



Đồ thị: Tìm miền
liên thông mạnh



Nội dung

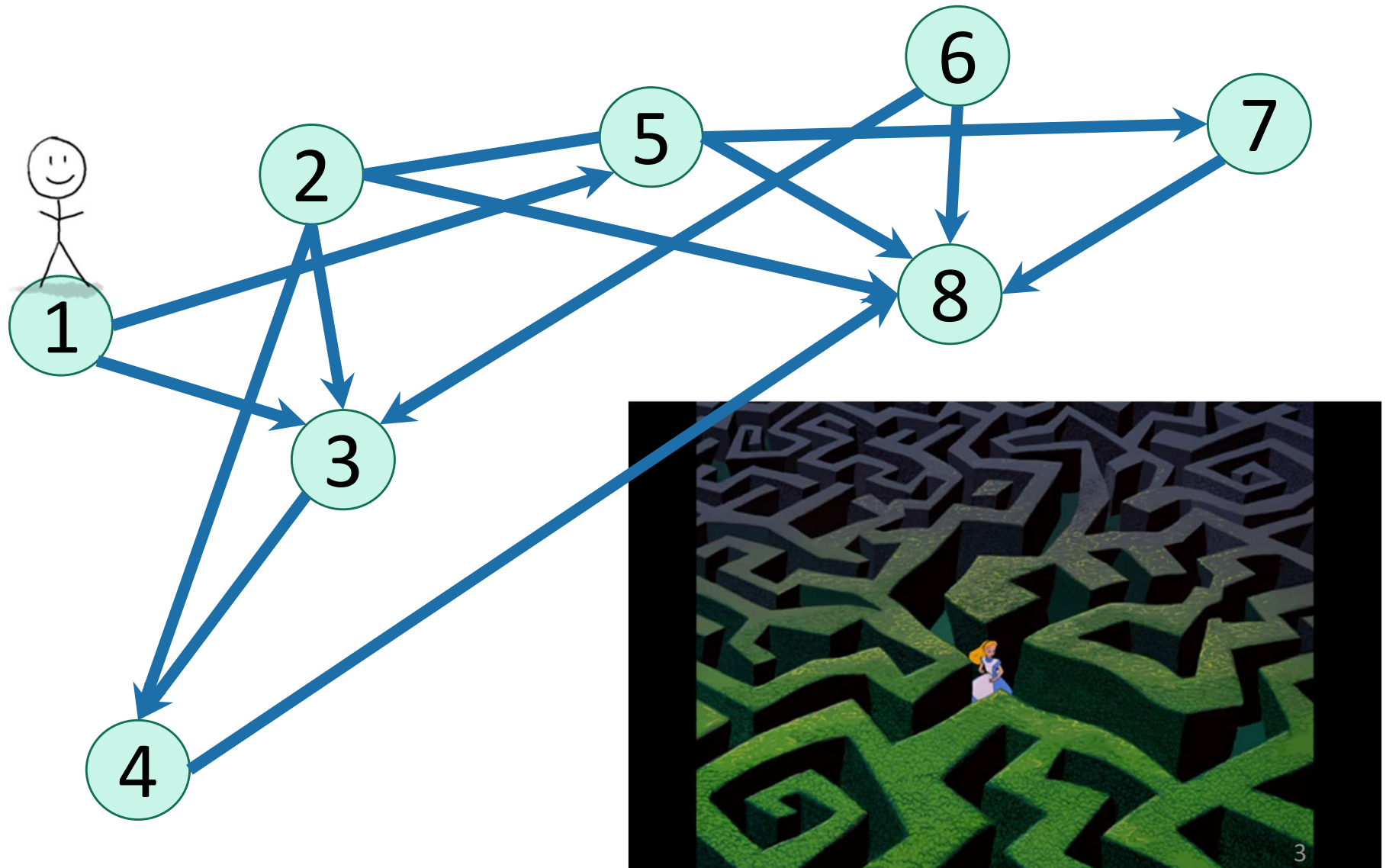
- Duyệt đồ thị có hướng
 - DFS tree
 - DFS forest
- Tìm miền liên thông mạnh (Finding strongly connected components)

Sử dụng một phần tài liệu bài giảng CS161 Stanford University

Recall: DFS

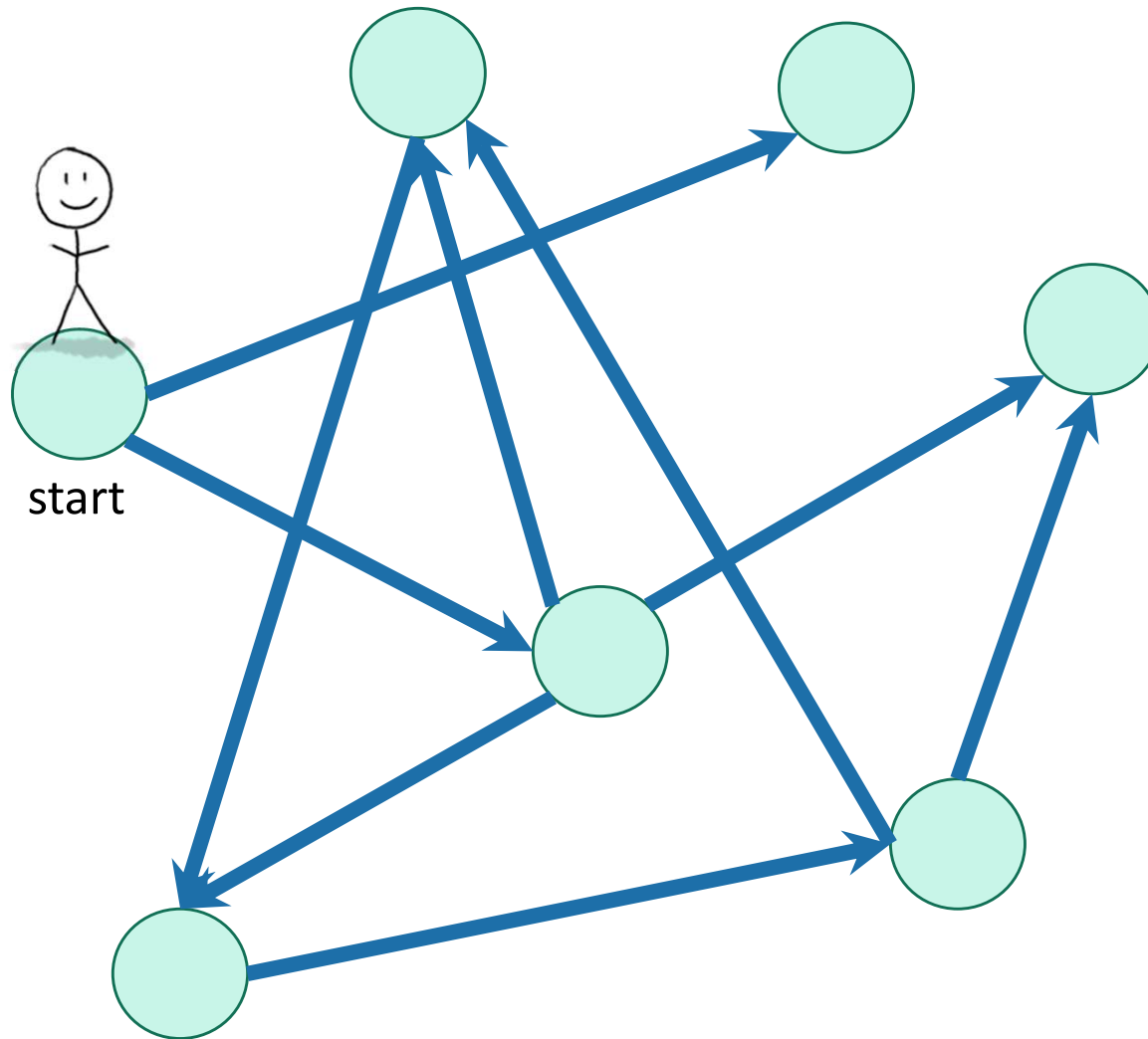
It's how you'd explore a labyrinth with chalk and a piece of string.




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



Depth First Search

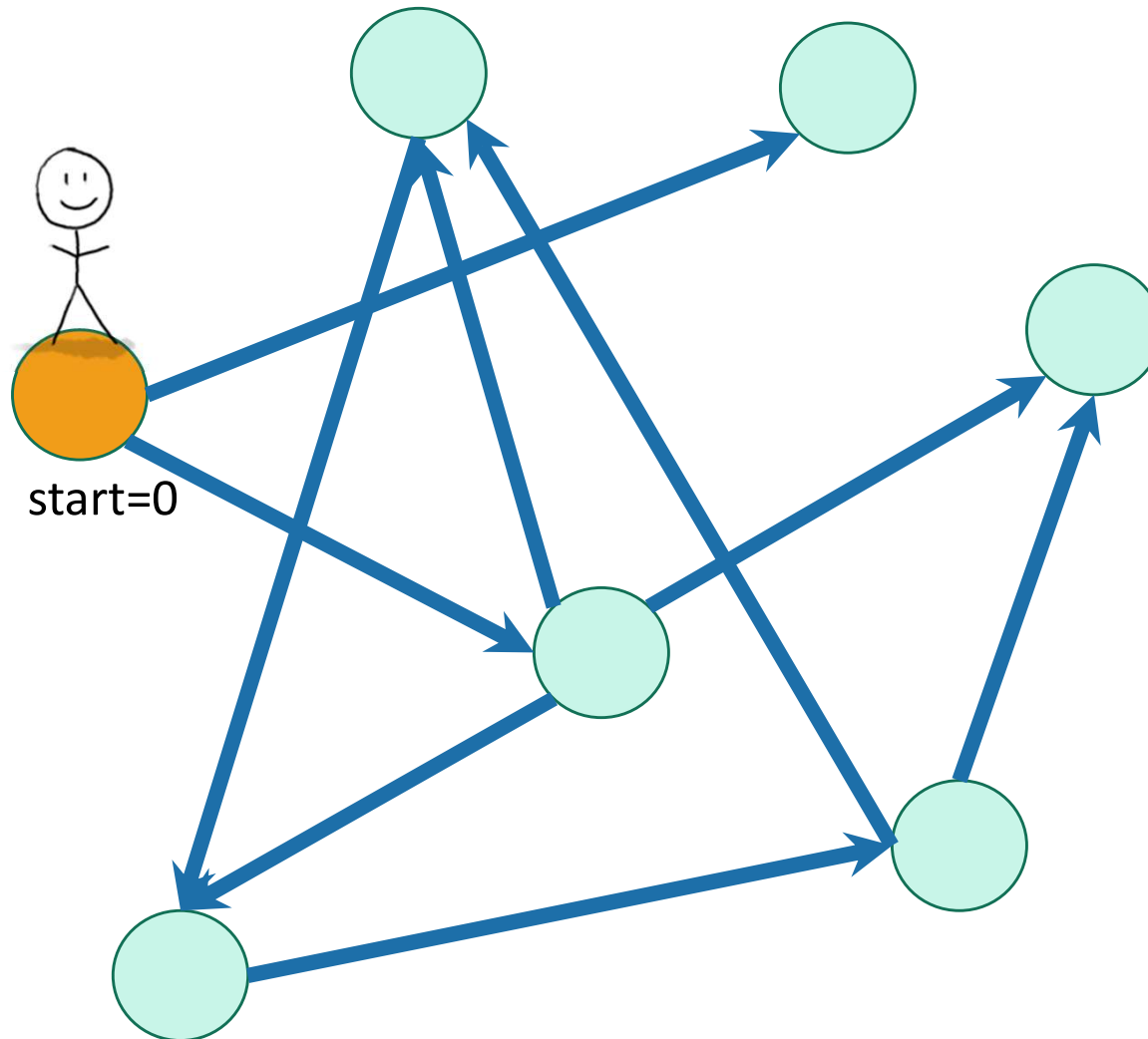
Exploring a labyrinth with chalk and a piece of string






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

Depth First Search

Exploring a labyrinth with chalk and a piece of string



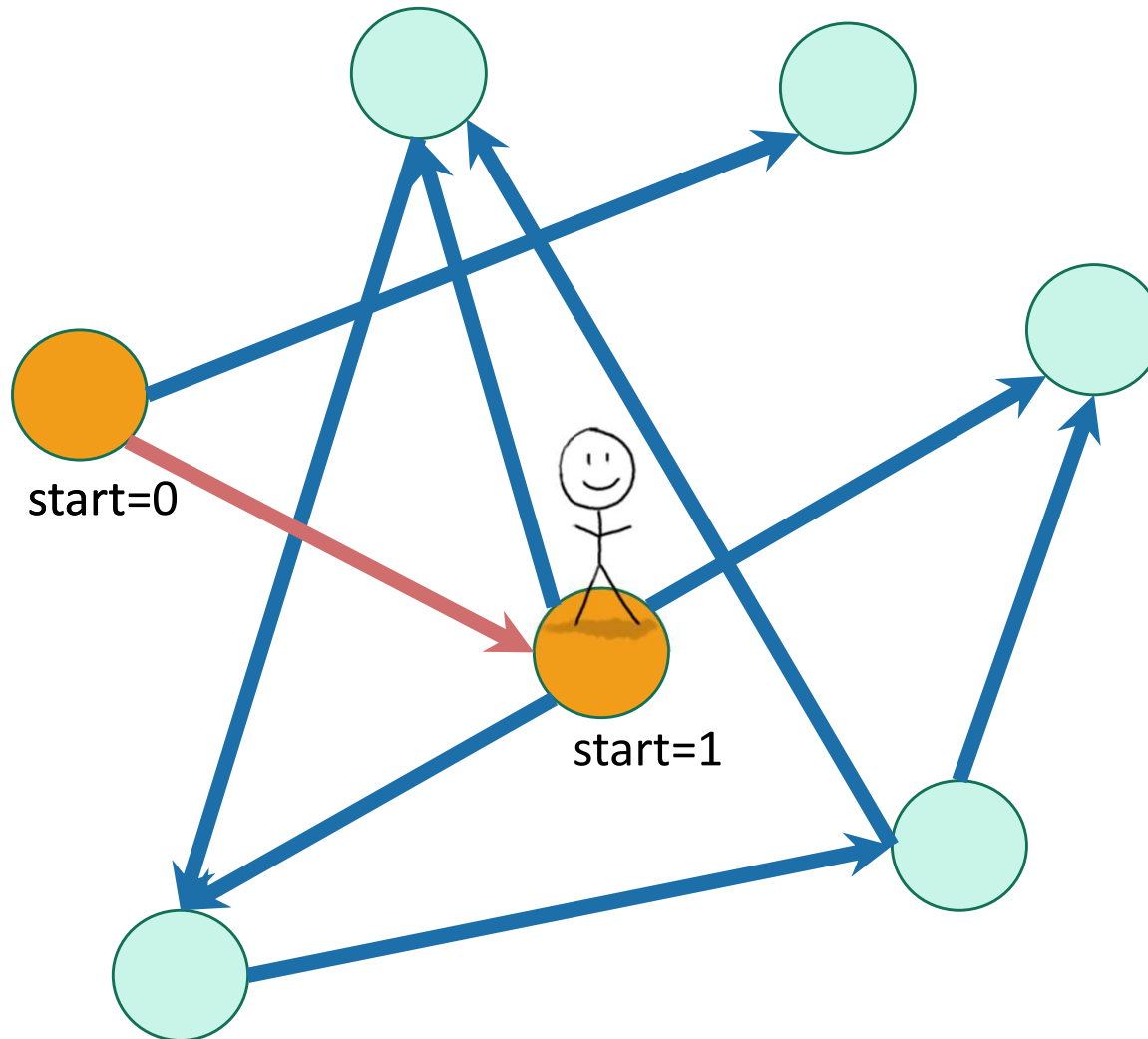
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






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



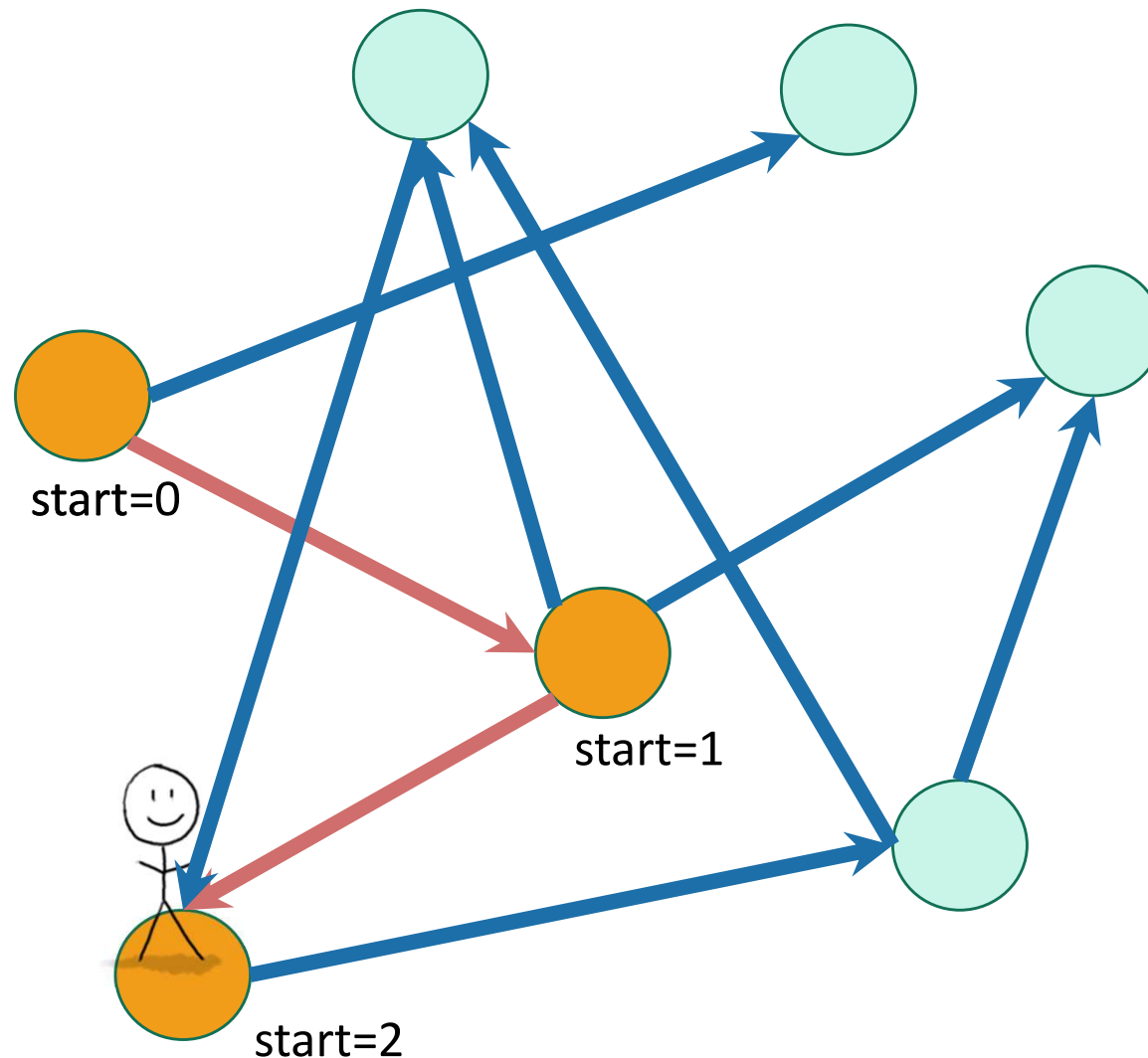
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






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



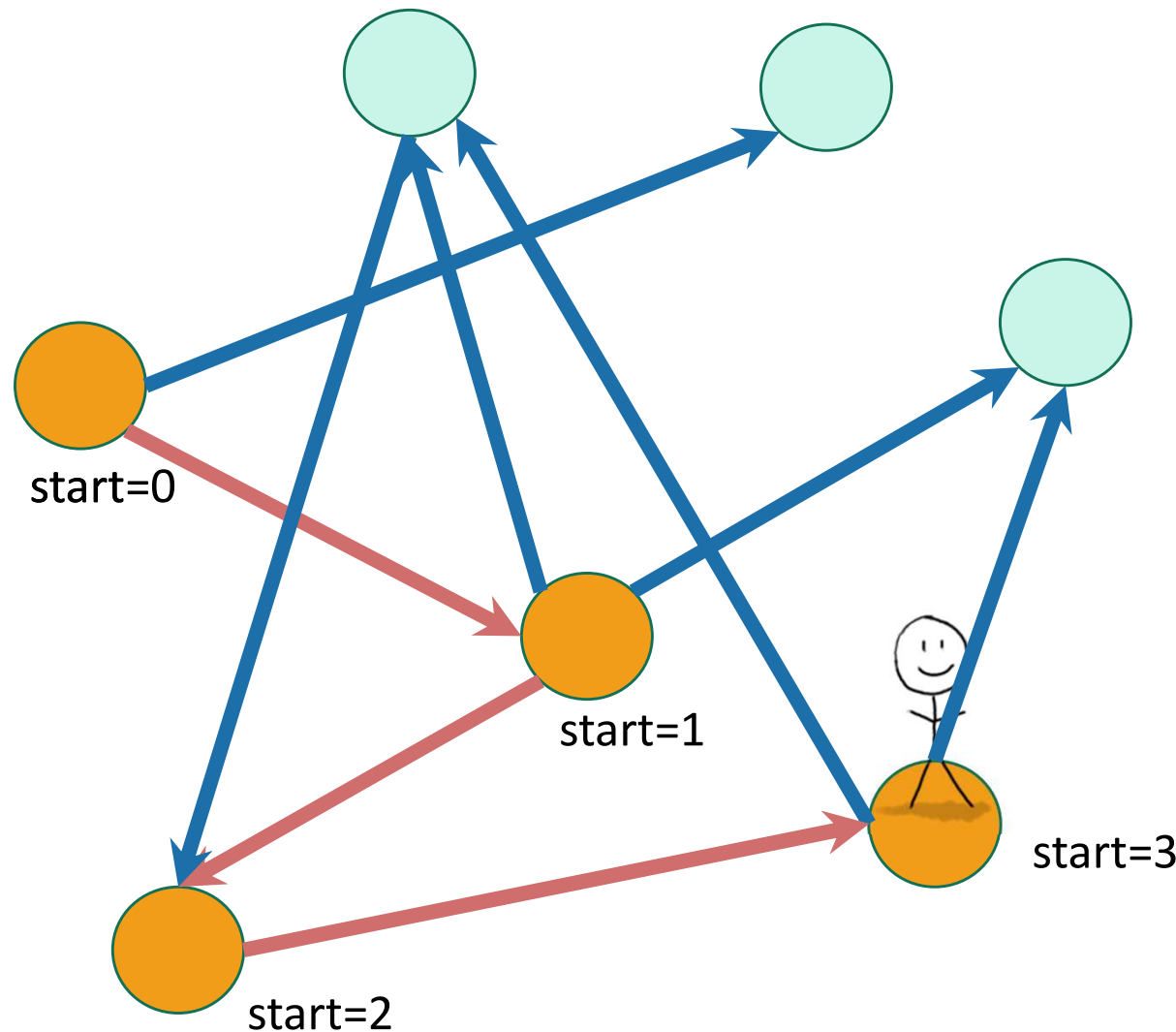
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






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



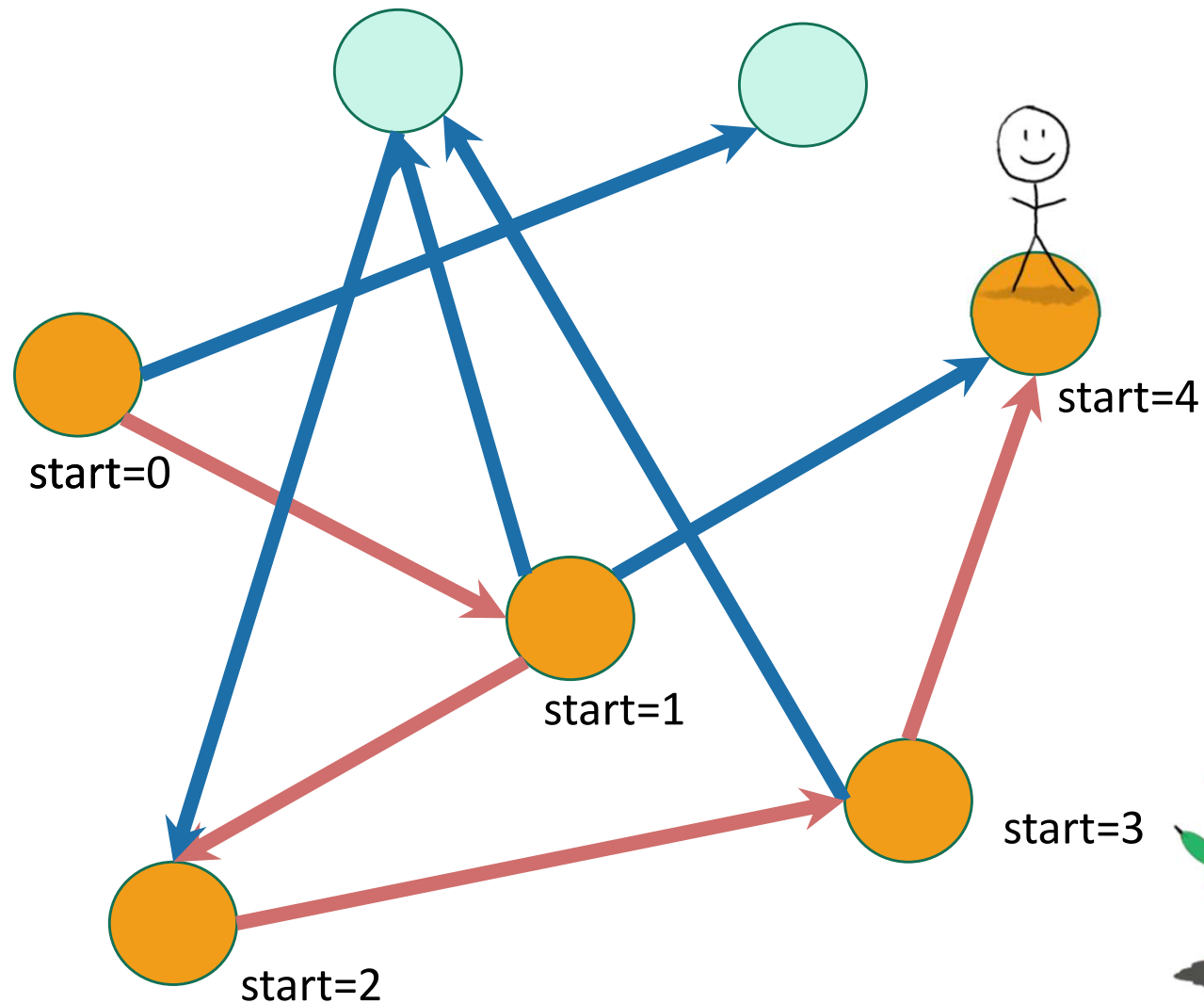
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.






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



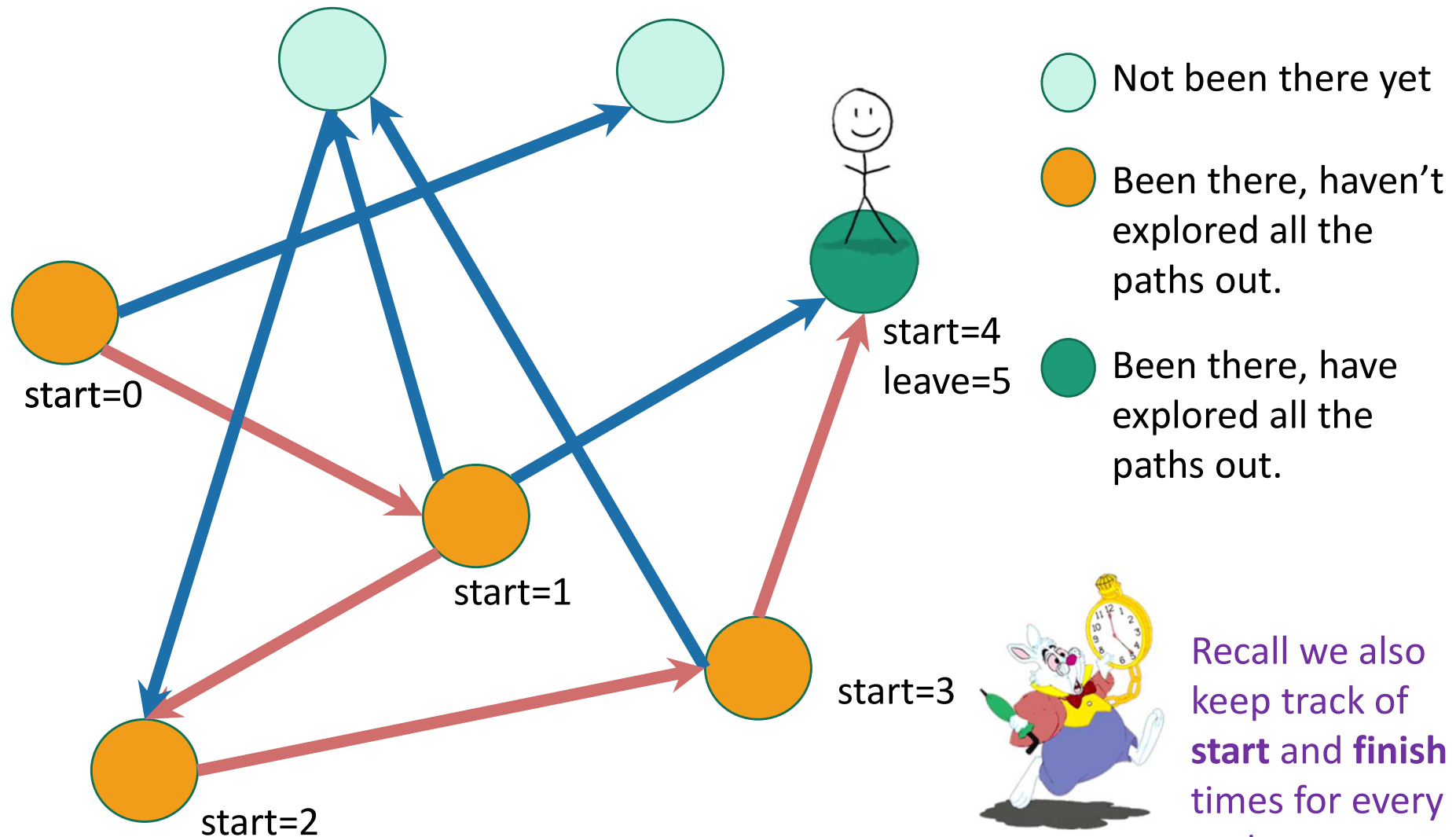
-  Not been there yet
-  Been there, haven't explored all the paths out.
-  Been there, have explored all the paths out.



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

Depth First Search

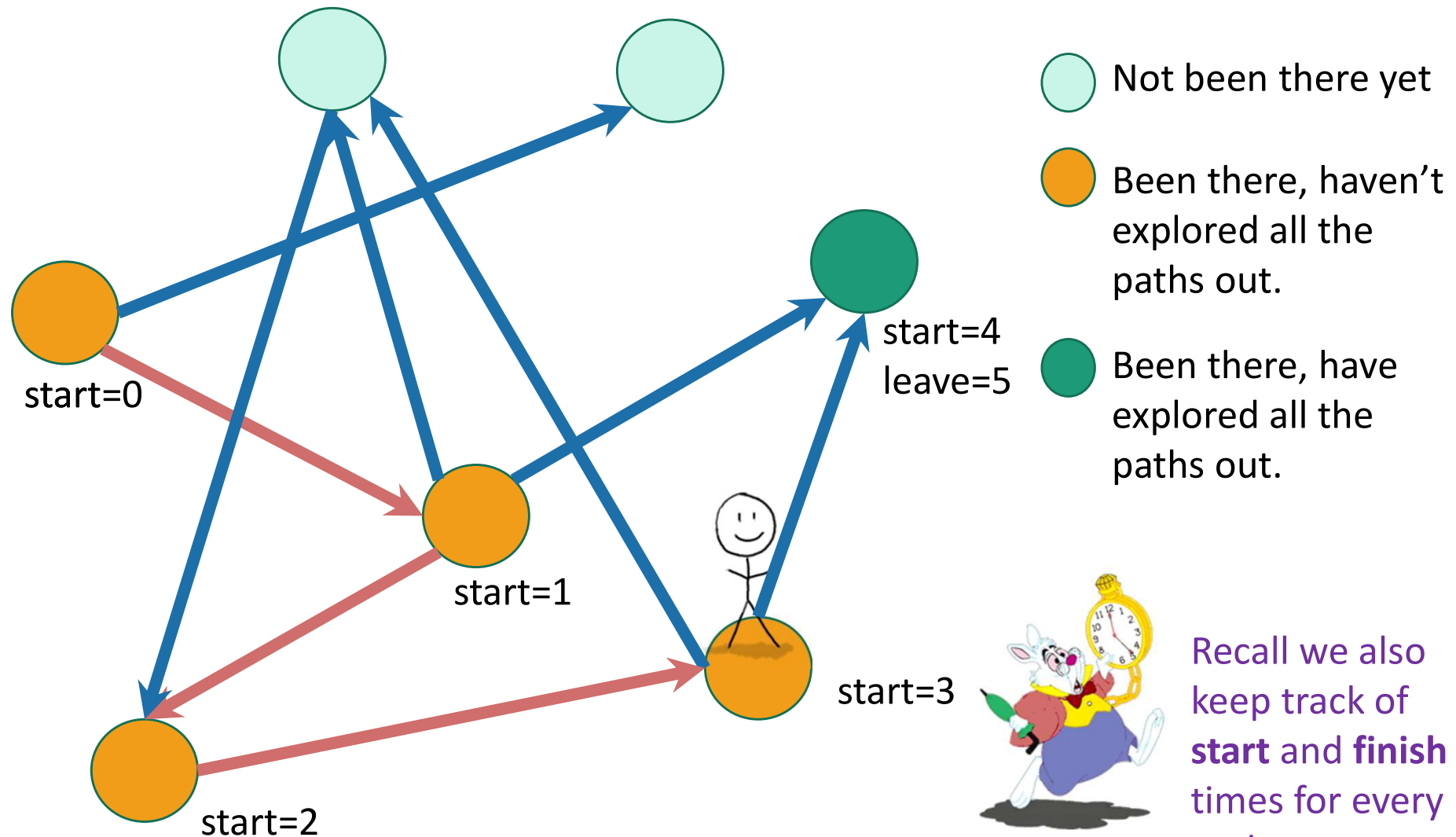
Exploring a labyrinth with chalk and a piece of string



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

Depth First Search

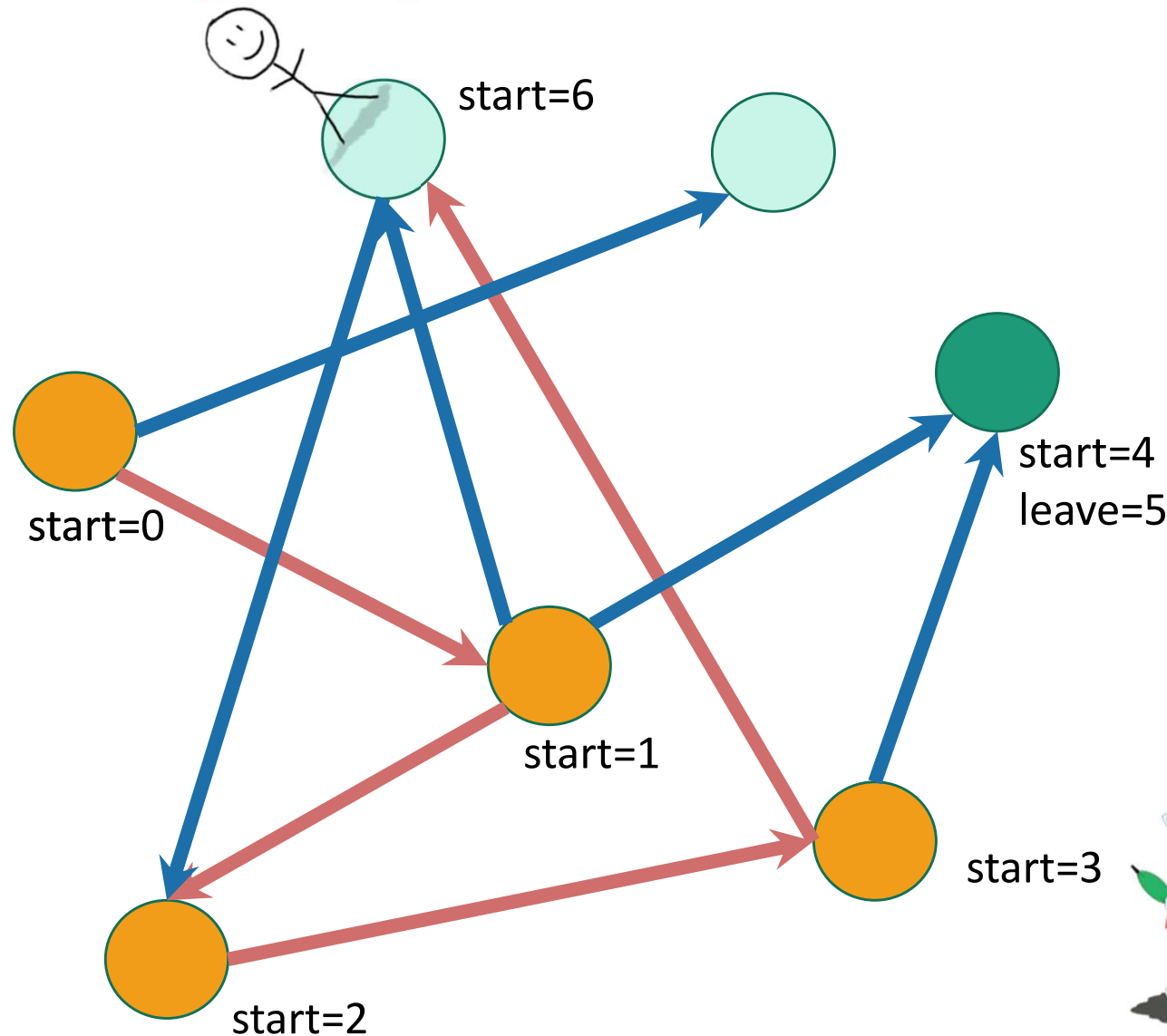
Exploring a labyrinth with chalk and a piece of string



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



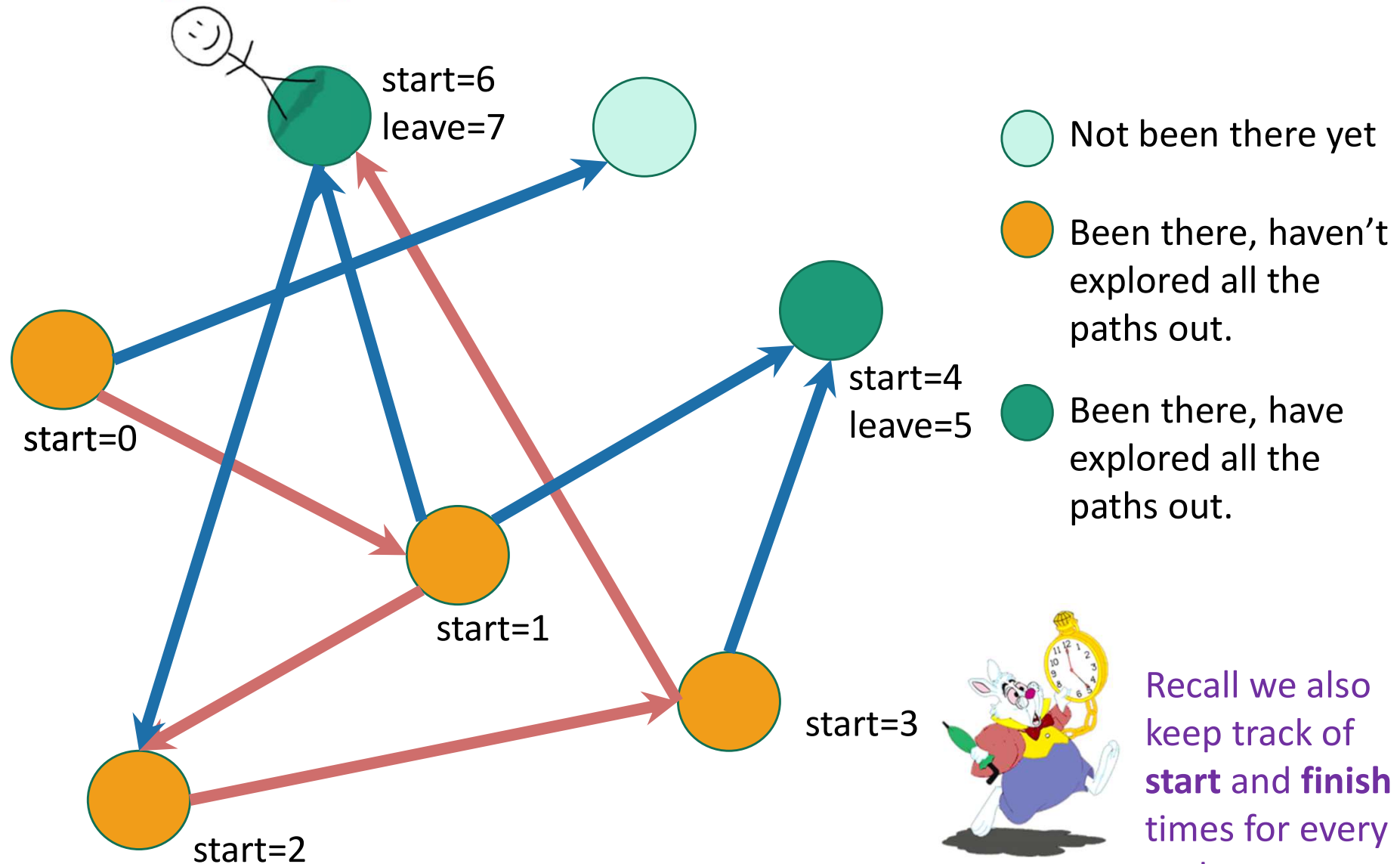
- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.



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

Depth First Search

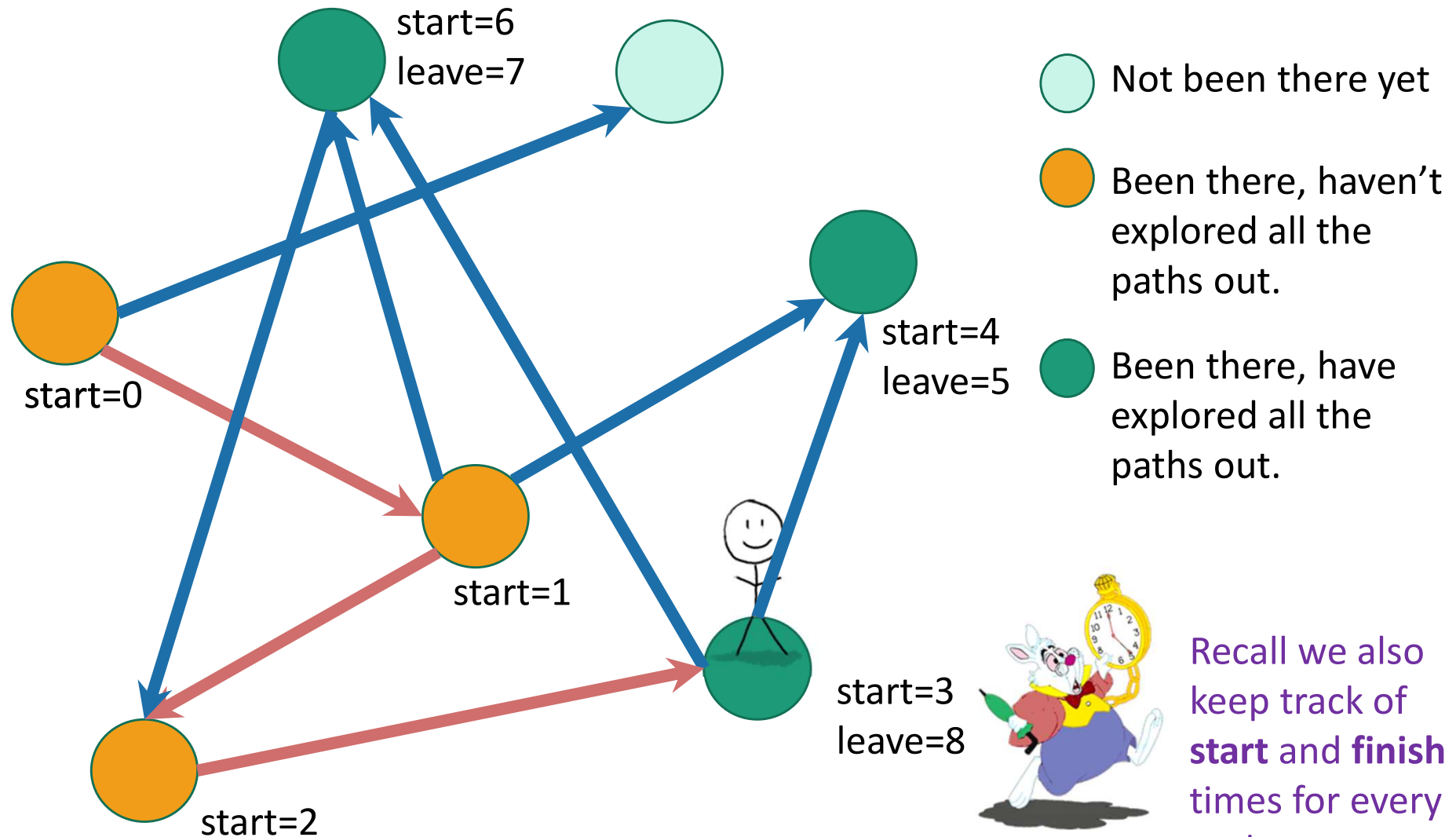
Exploring a labyrinth with chalk and a piece of string



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

Depth First Search

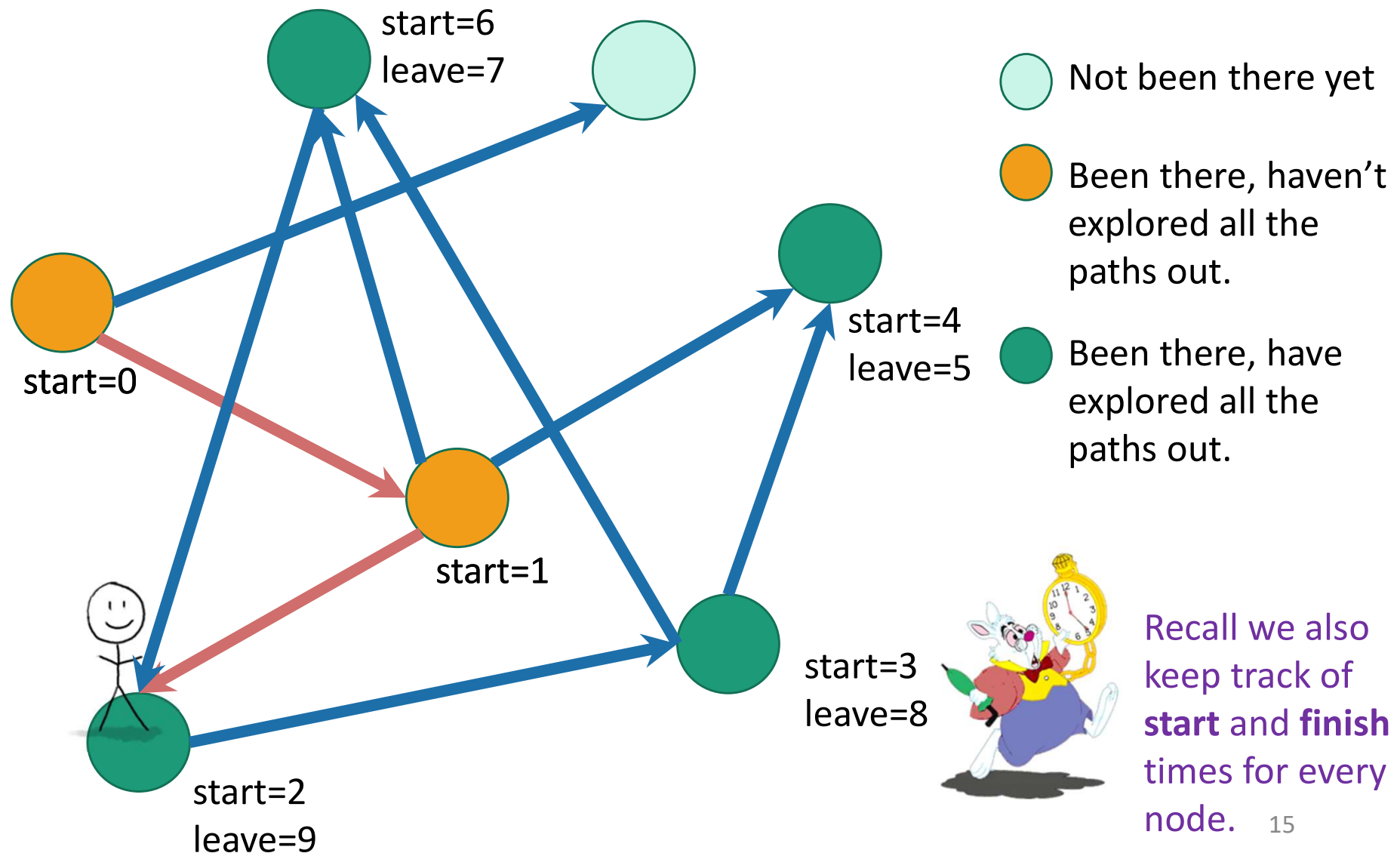
Exploring a labyrinth with chalk and a piece of string



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

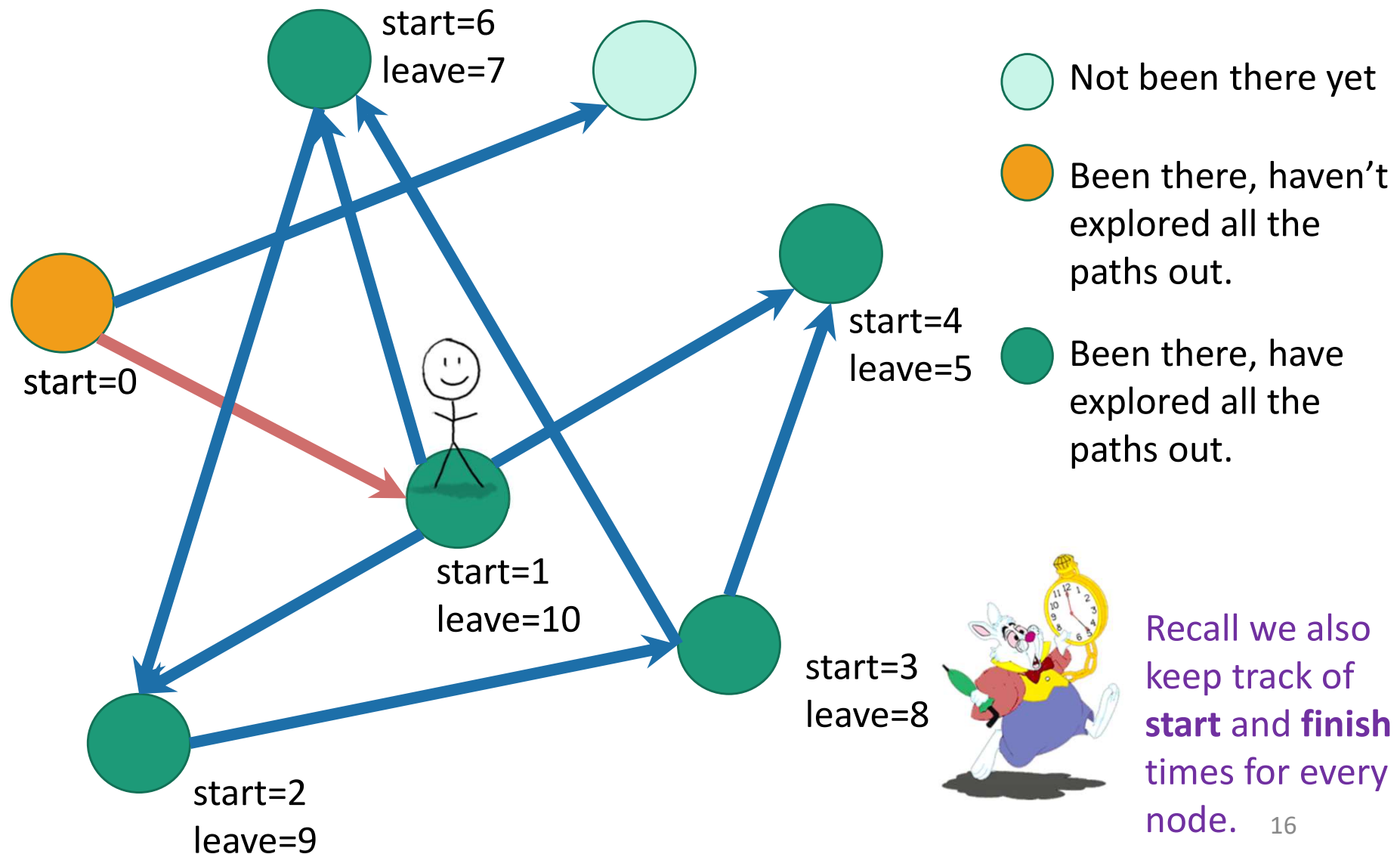
Depth First Search

Exploring a labyrinth with chalk and a piece of string



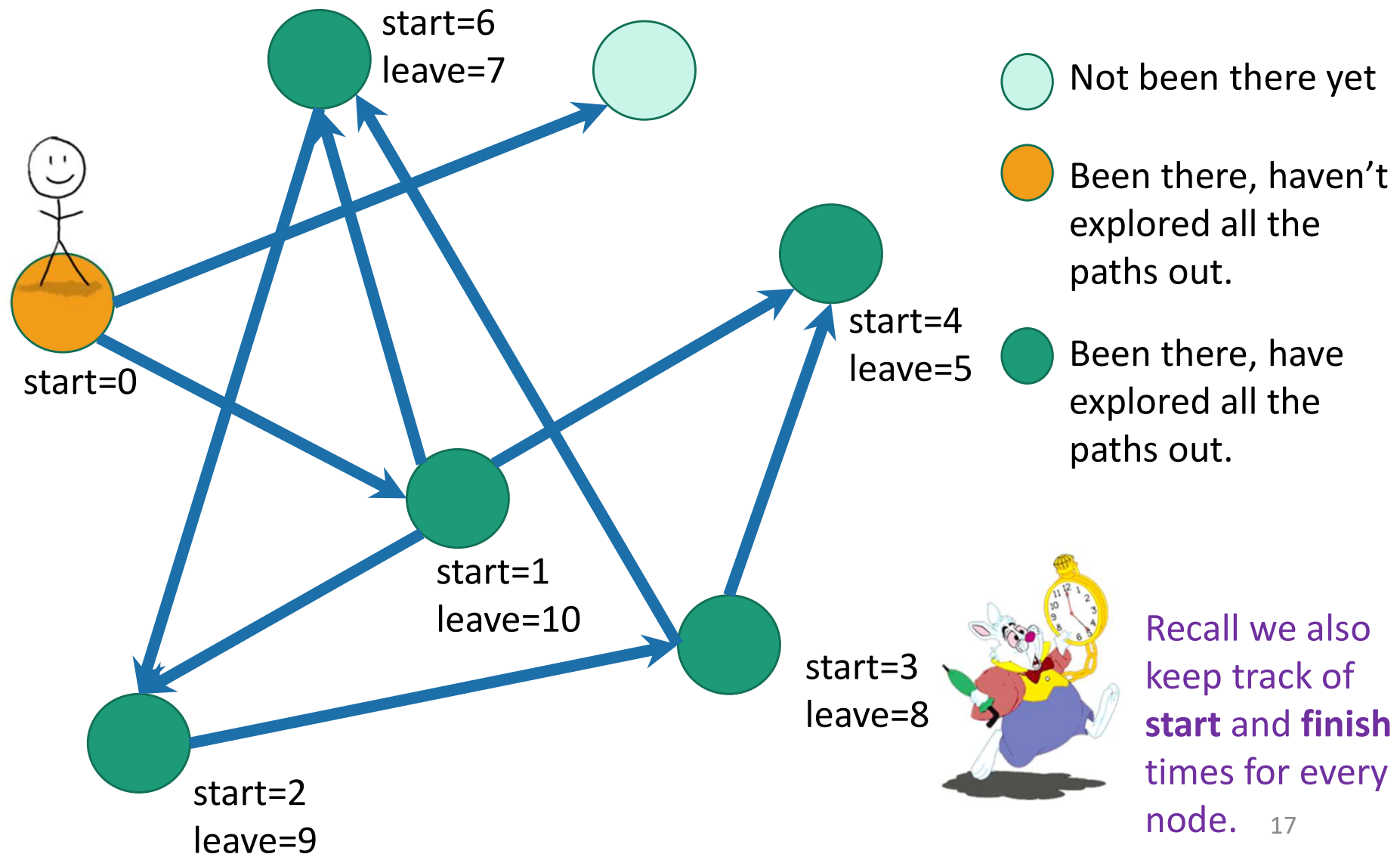
Depth First Search

Exploring a labyrinth with chalk and a piece of string



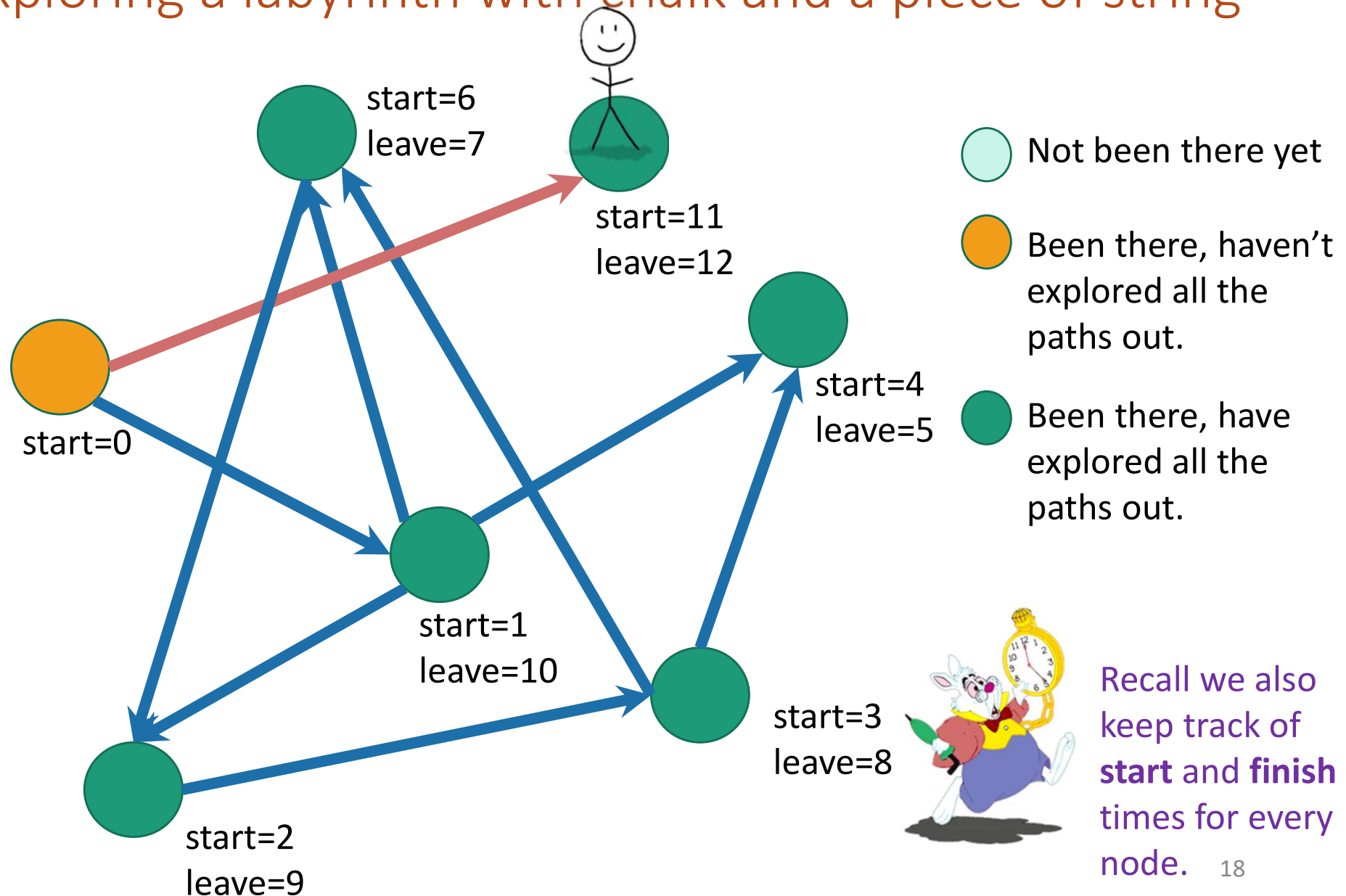
Depth First Search

Exploring a labyrinth with chalk and a piece of string



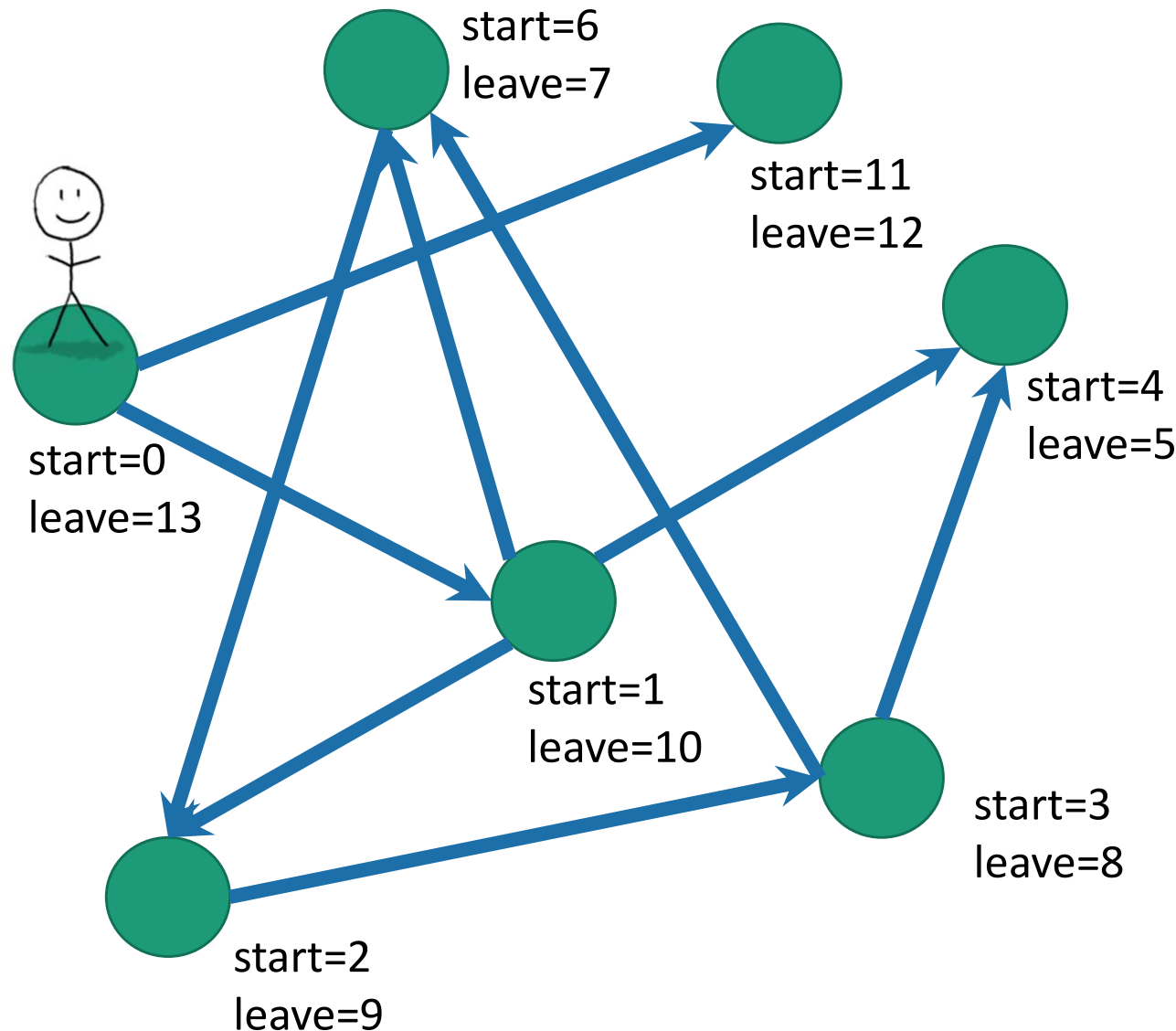
Depth First Search

Exploring a labyrinth with chalk and a piece of string



Depth First Search

Exploring a labyrinth with chalk and a piece of string



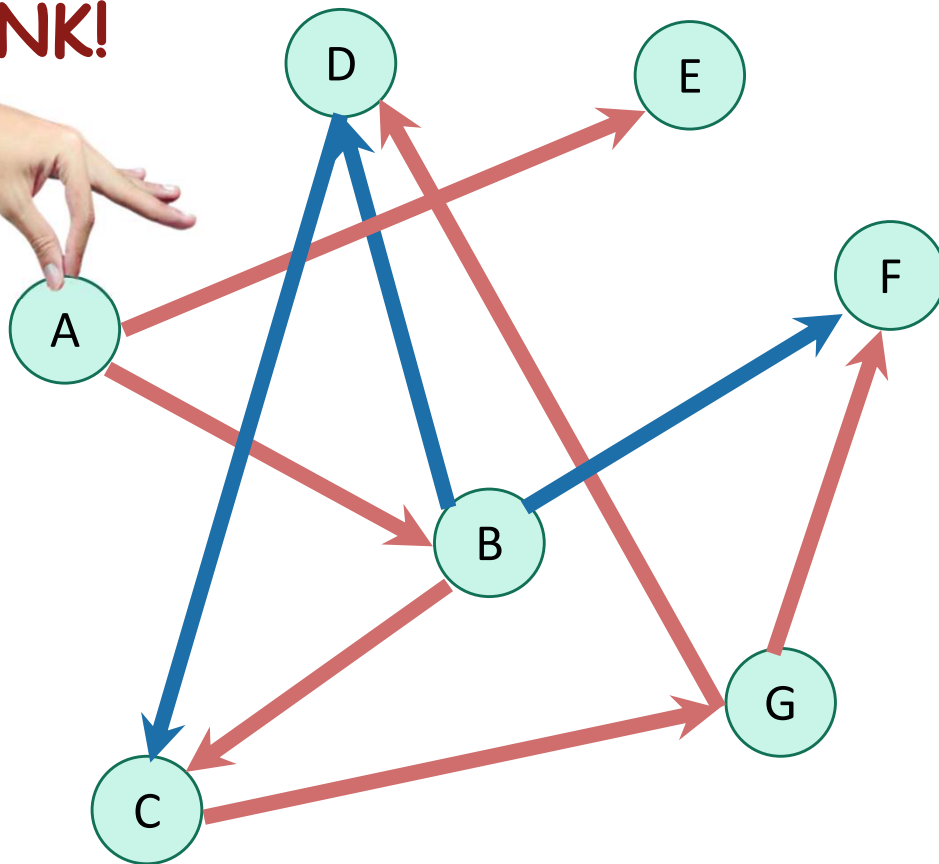
- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

**Labyrinth:
explored!**

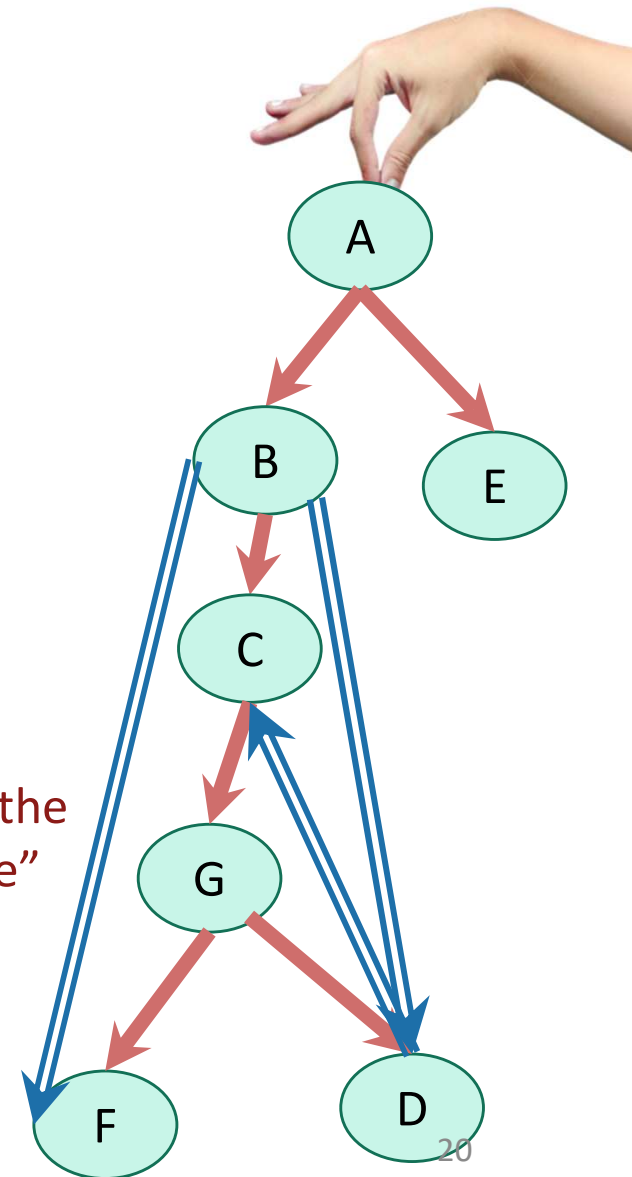
Depth first search

implicitly creates a tree on everything you can reach

YOINK!

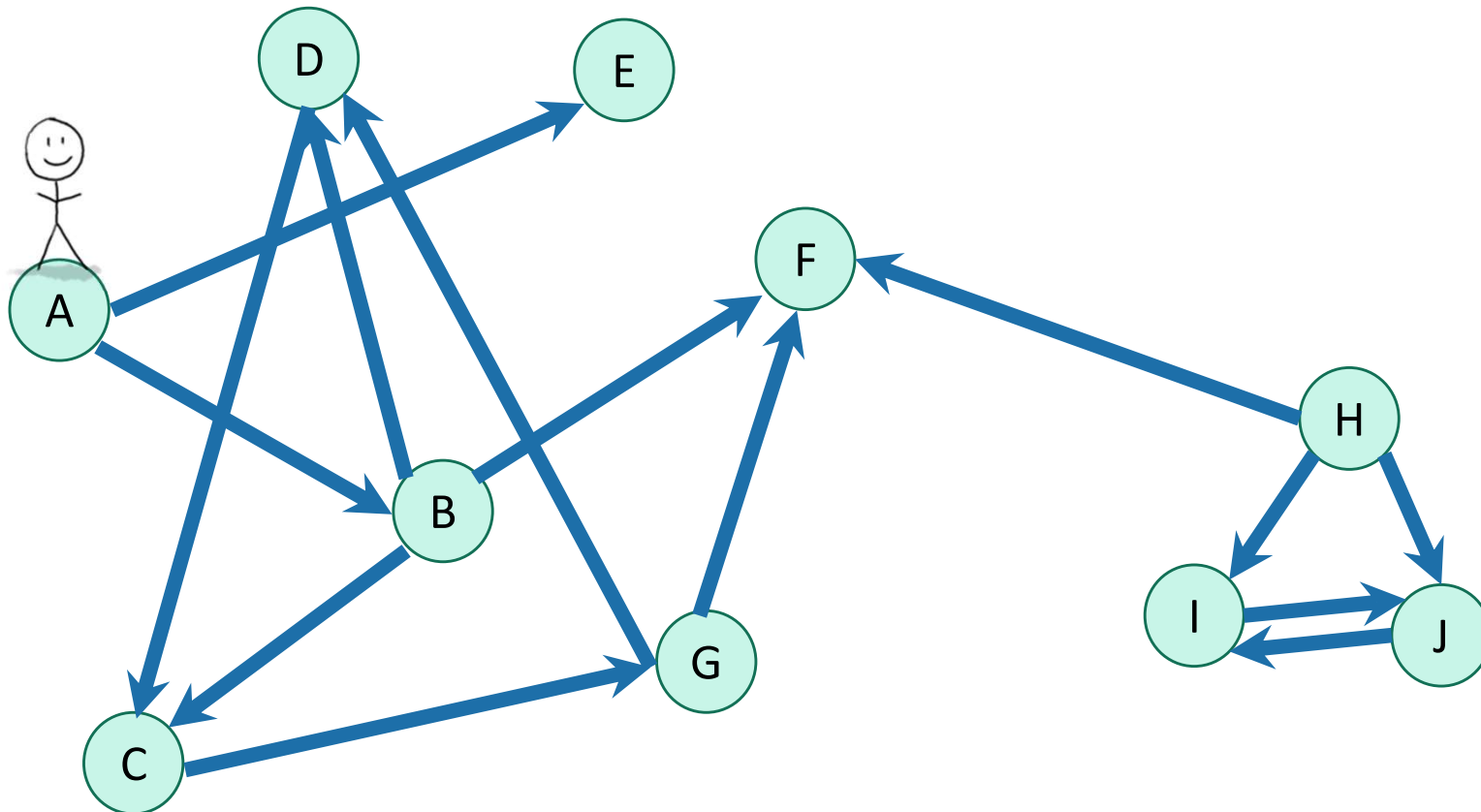


Call this the
"DFS tree"



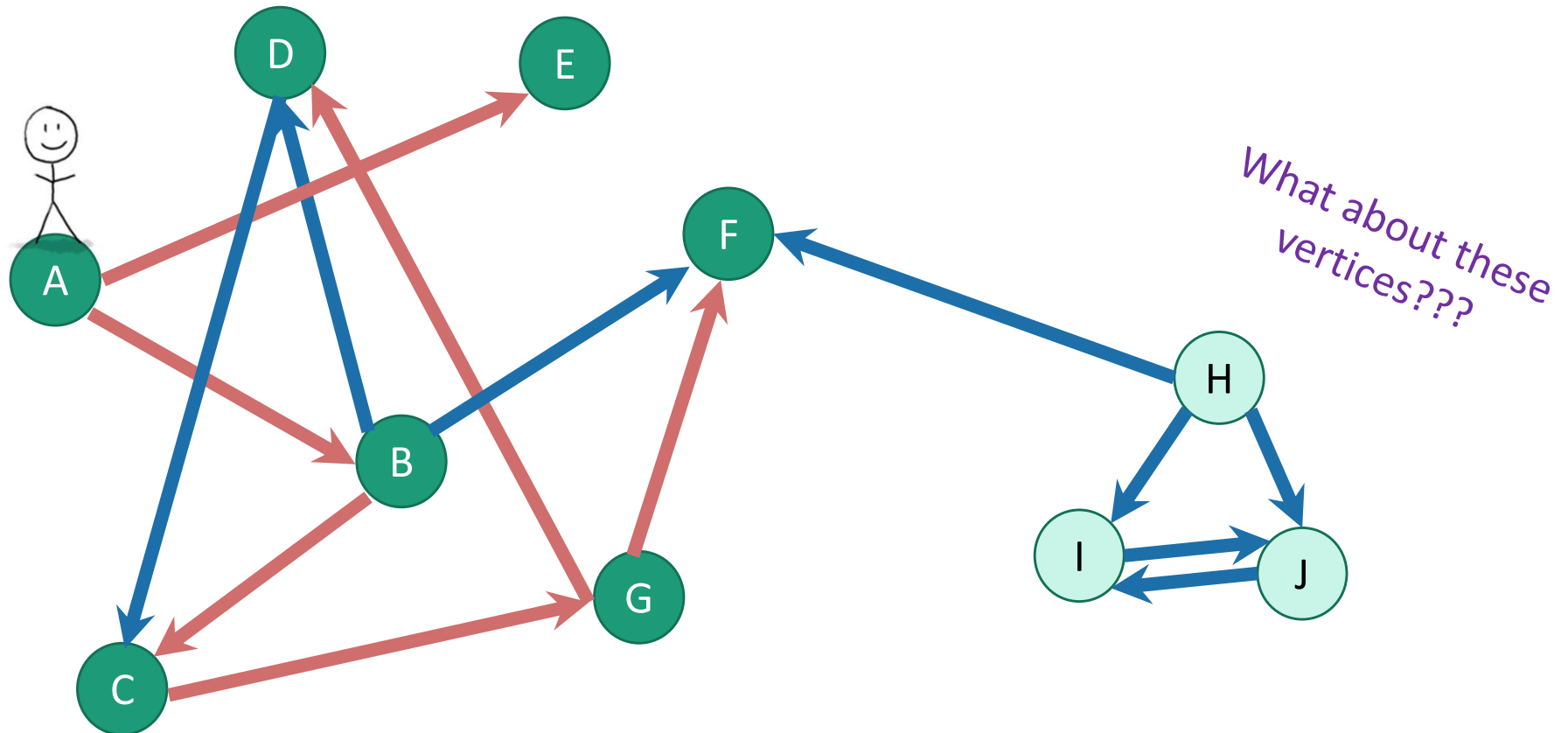
When you can't reach everything

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



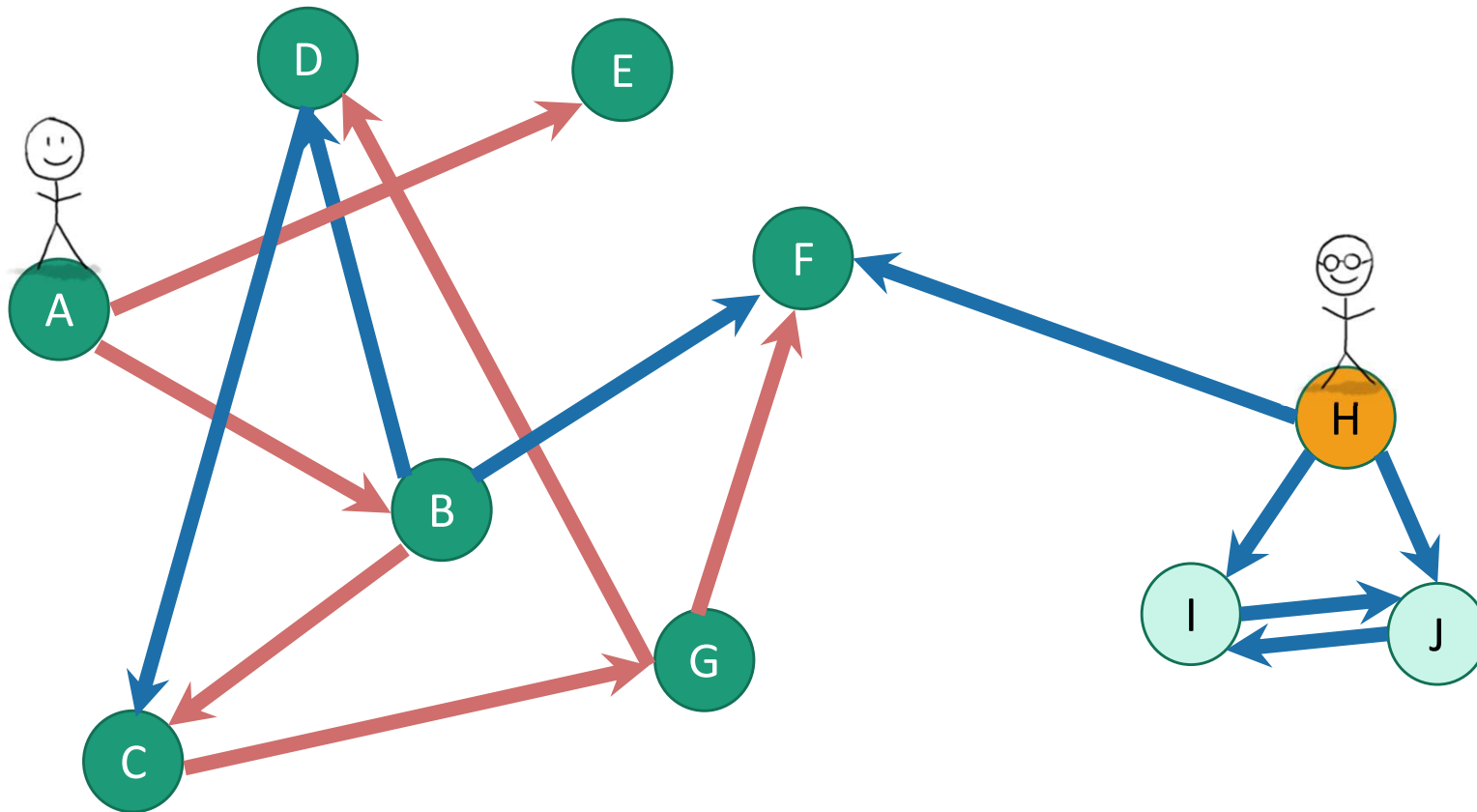
When you can't reach everything

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



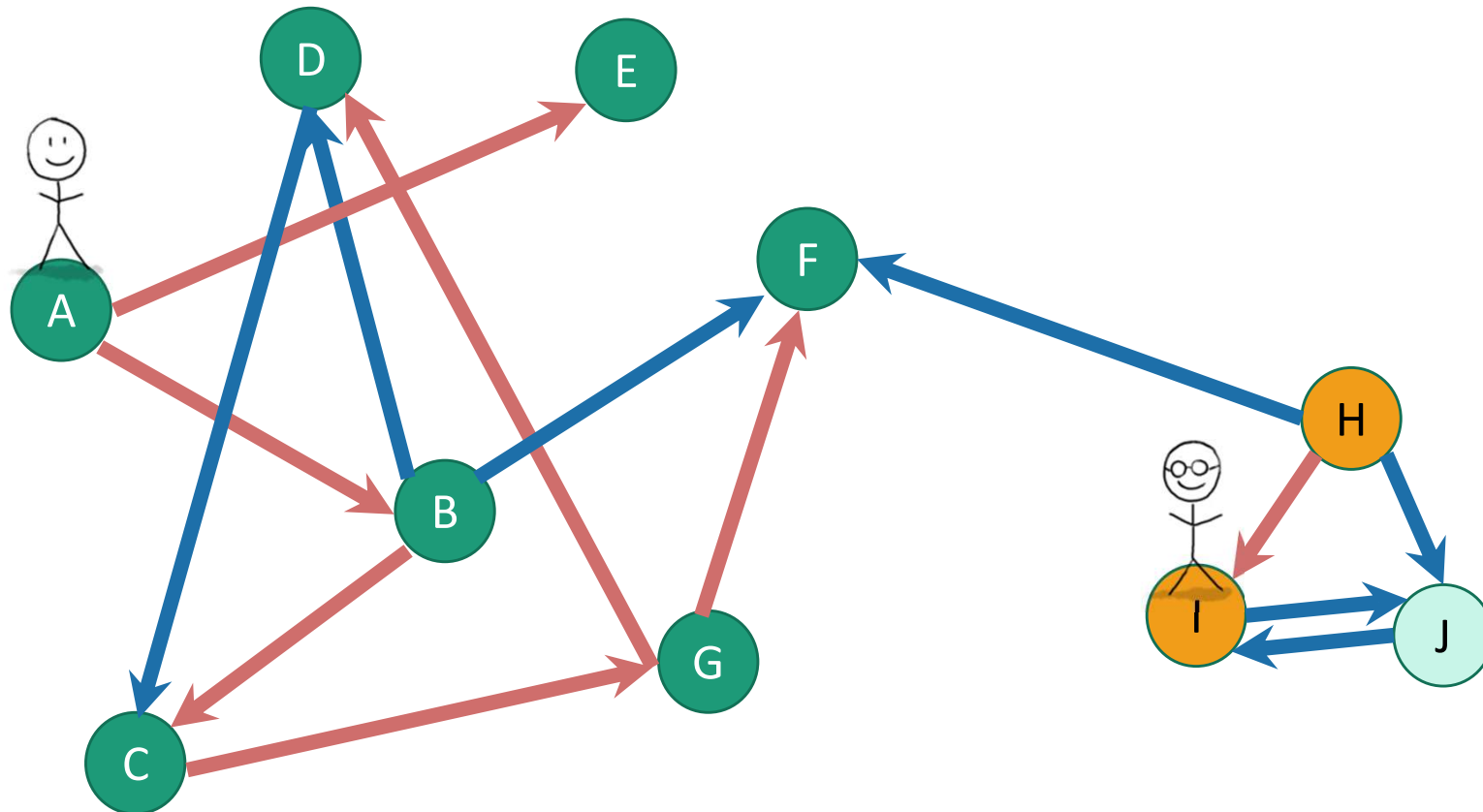
When you can't reach everything

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



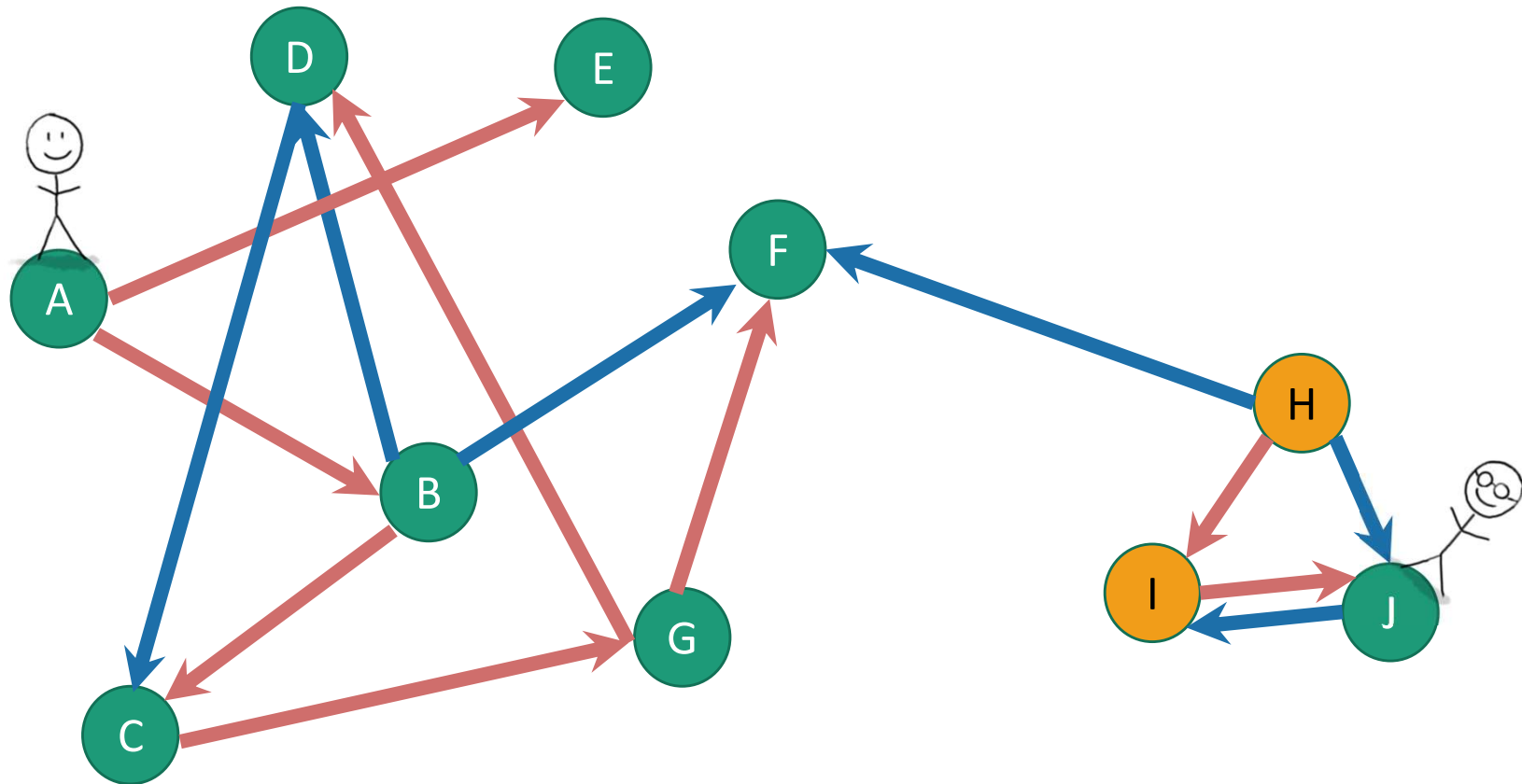
When you can't reach everything

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



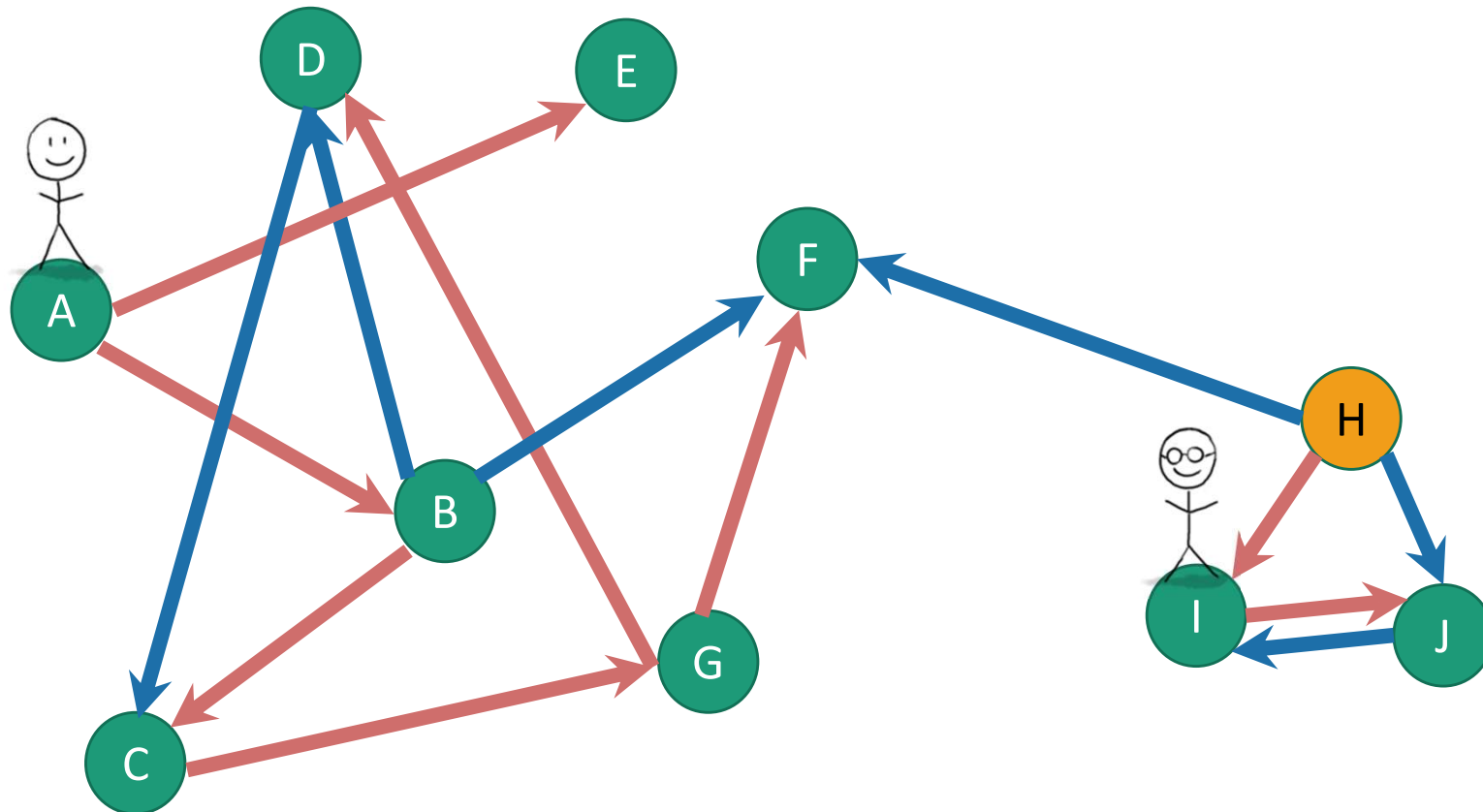
When you can't reach everything

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



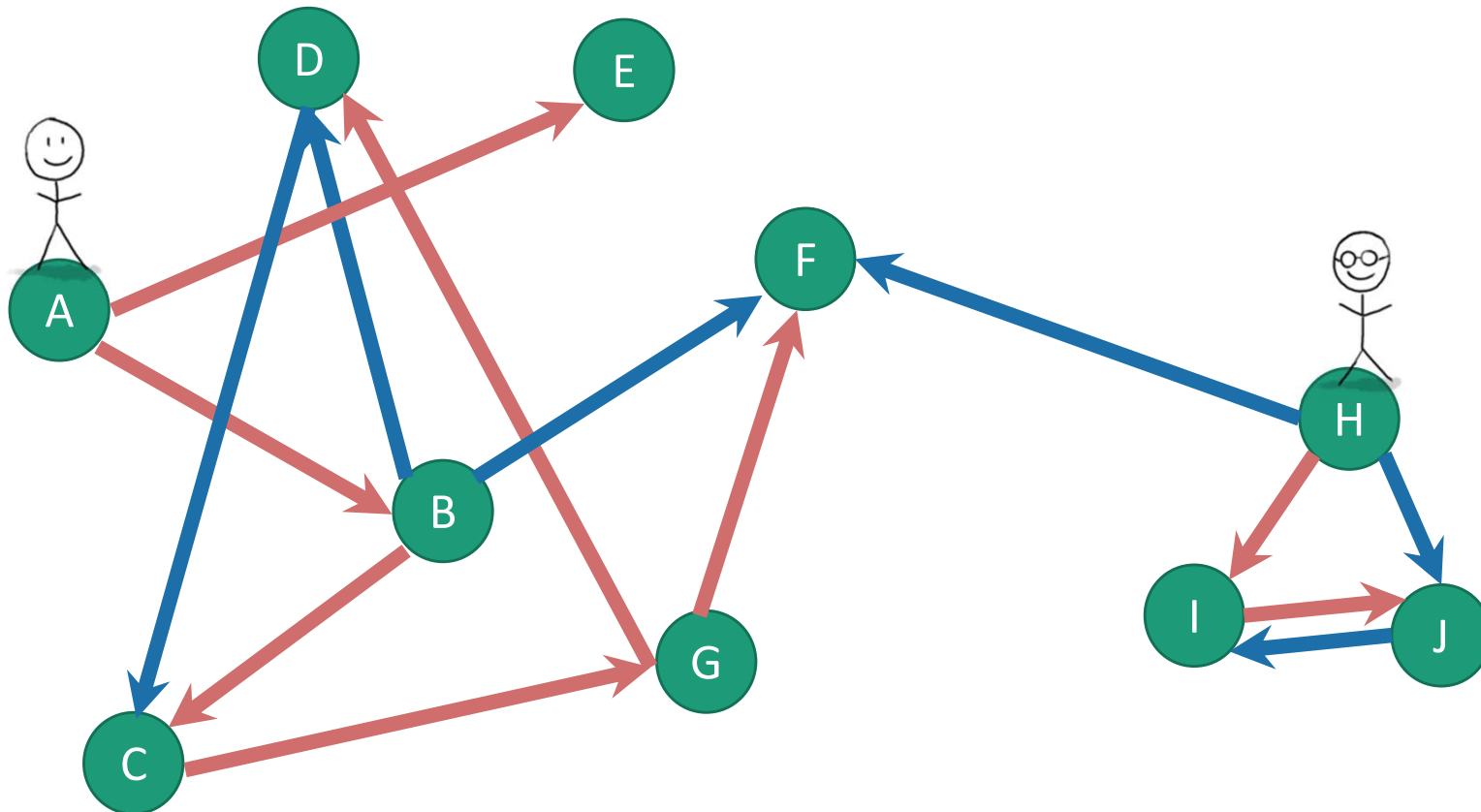
When you can't reach everything

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



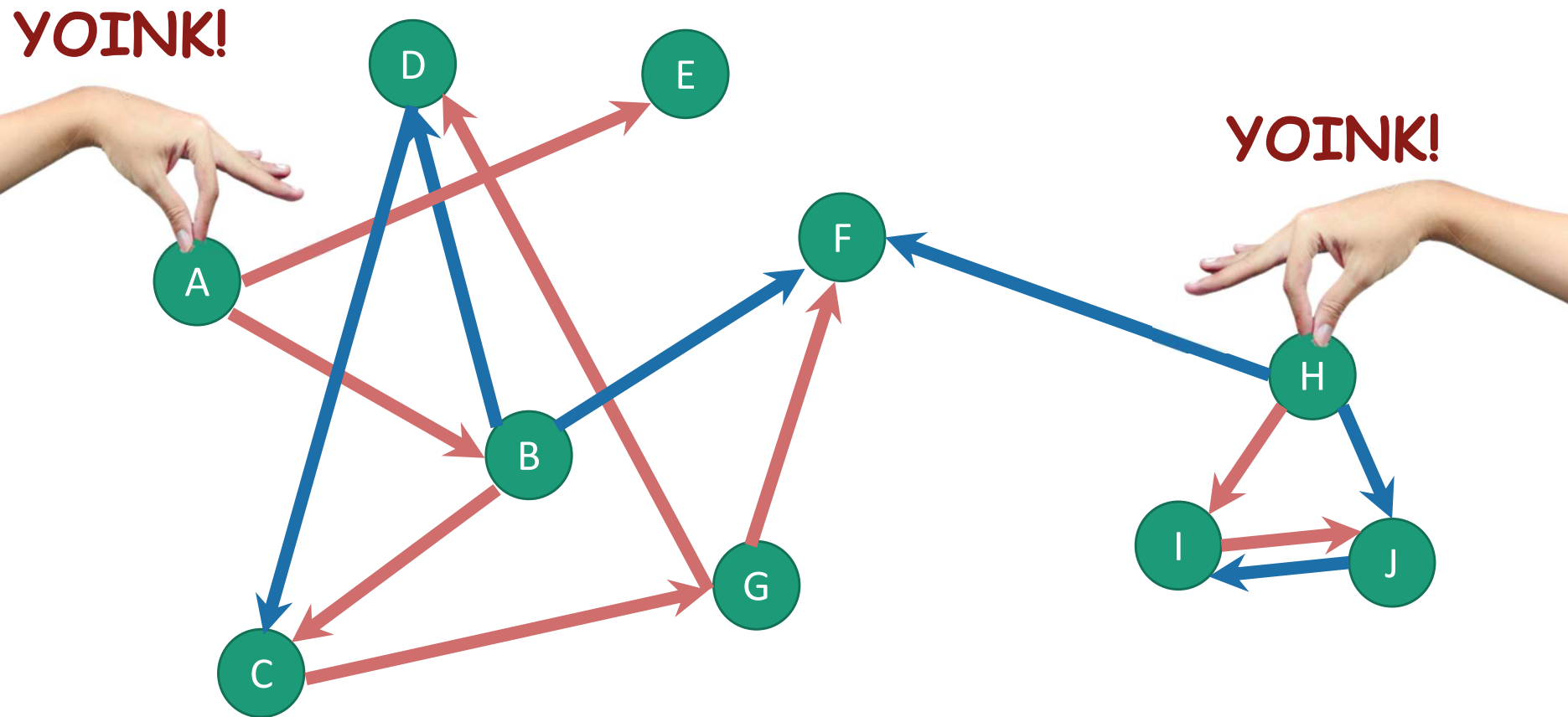
When you can't reach everything

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



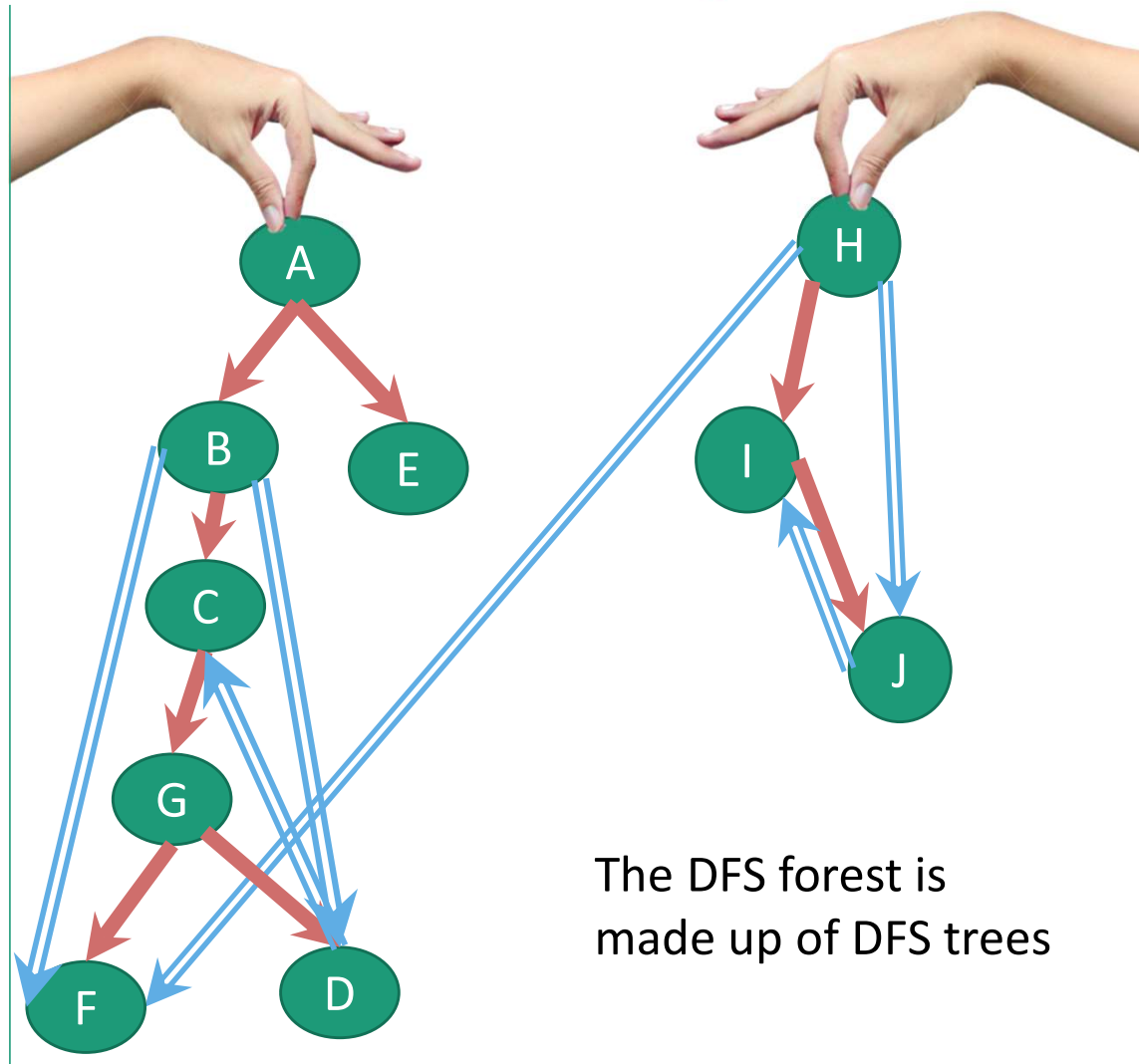
When you can't reach everything

- 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:



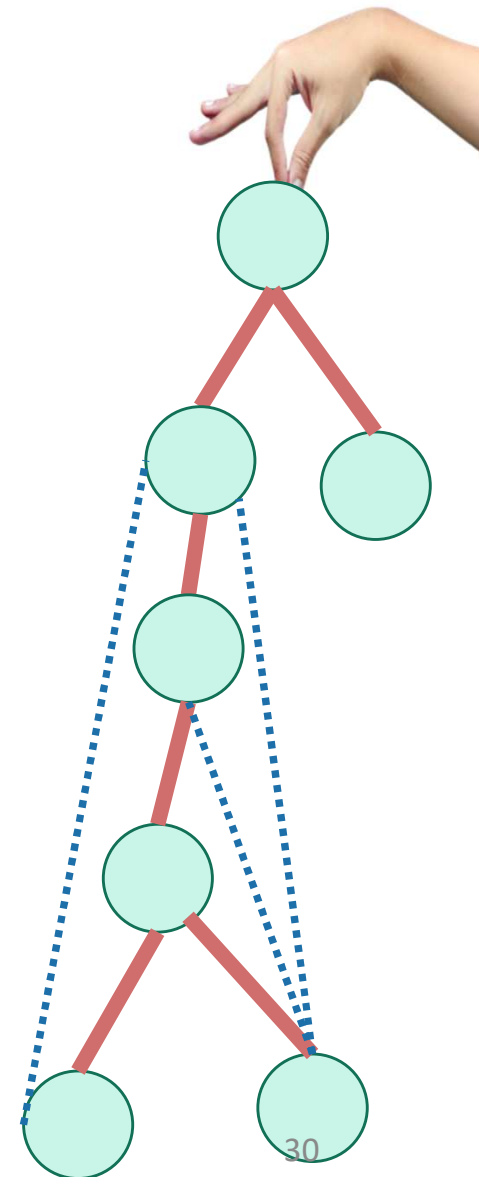
- If neither are descendants of each other:



(or the other way around)

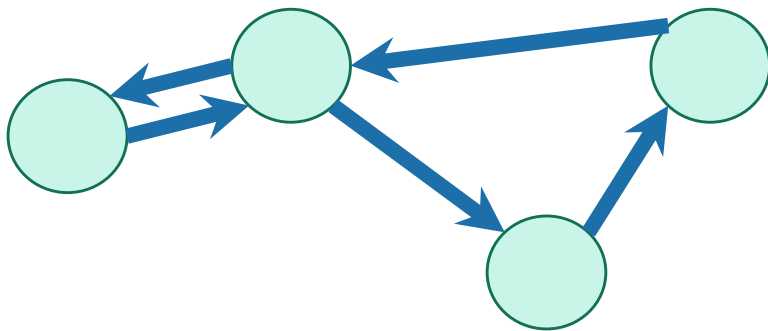
If v and w are in different trees, it's always this last one.

DFS tree

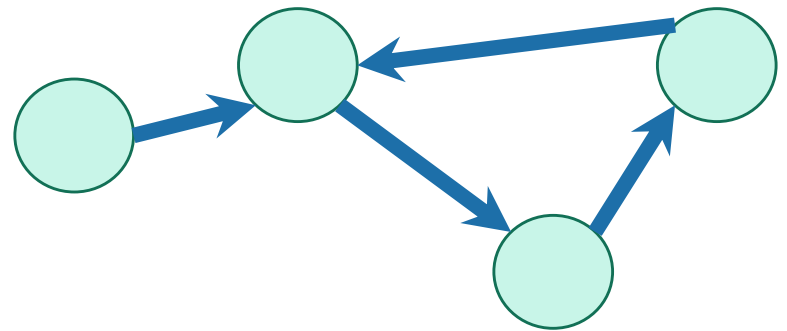


Strongly connected components

- 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 .



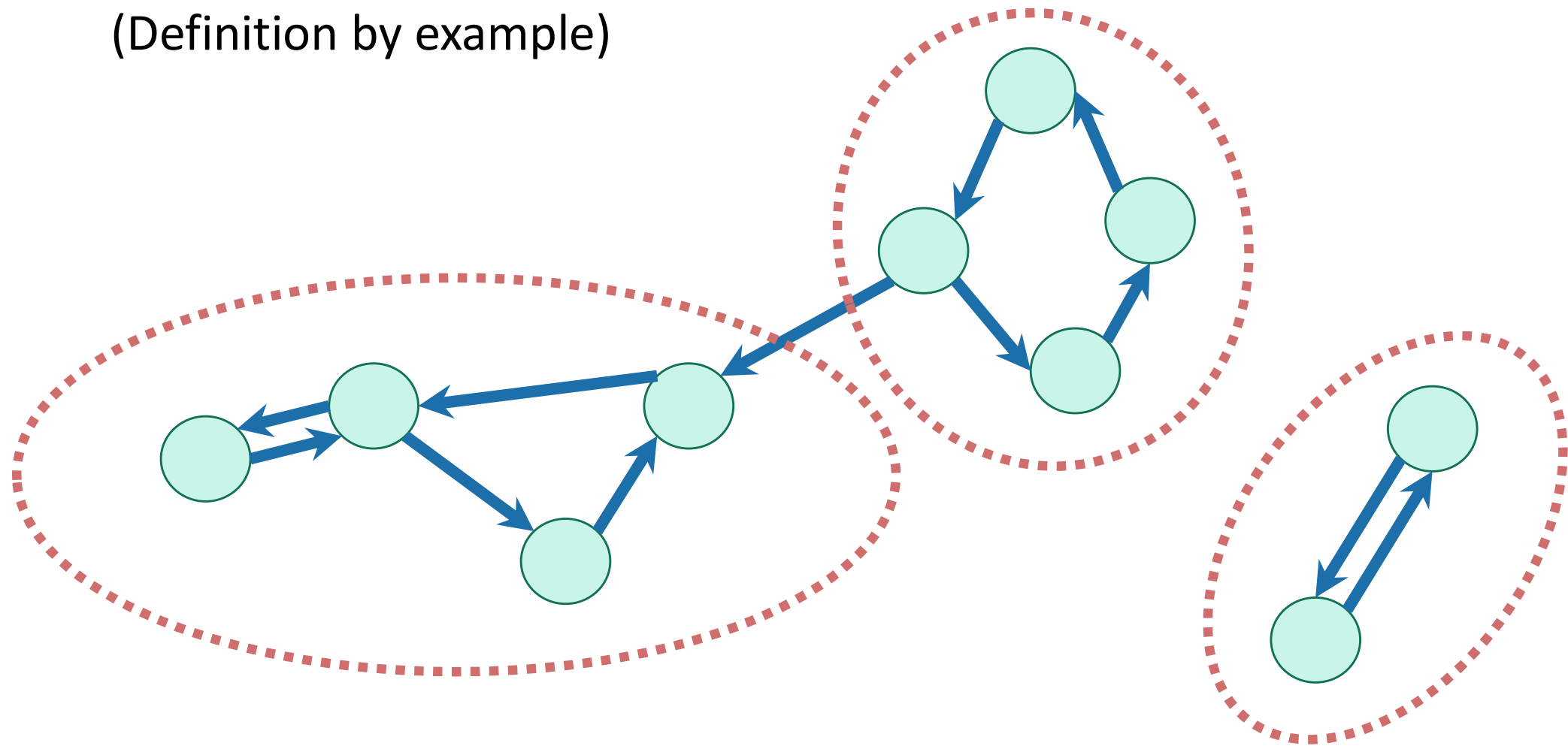
strongly connected



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?

- Strongly connected components tell you about **communities**.
- Lots of graph algorithms only make sense on SCCs.
 - So sometimes we want to find the SCCs as a first step.
 - E.g., algorithms for solving 2-SAT (you're not expected to know this).

$$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$$

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.

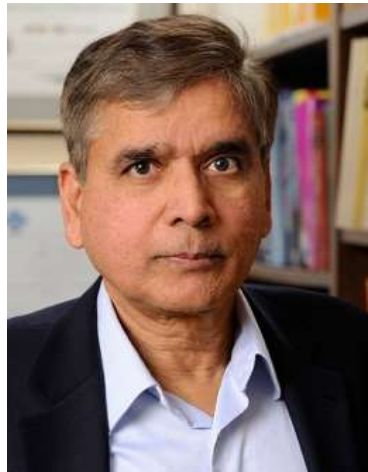
This will not be our final solution so don't worry too much about it...



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

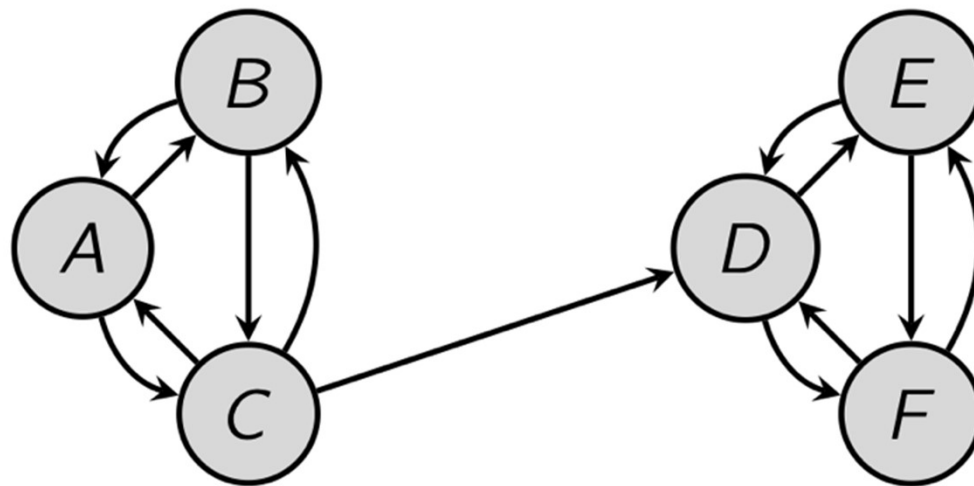
Thuật toán Kosaraju

- Tìm miền liên thông mạnh (SCC) với độ phức tạp $O(n+m)$!!!



Ví dụ khởi động

- Run DFS starting at D:



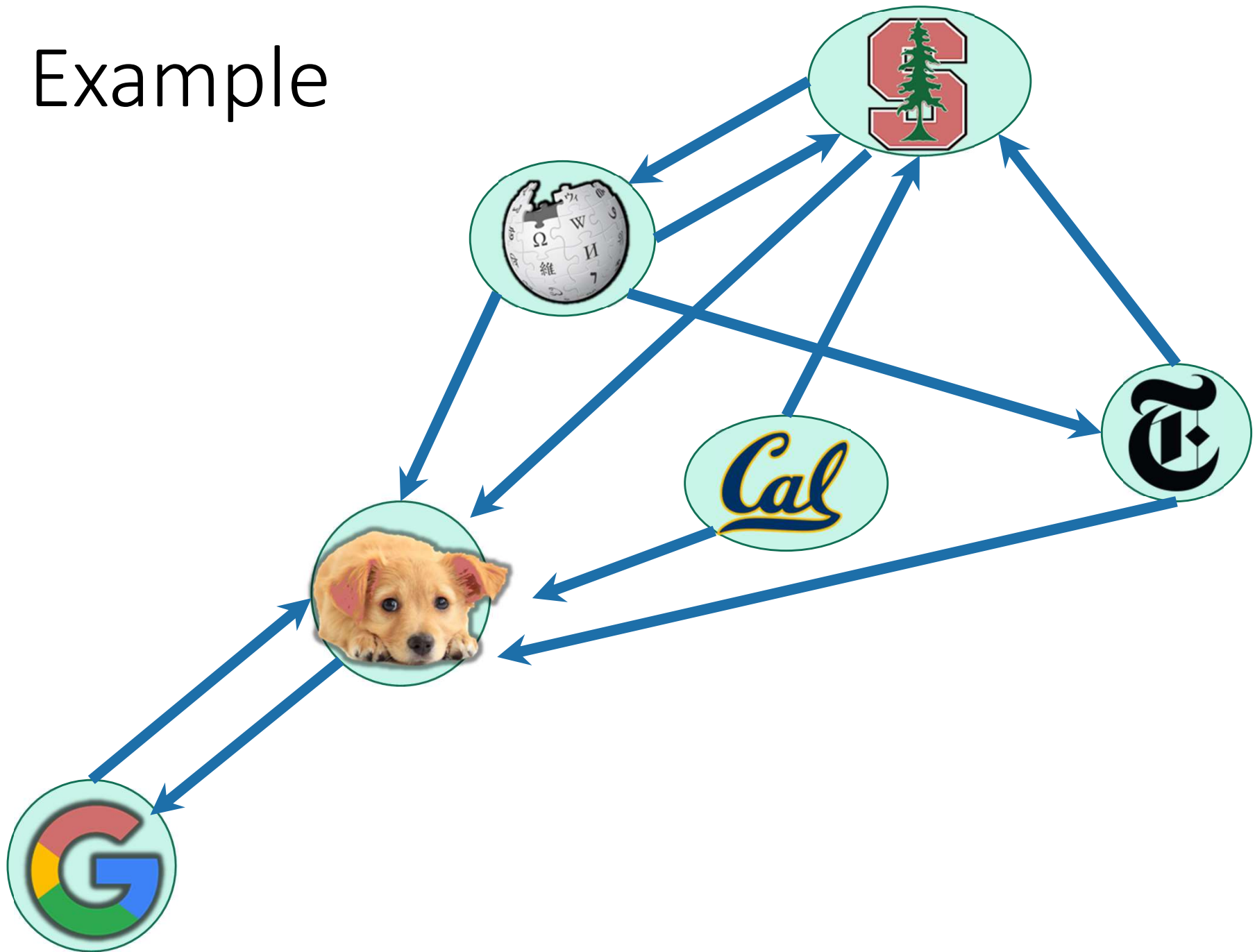
- That will identify SCCs...
- Issues:
 - How do we know where to start DFS?
 - It wouldn't have found the SCCs if we started from A.

Algorithm

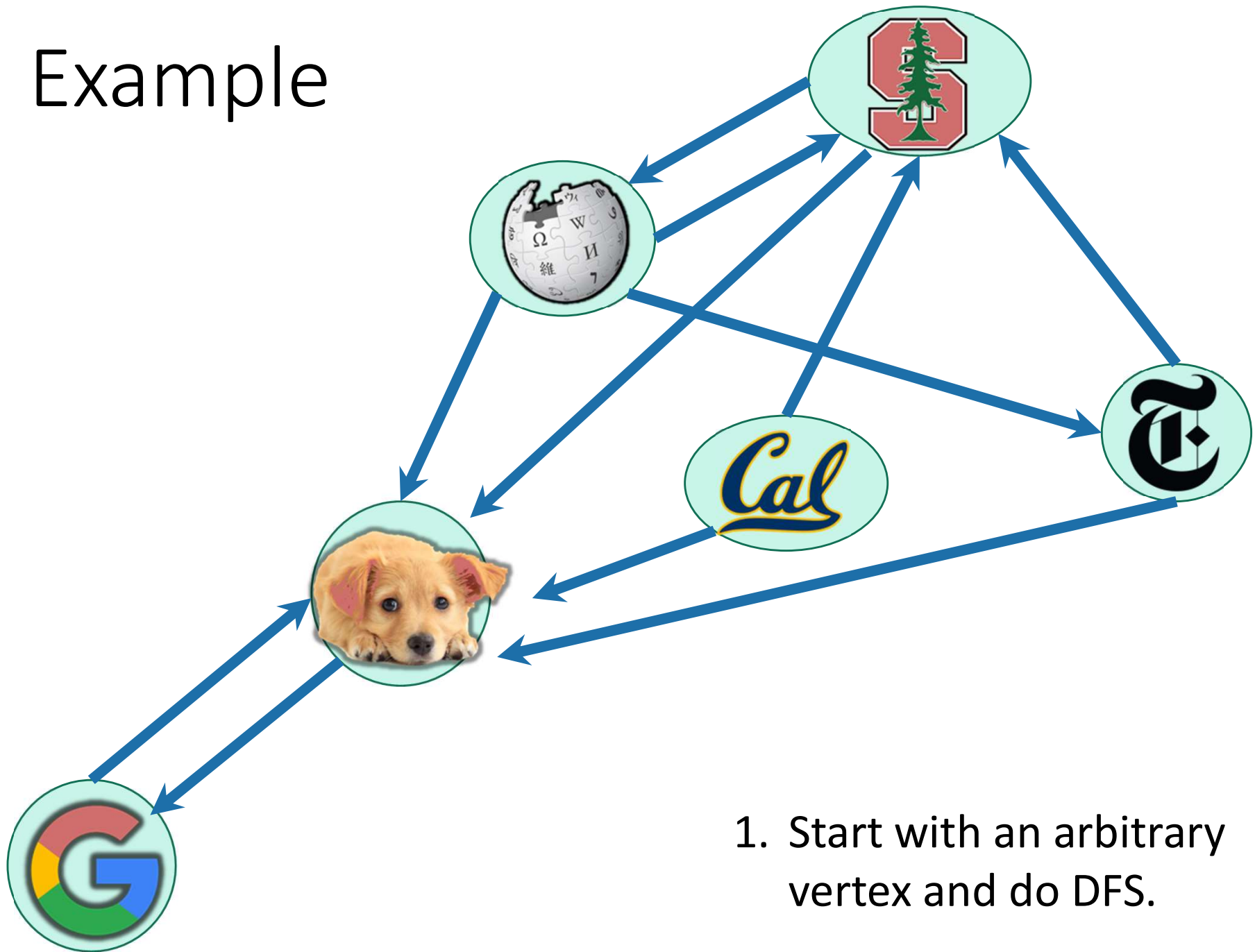
Running time: $O(n + m)$

- 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**.

Example

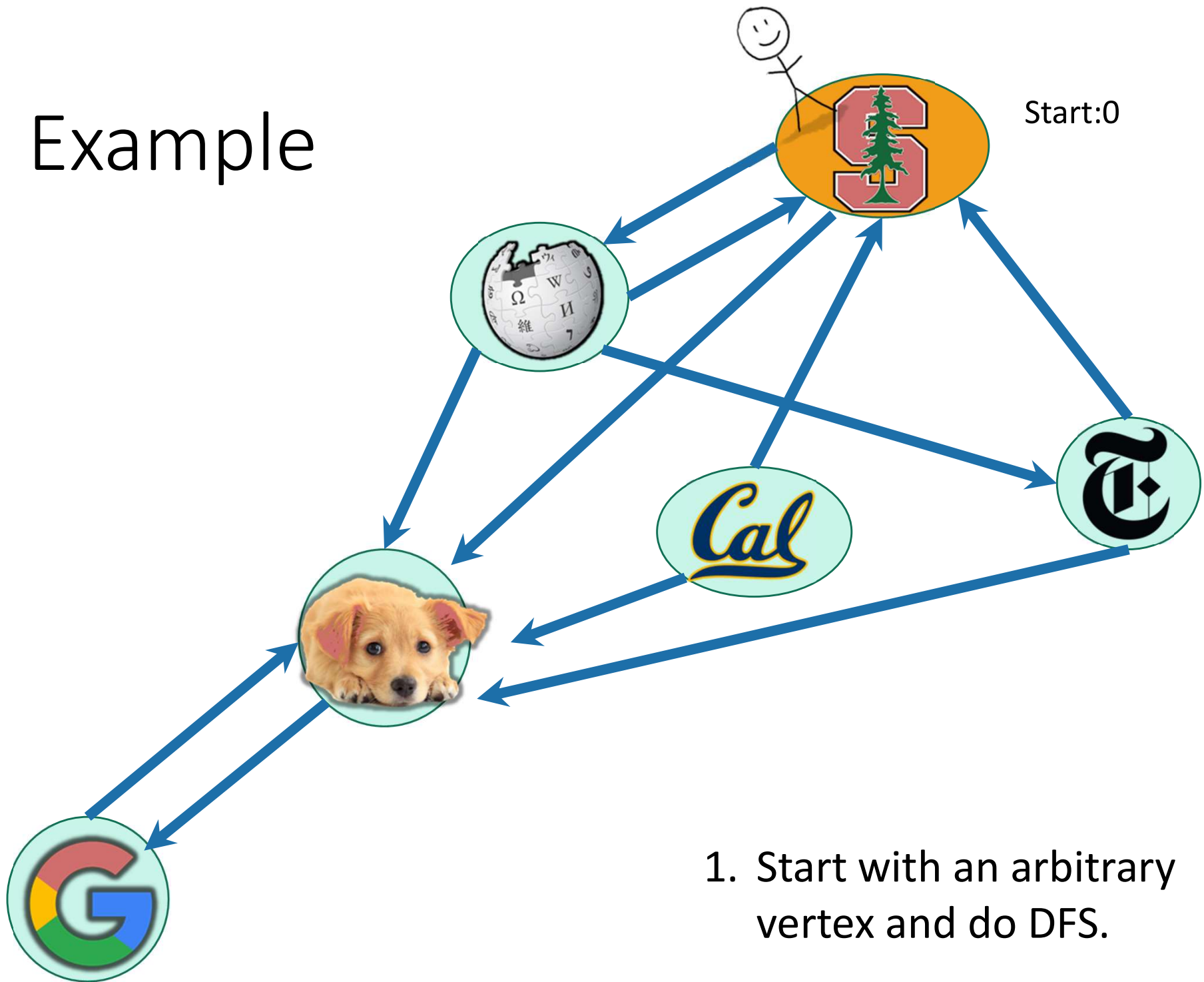


Example

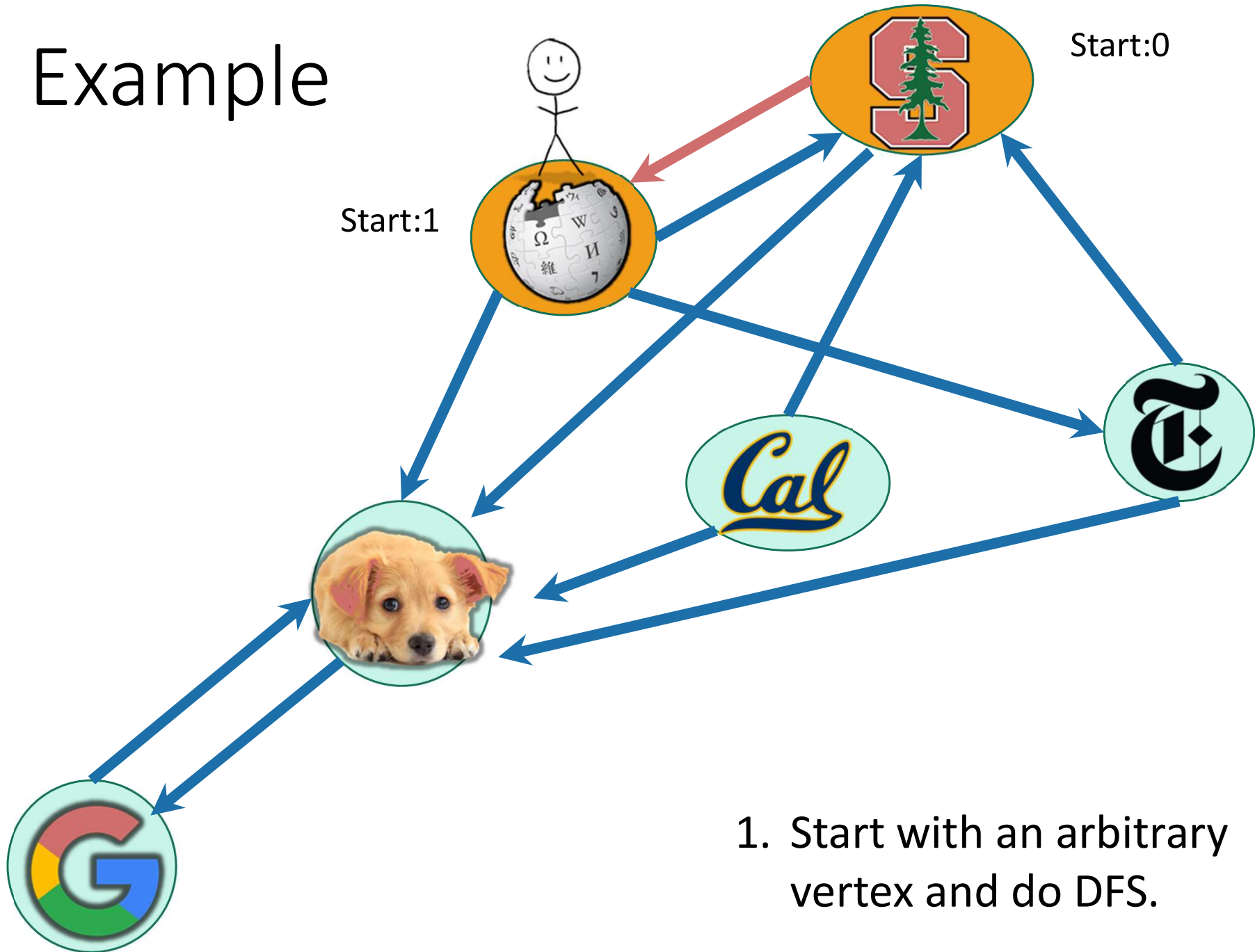


1. Start with an arbitrary vertex and do DFS.

Example

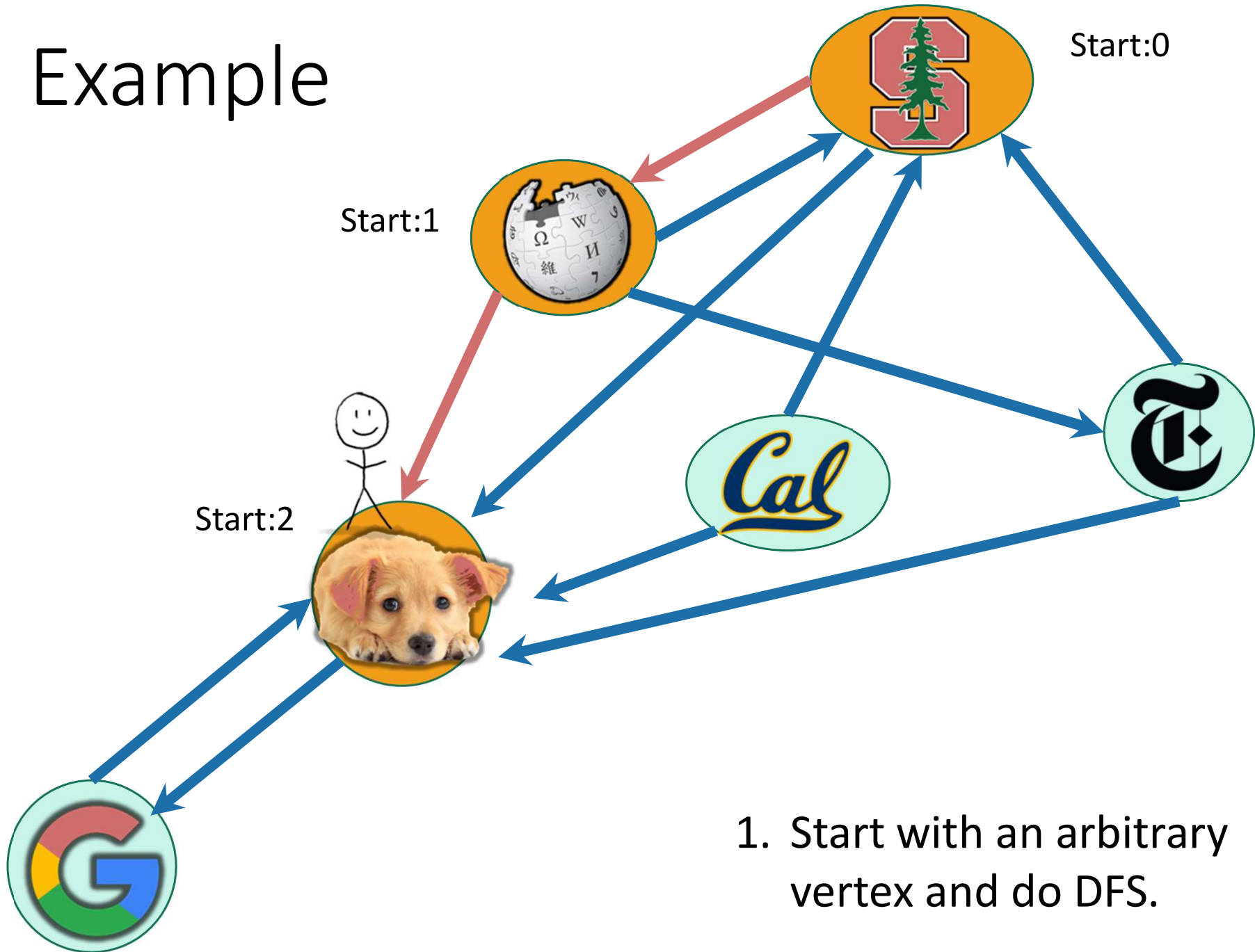


Example



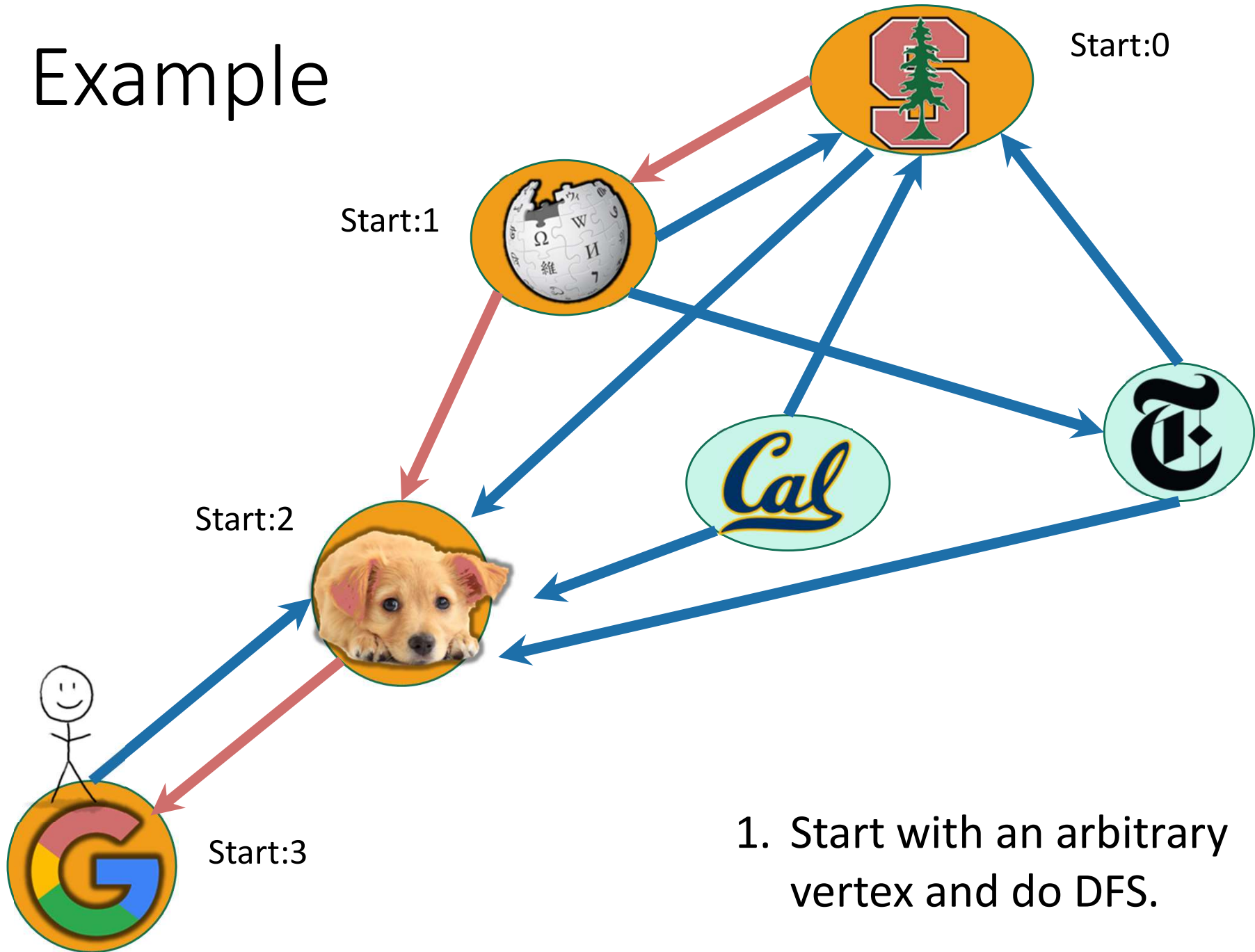
1. Start with an arbitrary vertex and do DFS.

Example



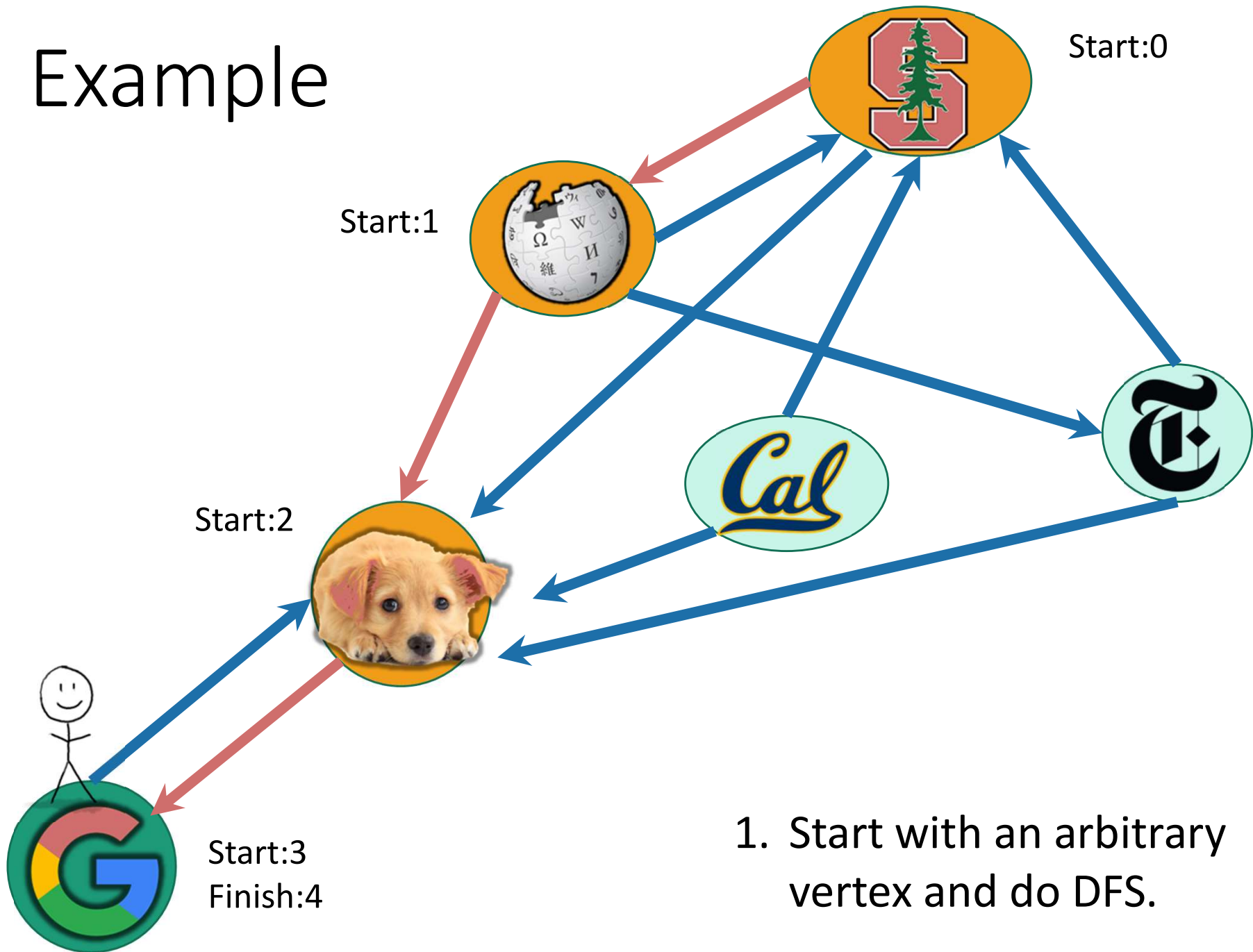
1. Start with an arbitrary vertex and do DFS.

Example



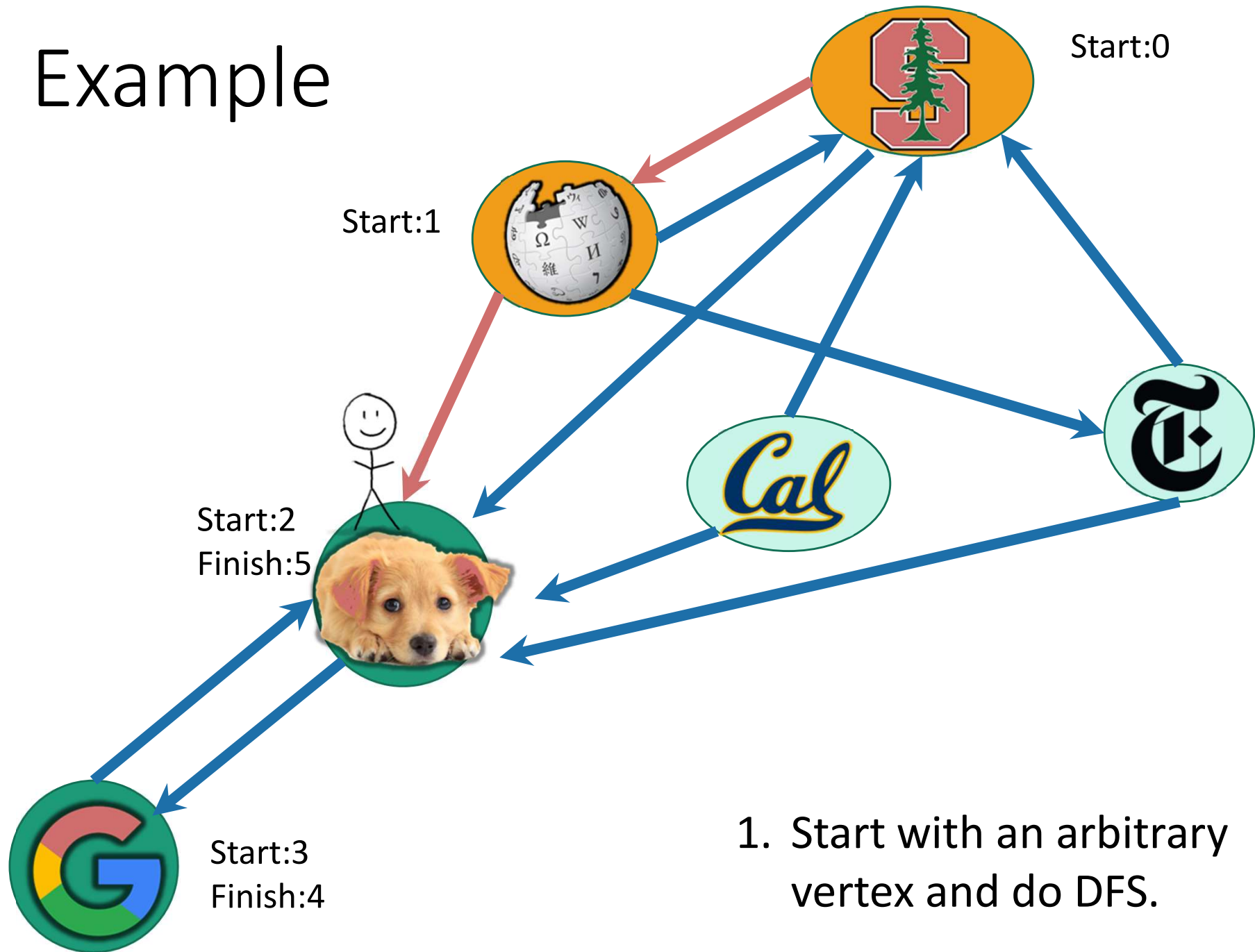
1. Start with an arbitrary vertex and do DFS.

Example



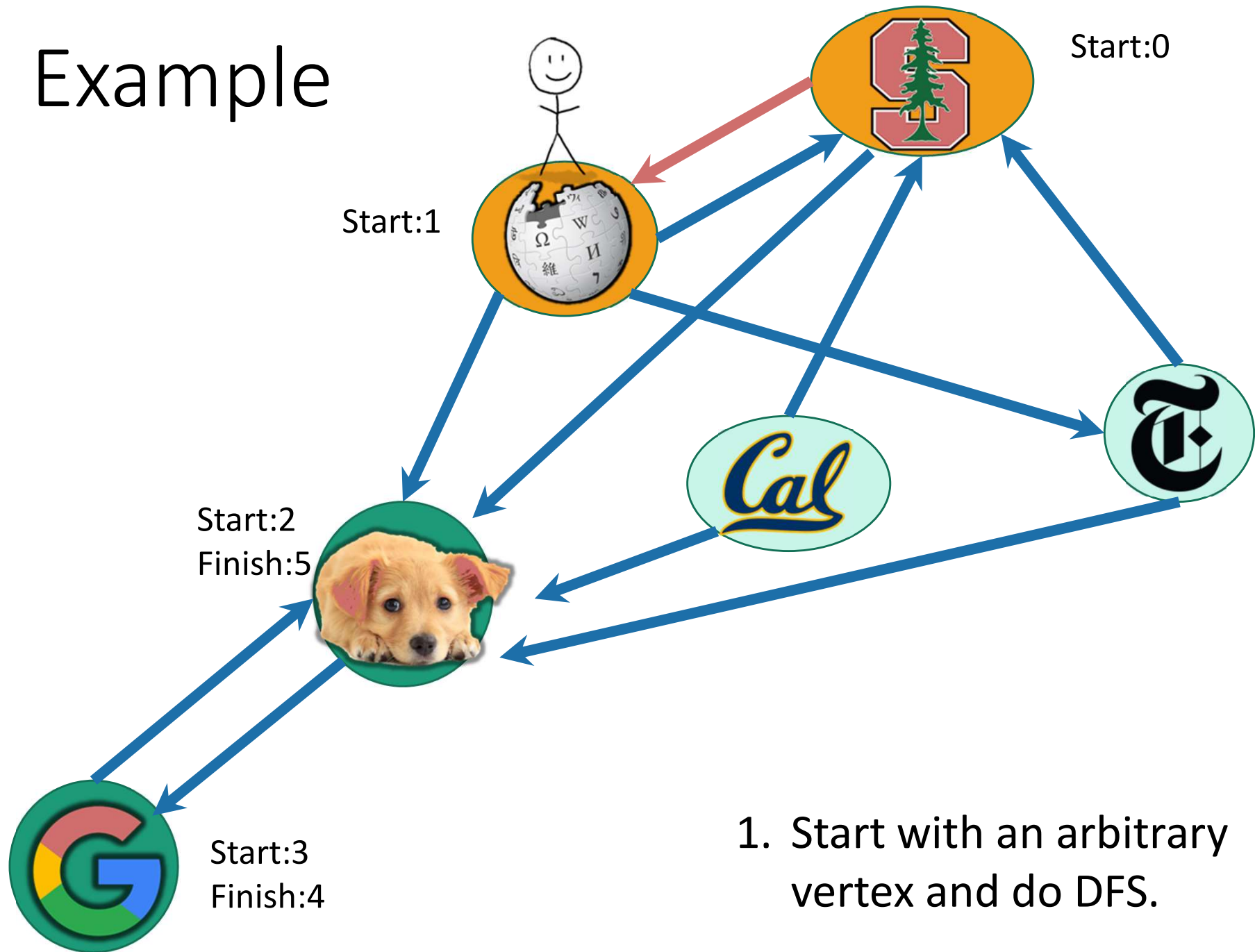
1. Start with an arbitrary vertex and do DFS.

Example



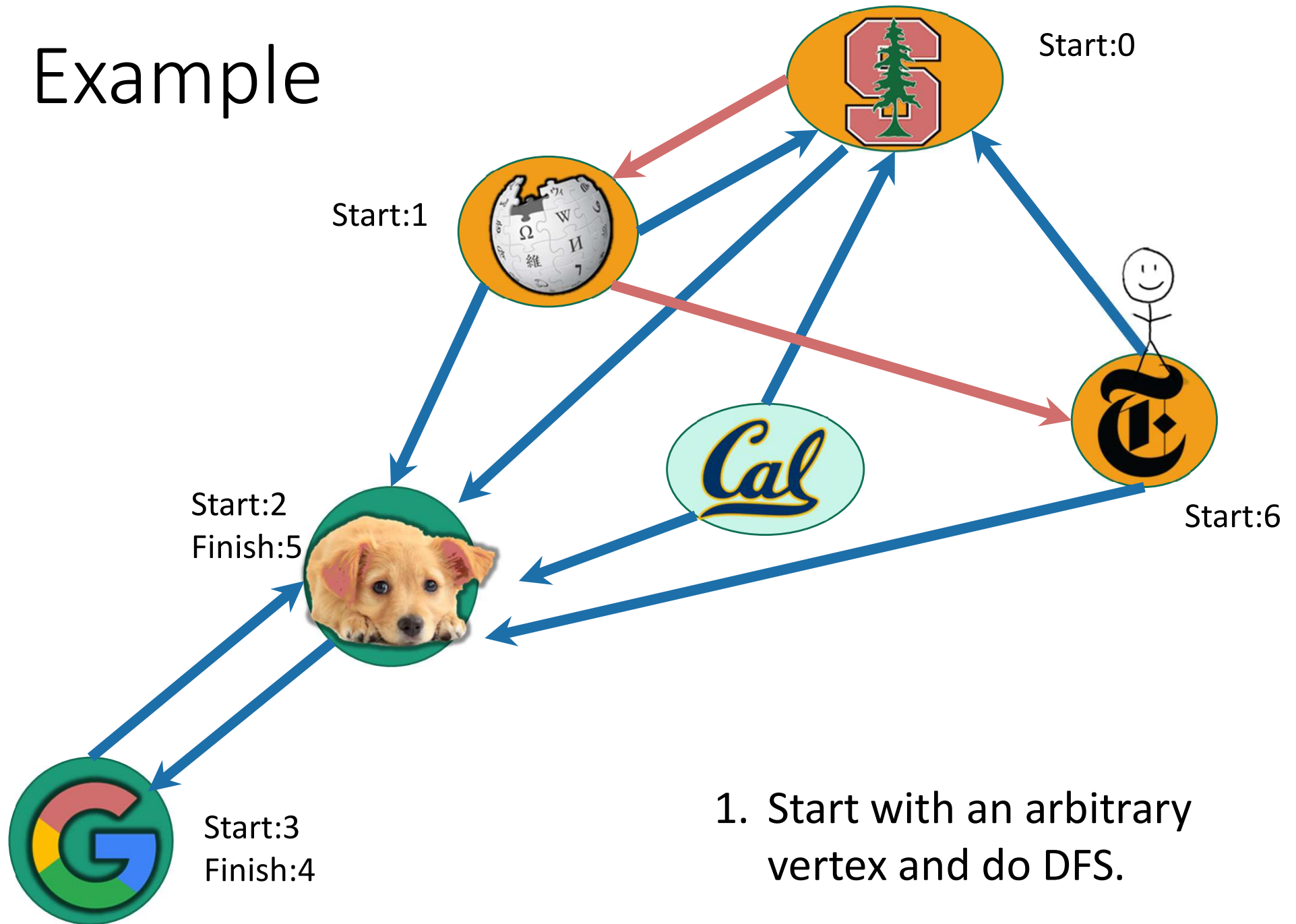
1. Start with an arbitrary vertex and do DFS.

Example

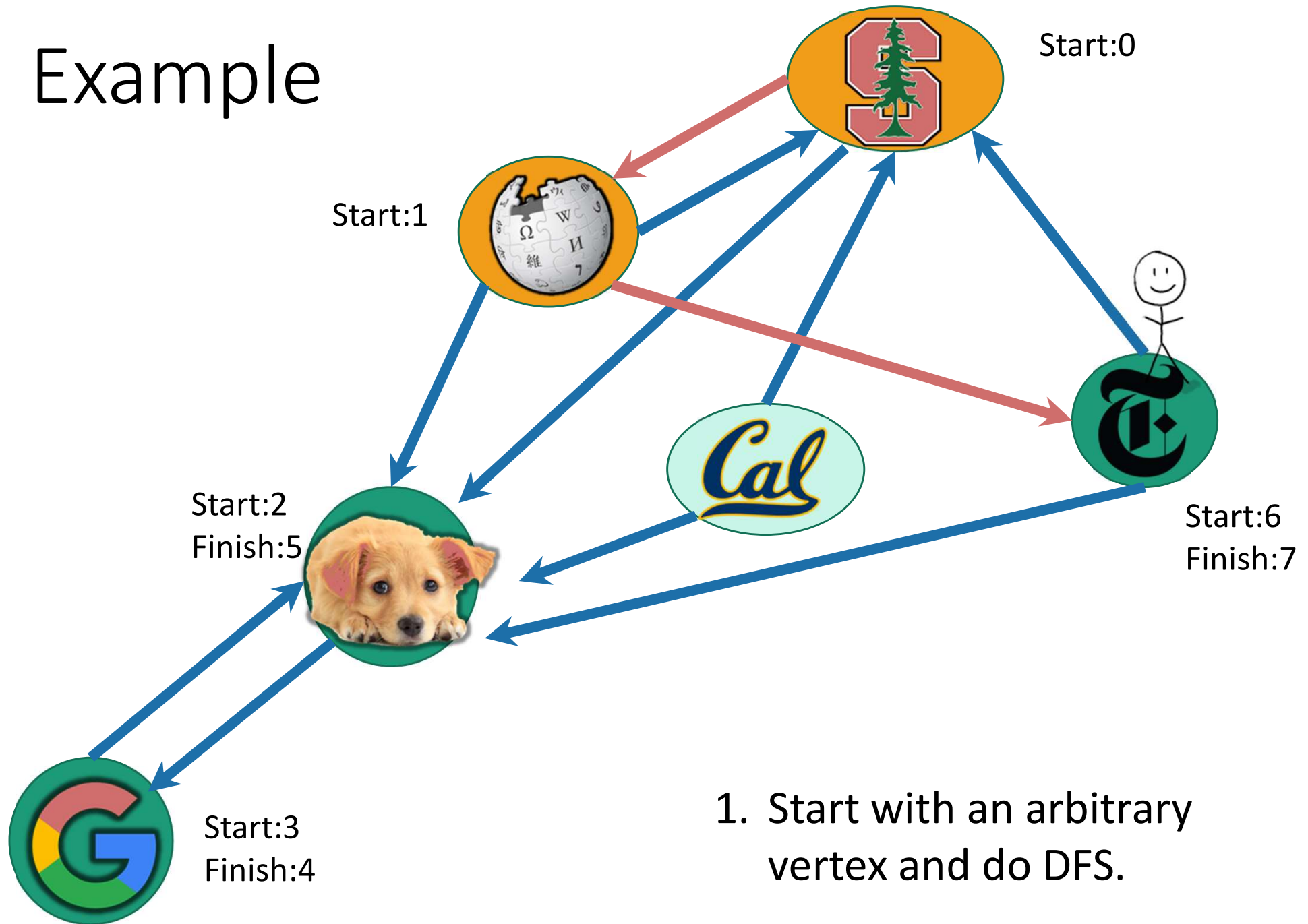


1. Start with an arbitrary vertex and do DFS.

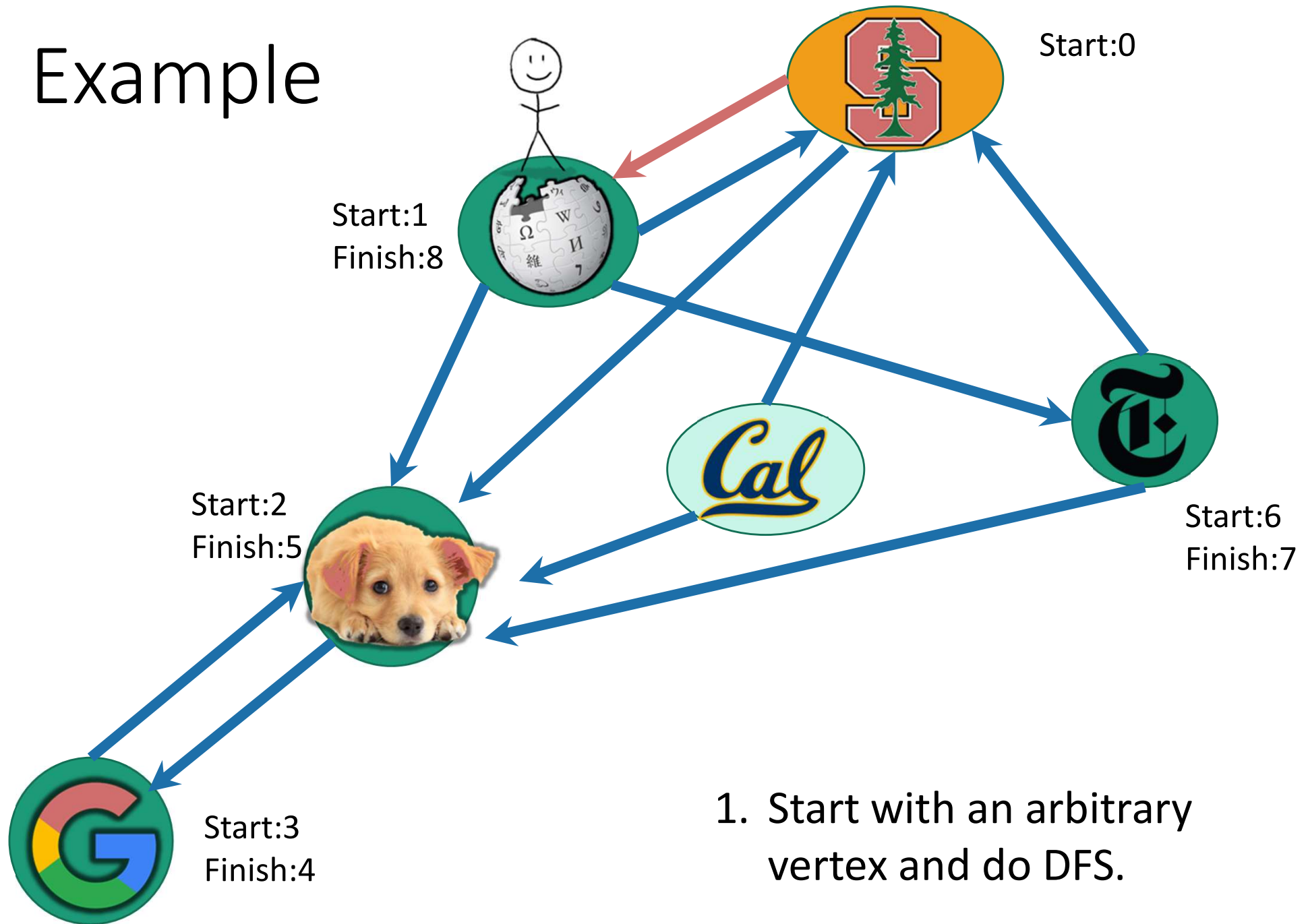
Example



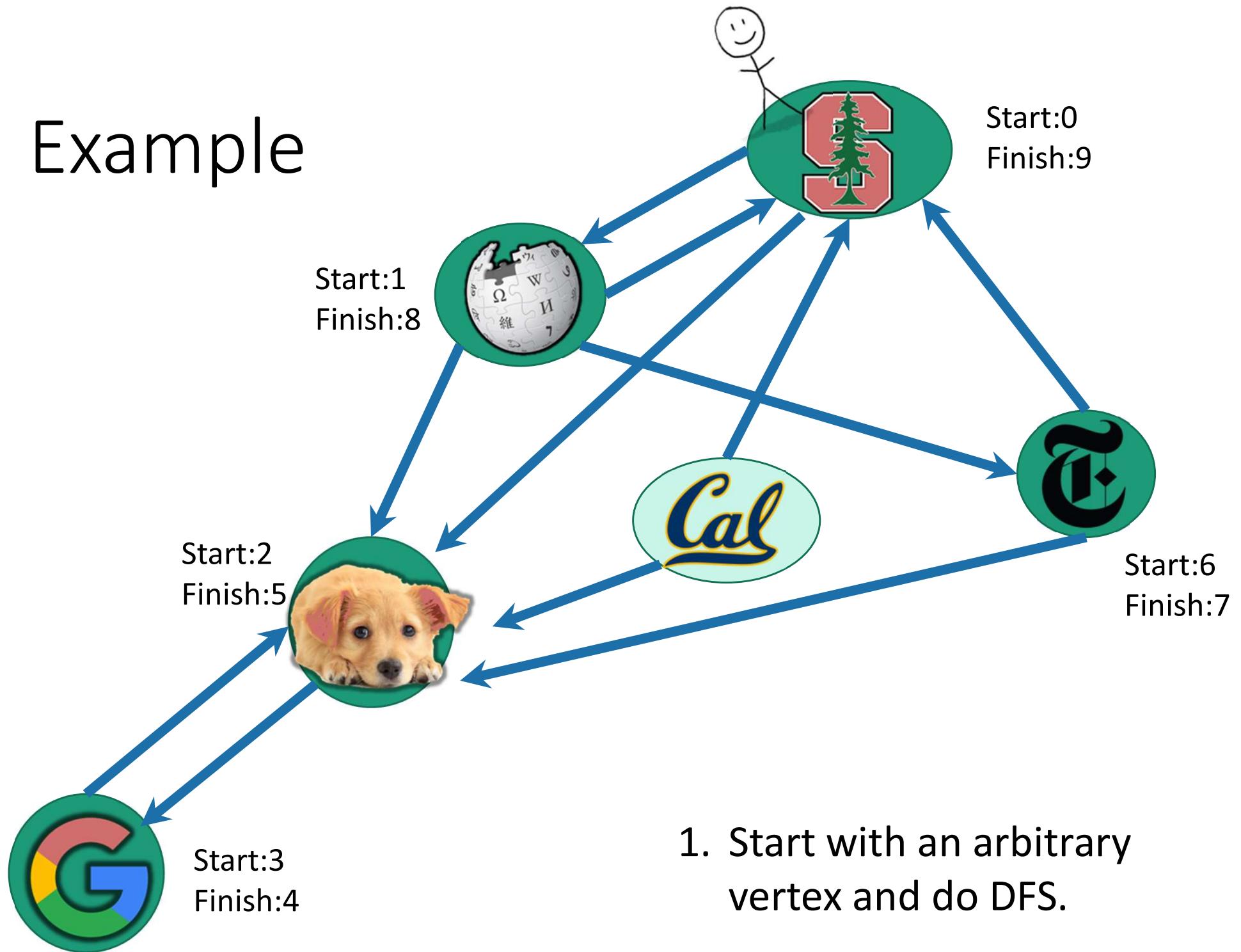
Example



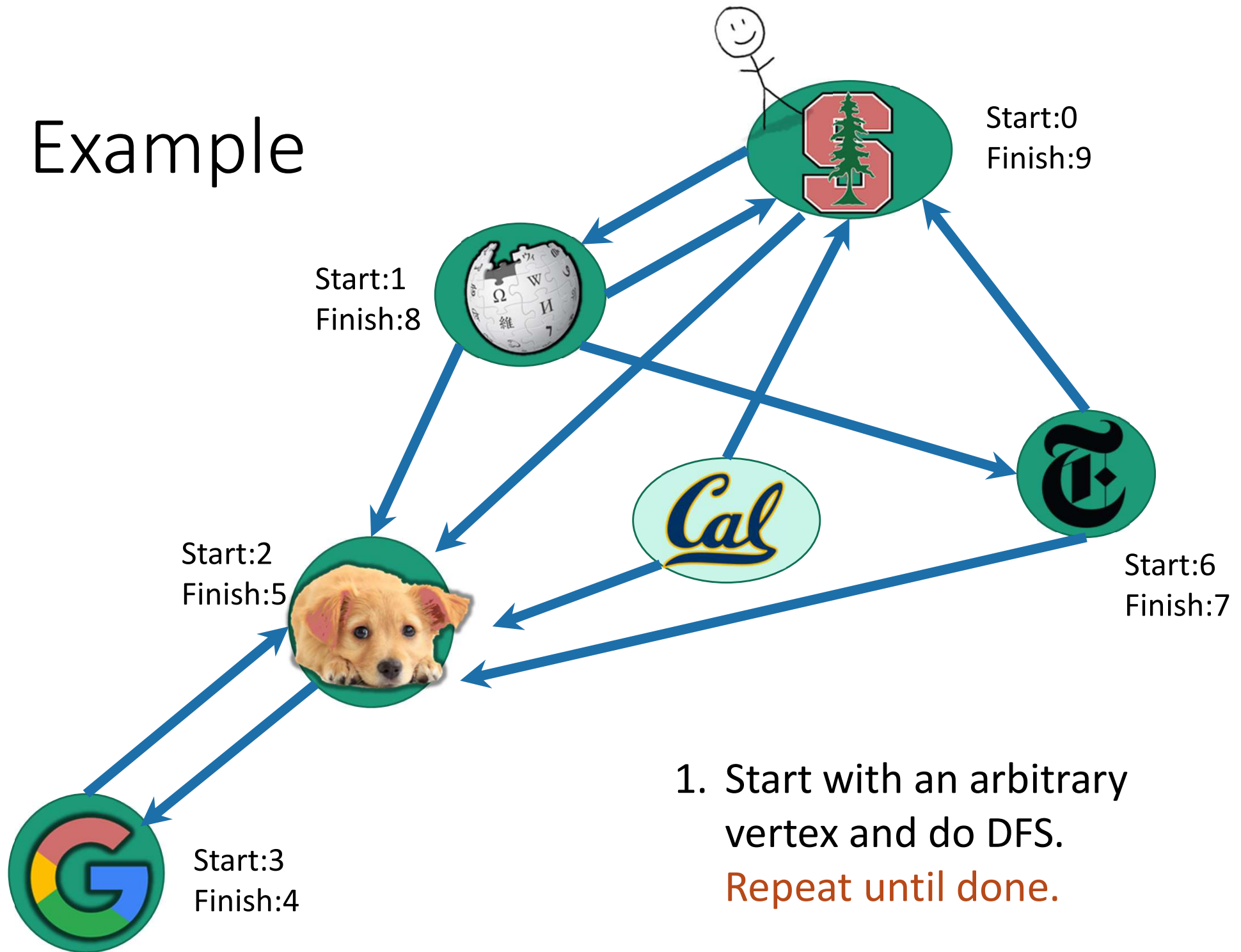
Example



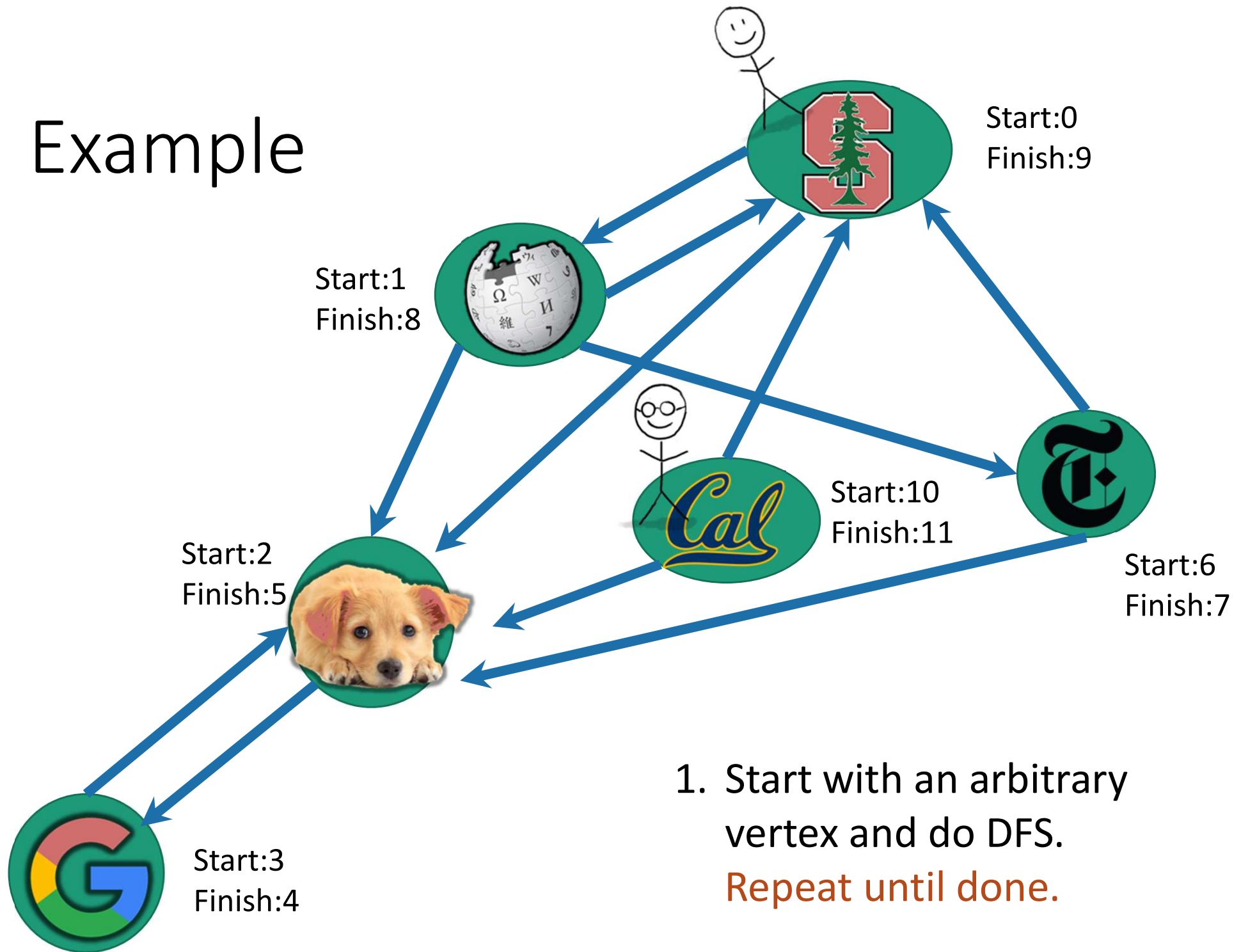
Example



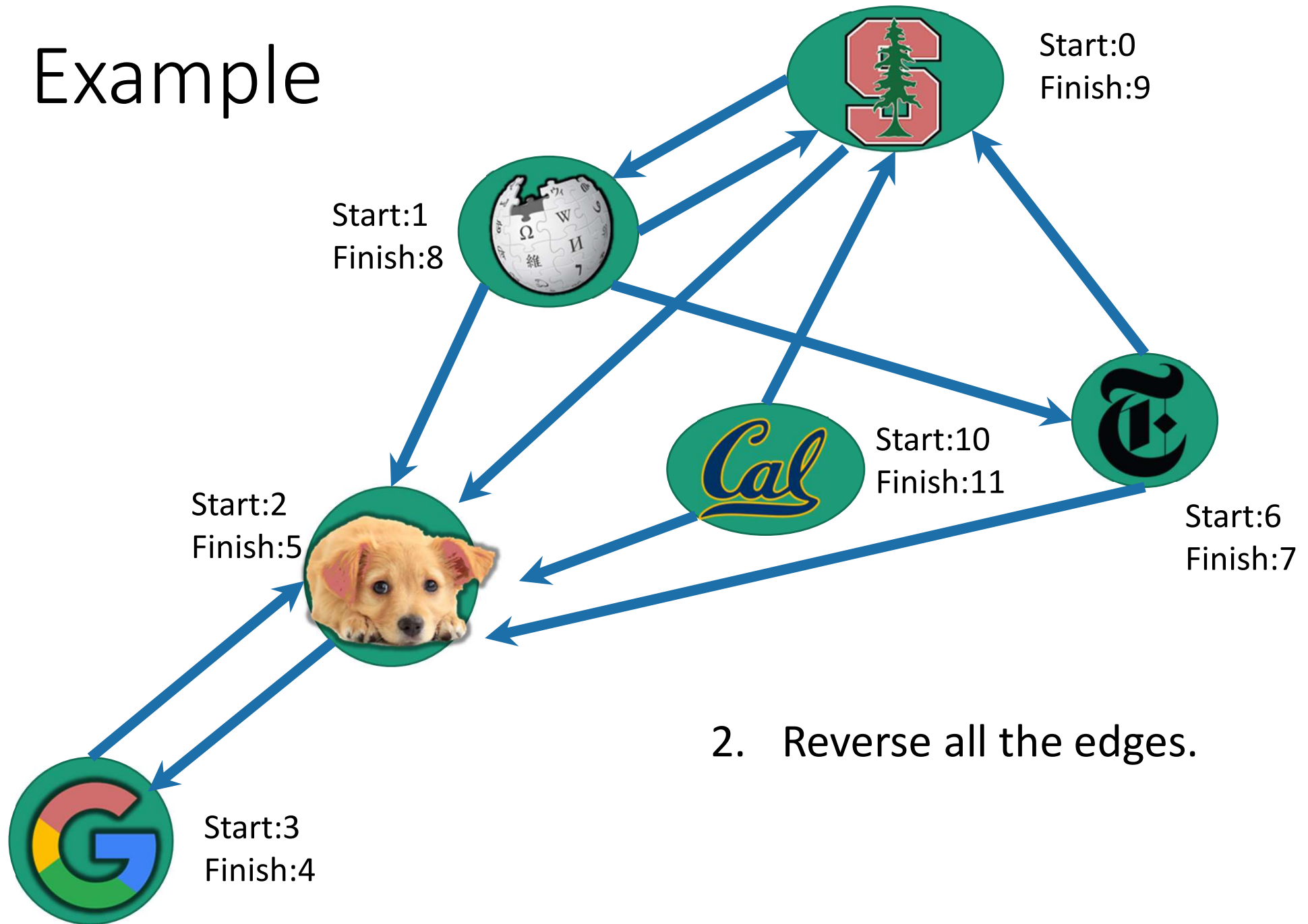
Example



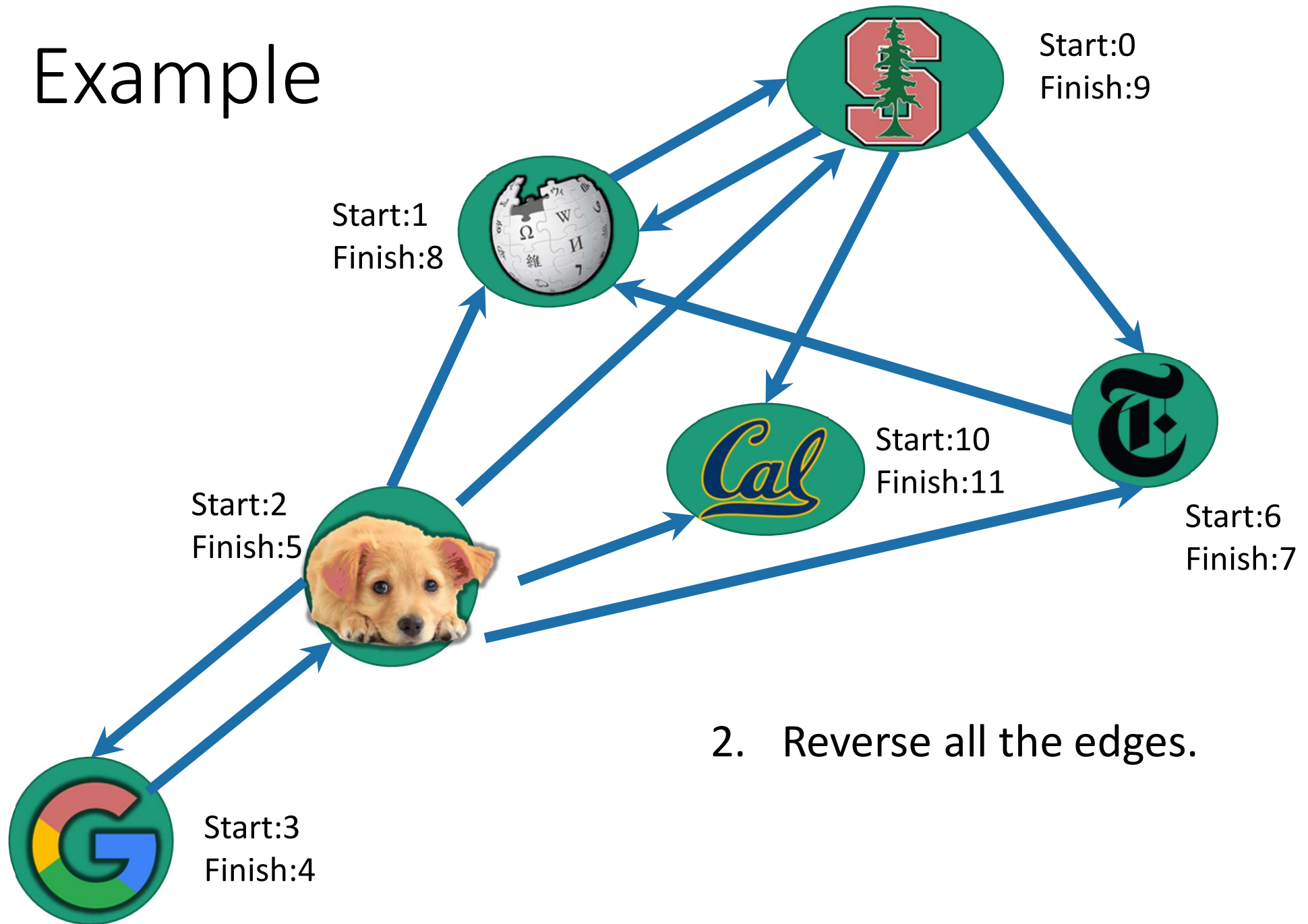
Example



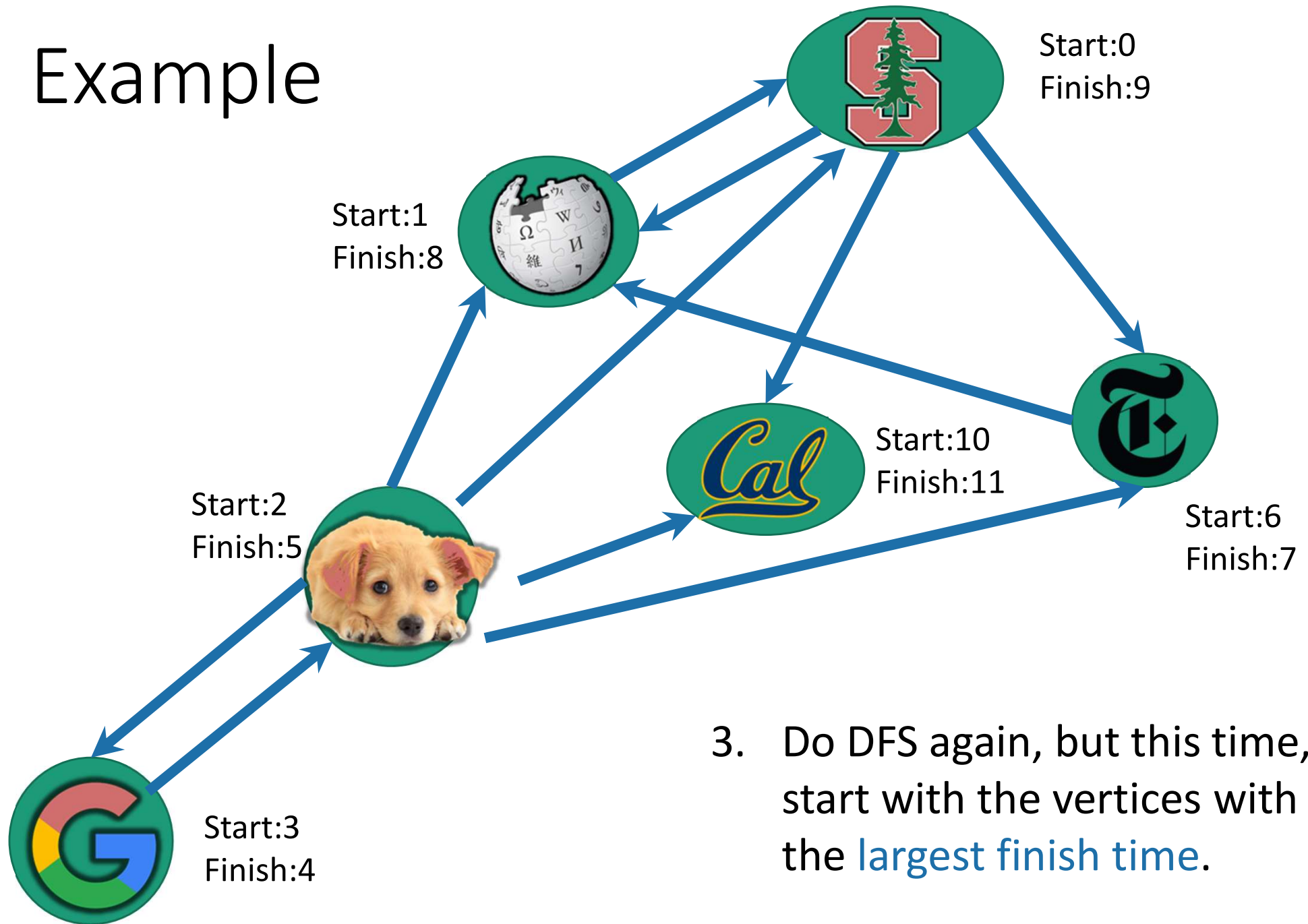
Example



Example

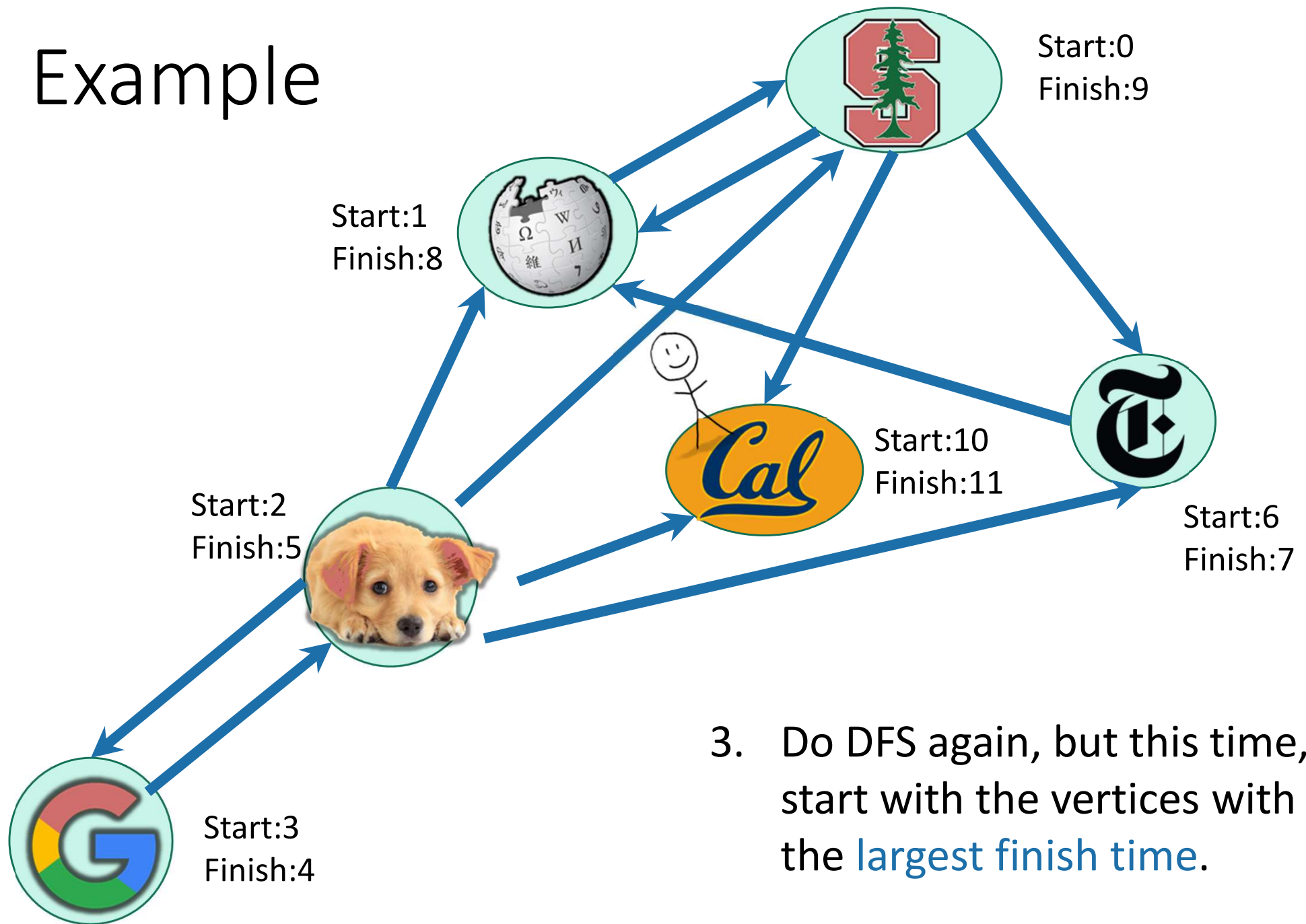


Example



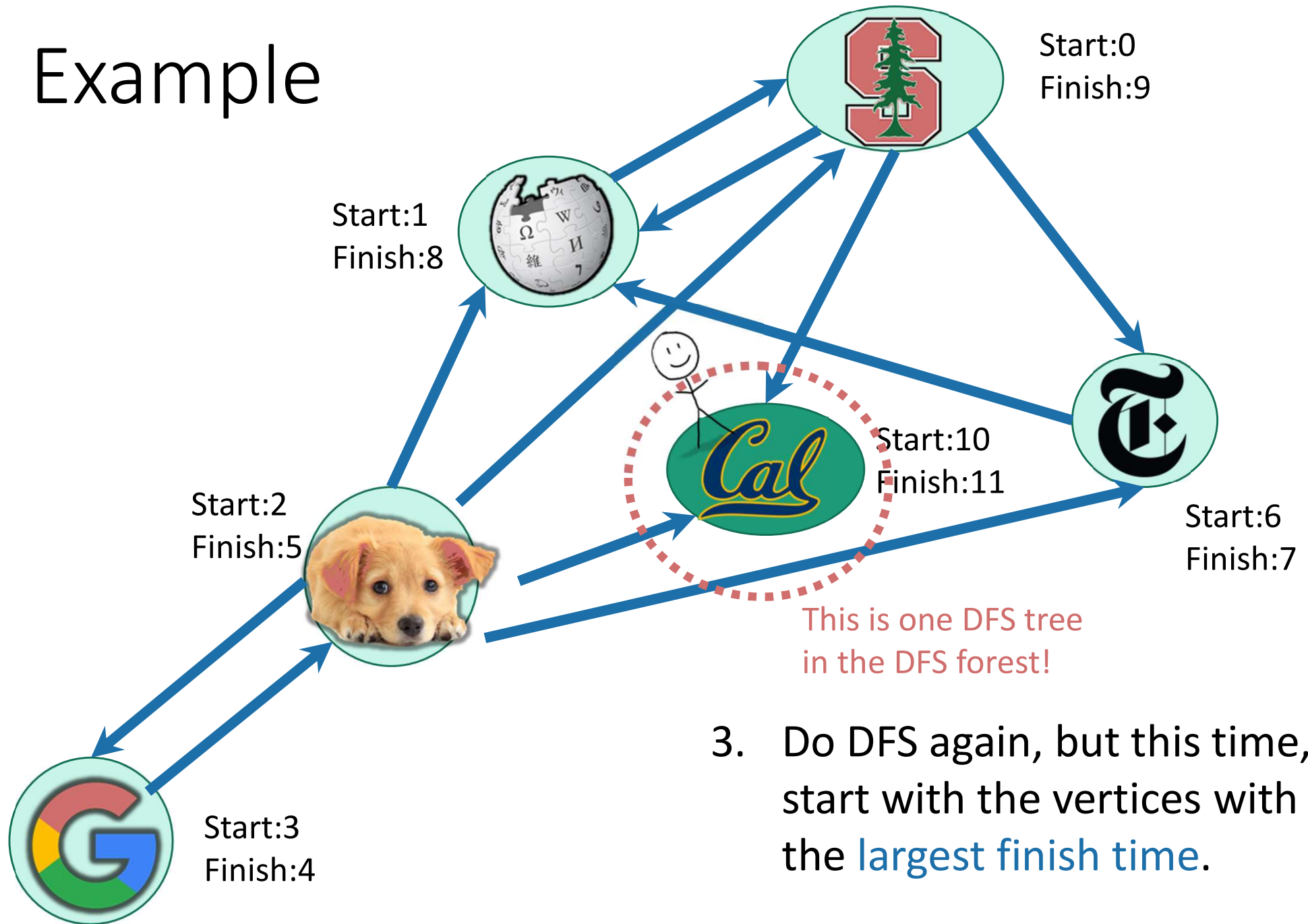
3. Do DFS again, but this time, start with the vertices with the **largest finish time**.

Example

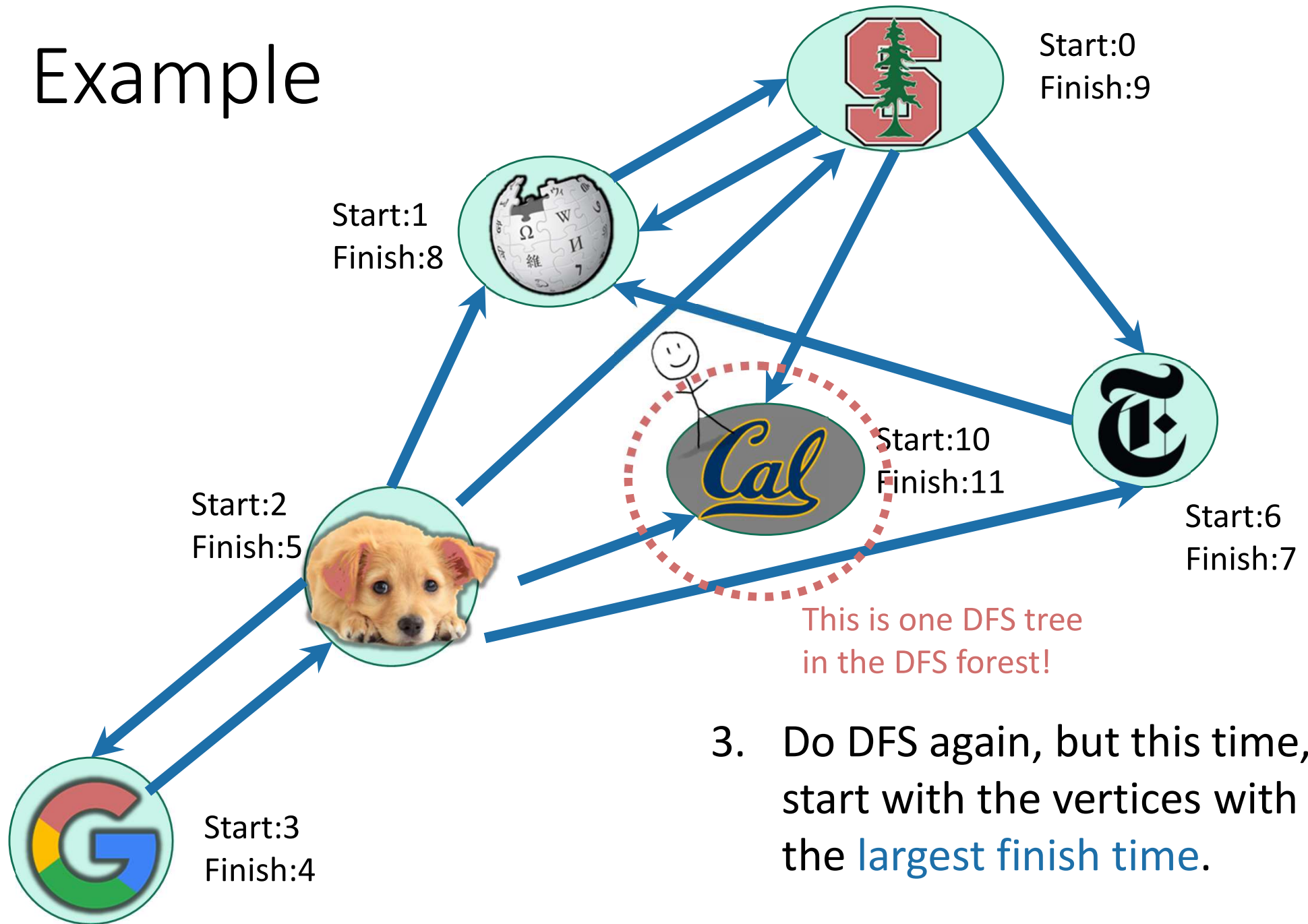


3. Do DFS again, but this time, start with the vertices with the **largest finish time**.

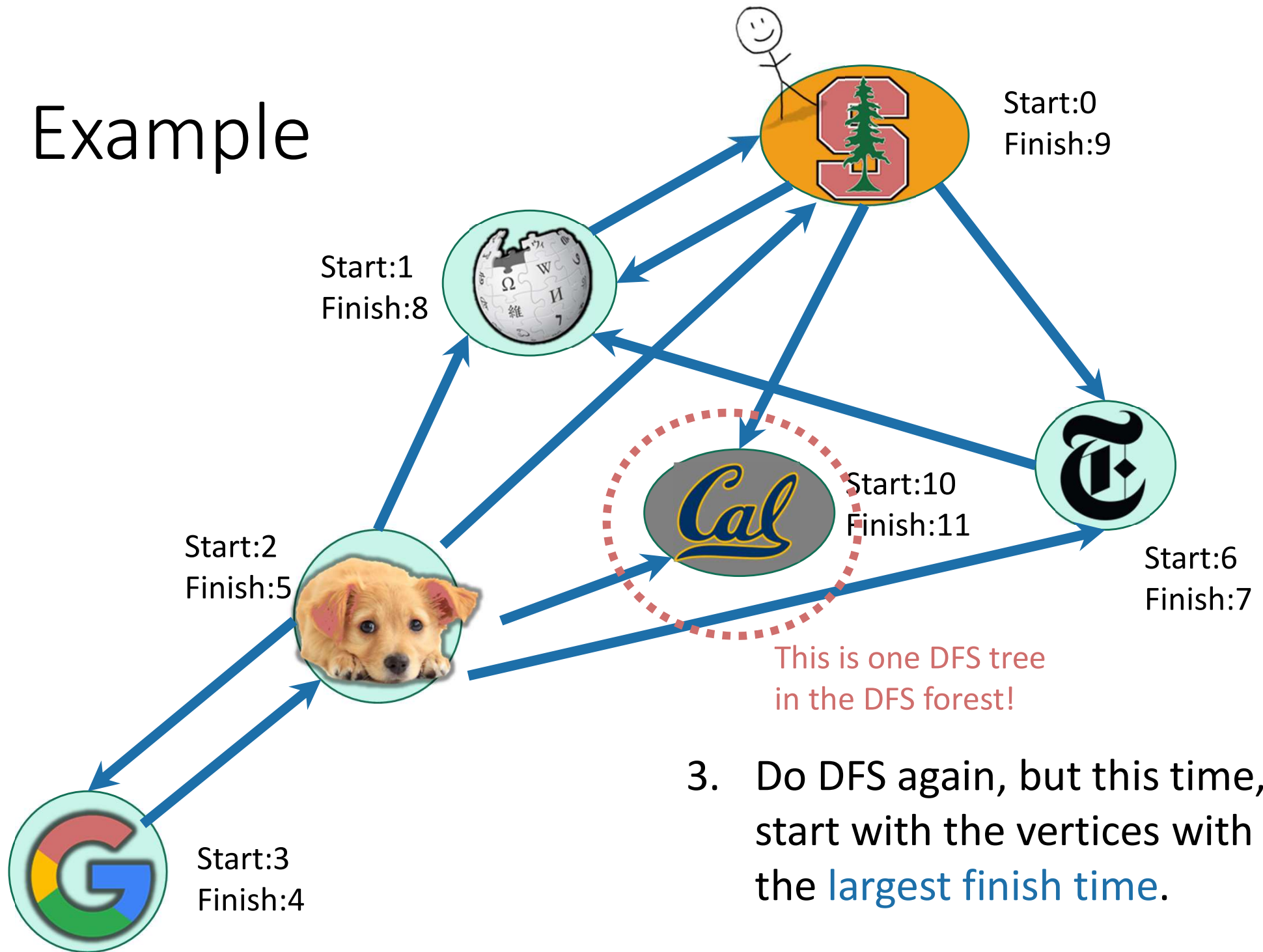
Example



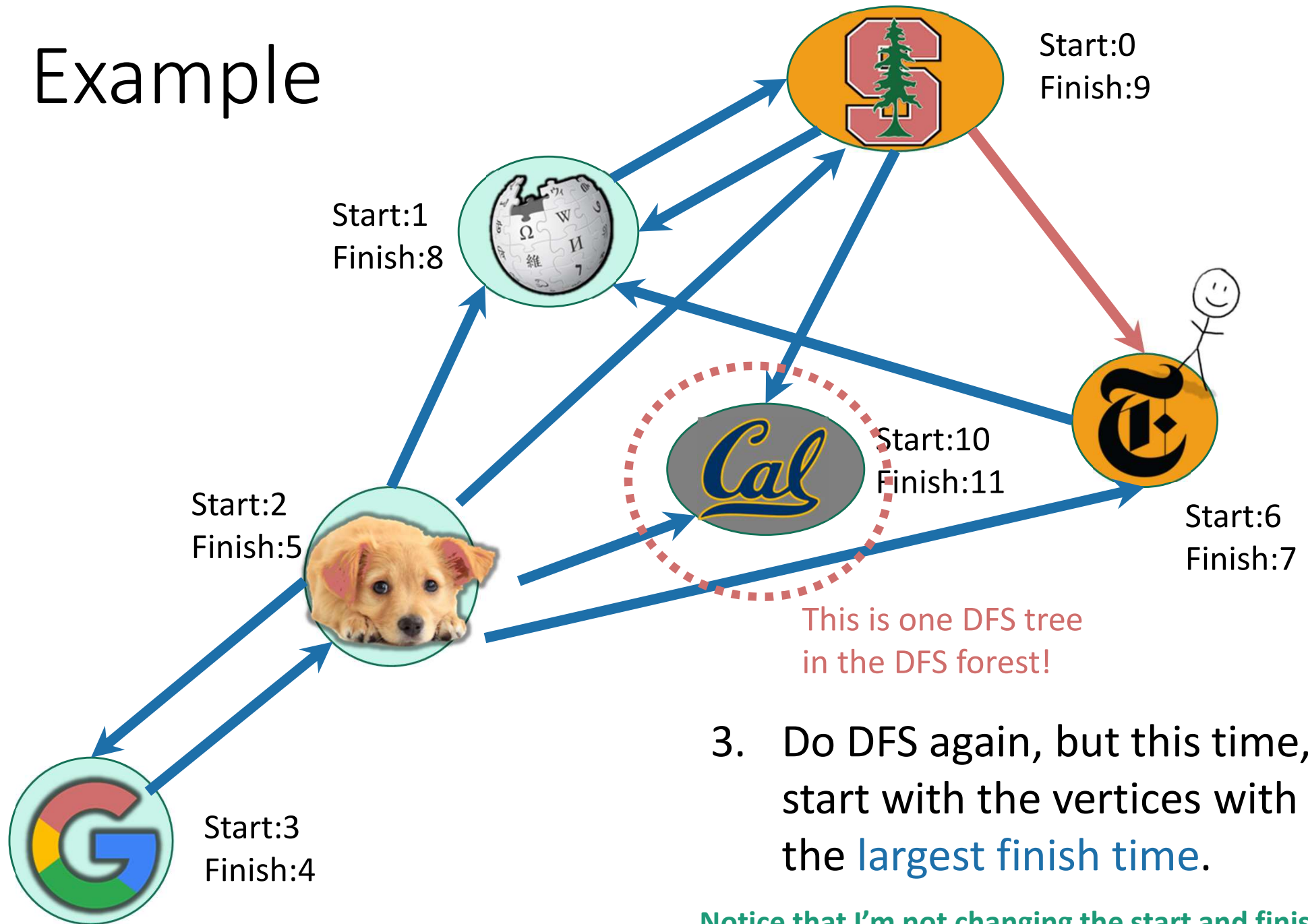
Example



Example



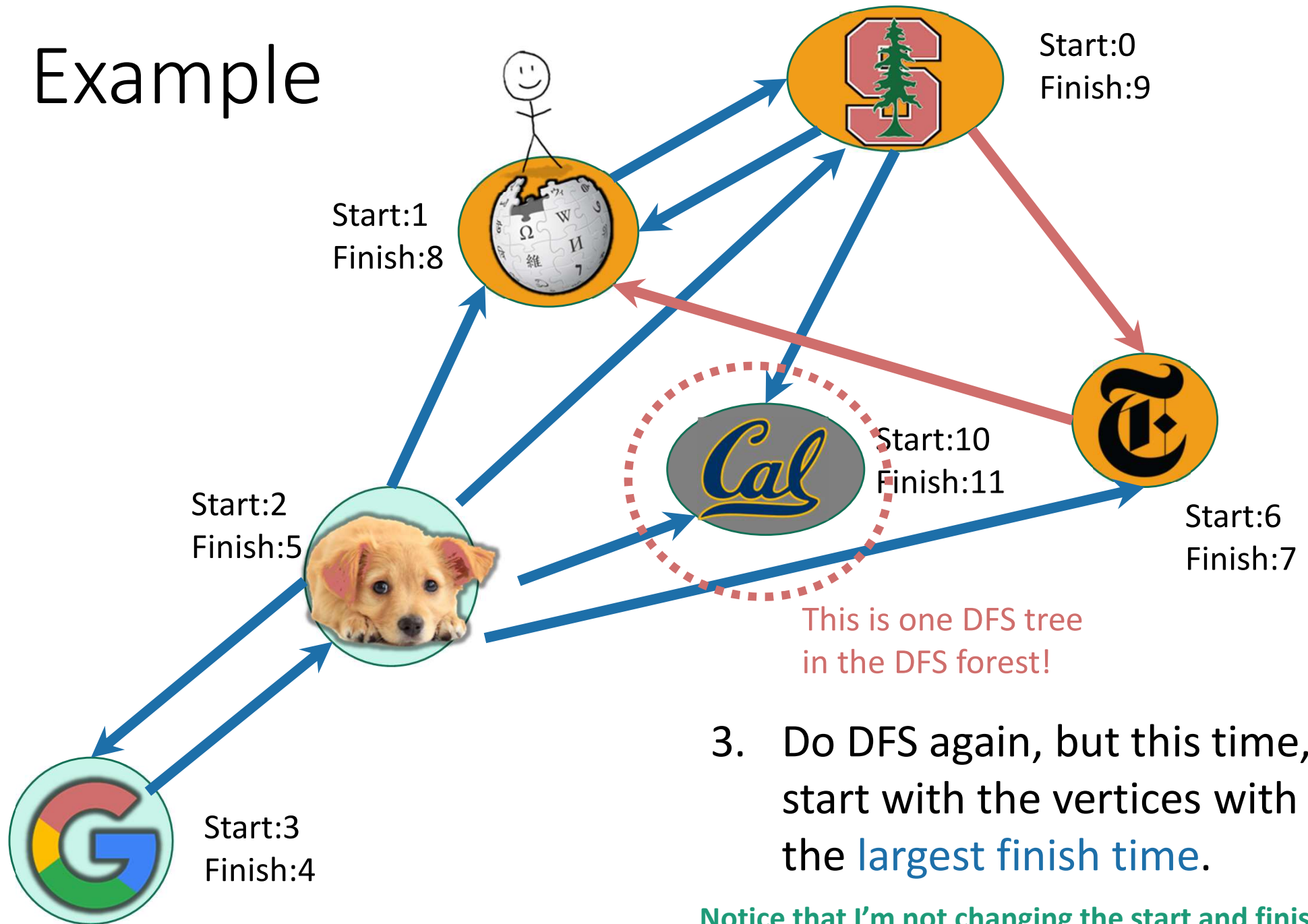
Example



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.

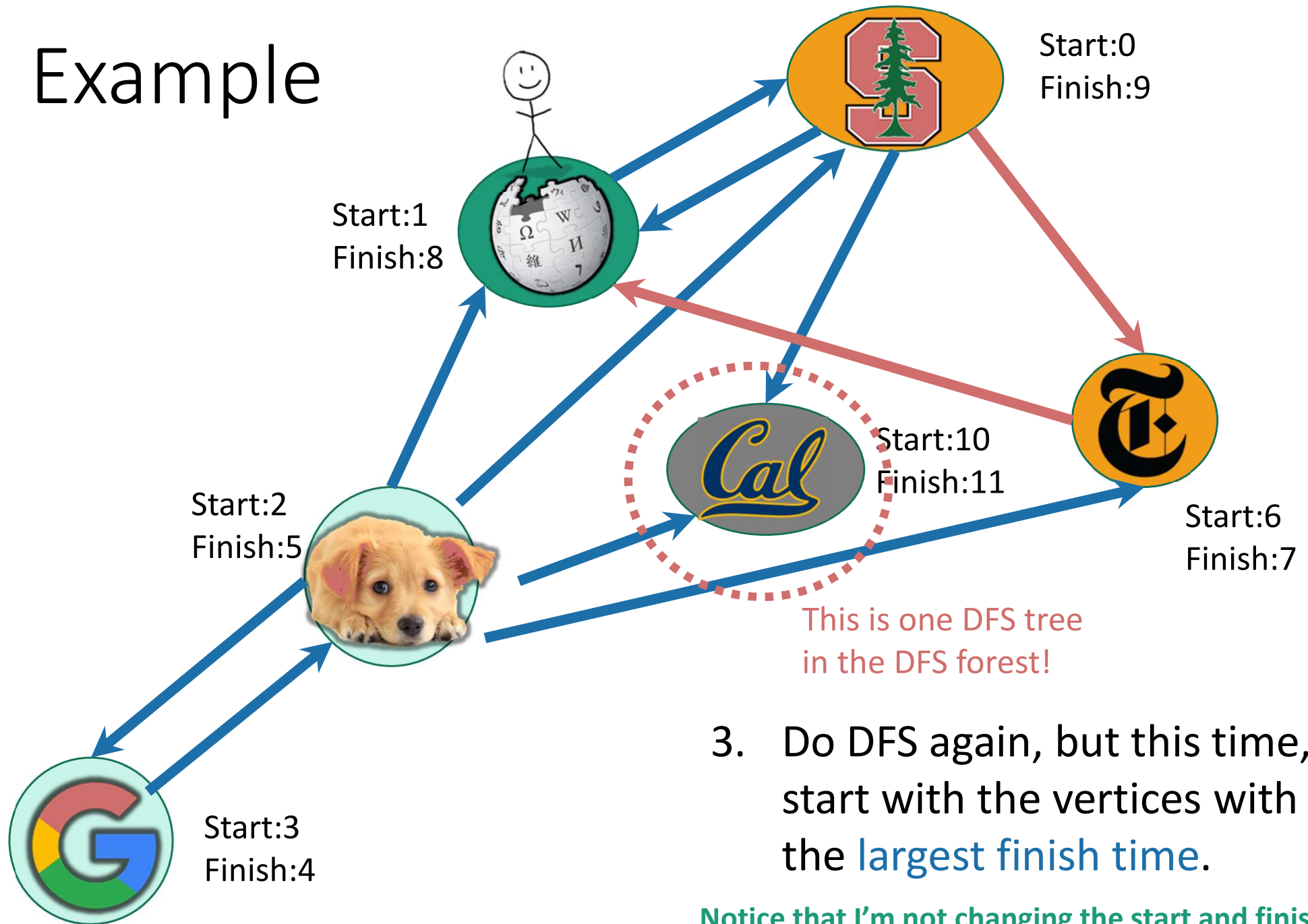
Example



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.

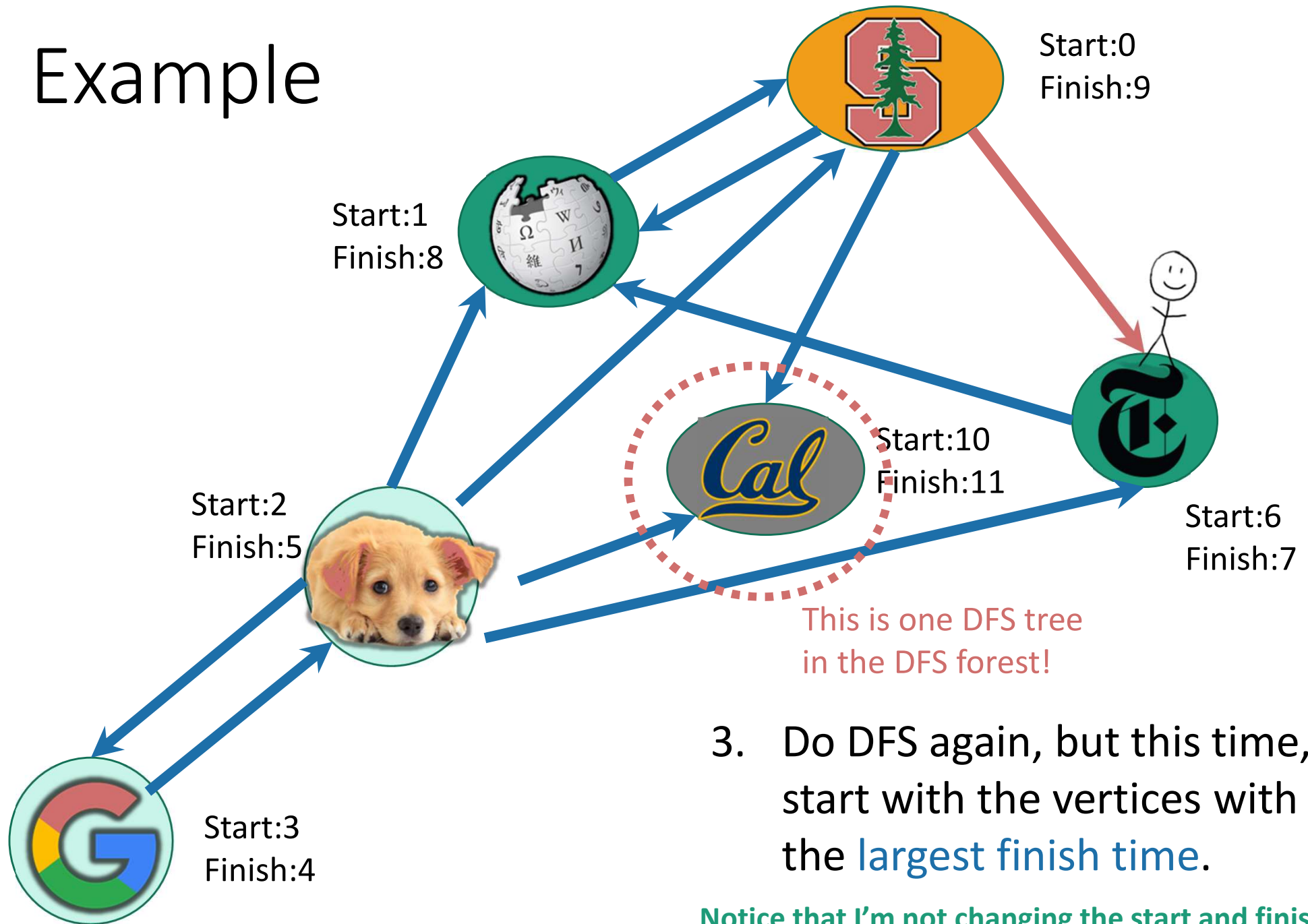
Example



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.

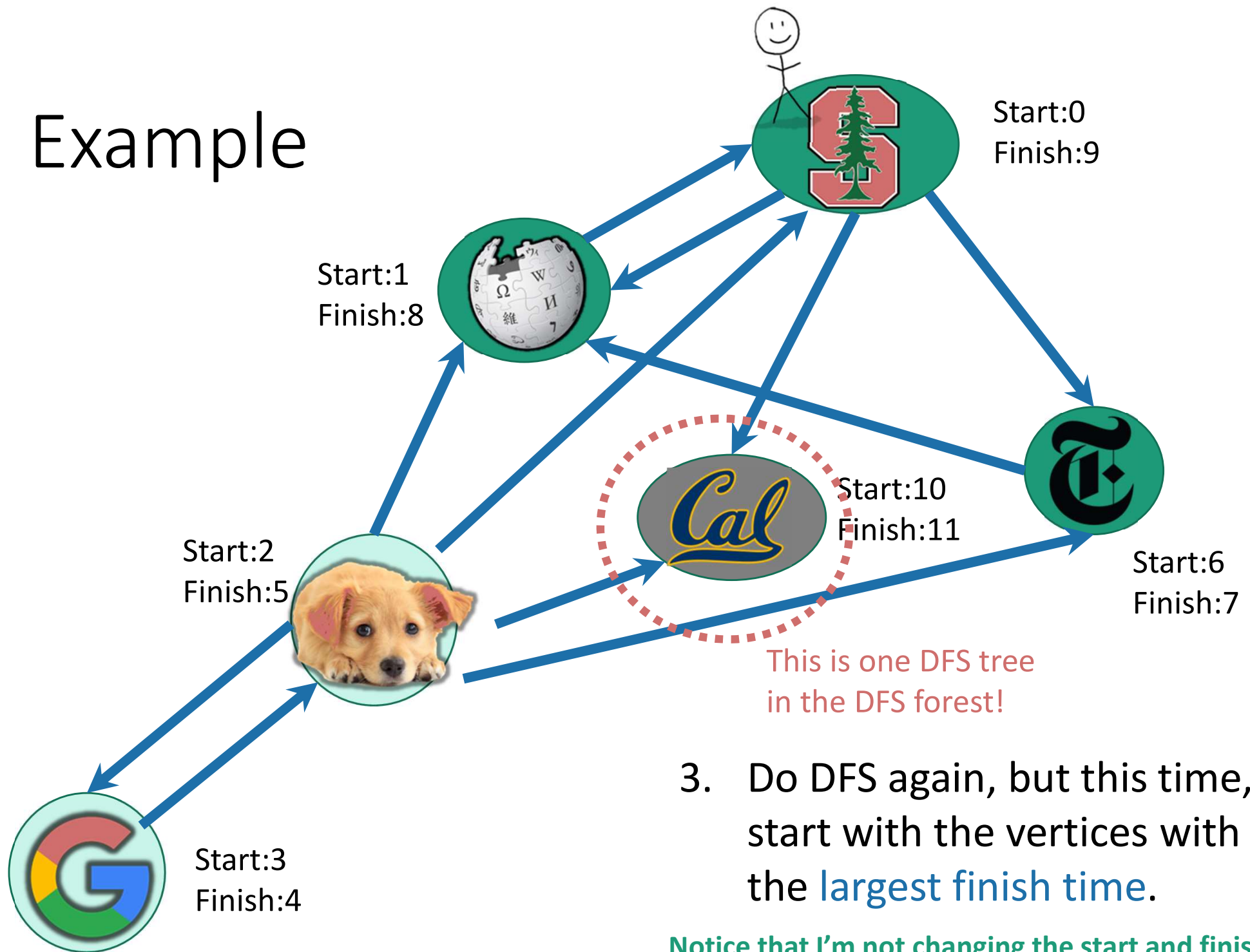
Example



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.

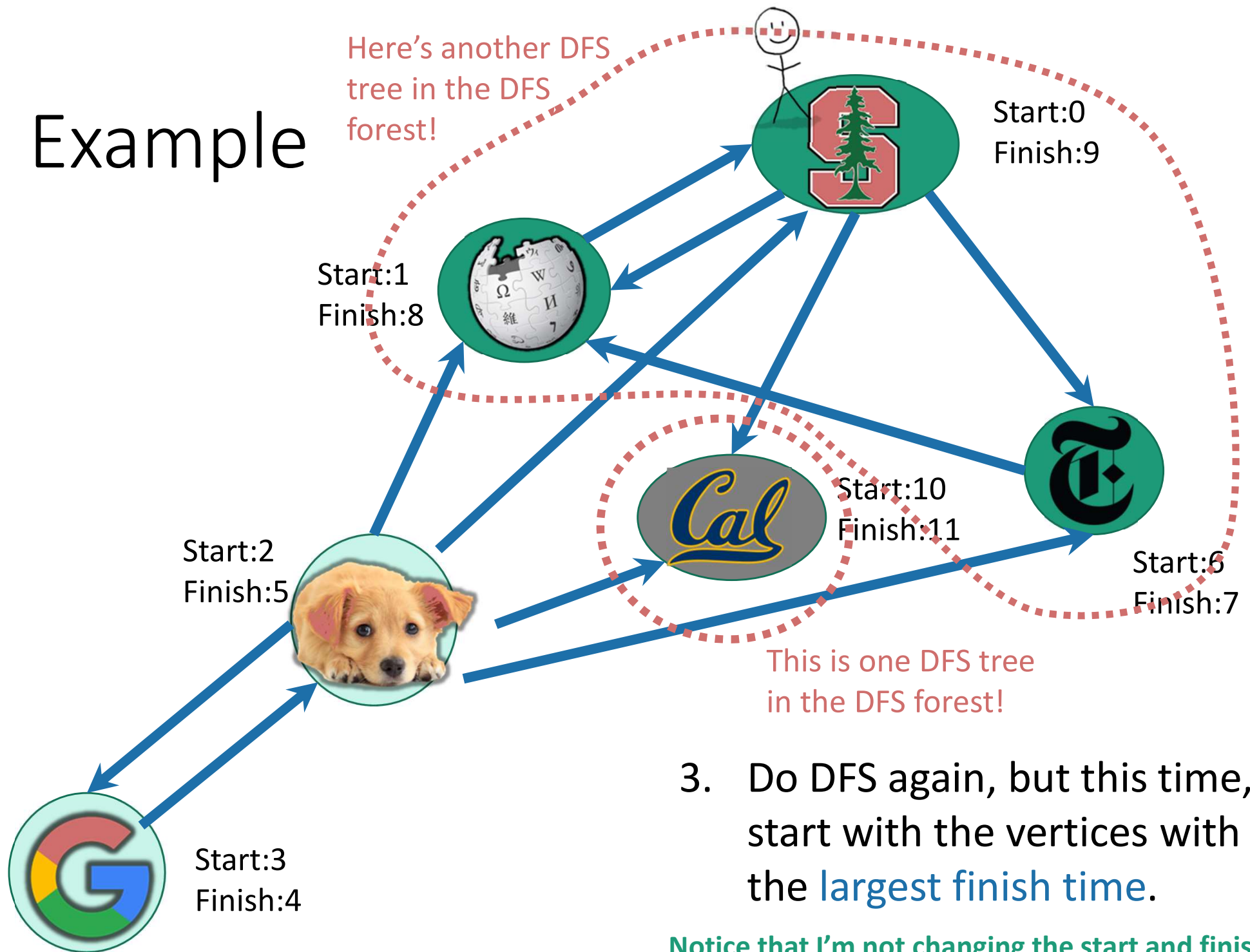
Example



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.

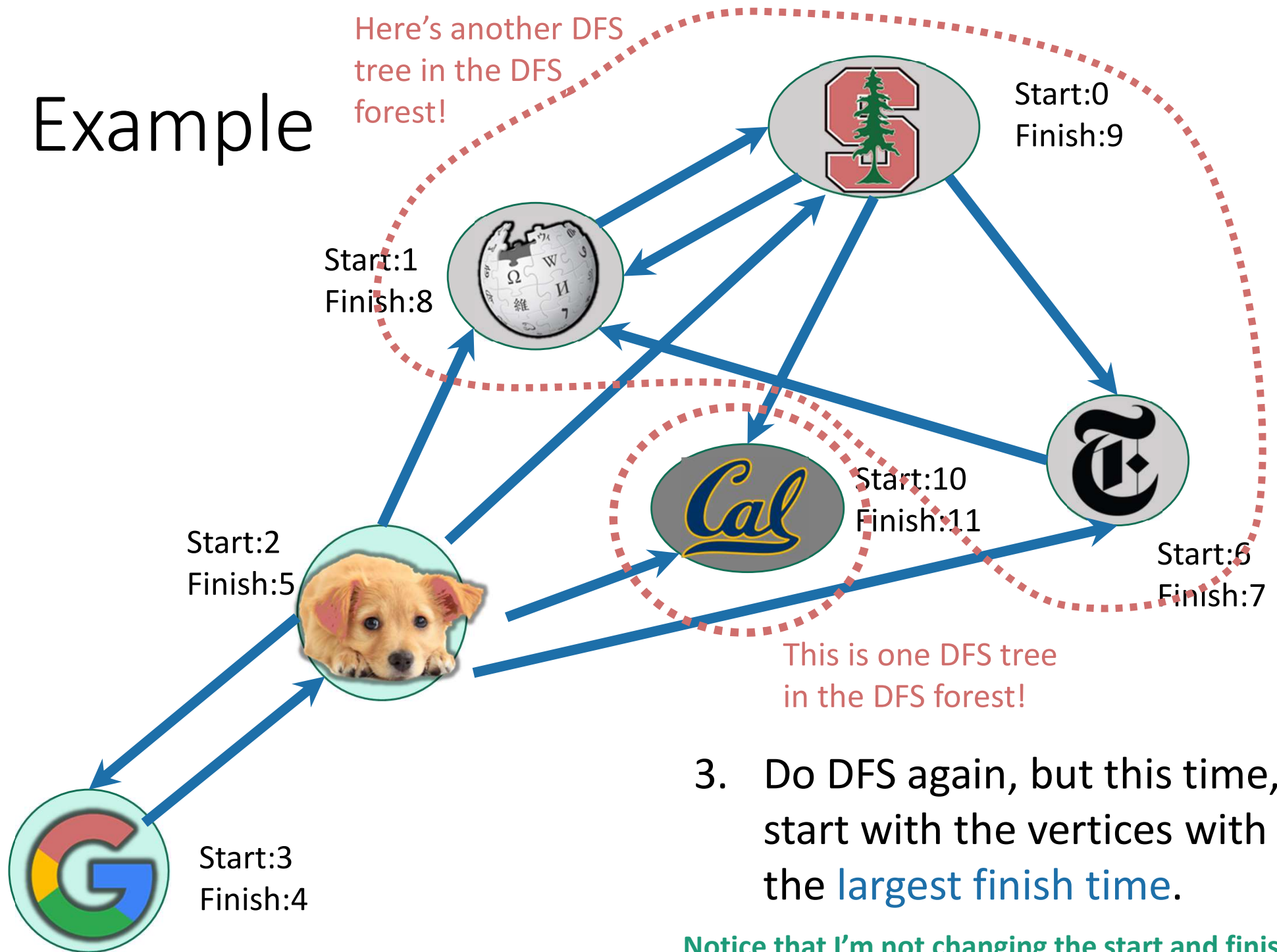
Example



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.

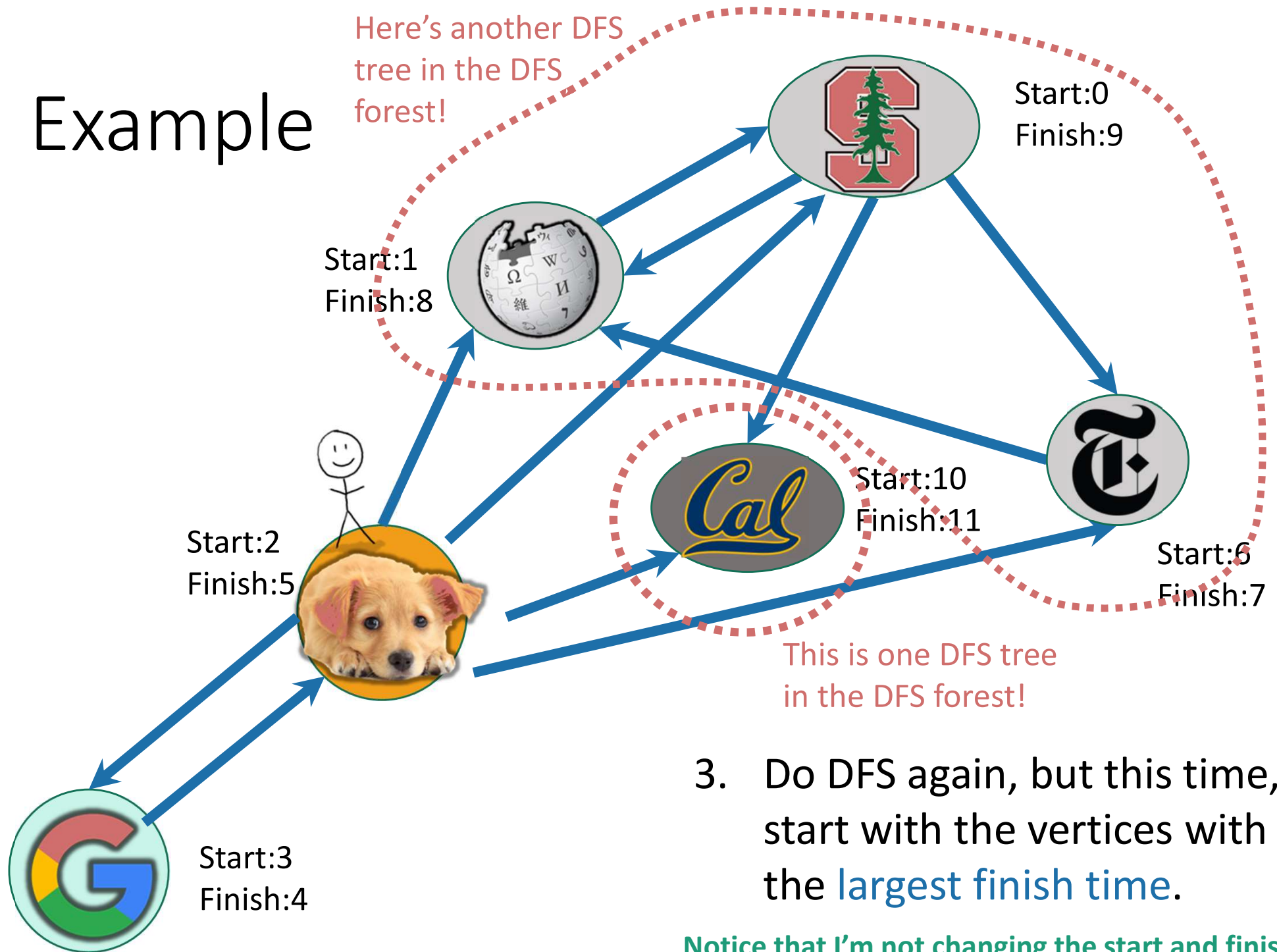
Example



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.

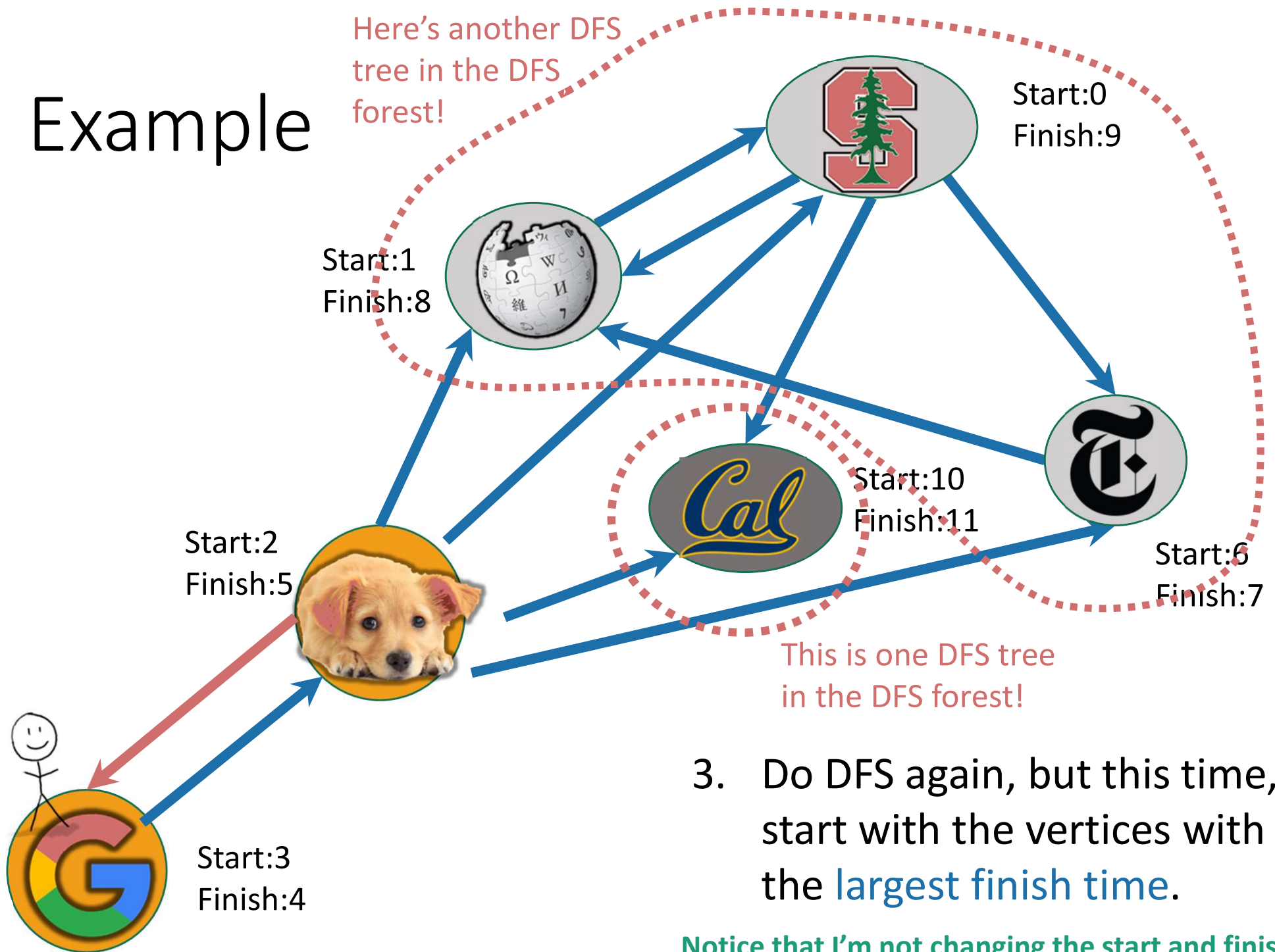
Example



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.

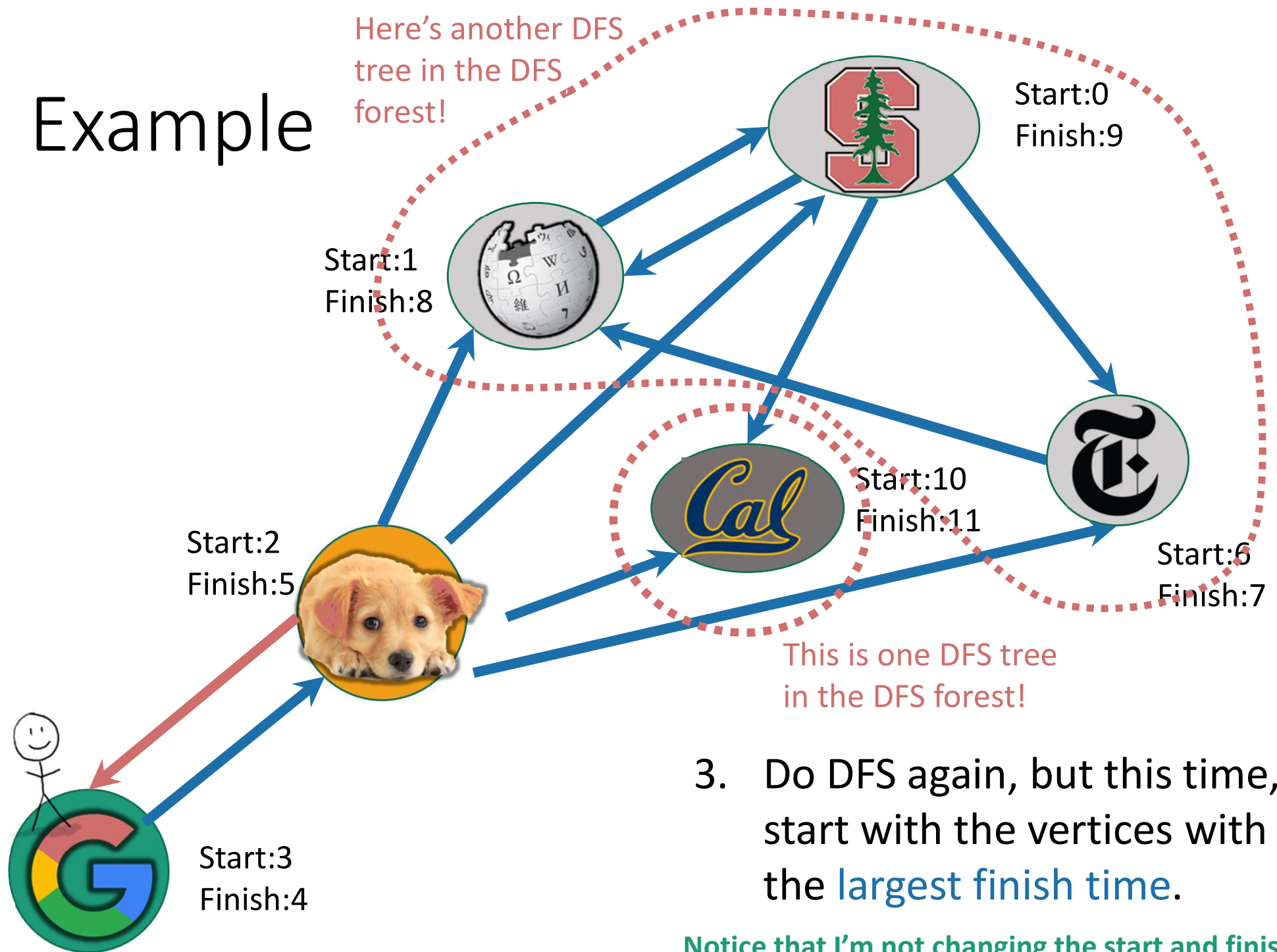
Example



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

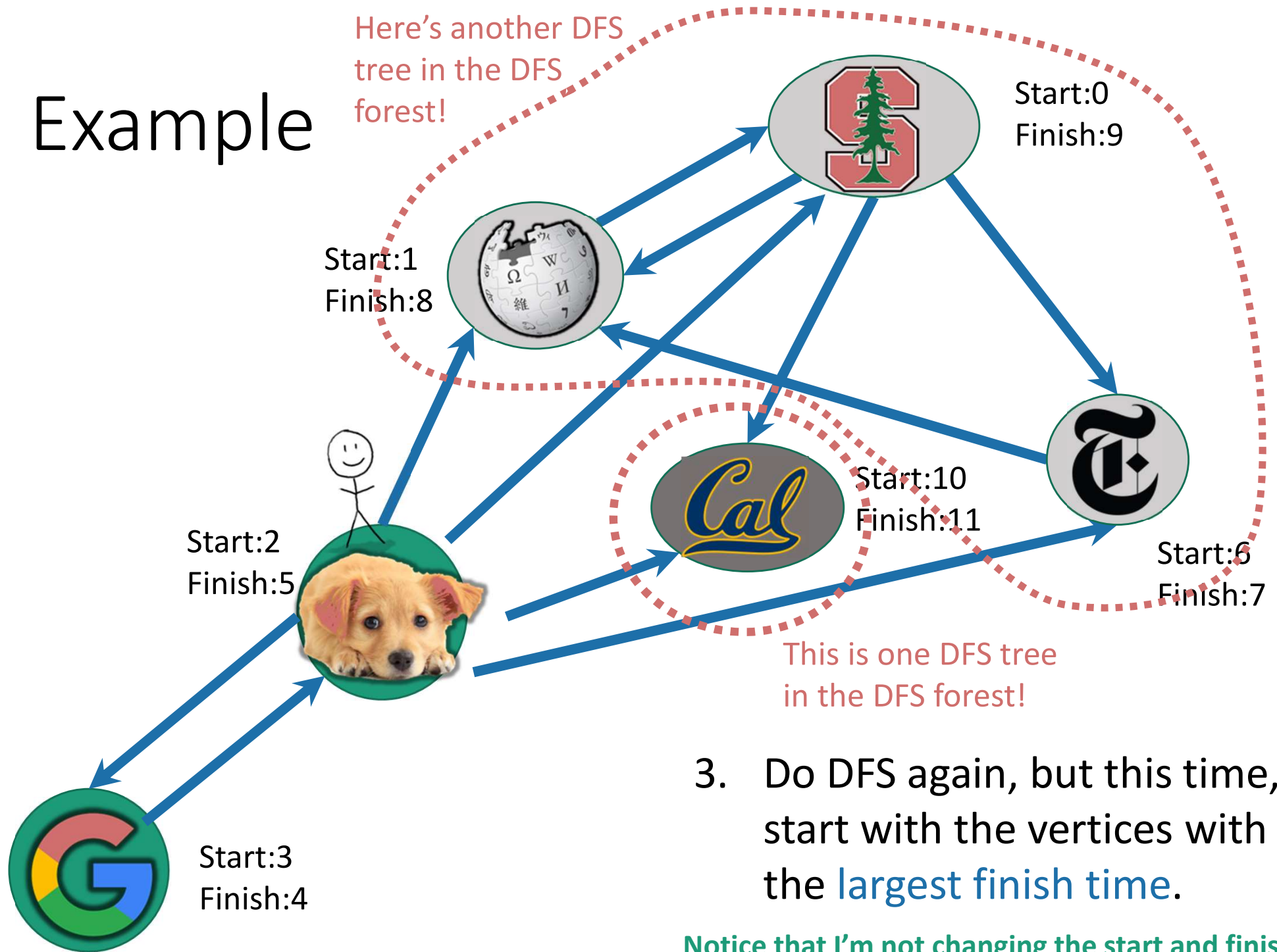
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

Example



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

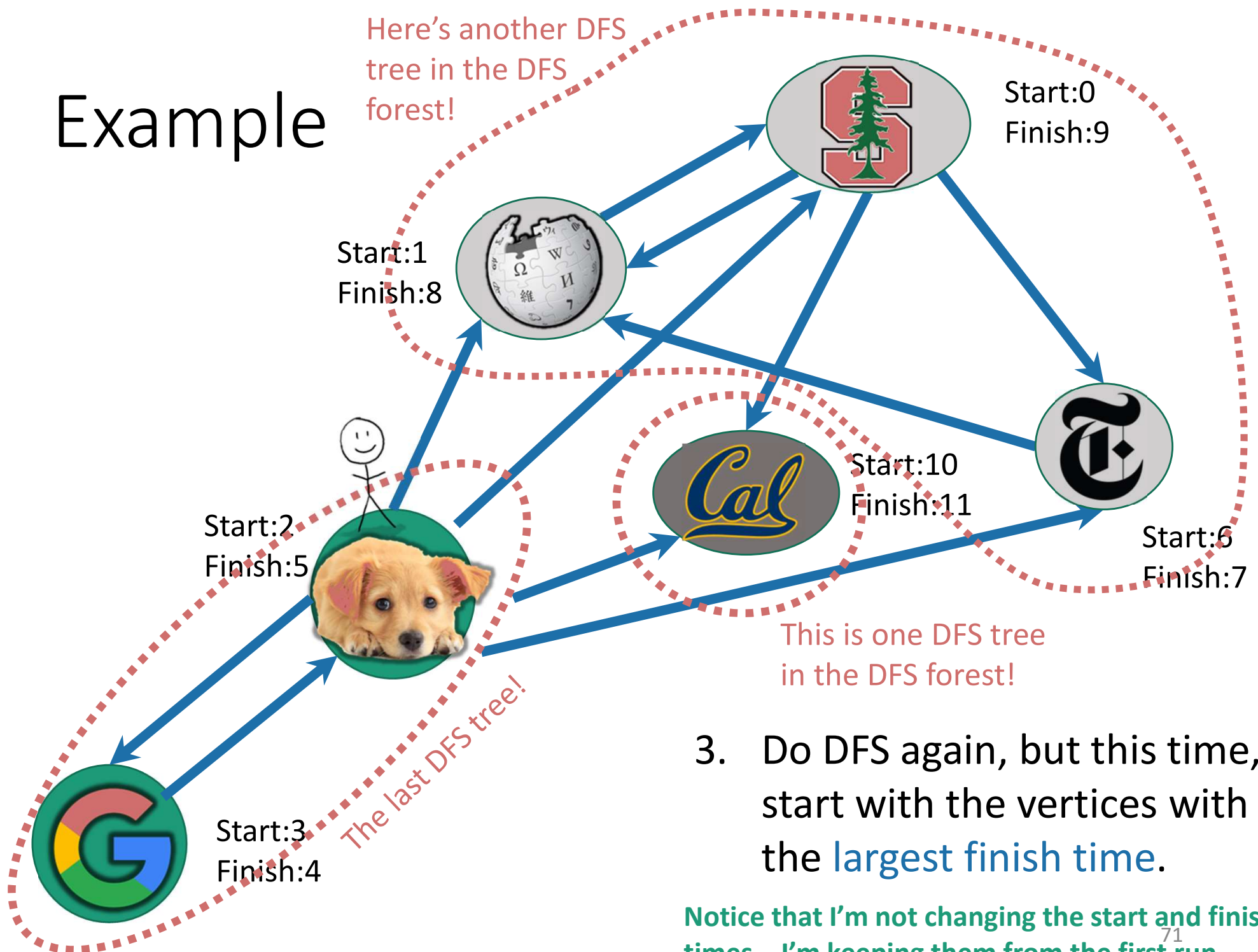
Example



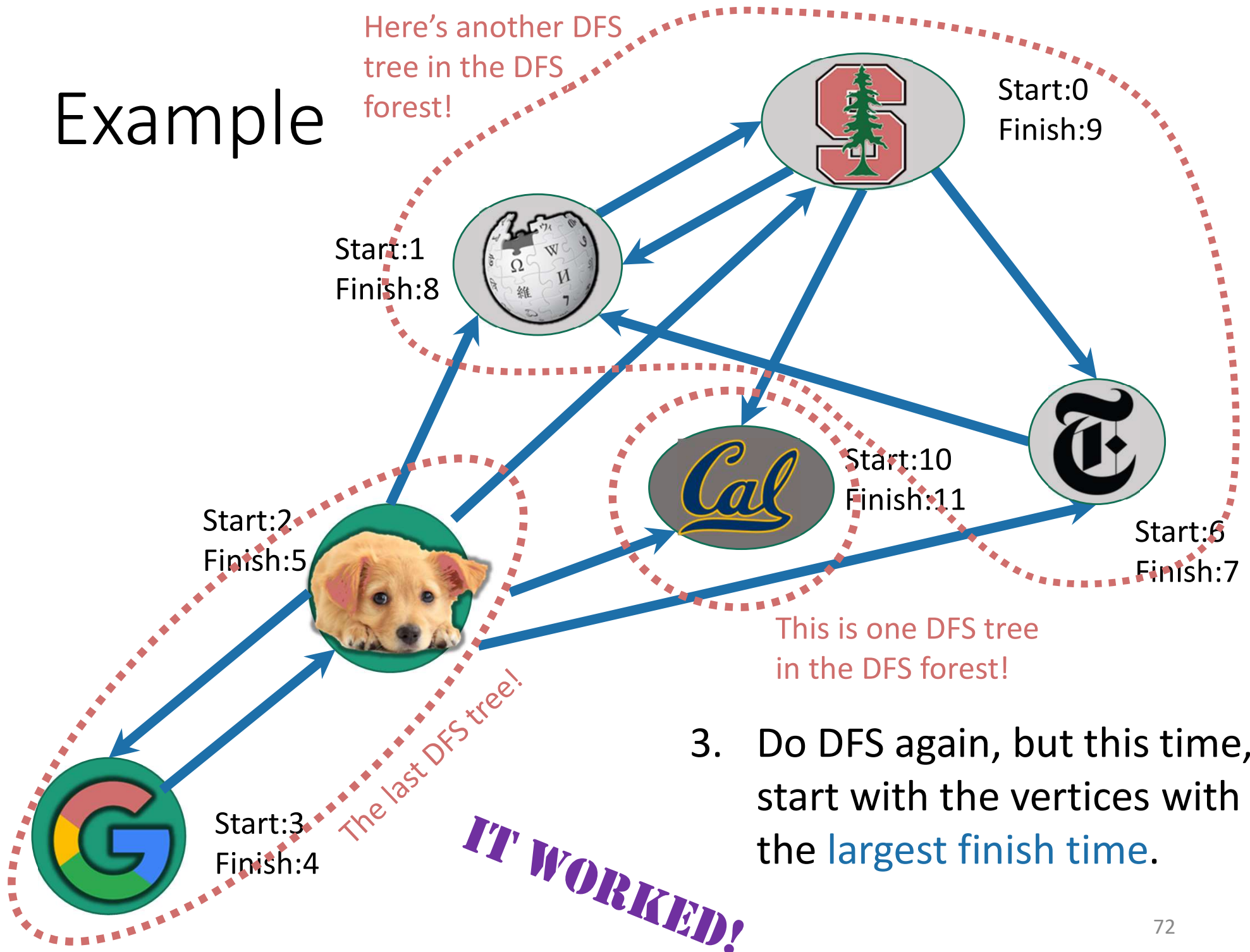
3. Do DFS again, but this time, start with the vertices with the **largest finish time**.

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

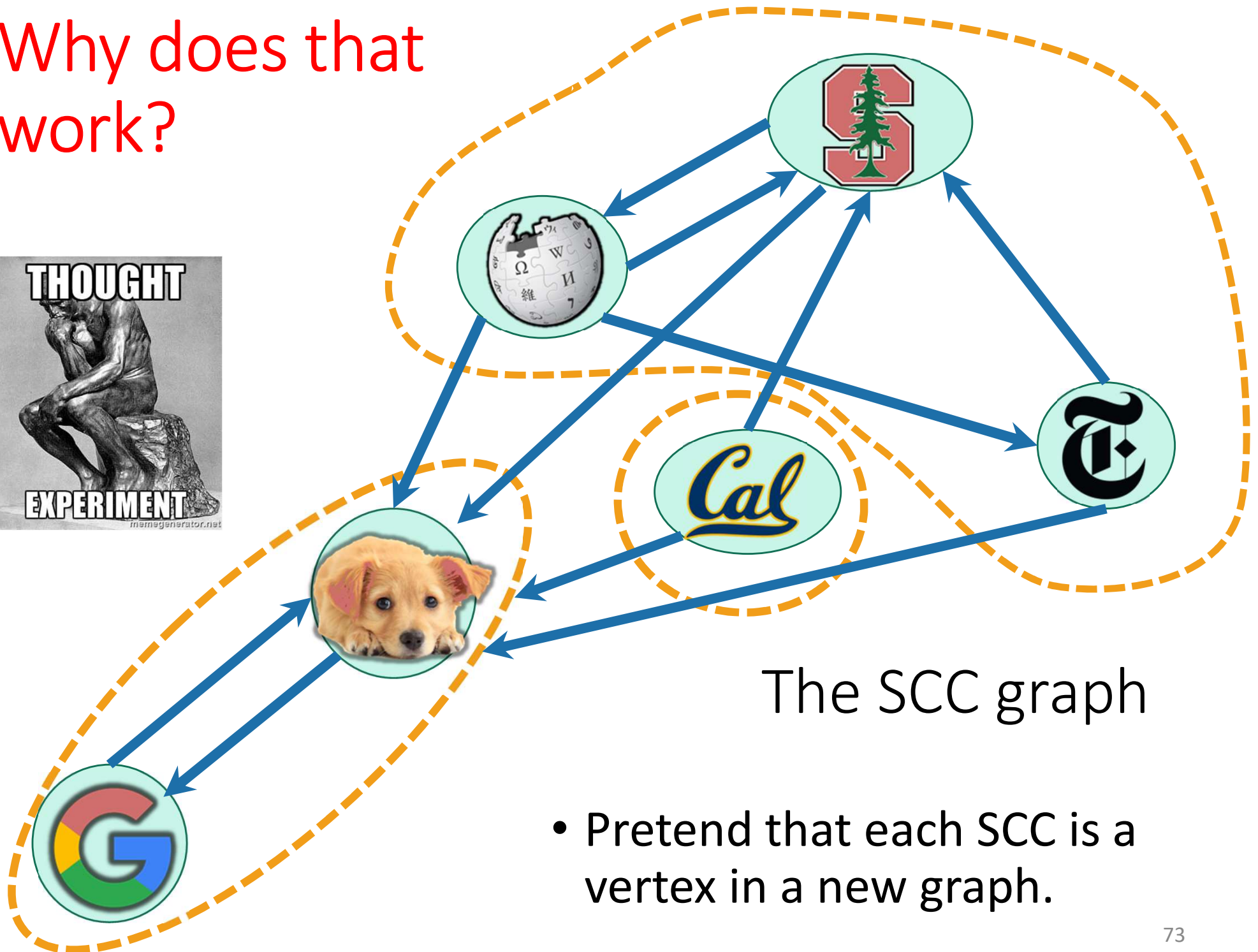
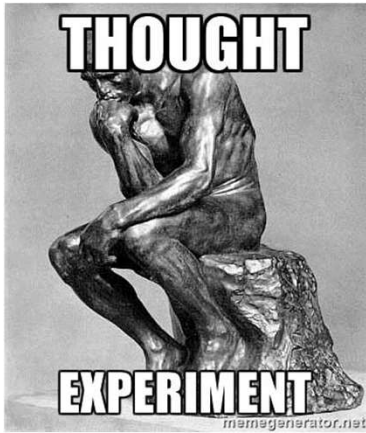
Example



Example



Why does that work?



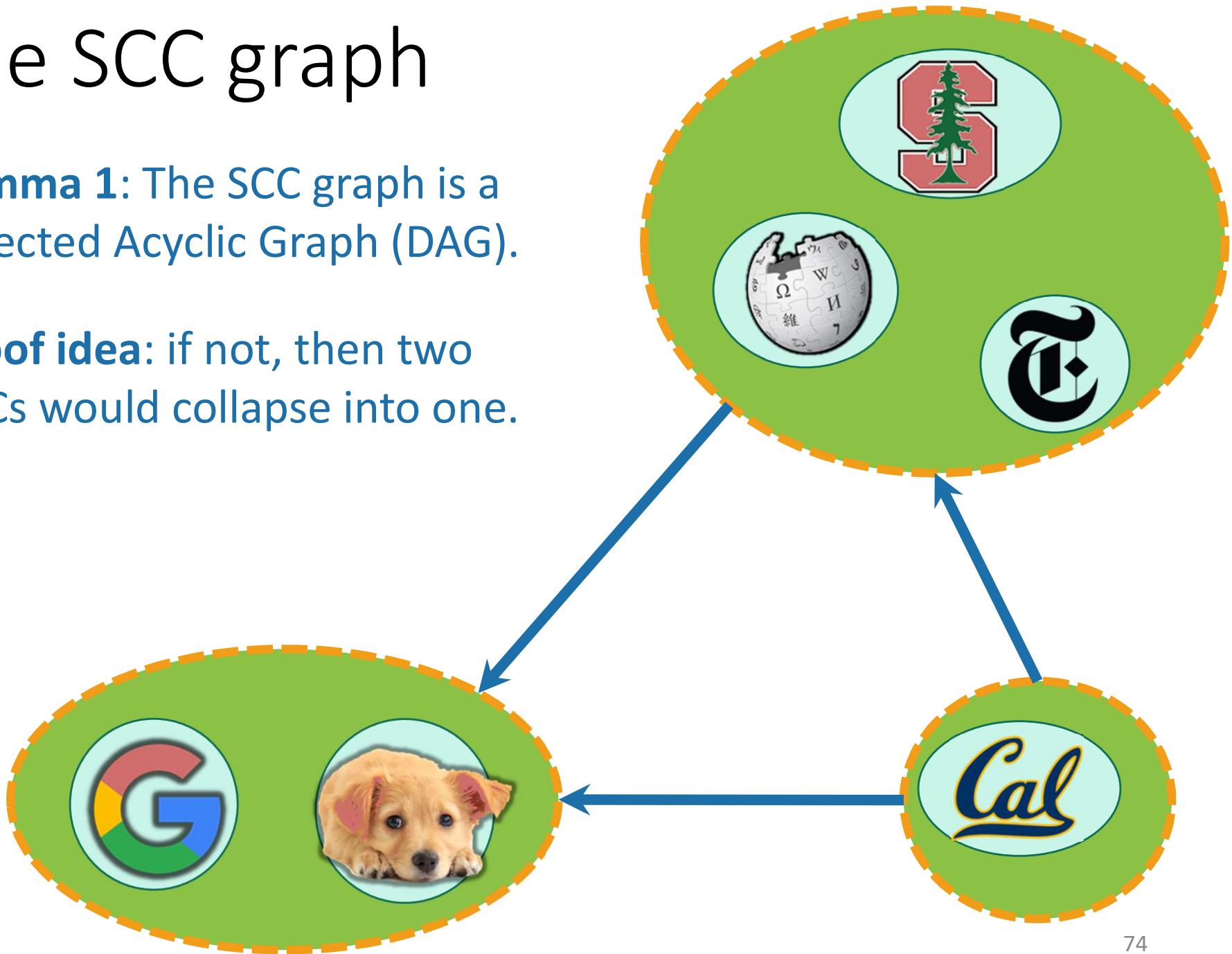
The SCC graph

- 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).

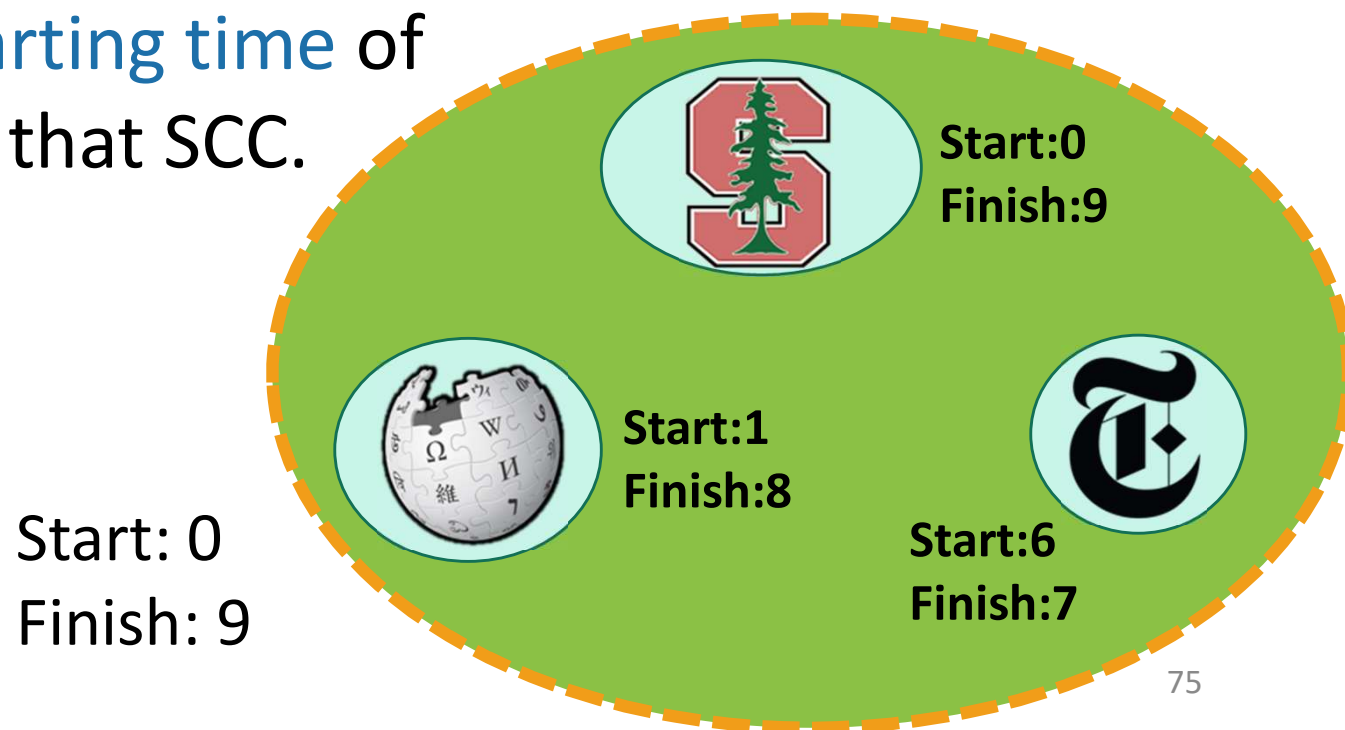
Proof idea: if not, then two SCCs would collapse into one.



Starting and finishing times in a SCC

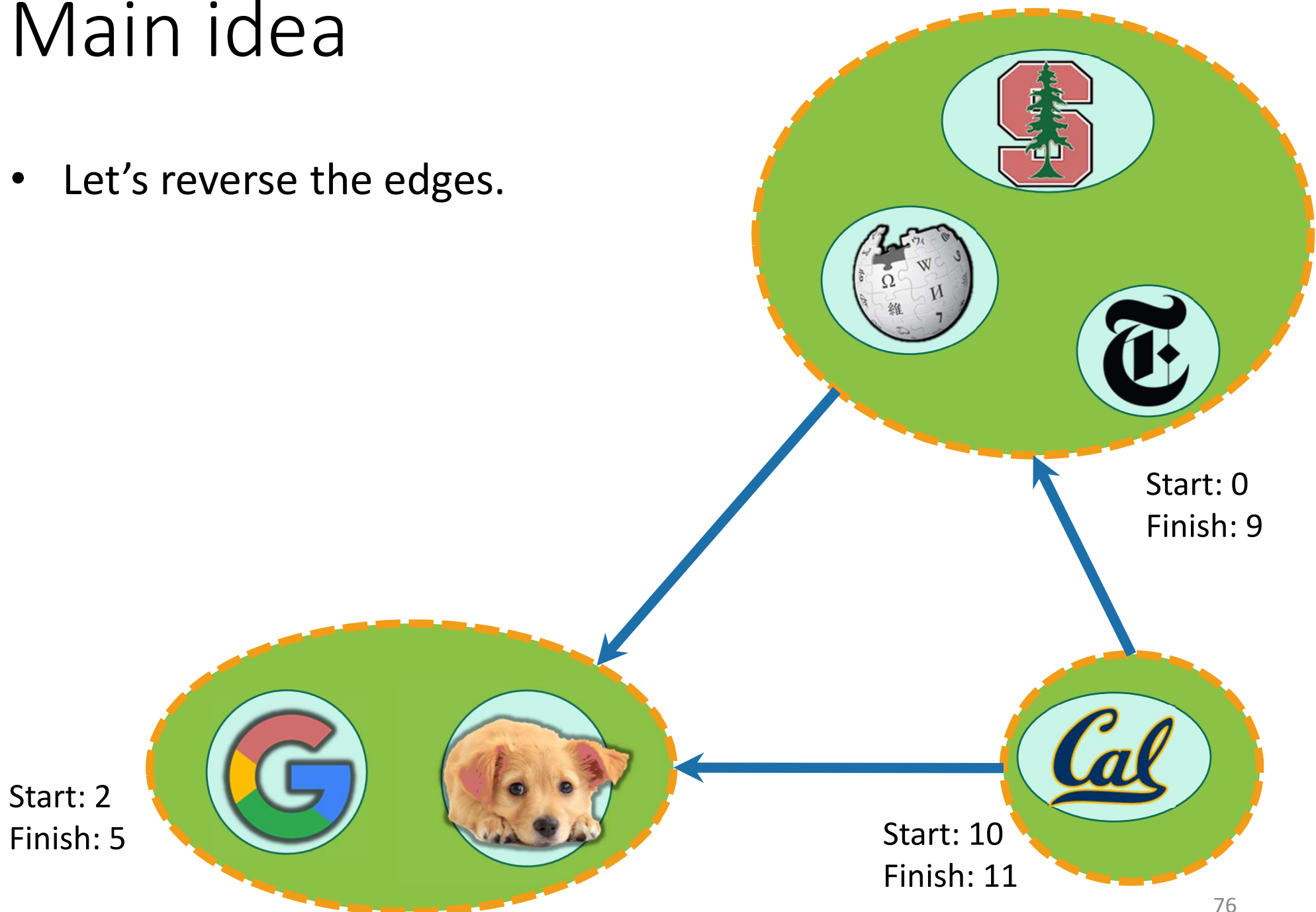
Definitions:

- 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.



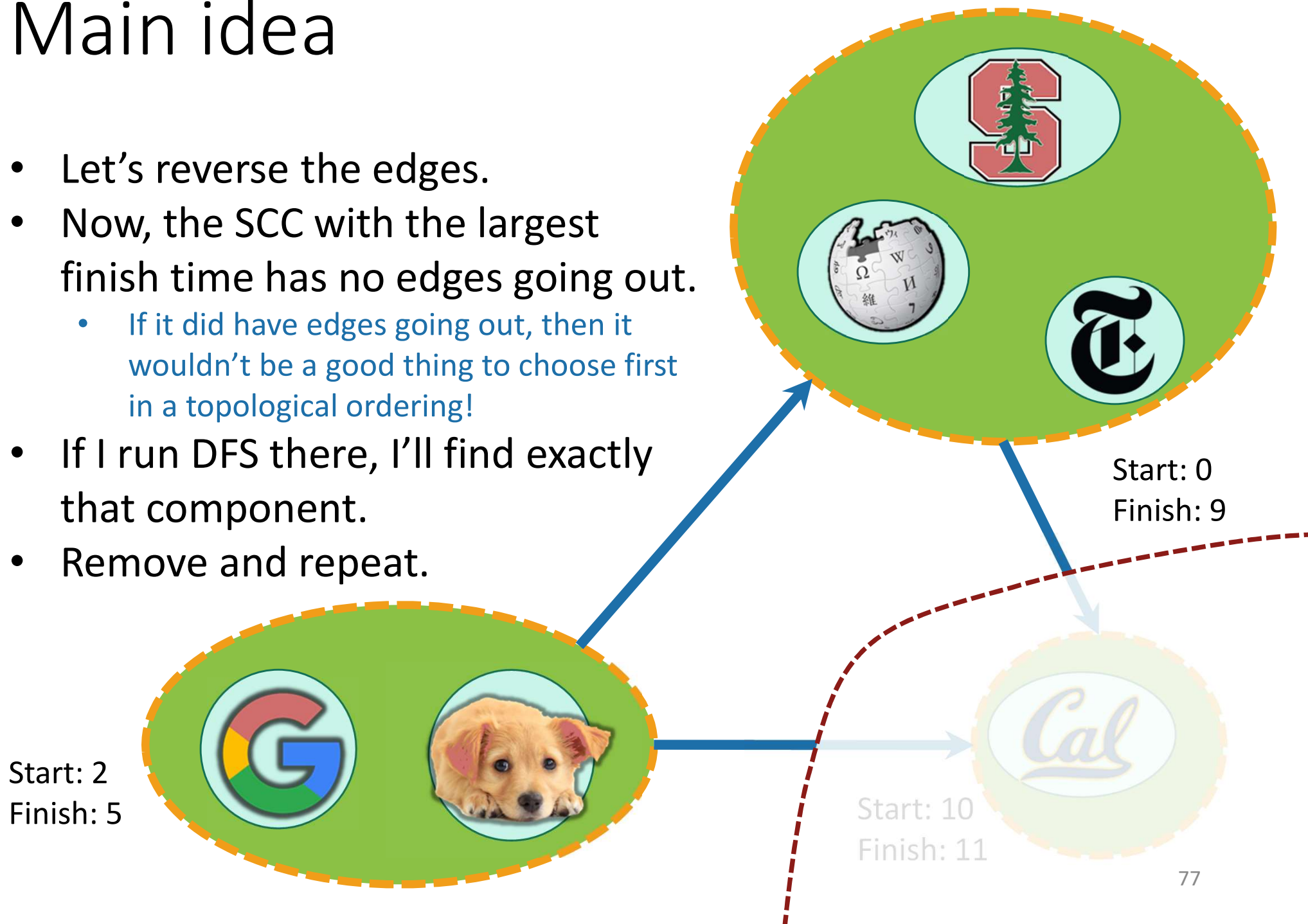
Main idea

- 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.



Tổng kết

- BFS có thể dùng để tìm đường đi ngắn nhất trong đồ thị không có trọng số
- DFS có thể khám phá các cấu trúc hữu ích
 - Áp dụng cho duyệt **Topological Sorting** với độ phức tạp $O(n + m)$
 - Tìm miền liên thông mạnh (**Strongly Connected Components**) với độ phức tạp $O(n + m)$
 - Đây là một bài toán không tầm thường (non-trivial).