CS 170 HW 3

Due on 2018-09-16, at 9:59 pm

1 (★) Study Group

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), has at least two vertices, and is represented in adjacency list format.

For each part, prove that your answer is correct (some parts are simple enough that the proof can be a brief justification; others will be more involved).

(a) Let T be a tree produced by running DFS on G with root $r \in V$. (In particular, $T = (V, E_T)$ is a spanning tree of G.) Given T, find an efficient way to calculate $f(r)$.

(b) Let $v \in V$ be some vertex that is not the root of T (i.e., $v \neq r$). Suppose further that no descendant of v in T has any non-tree edge (i.e. edge in $E \setminus E_T$) to any ancestor of v in T. How could you calculate $f(v)$ from T in an efficient way?

(c) For $w \in V$, let $D_T(w)$ be the set of descendants of w in T including w itself. For a set $S \subseteq V$, let $N_G(S)$ be the set of *neighbours* of S in G, i.e. $N_G(S) = \{y \in V : \exists x \in S \text{ s.t. } \{x, y\} \in E\}$. We define $\mathsf{up}_T(w) := \min_{y \in N_G(D_T(w))} \mathsf{depth}_T(y)$, i.e. the smallest depth in T of any neighbour in G of any descendant of w in T.

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

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 which, on input G, T, computes $\mathsf{up}_T(v)$ for all vertices $v \in V$, in linear time.

(e) Given G, describe how to compute $f(v)$ for all vertices $v \in V$, in linear time.

Solution: Lemma: Let T be a DFS tree, and let $u, v \in V$ be such that u is neither a descendant nor an ancestor of v in T. Then there is no edge $\{u, v\} \in E$.

Proof: Suppose that there is an edge in $\{u, v\} \in E$, and suppose that u is visited first in the DFS. Then at some point we leave u without traversing $\{u, v\}$ (else v would be a child of u). But this means that v was visited between entering and leaving u, and so it is a descendant of u. If v was visited first, the same argument shows that v would be an ancestor of u.

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

Proof: Let k be the number of children of r, and let $T_1, \dots, T_k$ be the subtrees of T rooted at those children. If $u \in T_i, v \in T_j$ for $i \neq j$ then the conditions of the lemma hold and so $\{u, v\} \notin E$, and so each T_i is a connected component when we remove r.

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

Proof: Consider a partition of V into three sets: A, the ancestors of v, B, the tree rooted at v, and C, the rest of the graph. It suffices to show that there are no edges from $B - v$ to $A \cup C$, since $A \cup C$ is connected; applying the previous subpart to B then gives the result. For every $u \in B - v$, all descendants and ancestors of u lie in $A \cup B$, and so by the lemma there is no edge from u to C. By assumption there are no edges from u to A, and so $B - v$ has no edges to $A \cup C$.

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

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

If c is connected to a proper ancestor a of v by a path that excludes v then $\mathsf{up}_T(c) \leq \mathsf{depth}(a) < \mathsf{depth}(v)$. Conversely, if $\mathsf{up}_T(c) < \mathsf{depth}(v)$, then there is an edge from a descendant d of v to some vertex w which has smaller depth than v. Since w cannot be a descendant of d, it must be an ancestor of d by the lemma, and since it has smaller depth than v, it is also an ancestor of v.

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

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

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

For a leaf, $\mathsf{up}_T(v)$ 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 $\mathsf{up}_T(\cdot)$ 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 $\mathsf{up}_T(\cdot)$, 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:

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

where ω is a primitive n-th root of unity.

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

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

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

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

(b) Let A be a square matrix with complex entries. The *conjugate transpose* $A^\dagger$ of A is given by taking the complex conjugate of each entry of A^T. A matrix A is called *unitary* if its inverse is equal to its conjugate transpose, i.e. $A^{-1} = A^\dagger$. Show that $\frac{1}{\sqrt{n}}M_n(\omega)$ is unitary.

(c) 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 $C(x)$.

(d) 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:

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

The notation $\omega^{-j}M_{n/2}(\omega^{-2})$ is used to mean the matrix obtained from $M_{n/2}(\omega^{-2})$ 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) Observe that $(M_n(\omega))^\dagger = M_n(\omega^{-1})$. So $\frac{1}{\sqrt{n}}M_n(\omega)\frac{1}{\sqrt{n}}(M_n(\omega))^\dagger = \frac{1}{n}M_n(\omega)M_n(\omega^{-1}) = I$.

(c) 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 $\begin{bmatrix} C(1) & C(\omega) & \ldots & C(\omega^{n-1}) \end{bmatrix}$.

(d) 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 (★★) Vandermonde matrix

Recall that the Vandermonde matrix $V_n(x_1, \ldots, x_n)$ is given by

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{bmatrix}$$

(a) Let $p(x) = x^3 + 2x + 1$. Use a Vandermonde matrix to find $p(0), p(1), p(4), p(7)$.

(b) Use a Vandermonde matrix to find the polynomial q of minimal degree such that $q(0) = 2, q(1) = 0, q(4) = 6, q(7) = 30$.

(c) Find $x_1, \ldots, x_n \in \mathbb{C}$ such that the FFT matrix $M_n(\omega) = V_n(x_1, \ldots, x_n)$, where ω is a primitive n-th root of unity.

Solution:

(a)

$$V_4(0,1,4,7) \cdot \begin{pmatrix}1\\2\\0\\1\end{pmatrix} = \begin{pmatrix}1 & 0 & 0 & 0\\1 & 1 & 1 & 1\\1 & 4 & 16 & 64\\1 & 7 & 49 & 343\end{pmatrix} \cdot \begin{pmatrix}1\\2\\0\\1\end{pmatrix} = \begin{pmatrix}1\\4\\73\\358\end{pmatrix}$$

$p(0) = 1, p(1) = 4, p(4) = 73, p(7) = 358.$

(b) We can reuse $V_4(0,1,4,7)$ from the previous part. We're looking for v such that $V_4(0,1,4,7)\cdot v = (2,0,6,30)$. The solution is $v = (2,-3,1,0)$. The corresponding polynomial is $q(x) = x^2 - 3x + 2$.

(c) $M_n(\omega) = V_n(1, \omega, \ldots, \omega^{n-1})$.

5 (★★) Connectivity vs Strong Connectivity

(a) Prove that in any connected undirected graph $G = (V,E)$ there is a vertex $v \in V$ such that removing v from G gives another connected graph.

(b) Give an example of a strongly connected directed graph $G = (V,E)$ such that, for *every* $v \in V$, removing v from G gives a directed graph that is not strongly connected.

(c) Let $G = (V,E)$ be a connected undirected graph such that G remains connected after removing any vertex. Show that for every pair of vertices u, v where $(u,v) \notin E$ there exist two different u-v paths.

Solution:

(a) Let T be a DFS tree on G, and let v be a leaf of T. Then $T - v$ is a connected graph because any simple path P from u to w ($u, w \neq v$) in T cannot pass through v (since v has degree 1). Since $T - v$ is a subgraph of $G - v$, $G - v$ is also connected.

(b) A directed cycle of three nodes is an example here (i.e. $V = \{a,b,c\}$ and $E = \{(a,b),(b,c),(c,a)\}$).

(c) Let P be a u-v path in G; then P contains a vertex w which is not u or v. The graph $G - w$ does not contain P, but there exists a path P' connecting u and v since $G - w$ is connected. P' exists in G and is different from P.

6 (★★★) Finding Clusters

We are given a directed graph $G = (V, E)$, where $V = \{1, \dots, n\}$, i.e. the vertices are integers in the range 1 to n. For every vertex i we would like to compute the value $m(i)$ defined as follows: $m(i)$ is the smallest j such that vertex j is reachable from vertex i. (As a convention, we assume that i is reachable from i.) Show that the values $m(1), \dots, m(n)$ can be computed in $O(|V| + |E|)$ time.

Solution: Let G^R be the graph G with its edge directions reversed. The algorithm is as follows.

```
procedure DFS-Clusters($G$)
    while there are unvisited nodes in $G$ do
        Run DFS on $G^R$ starting from the numerically-first unvisited node $i$
        for $j$ visited by this DFS do $m(j) := i$
```

To see that this algorithm is correct, note that if a vertex i is assigned a value then that value is the smallest of the nodes that can reach it in G^R, and every node is assigned a value because the loop does not terminate until this happens. Now observe that the set of vertices reachable by i in G^R is the set of vertices which can reach i in G.

The running time is $O(|V|+|E|)$ since computing G^R can be done in linear time (or faster if we use an adjacency matrix!), and we process every vertex and edge exactly once in the DFS.

7 (★★★) Arbitrage

Shortest-path algorithms can also be applied to currency trading. Suppose we have n currencies $C = \{c_1, c_2, \dots, c_n\}$: e.g., dollars, Euros, bitcoins, dogecoins, etc. For any pair i, j of currencies, there is an exchange rate $r_{i,j}$: you can buy $r_{i,j}$ units of currency c_j at the price of one unit of currency c_i. Assume that $r_{i,i} = 1$ and $r_{i,j} \geq 0$ for all i, j.

The Foreign Exchange Market Organization (FEMO) has hired Oski, a CS170 alumnus, to make sure that it is not possible to generate a profit through a cycle of exchanges; that is, for any currency $i \in C$, it is not possible to start with one unit of currency i, perform a series of exchanges, and end with more than one unit of currency i. (That is called *arbitrage*.)

More precisely, arbitrage is possible when there is a sequence of currencies $c_{i_1}, \dots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot \dots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. This means that by starting with one unit of currency c_{i_1} and then successively converting it to currencies $c_{i_2}, c_{i_3}, \dots, c_{i_k}$ and finally back to c_{i_1}, you would end up with more than one unit of currency c_{i_1}. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for profit.

We say that a set of exchange rates is arbitrage-free when there is no such sequence, i.e. it is not possible to profit by a series of exchanges.

(a) Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$ which is *arbitrage-free*, and two specific currencies s, t, find the most advantageous sequence of currency exchanges for converting currency s into currency t.

Hint: represent the currencies and rates by a graph whose edge weights are real numbers.

(b) Oski is fed up of manually checking exchange rates, and has asked you for help to write a computer program to do his job for him. Give an efficient algorithm for detecting the possibility of arbitrage. You may use the same graph representation as for part (a).

Solution:

(a) **Main Idea:**
We represent the currencies as the vertex set V of a complete directed graph G and the exchange rates as the edges E in the graph. Finding the best exchange rate from s to t corresponds to finding the path with the largest product of exchange rates. To turn this into a shortest path problem, we weigh the edges with the negative log of each exchange rate. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.

Pseudocode:

```
1: function BestConversion($(s, t)$)
2:     $G \leftarrow$ Complete directed graph, $c_i$ as vertices, edge lengths $l = \{-\log(r_{i,j}) \mid (i, j) \in E\}$.
3:     dist, prev $\leftarrow$ BellmanFord($G, l, s$)
4:     return Best rate: $e^{-\texttt{dist}[t]}$, Conversion Path: Follow pointers from $t$ to $s$ in prev
```

Proof of Correctness:
To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph, taking s as origin. The correctness of the entire algorithm follows from the proof of correctness of Bellman-Ford.

Runtime:
Same as runtime of Bellman-Ford, $O(|V|^3)$ since the graph is complete.

(b) **Main Idea:**
Just iterate the updating procedure once more after $|V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

Pseudocode: This algorithm takes in the same graph constructed in the previous part.

```
1: function HasArbitrage($G$)
2:     dist, prev $\leftarrow$ BellmanFord($G, l, s$)
3:     dist* $\leftarrow$ Update all edges one more time
4:     return True if for some $v$, dist[$v$] > dist*[$v$]
```

Proof of Correctness:
Same as the proof for the modification of Bellman-Ford to find negative edges.

Runtime:
Same as Bellman-Ford, $O(|V|^3)$.

Note:
Both questions can be also solved with a variation of Bellman-Ford's algorithm that works for multiplication and maximizing instead of addition and minimizing.

8 (★★★★) Bounded Bellman-Ford

Modify the Bellman-Ford algorithm to find the weight of the lowest-weight path from s to t with the restriction that the path must have at most k edges.

Solution: The obvious instinct is to run the outer loop of Bellman-Ford for k steps instead of $|V|-1$ steps. However, what this does is to guarantee that all shortest paths using at most k edges would be found, but some shortest paths using more that k edges might also be found. For example, consider a path on 10 nodes starting at s and ending at t, and set $k = 2$. If Bellman-Ford processes the vertices in the order of their increasing distance from s (we cannot guarantee beforehand that this will **not** happen) then just one iteration of the outer loop finds the shortest path from s to t, which contains 10 edges, as opposed to our limit of 2. We therefore need to limit Bellman-Ford so that results computed during a given iteration of the outer loop are not used to improve the distance estimates of other vertices during the **same** iteration.

We therefore modify the Bellman-Ford algorithm to keep track of the distances calculated in the previous iteration.

Algorithm 1 Modified Bellman-Ford

```
Require: Directed Graph $G = (V, E)$; edge lengths $l_e$ on the edges, vertex $s \in V$, and an
  integer $k > 0$.
Ensure: For all vertices $u \in V$, $dist[u]$, which is the length of path of lowest weight from $s$
  to $u$ containing at most $k$ edges.
  for $v \in V$ do
      dist$[u] \leftarrow \infty$
      new-dist$[u] \leftarrow \infty$
  dist$[s] \leftarrow 0$
  new-dist$[s] \leftarrow 0$
  for $i = 1, \ldots, k$ do
      for $v \in V$ do
          previous-dist$[v] \leftarrow$ new-dist$[v]$
      for $e = (u, v) \in E$ do
          new-dist$[v] \leftarrow \min($new-dist$[v]$, previous-dist$[u] + l_e$
```

Assume that at the beginning of the ith iteration of the outer loop, `new-dist`$[v]$ contains the lowest possible weight of a path from s to v using at most $i-1$ edges, for all vertices v. Notice that this is true for $i = 1$, due to our initialization step. We will now show that the statement also remains true at the beginning of the $(i+1)$th iteration of the loop. This will prove the correctness of the algorithm by induction. We first consider the case where there is no path from s to v of length at most i. In this case, for all vertices u such that $(u, v) \in E$, we must have `new-dist`$[u] = \infty$ at the beginning of the loop. Thus, `new-dist`$[v] = \infty$ at the end of the loop as well. Now, suppose that there exists a path (not necessarily simple) of length at most i from s to v, and consider such a path of smallest possible weight w. We want to show that `new-dist`$[v] = w$.

Let u be the vertex just before v on this path. By the induction hypothesis, at the end of the loop on line 7, `previous-dist`$[u]$ stores the weight of the lowest weight path of length at most $i-1$ from s to u, so that when the edge (u,v) is proceed in the loop on line 9, we get `new-dist`$[v] \le w$.

Now, we observe that at the end of the loop on line 9, we have

$$\texttt{new-dist}[v] = \min\left(\texttt{previous-dist}[v], \min_{u:(u,v)\in E}\left(\texttt{previous-dist}[u] + l_{(u,v)}\right)\right).$$

Note that by the induction hypothesis, each term in the minimum expression represents the length of a (not necessarily simple) path from s to v of length at most i. Thus, in particular, none of these terms can be smaller than w, so that `new-dist`$[v] \ge w$. Combining with `new-dist`$[v] \le w$ obtained above, we get `new-dist`$[v] = w$ as required.

9 (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 A 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:

We'll call A *fillable* if there is a way to fill in the blanks to make the columns of A linearly dependent. A is fillable if and only if there exists a matrix $\bar{A}$ and n-dimensional row vectors x, y such that $\bar{A} = x^T y$ and $\bar{A}$ agrees with A where A is filled. Thus we can focus on finding x, y.

Let $G = (V, E)$ be the following weighted directed multigraph. $V = \{1, \dots, n\} = [n]$ correspond to the columns of A. For each $i, j, k \in [n]$ there is an edge (i, j) in E with weight $\log(A_{ik}/A_{jk})$ if A_{ik}, A_{jk} are both filled in, and A_{lk} is not filled for all l *between* i and j. G has at most n^2 edges. Note that if there is an edge $(i,j) \in E$ with weight w then there is an edge $(j,i) \in E$ with weight $-w$; we'll say G is *symmetric*. We first prove some useful properties of G.

Lemma: A is fillable if and only if for all $i, j \in V$, every $i-j$ path in G has the same weight w_{ij}.

Proof: Suppose that A is fillable, and let A' be some filled-in A. For $r \in [n]$, let A_r denote the r-th column of A. For any $r \in [n]$, we have $A'_i = c_r A'_r$ for some c_r, so for all $r, s \in [n]$, $c_r A'_r = c_s A'_s$. Thus for every k such that A_{rk}, A_{sk} are both filled in, $A_{rk}/A_{sk} = c_s/c_r$. Let P be an $i-j$ path in G. Then the weight of P is $\log(\prod_{(j,l)\in P} c_s/c_r) = \log(c_j/c_i) =: w_{ij}$.

First suppose that G is connected. For each $j \in V$, w_j be the weight of a $1-j$ path. Let $x_j := 2^{-w_j}$. Now for every filled entry A_{jk}, let $y_k := 2^{w_j} A_{jk}$. Note that $(x^T y)_{jk} = A_{jk}$. In order for this to be well-defined, we need that $2^{w_j} A_{jk} = 2^{w_l} A_{lk}$ for all $j, l \in V$ where A_{jk}, A_{lk} are not blank. By construction there exists a path from j to l with weight $\log(A_{jk}/A_{lk})$, and so $w_l = w_j + \log(A_{jk}/A_{lk})$, which gives $2^{w_l} A_{lk} = 2^{w_j} A_{jk}$.

If G is not connected, we can do the above procedure for each connected component of G independently (replacing 1 with the first node in the CC), since if $j, l \in V$ with A_{jk}, A_{lk} not blank, j, l are in the same connected component.

So now we have reduced the problem to: given a directed symmetric multigraph G, is it the case that for all $i, j \in V$, every $i - j$ path has the same weight? This seems a little challenging, so we'll make a few simplifications. First, let's look at all the edges from i to j. If their weights are not all the same, we can immediately reject. Otherwise we can 'collapse' all of these edges into one, leaving a directed symmetric graph (not multigraph). Now we'll show that this property is equivalent to a property of cycles.

Lemma: Let G be a directed symmetric graph. Every (possibly non-simple) cycle in G has weight zero if and only if for all $i, j \in V$, every $i - j$ path has the same weight w_{ij}.

Proof: Suppose that every $i - j$ path has the same weight. Let C be a cycle in G, and let i be a vertex on that cycle. There is an $i - i$ path of weight zero (i.e. the empty path), and so since C is also an $i - i$ path, C has weight zero. Now suppose that every cycle in G has weight zero. Let $i, j \in V$, and suppose towards contradiction that there is a path P from i to j of length k and a path P' from i to j of length $k' \neq k$. Then since G is symmetric, there is a path $\hat{P}'$ from j to i of length $-k'$. But then the union of P and $\hat{P}'$ is a cycle of nonzero weight $k - k'$.

The algorithm is as follows.

```
procedure LAZY-BELLMAN-FORD(graph $H$, vertex $i \in V(H)$)
    Run DFS on $H$ rooted at $i$, get tree $T$
    compute distances in $T$ from $i$ to all $j \in H$
    run one Bellman-Ford pass on these distances
    if any distance is updated then
        abort, return 'not fillable'
    else
        return shortest path tree $T$
procedure MATRIX-FILLING(matrix $A$)
    Build $G$ as described.
    for $i, j \in V$ do
        $w_{ij} \leftarrow$ weight of some $i - j$ edge
        if there is an $i - j$ edge whose weight is not $w_{ij}$ then
            return 'not fillable'
        collapse all $i - j$ edges into a single edge with weight $w_{ij}$
    for $C$ connected component of $G$ do
        let $i$ be the first vertex of $C$
        $T \leftarrow$LAZY-BELLMAN-FORD($C$,$i$)
        let $C'$ be $C$ with every edge weight negated
        LAZY-BELLMAN-FORD($C'$,$i$)
        for $j \in C$ do
            $w_j \leftarrow$ length of $i - j$ path in $T$
            $x_j \leftarrow 2^{-w_j}$
            for $k \in n$ do
                if $A_{jk}$ filled then $y_k \leftarrow 2^{w_j} A_{jk}$
```

Why does this work? Suppose that the algorithm outputs 'no'. This happens either in the simple case where there are edges from i to j with different weights, or if the DFS tree is updated by a Bellman-Ford pass. The latter happens if and only if there is a negative weight cycle in G or G', i.e. if there is a non-zero weight cycle in G, since otherwise the DFS tree is already a shortest path tree. The matrix is then filled according to the lemma.

The algorithm runs in linear time because both DFS and a single Bellman-Ford pass run in linear time.