# Graph Algorithms I

Summer 2018 • Lecture 07/19

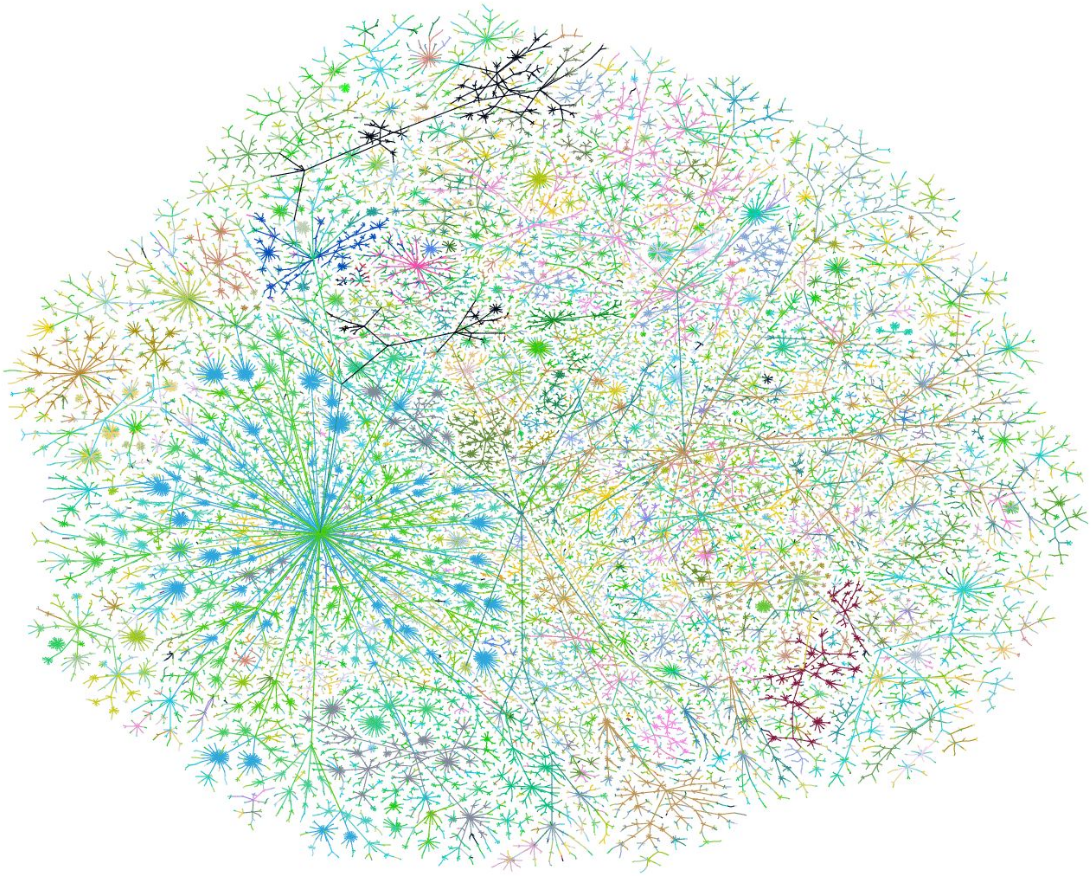
# Outline for Today

- Graph algorithms
  - Graph Basics
    - DFS: topological sort, in-order traversal of BSTs, exact traversals
    - BFS: shortest paths, bipartite graph detection
  - Kosaraju's Algorithm for SCC's

# Graph Basics

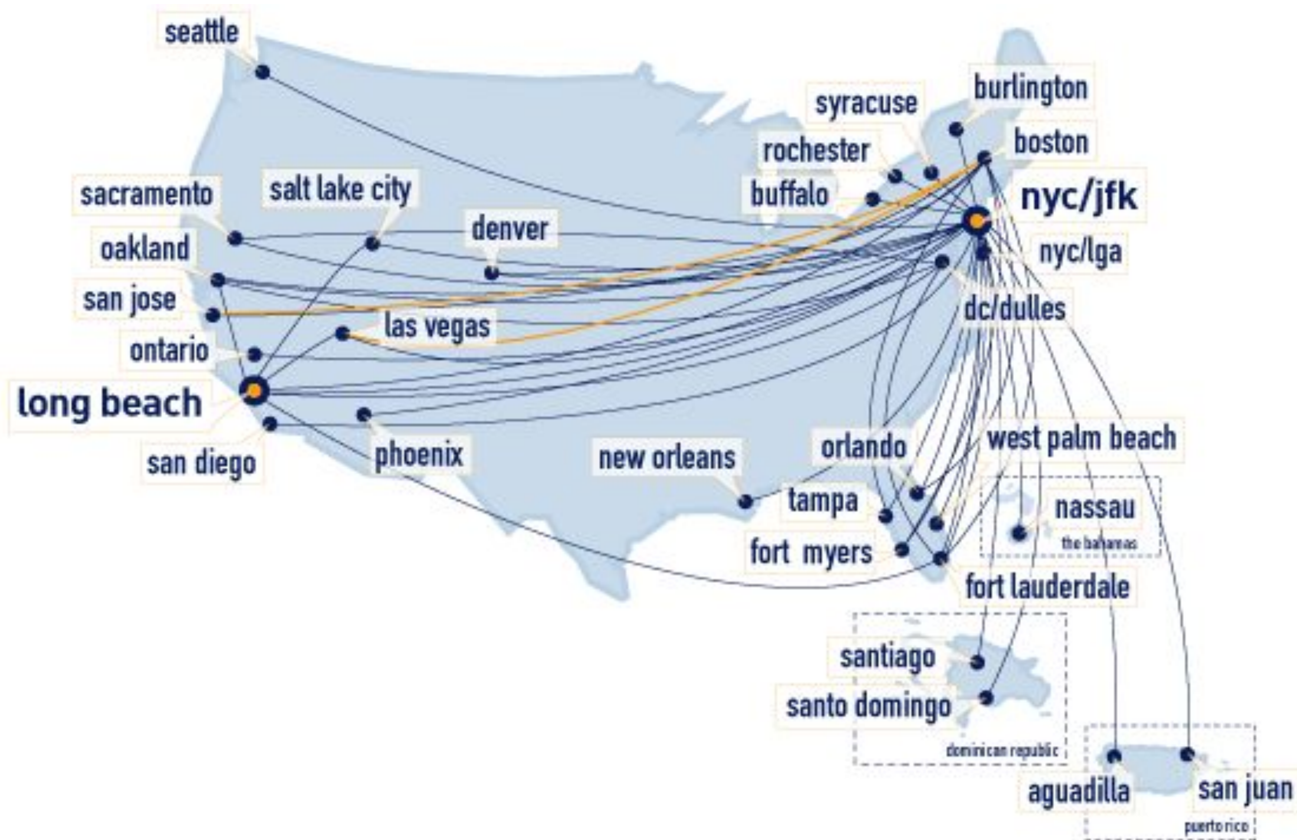
# Examples of Graphs

The Internet (circa 1999)



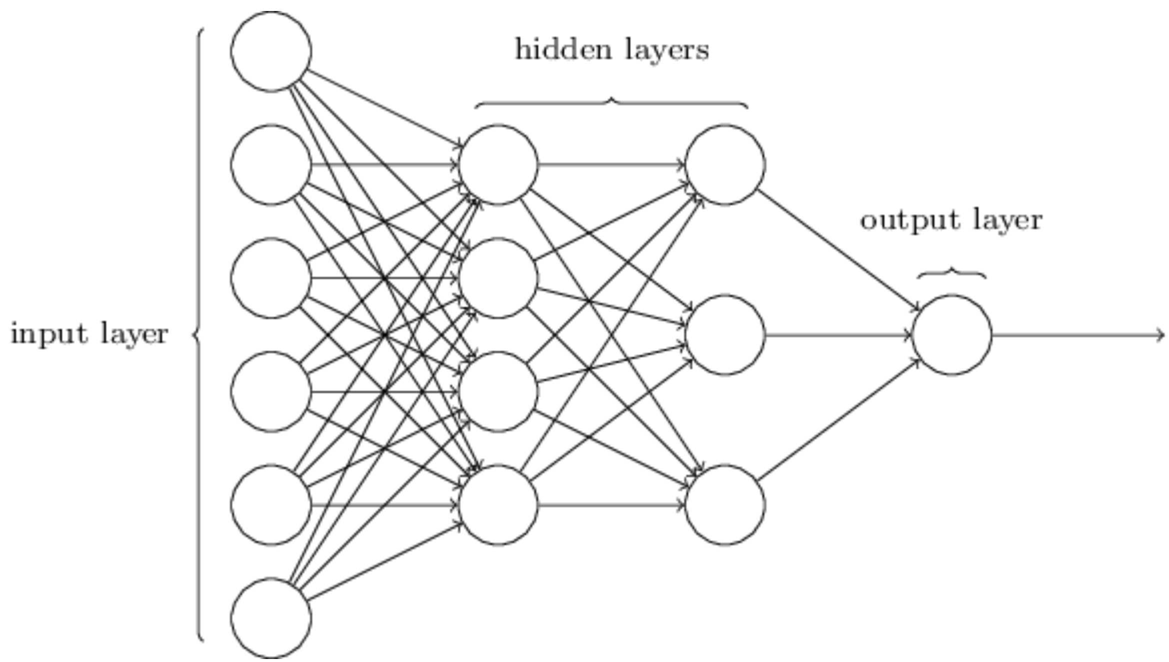
# Examples of Graphs

Flight networks (Jet Blue, for example)



# Examples of Graphs

Neural networks



# Graphs

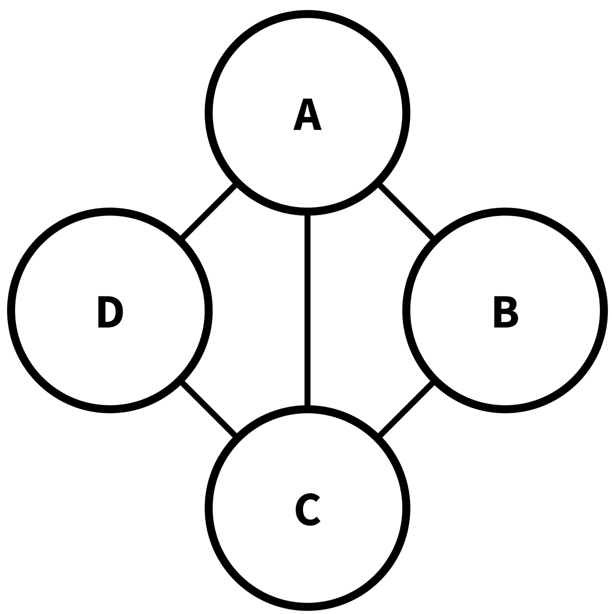
We might want to answer one of several questions about G.

- Finding the shortest path between two vertices (SPSP) for efficient routing.
- Finding strongly connected components for community detection or clustering.
- Finding the topological ordering to respect dependencies.

# Undirected Graphs

An undirected graph has vertices and edges.

- V is the set of vertices and E is the set of edges.
- Formally, an undirected graph is G = (V, E).
- e.g. V = {A, B, C, D} and E = { {A, B}, {A, C}, {A, D}, {B, C}, {C, D} }



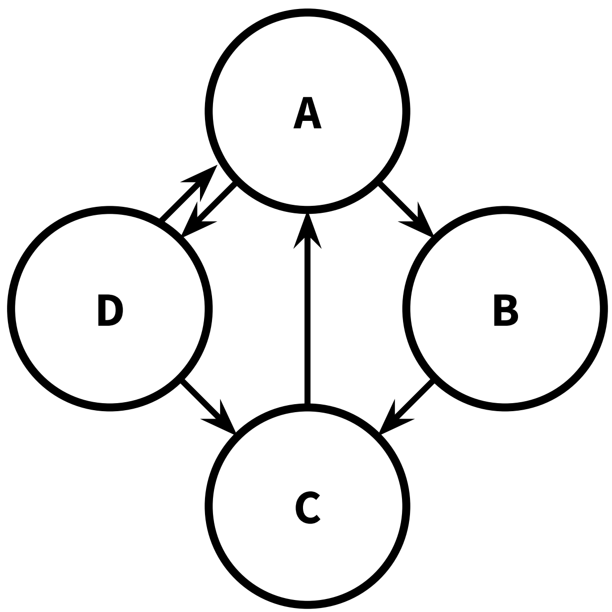
# Directed Graphs

A directed graph has vertices and **directed** edges.

V is the set of vertices and E is the set of directed edges.

Formally, a directed graph is G = (V, E)

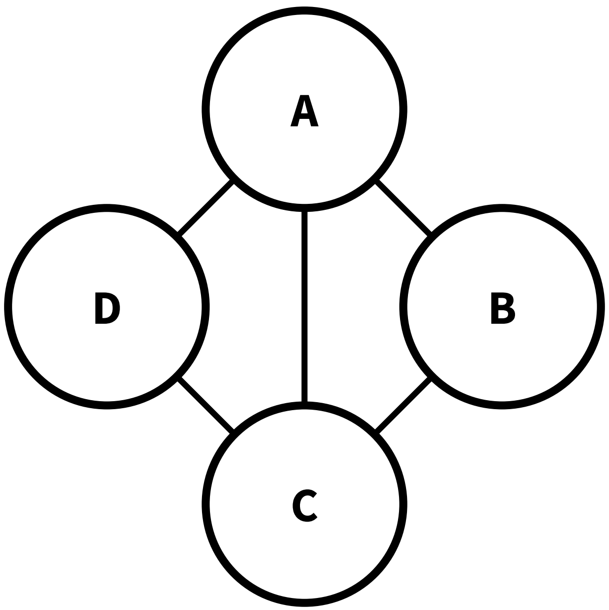
e.g. V = {A, B, C, D} and E = { [A, B], [A, D], [B, C], [C, A], [D, A], [D, C] }



# Graph Representations

How do we represent graphs?

**(1) Adjacency matrix**

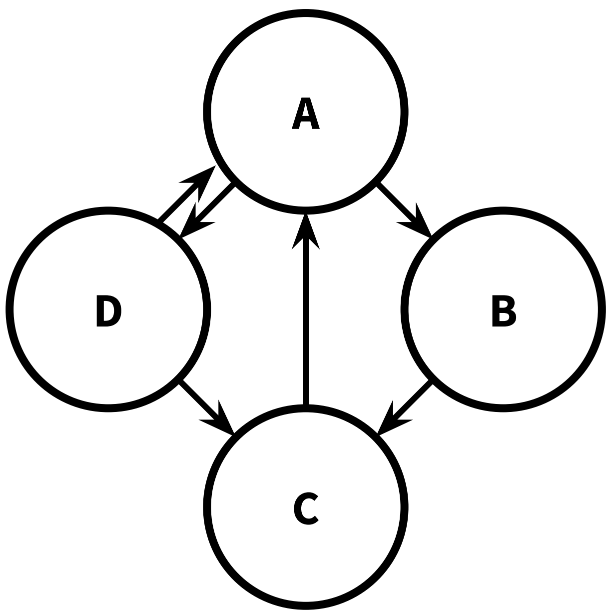



$$
\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} &
\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}
\end{array}
$$

# Graph Representations

How do we represent graphs?

**(1) Adjacency matrix**




destination

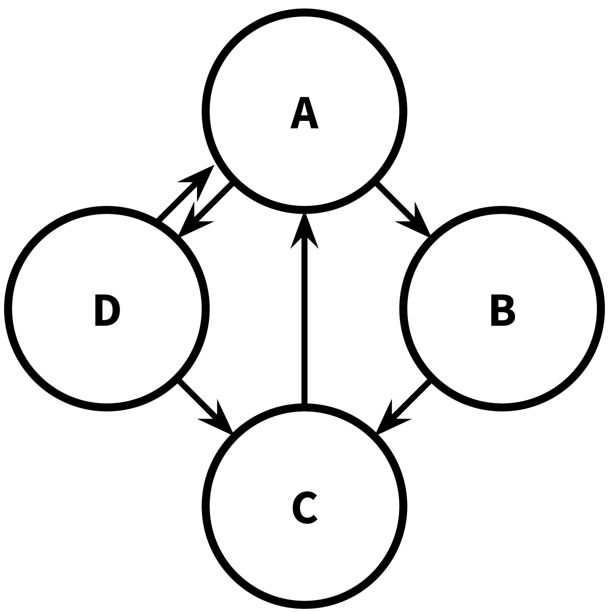
$$\text{source}\quad \begin{array}{cc} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

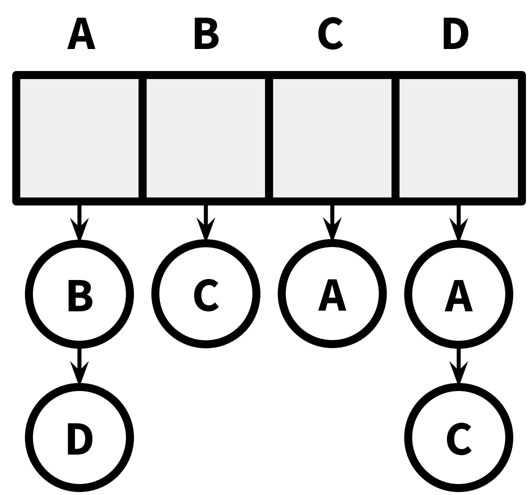
# Graph Representations

How do we represent graphs?

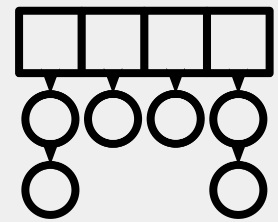
(1) Adjacency matrix

**(2) Adjacency list**



# Graph Representations

| For G = (V, E) | $\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$ |  |
|---|---|---|
| **Edge Membership**<br>Is e = {u, v} in E? | O(1) | O(deg(u))<br>**or**<br>O(deg(v)) |
| **Neighbor Query**<br>What are the neighbors of u? | O( \|V\| ) | O(deg(v)) |
| **Space requirements** | O( \|V\|$^2$) | O(\|V\|+\|E\|) |

Generally, better for sparse graphs.

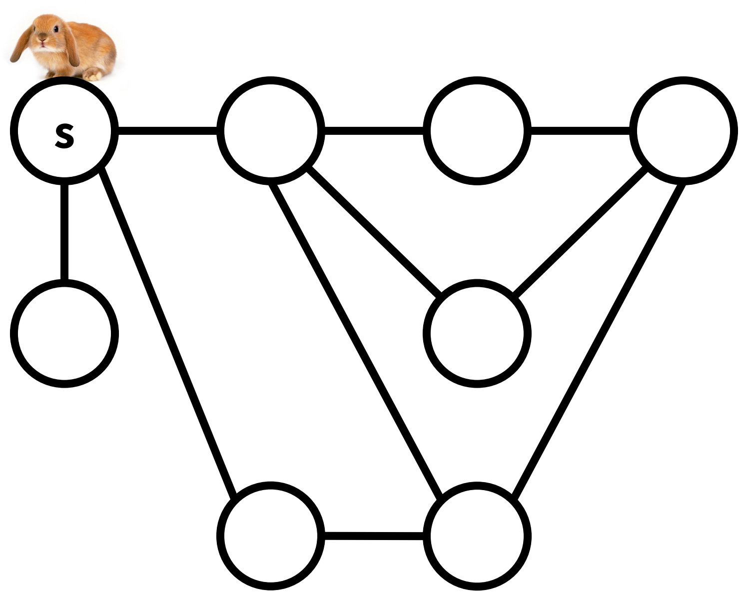
We'll assume this representation, unless otherwise stated.

# Depth-First Search

# Depth-First Search

## An analogy

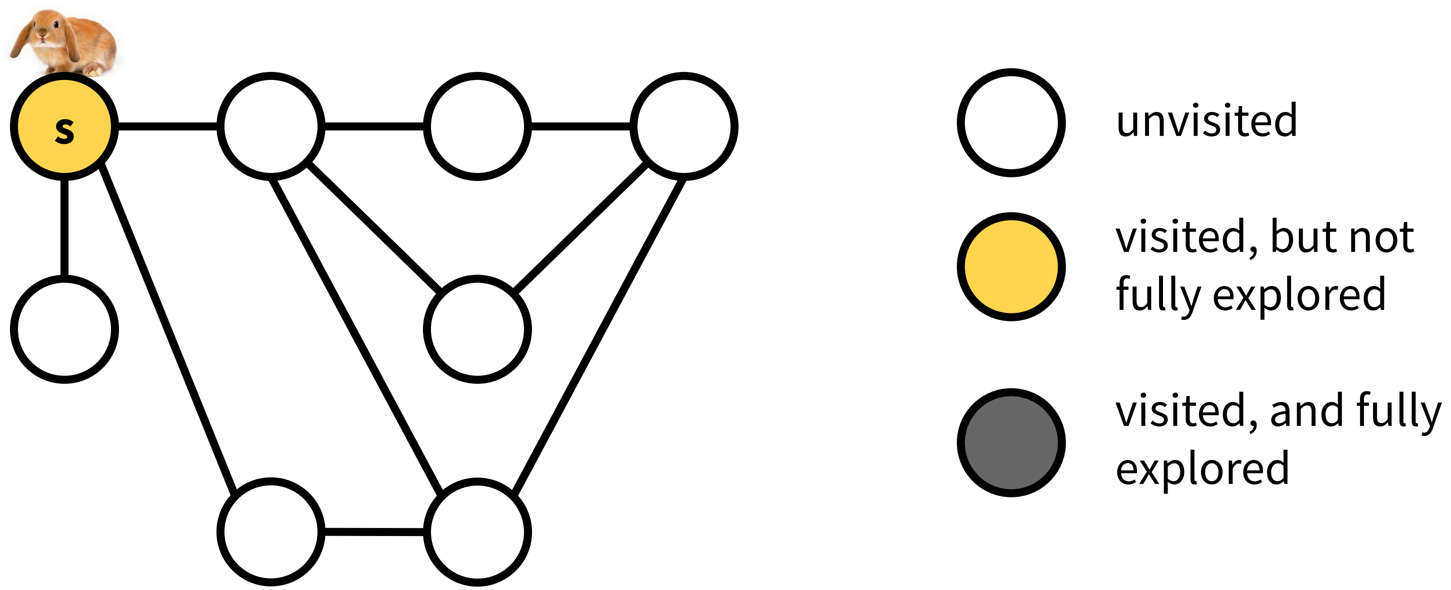
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).




# Depth-First Search

## An analogy

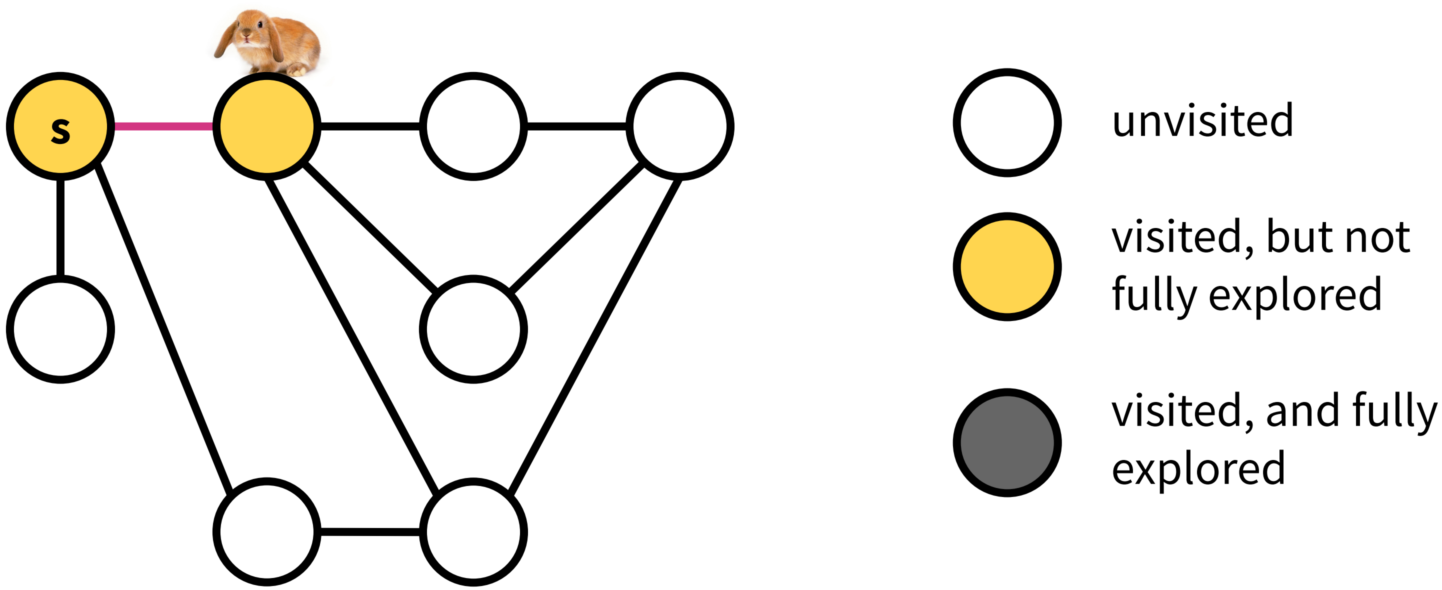
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

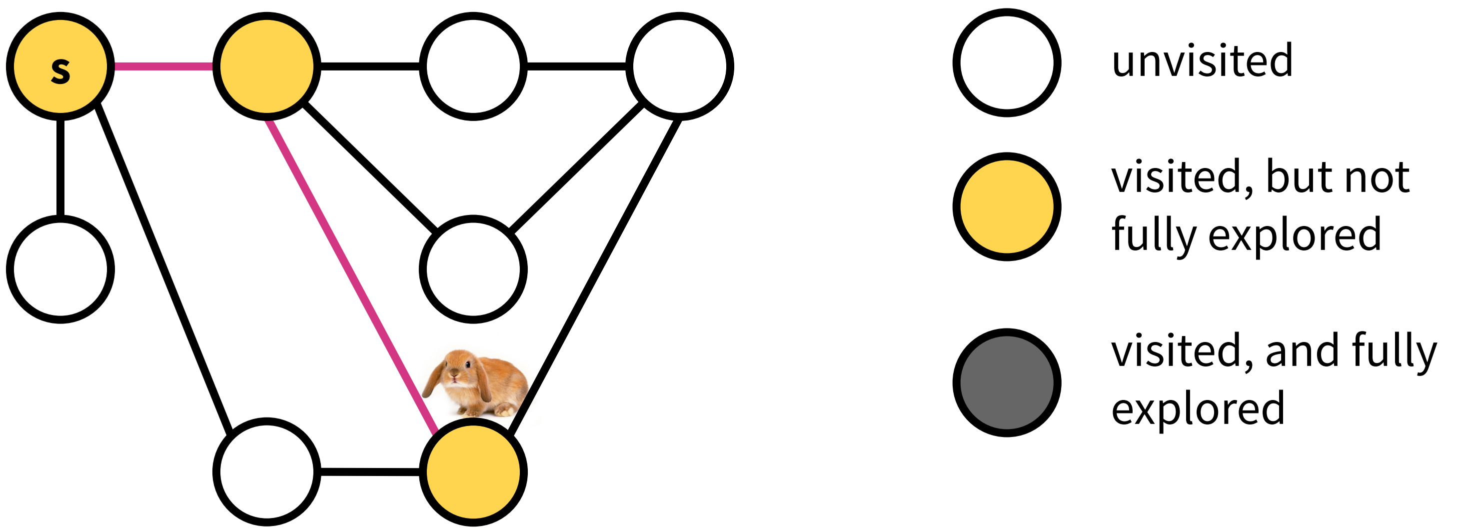
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

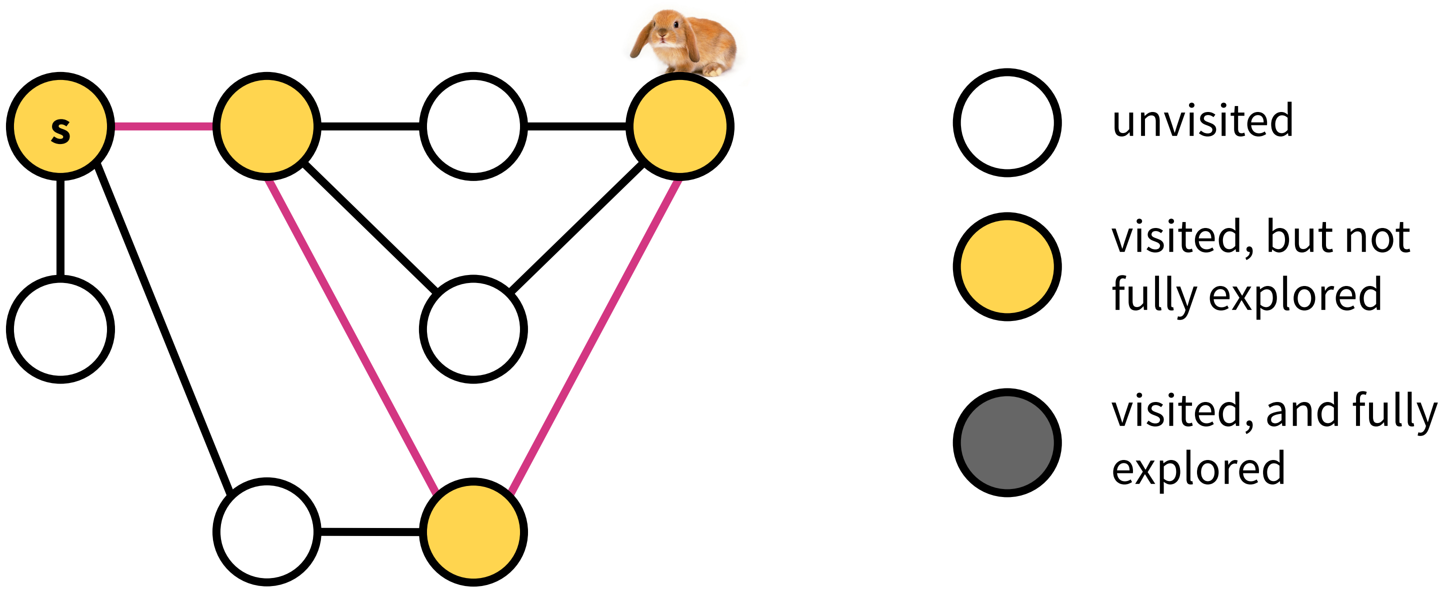
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

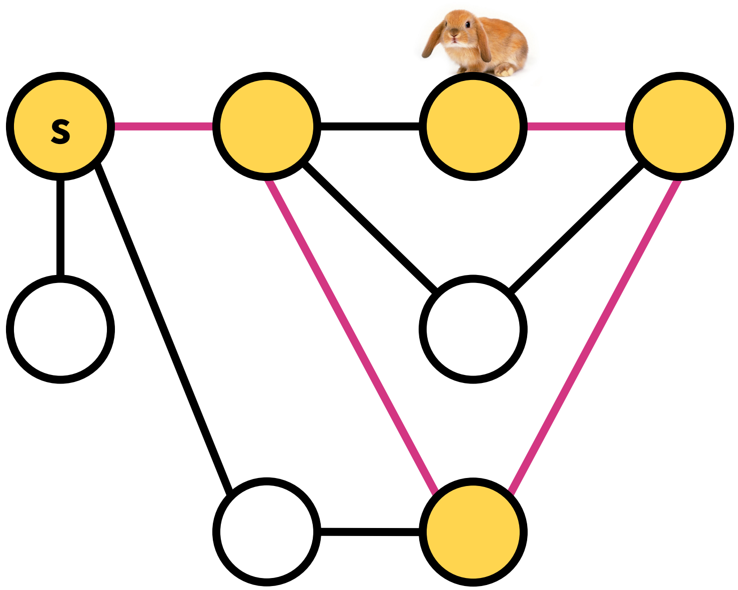
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

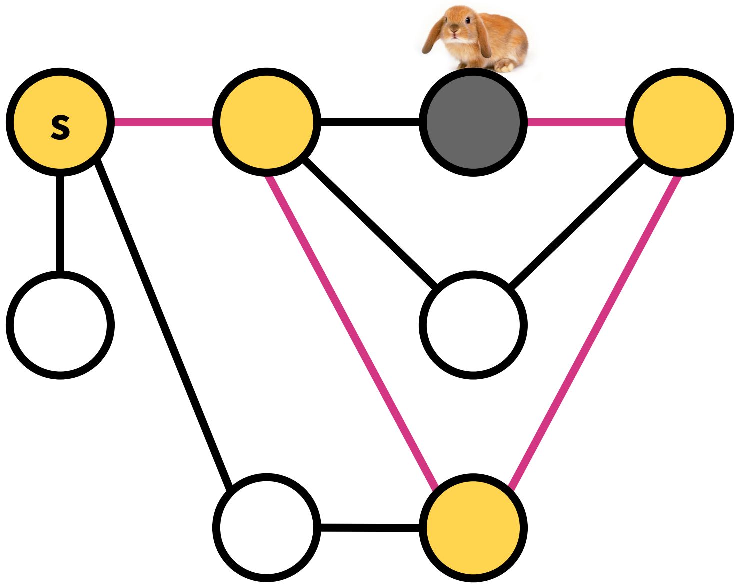
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).




# Depth-First Search

## An analogy

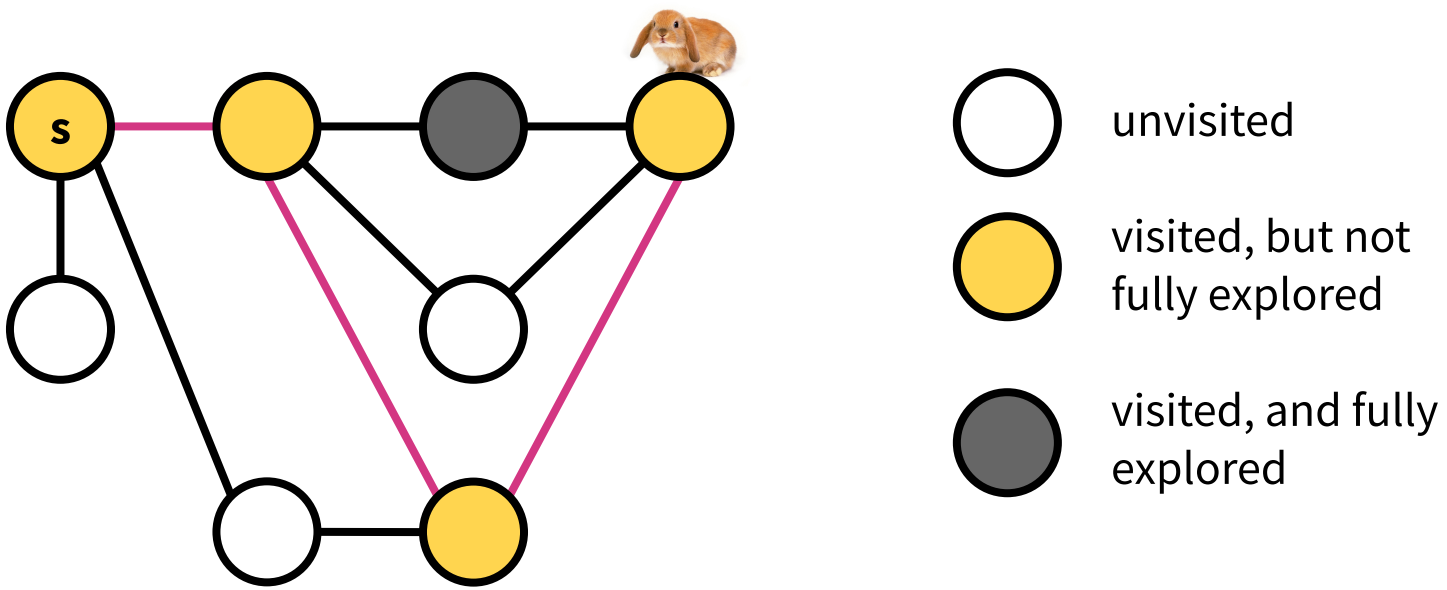
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).




# Depth-First Search

## An analogy

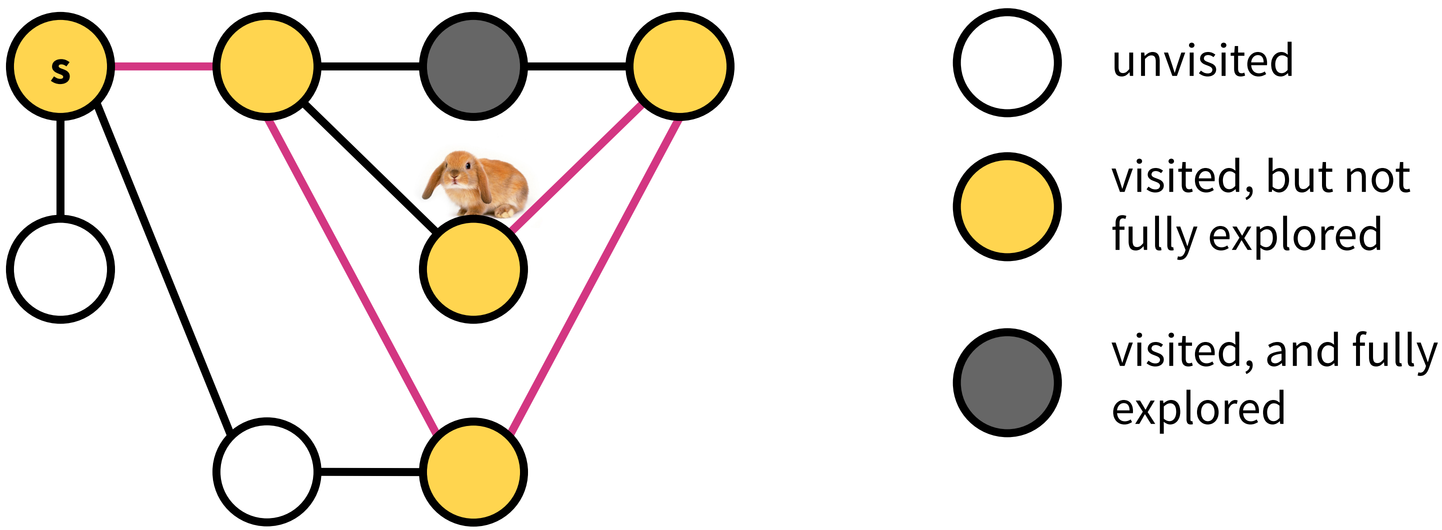
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

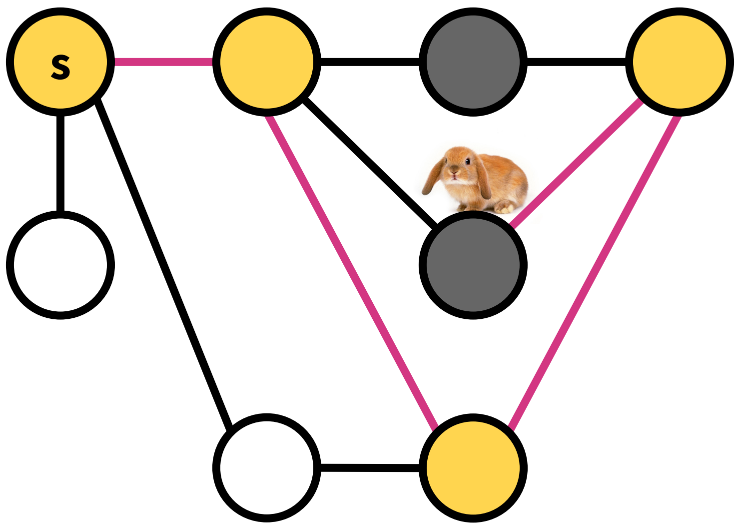
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

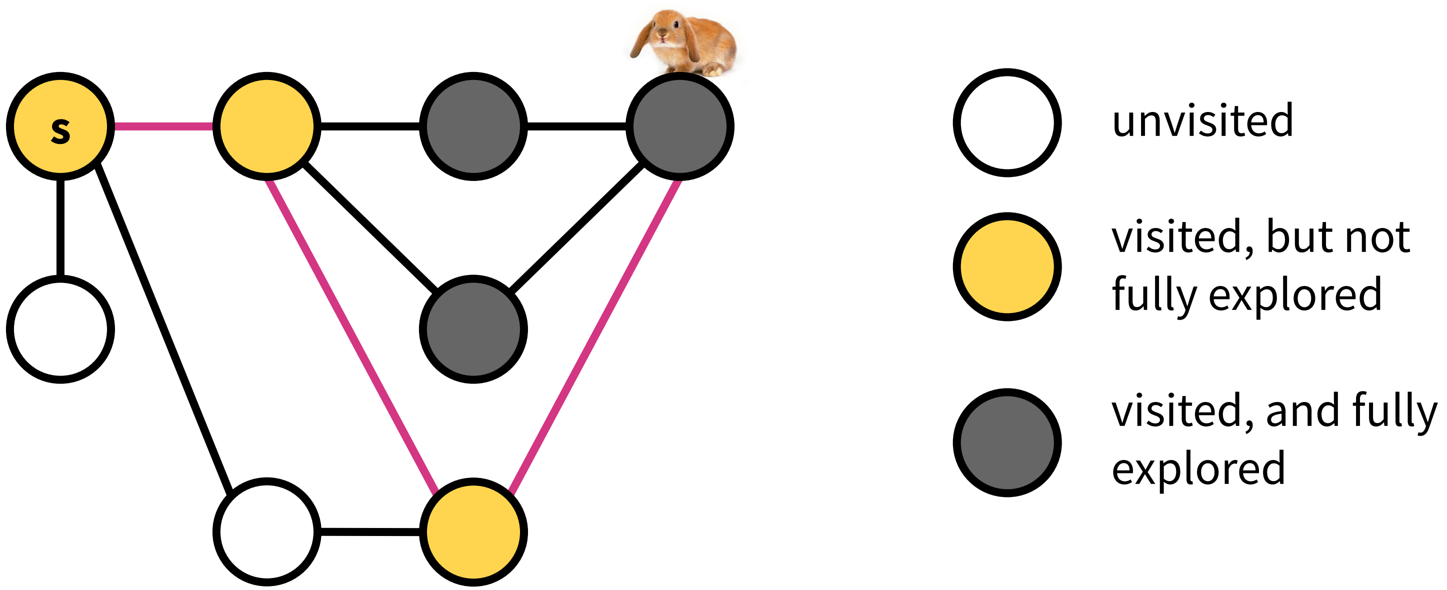
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).




# Depth-First Search

## An analogy

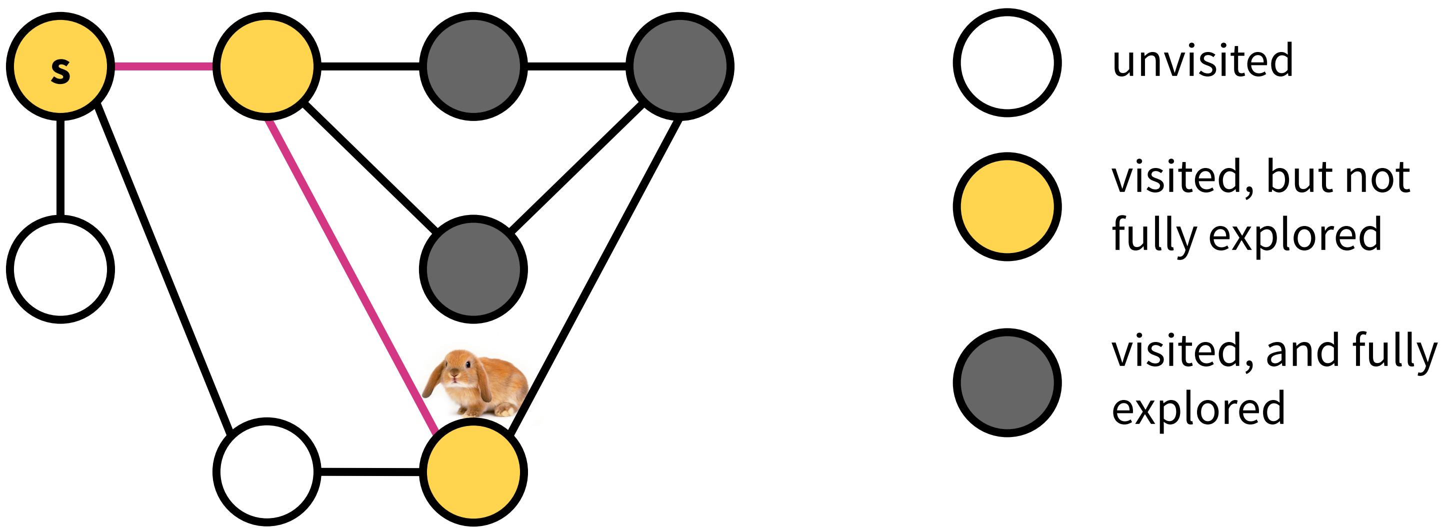
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

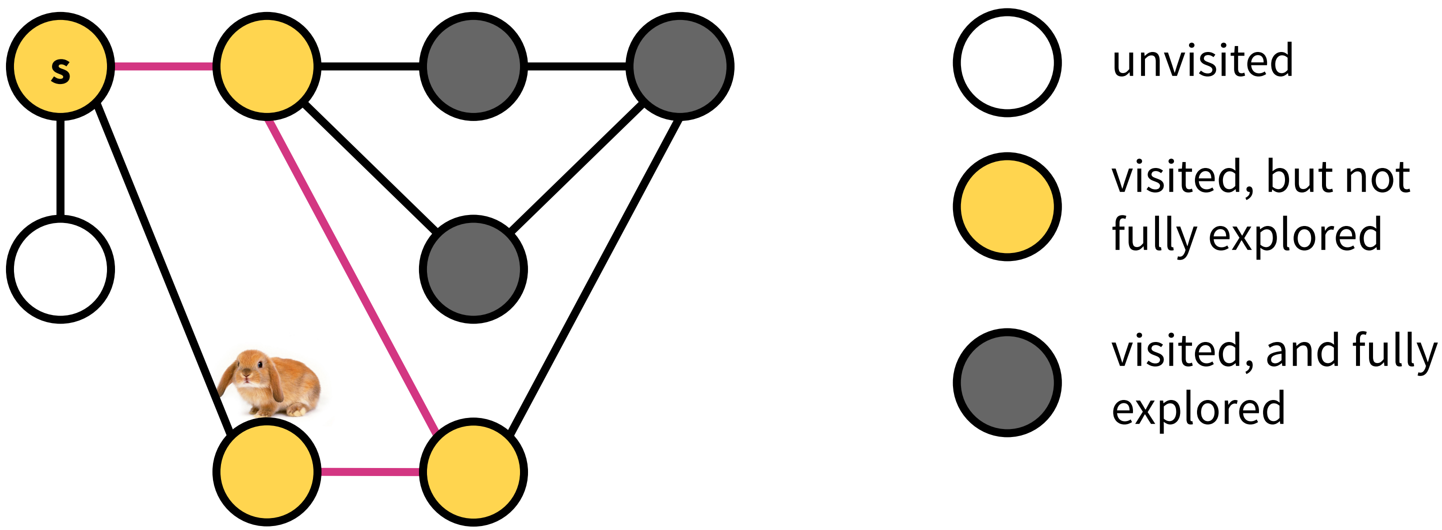
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

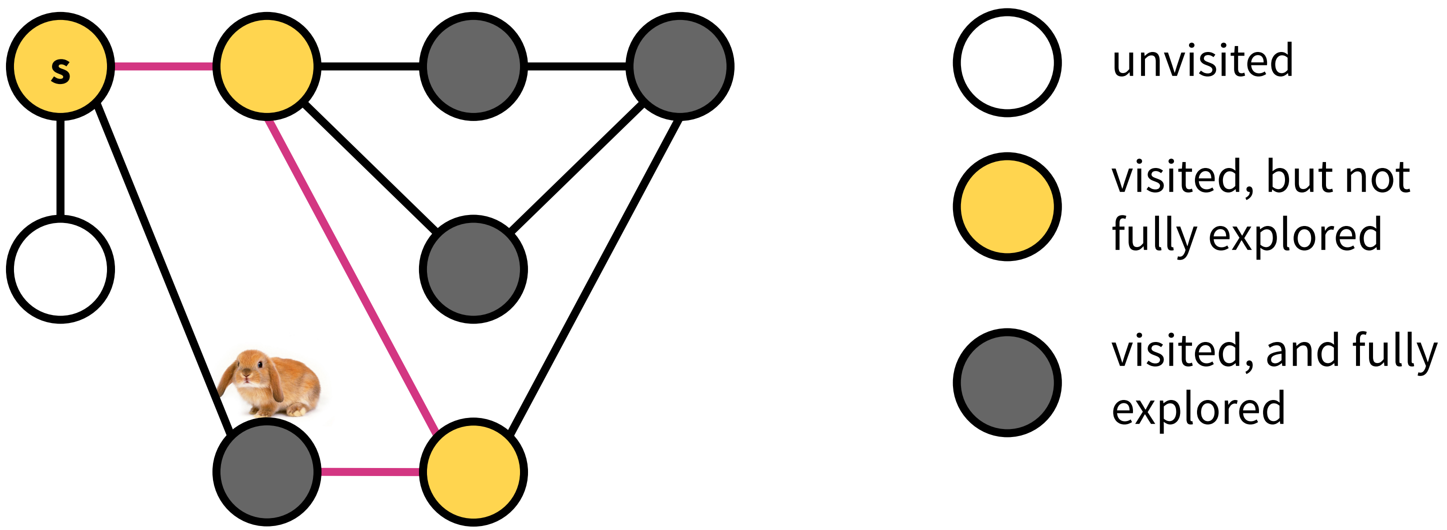
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

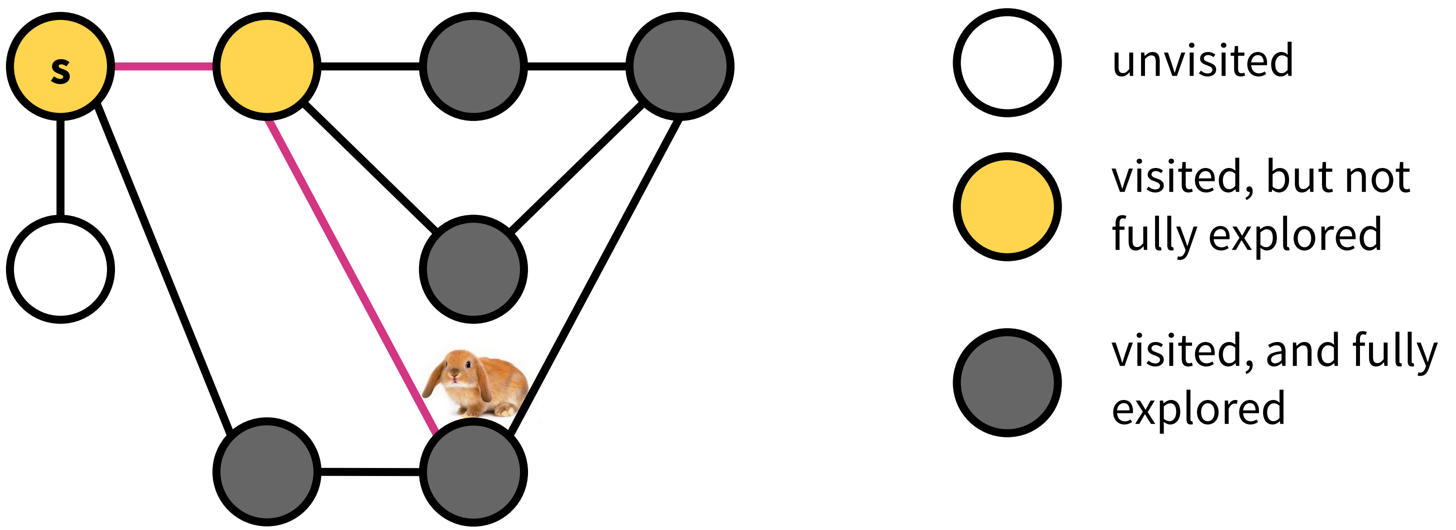
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

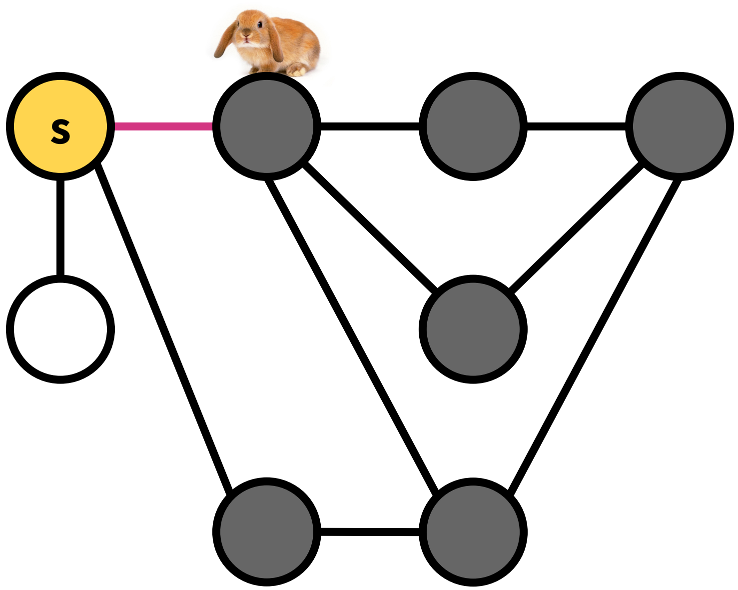
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

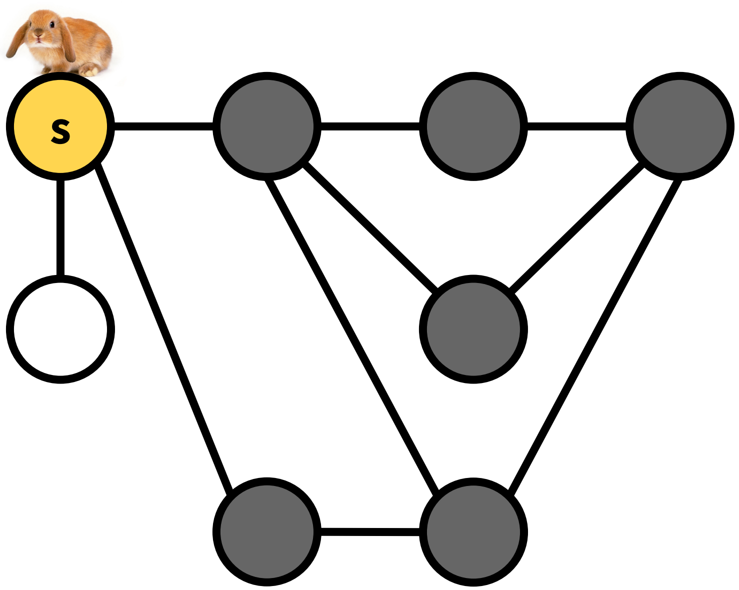
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).




# Depth-First Search

## An analogy

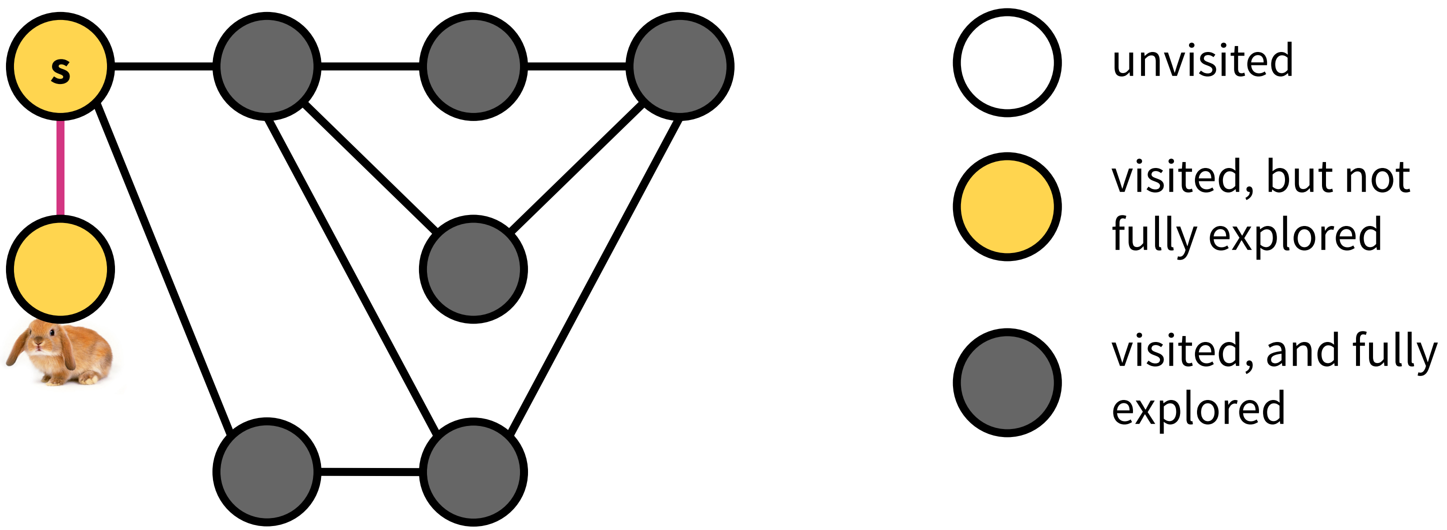
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).




# Depth-First Search

## An analogy

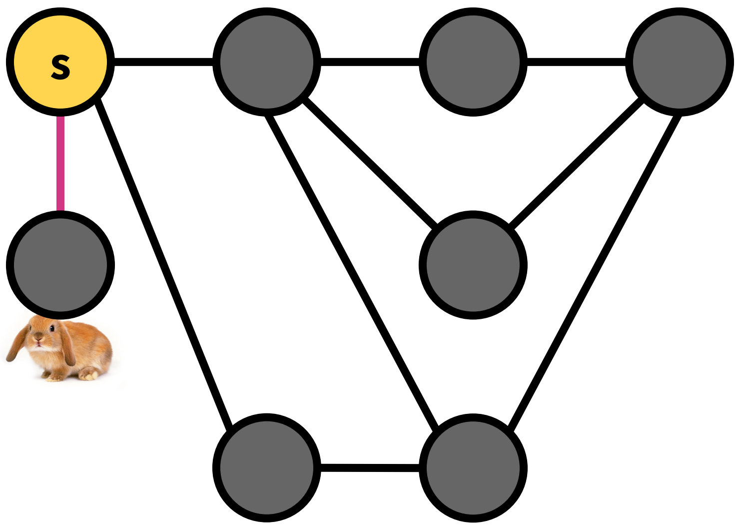
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).

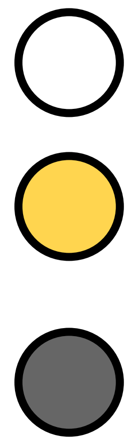


# Depth-First Search

## An analogy

A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).

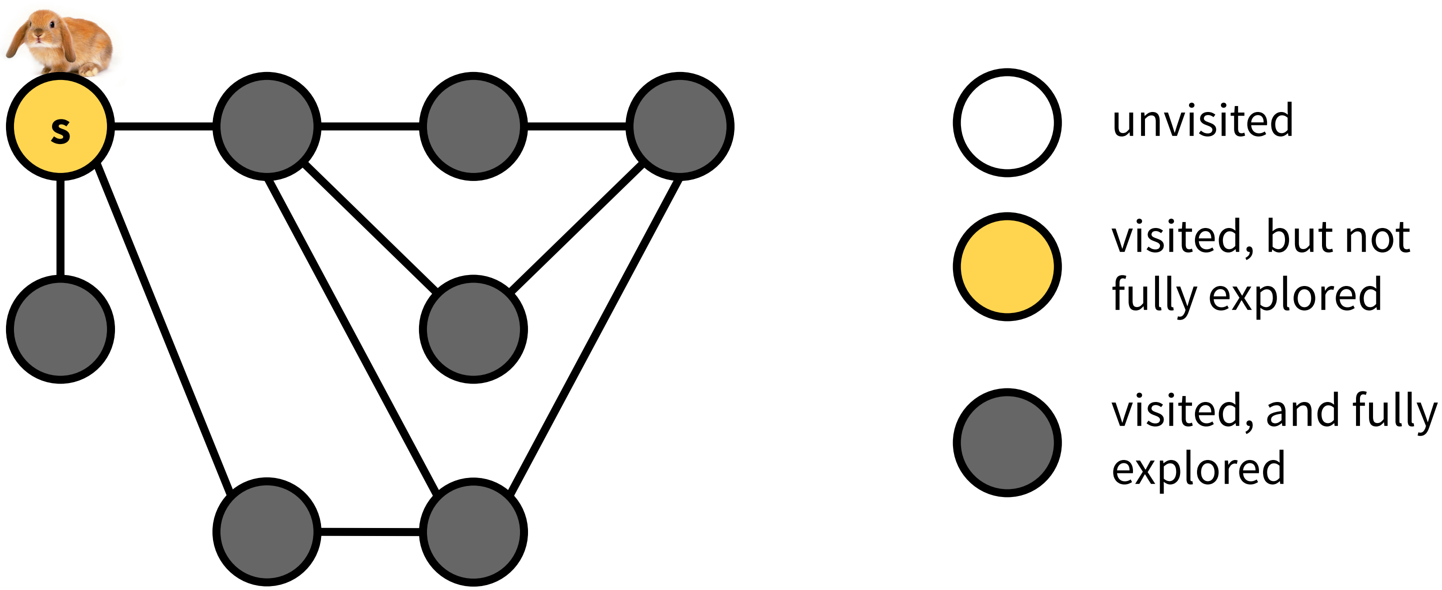


# Depth-First Search

## An analogy

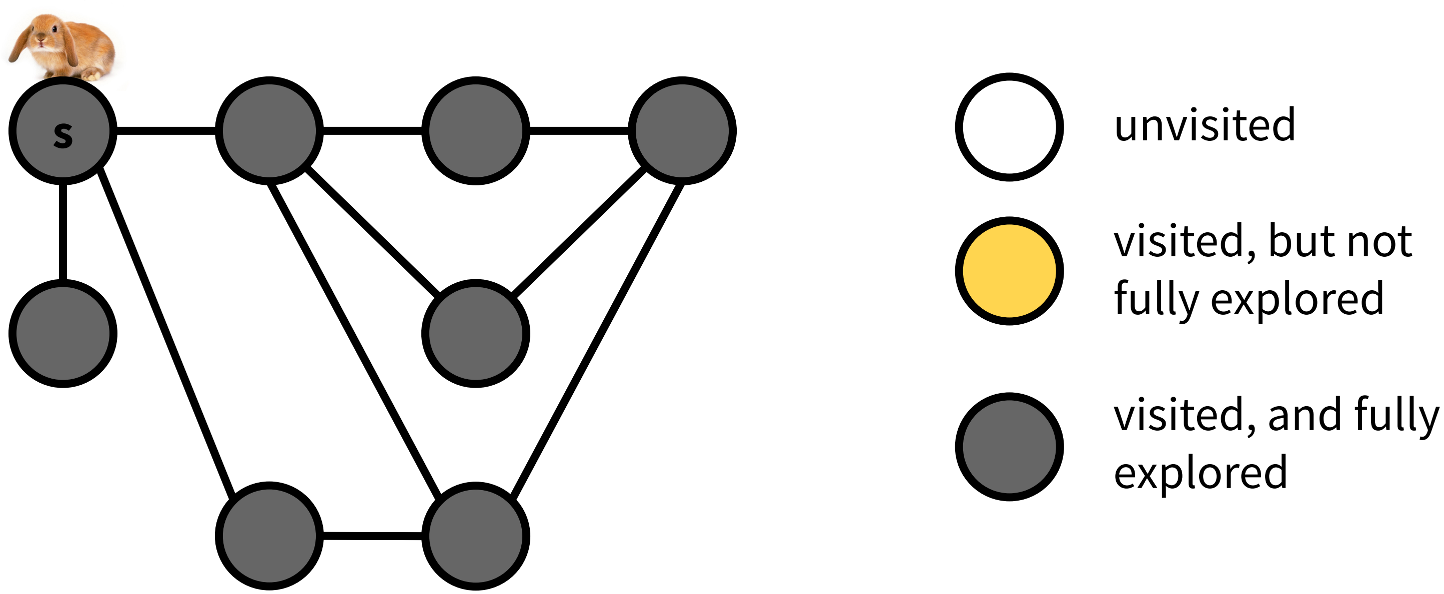
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



# Depth-First Search

## An analogy

A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).

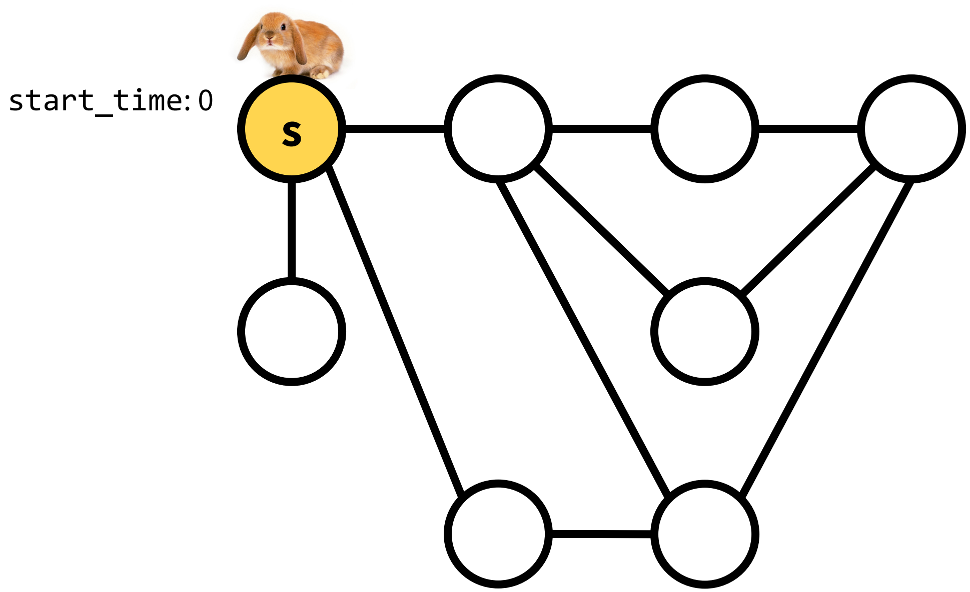


# Depth-First Search

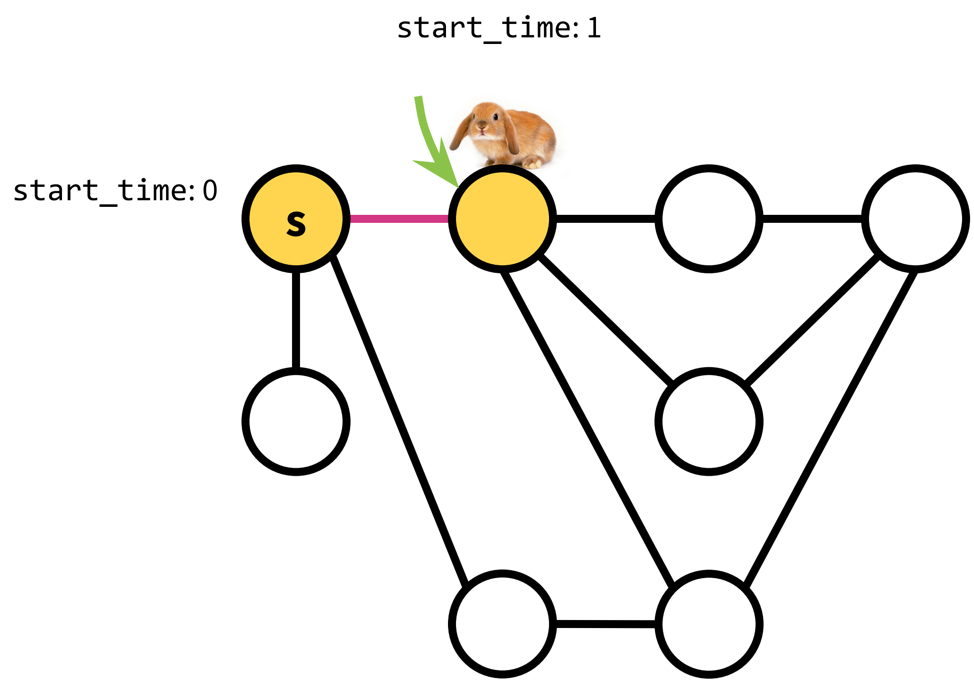
```
def dfs(u, cur_time):
  u.start_time = cur_time
  cur_time += 1
  u.status = "in_progress"
  for v in u.neighbors:
    if v.status is "unvisited":
      cur_time = dfs(v, cur_time)
      cur_time += 1
  u.end_time = cur_time
  u.status = "done"
  return cur_time
```

**Runtime:** $O(|V|+|E|)$

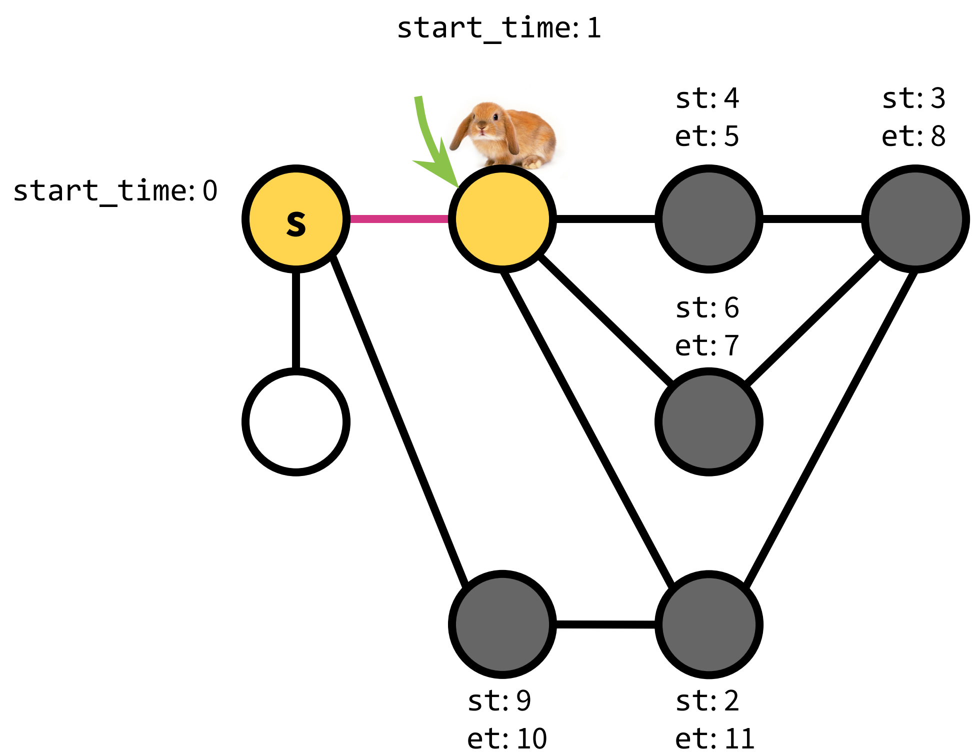
# Depth-First Search



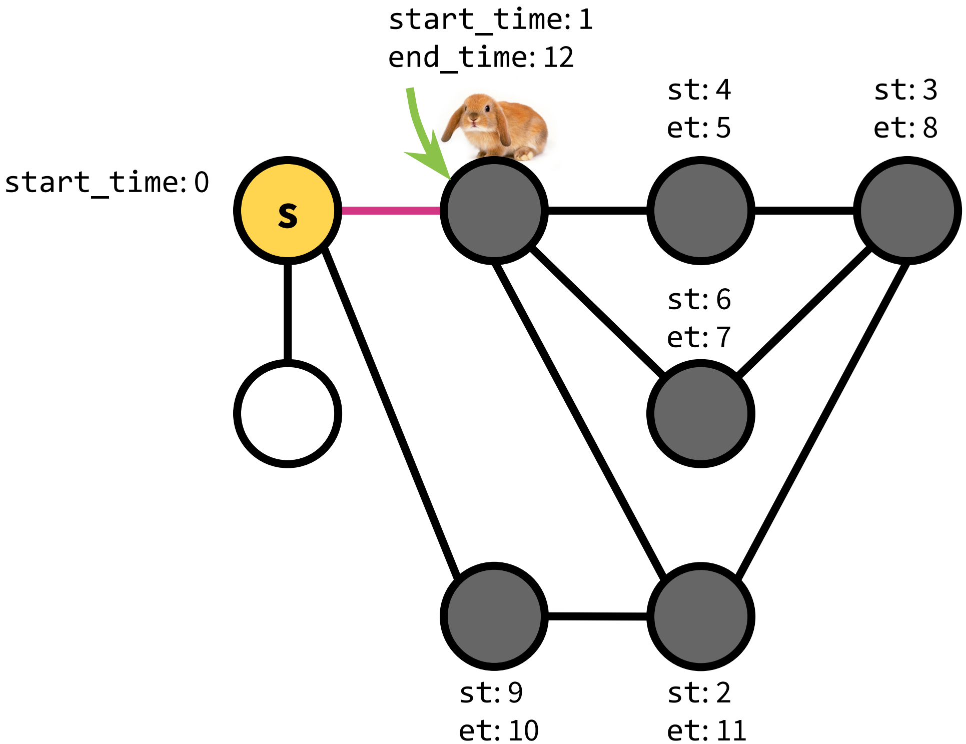
# Depth-First Search



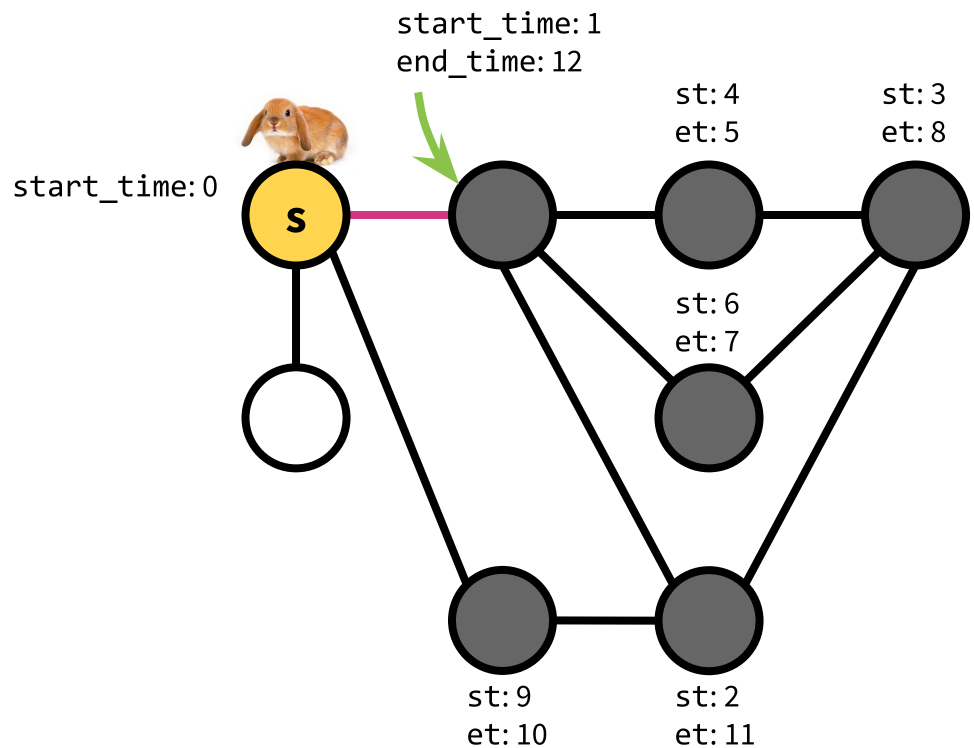
# Depth-First Search



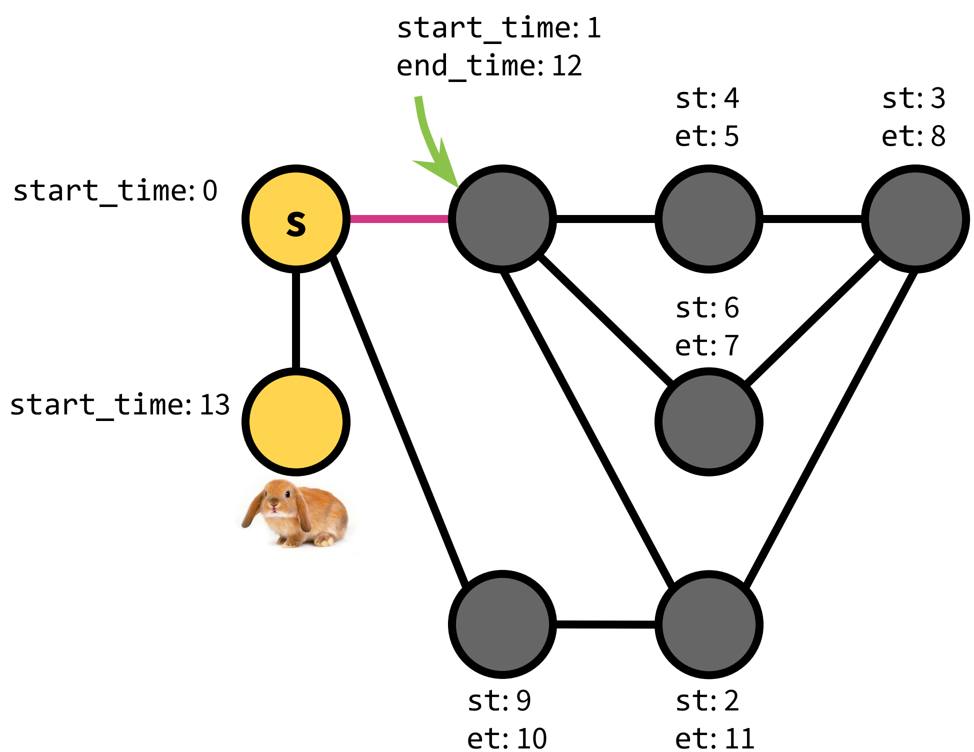
# Depth-First Search



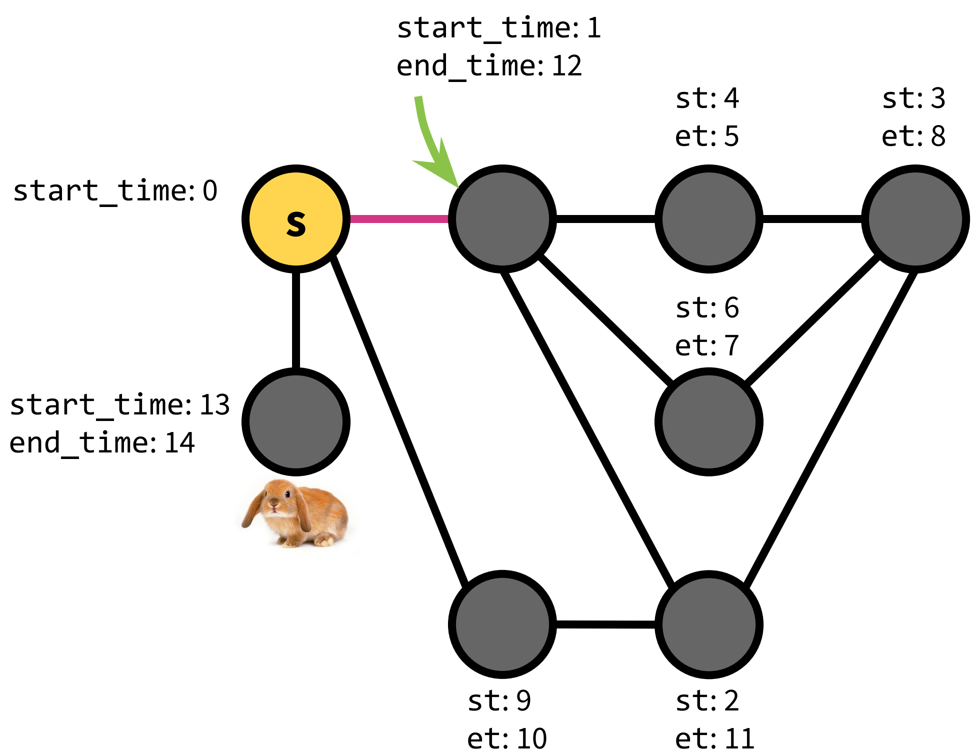
# Depth-First Search



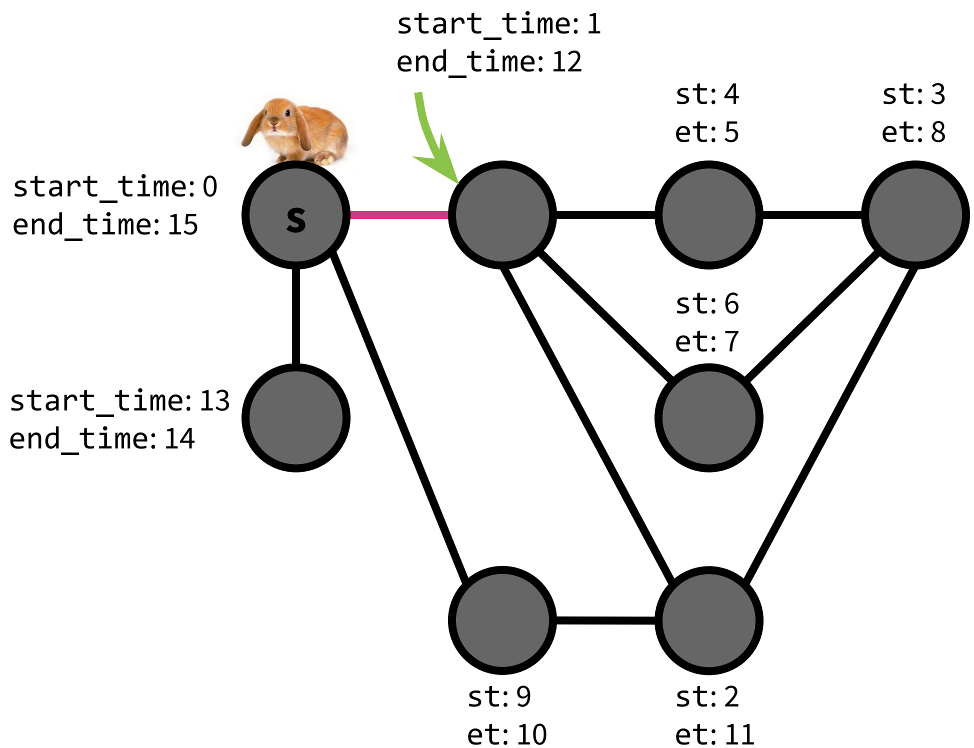
# Depth-First Search



# Depth-First Search



# Depth-First Search

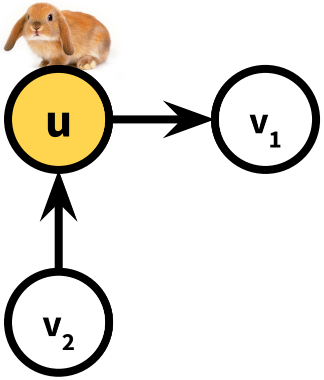


# Depth-First Search

DFS finds all vertices reachable from the starting point, called a **connected component**.

DFS works fine on directed graphs as well.

e.g. From u, only visit $v_1$ not $v_2$.

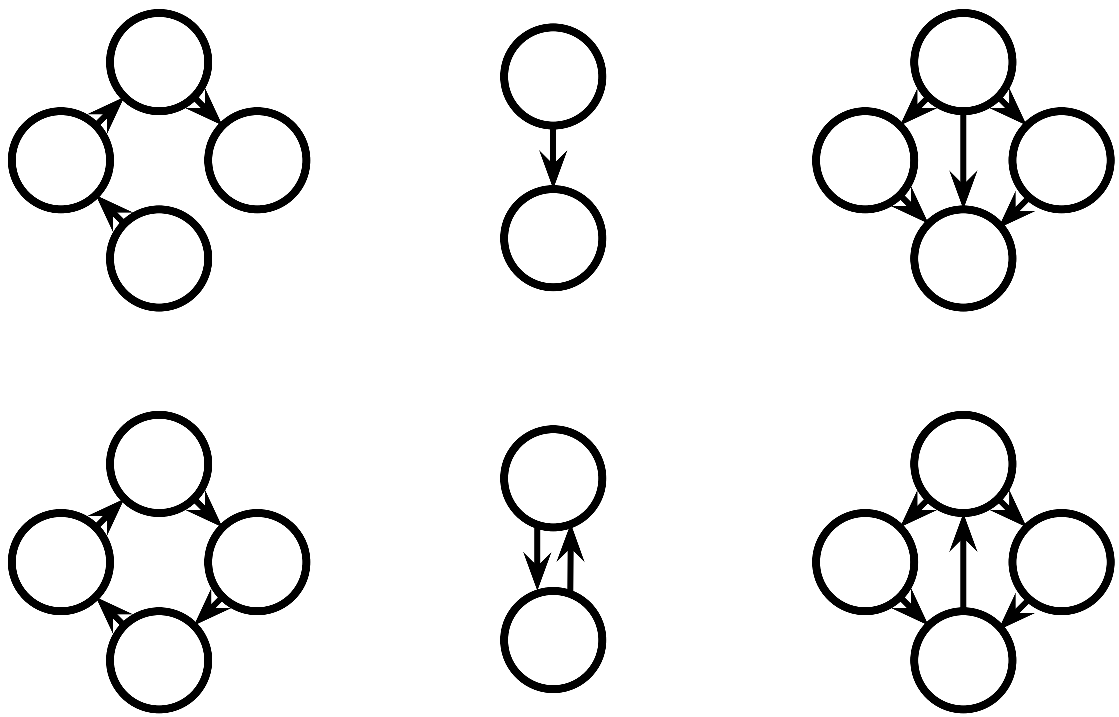


# Topological Ordering

# Aside: Directed Acyclic Graphs

A dependency graph is an instantiation of a directed acyclic graph (DAG) i.e. a directed graph with no directed cycles.

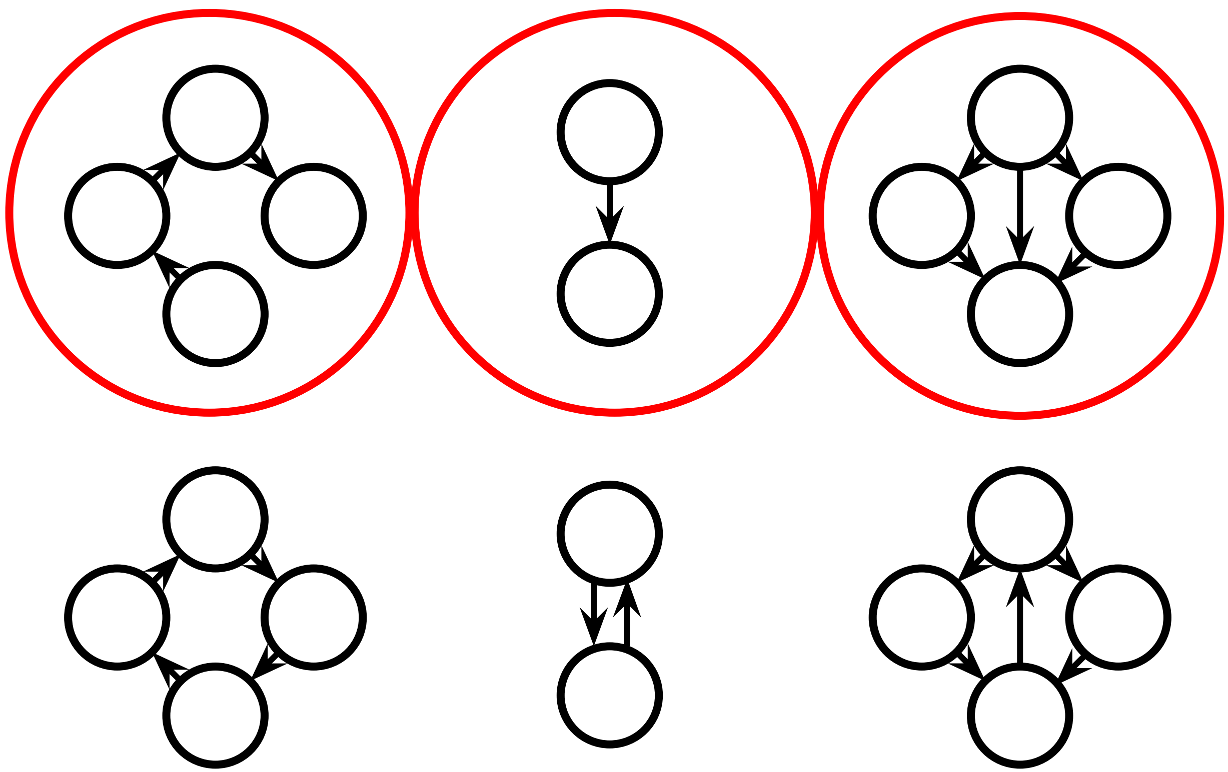
Which of these graphs are valid DAGs? 🤔



# Aside: Directed Acyclic Graphs

A dependency graph is an instantiation of a directed acyclic graph (DAG) i.e. a directed graph with no directed cycles.

Which of these graphs are valid DAGs? 🤔



# Topological Ordering

**Application of DFS:** Given a package dependency graph, in what order should packages be installed?

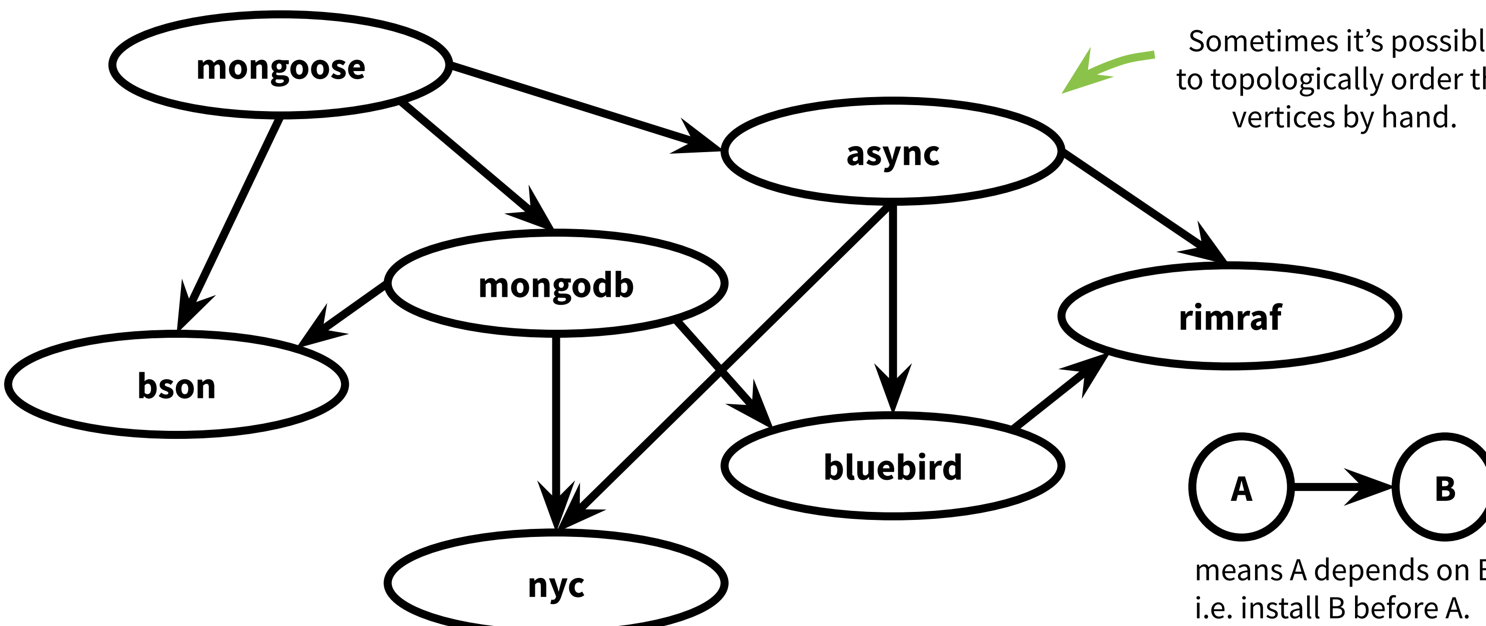
- DFS produces a **topological ordering**, which solves this problem.
- Definition: The topological ordering of a DAG is an ordering of its vertices such that for every directed edge $(u, v) \in E$, u precedes v in the ordering.

# Topological Ordering

**Application of DFS:** Given a package dependency graph, in what order should packages be installed?

DFS produces a **topological ordering**, which solves this problem.

Definition: The topological ordering of a DAG is an ordering of its vertices such that for every directed edge (u, v) ∈ E, u precedes v in the ordering.




Sometimes it's possible to topologically order the vertices by hand.

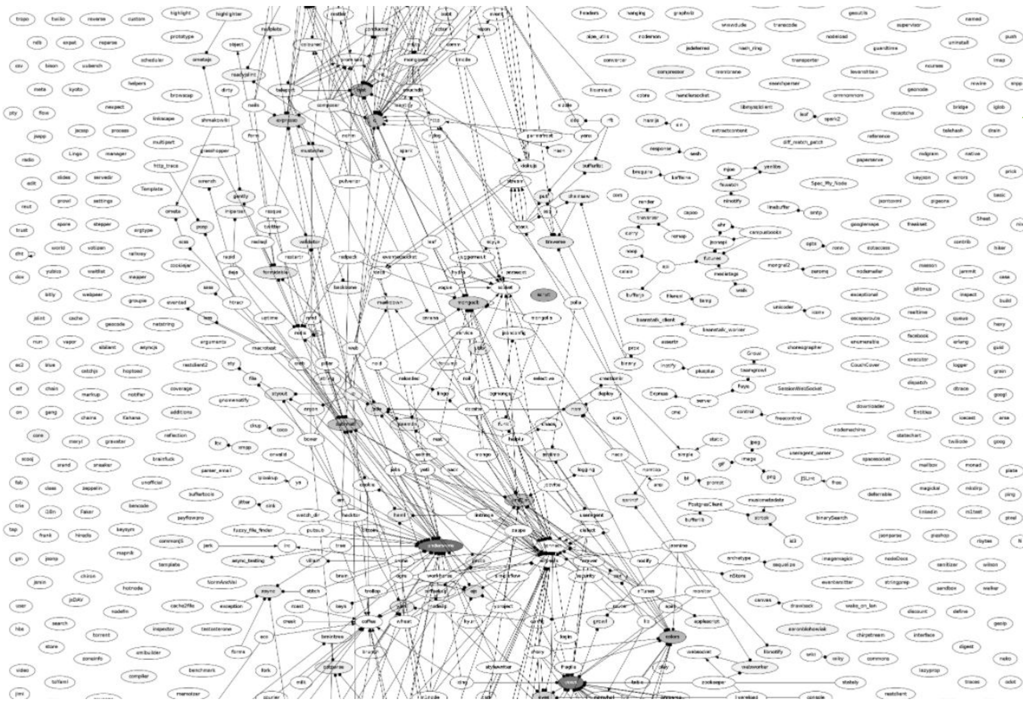
means A depends on B
i.e. install B before A.

# Topological Ordering

**Application of DFS:** Given a package dependency graph, in what order should packages be installed?

DFS produces a **topological ordering**, which solves this problem.

Definition: The topological ordering of a DAG is an ordering of its vertices such that for every directed edge $(u, v) \in E$, $u$ precedes $v$ in the ordering.



Sometimes it's not …

# Topological Ordering

**Claim:** If (u, v) ∈ E, then `end_time` of u > `end_time` of v.

**Intuition:** `dfs` visits and finishes with all of the neighbors of u before finishing u itself. Also, a DAG does not have cycles, so `dfs` will never traverse to an in-progress vertex (only unvisited and done vertices).

# Topological Ordering

```
def dfs(u, cur_time):
  u.start_time = cur_time
  cur_time += 1
  u.status = "in_progress"
  for v in u.neighbors:
    if v.status is "unvisited":
      cur_time = dfs(v, cur_time)
      cur_time += 1
  u.end_time = cur_time
  u.status = "done"
  return cur_time
```

**Runtime:** $O(|V|+|E|)$

# Topological Ordering

```
reversed_topological_list = []
def dfs(u, cur_time):
  u.start_time = cur_time
  cur_time += 1
  u.status = "in_progress"
  for v in u.neighbors:
    if v.status is "unvisited":
      cur_time = dfs(v, cur_time)
      cur_time += 1
  u.end_time = cur_time
  u.status = "done"
  reversed_topological_list.append(u)
  return cur_time
```

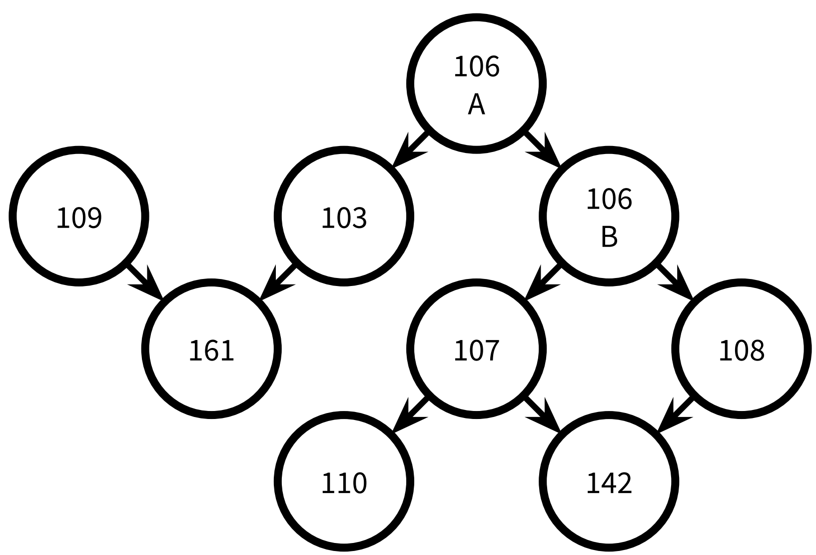
**Runtime:** O(|V|+|E|)

# Topological Ordering

For the package dependency graph, packages should be installed in reverse topological order, so we can just return `reversed_topological_list`.

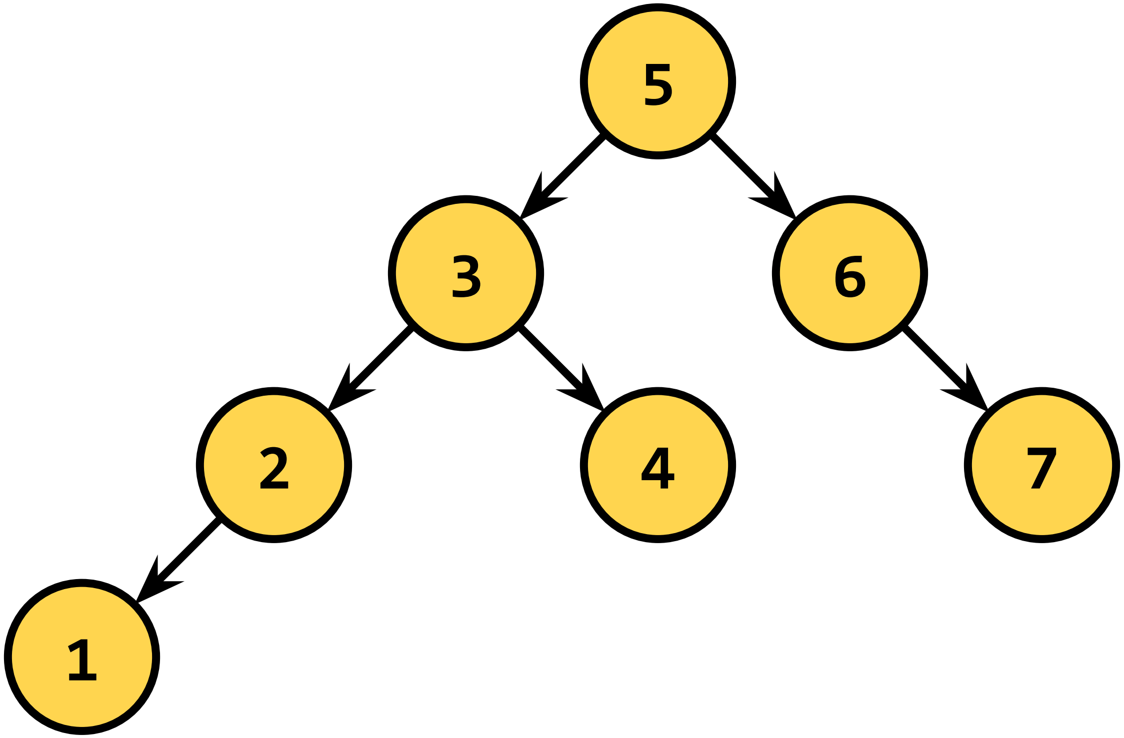
To compute the topological ordering in general, reverse the order of `reversed_topological_list`.

e.g. Finding an order to take courses that satisfies prerequisites.



# In-Order Traversal of BSTs

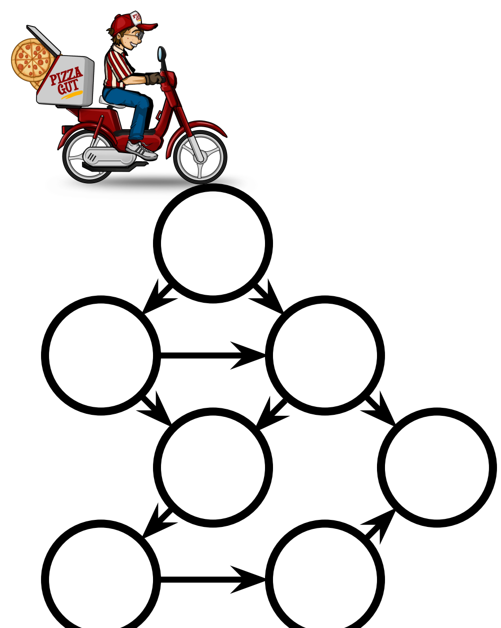
**Application of DFS:** Given a BST, output the vertices in order.



# Exact Traversals of Graphs

**Application of DFS:** Find an exact traversal, a path that touches all vertices exactly once.

Suppose I deliver pizzas in SF. My route has 6 stops but since I bike and the terrain is hilly, I can only bike from one stop to another in one direction. Can I plan the most efficient route that visits each destination once?

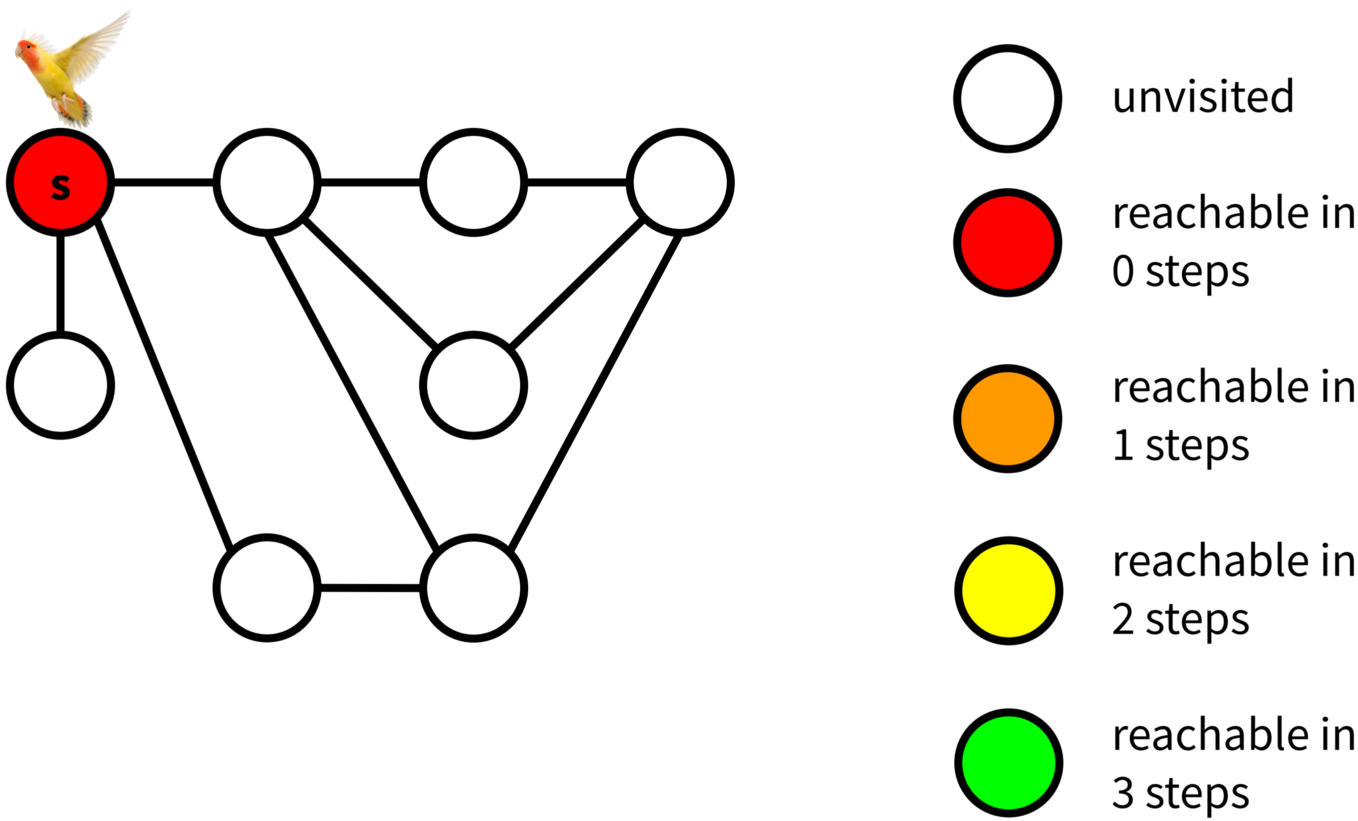


# Breadth-First Search

# Breadth-First Search

## An analogy

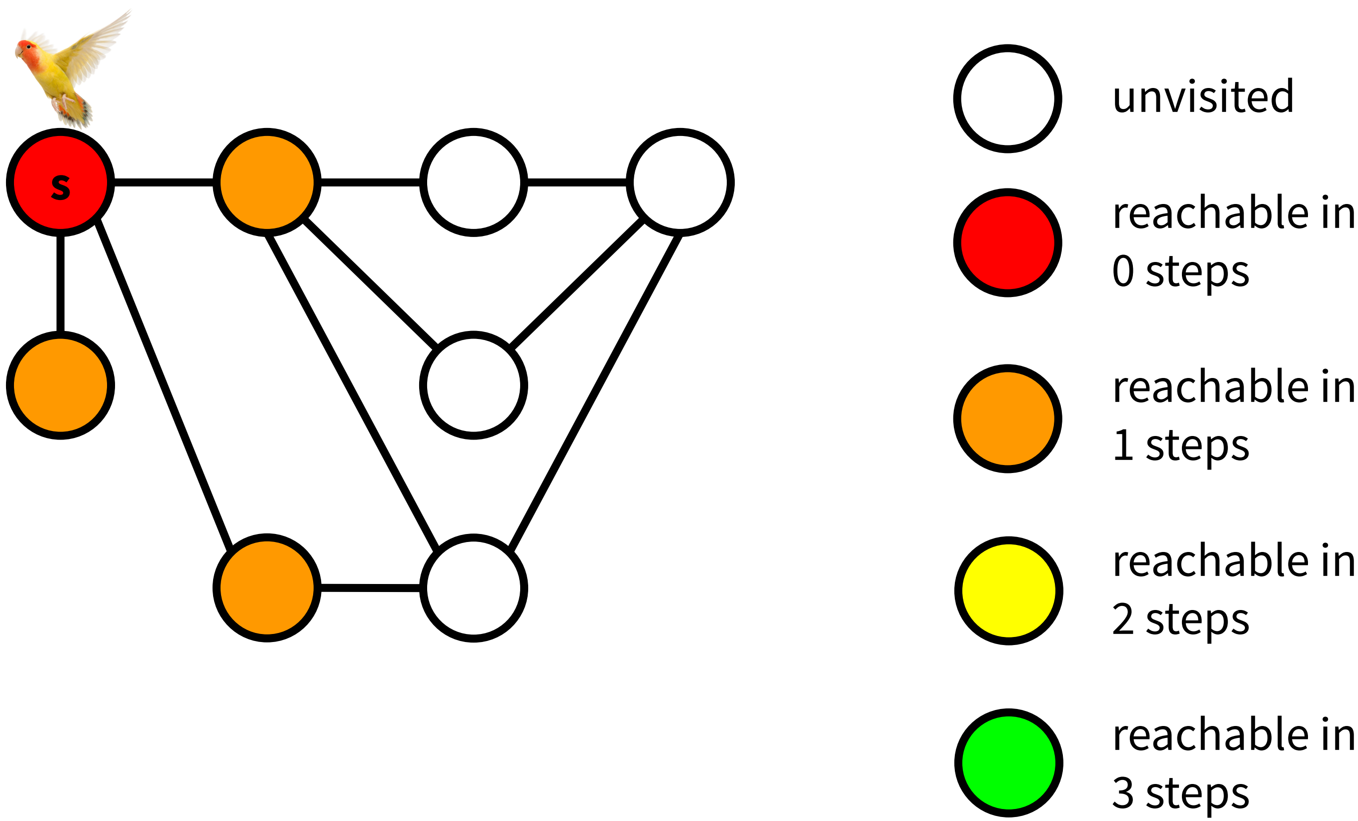
A bird exploring a labyrinth from above (with a bird's eye view).



# Breadth-First Search

## An analogy

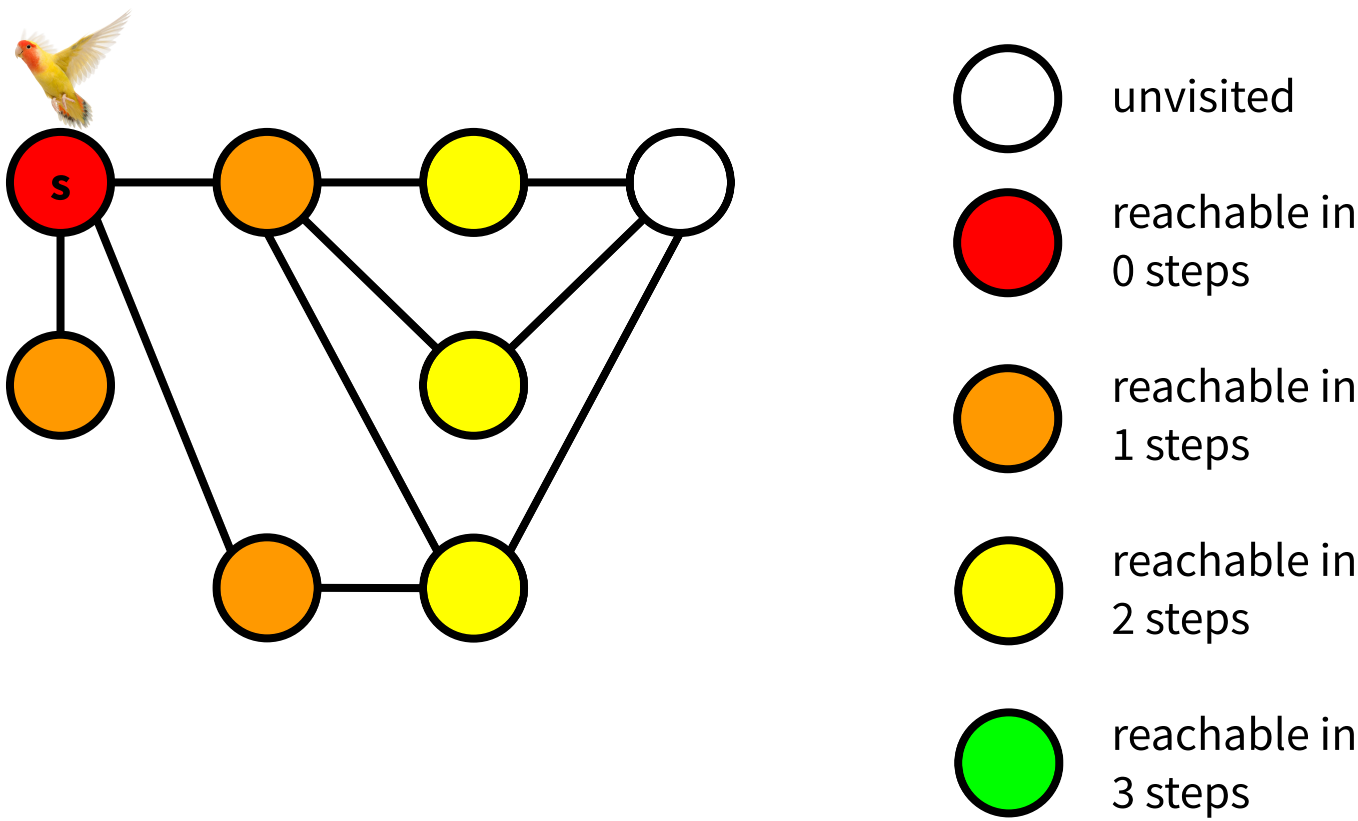
A bird exploring a labyrinth from above (with a bird's eye view).



# Breadth-First Search

## An analogy

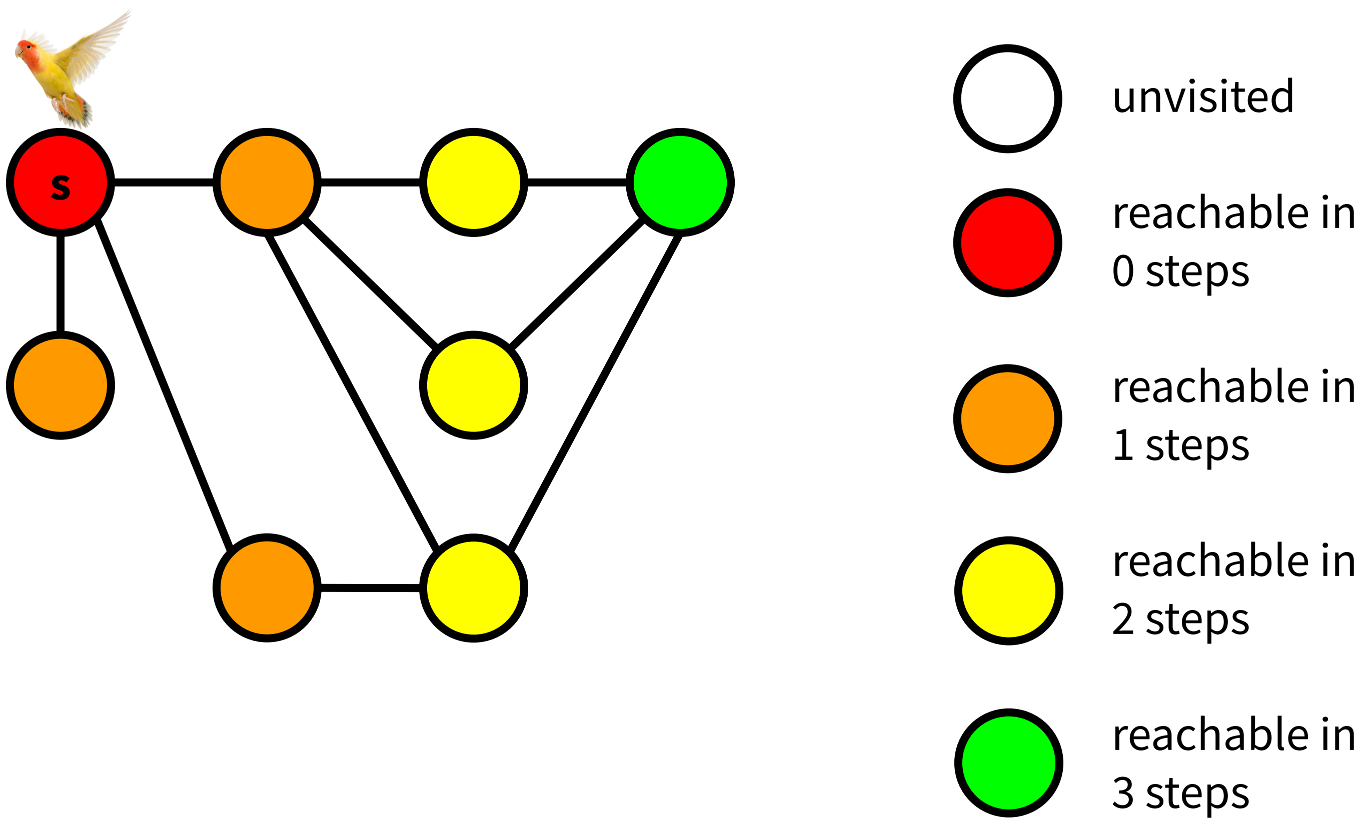
A bird exploring a labyrinth from above (with a bird's eye view).



# Breadth-First Search

## An analogy

A bird exploring a labyrinth from above (with a bird's eye view).



# Breadth-First Search

```
def bfs(s):
  L = []
  for i = 0 to n-1:
    L[i] = {}
  L[0] = {s}
  for i = 0 to n-1:
    for u in L[i]:
      for v in u.neighbors:
        if v.status is "unvisited":
          v.status = "visited"
          L[i+1].add(v)
```

**Runtime:** $O(|V|+|E|)$

# Shortest Path

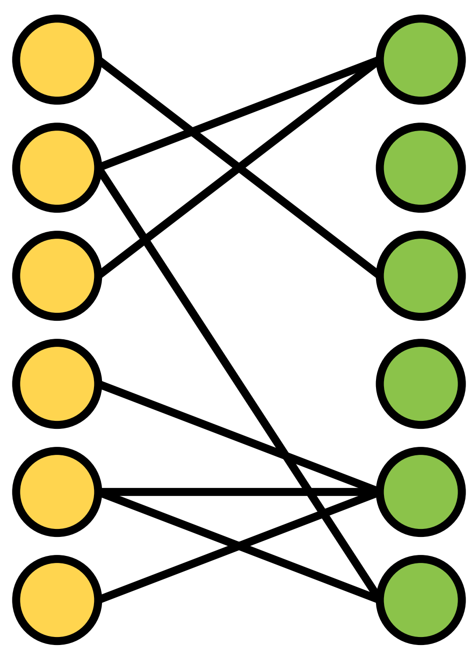
**Application of BFS:** How long is the shortest path between vertices u and v?

- Call `bfs(u)`.
- For all vertices in `L[i]`, the shortest path between u and these vertices has length i.
- If v isn't in `L[i]` for any i, then it's unreachable from u.

# Aside: Bipartiteness

A graph is **bipartite** iff there exists a two-coloring such that there are no edges between same-colored vertices.

e.g. Matching university hackathon guests and hosts.

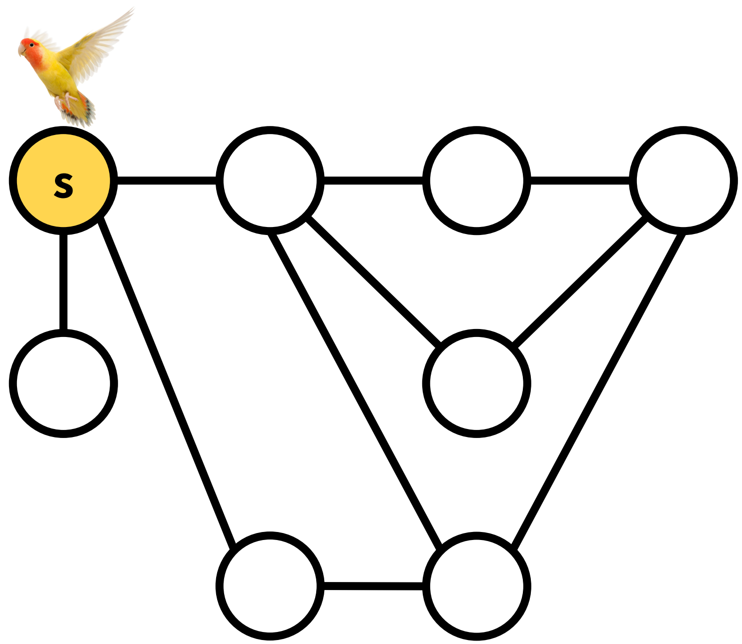


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

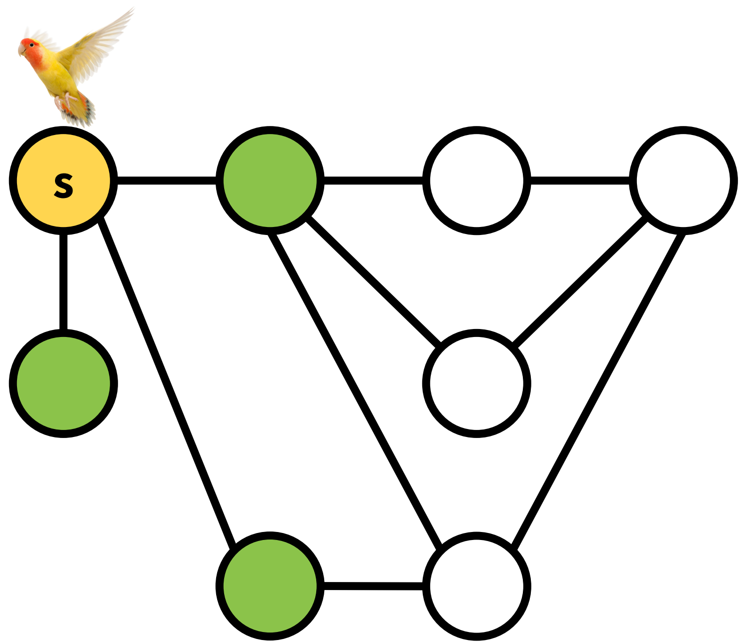


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

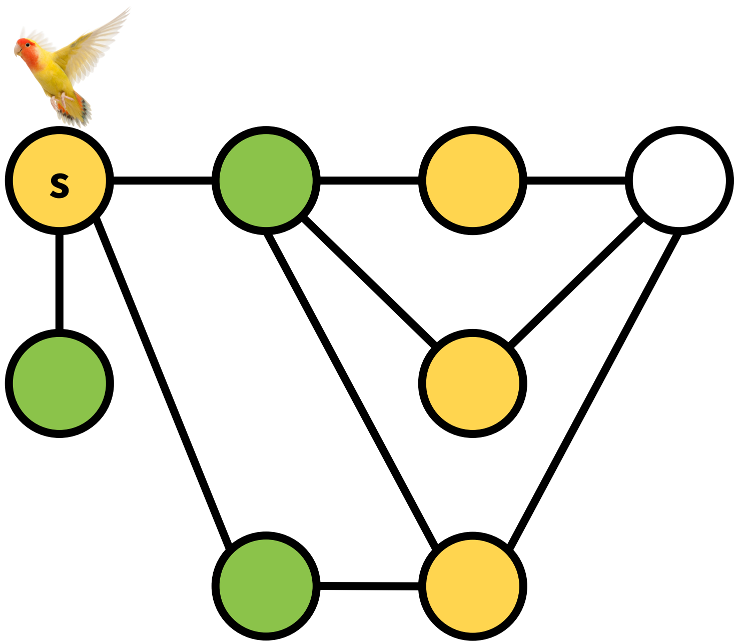


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

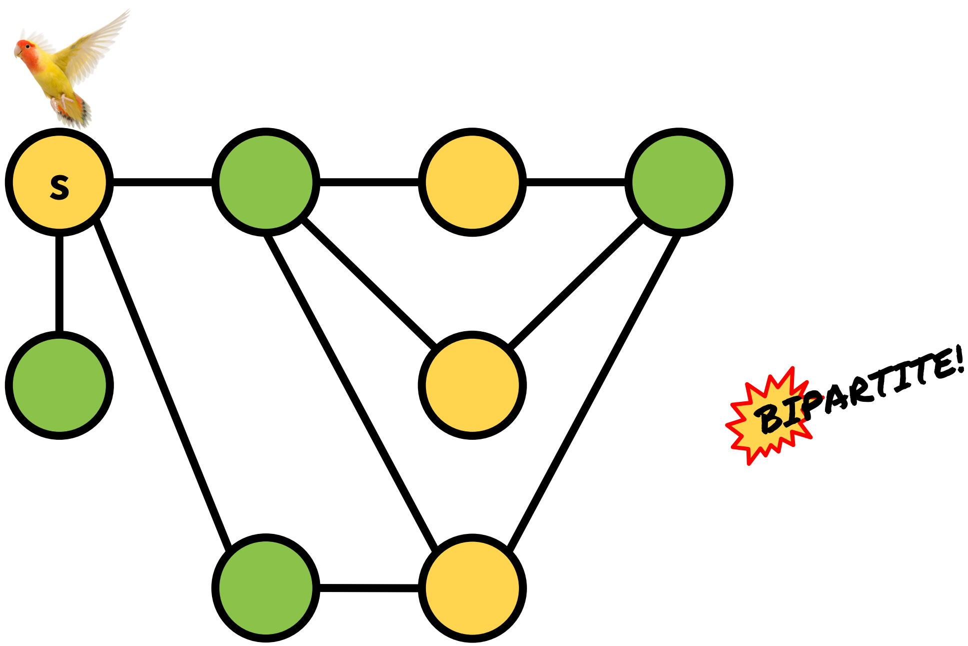


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

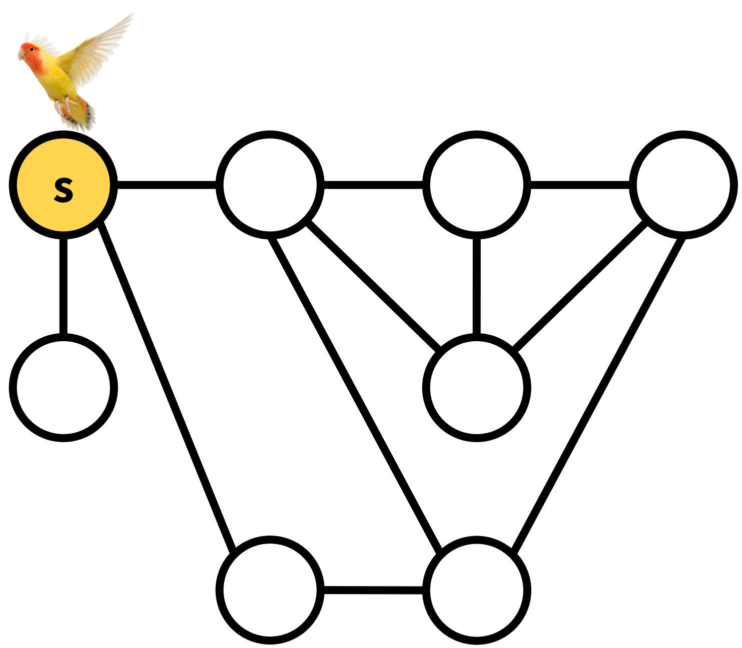


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

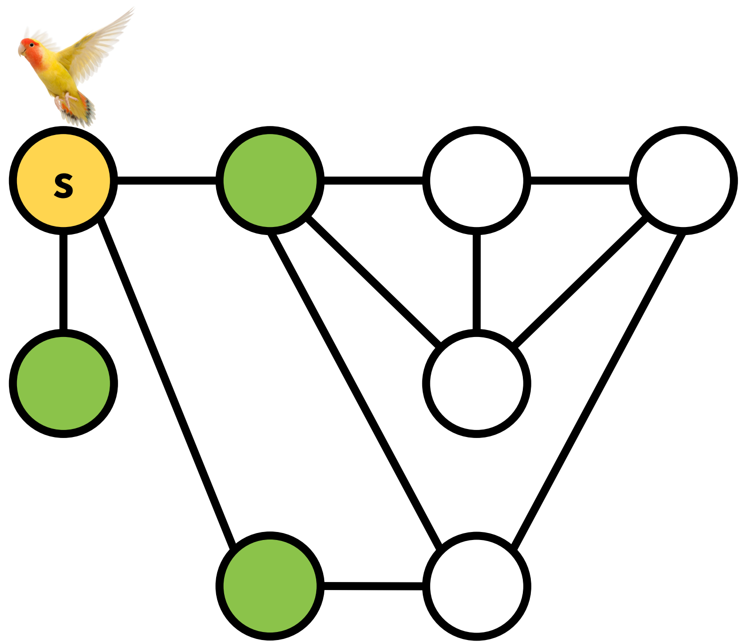


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

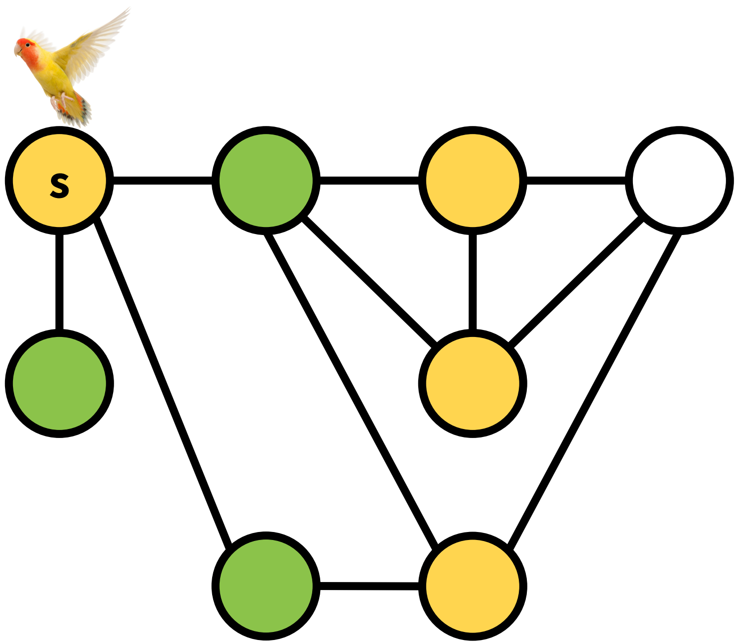


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

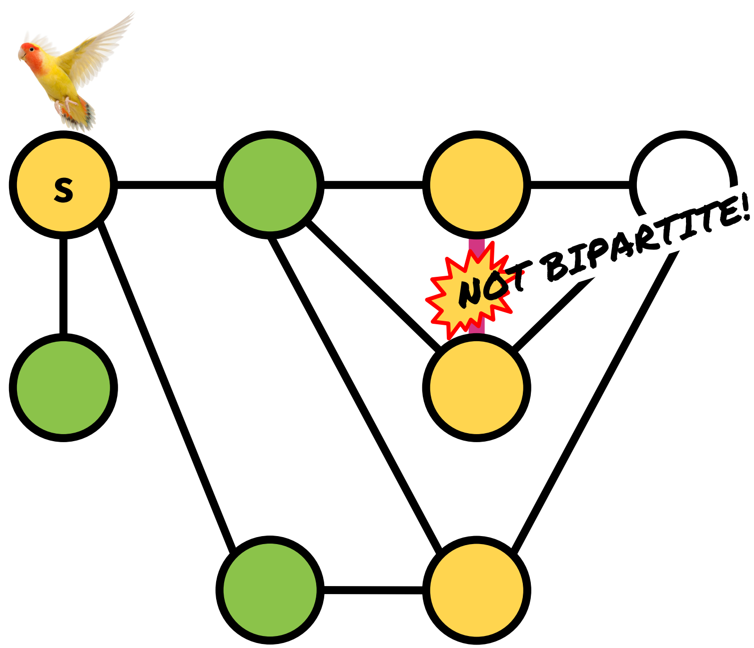


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

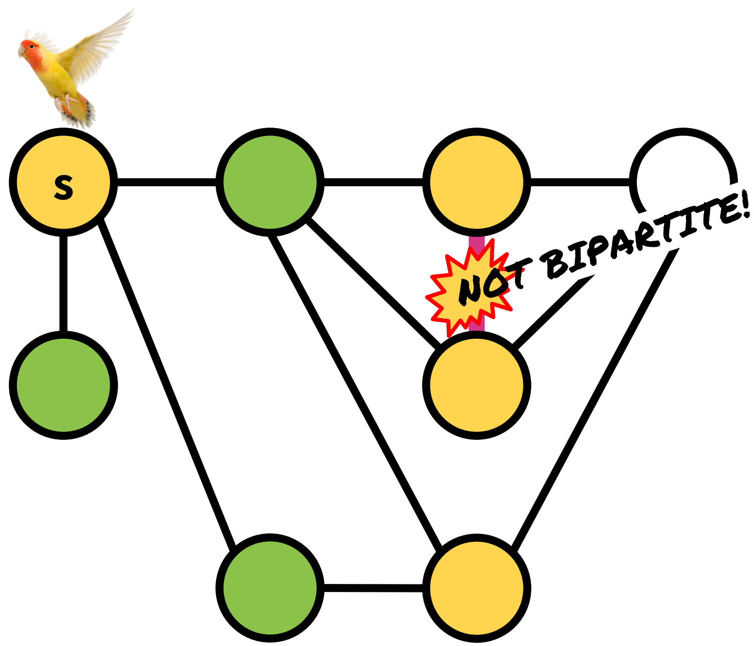


# Shortest Path

**Application of BFS:** Is a graph bipartite?

Call `bfs` from any vertex and color vertices alternating colors.

If it attempts to color the same vertex different colors, then the graph isn't bipartite; otherwise it is.

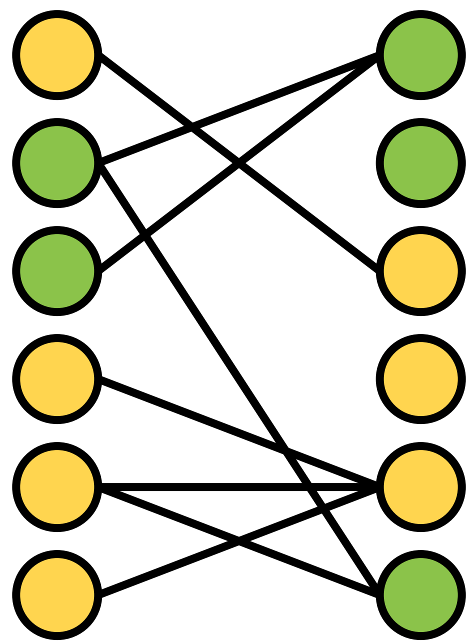



Is anyone incredulous that this actually works? 🤔

# Shortest Path

There exist many poor colorings on legitimate bipartite graphs.

- Just because **this** coloring that doesn't work, why does that mean that **no** coloring works? 🤔



This is a poor coloring on an obviously bipartite graph.

# Shortest Path

**Theorem:** `bfs` colors two neighbors the same color iff the graph is not bipartite.

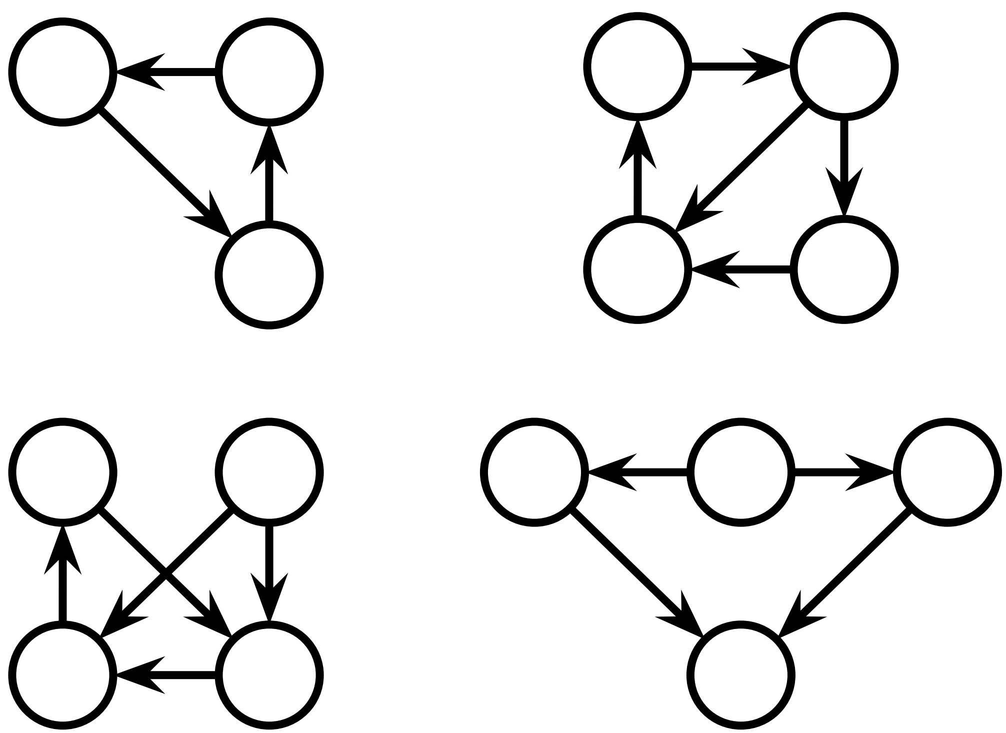
**Proof:**

Since `bfs` colors vertices alternating colors, it colors two neighbors the same color iff it's found a cycle of odd length in the graph. Therefore, the graph contains an odd cycle as a subgraph. But it's impossible to color an odd cycle with two colors such that no two neighbors have the same color. Therefore, it's impossible to two-color the graph such that the no edges between same-colored vertices, and the graph must not be bipartite.

# 3 min break

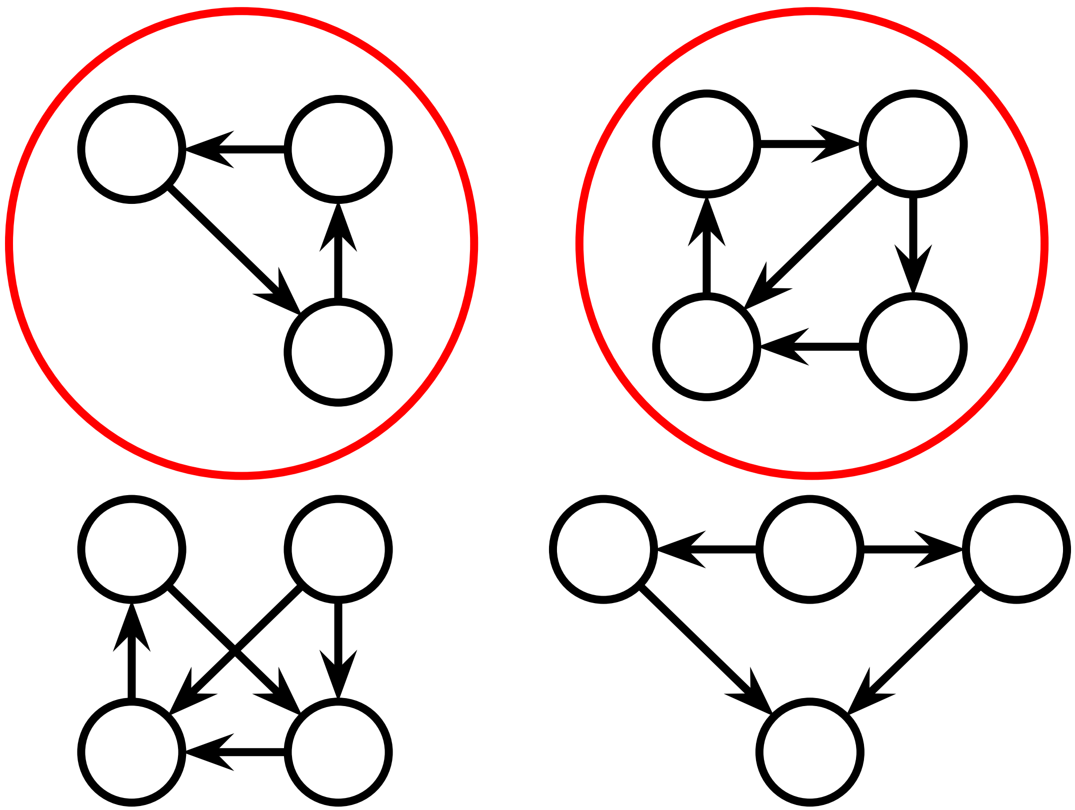
# Strongly Connected Components

A directed graph G = (V, E) is strongly connected if,
for all pairs of vertices u and v, there's a path from u to v
and a path from v to u.



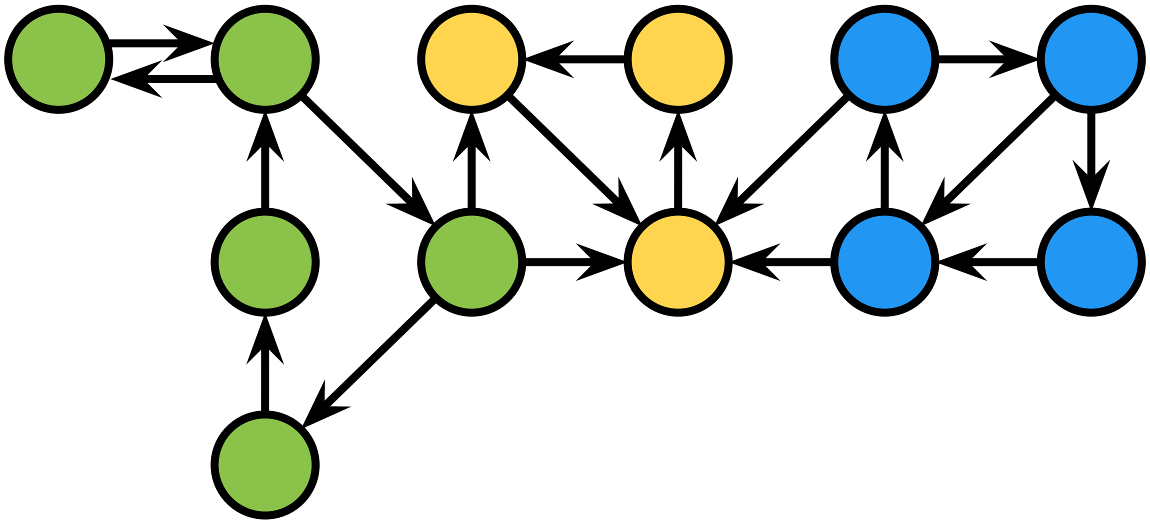
# Strongly Connected Components

A directed graph G = (V, E) is strongly connected if,
for all pairs of vertices u and v, there's a path from u to v
and a path from v to u.



# Strongly Connected Components

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



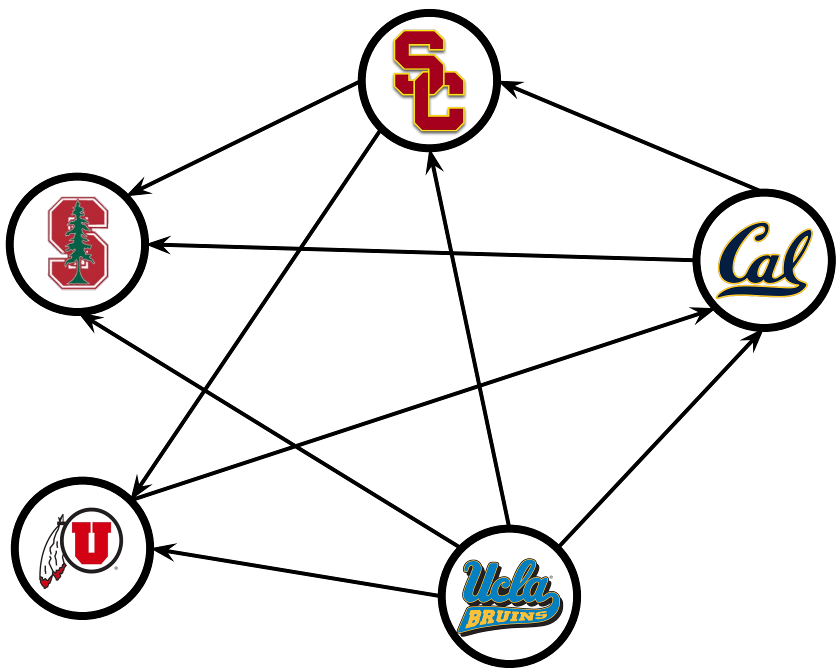
# Strongly Connected Components

Why do we care about SCCs?

- SCCs provide information about communities.
  - A computer scientist might want to decompose the Internet into SCCs to find related topics.
  - An economist might want to decompose labor market data into SCCs before making sense of it.
  - A football executive might want to determine which Pac-12 school should play in the Rose Bowl.

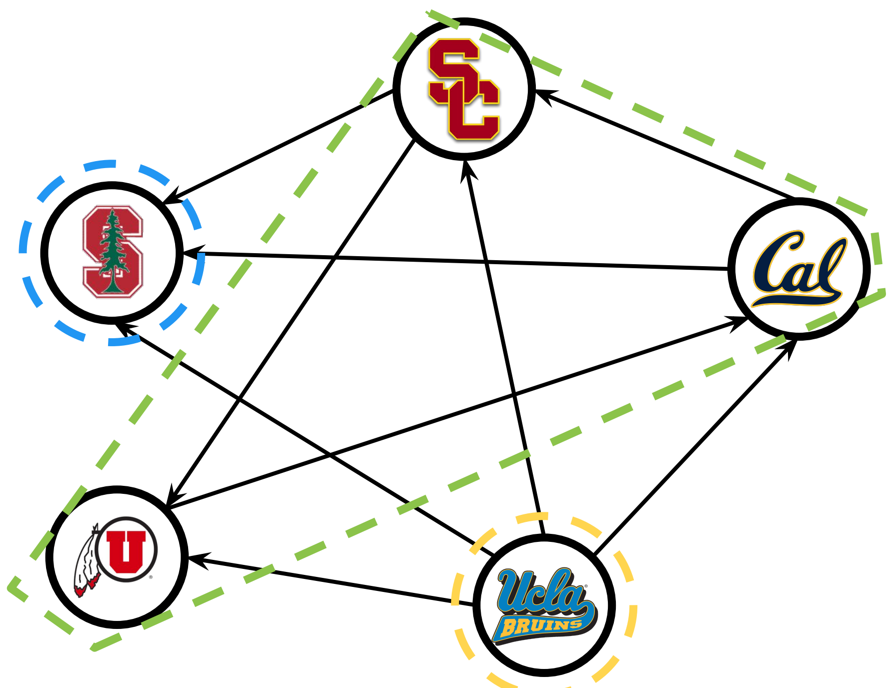
# Strongly Connected Components

How many SCCs are in this graph? 🤔



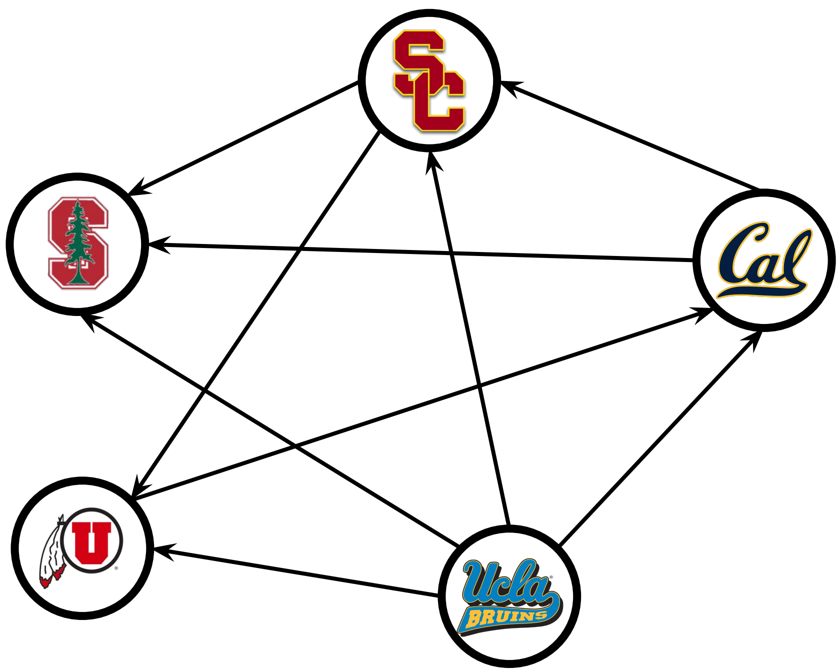
# Strongly Connected Components

How many SCCs are in this graph? 🤔 3; let's find them!



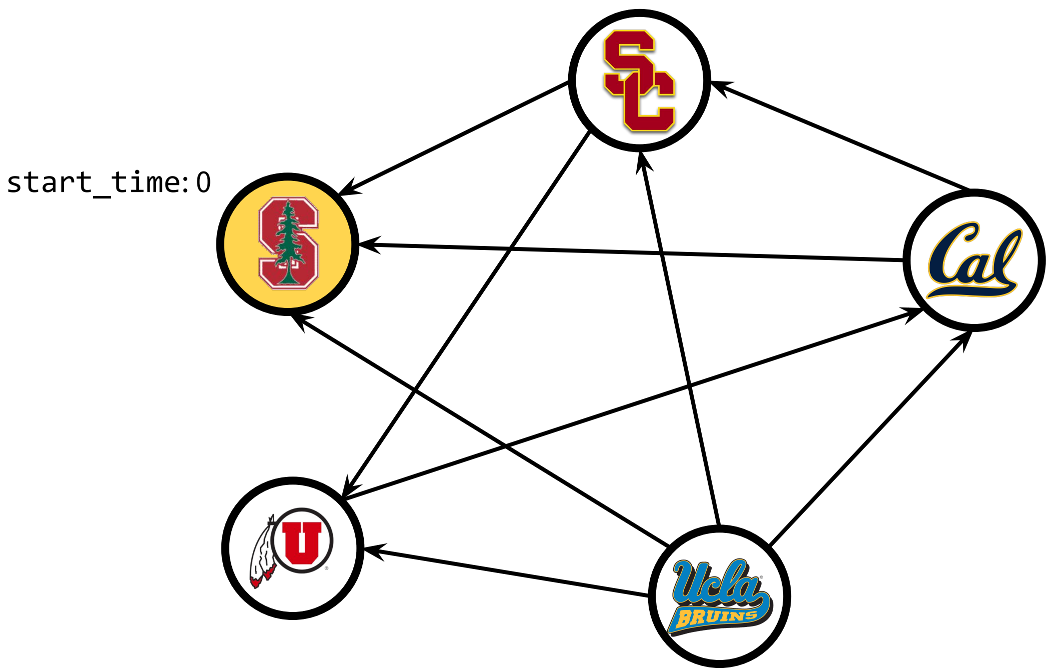
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



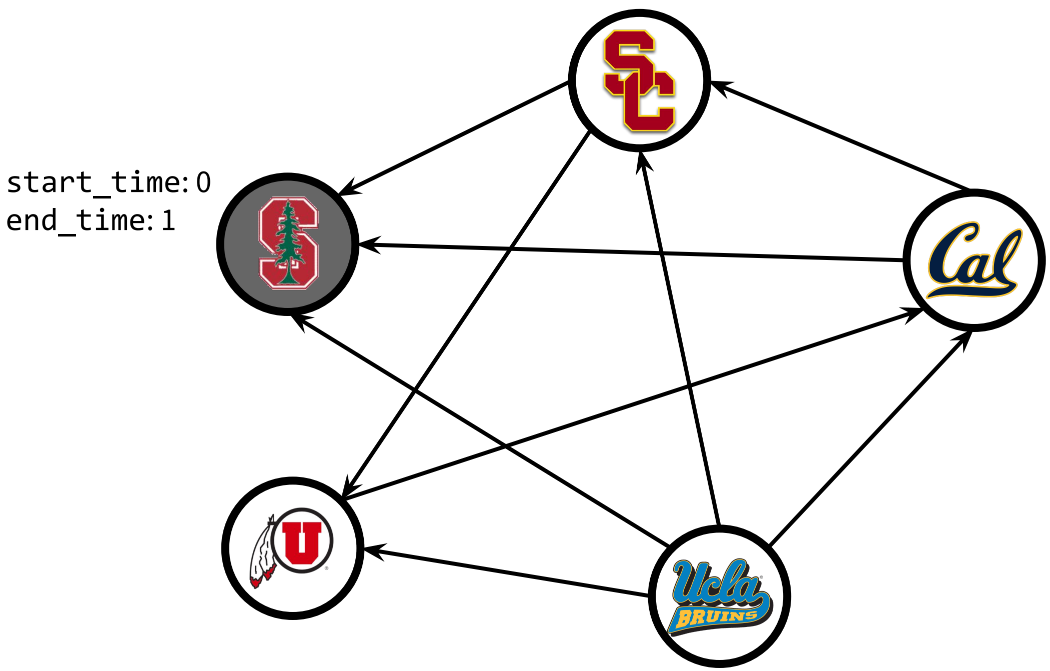
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



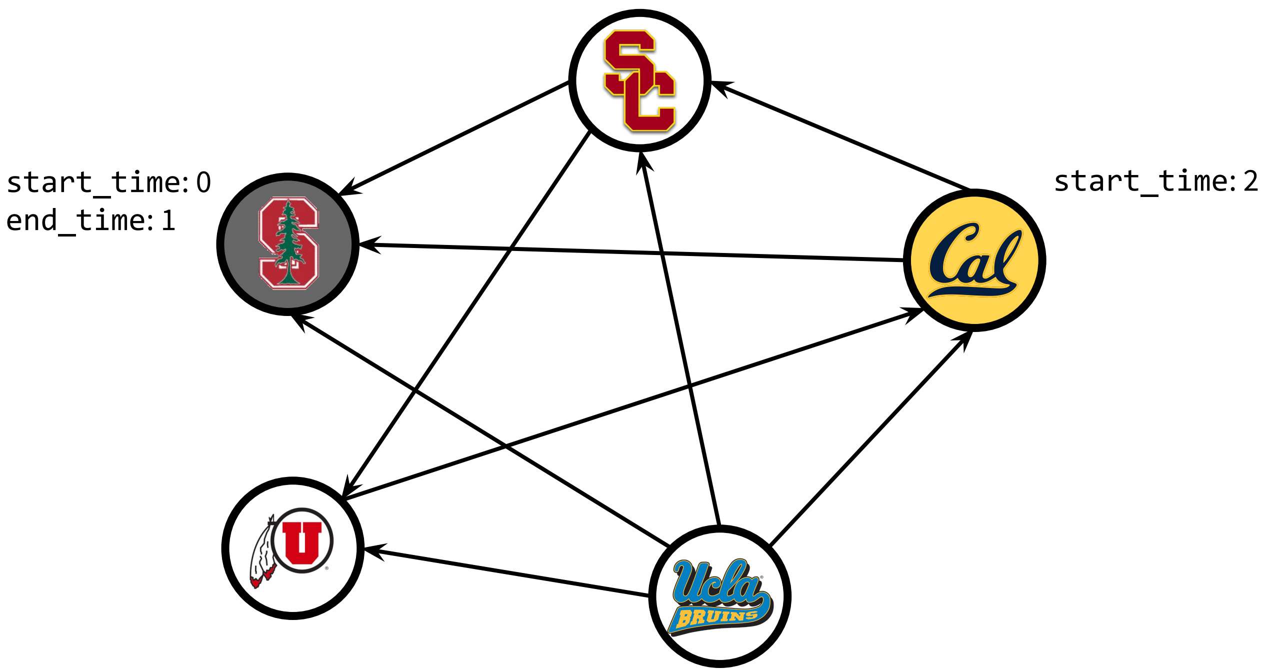
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



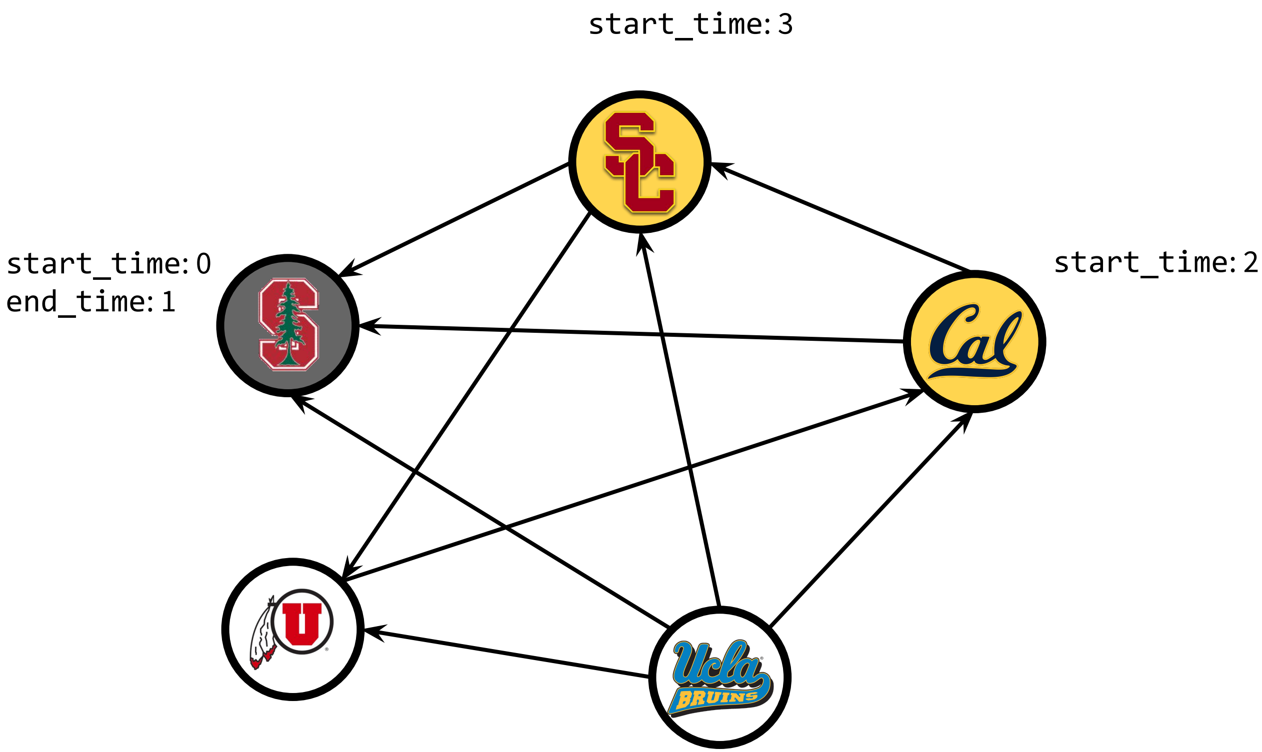
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



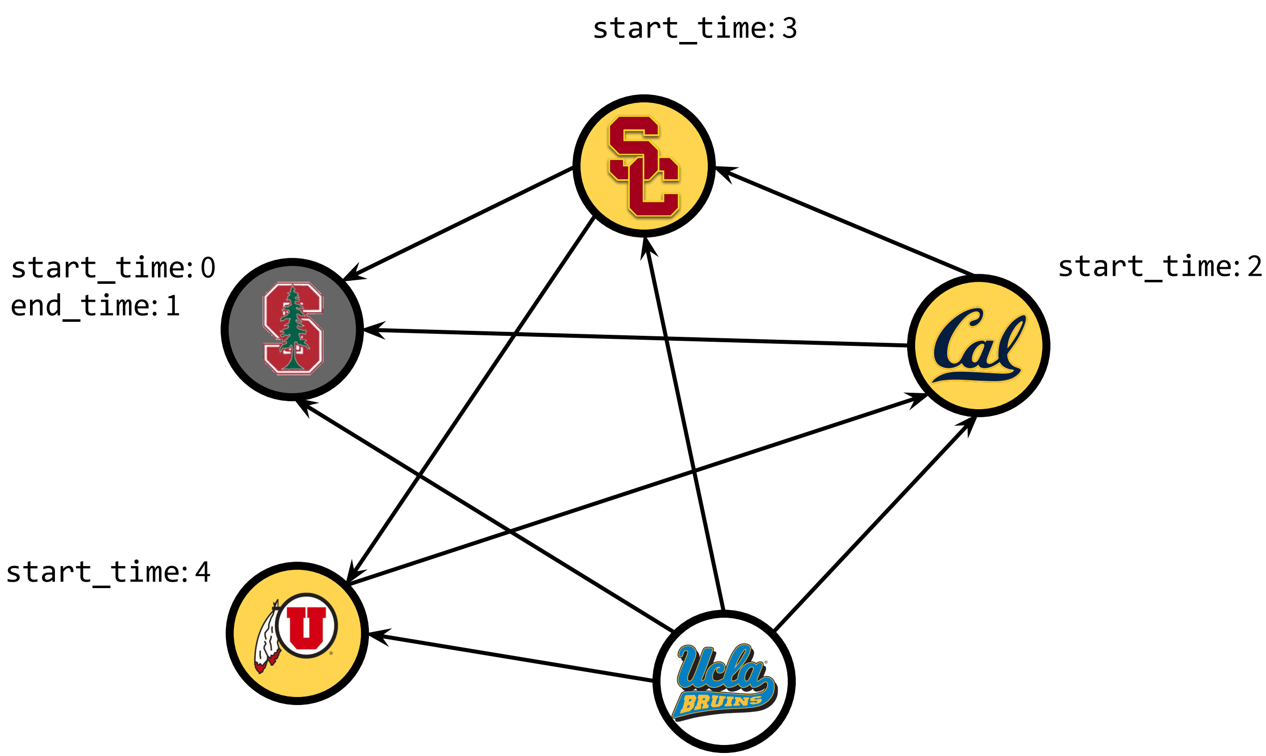
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



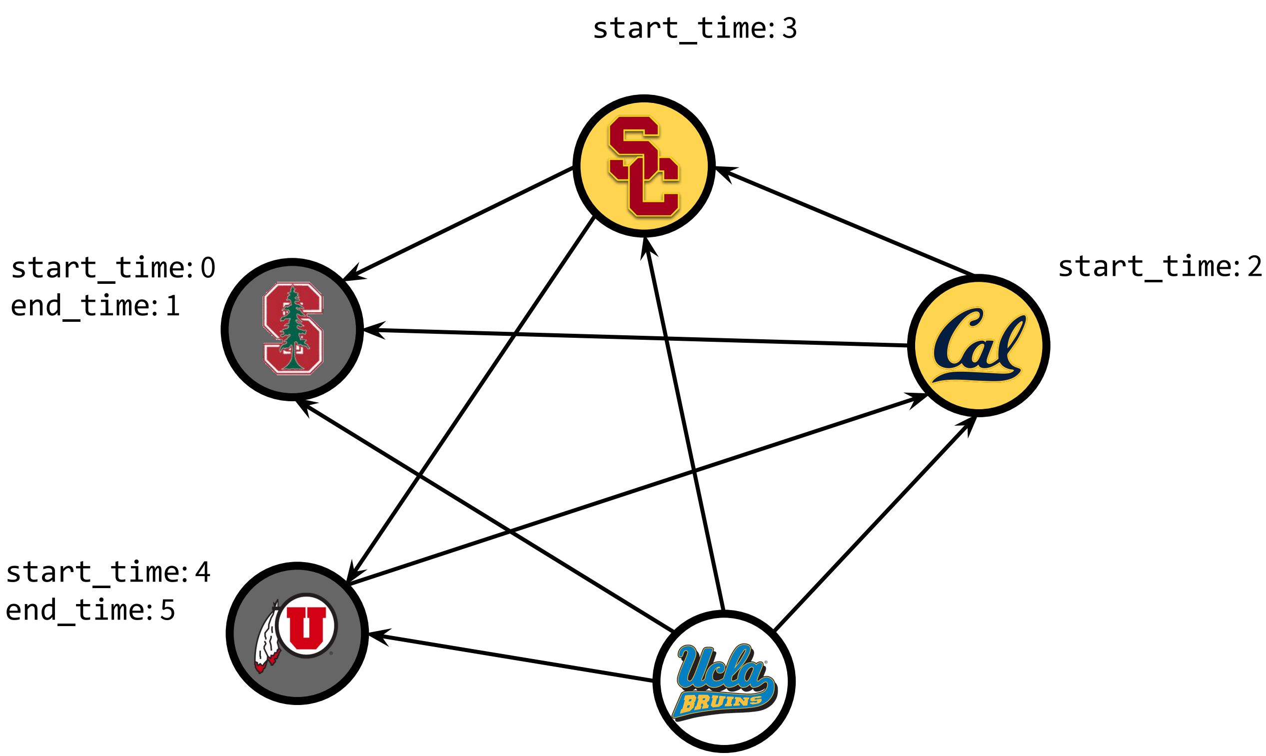
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



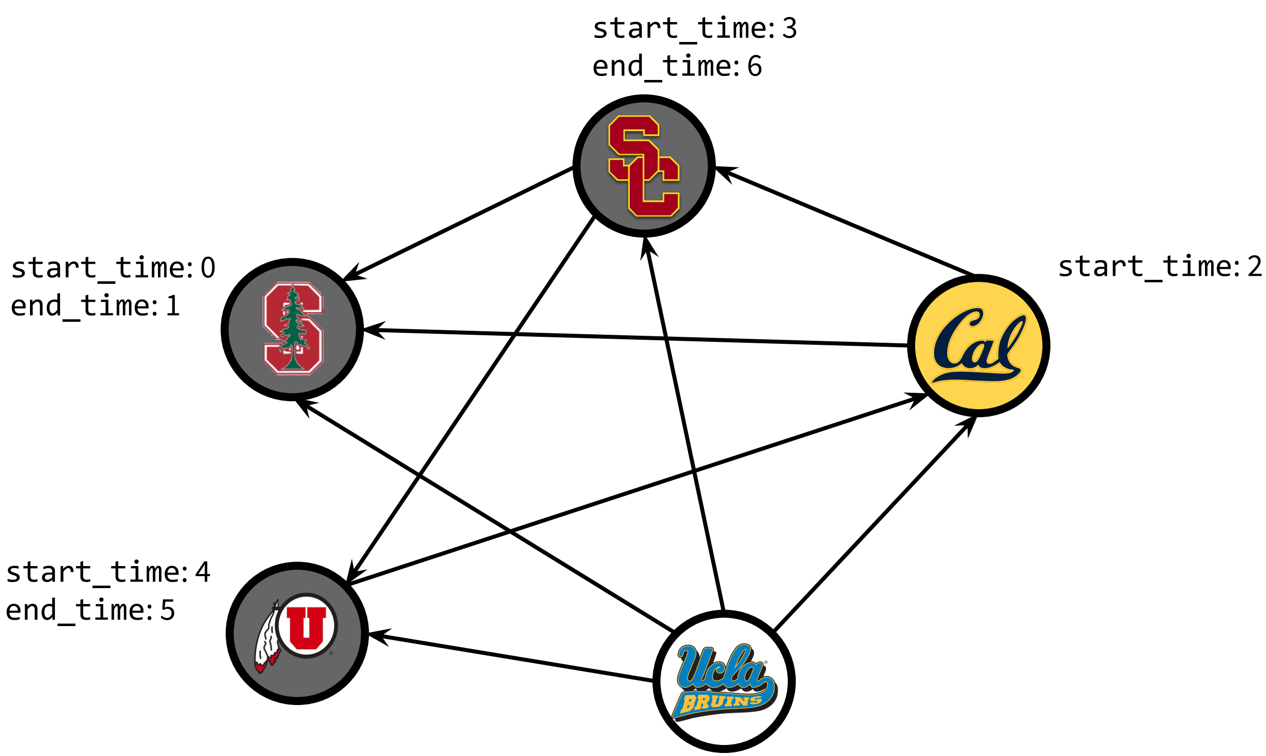
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



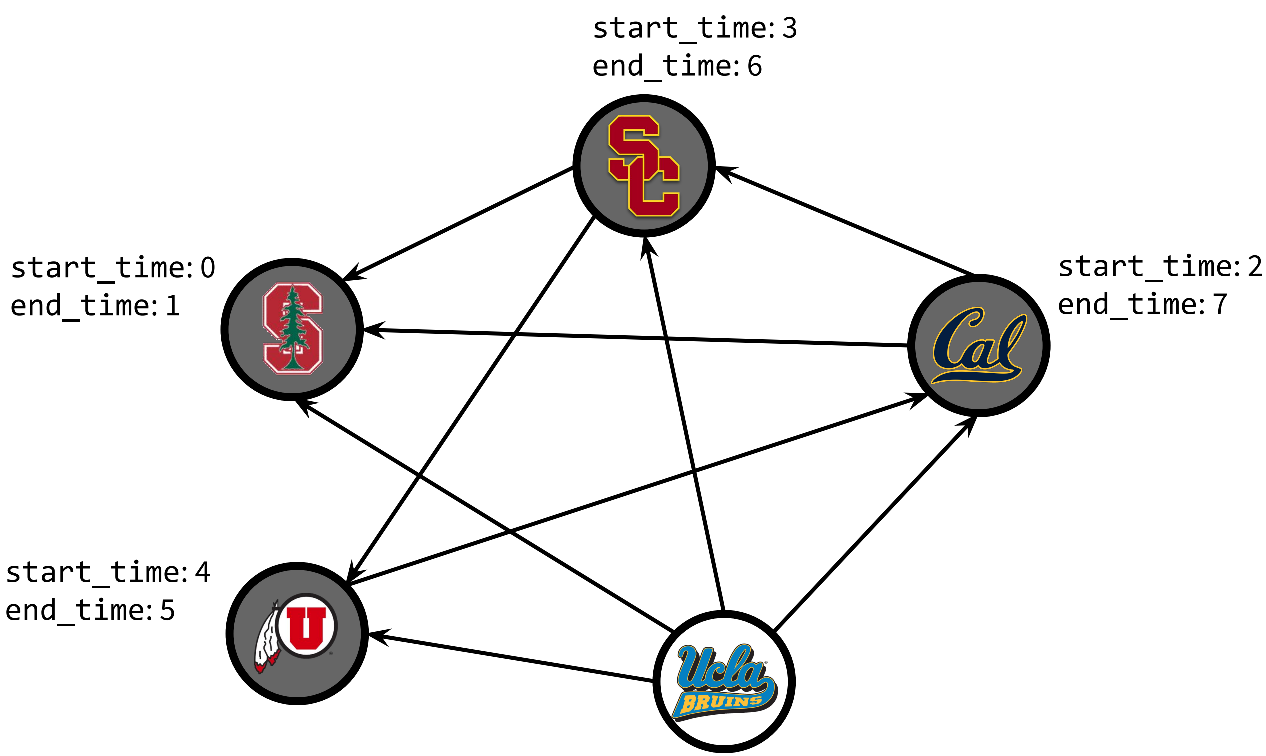
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



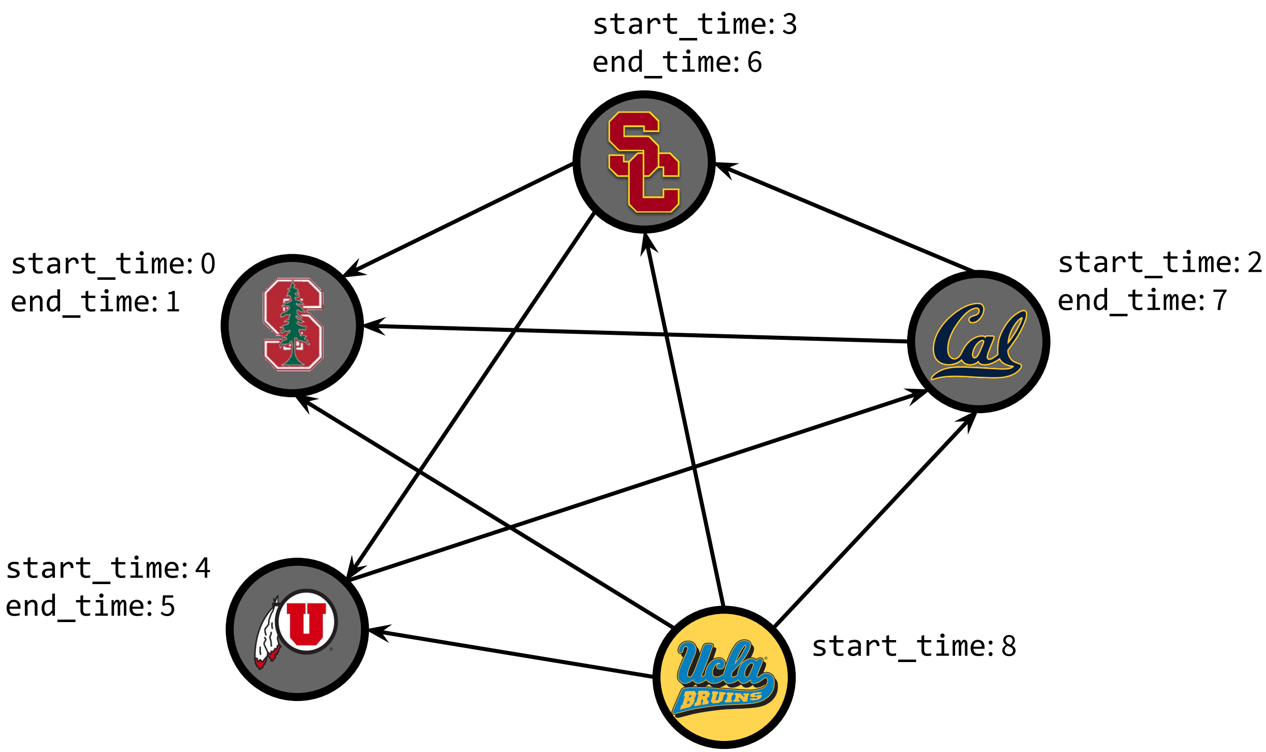
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



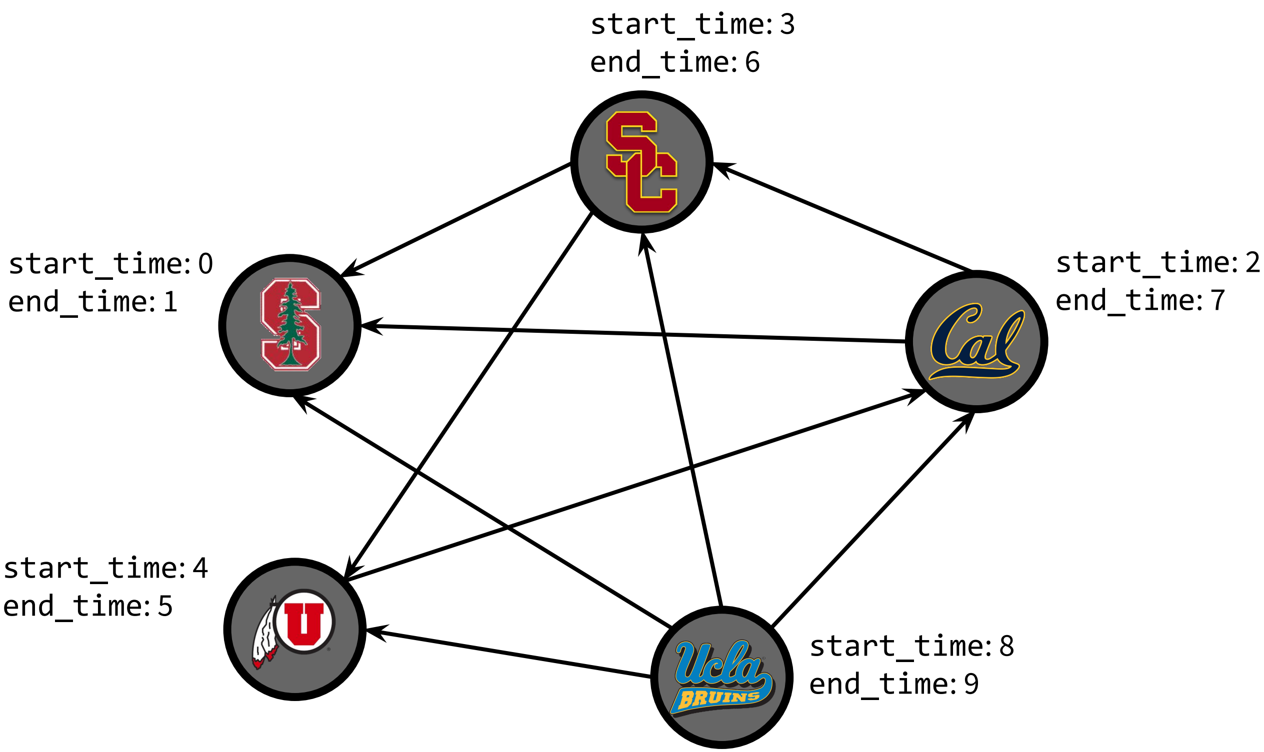
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



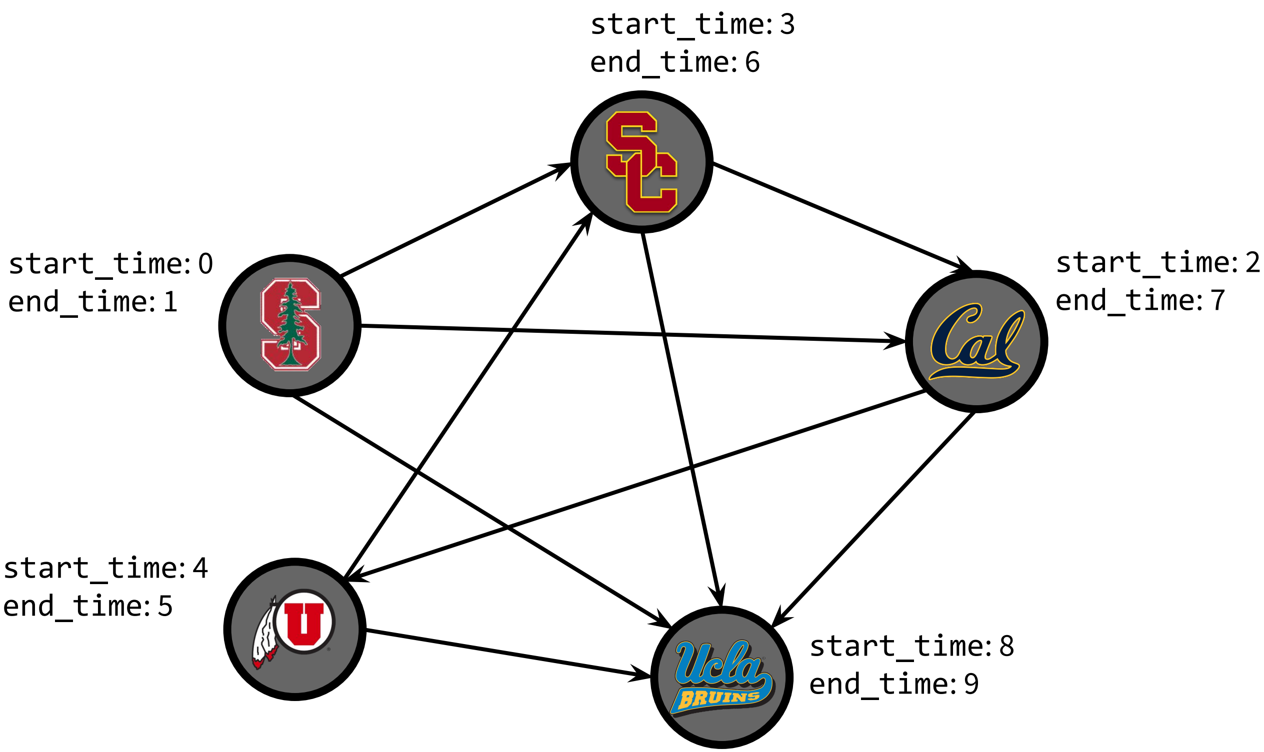
# Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



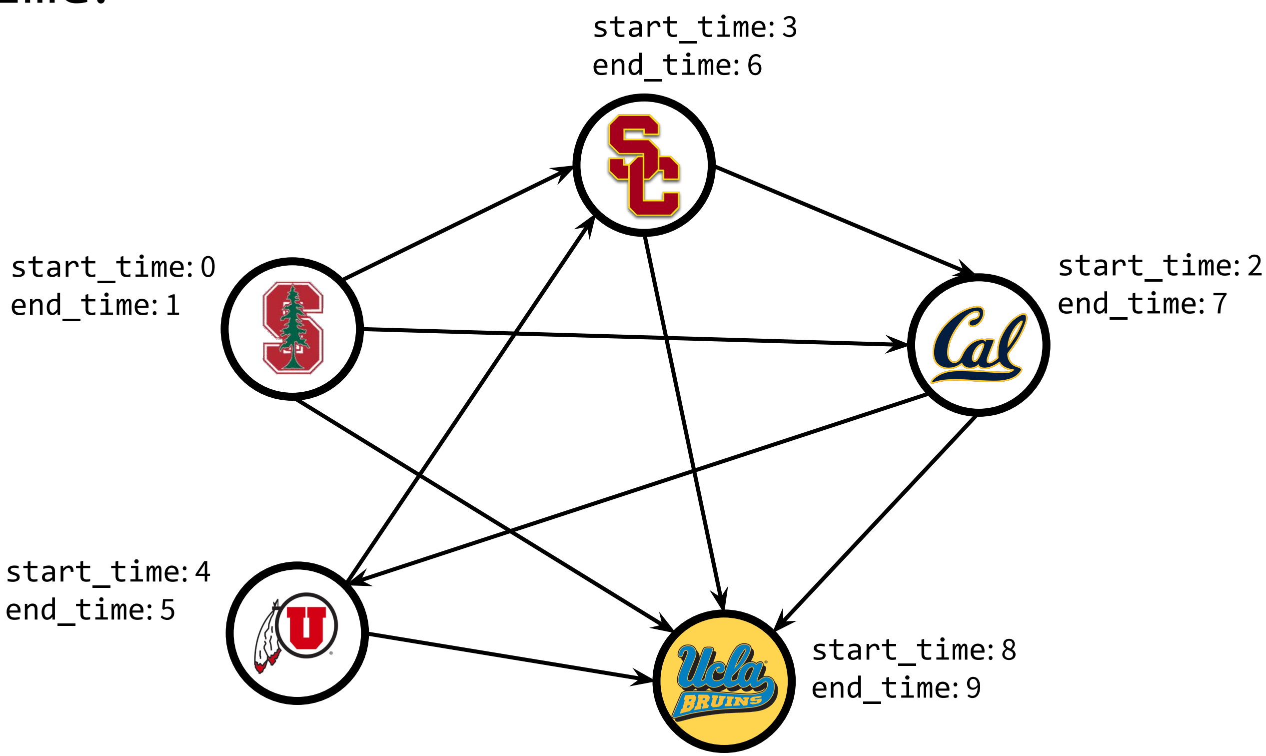
# Kosaraju's Algorithm

2. Reverse all of the edges.



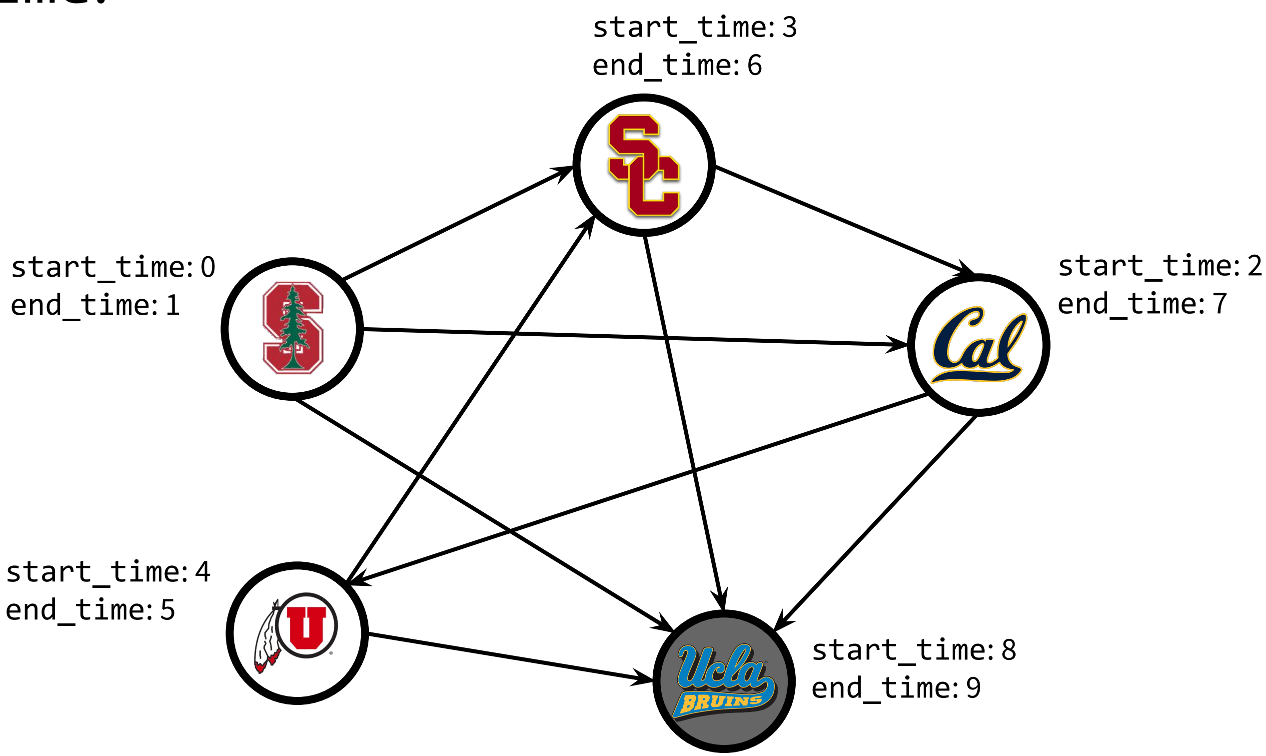
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



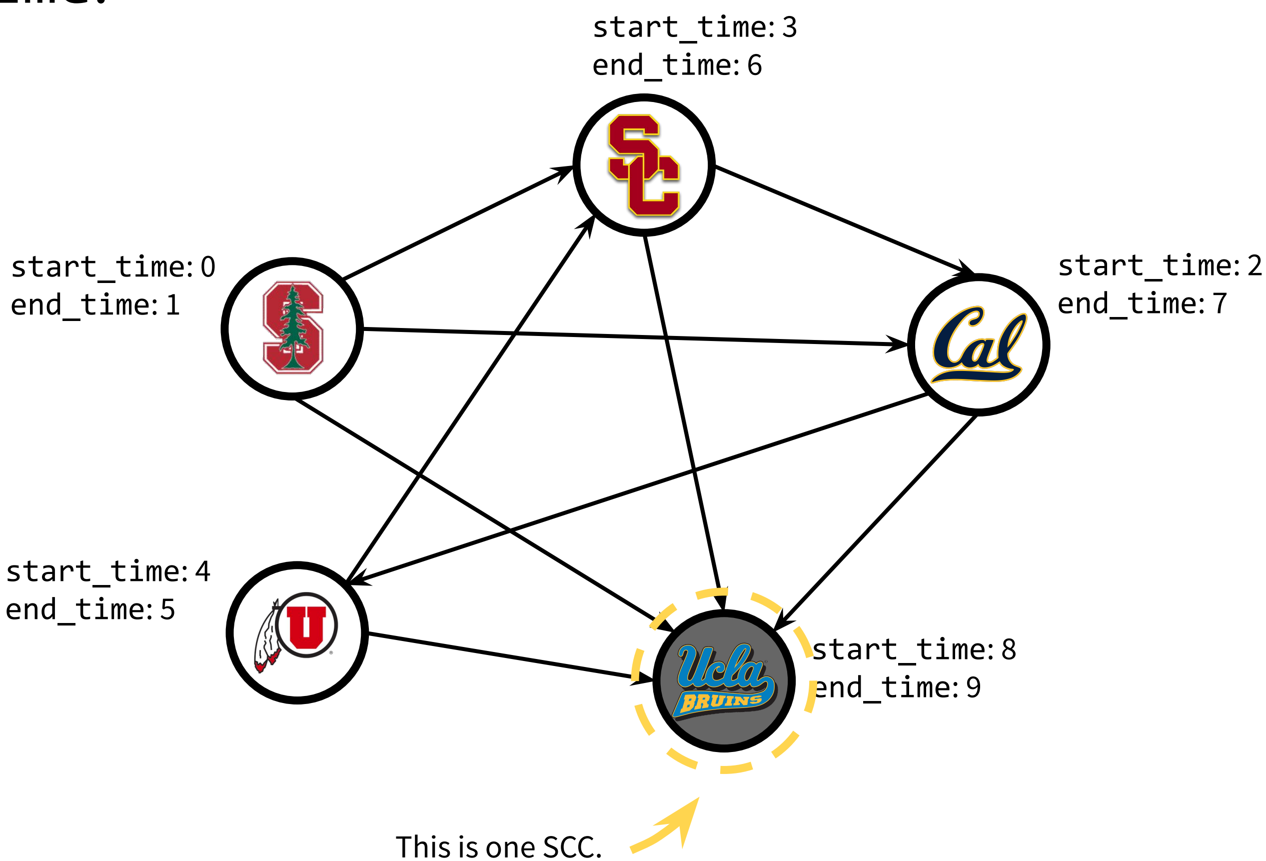
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



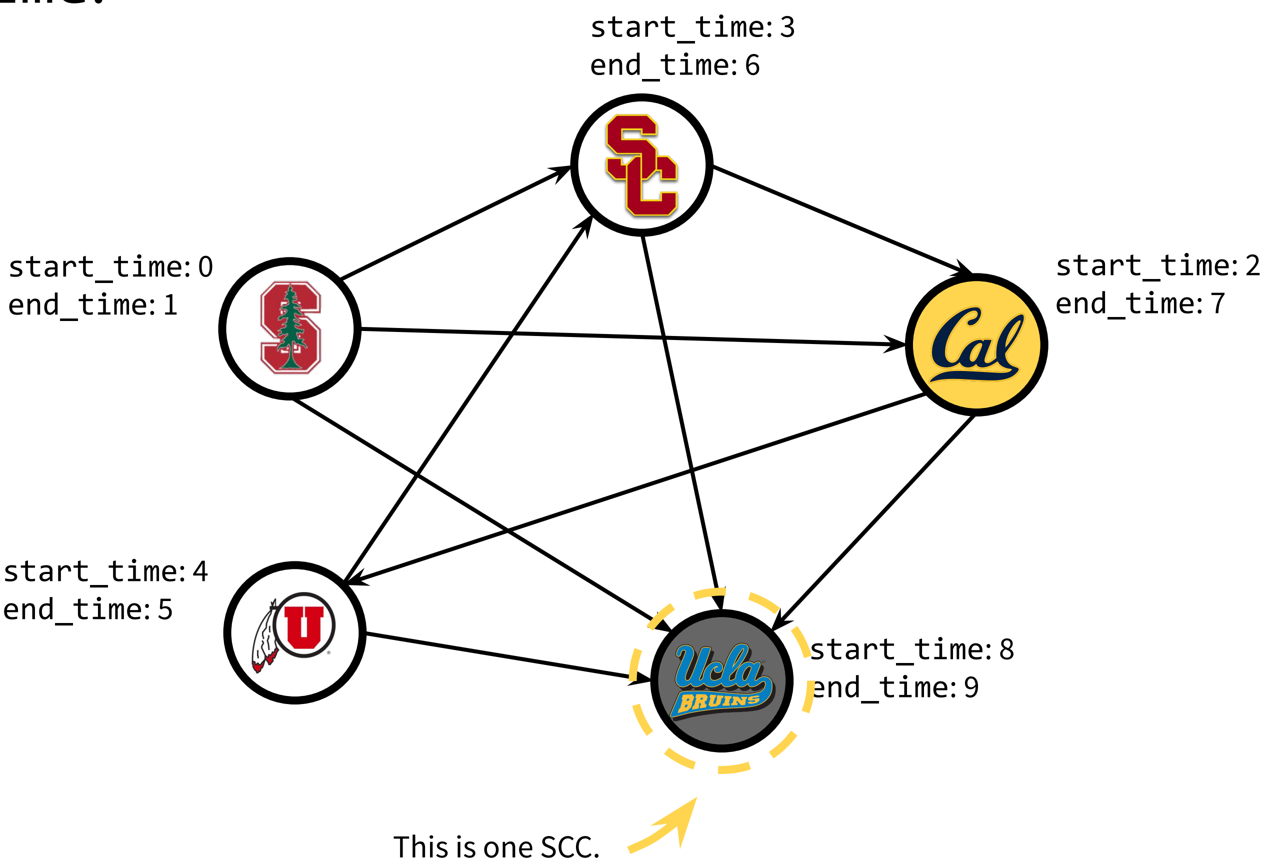
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



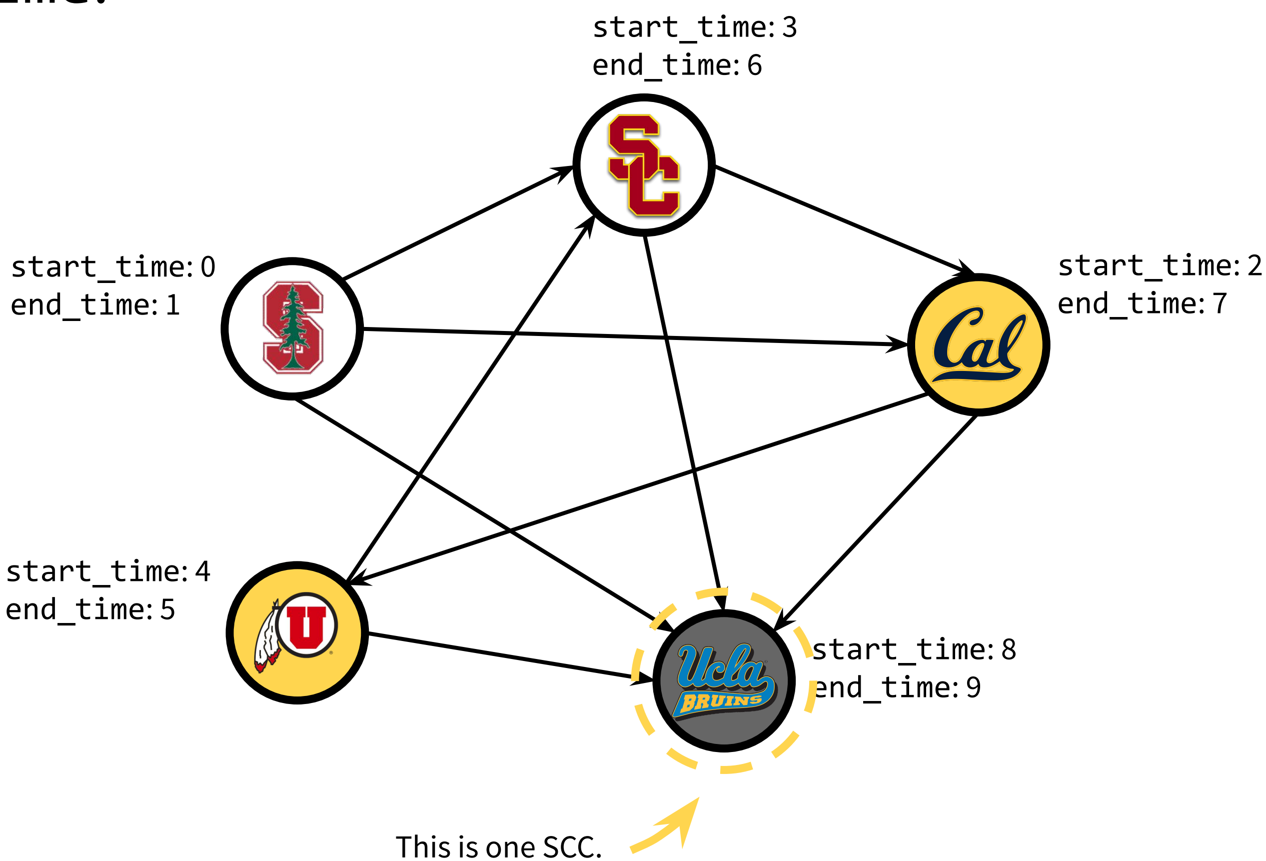
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



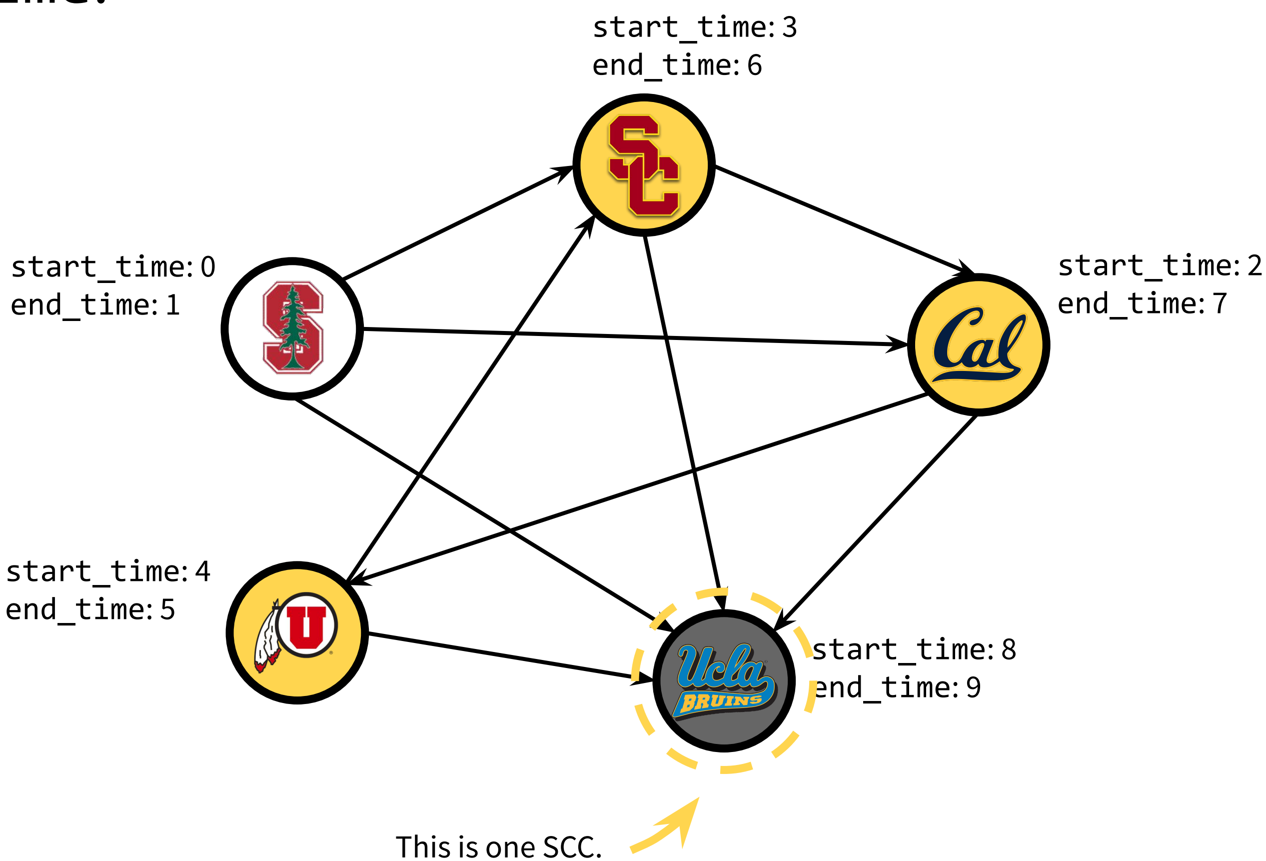
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



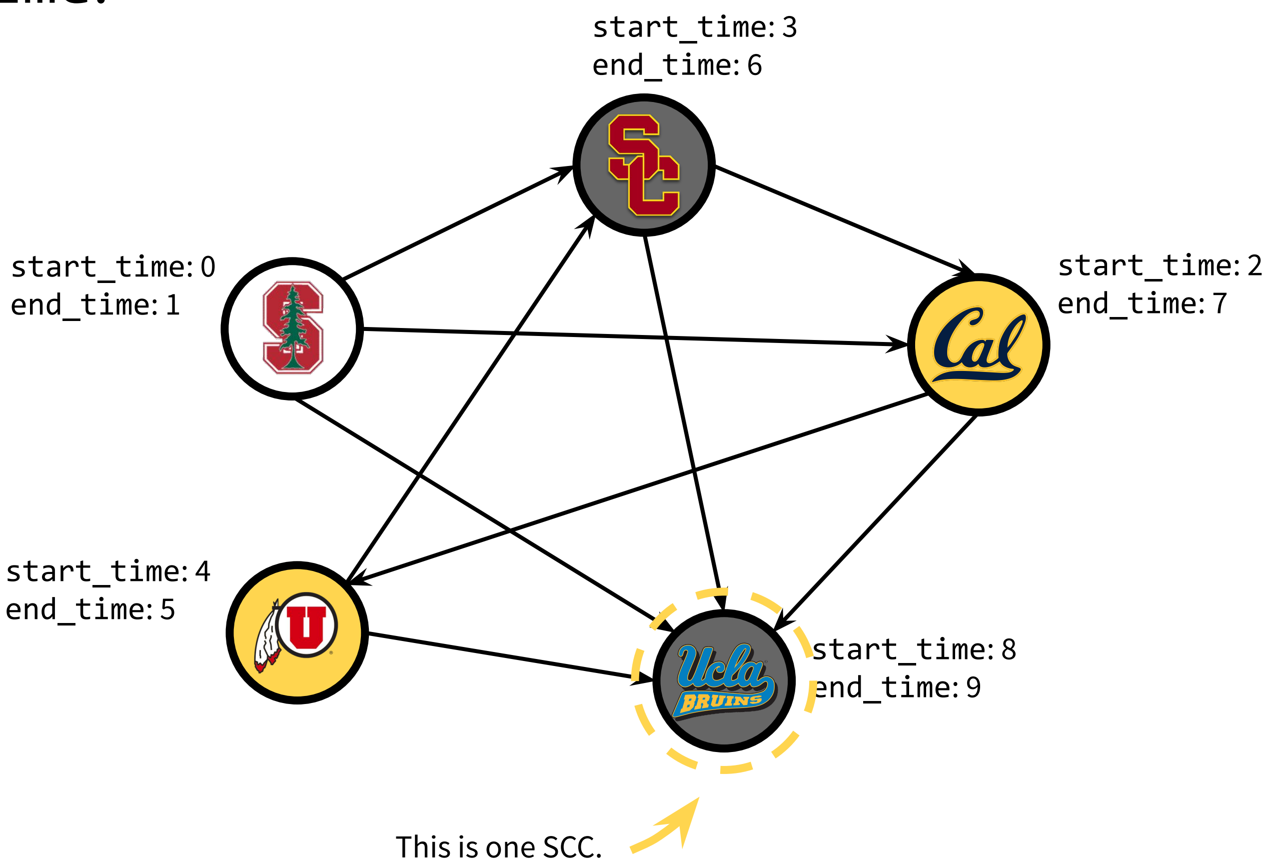
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



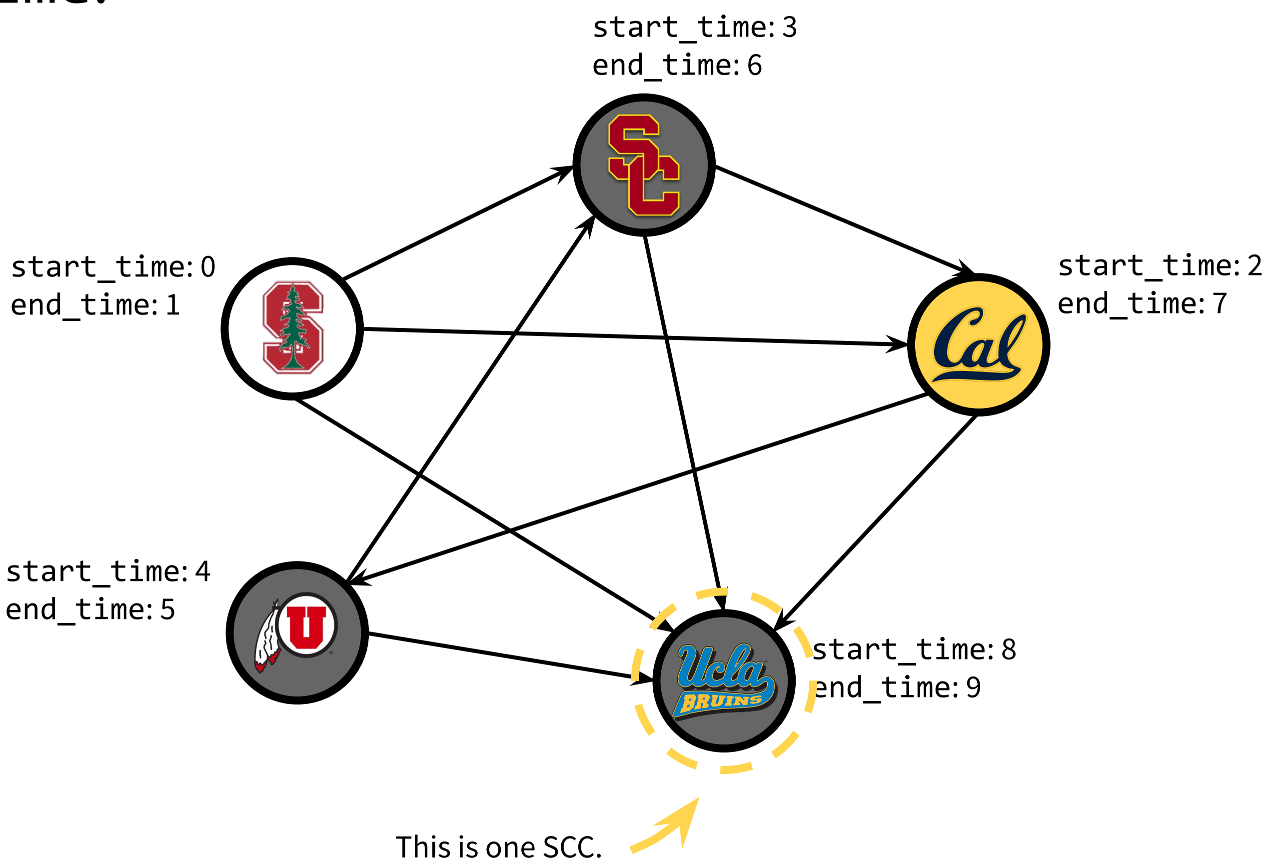
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



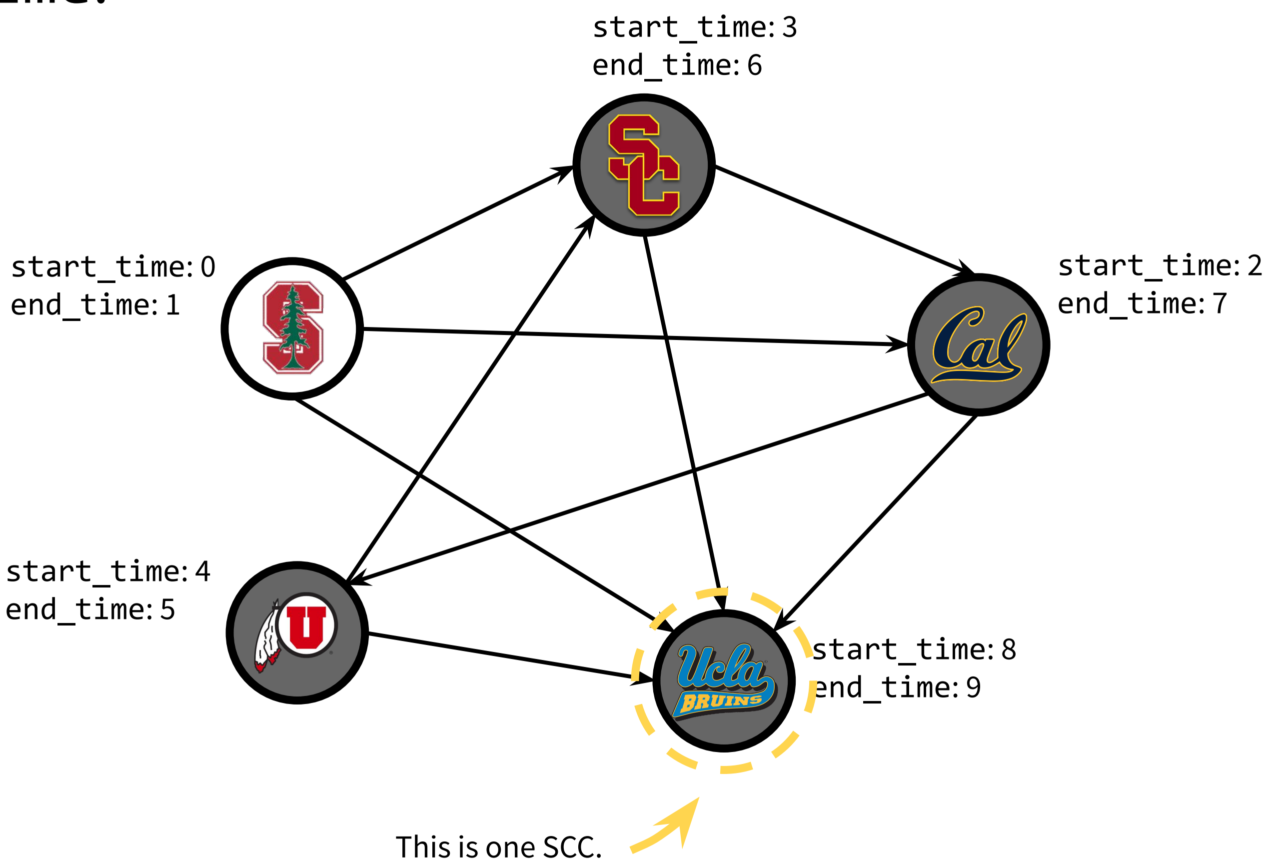
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



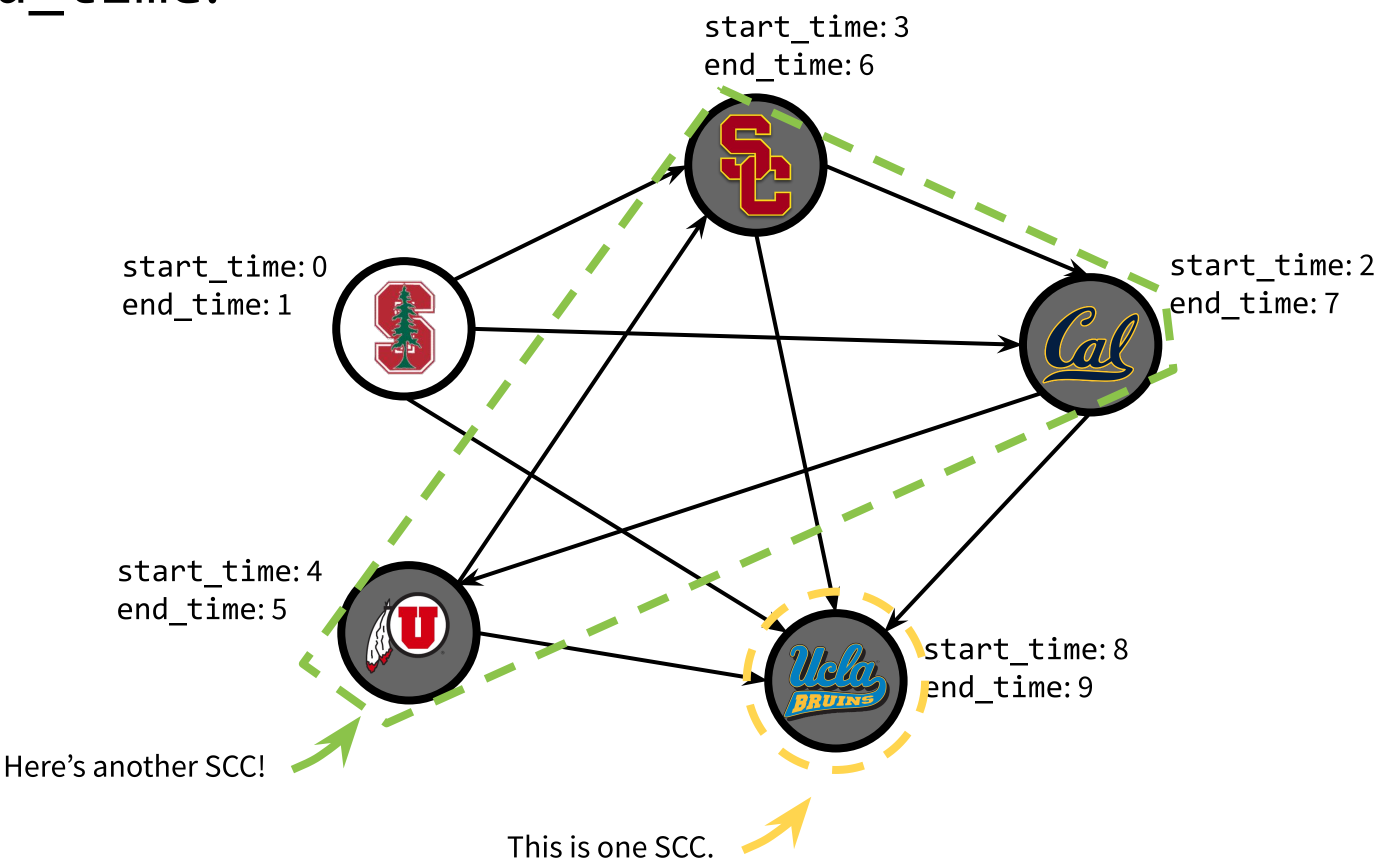
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



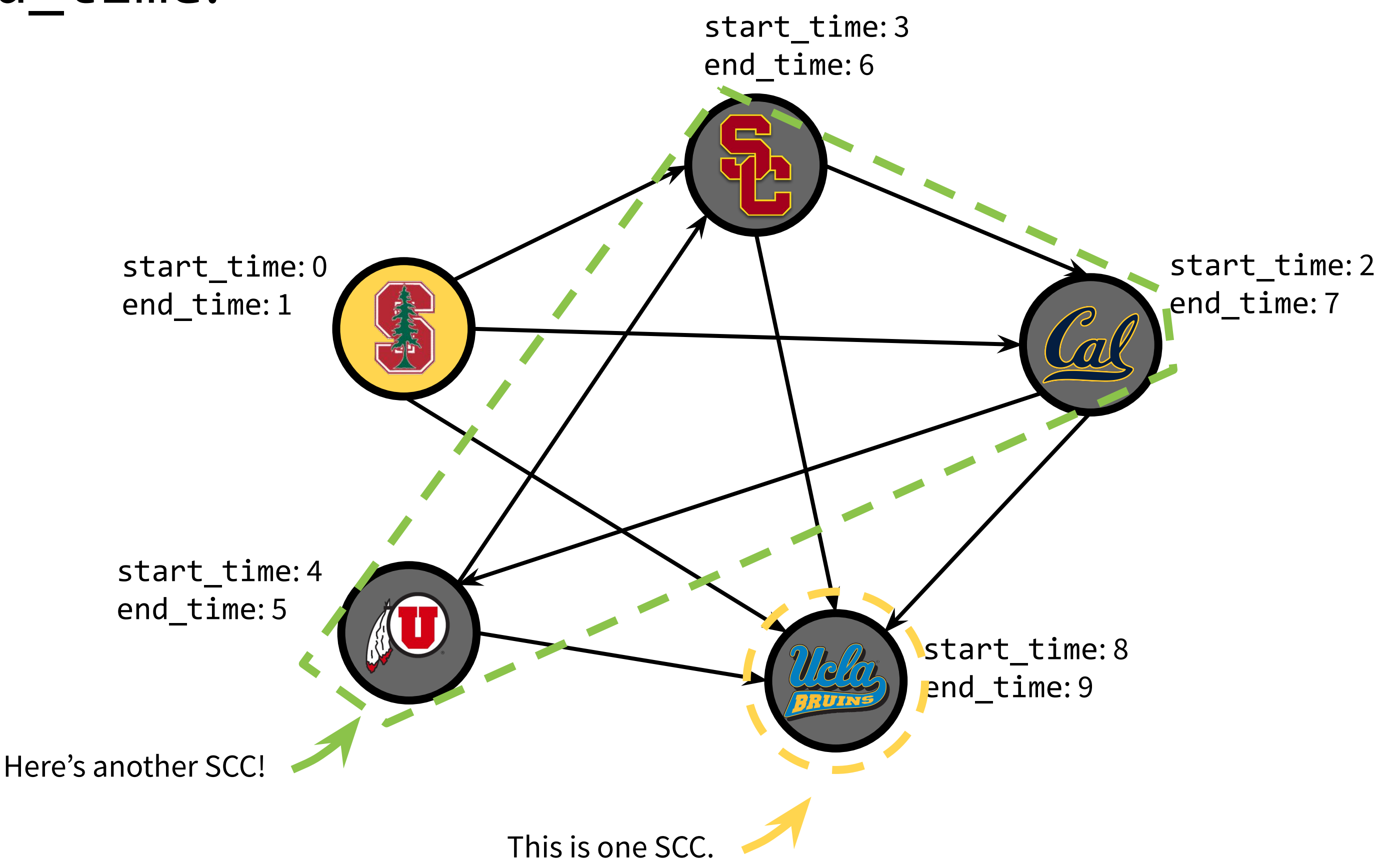
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



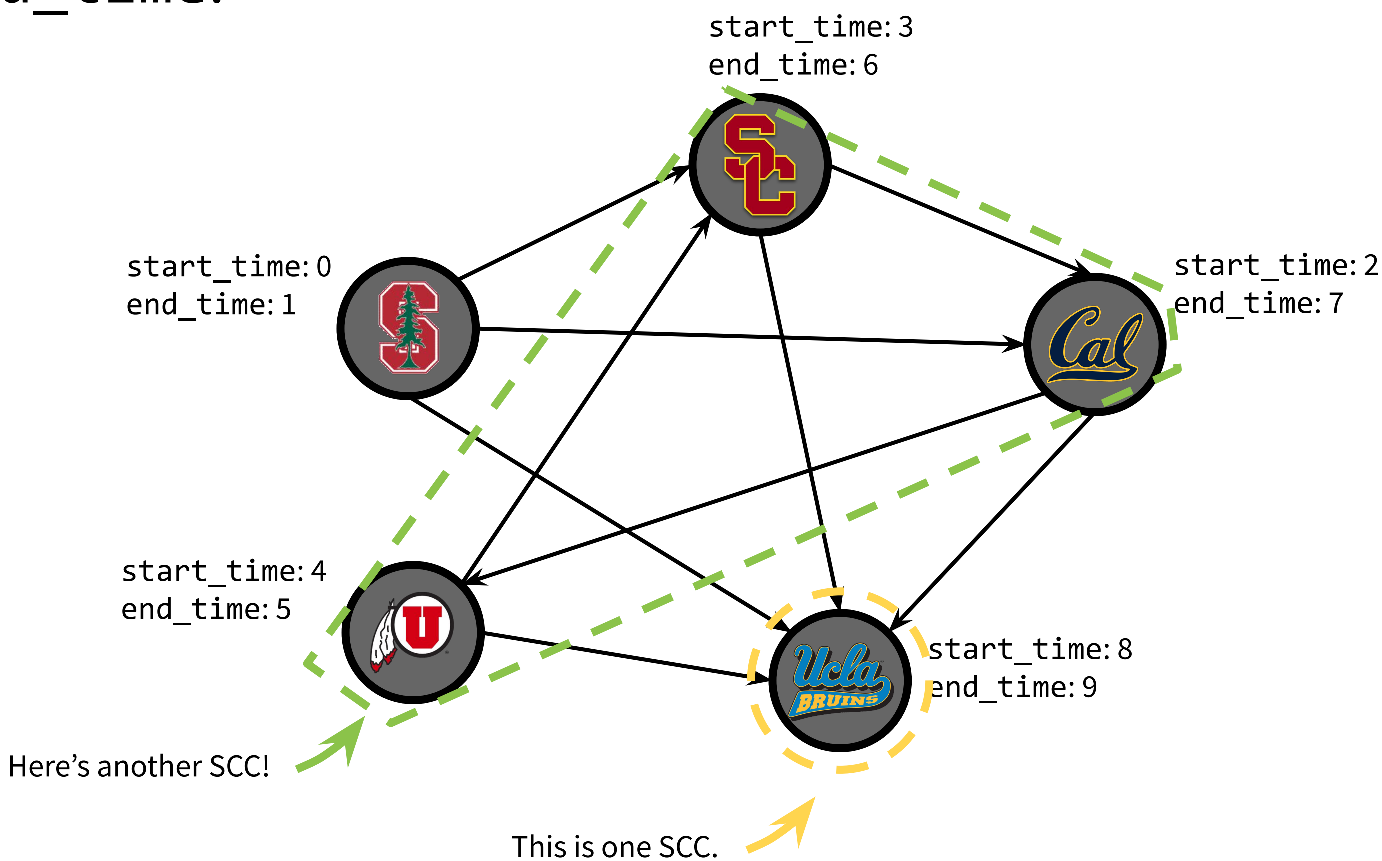
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



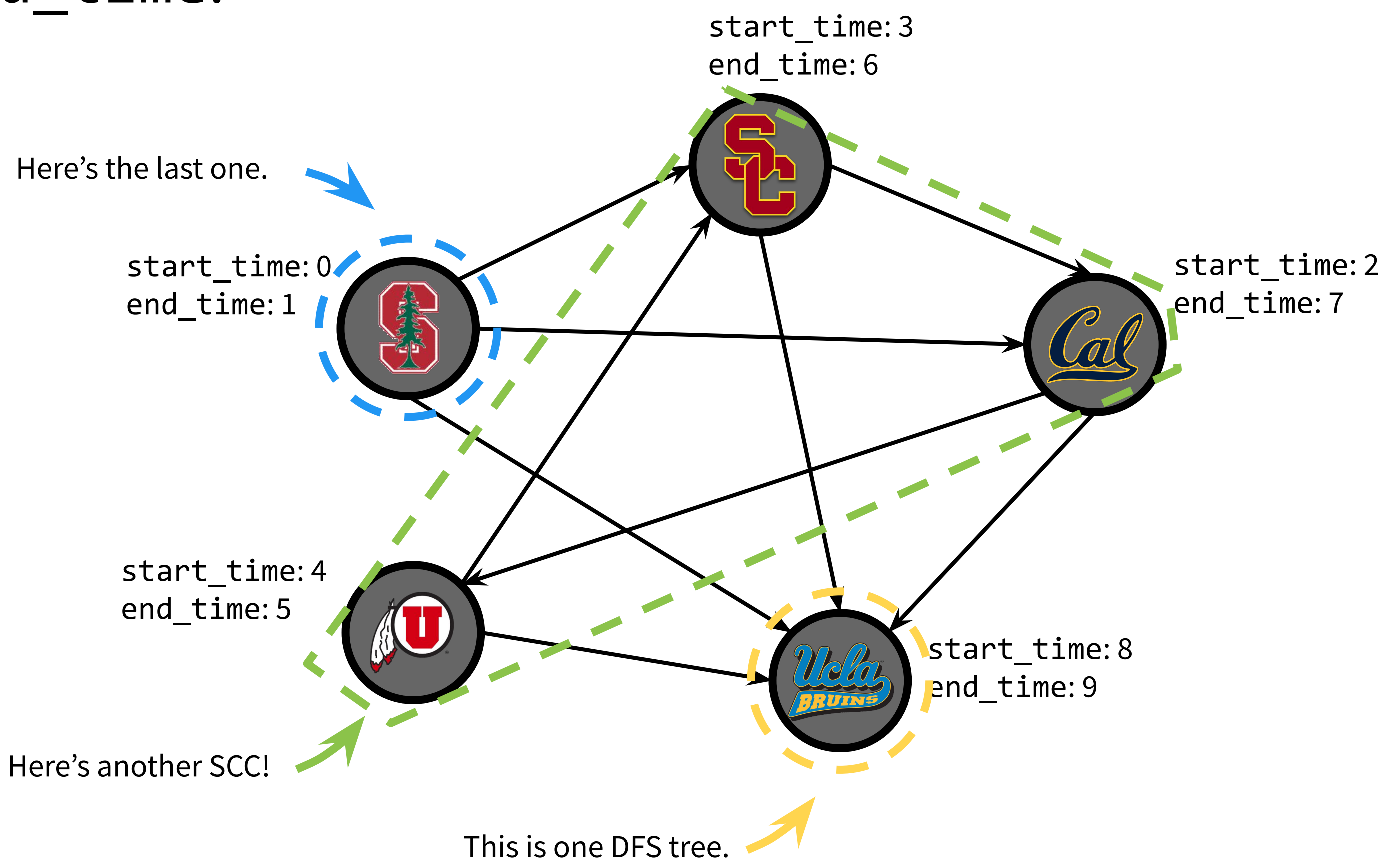
# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.



# Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.

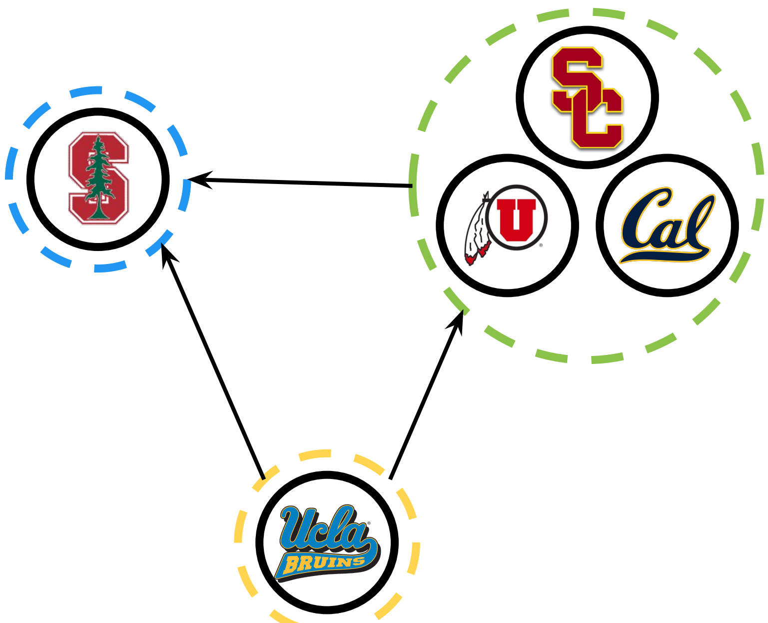


# Kosaraju's Algorithm

Whoa. How did that work?

**Lemma 1:** The SCC metagraph is a DAG.

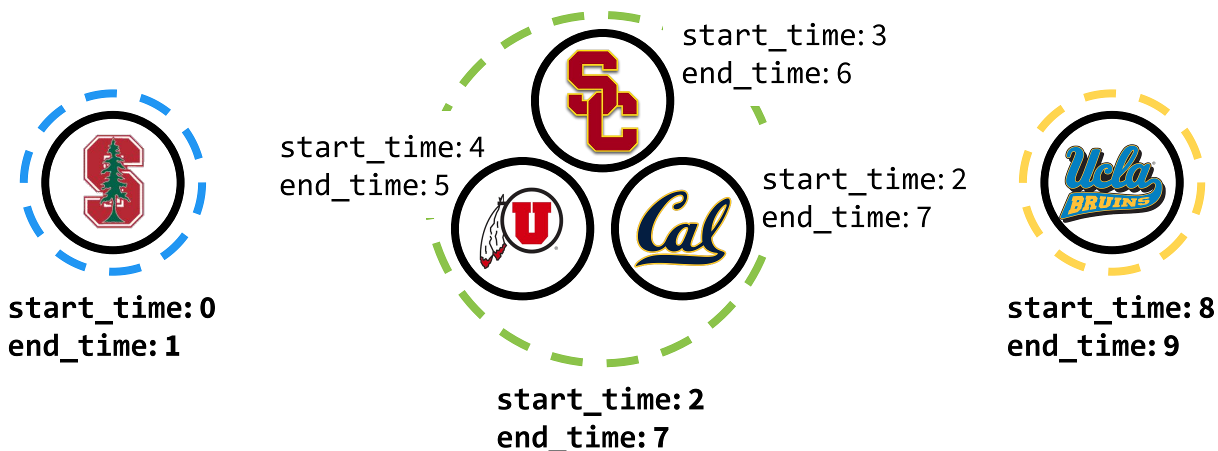
**Intuition:** If not, then two SCCs would collapse into one.



# Kosaraju's Algorithm

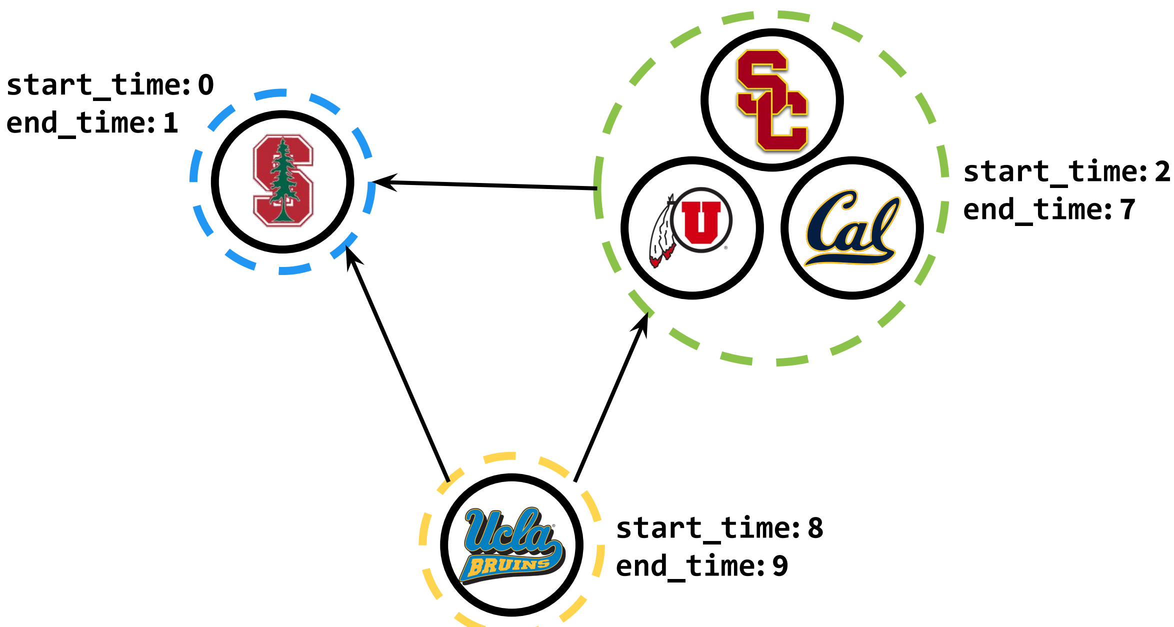
Let the **end time** of a SCC be the largest end time of any element of that SCC.

Let the **starting time** of a SCC be the smallest starting time of any element of that SCC.



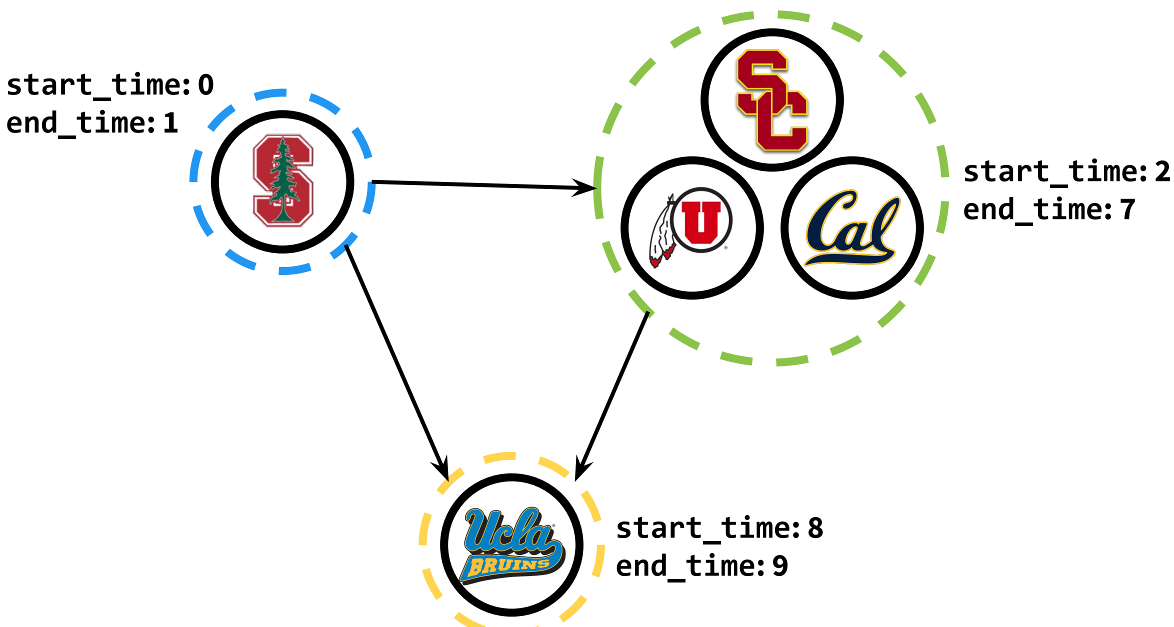
# Kosaraju's Algorithm

The main idea leverages the fact that vertex in the SCC metagraph with the largest `end_time` has no incoming edges.



# Kosaraju's Algorithm

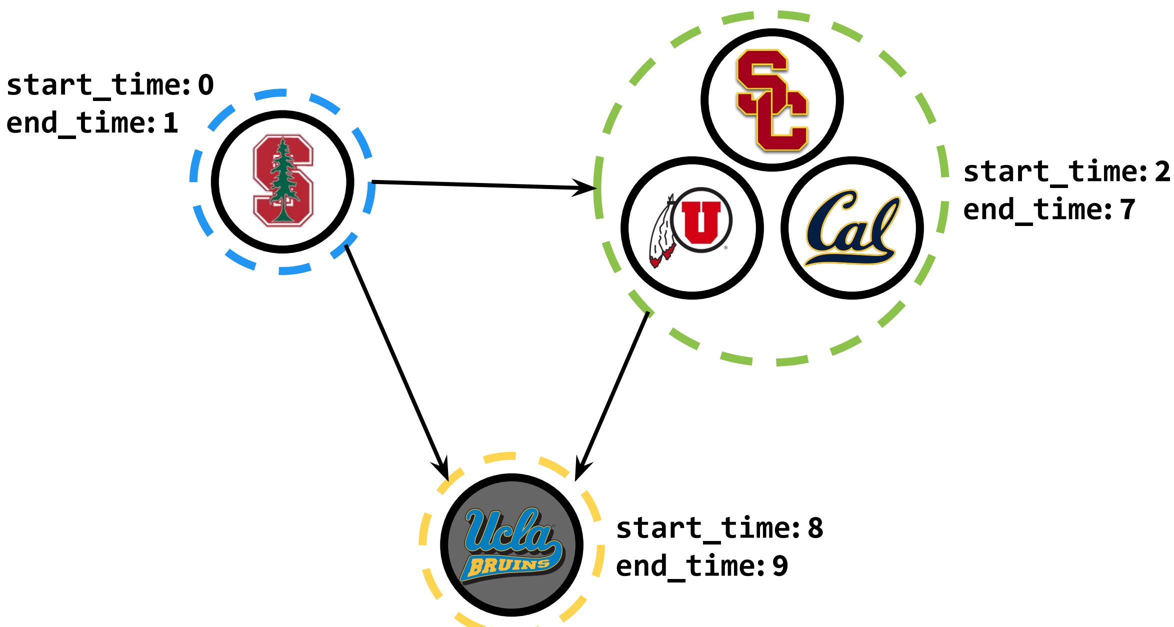
The main idea leverages the fact that vertex in the SCC metagraph with the largest `end_time` has no incoming edges. After reversing the edges, it has no outgoing edges.



# Kosaraju's Algorithm

The main idea leverages the fact that vertex in the SCC metagraph with the largest `end_time` has no incoming edges. After reversing the edges, it has no outgoing edges.

Running `dfs` on that vertex finds exactly that component.

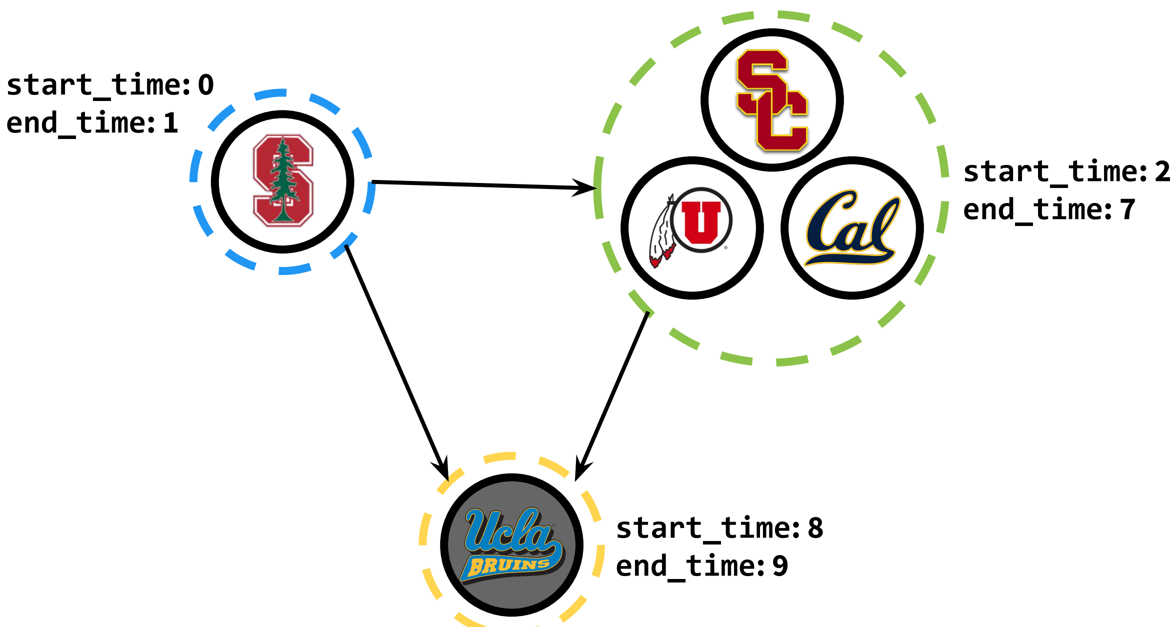


# Kosaraju's Algorithm

The main idea leverages the fact that vertex in the SCC metagraph with the largest `end_time` has no incoming edges. After reversing the edges, it has no outgoing edges.

Running `dfs` on that vertex finds exactly that component.

Same argument for the rest.

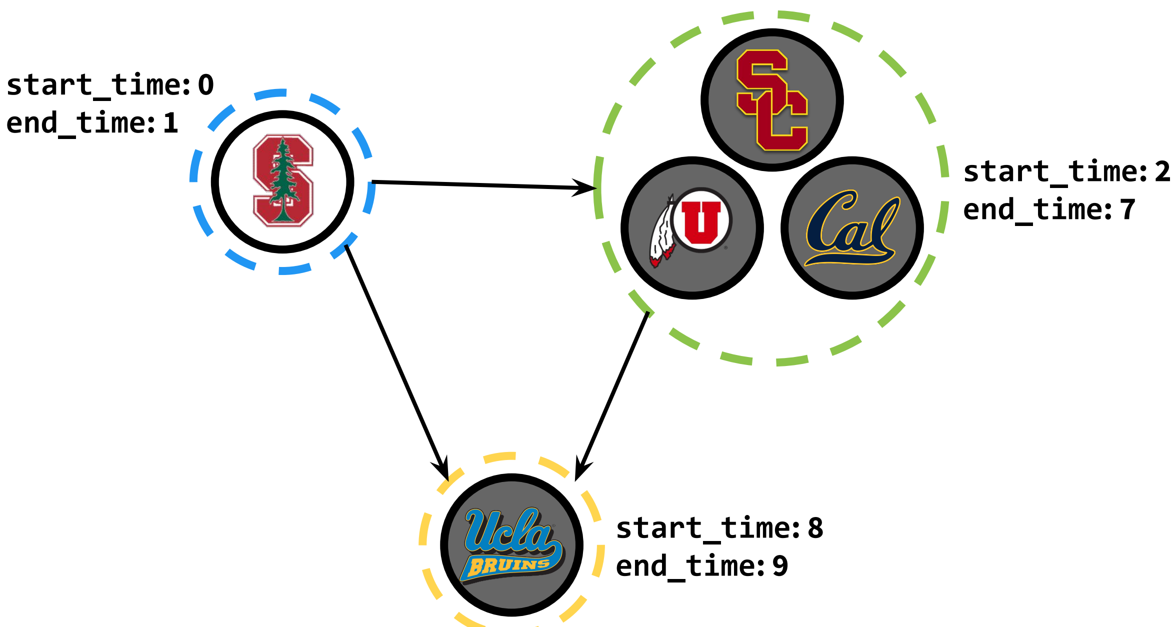


# Kosaraju's Algorithm

The main idea leverages the fact that vertex in the SCC metagraph with the largest `end_time` has no incoming edges. After reversing the edges, it has no outgoing edges.

Running `dfs` on that vertex finds exactly that component.

Same argument for the rest.

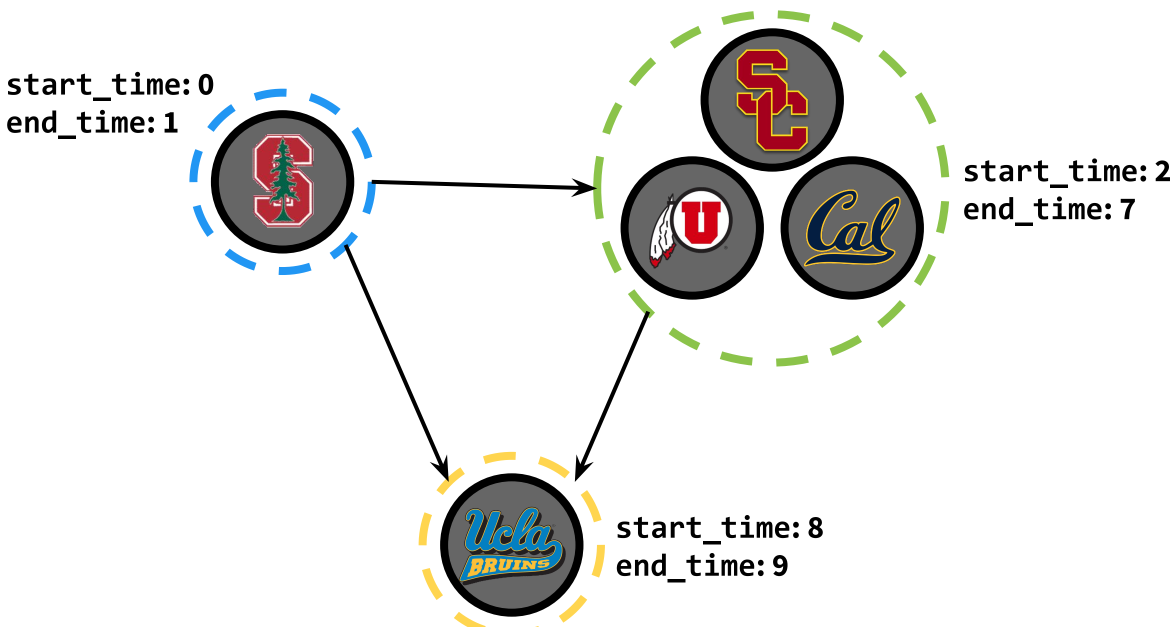


# Kosaraju's Algorithm

The main idea leverages the fact that vertex in the SCC metagraph with the largest `end_time` has no incoming edges. After reversing the edges, it has no outgoing edges.

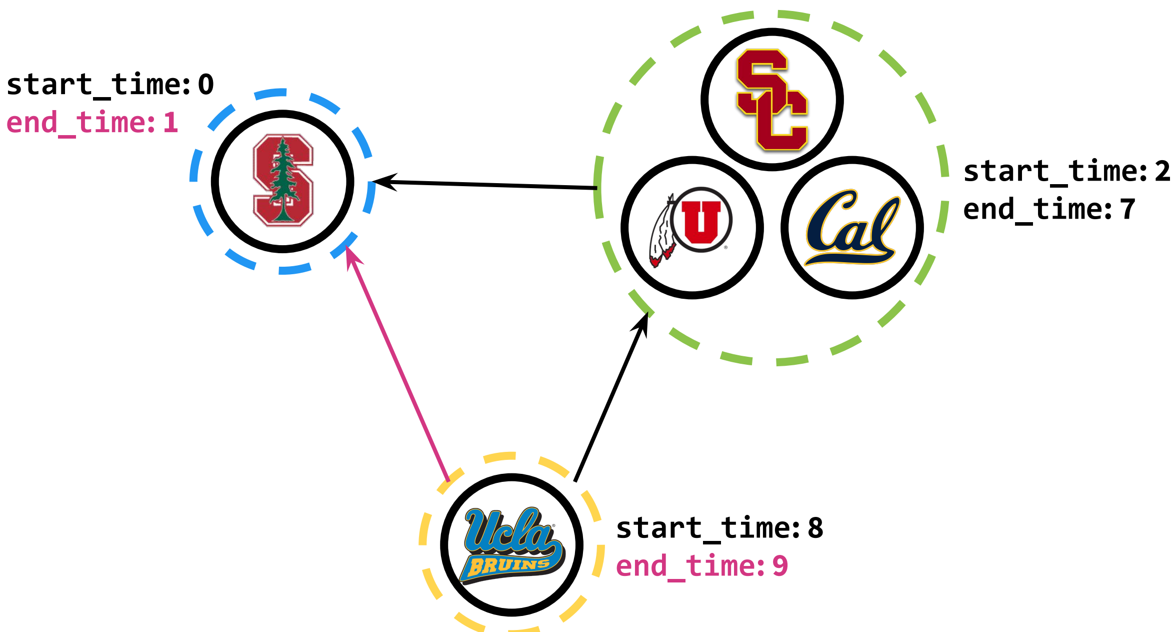
Running `dfs` on that vertex finds exactly that component.

Same argument for the rest.



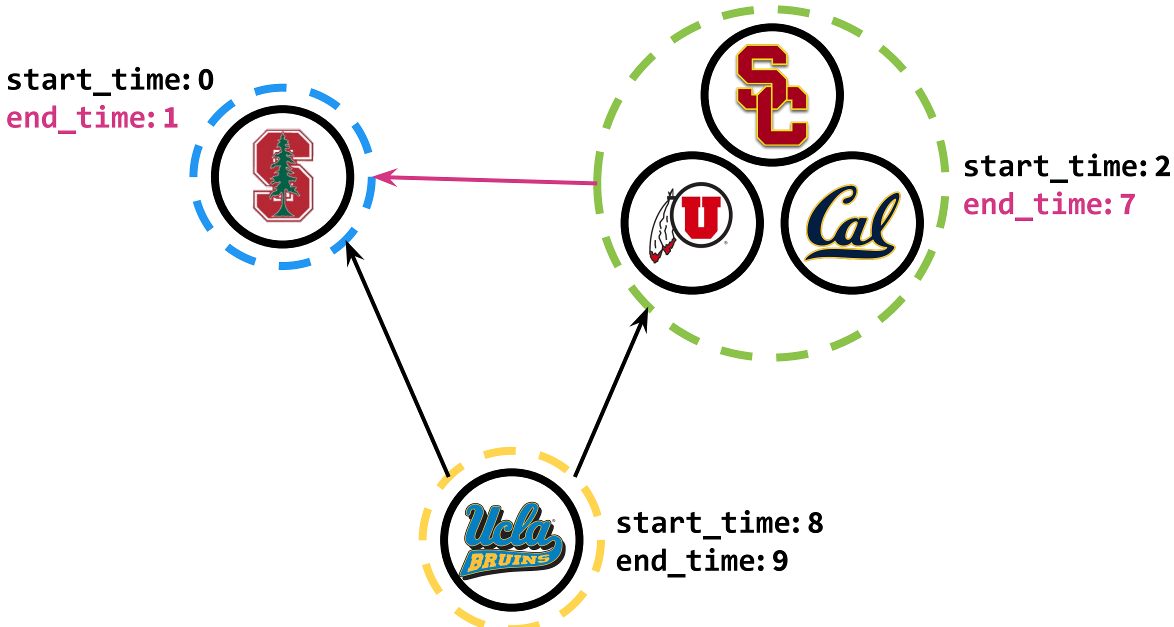
# Kosaraju's Algorithm

**Claim:** For each edge (u, v) in the SCC metagraph where $u \in C_1$ and $v \in C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.



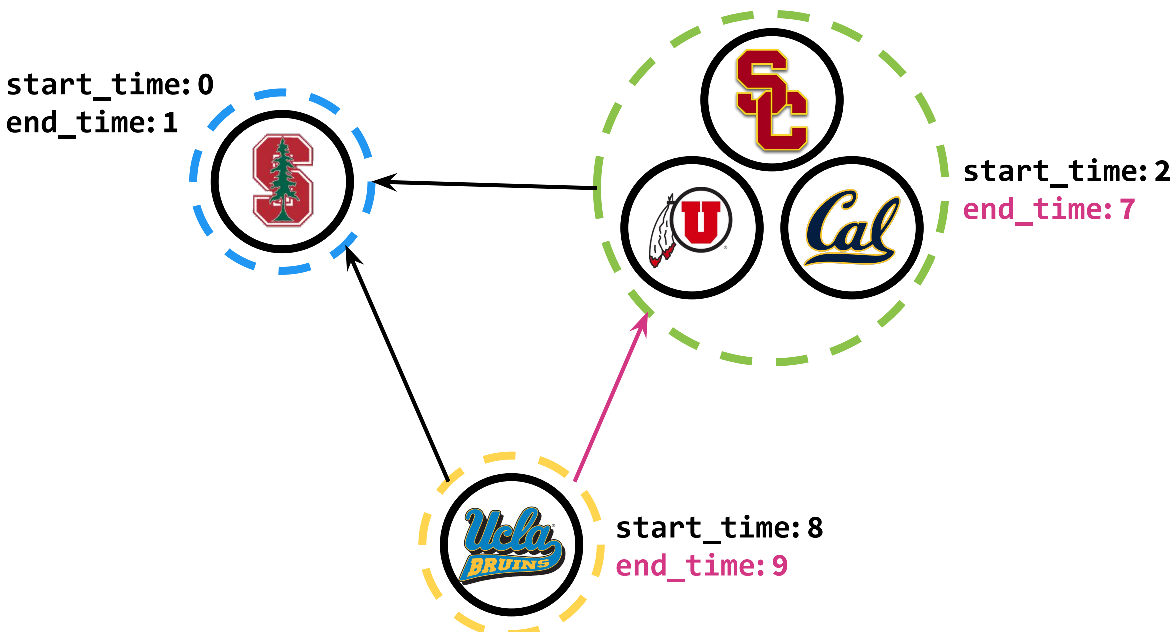
# Kosaraju's Algorithm

**Claim:** For each edge (u, v) in the SCC metagraph where $u \in C_1$ and $v \in C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.



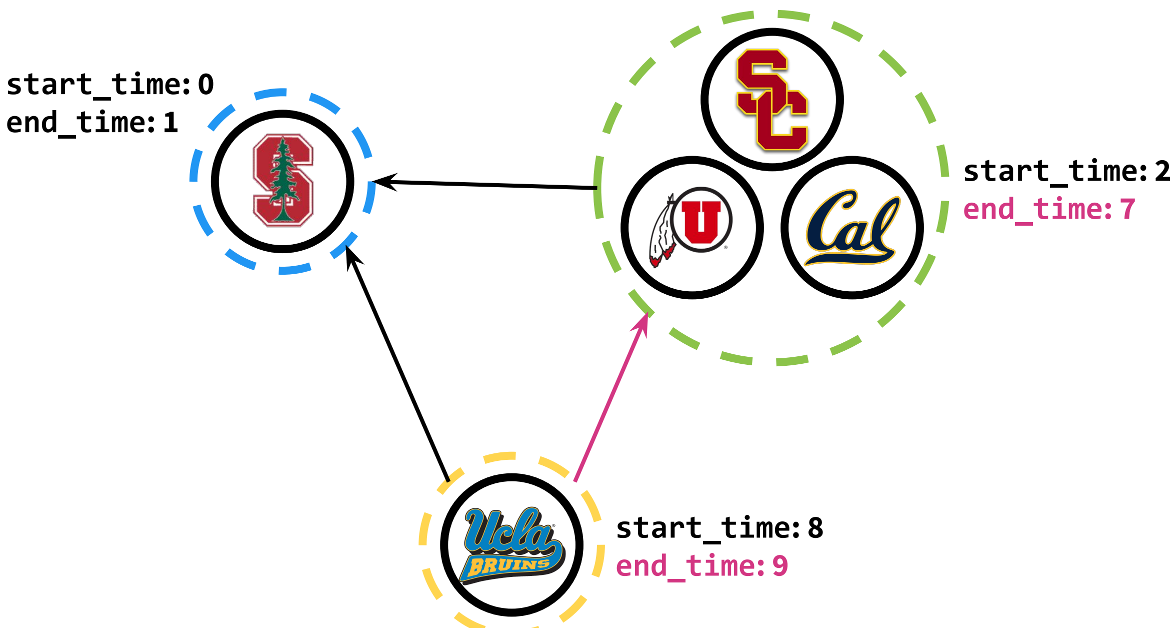
# Kosaraju's Algorithm

**Claim:** For each edge (u, v) in the SCC metagraph where $u \in C_1$ and $v \in C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.



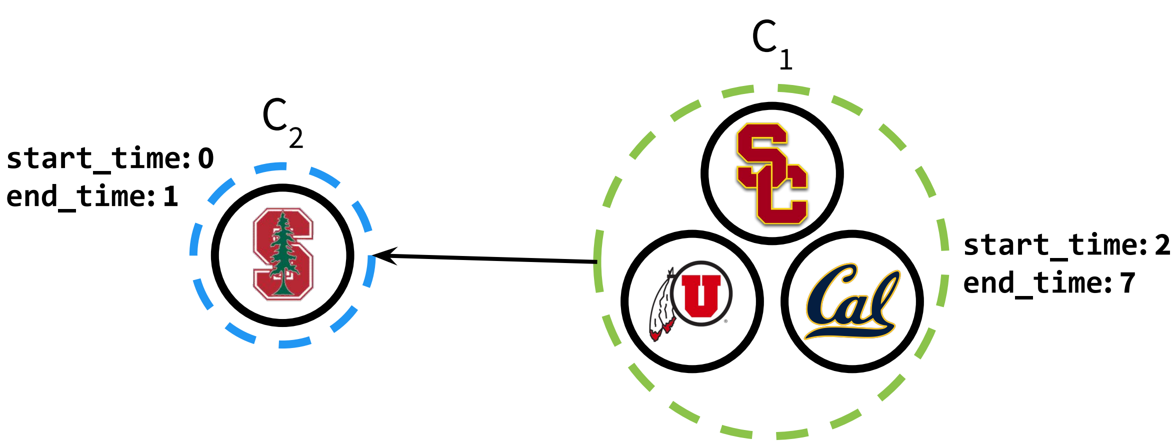
# Kosaraju's Algorithm

**Claim:** For each edge (u, v) in the SCC metagraph where u ∈ $C_1$ and v ∈ $C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.



# Kosaraju's Algorithm

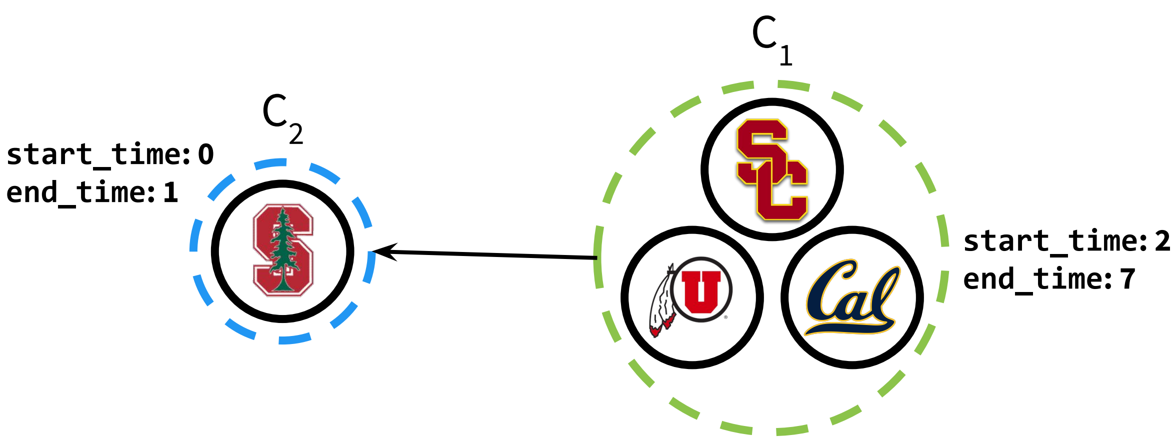
**Claim:** For each edge (u, v) in the SCC metagraph where u ∊ $C_1$ and v ∊ $C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.



# Kosaraju's Algorithm

**Claim:** For each edge (u, v) in the SCC metagraph where u ∈ $C_1$ and v ∈ $C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.

**Intuition:** In order for the `end_time` of $C_1$ to be smaller than the `end_time` of $C_2$, all vertices in $C_1$ must have `end_times` smaller than at least one `end_time` of $C_2$.

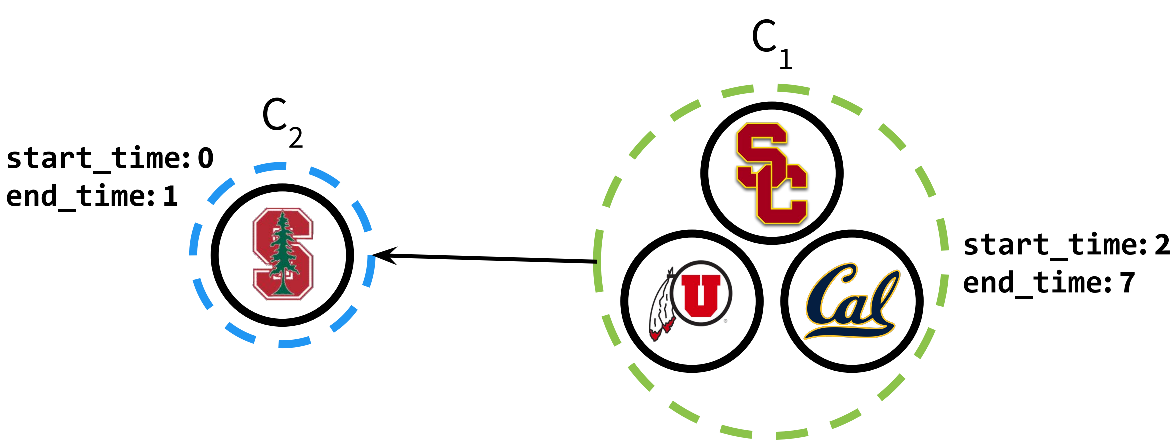


# Kosaraju's Algorithm

**Claim:** For each edge (u, v) in the SCC metagraph where $u \in C_1$ and $v \in C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.

**Intuition:** In order for the `end_time` of $C_1$ to be smaller than the `end_time` of $C_2$, all vertices in $C_1$ must have `end_times` smaller than at least one `end_time` of $C_2$.

For this to occur, the first `dfs` must have marked all vertices in $C_1$ as "done" before at least one vertex in $C_2$.

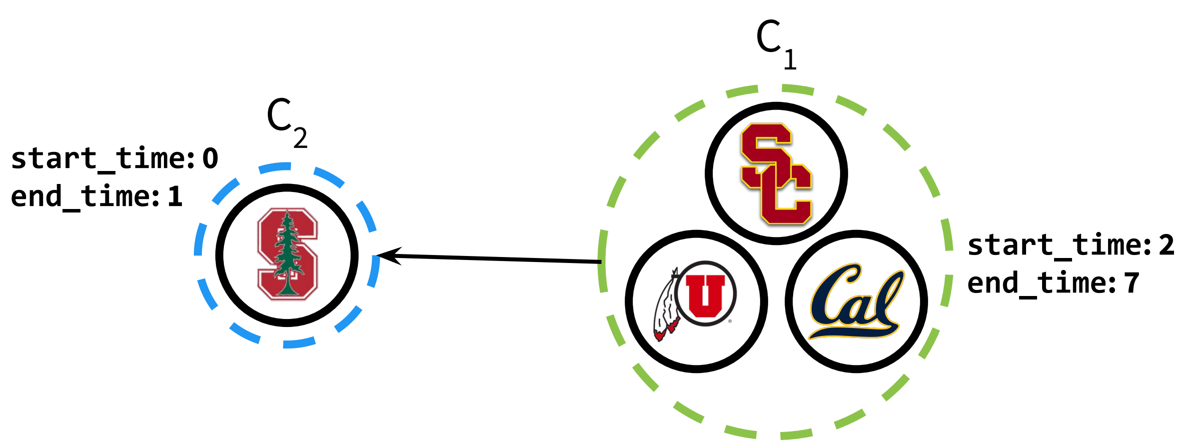


# Kosaraju's Algorithm

**Claim:** For each edge (u, v) in the SCC metagraph where $u \in C_1$ and $v \in C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.

**Intuition:** In order for the `end_time` of $C_1$ to be smaller than the `end_time` of $C_2$, all vertices in $C_1$ must have `end_times` smaller than at least one `end_time` of $C_2$.

For this to occur, the first `dfs` must have marked all vertices in $C_1$ as "done" before at least one vertex in $C_2$. But this is impossible since the first `dfs` must have explored edge (u, v) before marking all vertices in $C_1$ as "done."



# Kosaraju's Algorithm

**Claim:** For each edge (u, v) in the SCC metagraph where u ∈ $C_1$ and v ∈ $C_2$, `end_time` of $C_1$ must be larger than `end_time` of $C_2$.

**Intuition:** In order for the `end_time` of $C_1$ to be smaller than the `end_time` of $C_2$, all vertices in $C_1$ must have `end_times` smaller than at least one `end_time` of $C_2$.

For this to occur, the first `dfs` must have marked all vertices in $C_1$ as "done" before at least one vertex in $C_2$. But this is impossible since the first `dfs` must have explored edge (u, v) before marking all vertices in $C_1$ as "done." Since $C_2$ is an SCC, all vertices in it are reachable from v; therefore, all must have an `end_time` smaller than u.