# Introduction to Algorithms

**George Voutsadakis**[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

1. Elementary Graph Algorithms
   - Representations of Graphs
   - Breadth-First Search
   - Depth-First Search
   - Topological Sort
   - Strongly Connected Components

Subsection 1

# Representations of Graphs

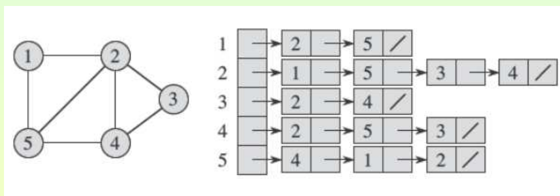
# Methods of Representation

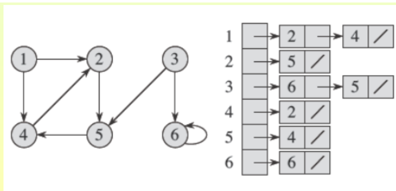
- There are two standard ways to represent a graph $G = (V, E)$.
  - As a collection of adjacency lists;
  - As an adjacency matrix.
- Either way applies to both directed and undirected graphs.
- The adjacency-list representation provides a compact way to represent **sparse graphs** (with $|E|$ much less than $|V|^2$).

  Most of our graph algorithms assume that an input graph is represented in adjacency list form.
- The adjacency-matrix representation is preferable when the graph is **dense** ($|E|$ is close to $|V|^2$) or when we need to be able to tell quickly if there is an edge connecting two given vertices.

# The Adjacency List Representation

- The **adjacency-list** representation of a graph $G = (V, E)$ consists of:
  - An array Adj of $|V|$ lists, one for each vertex in $V$;
  - For each $u \in V$, the adjacency list Adj$[u]$ contains all the vertices $v$, such that there is an edge $(u, v) \in E$.
    The vertices in each list are typically stored in an arbitrary order.





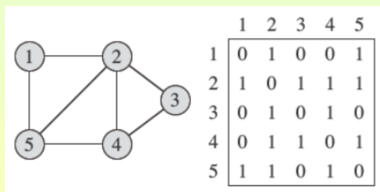
# Memory Requirements versus Flexibility

- If $G$ is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form $(u, v)$ is represented by having $v$ appear in $\text{Adj}[u]$.
- If $G$ is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if $(u, v)$ is an undirected edge, then $u$ appears in $v$'s adjacency list and vice versa.
- For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$.
- We can adapt adjacency lists to represent **weighted graphs**, that is, graphs for which each edge has an associated **weight**, typically given by a **weight function** $w : E \to \mathbb{R}$.

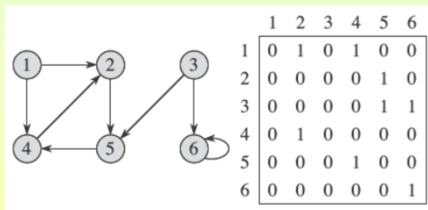
  Example: Let $G = (V, E)$ be a weighted graph with weight function $w$. We simply store the weight $w(u, v)$ of the edge $(u, v) \in E$ with vertex $v$ in $u$'s adjacency list.

# The Adjacency Matrix Representation

- For the **adjacency-matrix representation** of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner.
- Then the adjacency-matrix representation of a graph $G$ consists of a $|V| \times |V|$ matrix $A = (a_{ij})$, such that $a_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

# Memory Requirements and Flexibility

- The adjacency matrix of a graph requires $\Theta(|V|^2)$ memory, independent of the number of edges in the graph.
- Since in an undirected graph, $(u, v)$ and $(v, u)$ represent the same edge, for the adjacency matrix $A$ of an undirected graph $A = A^T$.
- An adjacency matrix can also represent a weighted graph.

  Example: If $G = (V, E)$ is a weighted graph with edge weight function $w$, we can store the weight $w(u, v)$ of the edge $(u, v) \in E$ as the entry in row $u$ and column $v$ of the adjacency matrix.

  If an edge does not exist, we can store a NIL value, but for many problems it is convenient to use 0 or $\infty$.
- Although the adjacency-list representation is asymptotically at least as space efficient as the adjacency-matrix representation, adjacency matrices are simpler.

  Moreover, adjacency matrices for unweighted graphs have the advantage that they require only one bit per entry.

# Representing Attributes

- Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges.
- We write $v.d$ for an attribute $d$ of a vertex $v$
- If edges have an attribute $f$, then we denote this attribute for edge $(u, v)$ by $(u, v).f$.
- If we represent a graph using adjacency lists, one design may represent vertex attributes in additional arrays, such as an array $d[1 \ldots |V|]$ that parallels the Adj array.

  If the vertices adjacent to $u$ are in Adj$[u]$, then what we call the attribute $u.d$ would actually be stored in the array entry $d[u]$.

Subsection 2

# Breadth-First Search

# Introducing Breadth-First Search

- Given a graph $G = (V, E)$ and a distinguished **source vertex** $s$, **breadth-first search** systematically explores the edges of G to "discover" every vertex that is reachable from $s$.
- It computes the distance (smallest number of edges) from $s$ to each reachable vertex.
- It also produces a "breadth-first tree" with root $s$ that contains all reachable vertices.
- For any vertex $v$ reachable from $s$, the simple path in the breadth-first tree from $s$ to $v$ corresponds to a "shortest path" from $s$ to $v$ in $G$, i.e., a path containing the smallest number of edges.
- The algorithm works on both directed and undirected graphs.

# Coloring Vertices During Discovery

- Breadth-first search expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.

  I.e., the algorithm discovers all vertices at distance $k$ from $s$ before discovering any vertices at distance $k+1$.
- To keep track of progress, breadth-first search colors each vertex white, gray or black.
  - All vertices start out white and may later become gray and then black.
  - A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite.
    Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner.
    - If $(u,v) \in E$ and vertex $u$ is black, then vertex $v$ is either gray or black, i.e., all vertices adjacent to black vertices have been discovered.
    - Gray vertices may have some adjacent white vertices. They represent the frontier between discovered and undiscovered vertices.

# The Breadth-First Tree

- Breadth-first search constructs a **breadth-first tree**.
- Initially, it contains only its root, which is the source vertex $s$.
- Whenever the search discovers a white vertex $v$ in the course of scanning the adjacency list of an already discovered vertex $u$, the vertex $v$ and the edge $(u, v)$ are added to the tree.
- We say that $u$ is the **predecessor** or **parent** of $v$ in the breadth-first tree.
- Since a vertex is discovered at most once, it has at most one parent.
- Ancestor and descendant relationships in the breadth-first tree are defined relative to the root $s$ as usual.

  If $u$ is on the simple path in the tree from the root $s$ to vertex $v$, then $u$ is an **ancestor** of $v$ and $v$ is a **descendant** of $u$.

# Setting the Variables

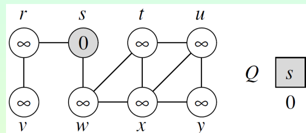
- The breadth-first search procedure BFS below assumes that the input graph $G = (V, E)$ is represented using adjacency lists.
- It attaches several additional attributes to each vertex in the graph.
  - We store the color of each vertex $u \in V$ in the attribute $u$.color;
  - We store the predecessor of $u$ in the attribute $u.\pi$.
    If $u$ has no predecessor (for example, if $u = s$ or $u$ has not been discovered), then $u.\pi =$ NIL.
  - The attribute $u.d$ holds the distance from the source $s$ to vertex $u$ computed by the algorithm.
- The algorithm also uses a first-in, first-out queue $Q$ to manage the set of gray vertices.

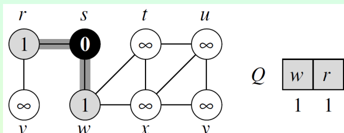
# The Procedure Breadth-First Search

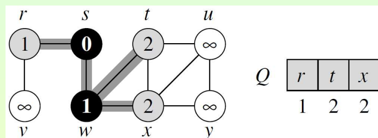
## BFS($G, s$)

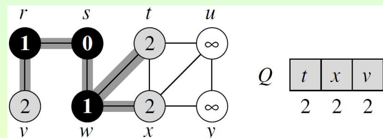
1. for each vertex $u \in G.V - \{s\}$
2.     $u$.color = WHITE
3.     $u.d = \infty$
4.     $u.\pi$ = NIL
5. $s$.color = GRAY
6. $s.d = 0$
7. $s.\pi$ = NIL
8. $Q = \emptyset$
9. EnQueue($Q, s$)
10. while $Q \neq \emptyset$
11.     $u$ = DeQueue($Q$)
12.     for each $v \in G$.Adj[$u$]
13.         if $v$.color == WHITE
14.             $v$.color = GRAY
15.             $v.d = u.d + 1$
16.             $v.\pi = u$
17.             EnQueue($Q, v$)
18.     $u$.color = BLACK

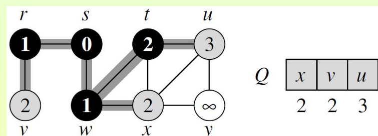
# An Example of Breadth-First Search

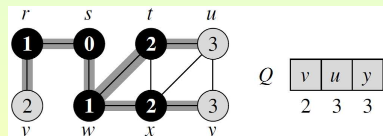


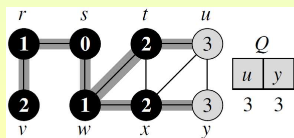



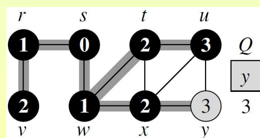



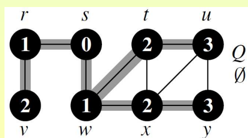



# How Breadth-First Search Works

- With the exception of the source vertex $s$, Lines 1-4:
  - Paint every vertex white;
  - Set $u.d$ to be infinity for each vertex $u$;
  - Set the parent of every vertex to be NIL.
- Line 5 paints $s$ gray, since we consider it to be discovered as the procedure begins.
- Line 6 initializes $s.d$ to 0.
- Line 7 sets the predecessor of the source to be NIL.
- Lines 8-9 initialize $Q$ to the queue containing just the vertex $s$.
- The while loop of Lines 10-18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined.

  This while loop maintains the following invariant:

  At the test in Line 10, the queue $Q$ consists of the set of gray vertices.

# How Breadth-First Search Works (Cont'd)

- Prior to the first iteration, the only gray vertex, and the only vertex in $Q$, is the source vertex $s$. Line 11 determines the gray vertex $u$ at the head of the queue $Q$ and removes it from $Q$. The for loop of Lines 12-17 considers each vertex $v$ in the adjacency list of $u$.
  - If $v$ is white, then it has not yet been discovered, and the algorithm discovers it by executing Lines 14-17.
    - It is first grayed, and its distance $v.d$ is set to $u.d + 1$.
    - Then, $u$ is recorded as its parent $v.\pi$.
    - Finally, it is placed at the tail of the queue $Q$.

  When all the vertices on $u$'s adjacency list have been examined, $u$ is blackened in Line 18.

The loop invariant is maintained:

- Whenever a vertex is painted gray (in Line 14) it is also enqueued (in Line 17);
- Whenever a vertex is dequeued (in Line 11) it is also painted black (in Line 18).

# Analysis

- We analyze the running time on an input graph $G = (V, E)$.
- We use aggregate analysis.
- After initialization, breadth-first search never whitens a vertex.
  Thus, the test in Line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once.
  The operations of enqueuing and dequeuing take $O(1)$ time.
  So the total time devoted to queue operations is $O(|V|)$.
- Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(|E|)$, the total time spent in scanning adjacency lists is $O(|E|)$.
- The overhead for initialization is $O(|V|)$.
- Thus, the total running time of the $\mathrm{BFS}$ procedure is $O(|V| + |E|)$.
  Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of $G$.

# Shortest Paths

- Define the **shortest-path distance** $\delta(s, v)$ from $s$ to $v$ as the minimum number of edges in any path from vertex $s$ to vertex $v$.

  If there is no path from $s$ to $v$, then $\delta(s, v) = \infty$.
- We call a path of length $\delta(s, v)$ from $s$ to $v$ a **shortest path** from $s$ to $v$.

## Lemma

Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.

- If $u$ is reachable from $s$, then so is $v$. In this case, the shortest path from $s$ to $v$ cannot be longer than the shortest path from $s$ to $u$ followed by the edge $(u, v)$. Thus, the inequality holds.

  If $u$ is not reachable from $s$, then $\delta(s, u) = \infty$, and the inequality holds.

# $v.d$ Bounds $\delta(s, v)$ From Above

**Lemma**

Let $G = (V, E)$ be a directed or undirected graph, and suppose that $\mathrm{BFS}$ is run on $G, s$, with $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by $\mathrm{BFS}$ satisfies $v.d \geq \delta(s, v)$.

- We use induction on the number of $\textsc{EnQueue}$ operations. The inductive hypothesis is that $v.d \geq \delta(s, v)$, for all $v \in V$.
  - The basis of the induction is the situation immediately after enqueuing $s$ in Line 9 of $\mathrm{BFS}$. The inductive hypothesis holds here, because $s.d = 0 = \delta(s, s)$ and $v.d = \infty \geq \delta(s, v)$, for all $v \in V - \{s\}$.
  - For the inductive step, consider a white vertex $v$ discovered during the search from $u$. By the inductive hypothesis, $u.d \geq \delta(s, u)$. From the assignment performed by Line 15 and from the lemma, we obtain $v.d = u.d + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$. Vertex $v$ is then enqueued, and it is never enqueued again because it is also grayed and the *then* clause of Lines 14-17 is executed only for white vertices. Thus, the value of $v.d$ never changes again, and the inductive hypothesis is maintained.

# Distance Values in the Queue

**Lemma**

Suppose that during the execution of BFS on $G = (V, E)$, the queue $Q$ contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where $v_1$ is the head of $Q$ and $v_r$ is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$, $i = 1, 2, \ldots, r - 1$.

- The proof is by induction on the number of queue operations.
  - Initially, when the queue contains only $s$, the lemma certainly holds.
  - For the inductive step, we must prove that the lemma holds after both dequeuing and enqueuing a vertex.
    - If the head $v_1$ of the queue is dequeued, $v_2$ becomes the new head. By the inductive hypothesis, $v_1.d \leq v_2.d$. Then $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$. So the remaining inequalities are unaffected.
    - When we enqueue a vertex $v$ in Line 17 of BFS, it becomes $v_{r+1}$. At that time, we have removed vertex $u$, whose adjacency list is currently being scanned, from $Q$. By the inductive hypothesis, the new head $v_1$ has $v_1.d \geq u.d$. Thus, $v_{r+1}.d = v.d = u.d + 1 \leq v_1.d + 1$. From the inductive hypothesis, $v_r.d \leq u.d + 1$. So $v_r.d \leq u.d + 1 = v.d = v_{r+1}.d$. Hence, the remaining inequalities are unaffected.

# Increasing Distance Values

- The following corollary shows that the $d$ values at the time that vertices are enqueued are monotonically increasing over time.

**Corollary**

Suppose that vertices $v_i$ and $v_j$ are enqueued during the execution of $\mathrm{BFS}$, and that $v_i$ is enqueued before $v_j$. Then $v_i.d \leq v_j.d$ at the time that $v_j$ is enqueued.

- Immediate from the preceding lemma and the property that each vertex receives a finite $d$ value at most once during the course of $\mathrm{BFS}$.

# Correctness of Breadth-First Search

## Theorem (Correctness of Breadth-First Search)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that $\mathrm{BFS}$ is run on $G$ from a given source vertex $s \in V$. Then, during its execution, $\mathrm{BFS}$ discovers every vertex $v \in V$ that is reachable from the source $s$, and upon termination, $v.d = \delta(s, v)$, for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from $s$, one of the shortest paths from $s$ to $v$ is a shortest path from $s$ to $v.\pi$ followed by the edge $(v.\pi, v)$.

- Assume some vertex receives a $d$ value not equal to its shortest-path distance. Let $v$ be the vertex with minimum $\delta(s, v)$ that receives such an incorrect $d$ value. Clearly $v \neq s$. By a previous lemma, $v.d \geq \delta(s, v)$, and, thus, we have that $v.d > \delta(s, v)$. Vertex $v$ must be reachable from $s$, for if it is not, then $\delta(s, v) = \infty \geq v.d$. Let $u$ be the vertex immediately preceding $v$ on a shortest path from $s$ to $v$, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, u) < \delta(s, v)$, and the choice of $v$, $u.d = \delta(s, u)$. Now we get, $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$.

# Correctness of Breadth-First Search (Cont'd)

- Now consider the time when $\mathrm{BFS}$ chooses to dequeue vertex $u$ from $Q$ in Line 11. At this time, vertex $v$ is either white, gray or black.
  - If $v$ is white, then Line 15 sets $v.d = u.d + 1$, a contradiction.
  - If $v$ is black, then it was already removed from the queue. By the preceding corollary, we have $v.d \leq u.d$, again a contradiction.
  - If $v$ is gray, then it was painted gray upon dequeuing some vertex $w$, which was removed from $Q$ earlier than $u$ and s.t. $v.d = w.d + 1$. By the preceding corollary, $w.d \leq u.d$. So $v.d = w.d + 1 \leq u.d + 1$, a contradiction.

  Thus, $v.d = \delta(s, v)$, for all $v \in V$.

  All vertices $v$ reachable from $s$ must be discovered, for otherwise they would have $\infty = v.d > \delta(s, v)$.

  To conclude the proof, observe that if $v.\pi = u$, then $v.d = u.d + 1$.

  Thus, we can obtain a shortest path from $s$ to $v$ by taking a shortest path from $s$ to $v.\pi$ and then traversing the edge $(v.\pi, v)$.

# Breadth-First Trees

- The procedure BFS builds a **breadth-first tree**, which corresponds to the $\pi$ attributes.
- For a graph $G = (V, E)$, with source $s$, we define the **predecessor subgraph** of $G$ as $G_\pi = (V_\pi, E_\pi)$, where
  - $V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$;
  - $E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$.
- The predecessor subgraph $G_\pi$ is a **breadth-first tree** if $V_\pi$ consists of the vertices reachable from $s$ and, for all $v \in V_\pi$, the subgraph $G_\pi$ contains a unique simple path from $s$ to $v$ that is also a shortest path from $s$ to $v$ in $G$.
- A breadth-first tree is in fact a tree, since it is connected and $|E_\pi| = |V_\pi| + 1$.
- We call the edges in $E_\pi$ **tree edges**.

# Breadth-First Trees

- The following lemma shows that the predecessor subgraph produced by $\mathrm{BFS}$ is a breadth-first tree.

**Lemma**

When applied to a directed or undirected graph $G = (V, E)$, procedure $\mathrm{BFS}$ constructs $\pi$ so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

- Line 16 of $\mathrm{BFS}$ sets $v.\pi = u$ if and only if $(u, v) \in E$ and $\delta(s, v) < \infty$, i.e., if $v$ is reachable from $s$. Thus, $V_\pi$ consists of the vertices in $V$ reachable from $s$. Since $G_\pi$ forms a tree, it contains a unique simple path from $s$ to each vertex in $V_\pi$. The preceding theorem shows that every such path is a shortest path in $G$.

# Printing a Shortest Path

- The following procedure prints out the vertices on a shortest path from $s$ to $v$, assuming that BFS has already computed a breadth-first tree.

**PRINTPATH$(G, s, v)$**

1. if $v == s$
2. print $s$
3. elseif $v.\pi ==$ NIL
4. print "no path from" $s$ "to" $v$ "exists"
5. else PRINTPATH$(G, s, v.\pi)$
6. print $v$

- Each recursive call is for a path one vertex shorter.

  Thus, PRINTPATH runs in time linear in the number of vertices in the path printed.

Subsection 3

# Depth-First Search

# Idea of Depth-First Search

- The strategy followed by depth-first search is to search "deeper" in the graph whenever possible.
  - Depth-first search explores edges out of the most recently discovered vertex $v$ that still has unexplored edges leaving it.
  - Once all of $v$'s edges have been explored, the search "backtracks" to explore edges leaving the vertex from which $v$ was discovered.

  This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.
- The algorithm repeats this entire process until it has discovered every vertex.

# The Depth-First Forest

- Whenever depth-first search discovers a vertex $v$ during a scan of the adjacency list of an already discovered vertex $u$, it records this event by setting $v$'s predecessor attribute $v.\pi$ to $u$.
- Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources.
- Therefore, we define the **predecessor subgraph** of a depth-first search as $G_\pi = (V, E_\pi)$, where:
  - $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$.
- The predecessor subgraph of a depth-first search forms a **depth-first forest** comprising several **depth-first trees**.
- The edges in $E_\pi$ are **tree edges**.

# Coloring and Timestamping

- Depth-first search colors vertices during the search to indicate their state.
  - Each vertex is initially white;
  - It is grayed when it is **discovered** in the search;
  - It is blackened when it is **finished**, that is, when its adjacency list has been examined completely.
- This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.
- Besides creating a depth-first forest, depth-first search also **timestamps** each vertex.

  Each vertex $v$ has two timestamps:
  - The first timestamp $v.d$ records when $v$ is first discovered (and grayed);
  - The second timestamp $v.f$ records when the search finishes examining $v$'s adjacency list (and blackens $v$).

  These timestamps provide important information about the structure of the graph.

# The Depth-First Search Procedure

- The procedure DFS below records when it discovers vertex $u$ in the attribute $u.d$ and when it finishes vertex $u$ in the attribute $u.f$.
- These timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices.
- For every vertex $u$, $u.d < u.f$.
- Vertex $u$ is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, and BLACK thereafter.
- The input graph $G$ may be undirected or directed.

**DFS($G$)**

```
1. for each vertex u ∈ G.V
2.    u.color = WHITE
3.    u.π = NIL
4. time = 0
5. for each vertex u ∈ G.V
6.    if u.color == WHITE
7.       DFSVisit(G, u)
```

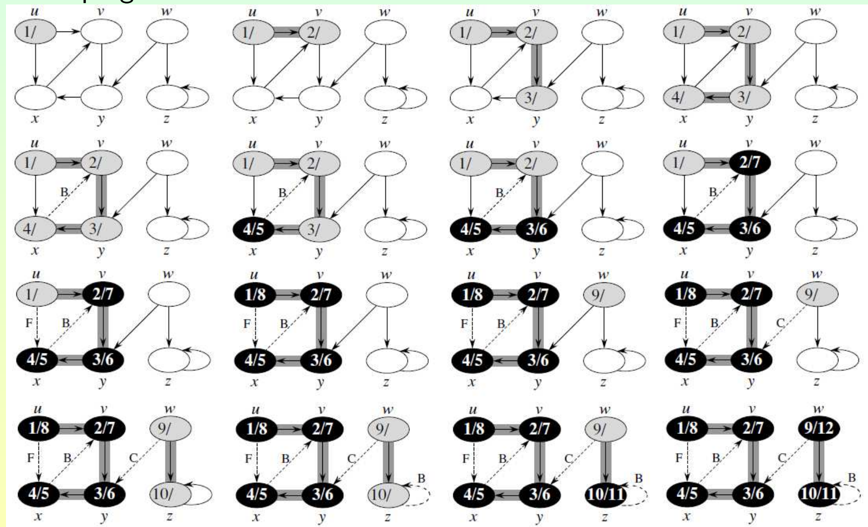
# The Subroutine DFSVisit

**DFSVisit($G, u$)**

1. time = time + 1 //white vertex $u$ has just been discovered
2. $u.d$ = time
3. $u$.color = GRAY
4. for each $v \in G$.Adj[$u$] //explore edge $(u, v)$
5.     if $v$.color == WHITE
6.         $v.\pi = u$
7.         DFSVisit($G, u$)
8. $u$.color = BLACK //blacken $u$; it is finished
9. time = time + 1
10. $u.f$ = time

# Illustration of DFS

- The progress of DFS:



# How DFS Works

- Lines 1-3 paint all vertices white and initialize $\pi$ attributes to NIL.
- Line 4 resets the global time counter.
- Lines 5-7 check each vertex in $V$ in turn and, when a white vertex is found, visit it using $\mathrm{DFSV{\scriptsize ISIT}}$:
  - Every time $\mathrm{DFSV{\scriptsize ISIT}}(G,u)$ is called in Line 7, vertex $u$ becomes the root of a new tree in the depth-first forest.
    - In each call $\mathrm{DFSV{\scriptsize ISIT}}(u)$, vertex $u$ is initially white.
    - Line 1 increments time.
    - Line 2 records the new value of time as the discovery time $u.d$.
    - Line 3 paints $u$ gray.
    - Lines 4-7 examine each vertex $v$ adjacent to $u$ and recursively visit $v$ if it is white.

      As each vertex $v \in \mathrm{Adj}[u]$ is considered in Line 4, we say that edge $(u,v)$ is **explored** by the depth-first search.
    - Finally, after every edge leaving $u$ has been explored, Lines 8-10 paint $u$ black, increment time and record the finishing time in $u.f$.
  - When DFS returns, every vertex $u$ has been assigned a **discovery time** $u.d$ and a **finishing time** $u.f$.

# Running Time of DFS

- The loops on Lines 1-3 and Lines 5-7 of DFS take time $\Theta(|V|)$, exclusive of the time to execute the calls to DFSVisit.
- We use aggregate analysis.

  The procedure DFSVisit is called exactly once for each vertex $v \in V$, since the vertex $u$ on which DFSVisit is invoked must be white and the first thing DFSVisit does is paint vertex $u$ gray.

  During an execution of $\text{DFSVisit}(G, v)$, the loop on Lines 4-7 executes $|\text{Adj}[v]|$ times.

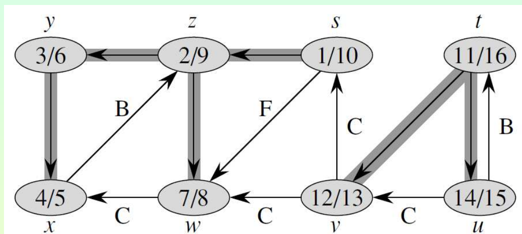
  Since $\sum_{v \in V} |\text{Adj}[v]| = \Theta(|E|)$, the total cost of executing Lines 4-7 of DFSVisit is $\Theta(|E|)$.
- The running time of DFS is therefore $\Theta(|V| + |E|)$.

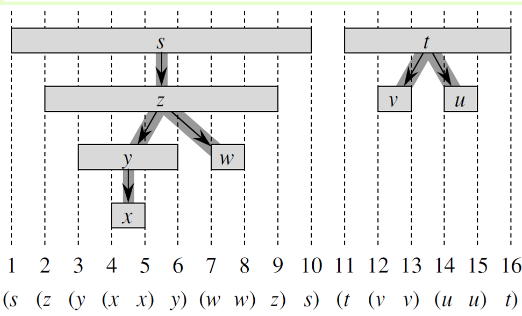
# Properties of Depth-First Search

- The most basic property of depth-first search is that the predecessor subgraph $G_\pi$ does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of $\text{DFSVisit}$:
  - $u = v.\pi$ if and only if $\text{DFSVisit}(G, v)$ was called during a search of $u$'s adjacency list.
  - Additionally, vertex $v$ is a descendant of vertex $u$ in the depth-first forest if and only if $v$ is discovered during the time in which $u$ is gray.
- Another important property of depth-first search is that discovery and finishing times have **parenthesis structure**:

  If we represent the discovery of vertex $u$ with a left parenthesis "$(u$" and represent its finishing by a right parenthesis "$u)$", the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested.

# Illustration of Parenthesis Structure



# Parenthesis Theorem

**Theorem (Parenthesis Theorem)**

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices $u$ and $v$, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither $u$ nor $v$ is a descendant of the other in the depth-first forest;
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and $u$ is a descendant of $v$ in a depth-first tree;
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and $v$ is a descendant of $u$ in a depth-first tree.

- Consider the case in which $u.d < v.d$. We consider two subcases, according to whether $v.d < u.f$ or not.

# Parenthesis Theorem (Cont'd)

- Case in which $u.d < v.d$.
  - The first subcase occurs when $v.d < u.f$. So $v$ was discovered while $u$ was still gray. This implies that $v$ is a descendant of $u$. Moreover, since $v$ was discovered more recently than $u$, all of its outgoing edges are explored, and $v$ is finished, before the search returns to and finishes $u$. In this case, therefore, the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$.
  - In the other subcase, $u.f < v.d$. By the inequality, $u.d < u.f < v.d < v.f$. Thus, the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray. So neither vertex is a descendant of the other.
- The case in which $v.d < u.d$ is symmetric to the above.

**Corollary (Nesting of Descendants' Intervals)**

Vertex $v$ is a proper descendant of vertex $u$ in the depth-first forest for a (directed or undirected) graph $G$ if and only if $u.d < v.d < v.f < u.f$.

# The White-Path Theorem

- The next theorem gives another important characterization of when one vertex is a descendant of another in the depth-first forest.

**Theorem (White-Path Theorem)**

In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex $v$ is a descendant of vertex $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting entirely of white vertices.

($\Rightarrow$) If $v = u$, then the path from $u$ to $v$ contains just vertex $u$, which is still white when we set the value of $u.d$. Now, suppose that $v$ is a proper descendant of $u$ in the depth-first forest. By the corollary, $u.d < v.d$. So $v$ is white at time $u.d$. Since $v$ can be any descendant of $u$, all vertices on the unique simple path from $u$ to $v$ in the depth-first forest are white at time $u.d$.

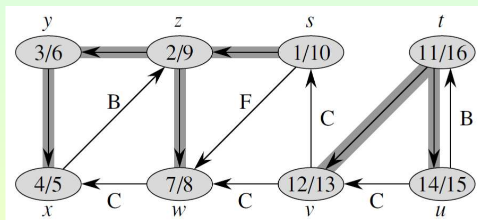
# The White-Path Theorem (The Reverse)

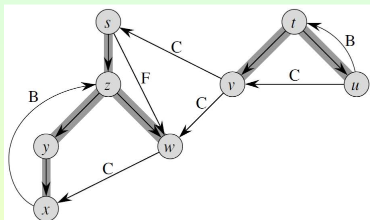
($\Leftarrow$) Suppose that there is a path of white vertices from $u$ to $v$ at time $u.d$, but $v$ does not become a descendant of $u$ in the depth-first tree. Without loss of generality, assume that every vertex other than $v$ along the path becomes a descendant of $u$. (Otherwise, let $v$ be the closest vertex to $u$ along the path that does not become a descendant of $u$.) Let $w$ be the predecessor of $v$ in the path, so that $w$ is a descendant of $u$ ($w$ and $u$ may in fact be the same vertex). By the corollary, $w.f \leq u.f$. Because $v$ must be discovered after $u$ is discovered, but before $w$ is finished, we have $u.d < v.d < w.f \leq u.f$. By the theorem, the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$. By the corollary, $v$ must be a descendant of $u$.

# Classification of Edges

- Depth-first search can be used to classify the edges of the input graph $G = (V, E)$.
- We can define four edge types in terms of the depth-first forest $G_\pi$ produced by a depth-first search on $G$:
  1. **Tree edges** are edges in the depth-first forest $G_\pi$. Edge $(u, v)$ is a tree edge if $v$ was first discovered by exploring edge $(u, v)$.
  2. **Back edges** are those edges $(u, v)$ connecting a vertex $u$ to an ancestor $v$ in a depth-first tree.
     We consider self-loops, which may occur in directed graphs, to be back edges.
  3. **Forward edges** are those nontree edges $(u, v)$ connecting a vertex $u$ to a descendant $v$ in a depth-first tree.
  4. **Cross edges** are all other edges.
     They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

# Illustrating the Classification of Edges




- Edge labels indicate edge types.
- In the right figure, all tree and forward edges head downward in a depth-first tree and all back edges go up.

# DFS and the Classification

- The DFS algorithm has enough information to classify some edges as it encounters them.
- The key idea is that when we first explore an edge $(u, v)$, the color of vertex $v$ tells us something about the edge.
  1. WHITE indicates a tree edge;
  2. GRAY indicates a back edge;
  3. BLACK indicates a forward or cross edge.
- The first case is immediate from the specification of the algorithm.
- For the second case, observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFSVISIT invocations. The number of gray vertices is one more than the depth in the depth-first forest of the vertex most recently discovered. Exploration always proceeds from the deepest gray vertex, so an edge that reaches another gray vertex has reached an ancestor.
- The third case handles the remaining possibility.

# Undirected Graphs: Forward and Cross Edges

- We now show that forward and cross edges never occur in a depth-first search of an undirected graph.

**Theorem**

In a depth-first search of an undirected graph $G$, every edge of $G$ is either a tree edge or a back edge.

- Let $(u, v)$ be an arbitrary edge of $G$. Suppose without loss of generality that $u.d < v.d$. Then the search must discover and finish $v$ before it finishes $u$ (while $u$ is gray), since $v$ is on $u$'s adjacency list. Consider the first time that the search explores edge $(u, v)$.
  - If it is in the direction from $u$ to $v$, then $v$ is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from $v$ to $u$. So $(u, v)$ becomes a tree edge.
  - If it is in the direction from $v$ to $u$, then $(u, v)$ is a back edge, since $u$ is still gray at the time the edge is first explored.

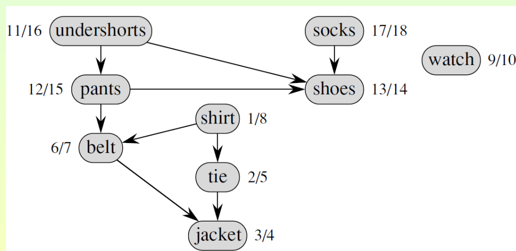
Subsection 4

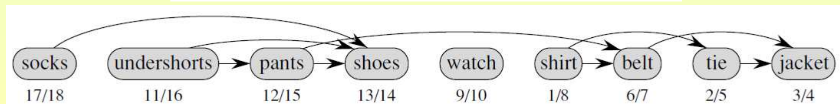
## Topological Sort

# Topological Sort

- A **topological sort** of a directed acyclic graph (DAG) $G = (V, E)$ is a linear ordering of all its vertices such that, if $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering.
- We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

  Example:



# The Topological Sort Procedure

**TopologicalSort($G$)**

1. call $\mathrm{DFS}(G)$ to compute finishing times $v.f$ for each vertex $v$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

- We can perform a topological sort in time $\Theta(|V| + |E|)$.
  - Depth-first search takes $\Theta(|V| + |E|)$ time;
  - It takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

# Characterizing DAGs

- The correctness of this algorithm uses the following key lemma characterizing directed acyclic graphs.

**Lemma**

A directed graph $G$ is acyclic if and only if a depth-first search of $G$ yields no back edges.

($\Rightarrow$) Suppose that a depth-first search produces a back edge $(u, v)$. Then vertex $v$ is an ancestor of vertex $u$ in the depth-first forest. Thus, $G$ contains a path from $v$ to $u$, and edge $(u, v)$ completes a cycle.

($\Leftarrow$) Suppose that $G$ contains a cycle $c$. We show that a depth-first search of $G$ yields a back edge. Let $v$ be the first vertex to be discovered in $c$. Let $(u, v)$ be the preceding edge in $c$. At time $v.d$, the vertices of $c$ form a path of white vertices from $v$ to $u$. By the White-Path Theorem, vertex $u$ becomes a descendant of $v$ in the depth-first forest. Therefore, $(u, v)$ is a back edge.

# Correctness of the Topological Sort

**Theorem**

TopologicalSort produces a topological sort of the directed acyclic graph provided as its input.

- Suppose that DFS is run on a given DAG $G = (V, E)$ to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if $G$ contains an edge from $u$ to $v$, then $v.f < u.f$. Consider any edge $(u, v)$ explored by DFS$(G)$. When this edge is explored, $v$ cannot be gray, since then $v$ would be an ancestor of $u$ and $(u, v)$ would be a back edge, contradicting the lemma. Therefore, $v$ must be either white or black.
  - If $v$ is white, it becomes a descendant of $u$. So $v.f < u.f$.
  - If $v$ is black, it has already been finished. So $v.f$ has already been set. Because we are still exploring from $u$, we have yet to assign a timestamp to $u.f$. So once we do, we will have $v.f < u.f$ as well.

  Thus, for any edge $(u, v)$ in the DAG, we have $v.f < u.f$.

Subsection 5

## Strongly Connected Components

# Strongly Connected Components and Transpose

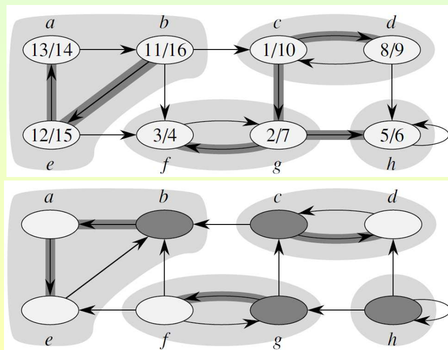
- A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$, such that for every pair of vertices $u$ and $v$ in $C$, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$, i.e., vertices $u$ and $v$ are reachable from each other.
- Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the **transpose** of $G$, $G^T = (V, E^T)$, where

$$E^T = \{(u, v) : (v, u) \in E\},$$

i.e., $E^T$ consists of the edges of $G$ with their directions reversed.

# Example

- Given an adjacency-list representation of $G$, the time to create $G^T$ is $O(|V| + |E|)$.
- Observe that $G$ and $G^T$ have exactly the same strongly connected components. $u$ and $v$ are reachable from each other in $G$ if and only if they are reachable from each other in $G^T$.



# Discovering Strongly Connected Components

- The following $\Theta(|V| + |E|)$-time algorithm computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on $G$ and one on $G^T$.

**StronglyConnectedComponents($G$)**

1. call DFS($G$) to compute finishing times $u.f$ for each vertex $u$
2. compute $G^T$
3. call DFS($G^T$), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in Line 3 as a separate strongly connected component

# The Component Graph

- Suppose $G$ has strongly connected components $C_1, C_2, \ldots, C_k$.
- The **component graph** $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$ of $G$ is define as follows:
  - The vertex set $V^{\mathrm{SCC}}$ is $\{v_1, v_2, \ldots, v_k\}$, and it contains a vertex $v_i$ for each strongly connected component $C_i$ of $G$;
  - There is an edge $(v_i, v_j) \in E^{\mathrm{SCC}}$ if $G$ contains a directed edge $(x, y)$, for some $x \in C_i$ and some $y \in C_j$.

# The Component Graph is a DAG

- The key property is that the component graph is a DAG.

**Lemma**

Let $C$ and $C'$ be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that $G$ contains a path $u \rightsquigarrow u'$. Then $G$ cannot also contain a path $v' \rightsquigarrow v$.

- If $G$ contains a path $v' \rightsquigarrow v$, then it contains paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$. Thus, $u$ and $u'$ are reachable from each other. This contradicts the assumption that $C$ and $C'$ are distinct strongly connected components.
- We shall see that by considering vertices in the second depth-first search in decreasing order of the finishing times that were computed in the first depth-first search, we are, in essence, visiting the vertices of the component graph (each of which corresponds to a strongly connected component of $G$) in topologically sorted order.

# Relating Strongly Connected Components

- When we discuss $u.d$ or $u.f$, the values always refer to the discovery and finishing times as computed by the first call of $\mathrm{DFS}$, in Line 1.
- If $U \subseteq V$, define $d(U) = \min_{u \in U} \{u.d\}$ and $f(U) = \max_{u \in U} \{u.f\}$. That is, $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time, respectively, of any vertex in $U$.

**Lemma**

Let $C$ and $C'$ be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

- We consider two cases, depending on which strongly connected component, $C$ or $C'$, had the first discovered vertex during the depth-first search.

# Relating Strongly Connected Components (Cont'd)

- If $d(C) < d(C')$, let $x$ be the first vertex discovered in $C$. At time $x.d$, all vertices in $C$ and $C'$ are white. At that time, $G$ contains a path from $x$ to each vertex in $C$ consisting only of white vertices. Because $(u, v) \in E$, for any vertex $w \in C'$, there is a path in $G$ at time $x.d$ $x \rightsquigarrow u \to v \rightsquigarrow w$ consisting only of white vertices. By the White-Path Theorem, all vertices in $C$ and $C'$ become descendants of $x$. By the corollary, $x.f = f(C) > f(C')$.
- If $d(C) > d(C')$, let $y$ be the first vertex discovered in $C'$. At time $y.d$, all vertices in $C'$ are white and $G$ contains a path from $y$ to each vertex in $C'$ consisting only of white vertices. By the White-Path Theorem, all vertices in $C'$ become descendants of $y$. By the corollary, $y.f = f(C')$. At time $y.d$, all vertices in $C$ are white. Since there is an edge $(u, v)$ from $C$ to $C'$, the lemma implies that there cannot be a path from $C'$ to $C$. Hence, no vertex in $C$ is reachable from $y$. At time $y.f$, therefore, all vertices in $C$ are still white. Thus, for any vertex $w \in C$, we have $w.f > y.f$. This implies that $f(C) > f(C')$.

# Finishing Times of Strongly Connected Components

- Each edge in $G^T$ that goes between different strongly connected components goes from a component with an earlier to a component with a later finishing time.

**Corollary**

Let $C$ and $C'$ be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

- Since $(u, v) \in E^T$, we have $(v, u) \in E$.

  But the strongly connected components of $G$ and $G^T$ are the same.

  The lemma implies that $f(C) < f(C')$.

# The Depth-First Search on $G^T$

- In the second depth-first search on $G^T$, we start with the strongly connected component $C$ whose finishing time $f(C)$ is maximum.
  - The search starts from some vertex $x \in C$, and it visits all vertices in $C$. By the corollary, $G^T$ contains no edges from $C$ to any other strongly connected component. So the search from $x$ will not visit vertices in any other component. Thus, the tree rooted at $x$ contains exactly the vertices of $C$.
  - Having completed visiting all vertices in $C$, the search in Line 3 selects as a root a vertex from some other strongly connected component $C'$ whose finishing time $f(C')$ is maximum over all components other than $C$. The search will visit all vertices in $C'$. By the corollary, the only edges in $G^T$ from $C'$ to any other component must be to $C$, which we have already visited.
  - In general, when the depth-first search of $G^T$ in Line 3 visits any strongly connected component, any edges out of that component must be to components that the search already visited. Each depth-first tree, therefore, will be exactly one strongly connected component.

# Correctness of StronglyConnectedComponents

**Theorem**

StronglyConnectedComponents correctly computes the strongly connected components of the directed graph G provided as its input.

- We argue, by induction on the number of depth-first trees found in the depth-first search of $G^T$ in Line 3, that the vertices of each tree form a strongly connected component.

  The inductive hypothesis is that the first $k$ trees produced in Line 3 are strongly connected components.
  - The basis for the induction, when $k = 0$, is trivial.
  - In the inductive step, we assume that each of the first $k$ depth-first trees produced in Line 3 is a strongly connected component. We consider the $(k+1)$-st tree produced.

# The Induction Step

- Let the root of this tree be vertex $u$, and let $u$ be in strongly connected component $C$. Because of how we choose roots in the depth-first search in Line 3, $u.f = f(C) > f(C')$ for any strongly connected component $C'$ other than $C$ that has yet to be visited.

  By the inductive hypothesis, at the time that the search visits $u$, all other vertices of $C$ are white. By the White-Path Theorem, therefore, all other vertices of $C$ are descendants of $u$ in its depth-first tree. Moreover, by the inductive hypothesis and by the corollary, any edges in $G^T$ that leave $C$ must be to strongly connected components that have already been visited. Thus, no vertex in any strongly connected component other than $C$ will be a descendant of $u$ during the depth-first search of $G^T$. It follows that the vertices of the depth-first tree in $G^T$ that is rooted at $u$ form exactly one strongly connected component. This completes the inductive step.