

# Lecture 10

Finding strongly connected components

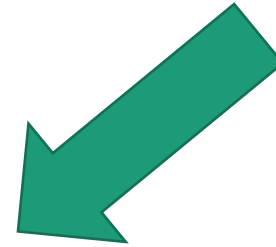
# Announcements

- MIDTERM 😊!
- Pick it up today after class.
  - CAs will be waiting outside
  - Go up to one, tell them your ID number, receive an envelope with your exam.
- During the exam period:
  - Piazza will be disabled while the midterm is out.
  - There will be no office hours Monday/Tuesday.
- During your exam:
  - Follow the instructions with the exam.
  - If you have a question about the exam, do not stop the exam; instead write on the exam that you found X confusing and that you assume it means Y.
- **DO NOT LOSE YOUR EXAM!!!!**
  - You must be in possession of your exam at all times.
  - If you lose your exam you will receive a zero.

# Last time

- Breadth-first and depth-first search
- Plus, applications!
  - Topological sorting
  - In-order traversal of BSTs
- The key was paying attention to the structure of the tree that these search algorithms implicitly build.

# Today

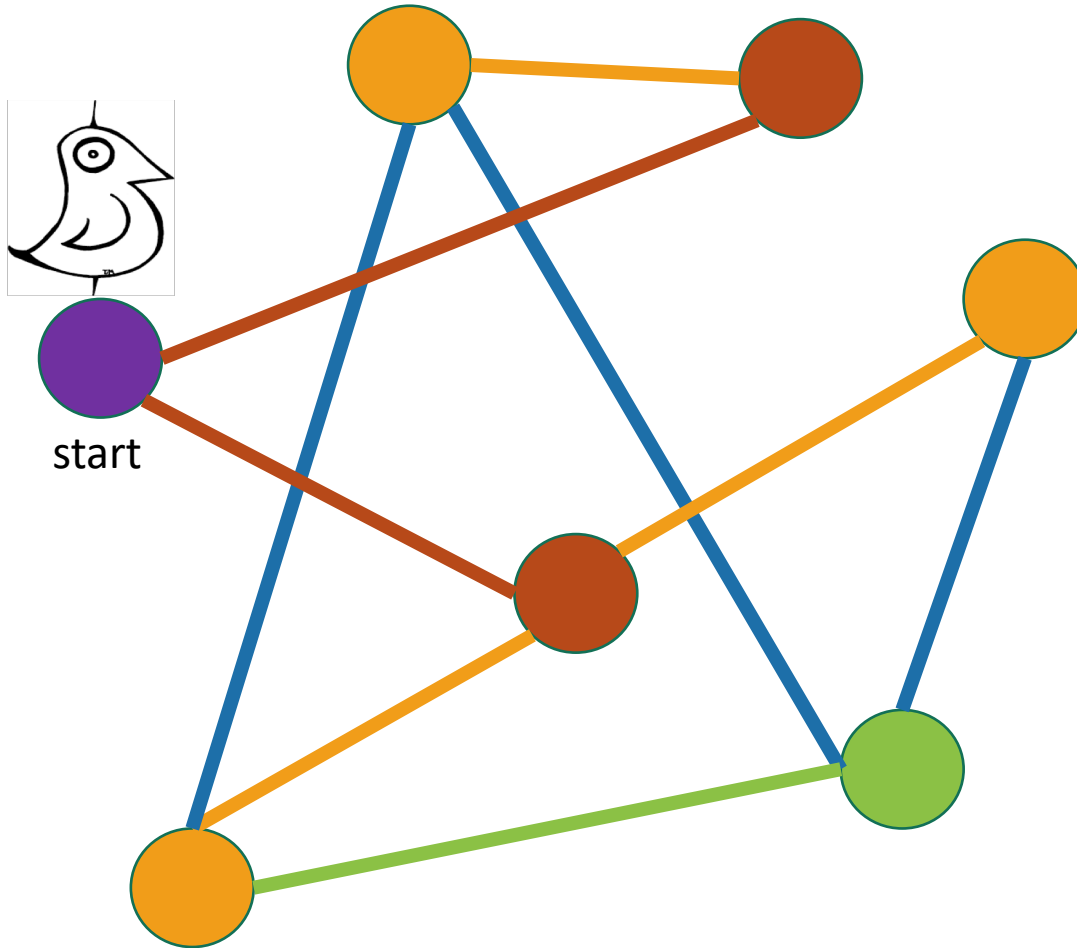


- Finish up BFS with an application:
  - Shortest path in unweighted graphs
  - (**Note:** on the slides from last week there's another application to testing bipartite-ness – we won't get to that in lecture due to time constraints, but you might want to check out the slides if you are interested!)
- One more application of DFS:

## Finding Strongly Connected Components

# Breadth-First Search

Exploring the world with a bird's-eye view



Not been there yet

Can reach there in zero steps

Can reach there in one step

Can reach there in two steps

Can reach there in three steps

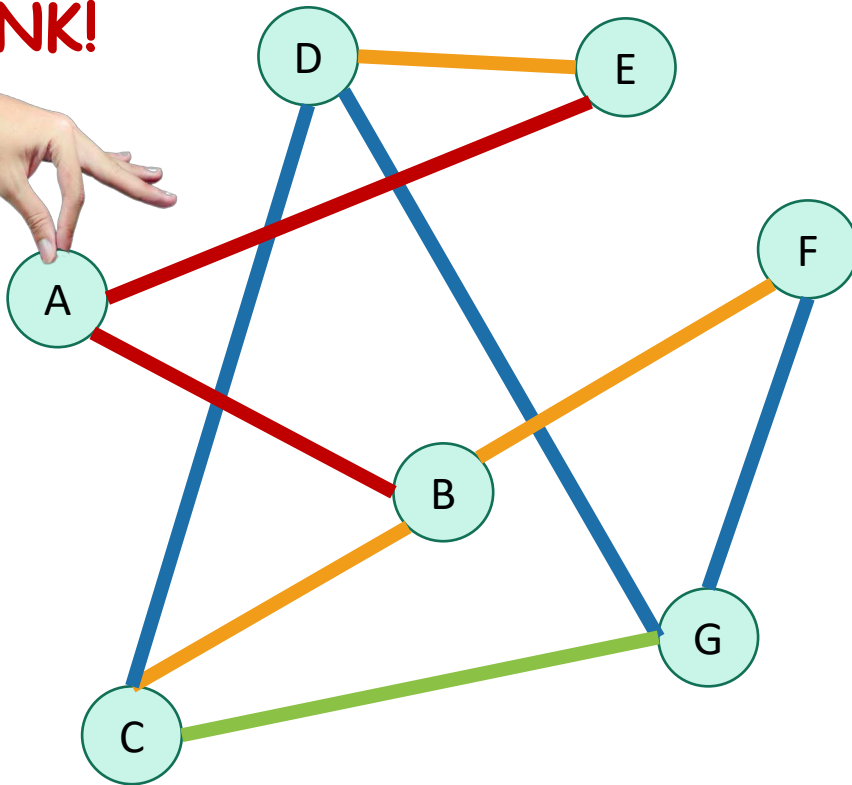
World:

**EXPLORED!**

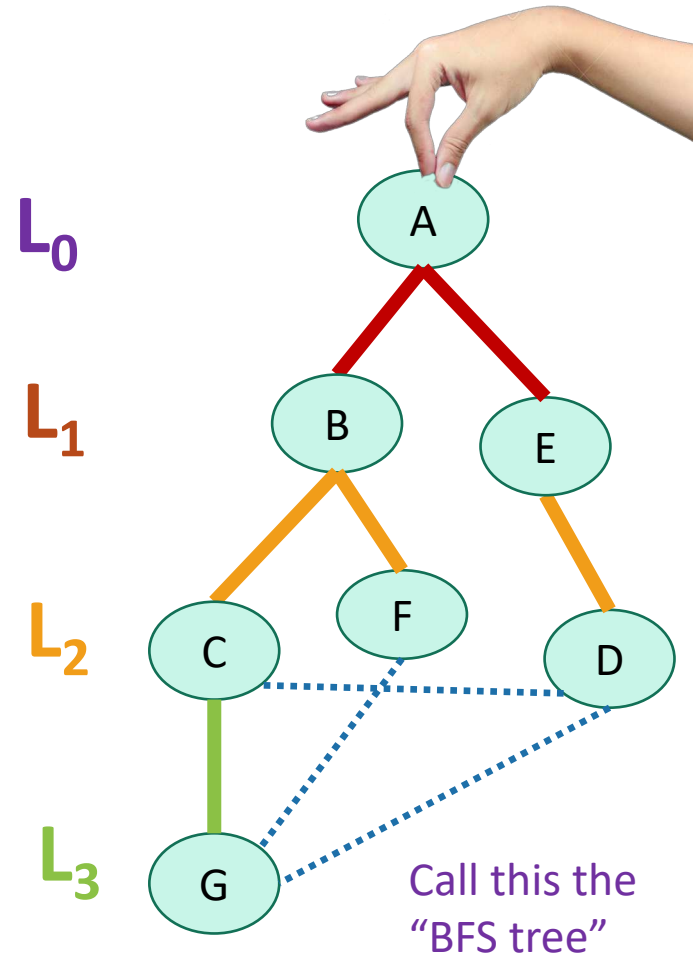
# Why is it called breadth-first?

- We are implicitly building a tree:

**YOINK!**

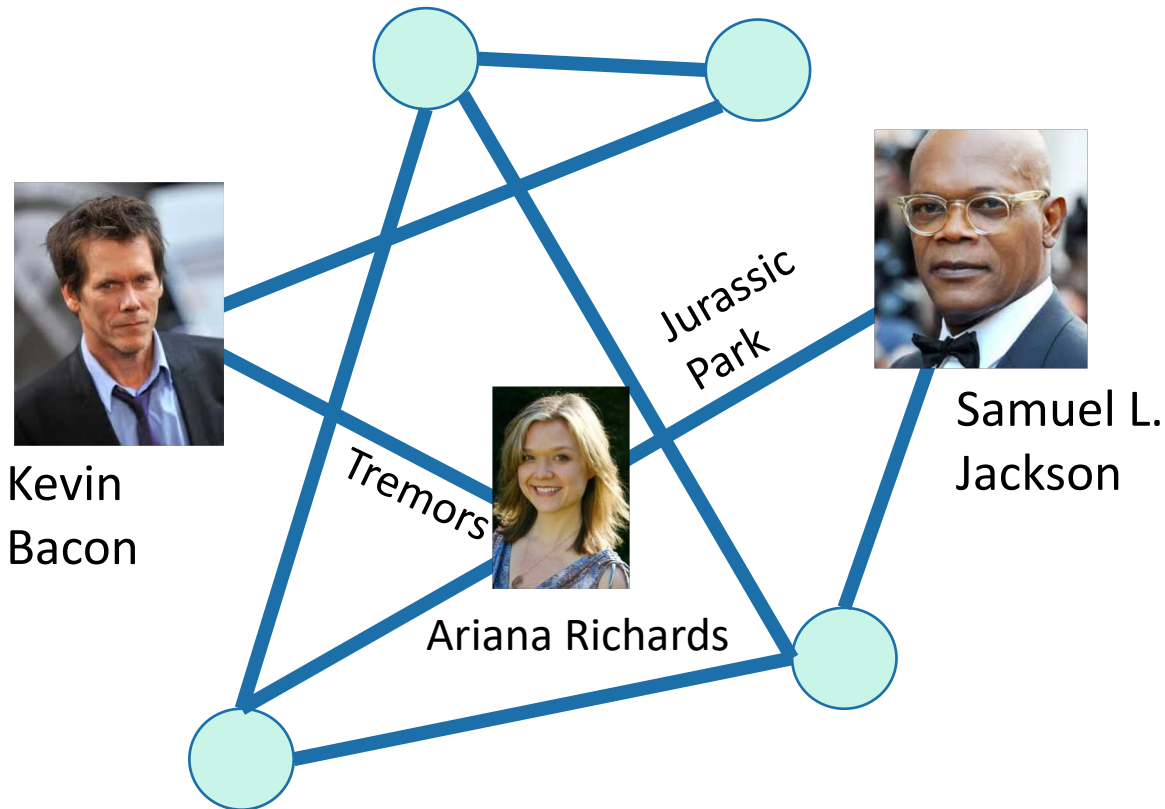


- First we go as broadly as we can.



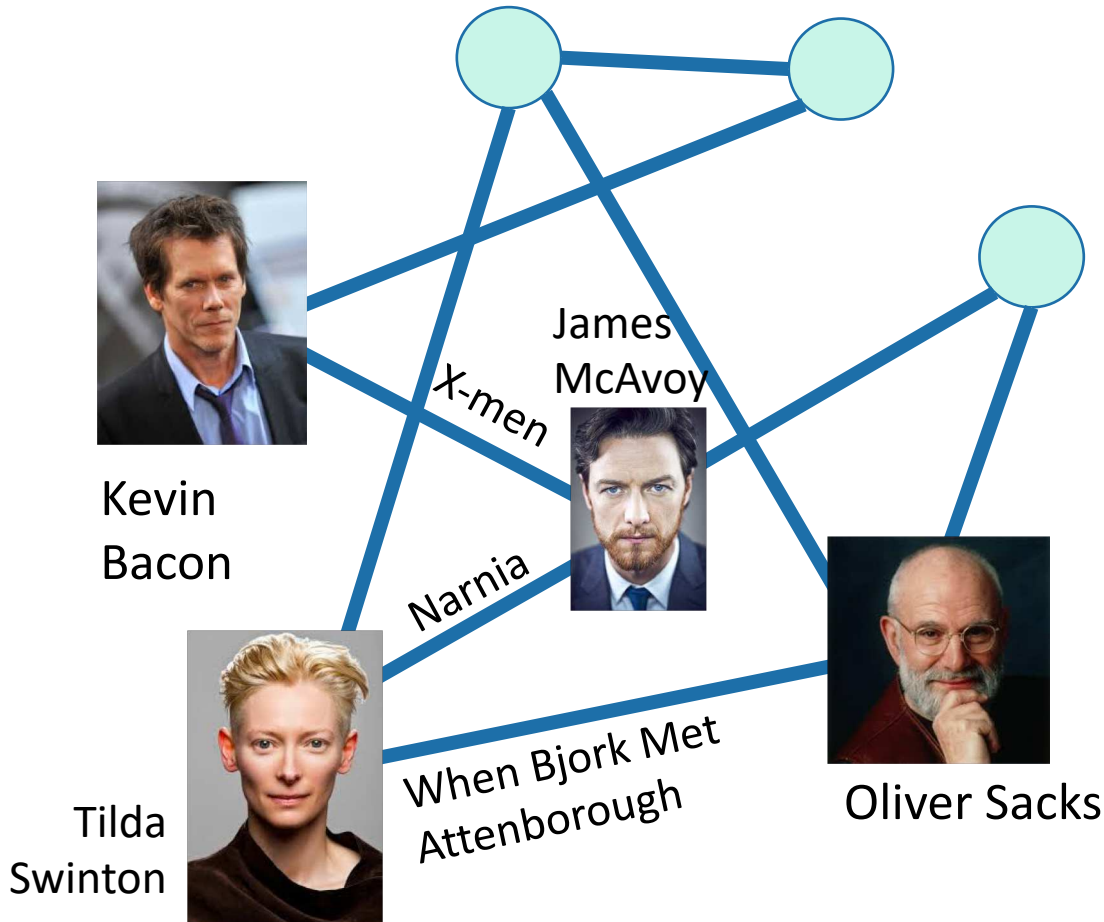
# Pre-lecture exercise from Lecture 9

- What Samuel L. Jackson's Bacon number?



(Answer: 2)

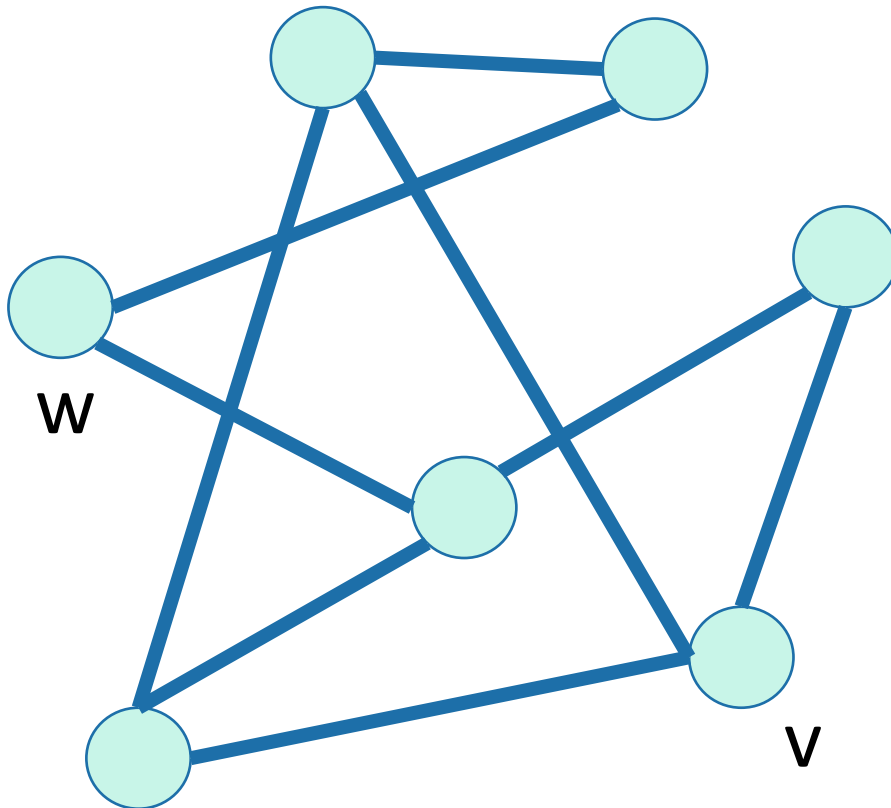
I wrote the pre-lecture exercise before I realized that I really wanted an example with distance 3



It is really hard to find  
people with Bacon  
number 3!

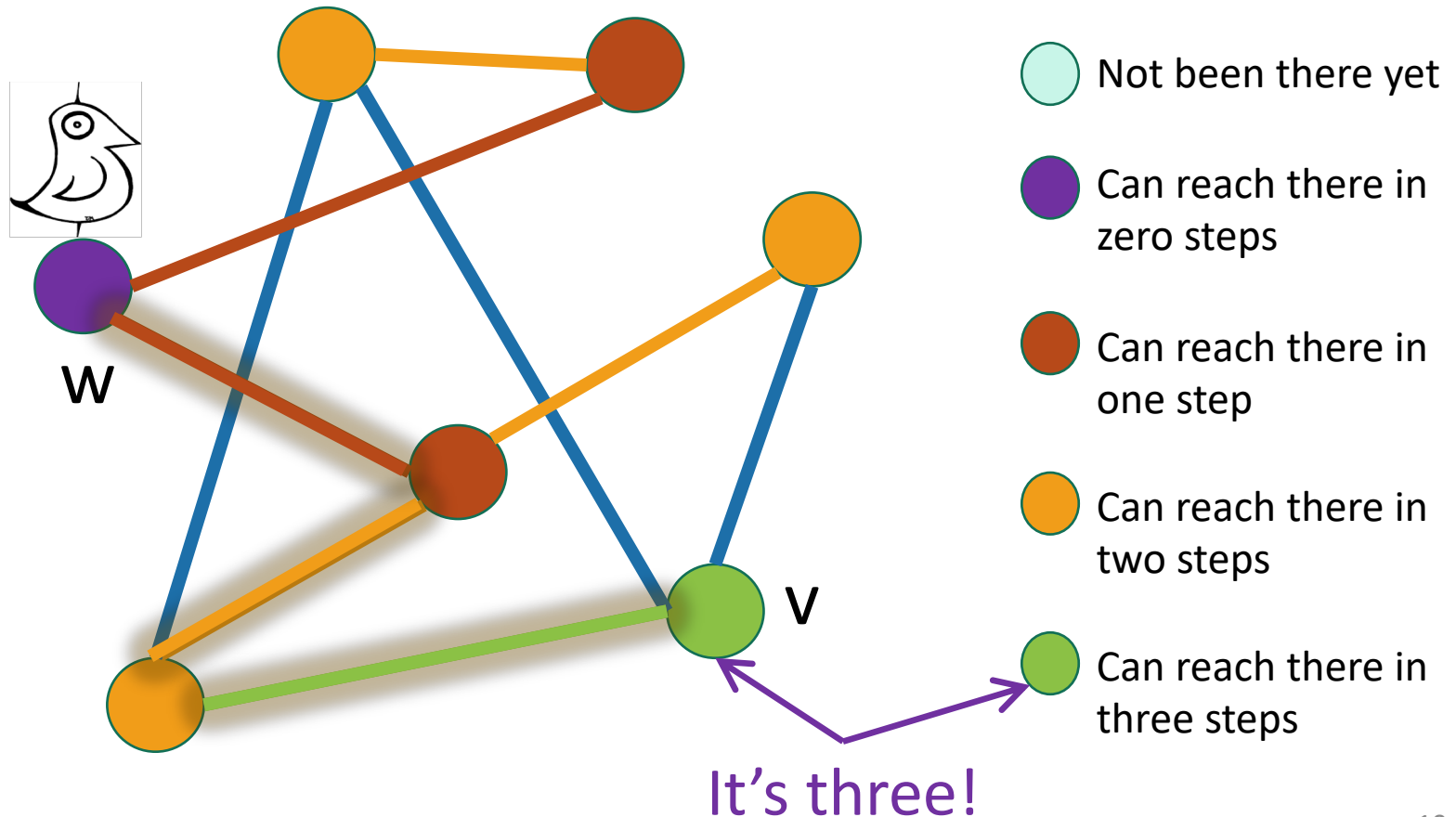
# Application of BFS: shortest path

- How long is the shortest path between  $w$  and  $v$ ?



# Application of BFS: shortest path

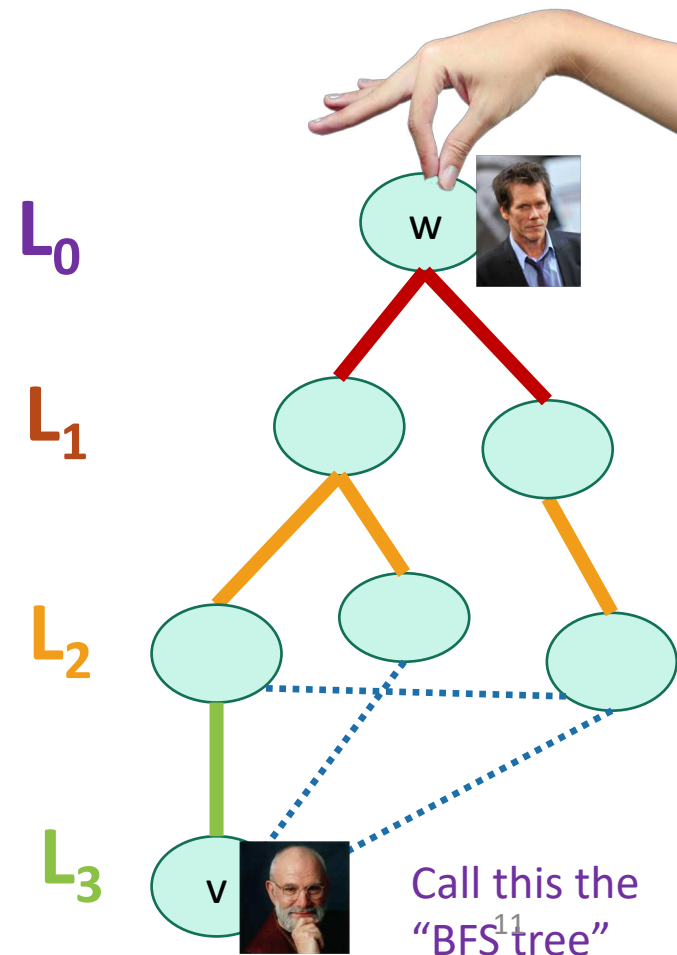
- How long is the shortest path between w and v?



# To find the **distance** between $w$ and all other vertices $v$

- Do a BFS starting at  $w$
- For all  $v$  in  $L_i$ 
  - The shortest path between  $w$  and  $v$  has length  $i$
  - A shortest path between  $w$  and  $v$  is given by the path in the BFS tree.
- If we never found  $v$ , the distance is infinite.

The **distance** between two vertices is the number of edges in the shortest path between them.



Modify the BFS pseudocode to return shortest paths!  
Prove that this indeed returns shortest paths!



Gauss has no Bacon number



# What have we learned?

- The BFS tree is useful for computing distances between pairs of vertices.
- We can find the shortest path between  $u$  and  $v$  in time  $O(m)$ .

# Today

- Finish up BFS with an application:
  - Shortest path in unweighted graphs
- One more application of DFS:

## Finding Strongly Connected Components

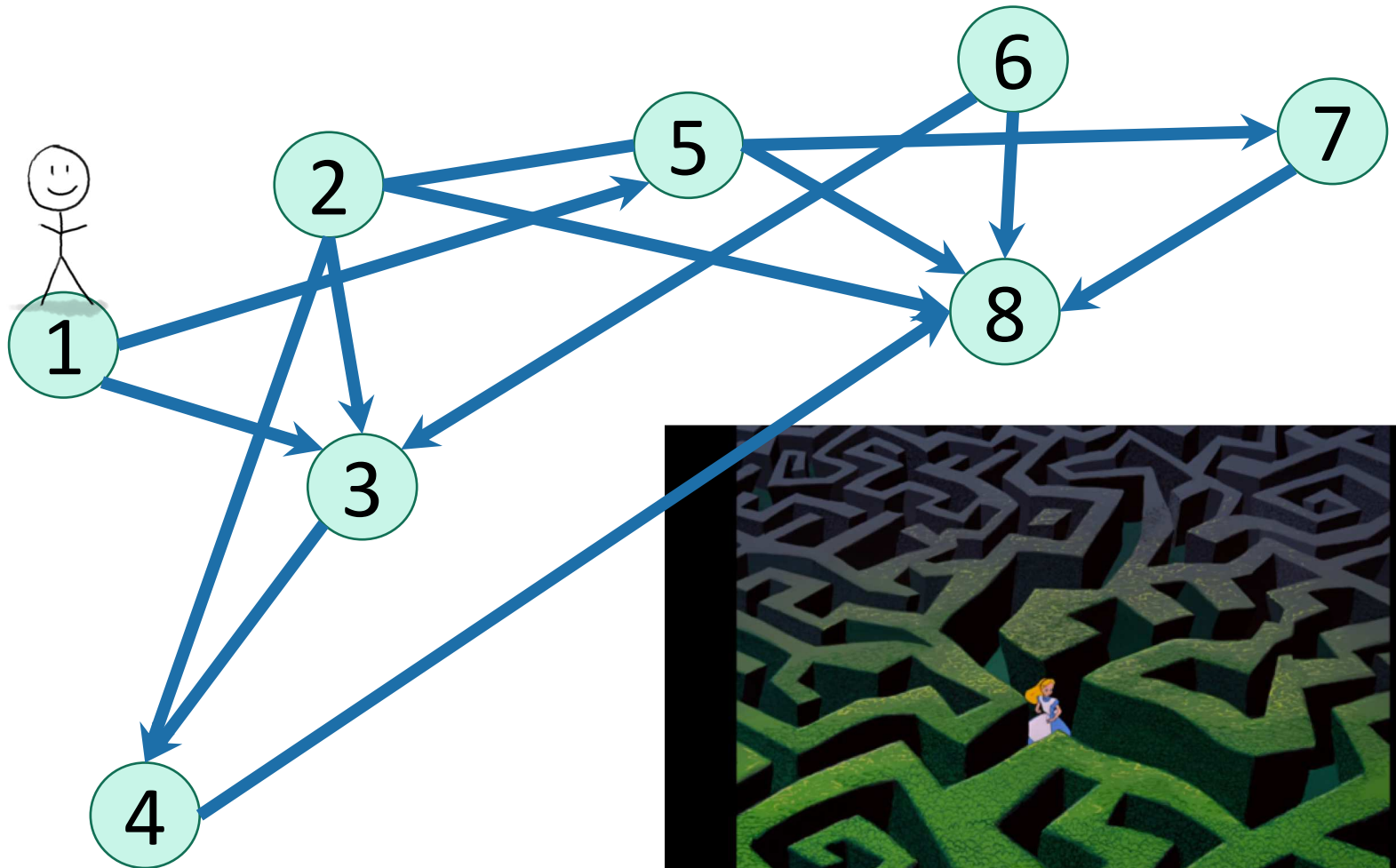


- But first! Let's briefly recap DFS...

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

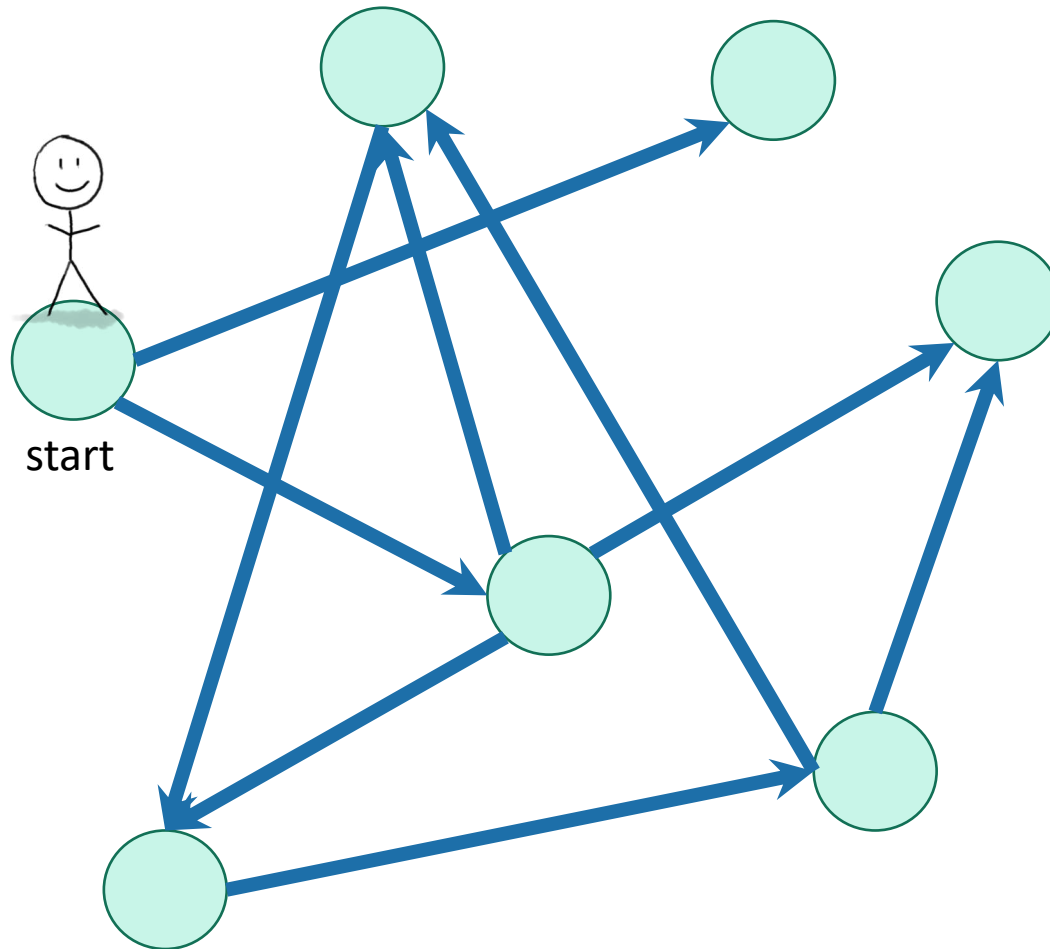
# Recall: DFS




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

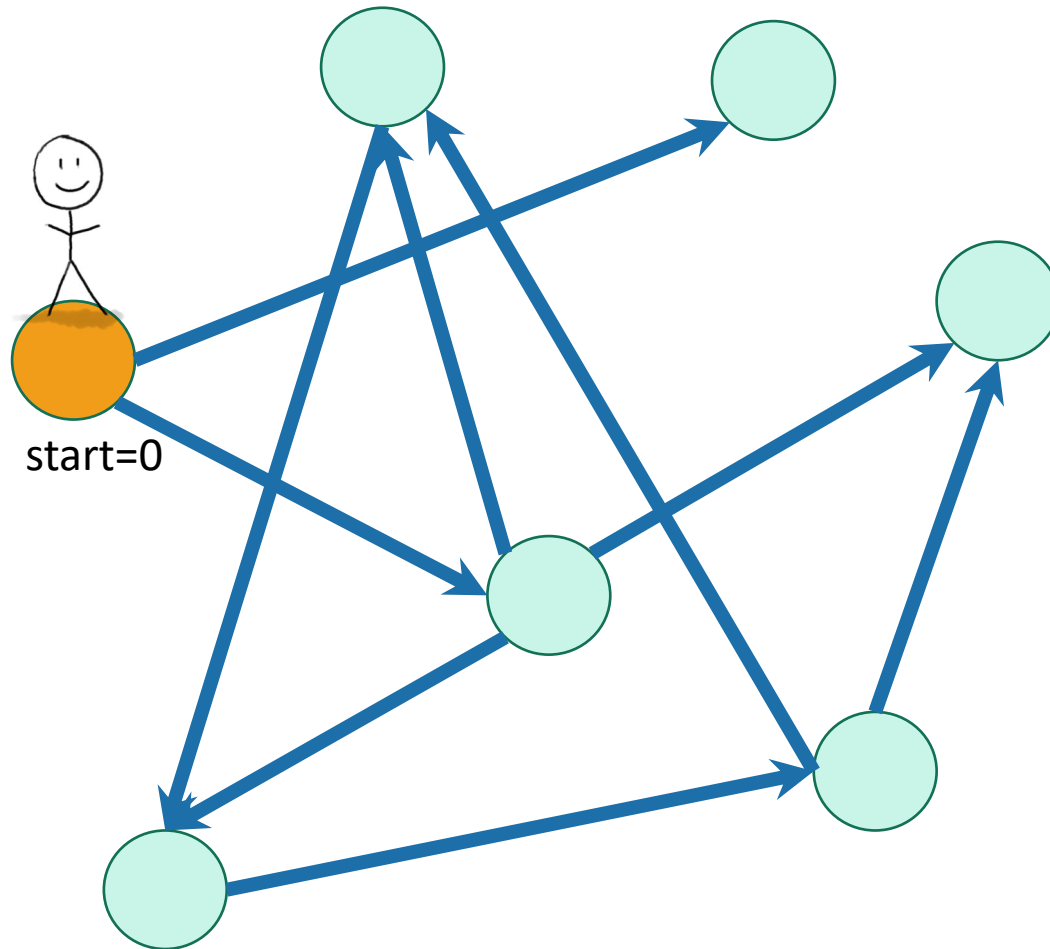





-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.

This is the same picture we had Monday, except I've directed all of the edges. Notice that there **ARE** cycles.

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



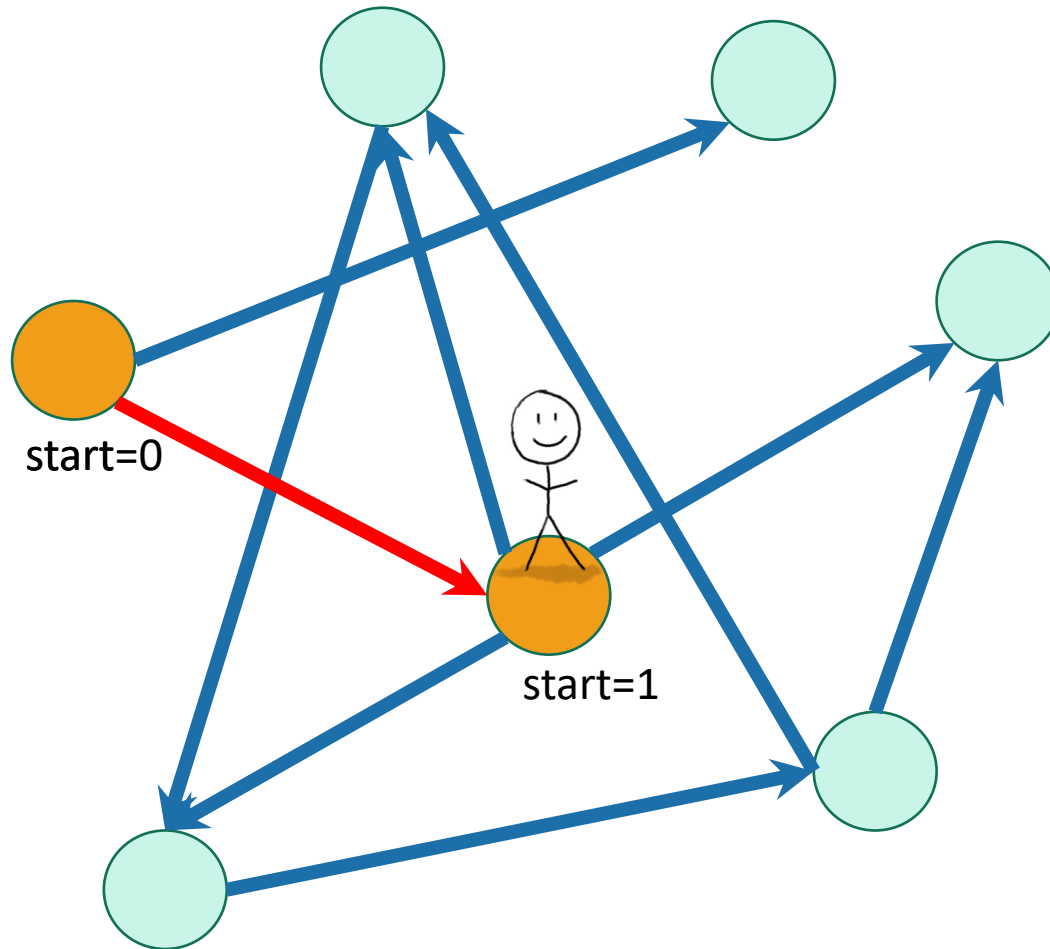
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 17

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



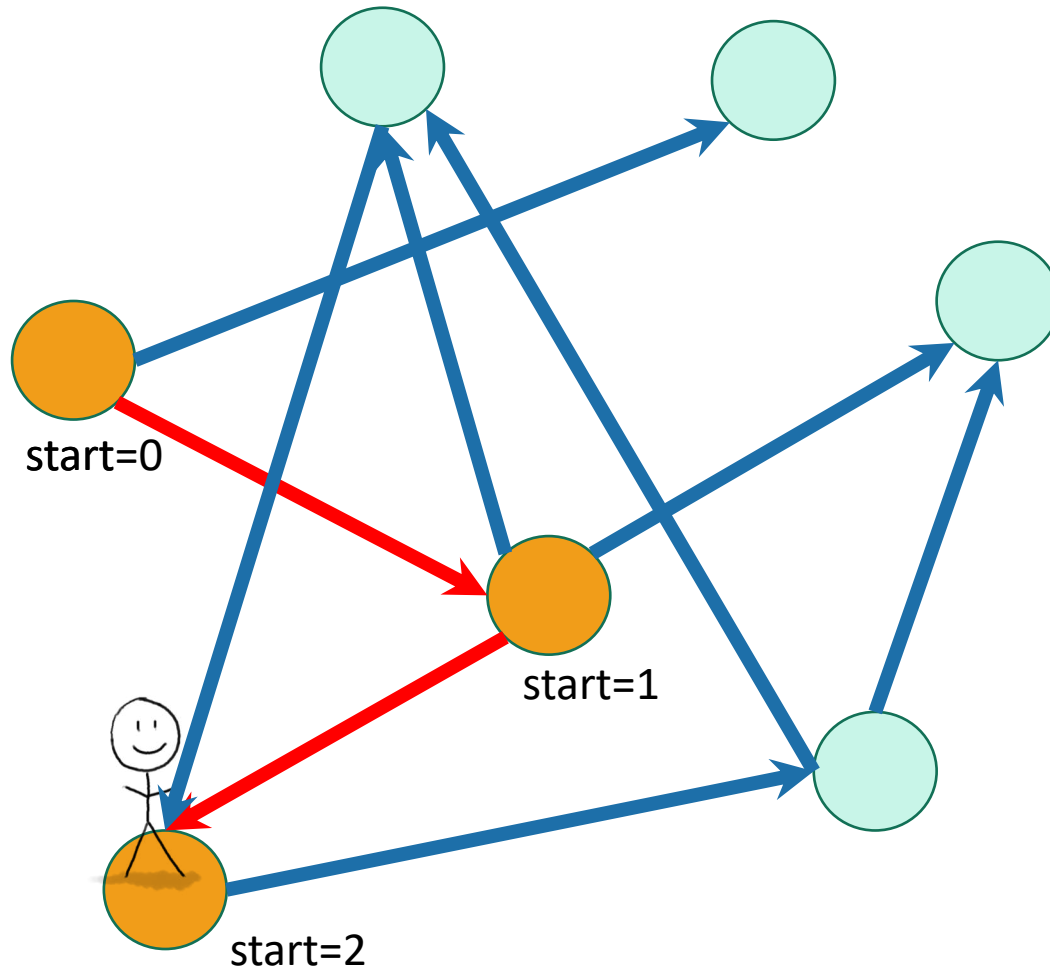
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 18

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



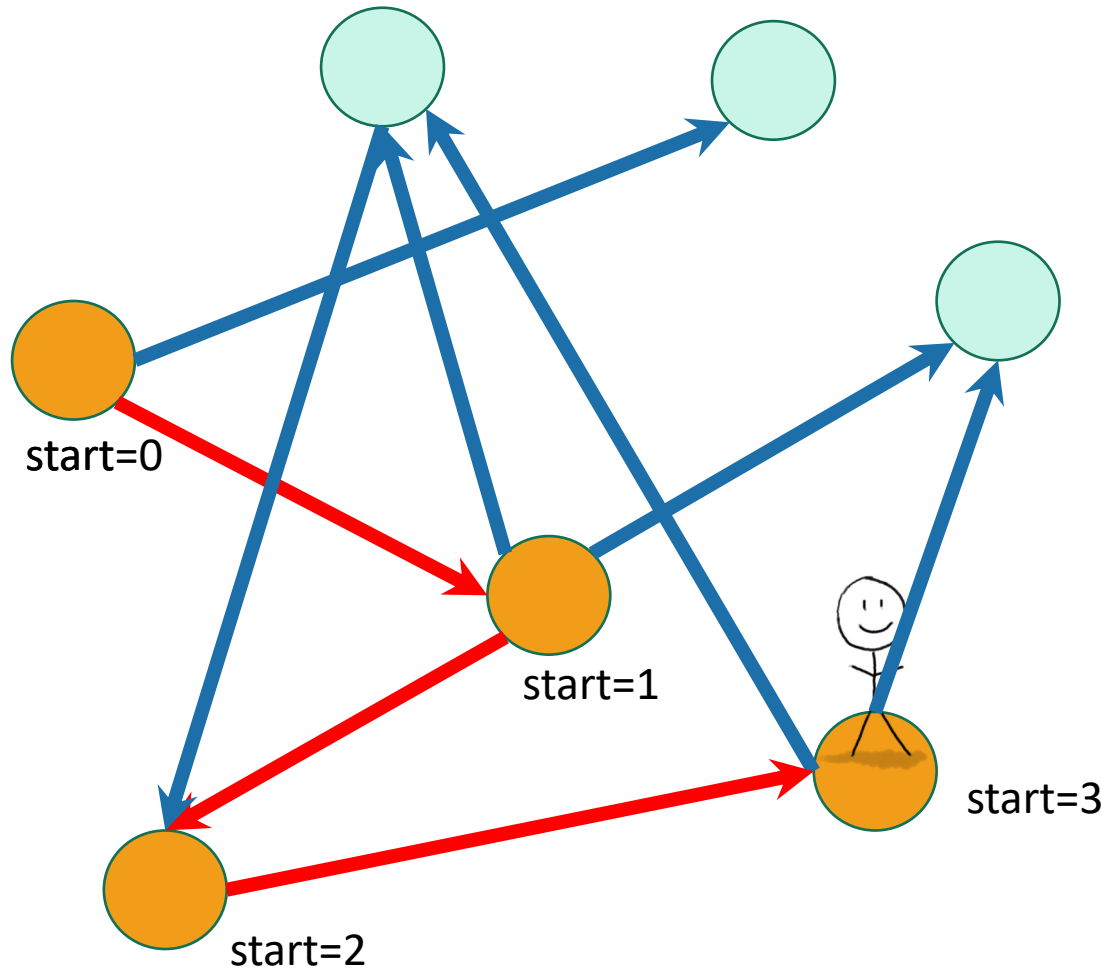
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 19

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



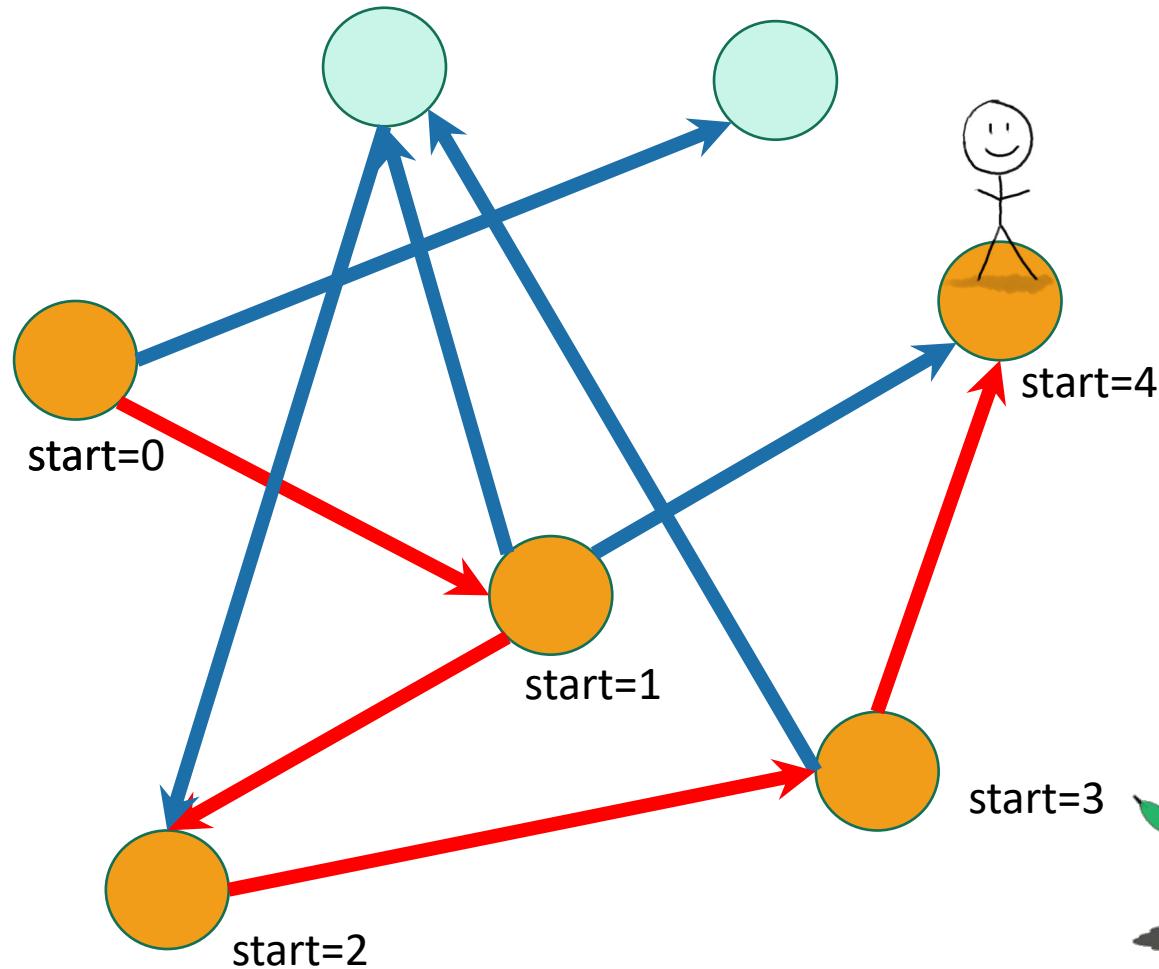
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 20

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



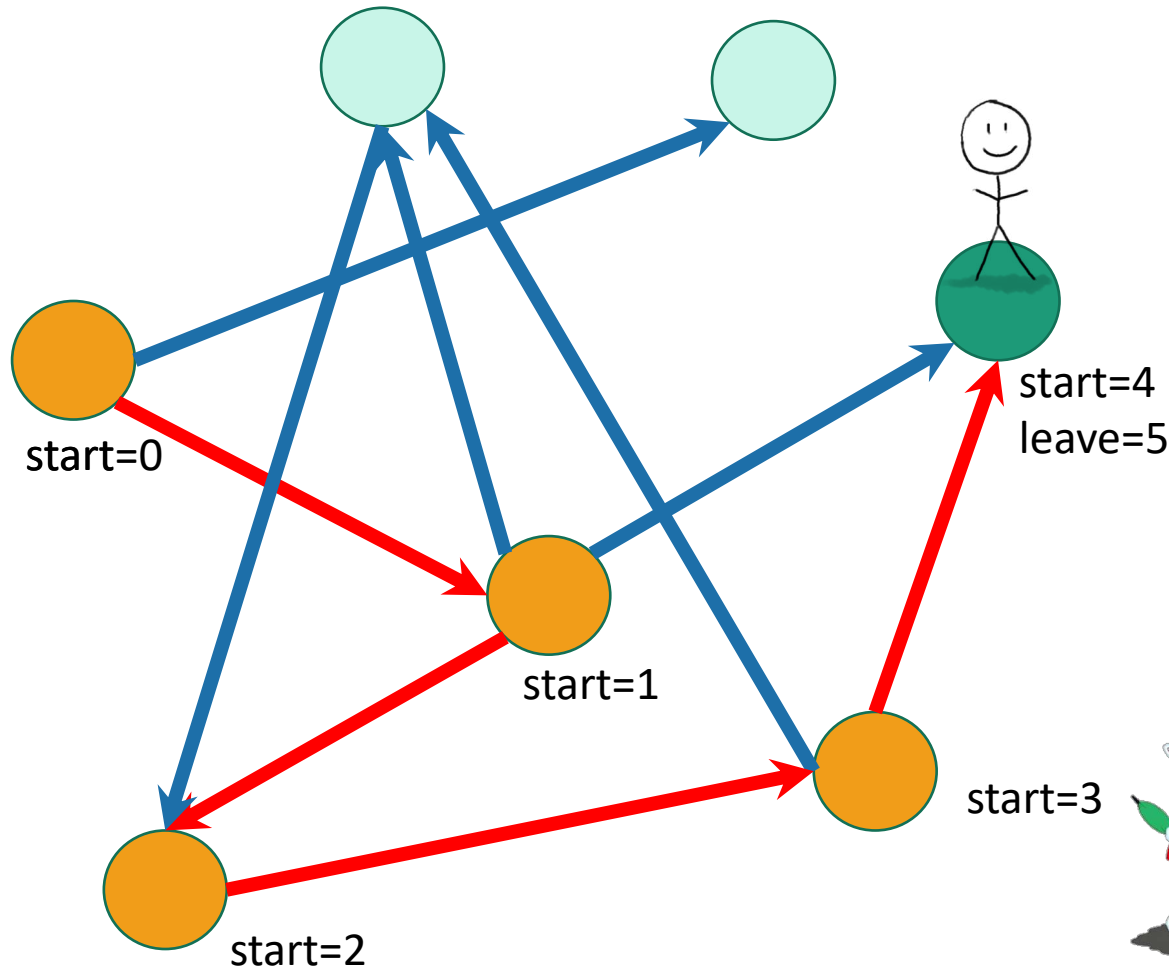
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 21

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



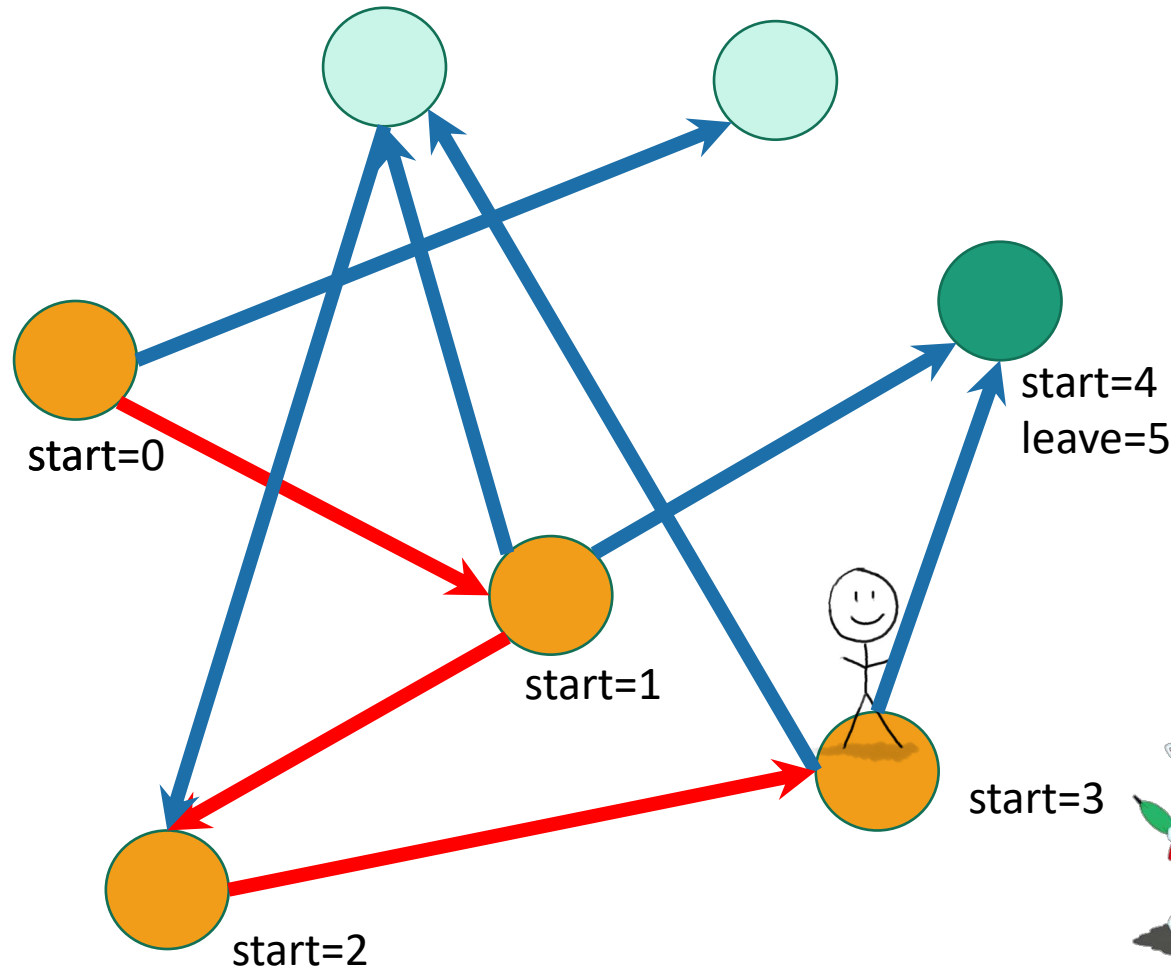
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 22

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



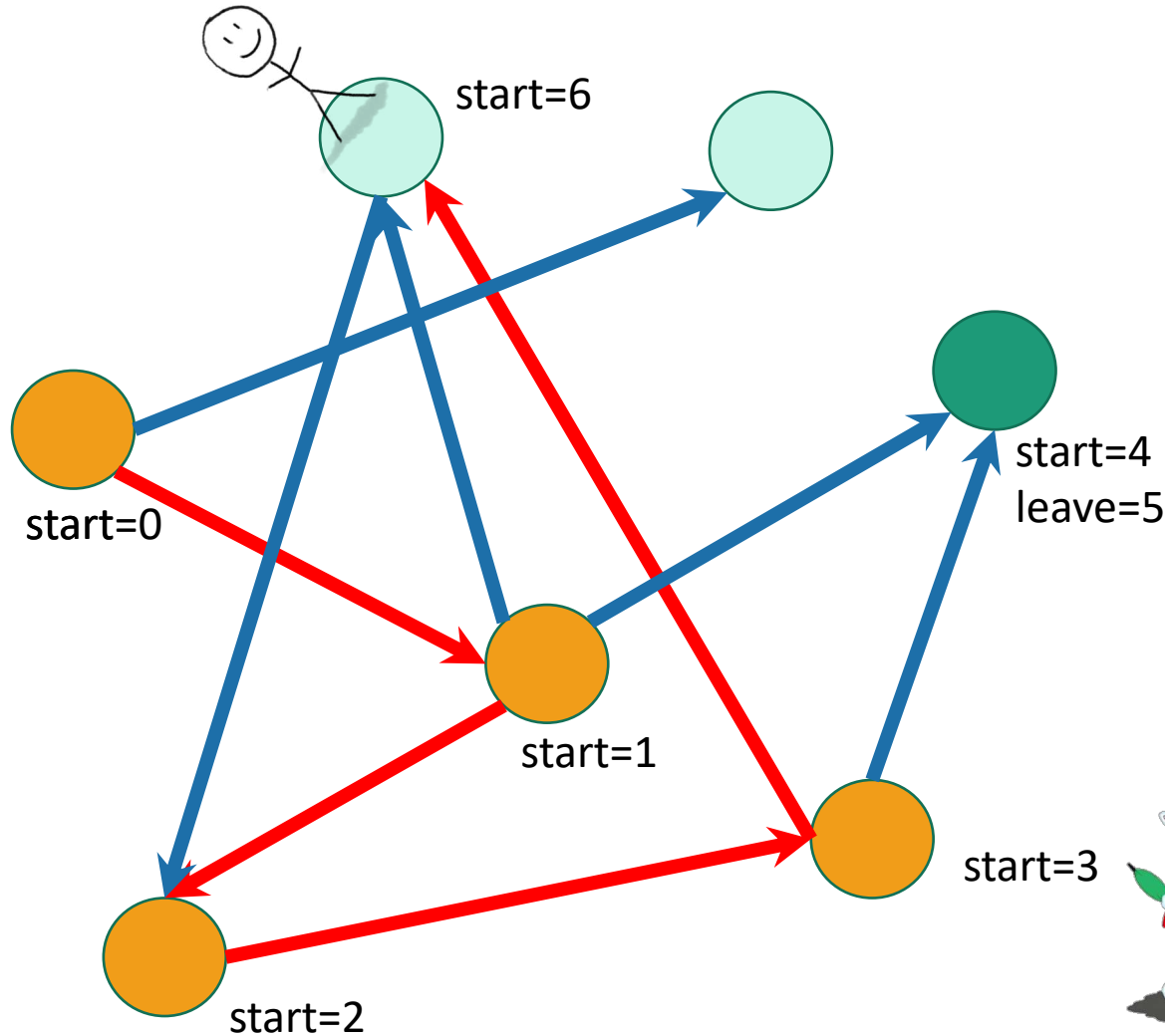
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node.

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



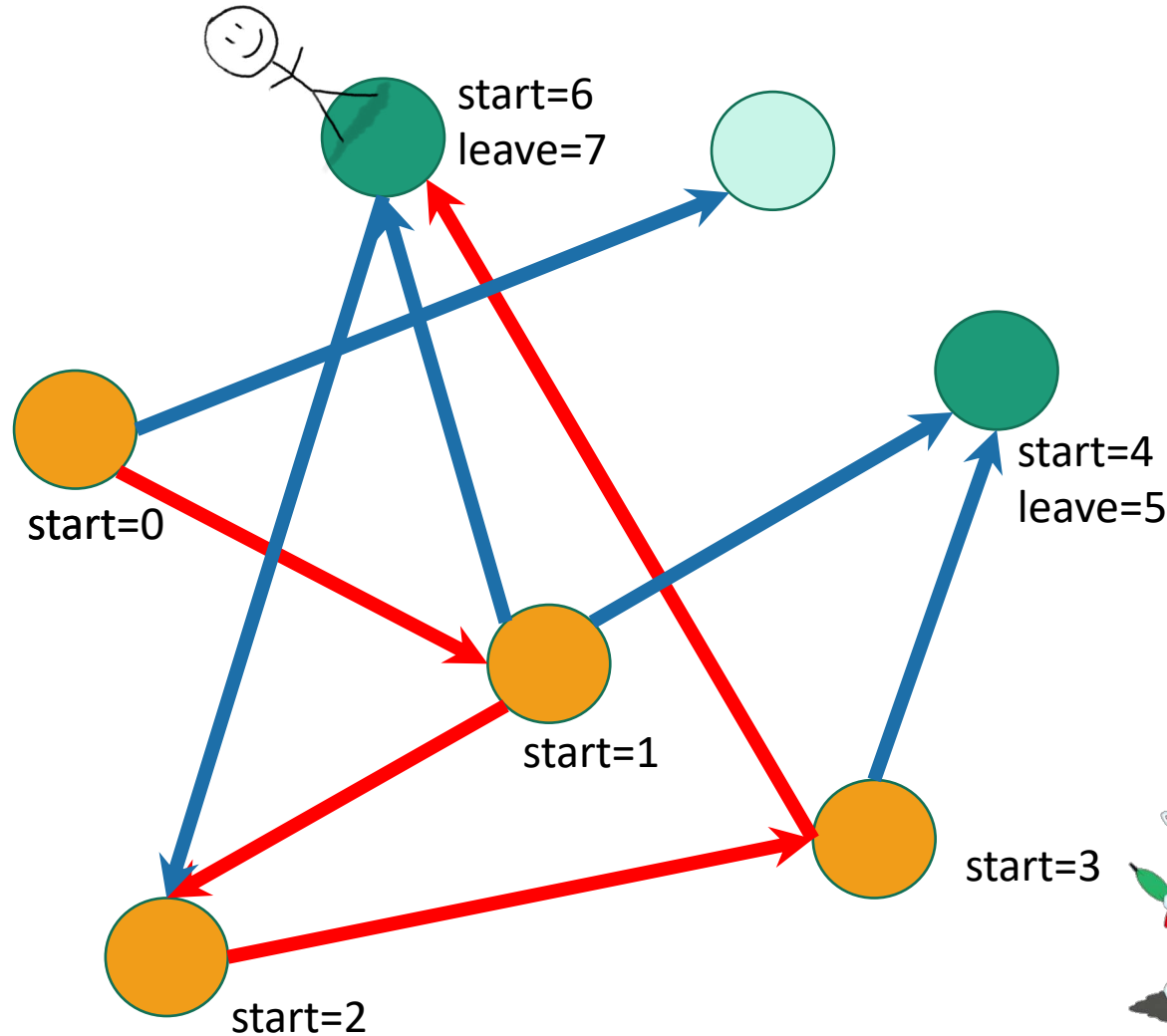
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node.

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



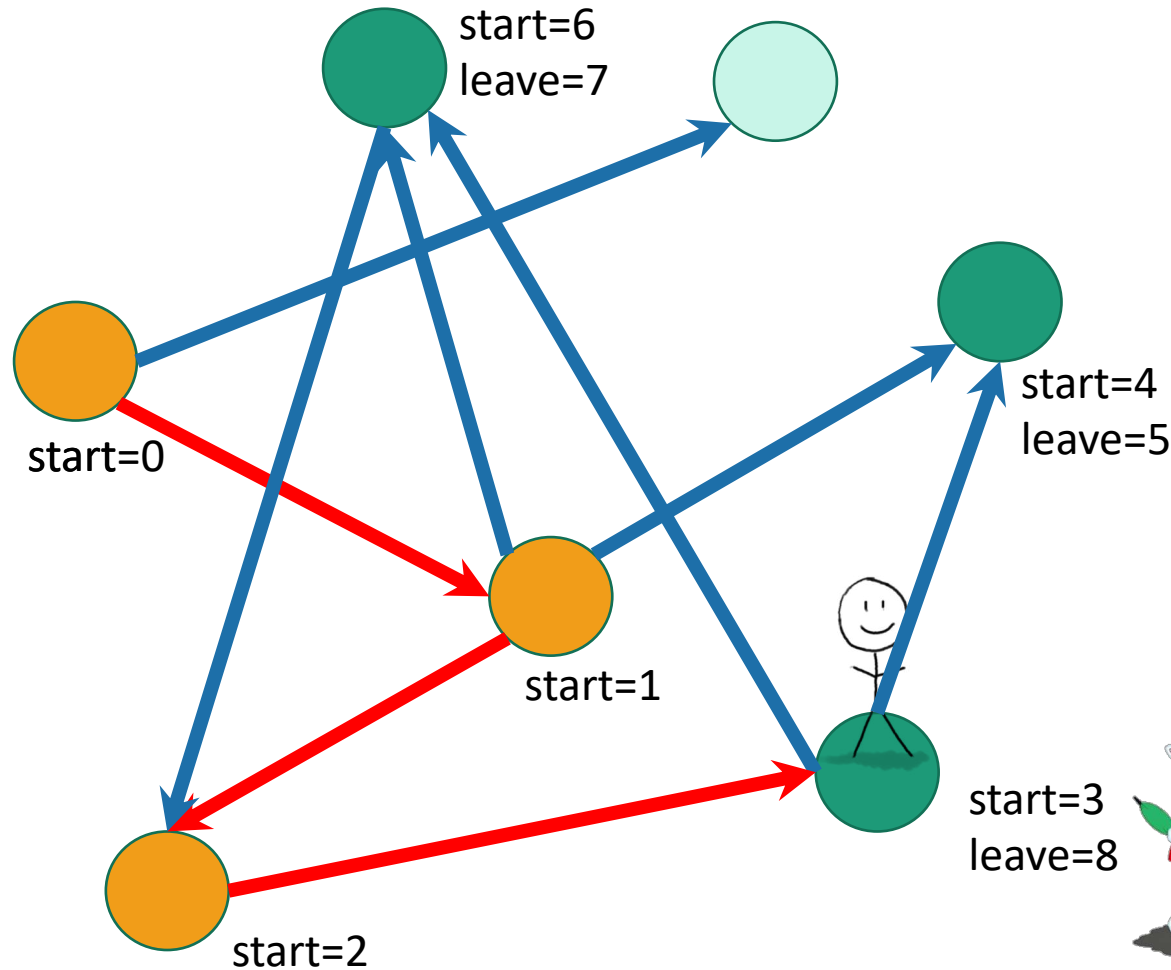
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node.

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



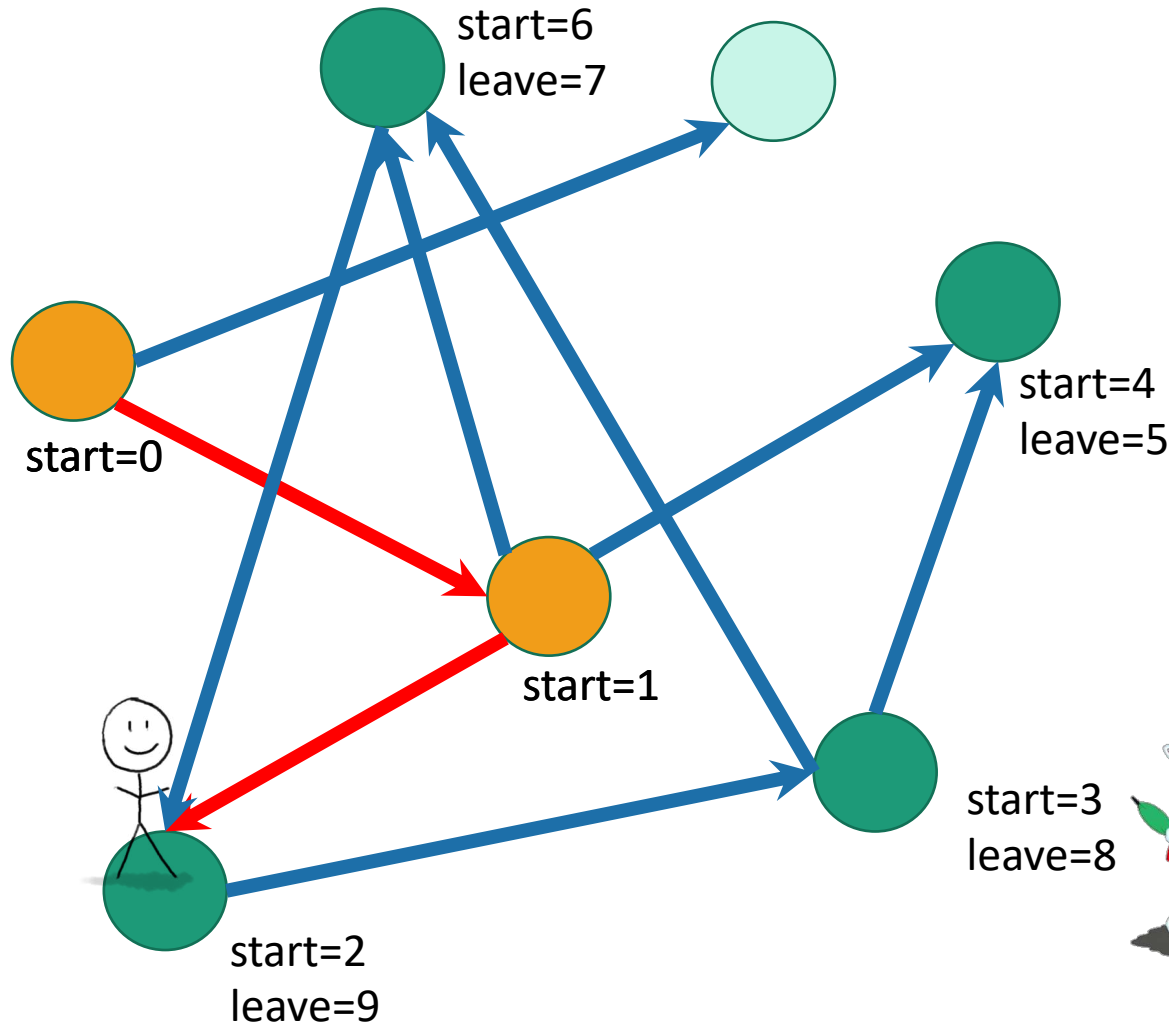
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 26

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



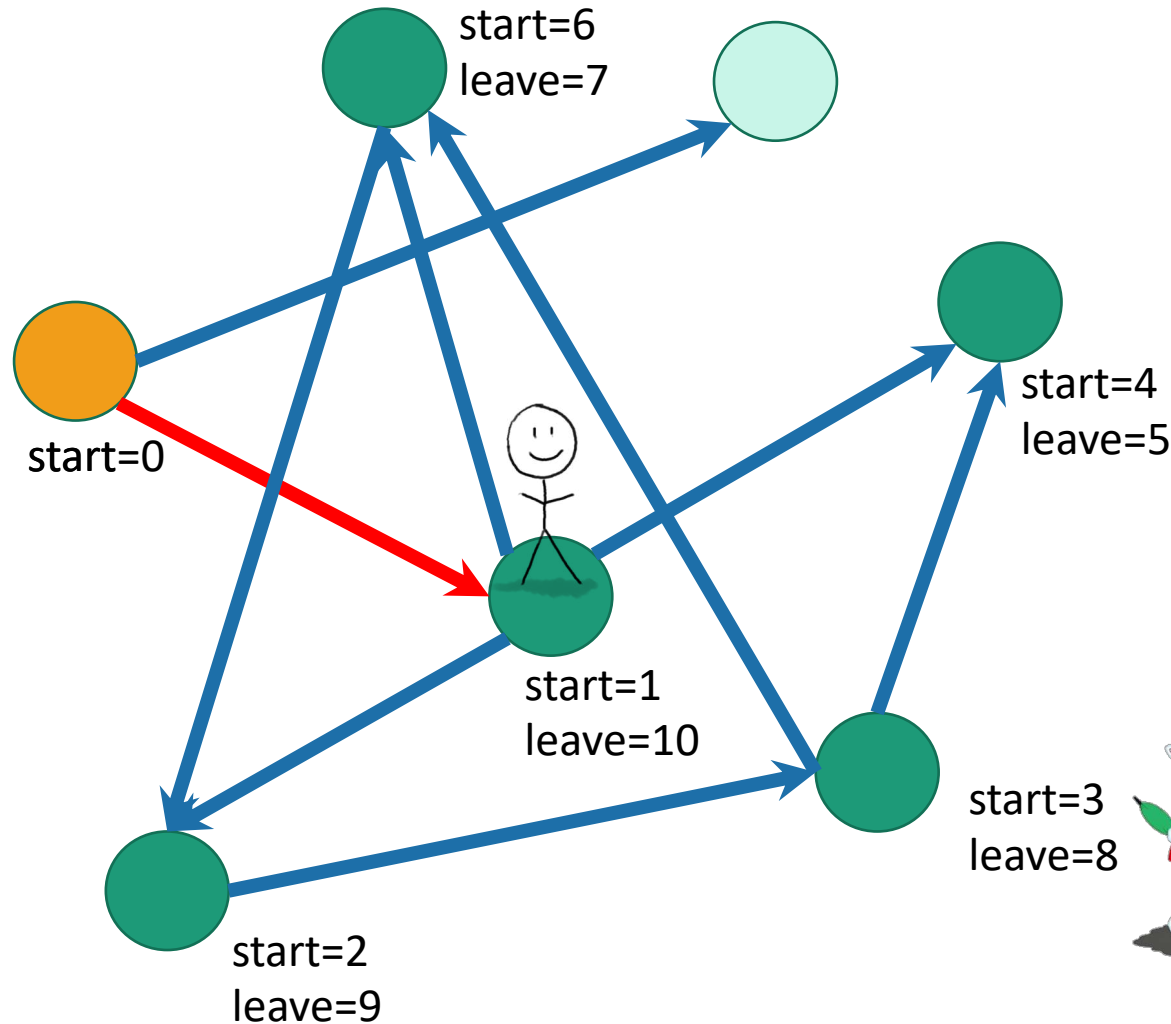
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 27

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



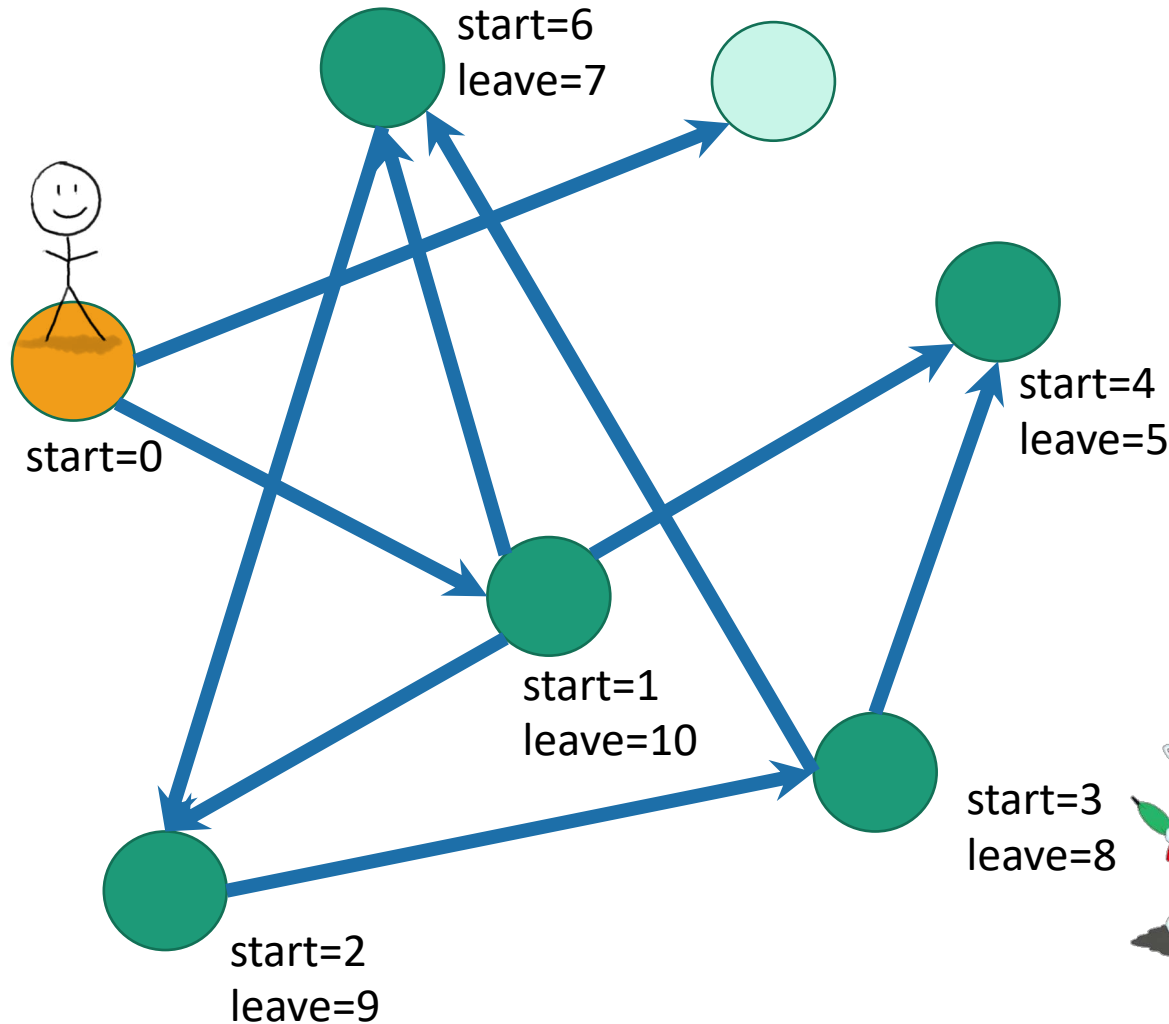
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 28

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



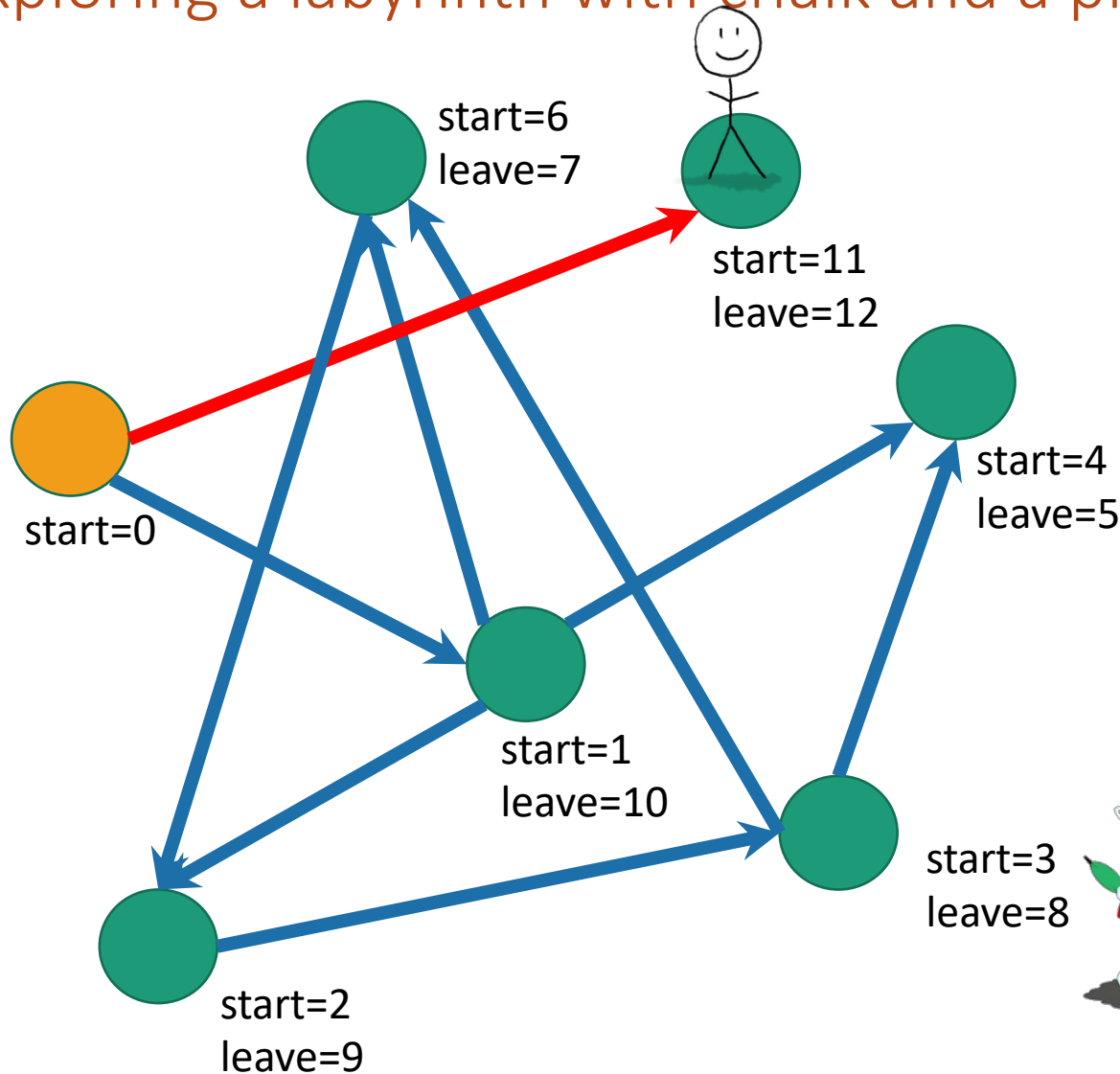
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






Recall we also keep track of **start** and **finish** times for every node. 29

# Depth First Search

Exploring a labyrinth with chalk and a piece of string



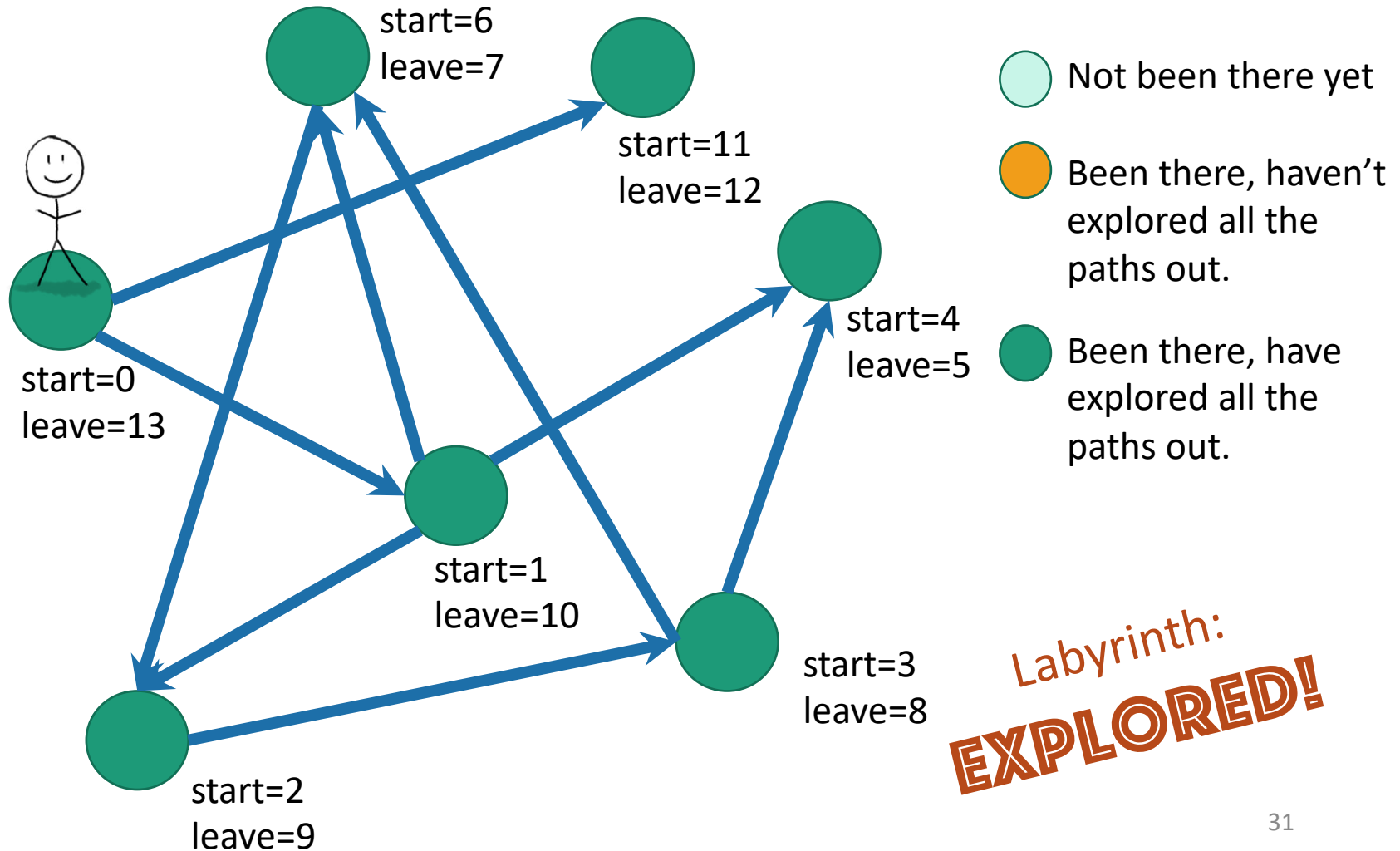
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.



Recall we also keep track of **start** and **finish** times for every node.

# Depth First Search

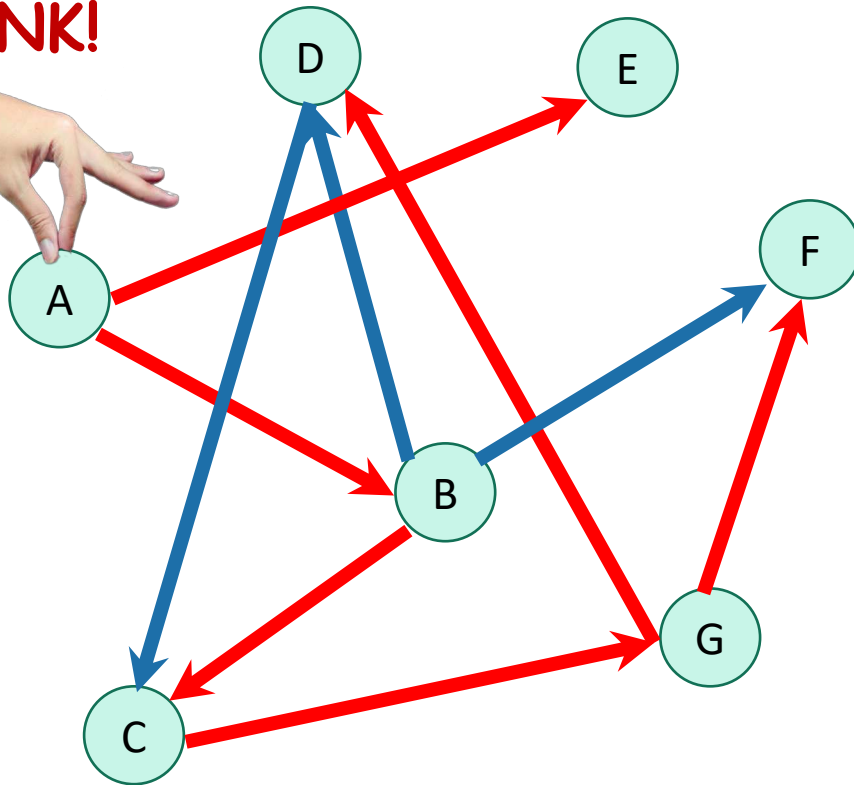
Exploring a labyrinth with chalk and a piece of string



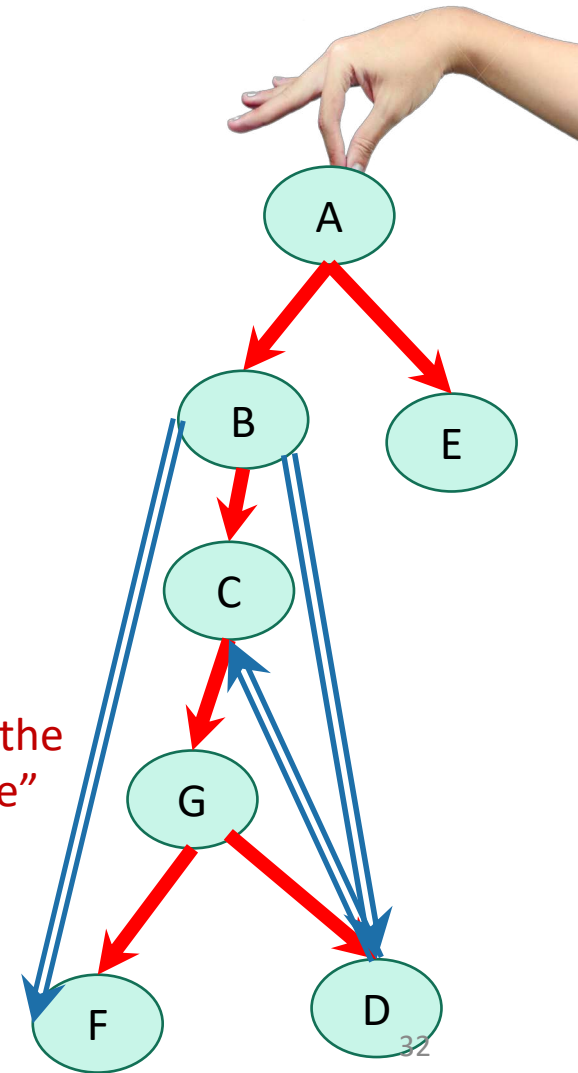
# Depth first search

implicitly creates a tree on everything you can reach

**YOINK!**

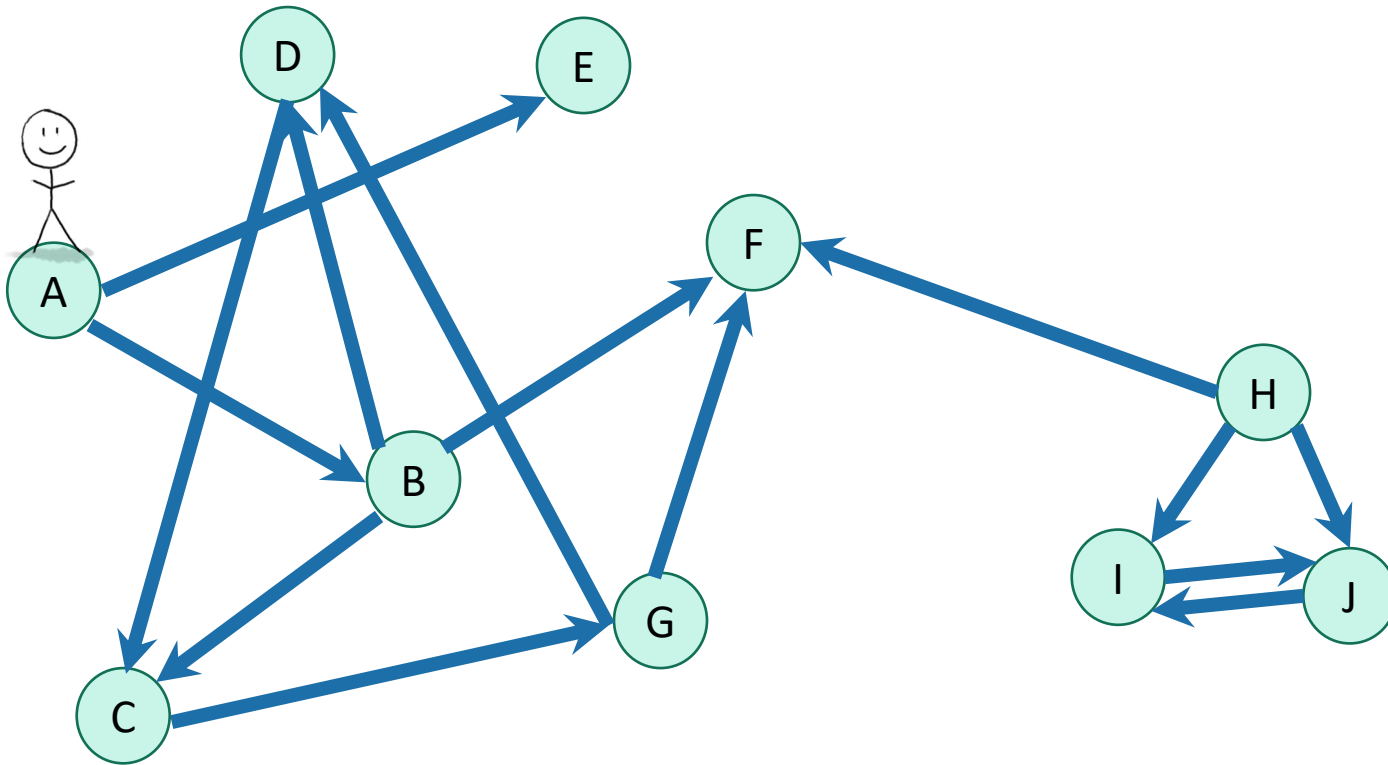


Call this the  
"DFS tree"



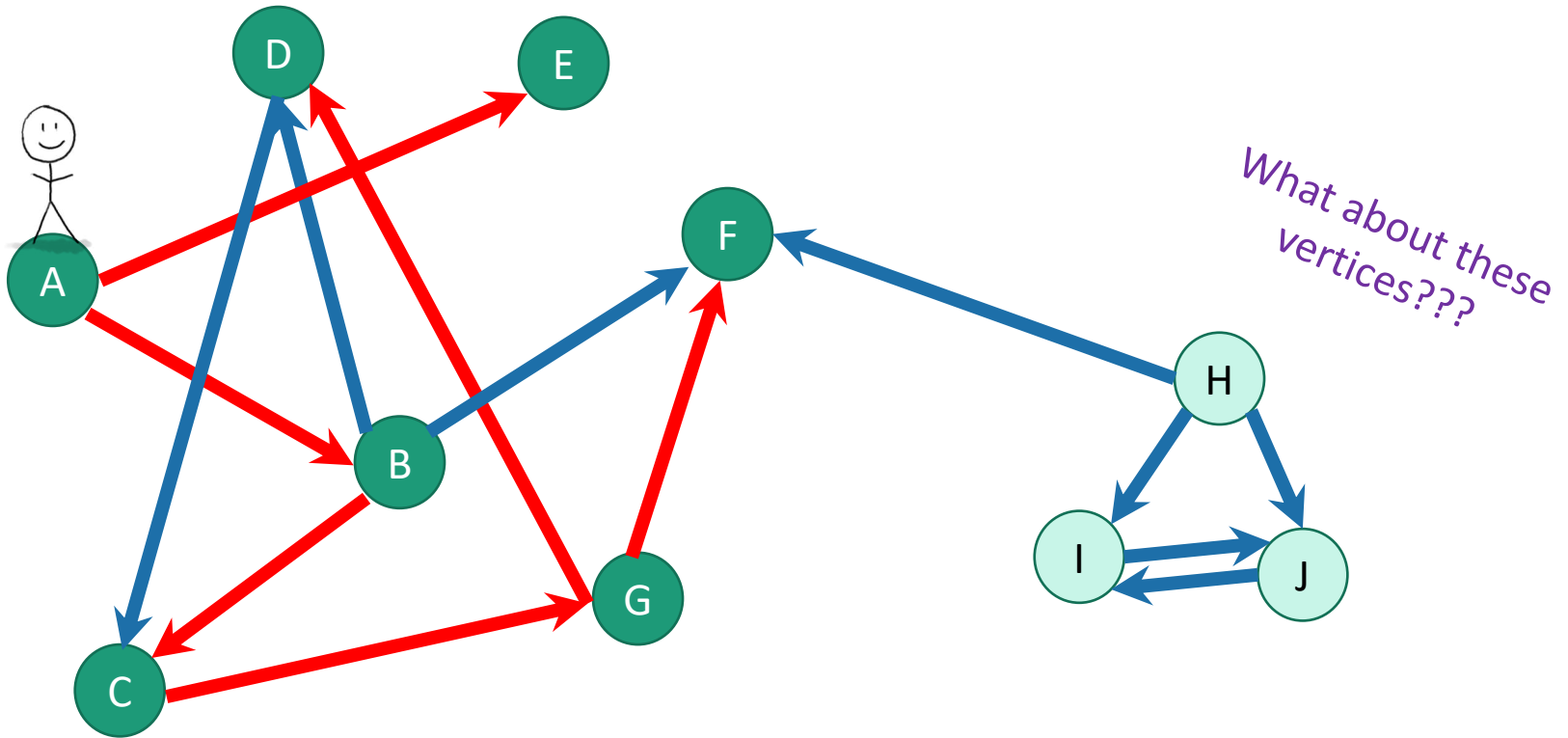
# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



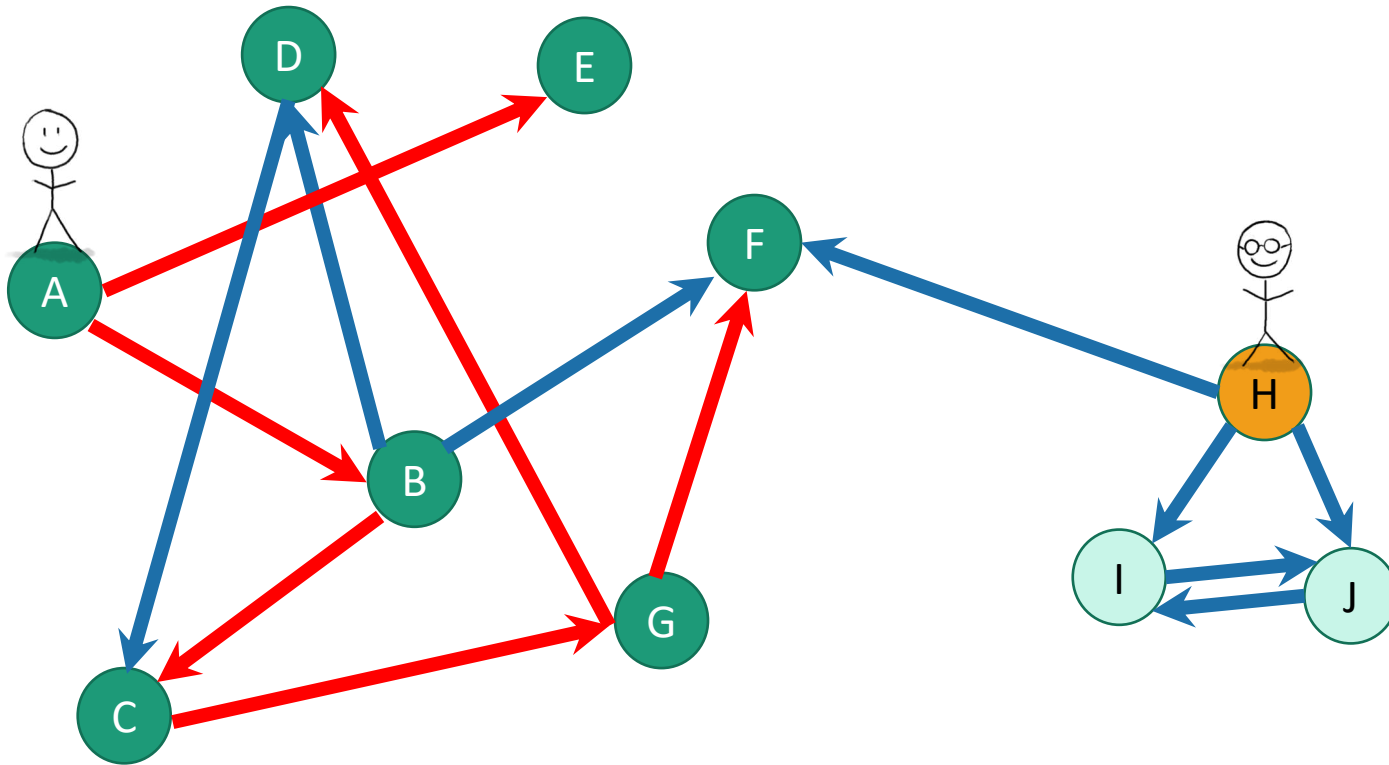
# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



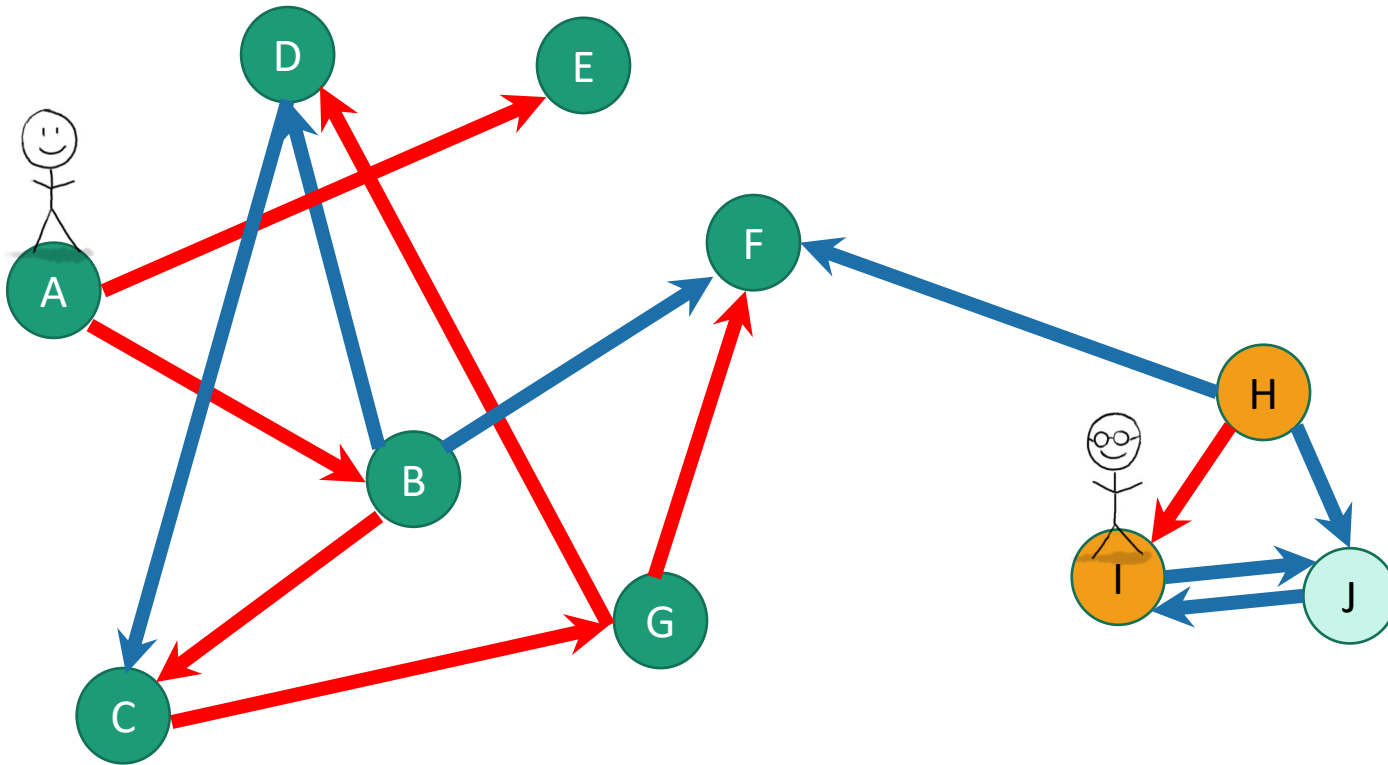
# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



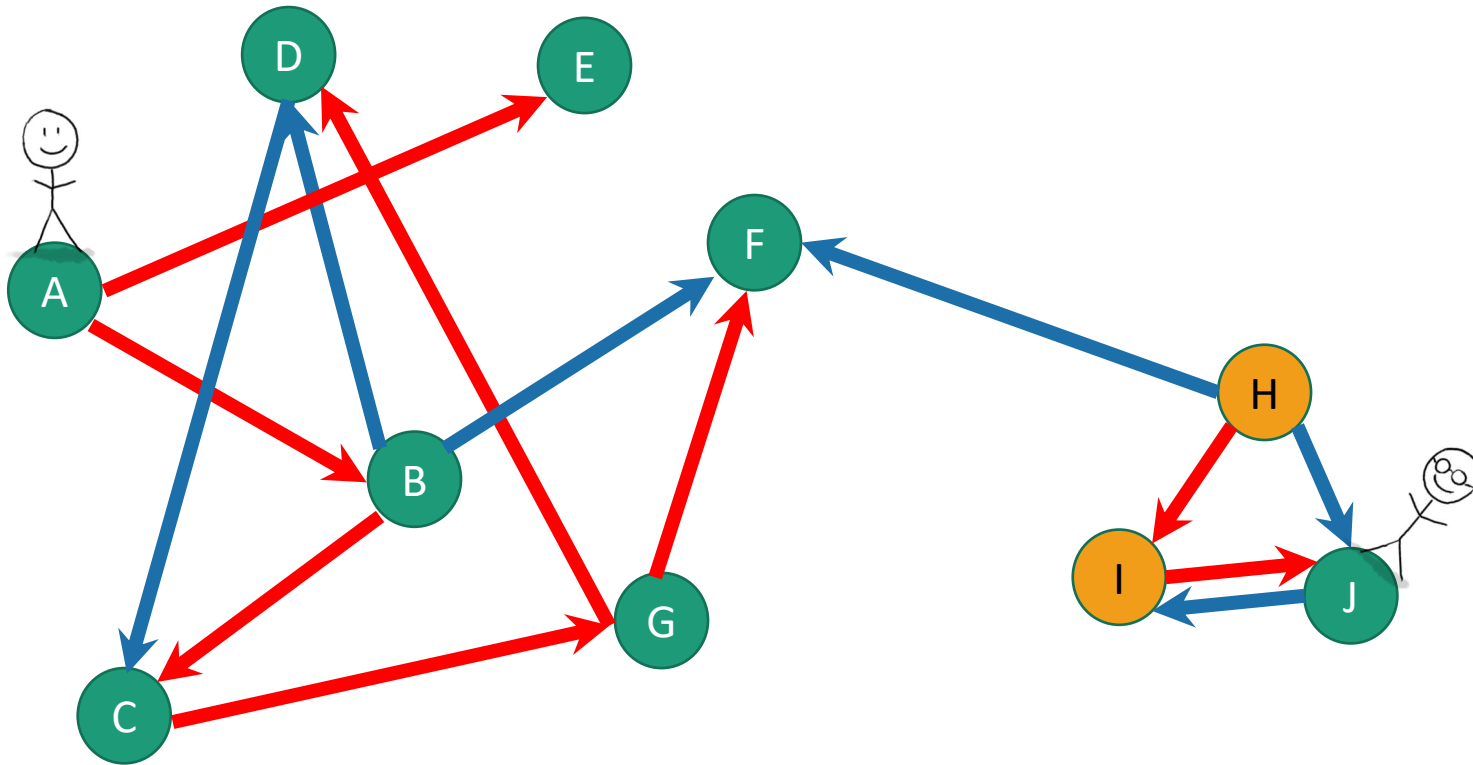
# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



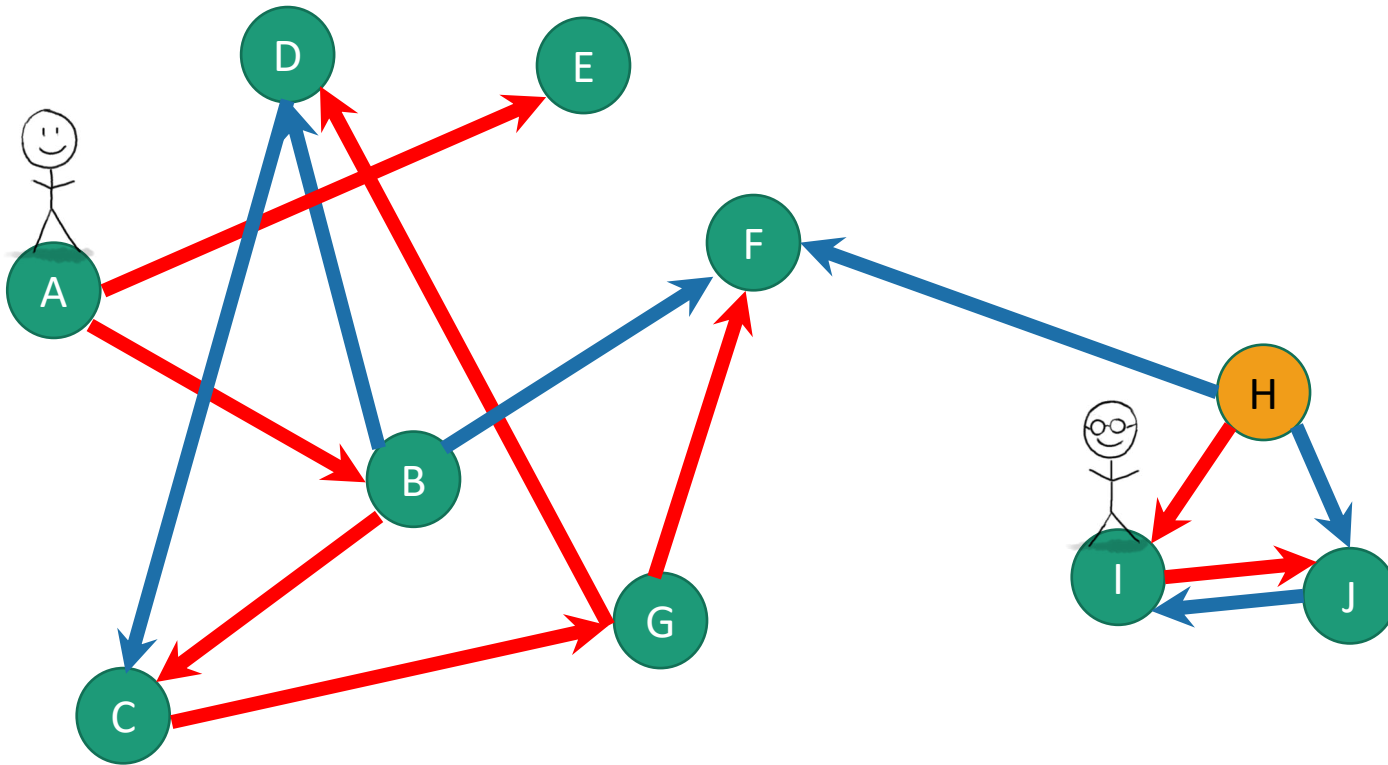
# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



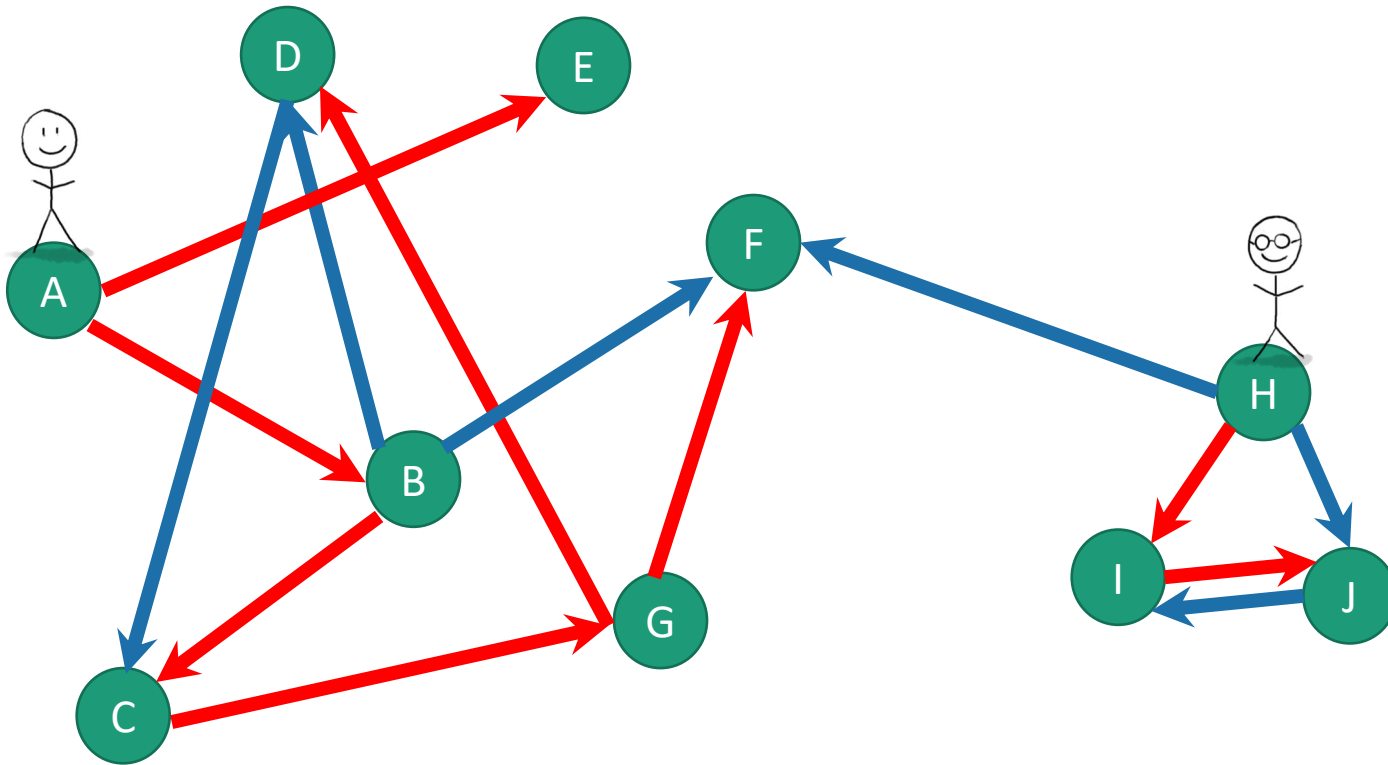
# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



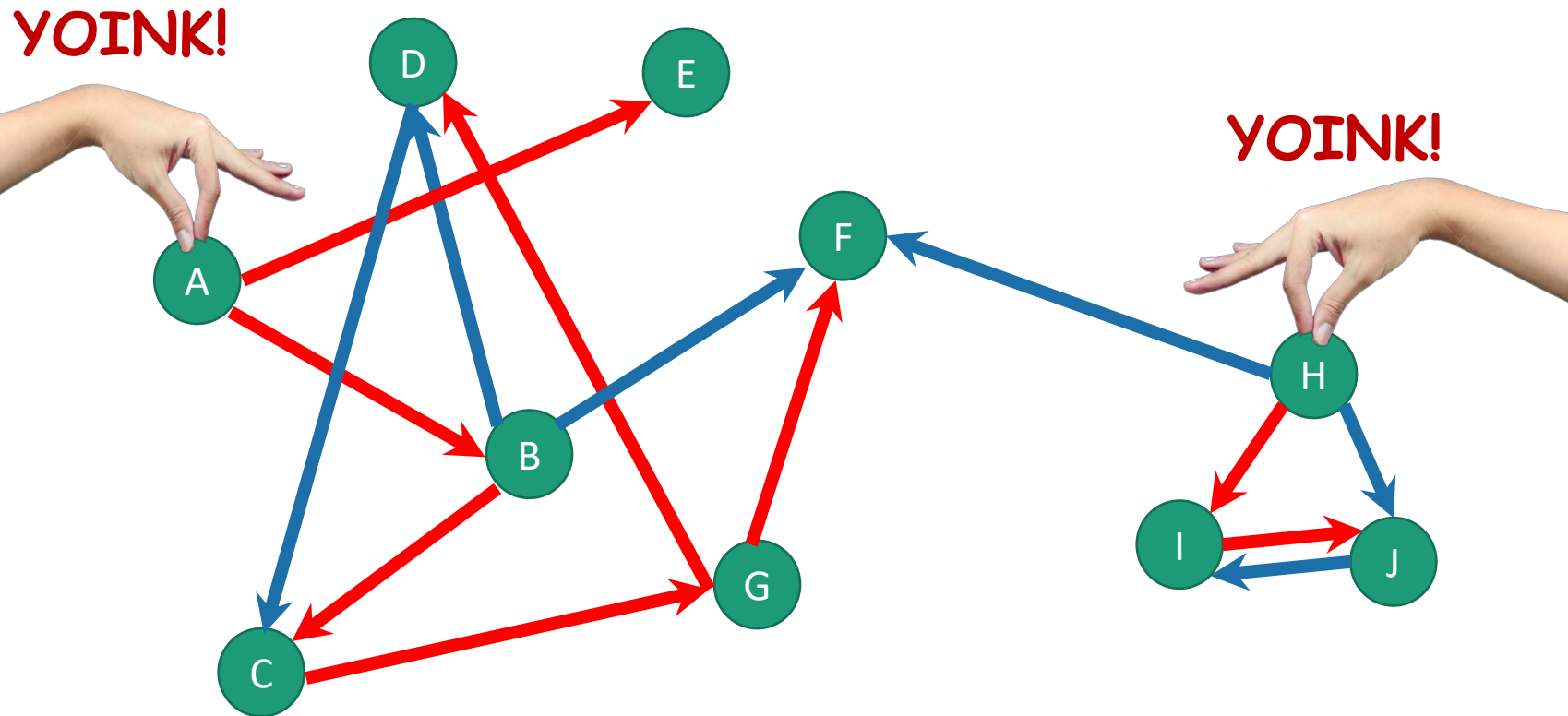
# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



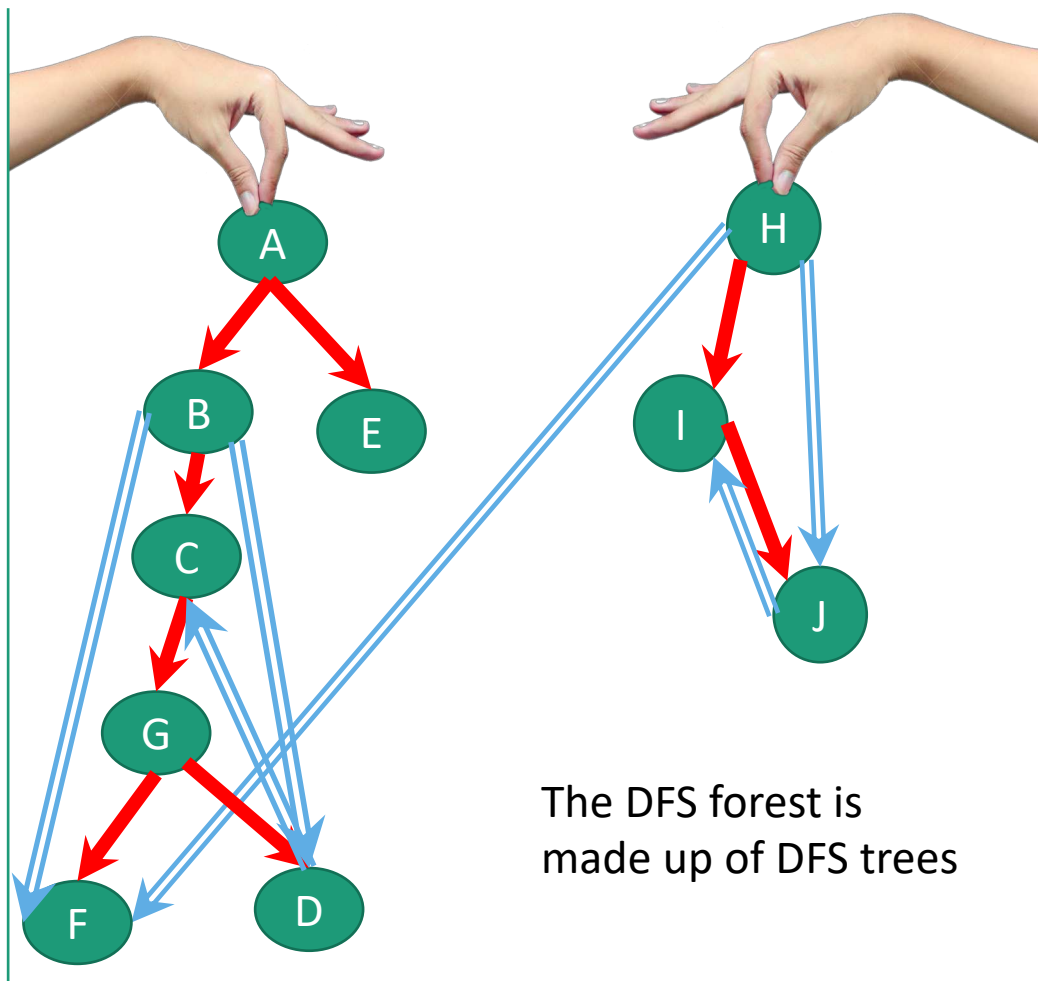
# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



# When you can't reach everything

- Run DFS repeatedly to get a **depth-first forest**



# Recall:

(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:



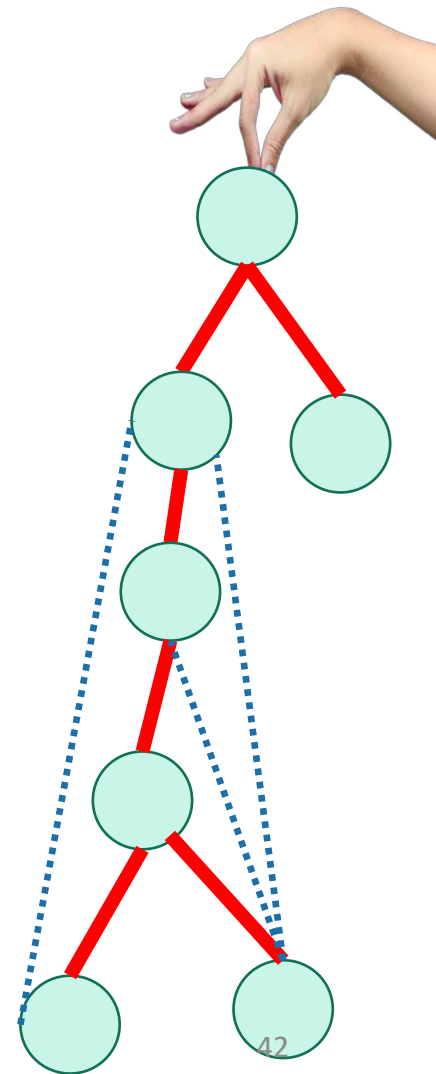
- If neither are descendants of each other:



(or the other way around)

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

DFS tree

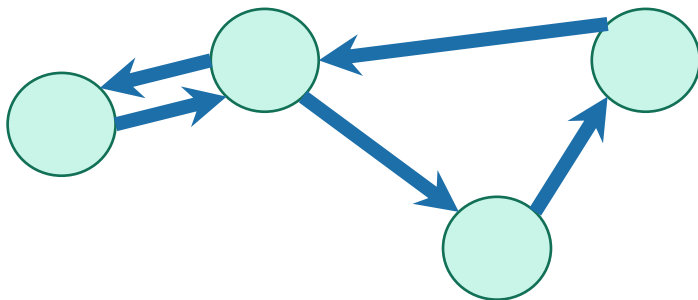


Enough of review

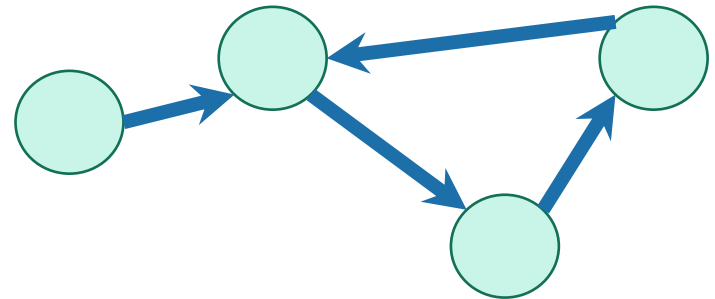
Strongly connected components

# 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  $v$ .



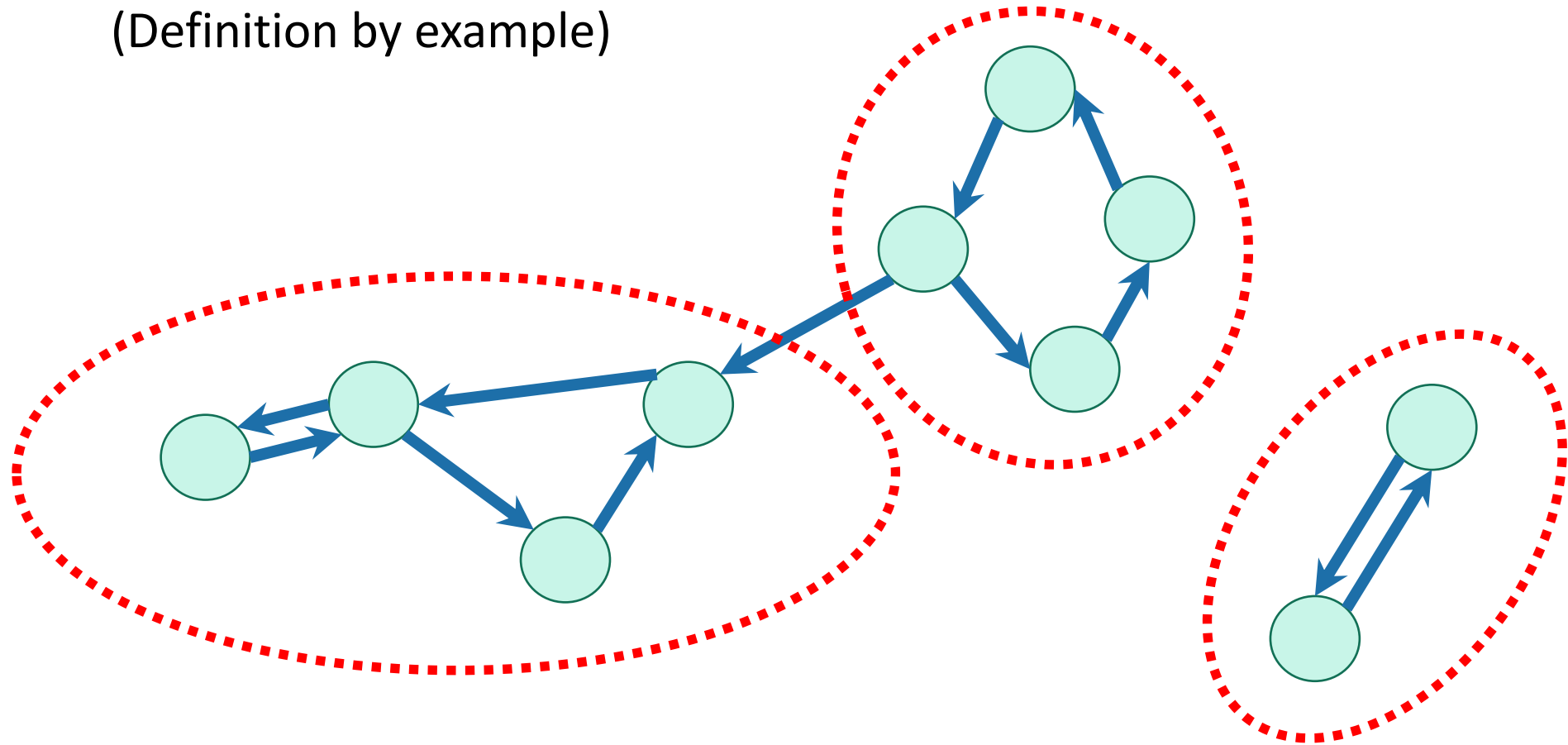
strongly connected



not strongly connected

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

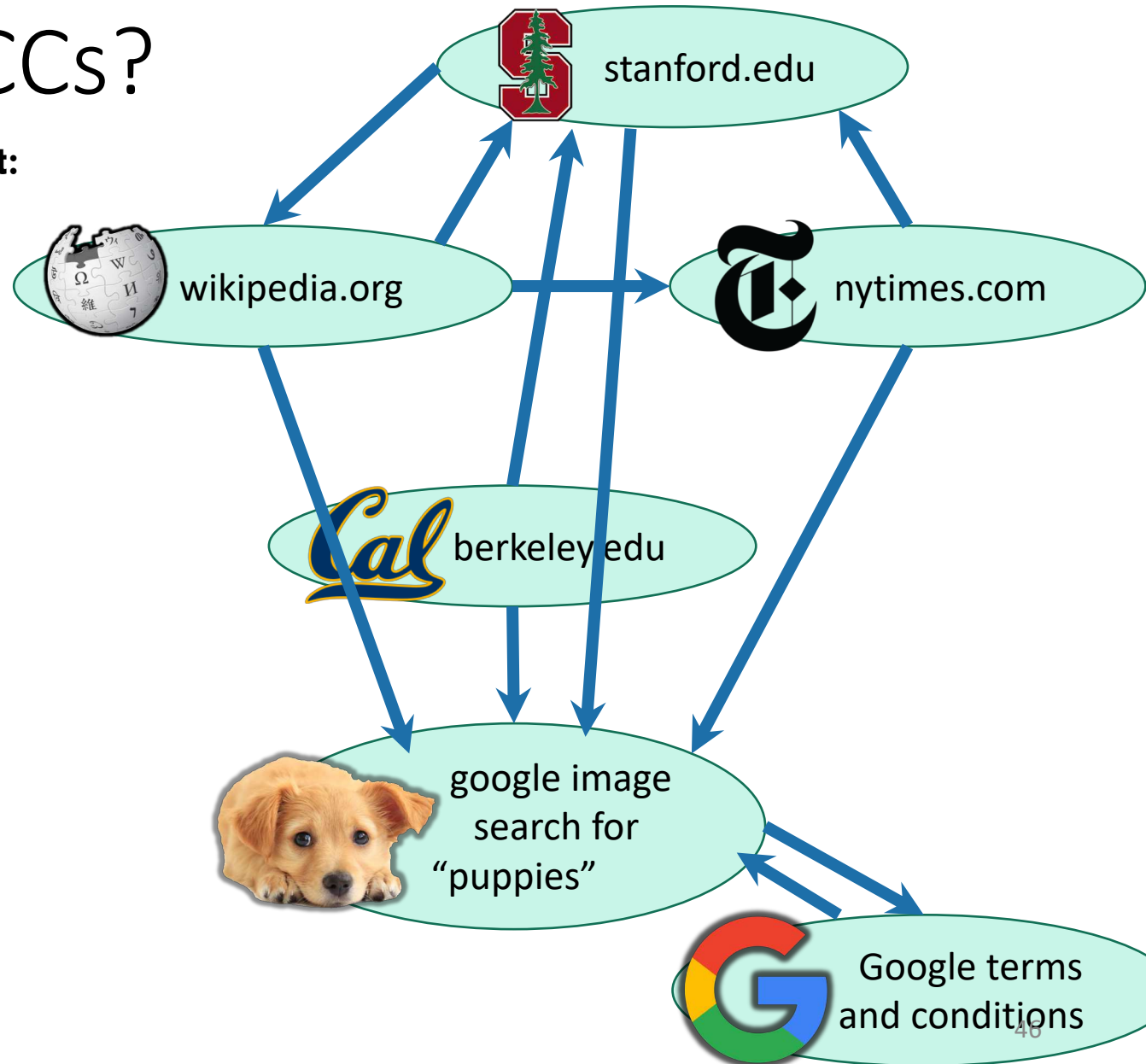
(Definition by example)



Definition by definition: The SCCs are the equivalence classes under the “are mutually reachable” equivalence relation.

# Why do we care about SCCs?

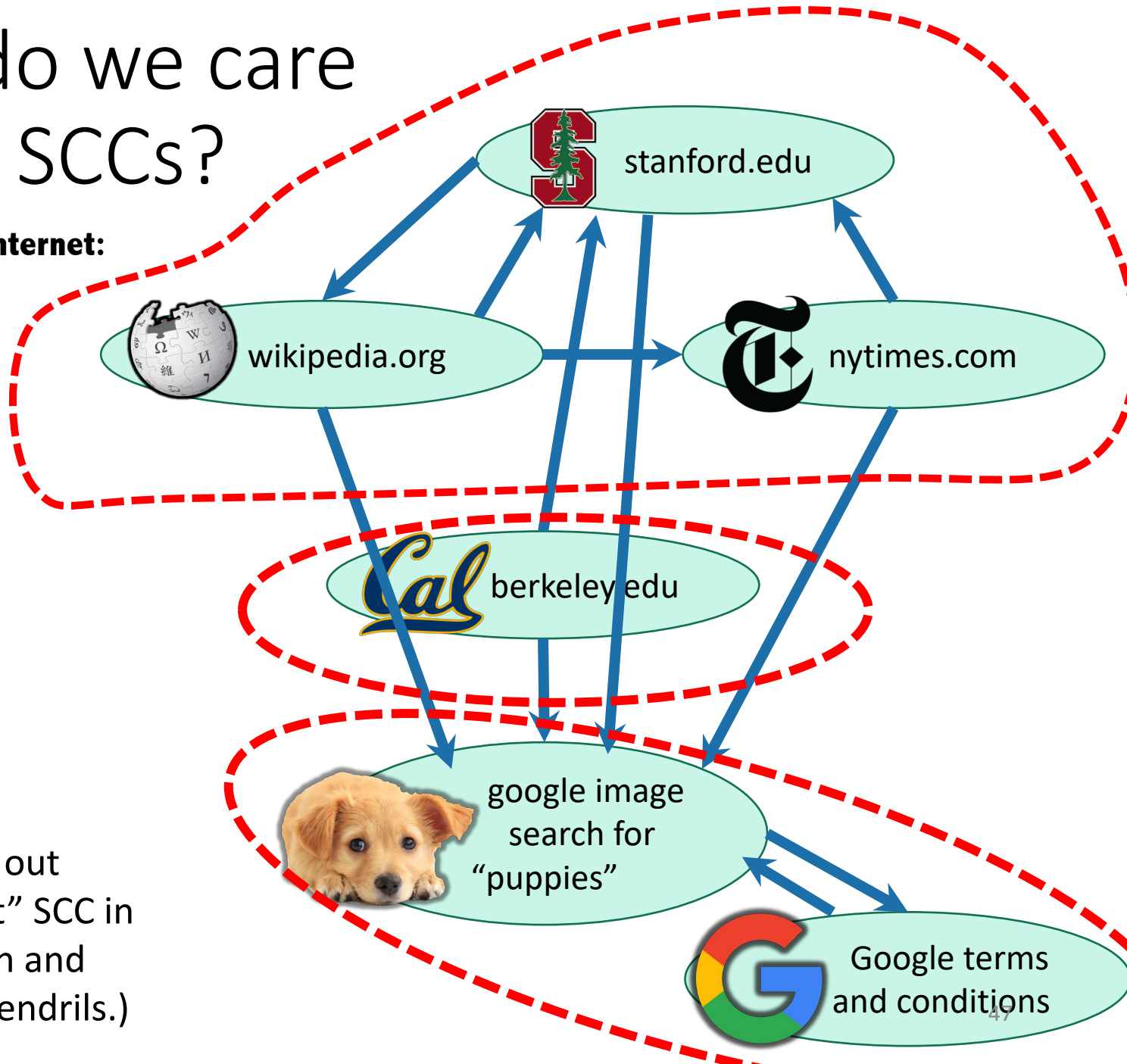
Consider the internet:



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:



(In real life, turns out there's one "giant" SCC in the internet graph and then a bunch of tendrils.)

# Why do we care about SCCs?

- 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 know an economist 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:

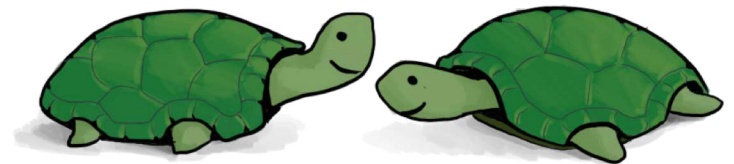
- Consider all possible decompositions and check.

## Try 2:

- Something like...
  - Run DFS a bunch to find out which  $u$ 's and  $v$ 's belong in the same SCC
  - Aggregate that information to figure out the SCCs

Come up with a straightforward way to use DFS to find SCCs. What's the running time?  
More than  $n^2$  or less than  $n^2$ ?

Think: 2 minutes.  
Pair+Share: 1 minute



# One straightforward solution

- SCCs = [ ]
- For each u:
  - Run DFS from u
  - For each vertex v that u can reach:
    - If v is in an SCC we've already found:
      - Run DFS from v to see if you can reach u
      - If so, add u to v's SCC
      - Break
    - If we didn't break, create a new SCC which just contains u.

This will not be our final solution so don't worry too much about it...



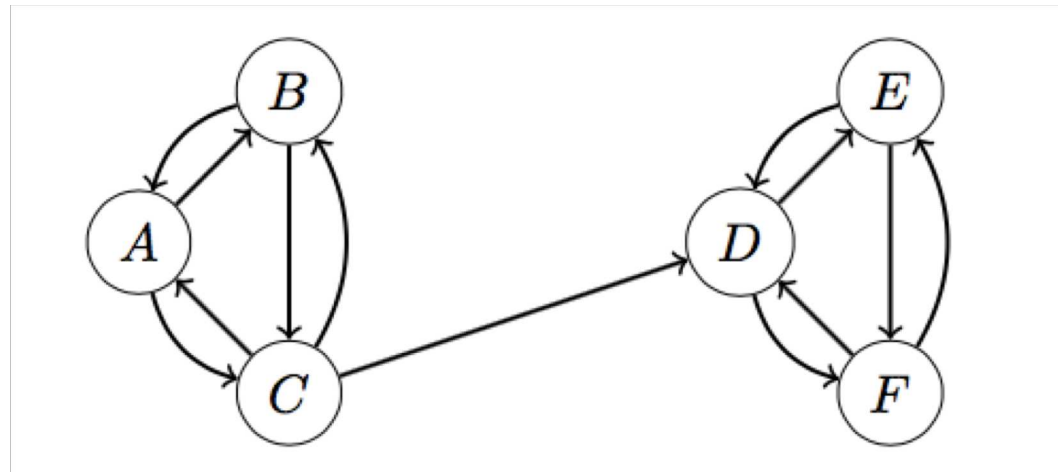
Running time AT LEAST  $\Omega(n^2)$ , no matter how smart you are about implementing the rest of it...

# Today

- We will see how to find strongly connected components in time  $O(n+m)$
- !!!!!

# Pre-Lecture exercise

- Run DFS starting at D:

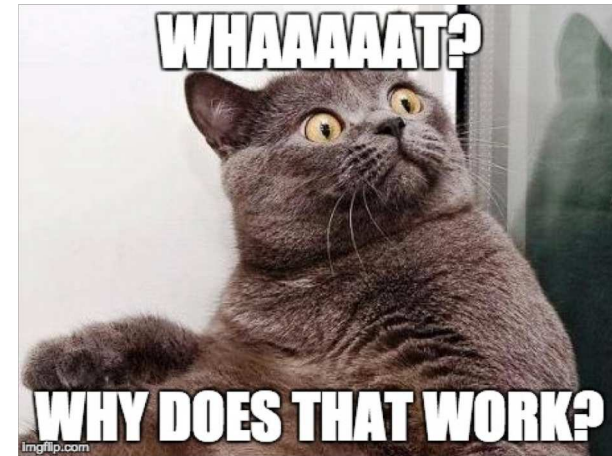


- That will identify SCCs...
- Issues:
  - How do we know where to start DFS?
  - It wouldn't have found the SCCs if we started from A.

# 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)
```

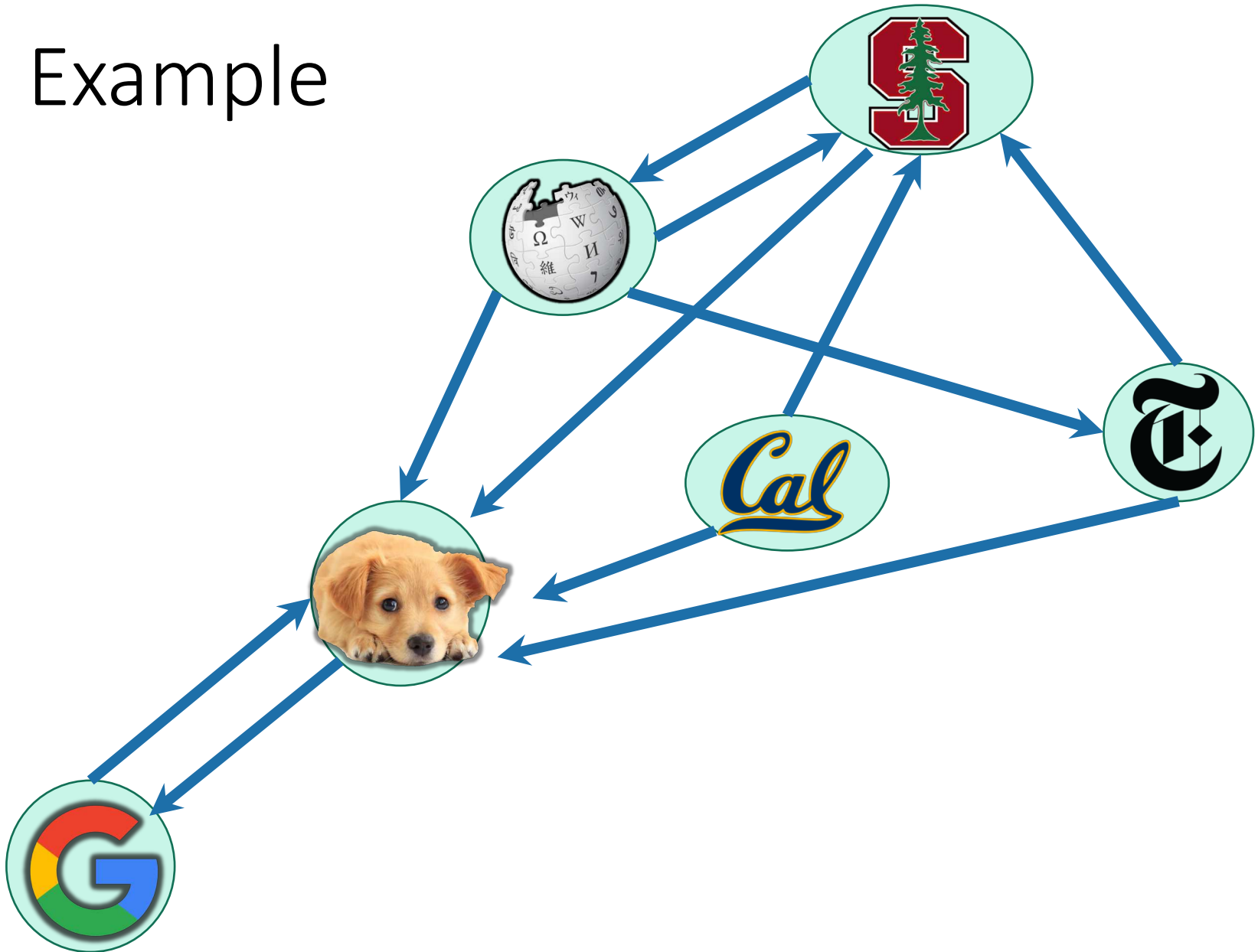
```
CS161Graph with:  
  Vertices:  
  Stanford,Wikipedia,NYTimes,Berkeley,Puppies,Google,  
  Edges:  
  (Stanford,Wikipedia) (Stanford,Puppies) (Wikipedia,Stanford) (Wikipedia,NYTimes) (Wikipedia,Puppies) (NYTimes,Stanford) (NYTimes,Puppies) (Berkeley,Stanford) (Berkeley,Puppies) (Puppies,Google) (Google,Puppies)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

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

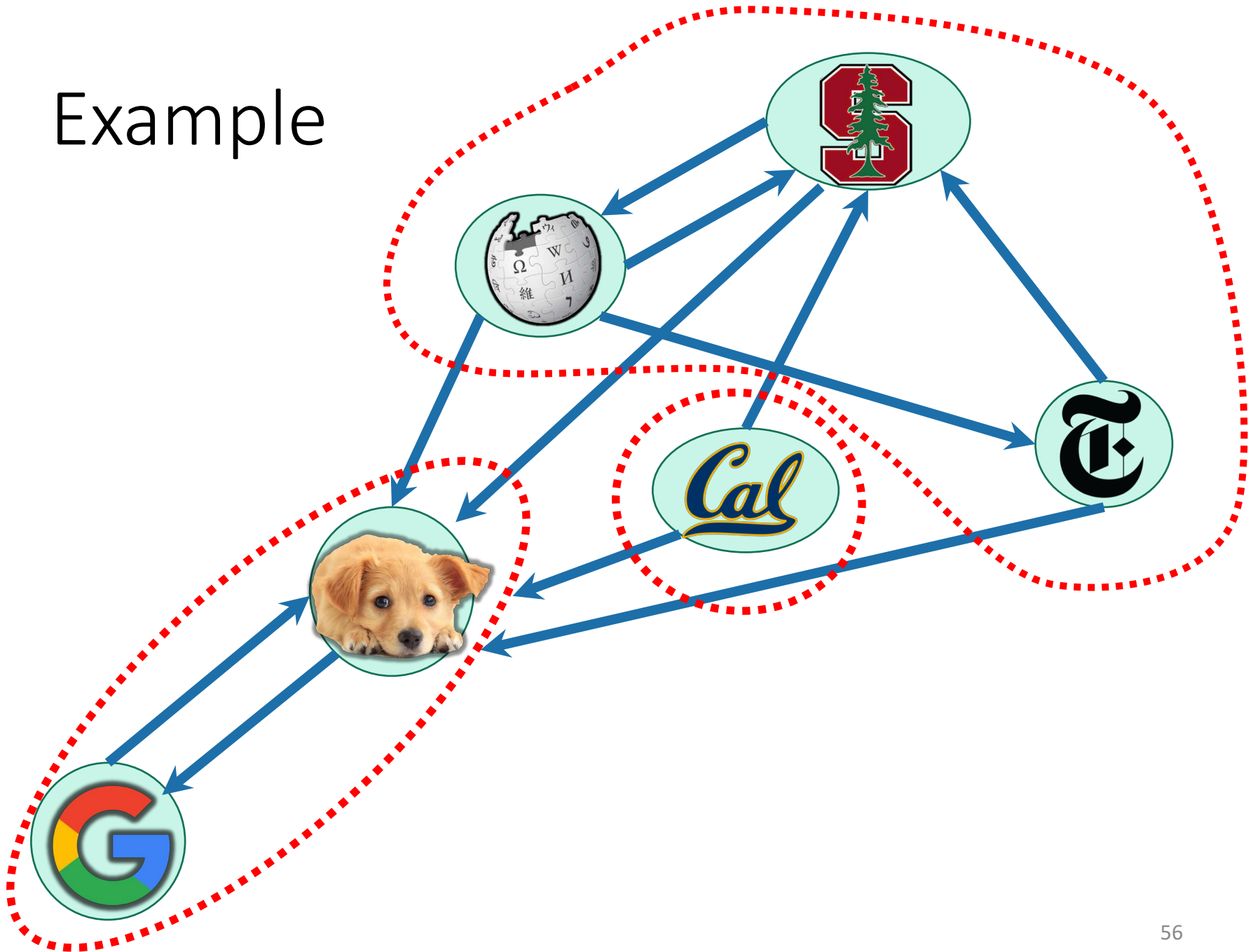
```
['Berkeley']  
['Stanford', 'NYTimes', 'Wikipedia']  
['Puppies', 'Google']
```

But let's break that down a bit...

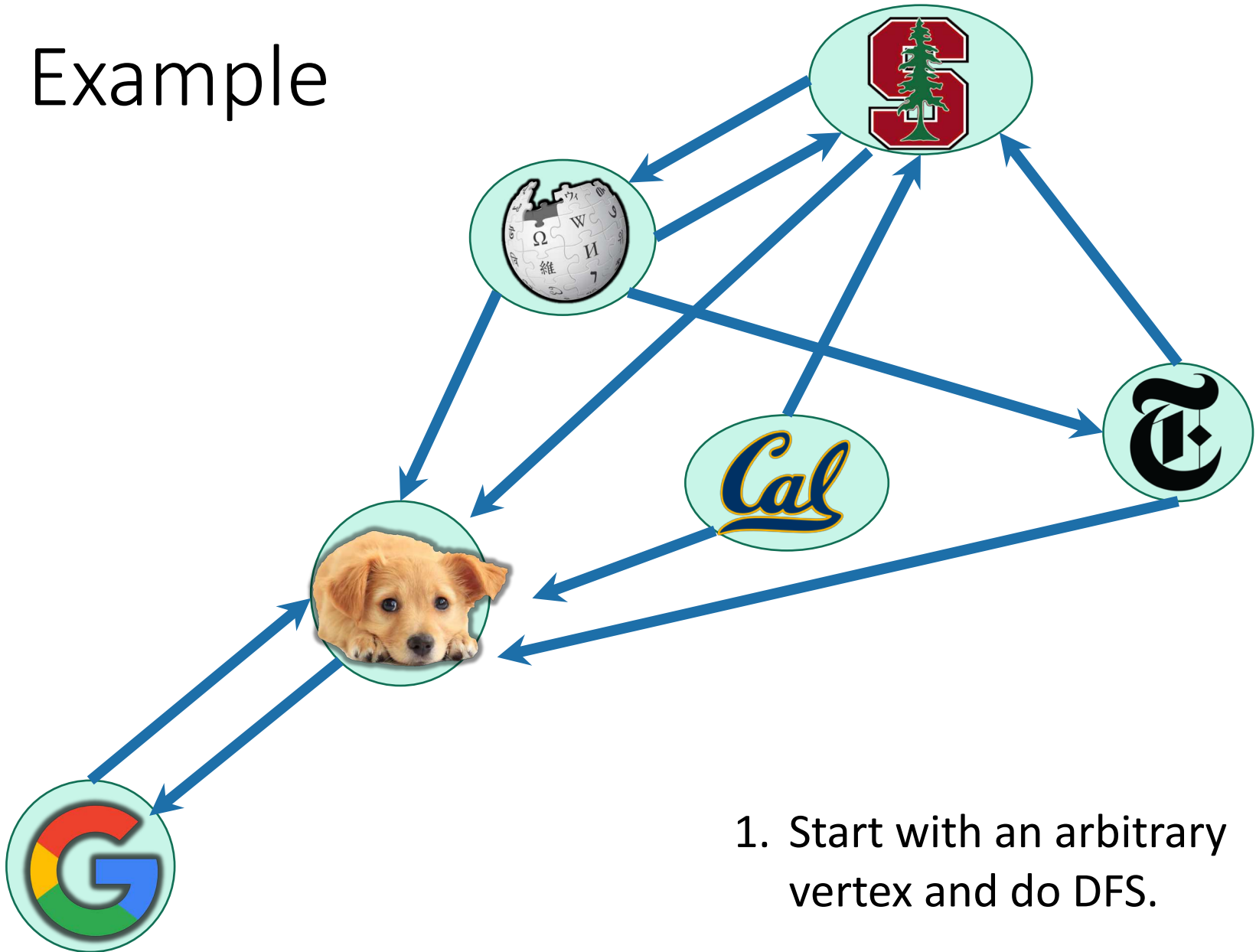
# Example



# Example

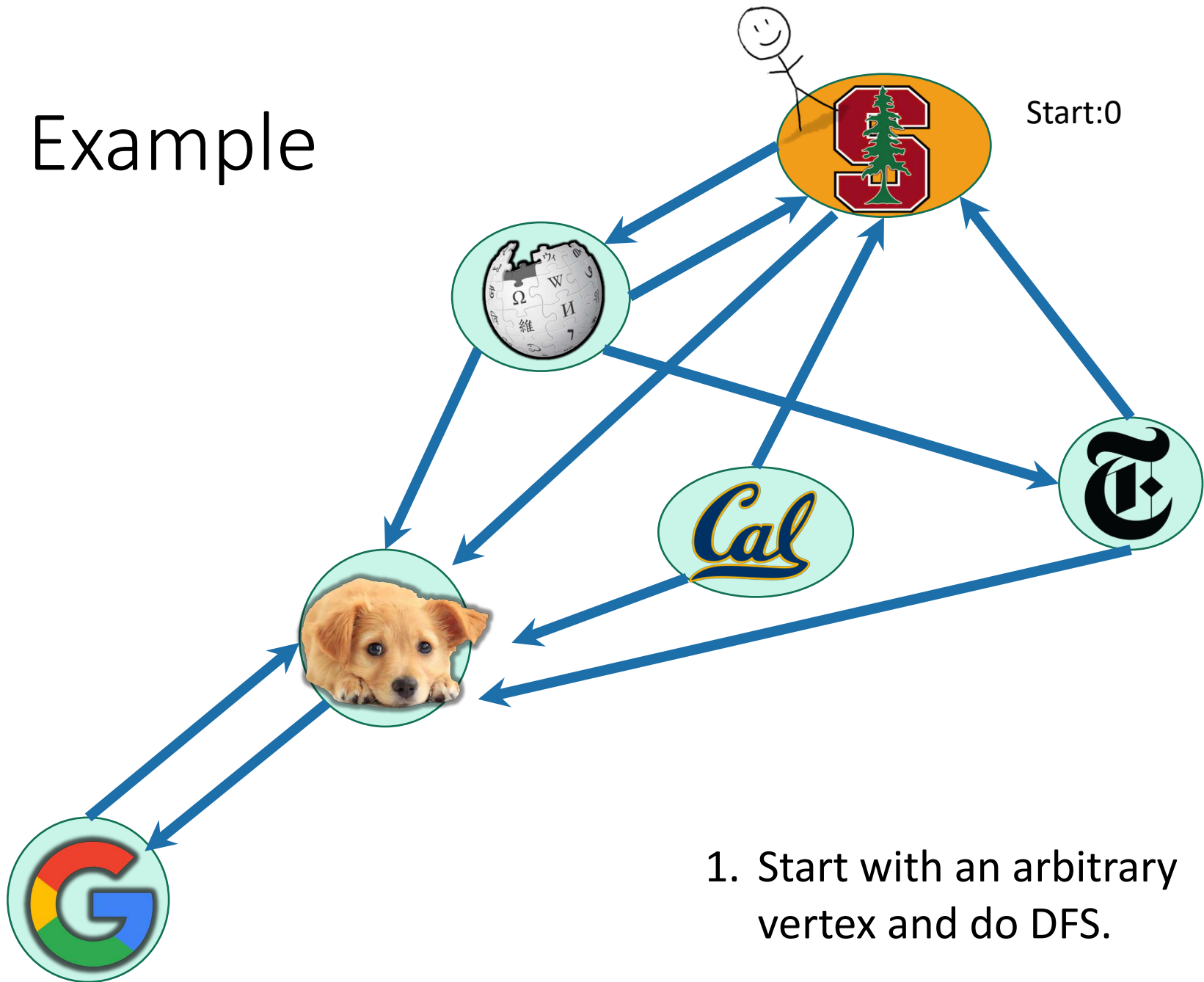


# Example



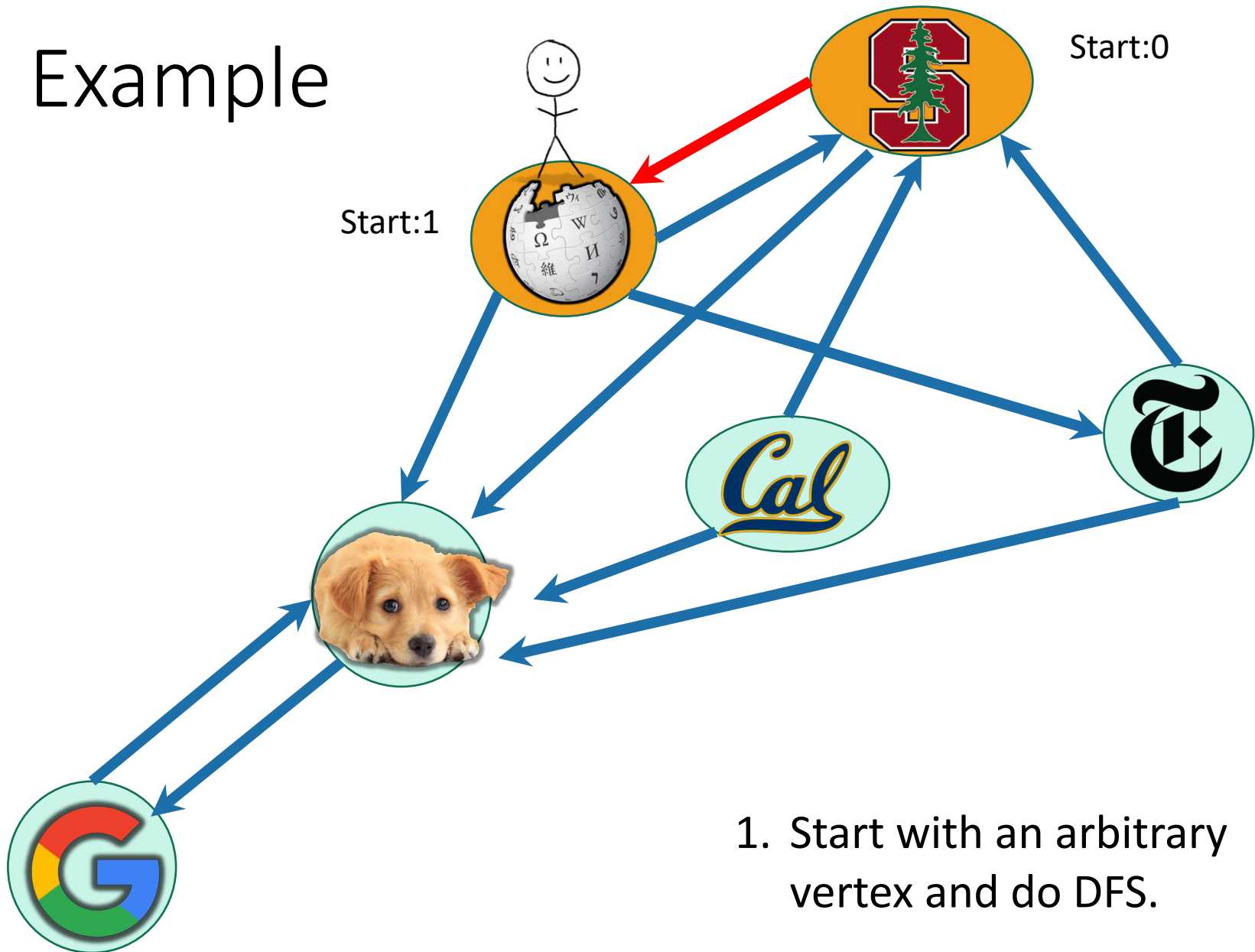
1. Start with an arbitrary vertex and do DFS.

# Example



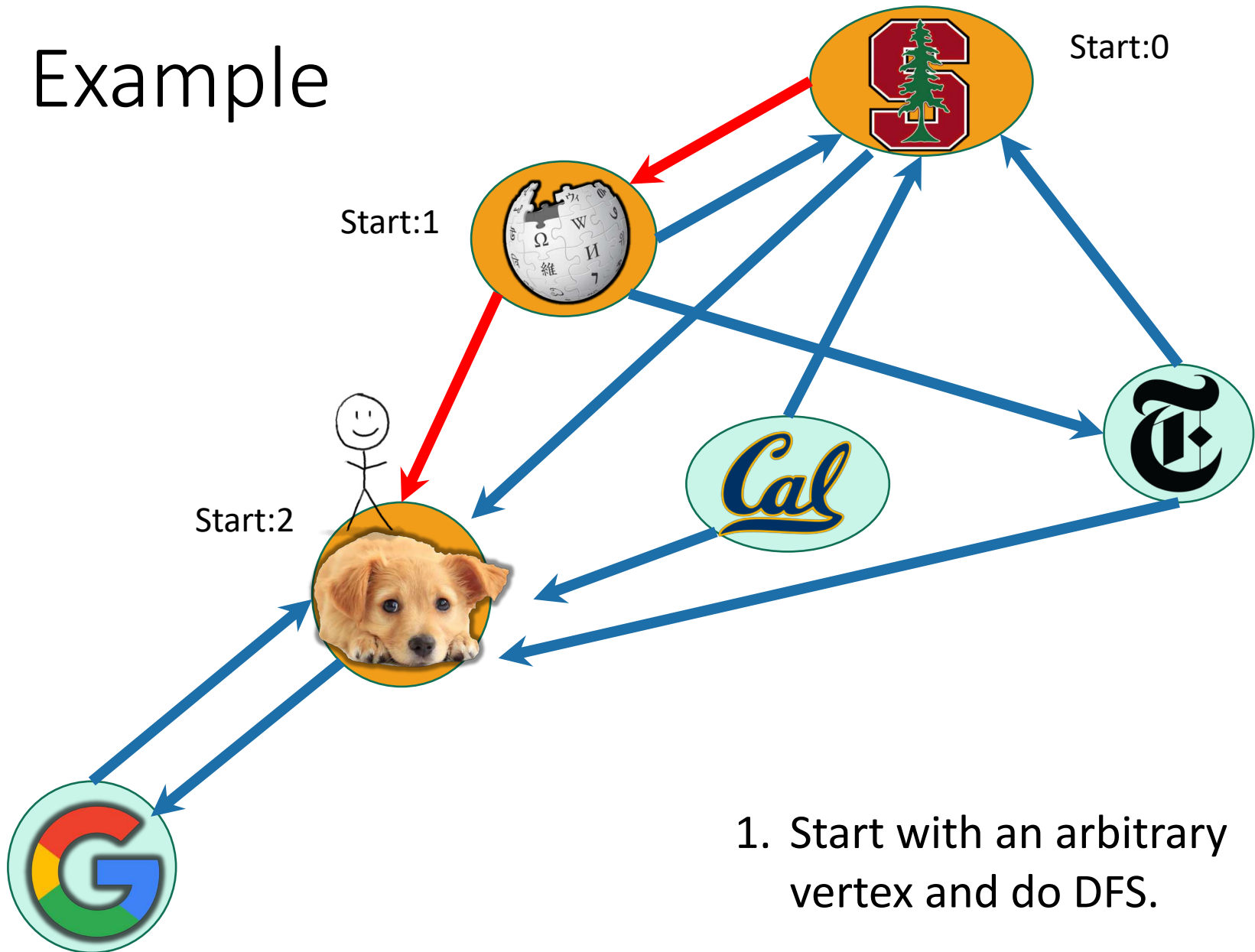
1. Start with an arbitrary vertex and do DFS.

# Example

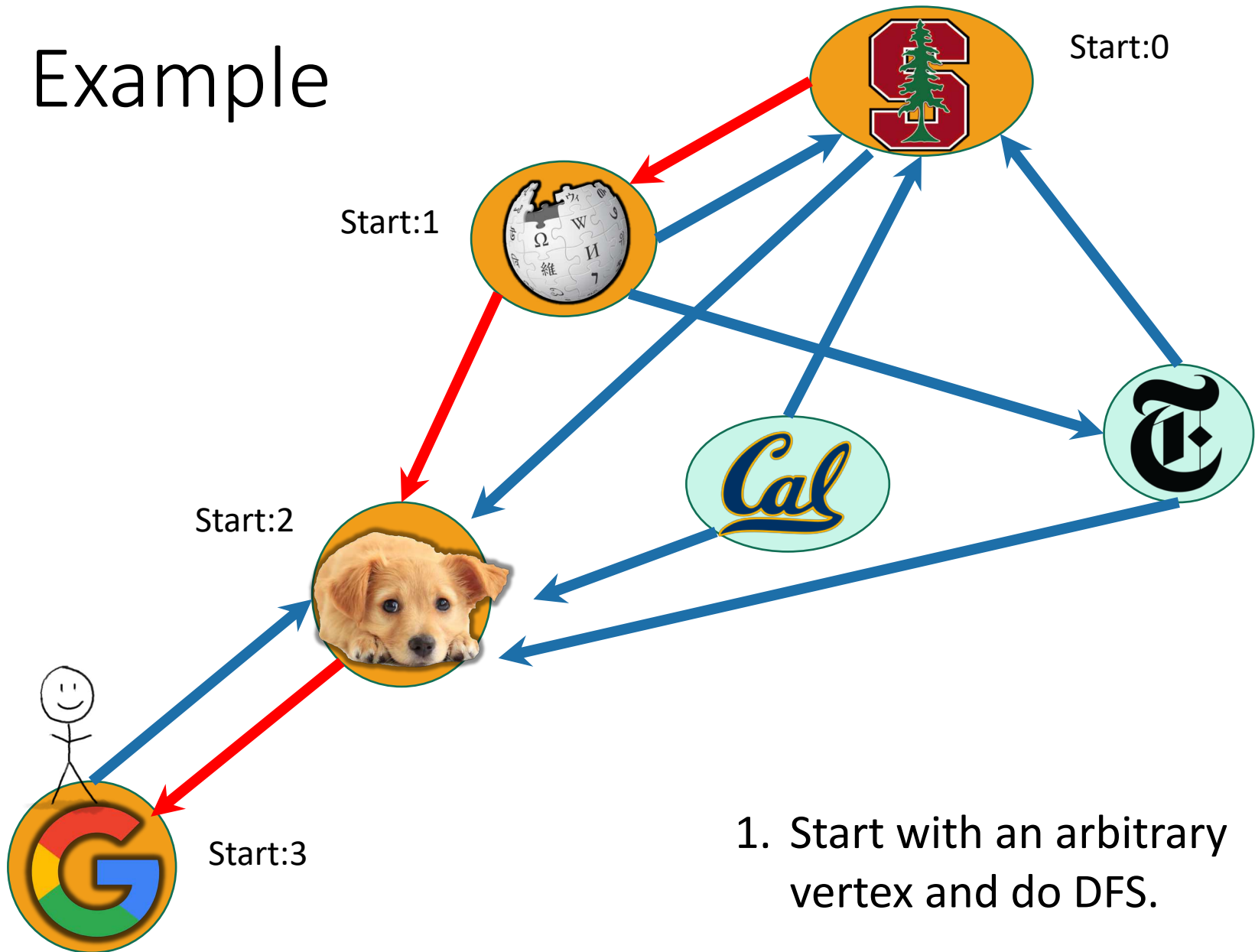


1. Start with an arbitrary vertex and do DFS.

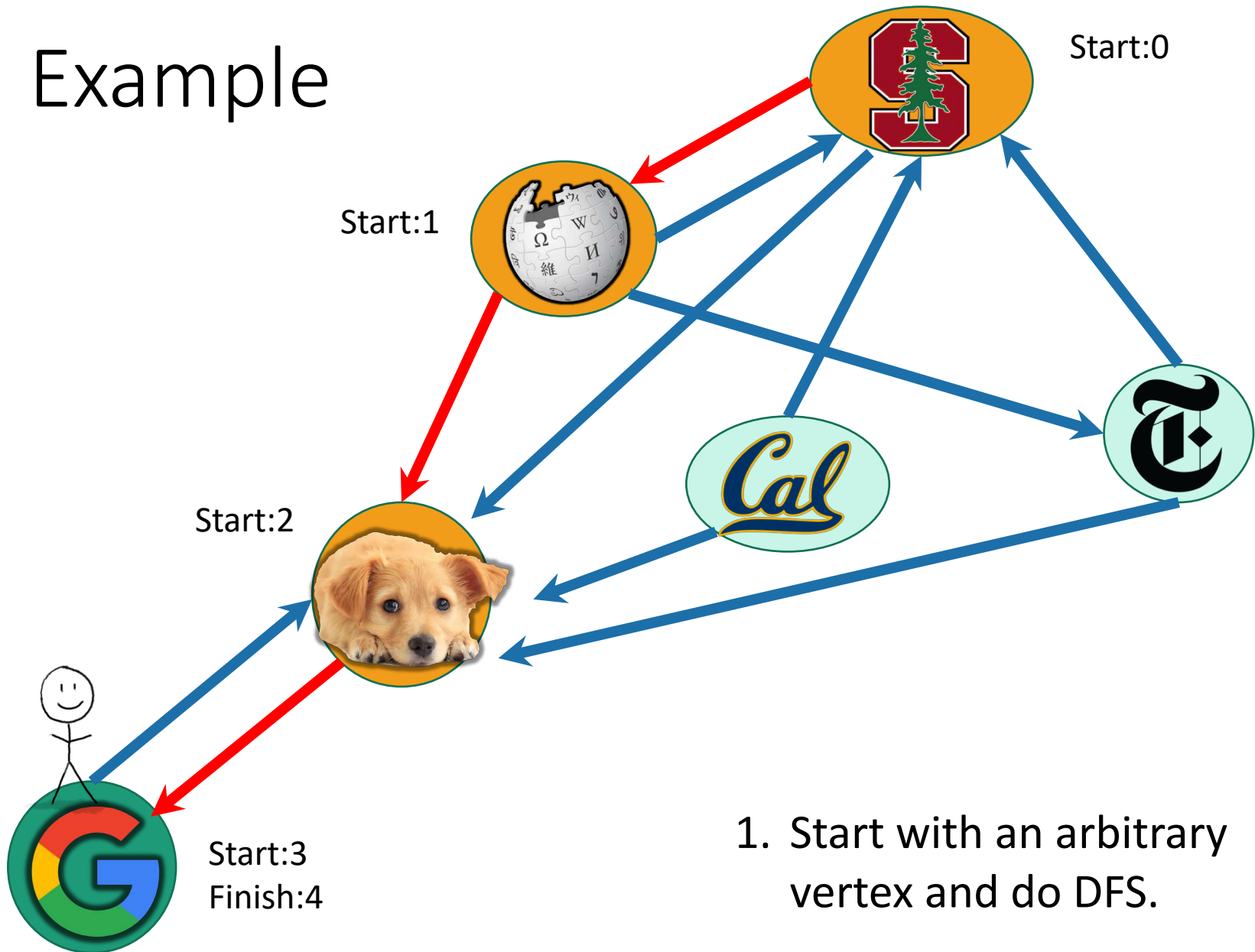
# Example



# Example

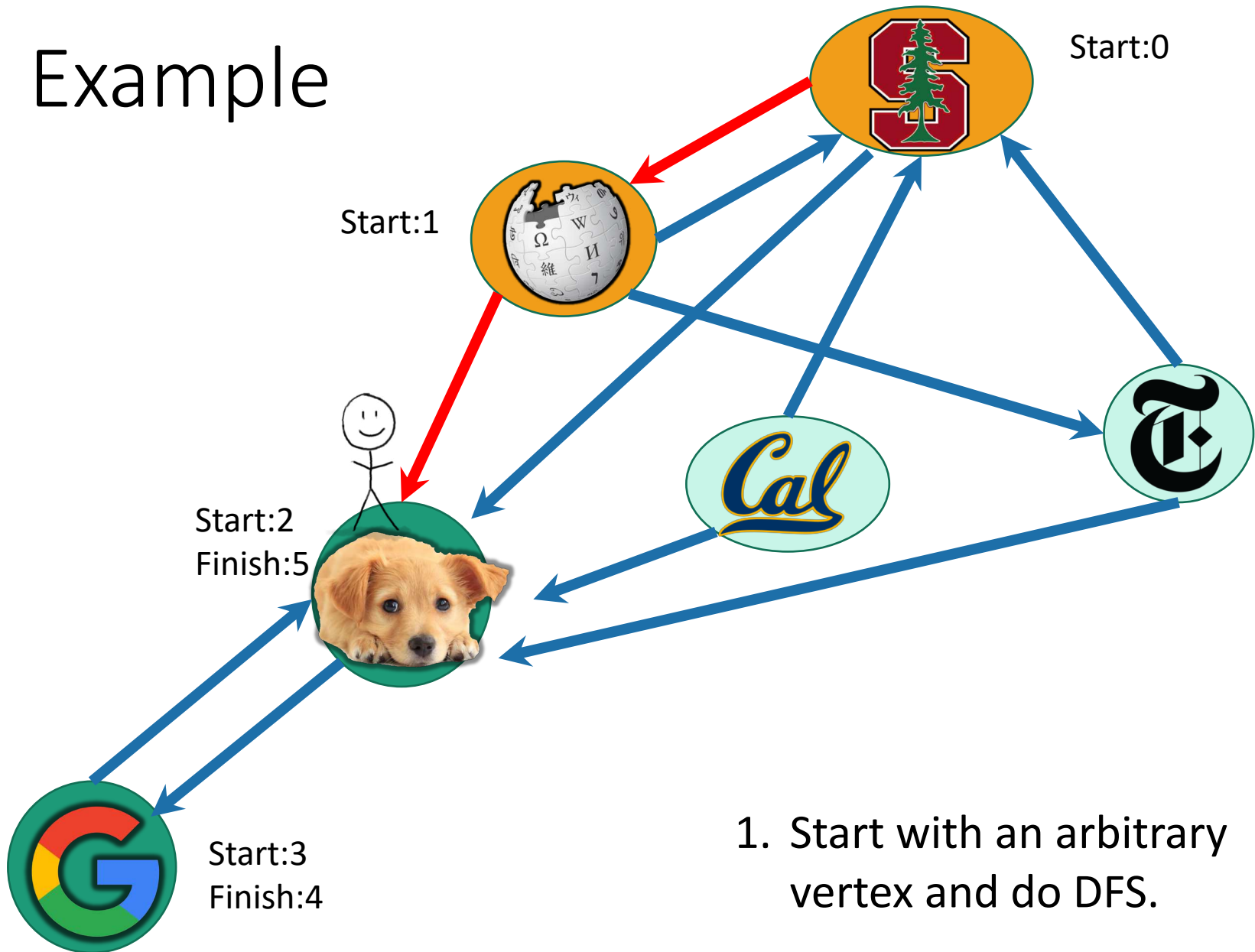


# Example



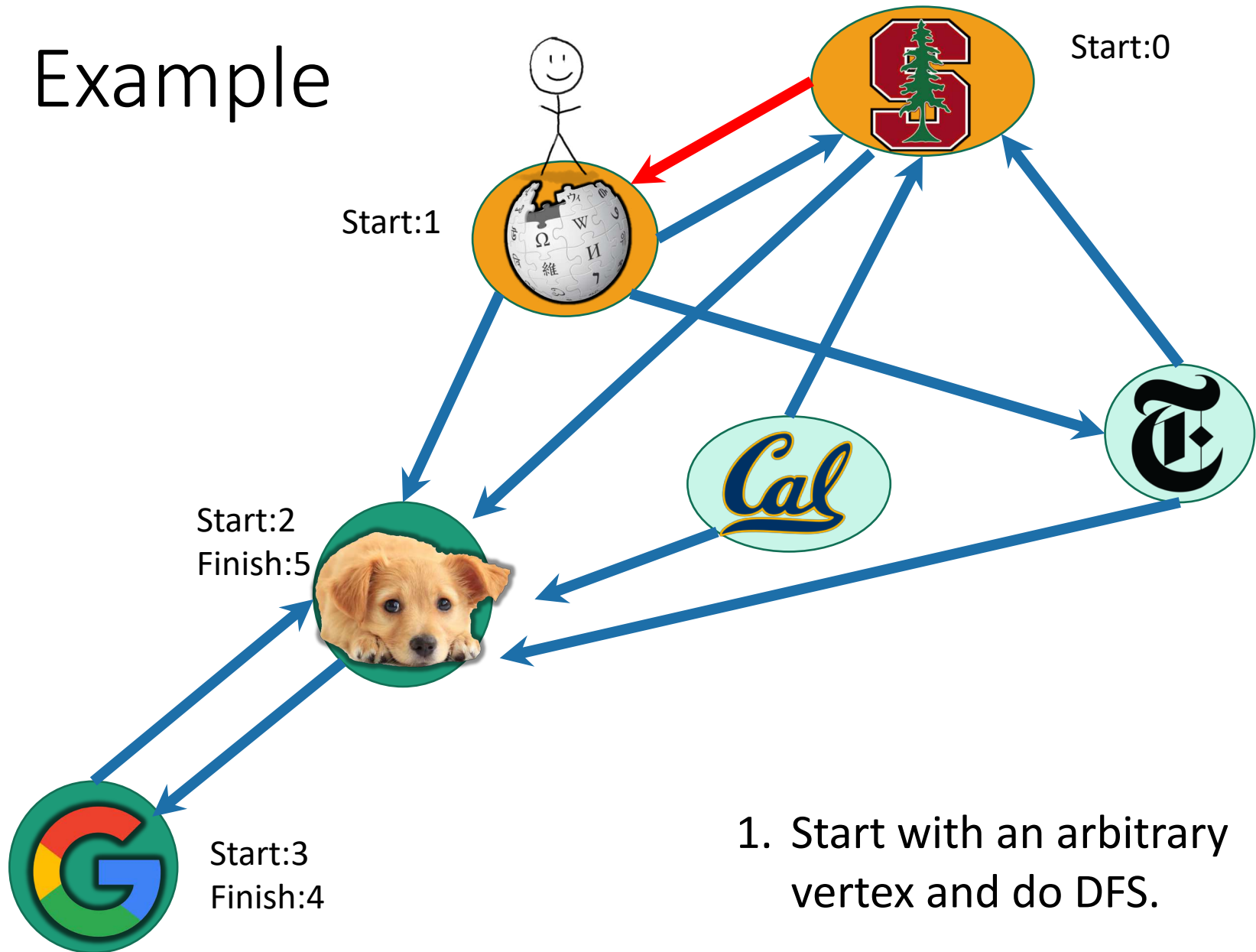
1. Start with an arbitrary vertex and do DFS.

# Example



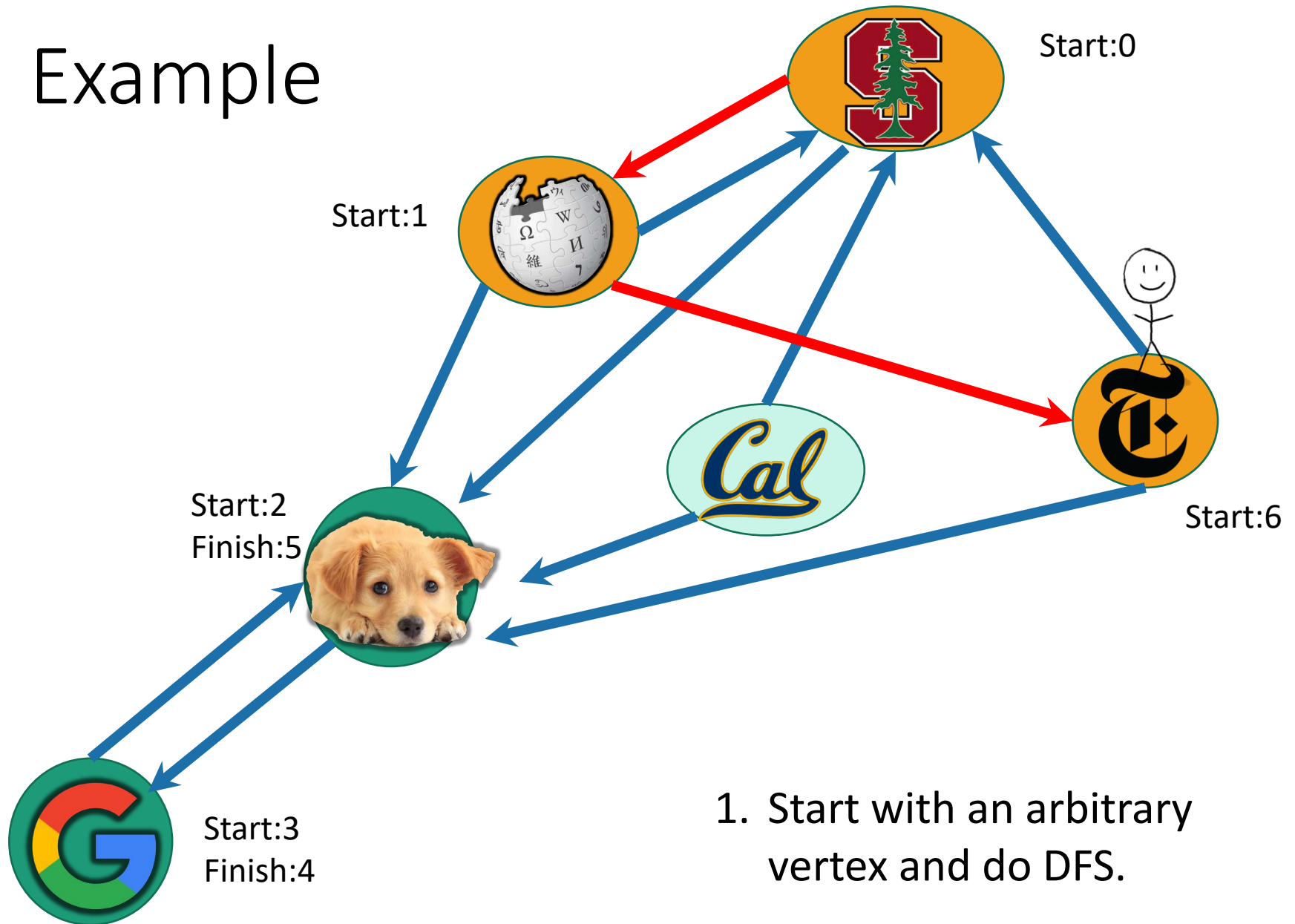
1. Start with an arbitrary vertex and do DFS.

# Example

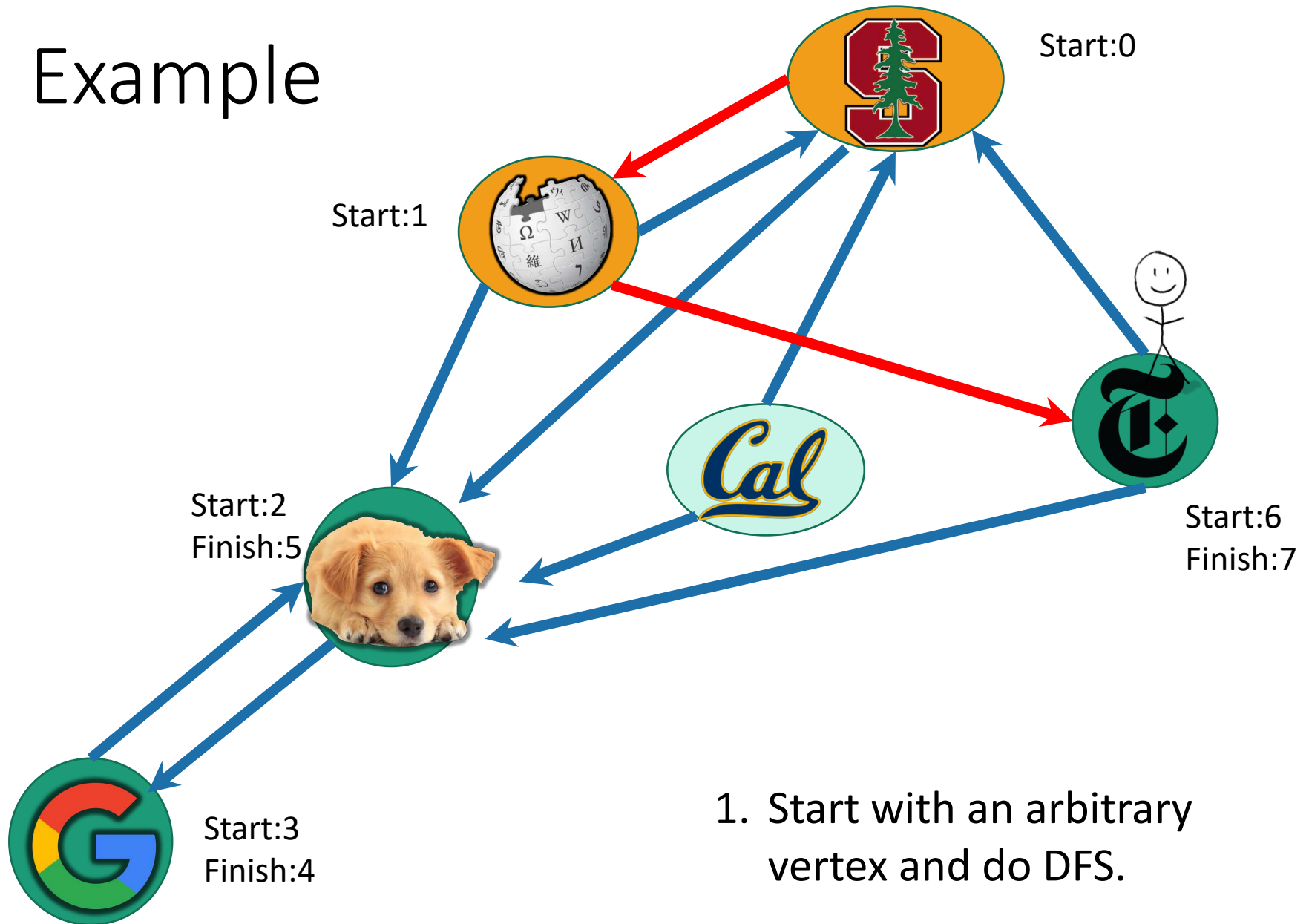


1. Start with an arbitrary vertex and do DFS.

# Example

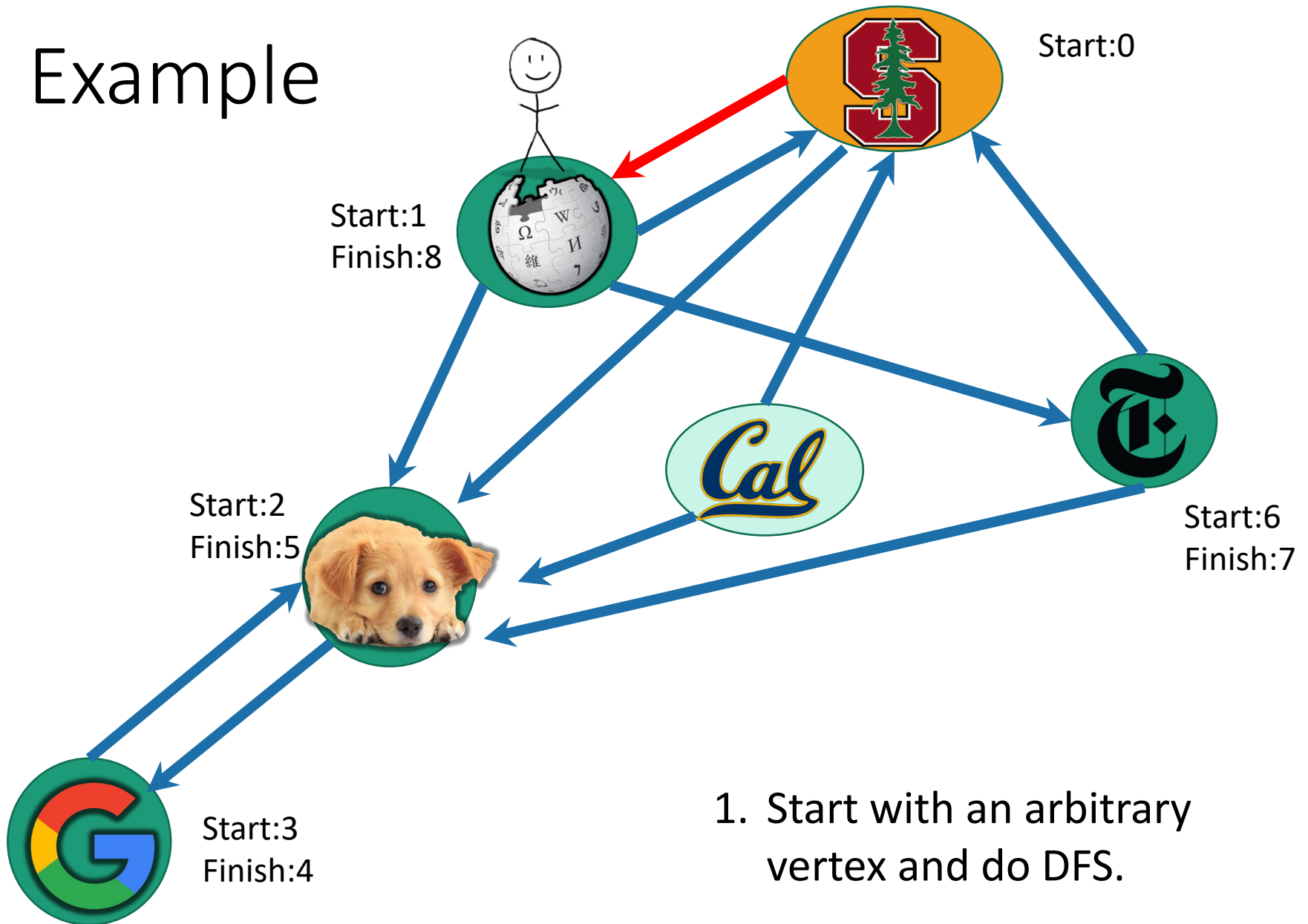


# Example

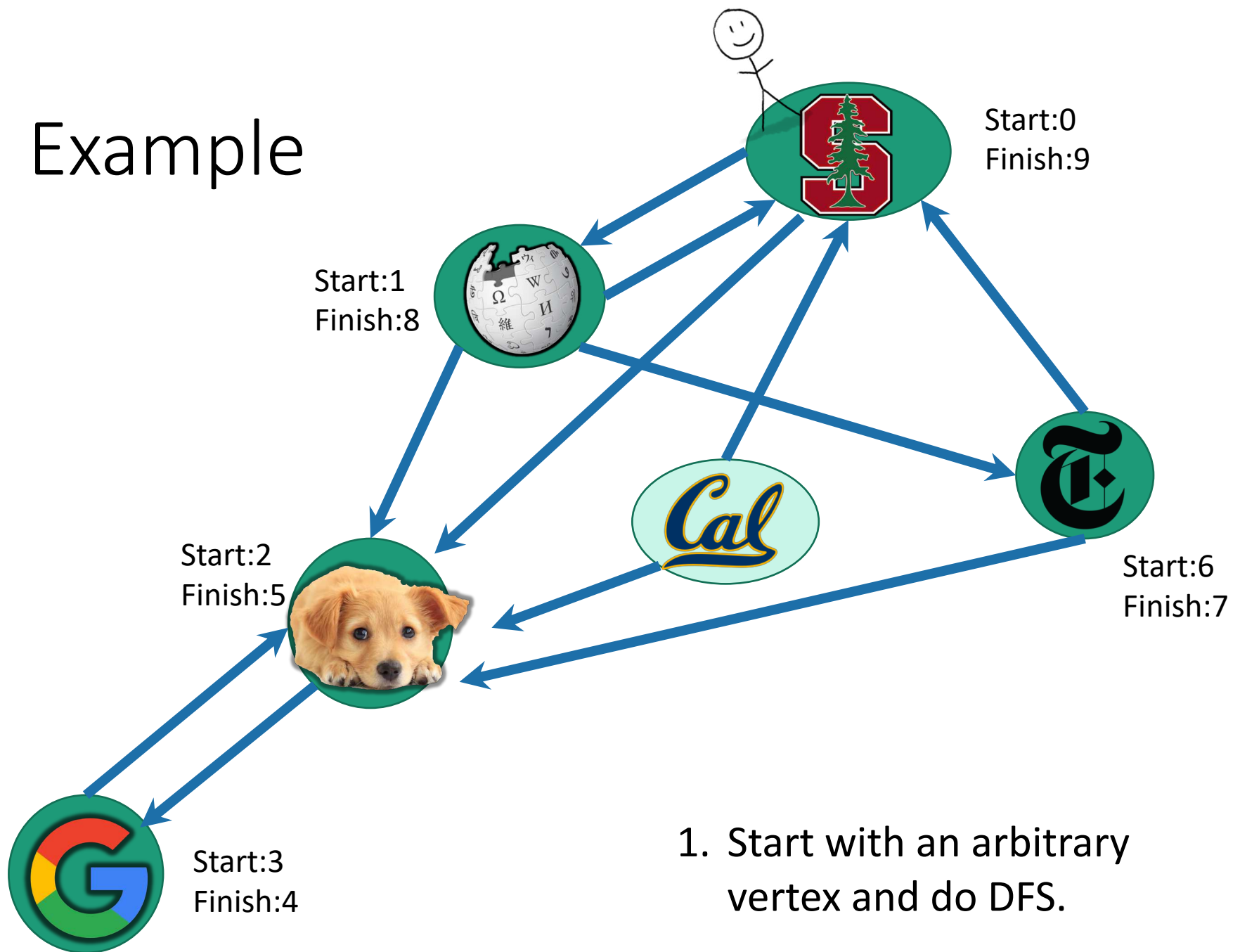


1. Start with an arbitrary vertex and do DFS.

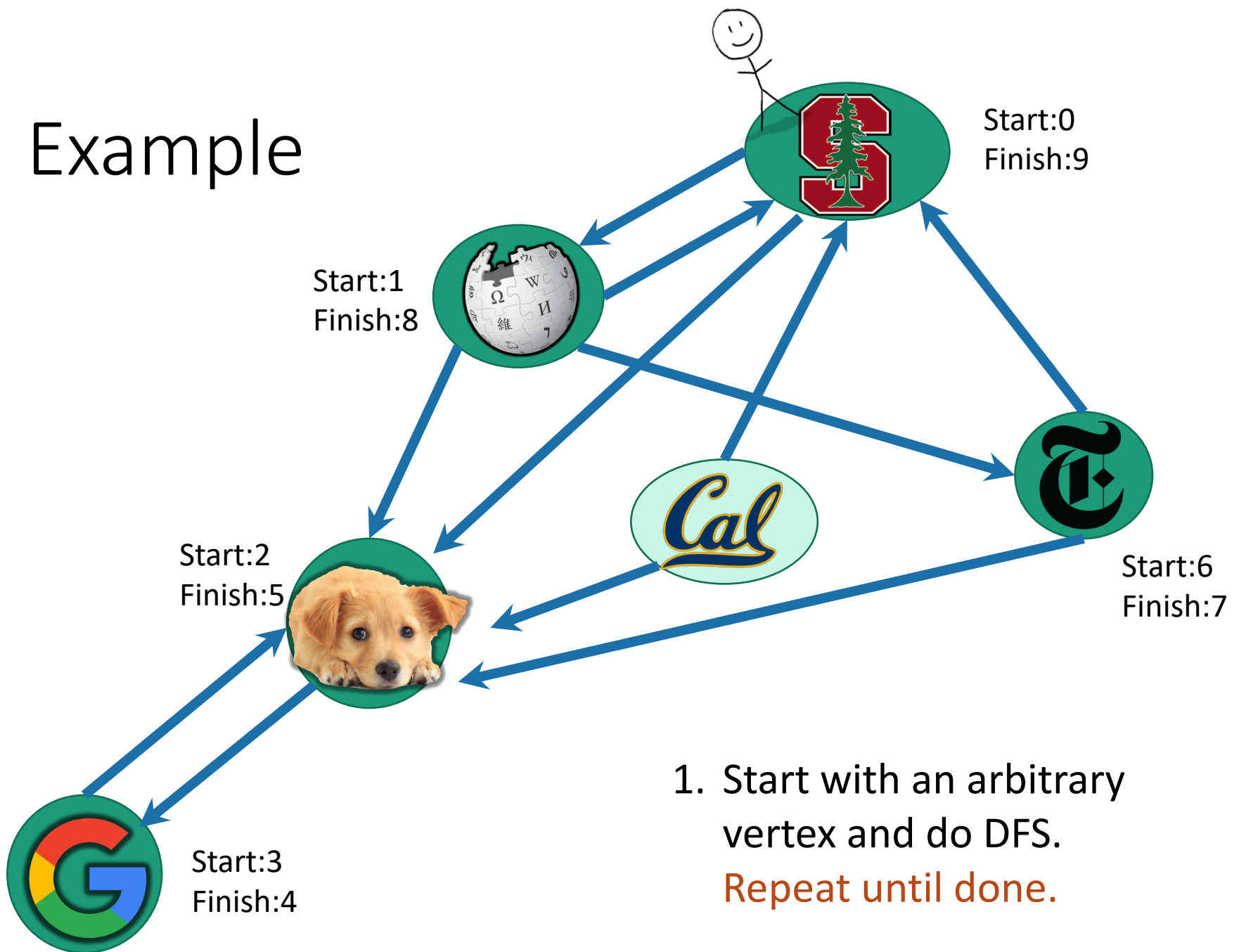
# Example



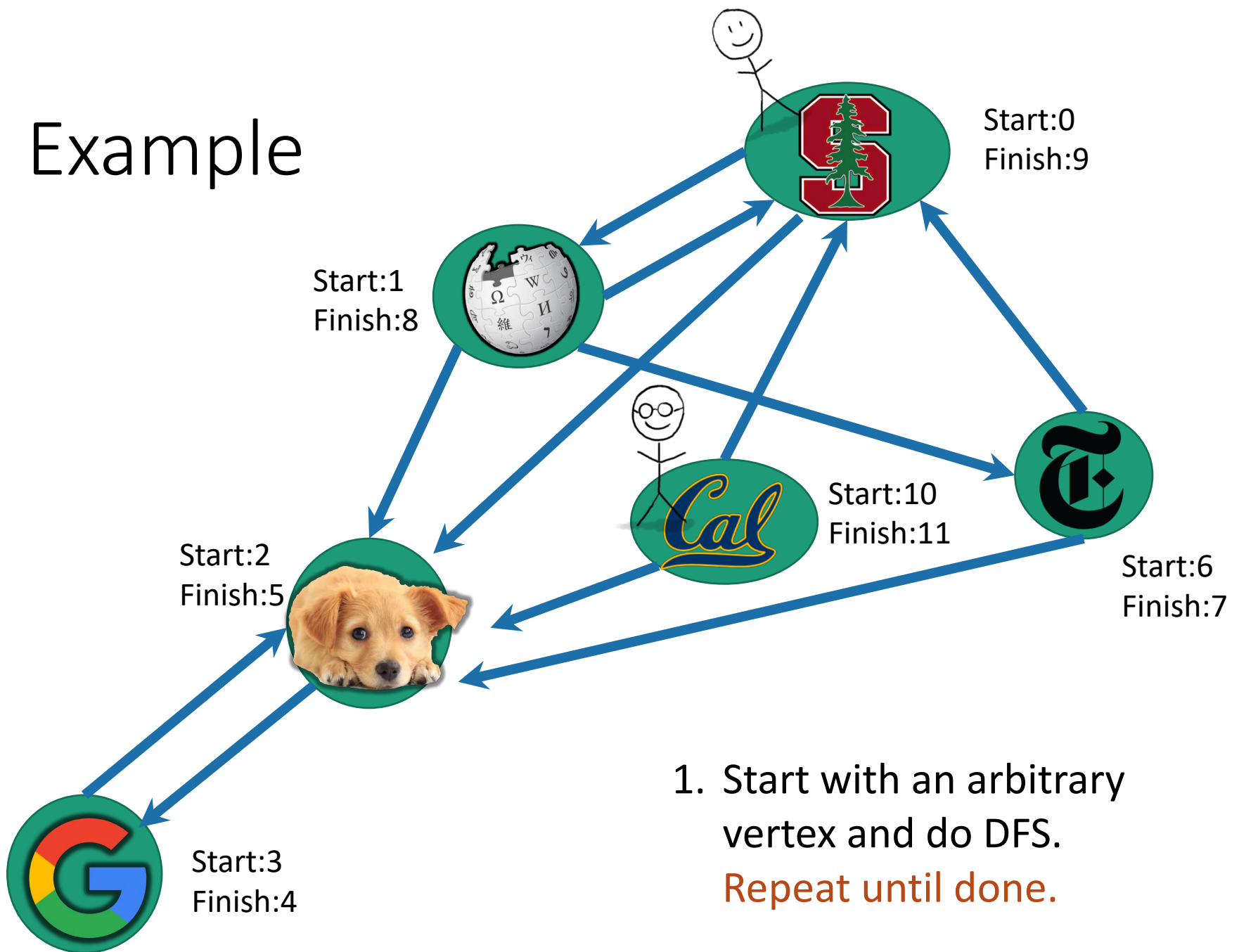
# Example



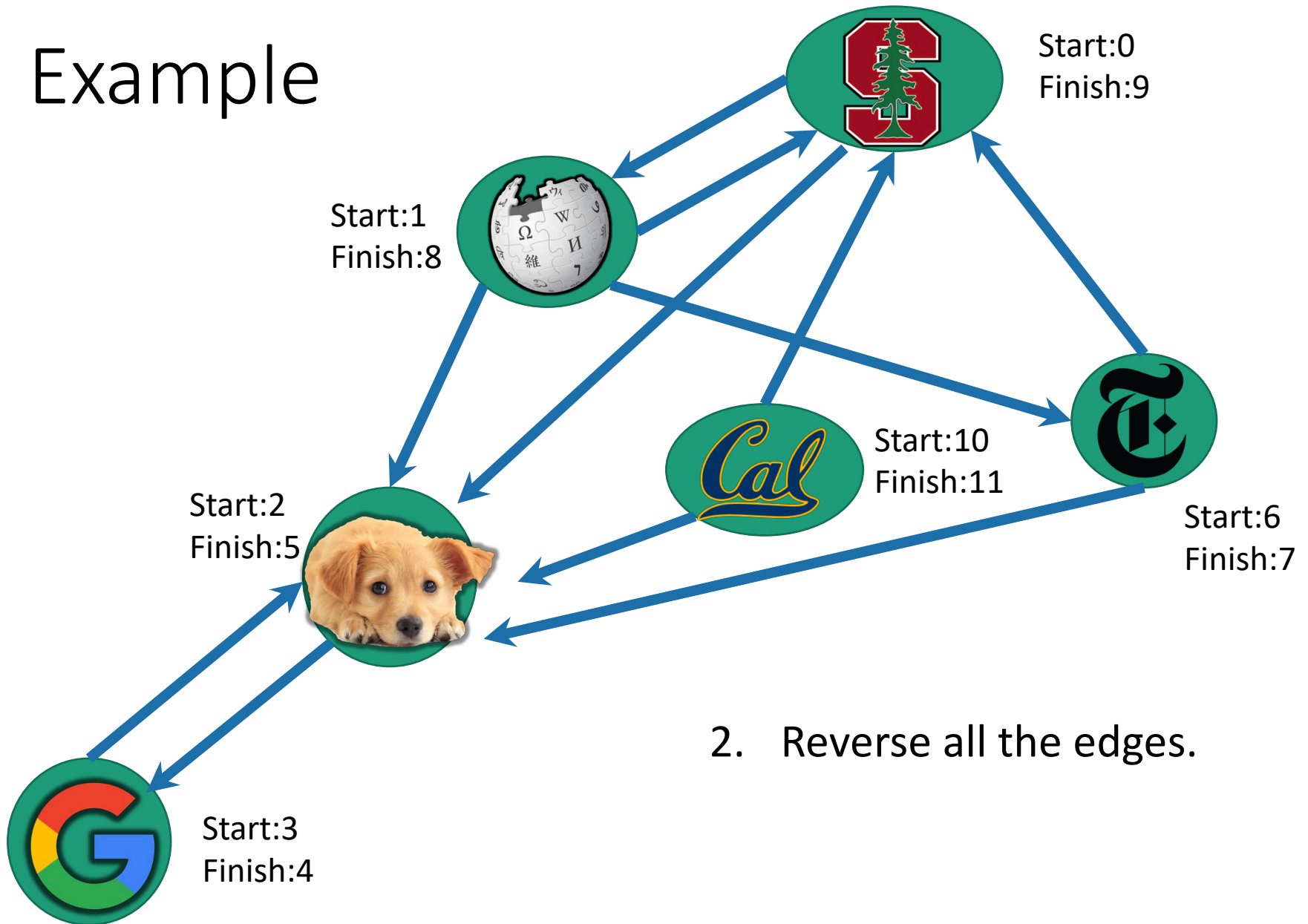
# Example



# Example

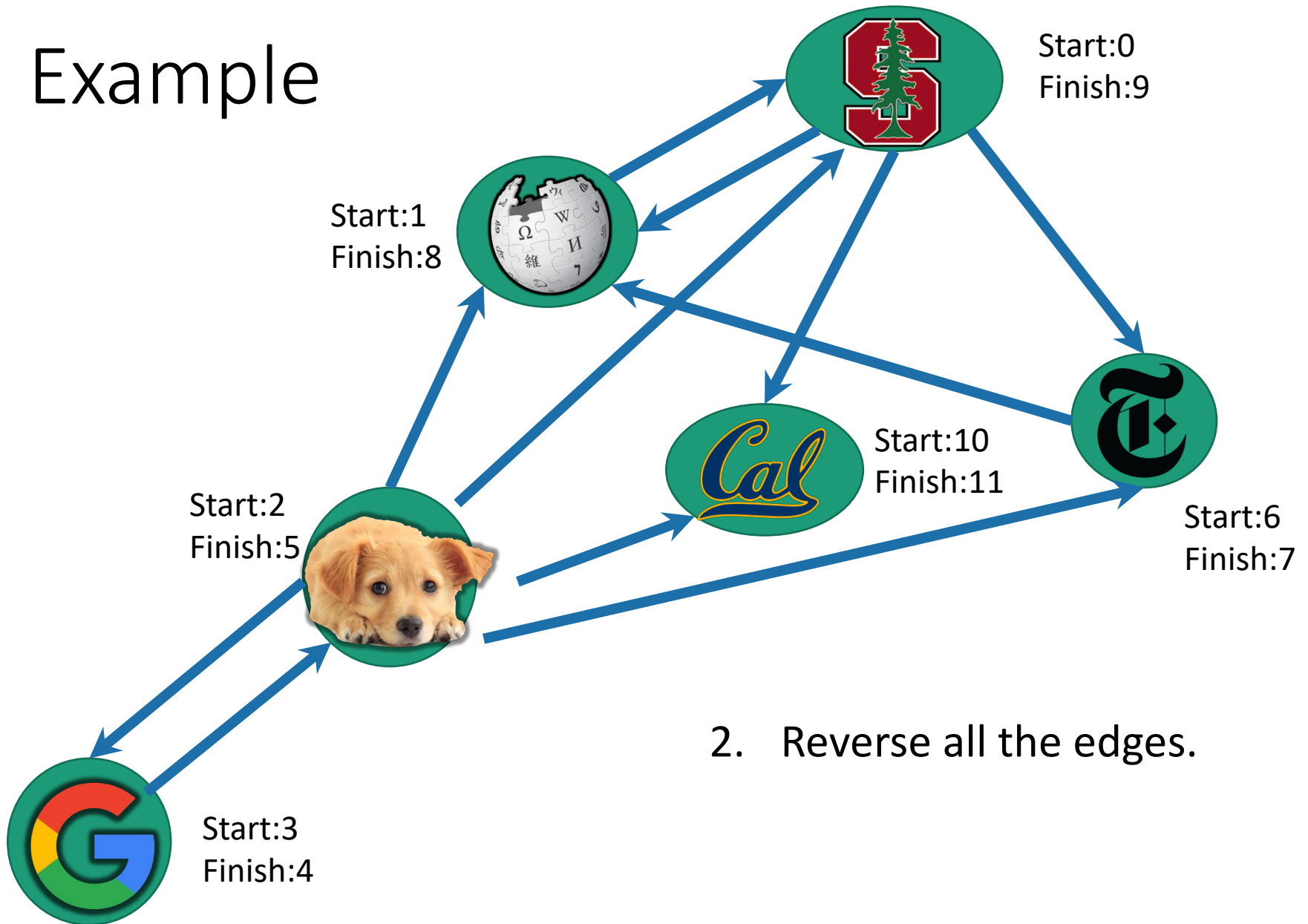


# Example

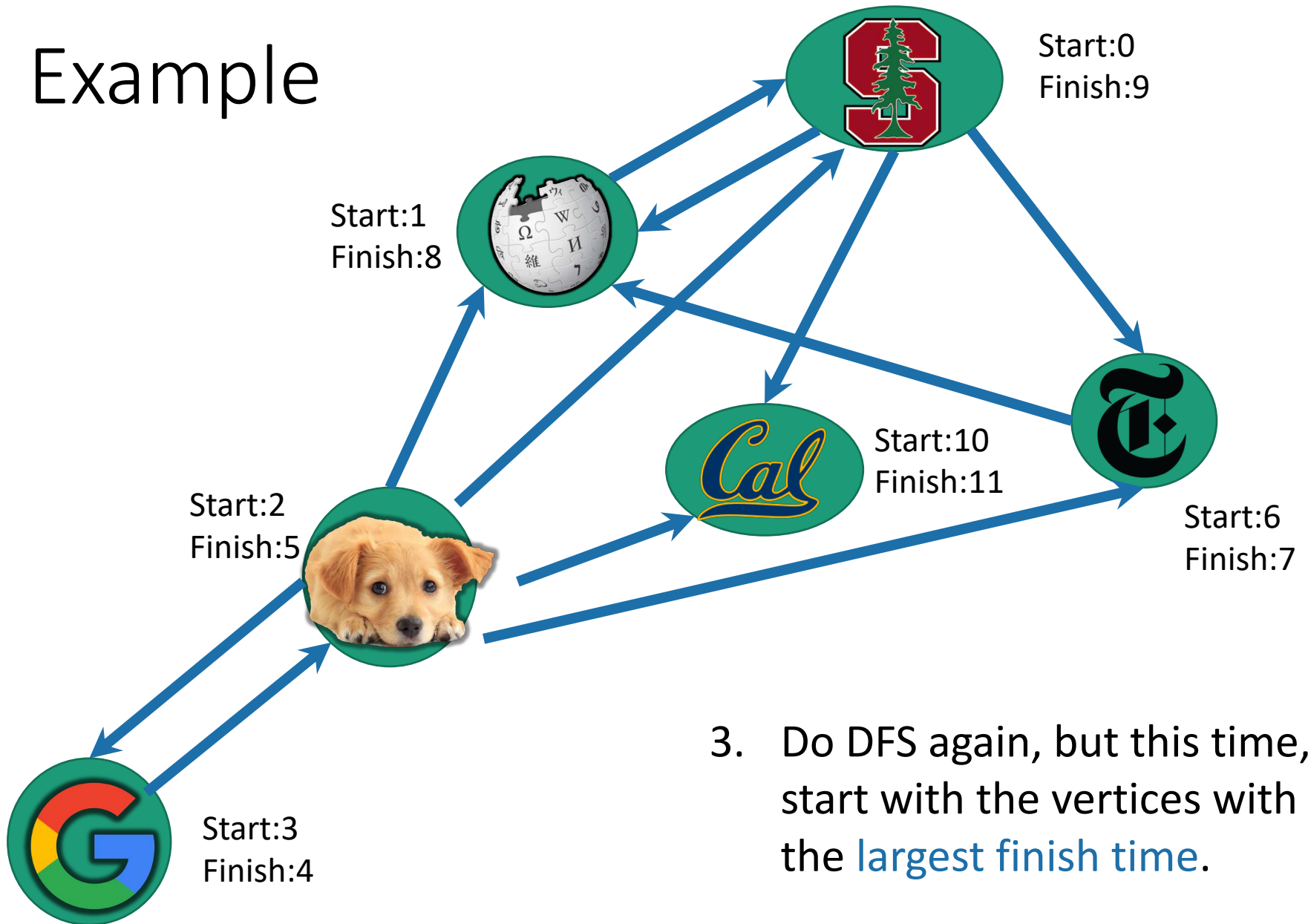


2. Reverse all the edges.

# Example

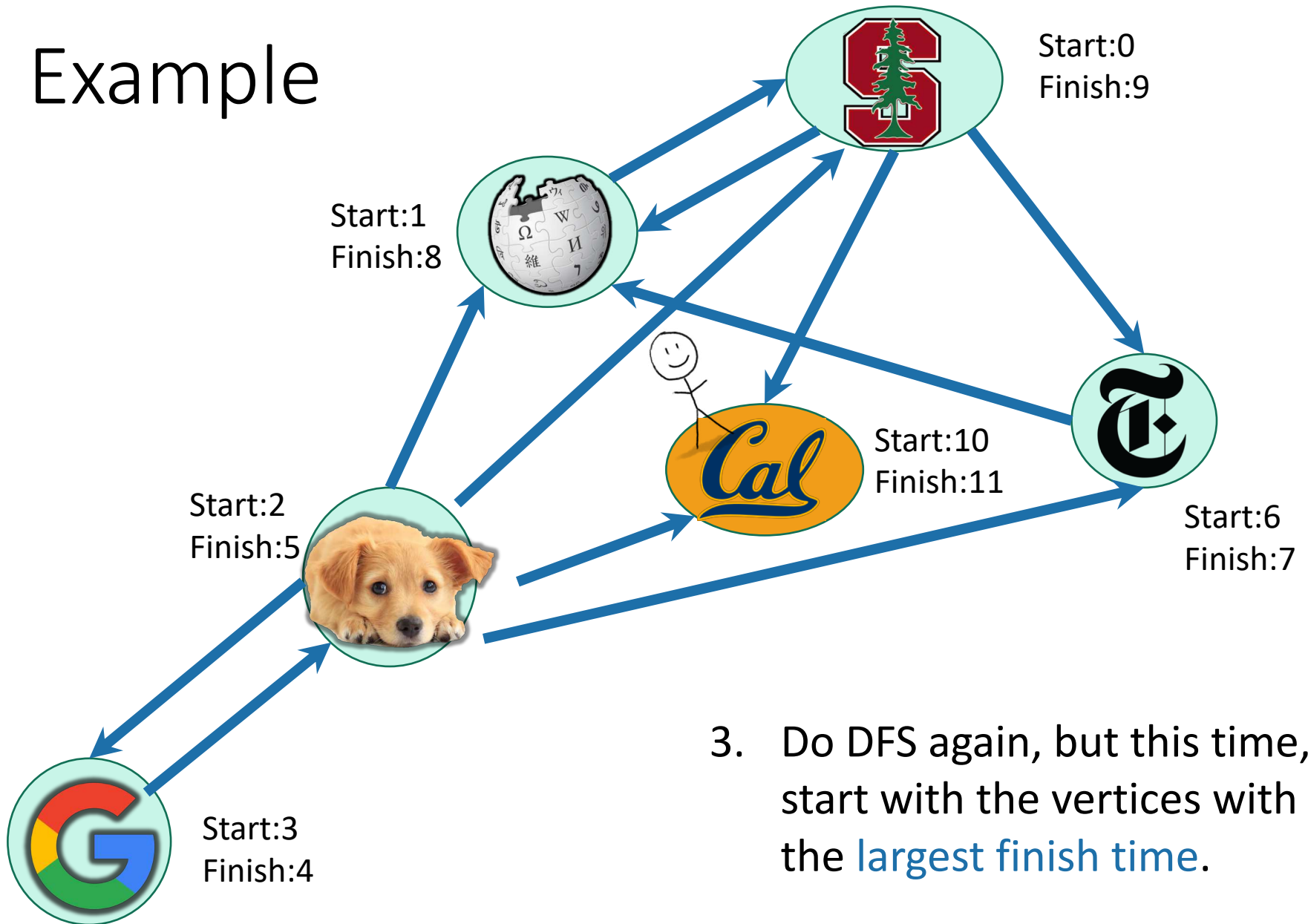


# Example



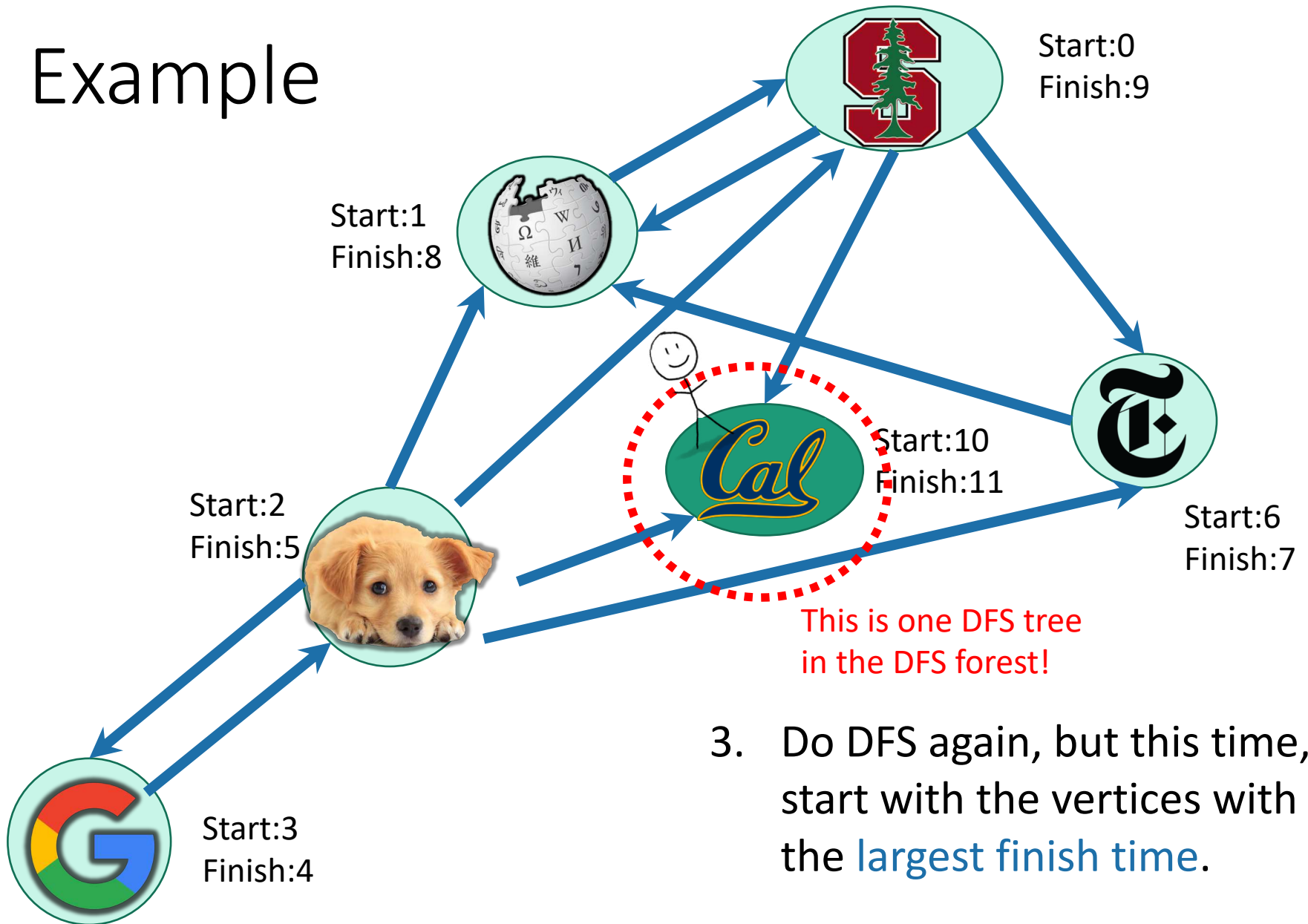
3. Do DFS again, but this time, start with the vertices with the largest finish time.

# Example



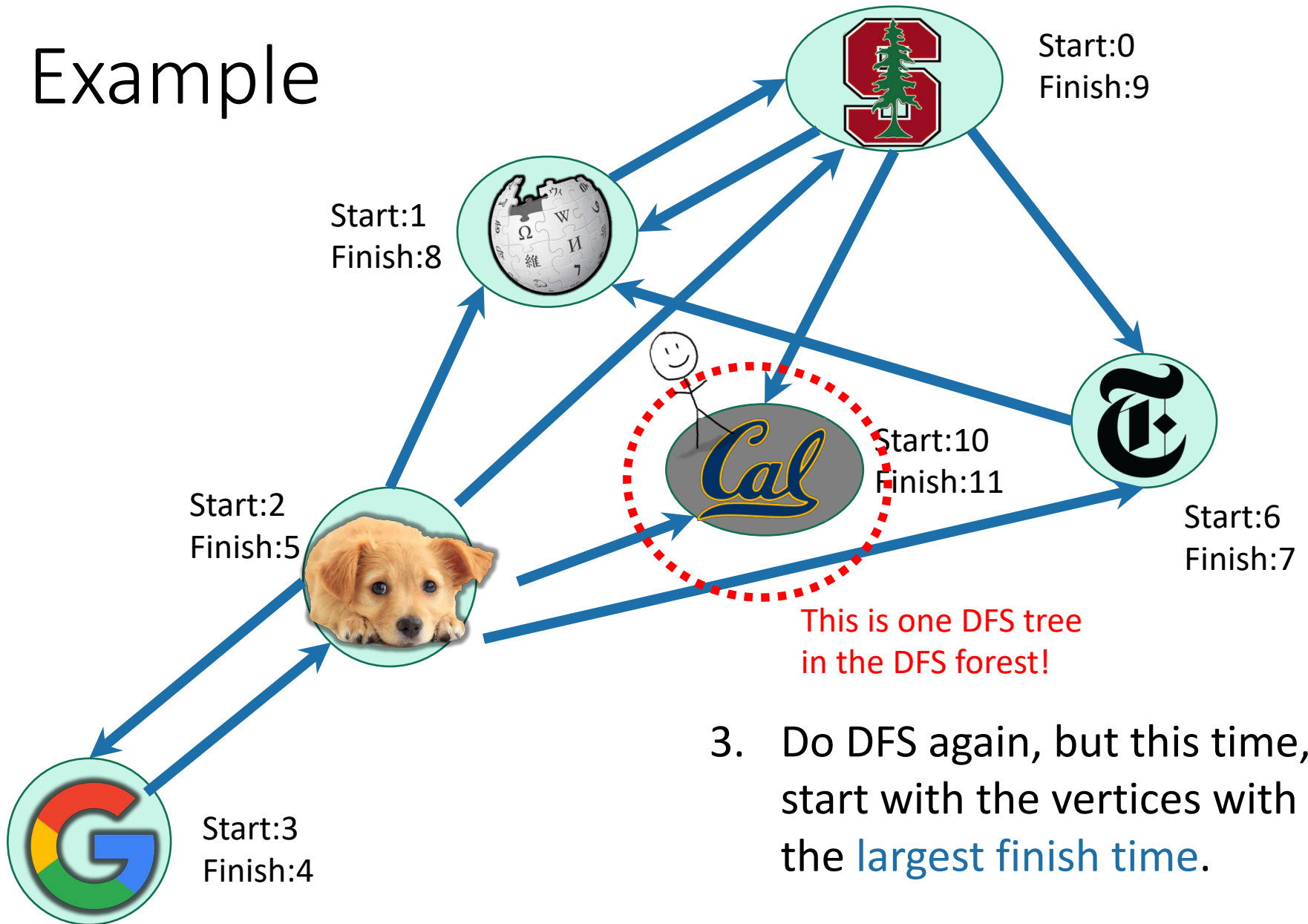
3. Do DFS again, but this time, start with the vertices with the largest finish time.

# Example



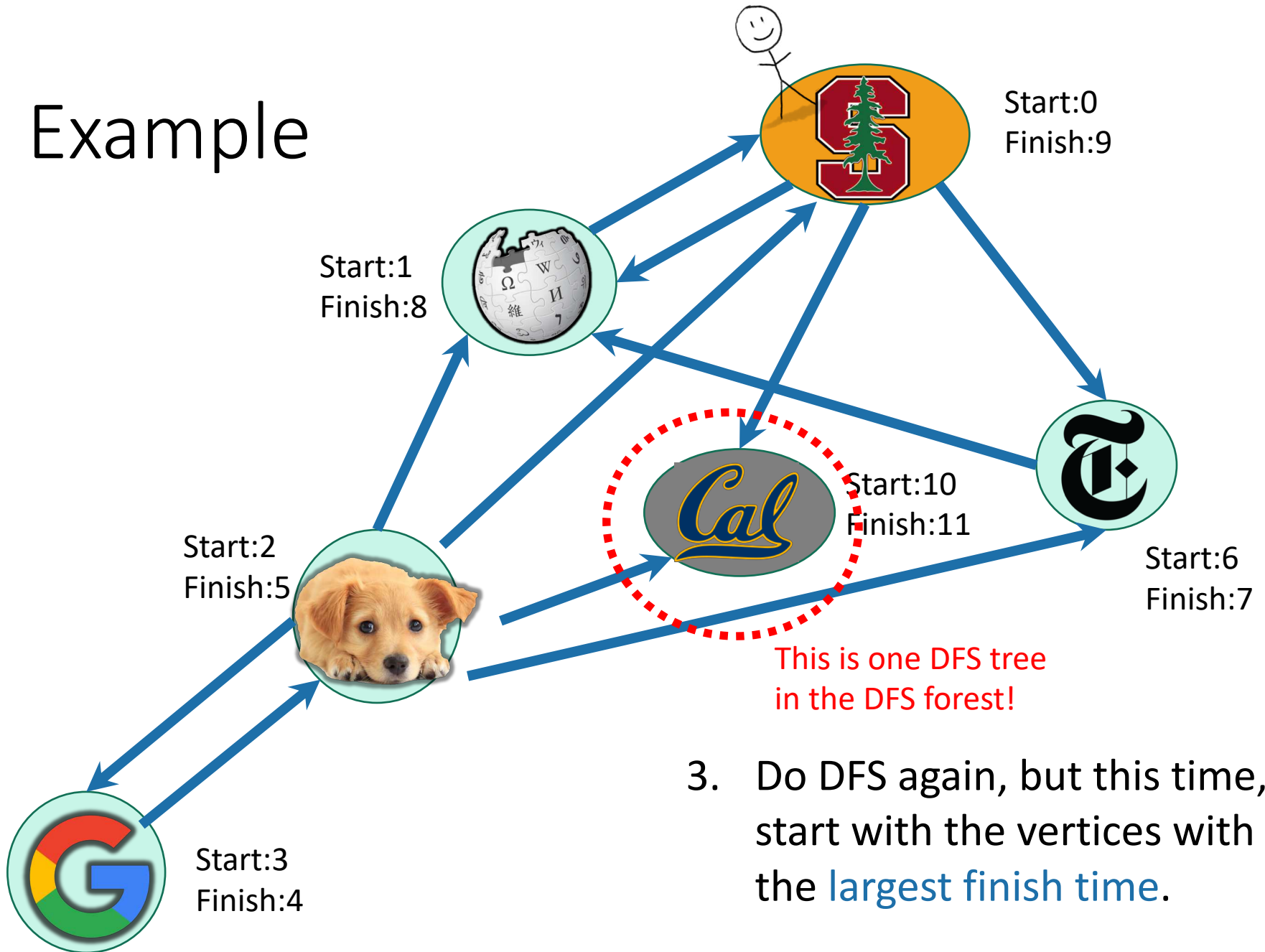
3. Do DFS again, but this time, start with the vertices with the **largest finish time**.

# Example



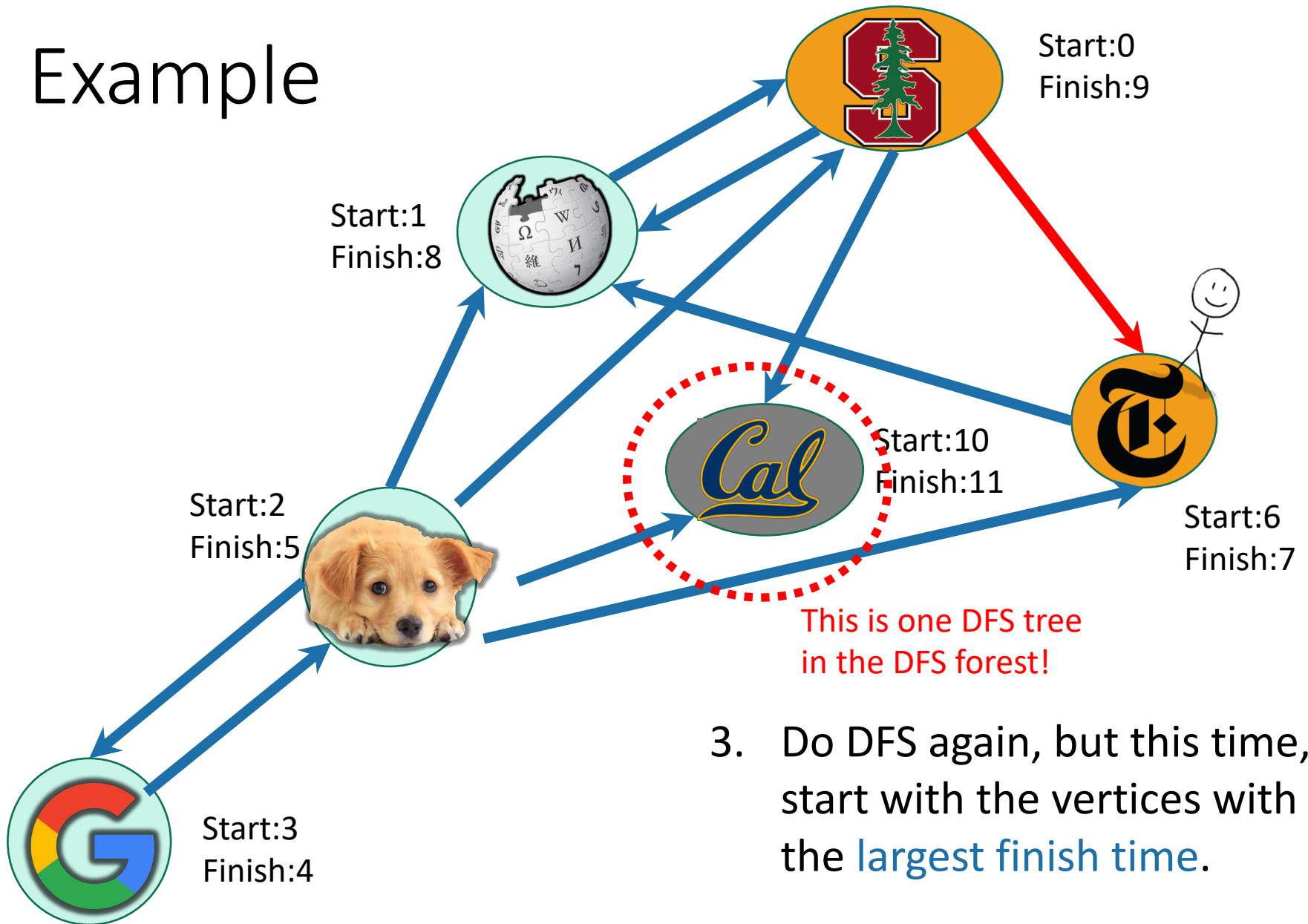
3. Do DFS again, but this time, start with the vertices with the **largest finish time**.

# Example

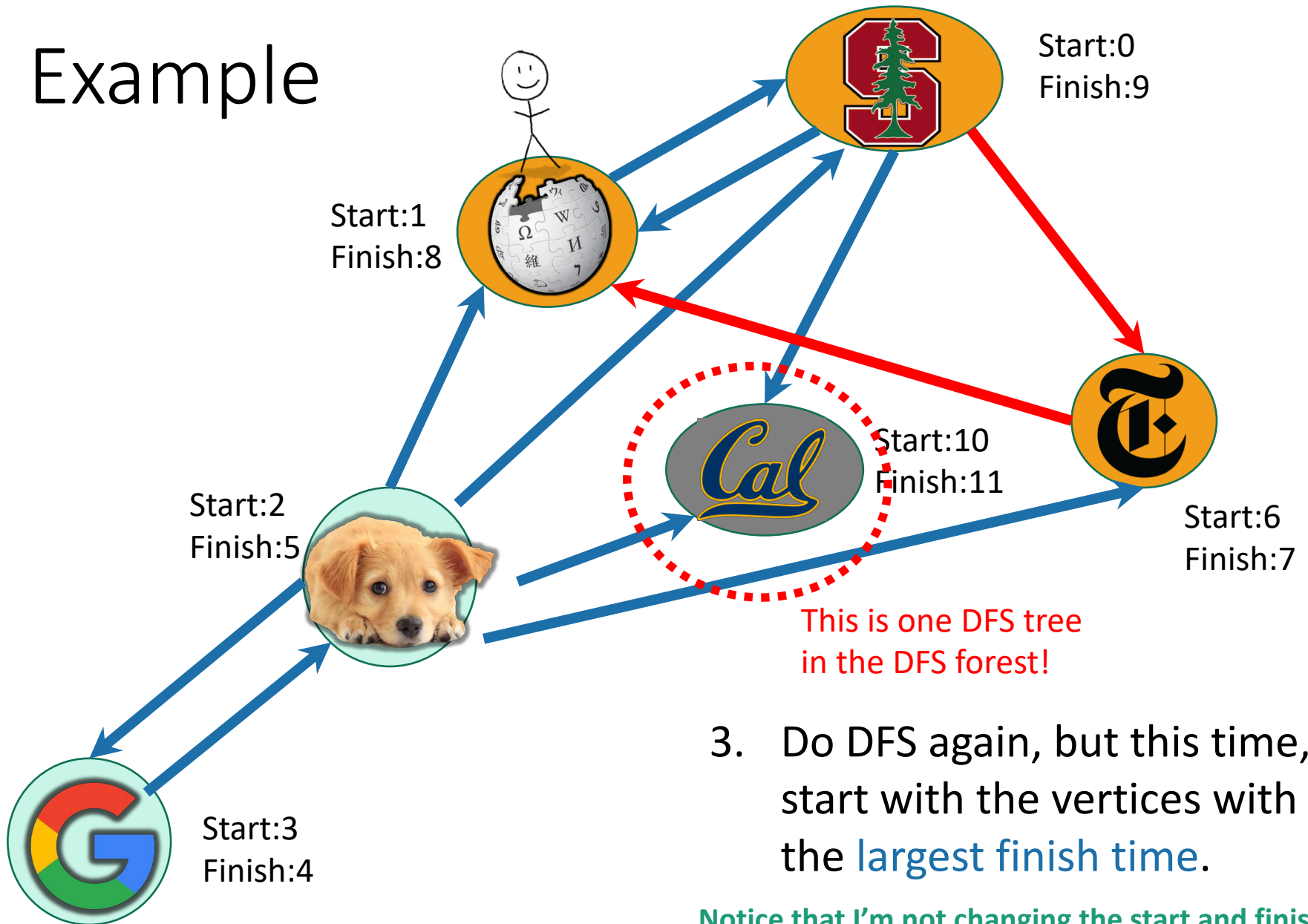


3. Do DFS again, but this time, start with the vertices with the **largest finish time**.

# Example



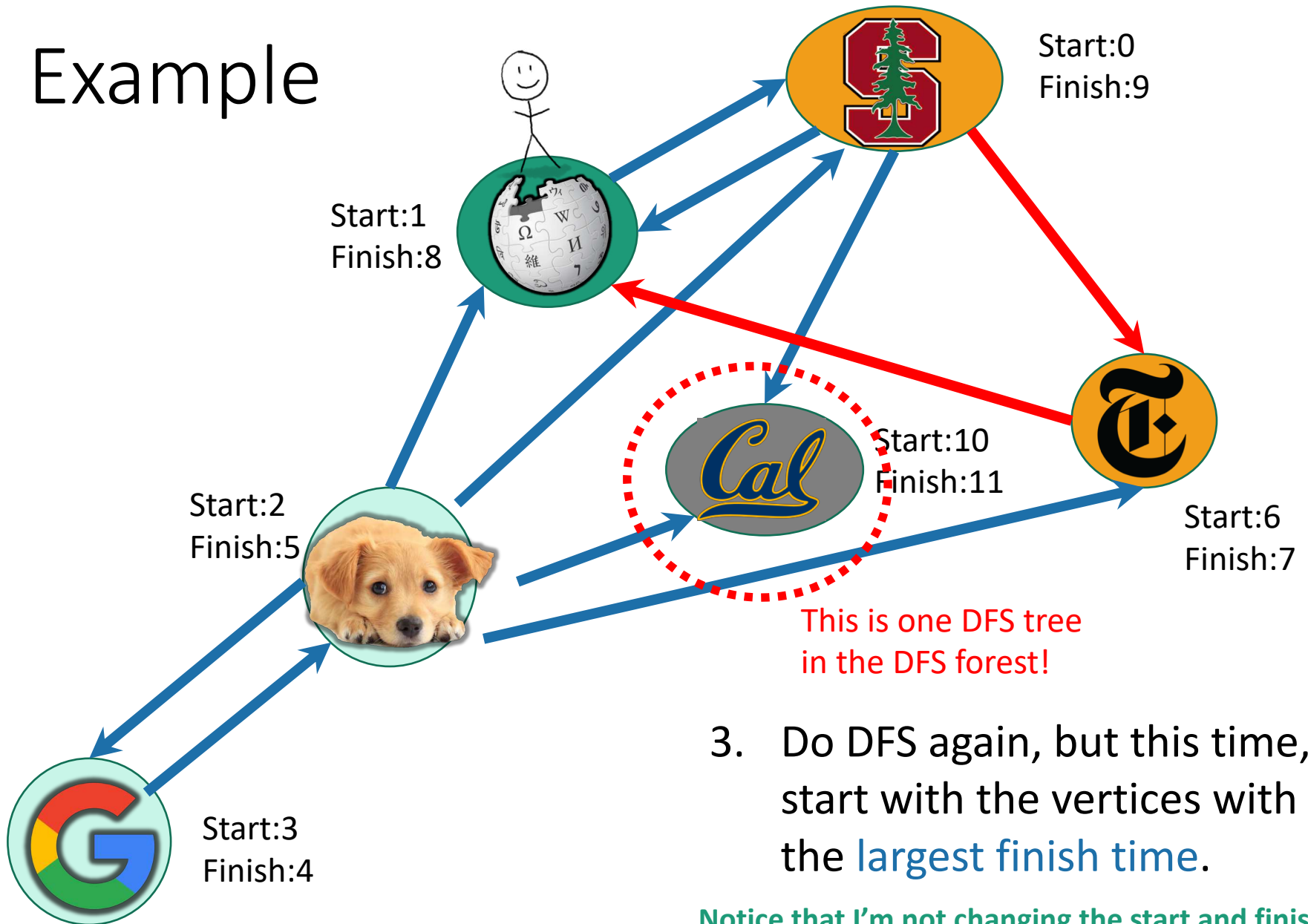
# Example



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

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

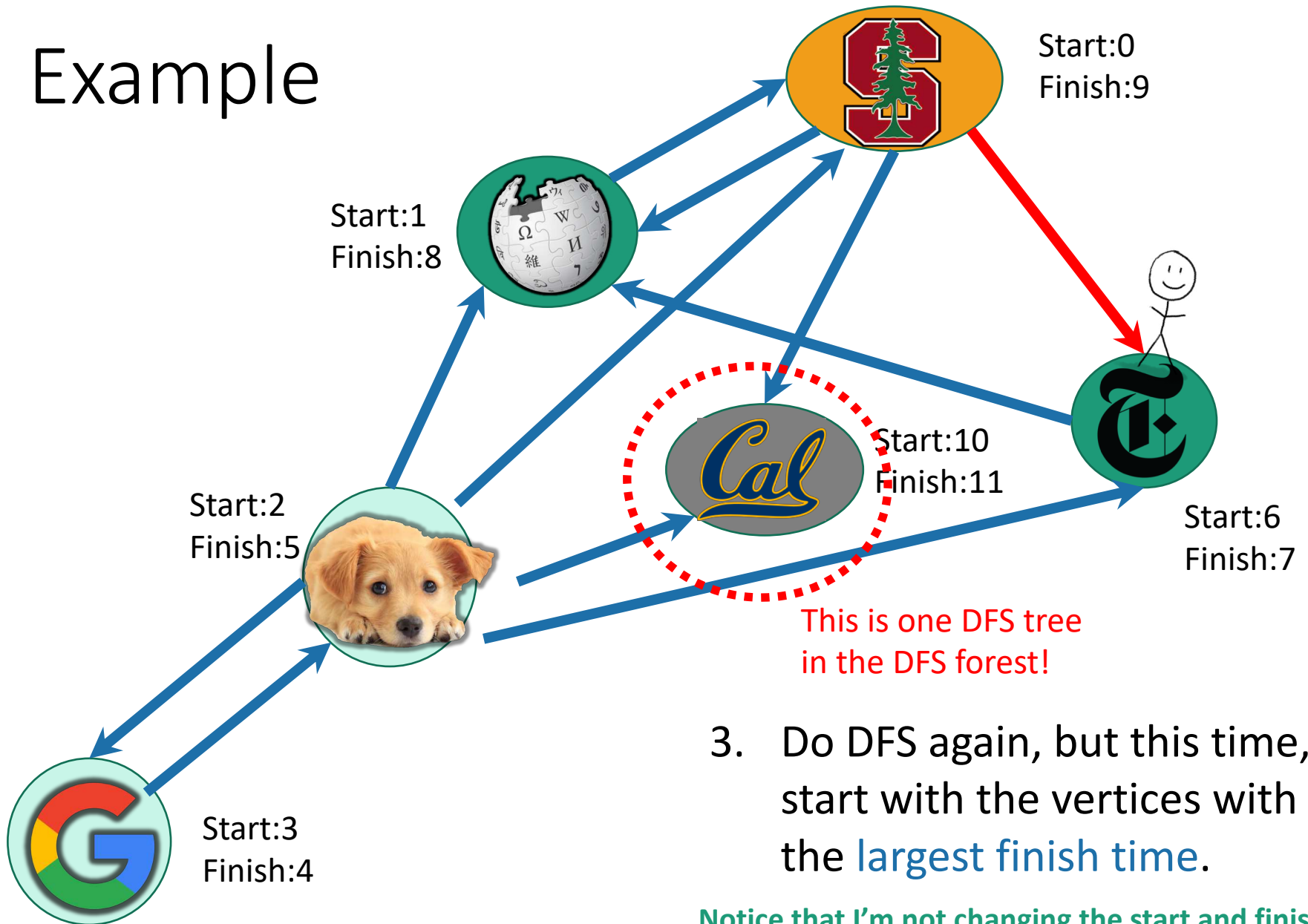
# Example



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

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

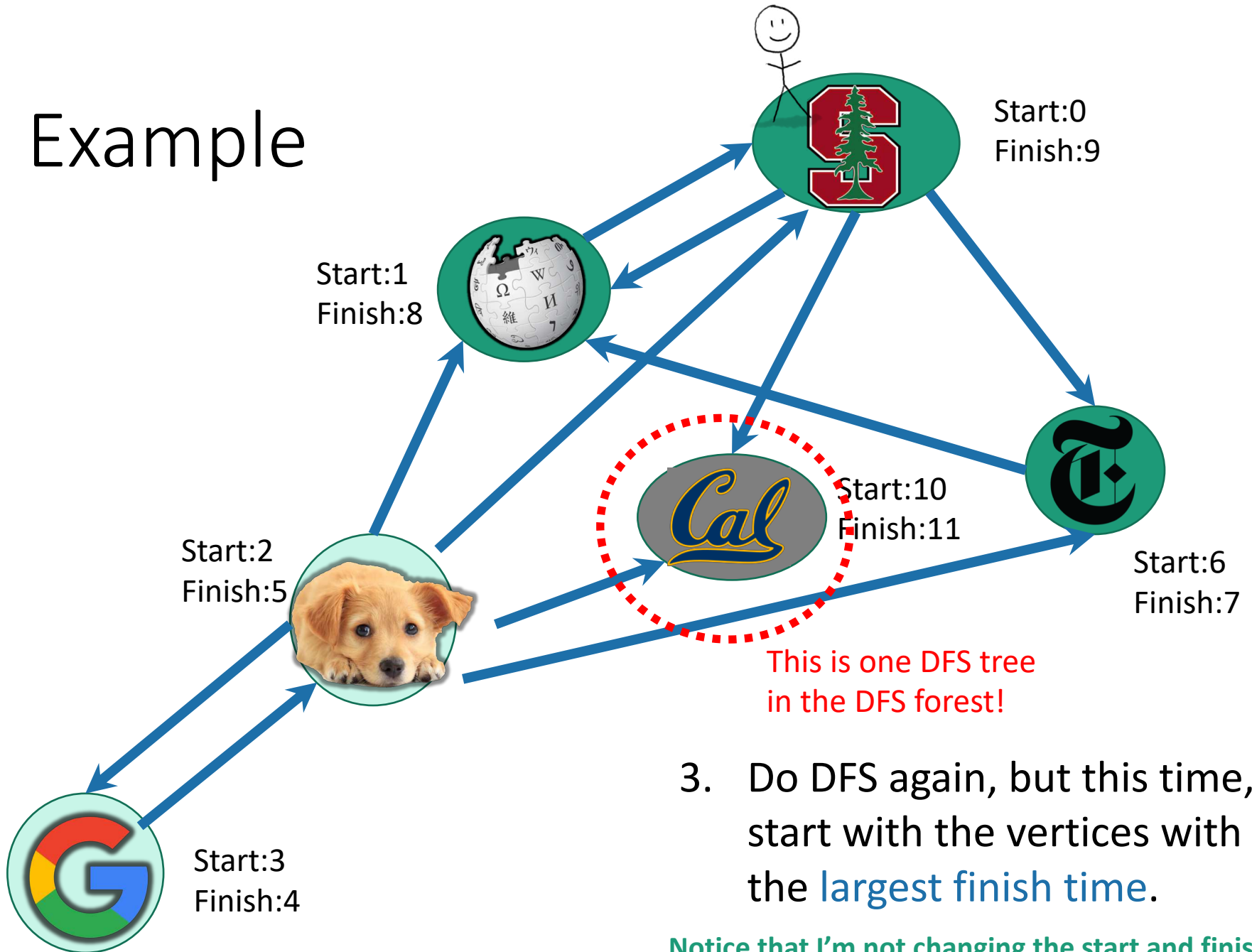
# Example



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

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

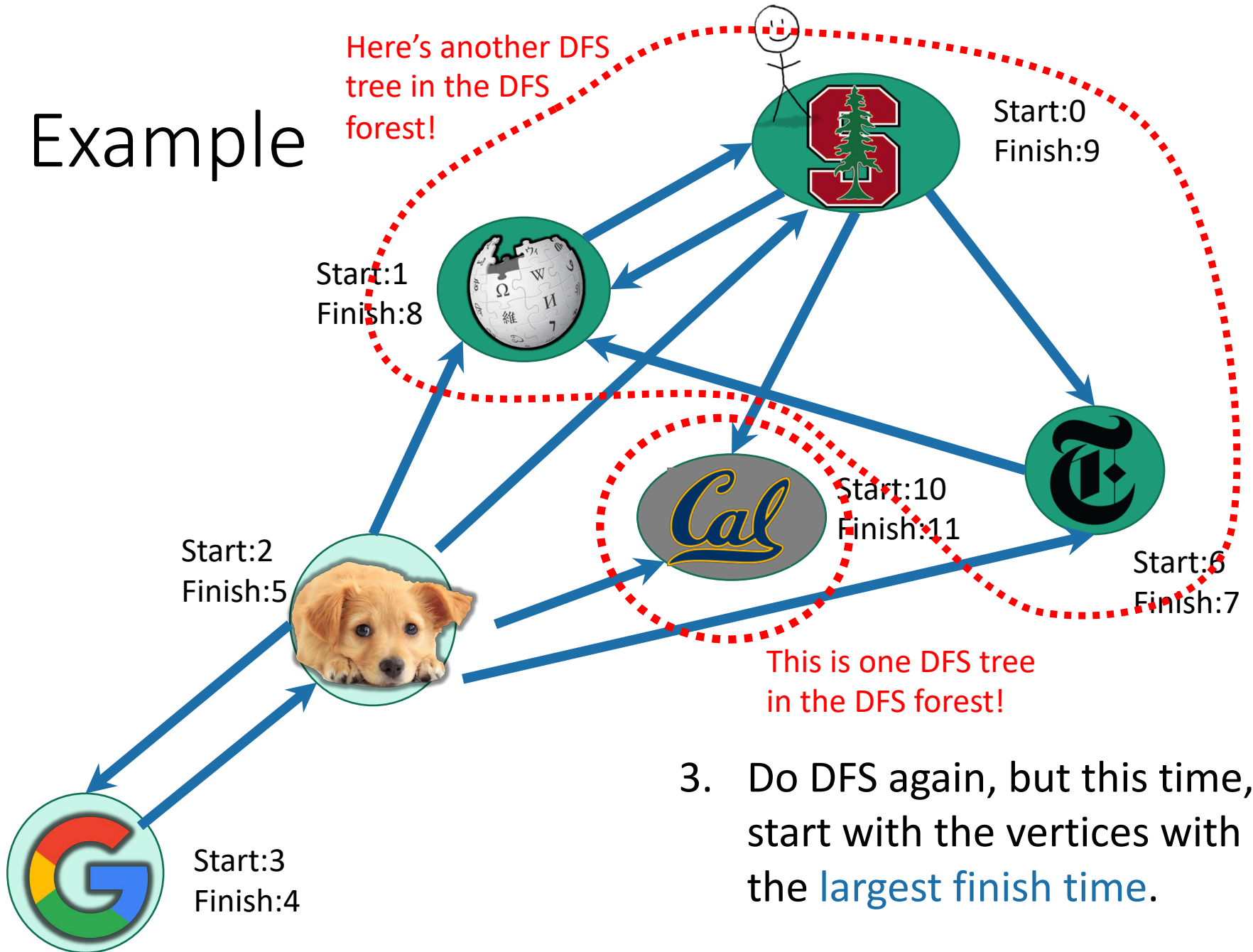
# Example



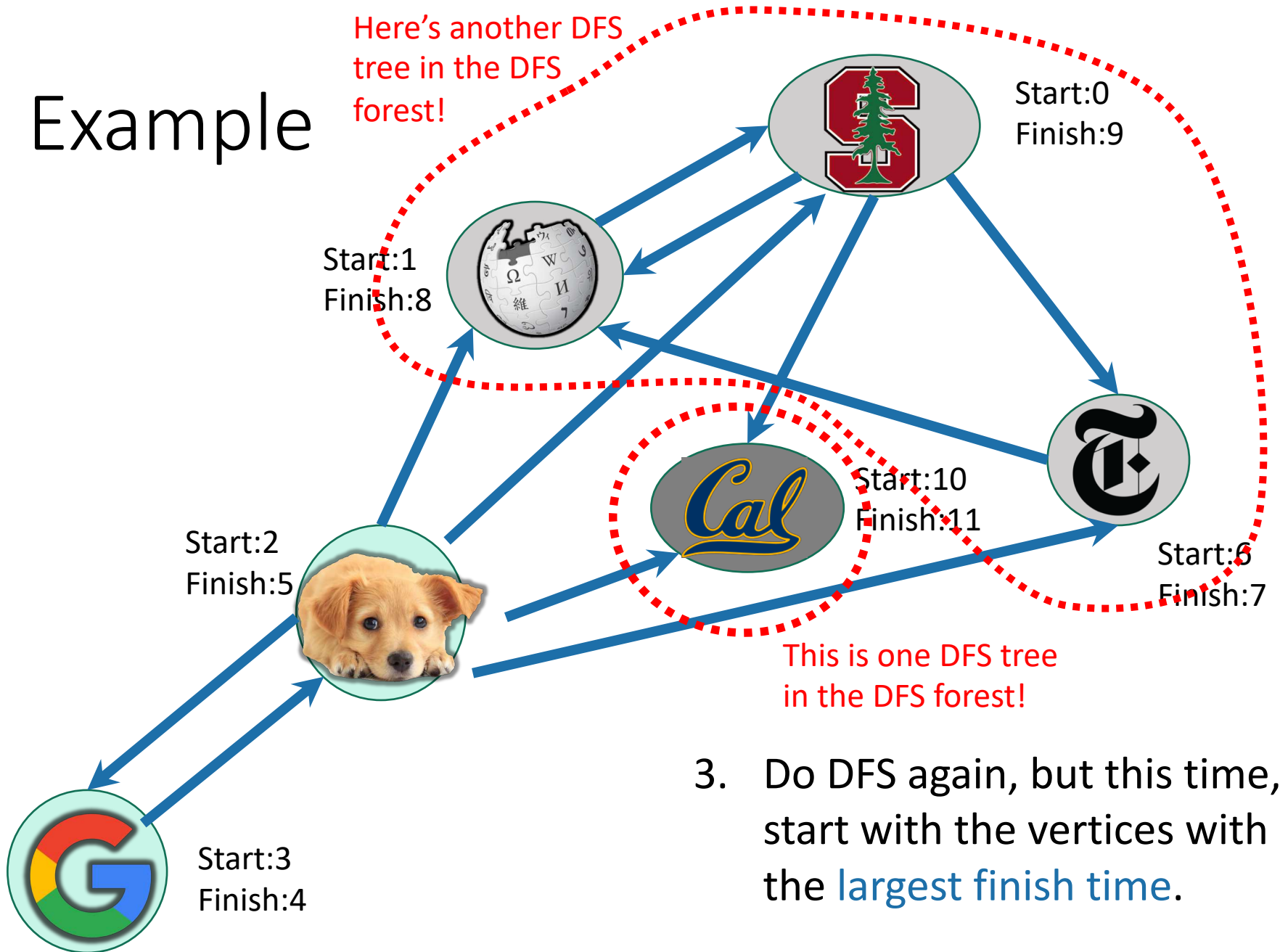
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

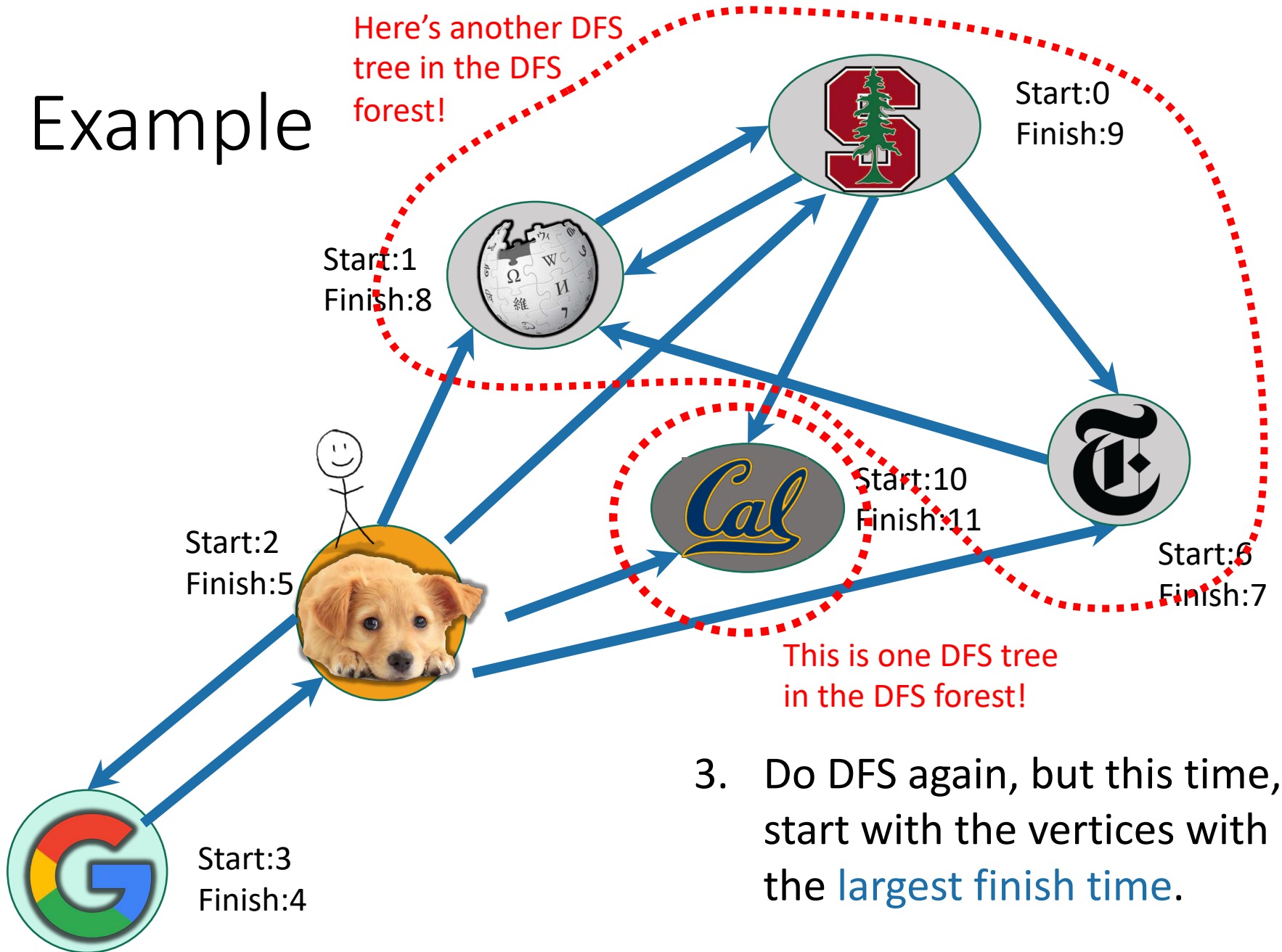
# Example



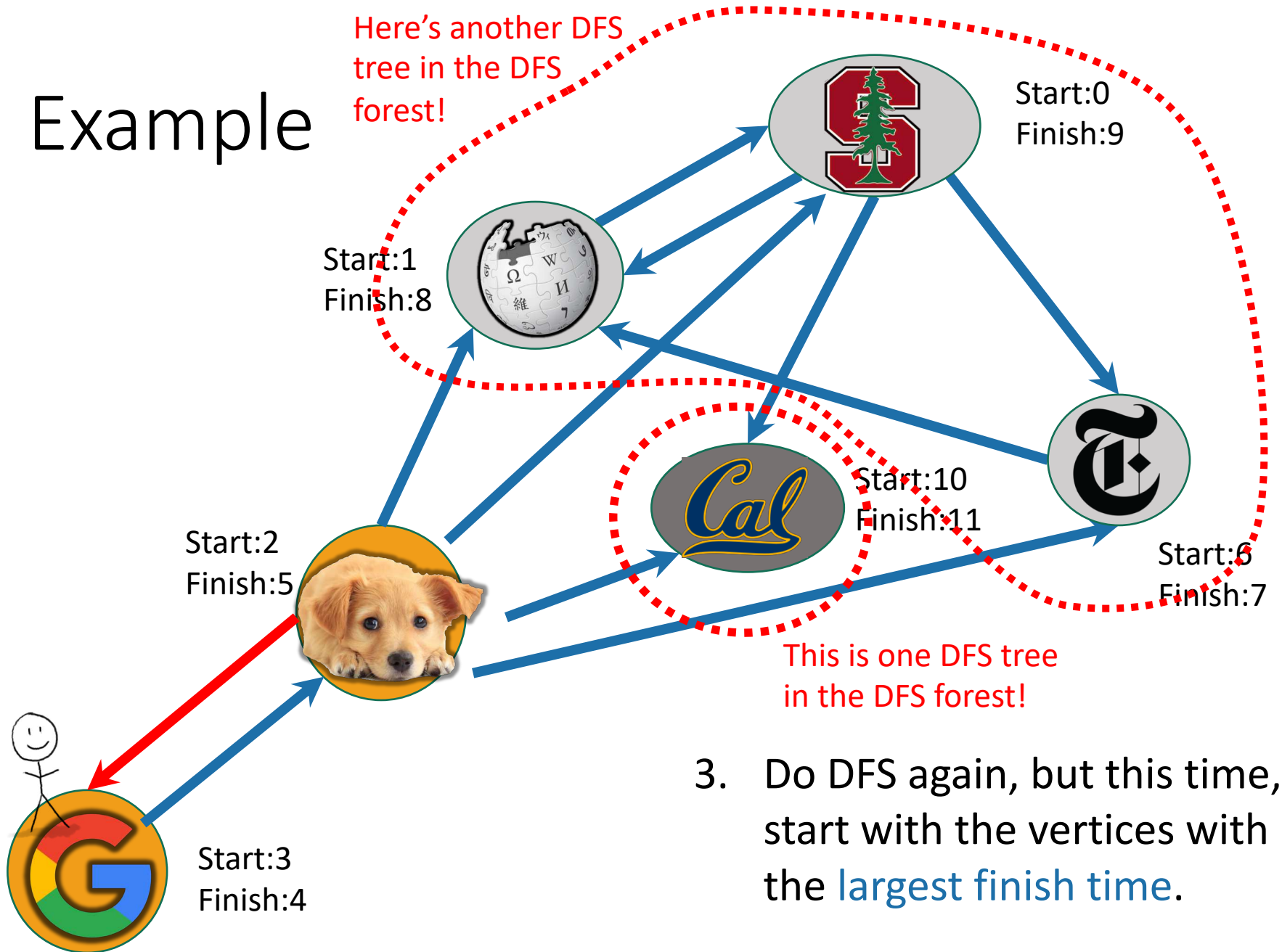
# Example



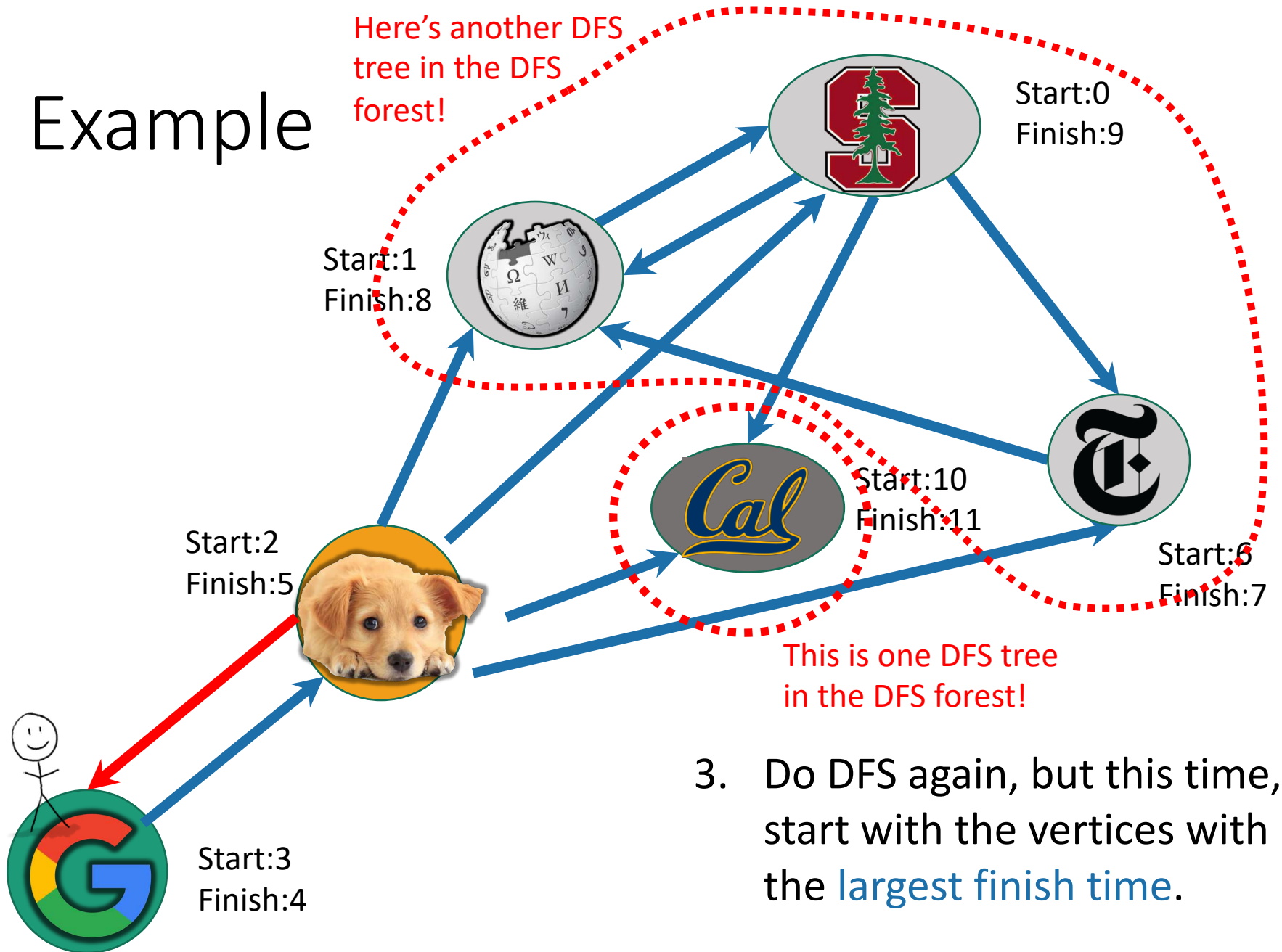
# Example



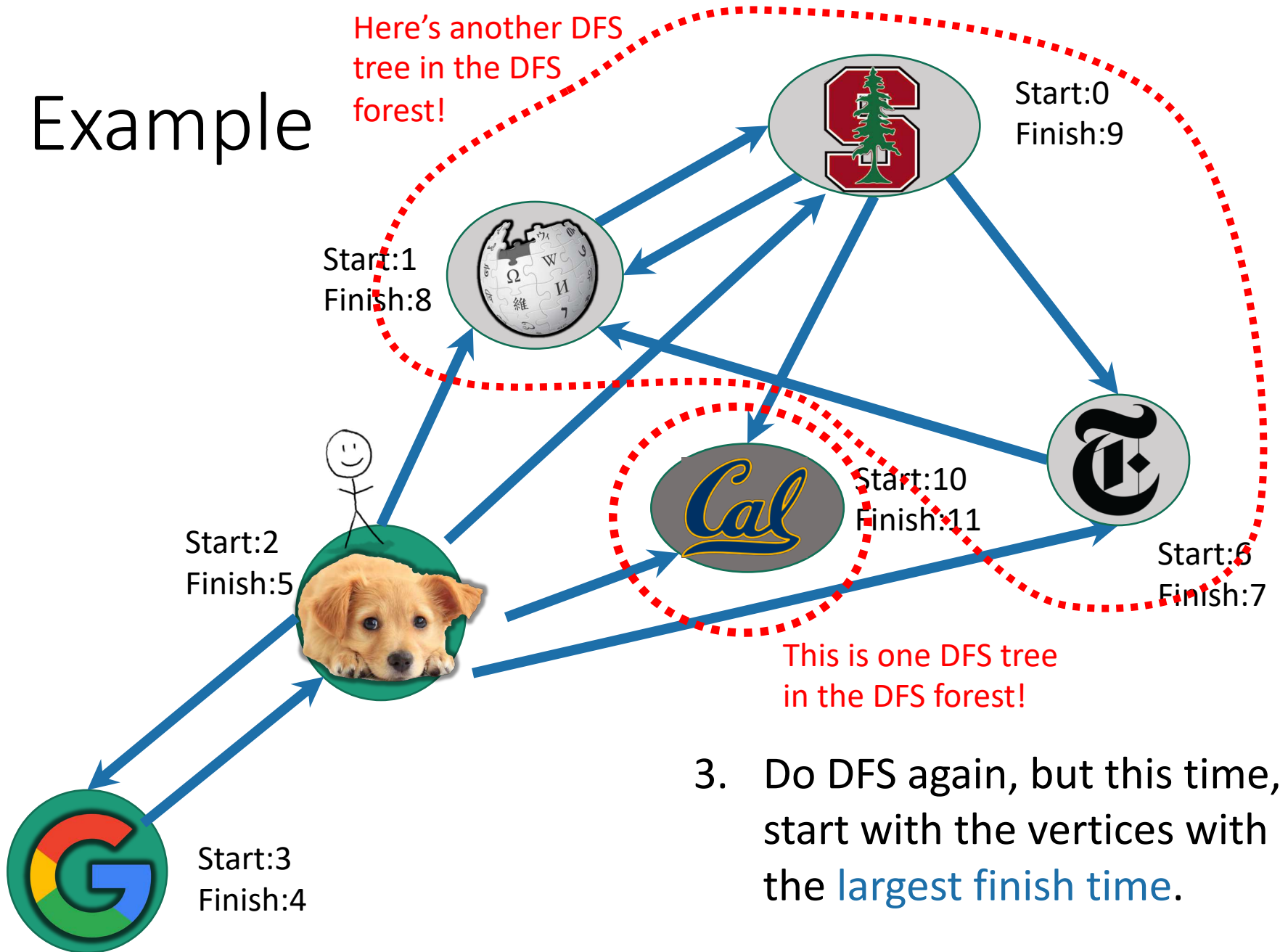
# Example



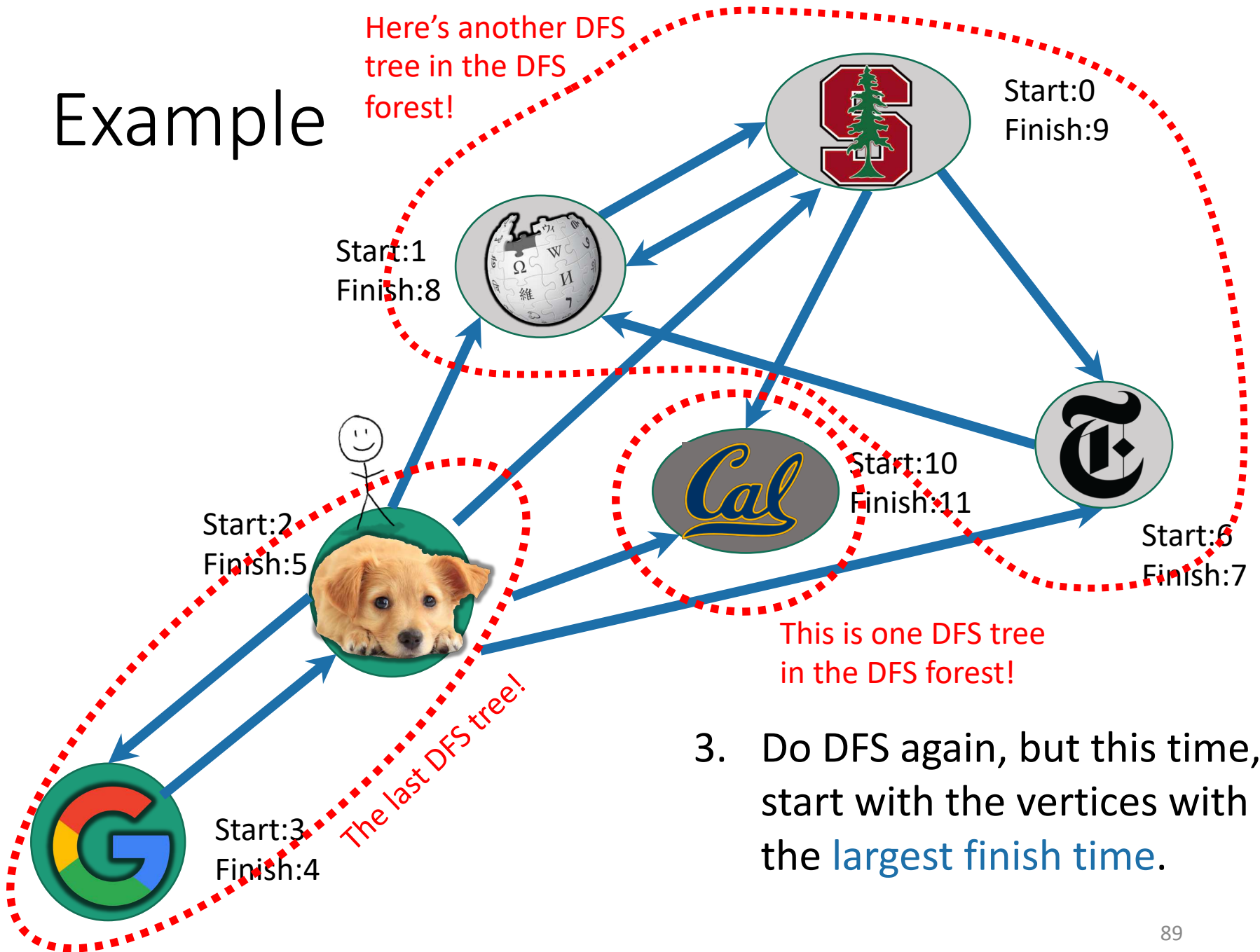
# Example



# Example

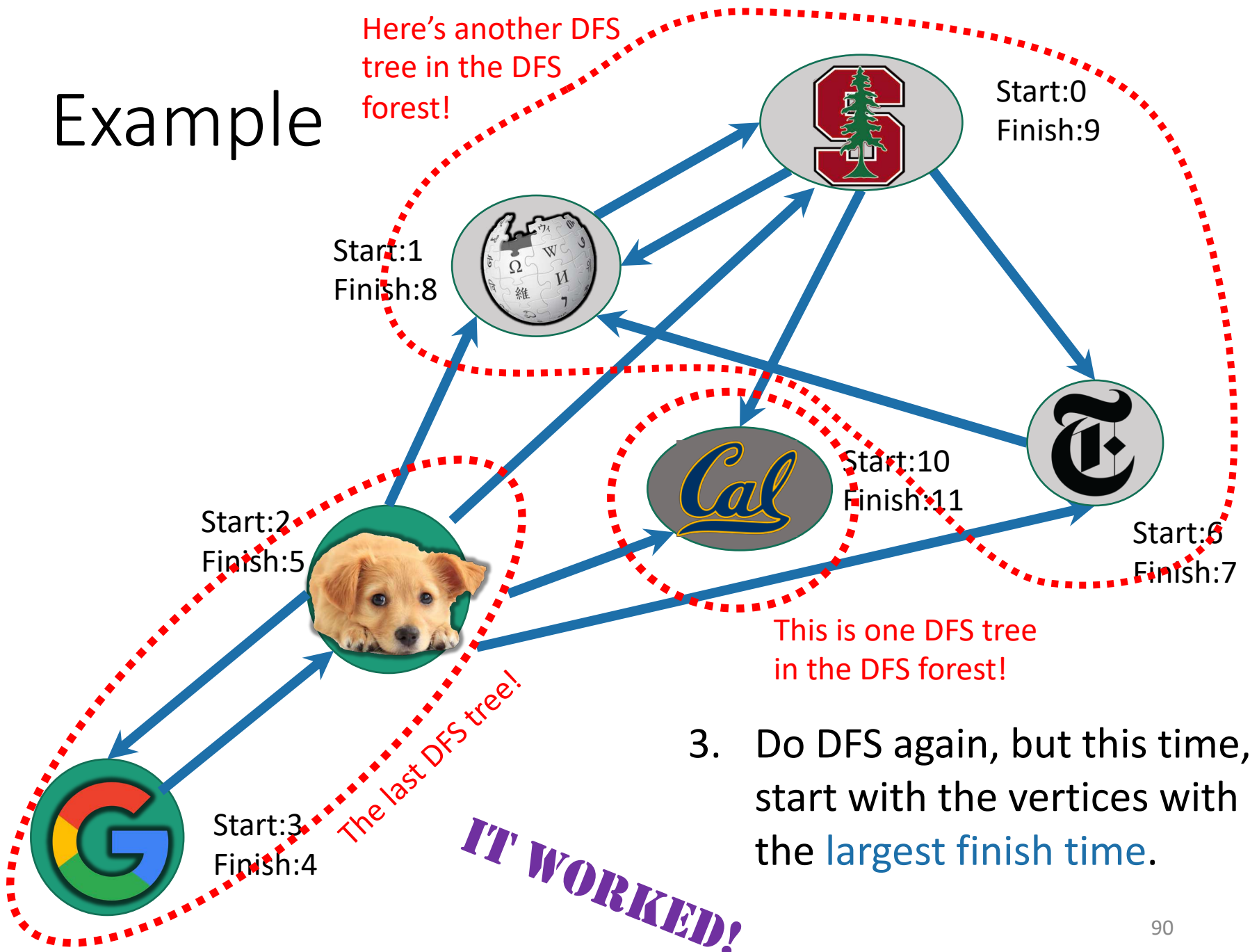


# Example



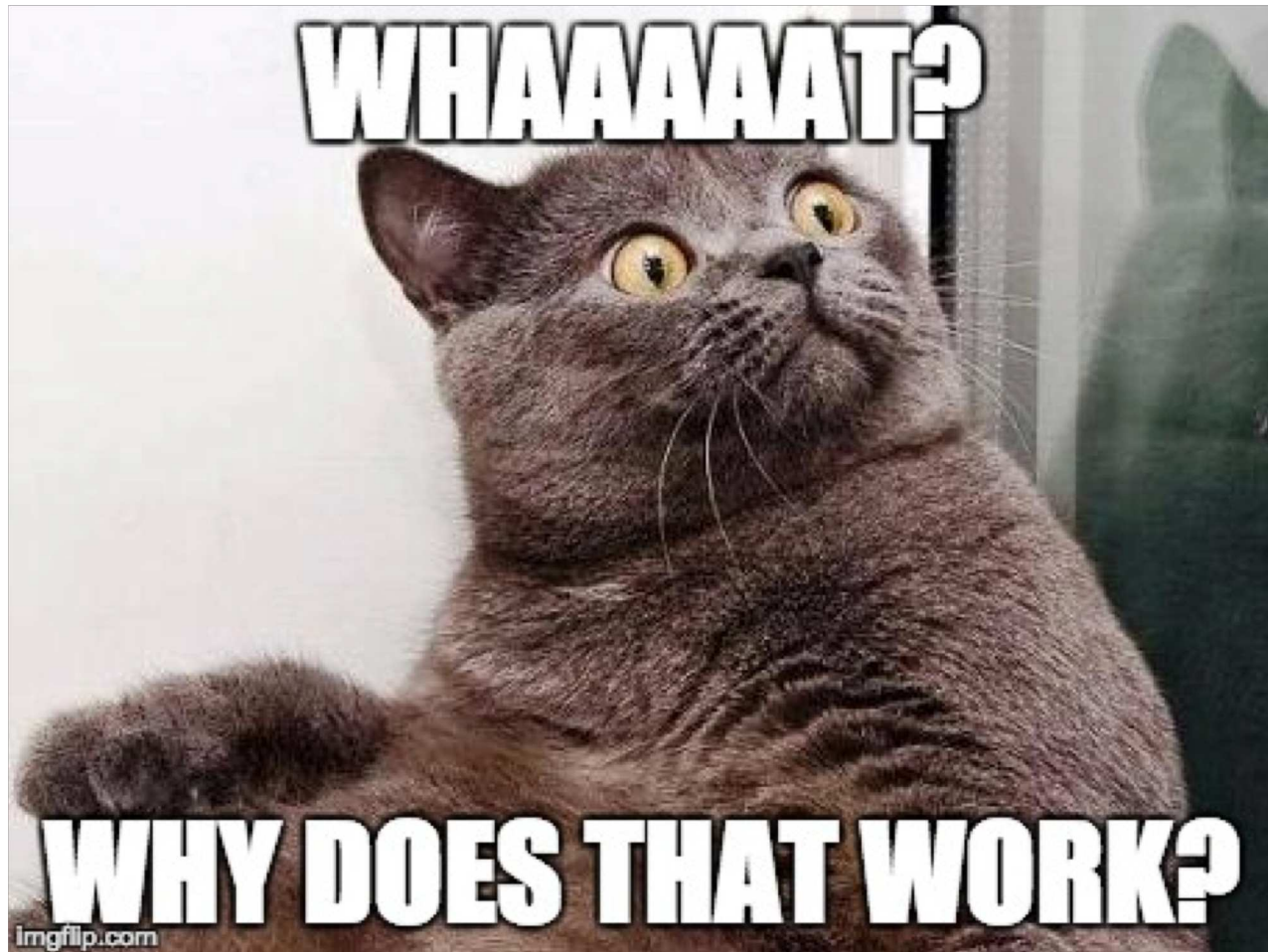
# Example

Here's another DFS tree in the DFS forest!

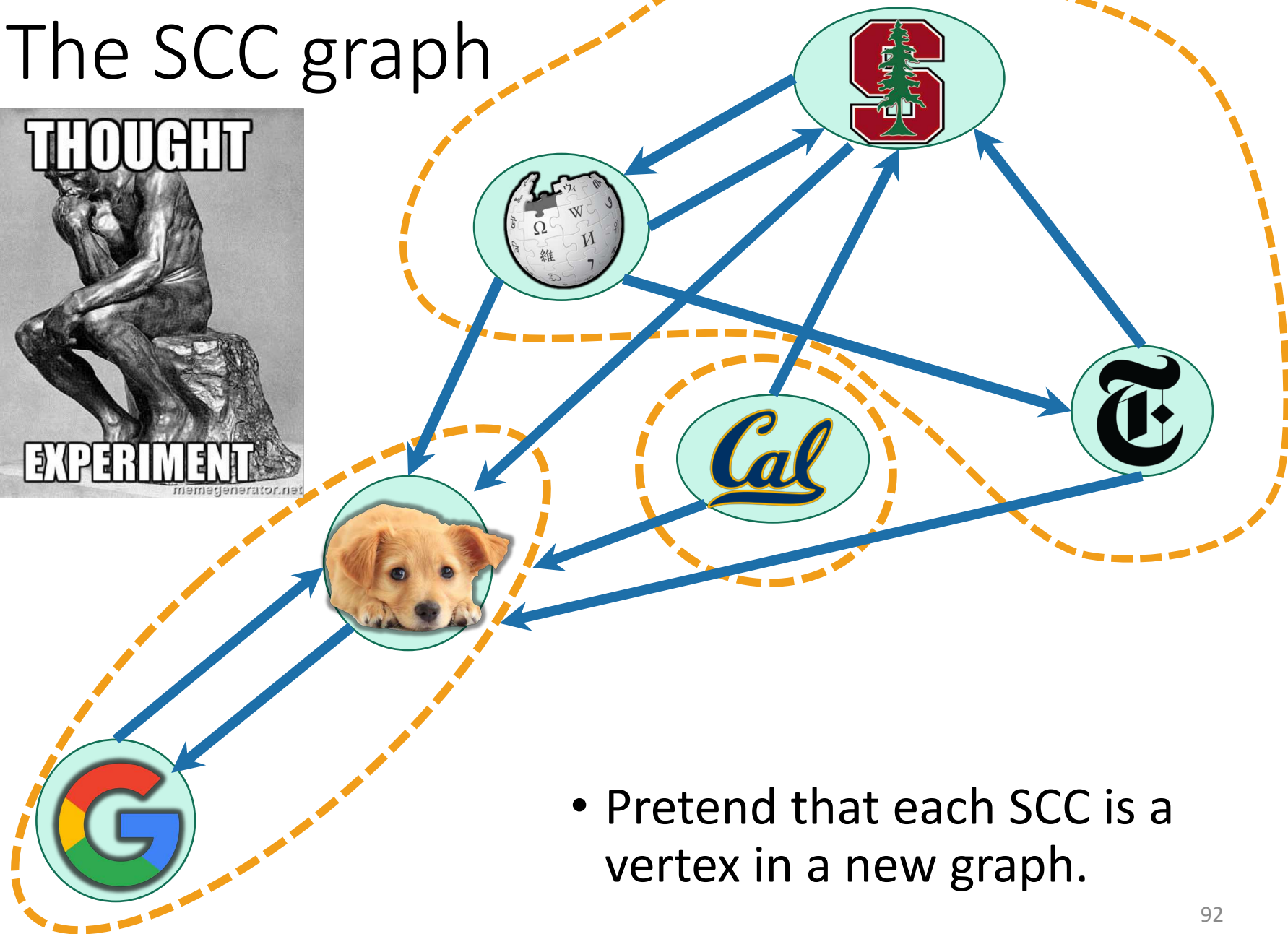
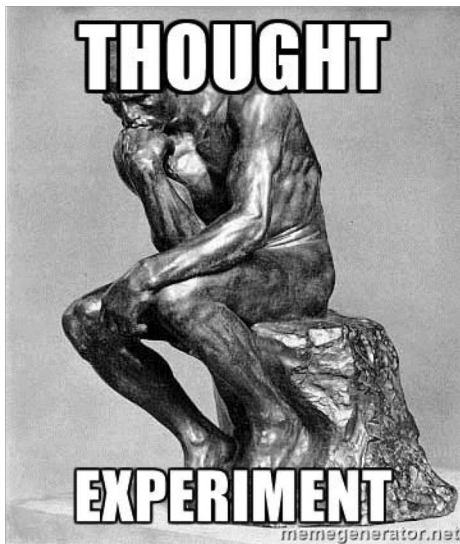


3. Do DFS again, but this time, start with the vertices with the largest finish time.

One question



# The SCC graph

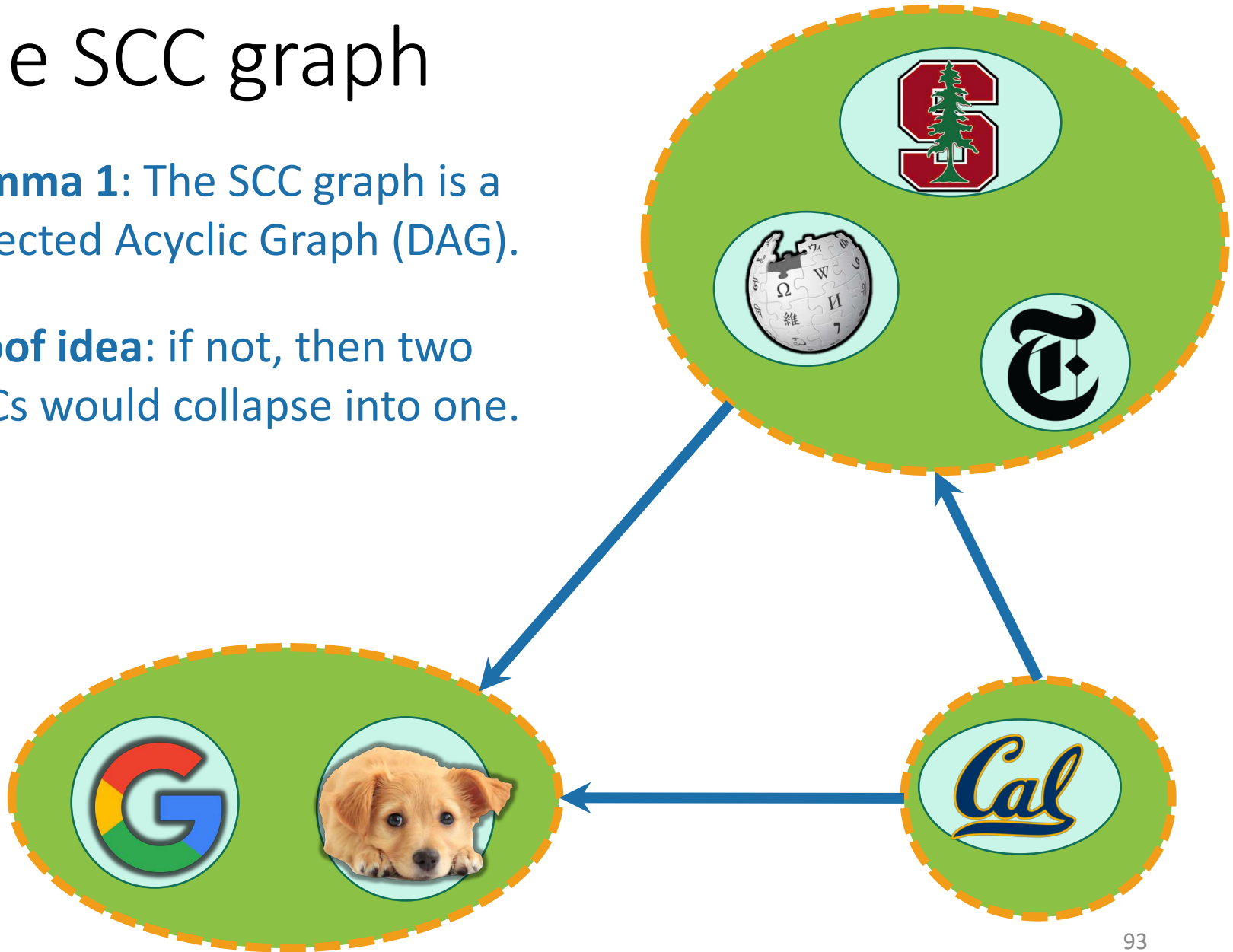


- Pretend that each SCC is a vertex in a new graph.

# 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.

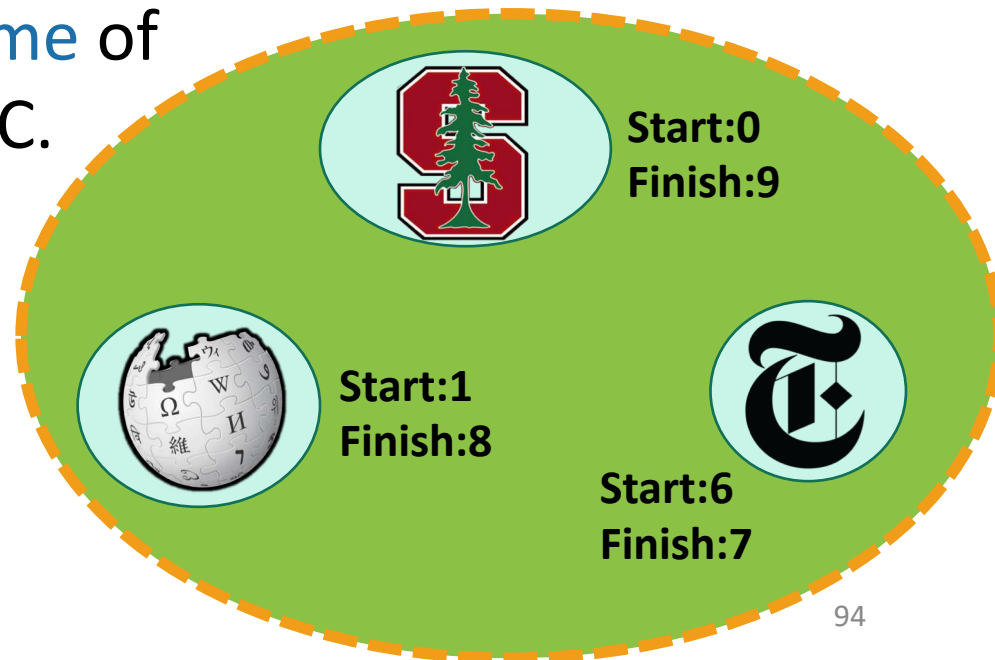


# Starting and finishing times in a SCC

Definitions:

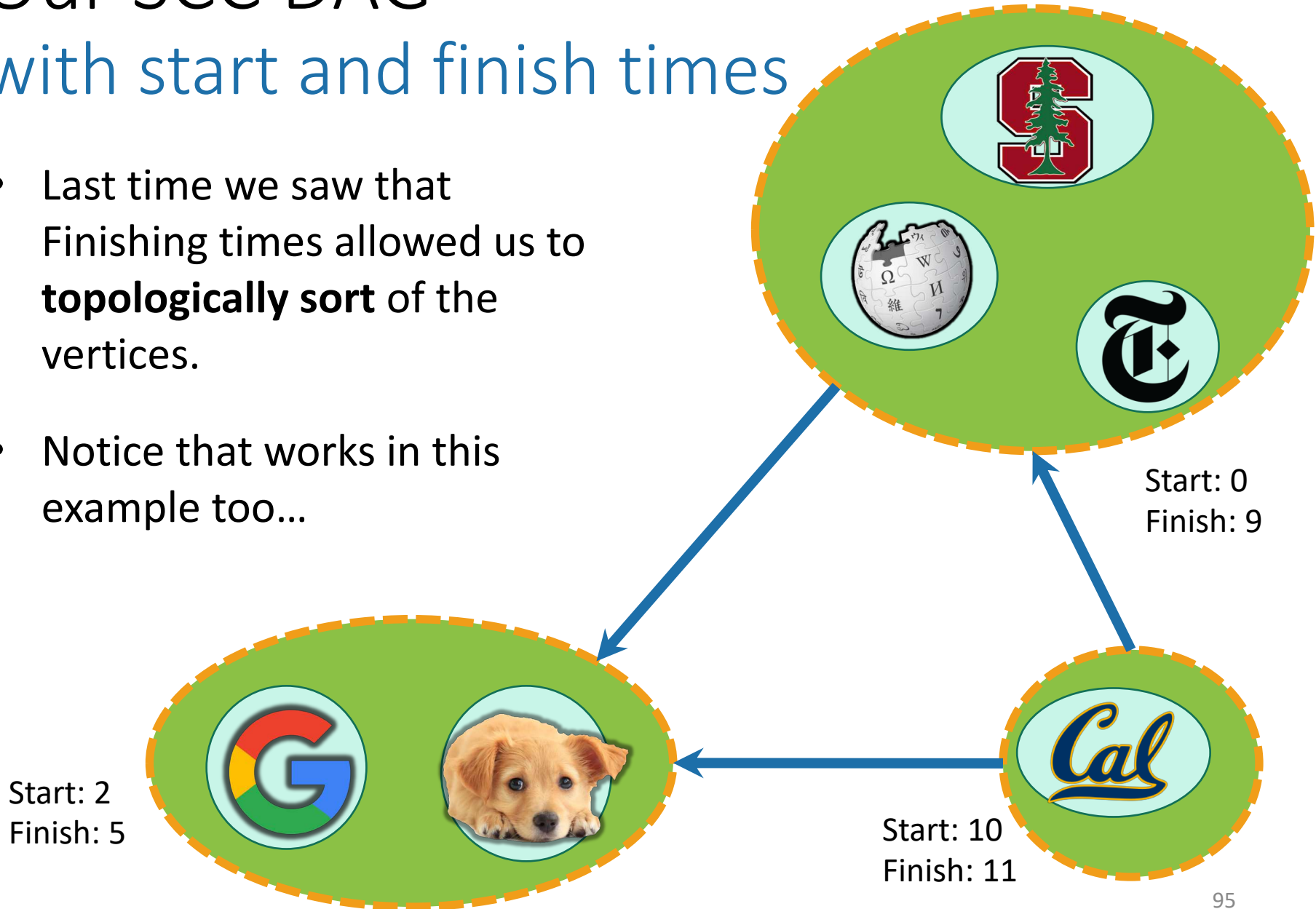
- The **finishing time** of a SCC is the **largest finishing time** of any element of that SCC.
- The **starting time** of a SCC is the **smallest starting time** of any element of that SCC.

Start: 0  
Finish: 9



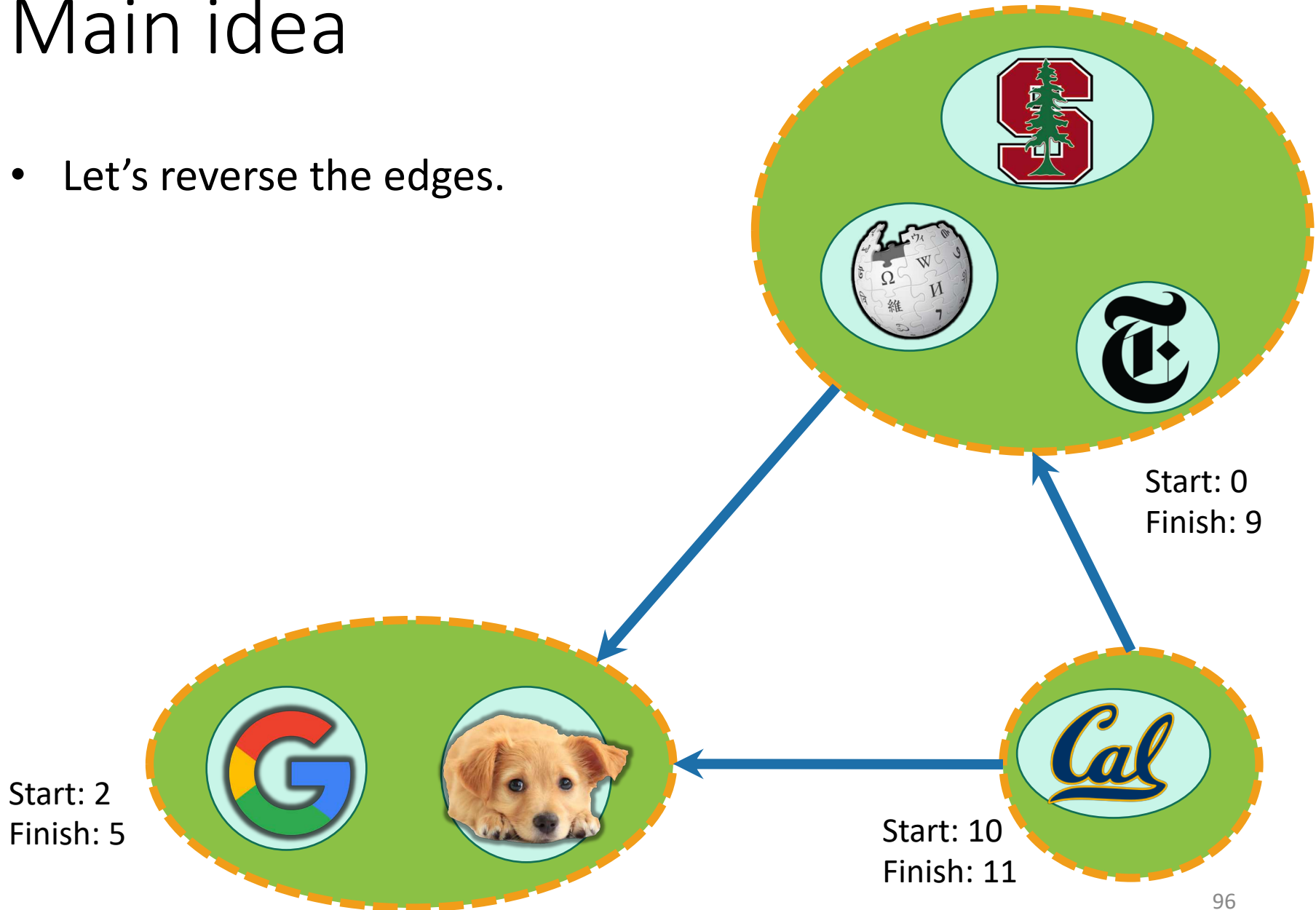
# Our SCC DAG with start and finish times

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



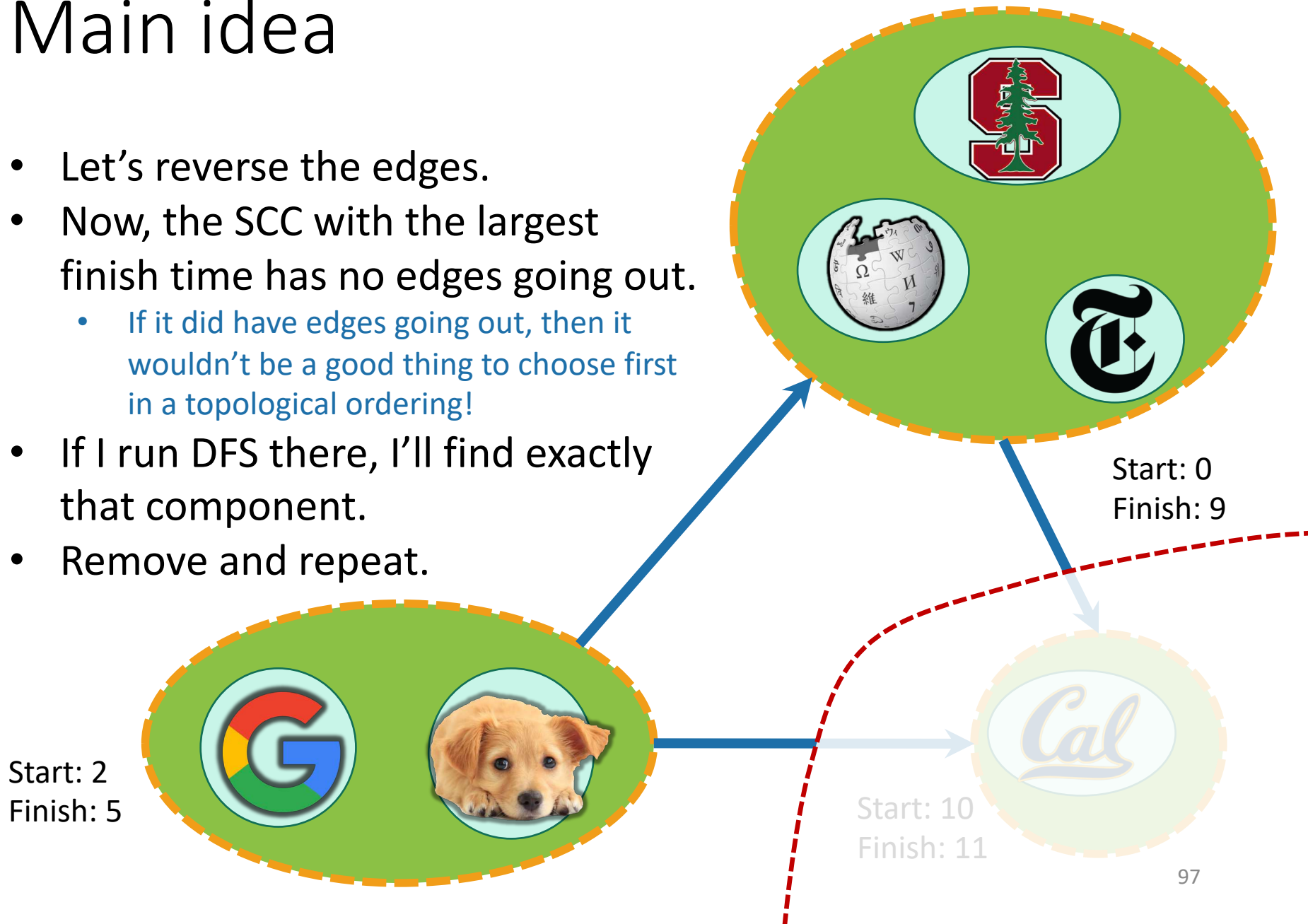
# Main idea

- Let's reverse the edges.



# Main idea

- Let's reverse the edges.
- Now, the SCC with the largest finish time has no edges going out.
  - If it did have edges going out, then it wouldn't be a good thing to choose first in a topological ordering!
- If I run DFS there, I'll find exactly that component.
- Remove and repeat.



Let's make this idea formal.

# Recall

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



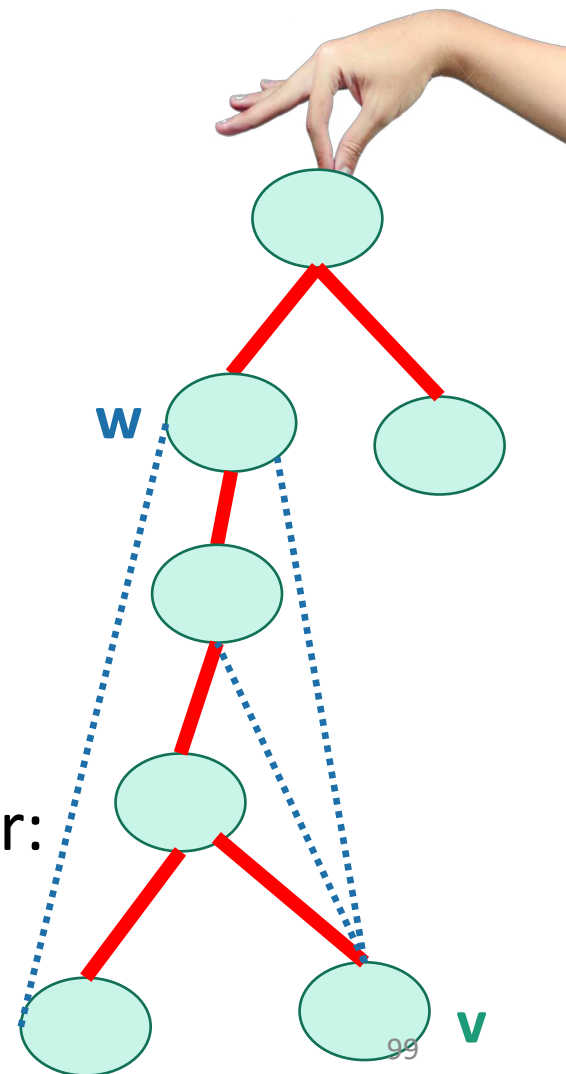
- If  $w$  is a descendent of  $v$  in this tree:



- If neither are descendants of each other:

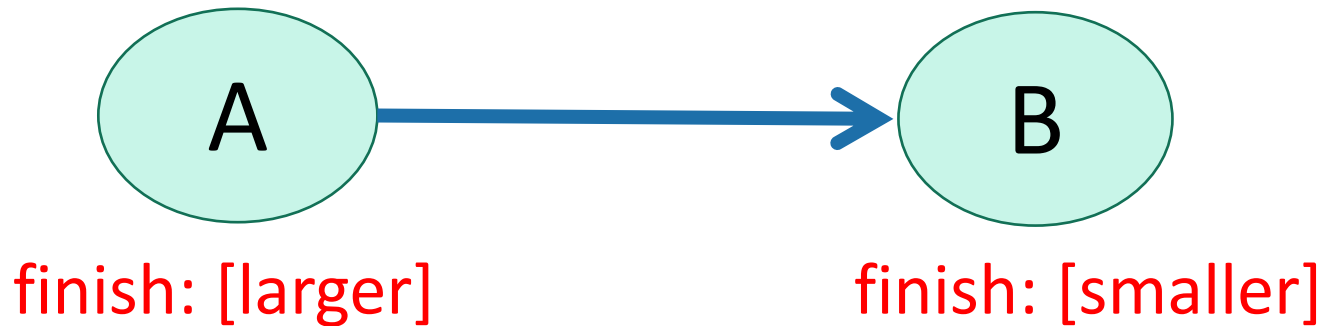


(or the other way around)



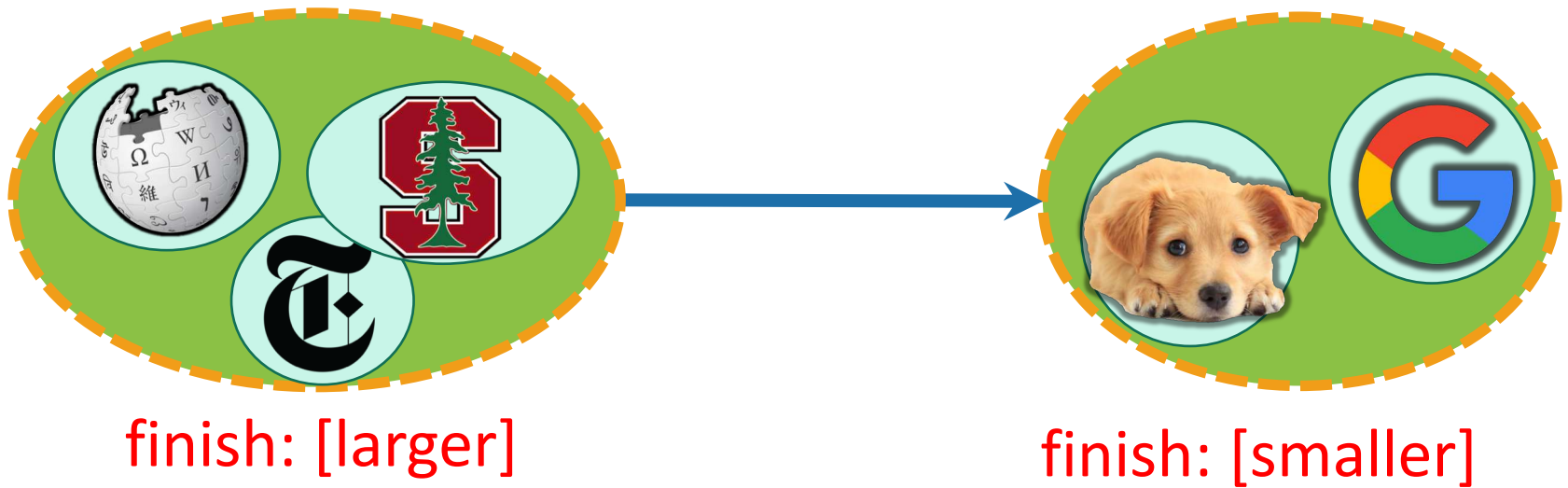
As we saw last time...

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



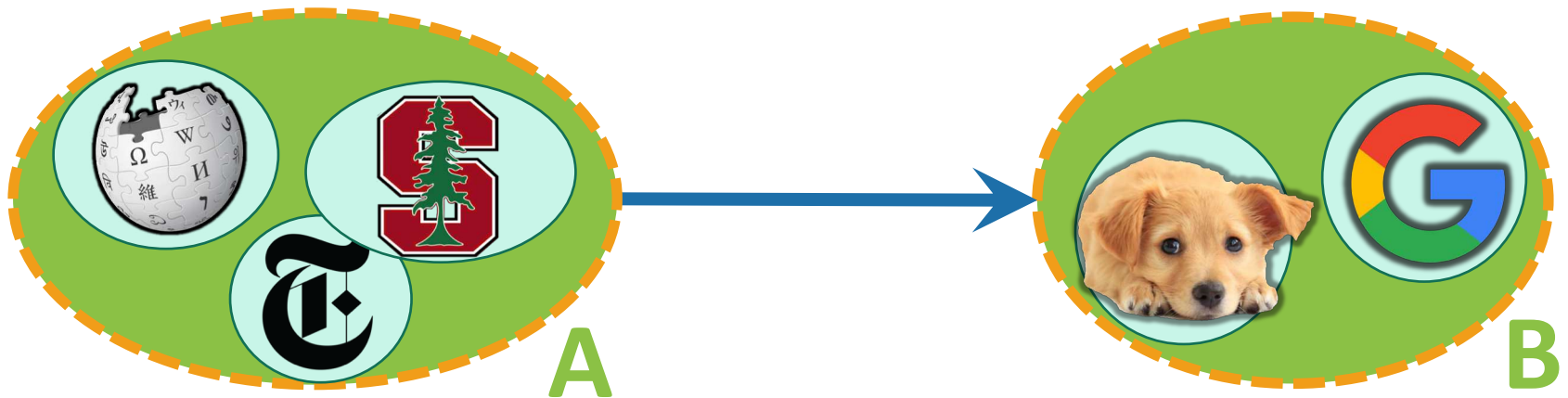
Same thing, in the SCC DAG.

- **Claim:** we'll always have



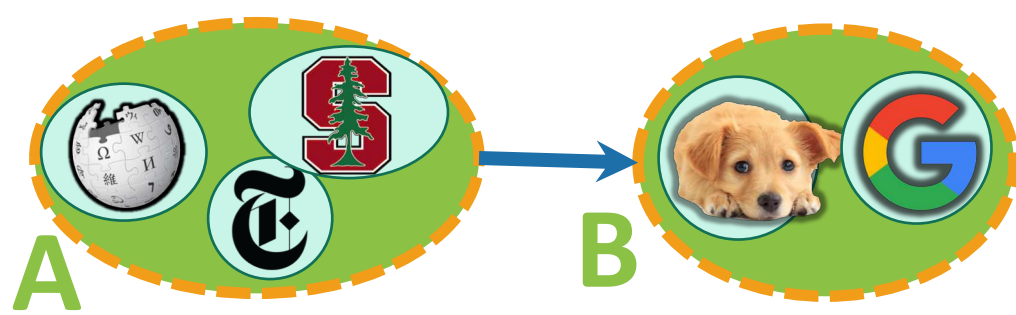
# Let's call it Lemma 2

- If there is an edge like this:



- Then  $A.\text{finish} > B.\text{finish}$ .

# Proof idea

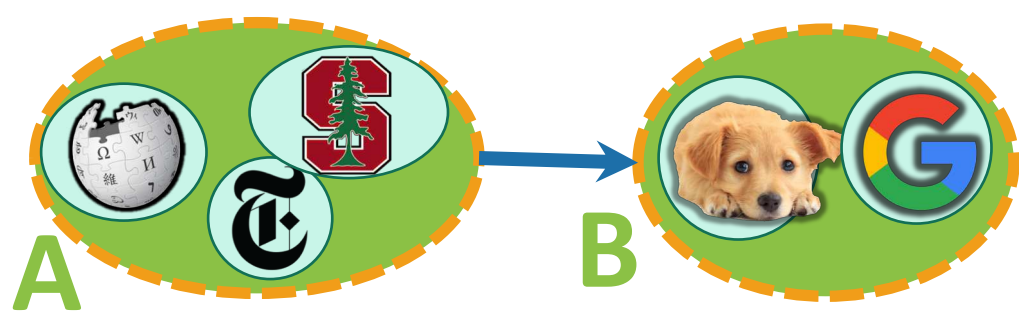


Want to show  $A.\text{finish} > B.\text{finish}$ .

- **Two cases:**

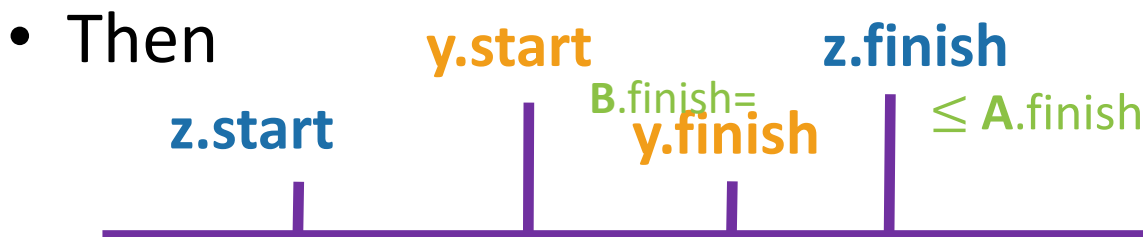
- We reached **A** before **B** in our first DFS.
- We reached **B** before **A** in our first DFS.

# Proof idea



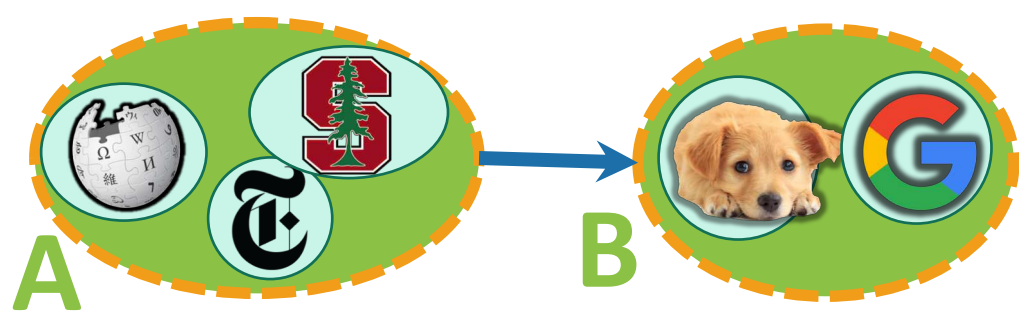
Want to show  $A.\text{finish} > B.\text{finish}$ .

- **Case 1:** We reached **A** before **B** in our first DFS.
- Say that:
  - **y** has the largest finish in **B**;  $B.\text{finish} = y.\text{finish}$
  - **z** was discovered first in **A**;  $A.\text{finish} \geq z.\text{finish}$
- Then:
  - Reach **A** before **B**
  - $\Rightarrow$  we will discover **y** via **z**
  - $\Rightarrow$  **y** is a descendant of **z** in the DFS forest.



aka,  
 **$A.\text{finish} > B.\text{finish}$**

# Proof idea

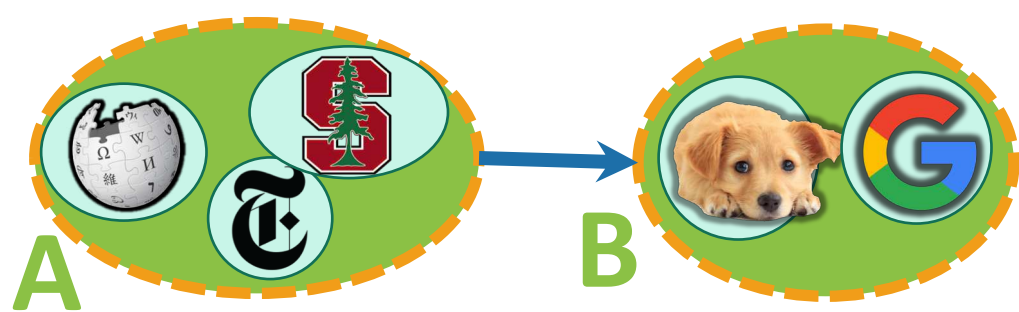


Want to show  $A.\text{finish} > B.\text{finish}$ .

- **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,  
 $A.\text{finish} > B.\text{finish}$

# Proof idea



Want to show  $A.\text{finish} > B.\text{finish}$ .

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

**$A.\text{finish} > B.\text{finish}$**

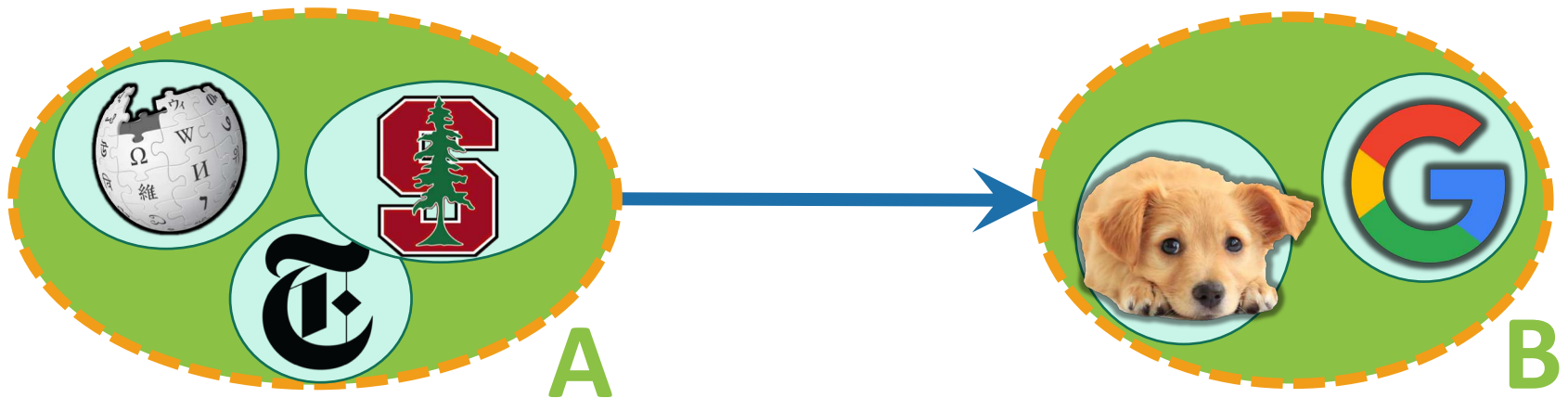
which is what we wanted to show.



Notice: this is exactly the same two-case argument that we did last time for topological sorting, just with the SCC DAG!

This establishes:  
Lemma 2

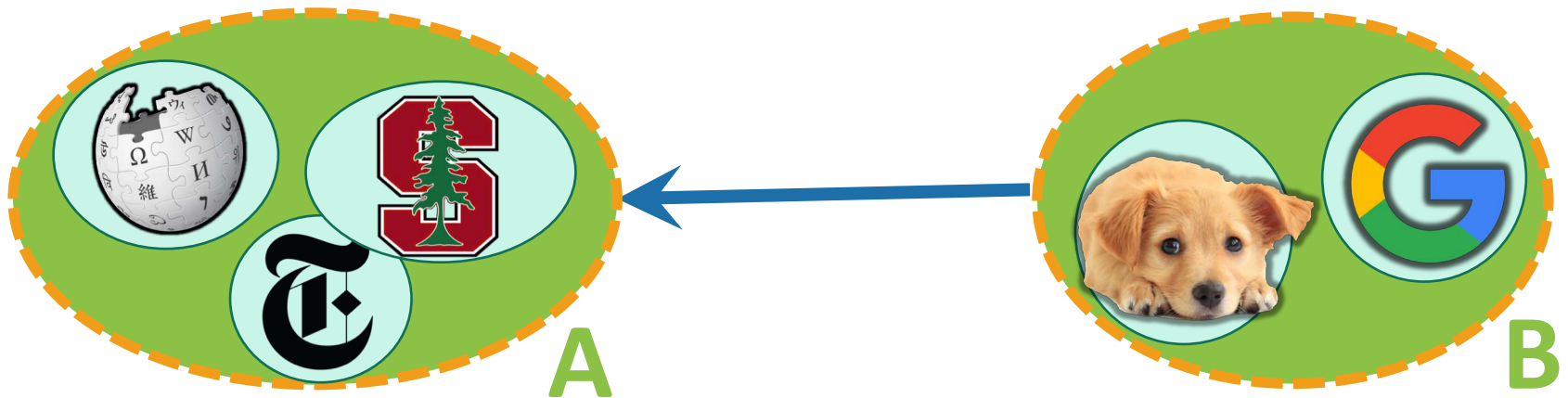
- If there is an edge like this:



- Then  $A.\text{finish} > B.\text{finish}$ .

This establishes:  
**Corollary 1**

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

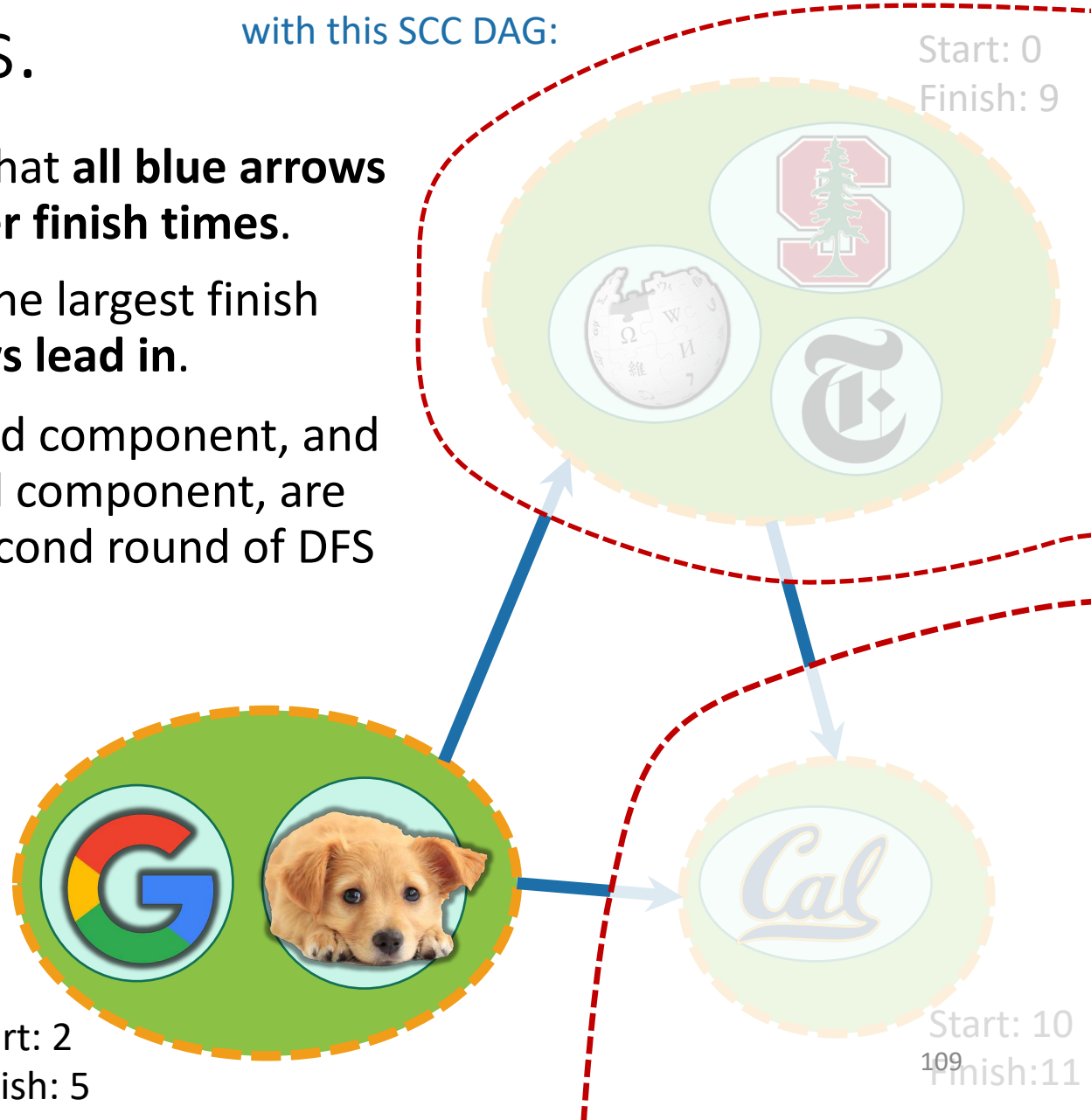


- Then  $A.\text{finish} > B.\text{finish}$ .

# Now we see why this finds SCCs.

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.
- **Base case: ( $t=0$ )**
  - The first 0 trees found in the reversed DFS forest are the 0 SCCs with the largest finish times. **(TRUE)**

# Inductive step [drawing on board to supplement]

- **Assume by induction that the first  $t$  trees are the last-finishing SCCs.**
- Consider the  $(t+1)^{\text{st}}$  tree produced, suppose the root is  $x$ .
- Suppose that  $x$  lives in the SCC  $A$ .
- Then  $A.\text{finish} > B.\text{finish}$  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  $(t+1)^{\text{st}}$  tree is the SCC with the  $(t+1)^{\text{st}}$  biggest finish time.**

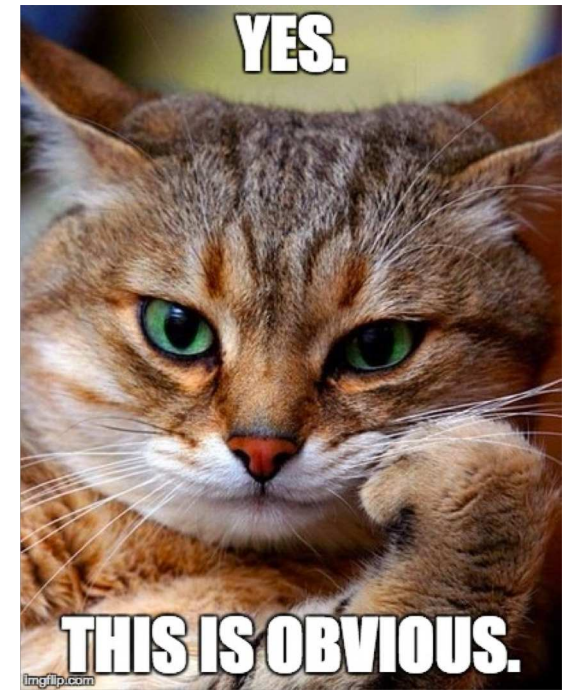
# Formally, we prove it by induction

- **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.
- **Base case:** *[done]*
- **Inductive step:** *[done]*
- **Conclusion:** The second (reversed) DFS forest contains all the SCCs as its trees!
  - (This is the **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

- Breadth First Search can be used to find shortest paths in unweighted graphs!
- Depth First Search reveals a very useful structure!
  - We saw last week 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

- Dijkstra's algorithm!

## **BEFORE** Next time

- Do the midterm!
- Pre-lecture exercise: weighted graphs!