22 Elementary Graph Algorithms

22.1 Representations of graphs

22.1-1

> Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

- The time to compute the out-degree of every vertex is

$$\sum_{v \in V} O(\text{out-degree}(v)) = O(|E| + |V|),$$

  which is straightforward.

- As for the in-degree, we have to scan through all adjacency lists and keep counters for how many times each vertex has been pointed to. Thus, the time complexity is also $O(|E| + |V|)$ because we'll visit all nodes and edges.

22.1-2

> Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.

- **Adjacency-list representation**

$$\begin{aligned}
1 &\to 2 \to 3 \\
2 &\to 1 \to 4 \to 5 \\
3 &\to 1 \to 6 \to 7 \\
4 &\to 2 \\
5 &\to 2 \\
6 &\to 3 \\
7 &\to 3
\end{aligned}$$

- **Adjacency-matrix representation**

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	0	1	1	0	0
3	1	0	0	0	0	1	1
4	0	1	0	0	0	0	0
5	0	1	0	0	0	0	0
6	0	0	1	0	0	0	0
7	0	0	1	0	0	0	0

22.1-3

> The ***transpose*** of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all its edges reversed. Describe efficient algorithms for computing G^T from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

- **Adjacency-list representation**

  Assume the original adjacency list is Adj.

```
let Adj'[1..|V|] be a new adjacency list of the transposed G^T
for each vertex u ∈ G.V
    for each vertex v ∈ Adj[u]
        INSERT(Adj'[v], u)
```

  Time complexity: $O(|E| + |V|)$.

- **Adjacency-matrix representation**

  Transpose the original matrix by looking along every entry above the diagonal, and swapping it with the entry that occurs below the diagonal.

  Time complexity: $O(|V|^2)$.

22.1-4

> Given an adjacency-list representation of a multigraph $G = (V, E)$, describe an $O(V + E)$-time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G' = (V, E')$, where E' consists of the edges in E with all multiple edges between two vertices replaced by a single edge and with all self-loops removed.

```
EQUIVALENT-UNDIRECTED-GRAPH
    let Adj'[1..|V|] be a new adjacency list
    let A be a 0-initialized array of size |V|
    for each vertex u ∈ G.V
        for each v ∈ Adj[u]
            if v != u && A[v] != u
                A[v] = u
                INSERT(Adj'[u], v)
```

Note that A does not contain any element with value u before each iteration of the inner for-loop. That's why we use $A[v] = u$ to mark the existence of an edge (u, v) in the inner for-loop. Since we lookup in the adjacency-list Adj for $|V| + |E|$ times, the time complexity is $O(|V| + |E|)$.

22.1-5

> The ***square*** of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe efficient algorithms for computing G^2 from G for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

- **Adjacency-list representation**

  To compute G^2 from the adjacency-list representation Adj of G, we perform the following for each $Adj[u]$:

```
for each v ∈ Adj[u]
    INSERT(Adj2[u], v)
    for each w ∈ Adj[v]
        // edge(u, w) ∈ E^2
        INSERT(Adj2[u], w)
```

  where $Adj2$ is the adjacency-list representation of G^2. For every edge in Adj we scan at most $|V|$ vertices, we compute $Adj2$ in time $O(|V||E|)$.

  After we have computed $Adj2$, we have to remove duplicate edges from the lists. Removing duplicate edges is done in $O(V + E')$ where $E' = O(VE)$ is the number of edges in $Adj2$ as shown in exercise 22.1-4. Thus the total running time is

$$O(VE) + O(V + VE) = O(VE).$$

  However, if the original graph G contains self-loops, we should modify the algorithm so that self-loops are not removed.

- **Adjacency-matrix representation**

  Let A denote the adjacency-matrix representation of G. The adjacency-matrix representation of G^2 is the square of A. Computing A^2 can be done in time $O(V^3)$ (and even faster, theoretically; Strassen's algorithm for example will compute A^2 in $O(V^{\lg 7})$).

22.1-6

> Most graph algorithms that take an adjacency-matrix representation as input require time $\Omega(V^2)$, but there are some exceptions. Show how to determine whether a directed graph G contains a ***universal sink*** – a vertex with in-degree $|V| - 1$ and out-degree 0 – in time $O(V)$, given an adjacency matrix for G.

Start by examining position $(1, 1)$ in the adjacency matrix. When examining position (i, j),

- if a 1 is encountered, examine position $(i + 1, j)$, and
- if a 0 is encountered, examine position $(i, j + 1)$.

Once either i or j is equal to $|V|$, terminate.

```
IS-CONTAIN-UNIVERSAL-SINK(M)
    i = j = 1
    while i < |V| and j < |V|
        // There's an out-going edge, so examine the next row
        if M[i, j] == 1
            i = i + 1
        // There's no out-going edge, so see if we could reach the last
column of current row
        else if M[i, j] == 0
            j = j + 1
    check if vertex i is a universal sink
```

If a graph contains a universal sink, then it must be at vertex i.

To see this, suppose that vertex k is a universal sink. Since k is a universal sink, row k will be filled with 0's, and column k will be filled with 1's except for $M[k, k]$, which is filled with a 0. Eventually, once row k is hit, the algorithm will continue to increment column j until $j = |V|$.

To be sure that row k is eventually hit, note that once column k is reached, the algorithm will continue to increment i until it reaches k.

This algorithm runs in $O(V)$ and checking if vertex i is a universal sink is done in $O(V)$. Therefore, the total running time is $O(V) + O(V) = O(V)$.

22.1-7

> The ***incidence matrix*** of a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that
>
> $$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i, \\ 1 & \text{if edge } j \text{ enters vertex } i, \\ 0 & \text{otherwise.} \end{cases}$$
>
> Describe what the entries of the matrix product BB^T represent, where B^T is the transpose of B.

$$BB^T(i, j) = \sum_{e \in E} b_{ie} b_{ej}^T = \sum_{e \in E} b_{ie} b_{je}.$$

- If $i = j$, then $b_{ie} b_{je} = 1$ (it is $1 \cdot 1$ or $(-1) \cdot (-1)$) whenever e enters or leaves vertex i, and 0 otherwise.
- If $i \neq j$, then $b_{ie} b_{je} = -1$ when $e = (i, j)$ or $e = (j, i)$, and 0 otherwise.

Thus,

$$BB^T(i, j) = \begin{cases} \text{degree of } i = \text{in-degree} + \text{out-degree} & \text{if } i = j, \\ -(\#\text{ of edges connecting } i \text{ and } j) & \text{if } i \neq j. \end{cases}$$

22.1-8

> Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u, v) \in E$. If all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph? What disadvantages does this scheme have? Suggest an alternate data structure for each edge list that solves these problems. Does your alternative have disadvantages compared to the hash table?

The expected lookup time is $O(1)$, but in the worst case it could take $O(|V|)$.

If we first sorted vertices in each adjacency list then we could perform a binary search so that the worst case lookup time is $O(\lg |V|)$, but this has the disadvantage of having a much worse expected lookup time.

22.2 Breadth-first search

22.2-1

> Show the d and π values that result from running breadth-first search on the directed graph of Figure 22.2(a), using vertex 3 as the source.

vertex	1	2	3	4	5	6
d	∞	3	0	2	1	1
π	NIL	4	NIL	5	3	3

22.2-2

> Show the d and π values that result from running breadth-first search on the undirected graph of Figure 22.3, using vertex u as the source.

vertex	r	s	t	u	v	w	x	y
d	4	3	1	0	5	2	1	1
π	s	w	u	NIL	r	t	u	u

22.2-3

> Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if lines 5 and 14 were removed.

The textbook introduces the GRAY color for the pedagogical purpose to distinguish between the GRAY nodes (which are enqueued) and the BLACK nodes (which are dequeued).

Therefore, it suffices to use a single bit to store each vertex color.

22.2-4

> What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

The time of iterating all edges becomes $O(V^2)$ from $O(E)$. Therefore, the running time is $O(V + V^2) = O(V^2)$.

22.2-5

> Argue that in a breadth-first search, the value $u.d$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list. Using Figure 22.3 as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

First, we will show that the value d assigned to a vertex is independent of the order that entries appear in adjacency lists. To show this, we rely on theorem 22.5, which proves correctness of BFS. In particular, the theorem states that $v.d = \delta(s, v)$ at the termination of BFS. Since $\delta(s, v)$ is a property of the underlying graph, for any adjacency list representation of the graph (including any reordering of the adjacency lists), $\delta(s, v)$ will not change. Since the d values are equal to $\delta(s, v)$ and $\delta(s, v)$ is invariant for any ordering of the adjacency list, d is also not dependent of the ordering of the adjacency list.

Now, to show that π does depend on the ordering of the adjacency lists, we will be using Figure 22.3 as a guide.

First, we note that in the given worked out procedure, we have that in the adjacency list for w, t precedes x. Also, in the worked out procedure, we have that $u.\pi = t$.

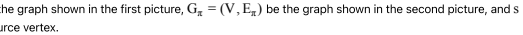
Now, suppose instead that we had x preceding t in the adjacency list of w. Then, it would get added to the queue before t, which means that it would u as it's child before we have a chance to process the children of t. This will mean that $u.\pi = x$ in this different ordering of the adjacency list for w.

22.2-6

> Give an example of a directed graph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique simple path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running BFS on G, no matter how the vertices are ordered in each adjacency list.

Let G be the graph shown in the first picture, $G_\pi = (V, E_\pi)$ be the graph shown in the second picture, and s be the source vertex.

We could see that E_π will never be produced by running BFS on G.




- If y precedes v in the $\mathrm{Adj}[s]$. We'll dequeue y before v, so $u.\pi$ and $x.\pi$ are both y. However, this is not the case.
- If v preceded y in the $\mathrm{Adj}[s]$. We'll dequeue v before y, so $u.\pi$ and $x.\pi$ are both v, which again isn't true.

Nonetheless, the unique simple path in G_π from s to any vertex is a shortest path in G.

22.2-7

> There are two types of professional wrestlers: "babyfaces" ("good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n + r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

This problem is basically just a obfuscated version of two coloring. We will try to color the vertices of this graph of rivalries by two colors, "babyface" and "heel". To have that no two babyfaces and no two heels have a rivalry is the same as saying that the coloring is proper. To two color, we perform a breadth first search of each connected component to get the d values for each vertex. Then, we give all the odd ones one color say "heel", and all the even d values a different color. We know that no other coloring will succeed where this one fails since if we gave any other coloring, we would have that a vertex v has the same color as $v.\pi$ since v and $v.\pi$ must have different parities for their d values. Since we know that there is no better coloring, we just need to check each edge to see if this coloring is valid. If each edge works, it is possible to find a designation, if a single edge fails, then it is not possible. Since the BFS took time $O(n + r)$ and the checking took time $O(r)$, the total runtime is $O(n + r)$.

22.2-8 *

> The ***diameter*** of a tree $T = (V, E)$ is defined as $\max_{u,v \in V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

Suppose that a and b are the endpoints of the path in the tree which achieve the diameter, and without loss of generality assume that a and b are the unique pair which do so. Let s be any vertex in T. We claim that the result of a single BFS will return either a or b (or both) as the vertex whose distance from s is greatest.

To see this, suppose to the contrary that some other vertex x is shown to be furthest from s. (Note that x cannot be on the path from a to b, otherwise we could extend). Then we have

$$d(s, a) < d(s, x)$$

and

$$d(s, b) < d(s, x).$$

Let c denote the vertex on the path from a to b which minimizes $d(s, c)$. Since the graph is in fact a tree, we must have

$$d(s, a) = d(s, c) + d(c, a)$$

and

$$d(s, b) = d(s, c) + d(c, b).$$

(If there were another path, we could form a cycle). Using the triangle inequality and inequalities and equalities mentioned above we must have

$$\begin{aligned} d(a, b) + 2d(s, c) &= d(s, c) + d(c, b) + d(s, c) + d(c, a) \\ &< d(s, x) + d(s, c) + d(c, b). \end{aligned}$$

I claim that $d(x, b) = d(s, x) + d(s, b)$. If not, then by the triangle inequality we must have a strict less-than. In other words, there is some path from x to b which does not go through c. This gives the contradiction, because it implies there is a cycle formed by concatenating these paths. Then we have

$$d(a, b) < d(a, b) + 2d(s, c) < d(x, b).$$

Since it is assumed that $d(a, b)$ is maximal among all pairs, we have a contradiction. Therefore, since trees have $|V| - 1$ edges, we can run BFS a single time in $O(V)$ to obtain one of the vertices which is the endpoint of the longest simple path contained in the graph. Running BFS again will show us where the other one is, so we can solve the diameter problem for trees in $O(V)$.

22.2-9

> Let $G = (V, E)$ be a connected, undirected graph. Give an $O(V + E)$-time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

First, the algorithm computes a minimum spanning tree of the graph. Note that this can be done using the procedures of Chapter 23. It can also be done by performing a breadth first search, and restricting to the edges between v and $v.\pi$ for every v. To aide in not double counting edges, fix any ordering $\leq$ on the vertices before hand. Then, we will construct the sequence of steps by calling MAKE-PATH(s), where s was the root used for the BFS.

```
MAKE-PATH(u)
    for each v ∈ Adj[u] but not in the tree such that u ≤ v
        go to v and back to u
    for each v ∈ Adj[u] but not equal to u.π
        go to v
        perform the path proscribed by MAKE-PATH(v)
    go to u.π
```

22.3 Depth-first search

22.3-1

> Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

According to Theorem 22.7 (Parenthesis theorem), there are 3 cases of relationship between intervals of vertex u and v:

- $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjointed,
- $[u.d, u.f] \subset [v.d, v.f]$, and
- $[v.d, v.f] \subset [u.d, u.f]$.

We judge the possibility according to this Theorem.

- For **directed graph**, we can use the edge classification given by exercise 22.3-5 to simplify the problem.

from \ to	WHITE	GRAY	BLACK
WHITE	All kinds	Cross, Back	Cross
GRAY	Tree, Forward	Tree, Forward, Back	Tree, Forward, Cross
BLACK	–	Back	All kinds

- For **undirected graph**, starting from directed chart, we remove the forward edge and the cross edge, and
  - when a back edge exist, we add a tree edge;
  - when a tree edge exist, we add a back edge.

  This is correct for the following reasons:

  1. Theorem 22.10: In a depth-first search of an undirected graph G, every edge of G is either a tree or back edge. So tree and back edge only.
  2. If (u, v) is a tree edge from u's perspective, (u, v) is also a back edge from v's perspective.

from \ to	WHITE	GRAY	BLACK
WHITE	–	Tree, Back	Tree, Back
GRAY	Tree, Back	Tree, Back	Tree, Back
BLACK	Tree, Back	Tree, Back	–

22.3-2

> Show how depth-first search works on the graph of Figure 22.6. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

The following table gives the discovery time and finish time for each vetex in the graph.

See the C++ demo.

Vertex	Discovered	Finished
q	1	16
r	17	20
s	2	7
t	8	15
u	18	19
v	3	6
w	4	5
x	9	12
y	13	14
z	10	11

- **Tree edges:** (q, s), (s, v), (v, w), (q, t), (t, x), (x, z), (t, y), (r, u).
- **Back edges:** (w, s), (z, x), (y, q).
- **Forward edges:** (q, w).
- **Cross edges:** (r, y), (u, y).

22.3-3

> Show the parenthesis structure of the depth-first search of Figure 22.4.

The parentheses structure of the depth-first search of Figure 22.4 is $(u(v(y(xx)y)v)u)(w(zz)w)$.

22.3-4

> Show that using a single bit to store each vertex color suffices by arguing that the DFS procedure would produce the same result if line 3 of DFS-VISIT was removed.

Change line 3 to `color = BLACK` and remove line 8. Then, the algorithm would produce the same result.

22.3-5

> Show that edge (u, v) is
>
> **a.** a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$,
>
> **b.** a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, and
>
> **c.** a cross edge if and only if $v.d < v.f < u.d < u.f$.

a. u is an ancestor of v.

b. u is a descendant of v.

c. v is visited before u.

22.3-6

> Show that in an undirected graph, classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search is equivalent to classifying it according to the ordering of the four types in the classification scheme.

By Theorem 22.10, every edge of an undirected graph is either a tree edge or a back edge. First suppose that v is first discovered by exploring edge (u, v). Then by definition, (u, v) is a tree edge. Moreover, (u, v) must have been discovered before (v, u) because once (v, u) is explored, v is necessarily discovered. Now suppose that v isn't first discovered by (u, v). Then it must be discovered by (r, v) for some $r \neq u$. If u hasn't yet been discovered then if (u, v) is explored first, it must be a back edge since v is an ancestor of u. If u has been discovered then u is an ancestor of v, so (v, u) is a back edge.

22.3-7

> Rewrite the procedure DFS, using a stack to eliminate recursion.

See the C++ demo.

Also, see this issue for @i-to's discussion.

```
DFS-STACK(G)
    for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    for each vertex u ∈ G.V
        if u.color == WHITE
            DFS-VISIT-STACK(G, u)
```

```
DFS-VISIT-STACK(G, u)
    S = ∅
    PUSH(S, u)
    time = time + 1             // white vertex u has just been discovered
    u.d = time
    u.color = GRAY
    while !STACK-EMPTY(S)
        u = TOP(S)
        v = FIRST-WHITE-NEIGHBOR(G, u)
        if v == NIL
            // u's adjacency list has been fully explored
            POP(S)
            time = time + 1
            u.f = time
            u.color = BLACK     // blackend u; it is finished
        else
            // u's adjacency list hasn't been fully explored
            v.π = u
            time = time + 1
            v.d = time
            v.color = GRAY
            PUSH(S, v)
```

```
FIRST-WHITE-NEIGHBOR(G, u)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            return v
    return NIL
```

22.3-8

> Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, and if $u.d < v.d$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

Consider a graph with 3 vertices u, v, and w, and with edges (w, u), (u, w), and (w, v). Suppose that DFS first explores w, and that w's adjacency list has u before v. We next discover u. The only adjacent vertex is w, but w is already grey, so u finishes. Since v is not yet a descendant of u and u is finished, v can never be a descendant of u.

22.3-9

> Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \leq u.f$.

Consider the directed graph on the vertices $\{1, 2, 3\}$, and having the edges $(1, 2)$, $(1, 3)$, $(2, 1)$ then there is a path from 2 to 3. However, if we start a DFS at 1 and process 2 before 3, we will have $2.f = 3 < 4 = 3.d$ which provides a counterexample to the given conjecture.

22.3-10

> Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type. Show what modifications, if any, you need to make if G is undirected.

If G is undirected we don't need to make any modifications.

See the C++ demo.

```
DFS-VISIT-PRINT(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            print "(u, v) is a tree edge."
            v.π = u
            DFS-VISIT-PRINT(G, v)
        else if v.color == GRAY
            print "(u, v) is a back edge."
        else if v.d > u.d
            print "(u, v) is a forward edge."
        else
            print "(u, v) is a cross edge."
    u.color = BLACK
    time = time + 1
    u.f = time
```

22.3-11

> Explain how a vertex u of a directed graph can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.

Suppose that we have a directed graph on the vertices $\{1, 2, 3\}$ and having edges $(1, 2)$ and $(2, 3)$. Then, 2 has both incoming and outgoing edges.

If we pick our first root to be 3, that will be in its own DFS tree. Then, we pick our second root to be 2, since the only thing it points to has already been marked BLACK, we won't be exploring it. Then, picking the last root to be 1, we don't screw up the fact that 2 is along in a DFS tree even though it has both an incoming and outgoing edge in G.

22.3-12

> Show that we can use a depth-first search of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

The modifications work as follows: each time the **if**-condition of line 8 is satisfied in DFS-CC, we have a new root of a tree in the forest, so we update its cc label to be a new value of k. In the recursive calls to DFS-VISIT-CC, we always update a descendant's connected component to agree with its ancestor's.

See the C++ demo.

```
DFS-CC(G)
    for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    cc = 1
    for each vertex u ∈ G.V
        if u.color == WHITE
            u.cc = cc
            cc = cc + 1
            DFS-VISIT-CC(G, u)
```

```
DFS-VISIT-CC(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.cc = u.cc
            v.π = u
            DFS-VISIT-CC(G, v)
    u.color = BLACK
    time = time + 1
    u.f = time
```

22.3-13 *

> A directed graph $G = (V, E)$ is ***singly connected*** if $u \rightsquigarrow v$ implies that G contains at most one simple path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.

This can be done in time $O(|V||E|)$. To do this, first perform a topological sort of the vertices. Then, we will contain for each vertex a list of it's ancestors with in-degree 0. We compute these lists for each vertex in the order starting from the earlier ones topologically.

Then, if we ever have a vertex that has the same degree 0 vertex appearing in the lists of two of its immediate parents, we know that the graph is not singly connected. however, if at each step we have that at each step all of the parents have disjoint sets of degree 0 vertices as ancestors, the graph is singly connected. Since, for each vertex, the amount of time required is bounded by the number of vertices times the in-degree of the particular vertex, the total runtime is bounded by $O(|V||E|)$.

22.4 Topological sort

22.4-1

> Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

Our start and finish times from performing the DFS are

label	d	f
m	1	20
q	2	5
t	3	4
r	6	19
u	7	8
y	9	18
v	10	17
w	11	14
z	12	13
x	15	16
n	21	26
o	22	25
s	23	24
p	27	28

And so, by reading off the entries in decreasing order of finish time, we have the sequence $p, n, o, s, m, r, y, v, x, w, z, u, q, t$.

22.4-2

> Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex p to vertex v : pov, $poryv$, $posryv$, and $psryv$. (Your algorithm needs only to count the simple paths, not list them.)

The algorithm works as follows. The attribute $u.paths$ of node u tells the number of simple paths from u to v, where we assume that v is fixed throughout the entire process. First of all, a topo sort should be conducted and list the vertex between u, v as $\{v[1], v[2], \ldots, v[k-1]\}$. To count the number of paths, we should construct a solution from v to u. Let's call u as $v[0]$ and v as $v[k]$, to avoid overlapping subproblem, the number of paths between v_k and u should be remembered and used as k decrease to 0. Only in this way can we solve the problem in $\Theta(V + E)$.

An bottom-up iterative version is possible only if the graph uses adjacency matrix so whether v is adjacency to u can be determined in $O(1)$ time. But building a adjacency matrix would cost $\Theta(|V|^2)$, so never mind.

```
SIMPLE-PATHS(G, u, v)
    TOPO-SORT(G)
    let {v[1], v[2]..v[k - 1]} be the vertex between u and v
    v[0] = u
    v[k] = v
    for j = 0 to k - 1
        DP[j] = ∞
    DP[k] = 1
    return SIMPLE-PATHS-AID(G, DP, 0)
```

```
SIMPLE-PATHS-AID(G, DP, i)
    if i > k
        return 0
    else if DP[i] != ∞
        return DP[i]
    else
       DP[i] = 0
       for v[m] in G.adj[v[i]] and 0 < m ≤ k
            DP[i] += SIMPLE-PATHS-AID(G, DP, m)
       return DP[i]
```

22.4-3

> Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

(Removed)

22.4-4

> Prove or disprove: If a directed graph G contains cycles, then TOPOLOGICAL-SORT(G) produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the ordering produced.

This is not true. Consider the graph G consisting of vertices a, b, c, and d. Let the edges be (a, b), (b, c), (a, d), (d, c), and (c, a). Suppose that we start the DFS of TOPOLOGICAL-SORT at vertex c. Assuming that b appears before d in the adjacency list of a, the order, from latest to earliest, of finish times is c, a, d, b.

The "bad" edges in this case are (b, c) and (d, c). However, if we had instead ordered them by a, b, d, c then the only bad edges would be (c, a). Thus TOPOLOGICAL-SORT doesn't always minimizes the number of "bad" edges

22.4-5

> Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

(Removed)

22.5 Strongly connected components

22.5-1

> How can the number of strongly connected components of a graph change if a new edge is added?

It can either stay the same or decrease. To see that it is possible to stay the same, just suppose you add some edge to a cycle. To see that it is possible to decrease, suppose that your original graph is on three vertices, and is just a path passing through all of them, and the edge added completes this path to a cycle. To see that it cannot increase, notice that adding an edge cannot remove any path that existed before.

So, if u and v are in the same connected component in the original graph, then there are a path from one to the other, in both directions. Adding an edge wont disturb these two paths, so we know that u and v will still be in the same SCC in the graph after adding the edge. Since no components can be split apart, this means that the number of them cannot increase since they form a partition of the set of vertices.

22.5-2

> Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the graph of Figure 22.6. Specifically, show the finishing times computed in line 1 and the forest produced in line 3. Assume that the loop of lines 5–7 of DFS considers vertices in alphabetical order and that the adjacency lists are in alphabetical order.

The finishing times of each vertex were computed in exercise 22.3-2. The forest consists of 5 trees, each of which is a chain. We'll list the vertices of each tree in order from root to leaf: $r, u, q - y - t, x - z$, and $s - w - v$.

22.5-3

> Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of *increasing* finishing times. Does this simpler algorithm always produce correct results?

Professor Bacon's suggestion doesn't work out. As an example, suppose that our graph is on the three vertices $\{1, 2, 3\}$ and consists of the edges $(2, 1)$, $(2, 3)$, $(3, 2)$. Then, we should end up with $\{2, 3\}$ and $\{1\}$ as our SCC's. However, a possible DFS starting at 2 could explore 3 before 1, this would mean that the finish time of 3 is lower than of 1 and 2. This means that when we first perform the DFS starting at 3. However, a DFS starting at 3 will be able to reach all other vertices. This means that the algorithm would return that the entire graph is a single SCC, even though this is clearly not the case since there is neither a path from 1 to 2 of from 1 to 3.

22.5-4

> Prove that for any directed graph G, we have $((G^T)^{SCC})^T = G^{SCC}$. That is, the transpose of the component graph of G^T is the same as the component graph of G.

First observe that C is a strongly connected component of G if and only if it is a strongly connected component of G^T. Thus the vertex sets of G^{SCC} and $(G^T)^{SCC}$ are the same, which implies the vertex sets of $((G^T)^{SCC})^T$ and G^{SCC} are the same. It suffices to show that their edge sets are the same. Suppose (v_i, v_j) is an edge in $((G^T)^{SCC})^T$. Then (v_j, v_i) is an edge in $(G^T)^{SCC}$. Thus there exist $x \in C_j$ and $y \in C_i$ such that (x, y) is an edge of G^T, which implies (y, x) is an edge of G. Since components are preserved, this means that (v_i, v_j) is an edge in G^{SCC}. For the opposite implication we simply note that for any graph G we have $(G^T)^T = G$.

22.5-5

> Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

(Removed)

22.5-6

> Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that (a) G' has the same strongly connected components as G, (b) G' has the same component graph as G, and (c) E' is as small as possible. Describe a fast algorithm to compute G'.

(Removed)

22.5-7

> A directed graph $G = (V, E)$ is ***semiconnected*** if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

Algorithm:

1. Run $\text{STRONG-CONNECTED-COMPONENTS}(G)$.
2. Take each strong connected component as a virtual vertex and create a new virtual graph G'.
3. Run $\text{TOPOLOGICAL-SORT}(G')$.
4. Check if for all consecutive vertices (v_i, v_{i+1}) in a topological sort of G', there is an edge (v_i, v_{i+1}) in graph G'. if so, the original graph is semiconnected. Otherwise, it isn't.

Proof:

It is easy to show that G' is a DAG. Consider consecutive vertices v_i and v_{i+1} in G'. **I**f there is no edge from v_i to v_{i+1}, we also conclude that there is no path from v_{i+1} to v_i since v_i finished after v_{i+1}. From the definition of G', we conclude that, there is no path from any vertices in G who is represented as v_i in G' to those represented as v_{i+1}. Thus, G is not semi-connected. **I**f there is an edge between all consecutive vertices, we claim that there is an edge between any two vertices. Therefore, G is semi-connected.

Running-time: $O(V + E)$.

Problem 22-1 Classifying edges by breadth-first search

> A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.
>
> **a.** Prove that in a breadth-first search of an undirected graph, the following properties hold:
>
> 1. There are no back edges and no forward edges.
> 2. For each tree edge (u, v), we have $v.d = u.d + 1$.
> 3. For each cross edge (u, v), we have $v.d = u.d$ or $v.d = u.d + 1$.
>
> **b.** Prove that in a breadth-first search of a directed graph, the following properties hold:
>
> 1. There are no forward edges.
> 2. For each tree edge (u, v), we have $v.d = u.d + 1$.
> 3. For each cross edge (u, v), we have $v.d \leq u.d + 1$.
> 4. For each back edge (u, v), we have $0 \leq v.d \leq u.d$.

a.

1. **I**f we found a back edge, this means that there are two vertices, one a descendant of the other, but there is already a path from the ancestor to the child that doesn't involve moving up the tree. This is a contradiction since the only children in the bfs tree are those that are a single edge away, which means there cannot be any other paths to that child because that would make it more than a single edge away.

   To see that there are no forward edges, We do a similar procedure. A forward edge would mean that from a given vertex we notice it has a child that has already been processed, but this cannot happen because all children are only one edge away, and for it to of already been processed, it would need to have gone through some other vertex first.

2. An edge is placed on the list to be processed if it goes to a vertex that has not yet been considered. This means that the path from that vertex to the root must be at least the distance from the current vertex plus 1. **I**t is also at most that since we can just take the path that consists of going to the current vertex and taking its path to the root.

3. We know that a cross edge cannot be going to a depth more than one less, otherwise it would be used as a tree edge when we were processing that earlier element. **I**t also cannot be going to a vertex of depth more than one more, because we wouldn't of already processed a vertex that was that much further away from the root. Since the depths of the vertices in the cross edge cannot be more than one apart, the conclusion follows by possibly interchanging the roles of u and v, which we can do because the edges are unordered.

b.

1. To have a forward edge, we would need to have already processed a vertex using more than one edge, even though there is a path to it using a single edge. Since breadth first search always considers shorter paths first, this is not possible.

2. Suppose that (u, v) is a tree edge. Then, this means that there is a path from the root to v of length $u.d + 1$ by just appending (u, v) on to the path from the root to u. To see that there is no shorter path, we just note that we would of processed v sooner, and so wouldn't currently have a tree edge if there were.

3. To see this, all we need to do is note that there is some path from the root to v of length $u.d + 1$ obtained by appending (u, v) to $v.d$. Since there is a path of that length, it serves as an upper bound on the minimum length of all such paths from the root to v.

4. **I**t is trivial that $0 \leq v.d$, since it is impossible to have a path from the root to v of negative length. The more interesting inequality is $v.d \leq u.d$. We know that there is some path from v to u, consisting of tree edges, this is the defining property of (u, v) being a back edge. This means that is $v, v_1, v_2, \ldots, v_k, u$ is this path (it is unique because the tree edges form a tree). Then, we have that $u.d = v_k.d + 1 = v_{k-1}.d + 2 = \cdots = v_1.d + k = v.d + k + 1$. So, we have that $u.d > v.d$. **I**n fact, we just showed that we have the stronger conclusion, that $0 \leq v.d < u.d$.

Problem 22-2 Articulation points, bridges, and biconnected components

> Let $G = (V, E)$ be a connected, undirected graph. An ***articulation point*** of G is a vertex whose removal disconnects G. A ***bridge*** of G is an edge whose removal disconnects G. A ***biconnected component*** of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure 22.10 illustrates these definitions. We can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_\pi = (V, E_\pi)$ be a depth-first tree of G.
>
> **a.** Prove that the root of G_π is an articulation point of G if and only if it has at least two children in G_π.
>
> **b.** Let v be a nonroot vertex of G_π. Prove that v is an articulation point of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.
>
> **c.** Let
>
> $$v.\text{low} = \min \begin{cases} v.d, \\ w.d : (u, w) \text{ is a back edge for some descendant } u \text{ of } v. \end{cases}$$
>
> Show how to computer $v.\text{low}$ for all vertices $v \in V$ in $O(E)$ time.
>
> **d.** Show how to compute all articulation points in $O(E)$ time.
>
> **e.** Prove that an edge of G is a bridge if and only if it does not lie on any simple cycle of G.
>
> **f.** Show how to compute all the bridges of G in $O(E)$ time.
>
> **g.** Prove that the biconnected components of G partition the nonbridge edges of G.
>
> **h.** Give an $O(E)$-time algorithm to label each edge e of G with a positive integer $e.\text{bcc}$ such that $e.\text{bcc} = e'.\text{bcc}$ if and only if e and e' are in the same biconnected component.

a. First suppose the root r of G_π is an articulation point. Then the removal of r from G would cause the graph to disconnect, so r has at least 2 children in G. **I**f r has only one child v in G_π then it must be the case that there is a path from v to each of r's other children. Since removing r disconnects the graph, there must exist vertices u and w such that the only paths from u to w contain r.

To reach r from u, the path must first reach one of r's children. This child is connect to v via a path which doesn't contain r.

To reach w, the path must also leave r through one of its children, which is also reachable by v. This implies that there is a path from u to w which doesn't contain r, a contradiction.

Now suppose r has at least two children u and v in G_π. Then there is no path from u to v in G which doesn't go through r, since otherwise u would be an ancestor of v. Thus, removing r disconnects the component containing u and the component containing v, so r is an articulation point.

b. Suppose that v is a nonroot vertex of G_π and that v has a child s such that neither s nor any of s's descendants have back edges to a proper ancestor of v. Let r be an ancestor of v, and remove v from G. Since we are in the undirected case, the only edges in the graph are tree edges or back edges, which means that every edge incident with s takes us to a descendant of s, and no descendants have back edges, so at no point can we move up the tree by taking edges. Therefore r is unreachable from s, so the graph is disconnected and v is an articulation point.

Now suppose that for every child of v there exists a descendant of that child which has a back edge to a proper ancestor of v. Remove v from G. Every subtree of v is a connected component. Within a given subtree, find the vertex which has a back edge to a proper ancestor of v. Since the set T of vertices which aren't descendants of v form a connected component, we have that every subtree of v is connected to T. Thus, the graph remains connected after the deletion of v so v is not an articulation point.

c. Since v is discovered before all of its descendants, the only back edges which could affect $v.\text{low}$ are ones which go from a descendant of v to a proper ancestor of v. **I**f we know $u.\text{low}$ for every child u of v, then we can compute $v.\text{low}$ easily since all the information is coded in its descendants.

Thus, we can write the algorithm recursively: **I**f v is a leaf in G_π then $v.\text{low}$ is the minimum of $v.d$ and $w.d$ where (v, w) is a back edge. **I**f v is not a leaf, v is the minimum of $v.d$, $w.d$ where (v, w) is a back edge, and $u.\text{low}$, where u is a child of v. Computing $v.\text{low}$ for a vertex is linear in its degree. The sum of the vertices' degrees gives twice the number of edges, so the total runtime is $O(E)$.

d. First apply the algorithm of part (c) in $O(E)$ to compute $v.\text{low}$ for all $v \in V$. **I**f $v.\text{low} = v.d$ if and only if no descendant of v has a back edge to a proper ancestor of v, if and only if v is not an articulation point.

Thus, we need only check $v.\text{low}$ versus $v.d$ to decide in constant time whether or not v is an articulation point, so the runtime is $O(E)$.

e. An edge (u, v) lies on a simple cycle if and only if there exists at least one path from u to v which doesn't contain the edge (u, v), if and only if removing (u, v) doesn't disconnect the graph, if and only if (u, v) is not a bridge.

f. A edge (u, v) lies on a simple cycle in an undirected graph if and only if either both of its endpoints are articulation points, or one of its endpoints is an articulation point and the other is a vertex of degree 1. There's also a special case where there's only one edge whose incident vertices are both degree 1. We can check this case in constant time. Since we can compute all articulation points in $O(E)$ and we can decide whether or not a vertex has degree 1 in constant time, we can run the algorithm in part (d) and then decide whether each edge is a bridge in constant time, so we can find all bridges in $O(E)$ time.

g. **I**t is clear that every nonbridge edge is in some biconnected component, so we need to show that if C_1 and C_2 are distinct biconnected components, then they contain no common edges. Suppose to the contrary that (u, v) is in both C_1 and C_2.

Let (a, b) be any edge in C_1 and (c, d) be any edge in C_2.

Then (a, b) lies on a simple cycle with (u, v), consisting of the path

$$a, b, p_1, \ldots, p_k, u, v, p_{k+1}, \ldots, p_n, a.$$

Similarly, (c, d) lies on a simple cycle with (u, v) consisting of the path

$$c, d, q_1, \ldots, q_m, u, v, q_{m+1}, \ldots, q_l, c.$$

This means

$$a, b, p_1, \ldots, p_k, u, q_m, \ldots, q_1, d, c, q_l, \ldots, q_{m+1}, v, p_{k+1}, \ldots, p_n,$$

is a simple cycle containing (a, b) and (c, d), a contradiction. Thus, the biconnected components form a partition.

h. Locate all bridge edges in $O(E)$ time using the algorithm described in part (f). Remove each bridge from E. The biconnected components are now simply the edges in the connected components. Assuming this has been done, run the following algorithm, which clearly runs in $O(|E|)$ where $|E|$ is the number of edges originally in G.

```
VISIT-BCC(G, u, k)
    u.color = GRAY
    for each v ∈ G.Adj[u]
        (u, v).bcc = k
        if v.color == WHITE
            VISIT-BCC(G, v, k)
```

Problem 22-3 Euler tour

> An ***Euler tour*** of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.
>
> **a.** Show that G has an Euler tour if and only if $\text{in-degree}(v) = \text{out-degree}(v)$ for each vertex $v \in V$.
>
> **b.** Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists. (Hint: Merge edge-disjoint cycles.)

a. First, we'll show that it is necessary to have in degree equal out degree for each vertex. Suppose that there was some vertex v for which the two were not equal, suppose that $\text{in-degree}(v) - \text{out-degree}(v)$. Note that we may assume that in degree is greater because otherwise we would just look at the transpose graph in which we traverse the cycle backwards. **I**f v is the start of the cycle as it is listed, just shift the starting and ending vertex to any other one on the cycle. Then, in whatever cycle we take going though v, we must pass through v some number of times, in particular, after we pass through it a times, the number of unused edges coming out of v is zero, however, there are still unused edges goin in that we need to use. This means that there is no hope of using those while still being a tour, becase we would never be able to escape v and get back to the vertex where the tour started. Now, we show that it is sufficient to have the in degree and out degree equal for every vertex. To do this, we will generalize the problem slightly so that it is more amenable to an inductive approach. That is, we will show that for every graph G that has two vertices v and u so that all the vertices have the same in and out degree except that the indegree is one greater for u and the out degree is one greater for v, then there is an Euler path from v to u. This clearly lines up with the original statement if we pick $u = v$ to be any vertex in the graph. We now perform induction on the number of edges. **I**f there is only a single edge, then taking just that edge is an Euler tour. Then, suppose that we start at v and take any edge coming out of it.

Consider the graph that is obtained from removing that edge, it inductively contains an Euler tour that we can just post-pend to the edge that we took to get out of v.

b. To actually get the Euler circuit, we can just arbitrarily walk any way that we want so long as we don't repeat an edge, we will necessarily end up with a valid Euler tour. This is implemented in the following algorithm, $\text{EULER-TOUR}(G)$ which takes time $O(|E|)$. **I**t has this runtime because the for loop will get run for every edge, and takes a constant amount of time. Also, the process of initializing each edge's color will take time proportional to the number of edges.

```
EULER-TOUR(G)
    color all edges WHITE
    let (v, u) be any edge
    let L be a list containing v
    while there is some WHITE edge (v, w) coming out of v
        color (v, w) BLACK
        v = w
        append v to L
```

Problem 22-4 Reachability

> Let $G = (V, E)$ be a directed graph in which each vertex $u \in V$ is labeled with a unique integer $L(U)$ from the set $\{1, 2, \ldots, |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V : u \rightsquigarrow v\}$ be the set of vertices that are reachable from u. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, i.e., $\min(u)$ is the vertex v such that $L(v) = \min\{L(w) : w \in R(u)\}$. Give an $O(V + E)$-time algorithm that computes $\min(u)$ for all vertices $u \in V$.

1. Compute the component graph G^{SCC} (in order to remove simple cycles from graph G), and label each vertex in G^{SCC} with the smallest label of vertex in that G^{SCC}. Following chapter 22.5 the time complexity of this procedure is $O(V + E)$.

2. On G^{SCC}, execute the below algorithm. Notice that if we memorize this function it will be invoked at most $V + E$ times. **I**ts time complexity is also $O(V + E)$.

```
REACHABILITY(u)
    u.min = u.label
    for each v ∈ Adj[u]
        u.min = min(u.min, REACHABILITY(v))
    return u.min
```

3. Back to graph G, the value of $\min(u)$ on Graph G is the value of $\min(u.\text{scc})$ on Graph G^{SCC}.

Alternate solution: Transpose the graph. Call DFS, but in the main loop of DFS, consider the vertices in order of their labels. **I**n the DFS-VISIT subroutine, upon discovering a new node, we set its $\min$ to be the label of

its root.

23 Minimum Spanning Trees

23.1 Growing a minimum spanning tree

23.1-1

> Let (u, v) be a minimum-weight edge in a connected graph G. Show that (u, v) belongs to some minimum spanning tree of G.

(Removed)

23.1-2

> Professor Sabatier conjectures the following converse of Theorem 23.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

Let G be the graph with 4 vertices: u, v, w, z. Let the edges of the graph be $(u, v), (u, w), (w, z)$ with weights 3, 1, and 2 respectively.

Suppose A is the set $\{(u, w)\}$. Let $S = A$. Then S clearly respects A. Since G is a tree, its minimum spanning tree is itself, so A is trivially a subset of a minimum spanning tree.

Moreover, every edge is safe. In particular, (u, v) is safe but not a light edge for the cut. Therefore Professor Sabatier's conjecture is false.

23.1-3

> Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

Let T_0 and T_1 be the two trees that are obtained by removing edge (u, v) from a MST. Suppose that V_0 and V_1 are the vertices of T_0 and T_1 respectively.

Consider the cut which separates V_0 from V_1. Suppose to a contradiction that there is some edge that has weight less than that of (u, v) in this cut. Then, we could construct a minimum spanning tree of the whole graph by adding that edge to $T_1 \cup T_0$. This would result in a minimum spanning tree that has weight less than the original minimum spanning tree that contained (u, v).

23.1-4

> Give a simple example of a connected graph such that the set of edges $\{(u, v)$ there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing $(S, V - S)\}$ does not form a minimum spanning tree.

(Removed)

23.1-5

> Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of G. That is, there is a minimum spanning tree of G that does not include e.

Let A be any cut that causes some vertices in the cycle on once side of the cut, and some vertices in the cycle on the other. For any of these cuts, we know that the edge e is not a light edge for this cut. Since all the other cuts won't have the edge e crossing it, we won't have that the edge is light for any of those cuts either. This means that we have that e is not safe.

23.1-6

> Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

(Removed)

23.1-7

> Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

First, we show that the subset of edges of minimum total weight that connects all the vertices is a tree. To see this, suppose not, that it had a cycle. This would mean that removing any of the edges in this cycle would mean that the remaining edges would still connect all the vertices, but would have a total weight that's less by the weight of the edge that was removed. This would contradict the minimality of the total weight of the subset of vertices. Since the subset of edges forms a tree, and has minimal total weight, it must also be a minimum spanning tree.

To see that this conclusion is not true if we allow negative edge weights, we provide a construction. Consider the graph K_3 with all edge weights equal to -1. The only minimum weight set of edges that connects the graph has total weight -3, and consists of all the edges. This is clearly not a MST because it is not a tree, which can be easily seen because it has one more edge than a tree on three vertices should have. Any MST of this weighted graph must have weight that is at least -2.

23.1-8

> Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.

Suppose that L' is another sorted list of edge weights of a minimum spanning tree. If $L' \neq L$, there must be a first edge (u, v) in T or T' which is of smaller weight than the corresponding edge (x, y) in the other set. Without loss of generality, assume (u, v) is in T.

Let C be the graph obtained by adding (u, v) to L'. Then we must have introduced a cycle. If there exists an edge on that cycle which is of larger weight than (u, v), we can remove it to obtain a tree C' of weight strictly smaller than the weight of T', contradicting the fact that T' is a minimum spanning tree.

Thus, every edge on the cycle must be of lesser or equal weight than (u, v). Suppose that every edge is of strictly smaller weight. Remove (u, v) from T to disconnect it into two components. There must exist some edge besides (u, v) on the cycle which would connect these, and since it has smaller weight we can use that edge instead to create a spanning tree with less weight than T, a contradiction. Thus, some edge on the cycle has the same weight as (u, v). Replace that edge by (u, v). The corresponding lists L and L' remain unchanged since we have swapped out an edge of equal weight, but the number of edges which T and T' have in common has increased by 1.

If we continue in this way, eventually they must have every edge in common, contradicting the fact that their edge weights differ somewhere. Therefore all minimum spanning trees have the same sorted list of edge weights.

23.1-9

> Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V. Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

Suppose that there was some cheaper spanning tree than T'. That is, we have that there is some T'' so that $w(T'') < w(T')$. Then, let S be the edges in T but not in T'. We can then construct a minimum spanning tree of G by considering $S \cup T'$ is, and T'' makes all the vertices in V' connected just like T' does.

However, we have that

$$w(S \cup T'') = w(S) + w(T'') < w(S) + w(T') = w(S \cup T') = w(T).$$

This means that we just found a spanning tree that has a lower total weight than a minimum spanning tree. This is a contradiction, and so our assumption that there was a spanning tree of V' cheaper than T' must be false.

23.1-10

> Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges in T. Show that T is still a minimum spanning tree for G. More formally, let T be a minimum spanning tree for G with edge weights given by weight function w. Choose one edge $(x, y) \in T$ and a positive number k, and define the weight function w' by
>
> $$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y), \\ w(x, y) - k & \text{if } (u, v) = (x, y). \end{cases}$$
>
> Show that T is a minimum spanning tree for G with edge weights given by w'.

(Removed)

23.1-11 ⋆

> Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges not in T. Give an algorithm for finding the minimum spanning tree in the modified graph.

If we were to add in this newly decreased edge to the given tree, we would be creating a cycle. Then, if we were to remove any one of the edges along this cycle, we would still have a spanning tree. This means that we look at all the weights along this cycle formed by adding in the decreased edge, and remove the edge in the cycle of maximum weight. This does exactly what we want since we could only possibly want to add in the single decreased edge, and then, from there we change the graph back to a tree in the way that makes its total weight minimized.

23.2 The algorithms of Kruskal and Prim

23.2-1

> Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.

Suppose that we wanted to pick T as our minimum spanning tree. Then, to obtain this tree with Kruskal's algorithm, we will order the edges first by their weight, but then will resolve ties in edge weights by picking an edge first if it is contained in the minimum spanning tree, and treating all the edges that aren't in T as being slightly larger, even though they have the same actual weight.

With this ordering, we will still be finding a tree of the same weight as all the minimum spanning trees $w(T)$. However, since we prioritize the edges in T, we have that we will pick them over any other edges that may be in other minimum spanning trees.

23.2-2

> Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

At each step of the algorithm we will add an edge from a vertex in the tree created so far to a vertex not in the tree, such that this edge has minimum weight. Thus, it will be useful to know, for each vertex not in the tree, the edge from that vertex to some vertex in the tree of minimal weight. We will store this information in an array A, where $A[u] = (v, w)$ if w is the weight of (u, v) and is minimal among the weights of edges from u to some vertex v in the tree built so far. We'll use $A[u].1$ to access v and $A[u].2$ to access w.

```
PRIM-ADJ(G, w, r)
    initialize A with every entry = (NIL, ∞)
    T = {r}
    for i = 1 to V
        if Adj[r, i] != 0
            A[i] = (r, w(r, i))
    for each u in V - T
        k = min(A[i].2)
        T = T u {k}
        k.π = A[k].1
        for i = 1 to V
            if Adf[k, i] != 0 and Adj[k, i] < A[i].2
                A[i] = (k, Adj[k, i])
```

23.2-3

> For a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$, is the implementation of Prim's algorithm with a Fibonacci heap asymptotically faster than the binary-heap implementation? What about for a dense graph, where $|E| = \Theta(V^2)$? How must the sizes $|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

Prim's algorithm implemented with a Binary heap has runtime $O((V + E) \lg V)$, which in the sparse case, is just $O(V \lg V)$. The implementation with Fibonacci heaps is

$$O(E + V \lg V) = O(V + V \lg V) = O(V \lg V).$$

- In the sparse case, the two algorithms have the same asymptotic runtimes.
- In the dense case.
    - The binary heap implementation has a runtime of
    $$O((V + E) \lg V) = O((V + V^2) \lg V) = O(V^2 \lg V).$$
    - The Fibonacci heap implementation has a runtime of
    $$O(E + V \lg V) = O(V^2 + V \lg V) = O(V^2).$$

  So, in the dense case, we have that the Fibonacci heap implementation is asymptotically faster.
- The Fibonacci heap implementation will be asymptotically faster so long as $E = \omega(V)$. Suppose that we have some function that grows more quickly than linear, say f, and $E = f(V)$.
- The binary heap implementation will have runtime of
$$O((V + E) \lg V) = O((V + f(V)) \lg V) = O(f(V) \lg V).$$

However, we have that the runtime of the Fibonacci heap implementation will have runtime of

$$O(E + V \lg V) = O(f(V) + V \lg V).$$

This runtime is either $O(f(V))$ or $O(V \lg V)$ depending on if $f(V)$ grows more or less quickly than $V \lg V$ respectively.

In either case, we have that the runtime is faster than $O(f(V) \lg V)$.

23.2-4

> Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

(Removed)

23.2-5

> Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

For the first case, we can use a van Emde Boas tree to improve the time bound to $O(E \lg \lg V)$. Comparing to the Fibonacci heap implementation, this improves the asymptotic running time only for sparse graphs, and it cannot improve the running time polynomially. An advantage of this implementation is that it may have a lower overhead.

For the second case, we can use a collection of doubly linked lists, each corresponding to an edge weight. This improves the bound to $O(E)$.

23.2-6 ⋆

> Suppose that the edge weights in a graph are uniformly distributed over the halfopen interval $[0, 1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?

For input drawn from a uniform distribution I would use bucket sort with Kruskal's algorithm, for expected linear time sorting of edges by weight. This would achieve expected runtime $O(E\alpha(V))$.

23.2-7 ⋆

> Suppose that a graph G has a minimum spanning tree already computed. How quickly can we update the minimum spanning tree if we add a new vertex and incident edges to G?

(Removed)

23.2-8

> Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1
>
> and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.
>
> Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

The algorithm fails. Suppose $E = \{(u, v), (u, w), (v, w)\}$, the weight of (u, v) and (u, w) is 1, and the weight of (v, w) is 1000, partition the set into two sets $V_1 = \{u\}$ and $V_2 = \{v, w\}$.

Problem 23-1 Second-best minimum spanning tree

> Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.
>
> We define a second-best minimum spanning tree as follows. Let $\square$ be the set of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** is a spanning tree T such that $w(T) = \min_{T'' \in \square - \{T'\}} \{w(T'')\}$.
>
> **a.** Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.
>
> **b.** Let T be the minimum spanning tree of G. Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.
>
> **c.** Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $\max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $\max[u, v]$ for all $u, v \in V$.
>
> **d.** Give an efficient algorithm to compute the second-best minimum spanning tree of G.

a. To see that the second best minimum spanning tree need not be unique, we consider the following example graph on four vertices. Suppose the vertices are {a, b, c, d}, and the edge weights are as follows:

	a	b	c	d
a	–	1	4	3
b	1	–	5	2
c	4	5	–	6
d	3	2	6	–

Then, the minimum spanning tree has weight 7, but there are two spanning trees of the second best weight, 8.

b. We are trying to show that there is a single edge swap that can demote our minimum spanning tree to a second best minimum spanning tree. In obtaining the second best minimum spanning tree, there must be some cut of a single vertex away from the rest for which the edge that is added is not light, otherwise, we would find the minimum spanning tree, not the second best minimum spanning tree. Call the edge that is selected for that cut for the second best minimum spanning tree (x, y). Now, consider the same cut, except look at the edge that was selected when obtaining T, call it (u, v). Then, we have that if consider $T - \{(u, v)\} \cup \{(x, y)\}$, it will be a second best minimum spanning tree. This is because if the second best minimum spanning tree also selected a non-light edge for another cut, it would end up more expensive than all the minimum spanning trees. This means that we need for every cut other than the one that the selected edge was light. This means that the choices all align with what the minimum spanning tree was.

c. We give here a dynamic programming solution. Suppose that we want to find it for (u, v). First, we will identify the vertex x that occurs immediately after u on the simple path from u to v. We will then make $\max[u, v]$ equal to the max of $w((u, x))$ and $\max[w, v]$. Lastly, we just consider the case that u and v are adjacent, in which case the maximum weight edge is just the single edge between the two. If we can find x in constant time, then we will have the whole dynamic program running in time $O(V^2)$, since that's the size of the table that's being built up. To find x in constant time, we preprocess the tree. We first pick an arbitrary root. Then, we do the preprocessing for Tarjan's off-line least common ancestors algorithm (See problem 21-3). This takes time just a little more than linear, $O(|V|\alpha(|V|))$. Once we've computed all the least common ancestors, we can just look up that result at some point later in constant time. Then, to find the w that we should pick, we first see if $u = LCA(u, v)$ if it does not, then we just pick the parent of u in the tree. If it does, then we flip the question on its head and try to compute $\max[v, u]$, we are guaranteed to not have this situation of $v = LCA(v, u)$ because we know that u is an ancestor of v.

d. We provide here an algorithm that takes time $O(V^2)$ and leave open if there exists a linear time solution, that is a $O(E + V)$ time solution. First, we find a minimum spanning tree in time $O(E + V \lg(V))$, which is in $O(V^2)$. Then, using the algorithm from part c, we find the double array max. Then, we take a running minimum over all pairs of vertices u, v, of the value of $w(u, v) - \max[u, v]$. If there is no edge between u and v, we think of the weight being infinite. Then, for the pair that resulted in the minimum value of this difference, we add in that edge and remove from the minimum spanning tree, an edge that is in the path from u to v that has weight $\max[u, v]$.

Problem 23-2 Minimum spanning tree in sparse graphs

> For a very sparse connected graph $G = (V, E)$, we can further improve upon the $O(E + V \lg V)$ running time of Prim's algorithm with Fibonacci heaps by preprocessing G to decrease the number of vertices before running Prim's algorithm. In particular, we choose, for each vertex u, the minimum-weight edge (u, v) incident on u, and we put (u, v) into the minimum spanning tree under construction. We then contract all chosen edges (see Section B.4). Rather than contracting these edges one at a time, we first identify sets of vertices that are united into the same new vertex. Then we create the graph that would have resulted from contracting these edges one at a time, but we do so by "renaming" edges according to the sets into which their endpoints were placed. Several edges from the original graph may be renamed the same as each other. In such a case, only one edge results, and its weight is the minimum of the weights of the corresponding original edges.

> Initially, we set the minimum spanning tree T being constructed to be empty, and for each edge $(u, v) \in E$, we initialize the attributes $(u, v).orig = (u, v)$ and $(u, v).c = w(u, v)$. We use the $orig$ attribute to reference the edge from the initial graph that is associated with an edge in the contracted graph. The c attribute holds the weight of an edge, and as edges are contracted, we update it according to the above scheme for choosing edge weights. The procedure MST-REDUCE takes inputs G and T, and it returns a contracted graph G' with updated attributes $orig'$ and c'. The procedure also accumulates edges of G into the minimum spanning tree T.
>
> ```
> MST-REDUCE(G, T)
>     for each v ∈ G.V
>         v.mark = false
>         MAKE-SET(v)
>     for each u ∈ G.V
>         if u.mark == false
>             choose v ∈ G.Adj[u] such that (u, v).c is minimized
>                 UNION(u, v)
>                 T = T ∪ {(u, v).orig}
>                 u.mark = v.mark = true
>     G'.V = {FIND-SET(v): v ∈ G.V}
>     G'.E = ∅
>     for each (x, y) ∈ G.E
>         u = FIND-SET(x)
>         v = FIND-SET(y)
>         if (u, v) ∉ G'.E
>              G'.E = G'.E ∪ {(u, v)}
>              (u, v).orig' = (x, y).orig
>              (u, v).c' = (x, y).c
>         else if (x, y).c < (u, v).c'
>              (u, v).orig' = (x, y).orig
>              (u, v).c' = (x, y).c
>     construct adjacency lists G'.Adj for G'
>     return G' and T
> ```
>
> **a.** Let T be the set of edges returned by MST-REDUCE, and let A be the minimum spanning tree of the graph G' formed by the call MST-PRIM(G', c', r), where c' is the weight attribute on the edges of $G'.E$ and r is any vertex in $G'.V$. Prove that $T \cup \{(x, y).orig' : (x, y) \in A\}$ is a minimum spanning tree of G.
>
> **b.** Argue that $|G'.V| \leq |V|/2$.
>
> **c.** Show how to implement MST-REDUCE so that it runs in $O(E)$ time. (Hint: Use simple data structures.)
>
> **d.** Suppose that we run k phases of MST-REDUCE, using the output G' produced by one phase as the input G to the next phase and accumulating edges in T. Argue that the overall running time of the k phases is $O(kE)$.
>
> **e.** Suppose that after running k phases of MST-REDUCE, as in part (d), we run Prim's algorithm by calling MST-PRIM(G', c', r), where G', with weight attribute c', is returned by the last phase and r is any vertex in $G'.V$. Show how to pick k so that the overall running time is $O(E \lg \lg V)$. Argue that your choice of k minimizes the overall asymptotic running time.
>
> **f.** For what values of $|E|$ (in terms of $|V|$) does Prim's algorithm with preprocessing asymptotically beat Prim's algorithm without preprocessing?

a. We'll show that the edges added at each step are safe. Consider an unmarked vertex u. Set $S = \{u\}$ and let A be the set of edges in the tree so far. Then the cut respects A, and the next edge we add is a light edge, so it is safe for A. Thus, every edge in T before we run Prim's algorithm is safe for T. Any edge that Prim's would normally add at this point would have to connect two of the trees already created, and it would be chosen as minimal. Moreover, we choose exactly one between any two trees. Thus, the fact that we only have the smallest edges available to us is not a problem. The resulting tree must be minimal.

b. We argue by induction on the number of vertices in G. We'll assume that $|V| > 1$, since otherwise MST-REDUCE will encounter an error on line 6 because there is no way to choose v. Let $|V| = 2$. Since G is connected, there must be an edge between u and v, and it is trivially of minimum weight. They are joined, and $|G'.V| = 1 = |V|/2$.

Suppose the claim holds for $|V| = n$. Let G be a connected graph on $n + 1$ vertices. Then $G'.V \leq n/2$ prior to the final vertex v being examined in the for-loop of line 4. If v is marked then we're done, and if v isn't marked then we'll connect it to some other vertex, which must be marked since v is the last to be processed.

Either way, v can't contribute an additional vertex to $G'.V$. so

$$|G'.V| \leq n/2 \leq (n + 1)/2.$$

c. Rather than using the disjoint set structures of chapter 21, we can simply use an array to keep track of which component a vertex is in. Let A be an array of length $|V|$ such that $A[u] = v$ if $v = \text{FIND-SET}(u)$. Then FIND-SET(u) can now be replaced with $A[u]$ and UNION(u, v) can be replaced by $A[v] = A[u]$. Since these operations run in constant time, the runtime is $O(E)$.

d. The number of edges in the output is monotonically decreasing, so each call is $O(E)$. Thus, k calls take $O(kE)$ time.

e. The runtime of Prim's algorithm is $O(E + V \lg V)$. Each time we run MST-REDUCE, we cut the number of vertices at least in half. Thus, after k calls, the number of vertices is at most $|V|/2^k$. We need to minimize

$$E + V/2^k \lg(V/2^k) + kE = E + \frac{V \lg V}{2^k} - \frac{Vk}{2^k} + kE$$

with respect to k. If we choose $k = \lg \lg V$ then we achieve the overall running time of $O(E \lg \lg V)$ as desired.

To see that this value of k minimizes, note that the $\frac{Vk}{2^k}$ term is always less than the kE term since $E \geq V$. As k decreases, the contribution of kE decreases, and the contribution of $\frac{V \lg V}{2^k}$ increases. Thus, we need to find the value of k which makes them approximately equal in the worst case, when $E = V$. To do this, we set $\frac{\lg V}{2^k} = k$. Solving this exactly would involve the Lambert W function, but the nicest elementary function which gets close is $k = \lg \lg V$.

f. We simply set up the inequality

$$E \lg \lg V < E + V \lg V$$

to find that we need

$$E < \frac{V \lg V}{\lg \lg V - 1} = O(\frac{V \lg V}{\lg \lg V}).$$

Problem 23-3 Bottleneck spanning tree

> A ***bottleneck spanning tree*** T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G. We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in T.
>
> **a.** Argue that a minimum spanning tree is a bottleneck spanning tree.
>
> Part (a) shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, we will show how to find a bottleneck spanning tree in linear time.
>
> **b.** Give a linear-time algorithm that given a graph G and an integer b, determines whether the value of the bottleneck spanning tree is at most b.
>
> **c.** Use your algorithm for part (b) as a subroutine in a linear-time algorithm for the bottleneck-spanning-tree problem. (Hint: You may want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 23-2.)

a. To see that every minimum spanning tree is also a bottleneck spanning tree. Suppose that T is a minimum spanning tree. Suppose there is some edge in it (u, v) that has a weight that's greater than the weight of the bottleneck spanning tree. Then, let V_1 be the subset of vertices of V that are reachable from u in T, without going though v. Define V_2 symmetrically. Then, consider the cut that separates V_1 from V_2. The only edge that we could add across this cut is the one of minimum weight, so we know that there are no edge across this cut of weight less than $w(u, v)$.

However, we have that there is a bottleneck spanning tree with less than that weight. This is a contradiction because a bottleneck spanning tree, since it is a spanning tree, must have an edge across this cut.

b. To do this, we first process the entire graph, and remove any edges that have weight greater than b. If the remaining graph is connected, we can just arbitrarily select any tree in it, and it will be a bottleneck spanning tree of weight at most b. Testing connectivity of a graph can be done in linear time by running a breadth first search and then making sure that no vertices remain white at the end.

c. Write down all of the edge weights of vertices. Use the algorithm from section 9.3 to find the median of this list of numbers in time $O(E)$. Then, run the procedure from part b with this median value as input. Then there are two cases:

First, we could have that there is a bottleneck spanning tree with weight at most this median. Then just throw away the edges with weight more than the median, and repeat the procedure on this new graph with half the edges.

Second, we could have that there is no bottleneck spanning tree with at most that weight. Then, we should run a procedure similar to problem 23-2 to contract all of the edges that have weight at most the weight of the median. This takes time $O(E)$ and then we are left solving the problem on a graph that now has half the edges.

Observe that both cases are $O(E)$ and each recursion reduces the problem size into half. The solution to this recurrence is therefore linear.

Problem 23-4 Alternative minimum-spanning-tree algorithms

In this problem, we give pseudocode for three different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Also describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.

a.

```
MAYBE-MST-A(G, w)
    sort the edges into nonincreasing order of edge weights w
    T = E
    for each edge e, taken in nonincreasing order by weight
        if T − {e} is a connected graph
            T = T − {e}
    return T
```

b.

```
MAYBE-MST-B(G, w)
    T = ∅
    for each edge e, taken in arbitrary order
        if T ∪ {e} has no cycles
            T = T ∪ {e}
    return T
```

c.

```
MAYBE-MST-C(G, w)
    T = ∅
    for each edge e, taken in arbitrary order
        T = T ∪ {e}
        if T has a cycle c
            let e' be a maximum-weight edge on c
            T = T − {e}
    return T
```

a. This does return an MST. To see this, we'll show that we never remove an edge which must be part of a minimum spanning tree. If we remove e, then e cannot be a bridge, which means that e lies on a simple cycle of the graph. Since we remove edges in nonincreasing order, the weight of every edge on the cycle must be less than or equal to that of e. By exercise 23.1-5, there is a minimum spanning tree on G with edge e removed.

To implement this, we begin by sorting the edges in $O(E \lg E)$ time. For each edge we need to check whether or not $T - e$ is connected, so we'll need to run a DFS. Each one takes $O(V + E)$, so doing this for all edges takes $O(E(V + E))$. This dominates the running time, so the total time is $O(E^2)$.

b. This doesn't return an MST. To see this, let G be the graph on 3 vertices a, b, and c. Let the eges be (a, b), (b, c), and (c, a) with weights $3, 2$, and 1 respectively. If the algorithm examines the edges in their order listed, it will take the two heaviest edges instead of the two lightest.

An efficient implementation will use disjoint sets to keep track of connected components, as in MST-REDUCE in problem 23-2. Trying to union within the same component will create a cycle. Since we make $|V|$ calls to MAKESET and at most $3|E|$ calls to FIND-SET and UNION, the runtime is $O(Ea(V))$.

c. This does return an MST. To see this, we simply quote the result from exercise 23.1-5. The only edges we remove are the edges of maximum weight on some cycle, and there always exists a minimum spanning tree which doesn't include these edges. Moreover, if we remove an edge from every cycle then the resulting graph cannot have any cycles, so it must be a tree.

To implement this, we use the approach taken in part (b), except now we also need to find the maximum weight edge on a cycle. For each edge which introduces a cycle we can perform a DFS to find the cycle and max weight edge. Since the tree at that time has at most one cycle, it has at most $|V|$ edges, so we can run DFS in $O(V)$. The runtime is thus $O(EV)$.

24 Single-Source Shortest Paths

24.1 The Bellman-Ford algorithm

24.1-1

Run the Bellman-Ford algorithm on the directed graph of Figure 24.4, using vertex z as the source. In each pass, relax edges in the same order as in the figure, and show the d and π values after each pass. Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

- Using vertex z as the source:
  - d values:

s	t	x	y	z
∞	∞	∞	∞	0
2	∞	7	∞	0
2	5	7	9	0
2	5	6	9	0
2	4	6	9	0

  - π values:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
z	NIL	z	NIL	NIL
z	x	z	s	NIL
z	x	y	s	NIL
z	x	y	s	NIL

- Changing the weight of edge (z, x) to 4:
  - d values:

s	t	x	y	z
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	−2

  - π values:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
NIL	s	NIL	s	NIL
NIL	s	y	s	t
NIL	x	y	s	t
NIL	x	y	s	t

Consider edge (z, x), it'll return FALSE since $x.d = 4 > z.d + w(z, x) = -2 + 4$.

24.1-2

Prove Corollary 24.3.

Suppose there is a path from s to v. Then there must be a shortest such path of length $\delta(s, v)$. It must have finite length since it contains at most $|V| - 1$ edges and each edge has finite length. By Lemma 24.2, $v.d = \delta(s, v) < \infty$ upon termination.

On the other hand, suppose $v.d < \infty$ when BELLMAN-FORD terminates. Recall that $v.d$ is monotonically decreasing throughout the algorithm, and RELAX will update $v.d$ only if $u.d + w(u, v) < v.d$ for some u adjacent to v. Moreover, we update $v.\pi = u$ at this point, so v has an ancestor in the predecessor subgraph. Since this is a tree rooted at s, there must be a path from s to v in this tree. Every edge in the tree is also an edge in G, so there is also a path in G from s to v.

24.1-3

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

By the upper bound theory, we know that after m iterations, no d values will ever change. Therefore, no d values will change in the $(m + 1)$-th iteration. However, we do not know the exact m value in advance, we cannot make the algorithm iterate exactly m times and then terminate. If we try to make the algorithm stop when every d values do not change anymore, then it will stop after $m + 1$ iterations.

24.1-4

Modify the Bellman-Ford algorithm so that it sets $v.d$ to $-\infty$ for all vertices v for which there is a negative-weight cycle on some path from the source to v.

```
BELLMAN-FORD'(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V| − 1
        for each edge (u, v) ∈ G.E
            RELAX(u, v, w)
    for each edge(u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            mark v
    for each vertex u ∈ marked vertices
        DFS-MARK(u)
```

```
DFS-MARK(u)
    if u != NIL and u.d != −∞
        u.d = −∞
        for each v in G.Adj[u]
            DFS-MARK(v)
```

After running BELLMAN-FORD', run DFS with all vertices on negative-weight cycles as source vertices. All the vertices that can be reached from these vertices should have their d attributes set to $-\infty$.

24.1-5 *

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Give an $O(VE)$-time algorithm to find, for each vertex $v \in V$, the value $\delta^*(v) = \min_{u \in V}\{\delta(u, v)\}$.

```
RELAX(u, v, w)
    if v.d > min(w(u, v), w(u, v) + u.d)
        v.d = min(w(u, v), w(u, v) + u.d)
        v.π = u.π
```

24.1-6 *

Suppose that a weighted, directed graph $G = (V, E)$ has a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

Based on exercise 24.1-4, DFS from a vertex u that $u.d = -\infty$, if the weight sum on the search path is negative and the next vertex is BLACK, then the search path forms a negative-weight cycle.

24.2 Single-source shortest paths in directed acyclic graphs

24.2-1

Run DAG-SHORTEST-PATHS on the directed graph of Figure 24.5, using vertex r as the source.

- d values:

r	s	t	x	y	z
0	∞	∞	∞	∞	∞
0	5	3	∞	∞	∞
0	5	3	11	∞	∞
0	5	3	10	7	5
0	5	3	10	7	5
0	5	3	10	7	5

- π values:

r	s	t	x	y	z
NIL	NIL	NIL	NIL	NIL	NIL
NIL	r	r	NIL	NIL	NIL
NIL	r	r	s	NIL	NIL
NIL	r	r	t	t	t
NIL	r	r	t	t	t
NIL	r	r	t	t	t

24.2-2

Suppose we change line 3 of DAG-SHORTEST-PATHS to read

```
3  for the first |V| − 1 vertices, taken in topologically sorted order
```

Show that the procedure would remain correct.

When we reach vertex v, the last vertex in the topological sort, it must have out-degree 0. Otherwise there would be an edge pointing from a later vertex to an earlier vertex in the ordering, a contradiction. Thus, the body of the for-loop of line 4 is never entered for this final vertex, so we may as well not consider it.

24.2-3

The PERT chart formulation given above is somewhat unnatural. In a more natural structure, vertices would represent jobs and edges would represent sequencing constraints; that is, edge (u, v) would indicate that job u must be performed before job v. We would then assign weights to vertices, not edges. Modify the DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

There are two ways to transform a PERT chart $G = (V, E)$ with weights on the vertices to a PERT chart $G' = (V', E')$ with weights on edges. Both ways satisfy $|V'| \le 2|V|$ and $|E'| \le |V| + |E|$, so we can scan G' using the same algorithm to find the longest path through a directed acyclic graph.

In the first way, we transform each vertex $v \in V$ into two vertices v' and v'' in V'. All edges in E that enters V will also enter V' in E', and all edges in E that leaves V will leave V'' in E'. Thus, if $(u, v) \in E$, then $(u'', v') \in E'$. All such edges have weight 0, so we can put edges (v', v'') into E' for all vertices $v \in V$, and these edges are given the weight of the corresponding vertex v in G. Finally, we get $|V'| \leq 2|V|$ and $|E'| \leq |V| + |E|$, and the edge weight of each path in G' equals the vertex weight of the corresponding path in G.

In the second way, we leave vertices in V, but try to add one new source vertex s to V', given that $V' = V \cup \{s\}$. All edges of E are in E', and E' also includes an edge (s, v) for every vertex $v \in V$ that has in-degree 0 in G. Thus, the only vertex with in-degree 0 in G' is the new source s. The weight of edge $(u, v) \in E'$ is the weight of vertex v in G. We have the weight of each entering edge in G' is the weight of the vertex it enters in G.

24.2-4

> Give an efficient algorithm to count the total number of paths in a directed acyclic graph. Analyze your algorithm.

We will compute the total number of paths by counting the number of paths whose start point is at each vertex v, which will be stored in an attribute $v.\text{paths}$. Assume that initial we have $v.\text{paths} = 0$ for all $v \in V$. Since all vertices adjacent to u occur later in the topological sort and the final vertex has no neighbors, line 4 is well-defined. Topological sort takes $O(V + E)$ and the nested for-loops take $O(V + E)$ so the total runtime is $O(V + E)$.

```
PATHS(G)
    topologically sort the vertices of G
    for each vertex u, taken in topologically sorted order
        for each v ∈ G.Adj[u]
            v.paths = u.paths + 1 + v.paths
    return the sum of all paths attributes
```

24.3 Dijkstra's algorithm

24.3-1

> Run Dijkstra's algorithm on the directed graph of Figure 24.2, first using vertex s as the source and then using vertex z as the source. In the style of Figure 24.6, show the d and π values and the vertices in set S after each iteration of the **while** loop.

- s as the source:
  - d values:

s	t	x	y	z
0	3	∞	5	∞
0	3	9	5	∞
0	3	9	5	11
0	3	9	5	11
0	3	9	5	11

  - π values:

s	t	x	y	z
NIL	s	NIL	NIL	NIL
NIL	s	t	s	NIL
NIL	s	t	s	y
NIL	s	t	s	y
NIL	s	t	s	y

- z as the source:
  - d values:

s	t	x	y	z
3	∞	7	∞	0
3	6	7	8	0
3	6	7	8	0
3	6	7	8	0
3	6	7	8	0

  - π values:

s	t	x	y	z
z	NIL	z	NIL	NIL
z	s	z	s	NIL
z	s	z	s	NIL
z	s	z	s	NIL
z	s	z	s	NIL

24.3-2

> Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesn't the proof of Theorem 24.6 go through when negative-weight edges are allowed?

Consider any graph with a negative cycle. RELAX is called a finite number of times but the distance to any vertex on the cycle is $-\infty$, so Dijkstra's algorithm cannot possibly be correct here. The proof of theorem 24.6 doesn't go through because we can no longer guarantee that

$$\delta(s, y) \leq \delta(s, u).$$

24.3-3

> Suppose we change line 4 of Dijkstra's algorithm to the following.
>
> ```
> 4  while |Q| > 1
> ```
>
> This change causes the **while** loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm correct?

Yes, the algorithm is correct. Let u be the leftover vertex that does not get extracted from the priority queue Q. If u is not reachable from s, then

$$u.d = \delta(s, u) = \infty.$$

If u is reachable from s, then there is a shortest path

$$p = s \to x \to u.$$

When the node x was extracted,

$$x.d = \delta(s, x)$$

and then the edge (x, u) was relaxed; thus,

$$u.d = \delta(s, u).$$

24.3-4

> Professor Gaedel has written a program that he claims implements Dijkstra's algorithm. The program produces $v.d$ and $v.\pi$ for each vertex $v \in V$. Give an $O(V + E)$-time algorithm to check the output of the professor's program. It should determine whether the d and π attributes match those of some shortest-paths tree. You may assume that all edge weights are nonnegative.

(Removed)

24.3-5

> Professor Newman thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex
>
> reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm could relax the edges of a shortest path out of order.

(Removed)

24.3-6

> We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

(Removed)

24.3-7

> Let $G = (V, E)$ be a weighted, directed graph with positive weight function $w : E \to \{1, 2, \ldots, W\}$ for some positive integer W, and assume that no two vertices have the same shortest-path weights from source vertex s. Now suppose that we define an unweighted, directed graph $G' = (V \cup V', E')$ by replacing each edge $(u, v) \in E$ with $w(u, v)$ unit-weight edges in series. How many vertices does G' have? Now suppose that we run a breadth-first search on G'. Show that the order in which the breadth-first search of G' colors vertices in V black is the same as the order in which Dijkstra's algorithm extracts the vertices of V from the priority queue when it runs on G.

$V + \sum_{(u,v) \in E} w(u, v) - E$.

24.3-8

> Let $G = (V, E)$ be a weighted, directed graph with nonnegative weight function $w : E \to \{0, 1, \ldots, W\}$ for some nonnegative integer W. Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

(Removed)

24.3-9

> Modify your algorithm from Exercise 24.3-8 to run in $O((V + E) \lg W)$ time. (Hint: How many distinct shortest-path estimates can there be in $V - S$ at any point in time?)

(Removed)

24.3-10

> Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

The proof of correctness, Theorem 24.6, goes through exactly as stated in the text. The key fact was that $\delta(s, y) \leq \delta(s, u)$. It is claimed that this holds because there are no negative edge weights, but in fact that is stronger than is needed. This always holds if y occurs on a shortest path from s to u and $y \neq s$ because all edges on the path from y to u have nonnegative weight. If any had negative weight, this would imply that we had "gone back" to an edge incident with s, which implies that a cycle is involved in the path, which would only be the case if it were a negative-weight cycle. However, these are still forbidden.

24.4 Difference constraints and shortest paths

24.4-1

> Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:
>
> $$\begin{aligned} x_1 - x_2 &\leq 1, \\ x_1 - x_4 &\leq -4, \\ x_2 - x_3 &\leq 2, \\ x_2 - x_5 &\leq 7, \\ x_2 - x_6 &\leq 5, \\ x_3 - x_6 &\leq 10, \\ x_4 - x_2 &\leq 2, \\ x_5 - x_1 &\leq -1, \\ x_5 - x_4 &\leq 3, \\ x_6 - x_3 &\leq 8 \end{aligned}$$

Our vertices of the constraint graph will be

$$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}.$$

The edges will be

$$(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_0, v_4), (v_0, v_5), (v_0, v_6), (v_2, v_1), (v_4, v_1), (v_3, v_2), (v_5, v_2), (v_6, v_2), (v_6, v_3),$$

with edge weights

$$0, 0, 0, 0, 0, 0, 1, -4, 2, 7, 5, 10, 2, -1, 3, -8$$

respectively. Then, computing

$$(\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \delta(v_0, v_4), \delta(v_0, v_5), \delta(v_0, v_6)),$$

we get

$$(-5, -3, 0, -1, -6, -8),$$

which is a feasible solution by Theorem 24.9.

24.4-2

> Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:
>
> $$\begin{aligned} x_1 - x_2 &\leq 4, \\ x_1 - x_5 &\leq 5, \\ x_2 - x_4 &\leq -6, \\ x_3 - x_2 &\leq 1, \\ x_4 - x_1 &\leq 3, \\ x_4 - x_3 &\leq 5, \\ x_4 - x_5 &\leq 10, \\ x_5 - x_3 &\leq -4, \\ x_5 - x_4 &\leq -8. \end{aligned}$$

There is no feasible solution because the constraint graph contains a negative-weight cycle: $(v_1, v_4, v_2, v_3, v_5, v_1)$ has weight -1.

24.4-3

> Can any shortest-path weight from the new vertex v_0 in a constraint graph be positive? Explain.

No, it cannot be positive. This is because for every vertex $v \neq v_0$, there is an edge (v_0, v) with weight zero. So, there is some path from the new vertex to every other of weight zero. Since $\delta(v_0, v)$ is a minimum weight of all paths, it cannot be greater than the weight of this weight zero path that consists of a single edge.

24.4-4

> Express the single-pair shortest-path problem as a linear program.

(Removed)

24.4-5

> Show how to modify the Bellman-Ford algorithm slightly so that when we use it to solve a system of difference constraints with m inequalities on n unknowns, the running time is $O(nm)$.

We can follow the advice of problem 14.4-7 and solve the system of constraints on a modified constraint graph in which there is no new vertex v_0. This is simply done by initializing all of the vertices to have a d value of 0 before running the iterated relaxations of Bellman Ford. Since we don't add a new vertex and the n edges going from it to to vertex corresponding to each variable, we are just running Bellman Ford on a graph with n vertices and m edges, and so it will have a runtime of $O(mn)$.

24.4-6

> Suppose that in addition to a system of difference constraints, we want to handle ***equality constraints*** of the form $x_i = x_j + b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of

> constraint system.

To obtain the equality constraint $x_i = x_j + b_k$ we simply use the inequalities $x_i - x_j \le b_k$ and $x_j - x_i \le -bk$, then solve the problem as usual.

24.4-7

> Show how to solve a system of difference constraints by a Bellman–Ford-like algorithm that runs on a constraint graph without the extra vertex v_0.

(Removed)

24.4-8 ⋆

> Let $Ax \le b$ be a system of m difference constraints in n unknowns. Show that the Bellman-Ford algorithm, when run on the corresponding constraint graph, maximizes $\sum_{i=1}^{n} x_i$ subject to $Ax \le b$ and $x_i \le 0$ for all x_i.

Bellman–Ford correctly solves the system of difference constraints so $Ax \le b$ is always satisfied. We also have that $x_i = \delta(v_0, v_i) \le w(v_0, v_i) = 0$ so $x_i \le 0$ for all i. To show that $\sum x_i$ is maximized, we'll show that for any feasible solution $(y_1, y_2, \dots, y_n)$ which satisfies the constraints we have $y_i \le \delta(v_0, v_i) = x_i$. Let $v_0, v_{i_1}, \dots, v_{i_k}$ be a shortest path from v_0 to v_i in the constraint graph. Then we must have the constraints $y_{i_2} - y_{i_1} \le w(v_{i_1}, v_{i_2}), \dots, y_{i_k} - y_{i_{k-1}} \le w(v_{i_{k-1}}, v_{i_k})$. Summing these up we have

$$y_i \le y_i - y_1 \le \sum_{m=2}^{k} w(v_{i_m}, v_{i_{m-1}}) = \delta(v_0, v_i) = x_i.$$

24.4-9 ⋆

> Show that the Bellman-Ford algorithm, when run on the constraint graph for a system $Ax \le b$ of difference constraints, minimizes the quantity $(\max\{x_i\} - \min\{x_i\})$ subject to $Ax \le b$. Explain how this fact might come in handy if the algorithm is used to schedule construction jobs.

We can see that the Bellman-Ford algorithm run on the graph whose construction is described in this section causes the quantity $\max\{x_i\} - \min\{x_i\}$ to be minimized. We know that the largest value assigned to any of the vertices in the constraint graph is a 0. **It** is clear that it won't be greater than zero, since just the single edge path to each of the vertices has cost zero. We also know that we cannot have every vertex having a shortest path with negative weight. To see this, notice that this would mean that the pointer for each vertex has it's p value going to some other vertex that is not the source. This means that if we follow the procedure for reconstructing the shortest path for any of the vertices, we have that it can never get back to the source, a contradiction to the fact that it is a shortest path from the source to that vertex.

Next, we note that when we run Bellman-Ford, we are maximizing $\min\{x_i\}$. The shortest distance in the constraint graphs is the bare minimum of what is required in order to have all the constraints satisfied, if we were to increase any of the values we would be violating a constraint.

This could be in handy when scheduling construction jobs because the quantity $\max\{x_i\} - \min\{x_i\}$ is equal to the difference in time between the last task and the first task. Therefore, it means that minimizing it would mean that the total time that all the jobs takes is also minimized. And, most people want the entire process of construction to take as short of a time as possible.

24.4-10

> Suppose that every row in the matrix A of a linear program $Ax \le b$ corresponds to a difference constraint, a single-variable constraint of the form $x_i \le b_k$, or a singlevariable constraint of the form $-x_i \le b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

(Removed)

24.4-11

> Give an efficient algorithm to solve a system $Ax \le b$ of difference constraints when all of the elements of b are real-valued and all of the unknowns x_i must be integers.

To do this, just take the floor of (largest integer that is less than or equal to) each of the b values and solve the resulting integer difference problem. These modified constraints will be admitting exactly the same set of assignments since we required that the solution have integer values assigned to the variables. This is because since the variables are integers, all of their differences will also be integers. For an integer to be less than or equal to a real number, it is necessary and sufficient for it to be less than or equal to the floor of that real number.

24.4-12 ⋆

> Give an efficient algorithm to solve a system $Ax \le b$ of difference constraints when all of the elements of b are real-valued and a specified subset of some, but not necessarily all, of the unknowns x_i must be integers.

To solve the problem of $Ax \le b$ where the elements of b are real-valued we carry out the same procedure as before, running Bellman–Ford, but allowing our edge weights to be real-valued. To impose the integer condition on the x_i's, we modify the RELAX procedure. Suppose we call $\text{RELAX}(v_i, v_j, w)$ where v_j is required to be integral valued. **If** $v_j.d > \lfloor v_i.d + w(v_i, v_j) \rfloor$, set $v_j.d = \lfloor v_i.d + w(v_i, v_j) \rfloor$. This guarantees that the condition that $v_j.d - v_i.d \le w(v_i, v_j)$ as desired. **It** also ensures that v_j is integer valued. Since the triangle inequality still holds, $x = (v_1.d, v_2.d, \dots, v_n.d)$ is a feasible solution for the system, provided that G contains no negative weight cycles.

24.5 Proofs of shortest-paths properties

24.5-1

> Give two shortest-paths trees for the directed graph of Figure 24.2 (on page 648) other than the two shown.

Since the induced shortest path trees on $\{s, t, y\}$ and on $\{t, x, y, z\}$ are independent and have to possible configurations each, there are four total arising from that. So, we have the two not shown in the figure are the one consisting of the edges $\{(s, t), (s, y), (y, x), (x, z)\}$ and the one consisting of the edges $\{(s, t), (t, y), (t, x), (y, z)\}$.

24.5-2

> Give an example of a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ and source vertex s such that G satisfies the following property: For every edge $(u, v) \in E$, there is a shortest-paths tree rooted at s that contains (u, v) and another shortest-paths tree rooted at s that does not contain (u, v).

Let G have 3 vertices s, x, and y. Let the edges be (s, x), (s, y), and (x, y) with weights 1, 1, and 0 respectively. There are 3 possible trees on these vertices rooted at s, and each is a shortest paths tree which gives $\delta(s, x) = \delta(s, y) = 1$.

24.5-3

> Embellish the proof of Lemma 24.10 to handle cases in which shortest-path weights are ∞ or $-\infty$.

To modify Lemma 24.10 to allow for possible shortest path weights of ∞ and $-\infty$, we need to define our addition as $\infty + c = \infty$, and $-\infty + c = -\infty$. This will make the statement behave correctly, that is, we can take the shortest path from s to u and tack on the edge (u, v) to the end. That is, if there is a negative weight cycle on your way to u and there is an edge from u to v, there is a negative weight cycle on our way to v. Similarly, if we cannot reach v and there is an edge from u to v, we cannot reach u.

24.5-4

> Let $G = (V, E)$ be a weighted, directed graph with source vertex s, and let G be initialized by $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$. Prove that if a sequence of relaxation steps sets $s.\pi$ to a non-NIL value, then G contains a negative-weight cycle.

(Removed)

24.5-5

> Let $G = (V, E)$ be a weighted, directed graph with no negative-weight edges. Let $s \in V$ be the source vertex, and suppose that we allow $v.\pi$ to be the predecessor of v on *any* shortest path to v from source s if $v \in V - \{s\}$ is reachable from s, and NIL otherwise. Give an example of such a graph G and an assignment of π values that produces a cycle in G_π. (By Lemma 24.16, such an assignment cannot be produced by a sequence of relaxation steps.)

Suppose that we have a grap hon three vertices $\{s, u, v\}$ and containing edges (s, u), (s, v), (u, v), (v, u) all with weight 0. Then, there is a shortest path from s to v of s, u, v and a shortest path from s to u of s v, u. Based off of these, we could set $v.\pi = u$ and $u.\pi = v$. This then means that there is a cycle consisting of u, v in G_π.

24.5-6

> Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$ and no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$. Prove that for every vertex $v \in V_\pi$, there exists a path from s to v in G_π and that this property is maintained as an invariant over any sequence of relaxations.

We will prove this by induction on the number of relaxations performed. For the base-case, we have just called $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$. The only vertex in V_π is s, and there is trivially a path from s to itself. Now suppose that after any sequence of n relaxations, for every vertex $v \in V_\pi$ there exists a path from s to v in G_π. Consider the $(n+1)$th relaxation. Suppose it is such that $v.d > u.d + w(u, v)$. When we relax v, we update $v.\pi = u.\pi$. By the induction hypothesis, there was a path from s to u in G_π. Now v is in V_π, and the path from s to u, followed by the edge $(u, v) = (v.\pi, v)$ is a path from s to v in G_π, so the claim holds.

24.5-7

> Let $G = (V, E)$ be a weighted, directed graph that contains no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$. Prove that there exists a sequence of $|V| - 1$ relaxation steps that produces $v.d = \delta(s, v)$ for all $v \in V$.

(Removed)

24.5-8

> Let G be an arbitrary weighted, directed graph with a negative-weight cycle reachable from the source vertex s. Show how to construct an infinite sequence of relaxations of the edges of G such that every relaxation causes a shortest-path estimate to change.

(Removed)

Problem 24-1 Yen's improvement to Bellman-Ford

> Suppose that we order the edge relaxations in each pass of the Bellman-Ford algorithm as follows. Before the first pass, we assign an arbitrary linear order $v_1, v_2, \dots, v_{|V|}$ to the vertices of the input graph $G = (V, E)$. Then, we partition the edge set E into $E_f \cup E_b$, where $E_f = \{(v_i, v_j) \in E : i < j\}$ and $E_b = \{(v_i, v_j) \in E : i > j\}$. (Assume that G contains no self-loops, so that every edge is in either E_f or E_b.) Define $G_f = (V, E_f)$ and $G_b = (V, E_b)$.
>
> **a.** Prove that G_f is acyclic with topological sort $\langle v_1, v_2, \dots, v_{|V|} \rangle$ and that G_b is acyclic with topological sort $\langle v_{|V|}, v_{|V|-1}, \dots, v_1 \rangle$.
>
> Suppose that we implement each pass of the Bellman–Ford algorithm in the following way. We visit each vertex in the order $v_1, v_2, \dots, v_{|V|}$, relaxing edges of E_f that leave the vertex. We then visit each vertex in the order $v_{|V|}, v_{|V|-1}, \dots, v_1$, relaxing edges of E_b that leave the vertex.
>
> **b.** Prove that with this scheme, if G contains no negative-weight cycles that are reachable from the source vertex s, then after only $\lceil |V|/2 \rceil$ passes over the edges, $v.d = \delta(s, v)$ for all vertices $v \in V$.
>
> **c.** Does this scheme improve the asymptotic running time of the Bellman-Ford algorithm?

a. Since in G_f edges only go from vertices with smaller index to vertices with greater index, there is no way that we could pick a vertex, and keep increasing it's index, and get back to having the index equal to what we started with. This means that G_f is acyclic. Similarly, there is no way to pick an index, keep decreasing it, and get back to the same vertex index. By these definitions, since G_f only has vertices going from lower indices to higher indices, $(v_1, \dots, v_{|V|})$ is a topological ordering of the vertices. Similarly, for G_b, $(v_{|V|}, \dots, v_1)$ is a topological ordering of the vertices.

b. Suppose that we are trying to find the shortest path from s to v. Then, list out the vertices of this shortest path $v_{k_1}, v_{k_2}, \dots, v_{k_m}$. Then, we have that the number of times that the sequence $\{k_i\}_i$ goes from increasing to decreasing or from decreasing to increasing is the number of passes over the edges that are necessary to notice this path. This is because any increasing sequence of vertices will be captured in a pass through E_f and any decreasing sequence will be captured in a pass through E_b. Any sequence of integers of length $|V|$ can only change direction at most $\lfloor |V|/2 \rfloor$ times. However, we need to add one more in to account for the case that the source appears later in the ordering of the vertices than v_{k_2}, as it is in a sense initially expecting increasing vertex indices, as it runs through E_f before E_b.

c. **It** does not improve the asymptotic runtime of Bellman ford, it just drops the runtime from having a leading coefficient of 1 to a leading coefficient of $\frac{1}{2}$. Both in the original and in the modified version, the runtime is $O(EV)$.

Problem 24-2 Nesting boxes

> A d-dimensional box with dimensions $(x_1, x_2, \dots, x_d)$ ***nests*** within another box with dimensions $(y_1, y_2, \dots, y_d)$ if there exists a permutation π on $\{1, 2, \dots, d\}$ such that $x_{\pi(1)} < y_1$, $x_{\pi(2)} < y_2$, …, $x_{\pi(d)} < y_d$.
>
> **a.** Argue that the nesting relation is transitive.
>
> **b.** Describe an efficient method to determine whether or not one d-dimensional box nests inside another.
>
> **c.** Suppose that you are given a set of n d-dimensional boxes $\{B_1, B_2, \dots, B_n\}$. Give an efficient algorithm to find the longest sequence $\langle B_{i_1}, B_{i_2}, \dots, B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \dots, k-1$. Express the running time of your algorithm in terms of n and d.

a. Suppose that box $x = (x_1, \dots, x_d)$ nests with box $y = (y_1, \dots, y_d)$ and box y nests with box $z = (z_1, \dots, z_d)$. Then there exist permutations π and σ such that $x_{\pi(1)} < y_1, \dots, x_{\pi(d)} < y_d$ and $y_{\sigma(1)} < z_1, \dots, y_{\sigma(d)} < z_d$. This implies $x_{\pi(\sigma(1))} < z_1, \dots, x_{\pi(\sigma(d))} < z_d$, so x nests with z and the nesting relation is transitive.

b. Box x nests inside box y if and only if the increasing sequence of dimensions of x is component-wise strictly less than the increasing sequence of dimensions of y. Thus, it will suffice to sort both sequences of dimensions and compare them. Sorting both length d sequences is done in $O(d \lg d)$, and comparing their elements is done in $O(d)$, so the total time is $O(d \lg d)$.

c. We will create a nesting-graph G with vertices $B_1, \dots, B_n$ as follows. For each pair of boxes B_i, B_j, we decide if one nests inside the other. **If** B_i nests in B_j, draw an arrow from B_i to B_j. **If** B_j nests in B_i, draw an arrow from B_j to B_i. **If** neither nests, draw no arrow. To determine the arrows efficiently, after sorting each list of dimensions in $O(nd \lg d)$ we compair all pairs of boxes using the algorithm from part (b) in $O(n^2 d)$. By part (a), the resulted graph is acyclic, which allows us to easily find the longest chain in it in $O(n^2)$ in a bottom-up manner. This chain is our answer. Thus, the total time is $O(nd \max(\lg d, n))$.

Problem 24-3 Arbitrage

> ***Arbitrage*** is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 \times 2 \times 0.0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent.
>
> Suppose that we are given n currencies $c_1, c_2, \dots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.
>
> **a.** Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \dots, c_{i_k} \rangle$ such that
>
> $$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$
>
> Analyze the running time of your algorithm.
>
> **b.** Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

a. To do this we take the negative of the natural log (or any other base will also work) of all the values c_i that are on the edges between the currencies. Then, we detect the presence or absence of a negative weight cycle by applying Bellman Ford. To see that the existence of an arbitrage situation is equivalent to there being a negative weight cycle in the original graph, consider the following sequence of steps:

$$\begin{aligned} R[i_1, i_2] \cdot R[i_2, i_3] \cdot \cdots \cdot R[i_k, i_1] &> 1 \\ \ln(R[i_1, i_2]) + \ln(R[i_2, i_3]) + \cdots + \ln(R[i_k, i_1]) &> 0 \\ -\ln(R[i_1, i_2]) - \ln(R[i_2, i_3]) - \cdots - \ln(R[i_k, i_1]) &< 0. \end{aligned}$$

b. To do this, we first perform the same modification of all the edge weights as done in part (a) of this problem. Then, we wish to detect the negative weight cycle. To do this, we relax all the edges $|V| - 1$ many times, as in BellmanFord algorithm. Then, we record all of the d values of the vertices. Then, we relax all the edges $|V|$ more times. Then, we check to see which vertices had their d value decrease since we recorded them. All of

these vertices must lie on some (possibly disjoint) set of negative weight cycles. Call S this set of vertices. To find one of these cycles in particular, we can pick any vertex in S and greedily keep picking any vertex that it has an edge to that is also in S. Then, we just keep an eye out for a repeat. This finds us our cycle. We know that we will never get to a dead end in this process because the set S consists of vertices that are in some union of cycles, and so every vertex has out degree at least 1.

Problem 24-4 Gabow's scaling algorithm for single-source shortest paths

A ***scaling*** algorithm solves a problem by initially considering only the highestorder bit of each relevant input value (such as an edge weight). It then refines the initial solution by looking at the two highest-order bits. It progressively looks at more and more high-order bits, refining the solution each time, until it has examined all bits and computed the correct solution.

In this problem, we examine an algorithm for computing the shortest paths from a single source by scaling edge weights. We are given a directed graph $G = (V, E)$ with nonnegative integer edge weights w. Let $W = \max_{(u,v)\in E}\{w(u, v)\}$. Our goal is to develop an algorithm that runs in $O(E \lg W)$ time. We assume that all vertices are reachable from the source.

The algorithm uncovers the bits in the binary representation of the edge weights one at a time, from the most significant bit to the least significant bit. Specifically, let $k = \lceil \lg(W + 1) \rceil$ be the number of bits in the binary representation of W, and for $i = 1, 2, \ldots, k$, let $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$. That is, $w_i(u, v)$ is the "scaled-down" version of $w(u, v)$ given by the i most significant bits of $w(u, v)$. (Thus, $w_k(u, v) = w(u, v)$ for all $(u, v) \in E$.) For example, if $k = 5$ and $w(u, v) = 25$, which has the binary representation $\langle 11001 \rangle$, then $w_3(u, v) = \langle 110 \rangle = 6$. As another example with $k = 5$, if $w(u, v) = \langle 00100 \rangle = 4$, then $w_3(u, v) = \langle 001 \rangle = 1$. Let us define $\delta_i(u, v)$ as the shortest-path weight from vertex u to vertex v using weight function w_i. Thus, $\delta_k(u, v) = \delta(u, v)$ for all $u, v \in V$. For a given source vertex s, the scaling algorithm first computes the shortest-path weights $\delta_1(s, v)$ for all $v \in V$, then computes $\delta_2(s, v)$ for all $v \in V$, and so on, until it computes $\delta_k(s, v)$ for all $v \in V$. We assume throughout that $|E| \geq |V| - 1$, and we shall see that computing δ_i from δ_{i-1} takes $O(E)$ time, so that the entire algorithm takes $O(kE) = O(E \lg W)$ time.

a. Suppose that for all vertices $v \in V$, we have $\delta(s, v) \leq |E|$. Show that we can compute $\delta(s, v)$ for all $v \in V$ in $O(E)$ time.

b. Show that we can compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time.

Let us now focus on computing δ_i from δ_{i-1}.

c. Prove that for $i = 2, 3, \ldots, k$, we have either $w_i(u, v) = 2w_{i-1}(u, v)$ or $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Then, prove that

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$$

for all $v \in V$.

d. Define for $i = 2, 3, \ldots, k$ and all $(u, v) \in E$,

$$\hat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v).$$

Prove that for $i = 2, 3, \ldots, k$ and all $u, v \in V$, the "reweighted" value $\hat{w}_i(u, v)$ of edge (u, v) is a nonnegative integer.

e. Now, define $\hat{\delta}_i(s, v)$ as the shortest-path weight from s to v using the weight function $\hat{w}_i$. Prove that for $i = 2, 3, \ldots, k$ and all $v \in V$,

$$\delta_i(s, v) = \hat{\delta}_i(s, v) + 2\delta_{i-1}(s, v)$$

and that $\hat{\delta}_i(s, v) \leq |E|$.

f. Show how to compute $\delta_i(s, v)$ from $\delta_{i-1}(s, v)$ for all $v \in V$ in $O(E)$ time, and conclude that we can compute $\delta(s, v)$ for all $v \in V$ in $O(E \lg W)$ time.

a. We can do this in $O(E)$ by the algorithm described in exercise 24.3-8 since our "priority queue" takes on only integer values and is bounded in size by E.

b. We can do this in $O(E)$ by the algorithm described in exercise 24.3-8 since w takes values in $\{0, 1\}$ and $V = O(E)$.

c. If the ith digit, read from left to right, of $w(u, v)$ is 0, then $w_i(u, v) = 2w_{i-1}(u, v)$. If it is a 1, then $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Now let $s = v_0, v_1, \ldots, v_n = v$ be a shortest path from s to v under w_i. Note that any shortest path under w_i is necessarily also a shortest path under w_{i-1}. Then we have

$$\begin{aligned}\delta_i(s, v) &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) \\ &\leq \sum_{m=1}^{n} [2w_{i-1}(u, v) + 1] \\ &\leq \sum_{m=1}^{n} w_{i-1}(u, v) + n \\ &\leq 2\delta_{i-1}(s, v) + |V| - 1.\end{aligned}$$

On the other hand, we also have

$$\begin{aligned}\delta_i(s, v) &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) \\ &\geq \sum_{m=1}^{n} 2w_{i-1}(v_{m-1}, v_m) \\ &\geq 2\delta_{i-1}(s, v).\end{aligned}$$

d. Note that every quantity in the definition of $\hat{w}_i$ is an integer, so $\hat{w}_i$ is clearly an integer. Since $w_i(u, v) \geq 2w_{i-1}(u, v)$, it will suffice to show that $w_{i-1}(u, v) + \delta_{i-1}(s, u) \geq \delta_{i-1}(s, v)$ to prove nonnegativity. This follows immediately from the triangle inequality.

e. First note that $s = v_0, v_1, \ldots, v_n = v$ is a shortest path from s to v with respect to $\hat{w}$ if and only if it is a shortest path with respect to w. Then we have

$$\begin{aligned}\hat{\delta}_i(s, v) &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) + 2\delta_{i-1}(s, v_{m-1}) - 2\delta_{i-1}(s, v_m) \\ &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) - 2\delta_{i-1}(s, v_n) \\ &= \delta_i(s, v) - 2\delta_{i-1}(s, v).\end{aligned}$$

f. By part (a) we can compute $\hat{\delta}_i(s, v)$ for all $v \in V$ in $O(E)$ time. If we have already computed $\delta_i - 1$ then we can compute δ_i in $O(E)$ time. Since we can compute δ_1 in $O(E)$ by part b, we can compute δ_i from scratch in $O(iE)$ time. Thus, we can compute $\delta = \delta_k$ in $O(Ek) = O(E \lg W)$ time.

Problem 24-5 Karp's minimum mean-weight cycle algorithm

Let $G = (V, E)$ be a directed graph with weight function $w : E \to \mathbb{R}$, and let $n = |V|$. We define the ***mean weight*** of a cycle $c = \langle e_1, e_2, \ldots, e_k \rangle$ of edges in E to be

$$\mu(c) = \frac{1}{k}\sum_{i=1}^{k} w(e_i).$$

Let $\mu^* = \min_c \mu(c)$, where c ranges over all directed cycles in G. We call a cycle c for which $\mu(c) = \mu^*$ a ***minimum mean-weight cycle***. This problem investigates an efficient algorithm for computing μ^*.

Assume without loss of generality that every vertex $v \in V$ is reachable from a source vertex $s \in V$. Let $\delta(s, v)$ be the weight of a shortest path from s to v, and let $\delta_k(s, v)$ be the weight of a shortest path from s to v consisting of *exactly* k edges. If there is no path from s to v with exactly k edges, then $\delta_k(s, v) = \infty$.

a. Show that if $\mu^* = 0$, then G contains no negative-weight cycles and $\delta(s, v) = \min_{0\leq k\leq n-1} \delta_k(s, v)$ for all vertices $v \in V$.

b. Show that if $\mu^* = 0$, then

$$\max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0$$

for all vertices $v \in V$. (Hint: Use both properties from part (a).)

c. Let c be a 0-weight cycle, and let u and v be any two vertices on c. Suppose that $\mu^* = 0$ and that the weight of the simple path from u to v along the cycle is x. Prove that $\delta(s, v) = \delta(s, u) + x$. (Hint: The weight of the simple path from v to u along the cycle is $-x$.)

d. Show that if $\mu^* = 0$, then on each minimum mean-weight cycle there exists a vertex v such that

$$\max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

(Hint: Show how to extend a shortest path to any vertex on a minimum meanweight cycle along the cycle to make a shortest path to the next vertex on the cycle.)

e. Show that if $\mu^* = 0$, then

$$\min_{v\in V} \max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

f. Show that if we add a constant t to the weight of each edge of G, then μ^* increases by t. Use this fact to show that

$$\mu^* = \min_{v\in V} \max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}.$$

g. Give an $O(VE)$-time algorithm to compute μ^*.

a. If $\mu^* = 0$, then we have that the lowest that $\frac{1}{k}\sum_{i=1}^{k} w(e_i)$ can be zero. This means that the lowest $\sum_{i=1}^{k} w(e_i)$ can be 0. This means that no cycle can have negative weight. Also, we know that for any path from s to v, we can make it simple by removing any cycles that occur. This means that it had a weight equal to some path that has at most $n - 1$ edges in it. Since we take the minimum over all possible number of edges, we have the minimum over all paths.

b. To show that

$$\max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0,$$

we need to show that

$$\max_{0\leq k\leq n-1} \delta_n(s, v) - \delta_k(s, v) \geq 0.$$

Since we have that $\mu^* = 0$, there aren't any negative weight cycles. This means that we can't have the minimum cost of a path decrease as we increase the possible length of the path past $n - 1$. This means that there will be a path that at least ties for cheapest when we restrict to the path being less than length n. Note that there may also be cheapest path of longer length since we necessarily do have zero cost cycles. However, this isn't guaranteed since the zero cost cycle may not lie along a cheapest path from s to v.

c. Since the total cost of the cycle is 0, and one part of it has cost x, in order to balance that out, the weight of the rest of the cycle has to be $-x$. So, suppose we have some shortest length path from s to u, then, we could traverse the path from u to v along the cycle to get a path from s to u that has length $\delta(s, u) + x$. This gets us that $\delta(s, v) \leq \delta(s, u) + x$.

To see the converse inequality, suppose that we have some shortest length path from s to v. Then, we can traverse the cycle going from v to u. We already said that this part of the cycle had total cost $-x$. This gets us that $\delta(s, u) \leq \delta(s, v) - x$. Or, rearranging, we have $\delta(s, u) + x \leq \delta(s, v)$. Since we have inequalities both ways, we must have equality.

d. To see this, we find a vertex v and natural number $k \leq n - 1$ so that $\delta_n(s, v) - \delta_k(s, v) = 0$. To do this, we will first take any shortest length, smallest number of edges path from s to any vertex on the cycle. Then, we will just keep on walking around the cycle until we've walked along n edges. Whatever vertex we end up on at that point will be our v. Since we did not change the d value of v after looking at length n paths, by part (a), we know that there was some length of this path, say k, which had the same cost. That is, we have $\delta_n(s, v) = \delta_k(s, v)$.

e. This is an immediate result of the previous problem and part (b). Part (a) says that the inequality holds for all v, so, we have

$$\min_{v\in V} \max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta(s, v)}{n - k} \geq 0.$$

The previous part says that there is some v on each minimum weight cycle so that

$$\max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta(s, v)}{n - k} = 0,$$

which means that

$$\min_{v\in V} \max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \leq 0.$$

Putting the two inequalities together, we have the desired equality.

f. If we add t to the weight of each edge, the mean weight of any cycle becomes

$$\mu(c) = \frac{1}{k}\sum_{i=1}^{k}(w(e_i) + t) = \frac{1}{k}\Big(\sum_{i}^{k} w(e_i)\Big) + \frac{kt}{k} = \frac{1}{k}\Big(\sum_{i}^{k} w(e_i)\Big) + t.$$

This is the original, unmodified mean weight cycle, plus t. Since this is how the mean weight of every cycle is changed, the lowest mean weight cycle stays the lowest mean weight cycle. This means that μ^* will increase by t. Suppose that we first compute μ^*. Then, we subtract from every edge weight the value μ^*. This will make the new μ^* equal zero, which by part (e) means that

$$\min_{v\in V} \max_{0\leq k\leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

Since they are both equal to zero, they are both equal to each other.

g. By the previous part, it suffices to compute the expression on the previous line. We will start by creating a table that lists $\delta_k(s, v)$ for every $k \in \{1, \ldots, n\}$ and $v \in V$. This can be done in time $O(V(E + V))$ by creating a $|V|$ by $|V|$ table, where the kth row and vth column represent $\delta_k(s, v)$ when wanting to compute a particular entry, we need look at a number of entries in the previous row equal to the in degree of the vertex we want to compute.

So, summing over the computation required for each row, we need $O(E + V)$. Note that this total runtime can be bumped down to $O(VE)$ by not including in the table any isolated vertices, this will ensure that $E \in \Omega(V)$. So, $O(V(E + V))$ becomes $O(VE)$. Once we have this table of values computed, it is simple to just replace each row with the last row minus what it was, and divide each entry by $n - k$, then, find the min column in each row, and take the max of those numbers.

Problem 24-6 Bitonic shortest paths

A sequence is ***bitonic*** if it monotonically increases and then monotonically decreases, or if by a circular shift it monotonically increases and then monotonically decreases. For example the sequences $\langle 1, 4, 6, 8, 3, -2 \rangle$, $\langle 9, 2, -4, -10, -5 \rangle$, and $\langle 1, 2, 3, 4 \rangle$ are bitonic, but $\langle 1, 3, 12, 4, 2, 10 \rangle$ is not bitonic. (See Problem 15-3 for the bitonic euclidean traveling-salesman problem.)

Suppose that we are given a directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, where all edge weights are unique, and we wish to find single-source shortest paths from a source vertex s. We are given one additional piece of information: for each vertex $v \in V$, the weights of the edges along any shortest path from s to v form a bitonic sequence.

Give the most efficient algorithm you can to solve this problem, and analyze its running time.

We'll use the Bellman-Ford algorithm, but with a careful choice of the order in which we relax the edges in order to perform a smaller number of RELAX operations. In any bitonic path there can be at most two distinct increasing sequences of edge weights, and similarly at most two distinct decreasing sequences of edge weights. Thus, by the path-relaxation property, if we relax the edges in order of increasing weight then decreasing weight twice (for a total of four times relaxing every edge) the we are guaranteed that $v.d$ will equal $\delta(s, v)$ for all $v \in V$. Sorting the edges takes $O(E \lg E)$. We relax every edge 4 times, taking $O(E)$. Thus the total runtime is $O(E \lg E) + O(E) = O(E \lg E)$, which is asymptotically faster than the usual $O(VE)$ runtime of Bellman-Ford.

25 All-Pairs Shortest Paths

25.1 Shortest paths and matrix multiplication

25.1-1

> Run SLOW-ALL-PAIRS-SHORTEST-PATHS on the weighted, directed graph of Figure 25.2, showing the matrices that result for each iteration of the loop. Then do the same for FASTER-ALL-PAIRS-SHORTEST-PATHS.

- Initial:

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

- Slow:

$m = 2$:

$$\begin{pmatrix} 0 & 6 & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 3 & -3 & 0 & 4 & \infty & -8 \\ -4 & 10 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & \infty & 0 \end{pmatrix}$$

$m = 3$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -2 & -3 & 0 & -1 & 2 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$m = 4$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -3 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$m = 5$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

- Fast:

$m = 2$:

$$\begin{pmatrix} 0 & 6 & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 3 & -3 & 0 & 4 & \infty & -8 \\ -4 & 10 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & \infty & 0 \end{pmatrix}$$

$m = 4$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -3 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$m = 8$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

25.1-2

> Why do we require that $w_{ii} = 0$ for all $1 \le i \le n$?

This is consistent with the fact that the shortest path from a vertex to itself is the empty path of weight 0. If there were another path of weight less than 0 then it must be a negative-weight cycle, since it starts and ends at v_i.

If $w_{ii} \neq 0$, then $L^{(1)}$ produced after the first run of EXTEND-SHORTEST-PATHS would not contain the minimum weight of any path from i to its neighbours. If $w_{ii} = 0$, then in line 7 of EXTEND-SHORTEST-PATHS, the second argument to $\min$ would not equal the weight of the edge going from i to its neighbours.

25.1-3

> What does the matrix
>
> $$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$
>
> used in the shortest-paths algorithms correspond to in regular matrix multiplication?

The identity matrix.

25.1-4

> Show that matrix multiplication defined by EXTEND-SHORTEST-PATHS is associative.

To verify associativity, we need to check that $(W^i W^j) W^p = W^i (W^j W^p)$ for all i, j and p, where we use the matrix multiplication defined by the EXTEND-SHORTEST-PATHS procedure. Consider entry (a, b) of the left hand side. This is:

$$\begin{aligned} \min_{1 \le k \le n} [W^i W^j]_{a,k} + W^p_{k,b} &= \min_{1 \le k \le n} \min_{1 \le q \le n} W^i_{a,q} + W^j_{q,k} + W^p_{k,b} \\ &= \min_{1 \le q \le n} W^i_{a,q} + \min_{1 \le k \le n} W^j_{q,k} + W^p_{k,b} \\ &= \min_{1 \le q \le n} W^i_{a,q} + [W^j W^p]_{q,b}, \end{aligned}$$

which is precisely entry (a, b) of the right hand side.

25.1-5

> Show how to express the single-source shortest-paths problem as a product of matrices and a vector. Describe how evaluating this product corresponds to a Bellman-Ford-like algorithm (see Section 24.1).

(Removed)

25.1-6

> Suppose we also wish to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix Π from the completed matrix L of shortest-path weights in $O(n^3)$ time.

For each source vertex v_i we need to compute the shortest-paths tree for v_i. To do this, we need to compute the predecessor for each $j \neq i$. For fixed i and j, this is the value of k such that $L_{i,k} + w(k, j) = L[i, j]$. Since there are n vertices whose trees need computing, n vertices for each such tree whose predecessors need computing, and it takes $O(n)$ to compute this for each one (checking each possible k), the total time is $O(n^3)$.

25.1-7

> We can also compute the vertices on shortest paths as we compute the shortestpath weights. Define $\pi^{(m)}_{ij}$ as the predecessor of vertex j on any minimum-weight path from i to j that contains at most m edges. Modify the EXTEND-SHORTESTPATHS and SLOW-ALL-PAIRS-SHORTEST-PATHS procedures to compute the matrices $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ as the matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ are computed.

To have the procedure compute the predecessor along the shortest path, see the modified procedures, EXTEND-SHORTEST-PATH-MOD and SLOW-ALL-PAIRS-SHORTEST-PATHS-MOD

```
EXTEND-SHORTEST-PATH-MOD(Π, L, W)
    n = L.row
    let L' = l'[i, j] be a new n × n matirx
    Π' = π'[i, j] is a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            l'[i, j] = ∞
            π'[i, j] = NIL
            for k = 1 to n
                if l[i, k] + l[k, j] < l[i, j]
                    l[i, j] = l[i, k] + l[k, j]
                    if k != j
                        π'[i, j] = k
                    else
                        π'[i, j] = π[i, j]
    return (Π', L')
```

```
SLOW-ALL-PAIRS-SHORTEST-PATHS-MOD(W)
    n = W.rows
    L(1) = W
    Π(1) = π[i, j](1) where π[i, j](1) = i if there is an edge from i to j,
    and NIL otherwise
    for m = 2 to n - 1
        Π(m), L(m) = EXTEND-SHORTEST-PATH-MOD(Π(m - 1), L(m - 1), W)
    return (Π(n - 1), L(n - 1))
```

25.1-8

> The FASTER-ALL-PAIRS-SHORTEST-PATHS procedure, as written, requires us to store $\lceil \lg(n-1) \rceil$ matrices, each with n^2 elements, for a total space requirement of $\Theta(n^2 \lg n)$. Modify the procedure to require only $\Theta(n^2)$ space by using only two $n \times n$ matrices.

We can overwrite matrices as we go. Let $A \cdot B$ denote multiplication defined by the EXTEND-SHORTEST-PATHS procedure. Then we modify FASTER-ALL-EXTEND-SHORTEST-PATHS(W). We initially create an n by n matrix L. Delete line 5 of the algorithm, and change line 6 to $L = W \cdot W$, followed by $W = L$.

25.1-9

> Modify FASTER-ALL-PAIRS-SHORTEST-PATHS so that it can determine whether the graph contains a negative-weight cycle.

For the modification, keep computing for one step more than the original, that is, we compute all the way up to $L^{(2k+1)}$ where $2^k > n - 1$. Then, if there aren't any negative weight cycles, then, we will have that the two matrices should be equal since having no negative weight cycles means that between any two vertices, there is a path that is tied for shortest and contains at most $n - 1$ edges.

However, if there is a cycle of negative total weight, we know that it's length is at most n, so, since we are allowing paths to be larger by $2k \ge n$ between these two matrices, we have that we would need to have all of the vertices on the cycle have their distance reduce by at least the negative weight of the cycle. Since we can detect exactly when there is a negative cycle, based on when these two matrices are different. This algorithm works. It also only takes time equal to a single matrix multiplication which is littlee oh of the unmodified algorithm.

25.1-10

> Give an efficient algorithm to find the length (number of edges) of a minimum-length negative-weight cycle in a graph.

(Removed)

25.2 The Floyd-Warshall algorithm

25.2-1

> Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 25.2. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

$k = 1$:

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

$k = 2$:

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$k = 3$:

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$k = 4$:

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$k = 5$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$k = 6$:

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

25.2-2

> Show how to compute the transitive closure using the technique of Section 25.1.

We set $w_{ij} = 1$ if (i, j) is an edge, and $w_{ij} = 0$ otherwise. Then we replace line 7 of EXTEND-SHORTEST-PATHS(L, W) by $l''_{ij} = l''_{ij} \vee (l_{ik} \wedge w_{kj})$. Then run the SLOW-ALL-PAIRS-SHORTEST-PATHS algorithm.

25.2-3

> Modify the FLOYD-WARSHALL procedure to compute the $\Pi^{(k)}$ matrices according to equations (25.6) and (25.7). Prove rigorously that for all $i \in V$, the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root i. (Hint: To show that $G_{\pi,i}$ is acyclic, first show that $\pi_{ij}^{(k)} = l$ implies $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$, according to the definition of $\pi_{ij}^{(k)}$. Then, adapt the proof of Lemma 23.16.)

```
MOD-FLOYD-WARSHALL(W)
    n = W.rows
    D(0) = W
    let π(0) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            if i != j and D[i, j](0) < ∞
                π[i, j](0) = i
    for k = 1 to n
        let D(k) be a new n × n matrix
        let π(k) be a new n × n matrix
        for i = 1 to n
            for j = 1 to n
                if d[i, j](k − 1) ≤ d[i, k](k − 1) + d[k, j](k − 1)
                    d[i, j](k) = d[i, j](k − 1)
                    π[i, j](k) = π[i, j](k − 1)
                else
                    d[i, j](k) = d[i, k](k − 1) + d[k, j](k − 1)
                    π[i, j](k) = π[k, j](k − 1)
```

In order to have that $\pi_{ij}^{(k)} = l$, we need that $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$. To see this fact, we will note that having $\pi_{ij}^{(k)} = l$ means that a shortest path from i to j last goes through l. A path that last goes through l corresponds to taking a chepest path from i to l and then following the single edge from l to j. However, This means that $d_{il} \leq d_{ij} - w_{lj}$, which we can rearrange to get the desired inequality. We can just continue following this inequality around, and if we ever get some cycle, $i_1, i_2, \ldots, i_c$, then we would have that $d_{i i_1} \leq d_{i i_1} + w_{i_1 i_2} + w_{i_2 i_3} + \cdots + w_{i_c i_1}$. So, if we subtract the common term sfrom both sides, we get that $0 \leq w_{i_c i_1} + \sum_{q=1}^{c-1} w_{i_q i_{q+1}}$. So, we have that we would only have a cycle in the precedessor graph if we ahvt that there is a zero weight cycle in the original graph. However, we would never have to go around the weight zero cycle since the constructed path of shortest weight favors ones with a fewer number of edges because of the way that we handle the equality case in equation (25.7).

25.2-4

> As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.
>
> ```
> FLOYD-WARSHALL'(W)
>     n = W.rows
>     D = W
>     for k = 1 to n
>         for i = 1 to n
>             for j = 1 to n
>                 d[i, j] = min(d[i, j], d[i, k] + d[k, j])
>     return D
> ```

(Removed)

25.2-5

> Suppose that we modify the way in which equation (25.7) handles equality:
>
> $$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$
>
> Is this alternative definition of the predecessor matrix Π correct?

If we change the way that we handle the equality case, we will still be generating a the correct values for the π matrix. This is because updating the π values to make paths that are longer but still tied for the lowest weight. Making $\pi_{ij} = \pi_{kj}$ means that we are making the shortest path from i to j passes through k at some point. This has the same cost as just going from i to j, since $d_{ij} = d_{ik} + d_{kj}$.

25.2-6

> How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle?

(Removed)

25.2-7

> Another way to reconstruct shortest paths in the Floyd-Warshall algorithm uses values $\phi_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$, where $\phi_{ij}^{(k)}$ is the highest-numbered intermediate vertex of a shortest path from i to j in which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. Give a recursive formulation for $\phi_{ij}^{(k)}$, modify the FLOYD-WARSHALL procedure to compute the $\phi_{ij}^{(k)}$ values, and rewrite the PRINT-ALLPAIRS-SHORTEST-PATH procedure to take the matrix $\Phi = \left(\phi_{ij}^{(n)}\right)$ as an input. How is the matrix Φ like the s table in the matrix-chain multiplication problem of Section 15.2?

We can recursively compute the values of $\phi_{ij}^{(k)}$ by, letting it be $\phi_{ij}^{(k-1)}$ if $d_{ik}^{(k)} + d_{kj}^{(k)} \geq d_{ij}^{(k-1)}$, and otherwise, let it be k. This works correctly because it perfectly captures whether we decided to use vertex k when we were repeatedly allowing ourselves use of each vertex one at a time. To modify Floyd-Warshall to compute this, we would just need to stick within the innermost for loop, something that computes $\phi(k)$ by this recursive rule, this would only be a constant amount of work in this innermost for loop, and so would not cause the asymptotic runtime to increase. It is similar to the s table in matrix-chain multiplication because it is computed by a similar recurrence.

If we already have the n^3 values in $\phi_{ij}^{(k)}$ provided, then we can reconstruct the shortest path from i to j because we know that the largest vertex in the path from i to j is $\phi_{ij}^{(n)}$, call it a_1. Then, we know that the largest vertex in the path before a_1 will be $\phi_{ia_1}^{(a_1-1)}$ and the largest after a_1 will be $\phi_{a_1 j}^{(a_1-1)}$. By continuing to recurse until we get that the largest element showing up at some point is NIL, we will be able to continue subdividing the path until it is entirely constructed.

25.2-8

> Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.

We can determine the vertices reachable from a particular vertex in $O(V + E)$ time using any basic graph searching algorithm. Thus we can compute the transitive closure in $O(VE + V^2)$ time by searching the graph with each vertex as the source. If $|V| = O(E)$, we're done as VE is now the dominating term in the running time bound. If not, we preprocess the graph and mark all degree-0 vertices in $O(V + E)$ time. The rows representing these vertices in the transitive closure are all 0s, which means that the algorithm remains correct if we ignore these vertices when searching. After preprocessing, $|V| = O(E)$ as $|E| \geq |V|/2$. Therefore searching can be done in $O(VE)$ time.

25.2-9

> Suppose that we can compute the transitive closure of a directed acyclic graph in $f(|V|, |E|)$ time, where f is a monotonically increasing function of $|V|$ and $|E|$. Show that the time to compute the transitive closure $G^* = (V, E^*)$ of a general directed graph $G = (V, E)$ is then $f(|V|, |E|) + O(V + E^*)$.

First, compute the strongly connected components of the directed graph, and look at it's component graph. This component graph is going to be acyclic and have at most as many vertices and at most as many edges as the original graph. Since it is acyclic, we can run our transitive closure algorithm on it. Then, for every edge (S_1, S_2) that shows up in the transitive closure of the component graph, we add an edge from each vertex in S_1 to a vertex in S_2. This takes time equal to $O(V + E')$. So, the total time required is $\leq f(|V|, |E|) + O(V + E)$.

25.3 Johnson's algorithm for sparse graphs

25.3-1

> Use Johnson's algorithm to find the shortest paths between all pairs of vertices in the graph of Figure 25.2. Show the values of h and $\hat{w}$ computed by the algorithm.

v	h(v)
1	−5
2	−3
3	0
4	−1
5	−6
6	−8

u	v	$\hat{w}(u, v)$	u	v	$\hat{w}(u, v)$
1	2	NIL	4	1	0
1	3	NIL	4	2	NIL
1	4	NIL	4	3	NIL
1	5	0	4	5	8
1	6	NIL	4	6	NIL
2	1	3	5	1	NIL
2	3	NIL	5	2	4
2	4	0	5	3	NIL
2	5	NIL	5	4	NIL
2	6	NIL	5	6	NIL
3	1	NIL	6	1	NIL
3	2	5	6	2	0
3	4	NIL	6	3	2
3	5	NIL	6	4	NIL
3	6	0	6	5	NIL

So, the d_{ij} values that we get are

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}.$$

25.3-2

> What is the purpose of adding the new vertex s to V', yielding V'?

This is only important when there are negative-weight cycles in the graph. Using a dummy vertex gets us around the problem of trying to compute $-\infty + \infty$ to find $\hat{w}$. Moreover, if we had instead used a vertex v in the graph instead of the new vertex s, then we run into trouble if a vertex fails to be reachable from v.

25.3-3

> Suppose that $w(u, v) \geq 0$ for all edges $(u, v) \in E$. What is the relationship between the weight functions w and $\hat{w}$?

If all the edge weights are nonnegative, then the values computed as the shortest distances when running Bellman-Ford will be all zero. This is because when constructing G' on the first line of Johnson's algorithm, we place an edge of weight zero from s to every other vertex. Since any path within the graph has no negative edges, its cost cannot be negative, and so, cannot beat the trivial path that goes straight from s to any given vertex. Since we have that $h(u) = h(v)$ for every u and v, the reweighting that occurs only adds and subtracts 0, and so we have that $w(u, v) = \hat{w}(u, v)$

25.3-4

> Professor Greenstreet claims that there is a simpler way to reweight edges than the method used in Johnson's algorithm. Letting $w^* = \min_{(u,v) \in E}\{w(u, v)\}$, just define $\hat{w}(u, v) = w(u, v) - w^*$ for all edges $(u, v) \in E$. What is wrong with the professor's method of reweighting?

(Removed)

25.3-5

> Suppose that we run Johnson's algorithm on a directed graph G with weight function w. Show that if G contains a 0-weight cycle c, then $\hat{w}(u, v) = 0$ for every edge (u, v) in c.

If $\delta(s, v) - \delta(s, u) \leq w(u, v)$, we have

$$\delta(s, u) \leq \delta(s, v) + (0 - w(u, v)) < \delta(s, u) + w(u, v) - w(u, v) = \delta(s, u),$$

which is impossible, thus $\delta(s, v) - \delta(s, u) = w(u, v)$, $\hat{w}(u, v) = w(u, v) + \delta(s, u) - \delta(s, v) = 0$.

25.3-6

> Professor Michener claims that there is no need to create a new source vertex in line 1 of JOHNSON. He claims that instead we can just use $G' = G$ and let s be any vertex. Give an example of a weighted, directed graph G for which incorporating the professor's idea into JOHNSON causes incorrect answers. Then show that if G is strongly connected (every vertex is reachable from every other vertex), the results returned by JOHNSON with the professor's modification are correct.

(Removed)

Problem 25-1 Transitive closure of a dynamic graph

> Suppose that we wish to maintain the transitive closure of a directed graph $G = (V, E)$ as we insert edges into E. That is, after each edge has been inserted, we want to update the transitive closure of the edges inserted so far. Assume that the graph G has no edges initially and that we represent the transitive closure as a boolean matrix.
>
> **a.** Show how to update the transitive closure $G^* = (V, E^*)$ of a graph $G = (V, E)$ in $O(V^2)$ time when a new edge is added to G.
>
> **b.** Give an example of a graph G and an edge e such that $\Omega(V^2)$ time is required to update the transitive closure after the insertion of e into G, no matter what algorithm is used.
>
> **c.** Describe an efficient algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of n insertions, your algorithm should run in total time $\sum_{i=1}^{n} t_i = O(V^3)$,

> where t_i is the time to update the transitive closure upon inserting the ith edge. Prove that your algorithm attains this time bound.

a. We can update the transitive closure in time $O(V^2)$ as follows. Suppose that we add the edge (x_1, x_2). Then, we will consider every pair of vertices (u, v). In order to of created a path between them, we would need some part of that path that goes from u to x_1 and some second part of that path that goes from x_2 to v. This means that we add the edge (u, v) to the transitive closure if and only if the transitive closure contains the edges (u, x_1) and (x_2, v). Since we only had to consider every pair of vertices once, the runtime of this update is only $O(V^2)$.

b. Suppose that we currently have two strongly connected components, each of size $|V|/2$ with no edges between them. Then their transitive closures computed so far will consist of two complete directed graphs on $|V|/2$ vertices each. So, there will be a total of $|V|^2/2$ edges adding the number of edges in each together. Then, we add a single edge from one component to the other. This will mean that every vertex in the component the edge is coming from will have an edge going to every vertex in the component that the edge is going to. So, the total number of edges after this operation will be $|V|/2 + |V|/4$ So, the number of edges increased by $|V|/4$. Since each time we add an edge, we need to use at least constant time, since there is no cheap way to add many edges at once, the total amount of time needed is $\Omega(|V|^2)$.

c. We will have each vertex maintain a tree of vertices that have a path to it and a tree of vertices that it has a path to. The second of which is the transitive closure at each step. Then, upon inserting an edge, (u, v), we will look at successive ancestors of u, and add v to their successor tree, just past u. If we ever don't insert an edge when doing this, we can stop exploring that branch of the ancestor tree. Similarly, we keep doing this for all of the ancestors of v. Since we are able to short circuit if we ever notice that we have already added an edge, we know that we will only ever reconsider the same edge at most n times, and, since the number of edges is $O(n^2)$, the total runtime is $O(n^3)$.

Problem 25-2 Shortest paths in epsilon-dense graphs

> A graph $G = (V, E)$ is ***ϵ-dense*** if $|E| = \Theta(V^{1+\epsilon})$ for some constant ϵ in the range $0 < \epsilon \leq 1$. By using d-ary min-heaps (see Problem 6-2) in shortest-paths algorithms on ϵ-dense graphs, we can match the running times of Fibonacci-heap-based algorithms without using as complicated a data structure.
>
> **a.** What are the asymptotic running times for INSERT, EXTRACT-MIN, and DECREASE-KEY, as a function of d and the number n of elements in a d-ary min-heap? What are these running times if we choose $d = \Theta(n^\alpha)$ for some constant $0 < \alpha \leq 1$? Compare these running times to the amortized costs of these operations for a Fibonacci heap.
>
> **b.** Show how to compute shortest paths from a single source on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(E)$ time. (Hint: Pick d as a function of ϵ.)
>
> **c.** Show how to solve the all-pairs shortest-paths problem on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(VE)$ time.

> **d.** Show how to solve the all-pairs shortest-paths problem in $O(VE)$ time on an ϵ-dense directed graph $G = (V, E)$ that may have negative-weight edges but has no negative-weight cycles.

a.

- INSERT: $\Theta(\log_d n) = \Theta(1/\alpha)$.
- EXTRACT-MIN: $\Theta(d \log_d n) = \Theta(n^\alpha/\alpha)$.
- DECREASE-KEY: $\Theta(\log_d n) = \Theta(1/\alpha)$.

b. Dijkstra, $O(d \log_d V \cdot V + \log_d V \cdot E)$, if $d = V^\epsilon$, then

$$\begin{aligned} O(d \log_d V \cdot V + \log_d V \cdot E) &= O(V^\epsilon \cdot V/\epsilon + E/\epsilon) \\ &= O((V^{1+\epsilon} + E)/\epsilon) \\ &= O((E + E)/\epsilon) \\ &= O(E). \end{aligned}$$

c. Run $|V|$ times Dijkstra, since the algorithm is $O(E)$ based on (b), the total time is $O(VE)$.

d. Johnson's reweight is $O(VE)$.

26 Maximum Flow

26.1 Flow network

26.1-1

> Show that splitting an edge in a flow network yields an equivalent network. More formally, suppose that flow network G contains edge (u, v), and we create a new flow network G' by creating a new vertex x and replacing (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$. Show that a maximum flow in G' has the same value as a maximum flow in G.

Suppose the maximum flow of a graph $G = (V, E)$ with source s and destination t is $|f| = \sum f(s, v)$, where $v \in V$ are vertices in the maximum flow between s and t.

We know every vertex $v \in V$ must obey the Flow conservation rule. Therefore, if we can add or delete some vertices between s and t without changing $|f|$ or violating the Flow conversation rule, then the new graph $G' = (V', E')$ will have the same maximum flow as the original graph G, and that's why we can replace edge (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$.

After doing so, vertex v_1 and v_2 still obey the Flow conservation rule since the values flow in to or flow out of v_1 and v_2 do not change at all. Meanwhile, the value $|f| = \sum f(s, v)$ remains the same.

In fact, we can split any edges in this way, even if two vertex u and v doesn't have any connection between them, we can still add a vertex y and make $c(u, y) = c(y, v) = 0$.

To conclude, we can transform any graph with or without antiparallel edges into an equivalent graph without antiparallel edges and have the same maximum flow value.

26.1-2

> Extend the flow properties and definitions to the multiple-source, multiple-sink problem. Show that any flow in a multiple-source, multiple-sink flow network corresponds to a flow of identical value in the single-source, single-sink network obtained by adding a supersource and a supersink, and vice versa.

Capacity constraint: for all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

Flow conservation: for all $u \in V - S - T$, we require $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

26.1-3

> Suppose that a flow network $G = (V, E)$ violates the assumption that the network contains a path $s \nrightarrow v \nrightarrow t$ for all vertices $v \in V$. Let u be a vertex for which there is no path $s \nrightarrow u \nrightarrow t$. Show that there must exist a maximum flow f in G such that $f(u, v) = f(v, u) = 0$ for all vertices $v \in V$.

(Removed)

26.1-4

> Let f be a flow in a network, and let α be a real number. The ***scalar flow product***, denoted αf, is a function from $V \times V$ to $\mathbb{R}$ defined by
>
> $$(\alpha f)(u, v) = \alpha \cdot f(u, v).$$
>
> Prove that the flows in a network form a ***convex set***. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α in the range $0 \leq \alpha \leq 1$.

(Removed)

26.1-5

> State the maximum-flow problem as a linear-programming problem.

$$\begin{aligned} \max \quad & \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \\ \text{s. t.} \quad & 0 \leq f(u, v) \leq c(u, v) \\ & \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) = 0 \end{aligned}$$

26.1-6

> Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining whether both his children can go to the same school as a maximum-flow problem.

(Removed)

26.1-7

> Suppose that, in addition to edge capacities, a flow network has ***vertex capacities***. That is each vertex v has a limit $l(v)$ on how much flow can pass though v. Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G. How many vertices and edges does G' have?

(Removed)

26.2 The Ford-Fulkerson method

26.2-1

> Prove that the summations in equation (26.6) equal the summations in equation (26.7).

(Removed)

26.2-2

> In Figure 26.1(b), what is the flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

$$\begin{aligned} f(S, T) &= f(s, v_1) + f(v_2, v_1) + f(v_4, v_3) + f(v_4, t) - f(v_3, v_2) = 11 + 1 + 7 + 4 - 4 = 19, \\ c(S, T) &= c(s, v_1) + c(v_2, v_1) + c(v_4, v_3) + c(v_4, t) = 16 + 4 + 7 + 4 = 31. \end{aligned}$$

26.2-3

> Show the execution of the Edmonds-Karp algorithm on the flow network of Figure 26.1(a).

If we perform a breadth first search where we consider the neighbors of a vertex as they appear in the ordering $\{s, v_1, v_2, v_3, v_4, t\}$, the first path that we will find is s, v_1, v_3, t. The min capacity of this augmenting path is 12, so we send 12 units along it. We perform a BFS on the resulting residual network. This gets us the path s, v_2, v_4, t. The min capacity along this path is 4, so we send 4 units along it. Then, the only path remaining in the residual network is $\{s, v_2, v_4, v_3, t\}$ which has a min capacity of 7, since that's all that's left, we find it in our BFS. Putting it all together, the total flow that we have found has a value of 23.

26.2-4

> In the example of Figure 26.6, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which one cancels flow?

A minimum cut corresponding to the maximum flow is $S = \{s, v_1, v_2, v_4\}$ and $T = \{v_3, t\}$. The augmenting path in part (c) cancels flow on edge (v_3, v_2).

26.2-5

> Recall that the construction in Section 26.1 that converts a flow network with multiple sources and sinks into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original network with multiple sources and sinks have finite capacity.

Since the only edges that have infinite value are those going from the supersource or to the supersink, as long as we pick a cut that has the supersource and all the original sources on one side, and the other side has the supersink as well as all the original sinks, then it will only cut through edges of finite capacity. Then, by Corollary 26.5, we have that the value of the flow is bounded above by the value of any of these types of cuts, which is finite.

26.2-6

> Suppose that each source s_i in a flow network with multiple sources and sinks produces exactly p_i units of flow, so that $\sum_{v \in V} f(s_i, v) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $\sum_{v \in V} f(v, t_j) = q_j$, where $\sum_i p_i = \sum_j q_j$. Show how to convert the problem of finding a flow f that obeys these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

$c(s, s_i) = p_i$, $c(t_j, t) = q_j$.

26.2-7

> Prove Lemma 26.2.

To check that f_p is a flow, we make sure that it satisfies both the capacity constraints and the flow constraints. First, the capacity constraints. To see this, we recall our definition of $c_f(p)$, that is, it is the smallest residual capacity of any of the edges along the path p. Since we have that the residual capacity is always less than or equal to the initial capacity, we have that each value of the flow is less than the capacity. Second, we check the flow constraints, Since the only edges that are given any flow are along a path, we have that at each vertex interior to the path, the flow in from one edge is immediately canceled by the flow out to the next vertex in the path. Lastly, we can check that its value is equal to $c_f(p)$ because, while s may show up at spots later on in the path, it will be canceled out as it leaves to go to the next vertex. So, the only net flow from s is the initial edge along the path, since it (along with all the other edges) is given flow $c_f(p)$, that is the value of the flow f_p.

26.2-8

> Suppose that we redefine the residual network to disallow edges into s. Argue that the procedure FORD-FULKERSON still correctly computes a maximum flow.

(Removed)

26.2-9

> Suppose that both f and f' are flows in a network G and we compute flow $f \uparrow f'$. Does the augmented flow satisfy the flow conservation property? Does it satisfy the capacity constraint?

(Removed)

26.2-10

> Show how to find a maximum flow in a network $G = (V, E)$ by a sequence of at most $|E|$ augmenting paths. (Hint: Determine the paths after finding the maximum flow.)

Suppose we already have a maximum flow f. Consider a new graph G where we set the capacity of edge (u, v) to $f(u, v)$. Run Ford-Fulkerson, with the modification that we remove an edge if its flow reaches its capacity. In other words, if $f(u, v) = c(u, v)$ then there should be no reverse edge appearing in residual network. This will still produce correct output in our case because we never exceed the actual maximum flow through an edge, so it is never advantageous to cancel flow. The augmenting paths chosen in this modified version of Ford-Fulkerson are precisely the ones we want. There are at most $|E|$ because every augmenting path produces at

least one edge whose flow is equal to its capacity, which we set to be the actual flow for the edge in a maximum flow, and our modification prevents us from ever destroying this progress.

26.2-11

> The ***edge connectivity*** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

Create an directed version of the graph. Then create a flow network out of it, resolving all antiparallel edges. All edges' capacities are set to 1. Pick any vertex that wasn't created for antiparallel workaround as the sink and run maximum-flow algorithm with all vertexes that aren't for antiparrrel workaround (except the sink) as sources. Find the minimum value out of all $|V| - 1$ maximum flow values.

26.2-12

> Suppose that you are given a flow network G, and G has edges entering the source s. Let f be a flow in G in which one of the edges (v, s) entering the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(E)$-time algorithm to compute f', given f, and assuming that all edge capacities are integers.

(Removed)

26.2-13

> Suppose that you wish to find, among all minimum cuts in a flow network G with integral capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G.

(Removed)

26.3 Maximum bipartite matching

26.3-1

> Run the Ford-Fulkerson algorithm on the flow network in Figure 26.8 (c) and show the residual network after each flow augmentation. Number the vertices in L top to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick the augmenting path that is lexicographically smallest.

First, we pick an augmenting path that passes through vertices 1 and 6. Then, we pick the path going through 2 and 8. Then, we pick the path going through 3 and 7. Then, the resulting residual graph has no path from s to t.

So, we know that we are done, and that we are pairing up vertices $(1, 6)$, $(2, 8)$, and $(3, 7)$. This number of unit augmenting paths agrees with the value of the cut where you cut the edges $(s, 3)$, $(6, t)$, and $(7, t)$.

26.3-2

> Prove Theorem 26.10.

We proceed by induction on the number of iterations of the while loop of Ford-Fulkerson. After the first iteration, since c only takes on integer values and $(u, v).f$ is set to 0, c_f only takes on integer values. Thus, lines 7 and 8 of Ford-Fulkerson only assign integer values to $(u, v).f$. Assume that $(u, v).f \in \mathbb{Z}$ for all (u, v) after the nth iteration. On the $(n + 1)$th iteration $c_f(p)$ is set to the minimum of $c_f(u, v)$ which is an integer by the induction hypothesis. Lines 7 and 8 compute $(u, v).f$ or $(v, u).f$. Either way, these the the sum or difference of integers by assumption, so after the $(n + 1)$th iteration we have that $(u, v).f$ is an integer for all $(u, v) \in E$. Since the value of the flow is a sum of flows of edges, we must have $|f| \in \mathbb{Z}$ as well.

26.3-3

> Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

(Removed)

26.3-4 ⋆

> A ***perfect matching*** is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the ***neighborhood*** of X as
>
> $$N(X) = \{y \in V : (x, y) \in E \text{ for some } x \in X\},$$
>
> that is, the set of vertices adjacent to some member of X. Prove ***Hall's theorem***: there exists a perfect matching in G if and only if $|A| \le |N(A)|$ for every subset $A \subseteq L$.

First suppose there exists a perfect matching in G. Then for any subset $A \subseteq L$, each vertex of A is matched with a neighbor in R, and since it is a matching, no two such vertices are matched with the same vertex in R. Thus, there are at least $|A|$ vertices in the neighborhood of A.

Now suppose that $|A| \le |N(A)|$ for all $A \subseteq L$. Run Ford-Fulkerson on the corresponding flow network. The flow is increased by 1 each time an augmenting path is found, so it will suffice to show that this happens $|L|$ times. Suppose the while loop has run fewer than L times, but there is no augmenting path. Then fewer than L edges from L to R have flow 1.

Let $v_1 \in L$ be such that no edge from v_1 to a vertex in R has nonzero flow. By assumption, v_1 has at least one neighbor $v_1' \in R$. If any of v_1's neighbors are connected to t in G_f then there is a path, so assume this is not the case. Thus, there must be some edge (v_2, v_1) with flow 1. By assumption, $N(\{v_1, v_2\}) \ge 2$, so there must exist $v_2' \ne v_1'$ such that $v_2' \in N(\{v_1, v_2\})$. If (v_2', t) is an edge in the residual network we're done since v_2' must be a neighbor of v_2, so s, v_1, v_1', v_2, v_2', and t is a path in G_f. Otherwise v_2' must have a neighbor $v_3 \in L$ such that (v_3, v_2') is in G_f. Specifically, $v_3 \ne v_1$ since (v_3, v_2') has flow 1, and $v_3 \ne v_2$ since (v_2, v_1') has flow 1, so no more flow can leave v_2 without violating conservation of flow. Again by our hypothesis, $N(\{v_1, v_2, v_3\}) \ge 3$, so there is another neighbor $v_3' \in R$.

Continuing in this fashion, we keep building up the neighborhood v_i', expanding each time we find that (v_i', t) is not an edge in G_f. This cannot happen L times, since we have run the Ford-Fulkerson while-loop fewer than $|L|$ times, so there still exist edges into t in G_f. Thus, the process must stop at some vertex v_k', and we obtain an augmenting path

$$s, v_1, v_1', v_2, v_2', v_3, \ldots, v_k, v_k', t,$$

contradicting our assumption that there was no such path. Therefore the while loop runs at least $|L|$ times. By Corollary 26.3 the flow strictly increases each time by f_p. By Theorem 26.10 f_p is an integer. In particular, it is equal to 1. This implies that $|f| \ge |L|$. It is clear that $|f| \le |L|$, so we must have $|f| = |L|$. By Corollary 26.11 this is the cardinality of a maximum matching. Since $|L| = |R|$, any maximum matching must be a perfect matching.

26.3-5 ⋆

> We say that a bipartite graph $G = (V, E)$, where $V = L \cup R$, is ***d-regular*** if every vertex $v \in V$ has degree exactly d. Every d-regular bipartite graph has $|L| = |R|$. Prove that every d-regular bipartite graph has a matching of cardinality $|L|$ by arguing that a minimum cut of the corresponding flow network has capacity $|L|$.

We convert the bipartite graph into a flow problem by making a new vertex for the source which has an edge of unit capacity going to each of the vertices in L, and a new vertex for the sink that has an edge from each of the vertices in R, each with unit capacity. We want to show that the number of edge between the two parts of the cut is at least L, this would get us by the max-flow-min-cut theorem that there is a flow of value at least $|L|$. The, we can apply the integrality theorem that all of the flow values are integers, meaning that we are selecting $|L|$ disjoint edges between L and R.

To see that every cut must have capacity at lest $|L|$, let S_1 be the side of the cut containing the source and let S_2 be the side of the cut containing the sink. Then, look at $L \cap S_1$. The source has an edge going to each of $L \cap (S_1)^c$, and there is an edge from $R \cap S_1$ to the sink that will be cut. This means that we need that there are at least $|L \cap S_1| - |R \cap S_1|$ many edges going from $L \cap S_1$ to $R \cap S_2$. If we look at the set of all neighbors of $L \cap S_1$, we get that there must be at least the same number of neighbors in R, because otherwise we could sum up the degrees going from $L \cap S_1$ to R on both sides, and get that some of the vertices in R would need to have a degree higher than d. This means that the number of neighbors of $L \cap S_1$ is at least $L \cap S_1$, but we have that they are in S_1, but there are only $|R \cap S_1|$ of those, so, we have that the size of the set of neighbors of $L \cap S_1$ that are in S_2 is at least $|L \cap S_1| - |R \cap S_1|$. Since each of these neighbors has an edge crossing the cut, we have that the total number of edges that the cut breaks is at least

$$(|L| - |L \cap S_1|) + (|L \cap S_1| - |R \cap S_1|) + |R \cap S_1| = |L|.$$

Since each of these edges is unit valued, the value of the cut is at least $|L|$.

26.4 Push-relabel algorithms

26.4-1

> Prove that, after the procedure INITIALIZE-PREFLOW(G, S) terminates, we have $s.e \le -|f^*|$, where f^* is a maximum flow for G.

(Removed)

26.4-2

> Show how to implement the generic push-relabel algorithm using $O(V)$ time per relabel operation, $O(1)$ time per push, and $O(1)$ time to select an applicable operation, for a total time of $O(V^2E)$.

We must select an appropriate data structure to store all the information which will allow us to select a valid operation in constant time. To do this, we will need to maintain a list of overflowing vertices. By Lemma 26.14, a push or a relabel operation always applies to an overflowing vertex. To determine which operation to perform, we need to determine whether $u.h = v.h + 1$ for some $v \in N(u)$. We'll do this by maintaining a list $u.\text{high}$ of all neighbors of u in G_f which have height greater than or equal to u. We'll update these attributes in the PUSH and RELABEL functions. It is clear from the pseudocode given for PUSH that we can execute it in constant time, provided we have maintain the attributes $\delta_f(u, v)$, $u.e$, $c_f(u, v)$, $(u, v).f$ and $u.h$. Each time we call PUSH(u, v) the result is that u is no longer overflowing, so we must remove it from the list.

Maintain a pointer $u.\text{overflow}$ to u's position in the overflow list. If a vertex u is not overflowing, set $u.\text{overflow} = \text{NIL}$. Next, check if v became overflowing. If so, set $v.\text{overflow}$ equal to the head of the overflow list. Since we can update the pointer in constant time and delete from a linked list given a pointer to the element to be deleted in constant time, we can maintain the list in $O(1)$.

The RELABEL operation takes $O(V)$ because we need to compute the minimum $v.h$ from among all $(u, v) \in E_f$, and there could be $|V| - 1$ many such v. We will also need to update $u.\text{high}$ during RELABEL. When RELABEL(u) is called, set $u.\text{high}$ equal to the empty list and for each vertex v which is adjacent to u, if $v.h = u.h + 1$, add u to the list $v.\text{high}$. Since this takes constant time per adjacent vertex we can maintain the attribute in $O(V)$ per call to relabel.

26.4-3

> Prove that the generic push-relabel algorithm spends a total of only $O(VE)$ time in performing all the $O(V^2)$ relabel operations.

(Removed)

26.4-4

> Suppose that we have found a maximum flow in a flow network $G = (V, E)$ using a push-relabel algorithm. Give a fast algorithm to find a minimum cut in G.

(Removed)

26.4-5

> Give an efficient push-relabel algorithm to find a maximum matching in a bipartite graph. Analyze your algorithm.

First, construct the flow network for the bipartite graph as in the previous section. Then, we relabel everything in L. Then, we push from every vertex in L to a vertex in R, so long as it is possible.

Keeping track of those that vertices of L that are still overflowing can be done by a simple bit vector. Then, we relabel everything in R and push to the last vertex. Once these operations have been done, The only possible valid operations are to relabel the vertices of L that weren't able to find an edge that they could push their flow along, so could possibly have to get a push back from R to L. This continues until there are no more operations to do. This takes time of $O(V(E + V))$.

26.4-6

> Suppose that all edge capacities in a flow network $G = (V, E)$ are in the set $\{1, 2, \ldots, k\}$. Analyze the running time of the generic push-relabel algorithm in terms of $|V|$, $|E|$, and k. (Hint: How many times can each edge support a nonsaturating push before it becomes saturated?)

The number of relabel operations and saturating pushes is the same as before. An edge can handle at most k nonsaturating pushes before it becomes saturated, so the number of nonsaturating pushes is at most $2k|V||E|$. Thus, the total number of basic operations is at most $2|V|^2 + 2|V||E| + 2k|V||E| = O(kVE)$.

26.4-7

> Show that we could change line 6 of INITIALIZE-PREFLOW to
>
> ```
> 6 s.h = |G.V| - 2
> ```
>
> without affecting the correctness or asymptotic performance of the generic pushrelabel algorithm.

(Removed)

26.4-8

> Let $\delta_f(u, v)$ be the distance (number of edges) from u to v in the residual network G_f. Show that the GENERIC-PUSH-RELABEL procedure maintains the properties that $u.h < |V|$ implies $u.h \le \delta_f(u, t)$ and that $u.h \ge |V|$ implies $u.h - |V| \le \delta_f(u, s)$.

We'll prove the claim by induction on the number of push and relabel operations. Initially, we have $u.h = |V|$ if $u = s$ and 0 otherwise. We have $s.h - |V| = 0 \le \delta_f(s, s) = 0$ and $u.h = 0 \le \delta_f(u, t)$ for all $u \ne s$, so the claim holds prior to the first iteration of the while loop on line 2 of the GENERIC-PUSH-RELABEL algorithm.

Suppose that the properties have been maintained thus far. If the next iteration is a nonsaturating push then the properties are maintained because the heights and existence of edges in the residual network are preserved. If it is a saturating push then edge (u, v) is removed from the residual network, which increases both $\delta_f(u, t)$ and $\delta_f(u, s)$, so the properties are maintained regardless of the height of u.

Now suppose that the next iteration causes a relabel of vertex u. For all v such that $(u, v) \in E_f$ we must have $u.h \le v.h$. Let $v' = \min\{v.h \mid (u, v) \in E_f\}$. There are two cases to consider.

- First, suppose that $v.h < |V|$. Then after relabeling we have
  $$u.h = 1 + v'.h \le 1 + \min_{(u,v)} \in E_f \delta_f(v, t) = \delta_f(u, t).$$
- Second, suppose that $v'.h \ge |V|$. Then after relabeling we have
  $$u.h = 1 + v'.h \le 1 + |V| + \min_{(u,v)} \in E_f \delta_f(v, s) = \delta_f(u, s) + |V|,$$
  which implies that $u.h - |V| \le \delta_f(u, s)$.

Therefore, the GENERIC-PUSH-RELABEL procedure maintains the desired properties.

26.4-9 ⋆

> As in the previous exercise, let $\delta_f(u, v)$ be the distance from u to v in the residual network G_f. Show how to modify the generic push-relabel algorithm to maintain the property that $u.h < |V|$ implies $u.h = \delta_f(u, t)$ and that $u.h \ge |V|$ implies $u.h - |V| = \delta_f(u, s)$. The total time that your implementation dedicates to maintaining this property should be $O(VE)$.

What we should do is to, for successive backwards neighborhoods of t, relabel everything in that neighborhood. This will only take at most $O(VE)$ time (see 26.4-3). This also has the upshot of making it so that once we are done with it, every vertex's height is equal to the quantity $\delta_f(u, t)$. Then, since we begin with equality, after doing this, the inductive step we had in the solution to the previous exercise shows that this equality is preserved.

26.4-10

> Show that the number of nonsaturating pushes executed by the GENERIC-PUSH-RELABEL procedure on a flow network $G = (V, E)$ is at most $4|V|^2|E|$ for $|V| \ge 4$.

Each vertex has maximum height $2|V| - 1$. Since heights don't decrease, and there are $|V| - 2$ vertices which can be overflowing, the maximum contribution of relabels to Φ over all vertices is $(2|V| - 1)(|V| - 2)$. A saturating push from u to v increases Φ by at most $v.h \le 2|V| - 1$, and there are at most $2|V||E|$ saturating pushes, so the total contribution over all saturating pushes to Φ is at most $(2|V| - 1)(2|V||E|)$. Since each nonsaturating push decrements Φ by at least on and Φ must equal zero upon termination, we must have that the number of nonsaturating pushes is at most

$$(2|V|-1)(|V|-2)+(2|V|-1)(2|V||E|)=4|V|^2|E|+2|V|^2-5|V|+3-2|V||E|.$$

Using the fact that $|E| \geq |V|-1$ and $|V| \geq 4$ we can bound the number of saturating pushes by $4|V|^2|E|$.

26.5 The relabel-to-front algorithm

26.5-1

Illustrate the execution of RELABEL-TO-FRONT in the manner of Figure 26.10 for the flow network in Figure 26.1(a). Assume that the initial ordering of vertices in L is $\langle v_1, v_2, v_3, v_4 \rangle$ and that the neighbor lists are

$$\begin{aligned} v_1.N &= \langle s, v_2, v_3 \rangle, \\ v_2.N &= \langle s, v_1, v_3, v_4 \rangle, \\ v_3.N &= \langle v_1, v_2, v_4, t \rangle, \\ v_4.N &= \langle v_2, v_3, t \rangle. \end{aligned}$$

When we initialize the preflow, we have 29 units of flow leaving s. Then, we consider v_1 since it is the first element in the L list. When we discharge it, we increase it's height to 1 so that it can dump 12 of it's excess along its edge to vertex v_3, to discharge the rest of it, it has to increase it's height to $|V|+1$ to discharge it back to s. It was already at the front, so, we consider v_2. We increase its height to 1. Then, we send all of its excess along its edge to v_4. We move it to the front, which means we next consider v_1, and do nothing because it is not overflowing. Up next is vertex v_3. After increasing its height to 1, it can send all of its excess to t. This puts v_3 at the front, and we consider the non-overflowing vertices v_2 and v_1. Then, we consider v_4, it increases its height to 1, then sends 4 units to t. Since it still has an excess of 9 units, it increases its height once again. Then it becomes valid for it to send flow back to v_2 or to v_3. It considers v_2 first because of the ordering of its neighbor list. This means that 9 units of flow are pushed back to v_2. Since $v_4.h$ increased, it moves to the front of the list Then, we consider v_2 since it is the only still overflowing vertex. We increase its height to 3. Then, it is overflowing by 9 so it increases its height to 3 to send 9 units to v_4. It's height increased so it goes to the of the list. Then, we consider v_4, which is overflowing. It pushes 7 units to v_3. Since it is still overflowing by 2, it increases its height to 4 and pushes the rest back to v_2 and goes to the front of the list. Up next is v_2, which increases its height by 2 to sends its overflow to v_4. The excess flow keeps bobbing around the four vertices, each time requiring them to increase their height a bit to discharge to a neighbor only to have that neighbor increase to discharge it back until v_2 has increased in height enough to send all of it's excess back to s. Last but not least, v_3 pushes its overflow of 7 units to t, and gives us a maximum flow of 23.

26.5-2 ⋆

We would like to implement a push-relabel algorithm in which we maintain a firstin, first-out queue of overflowing vertices. The algorithm repeatedly discharges the vertex at the head of the queue, and any vertices that were not overflowing before the discharge but are overflowing afterward are placed at the end of the queue. After the vertex at the head of the queue is discharged, it is removed. When the queue is empty, the algorithm terminates. Show how to implement this algorithm to compute a maximum flow in $O(V^3)$ time.

Initially, the vertices adjacent to s are the only ones which are overflowing. The implementation is as follows:

```
PUSH-RELABEL-QUEUE(G, s)
    INITIALIZE-PREFLOW(G, s)
    let q be a new empty queue
    for v ∈ G.Adj[s]
        PUSH(q, v)
    while q.head != NIL
        DISCHARGE(q.head)
        POP(q)
```

Note that we need to modify the DISCHARGE algorithm to push vertices v onto the queue if v was not overflowing before a discharge but is overflowing after one.

Between lines 7 and 8 of $\text{DISCHARGE}(u)$, add the line "if $v.e > 0$, $\text{PUSH}(q, v)$." This is an implementation of the generic push-relabel algorithm, so we know it is correct. The analysis of runtime is almost identical to that of Theorem 26.30. We just need to verify that there are at most $|V|$ calls to DISCHARGE between two consecutive relabel operations. Observe that after calling $\text{PUSH}(u, v)$, Corollary 26.28 tells us that no admissible edges are entering v. Thus, once v is put into the queue because of the push, it can't be added again until it has been relabeled. Thus, at most $|V|$ vertices are added to the queue between relabel operations.

26.5-3

Show that the generic algorithm still works if RELABEL updates $u.h$ by simply computing $u.h = u.h + 1$. How would this change affect the analysis of RELABEL-TO-FRONT?

If we change relabel to just increment the value of u, we will not be ruining the correctness of the Algorithm. This is because since it only applies when $u.h \leq v.h$, we won't be every creating a graph where h ceases to be a height function, since $u.h$ will only ever be increasing by exactly 1 whenever relabel is called, ensuring that $u.h + 1 \leq v.h$. This means that Lemmatae 26.15 and 26.16 will still hold. Even Corollary 26.21 holds since all it counts on is that relabel causes some vertex's h value to increase by at least 1, it will still work when we have all of the operations causing it to increase by exactly 1. However, Lemma 26.28 will no longer hold. That is, it may require more than a single relabel operation to cause an admissible edge to appear, if for example, $u.h$ was strictly less than the h values of all its neighbors. However, this lemma is not used in the proof of Exercise 26.4-3, which bounds the number of relabel operations. Since the number of relabel operations still have the same bound, and we know that we can simulate the old relabel operation by doing (possibly many) of these new relabel operations, we have the same bound as in the original algorithm with this different relabel operation.

26.5-4 ⋆

Show that if we always discharge a highest overflowing vertex, we can make the push-relabel method run in $O(V^3)$ time.

We'll keep track of the heights of the overflowing vertices using an array and a series of doubly linked lists. In particular, let A be an array of size $|V|$, and let $A[i]$ store a list of the elements of height i. Now we create another list L, which is a list of lists. The head points to the list containing the vertices of highest height. The next pointer of this list points to the next nonempty list stored in A, and so on. This allows for constant time insertion of a vertex into A, and also constant time access to an element of largest height, and because all lists are doubly linked, we can add and delete elements in constant time. Essentially, we are implementing the algorithm of Exercise 26.5-2, but with the queue replaced by a priority queue with constant time operations. As before, it will suffice to show that there are at most $|V|$ calls to discharge between consecutive relabel operations.

Consider what happens when a vertex v is put into the priority queue. There must exist a vertex u for which we have called $\text{PUSH}(u, v)$. After this, no ad- missible edge is entering v, so it can't be added to the priority queue again until after a relabel operation has occurred on v. Moreover, every call to DISCHARGE terminates with a PUSH, so for every call to DISCHARGE there is another vertex which can't be added until a relabel operation occurs. After $|V|$ DISCHARGE operations and no relabel operations, there are no remaining valid PUSH operations, so either the algorithm terminates, or there is a valid relabel operation which is performed. Thus, there are $O(V^3)$ calls to DISCHARGE. By carrying out the rest of the analysis of Theorem 26.30, we conclude that the runtime is $O(V^3)$.

26.5-5

Suppose that at some point in the execution of a push-relabel algorithm, there exists an integer $0 < k \leq |V|-1$ for which no vertex has $v.h = k$. Show that all vertices with $v.h > k$ are on the source side of a minimum cut. If such a k exists, the ***gap heuristic*** updates every vertex $v \in V - \{s\}$ for which $v.h > k$, to set $v.h = \max(v.h, |V|+1)$. Show that the resulting attribute h is a height function. (The gap heuristic is crucial in making implementations of the push-relabel method perform well in practice.)

Suppose to try and obtain a contradiction that there were some minimum cut for which a vertex that had $v.h > k$ were on the sink side of that cut. For that minimum cut, there is a residual flow network for which that cut is saturated. Then, if there were any vertices that were also on the sink side of the cut which had an edge going to v in this residual flow network, since it's h value cannot be equal to k, we know that it must be greater than k since it could be only at most one less than v. We can continue in this way to let S be the set of vertices on the sink side of the graph which have an h value greater than k. Suppose that there were some simple path from a vertex in S to s. Then, at each of these steps, the height could only decrease by at most 1, since it cannot get from above k to 0 without going through k, we know that there is no path in the residual flow network going from a vertex in S to s. Since a minimal cut corresponds to disconnected parts of the residual graph for a maximum flow, and we know there is no path from S to s, there is a minimum cut for which S lies entirely on the source side of the cut. This was a contradiction to how we selected v, and so have shown the first claim.

Now we show that after updating the h values as suggested, we are still left with a height function. Suppose we had an edge (u, v) in the residual graph. We knew from before that $u.h \leq v.h + 1$. However, this means that if $u.h > k$, so must be $v.h$. So, if both were above k, we would be making them equal, causing the inequality to still hold. Also, if just $v.k$ were above k, then we have not decreased it's h value, meaning that the inequality also still must hold. Since we have not changed the value of $s.h$, and $t.h$, we have all the required properties to have a height function after modifying the h values as described.

Problem 26-1 Escape problem

An $n \times n$ ***grid*** is an undirected graph consisting of n rows and n columns of vertices, as shown in Figure 26.11. We denote the vertex in the ith row and the jth column by (i, j). All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points (i, j) for which $i = 1$, $i = n$, $j = 1$, or $j = n$.

Given $m \leq n^2$ starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ in the grid, the ***escape problem*** is to determine whether or not there are m vertex-disjoint paths from the starting points to any m different points on the boundary. For example, the grid in Figure 26.11(a) has an escape, but the grid in Figure 26.11(b) does not.

a. Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show that determining the maximum flow in a network with edge and vertex capacities can be reduced to an ordinary maximum-flow problem on a flow network of comparable size.

b. Describe an efficient algorithm to solve the escape problem, and analyze its running time.

a. This problem is identical to exercise 26.1-7.

b. Construct a vertex constrained flow network from the instance of the escape problem by letting our flow network have a vertex (each with unit capacity) for each intersection of grid lines, and have a bidirectional edge with unit capacity for each pair of vertices that are adjacent in the grid. Then, we will put a unit capacity edge going from s to each of the distinguished vertices, and a unit capacity edge going from each vertex on the sides of the grid to t. Then, we know that a solution to this problem will correspond to a solution to the escape problem because all of the augmenting paths will be a unit flow, because every edge has unit capacity. This means that the flows through the grid will be the paths taken. This gets us the escaping paths if the total flow is equal to m (we know it cannot be greater than m by looking at the cut which has s by itself). And, if the max flow is less than m, we know that the escape problem is not solvable, because otherwise we could construct a flow with value m from the list of disjoint paths that the people escaped along.

Problem 26-2 Minimum path cover

A ***path cover*** of a directed graph $G = (V, E)$ is a set P of vertex-disjoint paths such that every vertex in V is included in exactly one path in P. Paths may start and end anywhere, and they may be of any length, including 0. A ***minimum path cover*** of G is a path cover containing the fewest possible paths.

a. Give an efficient algorithm to find a minimum path cover of a directed acyclic graph $G = (V, E)$. (Hint: Assuming that $V = \{1, 2, \ldots, n\}$, construct the graph $G' = (V', E')$, where

$$\begin{aligned} V' &= \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\}, \\ E' &= \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}, \end{aligned}$$

and run a maximum-flow algorithm.)

b. Does your algorithm work for directed graphs that contain cycles? Explain.

a. Set up the graph G' as defined in the problem, give each edge capacity 1, and run a maximum-flow algorithm. If claim that if (x_i, y_j) has flow 1 in the maximum flow and we set (i, j) to be an edge in our path cover, then the result is a minimum path cover. First observe that no vertex appears twice in the same path. If it did, then we would have $f(x_i, y_j) = f(x_k, y_j)$ for some $i \neq k \neq j$. However, this contradicts the conservation of flow, since the capacity leaving y_j is only 1. Moreover, since the capacity from s to x_i is 1, we can never have two edges of the form (x_i, y_j) and (x_i, y_k) for $k \neq j$. We can ensure every vertex is included in some path by asserting that if there is no edge (x_i, y_j) or (x_j, y_i) for some j, then j will be on a path by itself. Thus, we are guaranteed to obtain a path cover. If there are k paths in a cover of n vertices, then they will consist of $n-k$ edges in total. Given a path cover, we can recover a flow by assigning edge (x_i, y_j) flow 1 if and only if (i, j) is an edge in one of the paths in the cover. Suppose that the maximum flow algorithm yields a cover with k paths, and hence flow $n-k$, but a minimum path cover uses strictly fewer than k paths. Then it must use strictly more than $n-k$ edges, so we can recover a flow which is larger than the one previously found, contradicting the fact that the previous flow was maximal. Thus, we find a minimum path cover. Since the maximum flow in the graph corresponds to finding a maximum matching in the bipartite graph obtained by considering the induced subgraph of G' on $\{1, 2, \ldots, n\}$, section 26.3 tells us that we can find a maximum flow in $O(VE)$.

b. This doesn't work for directed graphs which contain cycles. To see this, consider the graph on $\{1, 2, 3, 4\}$ which contains edges $(1, 2)$, $(2, 3)$, $(3, 1)$, and $(4, 3)$. The desired output would be a single path $4, 3, 1, 2$ but flow which assigns edges (x_1, y_2), (x_2, y_3), and (x_3, y_1) flow 1 is maximal.

Problem 26-3 Algorithmic consulting

Professor Gore wants to open up an algorithmic consulting company. He has identified n important subareas of algorithms (roughly corresponding to different portions of this textbook), which he represents by the set $A = \{A_1, A_2, \ldots, A_n\}$. In each subarea A_k, he can hire an expert in that area for c_k dollars. The consulting company has lined up a set $J = \{J_1, J_2, \ldots, J_m\}$ of potential jobs. In order to perform job J_i, the company needs to have hired experts in a subset $R_i \subseteq A$ of subareas. Each expert can work on multiple jobs simultaneously. If the company chooses to accept job J_i, it must have hired experts in all subareas in R_i, and it will take in revenue of p_i dollars.

Professor Gore's job is to determine which subareas to hire experts in and which jobs to accept in order to maximize the net revenue, which is the total income from jobs accepted minus the total cost of employing the experts.

Consider the following flow network G. It contains a source vertex s, vertices $A_1, A_2, \ldots, A_n$, vertices $J_1, J_2, \ldots, J_m$, and a sink vertex t. For $k = 1, 2, \ldots, n$, the flow network contains an edge (s, A_k) with capacity $c(s, A_k) = c_k$, and for $i = 1, 2, \ldots, m$, the flow network contains an edge (J_i, t) with capacity $c(J_i, t) = p_i$. For $k = 1, 2, \ldots, n$ and $i = 1, 2, \ldots, m$, if $A_k \in R_i$, then G contains an edge (A_k, J_i) with capacity $c(A_k, J_i) = \infty$.

a. Show that if $J_i \in T$ for a finite-capacity cut (S, T) of G, then $A_k \in T$ for each $A_k \in R_i$.

b. Show how to determine the maximum net revenue from the capacity of a minimum cut of G and the given p_i values.

c. Give an efficient algorithm to determine which jobs to accept and which experts to hire. Analyze the running time of your algorithm in terms of m, n, and $r = \sum_{i=1}^{m} |R_i|$.

a. Suppose to a contradiction that there were some $J_i \in T$, and some $A_k \in R_i$ so that $A_k \notin T$. However, by the definition of the flow network, there is an edge of infinite capacity going from A_k to J_i because $A_k \in R_i$. This means that there is an edge of infinite capacity that is going across the given cut. This means that the capacity of the cut is infinite, a contradiction to the given fact that the cut was finite capacity.

b. Though tempting, it doesn't suffice to just look at the experts that are on the s side of the cut. To see why this doesn't work, imagine there's one specialized skill areal, such as "Computer power switch operator", that is required for every job. Then, any finite cut that would include any job getting done would requiring that this expert be hired. However, since there is an infinite capacity edge coming from him to every other job, then all of the experts need for all the other jobs would also need to be hired. So, if we have this obiquitously required employee, any minimum cut would have to be all or nothing, but it is trivial to find a counterexample to this being optimal.

In order for this problem to be solvable, one must assume that for every expert you've hired, you do all of the jobs that he is required for. If this is the case, then let $S_k \subseteq [n]$ be the indices of the experts that lie on the source side of the cut, and let $S_i \subseteq [m]$ be the indices of jobs that lie on the source side of the cut, then the net revenue is just

$$\sum_{S_i} p_i - \sum_{S_k} c_k$$

To see this is minimum, transferring over some set of experts and tasks from the sink side to the source side causes the capacity to go down by the cost of those experts and go up by the revenue of those jobs. If the cut was minimal than this must be a positive change, so the revenue isn't enough to justify the hire, meaning that those jobs that were on the source side in the minimal cut are exactly the jobs to attempt.

c. Again, to get a solution, we must make the assumption that for every expert that is hired, all jobs that that expert is required for must be completed. Basically just run either the $O(V^3)$ relabel-to-front algorithm described in section 26.5 on the flow network, and hire the experts that are on the source side of the cut. By the previous part, we know that this gets us the best outcome. The number of edges in the flow network is $m + n + r$, and the number of vertices is $2 + m + n$, so the runtime is just $O((2 + m + n)^3)$, so it's cubic in $\max(m, n)$. There is no dependence on R using this algorithm, but this is reasonable since we have the inherent bound that $r < mn$, which is a lower order term. Without this unstated assumption, I suspect that there isn't an efficient solution possible, but cannot think of what NP-complete problem you would use for the reduction.

Problem 26-4 Updating maximum flow

Let $G = (V, E)$ be a flow network with source s, sink t, and integer capacities. Suppose that we are given a maximum flow in G.

a. Suppose that we increase the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

b. Suppose that we decrease the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

a. If there exists a minimum cut on which (u, v) doesn't lie then the maximum flow can't be increased, so there will exist no augmenting path in the residual network. Otherwise it does cross a minimum cut, and we can possibly increase the flow by 1. Perform one iteration of Ford-Fulkerson. If there exists an augmenting path, it will be found and increased on this iteration. Since the edge capacities are integers, the flow values are all integral. Since flow strictly increases, and by an integral amount each time, a single iteration of the while loop of line 3 of Ford-Fulkerson will increase the flow by 1, which we know to be maximal. To find an augmenting path we use a BFS, which runs in $O(V + E') = O(V + E)$.

b. If the edge's flow was already at least 1 below capacity then nothing changes. Otherwise, find a path from s to t which contains (u, v) using BFS in $O(V + E)$. Decrease the flow of every edge on that path by 1. This decreases total flow by 1. Then run one iteration of the while loop of Ford-Fulkerson in $O(V + E)$. By the argument given in part a, everything is integer valued and flow strictly increases, so we will either find no augmenting path, or will increase the flow by 1 and then terminate. "

Problem 26-5 Maximum flow by scaling

Let $G = (V, E)$ be a flow network with source s, sink t, and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max_{(u,v)\in E} c(u, v)$.

a. Argue that a minimum cut of G has capacity at most $C|E|$.

b. For a given number K, show how to find an augmenting path of capacity at least K in $O(E)$ time, if such a path exists.

We can use the following modification of FORD-FULKERSON-METHOD to compute a maximum flow in G:

```
MAX-FLOW-BY-SCALING(G, s, t)
    C = max_{(u, v) ∈ E} c(u, v)
    initialize flow f to 0
    K = 2^{floor(lg C)}
    while K ≥ 1
        while there exists an augmenting path p of capacity at least K
 augment flow f along p
        K = K / 2
    return f
```

c. Argue that MAX-FLOW-BY-SCALING returns a maximum flow.

d. Show that the capacity of a minimum cut of the residual network G_f is at most $2K|E|$ each time line 4 is executed.

e. Argue that the inner **while** loop of lines 5–6 executes $O(E)$ times for each value of K.

f. Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^2 \lg C)$ time.

a. Since the capacity of a cut is the sum of the capacity of the edges going from a vertex on one side to a vertex on the other, it is less than or equal to the sum of the capacities of all of the edges. Since each of the edges has a capacity that is $\leq C$, if we were to replace the capacity of each edge with C, we would only be potentially increasing the sum of the capacities of all the edges. After so changing the capacities of the edges, the sum of the capacities of all the edges is equal to $C|E|$, potentially an overestimate of the original capacity of any cut, and so of the minimum cut.

b. Since the capacity of a path is equal to the minimum of the capacities of each of the edges along that path, we know that any edges in the residual network that have a capacity less than K cannot be used in such an augmenting path. Similarly, so long as all the edges have a capacity of at least K, then the capacity of the augmenting path, if it is found, will be of capacity at least K. This means that all that needs be done is remove from the residual network those edges whose capacity is less than K and then run BFS.

c. Since K starts out as a power of 2, and through each iteration of the while loop on line 4, it decreases by a factor of two until it is less than 1. There will be some iteration of that loop when $K = 1$. During this iteration, we will be using any augmenting paths of capacity at least 1 when running the loop on line 5. Since the original capacities are all integers, the augmenting paths at each step will be integers, which means that no augmenting path will have a capacity of less than 1. So, once the algorithm terminates, there will be no more augmenting paths since there will be no more augmenting paths of capacity at least 1.

d. Each time line 4 is executed we know that there is no augmenting path of capacity at least $2K$. To see this fact on the initial time that line 4 is executed we just note that $2K = 2 \cdot 2^{\lfloor \lg C \rfloor} > 2 \cdot 2^{\lg C - 1} = 2^{\lg C} = C$. Then, since an augmenting path is limited by the capacity of the smallest edge it contains, and all the edges have a capacity at most C, no augmenting path will have a capacity greater than that. On subsequent times executing line 4, the loop of line 5 during the previous execution of the outer loop will of already used up and capacious augmenting paths, and would only end once there are no more.

Since any augmenting path must have a capacity of less than $2K$, we can look at each augmenting path p, and assign to it an edge e_p which is any edge whose capacity is tied for smallest among all the edges along the path. Then, removing all of the edges e_p would disconnect the residual network since every possible augmenting path goes through one of those edge. We know that there are at most $|E|$ of them since they are a subset of the edges. We also know that each of them has capacity at most $2K$ since that was the value of the augmenting path they were selected to be tied for cheapest in. So, the total cost of this cut is $2K|E|$.

e. Each time that the inner while loop runs, we know that it adds an amount of flow that is at least K, since that's the value of the augmenting path. We also know that before we start that while loop, there is a cut of cost $\leq 2K|E|$. This means that the most flow we could possibly add is $2K|E|$. Combining these two facts, we get that the most cuts possible is $\frac{2K|E|}{K} = 2|E| \in O(|E|)$.

f. We only execute the outermost **for** loop $\lg C$ many times since $\lg(2^{\lfloor \lg C \rfloor}) \leq \lg C$. The inner while loop only runs $O(|E|)$ many times by the previous part. Finally, every time the inner **for** loop runs, the operation it does can be done in time $O(|E|)$ by part (b). Putting it all together, the runtime is $O(|E|^2 \lg C)$.

Problem 26-6 The Hopcroft-Karp bipartite matching algorithm

In this problem, we describe a faster algorithm, due to Hopcroft and Karp, for p finding a maximum matching in a bipartite graph. The algorithm runs in $O(\sqrt{V}E)$ time. Given an undirected, bipartite graph $G = (V, E)$, where $V = L \cup R$ and all edges have exactly one endpoint in L, let M be a matching in G. We say that a simple path P in G is an ***augmenting path*** with respect to M if it starts at an unmatched vertex in L, ends at an unmatched vertex in R, and its edges belong alternately to M and $E - M$. (This definition of an augmenting path is related to, but different from, an augmenting path in a flow network.) In this problem, we treat a path as a sequence of edges, rather than as a sequence of vertices. A shortest augmenting path with respect to a matching M is an augmenting path with a minimum number of edges.

Given two sets A and B, the ***symmetric difference*** $A \oplus B$ is defined as $(A - B) \cup (B - A)$, that is, the elements that are in exactly one of the two sets.

a. Show that if M is a matching and P is an augmenting path with respect to M, then the symmetric difference $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$. Show that if $P_1, P_2, \ldots, P_k$ are vertex-disjoint augmenting paths with respect to M, then the symmetric difference $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ is a matching with cardinality $|M| + k$.

The general structure of our algorithm is the following:

```
HOPCROPFT-KARP(G)
    M = ∅
    repeat
        let P = {P[1], P[2], ..., P[k]} be a maximal set of vertex-
disjoint shortest augmenting paths with respect to M
        M = M ⊕ (P[1] ∪ P[2] ∪ ... ∪ P[k])
    until P == ∅
    return M
```

The remainder of this problem asks you to analyze the number of iterations in the algorithm (that is, the number of iterations in the **repeat** loop) and to describe an implementation of line 3.

b. Given two matchings M and M^* in G, show that every vertex in the graph $G' = (V, M \oplus M^*)$ has degree at most 2. Conclude that G' is a disjoint union of simple paths or cycles. Argue that edges in each such simple path or cycle belong alternately to M or M^*. Prove that if $|M| \leq |M^*|$, then $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M.

Let l be the length of a shortest augmenting path with respect to a matching M, and let $P_1, P_2, \ldots, P_k$ be a maximal set of vertex-disjoint augmenting paths of length l with respect to M. Let $M' = M \oplus (P_1 \cup \cdots \cup P_k)$, and suppose that P is a shortest augmenting path with respect to M'.

c. Show that if P is vertex-disjoint from $P_1, P_2, \ldots, P_k$, then P has more than l edges.

d. Now suppose that P is not vertex-disjoint from $P_1, P_2, \ldots, P_k$. Let A be the set of edges $(M \oplus M') \oplus P$. Show that $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ and that $|A| \geq (k + 1)l$. Conclude that P has more than l edges.

e. Prove that if a shortest augmenting path with respect to M has l edges, the size of the maximum matching is at most $|M| + |V|/(l + 1)$.

f. Show that the number of **repeat** loop iterations in the algorithm is at most $2\sqrt{|V|}$. (Hint: By how much can M grow after iteration number $\sqrt{|V|}$?)

g. Give an algorithm that runs in $O(E)$ time to find a maximal set of vertexdisjoint shortest augmenting paths $P_1, P_2, \ldots, P_k$ for a given matching M. Conclude that the total running time of HOPCROFT-KARP is $O(\sqrt{V}E)$.

a. Suppose M is a matching and P is an augmenting path with respect to M. Then P consists of k edges in M, and $k + 1$ edges not in M. This is because the first edge of P touches an unmatched vertex in L, so it cannot be in M. Similarly, the last edge in P touches an unmatched vertex in R, so the last edge cannot be in M. Since the edges alternate being in or not in M, there must be exactly one more edge not in M than in M. This implies that

$$|M \oplus P| = |M| + |P| - 2k = |M| + 2k + 1 - 2k = |M| + 1,$$

since we must remove each edge of M which is in P from both M and P. Now suppose $P_1, P_2, \ldots, P_k$ are vertex-disjoint augmenting paths with respect to M. Let k_i be the number of edges in P_i which are in M, so that $|P_i| = 2k + i + 1$. Then we have

$$M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) = |M| + |P_1| + \cdots + |P_k| - 2k_1 - 2k_2 - \cdots - 2k_k = |M| + k.$$

To see that we in fact get a matching, suppose that there was some vertex v which had at least 2 incident edges e and e'. They cannot both come from M, since M is a matching. They cannot both come from P since P is simple and every other edge of P is removed. Thus, $e \in M$ and $e' \in P \backslash M$. However, if $e \in M$ then $e \in P$, so $e \notin M \oplus P$, a contradiction. A similar argument gives the case of $M \oplus (P_1 \cup \cdots \cup P_k)$.

b. Suppose some vertex in G' has degree at least 3. Since the edges of G' come from $M \oplus M^*$, at least 2 of these edges come from the same matching. However, a matching never contains two edges with the same endpoint, so this is impossible. Thus every vertex has degree at most 2, so G' is a disjoint union of simple paths and cycles. If edge (u, v) is followed by edge (z, w) in a simple path or cycle then we must have $v = z$. Since two edges with the same endpoint cannot appear in a matching, they must belong alternately to M and M^*. Since edges alternate, every cycle has the same number of edges in each matching and every path has at most one more edge in one matching than in the other. Thus, if $|M| \leq |M^*|$ there must be at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M.

c. Every vertex matched by M must be incident with some edge in M'. Since P is augmenting with respect to M', the left endpoint of the first edge of P isn't incident to a vertex touched by an edge in M'. In particular, P starts at a vertex in L which is unmatched by M since every vertex of M is incident with an edge in M'. Since P is vertex disjoint from $P_1, P_2, \ldots, P_k$, any edge of P which is in M' must in fact be in M and any edge of P which is not in M' cannot be in M. Since P has edges alternately in M' and $E - M'$, P must in fact have edges alternately in M and $E - M$. Finally, the last edge of P must be incident to a vertex in R which is unmatched by M'. Any vertex unmatched by M' is also unmatched by M, so P is an augmenting path for M. P must have length at least l since l is the length of the shortest augmenting path with respect to M. If P had length exactly l, then this would contradict the fact that $P_1 \cup \cdots \cup P_k$ is a maximal set of vertex disjoint paths of length l because we could add P to the set. Thus P has more than l edges.

d. Any edge in $M \oplus M'$ is in exactly one of M or M'. Thus, the only possible contributing edges from M' are from $P_1 \cup \cdots \cup P_k$. An edge from M can contribute if and only if it is not in exactly one of M and $P_1 \cup \cdots \cup P_k$, which means it must be in both. Thus, the edges from M are redundant so $M \oplus M' = (P_1 \cup \cdots \cup P_k)$ which implies $A = (P_1 \cup \cdots \cup P_k) \oplus P$.

Now we'll show that P is edge disjoint from each P_i. Suppose that an edge e of P is also an edge of P_i for some i. Since P is an augmenting path with respect to M' either $e \in M'$ or $e \in E - M'$. Suppose $e \in M'$. Since P is also augmenting with respect to M, we must have $e \in M$. However, if e is in M and M', then e cannot be in any of the P_i's by the definition of M'. Now suppose $e \in E - M'$. Then $e \in E - M$ since P is augmenting with respect to M. Since e is an edge of P_i, $e \in E - M'$ implies that $e \in M$, a contradiction.

Since P has edges alternately in M' and $E - M'$ and is edge disjoint from $P_1 \cup \cdots \cup P_k$, P is also an augmenting path for M, which implies $|P| \geq l$. Since every edge in A is disjoint we conclude that $|A| \geq (k + 1)l$.

e. Suppose M^* is a matching with strictly more than $|M| + |V|/(l + 1)$ edges. By part (b) there are strictly more than $|V|/(l + 1)$ vertex-disjoint augmenting paths with respect to M. Each one of these contains at least l edges, so it is incident on $l + 1$ vertices. Since the paths are vertex disjoint, there are strictly more than $|V|(l + 1)/(l + 1)$ distinct vertices incident with these paths, a contradiction. Thus, the size of the maximum matching is at most $|M| + |V|/(l + 1)$.

f. Consider what happens after iteration number $\sqrt{|V|}$. Let M^* be a maximal matching in G. Then $|M^*| \geq |M|$ so by part (b), $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex disjoint augmenting paths with respect to M. By part (c), each of these is also a an augmenting path for M. Since each has length $\sqrt{|V|}$, there can be at most $\sqrt{|V|}$ such paths, so $|M^*| - |M| \leq \sqrt{|V|}$. Thus, only $\sqrt{|V|}$ additional iterations of the repeat loop can occur, so there are at most $2\sqrt{|V|}$ iterations in total.

g. For each unmatched vertex in L we can perform a modified BFS to find the length of the shortest path to an unmatched vertex in R. Modify the BFS to ensure that we only traverse an edge if it causes the path to

alternate between an edge in M and an edge in $E - M$. The first time an unmatched vertex in R is reached we know the length k of a shortest augmenting path.

We can use this to stop our search early if at any point we have traversed more than that number of edges. To find disjoint paths, start at the vertices of R which were found at distance k in the BFS. Run a DFS backwards from these, which maintains the property that the next vertex we pick has distance one fewer, and the edges alternate between being in M and $E - M$. As we build up a path, mark the vertices as used so that we never traverse them again. This takes $O(E)$, so by part (f) the total runtime is $O(\sqrt{V}E)$.