Graph Theory

1. Applications:
   - World Wide Web
   - Scheduling
   - Chip Design
   - Network Analysis
   - Flow Charts

2. Definitions:
   1. A graph $G=(V,E)$ consists of a set of vertices (nodes), denoted by V, and a set of edges, denoted by E.

   2. $n=|V|$. I.e. n is the number of nodes.

   3. $m=|E|$. I.e. m is the number of edges.

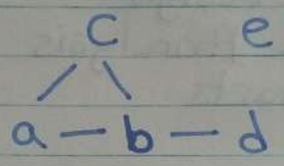
   4. In an undirected graph, each edge is a set of 2 vertices, $\{u,v\}$. This makes (u,v) and (v,u) the same. Furthermore, self-loops are not allowed.
      Note: When it's clear from context, we will use (u,v) for $\{u,v\}$.

   5. In a directed graph, each edge is an ordered pair of nodes. Therefore, (u,v) is different from (v,u). Furthermore, self-loops are allowed. This means that (u,u) is allowed.

Hilroy

6. Two vertices are adjacent iff there is an edge between them.

E.g. Consider the graph below.



We can store which nodes are adjacent in 2 ways.

1. Adjacency Matrix

	a	b	c	d	e
a		✓	✓		
b	✓		✓	✓	
c	✓	✓			
d		✓			
e					

- An adjacency matrix is a 2-D array.

- Space: $\Theta(n^2)$
- who are adjacent to v: $\Theta(n)$ t
- are v and w adjacent: $\Theta(1)$ t
- Convenient for some other operations and queries.

2. Adjacency Lists

	Is adjacent to
a	b, c
b	a, c, d
c	a, b
d	b
e	

- With adjacency lists, we store the vertices in a 1-D array or dictionary. At entry A[i], we store the neighbours of v_i.

- If the graph is directed, we store only the out-neighbours.

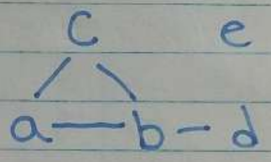
- Space: $\Theta(m+n)$
- who are adj to v:
  $\Theta(\deg(v))$ time
  I.e. Length of adj list
- are v and w adj:
  $\Theta(\deg(v))$ time if a list

- Optimal for graph searches.

7. **Traversal**: Visit each vertex of a graph.

8. **Path**: A sequence of edges which connect a sequence of distinct vertices.
I.e. You can't go through a vertex twice.

E.g. Consider the graph below.



<d> is a path of length 0.
<d, b, c> is a path of len 2.
<d, a, b> is not a path.

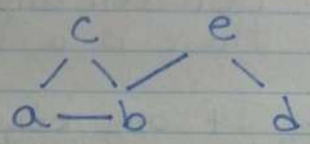
9. v is **reachable** from u iff there is a path from u to v.

10. A **simple cycle** is a non-empty sequence of vertices in which:

1. Consecutive vertices are adjacent
2. First Vertex = Last Vertex
3. Vertices are distinct, except for the first and last
4. Edges used are distinct

Note: $\langle v \rangle$ is NOT a cycle.

E.g. Consider the graph below.



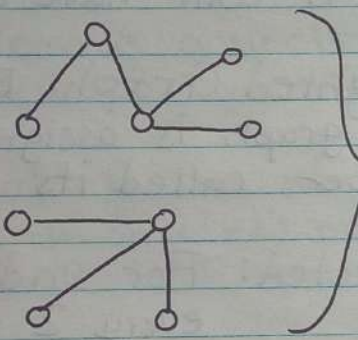
1. $\langle b, c, a, b \rangle$ is a simple cycle of length 3.

2. $\langle b, c, a, b, d, e, b \rangle$ is not a simple cycle.

3. $\langle b, d, b \rangle$ is not a cycle because it uses $\{b, d\}$ twice.

11. A tree is a graph that is connected but has no cycles.

E.g.



Trees

A forest is a collection of trees.

Note: Acyclic means that there are no cycles.

Trees have the following properties:

1. Between any 2 vertices, there is a unique path.

2. A tree is connected by default, but if an edge is removed, it becomes disconnected.

3. # edges = # vertices -1
   I.e. $m = n - 1$

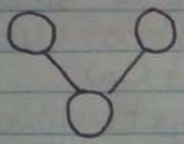
4. Acyclic by default, but if a new edge is added, then it will have a cycle.

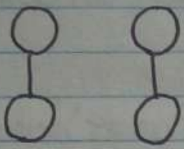
12. Weighted Graph: Each edge in the graph is assigned a real number, called its weight.

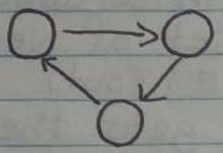
13. Connected: For undirected graphs, every 2 vertices have a path between them.

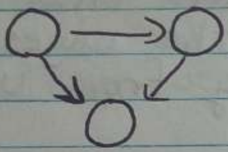
14. Strongly Connected: For directed graphs, for any 2 vertices, u, v, there is a directed path from u to v.

E.g.

1.  Connected

2.  Not Connected

3.  Strongly Connected

4.  Not Strongly Connected

3. Operations:

1. Add / Remove a vertex/edge.

2. Edge Query: Given 2 vertices, u, v, find out if the edge (u,v) (if the graph is directed) or the edge $\{u,v\}$ is in E.

3. Neighbourhood: Given a vertex u in an undirected graph, get the set of vertices $\{v \mid \{u,v\} \in E\}$.

4. In-neighbourhood: Given a vertex u in a directed graph, get the set of vertices $\{v \mid (v,u) \in E\}$ (in).
   I.e. This gets the set of vertices whose edges lead to u.

5. Out-neighbourhood: Given a vertex u in a directed graph, get the set of vertices $\{v \mid (u,v) \in E\}$ (out).
   I.e. This gets the set of vertices that can be reached by the edges that lead away from u.

6. Degree: Computes the size of the neighbourhood.

7. In-Degree: Computes the size of the in-neighbourhood.

8. Out-Degree: Computes the size of the out-neighbourhood.

4. Breadth-First Search (BFS):

1. Algorithim:

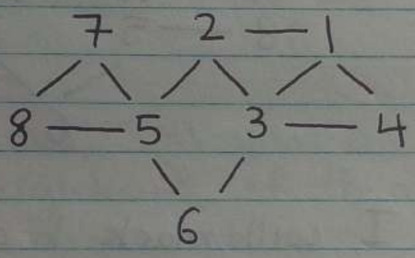
   1. Start at v. Visit v and mark as visited.

   2. Visit every unmarked neighbour of v and mark each neighbour as visited.

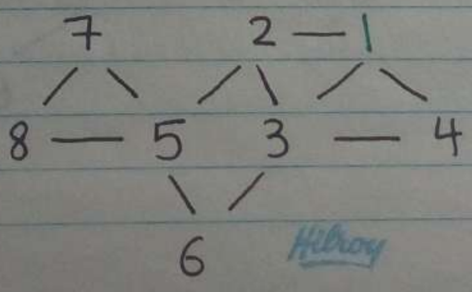
   3. Mark v finished.

   4. Recurse on each vertex marked as visited in the order they were visited.

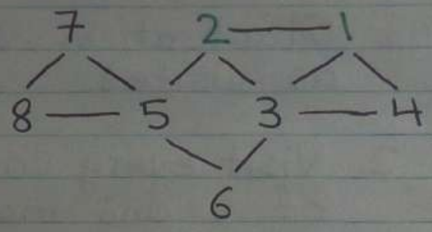
E.g. Consider the graph below.

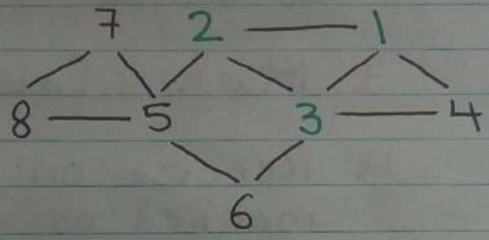


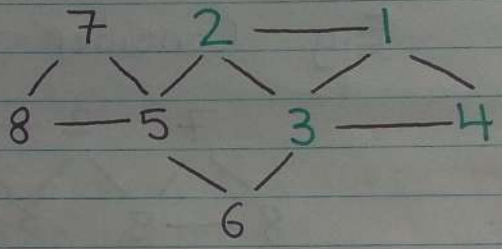
1. Start at 1. I'll mark a node as visited by writing it in green.



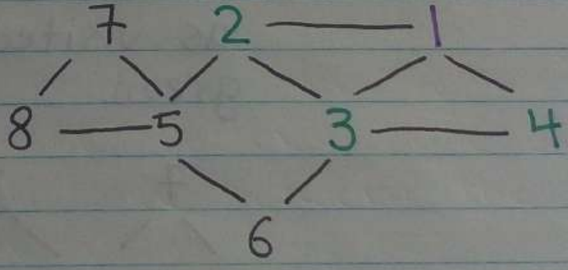
2. I will visit all the unmarked neighbours of 1 in the following order: 2, 3, 4.



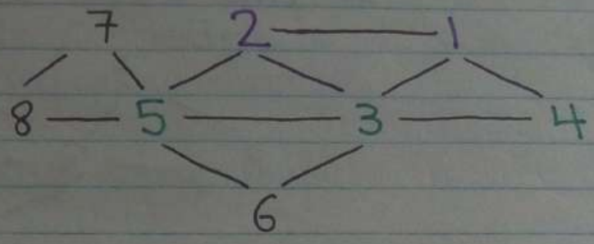




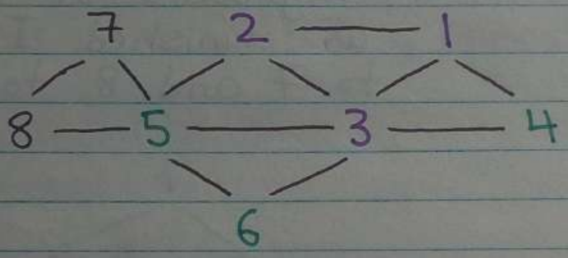
3. I will mark 1 as finished by writing it in purple.



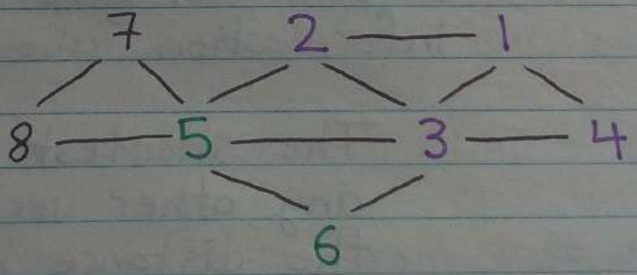
4. Since I visited 2 first, I will visit every unmarked neighbour of 2 and then mark 2 as finished.



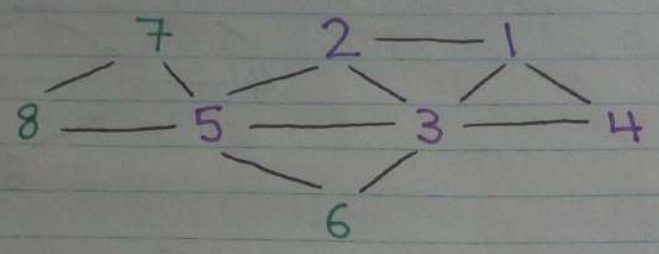
5. Next, I will visit all the unmarked neighbours of 3 and then mark 3 as finished.



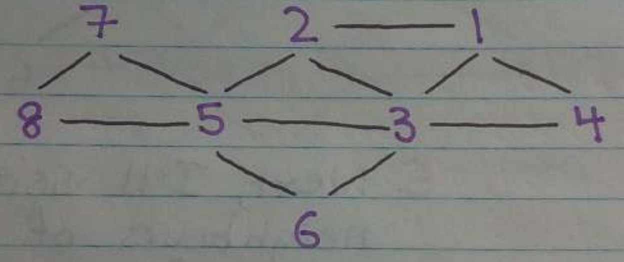
6. Next, I'll visit all the unmarked neighbours of 4 and then mark 4 as finished.



7. I'll visit all the unmarked neighbours of 5 and then I'll mark 5 as finished.
I'll visit the unmarked neighbours in this order: 7, 8.



8. I'll visit all the unmarked neighbours of 6, and mark it as finished. I'll do the same to 7 and 8, too.



A BFS can give the following information about a graph.

1. The shortest path from v to any other vertex u. We denote the distance between the nodes as d(v).

2. Whether the graph is connected.

3. The number of connected components.

A BFS constructs a tree that visits every node connected to v. We call this a spanning tree.

2. Implementing BFS:

We can use a queue to implement a BFS given an adjacency list representation of a graph.

A queue is FIFO (First in, First out) and has the following operations:

1. Enqueue(Q, v)
2. Dequeue(Q)
3. Isempty(Q)

Furthermore, we will need to store the following information for each v:

1. The current node, u, and it's state (visited, not visited, finished)

2. The predecessor, p[u]

3. The distance from u to v.

4. The order of discovery

3. Complexity:

- Since each node is enqueued at most once, the adjacency list of each node is examined at most once. Therefore, the total running time of BFS is $O(m+n)$ or linear in the size of the adjacency list.

Note: Each node is enqueued when it is not visited, at which point it is marked visited.

Note:
- BFS will only visit the nodes that are reachable from V.
- If the graph is connected (In the undirected case) or strongly-connected (In the directed case), then this will be all vertices.
  - If not, then we may have to call BFS multiple times in order to see the whole graph.

5. Depth First Search (DFS):

1. Algorithim:

- All vertices and edges start out unmarked.
- Start at vertex v and go as far as possible away from v visiting vertices.
- If the current vertex has not been visited, mark it as visited and the edge that is traversed as a DFS edge.
- If the current vertex has been visited, mark the traversed edge as a back-up edge, back up to the previous vertex.
- When the current vertex has only visited neighbours left, mark it as finished.
- Backtrack to the first vertex that is not finished.
- Continue

Just like BFS, DFS constructs a spanning-tree and gives connected component information.

However, DFS does not find the shortest distance between v and all other vertices.

2. Implementing A DFS:

- We can use a stack (LIFO) to store the edges with the usual operations:

  1. push((u,v))
  2. pop()
  3. is_empty()

- Furthermore, we need to store these data for each vertex in order to easily determine whether an edge is a back-edge or a DFS-edge:

  1. d[v] will indicate the discovery time.
  2. f[v] will indicate the finish time.

3. Complexity of DFS:

- A DFS visits the neighbours of a node exactly once. Therefore, the adjacency list of each vertex is visited at most once. So, the total running time is $\Theta(n+m)$.
  I.e. Linear in the size of the adjacency list.

- Note: The DFS edges form a tree called the DFS tree. However, the DFS tree is NOT unique for a given graph G, starting at s.

4. DFS Edges:

- We can specify edges (u,v) in a DFS-tree according to how they are traversed during the search.

- If v is visited for the first time, then (u,v) is a tree-edge in a DFS tree.

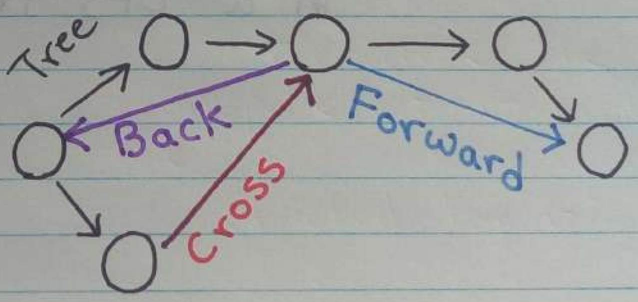
- If v has already been visited, then (u,v) is a:

1. back-edge: An edge from a vertex u to an ancestor v in the DFS tree.

2. forward-edge: An edge from a vertex u to a descendent v in the DFS tree.
   Note: This only applies to directed graphs.

3. cross-edge: All the other edges that are not part of the DFS tree.
   I.e. v is neither an ancestor nor a descendent of u in the DFS tree.
   Note: This only applies to directed graphs.

- E.g.



- We can use d[v] and f[v] to distinguish between the edges.

- There is a cycle in graph G iff there are any back-edges when DFS is run.

- We can detect a back-edge in a DFS' if the vertex we are visiting has been visited but not finished.

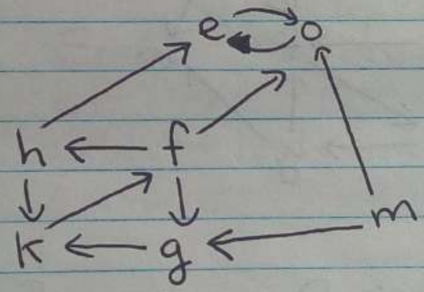
6. Strongly Connected Components (SCC):

1. Definition:

- SCC: Is the maximal subset of vertices from each other in a directed graph.

- Example:

Consider the graph below.



The scc's are:

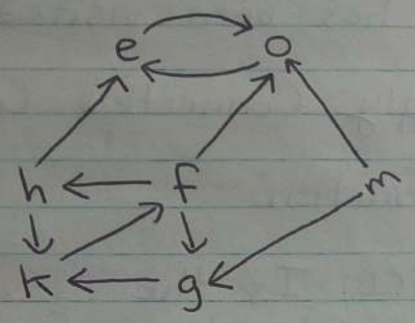
1. {e, o}
2. {m}
3. {h, f, k, g}

2. Transpose of G:

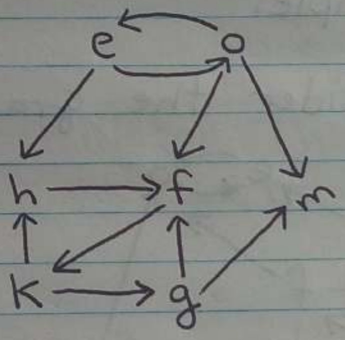
- The transpose of G, denoted by G^T, is a graph with the same vertices as G, but the edges are reversed.

- E.g.

G:



G^T:



- Note: Do not confuse the transpose of G with the complement of G, denoted by G^c.

The complement of G is all possible edges minus all the existing edges.

- Note: G^T has the same SCC as G.

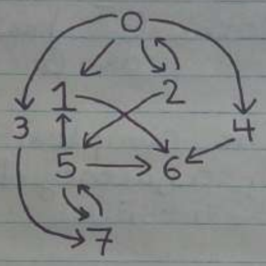
- The complexity of computing an adjacency list of G^T is $O(|V| + |E|)$.

7. Kosaraju's SCC Algorithim:

1. Overview:
   - DFS on G. Visit all the vertices, note finish times and accumulate vertices in reverse finishing order.
   - Compute the adjacency lists of G^T.
   - DFS on G^T, using the above order to pick start/restart vertices.
   - Each tree found has the vertices of one SCC. In total, this takes $O(|V| + |E|)$ time.

2. Example:

Consider the graph below.



1. Start at 0 and go to 1.
I will store the visited nodes in a list and the finished vertices in a separate list.

visited: 0, 1
finished:

2. From 1, I will go to 6.

visited: 0, 1, 6
finished:

3. Since there is nowhere to go from 6, I will mark it as finished and backtrack to 1.

visited: 0, 1, 6
finished: 6

4. Since there is nowhere to go from 1, I will mark it as finished and backtrack to 0.

visited: 0, 1, 6
finished: 6, 1

5. From 0, I will visit 2.

visited: 0, 1, 6, 2
finished: 6, 1

6. From 2, I will go to 5.
Note: I cannot go to 0 because I have already visited it.

visited: 0, 1, 6, 2, 5
finished: 6, 1

7. From 5, I will go to 7.
Note: I can't go to 1 or 6 as I have visited them already.

visited: 0, 1, 6, 2, 5, 7
finished: 6, 1

8. There is nowhere to go from 7, so I will mark it as finished and backtrack to 5.

visited: 0, 1, 6, 2, 5, 7
finished: 6, 1, 7

4

9. There is nowhere to go from 5, so I will mark it as finished and back-track to 2.

visited: 0, 1, 6, 2, 5, 7
finished: 6, 1, 7, 5

10. There is nowhere to go from 2, so I will mark it as finished and backtrack to 0.

visited: 0, 1, 6, 2, 5, 7
finished: 6, 1, 7, 5, 2

11. From 0, I will visit 3.

visited: 0, 1, 6, 2, 5, 7, 3
finished: 6, 1, 7, 5, 2

12. There is nowhere to go from 3, as I have already visited 7, so I will mark it as finished and backtrack to 0.

visited: 0, 1, 6, 2, 5, 7, 3
finished: 6, 1, 7, 5, 2, 3

13. From 0, I will visit 4.

visited: 0, 1, 6, 2, 5, 7, 3, 4
finished: 6, 1, 7, 5, 2, 3

4

14. There is nowhere to go from 4, so I will mark it as finished and backtrack to 0.

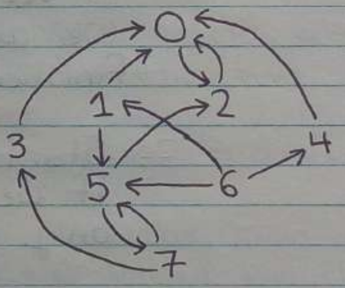
visited: 0, 1, 6, 2, 5, 7, 3, 4
finished: 6, 1, 7, 5, 2, 3, 4

15. There is nowhere to go from 0, so I will mark it as finished.
Note: The first node visited is also the last node visited.

visited: 0, 1, 6, 2, 5, 7, 3, 4
finished: 6, 1, 7, 5, 2, 3, 4, 0

16. Now, we find G^T, reverse the finished list and DFS G^T based on the ordering of the new finished list.

G^T:



finished: 0, 4, 3, 2, 5, 7, 1, 6

Hilroy

17. Starting at 0, if we do a DFS, the only vertex we can reach is 2.

$\therefore$ SCC #1 = $\{0, 2\}$
Furthermore, we can remove 0 and 2 from the finished list.

18. Starting at 4, we see that there is nowhere to go.

$\therefore$ SCC # 2 = $\{4\}$
We can remove 4 from the finished list.

19. Starting at 3, we see that there is nowhere to go.

$\therefore$ SCC # 3 = $\{3\}$
We can remove 3 from the finished list.

20. Starting at 5, we see that if we do a DFS, we can only go to 7.

$\therefore$ SCC # 4 = $\{5, 7\}$
We can remove 5 and 7 from the finished list.

21. Starting at 1, we see that there is nowhere to go.
$\therefore$ SCC # 5 = $\{1\}$
We can remove 1 from the finished list.

22. Starting at 6, we see that there is nowhere to go.

$\therefore$ SCC #6 = $\{6\}$

$\therefore$ In total, SCC = $\{0,2\}$, $\{4\}$, $\{3\}$, $\{5,7\}$, $\{1\}$, $\{6\}$

3. Proof of kosaraju's Algorithim:

1. Notation:

- We denoted $f(v)$ as the time at which vertex v is finished.
- $f(u) < f(v)$ means u is finished before v.
- Let C be an SCC. We define $f(C)$ to be the time at which the last node in C finishes. Formally, $f(C) = \max_{v \in C} f(v)$

2. Lemma: If s is the first node in SCC C visited by DFS, then $f(C) = f(s)$.

Proof:
Since s is the first node in C visited by DFS, all vertices in C are not finished. Furthermore, since C is a SCC,

every vertex in C is reachable from s. That means there is a path from s to every vertex in C. Thus, every node will be finished when DFS returns. Since the last step of the DFS is to finish s, this means that s is finished only after all other vertices are finished. Therefore, $f(s) > f(v)$ for any $v \in C$. By the definition of $f(C) = \max_{v \in C} f(v)$, $f(C) = f(s)$.

3. Thm: Suppose we run DFS starting at each node in G. Let C_1 and C_2 be SCCs in G. If (u, v) is an edge in G where $u \in C_1$ and $v \in C_2$, then $f(C_2) < f(C_1)$.

Proof:
- Let x_1 and x_2 be the first vertices DFS visits in C_1 and C_2, respectively.
- By our lemma, $f(C_1) = f(x_1)$ and $f(C_2) = f(x_2)$. Therefore, we will show $f(x_2) < f(x_1)$.
- Note: x_2 is reachable from x_1, because there is a path from x_1 to u in C_1, across (u, v) and a path from v to x_2 in C_2.

However, x_1 is not reachable from x_2, since then x_1 and x_2 would be strongly connected, contradicting that they belong in different SCCs.

- We have 2 cases:

1. $DFS(x_2)$ is called before $DFS(x_1)$:

   - Since x_1 is not reachable from x_2, x_2 will finish before x_1.

     $\therefore f(x_2) < f(x_1)$, as wanted

2. $DFS(x_1)$ is called before $DFS(x_2)$:

   - When $DFS(x_1)$ is called, all nodes in C_1 and C_2 have not been visited, so there is a DFS path from x_1 to x_2.

   - When $DFS(x_1)$ returns, x_2 will be finished.

   - Since x_1 will be finished just before $DFS(x_1)$ returns, this means that x_1 finished after x_2, so $f(x_2) < f(x_1)$.

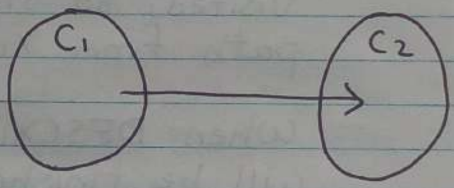
4. Corollary: Let C_1 and C_2 be distinct SCCs in $G = (V, E)$. Suppose there is an edge (u, v) in E^T where $u \in C_1$ and $v \in C_2$. Then $f(C_1) < f(C_2)$

5. Corollary: Let C_1 and C_2 be two distinct SCCs in $G = (V, E)$. If $f(C_1) > f(C_2)$, then there cannot be an edge from C_1 to C_2 in G^T.

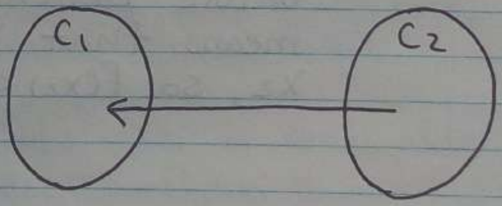
Consider this:

- Since we know that $f(C_1) > f(C_2)$, then there is an edge from C_1 to C_2. However, in G^T, that edge is reversed, so there is no longer an edge from C_1 to C_2.

G:



G^T:



This means that if we start the DFS on G^T at C_1, because there is no edge from C_1 to C_2, the DFS will only visit the vertices from C_1 and it will return a DFS tree that contains only vertices from C_1. Then, when you do DFS on C_2, even though there is an edge from C_2 to C_1, DFS will only visit the vertices in C_2 because we already finished C_1. We continue for all remaining SCCs.

Proof:

- Edge $(u,v) \in E^T$ implies $(v,u) \in E$.

- Since SCCs of G and G^T are the same, $f(C_2) > f(C_1)$.

- This completes the proof.