Graph Algorithms II

Summer 2017 • Lecture 07/20

A Few Notes

Homework 3

Due tomorrow at 11:59 p.m. on Gradescope.

Homework 4

Released tomorrow.

Outline for Today

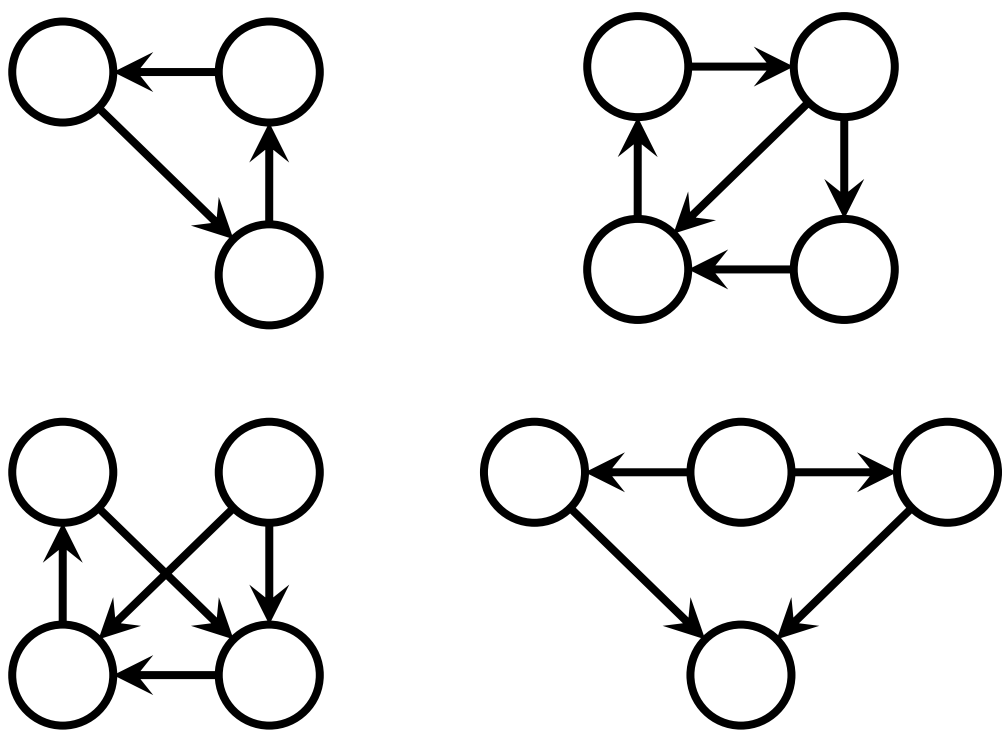
Graph algorithms

- Kosaraju's Algorithm for finding strongly connected components
- Karger's Algorithm for finding global minimum cuts

Kosaraju’s Algorithm

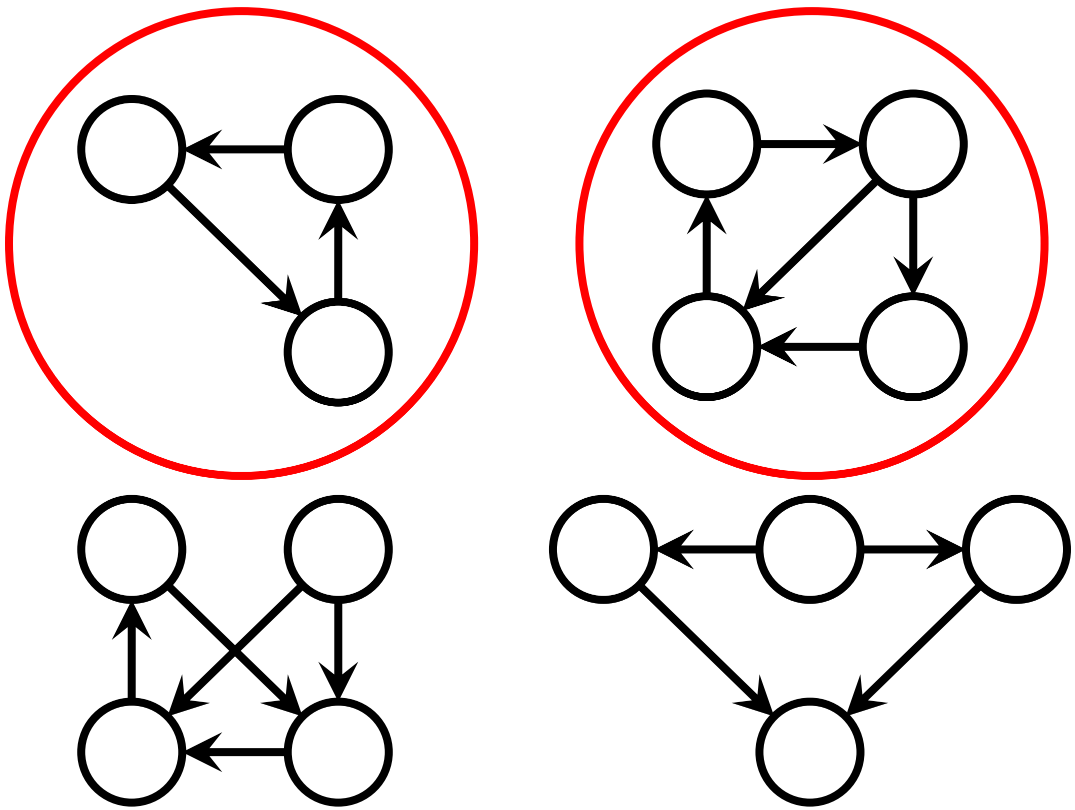
Strongly Connected Components

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.



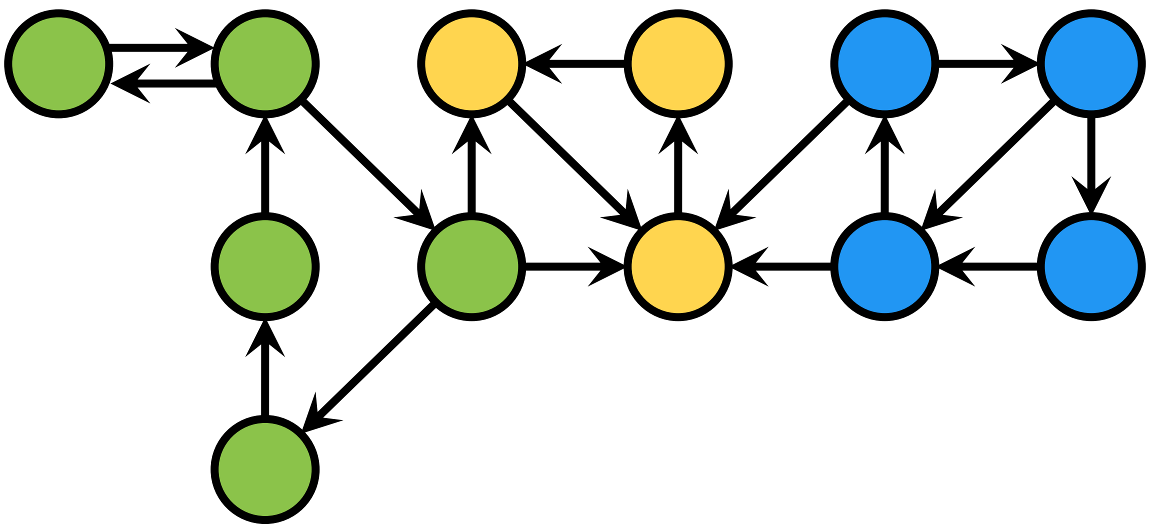
Strongly Connected Components

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).



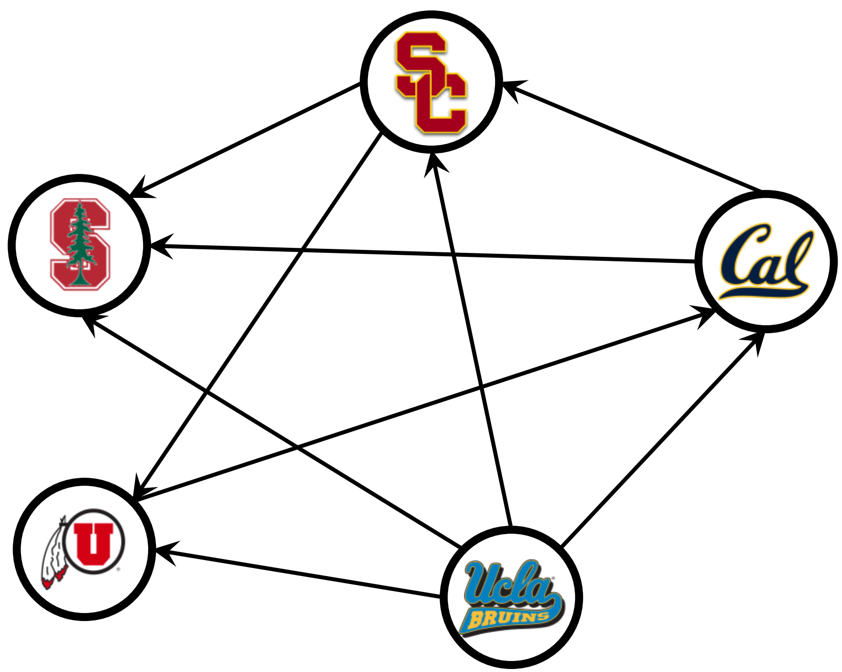
Strongly Connected Components

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.

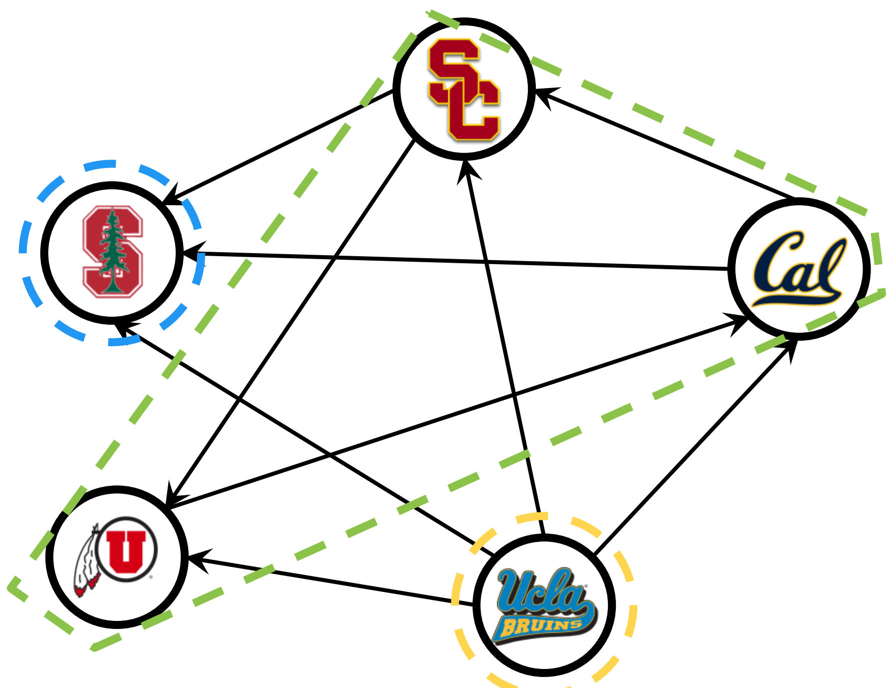
Strongly Connected Components

How many SCCs are in this graph? 🤔



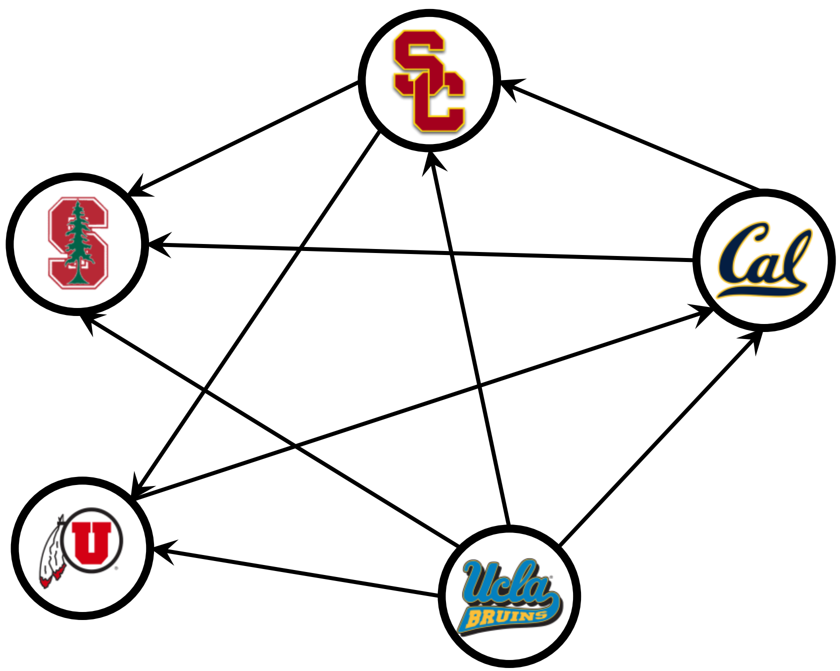
Strongly Connected Components

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



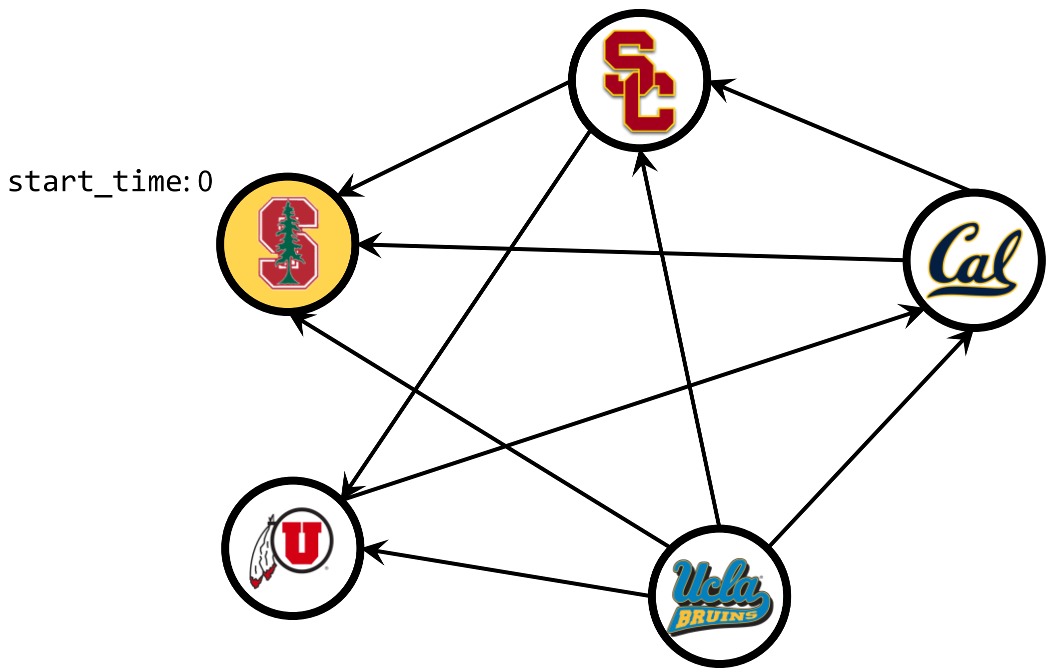
Kosaraju's Algorithm

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



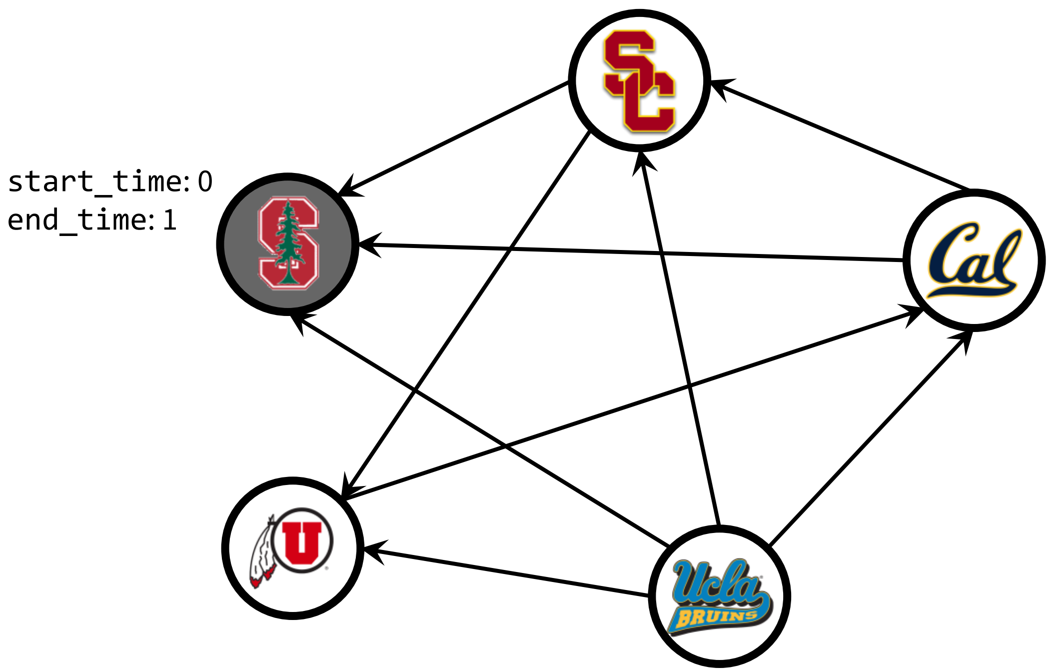
Kosaraju's Algorithm

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



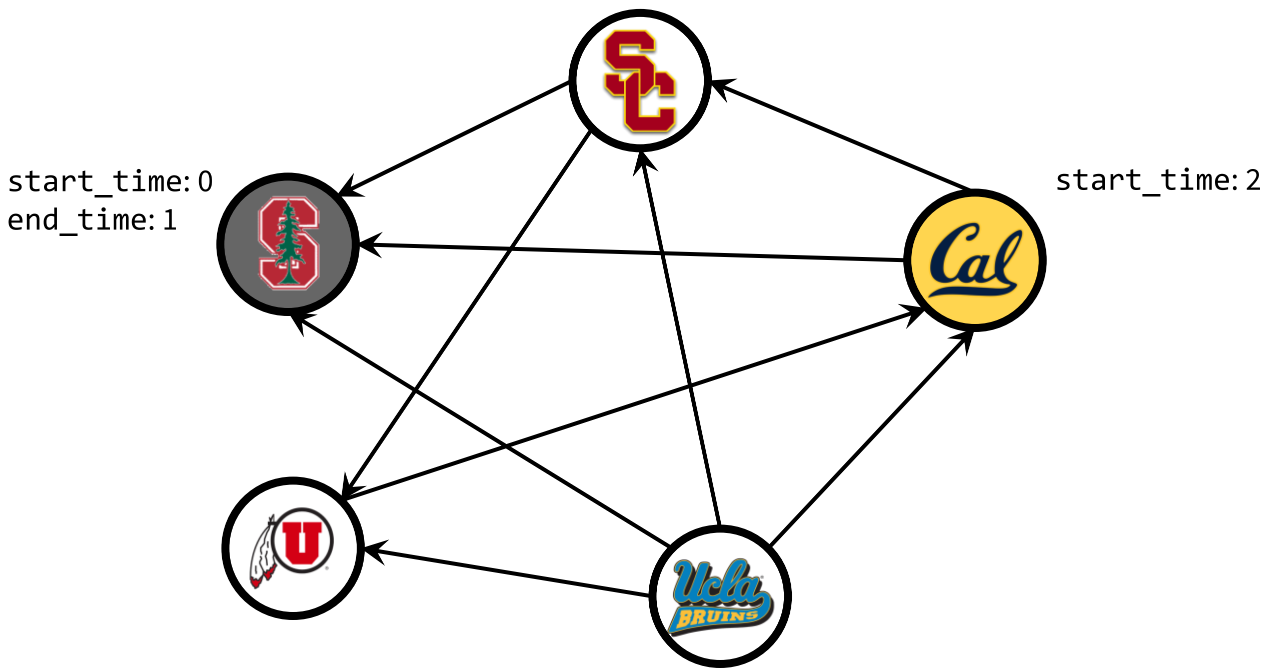
Kosaraju's Algorithm

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



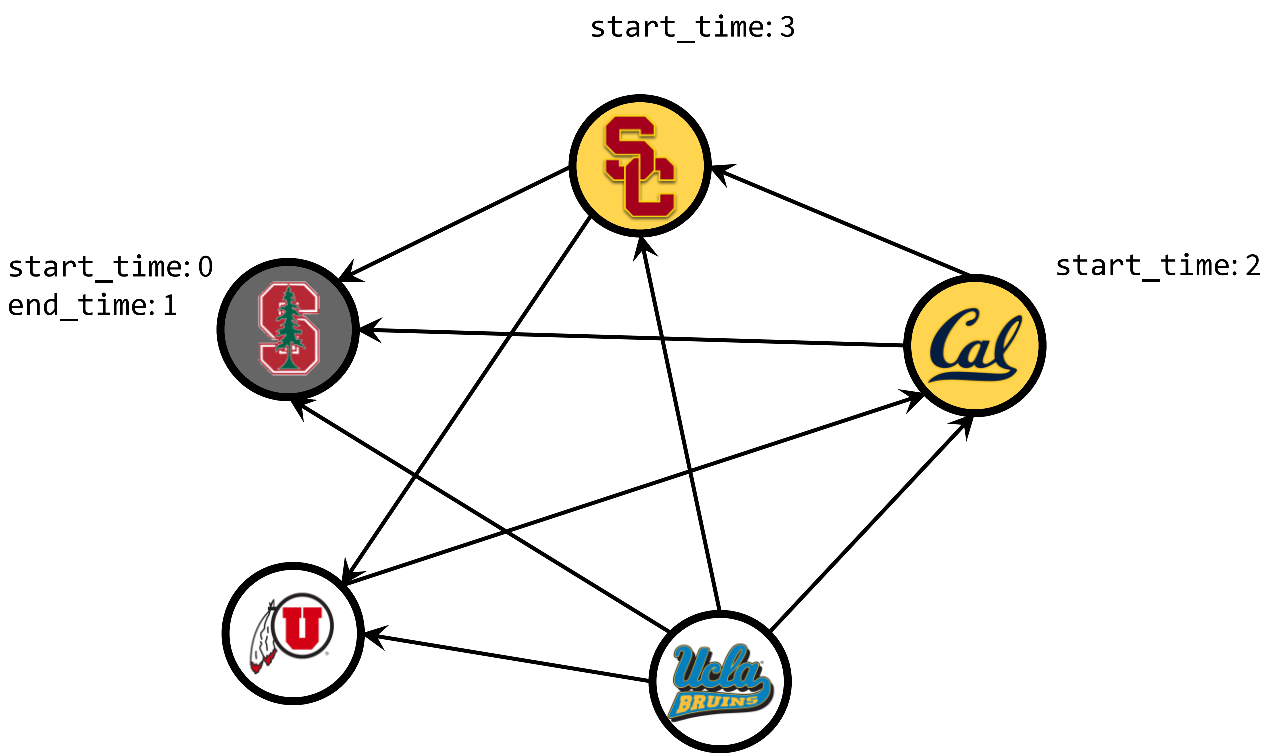
Kosaraju's Algorithm

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



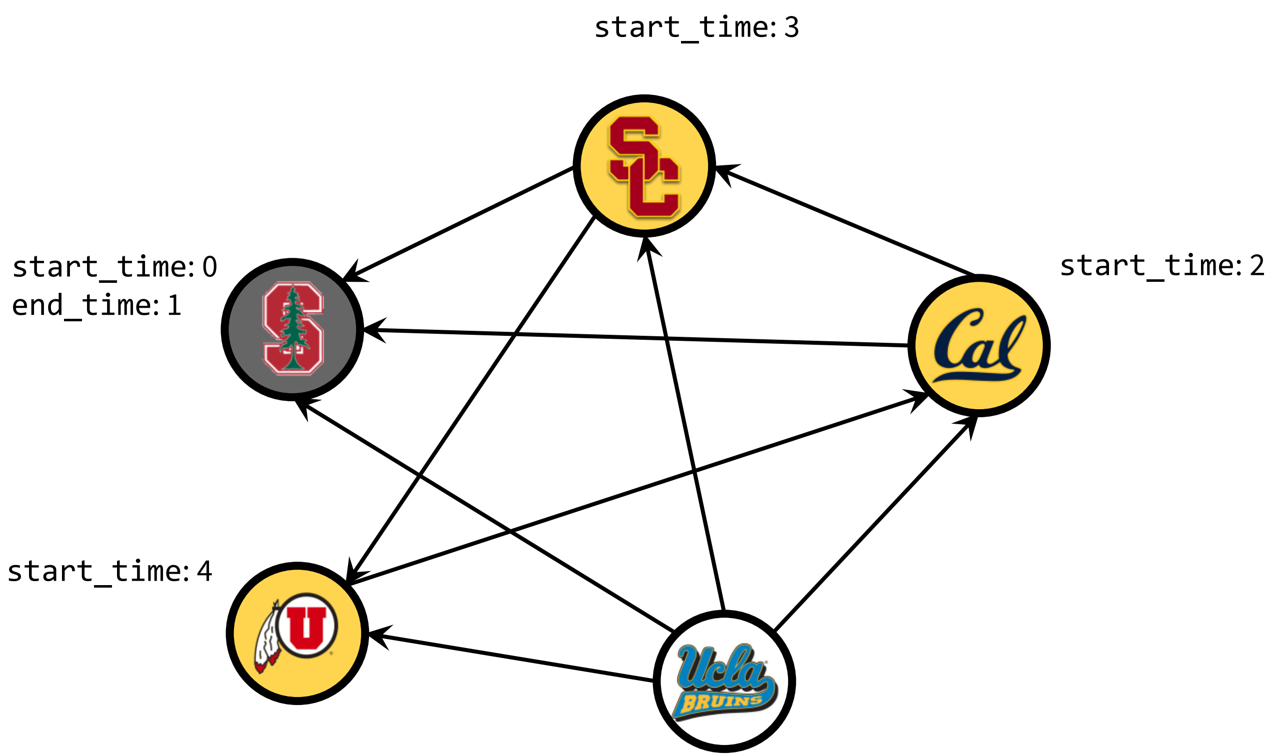
Kosaraju's Algorithm

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



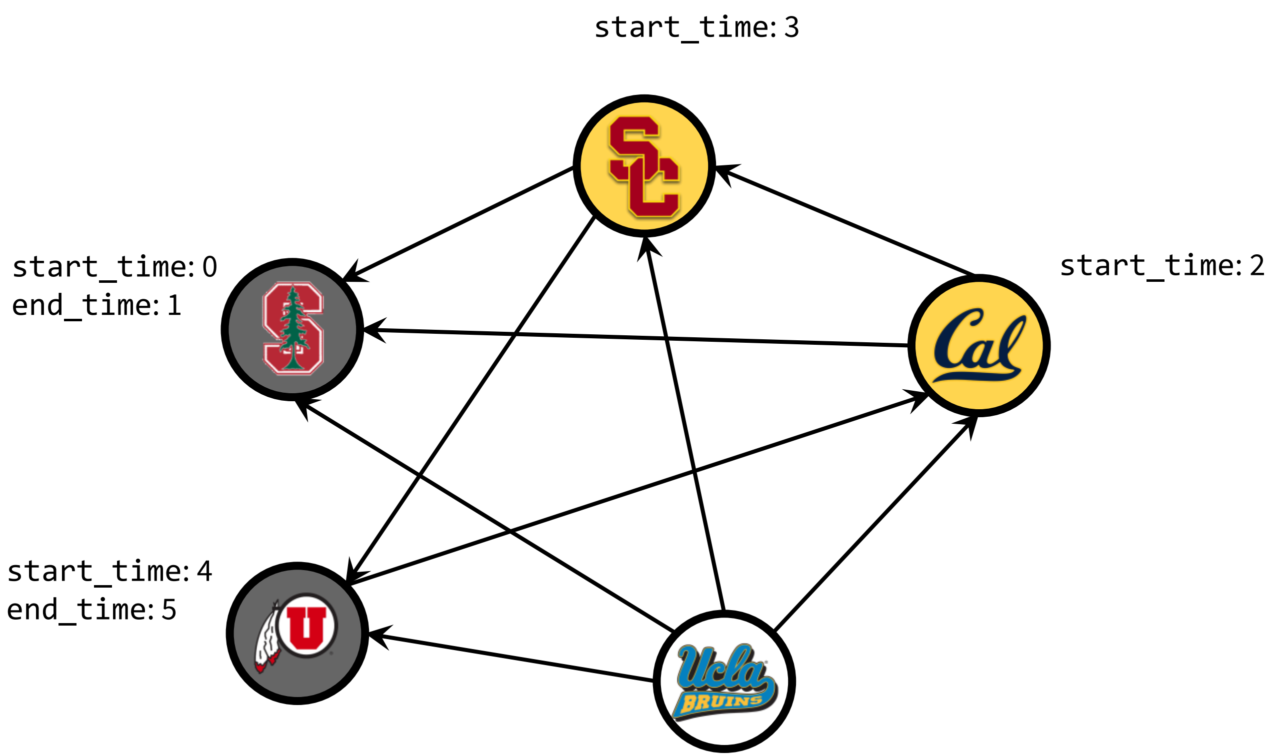
Kosaraju's Algorithm

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



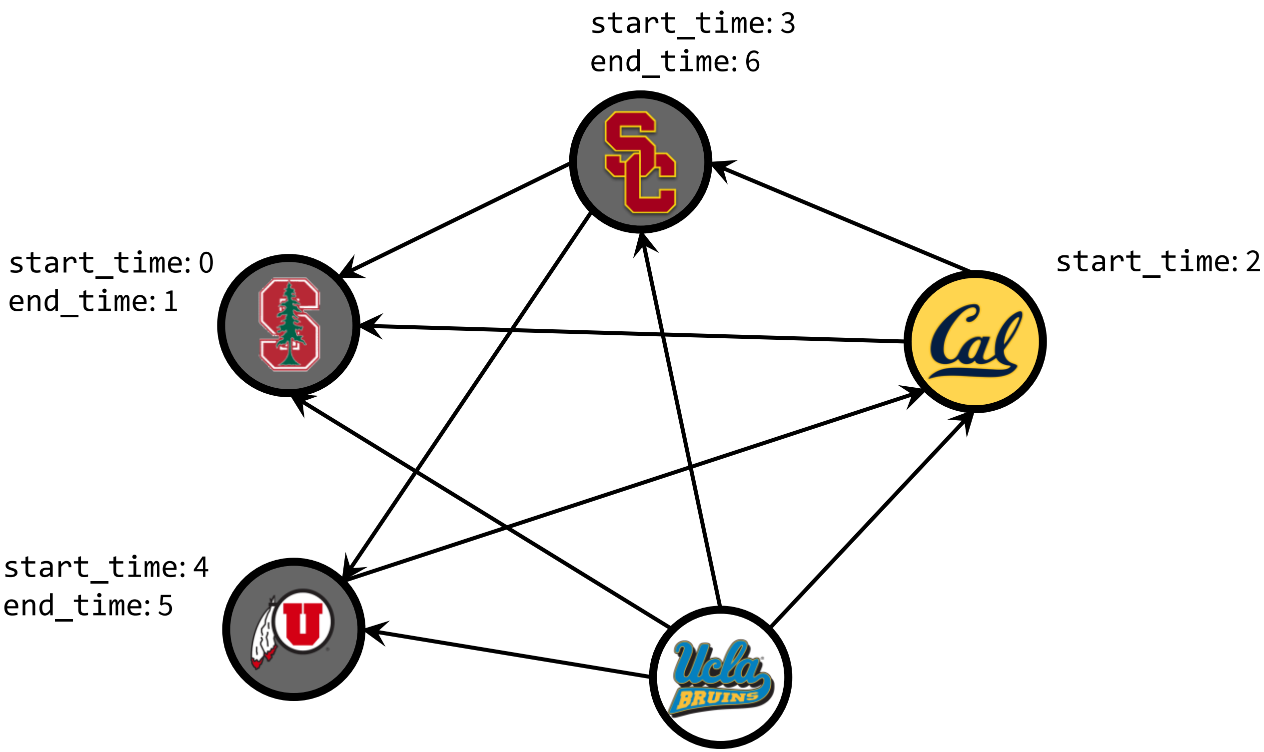
Kosaraju's Algorithm

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



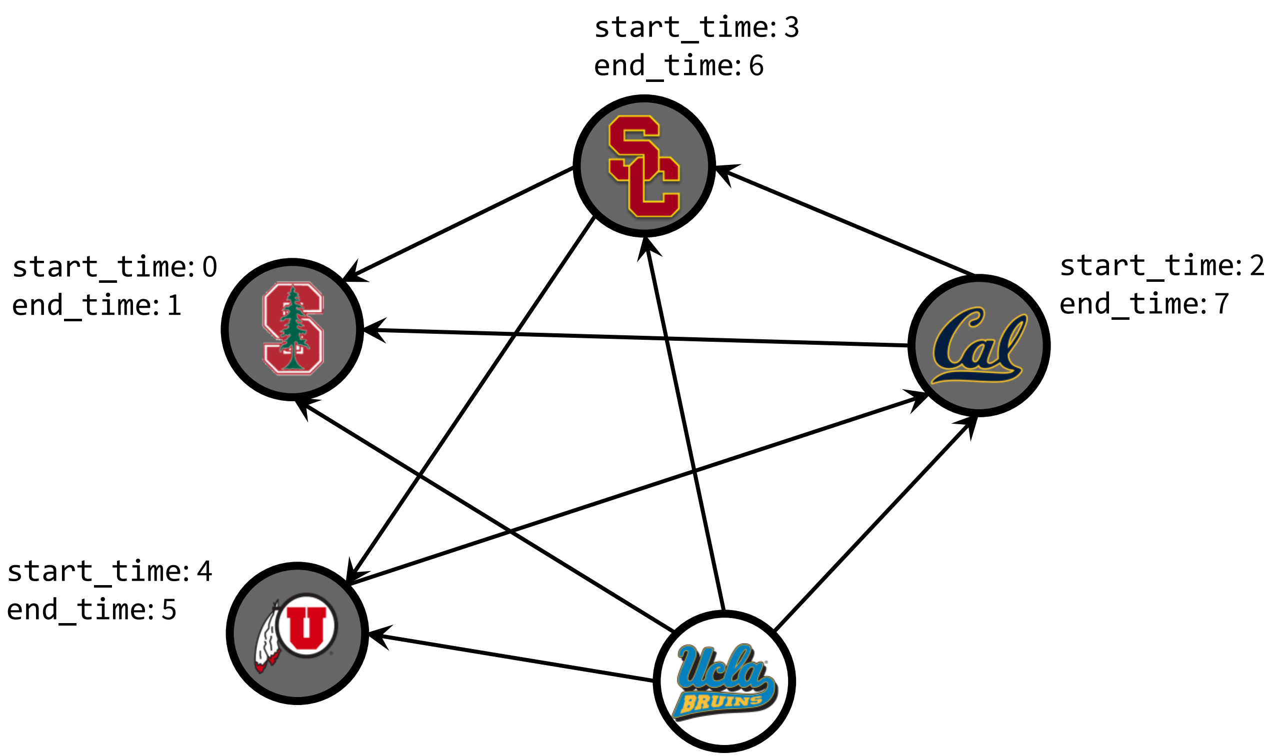
Kosaraju's Algorithm

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



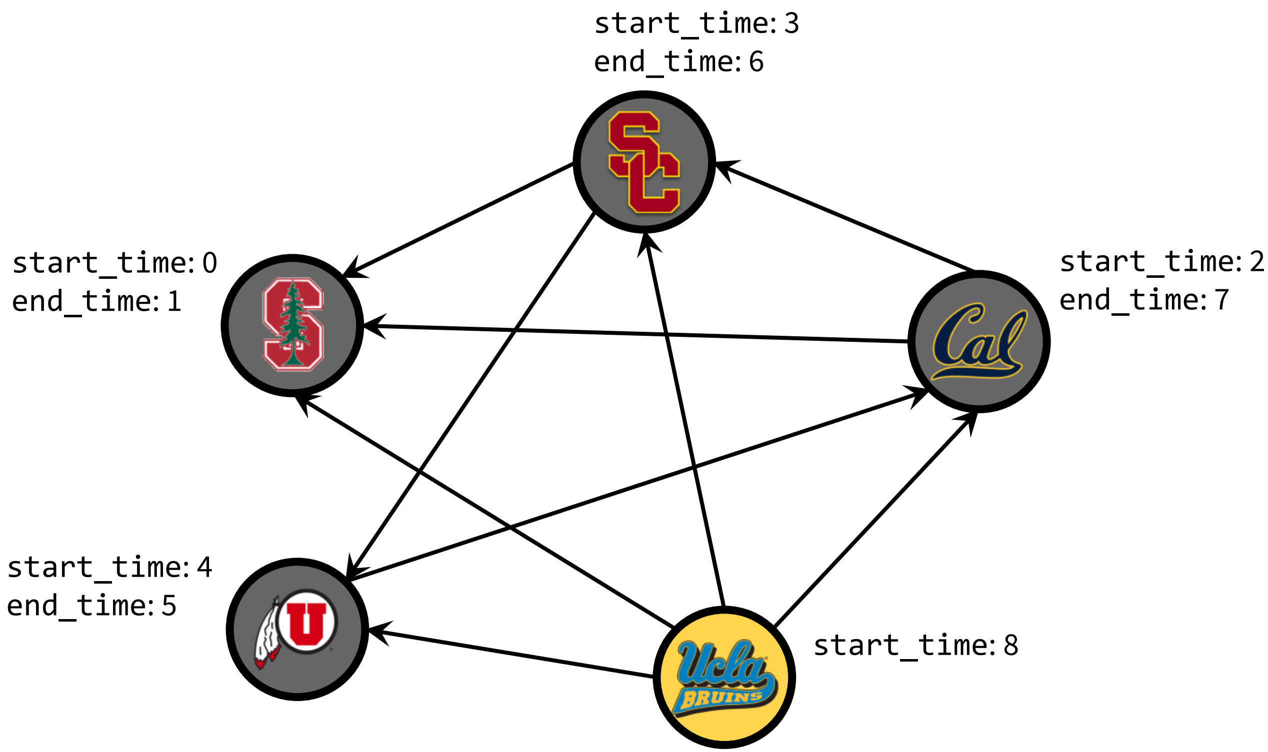
Kosaraju's Algorithm

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



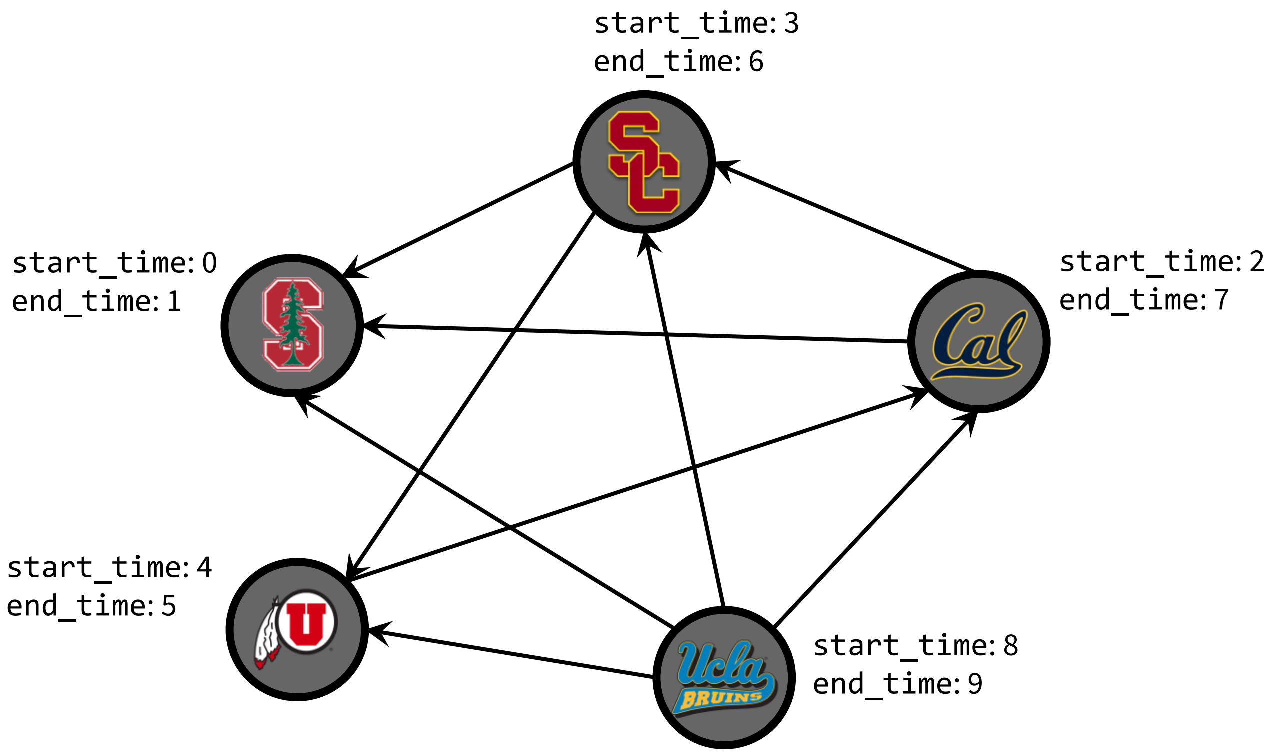
Kosaraju's Algorithm

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



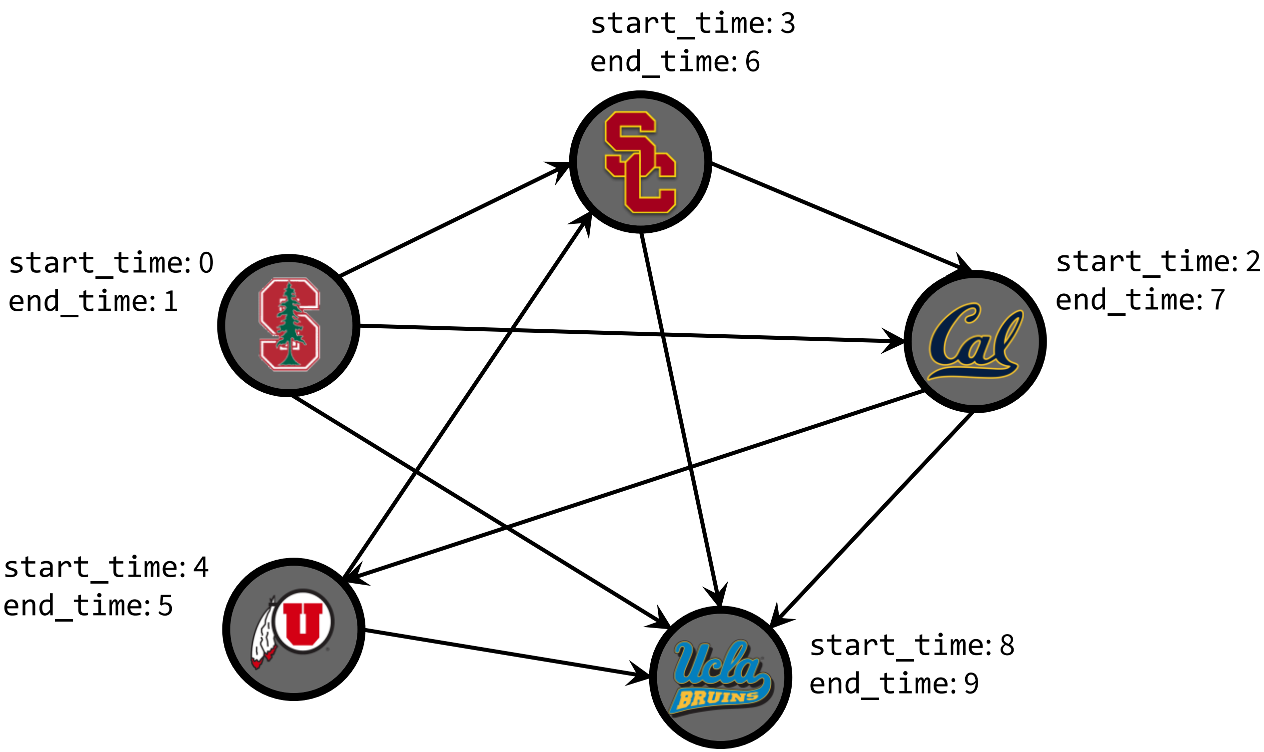
Kosaraju's Algorithm

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



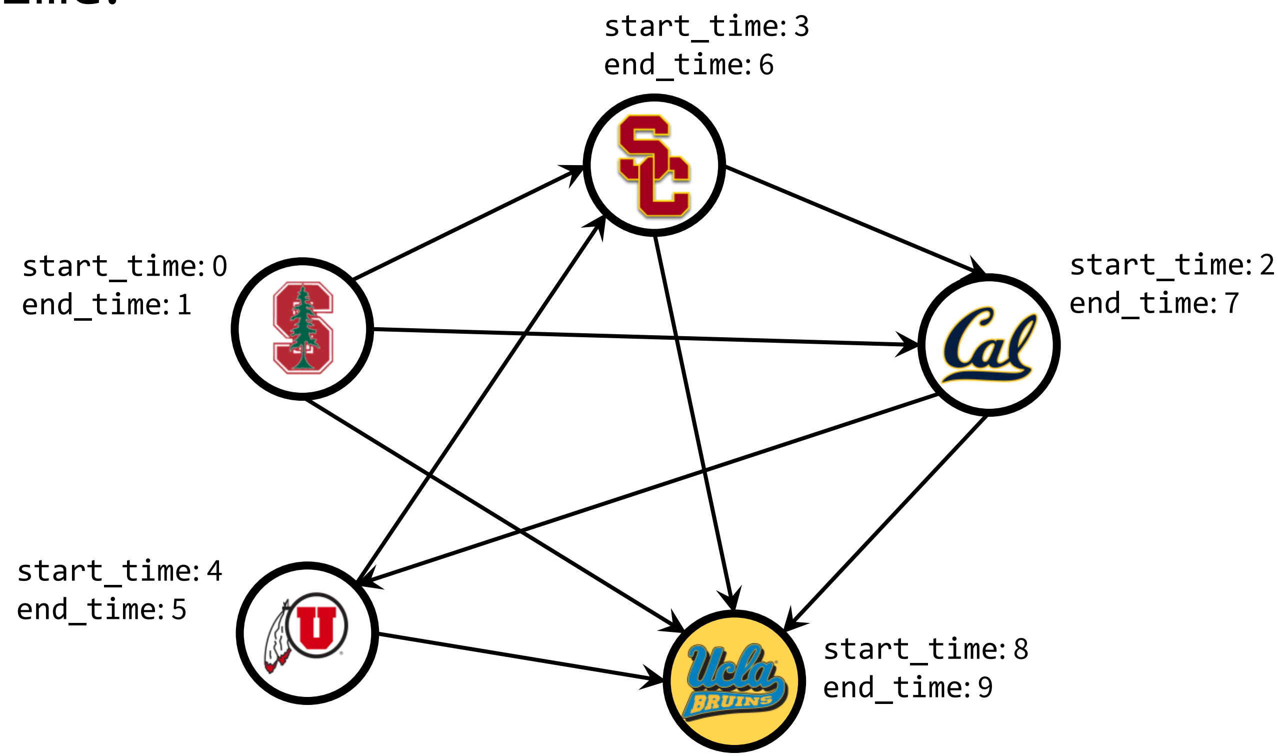
Kosaraju's Algorithm

2. Reverse all of the edges.



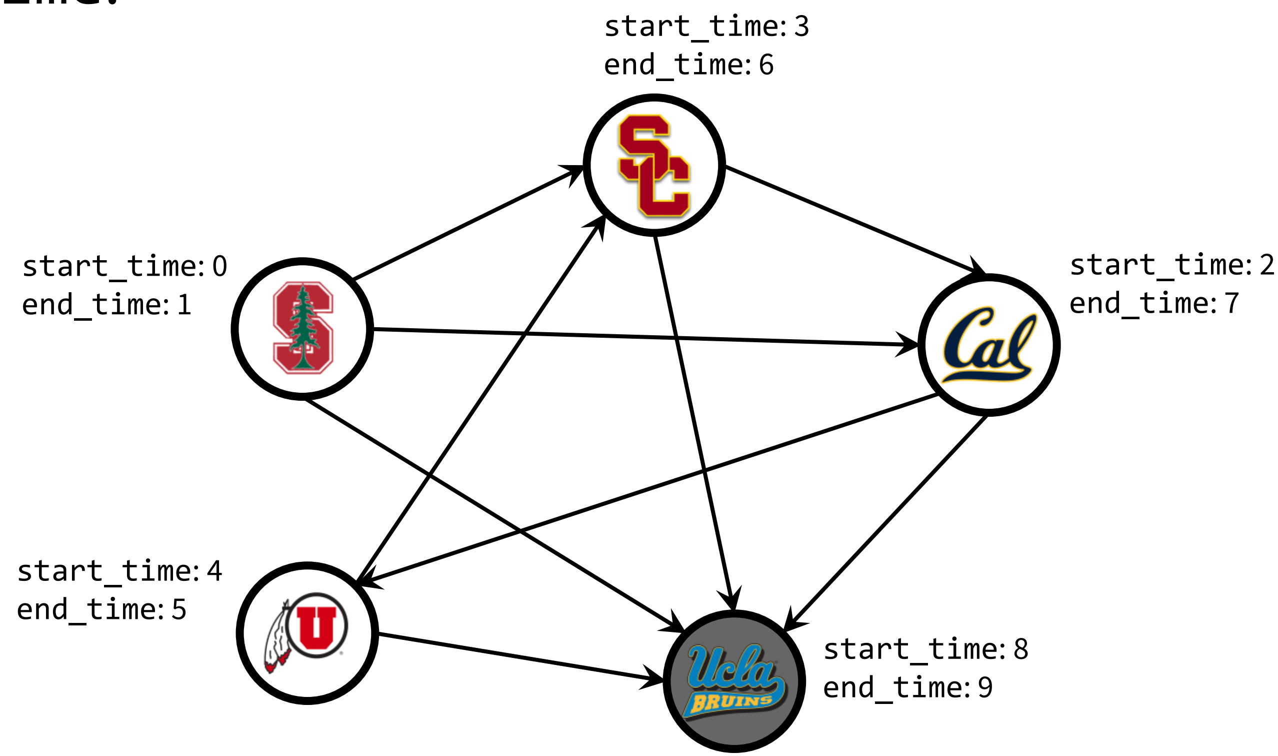
Kosaraju's Algorithm

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



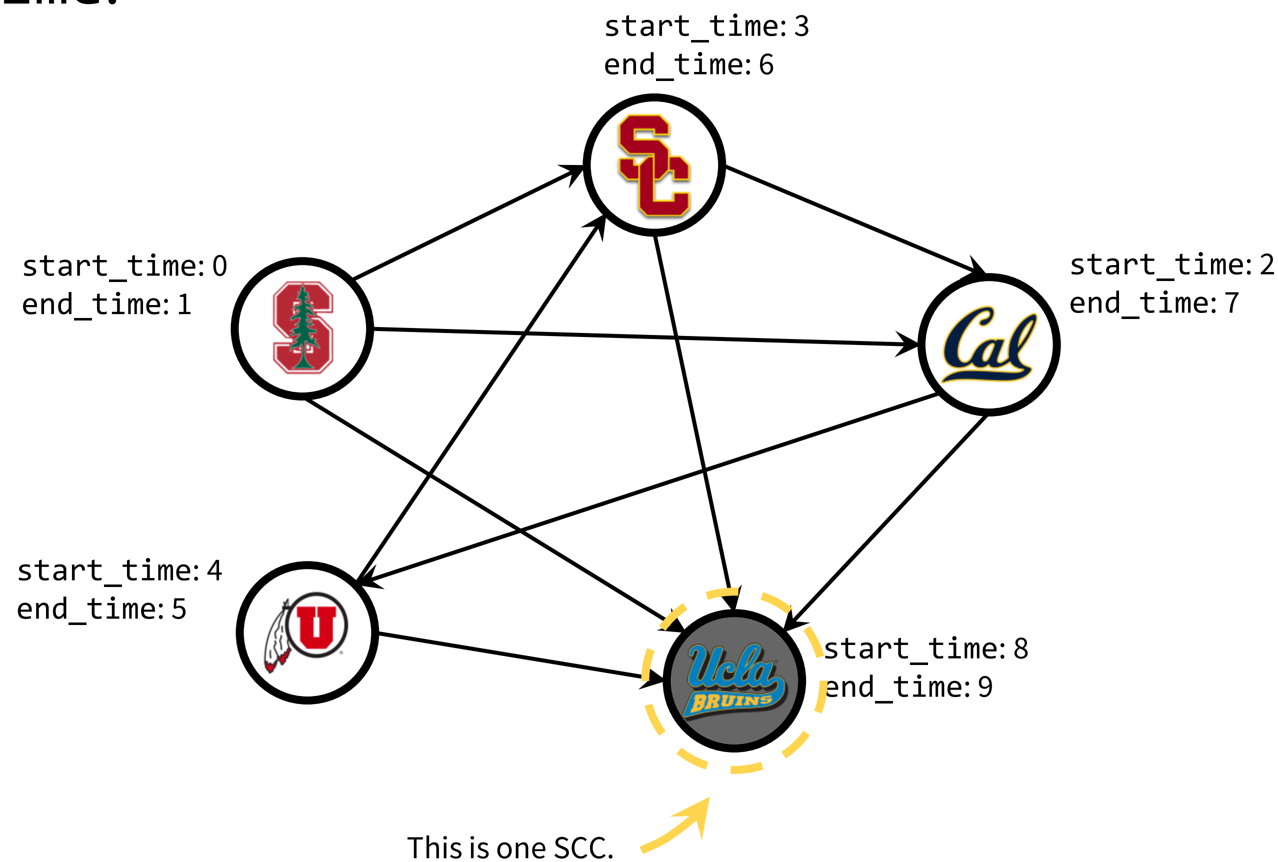
Kosaraju's Algorithm

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



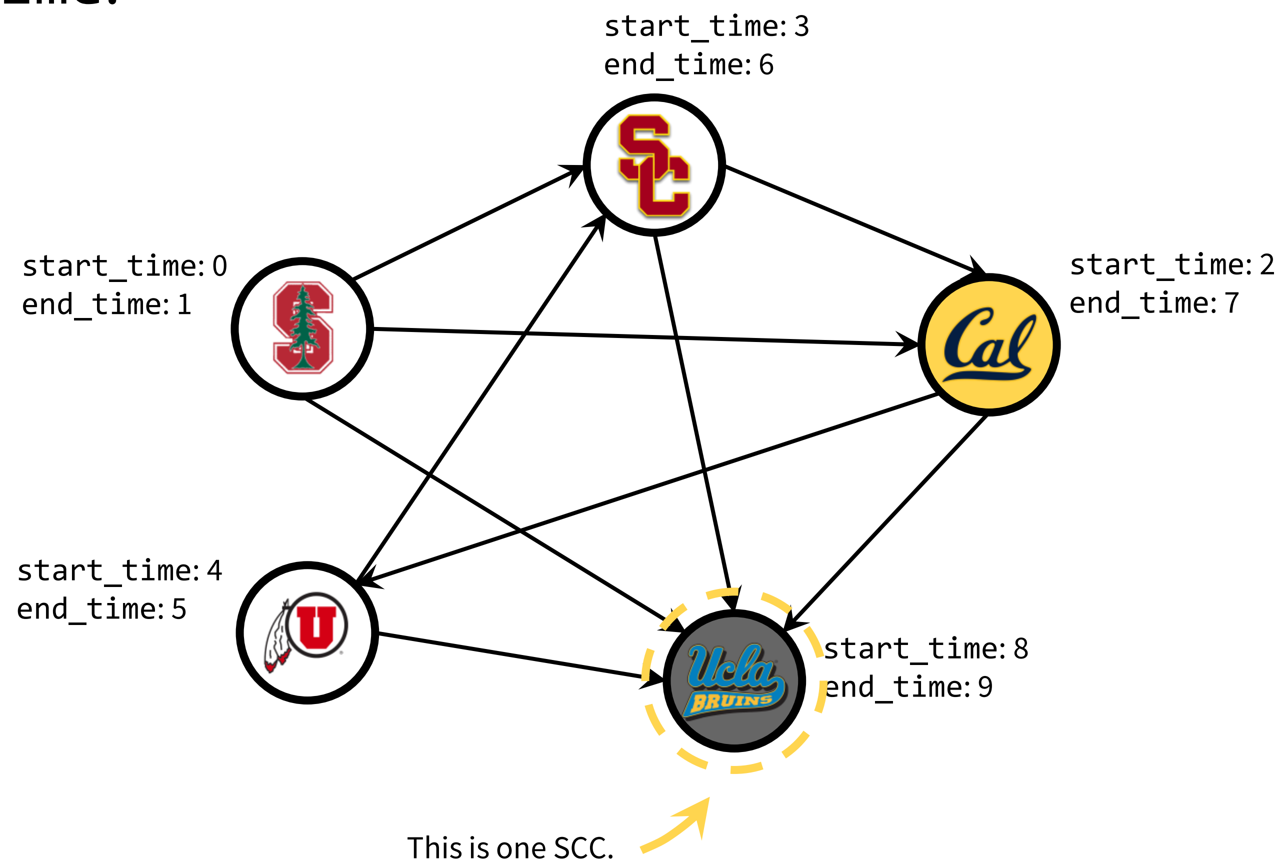
Kosaraju's Algorithm

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



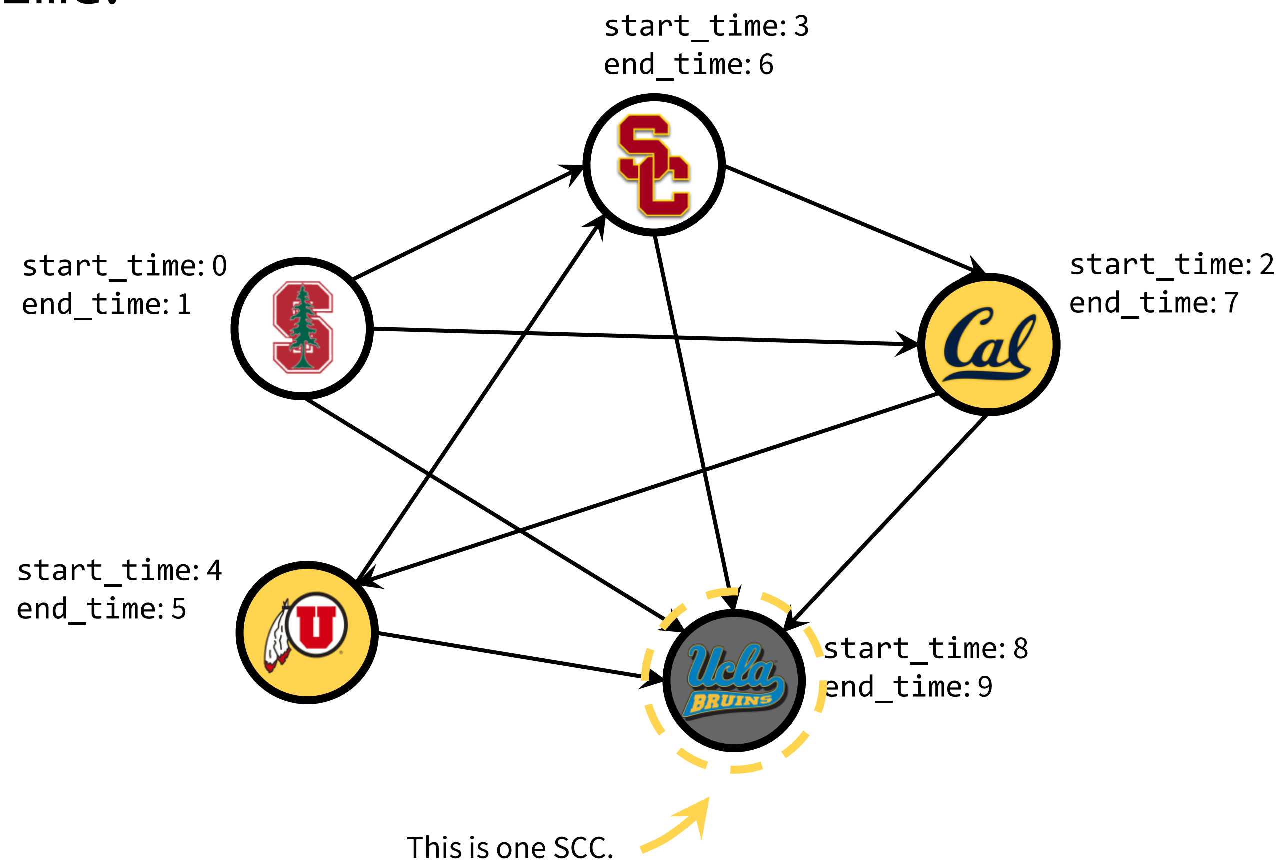
Kosaraju's Algorithm

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



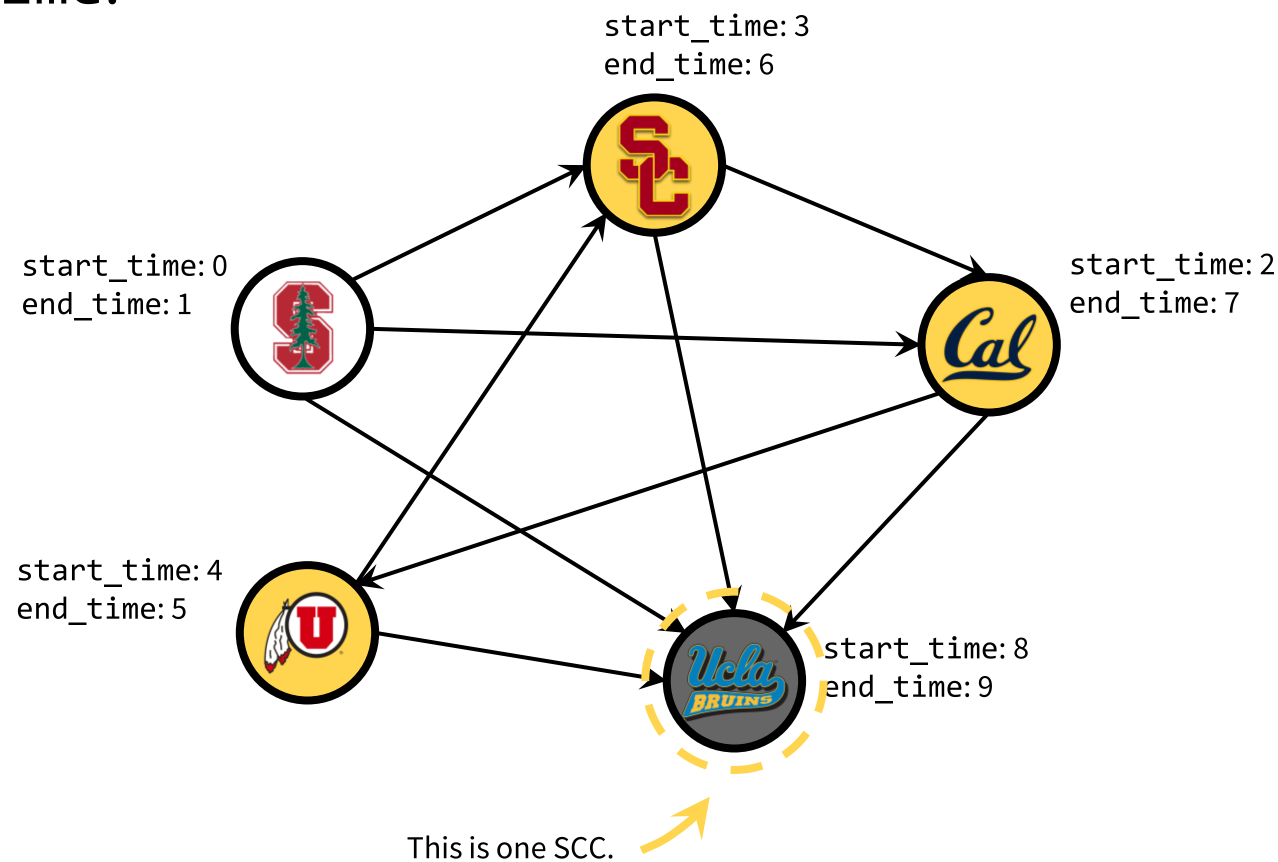
Kosaraju's Algorithm

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



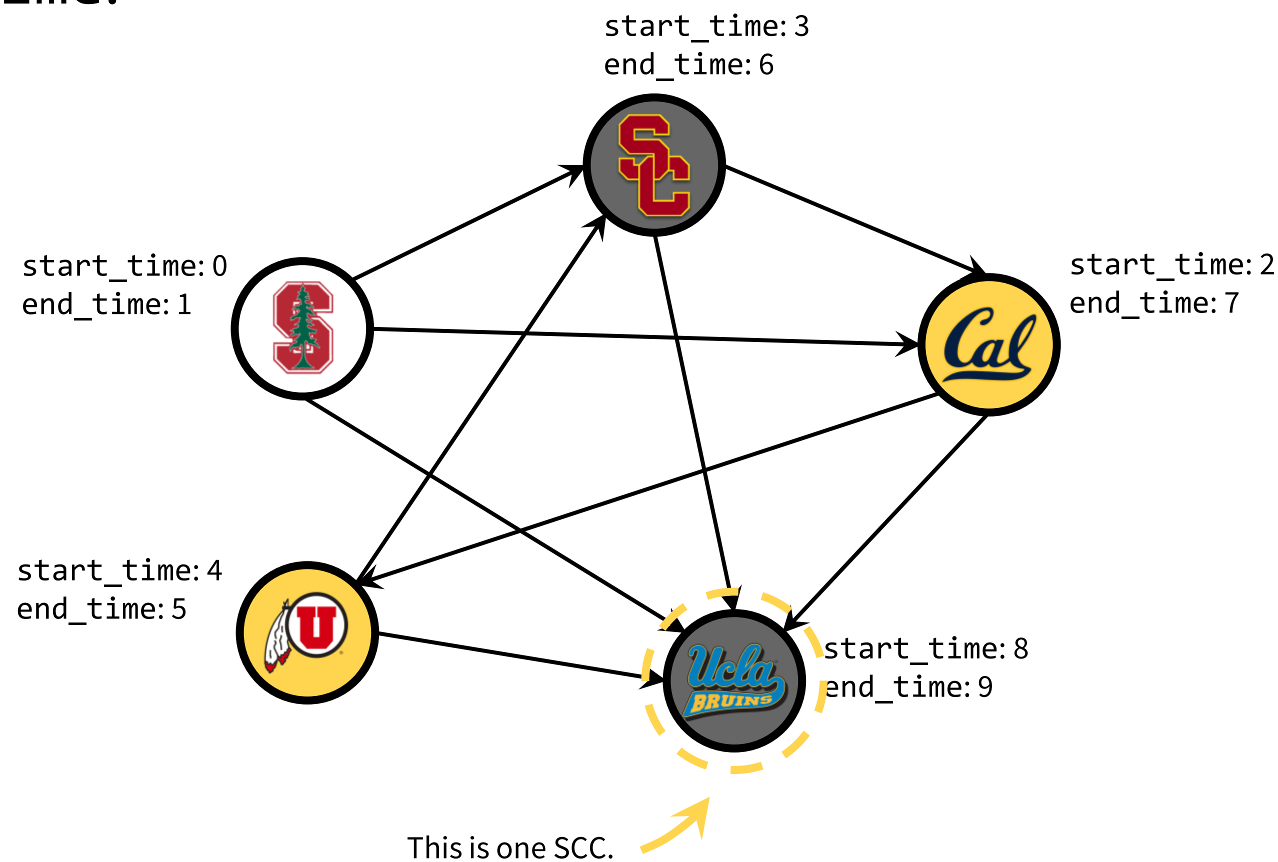
Kosaraju's Algorithm

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



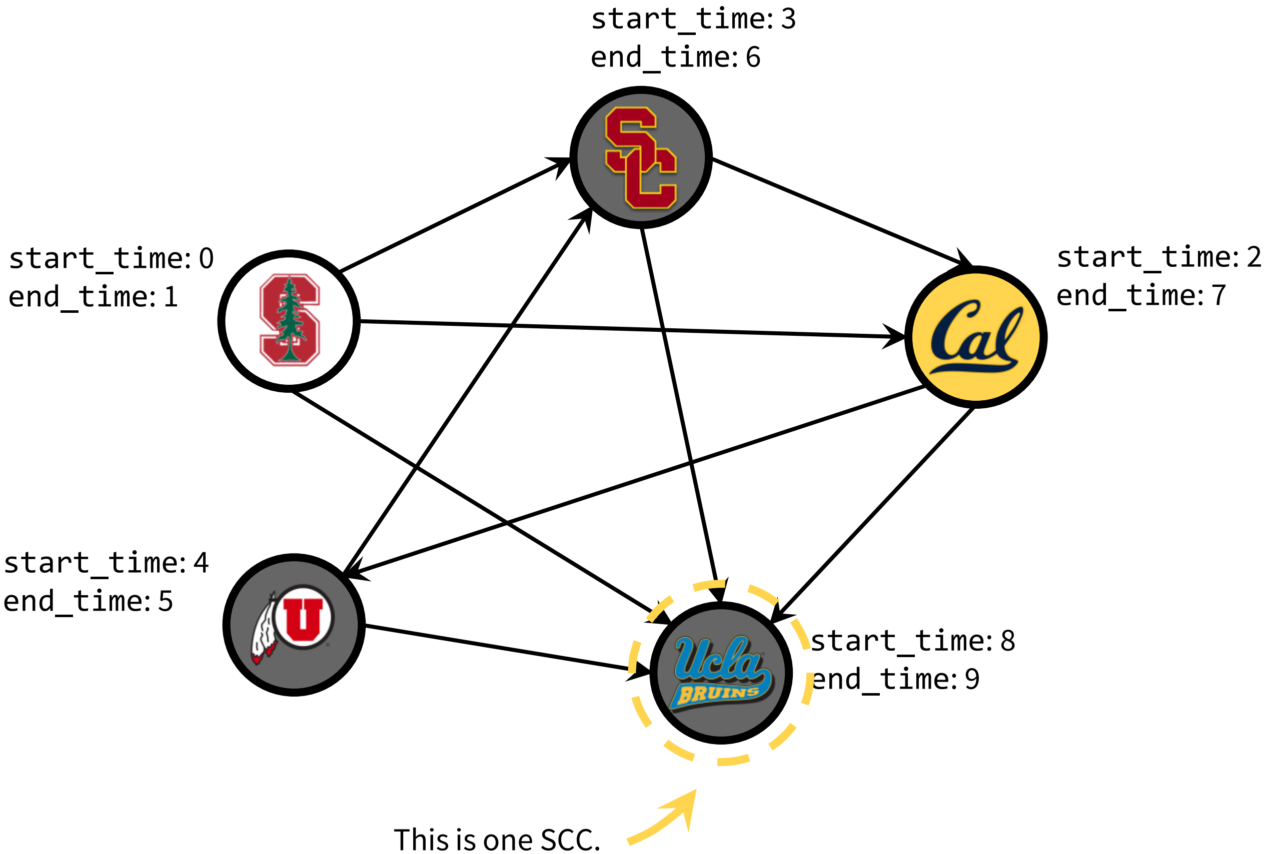
Kosaraju's Algorithm

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



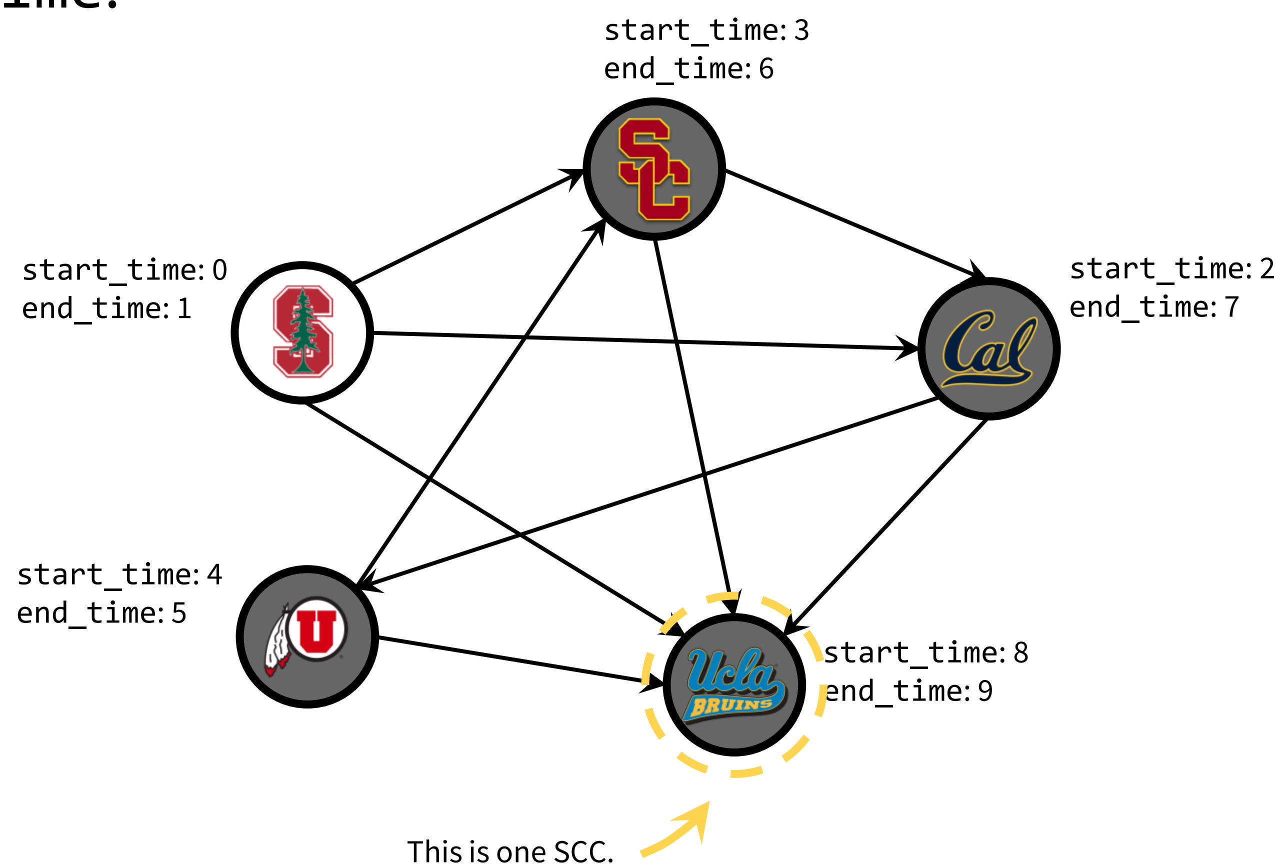
Kosaraju's Algorithm

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



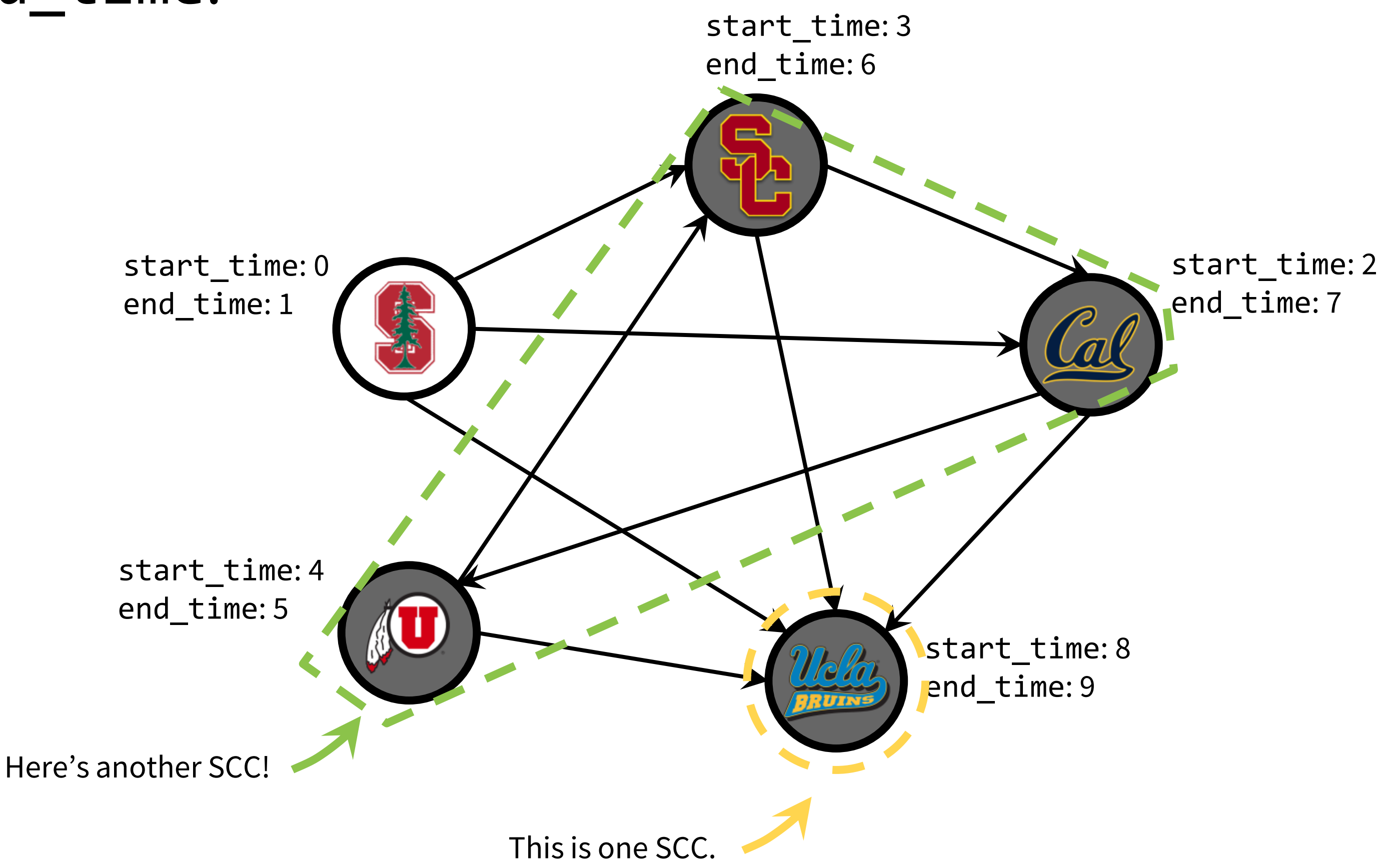
Kosaraju's Algorithm

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



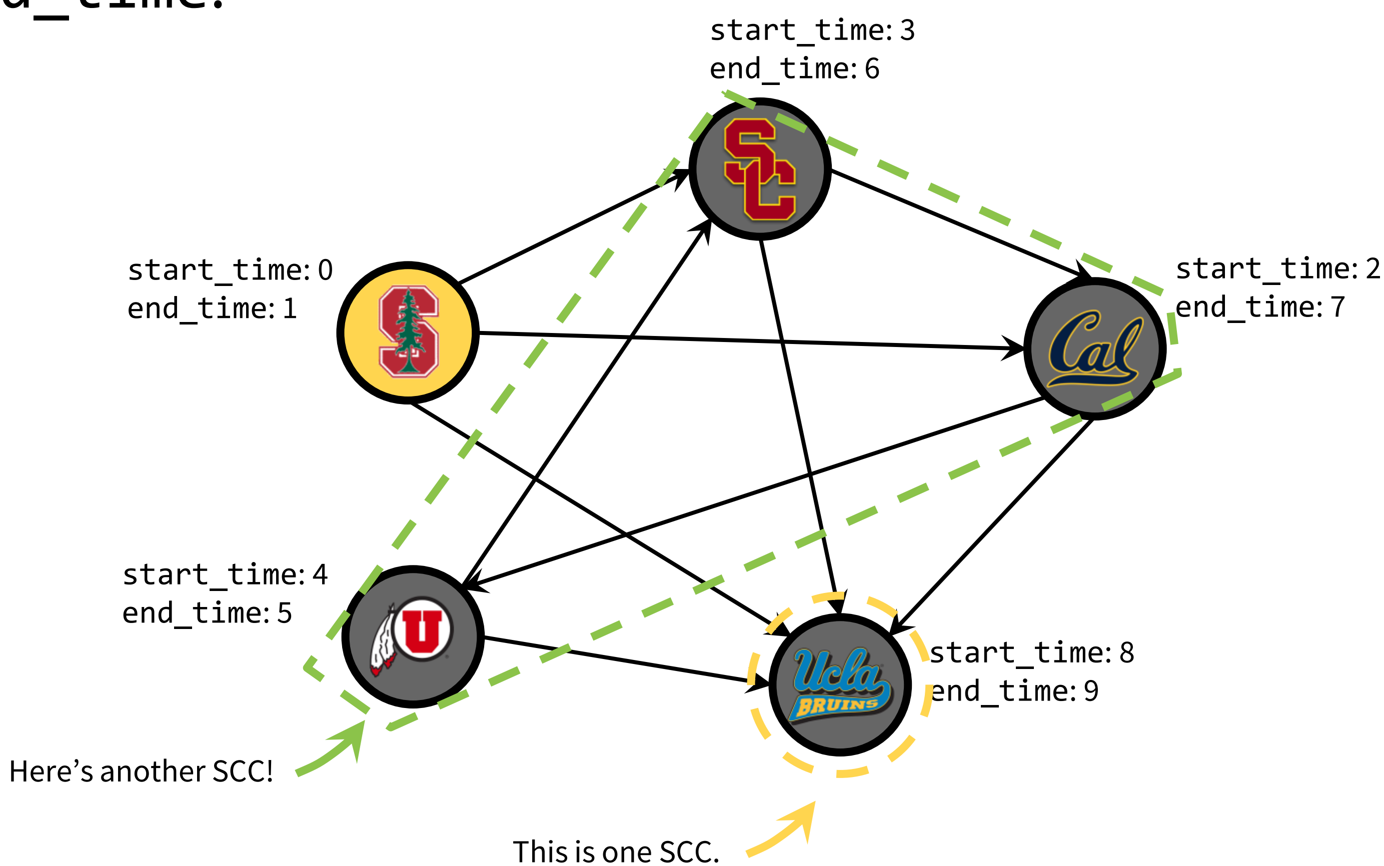
Kosaraju's Algorithm

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



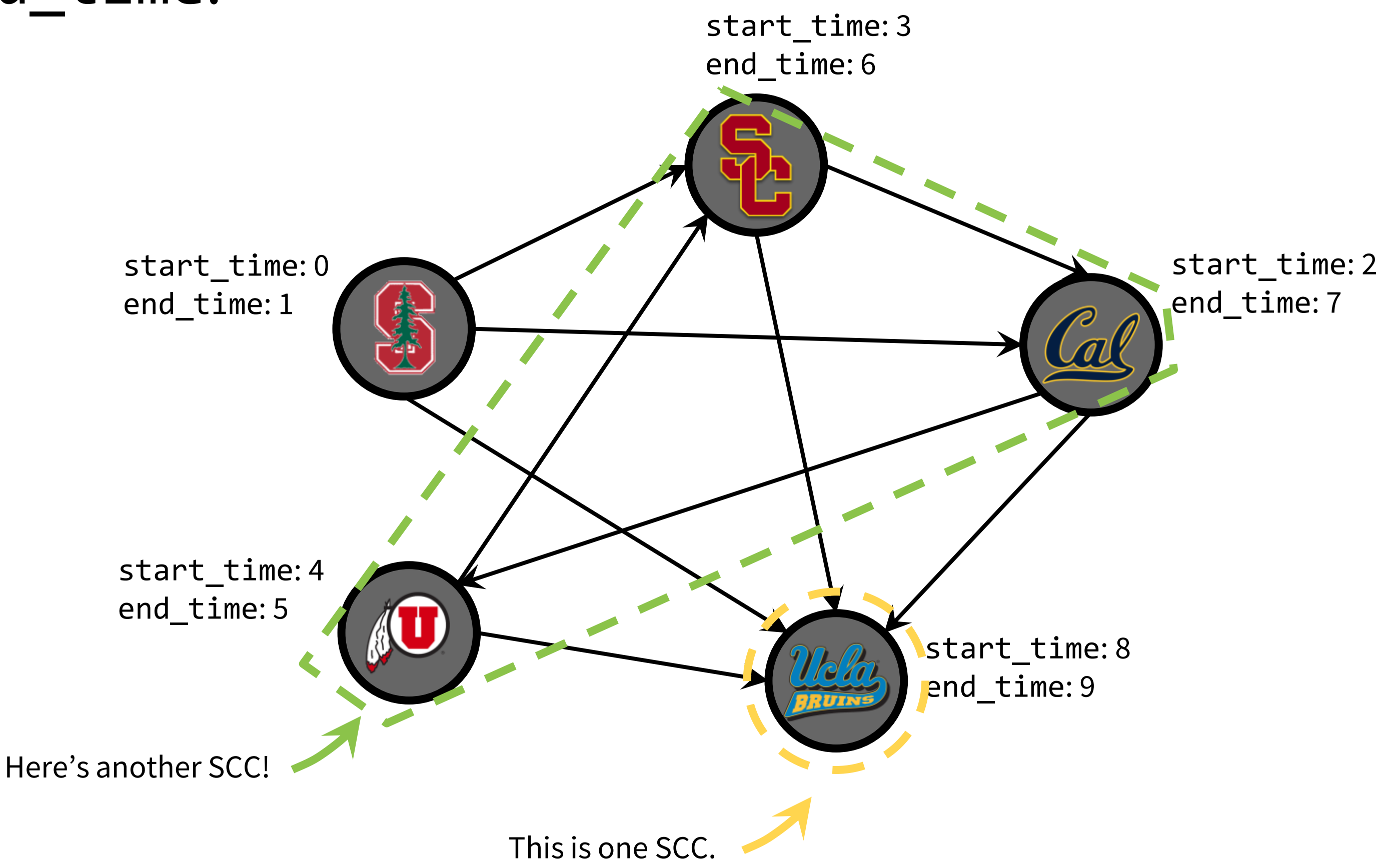
Kosaraju's Algorithm

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



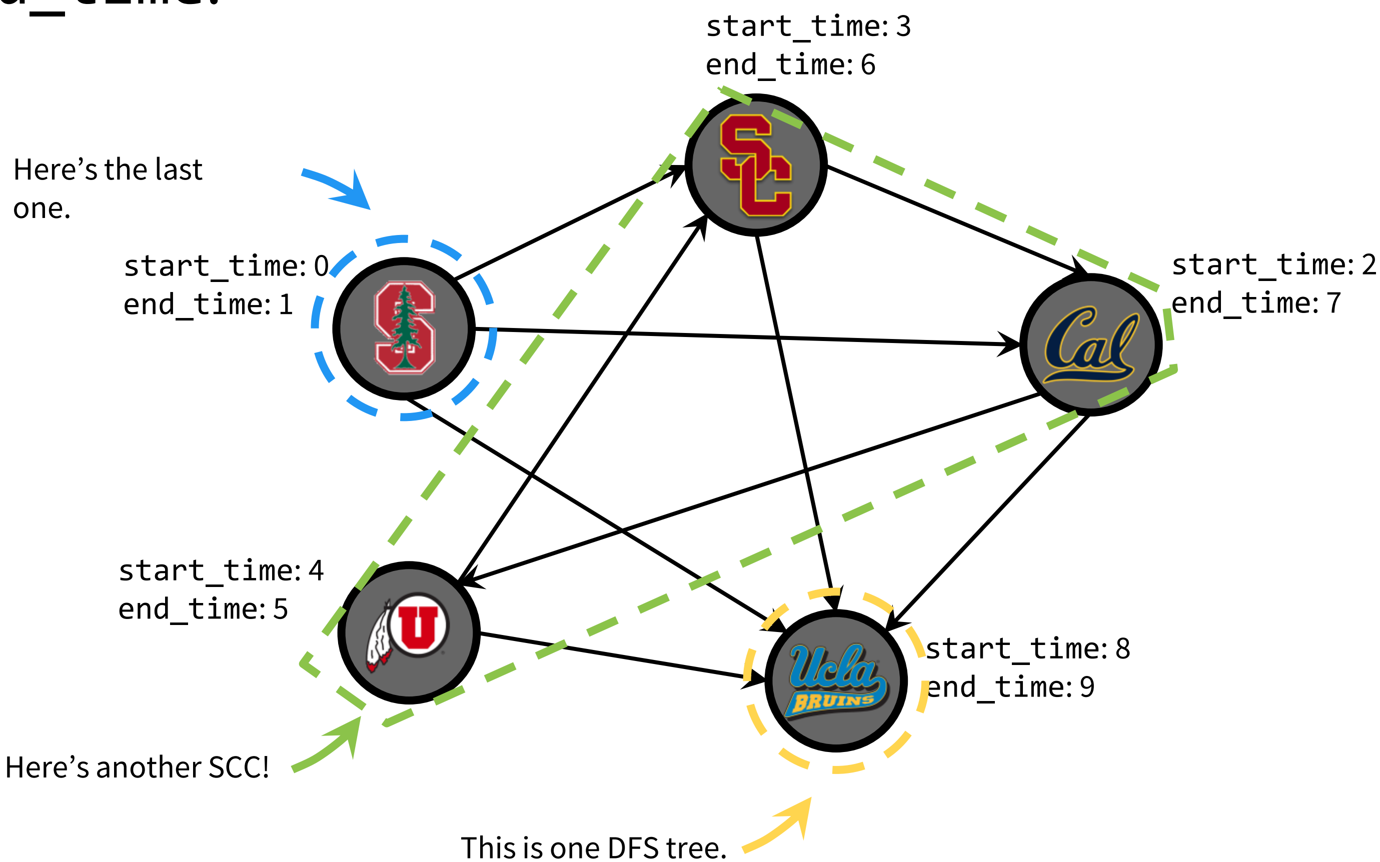
Kosaraju's Algorithm

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`.

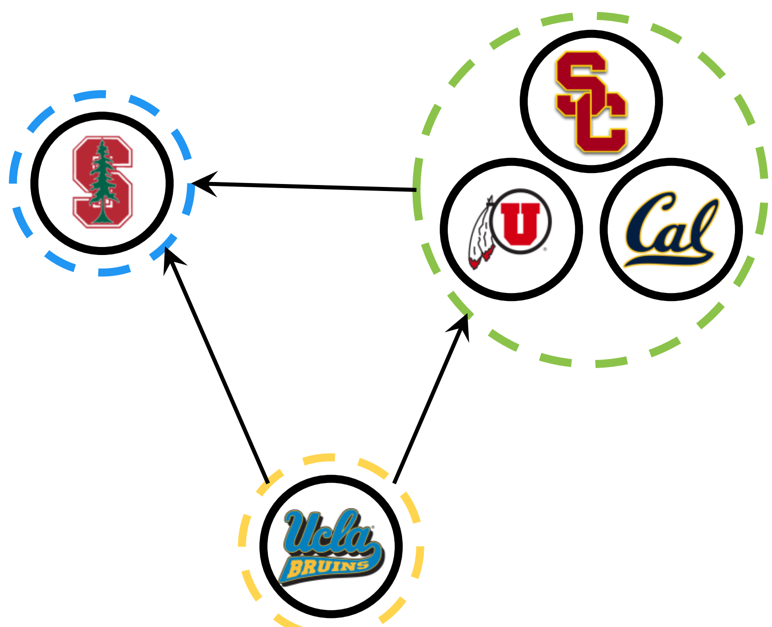


Kosaraju's Algorithm

Whoa. How did that work?

Lemma 1: The SCC metagraph is a DAG.

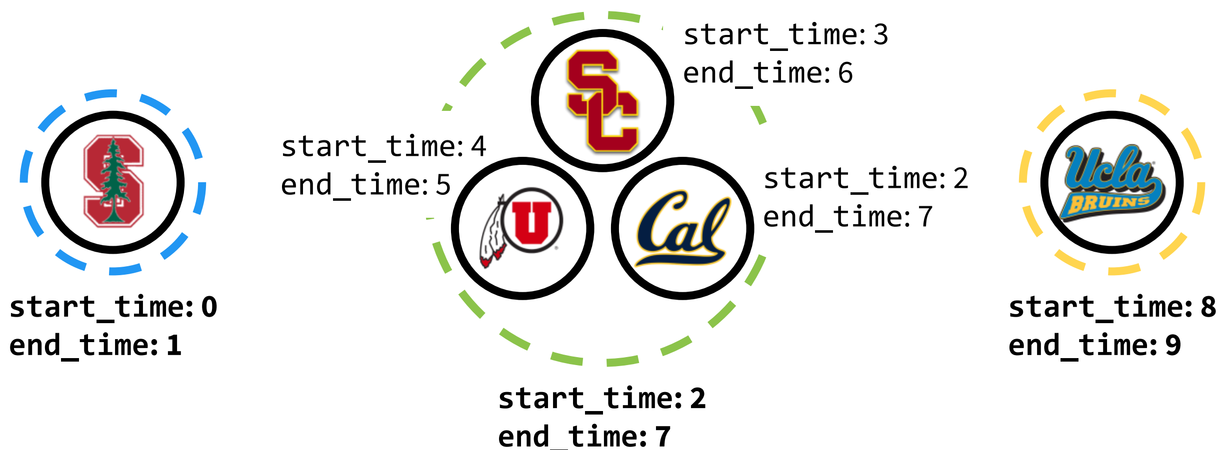
Intuition: If not, then two SCCs would collapse into one.



Kosaraju's Algorithm

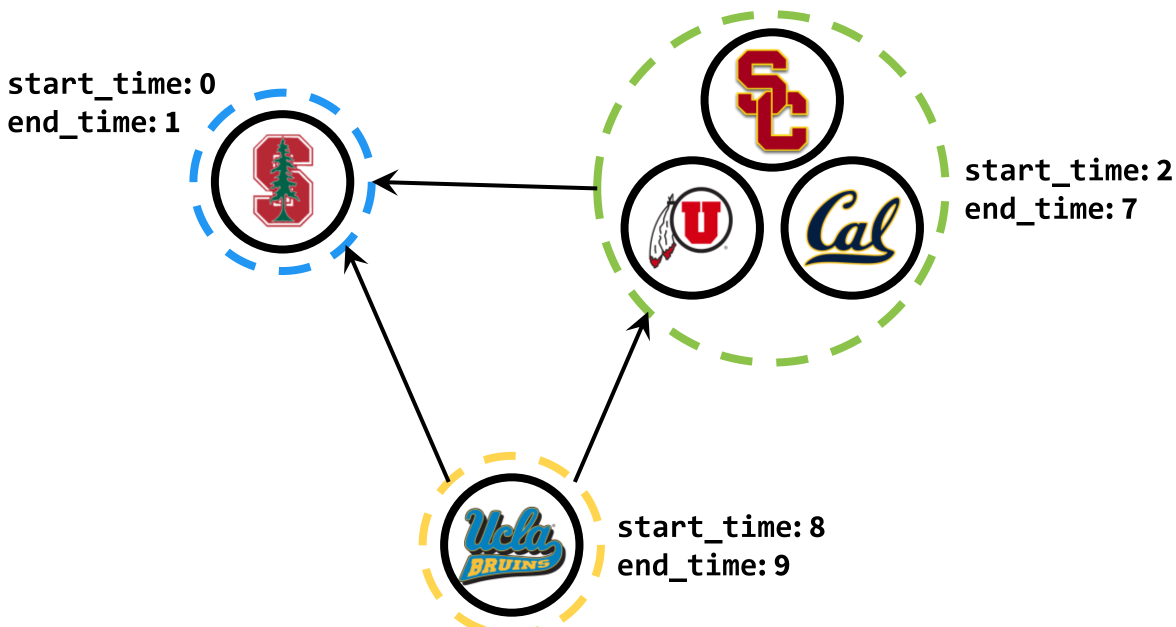
Let the **end time** of a SCC be the largest end time of any element of that SCC.

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



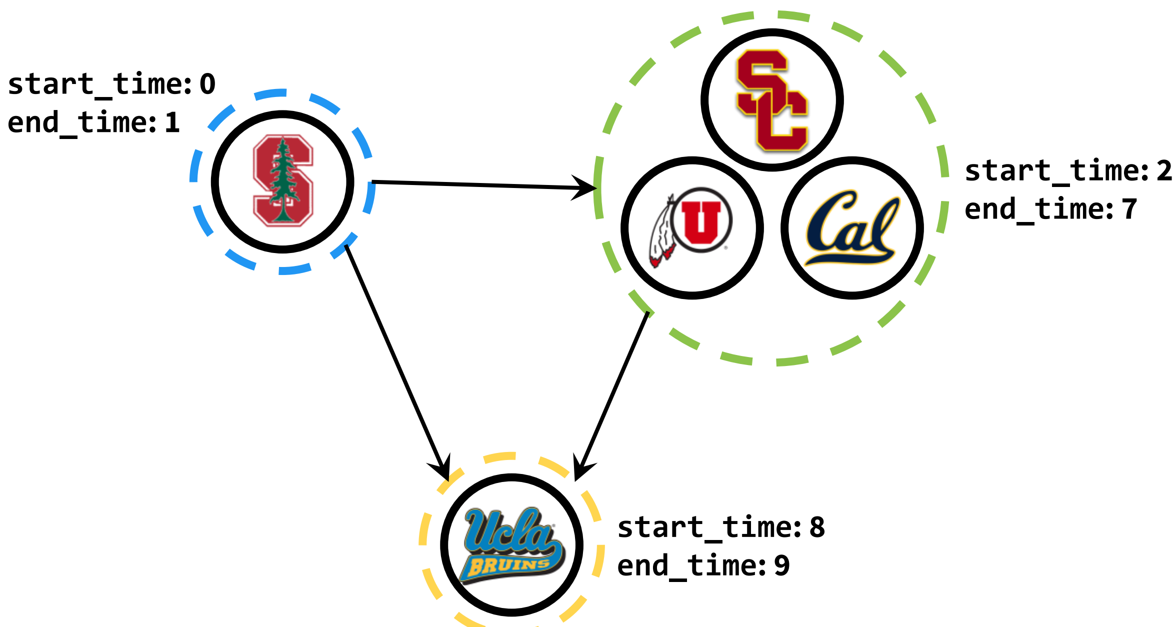
Kosaraju's Algorithm

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



Kosaraju's Algorithm

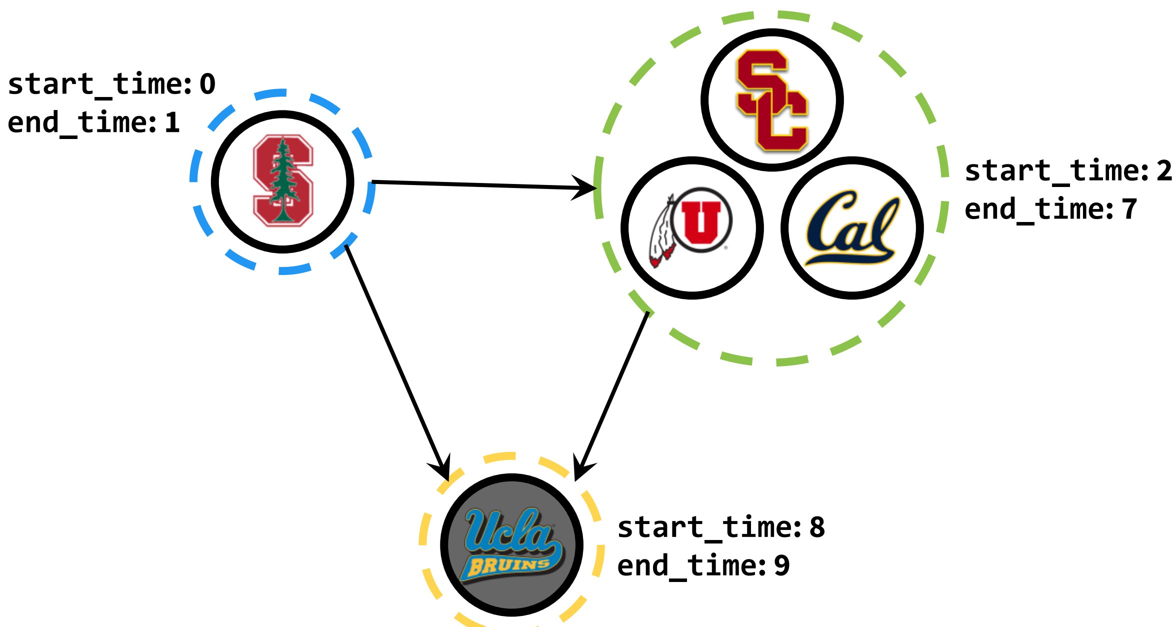
The main idea leverages the fact that vertex in the SCC metagraph with the largest `end_time` has no incoming edges. After reversing the edges, it has no outgoing edges.



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.

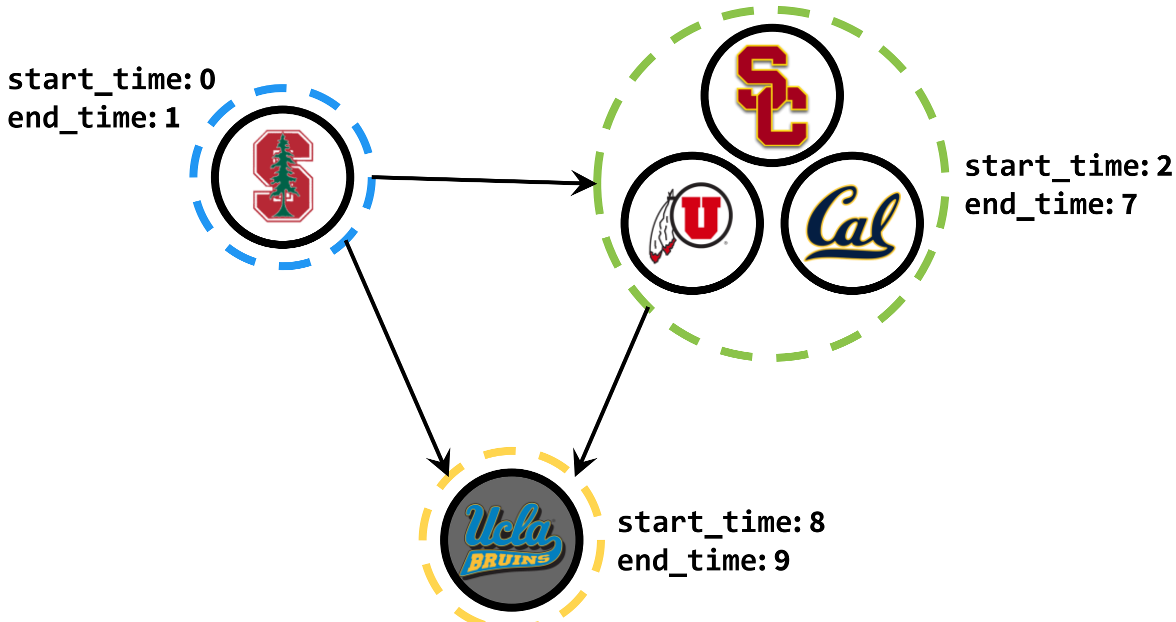


Kosaraju's Algorithm

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.

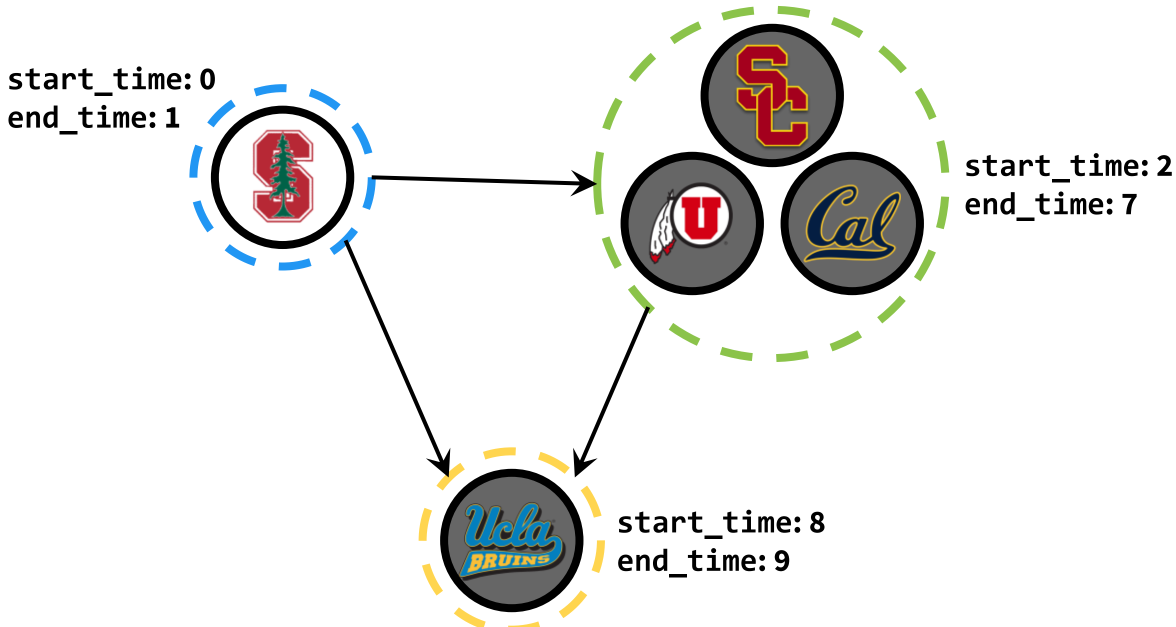


Kosaraju's Algorithm

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.

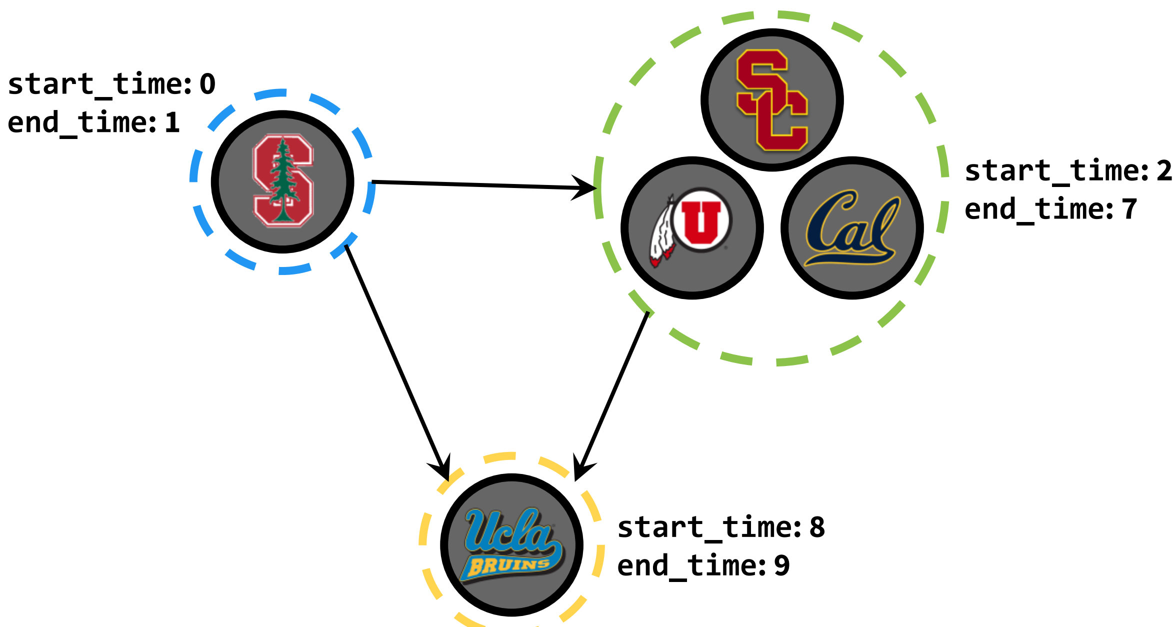


Kosaraju's Algorithm

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.

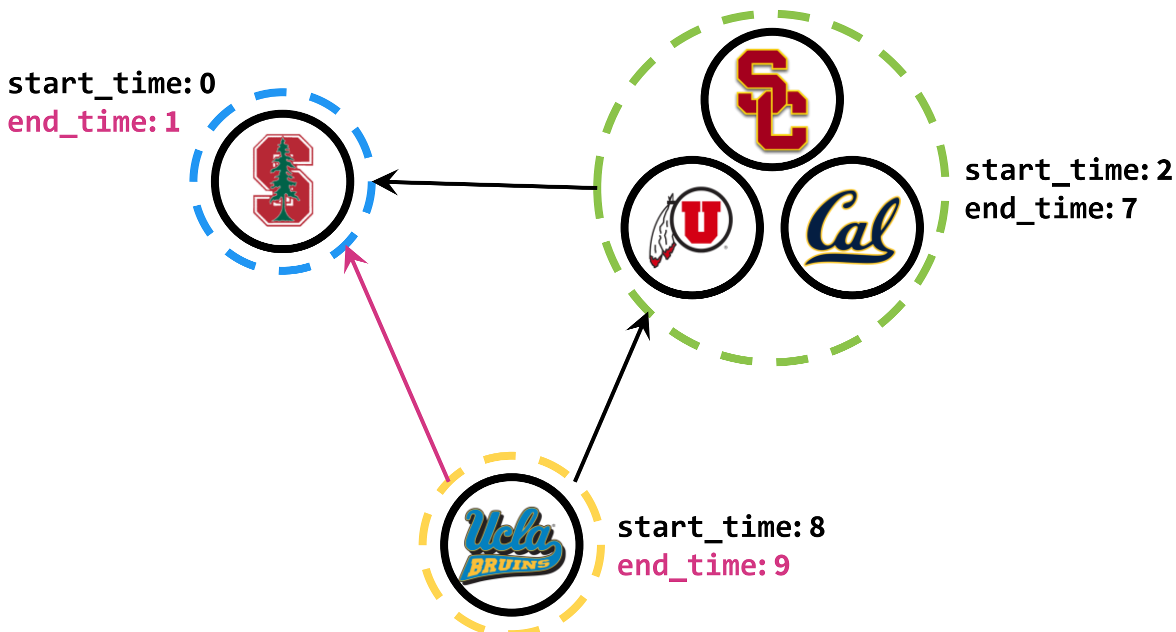
Running `dfs` on that vertex finds exactly that component.

Same argument for the rest.



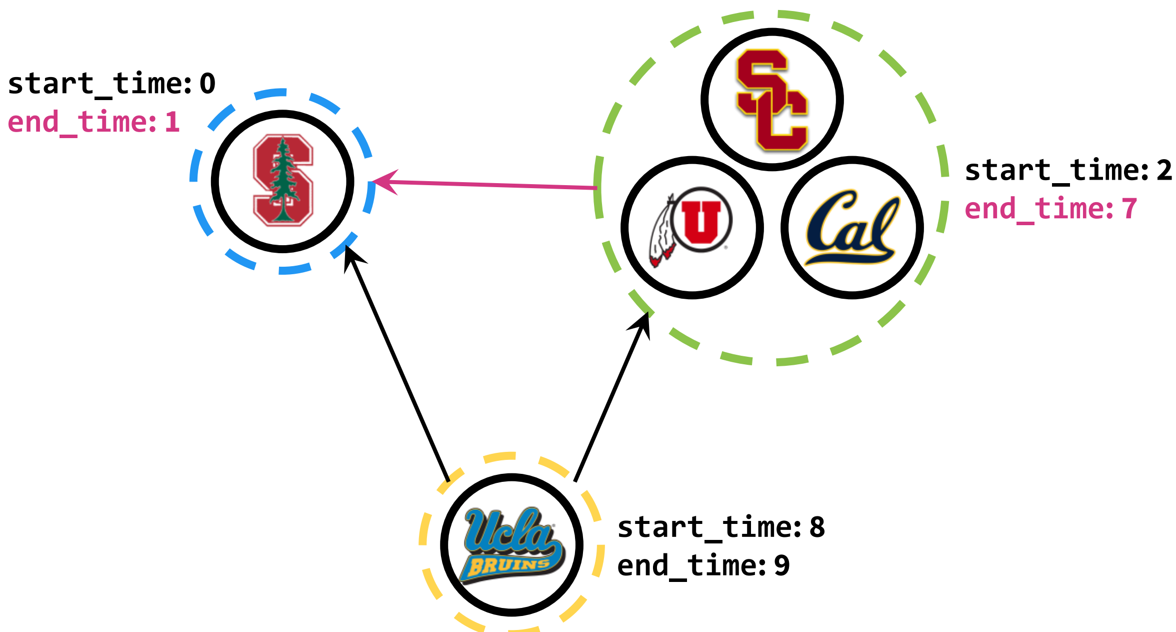
Kosaraju's Algorithm

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



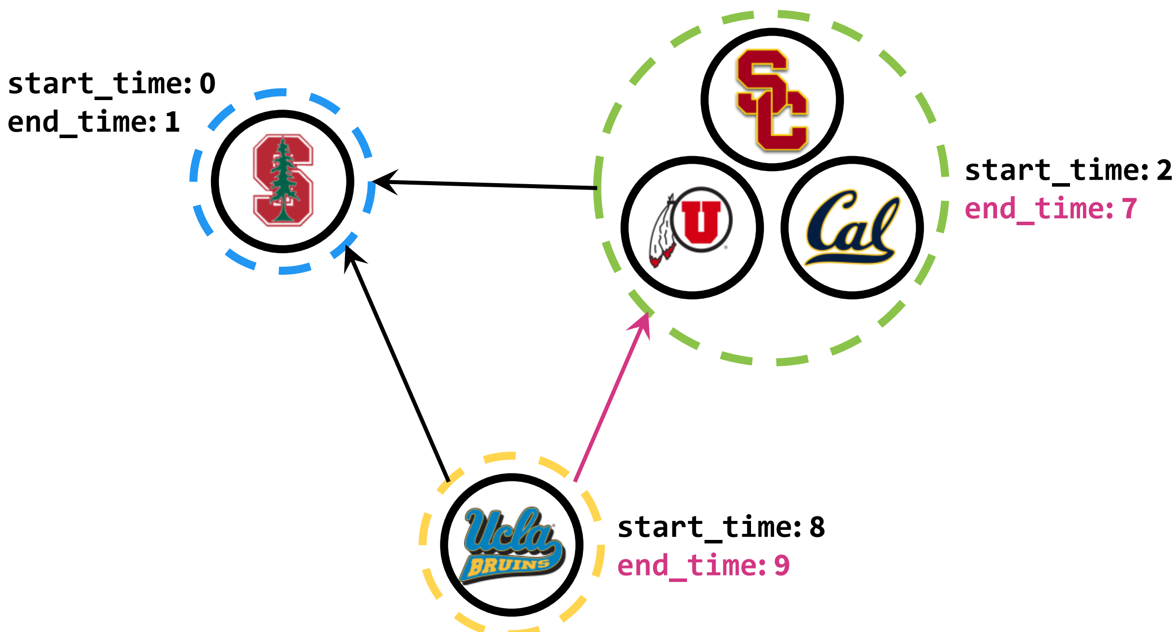
Kosaraju's Algorithm

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



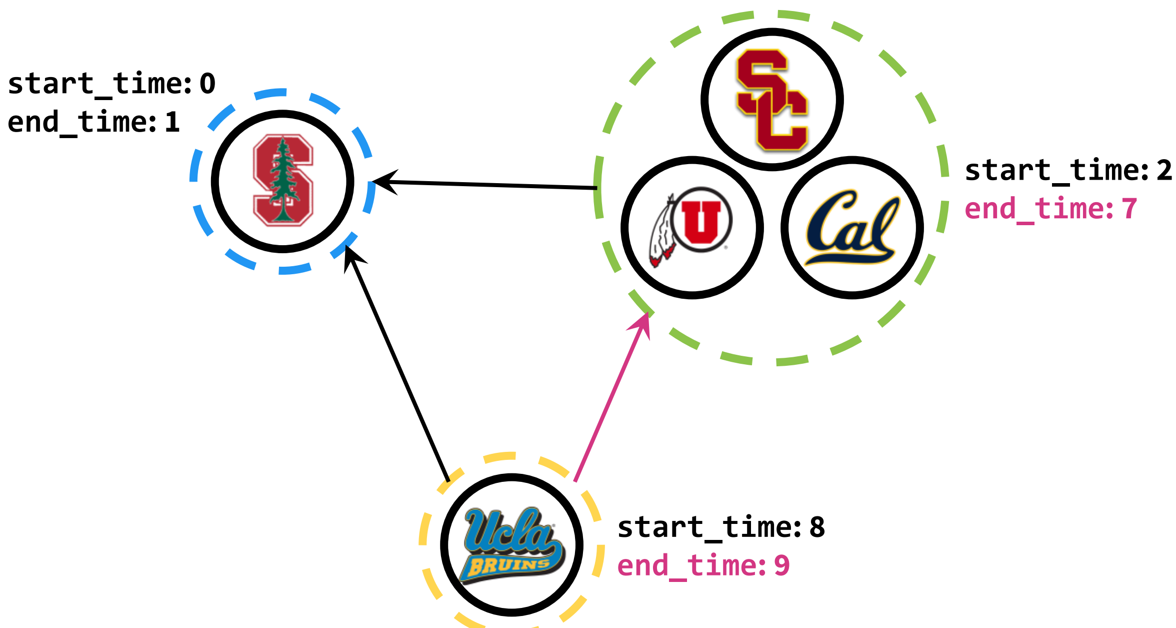
Kosaraju's Algorithm

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.



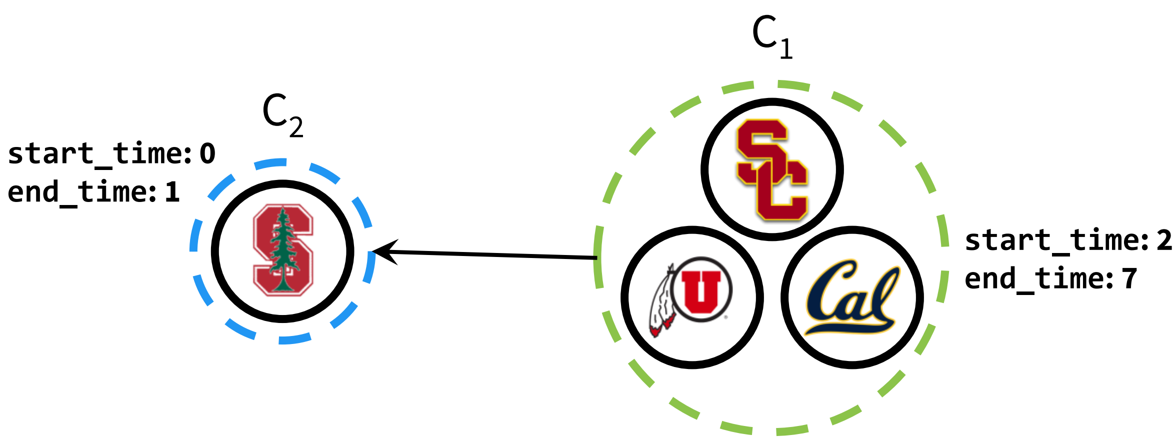
Kosaraju's Algorithm

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

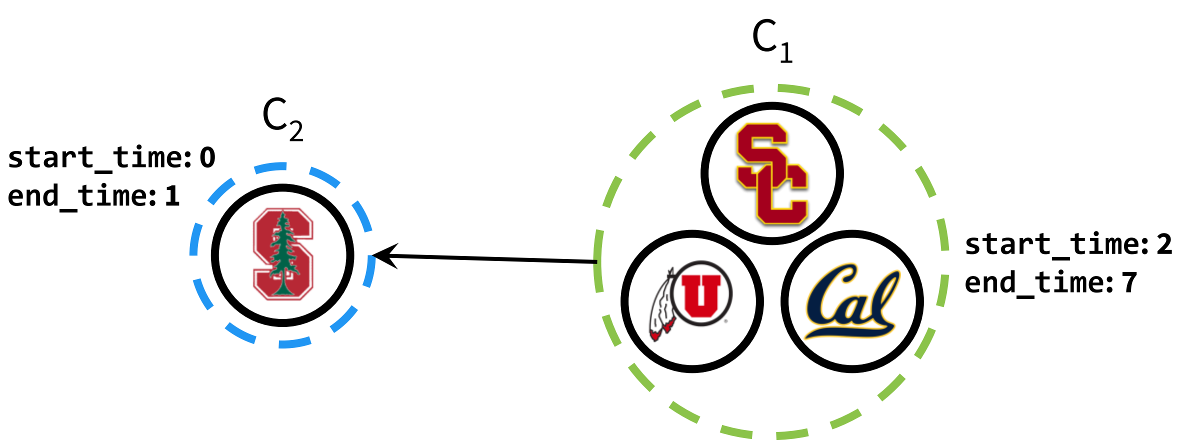
Claim: For each edge (u, v) in the SCC metagraph where u ∊ C_1 and v ∊ C_2, `end_time` of C_1 must be larger than `end_time` of C_2.



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.

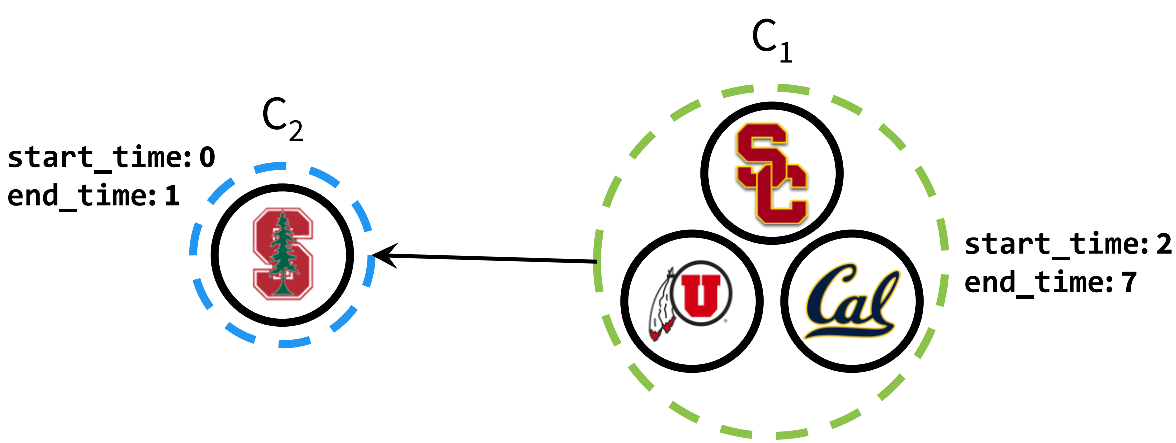


Kosaraju's Algorithm

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.

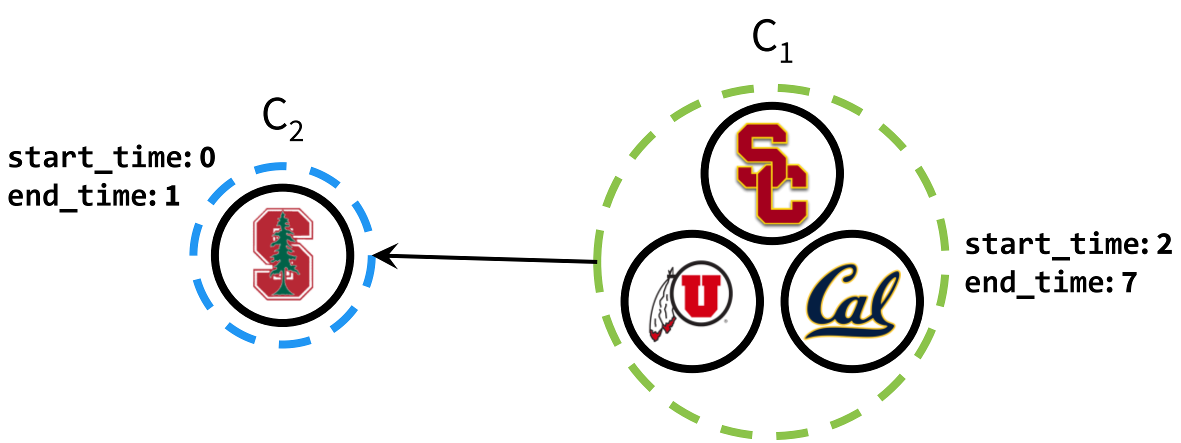


Kosaraju's Algorithm

Claim: For each edge (u, v) in the SCC metagraph where u ∊ 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."

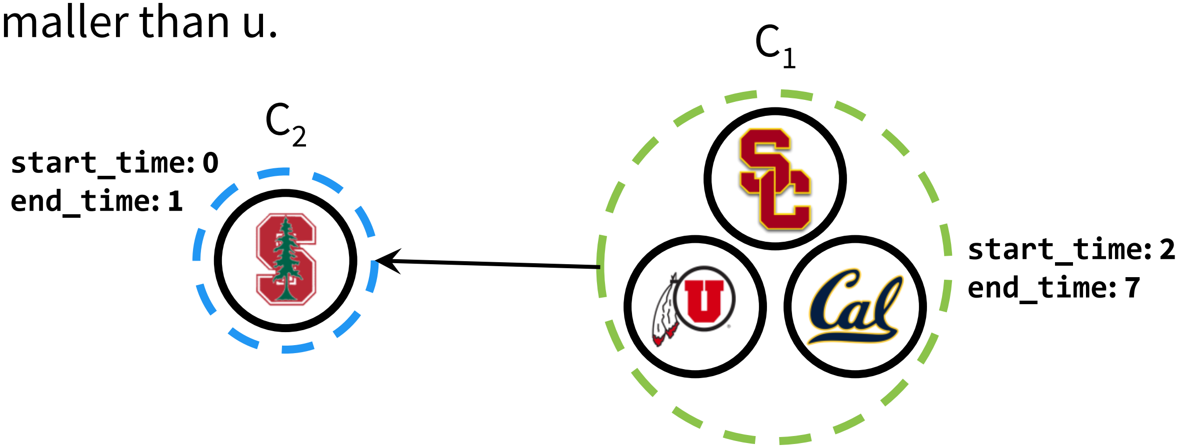


Kosaraju's Algorithm

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.



Kosaraju's Algorithm

The runtime of Kosaraju's algorithm is `O(|V|+|E|)`.

- Runtime for the first DFS is `O(|V|+|E|)`.
- Runtime for reversing the graph is `O(|V|+|E|)`.
- Runtime for the second DFS is `O(|V|+|E|)`.

We can find connected components and SCCs in the same (asymptotic) runtime!

3 min break

Karger's Algorithm

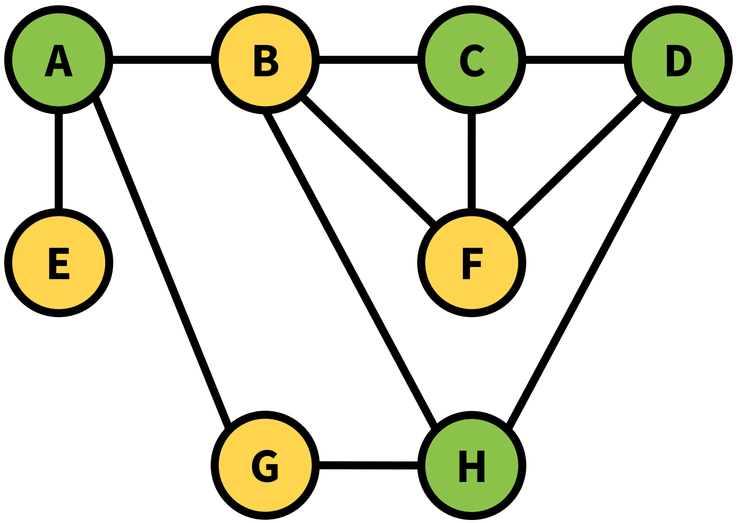
Cuts

A **cut** is a partition of the vertices into two nonempty parts.

Cuts

A **cut** is a partition of the vertices into two nonempty parts.

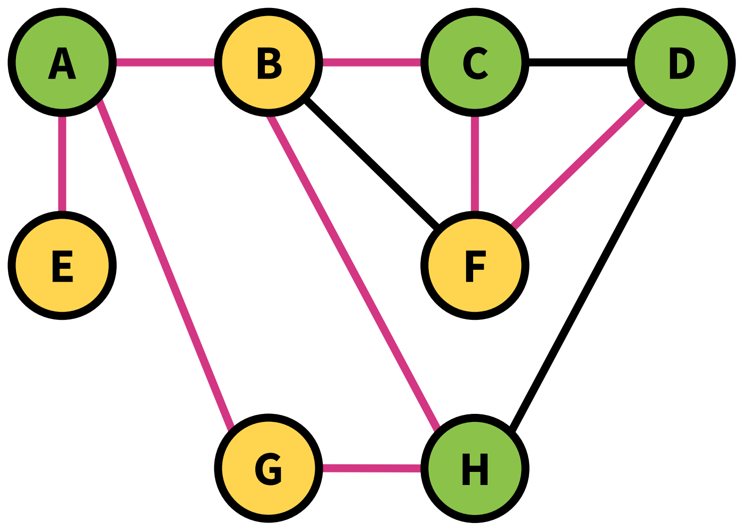
e.g. This is the cut "{A, C, D, H} and {B, E, F, G}".



Cuts

A **cut** is a partition of the vertices into two nonempty parts.

e.g. This is the cut “{A, C, D, H} and {B, E, F, G}”.

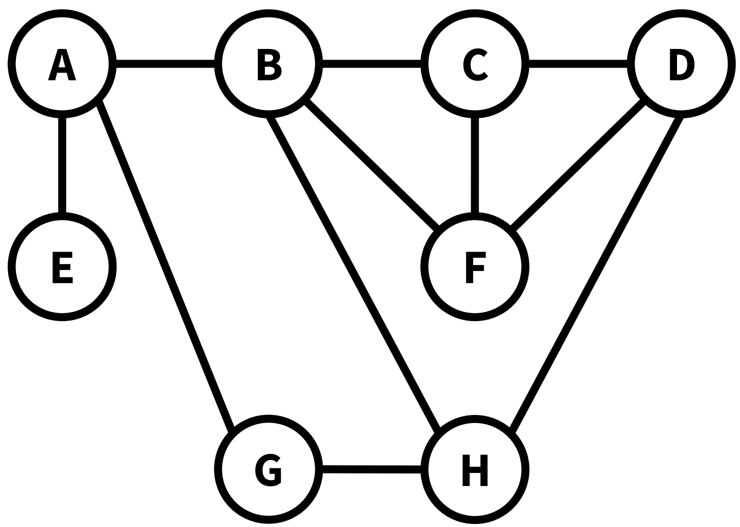


Edges that **cross the cut** go from one part to the other.

e.g. These edges cross the cut.

Cuts

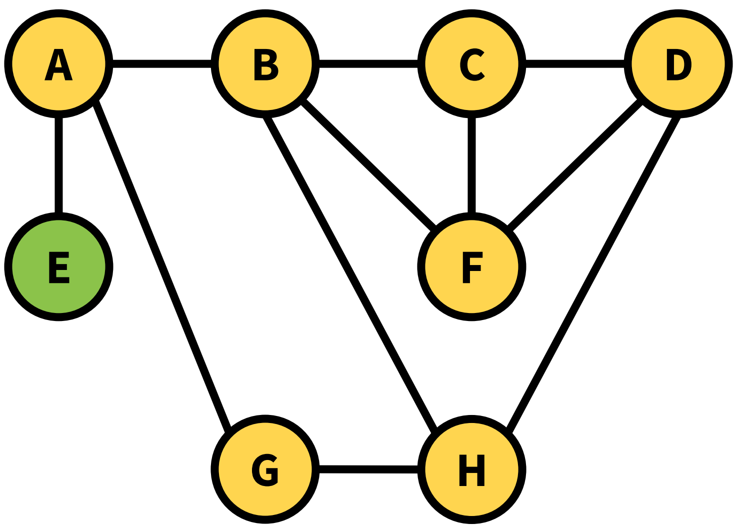
A **global minimum cut** is a cut that has the fewest edges possible crossing it.



Cuts

A **global minimum cut** is a cut that has the fewest edges possible crossing it.

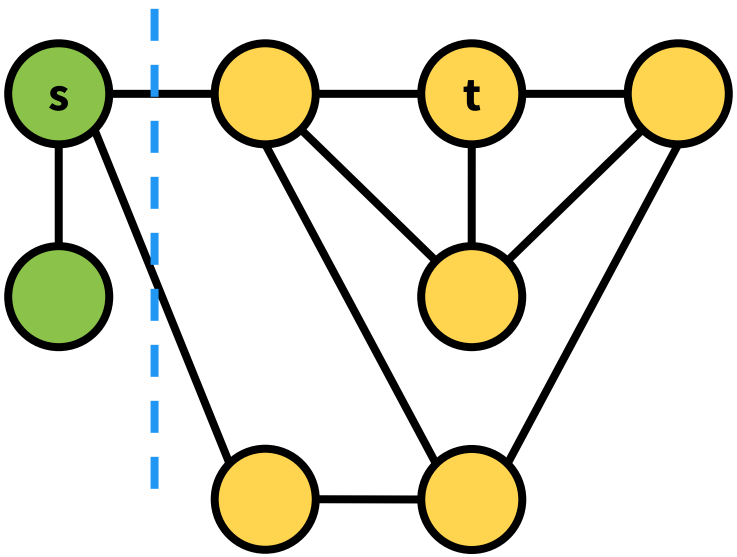
- e.g. The global minimum cut is “{A, B, C, D, F, G, H} and {E}”.



Cuts

Later this quarter, we'll talk about **minimum s-t cuts**, which separate specific vertices **s** and **t**.

e.g. The s-t minimum cut is this cut.



Global Minimum Cuts

Why might we care about global minimum cuts?

Application: Image segmentation

Karger's Algorithm

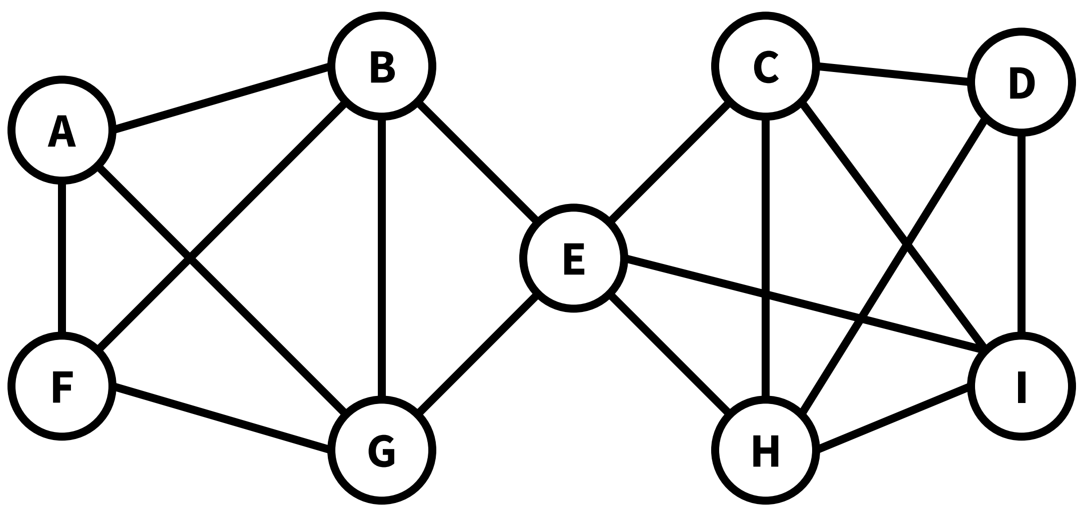
Karger's Algorithm finds global minimum cuts.

- It's a Monte Carlo randomized algorithm! Unlike `quicksort`, which is always correct but sometimes slow, Karger's algorithm is always fast but sometimes Incorrect.
  - For all inputs A, `quicksort` returns a sorted list. For all inputs A, with high probability over the choice of pivots, `quicksort` runs fast.
  - For all inputs G, `karger` runs fast. For all inputs G, with high probability over the randomness in the algorithm, `karger` returns a minimum cut.

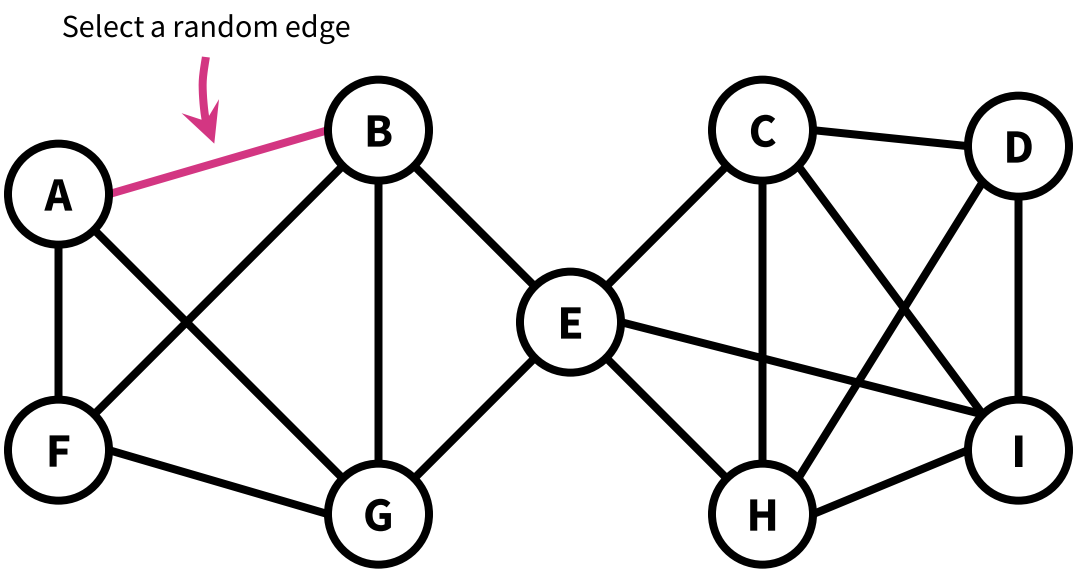
Karger’s Algorithm

The general idea is to pick random edges to “contract” until there are a minimal number of vertices and edges left.

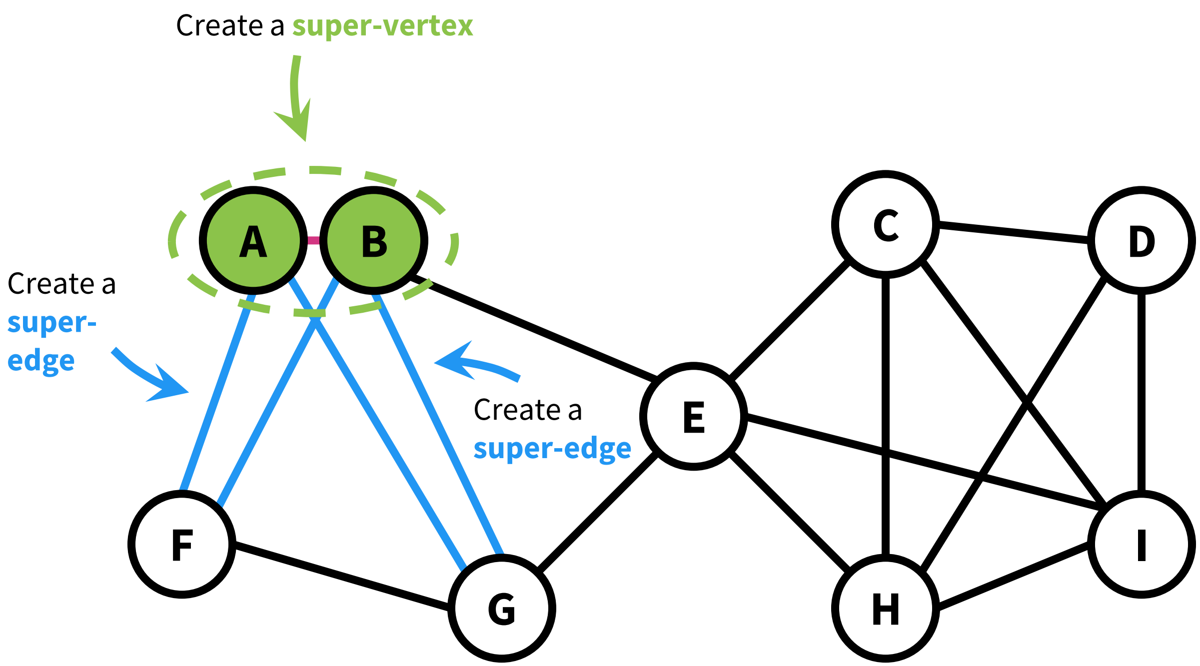
Karger's Algorithm



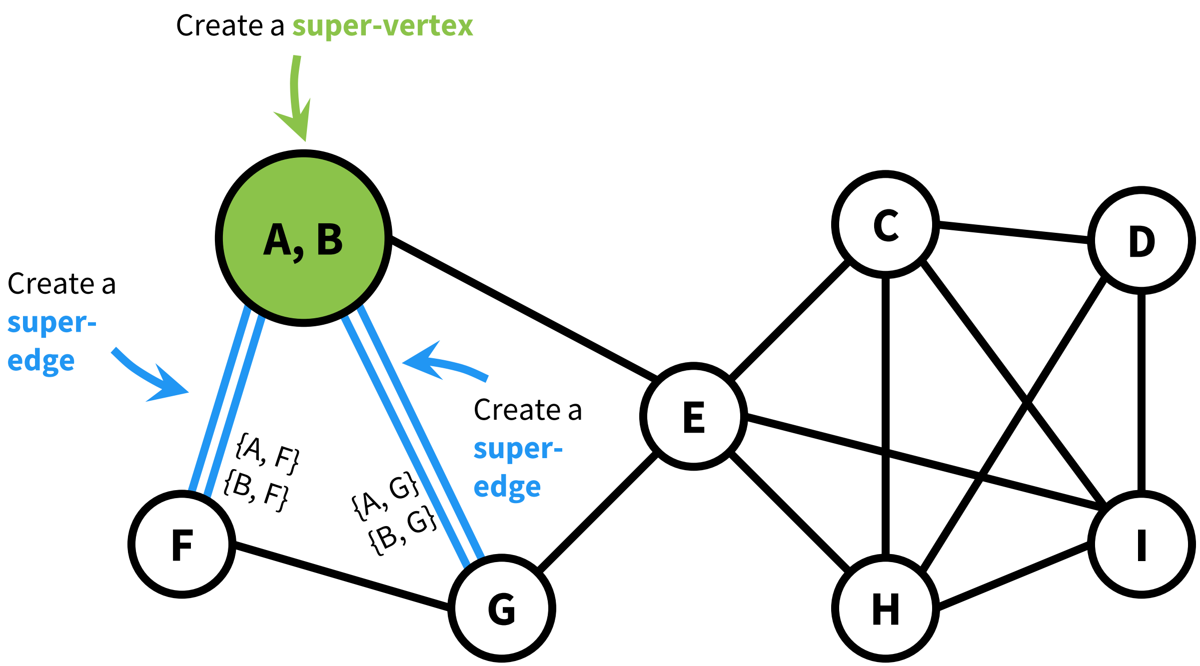
Karger's Algorithm



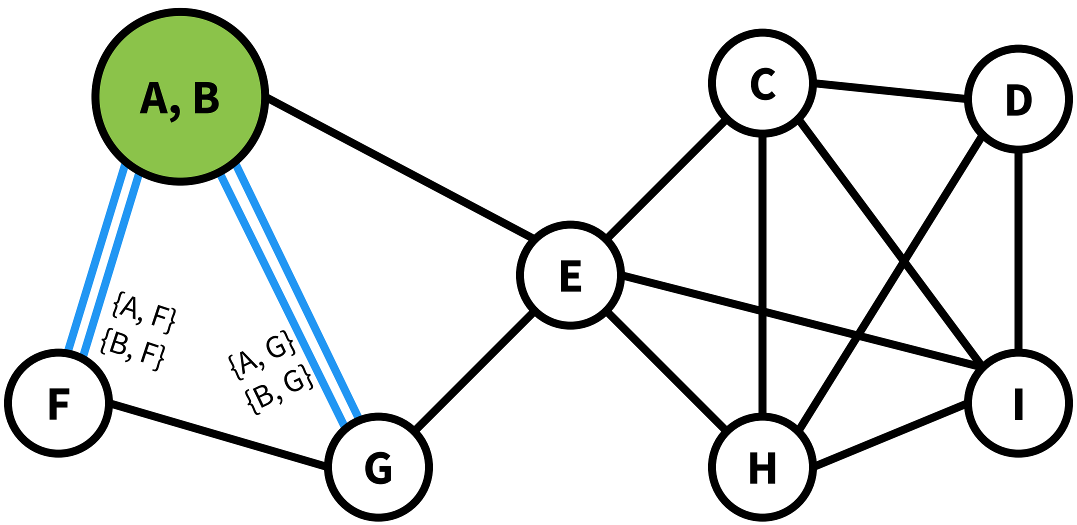
Karger's Algorithm



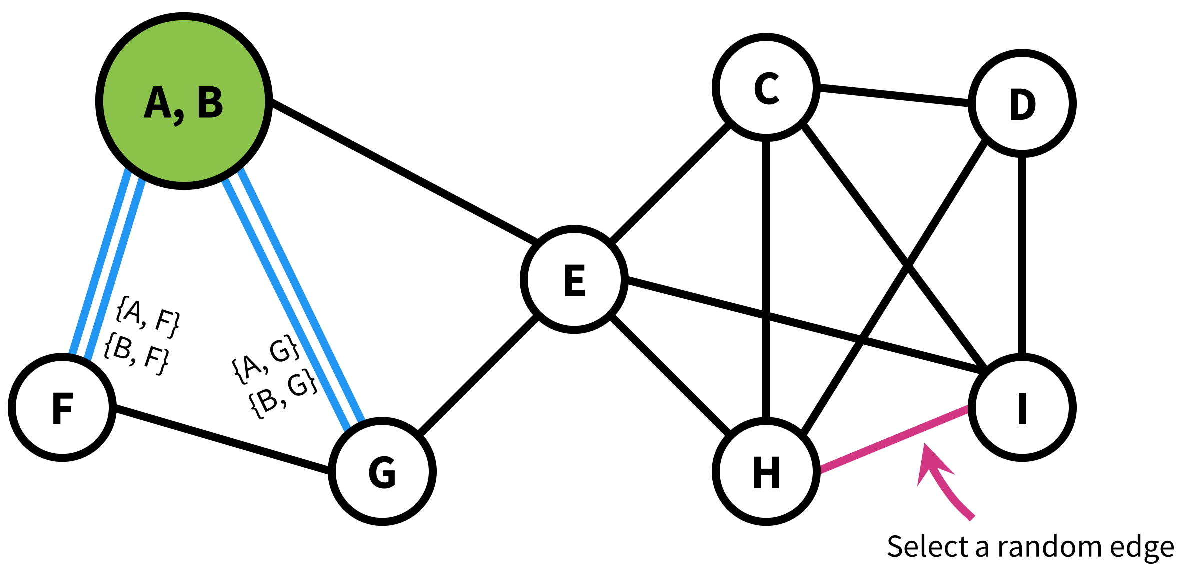
Karger's Algorithm



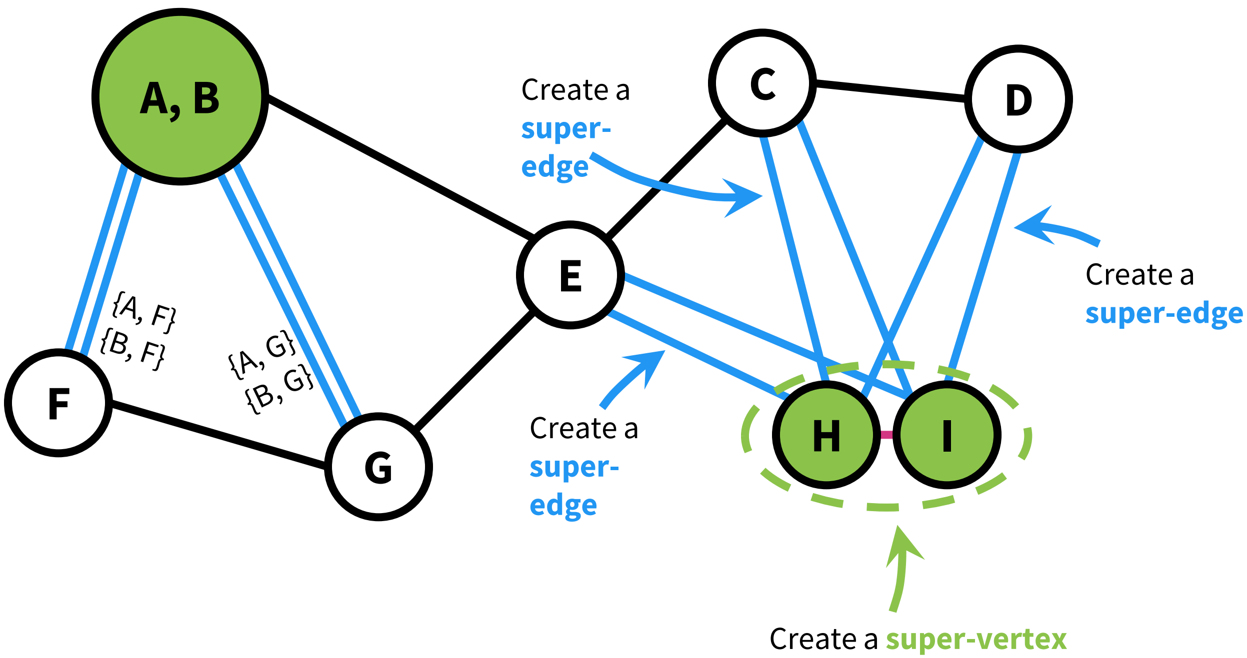
Karger's Algorithm



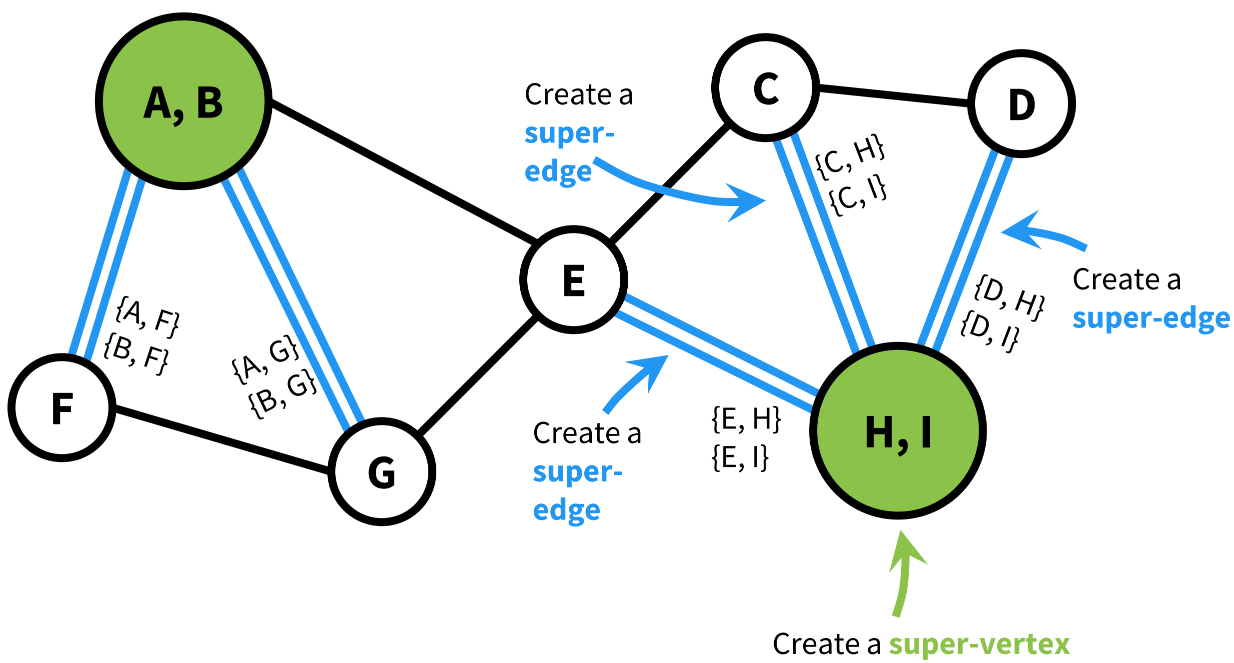
Karger's Algorithm



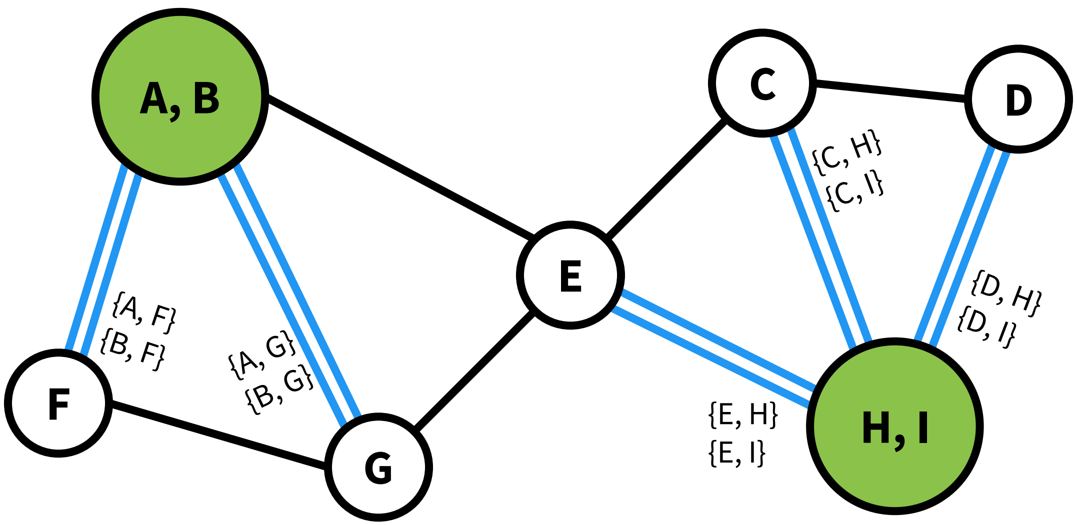
Karger's Algorithm



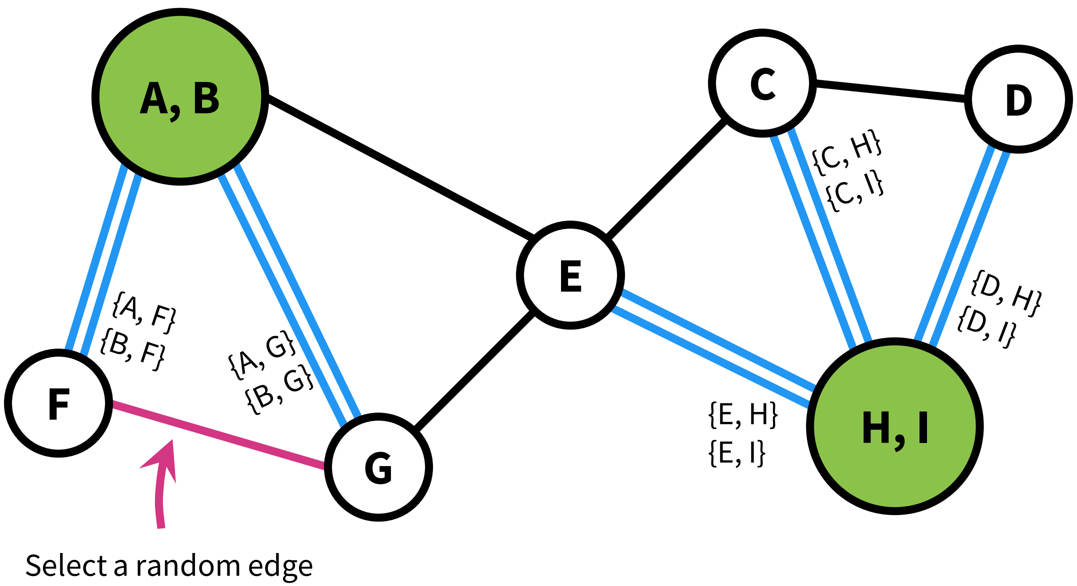
Karger's Algorithm



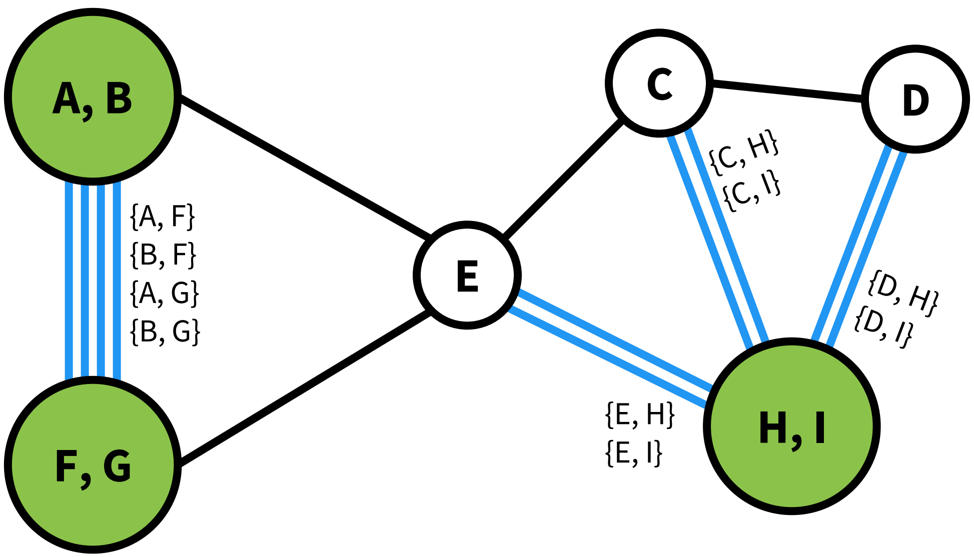
Karger's Algorithm



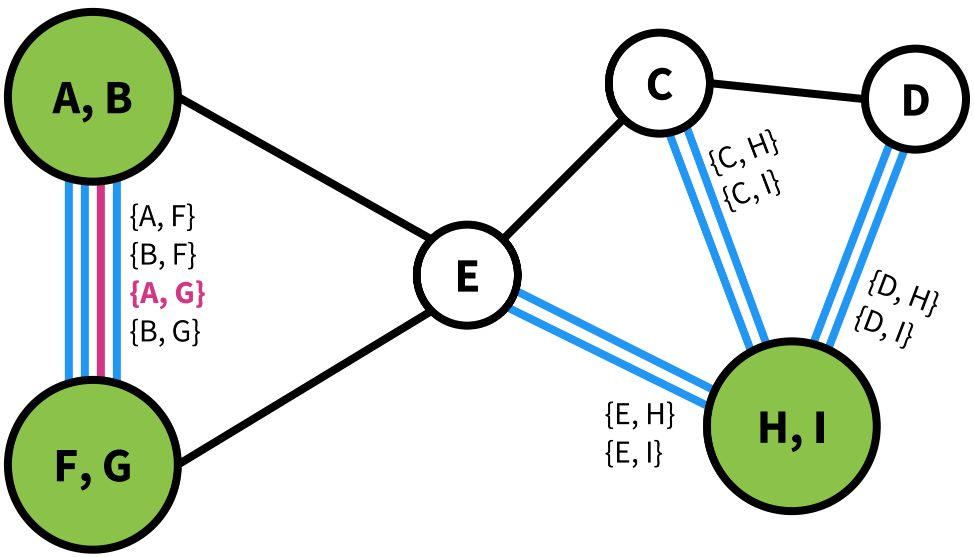
Karger's Algorithm



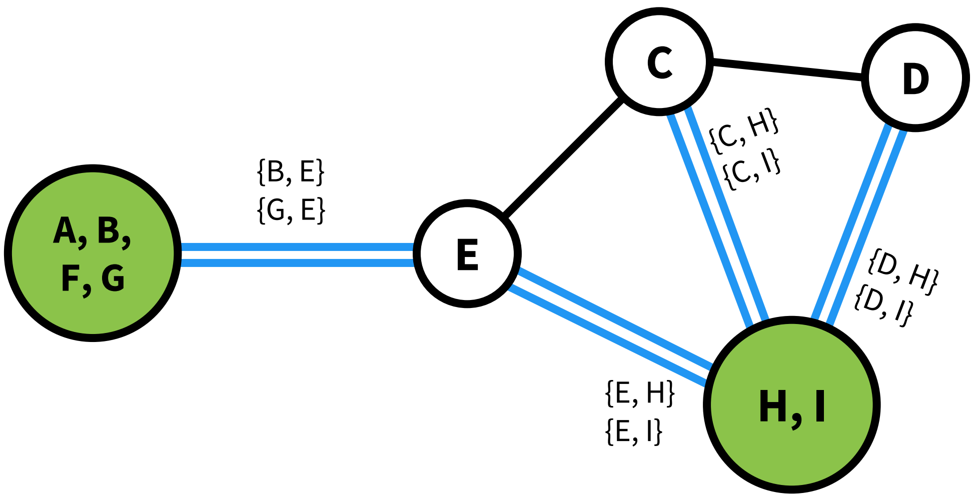
Karger's Algorithm



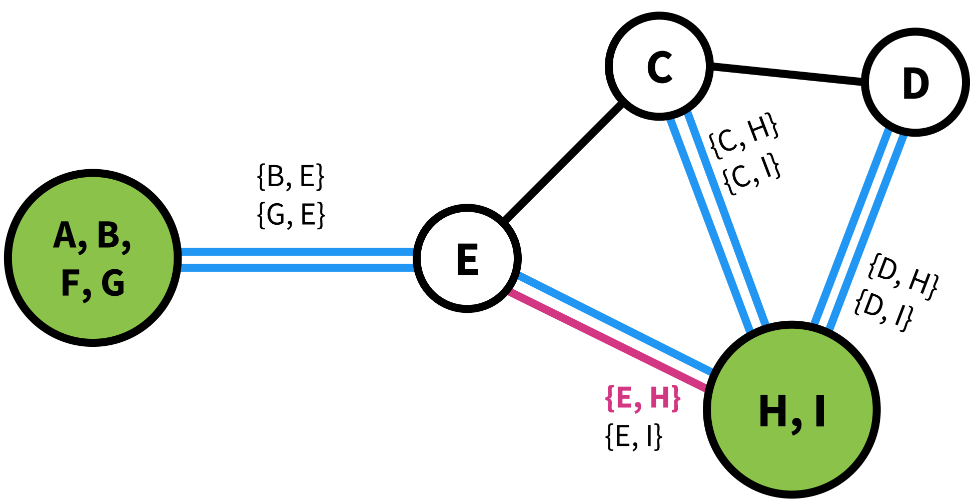
Karger's Algorithm



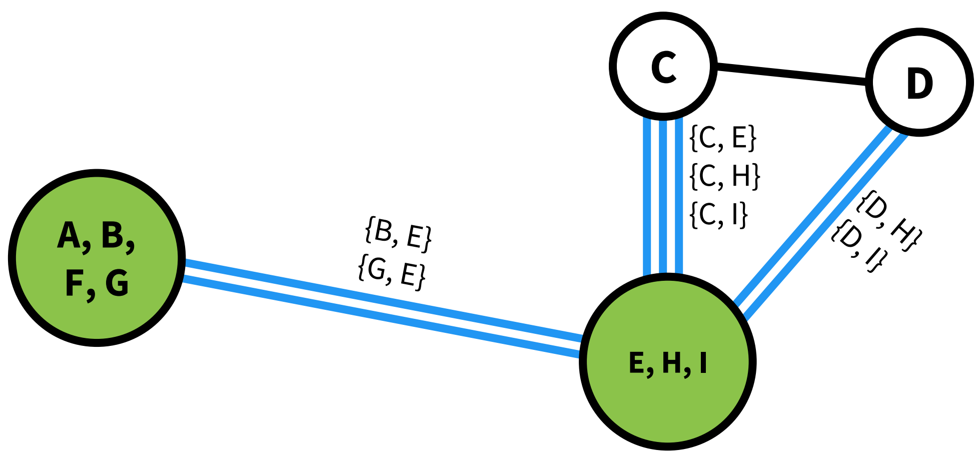
Karger's Algorithm



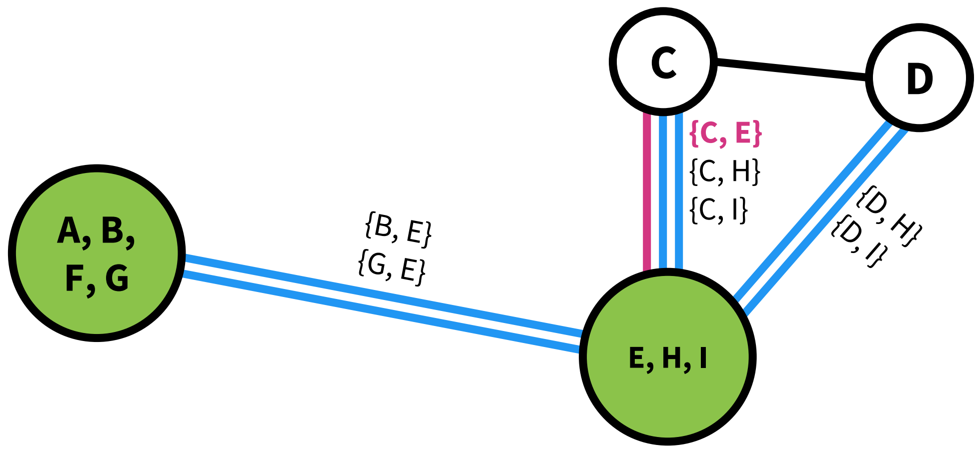
Karger's Algorithm



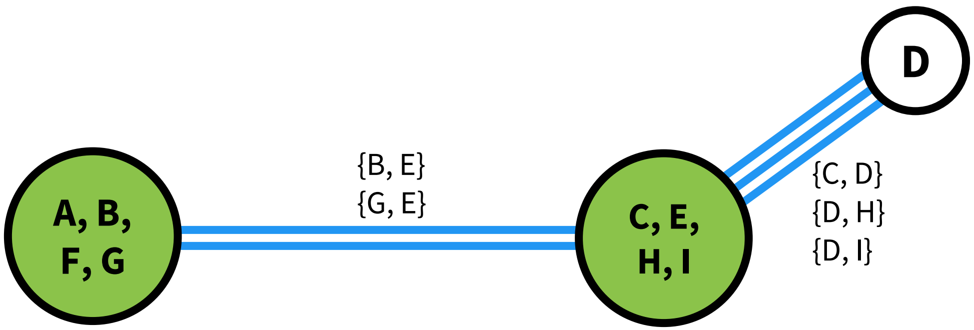
Karger's Algorithm



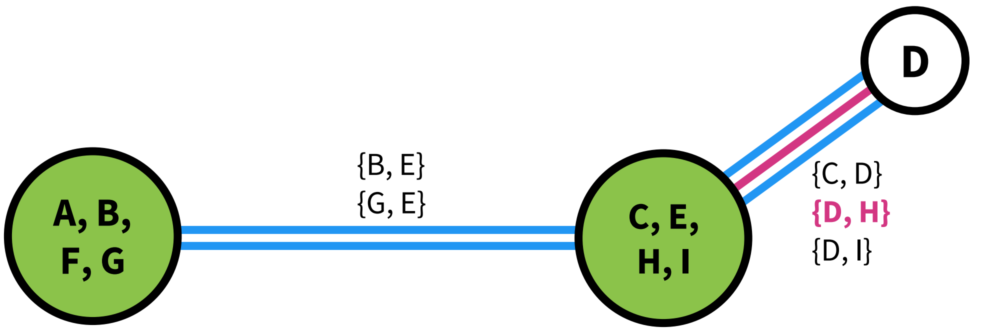
Karger's Algorithm



Karger's Algorithm



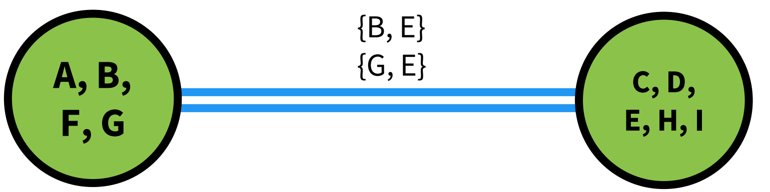
Karger's Algorithm



Karger's Algorithm

The minimum cut is given by the remaining super-vertices.

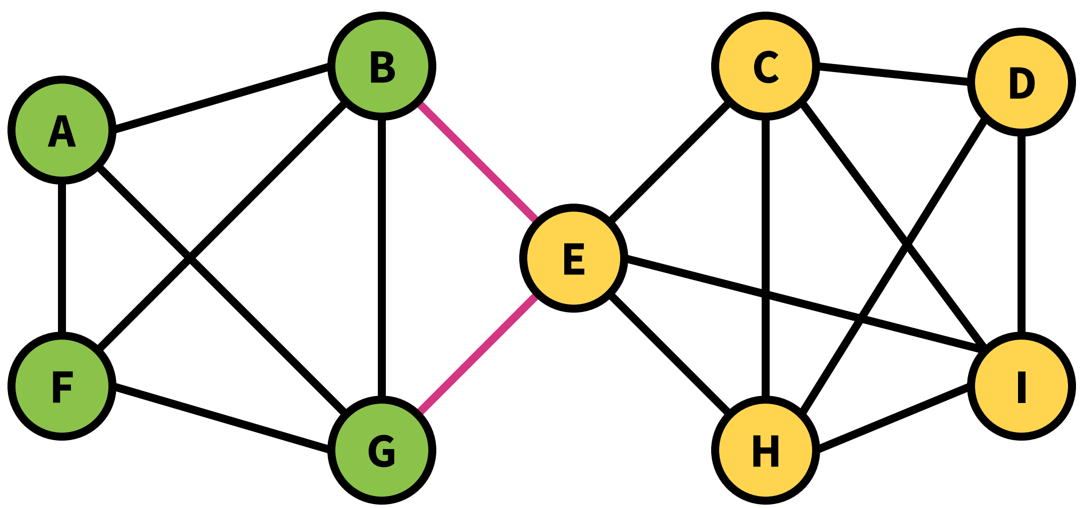
e.g. The cut is "{A, B, F, G} and {C, D, E, H, I}"; the edges that cross this cut are {B, E} and {G, E}.



Karger's Algorithm

The minimum cut is given by the remaining super-vertices.

e.g. The cut is "{A, B, F, G} and {C, D, E, H, I}"; the edges that cross this cut are {B, E} and {G, E}.



Karger's Algorithm

```
algorithm karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_{u'v'} = {(u,v)} for (u,v) E
  E_{u'v'} = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_{u'v'}
  return cut of the remaining super-vertices

algorithm merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_{x'w'} = E_{u'w'} ∪ E_{v'w'}
    Remove u' and v' from G' and add x'
```

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

```
algorithm karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_{u'v'} = {(u,v)} for (u,v) E
  E_{u'v'} = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_{u'v'}
  return cut of the remaining super-vertices

algorithm merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_{x'w'} = E_{u'w'} ∪ E_{v'w'}
    Remove u' and v' from G' and add x'
```

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

The while loop executes |V| - 2 times.

```
algorithm karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

algorithm merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

The while loop executes |V| - 2 times.

```
algorithm karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

algorithm merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

This takes O(|V|).

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

The while loop executes |V| - 2 times.

Removes all edges in the super-edge between super-vertices u' and v'.

```
algorithm karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  Eu'v' = {(u,v)} for (u,v) E
  Eu'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ Eu'v'
  return cut of the remaining super-vertices

algorithm merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    Ex'w' = Eu'w' ∪ Ev'w'
    Remove u' and v' from G' and add x'
```

This takes O(|V|).

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

The while loop executes |V| - 2 times.

Removes all edges in the super-edge between super-vertices u' and v'.

```
algorithm karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

algorithm merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

This takes O(|V|).

u' is the super-vertex containing u; v' is the super-vertex containing v.

Runtime: $O(|V|^2)$

Karger's Algorithm

```
algorithm karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  Eu'v' = {(u,v)} for (u,v) E
  Eu'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ Eu'v'
  return cut of the remaining super-vertices

algorithm merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    Ex'w' = Eu'w' ∪ Ev'w'
    Remove u' and v' from G' and add x'
```

These are the super-edges.

The while loop executes |V| - 2 times.

Removes all edges in the super-edge between super-vertices u' and v'.

This takes O(|V|).

u' is the super-vertex containing u; v' is the super-vertex containing v.

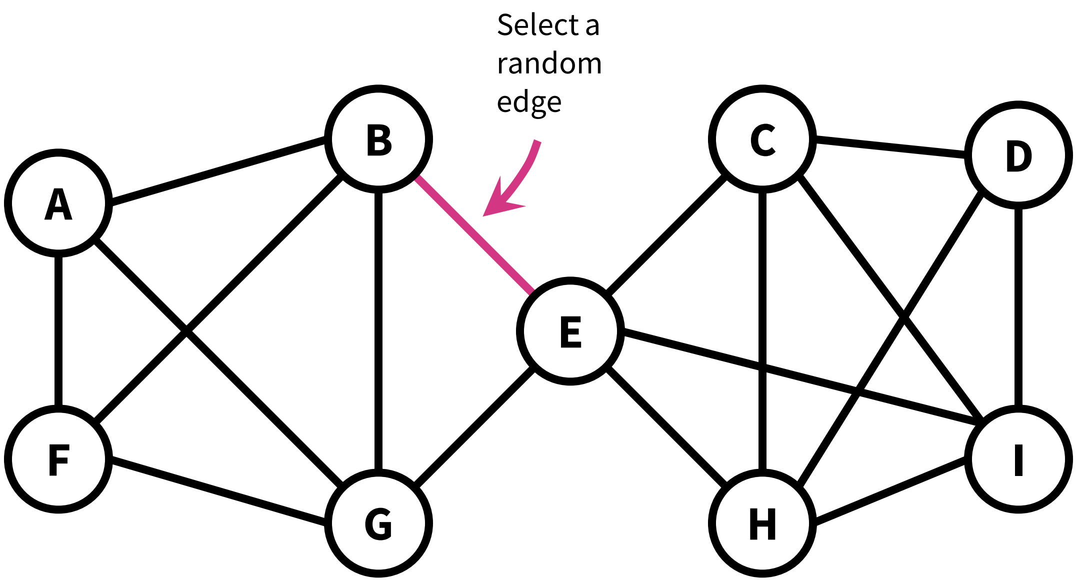
Runtime: $O(|V|^2)$

We can do better with fancy data structures, but this is fine for now.

Karger's Algorithm

We got really lucky!

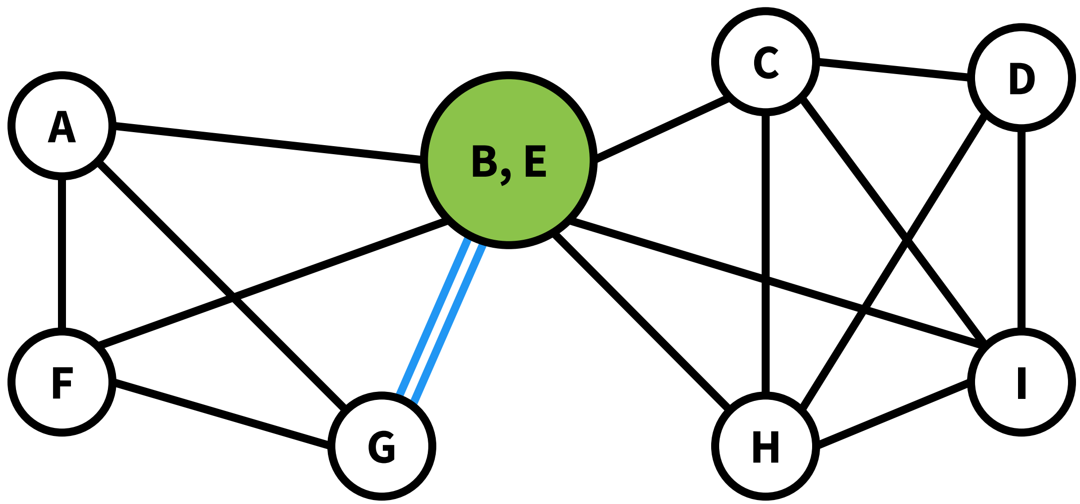
e.g. Suppose we had chosen this edge.



Karger's Algorithm

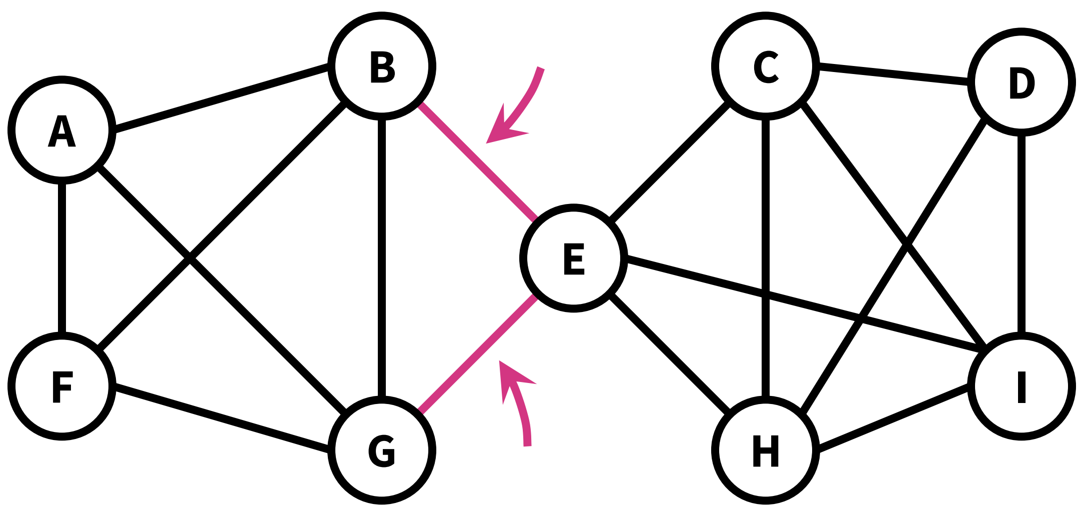
We got really lucky!

- e.g. Suppose we had chosen this edge. Now there's no way to return a cut that separates B and E.



Karger's Algorithm

If fact, if Karger's algorithm ever randomly selects **edges in the min-cut**, then it will be incorrect.



Karger's Algorithm

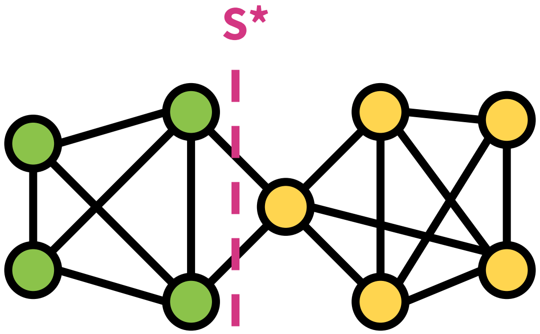
The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof:

Suppose S* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

Then P(karger returns S*) = P(no e_i crosses S*)



Karger's Algorithm

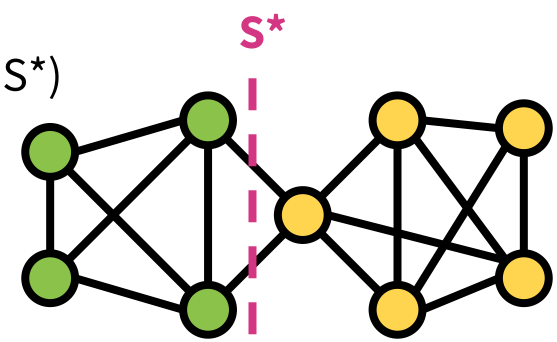
The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof:

Suppose S* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

Then P(karger returns S*) = P(no e_i crosses S*)
= P(e_1 doesn't cross S*)
× P(e_2 doesn't cross S* | e_1 doesn't cross S*)
…
× P(e_{n-2} doesn't cross S* | $e_1, \ldots, e_{n-3}$ doesn't cross S*)



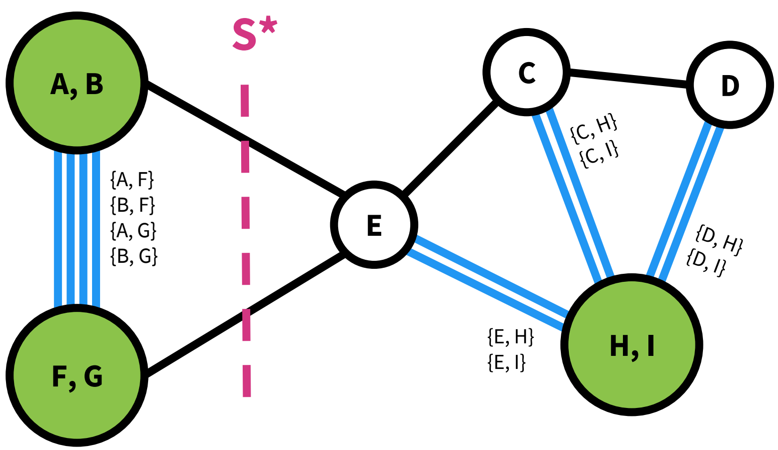
Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?



Karger's Algorithm

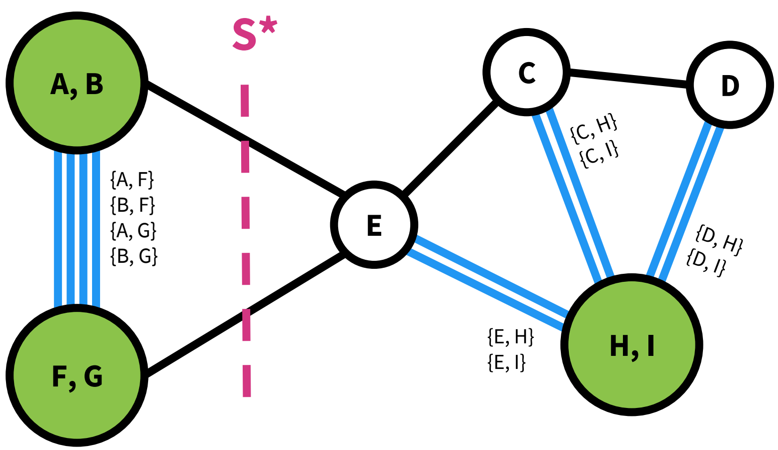
The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?

Suppose there are k edges that cross **S***.



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

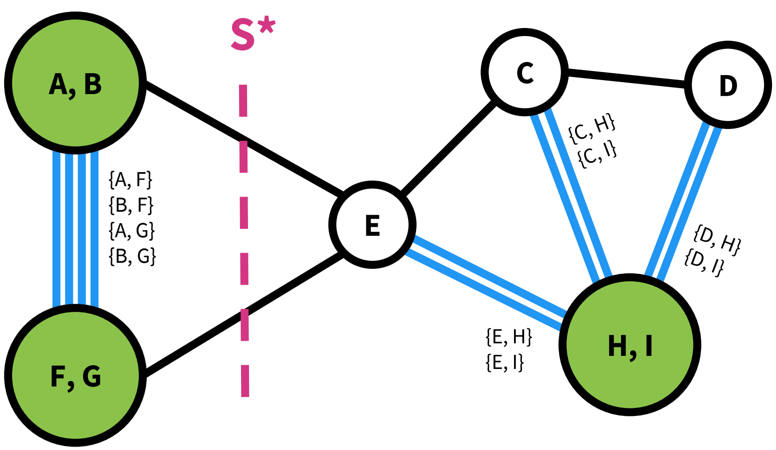
$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?

Suppose there are k edges that cross **S***.

All remaining vertices must have degree at least k (otherwise there would be a smaller cut).



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

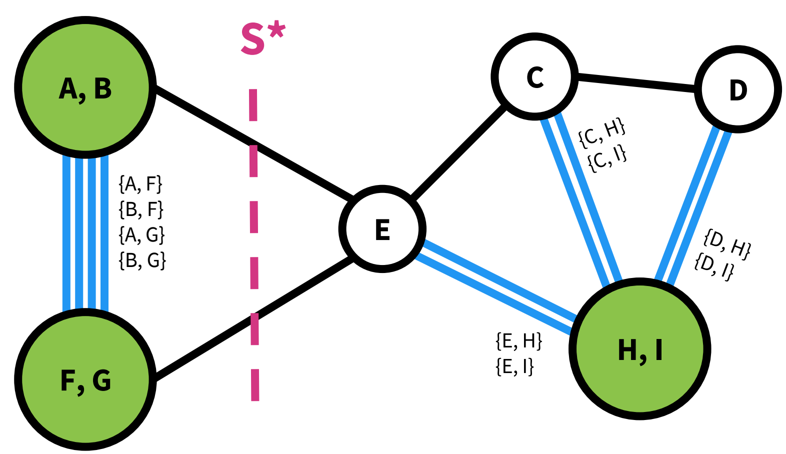
Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?

Suppose there are k edges that cross **S***.

All remaining vertices must have degree at least k (otherwise there would be a smaller cut).

So there are at least **(n-j+1)k/2** total edges.



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?

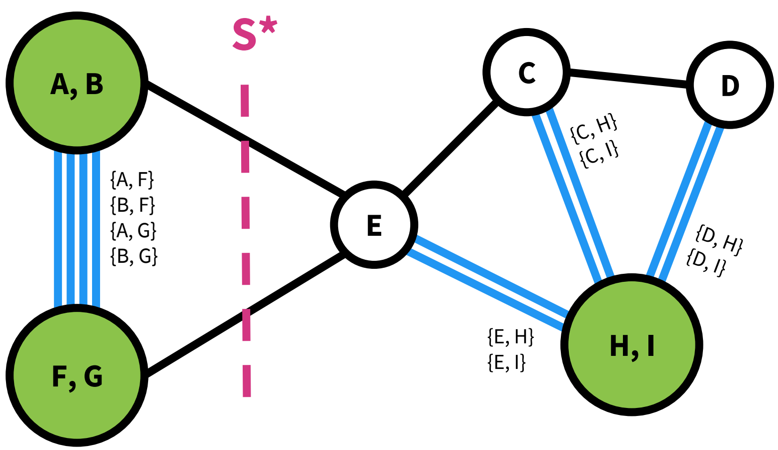
Suppose there are k edges that cross **S***.

All remaining vertices must have degree at least k (otherwise there would be a smaller cut).

So there are at least **(n-j+1)k/2** total edges.

So the probability that `karger` chooses one of the k edges crossing **S*** at step j is at most

$$\frac{k}{\frac{(n-j+1)k}{2}} = \frac{2}{n-j+1}$$



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

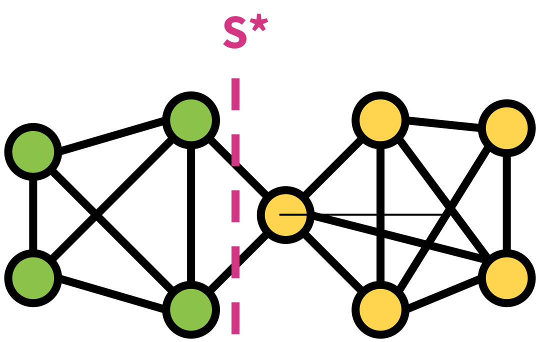
Suppose S^* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

Then P(karger returns S^*) = P(e_1 doesn't cross S^*)

× P(e_2 doesn't cross S^* | e_1 doesn't cross S^*)

…

× P(e_{n-2} doesn't cross S^* | $e_1, \ldots, e_{n-3}$ doesn't cross S^*)



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose S^* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

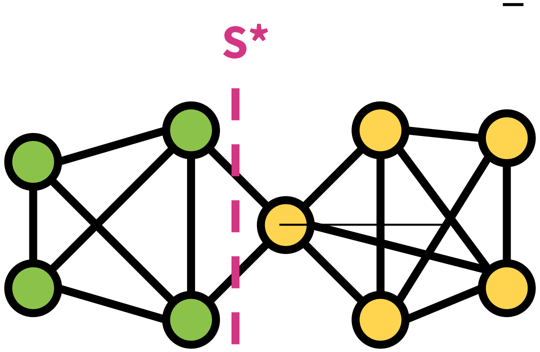
Then P(karger returns S^*) = P(e_1 doesn't cross S^*)

$\times$ P(e_2 doesn't cross S^* | e_1 doesn't cross S^*)

…

$\times$ P(e_{n-2} doesn't cross S^* | $e_1, \ldots, e_{n-3}$ doesn't cross S^*)

$$\geq \frac{(n-2)}{n} \frac{(n-3)}{(n-1)} \frac{(n-4)}{(n-2)} \frac{(n-5)}{(n-3)} \frac{(n-6)}{(n-4)} \cdots \frac{4}{6} \frac{3}{5} \frac{2}{4} \frac{1}{3}$$



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 \Big/ \binom{n}{2}$$

Proof, cont.:

Suppose S^* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

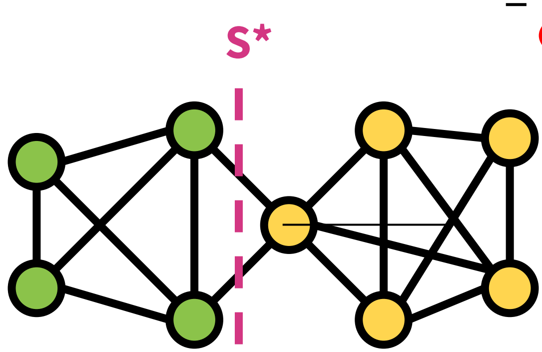
Then P(karger returns S^*) = P(e_1 doesn't cross S^*)

$\times$ P(e_2 doesn't cross S^* | e_1 doesn't cross S^*)

…

$\times$ P(e_{n-2} doesn't cross S^* | $e_1, \ldots, e_{n-3}$ doesn't cross S^*)

$$\geq \frac{(n-2)}{n} \; \frac{(n-3)}{(n-1)} \; \frac{(n-4)}{(n-2)} \; \frac{(n-5)}{(n-3)} \; \frac{(n-6)}{(n-4)} \cdots \frac{4}{6} \; \frac{3}{5} \; \frac{2}{4} \; \frac{1}{3}$$



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

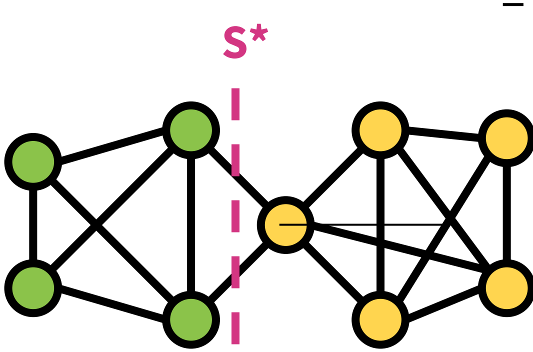
$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose S* is a min-cut and suppose we select edges e_1, e_2, …, e_{n-2}.

Then P(karger returns S*) = P(e_1 doesn't cross S*)
× P(e_2 doesn't cross S* | e_1 doesn't cross S*)
…
× P(e_{n-2} doesn't cross S* | e_1, …, e_{n-3} doesn't cross S*)

$$\geq \frac{(n-2)}{n}\;\frac{(n-3)}{(n-1)}\;\frac{(n-4)}{(n-2)}\;\frac{(n-5)}{(n-3)}\;\frac{(n-6)}{(n-4)}\;\cdots\;\frac{4}{6}\;\frac{3}{5}\;\frac{2}{4}\;\frac{1}{3}$$



= 2/(n(n-1))

= 1/(nC2)

Karger's Algorithm

1/(nC2) isn't all that great …

- For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

Karger's Algorithm

1/(nC2) isn't all that great …

- For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

- How many times T do we need to repeat `karger` to obtain this probability?

Karger's Algorithm

1/(nC2) isn't all that great …

- For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

- How many times T do we need to repeat `karger` to obtain this probability?
- Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
  P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)

Karger's Algorithm

1/(nC2) isn't all that great …

- For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

- How many times T do we need to repeat `karger` to obtain this probability?
- Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
  P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)
- P(find the min-cut after T times) ≥ 0.9

Karger's Algorithm

1/(nC2) isn't all that great …

- For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

- How many times T do we need to repeat `karger` to obtain this probability?
- Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
  P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)
- P(find the min-cut after T times) ≥ 0.9
  ⇔ P(don't find the min-cut after T times) ≤ 0.1.

Karger's Algorithm

1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)

P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$

Karger's Algorithm

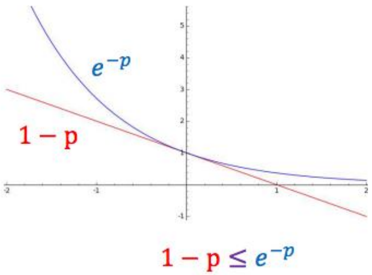
1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)



P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$

Karger's Algorithm

1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

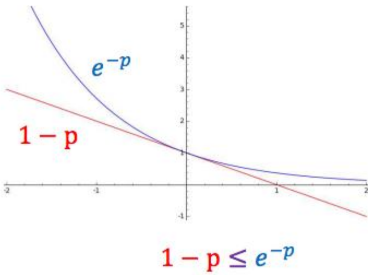
Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)

P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$

T = (nC2) ln (1/0.1) times



Karger's Algorithm

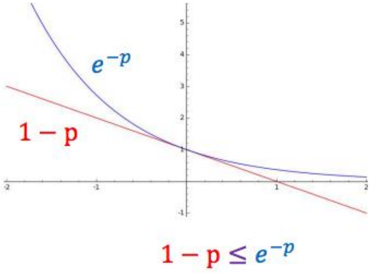
1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)



P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$

T = (nC2) ln (1/0.1) times

Suppose we want to find the min-cut with probability p.

Karger's Algorithm

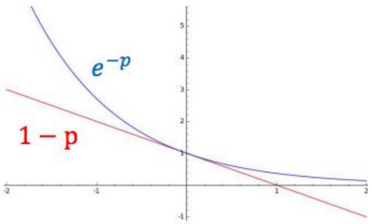
1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)



$1-p \leq e^{-p}$

P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$
T = (nC2) ln (1/0.1) times

Suppose we want to find the min-cut with probability p.
Then we must repeat karger **T = (nC2) ln (1/(1-p)) times**.

Karger's Algorithm

T = (nC2) ln (1/(1-p)) times = `O(|V|²)` times, so the overall runtime is `O(|V|⁴)`.

- Treating 1-p as a constant.
- If we use union-find data structures, then we can do better.
- This might seem lousy, but then consider that enumerating over all possible cuts to find the min-cut requires $O(2^{|V|})$.
- This is a huge improvement!

Karger's Algorithm

```
algorithm karger_loop(G=(V,E), threshold):
  cur_min_cut = None
  n = V.length, p = threshold
  for t = 1 to (nC2)ln(1/(1-p)) :
    candidate_cut = karger(G)
    if candidate_cut.size < cur_min_cut.size:
      cur_min_cut = candidate_cut
  return cur_min_cut
```

Runtime: $O(|V|^4)$

Karger's Algorithm

Upshot: Whenever we have a Monte-Carlo algorithm with a small probability of success, we can boost the probability of success by repeating it a bunch of times and taking the best solution!