Graph Algorithms I

Graph Algorithms II

Outline for Today

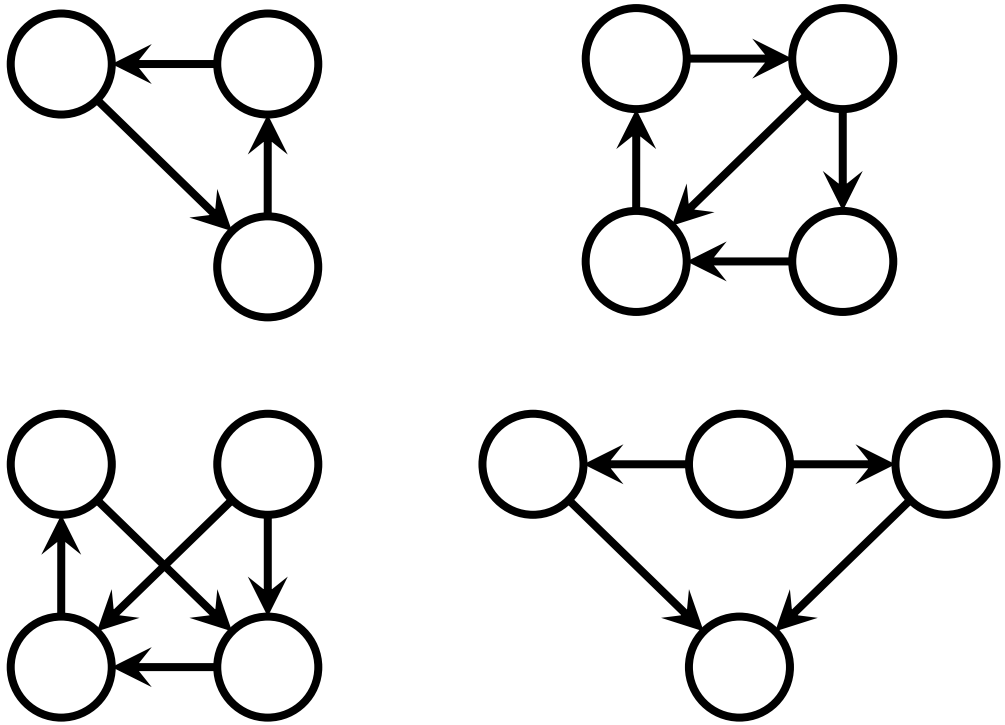
Graph algorithms

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

Kosaraju’s Algorithm

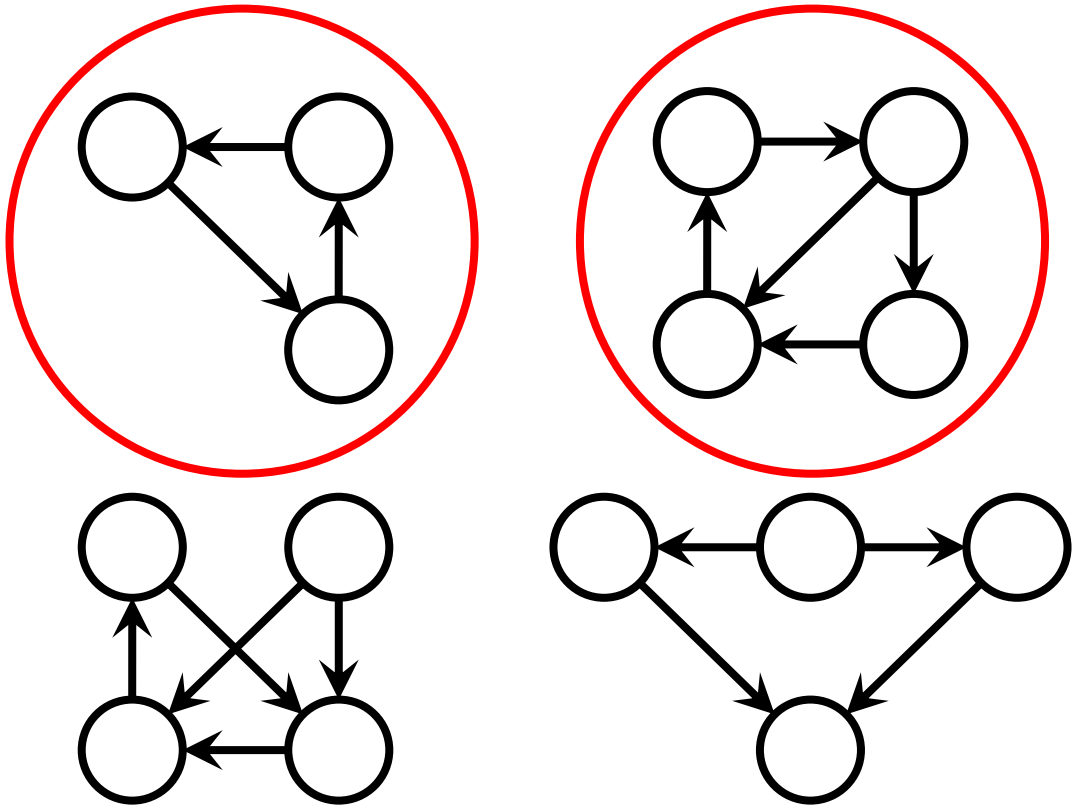
Strongly Connected Components

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.



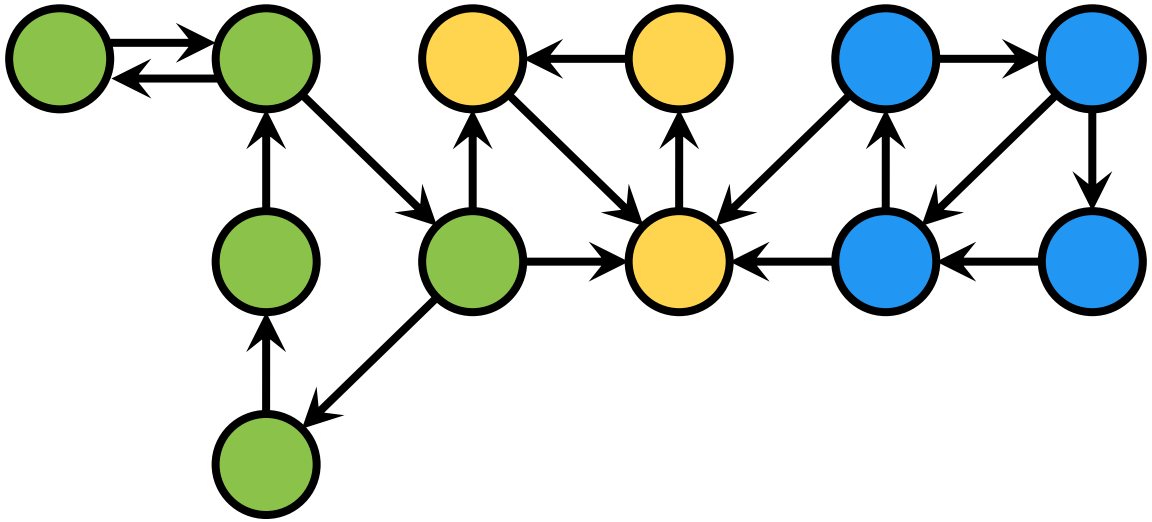
Strongly Connected Components

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.

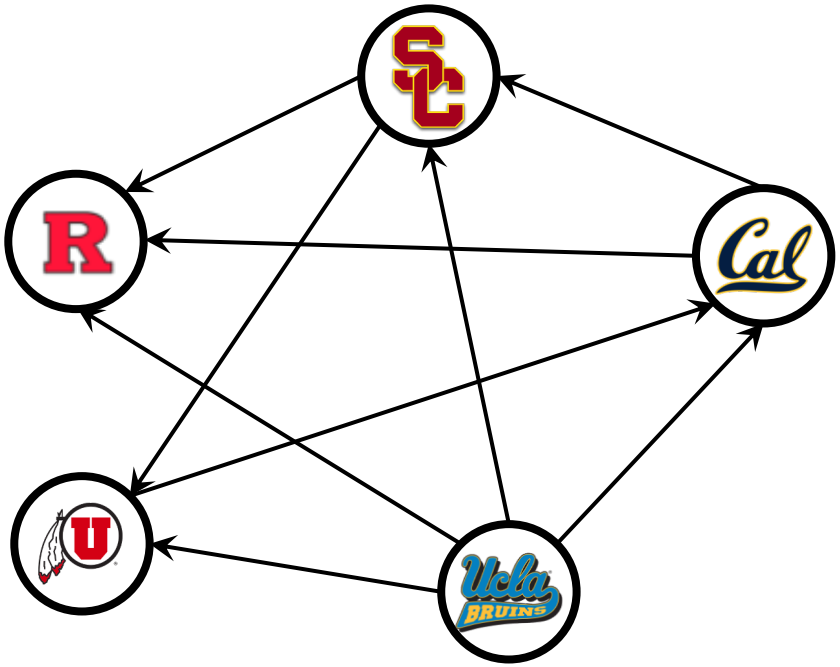
A computer scientist might want to decompose the Internet into SCCs to find connected areas.

An economist might want to decompose labor market data (e.g. employer-employee graph) into SCCs to analyze labor status.

A football executive might want to determine which schools should play in NCAA.

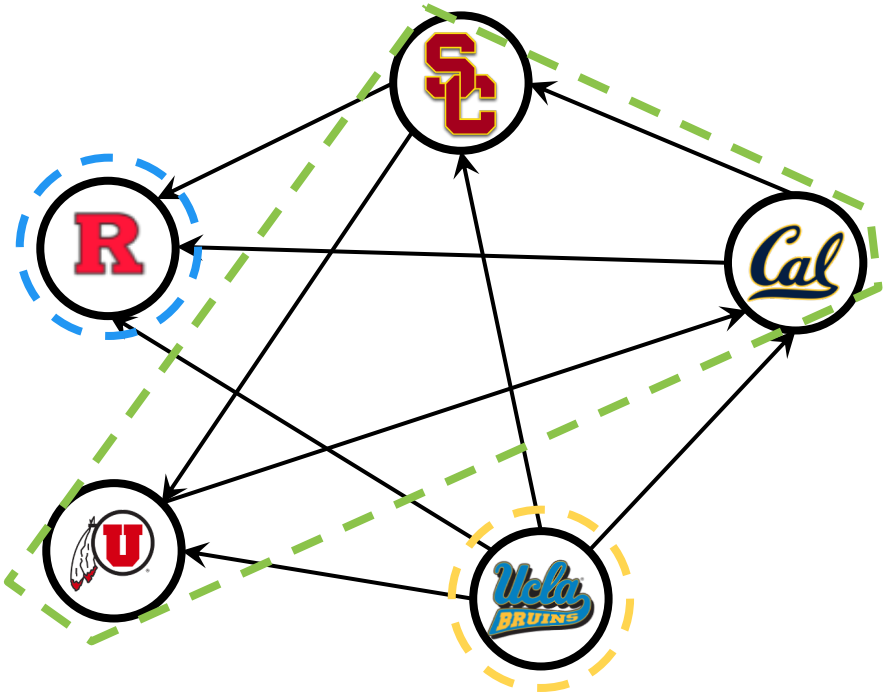
Strongly Connected Components

How many SCCs are in this graph? 🤔



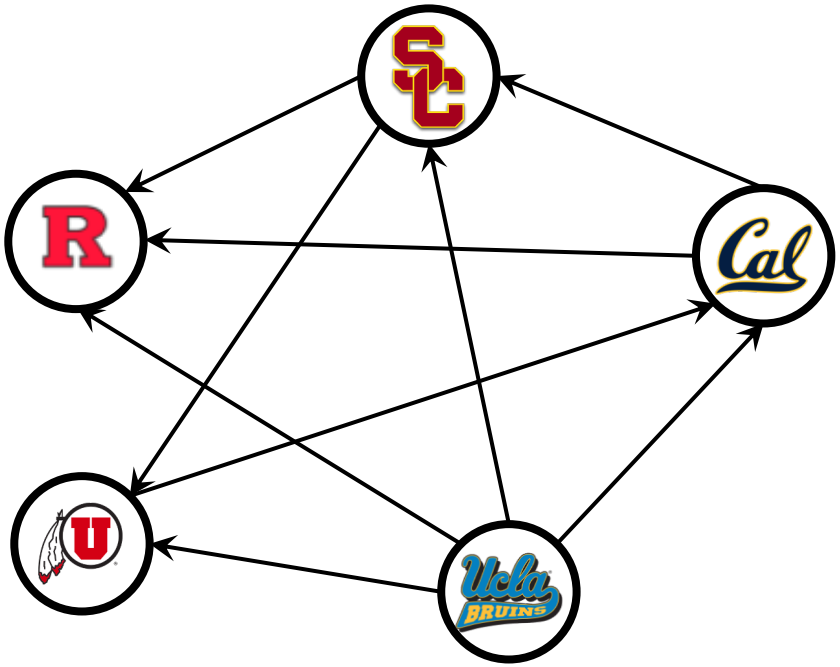
Strongly Connected Components

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



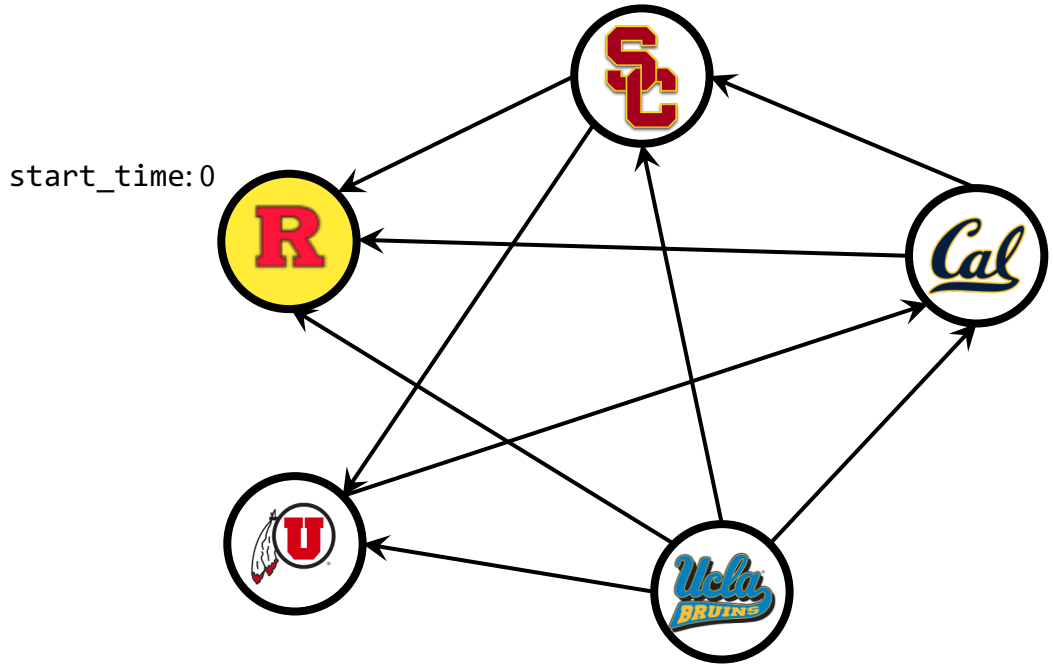
Kosaraju's Algorithm

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



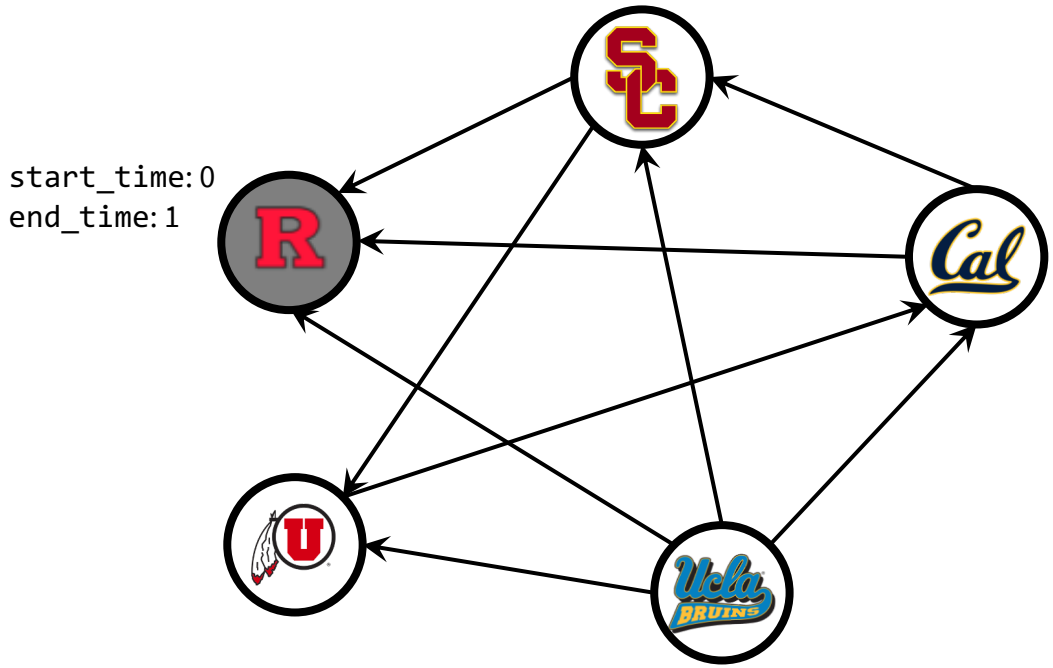
Kosaraju's Algorithm

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



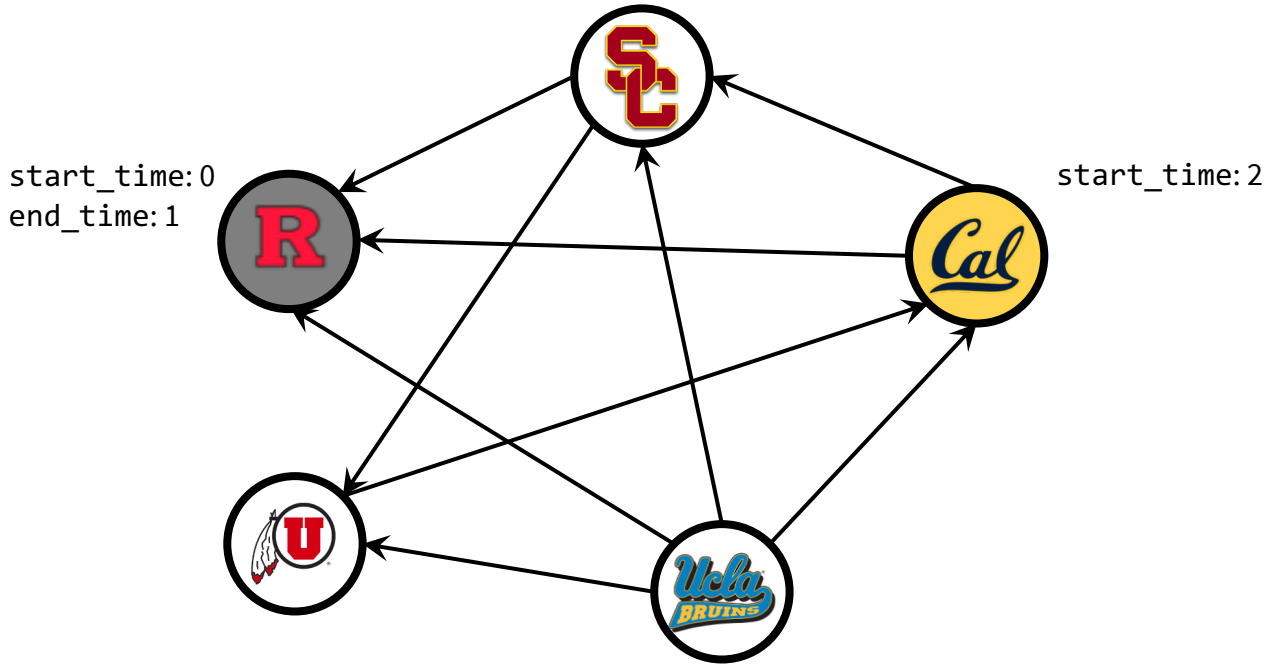
Kosaraju's Algorithm

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



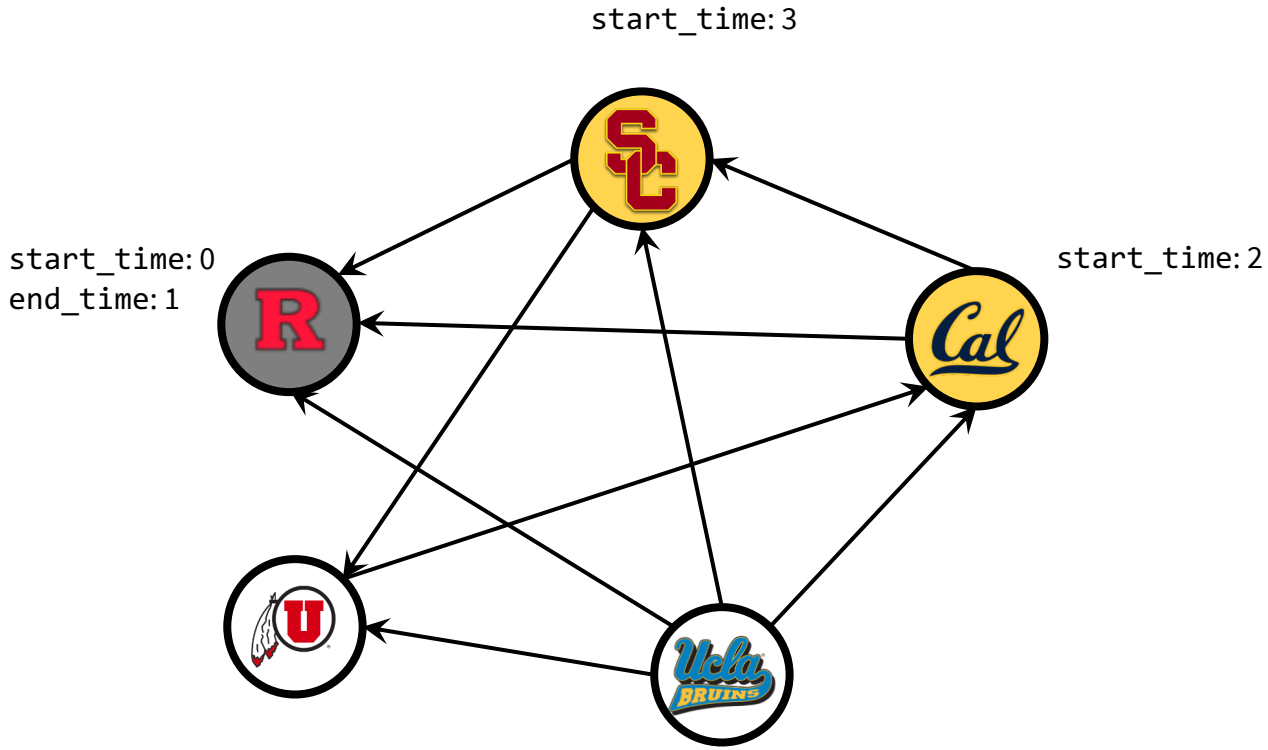
Kosaraju's Algorithm

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



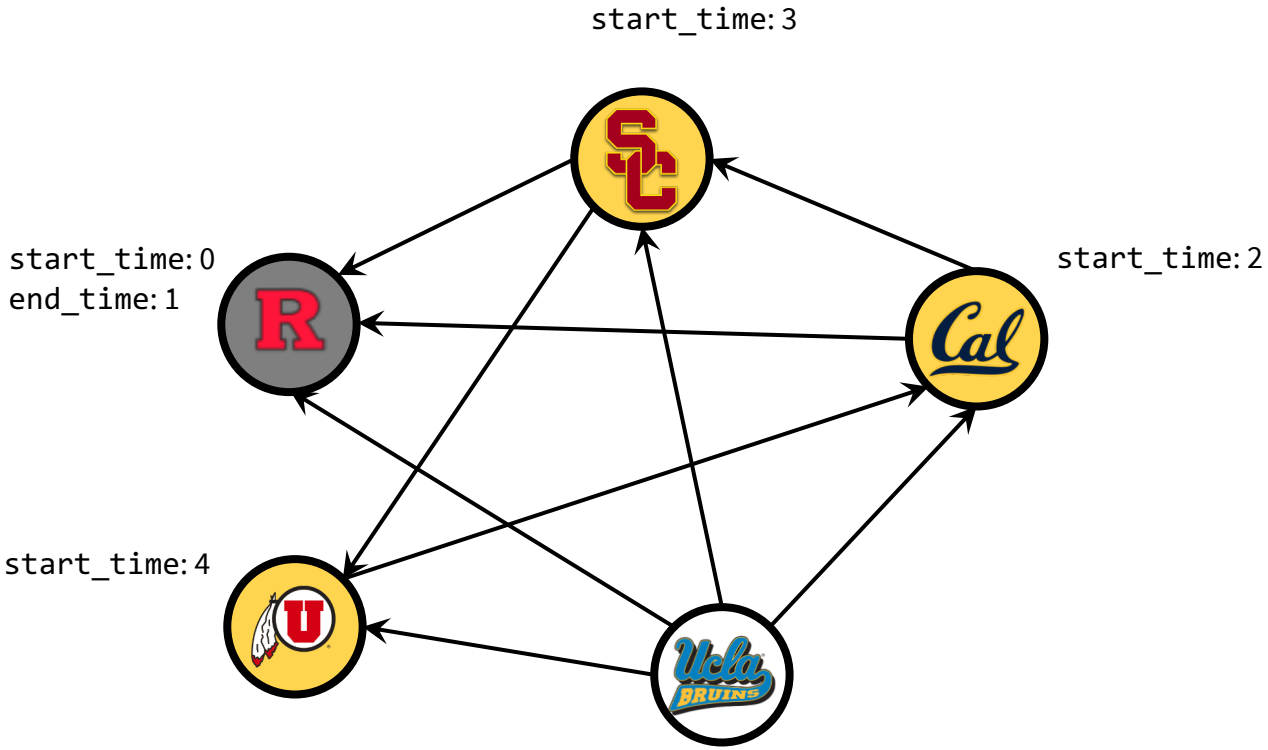
Kosaraju's Algorithm

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



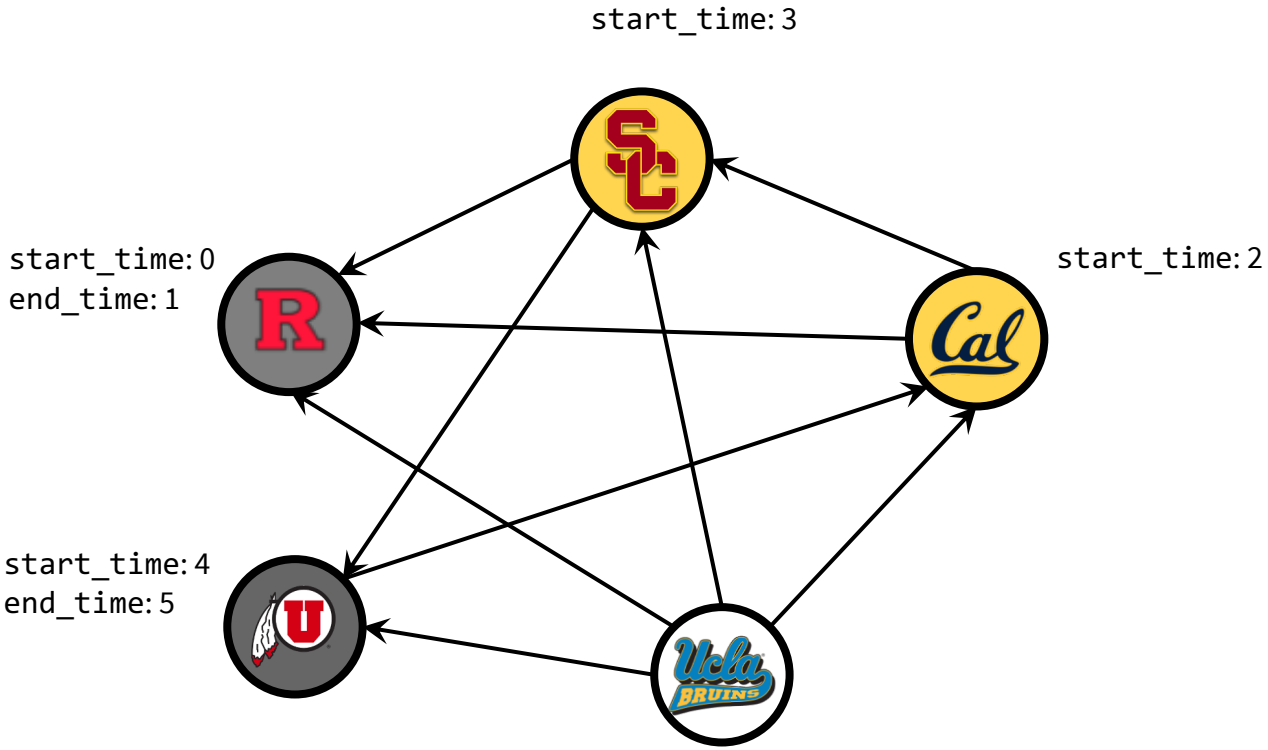
Kosaraju's Algorithm

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



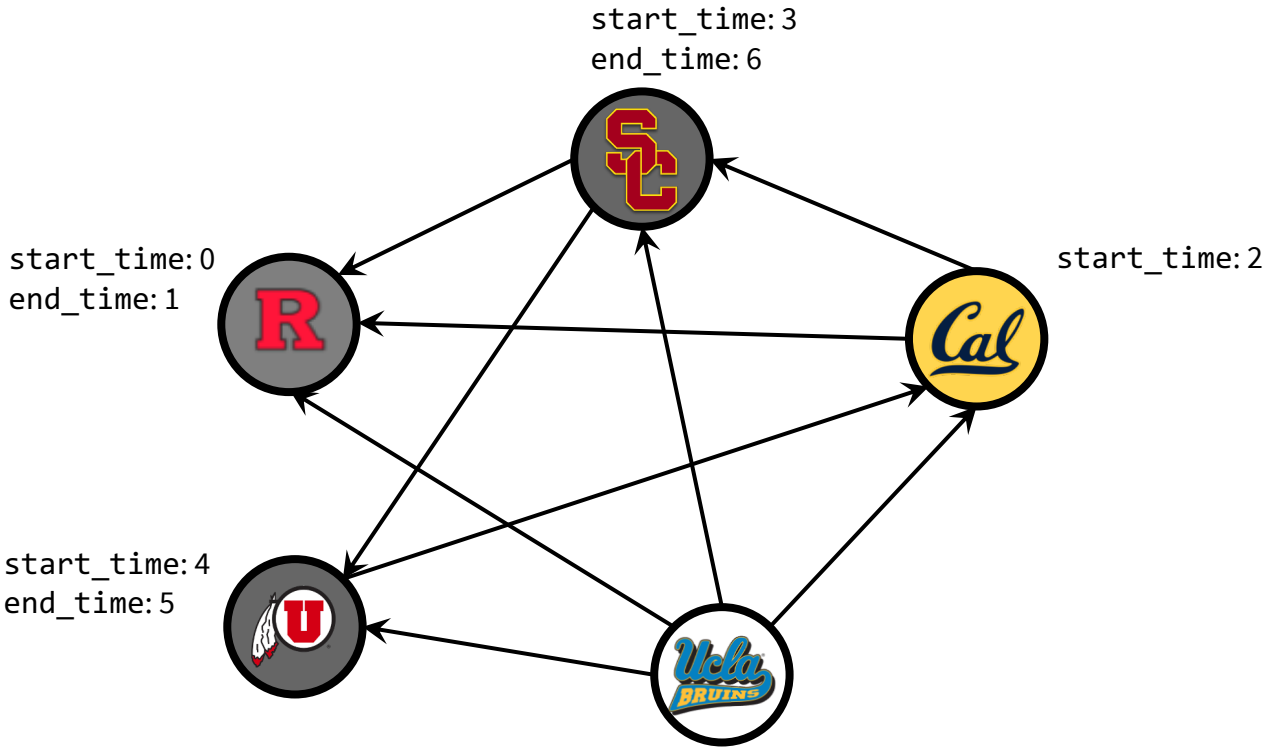
Kosaraju's Algorithm

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



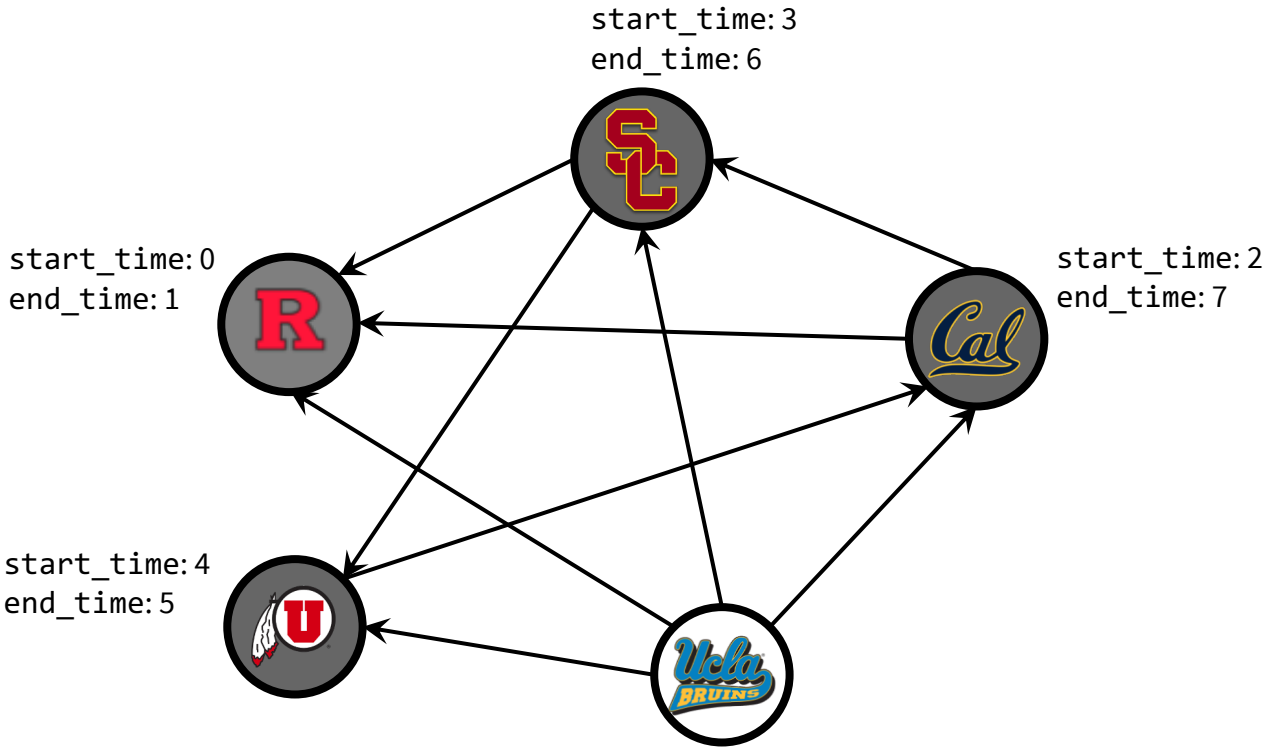
Kosaraju's Algorithm

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



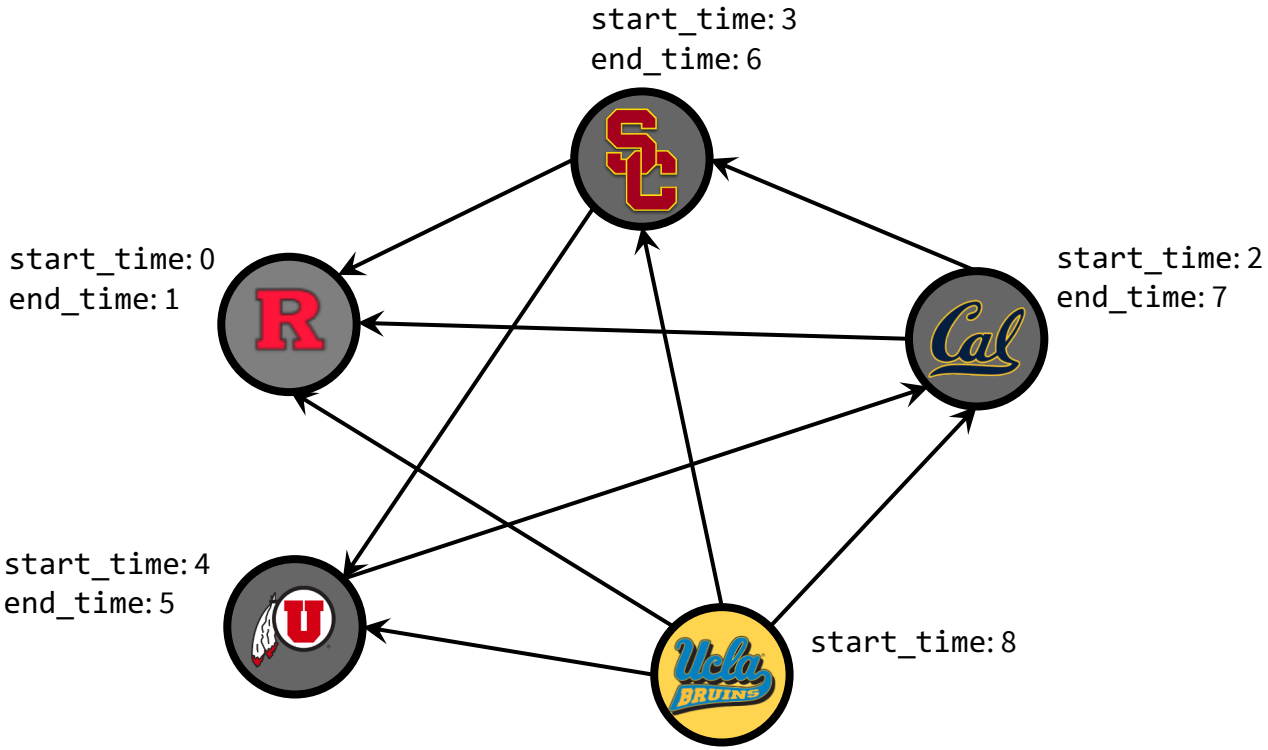
Kosaraju's Algorithm

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



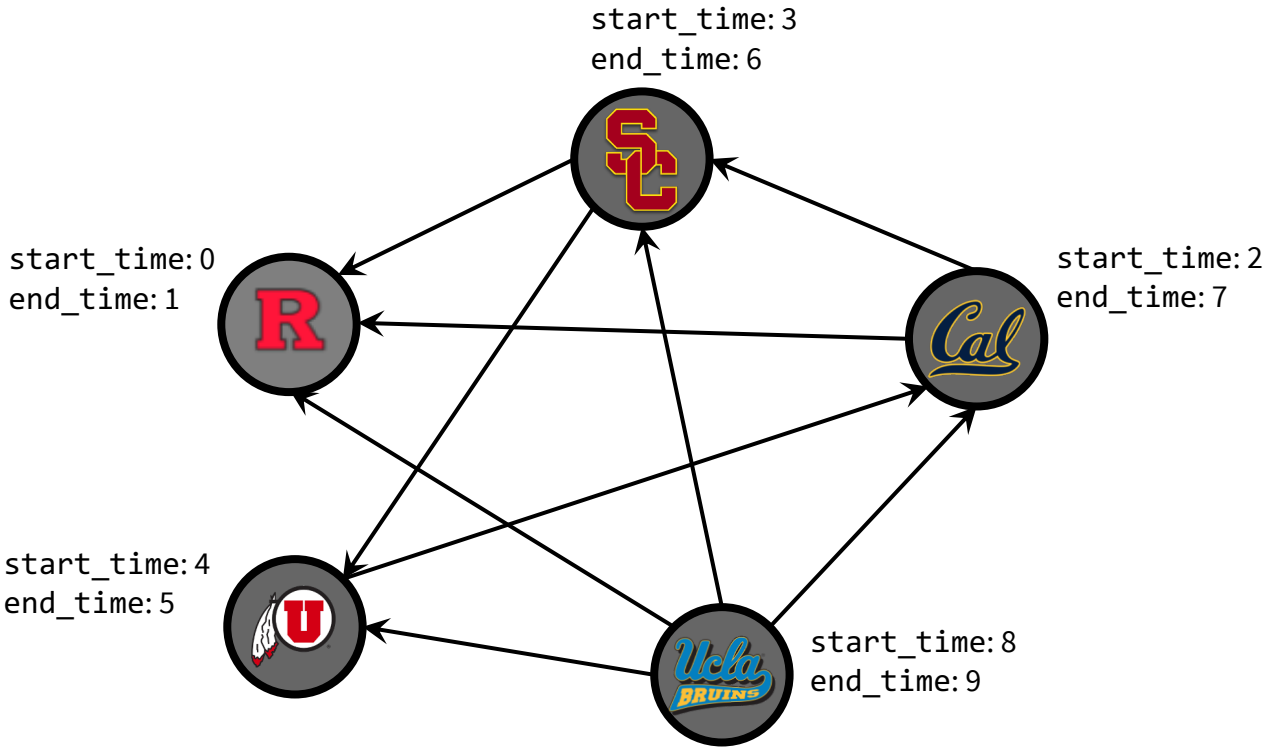
Kosaraju's Algorithm

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



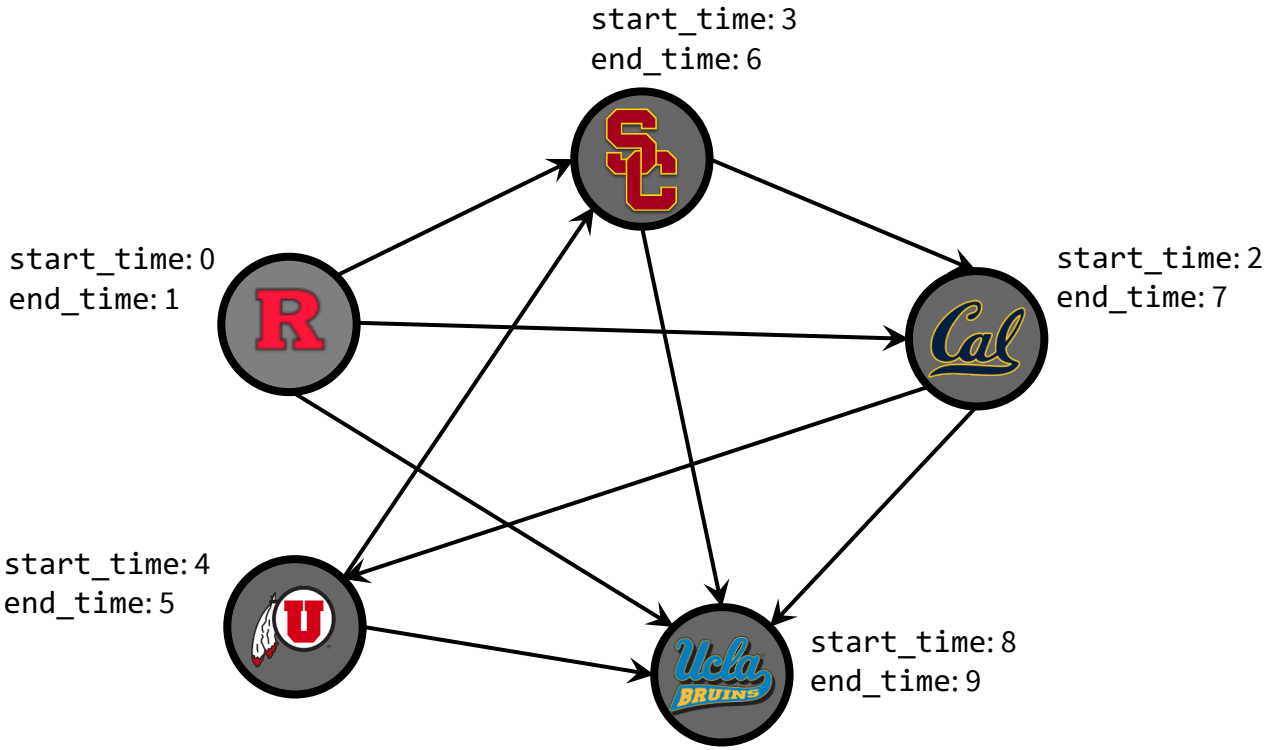
Kosaraju's Algorithm

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



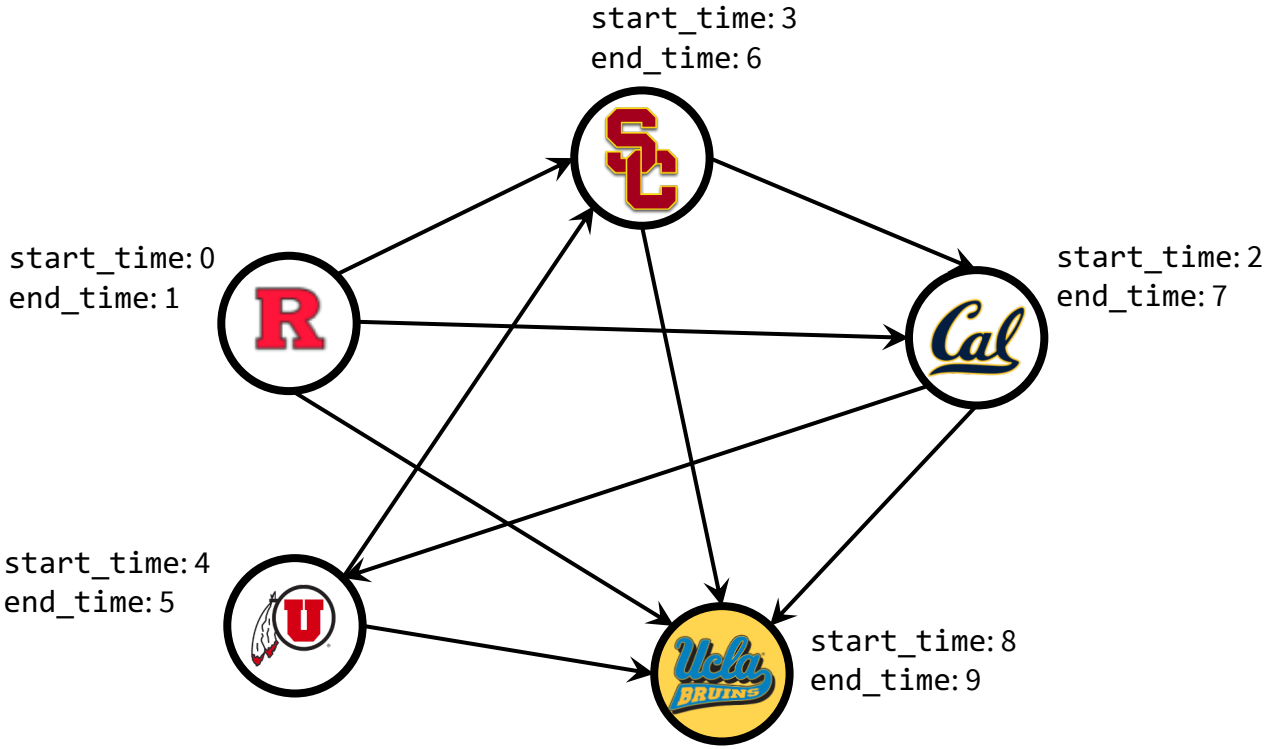
Kosaraju's Algorithm

2. Reverse all of the edges.



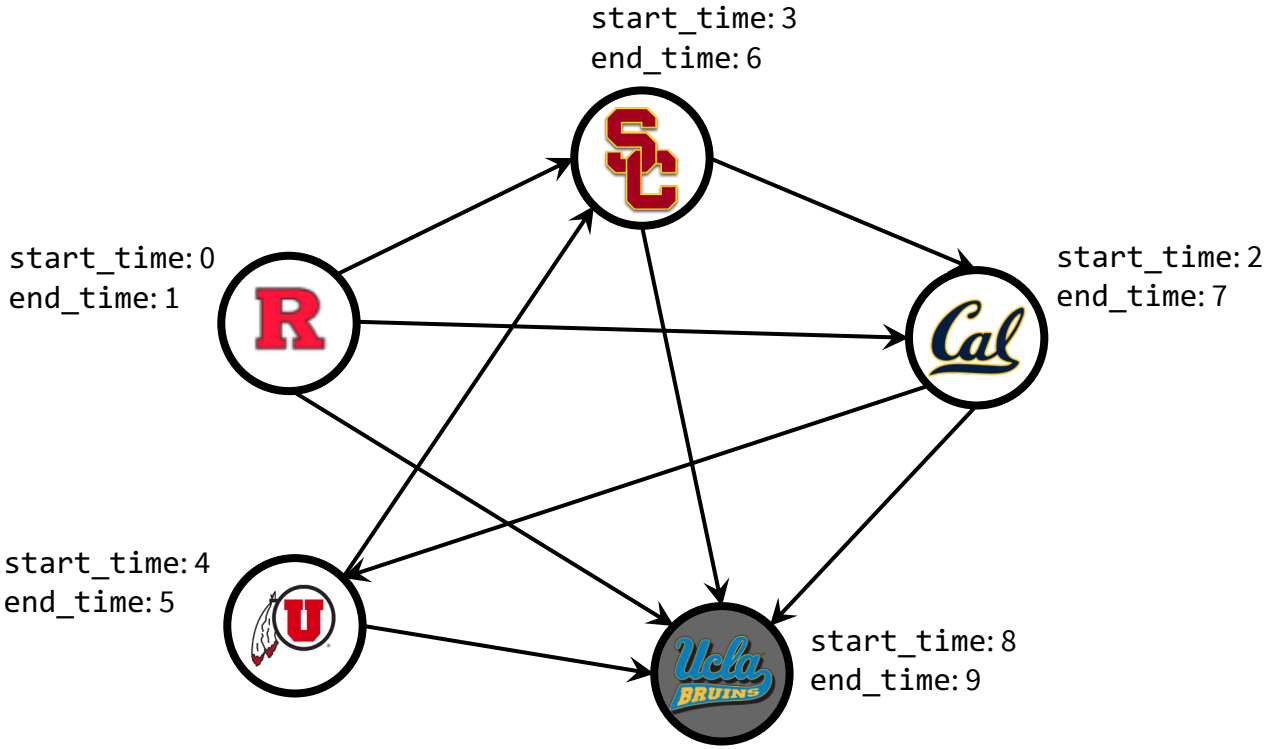
Kosaraju's Algorithm

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



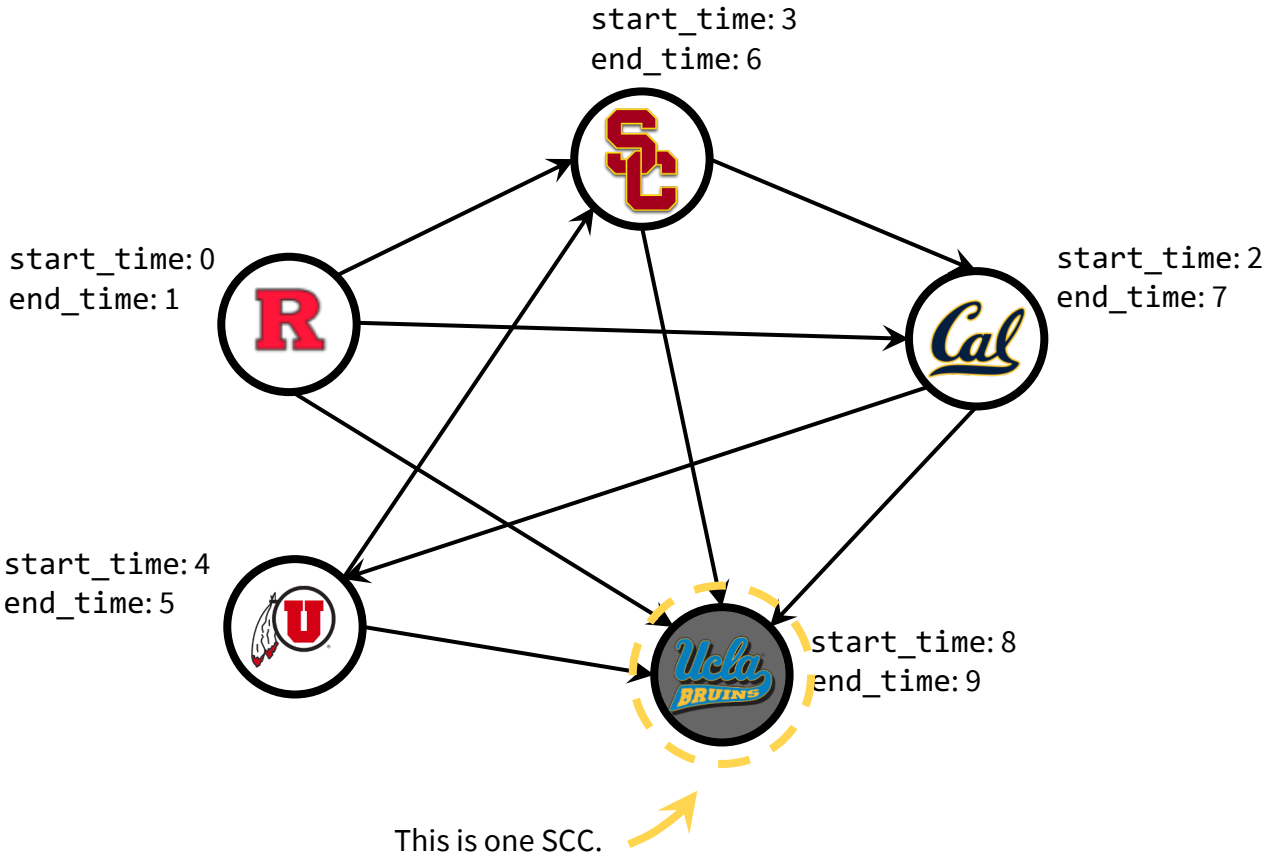
Kosaraju's Algorithm

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



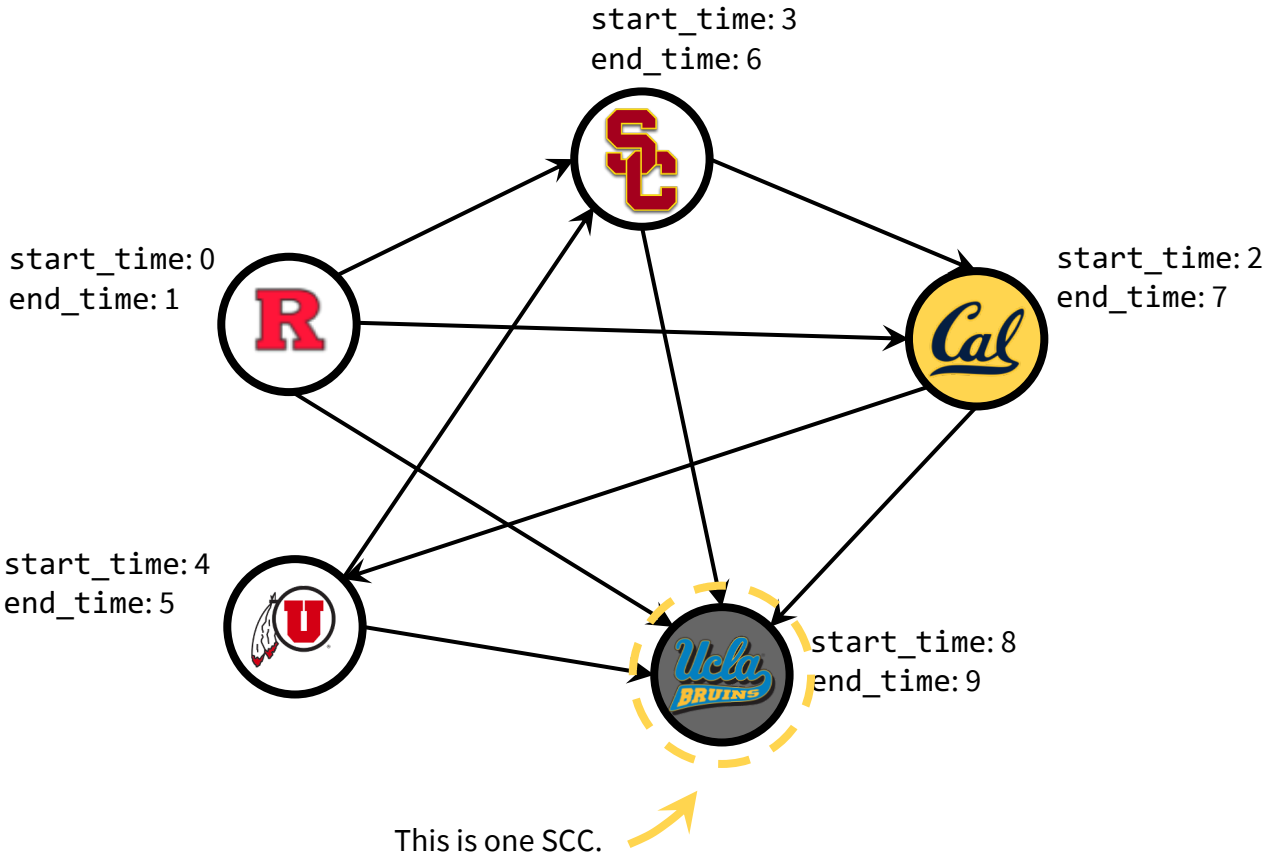
Kosaraju's Algorithm

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



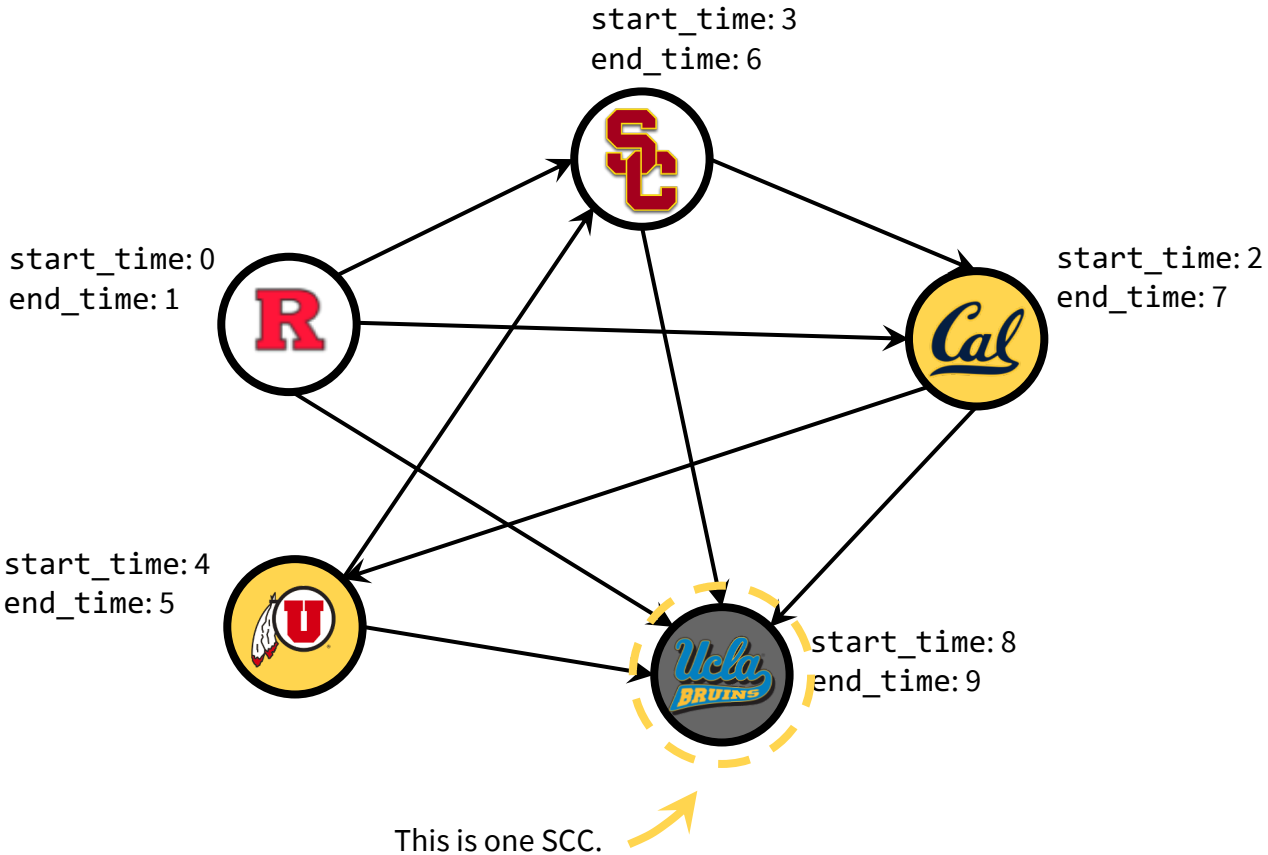
Kosaraju's Algorithm

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



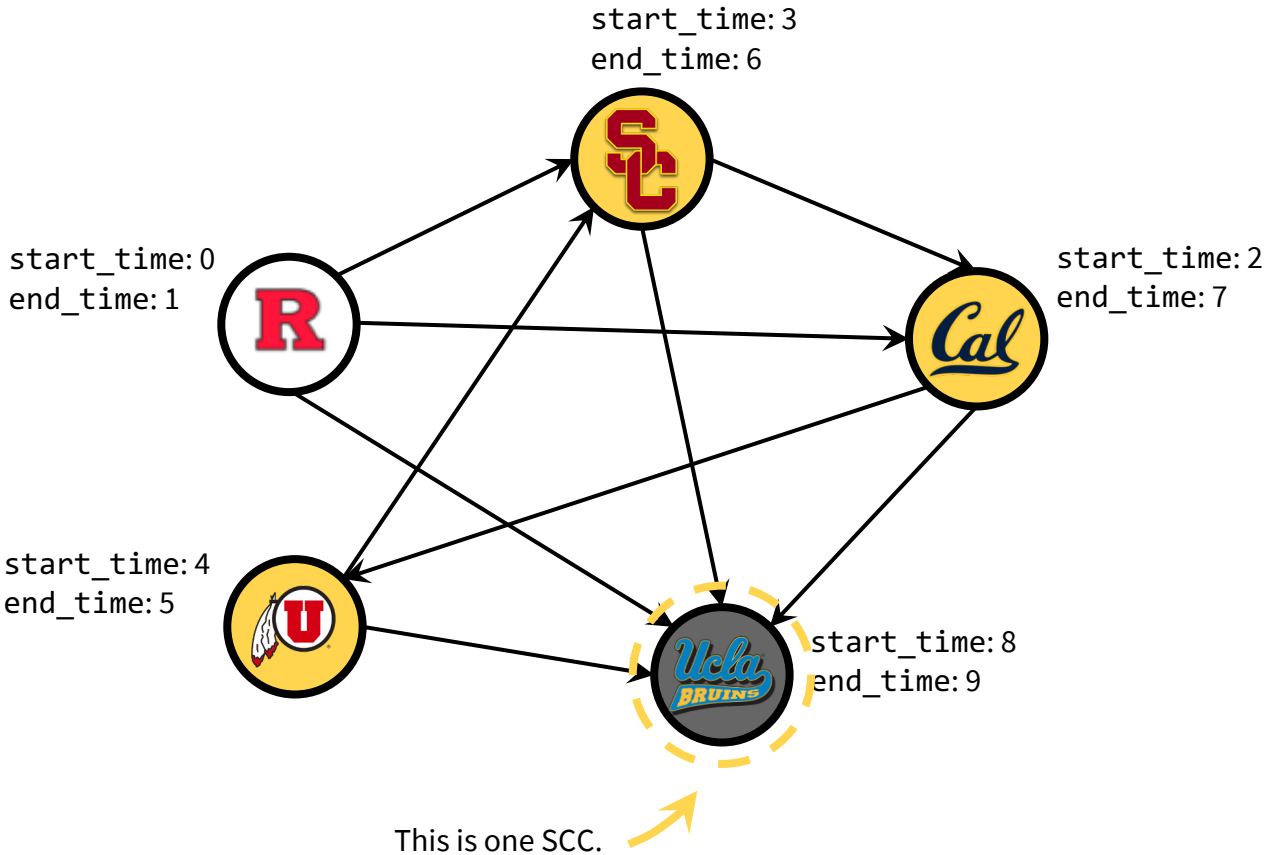
Kosaraju's Algorithm

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



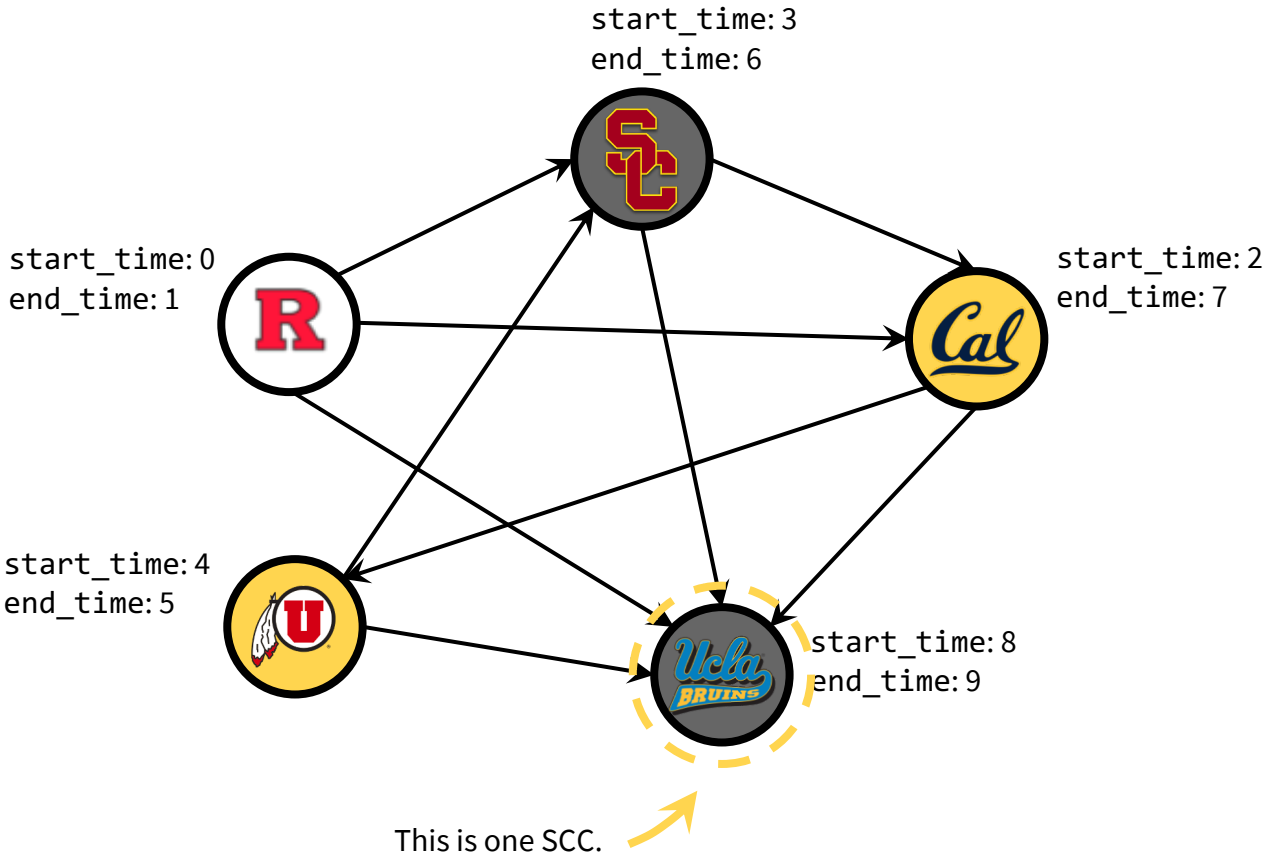
Kosaraju's Algorithm

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



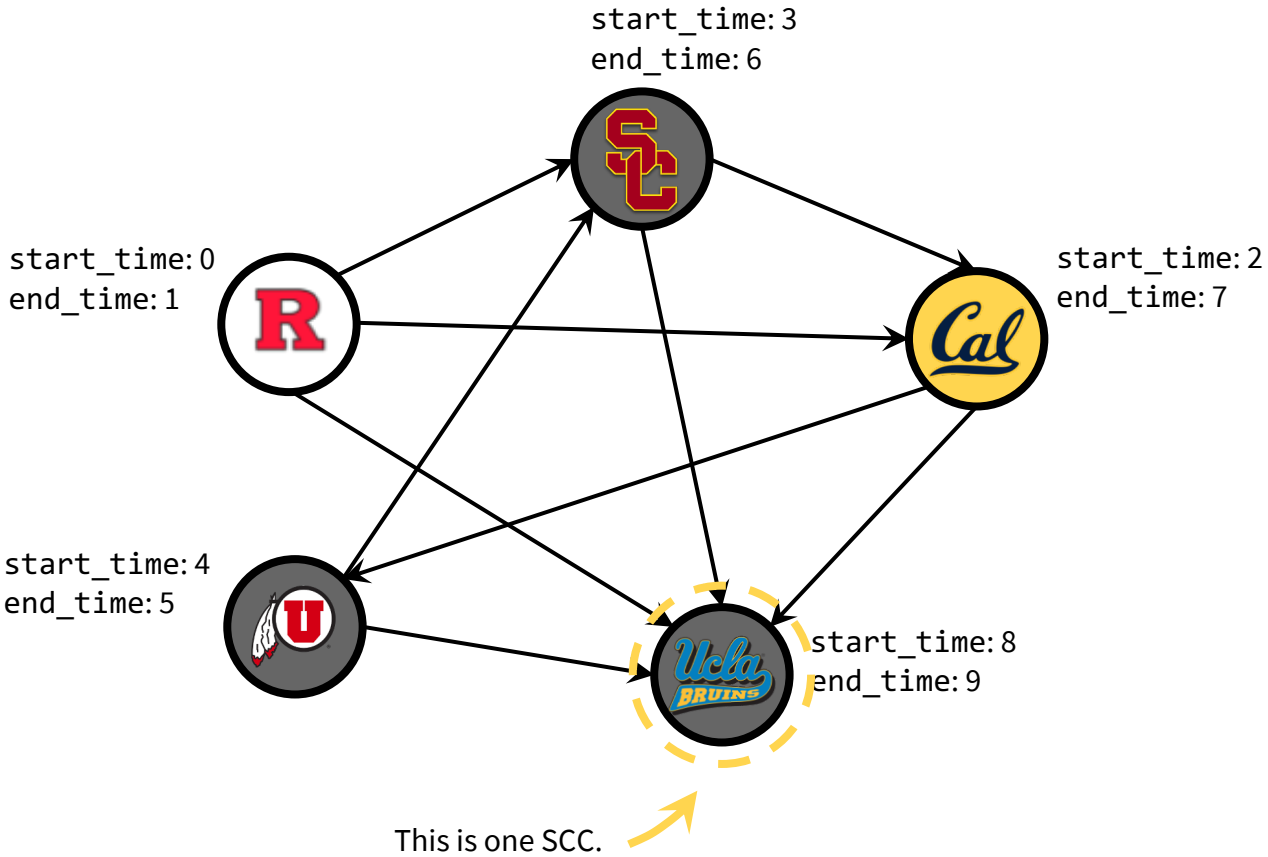
Kosaraju's Algorithm

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



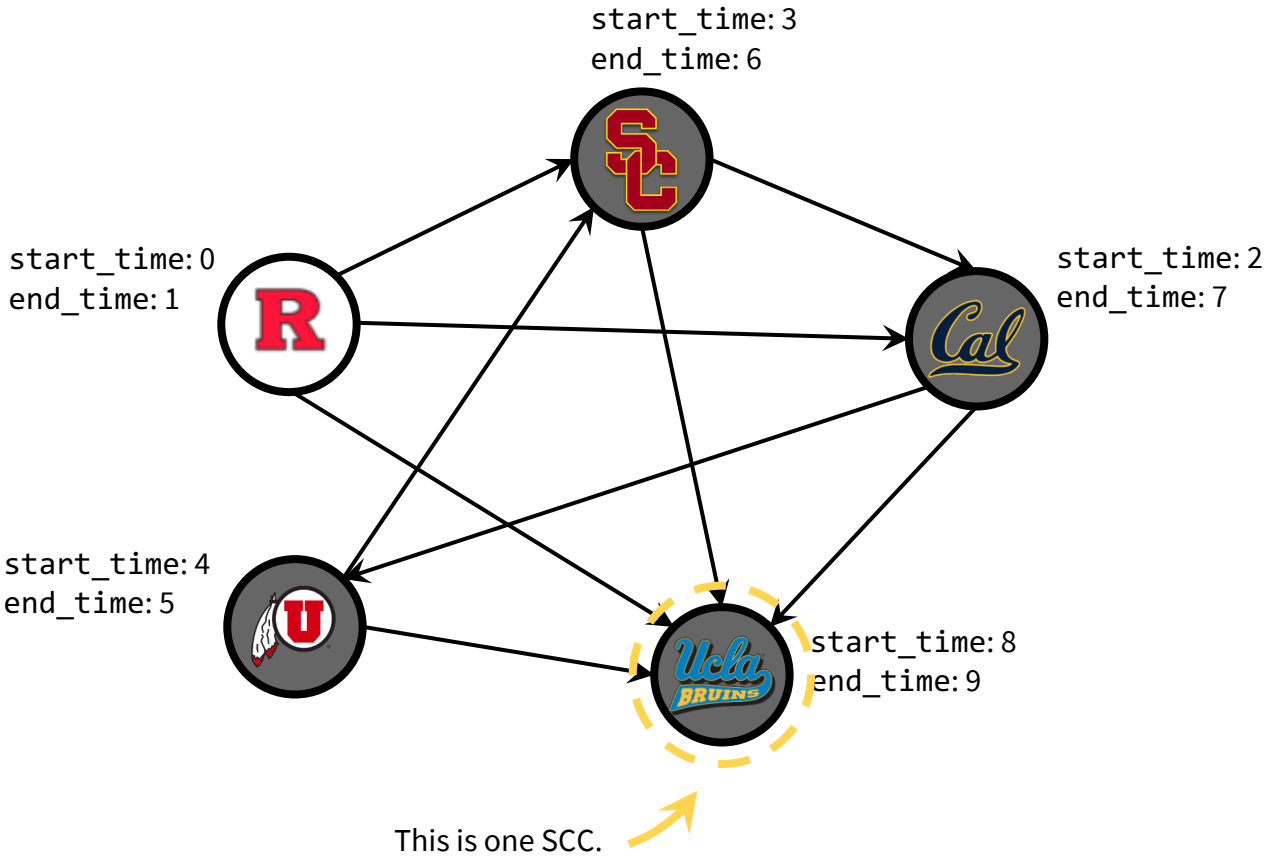
Kosaraju's Algorithm

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



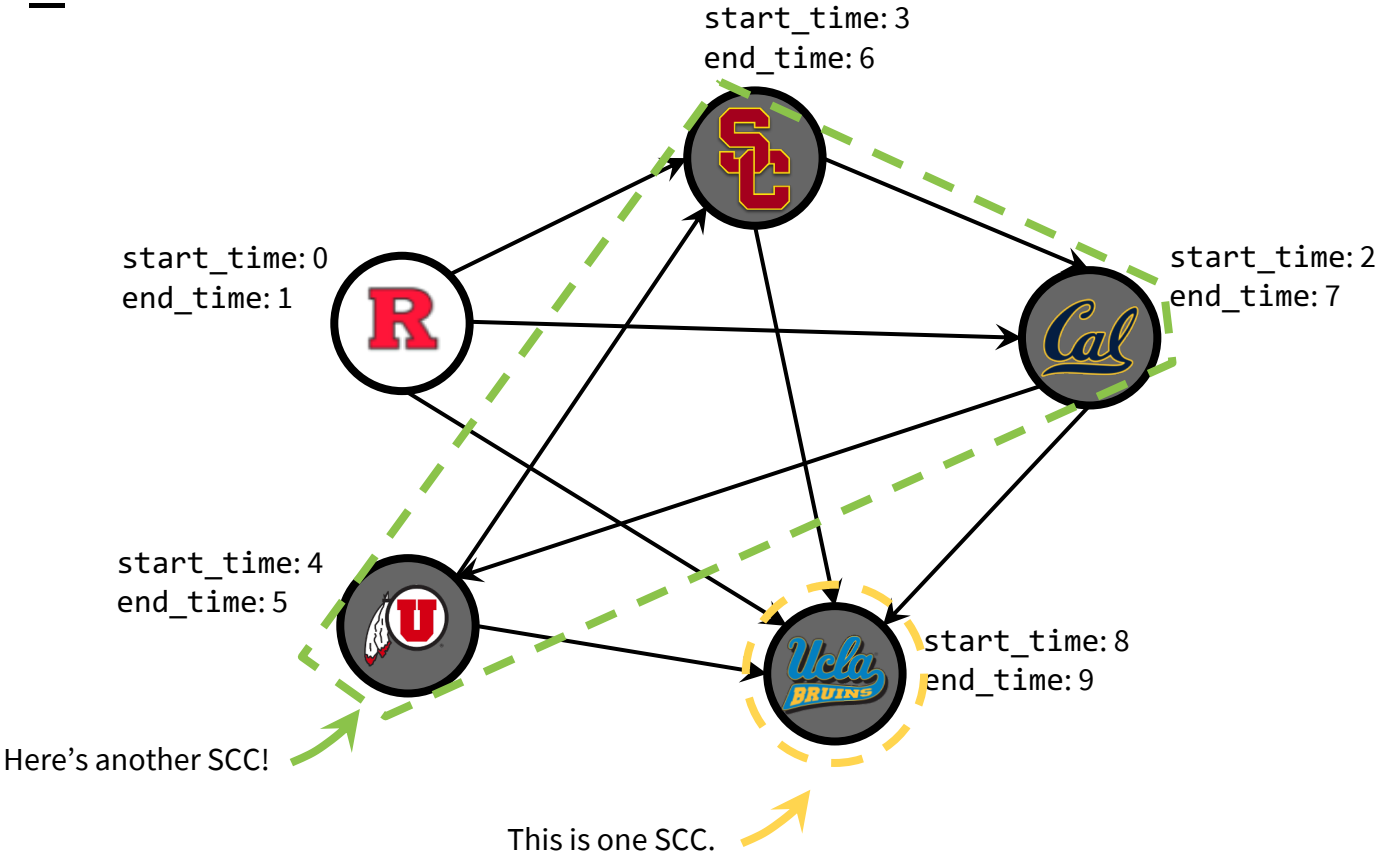
Kosaraju's Algorithm

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



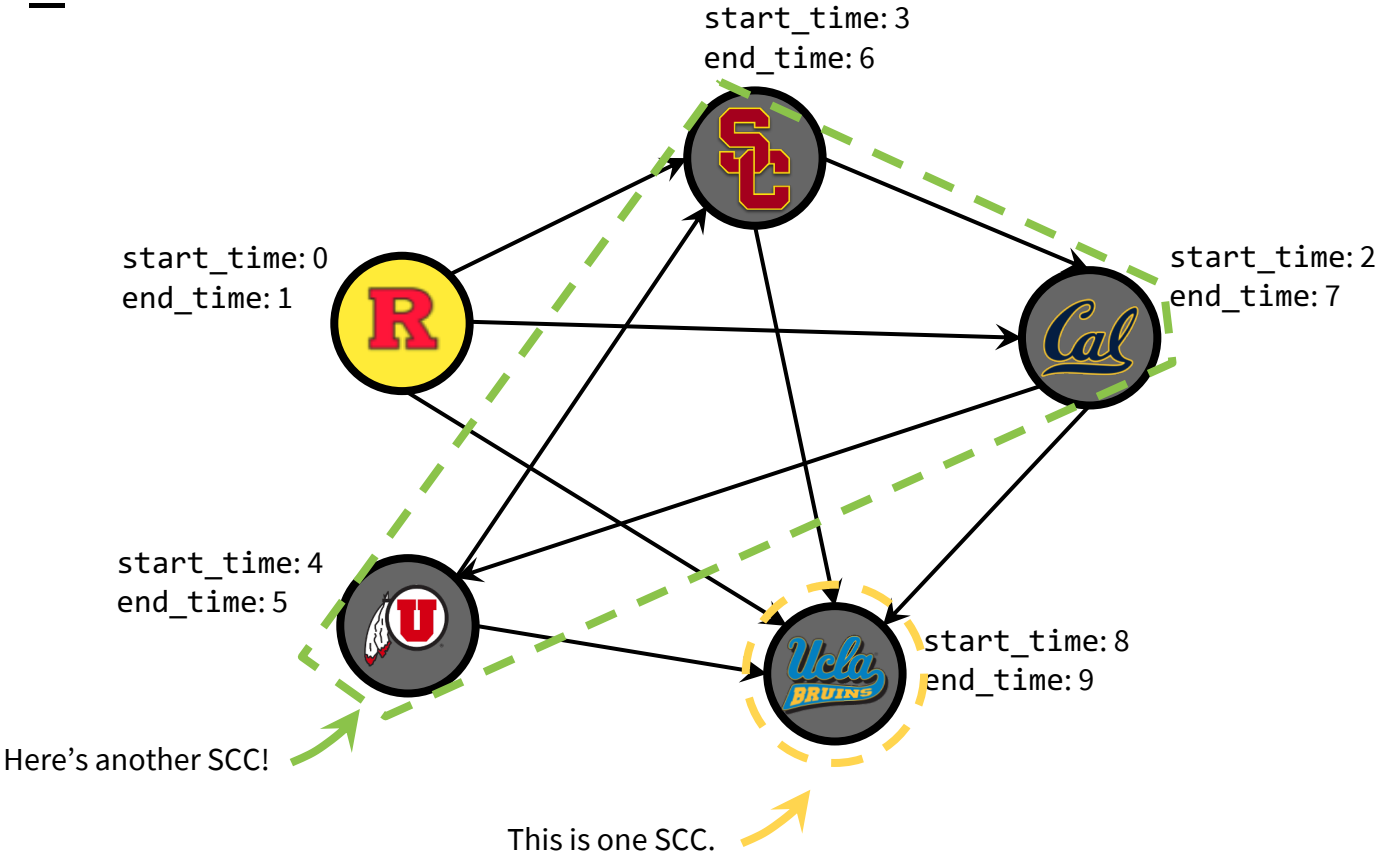
Kosaraju's Algorithm

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



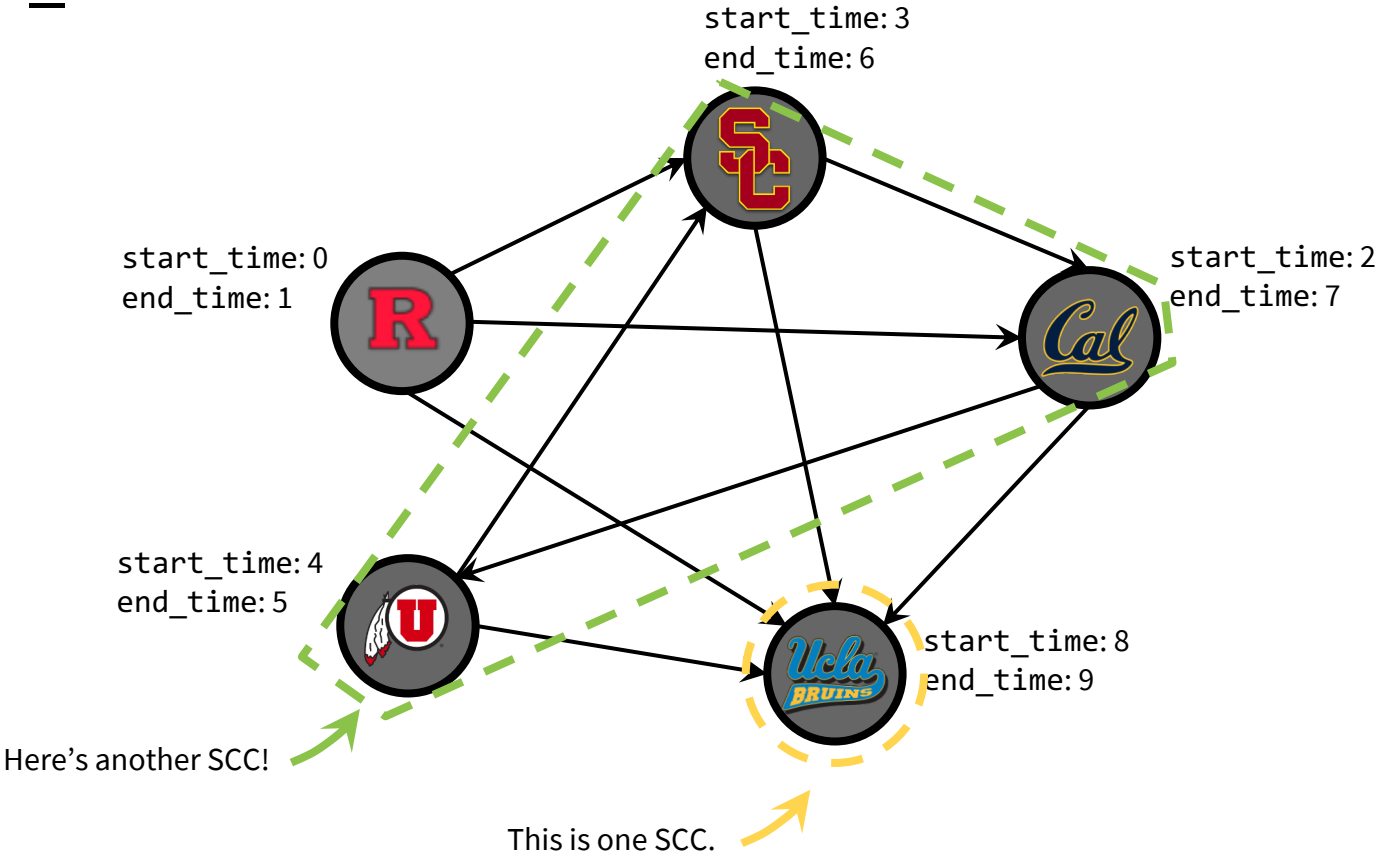
Kosaraju's Algorithm

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



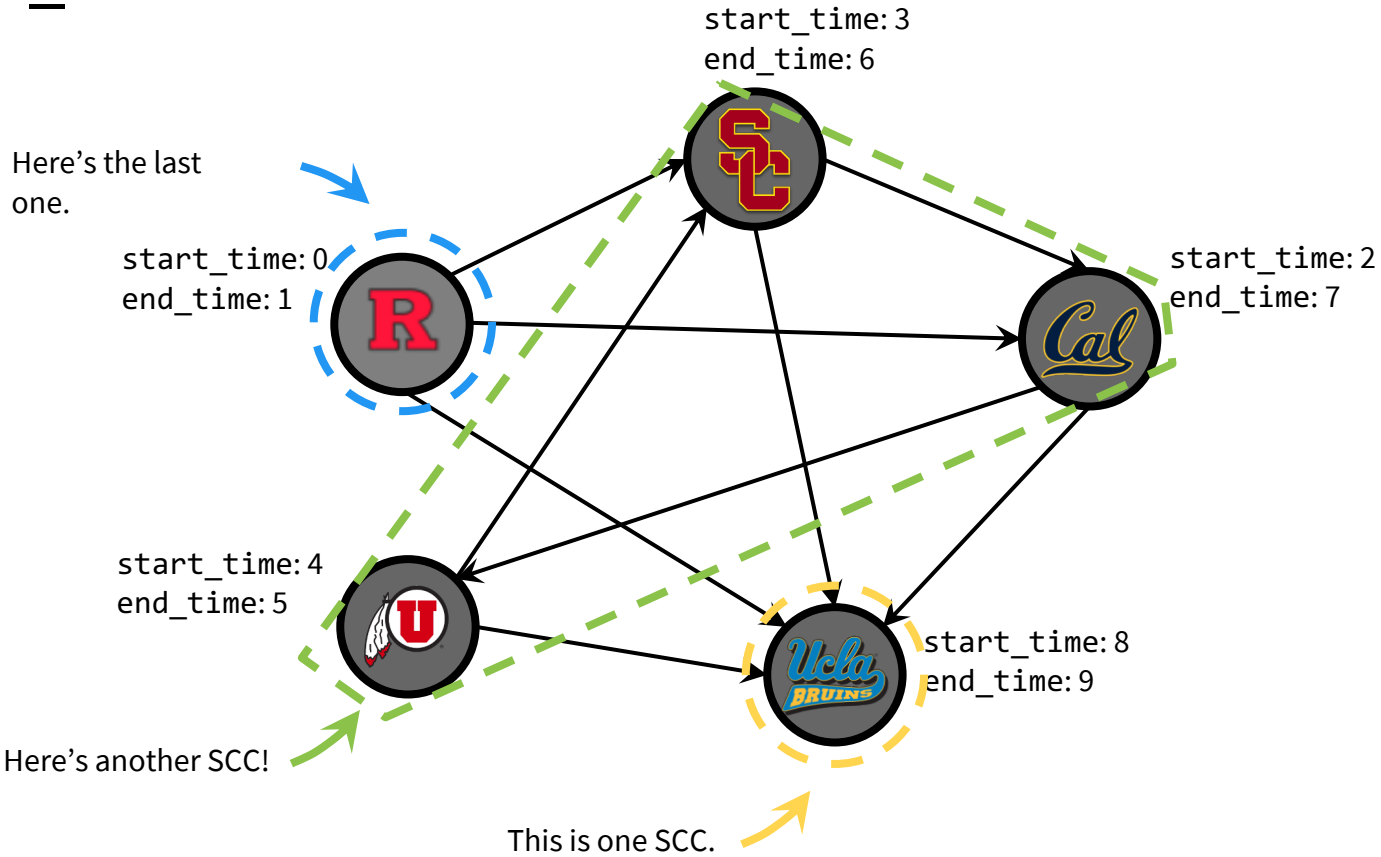
Kosaraju's Algorithm

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`.



Why Kosaraju's Algorithm Works

Kosaraju's Algorithm

Whoa. How did that work?

We explain by answering two questions:

1) Why do we use Depth First Search

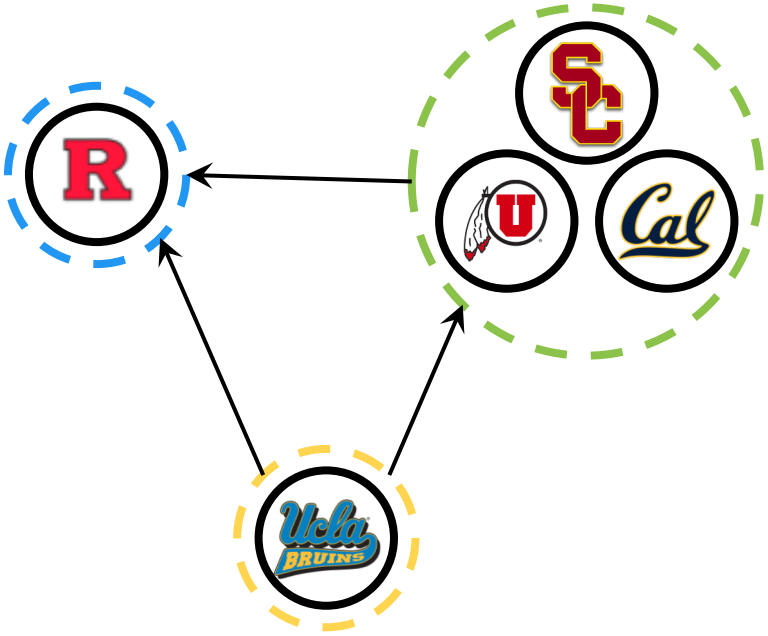
2) In the second DFS, why do we start from vertex with largest end_time in each round.

Kosaraju's Algorithm

Why do we use Depth First Search

Lemma 1: The SCC metagraph is a directed acyclic graph (DAG).

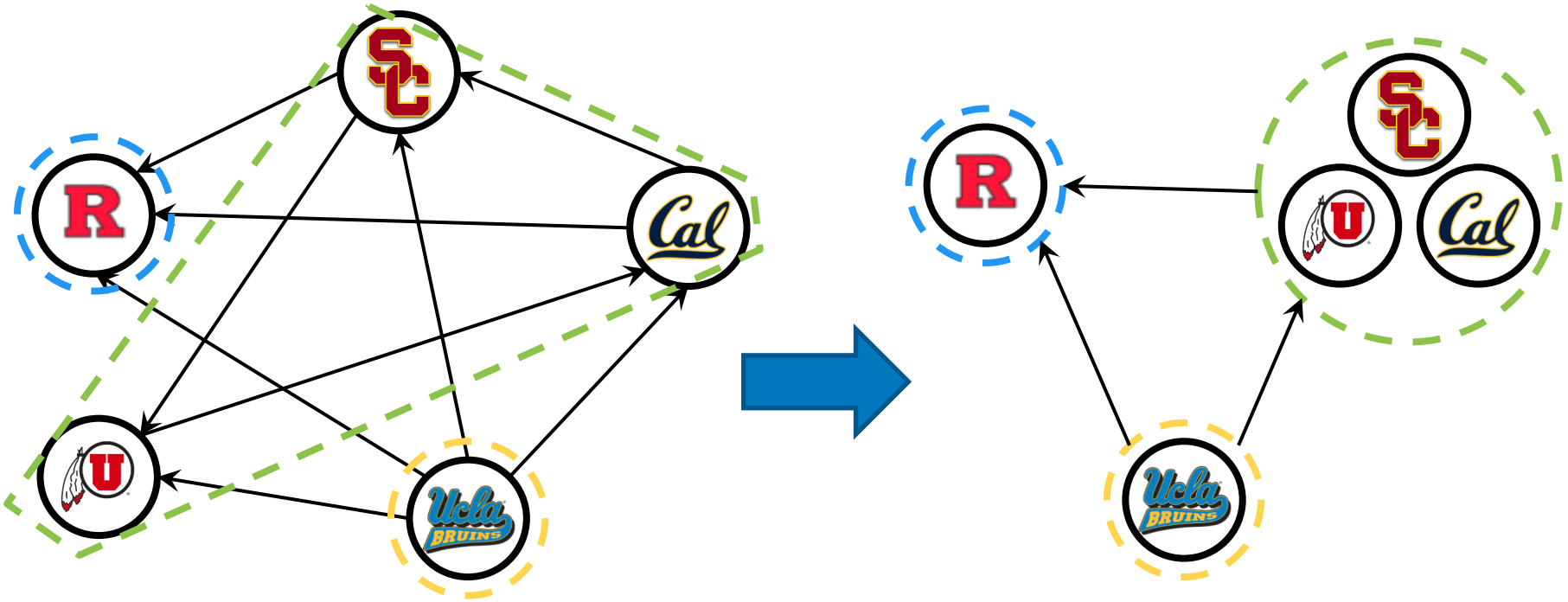
Intuition: If not, then two SCCs would collapse into one.



Kosaraju's Algorithm

Why do we use Depth First Search

Review: DAG, a directed graph with no directed cycles.
Metagraph: connect a pair of meta vertex as long as there exists an edge between any vertex pair from each meta vertex.



Original Graph

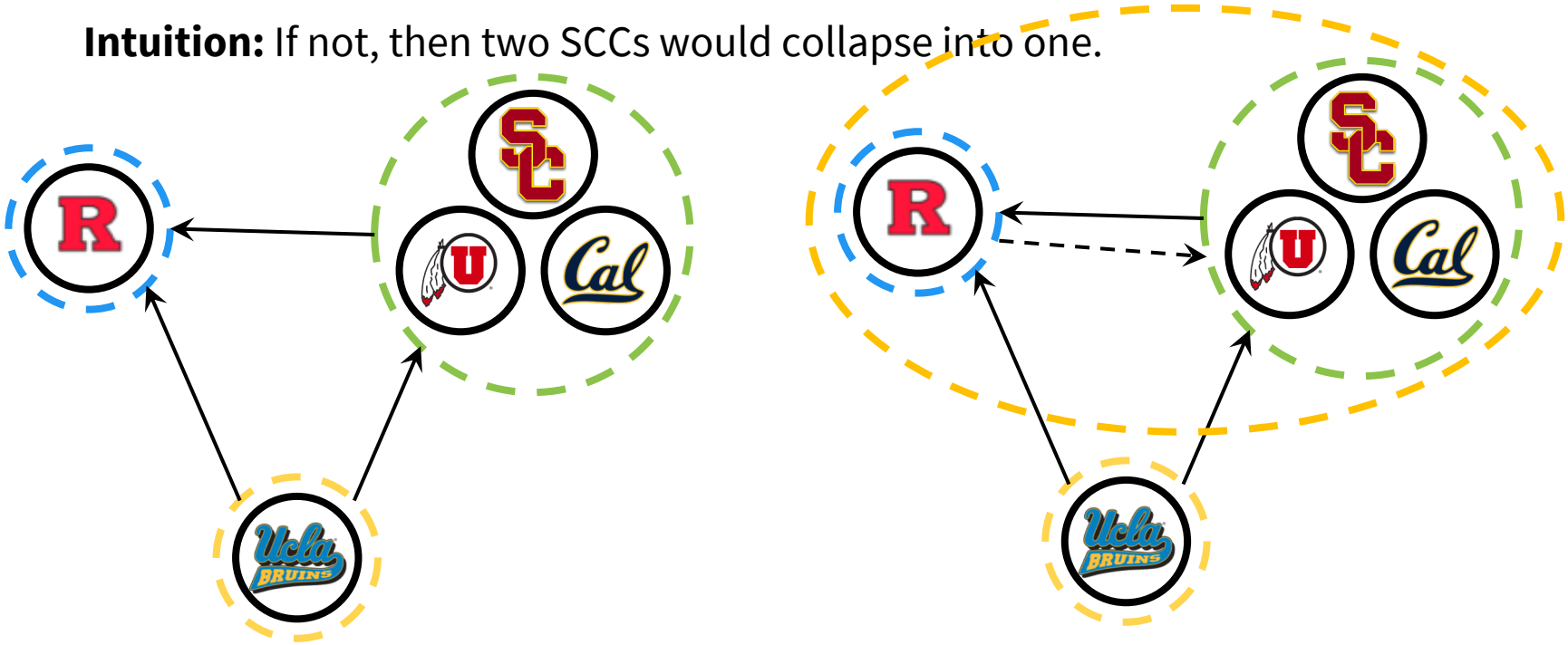
SCC MetaGraph

Kosaraju's Algorithm

Why do we use Depth First Search

Lemma 1: The SCC metagraph is a directed acyclic graph (DAG).

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



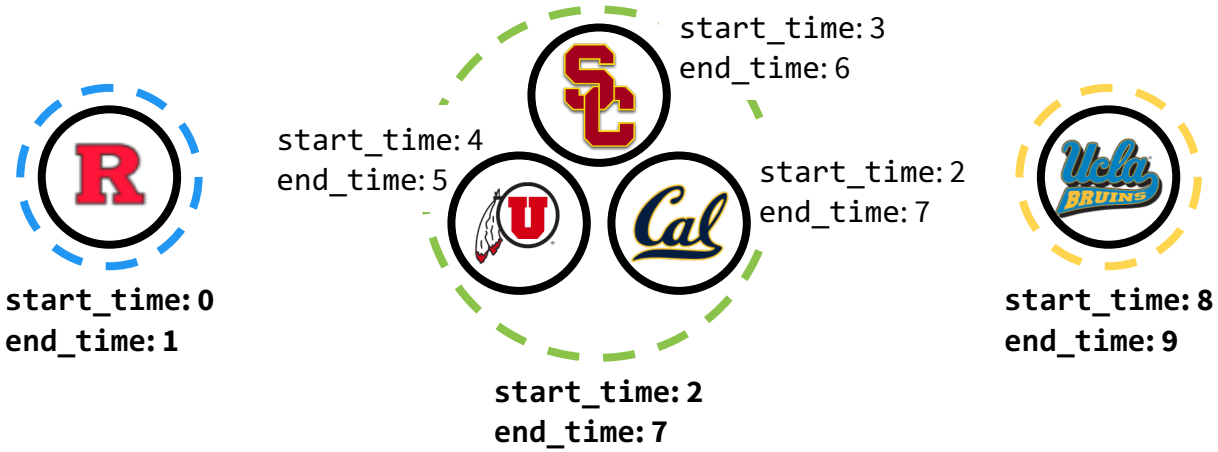
Proof: assume there exist a cycle path between two meta vertex, then they can be merged into a bigger metavertex.

Kosaraju's Algorithm

Why do we use Depth First Search

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

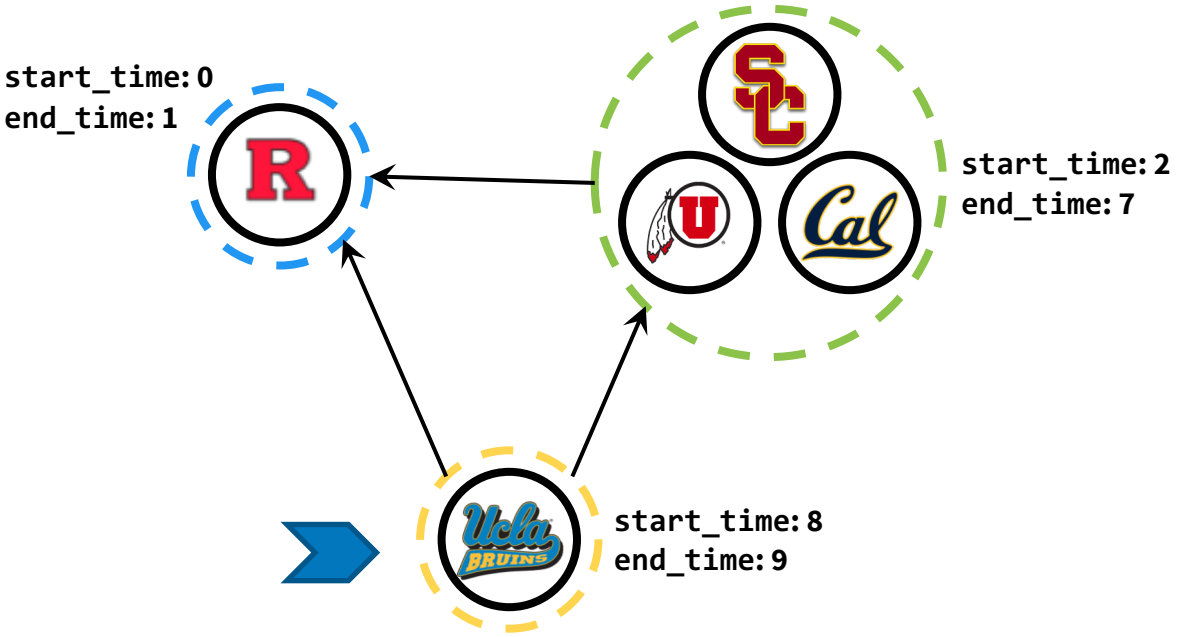
Let the **starting time** of a SCC be the smallest starting time of any element of that SCC.



Kosaraju's Algorithm

Why do we use Depth First Search

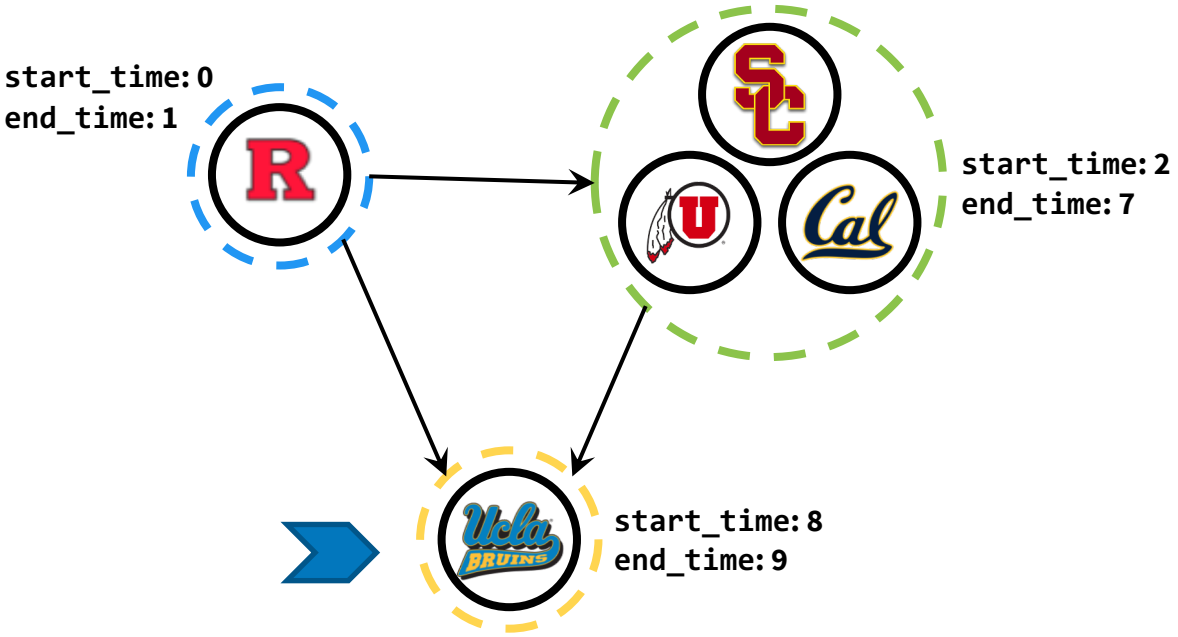
The main idea leverages the fact that vertex in the **SCC metagraph** with the largest `end_time` has no incoming edges.



Kosaraju's Algorithm

Why do we use Depth First Search

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.

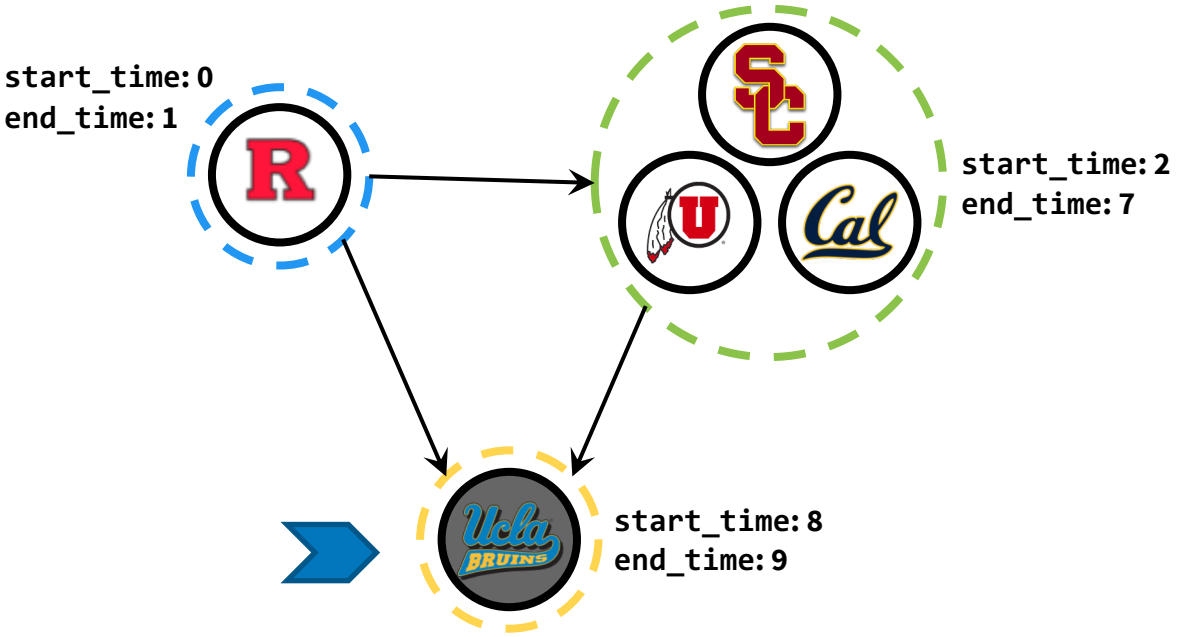


Kosaraju's Algorithm

Why do we use Depth First Search

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.



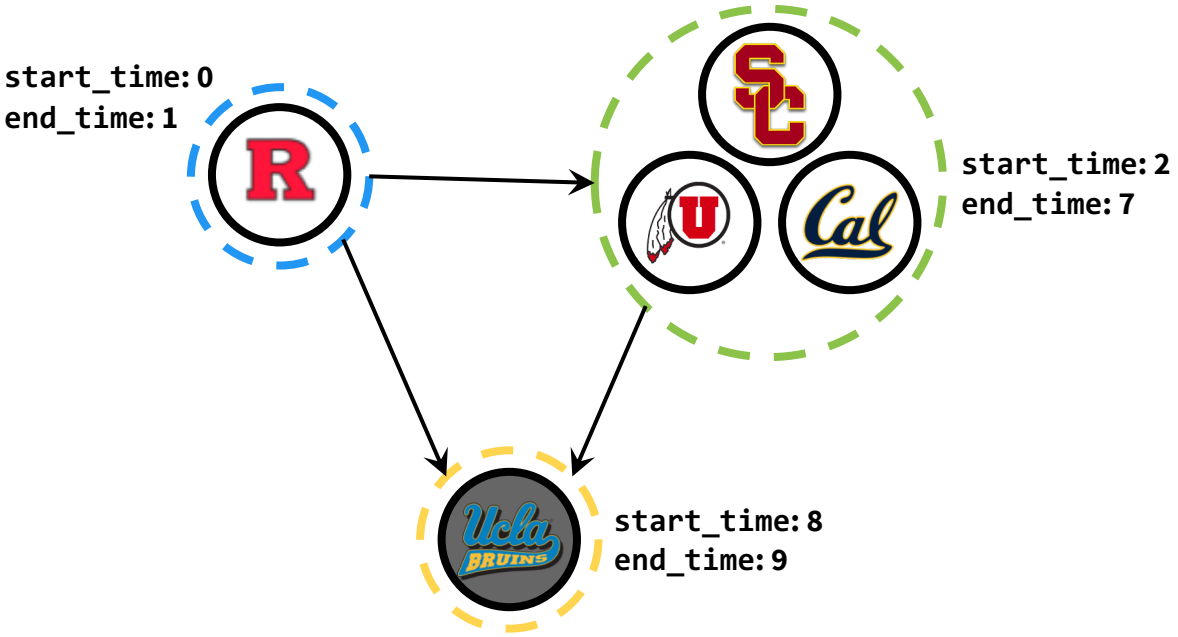
Kosaraju's Algorithm

Why do we use Depth First Search

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.



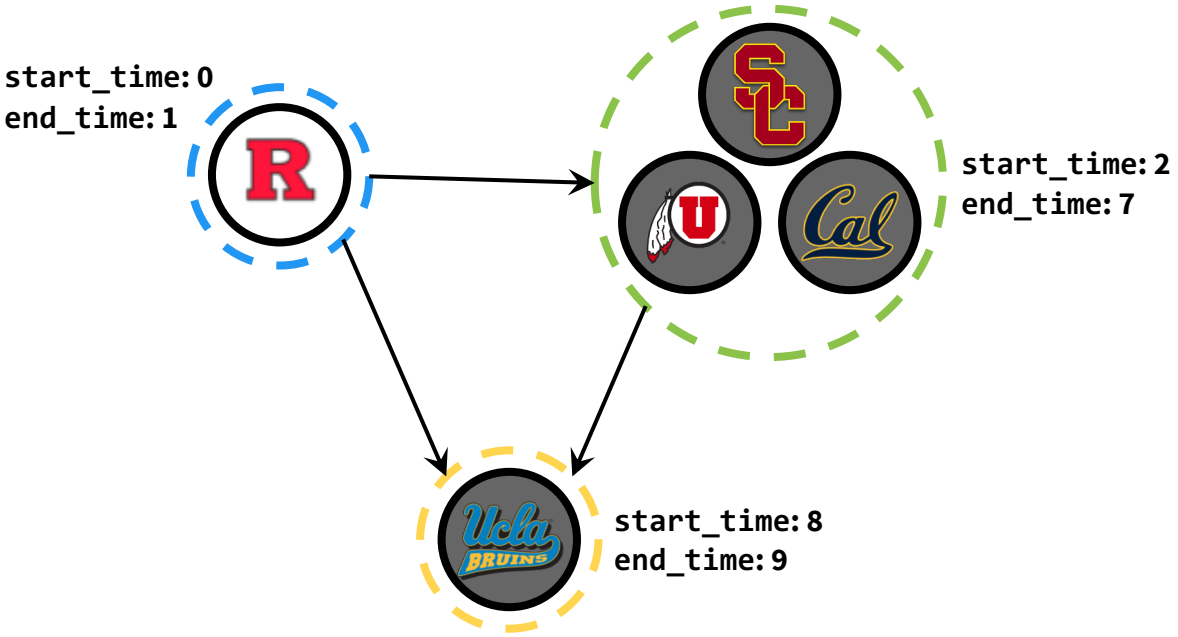
Kosaraju's Algorithm

Why do we use Depth First Search

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.



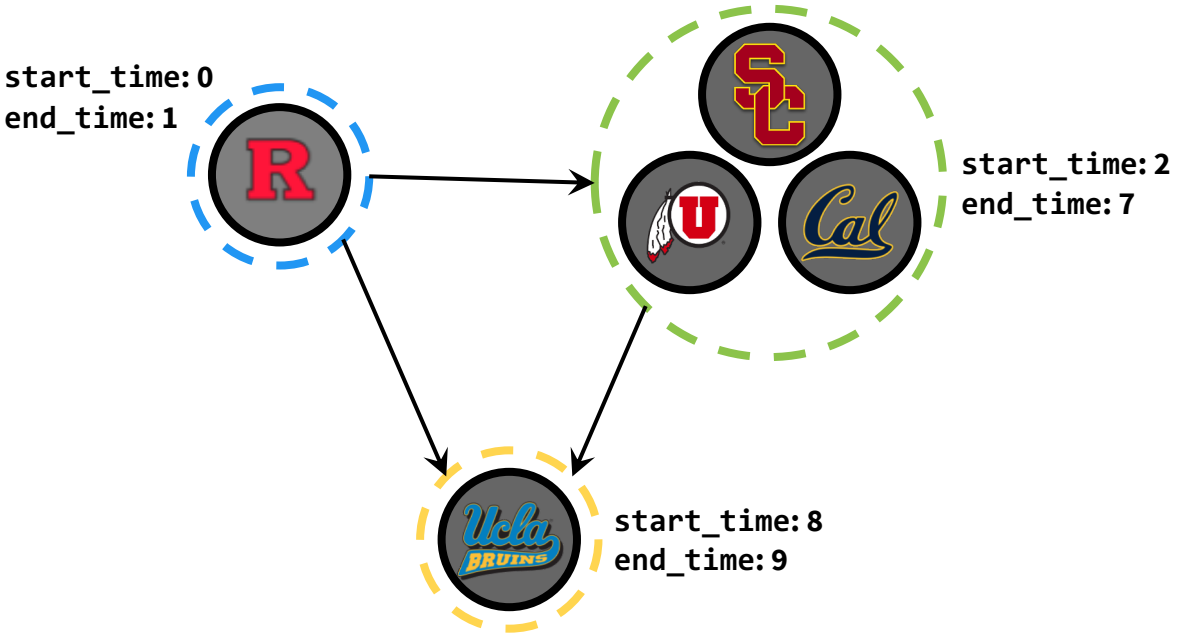
Kosaraju's Algorithm

Why do we use Depth First Search

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.

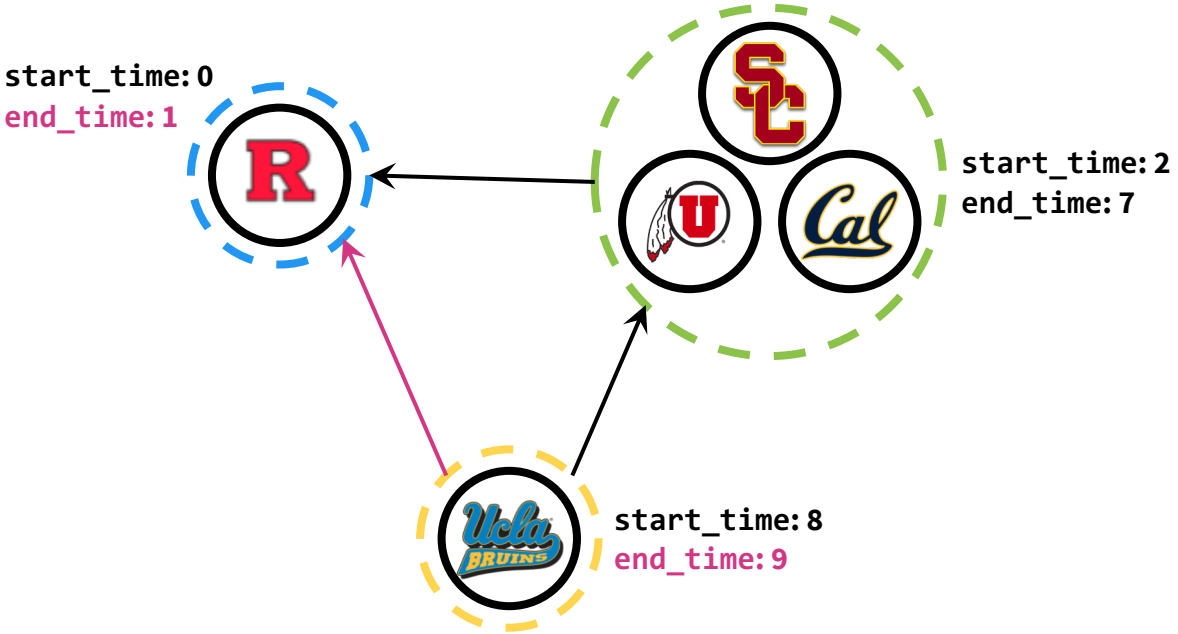
Same argument for the rest.



Kosaraju's Algorithm

Why do we use start with the largest end_time vertex

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.

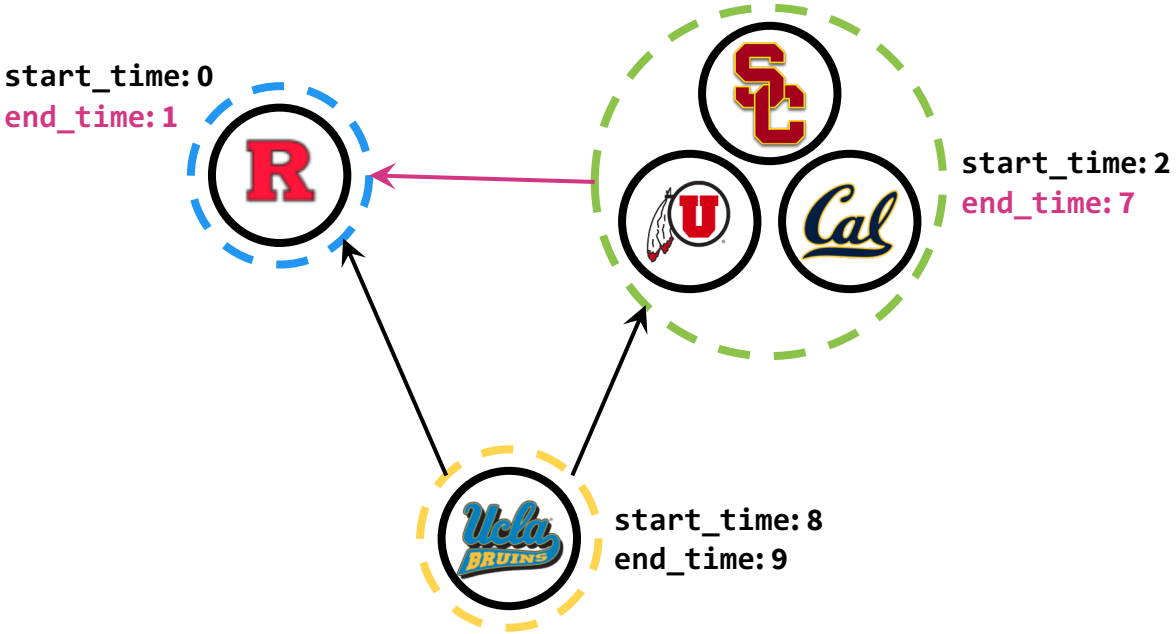


In this way, the second DFS will converge with an SCC before entering another SCC (because all edges are inversed in the second DFS).

Kosaraju's Algorithm

Why do we use start with the largest end_time vertex

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.

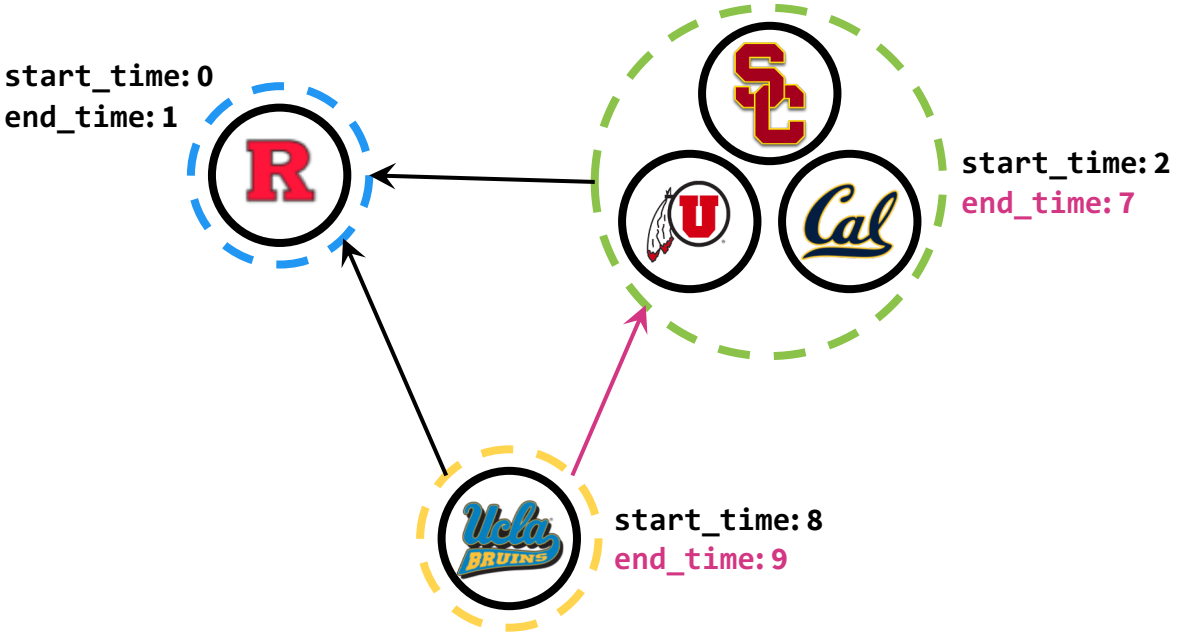


In this way, the second DFS will converge with an SCC before entering another SCC (because all edges are inversed in the second DFS).

Kosaraju's Algorithm

Why do we use start with the largest end_time vertex

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.



In this way, the second DFS will converge with an SCC before entering another SCC (because all edges are inversed in the second DFS).

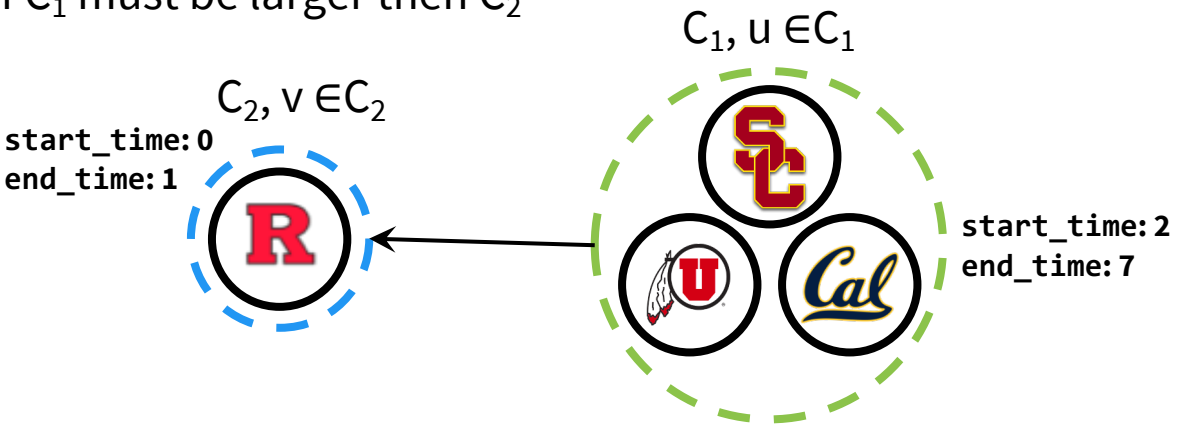
Kosaraju's Algorithm

Why do we use start with the largest end_time vertex

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: Suppose the first DFS started with C_2, it will only finish C_2 before entering C_1 (because there can be no edge from C_2 to C_1, or else C_2 and C_1 can be merged as a single SCC), so the `end_time` of C_1 must be larger then C_2.

Suppose the first DFS started with C_1, because of the existence of the edge (u,v), the algorithm will enter C_2 from u and reach v at some time during DFS; because C_2 is an SCC, all nodes in C_2 are reachable from v, so the DFS will visit all node in C_2; because there is no edge from C_2 to C_1 (same as above), so the DFS will only finish all nodes in C_2 and then trace back to C_1. As a result, the `end_time` of C_1 must be larger then C_2



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!

Karger’s Algorithm

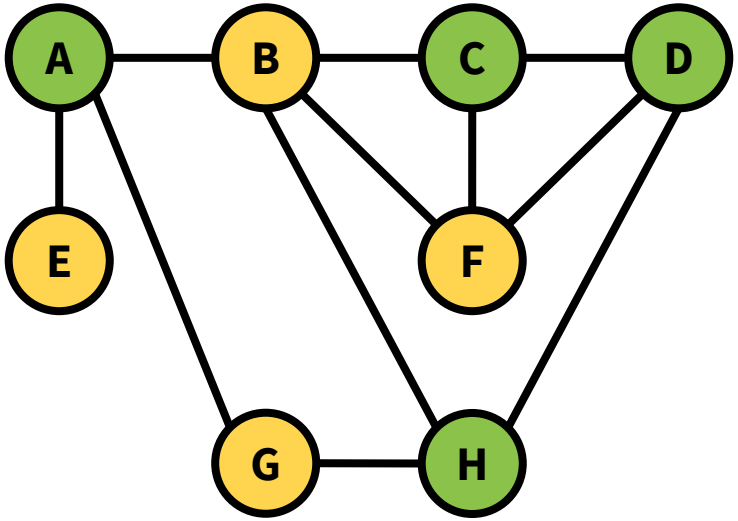
Cuts

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.

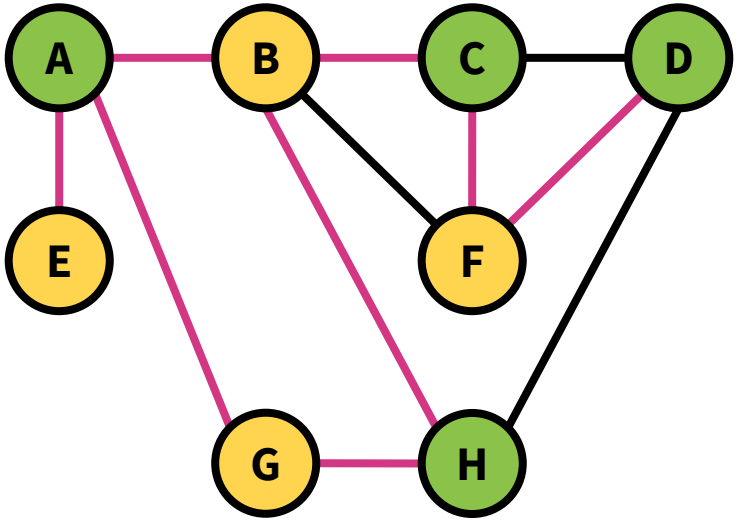
e.g. This is the cut “{A, C, D, H} and {B, E, F, G}”.



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}".



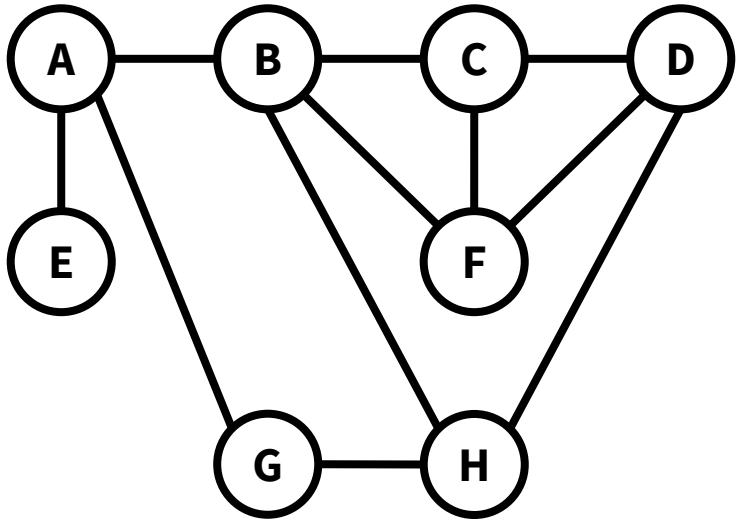
Edges that **cross the cut** go from one part to the other.

e.g. These edges cross the cut.

Explain on board: divide the vertices into two parts

Cuts

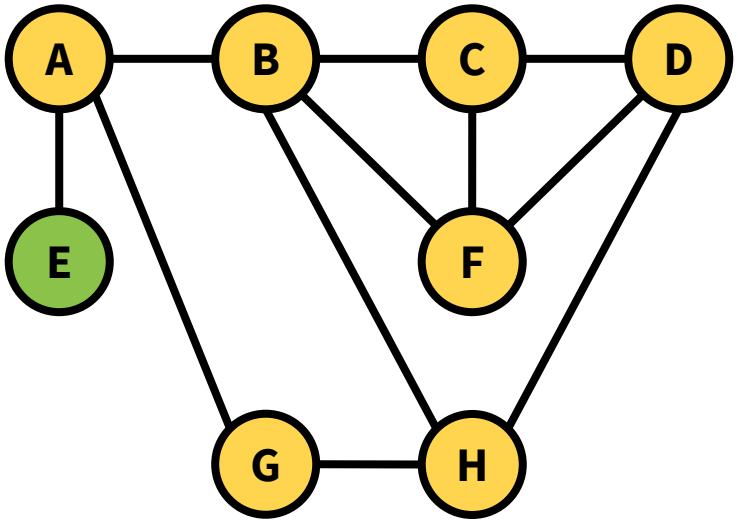
A **global minimum cut** is a cut that has the fewest edges possible crossing it.



Cuts

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

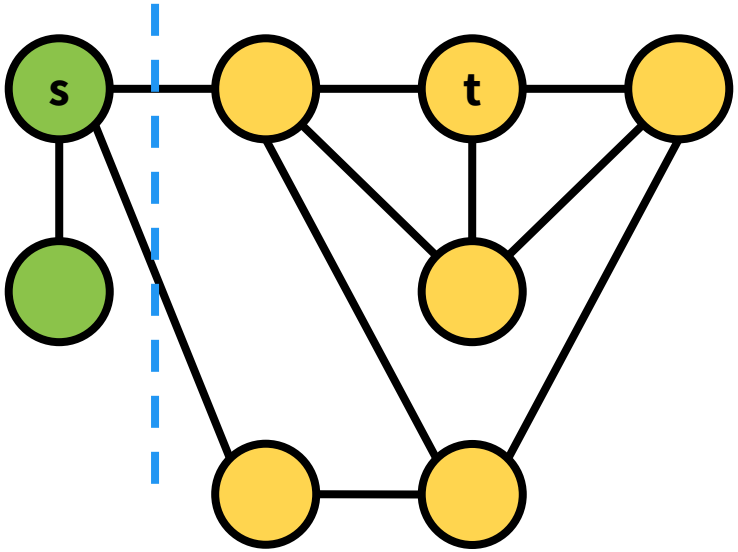
e.g. The global minimum cut is “{A, B, C, D, F, G, H} and {E}”.



Cuts

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

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



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.

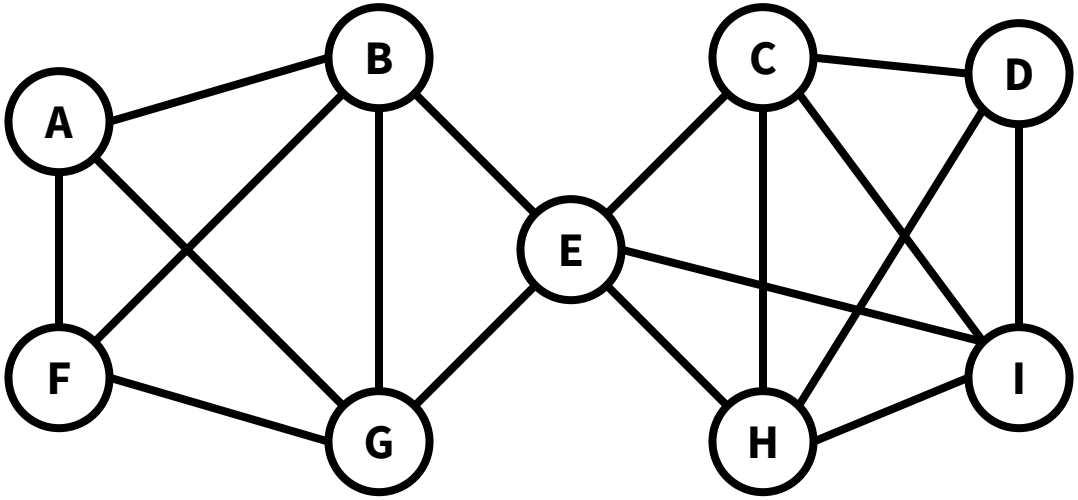
For all inputs A, `quicksort` returns a sorted list. For all inputs A, with high probability over the choice of pivots, `quicksort` runs fast.

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.

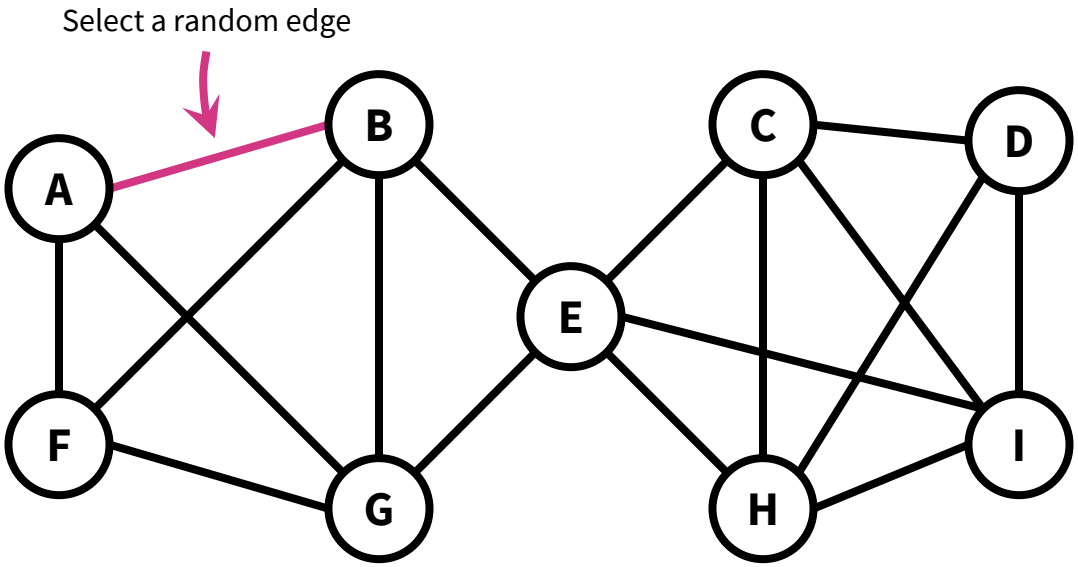
Karger's Algorithm

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

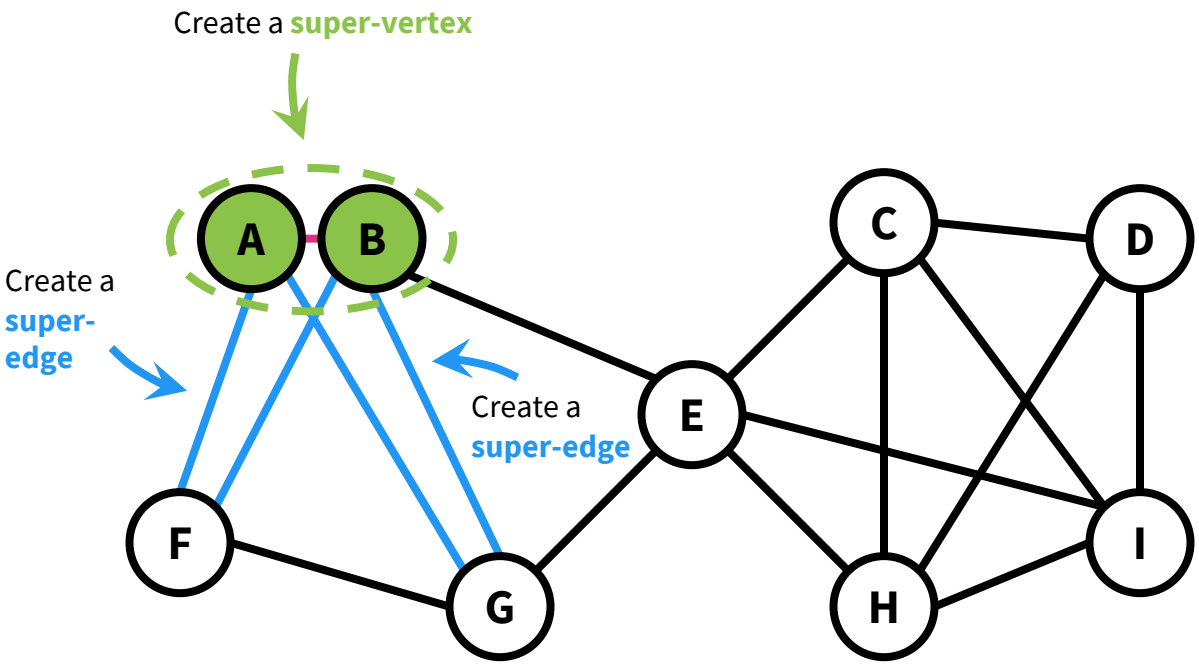
Karger's Algorithm



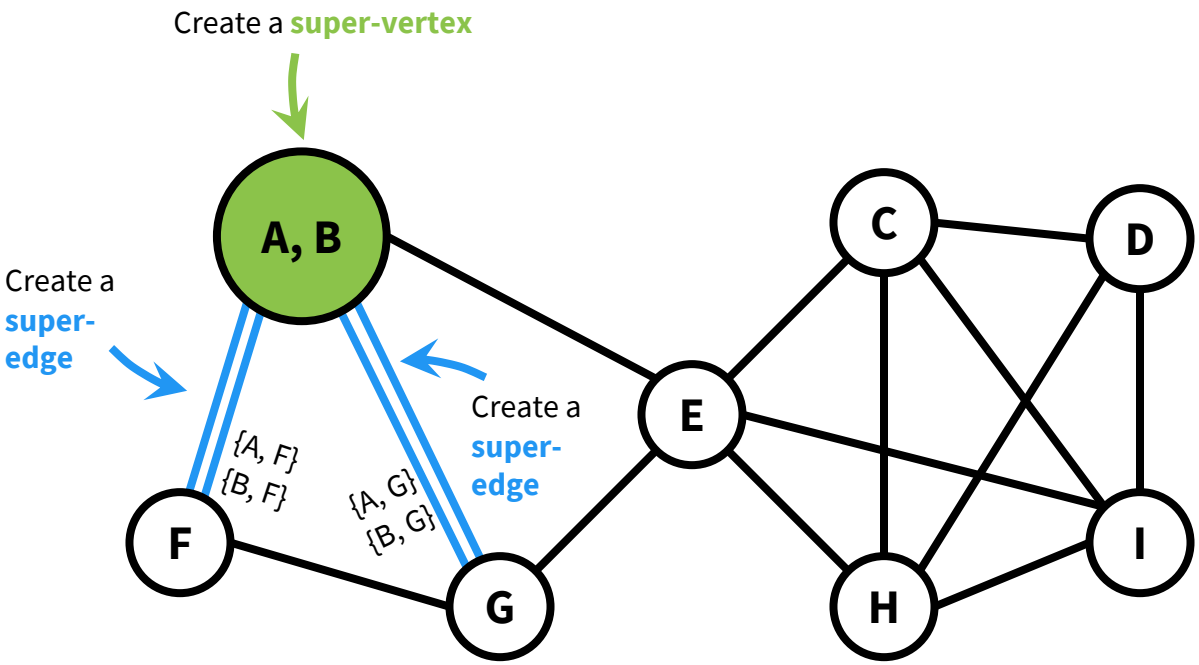
Karger's Algorithm



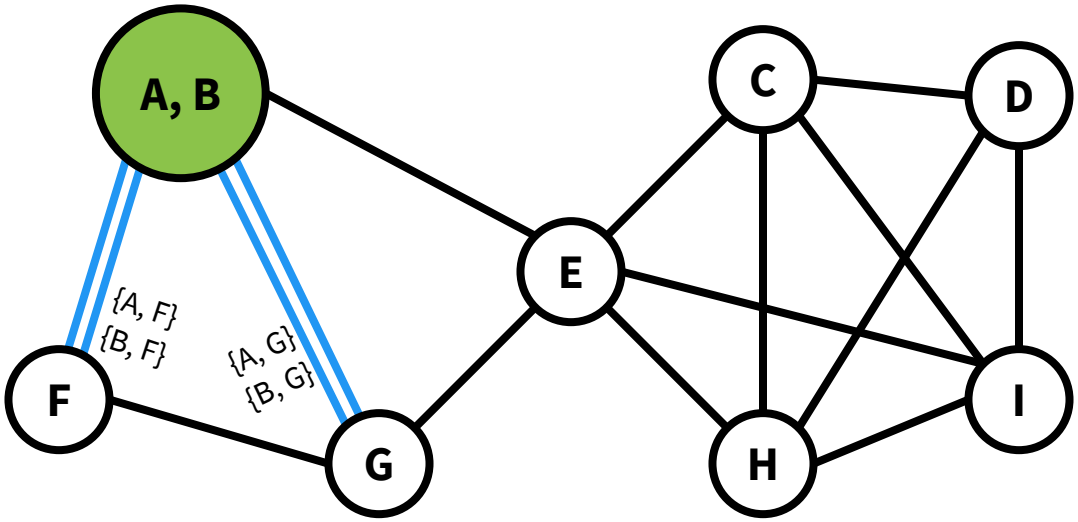
Karger's Algorithm



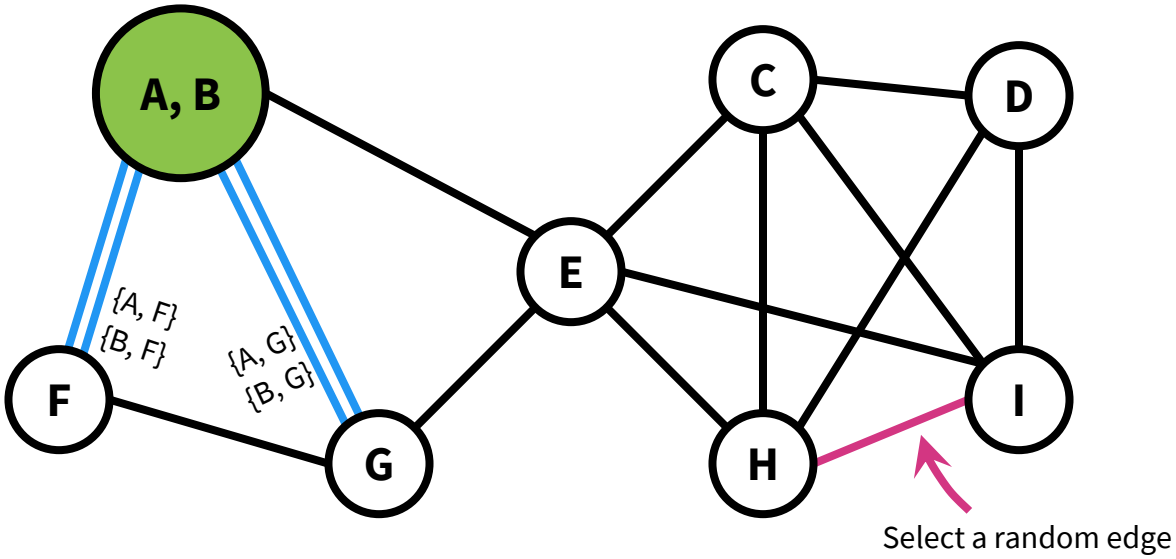
Karger's Algorithm



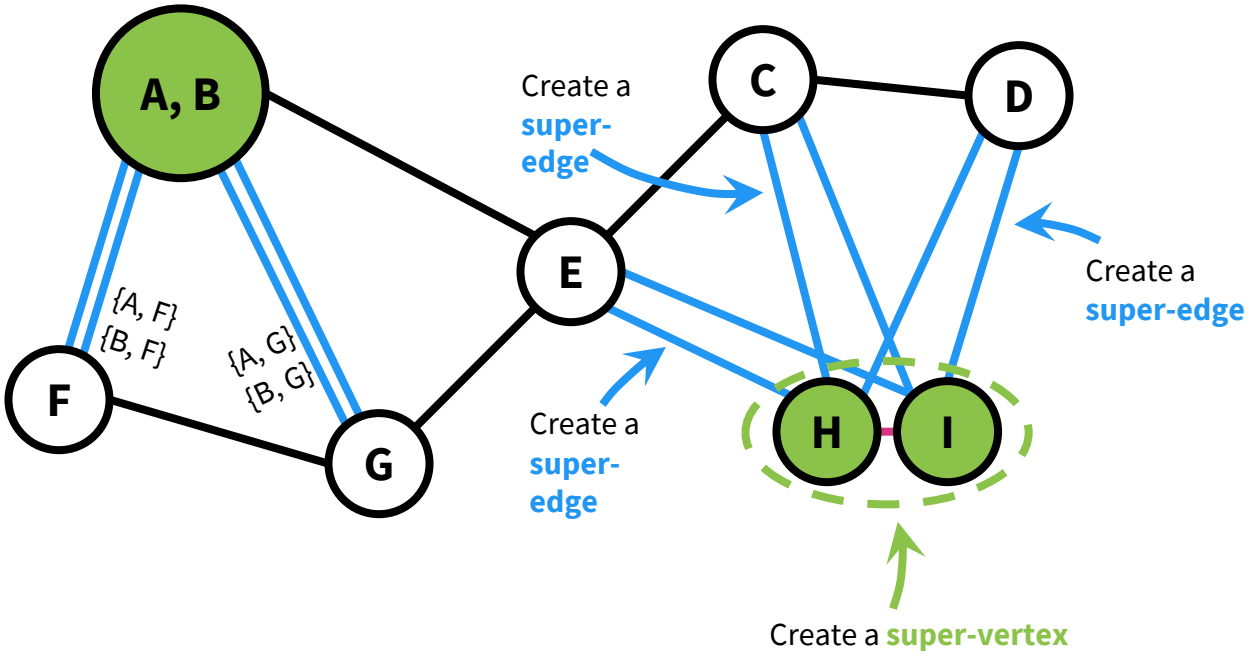
Karger's Algorithm



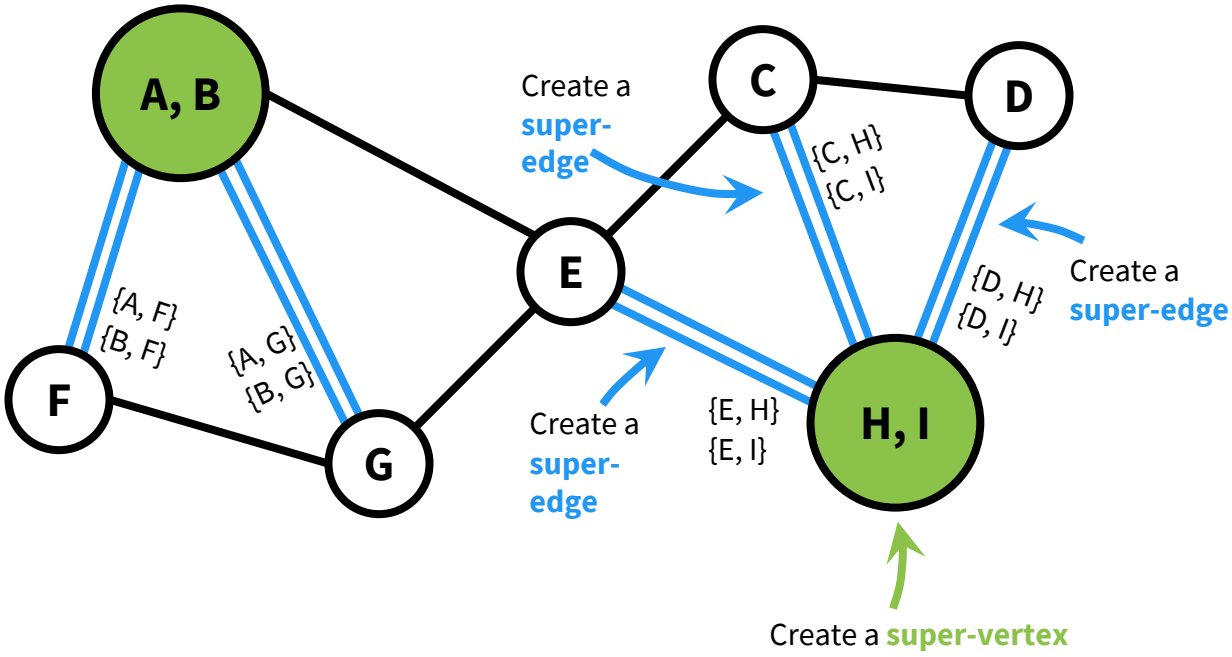
Karger's Algorithm



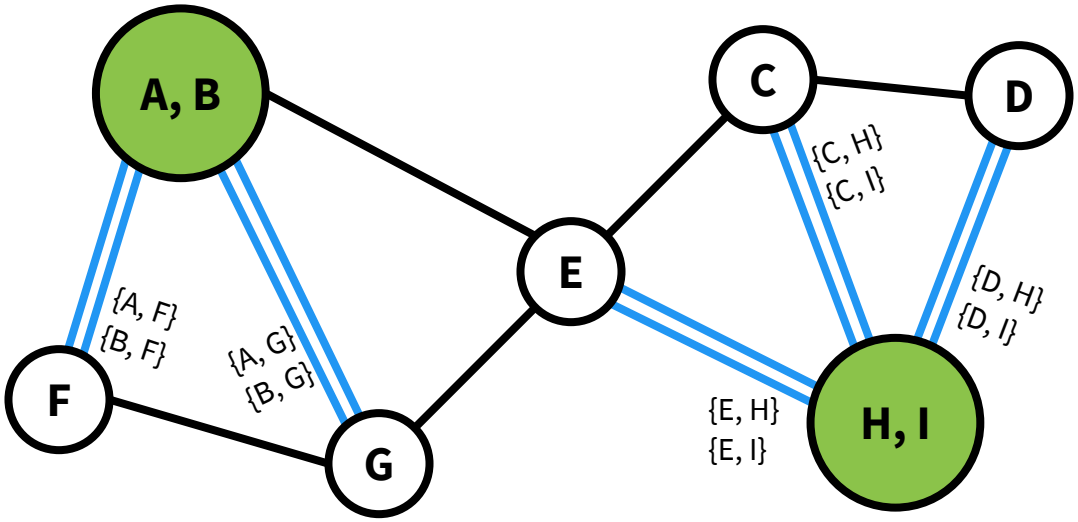
Karger's Algorithm



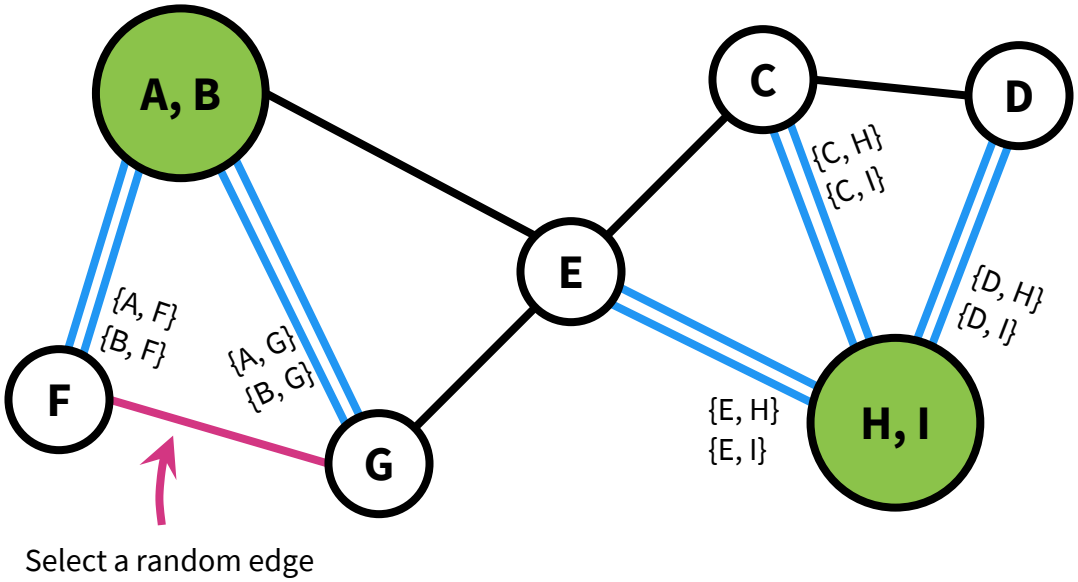
Karger's Algorithm



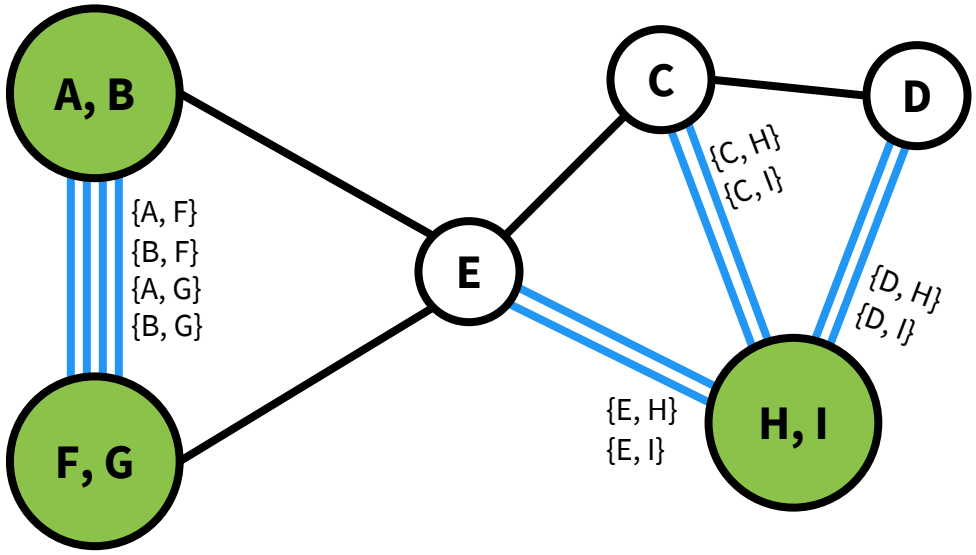
Karger's Algorithm



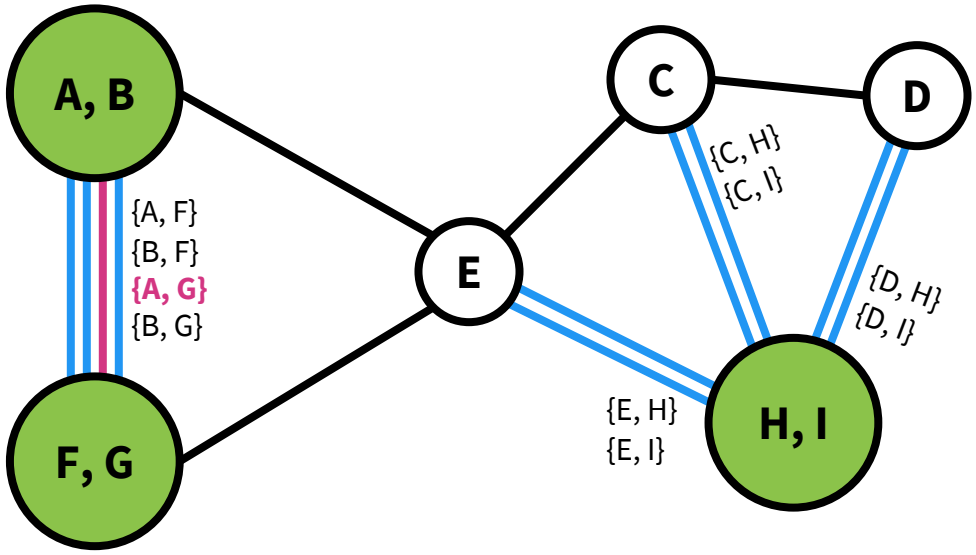
Karger's Algorithm



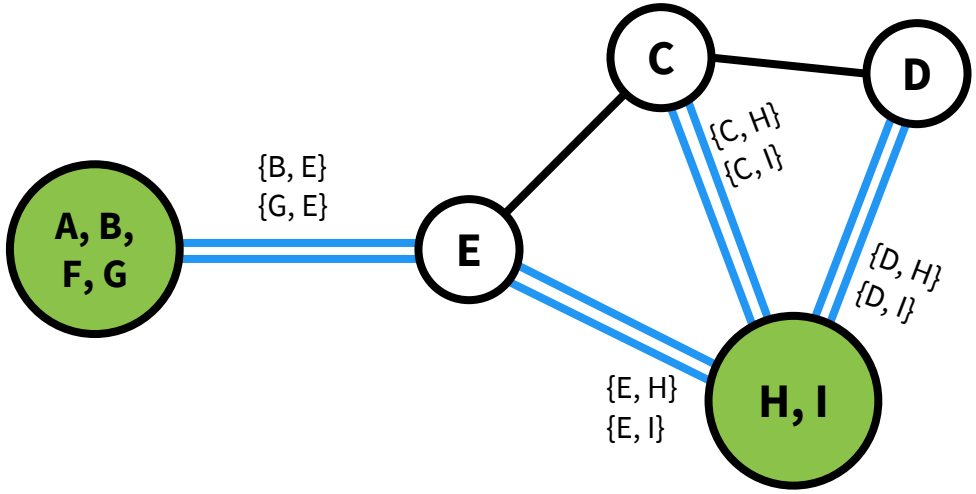
Karger's Algorithm



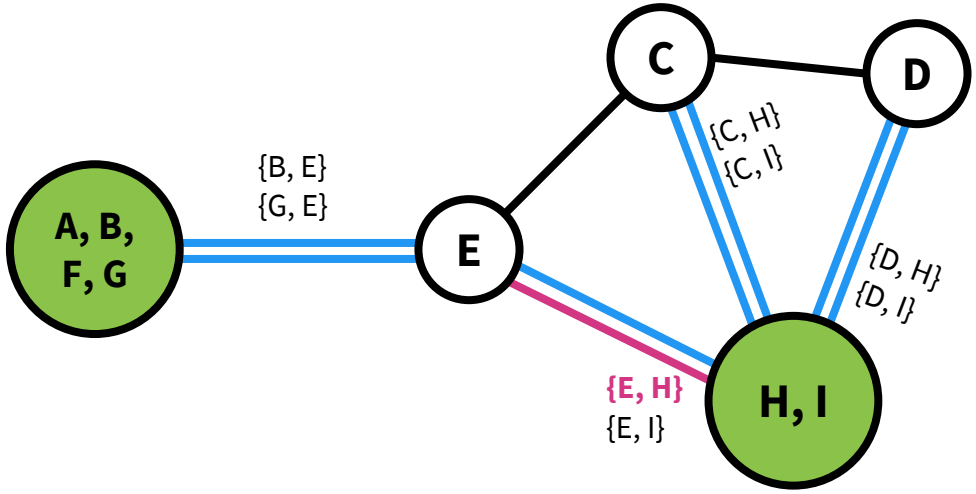
Karger's Algorithm



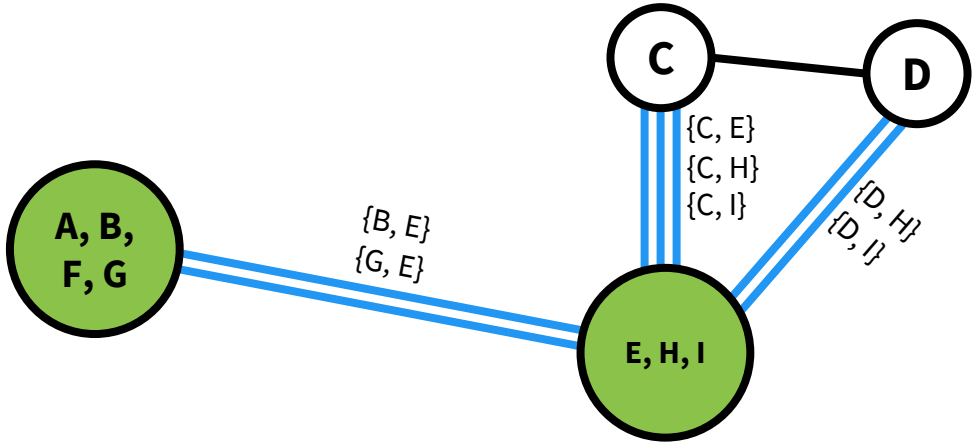
Karger's Algorithm



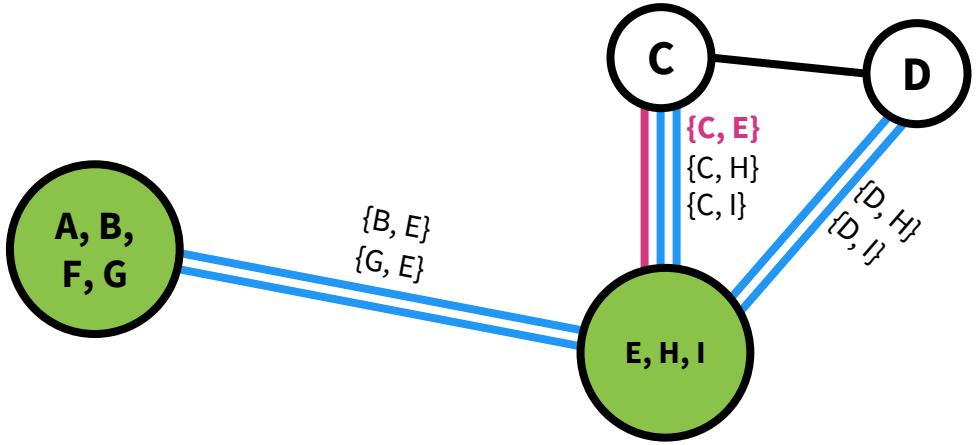
Karger's Algorithm



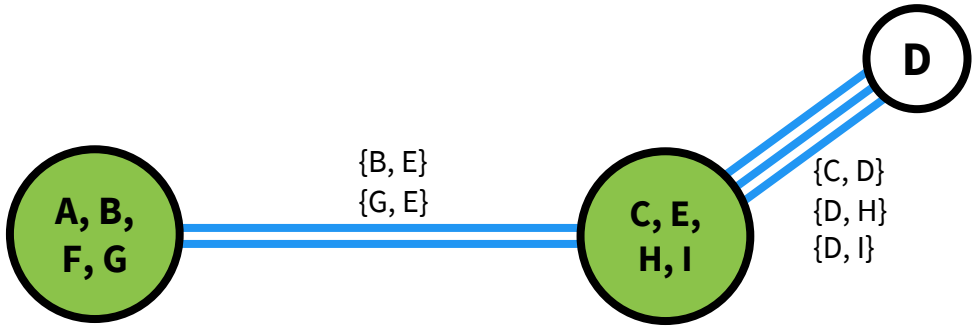
Karger's Algorithm



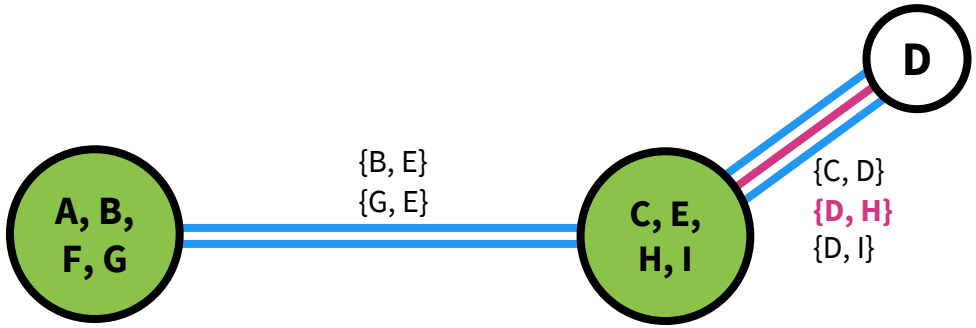
Karger's Algorithm



Karger's Algorithm



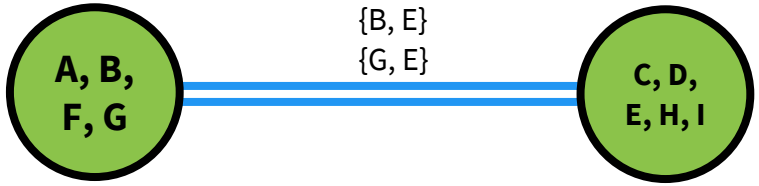
Karger's Algorithm



Karger's Algorithm

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

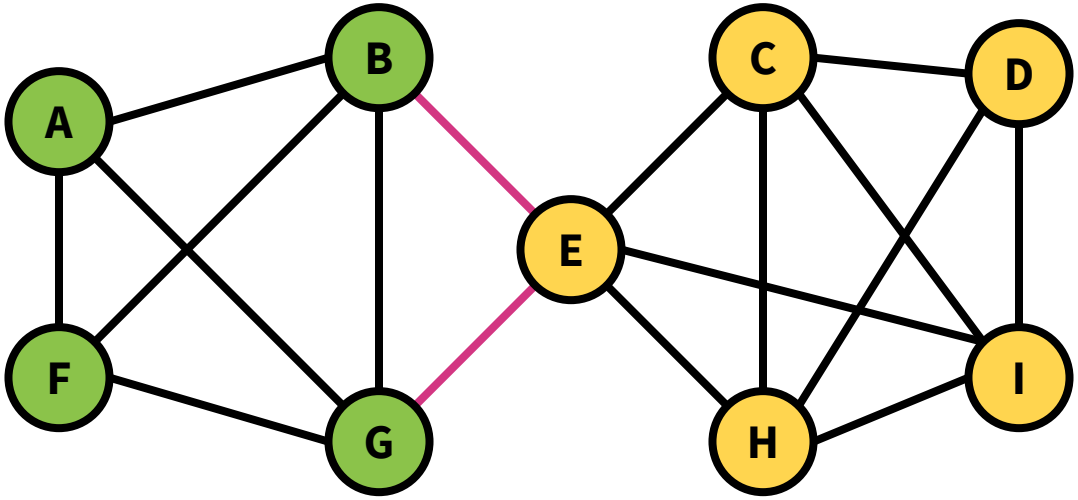
e.g. The cut is "{A, B, F, G} and {C, D, E, H, I}"; the edges that cross this cut are {B, E} and {G, E}.



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) in 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'
```

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.

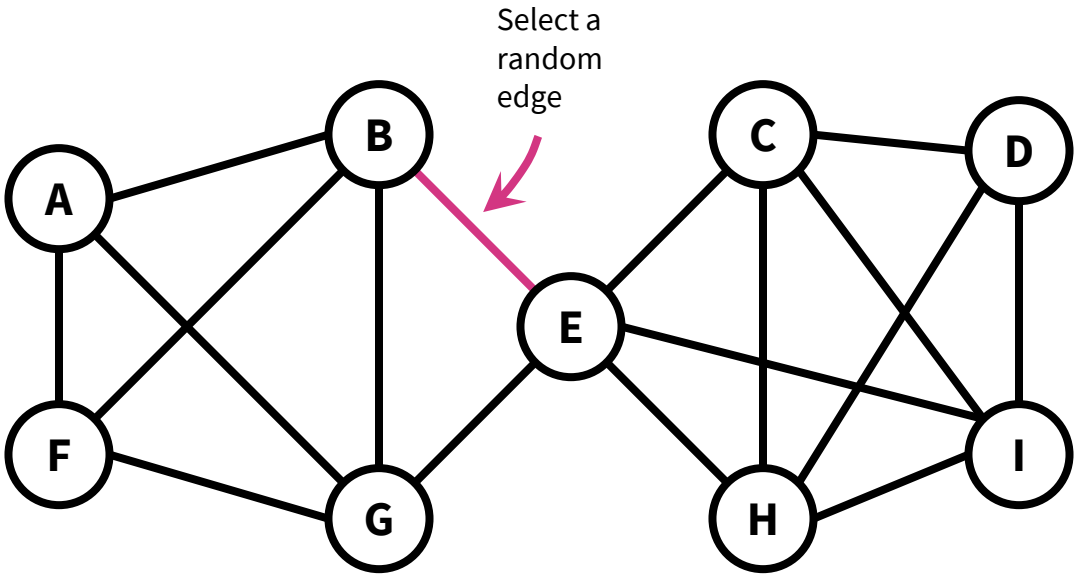
Runtime: $O(|V|^2)$

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

Karger's Algorithm

We got really lucky!

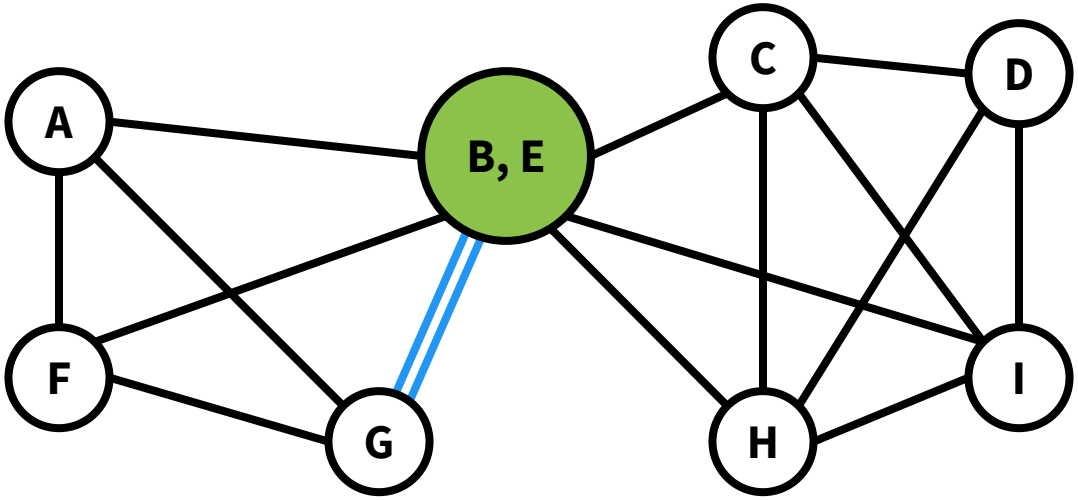
e.g. Suppose we had chosen this edge.



Karger's Algorithm

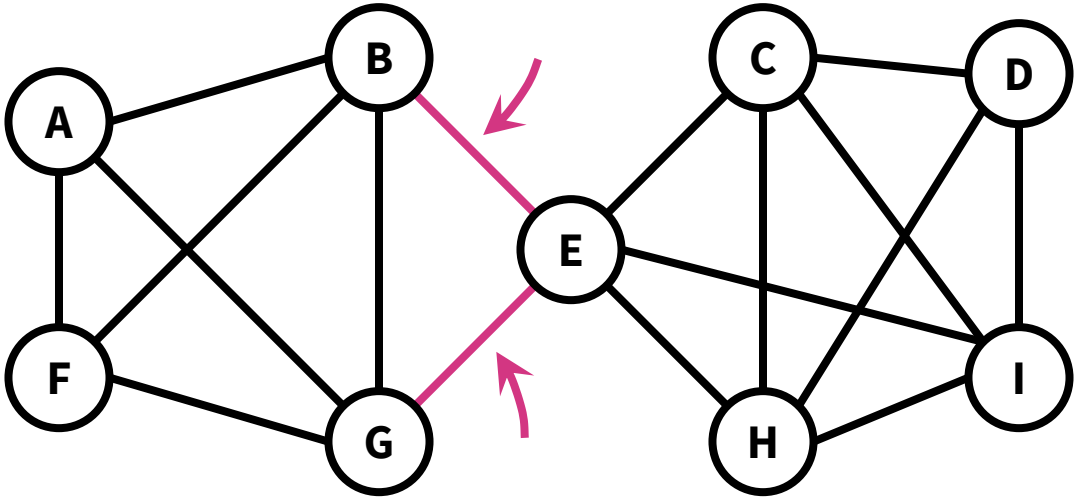
We got really lucky!

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 …

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

where n is the number of nodes.

Proof:

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

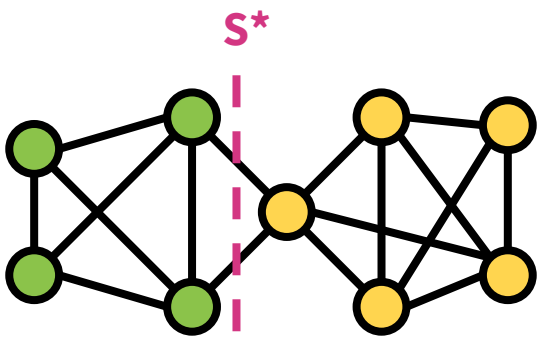
Then P(karger returns S*) = P(no e_i crosses S*)

= 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?

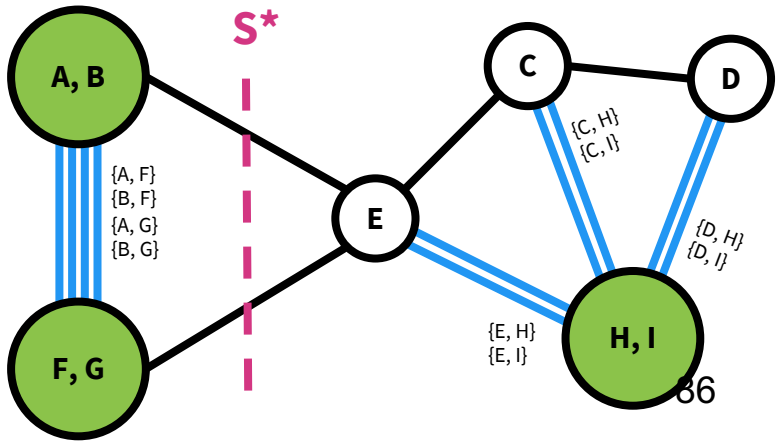
Suppose there are k edges that cross **S***.

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 $e_1, e_2, \ldots, e_{n-2}$.

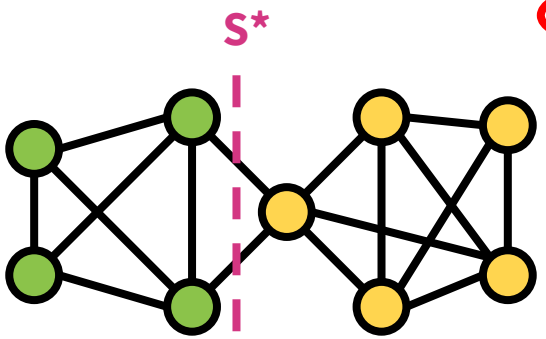
Then P(karger returns S*) = P(e_1 doesn't cross S*)

× 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*)

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



= 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.

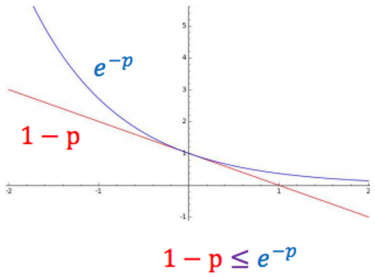
Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

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

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. 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

Key point: 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!

Many statistical machine learning algorithms work in this way!

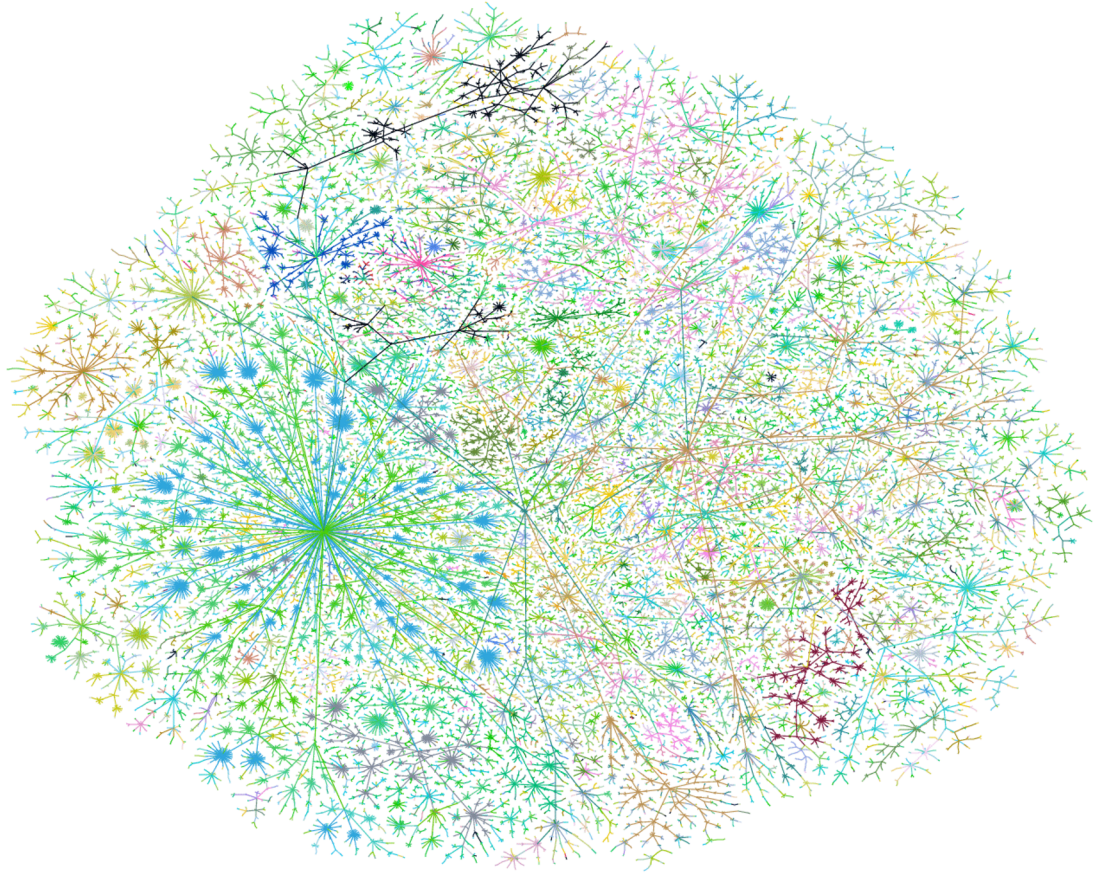
Outline for Today

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

Graph Basics

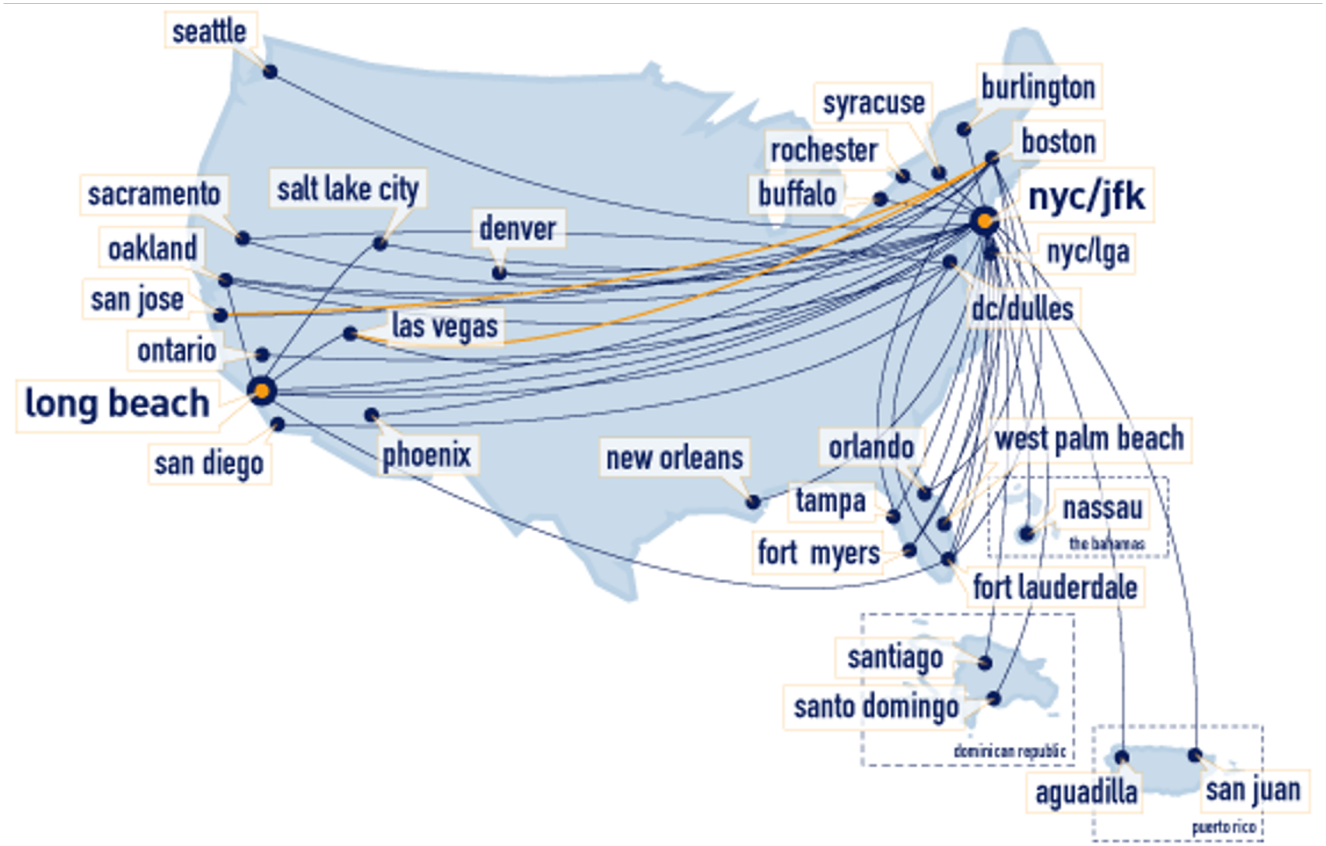
Examples of Graphs

The Internet (circa 1999)



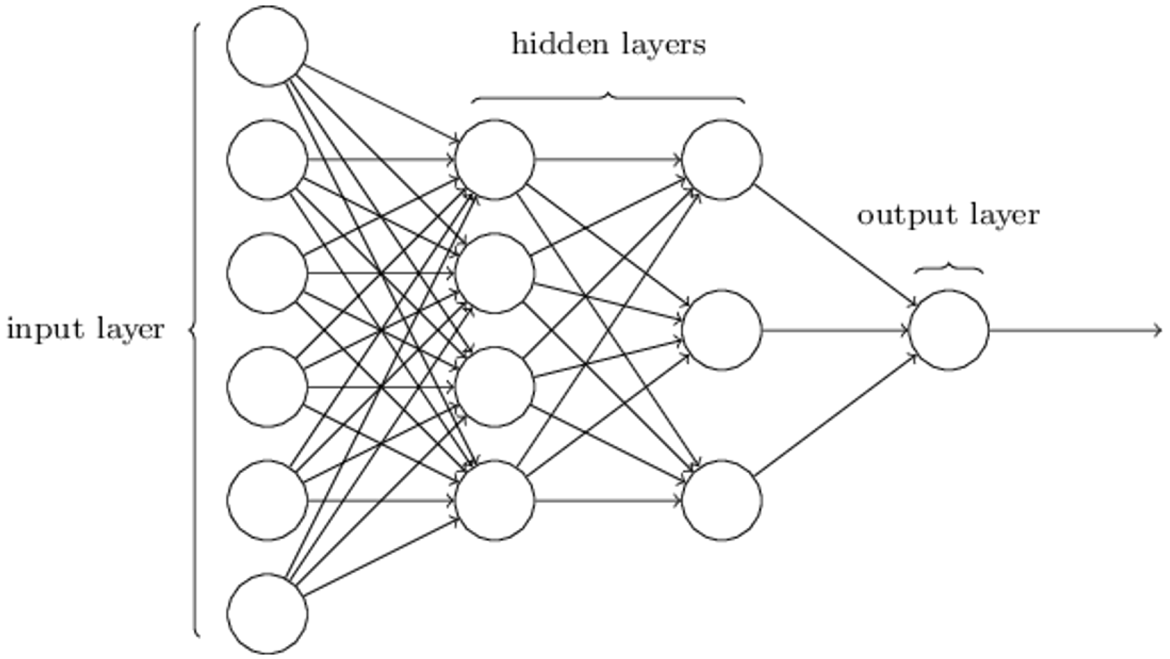
Examples of Graphs

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 for efficient routing.

Finding strongly connected components for community detection or clustering.

Finding the topological ordering to respect dependencies.

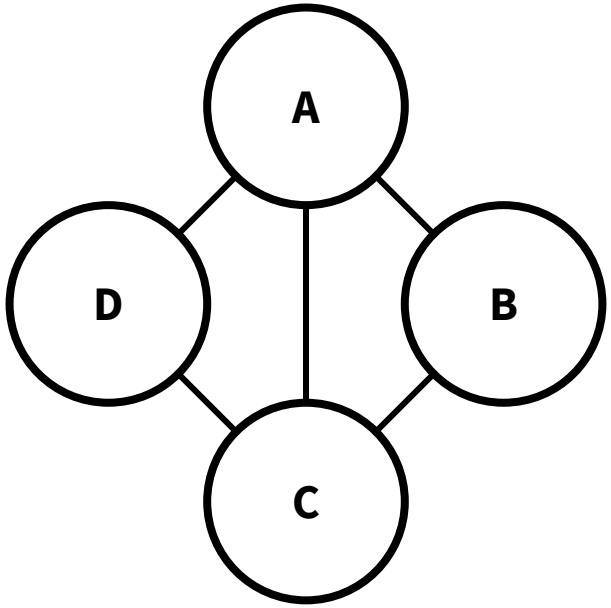
Undirected Graphs

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} }



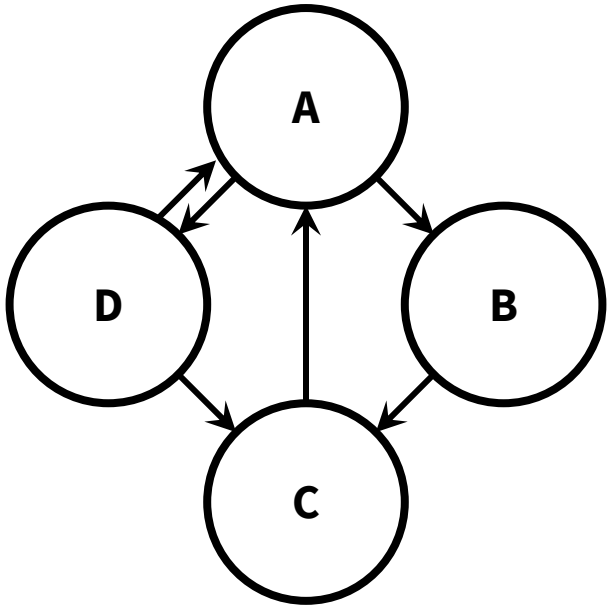
Directed Graphs

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)

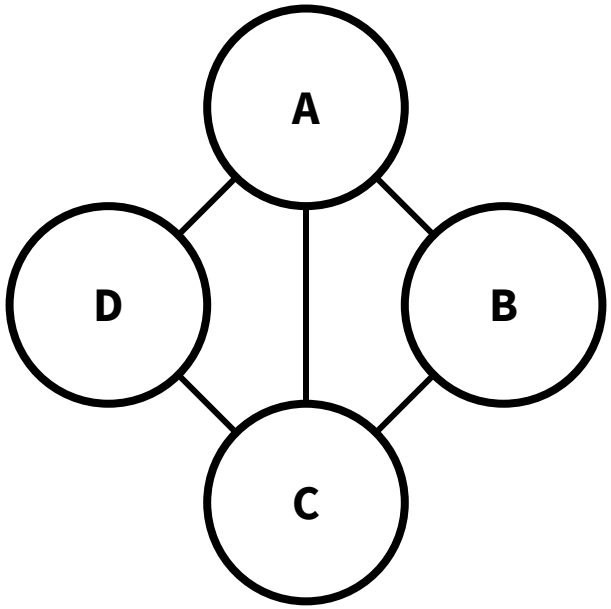
e.g. V = {A, B, C, D} and E = { [A, B], [A, D], [B, C], [C, A], [D, A], [D, C] }



Graph Representations

How do we represent graphs?

(1) Adjacency matrix



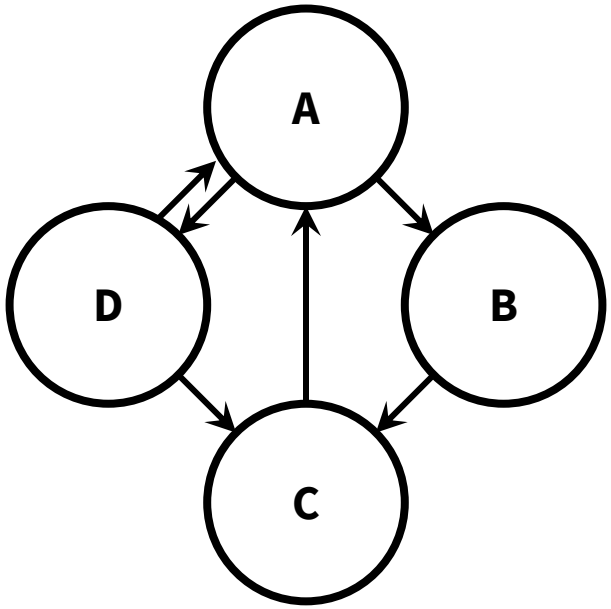
	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	0

Symmetric matrix

Graph Representations

How do we represent graphs?

(1) Adjacency matrix



destination

source	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	1	0	0	0
D	1	0	1	0

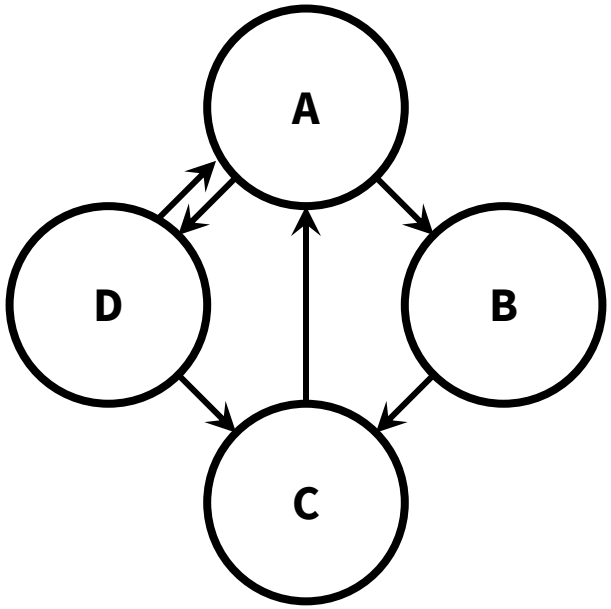
Unsymmetric matrix

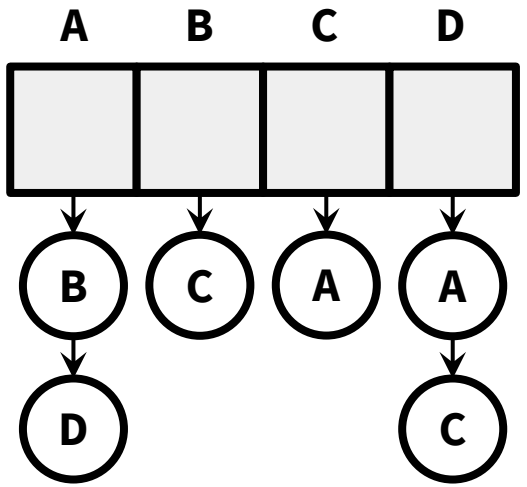
Graph Representations

How do we represent graphs?

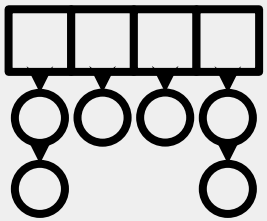
(1) Adjacency matrix

(2) Adjacency list





Graph Representations

For G = (V, E)	Adjacent matrix	Adjacent list
	$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$	
Edge Membership Is e = [u, v] in E?	$O(1)$	$O(\deg(u))$
Neighbor Query What are the neighbors of u?	$O(\lvert V \rvert)$	$O(\deg(u))$
Space requirements	$O(\lvert V \rvert^2)$	$O(\lvert V \rvert + \lvert E \rvert)$

Generally, better for sparse graphs.

We'll assume this representation, unless otherwise stated.

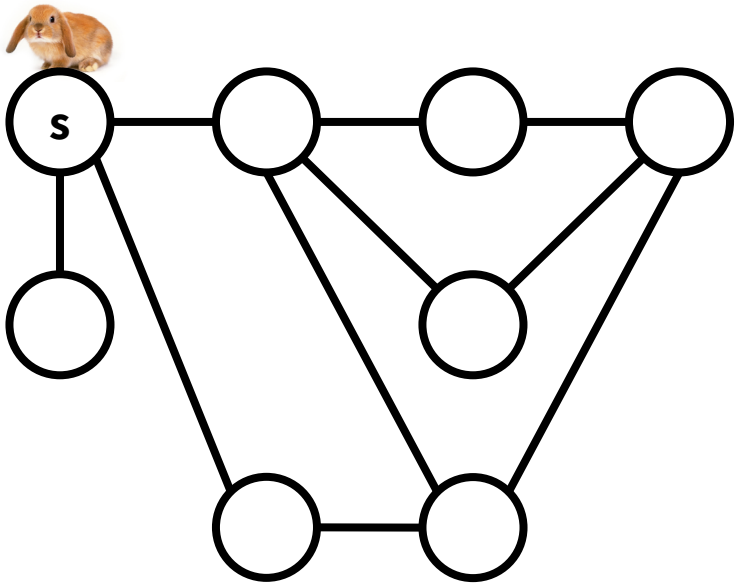
Explain with directed graph; relationship between the two representations; an example in e-commerce

Depth-First Search

Depth-First Search

An analogy

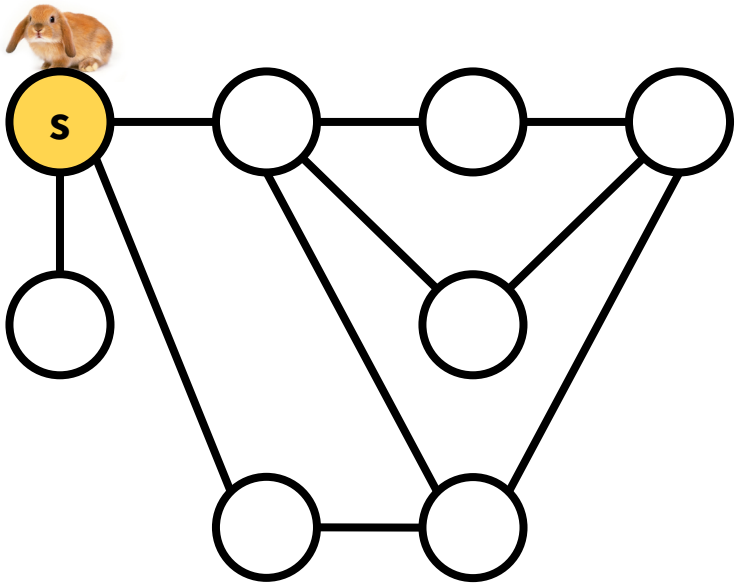
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

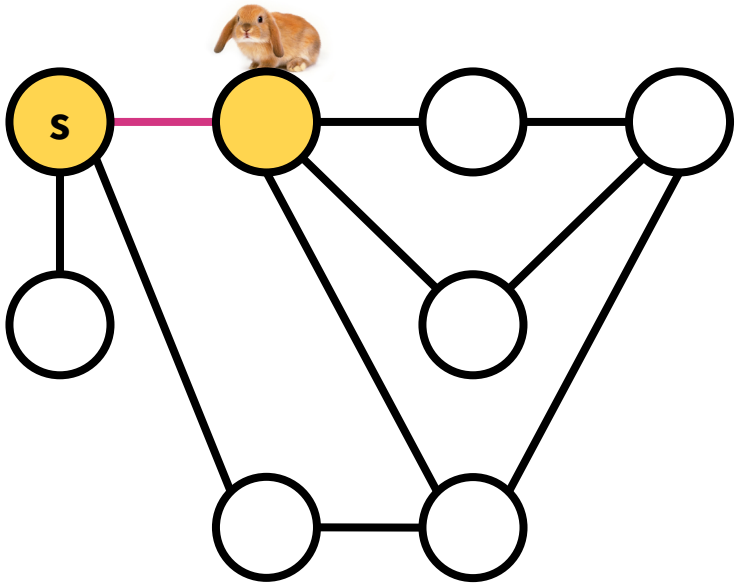
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

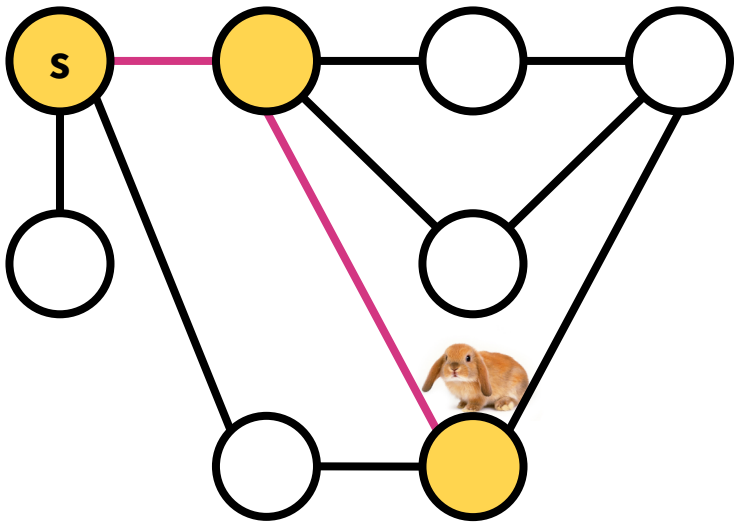
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

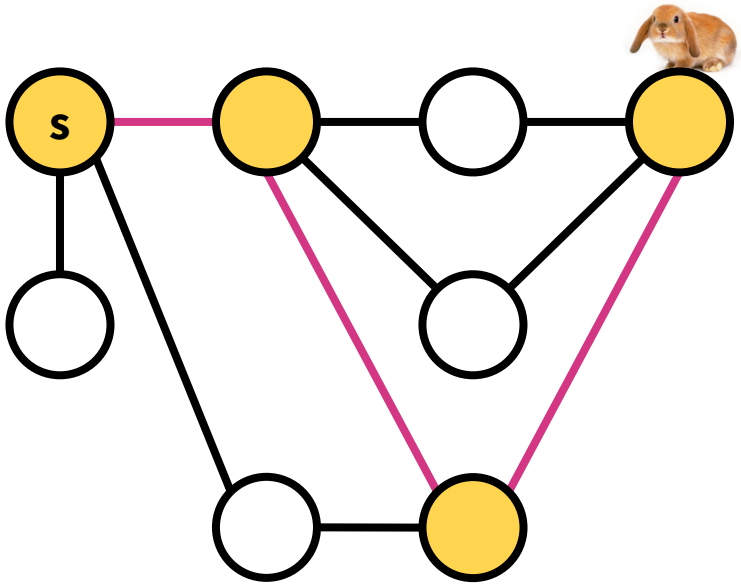
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

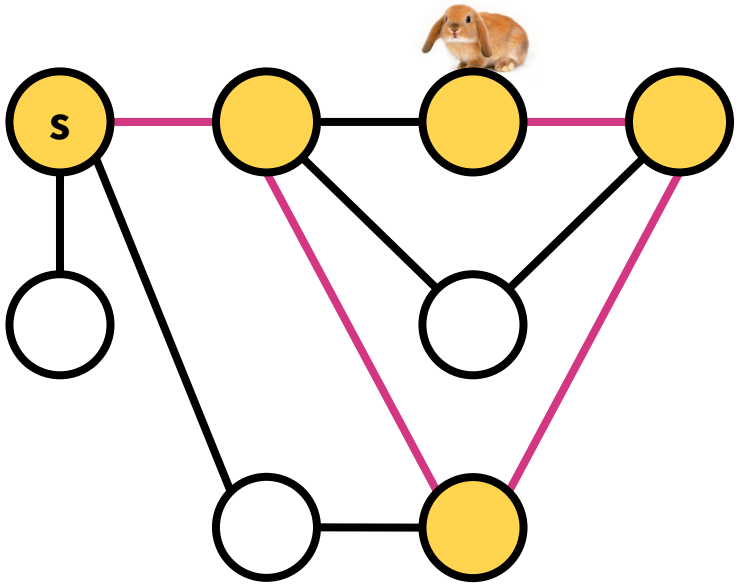
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

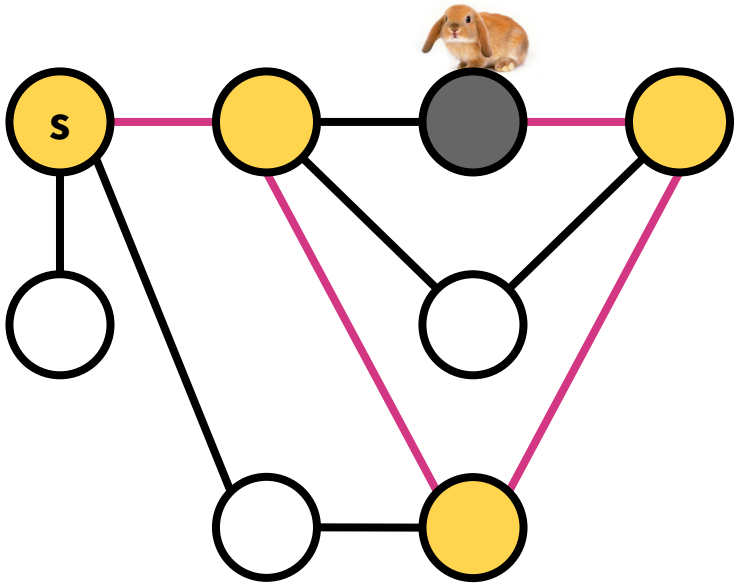
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

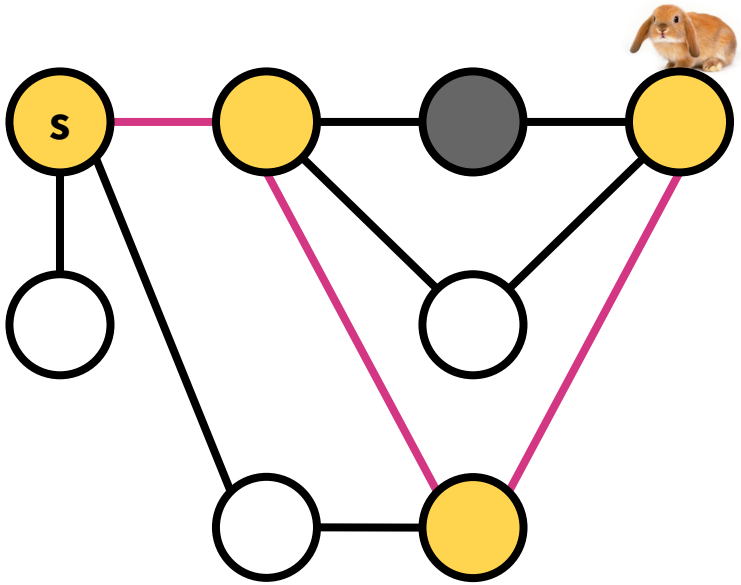
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

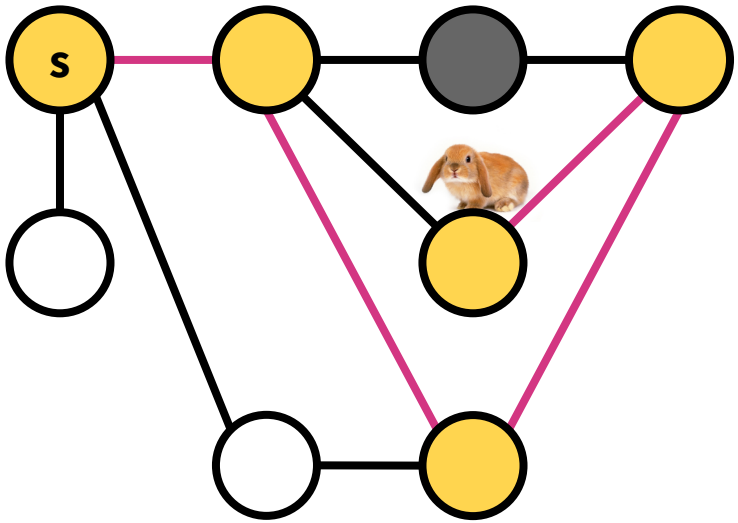
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

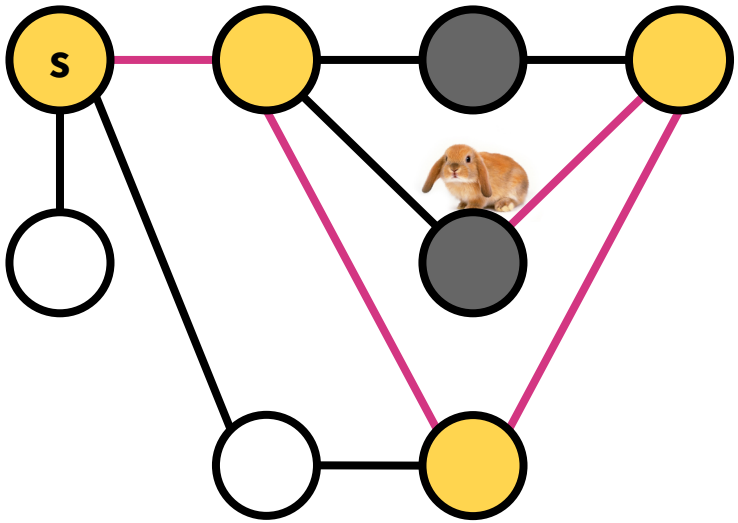
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

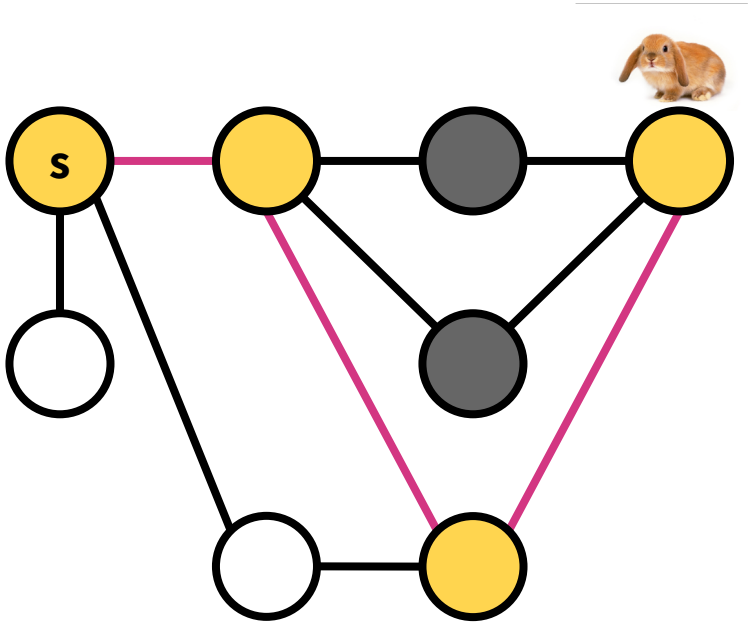
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

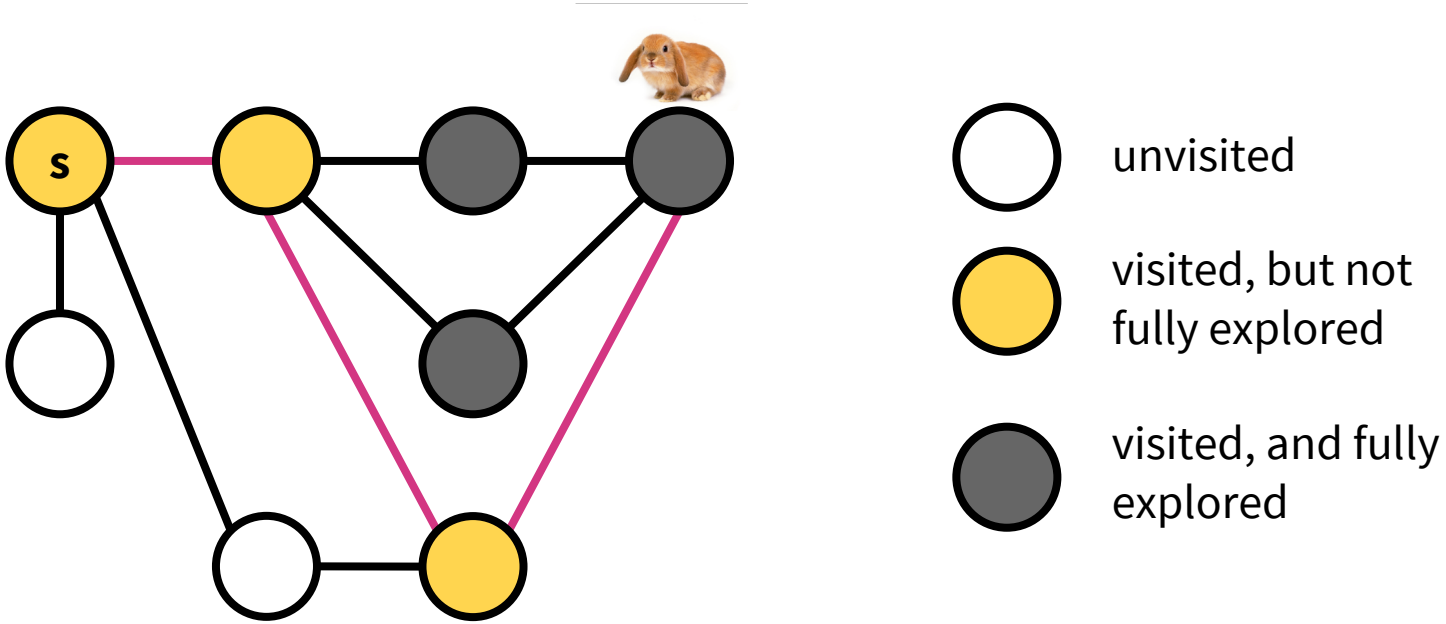
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

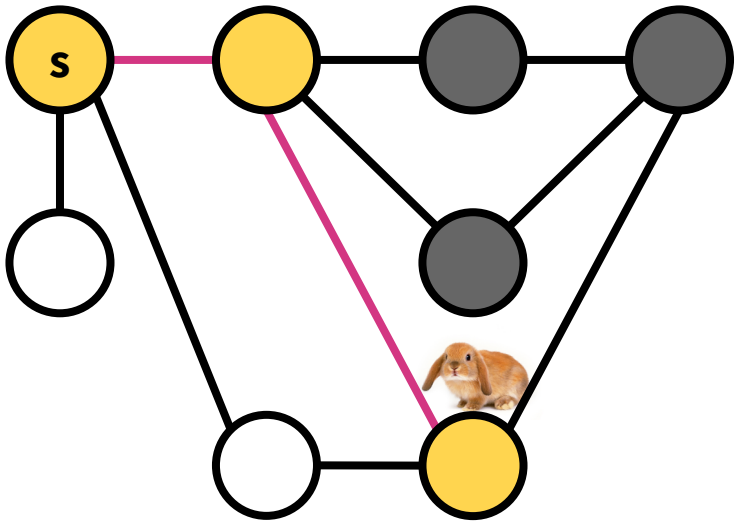
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

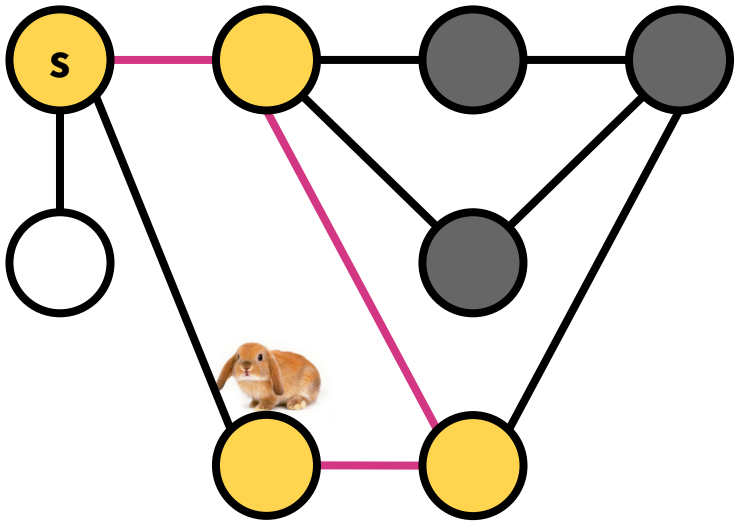
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

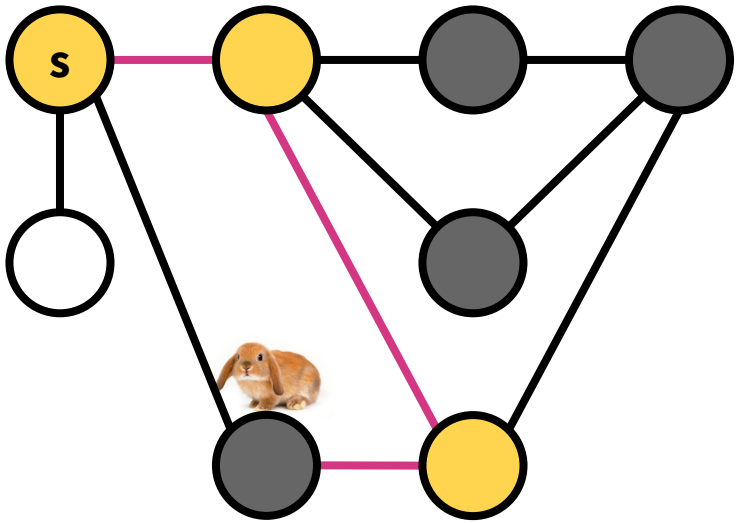
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

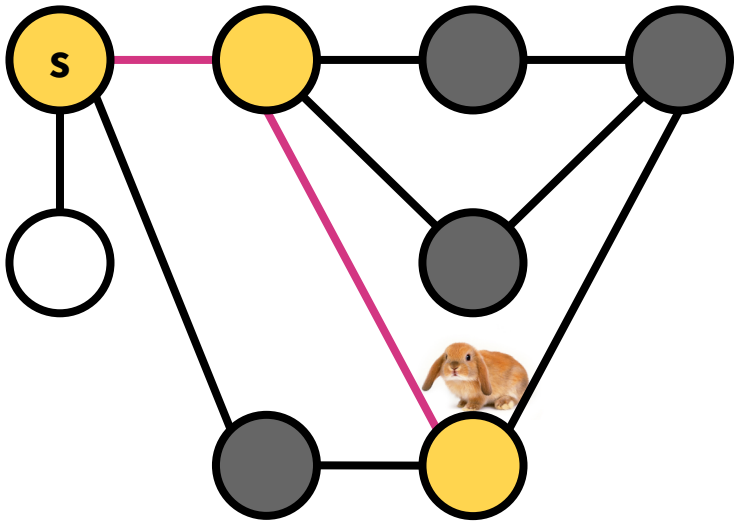
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

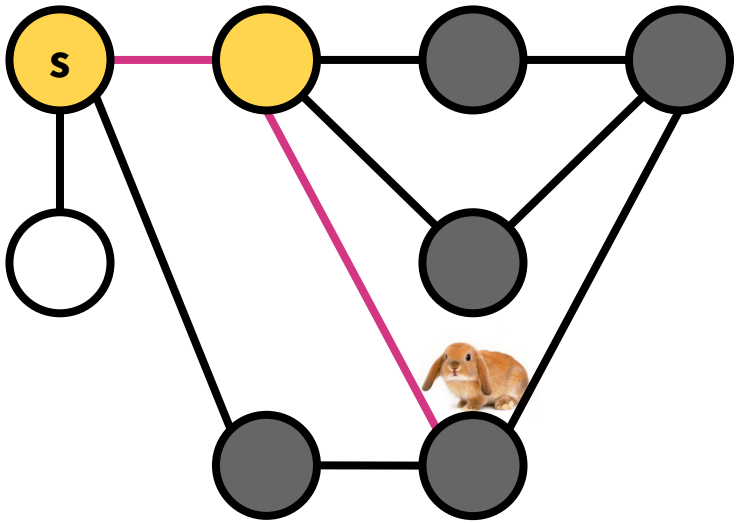
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

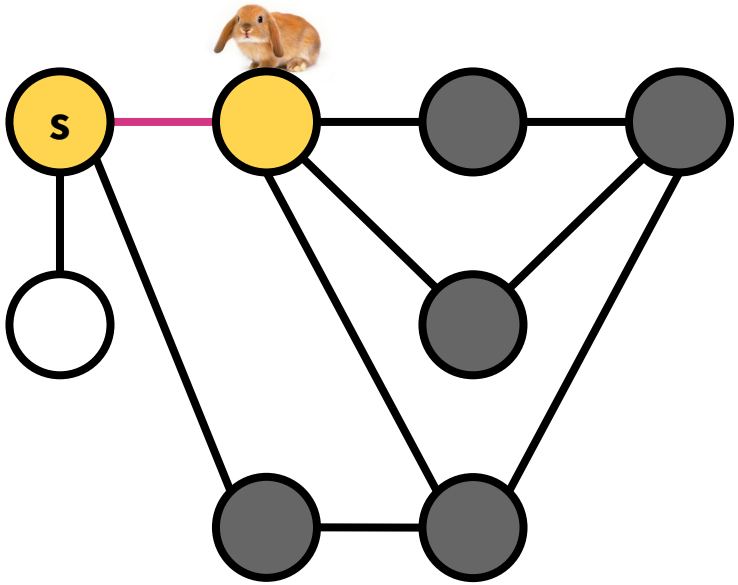
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

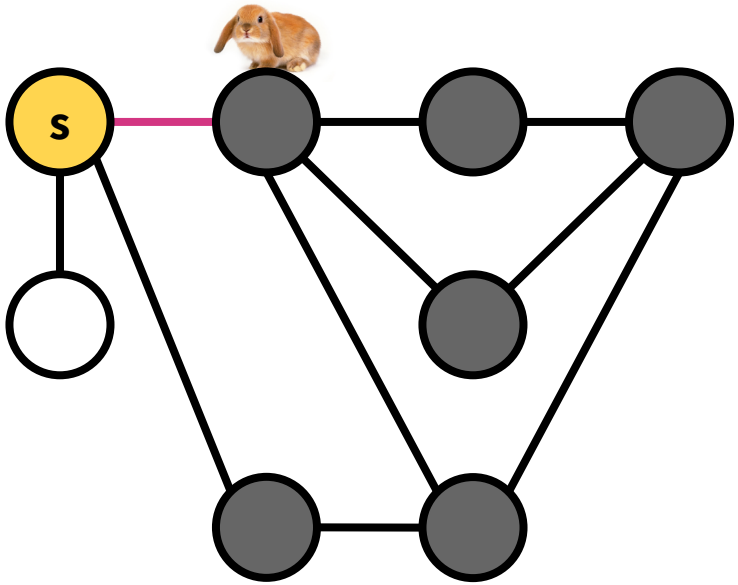
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

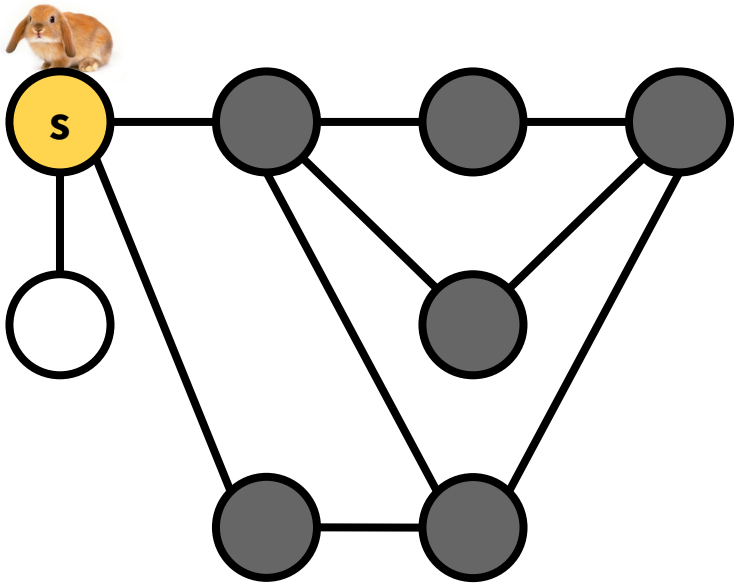
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

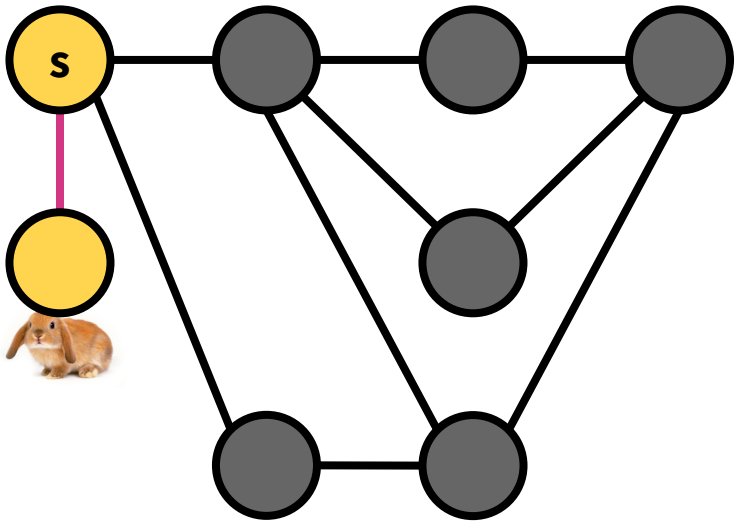
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

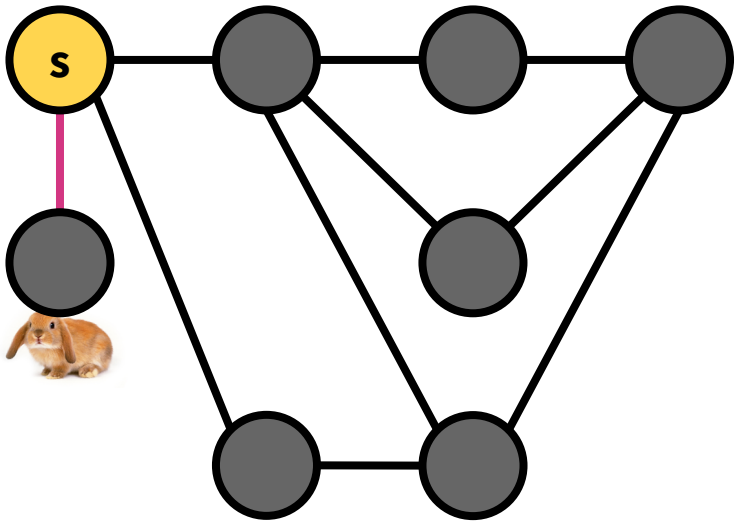
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

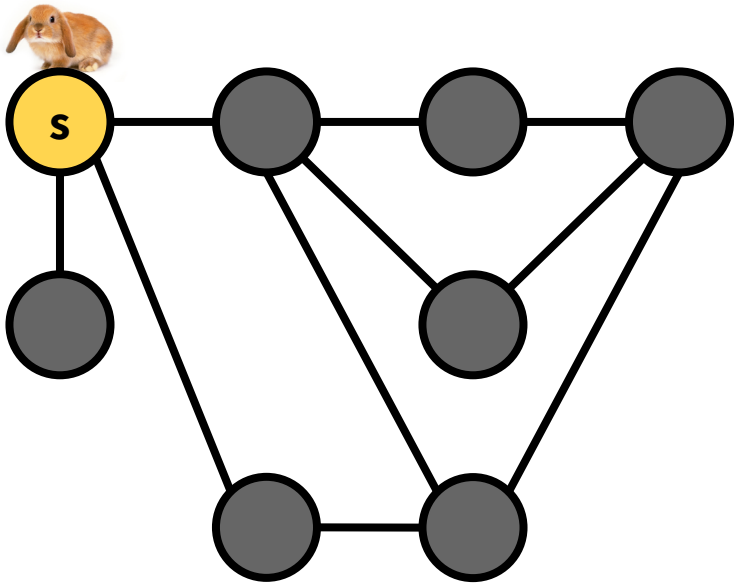
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

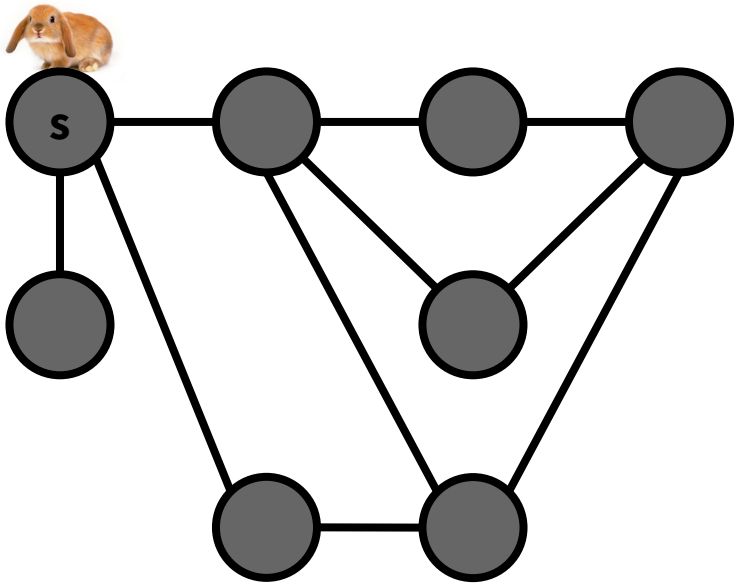
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

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

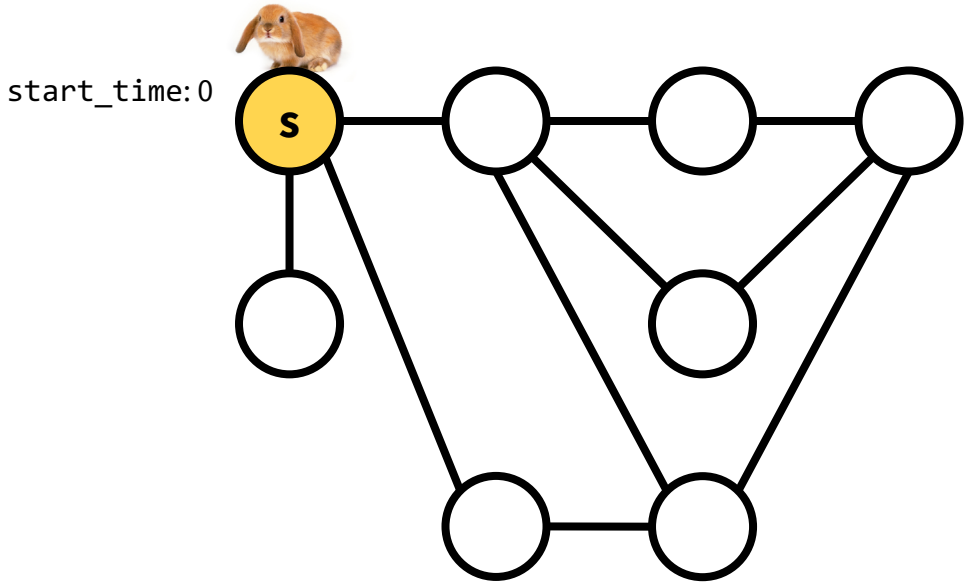


Depth-First Search

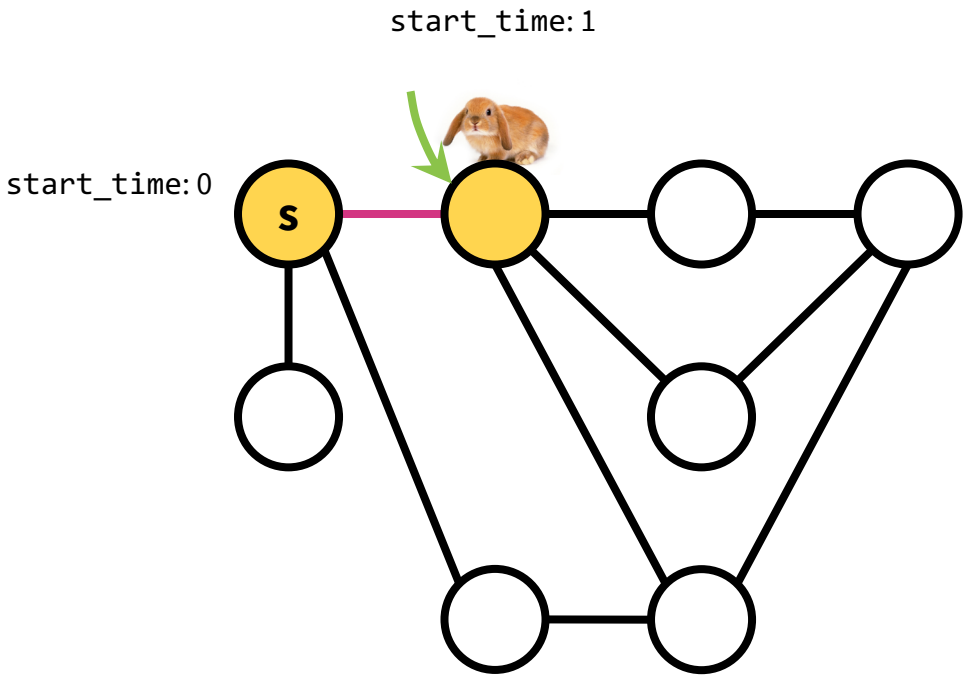
```
algorithm dfs(u, cur_time):
  u.start_time = cur_time
  cur_time += 1
  u.status = "in_progress"
  for v in u.neighbors:
    if v.status is "unvisited":
       cur_time = dfs(v, cur_time)
       cur_time += 1
  u.end_time = cur_time
  u.status = "done"
  return cur_time
```

Runtime: $O(2|V|+2|E|)=O(|V|+|E|)$

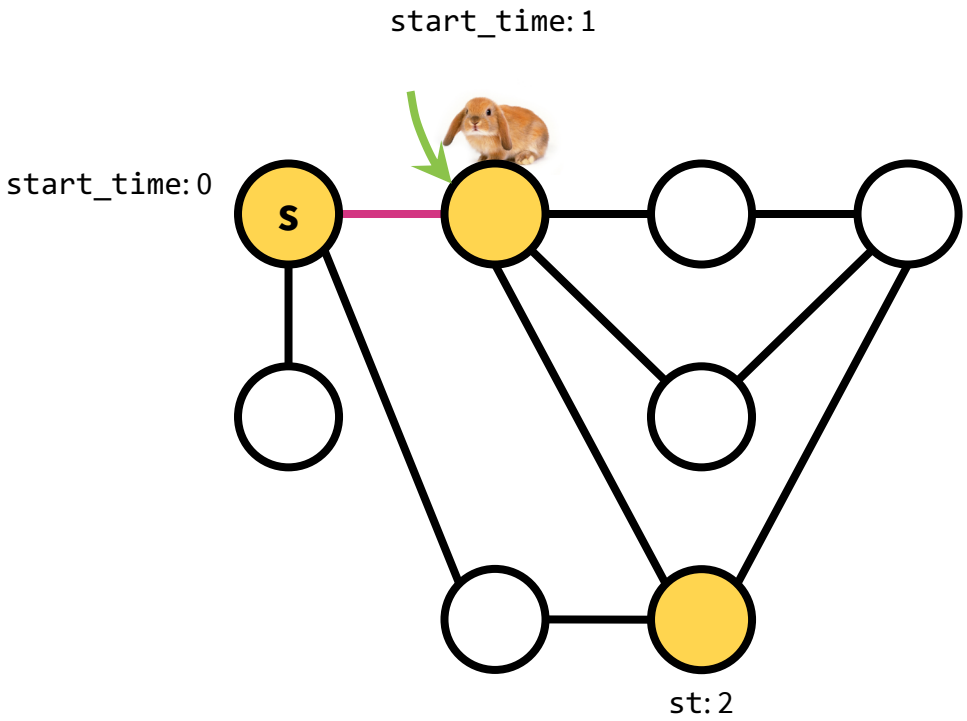
Depth-First Search



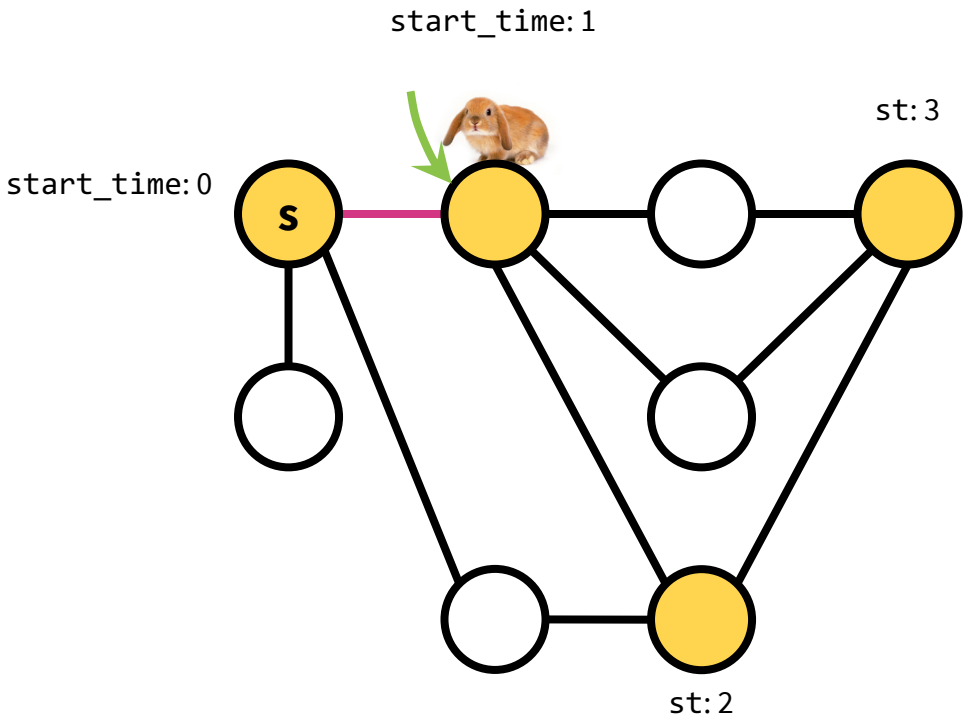
Depth-First Search



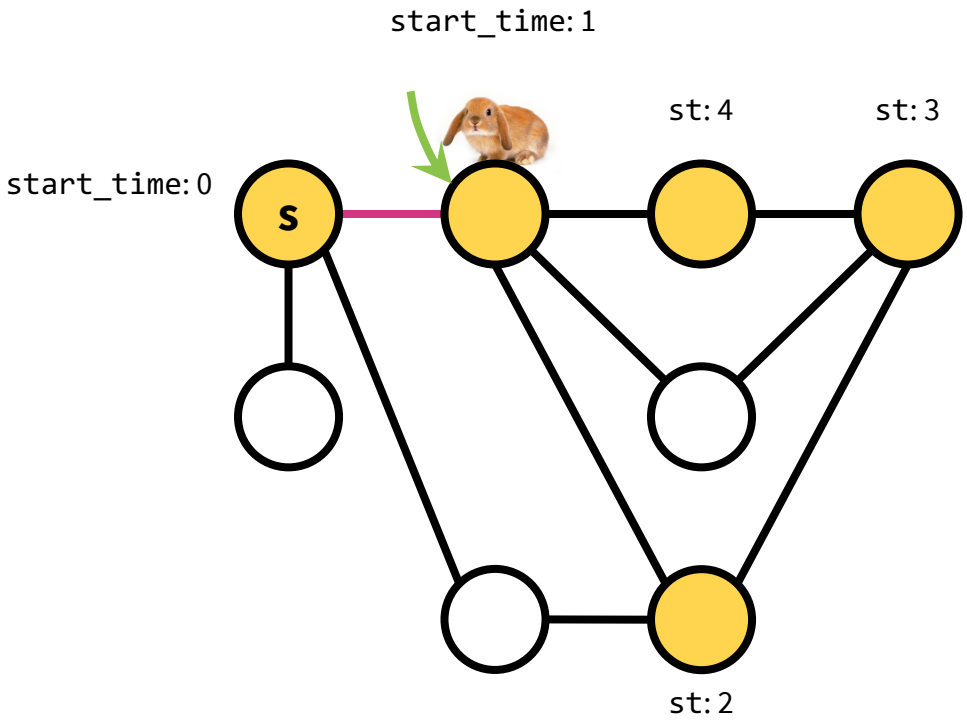
Depth-First Search



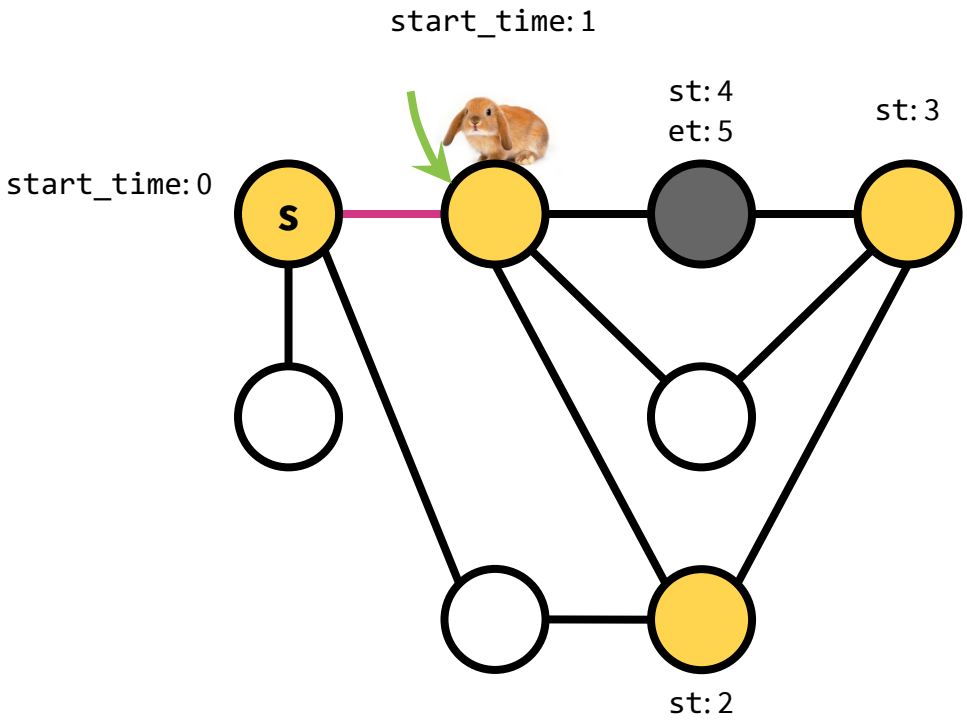
Depth-First Search



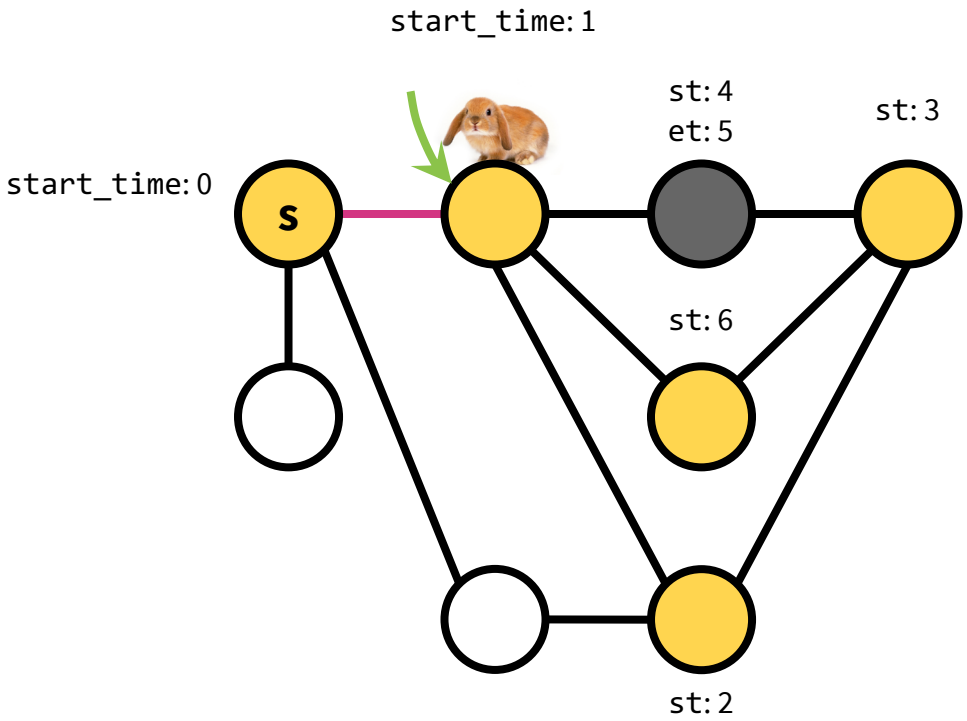
Depth-First Search



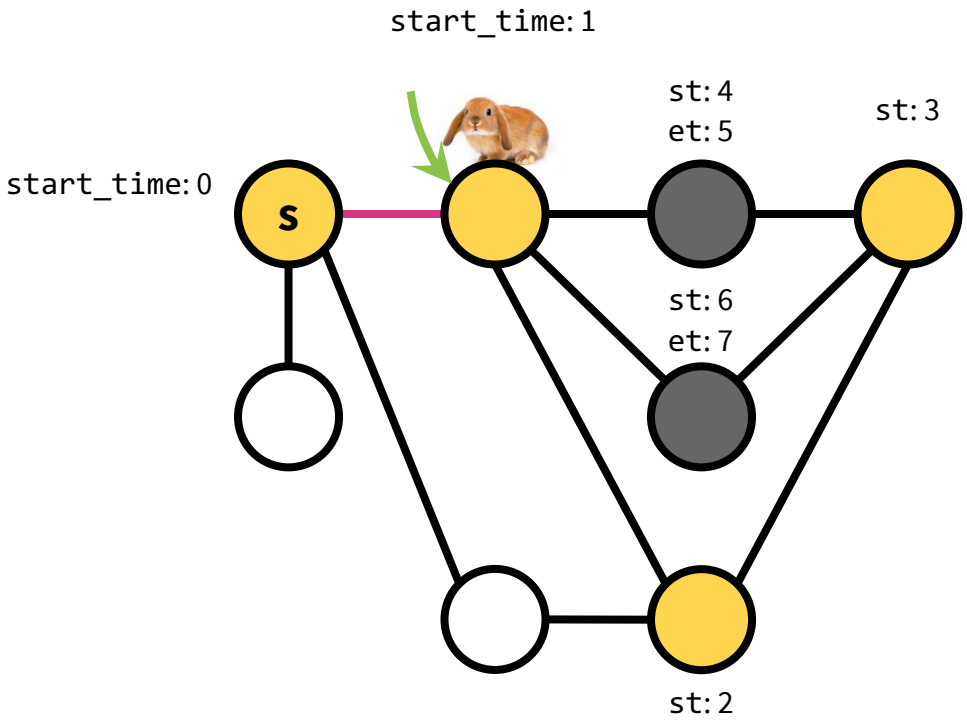
Depth-First Search



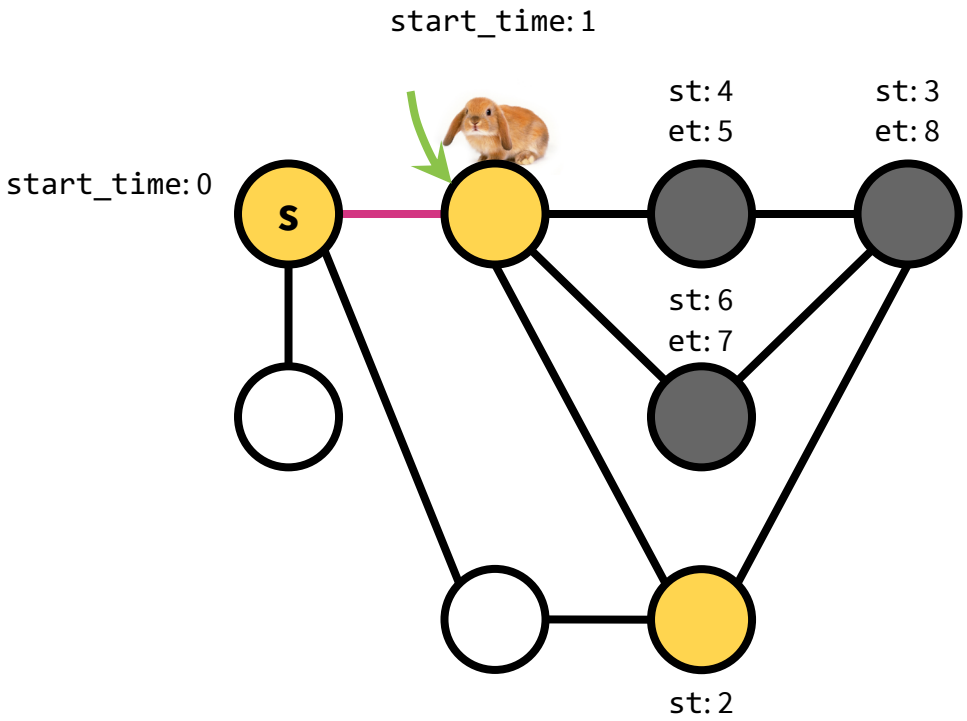
Depth-First Search



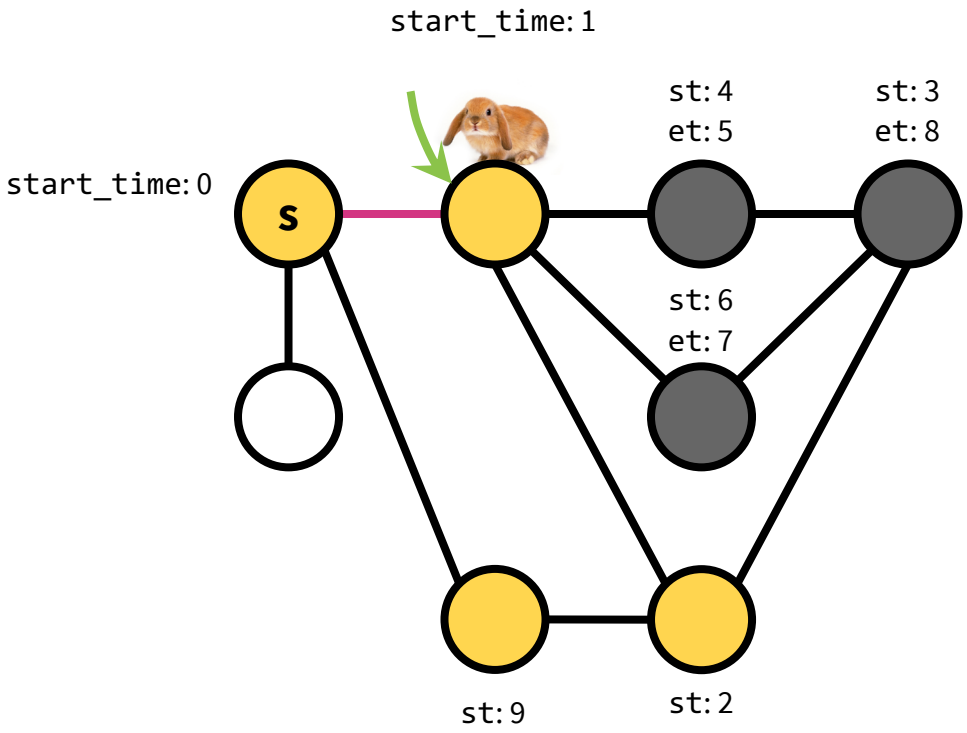
Depth-First Search



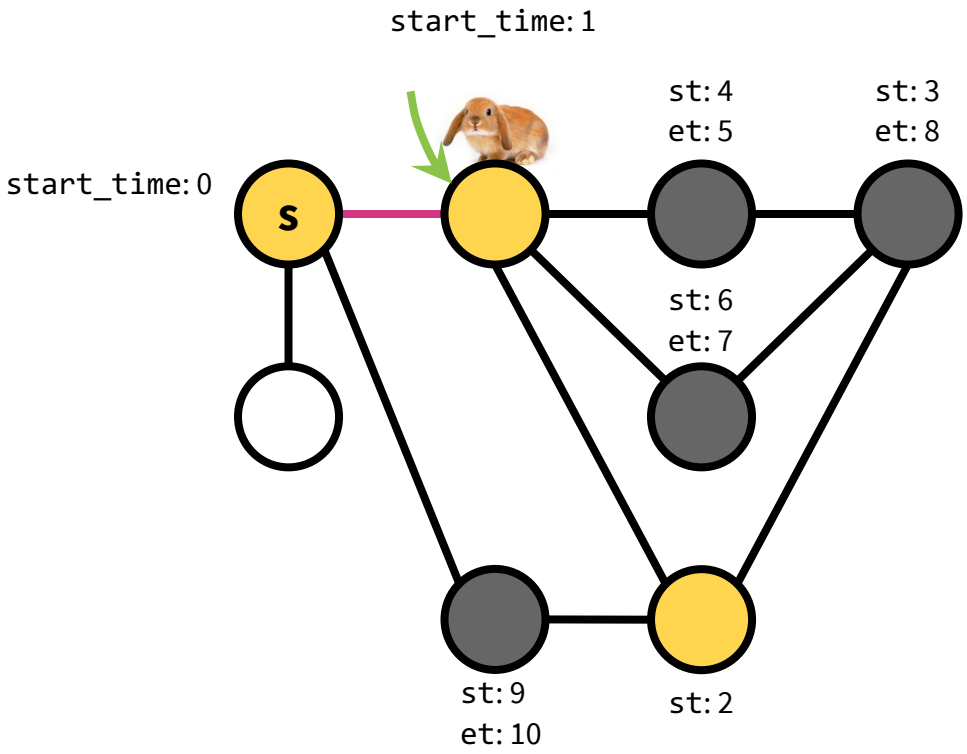
Depth-First Search



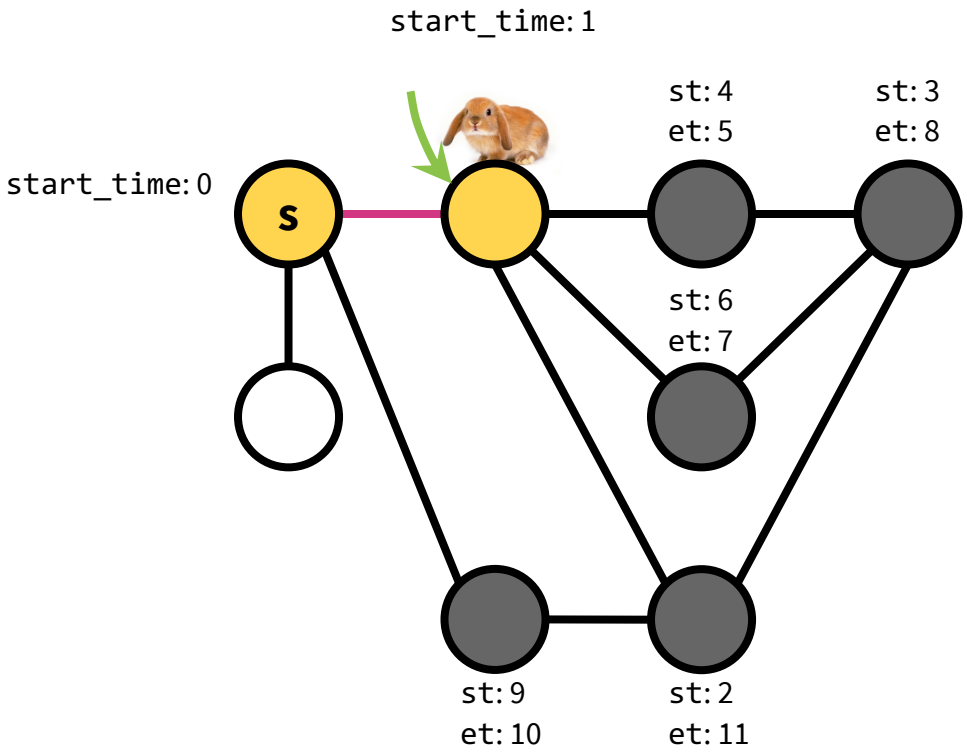
Depth-First Search



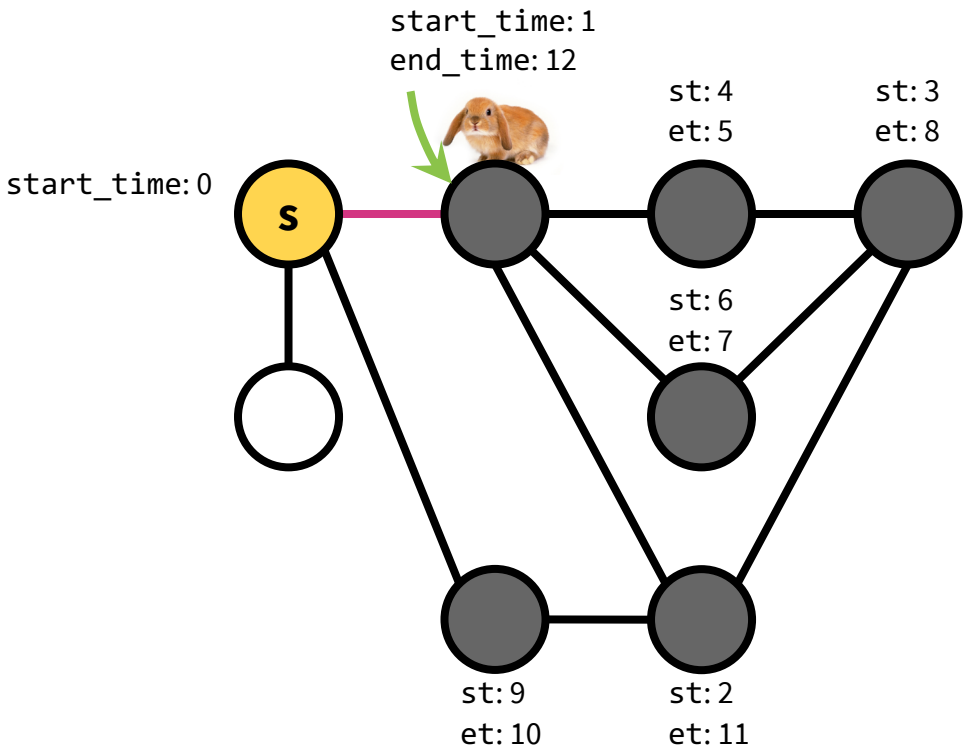
Depth-First Search



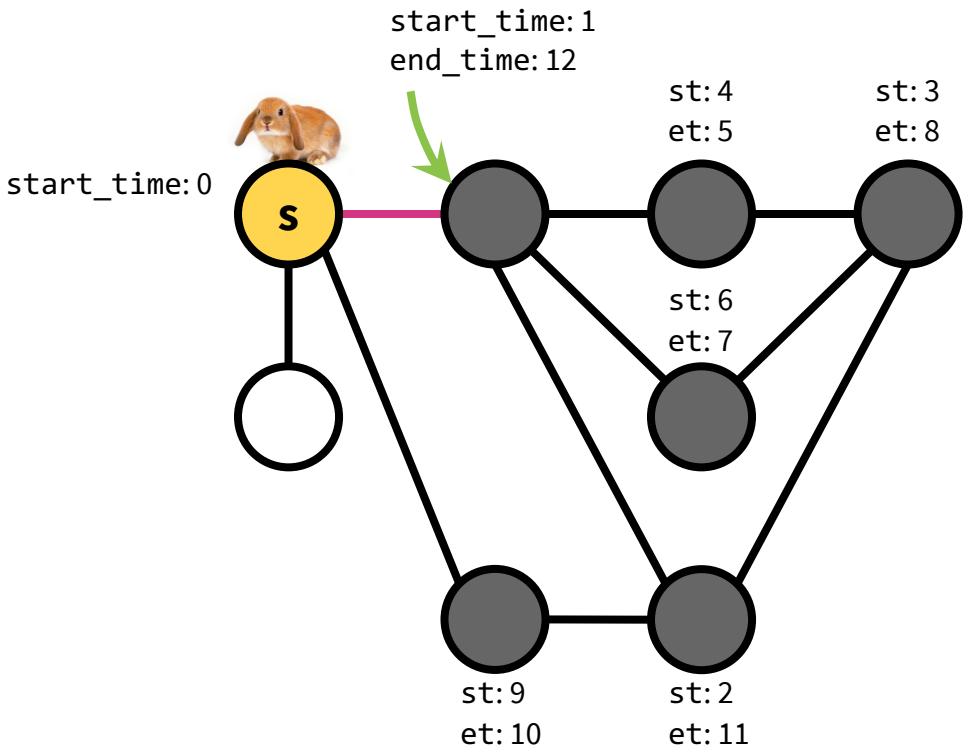
Depth-First Search



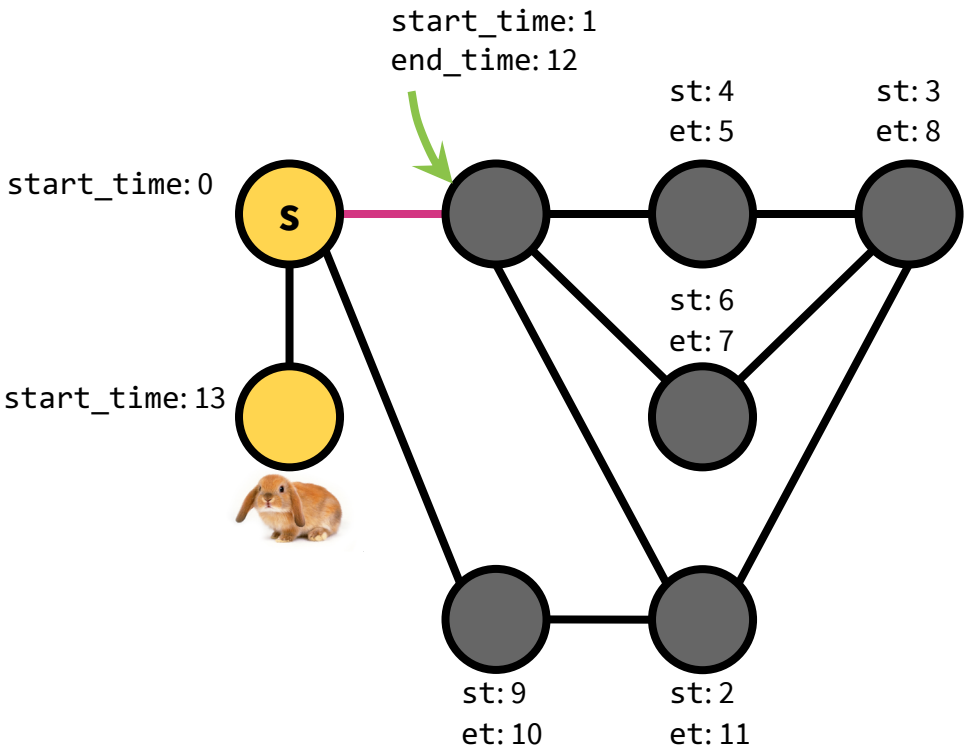
Depth-First Search



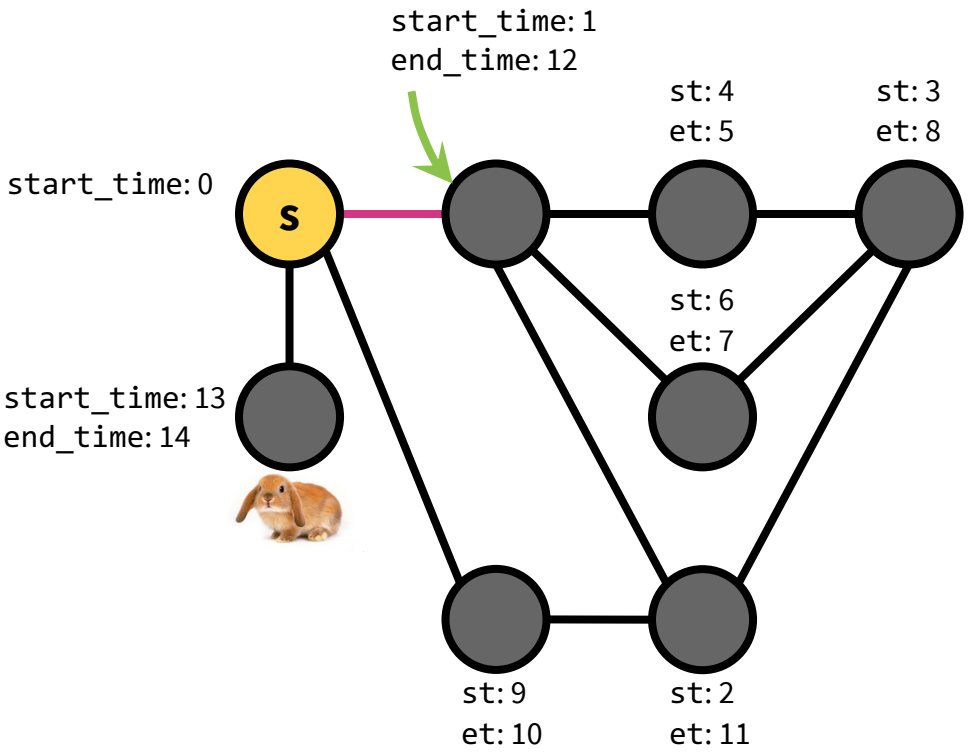
Depth-First Search



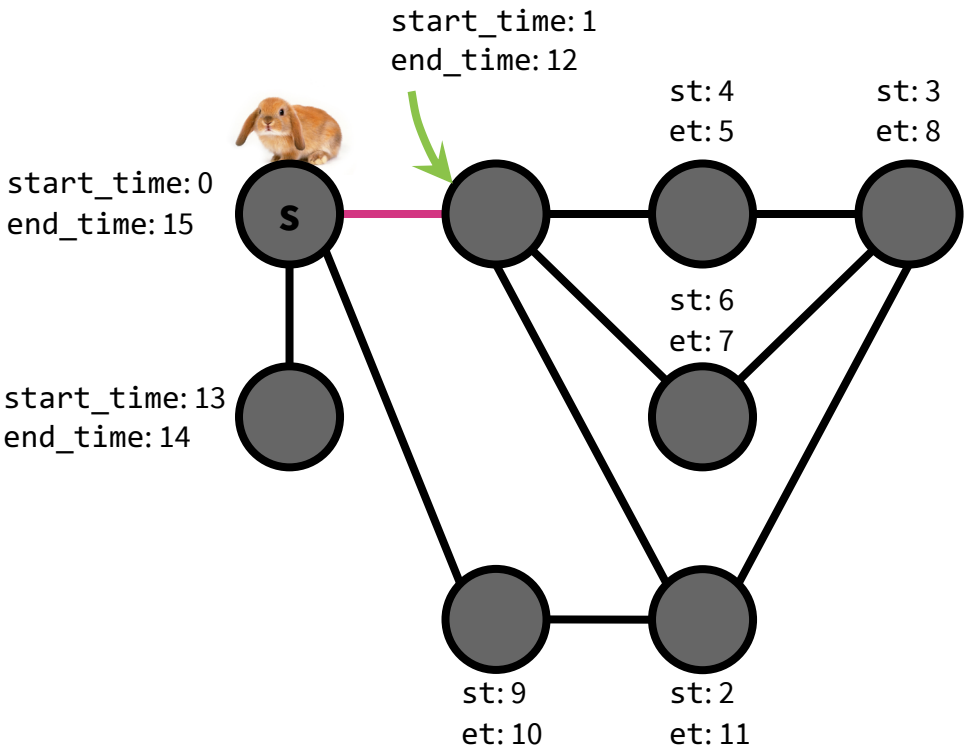
Depth-First Search



Depth-First Search



Depth-First Search



Depth-First Search

Another implementation by using a stack:

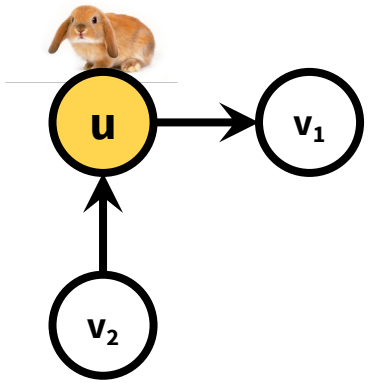
```
algorithm dfs-iterative(G, u):
  let S be a stack
  S.push(u)
  while S is not empty
    u = S.pop()
    if u is not labeled as visited:
       label u as visited
       for all neighbors v:
          S.push(v)
```

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.

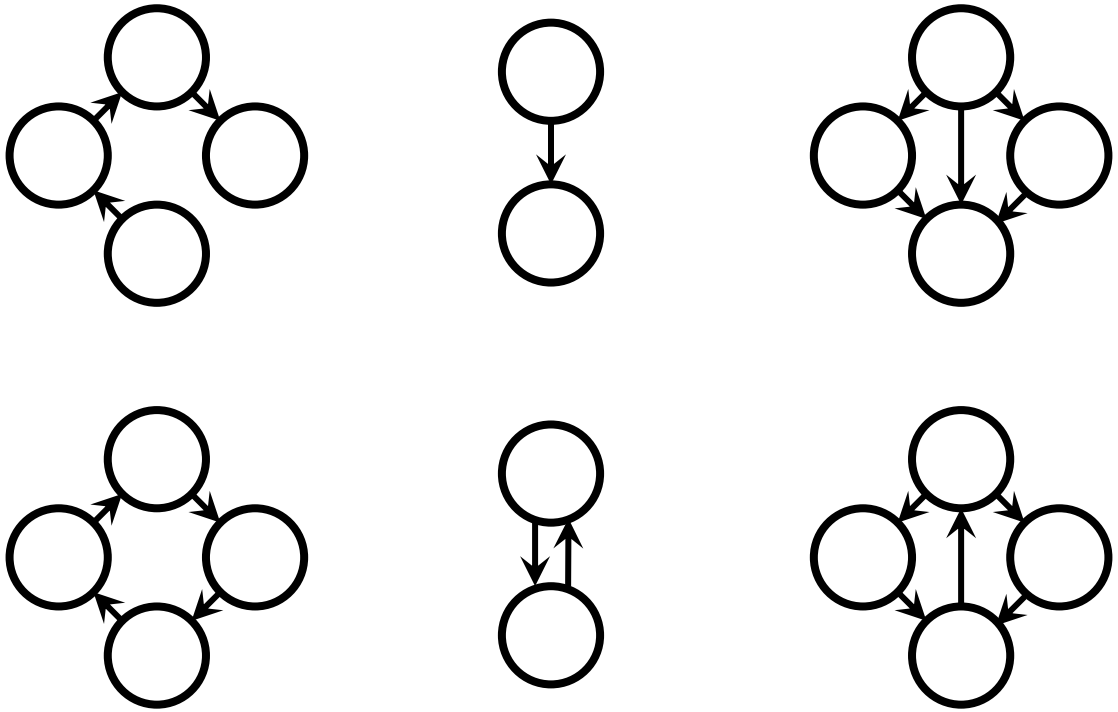


Topological Ordering

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.

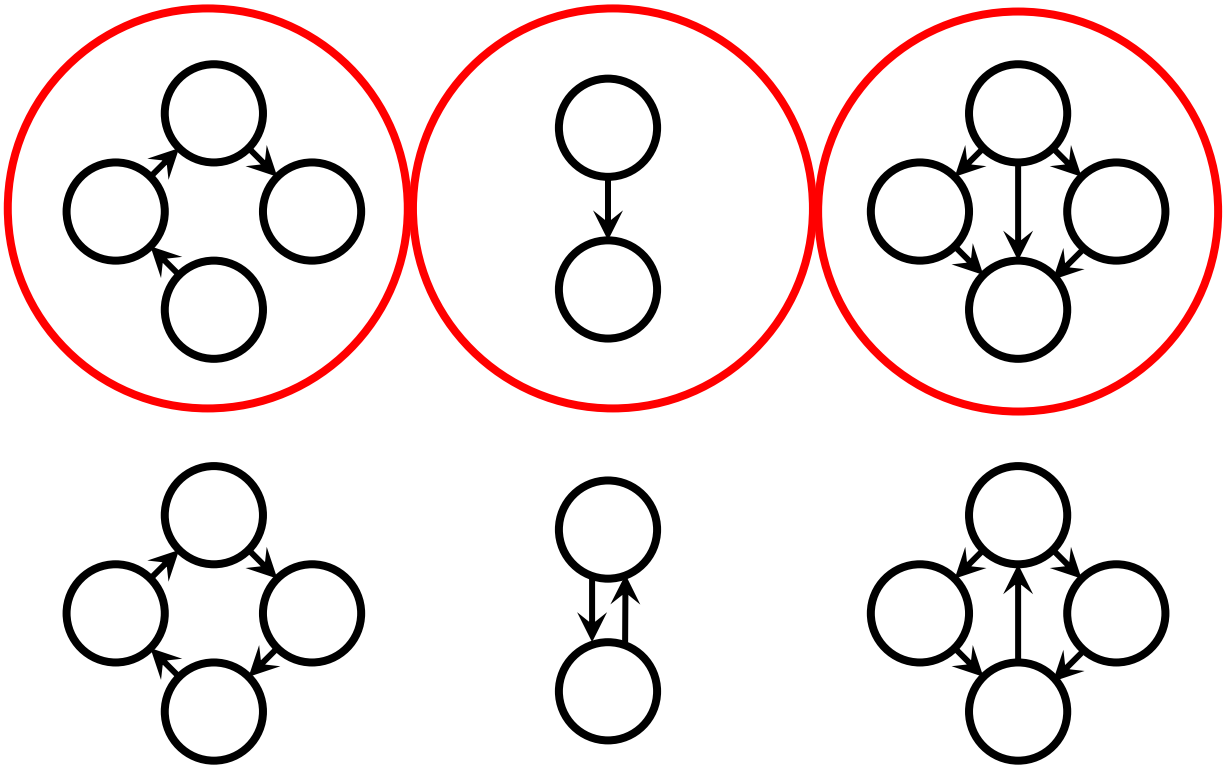
Which of these graphs are valid DAGs? 🤔



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.

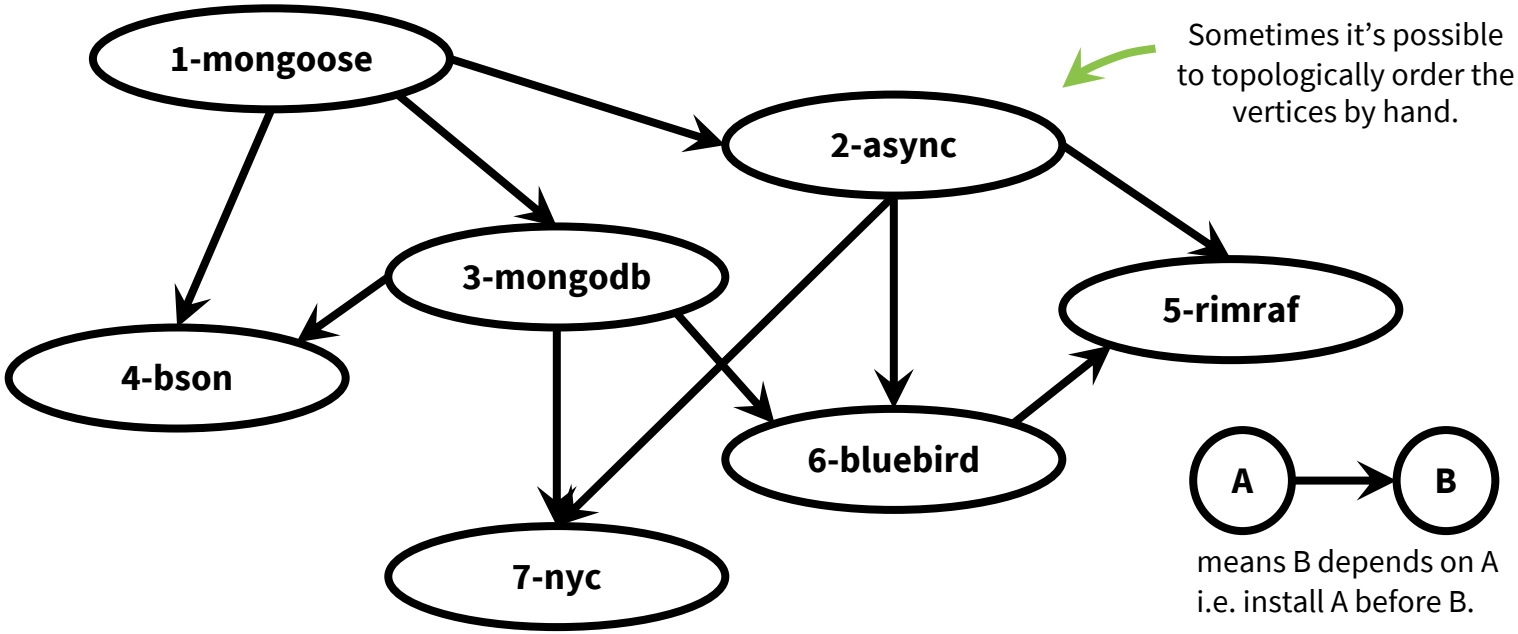
Definition: The topological ordering of a DAG is an ordering of its vertices such that for every directed edge (u, v) ∈ E, u precedes v in the ordering.

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.

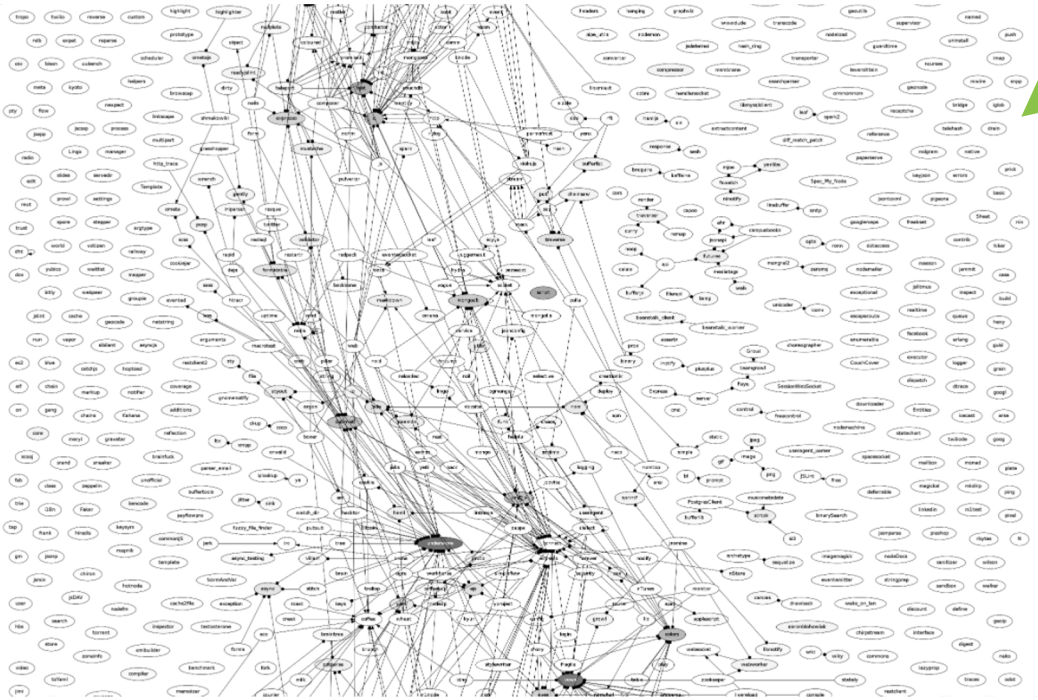


Topological Ordering

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 not …

Topological Ordering

```
algorithm 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

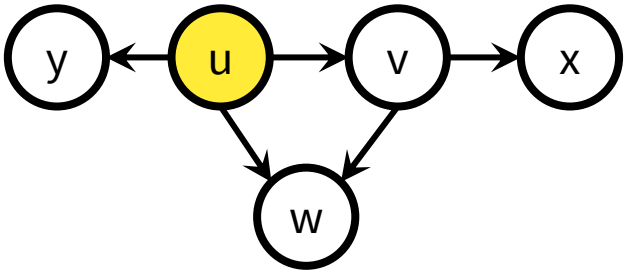
```
reversed_topological_list = []
algorithm dfs(u, cur_time):
  u.start_time = cur_time
  cur_time += 1
  u.status = "in_progress"
  for v in u.neighbors:
    if v.status is "unvisited":
       cur_time = dfs(v, cur_time)
       cur_time += 1
  u.end_time = cur_time
  u.status = "done"
  reversed_topological_list.append(u)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

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

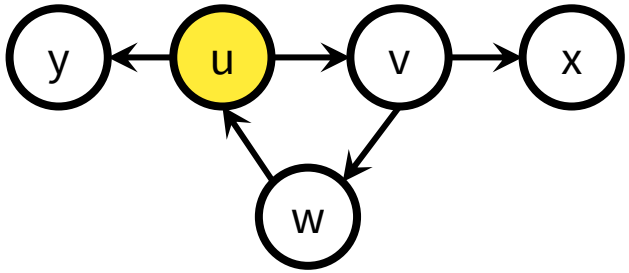
Topological Ordering

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

Intuition: `dfs` visits and finishes 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).



Finish v w y then finish u



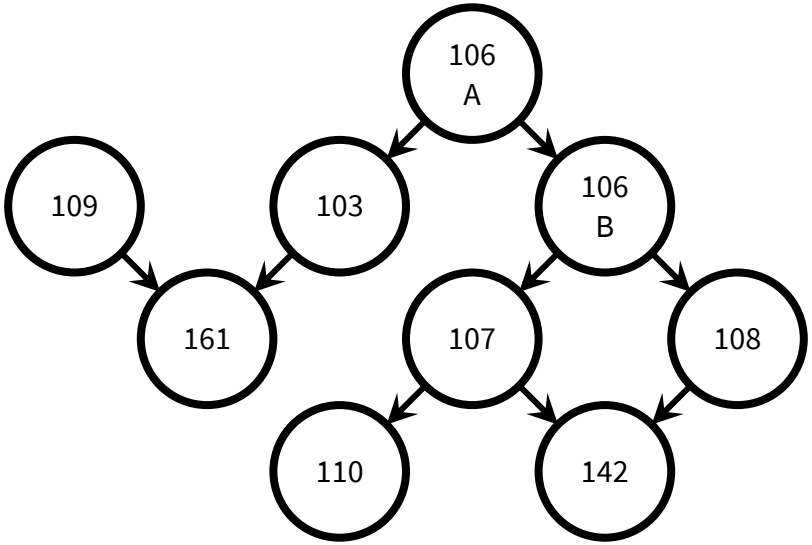
Finishing u requires finishing v thus w;
But finishing w requires finishing u (an in-progress node);
Which is a deadlock

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`.

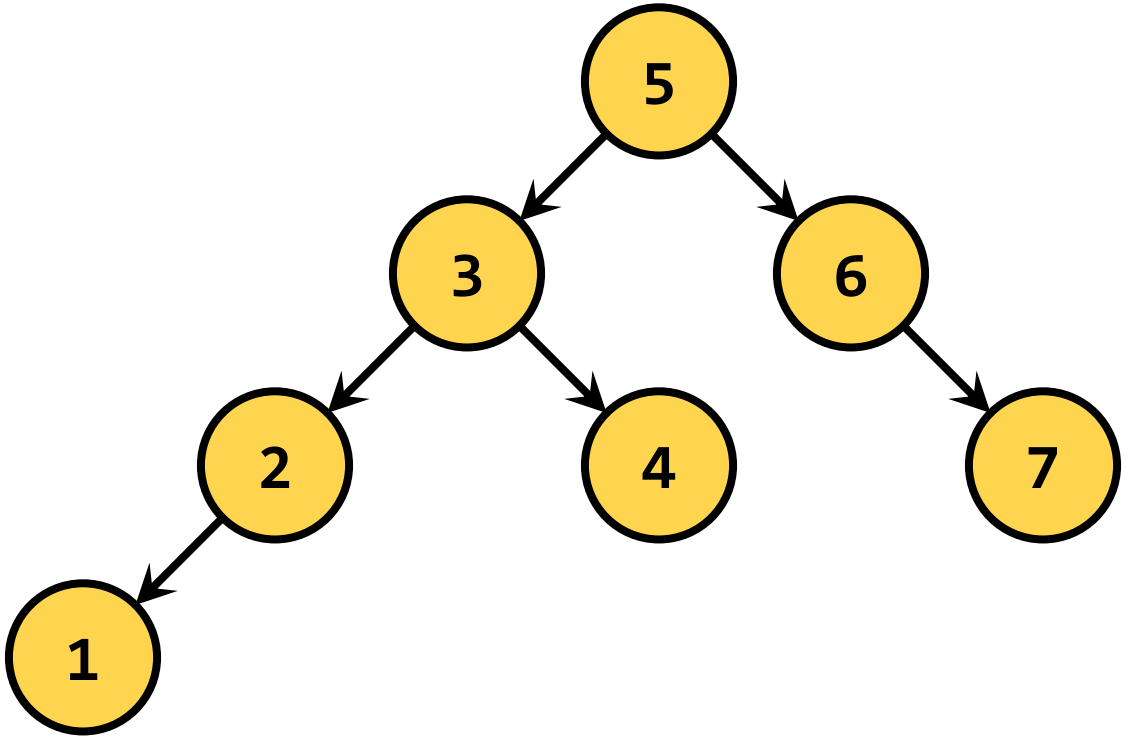
e.g. Finding an order to take courses that satisfies prerequisites.



Reversed_topological_list = 110, 142, 107, 108, 106B, 161, 103, 106A, 109

In-Order Traversal of BSTs

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



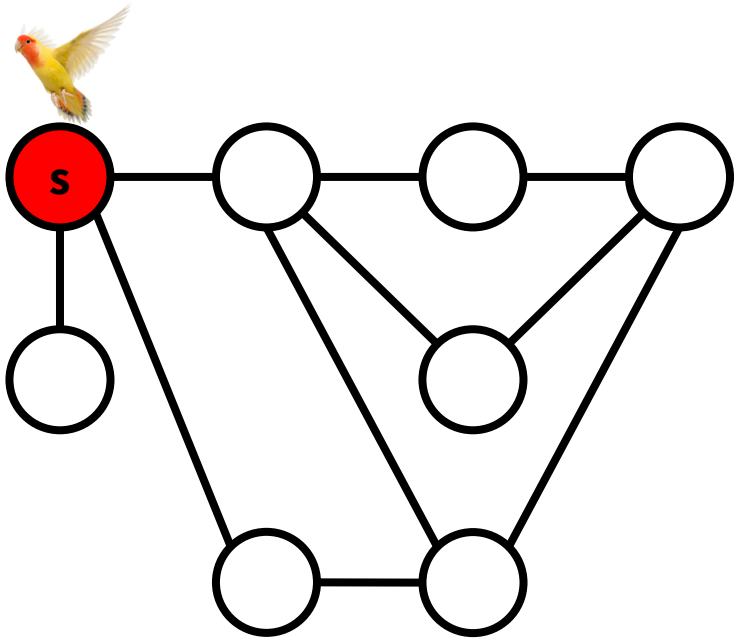
In-order traversal: visit left subtree -> visit the node -> visit the right tree

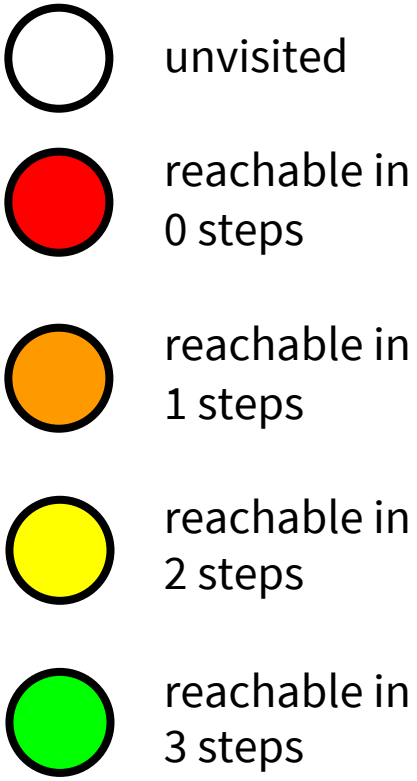
Breadth-First Search

Breadth-First Search

An analogy

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

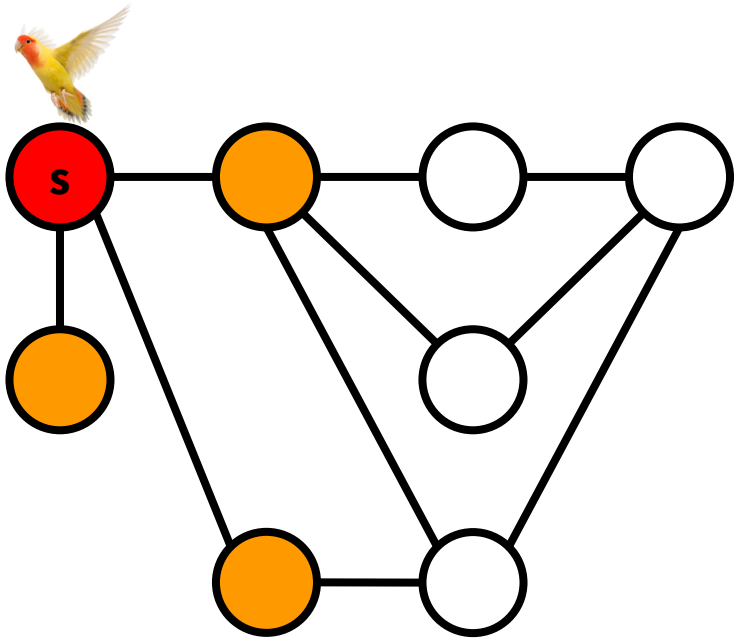


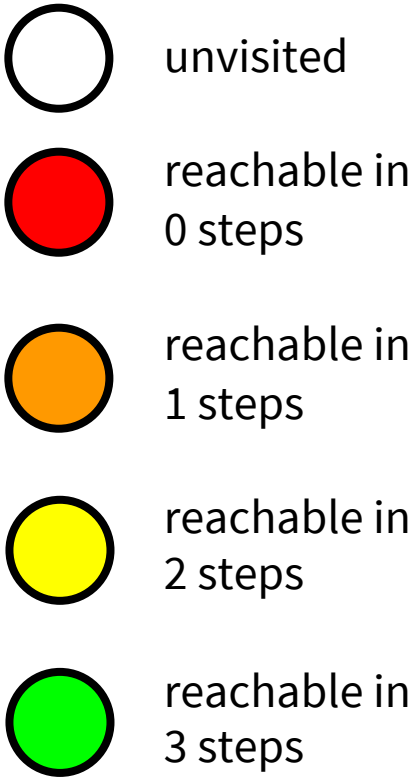


Breadth-First Search

An analogy

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

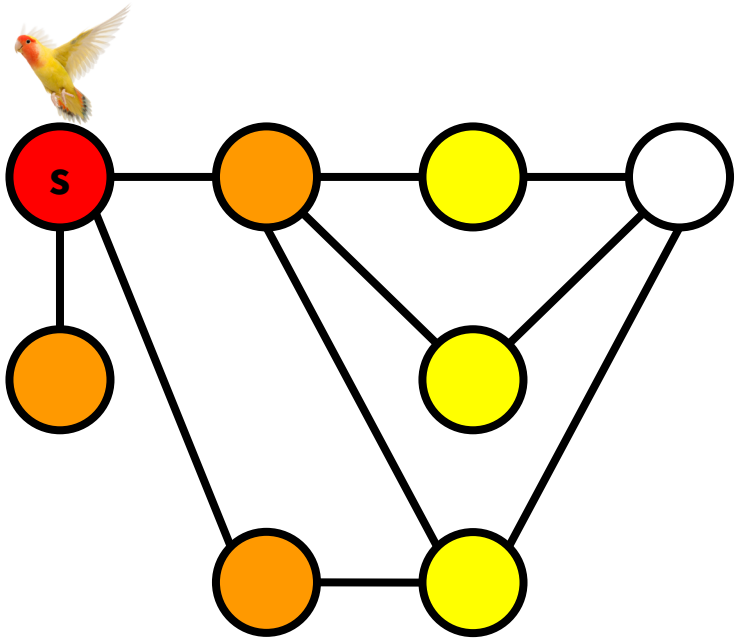


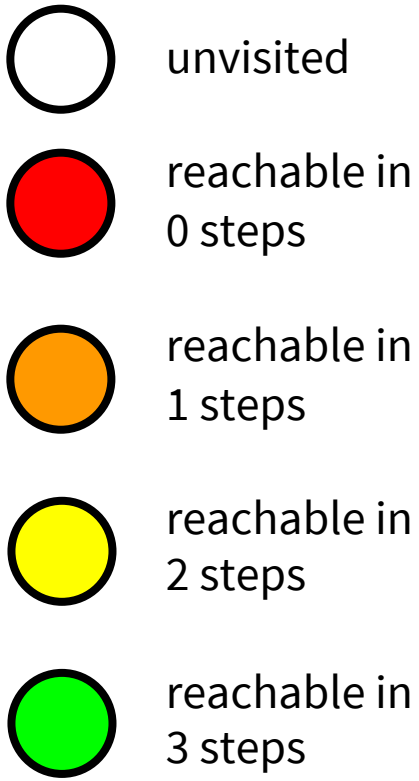


Breadth-First Search

An analogy

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

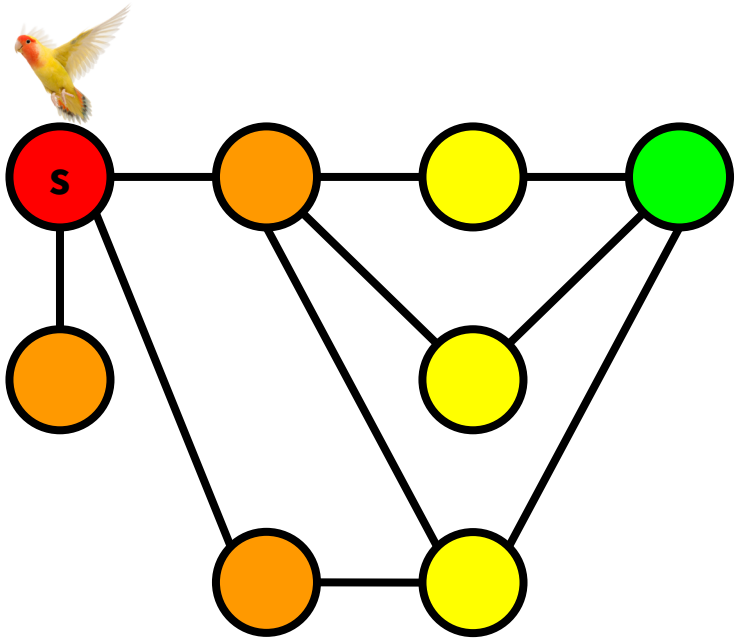


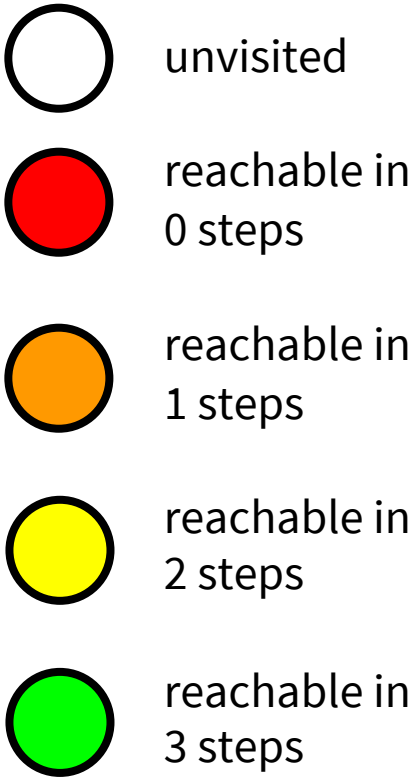


Breadth-First Search

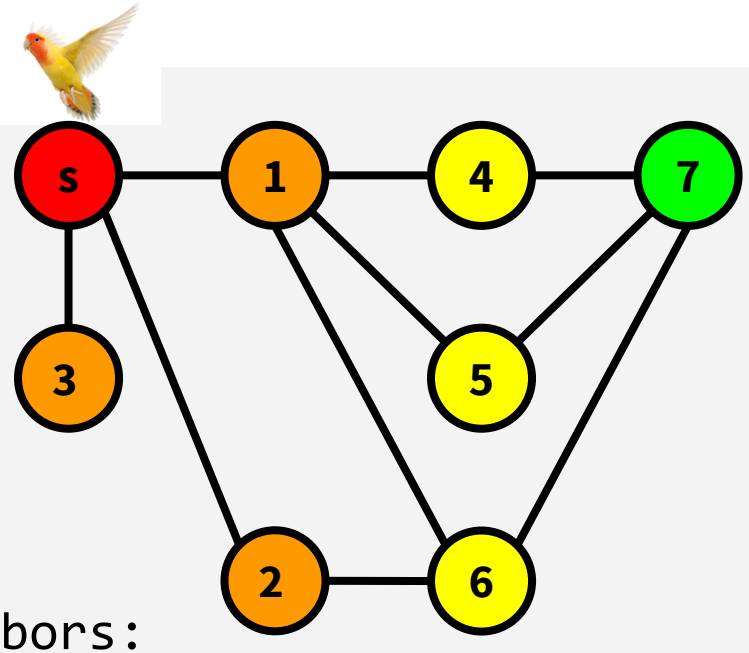
An analogy

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





Breadth-First Search

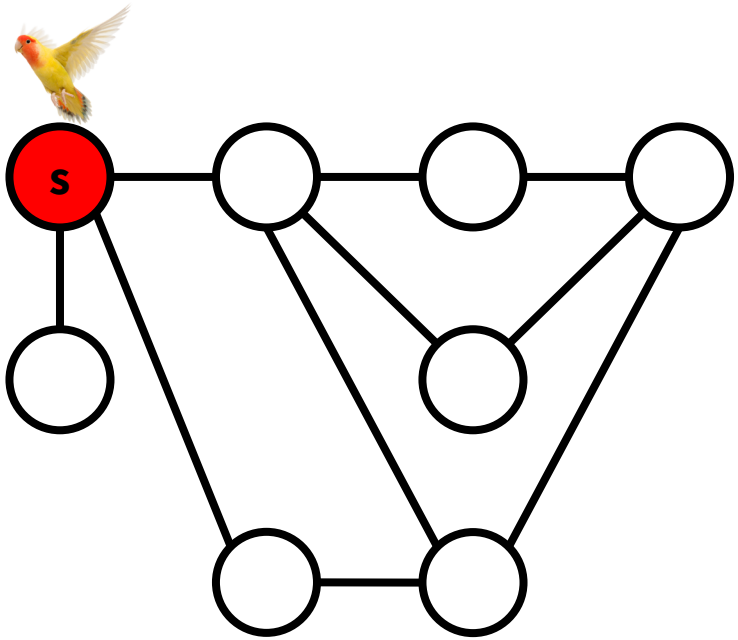


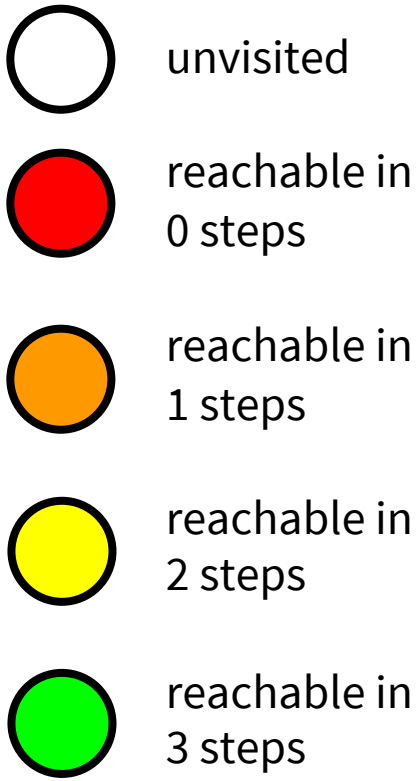

```
algorithm bfs(s):
  L = []
  for i = 0 to n-1:
    L[i] = {}
  L[0] = {s}
  for i = 0 to n-1:
    for u in L[i]:
      for v in u.neighbors:
        if v.status is "unvisited":
          v.status = "visited"
          L[i+1].add(v)
```

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

Breadth-First Search

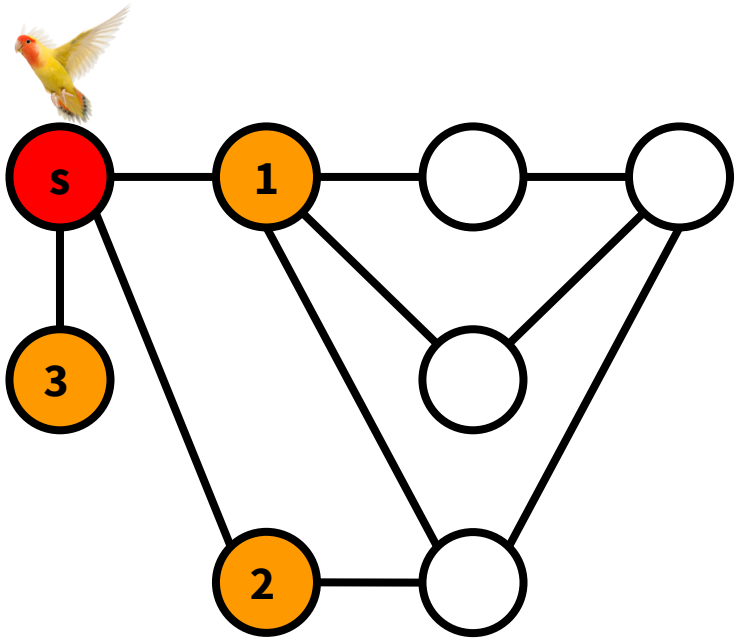
L[0] = {s} // Initialize

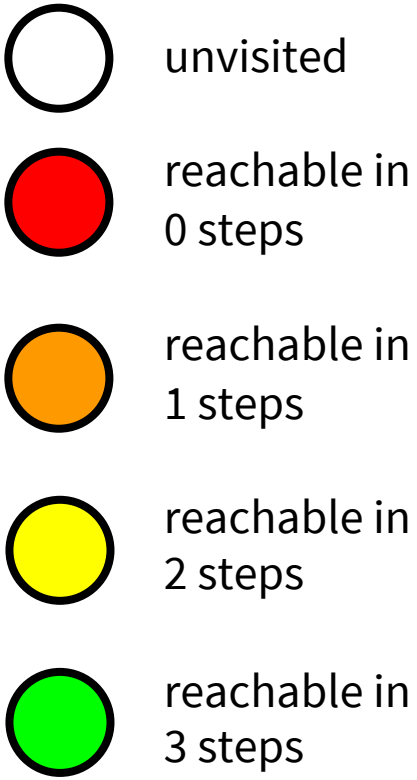




Breadth-First Search

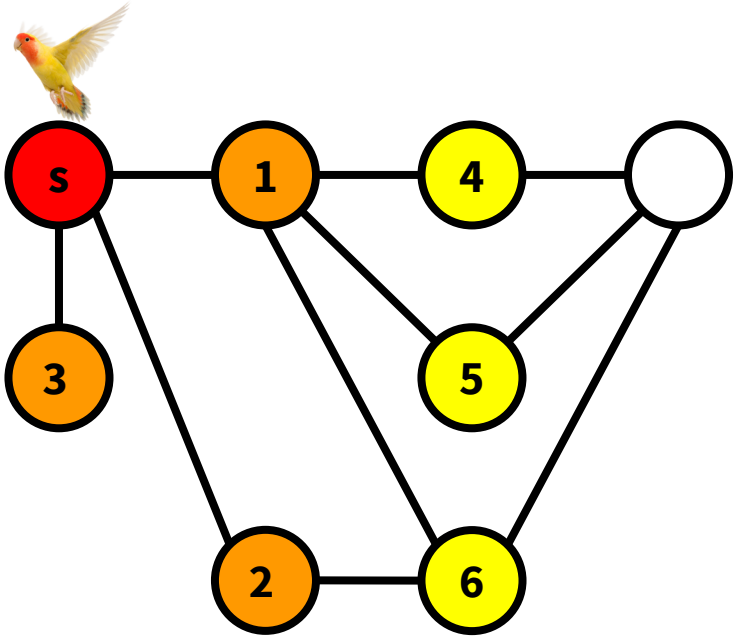
L[0] = {s} // Initialize
L[1] = {1, 2, 3} // Take out s from L[0], visit its (unvisited) neighbors and put them in L[1]

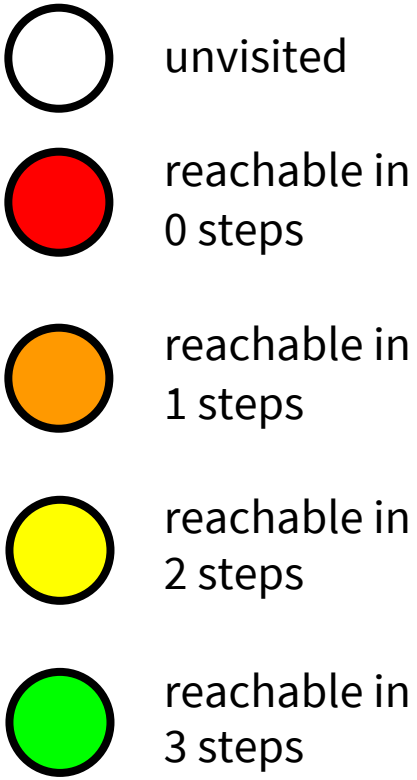




Breadth-First Search

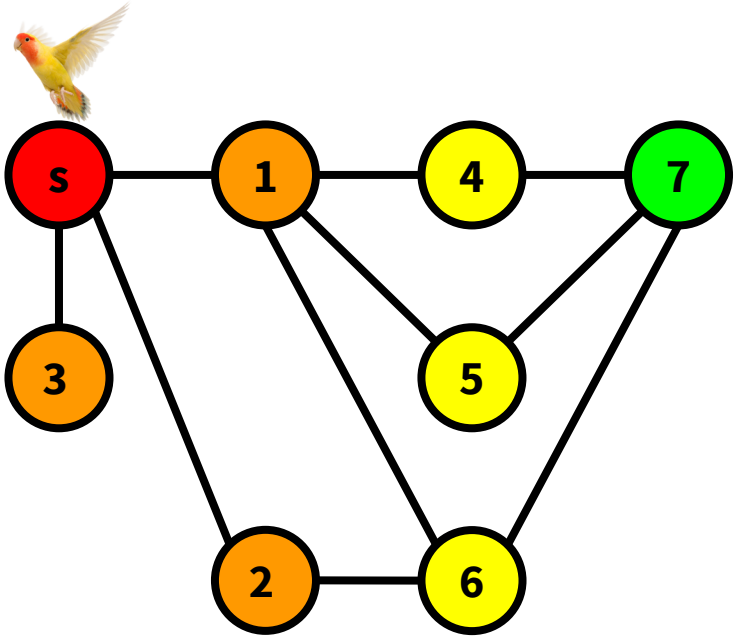
L[0] = {s} // Initialize
L[1] = {1, 2, 3} // Take out s from L[0], visit its (unvisited) neighbors and put them in L[1]
L[2] = {4, 5, 6} // Take out 1, 2, 3 from L[1], visit their (unvisited) neighbors and put them in L[2]

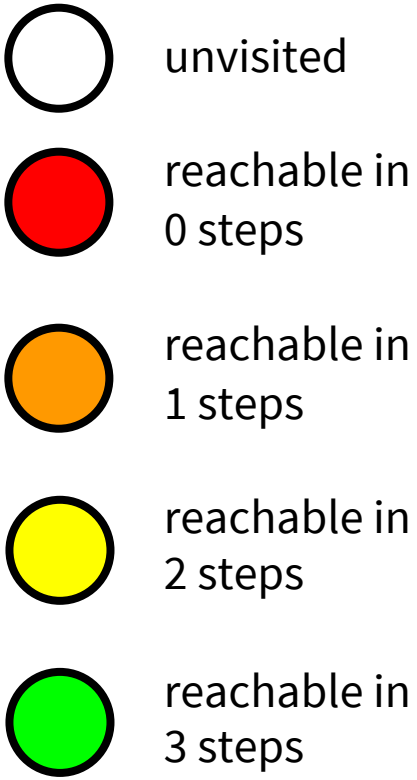




Breadth-First Search

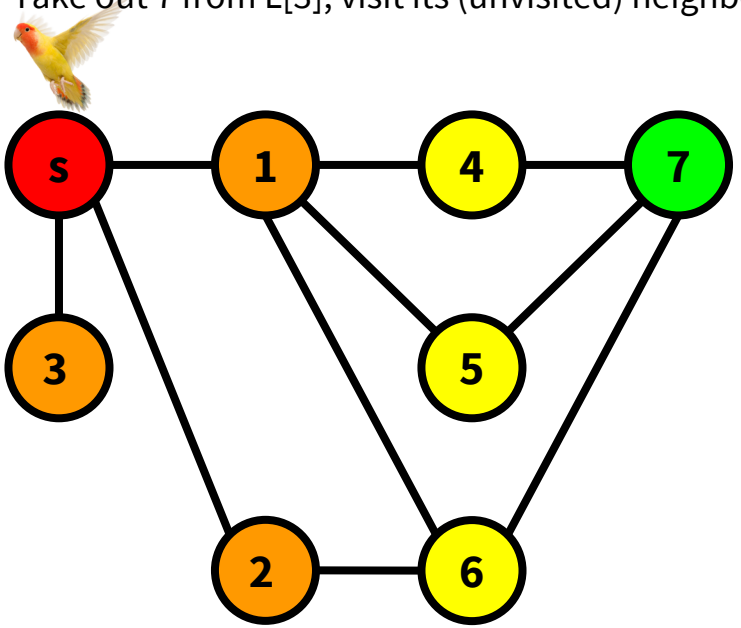
L[0] = {s} // Initialize
L[1] = {1, 2, 3} // Take out s from L[0], visit its (unvisited) neighbors and put them in L[1]
L[2] = {4, 5, 6} // Take out 1, 2, 3 from L[1], visit their (unvisited) neighbors and put them in L[2]
L[3] = {7} // Take out 4, 5, 6 from L[2], visit their (unvisited) neighbors and put them in L[3]

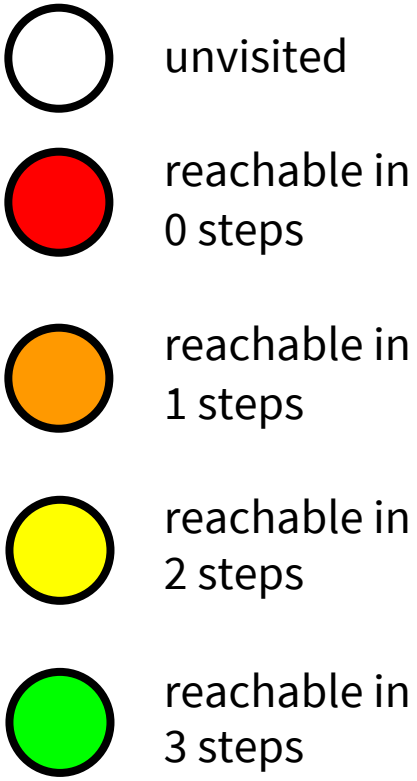




Breadth-First Search

L[0] = {s} // Initialize
L[1] = {1, 2, 3} // Take out s from L[0], visit its (unvisited) neighbors and put them in L[1]
L[2] = {4, 5, 6} // Take out 1, 2, 3 from L[1], visit their (unvisited) neighbors and put them in L[2]
L[3] = {7} // Take out 4, 5, 6 from L[2], visit their (unvisited) neighbors and put them in L[3]
L[4] = {} // Take out 7 from L[3], visit its (unvisited) neighbors, but there is no unvisited neighbor anymore, stop.





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.

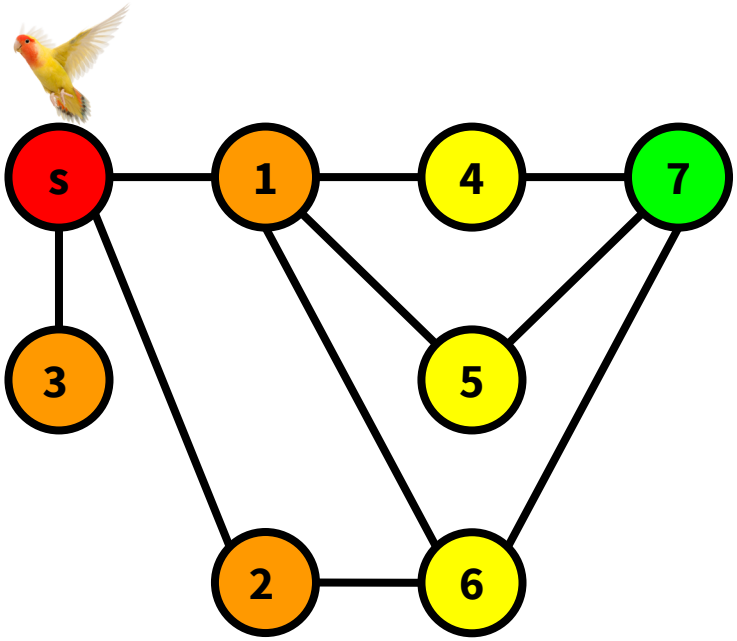
If v isn't in `L[i]` for any i, then it's unreachable from u.

Shortest Path

For example, by calling bfs(s) on node s, we have the following lists:

L[0] = {s} // Initialize
L[1] = {1, 2, 3} // Take out s from L[0], visit its (unvisited) neighbors and put them in L[1]
L[2] = {4, 5, 6} // Take out 1, 2, 3 from L[1], visit their (unvisited) neighbors and put them in L[2]
L[3] = {7} // Take out 4, 5, 6 from L[2], visit their (unvisited) neighbors and put them in L[3]
L[4] = {} // Take out 7 from L[3], visit its (unvisited) neighbors, but there is no unvisited neighbor anymore, stop.

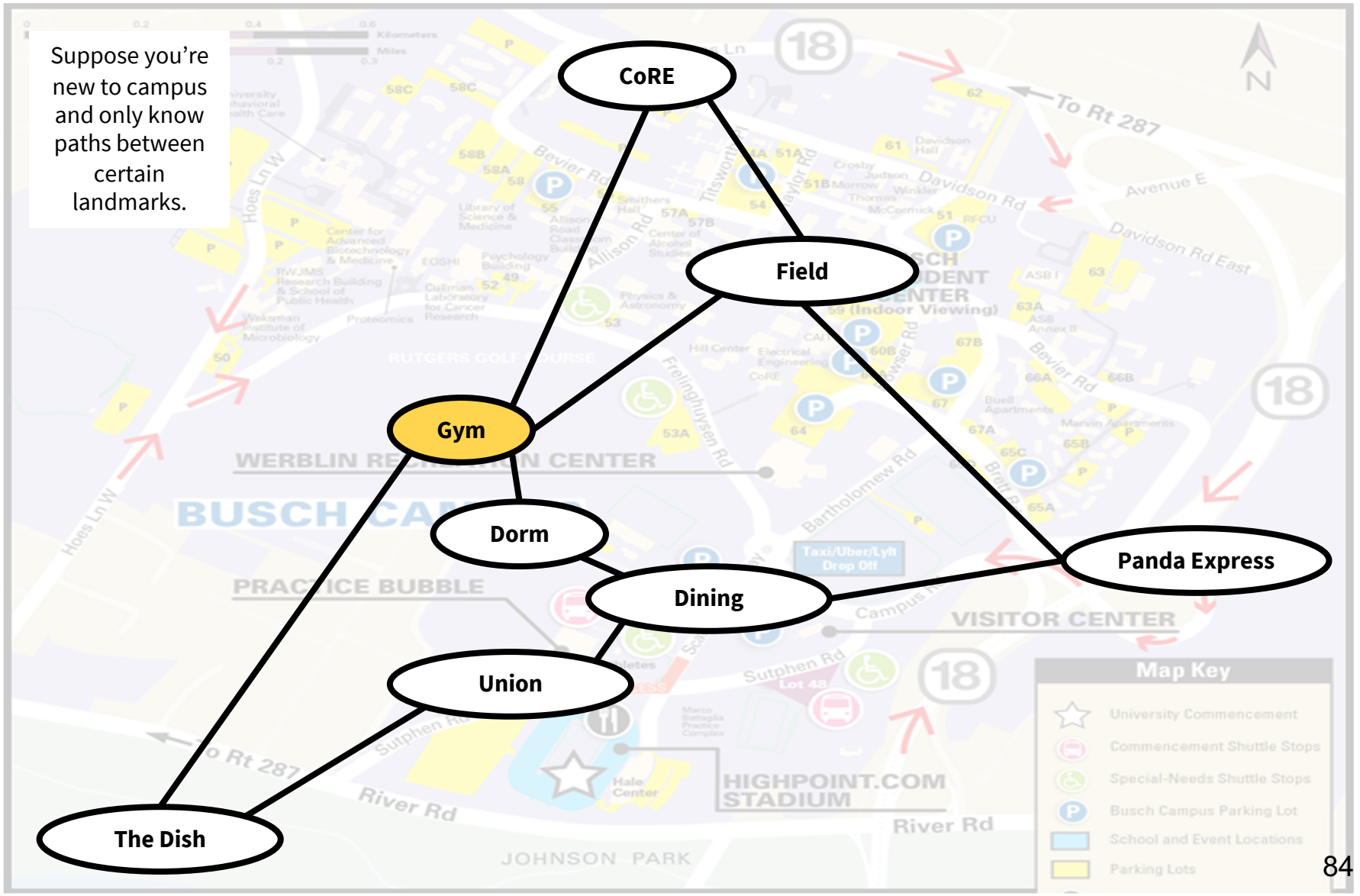
We know the shortest path between s and node 5 has length 2, because node 5 appears in L[2].



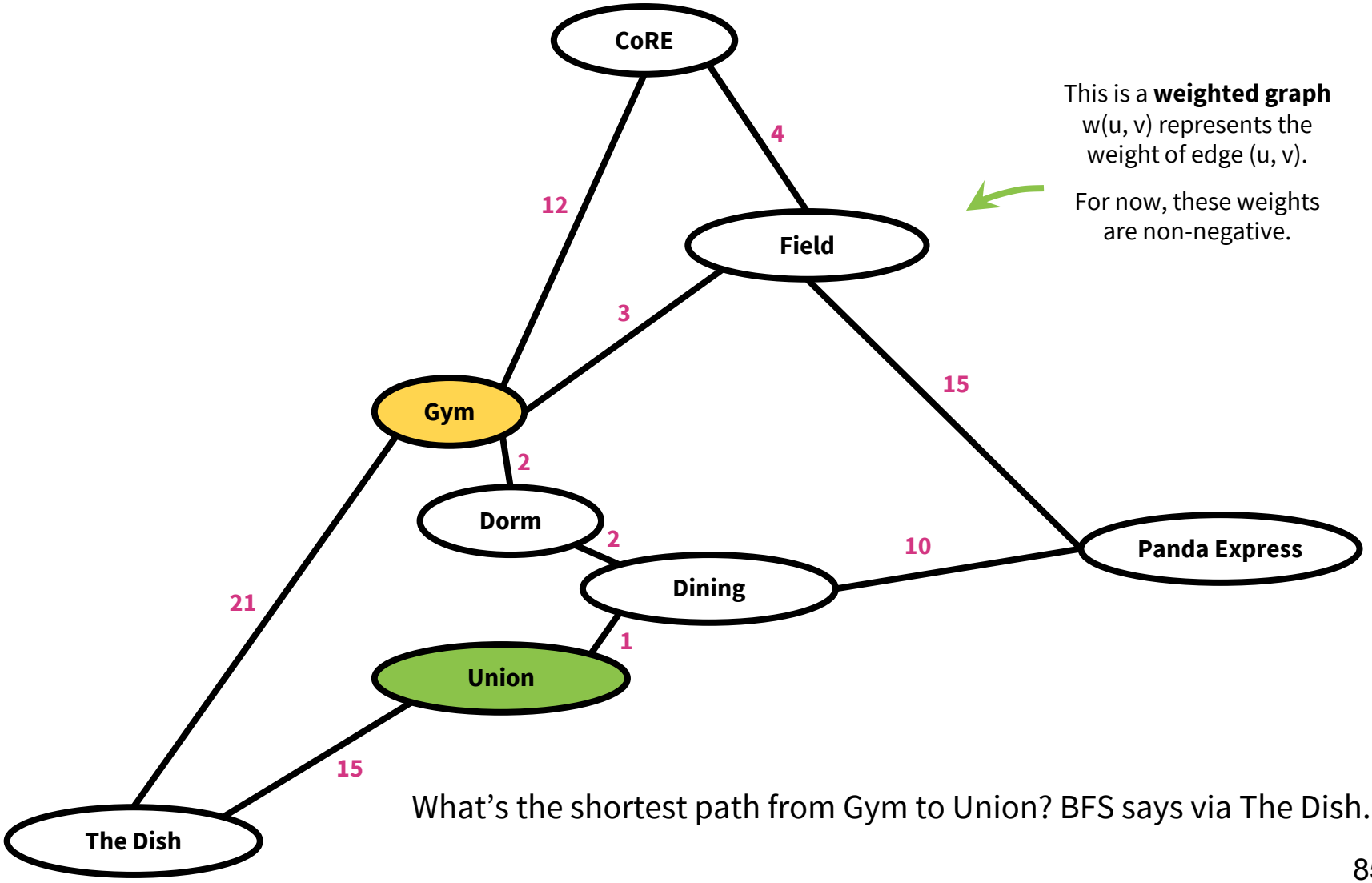
Dijkstra’s Algorithm

Shortest Path

Suppose you're new to campus and only know paths between certain landmarks.



Shortest Path



This is a **weighted graph**
w(u, v) represents the weight of edge (u, v).

For now, these weights are non-negative.

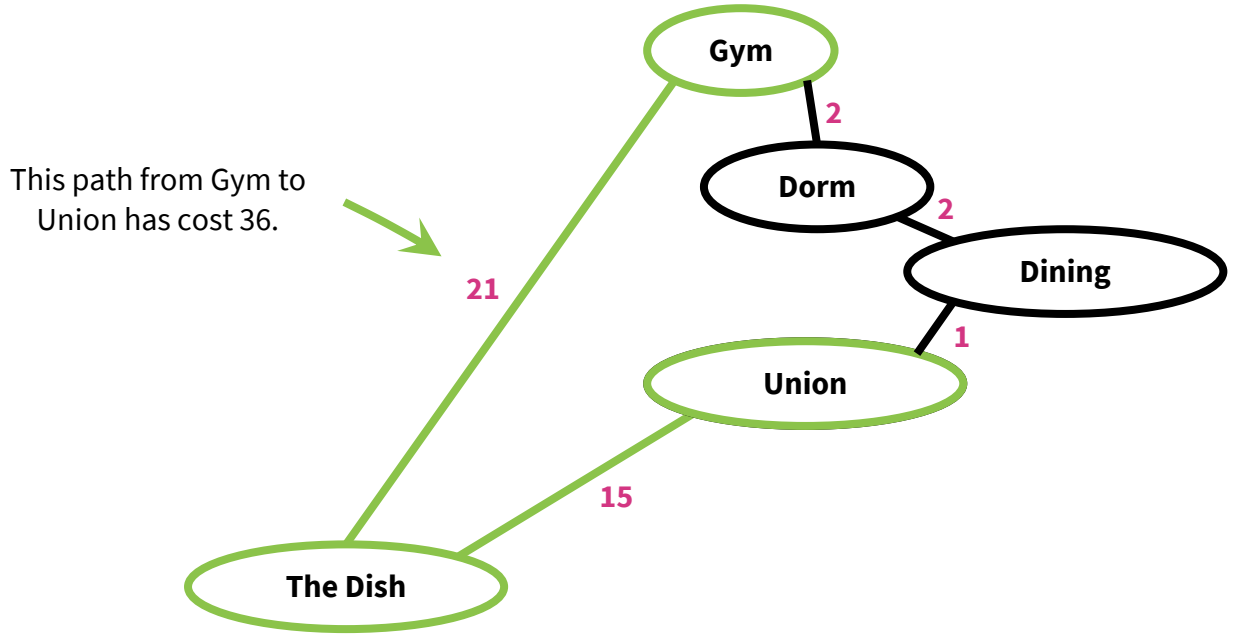
What's the shortest path from Gym to Union? BFS says via The Dish.

Shortest Path

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.

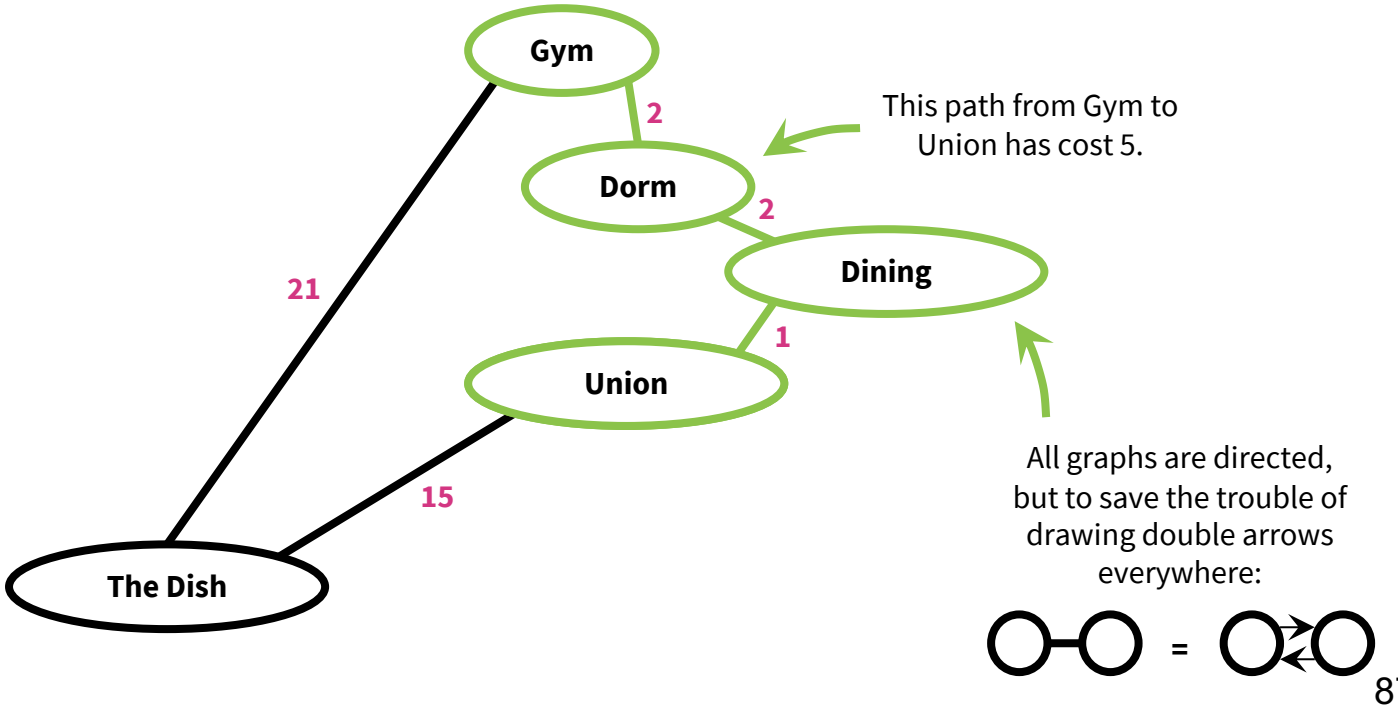


Shortest Path

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.

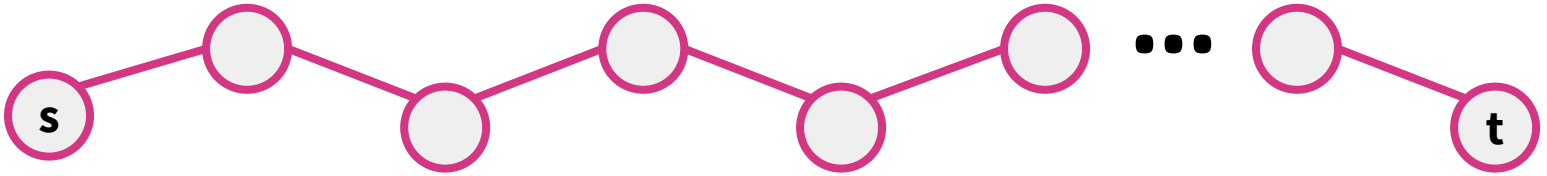
The shortest path is the one with the minimum cost.



Shortest Path

Claim: A subpath of a shortest path is also a shortest path.

Intuition:

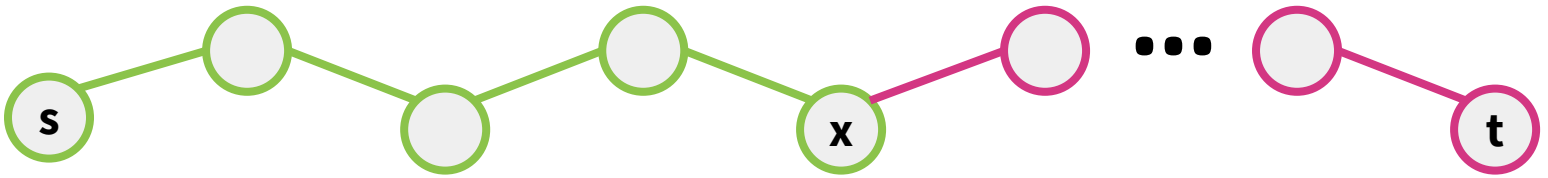


Suppose **this** is a shortest path from **s** to **t**.

Shortest Path

Claim: A subpath of a shortest path is also a shortest path.

Intuition:



Suppose **this** is a shortest path from **s** to **t**.

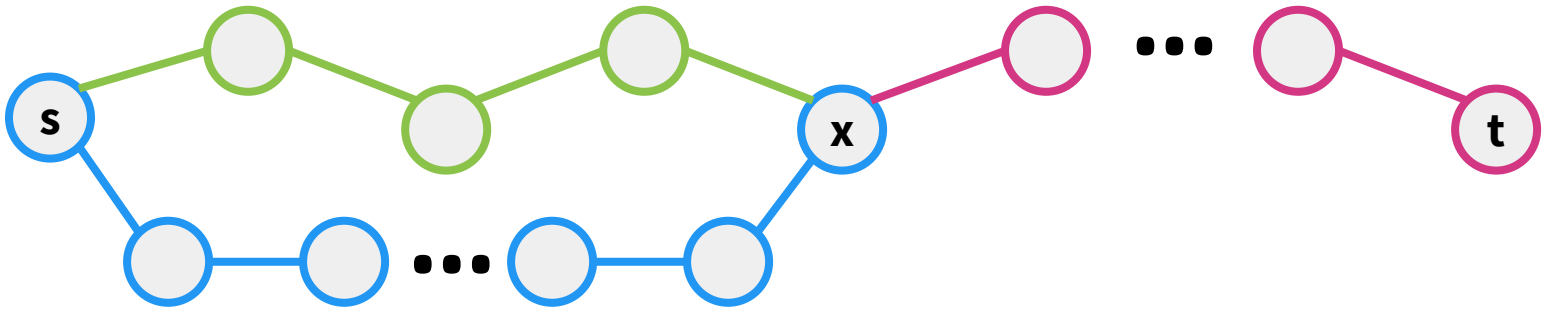
Then **this** is a shortest path from **s** to **x**.

Why? 🤔

Shortest Path

Claim: A subpath of a shortest path is also a shortest path.

Intuition:



Suppose **this** is a shortest path from **s** to **t**.

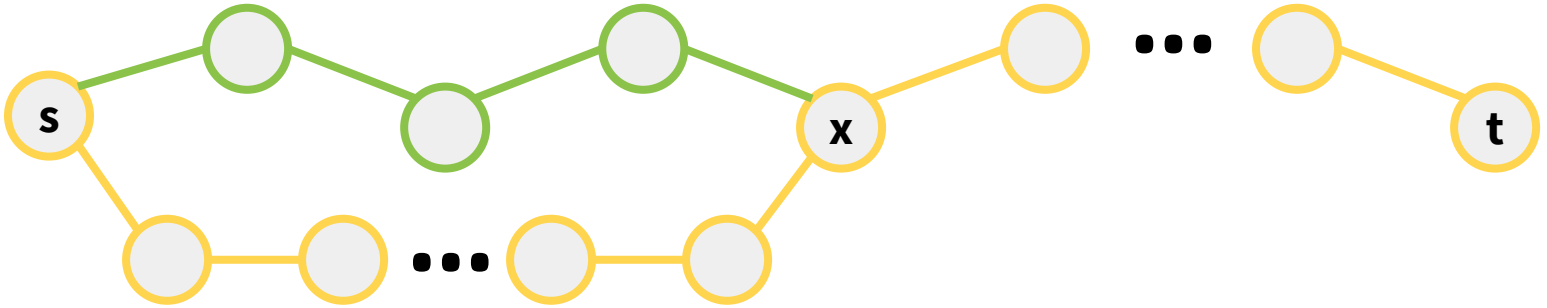
Then **this** is a shortest path from **s** to **x**.

Why? 🤔 By contradiction, suppose there exists a shorter path from **s** to **x**, namely **this** one.

Shortest Path

Claim: A subpath of a shortest path is also a shortest path.

Intuition:



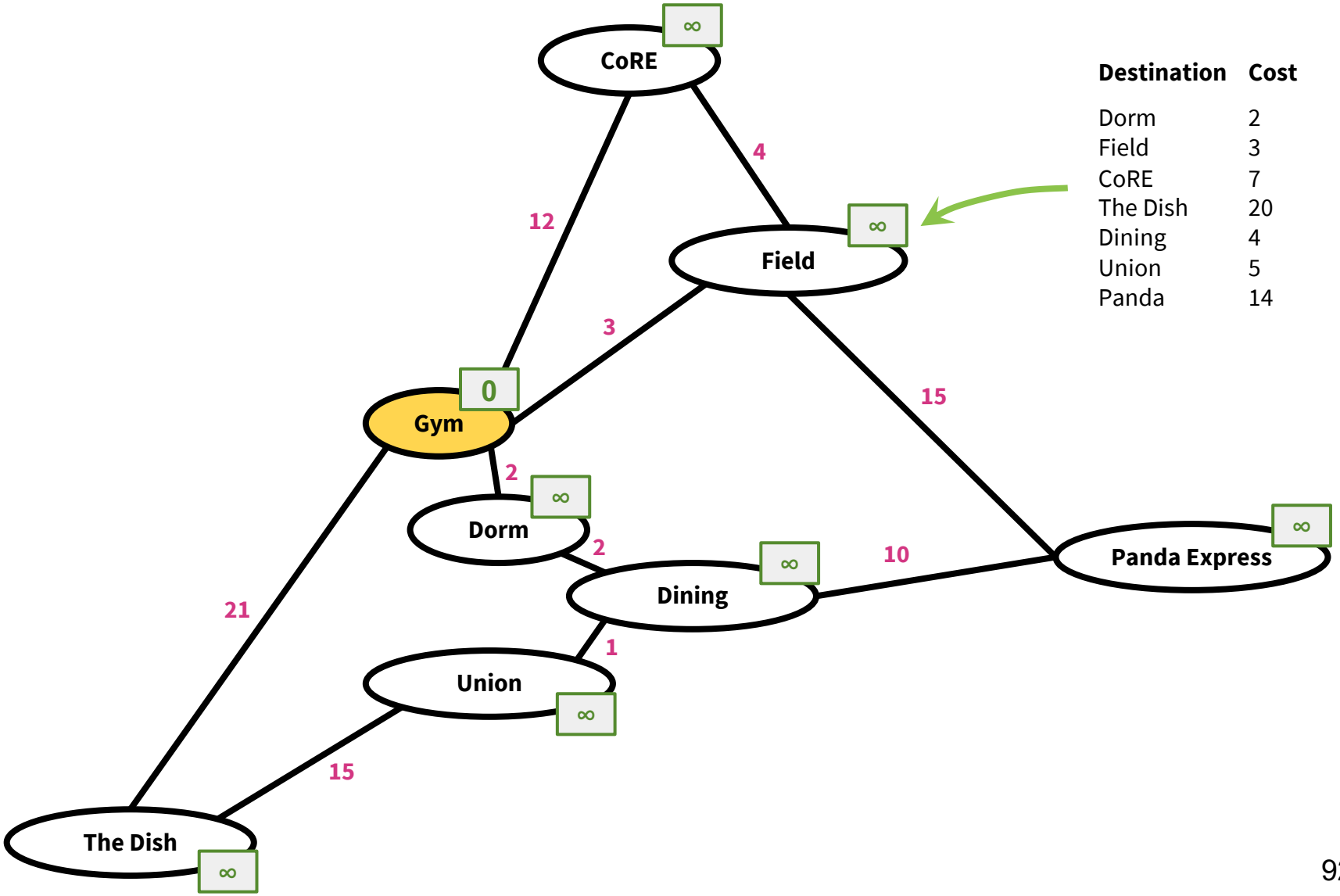
Suppose **this** is a shortest path from **s** to **t**.

Then **this** is a shortest path from **s** to **x**.

Why? 🤔 By contradiction, suppose there exists a shorter path from **s** to **x**, namely **this** one.

But then **this** is shorter than **this** shortest path from **s** to **t**.

Single-Source Shortest Path



Single-Source Shortest Path

Application: Finding the shortest path from CoRE building to [somewhere else] for a commuter using Bus, bike, walking, Uber, Lyft, etc.

Edge weights are a function of distance, time, money, that change depending on the commuter's mood on that day.

Application: Routing messages through the internet. Finding the shortest path from my computer to the desired server for packets using the Internet.

Edge weights are a function of link length, traffic, reliability, package loss probability, or other costs, etc.

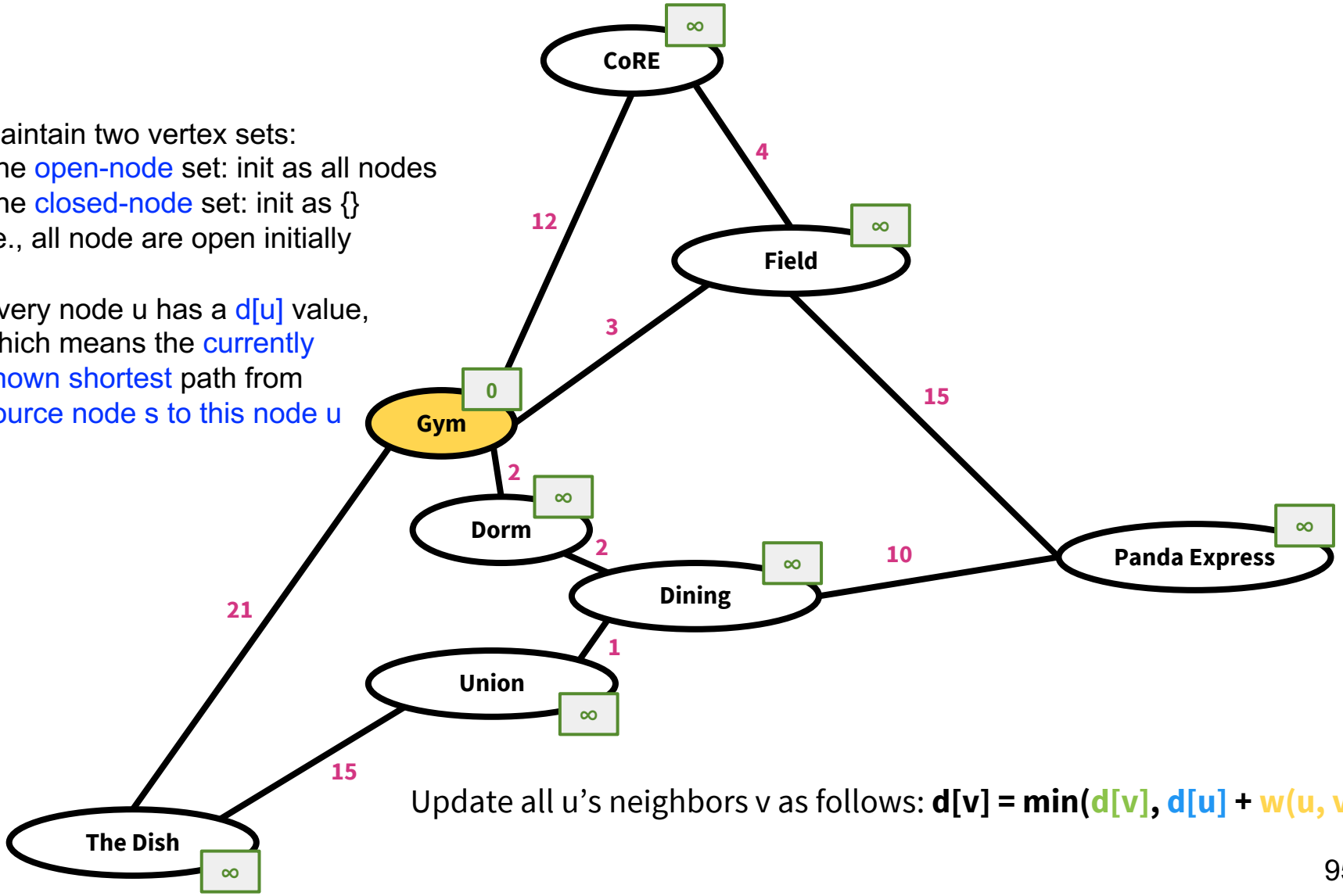
Dijkstra’s Algorithm

Dijkstra’s Algorithm solves the single-source shortest path problem.

Dijkstra's Algorithm

Maintain two vertex sets:
The open-node set: init as all nodes
The closed-node set: init as {}
i.e., all node are open initially

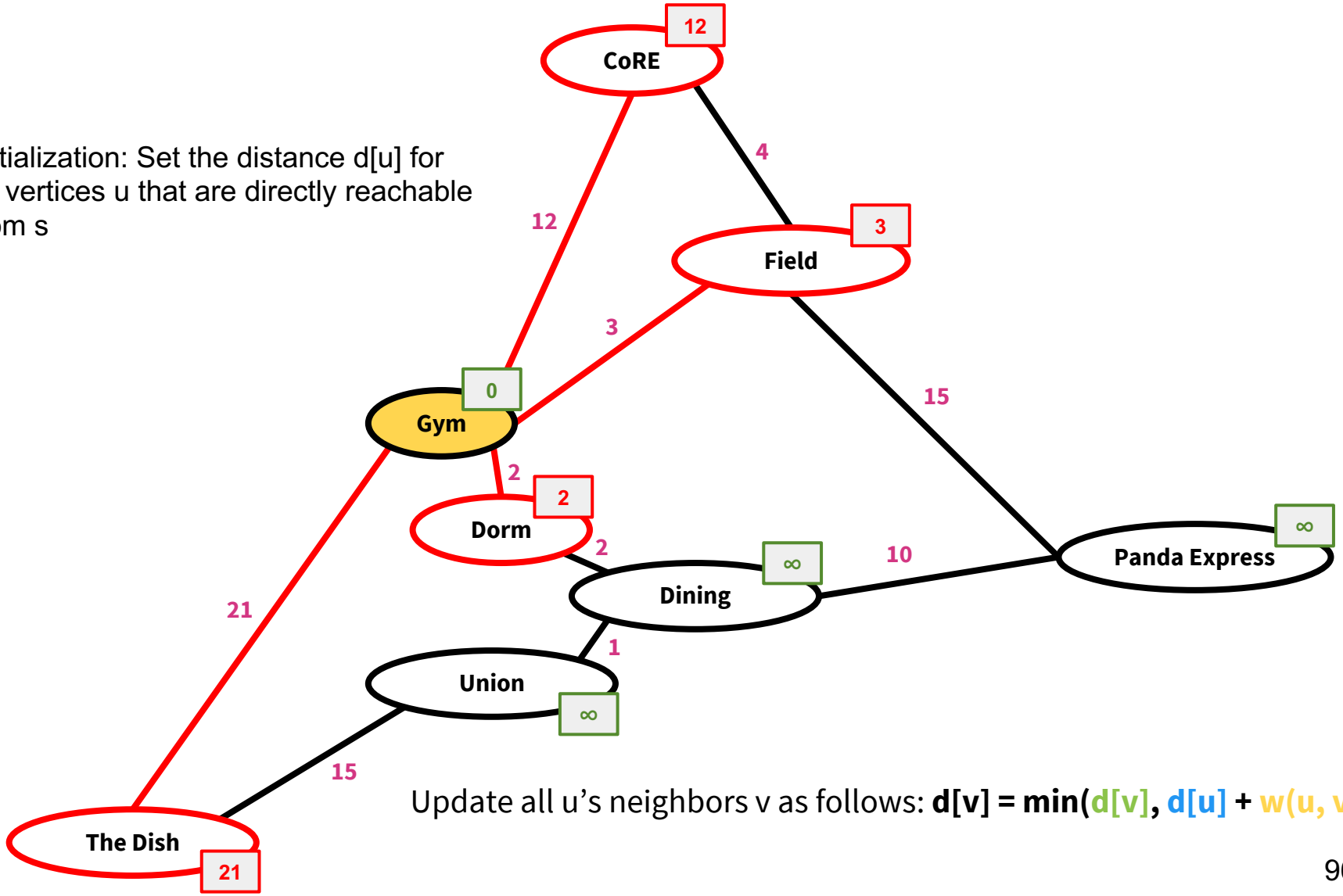
Every node u has a d[u] value, which means the currently known shortest path from source node s to this node u



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

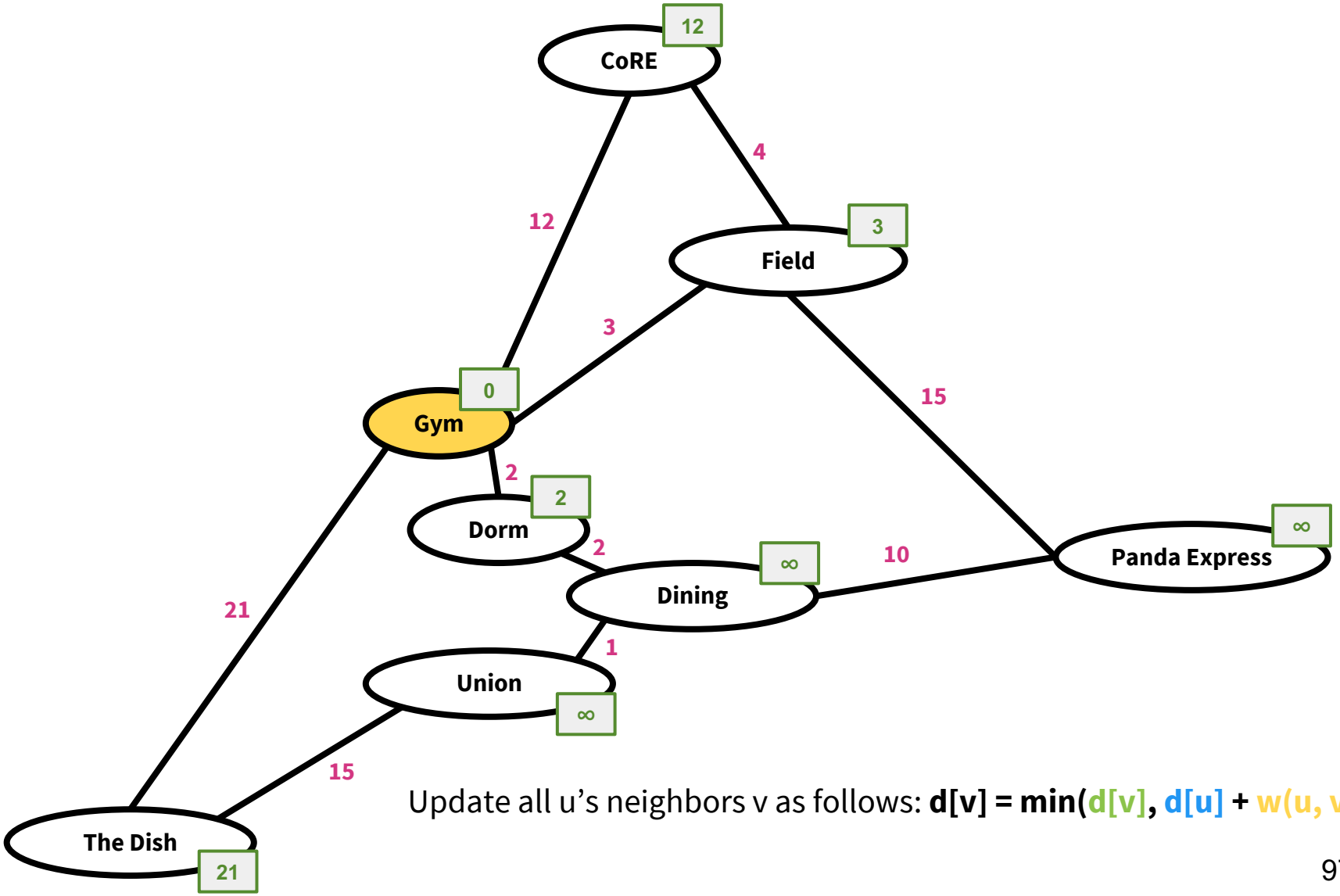
Dijkstra's Algorithm

Initialization: Set the distance d[u] for all vertices u that are directly reachable from s



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

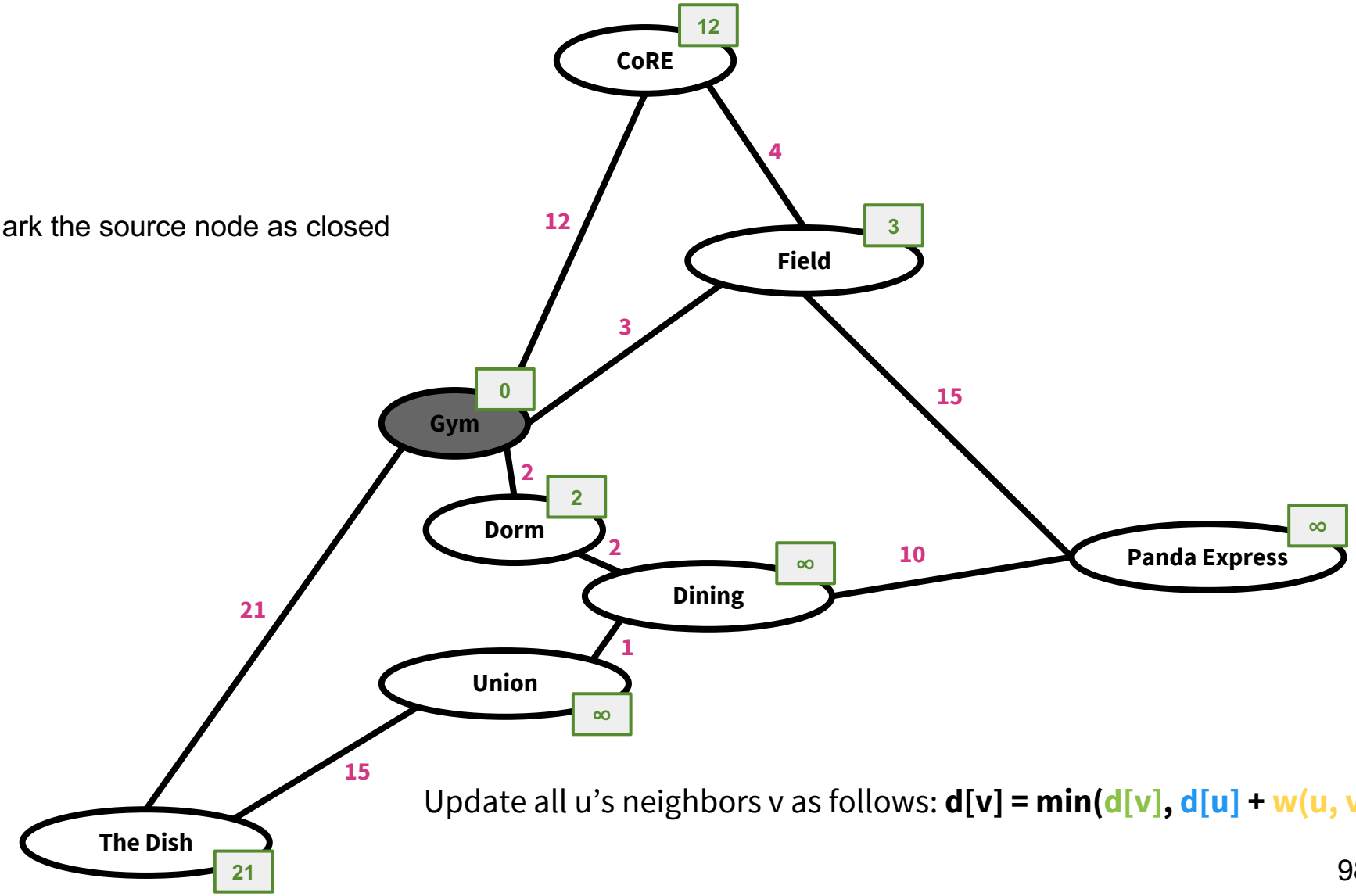
Dijkstra's Algorithm



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

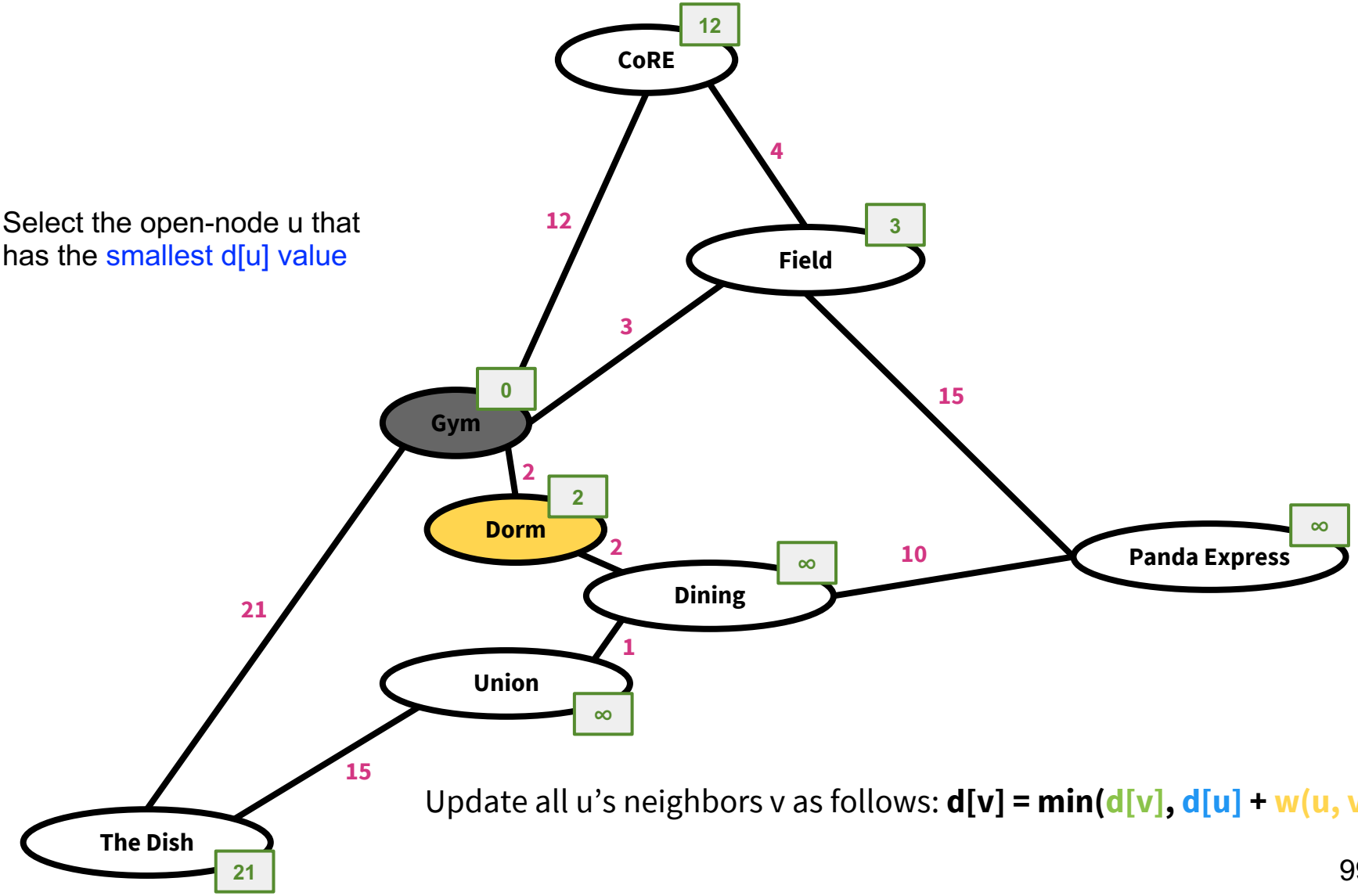
Mark the source node as closed



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

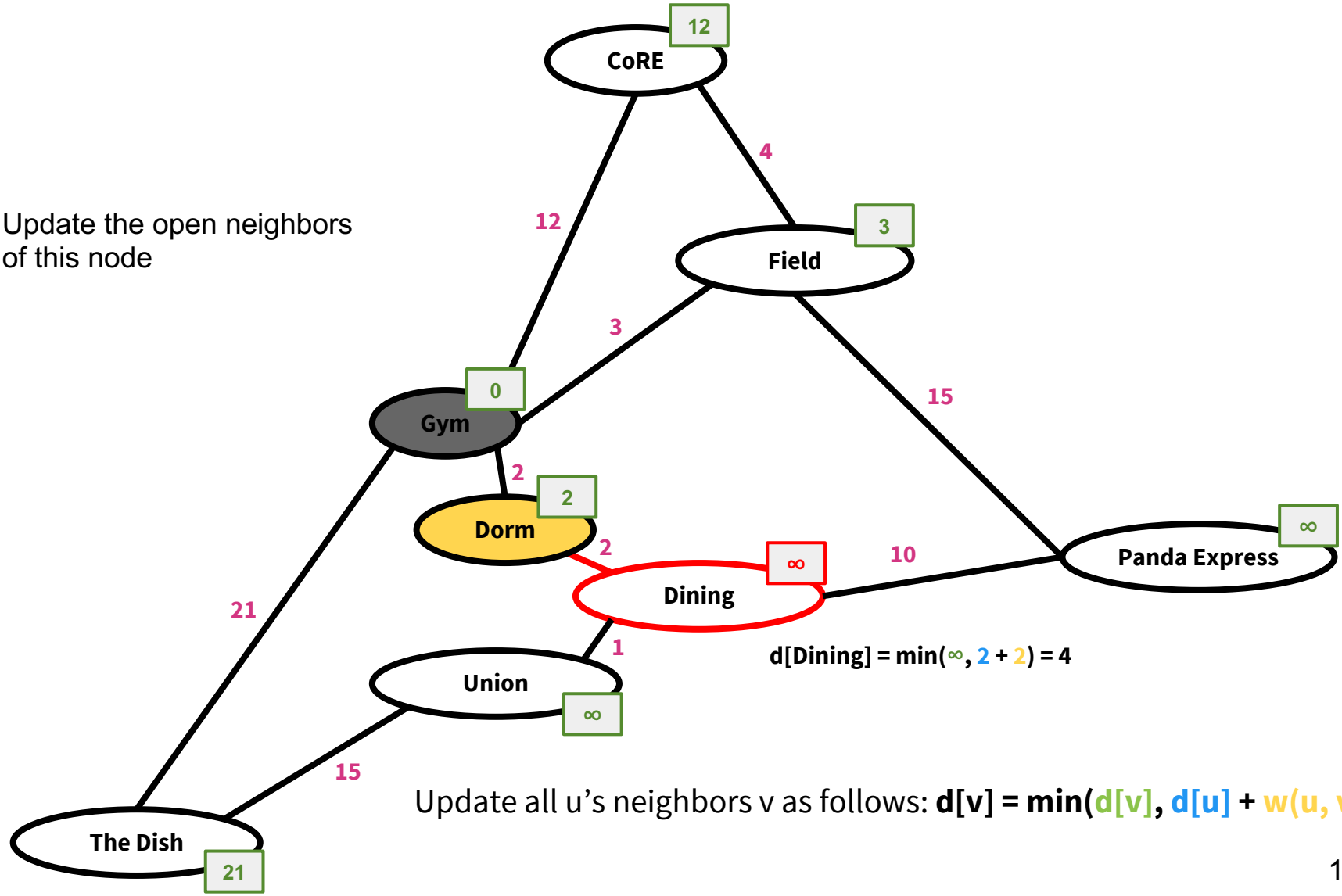
Select the open-node u that has the smallest d[u] value



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

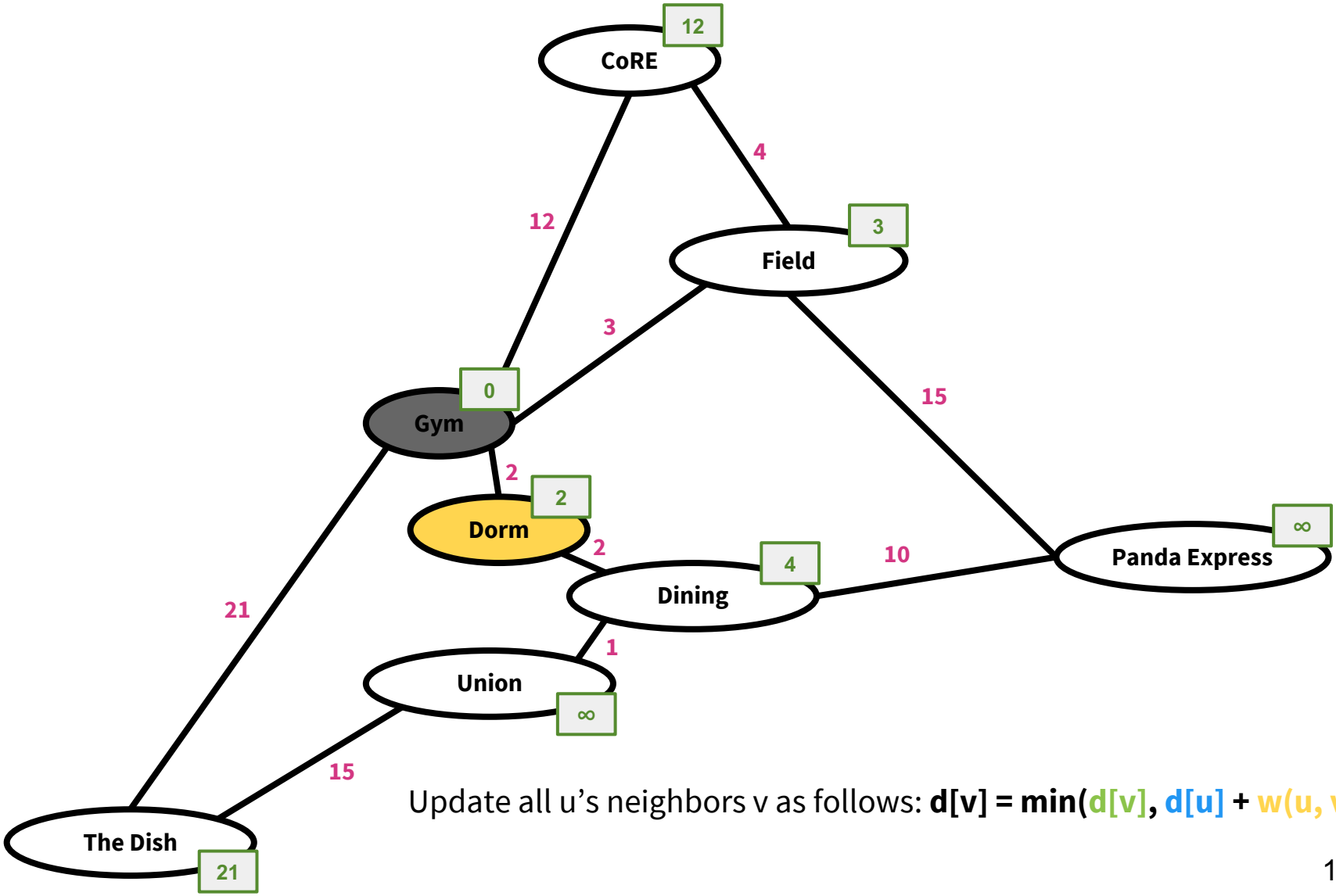
Dijkstra's Algorithm

Update the open neighbors of this node



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

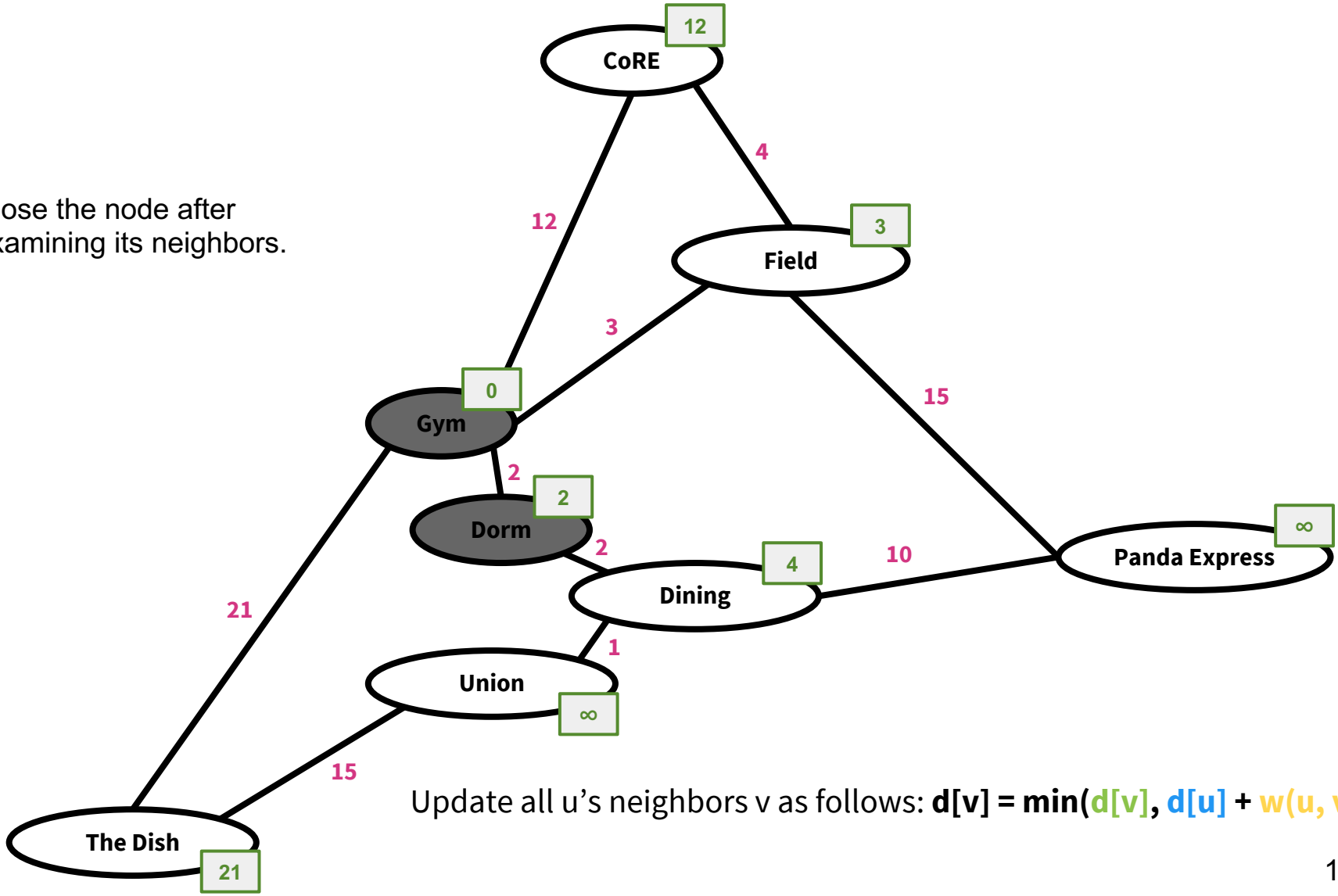
Dijkstra's Algorithm



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

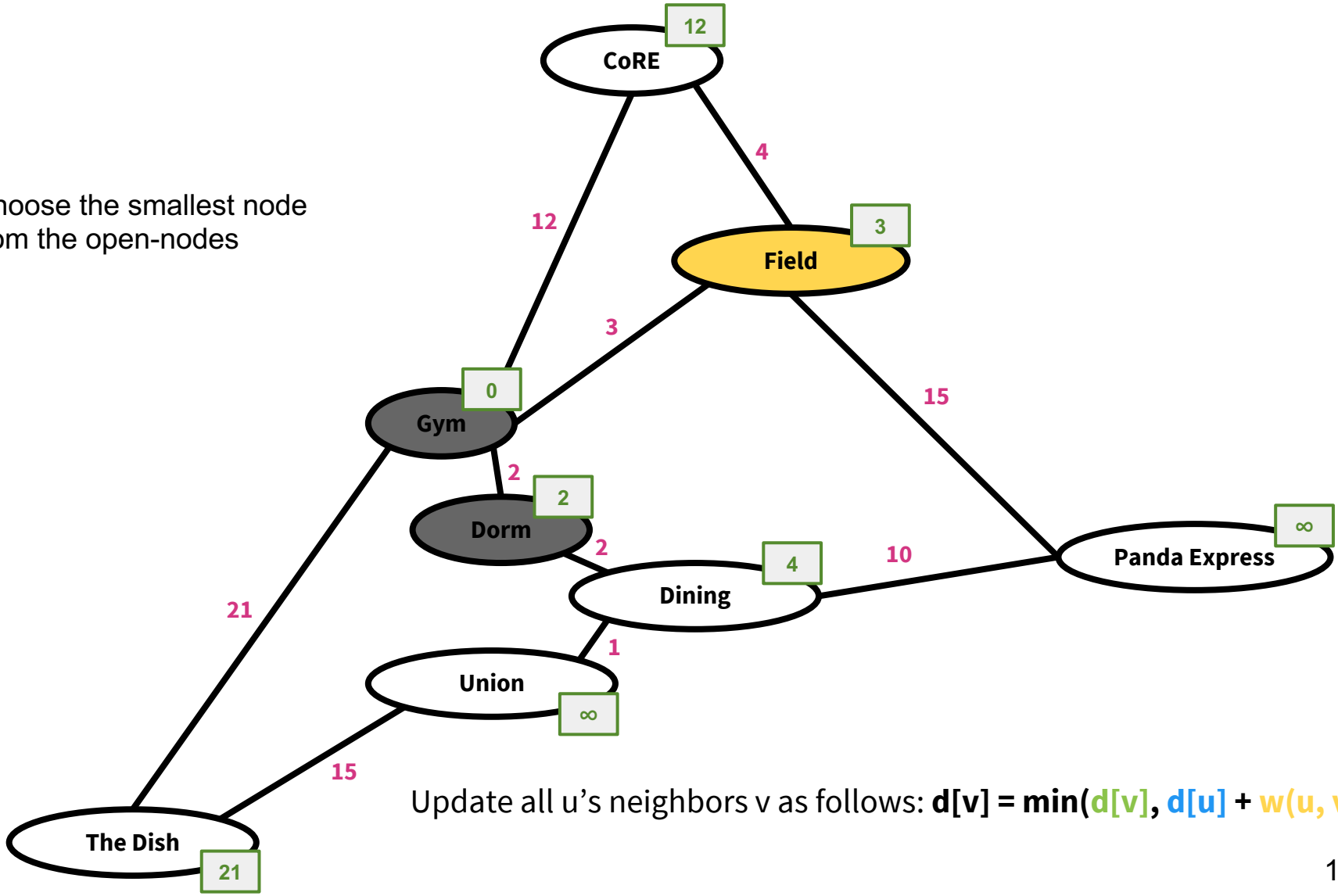
Close the node after examining its neighbors.



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

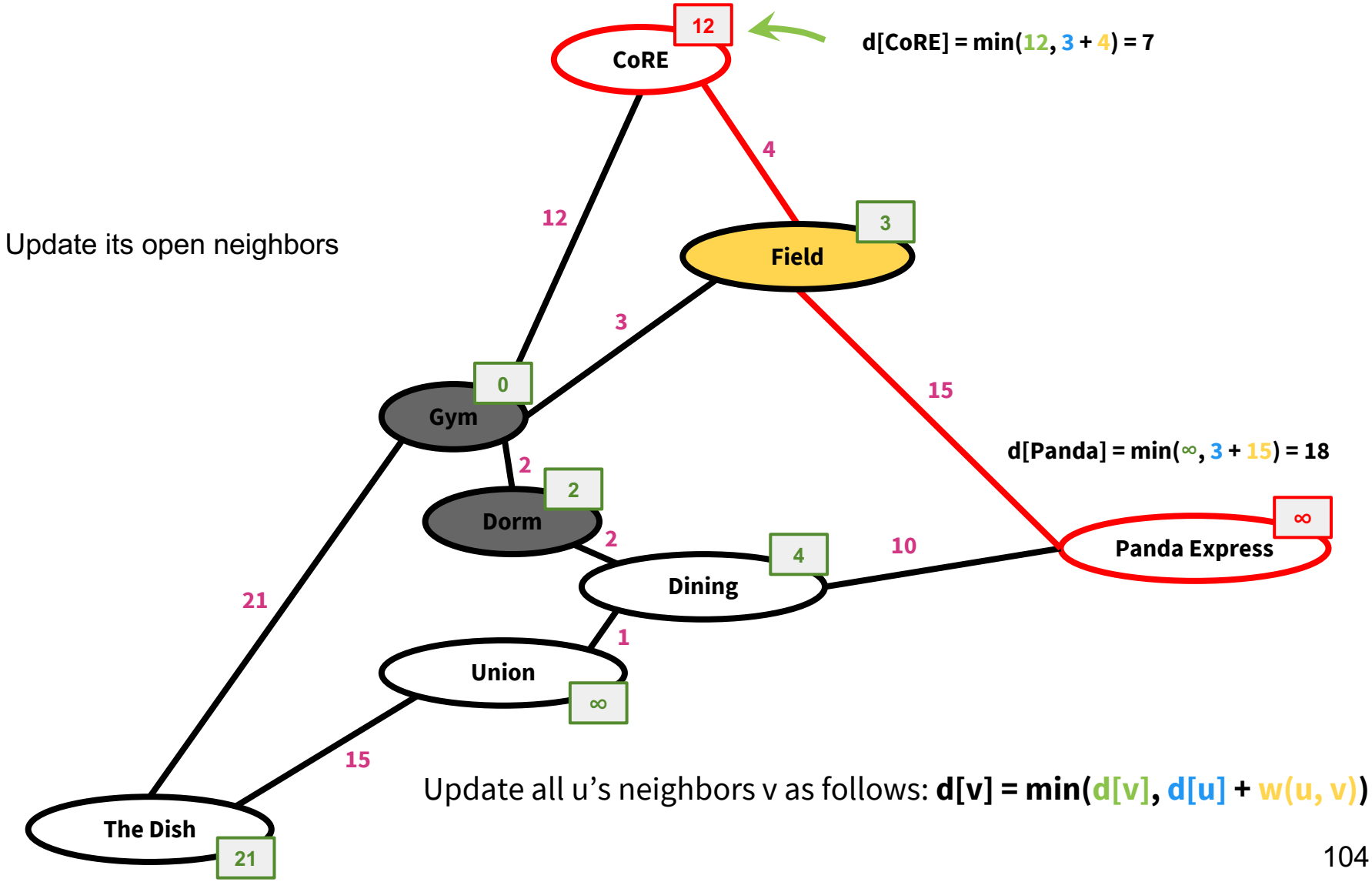
Dijkstra's Algorithm

Choose the smallest node from the open-nodes



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

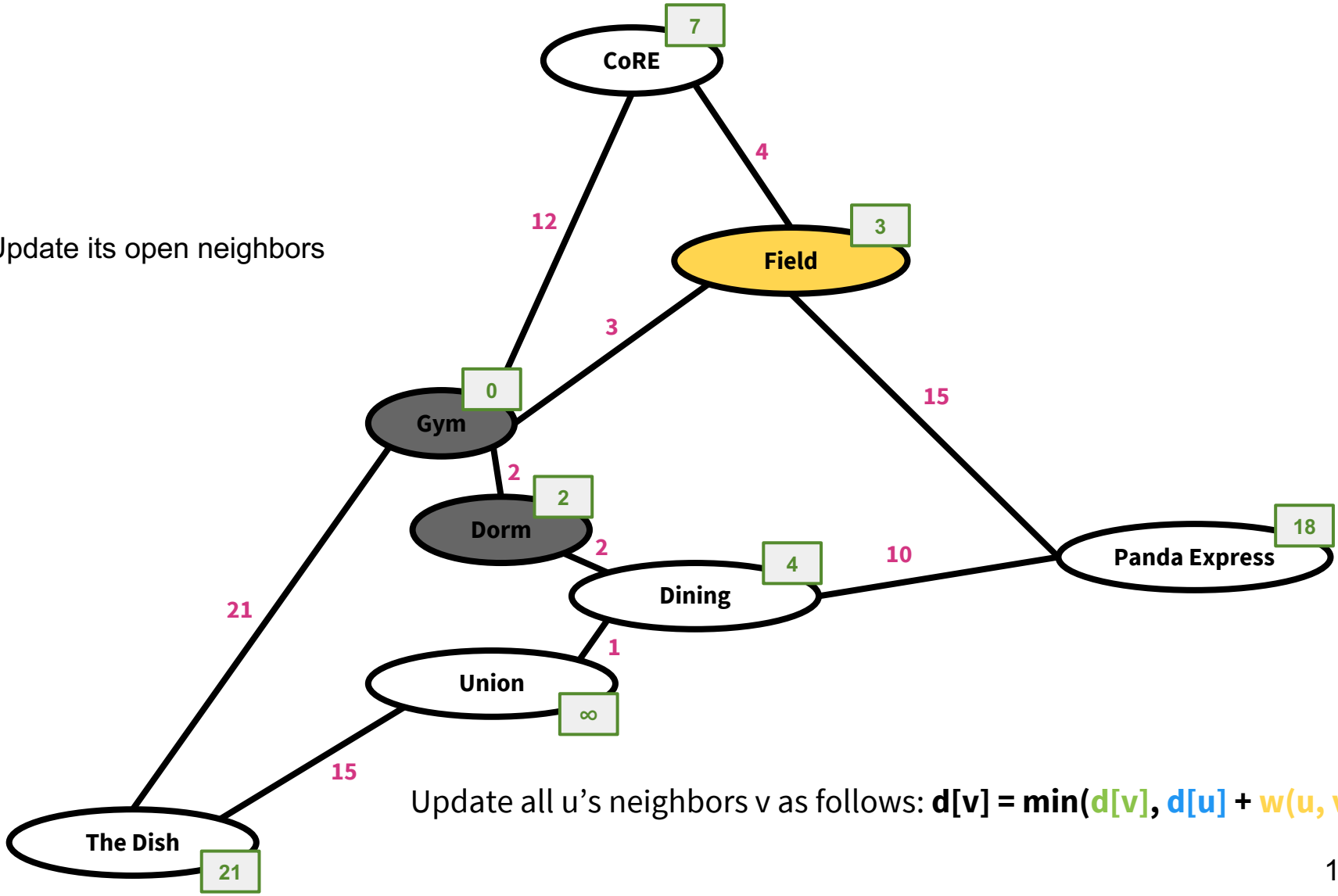
Dijkstra's Algorithm



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

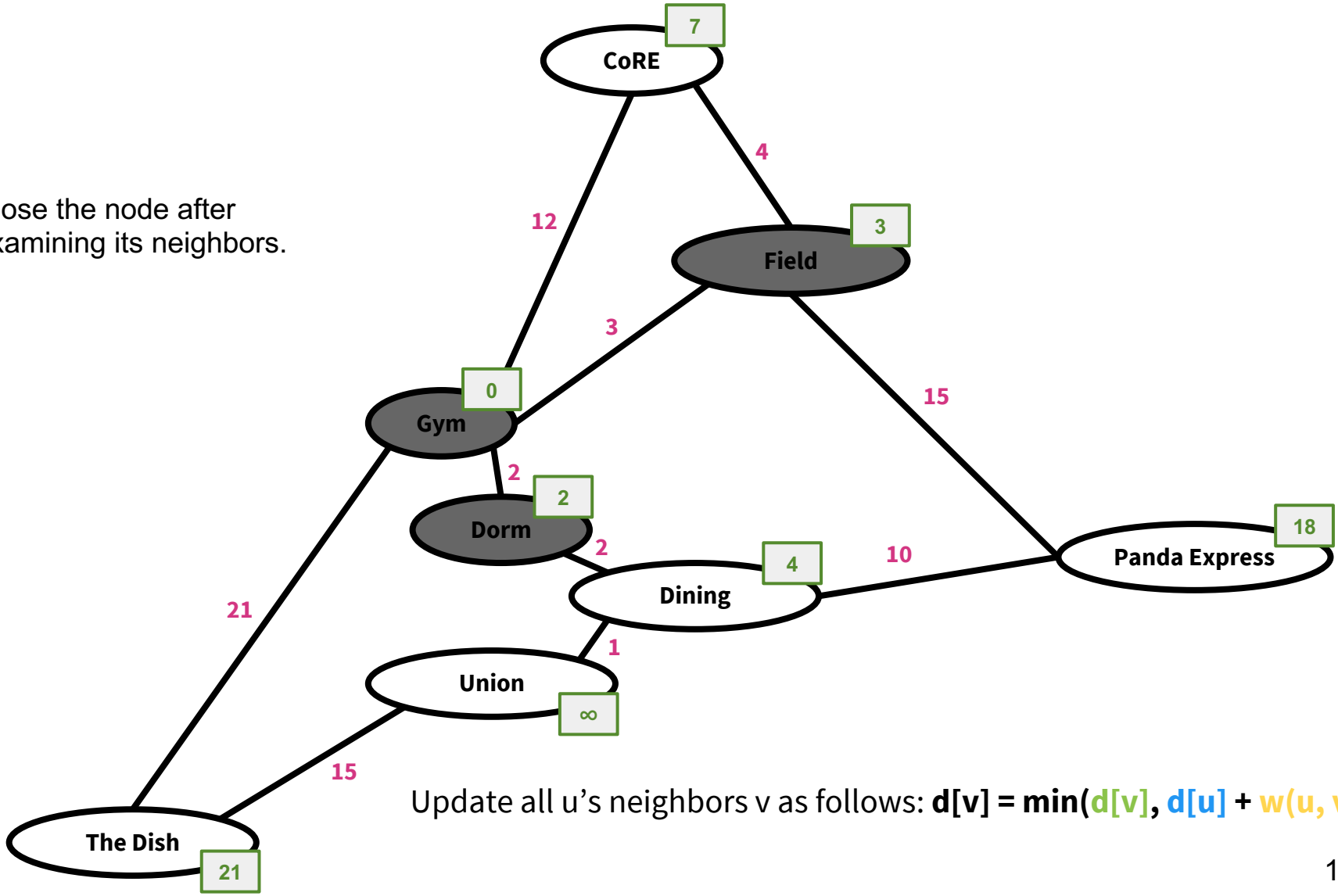
Update its open neighbors



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

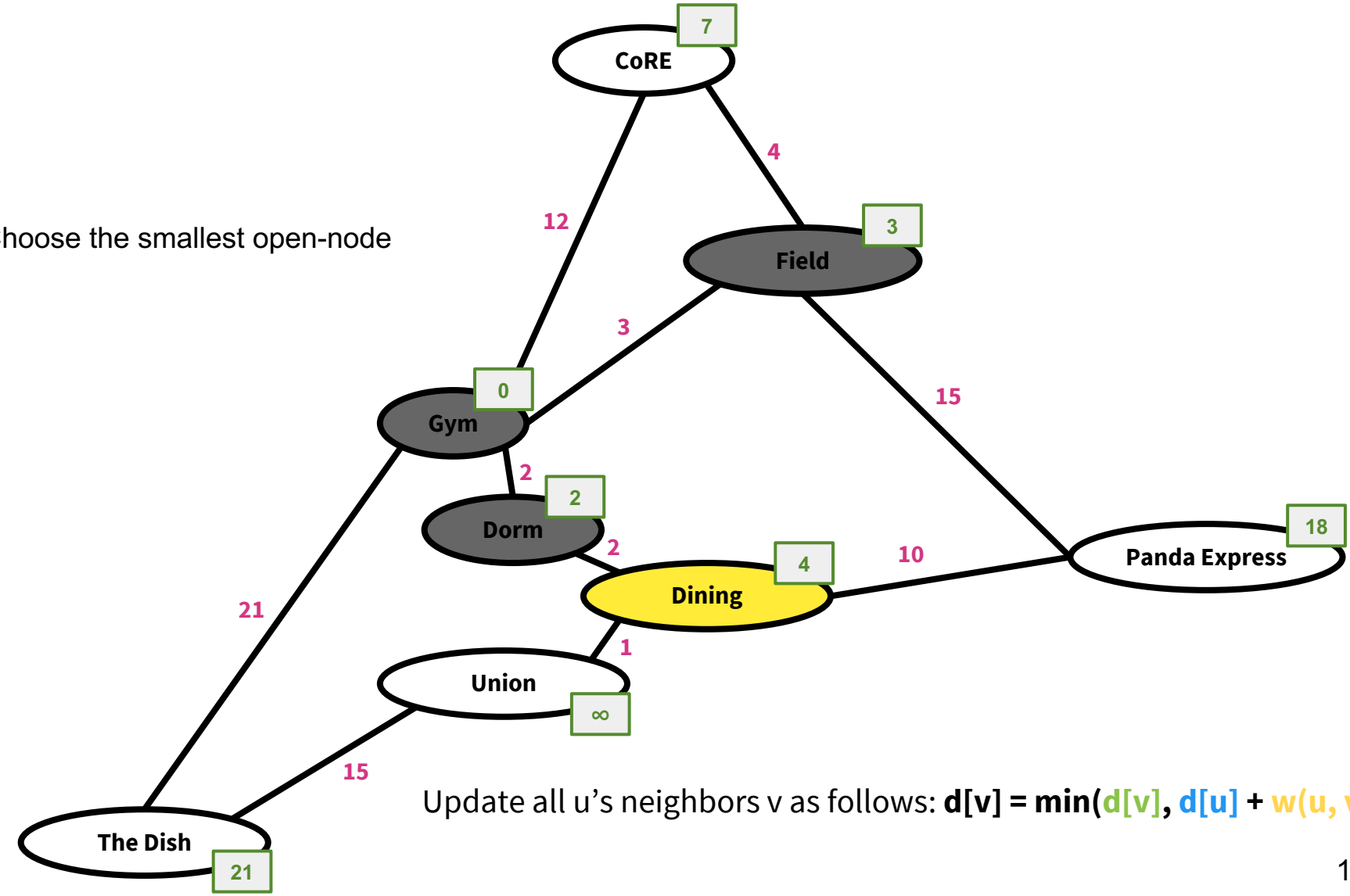
Close the node after examining its neighbors.



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

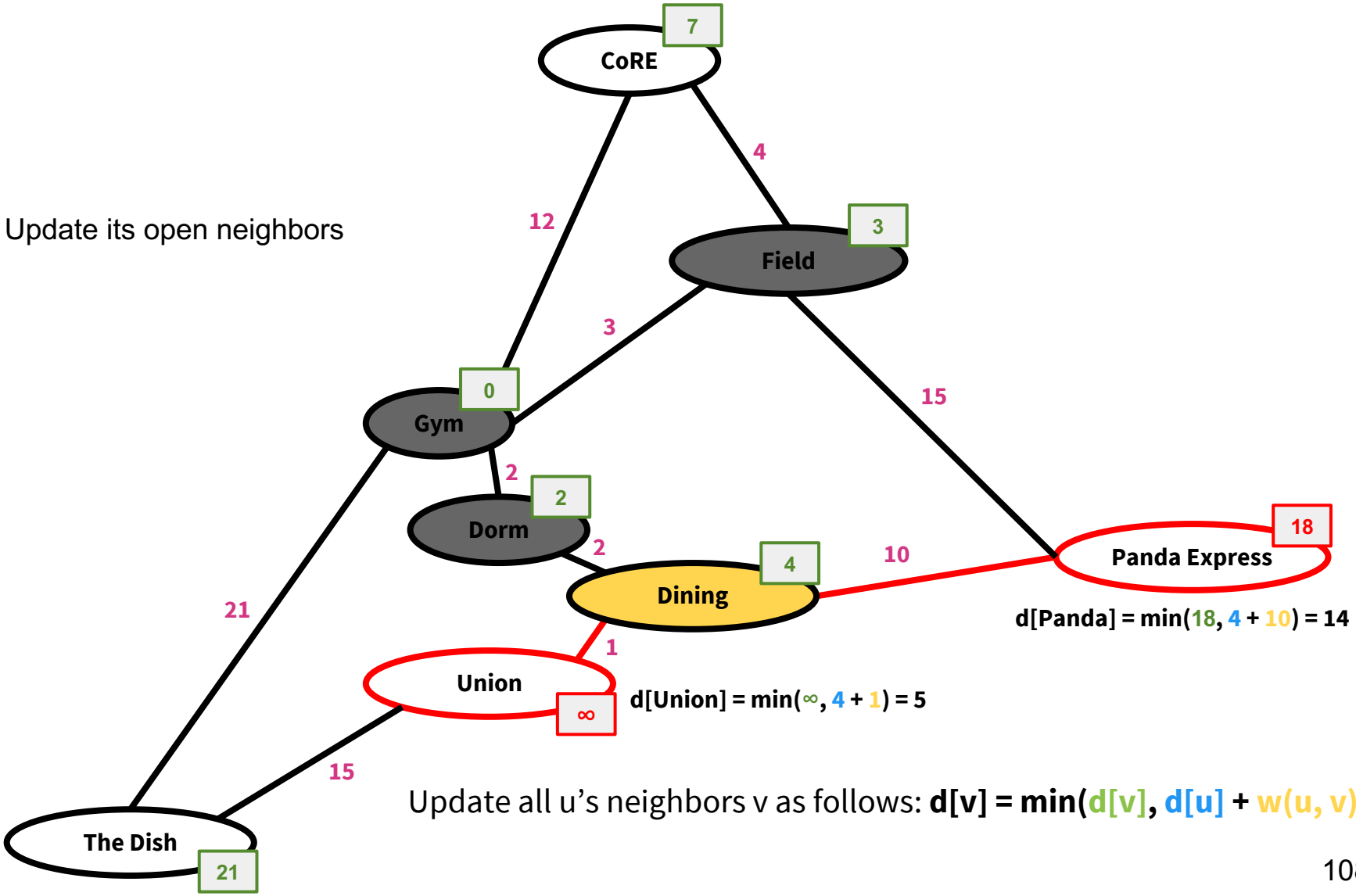
Choose the smallest open-node



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

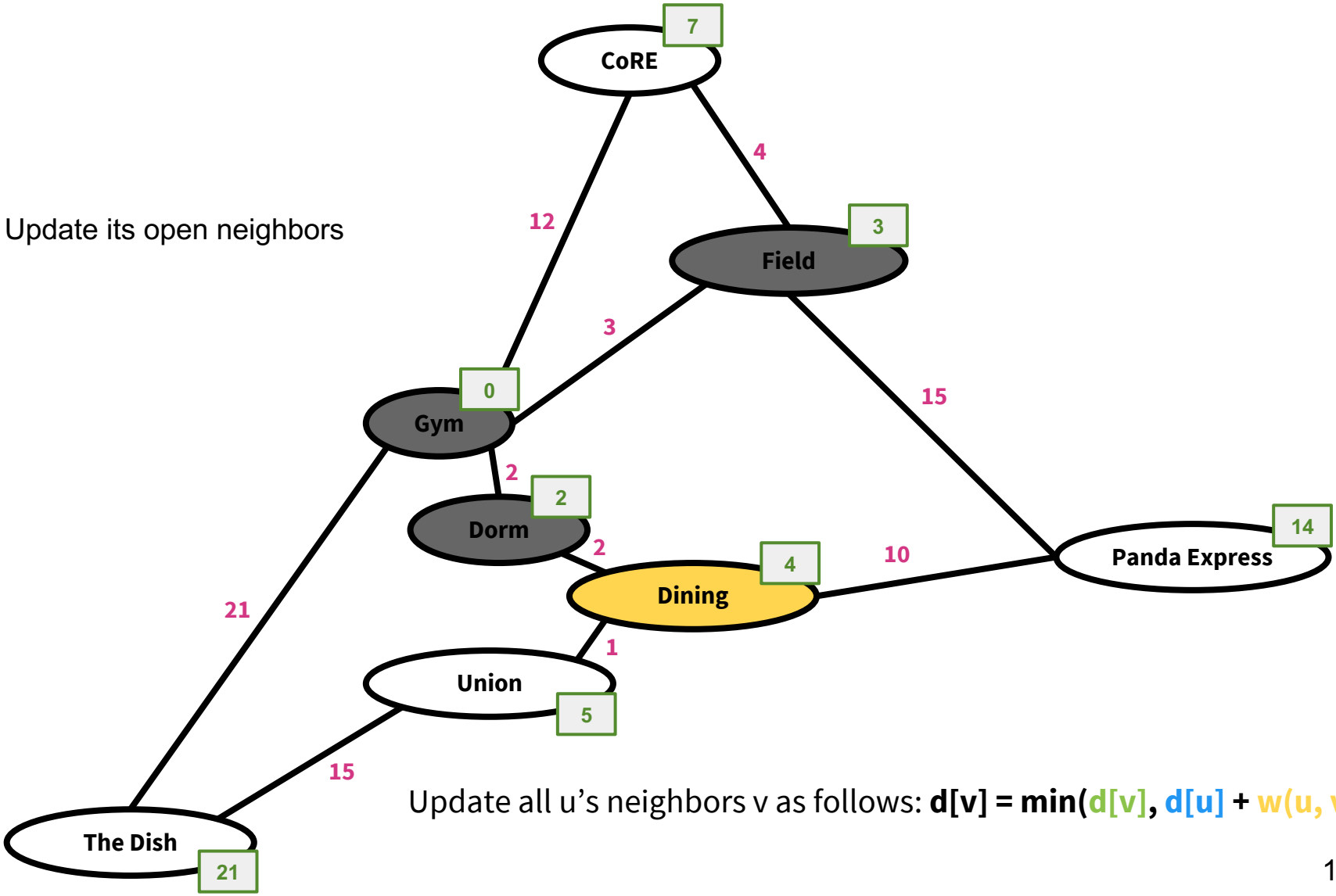
Update its open neighbors



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

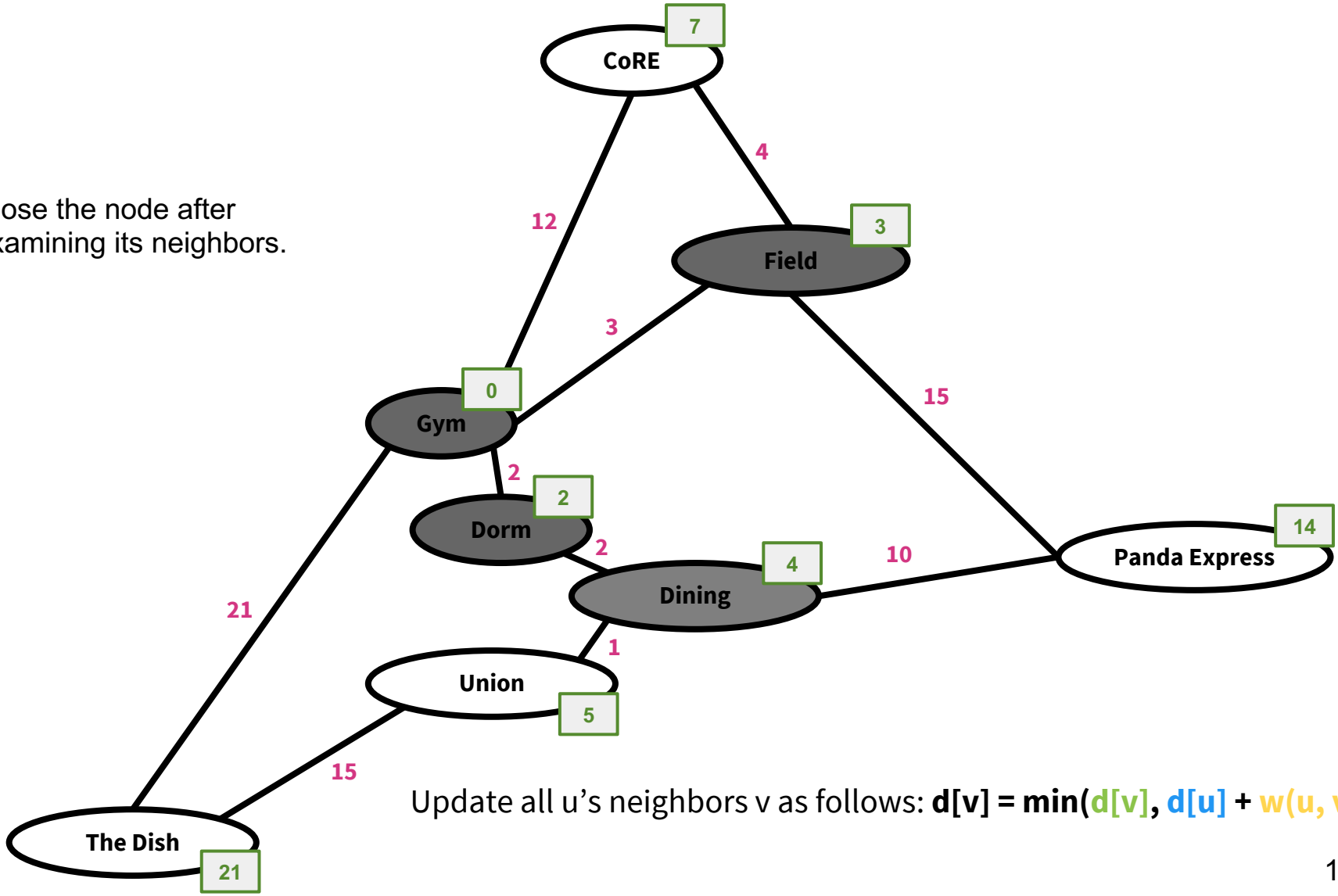
Update its open neighbors



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

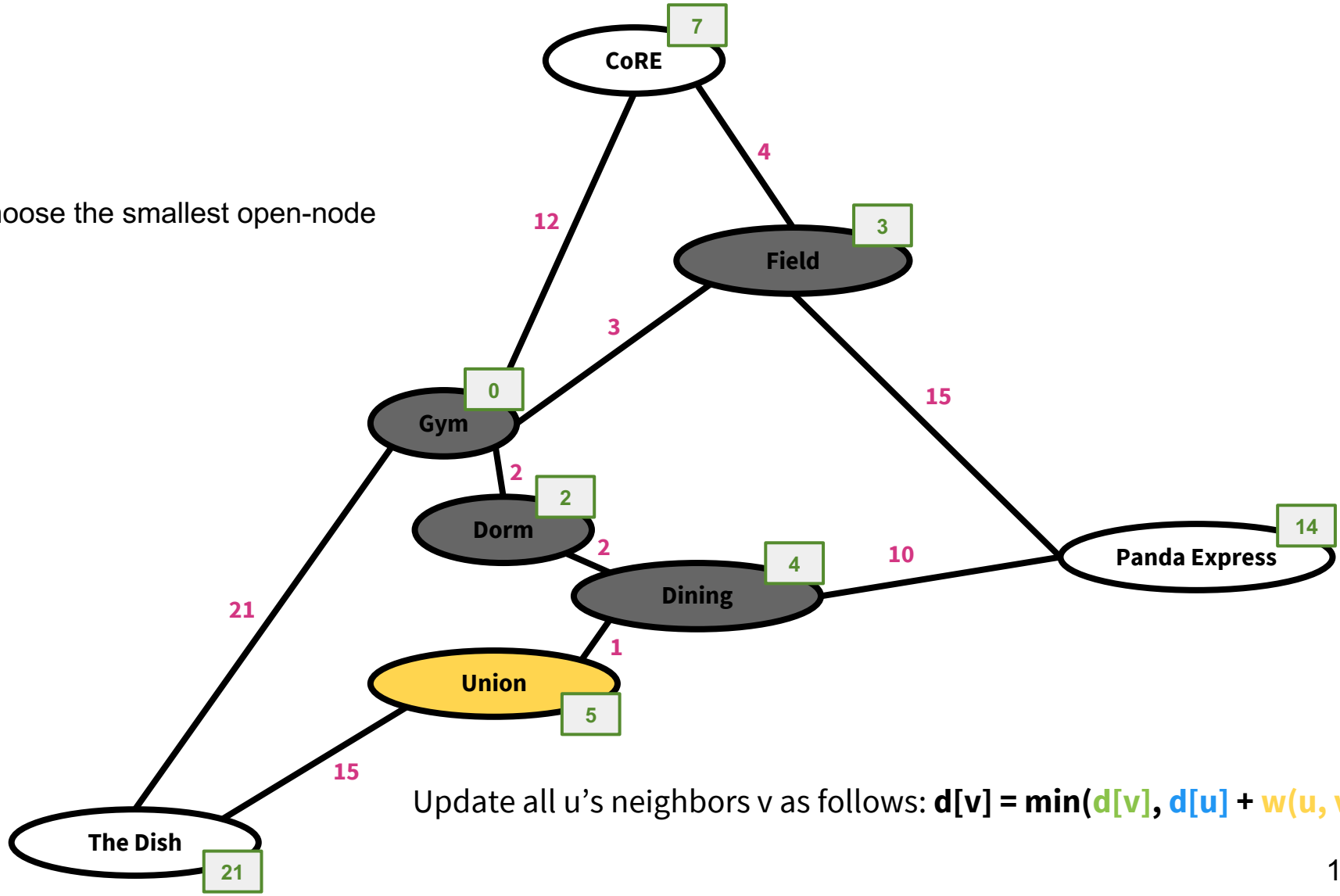
Close the node after examining its neighbors.



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

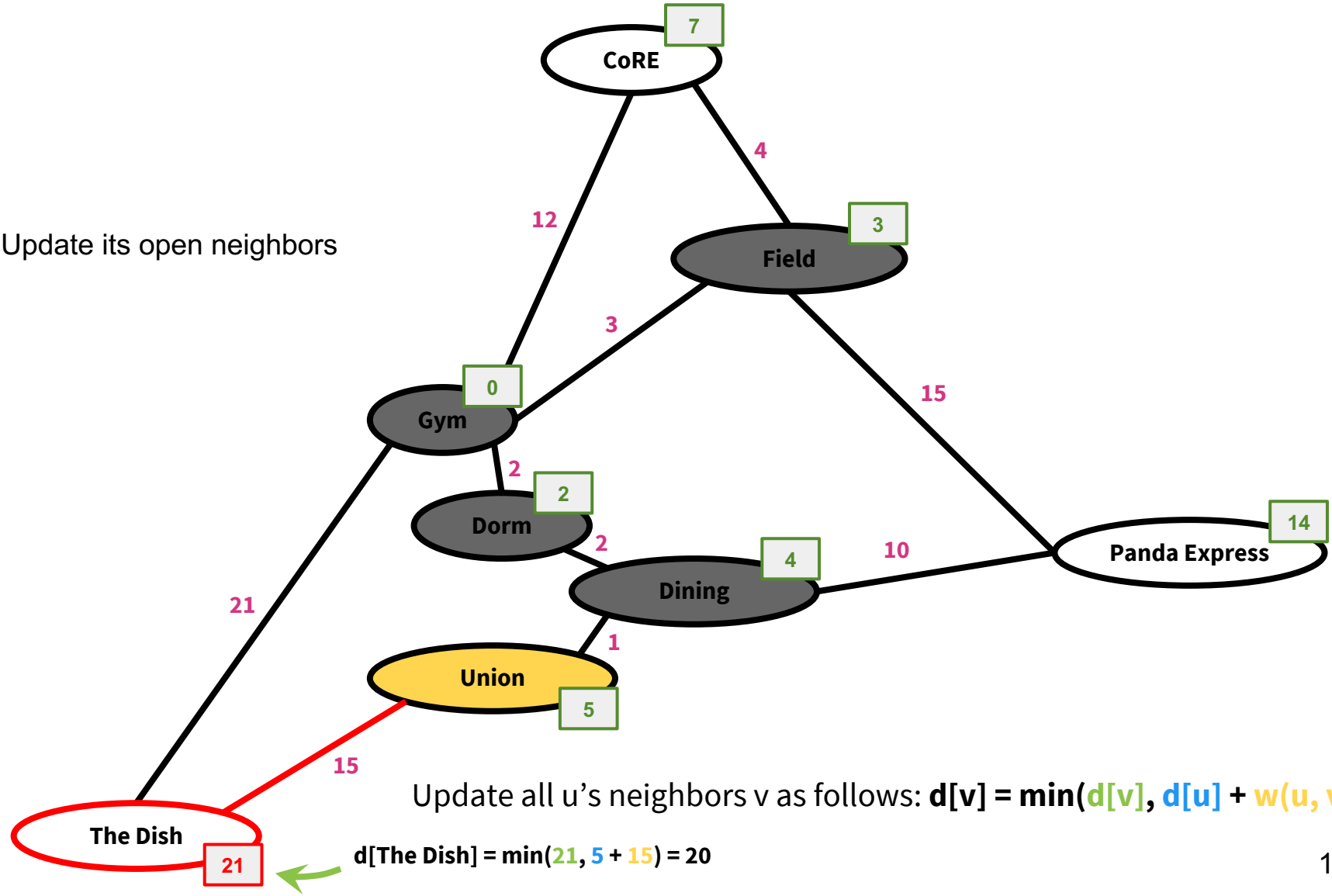
Choose the smallest open-node



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

Update its open neighbors

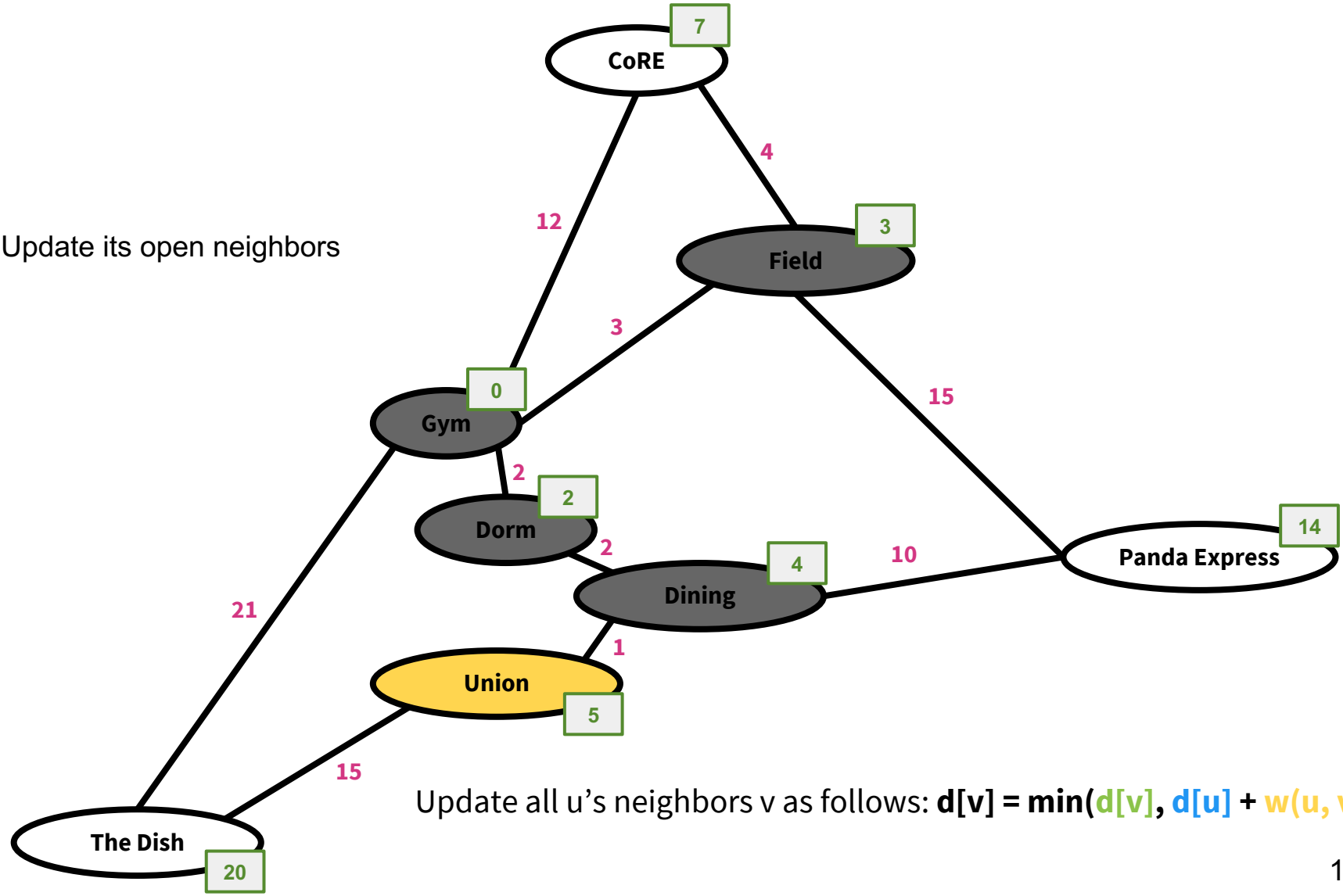


Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

d[The Dish] = min(21, 5 + 15) = 20

Dijkstra's Algorithm

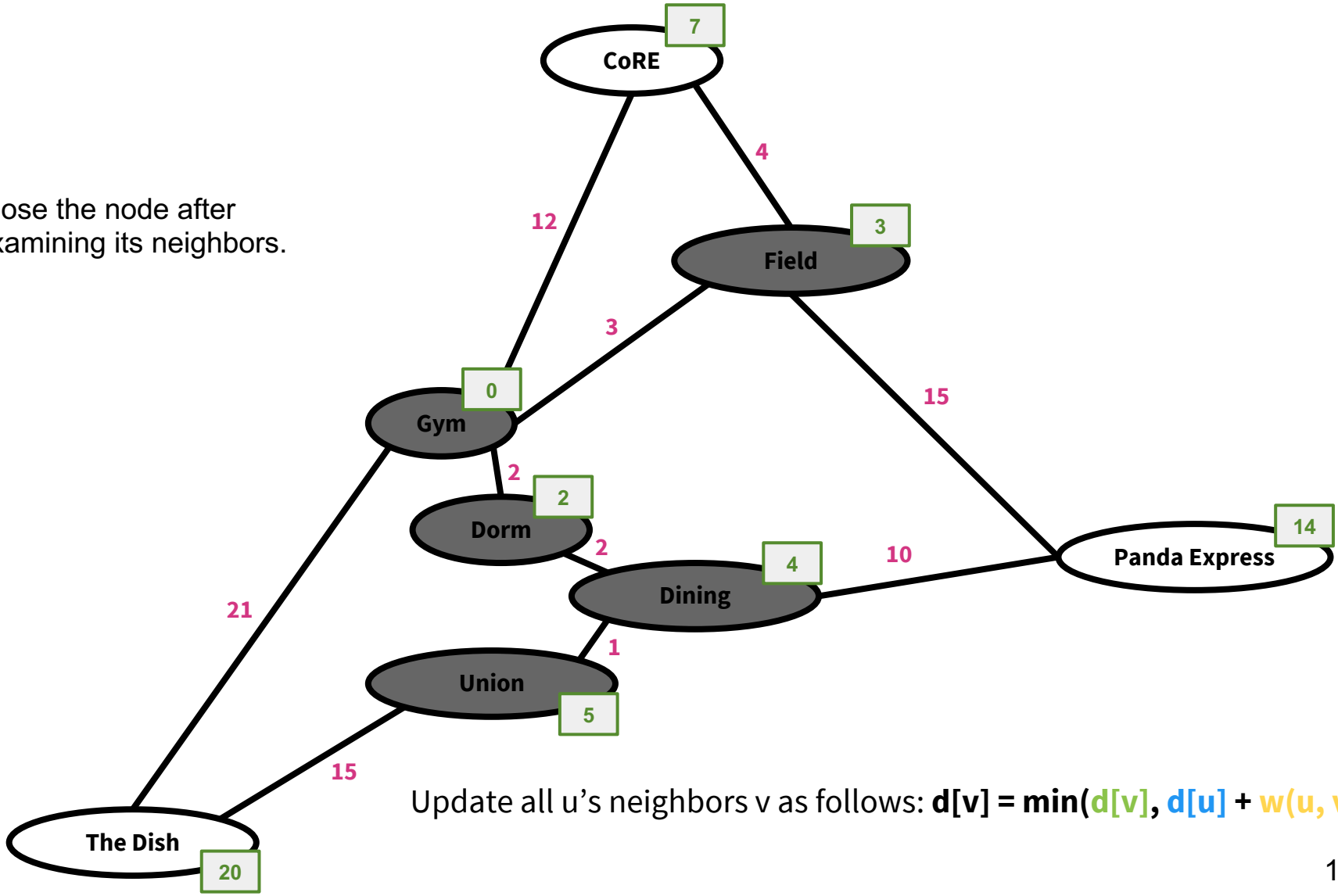
Update its open neighbors



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

Close the node after examining its neighbors.



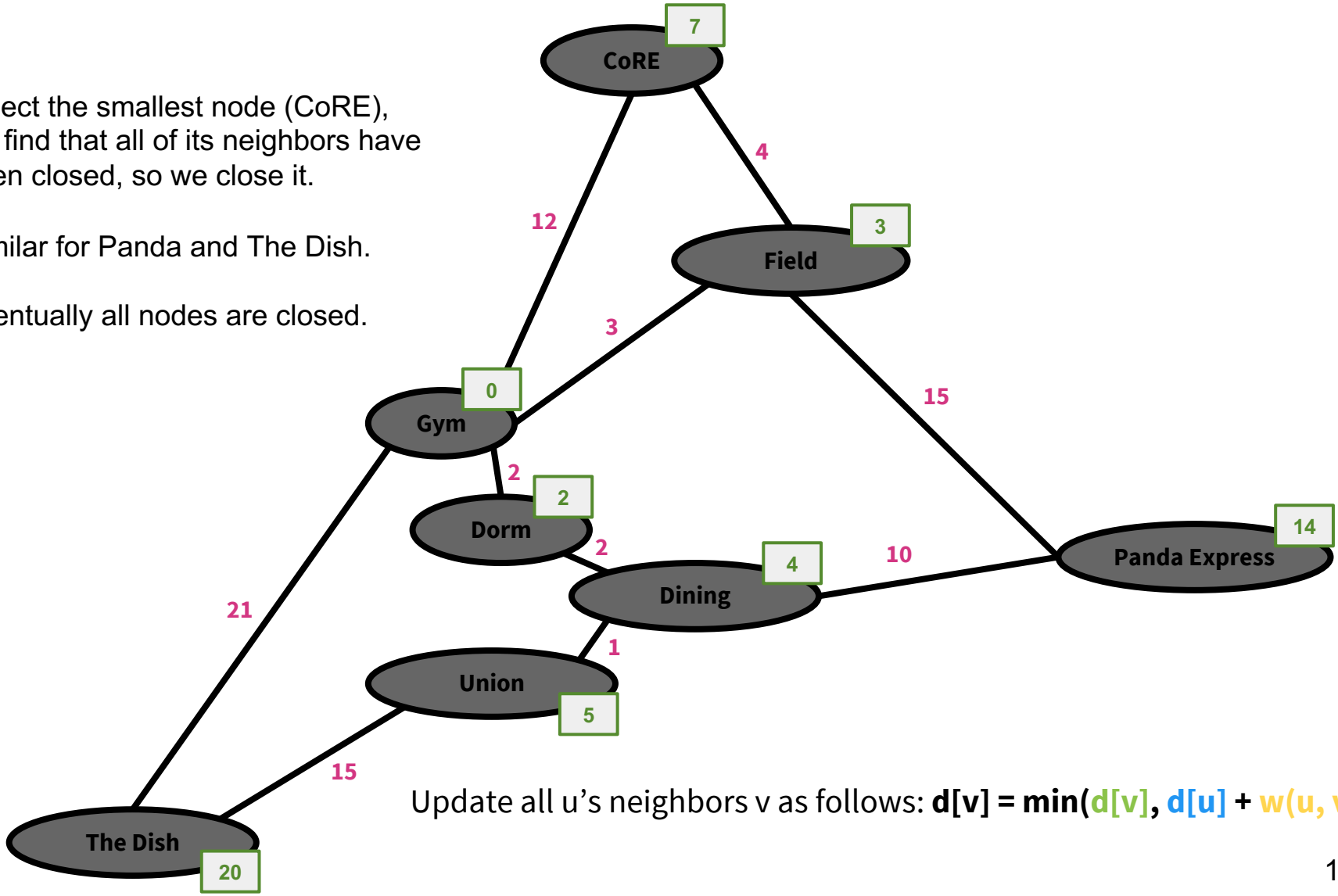
Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

Select the smallest node (CoRE), we find that all of its neighbors have been closed, so we close it.

Similar for Panda and The Dish.

Eventually all nodes are closed.



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

Why does this work?

Let s be the source node.

Theorem: After running Dijkstra's Algorithm, the estimate d[v] is the actual distance d(s, v).

Proof Outline:

Claim 1: For all v, $d[v] \geq d(s, v)$.

Claim 2: When a vertex v gets closed, $d[v] = d(s, v)$.

Together, claims 1 and 2 imply the theorem.

Dijkstra's Algorithm

Why does this work?

Let s be the source node.

Theorem: After running Dijkstra's Algorithm, the estimate d[v] is the actual distance d(s, v).

Proof Outline:

Claim 1: For all v, d[v] ≥ d(s, v).

Claim 2: When a vertex v gets closed, d[v] = d(s, v).

Together, claims 1 and 2 imply the theorem.

d[v] never increases, so **Claim 1** and **2** imply that d[v] weakly decreases until d[v] = d(s, v) then never changes again.

By the time we close v, d[v] = d(s, v).

All vertices are eventually closed (stopping condition in algorithm).

Therefore, all vertices end up with d[v] = d(s, v).

Dijkstra's Algorithm

Why does this work?

Claim 1: For all v, d[v] ≥ d(s, v).

Proof:

We proceed by induction on t, the number of iterations completed by the algorithm.

After t = 0 iterations, d(s, s) = 0 and d(s, v) ≤ ∞ which satisfy d[v] ≥ d(s, v).

For the inductive step, suppose the inductive hypothesis holds for iteration t. Then at iteration t + 1, the algorithm picks a vertex u and for each of its neighbors v sets: **d[v] = min(d[v], d[u] + w(u, v))** ≥ d(s, v).

By induction, d[v] ≥ d(s, v)

d[u] + w(u, v) ≥ d(s, u) + d(u, v) ≥ d(s, v)

Thus, the induction holds for t + 1.

Dijkstra's Algorithm

Why does this work?

Claim 2: When a vertex v gets closed, d[v] = d(s, v).

Proof:

We proceed by induction on t, the number of vertices marked as closed.

For the base case, note that after source node s is marked as "close", d[s] = d(s, s) = 0, which satisfies d[v] = d(s, v).

For the inductive step, assume that for all vertices v already marked as "closed", d[v] = d(s, v). Let x be the smallest remaining open node. We must prove d[x] = d(s, x) when x gets closed.

Dijkstra's Algorithm

Why does this work?

Claim 2: When a vertex v gets closed, d[v] = d(s, v).

Proof, cont.:

We proceed by contradiction. Suppose d[x] ≠ d(s, x).

Let p be the shortest path from s to x. There must exist some z on p such that d[z] = d(s, z). Let z be the closest such vertex to x. We know d[z] = d(s, z) ≤ d(s, x) < d[x].

z must exist since, at the very least, s is part of the shortest path, and d[s] = d(s, s).

Weights are non-negative.

Claim 1 implies d(s, x) ≤ d[x] and we assumed that d[x] ≠ d(s, x).

Otherwise, z would be the vertex with minimum distance estimate.

Therefore, d[z] < d[x]. Since d[z] < d[x] and x is the smallest open node, then we must have already processed z ahead of x, i.e., z must have already been "closed".

Dijkstra's Algorithm

Why does this work?

Claim 2: When a vertex v gets closed, d[v] = d(s, v).

Proof, cont.:

Since z is already closed, by inductive assumption we have d[z] = d(s, z),

and the edges out of z, including the edge (z, z') (where z' is also on p) have been processed by the algorithm with d[z']=min(d[z'],d(z)+w(z,z')), thus we have d[z'] ≤ d(z)+w(z, z') = d(s, z) + w(z, z') = d(s, z').

The last equality holds because both z and z' are on the shortest path from s to x, thus the path s-z-z' must be the shortest path from s to z'.

According to Claim 1 (for all v, d[v] ≥ d(s, v)), it can only be d[z']=d(s, z').

However, this contradicts z being the closest vertex on p to x satisfying d[z] = d(s, z). Thus, our assumption that d[x] ≠d(s, x) must be false, and it follows that d[x] = d(s, x) when x gets closed.

Summary

Graph Algorithms I

Basics Notations for Graphs
Depth First Search (DFS) and Topological Ordering
Breath First Search (BFS) and Shortest Path (for unweighted graphs)
Dijkstra's Algorithm for Single-Source Shortest Path Problem (on weighted graphs)

Summary

Graph Algorithms I

Basics Notations for Graphs
Depth First Search (DFS) and Topological Ordering
Breath First Search (BFS) and Shortest Path (for unweighted graphs)
Dijkstra's Algorithm for Single-Source Shortest Path Problem (on weighted graphs)

Acknowledgement: Part of the materials are adapted from Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.