CS 170 HW 3

Due on 2017-02-11, at 11:59 pm

1 (★) Study Group

Please note that this homework is due 24 hours earlier than usual.

List the names and SIDs of the members in your study group.

2 (★★★) Disrupting a Network of Spies

Let $G = (V, E)$ denote the "social network" of a group of spies. In other words, G is an undirected graph where each vertex $v \in V$ corresponds to a spy, and we introduce the edge $\{u, v\}$ if spies u and v have had contact with each other. The police would like to determine which spy they should try to capture, to disrupt the coordination of the group of spies as much as possible. More precisely, the goal is to find a single vertex $v \in V$ whose removal from the graph splits the graph into as many different connected components as possible. This problem will walk you through the design of a linear-time algorithm to solve this problem. In other words, the running time will be $O(|V| + |E|)$. In the following, let $f(v)$ denote the number of connected components in the graph obtained after deleting vertex v from G. Also, assume that initial graph G is connected (before any vertex is deleted) and is represented in adjacency list format.

Prove your answer to each part (some parts are simple enough that the proof can be a brief justification; others will be more involved).

(a) Perform a depth-first search starting from some vertex $r \in V$. How could you calculate $f(r)$ from the resulting depth-first search tree in an efficient way?

(b) Suppose $v \in V$ is a node in the resulting DFS tree, but v is not the root of the DFS tree (i.e., $v \neq r$). Suppose further that no descendant of v has any non-tree edge to any ancestor of v. How could you calculate $f(v)$ from the DFS tree in an efficient way?

(c) Definition: If w is a node in the DFS tree, let $\mathsf{up}(w)$ denote the depth of the shallowest node y such that there is some graph edge $\{x, y\} \in E$ where either x is a descendant of w or $x = w$. We'll define $\mathsf{up}(w) = \infty$ if there is no edge $\{x, y\}$ that satisfies these conditions.

Now suppose v is an arbitrary non-root node in the DFS tree, with children $w_1, \ldots, w_k$. Describe how to compute $f(v)$ as a function of k, $\mathsf{up}(w_1), \ldots, \mathsf{up}(w_k)$, and $\mathsf{depth}(v)$.

Hint: Think about what happened in part (b); think about what changes when we can have non-tree edges that go up from one of v's descendants to one of v's ancestors; and think about how you can detect it from the information provided.

(d) Design an algorithm to compute $\mathsf{up}(v)$ for each vertex $v \in V$, in linear time.

(e) Describe how to compute $f(v)$ for each vertex $v \in V$, in linear time.

Solution:

(a) $f(r) =$ the number of children of r.

Proof: The depth-first search will only return to the root node if it has completely explored the subtree rooted at each node. Therefore, the subtrees rooted at each child are unconnected, and removing r will disconnect each of them from the others.

(b) $f(v) = 1+$ the number of children of v.

Proof: This case differs from the previous one only in that there's another component, corresponding to the ancestors of v. The lack of non-tree edges from descendants to ancestors means that removing v disconnects every child of v from every ancestor of v. And as before, the children are all partitioned from each other by removing v.

(c) Let N denote the number of children c of v with the property that $\mathsf{up}(c) \geq \mathsf{depth}(v)$, i.e., $N = |\{c : c \text{ is a child of } v \text{ and } \mathsf{up}(c) \geq \mathsf{depth}(v)\}|$. Then $f(v) = N + 1$.

Proof: If we show that a child c has $\mathsf{up}(c) < \mathsf{depth}(v)$ iff c is connected to an ancestor of v by a route that excludes v, then the proof follows directly from (b) because these are the children we are overcounting as they in the same connected component as r.

If c is connected to a proper ancestor a of v by a route that excludes v then $\mathsf{up}(c) \leq \mathsf{depth}(a) < \mathsf{depth}(v)$. Conversely, if $\mathsf{up}(c) < \mathsf{depth}(v)$, then there is an edge from a descendant d of v to a vertex in the same connected component as r as the depth first search only returns to a vertex if it has completely explore the subtree rooted at the vertex.

(d) By definition, $\mathsf{up}(v)$ is the minimum of v's neighbors' depths, and the ups of v's descendants. Formally,

$$\mathsf{up}(v) = \min\left(\min\{\mathsf{depth}(w) : \{v, w\} \in E\},\ \min\{\mathsf{up}(w) : w \text{ is a child of } v\}\right).$$

We can thus compute $\mathsf{up}(v)$ by traversing the DFS tree bottom-up.

For a leaf, $\mathsf{up}(v)$ can be computed by minimizing over the depth of all neighboring vertices.

This can be computed in linear-time as each vertex and edge is considered a constant number of times.

(e) This follows immediately from parts (c)–(e). Pick some node as the root. Then compute $\mathsf{up}(\cdot)$ and $\mathsf{depth}(\cdot)$ at each node. Then, compute the function defined in part (d).

The running time is $\Theta(|V| + |E|)$. We needed to make three passes over the graph, one to compute $\mathsf{depth}(\cdot)$, one to compute $\mathsf{up}(\cdot)$, and a third to compute $f(\cdot)$. In each pass, we process each vertex once, and each edge at each vertex. This is $\Theta(|V| + |E|)$ for each pass, and $\Theta(3|V| + 3|E|) = \Theta(|V| + |E|)$.

Note that in a practical implementation, some of these separate passes could be combined, without affecting the asymptotic complexity of the algorithm.

3 (★★★) Inverse FFT

Recall that in class we defined M_n, the matrix involved in the Fourier Transform, to be the following matrix:

$$M_n = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

For the rest of this problem we will refer to this matrix as $M_n(\omega)$ rather than M_n.

(a) Define

$$M_n(\omega^{-1}) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

Recall that $\omega^{-1} = 1/\omega = \bar{\omega} = \exp(-2\pi i/n)$.

Show that $(1/n)M_n(\omega^{-1})$ is the inverse of $M_n(\omega)$, i.e. show that

$$(1/n)M_n(\omega^{-1})M_n(\omega) = I$$

where I is the $n \times n$ identity matrix– the matrix with all ones on the diagonal and zeros everywhere else.

(b) Suppose we have a polynomial $C(x)$ of degree at most $n-1$ and we know the values of $C(1), C(\omega), \dots, C(\omega^{n-1})$. Explain how we can use $M_n(\omega^{-1})$ to find the coefficients of $C(x)$.

(c) Show that $M_n(\omega^{-1})$ can be broken up into four $n/2 \times n/2$ matrices in almost the same way as $M_n(\omega)$. Specifically, suppose we rearrange columns of M_n so that the columns with an even index are on the left side of the matrix and the columns with an odd index are on the right side of the matrix (where the indexing starts from 0). Show that after this rearrangement, $M_n(\omega^{-1})$ has the form:

$$\begin{bmatrix} M_{n/2}(\omega^{-1}) & \omega^{-j} M_{n/2}(\omega^{-1}) \\ M_{n/2}(\omega^{-1}) & -\omega^{-j} M_{n/2}(\omega^{-1}) \end{bmatrix}$$

As in class, the notation $\omega^{-j} M_{n/2}(\omega^{-1})$ is used to mean the matrix obtained from $M_{n/2}(\omega^{-1})$ by multiplying the j^{th} row of this matrix by ω^{-j} (where the rows are indexed starting from 0). You may assume that n is a power of two.

Solution:

(a) We need to show that the entry at position (j, k) of $M_n(\omega^{-1})M_n(\omega)$ is n if $j = k$ and 0 otherwise. Recall that by definition of matrix multiplication, the entry at position (j, k) is (where we are indexing the rows and columns starting from 0):

$$\begin{aligned}\sum_{l=0}^{n-1} M_n(\omega^{-1})_{jl} M_n(\omega)_{lk} &= \sum_{l=0}^{n-1} \omega^{-lj}\omega^{kl} \\ &= \sum_{l=0}^{n-1} \omega^{-lj+kl} \\ &= \sum_{l=0}^{n-1} \omega^{l(k-j)}\end{aligned}$$

If $j = k$ then this just becomes

$$\begin{aligned}\sum_{l=0}^{n-1} \omega^{0\cdot l} &= \sum_{l=0}^{n-1} \omega^0 \\ &= \sum_{l=0}^{n-1} 1 \\ &= n\end{aligned}$$

On the other hand, if $j \neq k$ then $\omega^{k-j} \neq 1$ so we can use the formula for summing a geometric series, namely

$$\sum_{l=0}^{n-1} \omega^{l(k-j)} = \frac{1-\omega^{n(k-j)}}{1-\omega^{k-j}}$$

Now recall that since ω is an n^{th} root of unity, ω^{nm} for any integer m is equal to 1. Thus the expression above simplifies to

$$\frac{1-1}{1-\omega^{k-j}} = 0$$

Here's another nice way to see this fact. Observe that we can factor the polynomial $X^n - 1$ to get

$$X^n - 1 = (X-1)(X^{n-1} + X^{n-2} + \ldots + X + 1)$$

Now observe that ω^{k-j} is a root of $X^n - 1$ and thus it must be a root of either $X - 1$ or $X^{n-1} + X^{n-2} + \ldots + X + 1$. And if $k \neq j$ then $\omega^{k-j} \neq 1$ so it cannot be a root of $X - 1$. Thus it is a root of $X^{n-1} + X^{n-2} + \ldots + X + 1$, which is equivalent to the statement that $\omega^{(n-1)(k-j)} + \omega^{(n-2)(k-j)} + \ldots + \omega^{k-j} + 1 = 0$, which is exactly what we were trying to prove.

(b) Let $c_0, \ldots, c_{n-1}$ be the coefficients of $C(x)$. Then as we saw in class,

$$M_n(\omega)\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1}\end{bmatrix} = \begin{bmatrix} C(1) \\ C(\omega) \\ \vdots \\ C(\omega^{n-1})\end{bmatrix}$$

Thus

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = M_n(\omega)^{-1} \begin{bmatrix} C(1) \\ C(\omega) \\ \vdots \\ C(\omega^{n-1}) \end{bmatrix}$$

And as we showed in part (a), $M_n(\omega)^{-1} = (1/n)M_n(\omega^{-1})$. Thus to find the coefficients of $C(x)$ we simply have to multiply $(1/n)M_n(\omega^{-1})$ by the vector $\begin{bmatrix} C(1) & C(\omega) & \dots & C(\omega^{n-1}) \end{bmatrix}$.

(c) We note that if ω is a primitive nth root of unity, then ω^{-1} is also a primitive nth root of unity. This can be seen by recalling that $\omega^{-1} = e^{-i\theta}$ if $\omega = e^{i\theta}$. We now make a simple observation that the only property we used in providing that $M_n(\omega)$ decomposed into 4 smaller matrices was that ω was a primitive nth root of unity. Therefore, if we apply the mapping $\omega \mapsto \omega^{-1}$ which maintains this property, the proof still holds. Applying this map will yield the rearrangement in the problem statement.

4 (★★) The Greatest Roads in America

Arguably, one of the best things to do in America is take a great American road trip. And in America there are some amazing roads to drive on (think Pacific Crest Highway, Route 66, etc). An intrepid traveller has chosen to set course across America in search of some amazing driving. What is the length of the shortest path that hits at least k of these amazing roads?

Assume, that the roads in America can be expressed as a directed weighted graph $G = (V, E, d)$ and that our traveller wishes to drive across at least k roads from the subset $R \subset E$ of 'amazing' roads. Furthmore, assume that the traveller starts and ends at her home $h \in V$. Also, you can assume that the traveller is OK with repeating roads from R i.e. the k roads she chooses from R do not need to be unique.

Provide a 4 part solution with runtime in terms of $n = |V|, m = |E|, k$, and $r = |R|$.

Hint: First try out $k = 1$. How can G be modified so that we can use a 'common' algorithm to solve the problem?

Solution: The main idea is that we want to build a new graph G' such that we can apply Dijkstra's algorithm on this new graph. We start by creating $k + 1$ copies of the graph G, where each copy represents how many 'amazing' roads the traveller has crossed. We modify the special edges so that they now cross between the various copies. Then we apply Dijkstra's to find the distance between h in the 0th copy and h in the kth copy.

Pseudocode

Generate $k+1$ copies of the graph G. Call these copies $G_0, \dots, G_k$, and let $R_0, \dots, R_k$ be their respective amazing roads. For each edge $r_i = (u_i \to v_i) \in R_i$ for $i = 0, \dots, k-1$, modify the edge to be between u_i and v_{i+1}. Let the entire graph be G'. Run Dijkstra's algorithm on G' starting from h in G_0 and ending at h in G_k.

Runtime Since G' includes k copies of G, Dijkstra's algorithm will run in time $O((km + kn)\log(kn))$. Note $k \le m$ and $\log m = O(\log n)$, so the runtime is $O(k(m + n)\log n)$.

Correctness Assume there is a valid shorter path p in G than the one produced by this algorithm. Consider the equivalent path p' in G' formed by modifying the path to go to the next copy of G whenever crossing an edge of R. Since p is valid, p' must go from h in G_0 to

h in G_k. But then p' would be a shorter path in G' than the one produced by Dijkstra's, a contradiction.

5 (★★★) Activity Selection

Assume there are n activities each with its own start time a_i and end time b_i such that $a_i < b_i$. All these activities share a common resource (think computers trying to use the same printer). A feasible schedule of the activities is one such that no two activities are using the common resource simultaneously. Mathematically, the time intervals are disjoint: $(a_i, b_i) \cap (a_j, b_j) = \emptyset$. The goal is to find a feasible schedule that maximizes the number of activities k.

Here are two potential greedy algorithms for the problem.

Algorithm A: Select the earliest-ending activity that doesn't conflict with those already selected until no more can be selected.

Algorithm B: Select items by shortest duration that doesn't conflict with those already selected until no more can be selected.

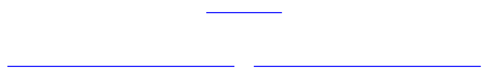
(a) Show that Algorithm B can fail to produce an optimal output.

(b) Show that Algorithm A will always produce an optimal output.

(c) (Extra Credit) Show that Algorithm B will always produce an output greater than 1/2 the optimal output.

Solution:

(a) There are many examples that work, but here is a simple one, easiest explained pictorally (each line represents an activity):



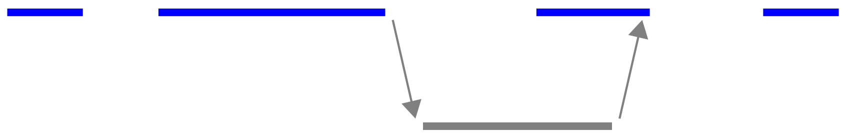
The optimal strategy is to pick the two longer activities, but strategy B picks the shorter activity, and then cannot pick another.

(b) Feasibility follows as we select the earliest activity that doesn't conflict. Let $E = ((a_{i_1}, b_{i_1}), \ldots, (a_{i_\ell}, b_{i_\ell}))$ by the output created by selecting the earliest-ending activity and $S = ((a_{j_1}, b_{j_1}), \ldots, (a_{j_k}, b_{j_k}))$ any other feasible schedule.

We claim that for all $m \leq k$, (a_{i_m}, b_{i_m}) exists and $b_{i_m} \leq b_{j_m}$ (or in layman's terms the mth activity in schedule E always ends before then mth activity in schedule S). Assume the claim is false and m is the smallest counterexample. Necessarily $m = 1$ cannot be a counterexample as the schedule E by construction takes the first-ending event and must end before an event in any other schedule.

If m is a counterexample, then $b_{i_{m-1}} \leq b_{j_{m-1}} \leq a_{j_m}$ and $b_{i_m} > b_{j_m}$. This means that the event j_m ends prior to the event i_m and is feasible with the other events of E. But then event j_m would have been chosen over event i_m, a contradiction (see picture). So the claim is true.

Therefore, the number of events in E is at least that of any other feasible schedule S, proving the optimality of E.



(c) Let I_{opt} be any optimal set of activities and I be the activities chosen by the shortest-duration greedy strategy.

We show the following:

(a) Every activity in I_{opt} overlaps at least 1 activity in I.

Suppose there exists activities in I_{opt} that does not overlap with any activity in I. Let $x \in I_{opt}$ be such an activity of shortest duration. If x had shorter duration than any other activities in I, it would have been added before the others were; contradiction. Otherwise, x would have been added just after all those in I are added; contradiction.

(b) Any activity in I can overlap at most 2 activities in I_{opt}.

Let I_k be the set of activities added by the greedy strategy just after the kth iteration. We show by induction that any activity in I_k overlaps with at most 2 activities in I_{opt}.

- Base case: $I_0 = \emptyset$; vacuously true.
- Inductive case: Assume true for I_k and let x be the activity added during the $k + 1$th iteration. If $x \in I_{opt}$, we are done. Otherwise, suppose for the purpose of contradiction that x overlapped with strictly more than two activities in I_{opt}. Then it must be that one such activity begins and ends while x is still active; contradiction.

The original claim follows immediately as a corollary.

The two claims above imply $|I_{opt}| \leq 2|I|$.

6 (Extra Credit Problem) Matrix Filling

(This is an *optional* challenge problem. It is not the most effective way to raise your grade in the course. Only solve it if you want an extra challenge.)

Consider the following problem: We are given an $n \times n$ matrix where some of the entries are blank. We would like to fill in the blanks so that all pairs of columns of the matrix are linearly dependent (two vectors v and u are linearly dependent if there exists some constant c such that $cv = u$) or report that there is no such way to fill in the blanks. Formulate this as a graph problem and design an $O(n^2)$ algorithm to solve it. You may assume that all the non-blank entries in the matrix are nonzero.

Solution:

Main Idea. Call the matrix given matrix A. Notice that if we fill in the entries of A correctly, every pair of columns in A will be related by a constant factor, and if two columns have filled in entries in the same row, then we immediately know what this constant has to be. Let's refer to this fact by saying that these entries in the common row "prove" how the two columns are related. If two columns both have filled in entries in two rows, and if the relationships proved by these two pairs of filled in entries disagree then we know right away that our task is hopeless. In fact, suppose there are three columns a, b, and c in the matrix, related in the following way: a and b both have a filled in entry in the first row, and these entries prove that $b = 5a$, b and c both have a filled in entry in the second row and these entries prove that $c = 3b$. Then if c and a share a filled in entry in the third row, these entries should prove that $c = 15a$. If they don't, then as in the previous scenario we can say that our task is hopeless. The main fact that we will show in this solution is that this type of problem is the only thing that can prevent us from successfully filling in the matrix.

Notice that this is starting to sound a lot like looking for a cycle in some graph. In particular, define G to be the undirected graph whose set of vertices V is the set of columns in A. There is an edge in G for every pair of filled in entries in the same row. We will allow multiple edges between the same two columns. Using this graph, we will try to determine the constant that relates the first column of A to every other column in A. If we ever discover an inconsistency then we will report that the task is impossible. If not, we will use this information to fill in the blank entries in A.

There are actually two problems with this approach. One is that G may have more than one connected component. This turns out not to be such a big problem. The other is that building G takes too long (imagine for instance that one row is already totally filled in– this already gives us $\Theta(n^2)$ edges in G and there are n rows). We will solve this problem by strategically leaving out some edges. In particular, each column will have at most two edges per filled in entry, one to the next column with a filled in entry in that row (if such a column exists) and one to the previous column with a filled in entry in that row (if such a column exists).

Algorithm.

```
FillInEntries($A[][]$):
1.  Initialize an empty array $C$ of length $n$ as a global variable
2.  For $i = 1$ up to $n$:
3.      If $C[i]$ is still empty:
4.          Set $C[i] = 1$
5.          Explore($i$)
6.  For $j = 1$ up to $n$:
7.      If no column has a filled in entry in row $j$:
8.          Fill in row $j$ with all zeros
9.      Else:
10.         Set $k$ to the index of the first column with a filled in entry in row $j$
11.         For $i$ from 1 up to $n$, fill in entry $A_{ji}$ with $\frac{C[i]}{C[k]} A_{jk}$
```

```
Explore($i$):
1.  For $j = 1$ up to $n$:
2.      If $A_{ji}$ is a filled in entry:
3.          Find the largest index $k$ smaller than $i$ such that $A_{jk}$ is filled in
4.          If $k$ exists:
5.              If $C[k]$ is still empty: Set $C[k] = C[i]\frac{A_{jk}}{A_{ji}}$ and call Explore($k$)
6.              Else: if $C[k] \neq C[i]\frac{A_{jk}}{A_{ji}}$ then halt the program and report "false"
7.          Repeat steps 3-6 but looking for the next column after $i$ with a filled in entry in row $j$
                instead of the last column before $i$ with a filled in entry in row $j$
```

Proof of Correctness. Notice that the algorithm is simply running DFS on the graph described at the end of the Main Idea section.

Claim 1. *If G is connected then for each column k, if there is a valid way to fill in the matrix then column i must be equal to column 1 times $C[k]$.*

Proof. We will show this by induction on the order in which DFS visits the columns. The base case is simply that column 1 must be $C[1] = 1$ times column 1, which is certainly true.

For the inductive case, assume that when we visit column k, the claim holds for all previously visited columns. Suppose column i is the parent of column k in the DFS tree. Then for some j, the entries in row j of both column i and column k are filled in. The algorithm sets $C[k] = C[i]\frac{A_{jk}}{A_{ji}}$. By the inductive hypothesis, in any valid way to fill in the matrix column i must be $C[i]$ times column 1. And as observed in the Main Idea section, column k must be $\frac{A_{jk}}{A_{ji}}$ times column i (since the constant that we multiply column i by to get column k must be give A_{jk} when multiplied by A_{ji}). Thus column k must indeed be $C[i]\frac{A_{jk}}{A_{ji}}$ times column 1. □

Claim 2. *If the algorithm returns false then there is no valid way to fill in the matrix.*

Proof. The algorithm returns false only if, while visiting some column i, there is some column k that is a neighbor of column i such that column k has already been visited and $C[k] \neq C[i]\frac{A_{jk}}{A_{ji}}$, where j is the index of some row in which both column i and k have a filled in entry. This implies that columns i and k are in the same connected component of G. Let column l be the first column in this connected component to be visited by the algorithm. It is easy to see that the proof of the previous claim also shows that for any valid way to fill in the matrix, column i must be equal to column l times $C[i]$ and column k must be equal to column l times $C[k]$. But by the same argument as in the inductive step of the proof of the claim above, column k must be equal to column i times $\frac{A_{jk}}{A_{ji}}$ and thus equal to column l times $C[i]\frac{A_{jk}}{A_{ji}}$. Since this is not equal to $C[k]$, there must be no valid way to fill in the matrix. □

Claim 3. *If the algorithm does not return false, then the way it fills in the matrix is valid.*

Proof. The way that we fill in entries in the matrix ensures that any entry we add in column i will be $C[i]$ times the entry in the same row in column 1 (since either this entry in column 1 is already filled in, in which case 1 is the index chosen in line 10 of the algorithm or column 1 is not filled in, in which case we fill in both entries to guarantee this fact). So the only way that the algorithm could fail is if there is some column i and some row j such that entry (j, i) is already filled in and it is not equal to $C[i]$ times the value in entry j of the first column (either an entry that the algorithm filled in or that was already filled in).

Suppose that this is the case. Let k be the index chosen in line 10 of the algorithm when we are attempting to fill in row j. If entry j in column 1 is already filled in, then $k = 1$. If the entry j in column 1 is not already filled in, then the algorithm fills it in with $\frac{C[1]}{C[k]}A_{jk} = \frac{A_{jk}}{C[k]}$. In either case, $C[i]$ times entry j in column 1 is $\frac{C[i]}{C[k]}A_{jk}$. Thus we have $A_{ji} \neq \frac{C[i]}{C[k]}A_{jk}$ or in other words, $C[k] \neq C[i]\frac{A_{jk}}{A_{ji}}$. Note that $k < i$ since in line 10 we choose the smallest possible index. Let $k = l_1 < l_2 < \ldots < l_m = i$ be the indices of the columns between columns k and i in which the entry in row j is already filled in. This corresponds to a path of $m - 1$ edges in the graph G. Since the algorithm does not return false, we must have

$$C[l_1] = C[l_2]\frac{A_{jl_1}}{A_{jl_2}}, C[l_2] = C[l_3]\frac{A_{jl_2}}{A_{jl_3}}, \ldots, C[l_{m-1}] = C[l_m]\frac{A_{jl_{m-1}}}{A_{jl_m}}$$

Thus we have:

$$\begin{aligned} C[k] = C[l_1] &= C[l_m]\frac{A_{jl_1}}{A_{jl_2}} \cdot \frac{A_{jl_2}}{A_{jl_3}} \cdot \ldots \cdot \frac{A_{jl_{m-1}}}{A_{jl_m}} \\ &= C[l_m]\frac{A_{jl_1}}{A_{jl_m}} \\ &= C[i]\frac{A_{jk}}{A_{ji}} \end{aligned}$$

which is a contradiction. Therefore the algorithm does not fail.

Really, the idea here is that the algorithm finds a number for each column and then fills in the entries trying to make each column equal to column 1 times the number it found for the column. If it finds that some already-filled-in-entry conflicts with this plan, then this means there must have been some edge in the graph which failed the check in line 6 of the explore subroutine. The only reason this proof may look a little complicated is because we needed to show that this proof still works when we leave out some of the edges (as described at the end of the Main Idea section). □

Running Time. Since we are simply running DFS, we visit each column once. When we visit a column, we look through each entry. If an entry is blank we do nothing. If an entry is filled in, we look forwards and backwards until we find the next filled in entry in that row in both directions. This involves looking at some number of blank entries. But note that each such blank entry only gets looked at two other times: when we visit that particular column and when we scan in the other direction from the column with the filled in entry in that row on the other side of the blank. By this reasoning, we examine each entry in the matrix at most 3 times: once when visiting its column and once when visiting the columns of the two filled in entries on either side of it in its row. Thus all calls to the explore subroutine take a total of $O(n^2)$ time. Filling in the entries in the matrix in the rest of the main routine then takes $O(n^2)$ time ($O(n)$ time to look for a filled in entry for each of n rows and then $O(n^2)$ time to actually fill in all the entries in the matrix). Thus the entire algorithm takes $O(n^2)$ time.

Note that it is not enough to say that we are running DFS on a graph with n vertices and at most n^2 edges because we need to take into account the amount of time it takes to calculate the edges in the graph For instance, finding a single edge out of a column can take up to $O(n)$ time, but with the analysis above, we can see that we still get a $O(n^2)$ running time despite this.