Lecture 10

Finding strongly connected components

Announcements

- HW4 Due Friday
- Next Week: Midterm 2!
  - Thursday 11/6, 9am-10:20am
    - 60 minutes for the exam, same as last time
    - Keep an eye out for a logistics post on Ed
  - Covers through Lecture 11 (Thursday)
    - Emphasis on stuff since the last midterm
    - We are aware that you won't have a chance to do HW on Lectures 10 or 11 yet
  - We'll post a practice exam soon

We hear your feedback about the first midterm!

- We will try to make the second midterm shorter (in terms of number/length of problems)
- It will have short questions, rather than a big long question worth a lot of points
- Unfortunately, we need to stick to 60 minutes for the midterm
  - Our class is 80 minutes long
  - We need time for ID-checking at the end (AIWG says we must check IDs)
  - Multiple out-of-class midterms are logistically challenging for a class of this size
  - In future years we will try to figure something else out, but for this year unfortunately this is what it is ☹

We also hear your feedback about wanting more practice problems!

- We agree!
- There will be extra problems on the Section material for extra practice
- You can also take a look at our textbook, and the recommended textbooks
  - See Resources->Textbooks on the course website
  - CLRS is free online via the Stanford library and has tons of practice problems

Last time

- Breadth-first and depth-first search
- Plus, applications!
  - Topological sorting
  - In-order traversal of BSTs
  - Finding Bacon numbers
  - ~~Testing for bipartite-ness~~ (we skipped this due to time, but either go back to the slides or figure out on your own how to use BFS to do this!)
- The key was paying attention to the structure of the tree that these search algorithms implicitly build.

Today

- One more application of DFS:

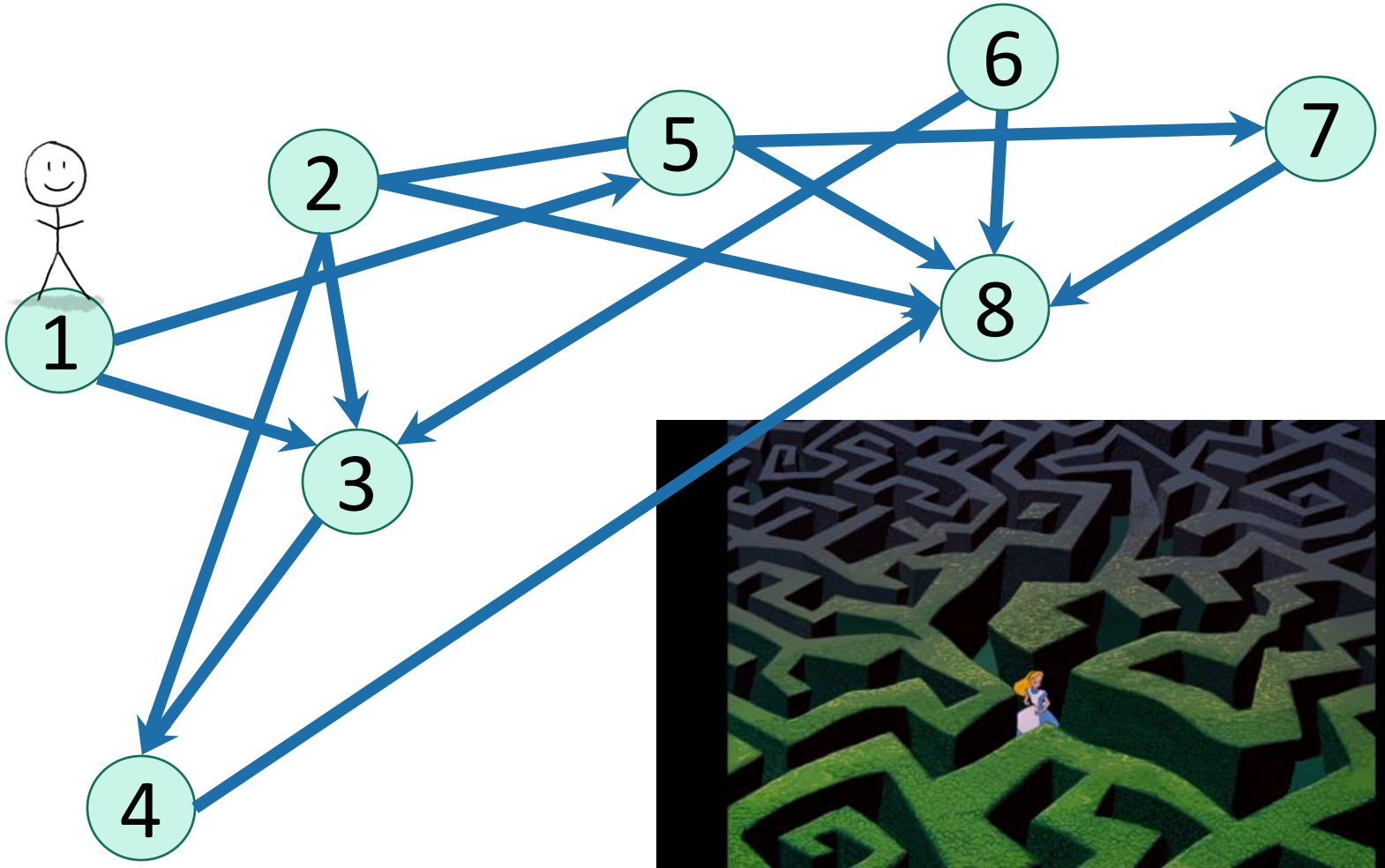
Finding

Strongly Connected Components

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

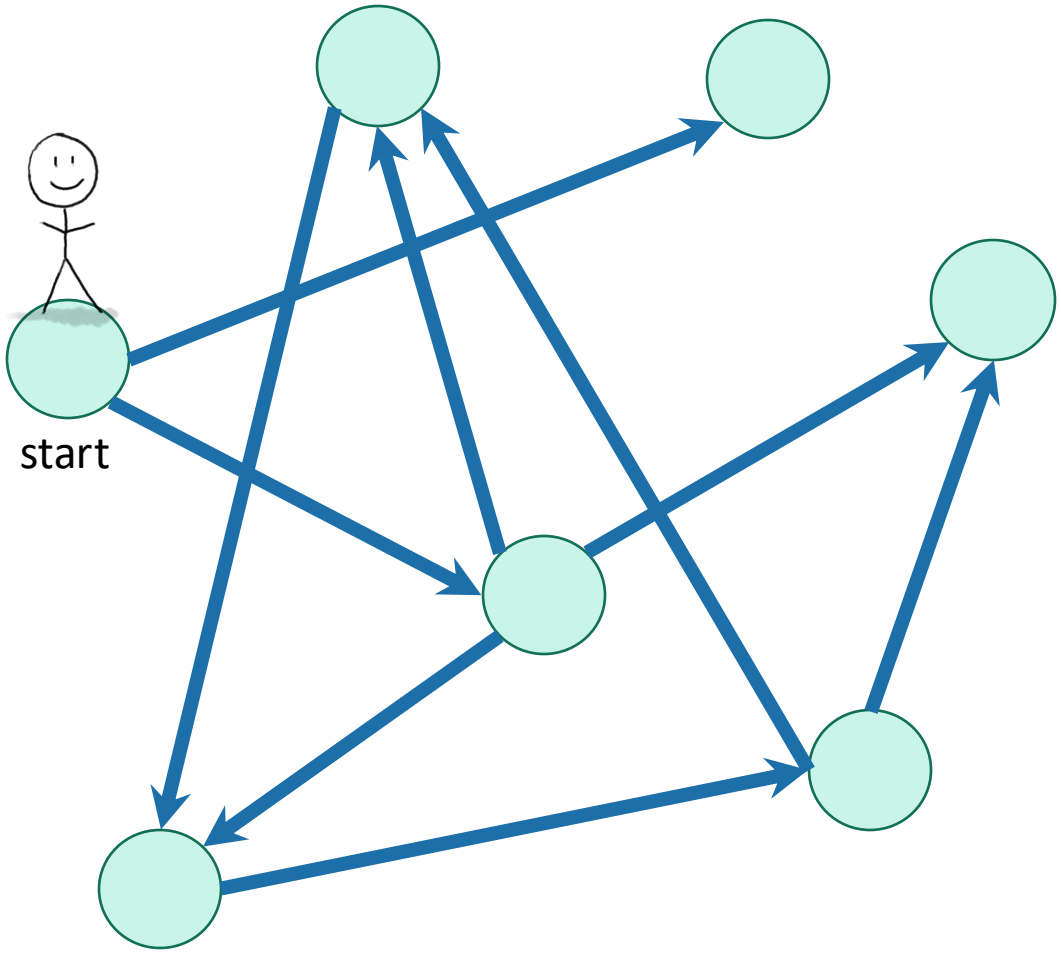
Recall: DFS

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

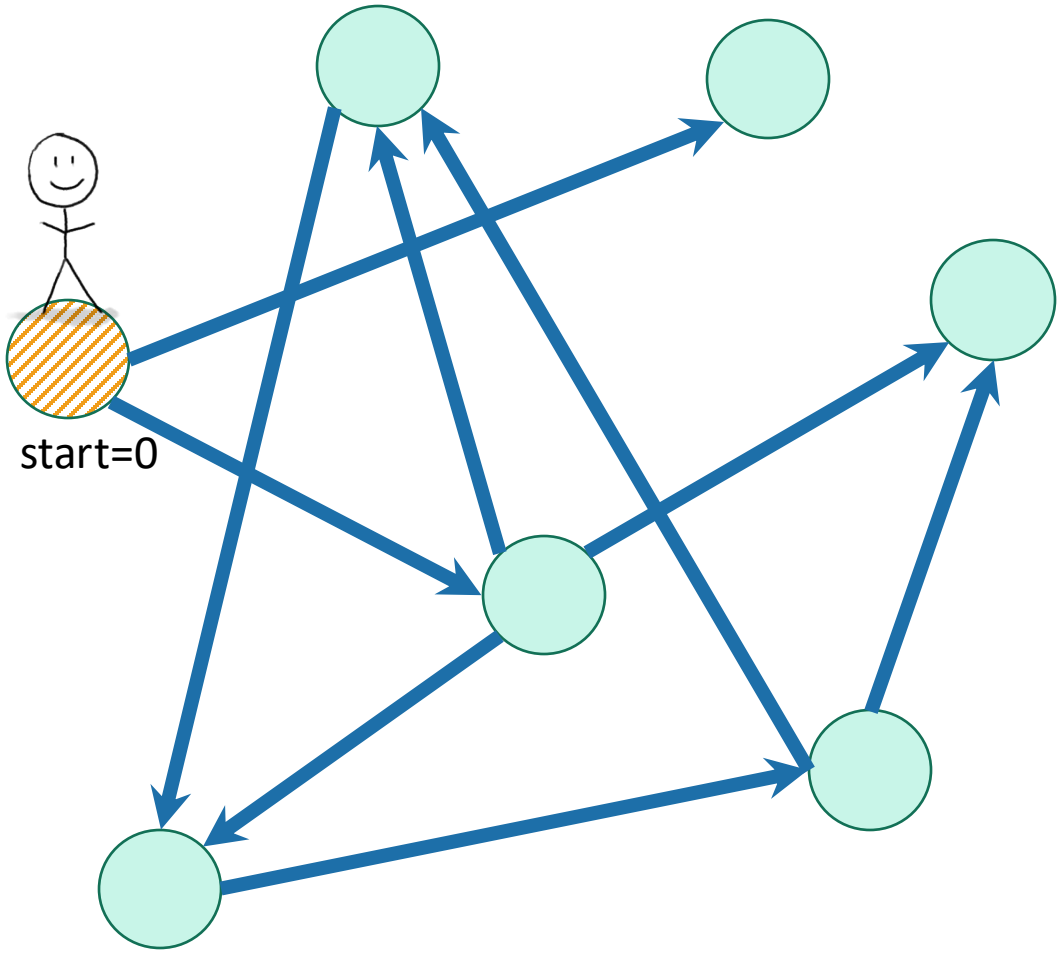
Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

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

Depth First Search

Exploring a labyrinth with chalk and a piece of string



Not been there yet

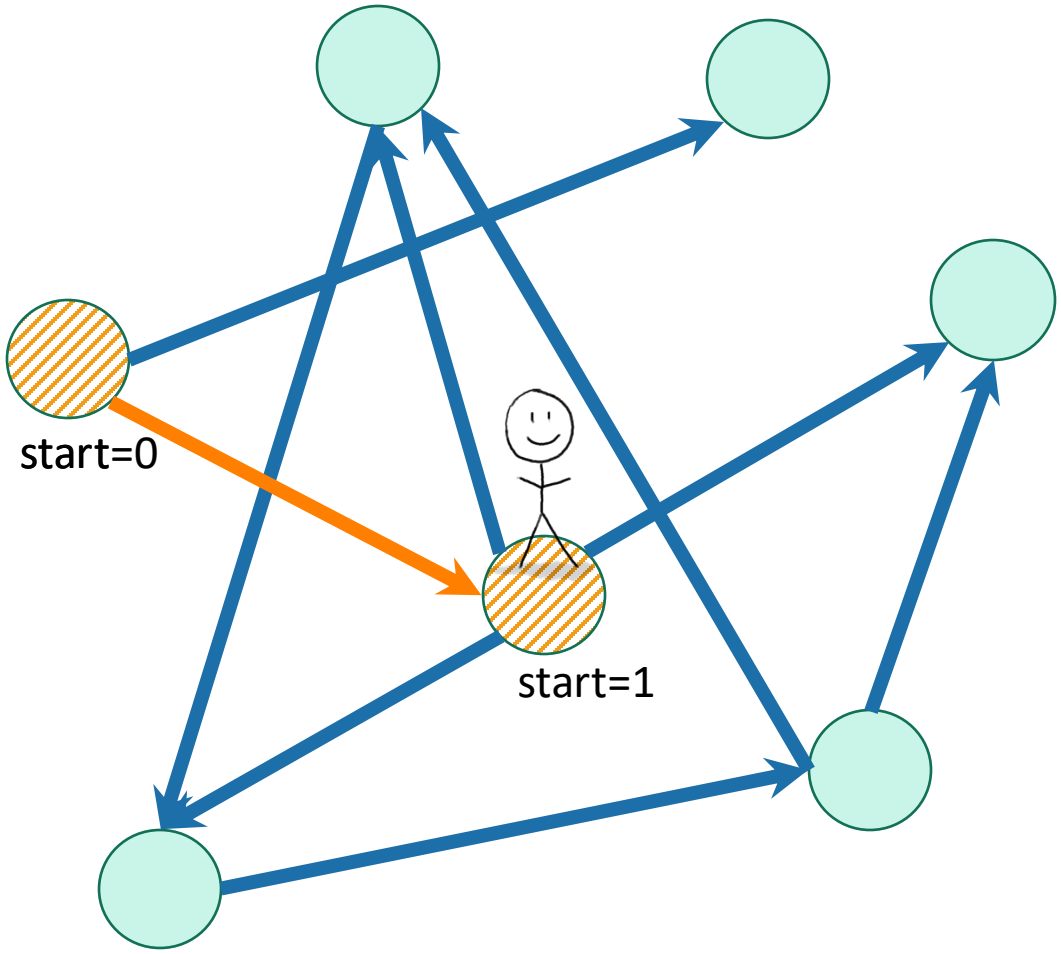
Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

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

Depth First Search

Exploring a labyrinth with chalk and a piece of string



Not been there yet

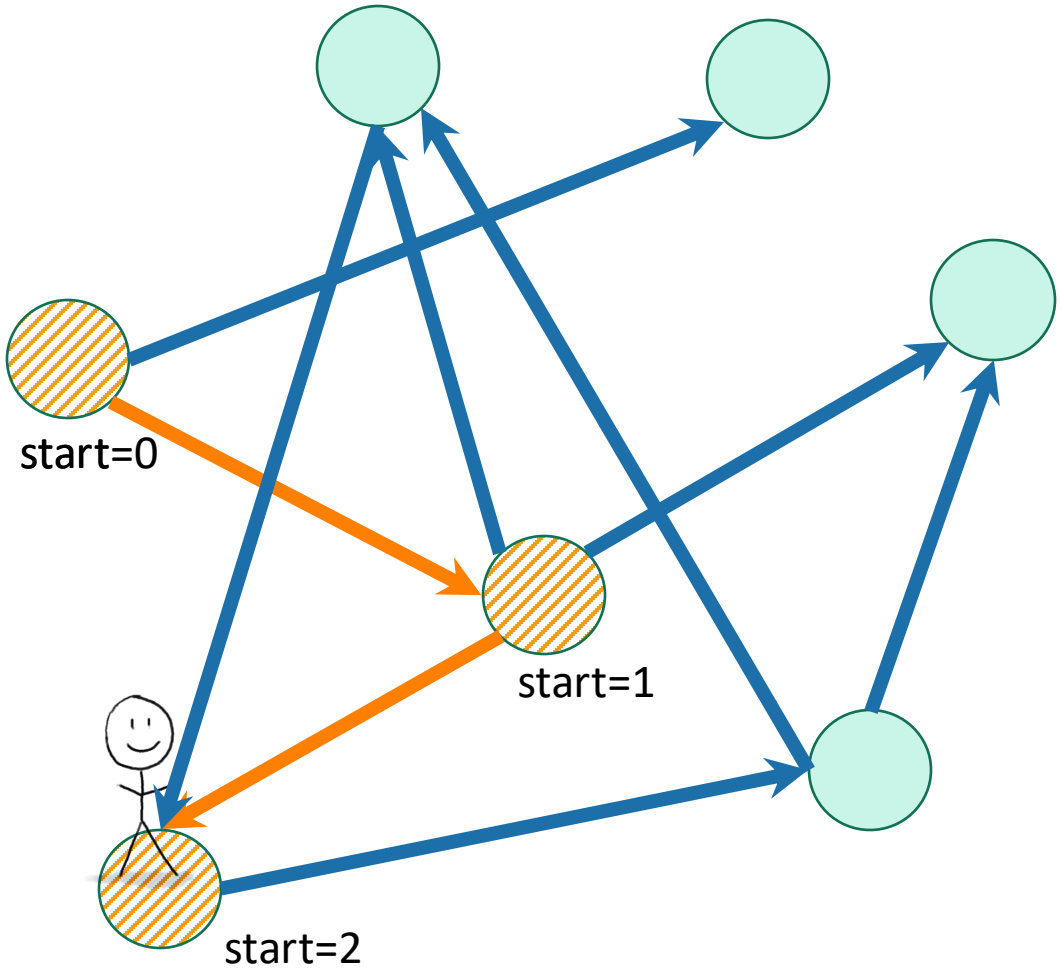
Been there, haven't explored all the paths out.

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

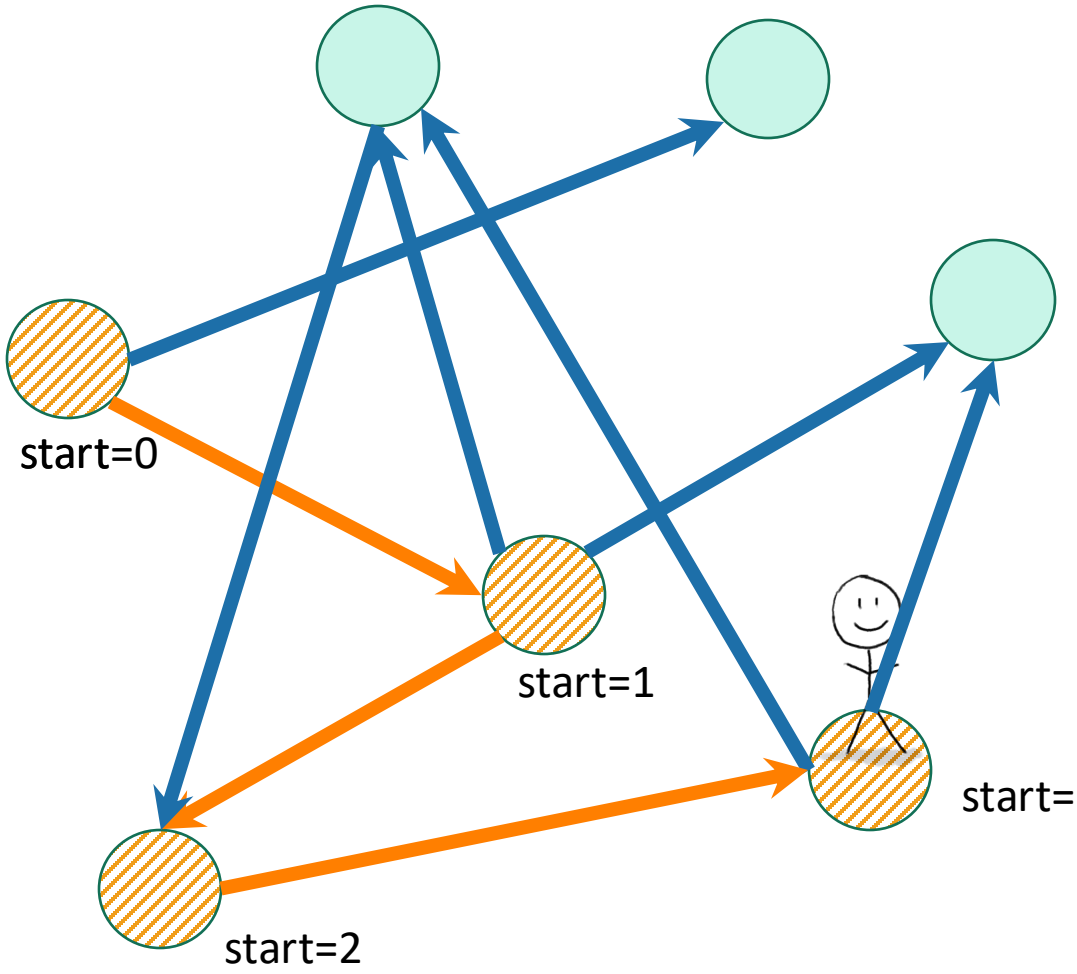
Been there, haven't explored all the paths out.

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

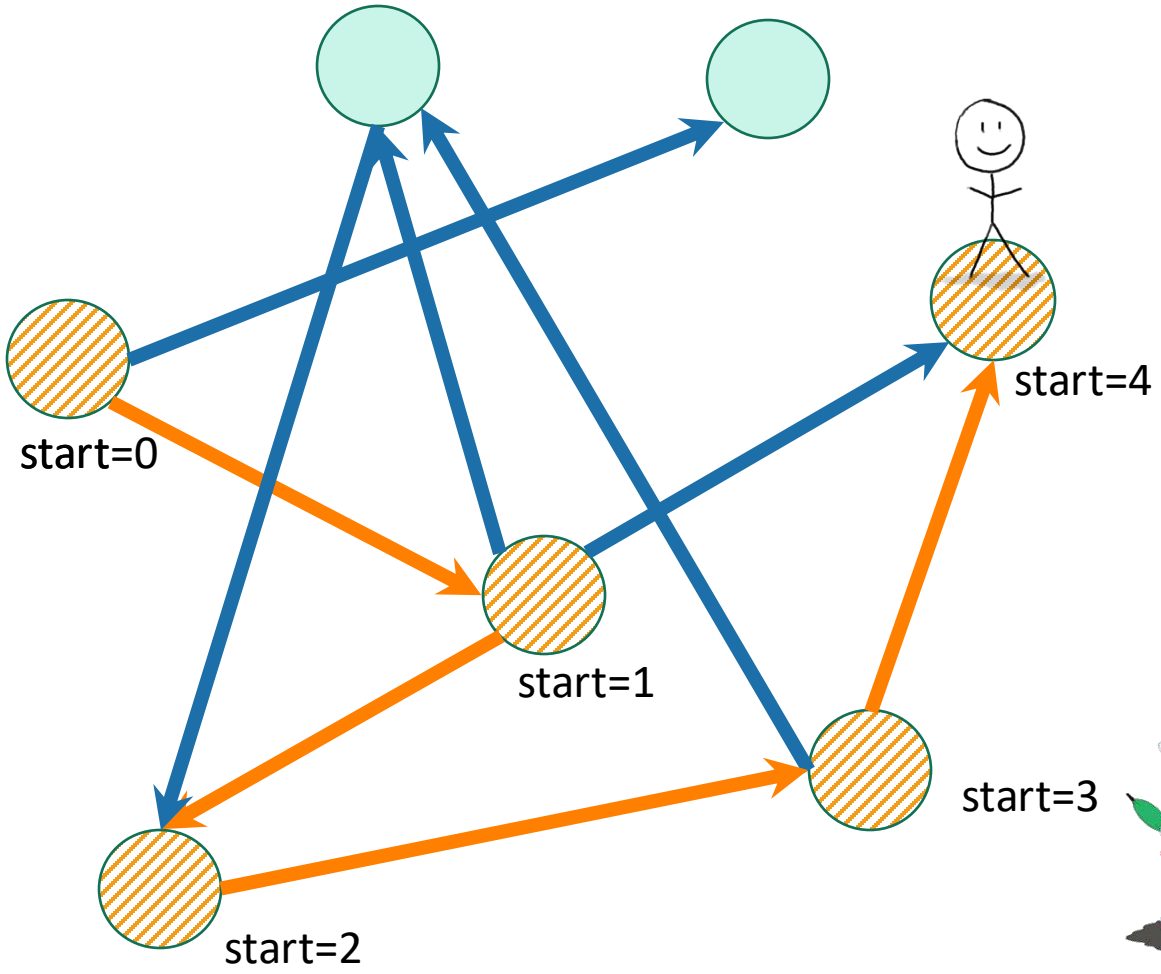
Been there, haven't explored all the paths out.

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

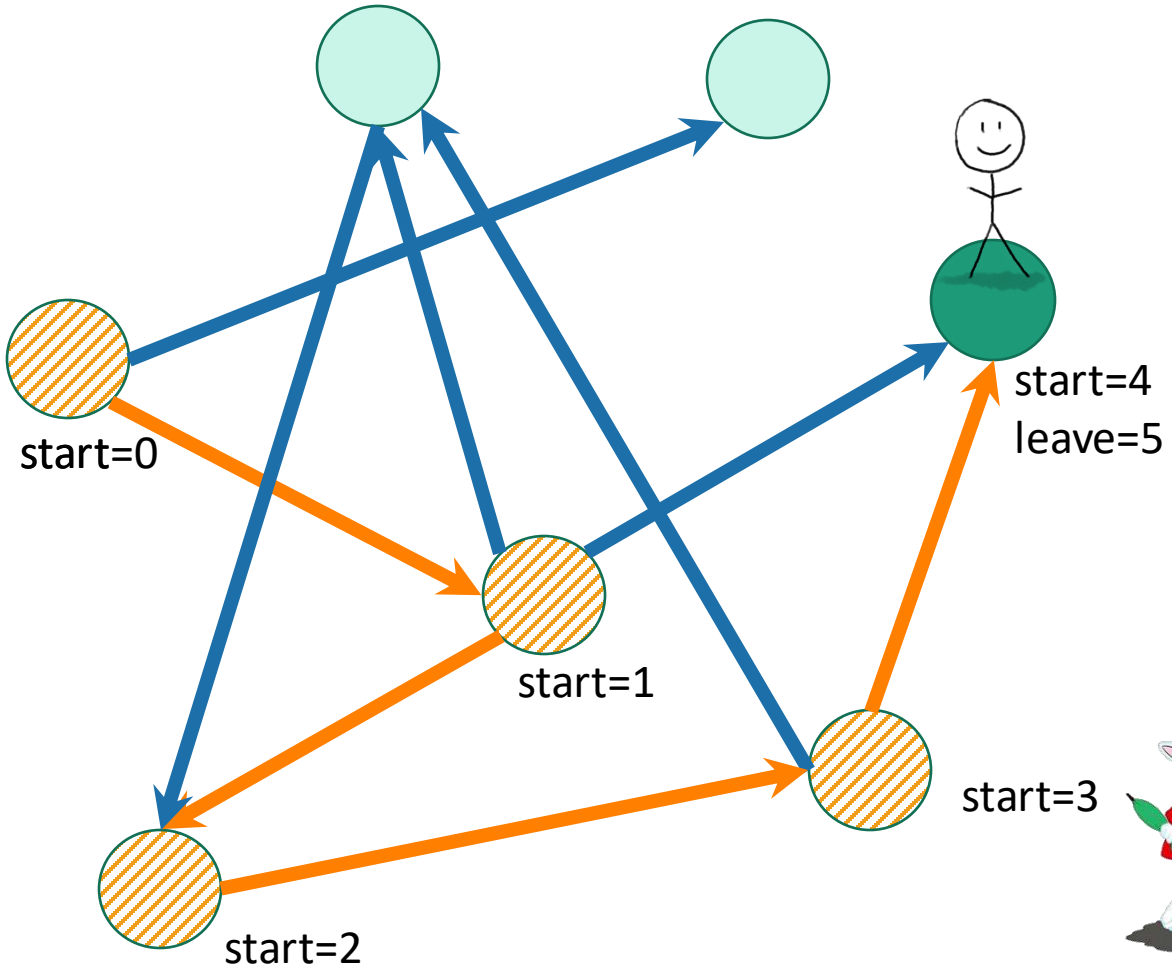
Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

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

Depth First Search

Exploring a labyrinth with chalk and a piece of string



Not been there yet

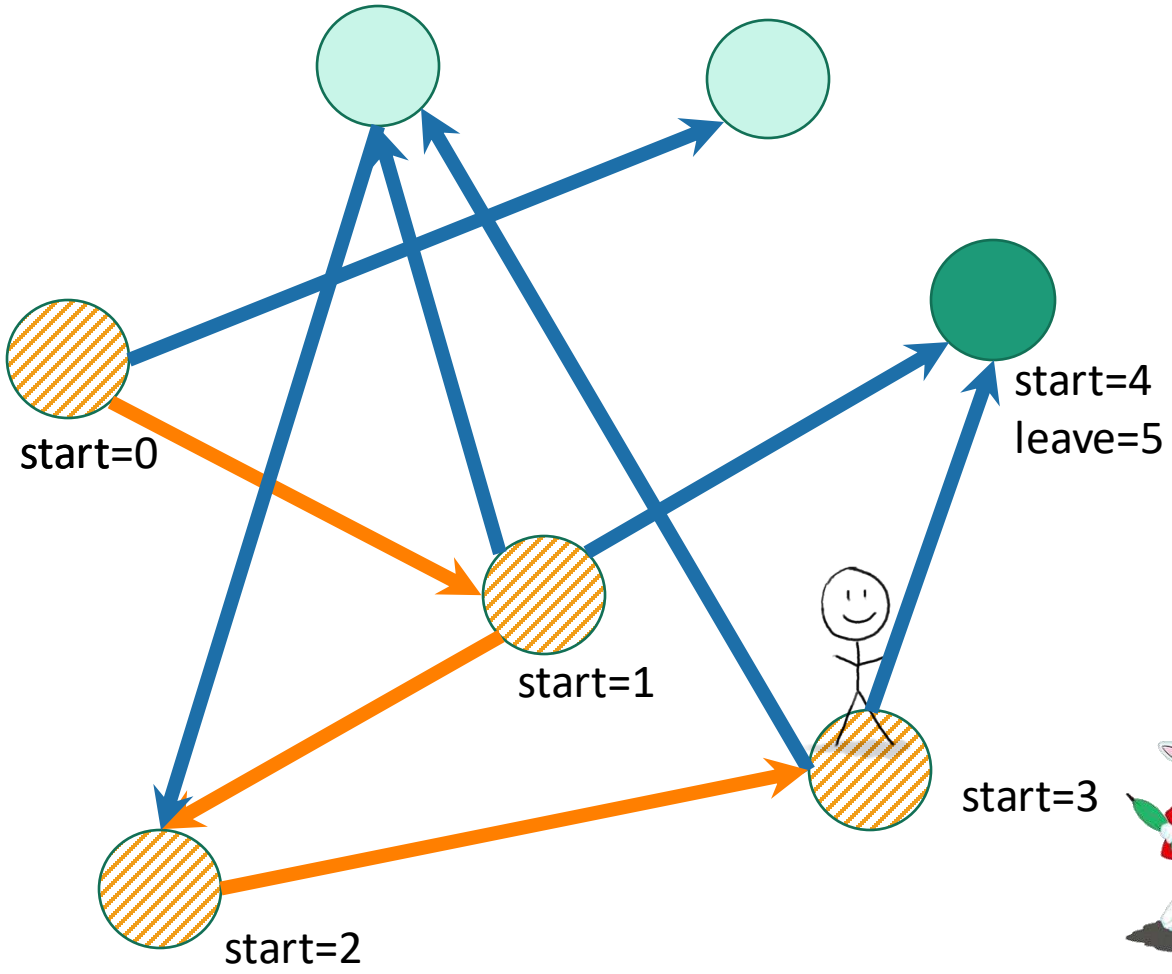
Been there, haven't explored all the paths out.

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

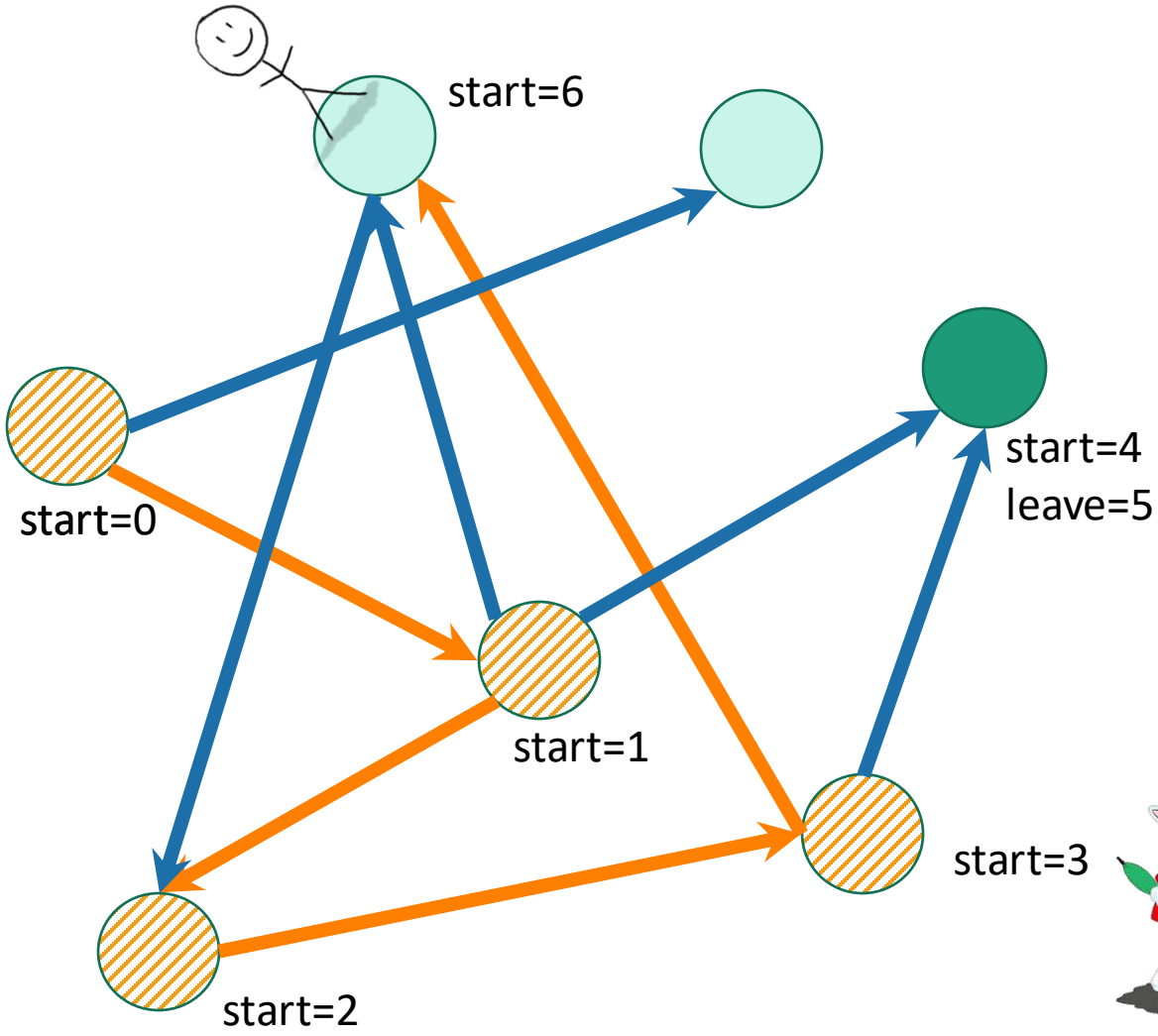
Been there, haven't explored all the paths out.

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

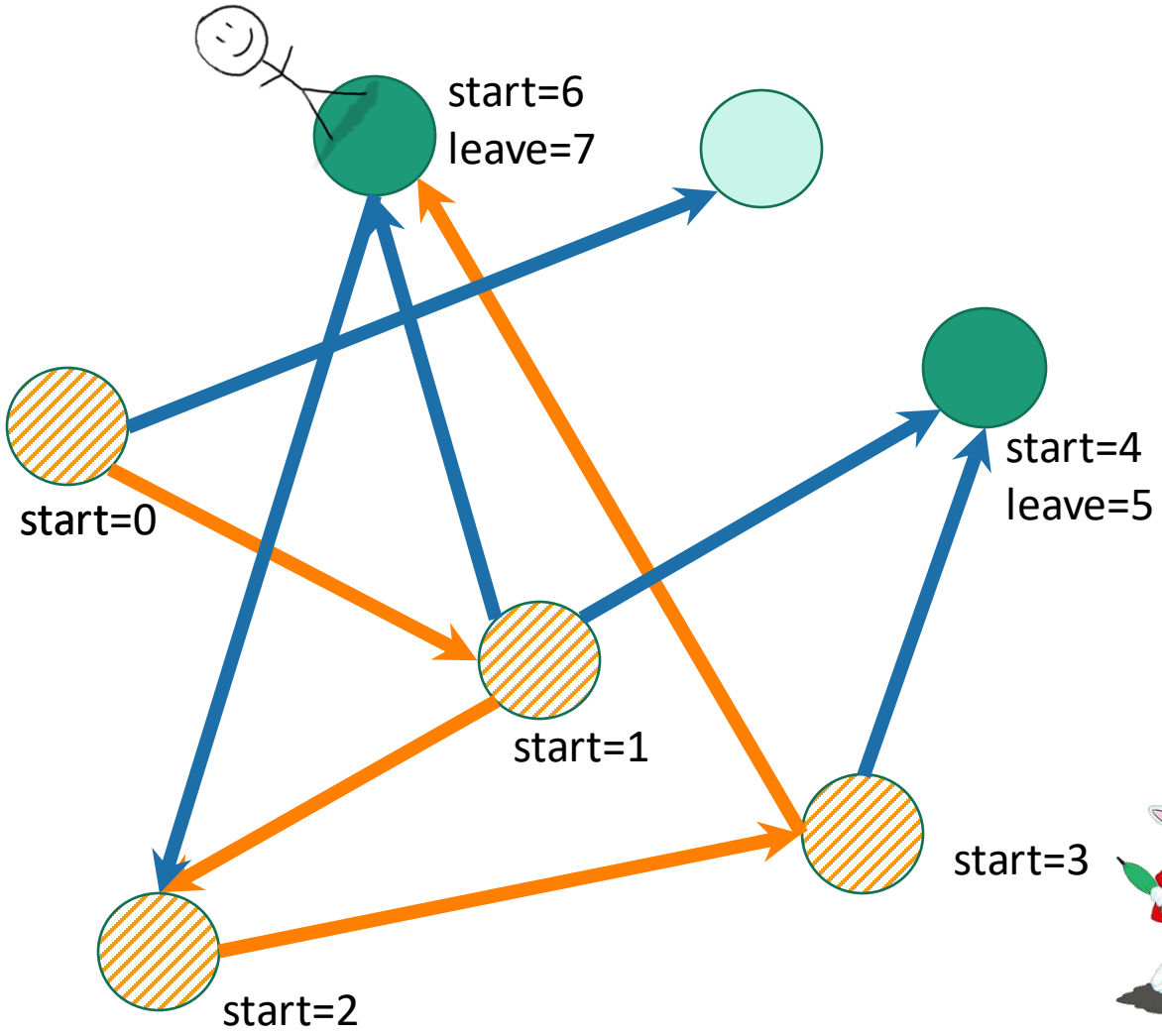
Been there, haven't explored all the paths out.

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

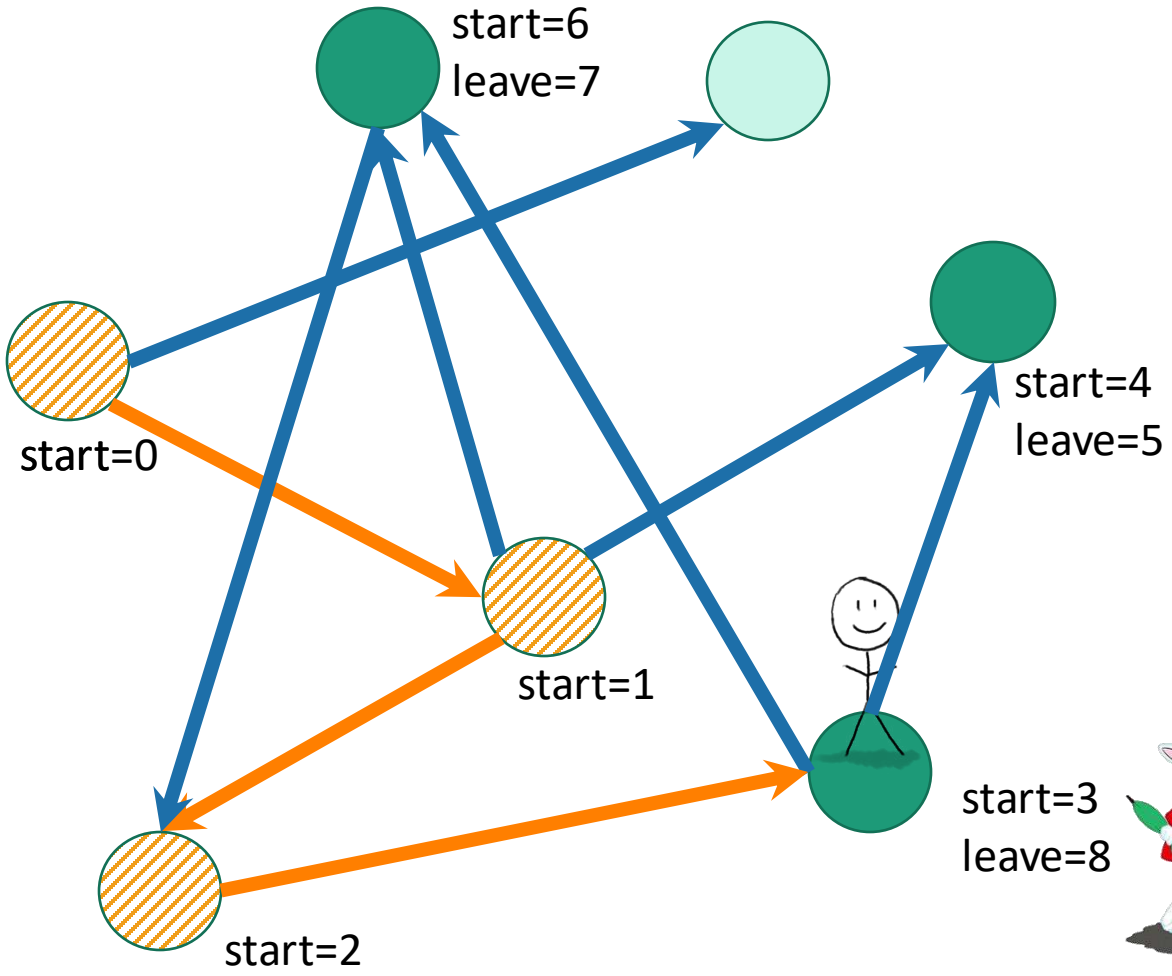
Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

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

Depth First Search

Exploring a labyrinth with chalk and a piece of string



Not been there yet

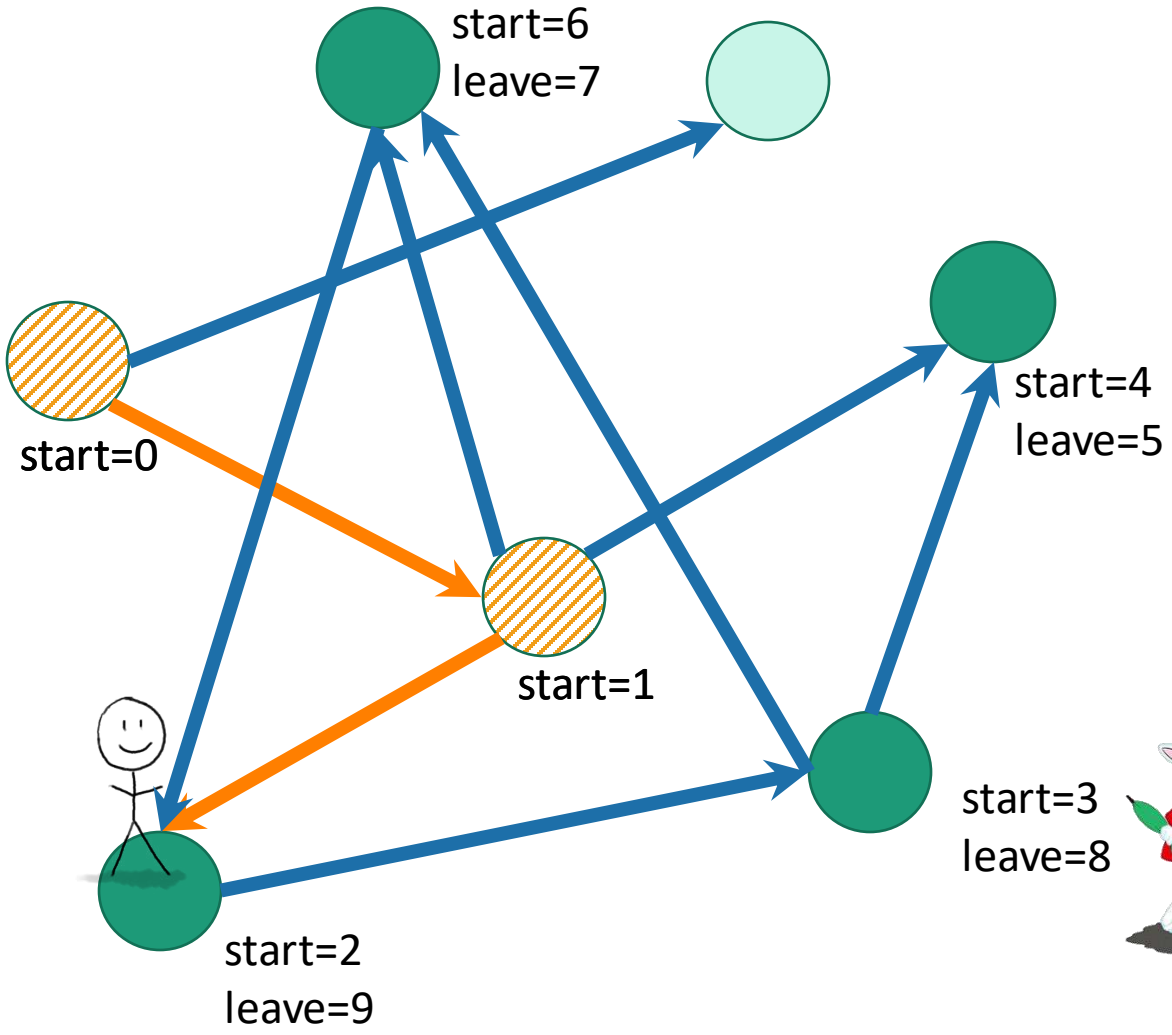
Been there, haven't explored all the paths out.

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

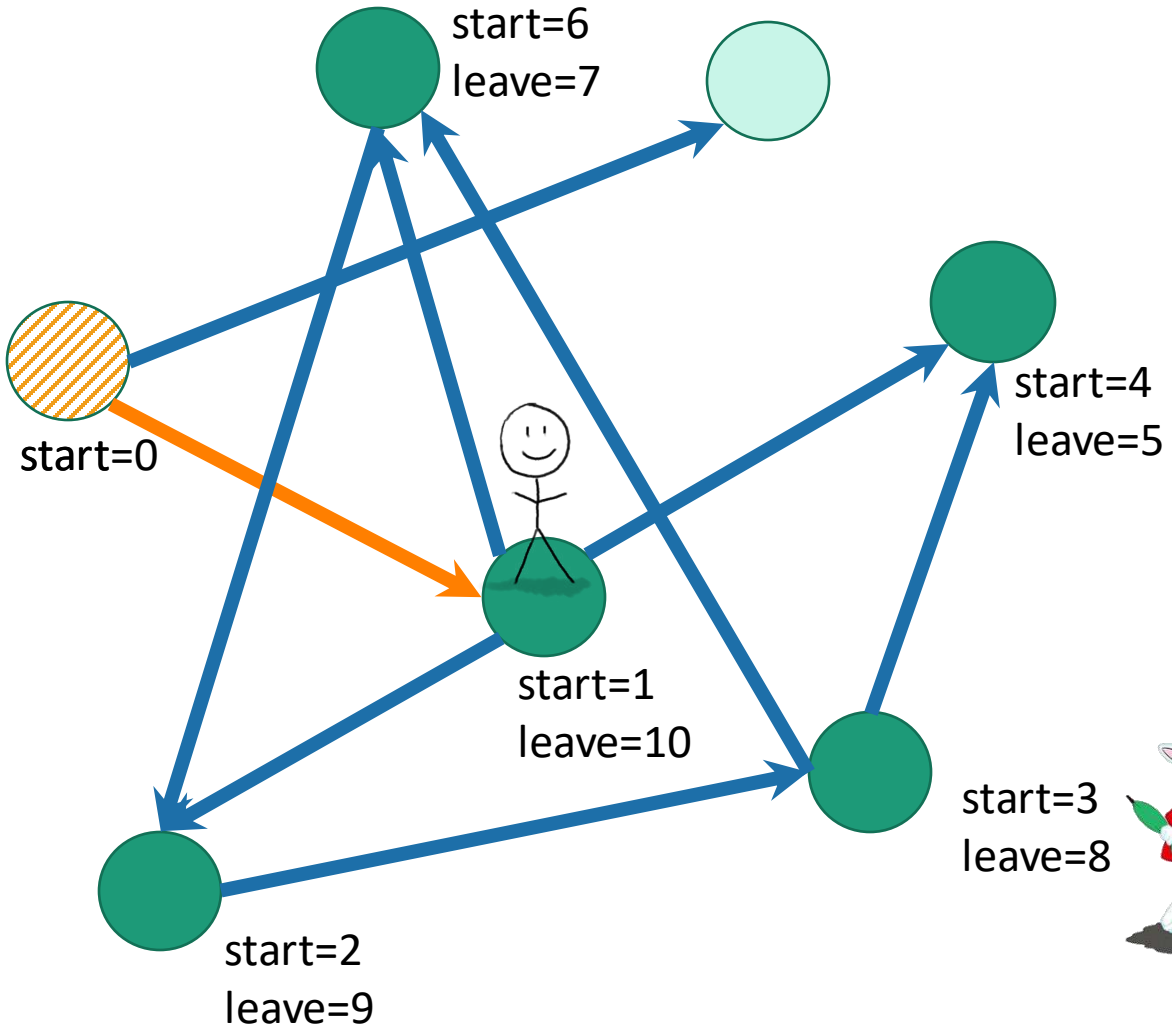
Been there, haven't explored all the paths out.

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

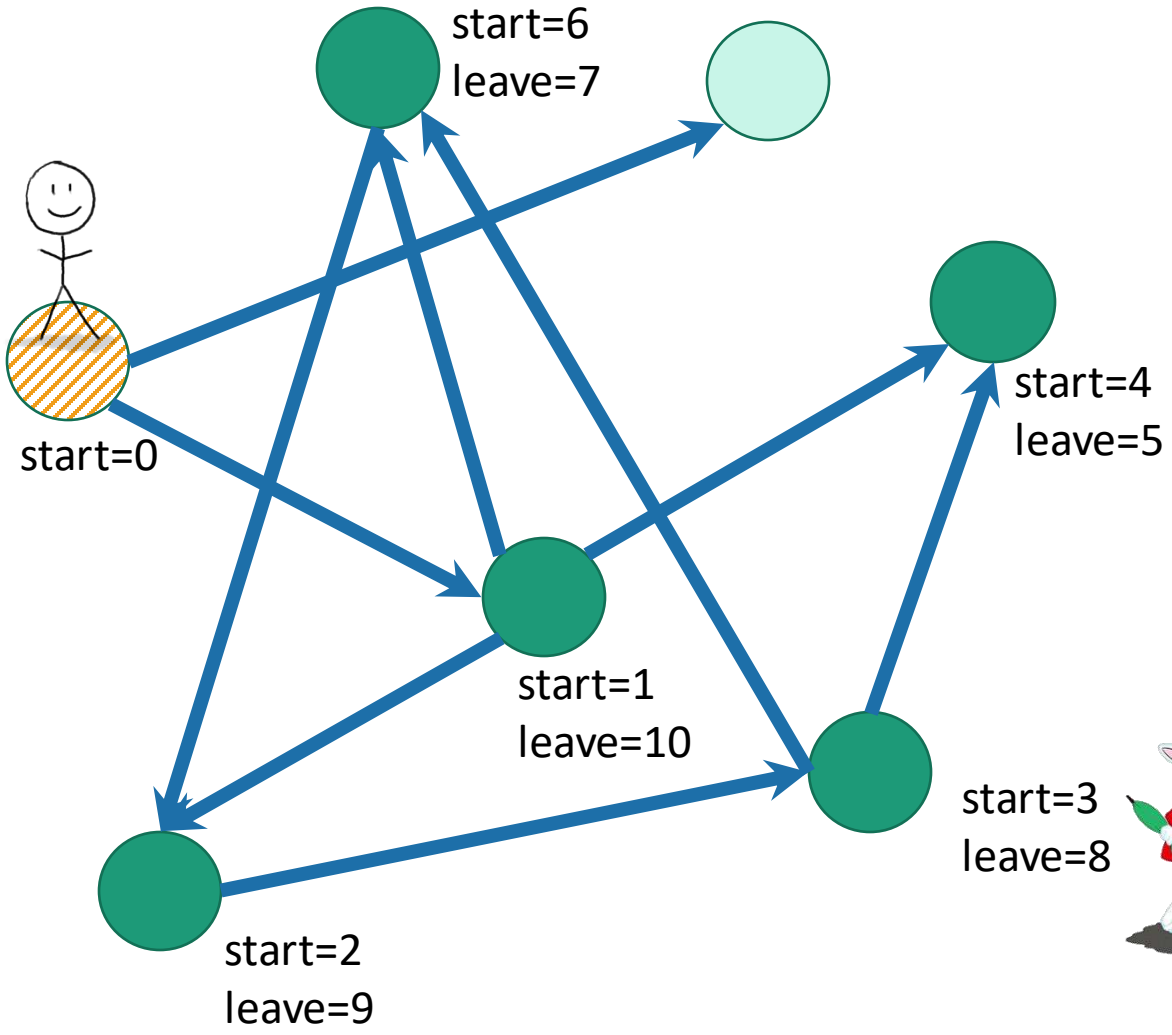
Been there, haven't explored all the paths out.

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

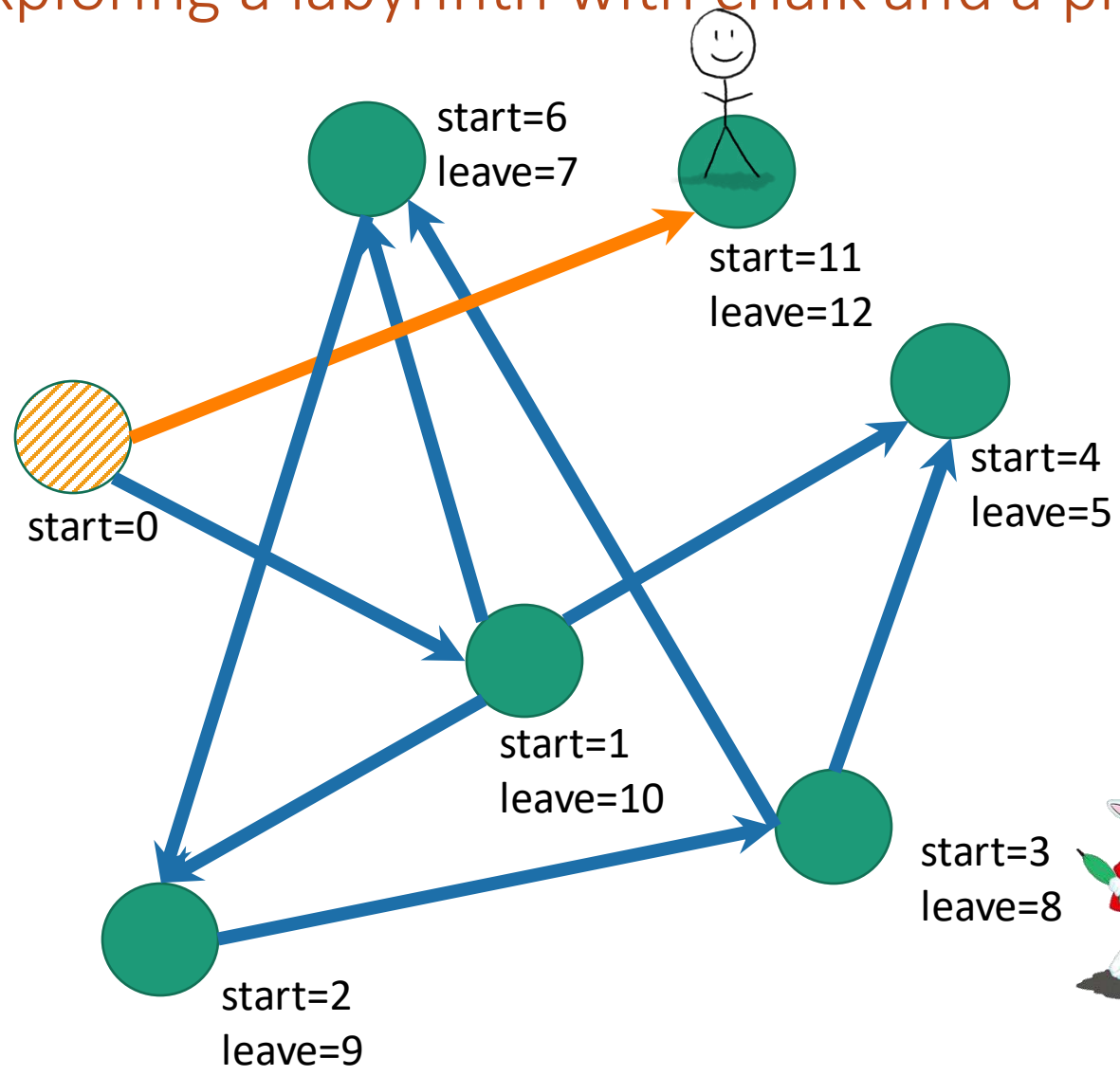
Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

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

Depth First Search

Exploring a labyrinth with chalk and a piece of string



Not been there yet

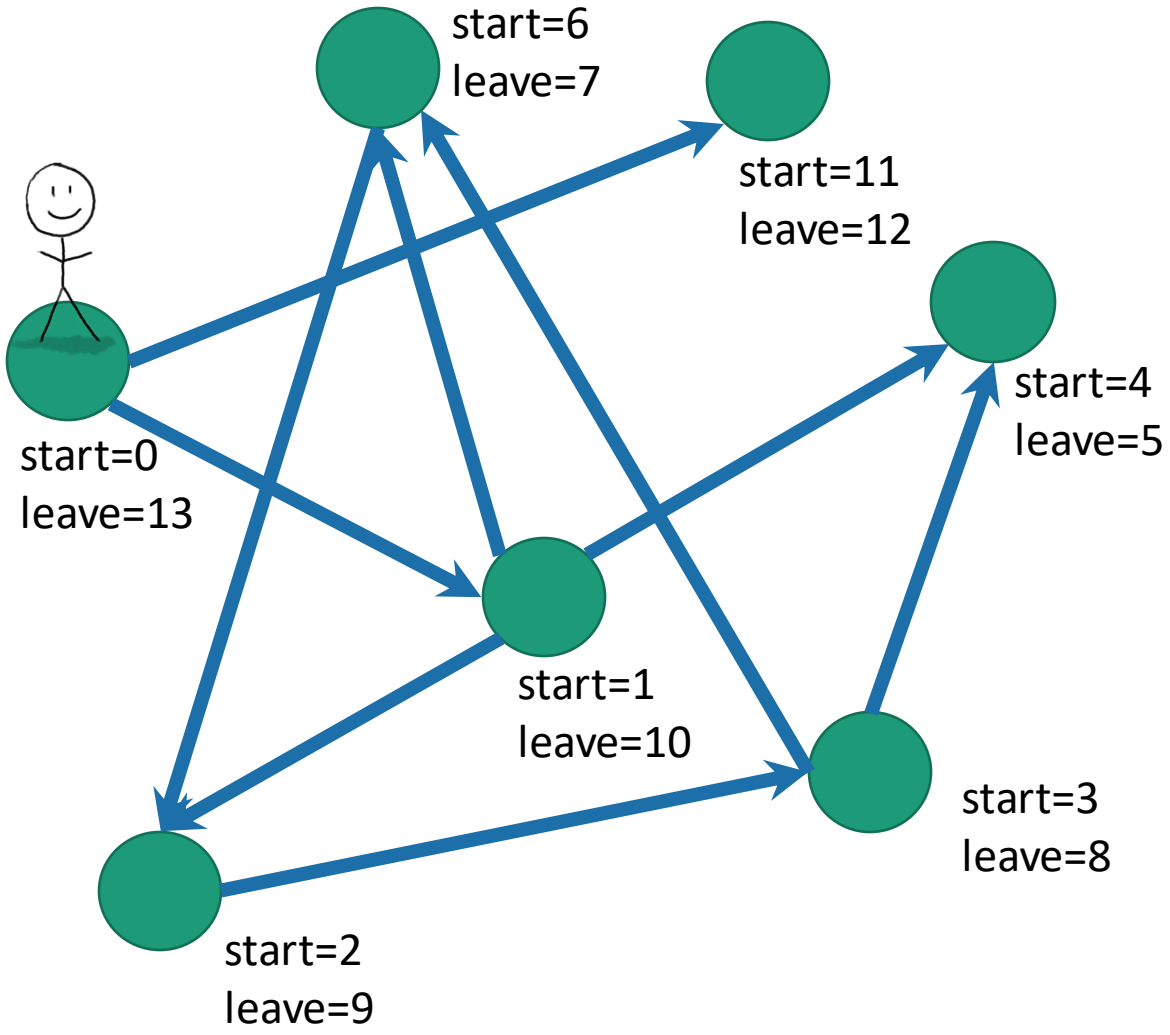
Been there, haven't explored all the paths out.

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

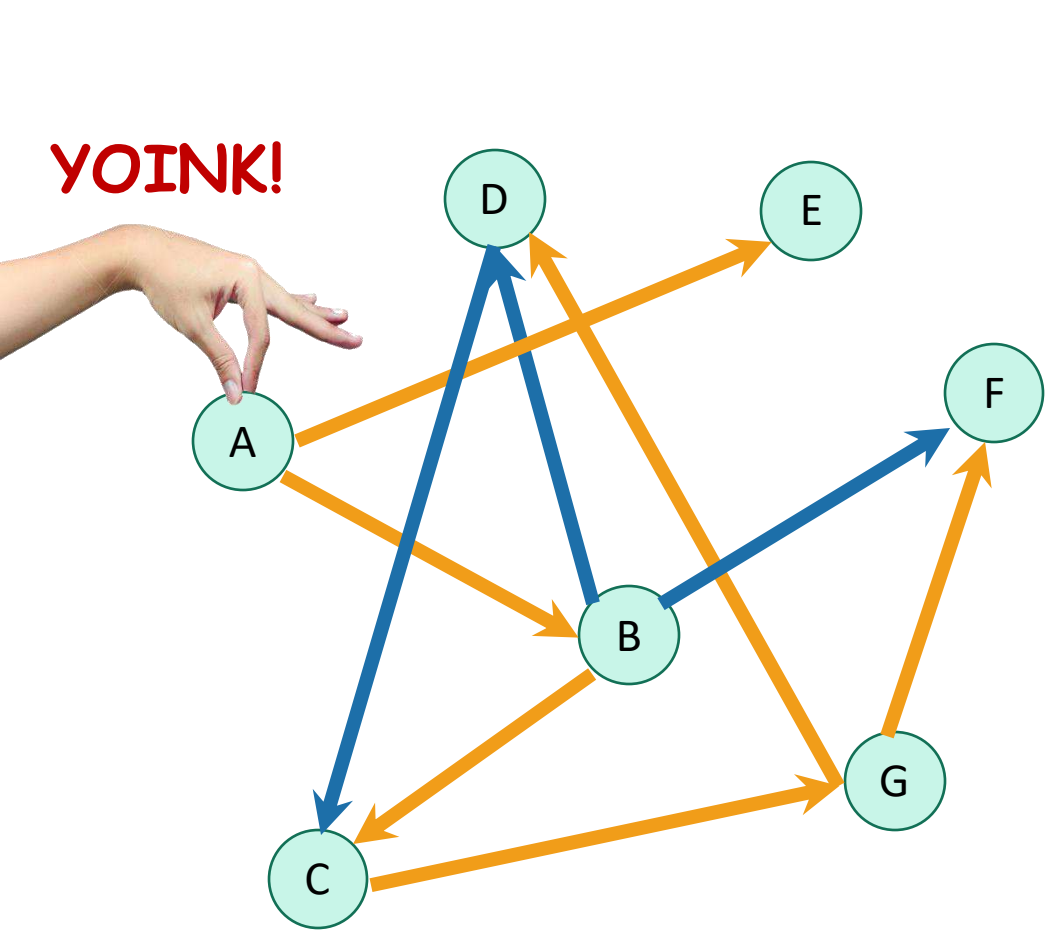
Been there, haven't explored all the paths out.

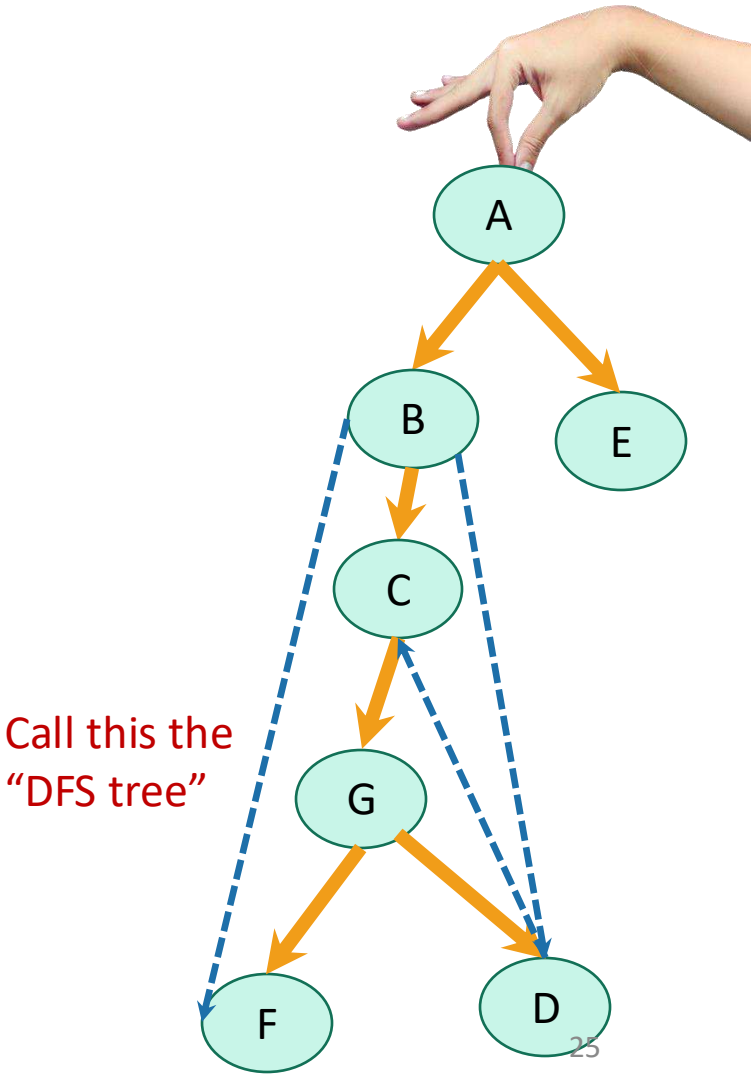
Been there, have explored all the paths out.

Labyrinth: explored!

Depth first search

implicitly creates a tree on everything you can reach

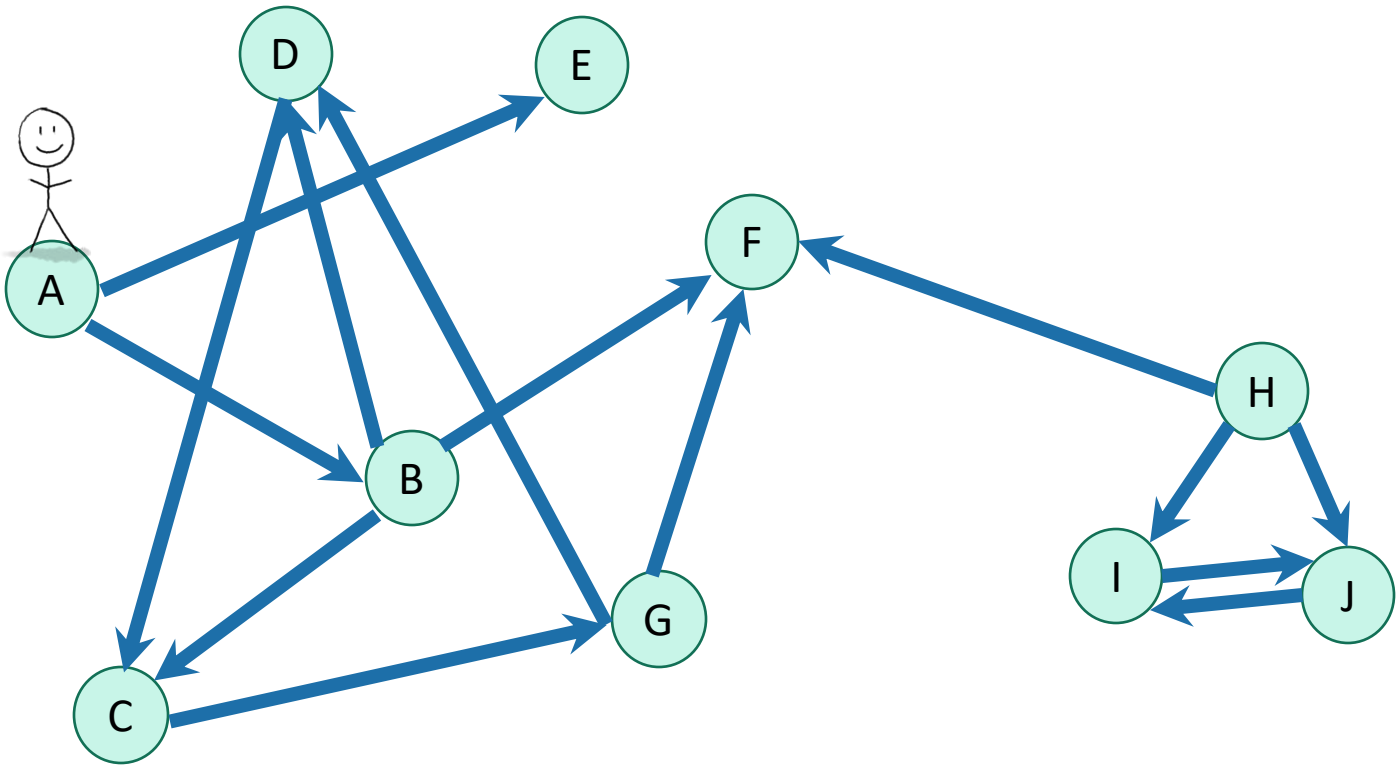




Call this the "DFS tree"

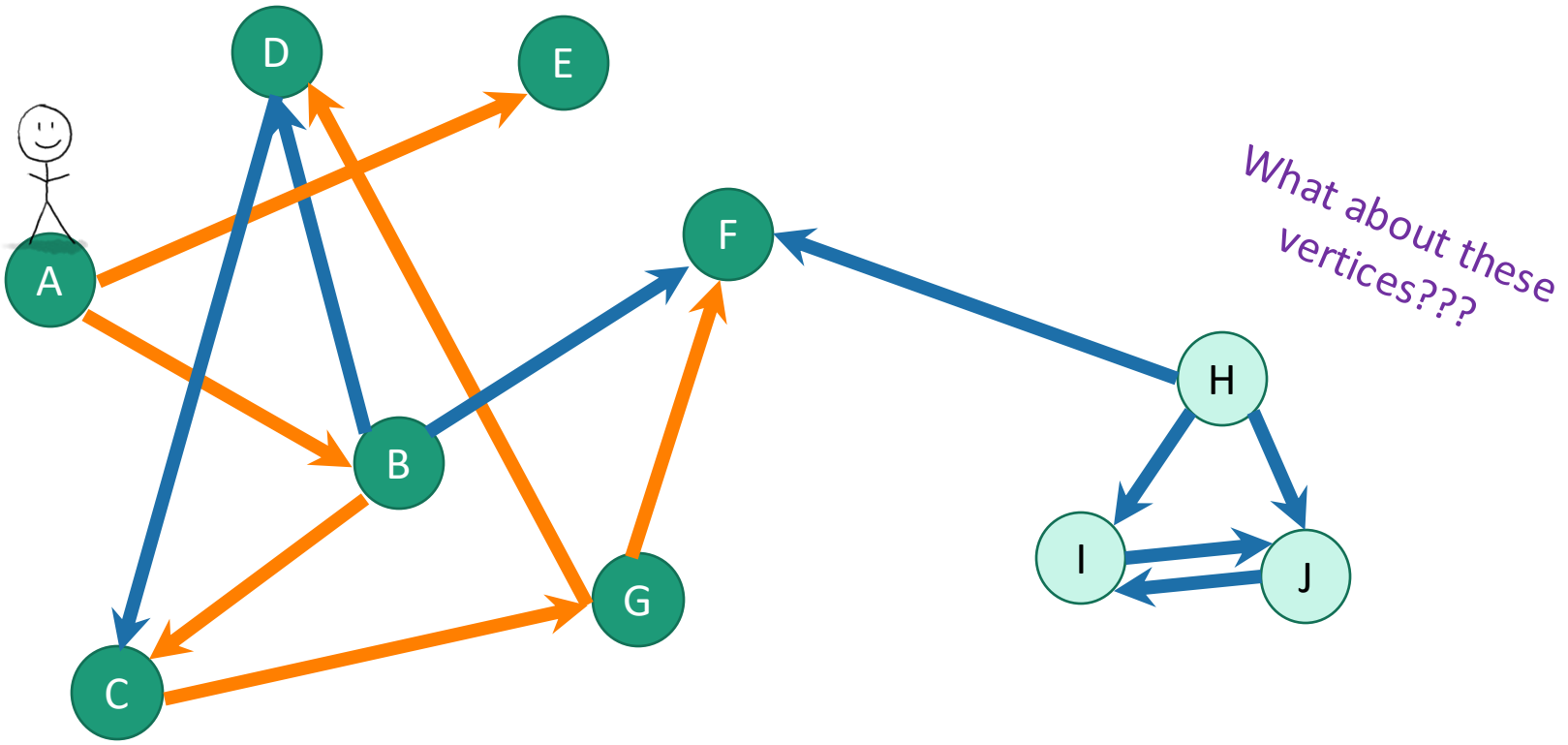
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



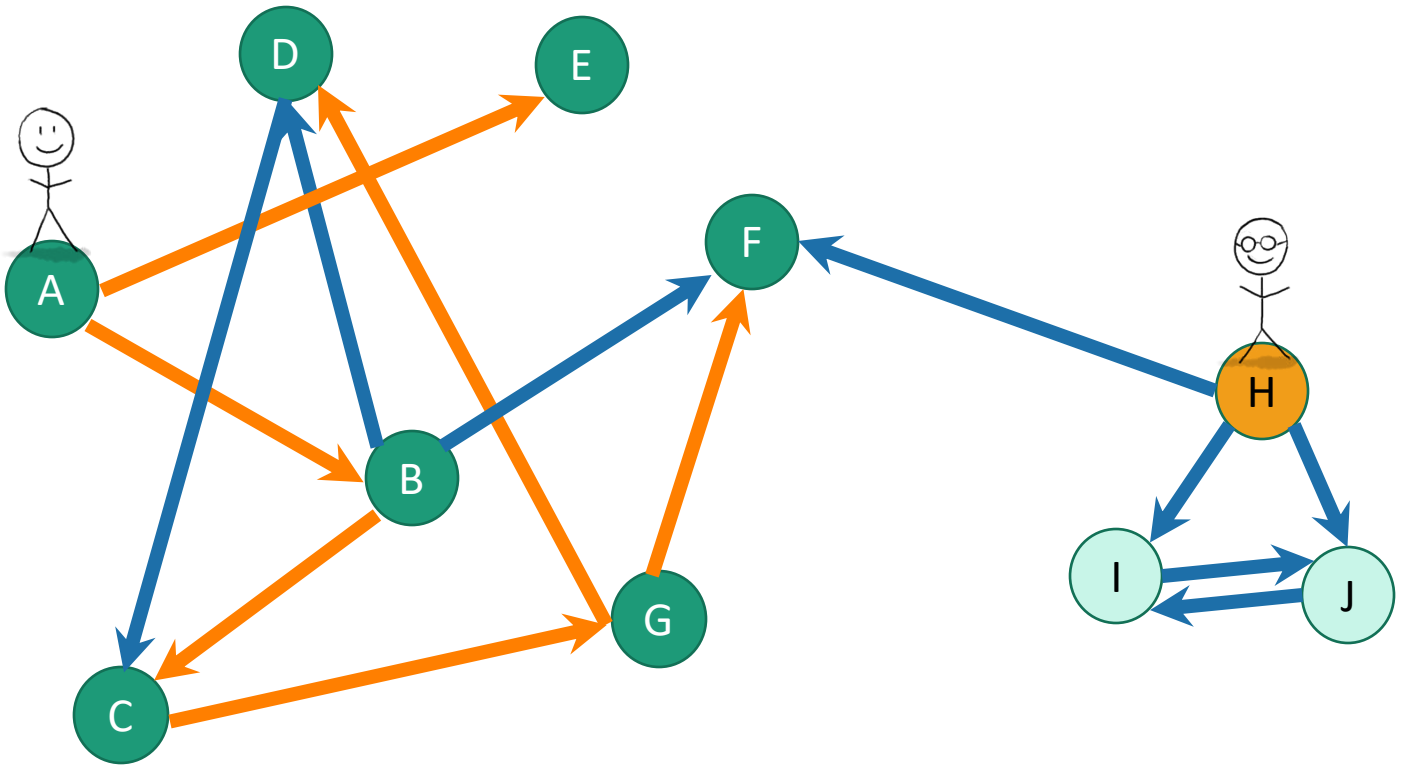
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



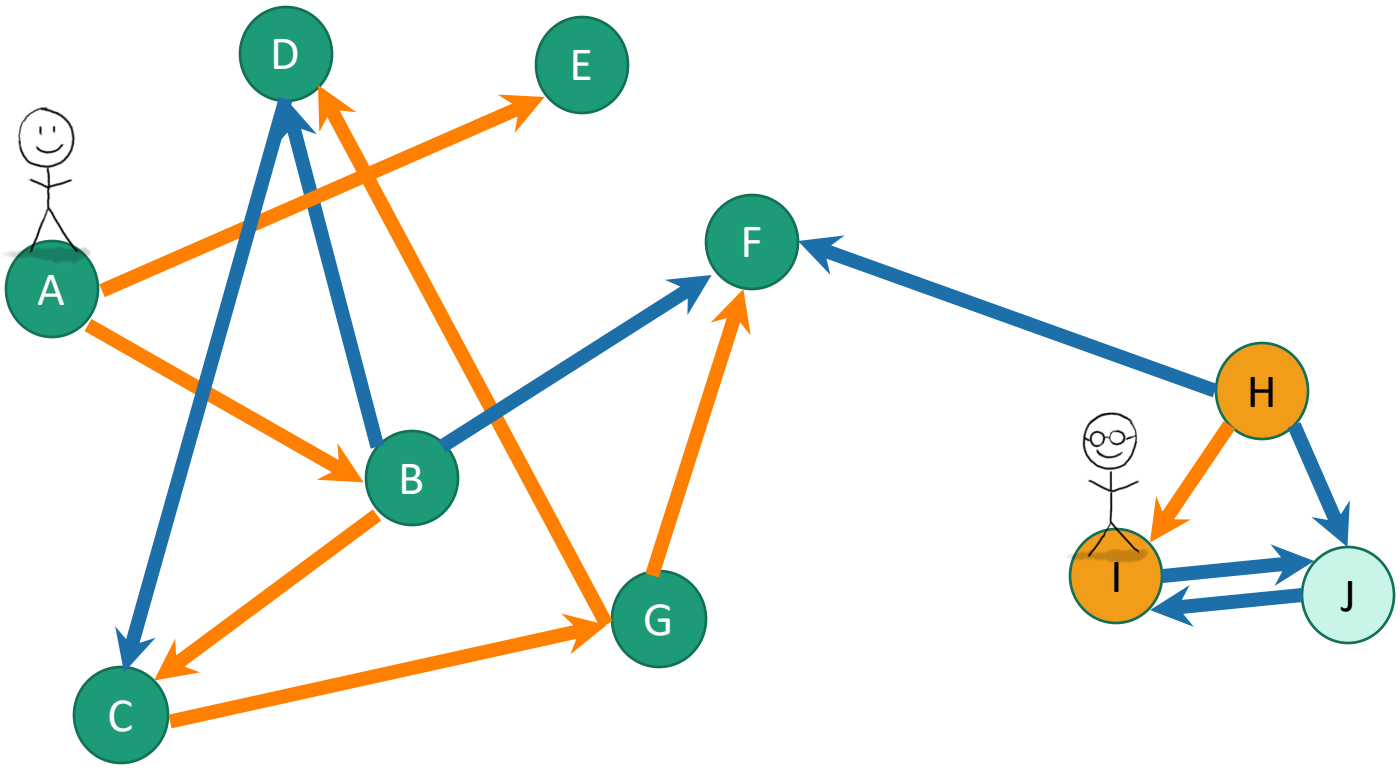
When you can’t reach everything

- Run DFS repeatedly to get a depth-first forest



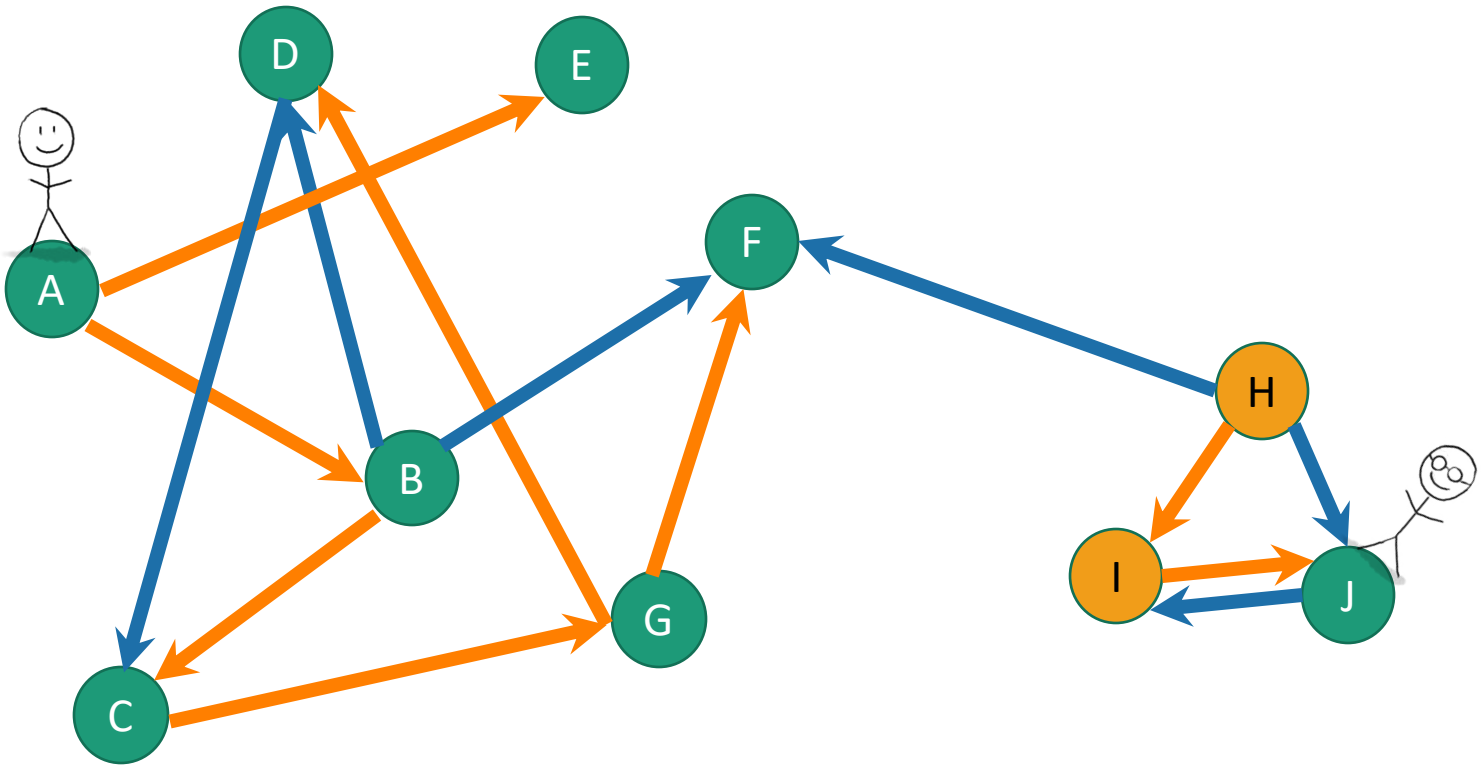
When you can’t reach everything

- Run DFS repeatedly to get a depth-first forest



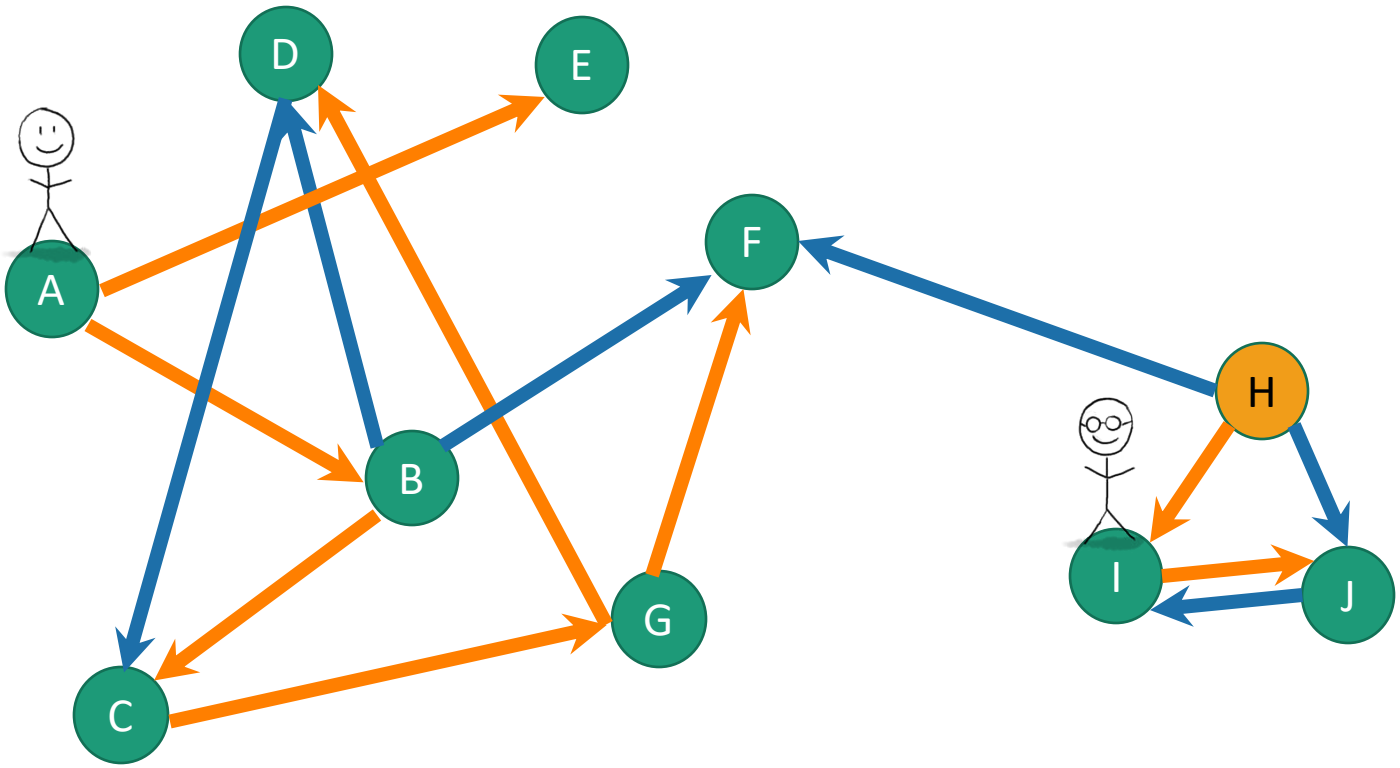
When you can’t reach everything

- Run DFS repeatedly to get a depth-first forest



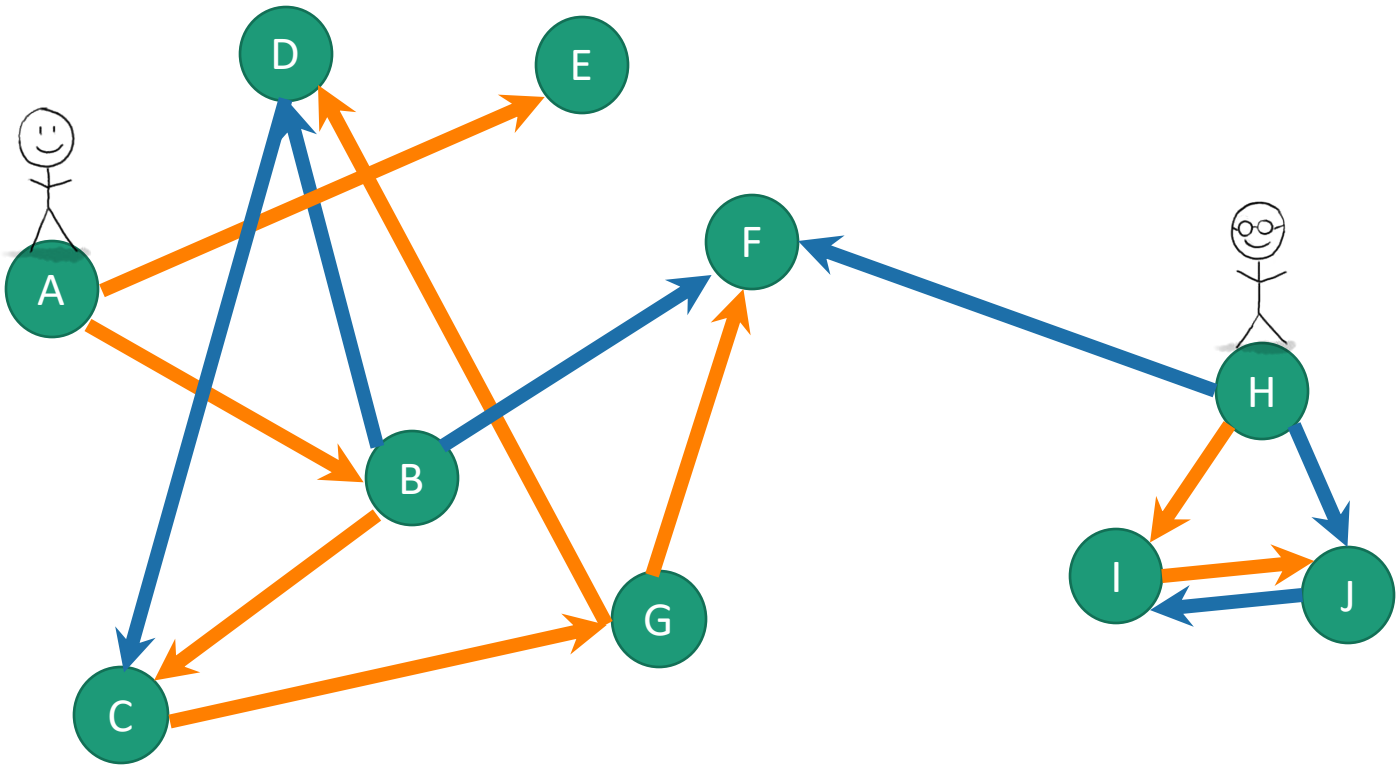
When you can’t reach everything

- Run DFS repeatedly to get a depth-first forest



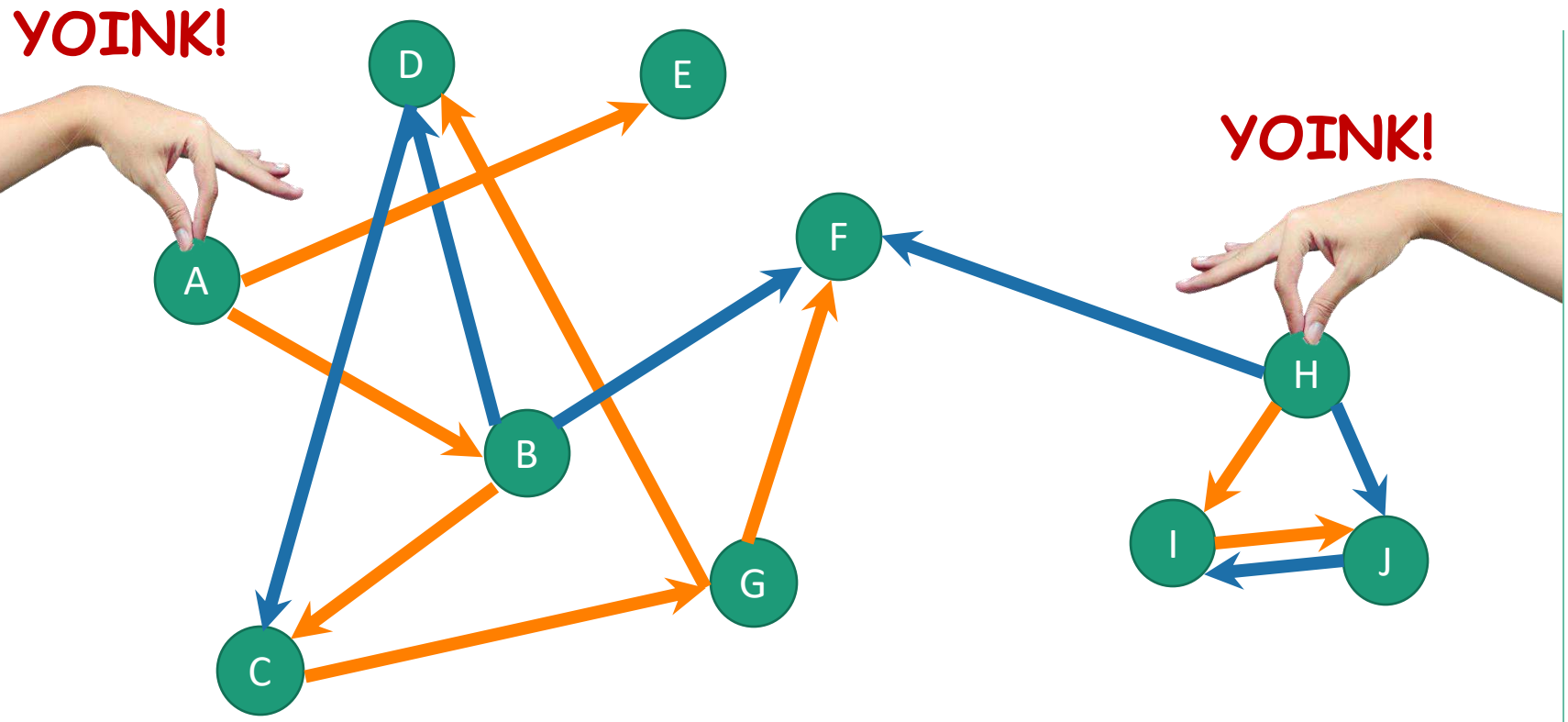
When you can’t reach everything

- Run DFS repeatedly to get a depth-first forest



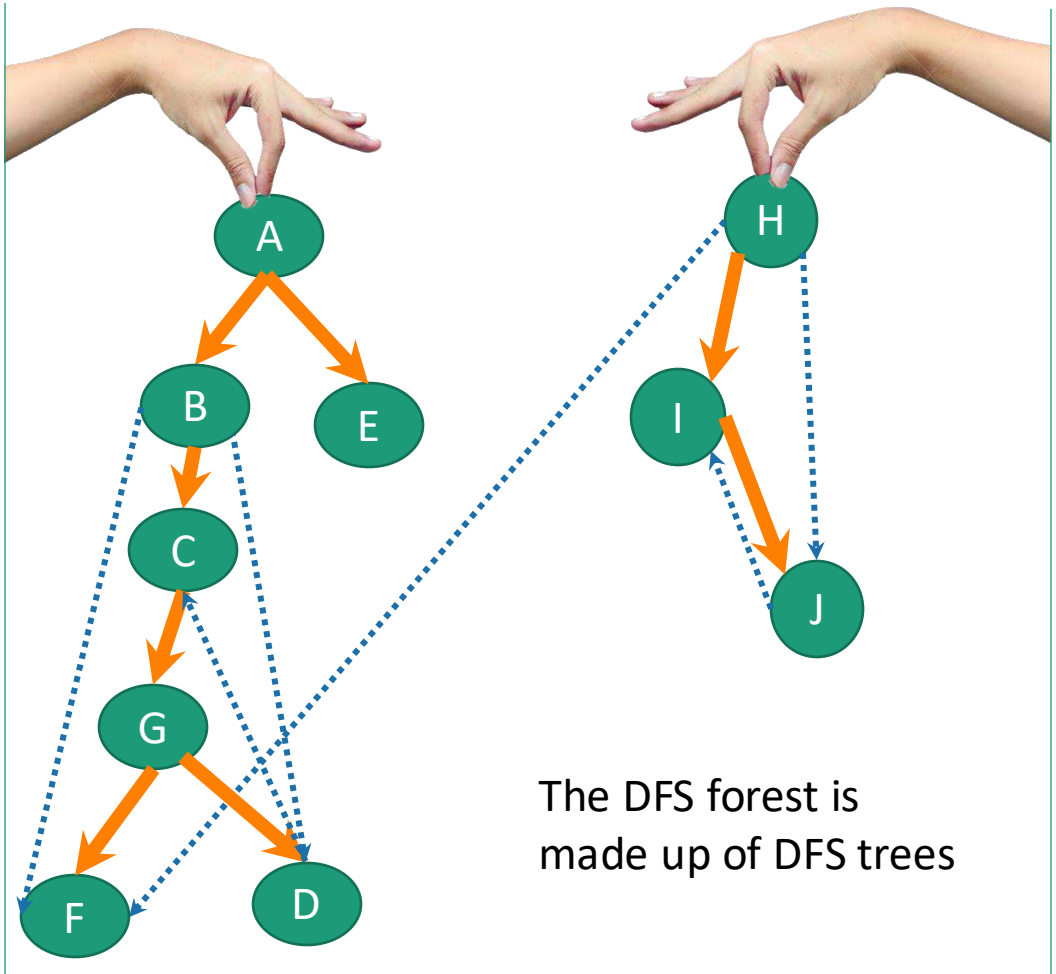
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



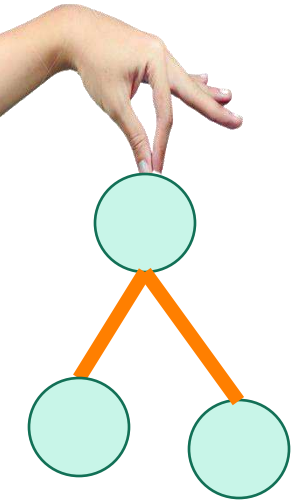
When you can't reach everything

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



Recall:

(Works the same with DFS forests)



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

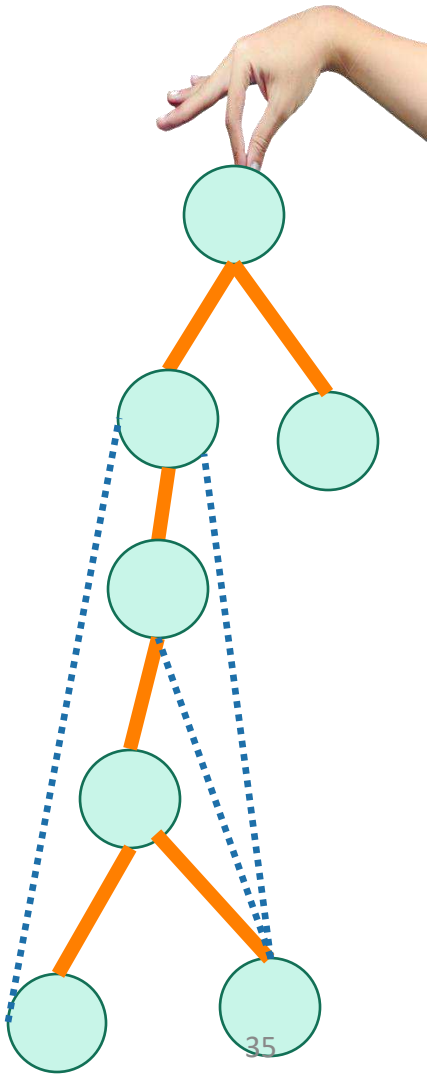
- If w is a descendent of v in this tree:

- 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 (or the other way around).

DFS tree

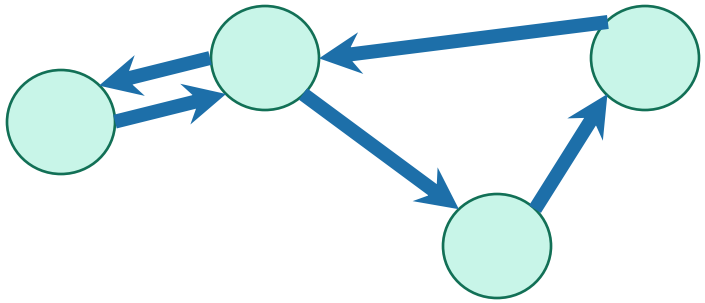


Enough of review

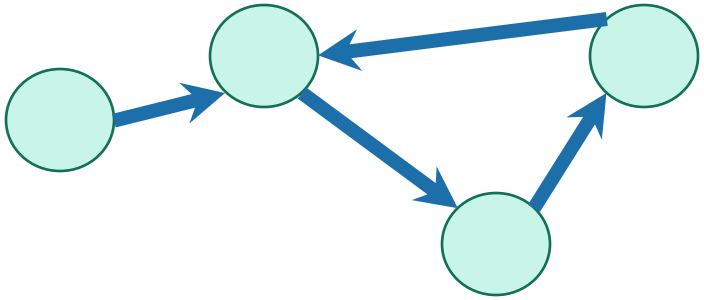
Strongly connected components

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.



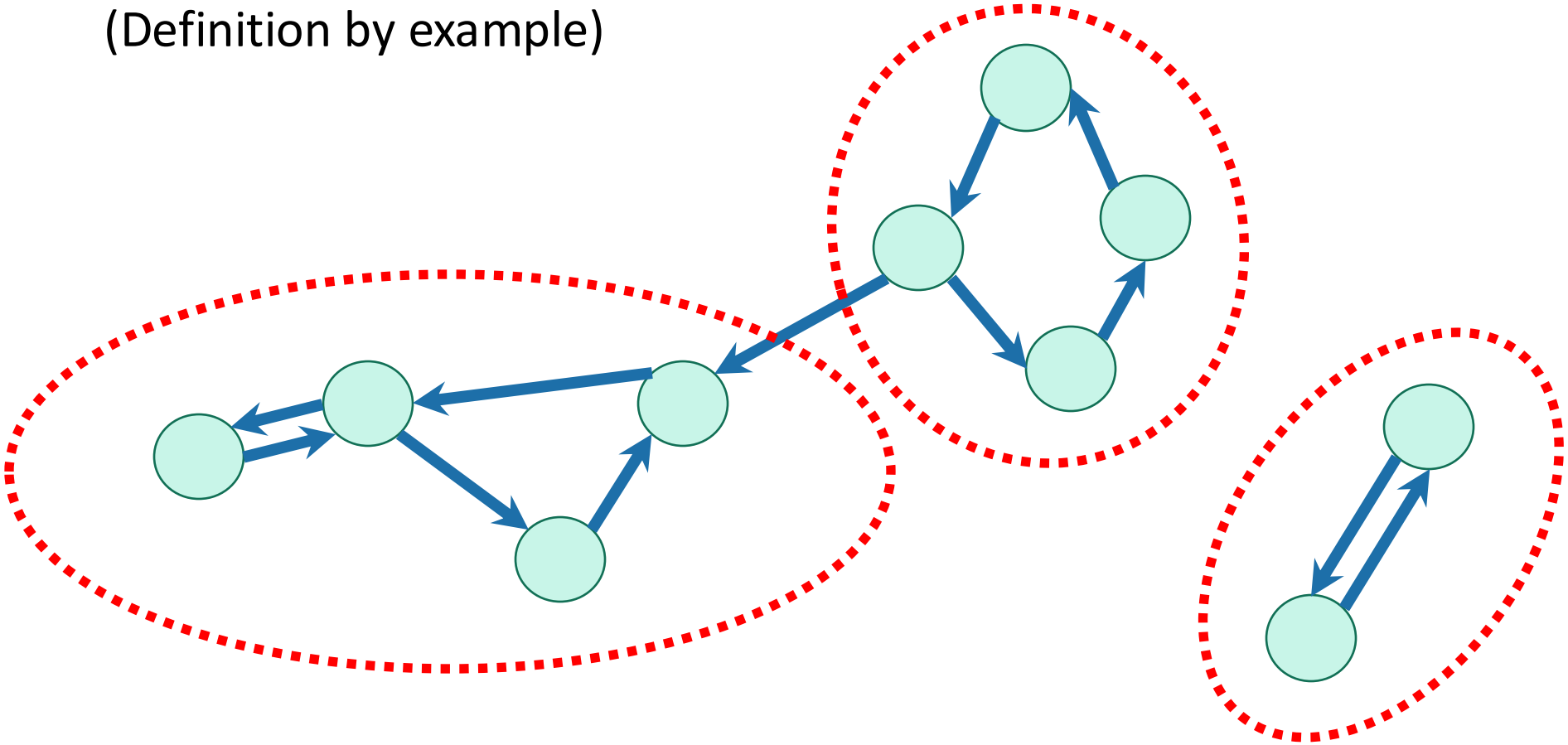
strongly connected



not strongly connected

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

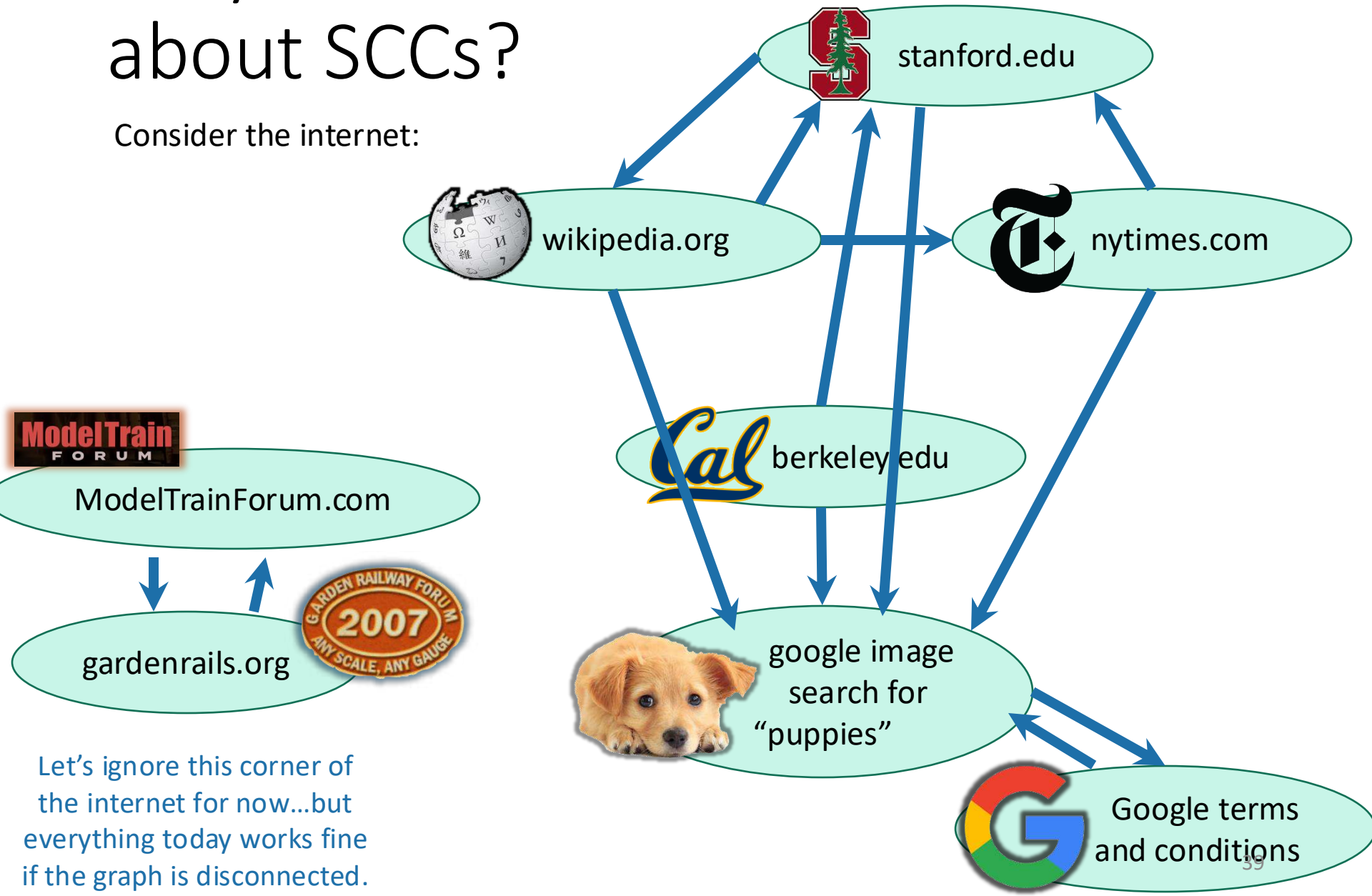
(Definition by example)



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

Why do we care about SCCs?

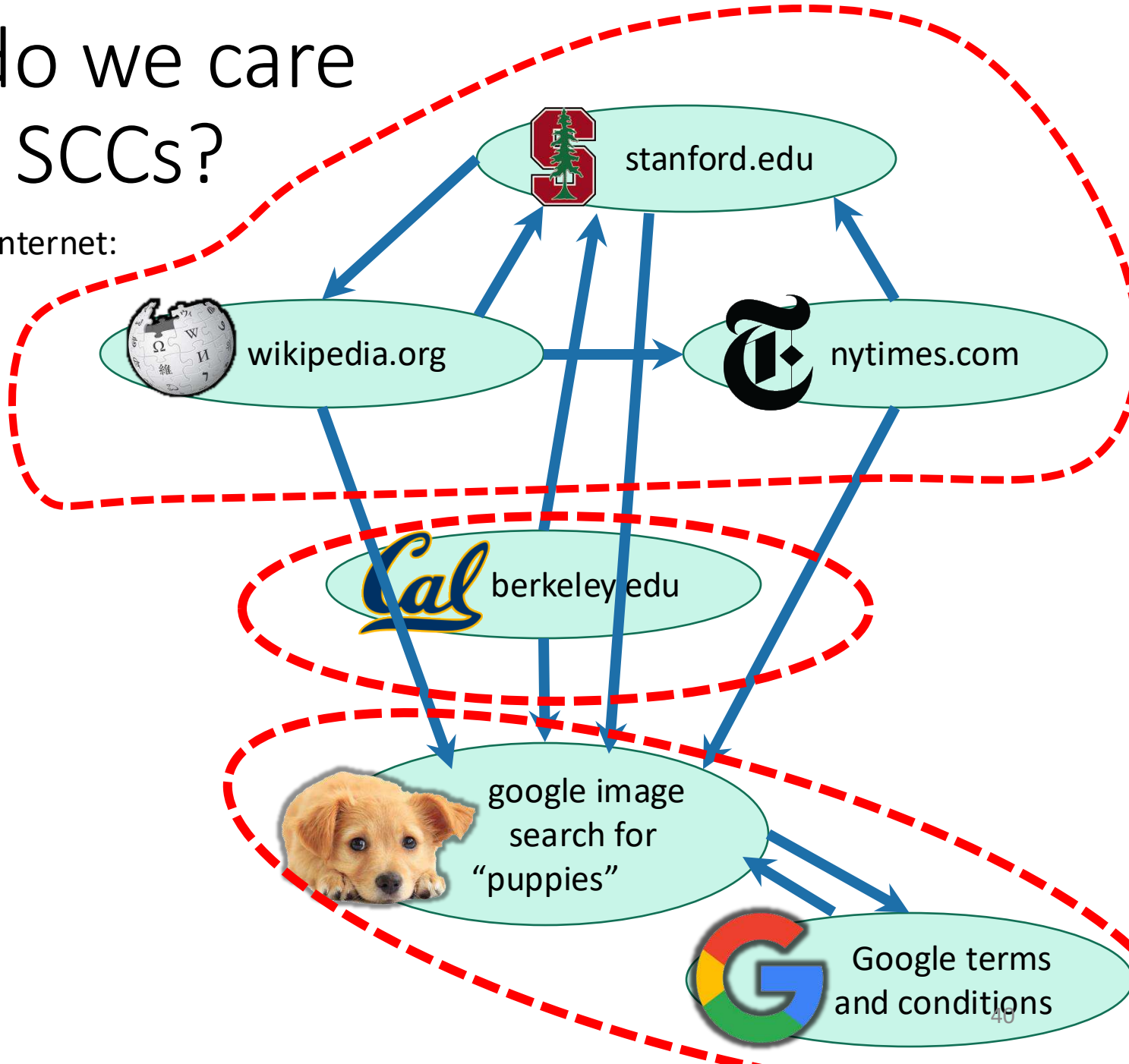
Consider the internet:



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

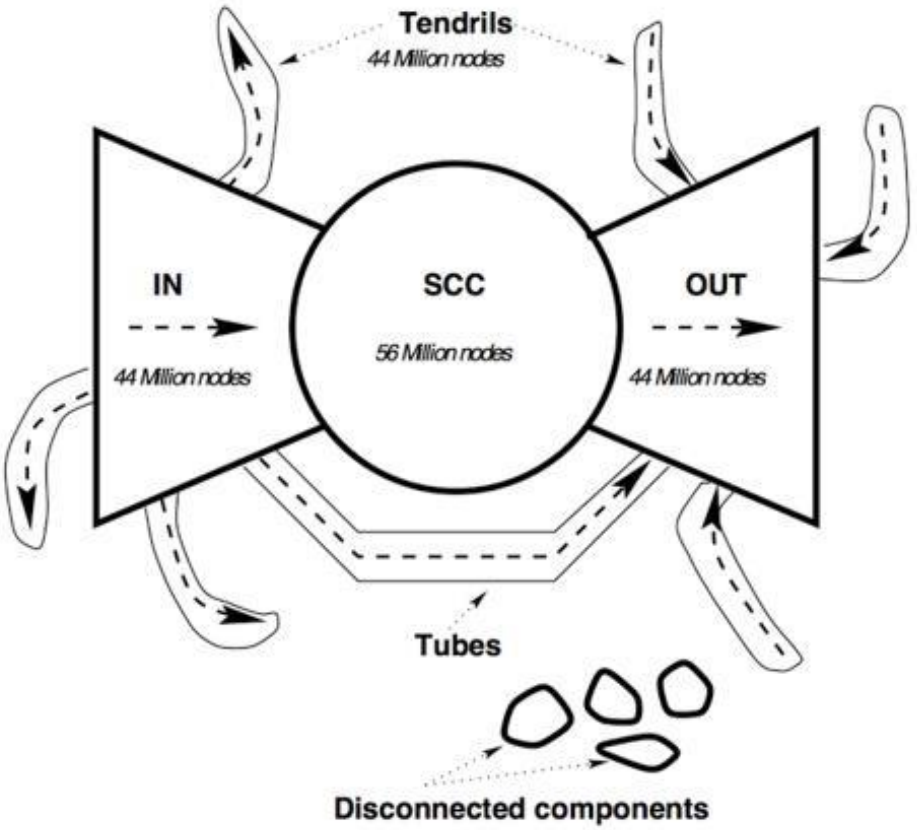
Why do we care about SCCs?

Consider the internet:



Aside: What the internet graph actually looks like

- Circa 2000[1]:

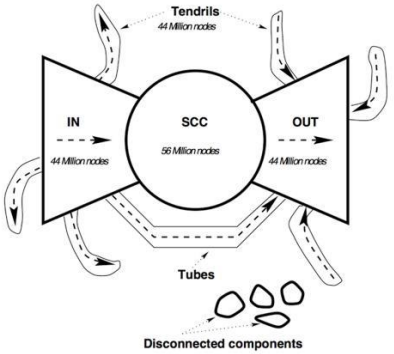


[1] "Graph Structure in the Web." Broder et al. *Computer Network*s. 2000.

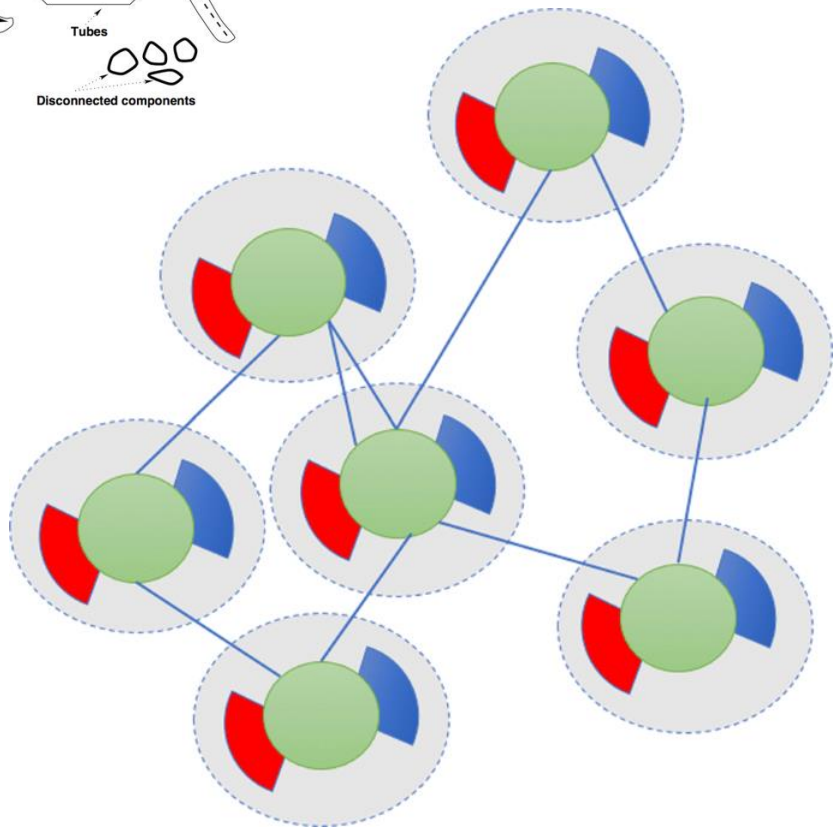
[2] "Local Bow-tie Structure of the Web." Fujita et al. *Applied Network Science*. 2019.

Aside: What the internet graph actually looks like

- Circa 2000[1]:



- Circa 2019[2]:

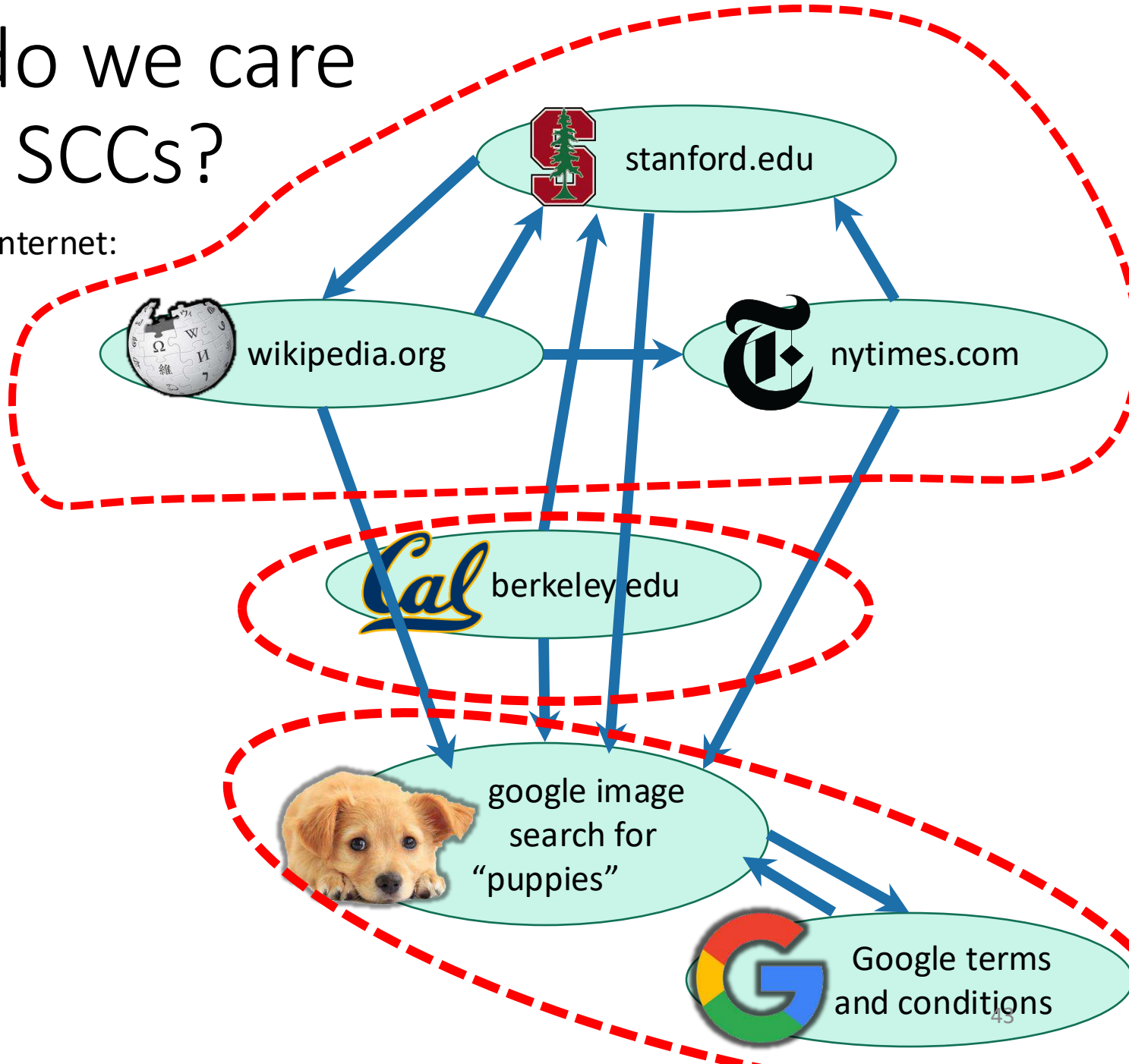


[1] "Graph Structure in the Web." Broder et al. *Computer Networks*. 2000.

[2] "Local Bow-tie Structure of the Web." Fujita et al. *Applied Network Science*. 2019.

Why do we care about SCCs?

Consider the internet:



Why do we care about SCCs?

- Strongly connected components tell you about **communities.**
- Lots of graph algorithms only make sense on SCCs, so we first want to find SCCs and run those algorithms on each part.
  - E.g., “Eigenvector Centrality” (related to PageRank)
- We will see later that the SCCs themselves form a DAG (the “SCC DAG” or “condensation graph”), which can also be useful to analyze!

How to find SCCs?

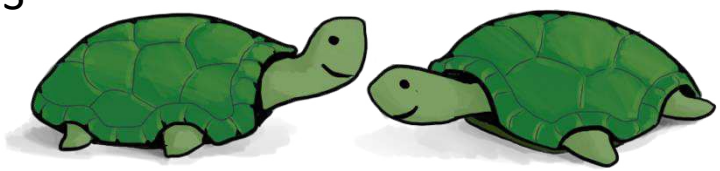
Try 1:

- 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?



One straightforward solution

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

- 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.

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

Today

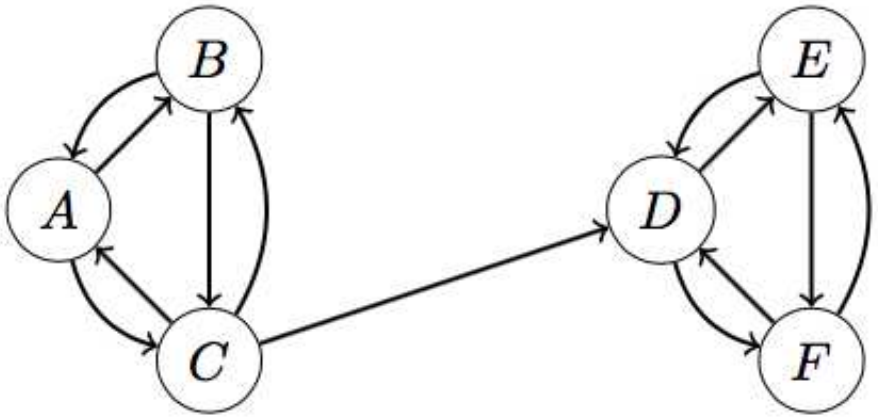
- We will see how to find strongly connected components in time O(n+m)

- !!!!!

- This is called Kosaraju's algorithm.
  - Heads up: the textbook has a slightly different presentation.

Pre-Lecture exercise

- What do you get when you run DFS from A?
- What about from D?



- Suggested algorithm: run DFS from the “right” place to identify SCCs
- Issue: what is the “right” place?

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)
```

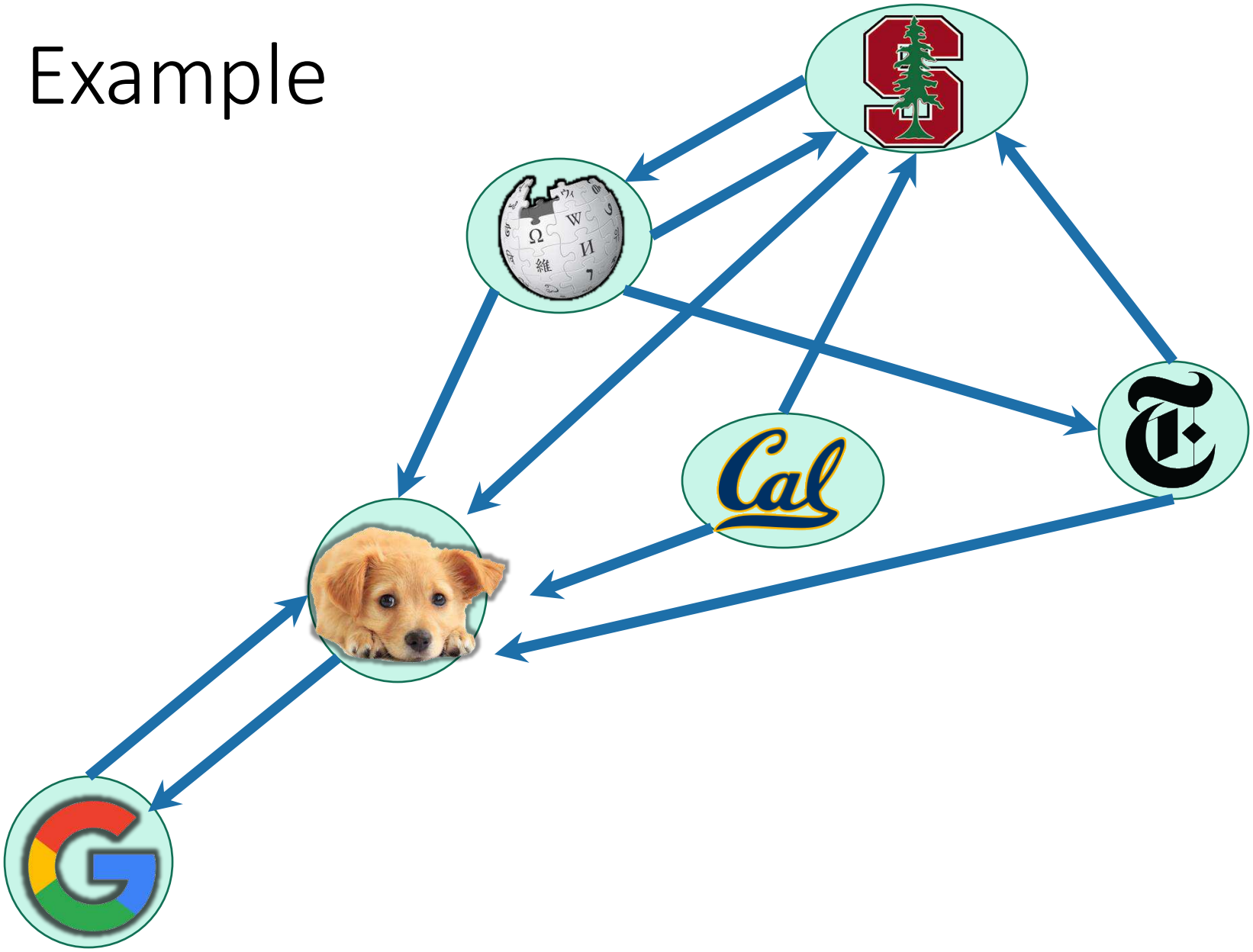
```
CS161Graph with:
          Vertices:
         Stanford,Wikipedia,NYTimes,Berkeley,Puppies,Google,
          Edges:
         (Stanford,Wikipedia) (Stanford,Puppies) (Wikipedia,Stanford) (Wik
ipedia,NYTimes) (Wikipedia,Puppies) (NYTimes,Stanford) (NYTimes,Puppies)
(Berkeley,Stanford) (Berkeley,Puppies)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

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

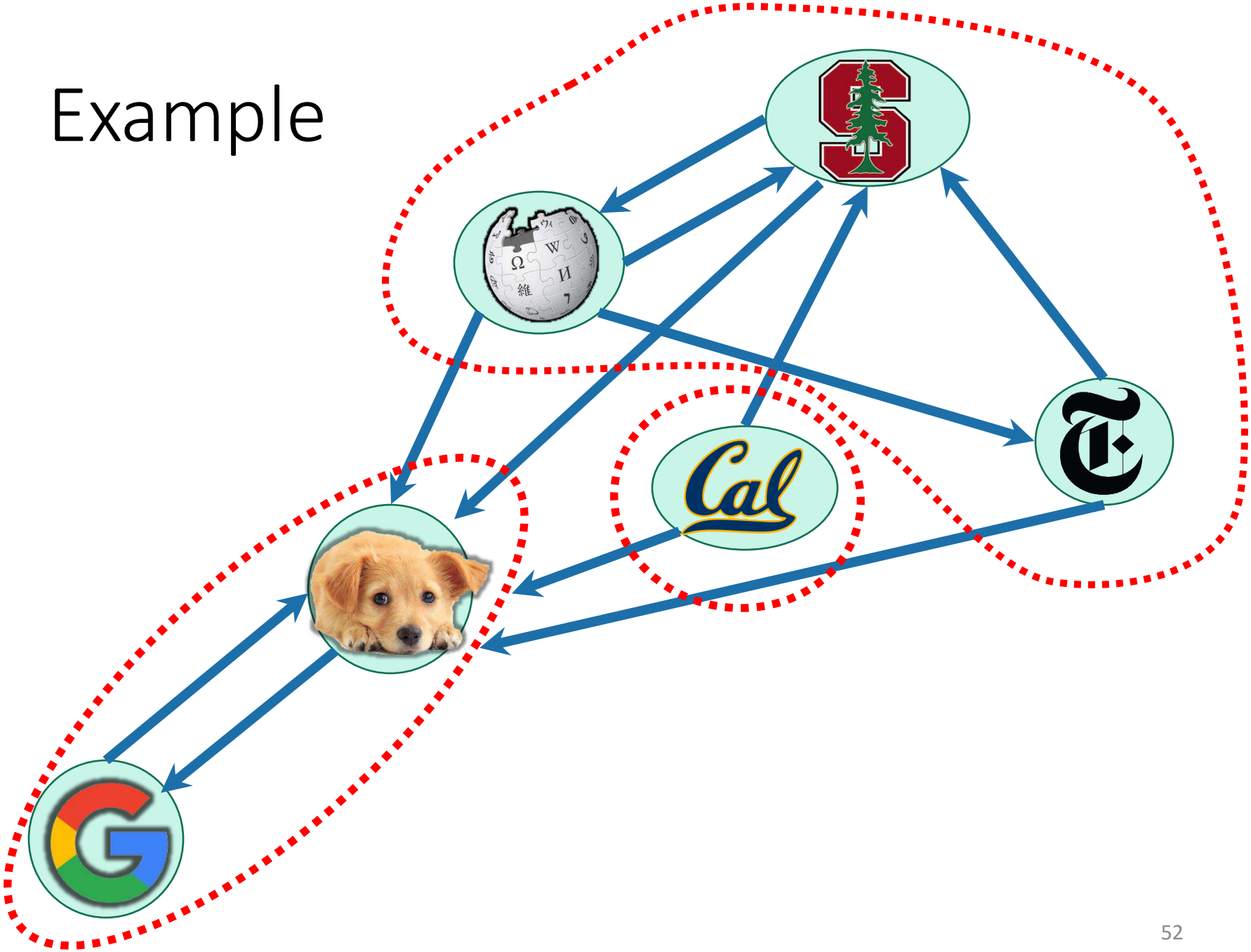
```
['Berkeley']
['Stanford', 'NYTimes', 'Wikipedia']
['Puppies', 'Google']
```

But let's dig into it…

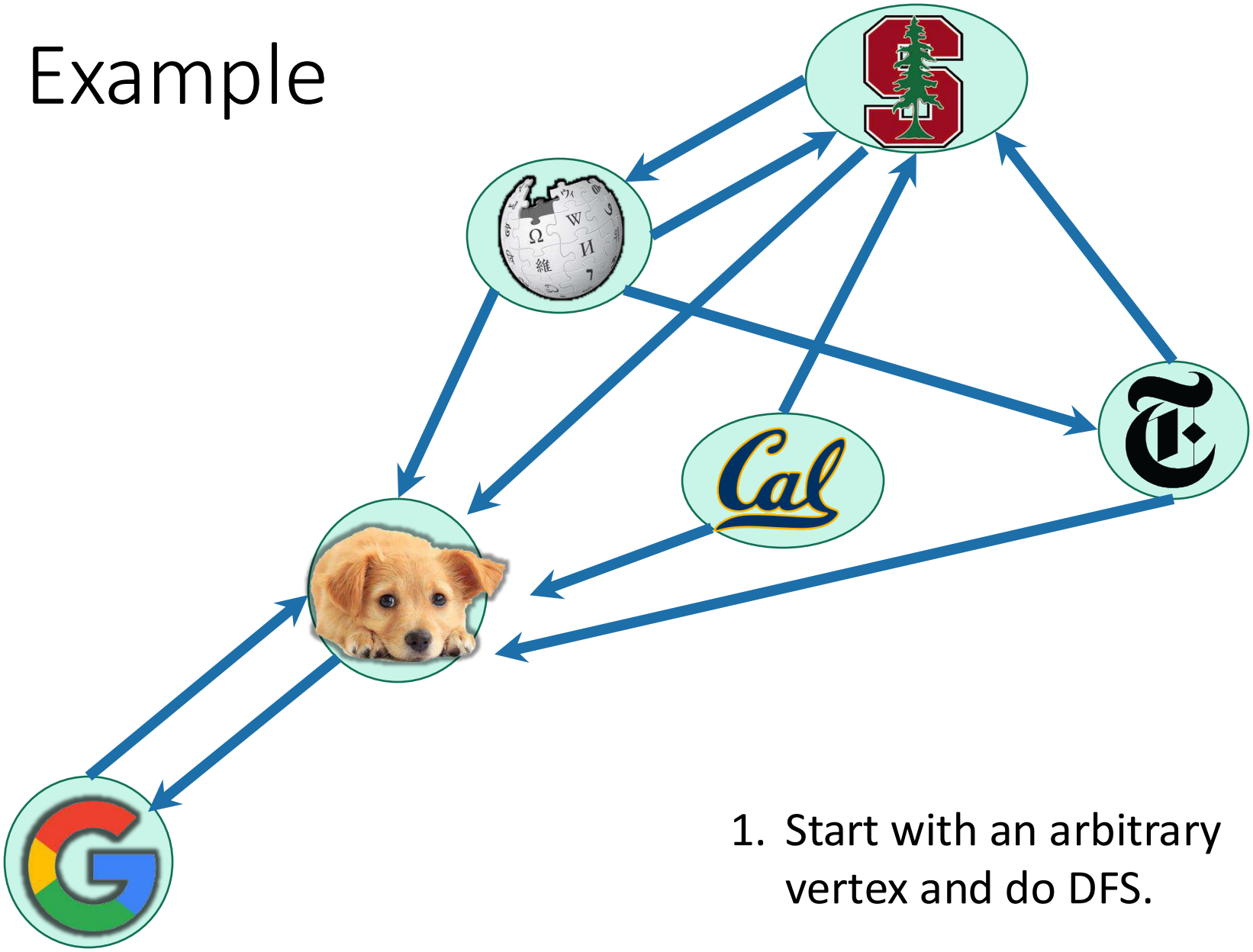
Example



Example

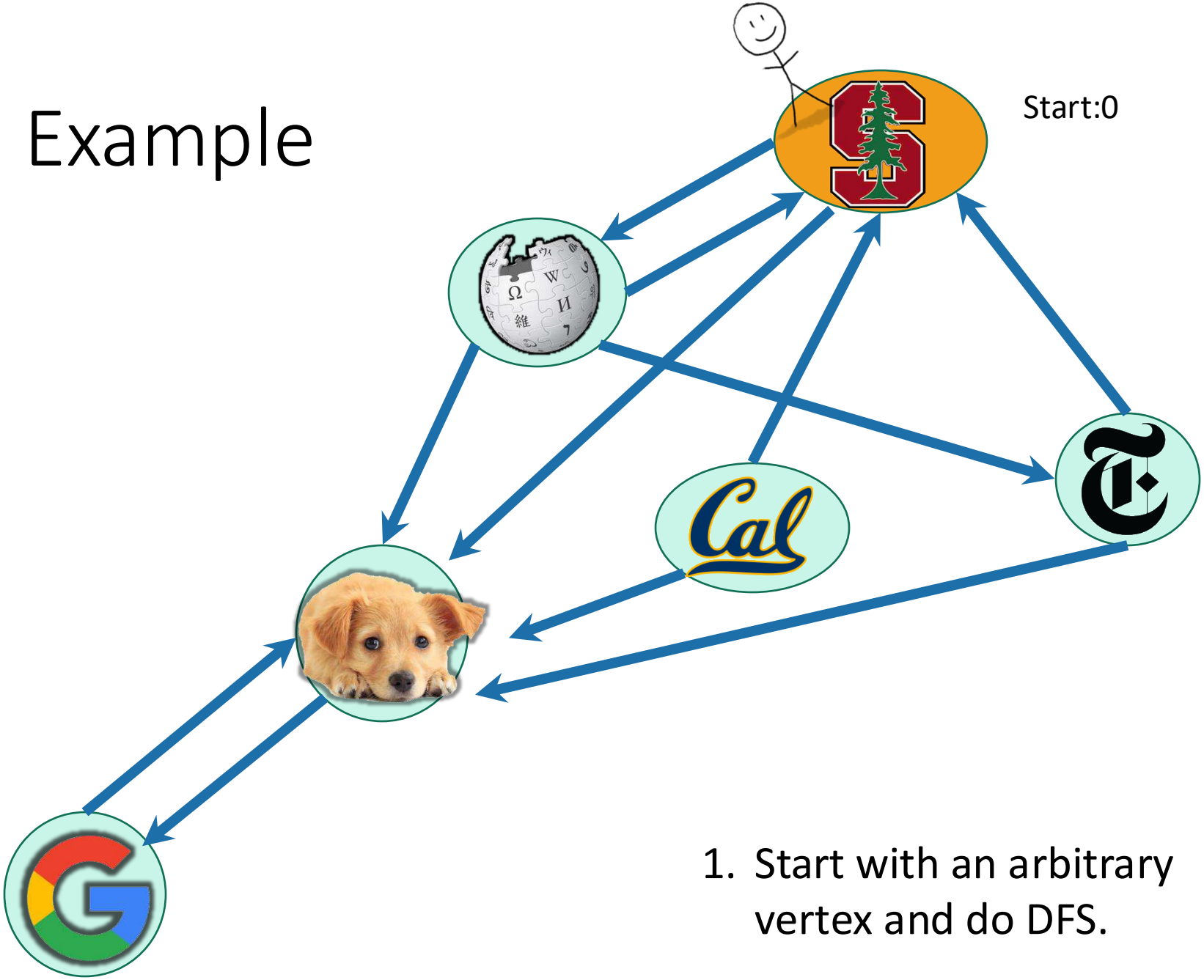


Example



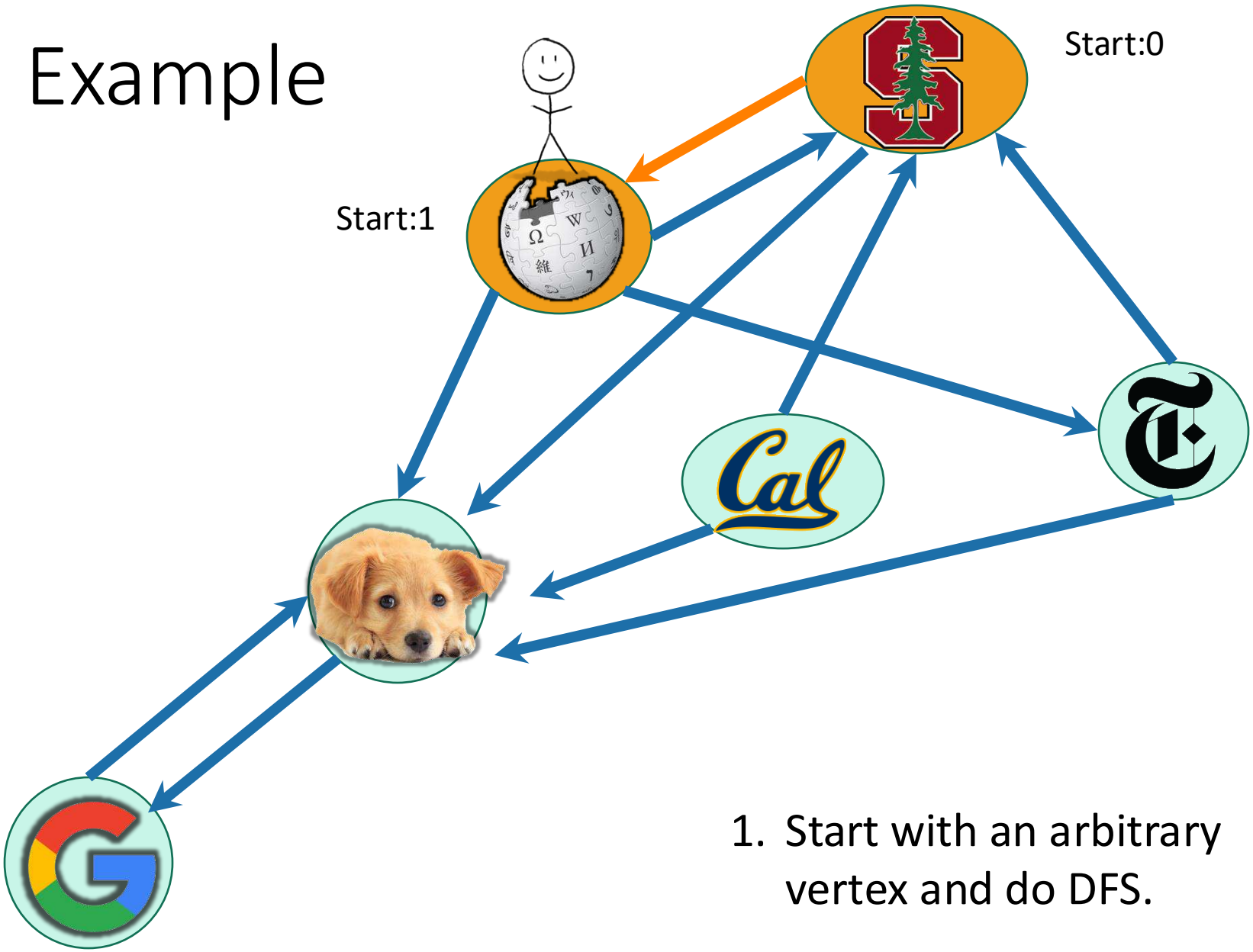
1. Start with an arbitrary vertex and do DFS.

Example



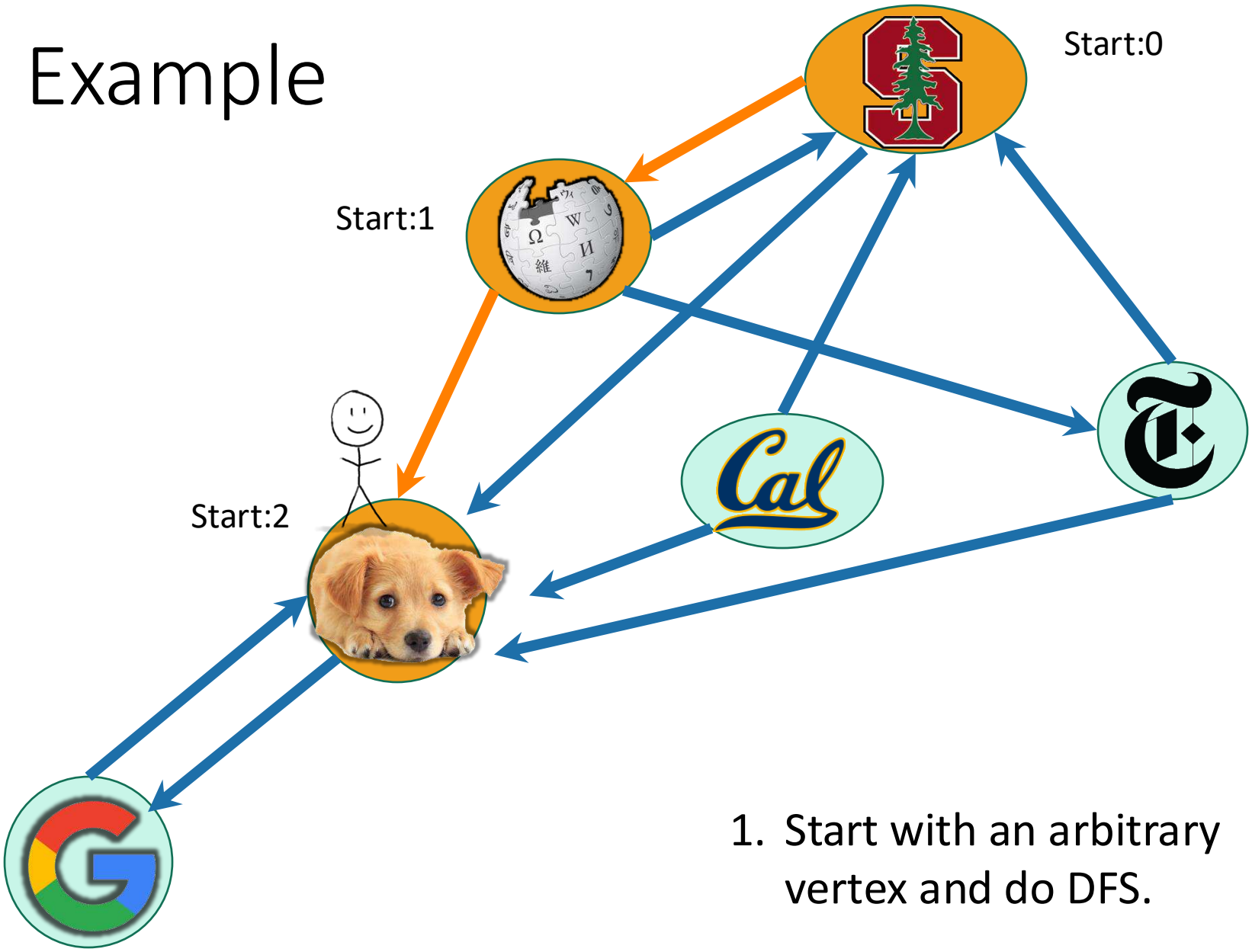
1. Start with an arbitrary vertex and do DFS.

Example



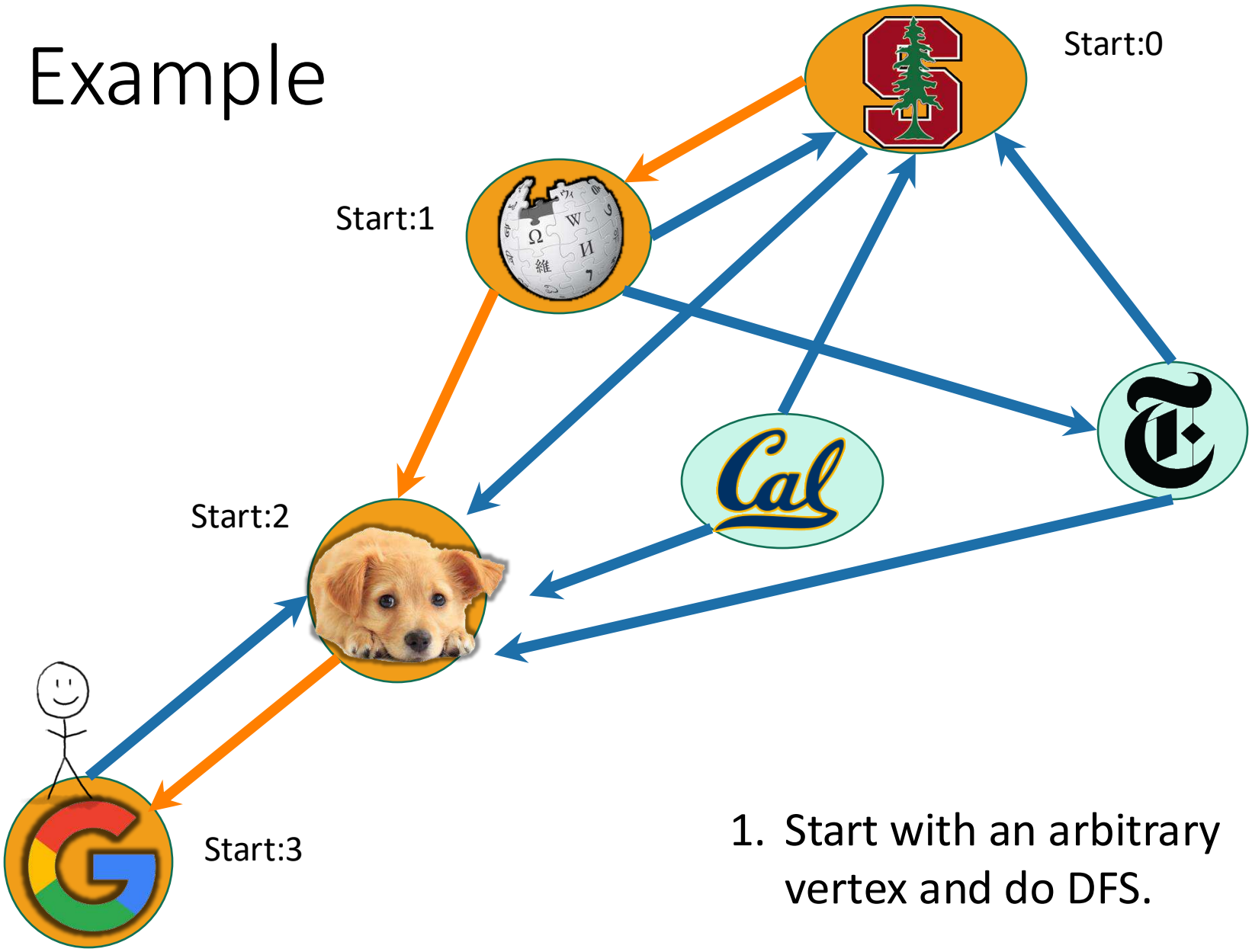
1. Start with an arbitrary vertex and do DFS.

Example



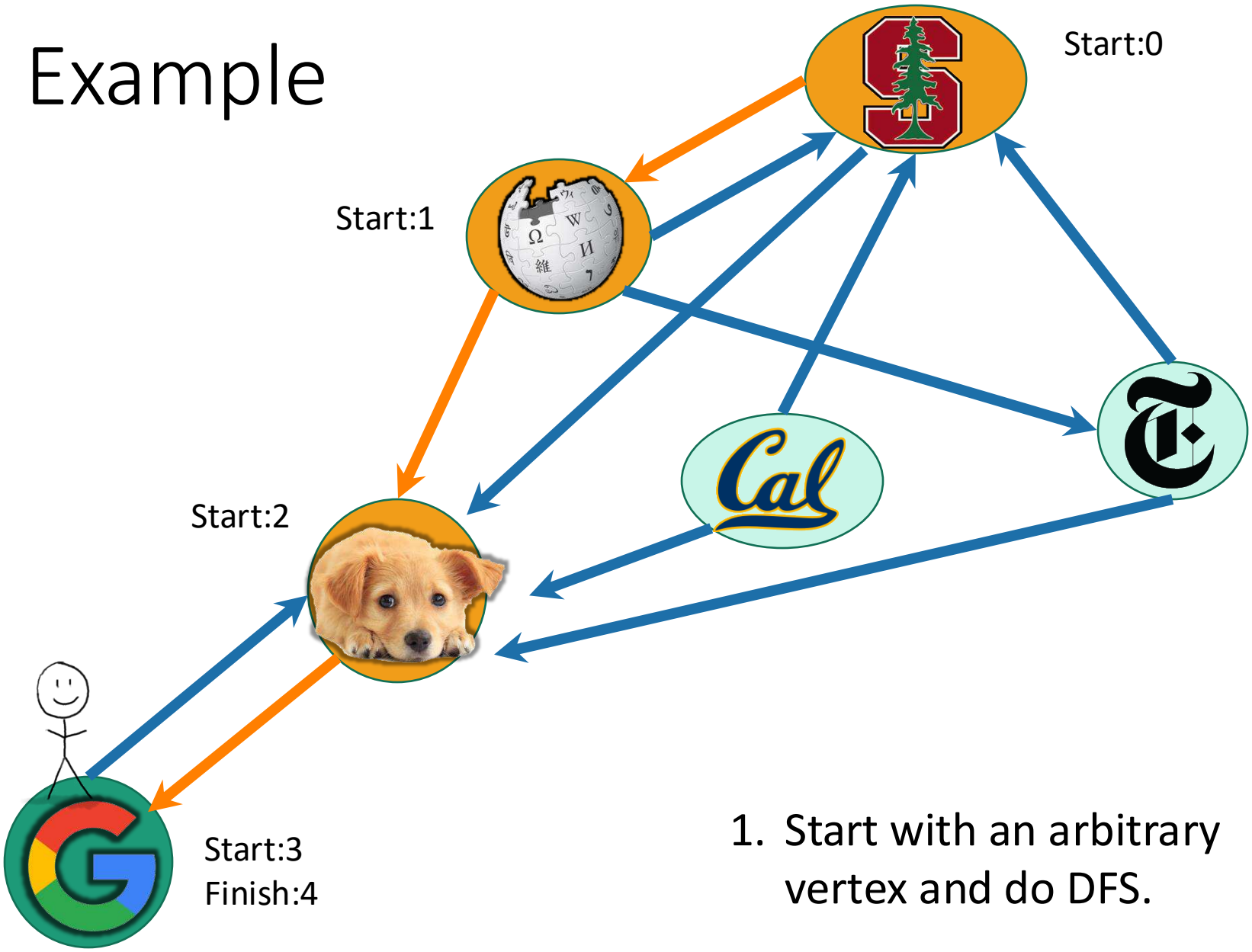
1. Start with an arbitrary vertex and do DFS.

Example



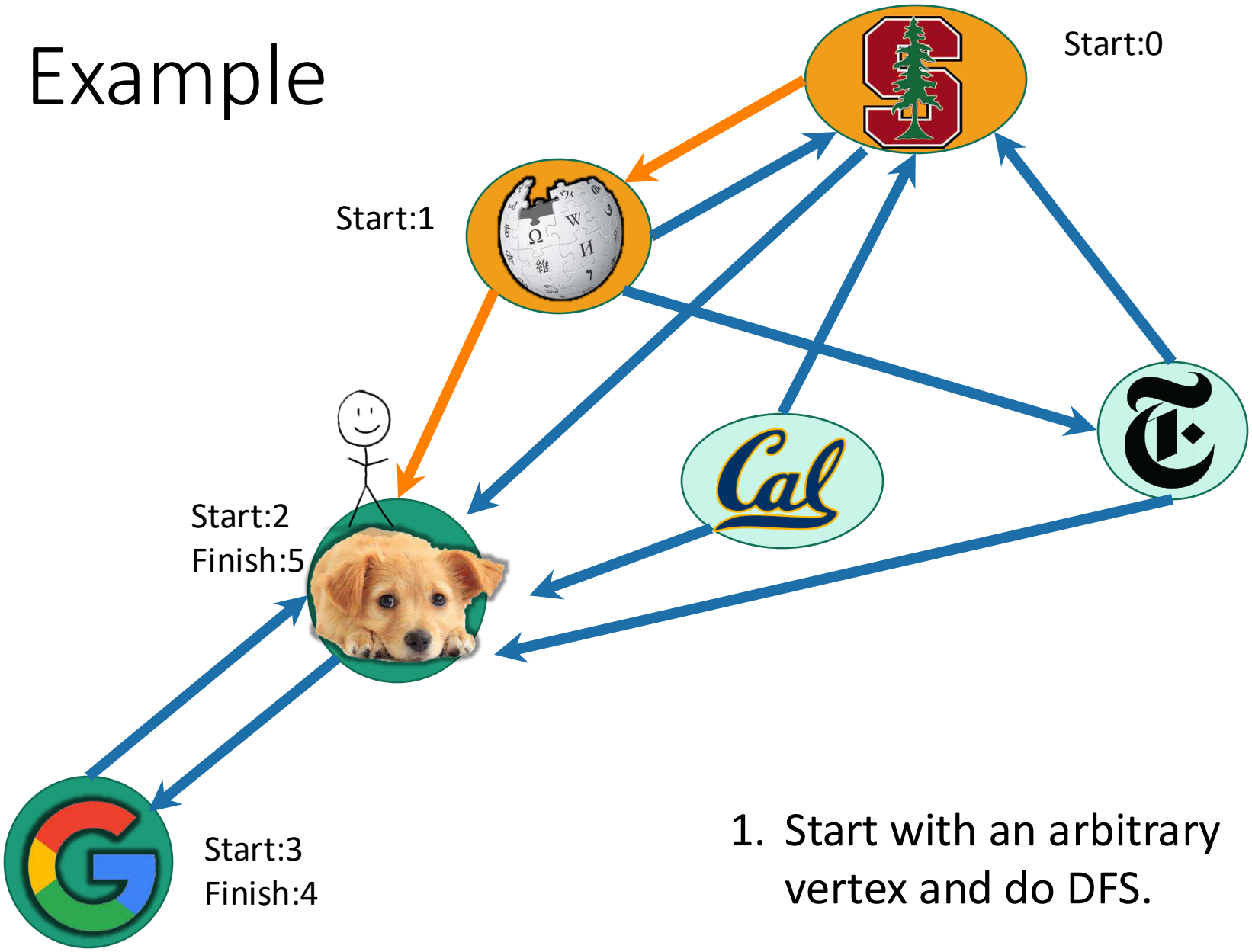
1. Start with an arbitrary vertex and do DFS.

Example



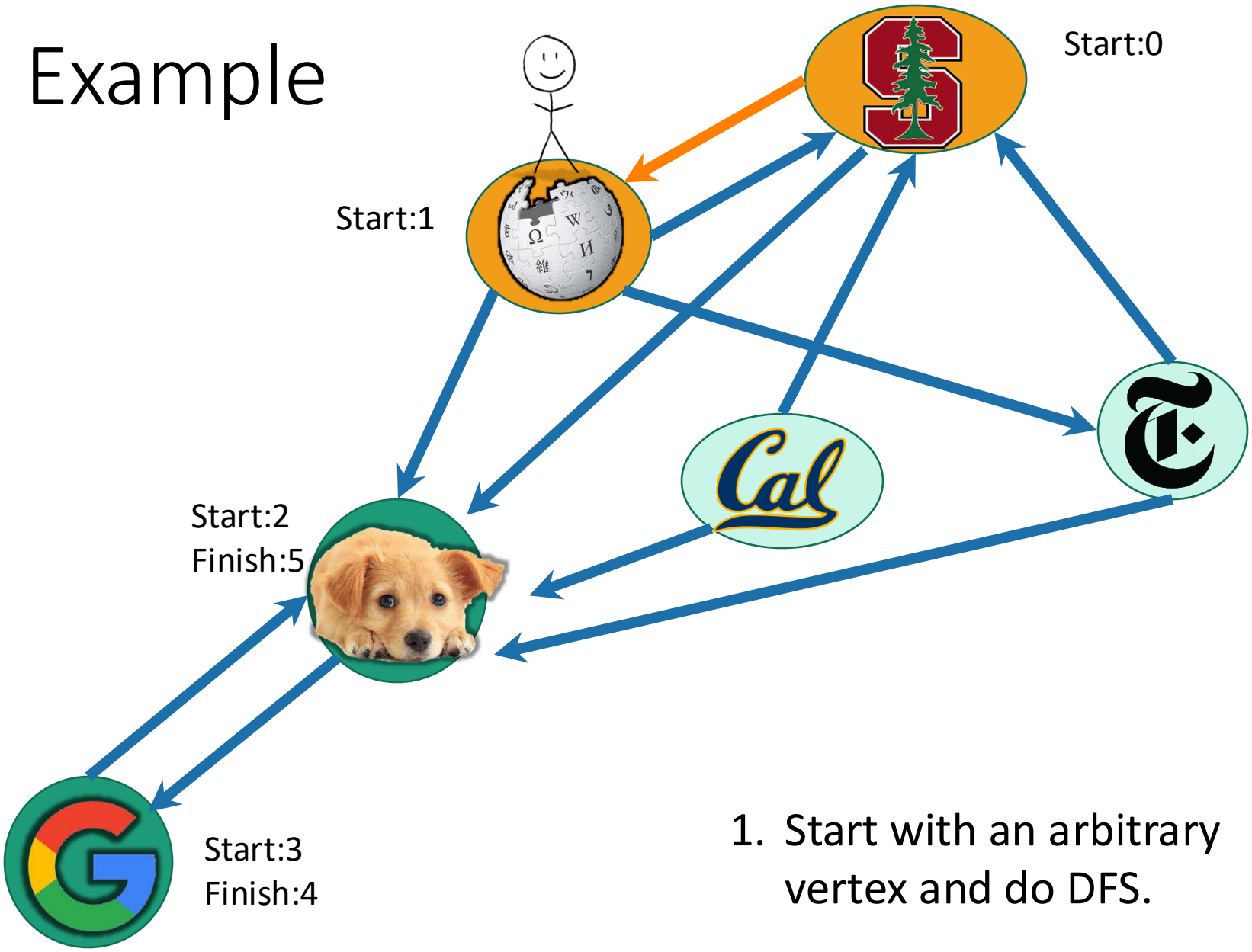
1. Start with an arbitrary vertex and do DFS.

Example



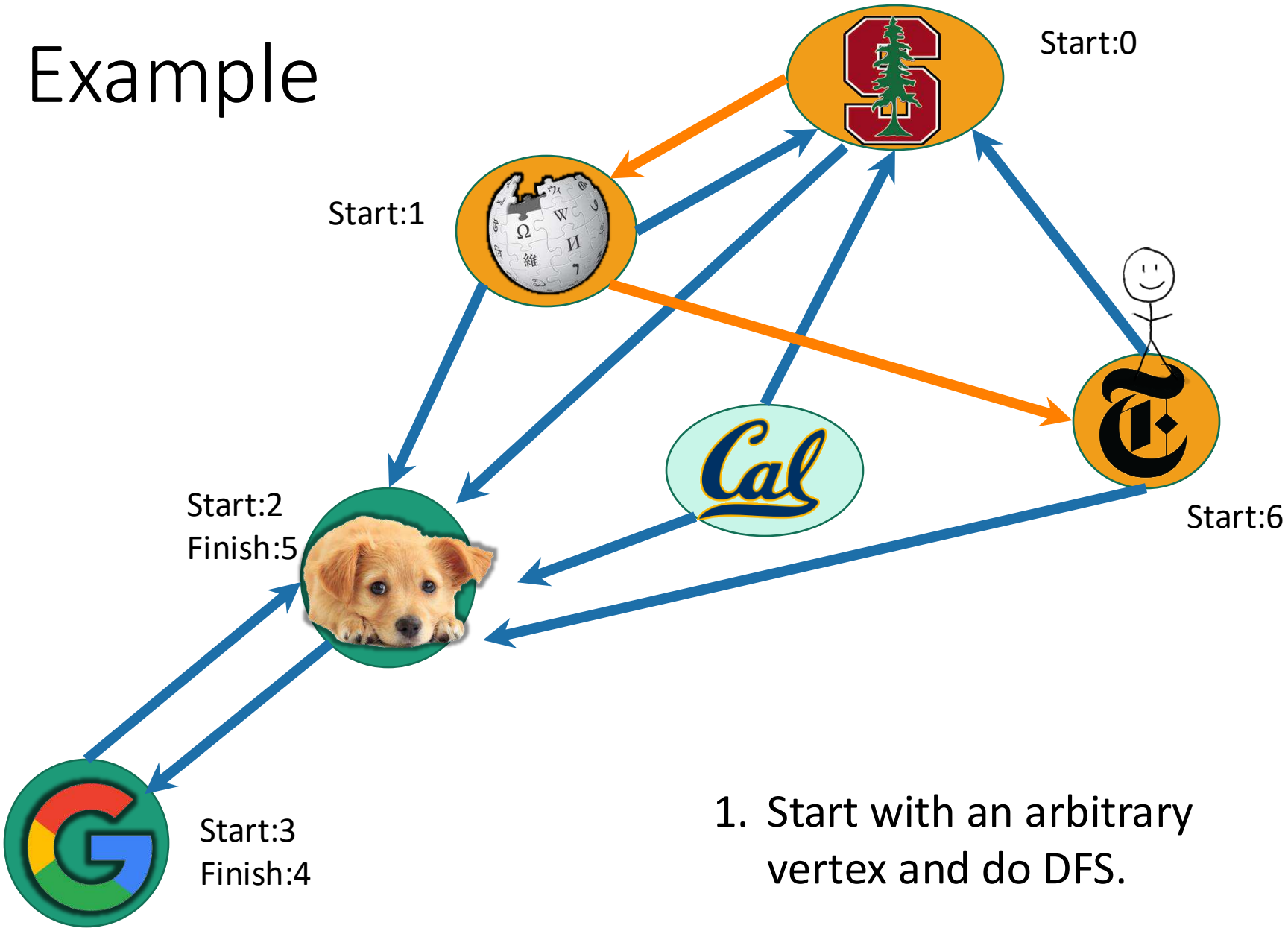
1. Start with an arbitrary vertex and do DFS.

Example



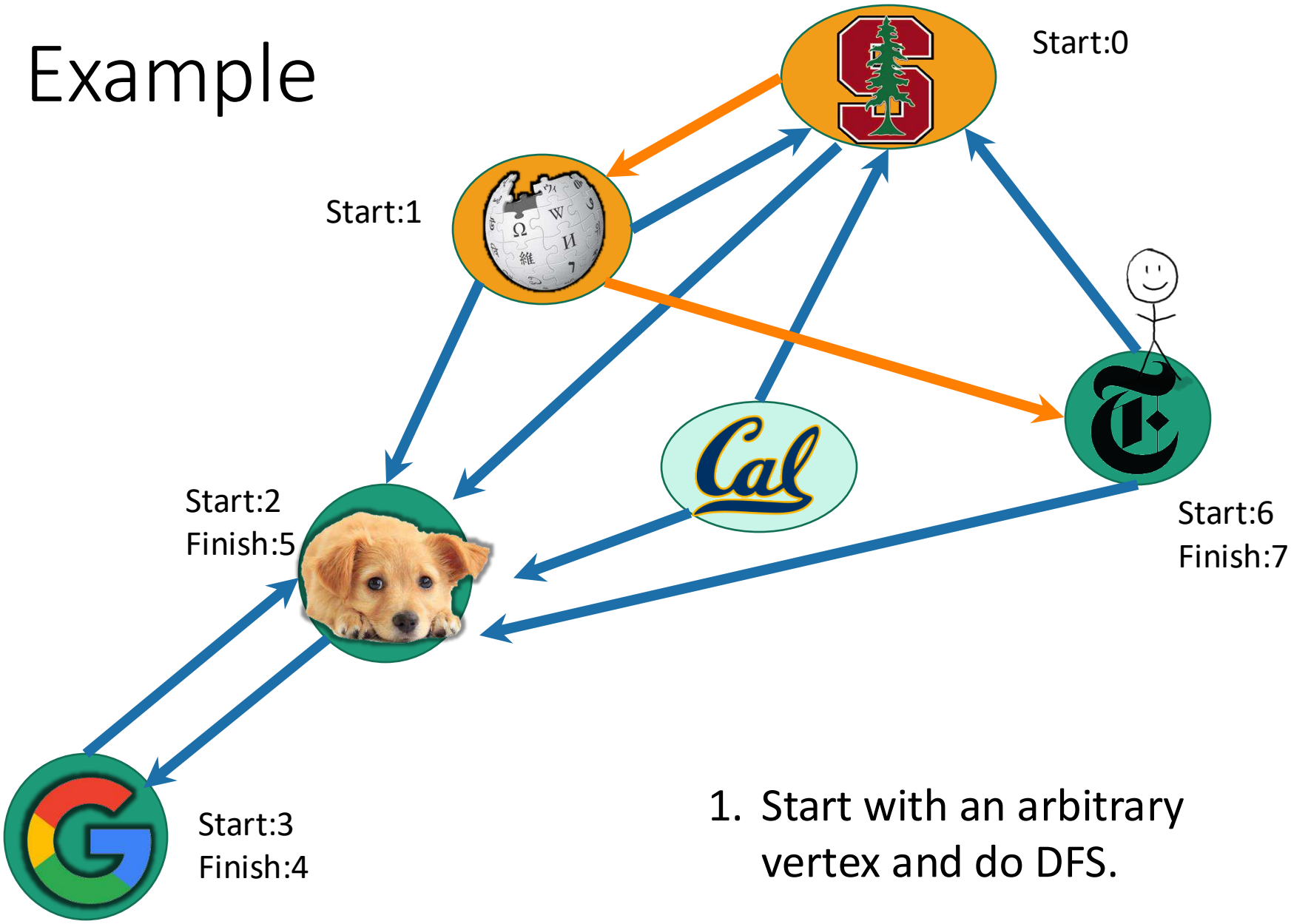
1. Start with an arbitrary vertex and do DFS.

Example



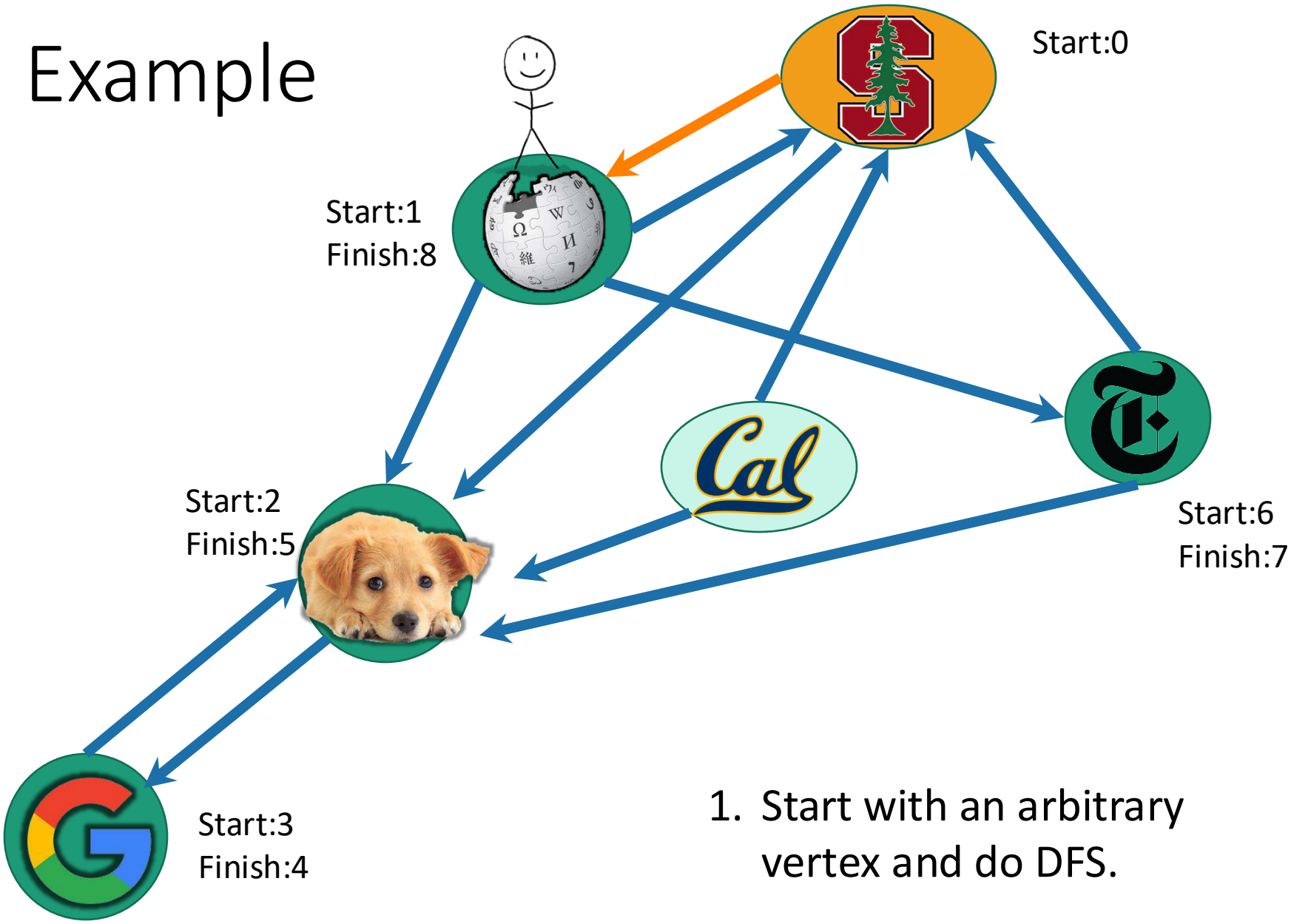
1. Start with an arbitrary vertex and do DFS.

Example



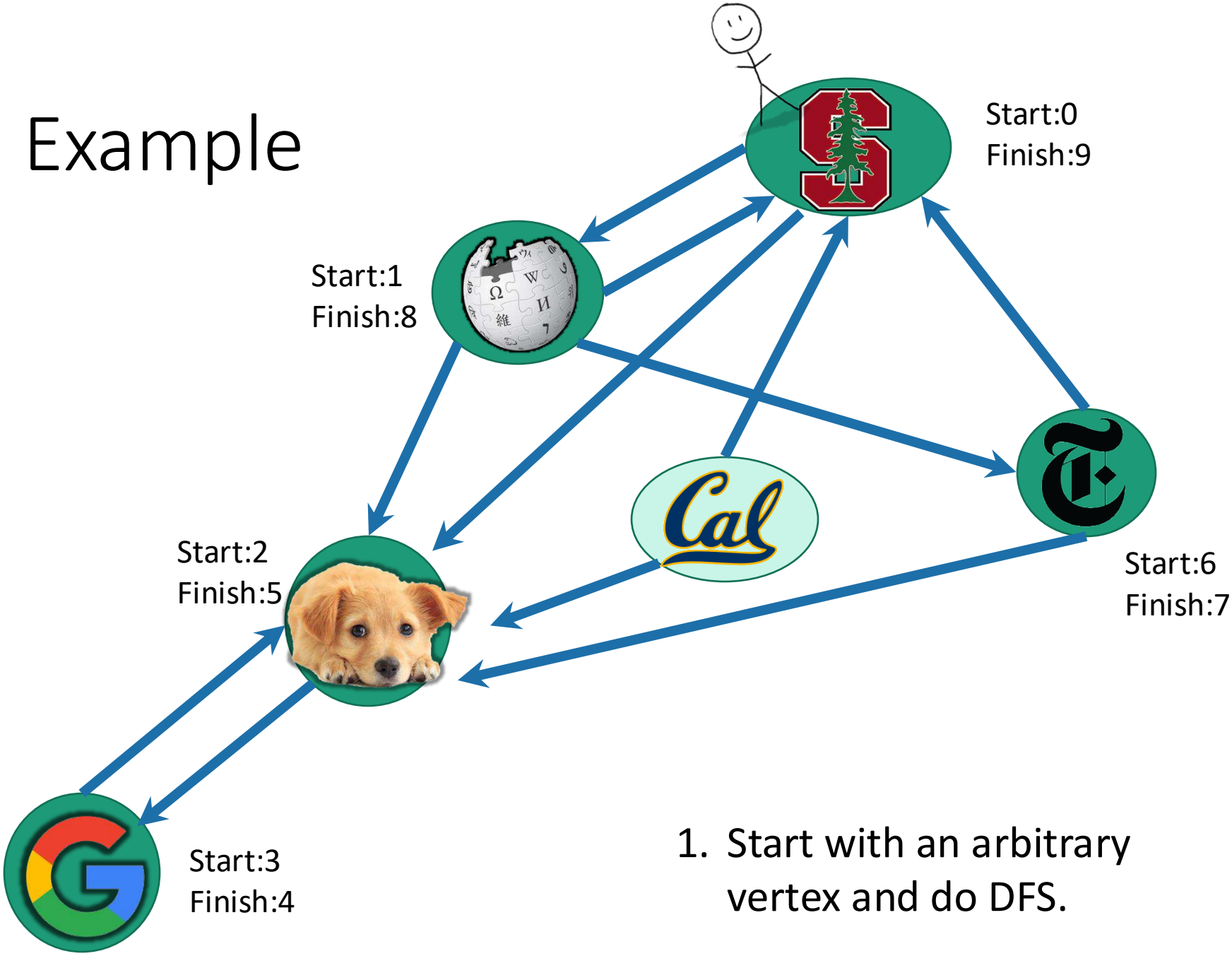
1. Start with an arbitrary vertex and do DFS.

Example



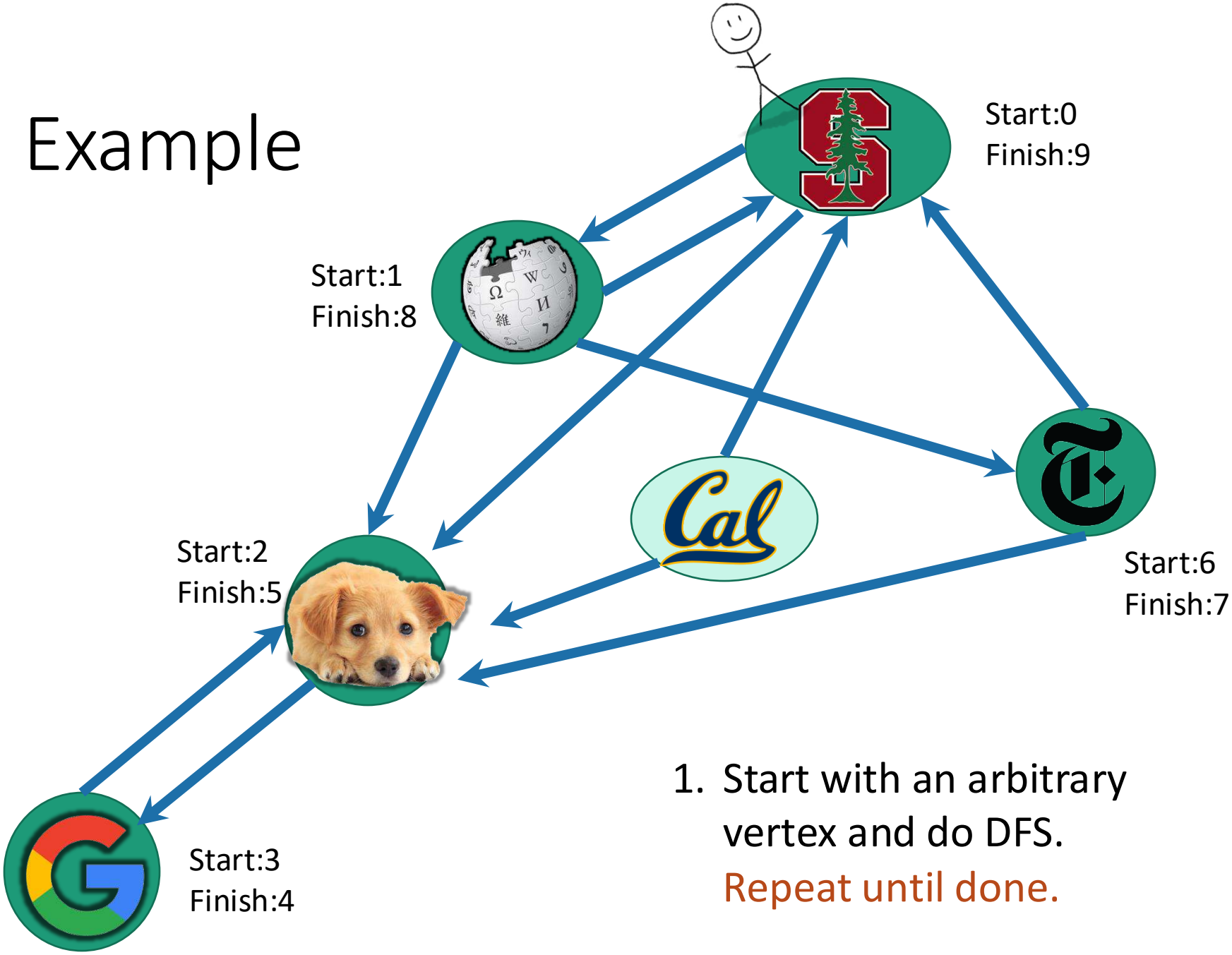
1. Start with an arbitrary vertex and do DFS.

Example



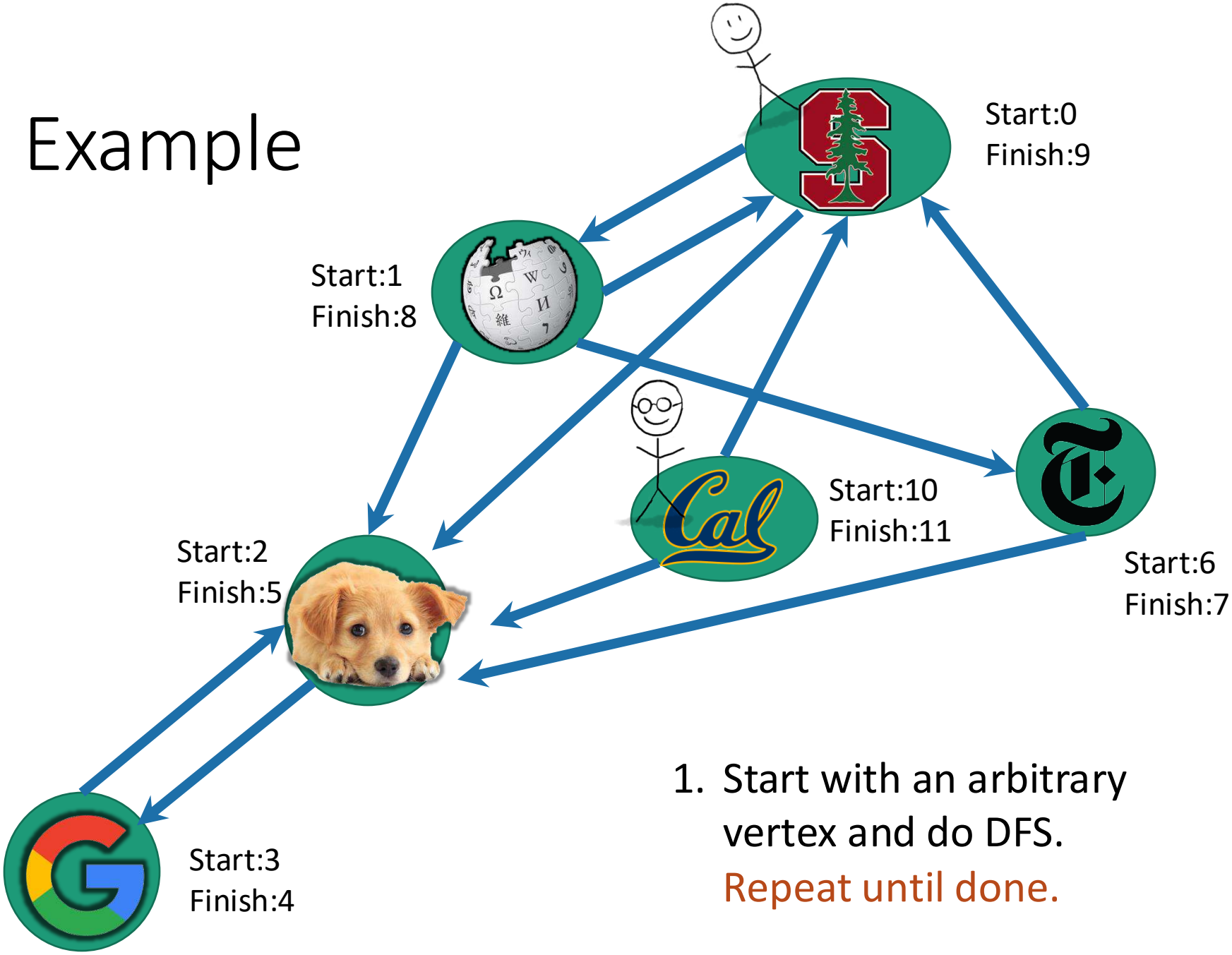
1. Start with an arbitrary vertex and do DFS.

Example



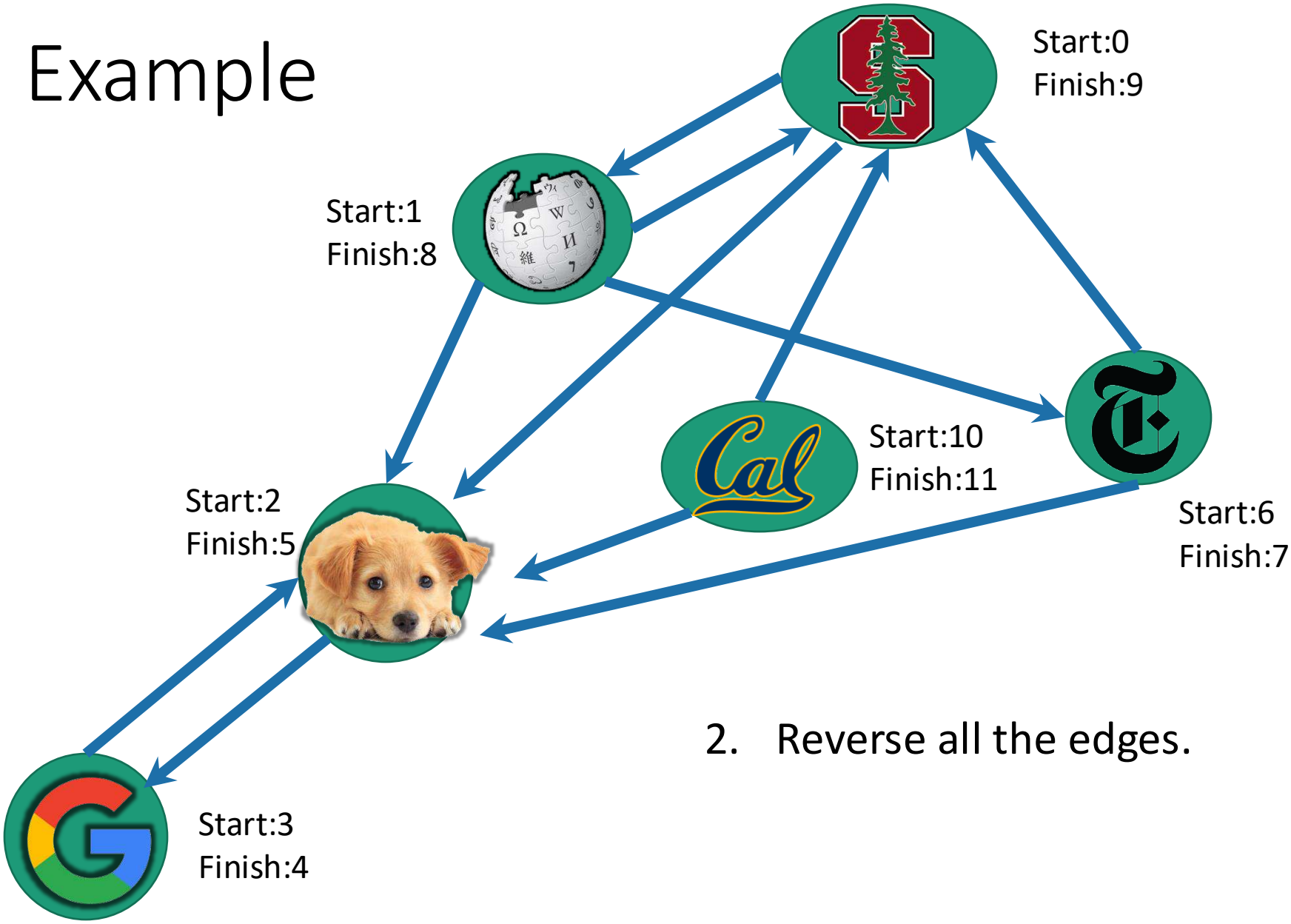
1. Start with an arbitrary vertex and do DFS.
   Repeat until done.

Example



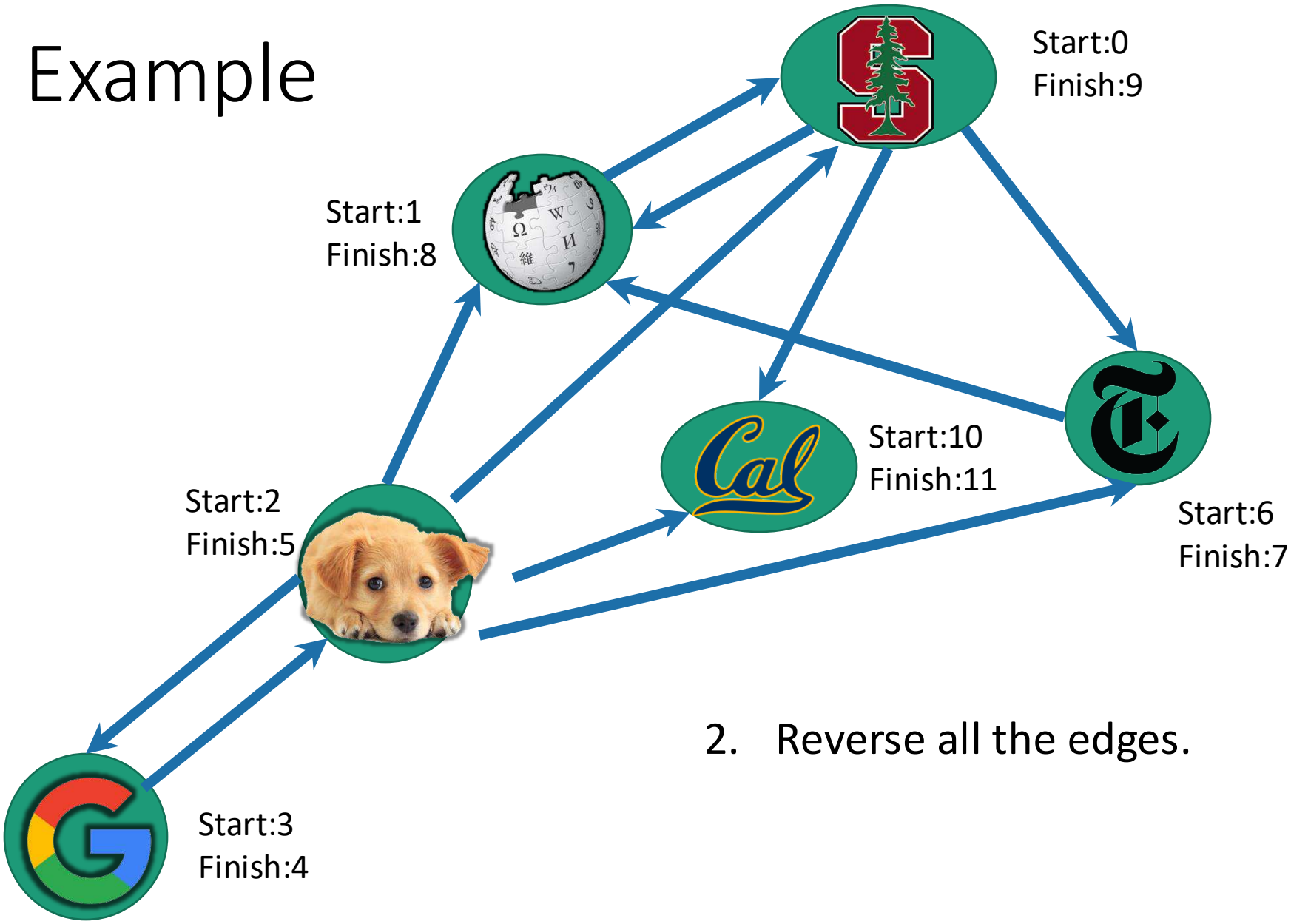
1. Start with an arbitrary vertex and do DFS.
   Repeat until done.

Example



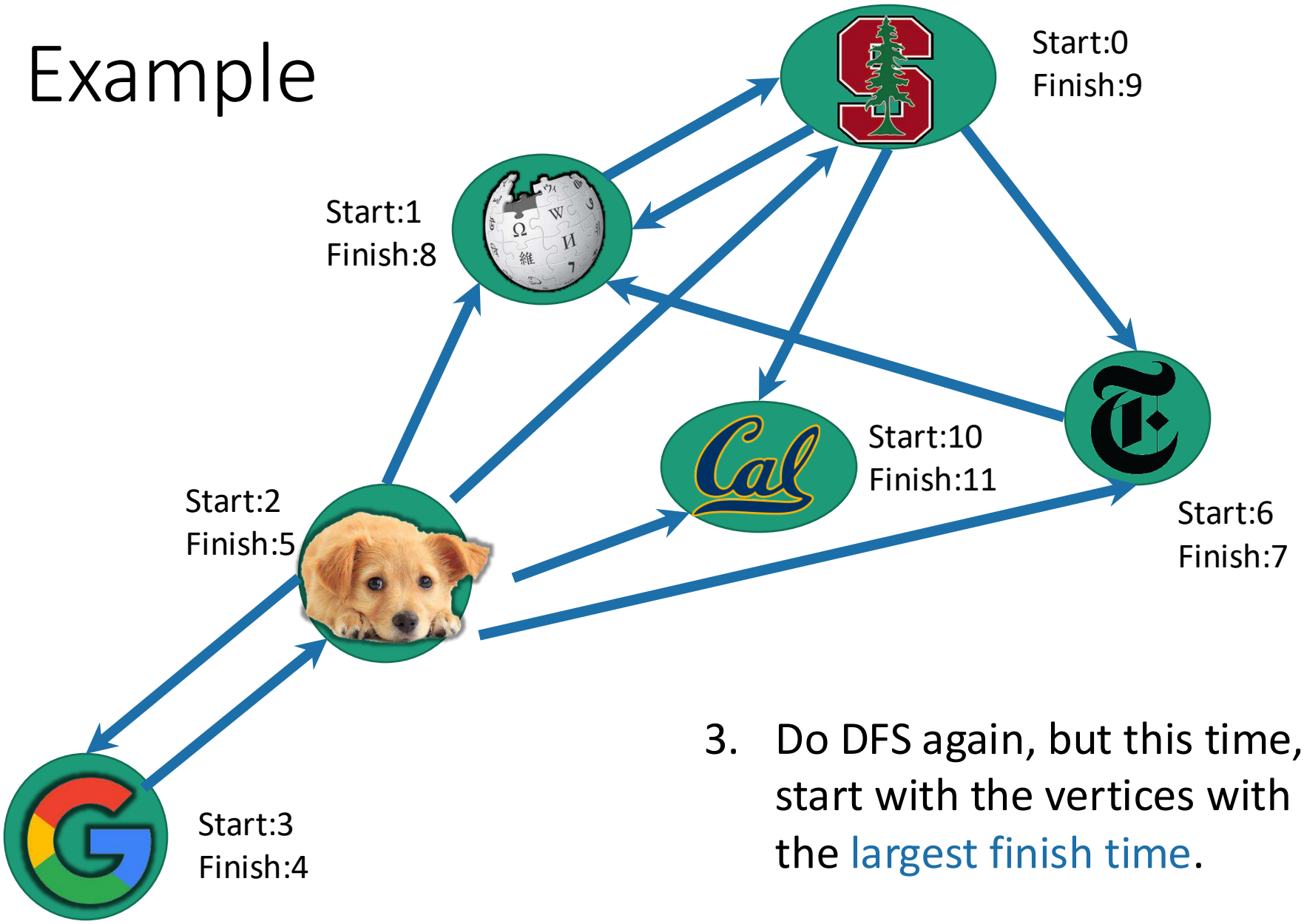
2. Reverse all the edges.

Example



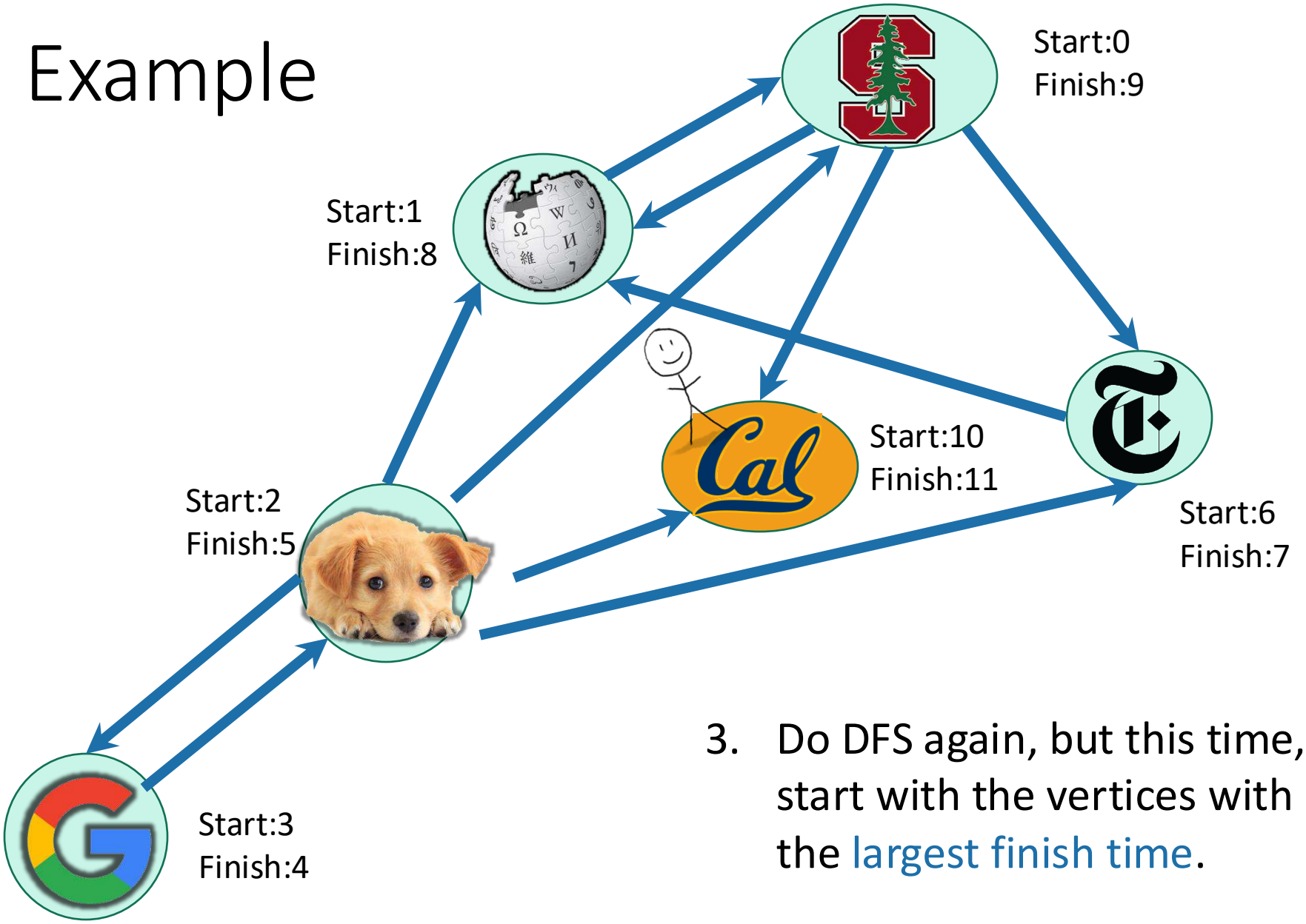
2. Reverse all the edges.

Example



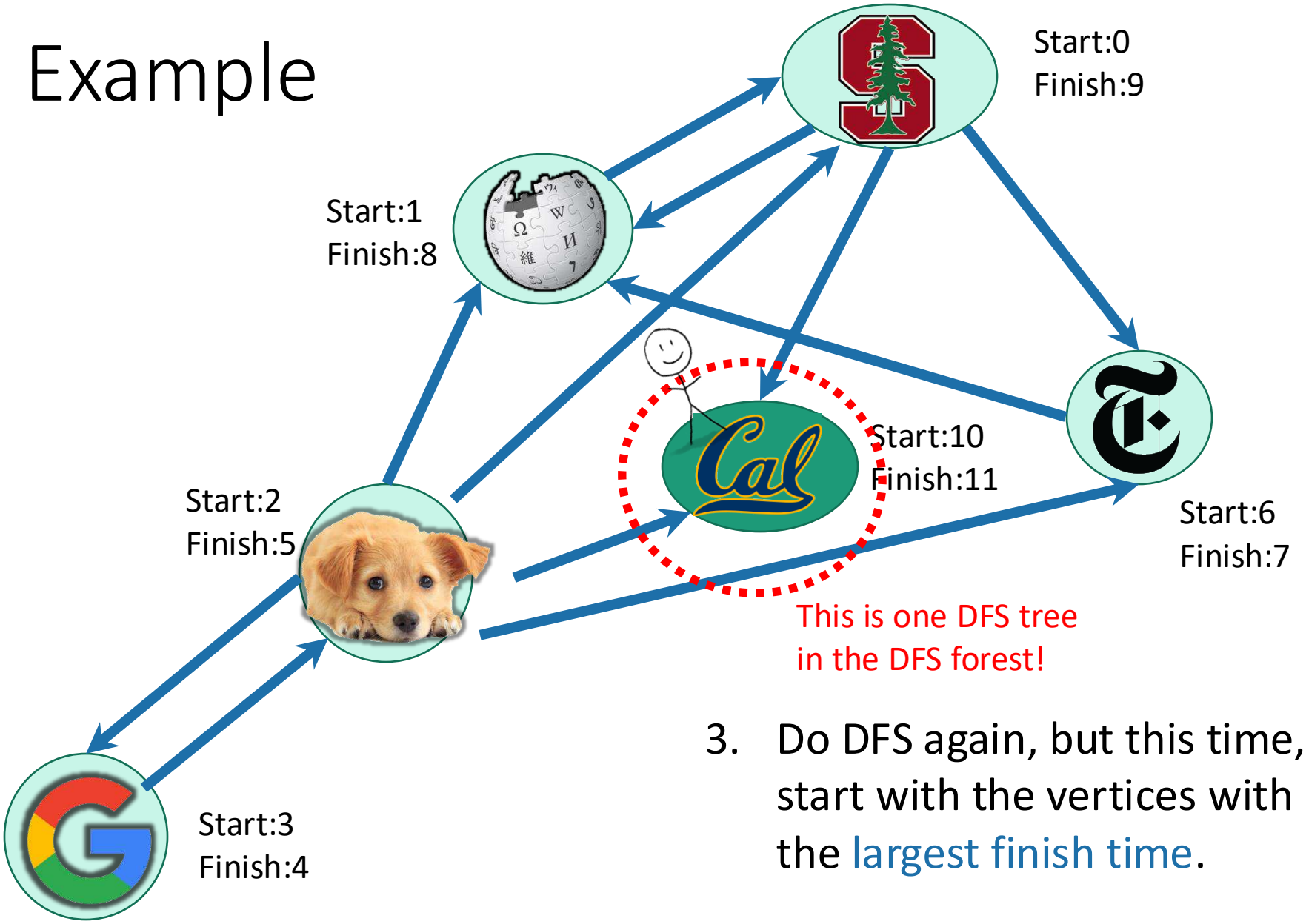
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



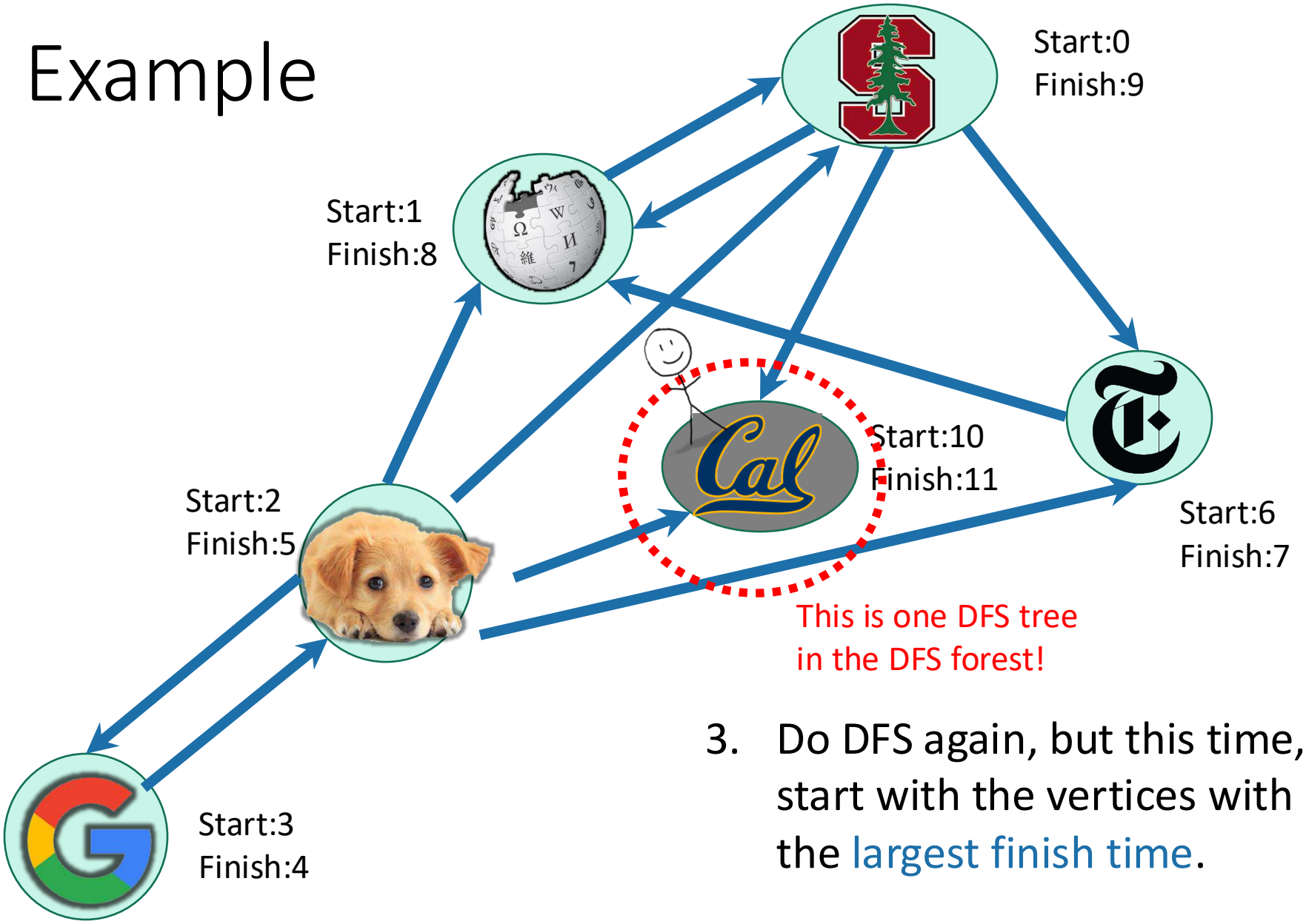
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



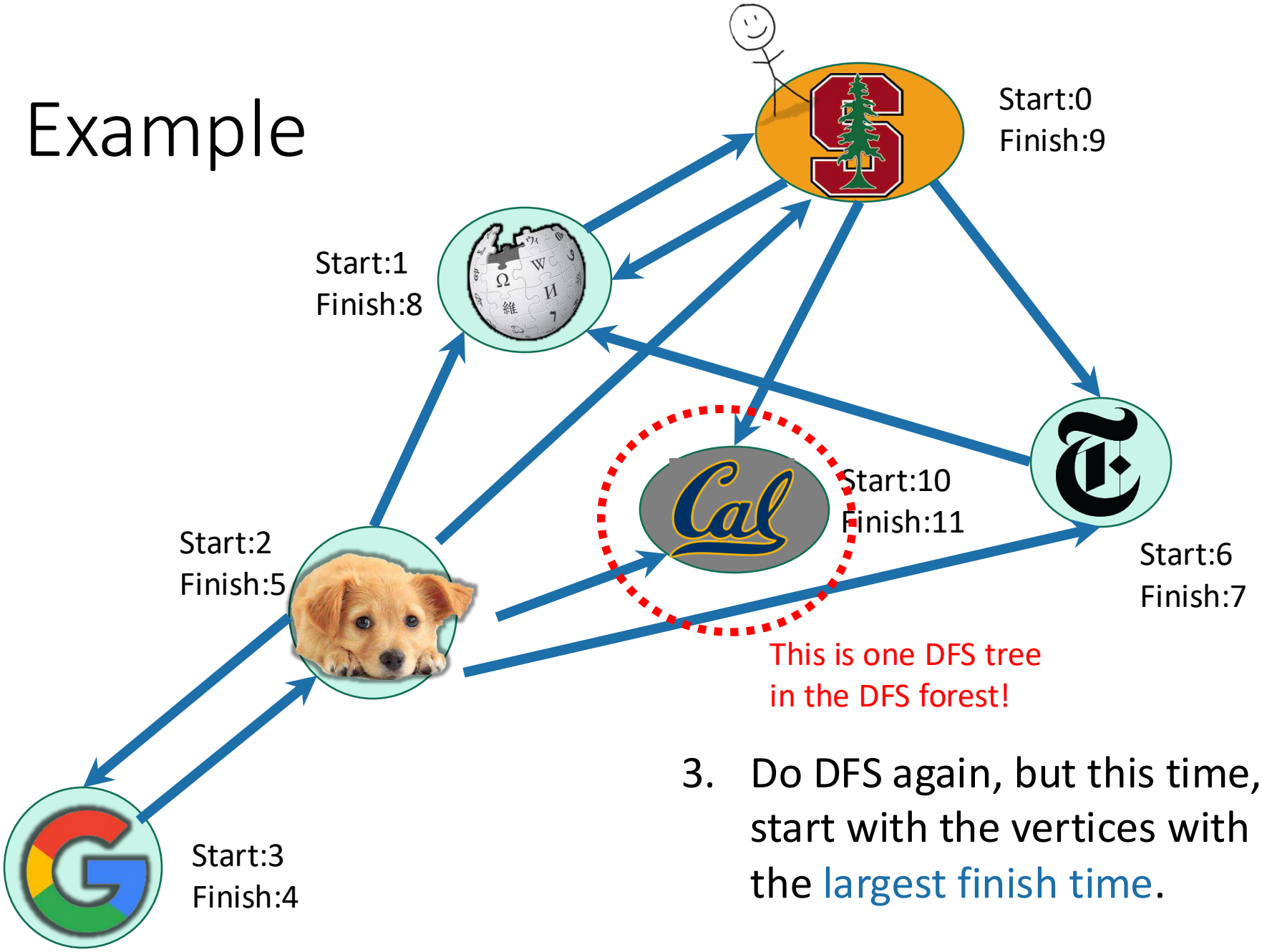
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



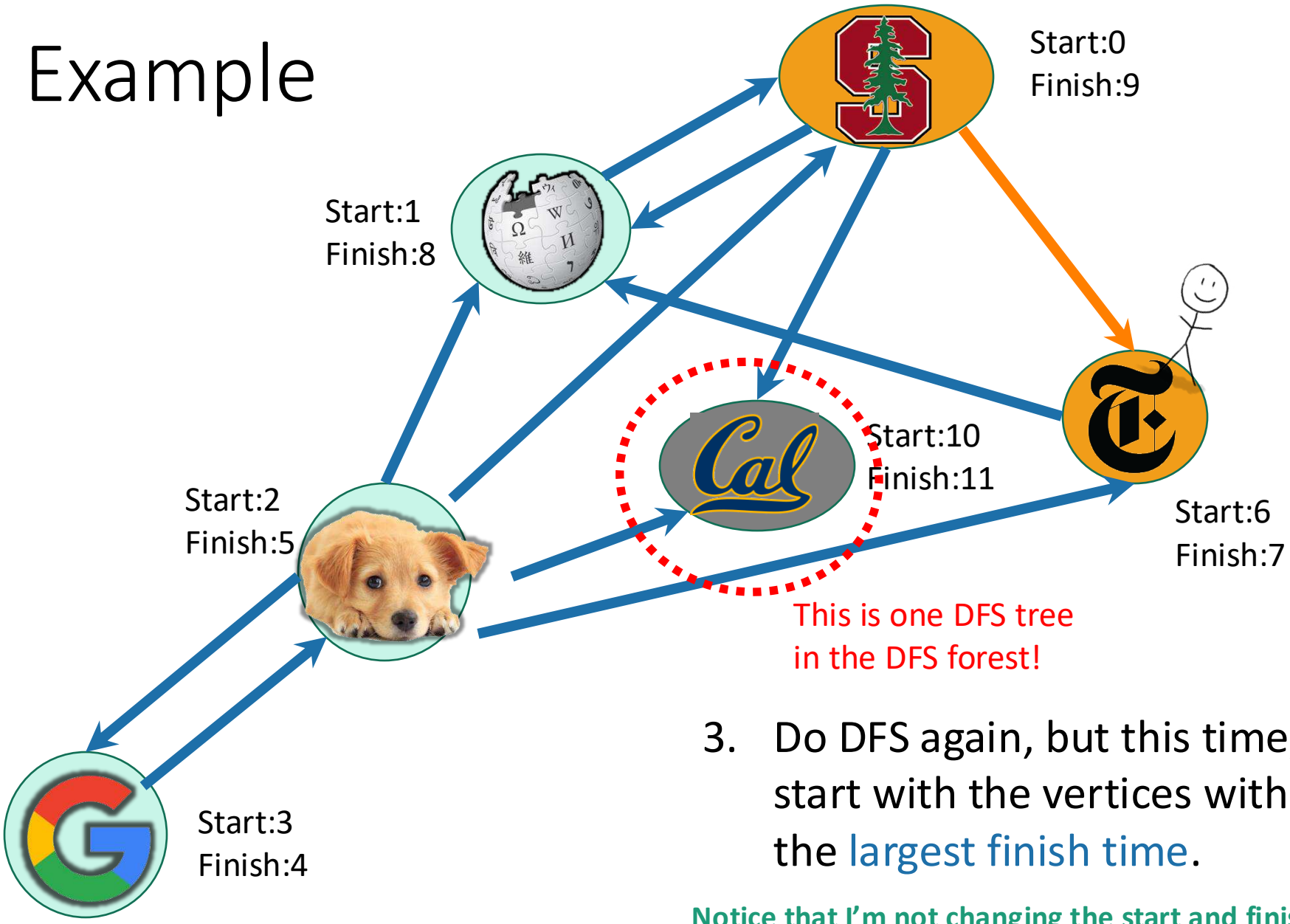
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



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

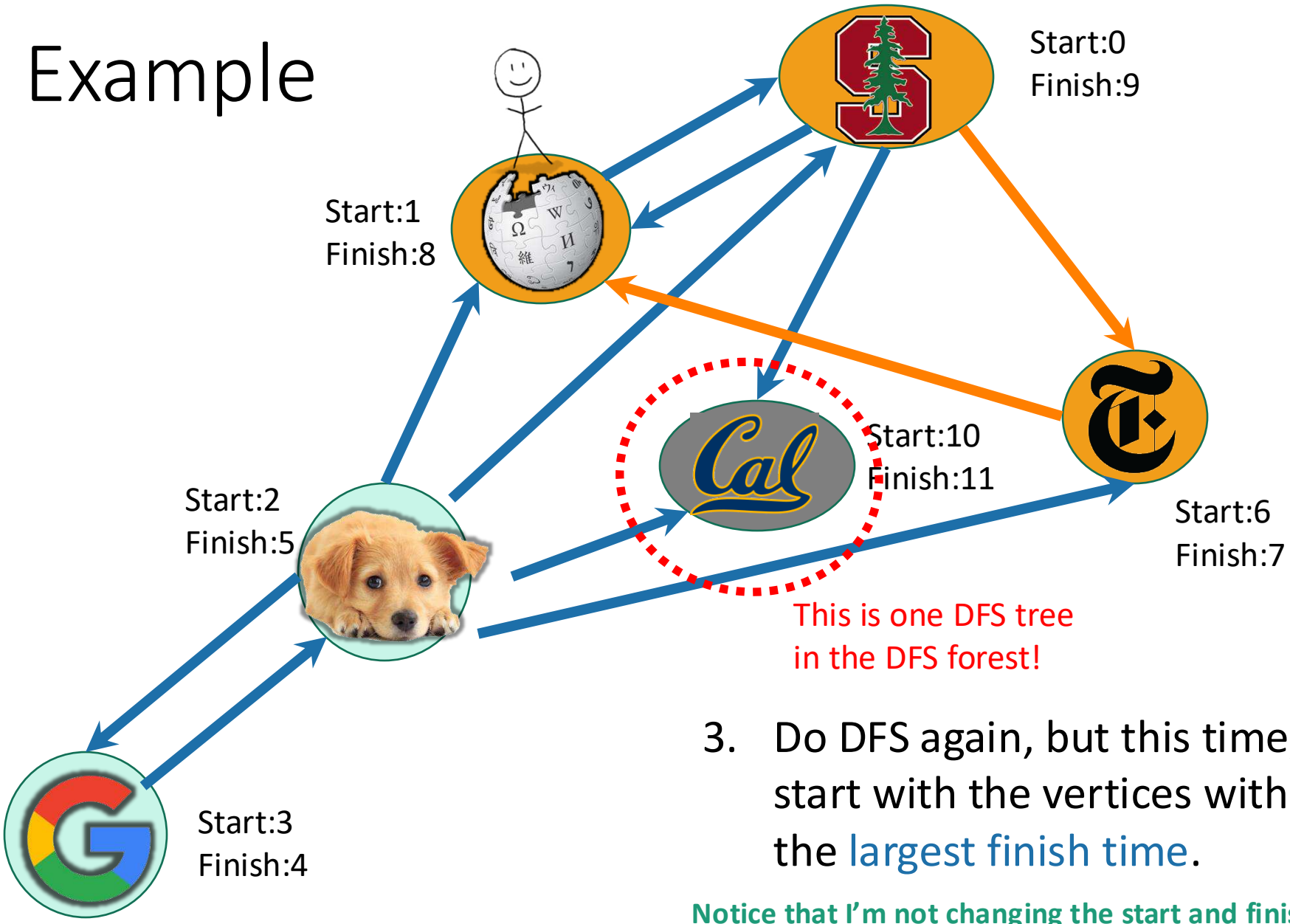
Example



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.

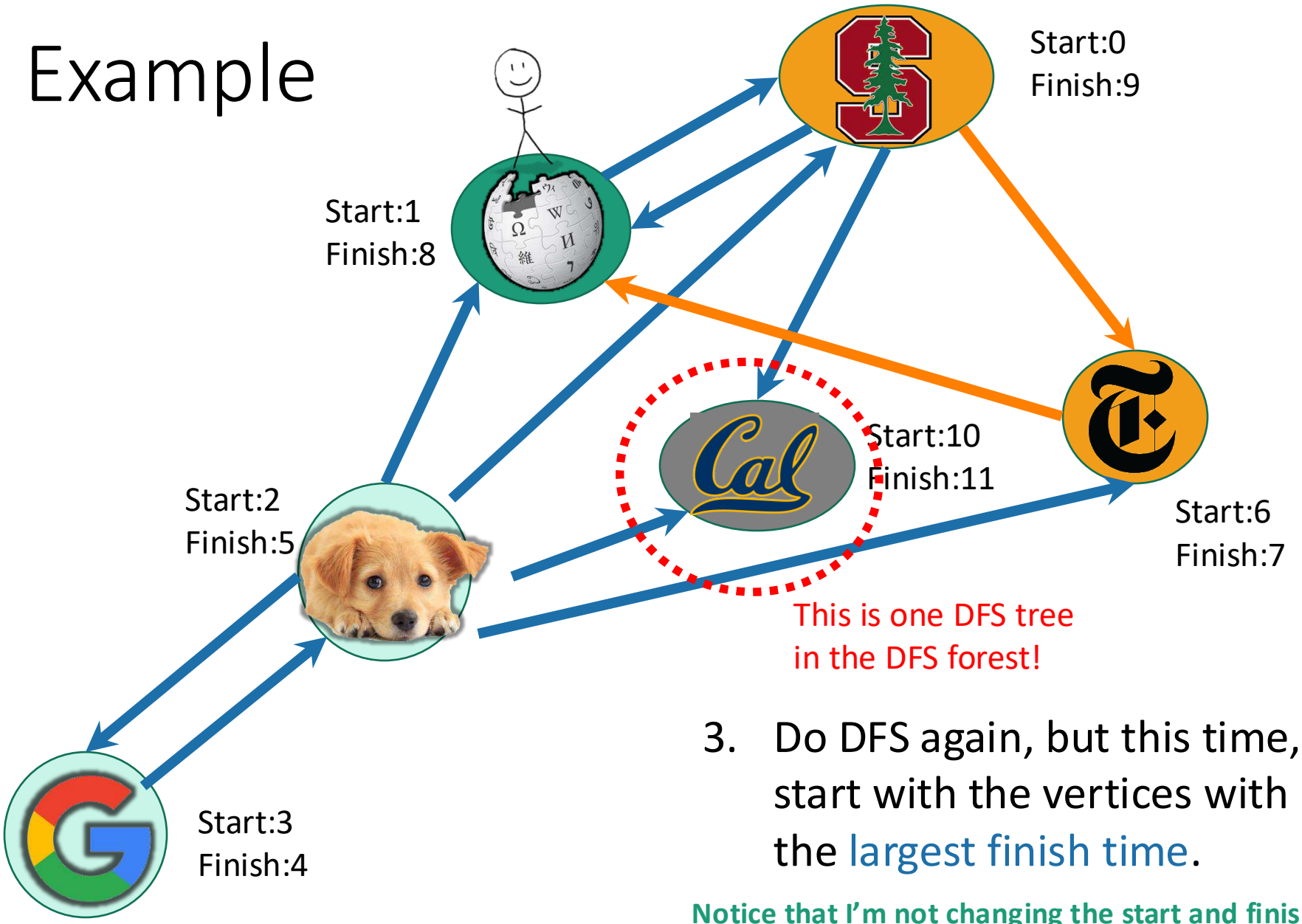
Example



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.

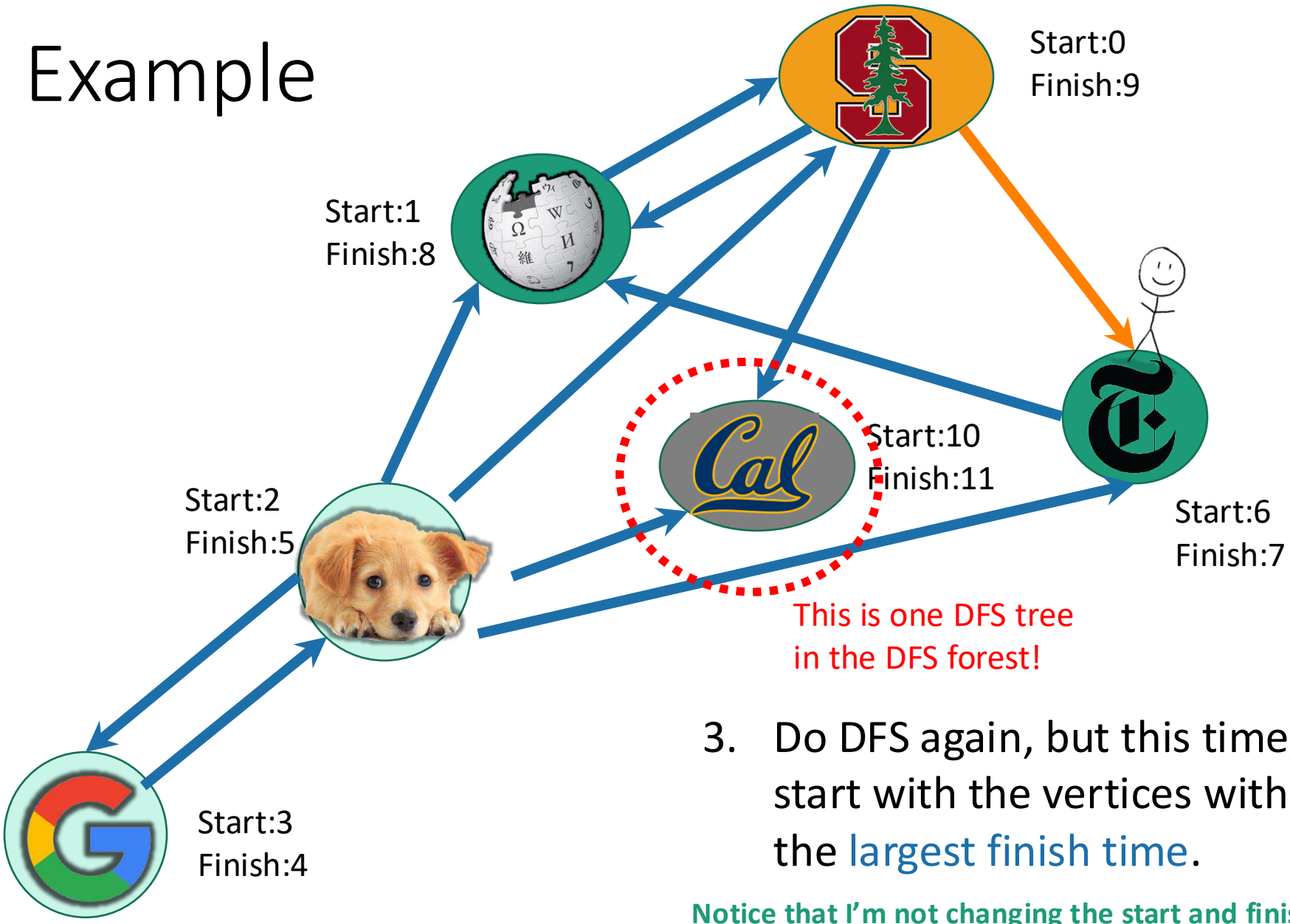
Example



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.

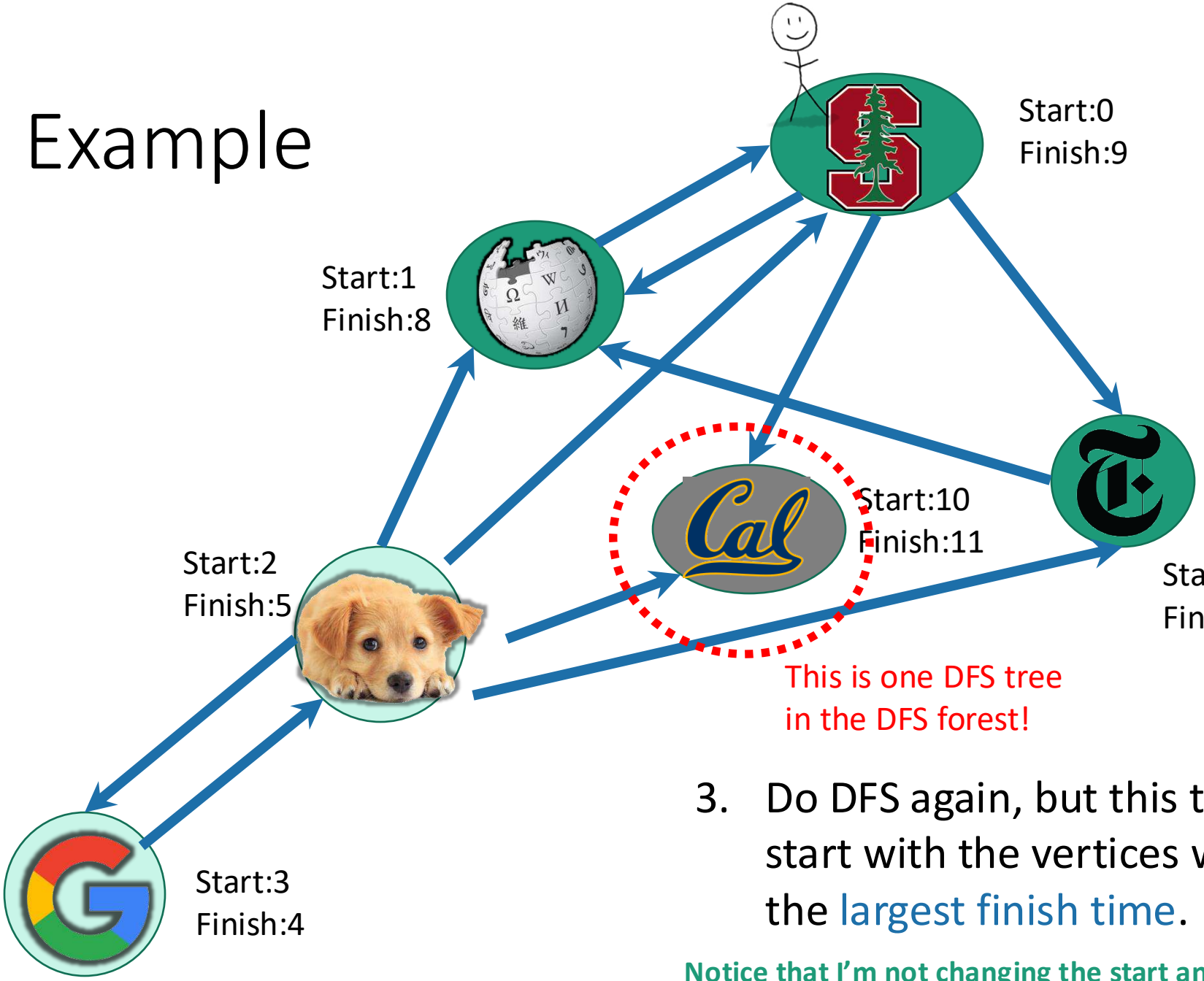
Example



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.

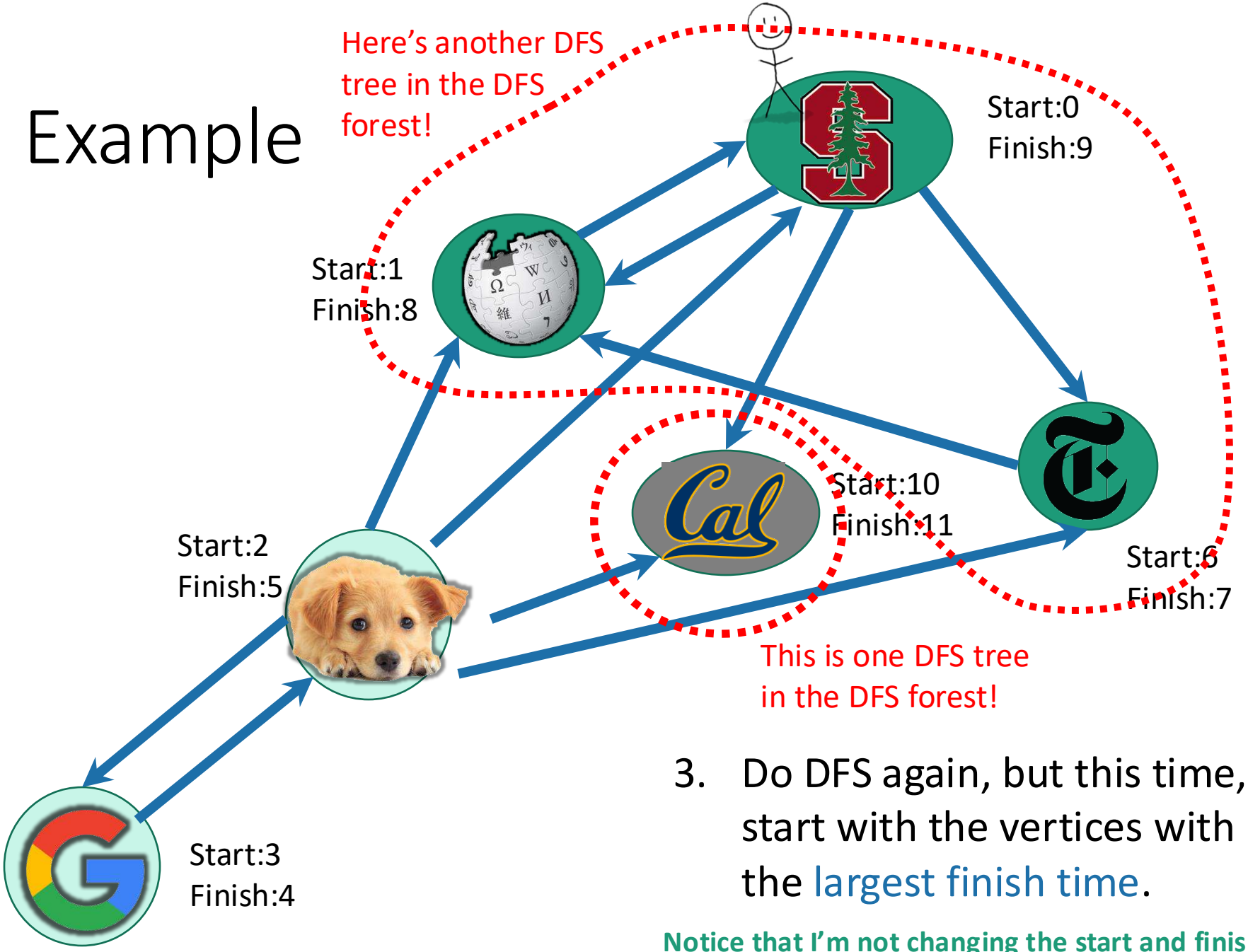
Example



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.

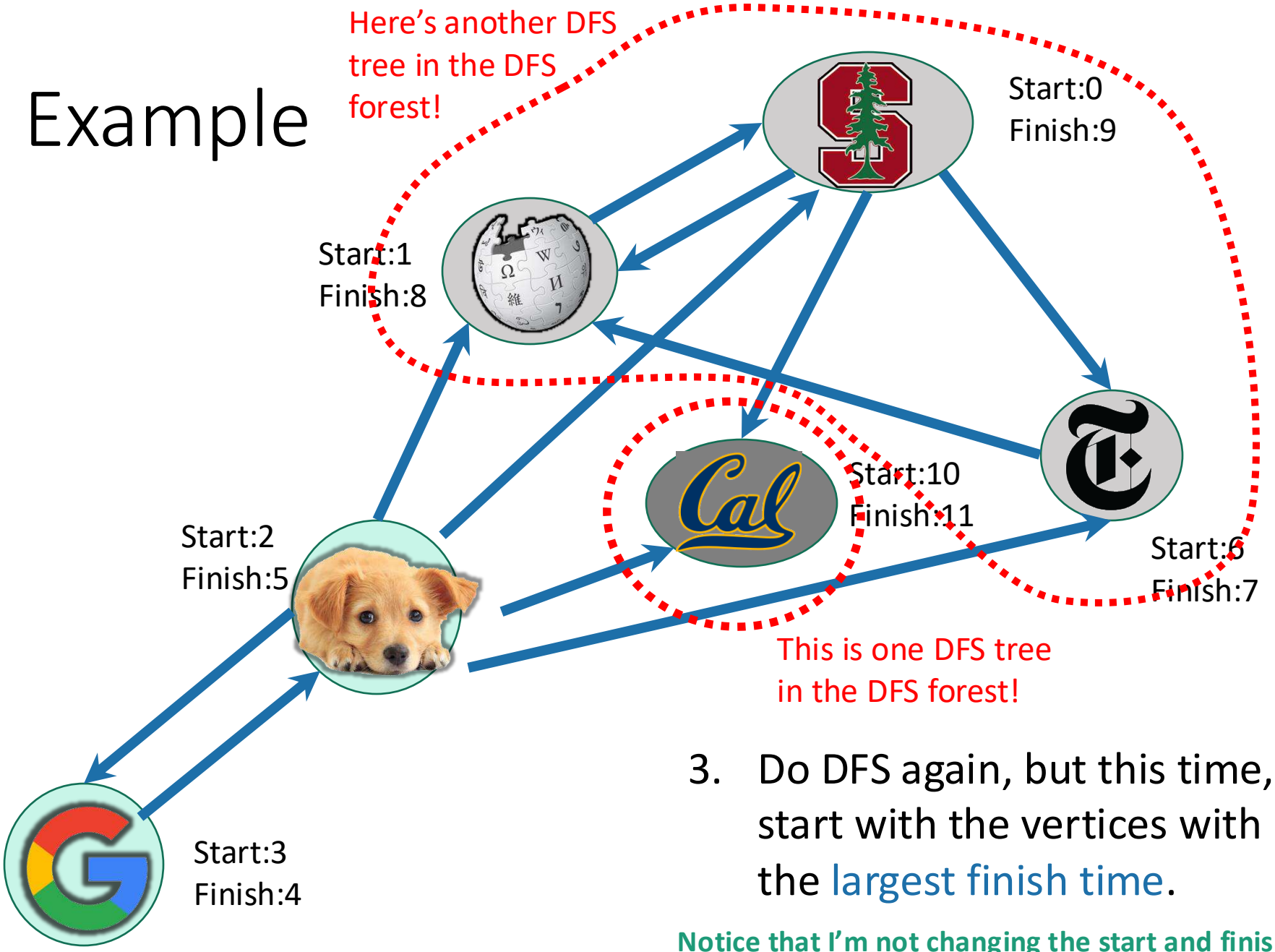
Example



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.

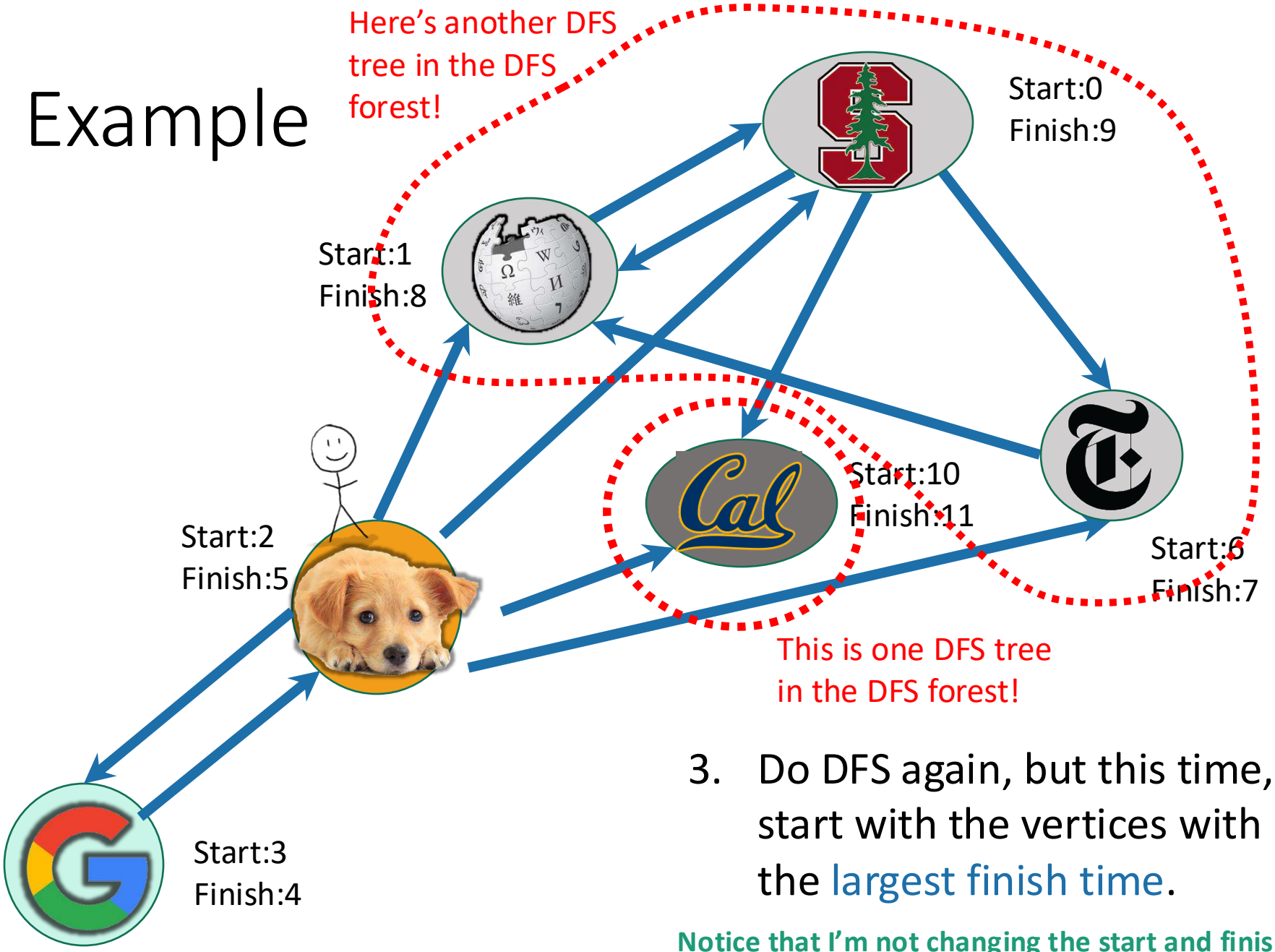
Example



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.

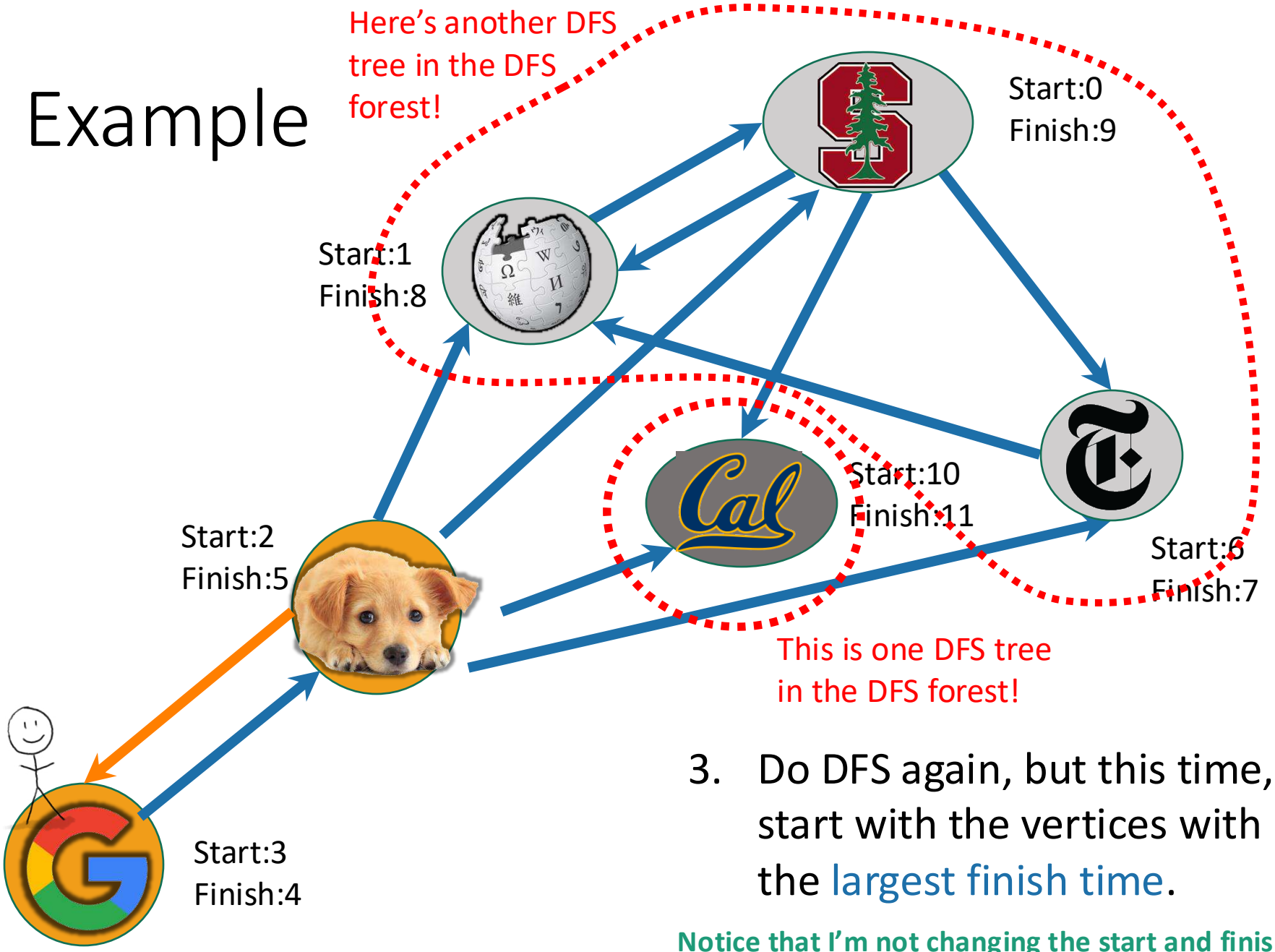
Example



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.

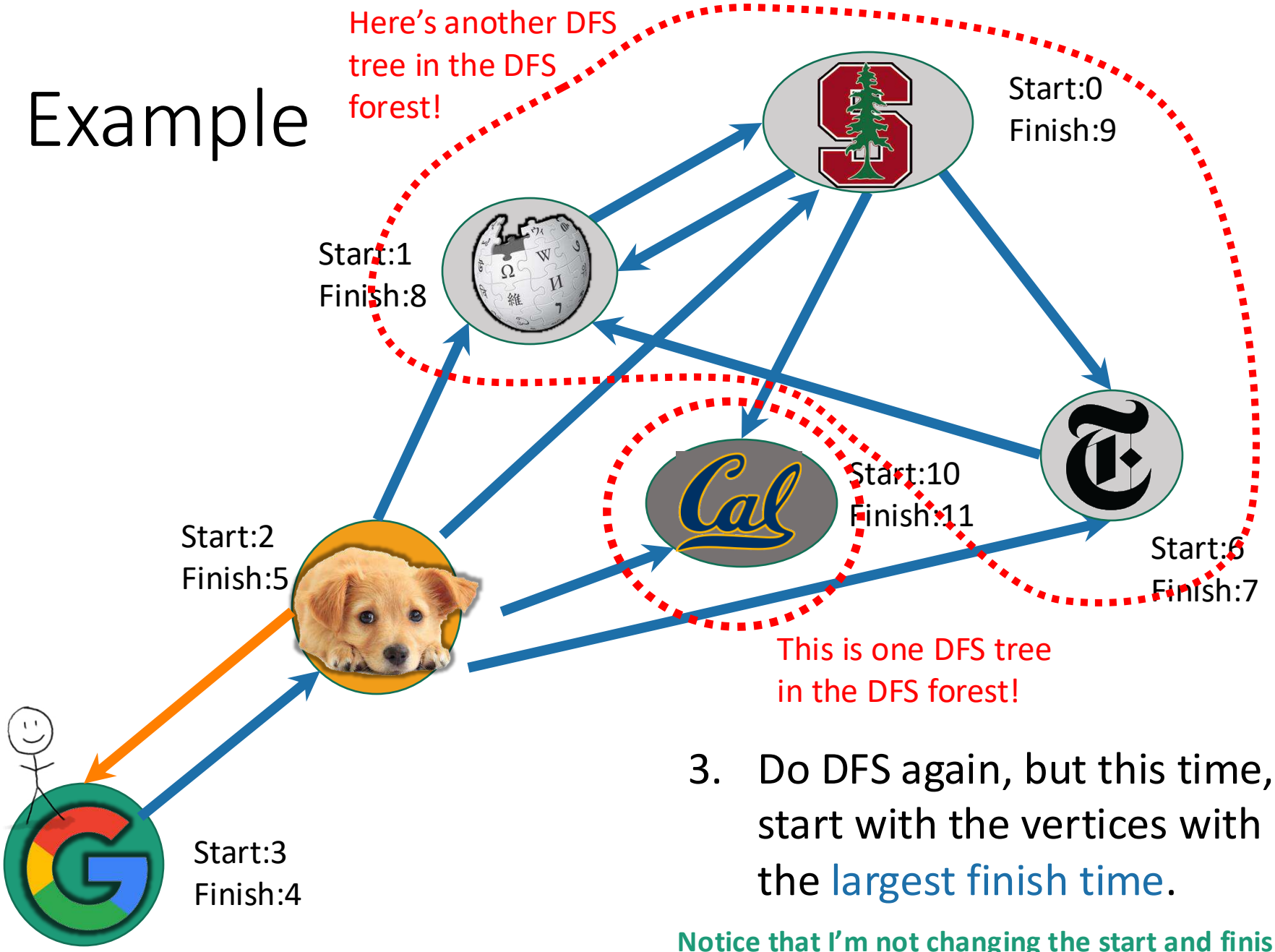
Example



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.

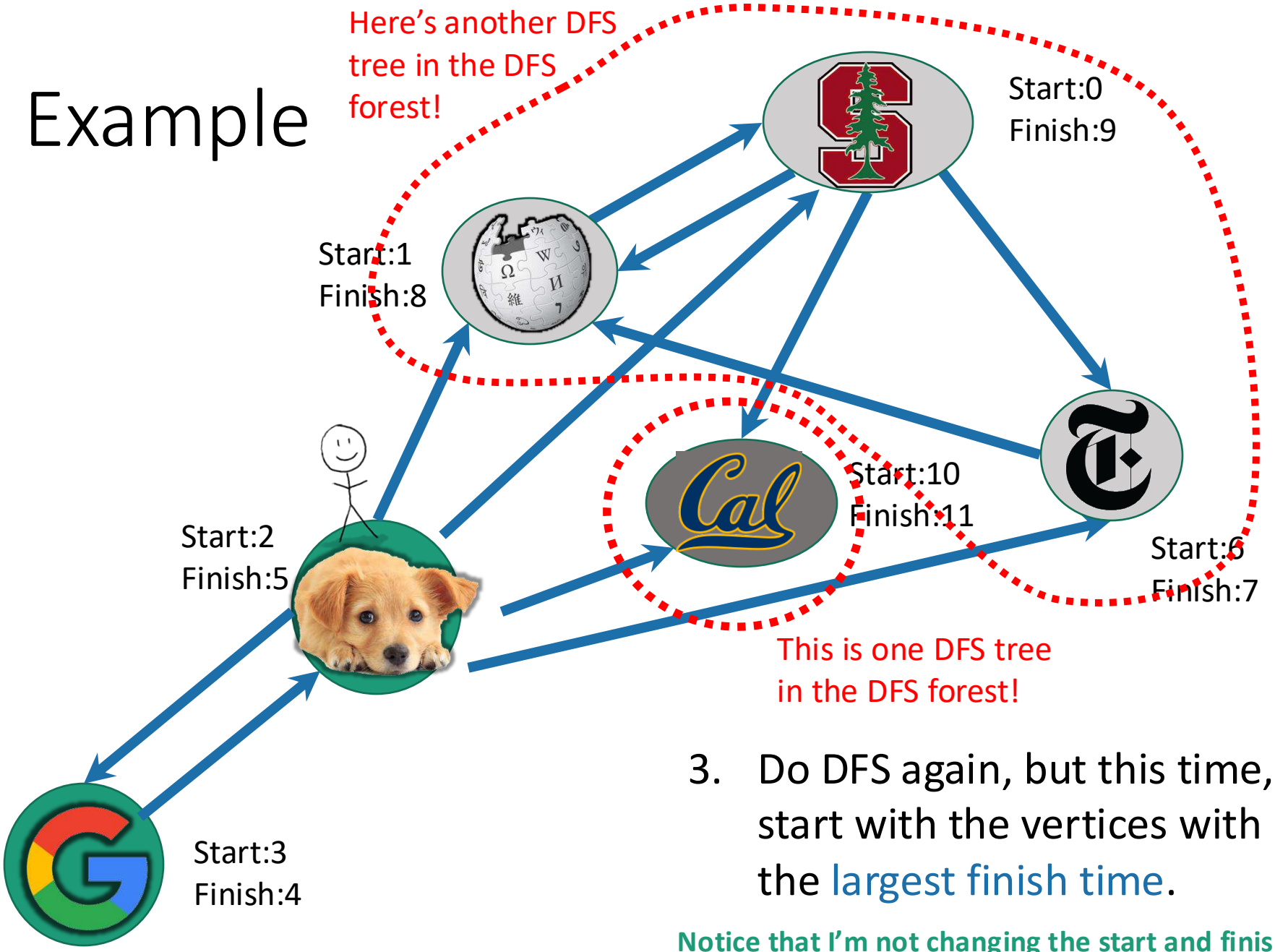
Example



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.

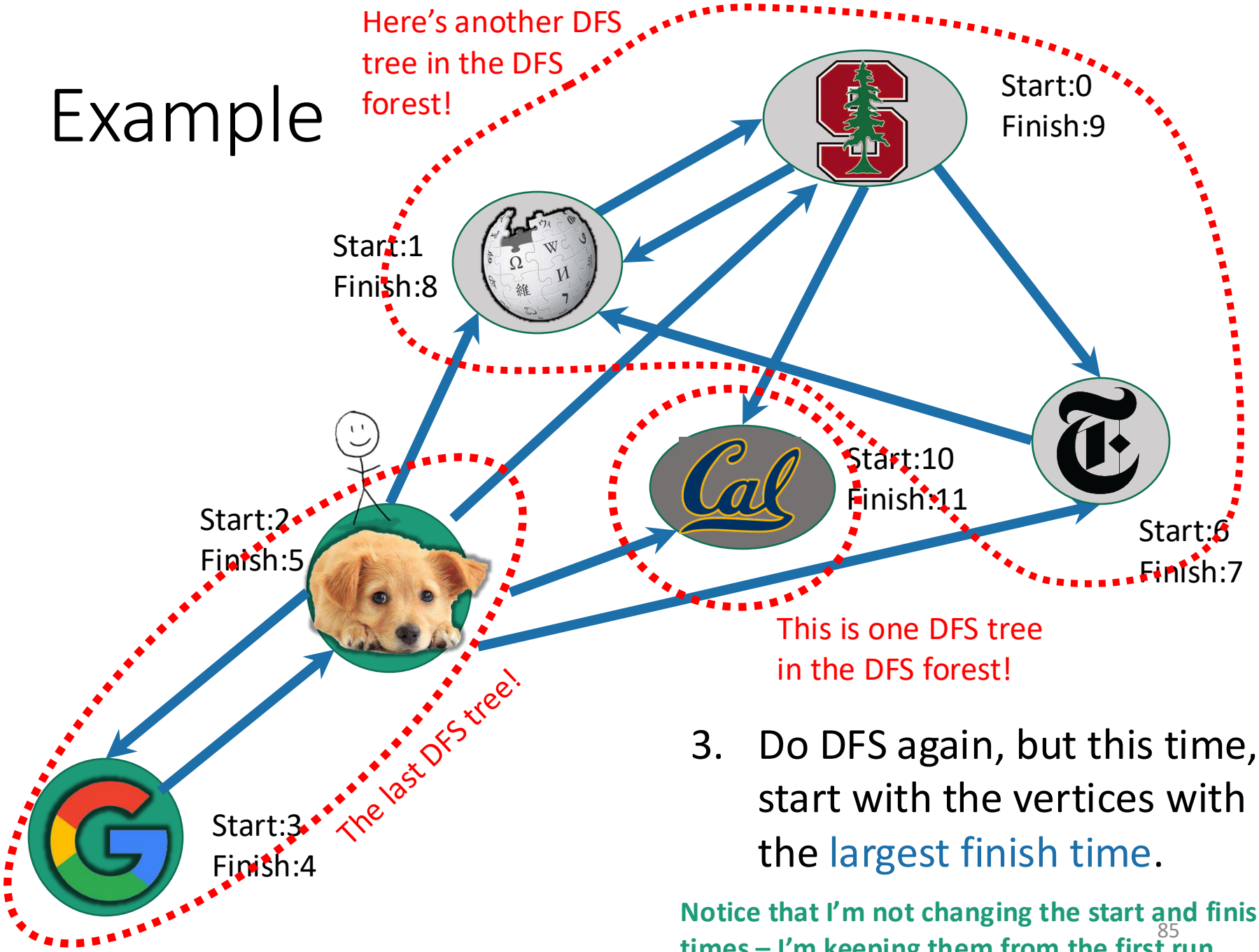
Example



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.

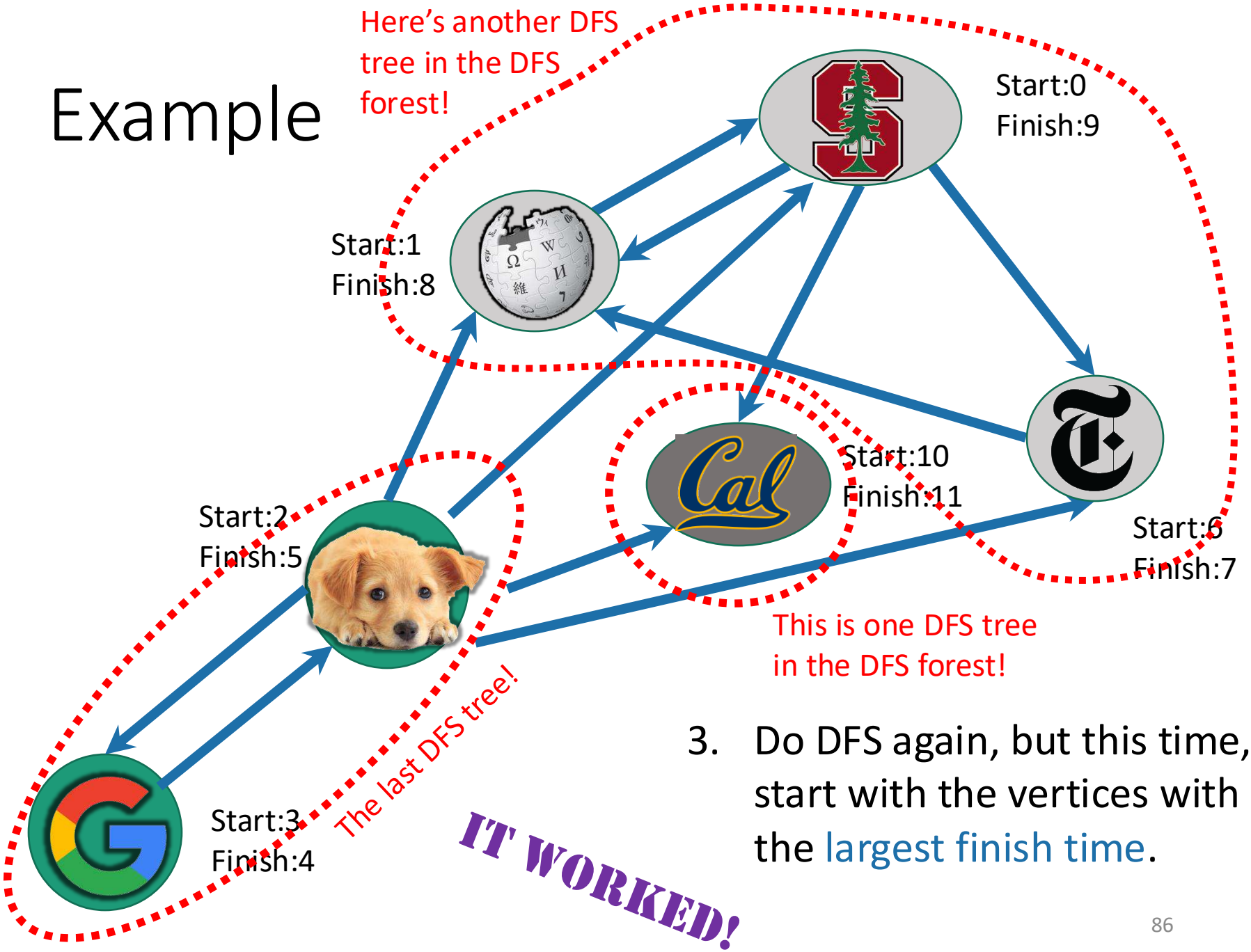
Example



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.

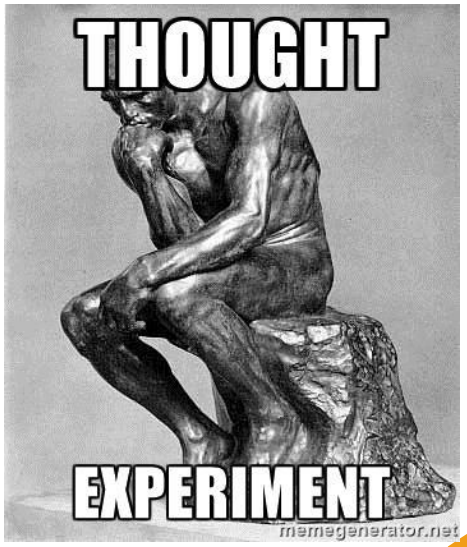
Example



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

One question

The SCC graph



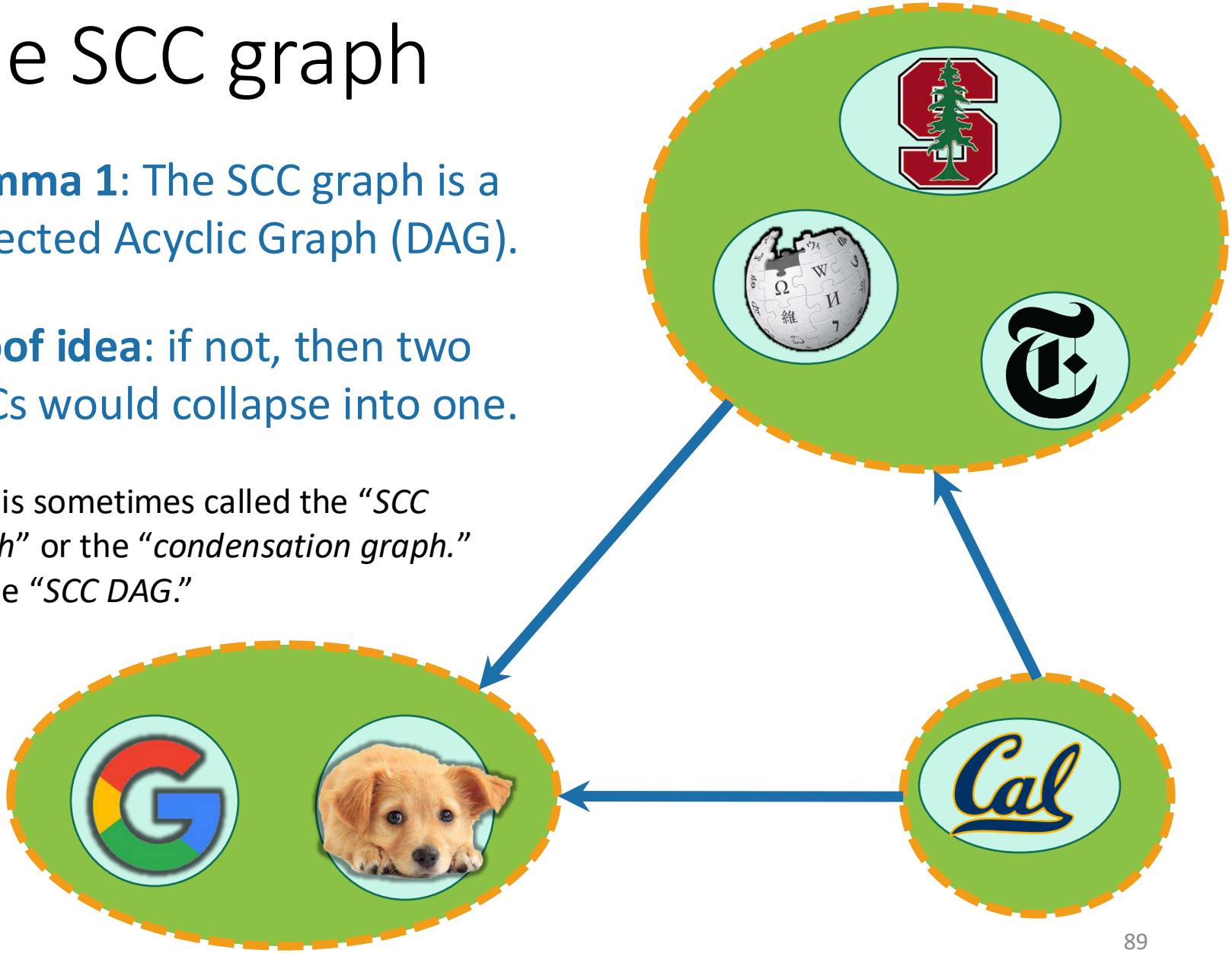
- Pretend that each SCC is a vertex in a new graph.

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.

This graph is sometimes called the "*SCC meta-graph*" or the "*condensation graph.*" I'll call it the "*SCC DAG*."

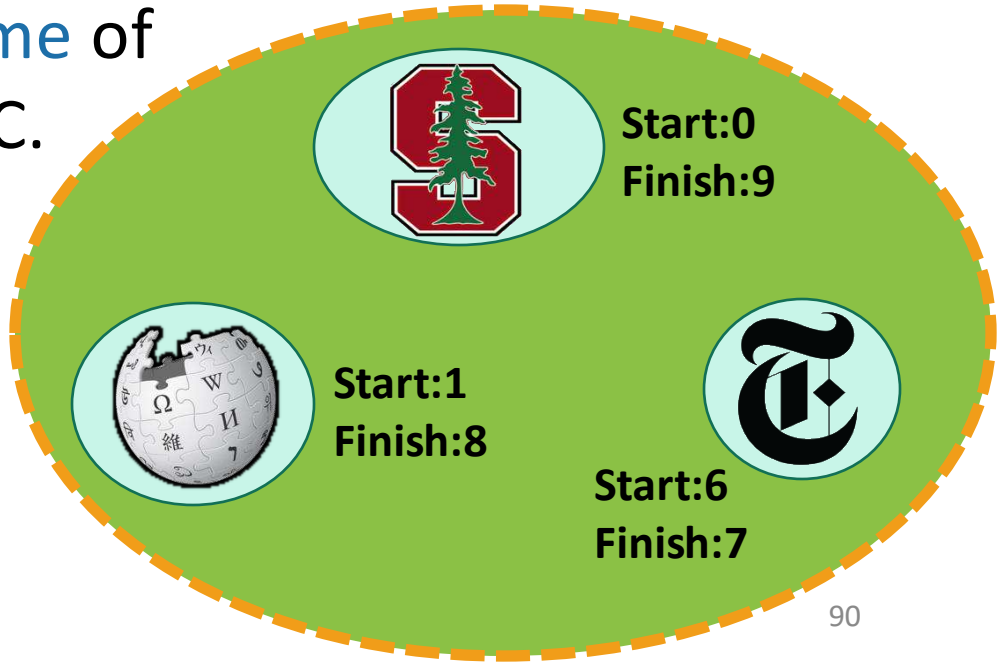


all times are with respect to the **first** DFS run

Starting and finishing times in a SCC

Definitions:

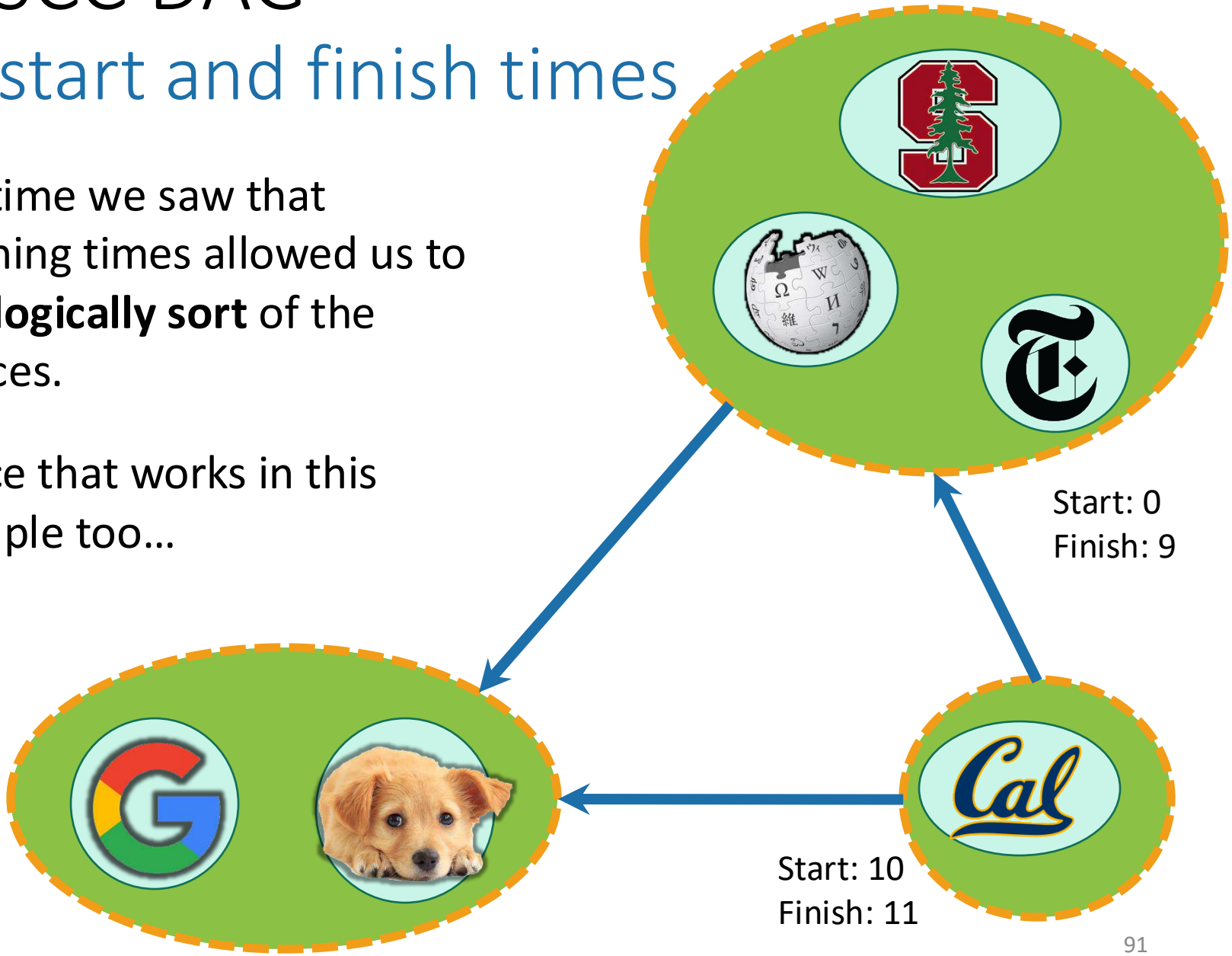
- The **finishing time** of a SCC is the largest finishing time of any element of that SCC.
- The **starting time** of a SCC is the smallest starting time of any element of that SCC.



Start: 0
Finish: 9

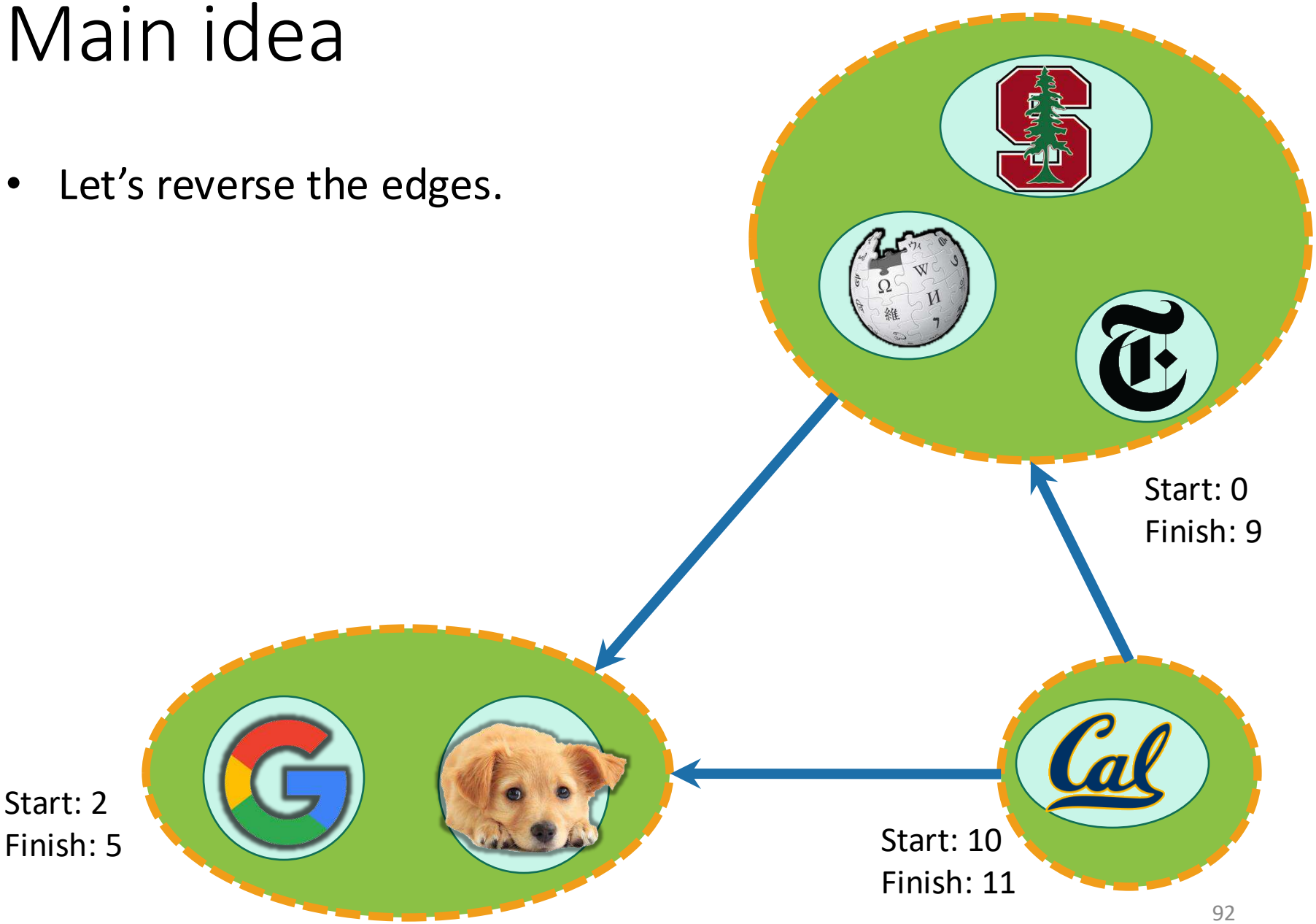
Our SCC DAG
with start and finish times

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



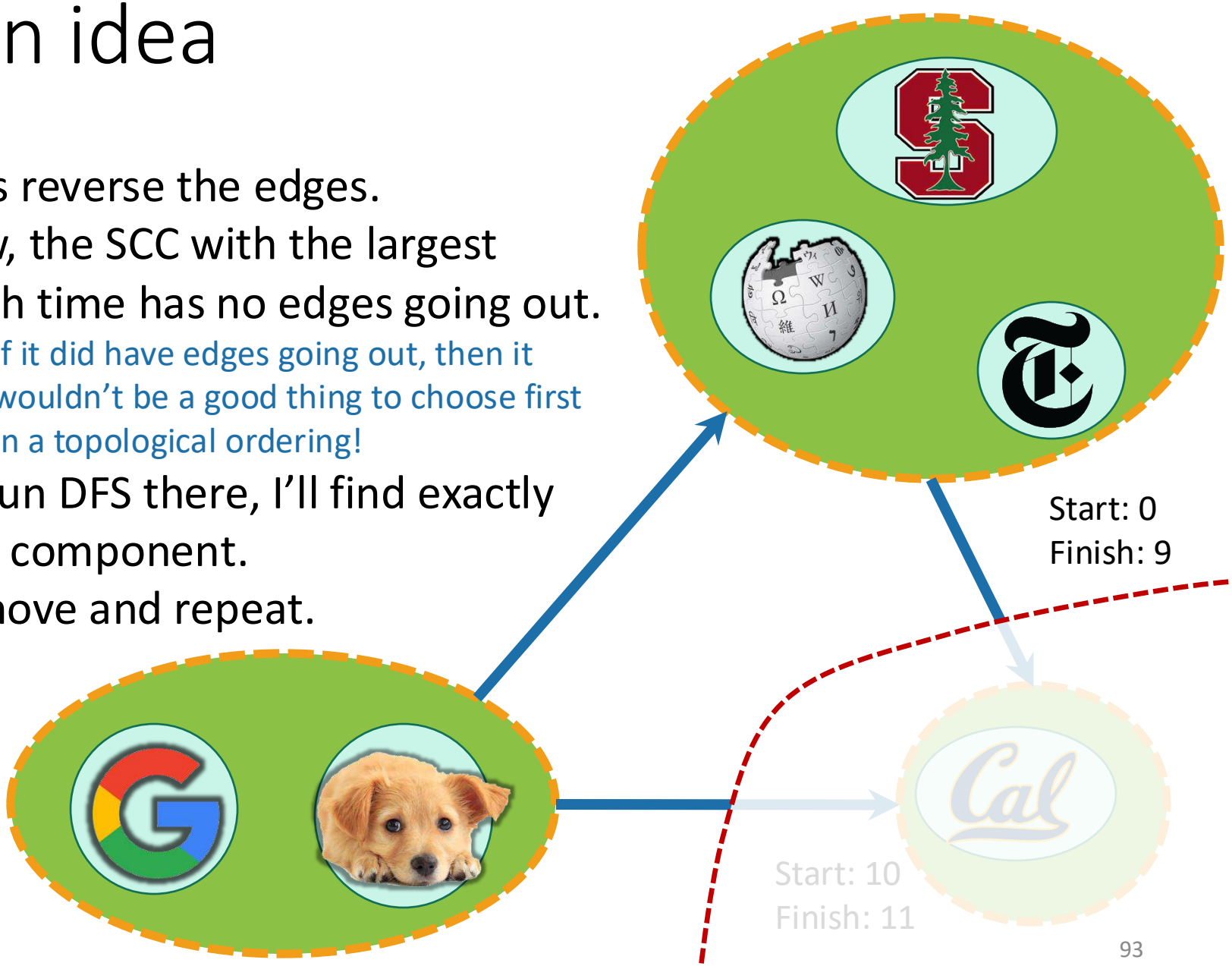
Main idea

- 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.

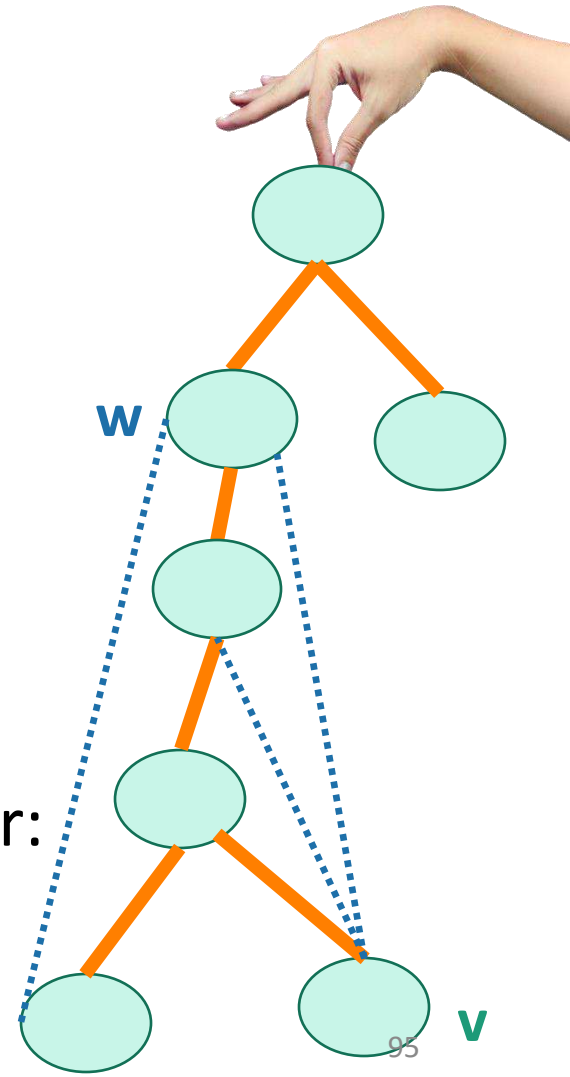
Recall

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

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

- If neither are descendents of each other:

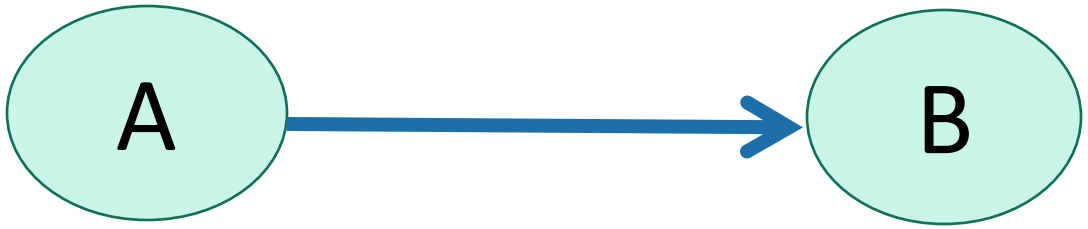
(or the other way around)



* The version of DFS where we keep restarting until we explore the whole graph

As we saw last time…

Claim: If we run DFS on a DAG:

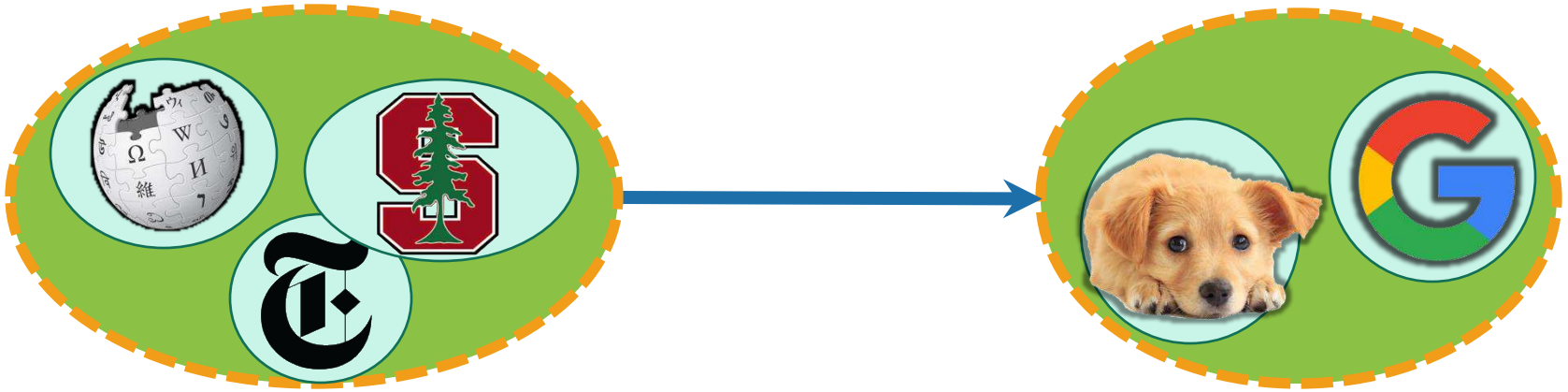


finish: [larger] finish: [smaller]

That is, if we run DFS* on a DAG and if there is an edge from A to B, A.finish > B.finish.

Same thing, in the SCC DAG.

- Claim: we'll always have

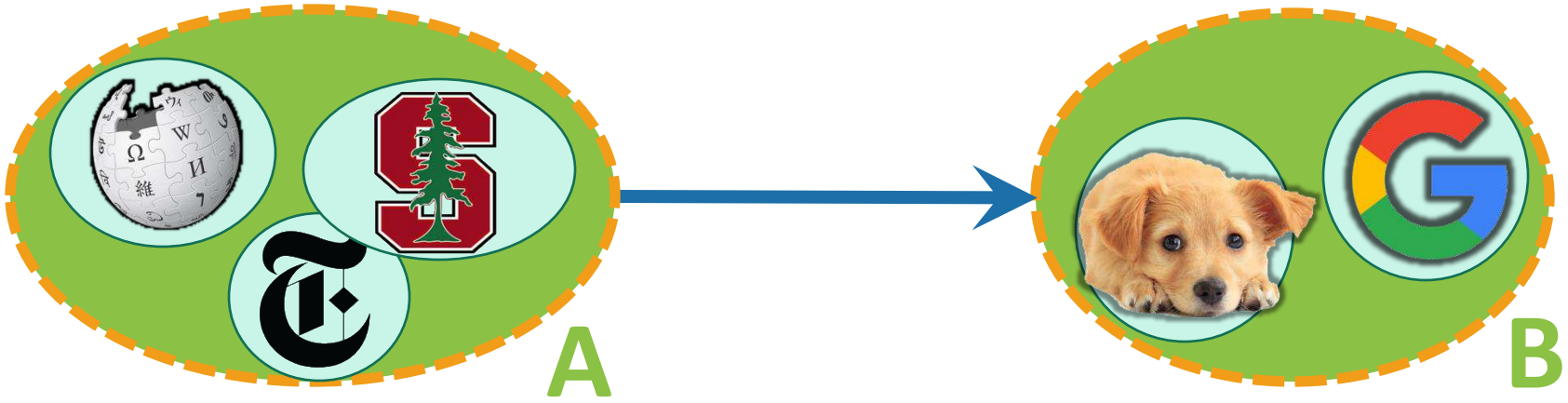


finish: [larger]

finish: [smaller]

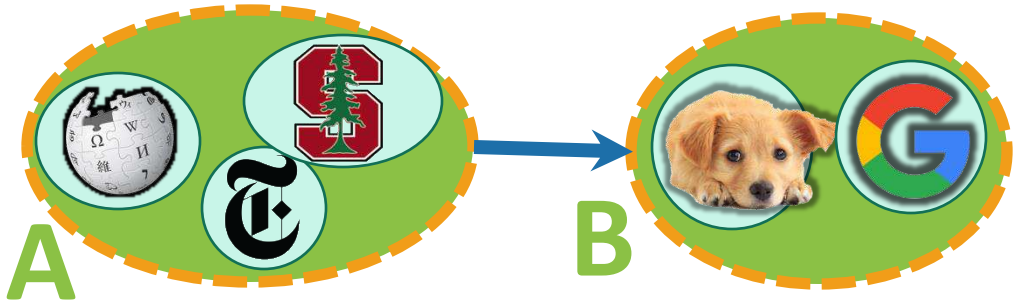
Let's call it Lemma 2

- If there is an edge like this:



- Then A.finish > B.finish.

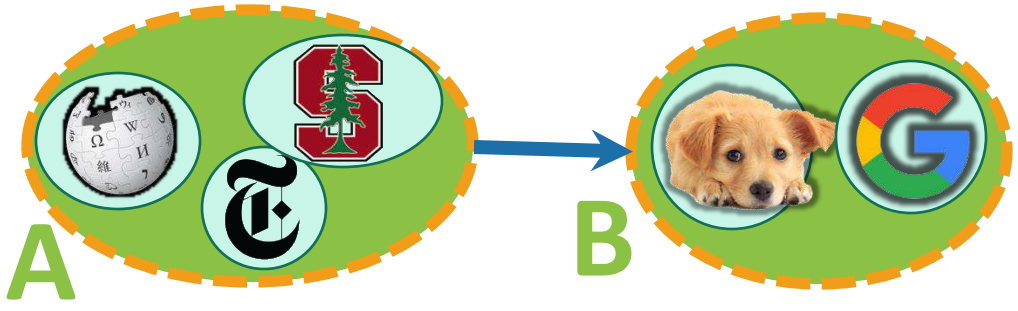
Proof idea



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.

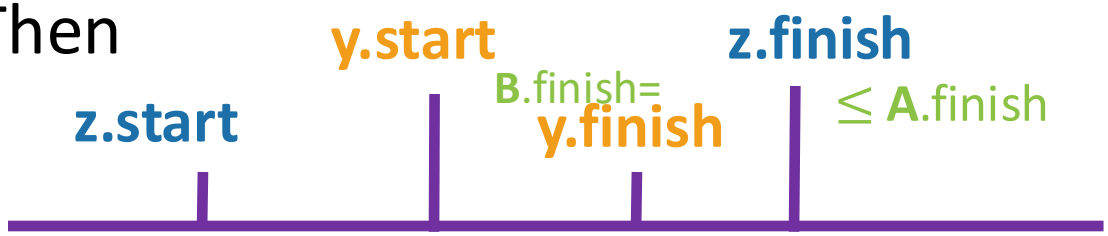
Proof idea



Want to show A.finish > B.finish.

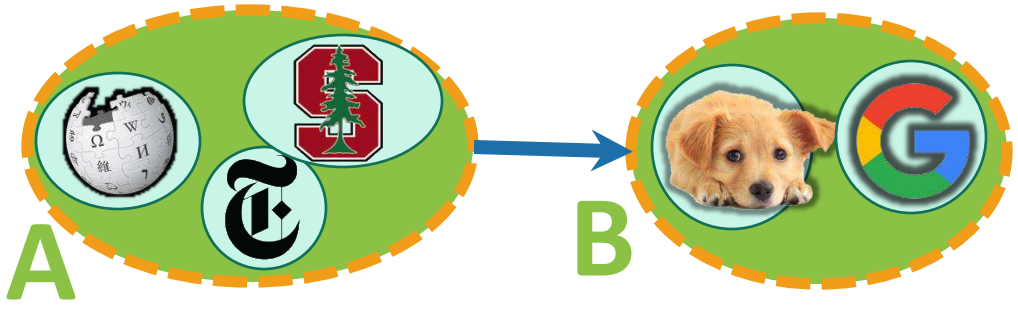
- **Case 1**: We reached **A** before **B** in our first DFS.
- Say that:
  - **y** has the largest finish in **B**; B.finish = y.finish
  - **z** was discovered first in **A**; A.finish >= z.finish
- Then:
  - Reach **A** before **B**
  - => we will discover **y** via **z**
  - => **y** is a descendant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

- Then



aka,
A.finish > B.finish

Proof idea



Want to show A.finish > B.finish.

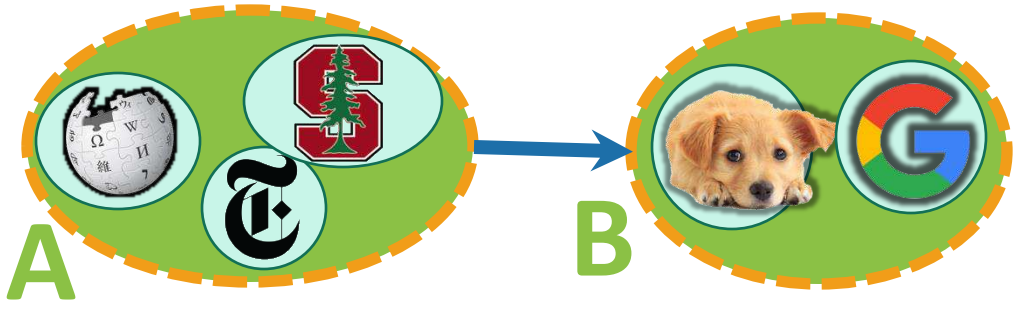
- **Case 2**: We reached **B** before **A** in our first DFS.

- 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:

A.finish > B.finish

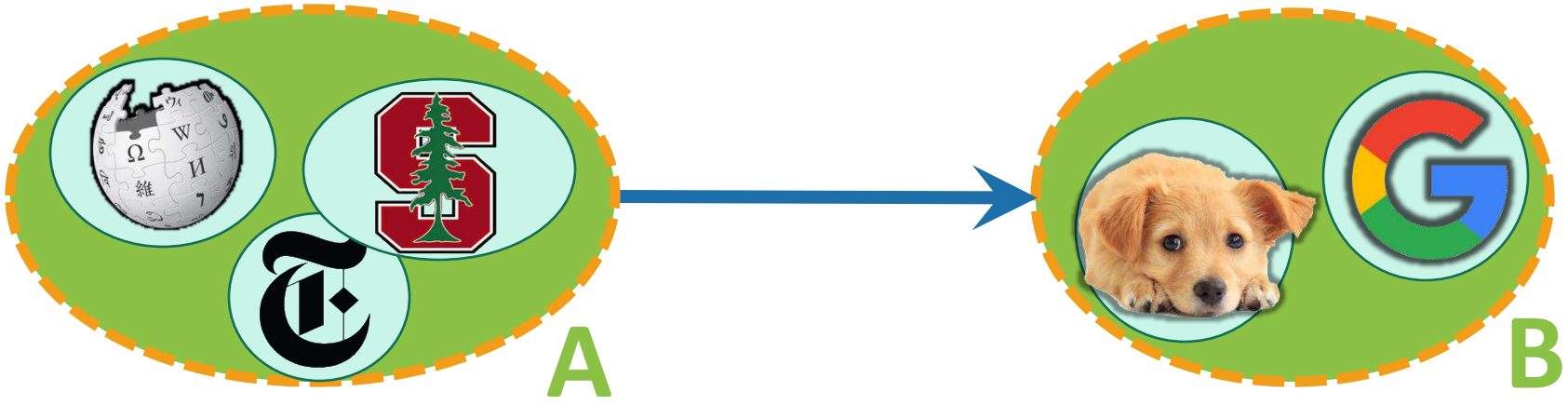
which is what we wanted to show.

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:

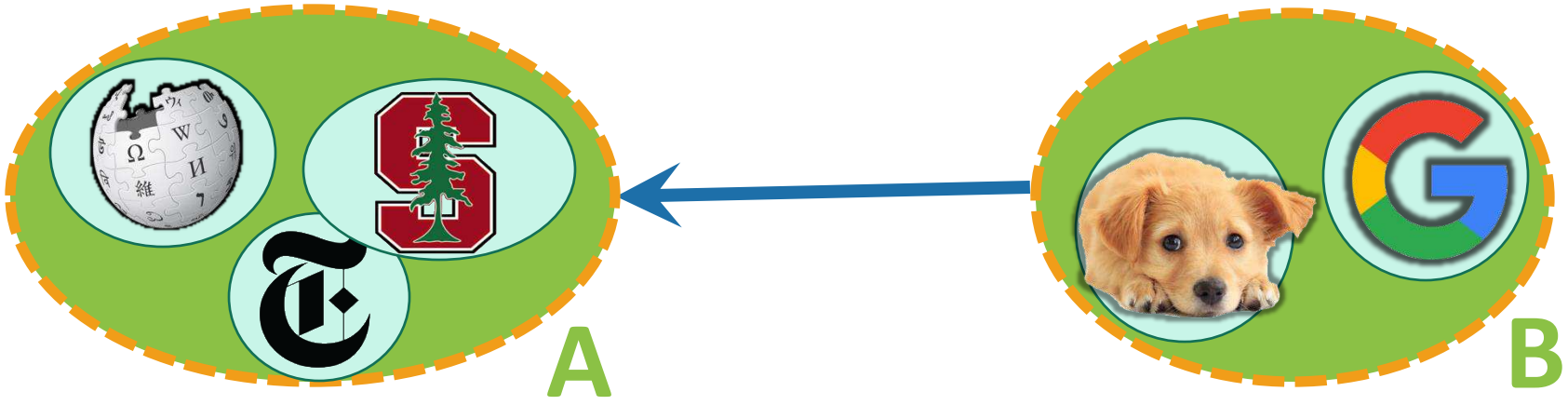


- Then A.finish > B.finish.

This establishes:

Corollary 1

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

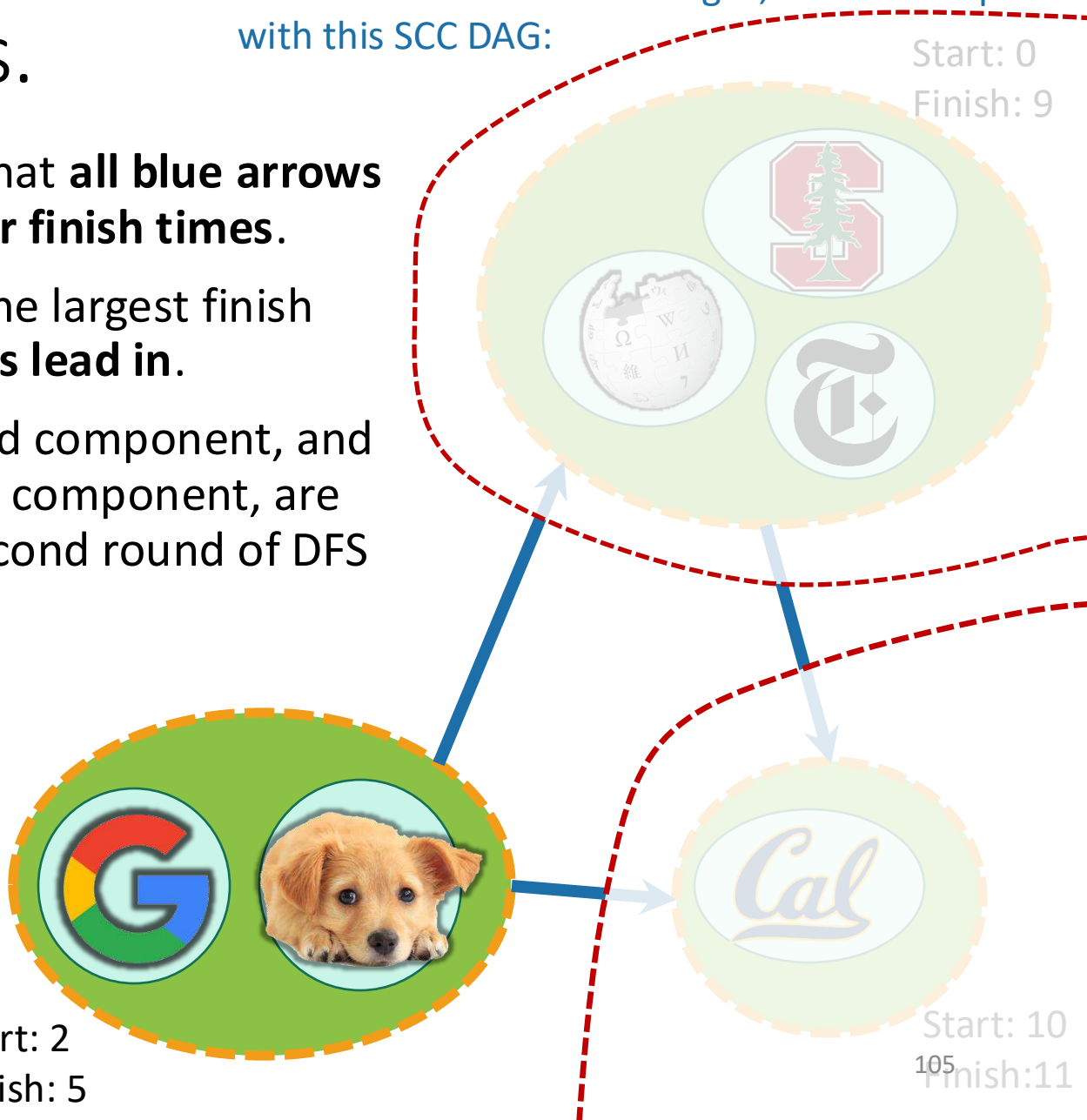


- Then A.finish > B.finish.

Now we see why this finds SCCs.

- 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

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



- 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!

Formally, we prove it by induction

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

- **Inductive hypothesis**:
  - The first t DFS trees found in the second (reversed) DFS forest are the t SCCs with the largest finish times.
- **Base case**: (t=0)
  - The first 0 DFS trees found in the second (reversed) DFS forest are the 0 SCCs with the largest finish times**.** **(TRUE)**
- **Inductive step:**
  - Fun exercise! [See skipped slide; we already did the intuition]
- **Conclusion:**
  - The IH holds when t = #SCCs, aka the second (reversed) DFS forest contains all the SCCs as its trees!

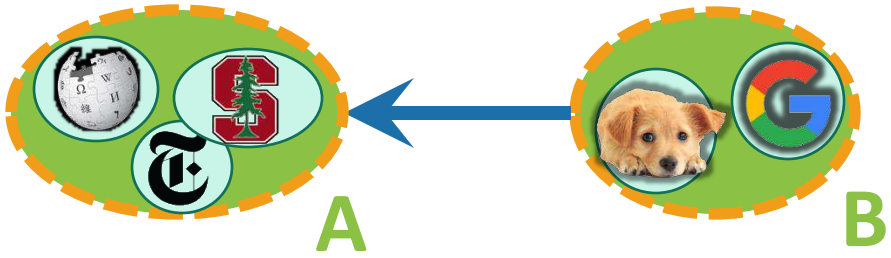
This slide skipped in class

Inductive step

- **Assume by induction that the first t trees are the last-finishing SCCs.**
- 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 (reversed) 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.**

Recap of proof idea

- The finish times of the first DFS basically do topological sort on the SCC DAG.
- Thus, when we reverse all the edges we have:
  - If there is an edge like this in the reversed graph:



  - Then A.finish > B.finish.

- Then when we do DFS again in reverse order of finish time, we pick off the 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 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 a non-obvious algorithm!
    - (at least, it was non-obvious 80 minutes ago)

Next time

- Dijkstra's algorithm!

BEFORE Next time

- Pre-lecture exercise: weighted graphs!