Graph Algorithms I

Summer 2017 • Lecture 07/18

A Few Notes

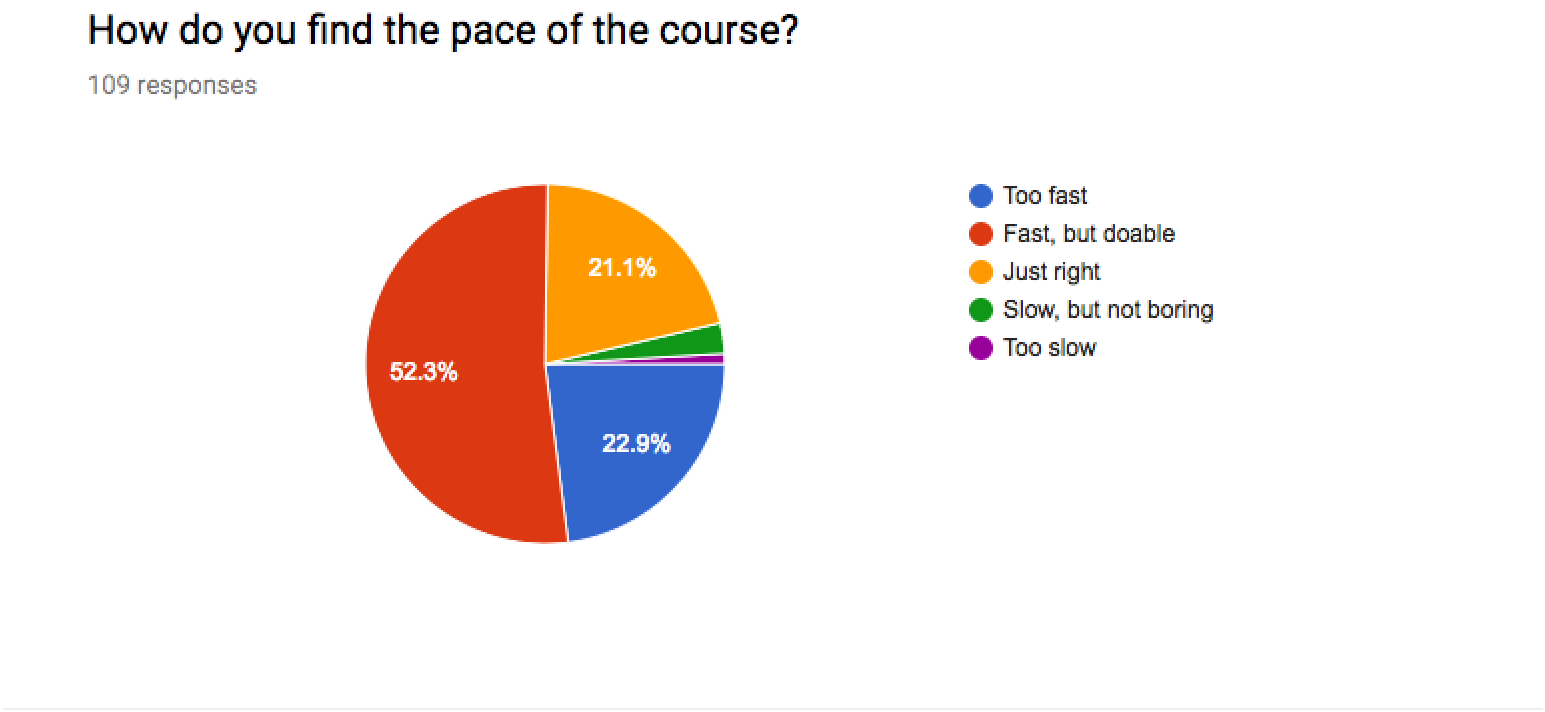
Homework 3

Due Friday 7/21 at 11:59 p.m. on Gradescope.

Homework 4

Released Friday 7/21.

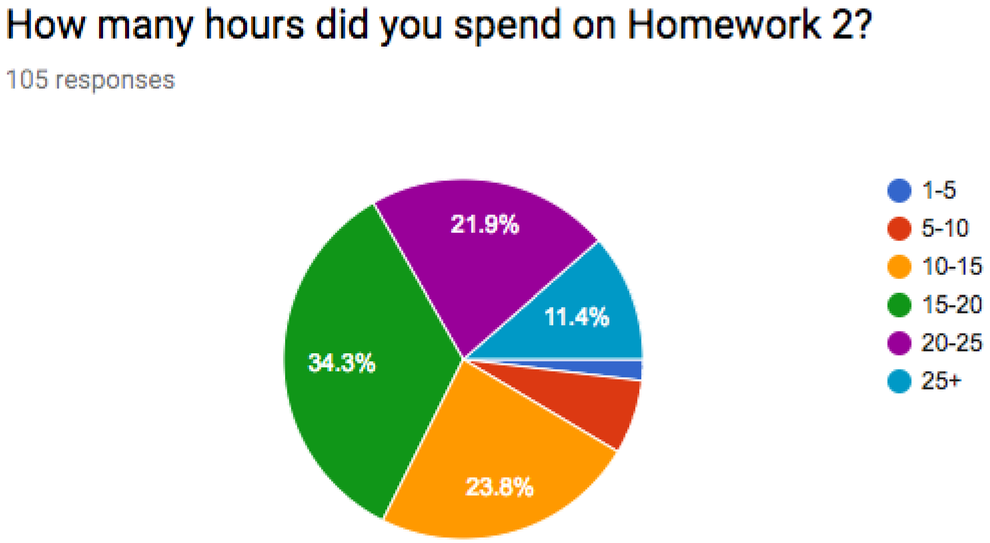
Week 3 Feedback



How do you find the pace of the course?

109 responses

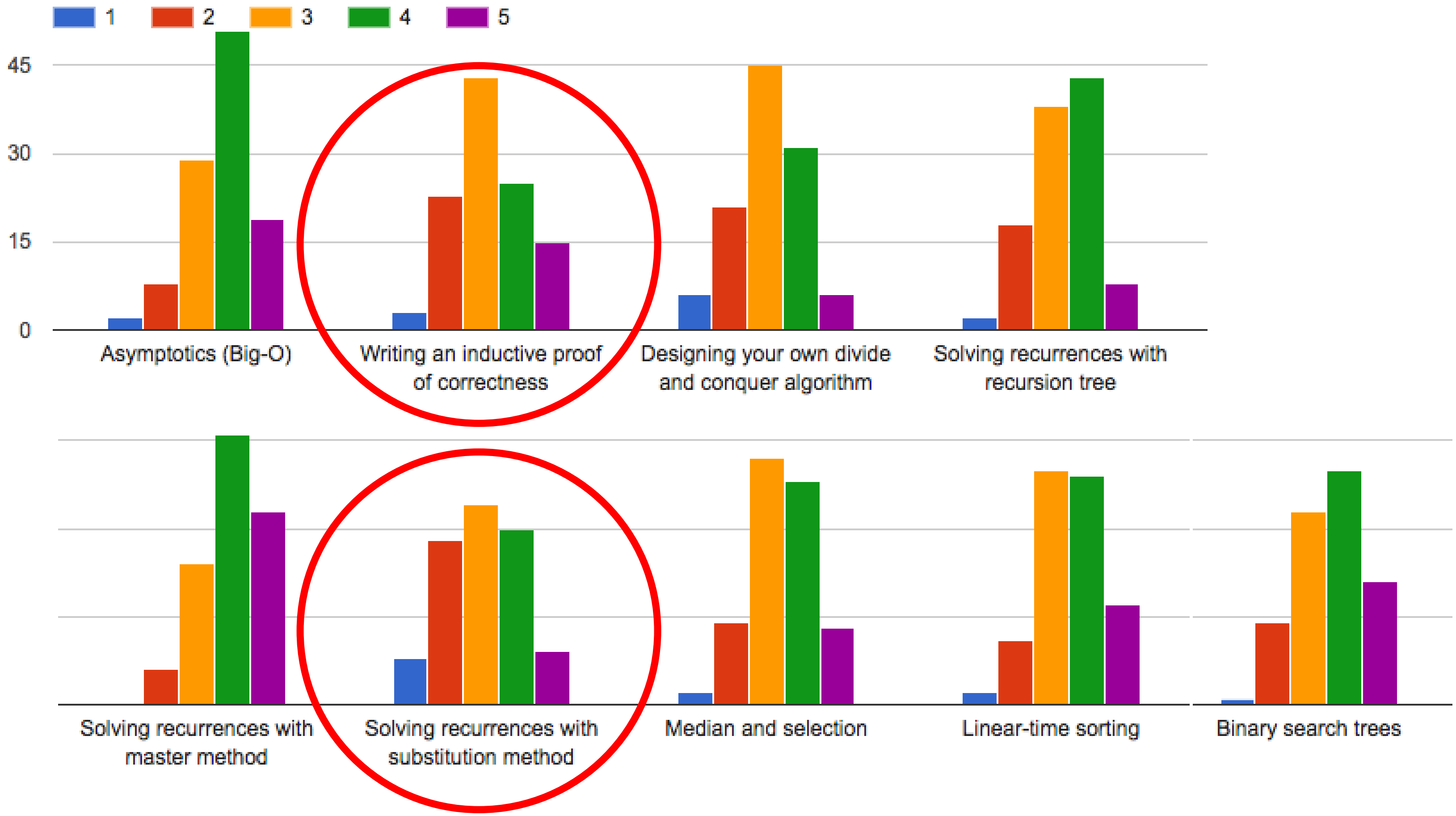
Week 3 Feedback



How many hours did you spend on Homework 2?

105 responses

Week 3 Feedback



Week 3 Feedback

What's one thing that you like about the course so far?

Office hours are very helpful. **Far fewer issues with office hours last week.**

I am actually learning things! :O **Yay!**

I like radix sort. **Me too!**

What's one thing that you wish was different about the course so far?

More homework-related examples during lecture. **Starting yesterday, I converted half of my office hours into a discussion section.**

Homework too long. **Starting this week, we'll ask 5 questions.**

Point out the reference material that can help with the homework. **Lecture notes from previous quarters and CLRS chapters have been on the website for the past two weeks.**

More link to application of these algorithm. **Starting this week, the homework will be more applied and motivated.**

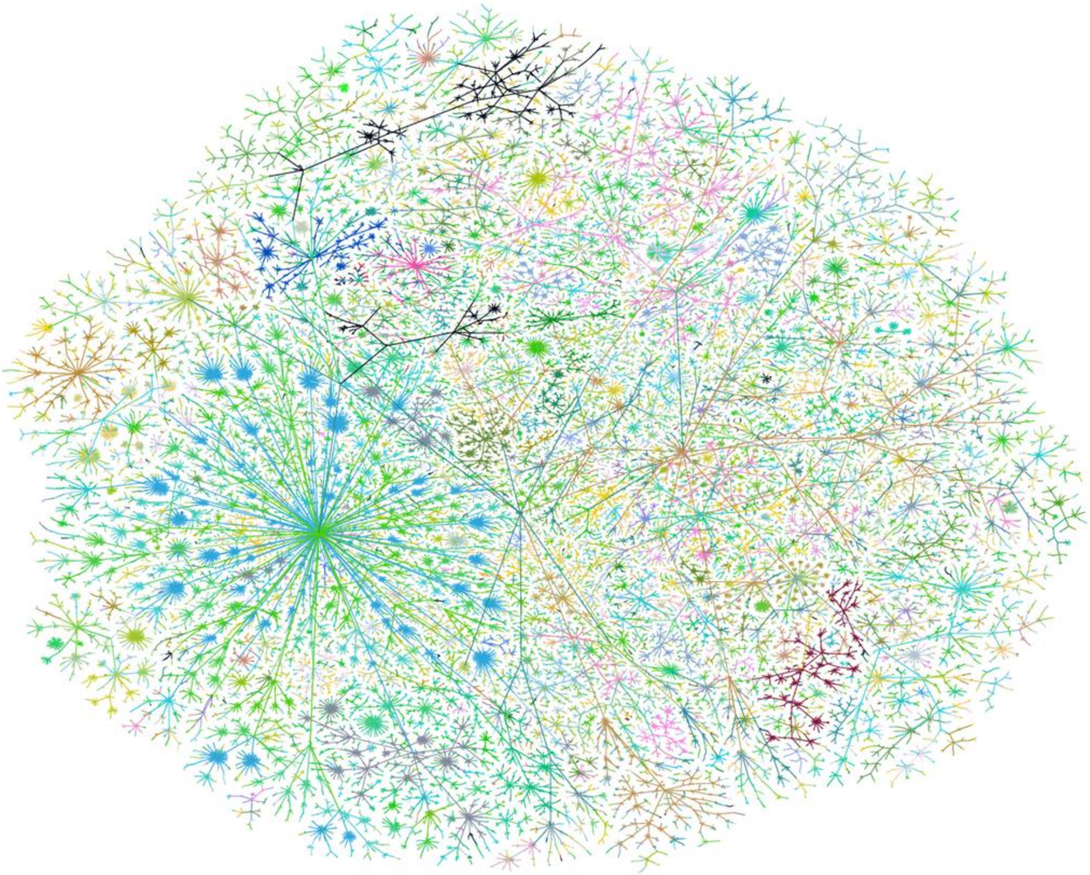
Outline for Today

- 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

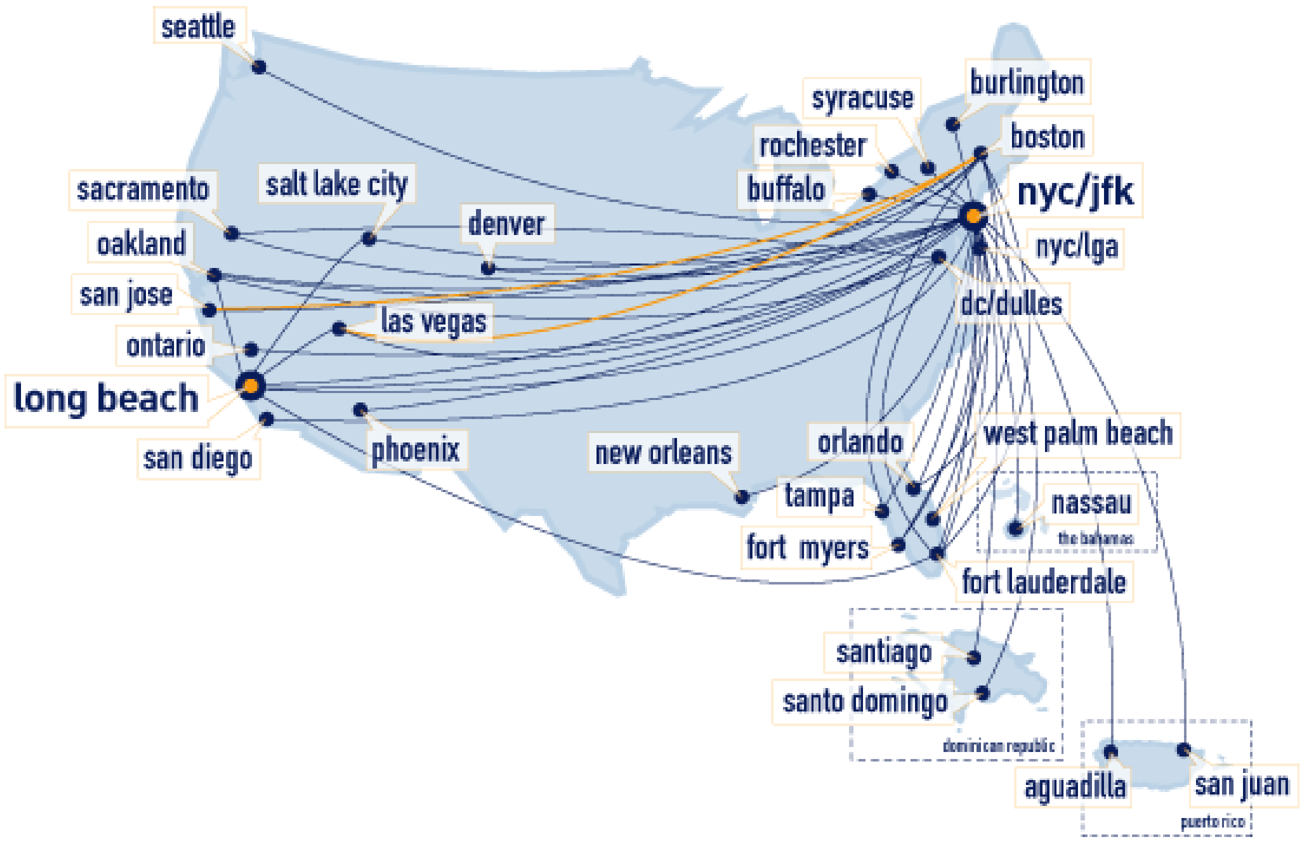
Examples of Graphs

The Internet (circa 1999)



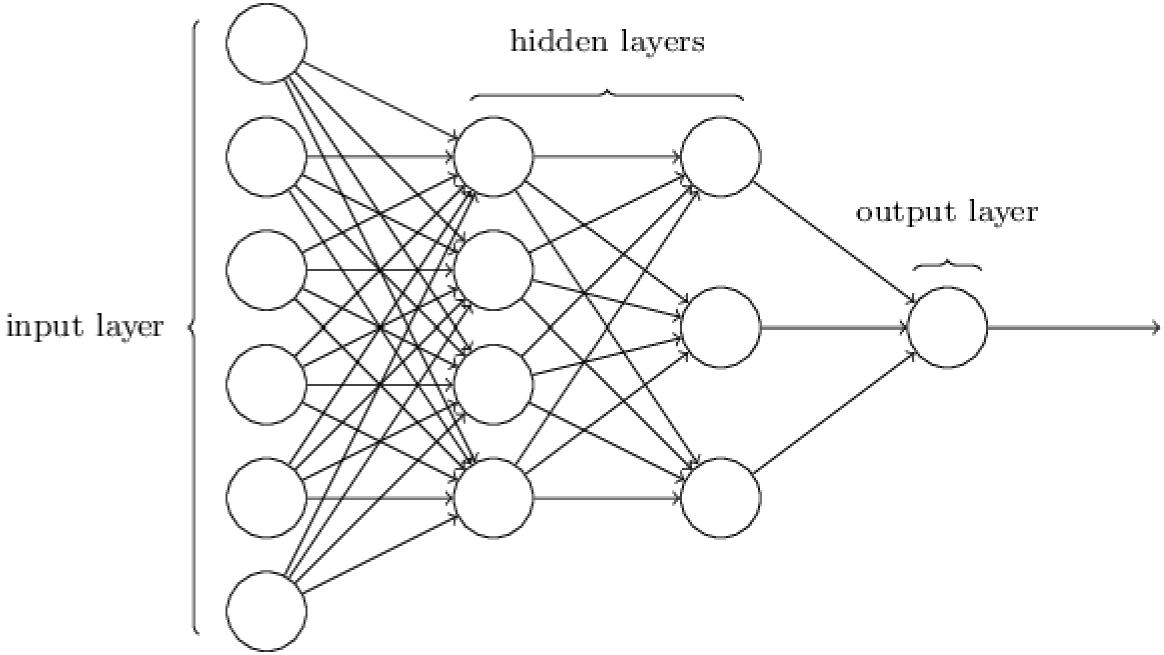
Examples of Graphs

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 (SPSP) for efficient routing.

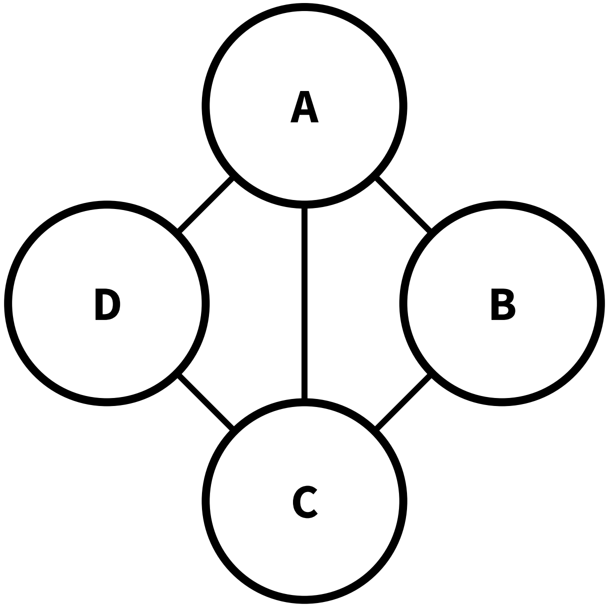
Finding strongly connected components for community detection or clustering.

Finding the topological ordering to respect dependencies.

Undirected Graphs

An undirected graph has vertices and edges.

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



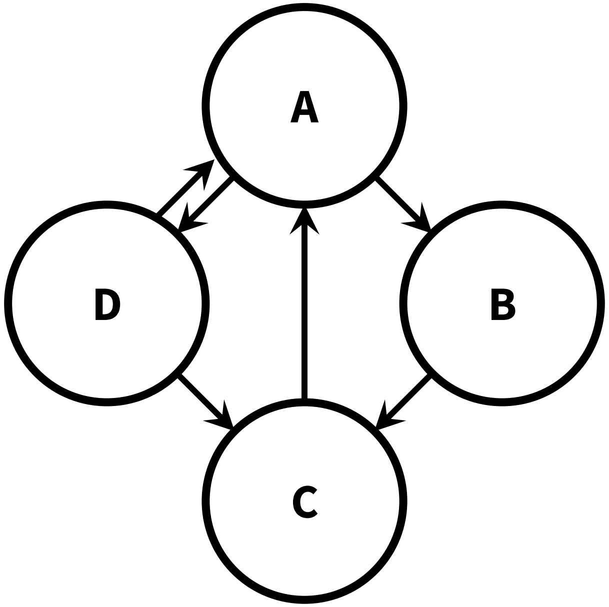
Directed Graphs

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

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

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

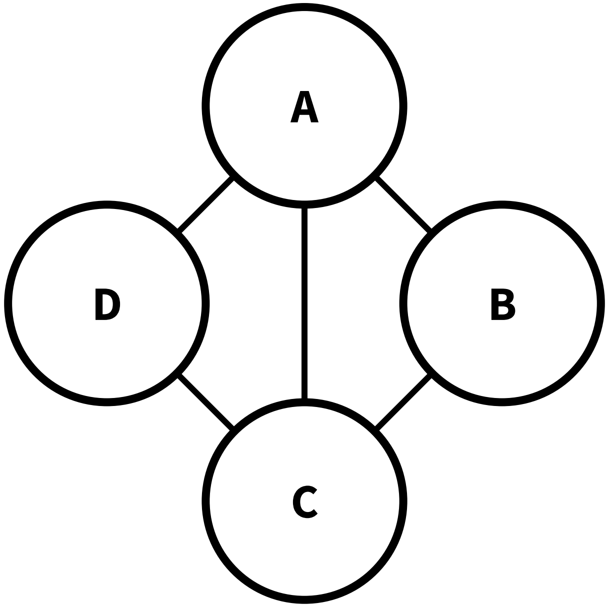
e.g. V = {A, B, C, D} and E = { [A, B], [A, D], [B, C], [C, A], [D, A], [D, C] }



Graph Representations

How do we represent graphs?

(1) Adjacency matrix

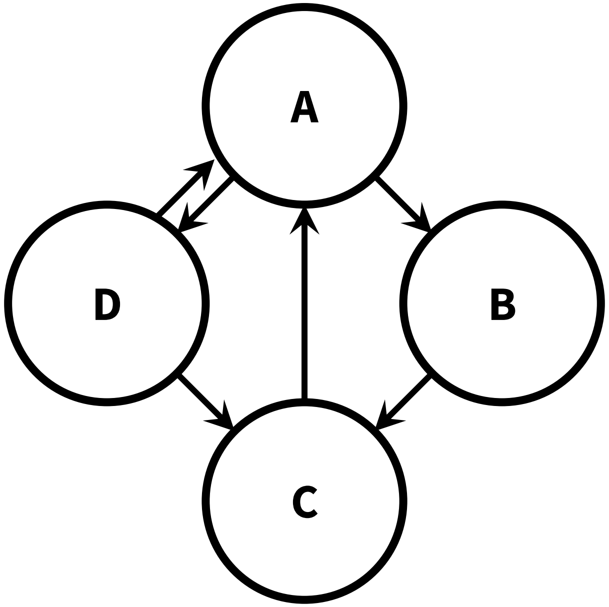


$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 1 & 0 & 1 & 0 \\ C & 1 & 1 & 0 & 1 \\ D & 1 & 0 & 1 & 0 \end{array}$$

Graph Representations

How do we represent graphs?

(1) Adjacency matrix



destination

source

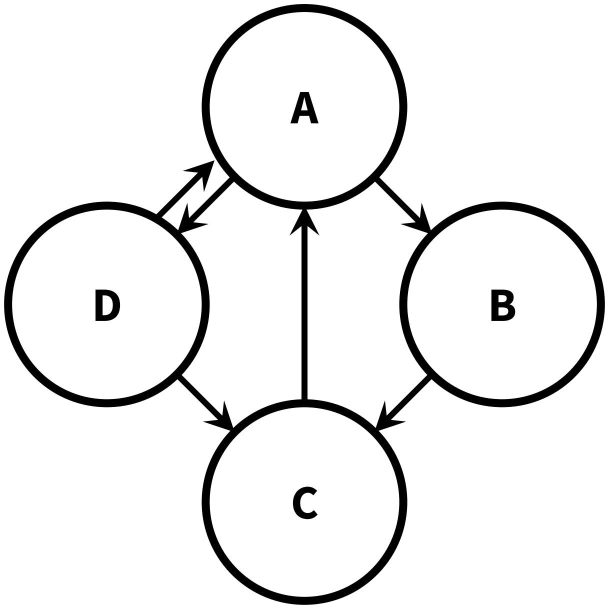
$$\begin{array}{cc} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

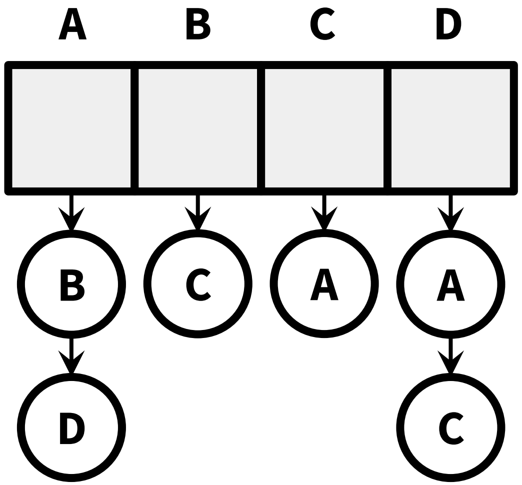
Graph Representations

How do we represent graphs?

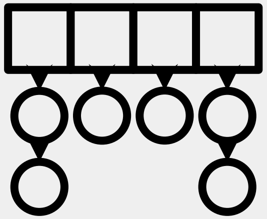
(1) Adjacency matrix

(2) Adjacency list





Graph Representations

For G = (V, E)	$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$	

Edge Membership
Is e = {u, v} in E?
	O(1)	O(deg(u))
		or
		O(deg(v))

Neighbor Query What are the neighbors of u?	O(|V|)	O(deg(v))
Space requirements	$O(\|V\|^2)$	O(|V|+|E|)

Generally, better for sparse graphs.

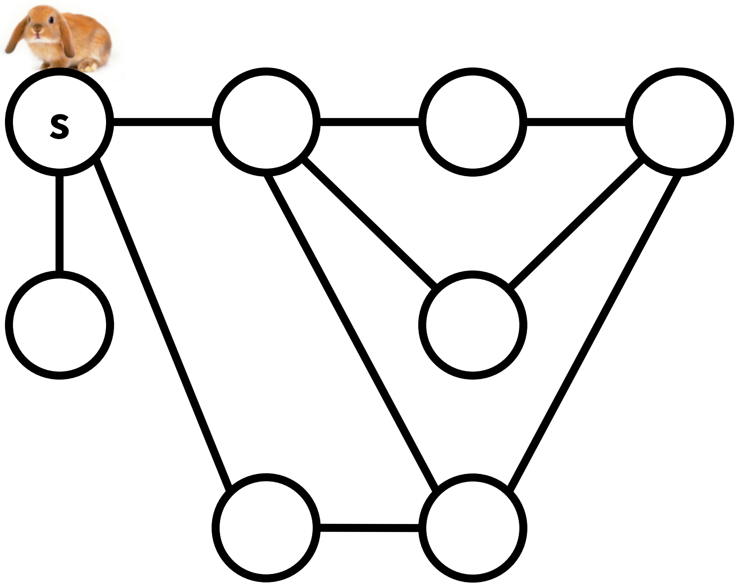
We'll assume this representation, unless otherwise stated.

Depth-First Search

Depth-First Search

An analogy

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



○ unvisited

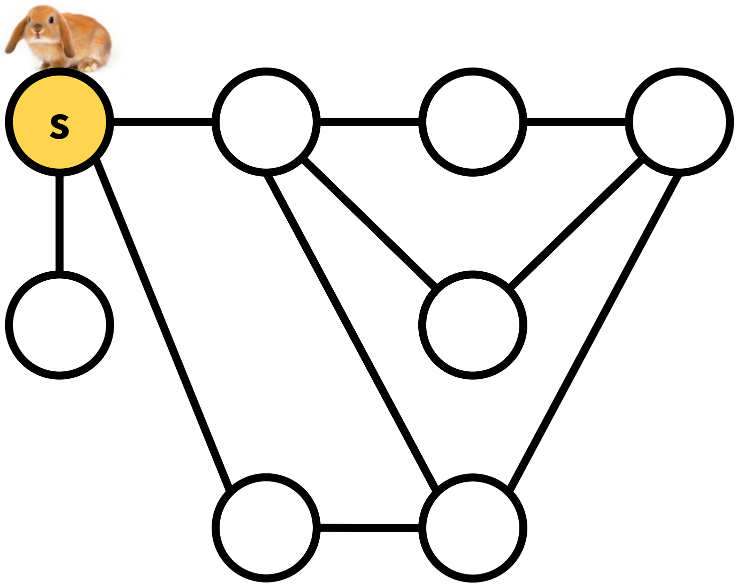
🟡 visited, but not fully explored

● 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).



○ unvisited

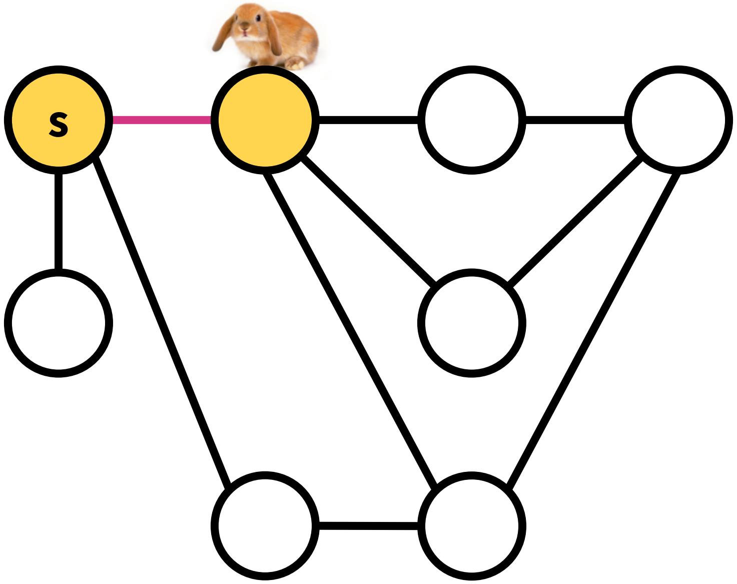
🟡 visited, but not fully explored

● 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).

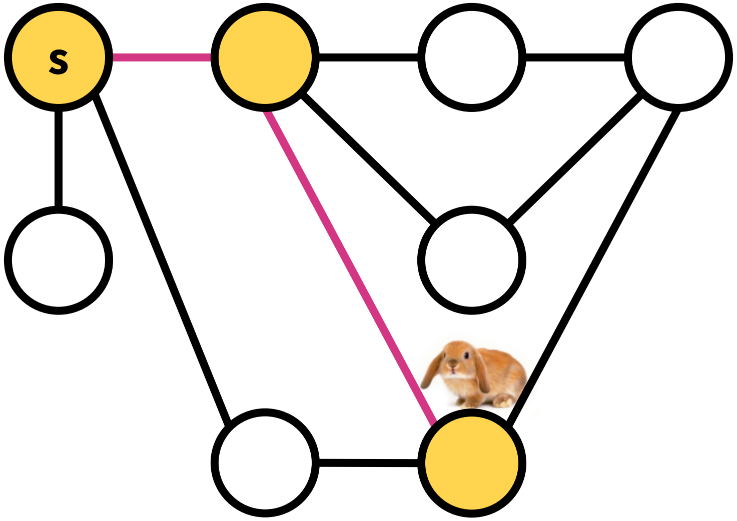


- ○ unvisited
- 🟡 visited, but not fully explored
- ● visited, and fully explored

Depth-First Search

An analogy

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

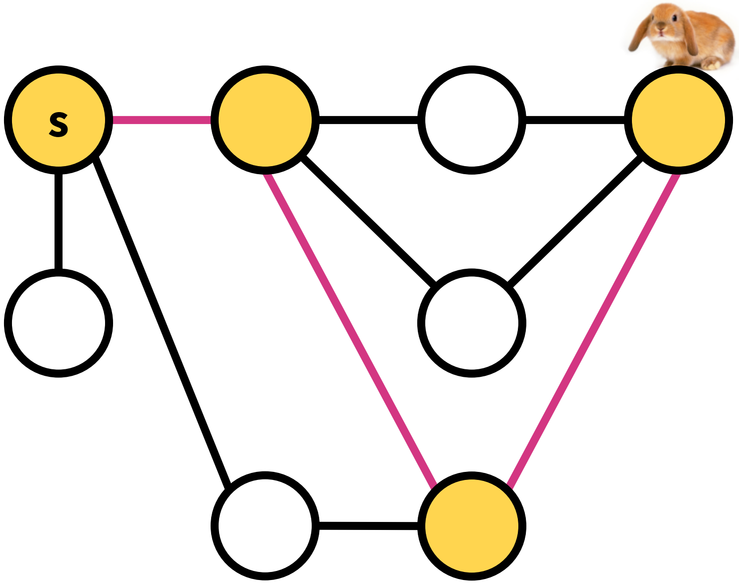


- ○ unvisited
- 🟡 visited, but not fully explored
- ● visited, and fully explored

Depth-First Search

An analogy

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



○ unvisited

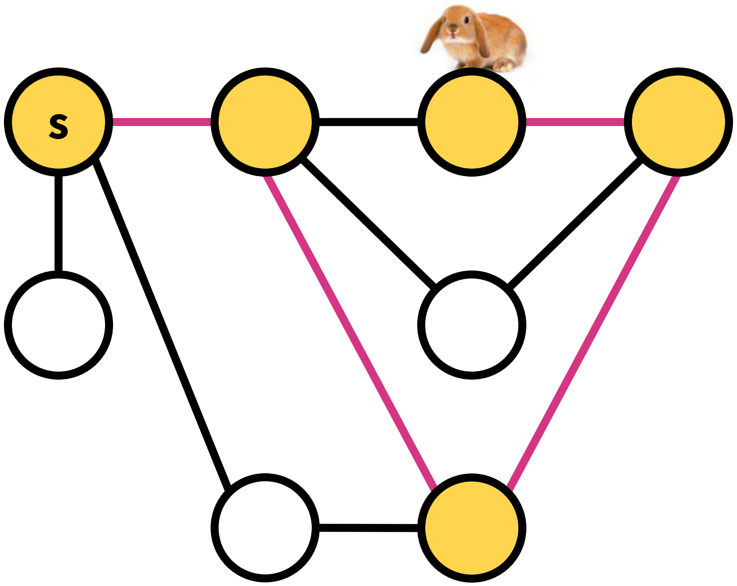
🟡 visited, but not fully explored

● 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).

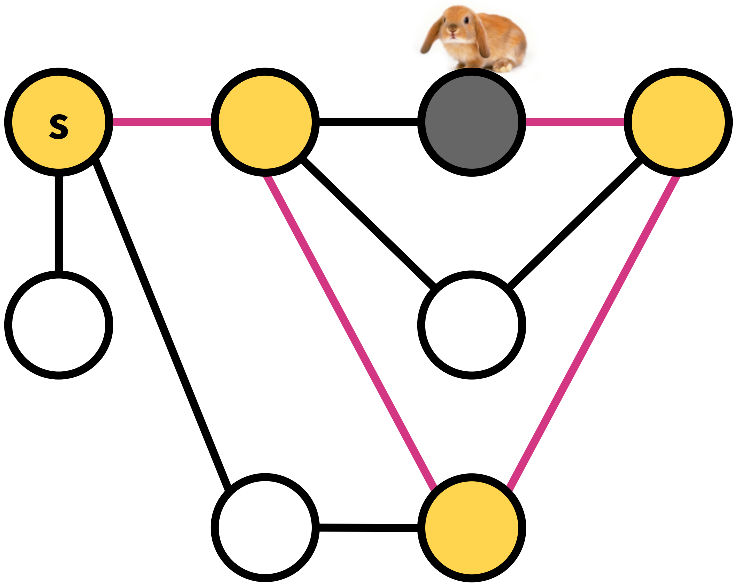


- ○ unvisited
- 🟡 visited, but not fully explored
- ⚫ visited, and fully explored

Depth-First Search

An analogy

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



○ unvisited

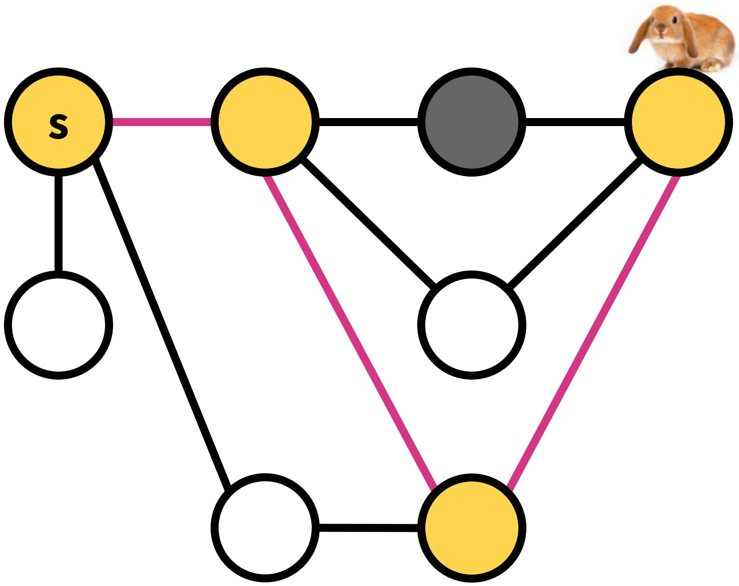
🟡 visited, but not fully explored

⚫ 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).



○ unvisited

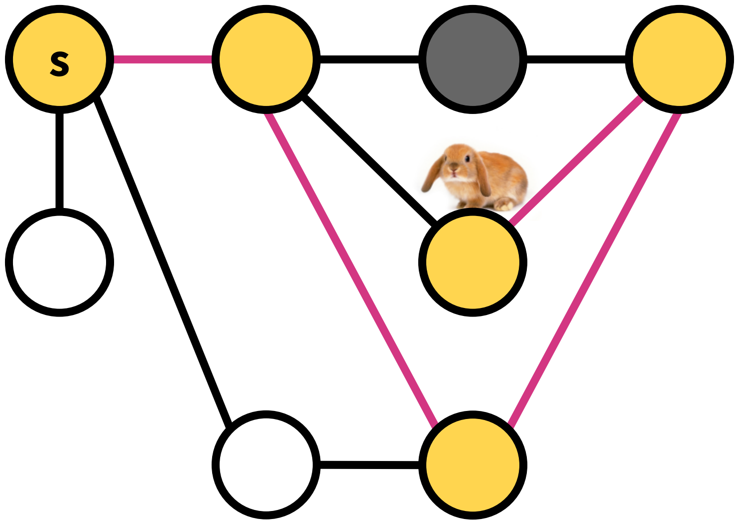
🟡 visited, but not fully explored

● 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).



○ unvisited

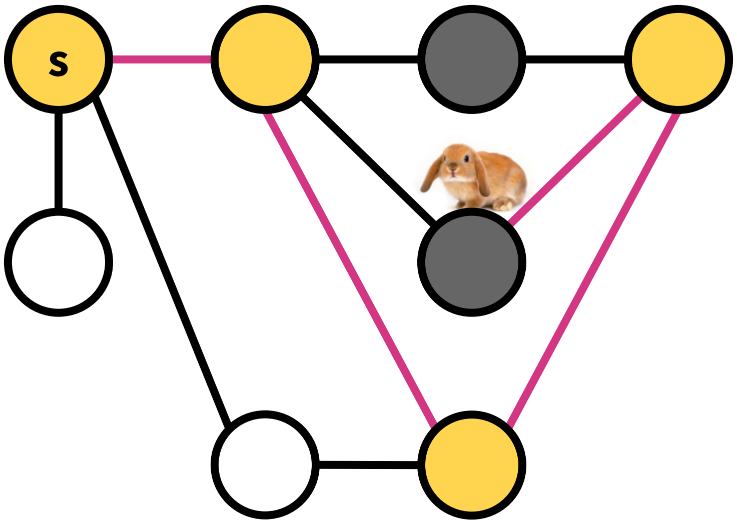
🟡 visited, but not fully explored

⚫ 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).



○ unvisited

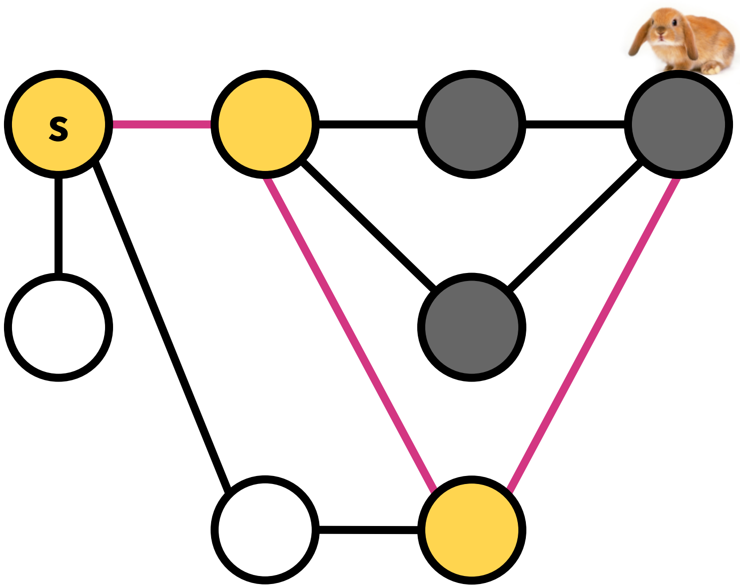
🟡 visited, but not fully explored

⚫ 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).

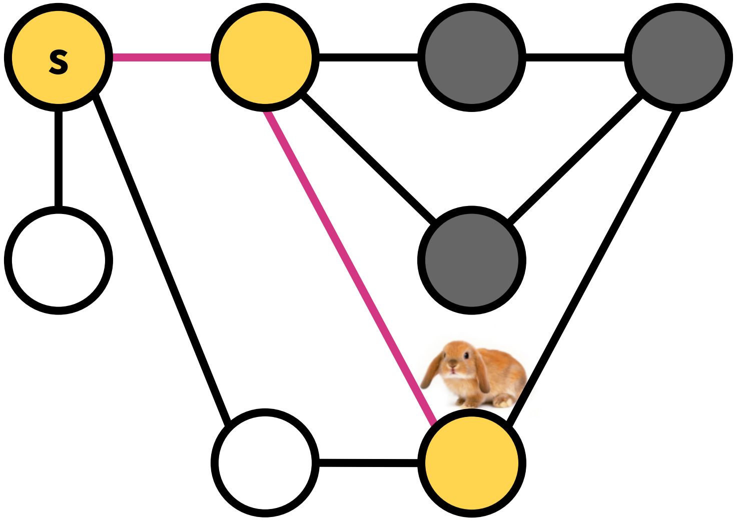


- ○ unvisited
- 🟡 visited, but not fully explored
- ● visited, and fully explored

Depth-First Search

An analogy

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



○ unvisited

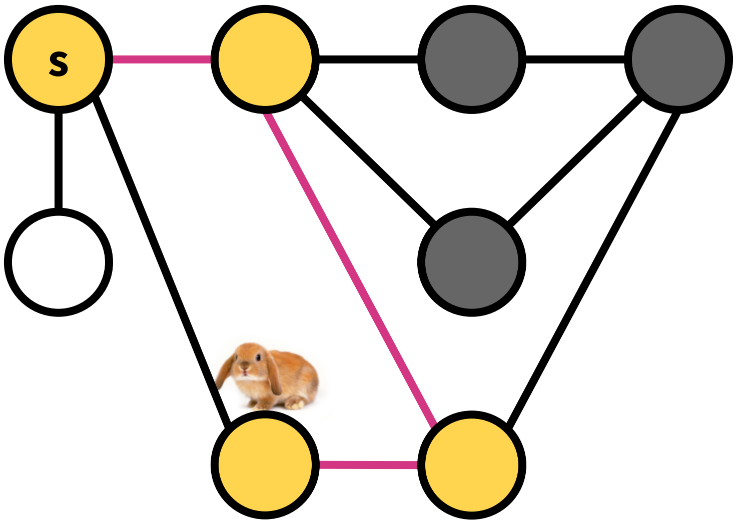
🟡 visited, but not fully explored

⚫ 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).

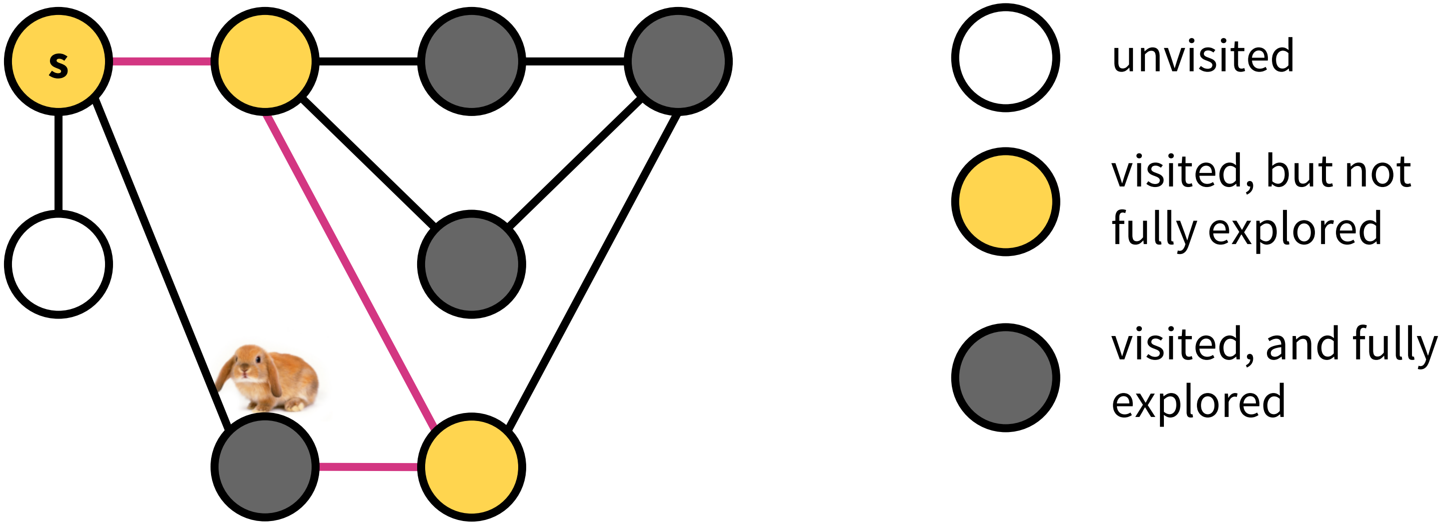


- ○ unvisited
- 🟡 visited, but not fully explored
- ⚫ visited, and fully explored

Depth-First Search

An analogy

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



○ unvisited

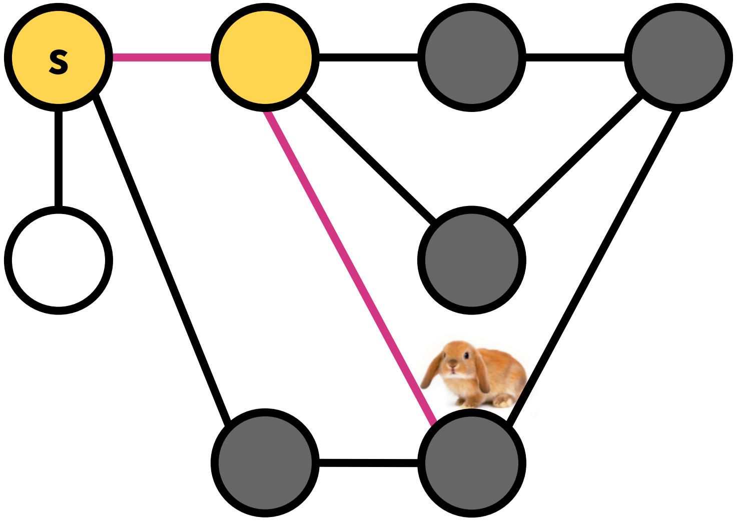
🟡 visited, but not fully explored

● 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).



○ unvisited

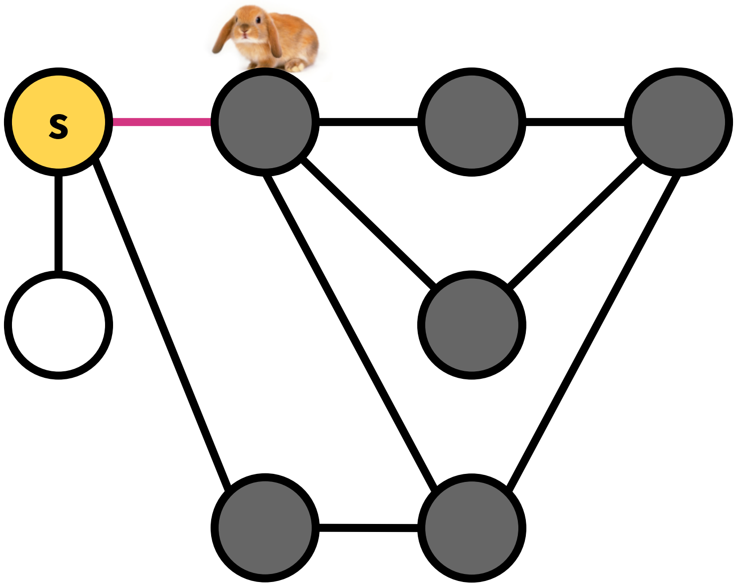
🟡 visited, but not fully explored

⚫ 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).



○ unvisited

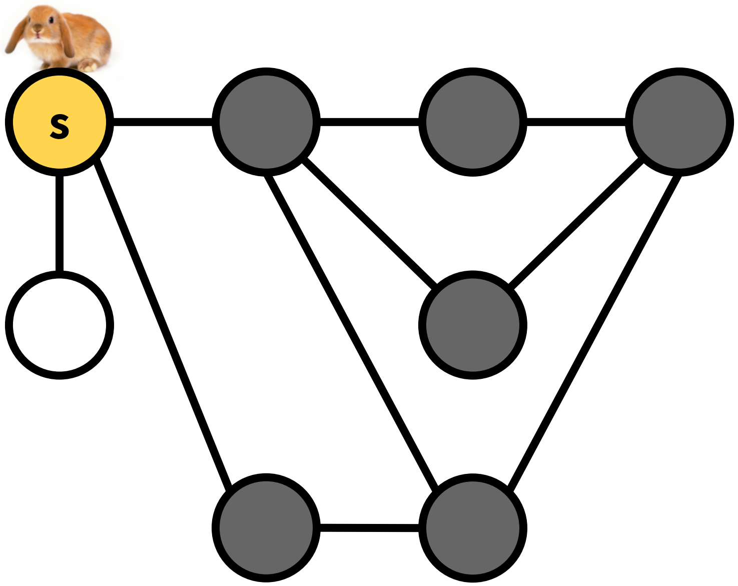
🟡 visited, but not fully explored

⚫ 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).



○ unvisited

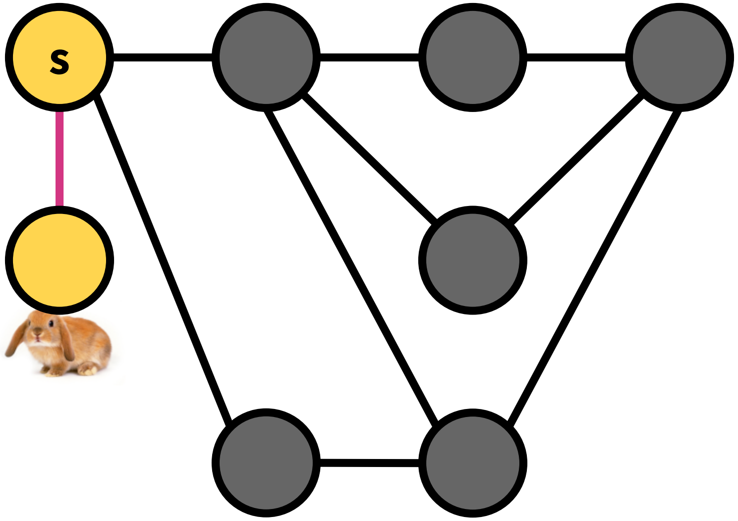
🟡 visited, but not fully explored

● 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).



○ unvisited

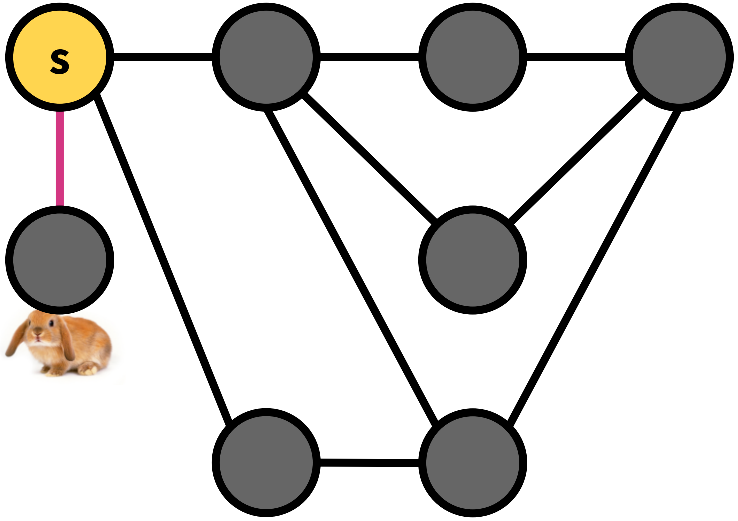
🟡 visited, but not fully explored

● 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).



○ unvisited

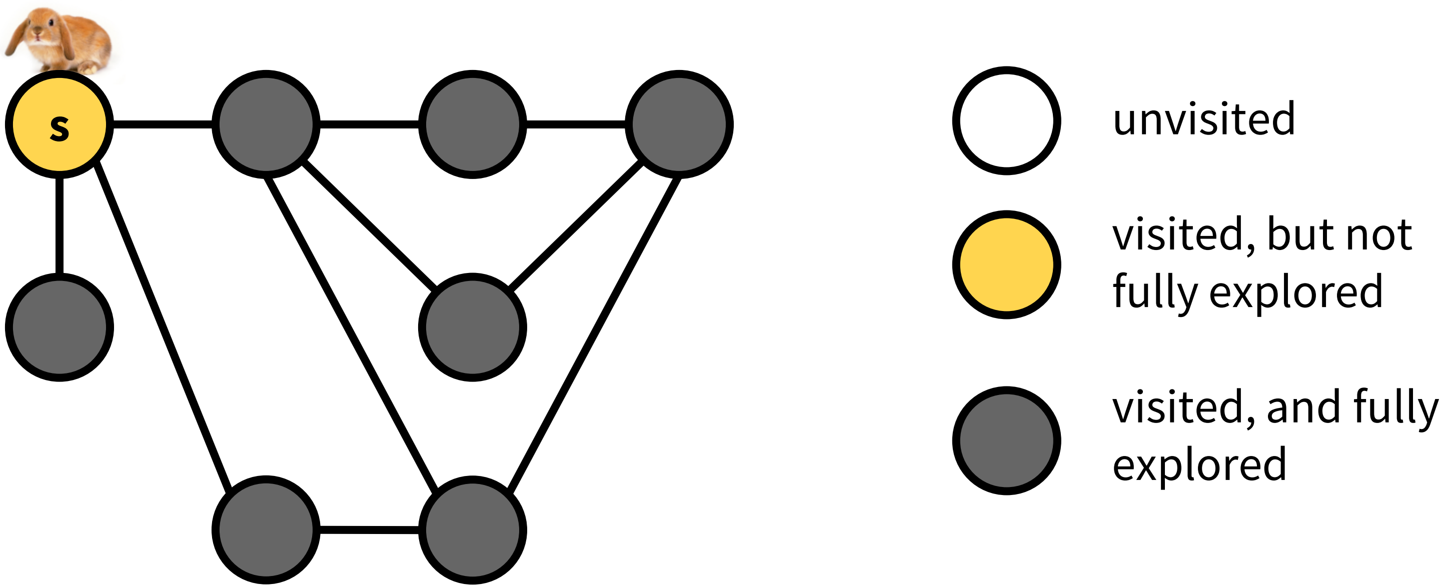
🟡 visited, but not fully explored

● 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).



unvisited

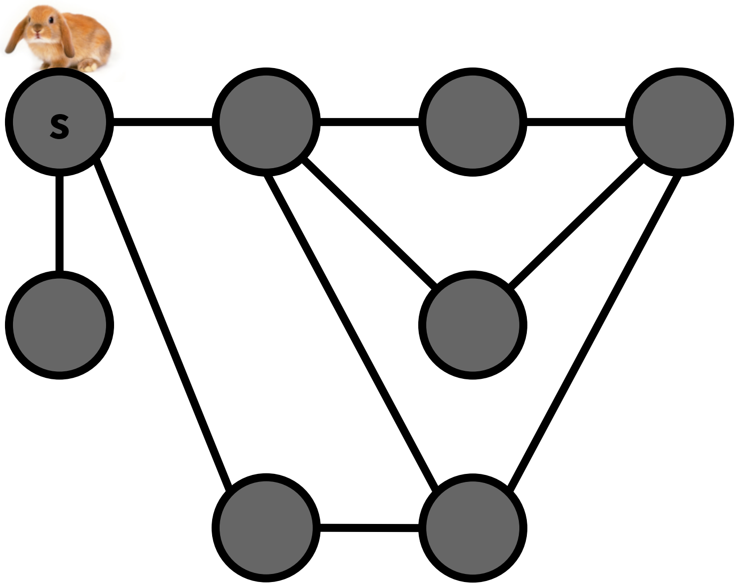
visited, but not fully explored

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).



○ unvisited

🟡 visited, but not fully explored

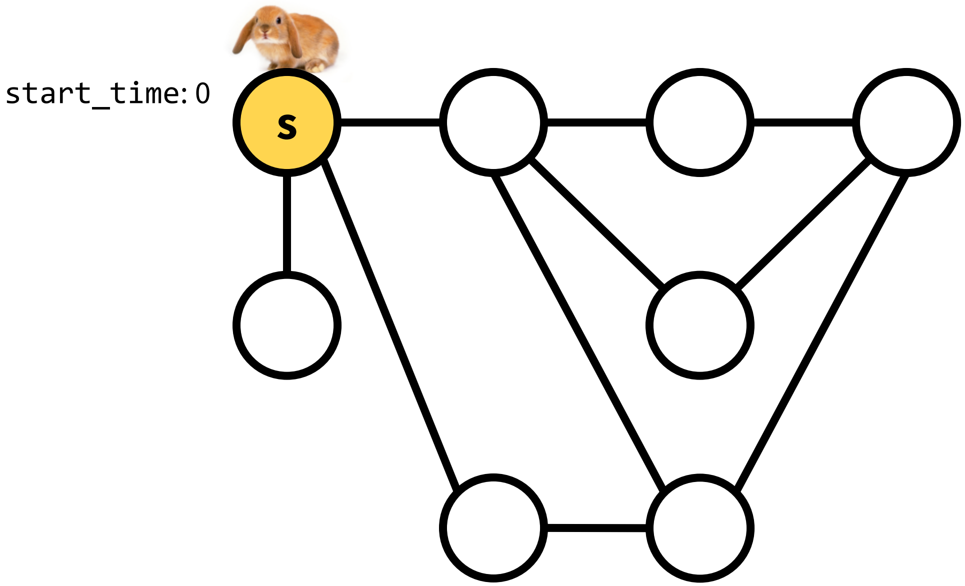
● visited, and fully explored

Depth-First Search

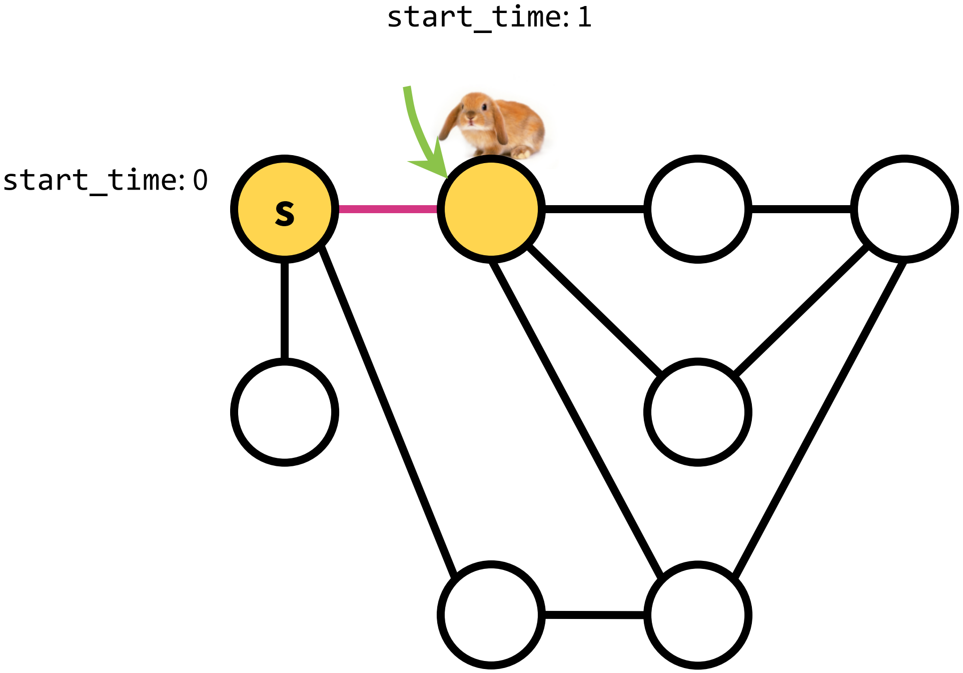
```
algorithm dfs(u, cur_time):
  u.start_time = cur_time
  cur_time += 1
  u.status = "in_progress" 🟡
  for v in u.neighbors:
    if v.status is "unvisited":
      cur_time = dfs(v, cur_time)
      cur_time += 1
  u.end_time = cur_time
  u.status = "done" ⚫
  return cur_time
```

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

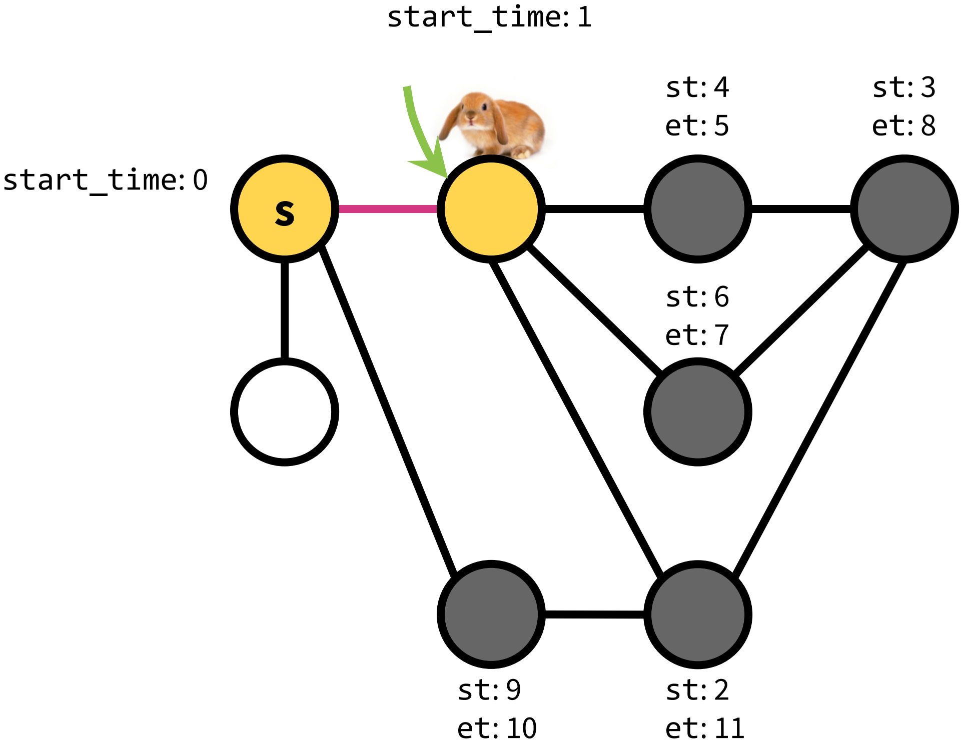
Depth-First Search



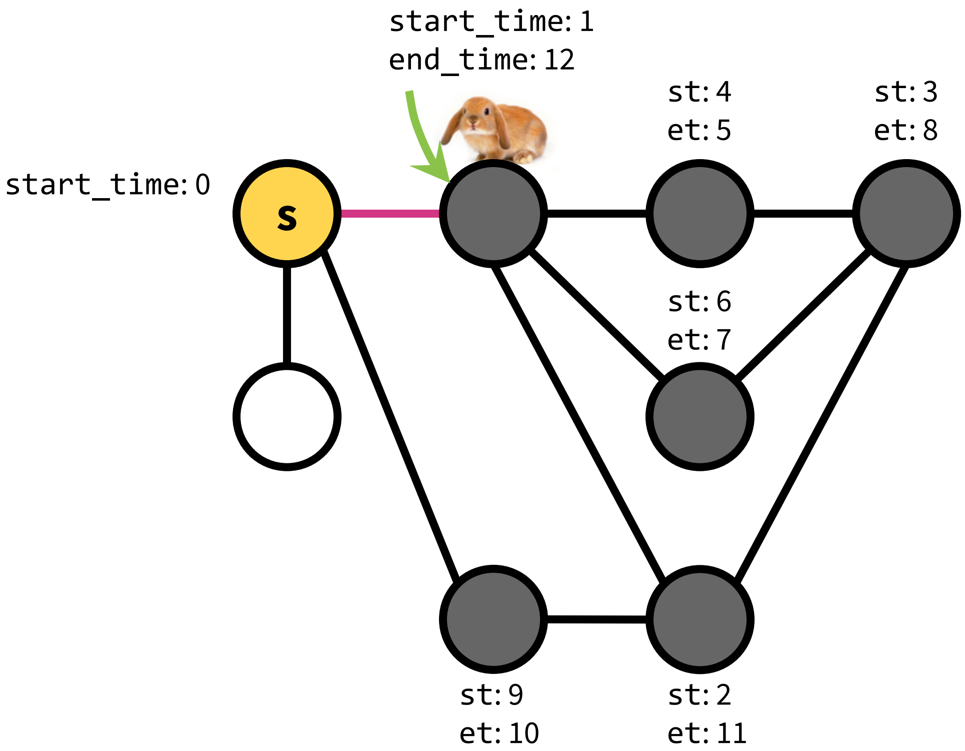
Depth-First Search



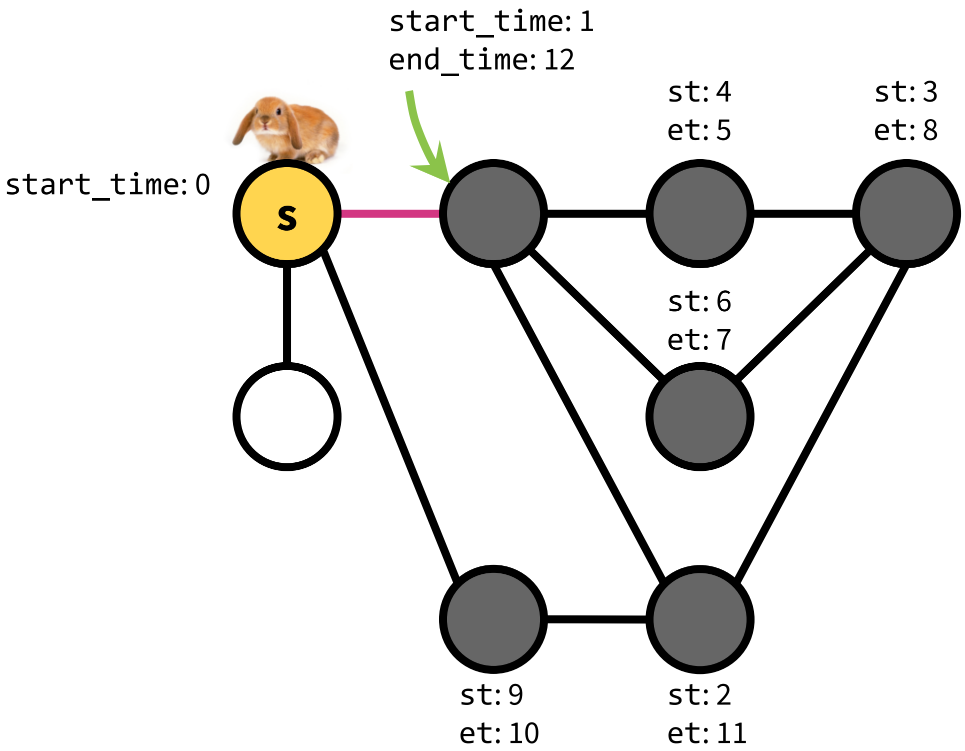
Depth-First Search



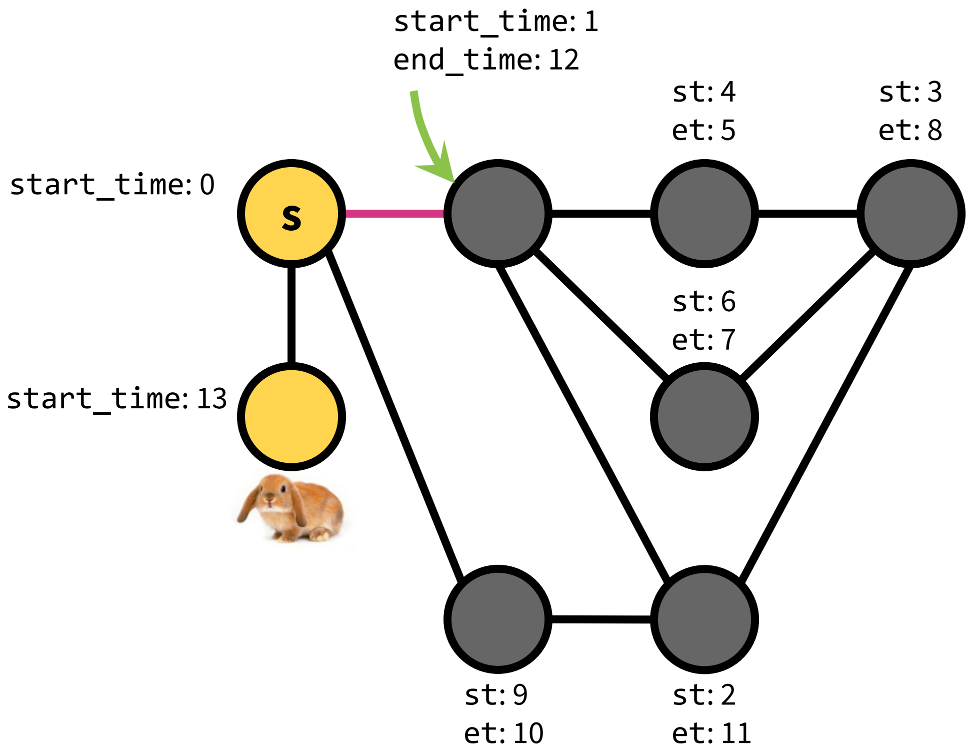
Depth-First Search



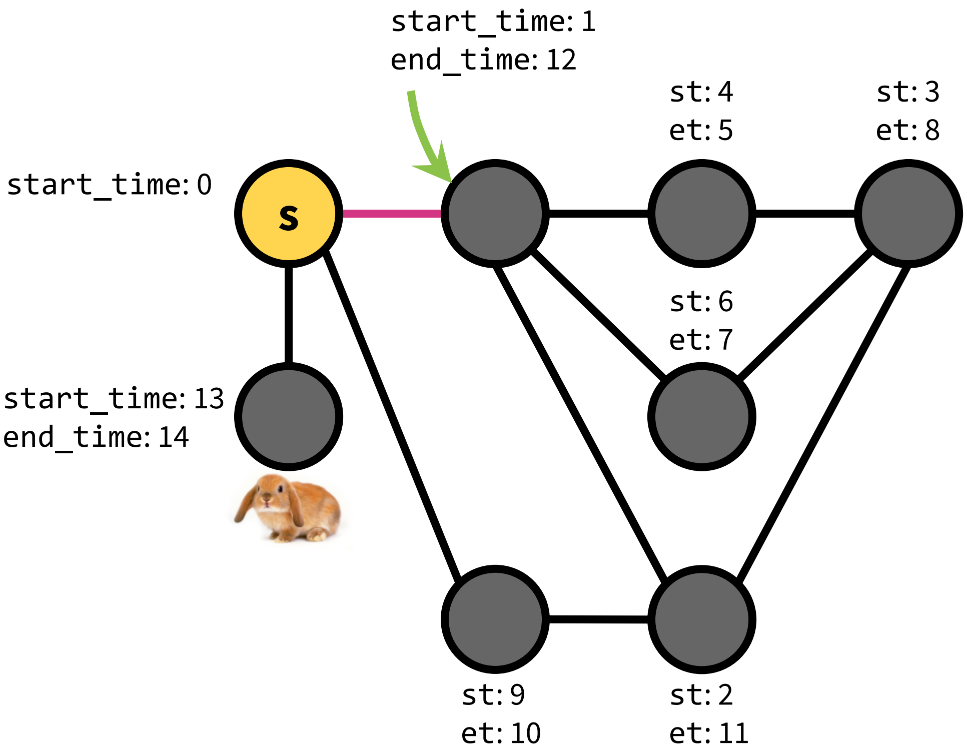
Depth-First Search



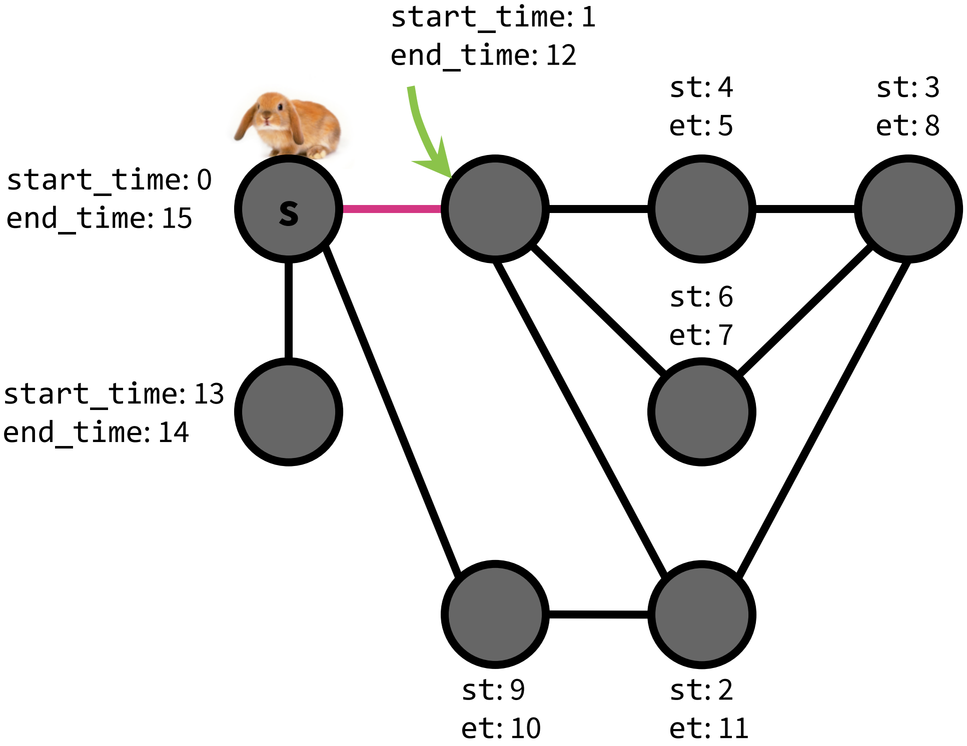
Depth-First Search



Depth-First Search



Depth-First Search

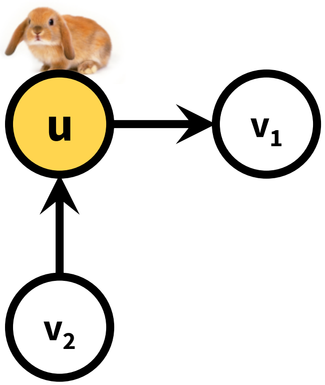


Depth-First Search

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

DFS works fine on directed graphs as well.

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

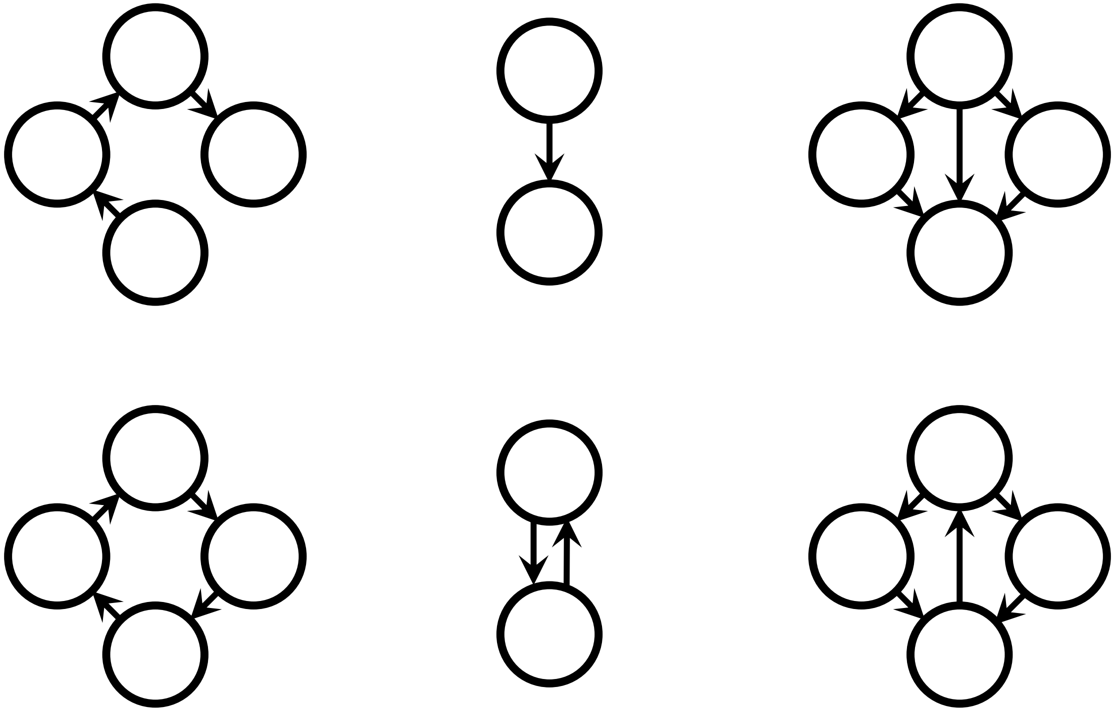


Topological Ordering

Aside: Directed Acyclic Graphs

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

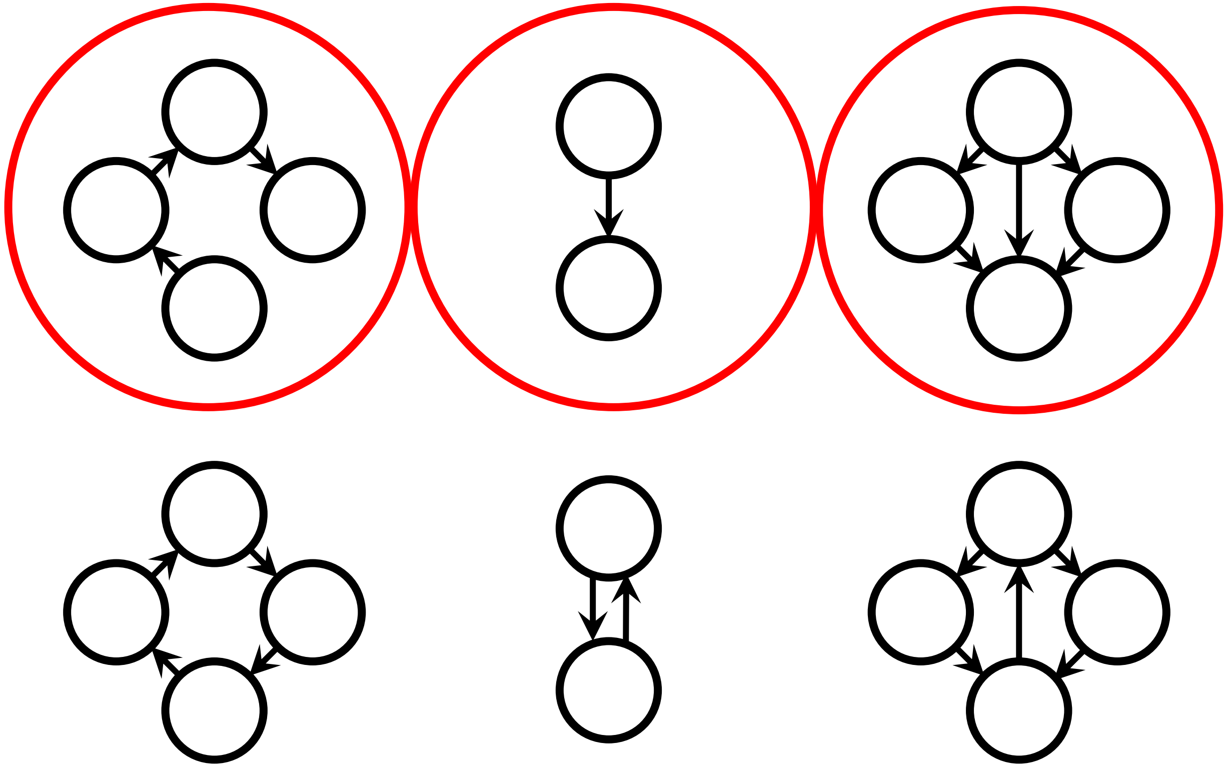
Which of these graphs are valid DAGs? 🤔



Aside: Directed Acyclic Graphs

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

Which of these graphs are valid DAGs? 🤔



Topological Ordering

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

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

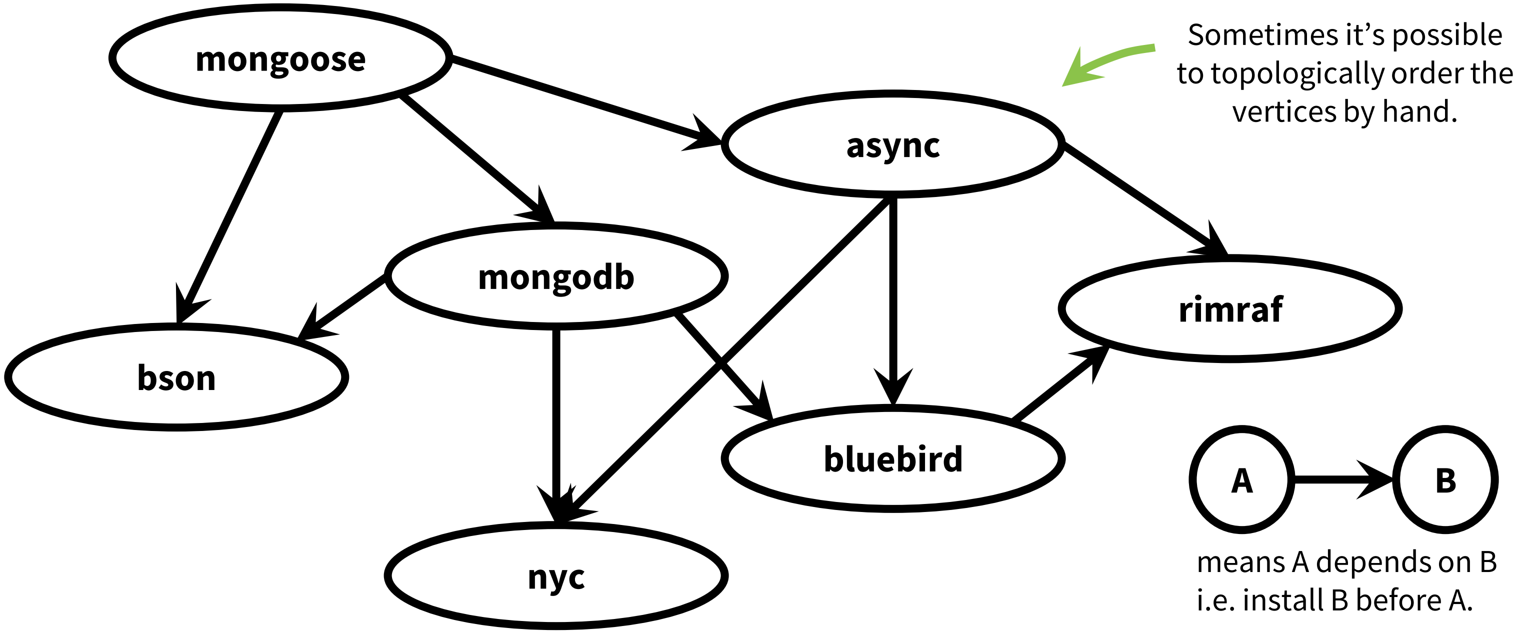
Definition: The topological ordering of a DAG is an ordering of its vertices such that for every directed edge (u, v) ∊ E, u precedes v in the ordering.

Topological Ordering

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

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

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



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

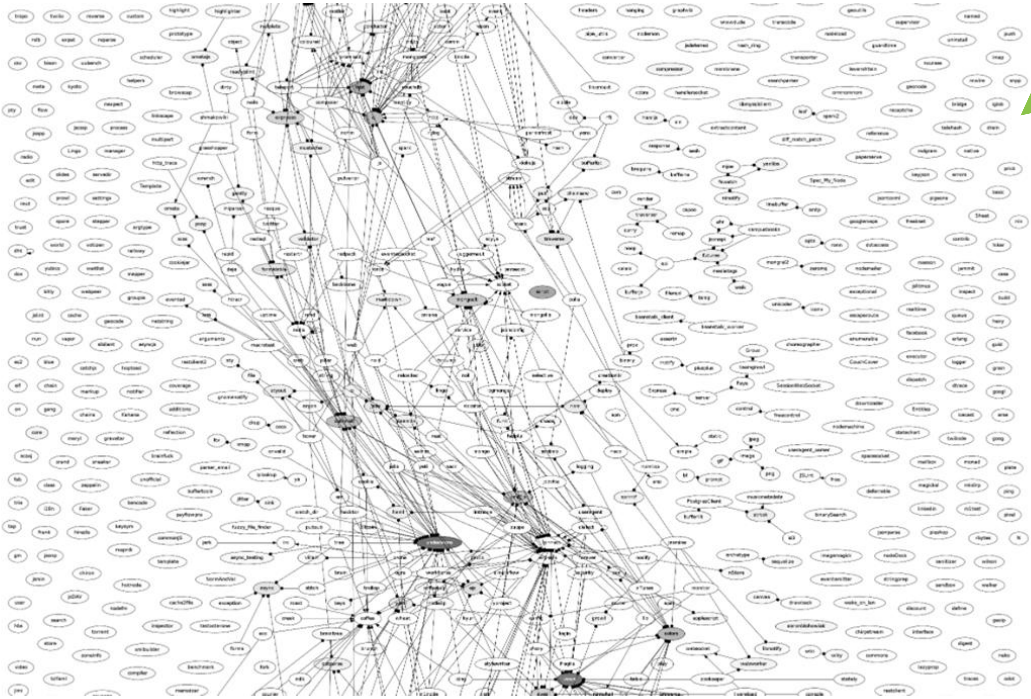
means A depends on B
i.e. install B before A.

Topological Ordering

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

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

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



Sometimes it's not …

Topological Ordering

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

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

Topological Ordering

```
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
```

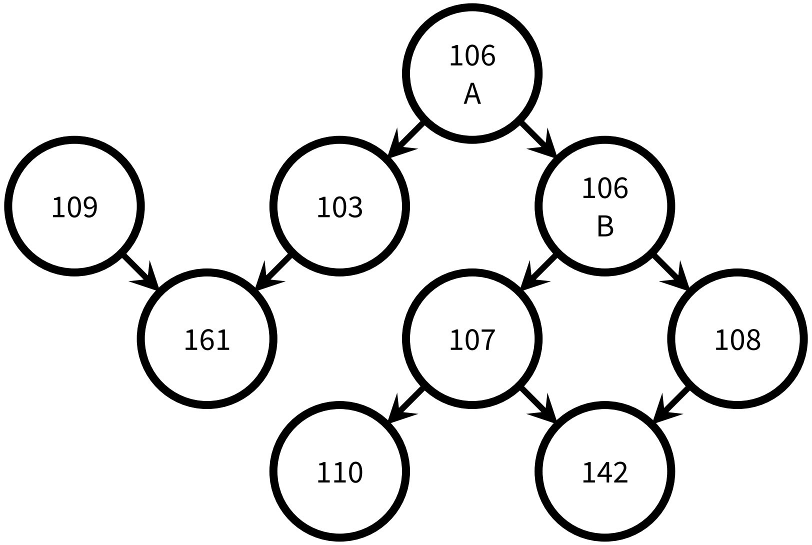
Runtime: O(|V|+|E|)

Topological Ordering

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

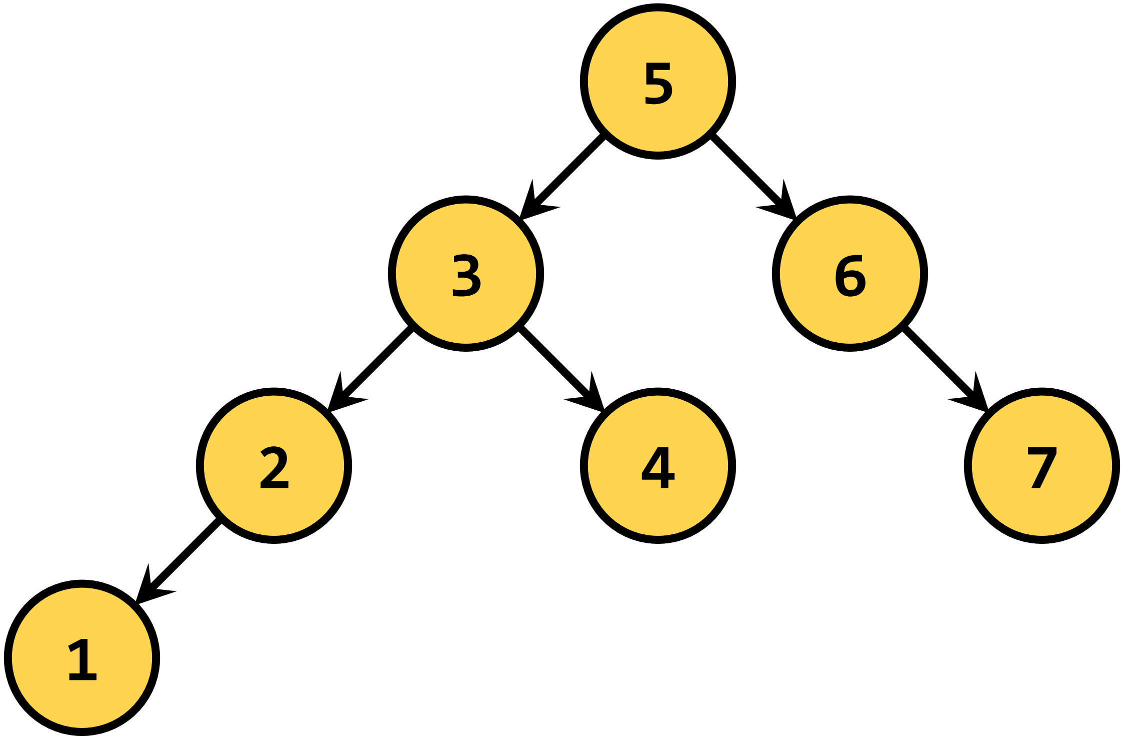
To compute the topological ordering in general, reverse the order of `reversed_topological_list`.

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



In-Order Traversal of BSTs

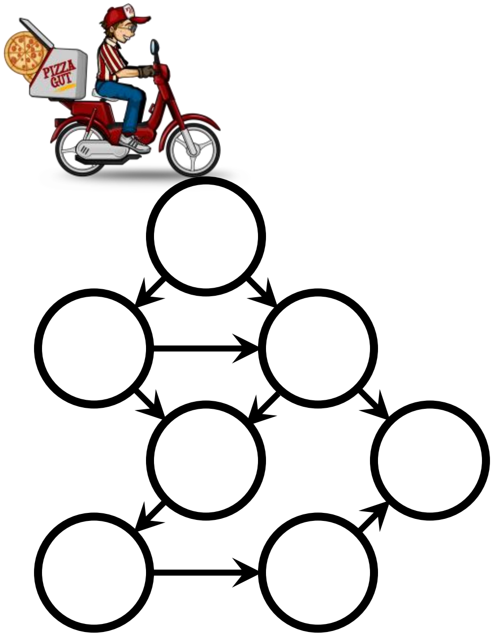
Application of DFS: Given a BST, output the vertices in order.



Exact Traversals of Graphs

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

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

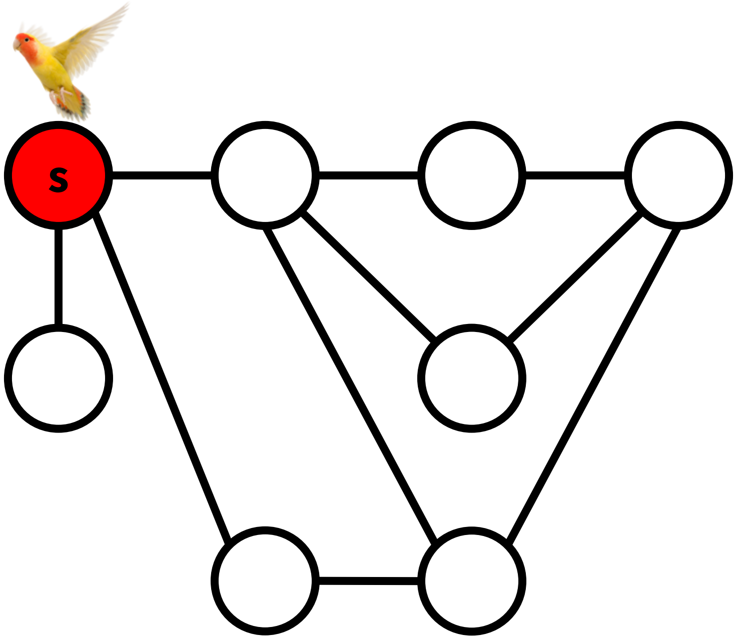


Breadth-First Search

Breadth-First Search

An analogy

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



○ unvisited

🔴 reachable in 0 steps

🟠 reachable in 1 steps

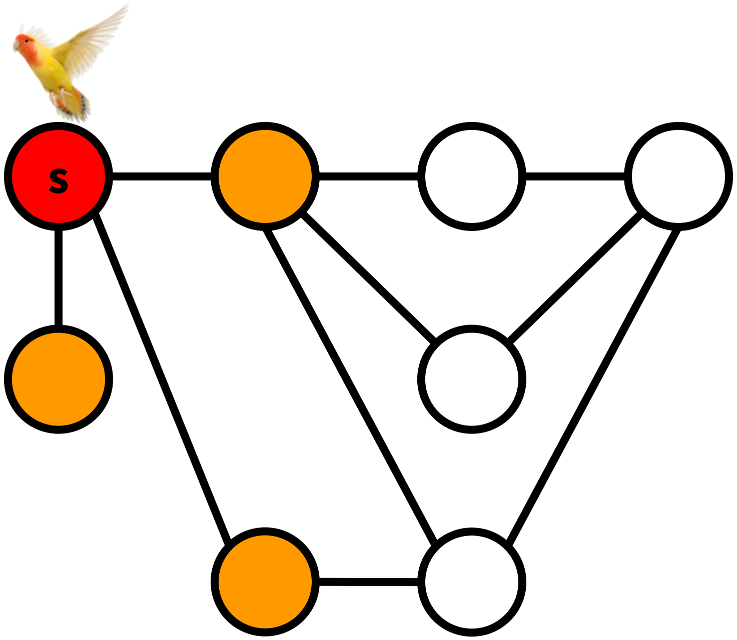
🟡 reachable in 2 steps

🟢 reachable in 3 steps

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

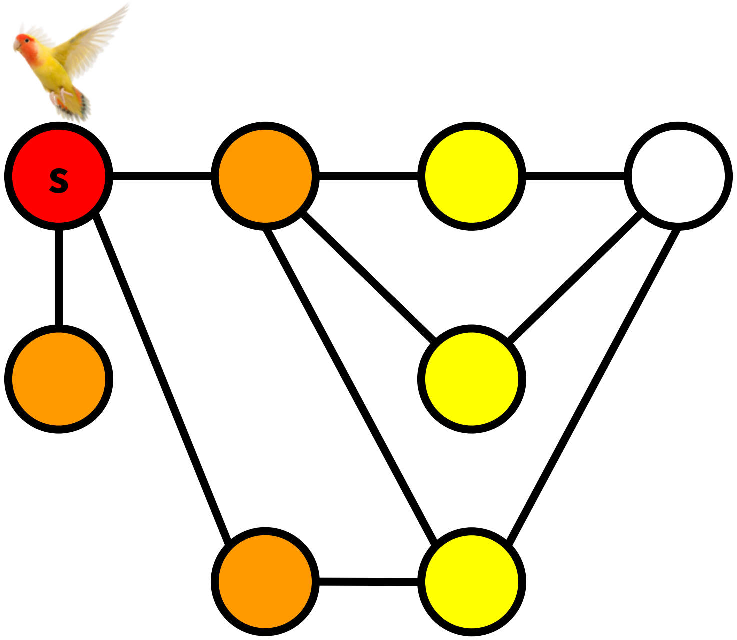
🟡 reachable in 2 steps

🟢 reachable in 3 steps

Breadth-First Search

An analogy

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



○ unvisited

🔴 reachable in 0 steps

🟠 reachable in 1 steps

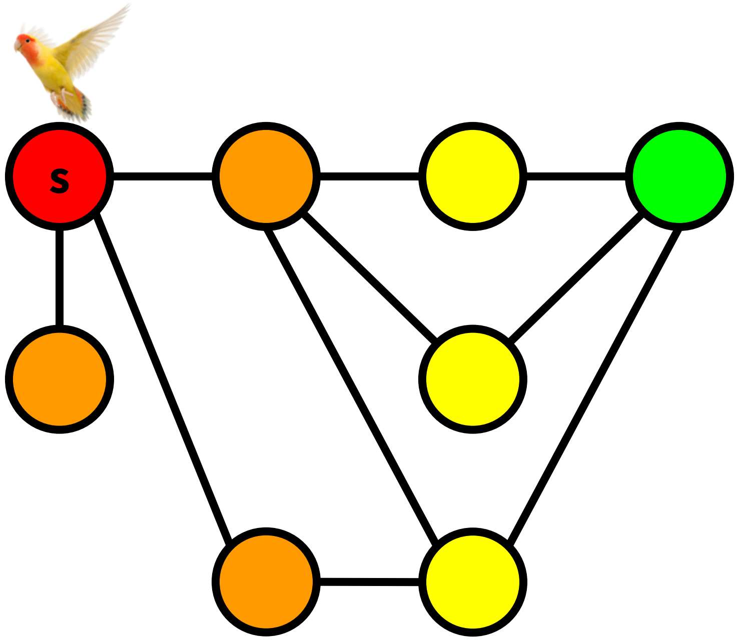
🟡 reachable in 2 steps

🟢 reachable in 3 steps

Breadth-First Search

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

Shortest Path

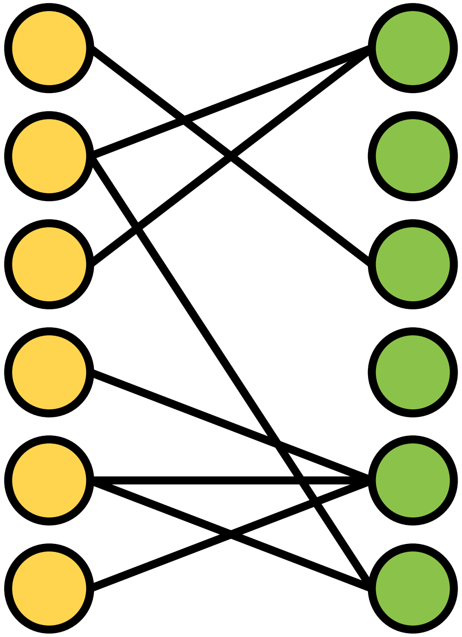
Application of BFS: How long is the shortest path between vertices u and v?

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

Aside: Bipartiteness

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

e.g. Matching university hackathon guests and hosts.

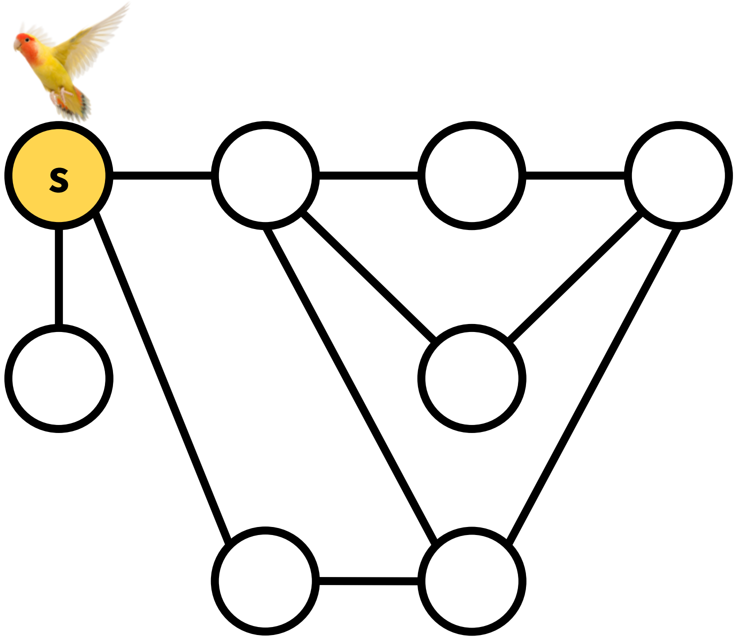


Shortest Path

Application of BFS: Is a graph bipartite?

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

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

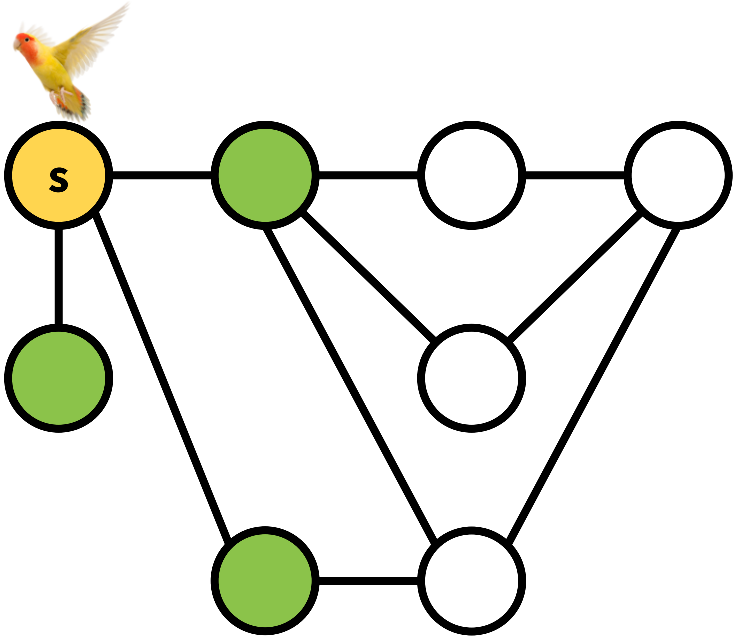


Shortest Path

Application of BFS: Is a graph bipartite?

Call bfs from any vertex and color vertices alternating colors.

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

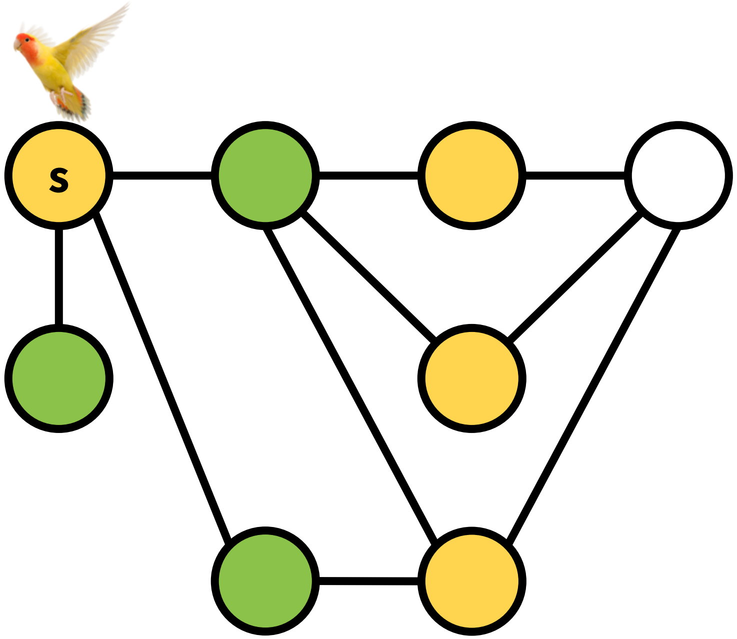


Shortest Path

Application of BFS: Is a graph bipartite?

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

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

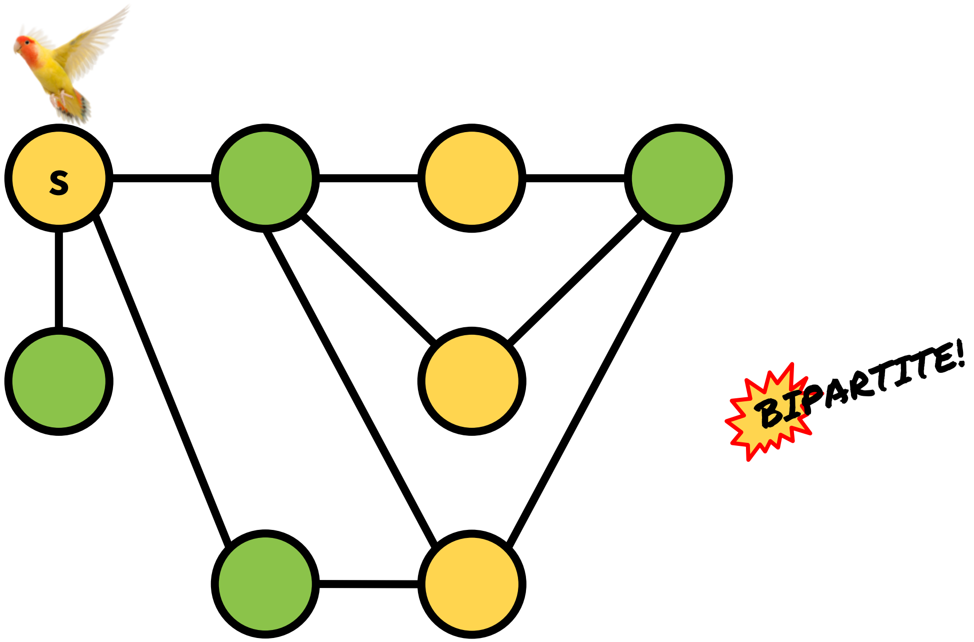


Shortest Path

Application of BFS: Is a graph bipartite?

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

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

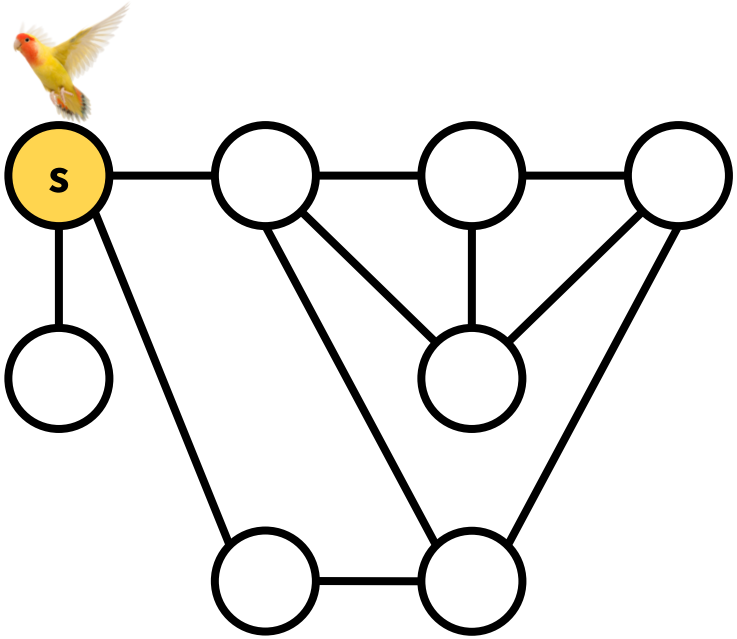


Shortest Path

Application of BFS: Is a graph bipartite?

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

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

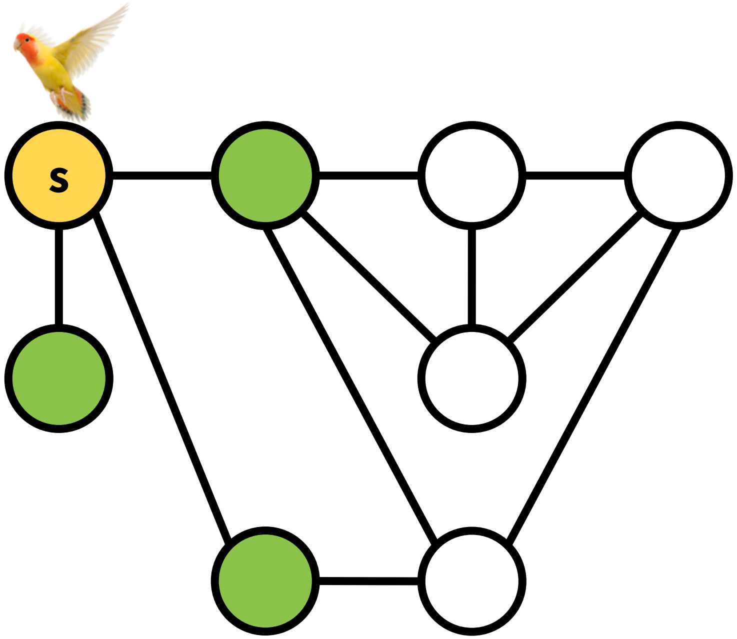


Shortest Path

Application of BFS: Is a graph bipartite?

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

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

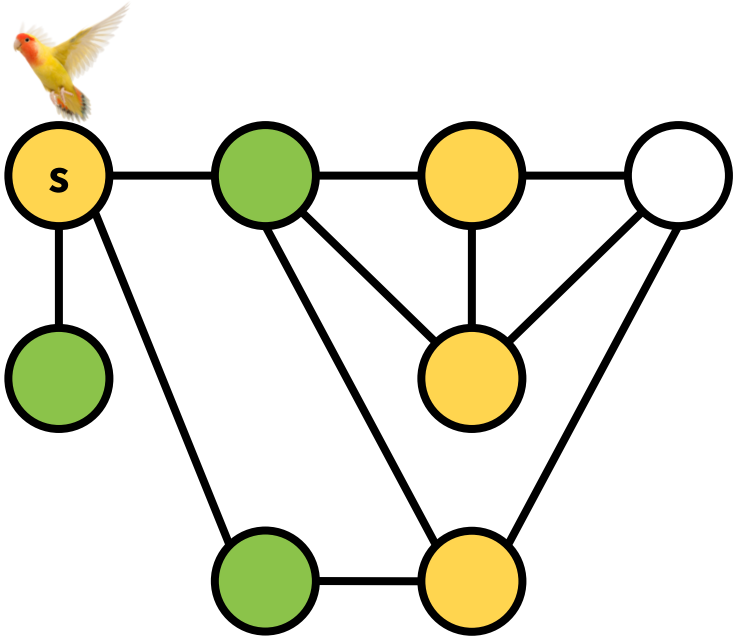


Shortest Path

Application of BFS: Is a graph bipartite?

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

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

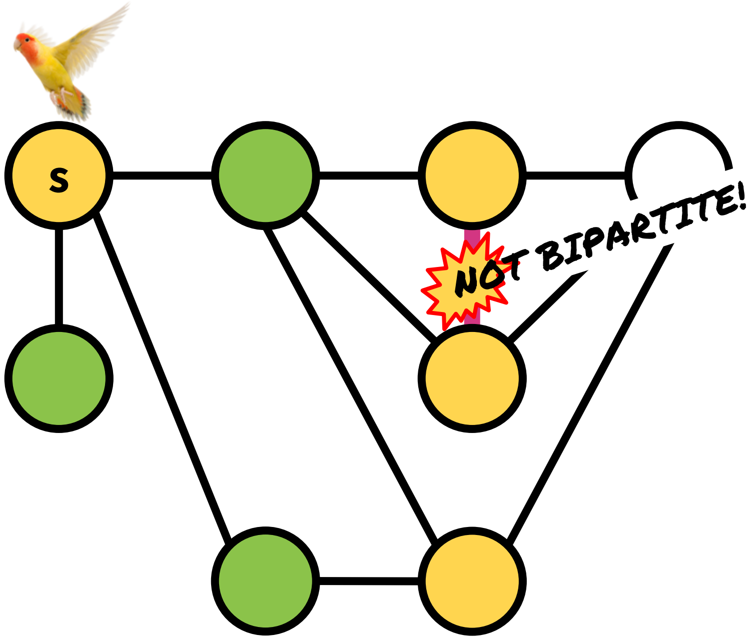


Shortest Path

Application of BFS: Is a graph bipartite?

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

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

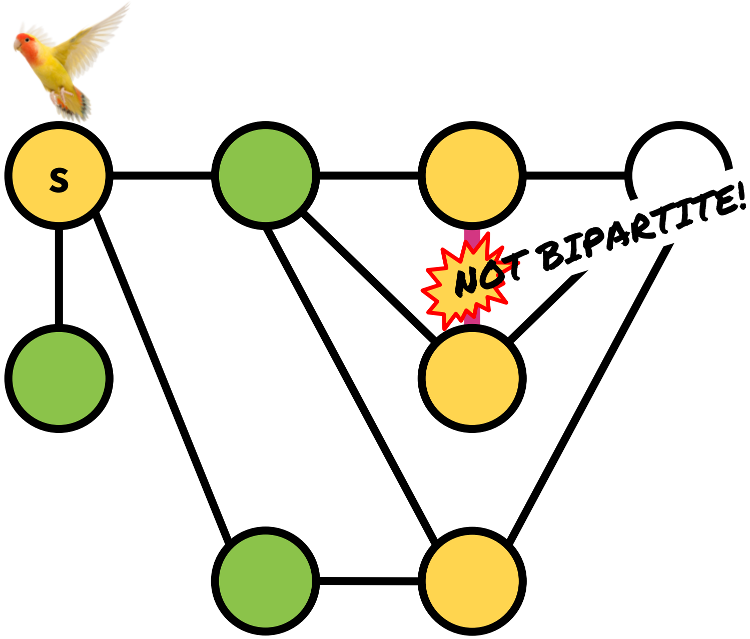


Shortest Path

Application of BFS: Is a graph bipartite?

Call bfs from any vertex and color vertices alternating colors.

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

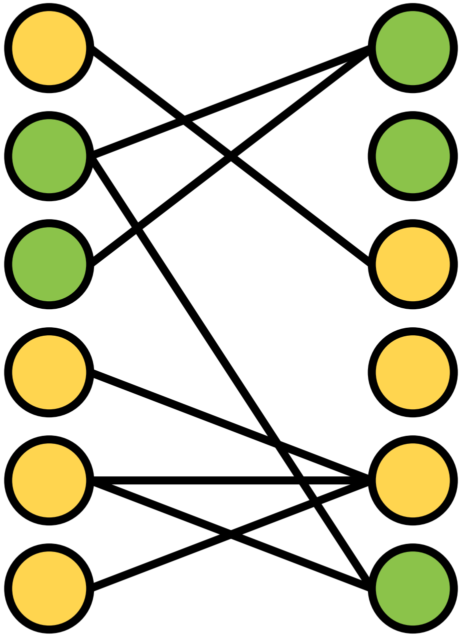


Is anyone incredulous that this actually works? 🤔

Shortest Path

There exist many poor colorings on legitimate bipartite graphs.

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



This is a poor coloring on an obviously bipartite graph.

Shortest Path

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

Proof:

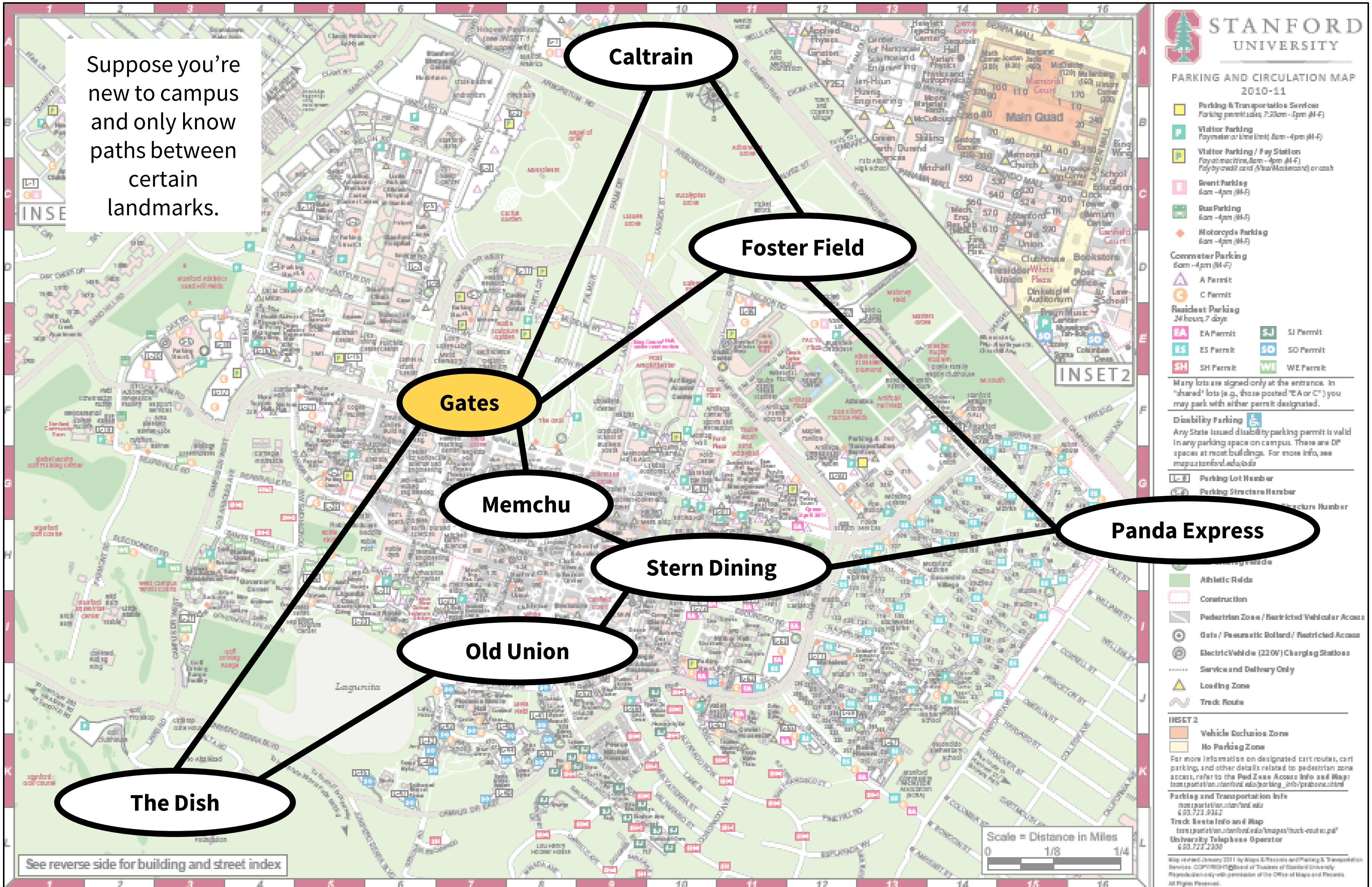
> Since `bfs` colors vertices alternating colors, it colors two neighbors the same color iff it's found a cycle of odd length in the graph. Therefore, the graph contains an odd cycle as a subgraph. But it's impossible to color an odd cycle with two colors such that no two neighbors have the same color.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

3 min break

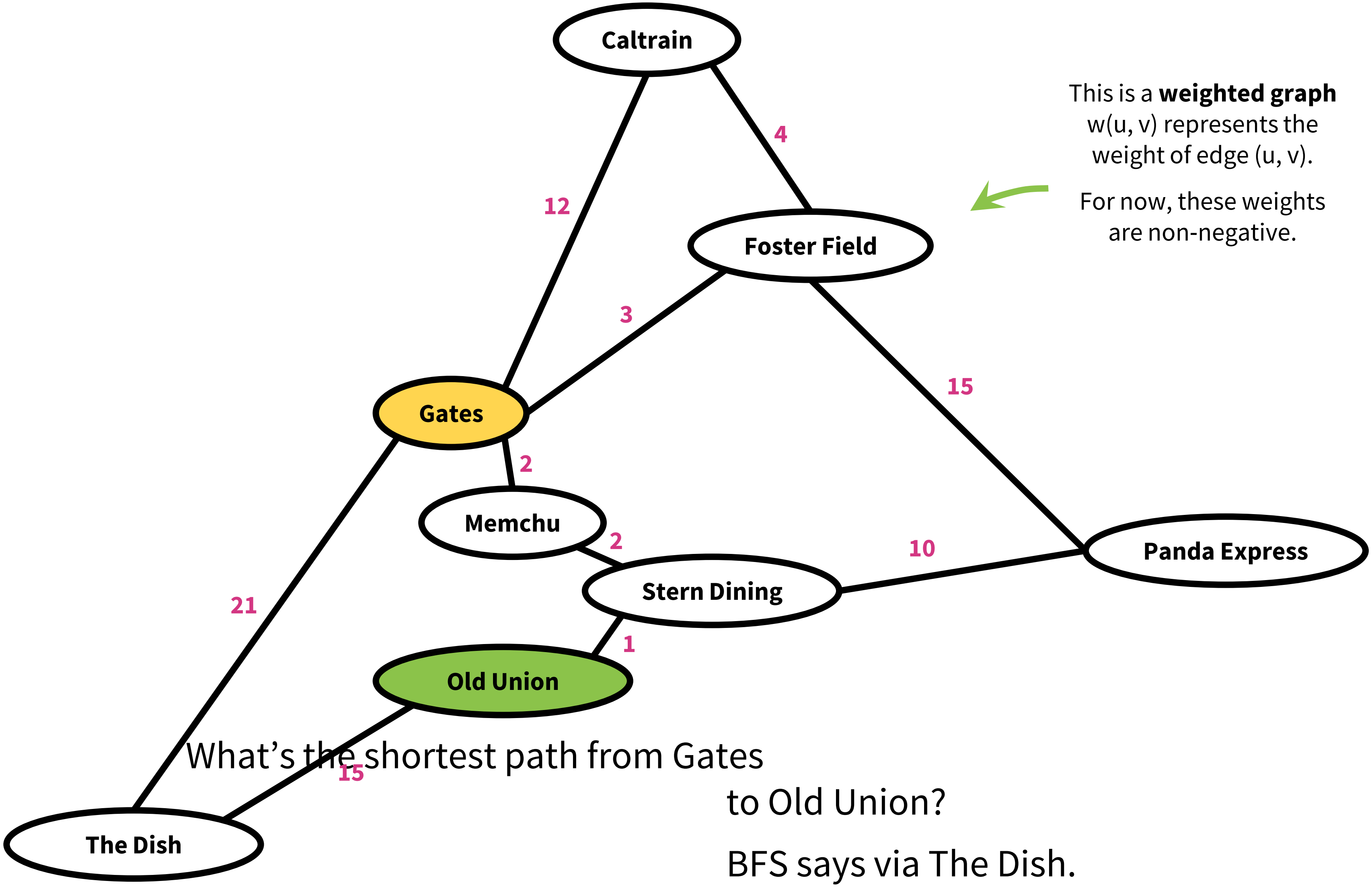
Dijkstra’s Algorithm

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 Gates
to Old Union?

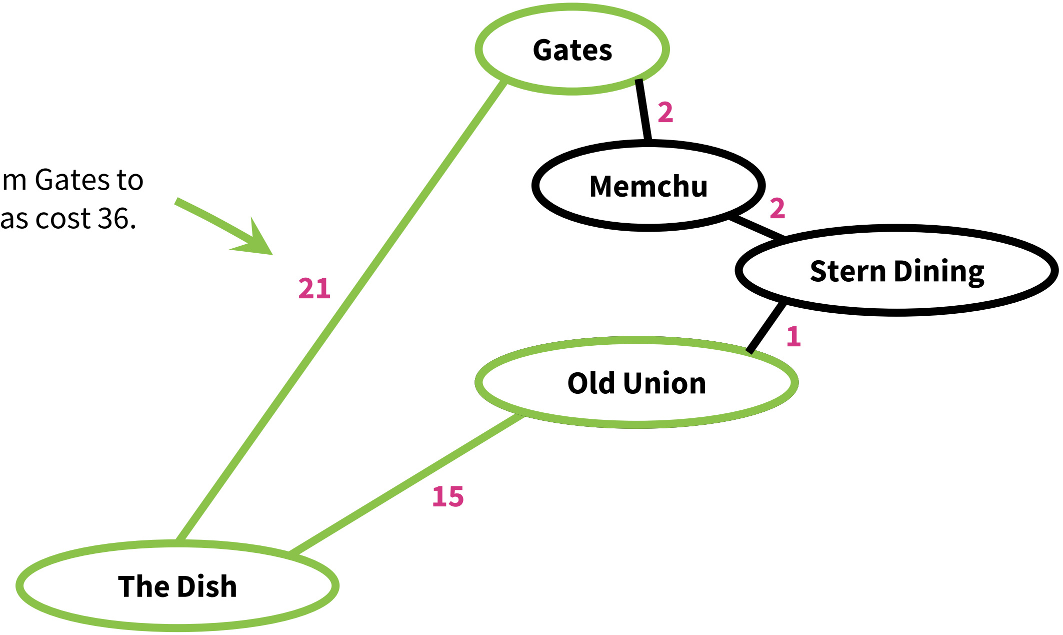
BFS says via The Dish.

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.

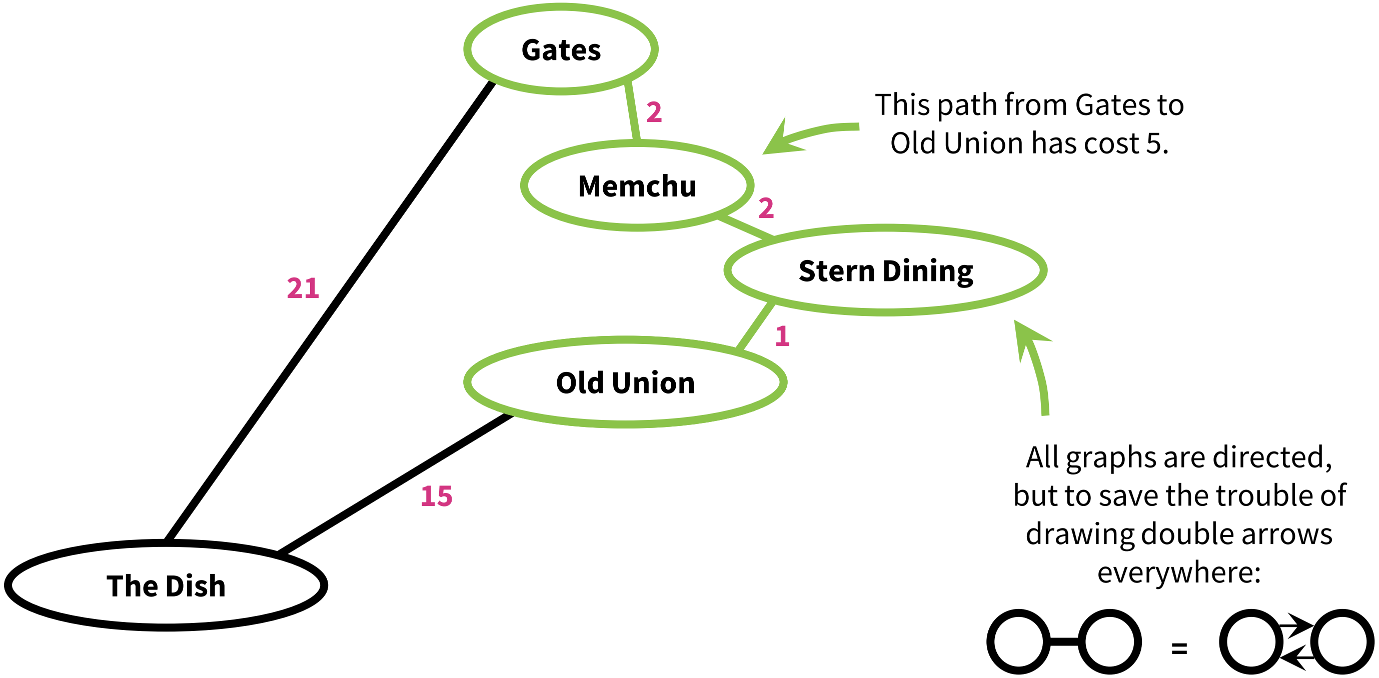


This path from Gates to Old Union has cost 36.

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.



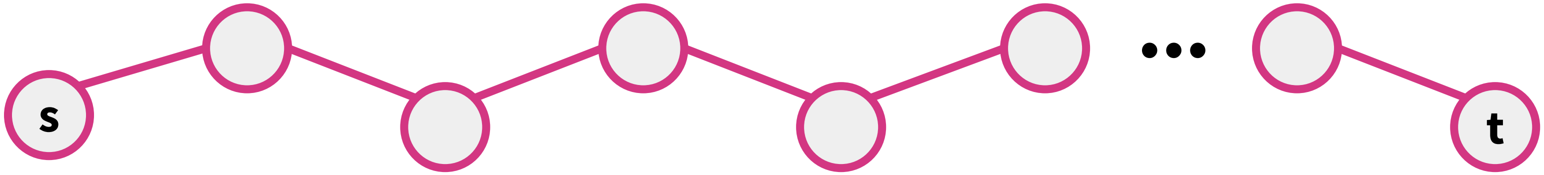
This path from Gates to Old Union has cost 5.

All graphs are directed, but to save the trouble of drawing double arrows everywhere:

Shortest Path

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

Intuition:

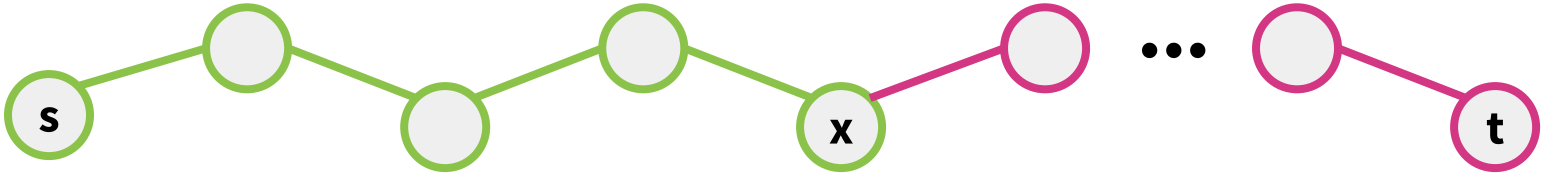


Suppose **this** is a shortest path from **s** to **t**.

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**.

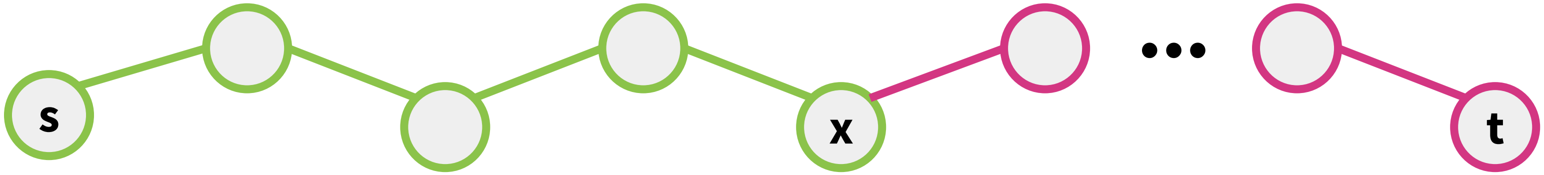
Then **this** is a shortest path from **s** to **x**.

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**.

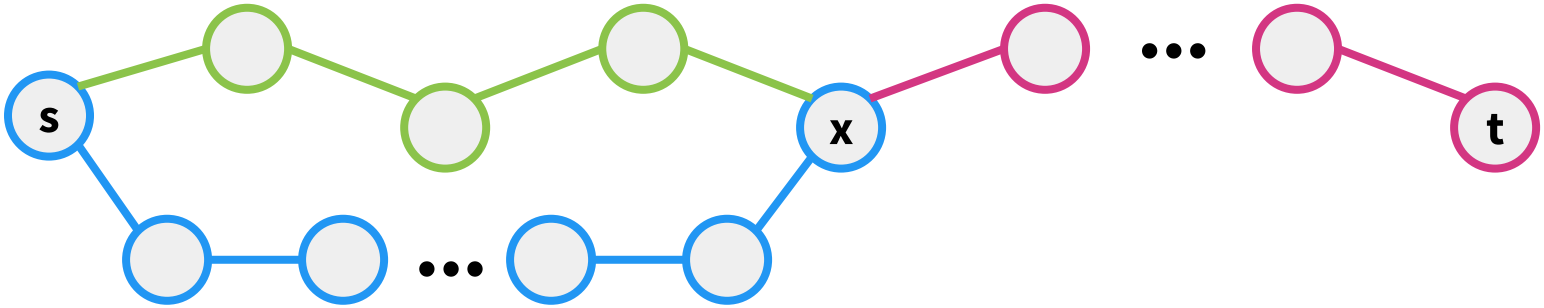
Then **this** is a shortest path from **s** to **x**.

Why? 🤔

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**.

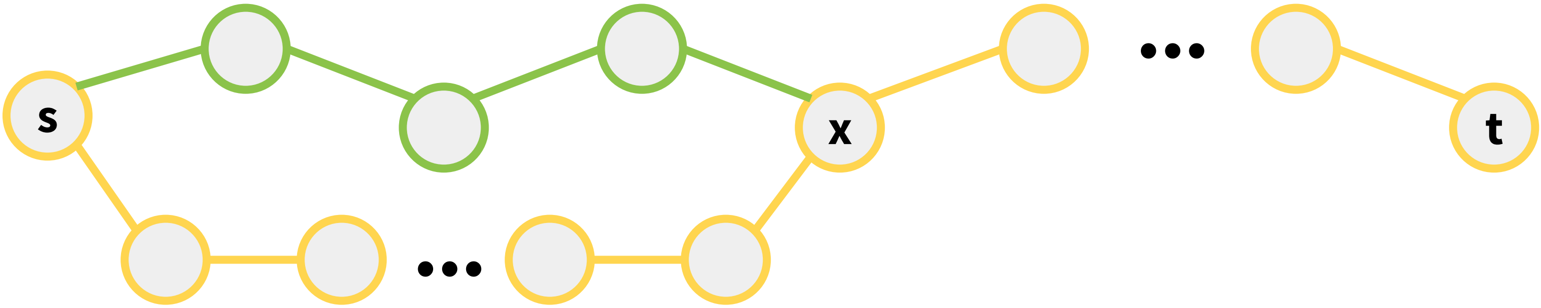
Then **this** is a shortest path from **s** to **x**.

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:



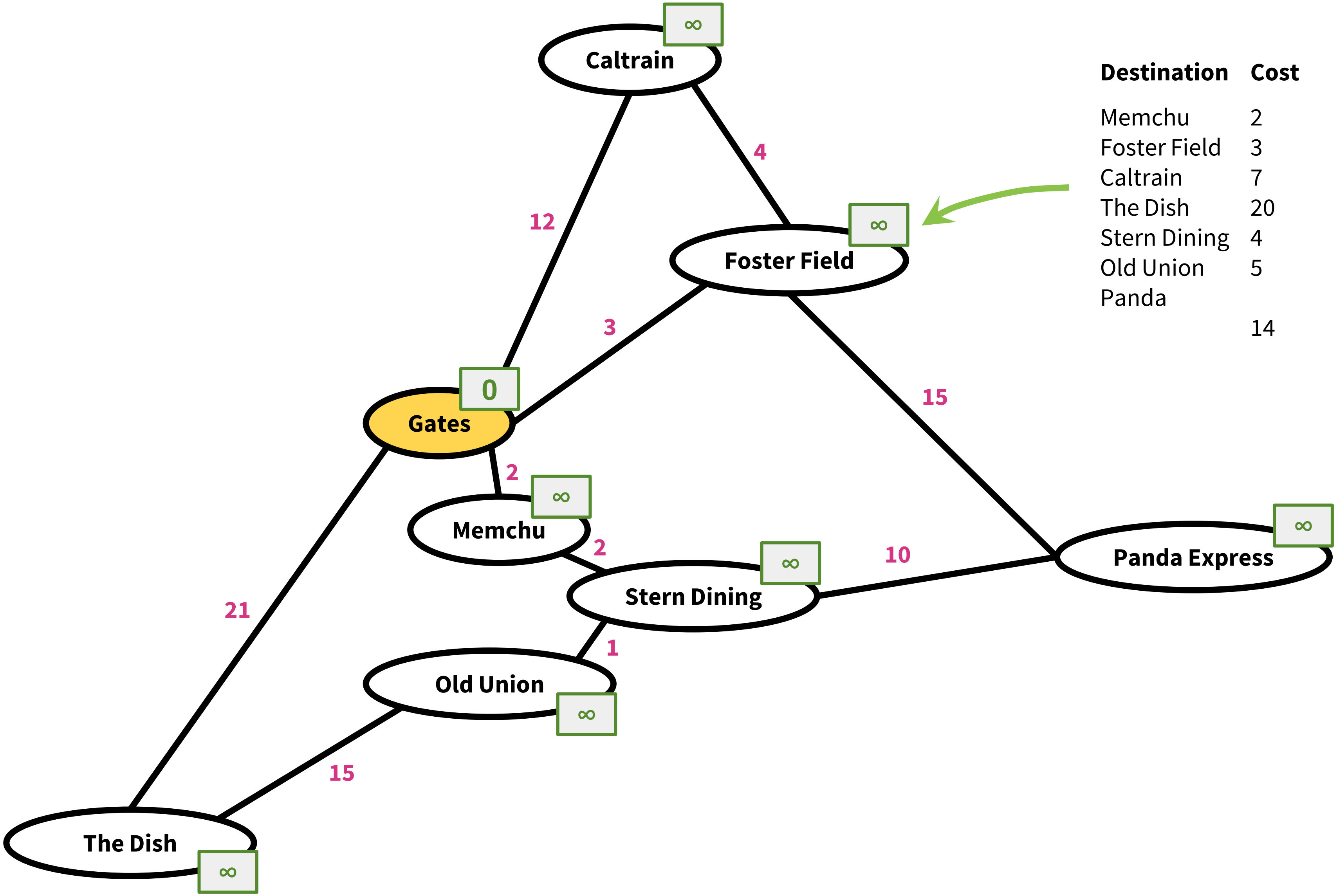
Suppose **this** is a shortest path from **s** to **t**.

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
Memchu	2
Foster Field	3
Caltrain	7
The Dish	20
Stern Dining	4
Old Union	5
Panda	
	14

Single-Source Shortest Path

Application: Finding the shortest path from Palo Alto to [somewhere else] for a commuter using BART, Caltrain, bike, walking, Uber, Lyft, etc.

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

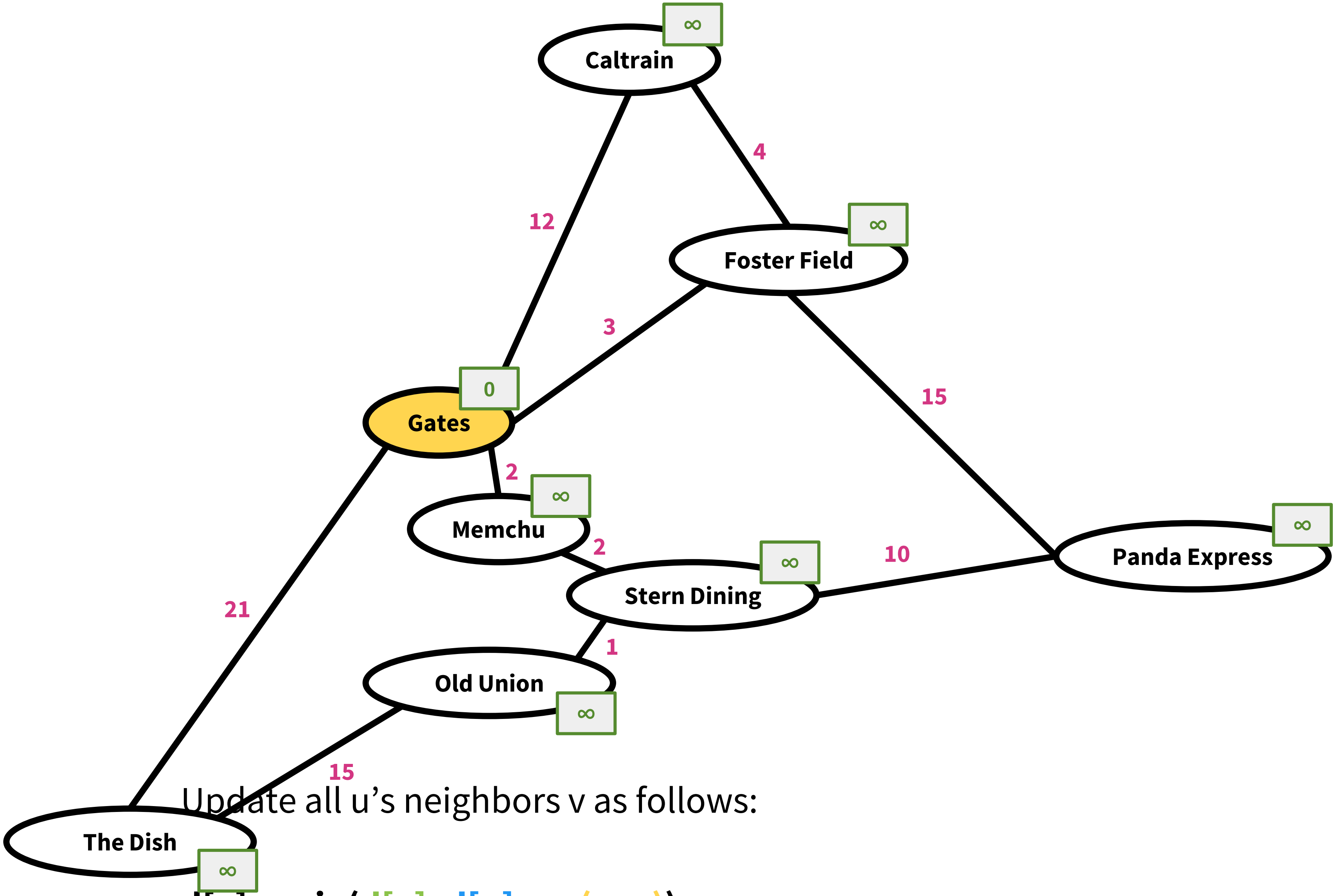
Application: Finding the shortest path from my computer to the desired server for packets using the Internet.

Edge weights are a function of link length, traffic, other costs, etc.

Dijkstra’s Algorithm

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

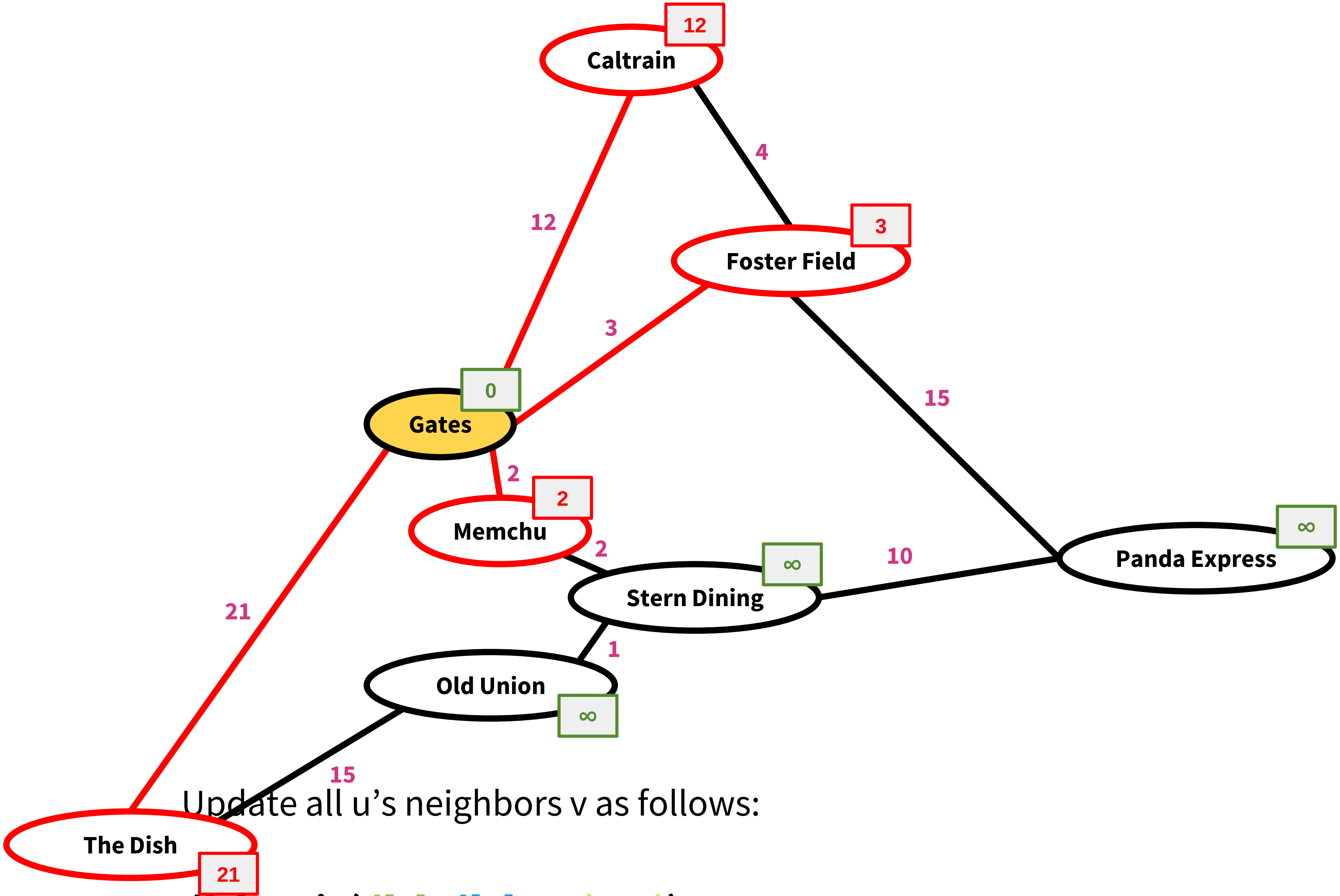
Dijkstra's Algorithm



Update all u's neighbors v as follows:

$d[v] = \min(d[v], d[u] + w(u, v))$

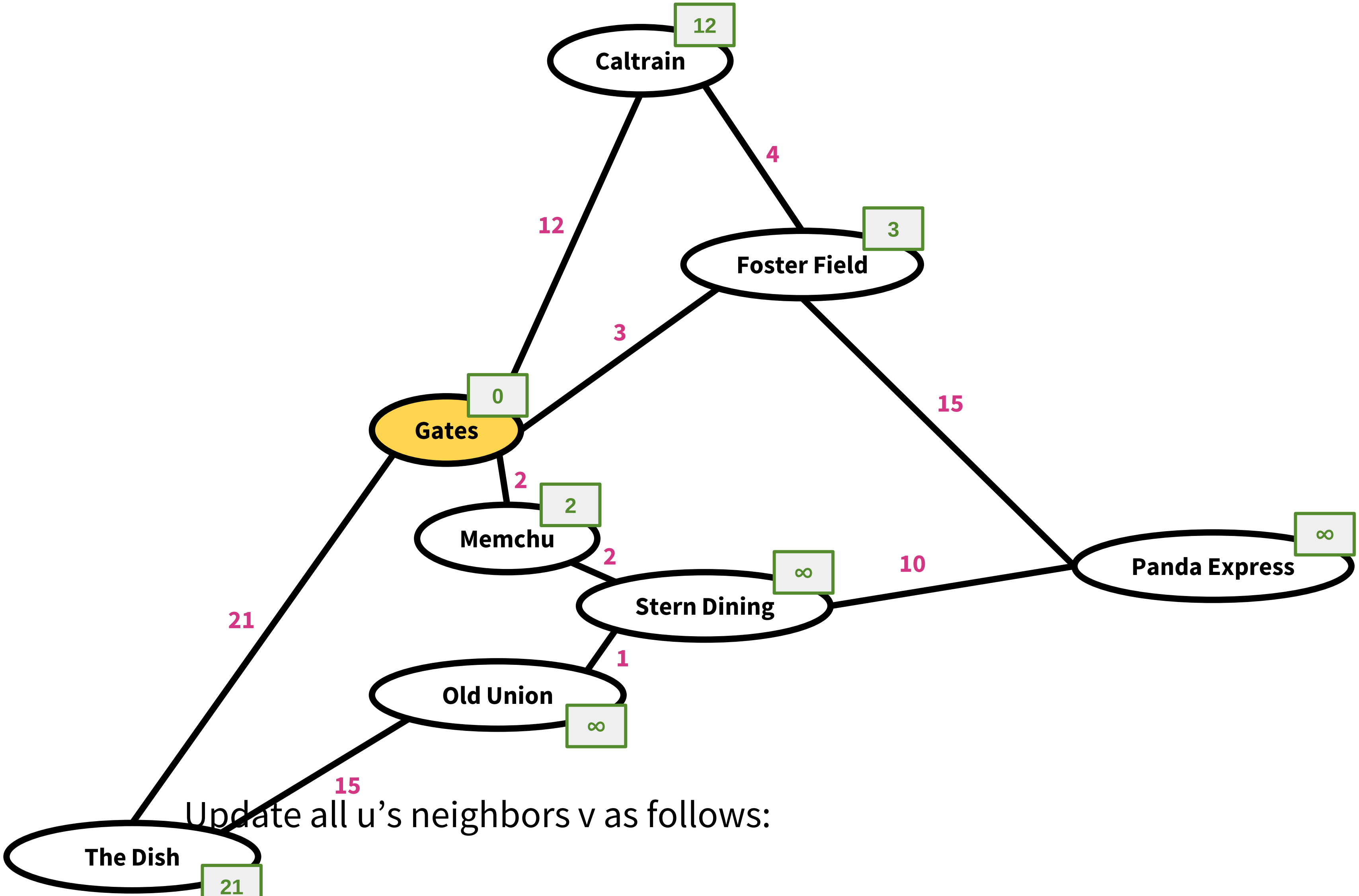
Dijkstra's Algorithm



Update all u's neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

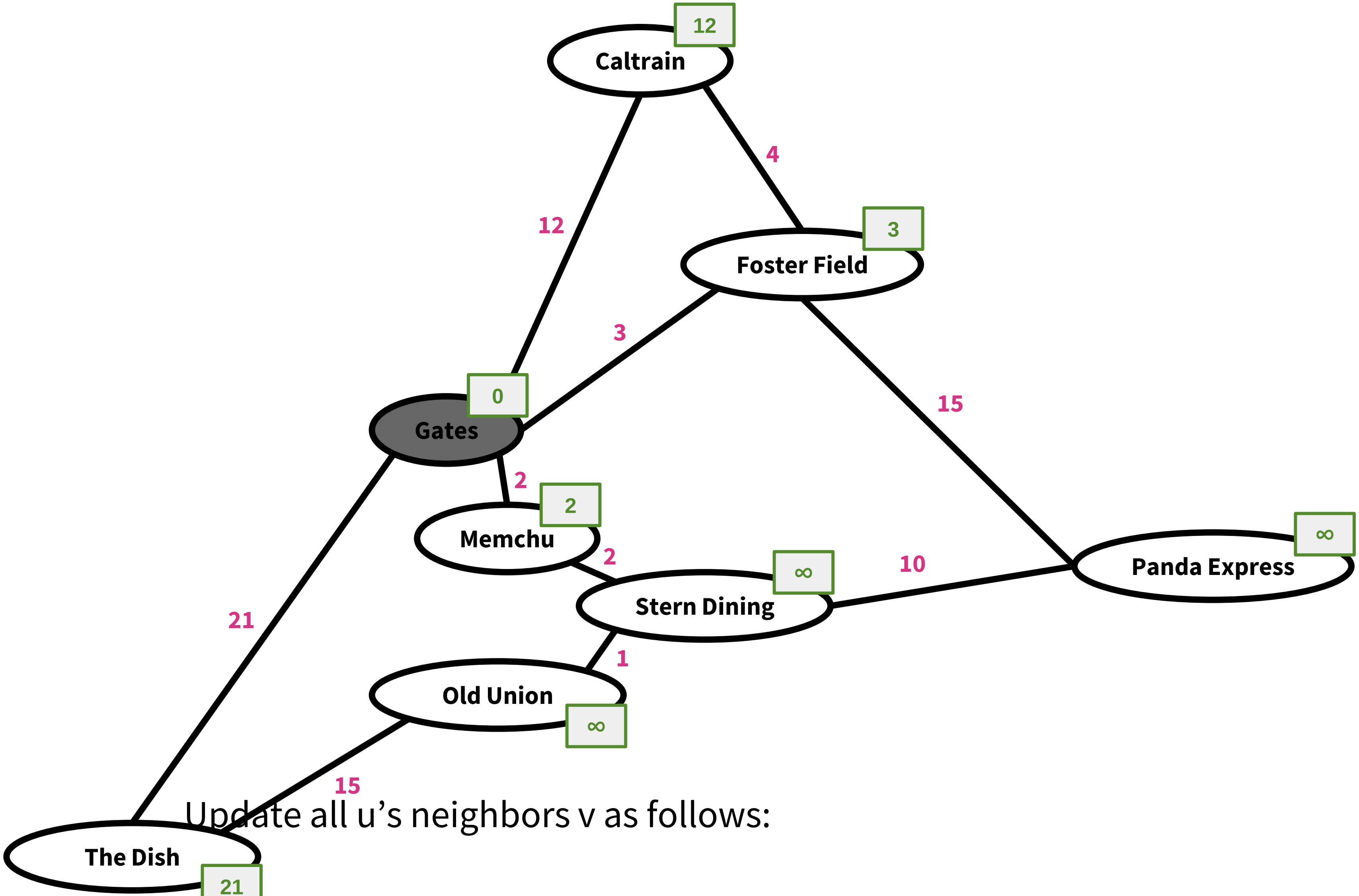
Dijkstra's Algorithm



Update all u's neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

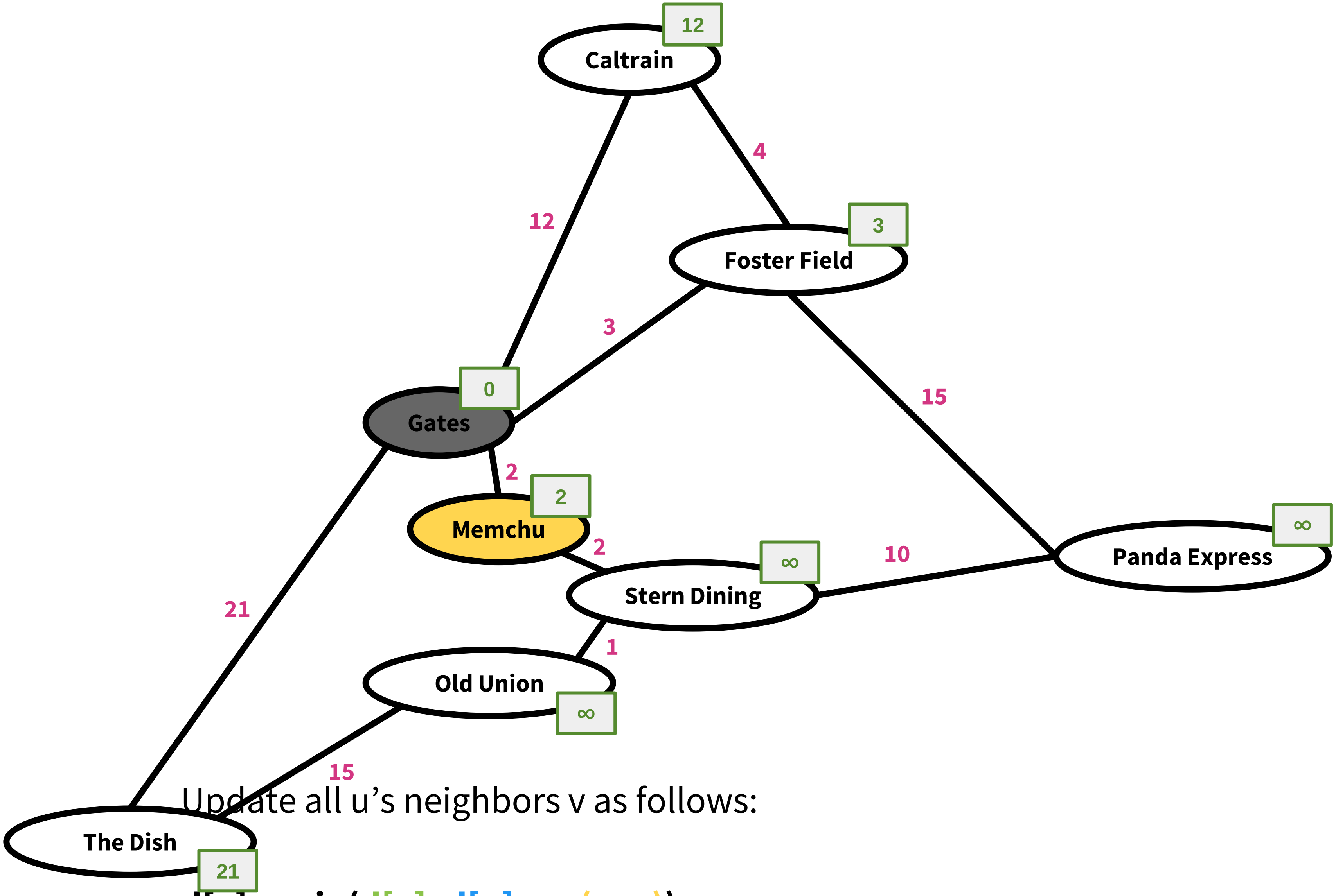
Dijkstra's Algorithm



Update all u's neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

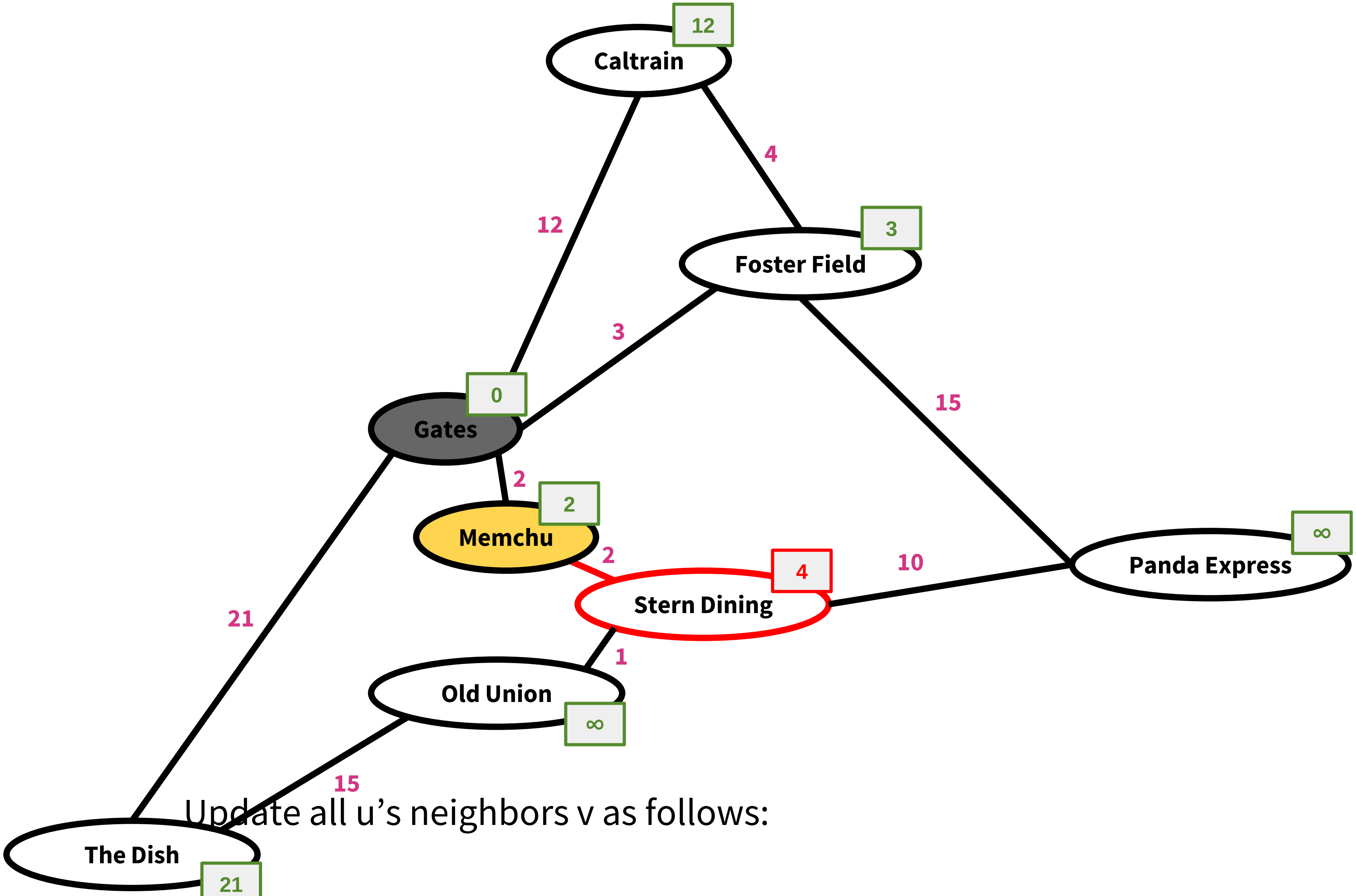
Dijkstra's Algorithm



Update all u's neighbors v as follows:

$$d[v] = \min(d[v], d[u] + w(u, v))$$

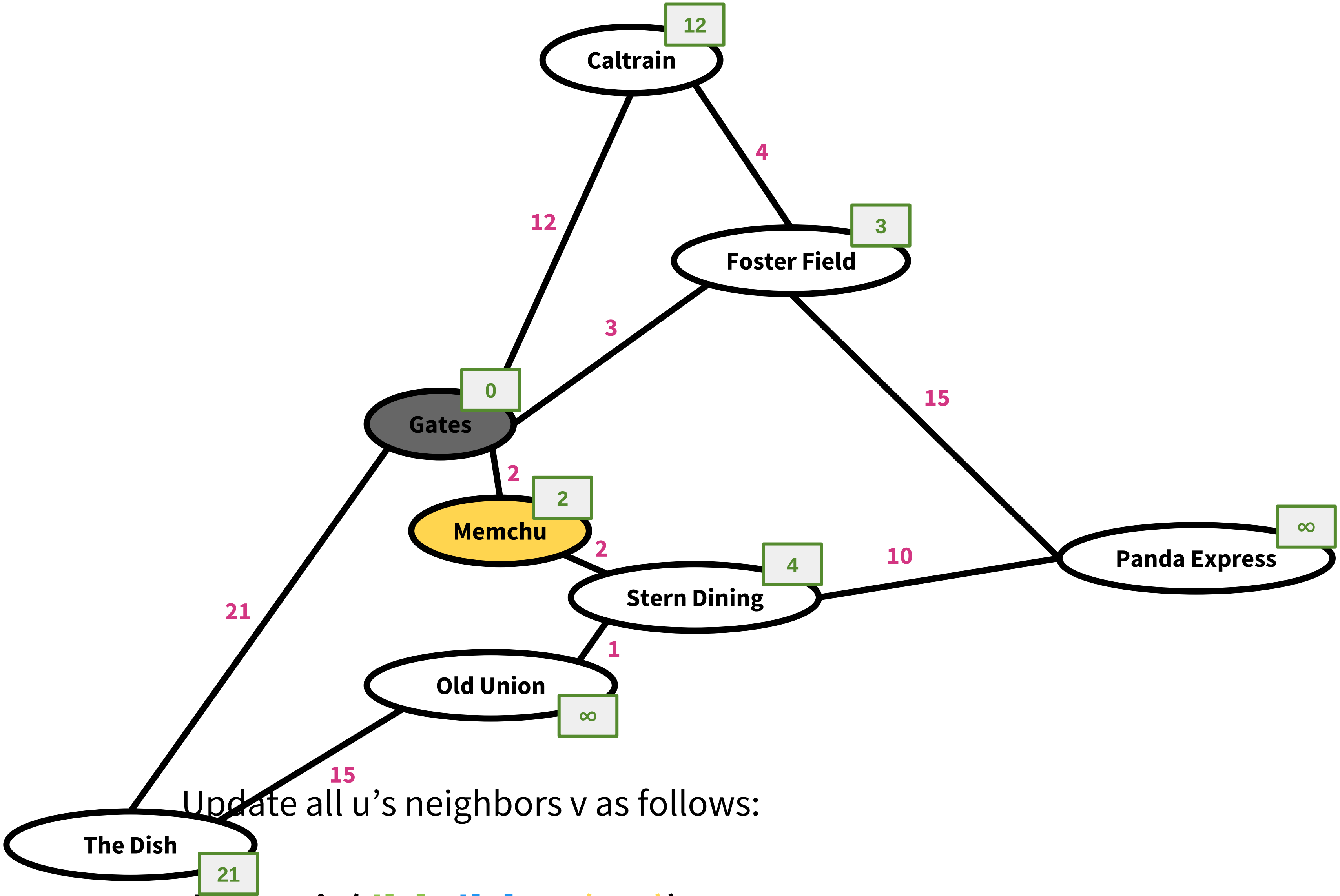
Dijkstra's Algorithm



Update all u's neighbors v as follows:

$d[v] = \min(d[v], d[u] + w(u, v))$

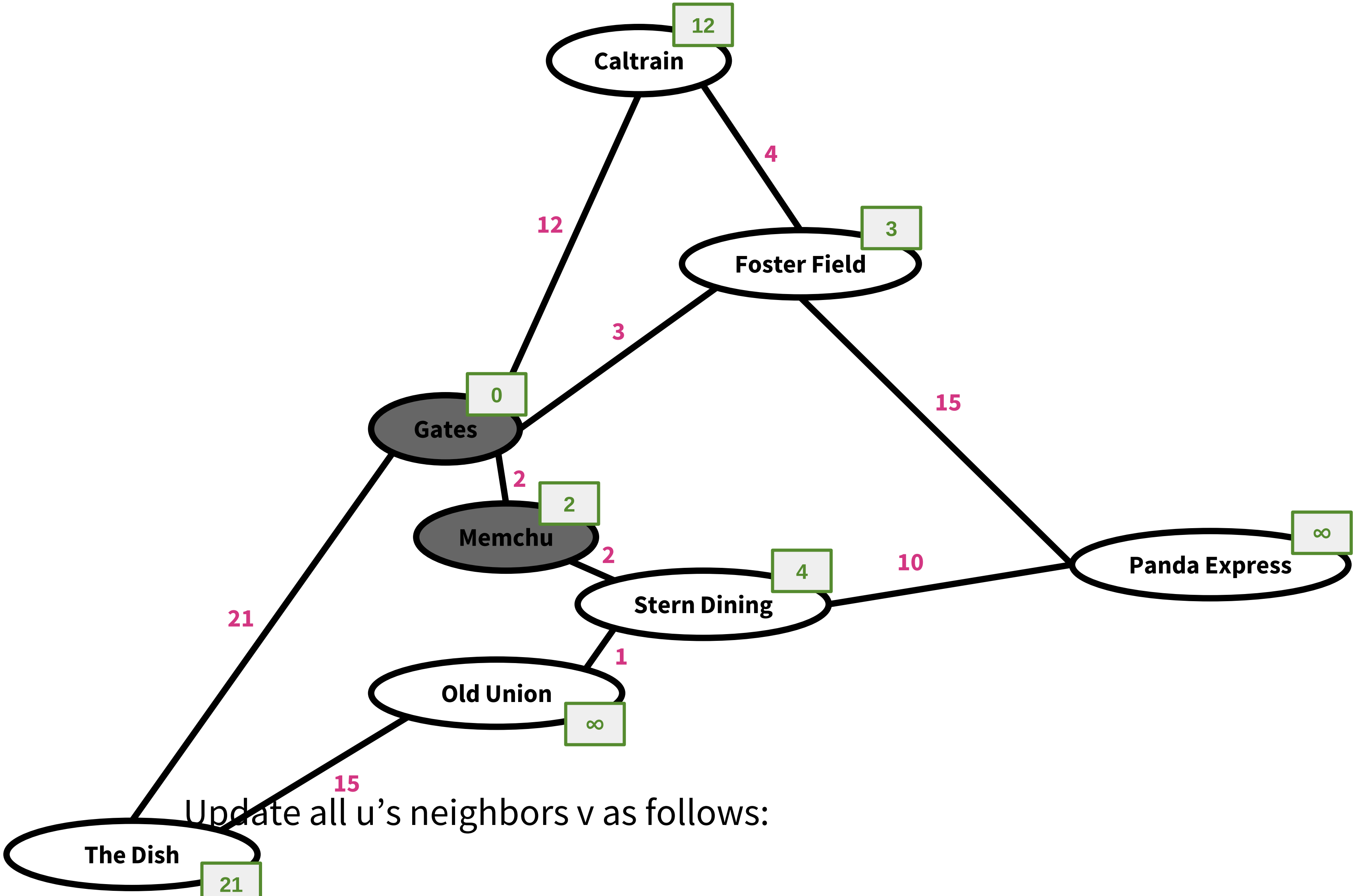
Dijkstra's Algorithm



Update all u's neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

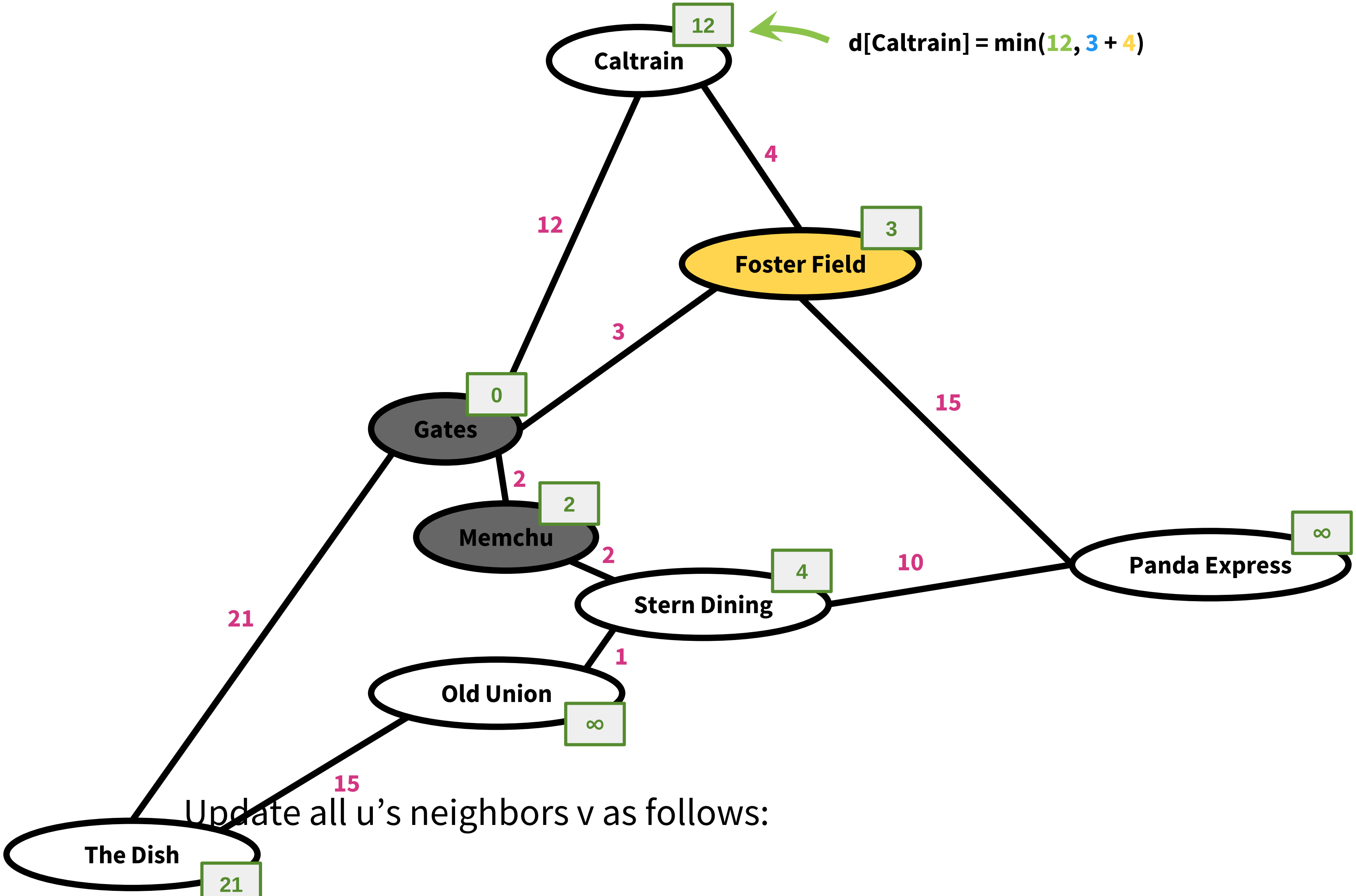
Dijkstra's Algorithm



Update all u's neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

Dijkstra's Algorithm

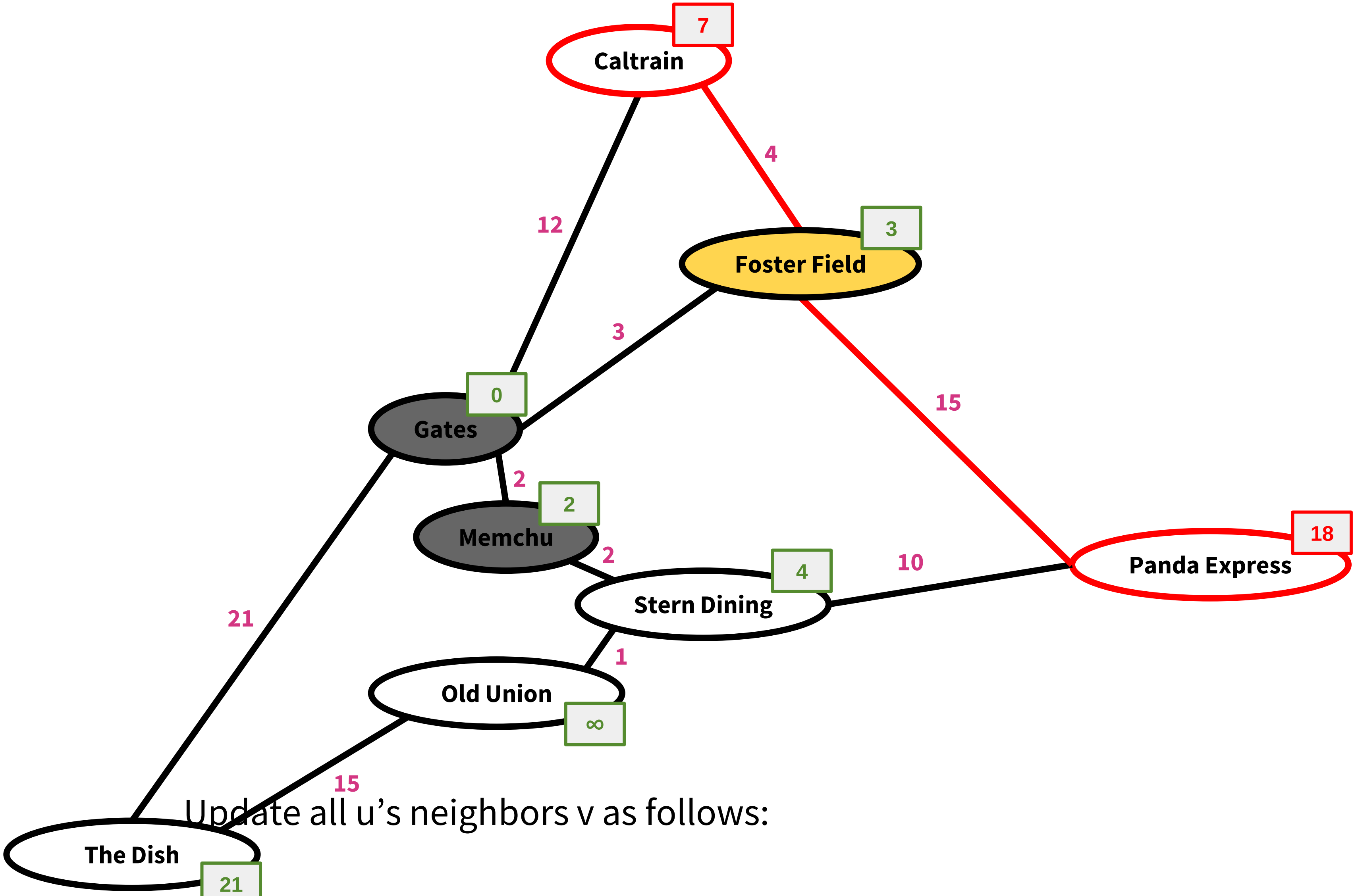


d[Caltrain] = min(12, 3 + 4)

Update all u's neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

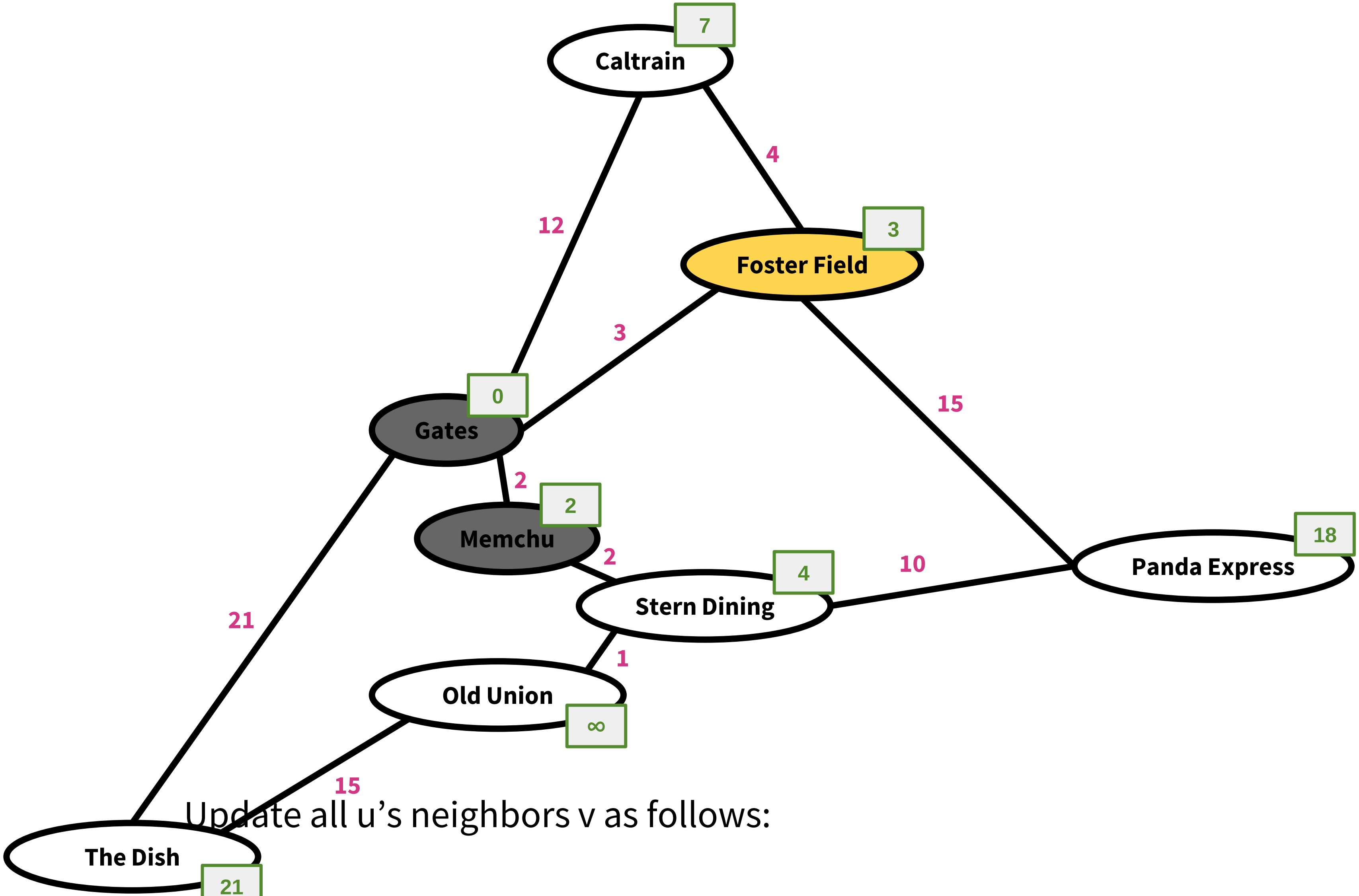
Dijkstra’s Algorithm



Update all u’s neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

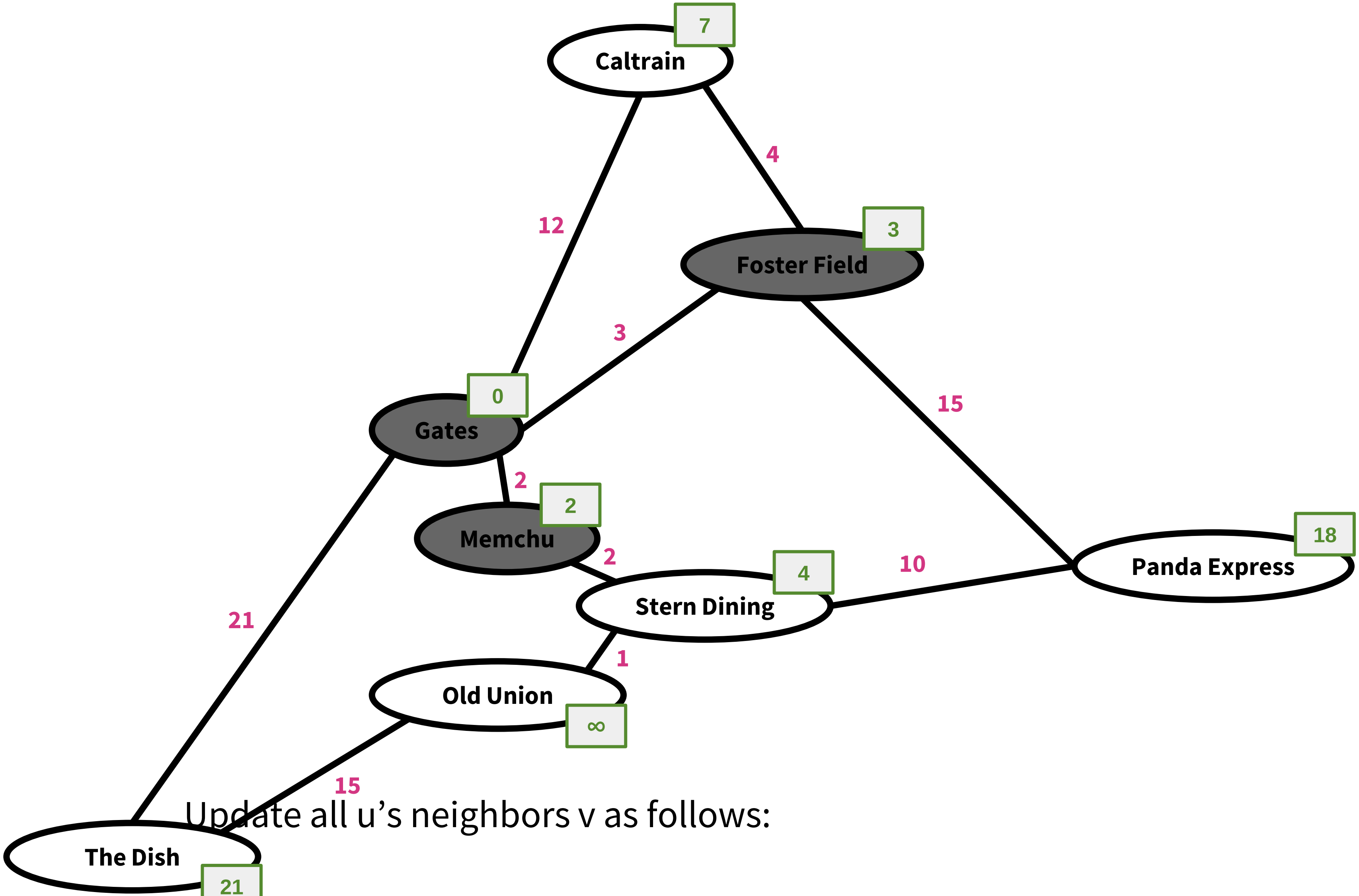
Dijkstra’s Algorithm



Update all u’s neighbors v as follows:

$$d[v] = \min(d[v], d[u] + w(u, v))$$

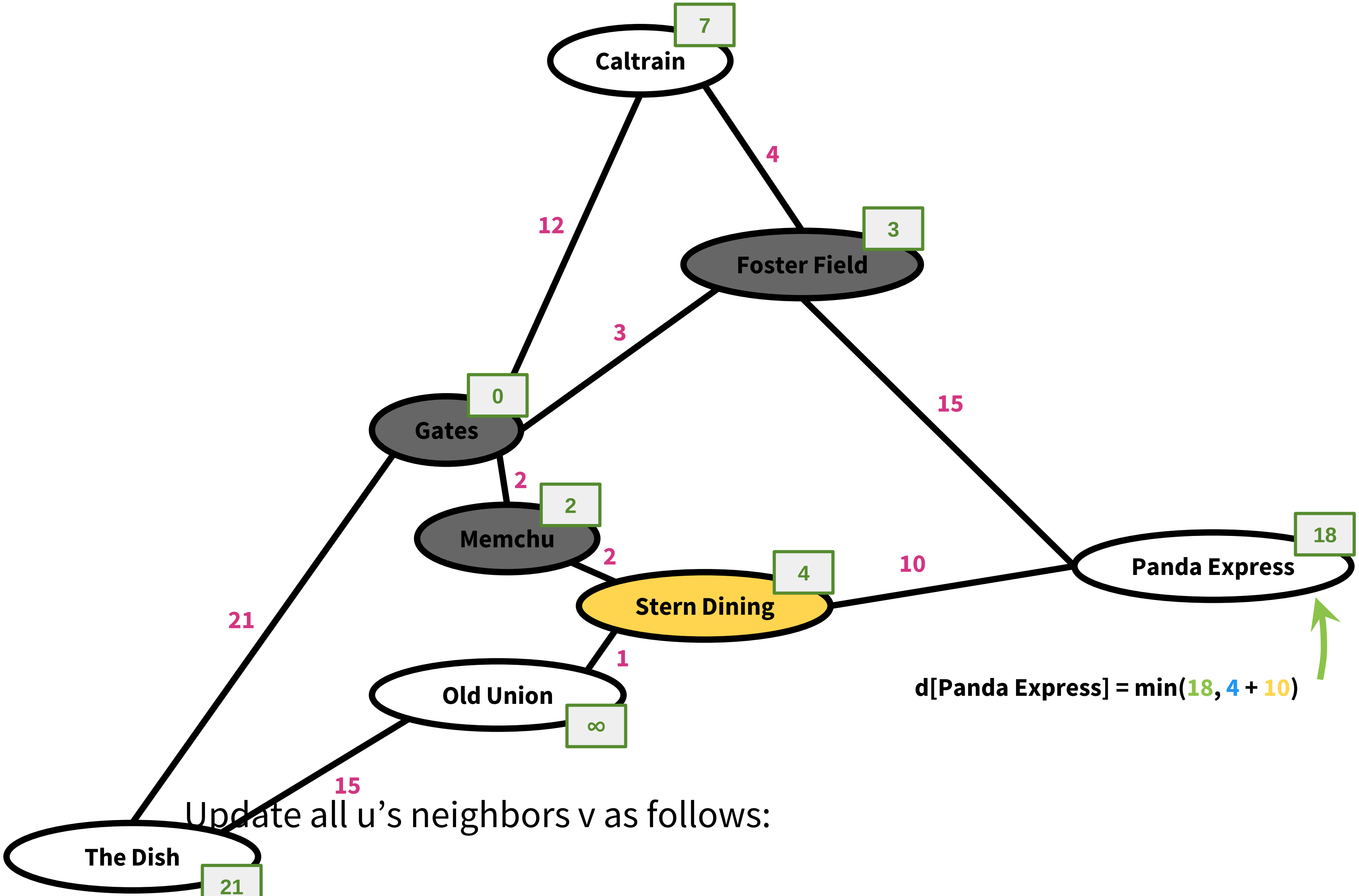
Dijkstra's Algorithm



Update all u's neighbors v as follows:

$d[v] = \min(d[v], d[u] + w(u, v))$

Dijkstra’s Algorithm

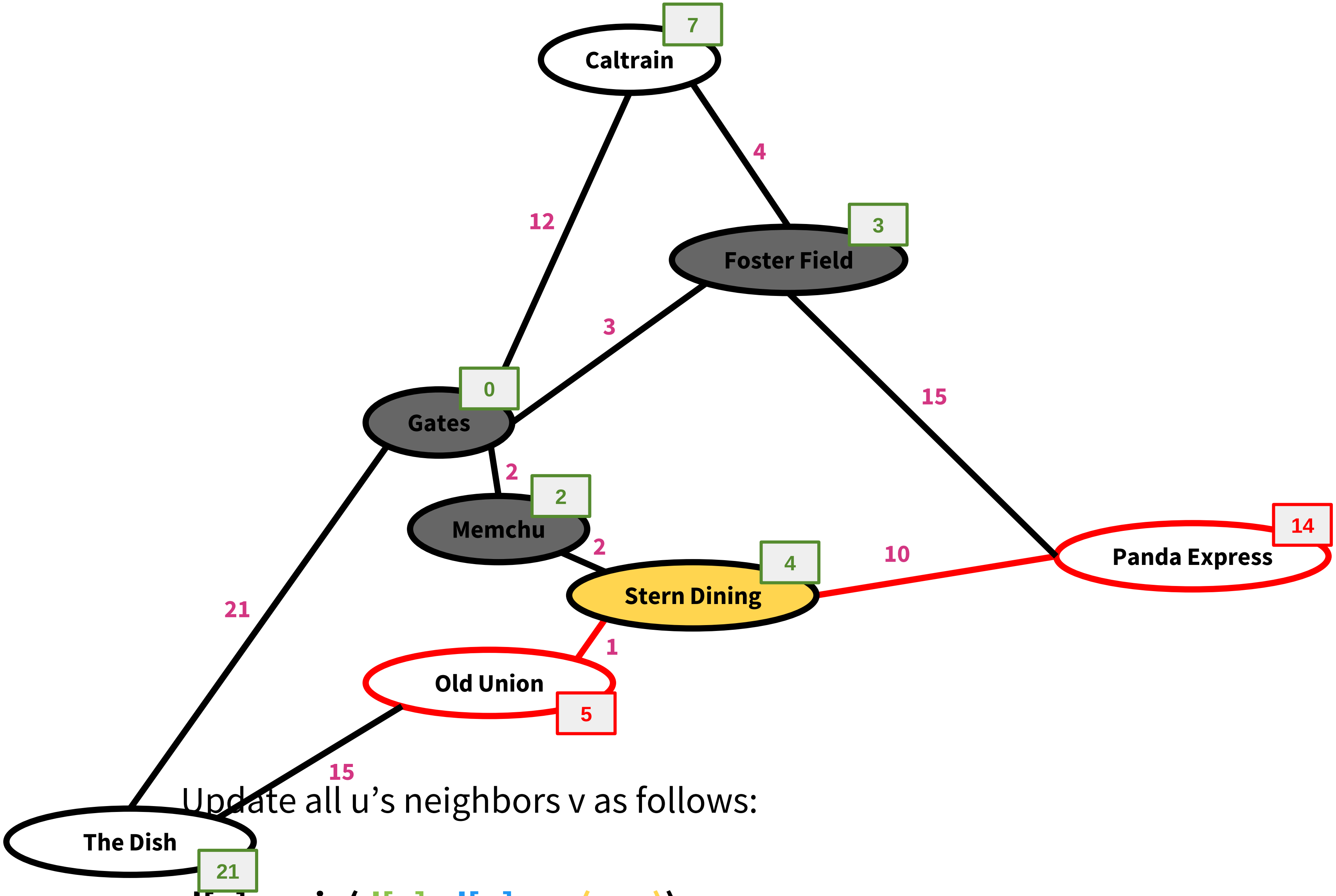


d[Panda Express] = min(18, 4 + 10)

Update all u’s neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

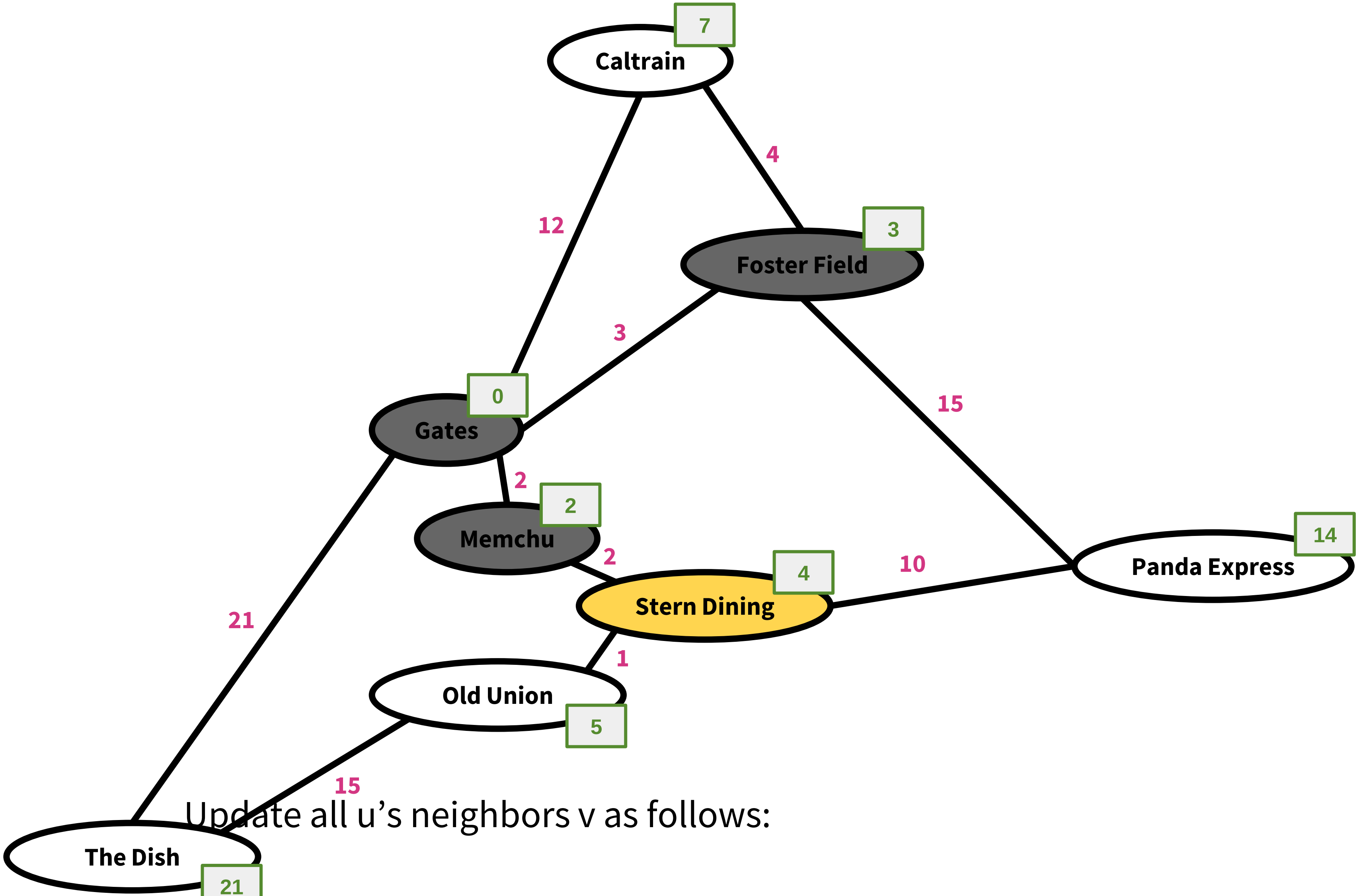
Dijkstra's Algorithm



Update all u's neighbors v as follows:

$$d[v] = \min(d[v], d[u] + w(u, v))$$

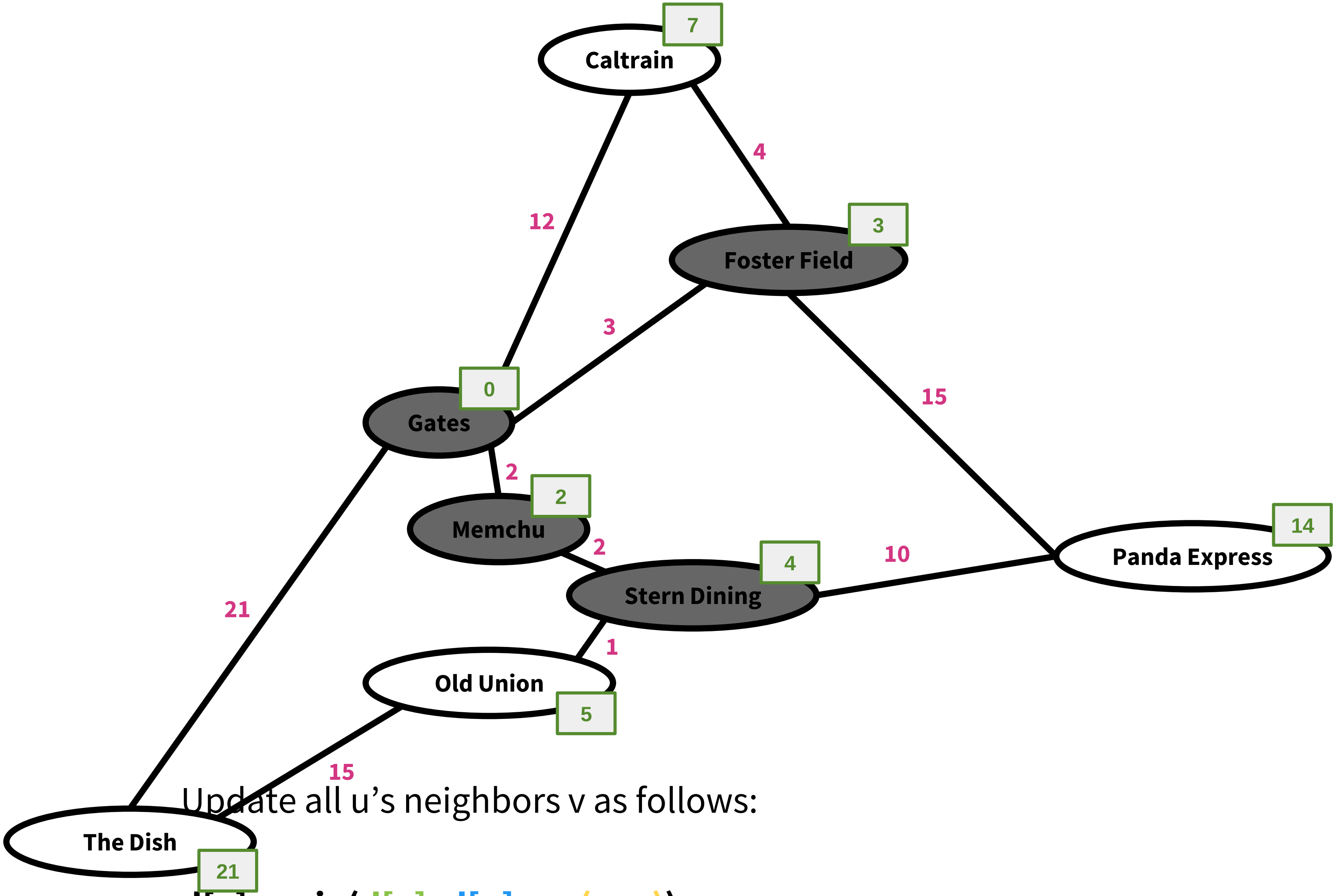
Dijkstra's Algorithm



Update all u's neighbors v as follows:

$$d[v] = \min(d[v],\ d[u] + w(u, v))$$

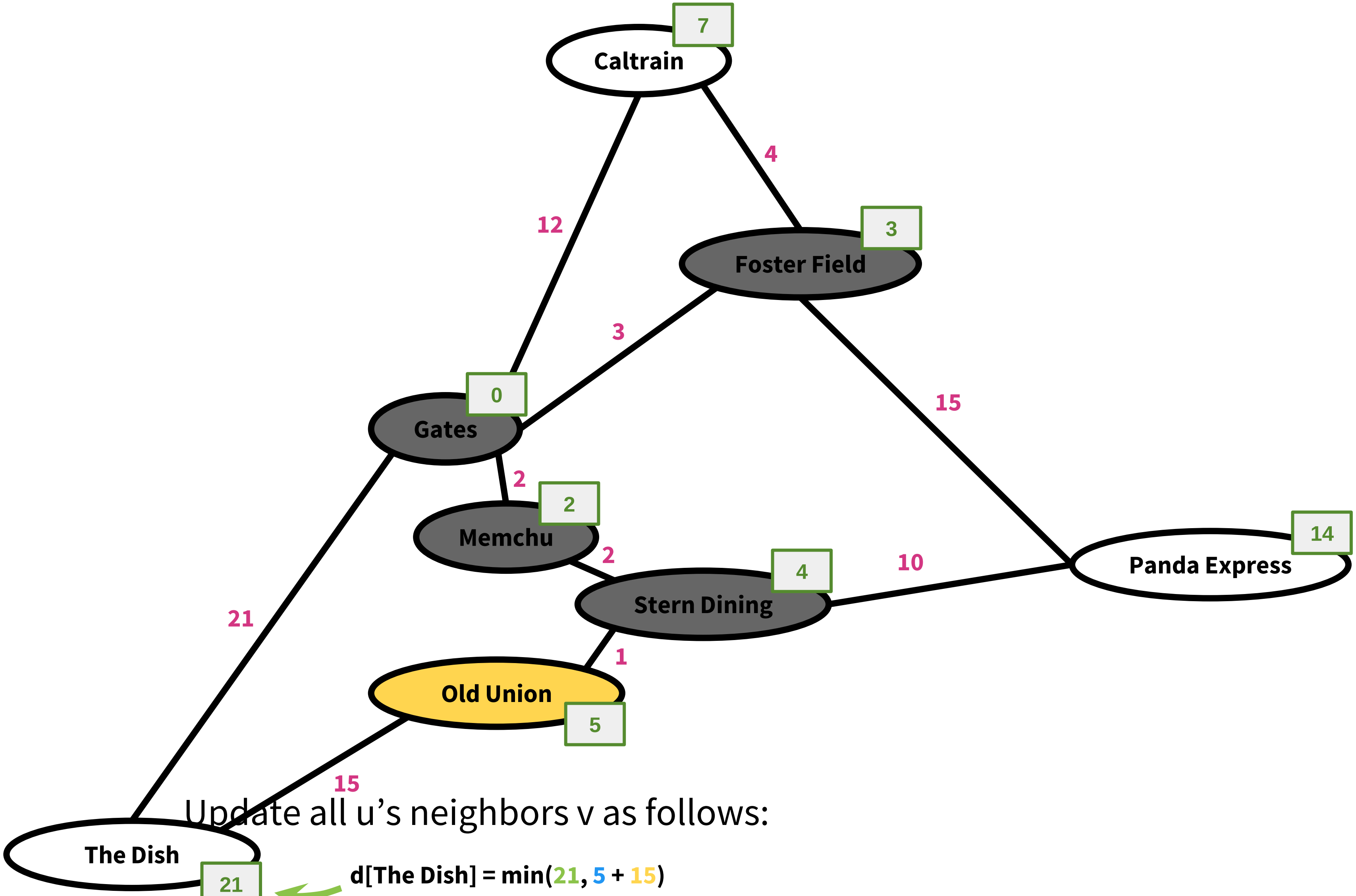
Dijkstra's Algorithm



Update all u's neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

Dijkstra's Algorithm

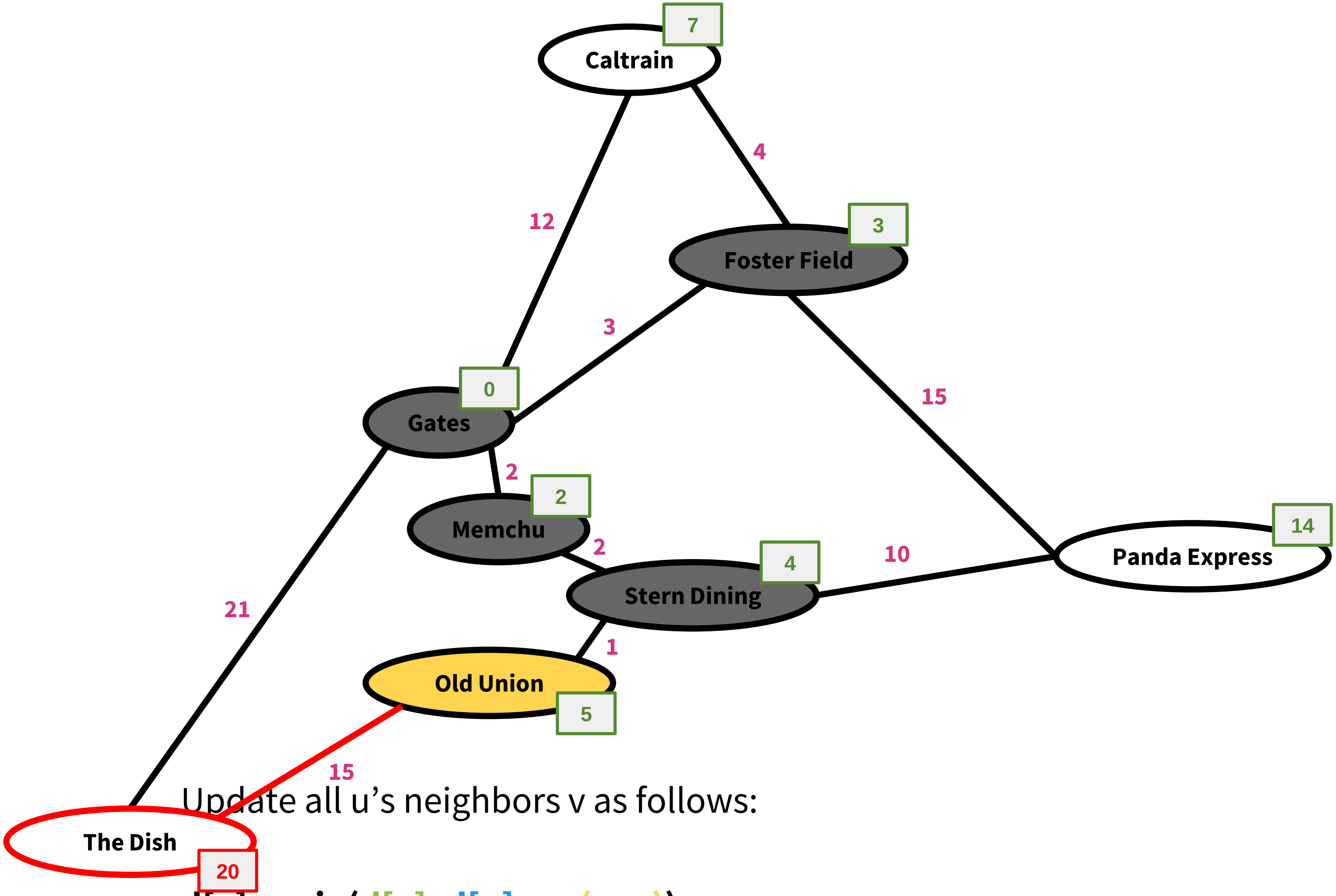


Update all u's neighbors v as follows:

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

d[v] = min(d[v], d[u] + w(u, v))

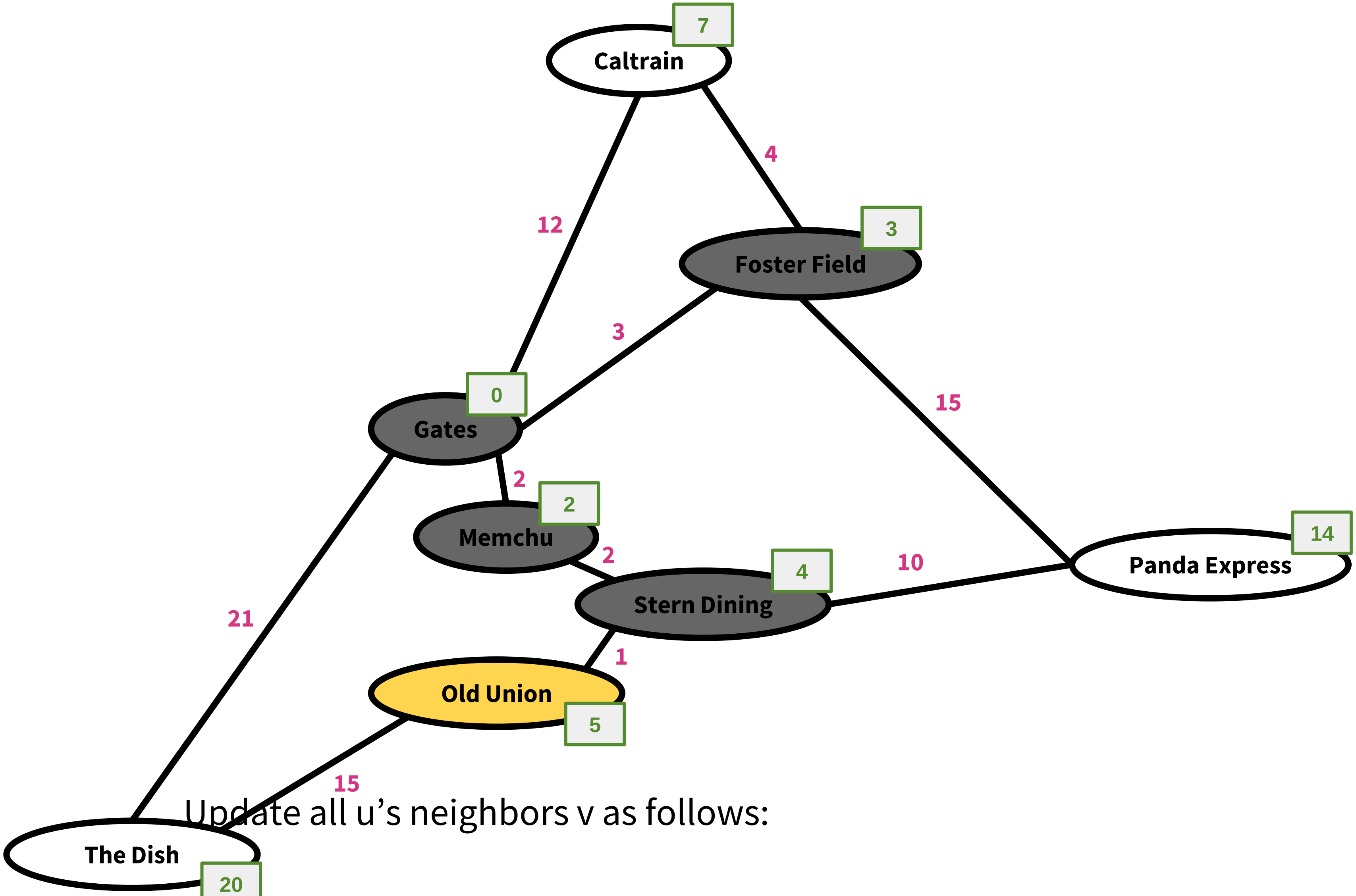
Dijkstra’s Algorithm



Update all u’s neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

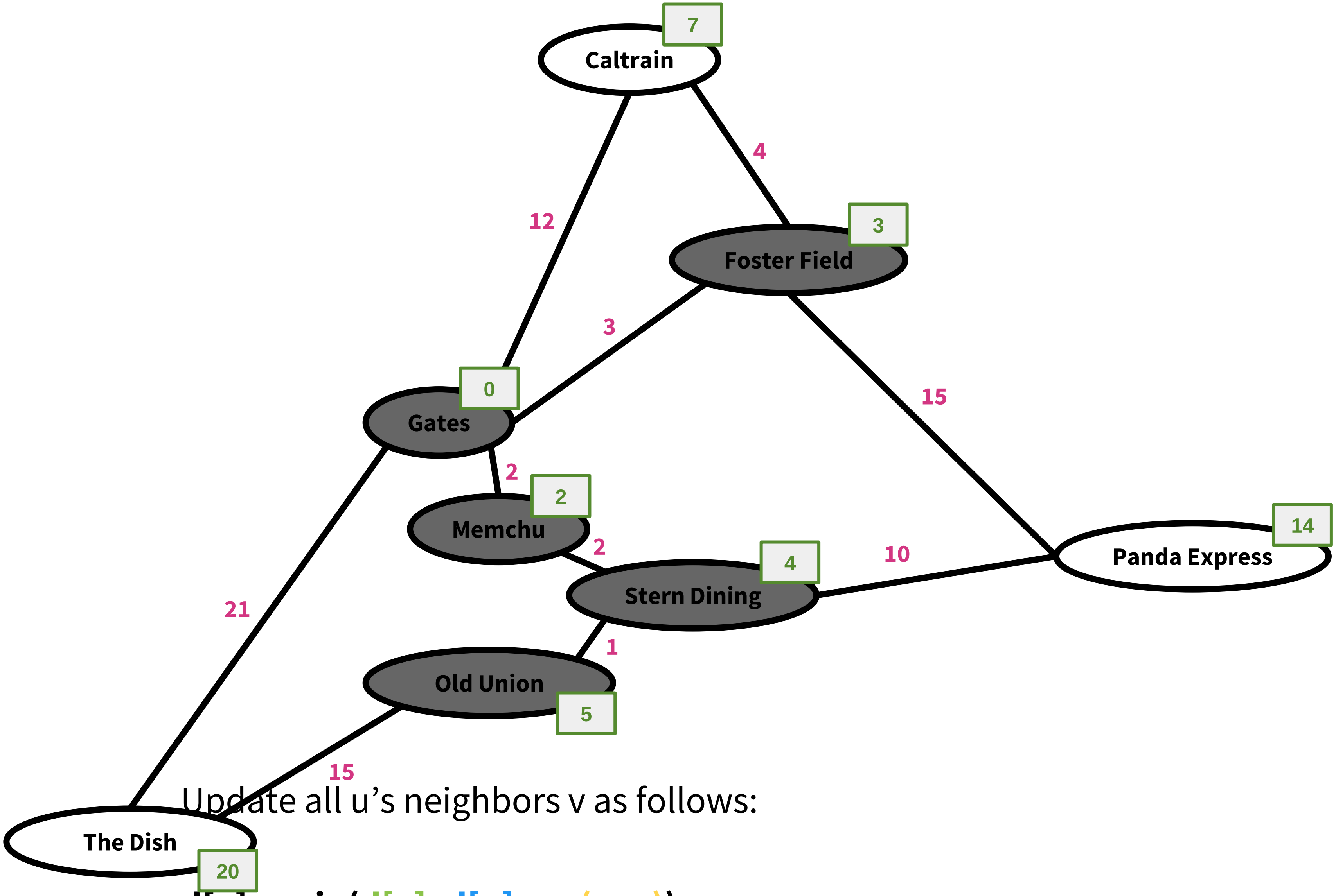
Dijkstra's Algorithm



Update all u's neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

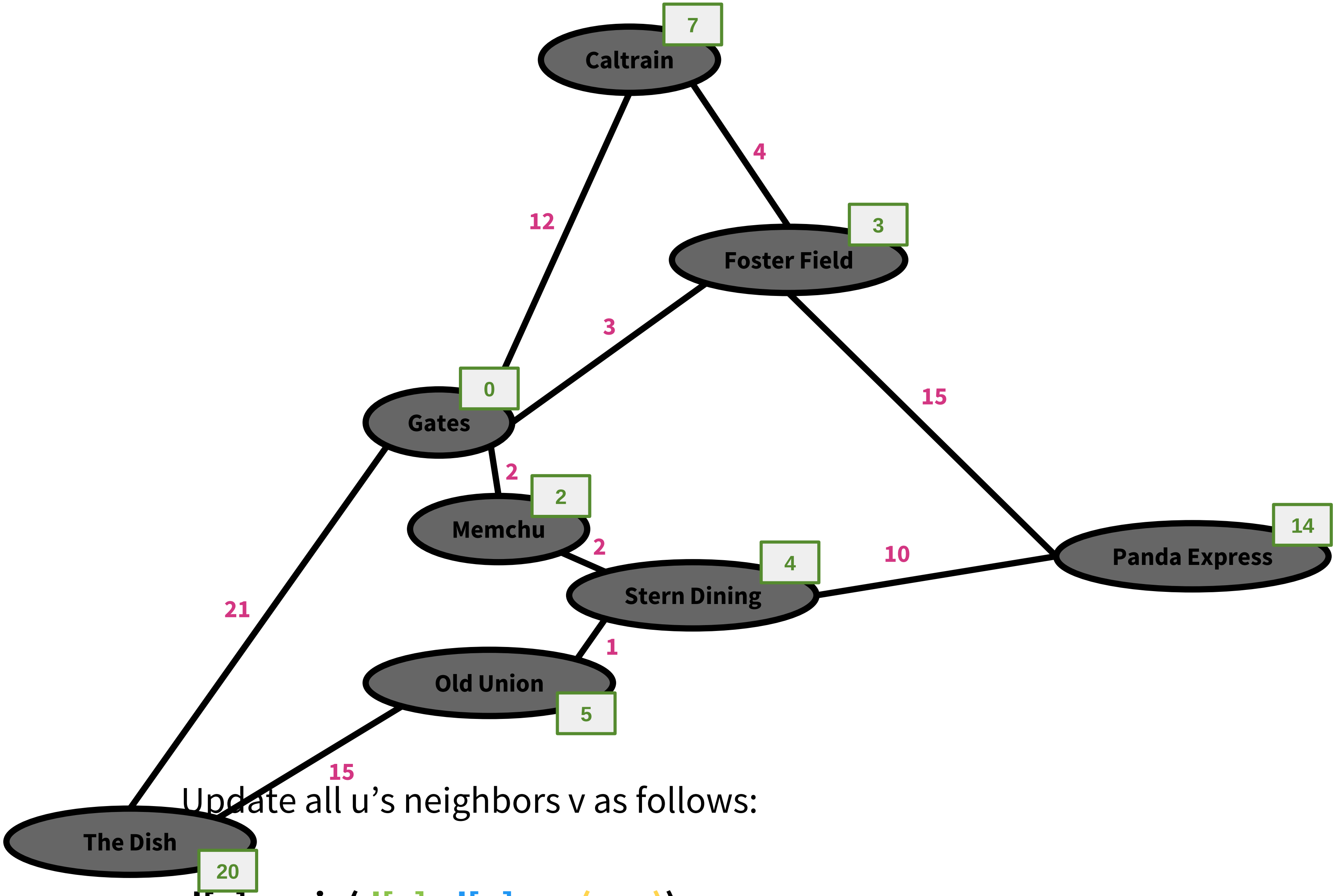
Dijkstra’s Algorithm



Update all u’s neighbors v as follows:

d[v] = min(d[v], d[u] + w(u, v))

Dijkstra's Algorithm



Update 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 single source.

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 marked “done”, d[v] = d(s, v).

Together, claims 1 and 2 imply the theorem.

Dijkstra's Algorithm

Why does this work?

Let s be the single source.

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 marked "done", 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.

Dijkstra’s Algorithm

Why does this work?

Let s be the single source.

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 marked “done”, 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’re sure about “done” about v, d[v] = d(s, v).

Dijkstra’s Algorithm

Why does this work?

Let s be the single source.

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 marked “done”, 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’re sure about “done” about v, d[v] = d(s, v).

All vertices are eventually “done” (stopping condition in algorithm).

Dijkstra's Algorithm

Why does this work?

Let s be the single source.

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 marked "done", 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're sure about "done" about v, d[v] = d(s, v).

All vertices are eventually "done" (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.

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).

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.

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], v[u] + w(u, v))** ≥ d(s, v).

By induction,
d[v] ≥ d(s, v)

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

Thus, the induction holds for t + 1.

Dijkstra’s Algorithm

Why does this work?

Claim 2: When a vertex v gets marked “done”, d[v] = d(s, v).

Proof:

We proceed by induction on t, the number of vertices marked as “done.”

Dijkstra's Algorithm

Why does this work?

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

Proof:

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

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

Dijkstra's Algorithm

Why does this work?

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

Proof:

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

For the base case, note that after s is marked as "done", 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 "done", d[v] = d(s, v). Let x be the vertex with minimum distance estimate. We must prove d[x] = d(s, x).

Dijkstra's Algorithm

Why does this work?

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

Proof, cont.:

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

Dijkstra's Algorithm

Why does this work?

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

Proof, cont.:

We proceed by contradiction. Suppose $d[x] \neq d(s, x)$.

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

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

Dijkstra’s Algorithm

Why does this work?

Claim 2: When a vertex v gets marked “done”, $d[v] = d(s, v)$.

Proof, cont.:

We proceed by contradiction. Suppose $d[x] \neq d(s, x)$.

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

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

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

Therefore, $d[z] < d[x]$. But this can’t be the case. Why not? Since $d[z] < d[x]$ and x is the vertex with minimum distance estimate, z must be already marked “done.”

Dijkstra's Algorithm

Why does this work?

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

Proof, cont.:

Since z is already marked "done," the edges out of z, including the edge (z, z') (where z' is also on p) have been relaxed by the algorithm

Dijkstra's Algorithm

Why does this work?

Claim 2: When a vertex v gets marked “done”, d[v] = d(s, v).

Proof, cont.:

Since z is already marked “done,” the edges out of z, including the edge (z, z’) (where z’ is also on p) have been relaxed by the algorithm i.e.
d[z’] ≤ d(s, z) + w(z, z’) = d(s, z’) since z is on the shortest path from s to z’ and the distance estimate of z’ must be correct.

Dijkstra's Algorithm

Why does this work?

Claim 2: When a vertex v gets marked “done”, $d[v] = d(s, v)$.

Proof, cont.:

Since z is already marked “done,” the edges out of z, including the edge (z, z') (where z’ is also on p) have been relaxed by the algorithm i.e.
$d[z'] \leq d(s, z) + w(z, z') = d(s, z')$ since z is on the shortest path from s to z’ and the distance estimate of z’ must be correct.

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