Chapter 22: Elementary Graph Algorithms.

Definitions:

1. \( G = (V, E) \), where \( V \) is a set of points and \( E \) is a set of edges connecting point-pairs from \( V \).

2. We use \( V \) synonymously with \(|V|\) and \( E \) with \(|E|\) when the context is clear.

3. Adjacency list representation: randomly addressable vector \( V \), with attributes as needed in an application, e.g. \( v.color \), \( v.cnt \). One attribute is \( v.adj \), which holds a pointer to a linked list of edges leaving vertex \( v \).

4. Adjacency list structure can be scanned in \( \Theta(V + E) \) time.
5. Adjacency matrix representation: $V \times V$ matrix $M$ with $M_{ij}$ holding information about the edge, if any, between vertices $i$ and $j$. Typically, vertices are represented as 1, 2, 3, ..., and we need a parallel structure to hold vertex attributes. Matrix cells can hold edge attributes.

6. Adjacency matrix structure can be scanned in $\Theta(V^2)$ time.

7. $E = o(V^2)$ implies the list structure is more efficient; $E = \Theta(V^2)$ implies the matrix structure is equally efficient.
Algorithm for calculating the out-degree of each vertex, given an adjacency list representation.

Get-out-degree(V) {
    for u ∈ V loop {
        u.out-degree = 0
        for v ∈ u.adj loop {
            u.out-degree = u.out-degree + 1
        }
    }
}

Since $E = \sum_{i=1}^{V} n_i$, we have $T = 3V + 2E + 1 = \Theta(V + E)$.

For a graph $g$, code will look more like...

Vertex[ ] u = g.vertices; // returns vertex vector
Edge e = null;

for (int i = 0; i < u.length; i++) {
    u[i].out-degree = 0;
    e = u.adj;
    while (e! = null) {
        u.out-degree = u.out-degree + 1;
        e = e.next;
    }
}
Algorithm for calculating the in-degree of each vertex, given an adjacency list representation.

Get-in-degree(V) {
    for u ∈ V loop
        u.in-degree = 0
    for u ∈ V loop
        for v ∈ u.adj loop
            v.in-degree = v.in-degree + 1
        ∑_{i=1}^{V} (n_i + 1), where n_i is the edge count of u
    ∑_{i=1}^{V} n_i
}

Since \( E = ∑_{i=1}^{V} n_i \), we have \( T = 4V + 2E + 2 = Θ(V + E) \).
Algorithm for calculating the transpose graph, given an adjacency list representation.

Get-transpose(V) {
    for u ∈ V loop 
        u.trans-adj = null
    for u ∈ V loop
        for v ∈ u.adj loop
            add u to v.trans-adj

Since \( E = \sum_{i=1}^{V} n_i \), we have \( T = 4V + 2E + 2 = \Theta(V + E) \).
Algorithm for eliminating self-loops and multiple edges, given an adjacency list representation.

Get-reduction(V) {
    for u ∈ V loop {
        u.reduced-adj = null; V + 1
        u.mark = null; V
    }
    for u ∈ V loop V + 1
        for v ∈ u.adj loop \[ \sum_{i=1}^{V} (n_i + 1) \]
            if (v \neq u) and (v.mark \neq u) \{ \[ \sum_{i=1}^{V} n_i \]
                add v to u.reduced-adj; 0 to \[ 2 \sum_{i=1}^{V} n_i \]
                v.mark = u;
            \}
    }
}

Observations:

1. Failure of the if-test implies the two following lines are not executed. Success implies they are both executed. Hence the count for the pair is zero in the best-case and \[ 2 \sum_{i=1}^{V} n_i \] in the worst-case.

2. Since \[ \sum_{i=1}^{V} n_i = E \], we have \[ 5V + 2E + 2 \leq T \leq 5V + 4E + 2 \], implying \[ T = \Theta(V + E) \].

3. When the first vertex starts processing as u, all v.mark values are null. As processing continues for u these v.mark values remain either null or u, depending on whether edge (u, v) has been seen. When the second vertex starts processing as u, all v.mark values are either null or the previous vertex. During the processing of this second u, the mark values may be null, or the first vertex, or the second vertex. And so forth. Hence we never need to reset the mark attribute for subsequent values u values in the outer loop.
Algorithm for calculating $G^2$, given an adjacency list representation. If $G = (V, E)$, $G^2 = (V, E^2)$, where $(u, v) \in E^2$ if and only if (a) $(u, v) \in E$ or (b) there exist a vertex $w$ with $(u, w) \in E$ and $(w, v) \in E$. (First step in constructing the transitive closure)

Get-transitive($V$) {
    for $u \in V$ loop {
        $u$.trans-adj = null;
        for $v \in u$.adj loop {
            add $v$ to $u$.trans-adj;
            for $w \in v$.adj loop
                add $w$ to $u$.trans-adj;
        }
    }
    run Get-reduction on $G$, using the trans-adj edges, to remove duplicate edges
}

Observations:

1. Third-level loop is initiated for each $(u, v)$ edge. The max-count occurs if destination vertex $v$ is connected to all of $V$.
2. Last operation to remove duplicate edges is linear: $\Theta(V + E)$.
3. Complexity, worst case,

$$T = 3V + 2EV + 3E + 1 + \Theta(V + E) = 2EV + \Theta(V + E) = \begin{cases} \Theta(V^3), & E = \Theta(V^2) \\ \Theta(V^2), & E = \Theta(V). \end{cases}$$
Compare adjacency matrix solution. \([a_{ij}]\) is the given adjacency matrix; \([b_{ij}]\) is matrix of transitive augmentation. Assume graph has \(n\) vertices.

Get-transitive\((a, b)\) {
    \[\text{for} \; i = 1 \; \text{to} \; n \; \text{loop} \;
    \text{for} \; j = 1 \; \text{to} \; n \; \text{loop} \; \;
    \text{if} \; a_{ij} = 1 \;
    b_{ij} = 1; \;
    \text{else} \; \{
    b_{ij} = 0; \;
    \text{for} \; k = 1 \; \text{to} \; n \; \text{loop} \;
    \text{if} \; a_{ik}a_{kj} == 1 \; \{
    b_{ij} = 1; \;
    \text{exit} \; k\text{-loop}; \;
    \}
    \}
    \]
}\}

The first if-statement executes \(V^2\) times. In a shortest execution, the statement will always succeed, adding \(2V^2\) to the \(2V^2 + 2V + 1\) noted in the code. In a longest execution, the statement will always fail, the subsequent if-statement always succeeds, and the exit will not be taken, adding \(V^2[2 + (V + 1) + 3V]\) to the \(2V^2 + 2V + 1\) noted in the code.

\[
\begin{align*}
T(n) & \geq 2V^2 + 2V + 1 + 2V^2 = 4V^2 + 2V + 1 \\
T(n) & \leq 2V^2 + 2V + 1 + 2V^2 + V^3 + V^2 + 3V^3 = 4V^3 + 5V^2 + 2V + 1 \\
T(n) & = \Omega(V^2) \\
T(n) & = O(V^3),
\end{align*}
\]

which is comparable to the adjacency list approach.
Breadth-first Search (BFS)

```plaintext
BFS(G, s) // G = (V, E), s ∈ V 
   { 
      for v ∈ V\{s} loop 
         v.color = white; 
         v.d = ∞; 
         v.π = null 
      } 
      s.color = gray; 
      s.d = 0; 
      s.π = null; 
      Q = φ; 
      Q.enqueue(s); 
      while Q ≠ φ loop 
         u = Q.dequeue(); 
         for v ∈ u.adj loop 
            if v.color = white 
               v.color = gray; 
               v.d = u.d + 1; 
               v.π = u; 
               Q.enqueue(v); 
            } 
         u.color = black; 
      } 
```

Observations:

1. Setup: $\Theta(V)$
   
   While-loop: Each $v ∈ V$ is enqueued at most once and assigned gray just prior to enqueue operation. Also no vertex ever reverts to white, which implies $O(V)$ while-loop iterations. Each iteration processes a disjoint set of edge links. Consequently, the while-loop activity is $O(V + E)$. BFS is then $O(V + E)$.

2. Cannot guarantee $\Omega(V + E)$ as the component containing $s$ might remain small as the graph size expands toward infinity, which would cause $(\text{instruction count})/(V + E) → 0$ as size increases.
**Definition:** \( \delta(s, v) \) is the shortest number of links in a path from \( s \) to \( v \), if such a path exists. Otherwise, \( \delta(s, v) = \infty \).

**Lemma 22.1 (triangle inequality):** \((u, v) \in E \Rightarrow \delta(s, v) \leq \delta(s, u) + 1.\)

Proof: If \( \delta(s, u) = \infty \), desired result is certainly true. Otherwise, any path from \( s \) to \( u \), including the shortest one of length \( \delta(s, u) \), followed by edge \((u, v)\), is a competitor for \( \delta(s, v) \), which implies the desired inequality. ■
We assume an adjacency list representation for the undirected graph to the left. The numbers specify the order in which edges appear on the various edge lists.

The vertex queue evolves as follows. Each vertex is represented a column vector. The top position records the vertex; the middle position records the \(d\) value; the final position records the \(\pi\) value (\(\phi\) means null).

\[
\begin{array}{cccc|cccc|cccc}
0 & s & w & r & s & w & r & t & x & s & w & r & t & x & v & s & w & r & t & x & v & u \\
\phi & \phi & s & s & \phi & s & s & w & w & \phi & s & s & w & w & r & \phi & s & s & w & w & r & t \\
0 & 1 & 1 & \Rightarrow & 0 & 1 & 1 & 2 & 2 & \Rightarrow & 0 & 1 & 1 & 2 & 2 & 3 \\
\phi & s & s & \phi & s & s & w & w & \phi & s & s & w & w & r & \phi & s & s & w & w & r & t \\
\Rightarrow & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & \Rightarrow & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\
\phi & s & s & w & w & r & t & x & \phi & s & s & w & w & r & t & x \\
\Rightarrow & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & \Rightarrow & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\
\phi & s & s & w & w & r & t & x & \phi & s & s & w & w & r & t & x \\
\end{array}
\]

The queue content is the segment between the two vertical lines.

left of the first vertical line (black) \(\Rightarrow\) removed from the queue

between the lines (gray) \(\Rightarrow\) evolving as the last removed item adds vertices from its adjacency list

Process assigns a \(\pi\) value to each vertex, except \(s\), that identifies the previous vertex leading to its discovery. We use these ancestor marks to evolve the following \(G_\pi\) tree.
\[
\begin{array}{ccccccc}
& r & 1 & 2 & s & 1 & t & 3 & 1 & u & 1 \\
& 2 & 1 & 2 & 2 & 2 & 3 & 3 \\
v & w & x & x & y & y \\
\end{array}
\]
Lemma 22.2: $G = (V, E)$, directed or undirected. On termination of BFS($G, s$), $v.d \geq \delta(s, v)$ for all $v \in V$.

Proof: (Via induction on the number of enqueue operations)

We note that $v.d$ is changed at most once for each $v \in V$. For nodes that do not pass through the queue, $v.d = \infty$, its initial value, at all times during the algorithm execution. Hence, for these nodes $v.d = \infty \geq \delta(v, s)$ holds at the conclusion of the algorithm.

For all other vertices, we note that its $d$ attribute is changed just before it is enqueued. As a base case, the first such enqueue operation places $s$ in the queue. At that point, we have $s.d = 0 = \delta(s, s)$. At this time, $s.d = 0 = \delta(s, s)$, and $\delta(s, v) = \infty$ for all other vertices $v$. Hence the inequality holds after the first enqueue operation.

Consider now a later enqueue of node $v$. The $v.d$ value is set just before it enters the queue. Specifically, $v.d = u.d + 1$ for a neighbor $u$ that has already passed through the queue. By induction, we can assume $u.d \geq \delta(u, s)$. So,

$$v.d = u.d + 1 \geq \delta(s, u) + 1 \geq \delta(s, v),$$

the last equality following because $(u, v) \in E$ and the triangle inequality applies. ■
**Lemma 22.3**: When the queue contains \( v_1, v_2, \ldots, v_r \), with \( v_1 \) being at the front (next to dequeue), then \( v_1.d \leq v_2.d \leq \ldots \leq v_r.d \leq v_1.d + 1 \).

Proof: The text proceeds via induction on the number of queue operations. Specifically, at the beginning of the while-loop, the queue contains only \( s \), and the inequality trivially holds.

Then, we change the queue only by dequeue and enqueue operations. We note that a dequeue operation cannot change the inequality.

Enqueue operations occur as the adjacency list of the last dequeued vertex, say \( u \), is examined. Just before \( u \) is dequeued, the induction hypothesis ensures that the \( \ast.d \) values in the queue must look like

\[
u.d, u.d, \ldots, u.d \quad \text{or} \quad u.d, u.d, \ldots, u.d + 1, u.d + 1, \ldots, u.d + 1.
\]

Any enqueues that occur as the adjacency list of \( u \) is explored place vertices at the back of the queue with \( u.d + 1 \) distance values. In either case, the inequality is preserved. ■

**Corollary 22.4**: If \( v_i \) is enqueued before \( v_j \), then \( v_i.d \leq v_j.d \).

Proof: From the above, vertices added to the end of the queue are greater than or equal to all vertices in the queue. Any vertices that have already passed through the queue are smaller yet, or perhaps just no larger. ■
**Theorem 22.5** BFS Correctness. \( G = (V, E) \) is directed or undirected, \( s \in V \). Then

1. BFS\((G, s)\) discovers each \( v \in V \) that is reachable from \( s \) and, on termination, 
\[ v.d = \delta(s, v) \] for all \( v \in V \).

2. If \( v \neq s \) is reachable from \( s \), then one shortest path (number of links) from \( s \) to \( v \) is a shortest path from \( s \) to \( v.\pi \) followed by the link \((v.\pi, v)\).

Proof: Let \( A = \{v \in V : v.d > \delta(s, v) \text{ at termination}\} \). Since Lemma 22.2 states that \( v.d \geq \delta(s, v) \) for all \( v \in V \), then \( A = \phi \) implies \( v.d = \delta(s, v) \) for all \( v \in V \).

We proceed by contradiction: suppose \( A \neq \phi \).

Choose \( v \in A \) with \( \delta(s, v) \leq \delta(s, w) \) for all \( w \in A \).

Note: \( v \neq s \) because \( s.d = 0 = \delta(s, s) \) is established at setup and never changes, which implies \( s \notin A \).

Note: \( v \) is reachable from \( s \).

Why? Suppose \( v \) is not reachable from \( s \), then \( \delta(s, v) = \infty \) by definition, which implies \( v.d > \delta(s, v) \) cannot hold, which implies \( v \notin A \) — a contradiction. So, \( \delta(s, v) < \infty \).

Now, \( v \) reachable from \( s \) means there exists a path from \( s \) to \( v \). Hence, there exists a shortest path from \( s \) to \( v \). Let \( u \) be the vertex just before \( v \) on a shortest path from \( s \) to \( v \). That is, \((u, v) \in E\).

\( \delta(s, v) = \delta(s, u) + 1 \). That is, the link count to \( u \) from \( s \) must be minimal or else we are not on a shortest path from \( s \) to \( v \).

\( \delta(s, u) = \delta(s, v) - 1 < \delta(s, v) \) (since \( \delta(s, v) < \infty \)), which implies, by our selection of \( v \), that \( u \notin A \).

Therefore \( u.d = \delta(s, u) < \infty \), the last because \( u \) is reachable from \( s \). Then, since \( v \in A \),

\[ (*) \quad v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1 \]
(**) \( v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1 \)

When BFS dequeues \( u, v \) is white, or gray, or black.

If \( v \) is white: \( v.d = u.d + 1 \) executes, contradicting (**).

If \( v \) is black, then \( v \) was dequeued earlier. Corollary 22.4 then forces \( v.d \leq u.d \), again contradicting (**).

If \( v \) is gray, then \( v \) is still in the queue from which \( u \) has just departed. As the maximal \(*.d\) value in the queue was \( u.d + 1 \) before \( u \) departed and because any new vertices enqueued as \( u \) scans its adjacency list would have \(*.d = u.d + 1\), we conclude that \( v.d \leq u.d + 1 \), again violating (**).

This contradiction implies \( A = \phi \) and \( v.d = \delta(s,v) \) for all \( v \in V \) at termination.

Also, if \( v \) is reachable from \( s \) and not discovered by BFS (that is, passed through the queue), then \( v.d = \infty > \delta(s,v) \), a contradiction. Therefore, BFS discovers all reachable \( v \in V \).

Finally, if \( v.\pi = u \), then \( v \) was enqueued while scanning the adjacency list of \( u \), which implies \( (u,v) \in E \) and \( v.d = u.d + 1 \), which gives \( \delta(s,v) = \delta(s,u) + 1 \). Therefore, one shortest path from \( s \) to \( v \) is a shortest path from \( s \) to \( u \), following by the link \( (u,v) = (v.\pi,v) \).

Note \((v.\pi,v)\) links form a tree, since each added vertex is white when discovered. These trees are called \( G_\pi \)-trees. A shortest path to a vertex \( v \) can be obtained in reverse as

\[
v, u_1 = v.\pi, u_2 = u_1.\pi, u_3 = u_2.\pi, \ldots, s = u_n.\pi.
\]
Depth-first Search (DFS)

\[
\text{DFS}(G) \quad // \quad G = (V, E) \{
\quad \text{for } v \in V \text{ loop}
\quad \quad v.\text{color} = \text{white};
\quad \quad v.\pi = \text{null}
\quad \}
\quad \text{time} = 0;
\quad \text{for } v \in V \text{ loop }\{
\quad \quad \text{if } v.\text{color} = \text{white}
\quad \quad \quad \text{DFS-Visit}(G, v);
\quad \}
\}
\]

\[
\text{DFS-Visit}(G, v) \{
\quad \text{time} = \text{time} + 1;
\quad v.d = \text{time};
\quad v.\text{color} = \text{gray};
\quad \text{for } u \in v.\text{adj} \text{ loop}
\quad \quad \text{if } u.\text{color} = \text{white} \{
\quad \quad \quad u.\pi = v;
\quad \quad \quad \text{DFS-Visit}(G, u);
\quad \quad \}
\quad \quad v.\text{color} = \text{black};
\quad \text{time} = \text{time} + 1;
\quad v.f = \text{time};
\}
\]

Note:

DFS is \(\Theta(V)\), exclusive of work done in DFS-Visit routines.

DFS is called on behalf of each \(v \in V\) exactly once, implying each adjacency list is scanned once, implying \(\Theta(E)\).

DFS is then \(\Theta(V + E)\).
Start-finish intervals and predecessor evolution.
$G_\pi$ trees and interval nestings.
Definitions: $e = (u, v) \in E$ is a

1. tree edge if $v$ is first discovered via $e = (u, v)$. That is, $v$ is white when it is unpacked from the $u$.adj. Note that vertices connected by tree edges form trees because each edge extends to a white (heretofore undiscovered) vertex and therefore cannot produce a cycle. These $G_\pi$ trees are called \textbf{depth-first trees}.

2. back edge if $v$ is an ancestor of $u$ in a depth-first tree.

3. forward edge if $e = (u, v)$ is not a tree edge and $v$ is a descendant of $u$ in a depth-first tree.

4. cross edge if $e$ is not a tree, back, or forward edge. These edges can connect vertices in the same tree that have no ancestor-descendant relationship between them, or it can connect vertices in different depth-first trees.
Theorem 22.7 (Parenthesis Theorem): $G = (V, E)$ is a directed or undirected graph scanned by DFS. For distinct $u, v \in V$, exactly one of the following holds

1. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint and $u, v$ have no ancestor-descendant relationship in a $G_\pi$ tree.
2. $[u.d, u.f] \subset [v.d, v.f]$ and $u$ is a descendant of $v$ in a $G_\pi$ tree.
3. $[v.d, v.f] \subset [u.d, u.f]$ and $v$ is a descendant of $u$ in a $G_\pi$ tree.

Note:

1. This theorem asserts that $[u.d, u.f]$ and $[v.d, v.f]$ cannot partially intersect. They are either disjoint or one is a subset of the other.
2. There are no repetitions among the \{u_1.d, u_2.d, \ldots, u_n.d, u_1.f, u_2.f, \ldots, u_n.f\}. The time counter is incremented before every assignment.
3. The set of discovery and finish times is \{1, 2, 3, \ldots, 2n\} for a graph with $n$ vertices.
Proof: Suppose $u.d < v.d$. Where is $u.f$?

1. $u.d < u.f < v.d < v.f$
2. $v$ discovered after $u$ is black.
3. $v$ discovered after stack-frame for $u$ has vanished.
4. $v$ discovered after all descendants of $u$ have been discovered.
5. $v$ is not a $G_\pi$ descendant of $u$.
6. Also, $u$ discovered before $v$ implies $u$ is not white when the scan starts exploring descendants of $v$.
7. So, $u$ is not a $G_\pi$ descendant of $v$.
8. Case (1) holds.

In a parallel fashion, we obtain either Case (1) or Case (2) when $v.d < u.d$, depending on whether $v.f$ occurs before or after $u.d$. □
**Corollary 22.8** \( v \) is a proper descendant of \( u \) in a DFS \( G_\pi \) tree if and only if \( u.d < v.d < v.f < u.f \).

**Theorem 22.9** (White-path theorem): \( G = (V,E) \) is a directed or undirected graph scanned by DFS. \( u,v \in V \). Then \( v \) is a descendant of \( u \) in a \( G_\pi \) tree if and only if, when \( u.d \) is assigned, there exists a path from \( u \) to \( v \) consisting entirely of white nodes, including the endpoints \( u \) and \( v \).

Proof: \((\Rightarrow)\) Assume \( v \) is a \( G_\pi \) descendant of \( u \). If \( v = u \), the path from \( u \) to \( v \) contains only \( u \), which is white when the assignment \( u.d = \) time is executed (see code).

If \( v \neq u \), then nodes, say \( w \), along the path from \( u \) to \( v \), including \( v \) itself, but excluding \( u \), are proper \( G_\pi \) descendants of \( u \). From Corollary 22.8, \( u.d < w.d \), which implies that \( w \) is white when \( u.d \) is assigned. The nodes \( w \) then constitute a white path from \( u \) to \( v \).

\((\Leftarrow)\) Assume that there exists a white path from \( u \) to \( v \) at the time \( u.d \) is assigned. For purposes of deriving a contradiction, suppose that \( v \) is not a \( G_\pi \) descendant of \( u \) in the \( G_\pi \) tree containing \( u \).

Let \( w \) be the first vertex on the white path from \( u \) to \( v \) that is not a descendant of \( u \). Then \( w \neq u \), since \( u \) is a descendant of itself. Consequently, there exists a predecessor of \( w \), say \( x \), on the path that must be a \( G_\pi \) descendant of \( u \). \( x \) could be \( u \) itself. We then have \( u.d \leq x.d < x.f \leq u.f \), since the parenthesis theorem says that descendants finish within their parents. But, \( w \) is on the adjacency list of \( x \), since the link \((x,w) \in E\). Therefore, since \( w \) is white, \( w \) will be discovered between \( x.d \) and \( x.f \).

That is, \( u.d \leq x.d < w.d < x.f \leq u.f \). However, since start-finish intervals cannot partially overlap, we must actually have \( u.d < x.d < w.d < w.f < x.f < u.f \), which implies \( w \) is a \( G_\pi \) descendant of \( u \), again by the parenthesis theorem. This contradiction concludes the proof. \( \blacksquare \)
Dynamic Edge Classification.

When DFS explores edge \((u, v)\) (that is, unpacks \(v\) from the adjacency list of \(u\)), we have

1. \(v\) white \(\Rightarrow\) \((u, v)\) is a tree edge
2. \(v\) gray \(\Rightarrow\) \((u, v)\) is a back edge
3. \(v\) black \(\Rightarrow\) \((u, v)\) is a forward, if \(u.d < v.d\), or cross edge, if \(u.d > v.d\).

If \(v\) is white, then \(v.\pi = u\) is executed, laying in a \(G_\pi\) tree edge.

If \(v\) is gray, then the activation record (stack frame) for the DFS-Visit(G, \(v\)) is still on the stack. At the top of the stack is the activation record for \(u\) as we are going through the adjacency list of \(u\) when we unpack \(v\). That is, the stack frames from the one that is unpacking the adjacency list of \(v\) up through the top-of-stack that is unpacking the adjacency list for \(u\) represent a chain of vertices,

\[v, x_1 \in v.\text{adj}, x_2 \in x_1.\text{adj}, \ldots, x_n \in x_{n-1}.\text{adj}, u \in x_n.\text{adj}\]

where \((v, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n), (x_n, u)\) are all tree edges. Hence \(v\) is an ancestor of \(u\) in a \(G_\pi\) tree.

Finally, if \(v\) is black, it is neither a tree edge or a back edge, and therefore forward or cross edge are the remaining possibilities. By the parenthesis theorem, if \((u, v)\) is a forward edge, then \(v\) is a descendant of \(u\), which implies \(u.d < v.d < v.f < u.f\). If \((u, v)\) is a cross edge, then there is no ancestor-descendant relationship between
u and v. The intervals \([v.d, v.f]\) and \([u.d, u.f]\) are then disjoint. Consequently, \(v.d\) cannot occur within the \([u.d, u.f]\) span — if it did occur there then it would finish there contradicting the disjoint nature of the intervals. So, \([v.d, v.f]\) occurs before \([u.d, u.f]\).

We can distinguish between forward and cross edges when \(v\) is black, choosing forward if \(u.d < v.d\) and cross if \(v.d < u.d\).

**Theorem 22.10**: DFS on an undirected graph finds only tree edges and back edges.

Proof: Consider \((u, v), (v, u) \in E\). Wolog \(u.d < v.d\). Then \(v\) on the adjacency list of \(u\) implies \(u.d < v.d < v.f < u.f\). However, the edge \((u, v)\) or the edge \((v, u)\) could have been explored first. The latter case occurs if some \(w\) earlier than \(v\) on the adjacency list of \(u\) leads to \(v\) via a sequence of tree edges. In processing \(v\), we find the edge \((v, u)\).

If this second case occurs, then \((u, v)\) is a back edge. Otherwise, \(v\) is white when unpacked from the adjacency list of \(u\), implying that \((u, v)\) is a tree edge. ■
Topological Sort of a directed acyclic graph.

Tentative algorithm: Does the example generalize?
Lemma 22.11: A directed graph is acyclic if and only if DFS finds no back edges.

Proof: (⇒) Suppose \( G = (V, E) \) is acyclic. For purposes of deriving a contradiction, suppose \((u, v)\) is a back edge. That is, \( v \) is an ancestor of \( u \) in a \( G_\pi \) tree. Then, there is a sequence of tree edges connecting \( v \to w_1 \to w_2 \to \ldots \to w_n \to u \). The edge \((u, v)\) then completes a cycle — a contradiction. We conclude that no back edges can be found.

(⇐) Suppose DFS finds no back edges. For purposes of deriving a contradiction, suppose there exists a cycle \( v_1 \to v_2 \to \ldots \to v_n \to v_1 \).

Let \( v_i \) be the first vertex of the cycle discovered by DFS. Let \( w \) be the previous vertex on the cycle; \( w = v_n \) if \( i = 1 \), otherwise \( w = v_{i-1} \). In any case, a link in the cycle connects \( w \) to \( v_i \). By the white-path theorem, at time \( v_i.d \), the cycle provides a white path to \( w \), which implies that \( w \) is a descendant of \( v_i \) in a \( G_\pi \) tree. Hence we have tree edges forming the path \( v_i \to x_1 \to x_2 \to \ldots \to w \). While exploring the adjacency list of \( w \), DFS finds \( v_i \) to be gray. Hence \((w, v_i)\) is a back edge — a contradiction. We conclude that no back edges implies no cycles. \( \blacksquare \)
Lemma 22.12 TopoSort works.

Proof: Need \( u \neq v, (u, v) \in E \Rightarrow u.f > v.f \).

Consider the time when \( v \) is unpacked from \( u.\text{adj} \). If \( v \) is gray, then \((u, v)\) is a back edge. This scenario is not possible because we assume that the input to TopoSort is acyclic.

Hence \( v \) is white or black.

\( v \) white \( \Rightarrow (u, v) \) is a tree edge \( \Rightarrow v \) is a descendant of \( u \) \( \Rightarrow u.d < v.d < v.f < u.f \) via the parenthesis theorem. Note \( u.f > v.f \) as desired.

\( v \) black (and \( u \) gray since we are still exploring its adjacency list) \( \Rightarrow v.f \) is assigned but \( u.f \) has not yet been assigned \( \Rightarrow u.f > v.f \) as desired. \( \blacksquare \)
**Strongly connected components** (Shamir algorithm)

Components in input graph:

![Graph Diagram]

Upon entering a component for the first time, say at \( u \), there is a white path to every node in the component, which implies all are descendents of \( u \) in a \( G_\pi \) tree. So, all are visited before exit via returning to the stack frame that entered the component.

\( E \) finishes first, then \( D \), then \( C \). New start finishes \( A \). Final start finishes \( B \). Decreasing finish times: \( BACDE \).

Components in transpose graph: Process in order \( BACDE \) from DFS driver.

![Graph Diagram]

Note that a new SCC is found whenever control returns to the driver.
Tentative algorithm.

SCC(G) // G = (V, E) {
    vertexList = null;
    for v ∈ V loop
        v.color = white;
        v.π = null
    }
    time = 0;
    for v ∈ V loop {
        if v.color = white
            DFS-Visit1(G, v);
    }
    compute G-transpose;
    for v ∈ V loop
        v.color = white;
        v.π = null
    }
    time = 0;
    for v ∈ vertexList loop {
        time = time + 1;
        if v.color = white
            DFS-Visit2(G, v);
    }
}

DFS-Visit1(G, v) {
    time = time + 1;
    v.d = time;
    v.color = gray;
    for u ∈ v.adj loop
        if u.color = white {
            u.π = v;
            DFS-Visit1(G, u);
        }
    v.color = black;
    time = time + 1;
    v.f = time;
    add v to front of vertexList;
}

DFS-Visit2(G, v) {
    v.scc = time;
    v.color = gray;
    for u ∈ v.trans-adj loop
        if u.color = white {
            u.π = v;
            DFS-Visit2(G, u);
        }
    v.color = black;
}

Time remains Θ(V + E).
**Definition**: Let $U$ be a set of vertices. Then, for a given DFS scan, let

\[
U.d = \min \{ u.d : u \in U \} \\
U.f = \max \{ u.f : u \in U \}.
\]

**Lemma 22.14**: Let $C, C'$ be strongly connected components in directed graph $G = (V, E)$. Suppose there exists $(u, v) \in E$ with $u \in C$ and $v \in C'$. Then $C'.f < C.f$ in a DFS scan of $G$.

Proof: Suppose $C'.d < C.d$. That is, $C'$ is discovered first. If a chain of DFS-Visit frames reaches $C$ before $C'.f$, then the chain will reach $u \in C$, then via $(u, v)$ it will reach $v \in C'$, thereby forming a cycle between components $C'$ and $C$. We conclude that $C'.f < C.d < C.f$.

Otherwise, $C.d < C'.d$. That is, $C$ is discovered first, say $C.d = x.d$ for $x \in C$. At $x.d$ time, there is a white path from $x$ to every vertex in $C \cup C'$ because the edge $(u, v)$ bridges the two components. Consequently, every vertex in $C' \cup C$ is a descendant of $x$. By the parenthesis theorem $C.d = x.d < y.d < y.f < x.f = C.f$ for all $y \in (C \cup C') \setminus \{x\}$. Since $C'.f = y.f$ for one of these $y$ vertices, we have $C'.f < C.f$.

**Corollary 22.15**: Let $C, C'$ be strongly connected components in directed graph $G = (V, E)$. Suppose there exists $(u, v) \in E^T$ with $u \in C$ and $v \in C'$. Then $C.f < C'.f$.

Proof: The strongly connected components of $G$ and $G^T$ are identical. Hence $(u, v) \in E^T$ implies $(v, u) \in E$, and the previous lemma then forces $C.f < C'.f$ in a DFS scan of $G$. 

**Theorem 22:16:** The SCC algorithm is correct.

Proof: The algorithm assigns a distinct component label to each $G_\pi$ tree produced by DFS in processing $G^T$. From this point, all references to discovery and finish times will refer to those times established in the first pass, when SCC is processing $G$. After this pass is finished, SCC processes $G^T$, initiating DFS-Visit2 chains from white vertices of decreasing finish times.

Suppose $G^T$-processing starts with SCC initiating DFS-Visit2 on the white node $w$ in component $C_1$. Then $C_1.f = w.f$ is the largest finish time over all vertices. In building this first $G_\pi$ tree, consider a first encounter with a viable vertex from another component, say $C_2$.

That is, in unpacking the transpose-adjacency list of $u \in C_1$, we find a white node $v \in C_2$. If follows that $(v, u) \in E$, the edge set of the original graph. That is, in the original graph $C_2$ points to $C_1$. By Lemma 22.14, $C_1$ must finish first. That is, $w.f = C_1.f < C_2.f$, which is a contradiction because $w.f$ is the largest finish time.

We conclude that the $G_\pi$-tree that starts with $w \in C_1$ includes only vertices in $C_1$. Moreover, it must include all such vertices since the component is strongly connected. When the last DFS-Visit2 call returns to SCC, the driving loop passes over the black vertices of $C_1$ and initiates another chain of DFS-Visit2 calls, starting with a vertex $w' \in C'_1$, for which $w'.f = C'_1.f$ is the largest finish time of the remaining vertices.

We can repeat the argument to show that another $G_\pi$-tree is produced that holds only the vertices of component $C'_1$. An appeal to induction concludes the proof. ■