Lecture 10

Finding strongly connected components

Animations have been removed to
make the pdf more printer-friendly.
See .pptx for multi-slide animations.

Announcements

- HW4 due Friday
- Nothing assigned Friday because…
- **MIDTERM** in class, Monday 10/30.
  - **Please show up.**
  - During class, 1:30-2:50
    - If your last name is A-M: 370-370 (here)
    - If your last name is N-V: 160-124
    - If your last name is W-Z: 160-323
  - You may bring one double-sided letter-size page of notes, that ***you have prepared yourself***.

  - Any material through Hashing (Lecture 8) is fair game.
  - Practice exams on the website

More midterm info

- There will be four sections:
  - 1. Multiple choice
    - Tests basic knowledge
  - 2. Short answer
    - Tests your ability to apply basic knowledge
  - 3. Algorithm Design
    - Similar to a alg. design HW problem (a bit easier)
  - 4. Proving Stuff
    - Similar to a proving-stuff HW problem (a bit easier)

- This may be a hard exam
  - **If it is, that means it's okay if you don't get all the questions.**
  - (Please don't freak out).

Last time

- Breadth-first and depth-first search
- Plus, applications!
  - Topological sorting
  - In-order traversal of BSTs
  - Shortest path in unweighted graphs
  - Testing bipartite-ness

- The key was paying attention to the structure of the tree that these search algorithms implicitly build.

Today

- One more application:
  - Finding

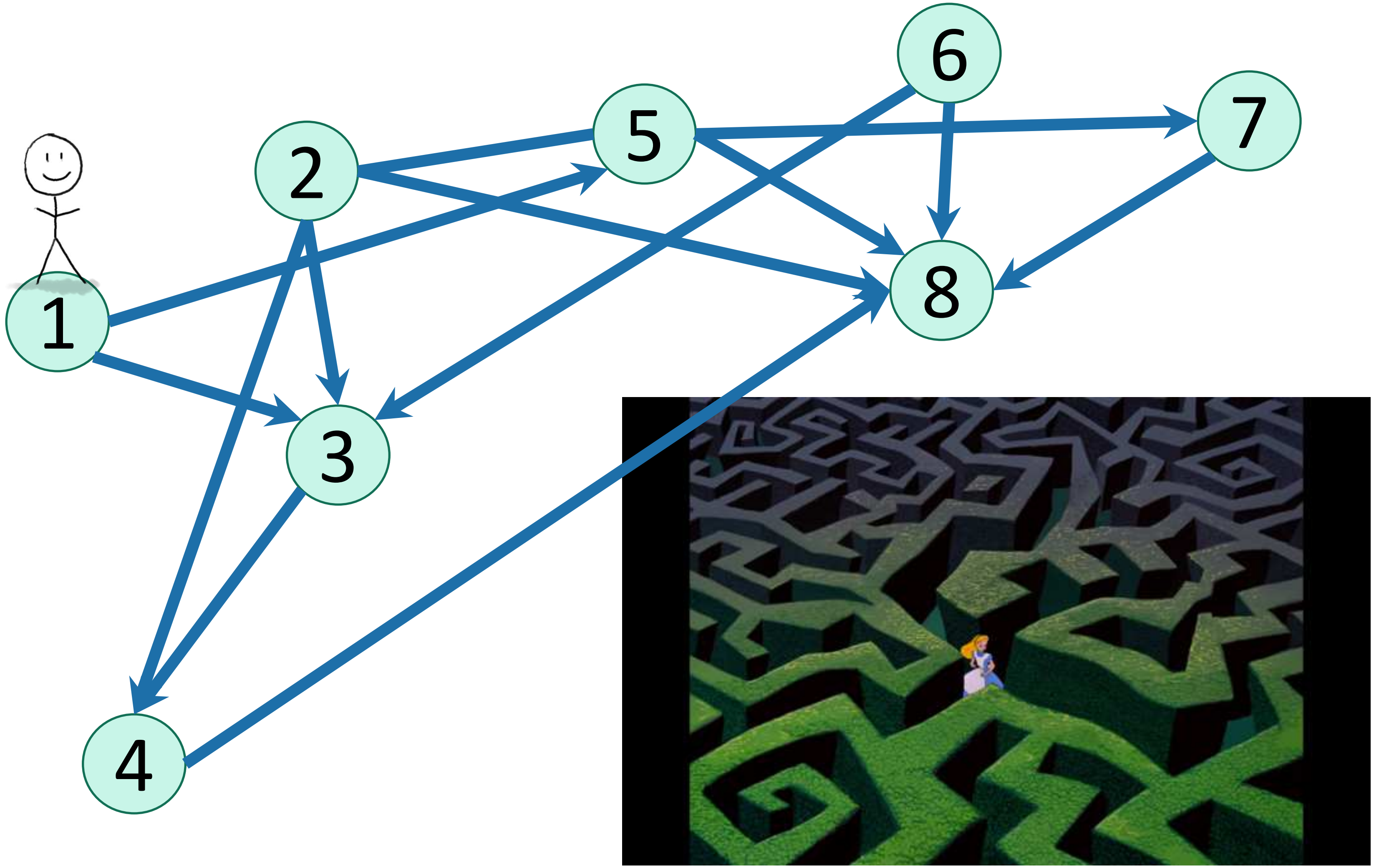
    strongly connected components

- But first! Let's briefly recap DFS…

Recall: DFS

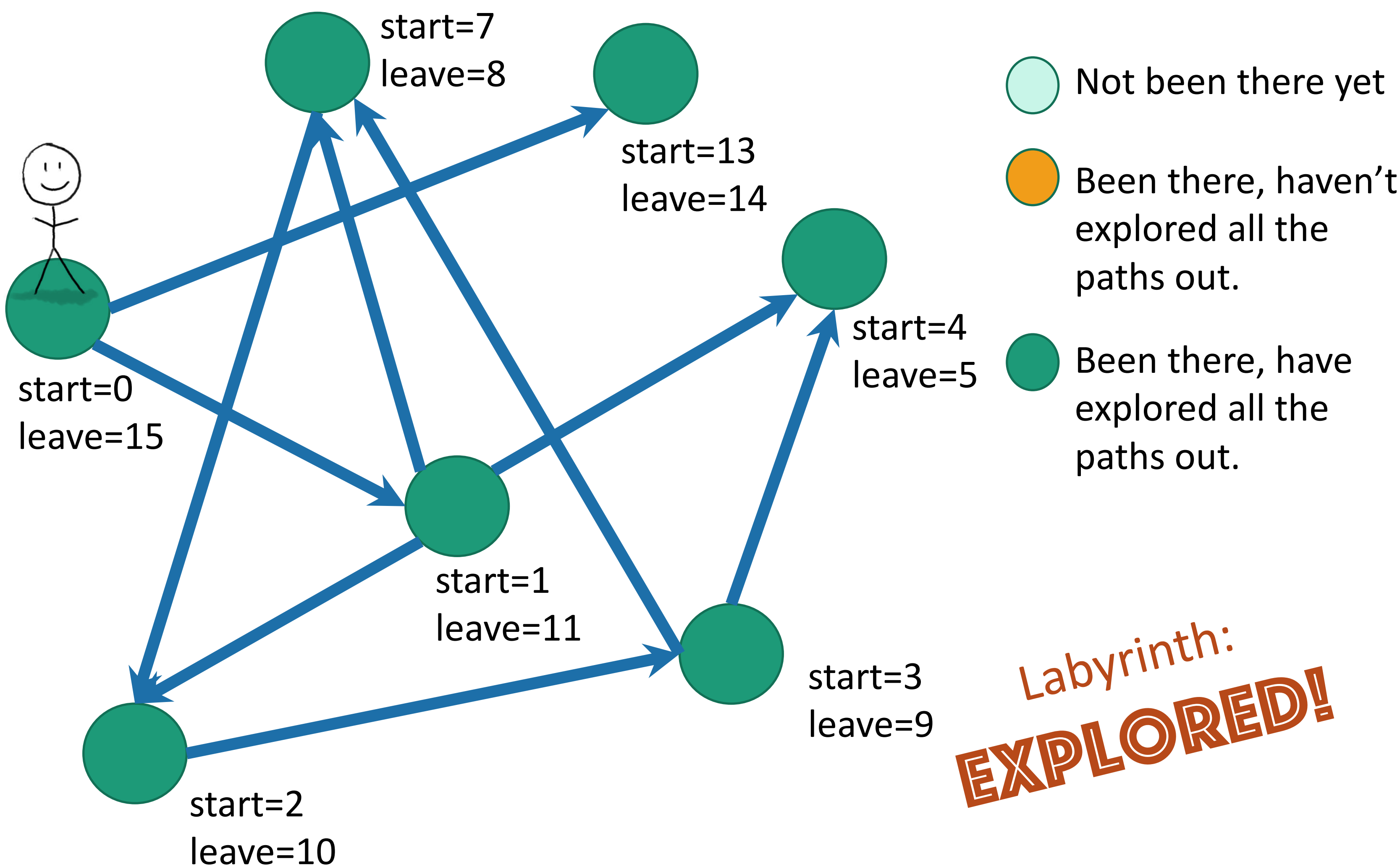
Today, all graphs are **directed**!
Check that the things we did on Monday still all work!

It's how you'd explore a labyrinth with chalk and a piece of string.



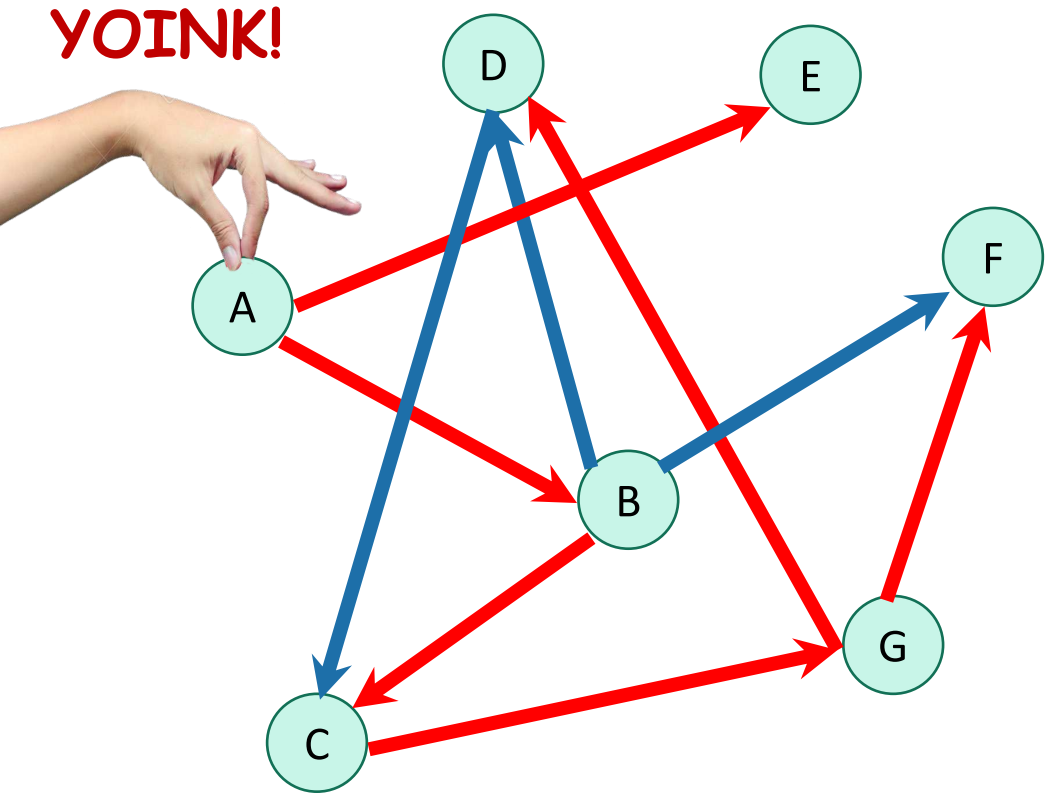
Depth First Search

Exploring a labyrinth with chalk and a piece of string

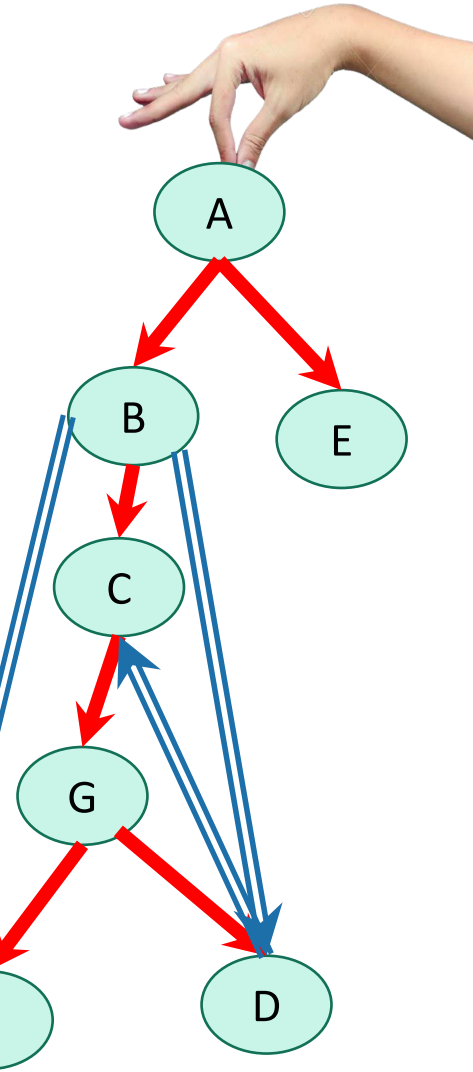


Depth first search

implicitly creates a tree on everything you can reach

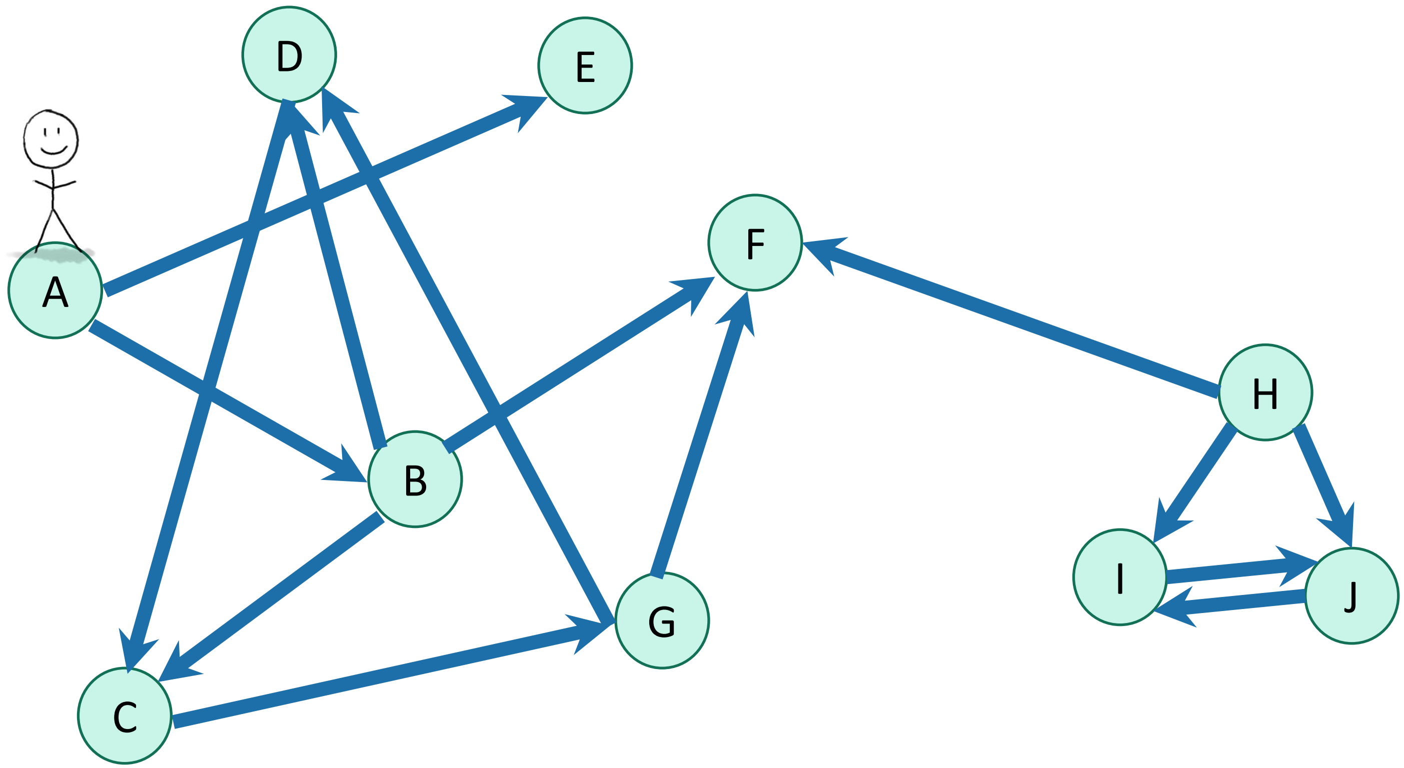


Call this the
"DFS tree"



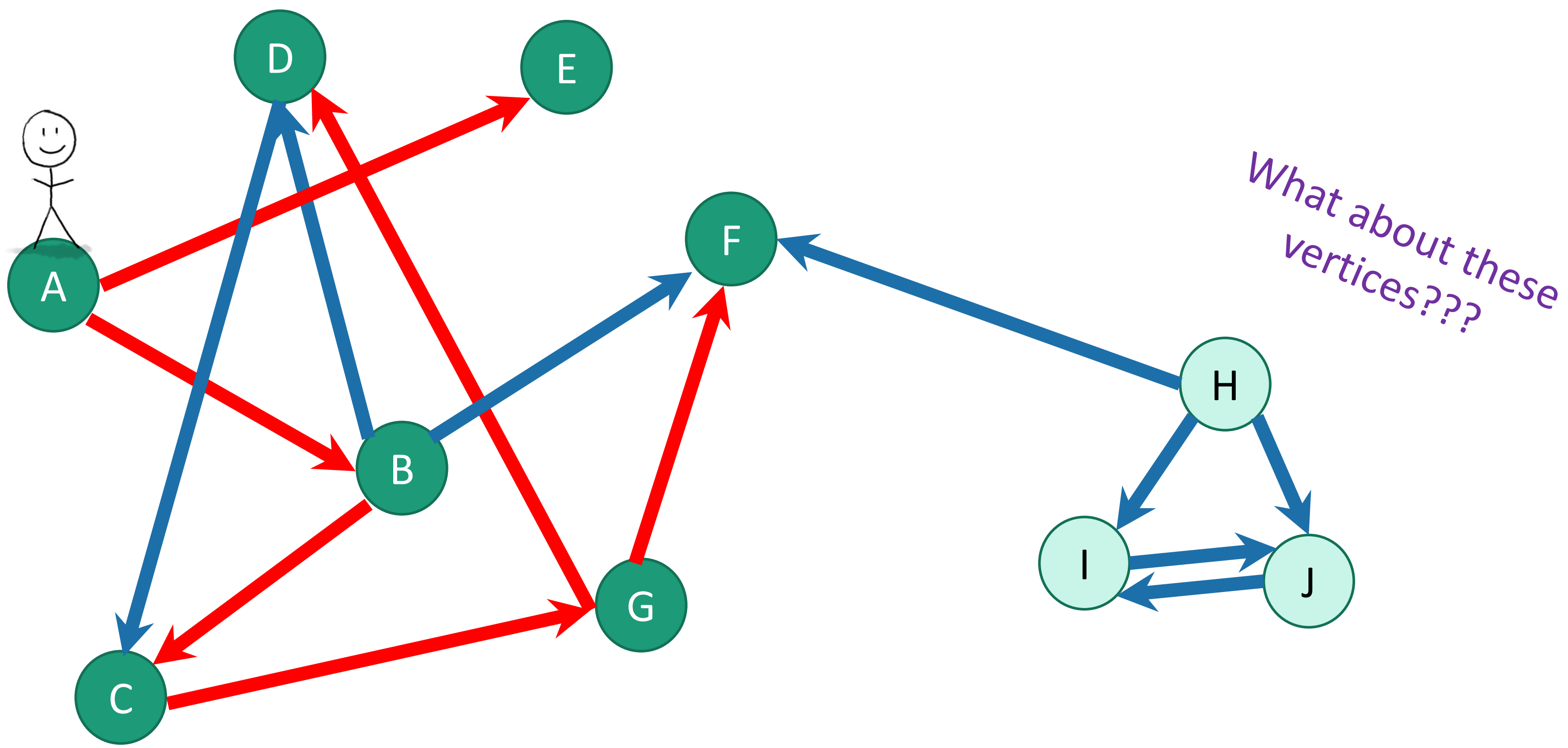
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



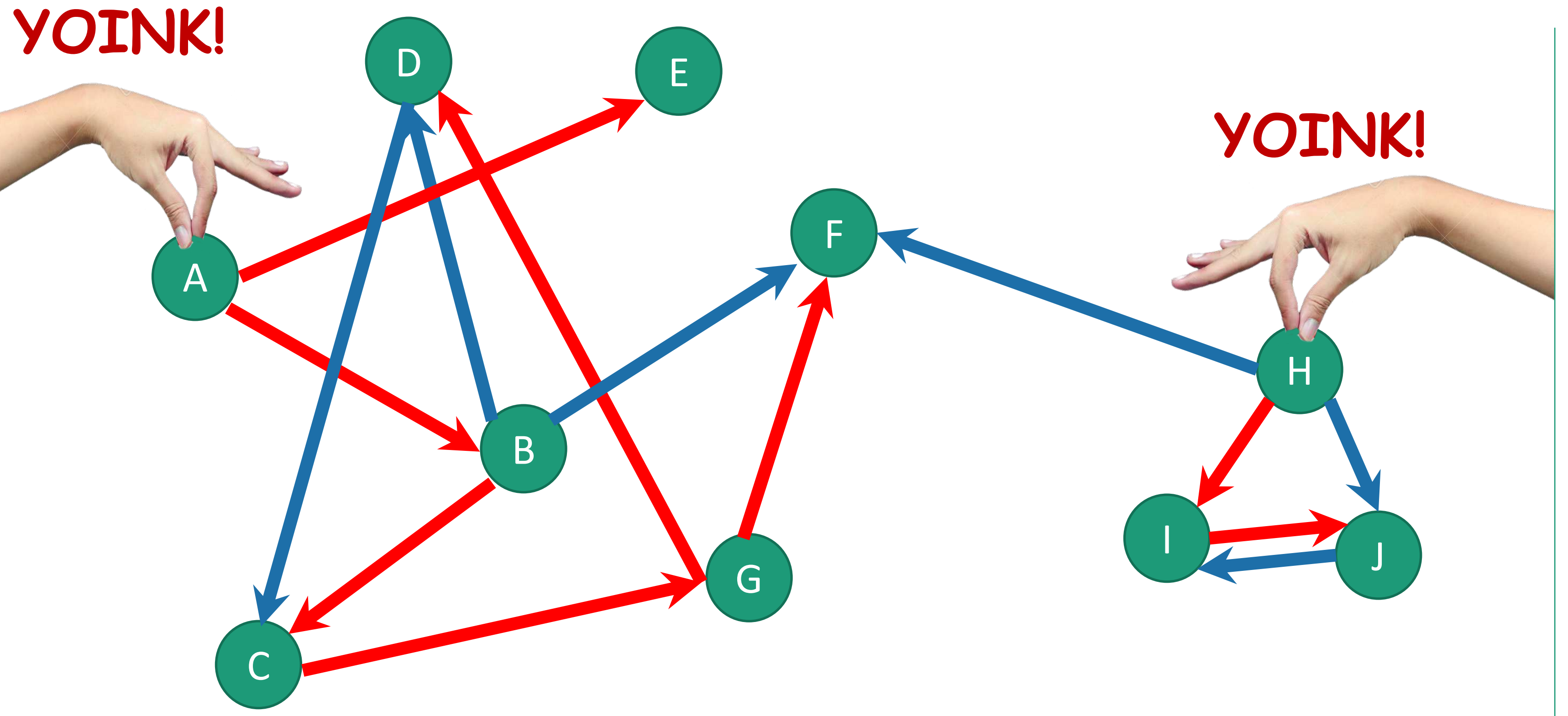
When you can’t reach everything

- Run DFS repeatedly to get a depth-first forest



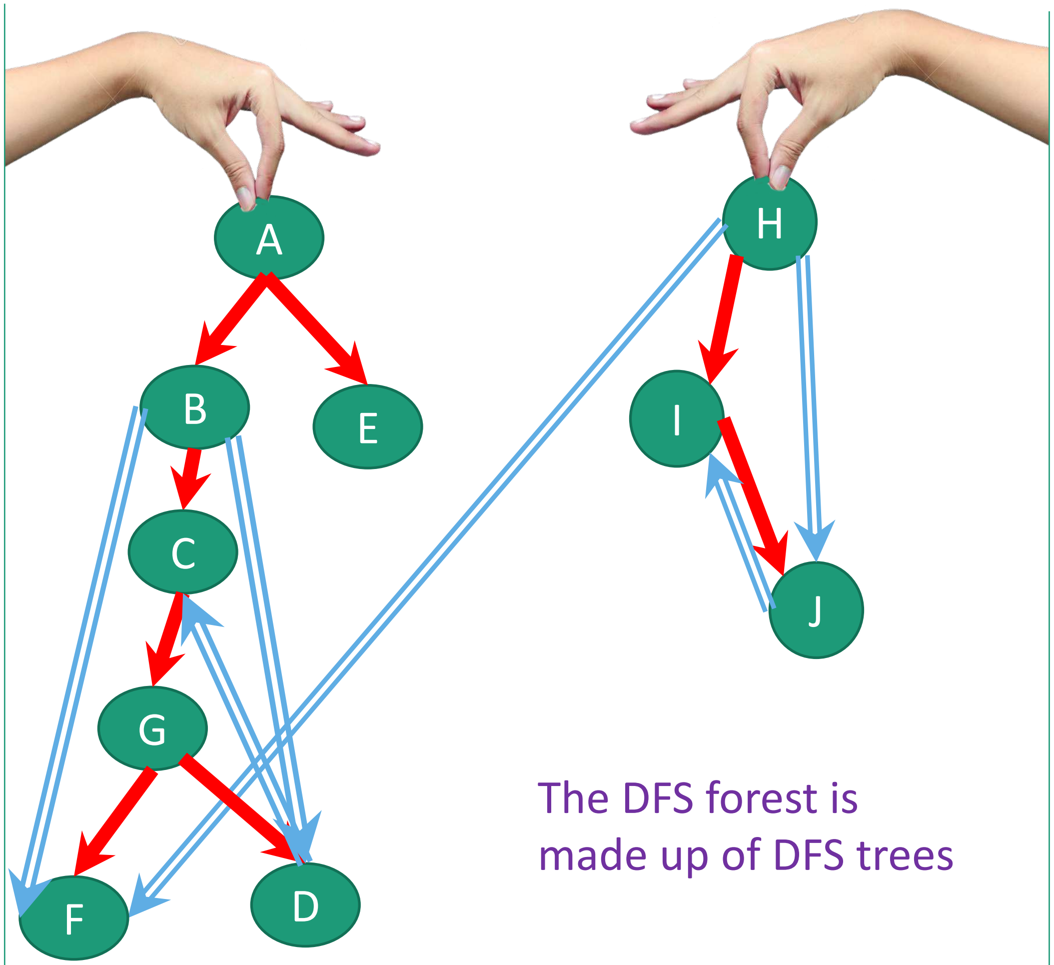
When you can’t reach everything

- Run DFS repeatedly to get a depth-first forest



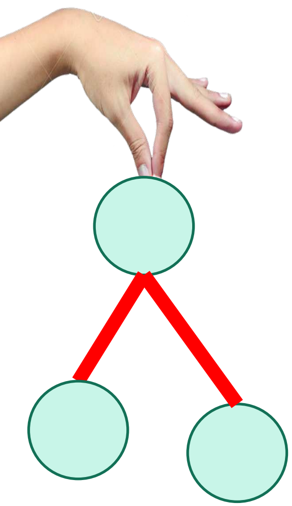
When you can’t reach everything

- Run DFS repeatedly to get a depth-first forest



Recall: the parentheses theorem

(Works the same with DFS forests)



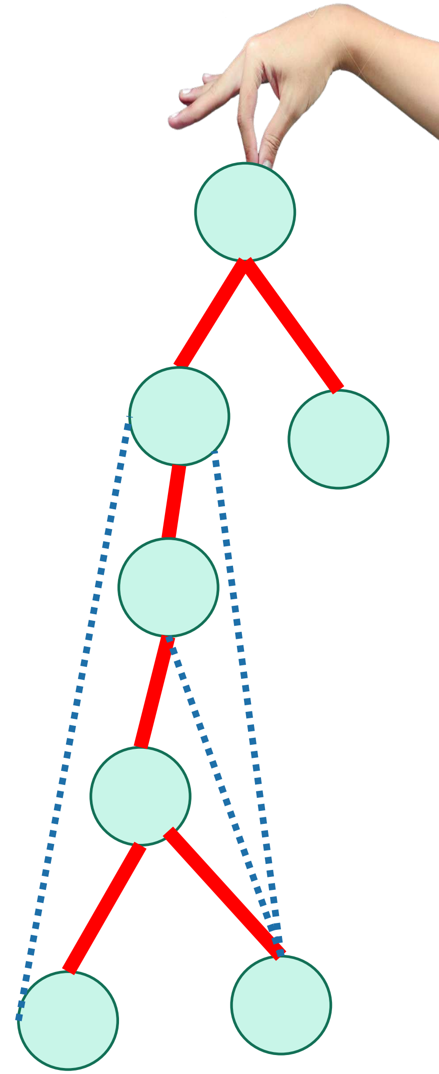
- If v is a descendent of w in this tree:

- If w is a descendent of v in this tree:

- If neither are descendents of each other:

(or the other way around)

If v and w are in different trees, it's always this last one.



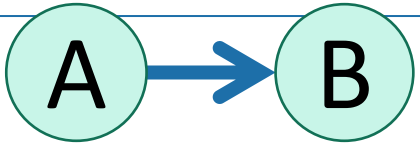
A **great** question from Monday

- Why don't **start times** work for Topological Sorting?

- I mean, demonstrably they don't (we saw some examples) but **what goes wrong** in the proof?

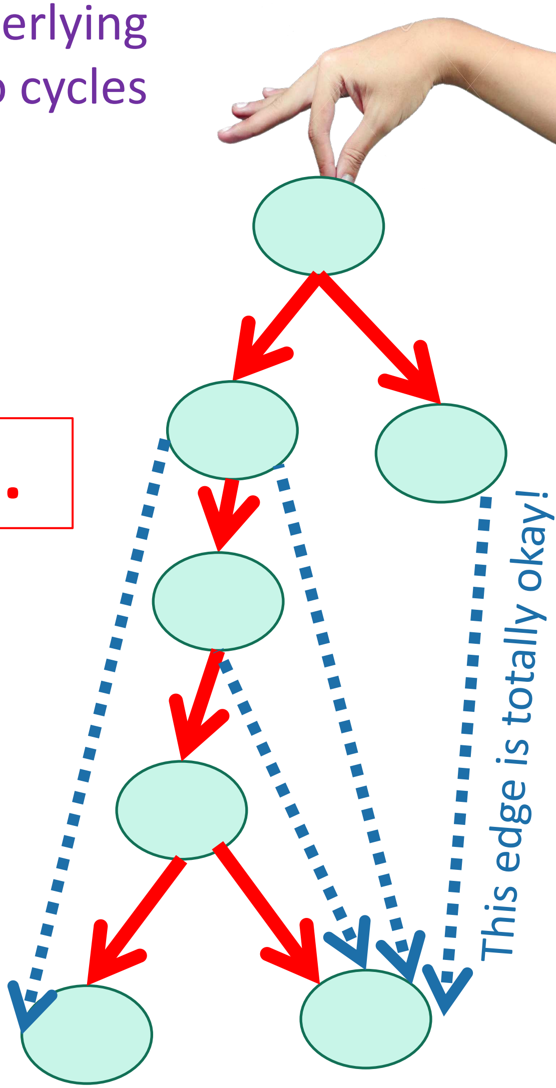
SLIDE FROM LAST TIME

So to prove this ->

If


Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles



- Since the graph has no cycles, B must be a descendant of A in that tree.
  - All edges go down the tree.

This is true.

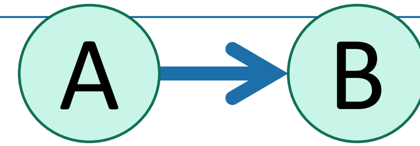
I MESSED UP! This is false.

- Then

- aka, B.finishTime < A.finishTime.

What it should have been

So to prove this ->

If


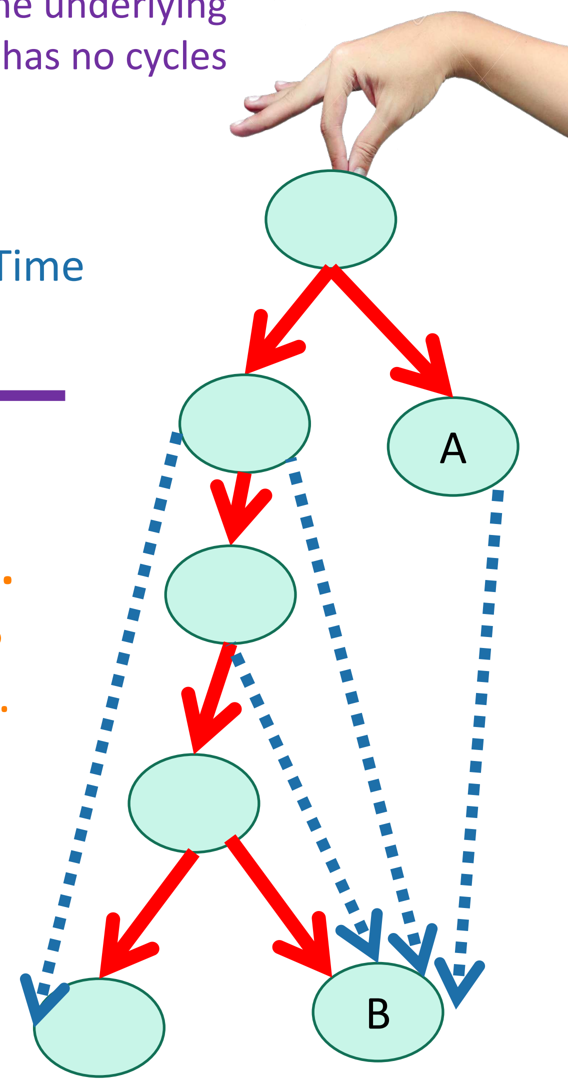
Then B.finishTime < A.finishTime

- If we got to B, then **either**:
  - B is a descendant of A in the DFS tree
  - (Same argument as before)

- **Or!**
  - B is ***not*** a descendant of A in the DFS tree
  - **Then we must have gotten to B before we got to A.** Otherwise we would have explored B from A, and B **would** have been a descendant of A in the DFS tree.

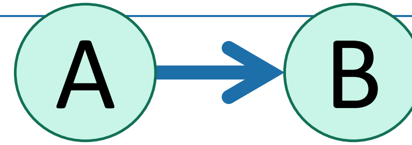
- either way, B.finishTime < A.finishTime.

Suppose the underlying graph has no cycles



What it should have been

So to prove this ->

If 

Then B.finishTime < A.finishTime

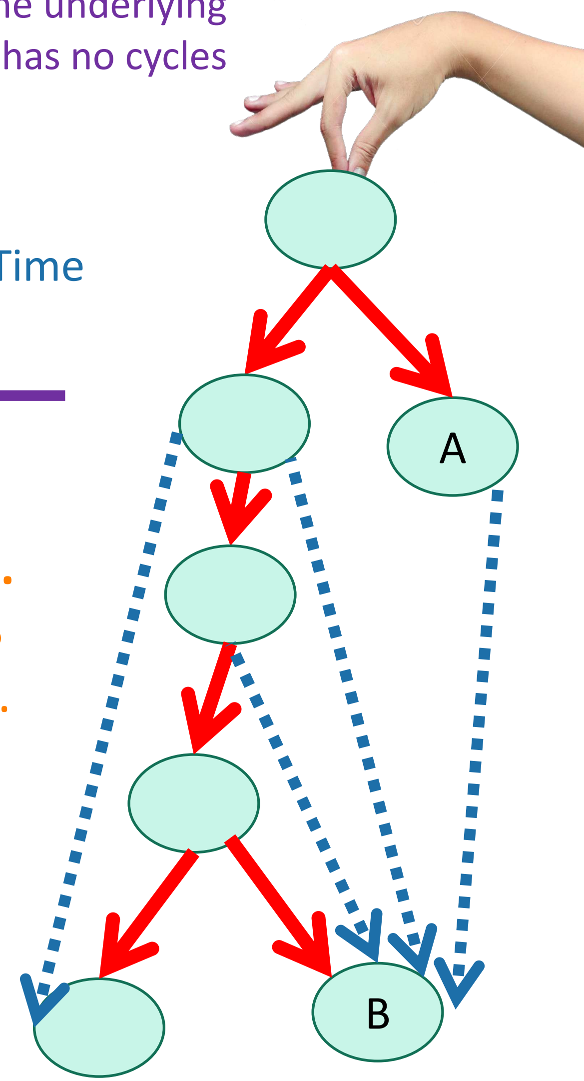
- If we got to B, then **either**:
  - B is a descendant of A in the DFS tree
  - (Same argument as before)

- **Or!**
  - B is ***not*** a d… ant of A in the DFS tree
  - **Th… tten to B before we got to A.**
  - …ve explored B from A, and B …endant of A in the DFS tree.

Note: the start times are in different orders in these two cases! That's why start times don't work.

- either way, B.finishTime < A.finishTime.

Suppose the underlying graph has no cycles

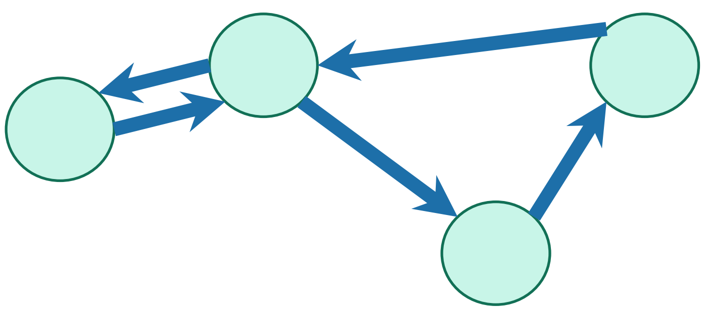


Enough of review (and enough of my shortcomings)

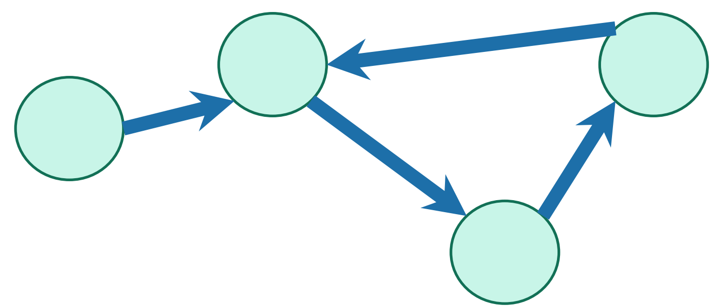
Strongly connected components

Strongly connected components

- A directed graph G = (V,E) is **strongly connected** if:
- for all v,w in V:
  - there is a path from v to w and
  - there is a path from w to v.



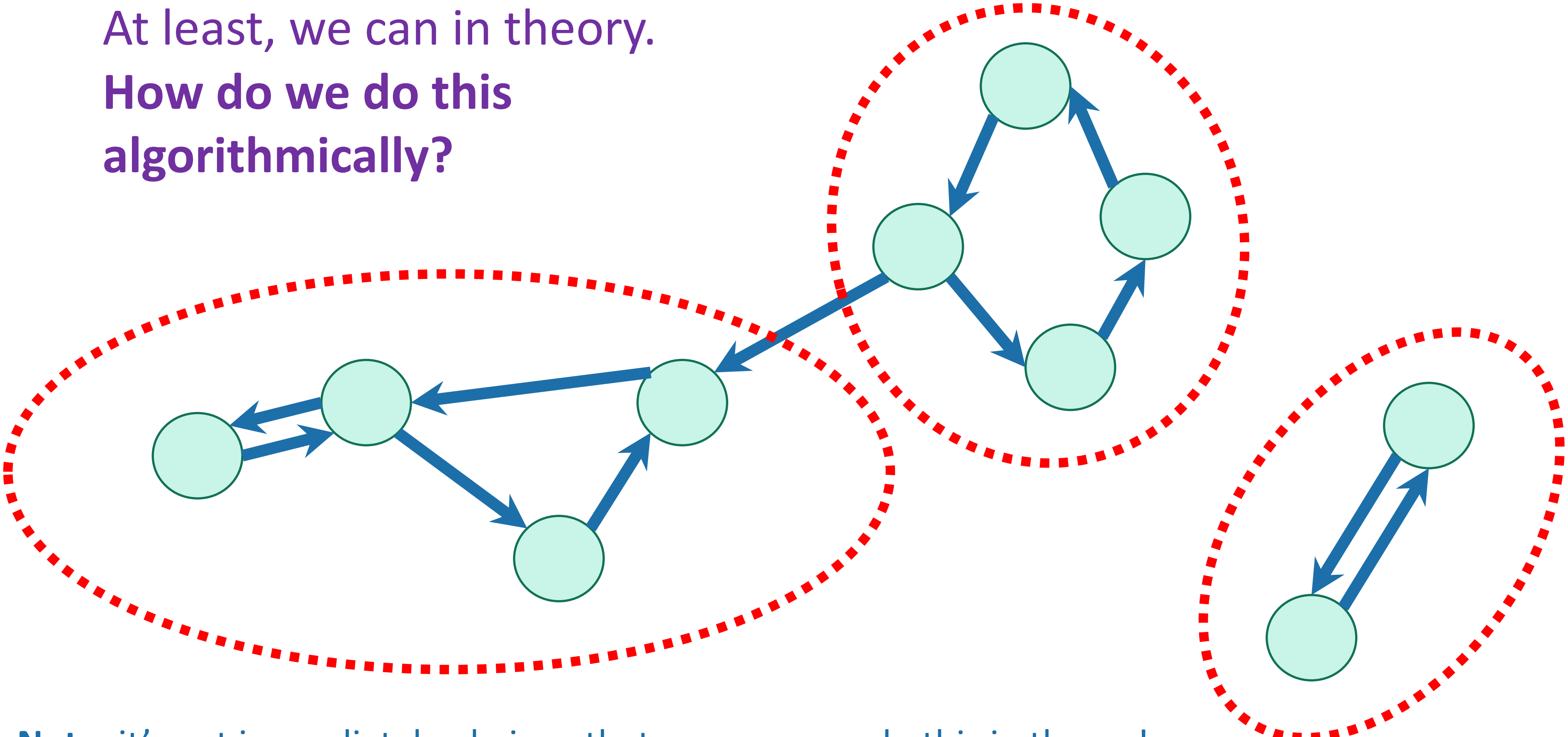
strongly connected



not strongly connected

We can decompose a graph into strongly connected components (SCCs)

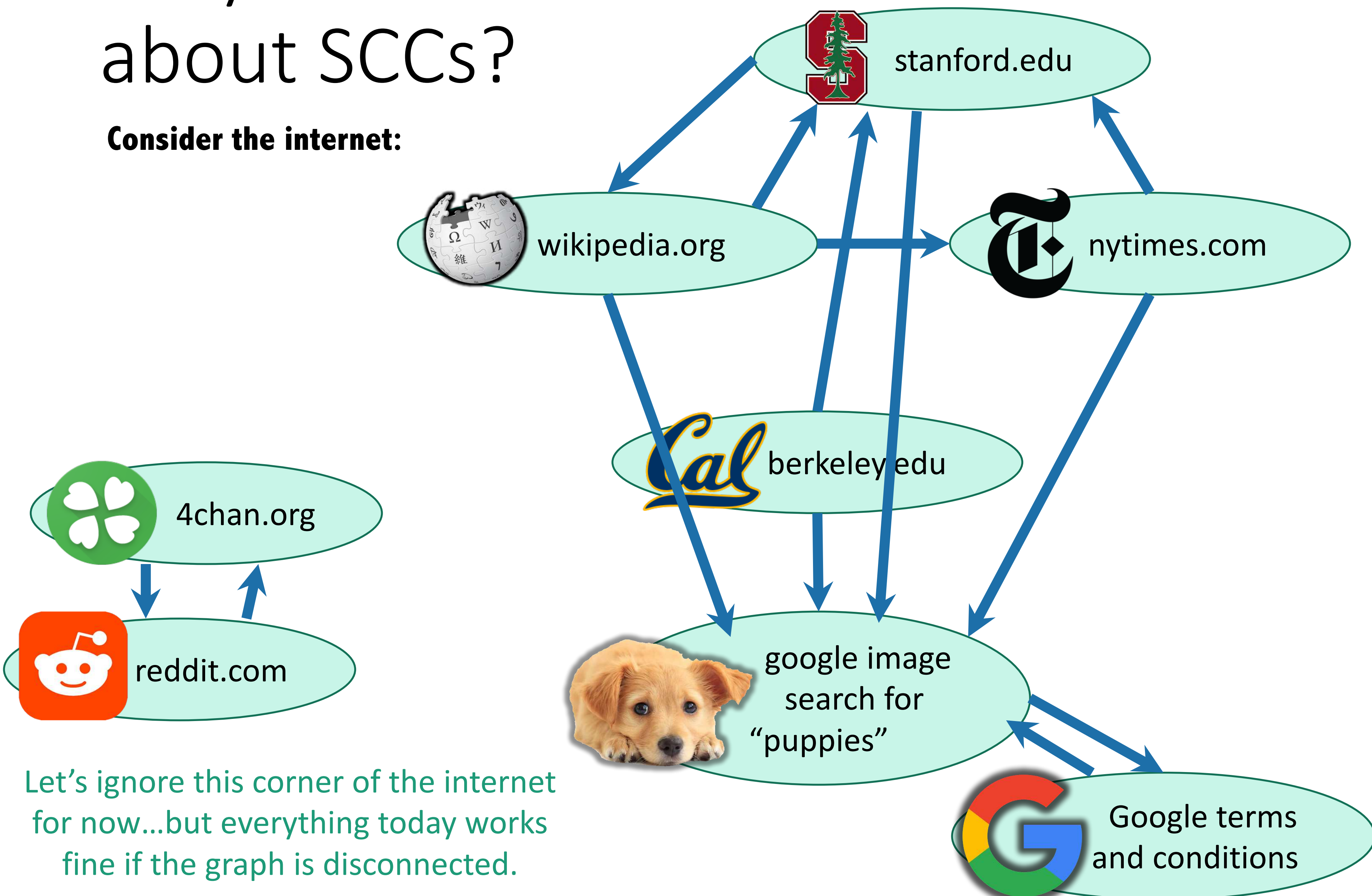
At least, we can in theory.
How do we do this algorithmically?



Note: it's not immediately obvious that we can even do this in theory!
The reason why is because "**two vertices are reachable from each other**" is an **equivalence relation**, and the SCCs are **equivalence classes.**

Why do we care about SCCs?

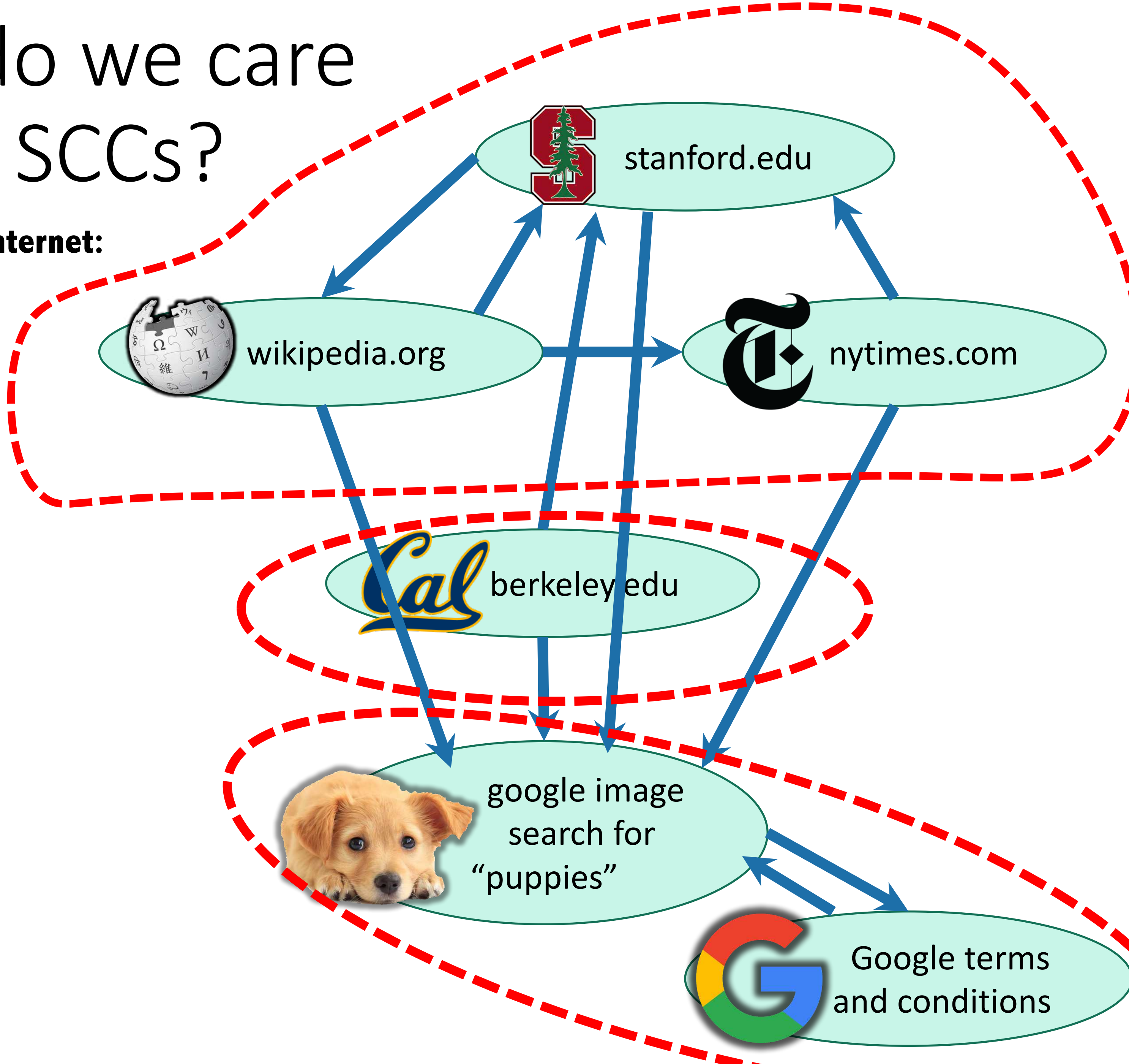
Consider the internet:



Let's ignore this corner of the internet for now…but everything today works fine if the graph is disconnected.

Why do we care about SCCs?

Consider the internet:



What are the SCCs of the internet?

- In real life, turns out there’s one “giant” one.
  - and then a bunch of tendrils.
- More generally:
  - Strongly connected components tell you about **communities.**
- Lots of graph algorithms only make sense on SCCs.
  - (So some times we want to find the SCCs as a first step)
  - Eg: I was talking to an economist the other day who has to first break up his labor market data into SCCs in order to make sense of it.

How to find SCCs?

Try 1:

- Consider all possible decompositions and check.

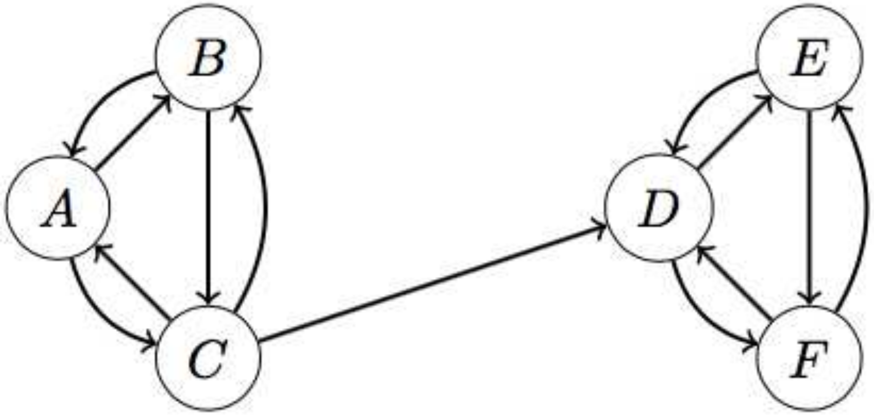
Try 2:

- For each pair (u,v),
  - use DFS to find if there are paths u to v and v to u.
- Aggregate accordingly.
- Running time: *[on board]*

(Definitely ***not*** any better than $O(n^2)$)

Pre-Lecture exercise

- Run DFS starting at D:



- That will identify SCCs…
- Issues:
  - How do we know where to start DFS?
  - It wouldn't have found the SCCs if we started from A.

Algorithm

Running time: O(n + m)

- Do DFS to create a **DFS forest**.
  - Choose starting vertices in any order.
  - Keep track of finishing times.
- Reverse all the edges in the graph.
- Do DFS again to create **another DFS forest**.
  - This time, order the nodes in the reverse order of the finishing times that they had from the first DFS run.
- The SCCs are the different trees in the **second DFS forest.**

Look, it works!

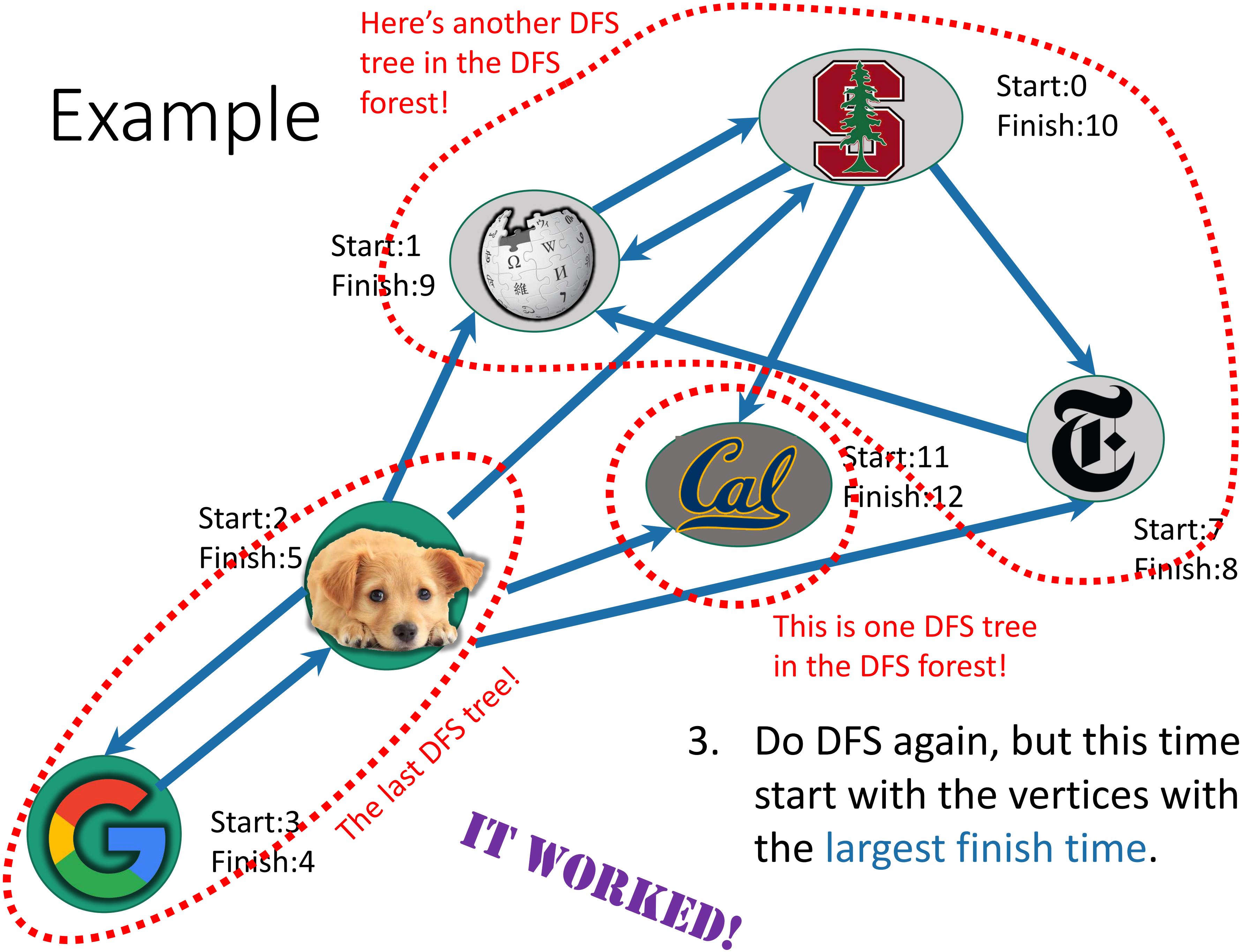
- (See IPython notebook)

```
In [4]: print(G)

CS161Graph with:
         Vertices:
        Stanford,Wikipedia,NYTimes,Berkeley,Puppies,Google,
         Edges:
        (Stanford,Wikipedia) (Stanford,Puppies) (Wikipedia,Stanford) (Wik
ipedia,NYTimes) (Wikipedia,Puppies) (NYTimes,Stanford) (NYTimes,Puppies)
(Berkeley,Stanford) (Berkeley,Puppies) (Puppies,Google) (Google,Puppies)
```

```
In [5]: SCCs = SCC(G, False)
        for X in SCCs:
            print ([str(x) for x in X])

['Berkeley']
['Stanford', 'NYTimes', 'Wikipedia']
['Puppies', 'Google']
```

But let’s break that down a bit…

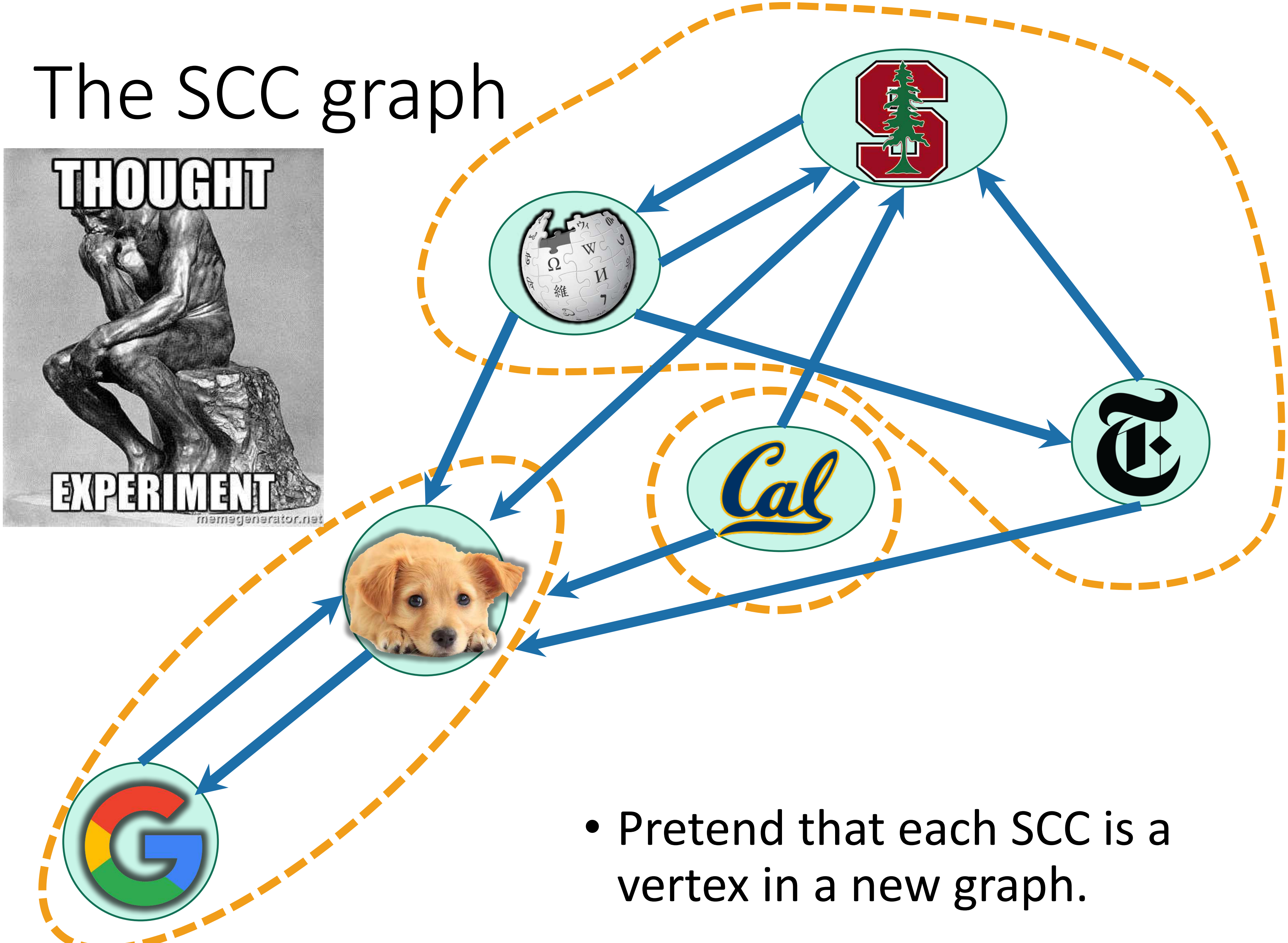
Example



3. Do DFS again, but this time, start with the vertices with the largest finish time.

One question

The SCC graph

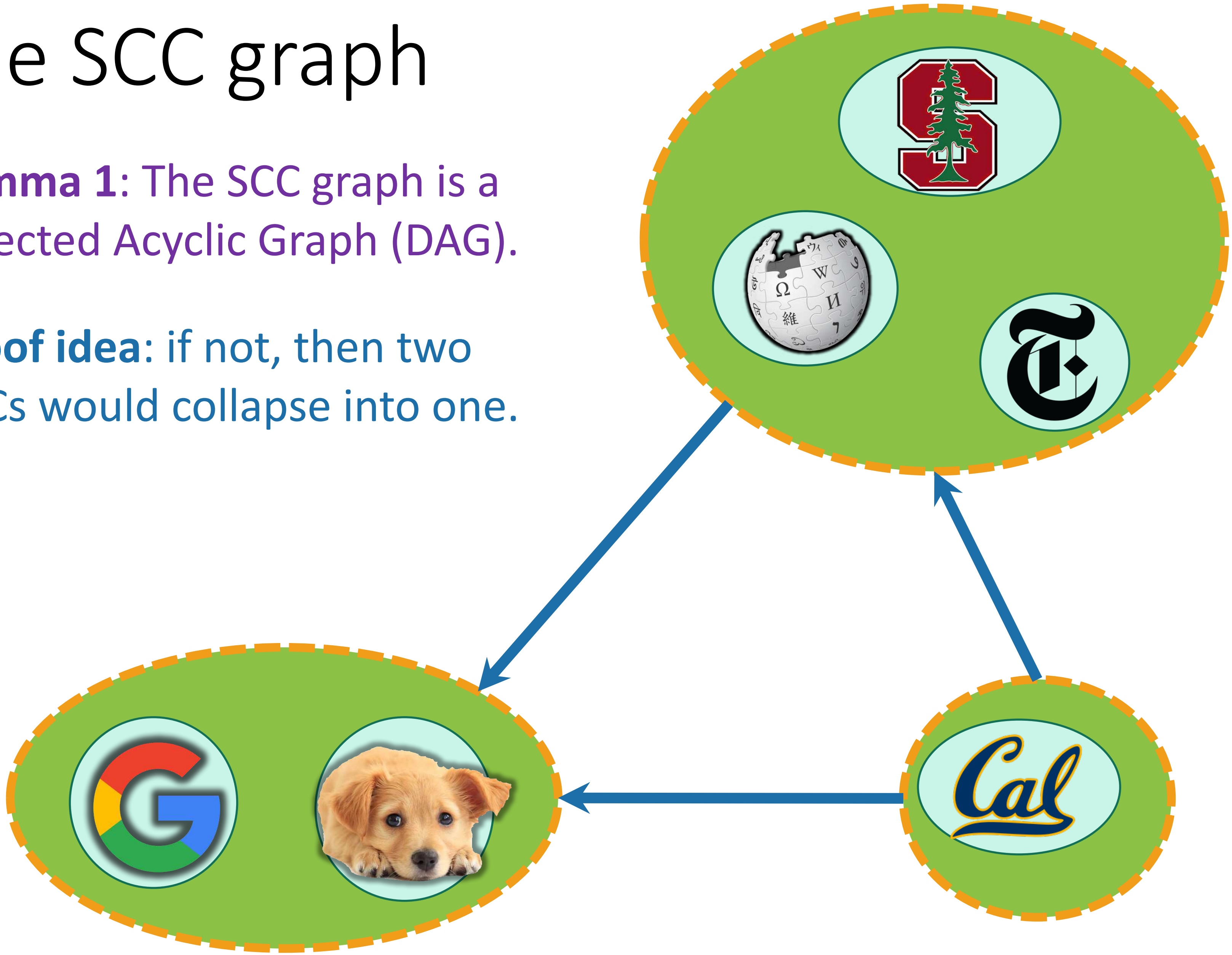


- Pretend that each SCC is a vertex in a new graph.

The SCC graph

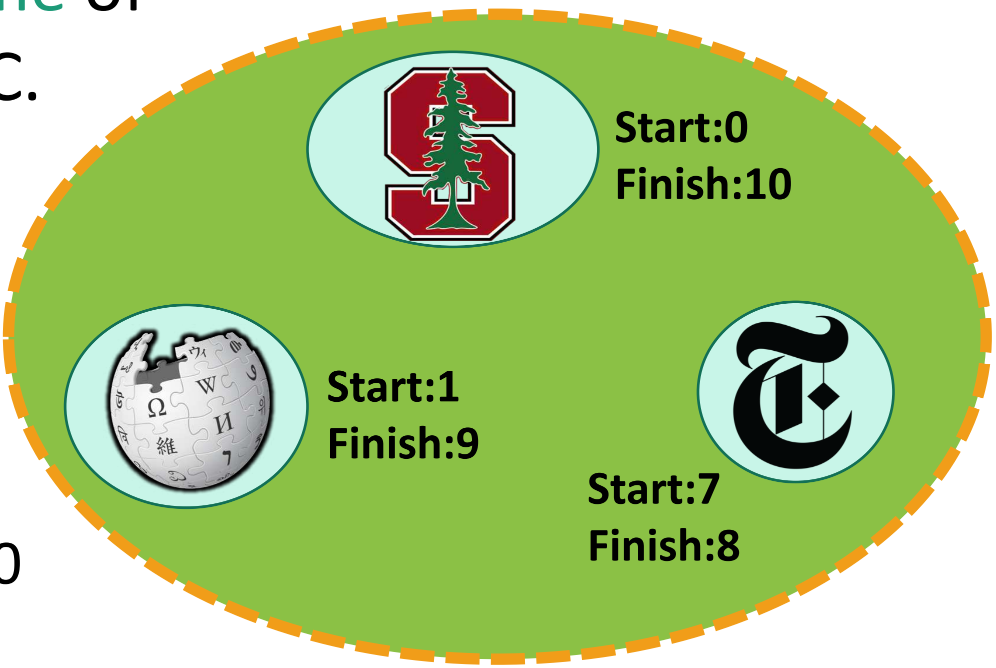
Lemma 1: The SCC graph is a Directed Acyclic Graph (DAG).

Proof idea: if not, then two SCCs would collapse into one.



Starting and finishing times in a SCC

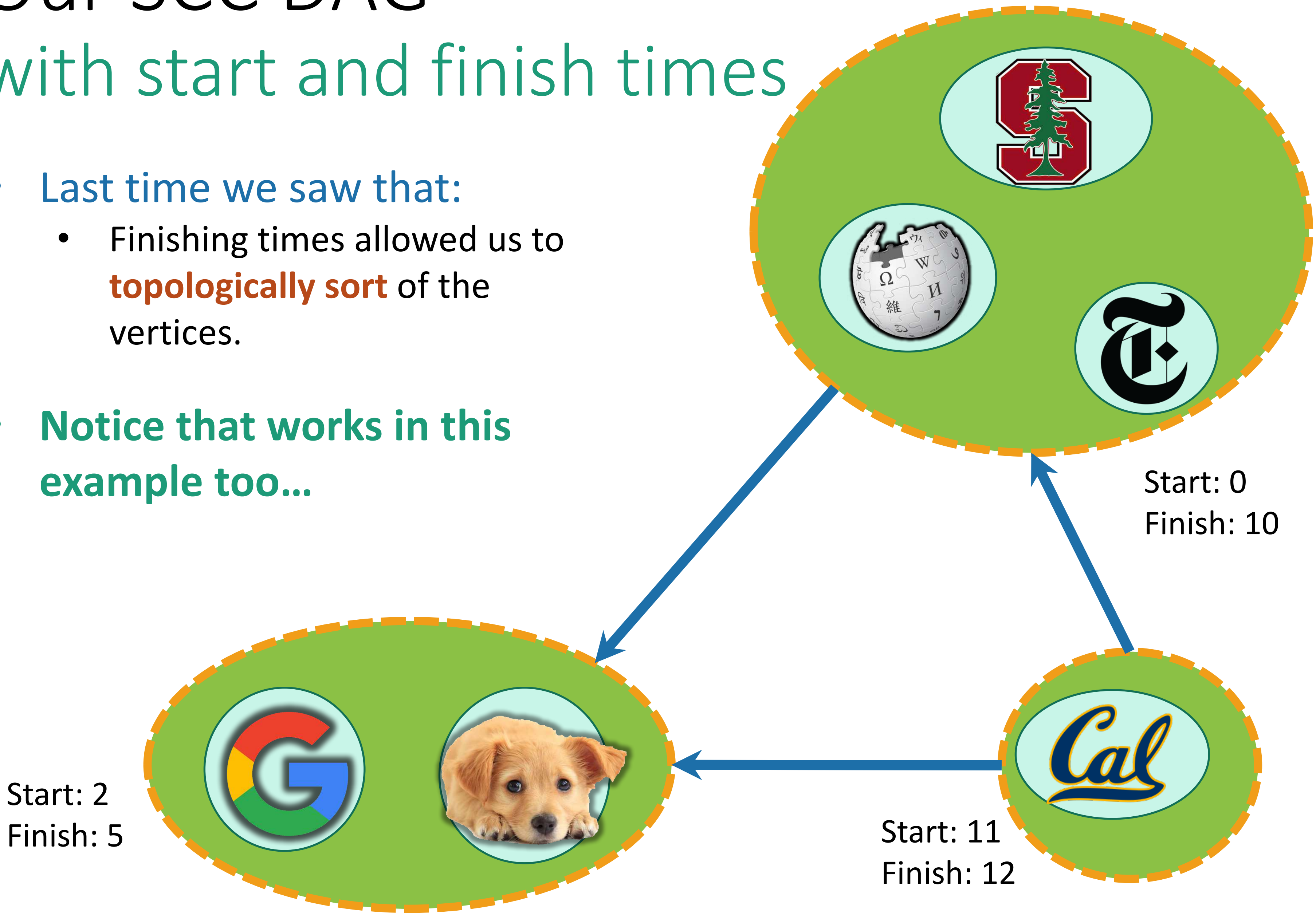
- The **finishing time** of a SCC is the largest finishing time of any element of that SCC.
- The **starting time** of a SCC is the smallest starting time of any element of that SCC.



Start: 0
Finish: 10

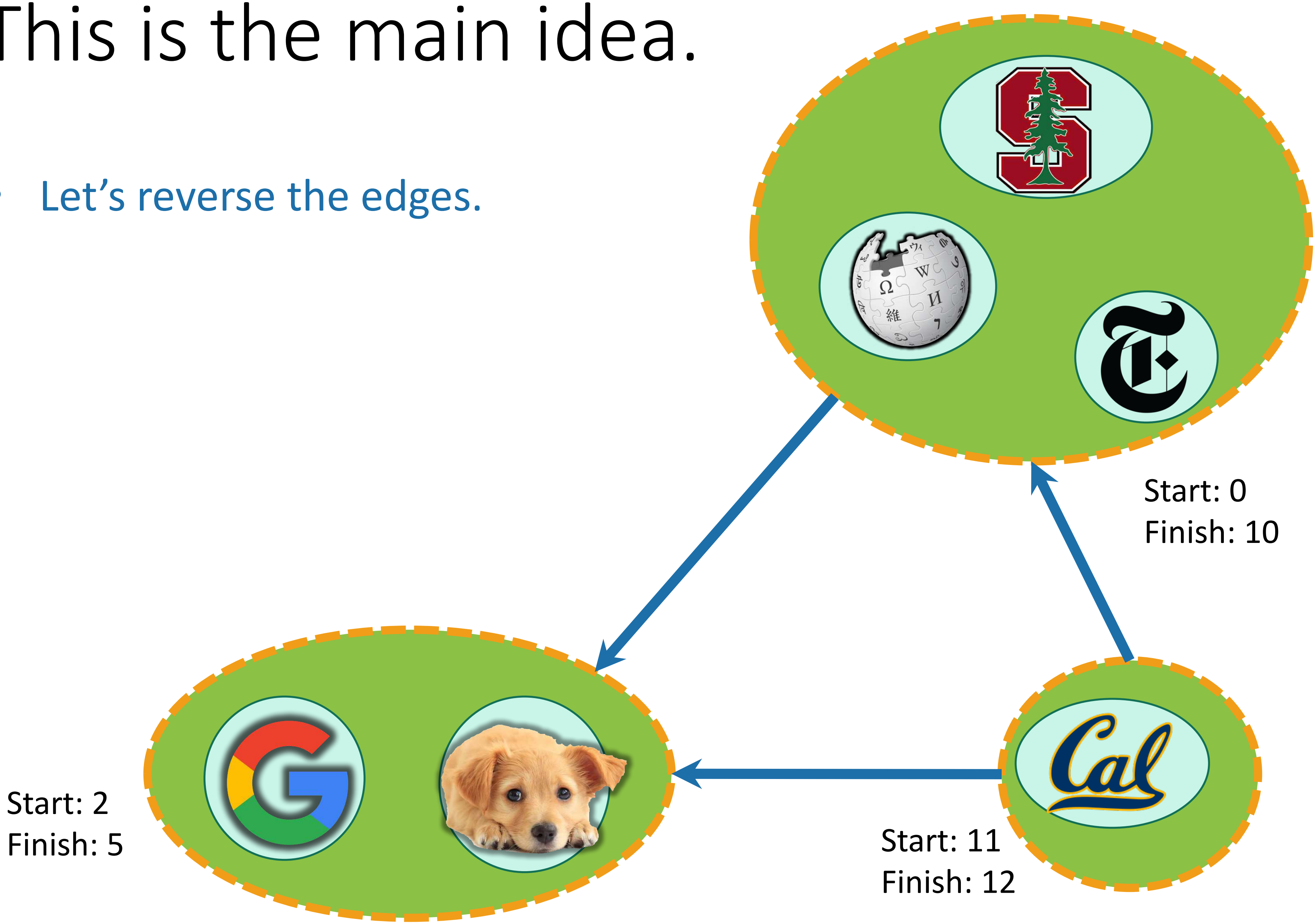
Our SCC DAG
with start and finish times

- Last time we saw that:
  - Finishing times allowed us to **topologically sort** of the vertices.
- **Notice that works in this example too…**



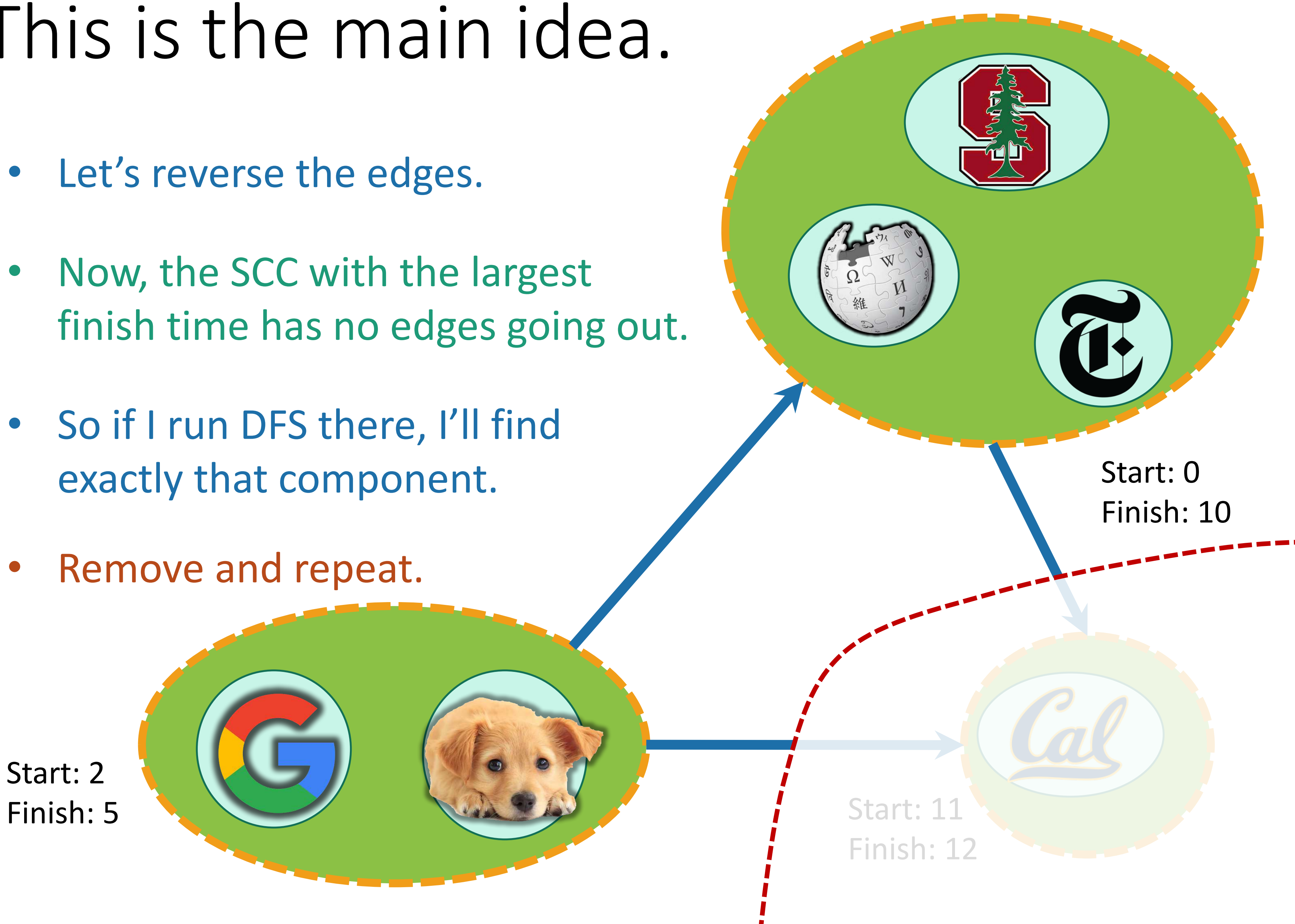
This is the main idea.

- Let's reverse the edges.



This is the main idea.

- Let’s reverse the edges.
- Now, the SCC with the largest finish time has no edges going out.
- So if I run DFS there, I’ll find exactly that component.
- Remove and repeat.



Let’s make this idea formal.

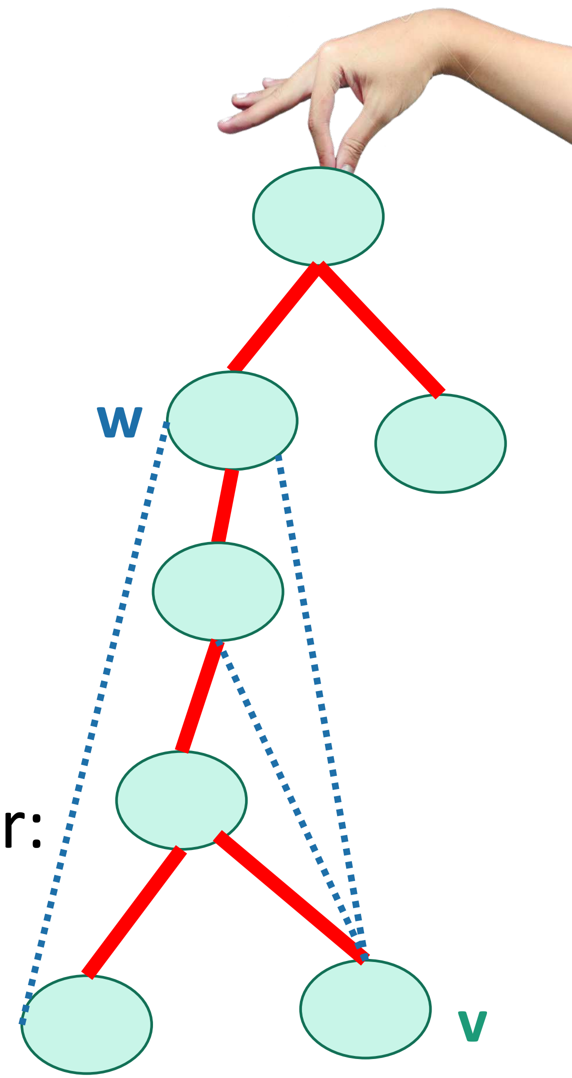
Back the the parentheses theorem

- If v is a descendent of w in this tree:

- If w is a descendent of v in this tree:

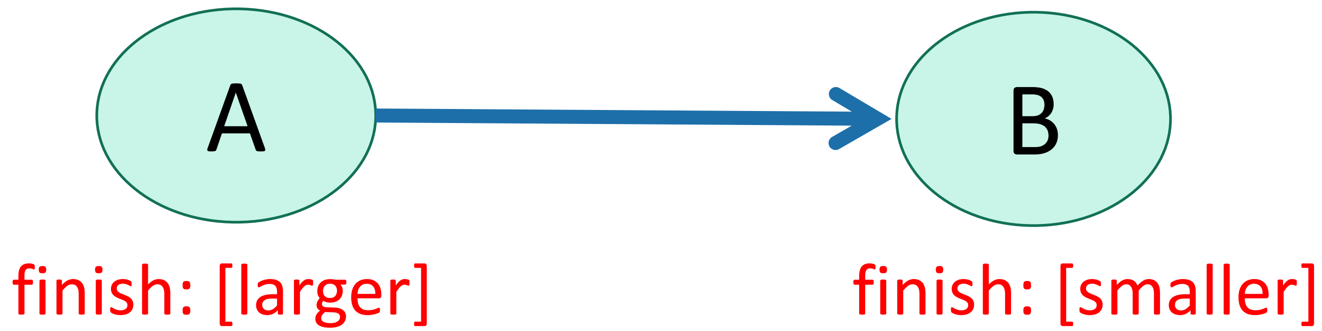
- If neither are descendents of each other:

(or the other way around)



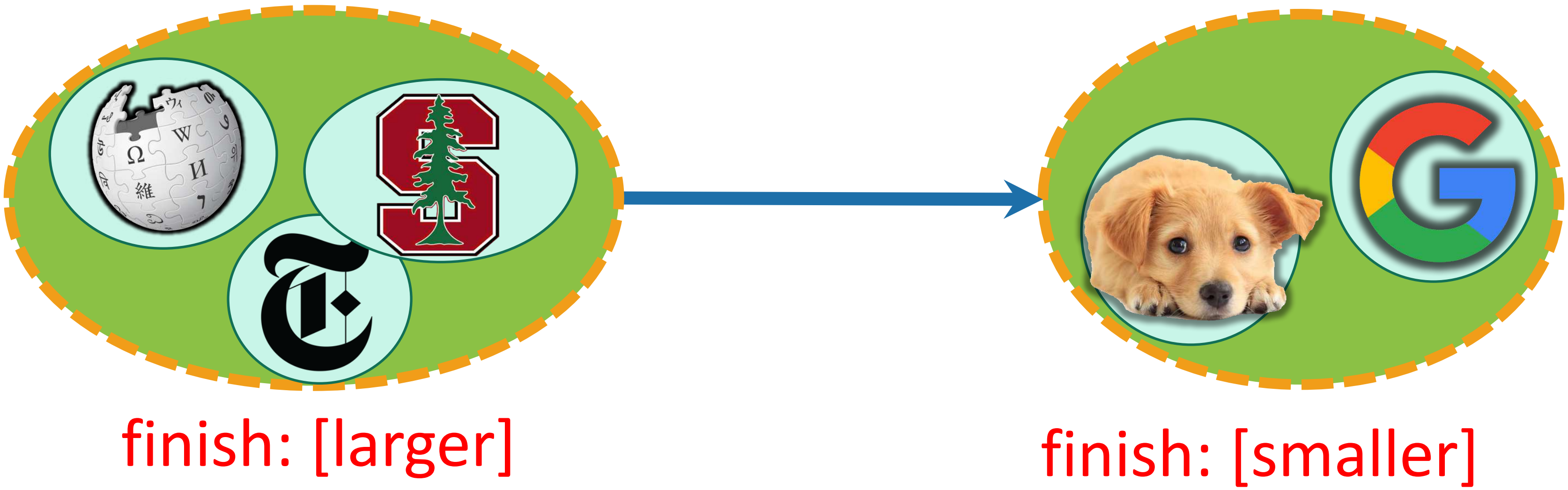
As we saw (correctly this time…)

Claim: In a DAG, we'll always have:

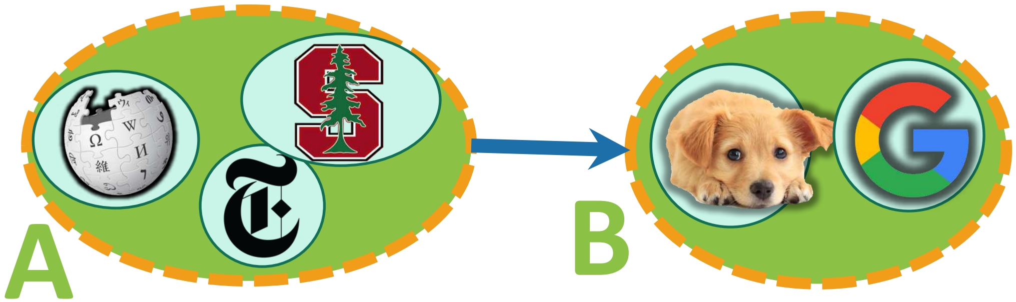


Same thing, in the SCC DAG.

- Claim: we'll always have



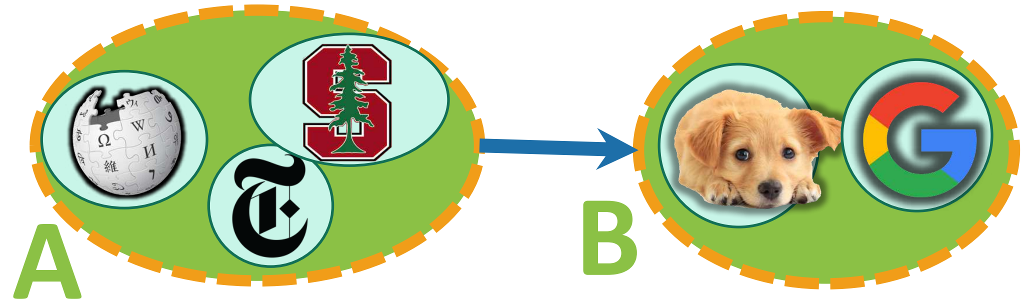
Proof idea



Want to show A.finish > B.finish.

- **Two cases**:
  - We reached A before B in our first DFS.
  - We reached B before A in our first DFS.

Proof idea



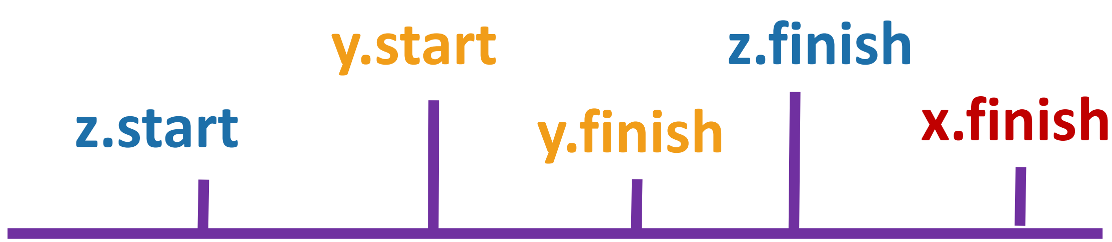
Want to show A.finish > B.finish.

- **Case 1**: We reached A before B in our first DFS.
- Say that:
  - x has the largest finish time in A;
  - y has the largest finish in B;
  - z was discovered first in A;
- Then:
  - Reach A before B
  - => we will discover y via z
  - => y is a descendant of z in the DFS forest.

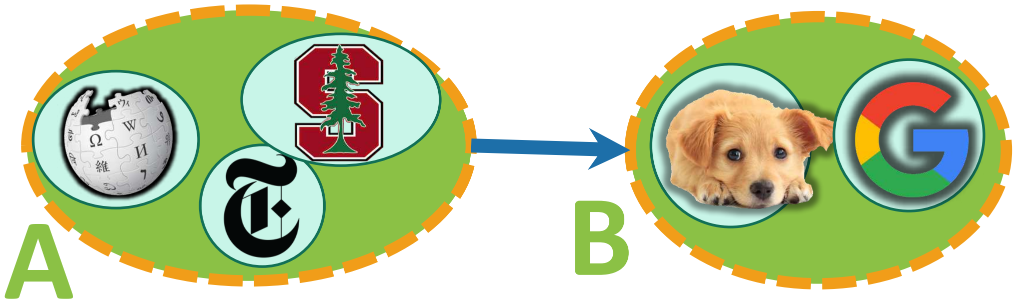
So A.finish = x.finish
B.finish = y.finish
x.finish >= z.finish

aka,
A.finish > B.finish

- Then



Proof idea



Want to show A.finish > B.finish.

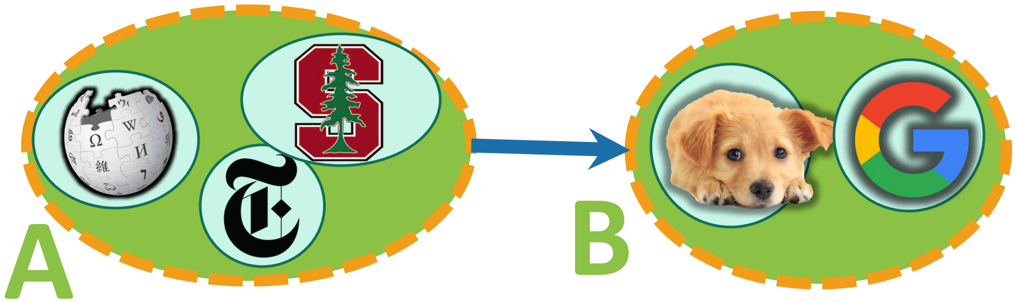
- **Case 2**: We reached B before A in our first DFS.

- There are no paths from B to A
  - because the SCC graph has no cycles

- So we completely finish exploring B and never reach A.
- A is explored later after we restart DFS.

aka,
A.finish > B.finish

Proof idea



Want to show A.finish > B.finish.

- **Two cases**:
  - We reached A before B in our first DFS.
  - We reached B before A in our first DFS.

- In either case:

$\boxed{\textbf{A.finish > B.finish}}$

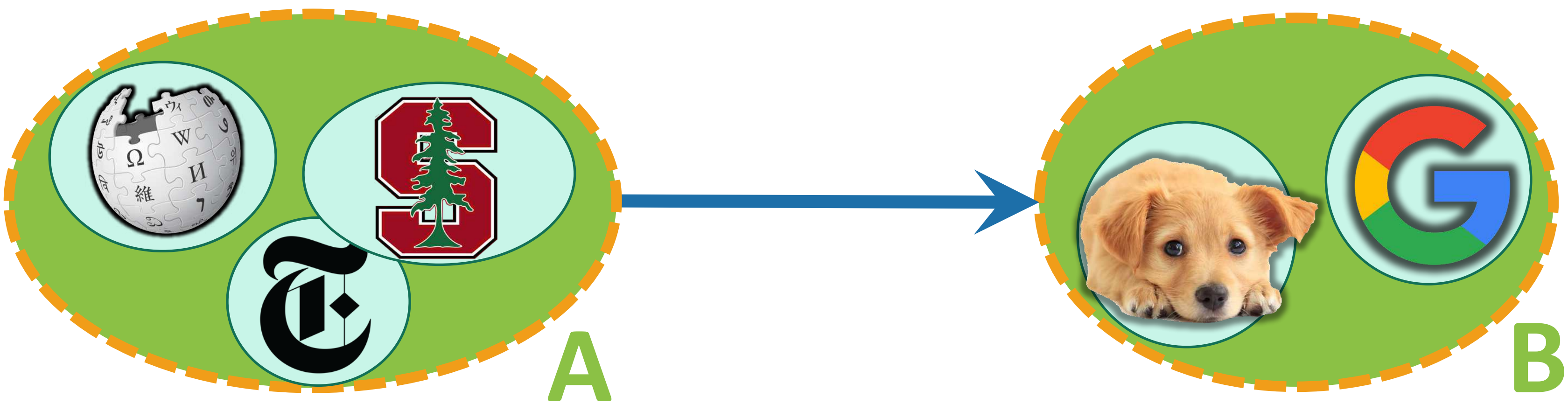
which is what we wanted to show.

Notice: this is exactly the same two-case argument that we did earlier, just with the SCC DAG!

This establishes:

Lemma 2

- If there is an edge like this:



finish: [larger]

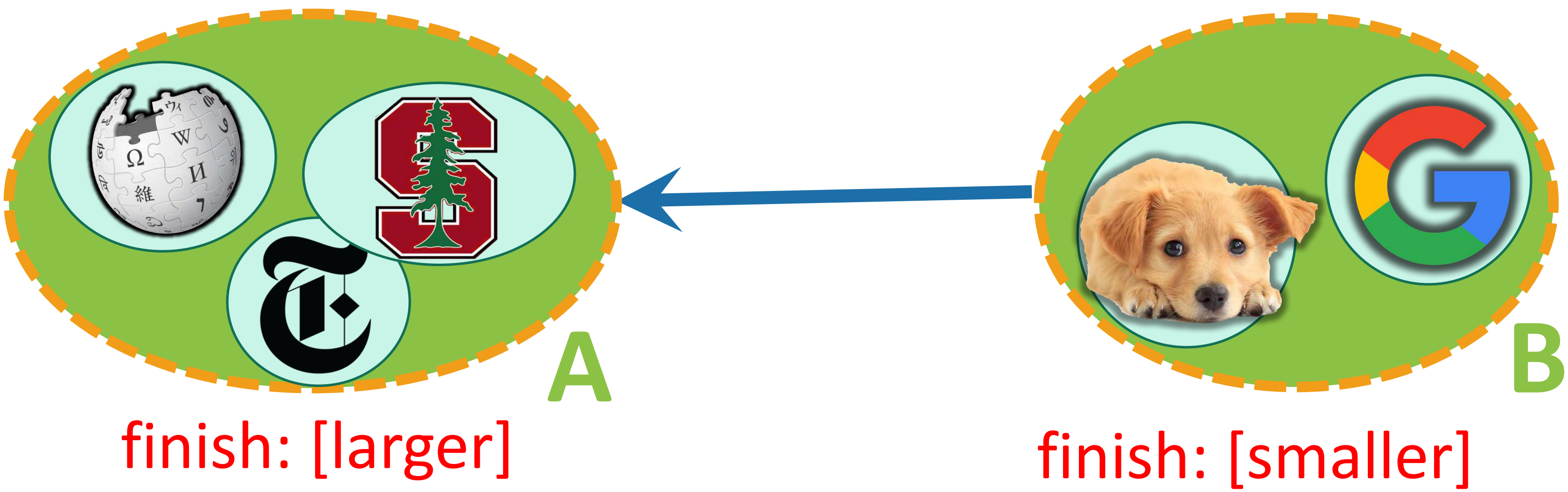
finish: [smaller]

- Then A.finish > B.finish.

This establishes:

Corollary 1

- If there is an edge like this in the **reversed graph**:

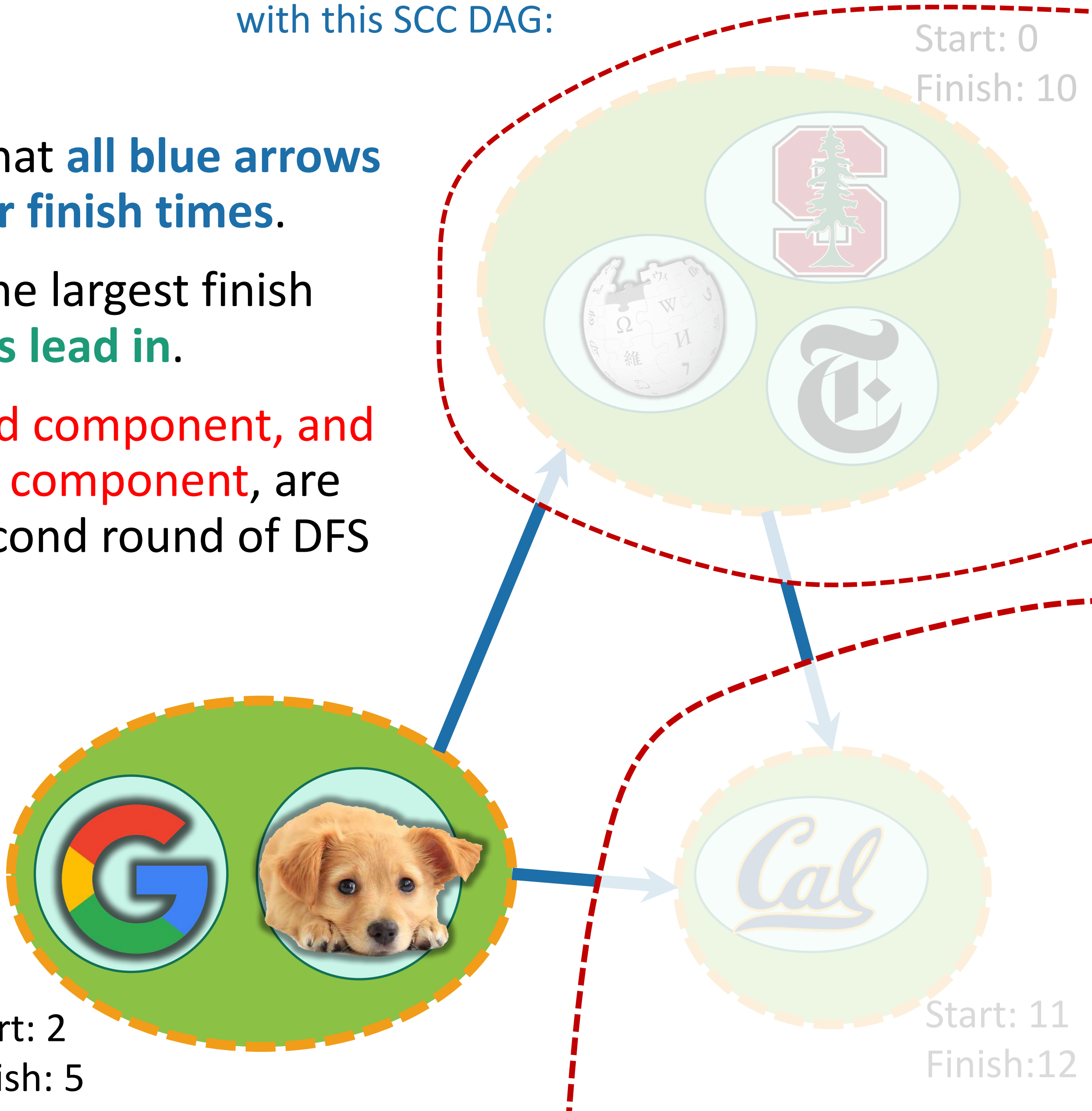


- Then A.finish > B.finish.

Now we see why this works.

Remember that after the first round of DFS, and after we reversed all the edges, we ended up with this SCC DAG:

- The Corollary says that **all blue arrows point towards larger finish times**.
- So if we start with the largest finish time, **all blue arrows lead in**.
- Thus, that connected component, and only that connected component, are reachable by the second round of DFS



- Now, we've deleted that first component.
- The next one has the **next biggest finishing time.**
- So **all remaining blue arrows lead in.**
- Repeat.

Formally, we prove it by induction

- **Theorem**: The algorithm we saw before will correctly identify strongly connected components.

- **Inductive hypothesis**:
  - The first t trees found in the second (reversed) DFS forest are the t SCCs with the largest finish times.
  - Moreover, what's left unvisited after these t trees have been explored is a DAG on the un-found SCCs.
- **Base case**: (t=0)
  - The first 0 trees found in the reversed DFS forest are the 0 SCCs with the largest finish times. **(TRUE)**
  - Moreover, what's left unvisited after 0 trees have been explored is a DAG on all the SCCs. **(TRUE by Lemma 1.)**

Inductive step [drawing on board to supplement]

- **Assume by induction that the first t trees are the last-finishing SCCs, and the remaining SCCs form a DAG.**
- Consider the $(t+1)^{st}$ tree produced, suppose the root is **x**.
- Suppose that **x** lives in the SCC **A**.
- Then **A**.finish > **B**.finish for all remaining SCCs **B**.
  - This is because we chose **x** to have the largest finish time.
- Then there are no edges leaving **A** in the remaining SCC DAG.
  - This follows from the Corollary.
- Then DFS started at **x** recovers exactly **A**.
  - It doesn't recover any more since nothing else is reachable.
  - It doesn't recover any less since A is strongly connected.
  - (Notice that we are using that A is still strongly connected when we reverse all the edges).
- **So the $(t+1)^{st}$ tree is the SCC with the $(t+1)^{st}$ biggest finish time.**

Formally, we prove it by induction

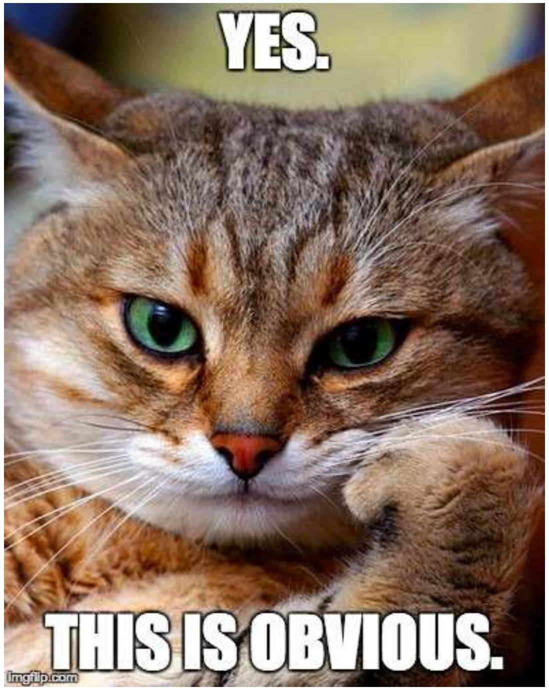
- **Theorem**: The algorithm we saw before will correctly identify strongly connected components.

- **Inductive hypothesis**:
  - The first t trees found in the second (reversed) DFS forest are the t SCCs with the largest finish times.
  - Moreover, what's left unvisited after these t trees have been explored is a DAG on the un-found SCCs.

- **Base case**: *[done]*
- **Inductive step**: *[done]*
- **Conclusion:** The second (reversed) DFS forest contains all the SCCs as its trees!
  - (This is the first bullet of **IH** when t = #SCCs)

Punchline:

we can find SCCs in time O(n + m)



`Algorithm:`

- Do DFS to create a **DFS forest**.
  - Choose starting vertices in any order.
  - Keep track of finishing times.
- Reverse all the edges in the graph.
- Do DFS again to create **another DFS forest**.
  - This time, order the nodes in the reverse order of the finishing times that they had from the first DFS run.
- The SCCs are the different trees in the **second DFS forest.**

(Clearly it wasn't obvious since it took all class to do! But hopefully it is less mysterious now.)

Recap

- Depth First Search reveals a very useful structure!
  - We saw Monday that this structure can be used to do **Topological Sorting** in time O(n+m)
  - Today we saw that it can also find **Strongly Connected Components** in time O(n + m)
  - This was pretty non-trivial.

Next time

- **MIDTERM**

BEFORE Next time

- Study for the midterm!