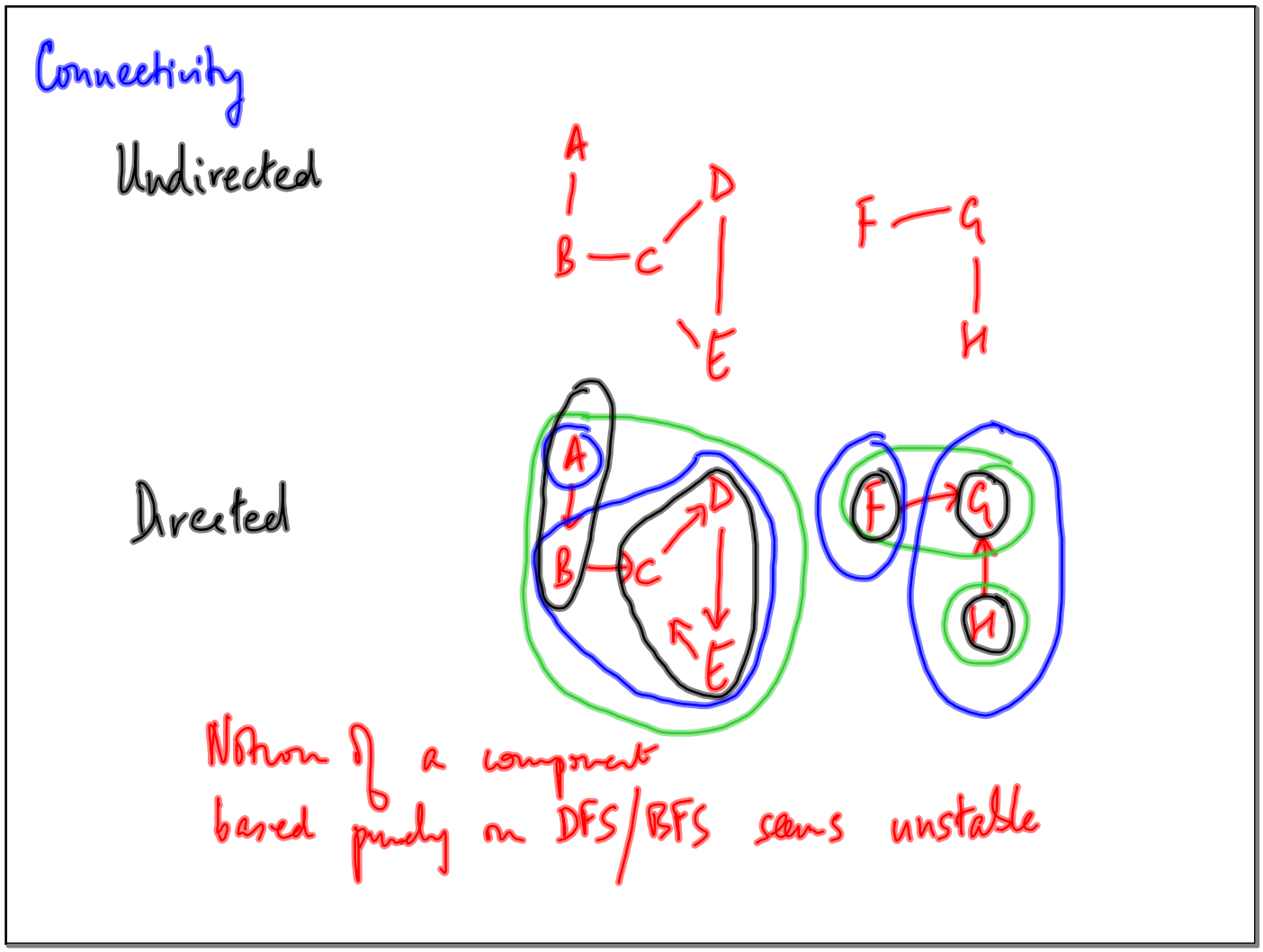
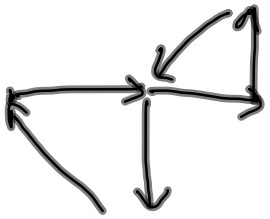

Connectivity
Undirected
A
B
C
D
E
F
G
H
Directed
Notion of a component
based purely on DFS/BFS seems unstable

When is DFS/BFS guaranteed to identify "same" component for a set of vertices

e.g. {C, D, E} is always in same component in previous example

but {C, B}, {A, B}, {F, G} -- are not



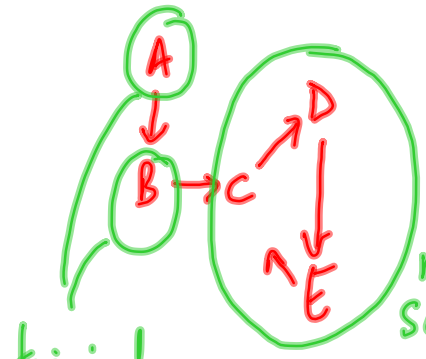
Symmetry: If $u \rightsquigarrow v$ is a path, so is $v \rightsquigarrow u$

"Strongly connected component"

$X \subseteq V$ is an s.c.c. if $\forall x, y \in X \quad x \rightsquigarrow y \;\&\; y \rightsquigarrow x$ ($x \neq y \Rightarrow$)

Typically we are interested in maximal scc's

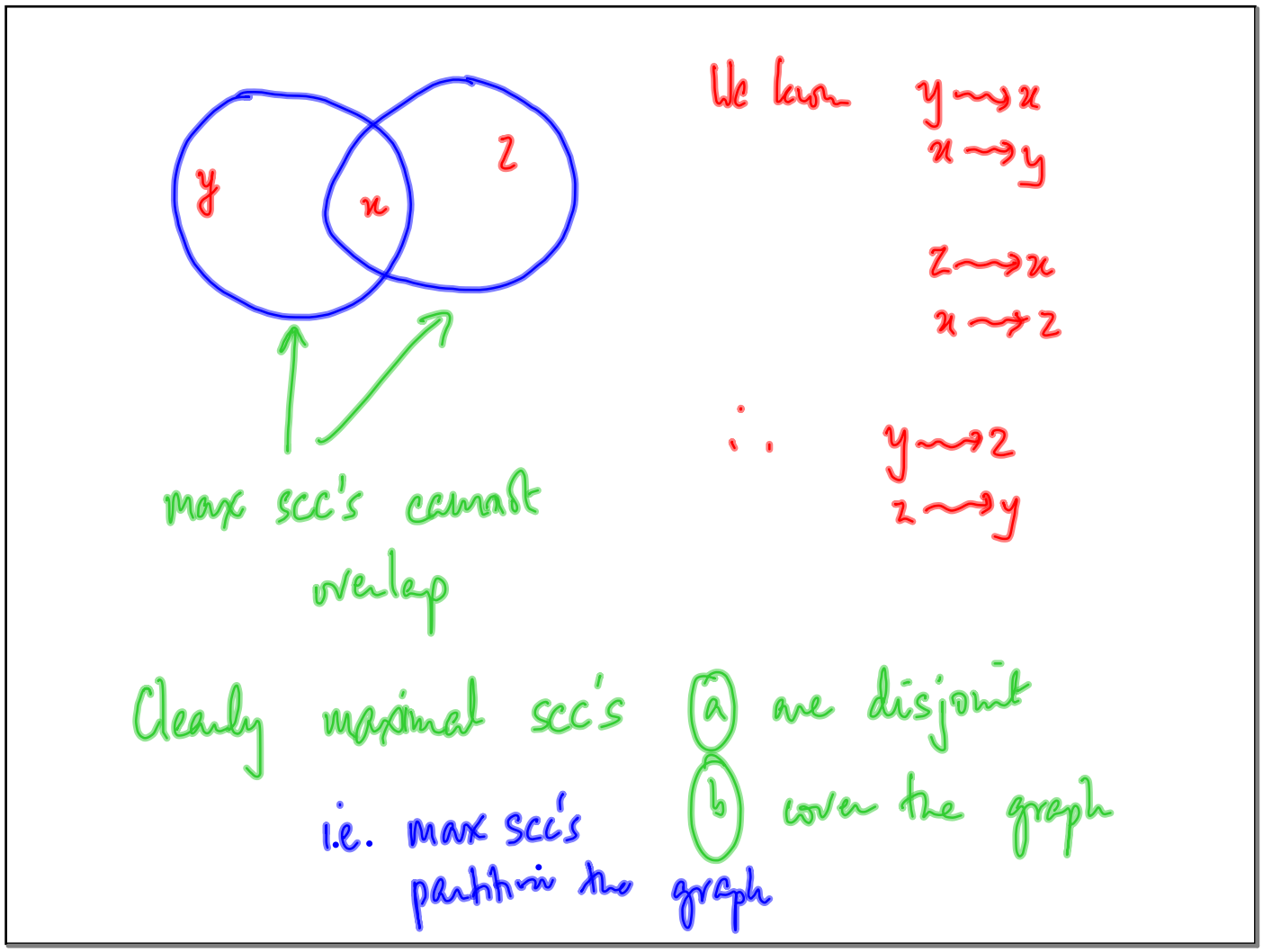
max.



maximal scc

trivial singleton scc's

Can scc's overlap?


y
x
z
We know
y ⇝ x
x ⇝ y
z ⇝ x
x ⇝ z
∴
y ⇝ z
z ⇝ y
max scc's cannot overlap
Clearly maximal scc's
i.e. max scc's partition the graph
(a) are disjoint
(b) cover the graph

How to identify & label the scc's in a graph?

We know:

scc's partition the graph

$G = (V, E)$ $\quad V = C_1 \dot\cup C_2 \dot\cup \ldots \dot\cup C_k$

each C_i is a (max) scc

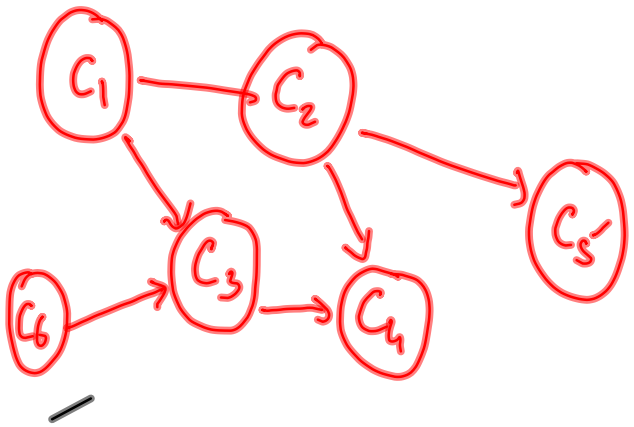
Construct SCC graph

Vertices = $\{C_1, C_2, \ldots, C_k\}$

Edges: $C_i \to C_j$ if $\exists v_i \in C_i, v_j \in C_j$ $v_i \to v_j$

Claim: The SCC graph is a dag

If $C_i \to C_j$ & $C_j \to C_i$ then $C_i \cup C_j$ is also an scc $\Rightarrow$ C_i, C_j are not maximal!



If we start DFS/BFS in C_5 or C_4, the component would be an SCC

Trick:

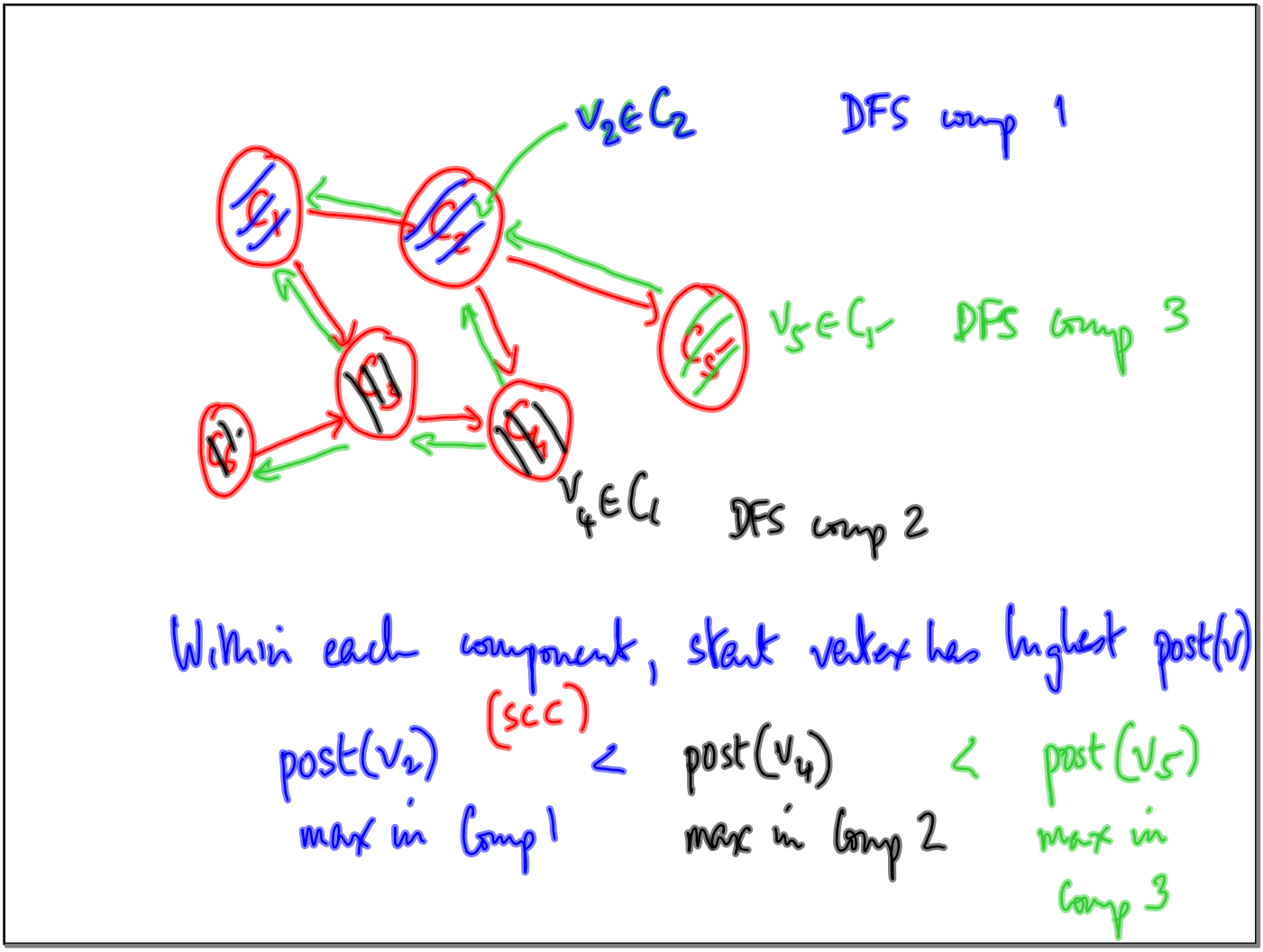
Reverse all edges in original G

Same SCC's

Dag structure of SCC graph is reversed

Run DFS on $G^{reverse}$

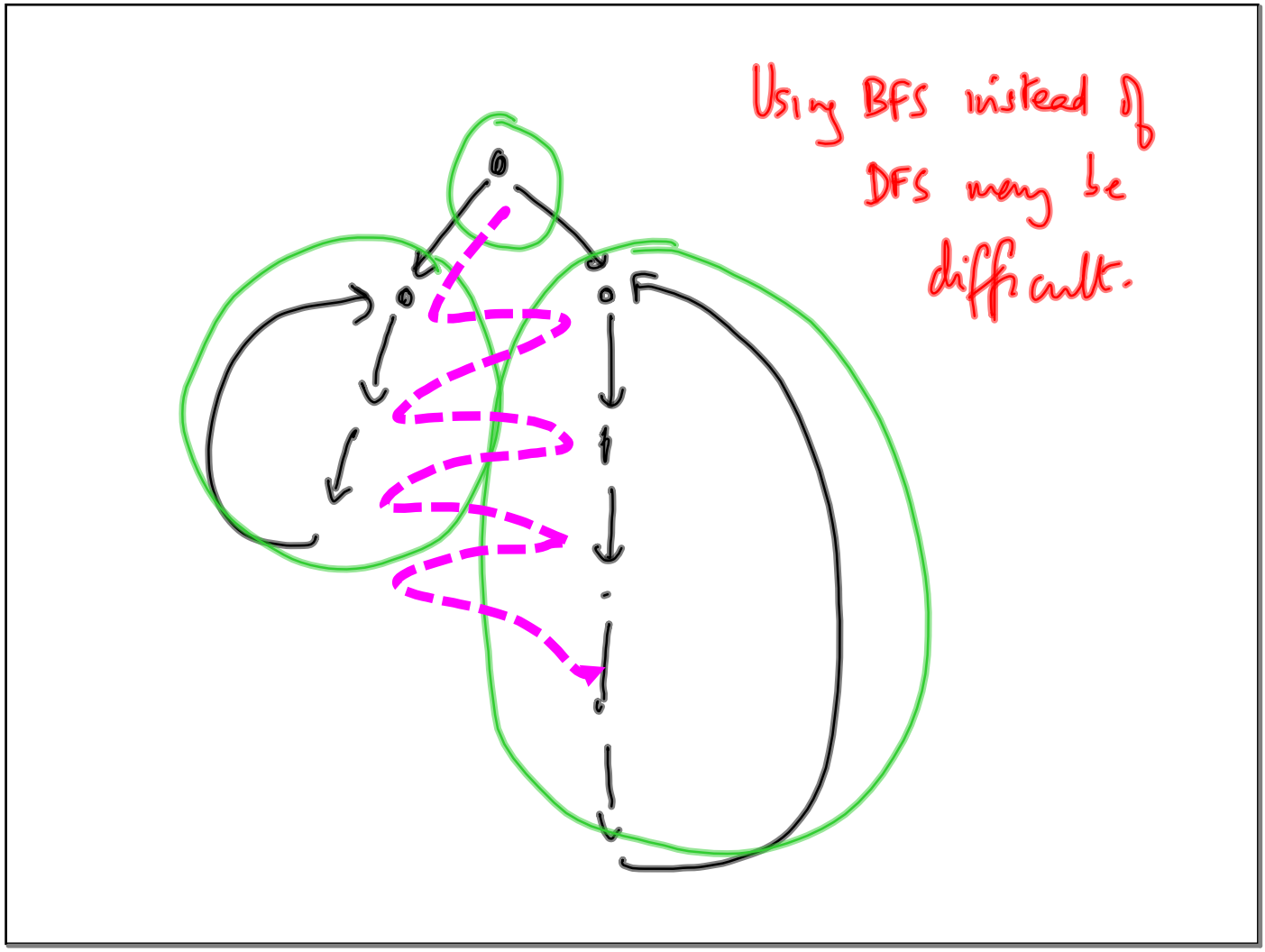
Associates post(v) to each v wrt. $G^{reverse}$


$v_2 \in C_2$
DFS comp 1
$v_5 \in C_1$
DFS comp 3
$v_4 \in C_1$
DFS comp 2
Within each component, start vertex has highest post(v)
(scc)
post(v_2) < post(v_4) < post(v_5)
max in Comp 1
max in Comp 2
max in Comp 3

Go back to original G

Run DFS in descending order of post(v)
as computed by DFS on G^{rev}

Each component identified by this DFS
will correspond exactly to an SCC


Using BFS instead of
DFS may be
difficult.

Back to topological sort

Given a DAG $G = (V, E)$

Scan G & compute indegree(v) for each v — $O(n+m)$ with adj list

"Preprocessing"

$X = V$

while $X \neq \phi$

print any $x \in X$ of indegree 0 — takes $O(n)$ time

remove x from X (& outgoing edges)

↑ updates only outdegree(x)

Maintain and update a list of vertices of indegree 0

More conventional to maintain a queue

If DAG is not empty, queue of indegree 0 vertices is not empty

Modified topological sort

```
Q = ∅
for each v ∈ V, compute indegree(v)
for each v ∈ V, if indegree(v) = 0, add v to Q

while Q is not empty
    extract head of Q as q & print it
    reduce indegree(v) for all v s.t. (q, v) ∈ E
     if indegree(v) = 0, add v to Q
```