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 $\text{in-degree } |V| - 1$ and $\text{out-degree } 0$ $-$ 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 $u.\pi = t$.

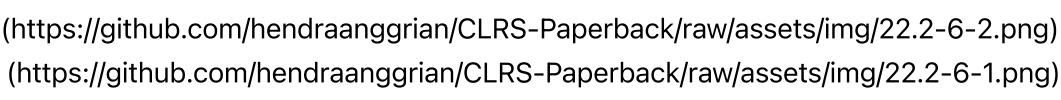
Now, suppose instead that we had x preceding t in the adjacency list of w. Then, it would get added to the queue before t, which means that it would u as it's child before we have a chance to process the children of t. This will mean that $u.\pi = x$ in this different ordering of the adjacency list for 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 $Adj[s]$. 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 $Adj[s]$. 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 $\text{MAKE-PATH}(s)$, 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 $\text{in-degree } 0$. 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 $\{v[1], v[2], \dots, v[k-1]\}$. 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'. If 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. If 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. 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 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
>
> $$v.\,low = \min \begin{cases} v.\,d, \\ w.\,d : (u, w) \text{ is a back edge for some descendant } u \text{ of } v. \end{cases}$$
>
> Show how to computer $v.\,low$ 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 $e.\,bcc$ such that $e.\,bcc = e'.\,bcc$ 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. 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 (v, 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. 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. 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

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

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. 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, $\text{EULER-TOUR}(G)$ 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.

```
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. Its 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 $\min(u.\,scc)$ 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. In the DFS-VISIT subroutine, upon discovering a new node, we set its $\min$ to be the label of

its root.