problem is that each day i have a specific density d_i that acts as a minimum value. To account for this, we have each Session connect to its respective day with a max capacity of 1 so that only one TA can be present at each session. Then from each day, we will connect directly to the sink with maximum capacity d_i therefore we need to adjust our edge capacity from the virtual sink to the sink to be $c - \sum d_i$ and connect each day to the virtual sink with max capacity ∞.

Now we have shown that each constraint has been accounted for, but to ensure that each one has held up, we simply will check the flow values of each of the edges from the source to the TA nodes are full and those from each day to the sink are full. □

2.4 Analysis

For this section we will use the following variables: T for number of TAs, S for number of sessions, C for number of connections between TAs and Sessions, $V = T + S + 7 + 4$ for total number of nodes and $E = 2 \cdot T + C + 2 \cdot S$ for total number of edges. We say then that we can make our calculations based off V + E.

Proposition 5. *The* <u>*space complexity*</u> *of this algorithm is* ***O(V+E)***

Proof.

This is due to the fact that we only store the nodes and the connections between them in a datastructure and the rest is stored in constant variables. □

Proposition 6. *The time complexity of this algorithm is* ***O(T·S + MaxFlowValue·E + C)***

Proof.

We can break this down into the loops which we have described in our algorithm:

1. TA Node Connection: we go through each TA node (T) and connect it to the source and virtual source (2) and then go through every session and see if it is a valid time for each TA to work (S) $\implies T \cdot (2 + S) \implies O(T \cdot S)$
2. Session Node: we have already connected each session to its corresponding TAs, now we just connect each one (S) to the specific day that it belongs to (1) $\implies O(S)$
3. Days: when we have to connect each day to the sink and virtual sink $(7 \cdot 2)$ which is considered arbitrary in the large cases $\implies O(1)$
4. MAX FLOW: Known to be $O(MaxFlowValue \cdot E)$
5. Check TAs and days: we check that each TA and day have met the minimum requirements $\implies O(T)$
6. Print out: Go through every edge going from TA to Session nodes $\implies O(C)$

Thus summing to be $T \cdot S + S + 1 + MaxFlowValue \cdot E + T + C$ which in Big-O becomes **O(T·S + MaxFlowValue·E + C)** □

3 Book 8.4: Resource Allocation

Mathematical Formulation

Given an input of n processes, $P = \{p_1, p_2, ..., p_n\}$, and a set of m resources, $R = \{r_1, r_2, ..., r_m\}$, each process requires a set of resources $R^*, R^* \subseteq R$. Each process in required active iff every resource $r \in R^*$ is allocated to it, however each resource can only be used once. Given a number k>0, determine if resources $\in R$ can be allocated so that at least k processes $\in P$ will be active. For the following cases, give a polynomial algorithm or proove it is NP-Complete.

(a) General Case i.e. k>0

(b) $k = 2$

(c) Have 2 types of resources, if either one is fufilled for a process, then the process is considered active

(d) Each resource can be allocated a maximum of 2 times

3.1 Part (a)

Proposition 7. *This is an NP-Complete problem.*

Proof.
We show first that this problem is NP. Given a set of k processes, check to see if there are resources shared between them all which take $k \cdot m^* = totalResources$ time where m^* is the average number of resources all of the processes contain. More specifically we can create a HashSet which we will add each resource as we see them in each set. Therefore we will check to see if the resource is already present in the set, if it is then there is a repetition; if we can get to the end, then there are no more.

Now we say that this is an NP-Hard problem by doing a reduction from the Independent Set problem i.e. $IndependentSet \leq_P Problem$. Assume that we have an independent set of graph G and the number k. We take the verticies, $V \in G$ to represent each process and all edges $E \in G$ to be resources. Therefore verticies are only considered to be adjacent if they have a common resource. So if $\underline{\text{there exists an independent set}}$, $G^* \in G$ s.t. $|G^*| \geq k$, we say that this implies that verticies within G^* have no common edges, i.e. there are no shared resources $\implies$ solved the problem. Now if $\overline{\text{there exist k processes}}$ which resources from a disjoint set, then the corresponding graph would have no edges in common for these processes $\implies$ independent set.

Since Independent Set is NP-hard $\implies$ this problem is at least NP-Hard

$$\therefore \text{this process is both NP and NP-Hard} \implies \text{NP-Complete.}$$

□

3.2 Part (b)

This can be solved in polynomial time by brute force since there only exist n^2 process combinations. Therefore doing the above mentioned algorithm for checking if sets are indeed valid, we can go through all pairs; if at any pair they have a disjoint set of resources, then we accept, if we go through every possible combination and never get into an accepting state, we will say it is impossible.

3.3 Part (c) and Part (d)

These are seen as more specialized versions of problem (a), therefore we can can generalize and say that these will also be NP-Complete.