CS 180 Practice Questions

Sudharsan Krishnaswamy

March 13, 2018

Problem 1

We have ‘n’ samples of DNA, each belonging to one of the 2 species A and B. We would like to divide the ‘n’ samples into 2 groups - those that belong to A and that belong to B, but it’s very hard to identify the true species of the individual samples. So, we use the following approach: For each pair of samples ‘i’ and ‘j’, we carefully study them side by side. If we’re confident enough in our judgment, we label them either ‘similar’ or ‘different’. Those pairs for which we are not confident enough are labeled ‘ambiguous’. So now we have a collection of ‘n’ samples, as well as a collection of ’m’ judgments (either ‘same’ or ‘different’) for the pairs that were not labeled ‘ambiguous’. We’d like to know if this data is consistent with the idea that each sample is from one of species ‘A’ or ‘B’. More concretely, we’ll declare the ‘m’ judgments to be consistent if it is possible to label each sample as either ‘A’ or ‘B’ in such a way that for each pair (i,j) labeled ‘same’, they have the same final label; and for each pair (i,j) labeled ‘different’, they have different labels. Can you come up with an O(m+n) algorithm that determines whether a given set of ‘m’ judgments are consistent?!

Problem 2

You are given an array of ‘n’ jobs. Each job ‘i’ requires b_i units of work to be done. There are K identical assignees available and you are also given an integer ‘T’, which is how much time an assignee takes to do one unit of a job (Note: An assignee takes ‘T’ time units to complete a single unit of any job, so the i^{th} job takes b_iT time units in total). Come up with an efficient algorithm to find the minimum time to finish all ‘n’ jobs with the following constraints:

1. An assignee can be assigned only contiguous jobs. For example, an assignee cannot be assigned jobs 1 and 3, but not 2.
2. Two assignees cannot share (or be co-assigned) a job (i.e.) a job cannot be partially assigned to one assignee and partially to another.

Problem 3

Given a undirected, connected, positive-weighted graph G=(V,E), we all know what a spanning tree of this graph is. Given a spanning tree T of G, we define a bottleneck edge to be an edge of T with the greatest weight/cost. A spanning tree T of G is a minimum-bottleneck spanning tree if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) Is every minimum-bottleneck spanning tree of a graph G a minimum spanning tree of G? Prove or give a counterexample.

(b) Is every minimum spanning tree of G a minimum bottleneck spanning tree of G? Prove or give a counter-example.

Problem 4

Given a n x n board where 'n' is of the form 2^k where $k \geq 1$ (i.e.) 'n' is a power of 2, with a minimum value equal to 2. The board has one missing cell (of size 1 x 1). Your task is to fill the board using L shaped tiles. An L shaped tile is a 2 x 2 tile with 1 cell of size 1 x 1 missing. Its illustrated in the figure below:

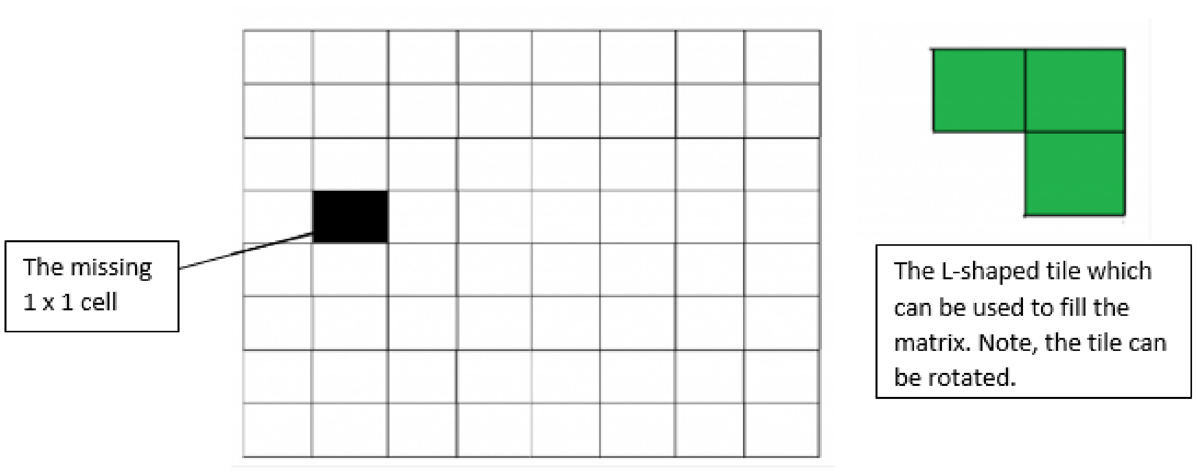

The missing 1 x 1 cell
The L-shaped tile which can be used to fill the matrix. Note, the tile can be rotated.

Figure 1: Problem 5-Question

Come up with a systematic algorithm for this problem.

Problem 5

In mid-term, we solved the problem of Longest Common Substring (LCS). Here, we try to solve a slightly different version of the same problem, known as the Longest Common Subsequence (also LCS!). By definition, a subsequence of a string 's' is a sequence of characters of 's' which need not be contiguous in 's', but has the same relative order. In other words, a subsequence of a string 's' is formed by removing some characters from 's'. For example, if our string is "ababc" , then "aa" , 'abc', "bc" , "ac", "c" are all valid subsequences, while "cb", "cba" are not. Given 2 strings s_1 and s_2, can you come up with an algorithm to find the length of their longest common subsequence in O(mn) time complexity?

Extension: In addition to s_1, s_2, you are also given an integer 'K'. Find the length of the longest common subsequence formed by consecutive segments of length at least 'K'. (i.e.) the matching portions of the subsequence must be of size at least 'K'.

Problem 6

To get in shape, you have decided to start running to work. You want a route that goes entirely uphill and then entirely downhill so that you can work up a sweat going uphill and then get a nice breeze at the end of your run as you run faster downhill. Your run will start at home and end at work and you have a map detailing the roads with 'm' road segments (any existing road between two intersections) and 'n' intersections. Each road segment has a positive length, and each intersection has a distinct elevation.

(a) Assuming that every road segment is either uphill or downhill, give an efficient algorithm to find the shortest route that meets your specifications.

(b) Give an efficient algorithm to solve the problem if some roads may be level (i.e., both intersections at the end of the road segments are at the same elevation) and therefore can be taken at any point.

Problem 7

We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed only one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container. In addition to coming up with such a sequence of operations, we also wish to perform the least number of such operations. Can you come up with an efficient algorithm for finding the shortest sequence?

Problem 8

Give an efficient algorithm which takes as input a directed graph G = (V, E), and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable.

Problem 9

You are given a directed graph with (possibly negative) weighted edges, in which the shortest path between any two vertices is guaranteed to have at most k edges. Give an algorithm that finds the shortest path between two vertices 'u' and 'v' in O(k$|E|$) time.

Problem 10

You are given a convex polygon P of ‘n’ vertices in a plane (specified by their x and y coordinates). A triangulation of P is a collection of n-3 diagonals of P such that no two diagonals intersect. Cost of a triangulation is given by the sum of lengths of diagonals in it. In this problem, you are asked to come up with an algorithm for finding a triangulation of minimum cost.

Problem 11

Alice wants to throw a party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five other people whom they dont know.

(a) Give an efficient algorithm that takes as input the list of ‘n’ people and the list of pairs who know each other and outputs the best choice of party invitees. (Hint: Try to identify people who should not be invited.)

(b) Prove the correctness of your algorithm from part (a).

Problem 12

A server has ‘n’ customers waiting to be served. The service time required by each customer is known in advance: it is t_i minutes for customer ‘i’. So if, for example, the customers are served in order of increasing ‘i’, then the i^{th} customer has to wait $\sum_{j=1}^{i} t_j$ minutes. We wish to minimize the total waiting time,

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer i})$$

Describe an efficient algorithm for computing the optimal order in which to process the customers and give a brief justification of its running time and correctness.

Problem 13

A subsequence is palindromic if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is not palindromic). Devise an algorithm that takes a sequence x[1,..,n] and returns the (length of the) longest palindromic subsequence. Its running time should be O(n^2).

Problem 14

An edge of a flow network is called critical if decreasing the capacity of this edge results in a decrease in the maximum flow. Give an efficient algorithm that finds a critical edge in a network.

Solutions

1) This is a famous 2-colorability problem. We first start with an undirected graph G=(V,E) as follows: Every sample is a vertex. There is an edge between 2 vertices if there is an unambiguous judgment involving the corresponding samples.
For each component of G we do as follows: Arbitrarily pick a vertex 'u' and label it 'A'. We start a BFS from 'u' and visit all its neighbours 'v' in G. If the edge (u,v) we take is labeled 'same', then we label 'v' with the same label as 'u' (i.e.) 'A'. And we repeat the same process until we exhaust all the vertices (in all the components). If there is any consistency while labeling the vertices (for eg. we may wish to label a vertex 'v' as 'A', but it was already labeled as 'B' or vice versa), then the given set of judgments are inconsistent.

Complexity Analysis: BFS takes O(n+m). The proof of correctness lies in the fact that the above way of labeling is the only way to label the vertices with the only exception of inverting the labels.

2) The idea is to use Binary Search. Say, we have a function 'A' that tells us if its possible to finish all jobs within a given time and a given number of assignees. We can solve this problem by doing a binary search for the answer. If the middle point of binary search is not possible, then search in second half, else search in first half. Lower bound for Binary Search for minimum time can be set to 0. The upper bound can be obtained by adding all given job times. So, how do we come up with such a function 'A'?! This function can be implemented using a Greedy Approach. Since we want to know if it is possible to finish all 'n' jobs within a given time, we traverse through all jobs and keep assigning jobs to the current assignee one by one while a job can be assigned within the given time limit. When time taken by the current assignee exceeds the given time (provided by Binary Search), create a new assignee and start assigning jobs to him/her. If the number of assignees needed is more than 'k', then return false, else return true.

3) The first statement is false while the second one is true. This can be proven as follows:

(a) Let G = (v_1, v_2, v_3, v_4) be a graph with edges between every pair of vertices. The weight of the edge (v_i,v_j) is (i+j). Then every spanning tree has a bottleneck edge of weight at least '5', so the tree with a path $v_2 - v_3 - v_1 - v_4$ is a minimum bottleneck spanning tree. But our minimum spanning tree is the one with edges from v_1 to every other vertex.

(b) This is true. Say T is a minimum spanning tree and let T$'$ be another spanning tree with a lighter bottleneck edge. This means, there exists an edge 'e' in T, which is heavier than all the edges present in T'. If the endpoints of 'e' are (u,v), then the role of 'e' in T is to join the subtrees containing 'u' and 'v'. There must exist an edge e$'$ in T$'$, which plays a similar role. But since e$'$ is lighter than 'e', T-e+e$'$ yields a spanning tree of weight smaller than that of T, which is a contradiction.

4) This is a classical example of Divide and Conquer.
Base Case: k = 1. A 2 x 2 square with one cell missing can be easily filled with a single L-shaped tile of appropriate orientation.
Induction: Say, we know how to fill a matrix of size 2^{k-1} x 2^{k-1}. Given a matrix of size 2^k x 2^k, place an L-shaped tile at the center of the matrix such that it does not cover the (n/2) x (n/2) sub-square that has the missing square. Now all four sub-squares of size (n/2) x (n/2) have a missing cell each (a cell that doesn't need to be filled), which can be solved using induction. See Figure 2 for clarity.

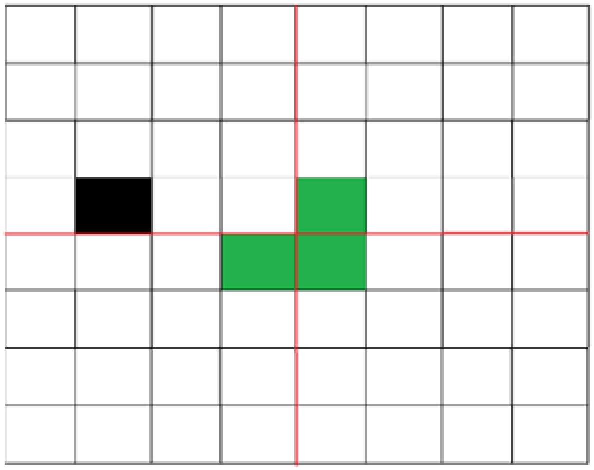


Figure 2: Problem 4 - Solution

5) This problem can be solved using Dynamic Programming as follows: Let LCS[i][j] = Length of the longest common subsequence of s_1[1,2...i] and s_2[1,2,...j]. There are only 2 cases.

Case 1: What if $s_1[i] \neq s_2[j]$? Our LCS cannot end at indexes 'i' and 'j' of s_1 and s_2. Therefore,

$$LCS[i][j] = \max(LCS[i-1][j], LCS[i][j-1])$$

Case 2: If $s_1[i] = s_2[j]$, we can have our LCS ending at indexes 'i' and 'j' of s_1 and s_2. Therefore,

$$LCS[i][j] = \max(LCS[i-1][j-1] + 1, LCS[i-1][j], LCS[i][j-1])$$

Extension: Let LCS_K[i][j] = Length of the longest common subsequence of s_1[1,2...i] and s_2[1,2,...j] made of consecutive segments of length at least 'k'. We maintain an auxiliary matrix cnt[i][j], which stores the length of the longest common substring of s_1 and s_2 ending at indexes 'i' and 'j' of strings s_1 and s_2 respectively (i.e.) the largest 'l' such that s_1[i-l+1,..,i] = s_2[j-l+1,..,j]. We can notice that the 'cnt' matrix can be calculated by Dynamic Programming as follows: cnt[i][j] = (cnt[i-1][j-1] + 1) if s_1[i] = s_2[j], and it is 0 if $s_1[i] \neq s_2[j]$. Now, the calculation of LCS_K[i][j] belongs to one of the 2 cases:

Case 1: What if cnt[i][j] < K? This means LCS_K[i][j] will never end at indices 'i' and 'j' of s_1 and s_2. Therefore, we have

$$LCS_K[i][j] = \max(LCS_K[i-1][j], LCS_K[i][j-1])$$

Case 2: When $cnt[i][j] \geq K$, we may have our LCS_K[i][j] ending at indexes 'i' and 'j'. But if that happens, our LCS_K[i][j] must have a suffix s1[i-t+1,..,i] (= s2[j-t+1,..,j]) for some t: $K \leq t \leq cnt[i][j]$. Thus,

$$LCS_K[i][j] = \max_{K \leq t \leq cnt[i][j]} (LCS_K[i-t][j-t] + t)$$

6)

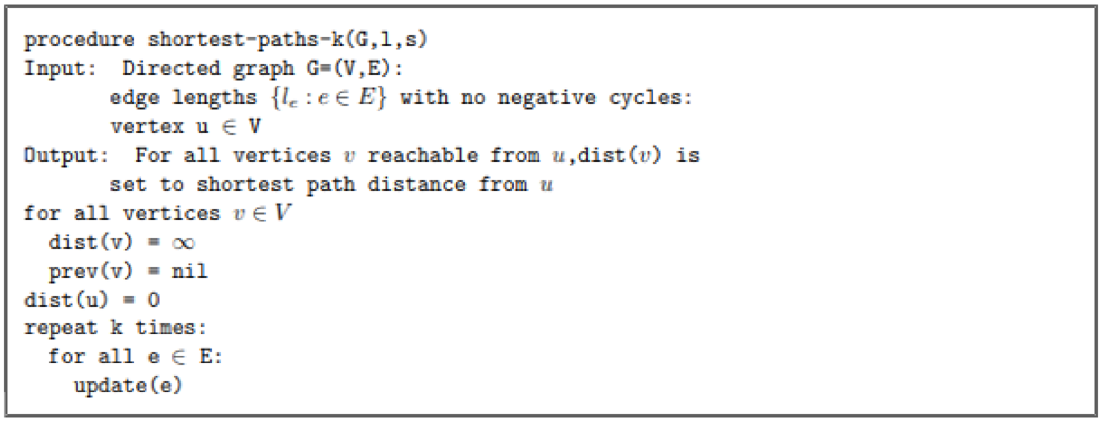
(a) Dijkstra's algorithm solves the single source shortest-paths problem on a general graph with non-negative edge weights in - O(m + n logn) time. In this problem we can actually do better and solve it in - O(m+n) time. The difference is that we must go uphill before we go downhill. With this constraint we know we have somewhere along the optimal path there will be a highest point. Call it 'v'. A consequence of this path being optimal is that there exist no other points for which the length of the best uphill path from home to the point plus the length of the best downhill path from the point to work is shorter than the best paths to and from v. So if we could find the best uphill path to each point and the best downhill path from each point, we can do a linear scan through the points to find the one with the smallest sum and this tells us the optimal path. Now we just have two subproblems of finding the single source shortest uphill paths to each point and the single goal shortest paths downhill from each point. Consider the uphill problem. We can solve this by throwing away all downhill edges, then because the path always moves uphill there can no longer be any cycles (since there are no level edges). Therefore we are dealing with a DAG and we know how to find the single source shortest paths in a DAG in linear time. Basically this involves finding a topological ordering on the vertices and then just computing the best paths in order. We can similarly solve the downhill problem. The total run time is - O(m) to produce each DAG. Then we solve two single source shortest path in a DAG problems which each run in - O(m+n) time. Finally, traversing the vertices to find the optimal peak vertex takes - O(n) time. Thus the total time is - O(m+n).

(b) In this case, we can no longer replace each edge with a directed edge that assures us that no cycles exist. Therefore our best method in this general case is to use Dijkstras algorithm after transforming the graphs as above replacing each level road with two directed arrows.

7) We can model this as a graph problem. We represent each potential state of the three containers with an ordered pair (A,B), where A represents the amount of water in the 7pint container and B represents the amount of water in the 4pint container. Hence the initial state would be (7,4). Note that we do not need an ordered triple as we know the total amount of water in all three containers (11 pints) so the total state is completely determined by water in the 7 and 4 pint containers. Hence we have 5 x 8 = 40 possible states for the system. We model these states as nodes in a directed graph and draw an edge from one node to another node if we can go from the configuration represented by one node to the configuration represented by another node through one pour. For example, there would be an edge from (7,4) to (0,4) but no edge from (1,3) to (5,1). Hence if we can find a path from one node to another, then we can go from one state to another through a series of pourings. Now, the problem reduces to finding the shortest path from (7,4) to (2,a) or (b,2) where 'a' can be any number from 0 to 5 and 'b' can be any number from 0 to 8. This can be solved using BFS in O(m+n) time.

8) We first find the strongly connected components in G. This can be done in O(m+n) using Kosaraju's algorithm. If we then shrink each component into a single node, we get a directed acyclic graph (Why?). There exists a vertex from which all others can be reached if and only if there is exactly one vertex in the DAG with in-degree 0. This is the vertex you're looking for - the so-called "mother vertex".

9) The algorithm we present here is a variant of Bellman-Ford algorithm.



```
procedure shortest-paths-k(G,l,s)
Input:  Directed graph G=(V,E):
       edge lengths $\{l_e : e \in E\}$ with no negative cycles:
       vertex u $\in$ V
Output:  For all vertices $v$ reachable from $u$,dist($v$) is
       set to shortest path distance from $u$
for all vertices $v \in V$
  dist(v) = $\infty$
  prev(v) = nil
dist(u) = 0
repeat k times:
  for all e $\in$ E:
    update(e)
```

Figure 3: Problem 9 - Pseudocode

Complexity Analysis: The algorithm is a modification of Bellman-Ford with shortest path lengths known as k from the problem definition. Complexity of the algorithm is O(k$|E|$).

10) We label the vertices of P by 1,...,n starting from an arbitrary vertex and walking clockwise. For $1 \leq i < j \leq n$, we denote a subproblem A[i][j] which computes the minimum cost triangulation of the polygon spanned by vertices i,i+1,...,j. Formulating the subproblem this way we are enumerating all the possible diagonals such that the diagonals do not cross each other. Since we are only considering in clockwise directions, diagonals generated by the subproblems can not cross each other. Recursive definition of A[i][j] is as follows:

$$A[i][j] = \min_{i \leq k \leq j} (A[i][k] + A[k][j] + d_{i,k} + d_{k,j})$$

where

$$d_{i,j} = \begin{cases} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, & \text{if } j - i \geq 2 \\ 0, & \text{otherwise} \end{cases}$$

The complexity of the above algorithm is O(n^3)

11) The trick is to notice that people who know less than five other people can not be part of the subset Alice wants, nor can people who know at least n - 5 other people. So the algorithm is the following:

```
Algorithm PickSubset(S)

while an element x of S knows fewer than 5 or more than |S| − 5 people
    remove x from S
endwhile

return S
```

Note that when we write "knowing fewer than 5 or more than $|S| - 5$", we mean with respect to the current set S, not the set the algorithm started with.

Proof of correctness: Let S_i denote the set S at the end of the i^{th} iteration of the while loop. We prove by induction on 'i' that the optimal solution to Alice's problem is a subset of S_i.
The base case is when $i = 0$, before the first iteration of the while loop. At that point, S_0 contains every person Alice has to choose from, and hence contains every member of the optimal solution.
Let us now consider the induction step. Suppose the statement was true after the i^{th} iteration of the loop, and consider now the set $S_i + 1$. The person 'x' removed from S_i to obtain $S_i + 1$ either knows fewer than 5 other people in S_i, which means he/she does not know enough people in the optimal subset to be invited, or he/she knows more than $|S| - 5$ people in S_i, which means there are not enough people in the optimal subset that he/she does not know. In both cases 'x' can not be part of the optimal solution, and hence the optimal solution must also be a subset of $S_i + 1$.
This completes the induction step. Now, when the while loop finally exits after 't' iterations, every person in S_t knows at least 5 other people, and there are at least 5 other people that he/she does not know. Hence S_t is a possible solution to Alice's problem. Because we proved that the optimal solution is a subset of S_t, this means that S_t is the optimal solution we were looking for.

12) We first observe that no matter what order we choose to serve the customers in, the total time taken does not change. The total time (let's call it X) required to serve all of the customers is always the sum of their service times. Thus, the total waiting time can be reformulated as:

$$T = \sum_{i=1}^{X} (\text{number of customers still waiting at time i})$$

Thus, we wish to minimize the number of customers that are still waiting at any given time. This means we want to serve the customers that require less time first, so our optimal order is the customers sorted by increasing service time (which could be proved by induction on X). Sorting takes O(n logn) time.

13) Definition: Sequence 'x' is called a palindrome if we form a sequence $\bar{x}$ by reversing the sequence 'x', and x = $\bar{x}$.
This definition allows us to formulate an optimal substructure for the problem. If we look at the sub-string x[i,..,j] of the string 'x', then we can find a palindrome of length at least 2 if x[i] = x[j]. If they are not same then we seek the maximum length palindrome in subsequences x[i+1,...,j] and x[i,...,j-1]. Also every character x[i] is a palindrome in itself of length 1. Let us define the maximum length palindrome for the substring x[i,...,j] as L[i][j]. Then, the recursive definition of L[i][j] as follows:

$$L[i][j] = \begin{cases} L[i+1][j-1]+2, & \text{if } x[i] = x[j] \\ \max(L[i][j-1], L[i+1][j]), & \text{otherwise} \end{cases}$$

$$L[i][i] = 1 \qquad \forall i \in (1, 2..n)$$

The complexity of the above solution is O(n^2) for obvious reasons.

14) Algorithm: First, run Ford-Fulkerson's max flow algorithm. If an edge is critical, it must be filled to capacity, otherwise it would be possible to reduce its capacity to the amount of flow going through it without affecting the maximum flow. Then, for each edge (u,v) filled to capacity in the residual graph, run DFS and see if there is a path from u to v. If there isnt, then that edge is a critical edge.

Proof of correctness: If an edge is not full when max flow is achieved, it is possible to reduce its capacity to the amount of flow going through it without affecting the maximum flow. By running the Ford-Fulkerson algorithm, we can determine which edges are full when the maximum flow is achieved.
It is not enough for the edge to be full, however. We must check that the flow cannot be re-routed along another path in the residual graph. Suppose (u,v) is an edge that is full after the Ford-Fulkerson algorithm is run. If there is a path from u to v not using (u,v), then if the amount of flow allowed through (u,v) is decreased by 1, then the flow can just be routed from u to v through the other path, without affecting the max flow. If there is no path from u to v , then there is no way to re-route the flow that would have gone through (u,v), so the maximum flow must decrease. Therefore, full edges are only critical edges if there does not exist any path from u to v. Running DFS finds if there exists a path between u and v. Remember, the entire process is done on the residual graph obtained after max-flow algorithm.

Complexity Analysis: To run this algorithm, first you must run the Ford-Fulkerson algorithm, which takes O(nm^2). Next, you must run DFS on at most O(n) vertices, taking O(n(n+m)) time. This term is dominated by the running time of the Ford-Fulkerson algorithm, so the overall running time is O(nm^2).