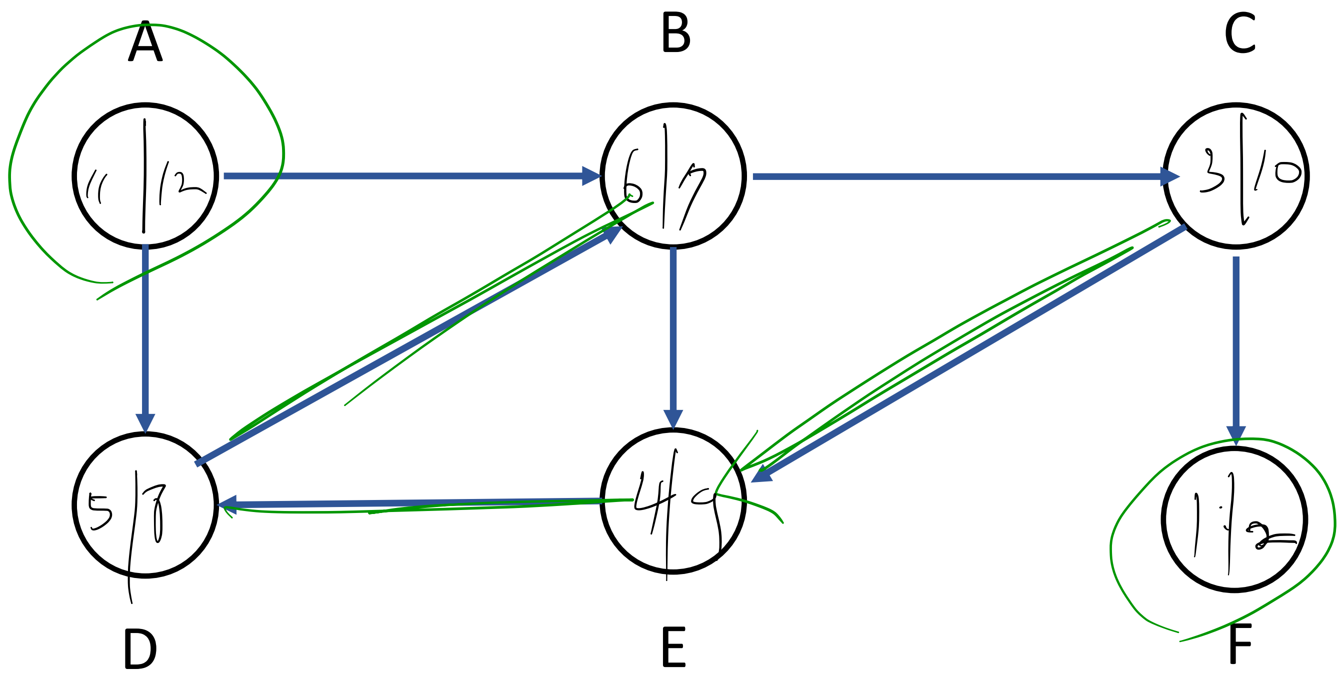
Supplemental Slides of Ch22

Sungjin Im

4/5/2023

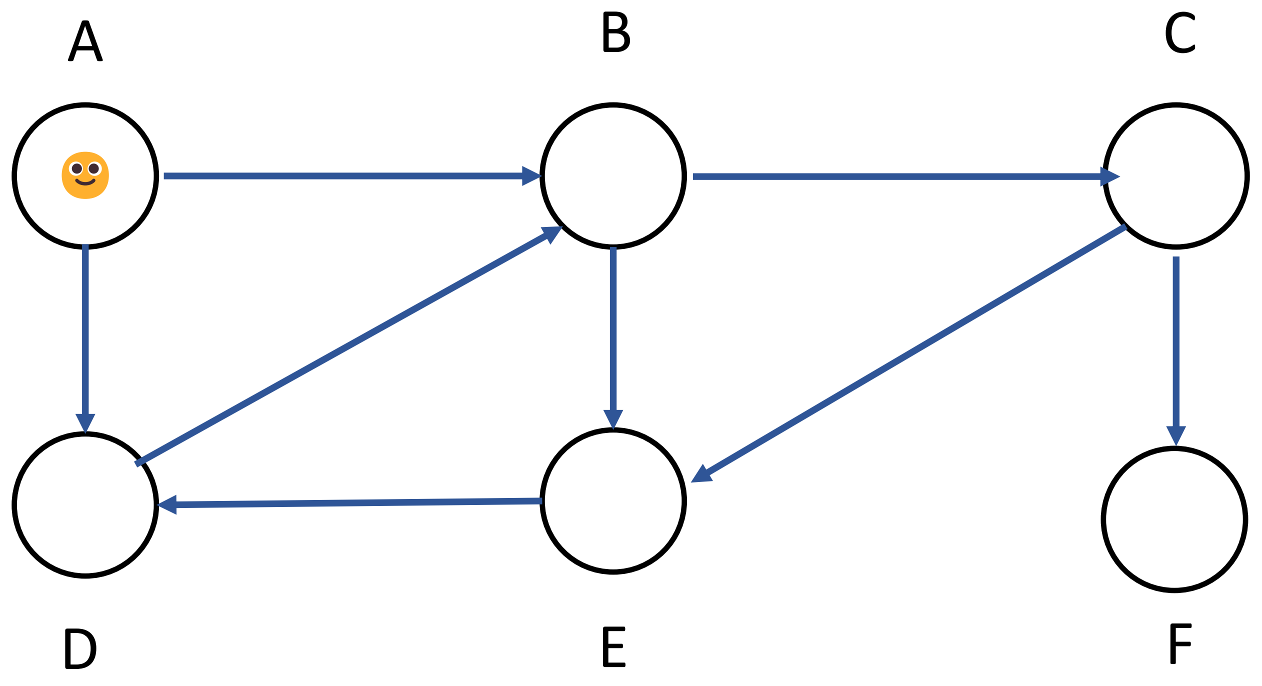
DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



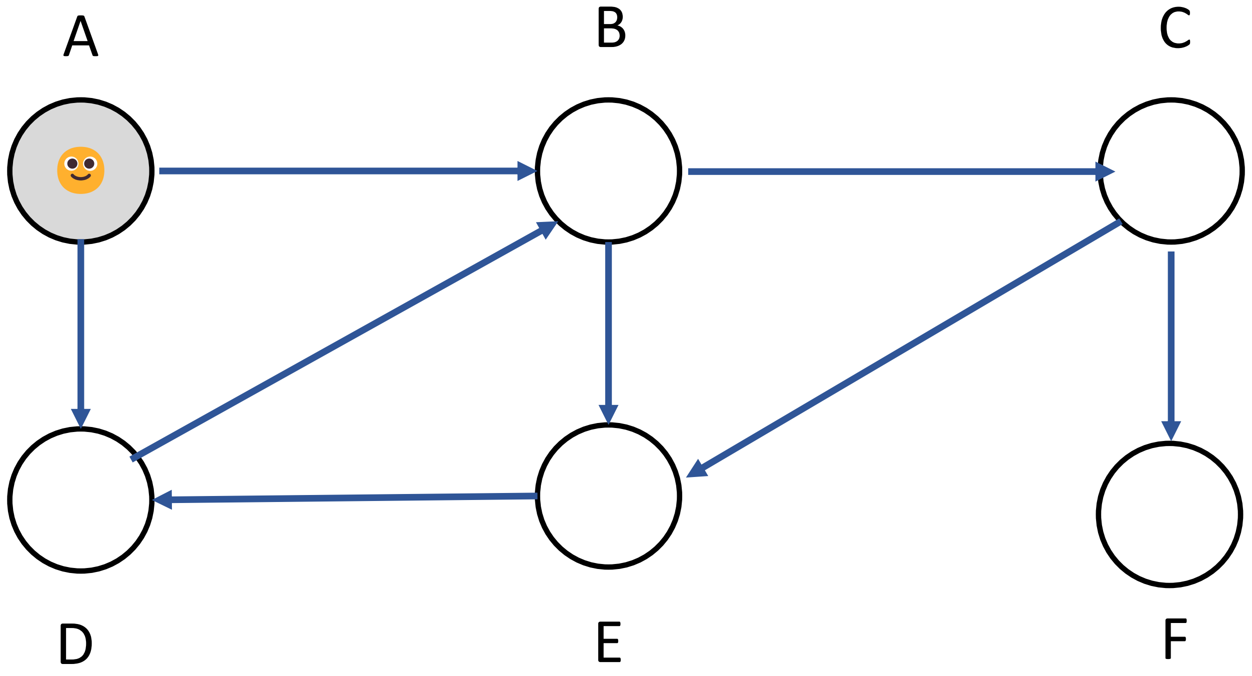
DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



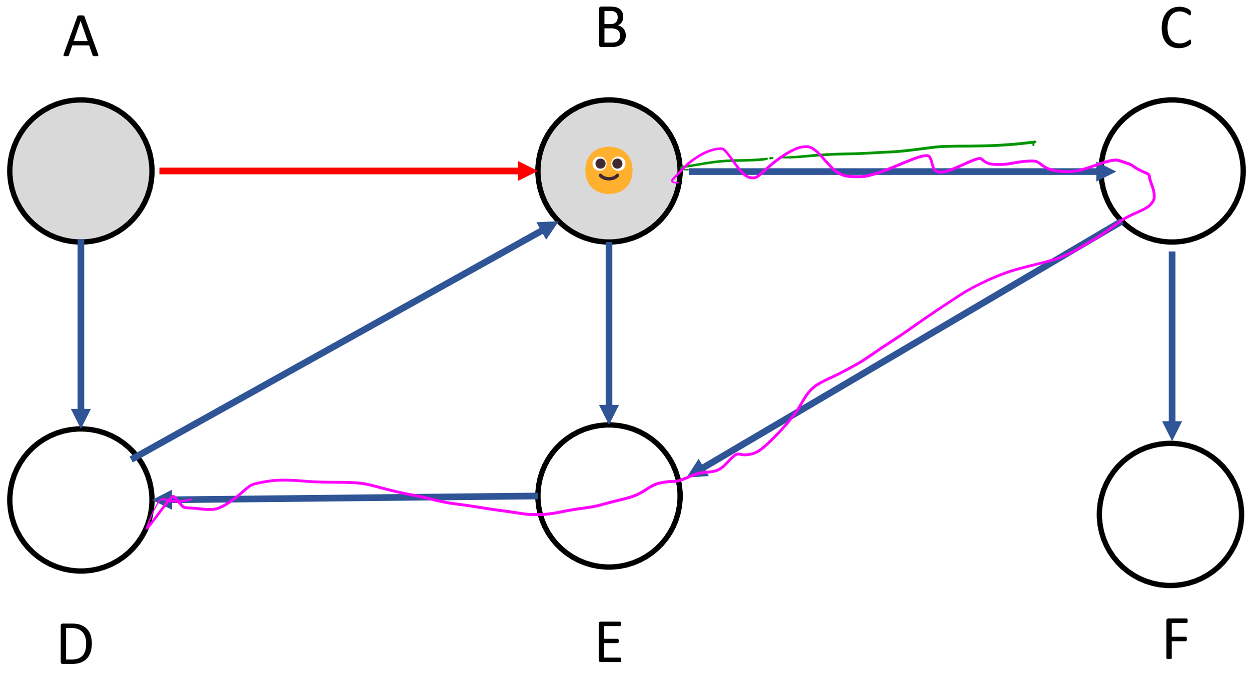
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 1

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



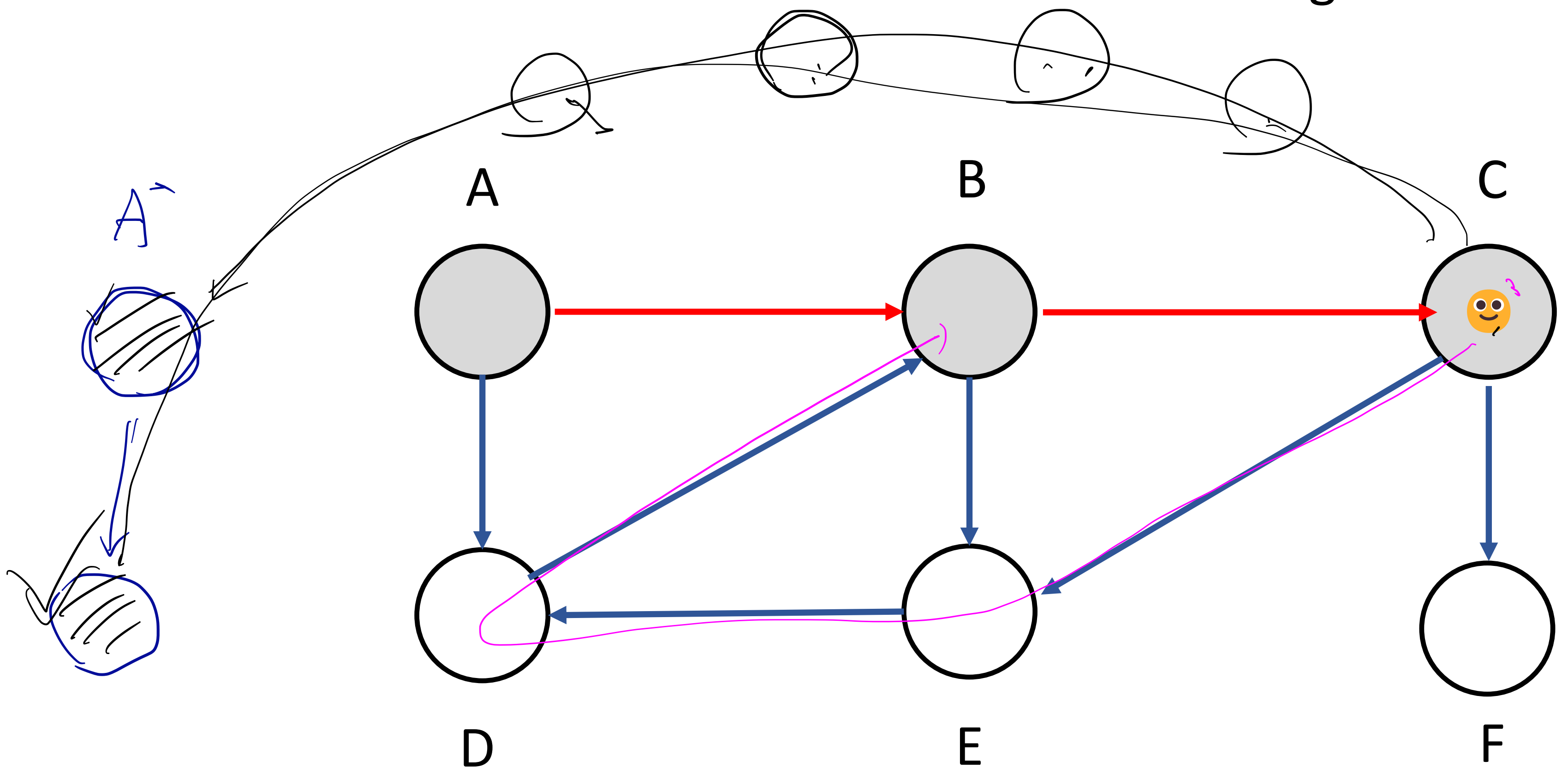
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 2

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



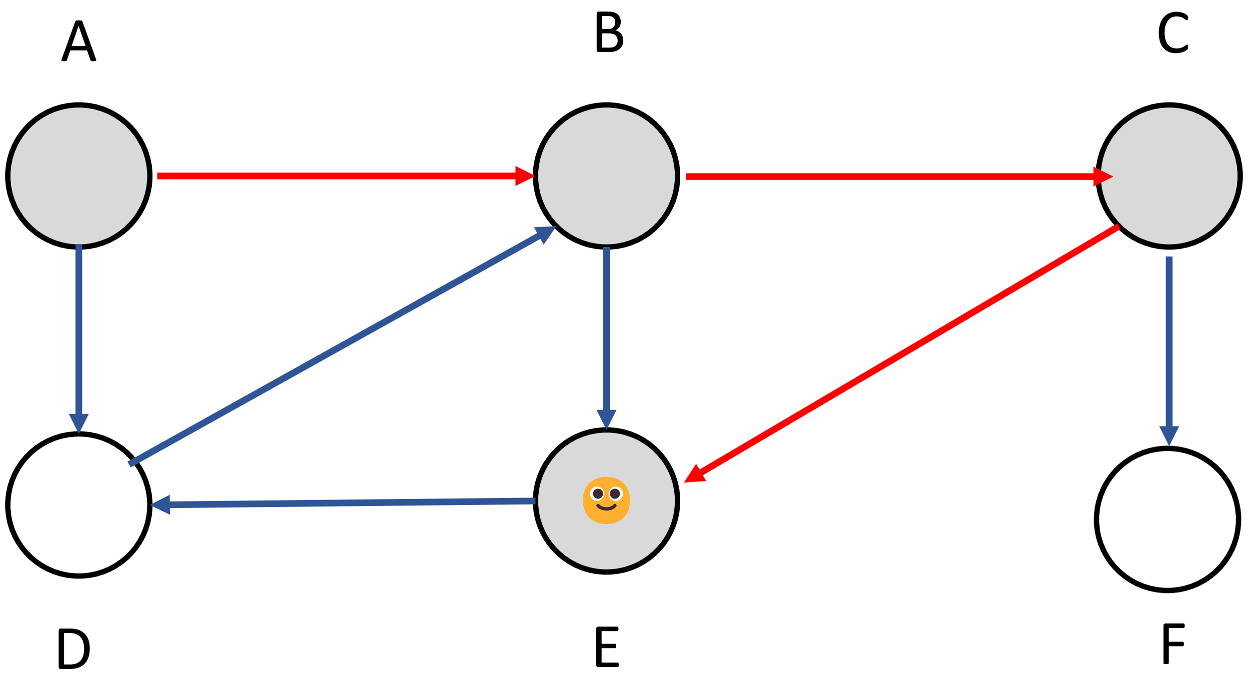
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 3

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



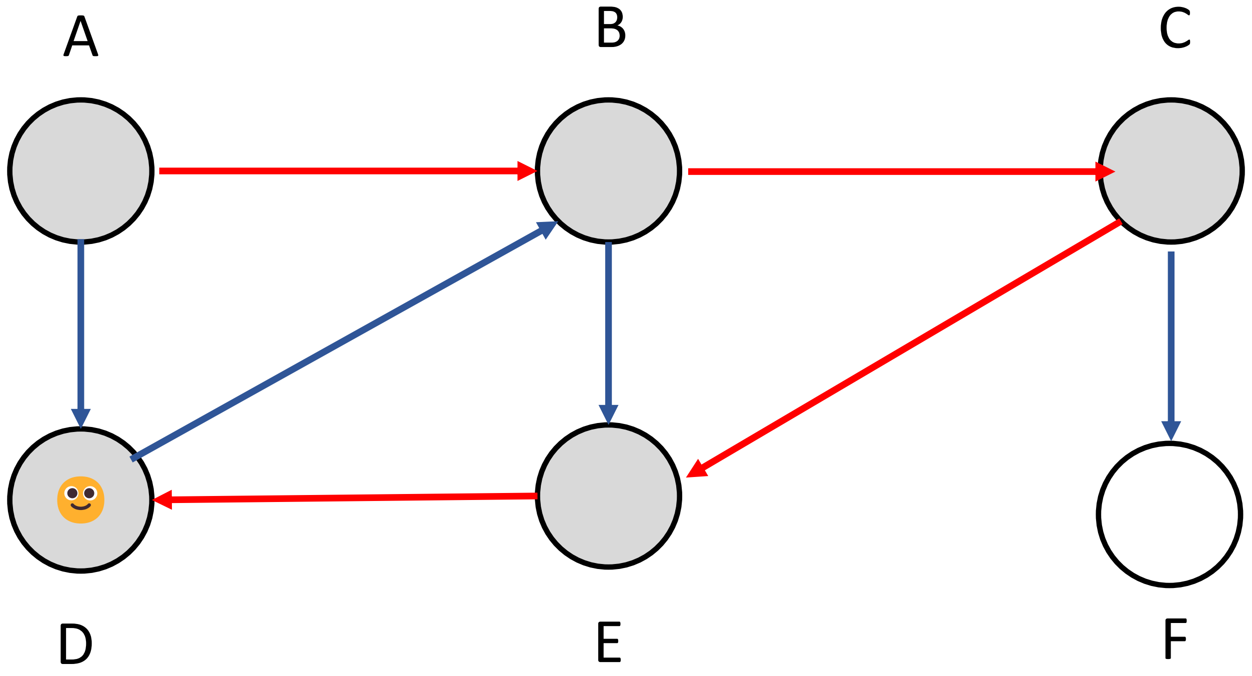
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 4

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



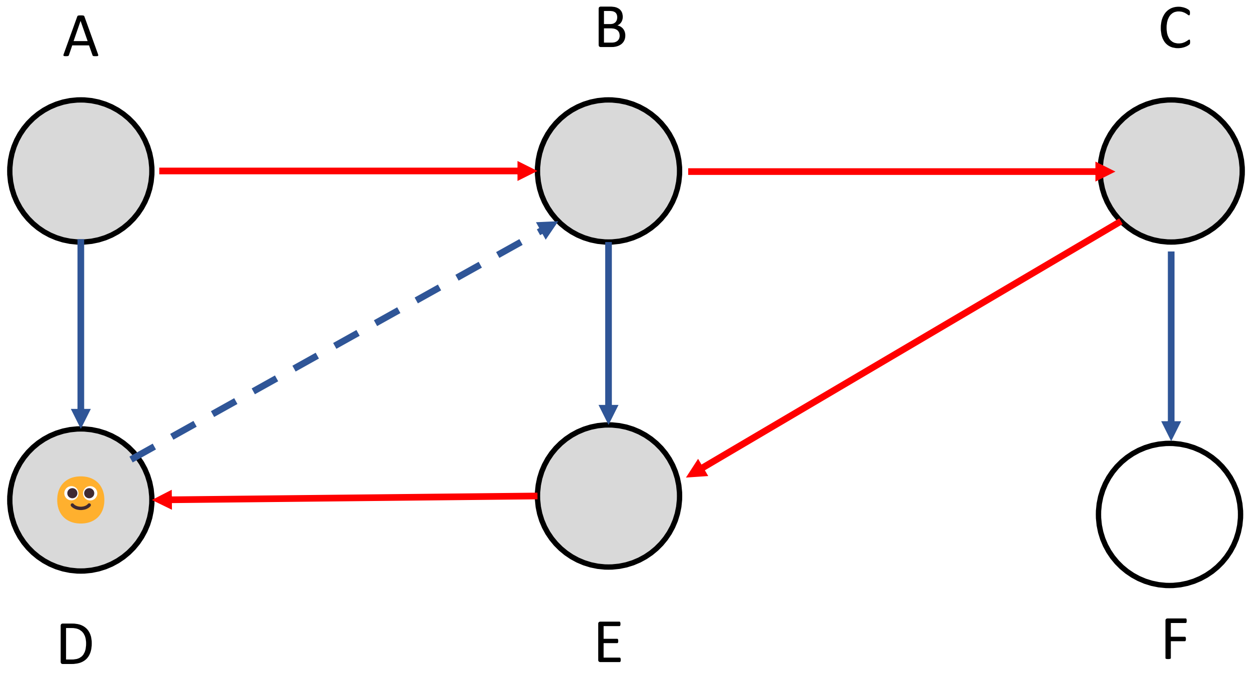
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 5

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



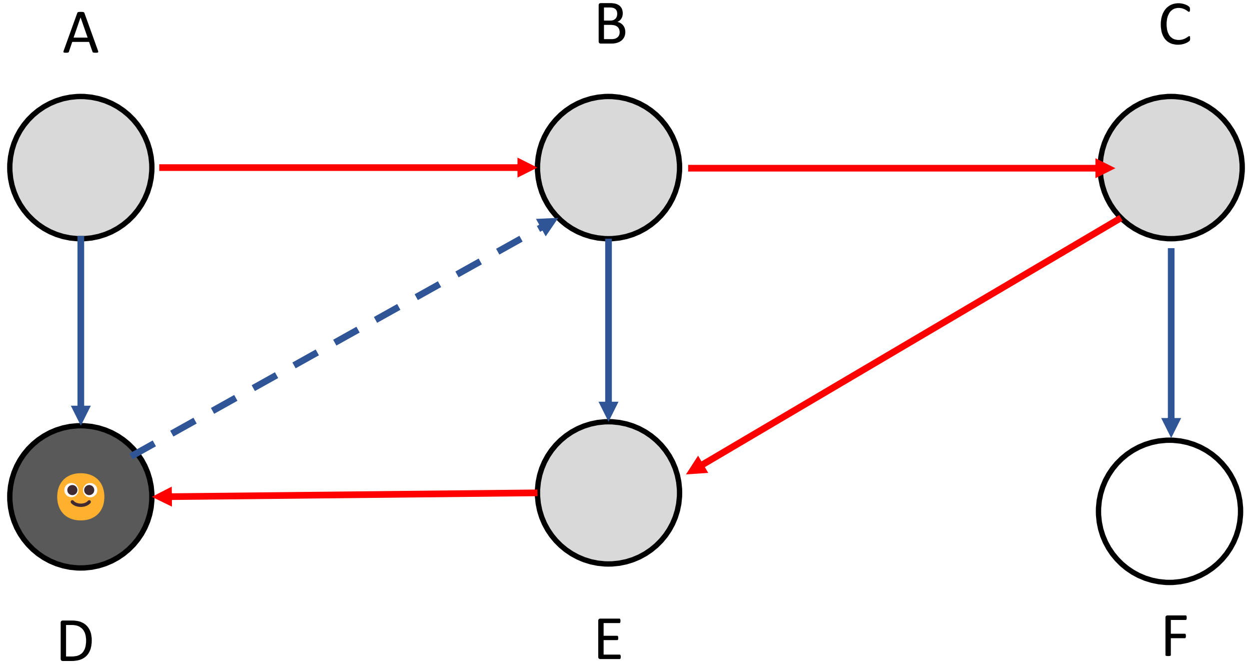
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 5

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



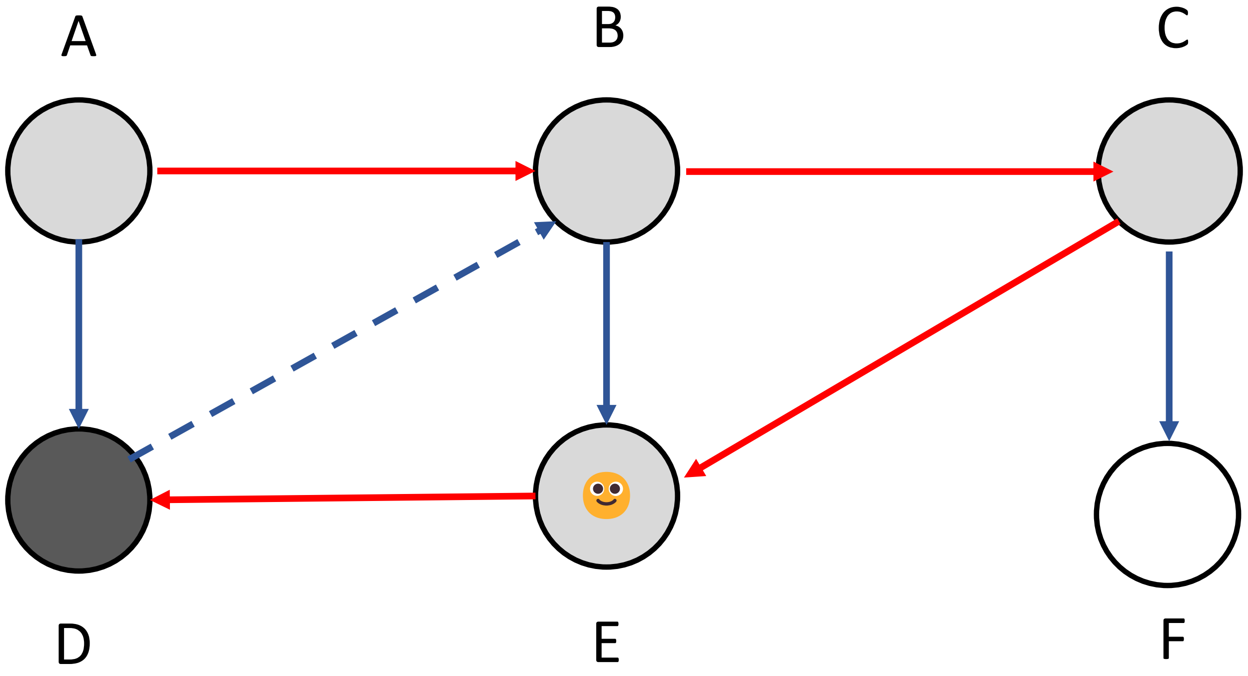
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 6

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



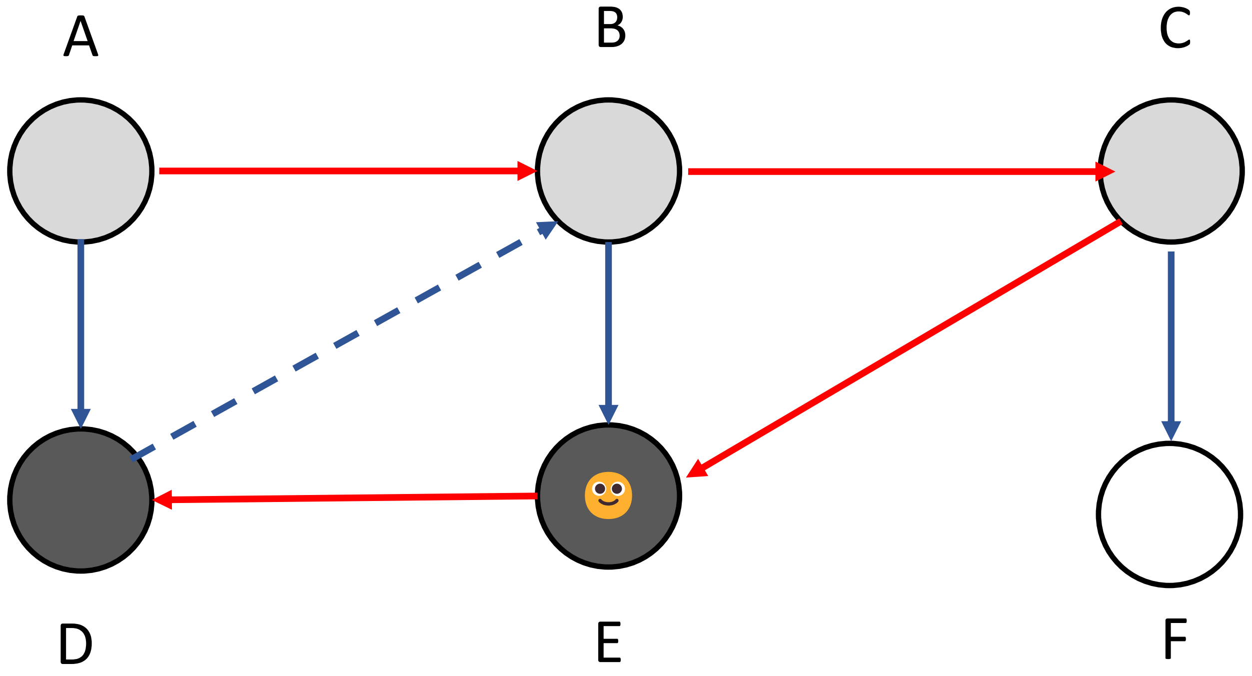
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 6

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



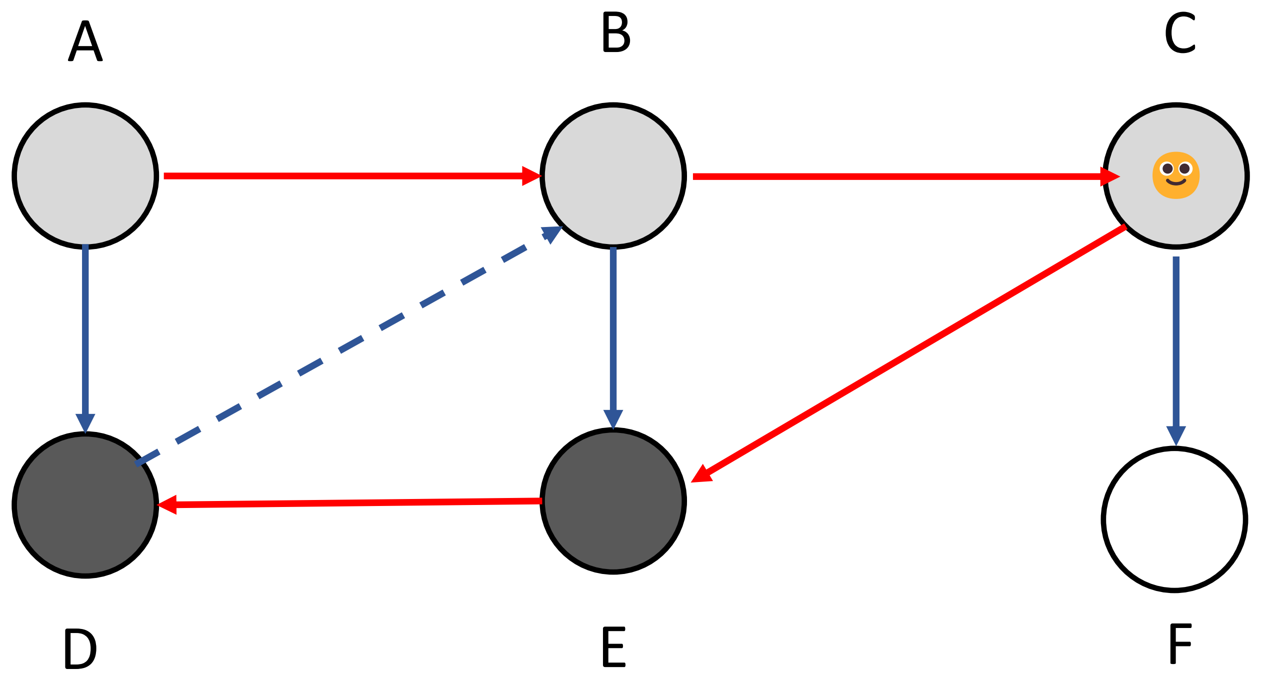
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 7

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



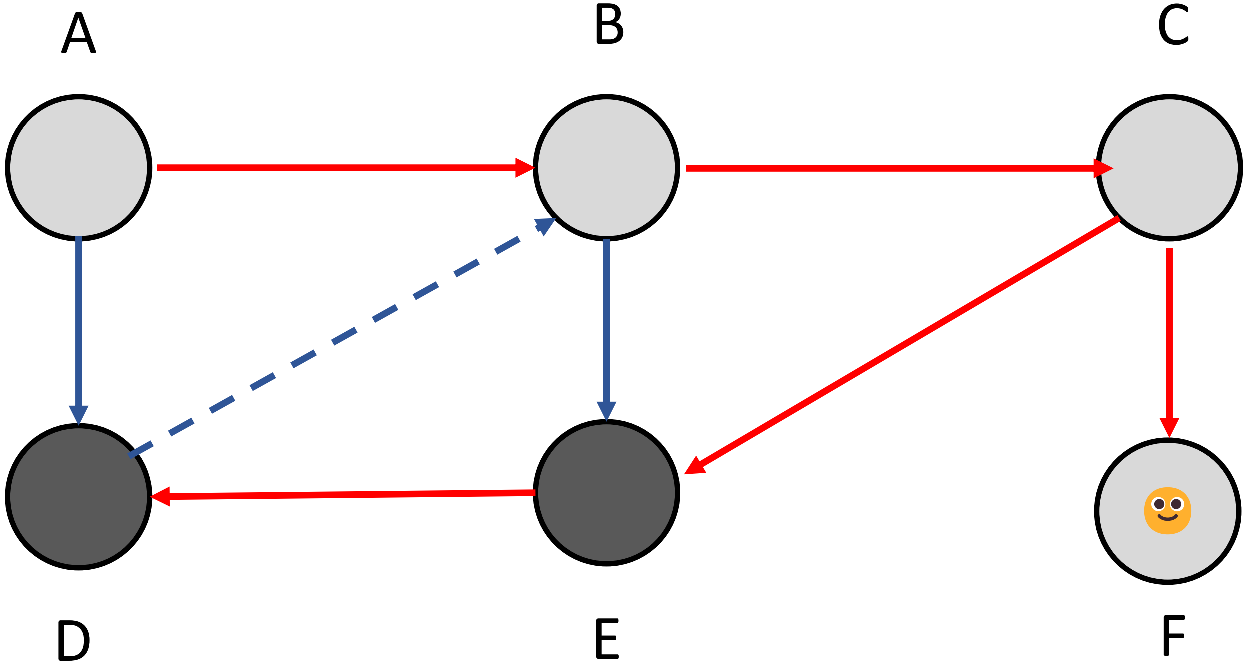
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 7

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



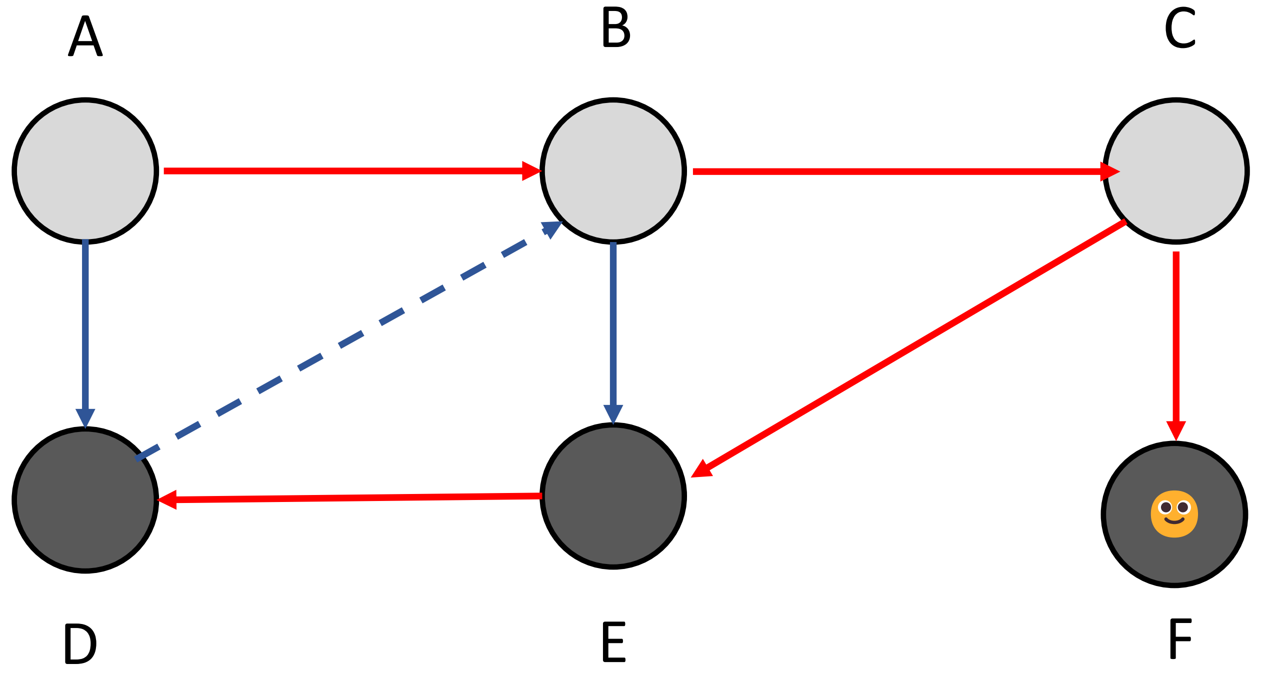
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 8

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



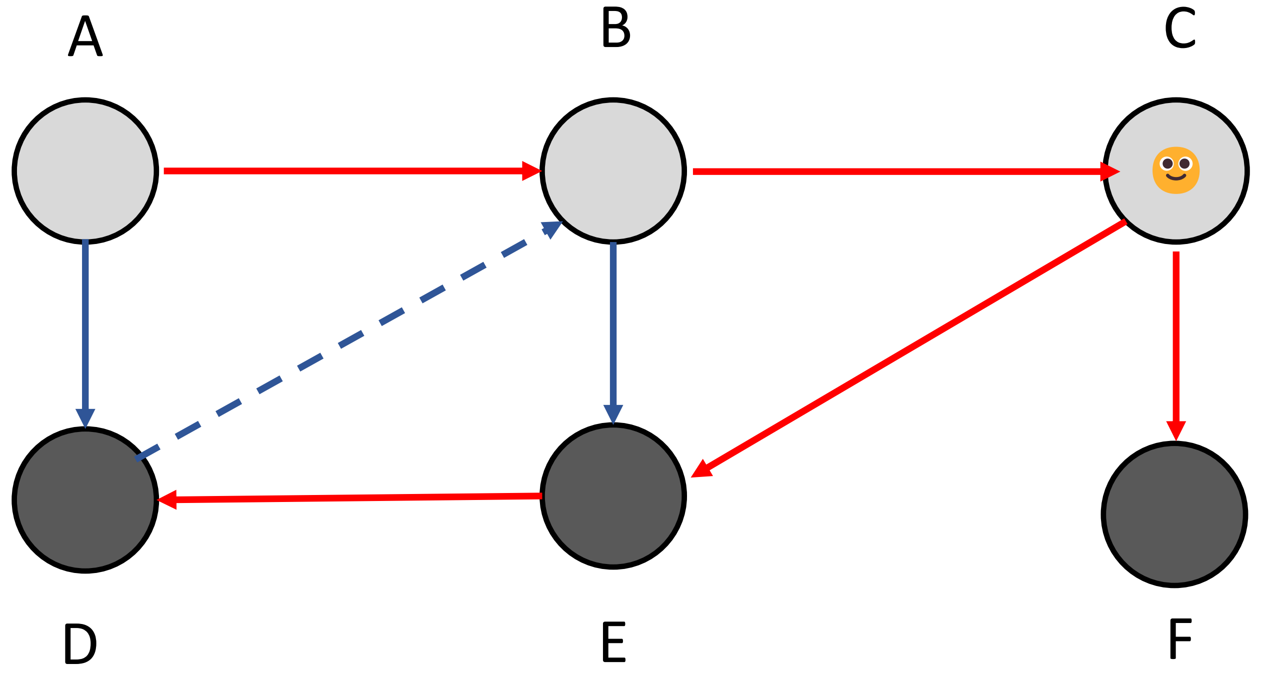
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 9

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



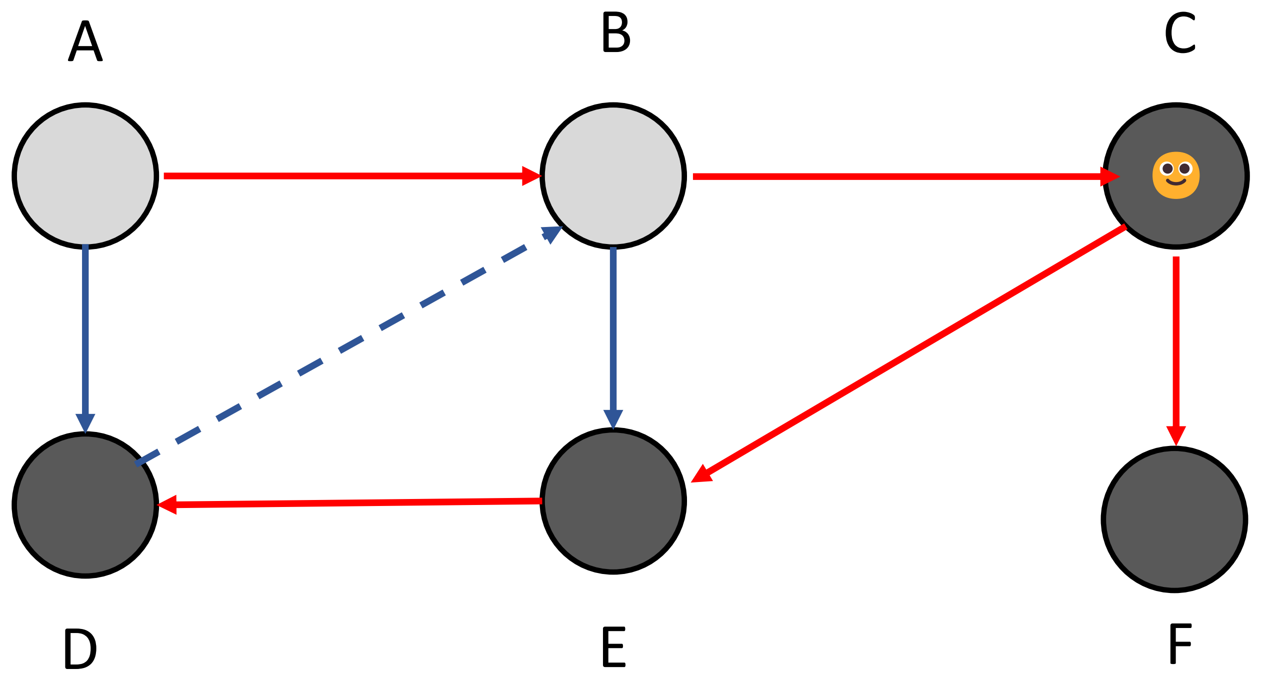
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 9

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



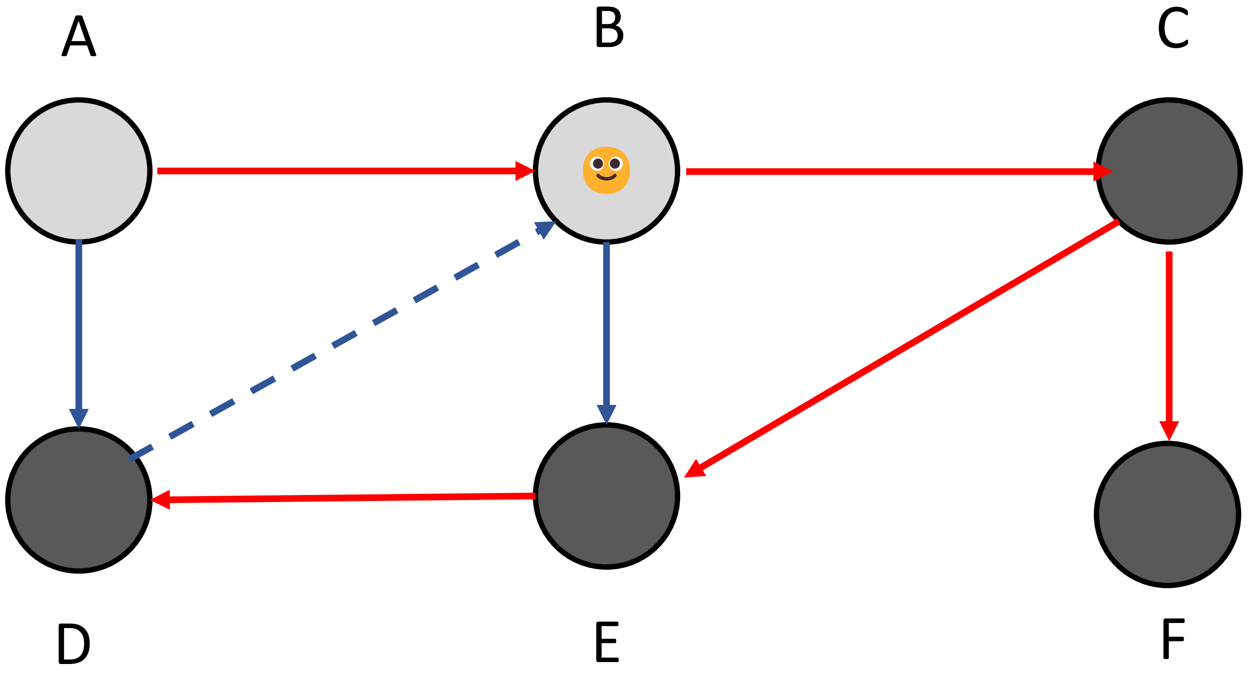
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 10

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



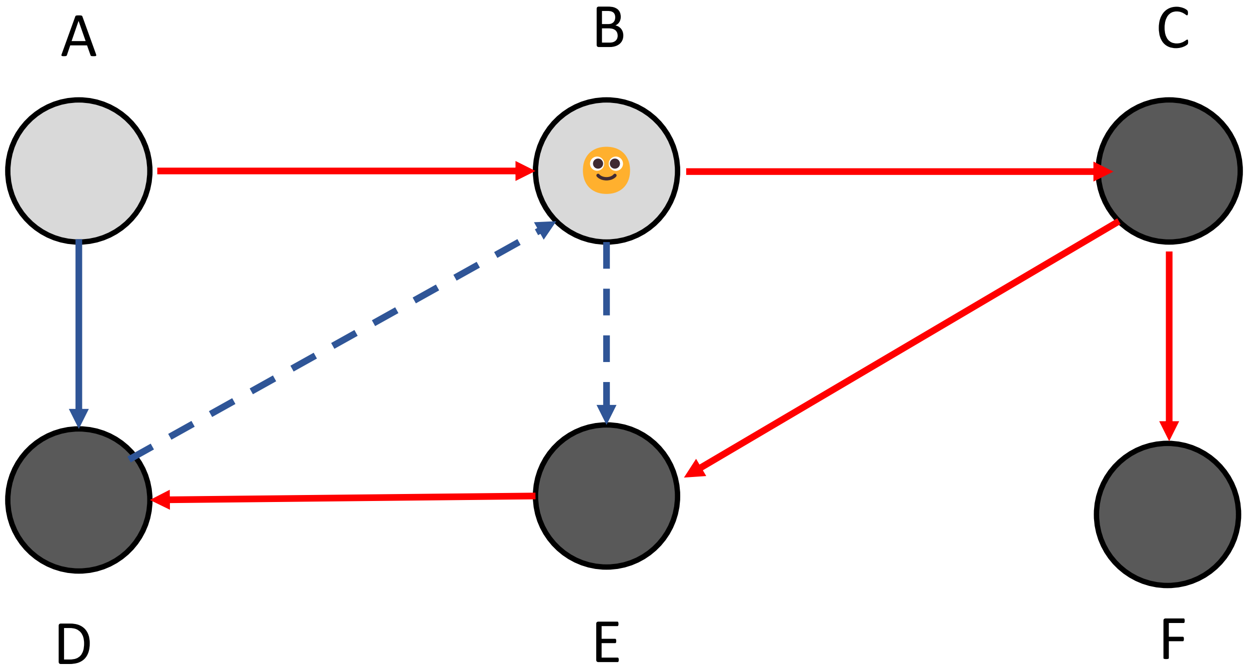
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 10

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



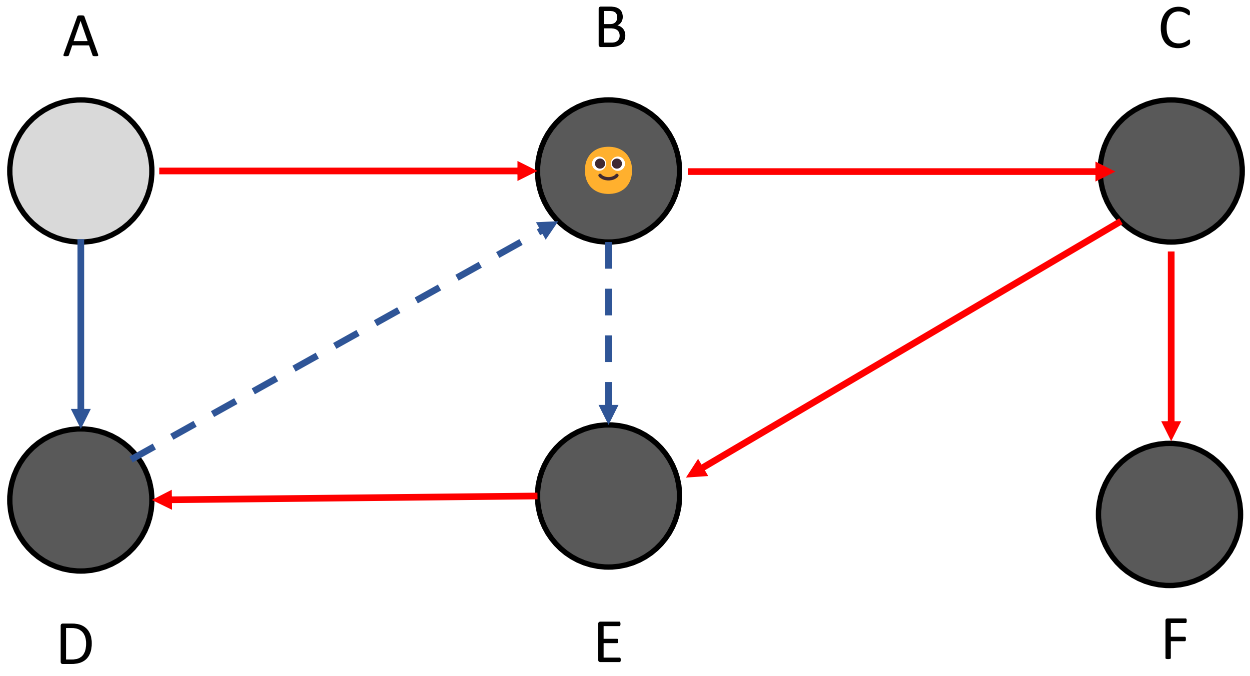
White: not visited
Grey: visited, but not "done"
Black: done (needs to pull back)

Time 10

We increment the clock
when a node's color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



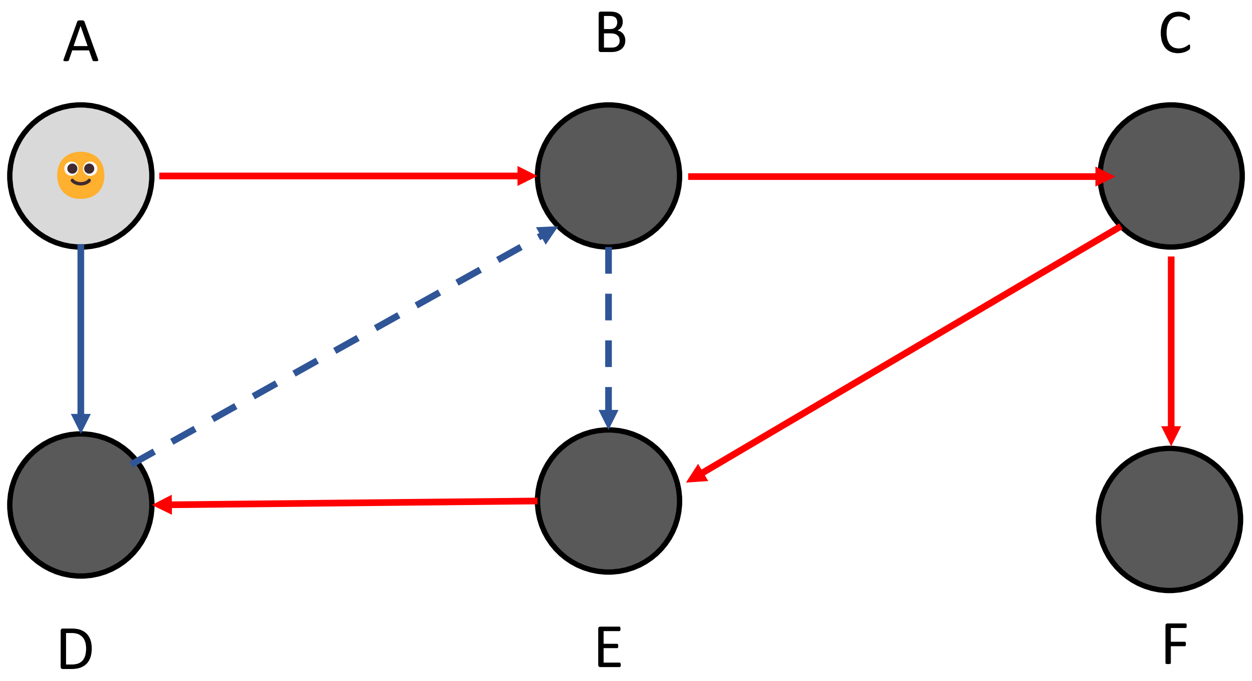
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 11

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



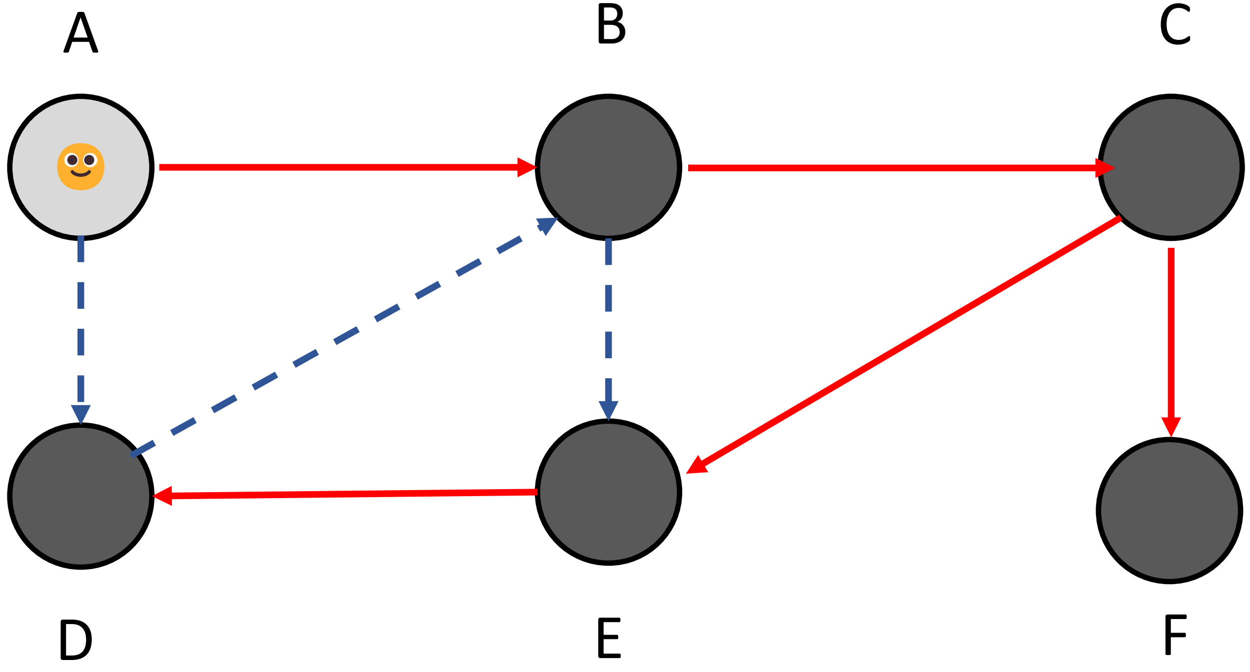
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 11

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



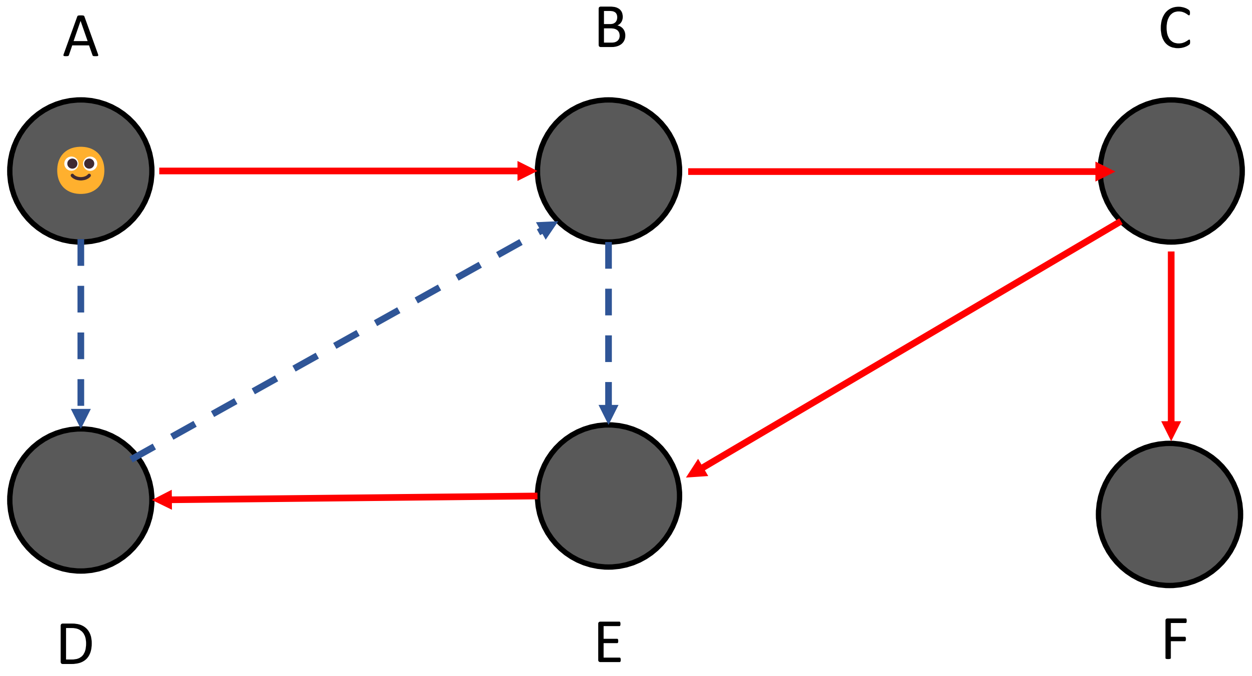
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 11

We increment the clock
when a node’s color changes

DFS Illustration

Assumptions:
Consider vertices in alphabetical order
Visit neighbors in alphabetical order



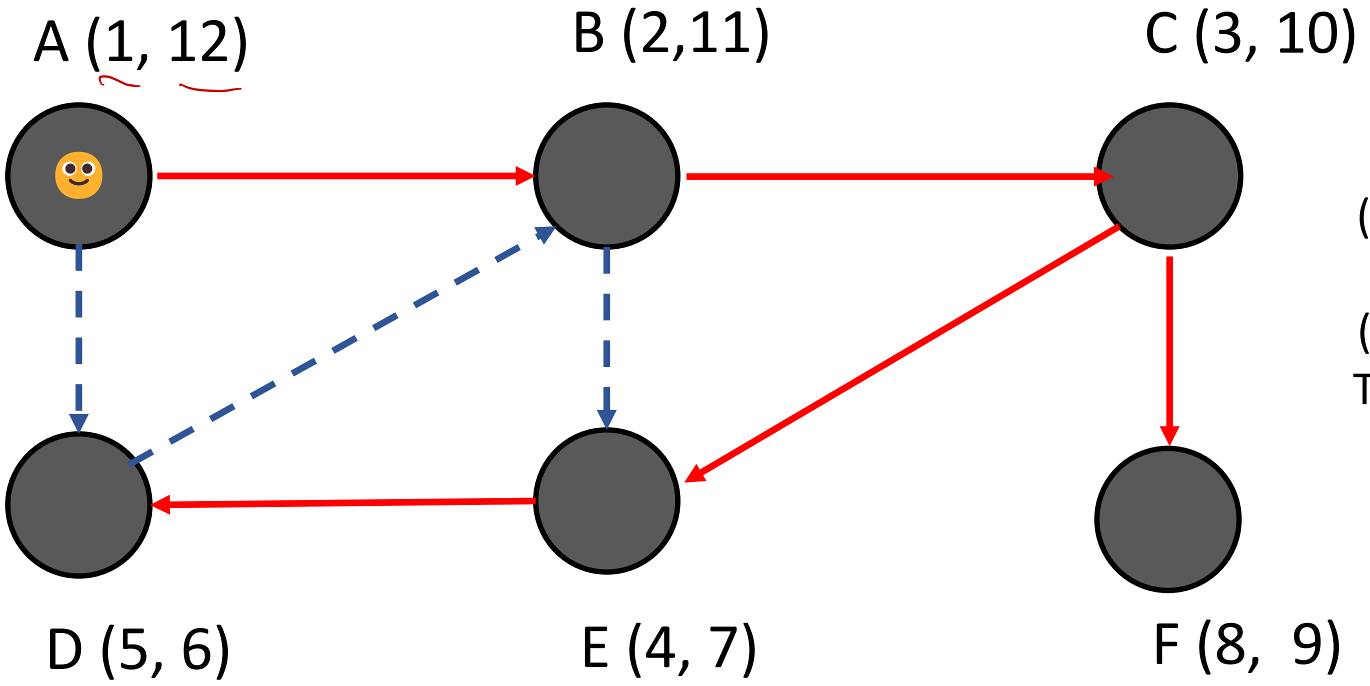
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)

Time 12

We increment the clock
when a node’s color changes

Time Stamp on Each Node

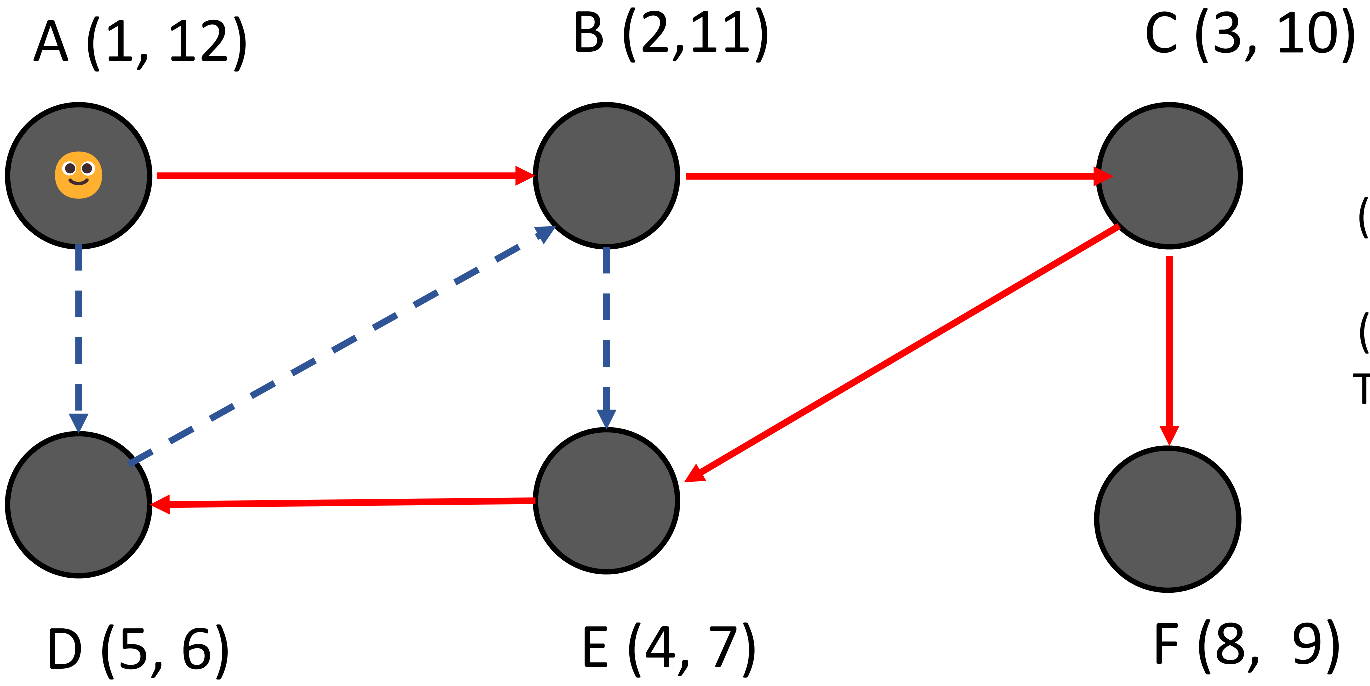
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)



(Discover time, Finish Time)
Or
(Time when it became grey,
Time when it became black)

Time Stamp on Each Node

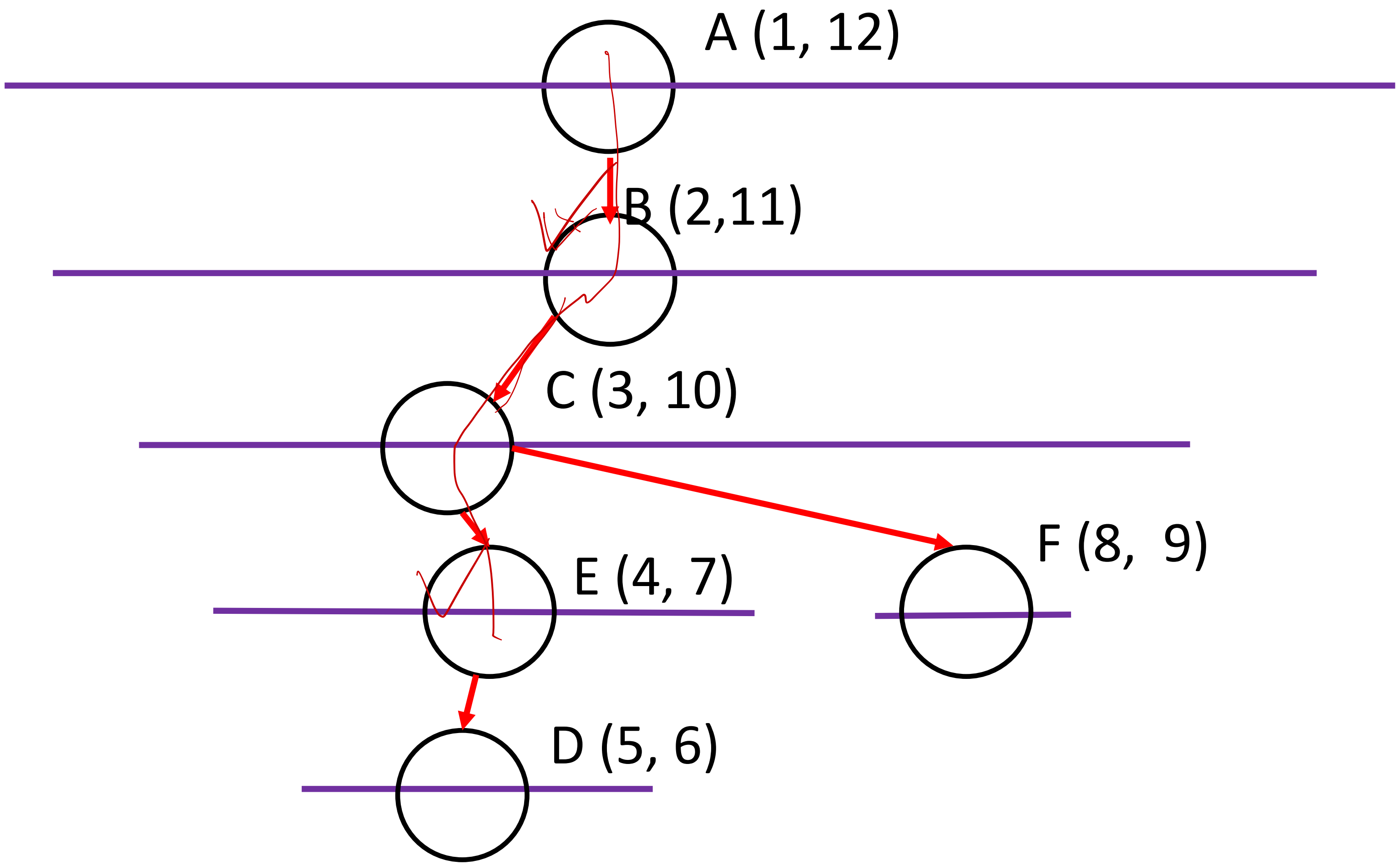
White: not visited
Grey: visited, but not “done”
Black: done (needs to pull back)



(Discover time, Finish Time)
Or
(Time when it became grey,
Time when it became black)

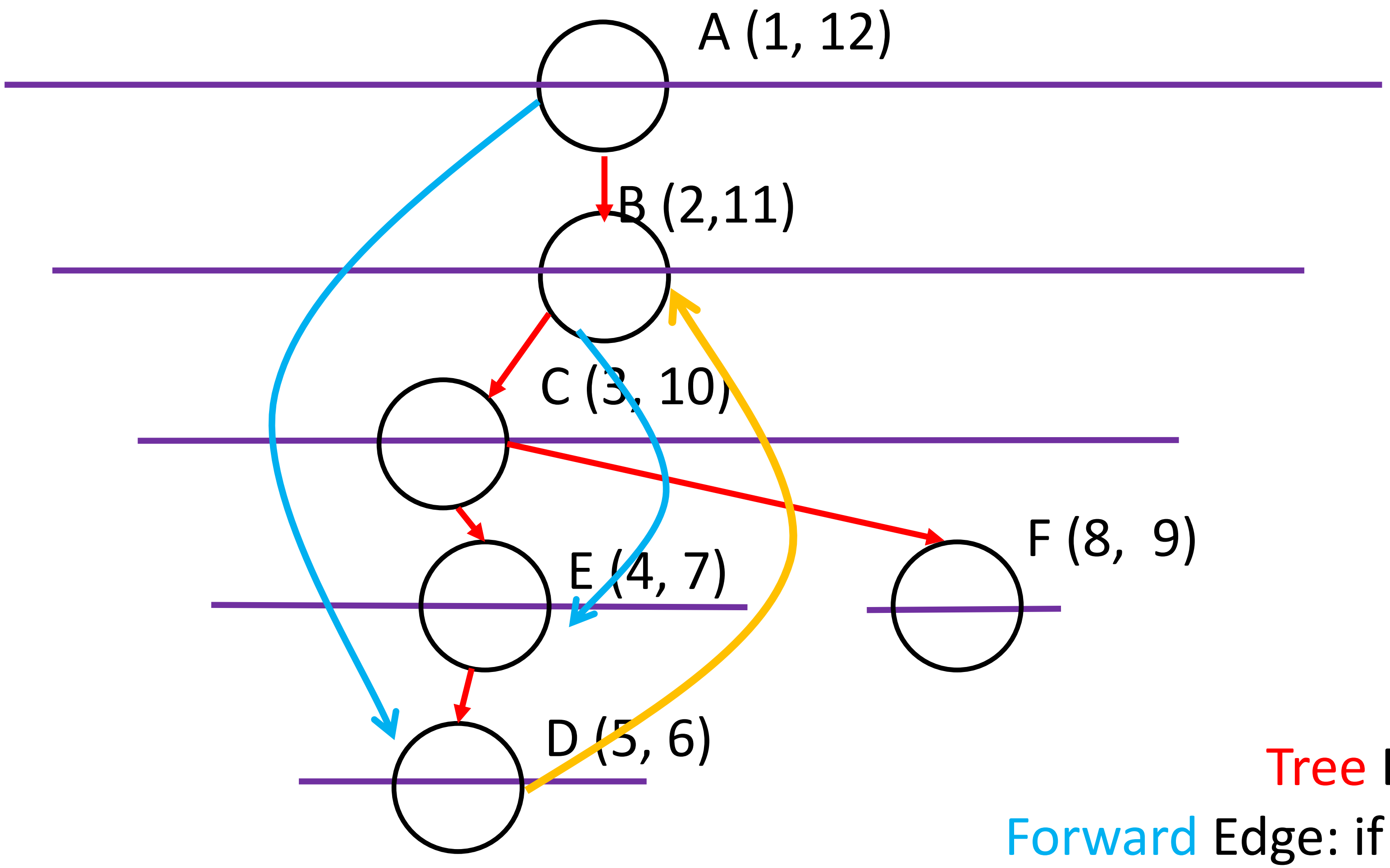
Nodes + red edges form a depth-first-tree (DFT)
There can be multiple DFTs, so a depth-first-forest (DFF).

Let’s Rearrange the Nodes



Do you see that for any pair of intervals, they are either disjoint, or one contains the other?

Let’s Rearrange the Nodes



(u, v) is

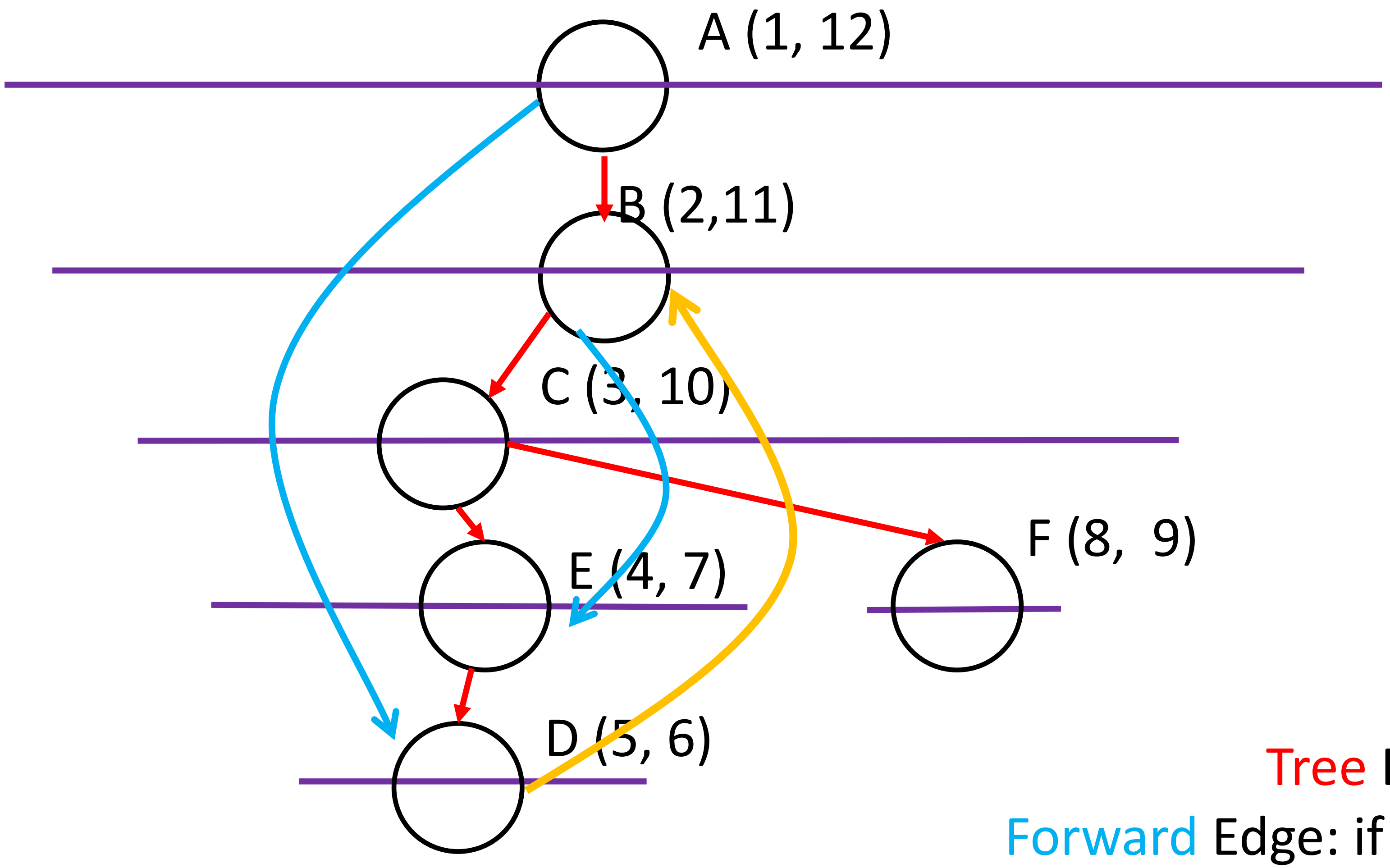
Tree Edge: if in DF Forest

Forward Edge: if non-tree & v is u’s descendant

Back Edge: if u is v’s descendant

Cross Edge: otherwise

Let’s Rearrange the Nodes



(u, v) is

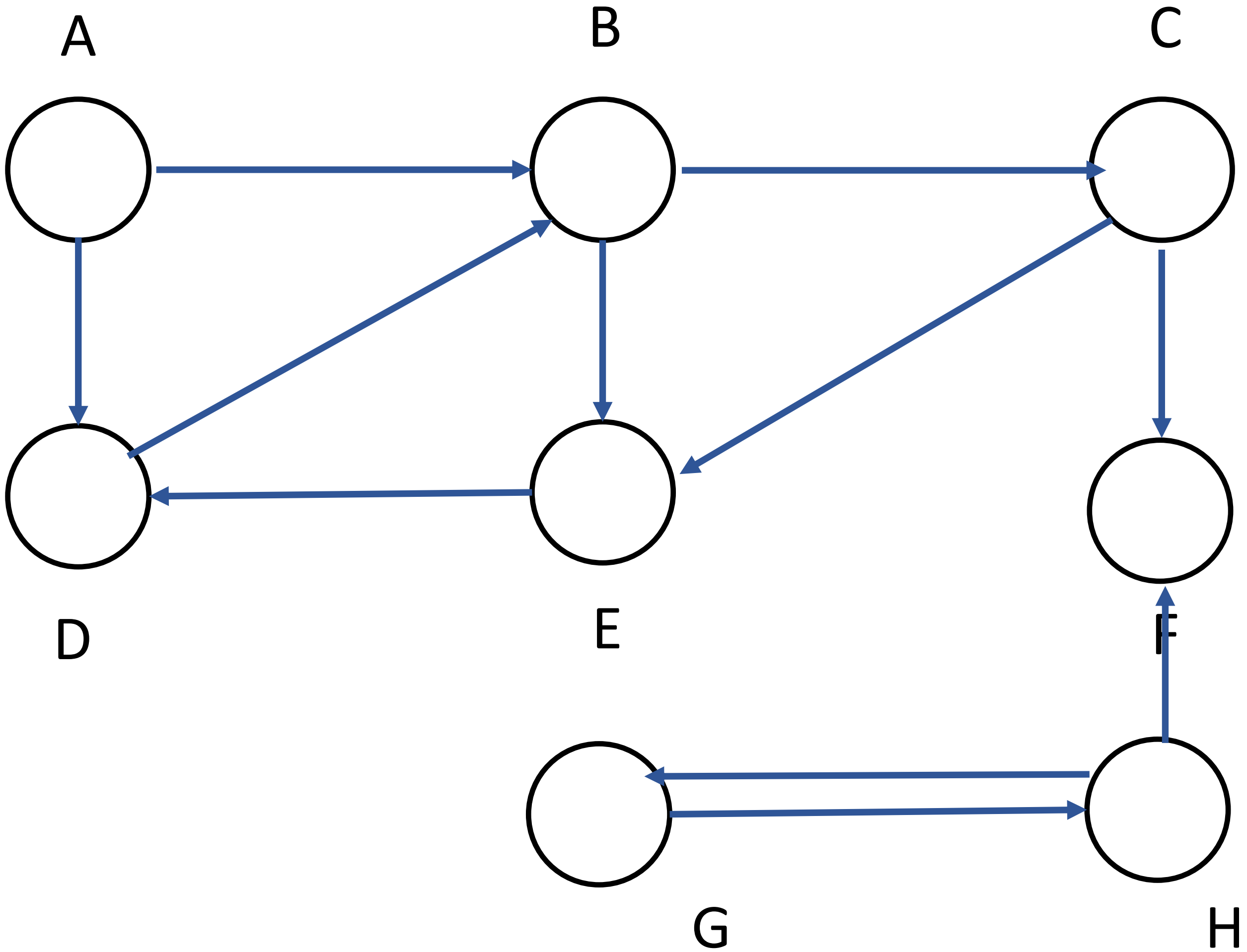
Tree Edge: if in DF Forest

Forward Edge: if non-tree & v is u’s descendant

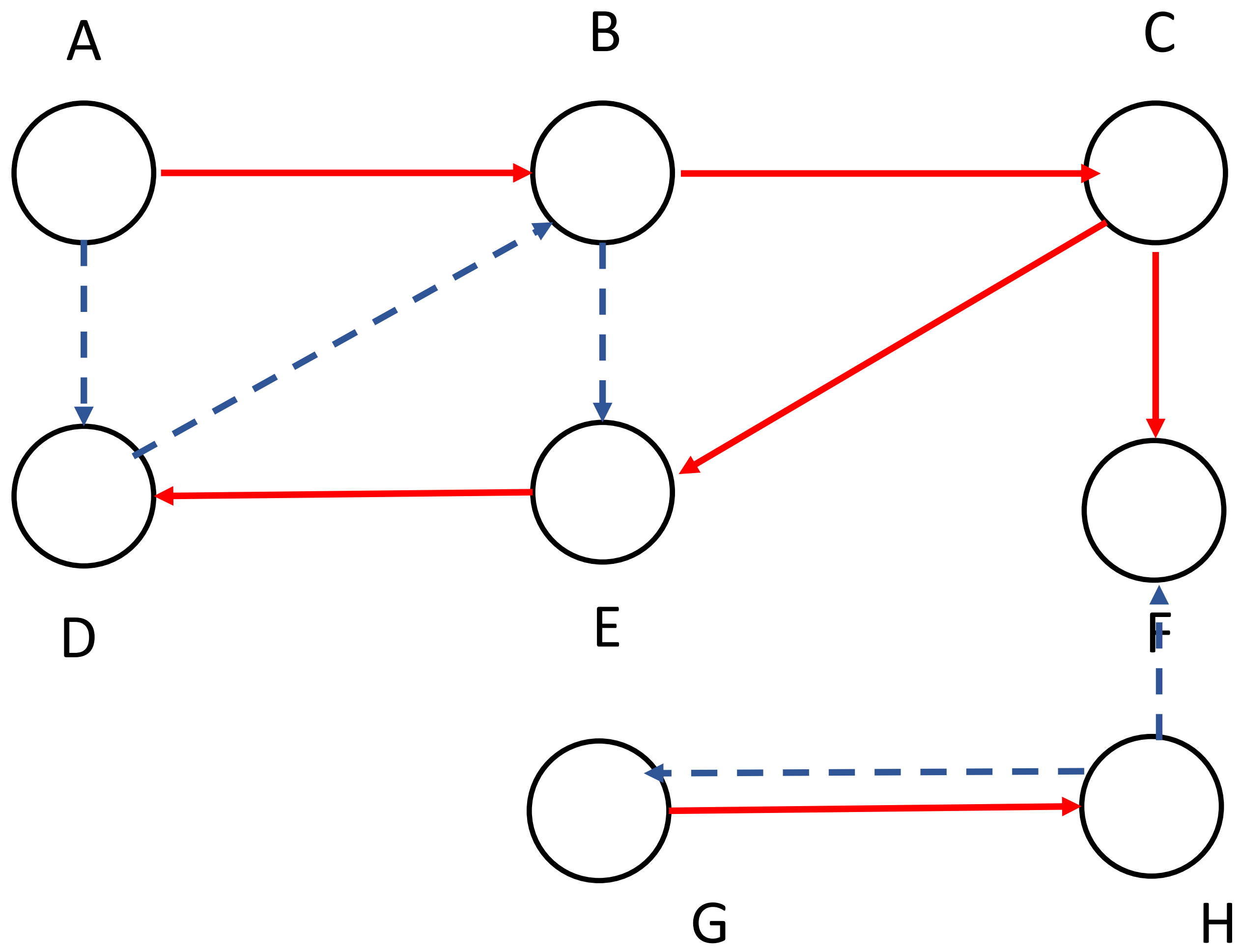
Back Edge: if u is v’s descendant

Cross Edge: otherwise

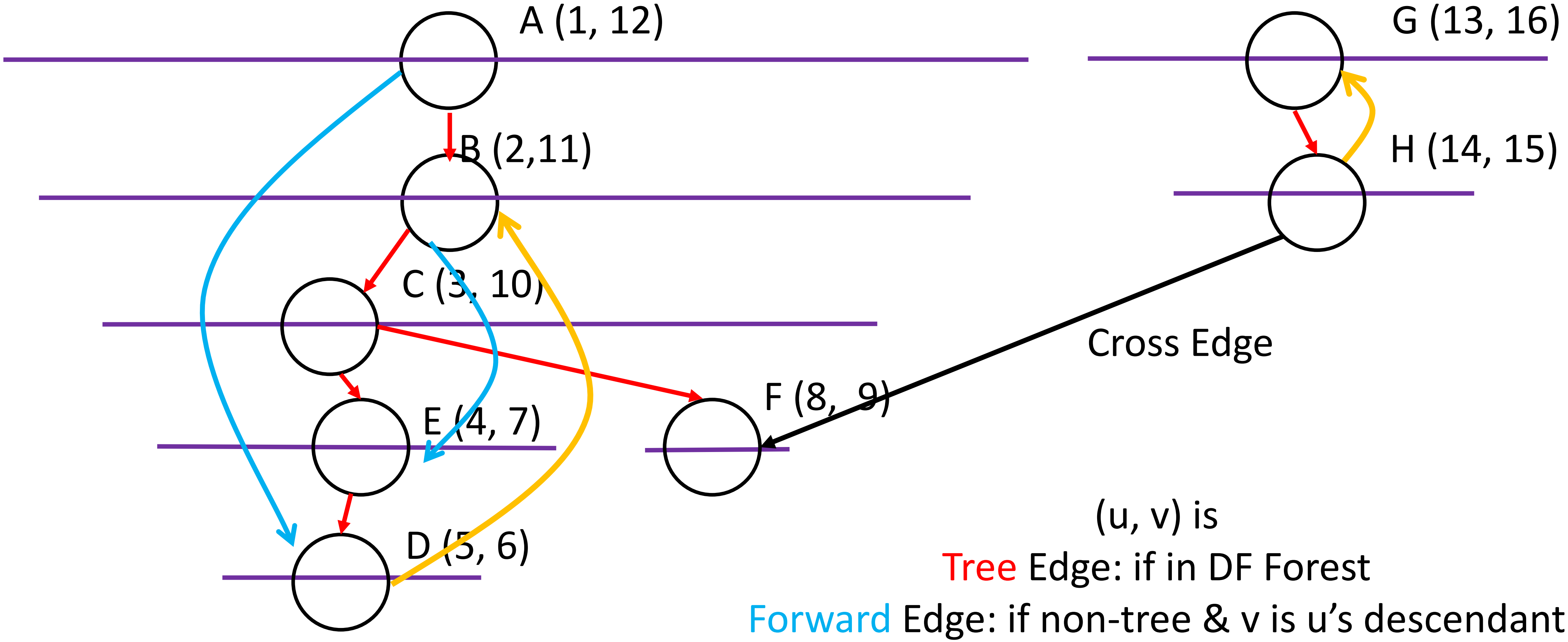
If Input was the Following:



Then, the resulting DFF is:



After rearranging:



(u, v) is

Tree Edge: if in DF Forest

Forward Edge: if non-tree & v is u's descendant

Back Edge: if non-tree & u is v's descendant

Cross Edge: otherwise

Parenthesis Theorem (Theorem 22.7)

- After running DFS, for any u, v in V, exactly one of the following three holds:
  - [u.d, u.f] and [v.d, v.f] are entirely disjoint
    - Neither u nor v is a descendant of the other in DFF
  - [u.d, u.f] is contained in [v.d, v.f]
    - u is a descendant of v in a DFT
  - [v.d, v.f] is contained in [u.d, u.f]
    - v is a descendant of u in a DFT

u.d: u’s discover time; u.f: u’s finish time.

White Path Theorem (Theorem 22.9)

- In the DFF, v is a descendant of u iff at time u.d (u's discover time), there is a path from u to v consisting entirely of white vertices

Three Applications

- How to determine if G has a cycle or not.

- Topological Sort

- Finding Strongly Connected Components

How to determine if G has a cycle or not

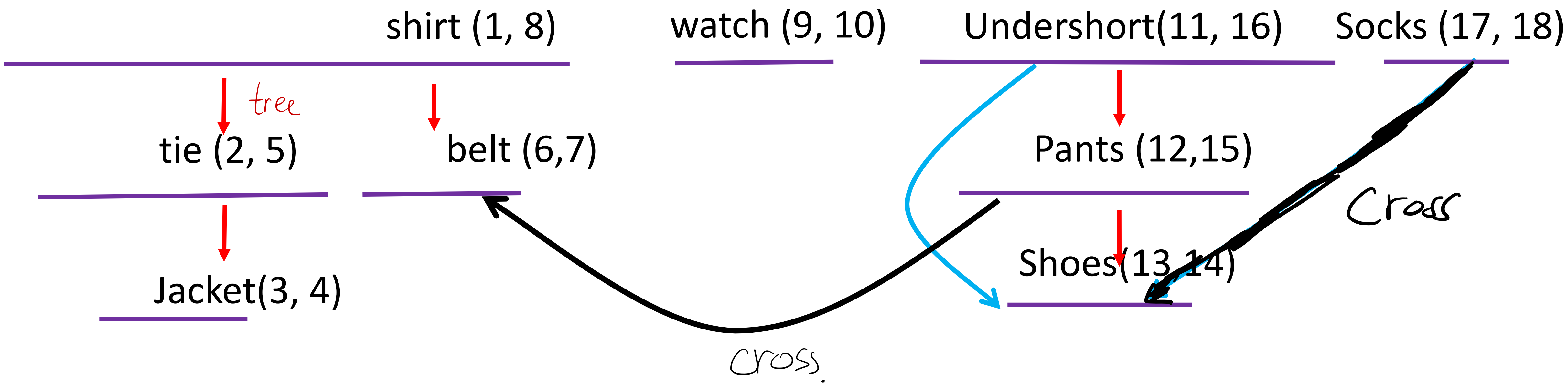
- Lemma 22.11. A directed graph G is acyclic if and only if a depth-first search of G yields no back edge.

- Proof.

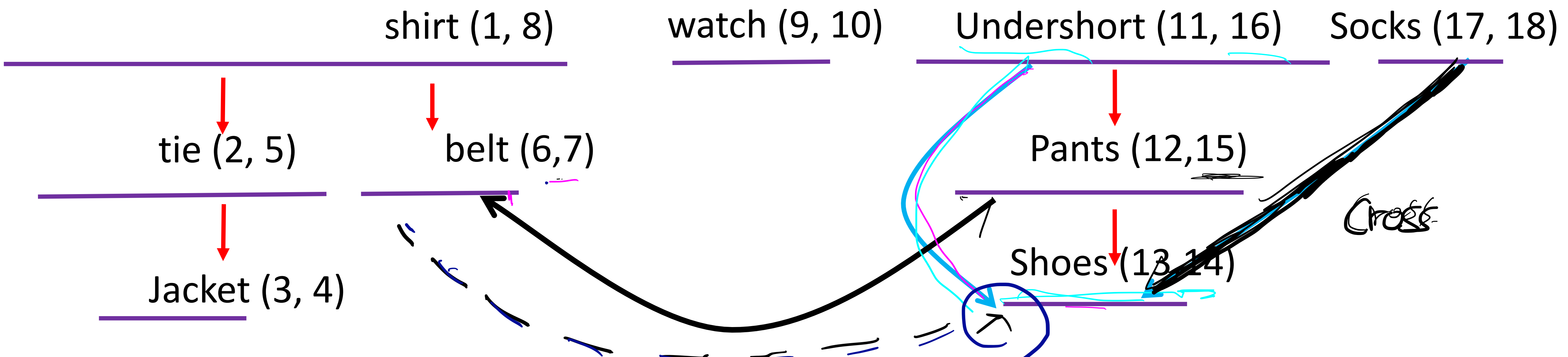
-> Back edge implies a cycle

<- Use the white-path theorem

Topological Sort (Why It Works)



Topological Sort (Why It Works)



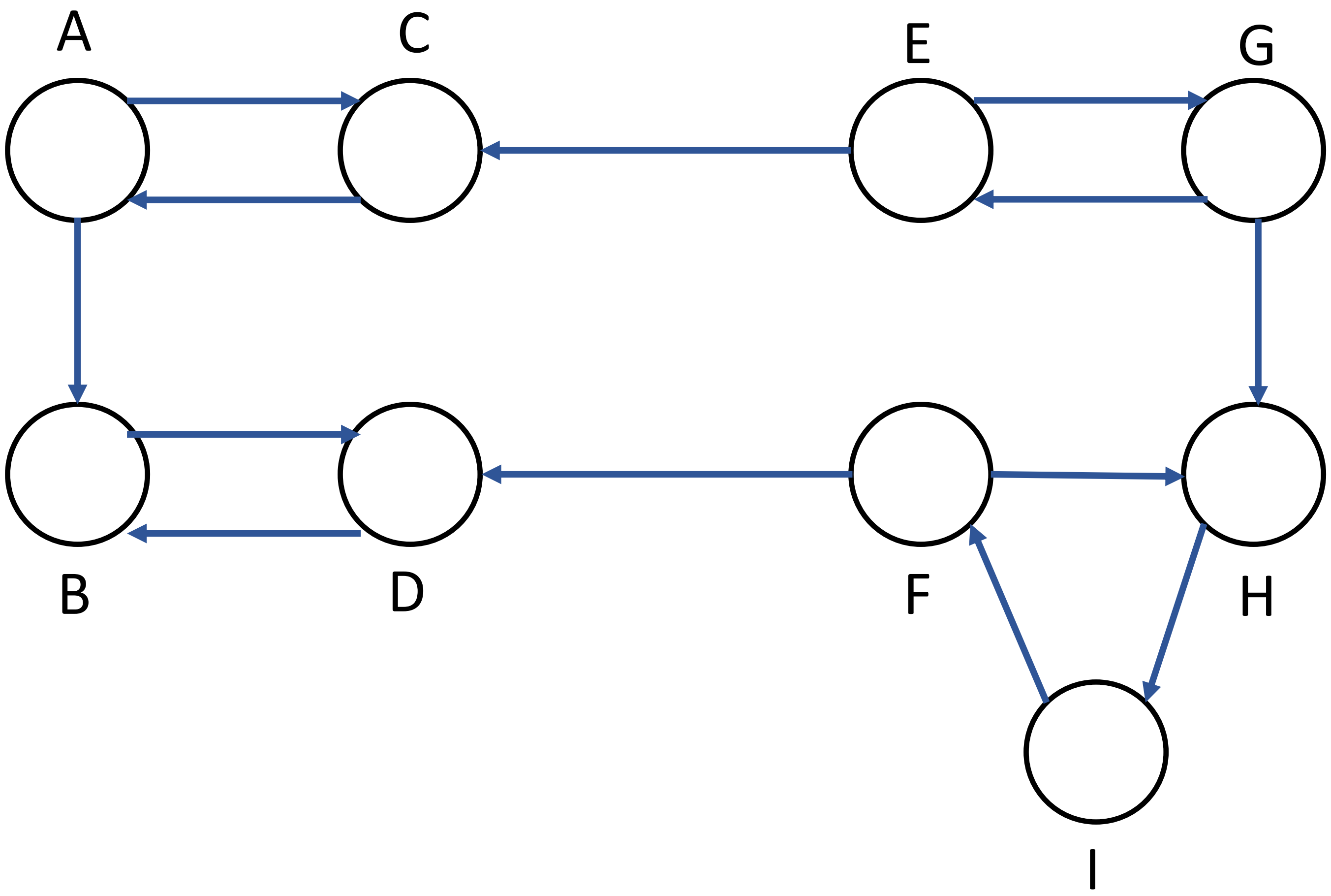
Tree edge (u, v): u finishes ~~before~~ after v

Forward edge (u, v): u finishes ~~before~~ after v

Cross edge (u, v): u finishes ~~before~~ after v: otherwise, when we discover u, v wasn't discovered. So, due to the white path theorem, v must become u's descendant

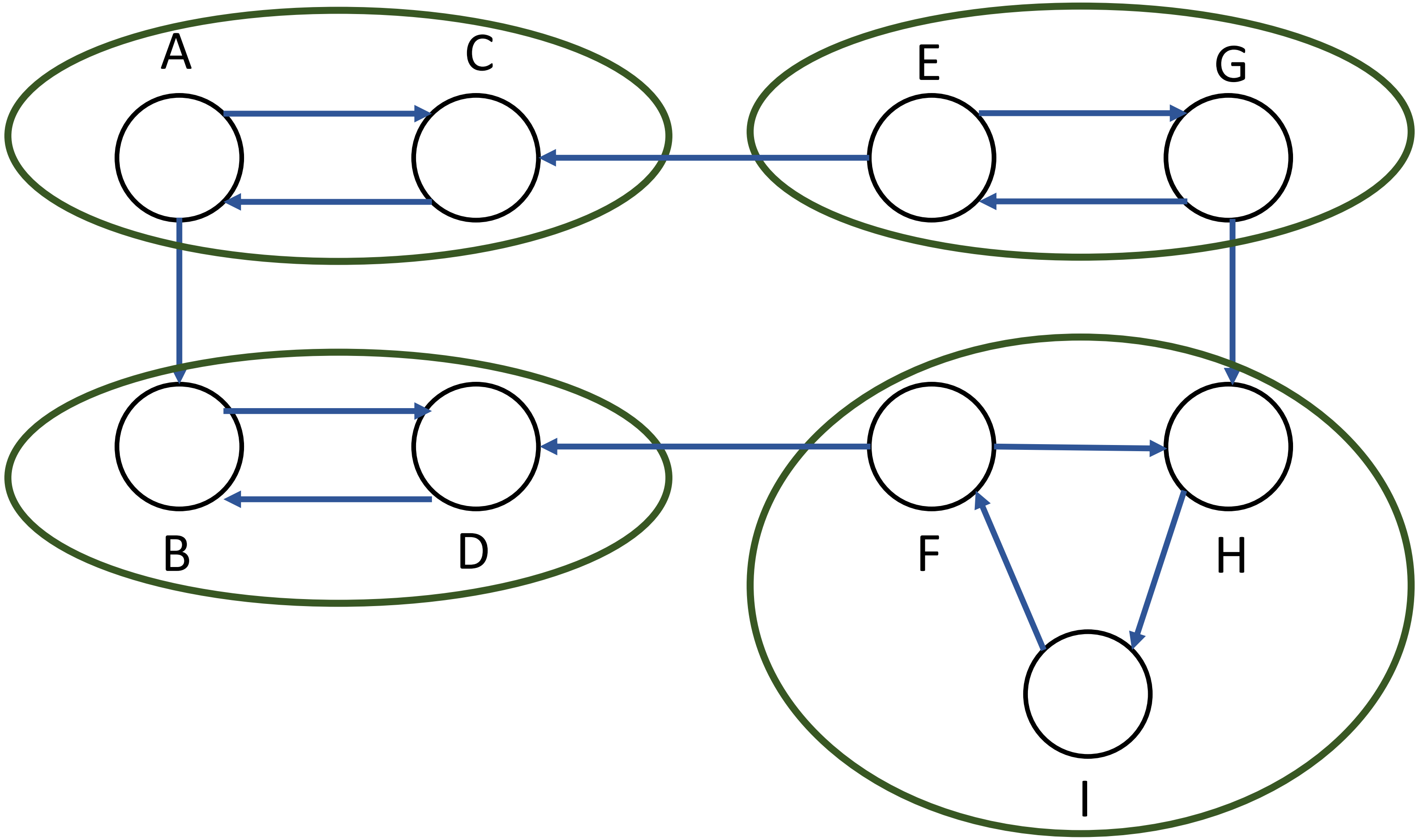
Algorithm for Computing SCCs
(Illustration and Intuitions)

Input Graph

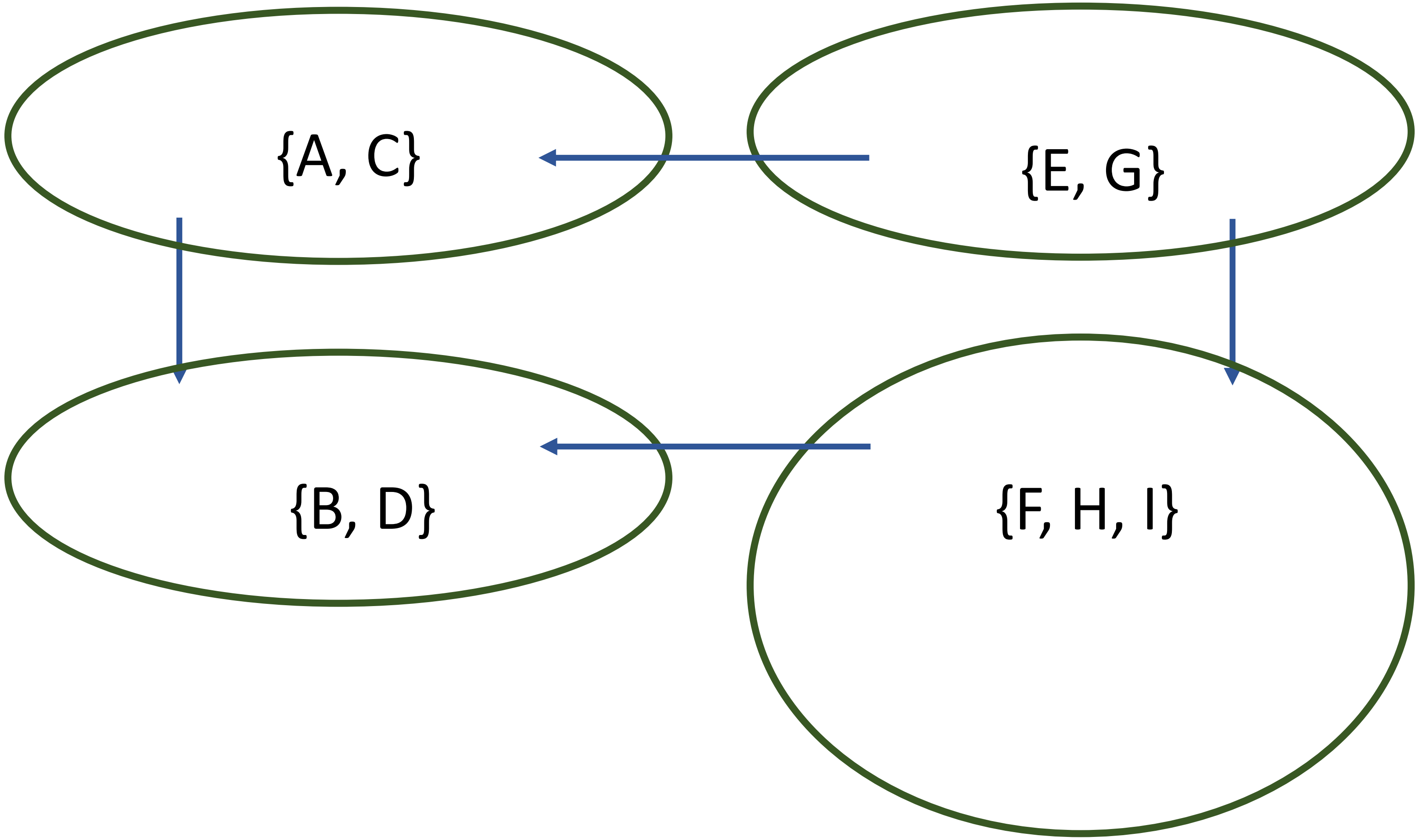


Desired Output

{A, C}, {E, G}, {B, D}, {F, I, H}

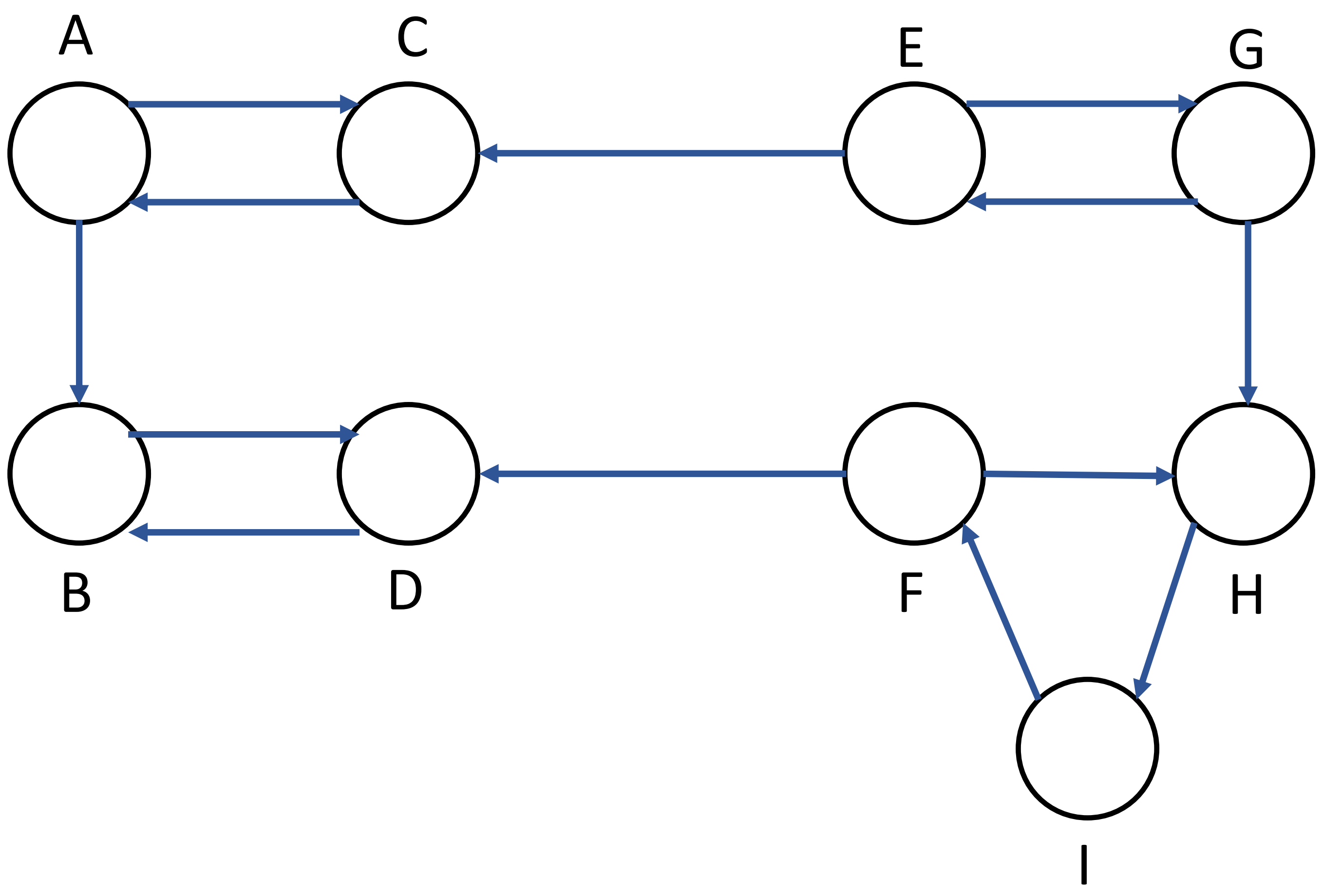


Note: Component Graph

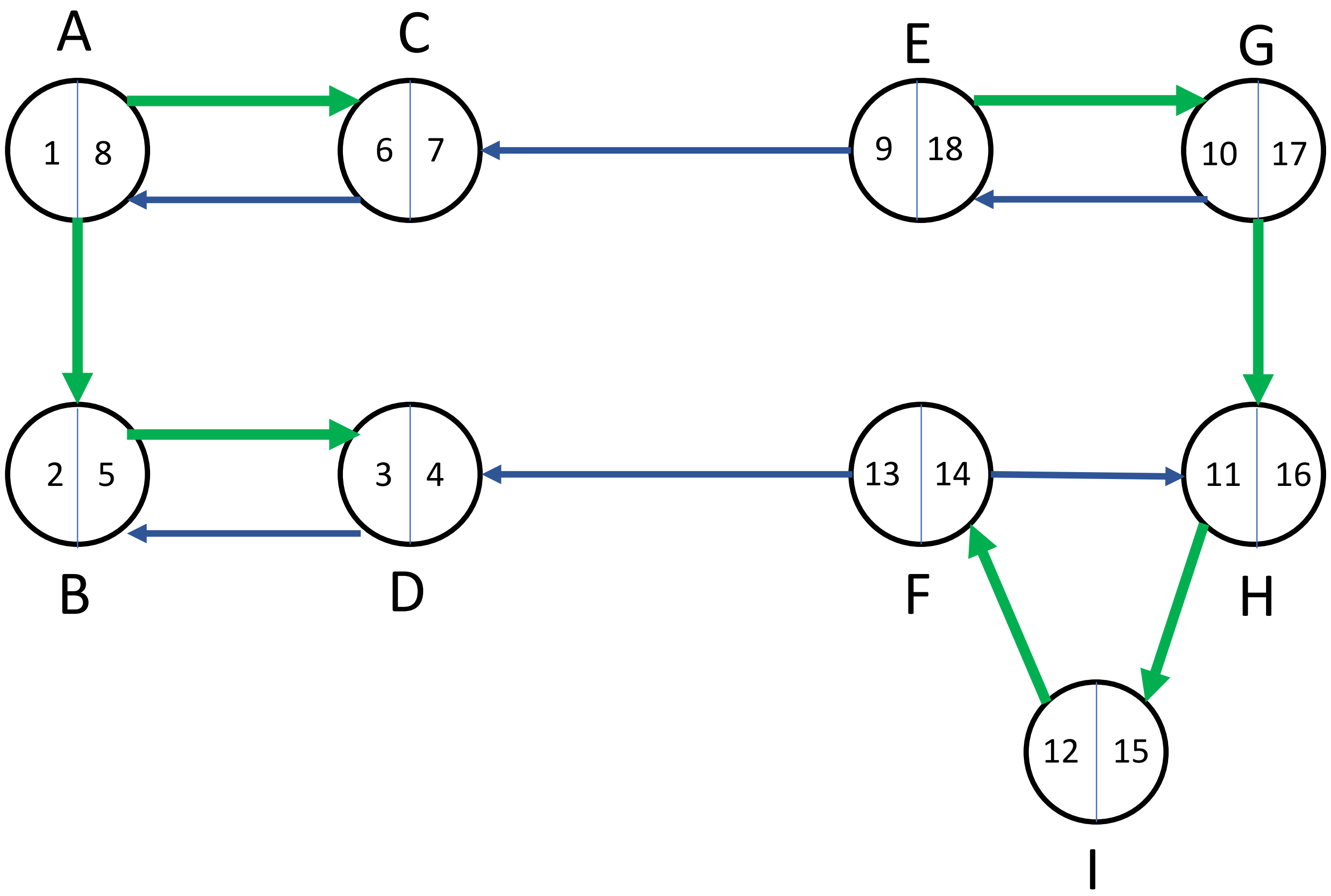


The component graph must be a DAG

Input Graph

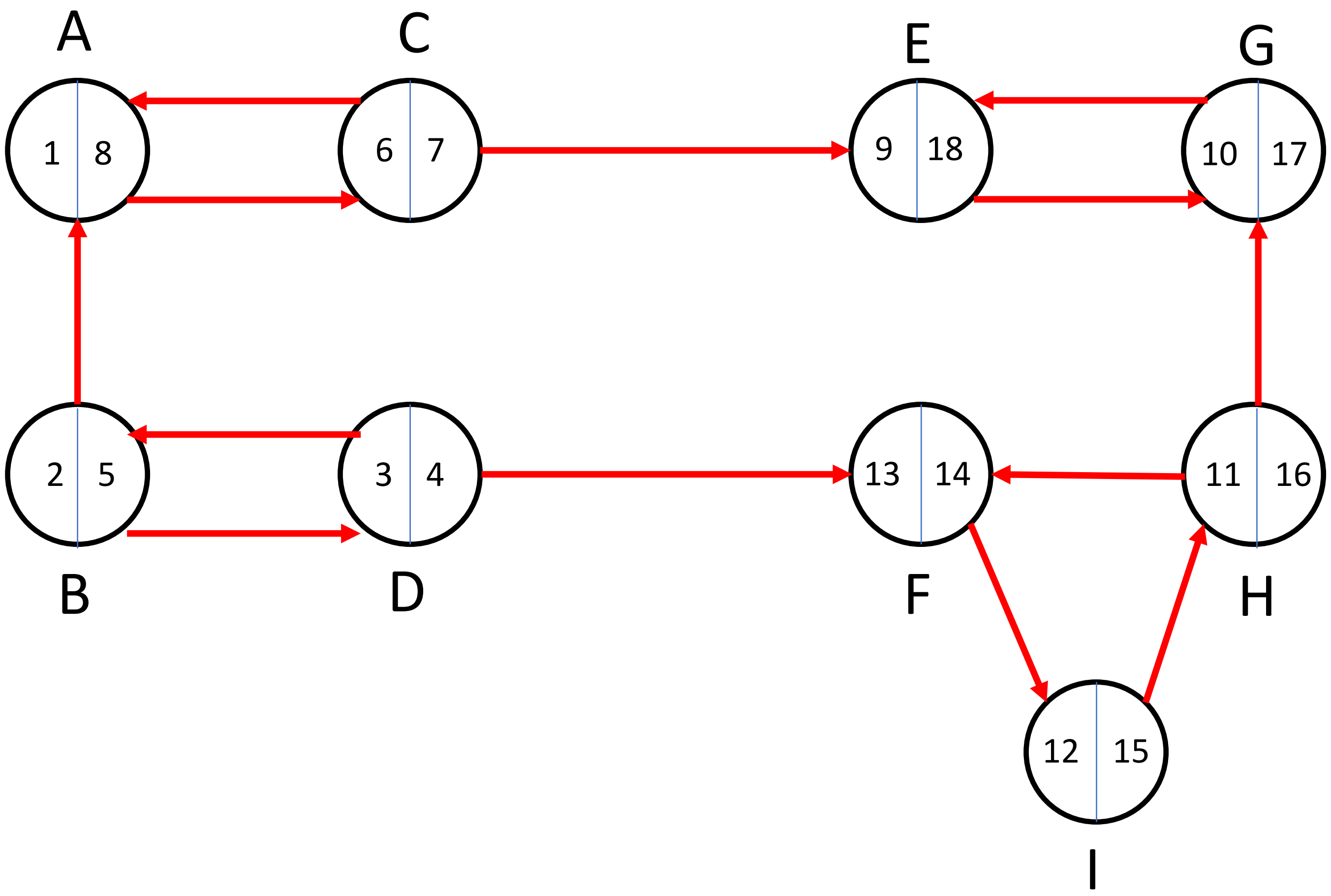


Algo: 1. Run DFS



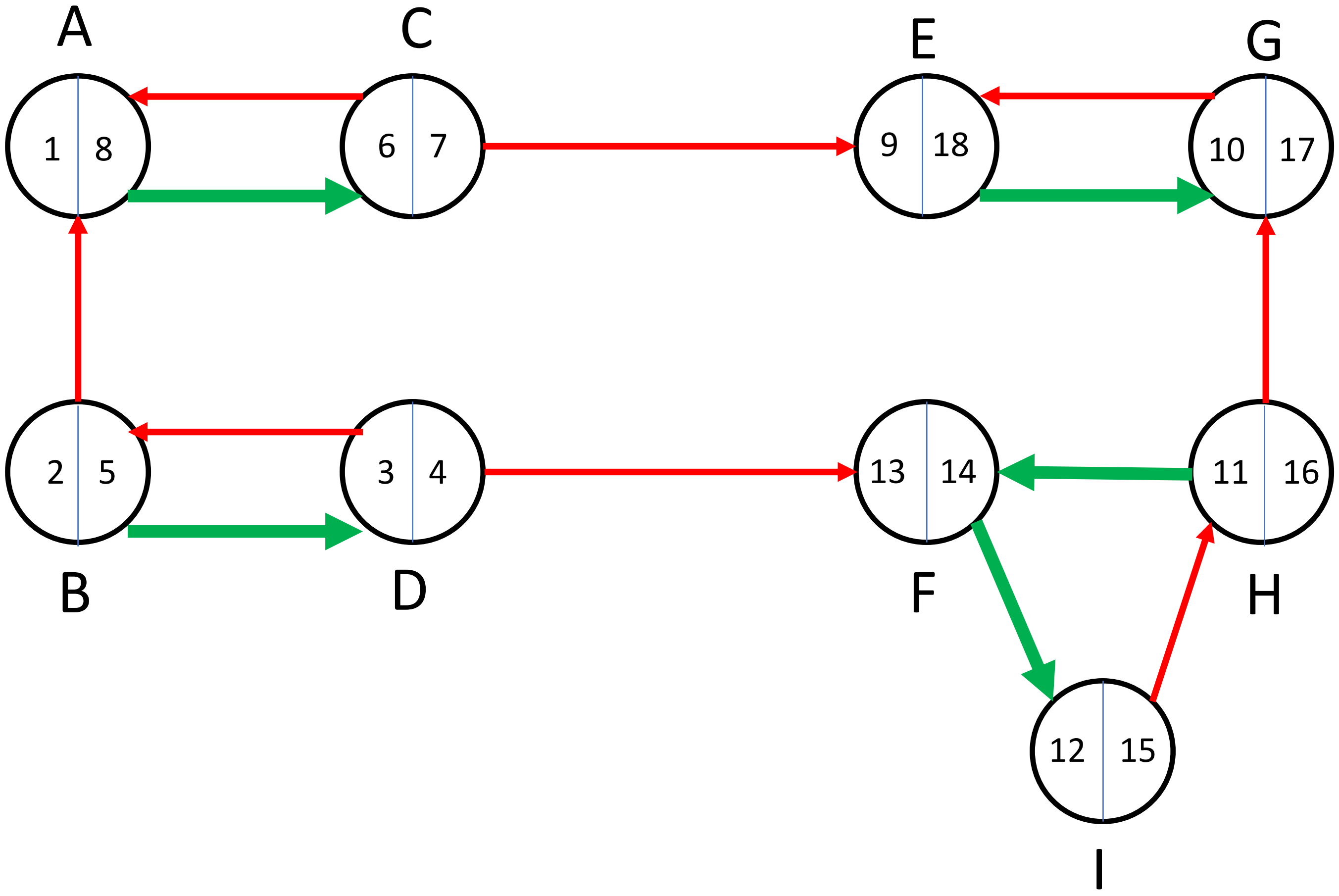
Here, we considered vertices in alphabetical order when staring DFTs.
Also considered each vertex’s neighbors in alphabetical order.

Algo: 2. Reverse Edge Directions

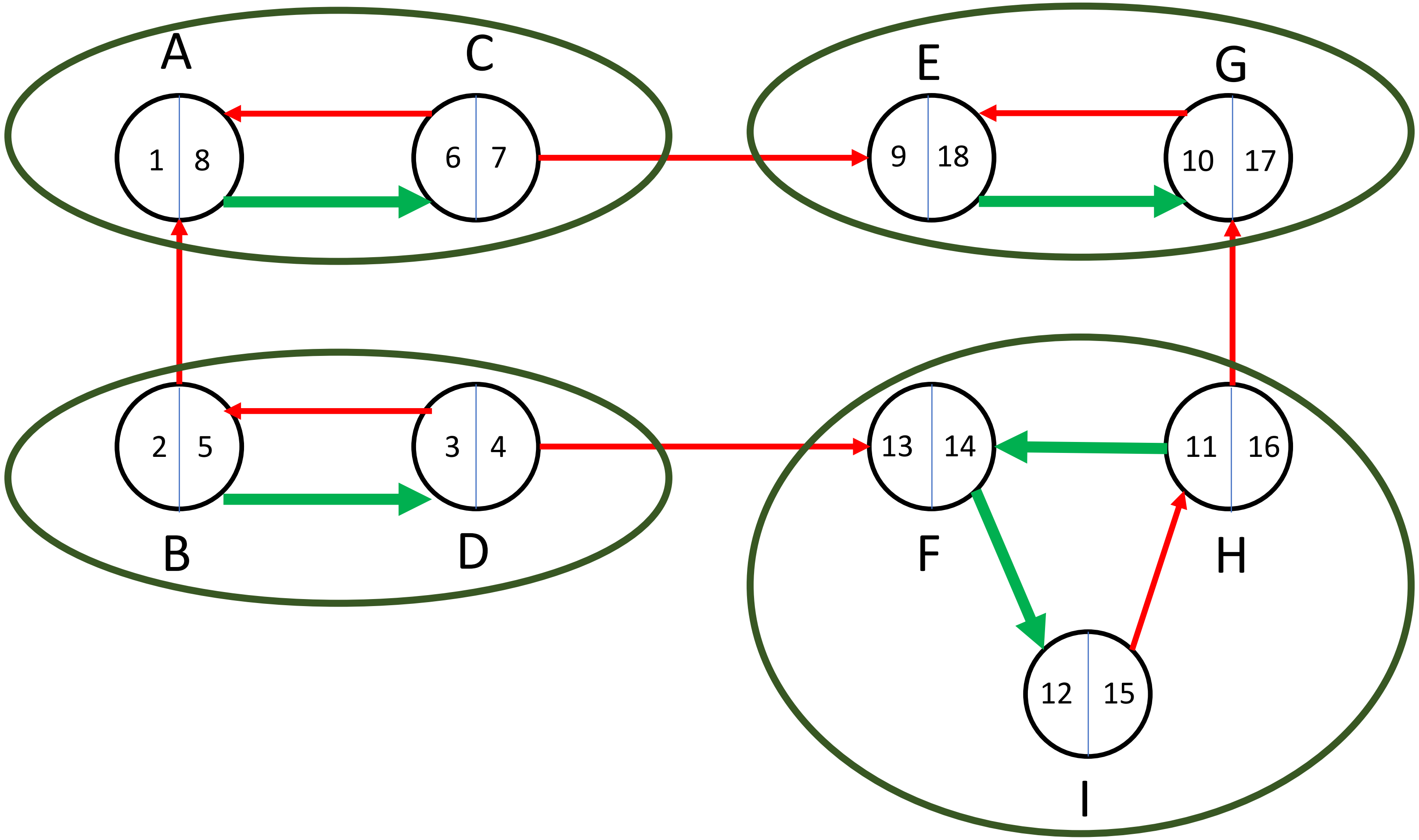


Blue: original directions
Red: reverse directions

Algo: 3. Run DFS considering vertices in decreasing order of their finish time to start new DFTs



Algo: 4. Output the vertices in each DFT as a SCC



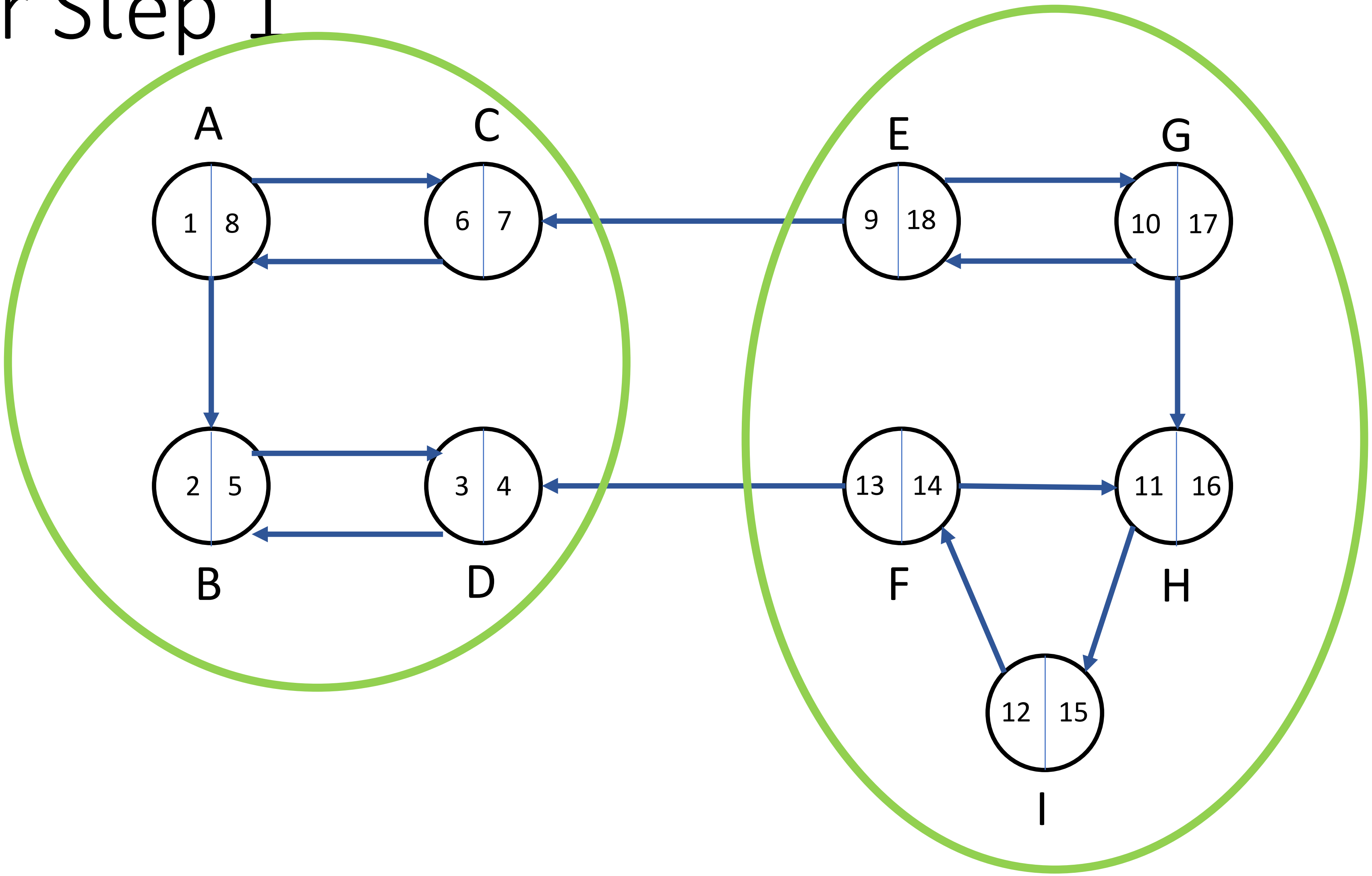
{A, C}, {E, G}, {B, D}, {F, I, H}

But why does the algorithm work?

- Let’s first see the DFT including the vertex with the max finish time forms a SCC.
- We will then repeat this argument.

Let’s first see the DFT including the vertex forms a SCC.

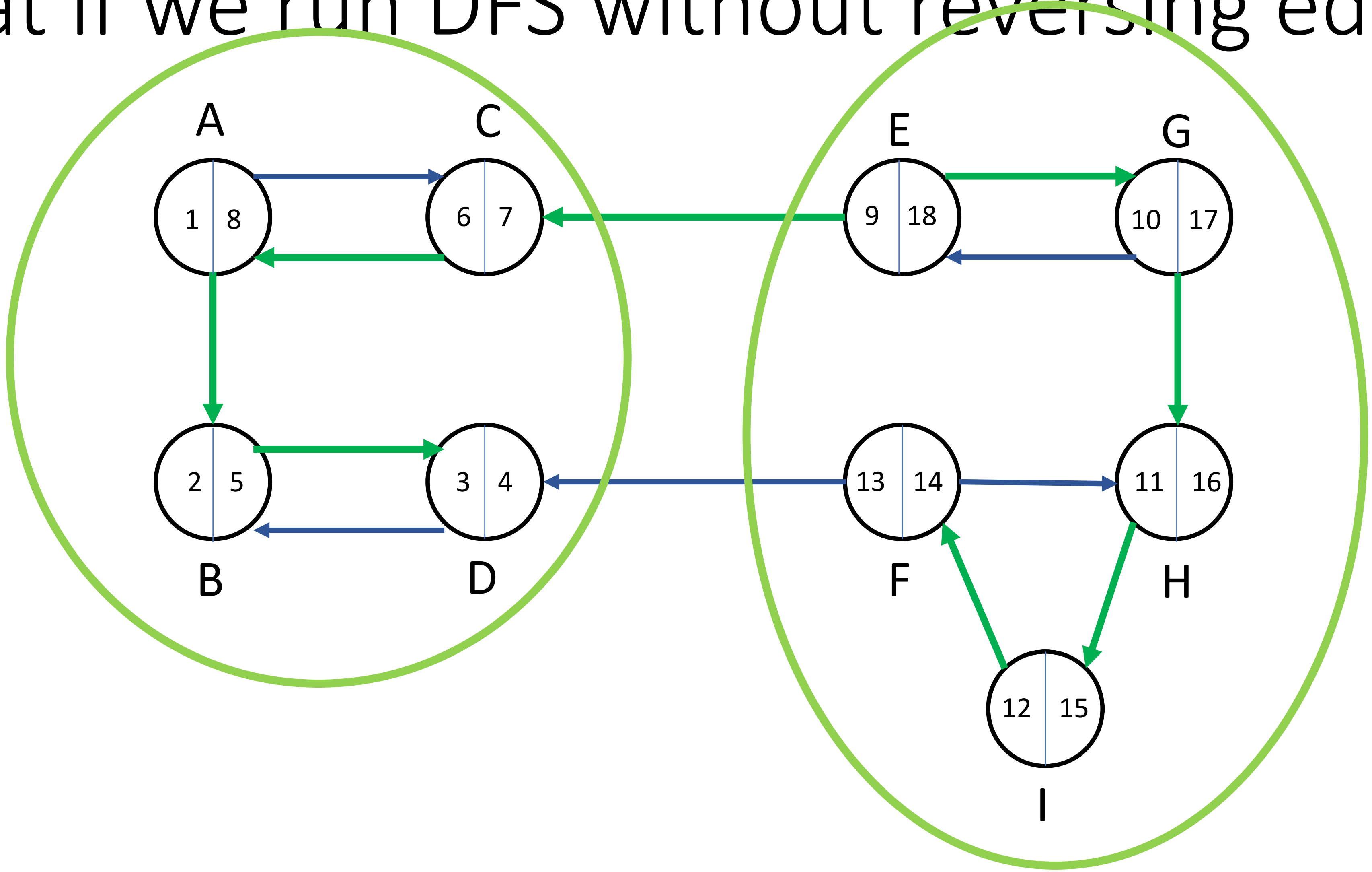
After Step 1



2 DFTs: We need to further refine!

Let’s first see the DFT including the vertex forms a SCC.

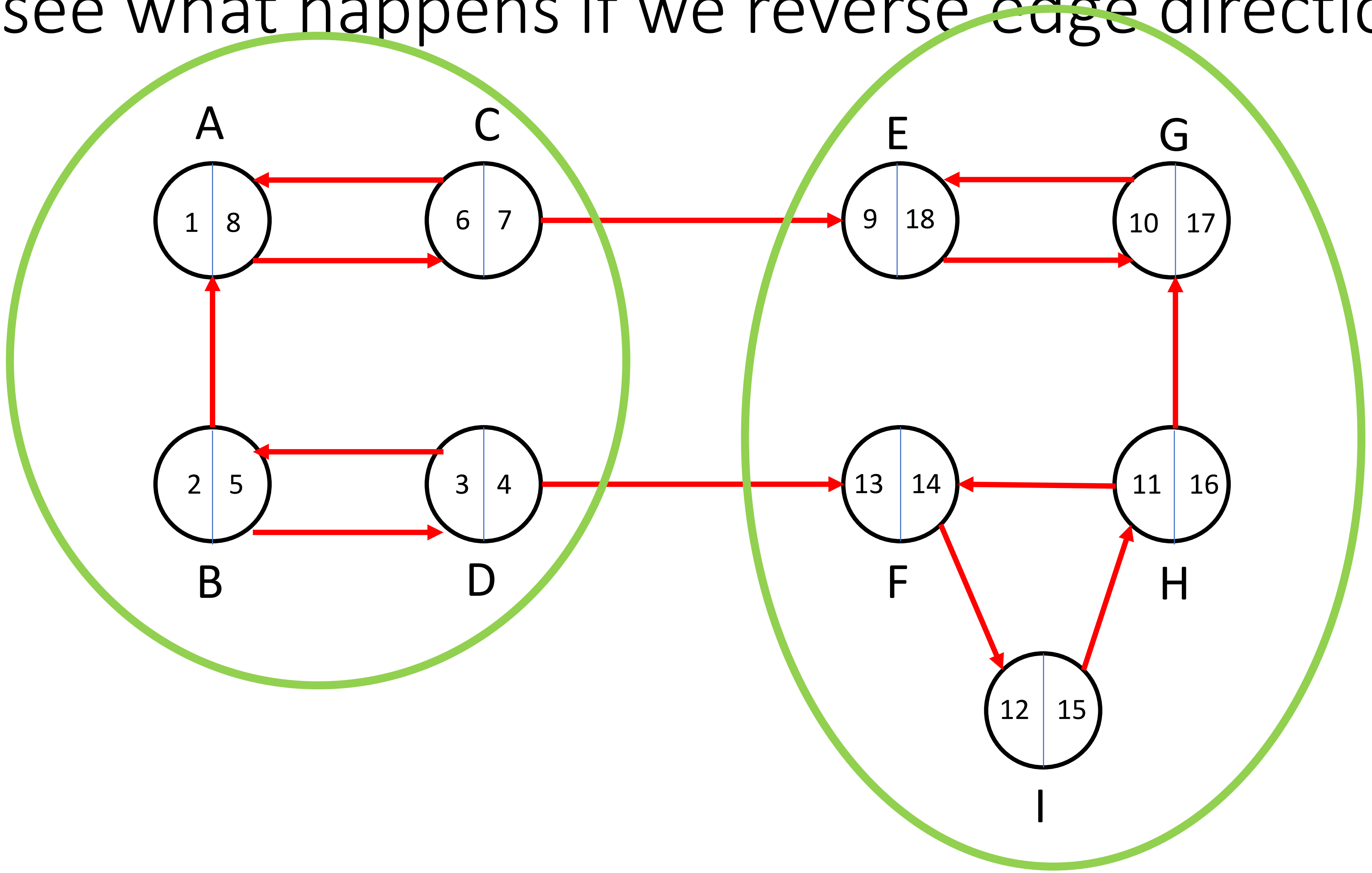
What if we run DFS without reversing edges?



Well, we get only one DFT at the end.. So, worse!

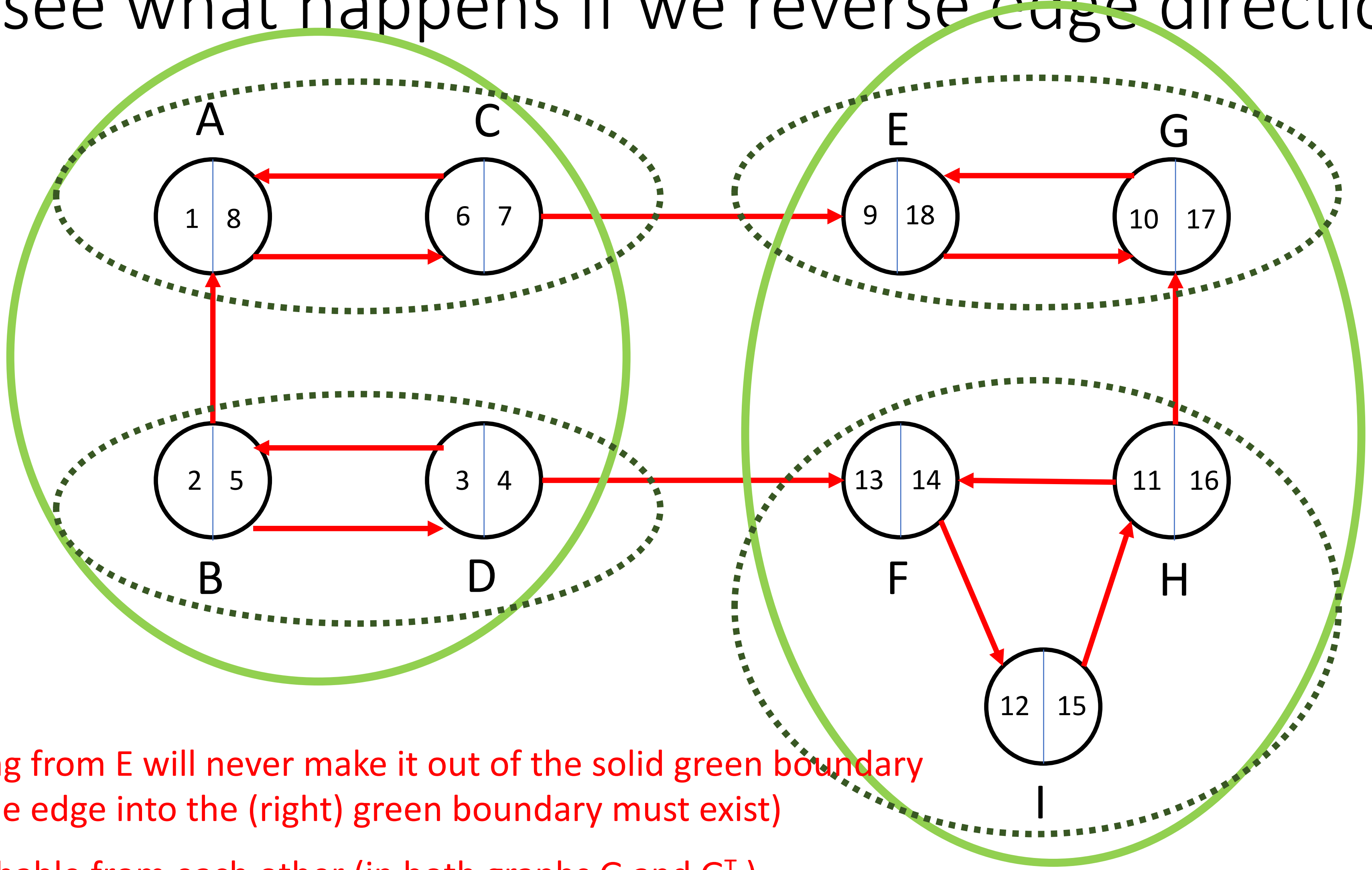
Let's first see the DFT including the vertex forms a SCC.

Let's see what happens if we reverse edge directions



Let's first see the DFT including the vertex forms a SCC.

Let's see what happens if we reverse edge directions



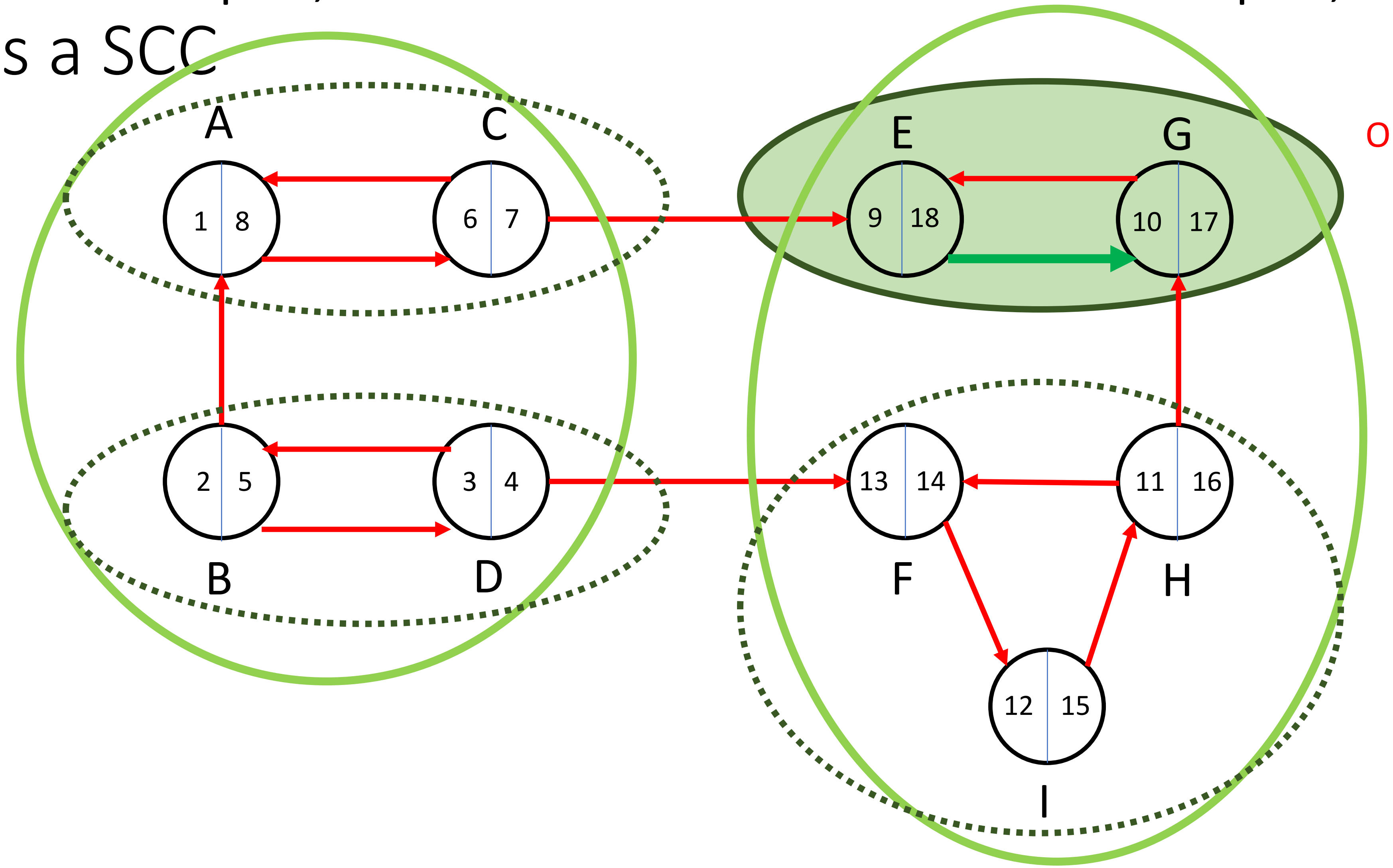
Then, DFS starting from E will never make it out of the solid green boundary
(otherwise, a blue edge into the (right) green boundary must exist)

E and G still reachable from each other (in both graphs G and G^T).

F, H, I reachable from E in G, but not in G^T (otherwise, E, F, I, H must be in the same SCC)

Let’s first see the DFT including the vertex forms a SCC.

So, after step 2, we obtain the first DFT in Step 3, which forms a SCC

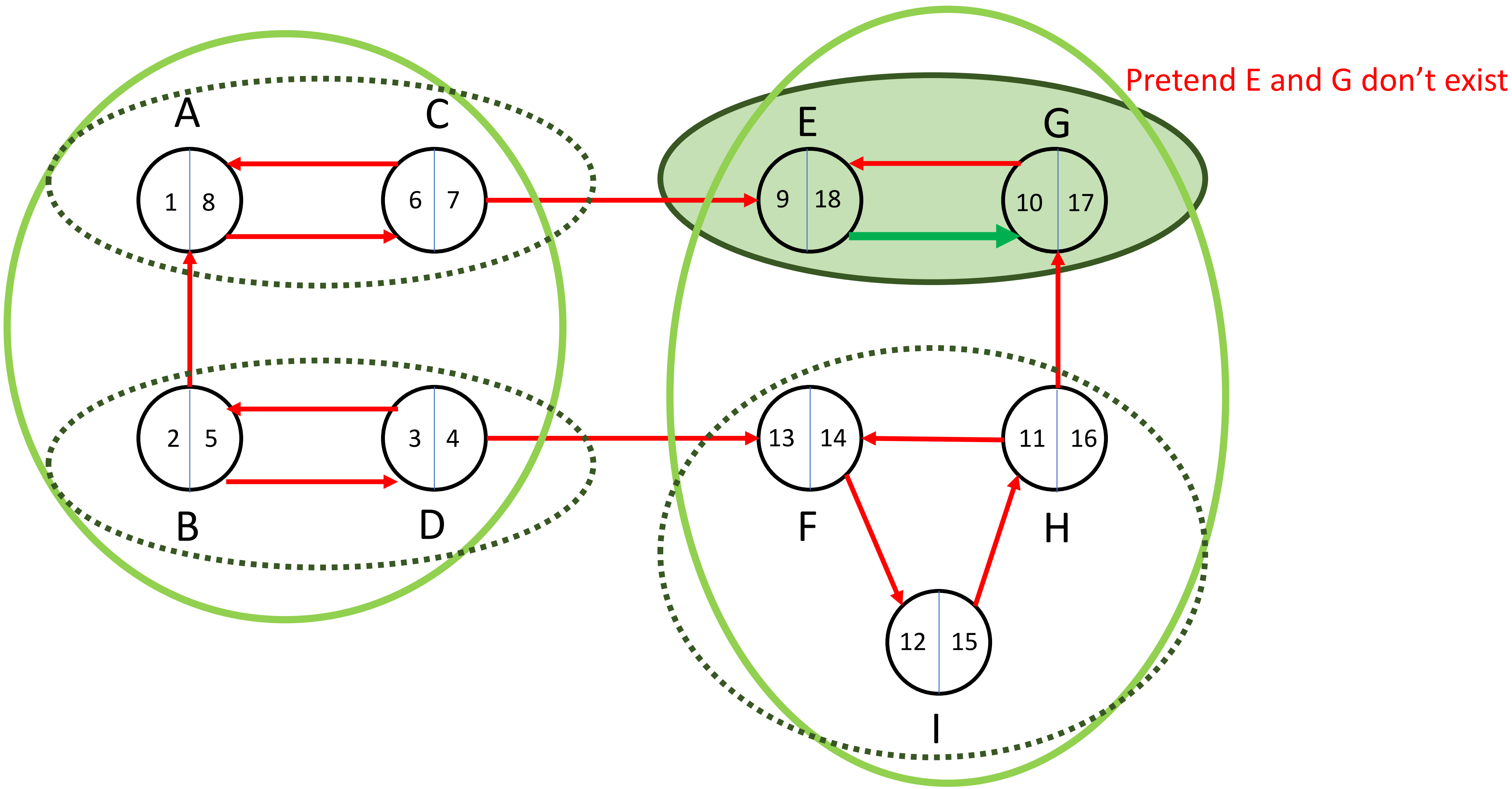


Obtained a SCC, {E, G}

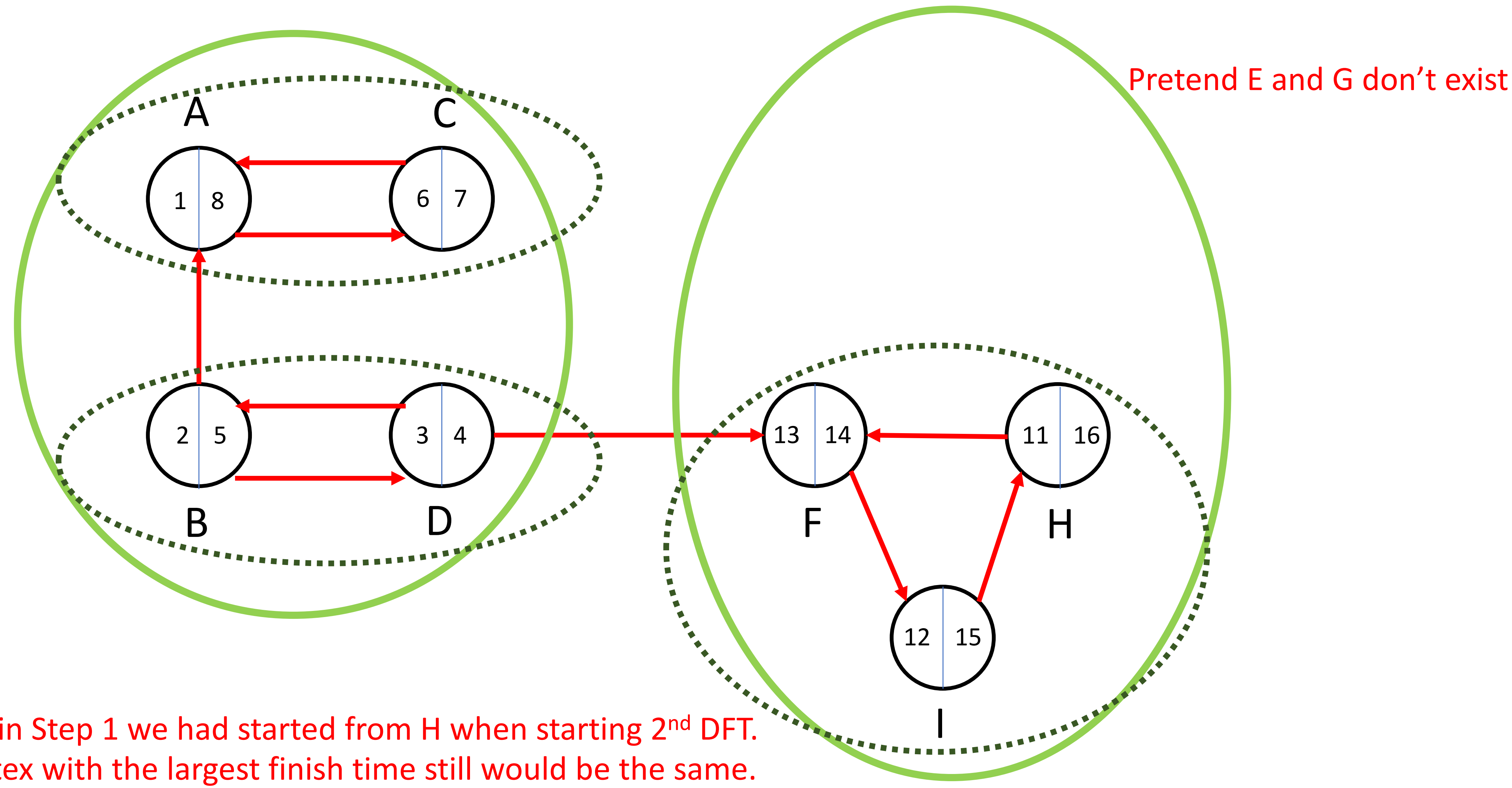
But why does the algorithm work?

- Let’s first see the DFT including the vertex with the max finish time forms a SCC.
- We will then repeat this argument.

We will then repeat this argument.



We will then repeat this argument.



Suppose that in Step 1 we had started from H when starting 2nd DFT.
Then, the vertex with the largest finish time still would be the same.
By repeating the previous argument, we would get a SCC {F, H, I}.
We repeat the same argument until we get all SCCs.