Chapter 22

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 22.1-1

Since it seems as though the list for the neighbors of each vertex v is just an undecorated list, to find the length of each would take time $O(out-degree(v))$. So, the total cost will be $\sum_{v \in V} O(outdegree(v)) = O(|E| + |V|)$. Note that the $|V|$ showing up in the asymptotics is necessary, because it still takes a constant amount of time to know that a list is empty. This time could be reduced to $O(|V|)$ if for each list in the adjacency list representation, we just also stored its length.

To compute the in degree of each vertex, we will have to scan through all of the adjacency lists and keep counters for how many times each vertex has appeared. As in the previous case, the time to scan through all of the adjacency lists takes time $O(|E| + |V|)$.

Exercise 22.1-2

The adjacency list representation:

$$
\begin{aligned}
&1 : 2, 3\\
&2 : 1, 4, 5\\
&3 : 1, 6, 7\\
&4 : 2\\
&5 : 2\\
&6 : 3\\
&7 : 3.
\end{aligned}
$$

The adjacency matrix representation:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Exercise 22.1-3

For the adjacency matrix representation, to compute the graph transpose, we just take the matrix transpose. This means looking along every entry above the diagonal, and swapping it with the entry that occurs below the diagonal. This takes time $O(|V|^2)$.

For the adjacency list representation, we will maintain an initially empty adjacency list representation of the transpose. Then, we scan through every list in the original graph. If we are in the list corresponding to vertex v and see u as an entry in the list, then we add an entry of v to the list in the transpose graph corresponding to vertex u. Since this only requires a scan through all of the lists, it only takes time $O(|E| + |V|)$

Exercise 22.1-4

Create an array A of size $|V|$. For a list in the adjacency list corresponding to vertex v, examine items on the list one by one. If any item is equal to v, remove it. If vertex u appears on the list, examine $A[u]$. If it's not equal to v, set it equal to v. If it's equal to v, remove u from the list. Since we have constant time lookup in the array, the total runtime is $O(V + E)$.

Exercise 22.1-5

From the adjacency matrix representation, if we take the square of the matrix, we are left an edge between all pairs of vertices that are separated by a path of exactly 2, so, to get the desired notion of the square of a graph, we also just have to add in the vertices that are separated by only a single edge in G, that is the entry u, v in the final resulting matrix should be one iff either $G^2[u, v]$ or $G[u, v]$ are one. Taking the square of a matrix can be done with a matrix multiplication, which at the time of writing, can be most efficiently done by the Coppersmith-Windograd algorithm which takes time $O(|V|^{2.3728639})$. Since the other operation for computing the final result only takes time $O(|V|^2)$, the total runtime is $O(|V|^{2.3728639})$.

If we are given an adjacency list representation, we can find the desired resulting graph by first computing the transpose graph G^T from exercise 22.1-3 in $O(|V| + |E|)$ time. Then, our initally empty adjacency list representation of

G^2 will be added to as follows. As we scan though the list of each vertex, say v, and see a entry going to u, then we add u to the list corresponding to v, but also add u to the list of everything on v's list in G^T. This means that we may take as much as $O(|E||V| + |V|)$ time since, we have to spend potentially $|V|$ time as we process each edge.

Exercise 22.1-6

Start by examining position (1,1) in the adjacency matrix. When examining position (i, j), if a 1 is encountered, examine position $(i+1, j)$. If a 0 is encountered, examine position $(i, j + 1)$. Once either i or j is equal to $|V|$, terminate. I claim that if the 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 in the adjacency matrix is all 0's, and column k is all 1's except for position (k, k) which is a 0. Thus, once row k is hit, the algorithm will continue to increment 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 whether or not i in fact corresponds to a sink is done in $O(V)$. Therefore the entire process takes $O(V)$.

Exercise 22.1-7

We have two cases, one for the diagonal entries and one for the non-diagonal entries.

The entry of $[i, i]$ for some i represents the sum of the in and ourt degrees of the vertex that i corresponds to. To see this, we recall that an entry in a matrix product is the dot product of row i in B and column i in B^T. But, column i in B^T is the same as row i in B. So, we have that the entry is just row i of B dotted with itself, that is

$$\sum_{j=1}^{|E|} b_{ij}^2$$

However, since b_{ij} only takes values in $\{-1, 0, 1\}$, we have that b_{ij}^2 only takes values in $\{0, 1\}$, taking zero iff $b_{i,j}$ is zero. So, the entry is the sum of all nonzero entries in row i of B, Since each edge leaving i is -1 and each edge going to i is 1, we are counting all the edges that either leave or enter i, as we wanted to show.

Now, suppose that our entry is indexed by $[i, j]$ where $i \neq j$. This is the dot product of row i in B with column j in B^T, which is row j in B. So, the entry is equal to

$$\sum_{k=1}^{|E|} b_{i,k} \cdot b_{j,k}$$

Each term in this sum is -1 if k goes between i and j, or 0 if it doesn't. Since we can't have that two different vertices ar both on the same side of an edge,

no terms may ever be 1. So, the entry is just -1 if there is an edge between i and j, and zero otherwise.

Exercise 22.1-8

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.

Exercise 22.2-1

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

Exercise 22.2-2

These are the results when we examine adjacent vertices in lexicographic order:

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

Exercise 22.2-3

As mentioned in the errata, the question should state that we are to show that a single bit suffices by removing line 18. To see why it is valid to remove line 18, consider the possible transitions between colors that can occur. In particular, it is impossible for a white vertex to go straight to black. This is because inorder for a vertex to be colored black, it must of been assigned to u on line 11. This means that we have to of enqueued the vertex in the queue at some point. This can only occur on line 17, however, if we are running line 17 on a vertex, we have to of run line 14 on it, giving it the color GRAY. Then, notice that the only testing of colors that is done anywhere is on line 13, in which we test whiteness. Since line 13 doesn't care if a vertex is GRAY or BLACK, and

we only ever assign black to a gray vertex, we don't affect the running of the algorithm at all by removing line 18. Since, once we remove line 18, we ever assign BLACK to a vertex, we can represent the color by a single bit saying whether the vertex is WHITE or GRAY.

Exercise 22.2-4

If we use an adjacency matrix, for each vertex u we dequeue we'll have to examine all vertices v to decide whether or not v is adjacent to u. This makes the for-loop of line 12 $O(V)$. In a connected graph we enqueue every vertex of the graph, so the worst case runtime becomes $O(V^2)$.

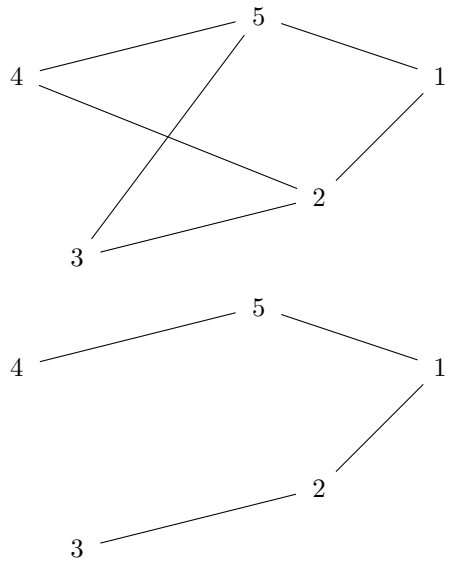
Exercise 22.2-5

First, we will show that the vale d assigned to a vertex is independent of the order that entries appear in adjacency lists. To do this, we rely on theorem 22.5 which proves correctness of BFS. In particular, that we have $\nu.d = \delta(s, \nu)$ at the end of the procedure. Since $\delta(s, \nu)$ is a property of the underlying graph, no matter which representation of the graph in terms of adjacency lists that we choose, this value will not change. Since the d values are equal to this thing that doesn't change when we mess with the adjacency lists, it too doesn't change when we mess with the adjacency lists.

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.

Exercise 22.2-6

Let G be the graph shown in the first picture, $G' = (V, E_\pi)$ be the graph shown in the second picture, and 1 be the source vertex. Let's see why E_π can never be produced by running BFS on G. Suppose that 2 precedes 5 in the adjacency list of 1. We'll dequeue 2 before 5, so $3.\pi$ and $4.\pi$ must both equal 2. However, this is not the case. Thus, 5 must have preceded 2 in the adjacency list. However, this implies that $3.\pi$ and $4.\pi$ both equal 5, which again isn't true. Nonetheless, it is easily seen that the unique simple path in G' from 1 to any vertex is a shortest path in G.



Exercise 22.2-7

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 $O(n+r)$.

Exercise 22.2-8

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. I 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

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

I claim that $d(x,b) = d(s,c) + d(s,b)$. It 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 $O(V)$.

Exercise 22.2-9

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.

Algorithm 1 MAKE-PATH(u)

for v adjacent to u in the original graph, but not in the tree such that $u \leq v$ **do**
    go to v and back to u
end for
for v adjacent to u in the tree, but not equal to $u.\pi$ **do**
    go to v
    perform the path proscribed by MAKE-PATH(v)
end for
go to $u.\pi$

Exercise 22.3-1
For directed graphs:

$from \backslash to$	$BLACK$	$GRAY$	$WHITE$
$BLACK$	$Allkinds$	$Back, Cross$	$Back, Cross$
$GRAY$	$Tree, Forward, Cross$	$Tree, Forward, Back$	$Back, Cross$
$WHITE$	$Cross, Tree, Forward$	$Cross, Back$	$allkinds$

For undirected graphs, note that the lower diagonal is defined by the upper diagonal:

$from \backslash to$	$BLACK$	$GRAY$	$WHITE$
$BLACK$	$Allkinds$	$Allkinds$	$Allkinds$
$GRAY$	–	$Tree, Forward, Back$	$Allkinds$
$WHITE$	–	–	$Allkinds$

Exercise 22.3-2

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

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

The tree edges are: $(q,s),(s,v),(v,w),(q,t),(t,x),(x,z),(t,y),(r,u)$. The back edges are: $(w,s),(y,q),(z,x)$. The forward edge is: (q,w). The cross edges are: $(u,y),(r,y)$.

Exercise 22.3-3

As pointed out in figure 22.5, the parentheses structure of the DFS of figure 22.4 is ((((())()))(()())

Exercise 22.3-4

Treat white vertices as 0 and non-white vertices as 1. Since we never check whether or not a vertex is gray, deleting line 3 doesn't matter. We need only know whether a vertex is white to get the same results.

Exercise 22.3-5

a. Since we have that $u.d < v.d$, we know that we have first explored u before v. This rules out back edges and rules out the possibility that v is on a tree that has been explored before exploring u's tree. Also, since we return from v before returning from u, we know that it can't be on a tree that was explored after exploring u. So, This rules out it being a cross edge. Leaving us with the only possibilities of being a tree edge or forward edge.

   To show the other direction, suppose that (u,v) is a tree or forward edge. In

that case, since v occurs further down the tree from u, we know that we have to explored u before v, this means that $u.d < v.d$. Also, since we have to of finished v before coming back up the tree, we have that $v.f < u.f$. The last inequality to show is that $v.d < v.f$ which is trivial.

b. By similar reasoning to part a, we have that we must have v being an ancestor of u on the DFS tree. This means that the only type of edge that could go from u to v is a back edge.

   To show the other direction, suppose that (u, v) is a back edge. This means that we have that v is above u on the DFS tree. This is the same as the second direction of part a where the roles of u and v are reversed. This means that the inequalities follow for the same reasons.

c. Since we have that $v.f < u.d$, we know that either v is a descendant of u or it comes on some branch that is explored before u. Similarly, since $v.d < u.d$, we either have that u is a descendant of v or it comes on some branch that gets explored before u. Putting these together, we see that it isn't possible for both to be descendants of each other. So, we must have that v comes on a branch before u, So, we have that u is a cross edge.

   To See the other direction, suppose that (u, v) is a cross edge. This means that we have explored v at some point before exploring u, otherwise, we would have taken the edge from u to v when exploring u, which would make the edge either a forward edge or a tree edge. Since we explored v first, and the edge is not a back edge, we must of finished exploring v before starting u, so we have the desired inequalities.

Exercise 22.3-6

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.

Exercise 22.3-7

See the algorithm DFS-STACK(G). Note that by a similar justification to 22.2-3, we may remove line 8 from the original DFS-VISIT algorithm without changing the final result of the program, that is just working with the colors white and gray.

Exercise 22.3-8

Algorithm 2 DFS-STACK(G)

```
for every u ∈ G.V do
    u.color = WHITE
    u.π = NIL
end for
time = 0
S is an empty stack
while there is a white vertex u in G do
    S.push(u)
    while S is nonempty do
        v = S.pop
        time++
        v.d = time
        for all neighbors w of v do
            if w.color == WHITE then
                w.color = GRAY
                w.π = v
                S.push(w)
            end if
        end for
        time++
        v.f = time
    end while
end while
```

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.

Exercise 22.3-9

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 $2.f = 3 < 2 = 2.d$ which provides a counterexample to the given conjecture.

Exercise 22.3-10

We need only update DFS-VISIT. If G is undirected we don't need to make any modifications. We simply note that lines 11 through 16 will never be executed.

Algorithm 3 DFS-VISIT-PRINT(G,u)

```
1:  time = time + 1
2:  u.d = time
3:  u.color = GRAY
4:  for each v ∈ G.Adj[u] do
5:      if v.color == white then
6:          Print "(u, v) is a Tree edge"
7:          v.π = u
8:          DFS-VISIT-PRINT(G, v)
9:      else if v.color == grey then
10:         Print "(u, v) is a Back edge"
11:     else
12:         if v.d > u.d then
13:             Print "(u, v) is a Forward edge"
14:         else
15:             Print "(u, v) is a Cross edge"
16:         end if
17:     end if
18: end for
```

Exercise 22.3-11

Suppose that we have a directed graph on the vertices $\{1,2,3\}$ and having edges $(1,2),(2,3)$ then, 2 has both incoming and outgoing edges. However, if we pick our first root to be 3, that will be in it's 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 wont 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 despite the fact that it has both an incoming and outgoing edge in G.

Exercise 22.3-12

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.

Algorithm 4 DFS-CC(G)

```
1:  for each vertex u ∈ G.V do
2:      u.color = white
3:      u.π = NIL
4:  end for
5:  time = 0
6:  k = 1
7:  for each vertex u ∈ G.V do
8:      if u.color == white then
9:          u.cc = k
10:         k = k + 1
11:         DFS-VISIT-CC(G,u)
12:     end if
13: end for
```

Algorithm 5 DFS-VISIT-CC(G,u)

```
1:  time = time + 1
2:  u.d = time
3:  u.color = GRAY
4:  for each v ∈ G.Adj[u] do
5:      v.cc = u.cc
6:      if v.color == white then
7:          v.π = u
8:          DFS-VISIT-CC(G, v)
9:      end if
10: end for
11: u.color = black
12: time = time + 1
13: u.f = time
```

Exercise 22.3-13

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 $O(|V||E|)$.

Exercise 22.4-1

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 $p, n, o, s, m, r, y, v, x, w, z, u, q, t$.

Exercise 22.4-2

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. To count the number of paths, we can sum the number of paths which leave from each of u's neighbors. Since we have no cycles, we will never risk adding a partially completed number of paths. Moreover, we can never consider the same edge twice among the recursive calls. Therefore, the total number of executions of the for-loop over all recursive calls is $O(V + E)$. Calling SIMPLE-PATHS(s, t) yields the desired result.

Algorithm 6 SIMPLE-PATHS(u,v)

```
1: if u == v then
2:     Return 1
3: else if u.paths ≠ NIL then
4:     Return u.paths
5: else
6:     for each w ∈ Adj[u] do
7:         u.paths = u.paths+ SIMPLE-PATHS(w, v)
8:     end for
9:     Return u.paths
10: end if
```

Exercise 22.4-3

We can't just use a depth first search, since that takes time that could be worst case linear in $|E|$. However we will take great inspiration from DFS, and just terminate early if we end up seeing an edge that goes back to a visited vertex. Then, we should only have to spend a constant amount of time processing each vertex. Suppose we have an acyclic graph, then this algorithm is the usual DFS, however, since it is a forest, we have $|E| \leq |V| - 1$ with equality in the case that it is connected. So, in this case, the runtime of $O(|E| + |V|)$ $O(|V|)$. Now, suppose that the procedure stopped early, this is because it found some edge coming from the currently considered vertex that goes to a vertex that has already been considered. Since all of the edges considered up to this point didn't do that, we know that they formed a forest. So, the number of edges considered is at most the number of vertices considered, which is $O(|V|)$. So, the total runtime is $O(|V|)$.

Exercise 22.4-4

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.

Exercise 22.4-5

Consider having a list for each potential in degree that may occur. We will also make a pointer from each vertex to the list that contains it. The initial construction of this can be done in time $O(|V| + |E|)$ because it only requires computing the in degree of each vertex, which can be done in time

$O(|V| + |E|)$ (see problem 22.1-3). Once we have constructed this sequence of lists, we repeatedly extract any element from the list corresponding to having in degree zero. We spit this out as the next element in the topological sort. Then, for each of the children c of this extracted vertex, we remove it from the list that contains it and insert it into the list of in degree one less. Since a deletion and an insertion in a doubly linked list can be done in constant time, and we only have to do this for each child of each vertex, it only has to be done $|E|$ many times. Since at each step, we are outputting some element of in degree zero with respect to all the vertices that hadn't yet been output, we have successfully output a topological sort, and the total runtime is just $O(|E| + |V|)$. We also know that we can always have that there is some element to extract from the list of in degree 0, because otherwise we would have a cycle somewhere in the graph. To see this, just pick any vertex and traverse edges backwards. You can keep doing this indefinitely because no vertex has in degree zero. However, there are only finitely many vertices, so at some point you would need to find a repeat, which would mean that you have a cycle.

If the graph was not acyclic to begin with, then we will have the problem of having an empty list of vertices of in degree zero at some point. That is, if the vertices left lie on a cycle, then none of them will have in degree zero.

Exercise 22.5-1

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.

Exercise 22.5-2

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 $s - w - v$.

Exercise 22.5-3

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.

Exercise 22.5-4

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 $(G^T)^T = G$.

Exercise 22.5-5

Given the procedure given in the section, we can compute the set of vertices in each of the strongly connected components. For each vertex, we will give it an entry SCC, so that $v.SCC$ denotes the strongly connected component (vertex in the component graph) that v belongs to. Then, for each edge (u, v) in the original graph, we add an edge from $u.SCC$ to $v.SCC$ if one does not already exist. This whole process only takes a time of $O(|V| + |E|)$. This is because the procedure from this section only takes that much time. Then, from that point, we just need a constant amount of work checking the existence of an edge in the component graph, and adding one if need be.

Exercise 22.5-6

By Exercise 22.5-5 we can compute the component graph in $O(V + E)$ time, and we may as well label each node with its component as we go (see exercise 22.3-12 for the specifics), as well as creating a list for each component which contains the vertices in that component by forming an array A such that $A[i]$ contains a list of the vertices in the i^{th} connected component. Then run DFS again, and for each edge encountered, check whether or not it connects two different components. If it doesn't, delete it. If it does, determine whether it is the first edge connecting them. If not, delete it. This can be done in constant time per edge since we can store the component edge information in a k by k matrix, where k is the number of connected components. The runtime of this is thus $O(V + E)$. Now the only edges we have are a minimal number which connect distinct connected components. The last step is place edges within the connected components in a minimal way. The fewest edges which can be used to create a connected component with n vertices is n, and this is done with a

cycle. For each connected component, let $v_1, v_2, \ldots, v_k$ be the vertices in that component. We find these by using the array A created earlier. Add in the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)$. This is linear in the number of vertices, so the total runtime is $O(V + E)$.

Exercise 22.5-7

First compute the component graph as in 22.5-5. Then, in order to have that every vertex either has a path to or from every other vertex, we need that this component graph also has this property. Since this is acyclic, we can perform a topological sort on it. For this to be the case, we want that there is a single path through this dag that hits every single vertex. This can only happen in the DAG if each vertex has an edge going to the vertex that appears next in the topological ordering. See the algorithm IS-SEMI-CONNECTED(G).

Algorithm 7 IS-SEMI-CONNECTED(G)

```
Compute the component graph of G, call it G'
Perform a topological sort on G' to get the ordering of its vertices
v_1, v_2, ..., v_k.
for i=1..k-1 do
    if there is no edge from v_i to v_{i+1} then
        return FALSE
    end if
end for
return TRUE
```

Problem 22-1

a) 1. If 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. It 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. It 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. It 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 $u.d > v.d$.

   In fact, we just showed that we have the stronger conclusion, that $0 \leq v.d < u.d$.

Problem 22-2

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. If 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 $v.low$ are ones which go from a descendant of v to a proper ancestor of v. If we know $u.low$ for every child u of v, then we can compute $v.low$ easily since all the information is coded in its descendants. Thus, we can write the algorithm recursively: If v is a leaf in G_π then $v.low$ is the minimum of $v.d$ and $w.d$ where (v, w) is a back edge. If v is not a leaf, v is the minimum of $v.d$, $w.d$ where w is a back edge, and $u.low$, where u is a child of v. Computing $v.low$ 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 $v.low$ for all $v \in V$. If $v.low = v.d$ 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 $v.low$ 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. 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. It 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 $a, b, p_1, \ldots, p_k, u, q_m, \ldots, q_1, d, c, q_l, \ldots, q_{m+1}, v, p_{k+1}, \ldots, p_n, a$ 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.

Algorithm 8 BCC(G)

```
1: for each vertex u ∈ G.V do
2:     u.color = white
3: end for
4: k = 1
5: for each vertex u ∈ G.V do
6:     if u.color == white then
7:         k = k + 1
8:         VISIT-BCC(G,u, k)
9:     end if
10: end for
```

Problem 22-3

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 wlog that in-degree - out-degree = a ¿ 0. Note that we

Algorithm 9 VISIT-BCC(G,u,k)

```
1: u.color = GRAY
2: for each v ∈ G.Adj[u] do
3:     (u, v).bcc = k
4:     if v.color == white then
5:         VISIT-BCC(G, v, k)
6:     end if
7: end for
```

may assume that in degree is greater because otherwise we would just look at the transpose graph in which we traverse the cycle backwards. If 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. If 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, EULER-TOUR(G) which takes time $O(|E|)$. It 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.

Problem 22-4

Begin by locating the element v of minimal label. We would like to make $u.min = v.label$ for all u such that $u \rightsquigarrow v$. Equivalently, this is the set of ver-

Algorithm 10 EULER-TOUR(G)

```
color all edges white
let (v, u) be any edge
let L be a list containing just v.
while there is some white edge (v, w) coming out of v do
    color (v,w) black
    v = w
    append v to L
end while
```

tices u which are reachable from v in G^T. We can implement the algorithm as follows, assuming that $u.min$ is initially set equal to NIL for all vertices $u \in V$, and simply call the algorithm on G^T.

Algorithm 11 REACHABILITY(G)

```
1: Use counting sort to sort the vertices by label from smallest to largest
2: for each vertex u ∈ V do
3:     if u.min == NIL then
4:         REACHABILITY-VISIT(u, u.label)
5:     end if
6: end for
```

Algorithm 12 REACHABILITY-VISIT(u, k)

```
1: u.min = k
2: for v ∈ G.Adj[u] do
3:     if v.min == NIL then
4:         REACHABILITY-VISIT(v, k)
5:     end if
6: end for
```