CS 161 Section 5

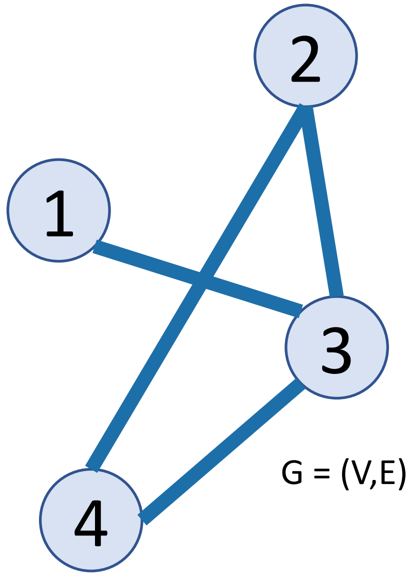
CA: [name of CA]

Lecture Recap

Undirected Graphs

- Have vertices and edges
  - V is the set of vertices
  - E is the set of edges
  - Formally, a graph is G = (V,E)

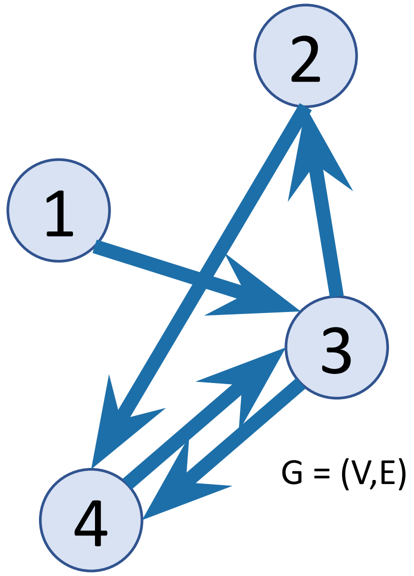
- Example
  - V = {1,2,3,4}
  - E = { {1,3}, {2,4}, {3,4}, {2,3} }



- The **degree** of vertex 4 is 2.
  - There are 2 edges coming out.
- Vertex 4's **neighbors** are 2 and 3

Directed Graphs

- Have vertices and edges
  - V is the set of vertices
  - E is the set of **DIRECTED** edges
  - Formally, a graph is G = (V,E)
- Example
  - V = {1,2,3,4}
  - E = { (1,3), (2,4), (3,4), (4,3), (3,2) }



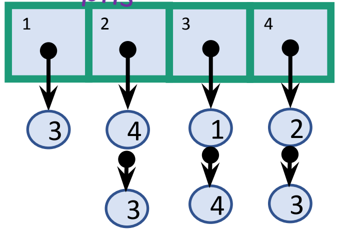
- The **in-degree** of vertex 4 is 2.
- The **out-degree** of vertex 4 is 1.
- Vertex 4's **incoming neighbors** are 2,3
- Vertex 4's **outgoing neighbor** is 3.

Trade-offs

n = # of vertices
m = # of edges

Say there are n vertices and m edges.

Generally better for **sparse** graphs

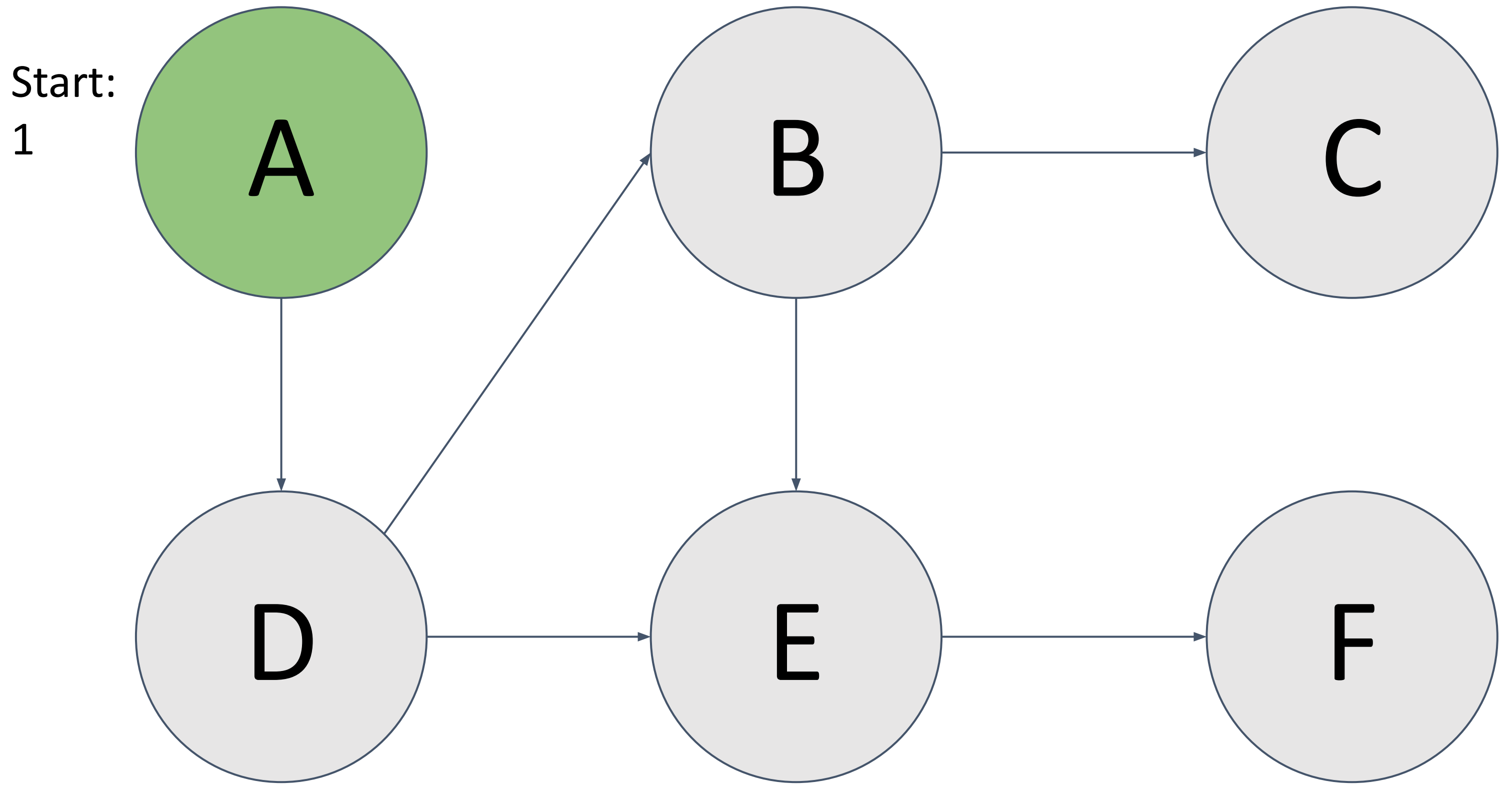
	$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$	
Edge membership Is e = {v,w} in E?	O(1)	O(deg(v)) or O(deg(w))
Neighbor query Give me v's neighbors.	O(n)	O(deg(v))
Space requirements	$O(n^2)$	O(n + m)

We'll assume this representation for the rest of the class

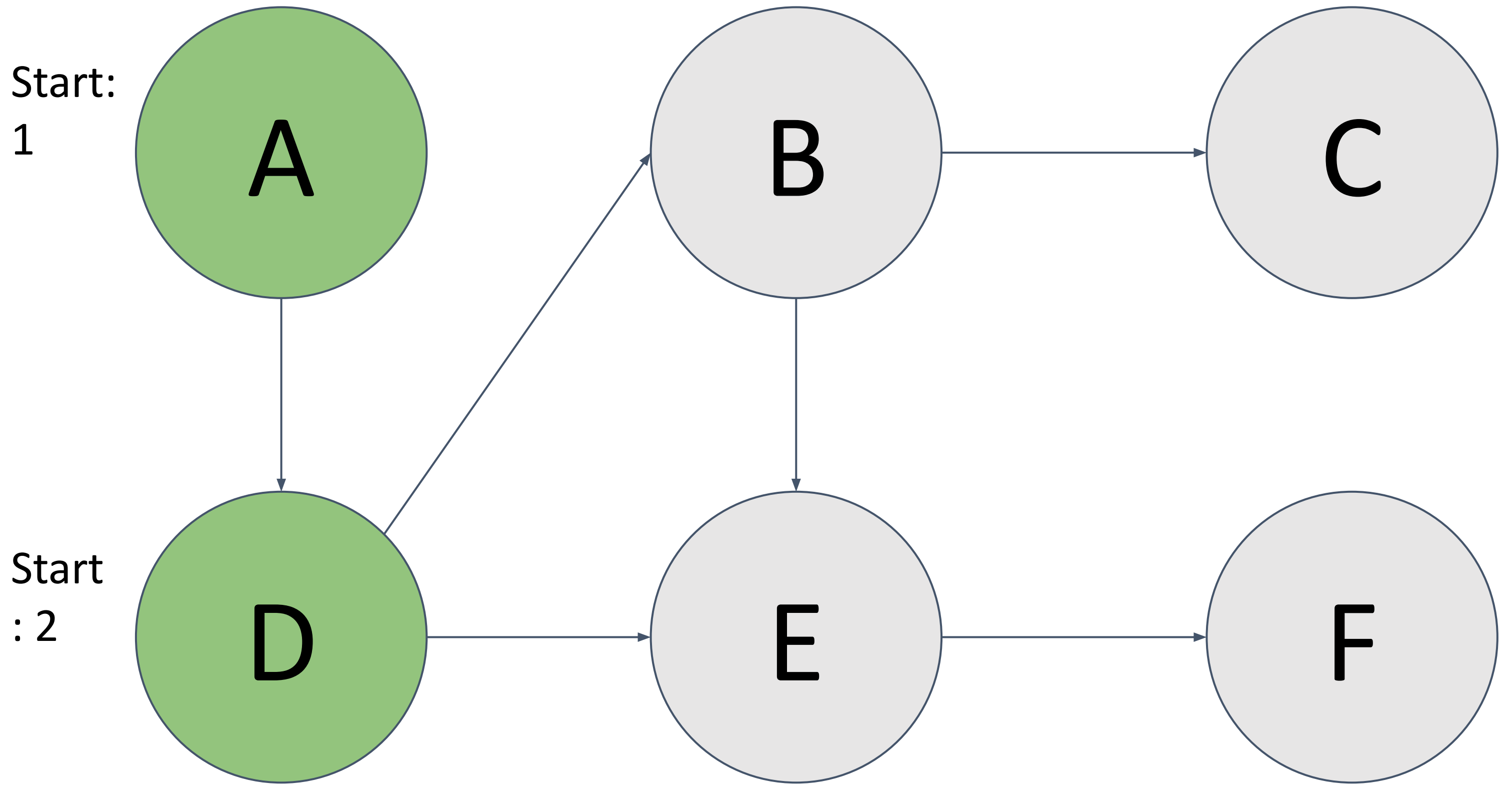
Depth-First Search: Pseudocode with start and end times

```
DFS(w, currentTime):
    w.startTime = currentTime
    currentTime ++
    Mark w as in progress.
    for v in w.neighbors:
        if v is unvisited:
            currentTime
              = DFS(v, currentTime)
            currentTime ++
    w.finishTime = currentTime
    Mark w as all done
    return currentTime
```

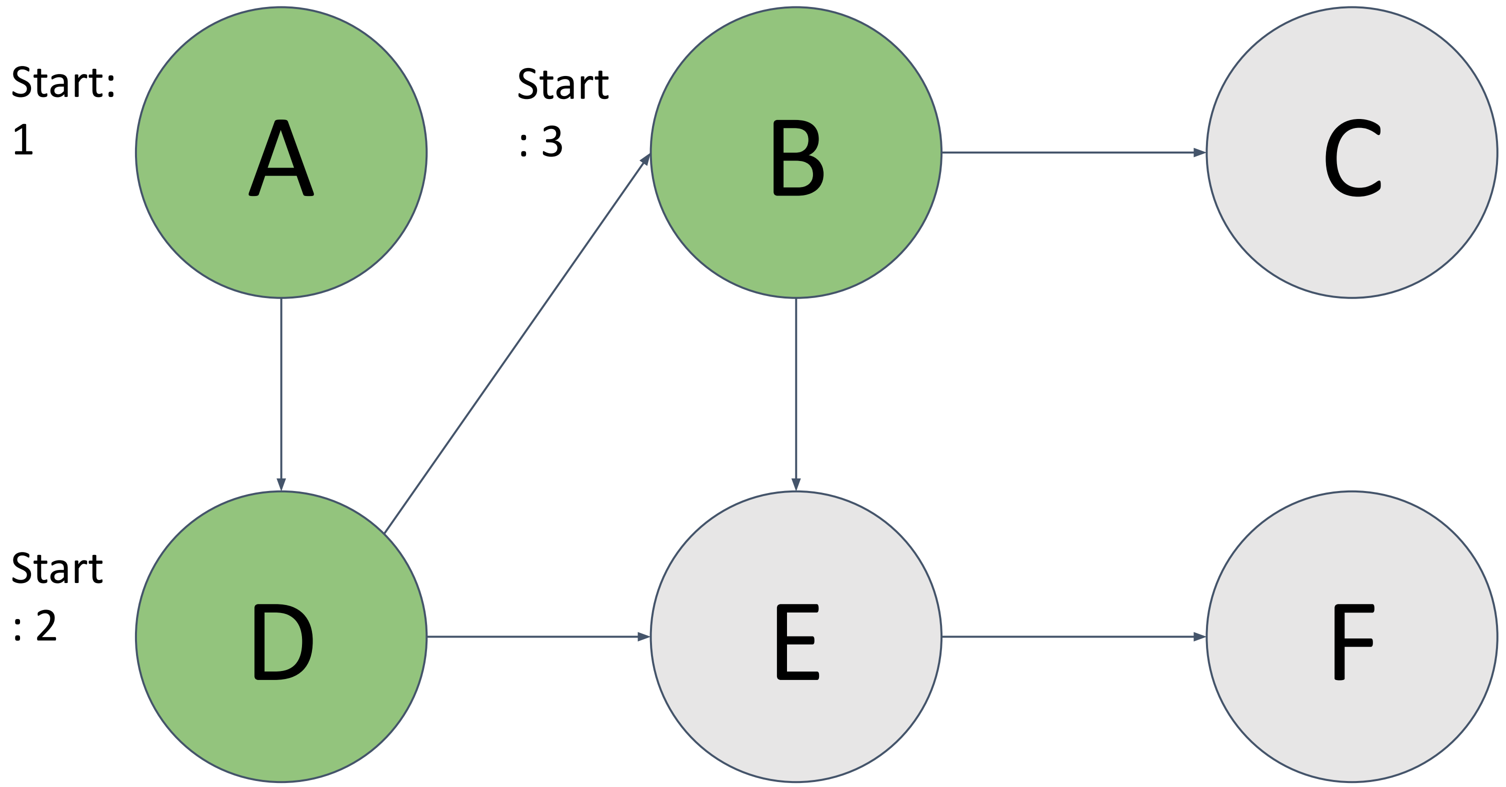
Example DFS (we sort edges in alphabetical order)



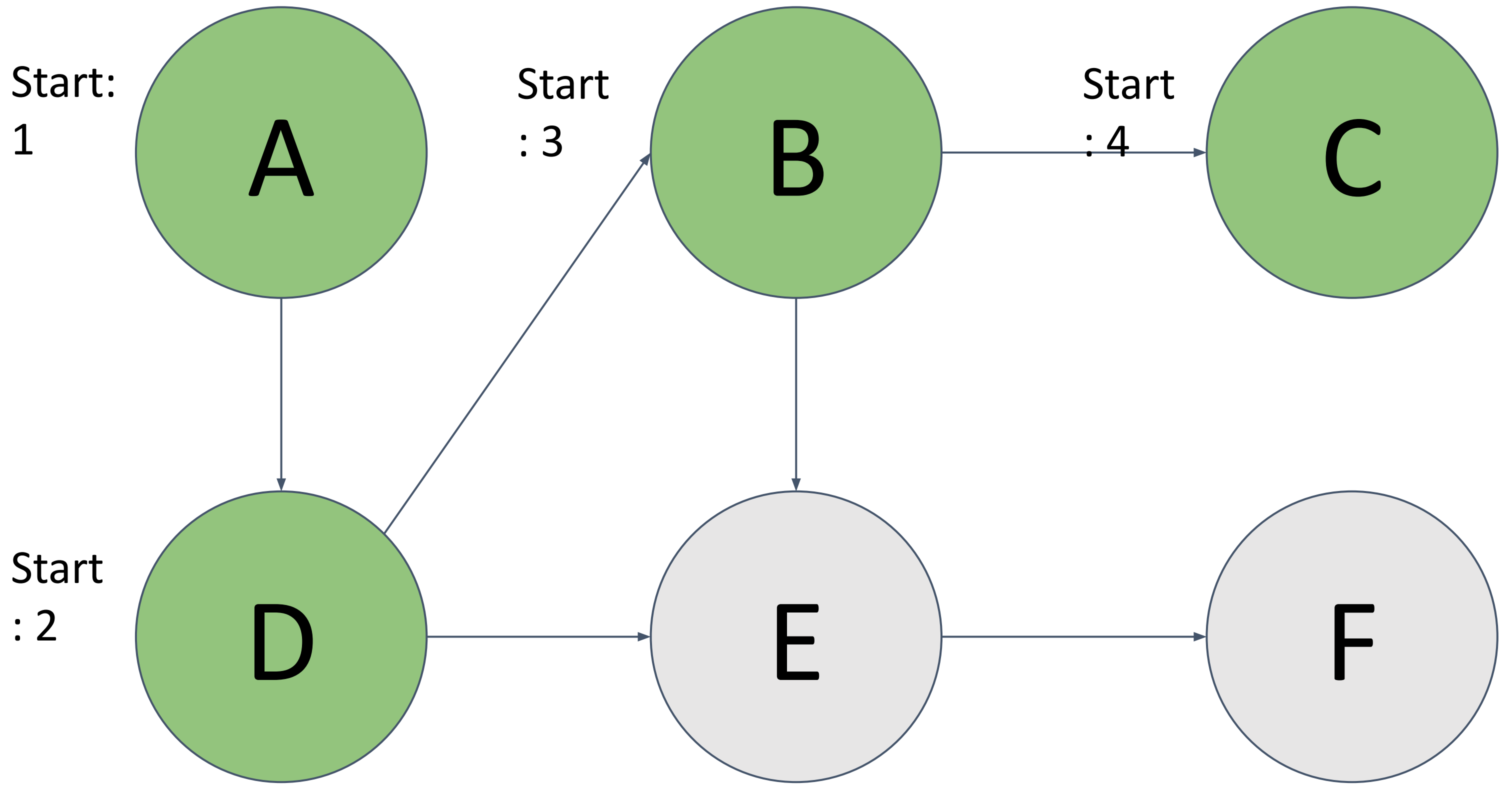
Example DFS (we sort edges in alphabetical order)



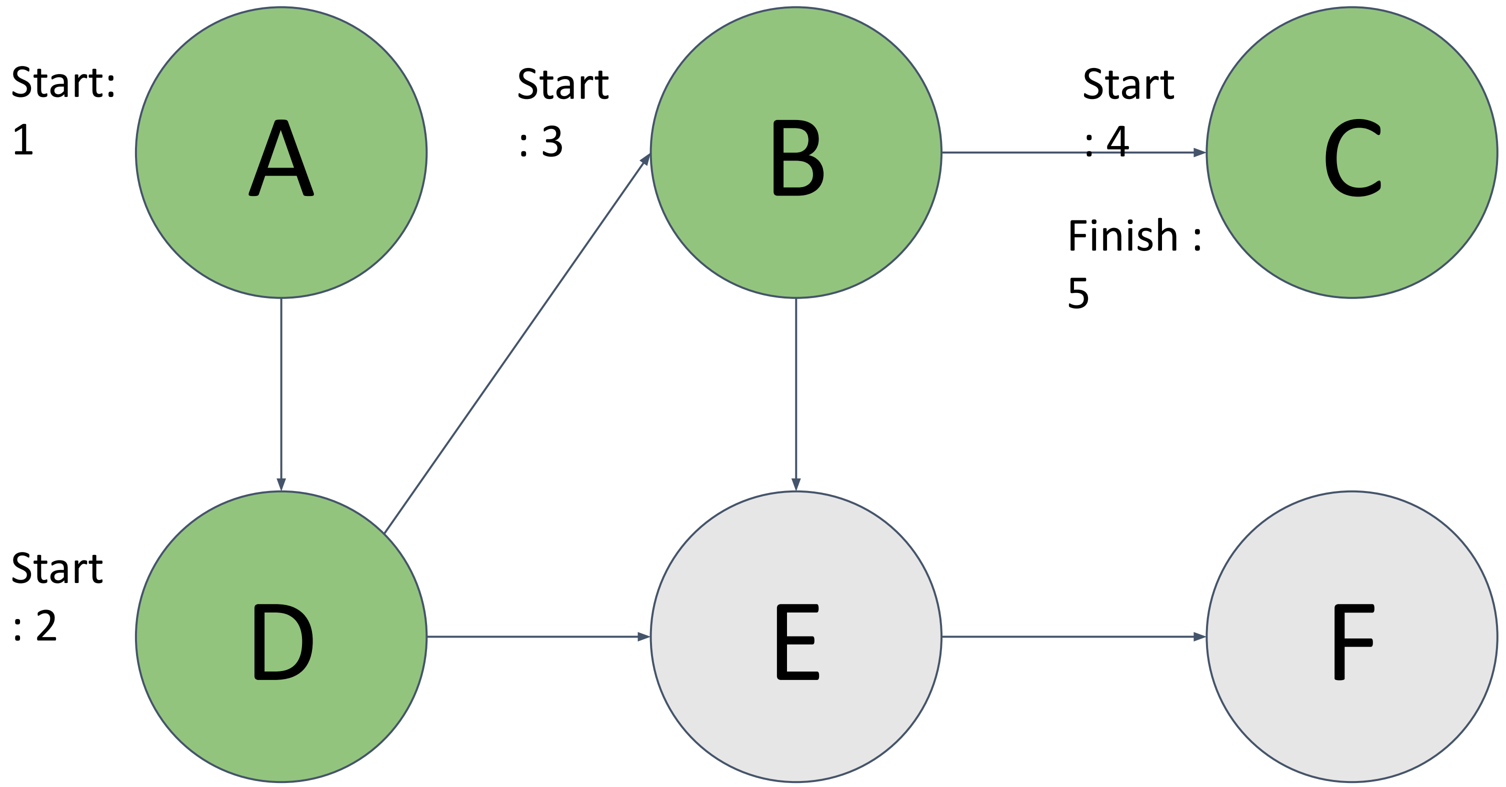
Example DFS (we sort edges in alphabetical order)



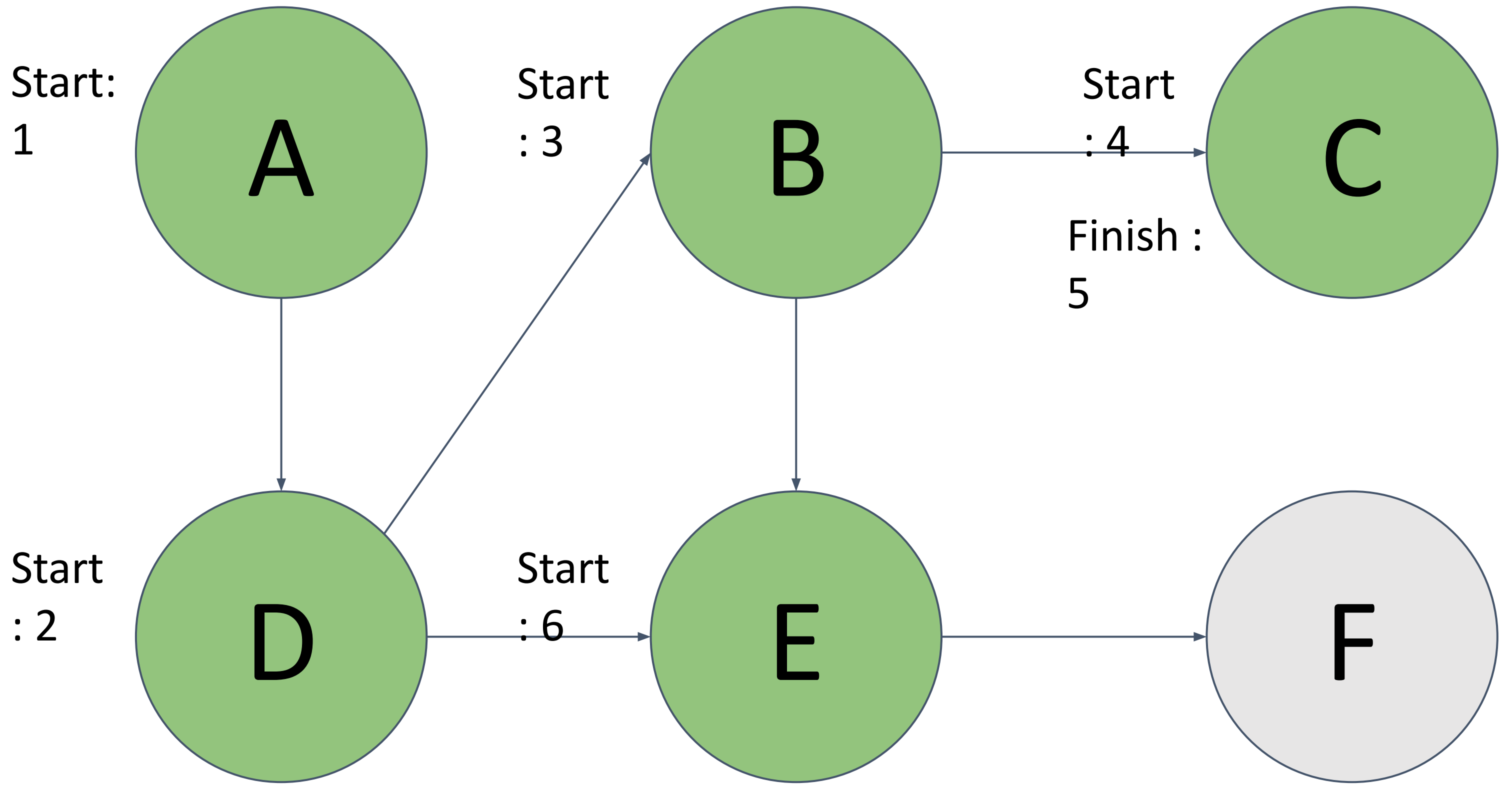
Example DFS (we sort edges in alphabetical order)



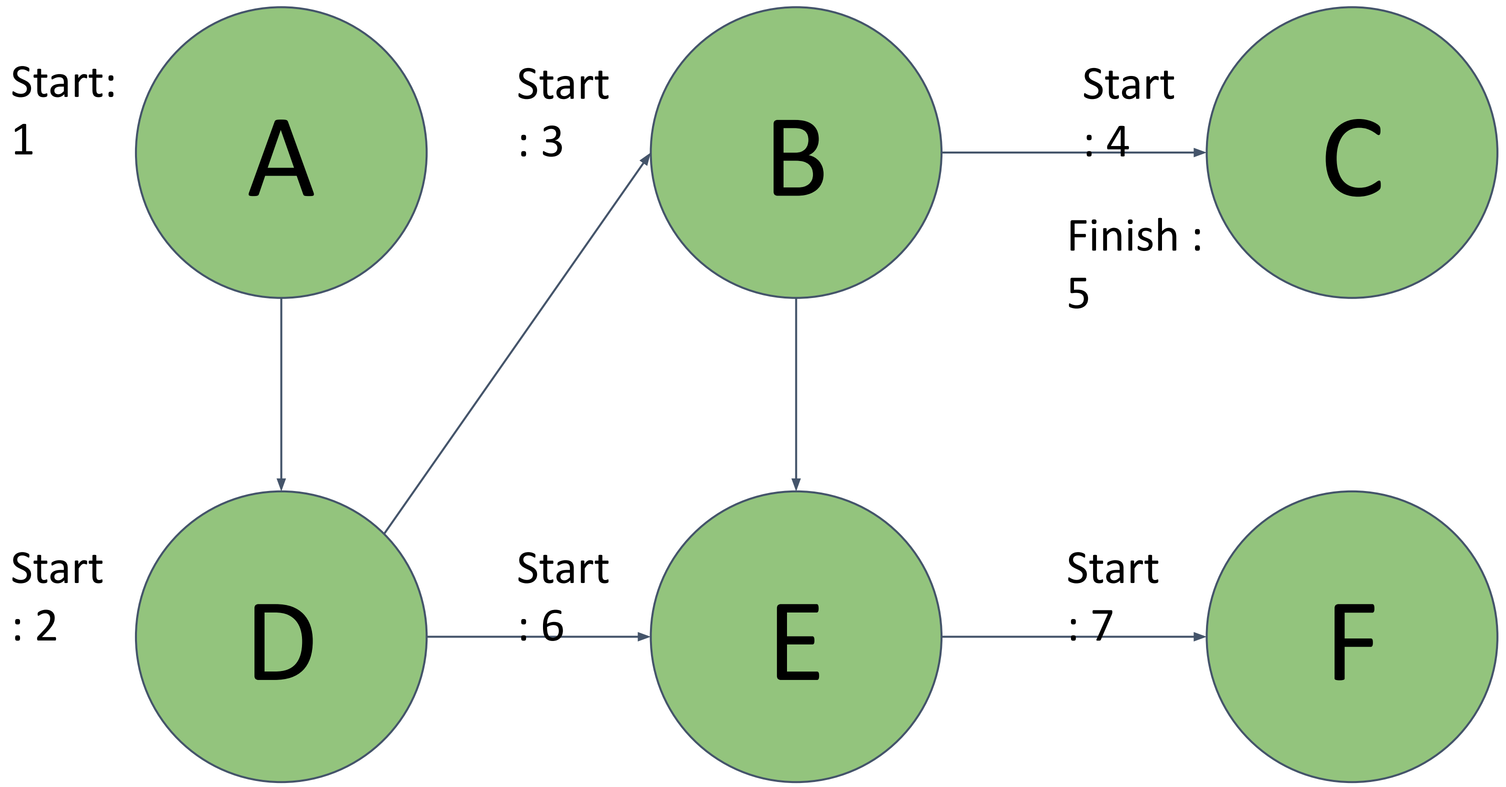
Example DFS (we sort edges in alphabetical order)



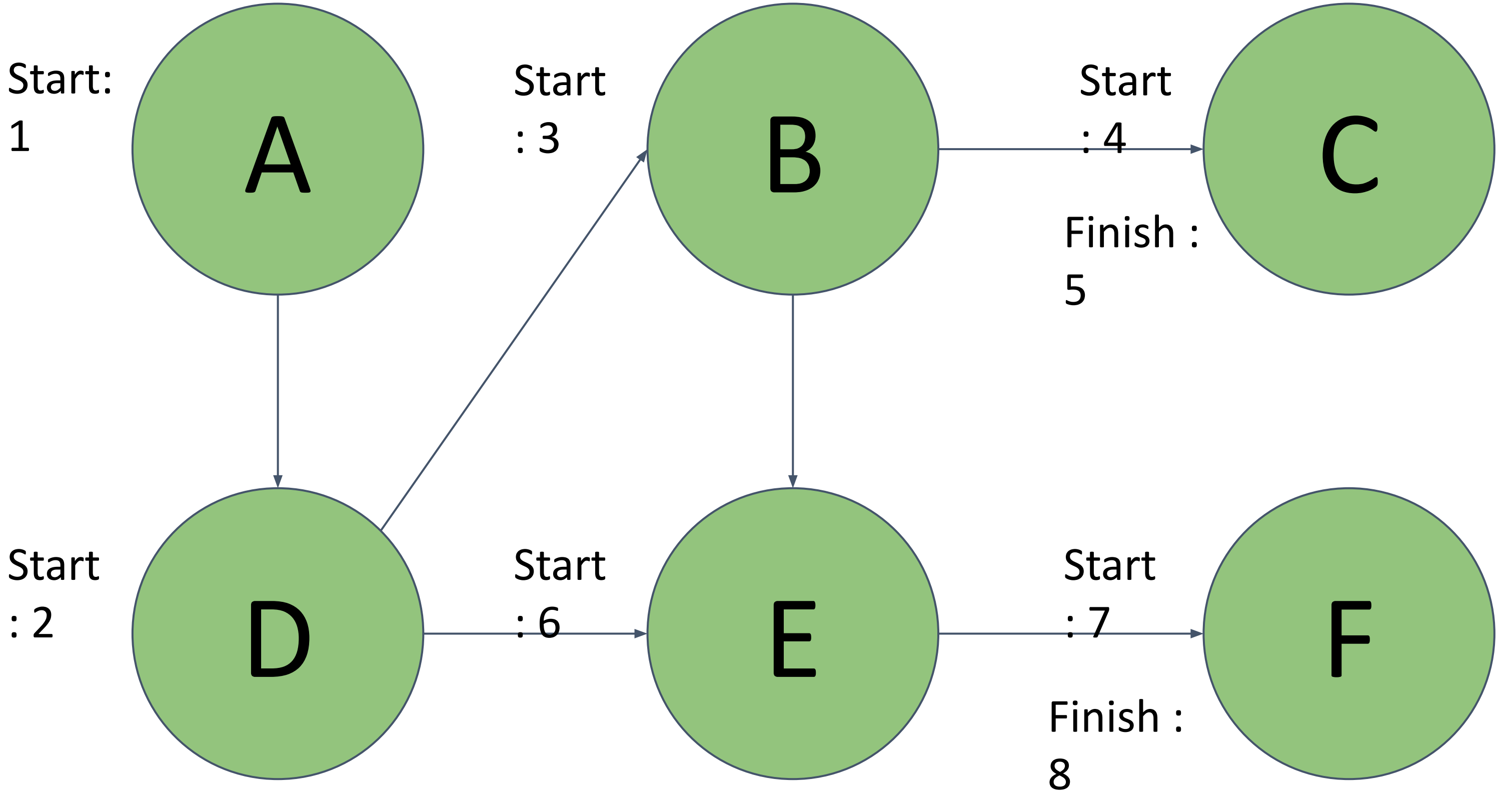
Example DFS (we sort edges in alphabetical order)



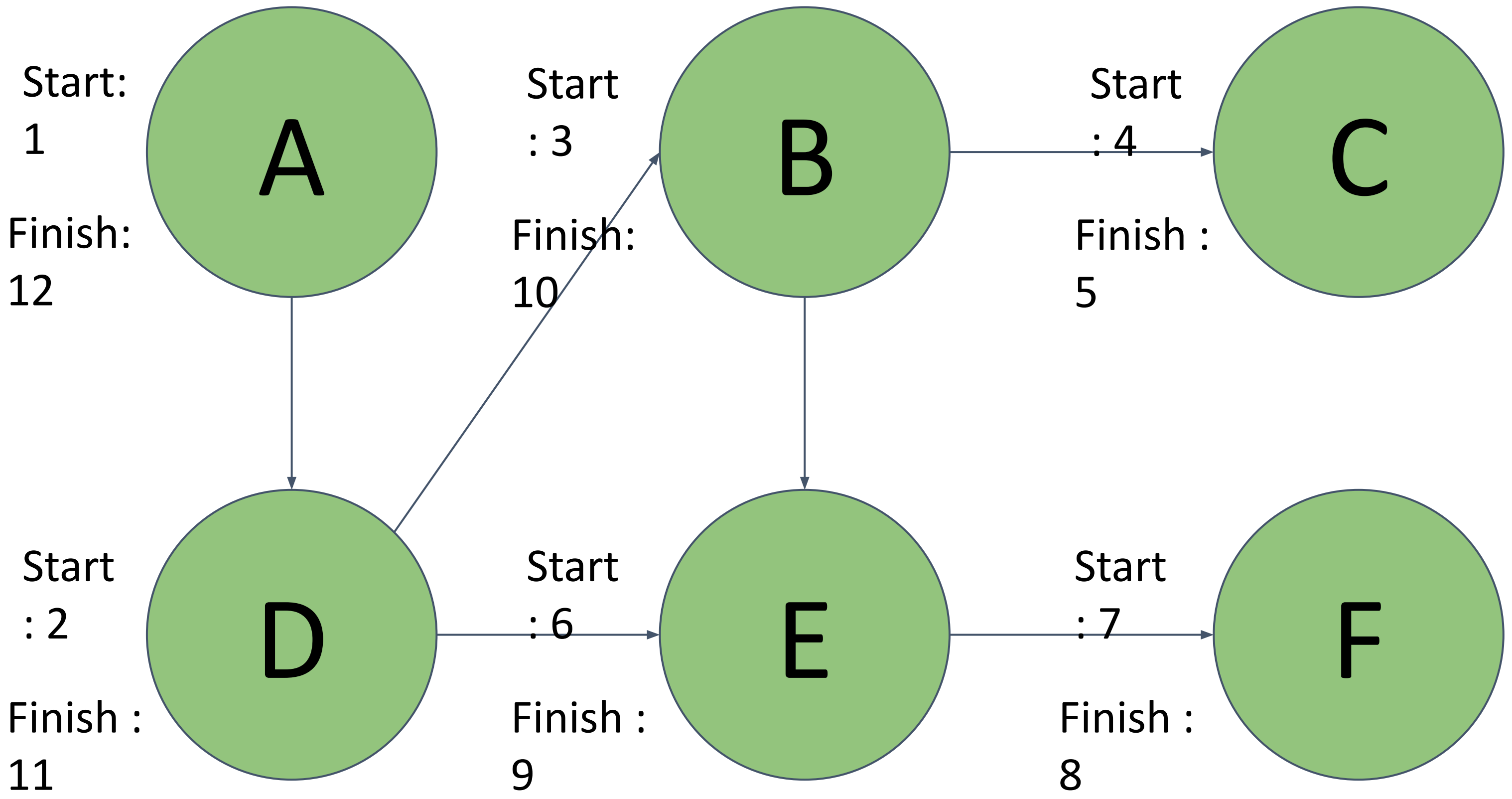
Example DFS (we sort edges in alphabetical order)



Example DFS (we sort edges in alphabetical order)



Example DFS (we sort edges in alphabetical order)



Relation in DFS tree and start/finish time

- u is v’s ancestor if start(u) < start(v) < finish(v) < finish(u)
- u is v’s descendant if start(v) < start(u) < finish(u) < finish(v)
- u is v’s cousin if start(v) < finish(v) < start(u) < finish(u)
- Notice that startTime and finishTime never interleave (start(v)<start(u)<finish(v)<finish(u)) will never happen.

Breadth-First Search: Pseudocode

- BFS(G, v)
- Input: graph G = (V, E); node v in V, edge e in E
- Output:
  - Array visited: V -> {True, False}
  - Layer i of vertices. Initialize L_0 with start vertex v.

Breadth-First Search: Pseudocode

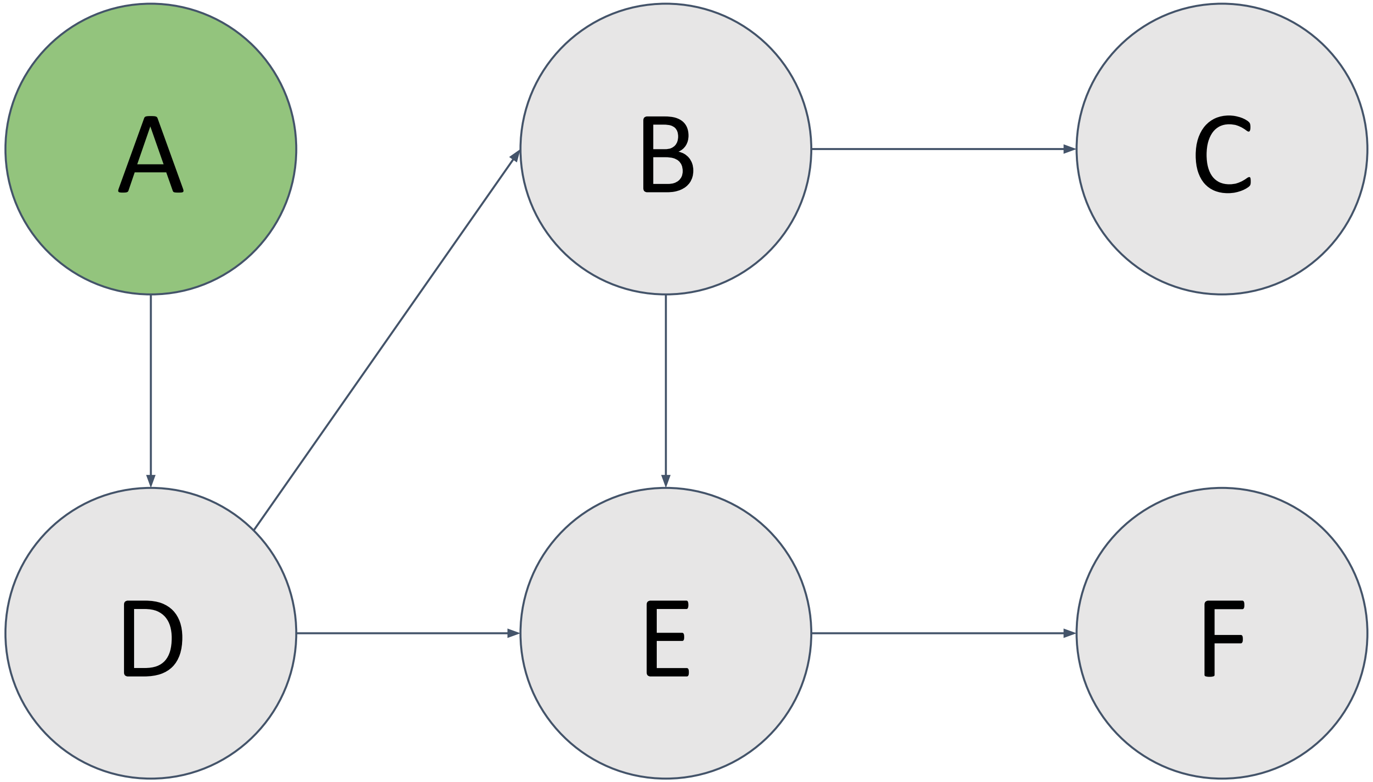
- BFS(G, v)
- Input: graph G = (V, E); node v in V, edge e in E
- Output:
    - Array visited: V -> {True, False}
    - Layer of vertices to visit.
        - Initialize L_0 with start vertex v. Mark v as visited
    - While L_i is non-empty: # keep looping as we have unexplored vertices
        - Visit neighbours of elements in L_i # vertices currently being explored

Breadth-First Search: Pseudocode

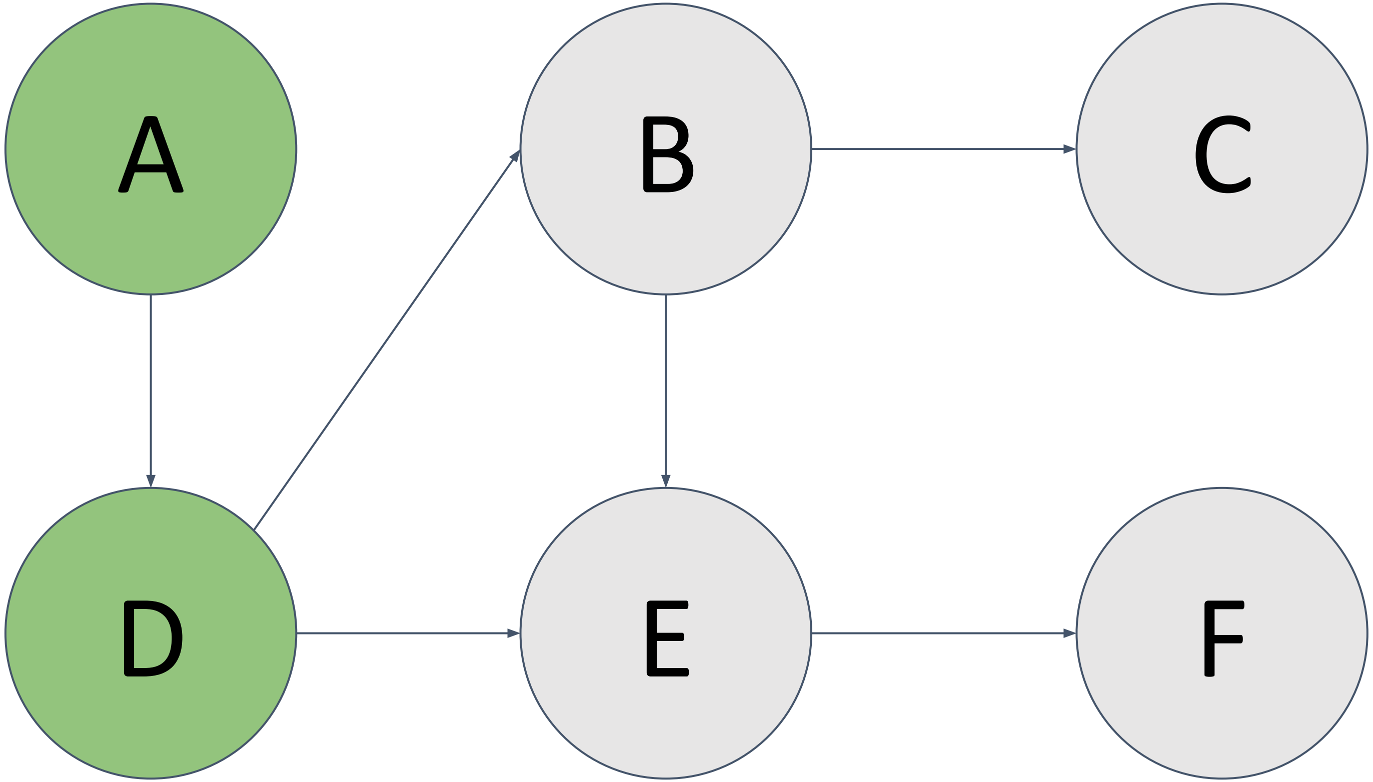
```
● BFS(G, v)
● Input: graph G = (V, E); node v in V, edge e in E
● Output:
    ○ Array visited: V -> {True, False}
    ○ Layer of vertices to visit.
        ■ Initialize $L_0$ with start vertex v. Mark v as visited
    ○ While $L_i$ is non-empty:     # keep looping as we have unexplored vertices
        ■ for u in $L_i$:       # vertices currently being explored
            ● for v neighbour of u:
                ○ if not visited v: # only work on vertices we have not explored
                    ■ mark v visited and add to $L_{i+1}$
        ■ i += 1
```

Can we use queues instead of lists?

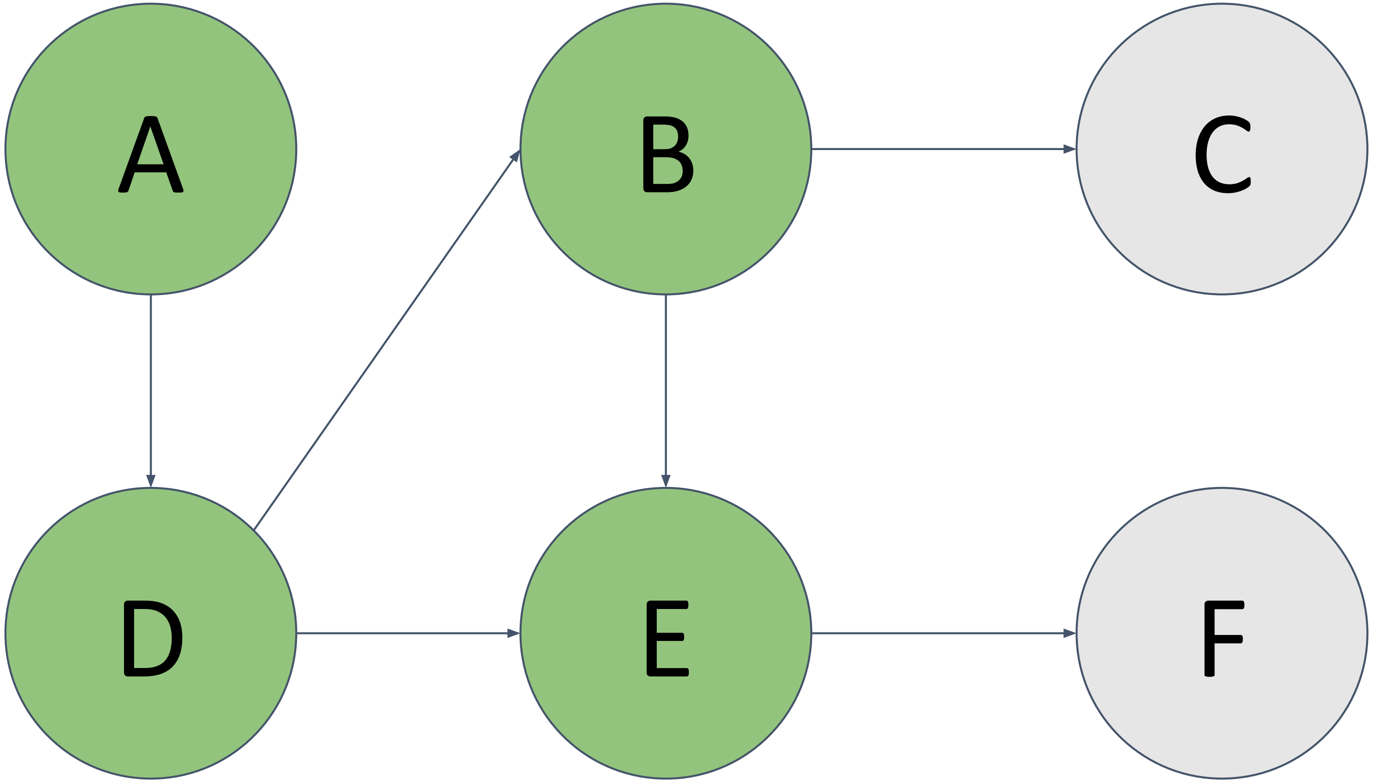
Example BFS (we sort edges in alphabetical order)



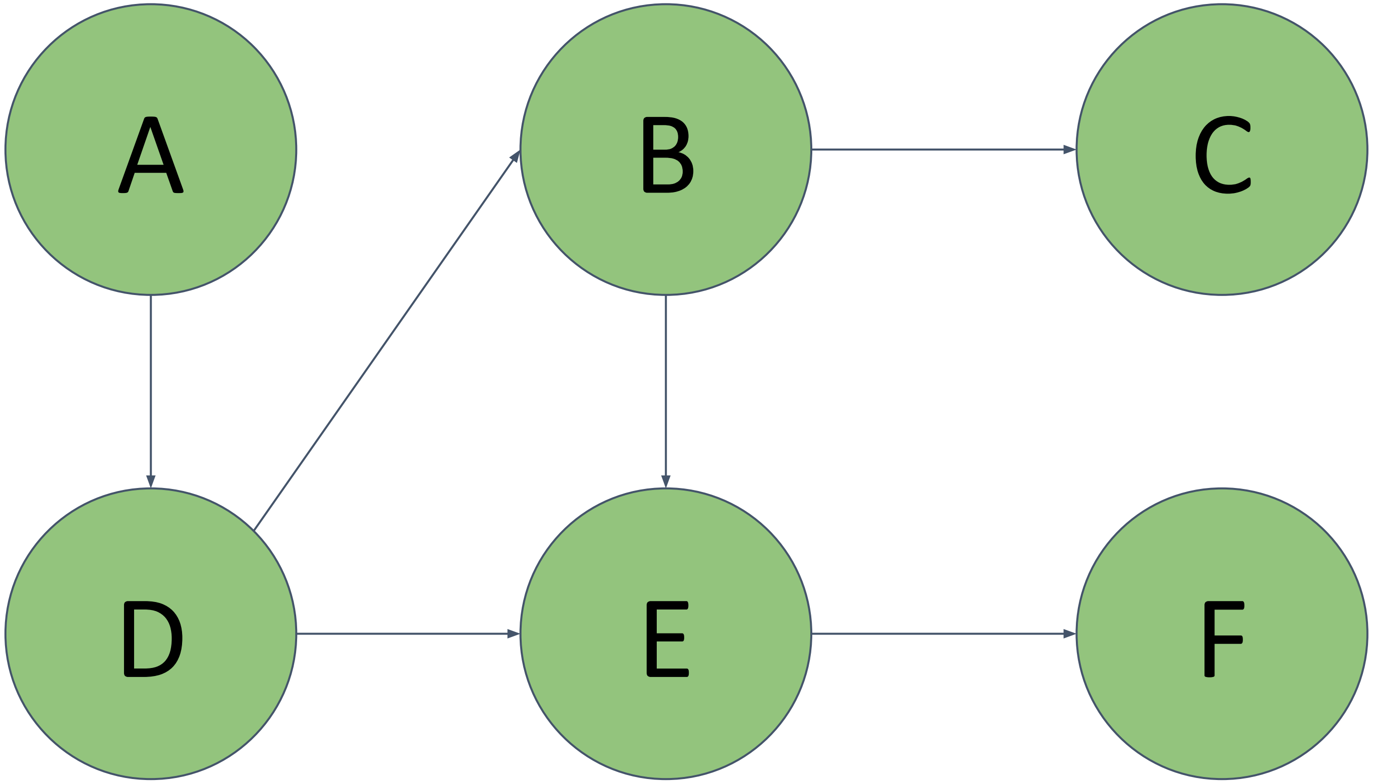
Example BFS (we sort edges in alphabetical order)



Example BFS (we sort edges in alphabetical order)

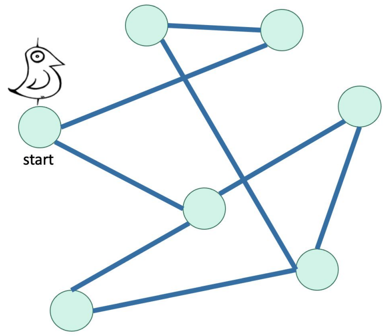


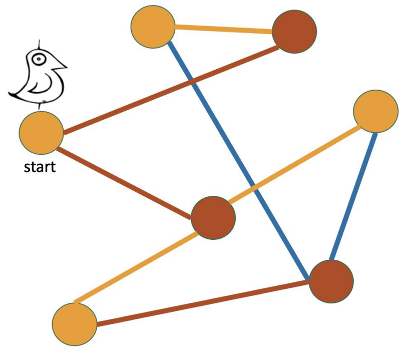
Example BFS (we sort edges in alphabetical order)



What can we do with BFS?

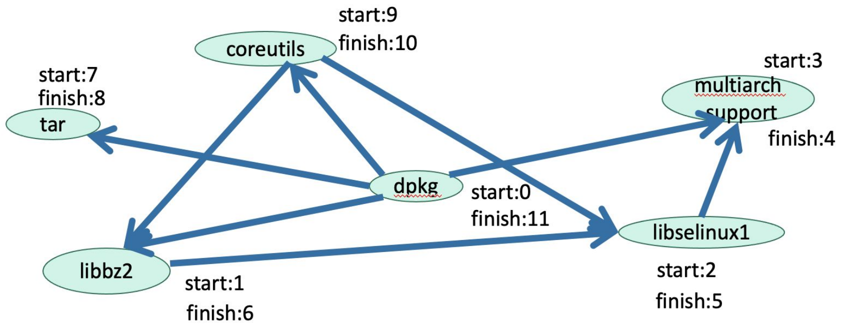
- Find all vertices reachable from the start vertex
- Find the shortest paths from a vertex to all the others
- Check if a graph is bipartite graph.
    - Color the levels of the BFS tree in alternating colors.
    - If you never color two connected nodes the same color, then it is bipartite.
    - Otherwise, it's not.

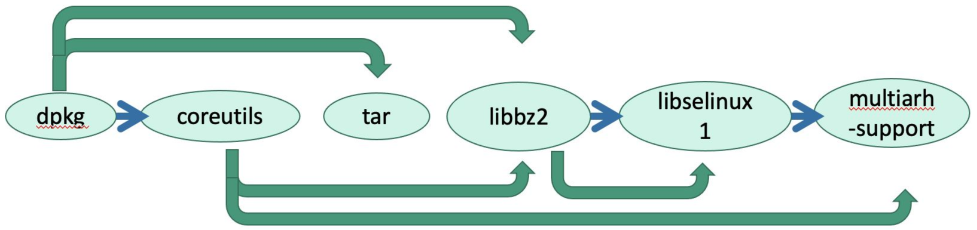




What can we do with DFS?

- Find all vertices reachable from the start vertex
- Topological sort for directed graph without any cycle (DAG)
    - In reverse order of finishing time in DFS!
- Both algorithms run in O(|V| + |E|).





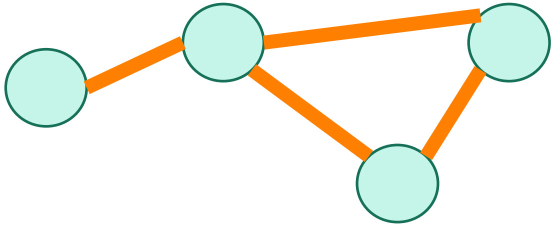
Strongly Connected Components

Please do not confuse them!

Strongly *vs* Connected Components

***Un*directed graphs**

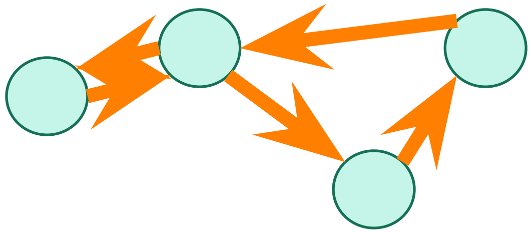
Connected Components (CC)



Find with BFS/DFS

Directed graphs

Strongly Connected Components (**S**CC)



Find with DFS (today)

[Actual algorithm!]

Algorithm

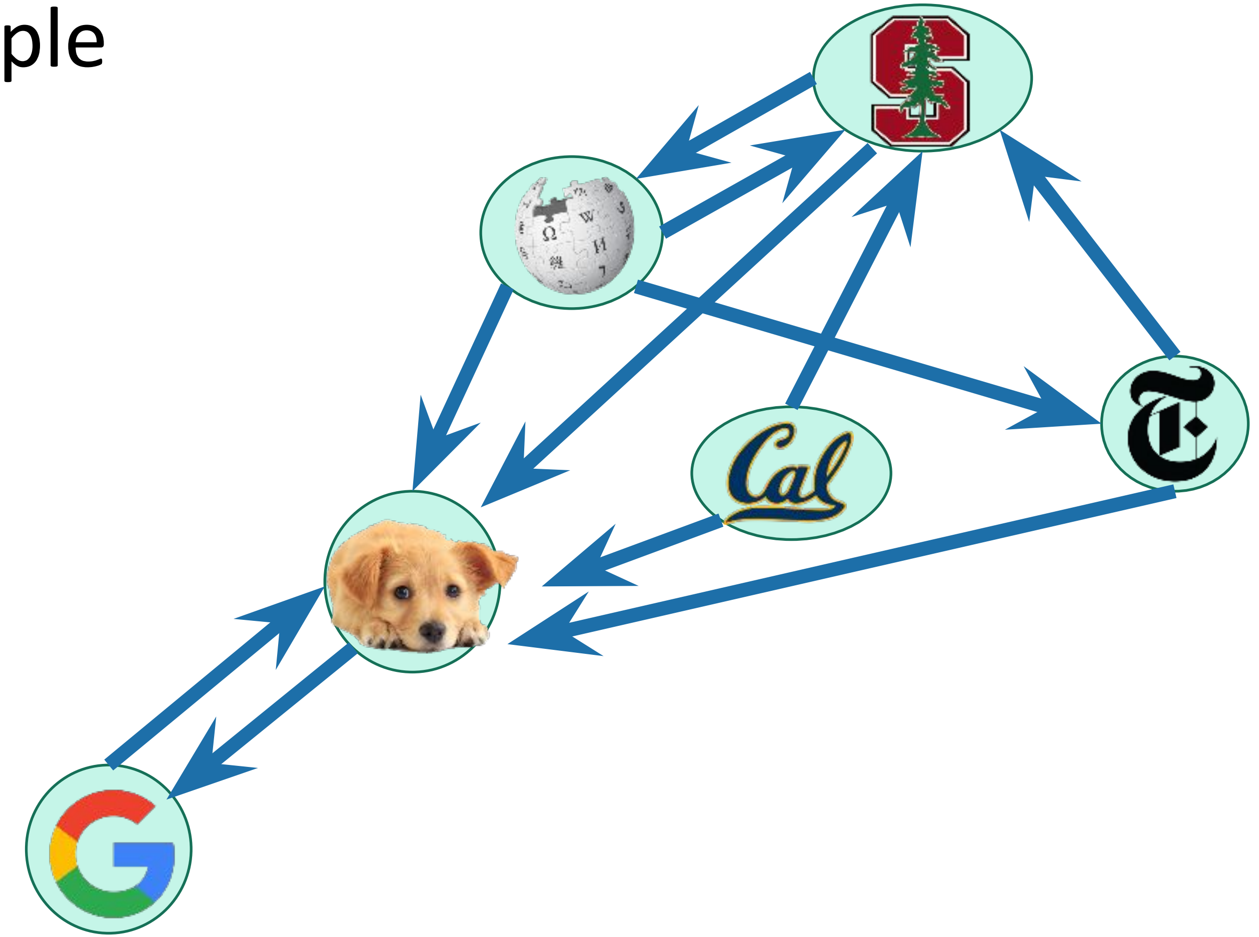
Running time: O(n + m)

- Run DFS.
  - Choose starting vertices in any order.
  - Order vertices by reverse finishing times.
- Reverse all the edges in the graph.
- Do DFS again to create **a new DFS forest**.
  - (Using the reverse-finish-time order from the first DFS run.)
- The SCCs are the different trees in the **new DFS forest.**

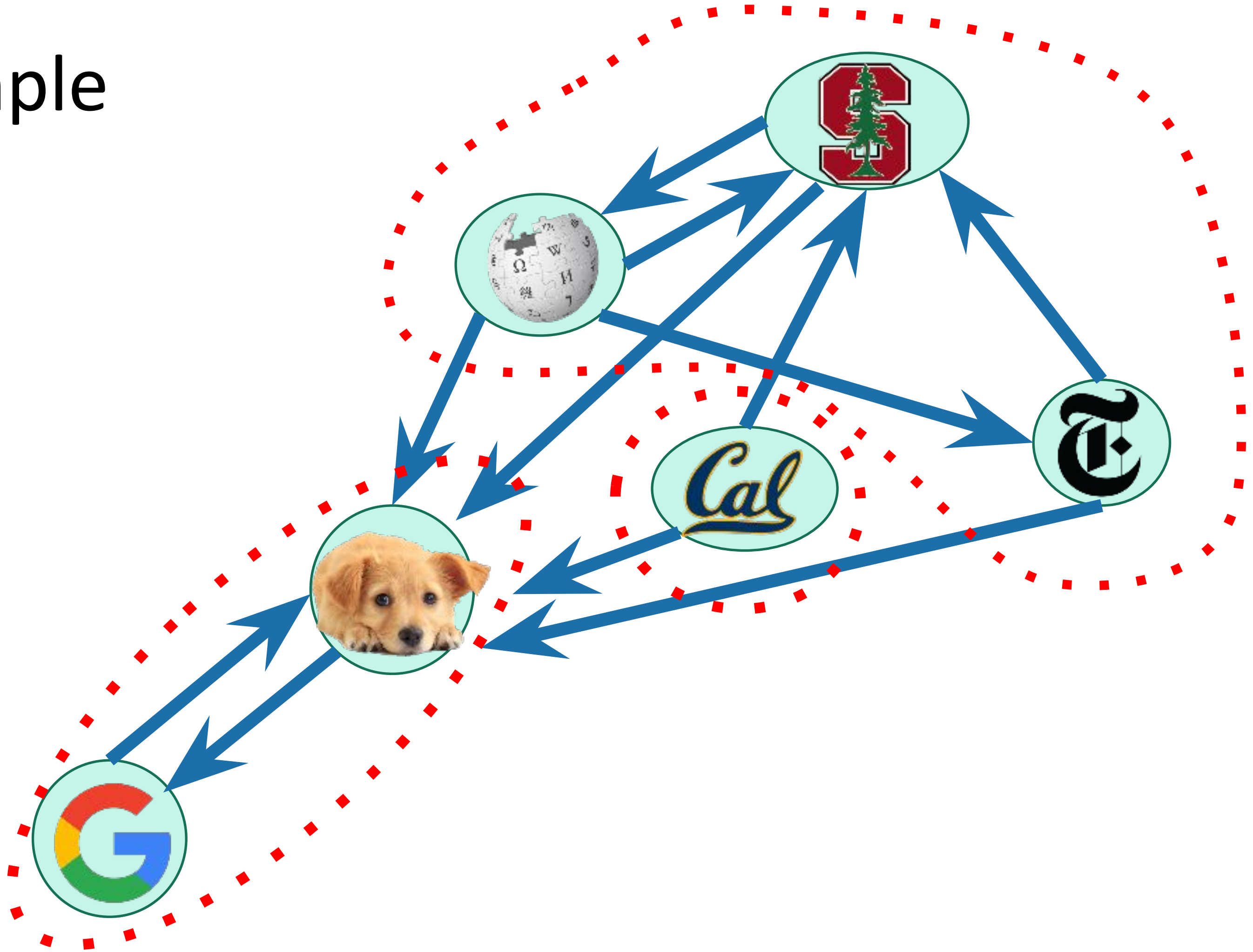
(This is basically "PretendTopoSort")

(This doesn't change SCCs)

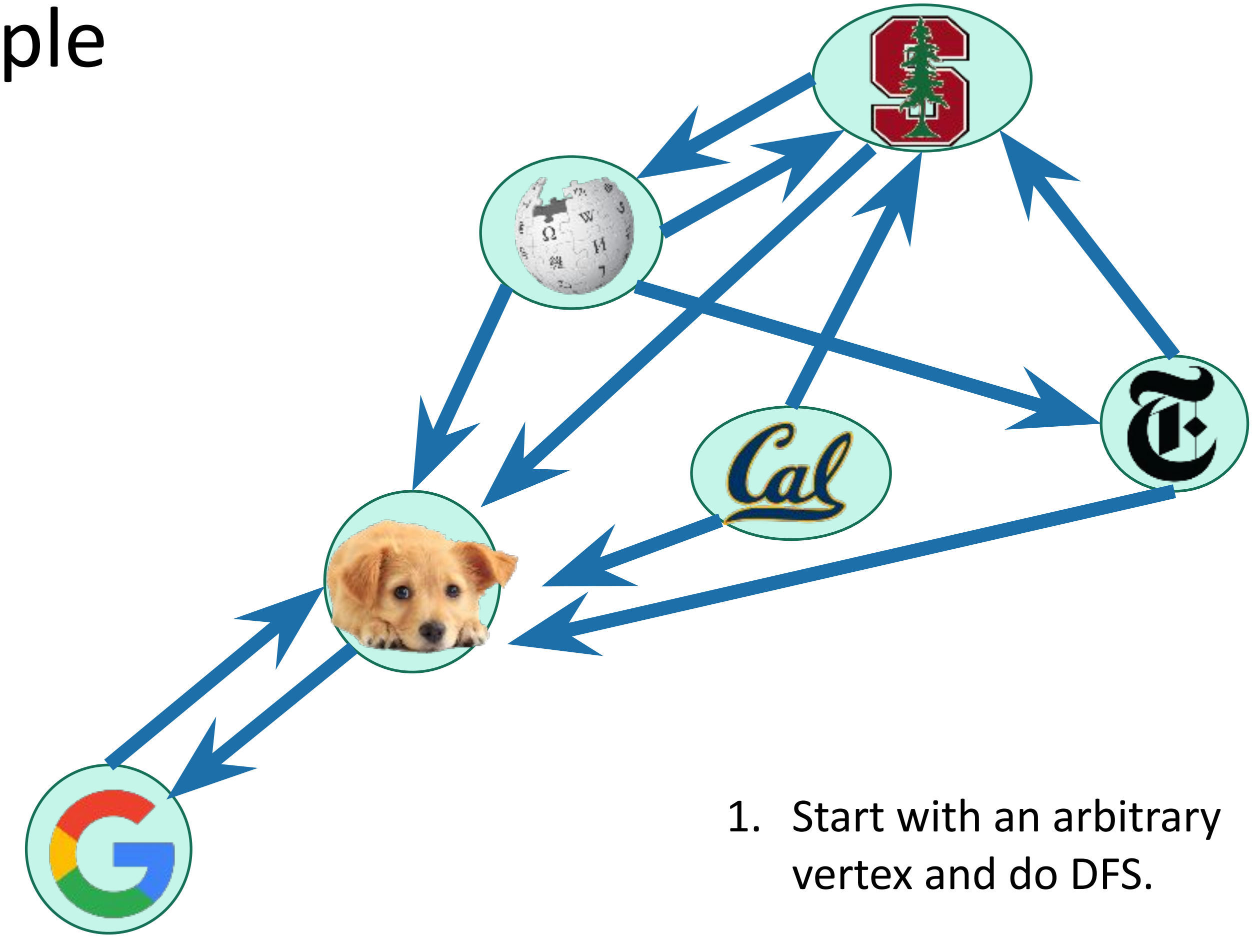
Example



Example

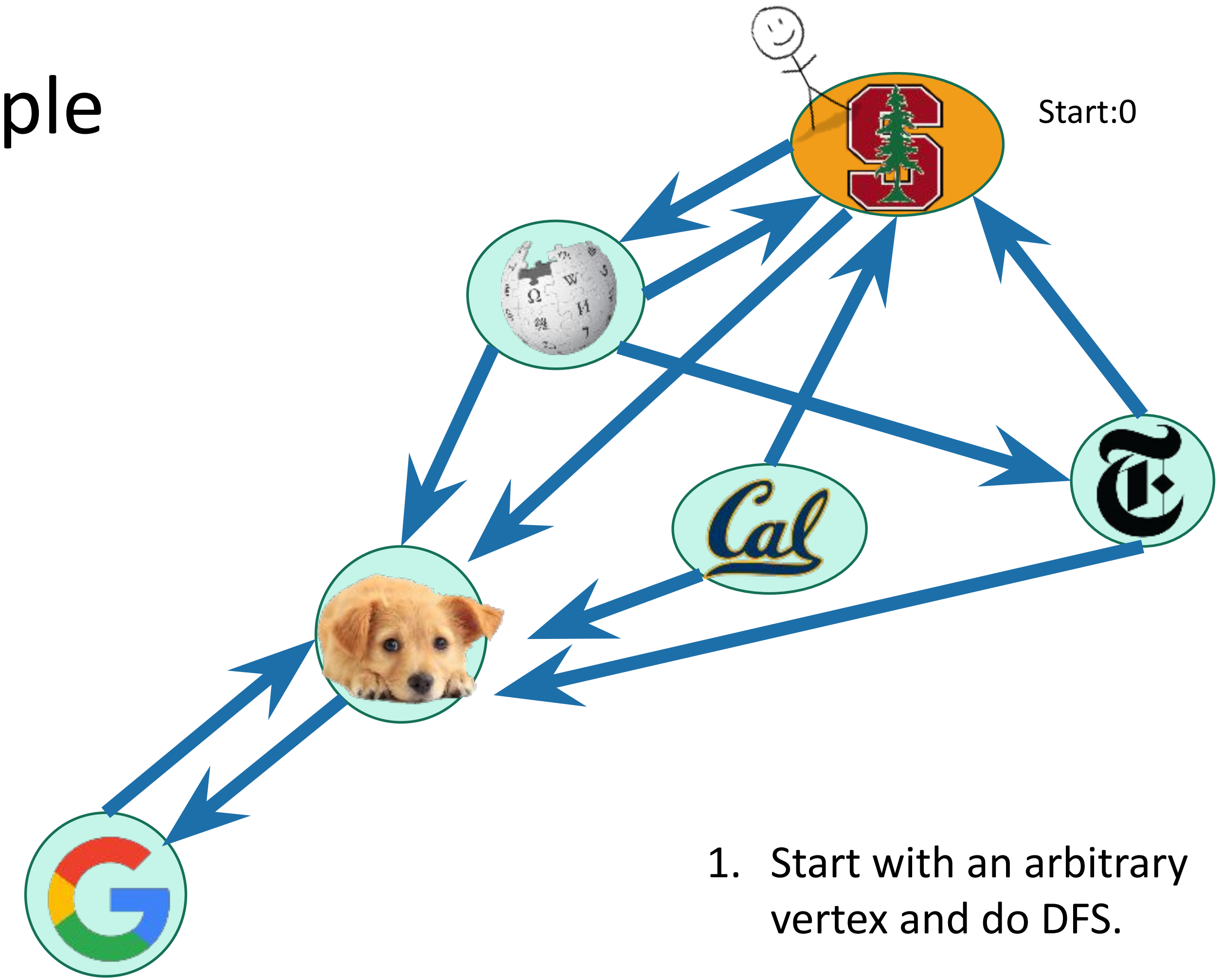


Example



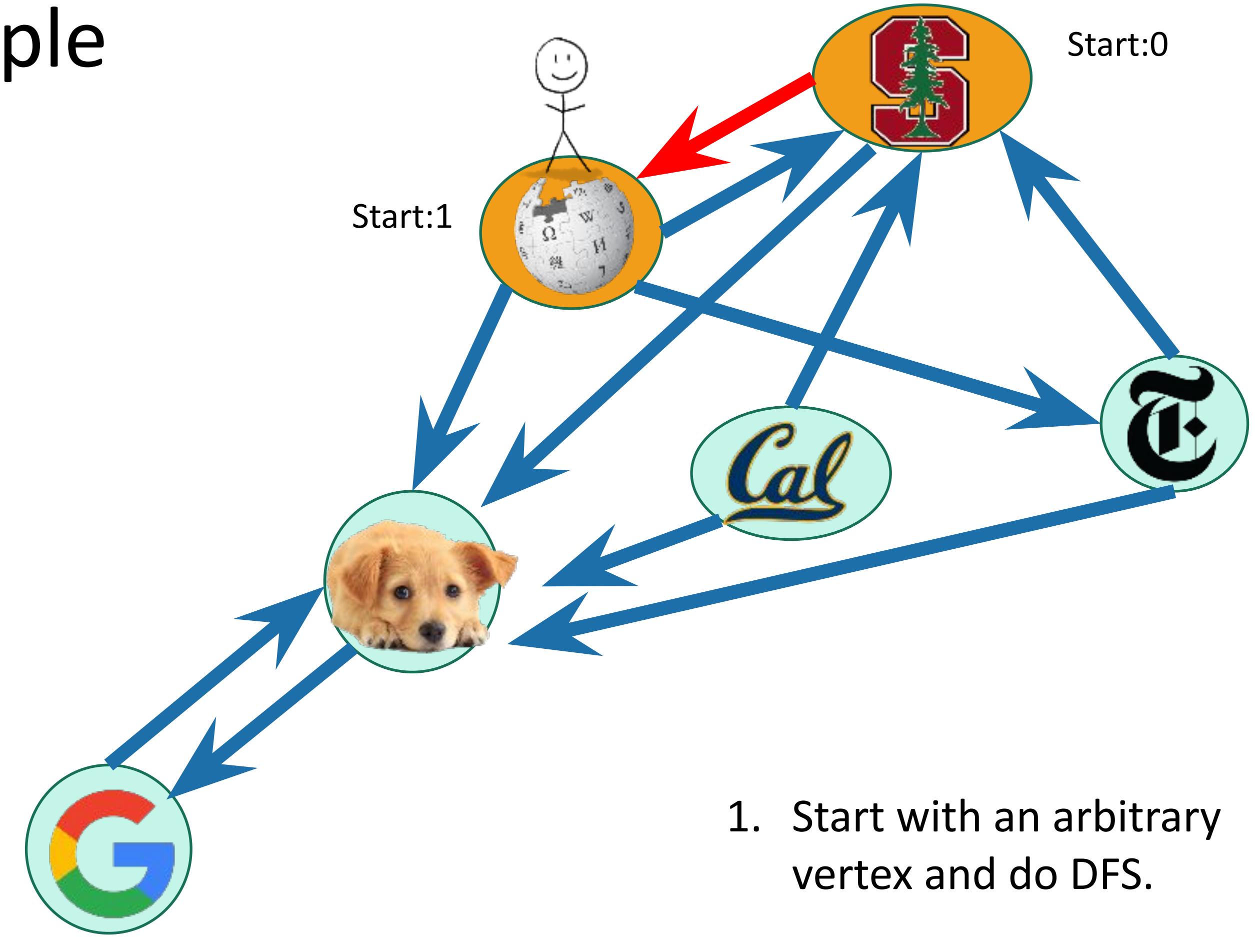
1. Start with an arbitrary vertex and do DFS.

Example



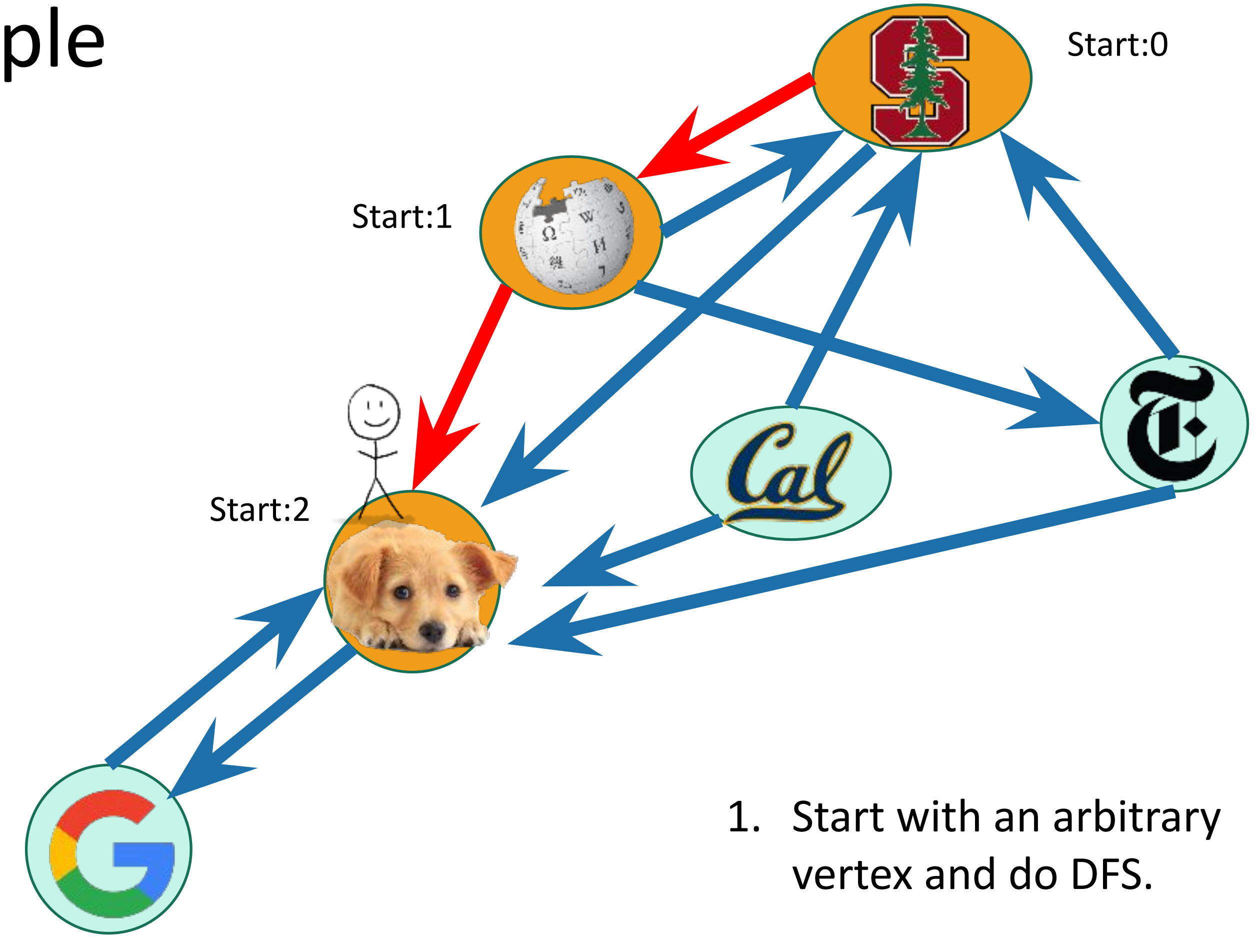
1. Start with an arbitrary vertex and do DFS.

Example



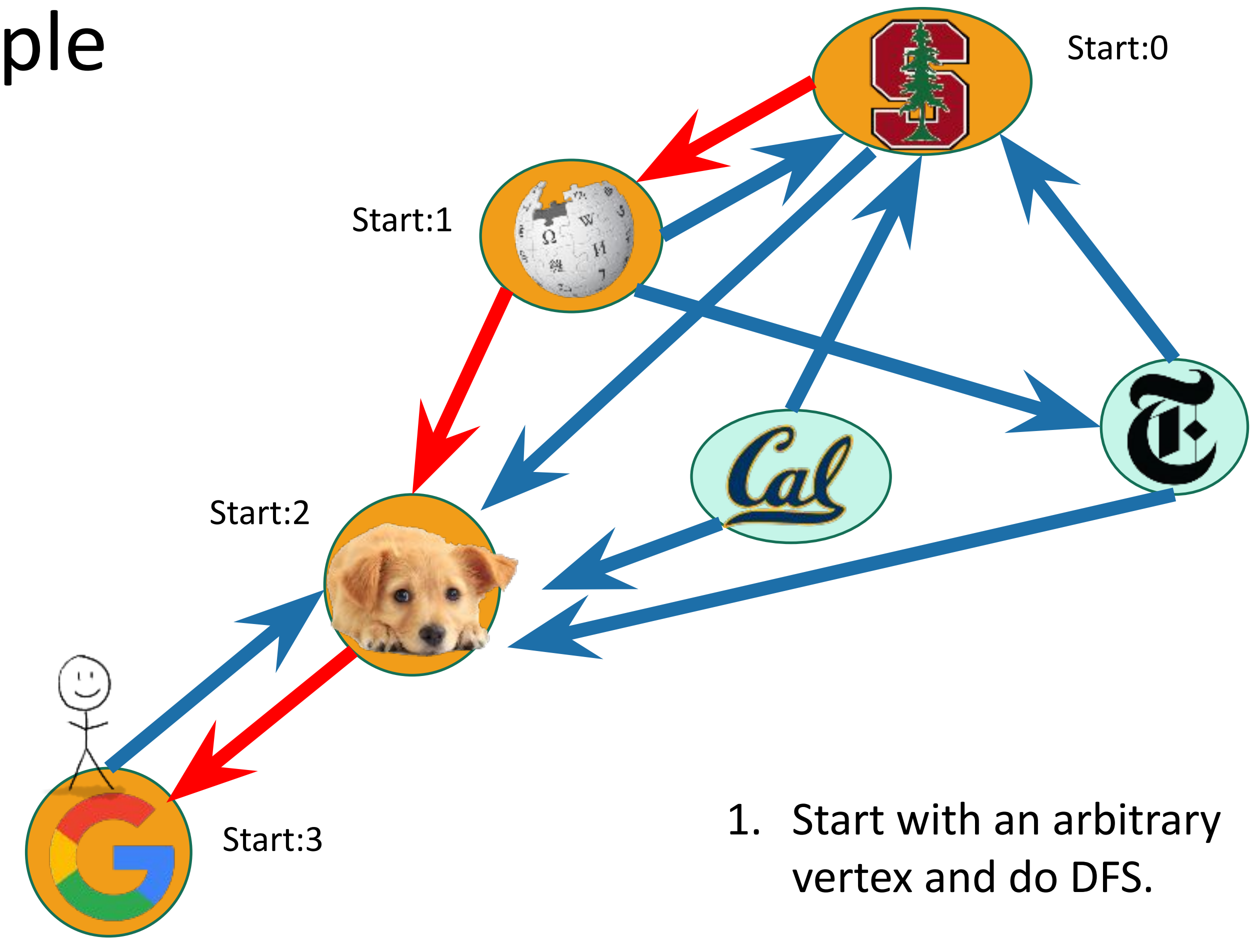
1. Start with an arbitrary vertex and do DFS.

Example



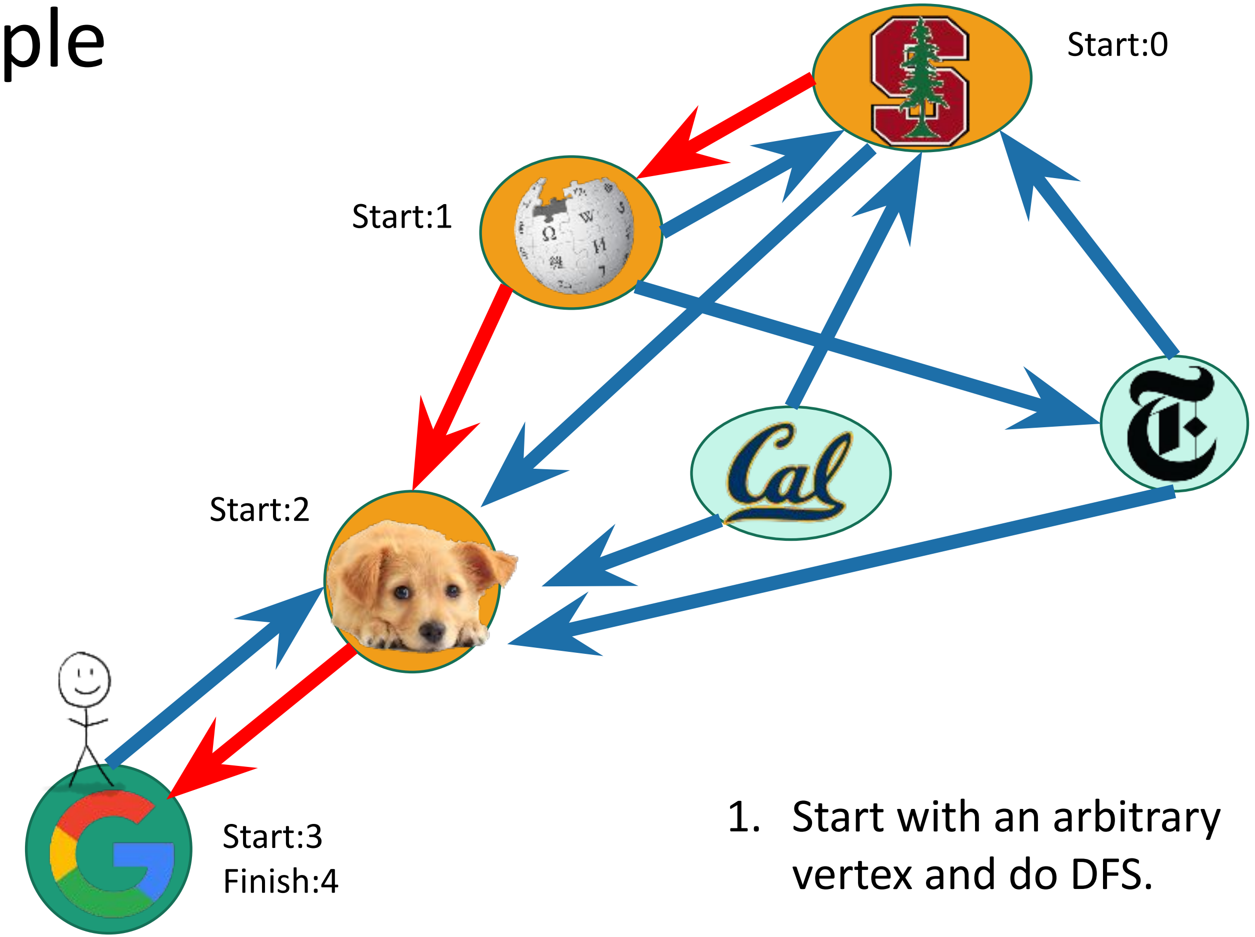
1. Start with an arbitrary vertex and do DFS.

Example



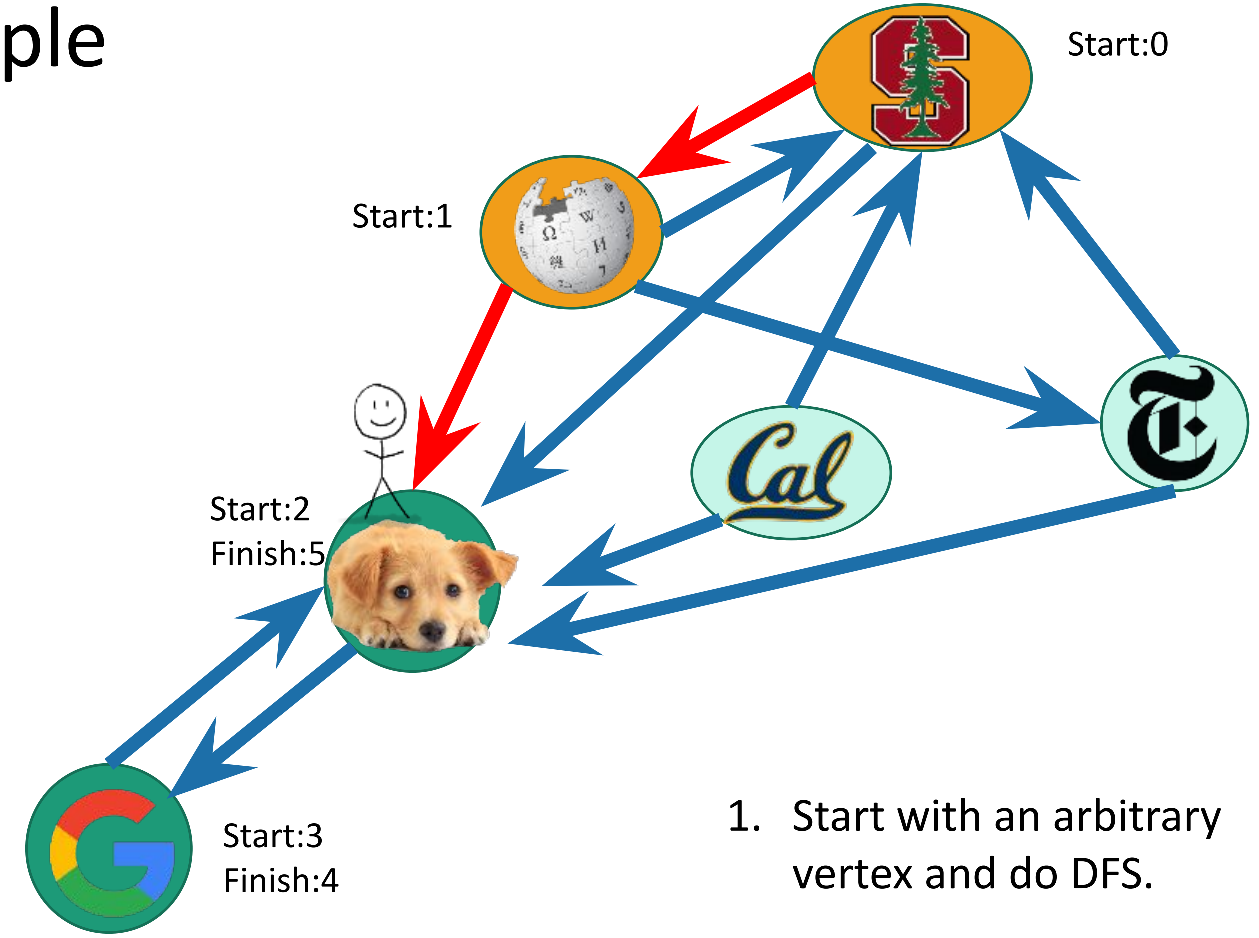
1. Start with an arbitrary vertex and do DFS.

Example



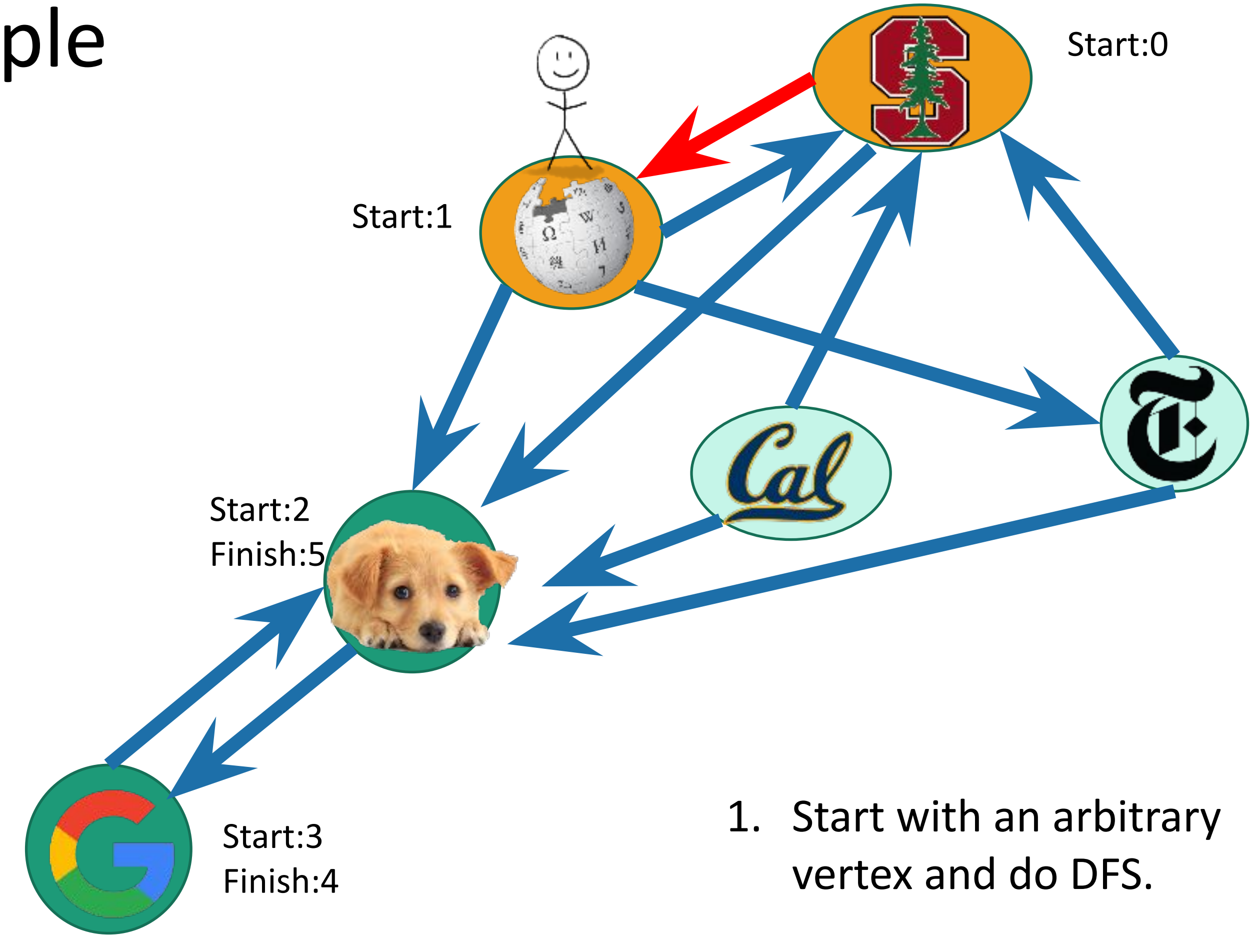
1. Start with an arbitrary vertex and do DFS.

Example



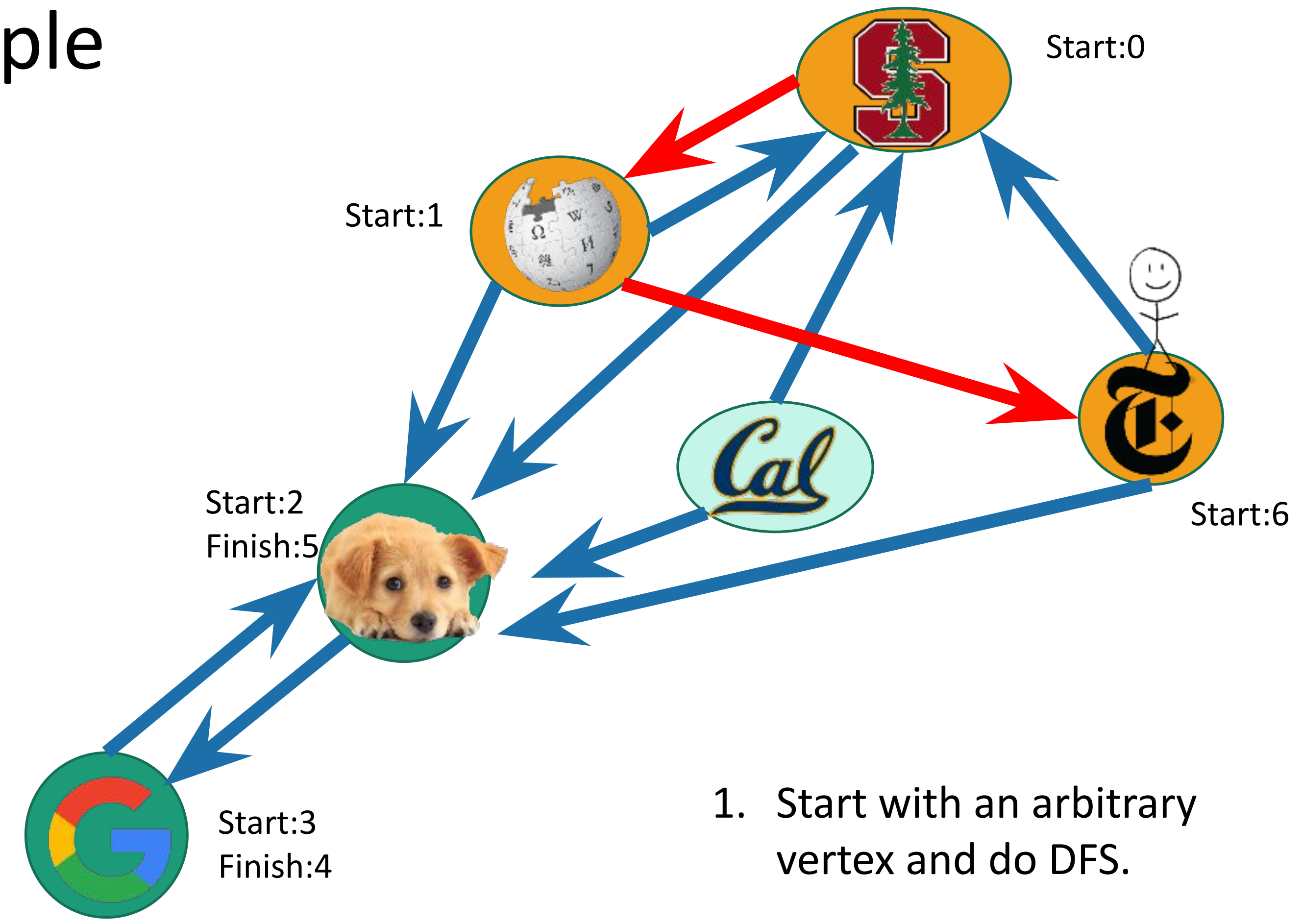
1. Start with an arbitrary vertex and do DFS.

Example



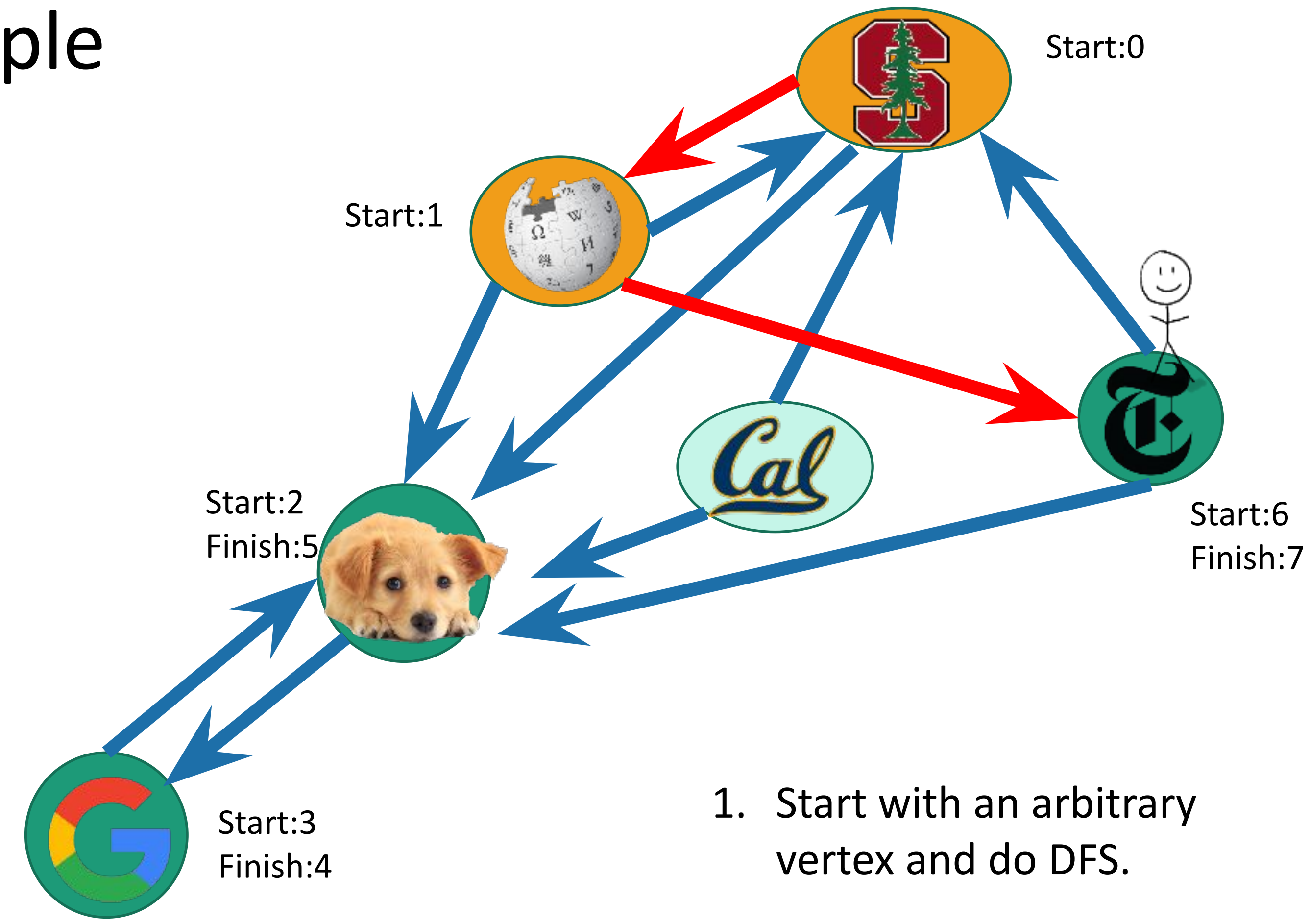
1. Start with an arbitrary vertex and do DFS.

Example



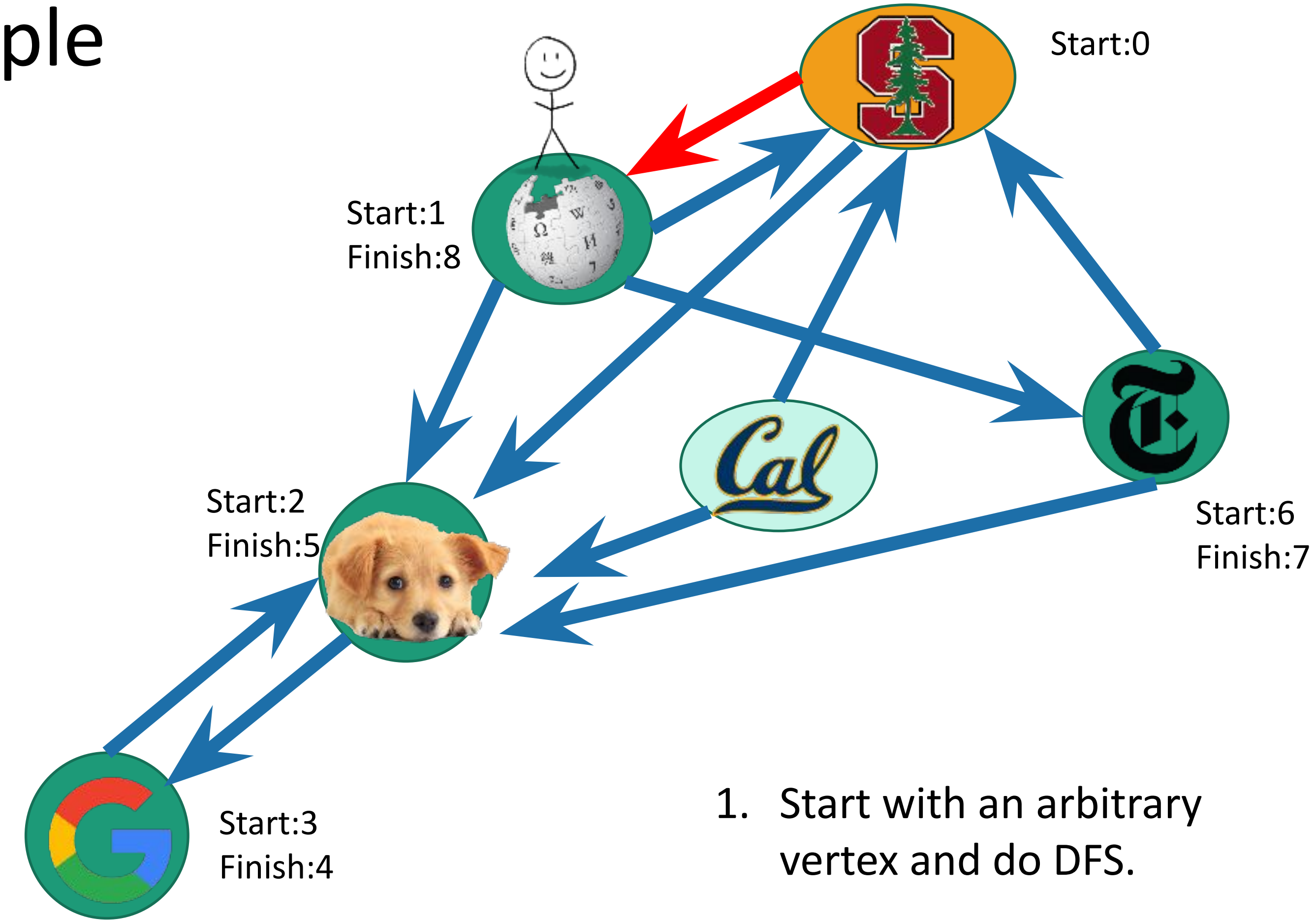
1. Start with an arbitrary vertex and do DFS.

Example



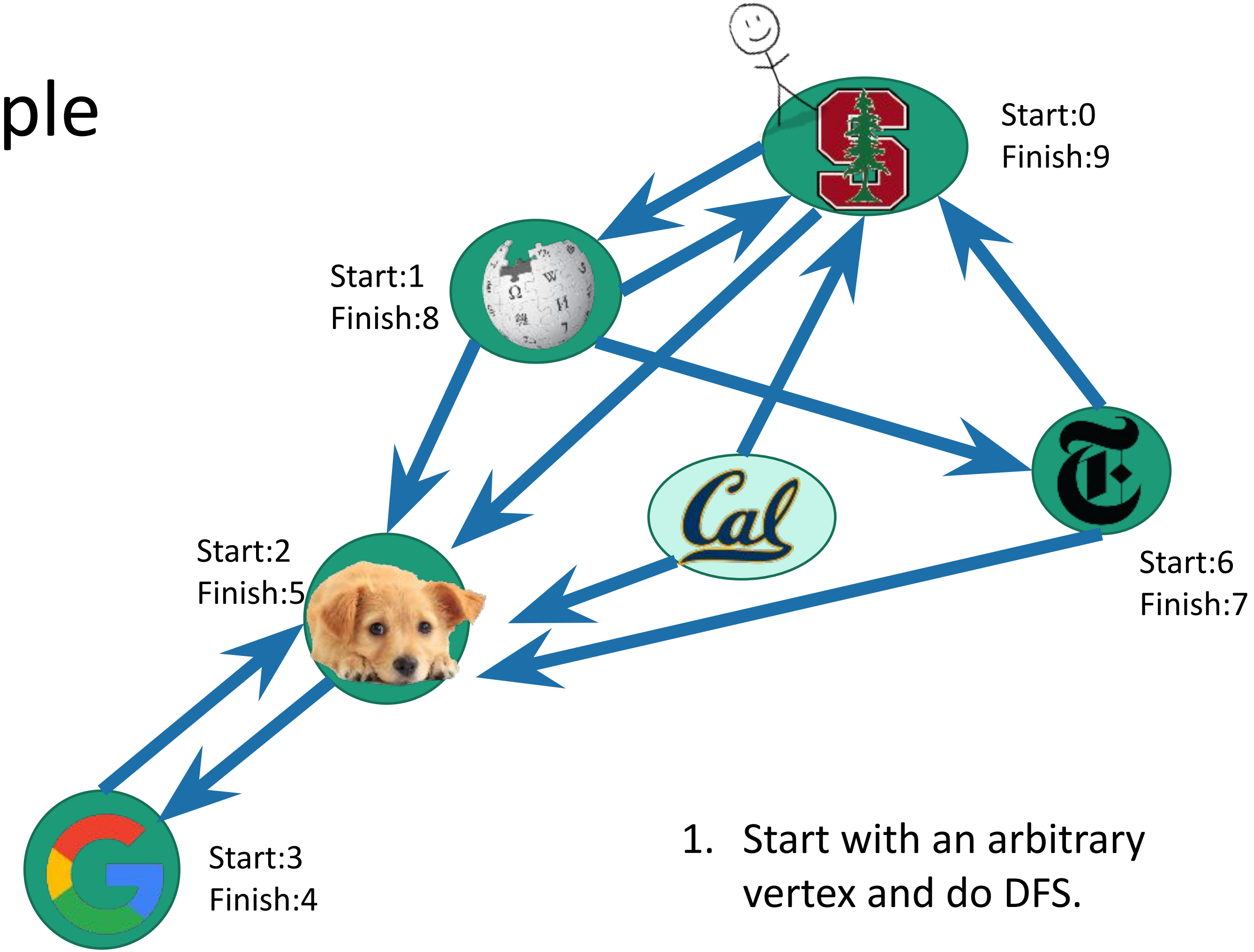
1. Start with an arbitrary vertex and do DFS.

Example



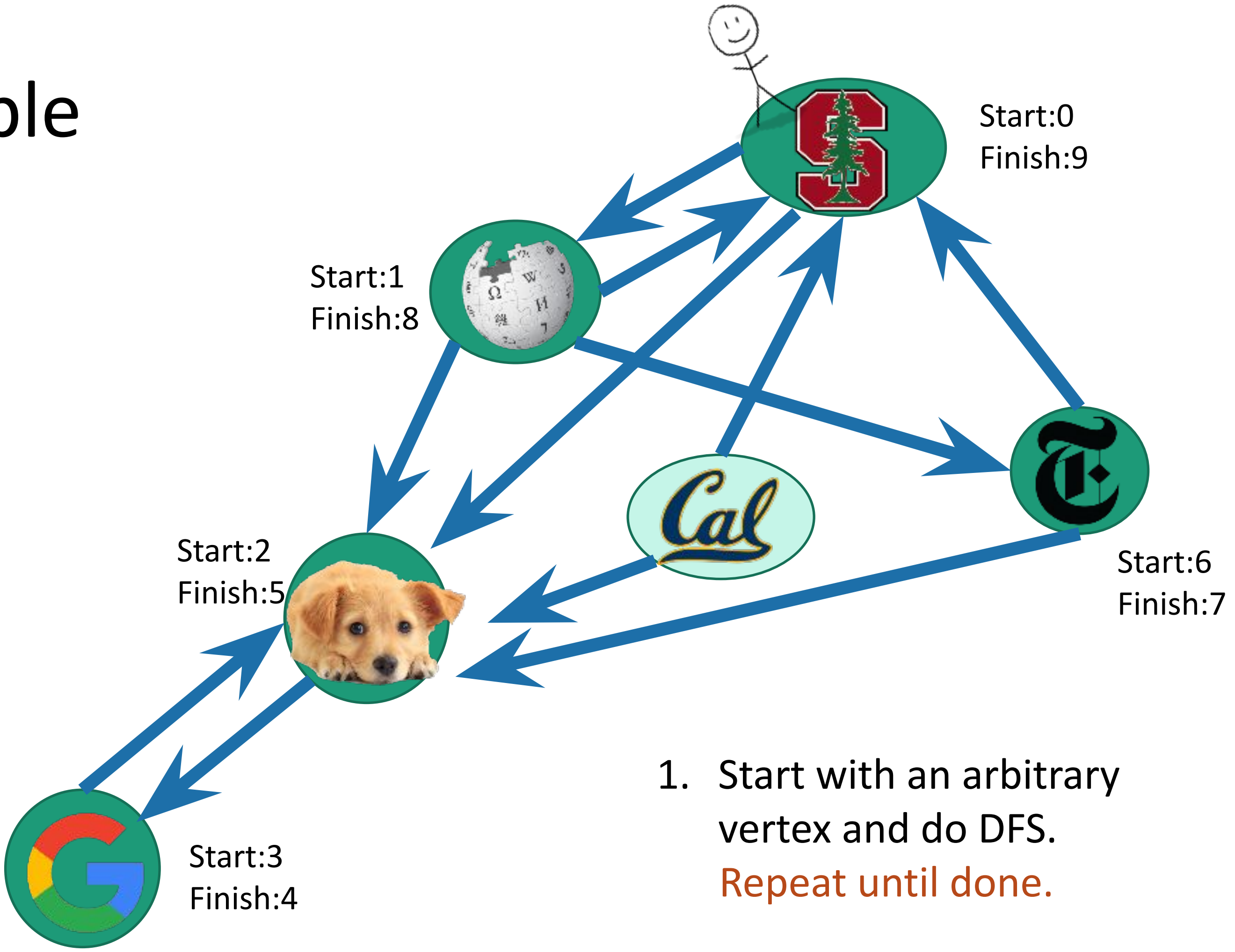
1. Start with an arbitrary vertex and do DFS.

Example



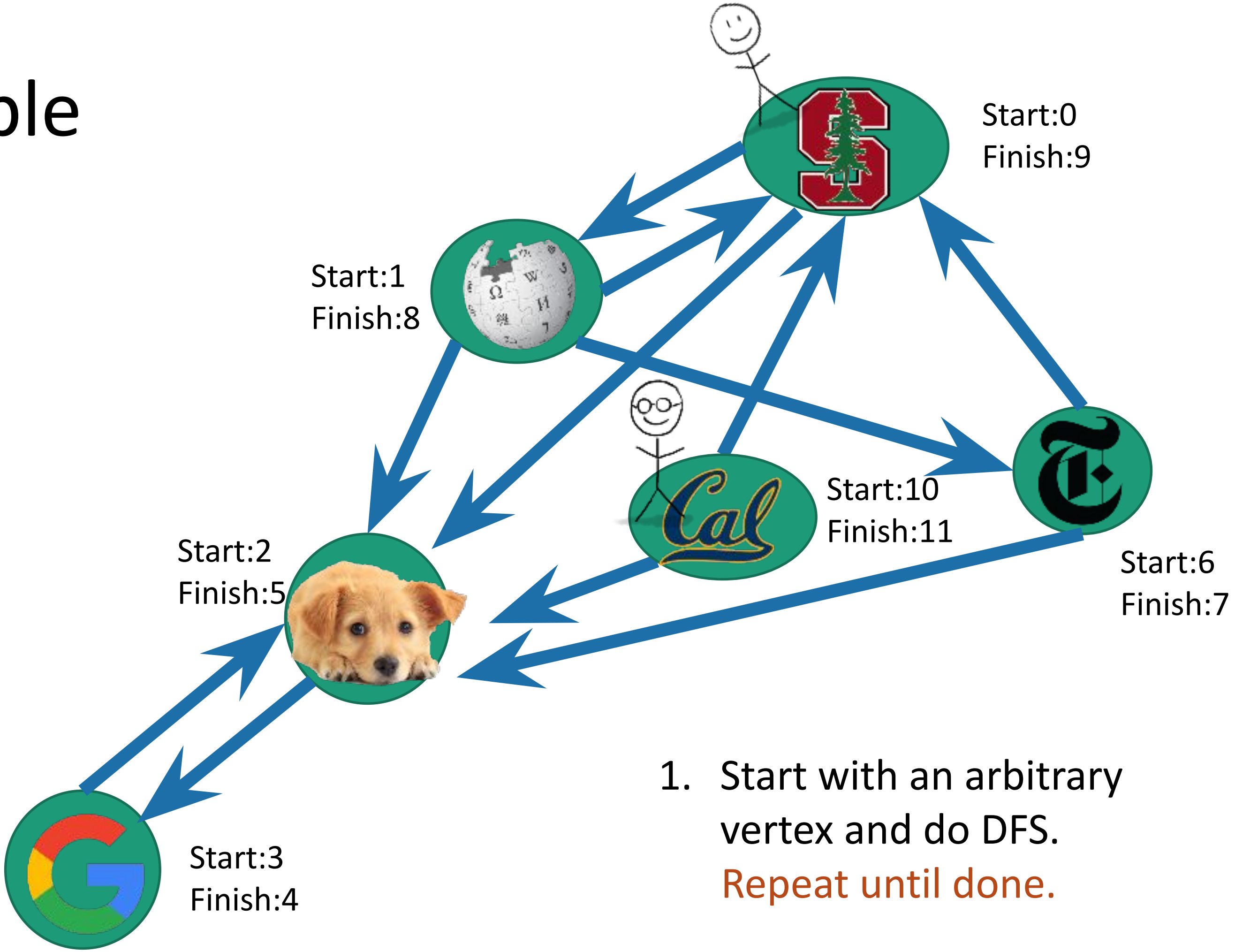
1. Start with an arbitrary vertex and do DFS.

Example



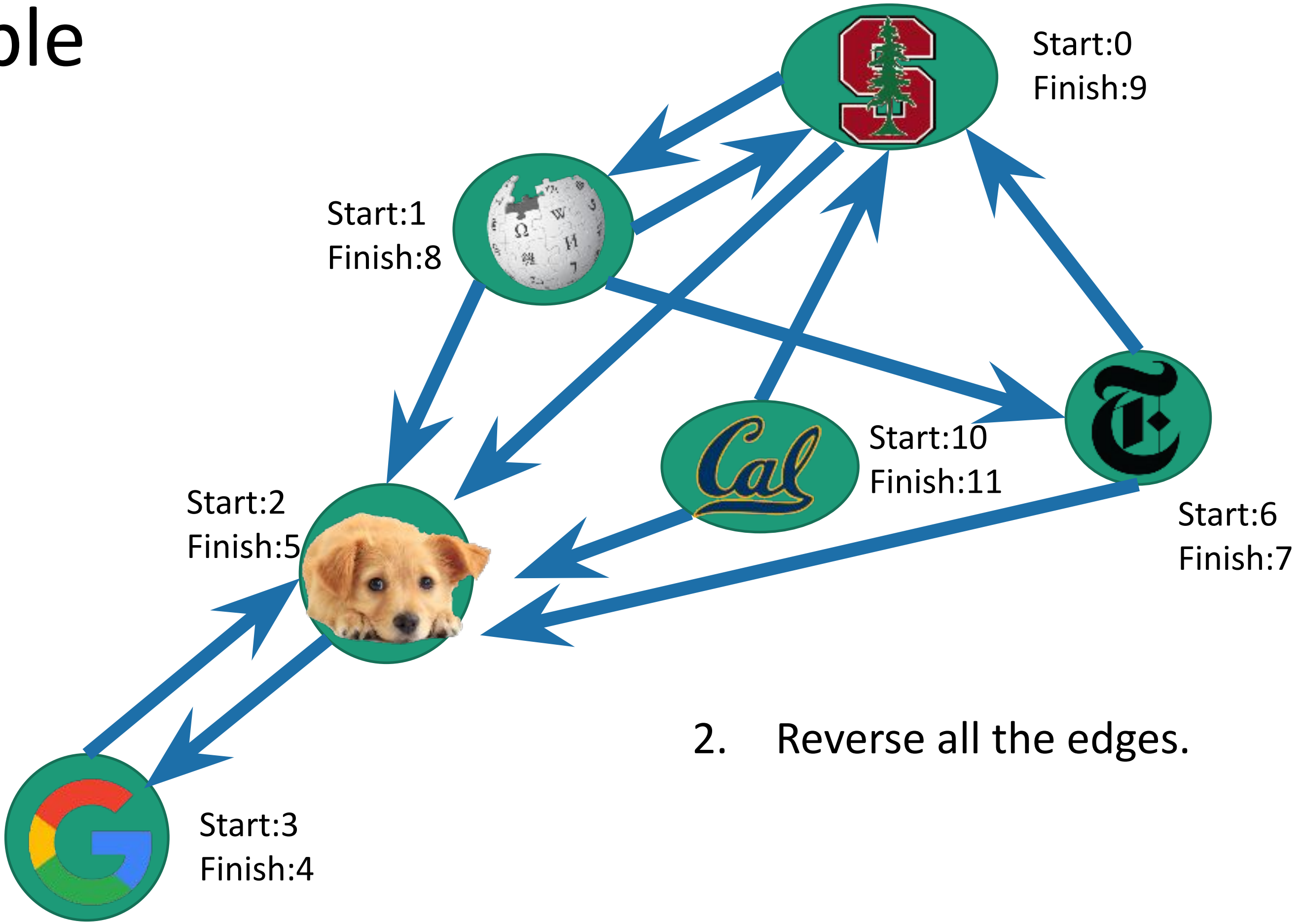
1. Start with an arbitrary vertex and do DFS. Repeat until done.

Example



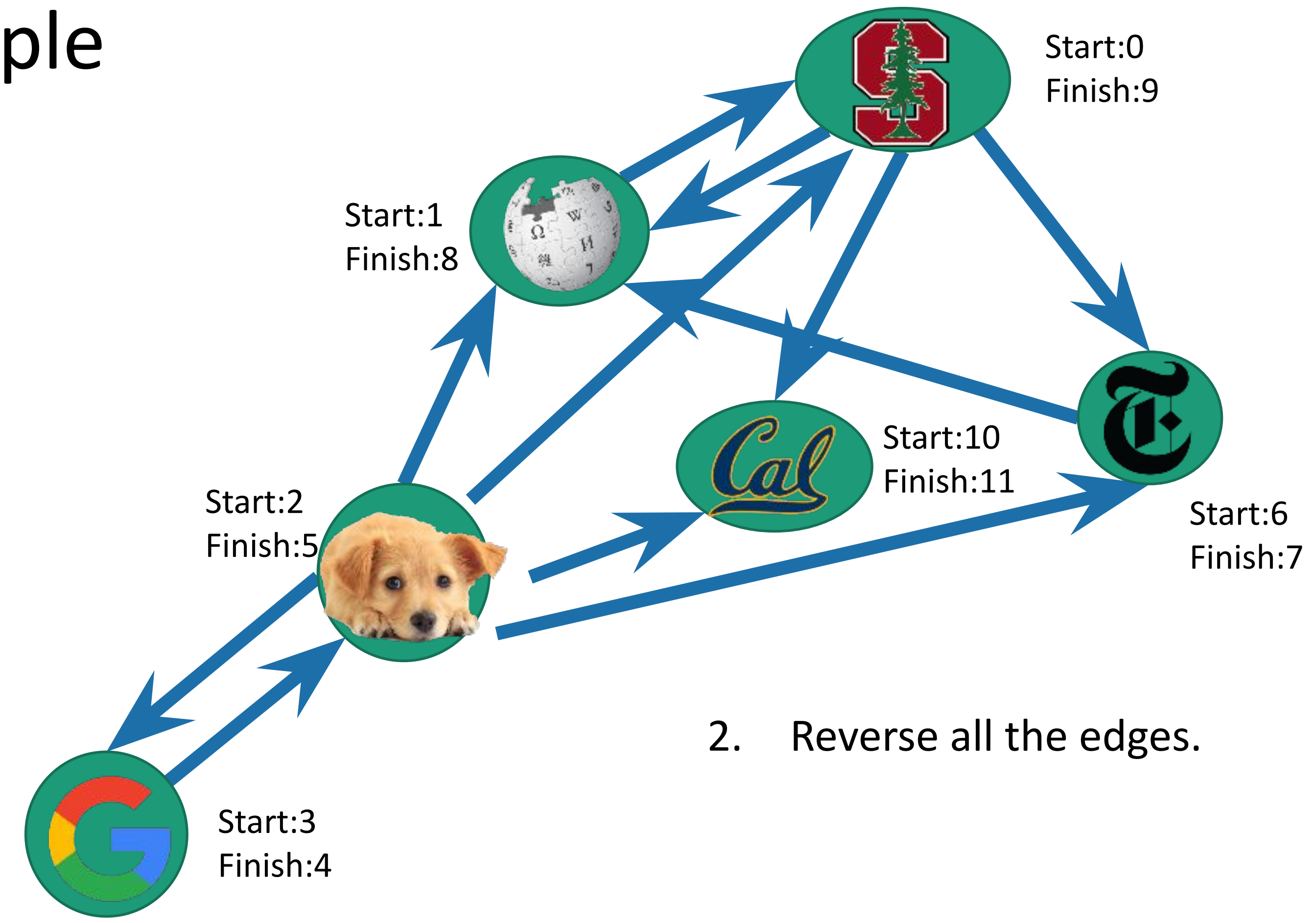
1. Start with an arbitrary vertex and do DFS. Repeat until done.

Example



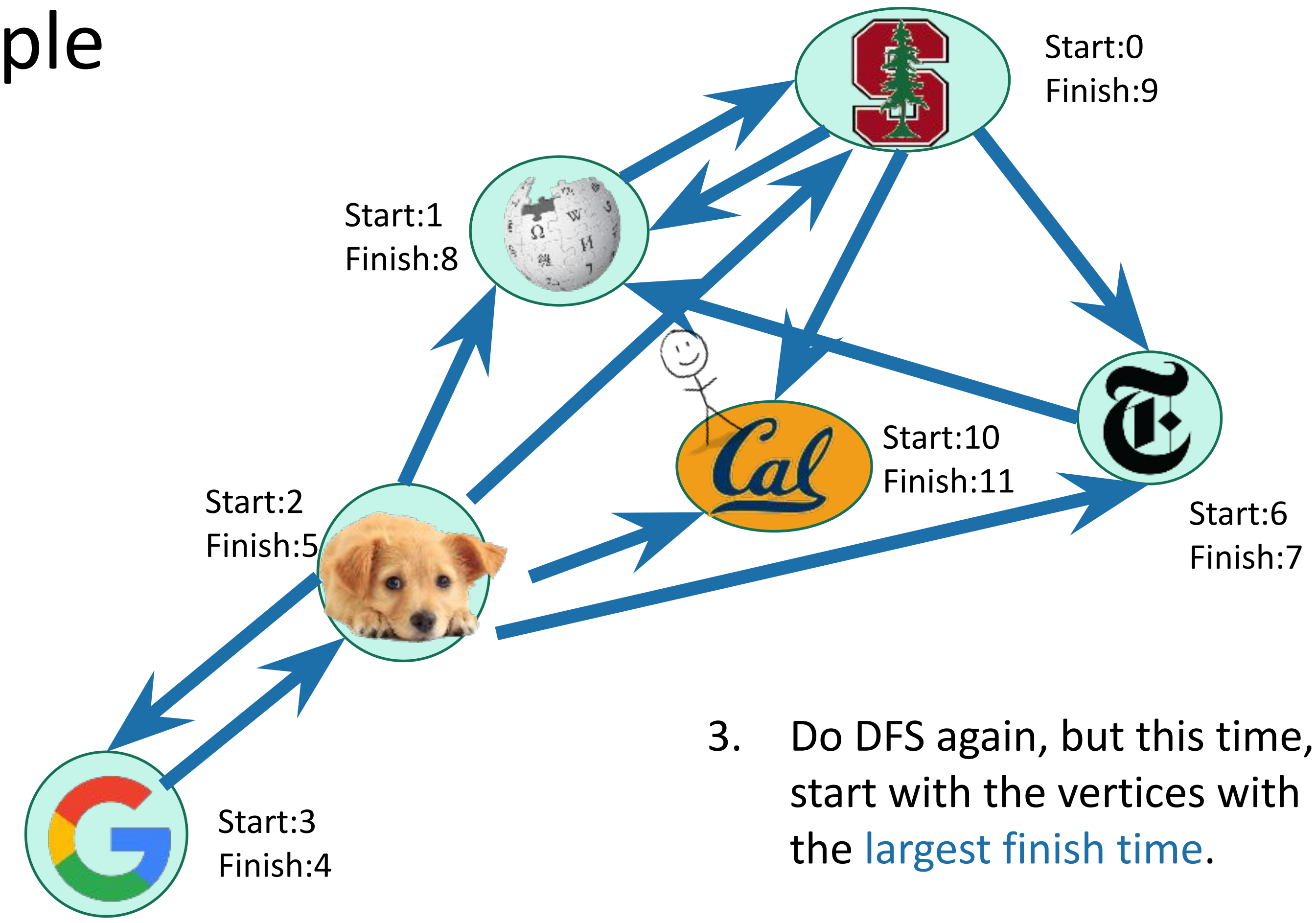
2. Reverse all the edges.

Example



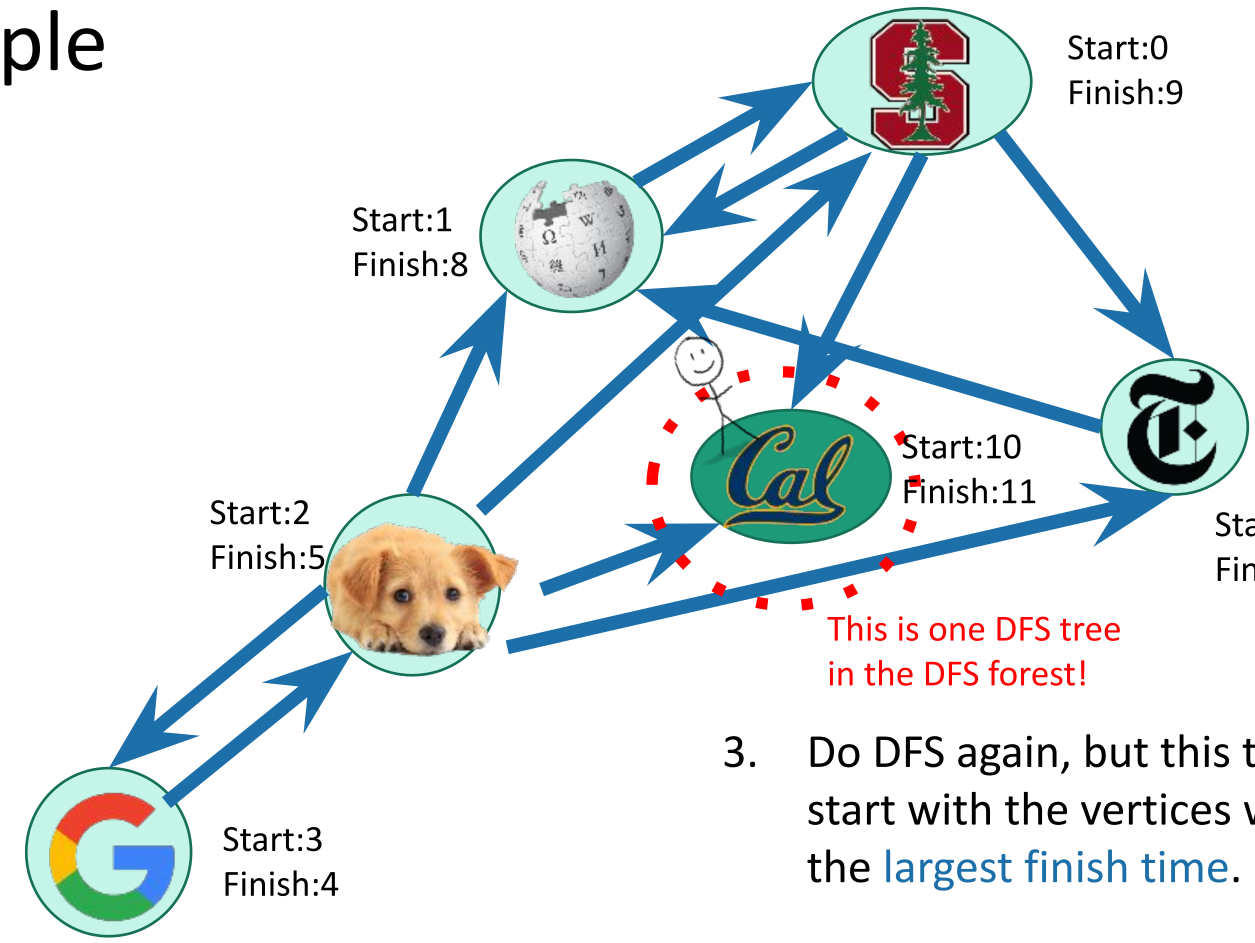
2. Reverse all the edges.

Example



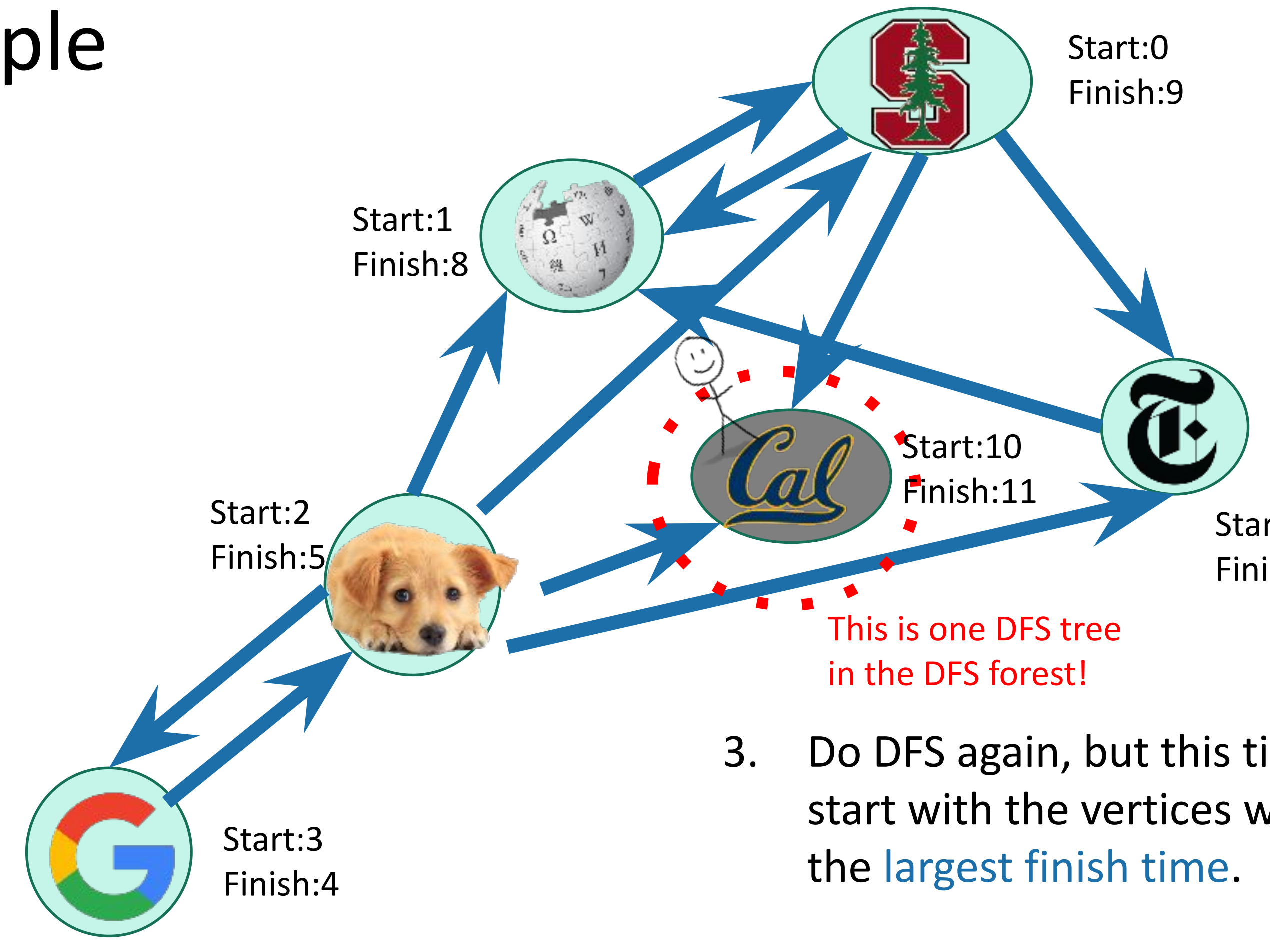
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



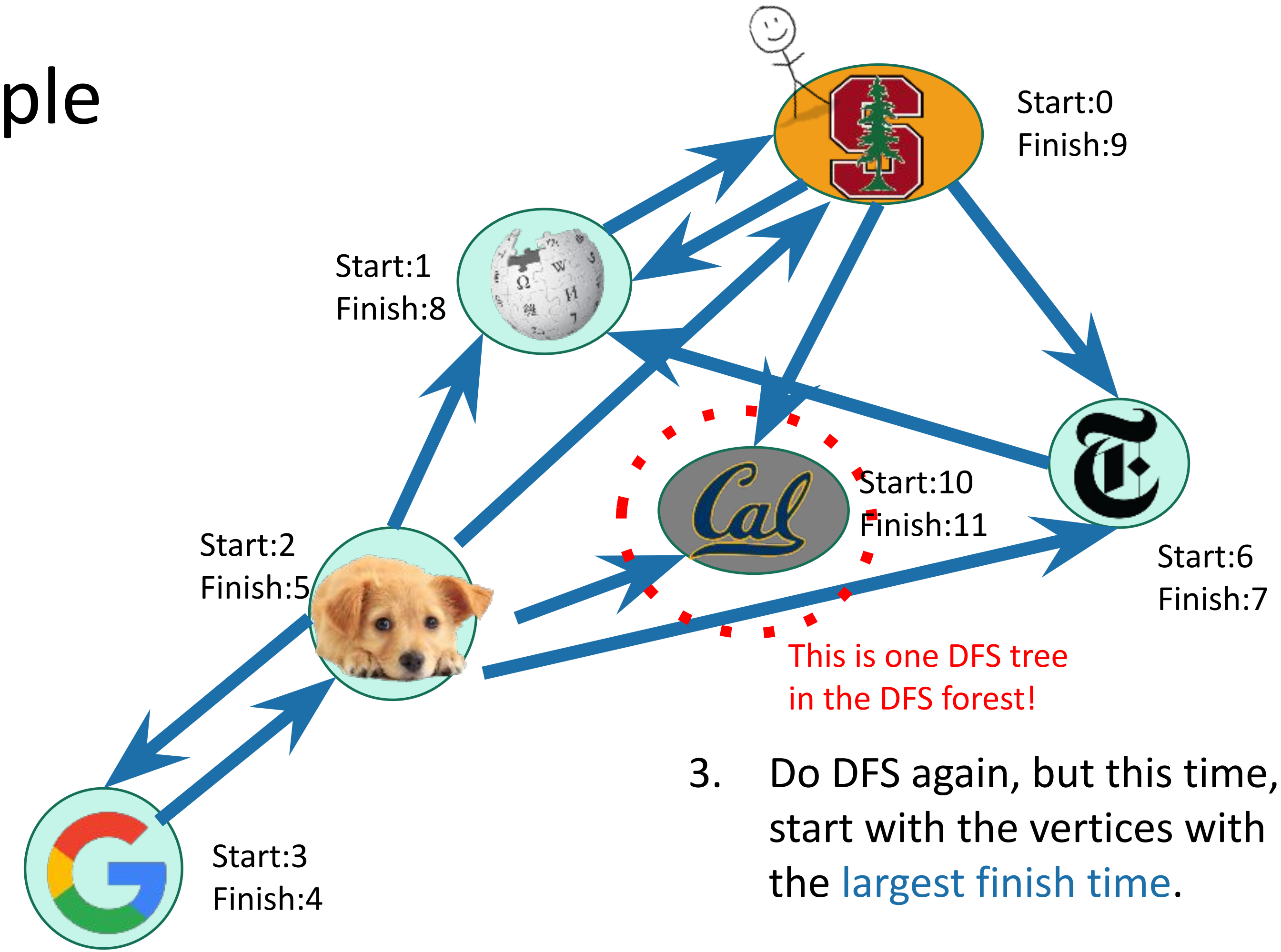
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



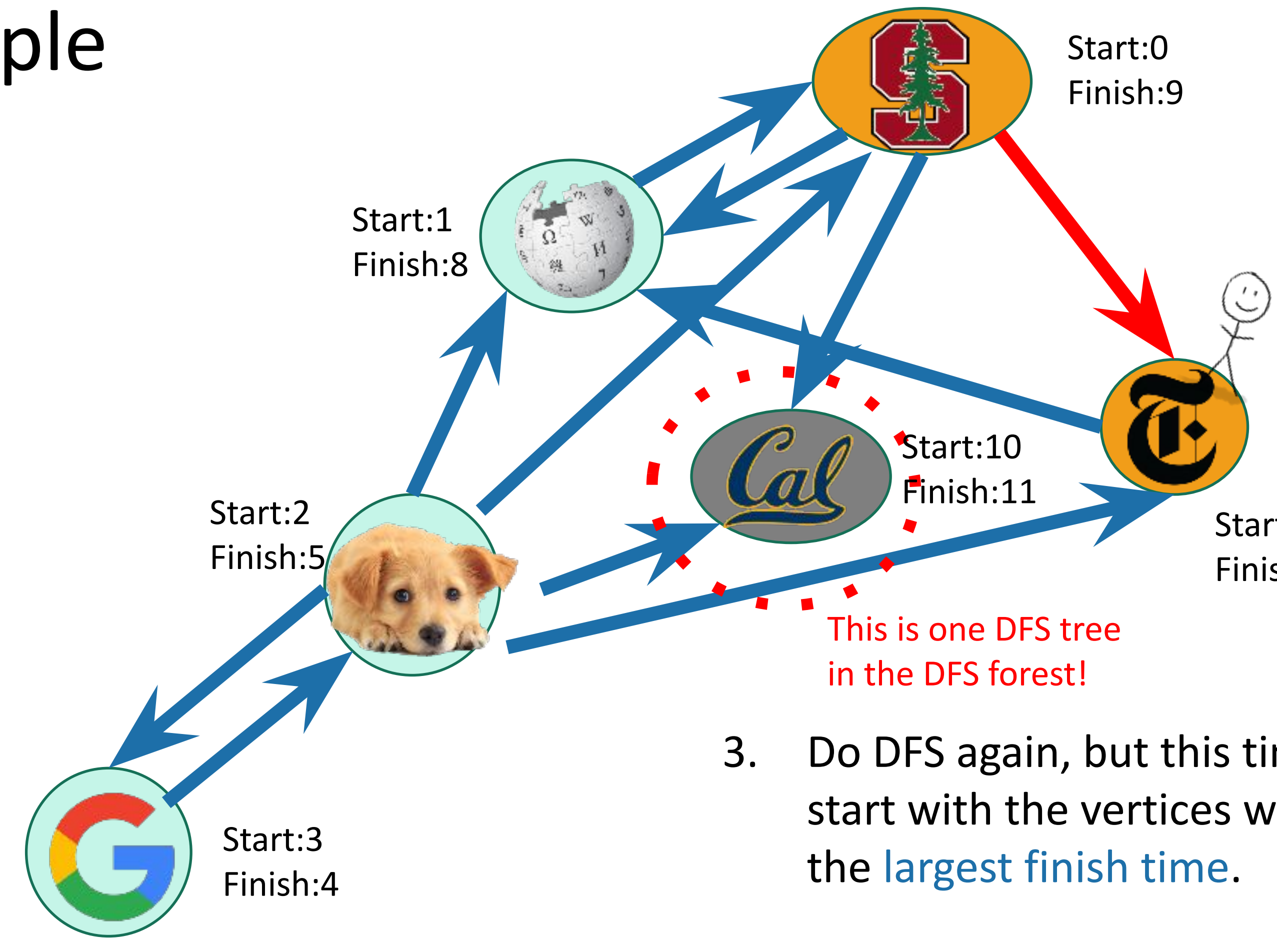
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



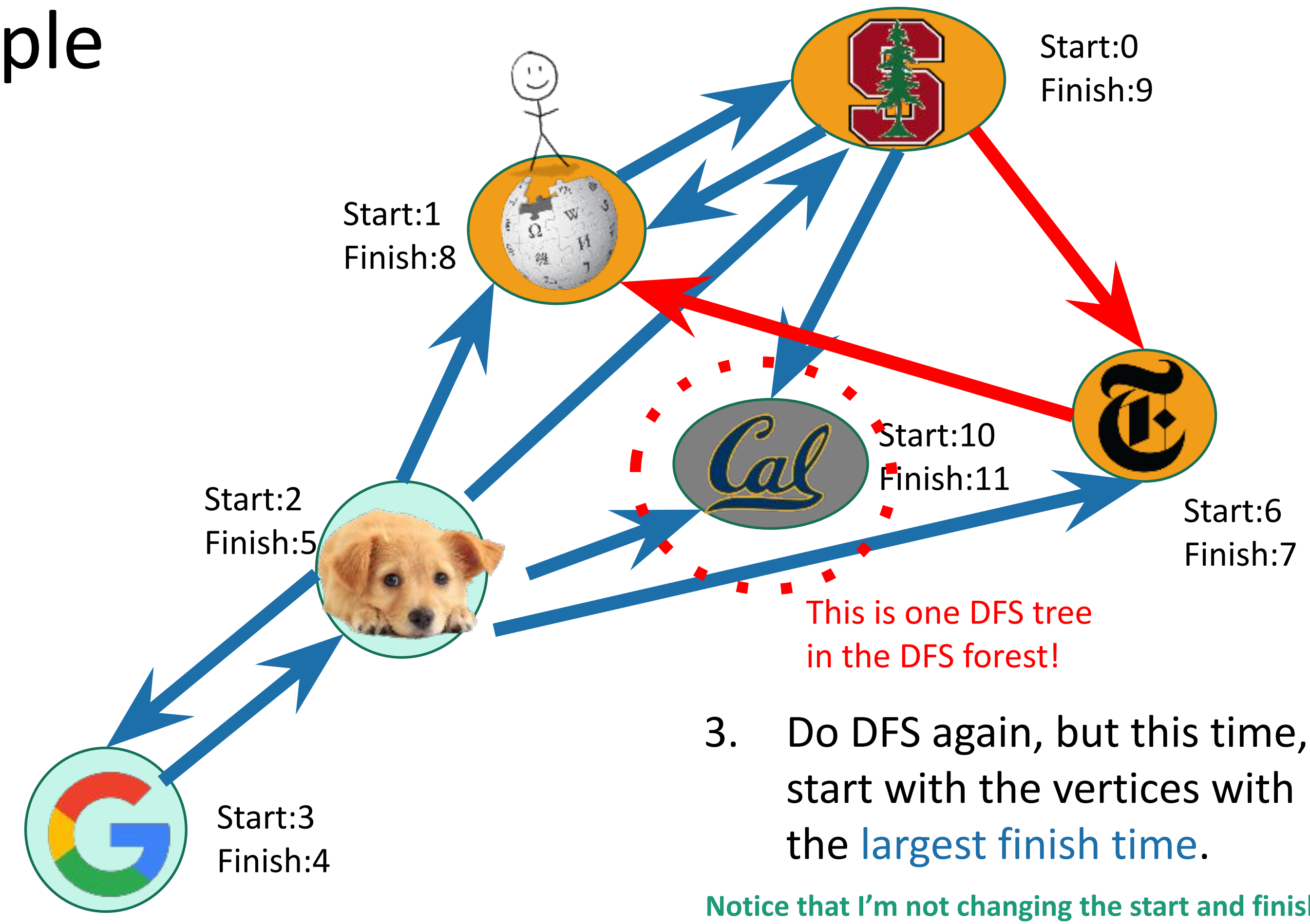
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



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

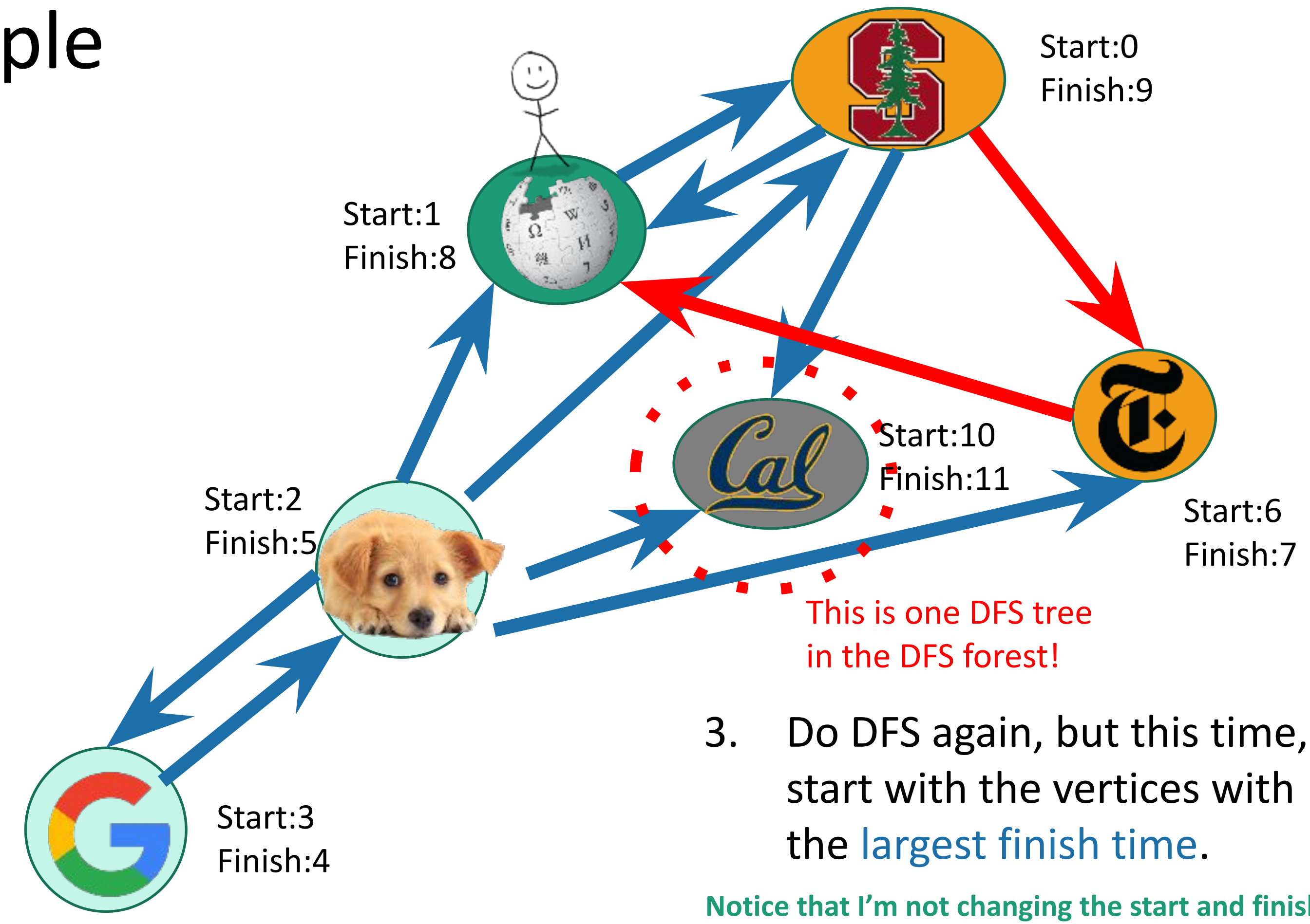
Example



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.

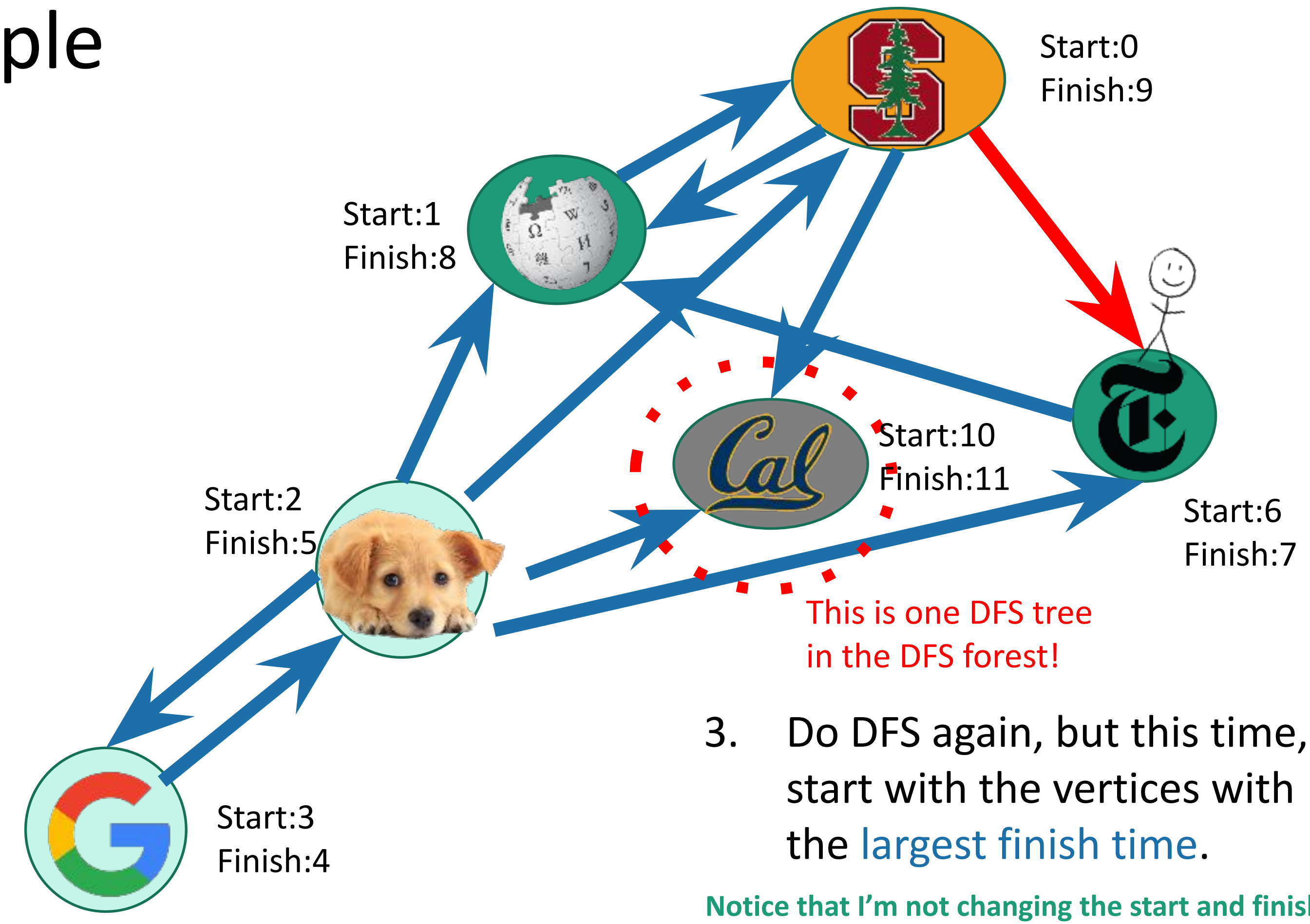
Example



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.

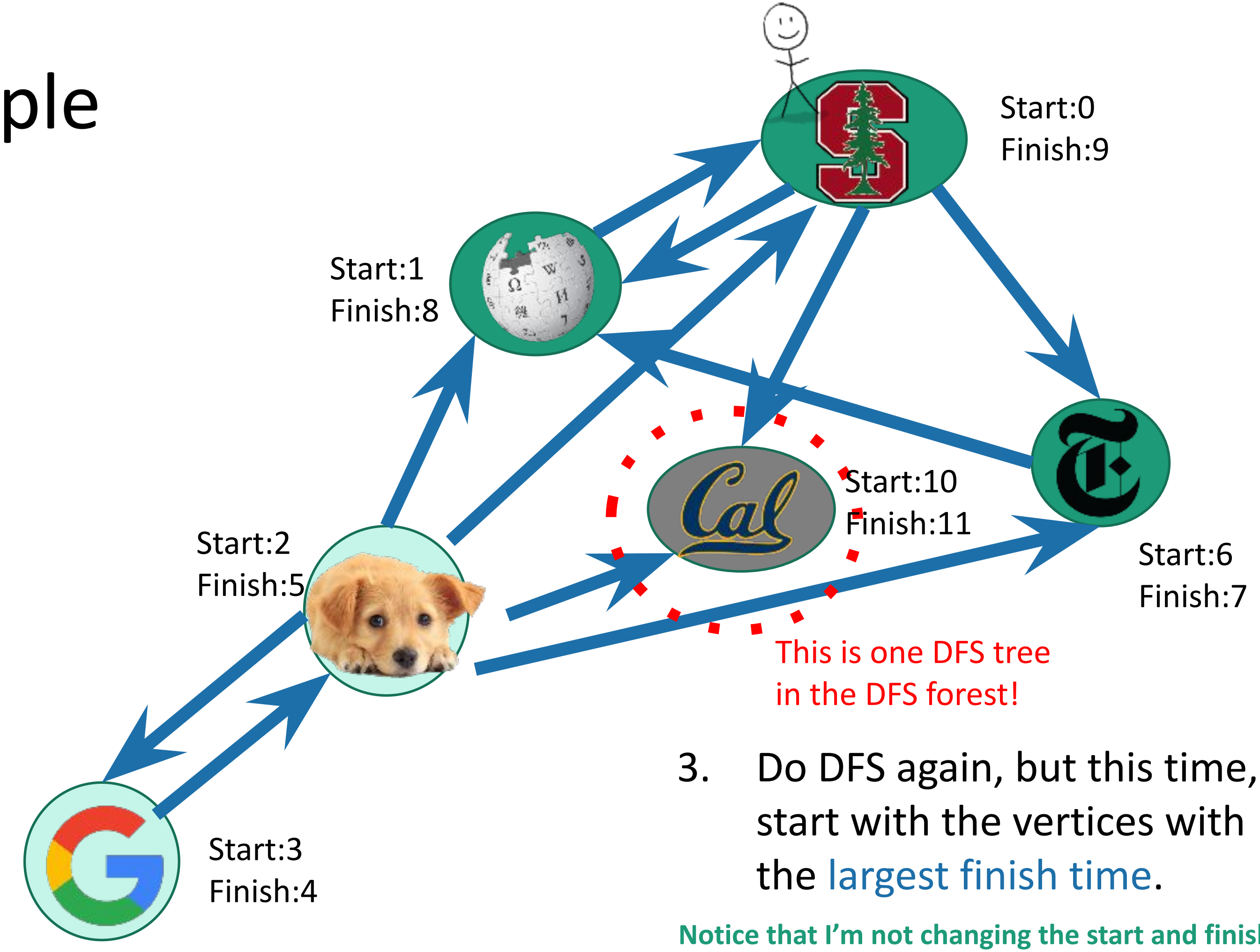
Example



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.

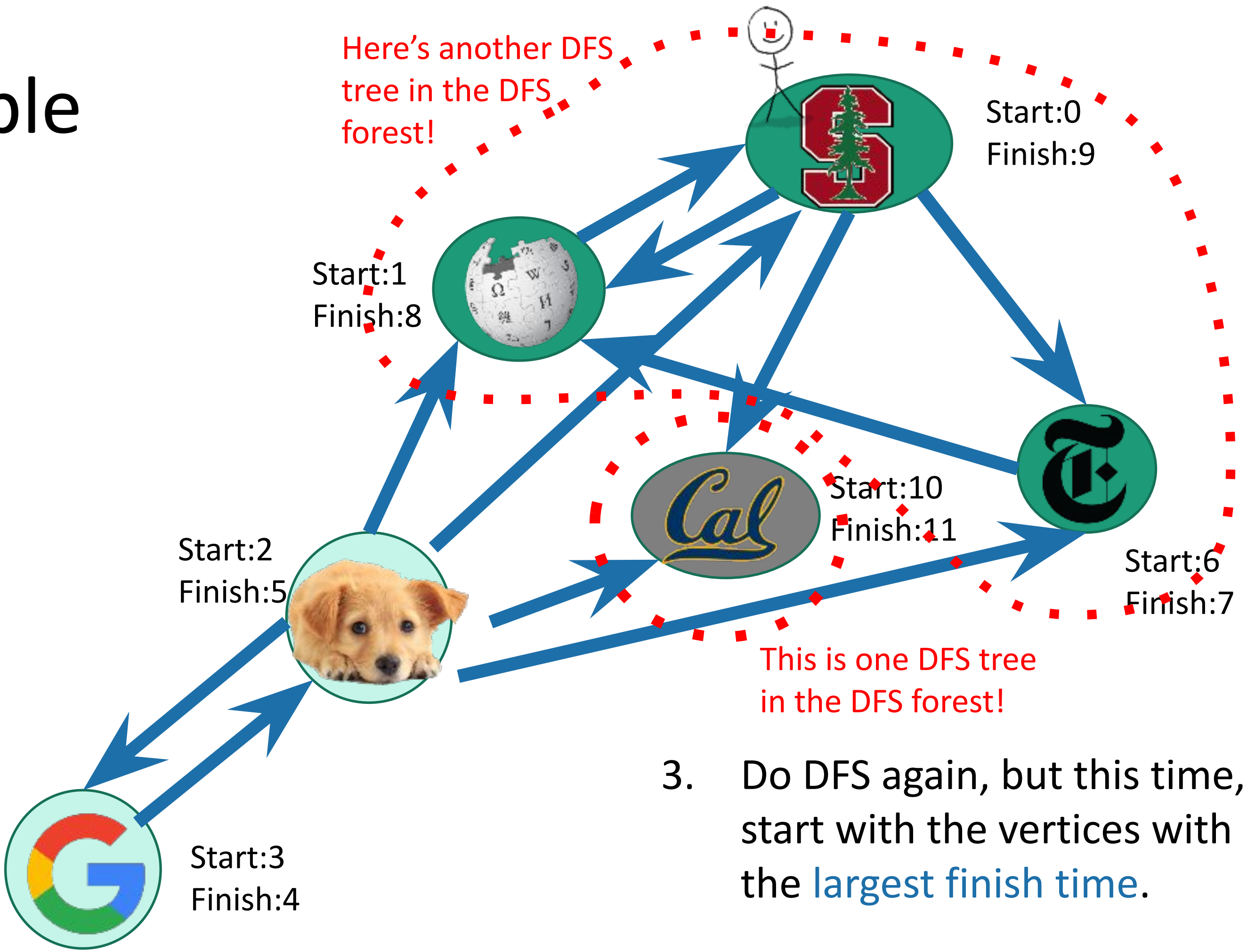
Example



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

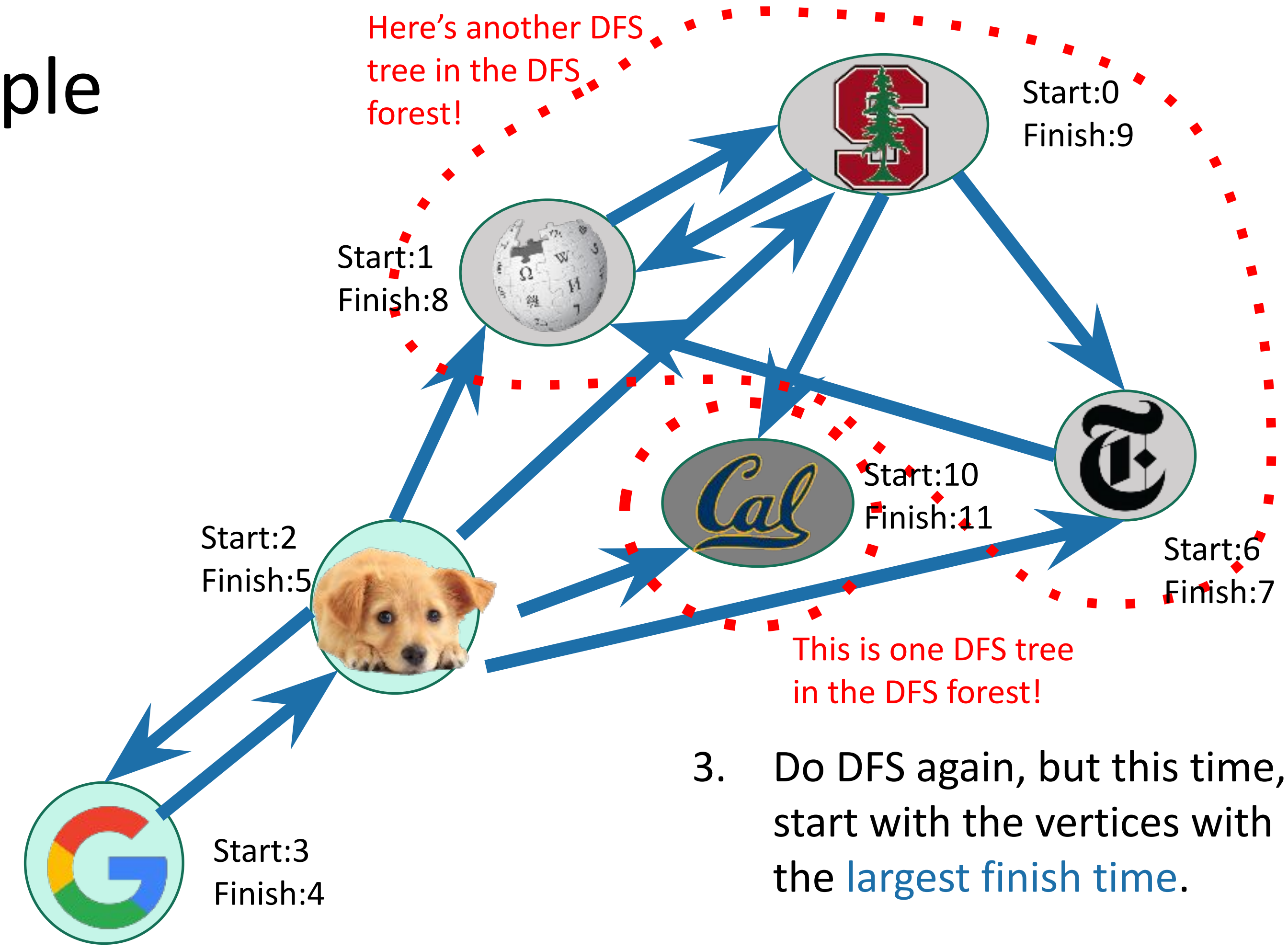
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

Example



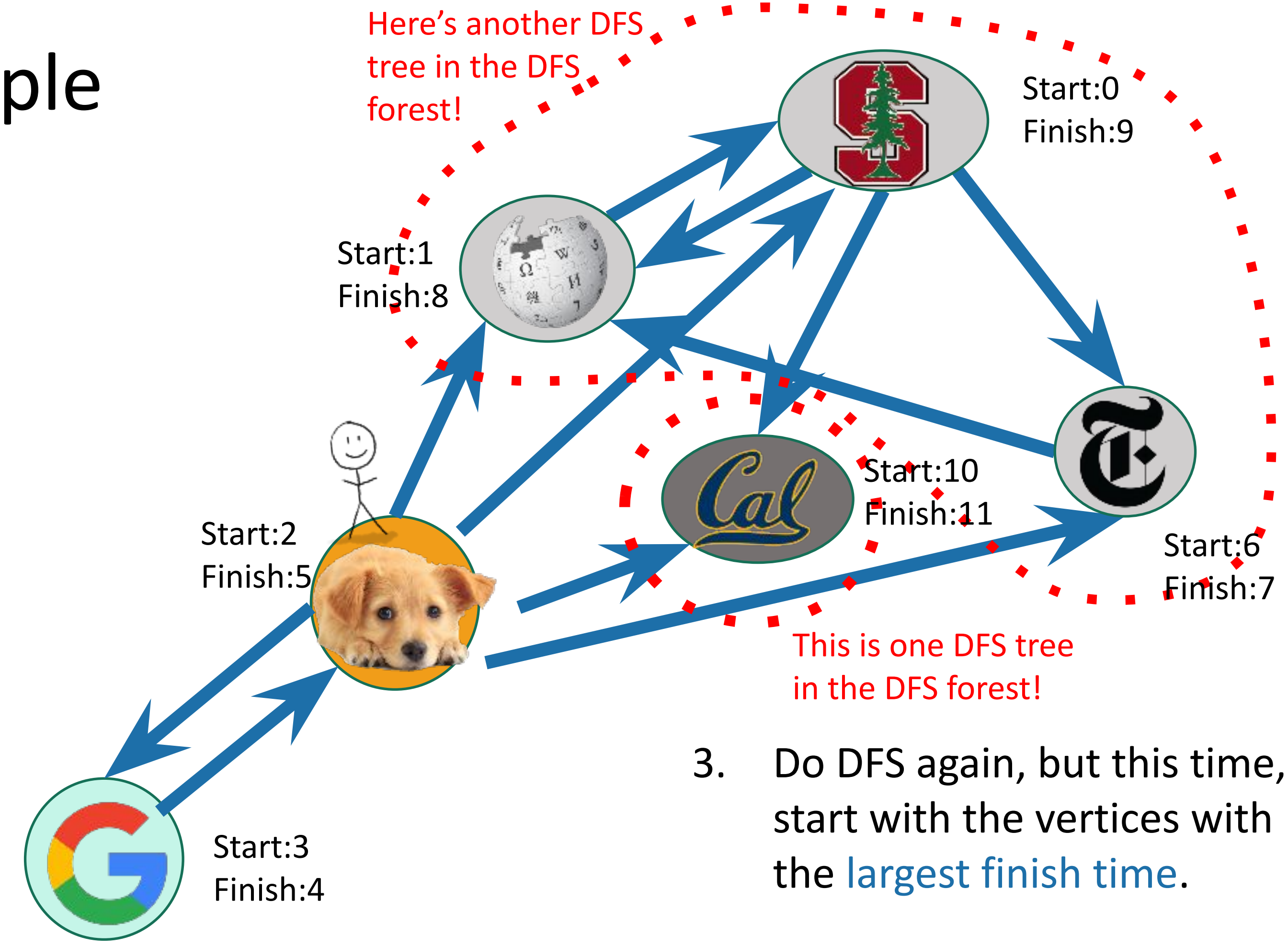
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



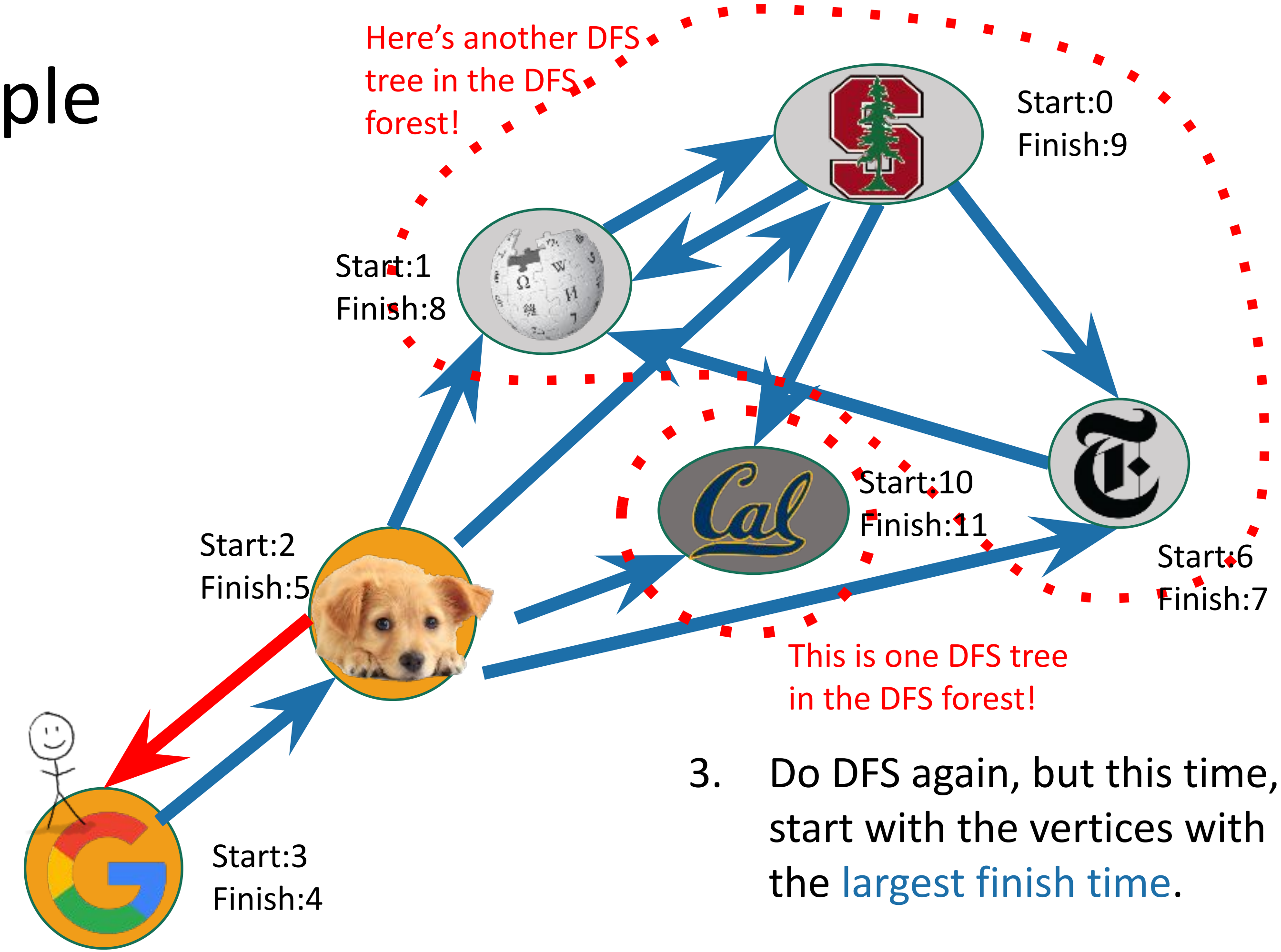
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



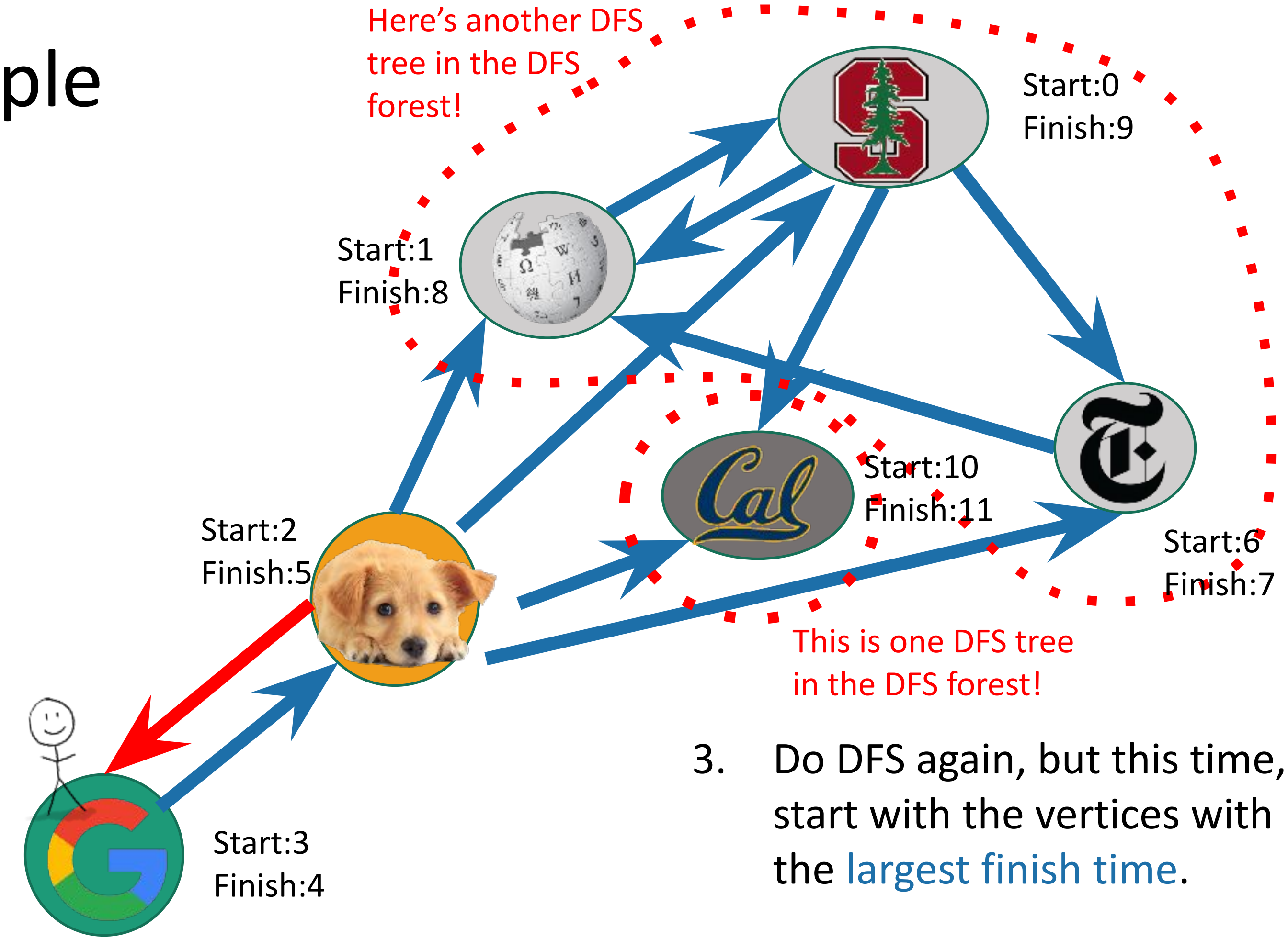
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



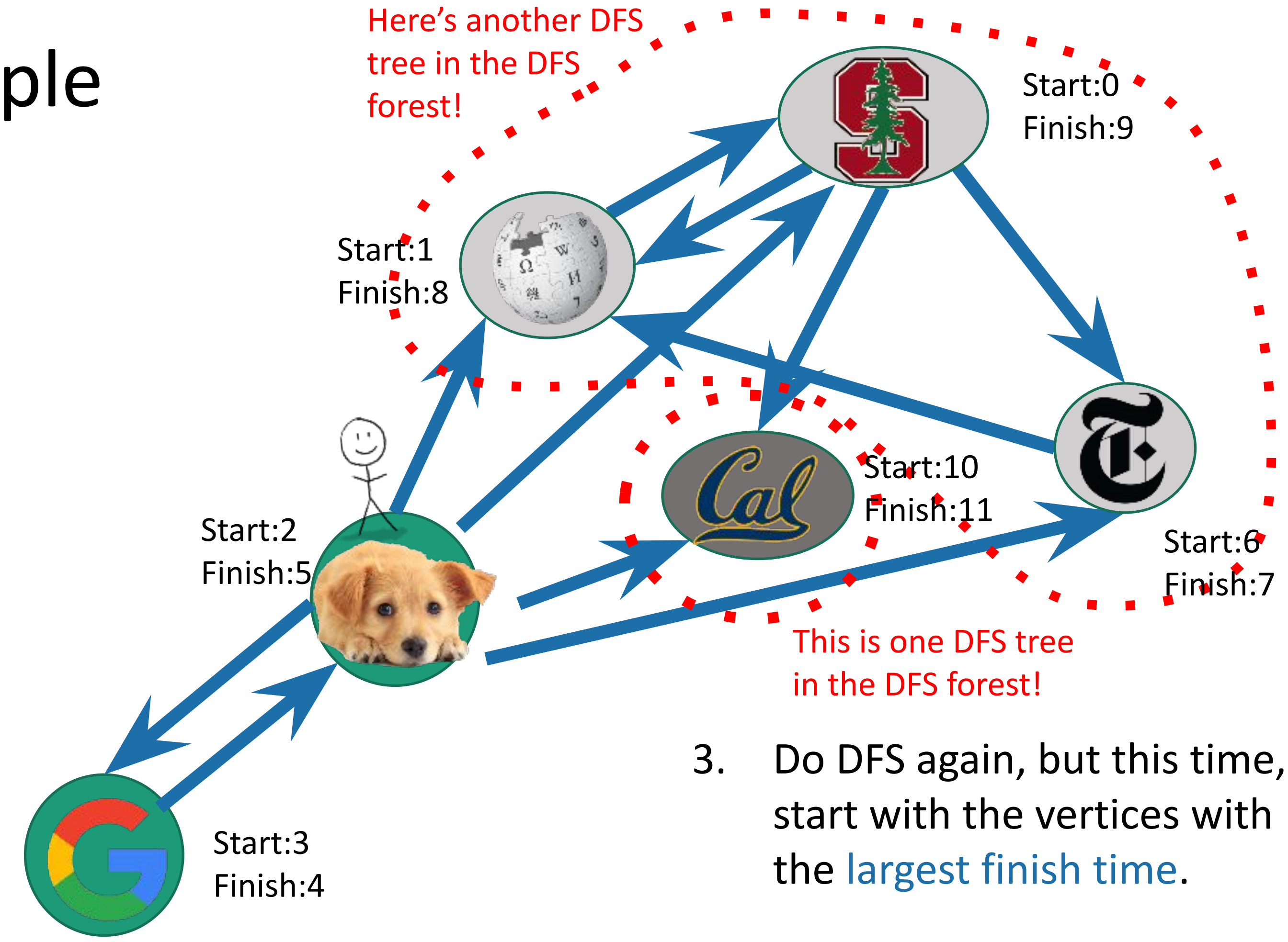
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



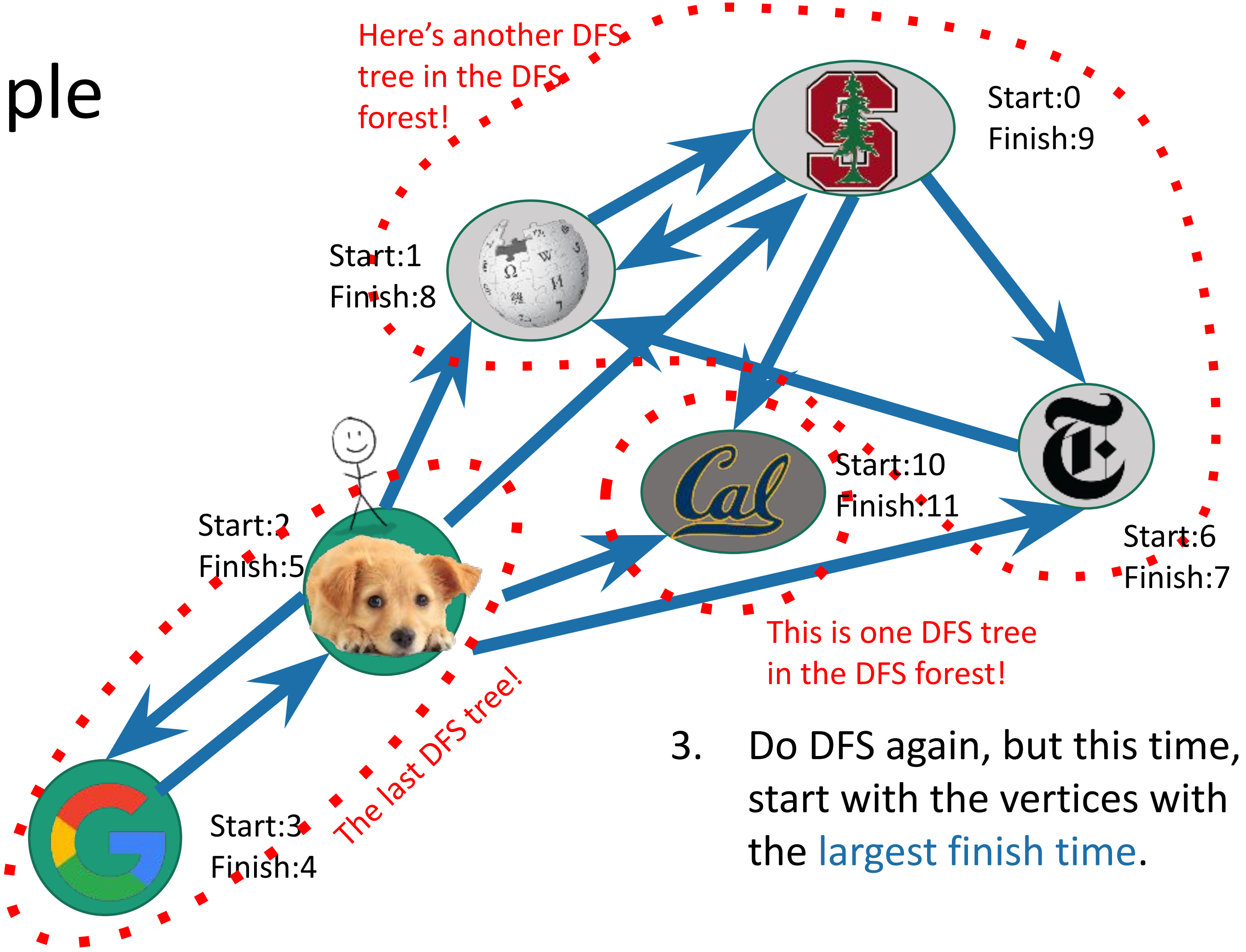
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



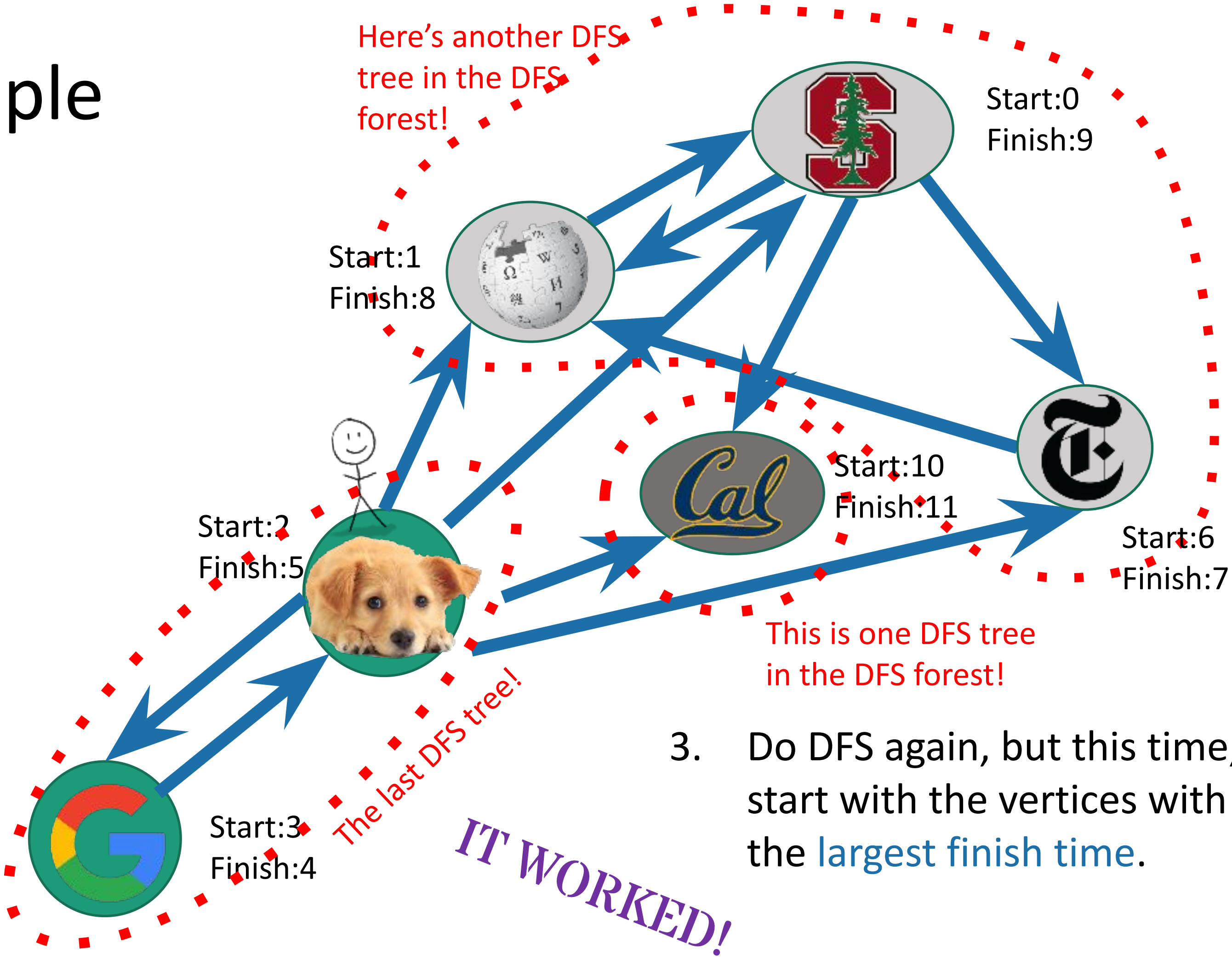
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example



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

Example

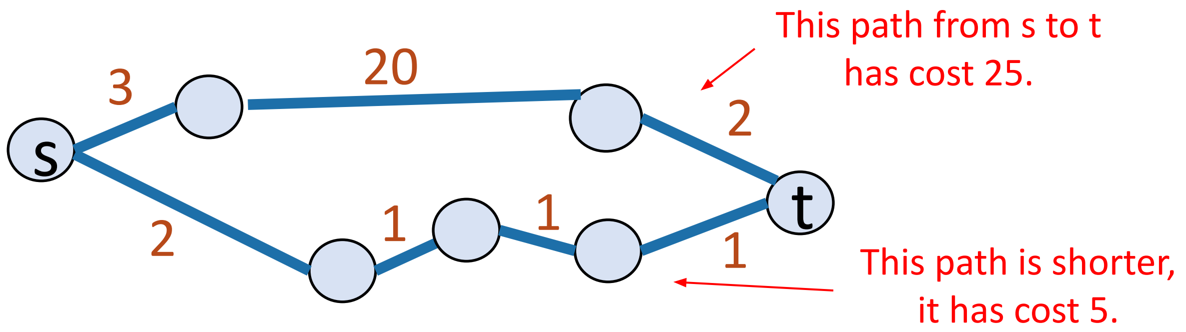


3. Do DFS again, but this time, start with the vertices with the largest finish time.

Shortest Path algorithms

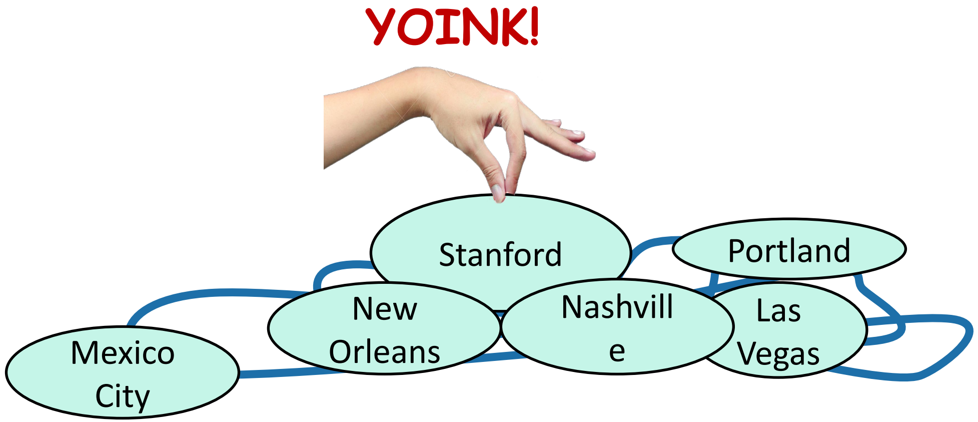
Shortest path problem

- What is the shortest path between u and v in a weighted graph?
  - the **cost** of a path is the sum of the weights along that path
  - The **shortest path** is the one with the minimum cost.

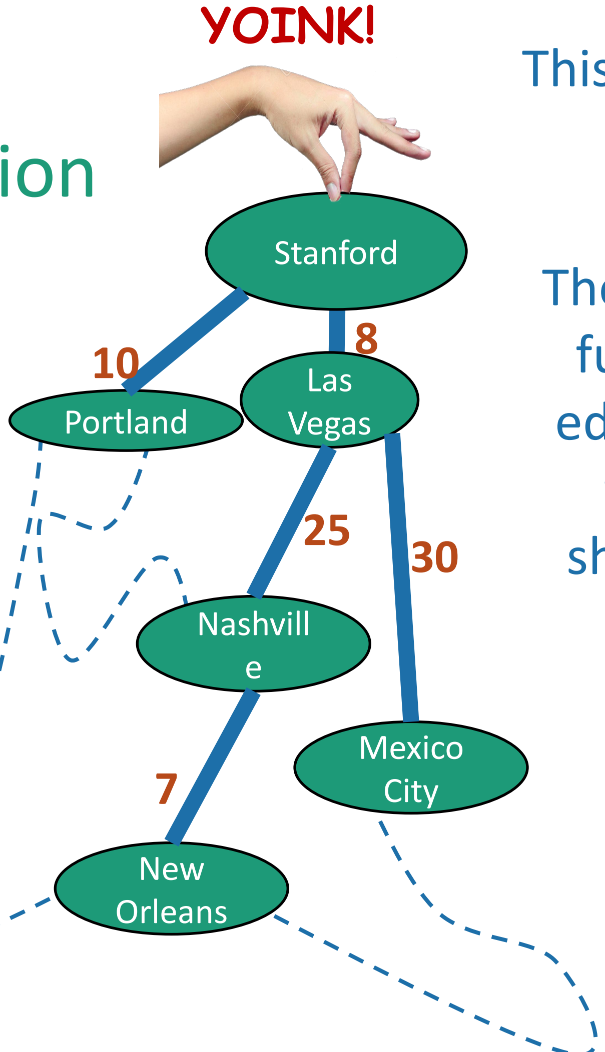

This path from s to t has cost 25.
3
20
2
s
t
2
1
1
1
This path is shorter, it has cost 5.

- The **distance** d(u,v) between two vertices u and v is the cost of the shortest path between u and v.

Dijkstra
visual intuition



Dijkstra
visual intuition



This creates a tree!

The highlighted + fully stretched edges along this tree are the shortest paths!

Dijkstra Pseudocode

Dijkstra(G,s):

```
• Set all vertices to not-sure
• d[v] = ∞ for all v in V (except s)
• d[s] = 0
• While there are not-sure nodes:
    • Pick the not-sure node u with the smallest estimate d[u].
    • Mark u as sure.
    • For v in u.neighbors:
        • d[v] ← min( d[v] , d[u] + edgeWeight(u,v))
• Now d(s, v) = d[v]
```

Shouldn't be used on graphs with negative edges!

Runtime? Will need to use Fibonacci heaps
to make operations highlighted operations run faster!

Need to use Fibonacci Heap to implement fastest Dijkstra

* - amortized time

	Sorted Arrays	Linked Lists	Binary Heap	Fibonacci Heap	Red-Black Trees
Search	O(log(n))	O(n)	O(n)	O(n)	O(log(n))
Delete	O(n)	Search +O(1)	Search +O(log(n))	Search +O(log(n))*	O(log(n))
Insert	O(n)	O(1)	O(log(n))	O(1)	O(log(n))
Extract-min	O(1)	O(n)	O(log(n))	O(log(n))*	O(log(n))
Decrease-key	O(n)	Search +O(1)	Search +O(log(n)	O(1)*	O(log(n)

Dijkstra with **Fibonacci Heap**

- T(extractMin) = O(log(n)) (amortized time*)
- T(decreaseKey) = O(1) (amortized time*)
- See CS166 for more! (or CLRS)

- Running time of Dijkstra

  = O(n(T(`extractMin`)) + m T(`decreaseKey`))

  = O(nlog(n) + m)

*This means that any sequence of d `extractMin` calls takes time at most O(dlog(n)). But a few of the d may take longer than O(log(n)) and some may take less time..

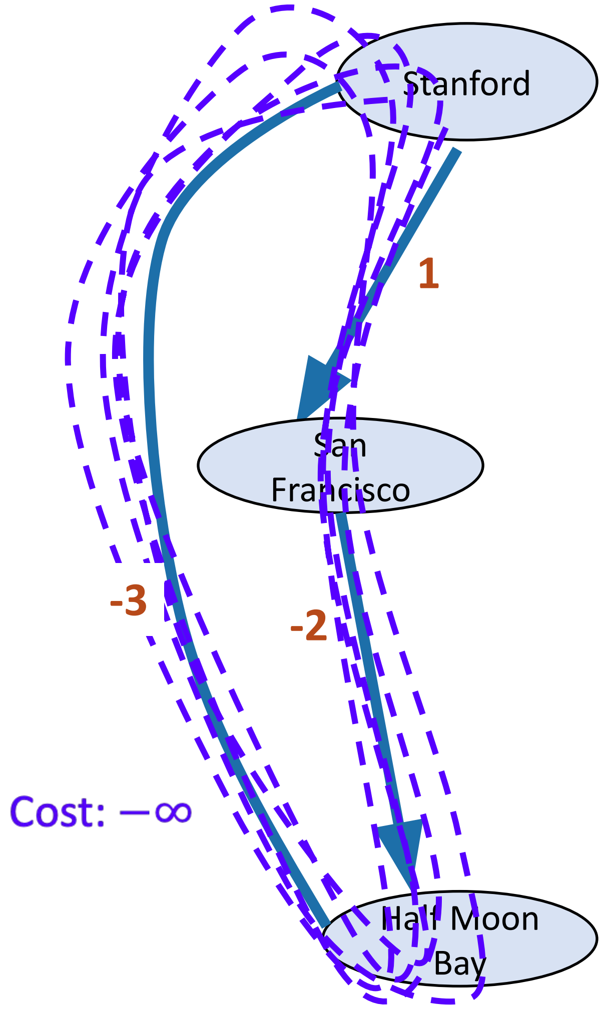
What about a Dynamic Programming approach to shortest path problems?

Assumption: no negative cycles

- Negative weights are possible (some algorithms won't work in that case!)
  - Example: negative costs because I pick up some passengers

- But if we have negative cycles…

- Shortest paths aren't defined if there are negative cycles!



Shortest path using DP

Substructure = path with small number of edges

- **Step 1:**

  Optimal substructure: shortest path using $\leq i$ edges

- **Step 2:**

Suppose we already know $d^i(s,u)$ for fixed s and all u

Recursive formulation: $d^{i+1}(s,v) = \min_u \{d^i(s,u) + w(u,v)\}$

last vertex before v

Optimal path from s to u + from u to v

Step 3: write the algorithm

The DP algorithm is exactly Bellman-Ford!

Bellman-Ford(G,s):

```
• $d^{(0)}[v] = \infty$ for all v in V
• $d^{(0)}[s] = 0$                         // initialize:
                                             $d^{(i)}[v]$ is distance from s to v
                                                    with $\leq$i edges

• For i=0,…,n-2:
    • $d^{(i+1)}[v] = d^{(i)}[v]$ for all v in V    // baseline distance:
                                                      v doesn't need (i+1)th edge
    • For v in V:
        • For u in v.neighbors:
            • $d^{(i+1)}[v] \leftarrow \min(d^{(i+1)}[v], d^{(i)}[u] + edgeWeight(u,v))$
                                                    // found a better path through u

• Return $d^{(n-1)}$
```

n = # of vertices
m = # of edges

Bellman-Ford take-aways

- Running time is O(mn)
  - For each of n rounds, update m edges.

```
• For i=0,…,n-1:
    • For u in V:
        • For v in u.neighbors:
```

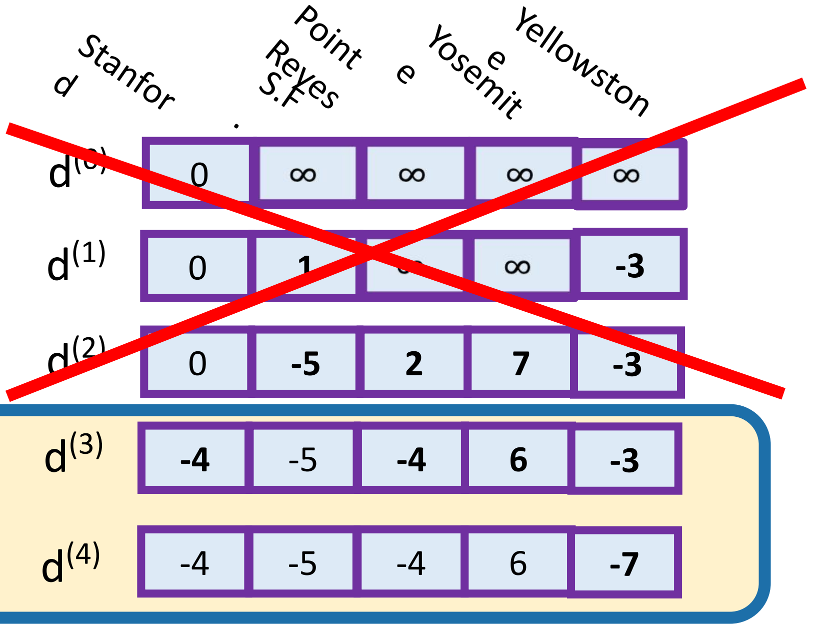
m = # of edges

$$= \frac{1}{2} \sum_{v \in V} \text{degree}(v)$$

- Works fine with negative edges.
- Does not work with negative cycles.
  - But it can detect negative cycles!

Note on implementation

- Don't actually keep all n arrays around.
- Just keep two at a time: "last round" and "this round"



	Stanford	S.F.	Point Reyes	Yosemite	Yellowstone
~~$d^{(0)}$~~	~~0~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~
~~$d^{(1)}$~~	~~0~~	~~**1**~~	~~∞~~	~~∞~~	~~**-3**~~
~~$d^{(2)}$~~	~~0~~	~~**-5**~~	~~**2**~~	~~**7**~~	~~**-3**~~
$d^{(3)}$	**-4**	-5	**-4**	**6**	**-3**
$d^{(4)}$	-4	-5	-4	6	**-7**

Only need these two in order to compute $d^{(4)}$

Thank you!