Fundamental Graph Algorithms

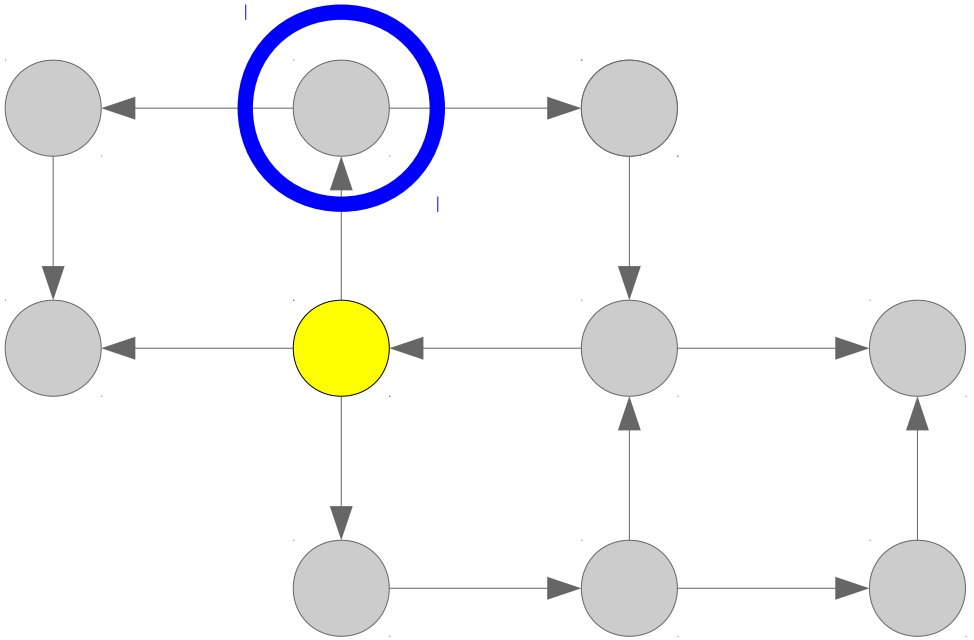
Part Three

Outline for Today

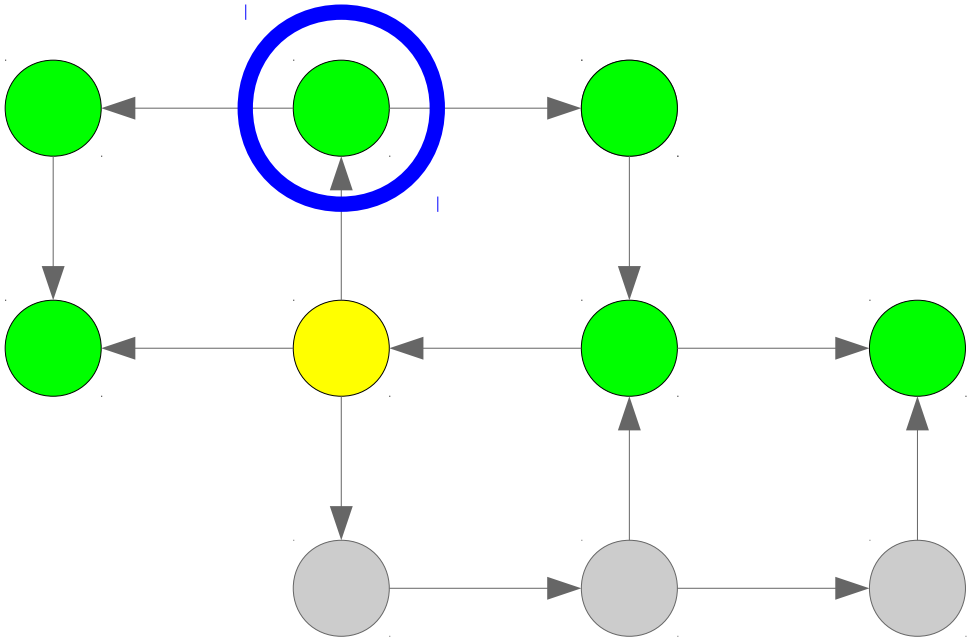
- **Topological Sorting, Part II**
  - How can we quickly compute a topological ordering on a DAG?
- **Connected Components**
  - How do we find the “pieces” of an undirected graph?
- **Strongly Connected Components**
  - What are the “pieces” of a *directed* graph?
- **Kosaraju's Algorithm, Part I**
  - How do we find strongly connected components?

A Correction from Last Time

Theorem: When DFS(*s*) returns, the set of nodes colored green by that call is precisely the set of nodes reachable from *s* along a path consisting purely of gray nodes (we'll call this a *gray path*).



Theorem: When DFS(*s*) returns, the set of nodes colored green by that call is precisely the set of nodes reachable from *s* along a path consisting purely of gray nodes (we'll call this a *gray path*).



***Lemma:* Suppose that when DFS(*s*) is called, *v* is gray and there is a path from *s* to *v* consisting solely of gray nodes. Then *v is* green when DFS(*s*) returns.**

Proof: DFS(s) returns only after all recursive DFS calls have returned. DFS(v) always colors v green, so if v was gray when *DFS*(s) was invoked, the only way v wouldn't be green when DFS(s) ends is if DFS(v) is never called.

Suppose there is a node v where a gray s-v path P exists when DFS(s) is called where DFS(v) is never invoked. Let x be the first node on path P where DFS(x) wasn't invoked. x can't be s, since DFS(s) is explicitly invoked, so x is preceded by some node y in P. Consider these cases:

Case 1: DFS(y) is never invoked. But then x is not the first node on path P where DFS was not called.

Case 2: DFS(y) was invoked. At this time, x would have been gray, so DFS(y) would have called DFS(x).

In both cases, we reach a contradiction, so our assumption must have been wrong. Thus all gray nodes reachable by a gray path must been green when DFS(x) returns. ■

Lemma: **Suppose that v was gray when DFS(s) was called and is green when DFS(s) ends. Then there is a path from s to v consisting purely of gray nodes.**

Proof: Since v is green when DFS(s) returns and was gray when DFS(s) was called, there must have been a path $P = s, x_1, x_2, \ldots, x_n, v$ from s to v formed by the recursive calls to DFS. This means that $x_1, x_2, \ldots, x_n$ must have been gray when DFS(s) was called, since otherwise these calls would not have been made. Consequently, P is a path consisting purely of gray nodes from s to v, as required. ■

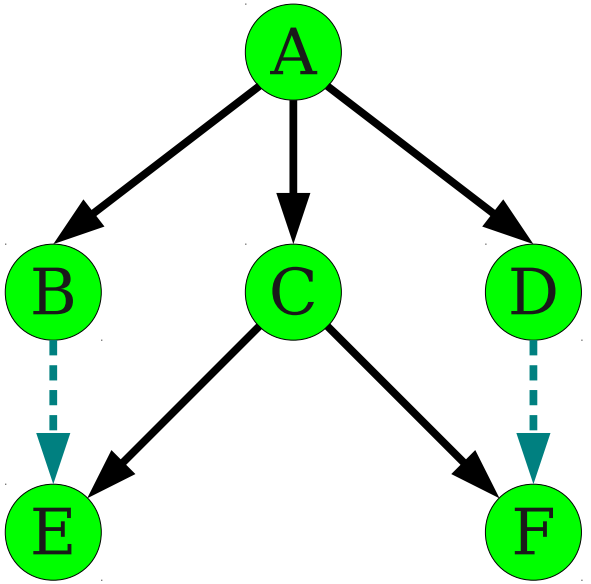
Back to Topological Sorting...

Topological Sorting

- **Goal:** Order the nodes of a DAG G such that if (u, v) is an edge in G, then u appears before v.
- One simple algorithm is as follows: repeatedly find a node with no incoming edges, remove it, and add it to the result.
- As mentioned in Kleinberg and Tardos, can be made to run in $\Theta(m + n)$ time.

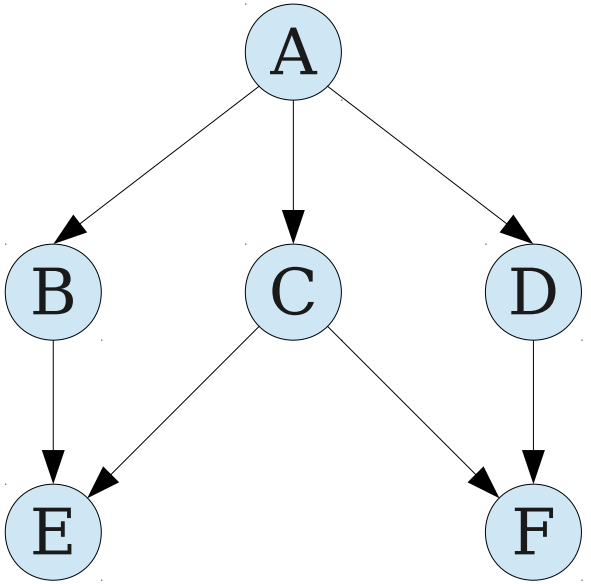
A Completely Different Algorithm

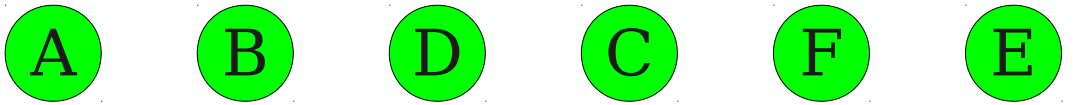
DFS on a DAG



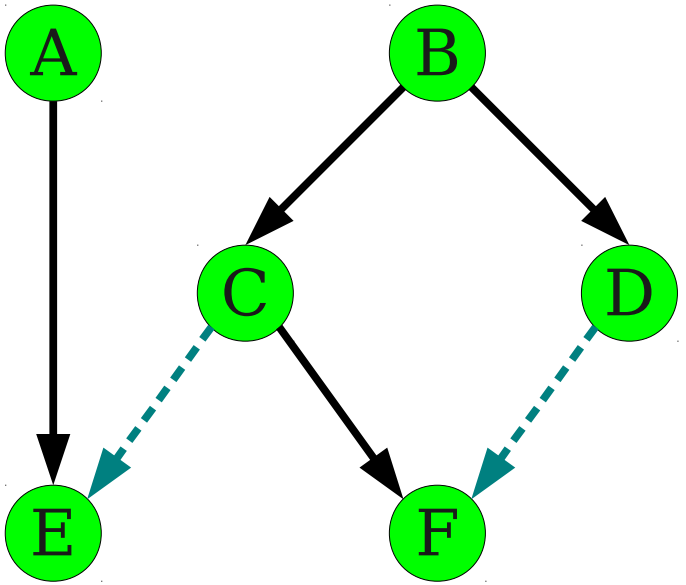
E F C D B A

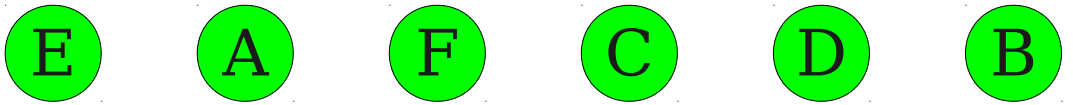
DFS on a DAG



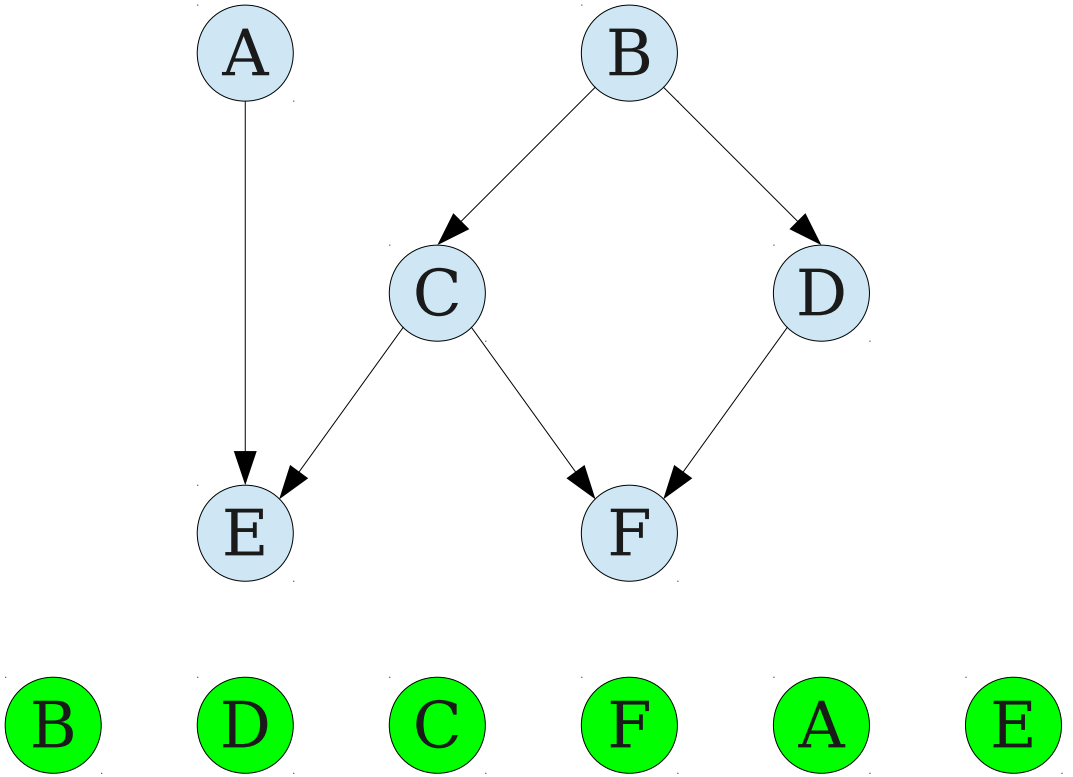


DFS on a DAG, Take II

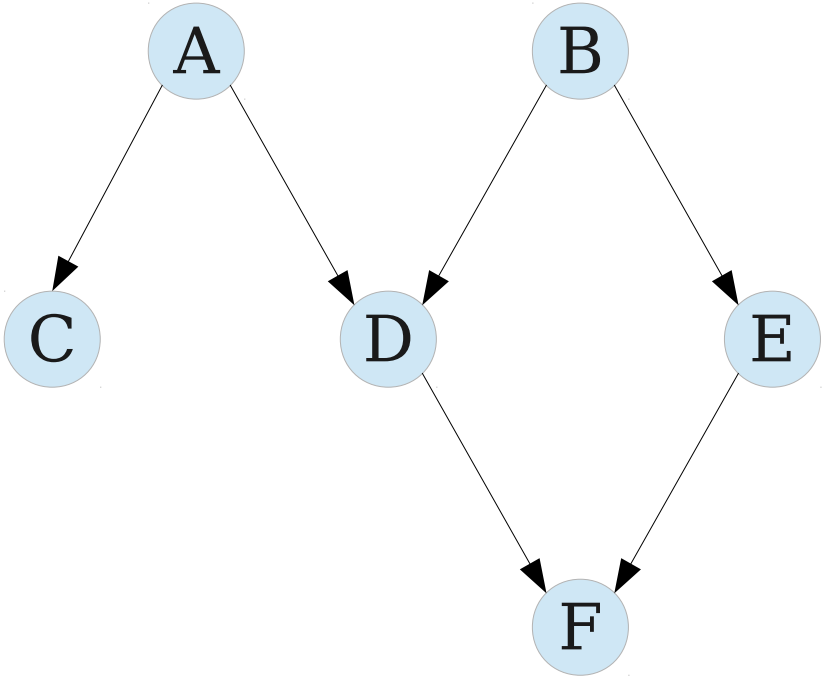




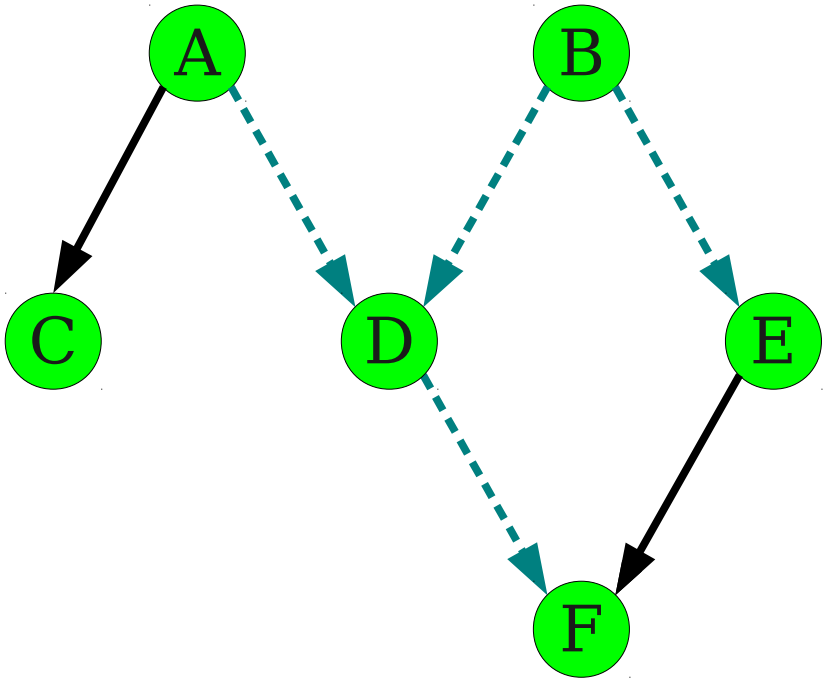
DFS on a DAG, Take II

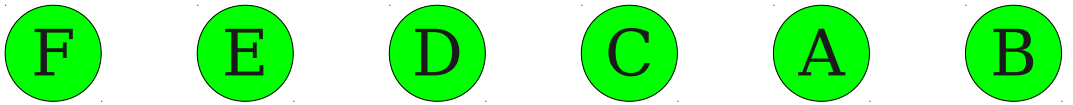


DFS on a DAG, Take III

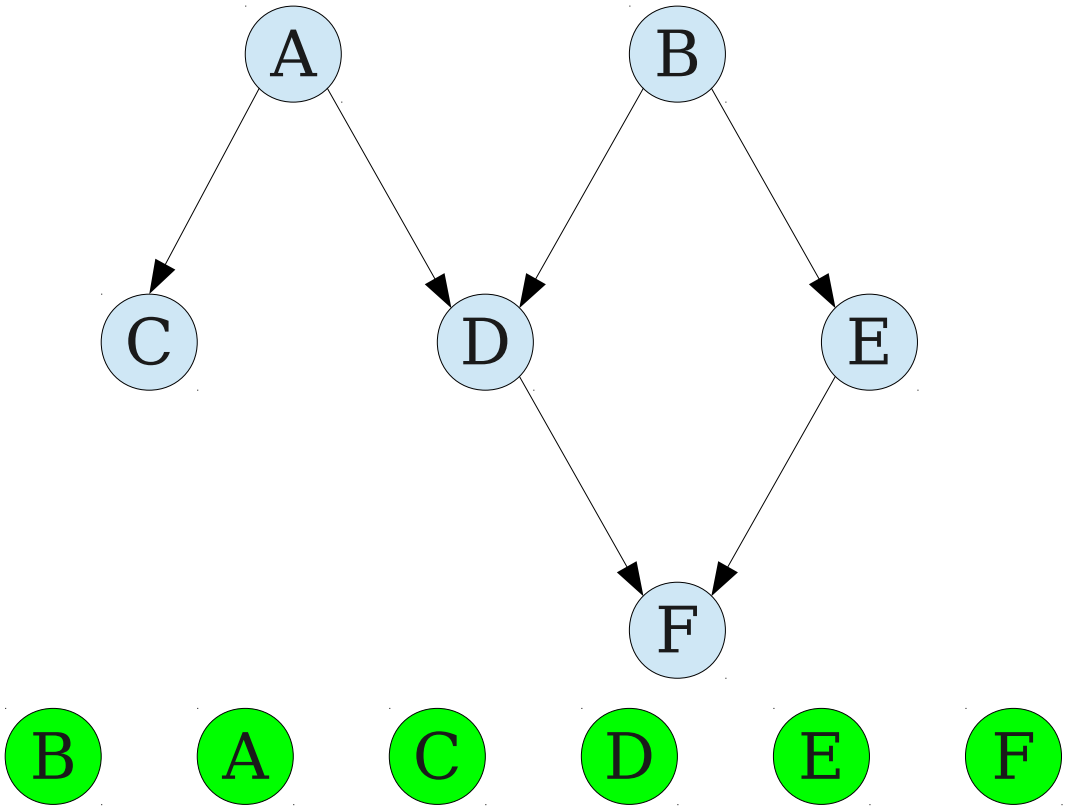


DFS on a DAG, Take III





DFS on a DAG, Take III



```
procedure dfsTopoSort(DAG G):
    for each node v in G:
        color v gray

    let result be an empty list.
    for each node v in G:
        if v is gray:
            run DFS starting from v,
            appending each node to result as
            soon as it is colored green.

    return reverse of result
```

Question 1: How do we know this actually produces a topological sort?

Question 2: How *efficiently* does this produce a topological sort?

Observation I

***Lemma:* Every node appears in the generated list exactly once.**

Proof: Nodes are added to the generated list only when they turn green, which can happen at most once. Moreover, every node has DFS called on it at least once, either by a recursive call or when the top-level loop calls it. ■

Observation II

***Lemma:* If there is an edge (*u*, *v*) in *G*, then *v* will be colored green before *u* is colored green.**

Proof: Note that there cannot be a path from *v* to *u*, since otherwise there would be a cycle in *G* (follow the path from *v* to *u*, then cross (*u*, *v*) to close the cycle).

Since DFS is called on each node in *G*, either DFS(*u*) is called before DFS(*v*) or vice-versa. So suppose DFS(*v*) is called before DFS(*u*). Since there is no path from *v* to *u*, when DFS(*v*) terminates *v* will be green and *u* will not be. Thus *v* becomes green before *u*.

Otherwise, DFS(*u*) is called before DFS(*v*). Since *u* and *v* are gray at this time, there is a path from *u* to *v* of gray nodes. Thus when DFS(*u*) terminates, *v* will be green. Since the last step of DFS(*u*) turns *u* green, this means that *v* became green before *u*. ■

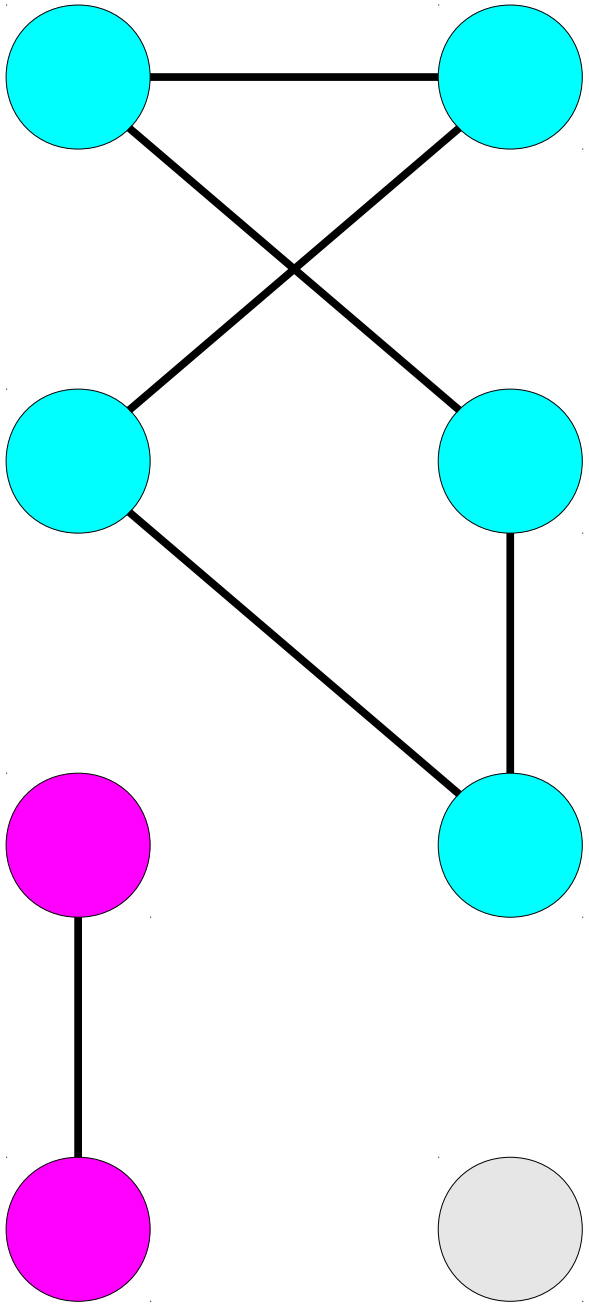
Question 1: How do we know this actually produces a topological sort?

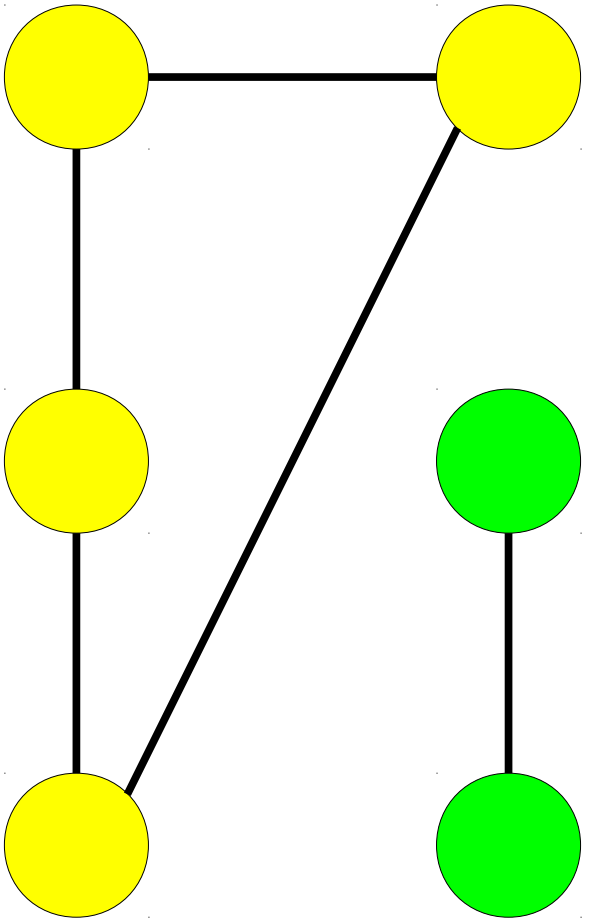
Question 2: How *efficiently* does this produce a topological sort?

DFS Topological Sort

- The time complexity of this algorithm is as follows:
  - Coloring all nodes gray can be done in $\Theta(n)$ time.
  - DFS will be invoked exactly once on each node, either by a top-level call in the loop or by a recursive call. This means each node and edge will be visited at most once by DFS. This step takes $\Theta(m + n)$ time.
  - The top-level loop visiting nodes requires $\Theta(n)$ work.
  - Reversing a list of n elements requires $\Theta(n)$ work.
  - Total work required: $\boldsymbol{\Theta(m + n)}$
- Asymptotically the same as our previous algorithm, but a lot easier to code up!

Connected Components





Connected Components

- Let $G = (V, E)$ be an undirected graph.
- Two nodes $u, v \in V$ are called **connected** iff there is a path from u to v.
- A **connected component** of G is a set $C \subseteq V$ with the following properties:
  - C is nonempty.
  - For any $u, v \in C$: u and v are connected.
  - For any $u \in C$, $v \in V - C$: u and v are not connected.

Properties of Connected Components

- All of the following are true; it's an interesting exercise to prove them:
  - Any two connected components C_1 and C_2 are either equal or disjoint.
  - Every node in a graph belongs to exactly one connected component.
  - The connected components of a graph form a partition of the nodes of the graph.

Finding Connected Components

- Recall: When DFS(u) terminates, u and all gray nodes reachable from u by gray paths will have turned green and no other nodes will have been colored green.
- Suppose that we call DFS in a connected component where we have previously not called DFS before.
- All nodes in the connected component are reachable from one another, and all nodes are gray.
- Therefore, DFS terminates having colored all nodes in that connected component green and coloring no other nodes green.

```
procedure findCCs(graph G):
    for each node v:
        color v gray

    let cc be an array of size n

    let index = 0
    for each node in v:
        if v is gray:
            run DFS(v), setting cc[u] = index
            whenever a node u is colored green
        index = index + 1

    return cc
```

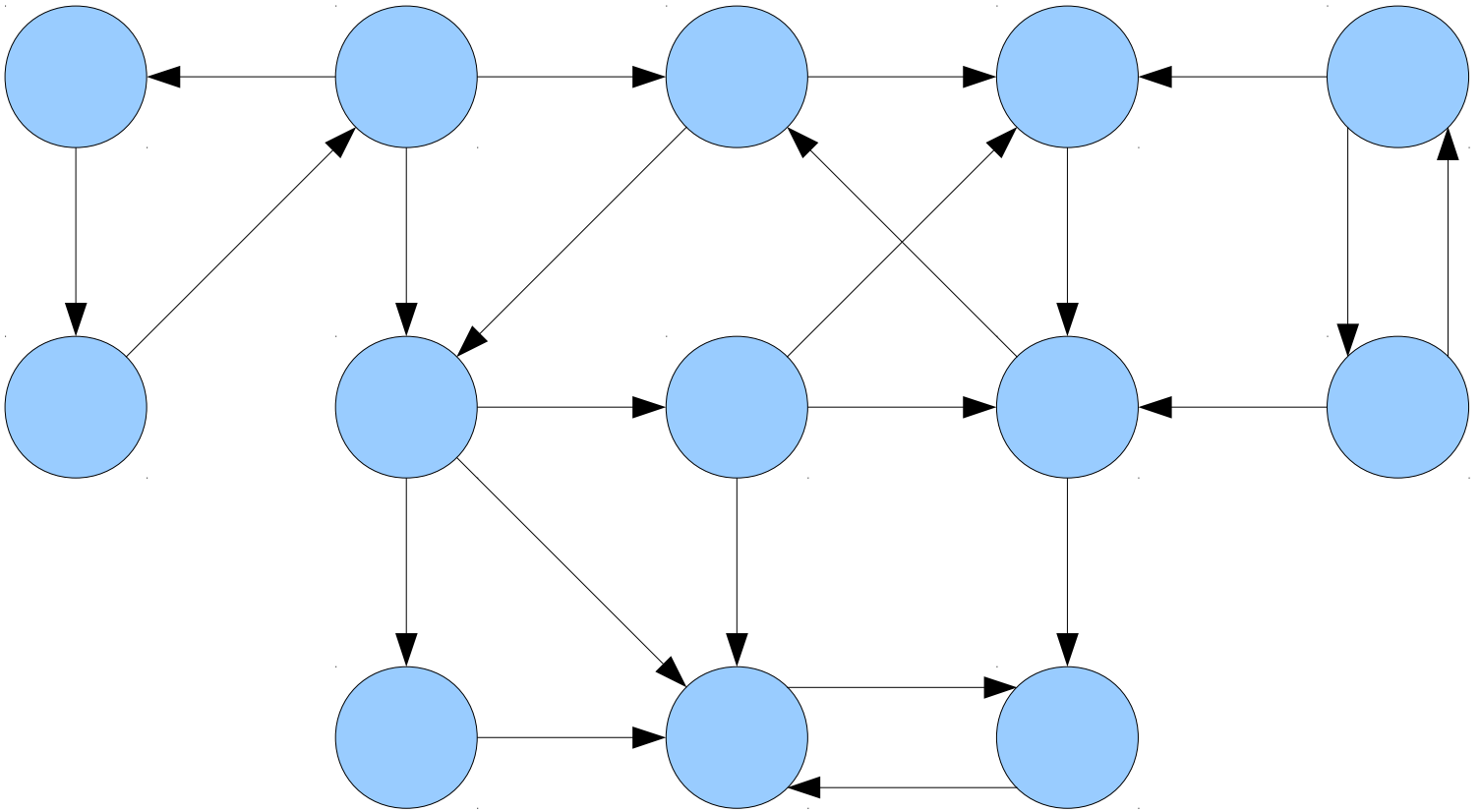
Analyzing the Runtime

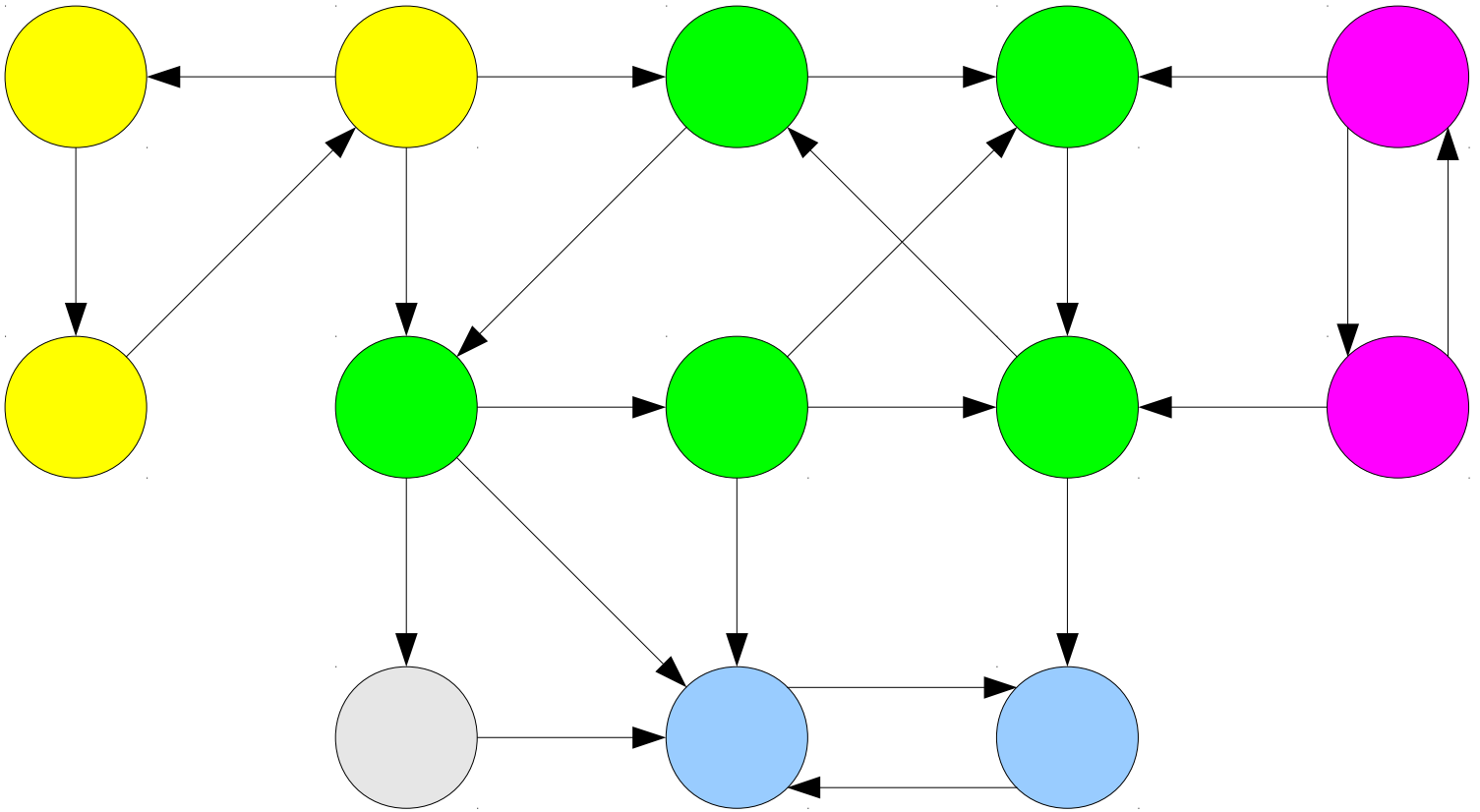
- We do $\Theta(n)$ work initially coloring each node gray.
- Across all iterations of DFS, each node is visited exactly once and each edge is visited exactly once. This takes $\Theta(m + n)$ time.
- Consequently, total work is **$\Theta(m + n)$**.
- Could we also use BFS here? If so, what would the runtime be?

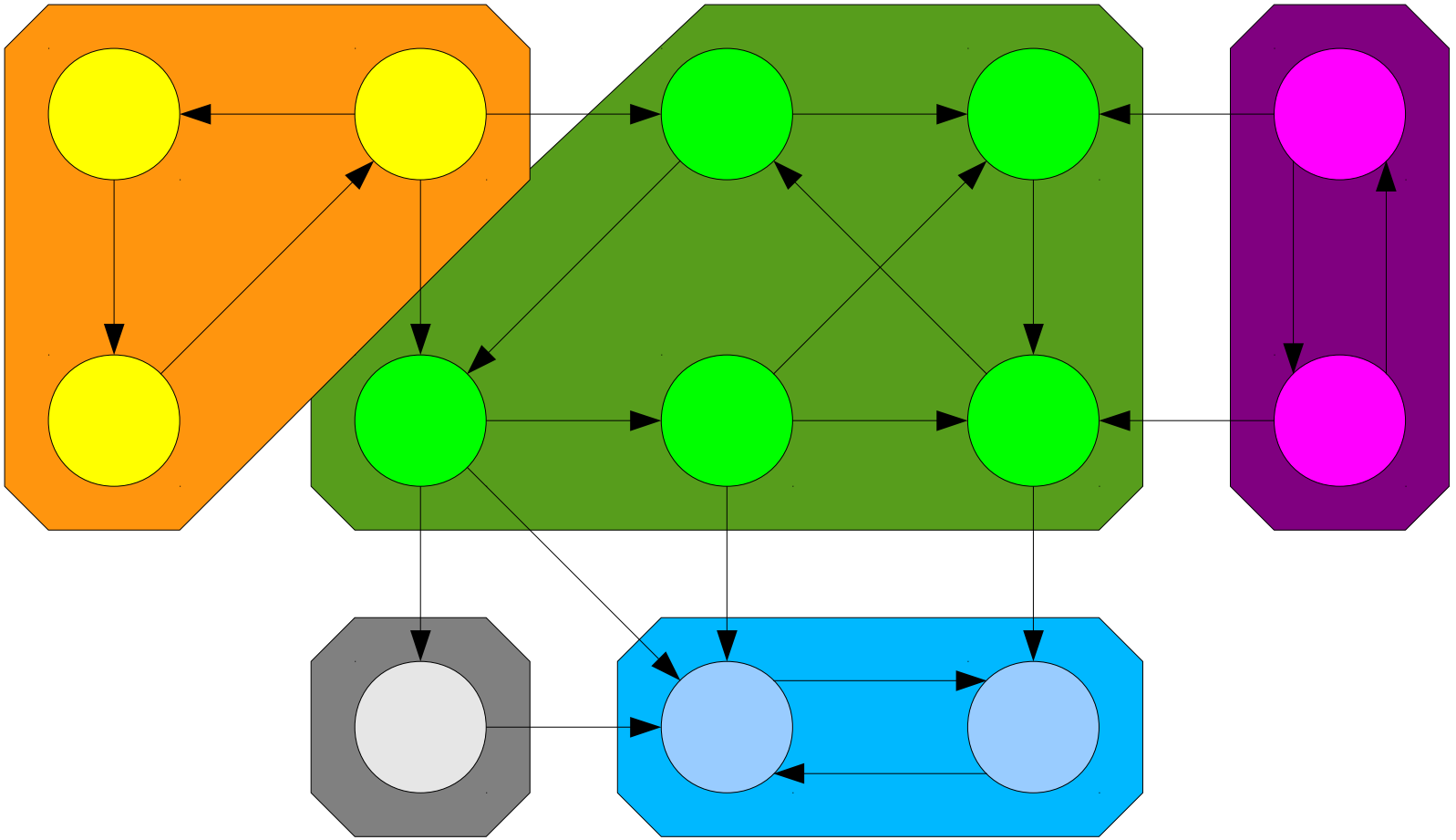
Strongly Connected Components

Directed Connectivity

- In a directed graph G, we say v is **reachable** from u iff there is a path from u to v.
- In an undirected graph, if there is a path from u to v, there is also a path from v to u.
- In a directed graph, it is possible for there v to be reachable from u, but for u not to be reachable from v.
- How would we generalize the idea of a connected component to a directed graph?

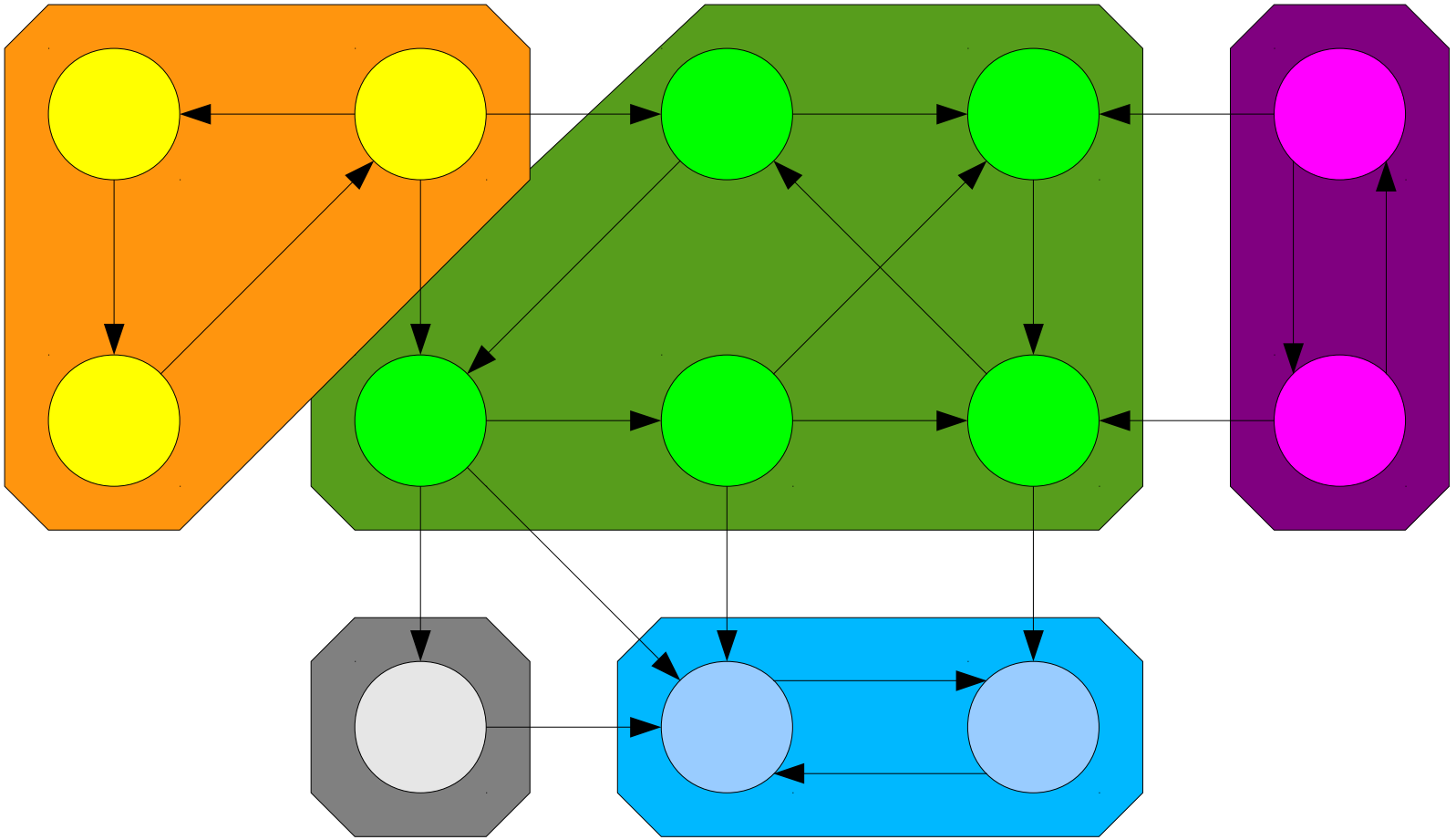


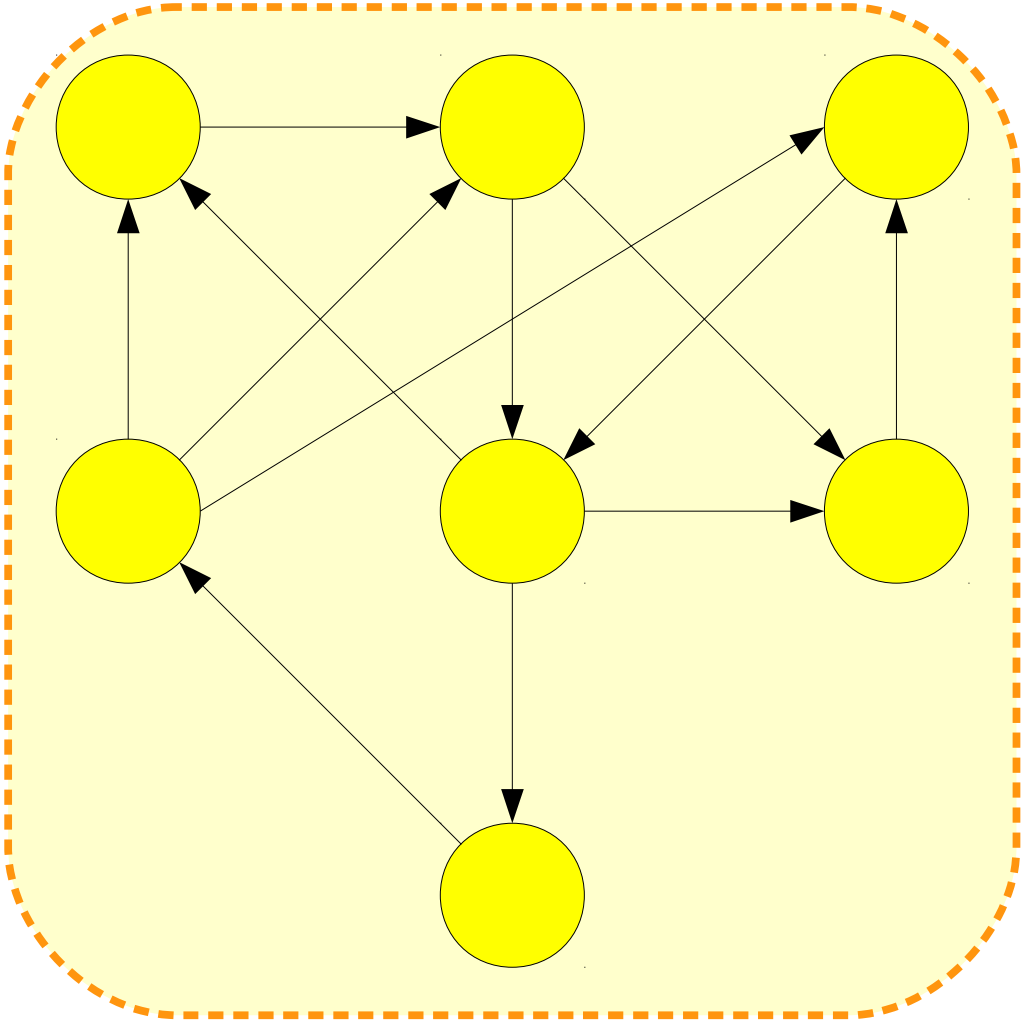


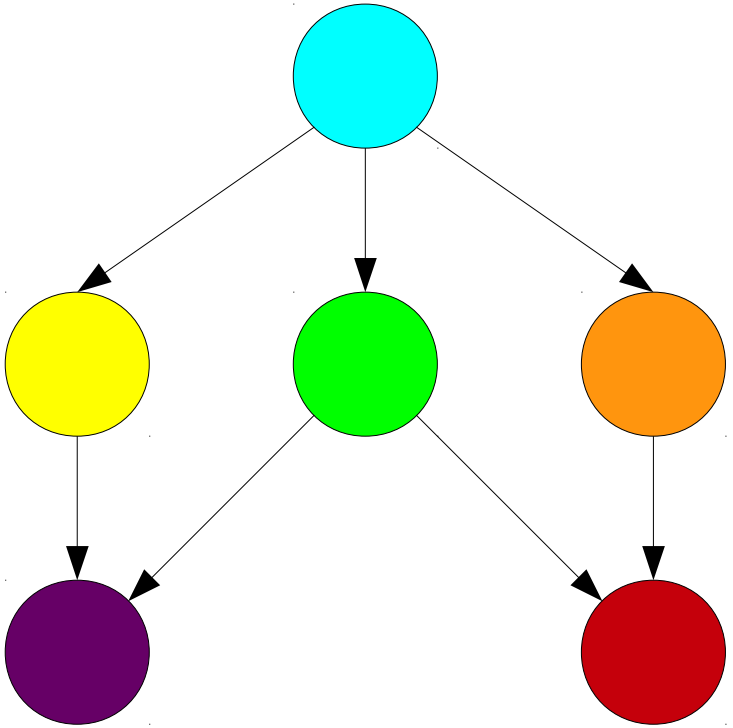


Strongly Connected Components

- Let $G = (V, E)$ be a directed graph.
- Two nodes $u, v \in V$ are called **strongly connected** iff v is reachable from u and u is reachable from v.
- A **strongly connected component** (or **SCC**) of G is a set $C \subseteq V$ such that
  - C is not empty.
  - For any $u, v \in C$: u and v are strongly connected.
  - For any $u \in C$ and $v \in V - C$: u and v are not strongly connected.







Properties of SCCs

- The following properties of SCCs are true; it's a good exercise to prove them.
  - Two SCCs C_1 and C_2 are either equal or disjoint.
  - Every node belongs to exactly one SCC.
  - The SCCs of a graph form a partition of the nodes of the graph.

Finding SCCs

- Every graph must have a collection of SCCs.
- In the *undirected* case, it was easy to find all the connected components of a graph by using DFS or BFS.
- Will this find all SCCs in a directed graph?
- **Question:** How can we determine all of the strongly connected components of a directed graph G?

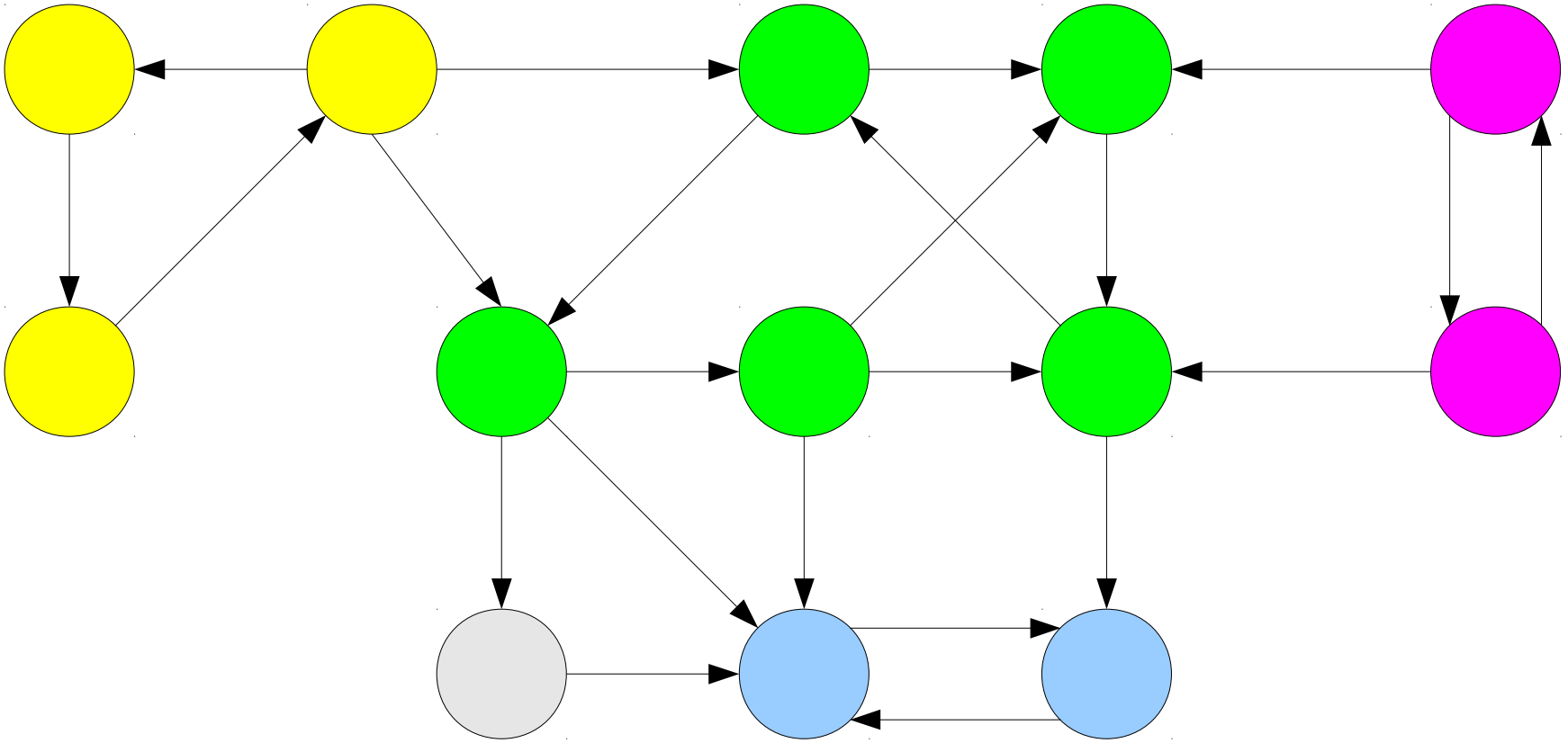
A Beautiful Observation

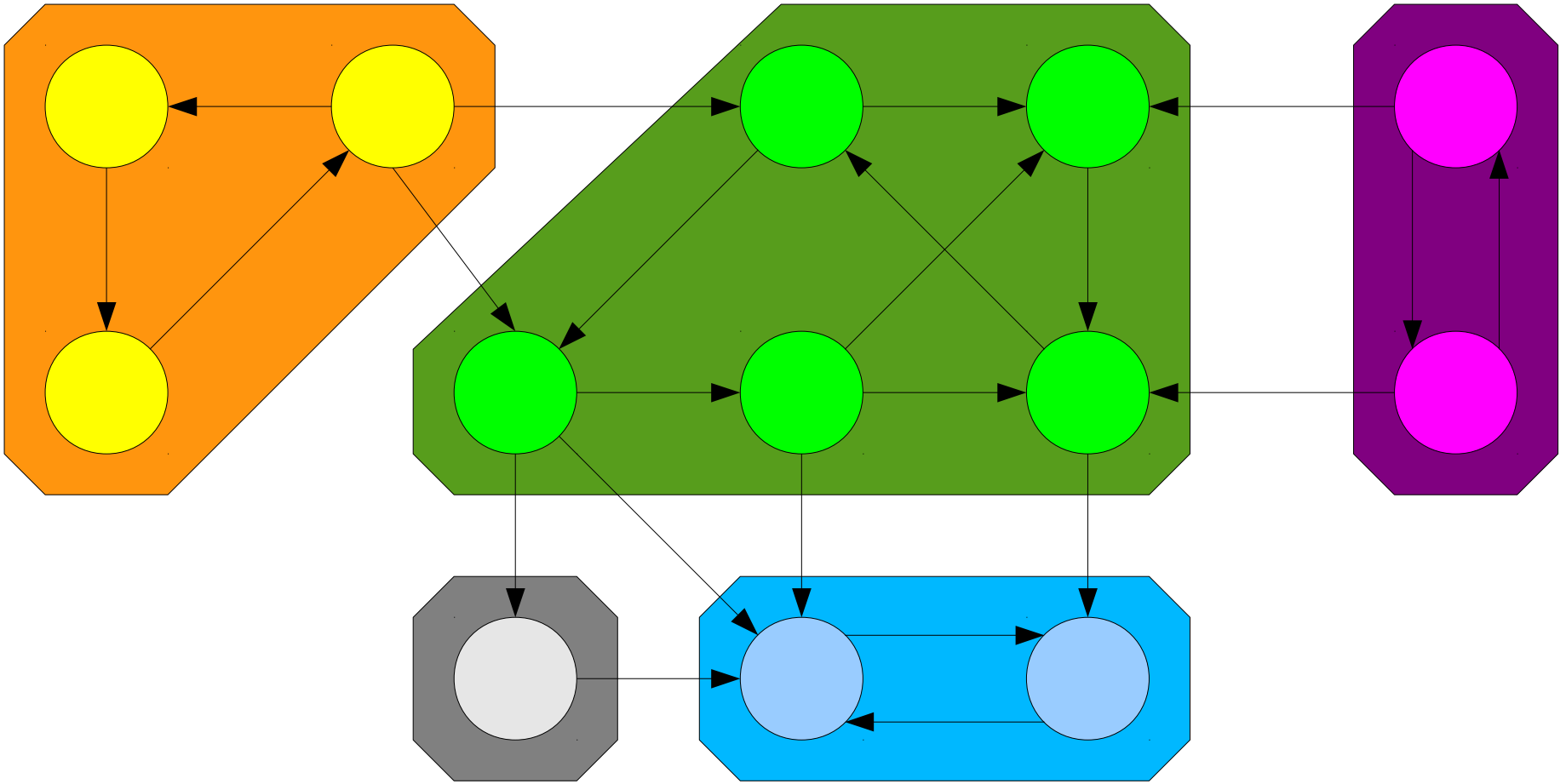
Condensation Graphs

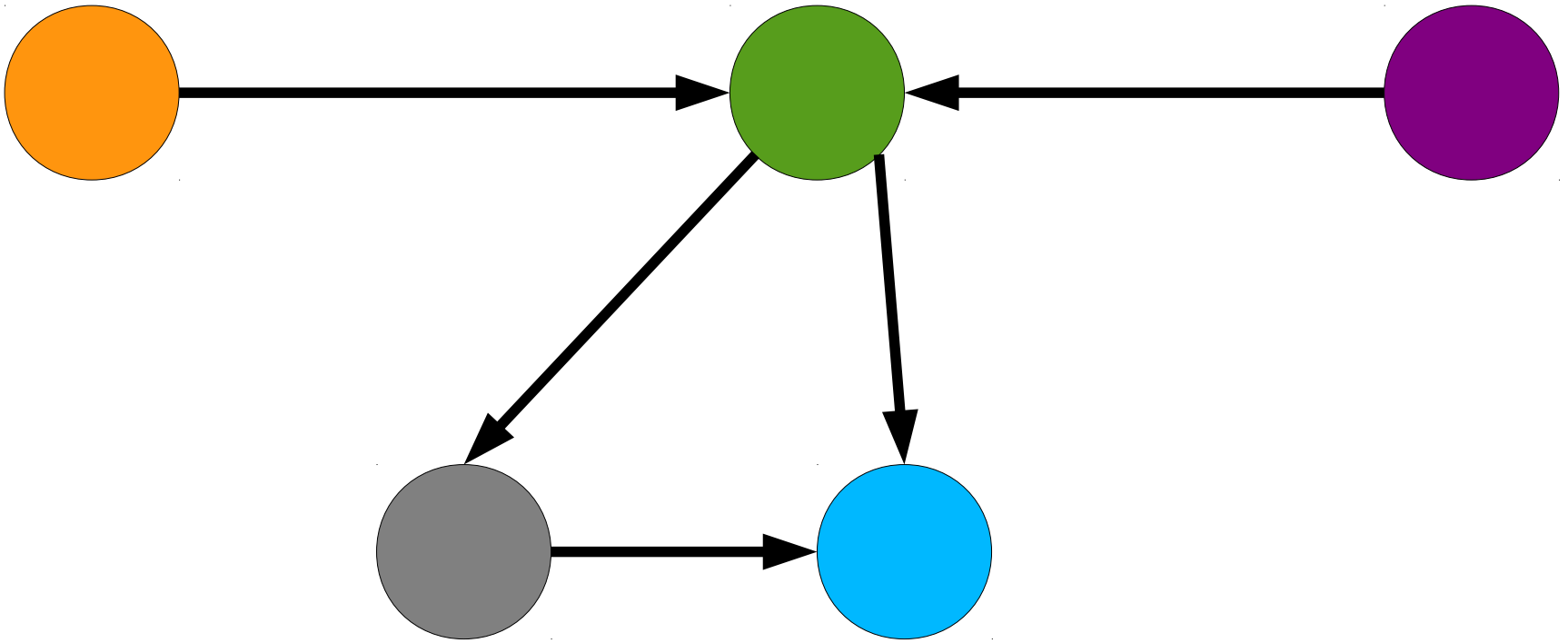
- The **condensation** of a directed graph G is the directed graph G^{SCC} whose nodes are the SCCs of G and whose edges are defined as follows:

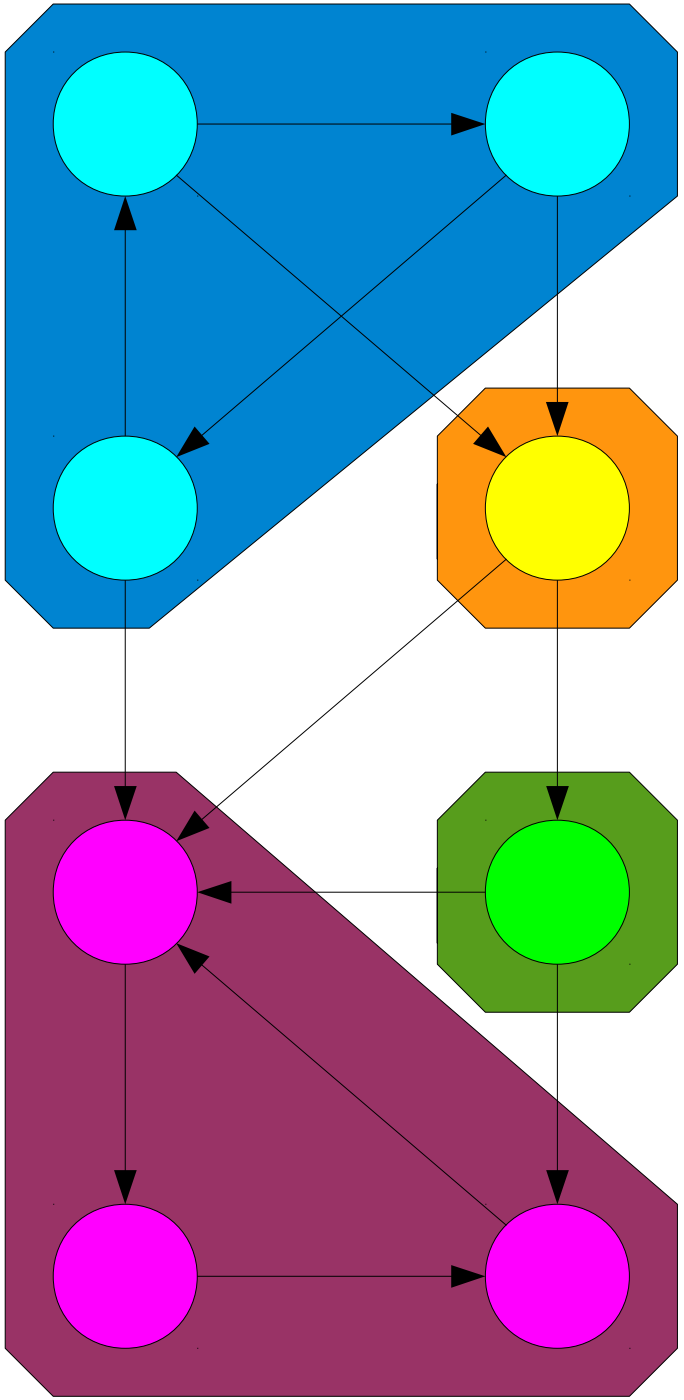
$$(C_1, C_2) \text{ is an edge in } G^{SCC} \text{ iff}$$
$$\exists u \in C_1, v \in C_2.\ (u, v) \text{ is an edge in } G.$$

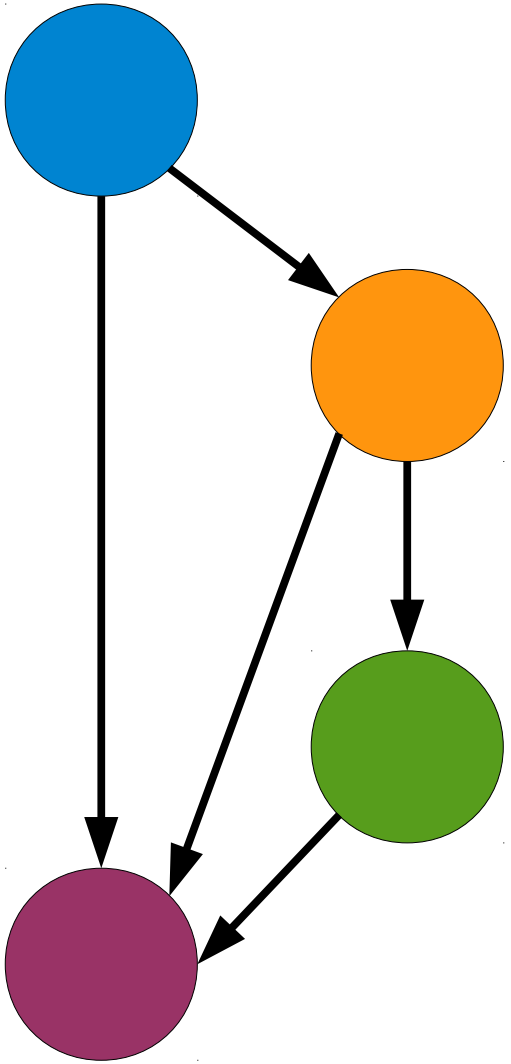
- In other words, if there is an edge in G from *any* node in C_1 to *any* node in C_2, there is an edge in G^{SCC} from C_1 to C_2.

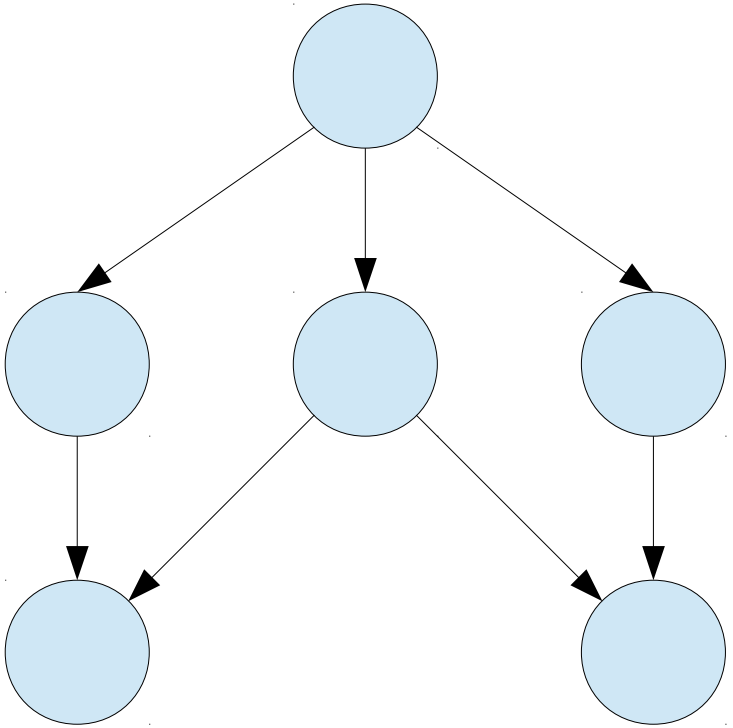


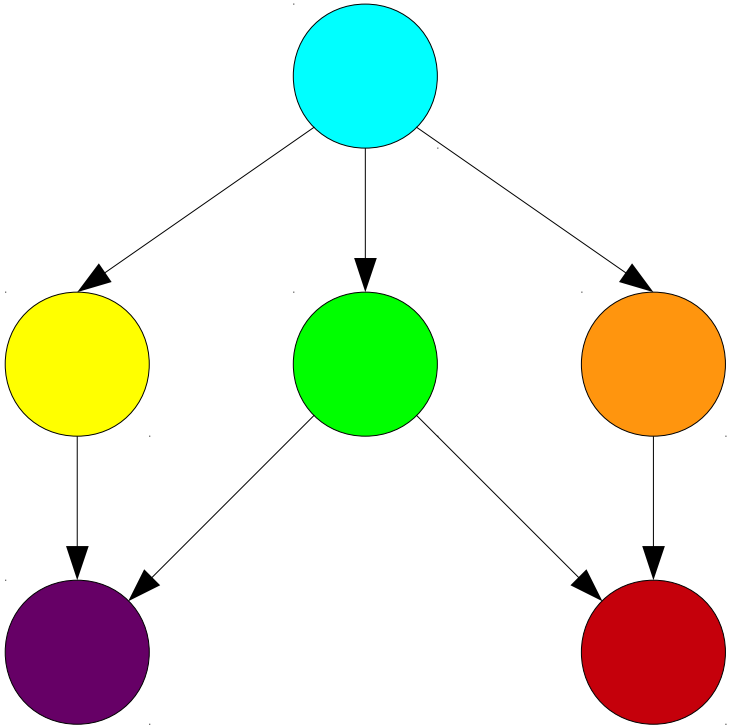






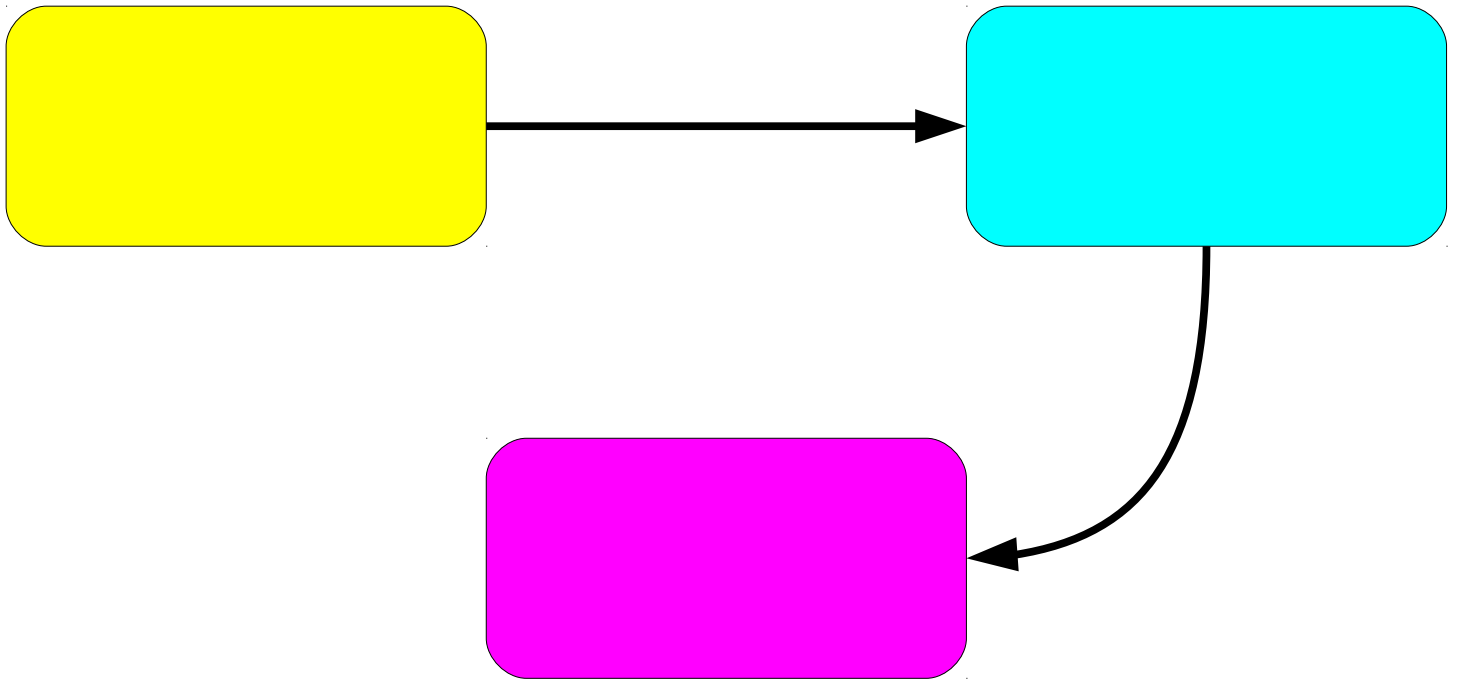






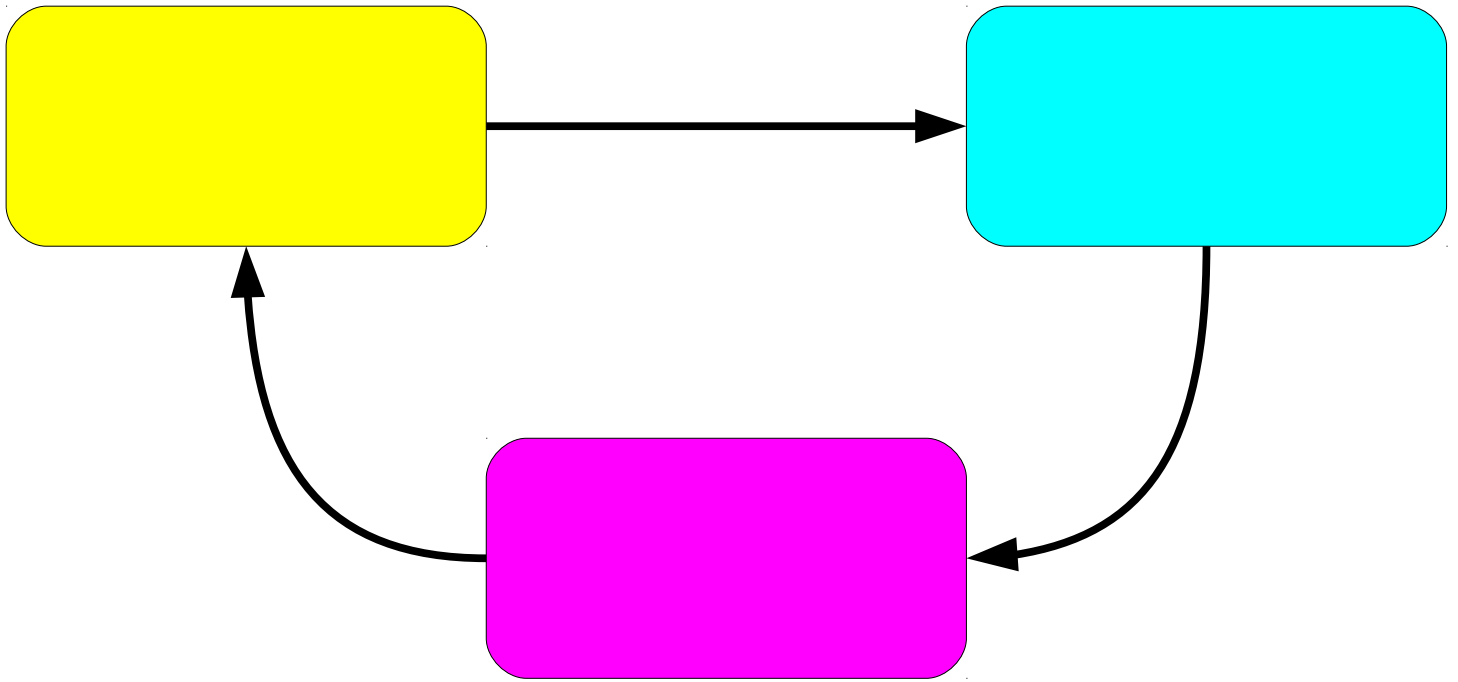
An Amazing Result

- ***Theorem:*** For any directed graph G, the condensation G^{SCC} of G is a DAG.
- *Proof Sketch:*



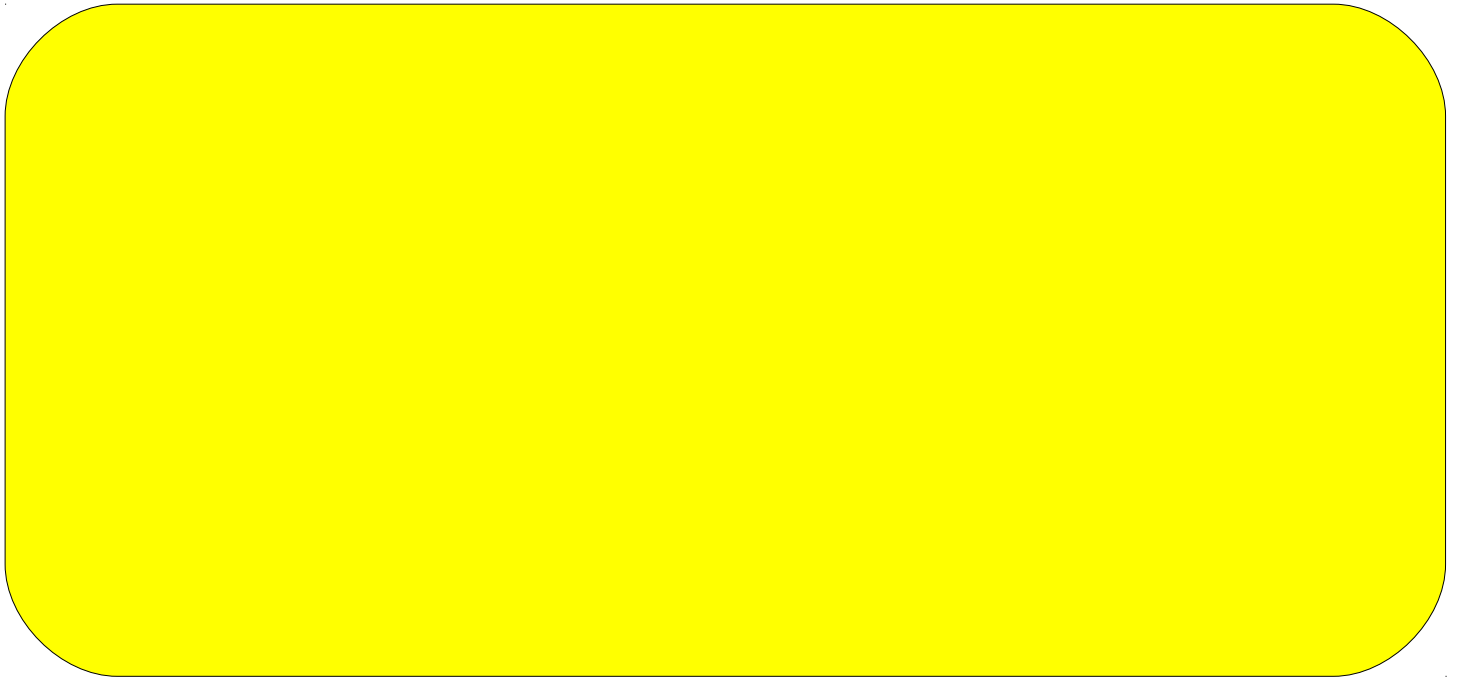
An Amazing Result

- ***Theorem:*** For any directed graph G, the condensation G^{SCC} of G is a DAG.
- *Proof Sketch:*



An Amazing Result

- ***Theorem:*** For any directed graph G, the condensation G^{SCC} of G is a DAG.
- *Proof Sketch:*

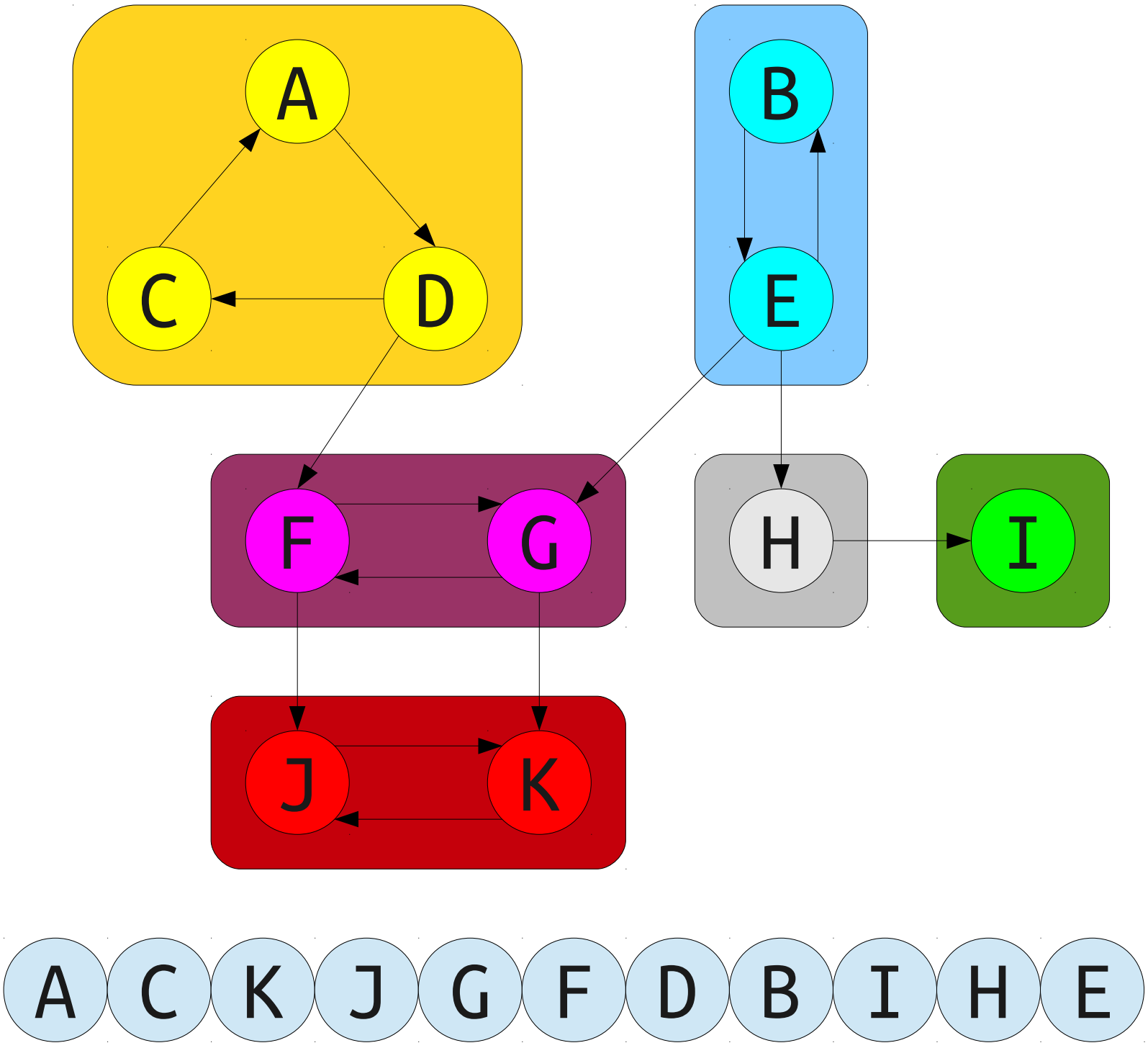


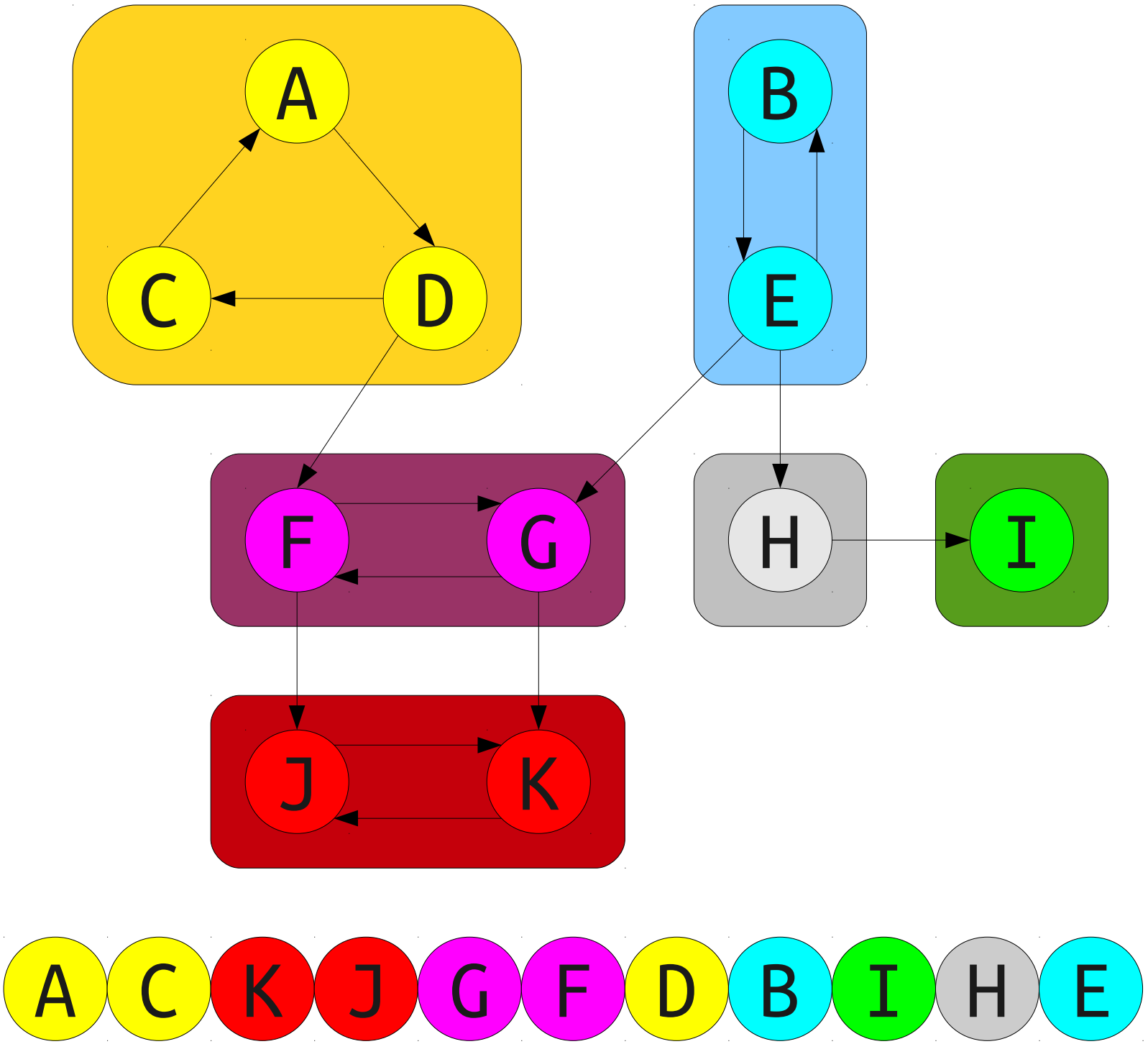
SCCs and DAGs

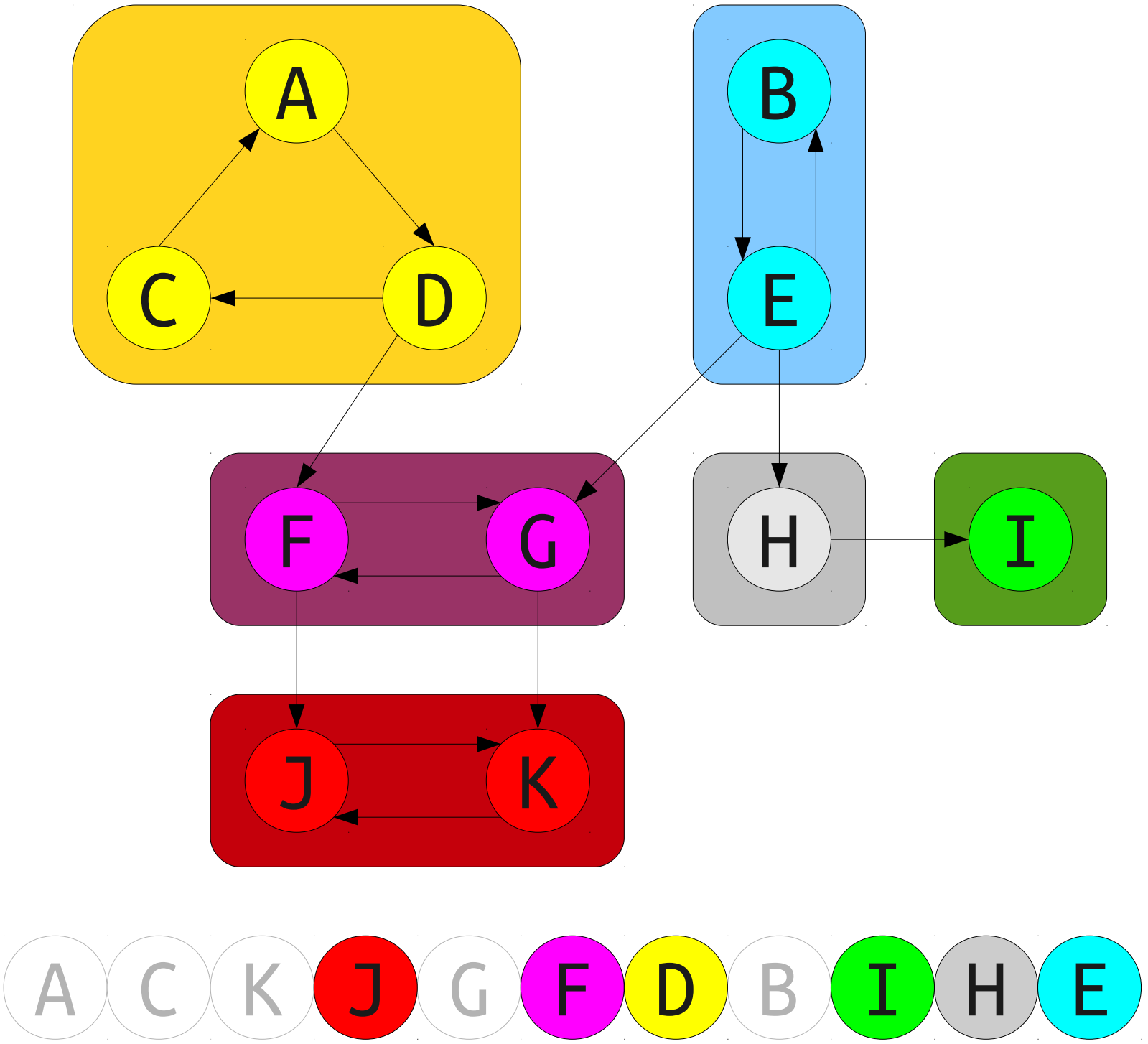
- We now see that there is a close connection between SCCs and DAGs: **the SCCs of a graph form a DAG**.
- Intuitively, you can think of a graph as a two-layer structure:
  - At a high level, a graph is a DAG of SCCs showing the top-level connections between clusters of nodes.
  - At a lower level, you can see the connections between nodes in the same SCC.

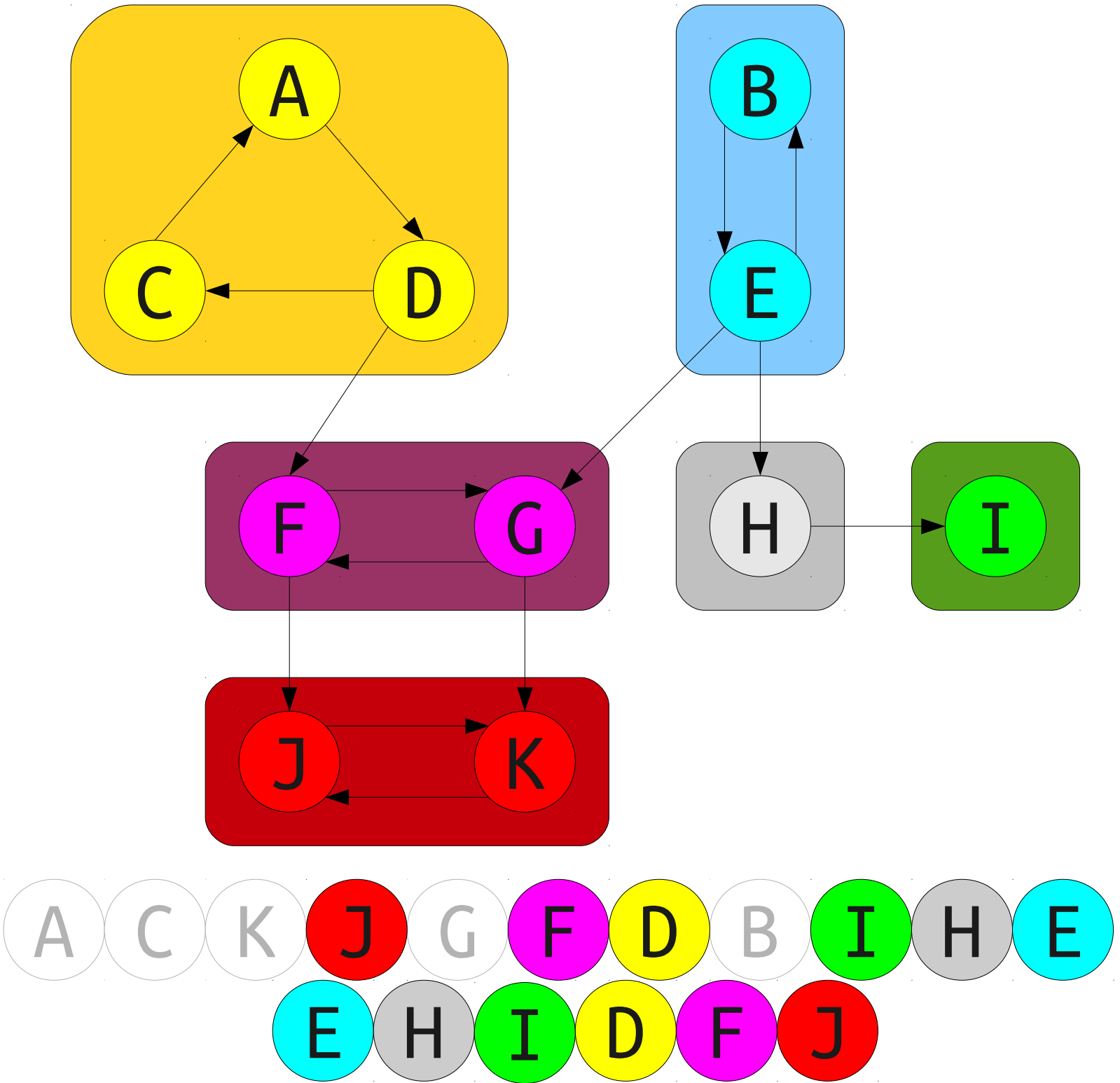
SCCs and DAGs

- Now that we have found a connection between SCCs and DAGs, can we adapt any of our algorithms on DAGs to find SCCs?
- Right now, our main operation on DAGs is topological sort, and we have two algorithms we can use:
  - *Repeatedly removing a source node.* That won't help us here, since we can't easily tell if a node is in a source SCC.
  - *Running DFS and reversing the result.* So what happens if we try that out?


A
B
C
D
E
F
G
H
I
J
K
A C K J G F D B I H E


A
B
C
D
E
F
G
H
I
J
K
A C K J G F D B I H E


A
B
C
D
E
F
G
H
I
J
K
A C K J G F D B I H E


A
B
C
D
E
F
G
H
I
J
K
A C K J G F D B I H E
E H I D F J

What's Going On?

- It looks like if we look purely at the *last* node from each SCC to turn green, we get a topological sort of G^{SCC} in reverse.
  - Here, each SCC is represented by a single node.
- This helps us get a better sense for how the SCCs are interlinked!
- However, we still don't have a reliable way to determine which node is the last node in each SCC to turn green...
- For starters, let's convince ourselves that this isn't a coincidence.

Some Notation

- We'll denote by $\boldsymbol{f(v)}$ the time at which node v is colored green by the algorithm.
  - $f(u) < f(v)$ means "node u was colored green before node v was colored green."
- Note that every node is eventually colored green, so this notation is well-defined.
- Let C be an SCC. Define

$$\boldsymbol{f(C) = \max_{v \in C} f(v)}$$

- In other words, $f(C)$ is the time at which the last node in C was colored green.

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

Proof: At the time DFS(s) is called, since s is the first node in C visited by DFS, all nodes in C are gray. Since C is an SCC, every node $v \in C$ is reachable from s. This means there is a gray path from s to v for every $v \in C$. Thus every node $v \in C$ will be green when DFS(s) returns.

Since the last step of DFS(s) is to color s green, this means that s is colored green only after all other nodes in C are colored green. Therefore, $f(s) \geq f(v)$ for any $v \in C$. Since by definition $f(C) = \max_{v \in C} f(v)$, this means $f(C) = f(s)$. ■

Theorem: **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 nodes 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, since we can go from x_1 to u, across (u, v), and from v to x_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 to different SCCs.

Now, suppose DFS(x_2) is called before DFS(x_1). Since x_1 is not reachable from x_2, x_1 will not be green when DFS(x_2) returns. Thus x_1 becomes green after x_2, so $f(x_2) < f(x_1)$.

Otherwise, DFS(x_1) was called before DFS(x_2). When DFS(x_1) is called, all nodes in C_1 and C_2 are gray, so there is a gray path from x_1 to x_2. Thus when DFS(x_1) returns, x_2 will be green. Since DFS(x_1) colors x_1 green just before it returns, this means that x_1 was colored green after x_2, so $f(x_2) < f(x_1)$. ■