# Class Notes
# CIS 502 Analysis of Algorithm
# 3-Graph Traversal

Da Kuang

University of Pennsylvania

# 1 Graphics Basics

- A graph G is an ordered pair of two sets $(V, E)$.
- $V$ is a set of vertices/points/nodes, which is always a finite set.
- $E$ is a set of unordered pair of vertices.
- An edge is represented as $(u, v)$. Here we abuse the notion of ordered pair to represent unordered pair.

## 1.1 Two representation of Graph

When we talked about graph without adjective, that mean it is a undirected graph. Suppose the number of vertices is $|V| = n$.

- A vertex is incident to an edge if the vertex is one of the two vertices the edge connects.
- If an edge $(u, v)$ has end points $u$ and $v$, we say it is an incident to vertex $u$ and $v$.
- $u$, $v$ are adjacent if $(u, v) \in \mathbb{E}$.
- The degree of vertex $v$ is the number of edges incident on $v$.

### 1.1.1 Adjacency Matrix

Adjacency Matrix is an symmetric matrix for undirected graph where

$$V_{ij} = \begin{cases} |(i,j)| & \text{, if } (i,j) \in \mathbb{E}. \\ 0 & \text{, otherwise.} \end{cases}$$

### 1.1.2 Adjacency List

Adjacency List is an array of size n of linked list, where $i$-th entry is a linked list consisting of the neighbors of vertex-$i$. It is default representation of graph.

Space = $O(n + m)$

## 1.2 Connectivity

### 1.2.1 Path

A path in a graph is a sequence of vertices

$$v_0 v_1 \cdots v_k$$

, such that $(v_i, v_{i+1}) \in \mathbb{E}$ for $i = 0, 1, 2, \cdots, k-1$ A simple path is a path that does not repeat vertices.

**Lemma:** If there is a path $(u, v)$, there must be a simple path $(u, v)$.

### 1.2.2 Cycle

A cycle in a graph is a sequence of vertices

$$v_0 v_1 \cdots v_k v_0$$

, such that $(v_i, v_{i+1}) \in \mathbb{E}$ for $i = 0, 1, 2, \cdots, k-1$ and $(v_k, v_0) \in \mathbb{E}$. All $v_i$s are distinct.

### 1.2.3 Connectivity

- $u$, $v$ is **connected** if there is a path between them.
- $G$ is **connected** if $\forall u, v \in V$, there is a path between $u$ and $v$.
- The **connected components** of $G$ are maximal subset of vertices that are pairwise connected.

### 1.2.4 Connection is equivalence relation

Connection relation in a graph is an equivalence relation.

- Reflexive Relation (take Path of length 0)
- Symmetric Relation (reversible path)
- Transitive Relation: If a Graph has a $uv$ path and also $vw$ path then it will also contain $uw$ path.

Because connection is the equivalence relation, pairwise connected vertices form a connected component.

# 2 Tree

Tree is a connected acyclic graph.

## 2.1 Rooted tree

### 2.1.1 Inductive Definition

A nice thing about Inductive definition is it is useful for the proofs by induction.

- **Rule 1:** A graph consist of a single vertex v is a rooted tree with v as the root.
- **Rule 2:** If $(T_1, r_1)$, $(T_2, r_2), \cdot, (T_k, r_k)$ are rooted trees, then the tree $(T, r)$ consisting of a new node $r$ as root and edges $(r, r_1), \cdots, (r, r_k)$ is a rooted tree.

## 2.2 Structural induction Proof

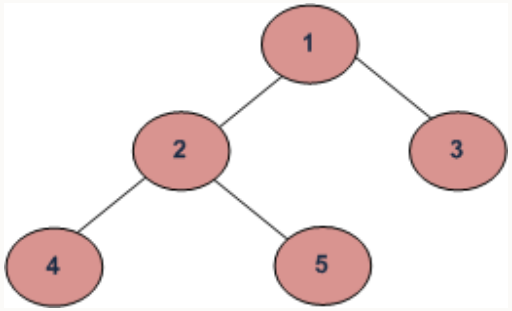
Statement: Any tree with $n$ nodes has $n - 1$ edges.

Since any tree can be transformed into a rooted tree, the induction can be as following:

- Statement: Any rooted tree on $n$ nodes has $n-1$ edges.
- Base case: Single node tree with no edge. The statement is true.
- Inductive hypothesis: For a rooted tree $T_r$, built up from $(T_1, r_1), (T_2, r_2), \cdots, (T_n, r_n)$ using rule 2. Assume the statement is true for all the trees $T_1, T_2, \cdots, T_k$ and prove it for $t$.
- Inductive step:
  - Let tree $T_i$ have $n_i$ nodes, $i = 1, 2, \cdots, k$. Then $T$ has $\sum_{i=1}^{k} n_i + 1$ nodes.
  - By the inductive hypothesis, $T_i$ has $n_i - 1$ edges.
  - Total number of edges is $T = \sum_{i=1}^{k}(n_i - 1) + k = \sum_{i=1}^{k} n_i$.
  - The number of edge is one less than the number of nodes. It proofs the inductive step.

# 3 Traversal

Traversal: Visiting all parts of the graph. Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.




- Depth First Traversals:
  - Inorder (Left, Root, Right) : 4 2 5 1 3
  - Preorder (Root, Left, Right) : 1 2 4 5 3
  - Postorder (Left, Right, Root) : 4 5 2 3 1
- Breadth First or Level Order Traversal : 1 2 3 4 5

## 3.1 Traversal rooted tree

### 3.1.1 Post-order traversal

- First traverse each of the children
- Visit the root.

## 3.2 Bread-first Search (BFS)

Most of the cases, BFS is used to find out the shortest path from an unweighted and undirected graph.

### 3.2.1 Basics

- Input:Graph $G = (V, E)$; a vertex $s$ as starting point.
- Goal: Visit all the vertices in systematic manner.
- Intuition: Think of the graph is drawn as a pond of water. Drop a stone at vertex v and watch the ripple expand with great regularity The order of vertex visited in BFS is like the wave of ripples visiting the vertices.

### 3.2.2 Algorithm

**3 possible states for each vertex:**

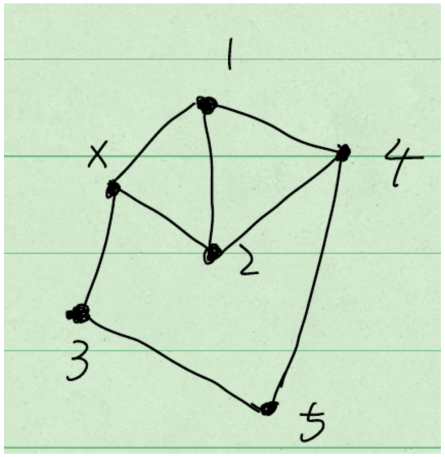
- Undiscovered
- Discovered: Just find out this vertex and have not check its neighbors.
- Finished: discover the node and all its neighbors.

**Queue:**

- Initialized with just vertex $s$. $Q \leftarrow \{x\}$. x is discovered.
- Work the follows repeatedly:
  - Pull a vertex out of the queue.
  - Put all its neighbors into the queue as discovered.

**Example:** An example of BFS starting from vertex $x$.

| Q | x |
|---|---|
| Q0 | 1, 3, 2 |
| Q1 | 3, 2, 4 |
| Q2 | 2, 4, 5 |
| Q3 | 4, 5 |
| Q4 | 5 |
| Q5 | ∅ |




**Remarks:**

- In $Q1$, we did not put vertex 2 into the queue since it is already in the Queue.
- Vertices in queue are discovered.
- Moving vertices out of the queue means it is finished.

### 3.2.3 Levels

**Definition.** Associate levels with vertices in BFS:

- The starting vertex is level 0: $\text{level}(s) = 0$.
- Any vertex $v$ is discovered when dequeue some vertex $u$, we set

$$\text{level}(v) = \text{level}(u) + 1$$

**Theorem 3.1** *The level of $v$ is the smallest number of hops to get from $s$ to $v$.*

**Proof.** Let $\delta(s, v)$ be the smallest number of hops from $x$ to $v$. Prove that for any $v$, $\delta(s, v) = \text{level}(v)$.

Claim: $\delta(s, v) \leq \text{level}(v)$.

Proof: If $v$ has level level$v$, there is a sequence of vertices

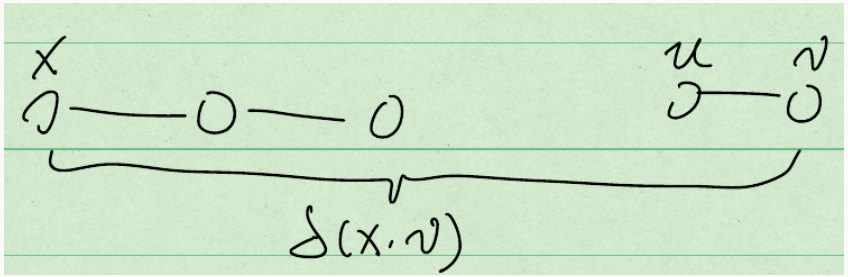
$$x,\ v_1,\ v_2,\ \cdots,\ v_{\text{level}(v)-1},\ v$$

, s.t. each one discovered from the previous.

We have shown one path from $x$ to $v$ which has $\text{level}(v)$ hops. Therefore, the smallest number of hops should be less or equal to the number of hops in this path. Therefore, $\delta(x, v) \leq \text{level}(v)$. ■

Claim: $\delta(s, v) \geq \text{level}(v)$.

Proof: Prove by contradiction. Suppose among all $v$ such that $\delta(s,v) < \text{level}(v)$, find the one with smallest $\delta(x,v)$.




Let $(s \cdots uv)$ be the path with fewest hops form $s$ to $v$.

$$\delta(s,u) = \delta(s,v) - 1$$

since $\delta(s,u) < \delta(s,v)$ and $\delta(s,v)$ is the smallest hops which less than level,

$\delta(s,u)$ must be larger than level$(u)$, i.e.

$$\text{level}(u) \leq \delta(s,u) = \delta(s,v) - 1$$

$v$ is the neighbor of $u$,

$$\text{level}(v) \leq \text{level}(u) + 1 \leq \delta(s,v)$$

But we assume $\delta(s,v) < \text{level}(v)$, so we meet a contradiction the claim is true.

**notes**

- $\text{level}(v) \leq \text{level}(u) + 1$.

  Because $v$ can be discovered early than discovering from $u$.

■

### 3.2.4 Remarks

Look at edges on which new vertex are discovered. These edges from a tree and the discover sequence becomes the direction between vertices. Think of the tree are directed away from $s$, even though the graph is undirected.

Why it is a tree? Each vertex can be discovered at most one time. Every vertex at most have one in-coming edge then the graph is acyclic.

Will the tree contains all the vertices in the graph?

No. The graph could have several isolated components. This is the expected result by design because BFS is used to find the shortest path from the source. So we do not want to find the vertex in different components because there is no path from the vertex to the source.

Will all the vertices in the same component with $s$ be discovered by BFS? Prove by contradiction. There are some vertices in the component of $s$ that BFS starts which are not discovered. Among all such vertices,

choose the one with the fewest hops away from the source.

## 3.3 Depth-first Search (DFS)

The usage of DFS is broader than BFS. Explore one path as far as it will go, back track as little as needs. Repeat.

### 3.3.1 Basics

**3 possible states for each vertex:**

- Undiscovered
- Discovered: Just find out this vertex and have not check its neighbors.
- Finished: discover the node and all its neighbors.

**Data Structure**

- BFS: Order of exploration of vertices is the same as the order of discovery. It is a FIFO, therefore we use queue to track discovered vertices.
- DFS: Last discovered vertex is the first to be fully explored. It is a LIFO, therefore we use a stack to track the discovered vertices.

You get a stack for free by using recursion because during the recursion the OS maintains a stack by itself.

Therefore, we can think about DFS as a recursive algorithm.

### 3.3.2 Algorithm

$DFS(G, s)$

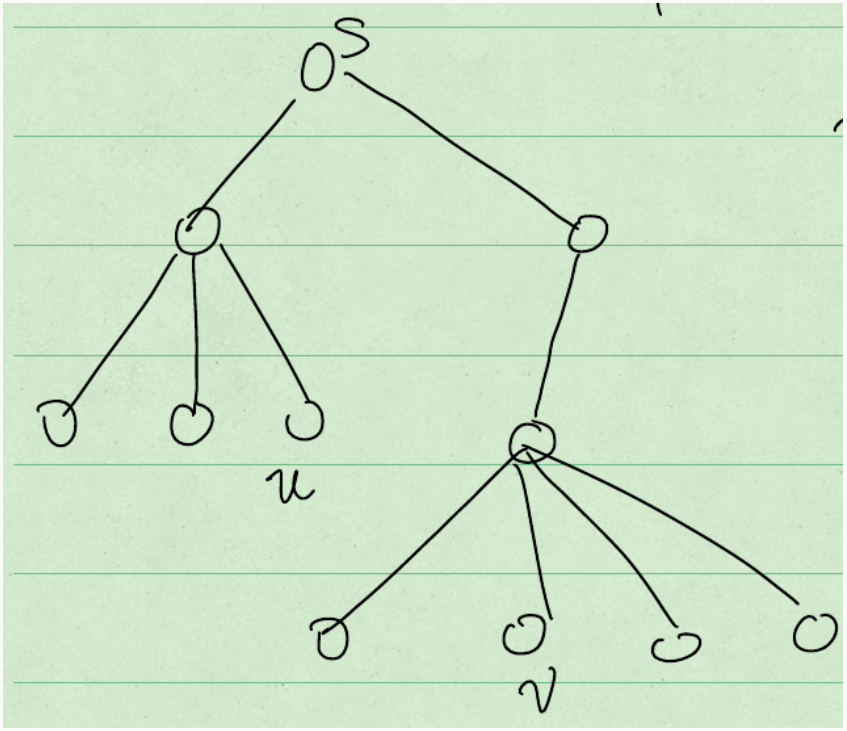
- Mark all vertices as undiscovered.
- Start with s and mark $s$ as discovered.
- While there is an undiscovered neighbor $u$ of $s$ call $DFS(G, u)$.
- Mark $s$ as finished.

### 3.3.3 DFS Tree

Formed by the set of edges on which new vertices are discovered.

**Theorem 3.2** *If $(u, v)$ is an edge in the graph that is not in the DFS tree. Then $u$ is an ancestor of $v$ or $v$ is an ancestor of $u$ in the DFS tree.*

**Proof.** Those edges are called "back edges". In an undirected graph you can only have tree edges and back edges.




This cannot exist because $v$ would have been discovered from $u$.

### 3.3.4 Time

Suppose $u$ is a vertex in a undirected connected graph $G$.

- Time advance at each call and return of DFS function. The range of time is $[1, 2n]$.

**Notes:** Every time you finish a recursive call of DFS, time advance by 1. Every time you make a new recursive call time advance by 1. Time is discrete and start from 1 at the first DFS call to $s$ and advance by 1 every time you finish or start a new recursive call.

- $s(u)$: **The start time** of $u$ is the time at which DFS of $u$ is invoked.
- $f(u)$: **The finish time** of $u$ is the time at which the DFS of $u$ is finished.

### 3.3.5 Duration

It is generally true for recursive function that the duration of an vertex $u$ is the integral from the start to the finish time.

$$\text{Duration}(u) = [s(u), f(u)]$$

$\forall u, v$, it is not the case that

$$s(u) < s(v) < f(u) < f(v)$$

Recursive algorithm ensures that it does not happen since it is last in first out. $v$ should finish first before $u$.

### 3.3.6 Lineage

- **Ancestor:** If you have a rooted tree, $u$ is call ancestor of $v$ if $u$ is on the path from $v$ to the root.
- **Descendant:** If $v$ is an ancestor of $u$, then $u$ is the descendant of $v$.
- By convention, we include itself as ancestor so we allow $v$ to be its own ancestor as well as its own descendant.

**Theorem 3.3** *Suppose u is discovered before v and there is a path* ***of undiscovered vertices*** *between u and v. Then v will become a descendant of u.*

**Proof.** Prove by contradiction. Suppose it is not the case and look at the first vertex on the path from u to v of undiscovered vertices that dose not become the descent of $u$.

Look at the first vertex to violate the rule would be a good way to prove a contradiction.

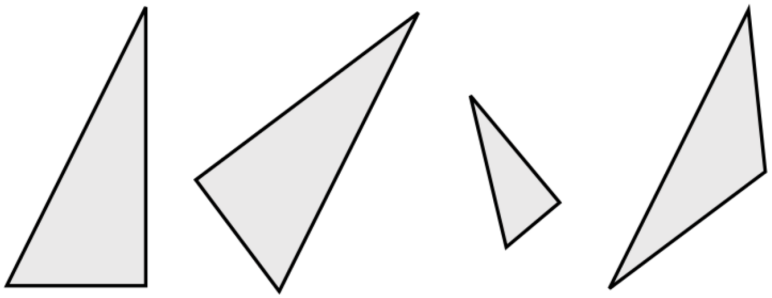
## 3.4 Notes

BFS is used to find the shortest path from the source in the graph therefore we start BFS from the source until we exhaust the component where the source lies.

DFS is also used to get a global picture of the entire graph and we often want DFS to be deterministic enough. We start from the source and do DFS. We only discover the vertices in the connected component of that source. But we want to discover other vertices. So we have a outer layer to restart the DFS from some undiscovered vertices in some other component every time we finish the search within one component. In this way, DFS is able to cover all the vertices and all the components. Therefore, often you only need to give DFS a graph without any source explicitly.

# 4 Equivalence Relation

- A relation $R$ on a set $A$ is an equivalence relation if it is reflexive, symmetric, and transitive.
- When the elements of some set $A$ have a notion of equivalence defined on them, then one may naturally split the set $A$ into equivalence classes.
- If $R$ is an equivalence relation on the set $A$, its equivalence classes form a partition of $A$.
- In each equivalence class, all the elements are related and every element in $A$ belongs to one and only one equivalence class.
- The relation $R$ determines the membership in each equivalence class, and every element in the equivalence class can be used to represent that equivalence class.
- In a sense, if you know one member within an equivalence class, you also know all the other elements in the equivalence class because they are all related according to $R$.
- Conversely, given a partition of $A$, we can use it to define an equivalence relation by declaring two elements to be related if they belong to the same component in the partition.



Congruence is an example of an equivalence relation. The leftmost two triangles are congruent, while the third and fourth triangles are not congruent to any other triangle shown here. Thus, the first two triangles are in the same equivalence class, while the third and fourth triangles are each in their own equivalence class.

Ref:

- equivalence
- Equivalence Class

# 5 Undirected Application: Bi-connectivity

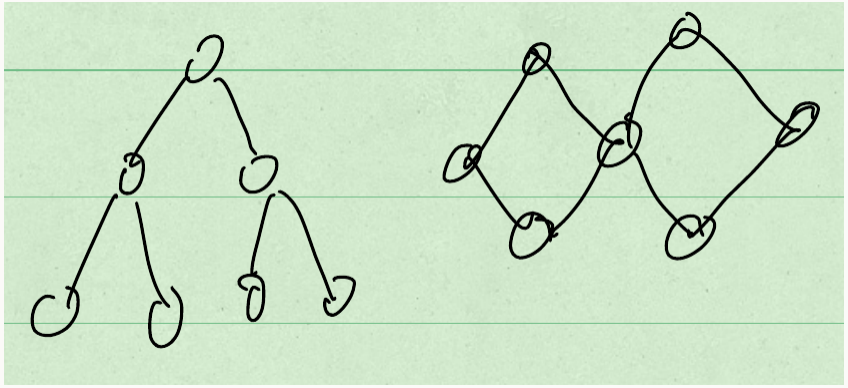
Connectivity is important but sometimes it is not enough and we want some robust connectivity. So we introduce **Bi-connectivity**.

## 5.1 Definitions

### 5.1.1 Bi-connectivity

- A graph is called biconnected if the removal of any one vertex leaves it connected.
- A graph is called k-connected if the removal of any $k - 1$ vertices leaves it connected.

Examples of connected graph which is not bi-connected.



### 5.1.2 Articulation Point

A vertex in an undirected connected graph is an **articulation point** (or cut vertex) **iff** removing it (and edges through it) disconnects the graph.

Articulation points represent vulnerabilities in a connected network-single points whose failure would split

the network into 2 or more disconnected components.

### 5.1.3 Bi-connected Components

Bi-connected Components are maximal pieces of the graph that are bi-connected.

### 5.1.4 Vertex-disjoint Path

Two paths are vertex-independent (alternatively, internally **vertex-disjoint**) if they do not have any internal vertex in common.

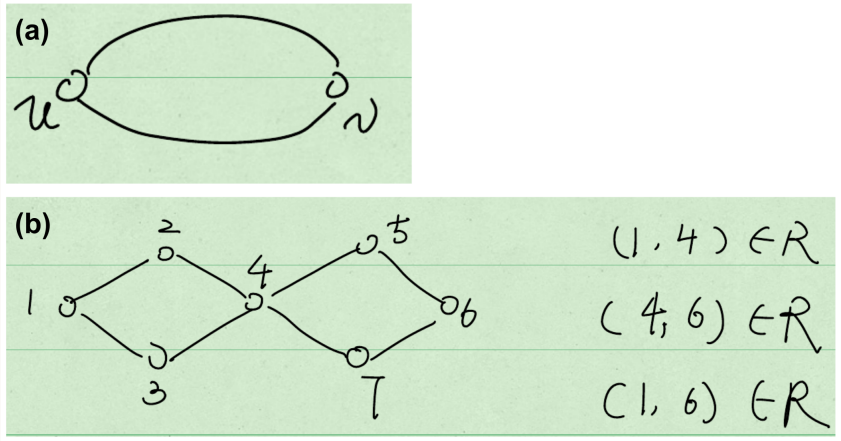
Similarly, two paths are edge-independent (or edge-disjoint) if they do not have any internal edge in common. Two internally vertex-disjoint paths are edge-disjoint, but the converse is not necessarily true.

### 5.1.5 Relations on edges

How to define the relation between two vertices?

**Proposal:** If there are two vertex-disjoint paths between $u$ and $v$, then $(u, v) \in R$. See example (a) in the following graph.

The proposal does not work because the relation is not Transitive. In other words, if $(u, v) \in R$ and $(v, w) \in R$ it is not always the case that $(u, w) \in R$. See example (b) in the following graph. $(1, 6) \in R$ is a false statement.




**The relation $R$ on edges defined as follows:**

- For any edge $e$, $(e, e) \in R$.
- If $e_1, e_2$ are two distinct edges, then $(e_1, e_2) \in R$ iff there is a simple cycle that passes through $e_1, e_2$.

Assume $R$ is an equivalent relation on edges. The property of equivalent relation is it divides the set on which it is defined into partitions of equivalent class.

Assume $R$ is an equivalence relation on edges. R partitions edges into equivalence class $E_1, E_2, \cdots, E_k$.

For a sanity check, if $k = 1$ then the graph is a bi-connected graph. Define component $G_i$ as the graph consist of edges $E_i$ together with a vertex set $v_i$ consist of the every point of $E_i$.

A vertex belongs to more than one components iff it is not clean.

## 5.2 Find Articulation Points

Assume $G$ is connected, start DFS at node $s$, so $s$ is the root of DFS tree.

### 5.2.1 Check root of DFS Tree

**Case 1:** Suppose $s$ has one child in DFS tree. In this case, $s$ is not an articulation point since the removal of $s$ won't disconnect the graph. See example in (a).

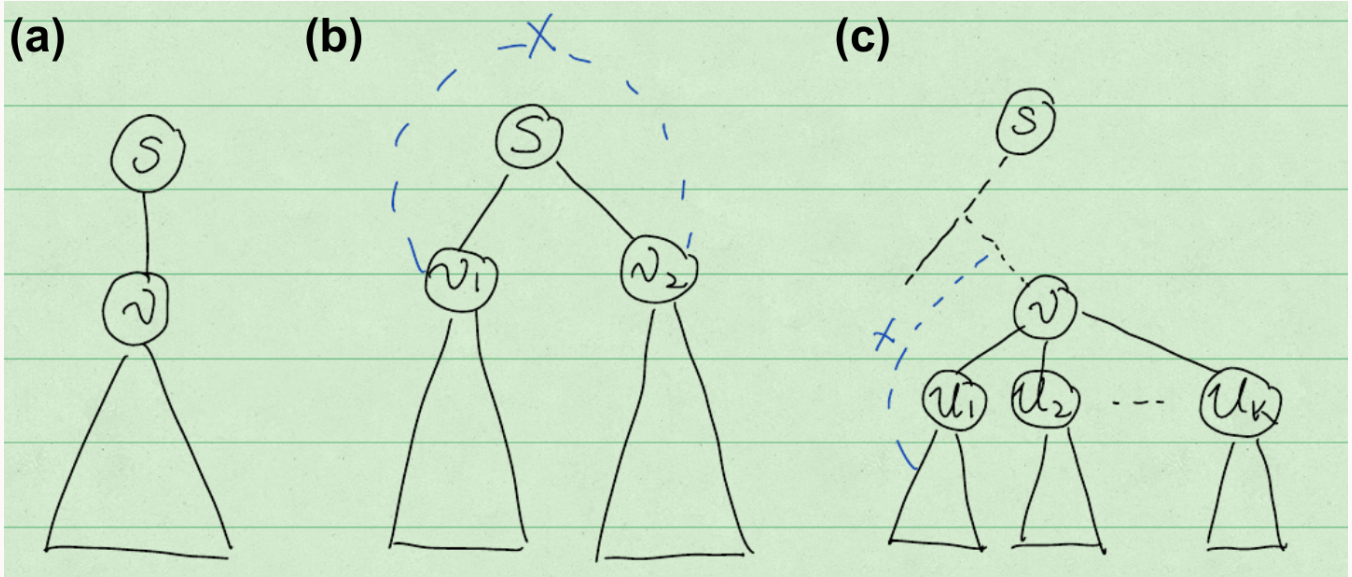
**Case 2:** Suppose $s$ has more than 1 child in DFS tree. $s$ is an articulation point because back edges only go between ancestor and descendant. The nodes in subtree $T(v_1)$ and $T(v_2)$ are not ancestor nor descendant between each other. Therefore all the paths connecting $T(v_1)$ and $T(v_2)$ must go through $s$. Hence $s$ is an articulation point. See example in (b).

### 5.2.2 Check Interior Nodes

Note that no leaf can be articulation point because moving a leaf the DFS tree is still connected.

Assume $v$ is an internal node and has $k$ leaves. What are the conditions for $v$ to be an articulation point?

Claim: An internal node $v$ is an articulation point **iff** it has a child $u_i$ such that here is no back edge from the subtree rooted at $u_i$ to a proper ancestor of $v$. See example in (c).




## 5.3 DFS: Find Bi-connected Components.

**Main idea:** Store visited edges in a stack while DFS on a graph and keep looking for Articulation Points.

- As soon as an Articulation Point $u$ is found, all edges visited while DFS from node $u$ onwards will form one biconnected component.
- When DFS completes for one connected component, all edges present in stack will form a biconnected component.
- If there is no Articulation Point in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.

### 5.3.1 How to check articulation point?

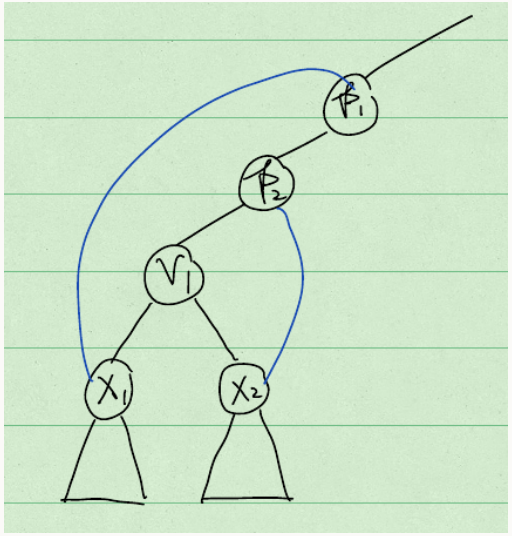
Claim: If $u$ is an ancestor of $v$ in the DFS tree, then $s(u) < s(v)$.

**Note:** smaller number of start time $s$ means higher up of the tree.

How the algorithm works? What does DFS keep track of?

We want to know which vertices as we go back out of the subtree could be articulation points. The higher a back edge goes, the more vertices are ruled out to be articulation points. So **keep track of the highest back edge going out from every subtree**. Hence we are able to find the highest ancestor that an edge going through from the subtree. Then we know every vertex along the way can not be articulation point because of this subtree. (But still, other subtrees may cause them to be an articulation point.)

This is where the postorder comes in. In the example, suppose $x_1$ goes to vertex $p_1$ which is the highest back edge to reach from any where of $T(x_1)$. So the smallest number of start time reached from $T(x_1)$ is $s(p_1)$. From $x_2$ we have some edge goes up to vertex $p_2$, which is the highest back edge to reach from any where of $T(x_2)$.




**Definition 5.1** ***low(x)*** *is the smallest* ***start time*** *of vertices reachable by a back edge from subtree rooted at x.*

Suppose $v$ has just two child $x_1$ and $x_2$,

$$\text{low}(v) = \min(\text{low}(x_1),\ \text{low}(x_2),\ \text{s}(u) : (v,u) \text{ is a back edge})$$

What make $v$ an articulation point?

Claim: For any node $v$ that is not the root, $v$ is an articulation point **if and only if**

there exist a child $u_i$ of $v$ such that $\text{low}(u_i) \geq s(v)$.

**Notes:** The claim means the child of $v$ will not send any back edge to any where above $v$. Sending back edges only to $v$ and below means removing $v$ the whole subtree will fall apart.

## 5.4 Wrapper of DFS

DFS is used to search the entire graph. Usually DFS is wrapped by another function to traverse all the vertices in the graph.

DepthFirstSearch($G$):

While there is an unexplored vertex $x$,

dfs($G$, $x$)

### 5.4.1 Time

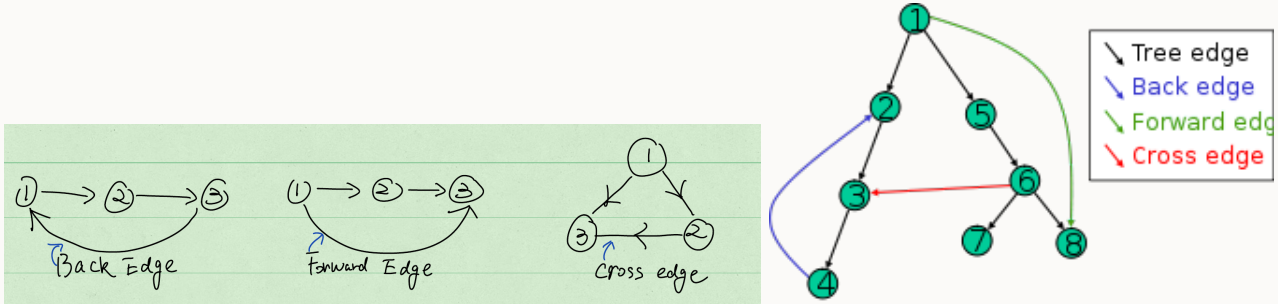
In this case, we still have start time and finish time continue all the way though the outside layer. We start from inner layer of one $s$ and you might advance to certain point of time then find out that you have no more vertex to visit. So we go back to the outer loop start DFS at some other point and keep the time going. So the start time and finish will still between 1 and $2n$.

# 6 Directed Graph

In directed graph, a edge is an ordered pair.

## 6.1 Edges

- Tree edges: edges where new vertices are discovered.
- Back edges: $(3 \to 1)$
- Forward edge $(1 \to 3)$ could be forward edge if DFS went $(1 \to 2 \to 3)$.
- Cross edge




Claim: textbfRestriction of cross edge: If $(u, v)$ is a cross edge, then $s(u) > s(v)$. In other words, $v$ is discovered before $u$.

Proof: By contradiction, if $(u, v)$ is a cross edge and $u$ is discovered before $v$, we cannot let DFS left $u$ and

discover $v$ without go though $(u, v)$. Therefore, $v$ will be a child of $u$ instead of a node in other DFS tree. ■

## 6.2 Directed Acyclic Graph (DAG)

### 6.2.1 Maximum Number of edges

- How many edges a undirected **acyclic** graph could have at most? $N - 1$
- How many edges a directed **acyclic** graph could have at most? $\frac{N(N-1)}{2}$

### 6.2.2 Algorithm

- Input: Directed acyclic graph $G$.
- Goal: Solve topological sort problem by finding the topological order. In other words, find an order of the vertices such that for each edge $(u, v)$, $u$ precedes $v$ in order.

**Theorem 6.1** *Any directed acyclic graph has a topological order.*

**Proof.** Show every DAG has some node with no incoming edges. Then remove the node from the graph and get a smaller DAG. Then use induction to prove the theorem is true.

Claim: Every DAG has some node with no incoming edge.

Proof: Given a graph DAG and reverse the directions of all the edges. Then we want to argue that there is no

outgoing edge in the graph.

For contradiction, suppose the argument is not true and every node has some outgoing edge. Then start walking from some vertex $v$ and this walking will keep on continuing since every vertex has some outgoing edge. But the graph is finite. Continuing walking on a finite graph means there must be cycles in the graph. We meet a contradiction since the graph is DAG. ■

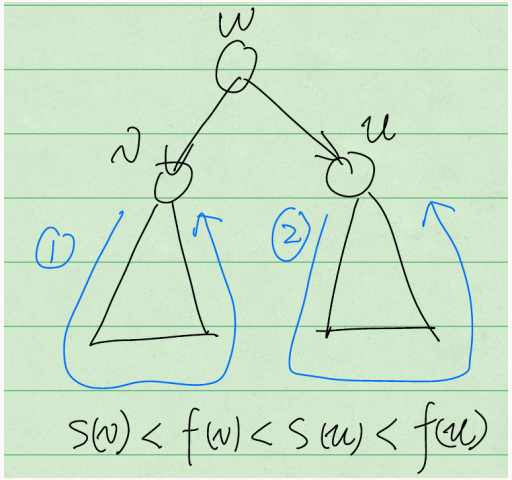
The first principle algorithm for topological sorting is to track the in-degree of every vertex, which is the number of edges come in to the vertex. First output the vertex $v$ with in-degree. Then go to the adjacency list of $v$ and decrease the in-degree of every neighbor by 1. Remove $v$ from the graph and repeat.

Solution based DFS.

**Theorem 6.2** *If $G$ is a DAG and $(u,v)$ is an edge, then on any DepthFirstSearch($G$), $f(u) \geq f(v)$.*

**Proof.** Prove by cases analysis.

- Case 1: $s(u) < s(v)$. $u$ is discovered before $v$. Then $v$ will become a descendant of $u$ (may not be child).
- Case 2: $s(u) > s(v)$. $v$ is discovered before $u$.
  - If we discovered $u$ in the course of DFS($v$), then there must be a path from $v$ to $u$, which means $G$ has a cycle. $\Rightarrow\Leftarrow$
  - If $u$ is not in the subtree of $v$ as the following example, then obviously $f(v) < f(u)$.




So,

$$s(v) < f(v) < s(u) < f(u)$$

Therefore the algorithm should be:

TopologicalSort($G$):

DepthFirstSearch($G$):

Output vertices in reverse order of finish time.

## 6.3 Strongly Connected Graph

### 6.3.1 Basics

Give a directed graph $G = (V, E)$, what is the correct connectivity relation?

We want the relation to be equivalent. The edge between two node is not symmetric relation in directed graph. Therefore we create a equivalent relation by setting symmetric in the definition.

**Definition 6.1** $R = \{(u, v) : \textit{There are paths from } u \textit{ to } v \textit{ and from } v \textit{ to } u\}$

$R$ is reflective, symmetric and transitive. So $R$ is an equivalent relation on the vertices.

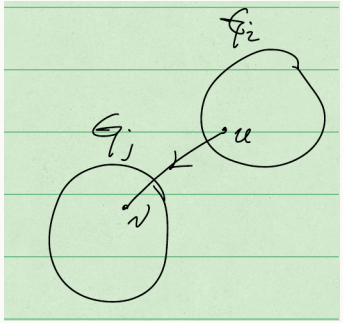
**Definition 6.2** *$R$ partitions $V$ onto $V_1$, $V_2$, $\cdots$, $V_k$. (Equivalence Classes)*

*If $V_1 \subset V$, which is the vertex set of $G = (V, E)$, it induces the sub-graphs*

$$G_{V_i} = (V_i,\ E \cap (V_i \times V_i))$$

*For simplicity, call $G_{V_i}$ as $G_i$. $G_i$'s are* ***the strongly connected components*** *of $G$.*

It is possible to have edges connecting different strongly connected components. See example as follows.

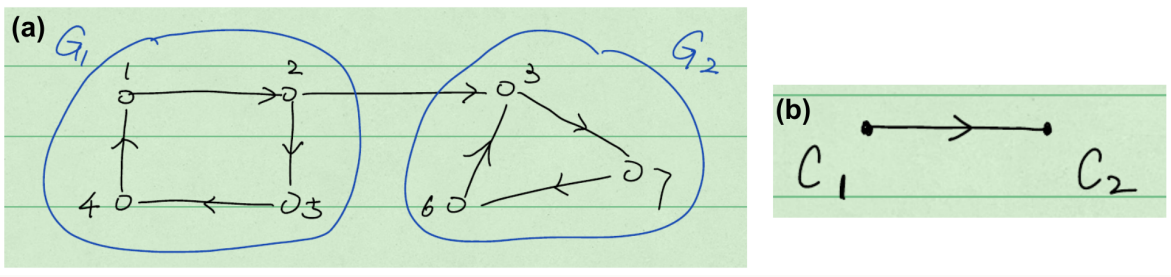


**Definition 6.3** *The **strongly connected graph** of directed graph $G(V,E)$ is $G_{SCG}(V_{SCG}, E_{SCG})$, where*

- $V_{SCC} = \{c_i : G_i \text{ is a strongly connected components}\}$.
- $E_{SCC} = \{(c_i, c_j) : \exists v_i \in G_i \text{ and } v_j \in G_j,\ (v_i, v_j) \in E\}$

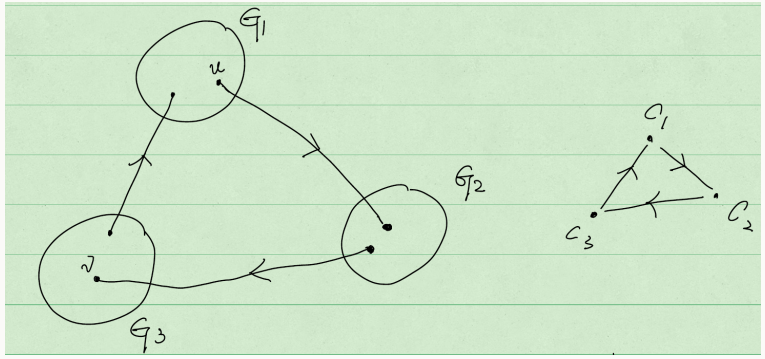
Here $G_{SCC}$ is actually a quotient graph.

Example, $(a)$ in the following plot is the Graph $G$ and $(b)$ is the strongly connected graph of $G$. The equivalence class of $G$ is $\{\{1,2,3,4\},\{3,6,7\}\}$.




**Theorem 6.3** *$G_{SCC}$ is acyclic.*

**Proof.** The components we have found are the maximal with respect to the relation on both side to reach other. Prove the theorem by contradiction. Suppose we have a cycle as the following plot. Because there must be a path from $u$ to $v$ and vice versa. Then , we are able to merging $G_1$, $G_3$. In fact, all three components can be merged together.




Many algorithm on directed graph works on first decomposing the graph into strongly connected components.

DFS decomposes a directed graph into SCC's.

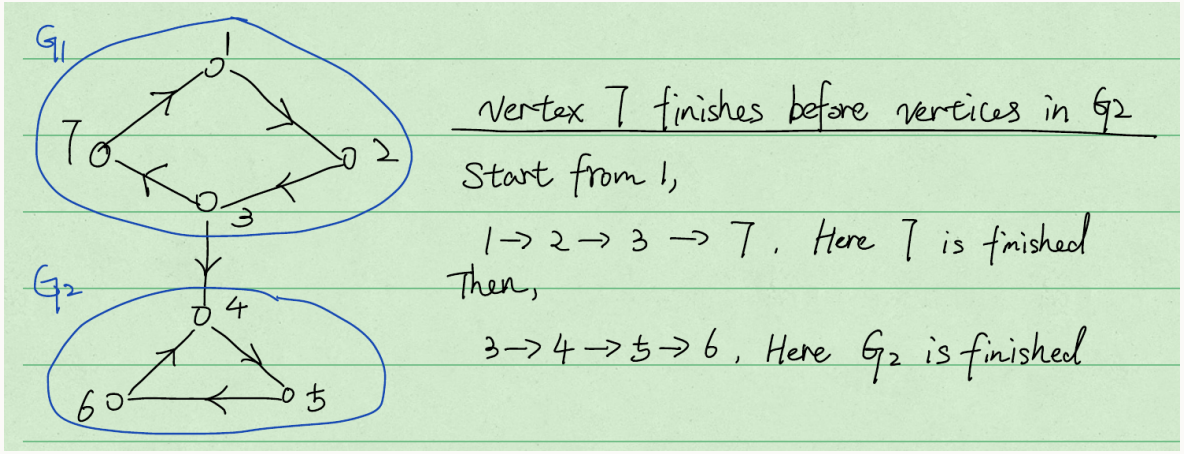
### 6.3.2 Finish Time

Now we first gave a **false theorem** and make it right.

**Theorem 6.4** ***[False Theorem]***

*Suppose $G$ is an arbitrary graph and $G_{SCC}$ is a DAG. If $c_i$, $c_2$ are nodes corresponding to two SCC in $G_{SCC}$ and $(c_i, c_2) \in E_{SCC}$, then in DFS(G), every vertex in $G_1$ finish after every vertex in $G_2$.*

**Proof.** Prove the theorem is false by a counter example as follows:




**Theorem 6.5** ***[Updated Theorem]***

*Suppose $G$ is an arbitrary graph and $G_{SCC}$ is a DAG. If $c_i$, $c_2$ are nodes corresponding to two SCC in $G_{SCC}$ and $(c_i, c_2) \in E_{SCC}$, then in DFS($G$), **there exist a vertex** in $G_1$ finish after every vertex in $G_2$.*

**Proof.** TODO: Read the proof in the book.

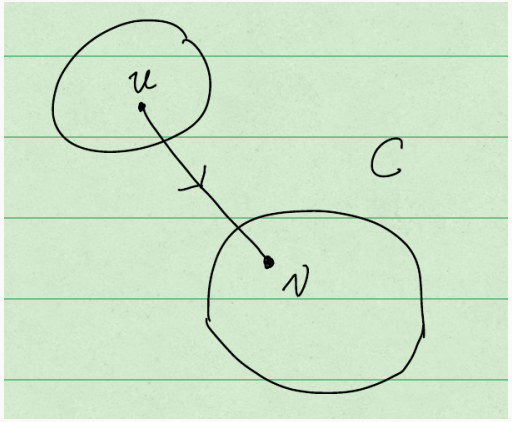
### 6.3.3 Kosaraju's Algorithm

**Definition 6.4** *$G_{SCC}$ is a DAG. So, $G_{SCC}$ has one or more vertices with no in-coming edges. These vertices are called **sources**.*

Claim: If $v$ is the vertex with largest $f(v)$ in a DFS($G$). Then $v$ must lies in the source component.

Proof: Prove by contradiction.

If $v$ does not lie in the source component but in some other component $C$. So $v$ must have some in-coming edge to make it not a source. That means that some vertex $u$ must finish later than everything in the component $C$ because of the theorem of finishing time. Therefore $v$ could not be the latest finishing vertex.




■

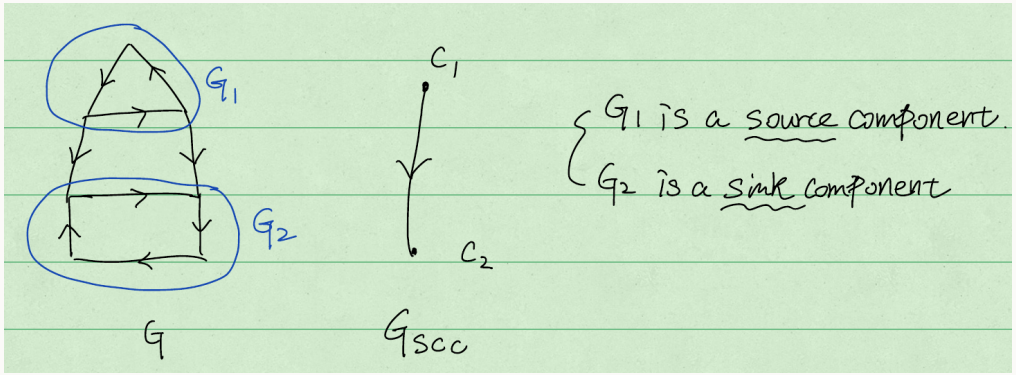
**Kosaraju Algorithm**

- One DFS(G) to record the nodes of finish times. Let $f_1(v)$ be the finish time of $v$ in this DFS.
- Reverse all the edges in $G$ to create the graph $G^R$.
- Do DFS($G^R$) with one constrain:

- In the outer procedure, always choose the undiscovered vertex $v$ with the greatest $f(v)$.

**Notes**

- The vertex $v$ with greatest $f_1(v)$ lies in a sink component of $G^R$.
- $G$ and $G^R$ have the same strongest connected component DFS($G^R$) start at $v$ will only discover the SCC containing $v$.
- When the inner call to DFS($G^R$, $v$) finishes, output all the discovered vertices as one SCC of $G$.
- When we return to the outer loop and pick the undiscovered vertex with the greatest finish time, we know inductively that we will find all SCC's.




**Running Time** Two DFS's and reverse the edges in the graph. All the steps are linear. So this is a $O(m+n)$ algorithm. $m, n$ are the numbers of vertices and edges.