Connected components

CS-6th

Instructor: Dr. Ayesha Enayet

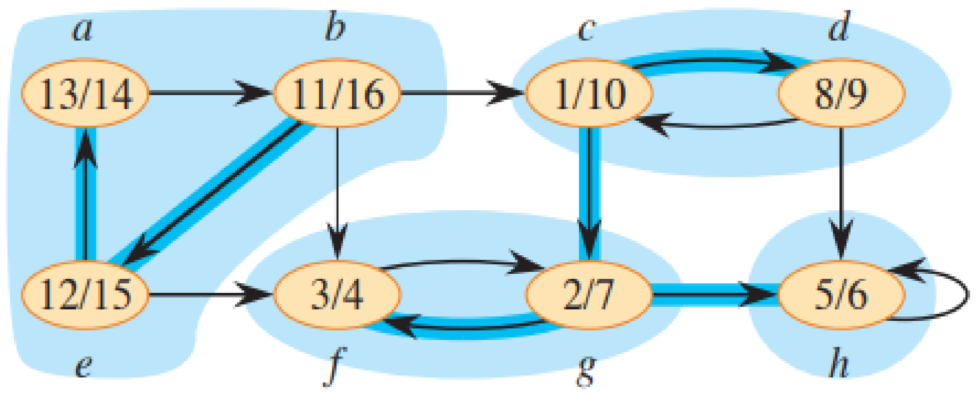
- Connected components→ undirected
- Strongly connected components → directed graphs

Strongly Connected Components (DFS)

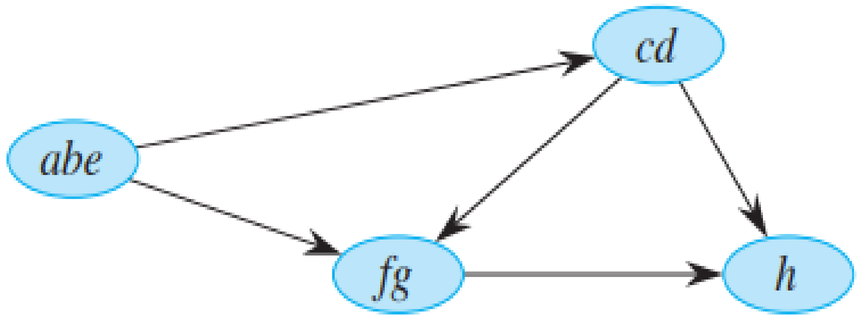
Definition

- 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 $u, v \in C$, both u and v are reachable from each other.

Example

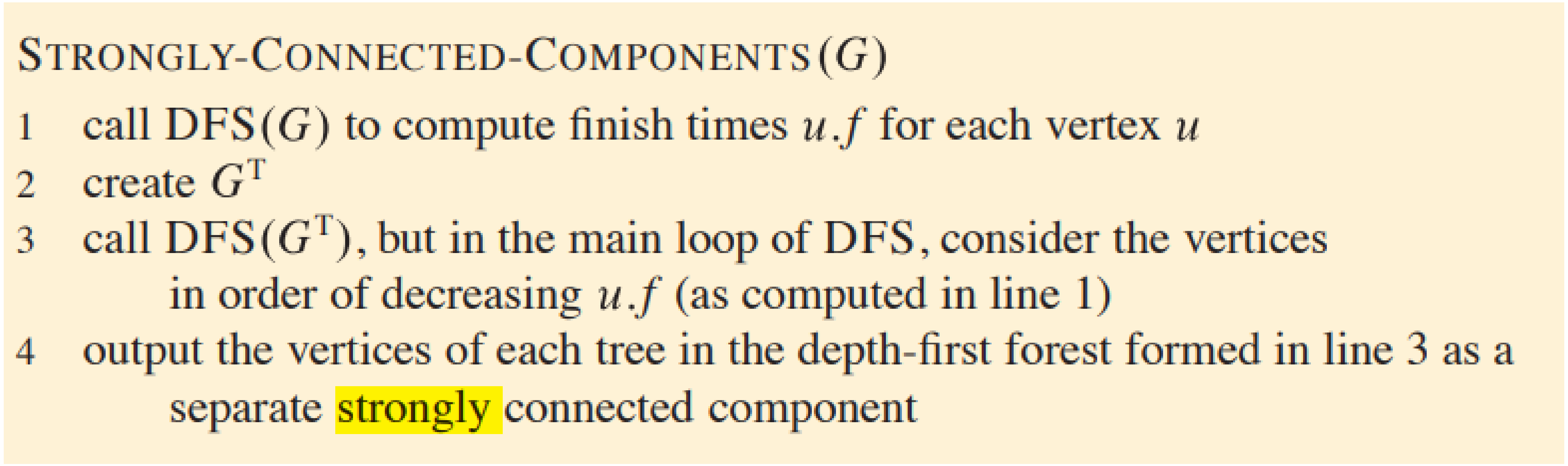


G=(V,E), where Each region shaded light blue is a strongly connected component of G



The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

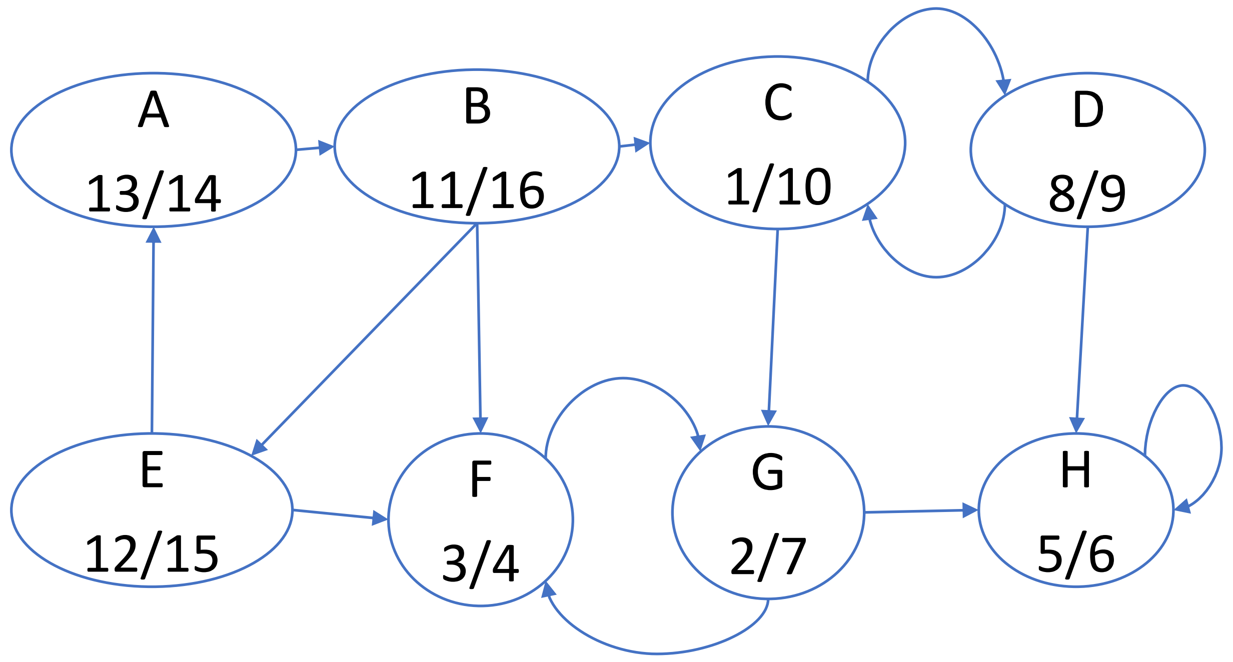
Algorithm (DFS)



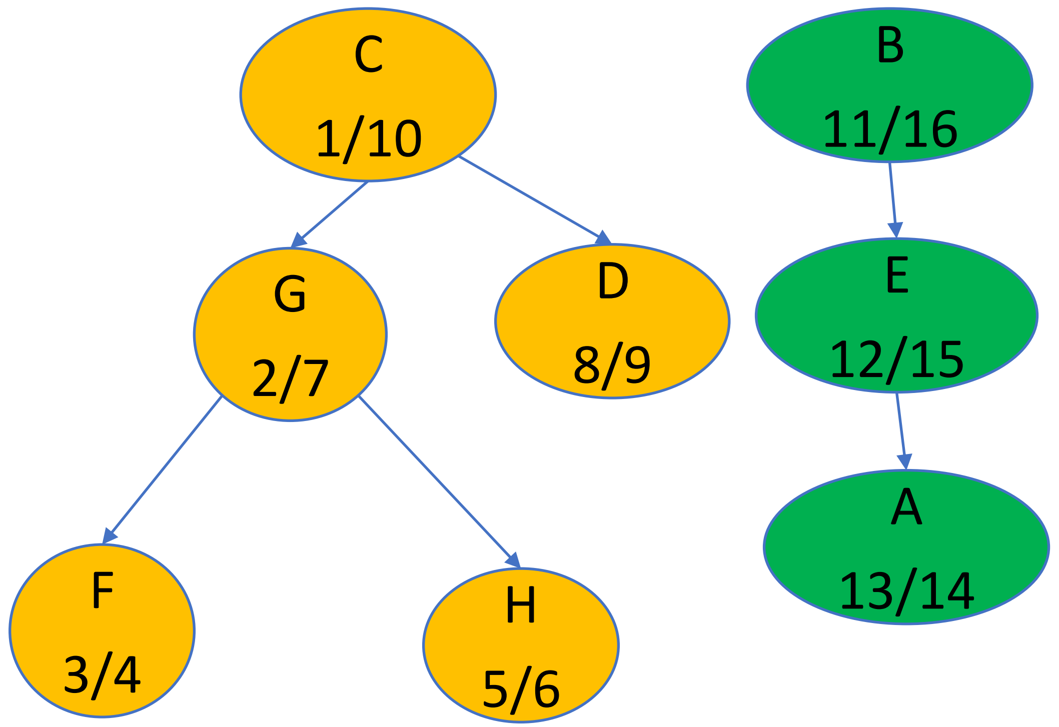
```
STRONGLY-CONNECTED-COMPONENTS($G$)
1  call DFS($G$) to compute finish times $u.f$ for each vertex $u$
2  create $G^T$
3  call DFS($G^T$), but in the main loop of DFS, consider the vertices
       in order of decreasing $u.f$ (as computed in line 1)
4  output the vertices of each tree in the depth-first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

Graph G (Example)

Graph G with discover and finish timestamps



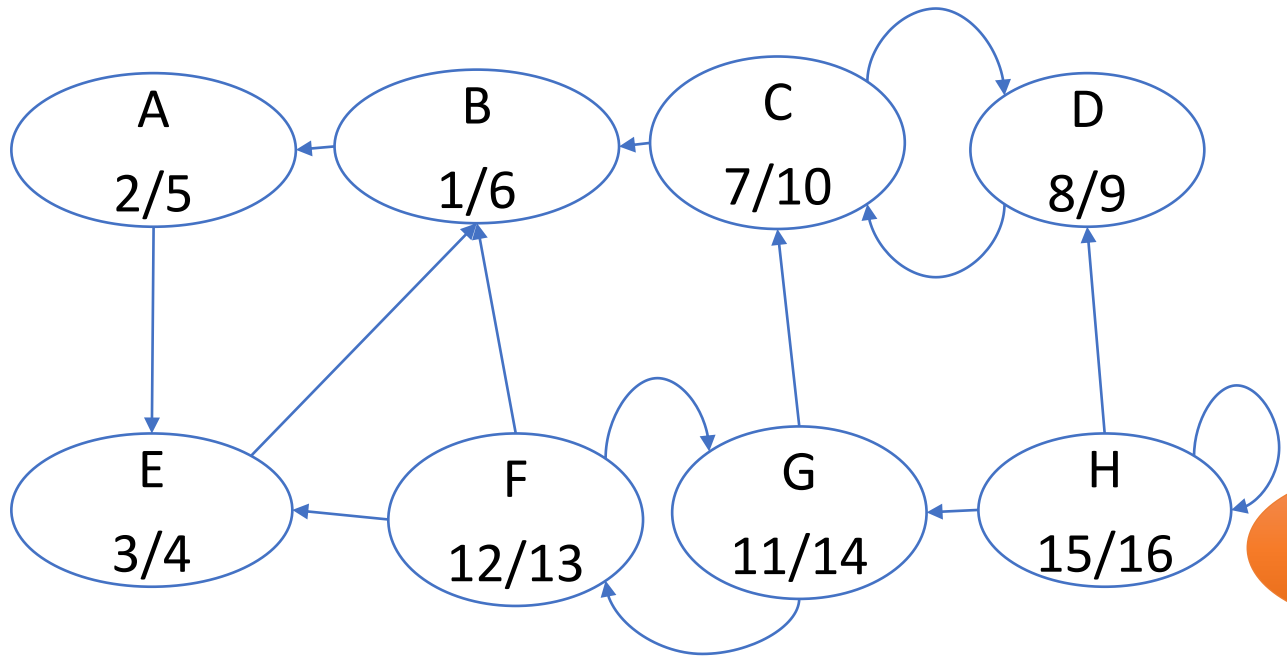
DFS forest

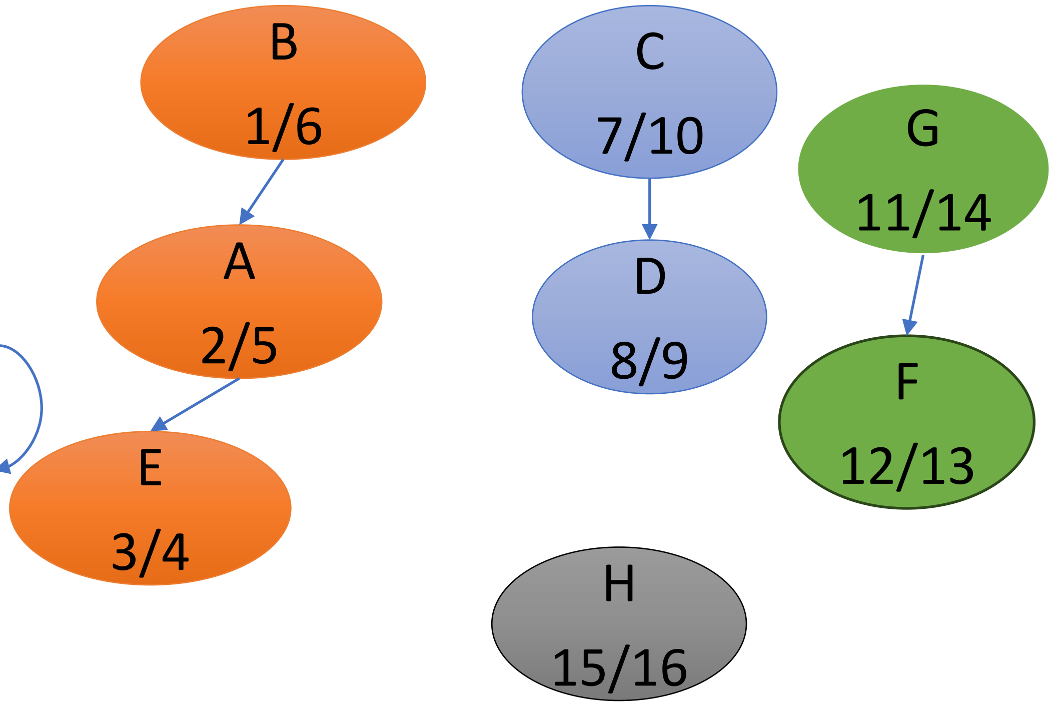


Graph G (Example)

Graph G^T with discover and finish timestamps

DFS forest





Theorem 20.16 The STRONGLY-CONNECTED-COMPONENTS procedure 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, and we consider the (k+1)st tree produced.
- Let the root of this tree be vertex u, and let u be in strongly connected component C. Because of how the depth-first search chooses roots in line 3, u.f=f .C> f .C’ for any strongly connected component C’ other than C that has yet to be visited.

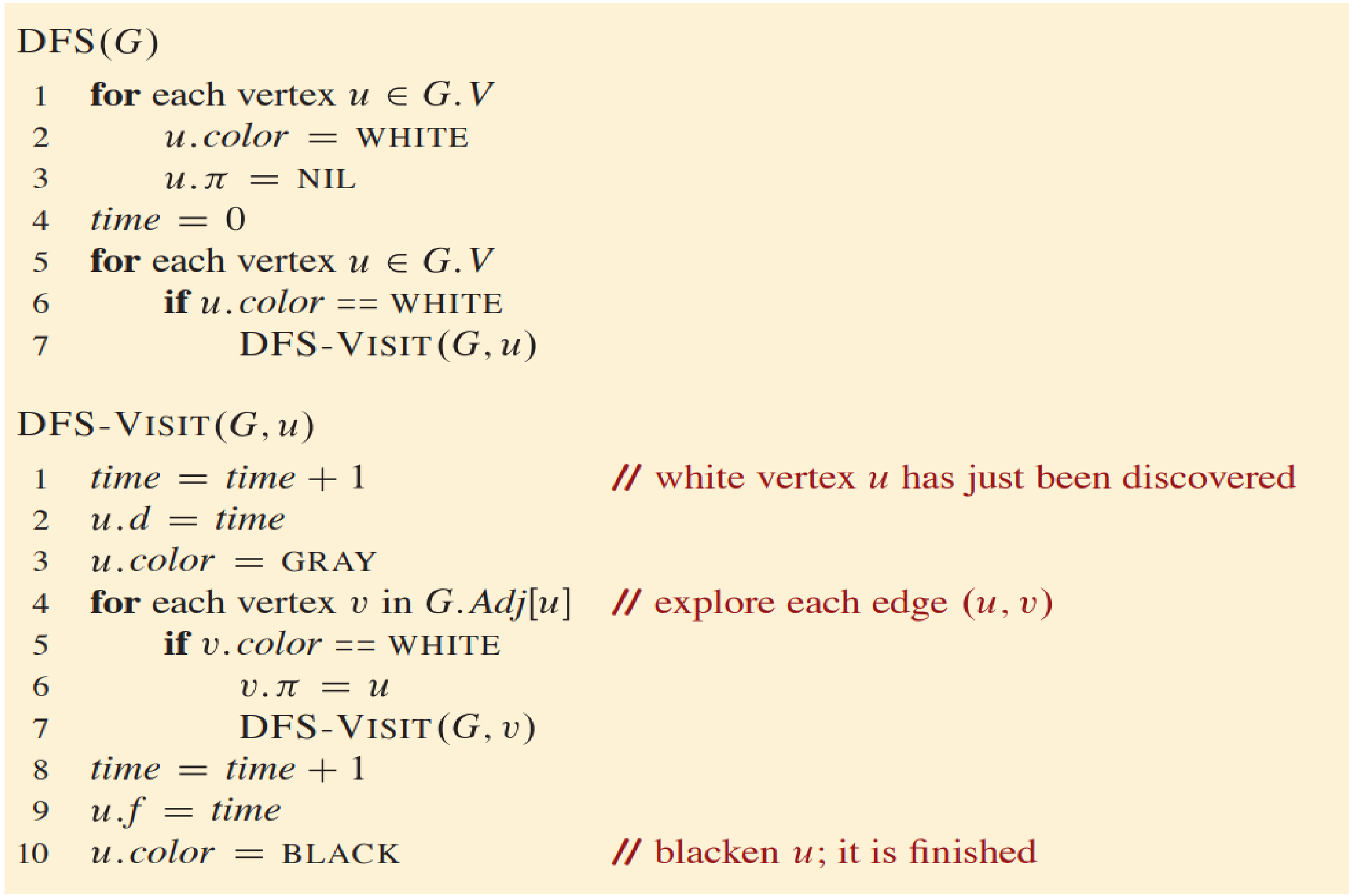
- By the inductive hypothesis, at the time that the search visits u, all other vertices of C are white. Therefore, all other vertices of C are descendants of u in its depth-first tree. Moreover, by the inductive hypothesis and by Corollary 20.15, any edges in G^T that leave C must be to strongly connected components that have already been visited.
- Thus, no vertex in any strongly connected component other than C is a descendant of u during the depth-first search of G^T . The vertices of the depth-first tree in G^T that is rooted at u form exactly one strongly connected component, which completes the inductive step and the proof.

Importance/Applications

- **Web Analysis:** SCCs help identify communities or clusters of related web pages or websites.
- **Social Network Analysis:** SCCs reveal closely connected groups of individuals or communities within social networks.
- **Compiler Optimization:** SCCs aid in optimizing code generation by identifying independent code blocks.

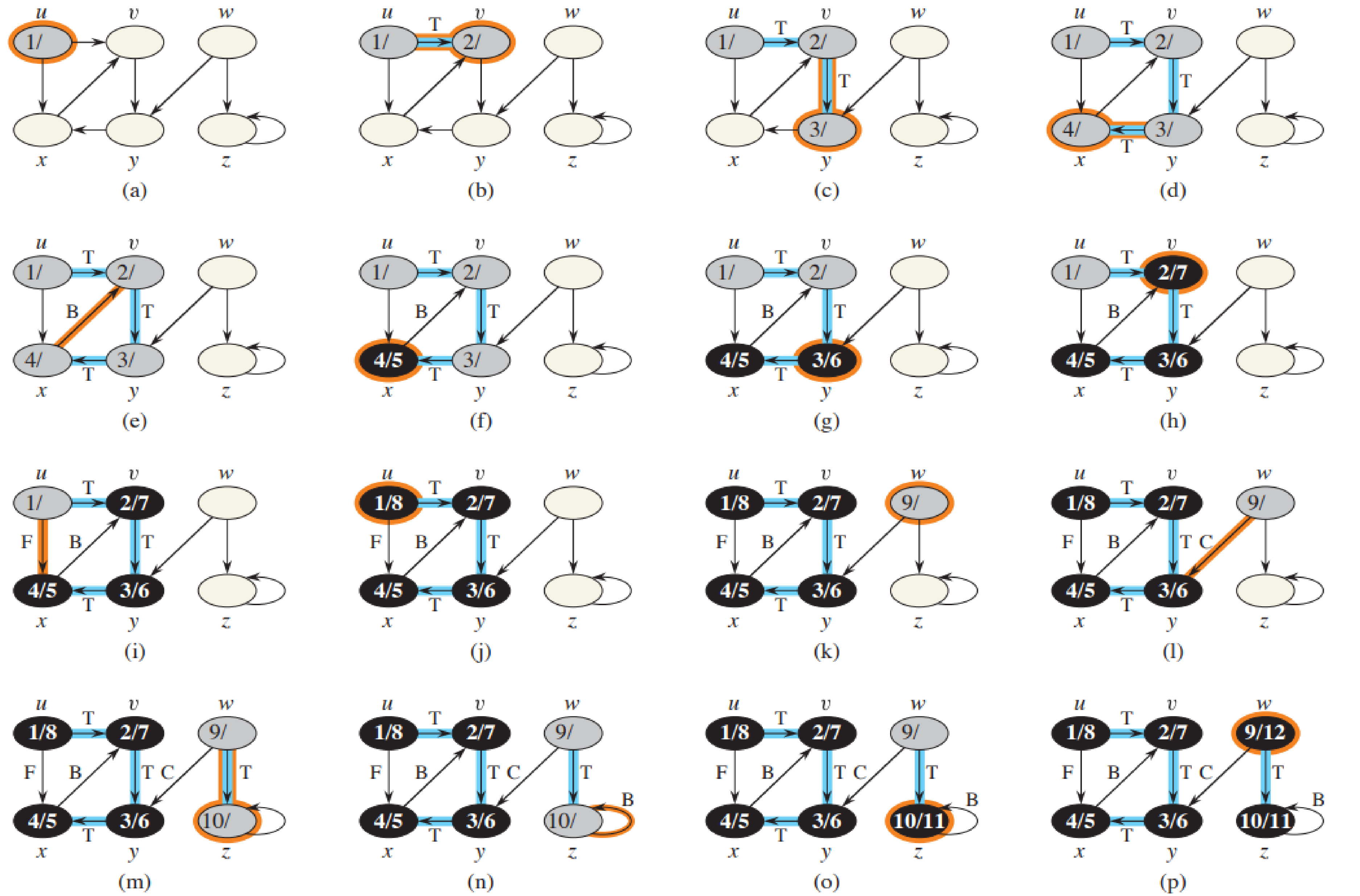
Reference: https://pratikbarjatya.medium.com/exploring-strongly-connected-components-and-the-kosaraju-algorithm-f36d70ec7c5d

Depth-First Search Algorithm



```
DFS($G$)
for each vertex $u \in G.V$
    $u.color$ = WHITE
    $u.\pi$ = NIL
$time = 0$
for each vertex $u \in G.V$
    if $u.color$ == WHITE
        DFS-VISIT($G, u$)

DFS-VISIT($G, u$)
$time = time + 1$                  // white vertex $u$ has just been discovered
$u.d = time$
$u.color$ = GRAY
for each vertex $v$ in $G.Adj[u]$  // explore each edge $(u, v)$
    if $v.color$ == WHITE
        $v.\pi = u$
        DFS-VISIT($G, v$)
$time = time + 1$
$u.f = time$
$u.color$ = BLACK                  // blacken $u$; it is finished
```



(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)

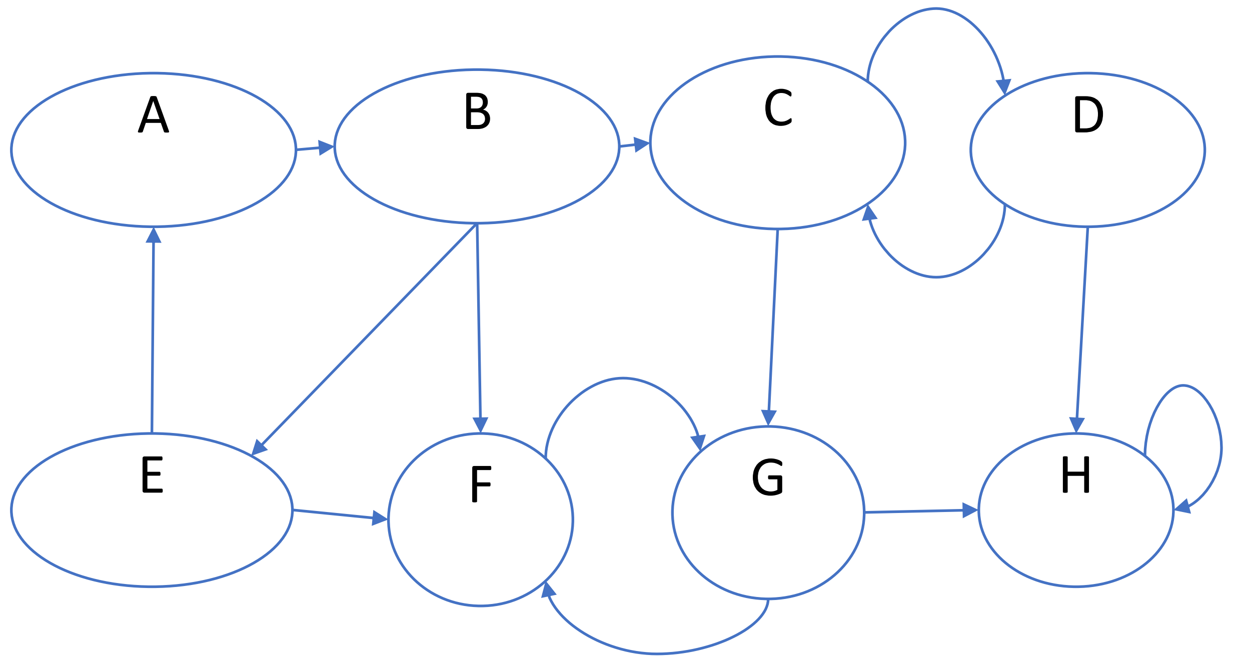
Time Complexity (DFS)

- Θ (V+E)
- The loops on lines 1-3 and lines 5-7 of DFS take O(V) time, exclusive of the time to execute the calls to DFS-VISIT.
- The procedure DFS-VISIT is called exactly once for each vertex v∈V , since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray.
- During an execution of DFS-VISIT (G,v), the loop in lines 4-7 executes Adj[v], where Adj[v]=O(E).
- The total cost is O(V+E)
- Note: The DFS algorithm is on the next slide.

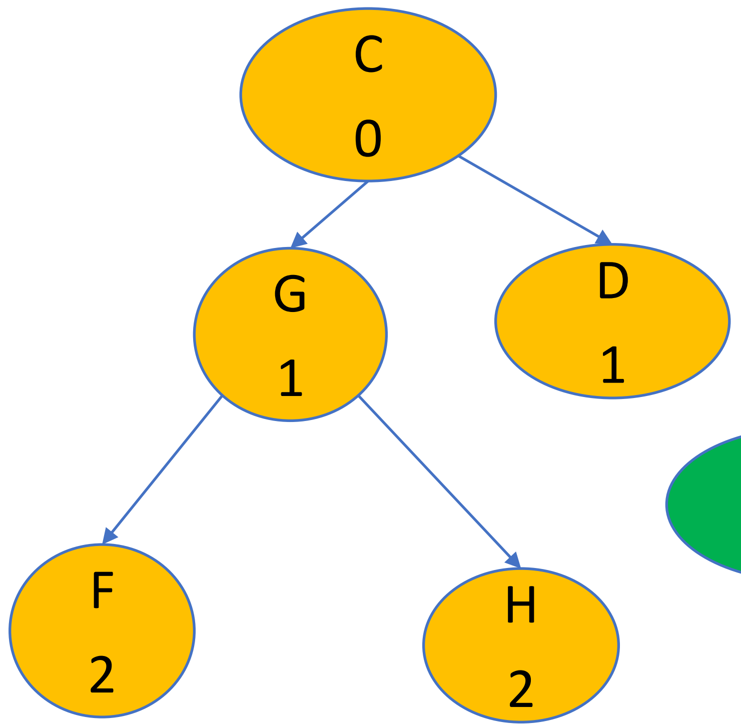
Connected components using BFS (directed):

Graph G (Example)

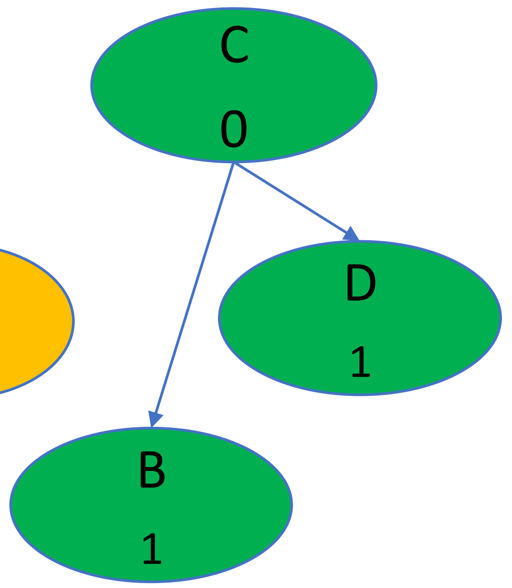
Graph G



BFS G, C1



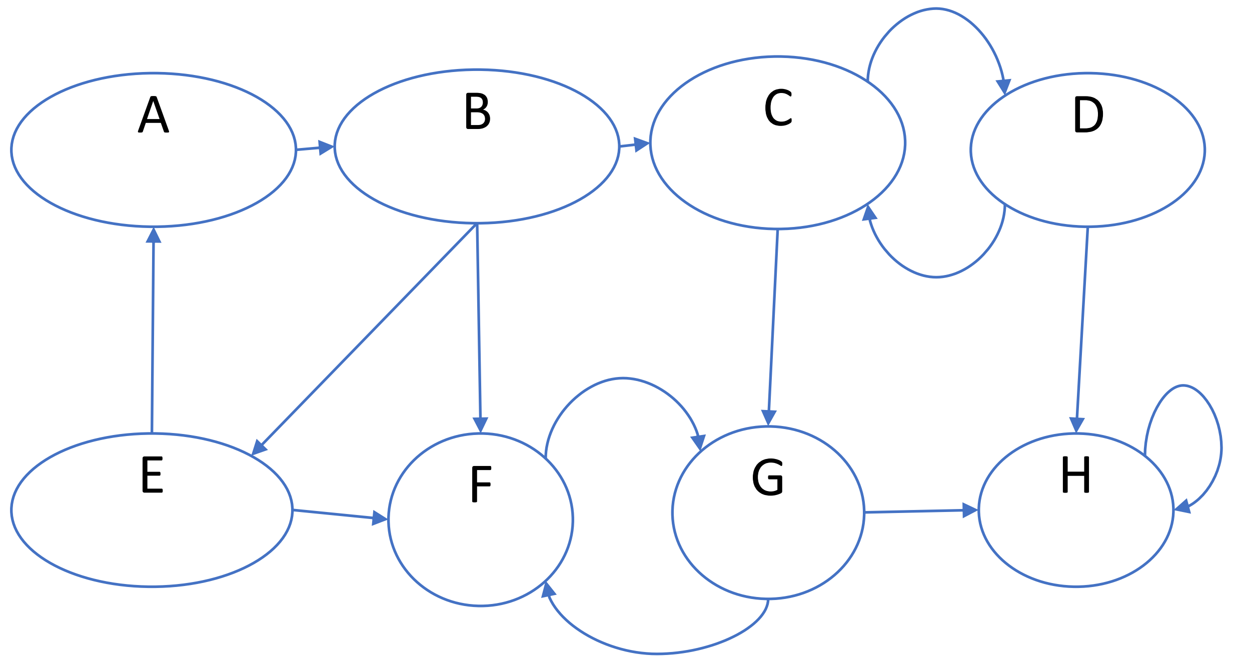
BFS G^T, C2



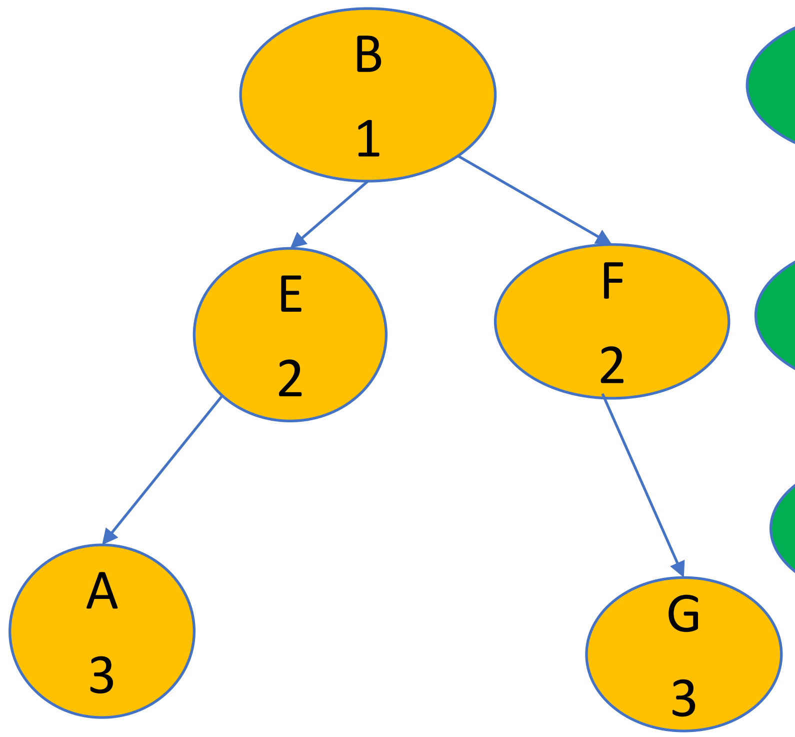
C1∩C2={C,D}, where C1 and C2 are the two BFS trees. {C,D} are in one component.

Graph G (Example)

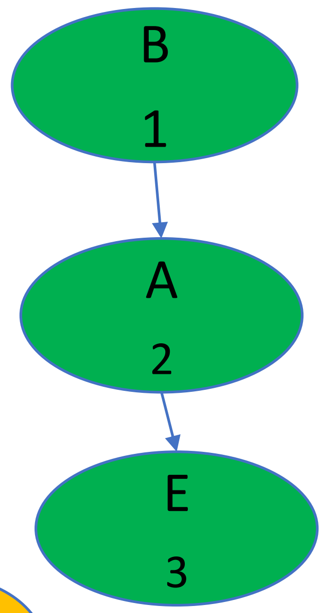
Graph G



BFS G, C1



BFS G^T, C2



C1∩C2={B,A,E}, where C1 and C2 are the two BFS trees. {B,A,E} are in one component.

Algorithm

1. Pick any node from G, run BFS on G to generate a BFS tree
2. Find G^T
3. Pick the same node as in step 1 to generate a BFS tree from G^T
4. Take the intersection of the two trees from step 1 and step 2. The common vertices are in one component
5. Repeat the process for the vertices not currently part of any discovered connected component

Credit: Dr. Shah Jamal

Connected components using BFS (undirected):

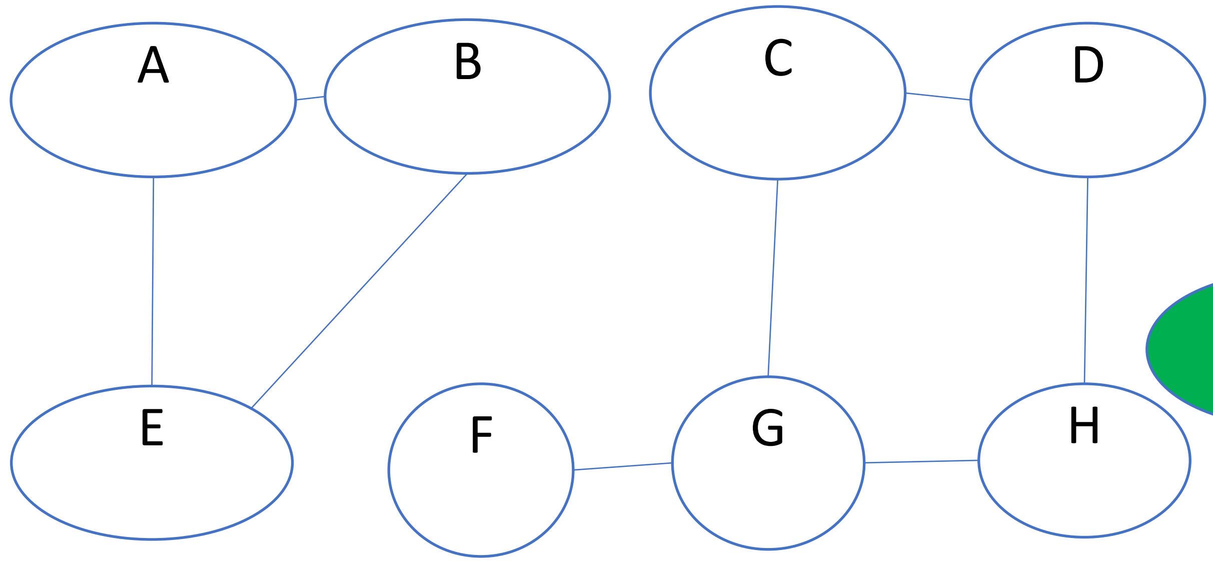
Maximal connected regions in the graph

```
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 vertex $v$ in $G.Adj[u]$    // search the neighbors of $u$
13          if $v.color$ == WHITE            // is $v$ being discovered now?
14              $v.color$ = GRAY
15              $v.d = u.d + 1$
16              $v.\pi = u$
17              ENQUEUE($Q, v$)              // $v$ is now on the frontier
18      $u.color$ = BLACK                    // $u$ is now behind the frontier
```

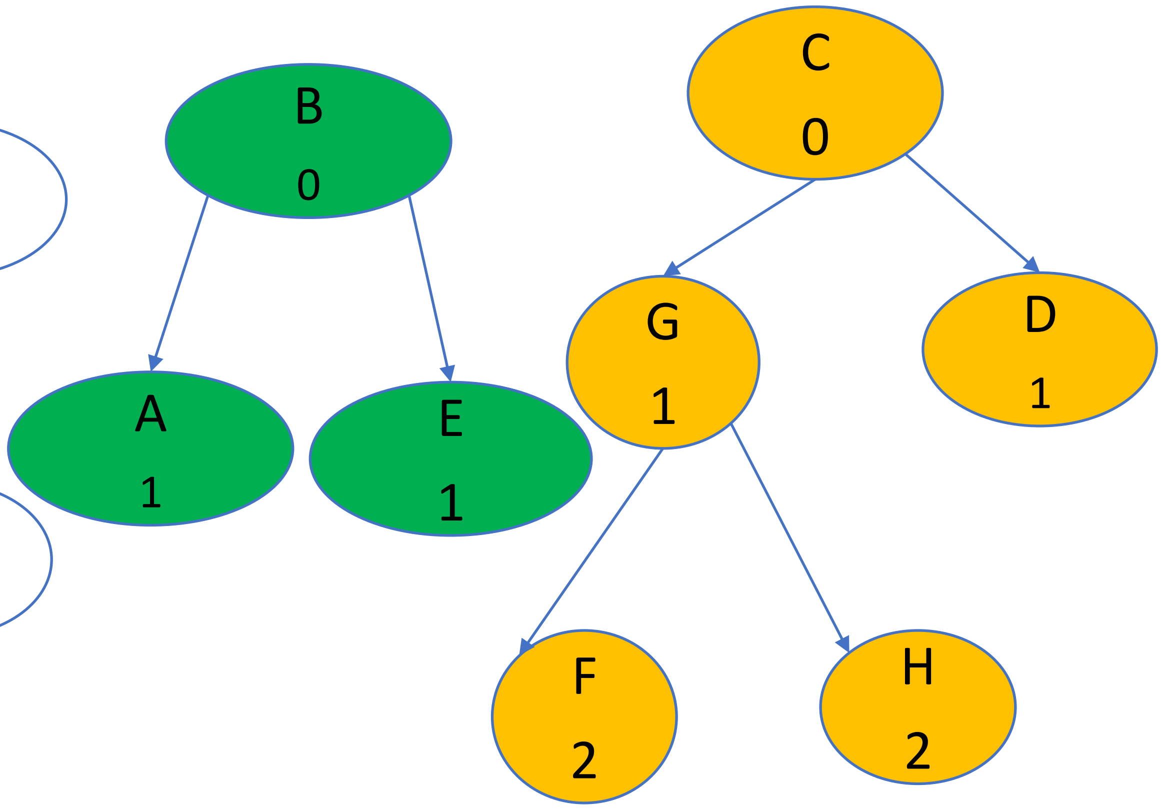
```
BFS_main(G)
1 for each vertex u∈G.v
2     if u unexplored
3         BFS(G,u)
```

Example:

Graph G



BFS forest



Exercise

- Write down an algorithm to find the total number of connected components in any given undirected/directed graph G.

- Steps:
- First, mark all vertices as unvisited.
- Iterate over all vertices.
- If a vertex is not visited, perform DFS on that vertex and increment the **count** by 1 (see Line 6 &7, DFS(G), slide 12).
- After iterating over all vertices, the value of count will be the number of connected components in the graph.

Taken from https://www.codingninjas.com/studio/library/count-number-of-connected-components

Exercise

- Write down an algorithm to find the size of largest component in the Graph G.

Exercise

- **Connected component analysis** (CCA) is a technique in image processing that is used to group pixels in an image that belongs to the same object or entity. The basic idea behind CCA is to identify and label all the connected components in an image, where a connected component is a set of pixels that are connected via some predefined criterion.
- Reference: https://www.scaler.com/topics/connected-component-analysis-in-image-processing/

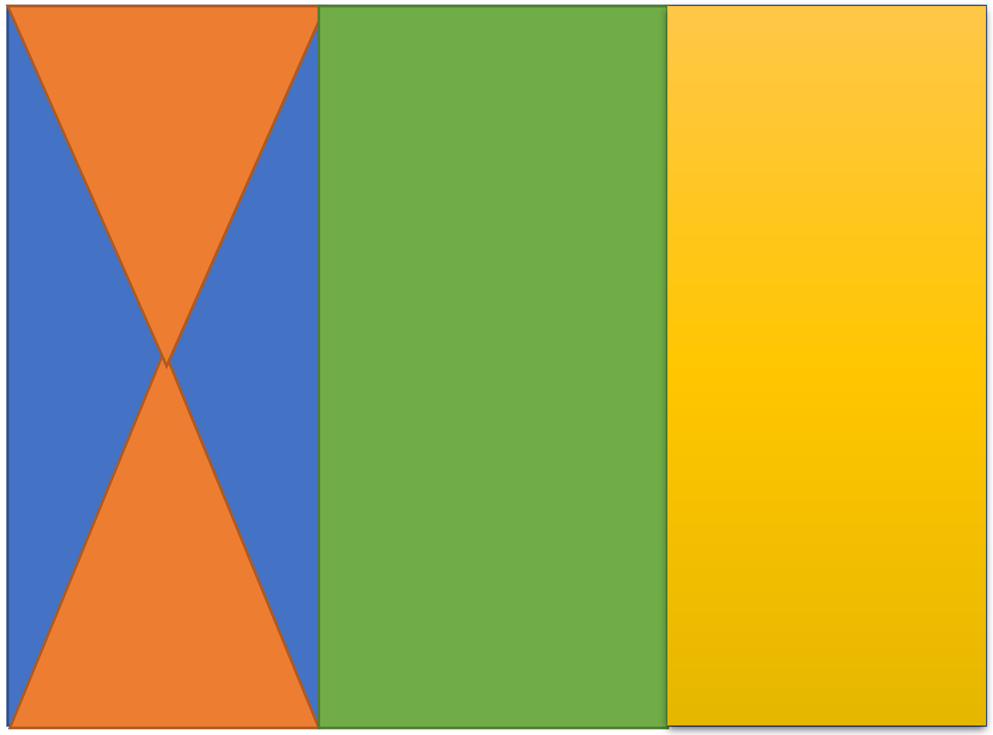
- **Connected component analysis (CCA)** is an important technique in computer vision that is used for a wide range of applications, such as image segmentation, object recognition, and motion tracking. The basic idea behind CCA is to group pixels in an image that are connected and form a single object or entity.

Exercise

- Write down an algorithm to find the connected component in Graph.

Example: How many connected components are there in the following figure?

- 5



In BFS, a queue data structure is used to keep track of the pixels that need to be explored. The algorithm works as follows:

1. Initialize the label of all pixels in the image to 0, indicating that they have not yet been visited.
2. Choose a starting pixel and set its label to a unique identifier (such as 1).
3. Add the starting pixel to the queue.
4. While the queue is not empty, do the following:
    1. Remove the first pixel from the queue.
    2. For each of its neighboring pixels that have not yet been labeled, set their label to the same identifier as the starting pixel, and add them to the queue.
5. Repeat steps 2-4 with another starting pixel until all pixels in the image have been labeled.

At the end of this process, all pixels in each connected component will have the same label

Reference: https://www.scaler.com/topics/connected-component-analysis-in-image-processing/