CS 170 — Efficient Algorithms and Intractable Problems
Spring 2016 — Alessandro Chiesa and Umesh Vazirani

HW Solution 3

Due February 8, 11:59pm

1. (15 pts.) Inverse FFT

The fast fourier transform can be used to efficiently multiply polynomials in the following way:

Input: Coefficients of two polynomials $A(x)$ and $B(x)$ each of degree at most d

Output: The coefficients of their product $C(x) = A(x)B(x)$

1. Find some power of two $n \geq 2d+1$
2. Use the FFT to evaluate A and B at the n^{th} roots of unity $1, \omega, \omega^2, \ldots, \omega^{n-1}$
3. Evaluate C at each of the n^{th} roots of unity: $C(1) = A(1)B(1), C(\omega) = A(\omega)B(\omega), \ldots, C(\omega^{n-1}) = A(\omega^{n-1})B(\omega^{n-1})$
4. Recover the coefficients of C from its evaluations on each of the n^{th} roots of unity

In class we saw how to do the second step efficiently using the FFT algorithm. In this question, we will see how to do the last step efficiently. For the remainder of this problem, ω will denote the first n^{th} root of unity. Recall that in class we defined M_n to be the following matrix:

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

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

(a) Define

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

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.

Solution:

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

Solution:

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) 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:

In this solution, ω will be used to refer to the first n^{th} root of unity and α will be used to refer to the first $(n/2)^{\text{th}}$ root of unity.

First we will show that the upper left quarter of the matrix is equal to $M_{n/2}(\omega^{-1})$. The entry at coordinate (j,k) of this quarter is simply ω^{-2jk} (because of the rearrangement of the columns). But as we have seen (see the discussion section for the second week for instance), since ω is the first n^{th} root of unity and n is even, ω^2 is the first $(n/2)^{\text{th}}$ root of unity. Thus this entry is simply α^{-jk}, exactly the entry at coordinate (j,k) of $M_{n/2}(\omega^{-1})$.

Now observe that the entry in coordinate (j,k) of the bottom left quarter of the matrix is simply

$$\omega^{-(n/2+j)(2k)} = \omega^{-(nk+2jk)} = \omega^{-nk}\omega^{-2jk}$$

And since $\omega^n = 1$, this is just ω^{-2jk}, exactly as in the previous case.

Now we turn to the upper right quarter of the matrix. We want to show that the entry at coordinate (j,k) of this quarter is simply ω^{-j} times the entry at coordinate (j,k) of the upper left quarter, namely ω^{-2jk}. The entry at coordinate (j,k) of the upper right quarter is

$$\omega^{-j(2k+1)} = \omega^{-j}\omega^{-2jk}$$

as desired.

Finally, we need to show that the entry at coordinate (j,k) of the lower right quarter of the matrix is simply $-\omega^{-j}$ times the entry at coordinate (j,k) of the upper left quarter, namely ω^{-2jk}. The entry in question is

$$\omega^{-(j+n/2)(2k+1)} = \omega^{-2jk-nk-j-n/2} = \omega^{-n/2}\omega^{-j}\omega^{-nk}\omega^{-2jk}$$

Now note that $(\omega^{-n/2})^2 = 1$ but $\omega^{-n/2} \neq 1$ so it must be -1 and, as before, $\omega^{-nk} = 1$. So the expression above simplifies to $-\omega^{-j}\omega^{-2jk}$.

Comment. In general, if x is a complex number and n and m are integers then $(x^n)^{1/m} \neq (x^{1/m})^n$ and so the expression $x^{n/m}$ is not really well defined. As a simple example of this, take $n = 2, m = 2$

and $x = -1$. Then $((-1)^2)^{1/2} = 1^{1/2} = 1$ but $((-1)^{1/2})^2 = i^2 = -1$. In light of this, in might seem worrisome that we were so cavalier with our $n/2$'s above. The reason that this is not a problem is that $n/2$ is an integer, and raising complex numbers to integer powers is much better behaved operation than raising them to noninteger powers.

As we saw in class, the result from part (c) means that we can efficiently multiply vectors by the matrix $M_n(\omega^{-1})$.

2. (15 pts.) Triple Sum

Design an efficient algorithm for the following problem. We are given an array $A[0..n-1]$ with n elements, where each element of A is an integer in the range $0 \le A[i] \le 10n$. We are also given an integer t. The algorithm must answer the following yes-or-no question: do there exist indices i, j, k such that $A[i] + A[j] + A[k] = t$?

Design an $O(n \log n)$ time algorithm for this problem.

Hint: define a polynomial of degree $O(n)$ based upon A, then use FFT for fast polynomial multiplication.

Solution:

Main idea: Exponentiation converts addition to multiplication. So, define

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}.$$

Notice that $p(x)^3$ contains a sum of terms, where each term has the form $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^{A[i]+A[j]+A[k]}$. Therefore, we just need to check whether $p(x)^3$ contains x^t as a term.

Pseudocode:

```
Algorithm TripleSum($A[0 \ldots n-1]$, $t$):
1. Set $p(x) := \sum_{i=0}^{n-1} x^{A[i]}$.
2. Set $q(x) := p(x) \cdot p(x) \cdot p(x)$, computed using the FFT.
3. Return whether the coefficient of $x^t$ in $q$ is nonzero.
```

Correctness: Observe that

$$q(x) = p(x)^3 = \left(\sum_{0 \le i < n} x^{A[i]} \right)^3 = \sum_{0 \le i,j,k < n} x^{A[i]} x^{A[j]} x^{A[k]} = \sum_{0 \le i,j,k < n} x^{A[i]+A[j]+A[k]}.$$

Therefore, the coefficient of x^t in q is nonzero if and only if there exist indices i, j, k such that $A[i] + A[j] + A[k] = t$. So the algorithm is correct. (In fact, it does more: the coefficient of x^t tells us *how many* such triples (i, j, k) there are.)

Constructing $p(x)$ clearly takes $O(n)$ time. Since $0 \le A[i] \le 10n$, $p(x)$ is a polynomial of degree at most $10n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. Finally, looking up the coefficient of x^t takes constant time, so overall the algorithm takes $O(n \log n)$ time.

Comment: This problem promised you that each element of the array is in the range $0 \ldots 10n$. What if we didn't have any such promise? Then the FFT-based method above becomes inefficient (because the degree of the polynomial is as large as the largest element of A). It is easy to find a $O(n^2)$ time algorithm, but no faster algorithm is known. In particular, it is a famous open problem (called the 3SUM problem) whether this problem can be solved more efficiently than $O(n^2)$ time. This problem has been studied extensively, because it is closely connected to a number of problems in computational geometry.

Comment: The technique used to solve this problem (namely encoding information in the coefficients of a polynomial and then manipulating the polynomial in some way) is closely related to the technique of generating functions, which are used in combinatorics to do many cool things such as giving a standard way to find a closed form solution to certain types of recurrence relations (for instance the recurrence relation defining the Fibonacci numbers).

3. (10 pts.) Depth

Let G be a connected undirected graph. You perform a depth-first search starting from some vertex $r \in V$. For each node v in the DFS tree, let $d(v)$ denote the depth of v in the DFS tree. In particular, the root r has depth 0; r's children have depth 1; r's grandchildren have depth 2; and so on.

Describe how to compute $d(v)$ for each vertex $v \in V$, in linear time. (You can assume the vertices are numbered $0..n-1$, so your goal is to build and initialize an array $\mathrm{d}[0..n-1]$ so that $\mathrm{d}[v]$ holds the depth of v.)

Solution:

We can do this by traversing the tree [not the original graph], either breadth- or depth-first. At each node v, set $d(v)$ to the depth of the node's parent, plus one. The key is to use a traversal order such that each node is visited (to calculate its depth) before any of its children are visited, so that we can calculate each child's depth as a function of its parent's depth.

4. (20 pts.) 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?

Solution: $f(r) =$ 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) 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?

Solution: $f(v) = 1+$ 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 eachother by removing v.

(c) Definition: If w is a node in the DFS tree, let $up(w)$ 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 $up(w) = \infty$ 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 $w_1,\dots,w_k$. Describe how to compute $f(v)$ as a function of k, $up(w_1),\dots,up(w_k)$, and $d(v)$.

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.

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

This case differs from that in part (b) because it might be that a child of v is connected to an ancestor of v by some route, not through v. This is equivalent to saying that v, some child, and some ancestor are connected by a cycle. We can use $up(\cdot)$ to detect this case.

Claim 1 *The set of nodes visited before node v, which includes all the tree ancestors of v, form a single connected component, which will not be split by removing v.*

Proof: The instant before we added v to the tree, the DFS tree connected all these nodes, and didn't include v. Removing v from the tree won't change that. □

Claim 2 *If c is a child of v in the DFS tree, and $up(c) < d(v)$, then removing v from the graph will not disconnect c from the tree component above v.*

Proof: By the definition of *up*, there is a chain of edges from c, possibly through descendants of c, to a node y at some depth less than v. By the definition of depth, all nodes at depth less than $d(v)$ are connected by a path through the root that does not include v Thus, removing v will not disconnect c or its children from that component. □

Claim 3 *If removing parent v will not disconnect child c from the component above v, then $up(c) < d(v)$.*

Proof: Since removing v does not disconnect c, there must be some path in the original graph, not containing v, that starts at c and ends at an already-visited node in the component "above" v. Call the end node in the path y. All nodes in the path except c and y will be descendants of c, since any node reachable from c that hasn't already been visited will be a descendant of c in the DFS tree.

So at the time that c was first visited, y had been visited, but had unexplored edges. Since DFS processes nodes in last-in-first-out (stack) order, this means y is an ancestor of c, and thus of v. Therefore, $d(y) < d(v)$. By definition of $up(\cdot)$, this means that $up(c) < d(v)$, since there is a chain of descendants of c leading to y with depth less than v. This proves the claim.

Another way to see it: We proved in Q3 that there are no cross edges in a DFS tree. All non-tree edges connect ancestors and descendants. If removing v doesn't disconnect a child, then there's an edge to a non-descendant, and therefore that edge must go to an ancestor. Ancestors have depth less than $d(v)$ and therefore $up(c) < d(v)$. □

Claims 2 and 3 establish that removing v disconnects each child from the component above v iff $up(c) < d(v)$. There can't be paths from one child to another, not passing through v or an ancestor of v; otherwise the DFS would have taken that path, and not returned to v. Let N be the number of children c of v with the property that $up(c) \geq d(v)$. Removing v forms $N + 1$ components: one for each child with the appropriate *up*, and one for the component above v.

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

Solution:

Main idea. By definition, $up(v)$ is the minimum of v's neighbors' depths, and the *up*s of v's descendants. Formally,

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

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

Pseudocode.

```
ComputeUp($v$):
1.  For each child $c$ of $v$ in the DFS tree:
2.      Call ComputeUp($c$).
3.  Set $up(v) := \min(\min\{d(w) : \{v,w\} \in E\},\ \min\{up(w) : w$ is a child of $v\})$.
```

Calling ComputeUp(r), where r is the root of the DFS tree, computes $up(v)$ for each vertex $v \in V$.

Proof of correctness. We will prove the correctness of our algorithm by using induction on the height of the vertex v (the difference between the depth of v and the depth of its deepest descendant).

Base case: The base case is when we call ComputeUp(v) with of a single vertex v that has no children. In this case, the procedure ComputeUp(v) sets $up(v)$ to $\min\{d(w) : \{v,w\} \in E\}$. This is the correct value since v has no descendants thus it should be the depth of the shallowest node w such that there is some graph edge $\{v,w\} \in E$ (see the definition of part (d)).

Inductive step: Assume that our procedure ComputeUp() sets the correct value to $up(w_i)$ if it is called with input the subtree rooted at w_i ($i = 1,2,\ldots,k$) where $w_1,\ldots,w_k$ are the children of v. (This is guaranteed by the inductive hypothesis, since $w_1,\ldots,w_k$ each have smaller height than v.) Thus, when we call ComputeUp(v), the procedure sets the correct values to $up(w_1),\ldots,up(w_k)$ by calling ComputeUp(w_i) for each child w_i of v (loop at lines 1-2). It finally sets

$$up(v) := \min(\min\{d(w) : \{v,w\} \in E\},\ \min\{up(w) : w \text{ is a child of } v\}),$$

which is the correct value of $up(v)$ since by definition, $up(v)$ is the minimum of v's neighbors' depths, and the *up*s of v's descendants. Therefore ComputeUp(v) sets the correct value to $up(v)$ which completes the proof.

Running time. $O(|V|+|E|)$. This follows because each vertex is processed once, and edges are only processed twice, once for each end of the edge.

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

Solution: This follows immediately from parts (c)–(e). Pick some node as the root. Then compute $up(\cdot)$ and $d(\cdot)$ 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 $d(\cdot)$, one to compute $up(\cdot)$, 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.

5. (20 pts.) Satisfying assignment

In the following problem, if x is a boolean variable, then $\bar{x}$ indicates its negation (i.e. if x is true then $\bar{x}$ is false and vice-versa). The negation of the negation of a variable is equivalent to the variable. A boolean variable and its negation are both referred to as literals.

Suppose we are given a list of boolean variables $x_1, \dots, x_n$ and a list of implications between literals: i.e. constraints of the form $x_i \Rightarrow x_j$ (which means that if x_i is assigned the value `true` then x_j must be assigned the value `true`), $\overline{x_i} \Rightarrow x_j$ (which means that if x_i is assigned the value `false` then x_j must be assigned the value `true`) and $x_i \Rightarrow \overline{x_j}$ (which means that if x_i is assigned the value `true` then x_j must be assigned the value `false`). We want to either return an assignment of `true` or `false` to each variable so that all the constraints are satisfied or report that no such assignment is possible. (The usual name for this problem is 2SAT. Later in the course we will see a related problem called 3SAT that is believed to be much harder.)

In this problem we will find an efficient way of solving this problem by reducing it to the problem of finding the strongly connected components of a directed graph. If there are n variables and m constraints then we will construct a directed graph $G = (V, E)$ as follows:

- G has $2n$ vertices: one for each variable and its negation (i.e. one for x_i and one for $\overline{x_i}$)
- G has $2m$ edges: for every implication between literals, $\alpha \Rightarrow \beta$, we introduce two edges, one from α to β and one from the negation of β to the negation of α. So for the constraint $\overline{x_i} \Rightarrow x_j$ we would add an edge from $\overline{x_i}$ to x_j and an edge from $\overline{x_j}$ to x_i. (It's a good idea to think about why both edges are necessary)

(a) Show that if G has a strongly connected component containing both x_i and $\bar{x}_i$ for some x_i then there is no satisfying assignment.

Solution:

Let α and β be literals. First suppose (α, β) is an edge in G. Then either $\alpha \Rightarrow \beta$ or $\overline{\beta} \Rightarrow \overline{\alpha}$ is a constraint. In either case since the contrapositive of an implication is equivalent to the implication, we know that for any assignment that satisfies the constraints if α is assigned the value `true` then β must also be assigned the value `true`.

Now suppose that there is a path in G from α to β. Using the above fact and indcution on the length of this path, for any assignment that satisfies the constraints if α is assigned the value `true` then β must also be assigned the value `true`.

Suppose that some strongly connected component contains both x_i and $\overline{x_i}$ for some i. Assume for contradiction that there is some assignment that satisfies all the constraints. Assume WLOG that x_i is assigned `true`. Therefore $\overline{x_i}$ is assigned the value `false`. Since x_i and $\overline{x_i}$ are in the same strongly connected component, there is a path in G from x_i to $\overline{x_i}$. But this means that $\overline{x_i}$ must be assigned the value `true`, a contradiction. (The case where x_i is assigned `false` is completely identical except that we use the fact that there is a path from $\overline{x_i}$ to x_i.)

(b) Now show the converse of part (a): namely that if none of G's strongly connected components contain both a literal and its negation then there must be a satisfying assignment. (Hint: Assign values to the variables as follows: repeatedly pick a sink strongly connected component in G. Assign `true` to all literals in the sink and `false` to all of their negations and then delete all of these. Show that this ends up discovering a satisfying assignment.)

Solution:

Assume that none of G's strongly connected components contain both a literal and its negation. We will show by induction on the number of variables that this ensures that there exists a satisfying assignment.

The base case is when there are no variables. In that case, it is vacuously true that there exists a satisfying assignment (namely the empty assignment).

Now assume for induction that there are $n > 0$ variables and that for any amount of variables less than n the claim holds true. Assume that no strongly connected component of G contains both a variable and its negation. There must exist some sink strongly connected component of G. Pick a single sink SCC. WLOG, suppose that $x_1, x_2, \ldots, x_k$ are the variables for which either they or their negation appears in this SCC. If we delete $x_1, x_2, \ldots, x_k$ and all constraints that involve these variables, then the graph associated with the resulting instance of the problem is just G with some vertices and edges removed. Therefore it does not have any strongly connected component in which both a variable and its negation appear, so by the inductive assumption there exists a satisfying assignment to this set of variables and constraints– i.e. some map $f : \{x_{k+1}, \ldots, x_n\} \to \{\texttt{true}, \texttt{false}\}$ that satisfies all constraints involving only the variables $x_{k+1}, \ldots, x_n$. Extend this to an assignment f' on the variables $x_1, x_2, \ldots, x_n$ by assigning true to a variable in $\{x_1, x_2, \ldots, x_k\}$ if it appears in the chosen sink SCC and assigning false to it otherwise. Note that since no SCC contains both a variable and its negation, if $\overline{x_i}$ appears in the chosen sink SCC then x_i does not and vice versa. So if $\overline{x_i}$ appears in the sink SCC then $f'(x_i) = \texttt{false}$ and if $i \le k$ and $\overline{x_i}$ does not appear then $f'(x_i) = \texttt{true}$.

We will now show that f' satsifies all the original constraints. Certainly it satisfies any constraints that don't involve the deleted variables. Now consider a constraint that does contain a deleted variable. Let the constraint be $\alpha \Rightarrow \beta$ where α and β are literals. There are several cases to consider:

Case 1: Suppose α and β are both vertices that were deleted. Note that if α is in the chosen sink SCC, then β must be as well (since otherwise it would not be a sink). And if β is not in the sink SCC then neither is α. So either β is assigned the value `true` (because it is in the sink SCC) or α is assigned the value `false` (because it is not in the sink SCC). In either case the constraint is satisfied.

Case 2: Suppose α was a deleted vertex, but β was not. Then α cannot be in the chosen sink SCC since β is not in the sink SCC. Thus α is assigned the value `false` and so the constraint is satisfied.

Case 3: Suppose α was not a deleted vertex, but β was. Then since the edge $\overline{\beta} \Rightarrow \overline{\alpha}$ is in G, $\overline{\beta}$ must not be in the sink SCC. But this means that β is in the sink SCC and so it assigned the value `true` and thus the constraint is satisfied.

And now that we know all constraints are satisfied, we are satisfied and thus conclude the proof.

Comment. Above, the phrase "α is assigned the value `true`" (and similar phrases) when α is a literal should be interpreted as follows: if α is x_i for some i then it should mean "x_i is assigned the value `true`." If instead α is $\overline{x_i}$ for some i then it should mean "x_i is assigned the value `false`."

(c) Conclude that there exists a linear time algorithm for finding a satisfying assignment.

Solution:

First we form the graph G defined above (this can be done in time linear in the number of variables and constraints). Then we compute the DAG of the strongly connected components of the graph (we saw in class that this can be done in linear time). Then we find a topological sort of this DAG (which we can do in linear time). Finally we process the SCCs of G in reverse topologically sorted order. For each SCC we process, assign `true` to all unassigned variables that appear in the SCC and assign `false` to all unassigned variables whose negations appear in the SCC (for each SCC, this takes time proportional to the number of vertices in the SCC). This is equivalent to the process inductively described in the proof of part (b) because in that process we never added edges (we only removed them) so an SCC that we got after deleting some vertices is part of an SCC in the original graph.

In pseudocode, this could be written as (assuming we have already formed G):

```
SatisfyingAssignment(G):
1. Decompose $G$ into SCCs
2. Find a topological sort $A_1, A_2, \ldots, A_k$ of these SCCs
3. For $i = k$ down to 1:
4.     For each literal $\alpha$ in $A_k$:
5.         If $\alpha = x_j$ for some $j$ and $x_j$ has no value assigned:
6.             Assign true to $x_j$
```

```
7.      If $\alpha = \overline{x_j}$ for some $j$ and $x_j$ has no value assigned:
8.          Assign false to $x_j$
```

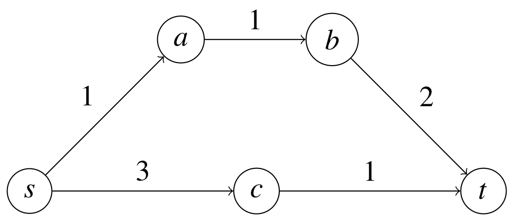
6. (20 pts.) Travel planning

You are given a set of n cities and an $n \times n$ matrix M, where $M(i,j)$ is the cost of the cheapest direct flight from city i to city j. All such costs are non-negative. You live in city A and want to find the cheapest and most convenient route to city B, but the direct flight might not be the best option. As far as you're concerned, the best route must have the following properties: the sum of the costs of the flights in the route should be as small as possible, but if there are several possible routings with the same total cost, then you want the one that minimizes the total number of flights. Design an efficient algorithm to solve this problem. Your algorithm should run in $O(n^2 \lg n)$ time.

(If you want, you can assume that the best route will require fewer than, say, 1000 flights, and that the cost of each flight is an integer number of dollars.)

A comment about all the solutions. In all the solution we build a complete graph whose vertices are the cities and the weight of the edge between city i and city j is given by $M(i,j)$. We will use the terms *edge* and *direct flight*, *path* and *route*, *weight* and *cost*, *vertex* and *city*, *shortest* and *cheapest* interchangeably.

Note that using Dijkstra's algorithm alone does not solve this problem. Consider the following graph. Dijkstra's algorithm will return the path $s \to a \to b \to t$, but the one with the smallest number of edges is just $s \to c \to t$.



Solution #1:

Main idea. We'll create a new matrix M' by adding a small penalty to the cost of each direct flight so that routes with more flights will cost more than routes with fewer flights. The penalty should be small enough that the cost of a cheapest route M' is still less than the cost of the next cheapest route in M.

Pseudocode.

1. Construct a new matrix M' by $M'(i,j) = M(i,j) + 10^{-3}$.
2. Run Dijkstra's algorithm from A on M'.
3. Return the shortest path from A to B.

Correctness. The cost of a route in M' is the same as the cost in M plus $k \cdot 10^{-3}$, where k is the number of flights on the route. Flight costs integers in M. This means that the cost of any non-cheapest route is higher by at least 1 than the shortest path.

Since the number of flights on the best route is less than 1000, the cost in M' of the best route in M is increased by less than 1. This means that the shortest path in M' is the shortest path in M with the least number of flights.

Running Time. Creating M' takes $O(n^2)$, running Dijkstra takes $O((|V|+|E|)\lg|V|) = O(n^2 \lg n)$ and returning the path takes $O(|V|+|E|) = O(n^2)$. This gives a total of $O(n^2 \lg n)$.

Solution #2:

Main idea. We can modify Dijkstra's algorithm, using a slightly different definition of the distance between two vertices: the distance from s to v is a *pair*, namely, the pair (c,d), where c is the lowest possible cost of all routes from s to v, and d counts the minimum number of direct flights needed to get from s to v via an itinerary of cost at most c. In other words, (c,d) is the cost of the "best" route from s to v, with "best" defined as in the problem statement. The operations in Dijkstra's algorithm that compare and add distances then need to be modified accordingly.

Pseudocode.

1. Create a new matrix M' by $M'(i,j) = (M(i,j),1)$. Or in other words, the new weight is the pair of the original weight and a number 1 indicating 1 direct flight.
2. Run Dijkstra's algorithm with the modified notion of distance as described next, from A on M'.
3. Return the shortest path from A to B.

Detailed explanation. The difficulty here is that, not only do we want to find the shortest path but, if there are ties, we want to use the fewest number of hops. Fortunately, we can *still* use Dijkstra's algorithm, we just need to redefine our notion of "distance". Here's one easy way to accomplish this. We will think of the distance of a path as a pair (c,d), where c is the cost of the whole trip, and d is the number of hops in the trip. Now, we will define a comparison function $<$ as follows:

$$(c_1,d_1) < (c_2,d_2) := \begin{cases} \text{if } c_1 < c_2, & \text{return } \texttt{true} \\ \text{if } c_1 = c_2 \text{ and } d_1 < d_2, & \text{return } \texttt{true} \\ \text{otherwise}, & \text{return } \texttt{false} \end{cases}$$

We also replace the constants 0 and ∞ which we use to initialize `dist` to the pairs $(0,0)$ and (∞,∞), respectively.

Finally, if we have a path to u with distance (c,d) and we add the edge (u,v), then the cost of this longer path is obviously $(c+\ell(u,v),d+1)$, where $\ell(u,v)$ is the length of the edge between u and v (i.e., the cost of a flight from city u to v). Now that we have a new "algebra" to add and compare distances, we may simply use Dijkstra's algorithm as before but with our redefined operations. (Note that we will need to make sure to use a priority queue where the DeleteMin operation finds the element that is smallest under our new comparison operator.) And, because our new notion of distance favors fewer hops among equal-cost paths, intuitively it makes sense that Dijkstra's algorithm should find the shortest path with the fewest number of hops.

Correctness. This algorithm can be proven correct by mimicking the proof of correctness of Dijkstra's algorithm, as proven in the textbook. This gets a bit tedious.

One key step is to prove the following lemma:

Lemma 1 *Suppose $s \rightsquigarrow u$ is a path with distance (c,d). If we add an edge $u \to v$ to the end of this path, then the new path $s \rightsquigarrow v$ will be longer than the original; i.e., if the distance of $s \rightsquigarrow v$ is (c',d'), then $(c,d) < (c',d')$.*

Proof: Suppose the edge $u \to v$ is of dollar cost c''. Then $(c',d') = (c+c'',d+1)$. Dollar costs are positive, so $c < c+c'' = c'$, so $(c,d) < (c',d')$ by the definition of $<$. □

I will leave it to you to verify that the rest of the standard proof goes through, with slight modifications, once this lemma is established. You would have needed to provide some argument in your solution about this.

Running time. We have a graph with n nodes and an edge for every city, so $|E| = n(n-1) = O(n^2)$. The running time is at most a constant factor larger than Dijkstra's algorithm, i.e., $O((|V|+|E|)\log|V|)$

(assuming you use a binary heap for the priority queue). So, for this particular graph the running time will be $O(n^2 \log n)$ as desired.

Solution #3:

Main idea. Run Dijkstra from the source to compute shortest paths and then remove edges so that the paths left are the shortest paths. At this point a simple BFS can find the shortest path with the least number of edges.

Pseudocode.

```
MinEdgeDijkstra($G = (V,E)$, $\ell$, $s$, $t$):
1. dist := Dijkstra($G$, $\ell$, $s$).
2. For each edge $(u,v) \in E$:
3.     Delete $e$ from $G$ if $\texttt{dist}(u) + \ell(u,v) > \texttt{dist}(v)$.
4. Run BFS on the modified graph.
5. Return the min-edge path from $s$ to $t$ by following the BFS prev pointers.
```

Proof of correctness. Line 1 computes the shortest distance $\texttt{dist}(\cdot)$ from s to every other vertex. In lines 2-3 we break all non-shortest paths so that the paths from s to t are exactly the shortest paths from s to t. Then we run BFS to find the shortest path with the smallest number of edges and return it.

Running time. This algorithm runs in time $O(n^2 \lg n)$.

Justification of running time. Dijkstra's algorithm takes $O((|V| + |E| \log |V|) = O(n^2 \lg n)$ time . Iterating over the edges take $O(|E|) = O(n^2)$, running BFS is also $O(|V| + |E|) = (n^2)$ and returning the shortest BFS path also takes $O(n^2)$ time. The total is $O(n^2 \lg n)$.

Solution #4:

Main idea. Modify Dijkstra's algorithm to keep track of the smallest number of edges found in the shortest path.

Pseudocode. We record for each vertex the number of edges in the shortest path with the least number of edges. In lines 1-2 we initialize it similarly to how `dist`is initialized.

When each edge is processed, if it improves `dist` (lines 3-4), then the number of edges in the shortest path handled so far is set to `nedges`. If the current edge is part of a path that has the same distance as a previously handle path, then we set `nedges`to be the minimum of the two.

```
    for all $v \in V$:
        dist$(v) = \infty$
1.      nedges$(v) = \infty$
    dist$(s) = 0$
2.  nedges$(s) = 0$

    $H =$ make-queue$(V)$
    while $H$ is not empty:
        $u =$ delete-min$(H)$

        for all edges $(u,v) \in E$:

3.          if dist$(v) > \ell(u,v) +$ dist$(u)$:
4.              nedges$(v) =$ nedges$(u) + 1$
5.          else if dist$(v) > \ell(u,v) +$ dist$(u)$:
6.              nedges$(v) = \min($nedges$(v)$, nedges$(u) + 1)$

            if dist$(v) > \ell(u,v) +$ dist$(u)$:
                dist$(v) =$ dist$(u) + \ell(u,v)$
                decrease-key$(H,v)$
```

Figure 1: Modified Dijkstra's Algorithm for solution #4. The numbered lines are the lines added to the regular Dijkstra's algorithm.

Finally, we trace back the min-edge shortest path by choosing an incoming edge with both `dist`$(u) =$ `dist`$(v) + l(u,v)$ and `nedges`$(u) =$ `nedges`$(v) + 1$.

Proof of correctness. Because this algorithm does not alter the computation of `dist`or which vertices are pushed into the heap, the modification does not change `dist`.

We will prove the following invariant. This is similar to the proof of correctness for Dijkstra's algorithm.

Invariant 1 *Let $R = V \setminus H$, that is the set of vertices already deleted from the queue. At the end of each iteration of the while loop, the following condition holds: for every node $u \in R$, the value* `nedges`(u) *is the smallest number of edges in the shortest path from s to u.*

Proof: By induction over the number of iterations. The base case is trivial $S = \{s\}$ at the end of the first iteration. Assume it is true for the first n iterations and consider the $n+1$ iteration. When u is removed from the queue, all of its incoming edges from vertices whose distances from s are smaller have already been handled and therefore `nedges`(v) is the 1 more than the minimum `nedges`(u) for any incoming edge (u,v) that is part of a shortest path to v. □

Running time. This algorithm runs in time $O((|V|+|E|\log|V|) = O(n^2 \lg n)$. The modification just adds constant time for each edge and for each vertex.

Comments: It is much easier to prove answer 1 and 3, because we don't need to re-prove Dijkstra's algorithm correct; we can just use it as a subroutine that's already known to be correct.

Answer 3 is also more general than answer 1, because it does not depend on the extra assumptions that edge weights are integers and that the best path has less than a thousand edges.

Answer 2 is harder to prove correct, because we need to re-prove the correctness of Dijkstra's algorithm. However, Answer 2 has the advantage of being more generic than 1 and also shows Dijkstra's algorithm actually works correctly with any notion of distance that is additive, totally ordered, and satisfies Lemma 1.

Answer 1 can be made a bit more general, by replacing the penalty 10^{-3} with $1/|V|$. Any shortest path from s to v must visit at most $|V|$ vertices (no vertex can be visited twice, because this would create a cycle, and then we could just bypass the cycle and get a shorter path), so uses at most $|V|-1$ edges. This eliminates the need to assume that the shortest path involves at most 1000 flights.

All answers can be sped up to run in $O(n^2)$ time, by using a slightly different data structure for the priority queue. In fact, it suffices to use an unsorted list of elements, with a pointer from each vertex v to the element associated with v. Then Insert and DecreaseKey can be done in $O(1)$ time, and DeleteMin can be implemented in $O(|V|) = O(n)$ time by scanning the entire list to find the minimum. Since Dijkstra's algorithm uses $O(|V|)$ Insert and DeleteMin calls and $O(|E|)$ DecreaseKey calls, the total running time is $O(n^2)$ with this data structure. For very dense graphs, this data structure leads to a small speed improvement; but for sparse graphs, the standard priority queue data structure based upon a binary heap is best.

7. (5 pts.) Optional bonus problem: Filling in matrix entries
(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.