Graph Algorithms I

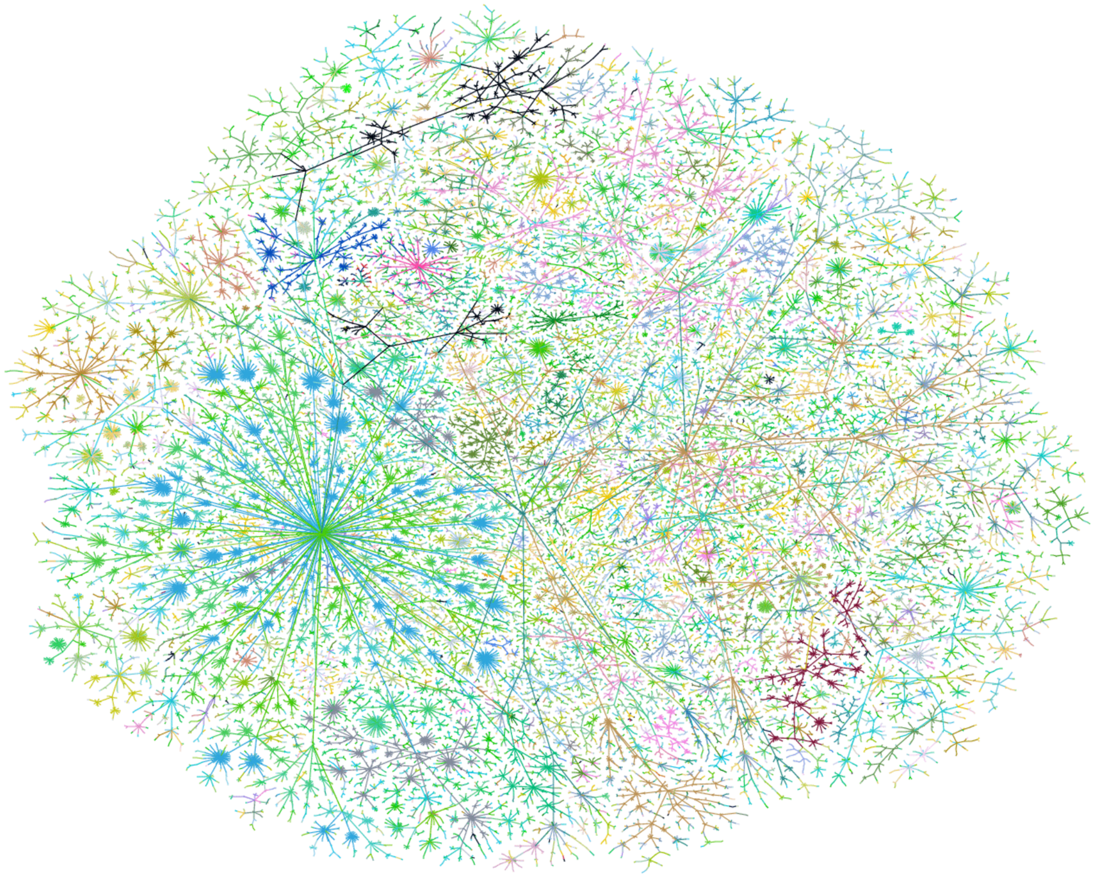
Outline for Today

- 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

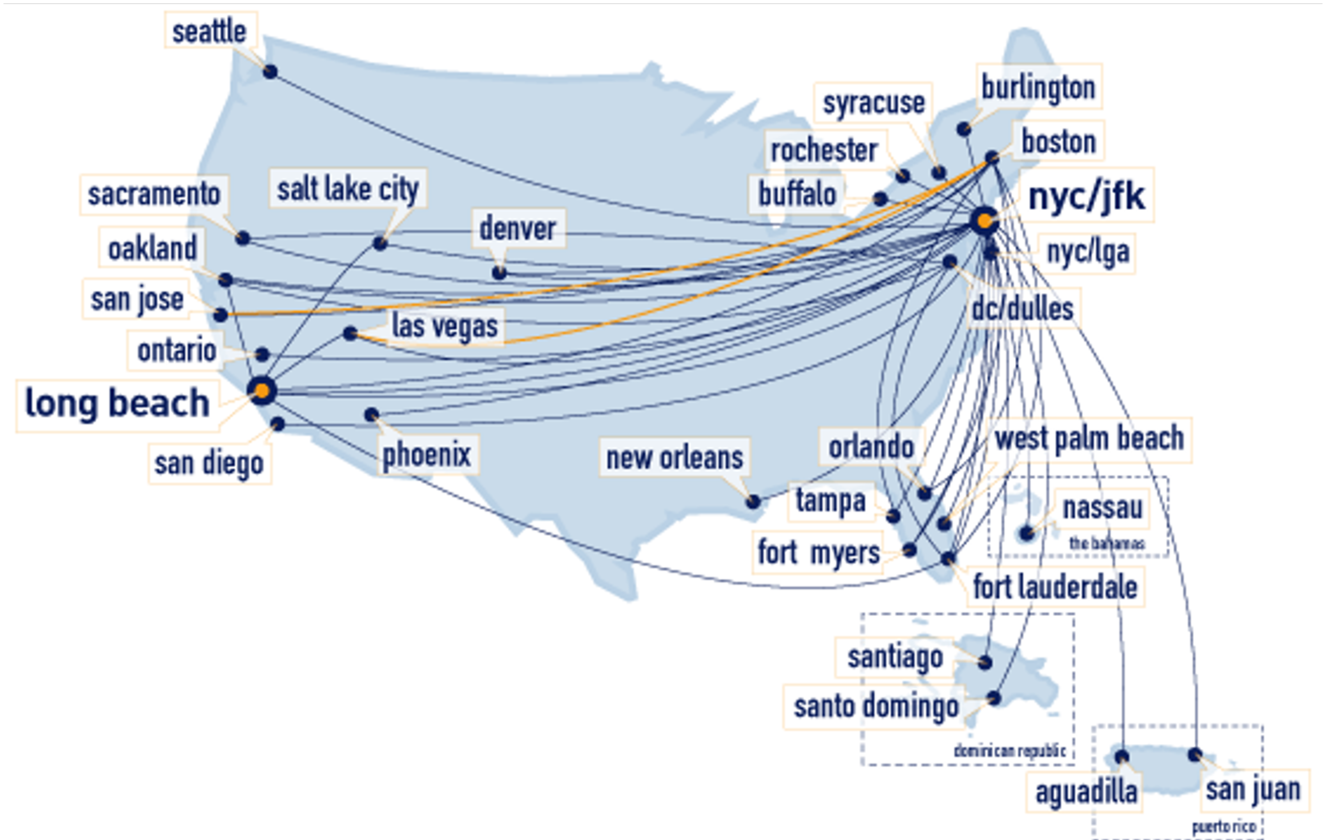
Examples of Graphs

The Internet (circa 1999)



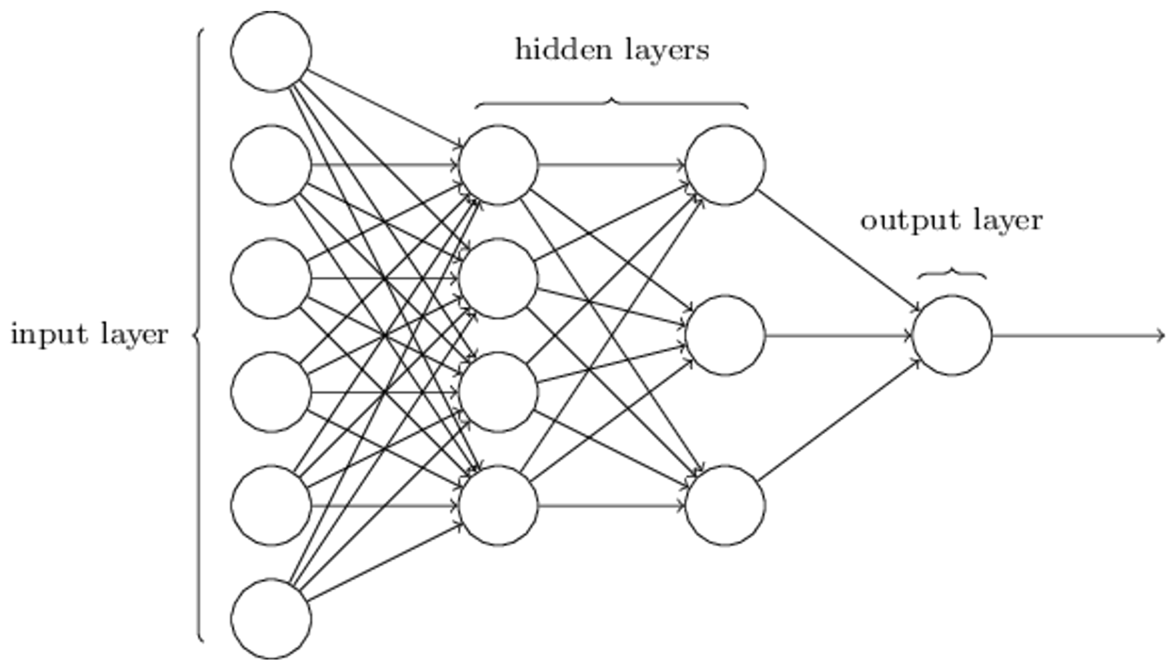
Examples of Graphs

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.

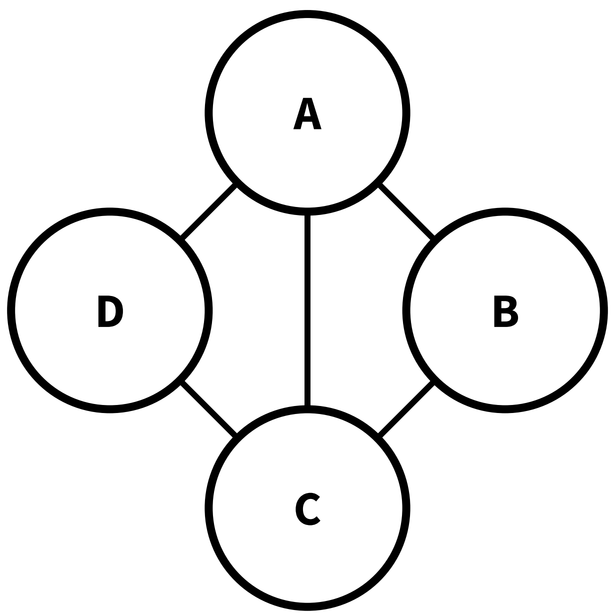
Finding strongly connected components for community detection or clustering.

Finding the topological ordering to respect dependencies.

Undirected Graphs

An undirected graph has vertices and edges.

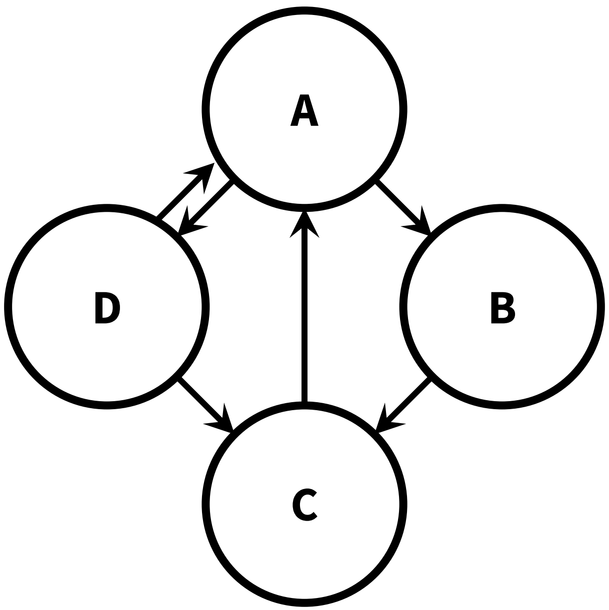
- V is the set of vertices and E is the set of edges.
- Formally, an undirected graph is G = (V, E).
- e.g. V = {A, B, C, D} and E = { {A, B}, {A, C}, {A, D}, {B, C}, {C, D} }



Directed Graphs

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

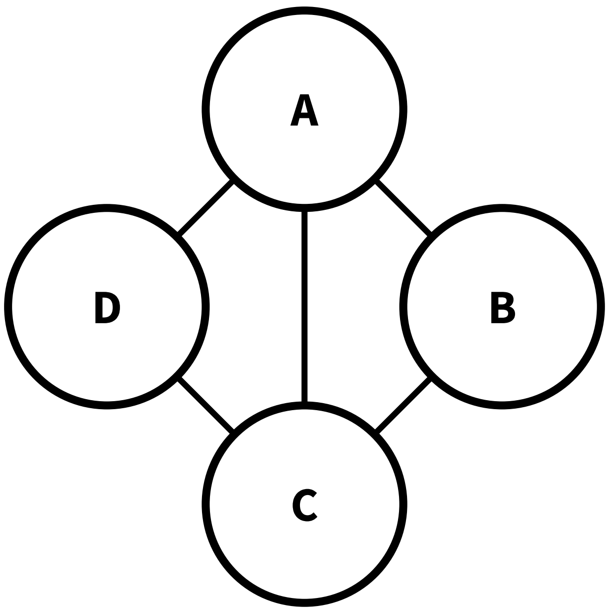
- V is the set of vertices and E is the set of directed edges.
- Formally, a directed graph is G = (V, E)
- e.g. V = {A, B, C, D} and E = { [A, B], [A, D], [B, C], [C, A], [D, A], [D, C] }



Graph Representations

How do we represent graphs?

(1) Adjacency matrix



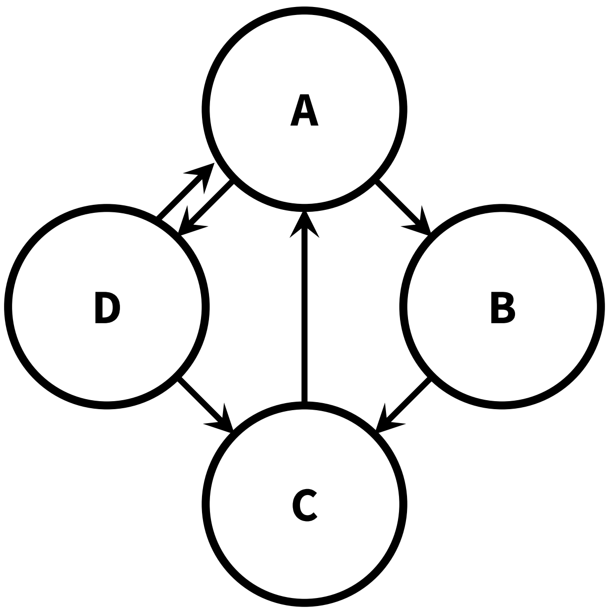
$$
\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}
\end{array}
$$

Symmetric matrix

Graph Representations

How do we represent graphs?

(1) Adjacency matrix



destination

source

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

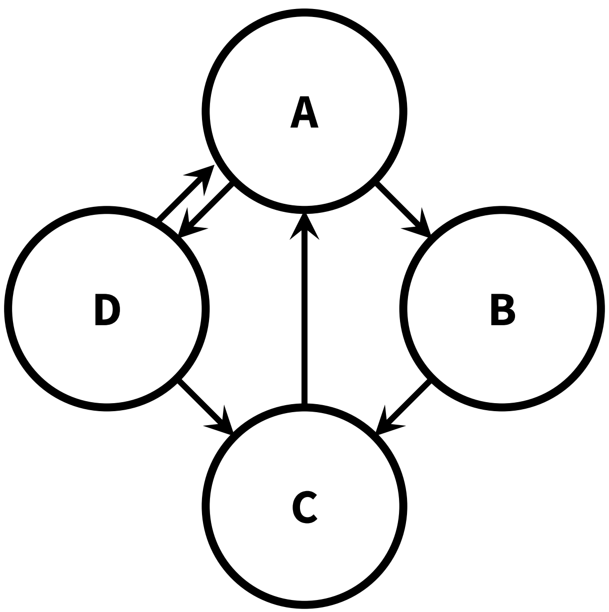
Unsymmetric matrix

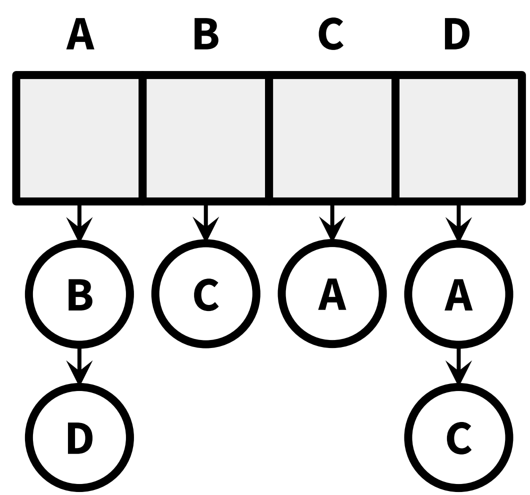
Graph Representations

How do we represent graphs?

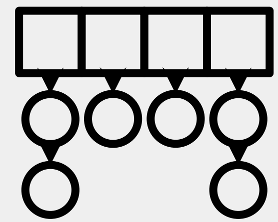
(1) Adjacency matrix

(2) Adjacency list





Graph Representations

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

Generally, better for sparse graphs.

We'll assume this representation, unless otherwise stated.

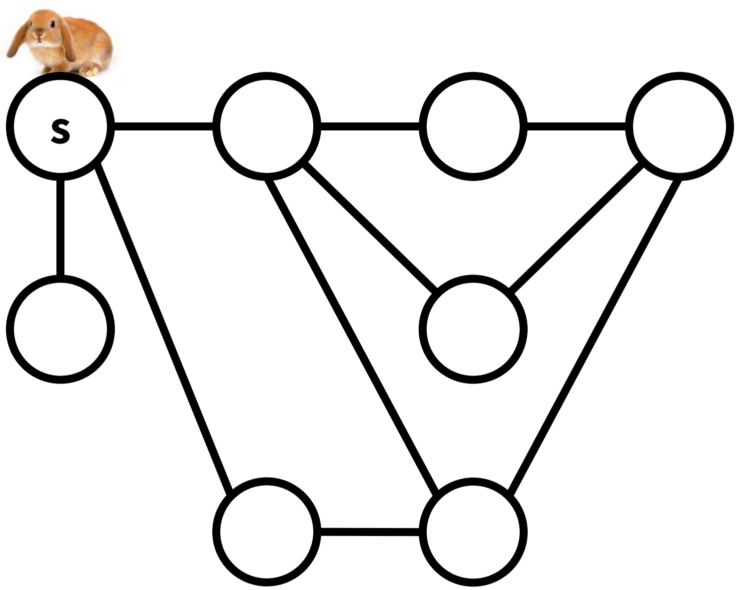
Explain with directed graph; relationship between the two representations; an example in e-commerce

Depth-First Search

Depth-First Search

An analogy

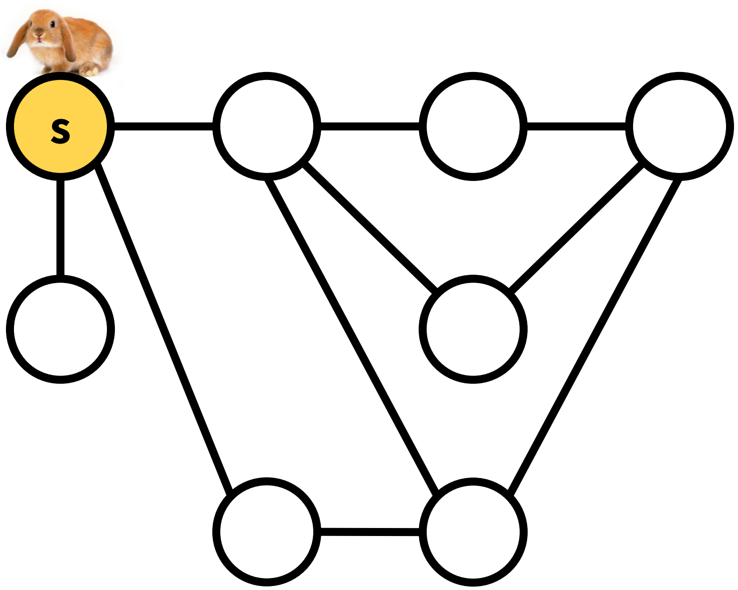
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

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



unvisited

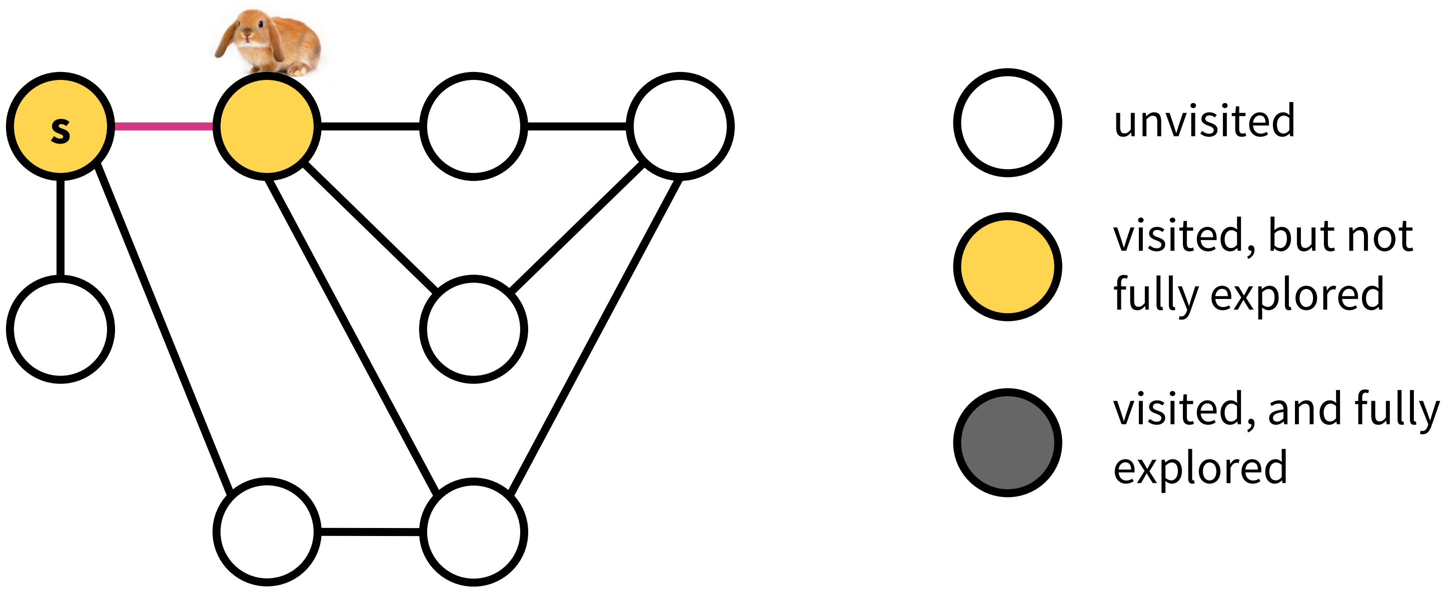
visited, but not fully explored

visited, and fully explored

Depth-First Search

An analogy

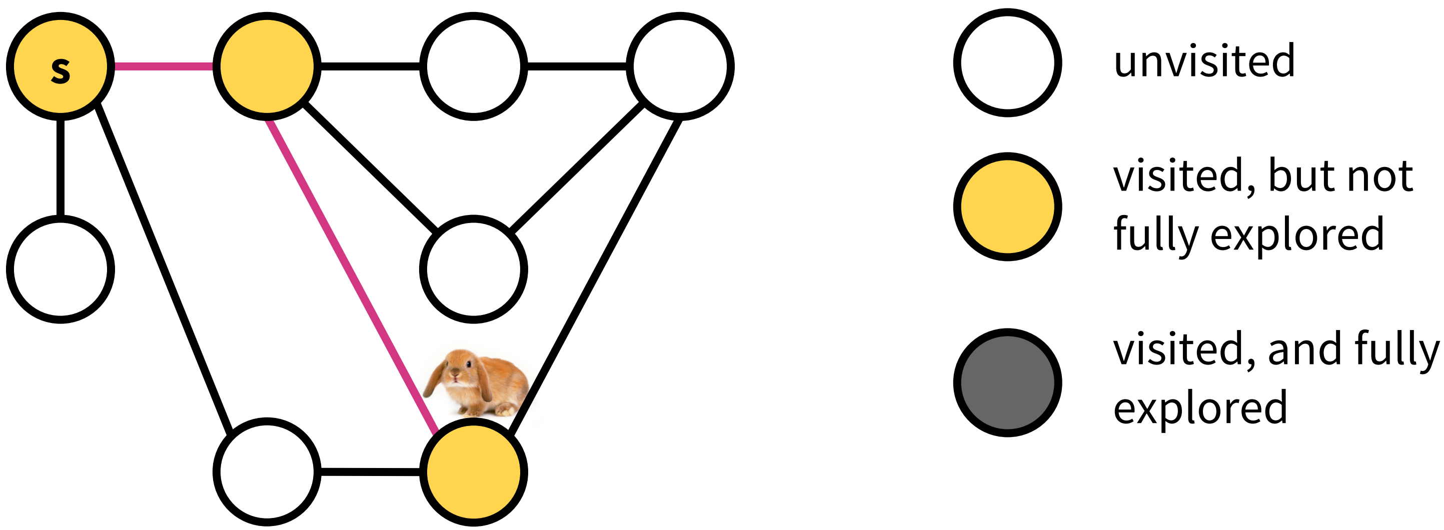
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

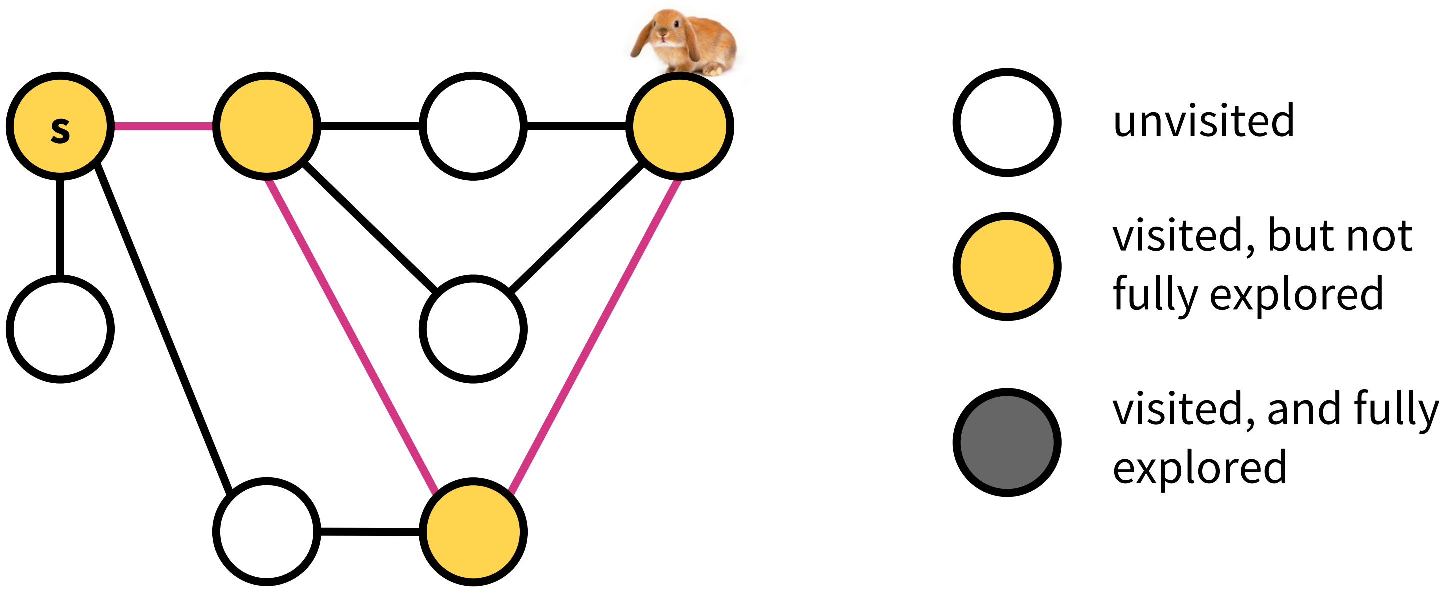
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

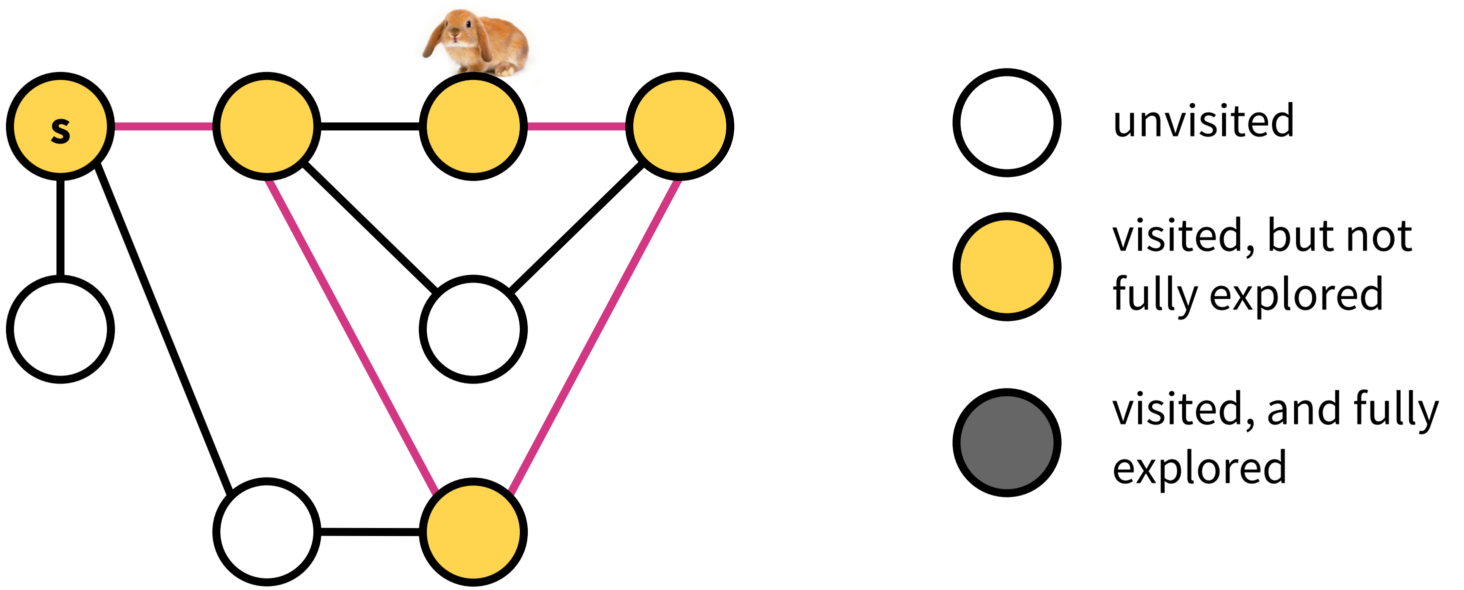
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

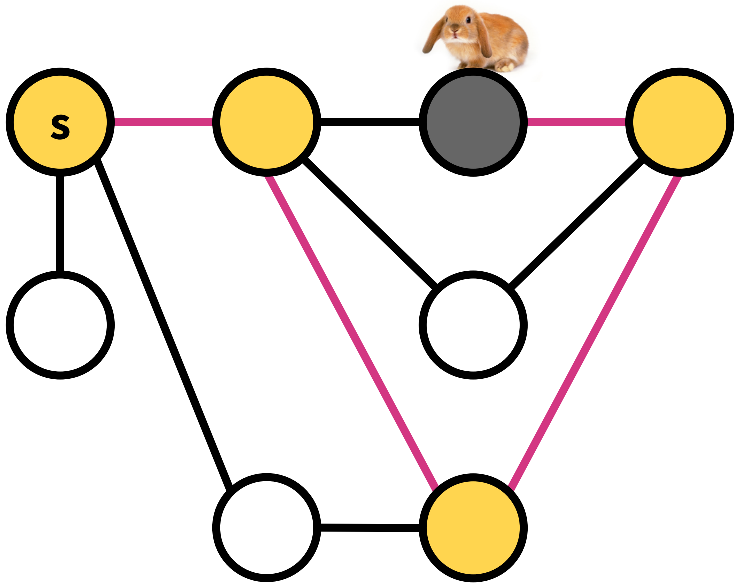
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

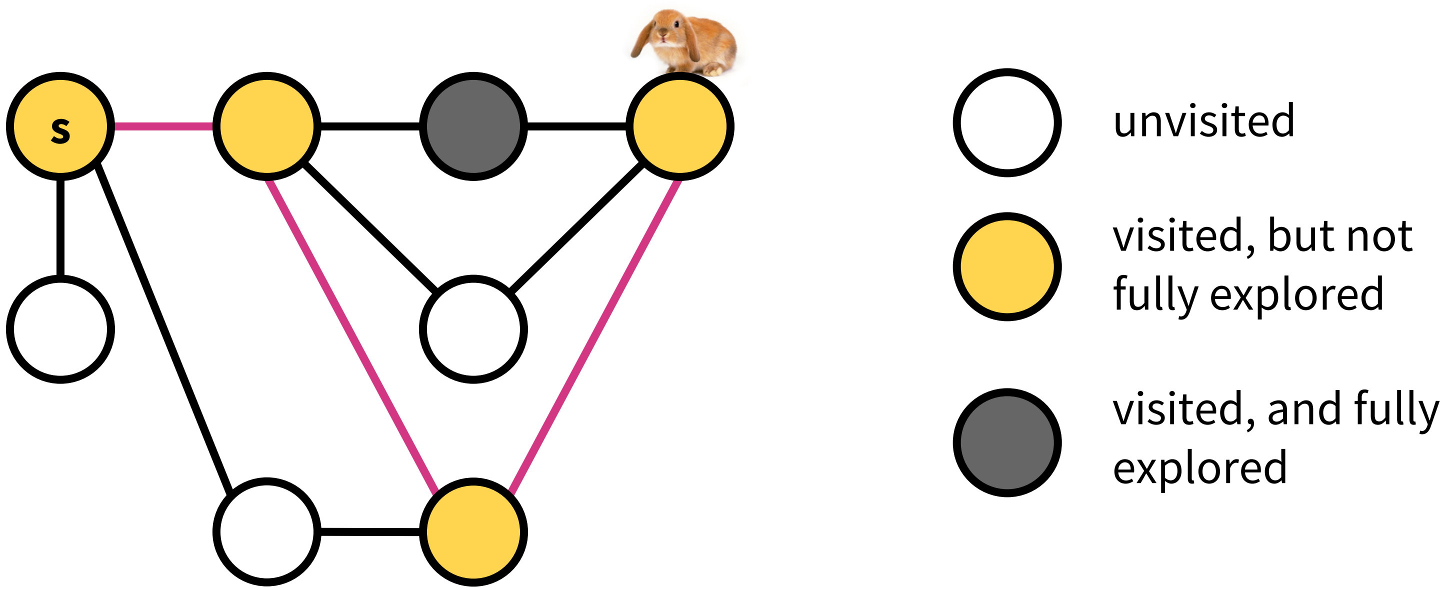
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

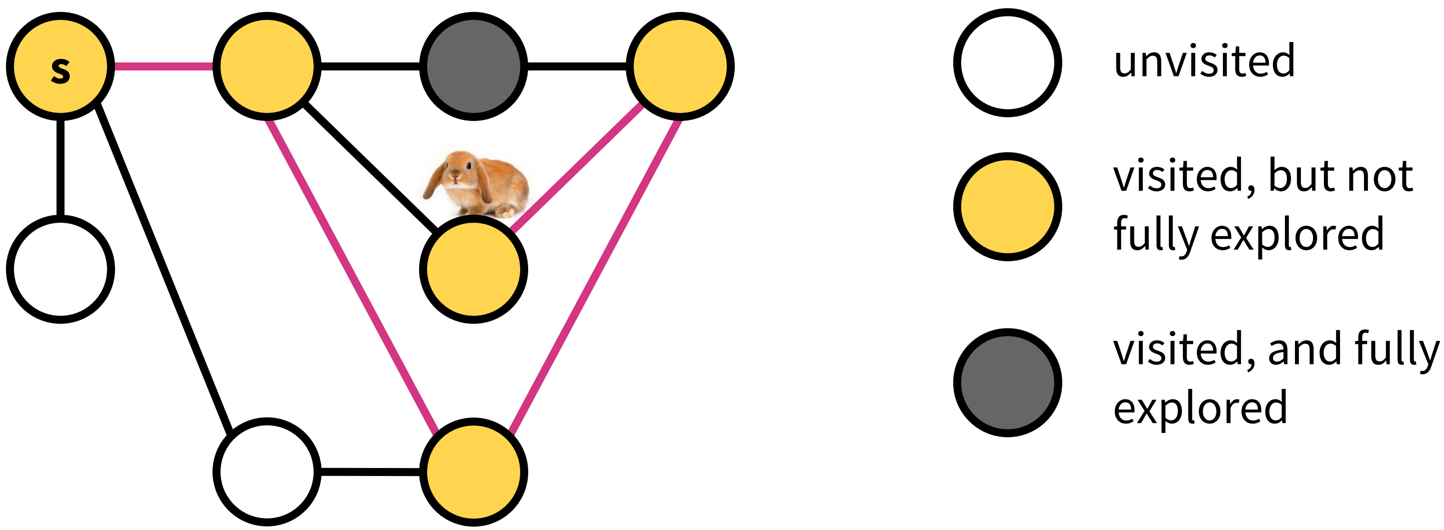
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

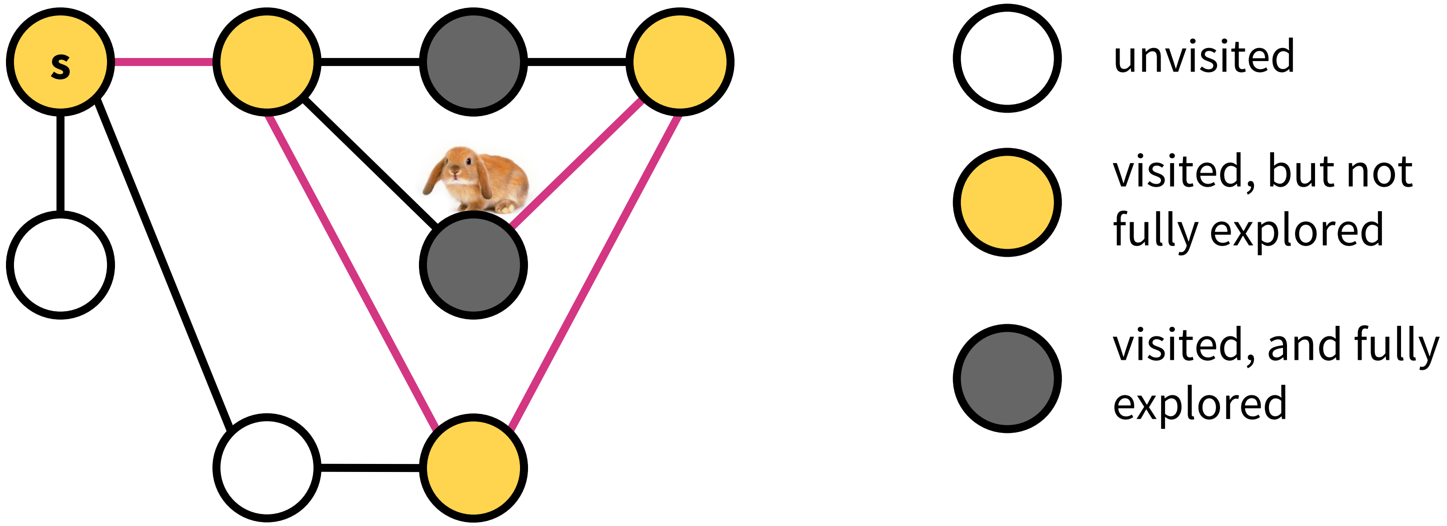
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

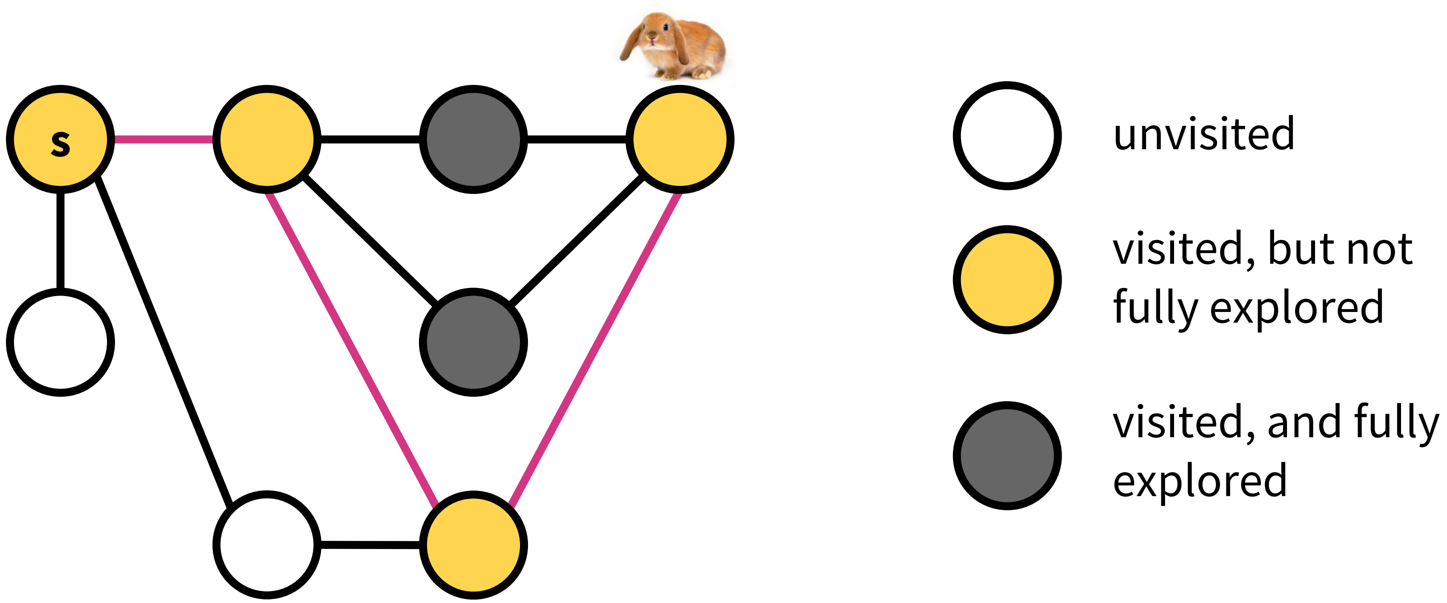
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

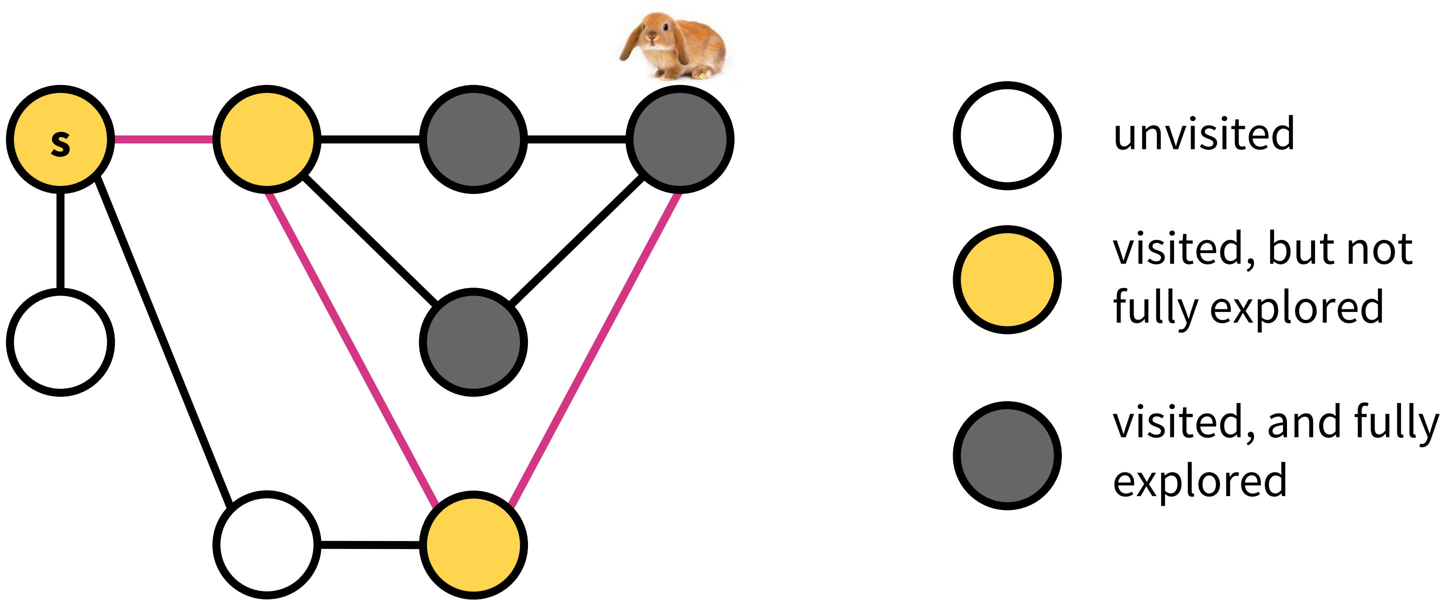
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

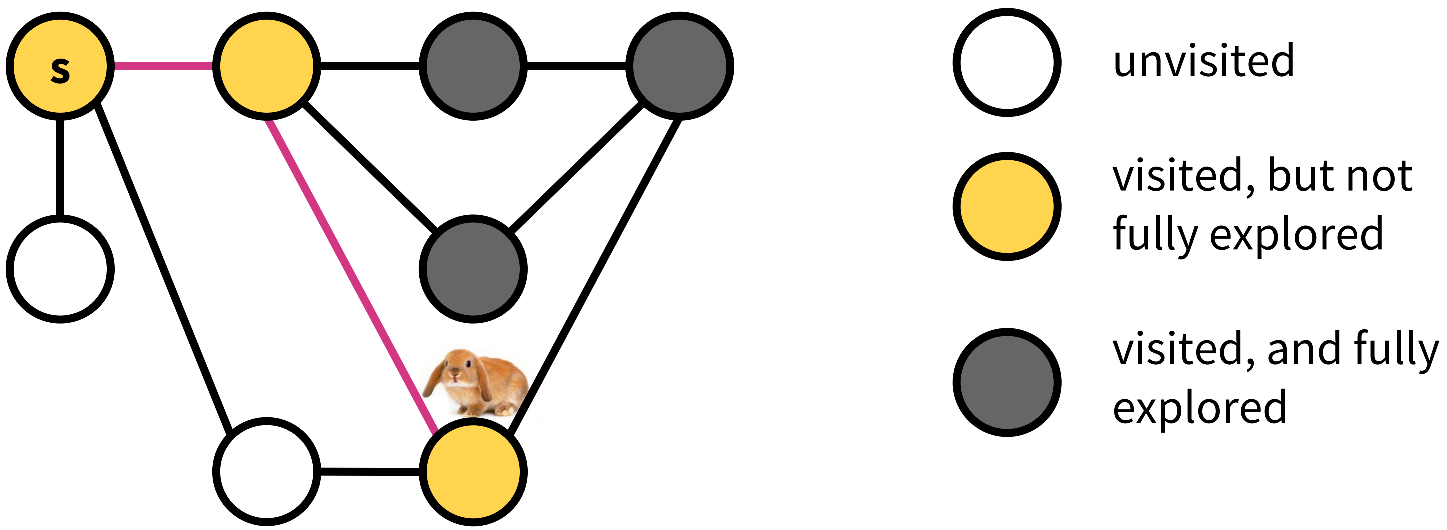
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

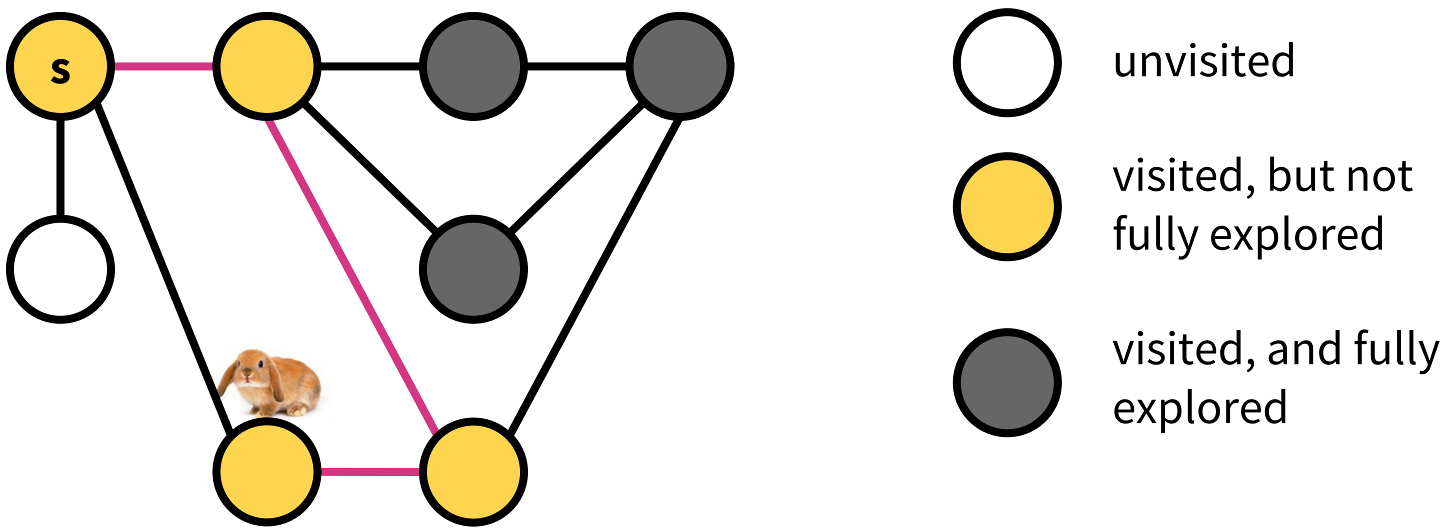
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

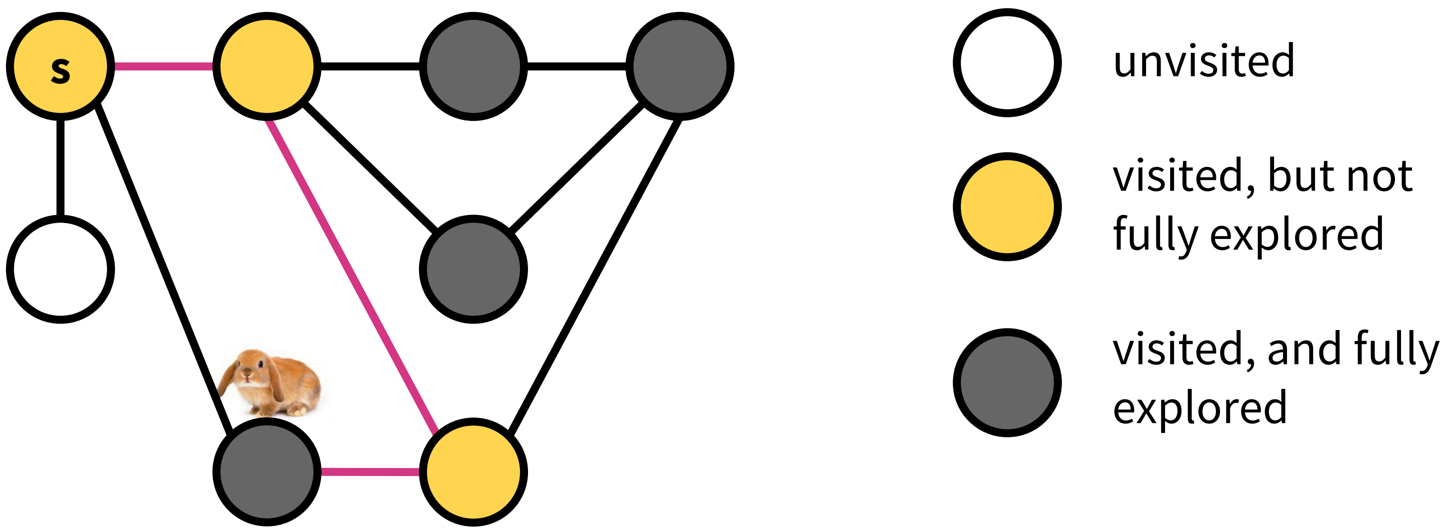
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

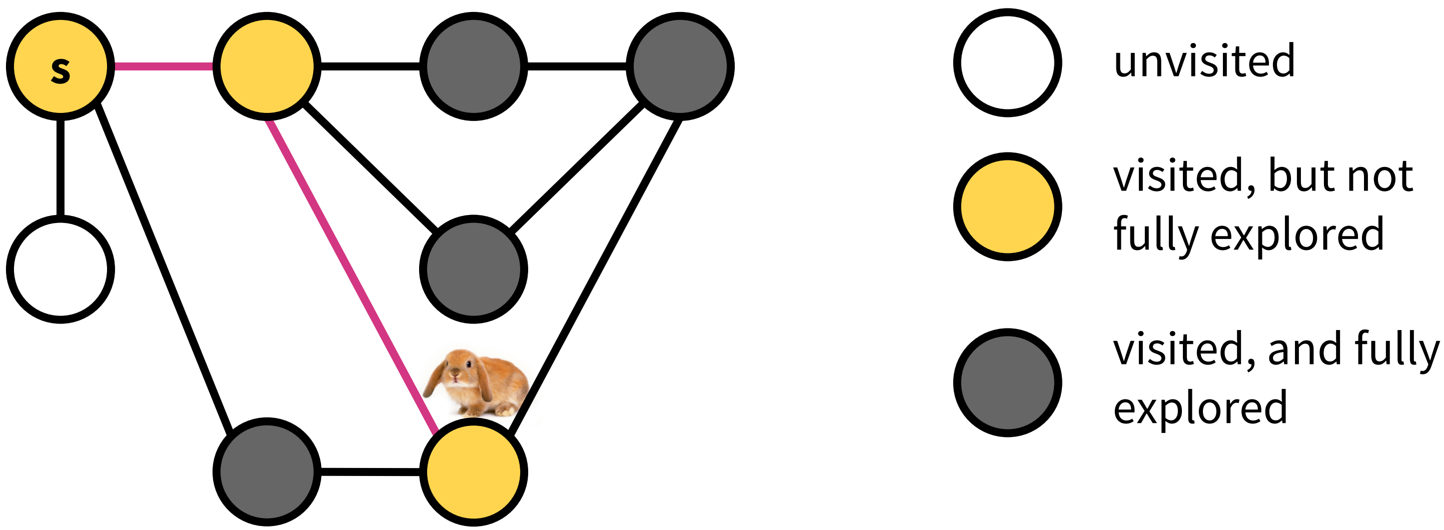
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

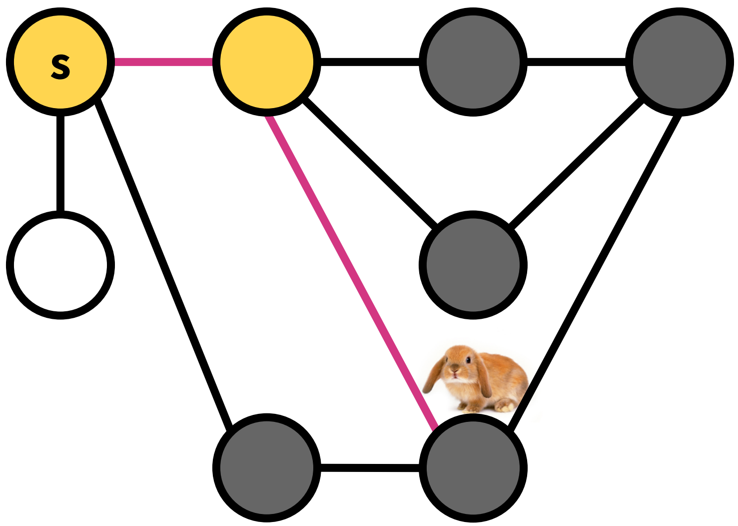
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

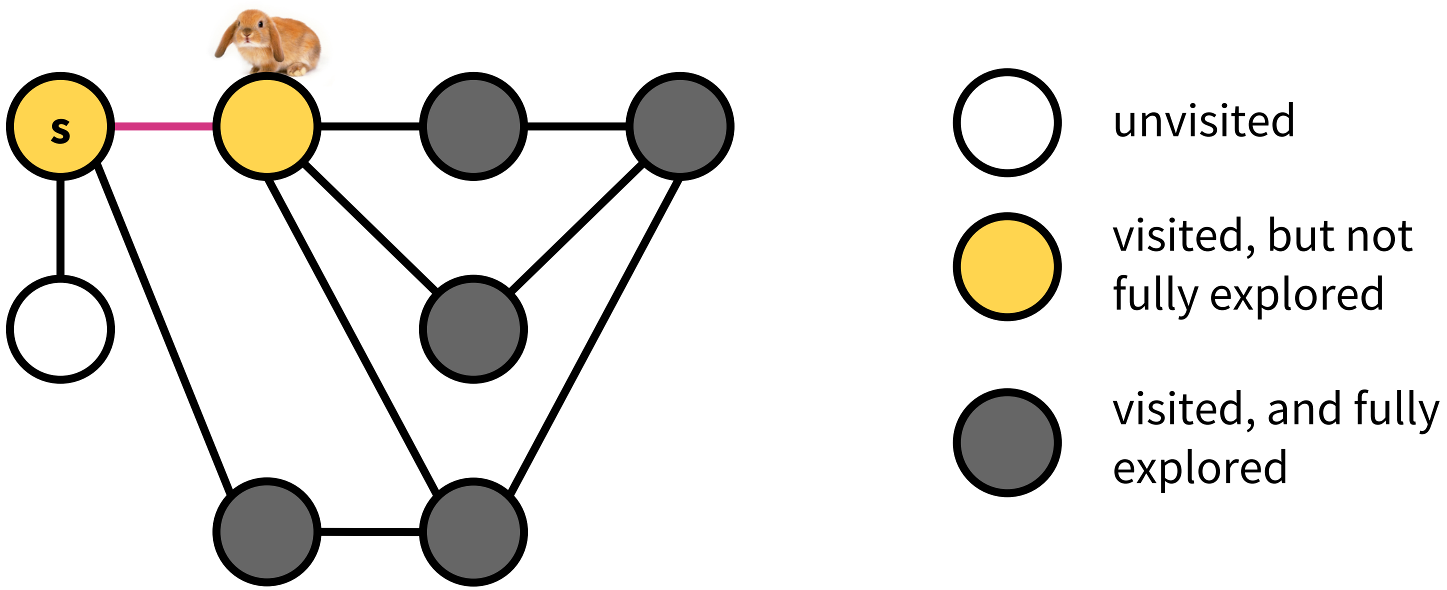
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

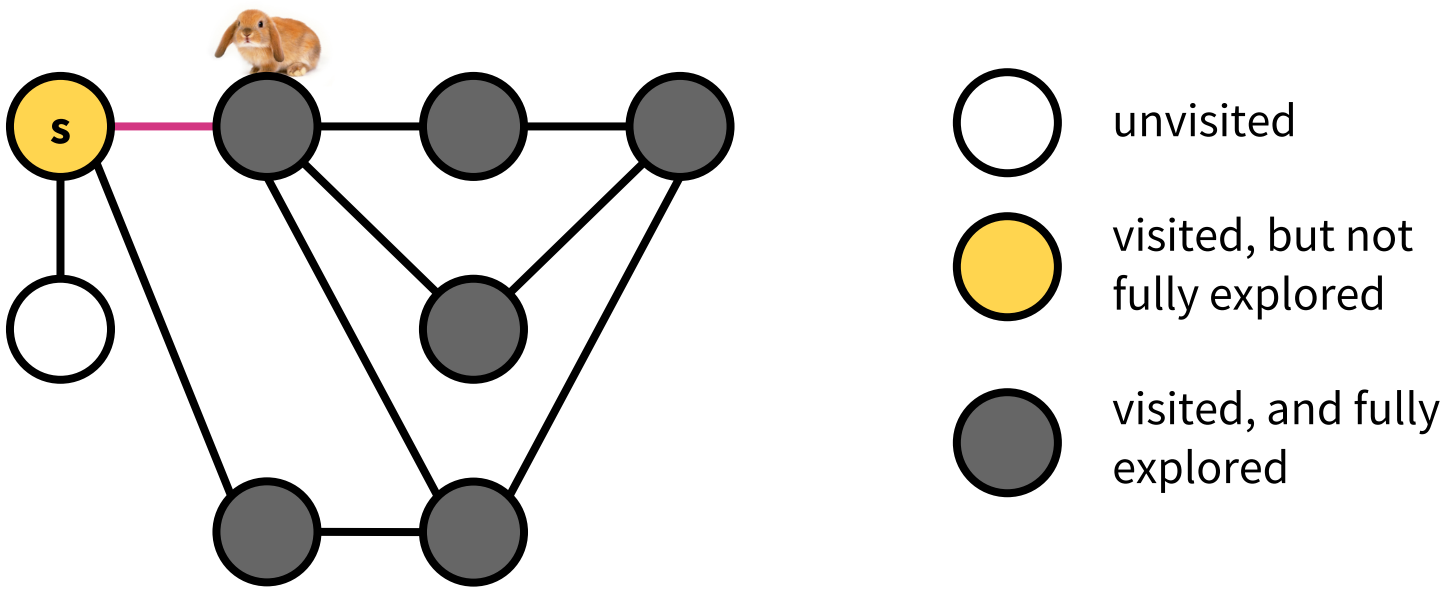
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

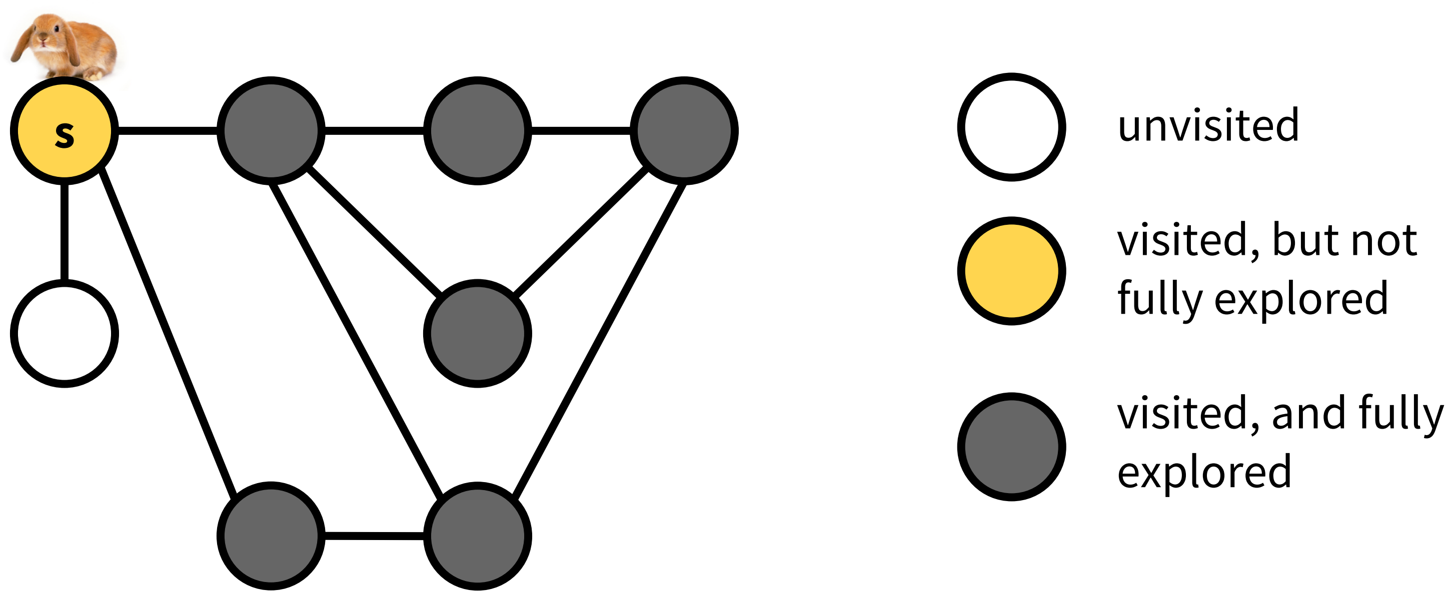
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

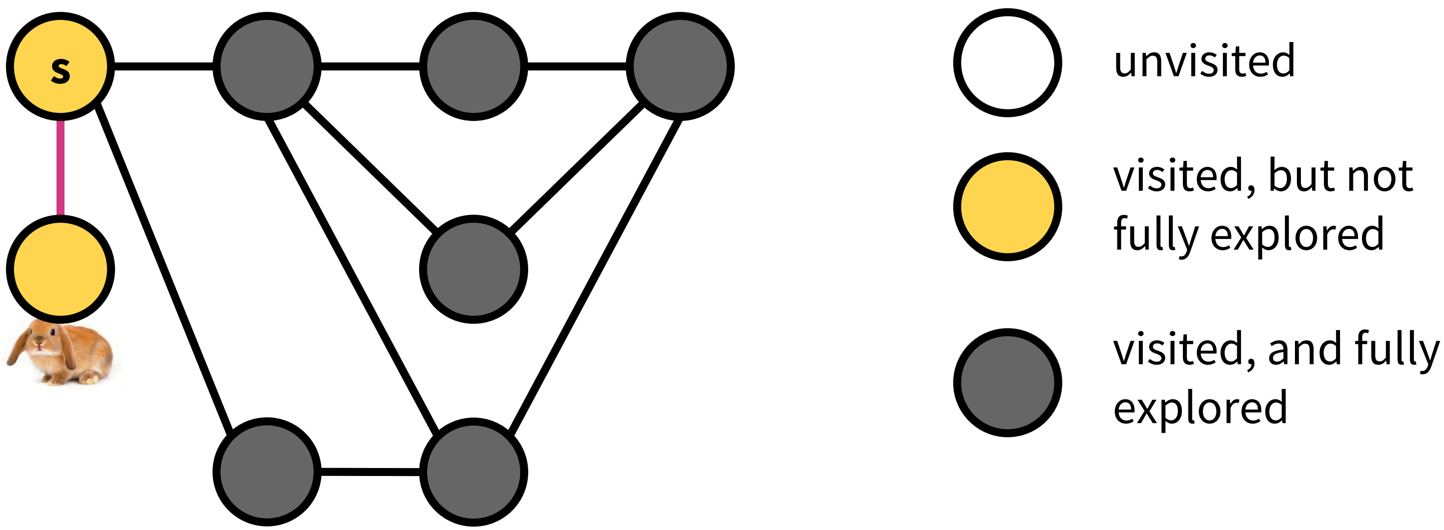
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

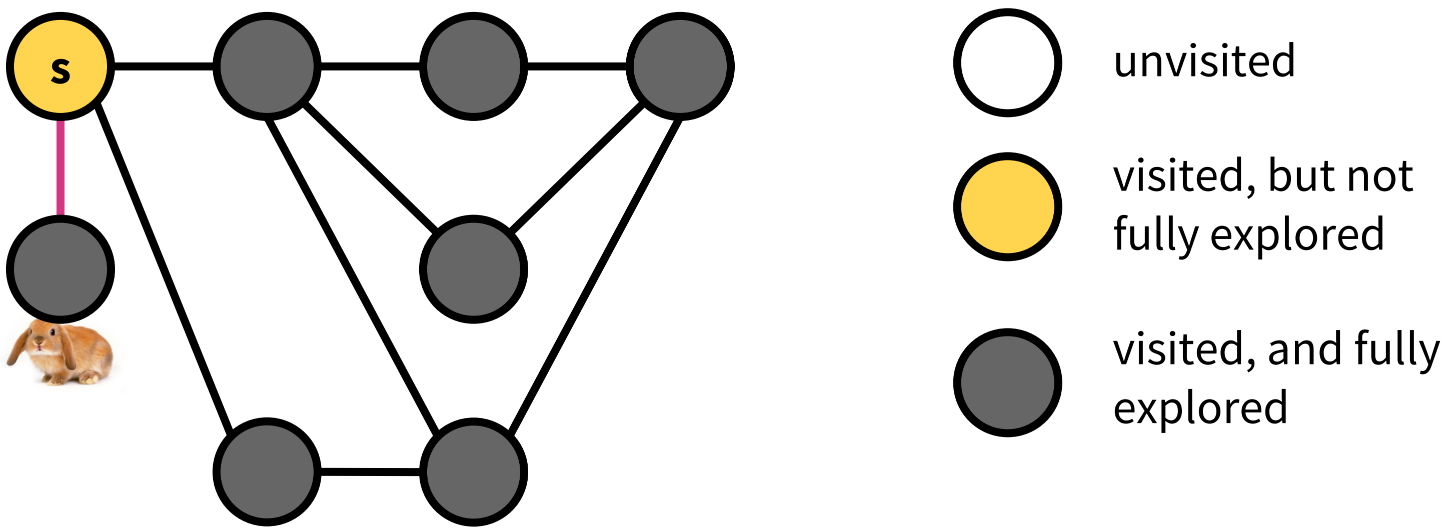
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

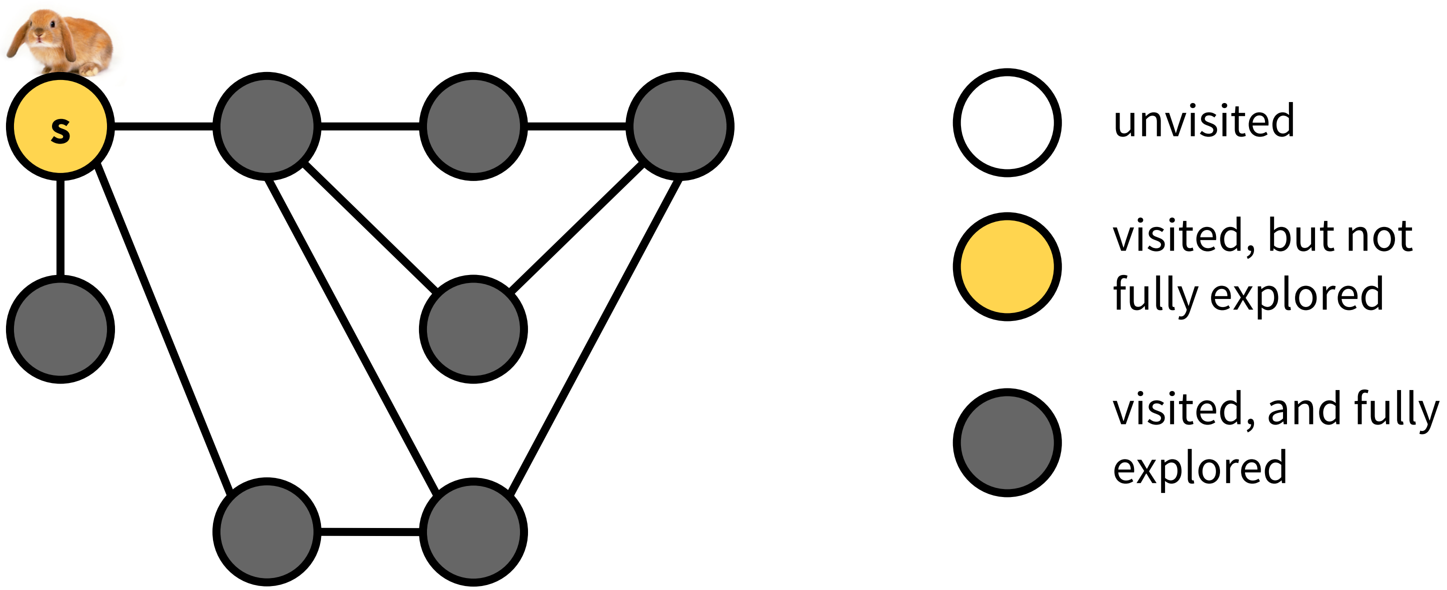
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

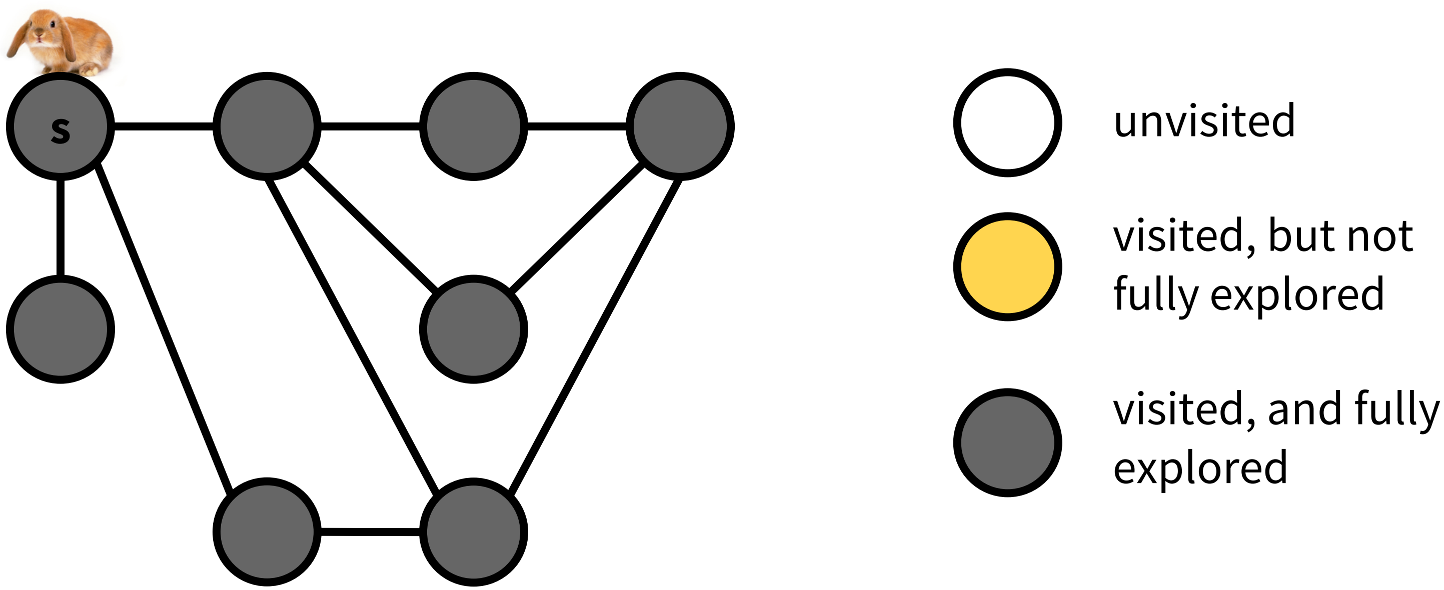
A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



Depth-First Search

An analogy

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

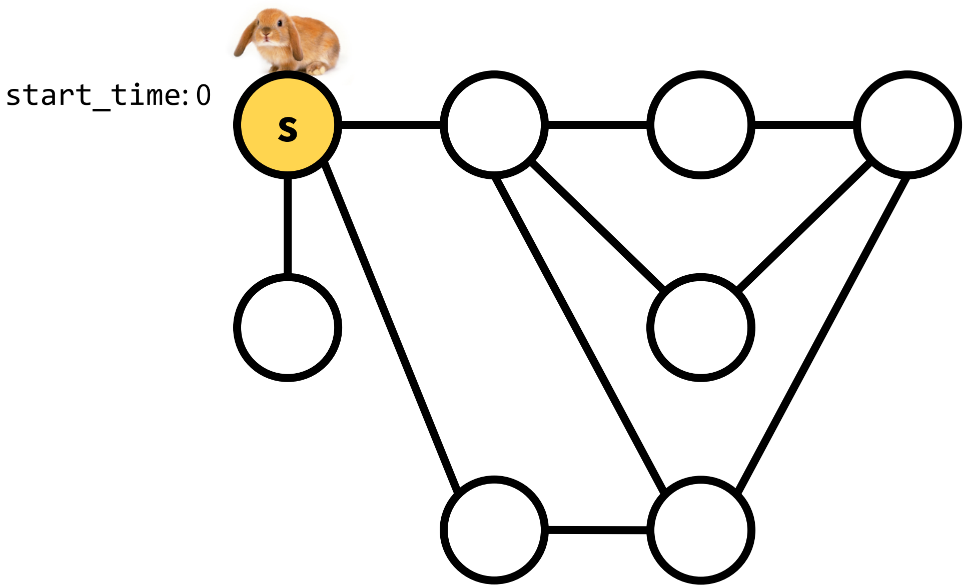


Depth-First Search

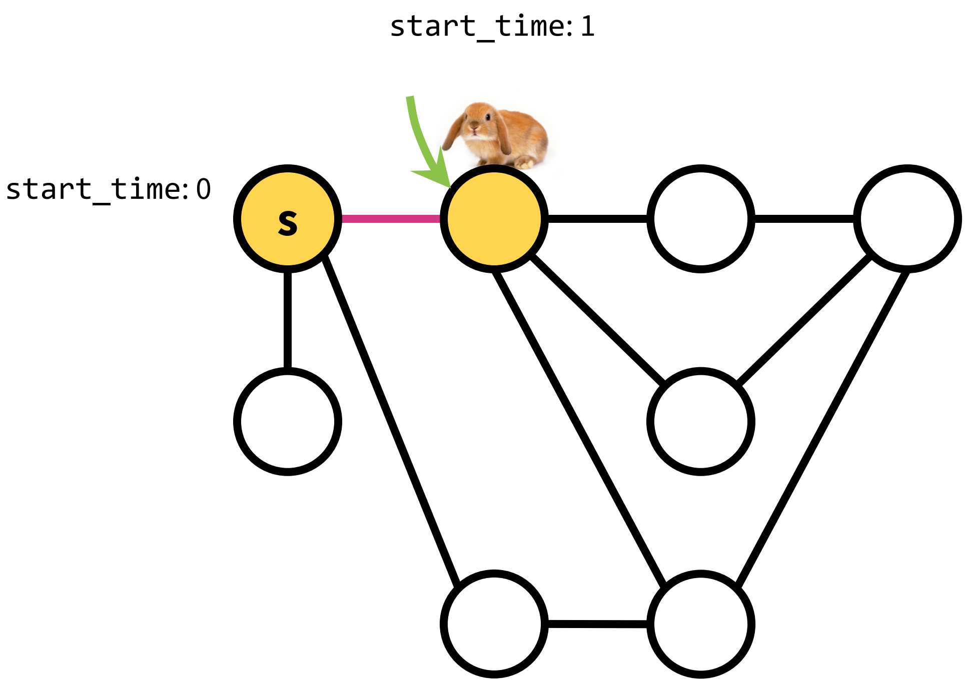
```
algorithm dfs(u, cur_time):
  u.start_time = cur_time
  cur_time += 1
  u.status = "in_progress"
  for v in u.neighbors:
    if v.status is "unvisited":
        cur_time = dfs(v, cur_time)
        cur_time += 1
  u.end_time = cur_time
  u.status = "done"
  return cur_time
```

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

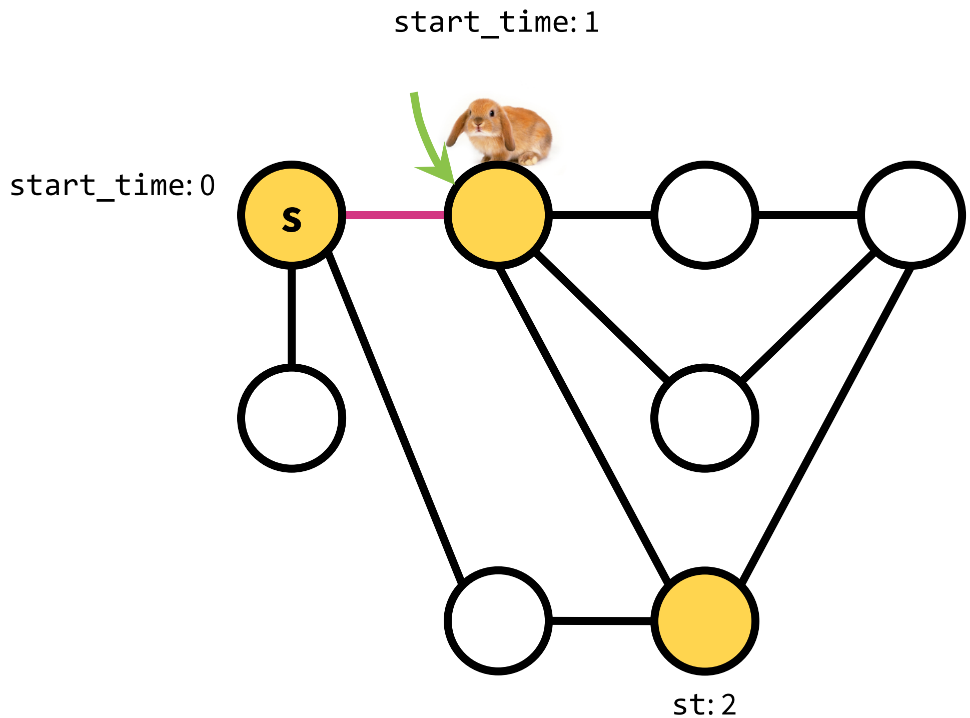
Depth-First Search



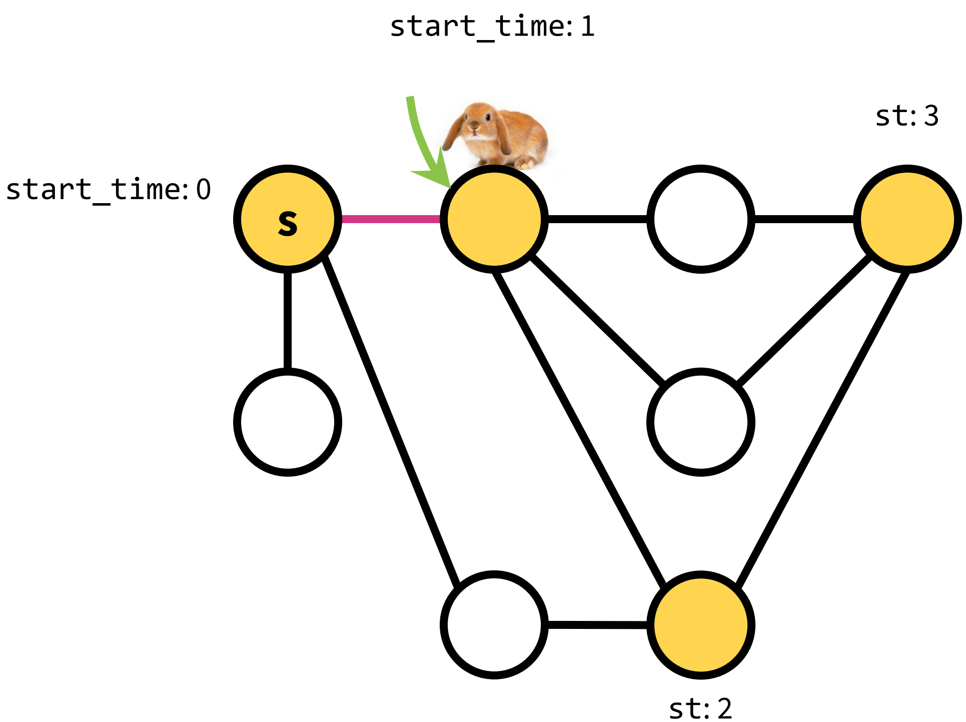
Depth-First Search



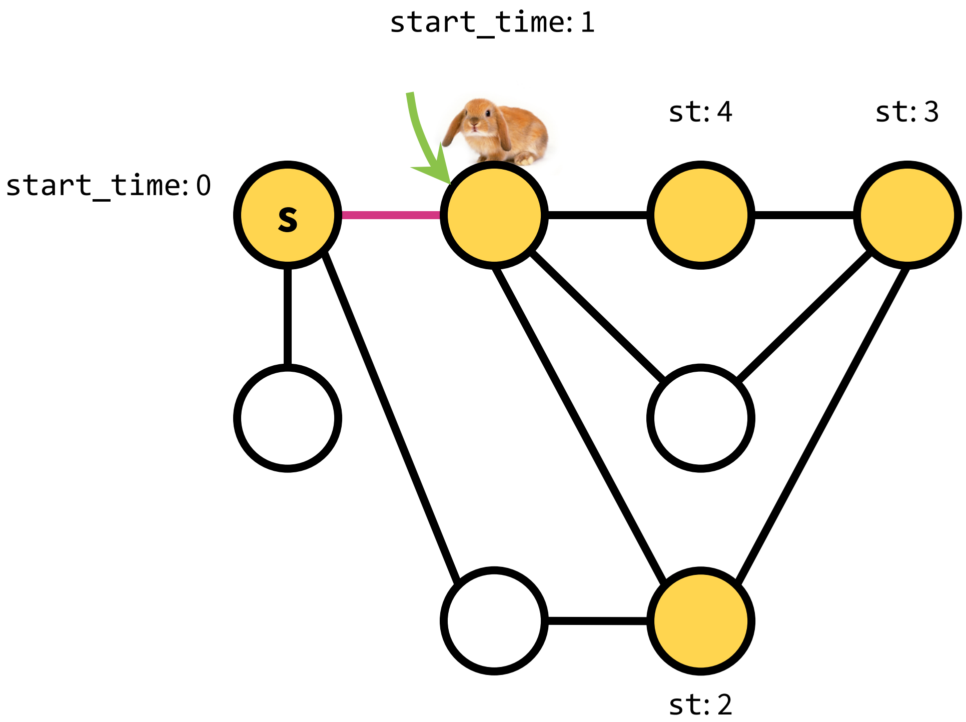
Depth-First Search



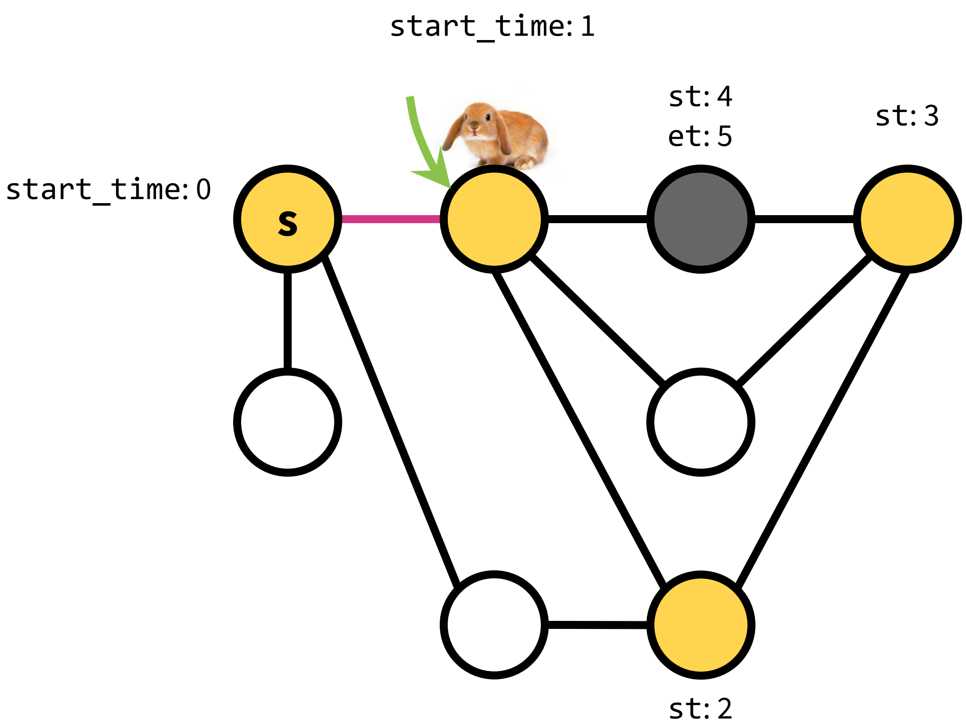
Depth-First Search



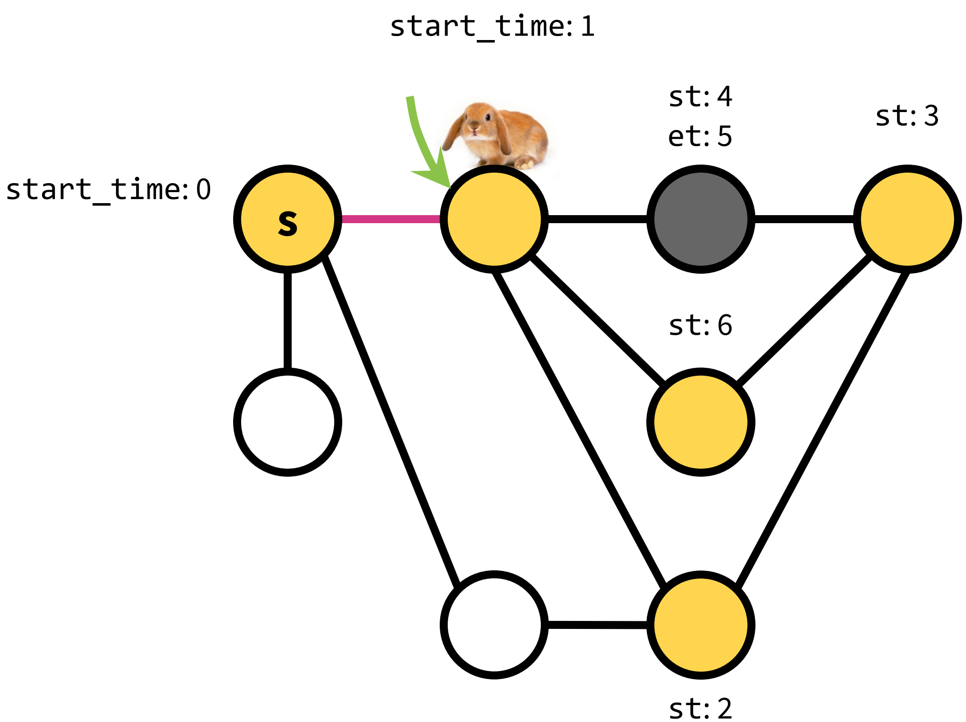
Depth-First Search



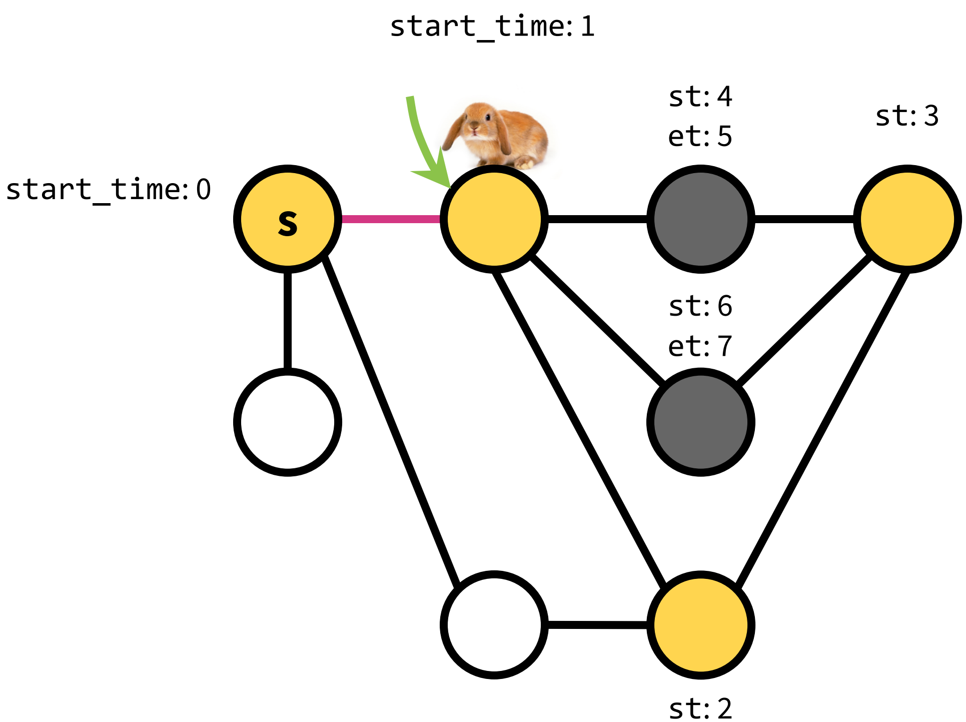
Depth-First Search



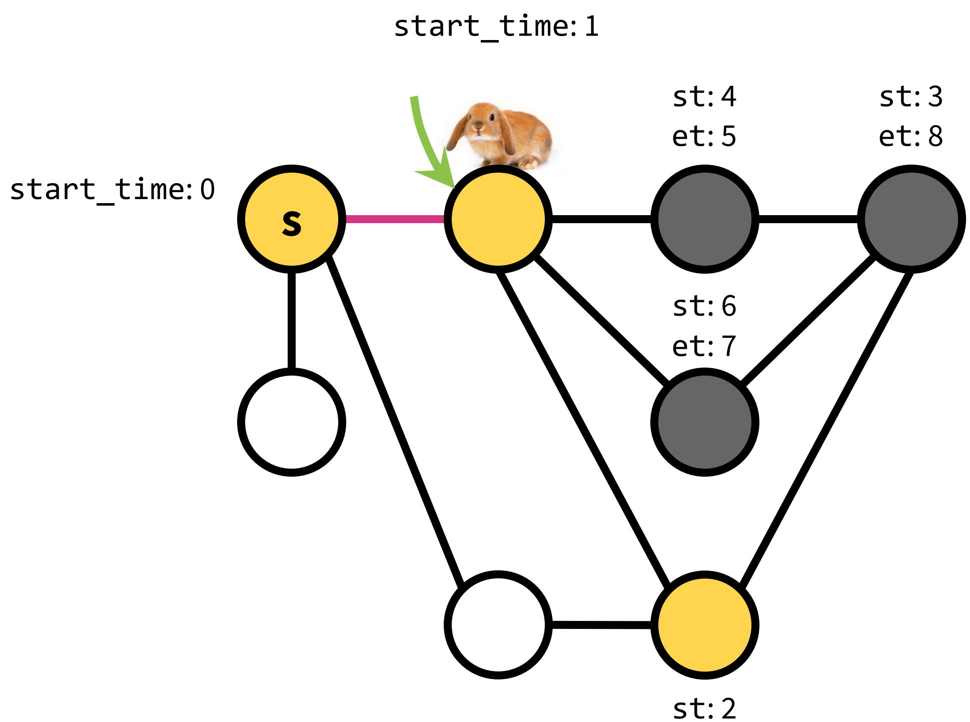
Depth-First Search



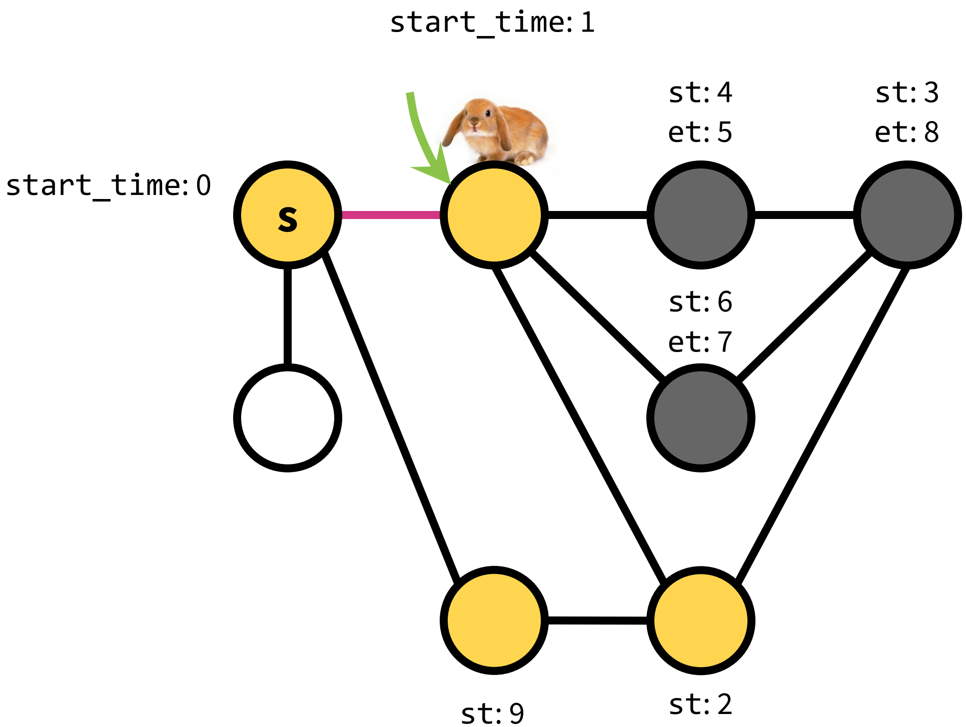
Depth-First Search



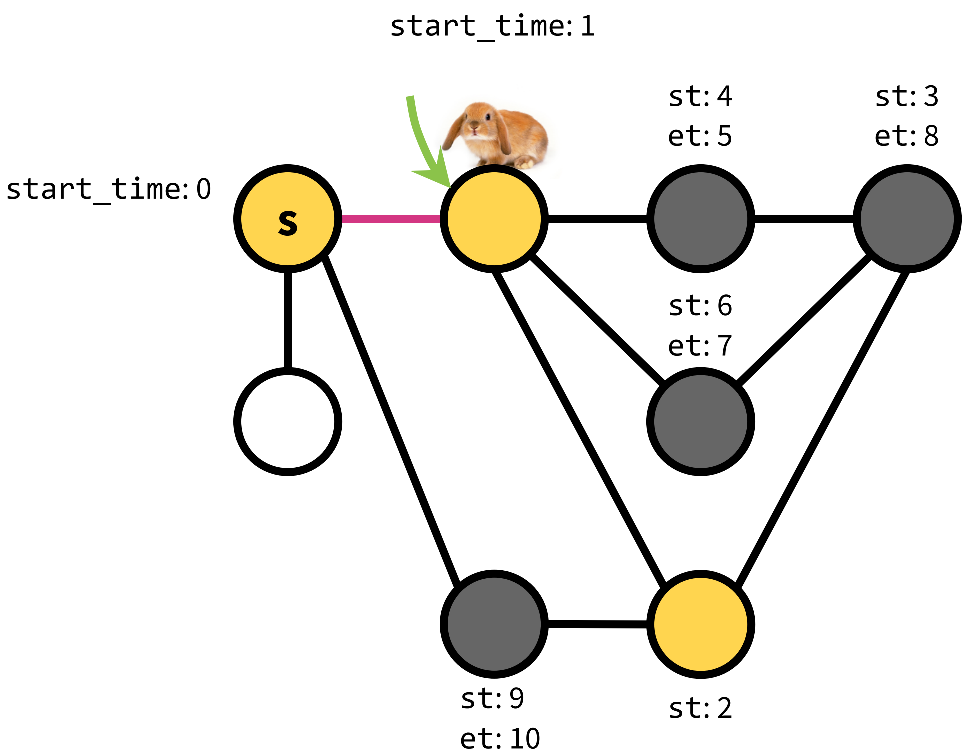
Depth-First Search



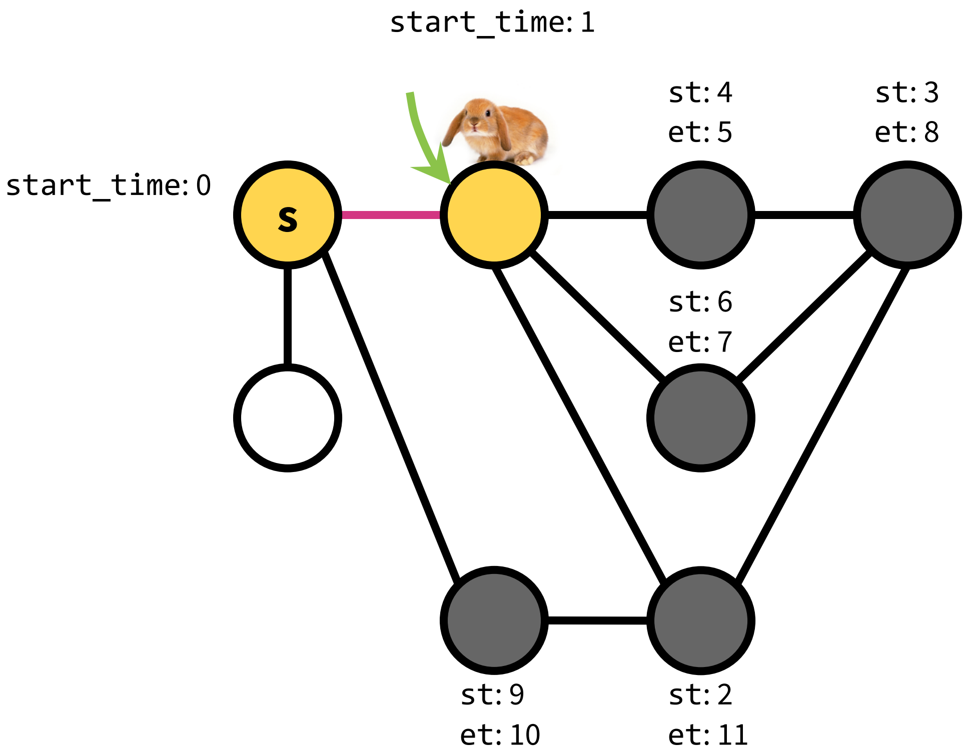
Depth-First Search



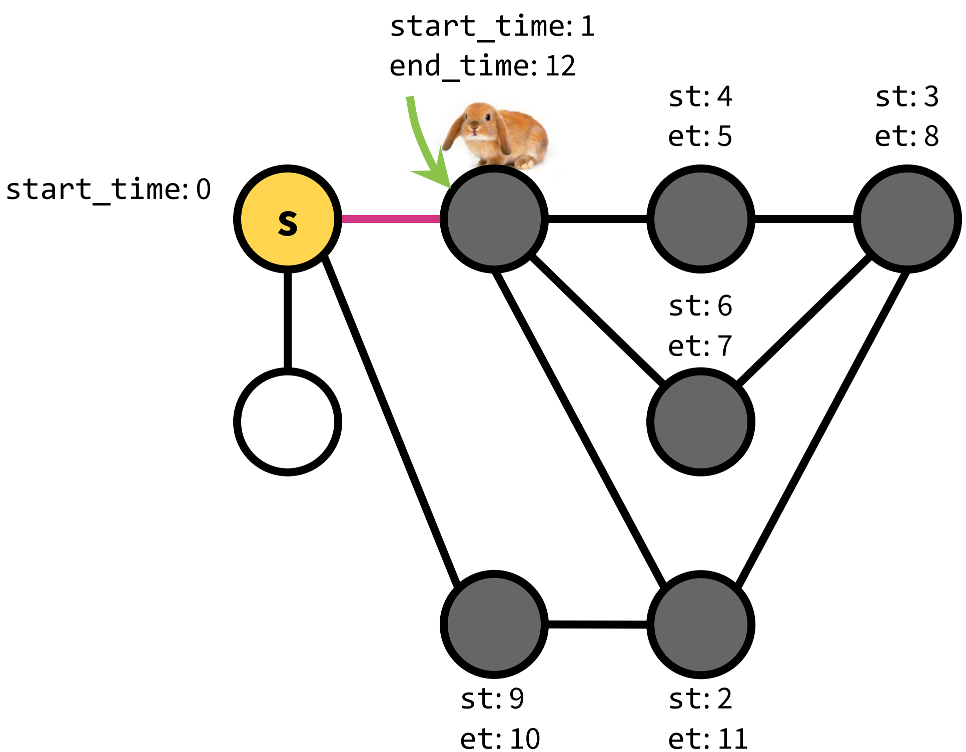
Depth-First Search



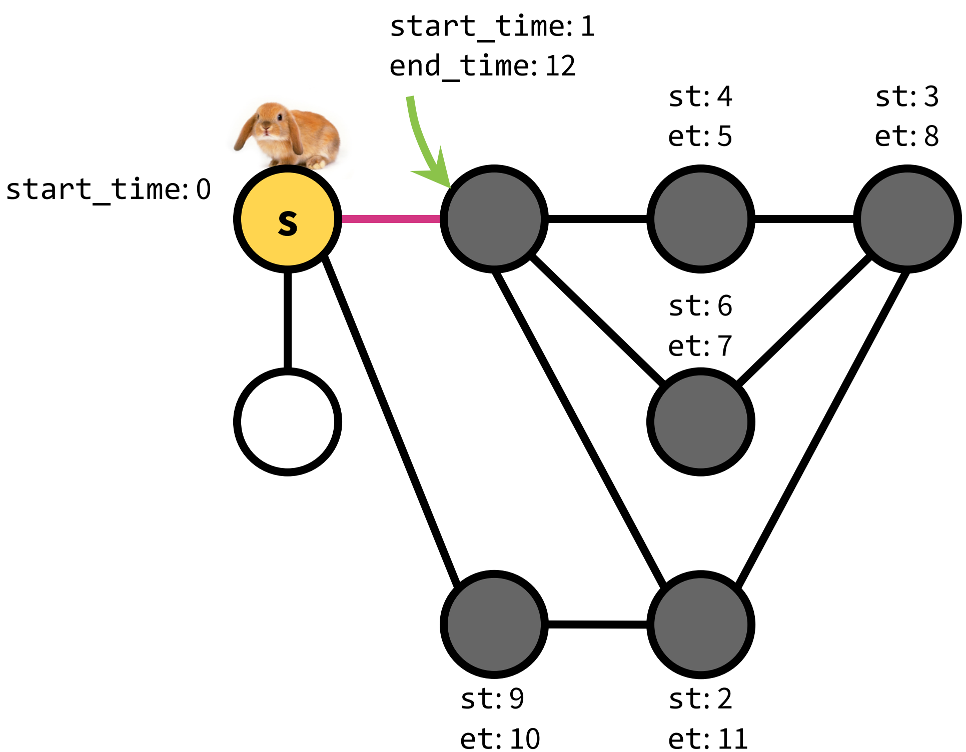
Depth-First Search



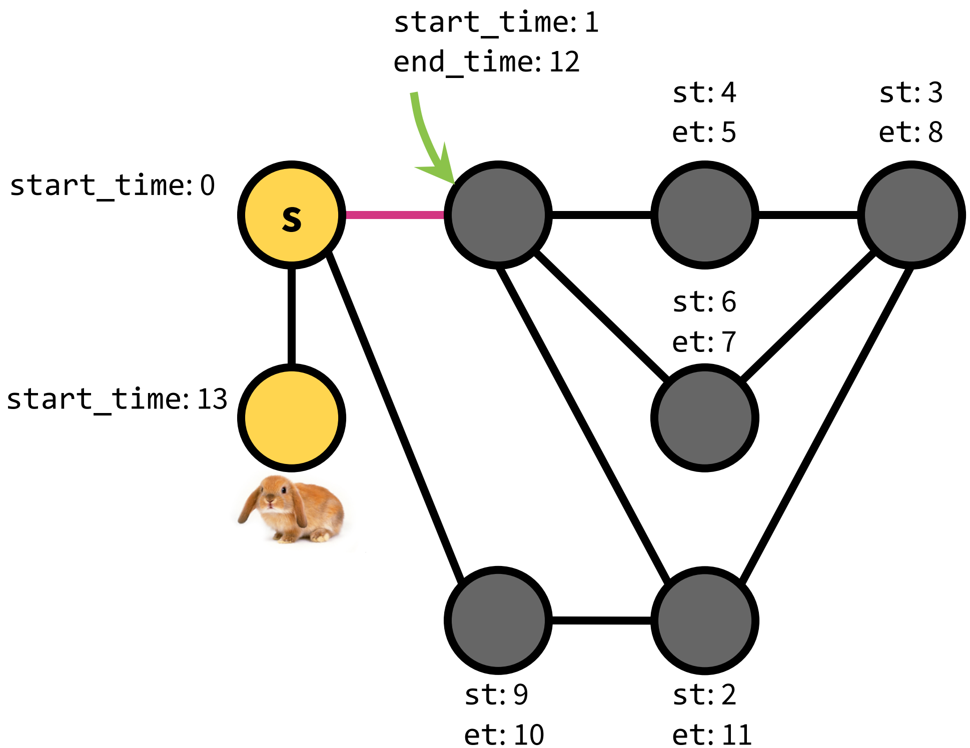
Depth-First Search



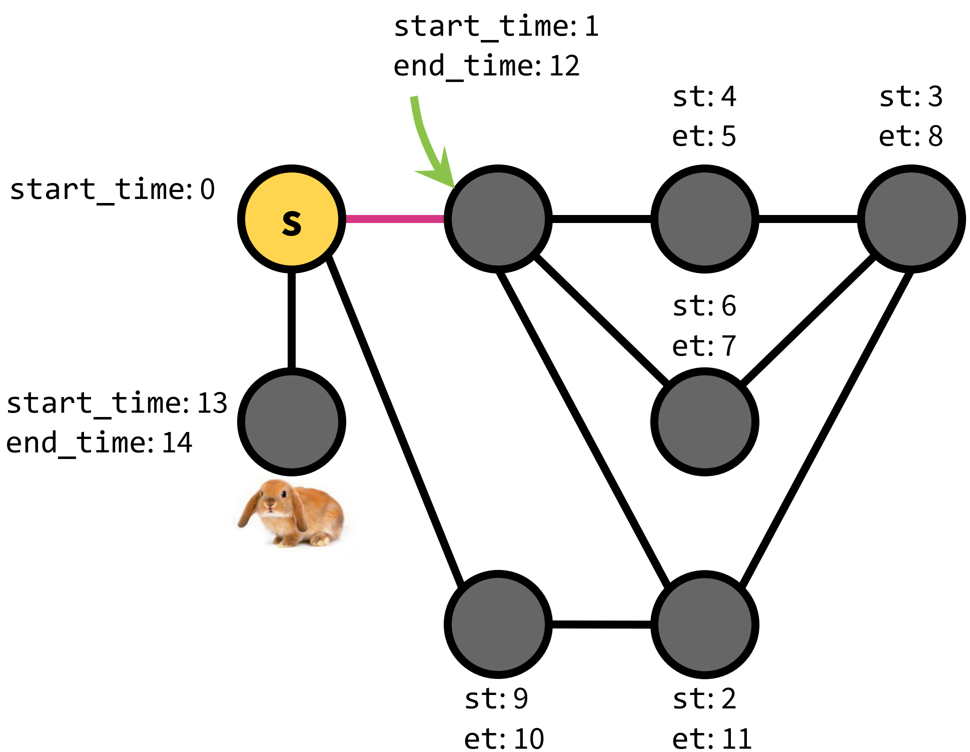
Depth-First Search



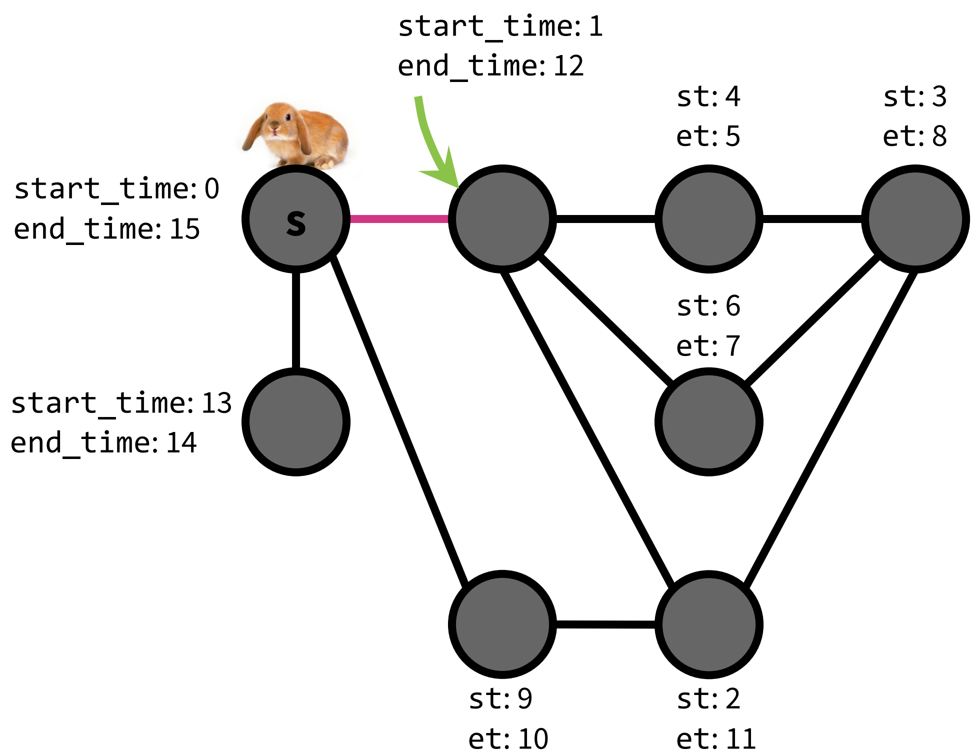
Depth-First Search



Depth-First Search



Depth-First Search



Depth-First Search

Another implementation by using a stack:

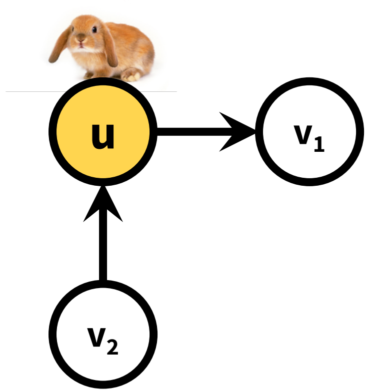
```
algorithm dfs-iterative(G, u):
  let S be a stack
  S.push(u)
  while S is not empty
    u = S.pop()
    if u is not labeled as visited:
       label u as visited
       for all neighbors v:
          S.push(v)
```

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.

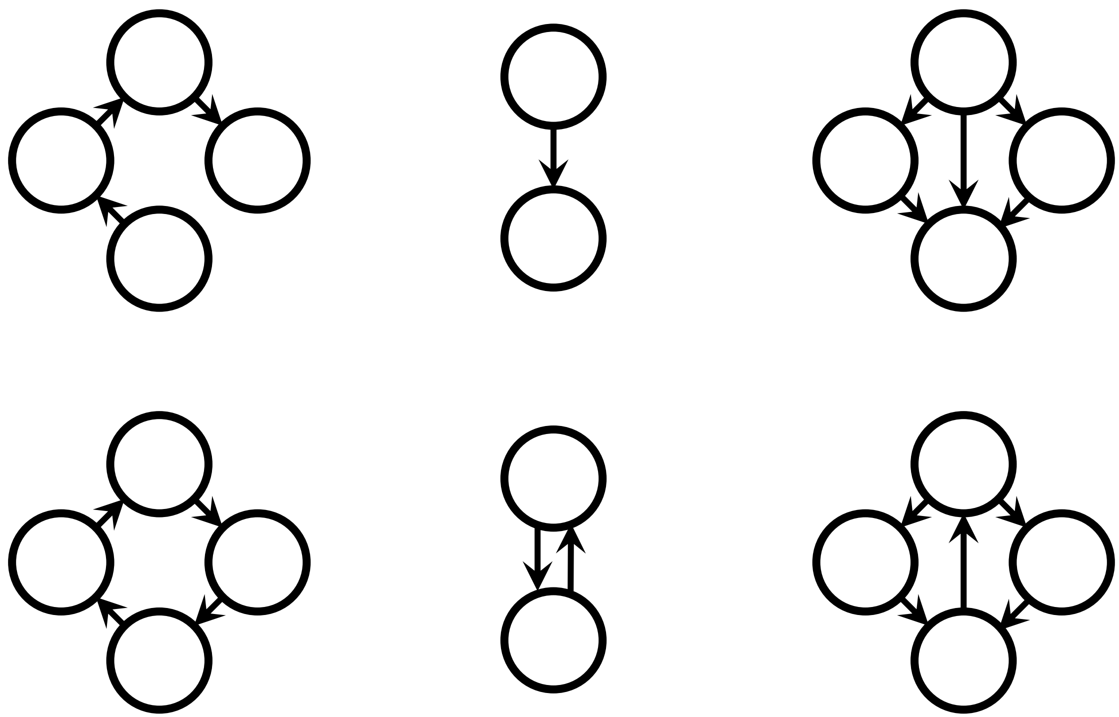


Topological Ordering

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.

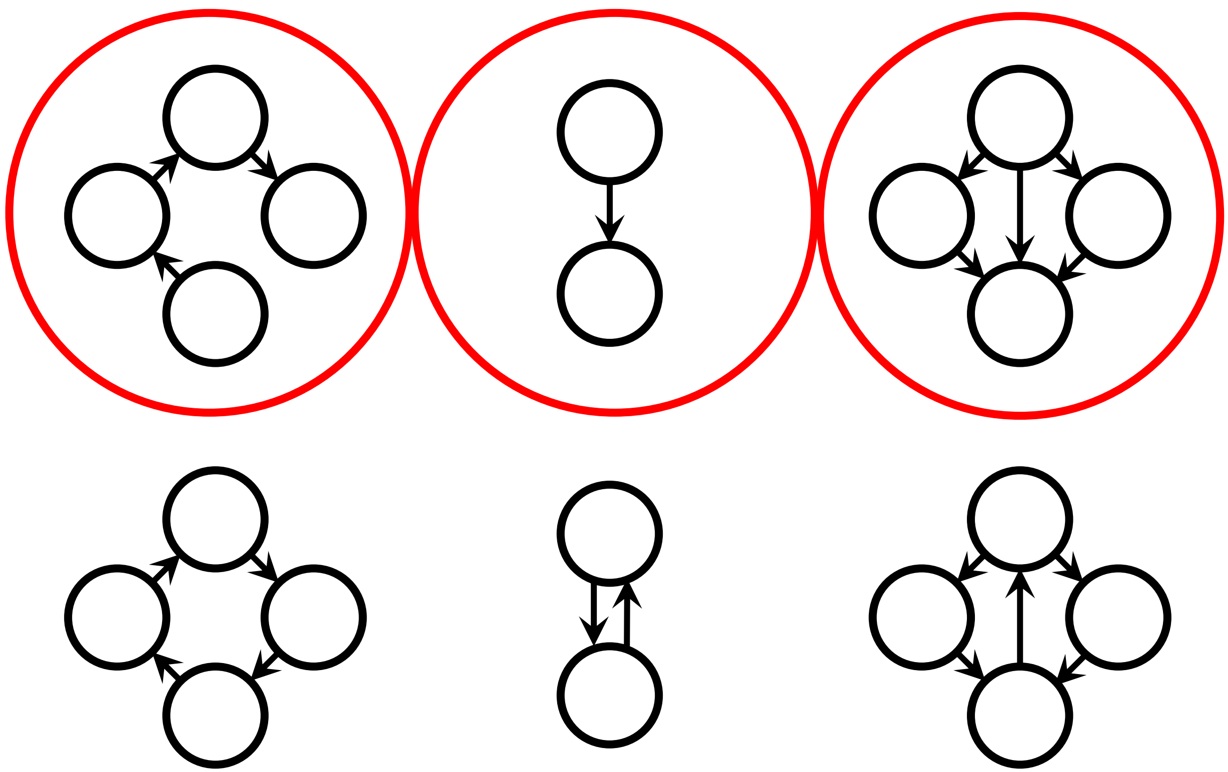
Which of these graphs are valid DAGs? 🤔



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?

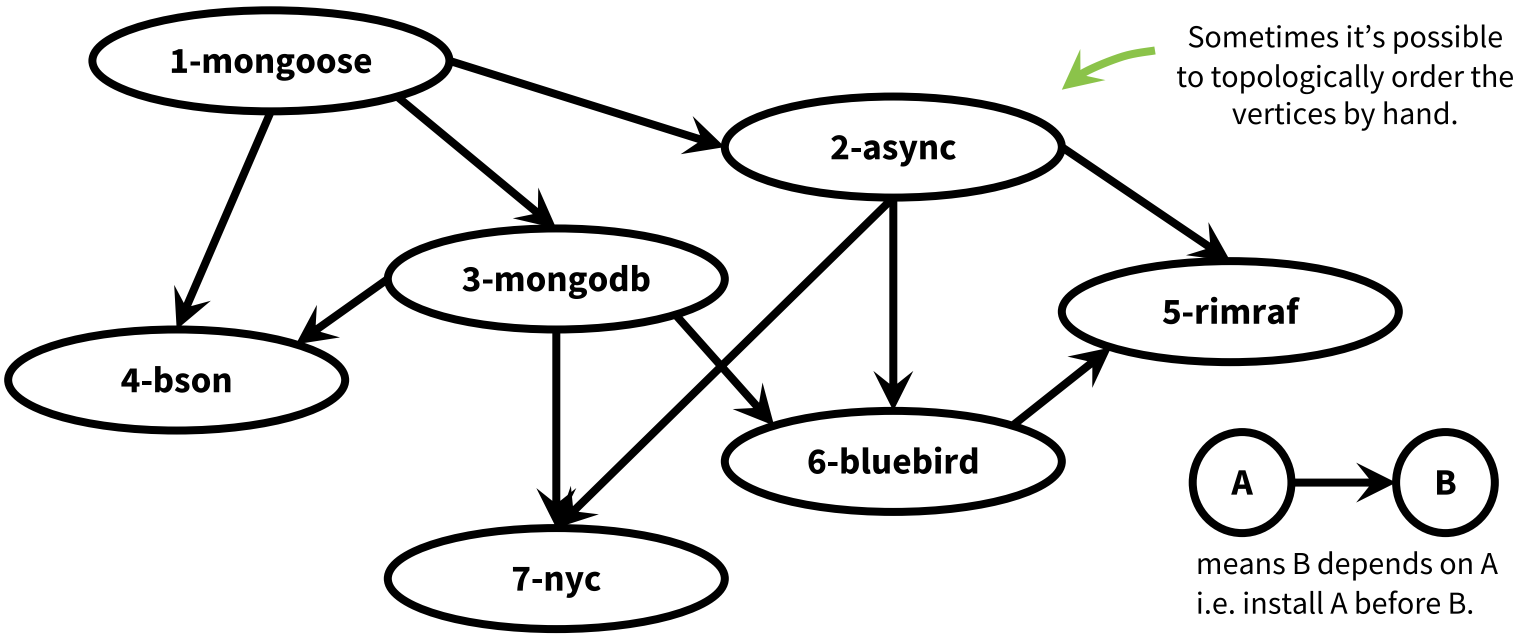
- DFS produces a **topological ordering**, which solves this problem.
- Definition: The topological ordering of a DAG is an ordering of its vertices such that for every directed edge $(u, v) \in E$, u precedes v in the ordering.

Topological Ordering

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

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

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

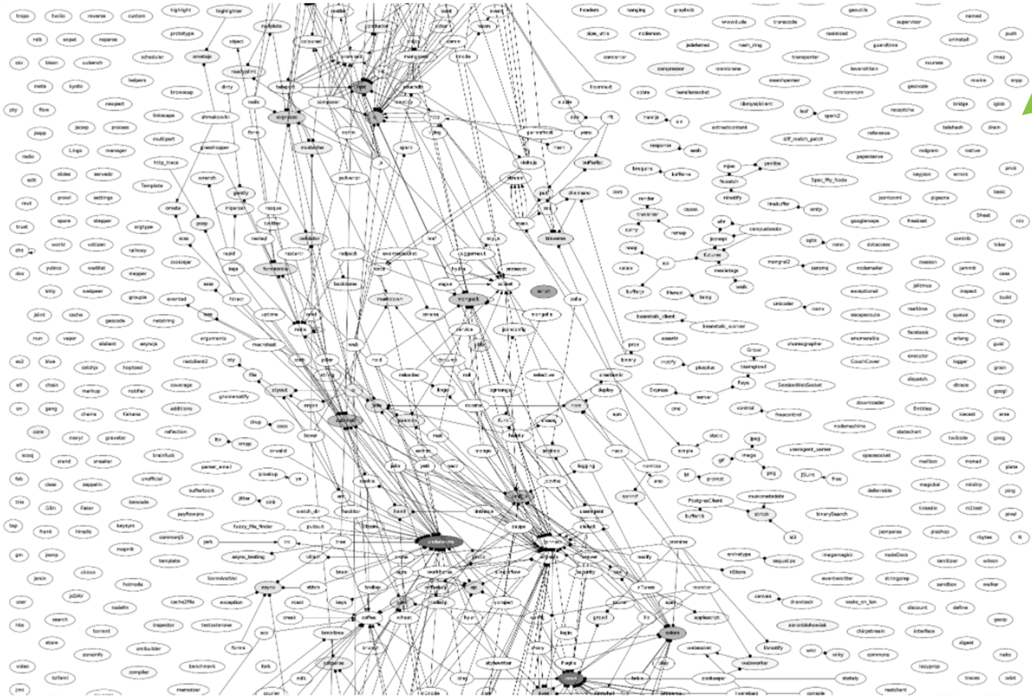


Topological Ordering

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

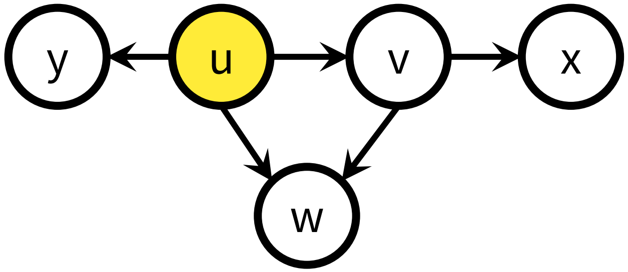
```
reversed_topological_list = []
algorithm dfs(u, cur_time):
  u.start_time = cur_time
  cur_time += 1
  u.status = "in_progress"
  for v in u.neighbors:
    if v.status is "unvisited":
      cur_time = dfs(v, cur_time)
      cur_time += 1
  u.end_time = cur_time
  u.status = "done"
  reversed_topological_list.append(u)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

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

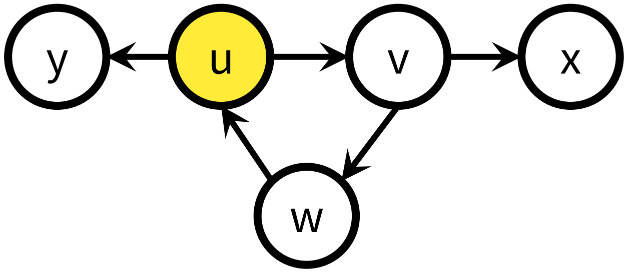
Topological Ordering

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



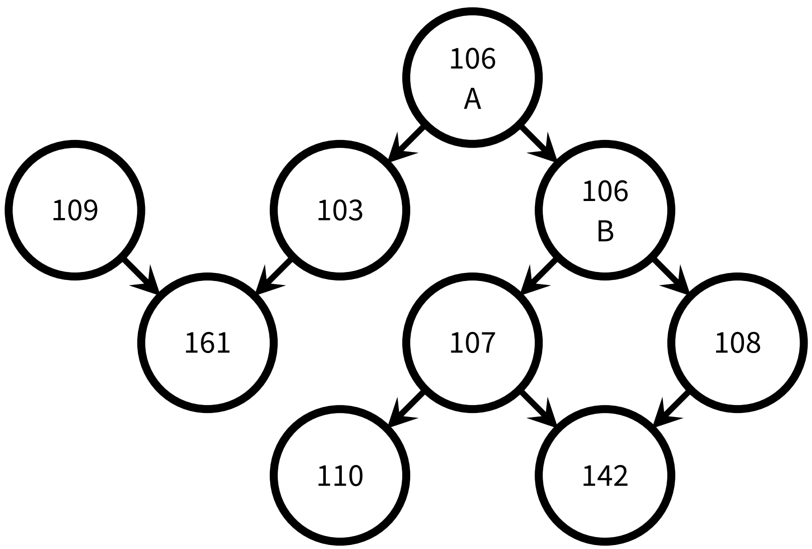
Finishing u requires finishing v thus w;
But finishing w requires finishing u (an in-progress node);
Which is a deadlock

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`.

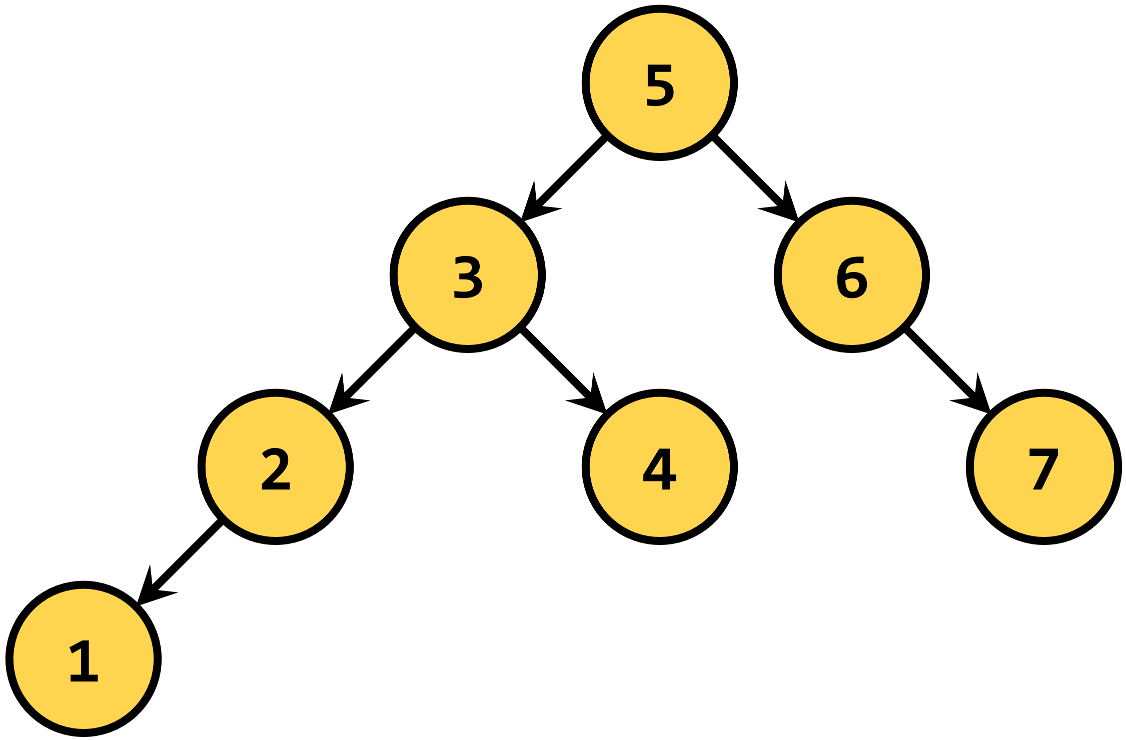
e.g. Finding an order to take courses that satisfies prerequisites.



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.



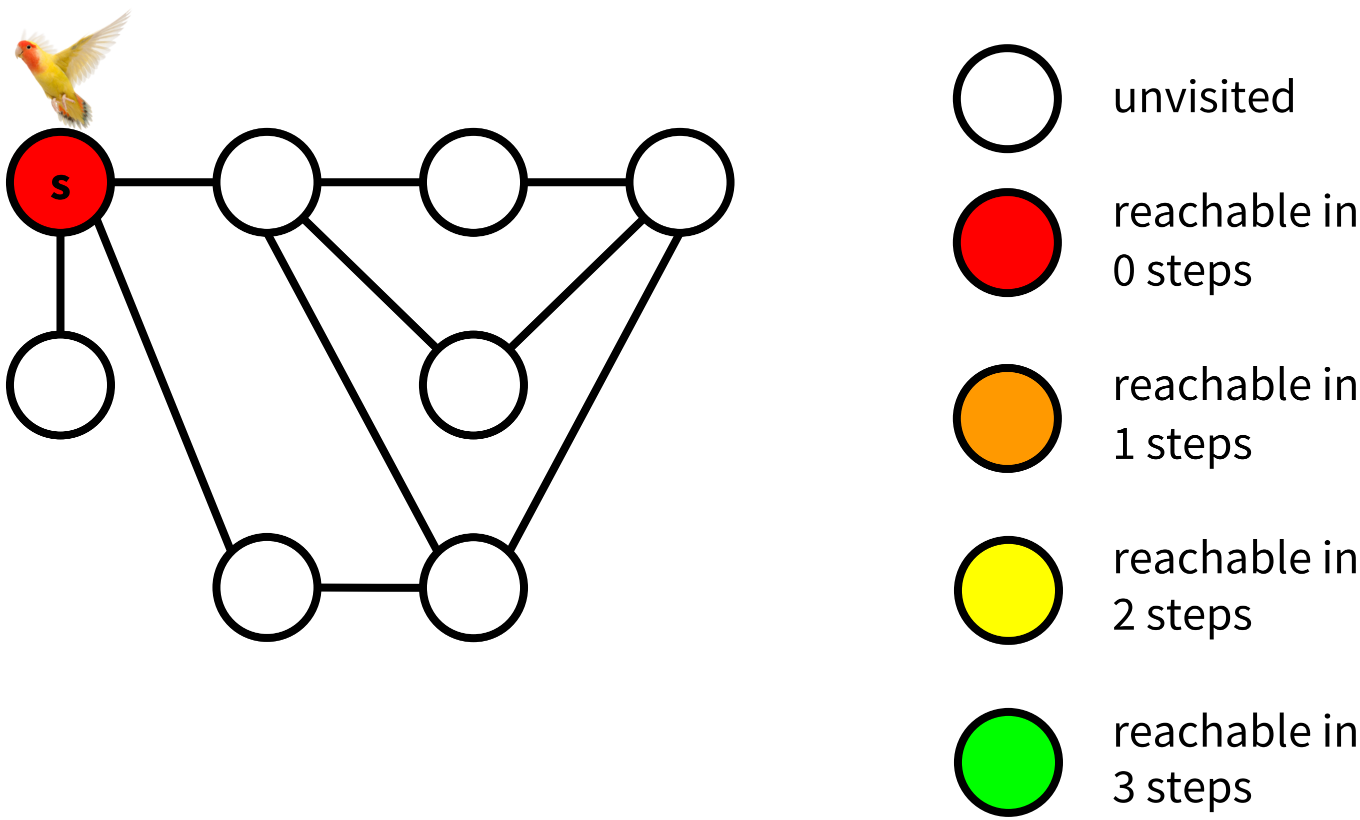
In-order traversal: visit left subtree -> visit the node -> visit the right tree

Breadth-First Search

Breadth-First Search

An analogy

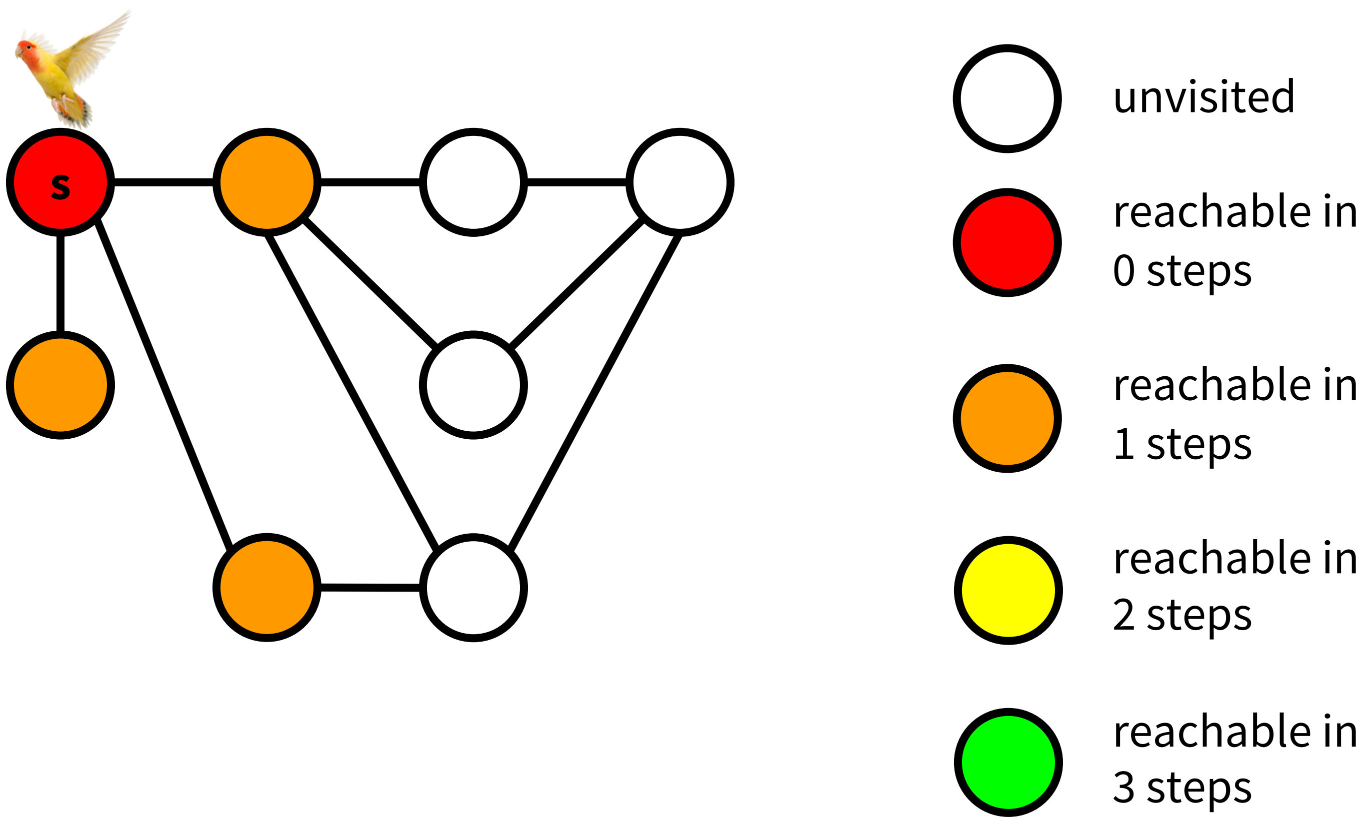
A bird exploring a labyrinth from above (with a bird's eye view).



Breadth-First Search

An analogy

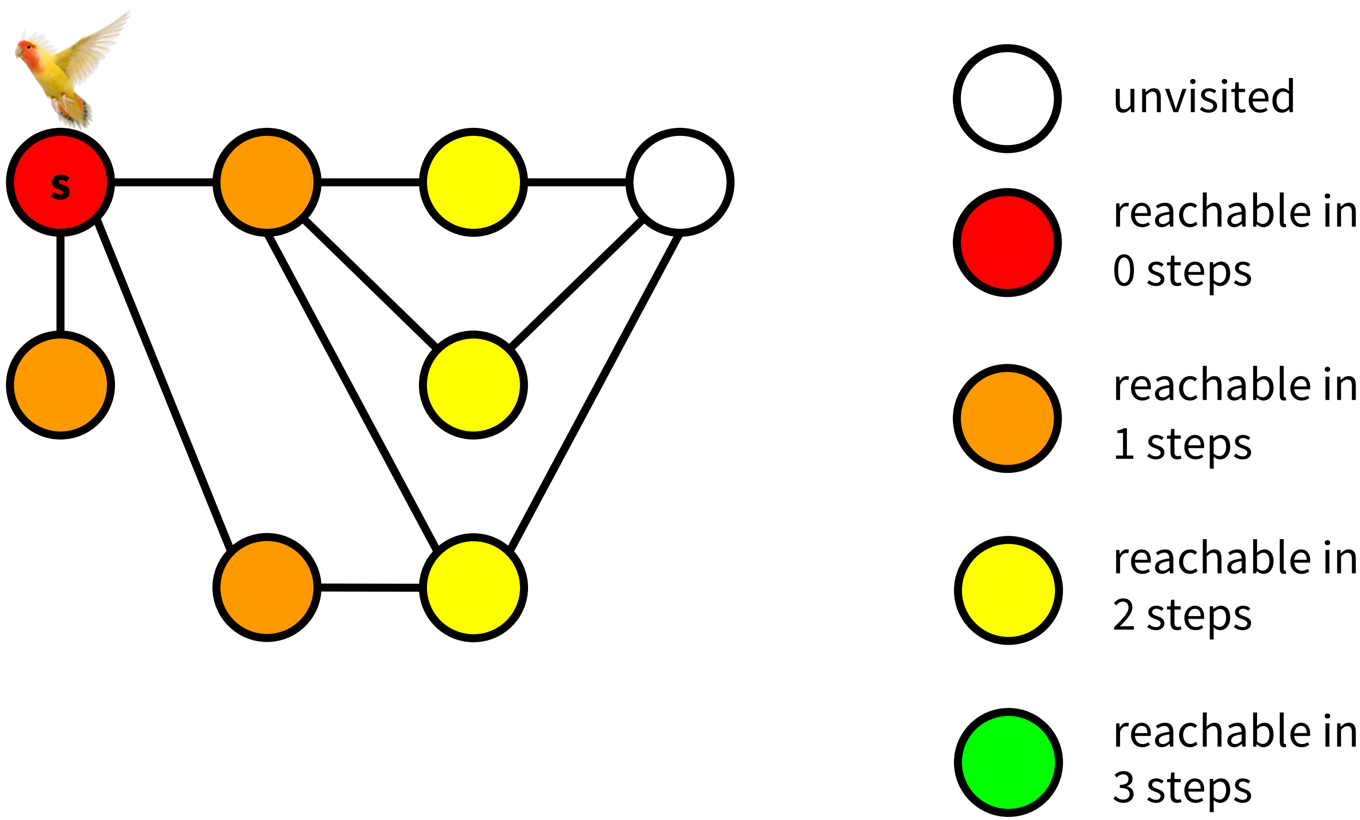
A bird exploring a labyrinth from above (with a bird's eye view).



Breadth-First Search

An analogy

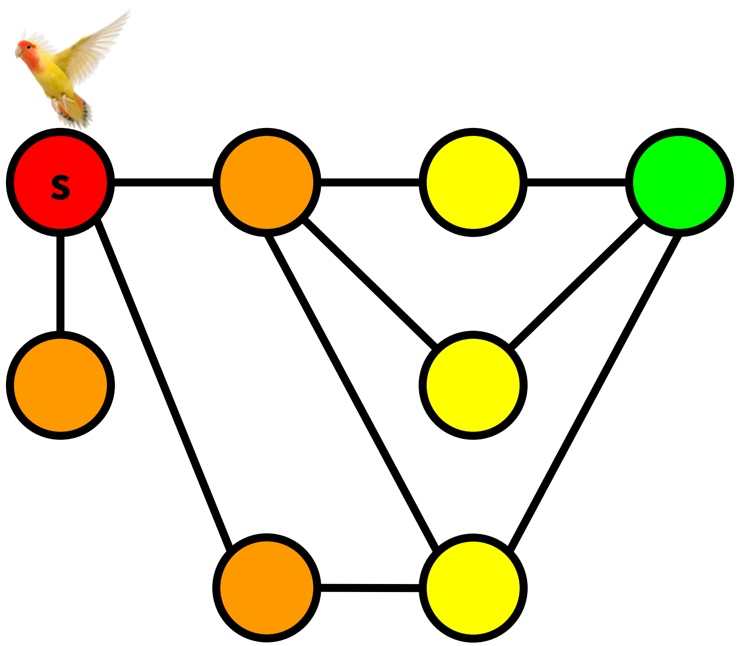
A bird exploring a labyrinth from above (with a bird's eye view).



Breadth-First Search

An analogy

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



unvisited

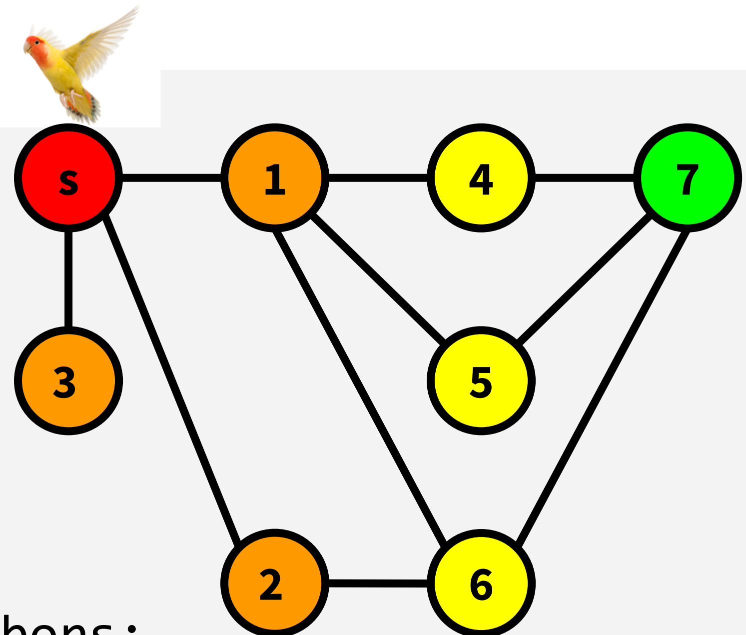
reachable in 0 steps

reachable in 1 steps

reachable in 2 steps

reachable in 3 steps

Breadth-First Search

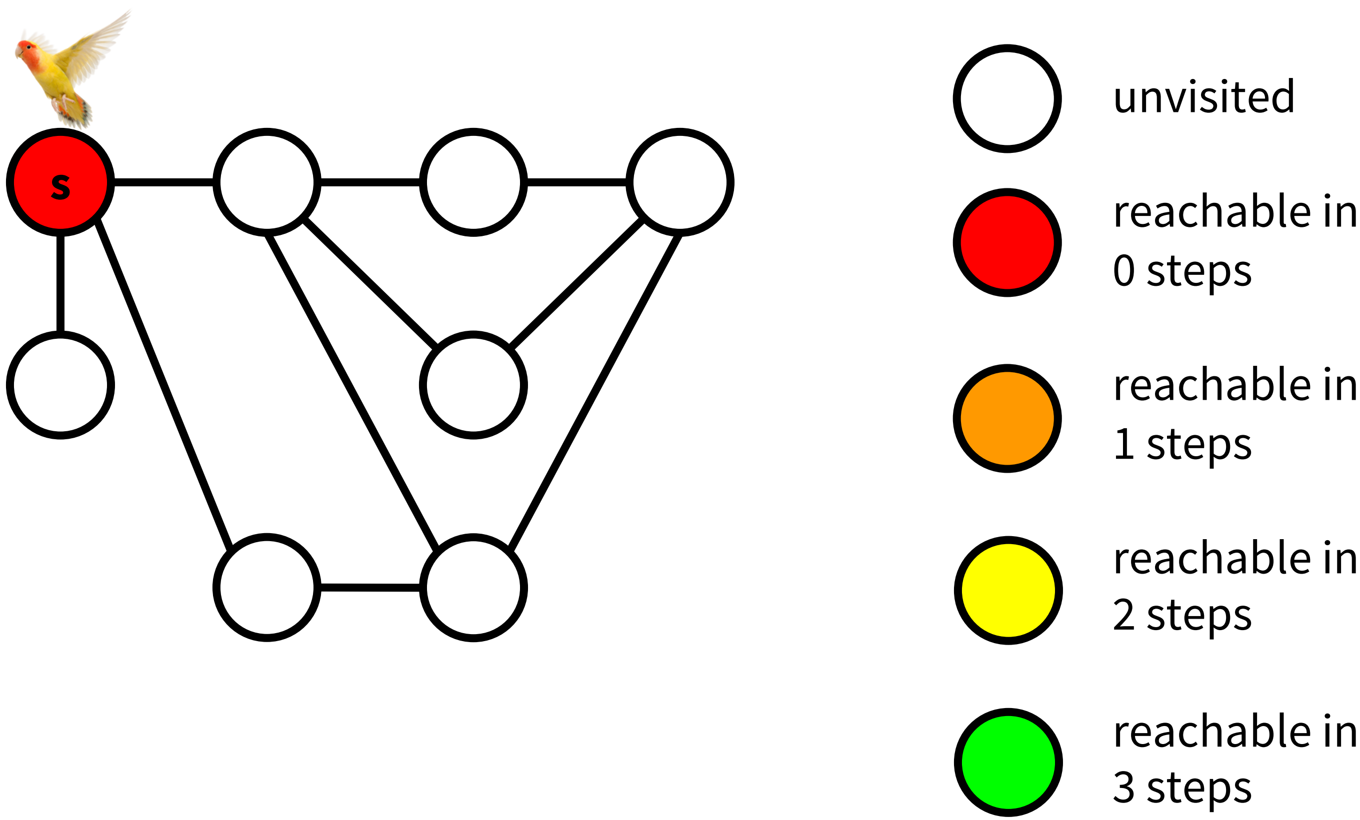



```
algorithm bfs(s):
  L = []
  for i = 0 to n-1:
    L[i] = {}
  L[0] = {s}
  for i = 0 to n-1:
    for u in L[i]:
      for v in u.neighbors:
        if v.status is "unvisited":
          v.status = "visited"
          L[i+1].add(v)
```

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

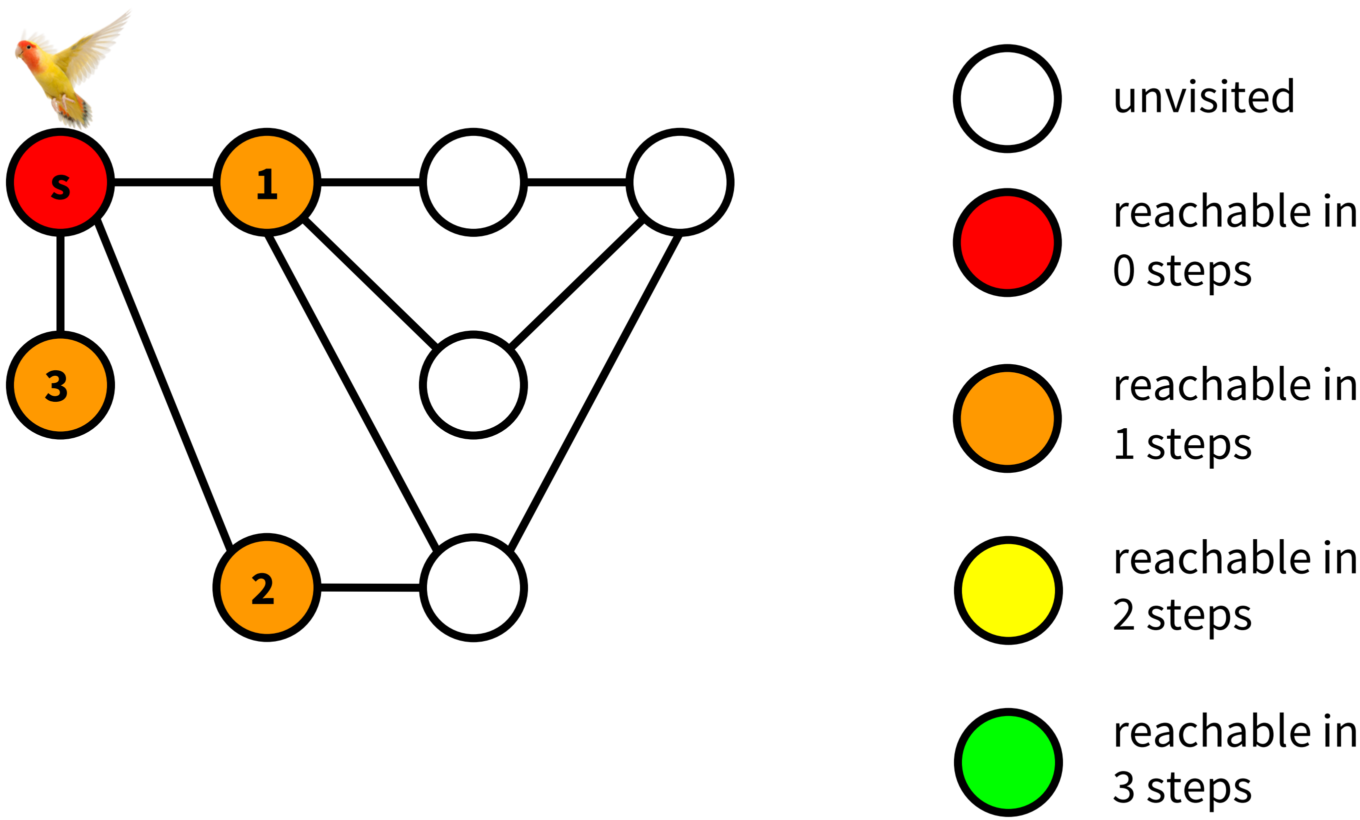
Breadth-First Search

L[0] = {s}   // Initialize



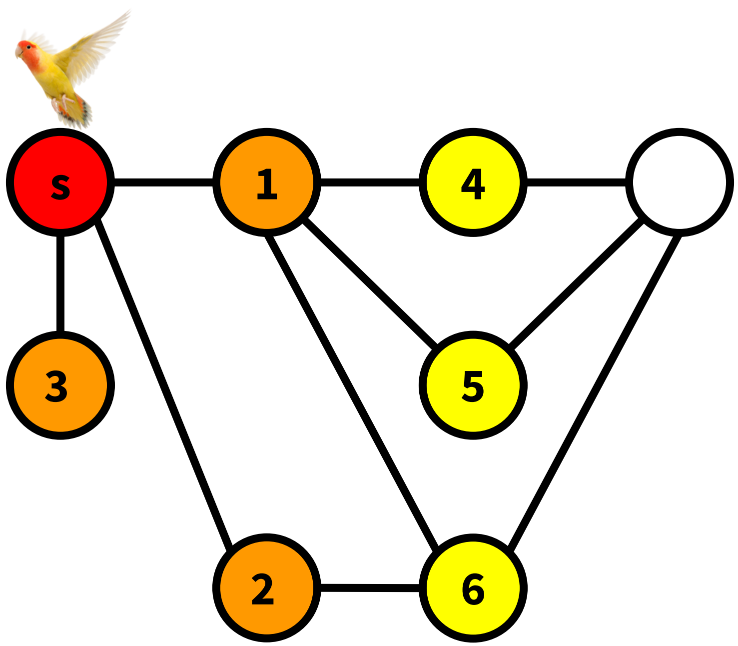
Breadth-First Search

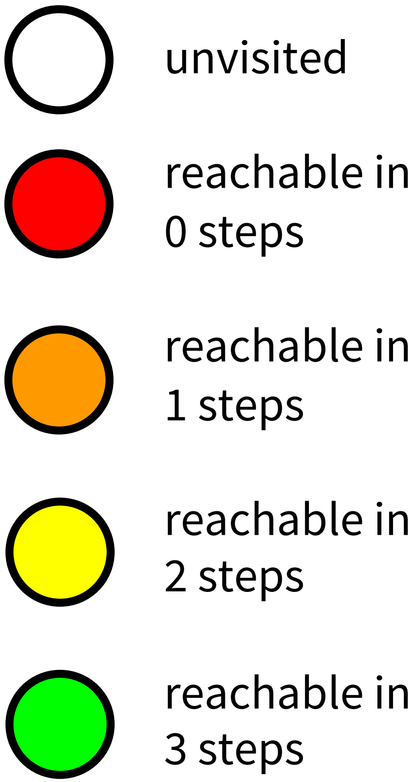
L[0] = {s}   // Initialize
L[1] = {1, 2, 3}   // Take out s from L[0], visit its (unvisited) neighbors and put them in L[1]



Breadth-First Search

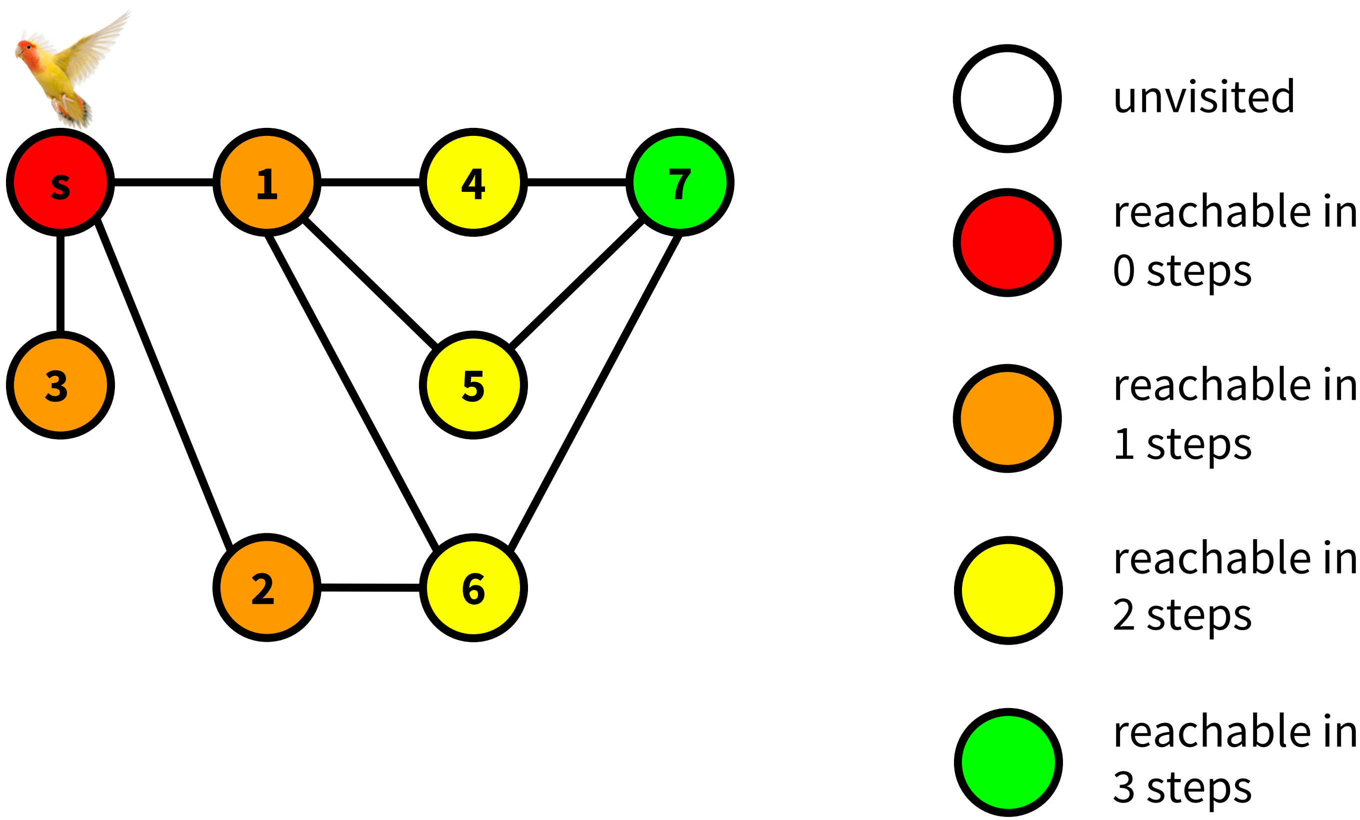
L[0] = {s} // Initialize
L[1] = {1, 2, 3} // Take out s from L[0], visit its (unvisited) neighbors and put them in L[1]
L[2] = {4, 5, 6} // Take out 1, 2, 3 from L[1], visit their (unvisited) neighbors and put them in L[2]





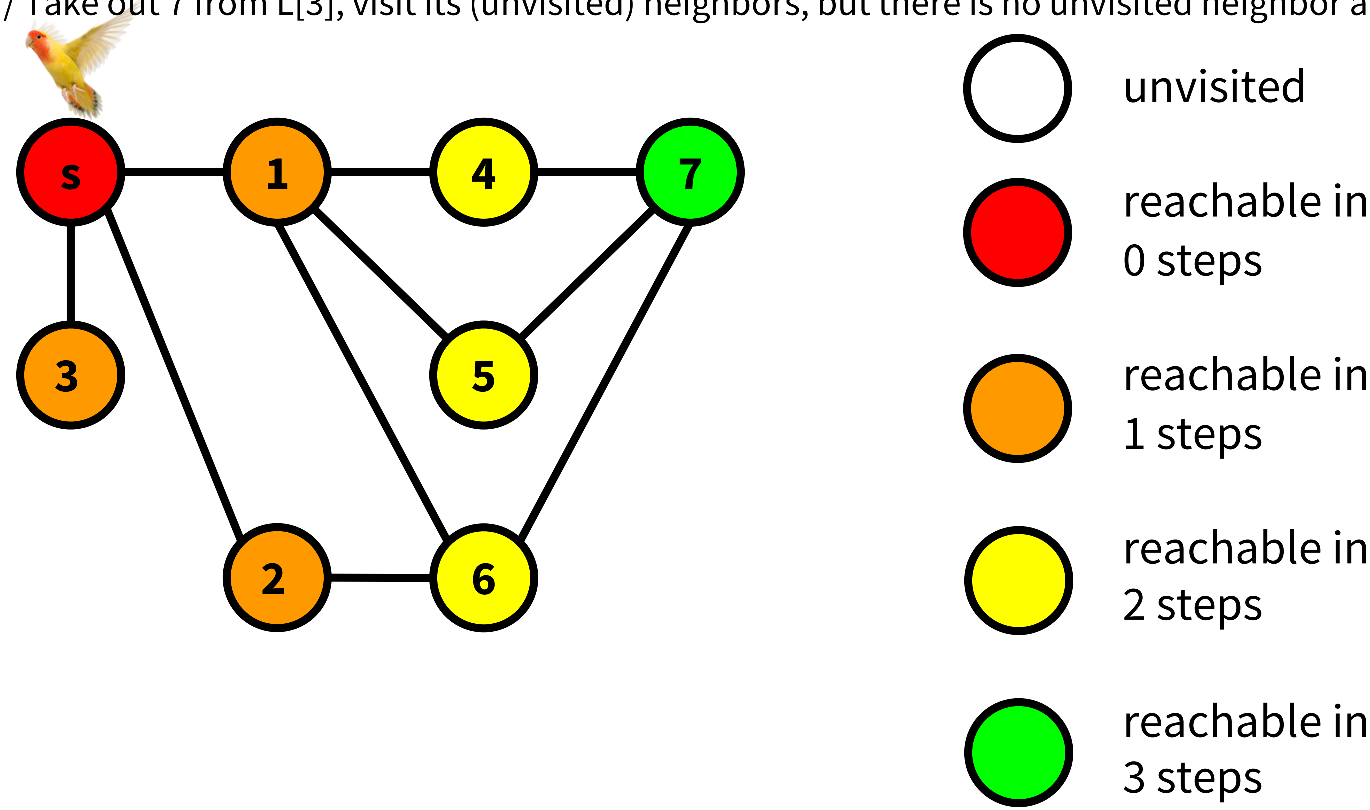
Breadth-First Search

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?

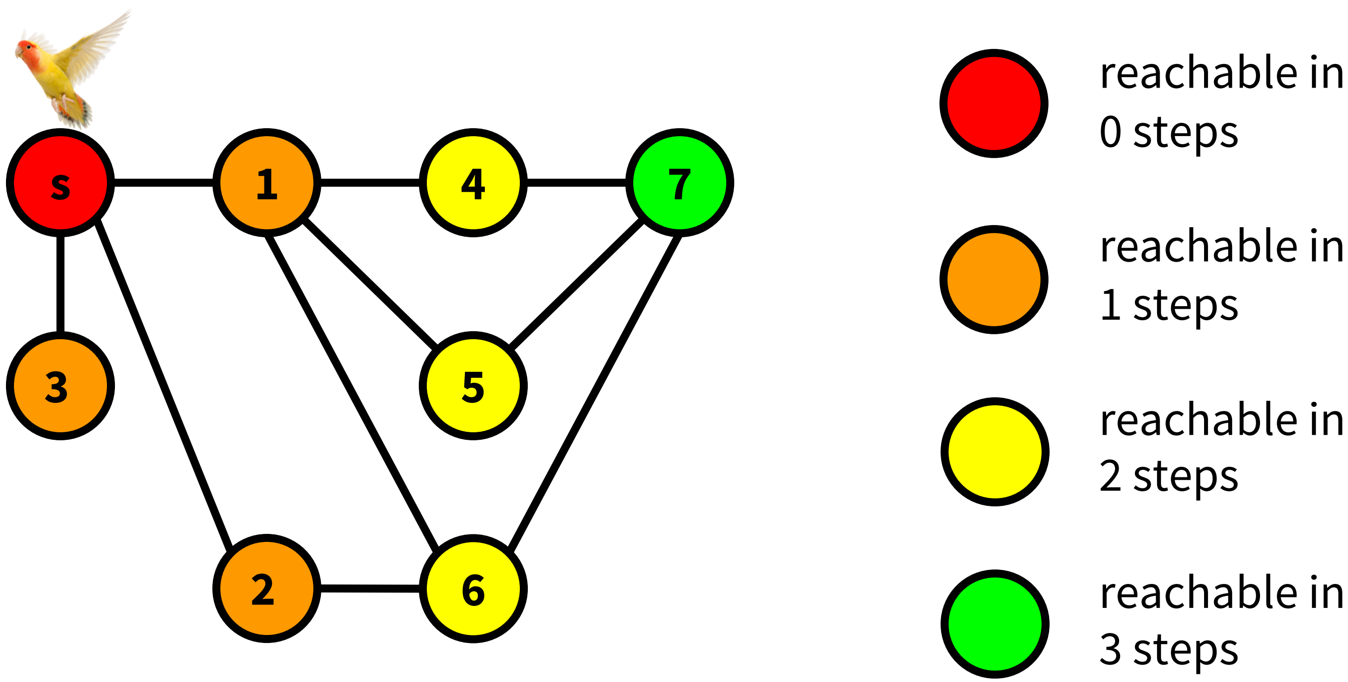
- Call `bfs(u)`.
- For all vertices in `L[i]`, the shortest path between u and these vertices has length i.
- If v isn't in `L[i]` for any i, then it's unreachable from u.

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.

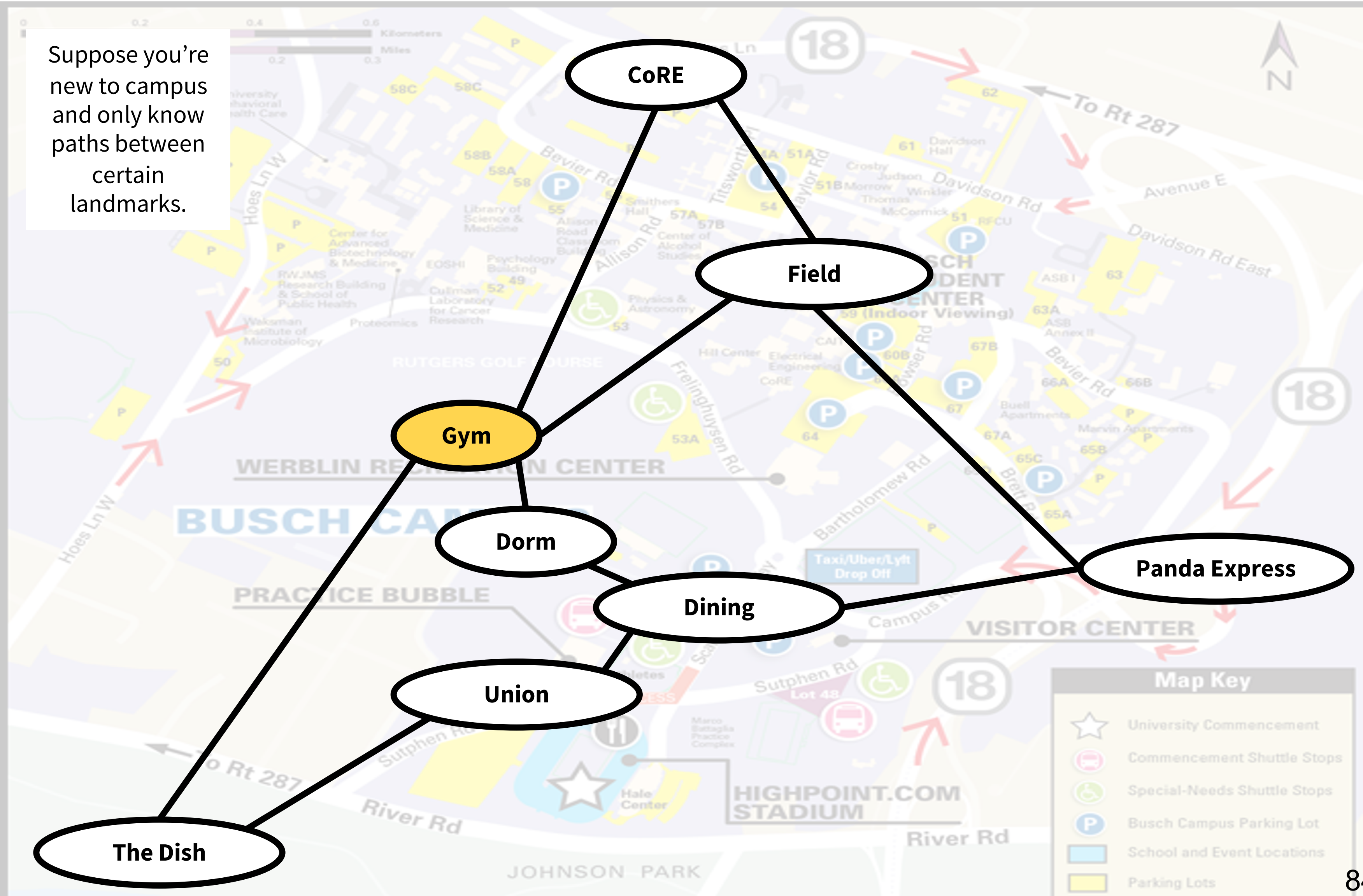
We know the shortest path between s and node 5 has length 2, because node 5 appears in L[2].



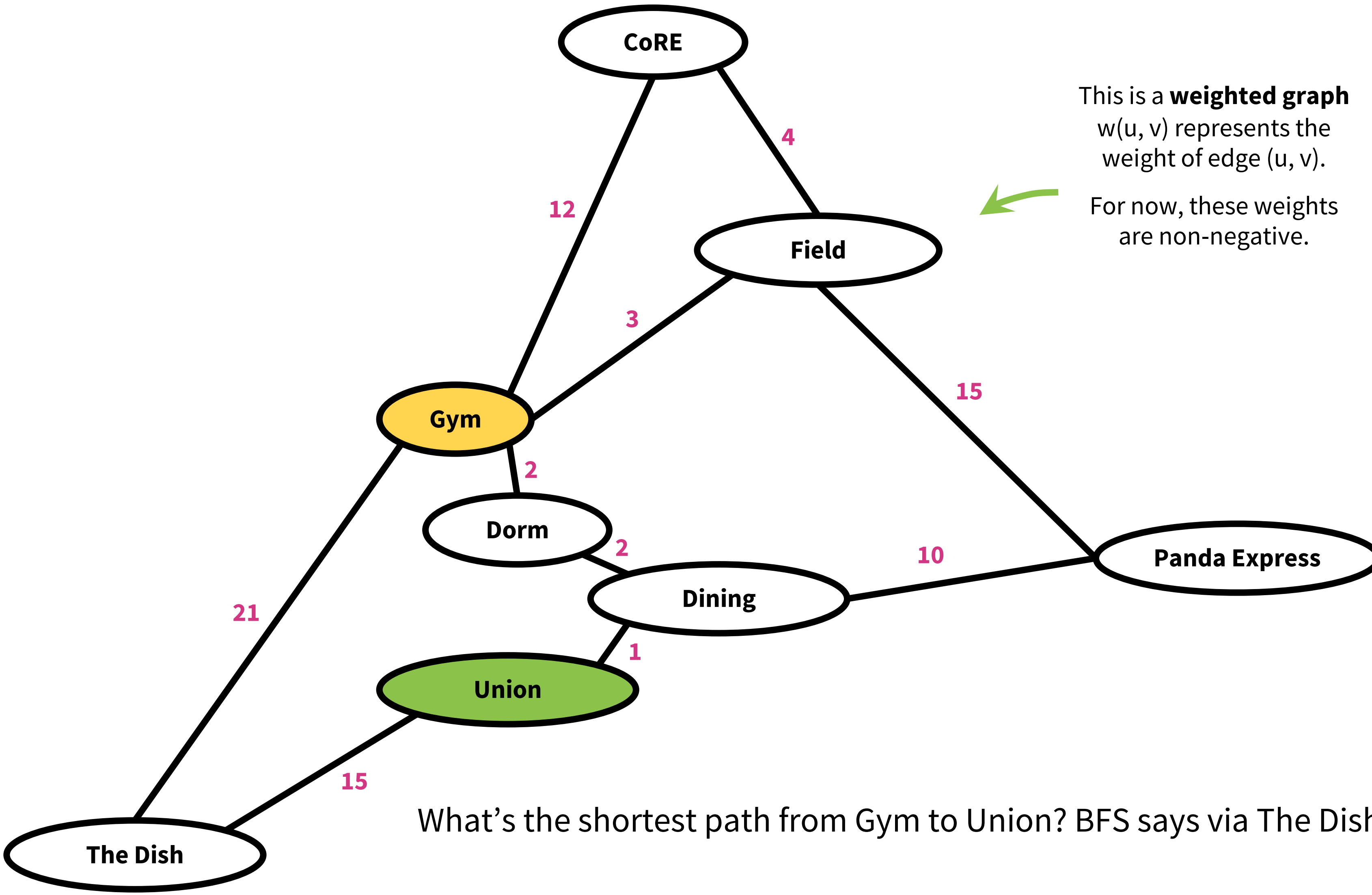
Dijkstra’s Algorithm

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.

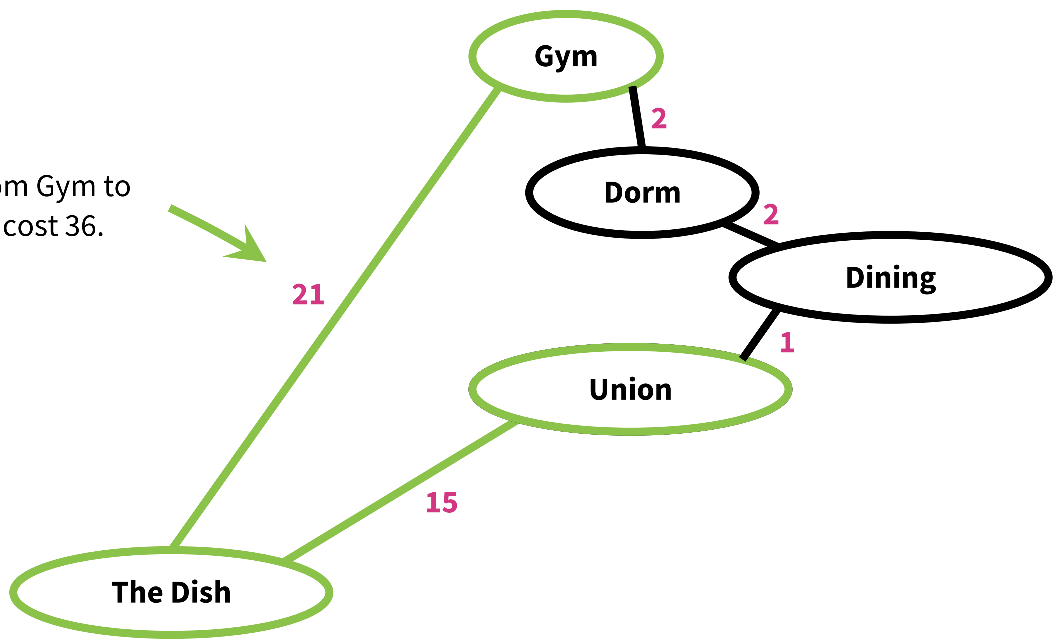
Shortest Path

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.

This path from Gym to Union has cost 36.

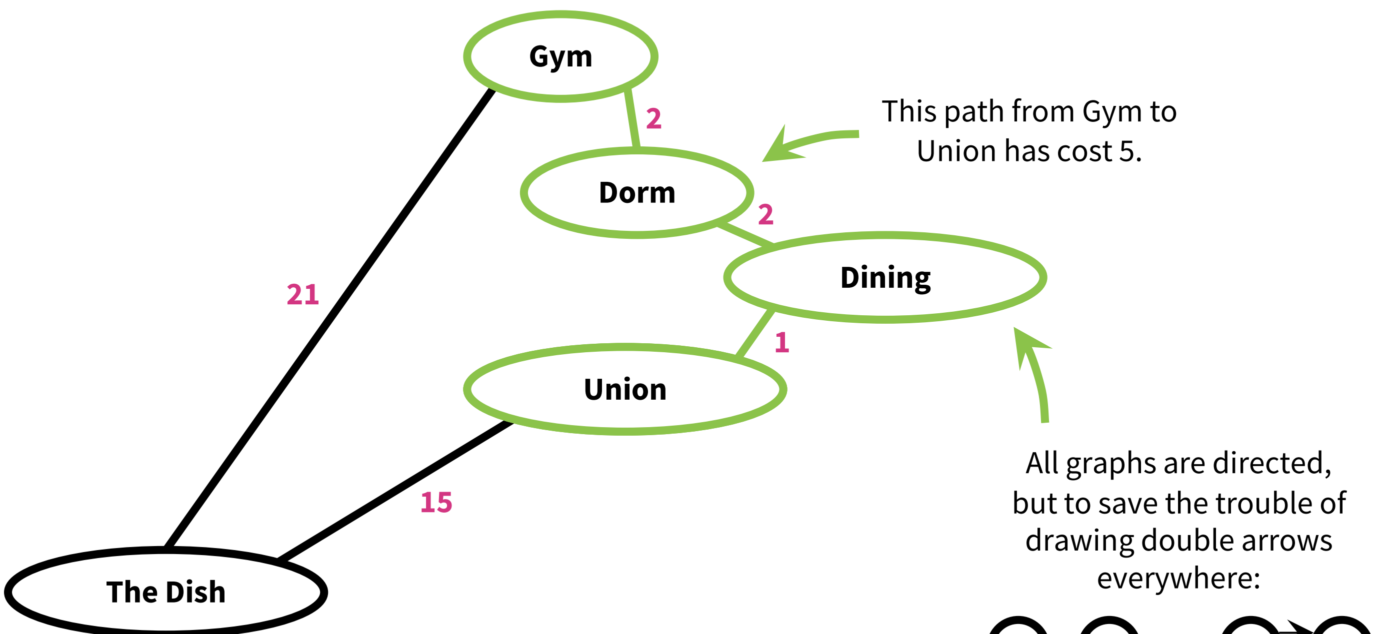


Shortest Path

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.

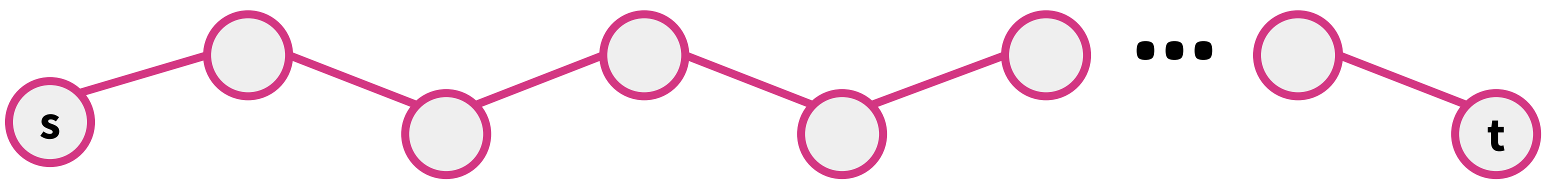
The shortest path is the one with the minimum cost.



Shortest Path

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

Intuition:

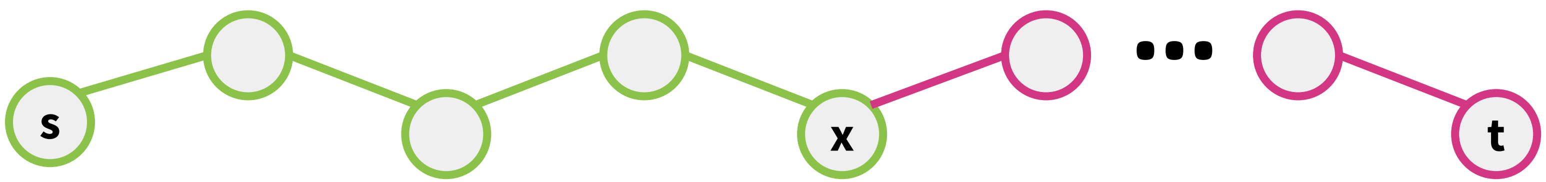


Suppose **this** is a shortest path from **s** to **t**.

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**.

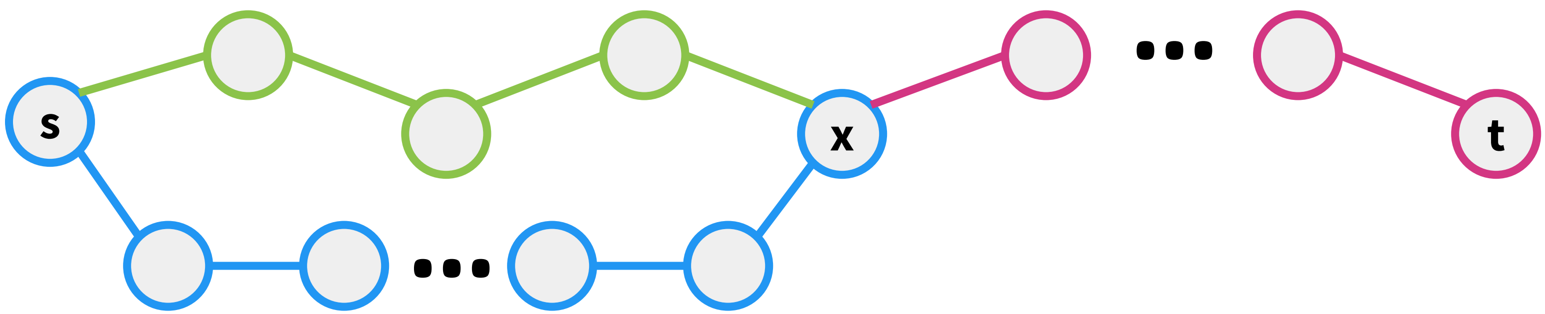
Then **this** is a shortest path from **s** to **x**.

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**.

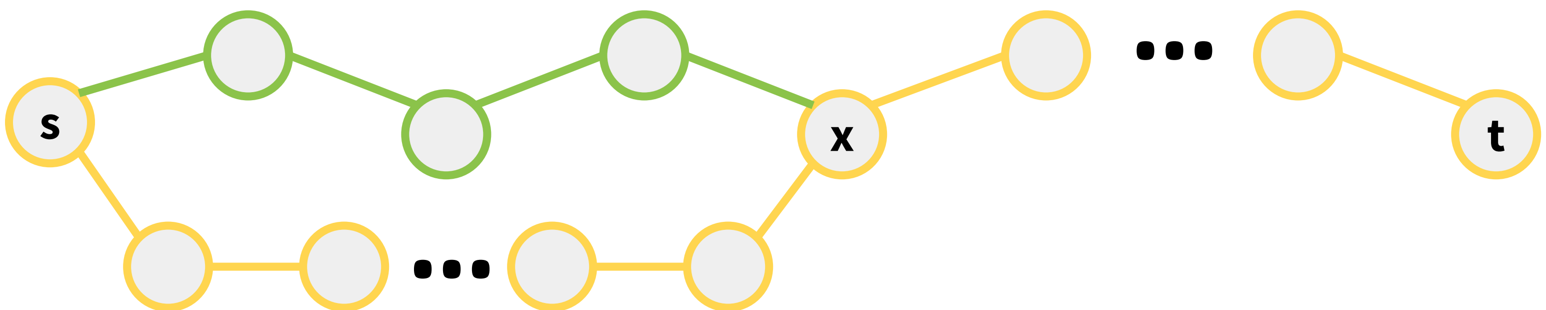
Then **this** is a shortest path from **s** to **x**.

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:



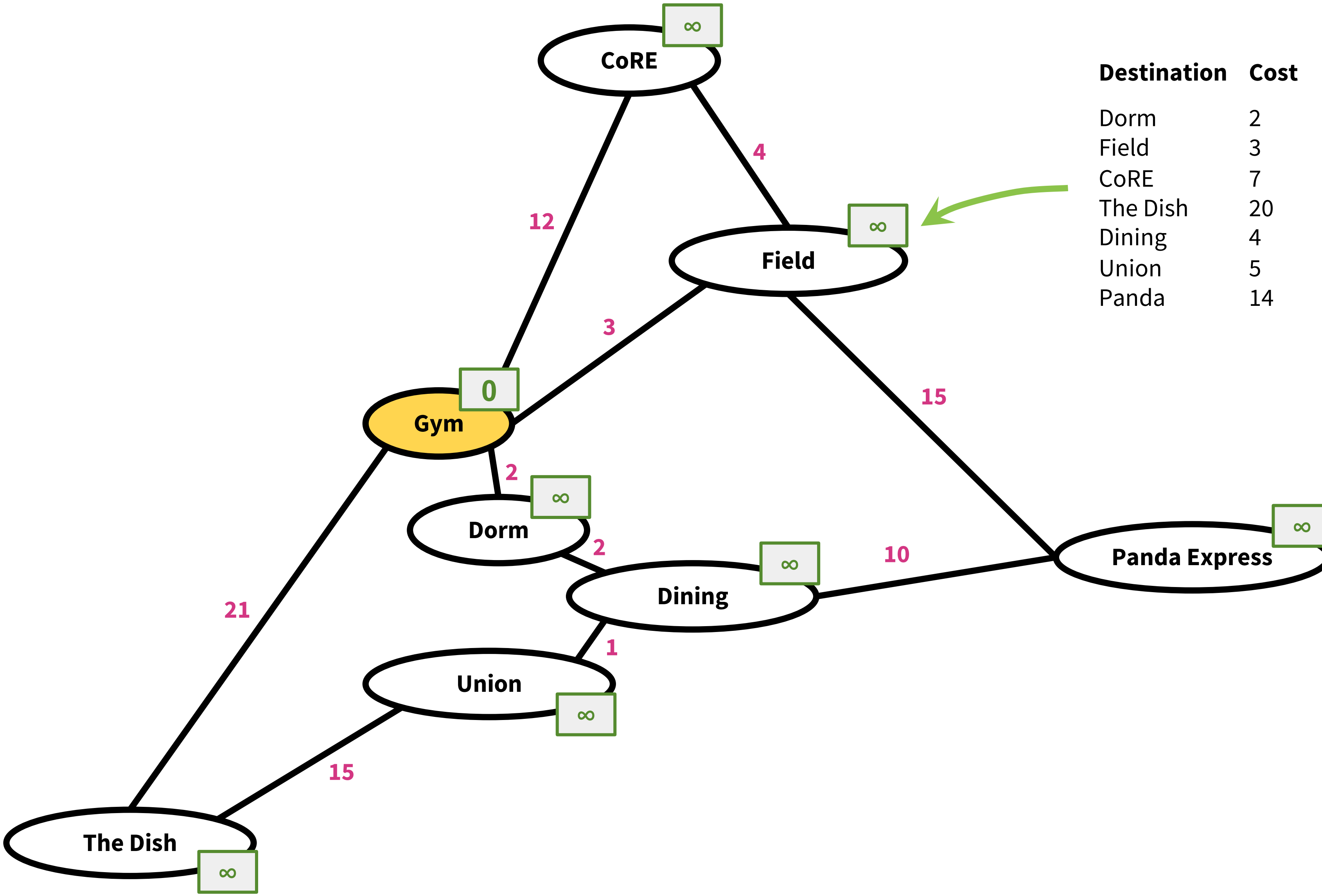
Suppose **this** is a shortest path from **s** to **t**.

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



Destination	Cost
Dorm	2
Field	3
CoRE	7
The Dish	20
Dining	4
Union	5
Panda	14

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.

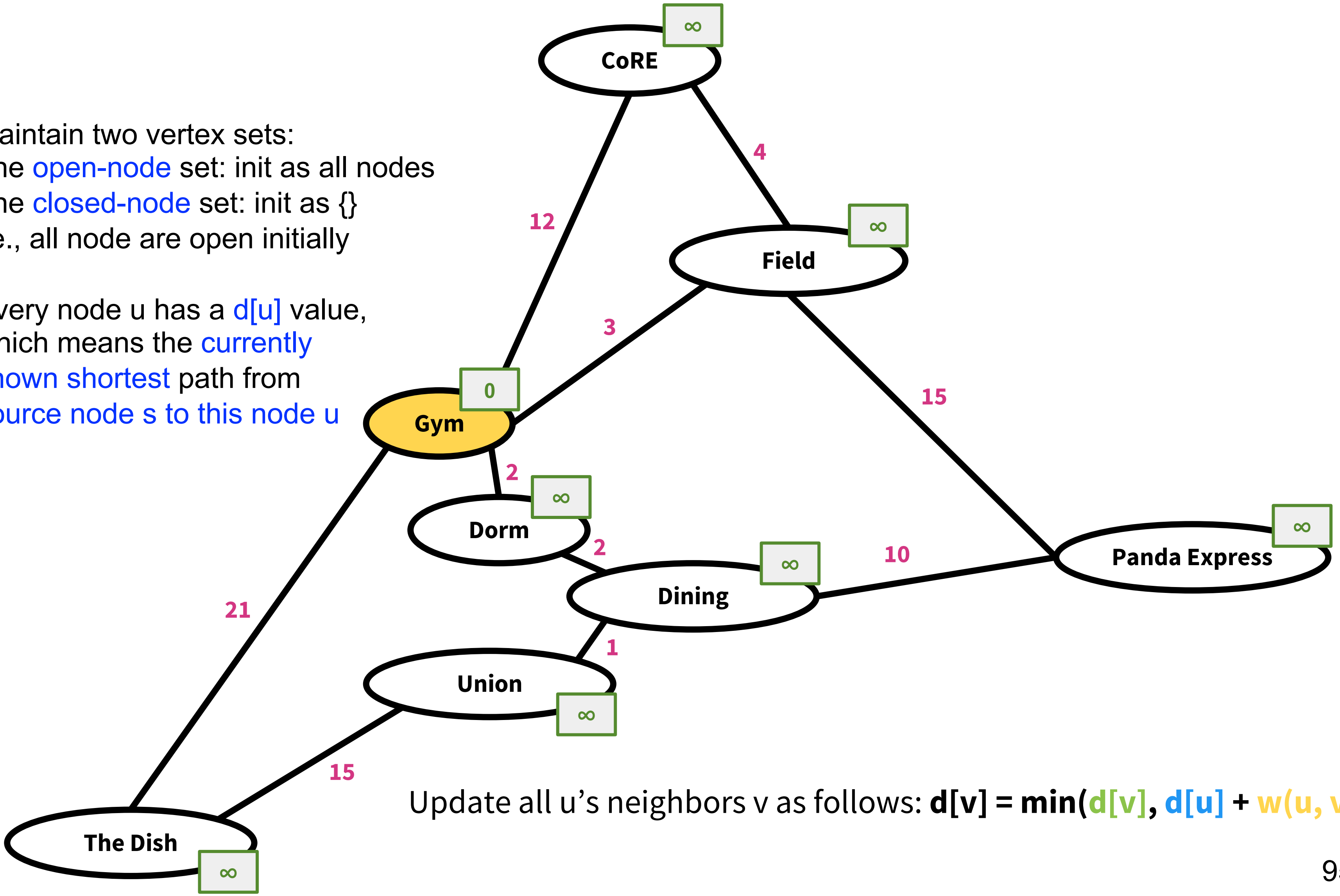
Dijkstra’s Algorithm

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

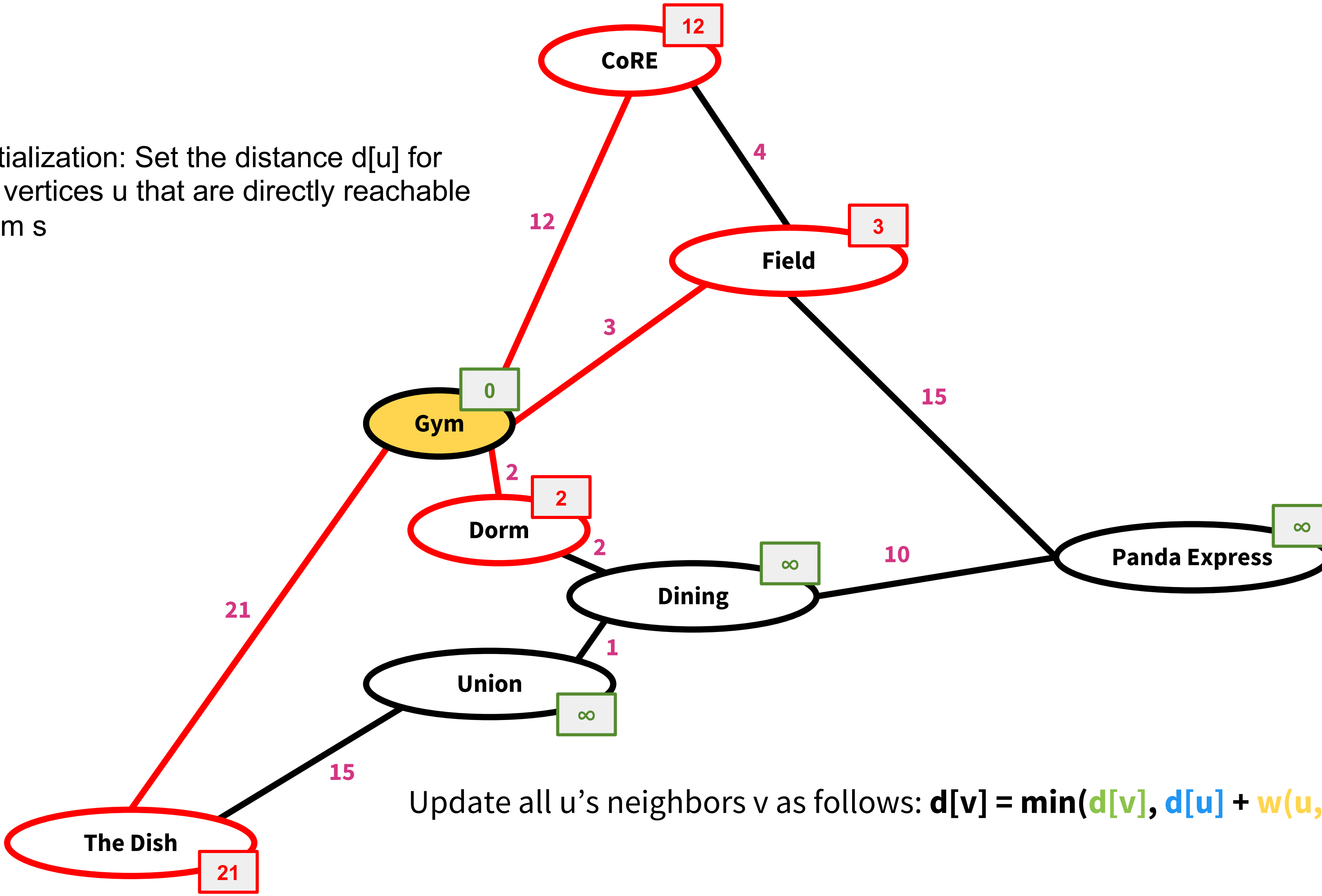
Every node u has a d[u] value, which means the currently known shortest path from source node s to this node u



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

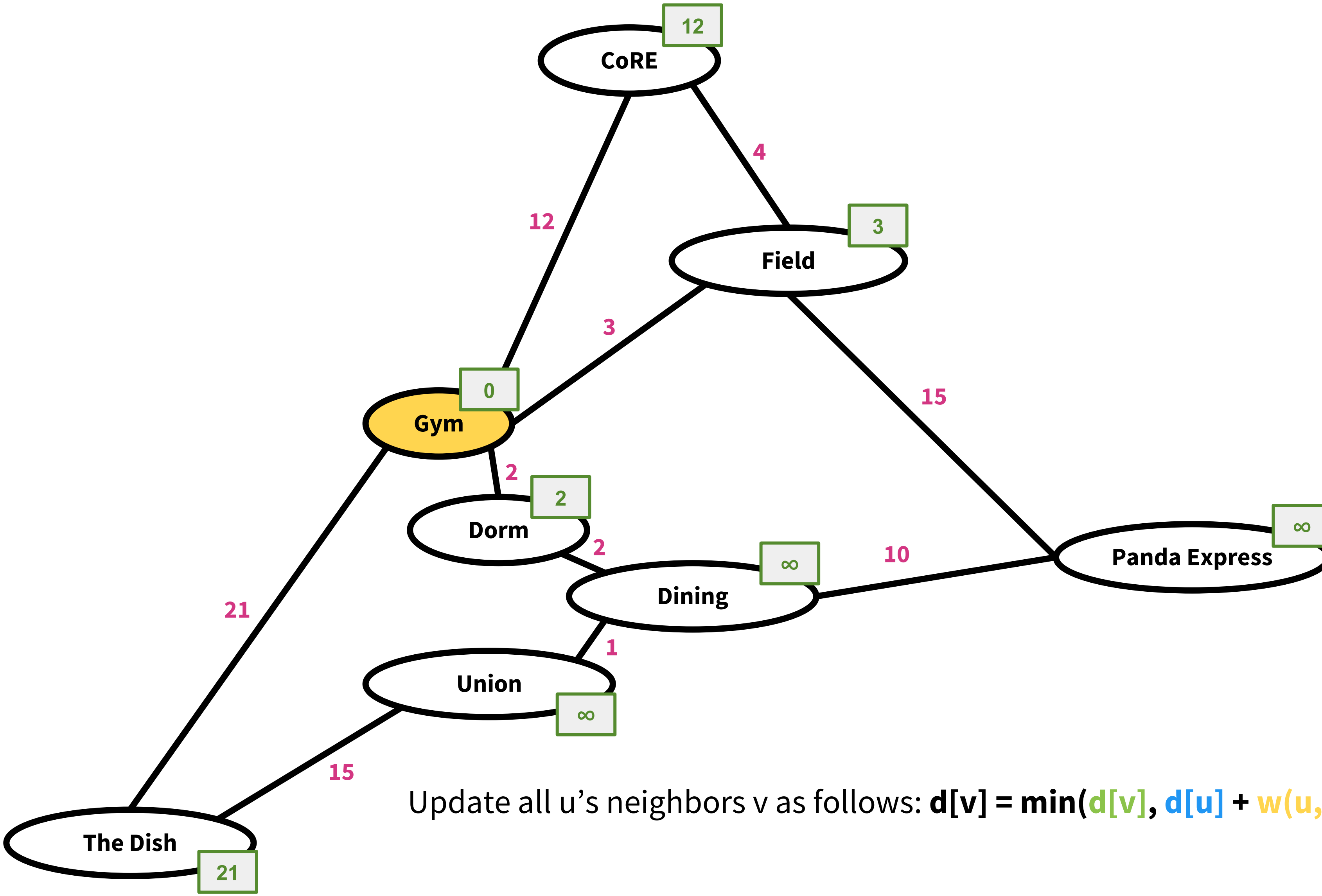
Dijkstra’s Algorithm

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))**

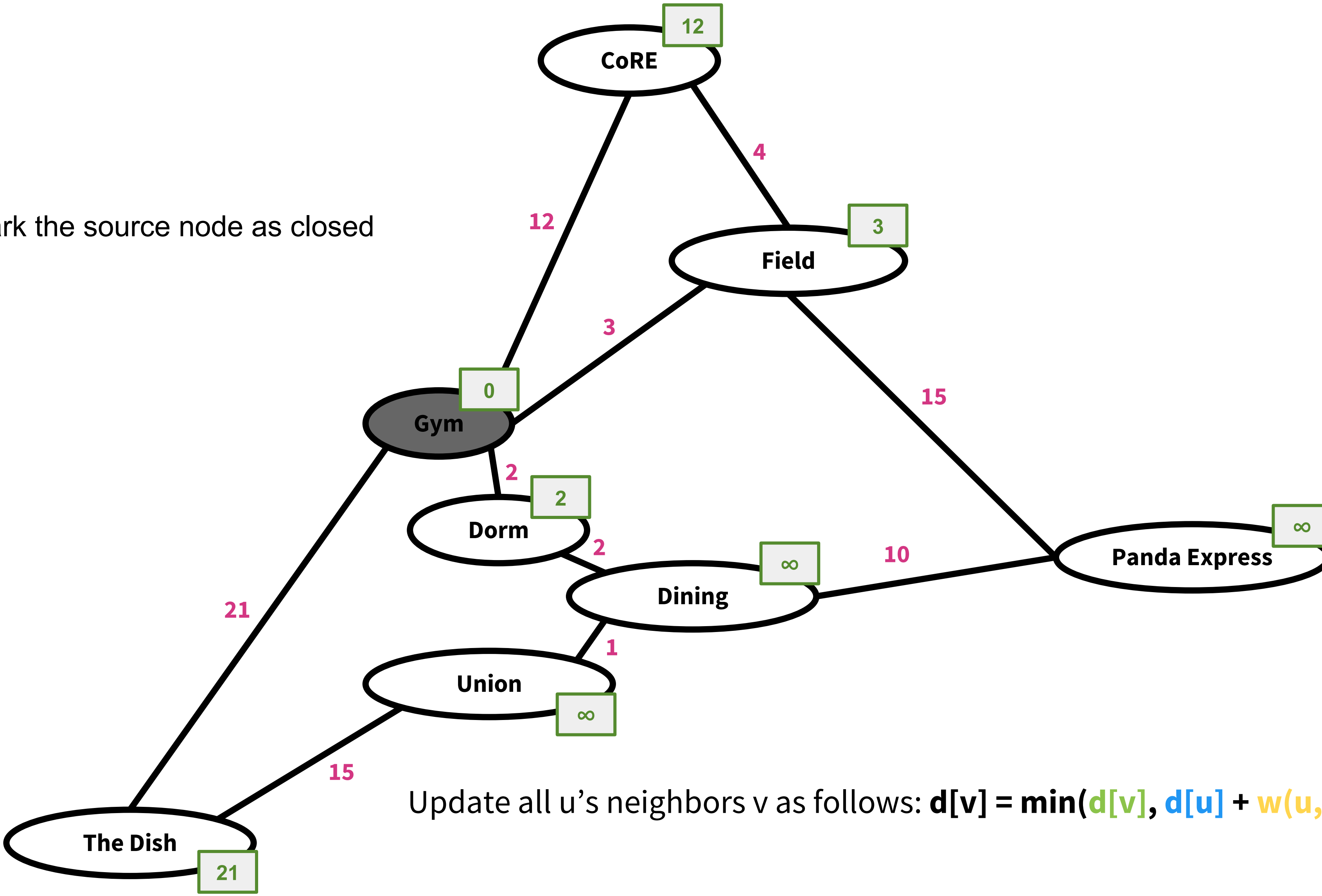
Dijkstra's Algorithm



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

Dijkstra’s Algorithm

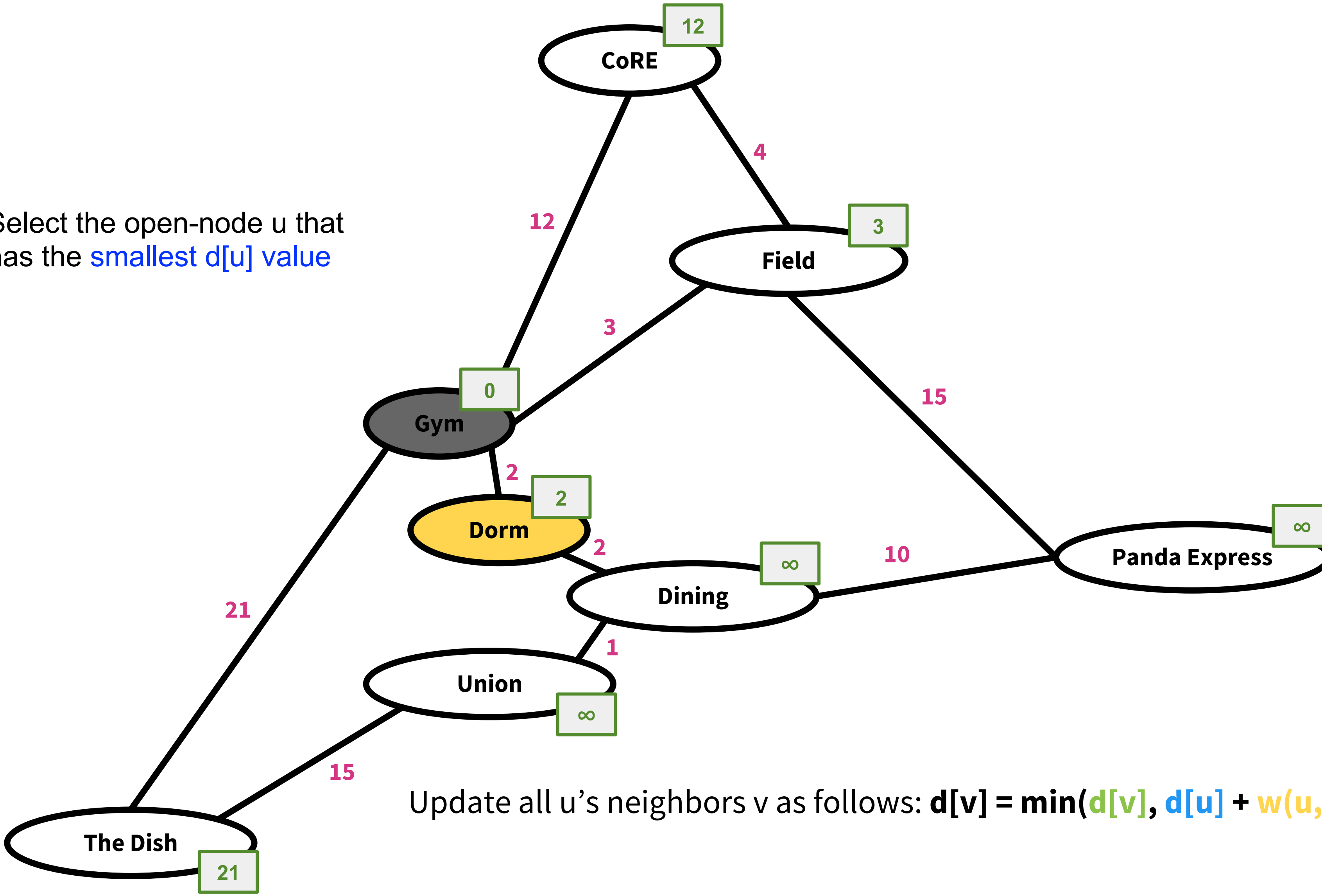
Mark the source node as closed



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

Dijkstra's Algorithm

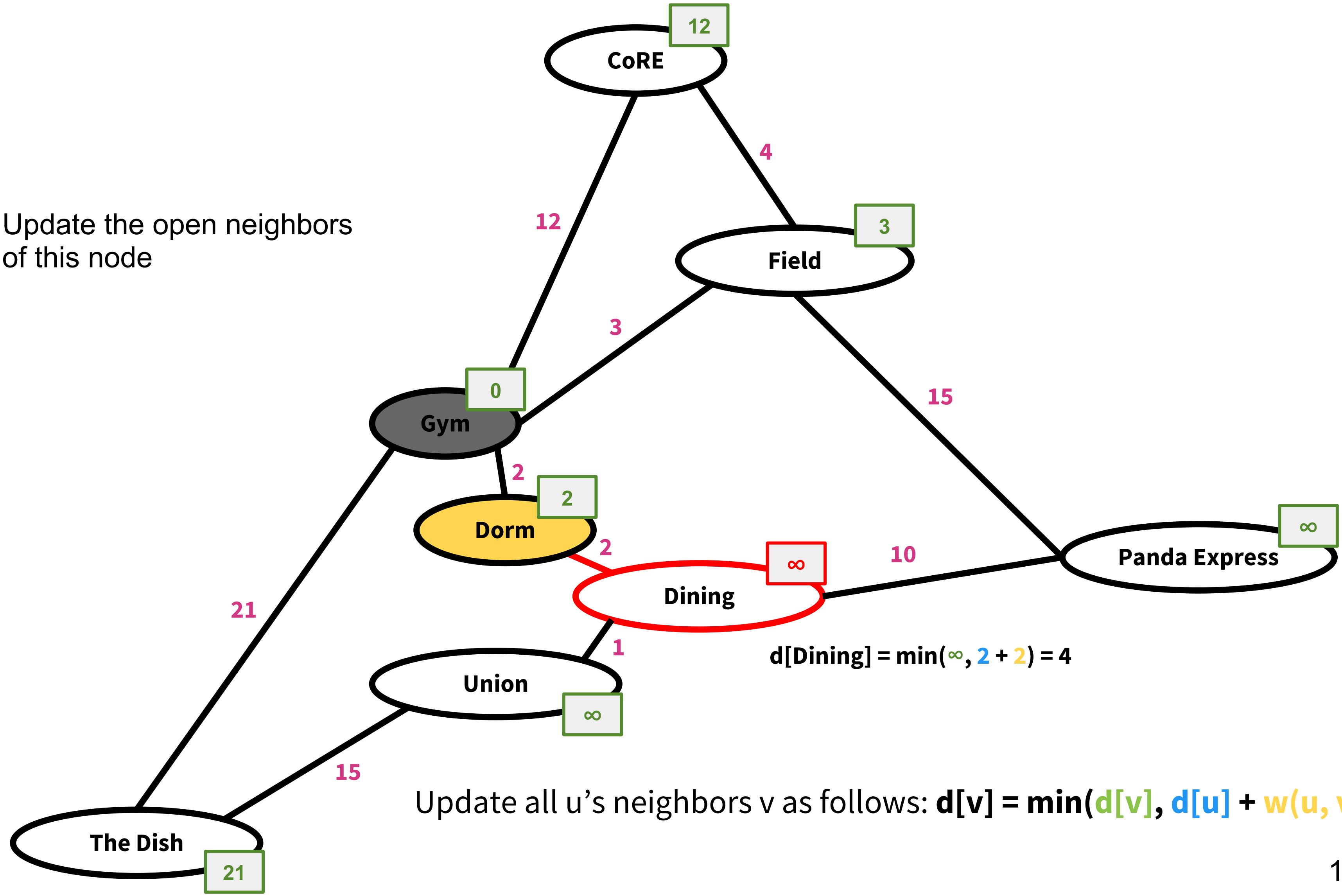
Select the open-node u that has the smallest d[u] value



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

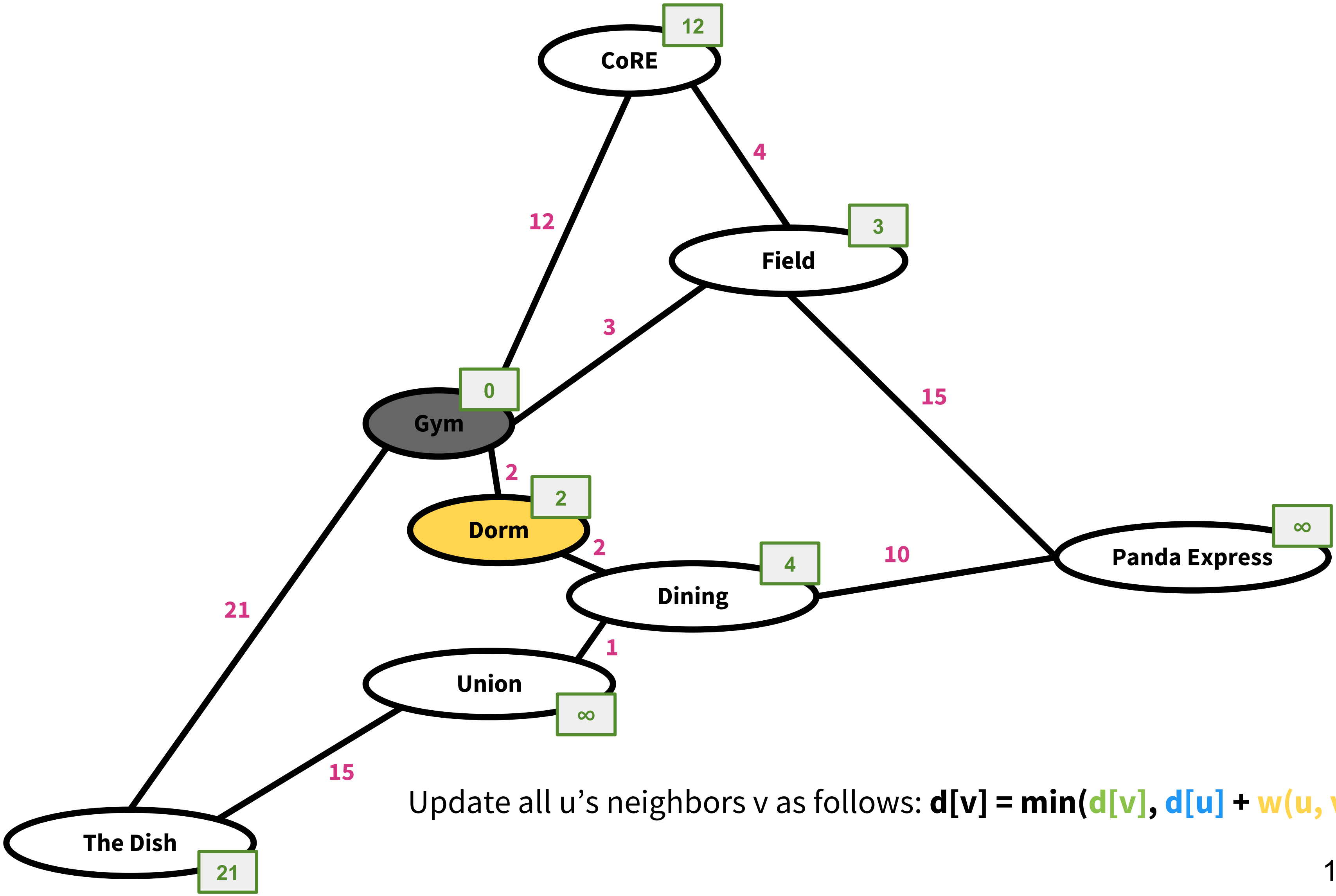
Dijkstra's Algorithm

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))**

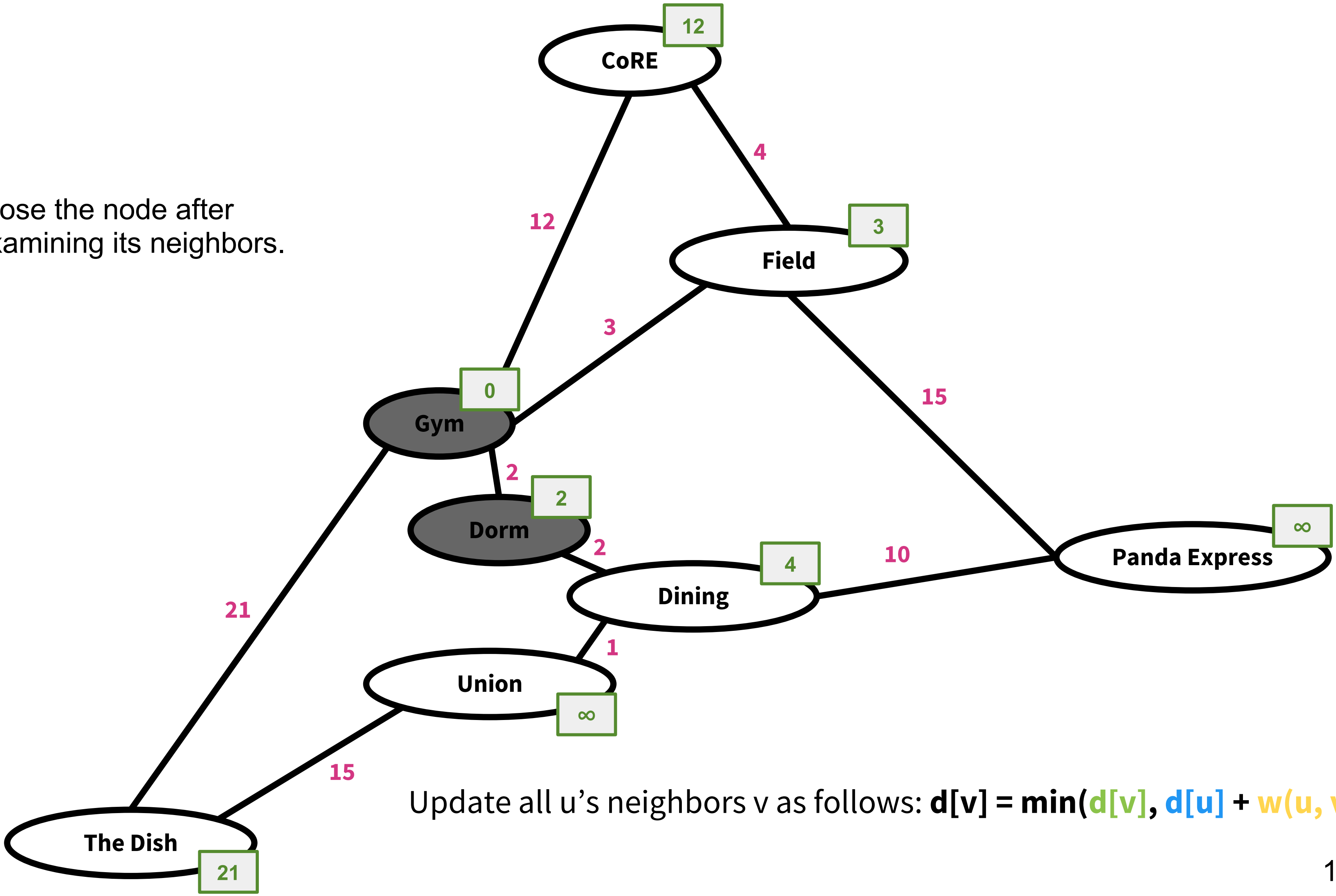
Dijkstra’s Algorithm



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

Dijkstra's Algorithm

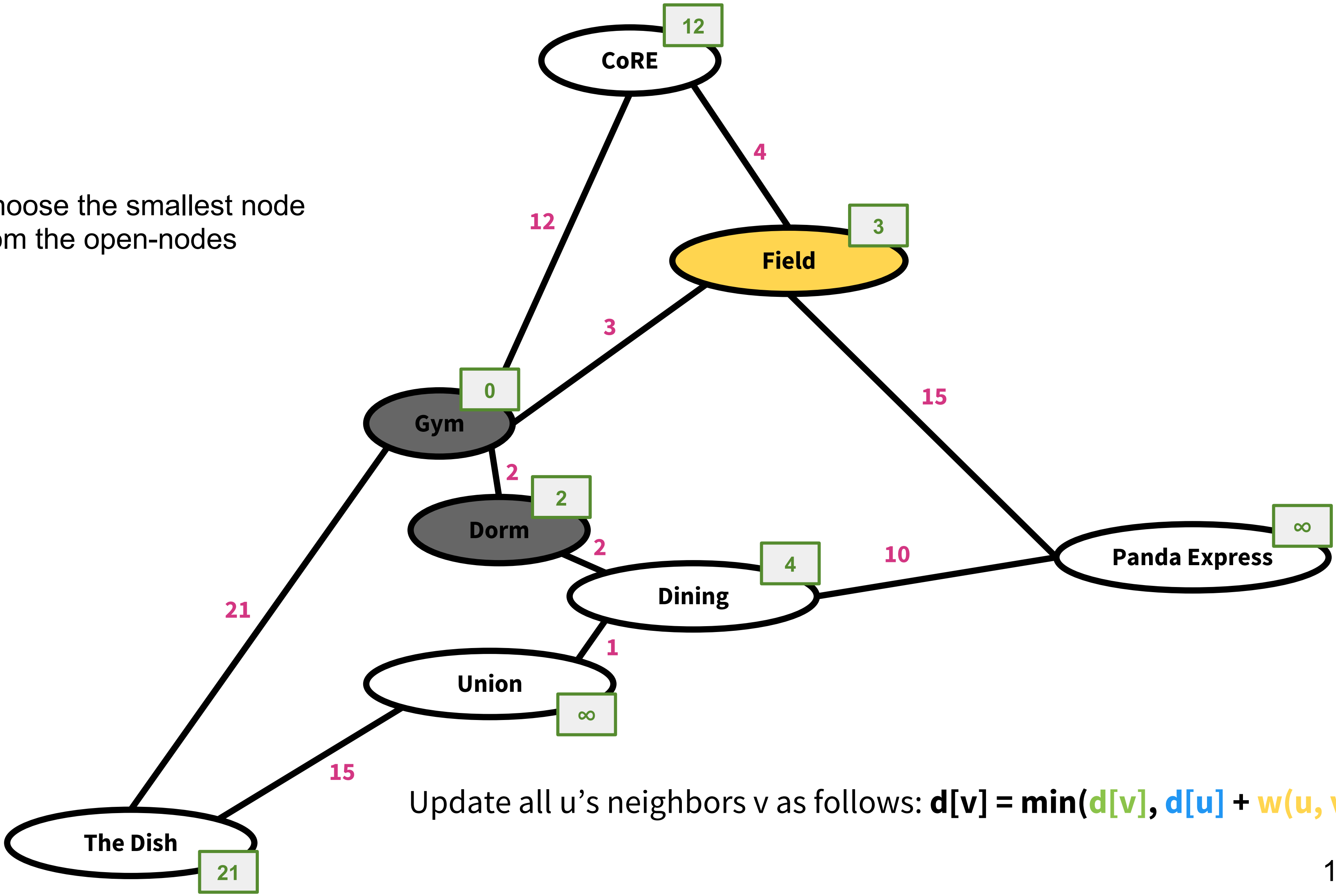
Close the node after examining its neighbors.



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

Dijkstra’s Algorithm

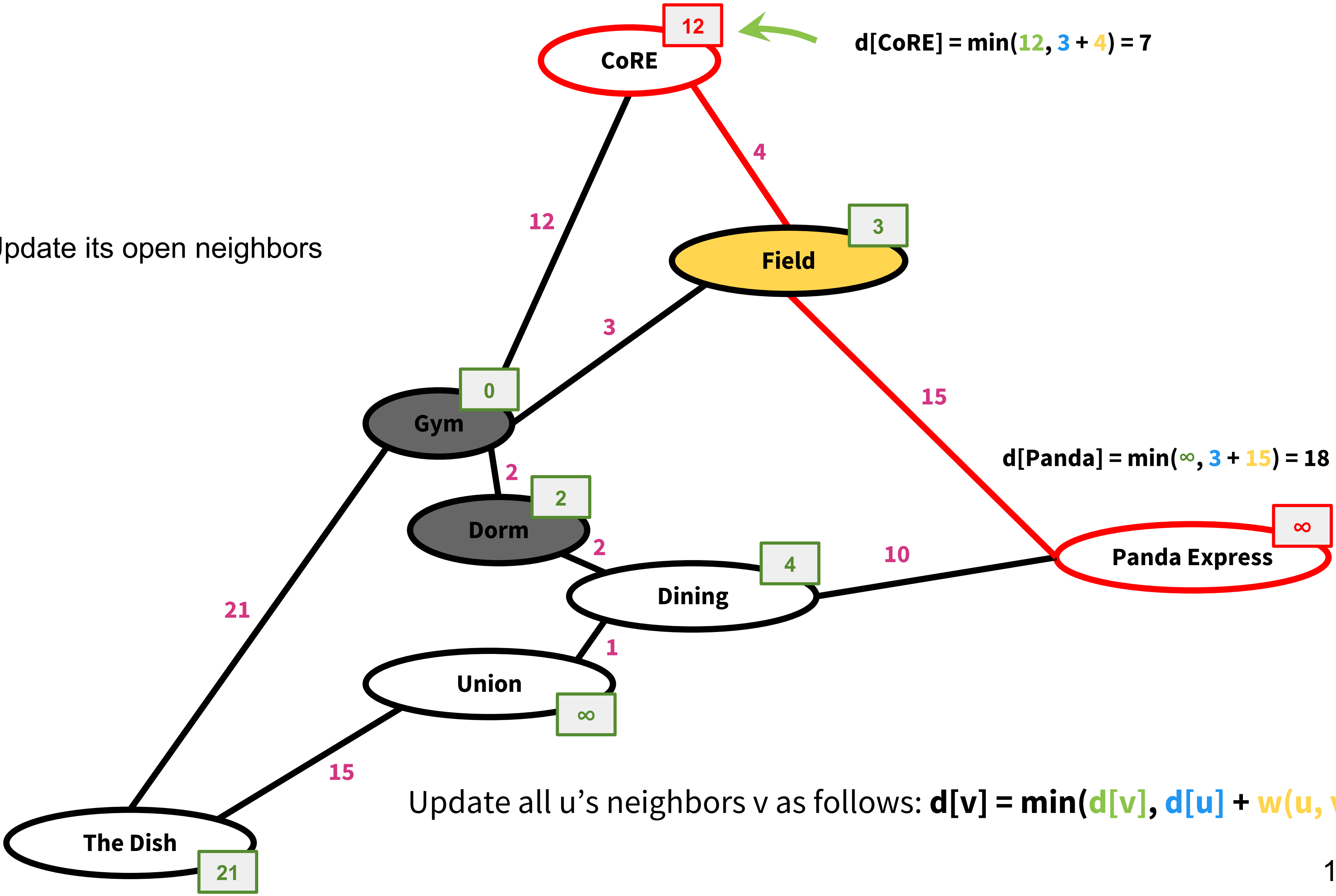
Choose the smallest node from the open-nodes



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

Dijkstra’s Algorithm

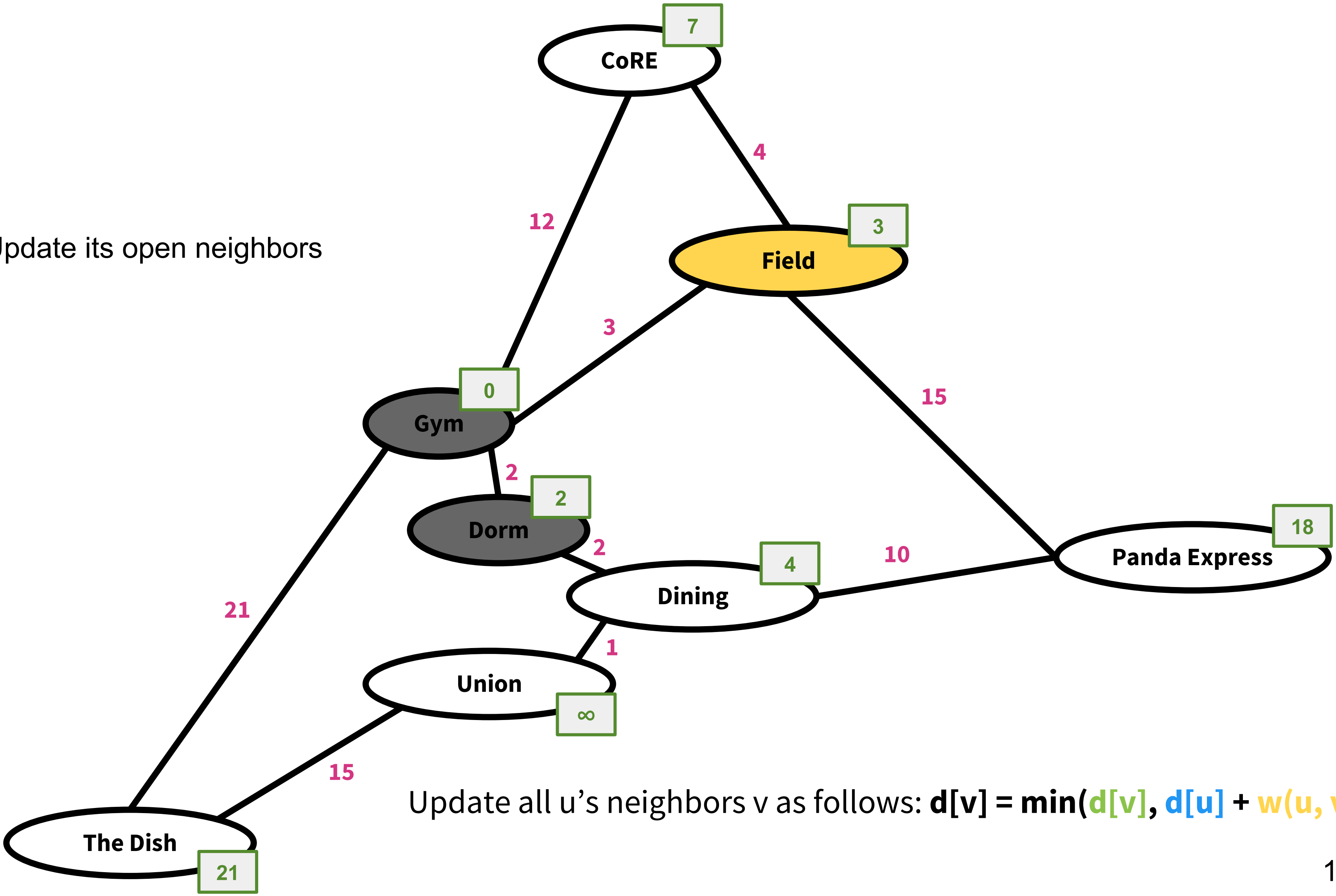
Update its open neighbors



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

Dijkstra's Algorithm

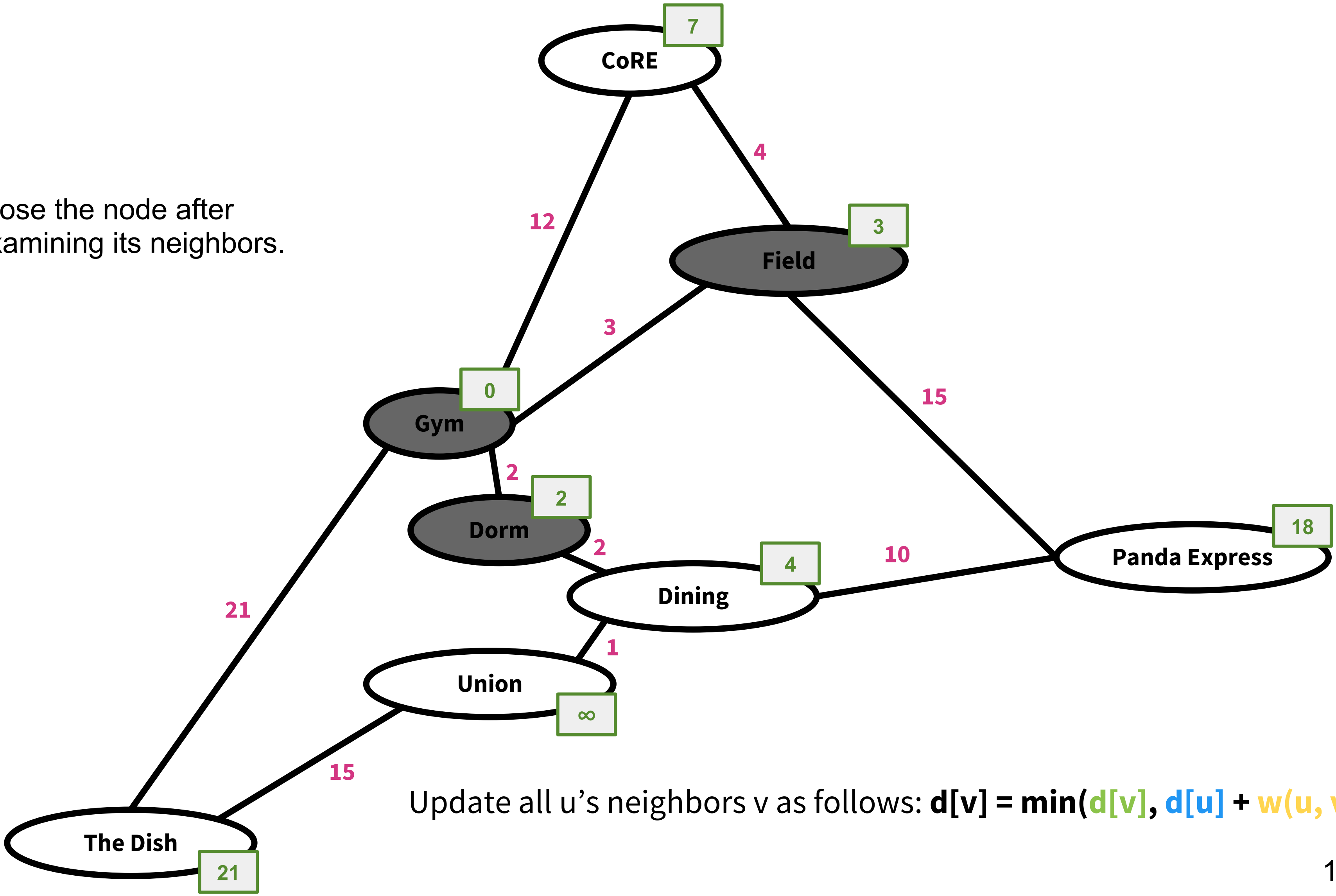
Update its open neighbors



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

Dijkstra's Algorithm

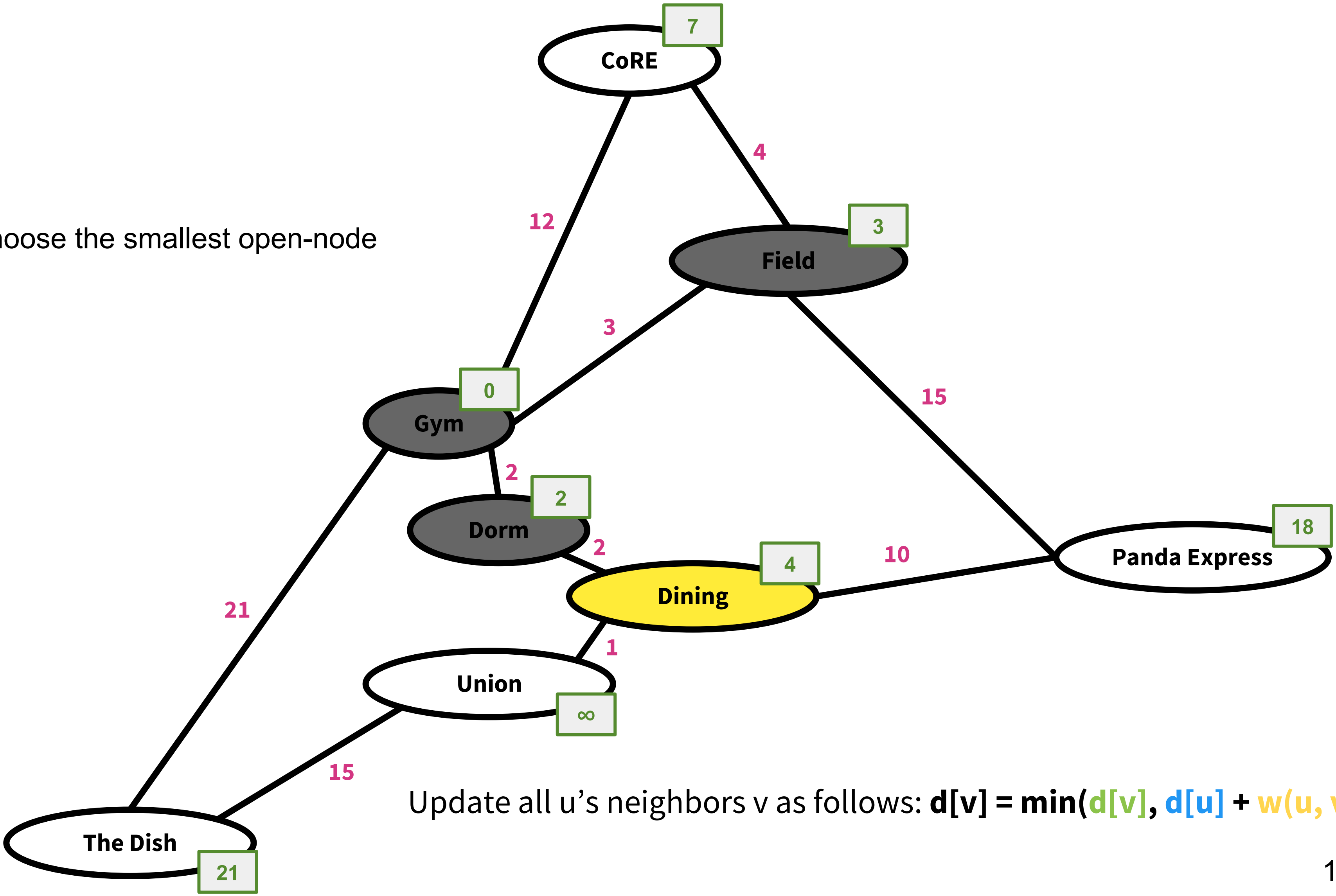
Close the node after examining its neighbors.



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

Dijkstra's Algorithm

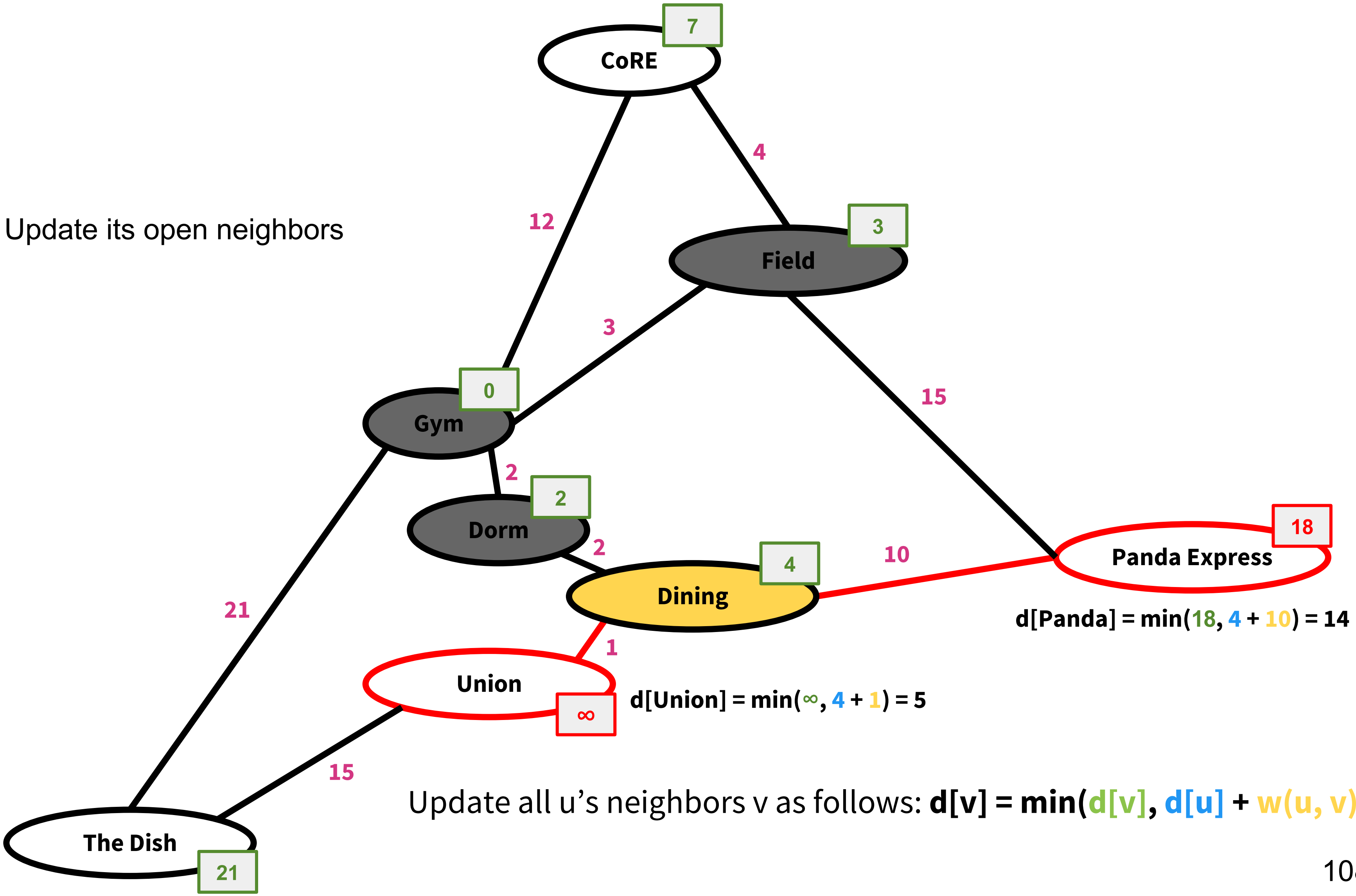
Choose the smallest open-node



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

Dijkstra's Algorithm

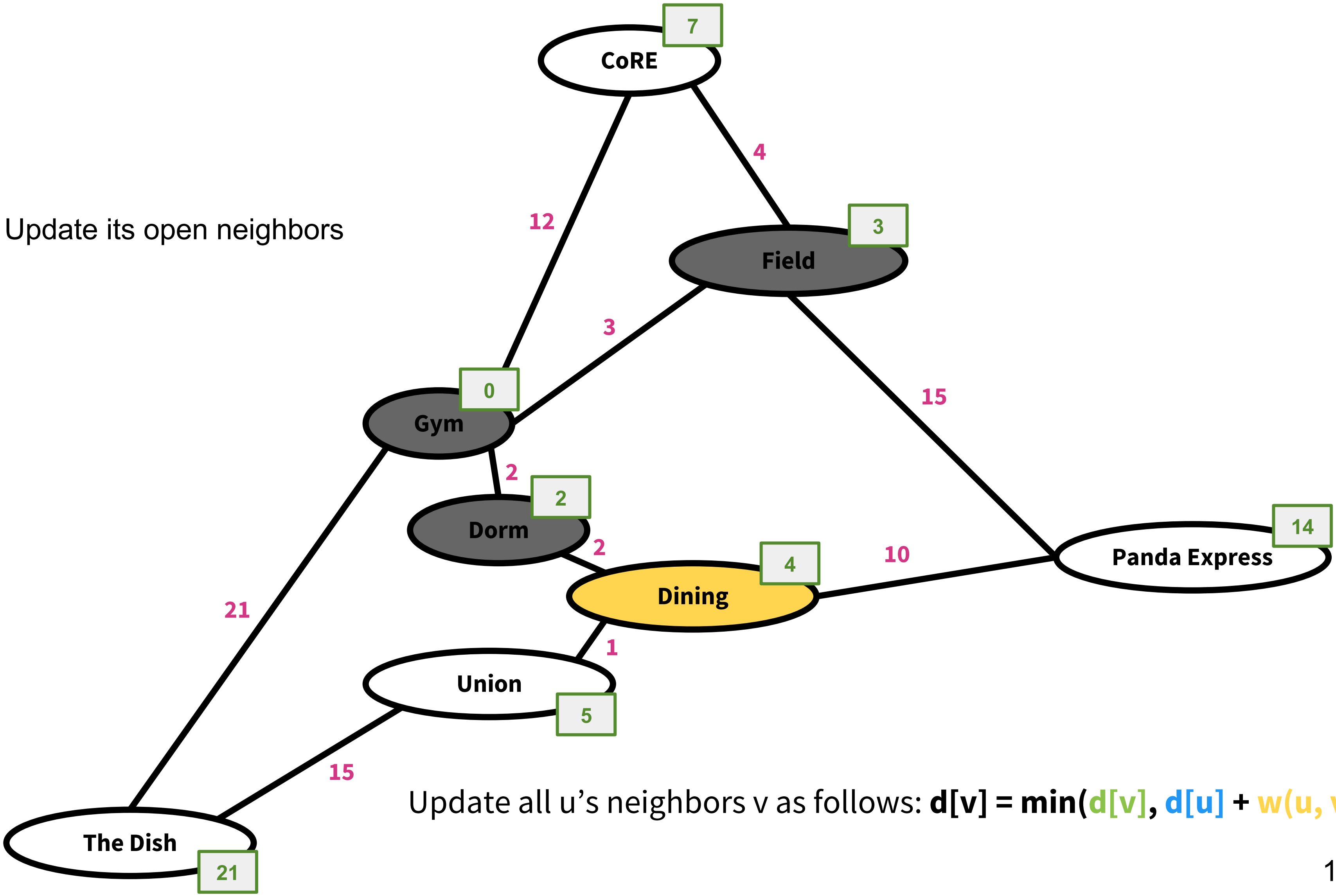
Update its open neighbors



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

Dijkstra's Algorithm

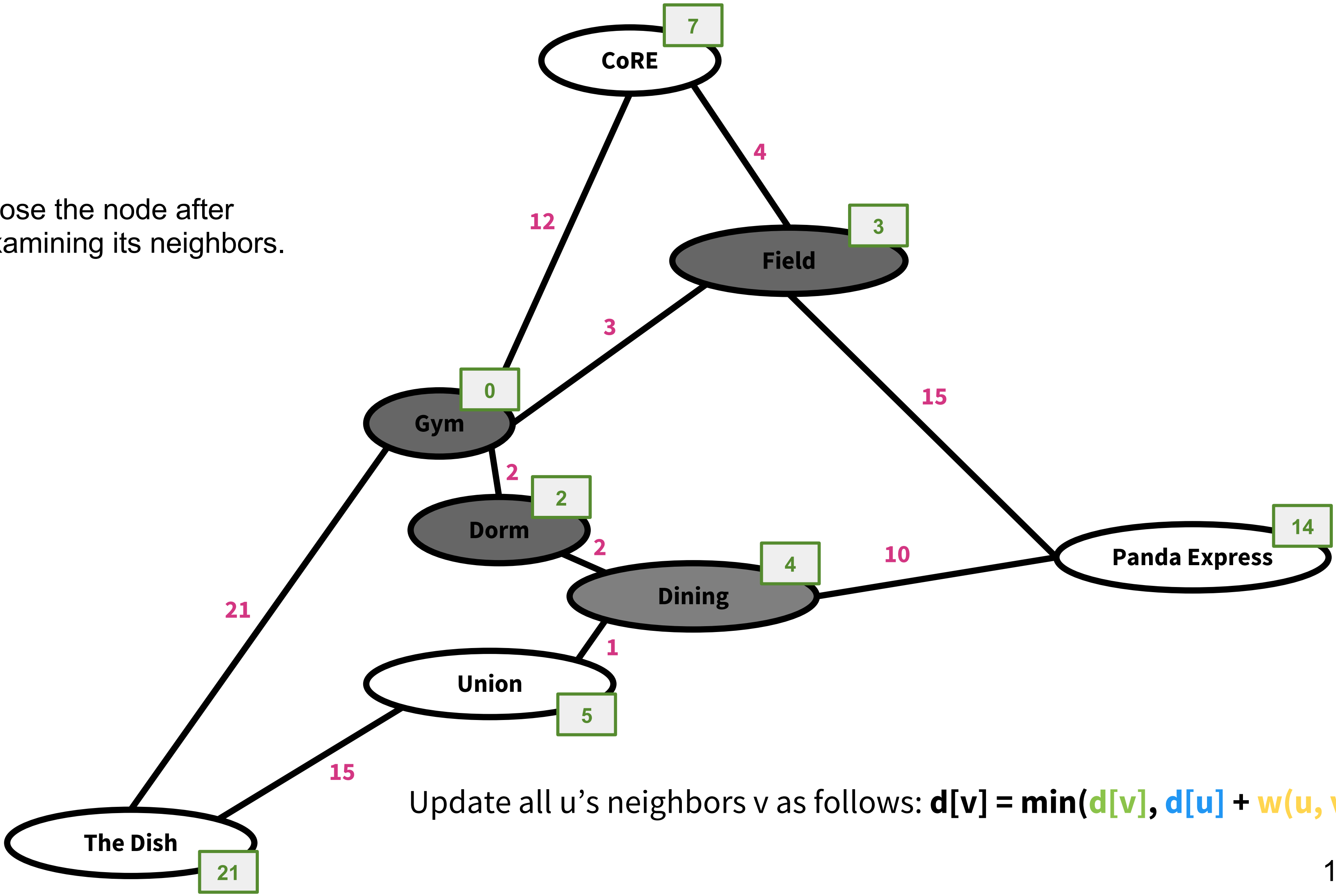
Update its open neighbors



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

Dijkstra's Algorithm

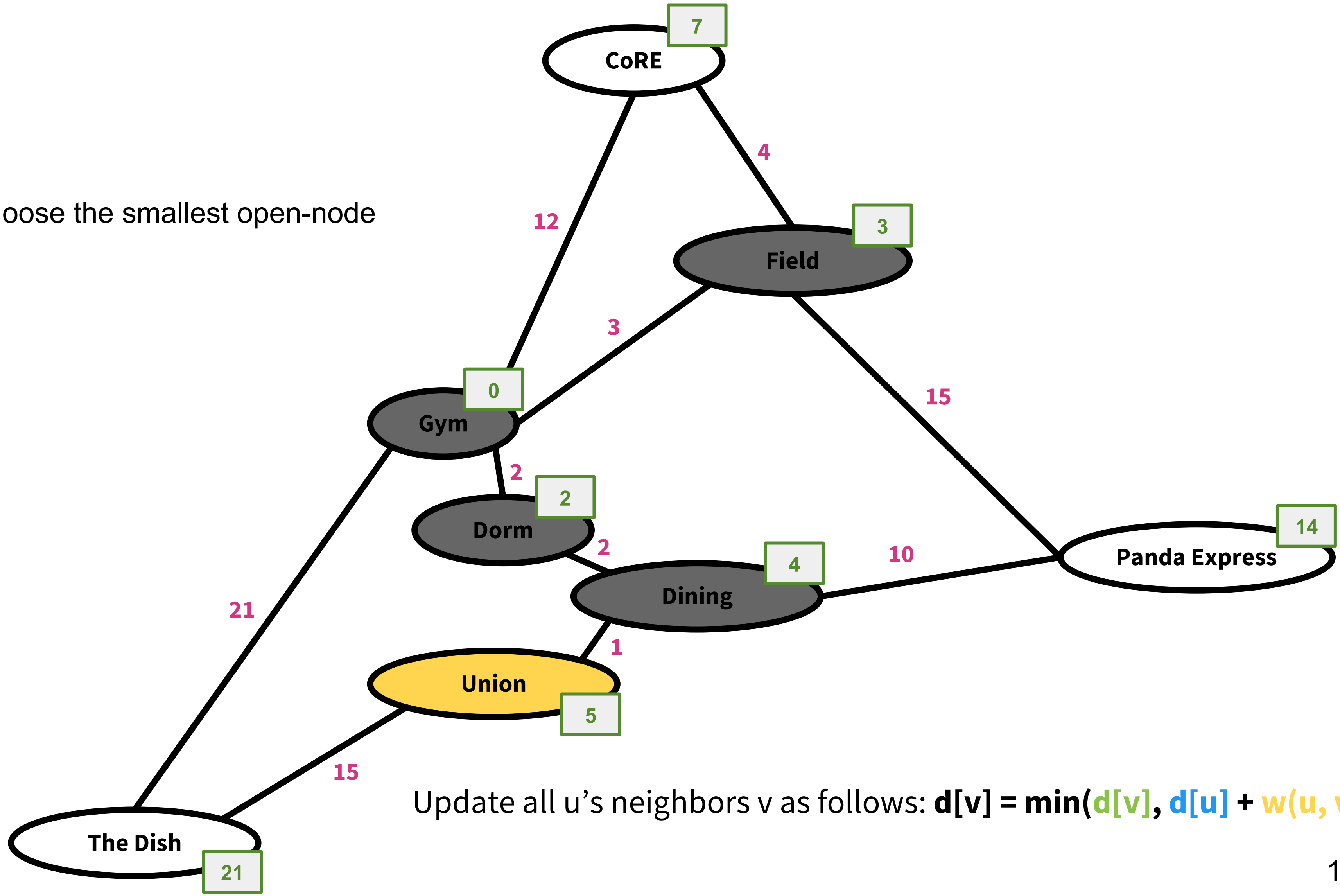
Close the node after examining its neighbors.



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

Dijkstra's Algorithm

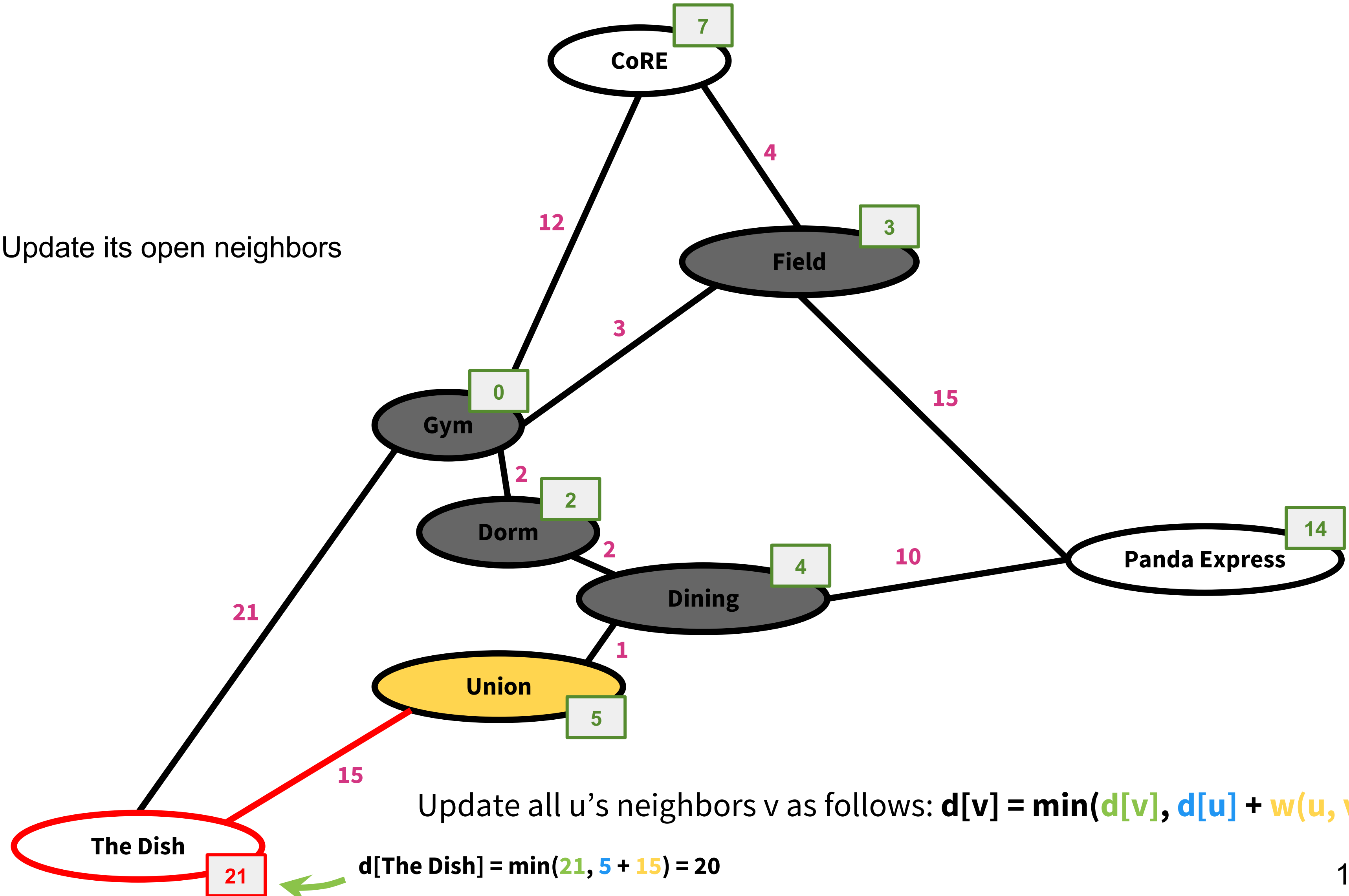
Choose the smallest open-node



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

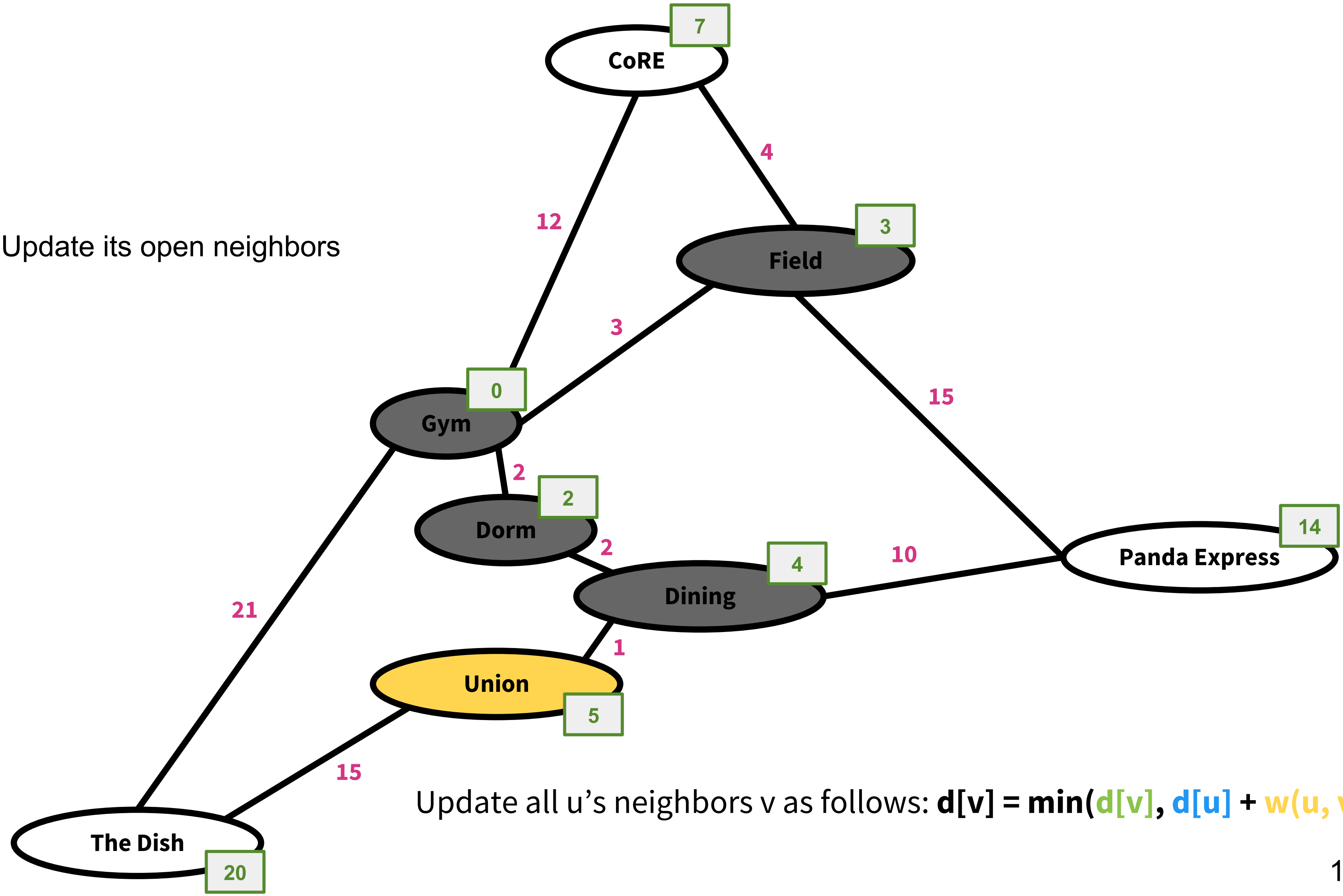


Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

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

Dijkstra’s Algorithm

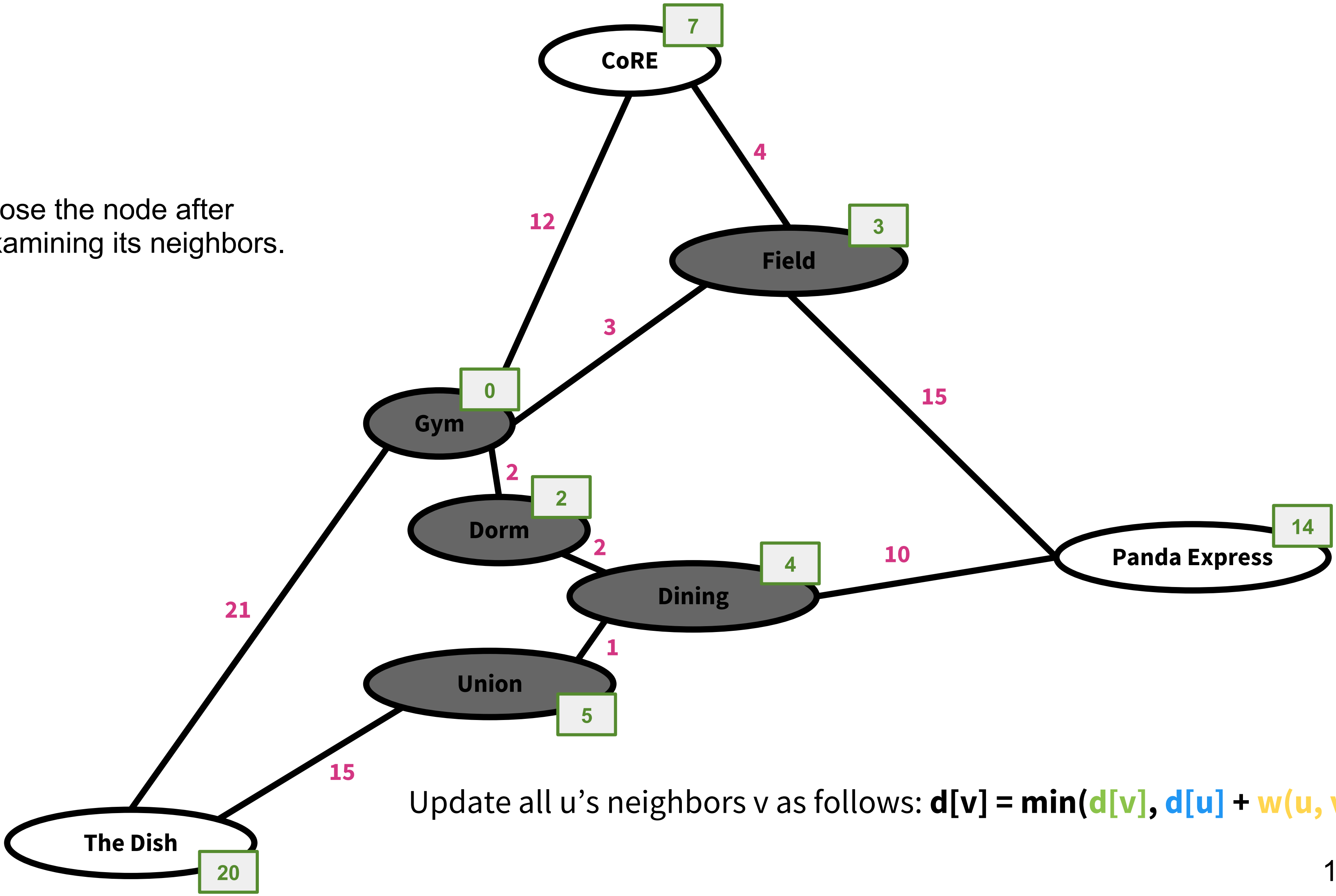
Update its open neighbors



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.



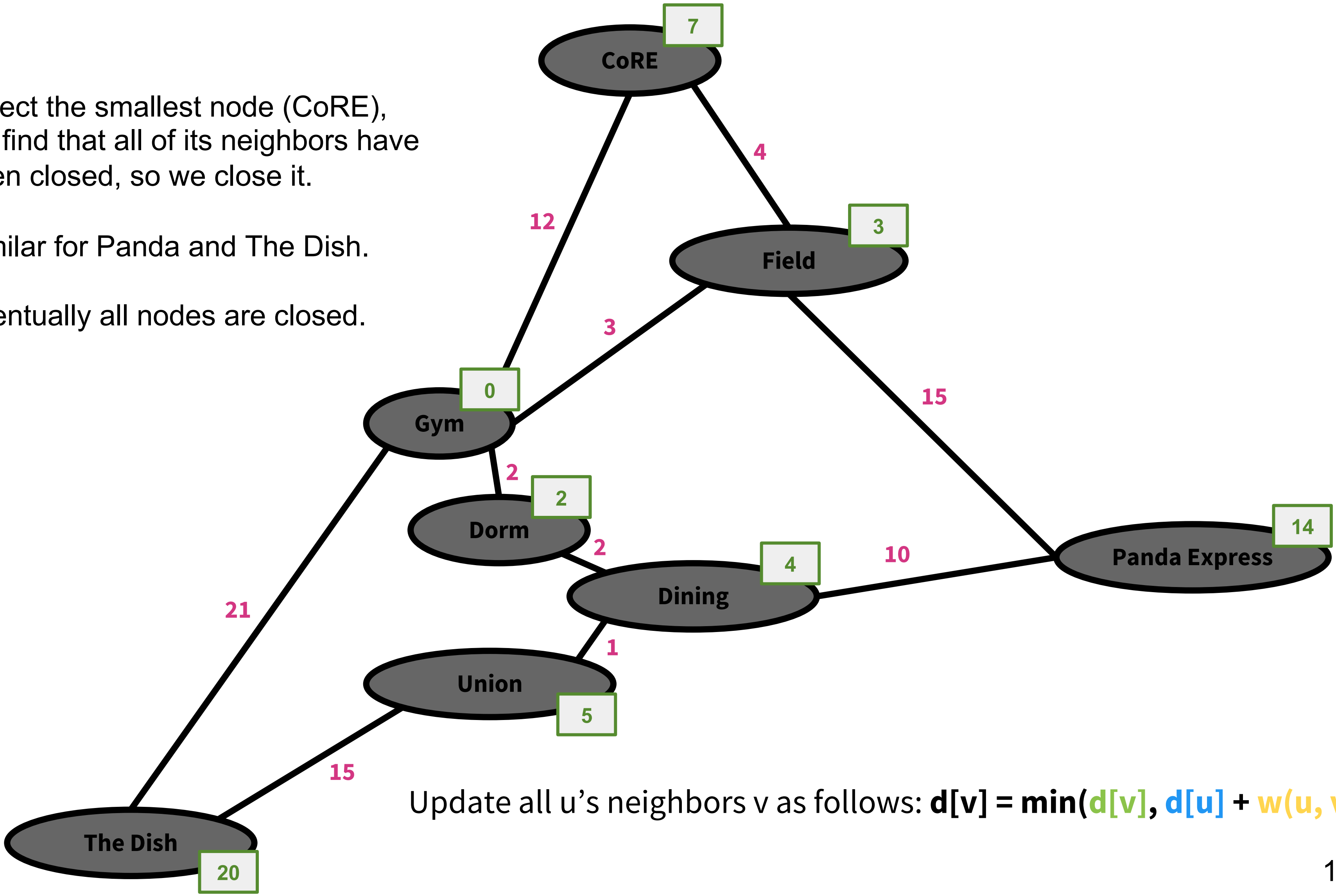
Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

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] ≥ 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 $d[x] \neq d(s, x)$.

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

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 $d[z] = d(s, z) \leq d(s, x) < d[x]$.

Weights are non-negative.

Claim 1 implies $d(s, x) \leq d[x]$ and we assumed that $d[x] \neq d(s, x)$.

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.