Graph Algorithms II

Summer 2018 • Lecture 07/24

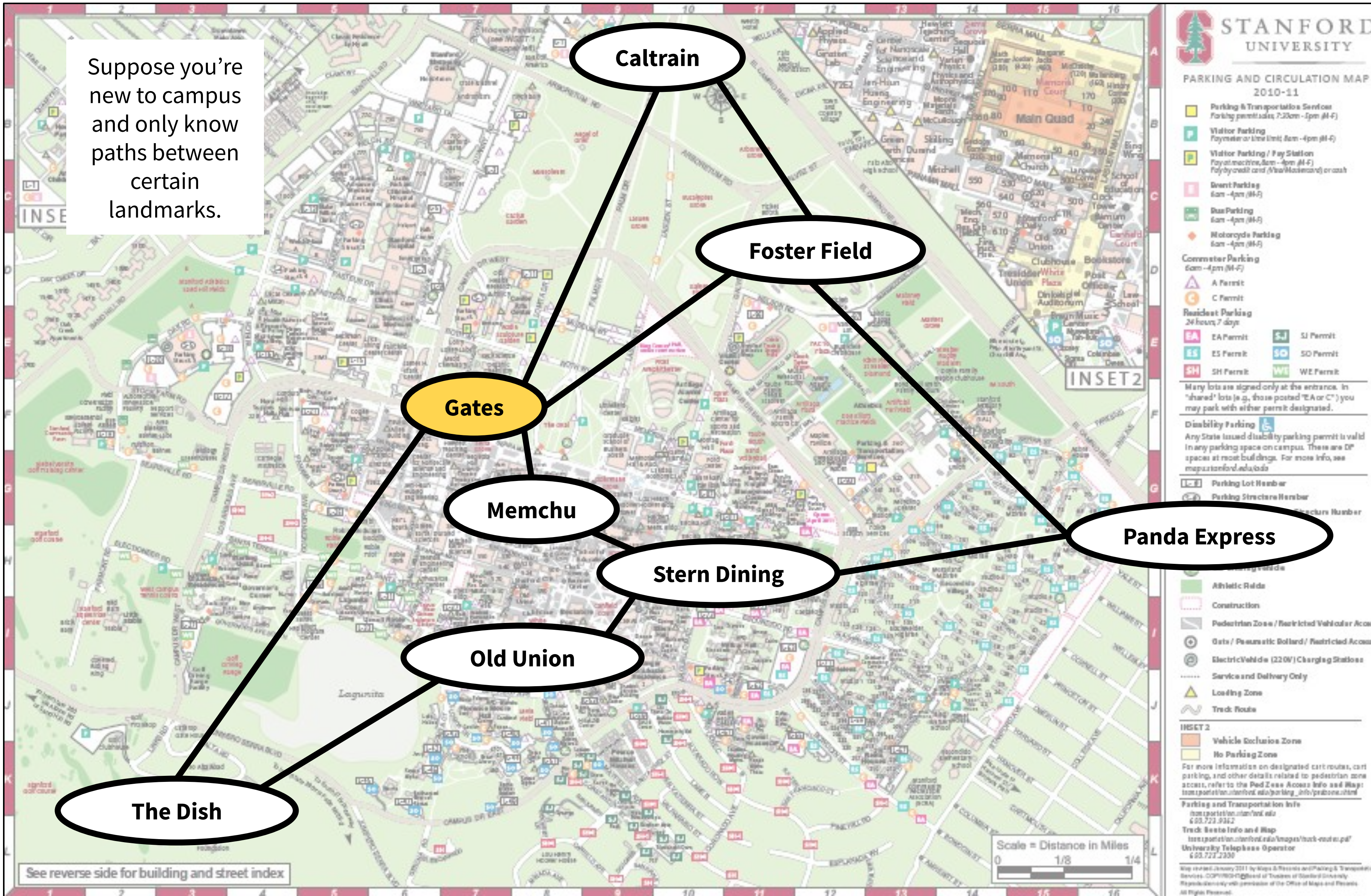
A Few Notes

- Midterm Review Session office hours
  - Today 12-1:20 p.m. in STLC 115
- Homework 3
  - Due today **(you can't use late days!)**

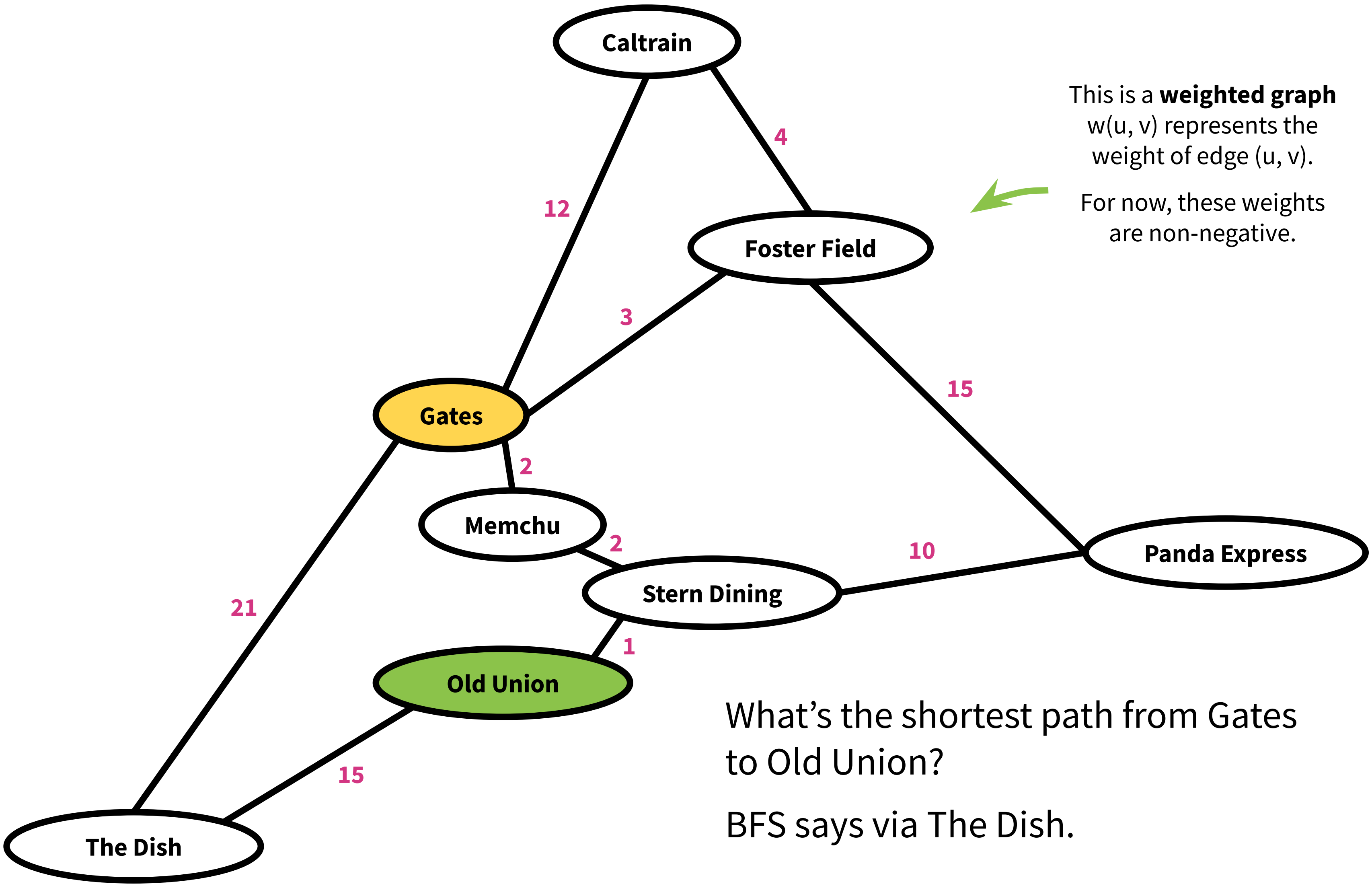
Dijkstra’s Algorithm

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?

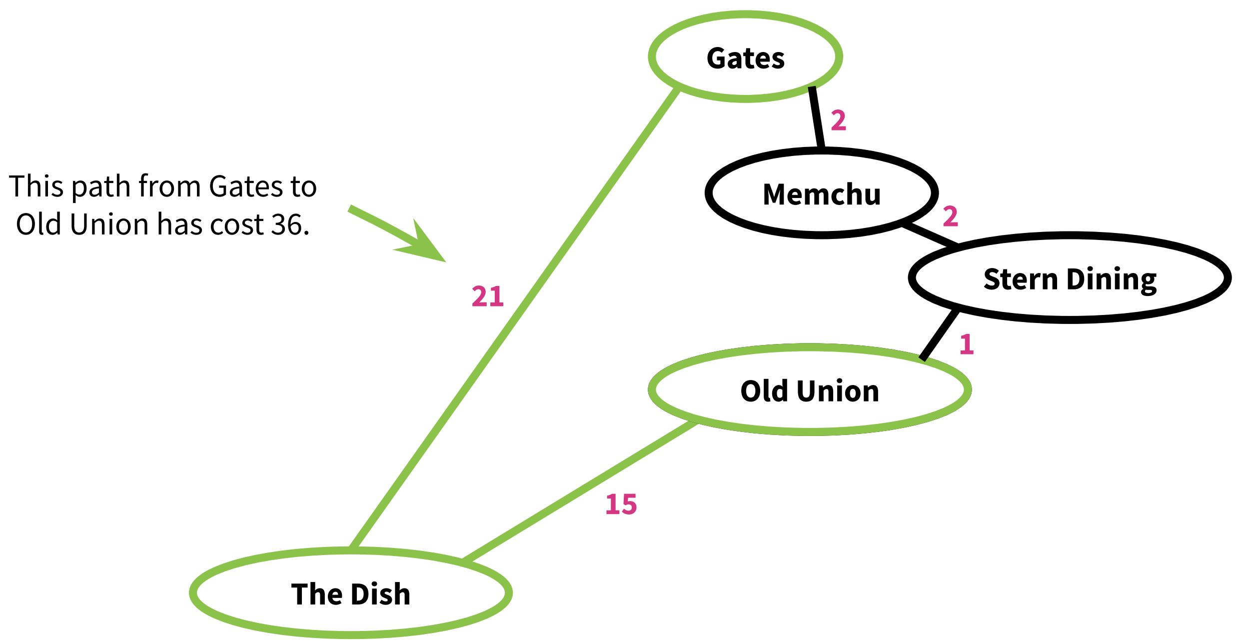
BFS says via The Dish.

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.

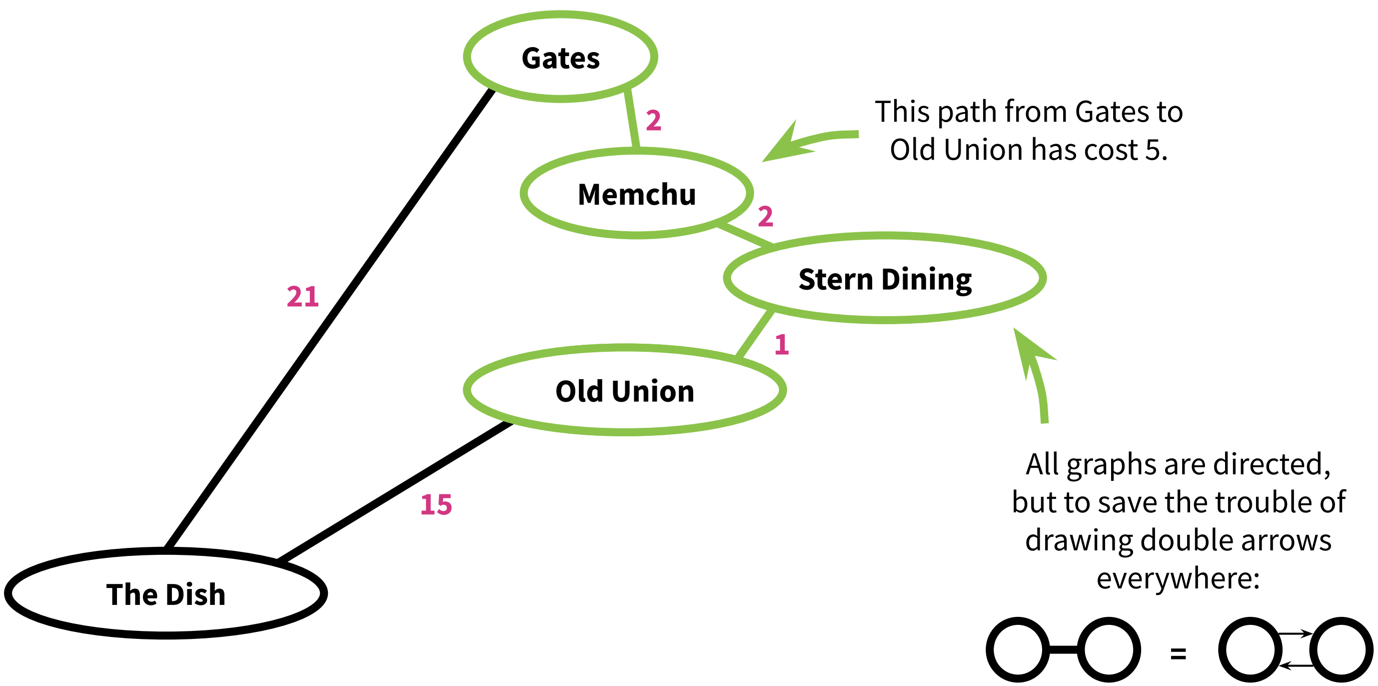


Shortest Path

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.

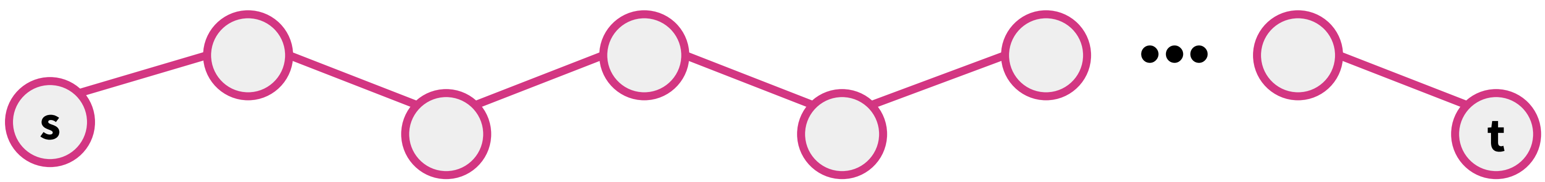
The shortest path is the one with the minimum cost.



Shortest Path

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

Intuition:

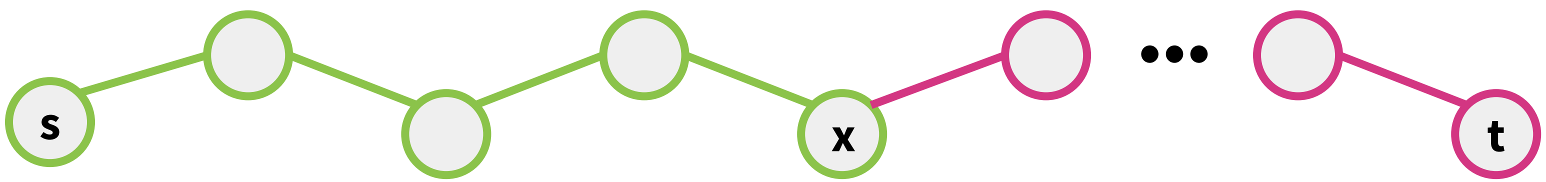


Suppose **this** is a shortest path from **s** to **t**.

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**.

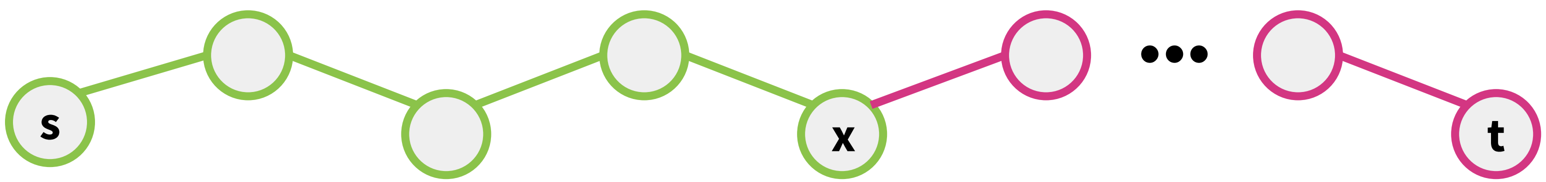
Then **this** is a shortest path from **s** to **x**.

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**.

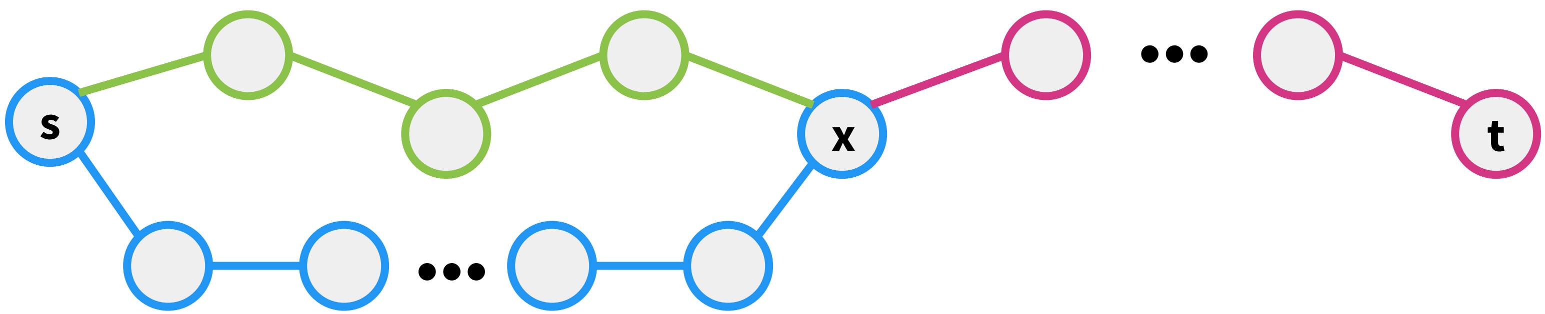
Then **this** is a shortest path from **s** to **x**.

Why? 🤔

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**.

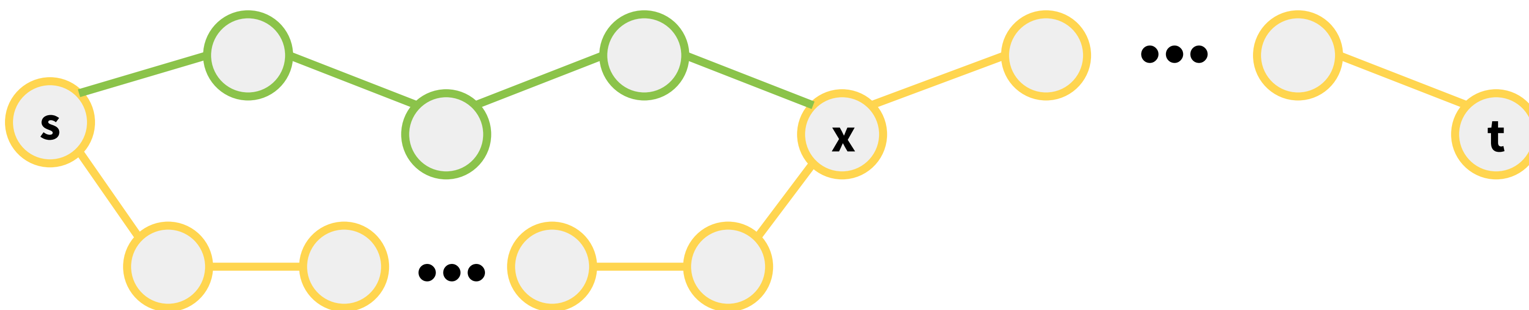
Then **this** is a shortest path from **s** to **x**.

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:



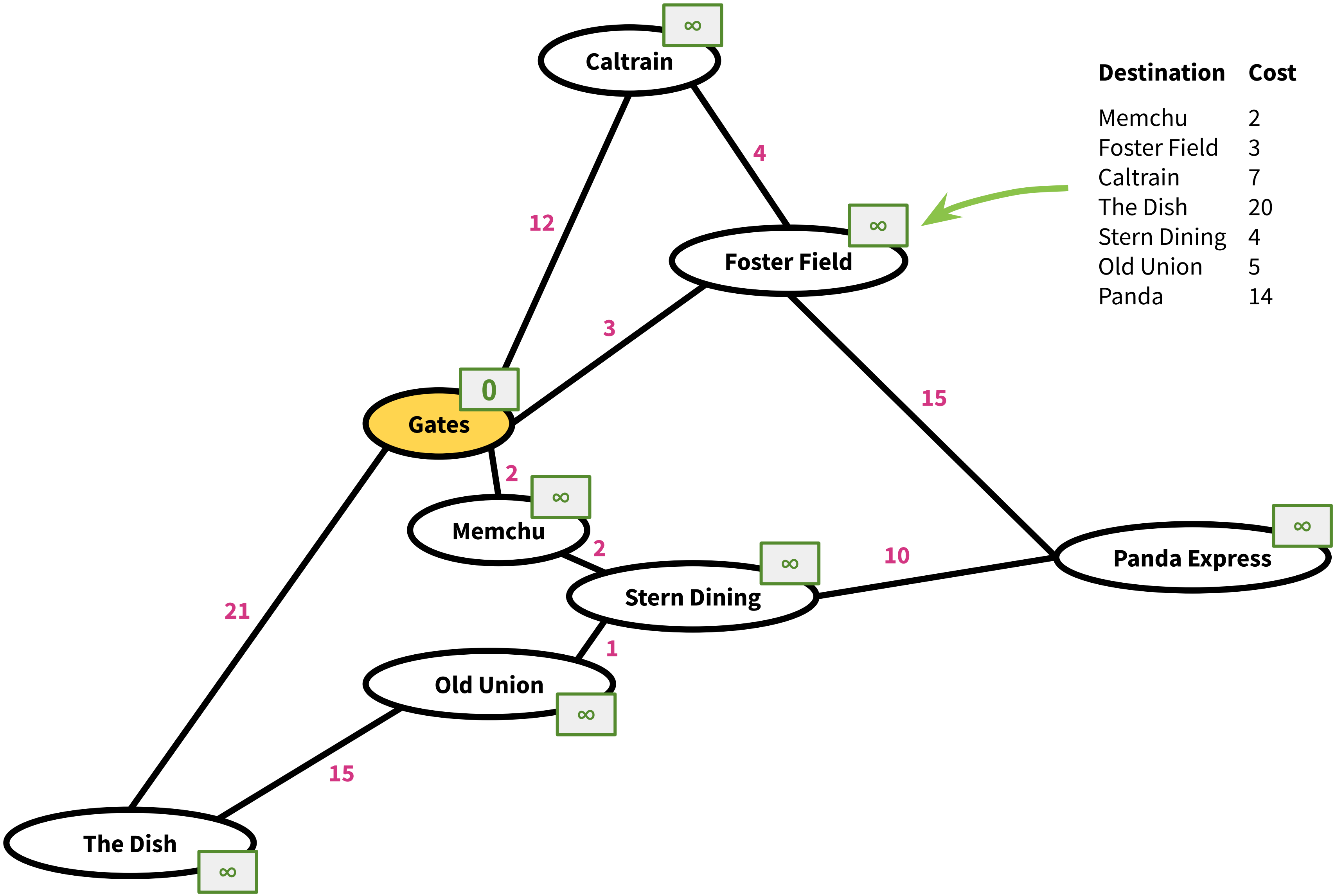
Suppose **this** is a shortest path from **s** to **t**.

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.

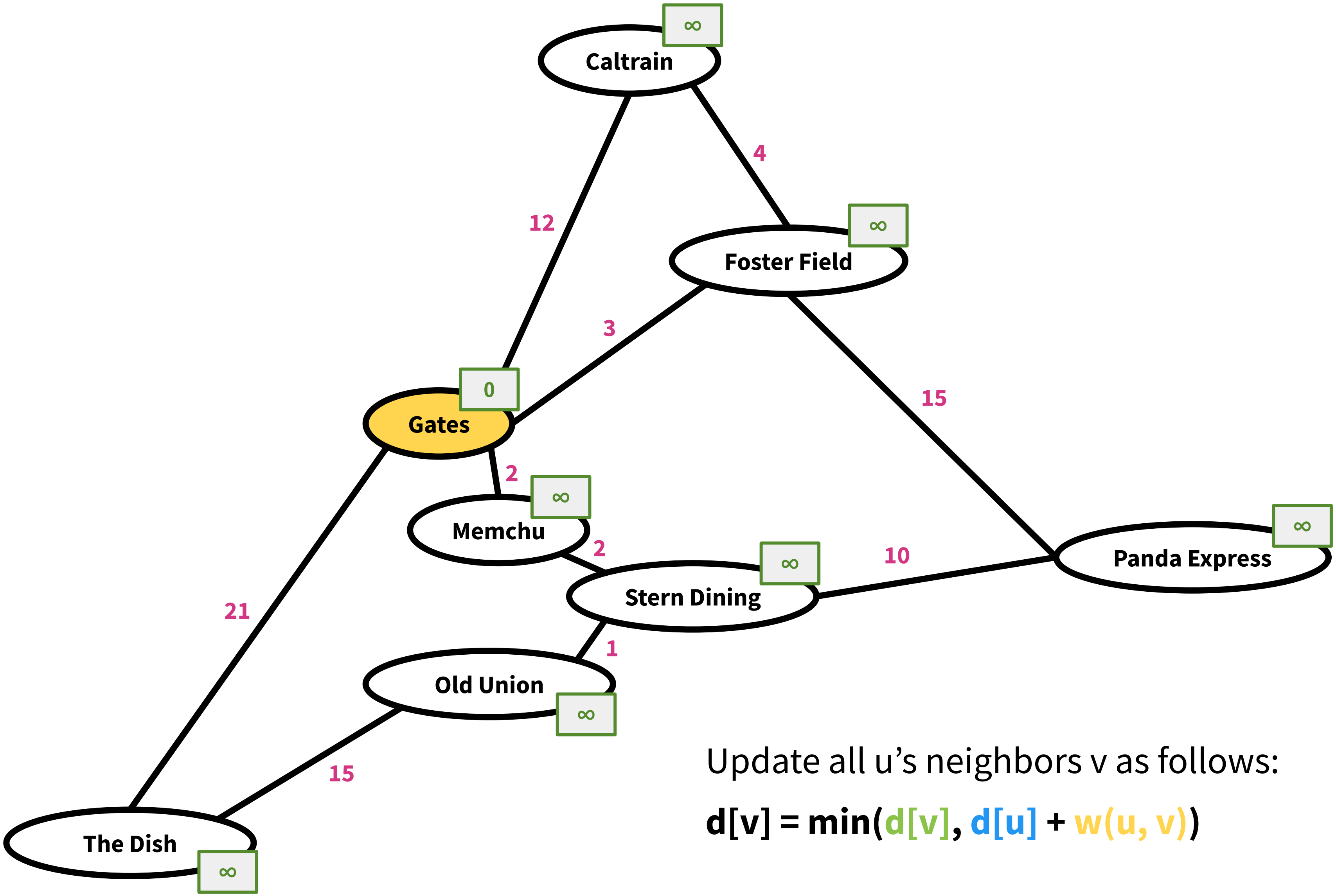
Application: Finding the shortest path from my computer to the desired server for packets using the Internet.

- 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.

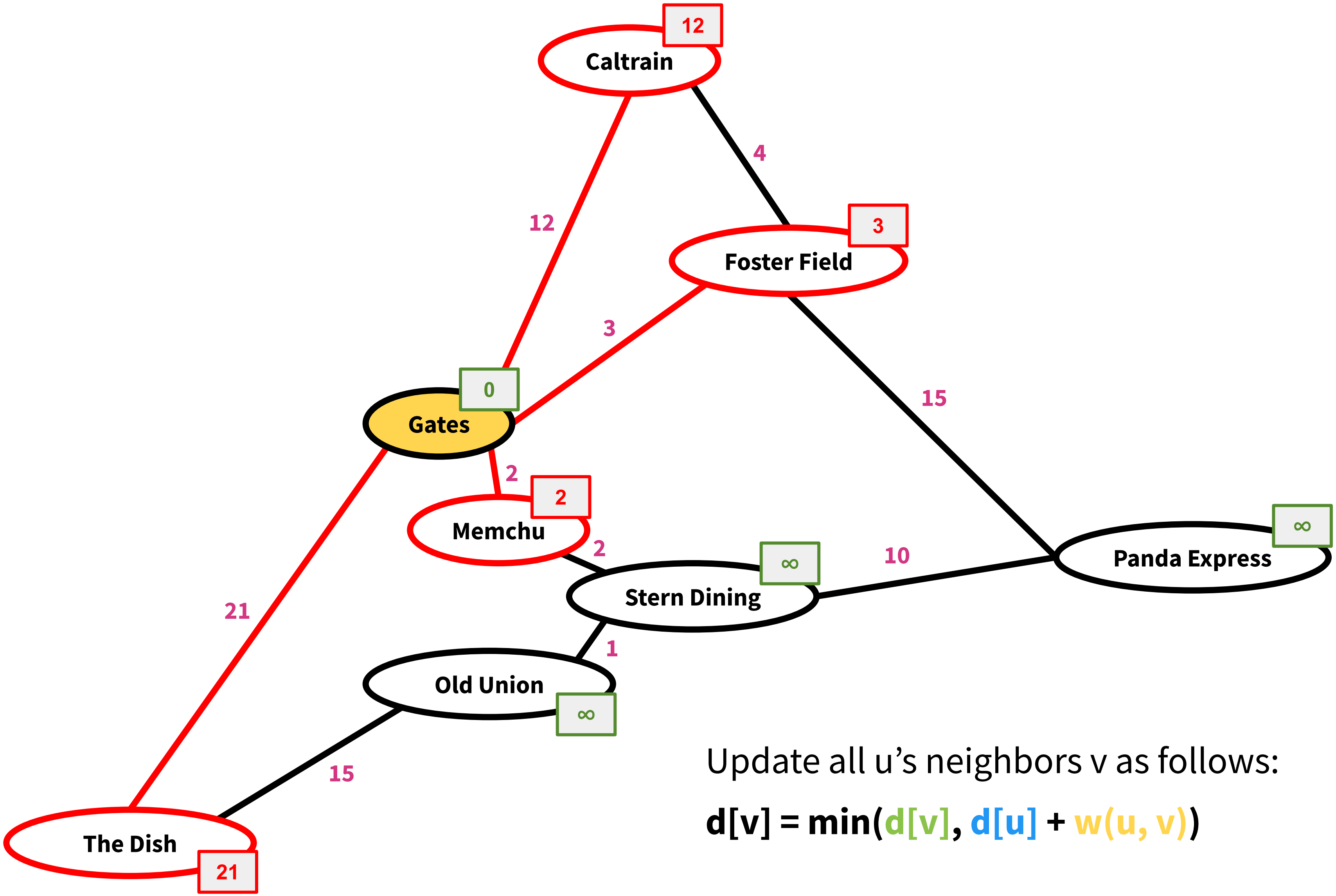
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

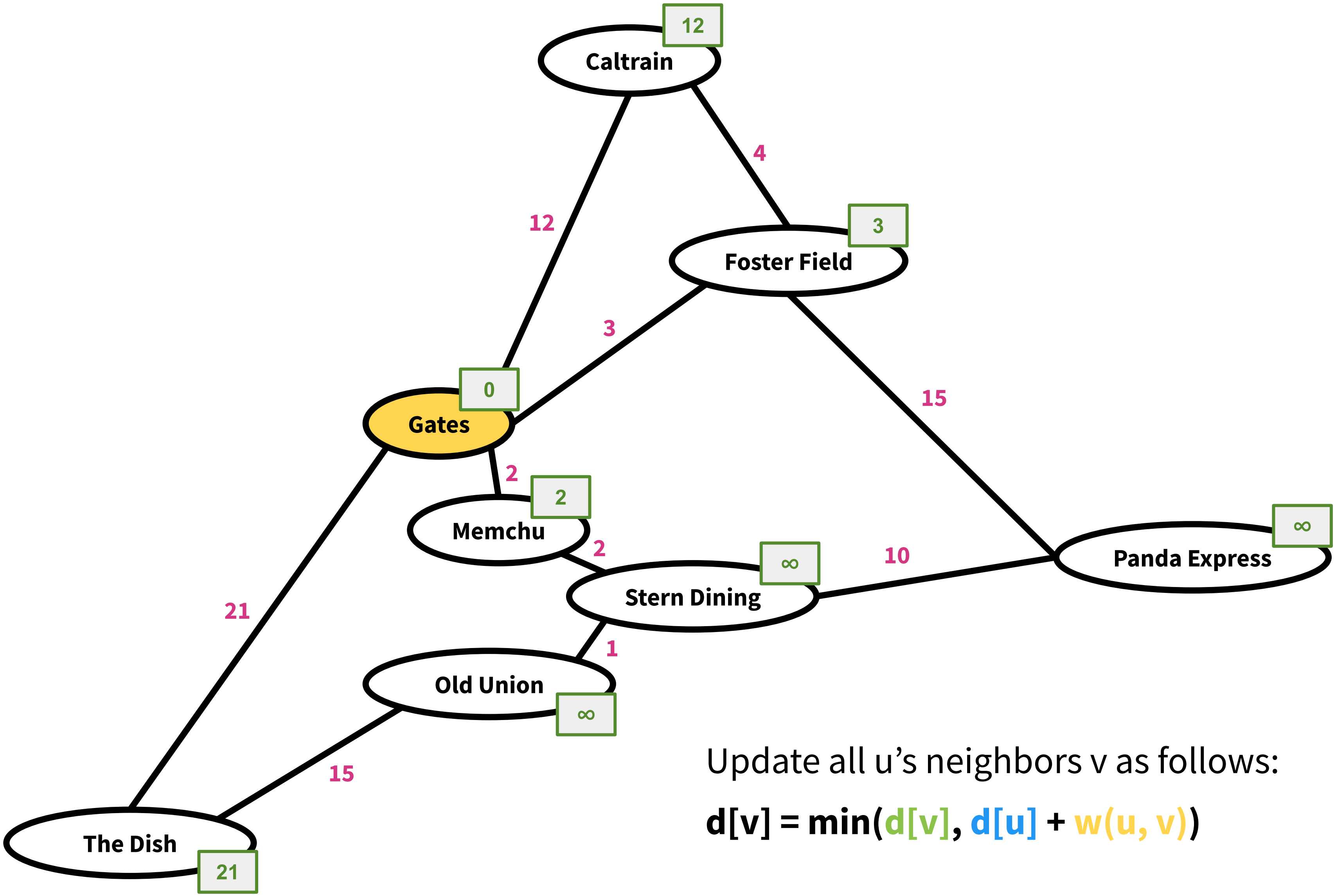
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

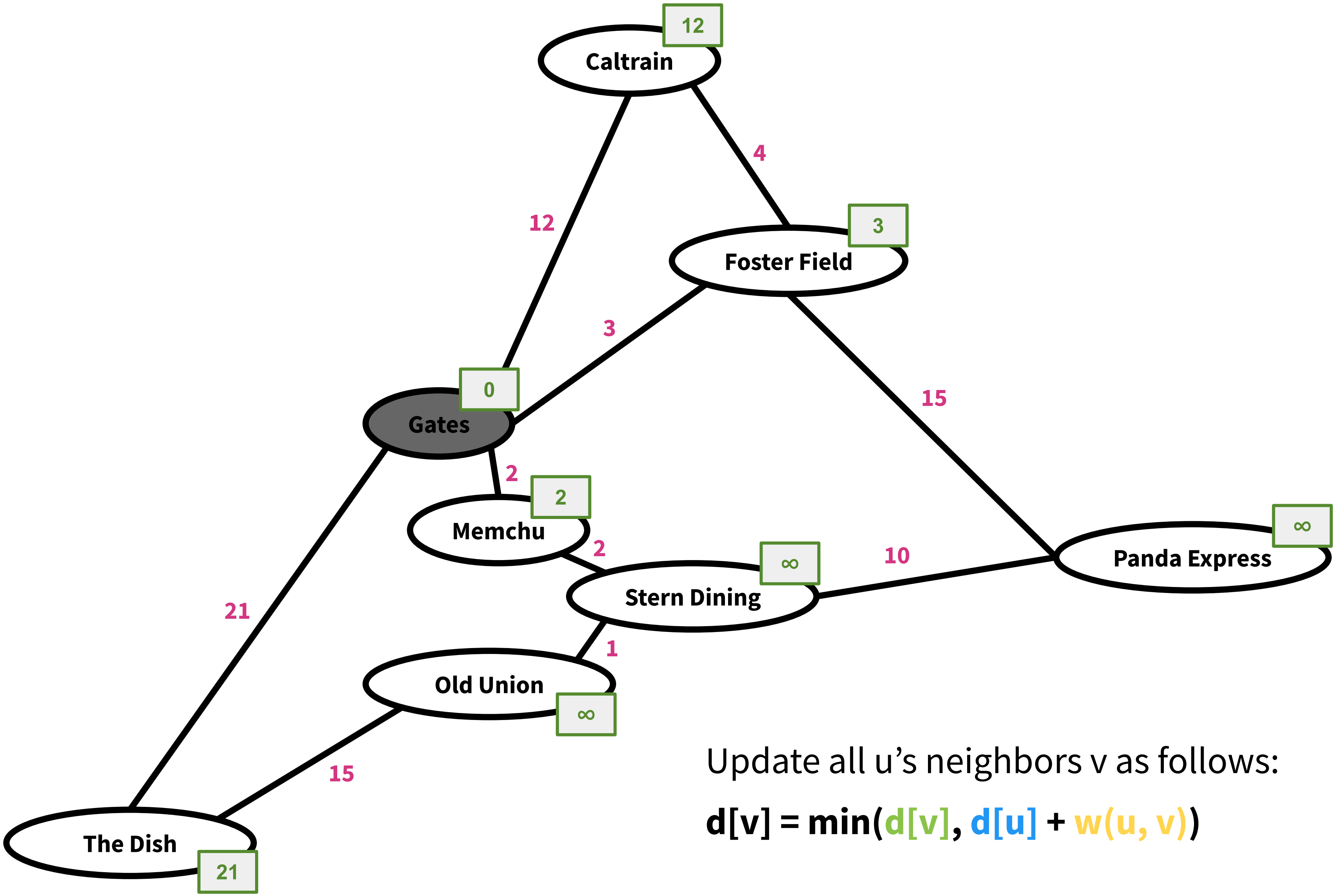
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

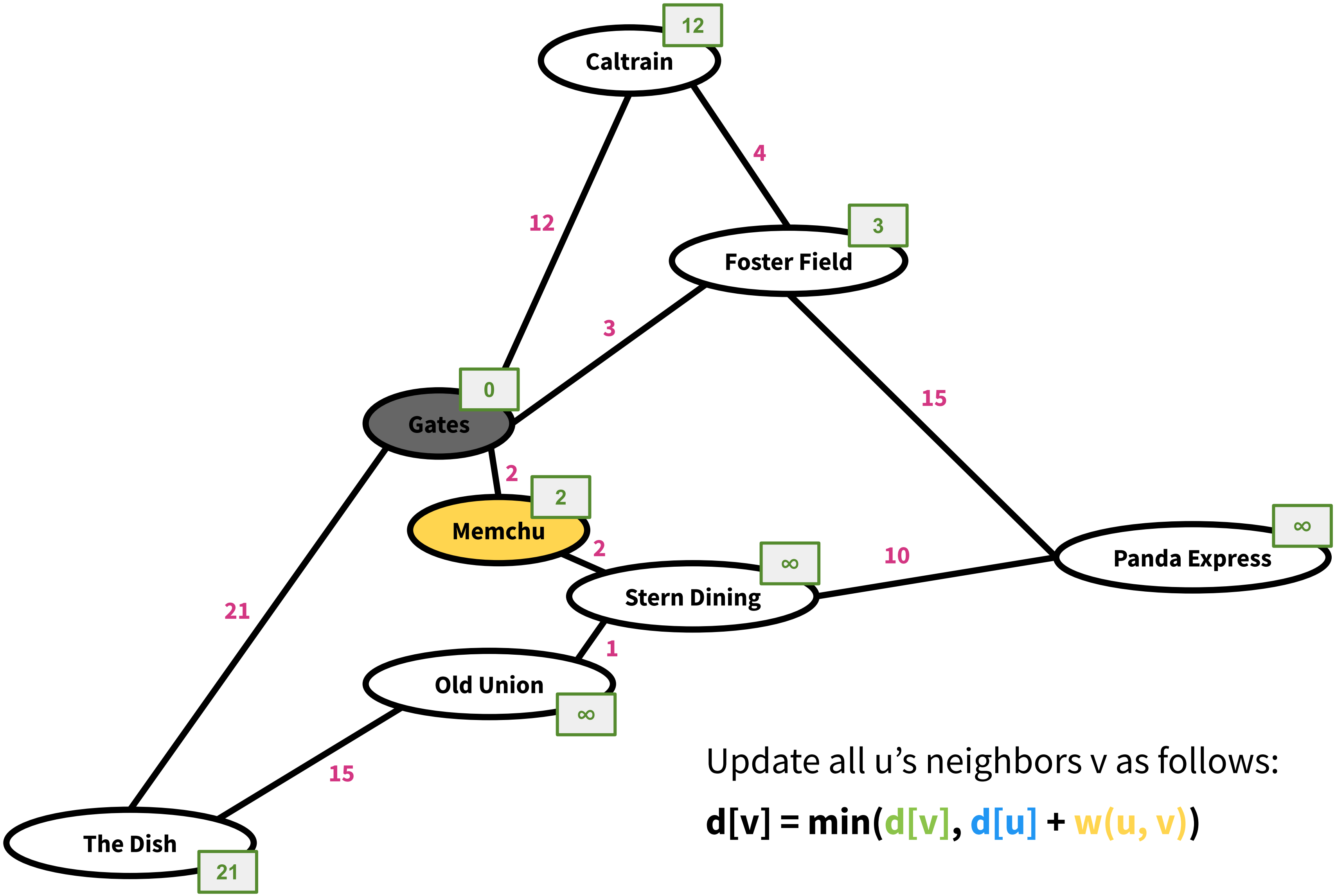
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

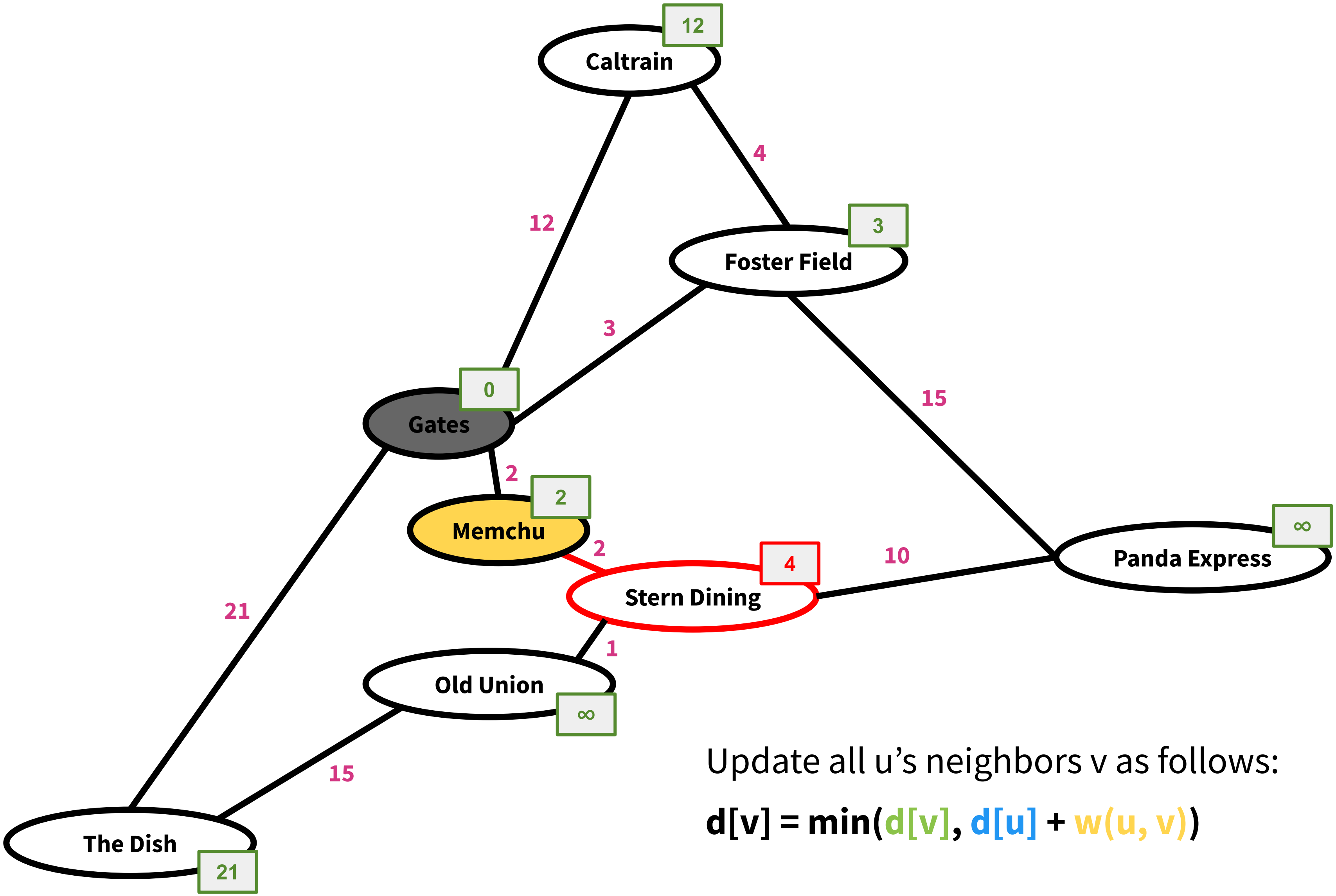
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

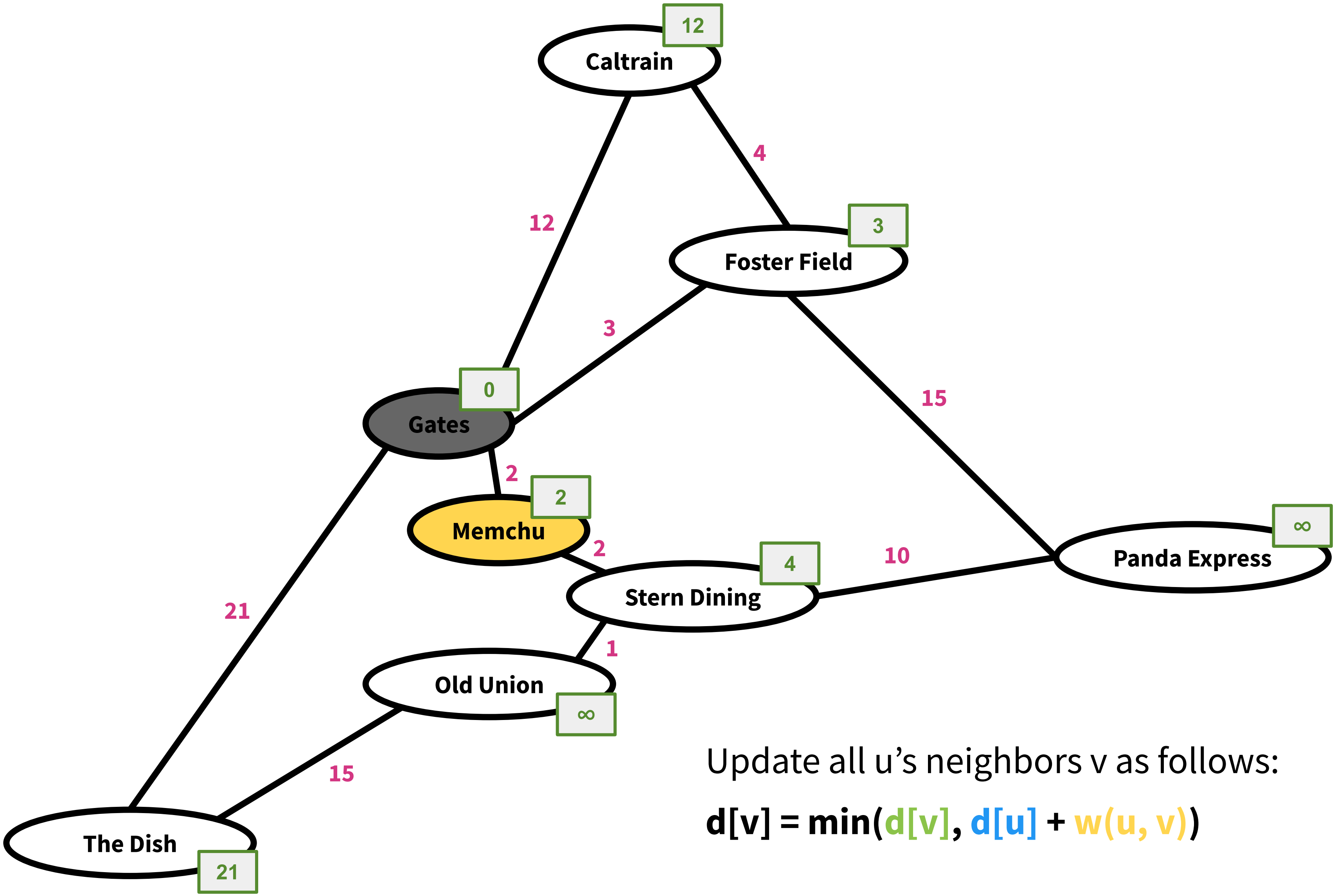
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

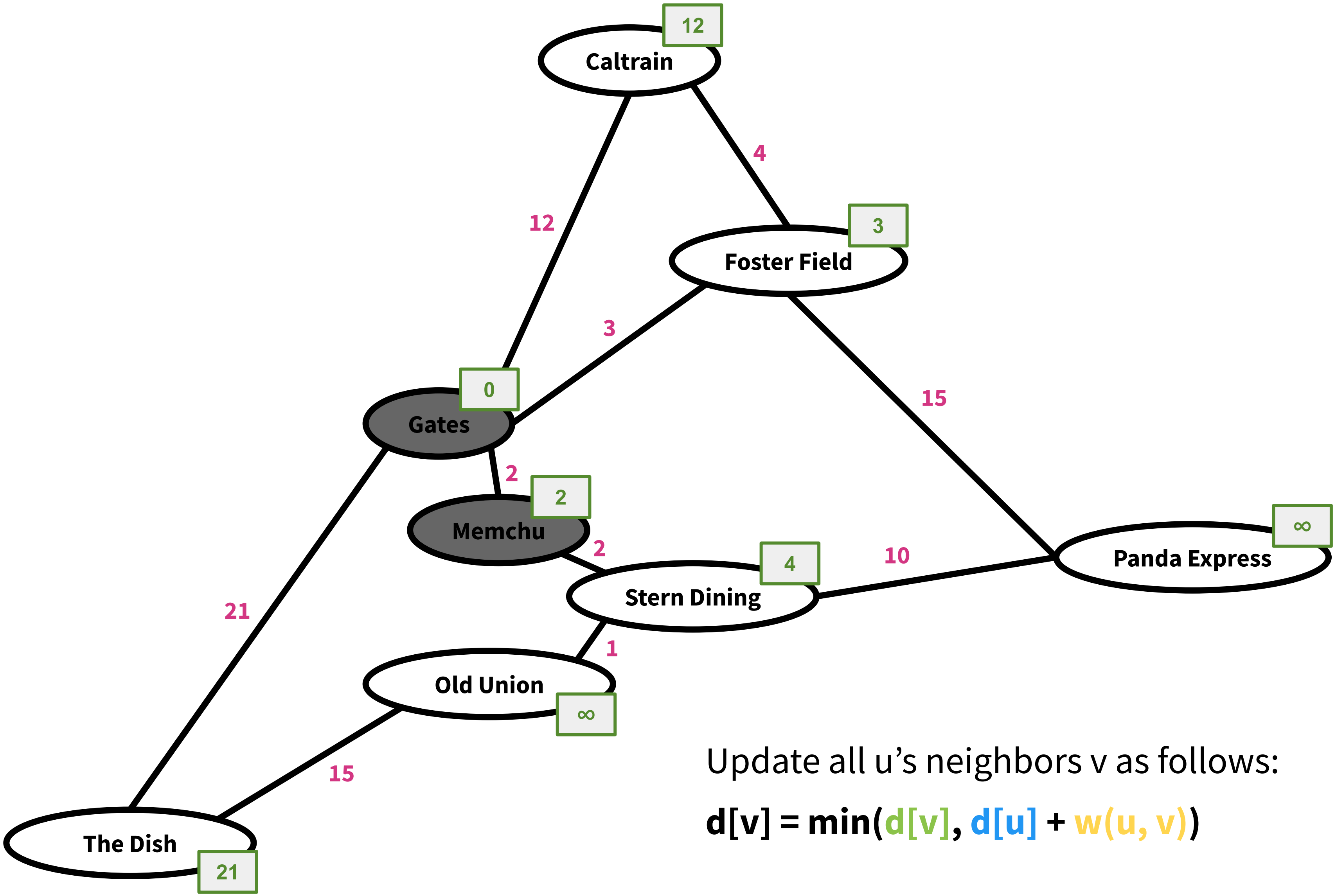
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

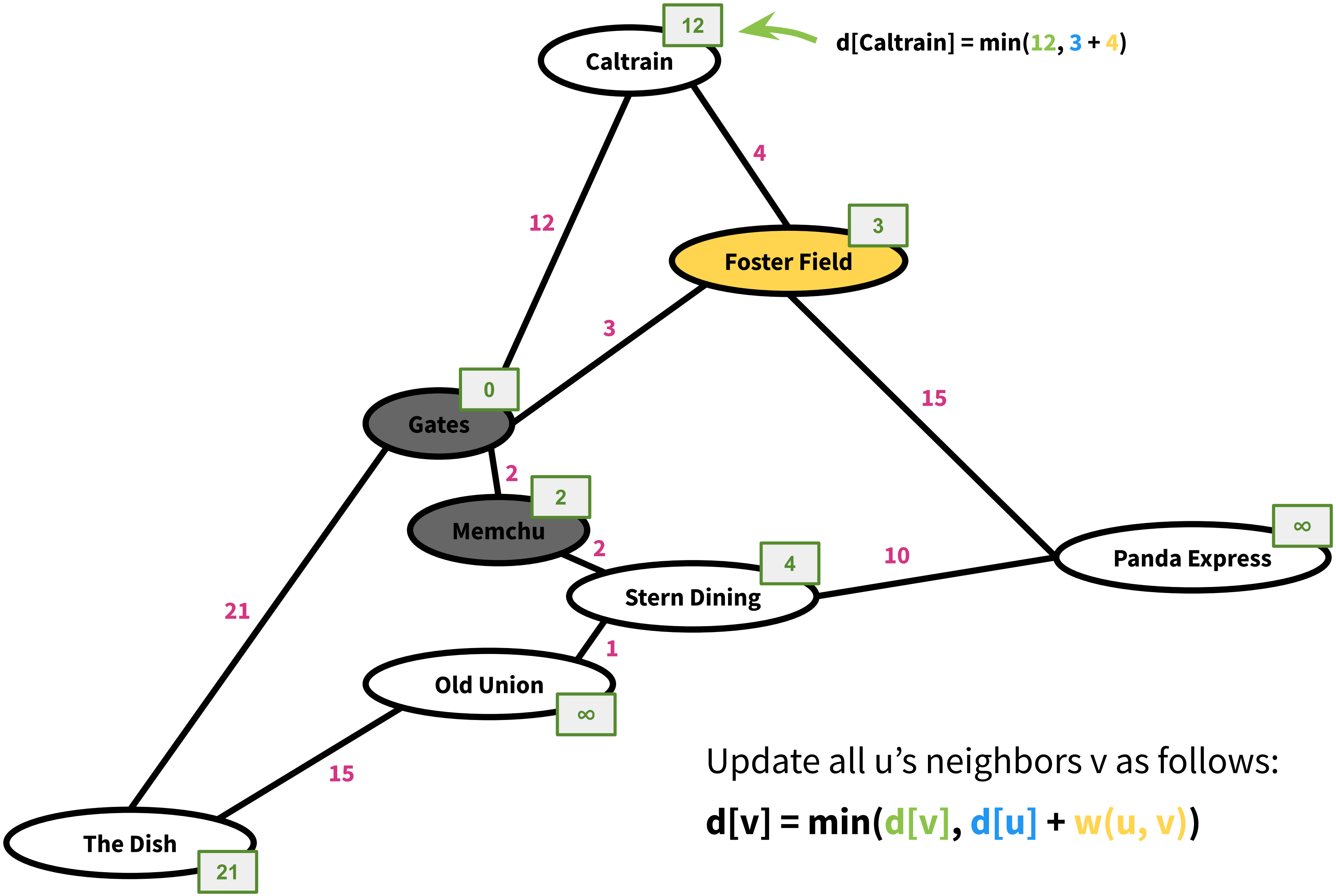
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

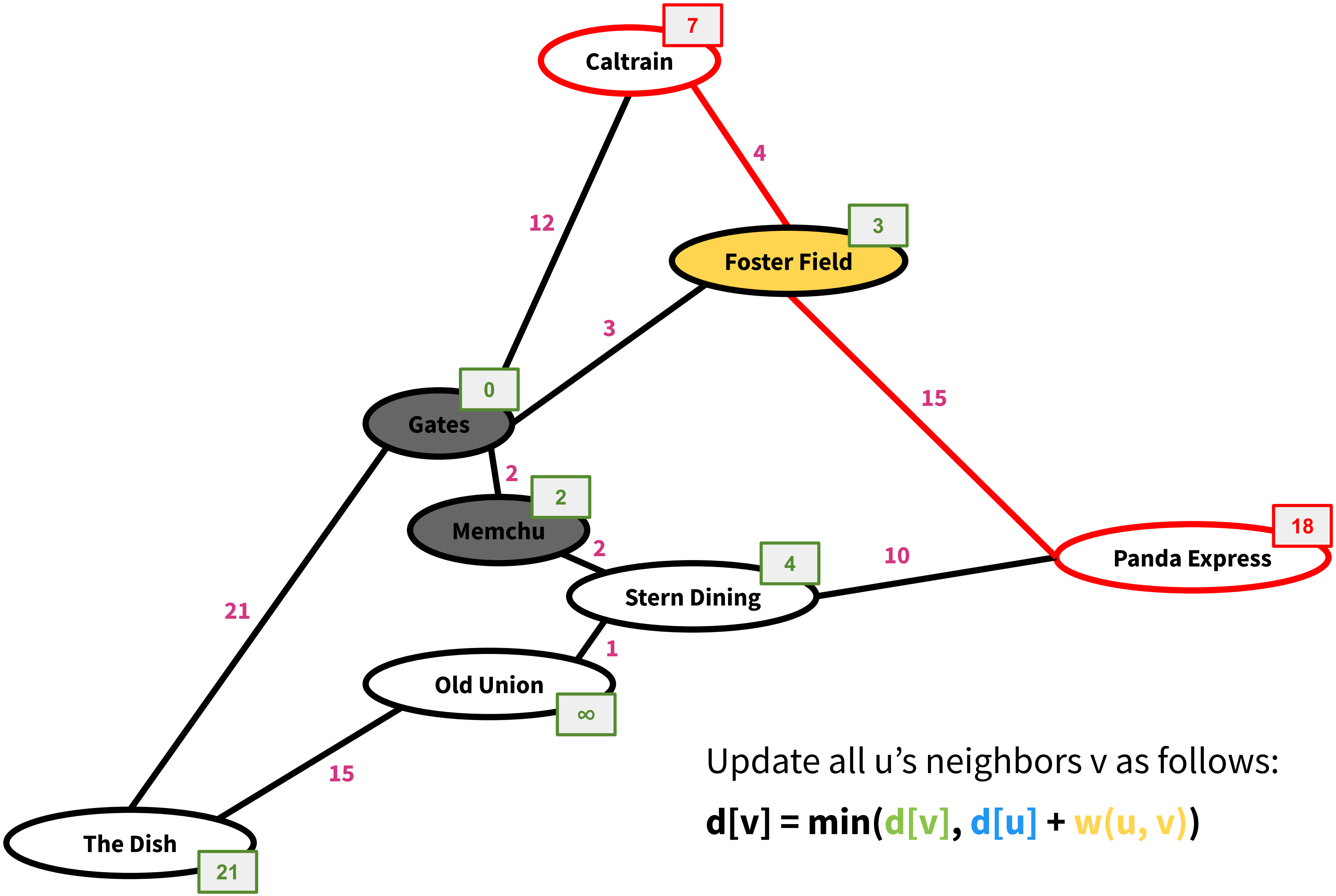
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

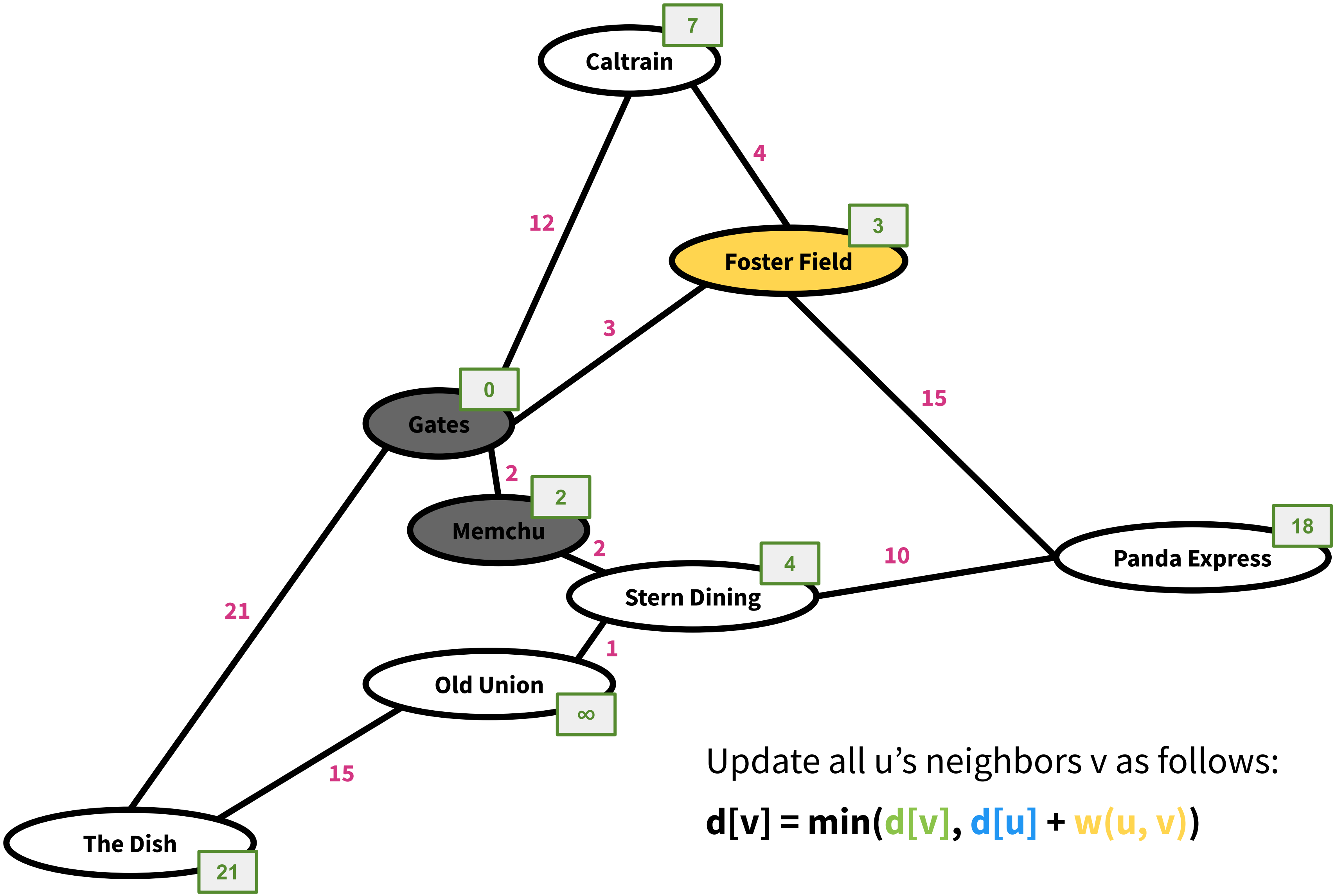
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

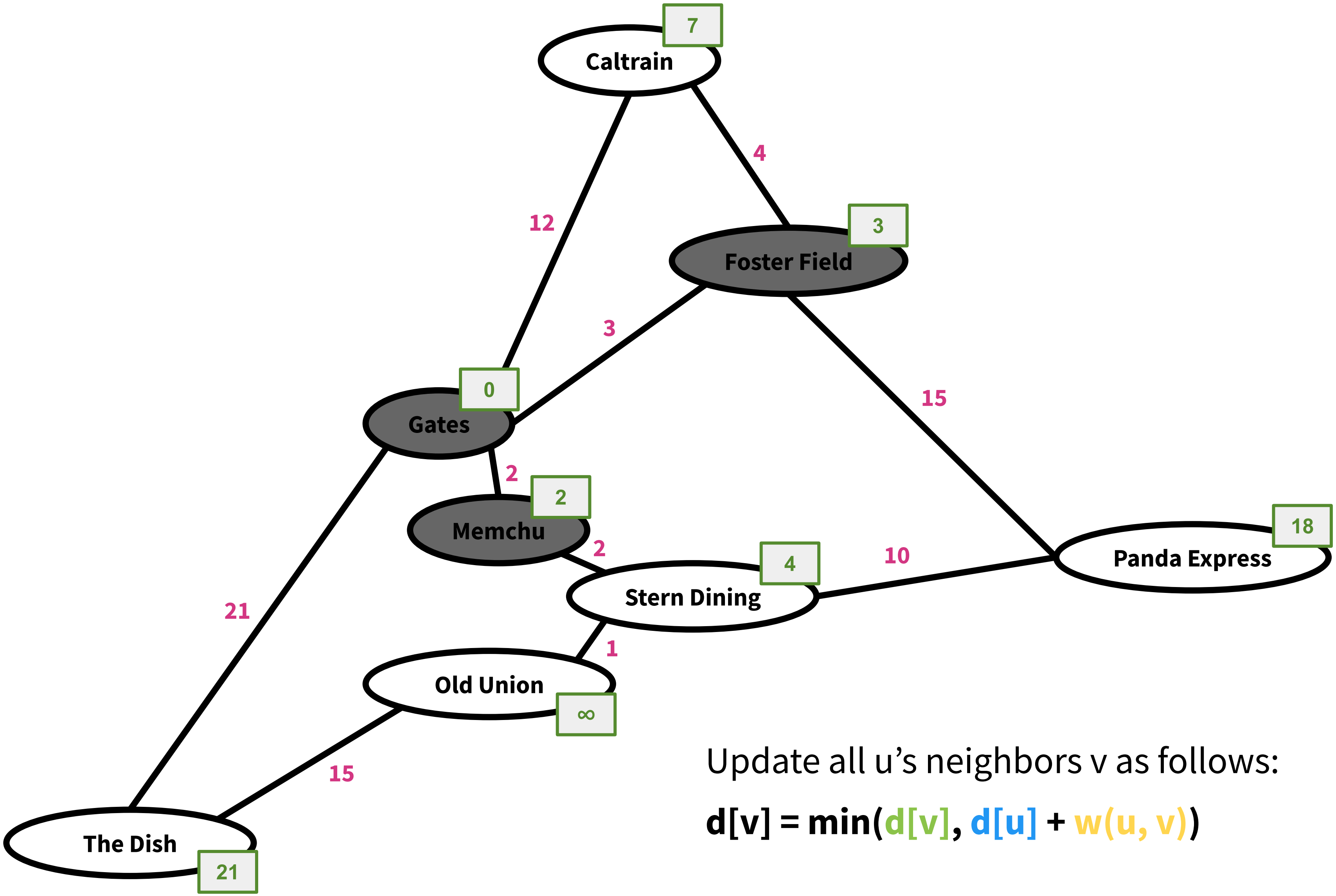
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

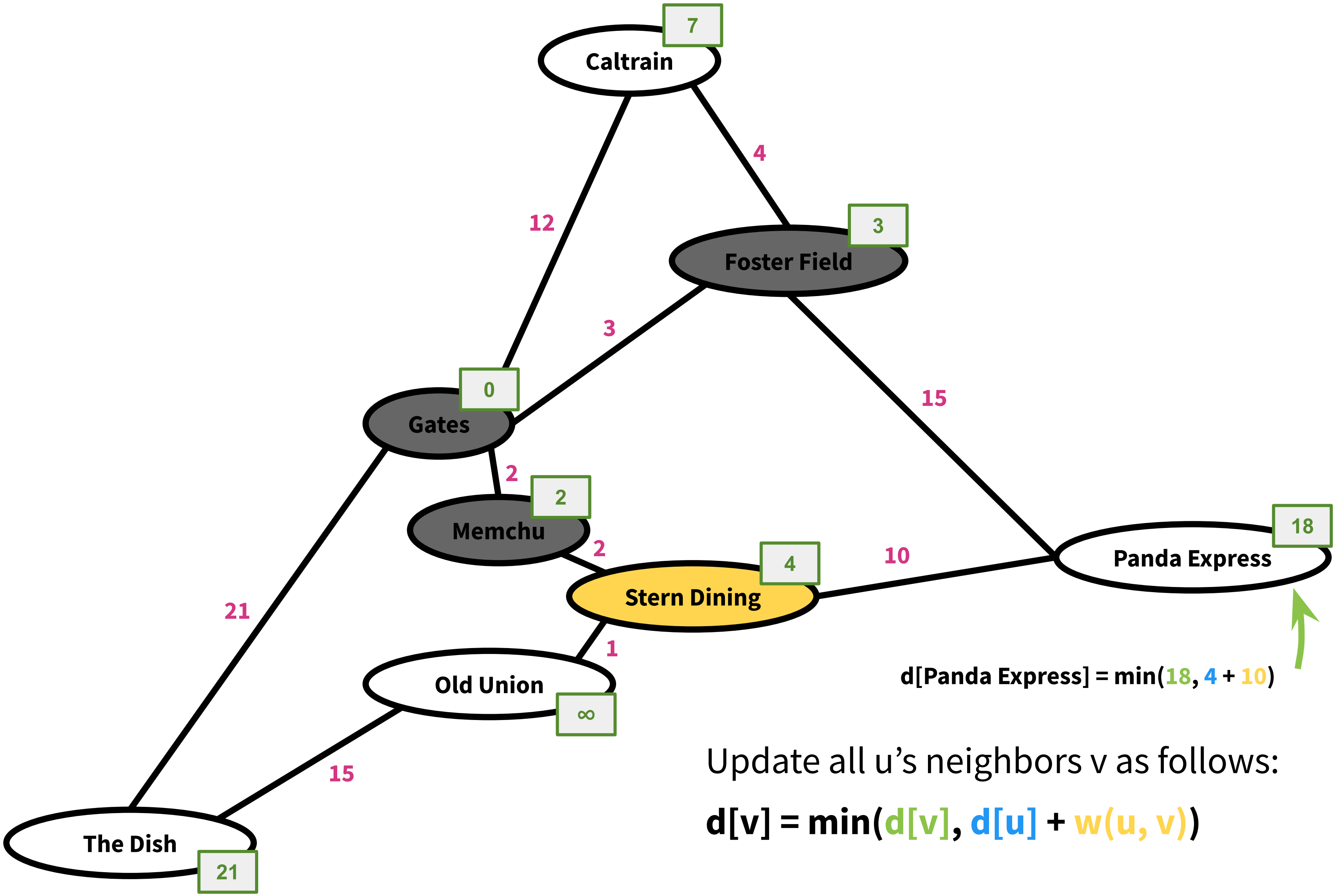
Dijkstra’s Algorithm



Update all u’s neighbors v as follows:

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

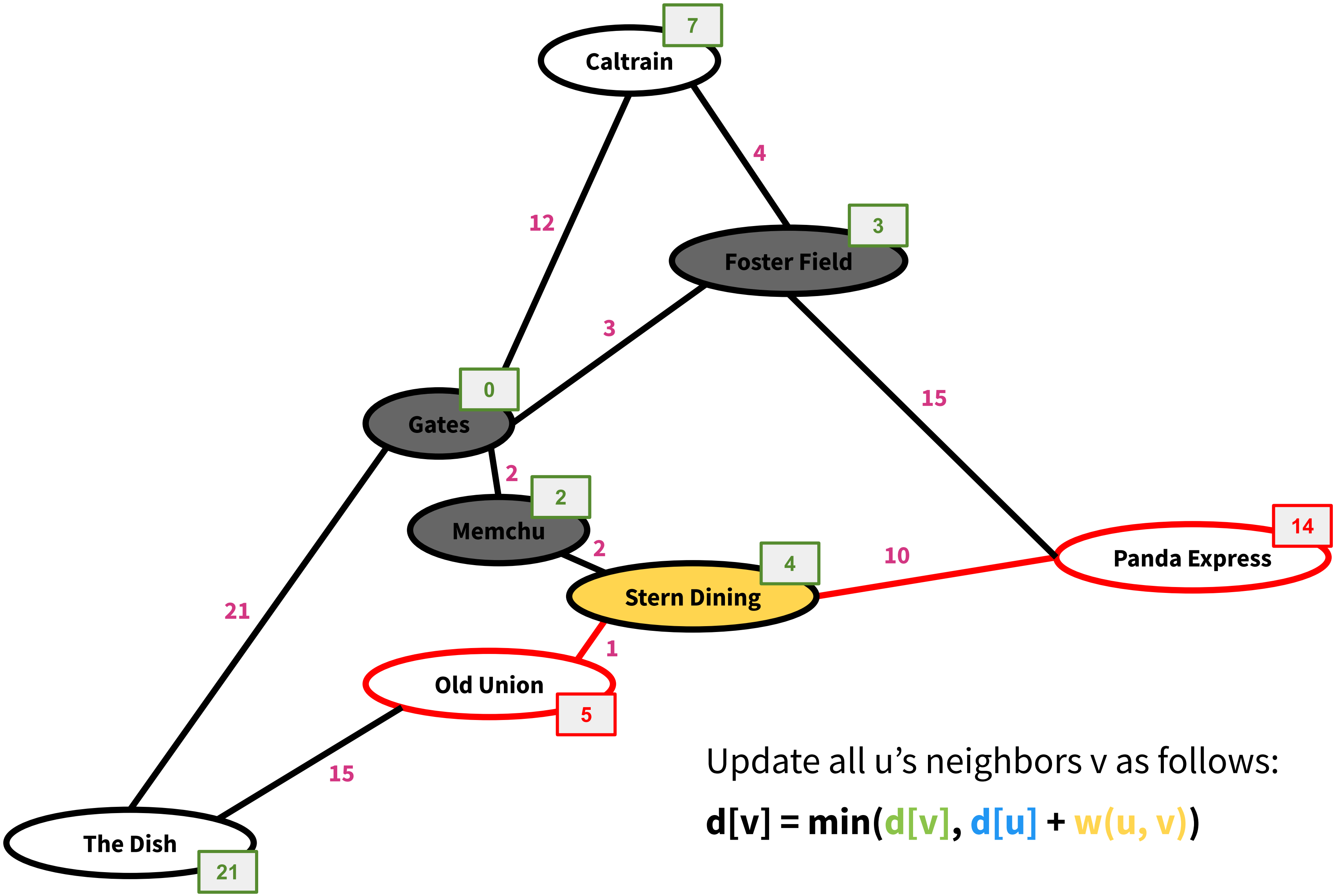
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

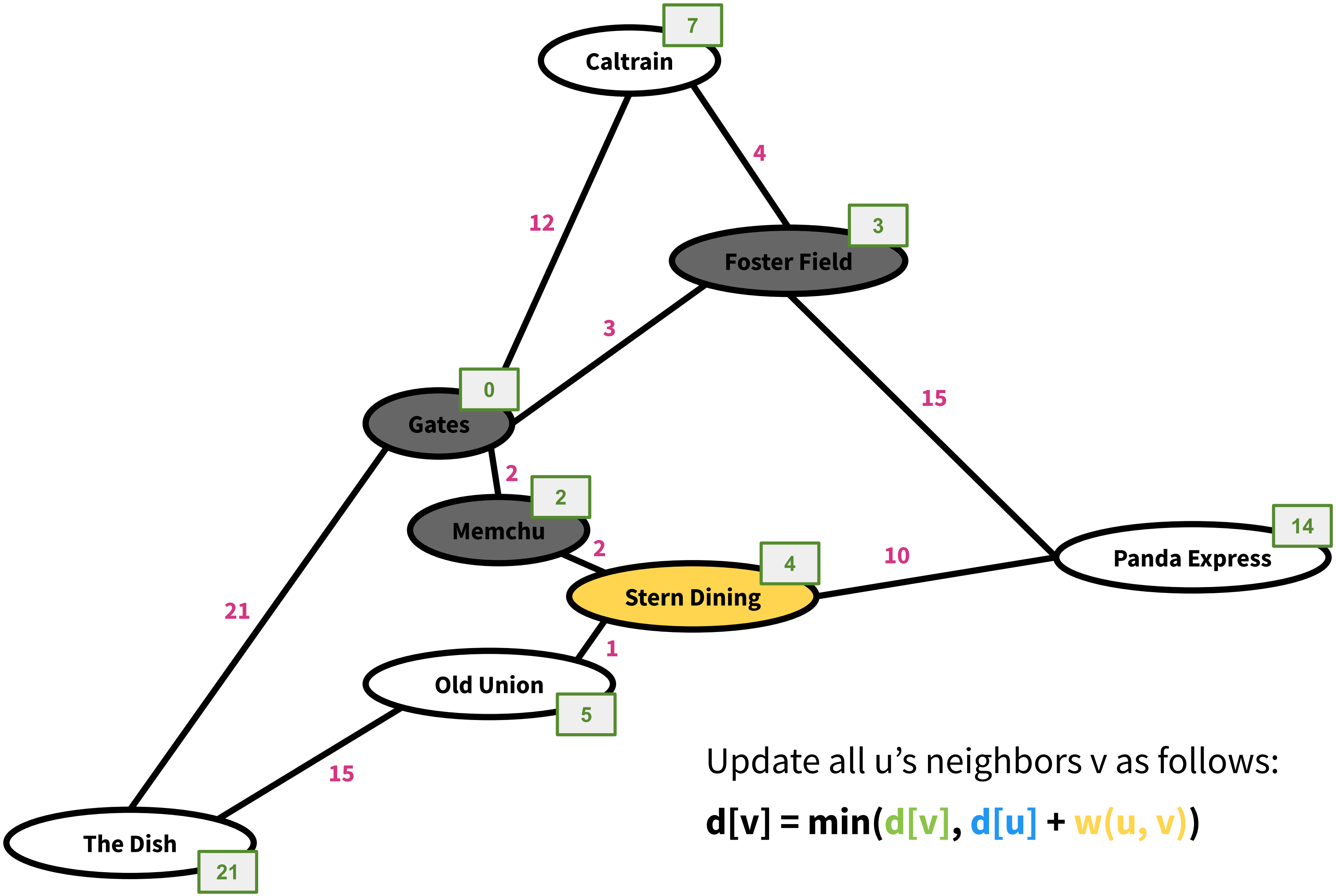
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

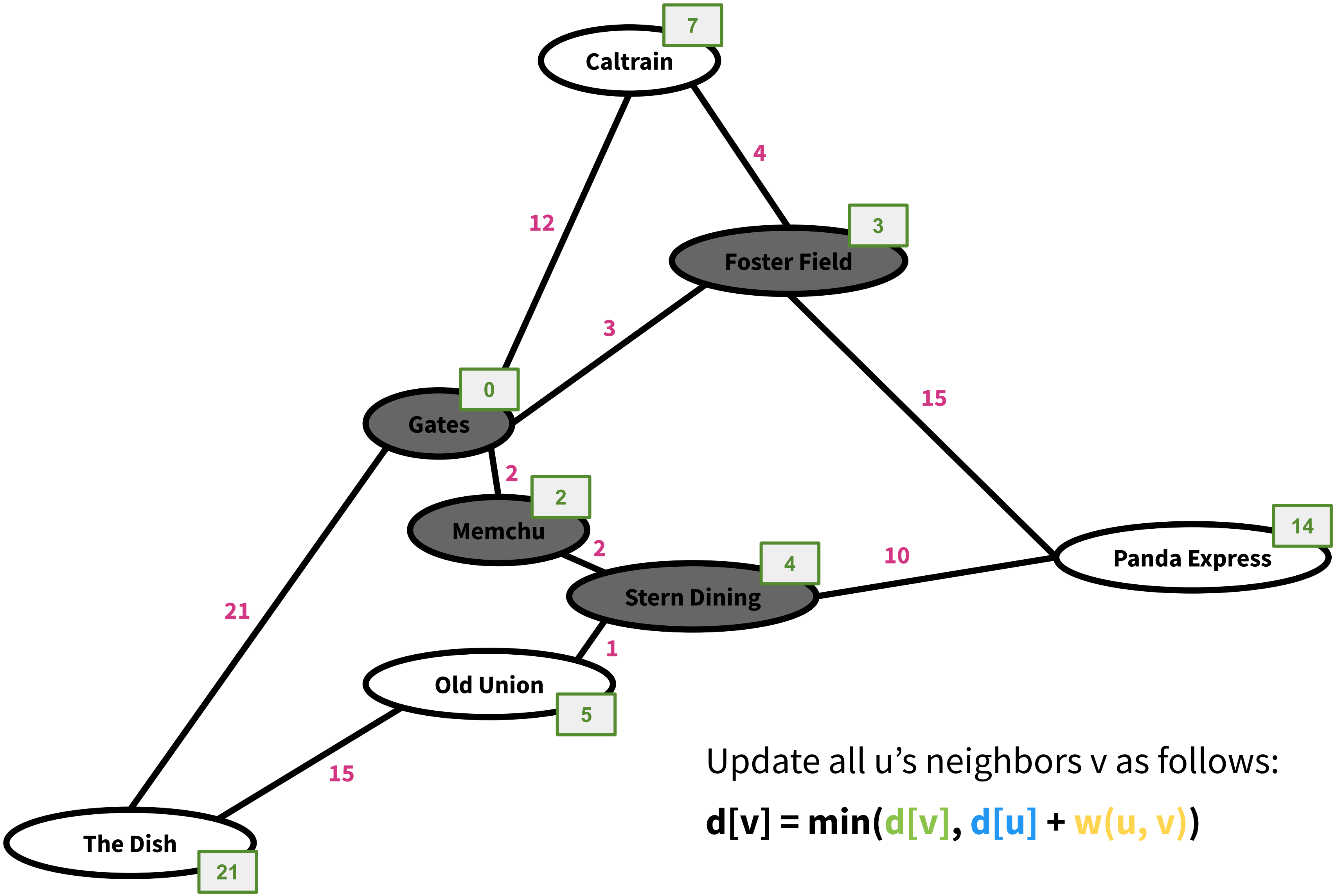
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

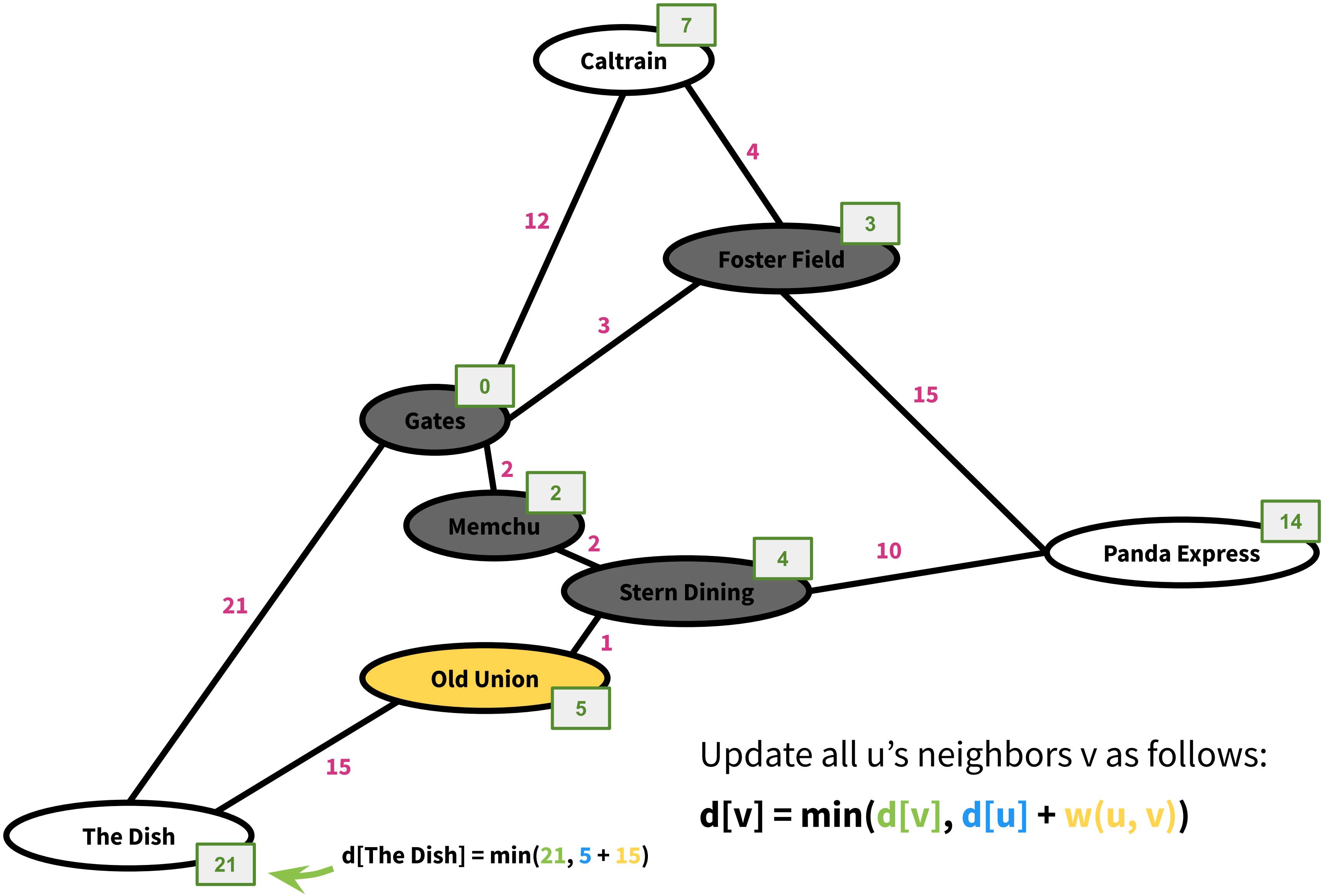
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

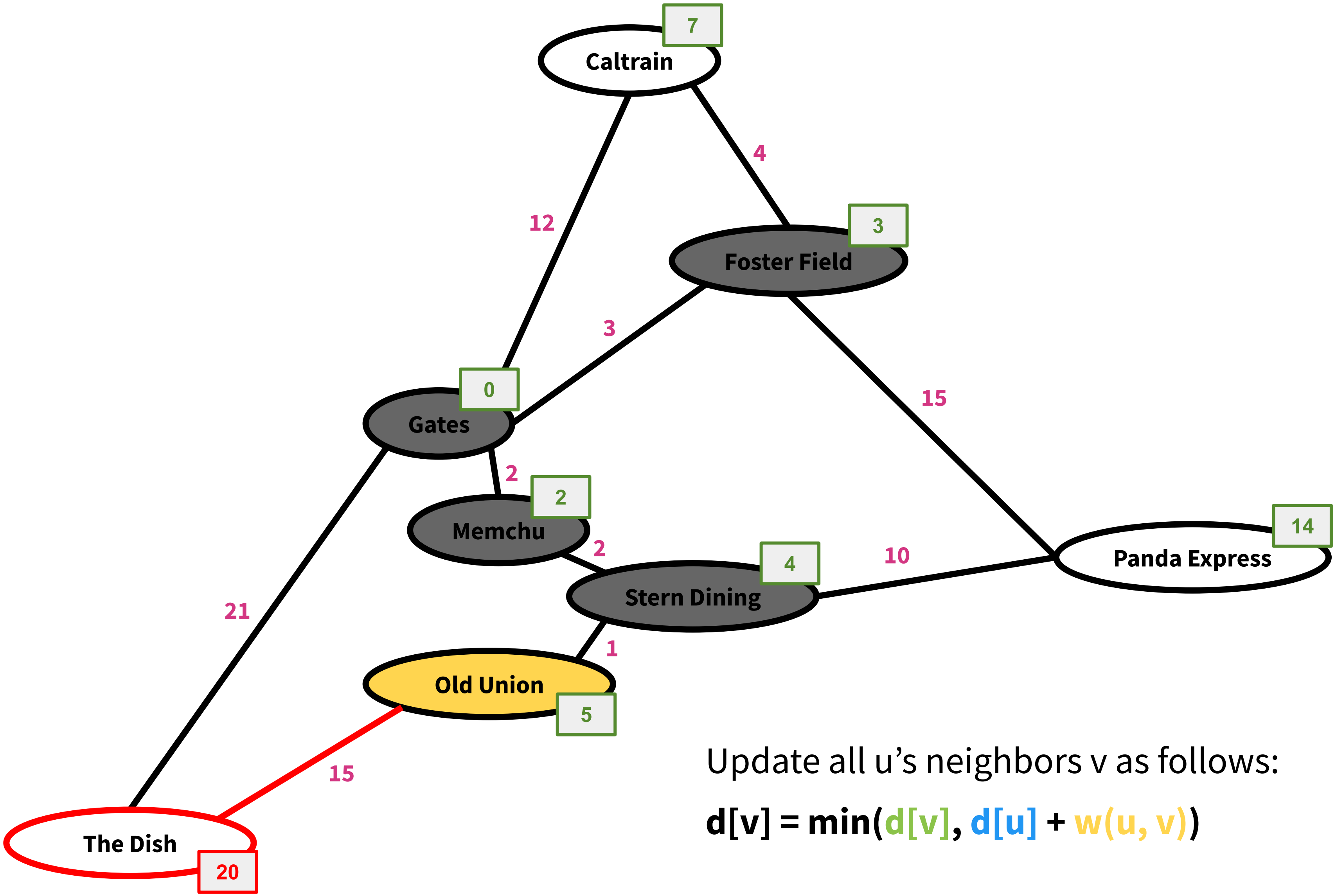
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

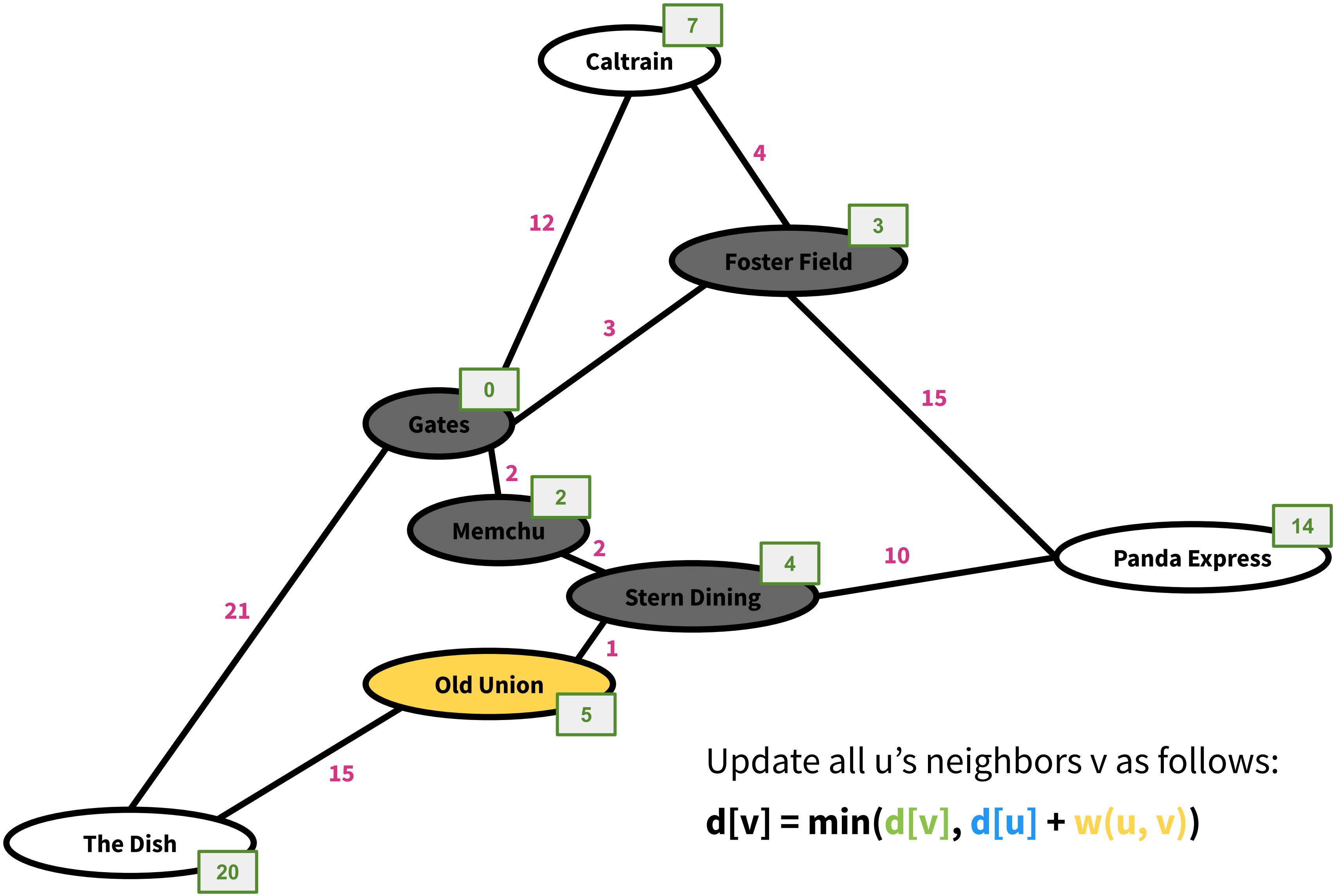
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

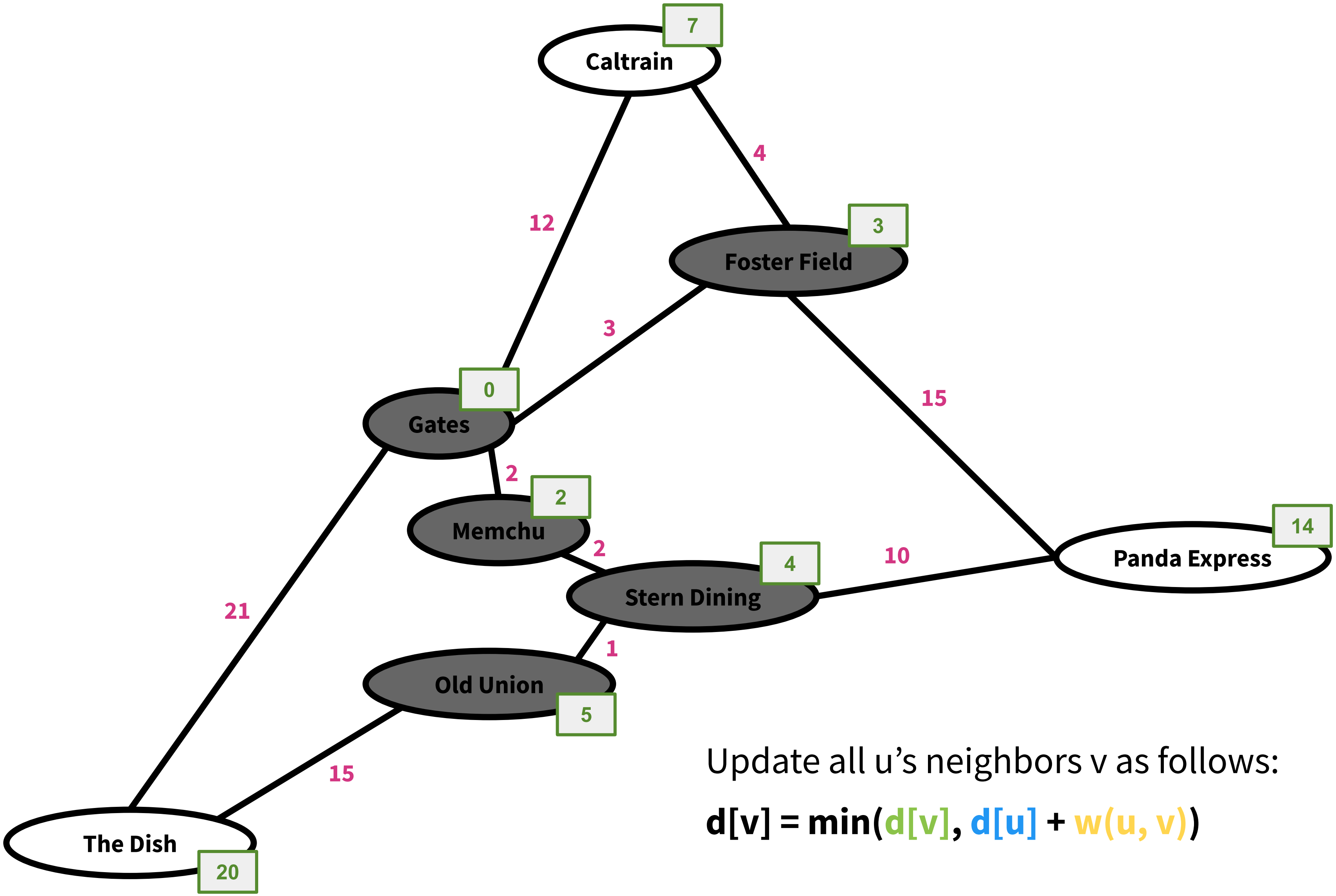
Dijkstra's Algorithm



Update all u's neighbors v as follows:

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

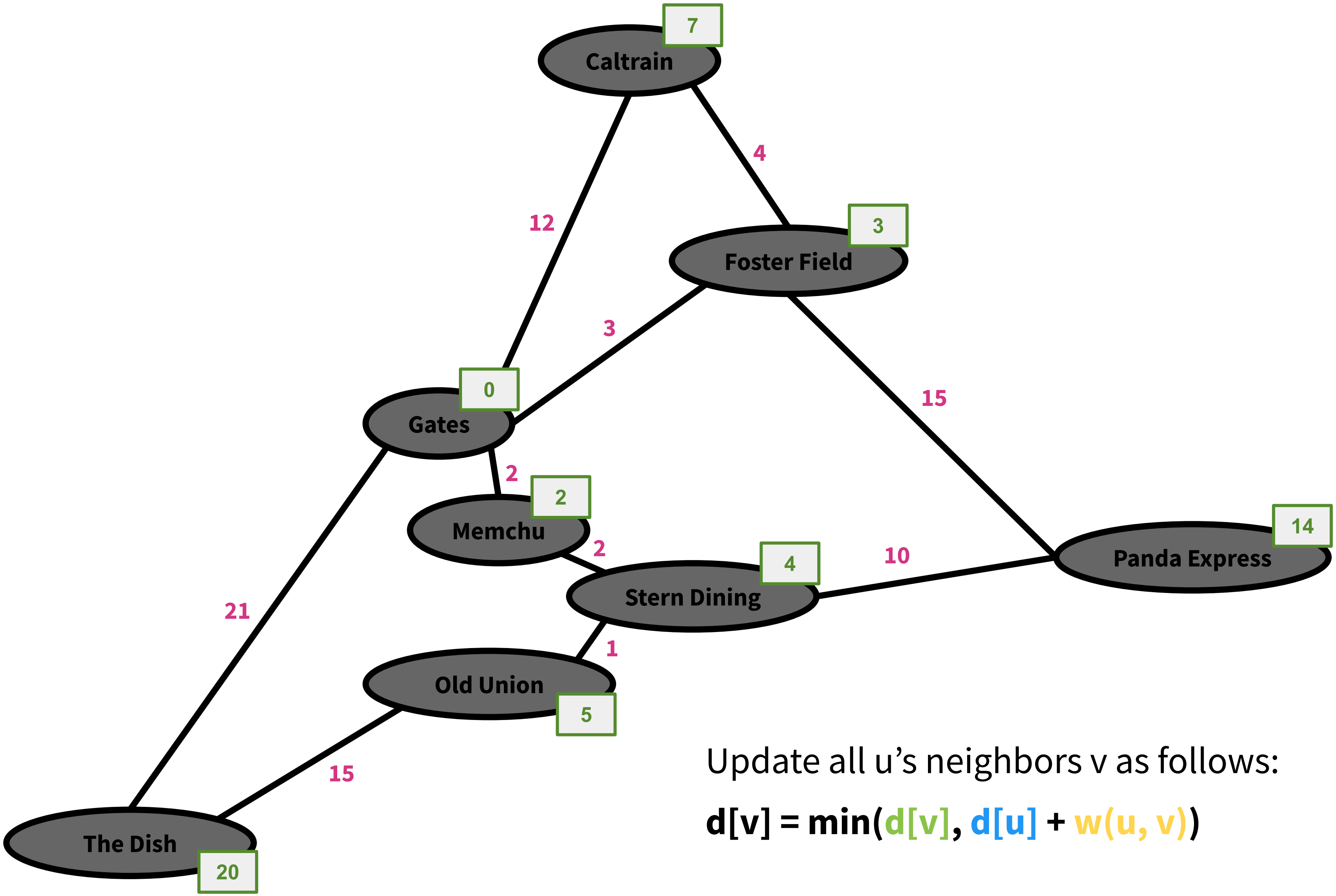
Dijkstra’s Algorithm



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

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, $d[v] \geq d(s, v)$.

Proof:

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

After $t = 0$ iterations, $d(s, s) = 0$ and $d(s, v) \leq \infty$ which satisfy $d[v] \geq d(s, v)$.

For the inductive step, suppose the inductive hypothesis holds for iteration t.

Dijkstra's Algorithm

Why does this work?

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

Proof:

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

After $t = 0$ iterations, $d(s, s) = 0$ and $d(s, v) \leq \infty$ which satisfy $d[v] \geq d(s, v)$.

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: $\mathbf{d[v] = \min(d[v], d[u] + w(u, v))} \geq d(s, v)$.

By induction, $d[v] \geq d(s, v)$

$d(s, v) \leq d(s, u) + d(u, v)$
$\leq d[u] + w(u, v)$

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 $d[x] \neq d(s, x)$.

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] ≠ 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) ≤ d[x] and we assumed that 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).

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) ≤ d(s, x) < d[x].

Weights are non-negative.

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

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

Therefore, d[z] < d[x]. 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.
$d[z'] \leq d(s, z) + w(z, z') = d(s, z')$ since z is on the shortest path from s to z' and the distance estimate of z' must be correct.

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.

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). ■

Dijkstra’s Algorithm

Another wording of Claim 2

- When a vertex v gets marked “done”, d[v] must be d(s, v).

Dijkstra's Algorithm

Another wording of Claim 2

- When a vertex v gets marked "done", d[v] must be d(s, v).
- By contradiction, assume there exists an x such that when it gets marked as "done," d[x] ≠ d(s, x).
- Consider the shortest path p from s to x.

Dijkstra's Algorithm

Another wording of Claim 2

- When a vertex v gets marked "done", d[v] must be d(s, v).
- By contradiction, assume there exists an x such that when it gets marked as "done," d[x] $\neq$ d(s, x).
- Consider the shortest path p from s to x.
- There must exist a vertex z closest to x on p for which d[z] = d(s, z). Notice, by our assumption that z $\neq$ x.

Dijkstra's Algorithm

Another wording of Claim 2

- When a vertex v gets marked “done”, d[v] must be d(s, v).
- By contradiction, assume there exists an x such that when it gets marked as “done,” d[x] ≠ d(s, x).
- Consider the shortest path p from s to x.
- There must exist a vertex z closest to x on p for which d[z] = d(s, z). Notice, by our assumption that z ≠ x.
- But z cannot be the closest vertex to x on p; simply consider the next vertex z’ along the path.

Dijkstra's Algorithm

Another wording of Claim 2

- When a vertex v gets marked “done”, d[v] must be d(s, v).
- By contradiction, assume there exists an x such that when it gets marked as “done,” d[x] ≠ d(s, x).
- Consider the shortest path p from s to x.
- There must exist a vertex z closest to x on p for which d[z] = d(s, z). Notice, by our assumption that z ≠ x.
- But z cannot be the closest vertex to x on p; simply consider the next vertex z’ along the path.
    - Since the subpath of a shortest path is also a shortest path, and p is a shortest path from s to z’ to x, then the subpath s to z’ must also be a shortest path.

Dijkstra’s Algorithm

Another wording of Claim 2

- When a vertex v gets marked “done”, d[v] must be d(s, v).
- By contradiction, assume there exists an x such that when it gets marked as “done,” d[x] ≠ d(s, x).
- Consider the shortest path p from s to x.
- There must exist a vertex z closest to x on p for which d[z] = d(s, z). Notice, by our assumption that z ≠ x.
- But z cannot be the closest vertex to x on p; simply consider the next vertex z’ along the path.
    - Since the subpath of a shortest path is also a shortest path, and p is a shortest path from s to z’ to x, then the subpath s to z’ must also be a shortest path.
    - z must have been marked as “done” since it’s on the subpath to x and weights are non-negative.

Dijkstra's Algorithm

Another wording of Claim 2

- When a vertex v gets marked "done", d[v] must be d(s, v).
- By contradiction, assume there exists an x such that when it gets marked as "done," d[x] ≠ d(s, x).
- Consider the shortest path p from s to x.
- There must exist a vertex z closest to x on p for which d[z] = d(s, z). Notice, by our assumption that z ≠ x.
- But z cannot be the closest vertex to x on p; simply consider the next vertex z' along the path.
    - Since the subpath of a shortest path is also a shortest path, and p is a shortest path from s to z' to x, then the subpath s to z' must also be a shortest path.
    - z must have been marked as "done" since it's on the subpath to x and weights are non-negative.
    - Before z was marked as "done", d[z'] was updated to d[z] + w(z, z'), which must equal d(s, z') since s to z to z' is a shortest path.

Dijkstra's Algorithm

Another wording of Claim 2

- When a vertex v gets marked "done", d[v] must be d(s, v).
- By contradiction, assume there exists an x such that when it gets marked as "done," d[x] $\neq$ d(s, x).
- Consider the shortest path p from s to x.
- There must exist a vertex z closest to x on p for which d[z] = d(s, z). Notice, by our assumption that z $\neq$ x.
- But z cannot be the closest vertex to x on p; simply consider the next vertex z' along the path.
  - Since the subpath of a shortest path is also a shortest path, and p is a shortest path from s to z' to x, then the subpath s to z' must also be a shortest path.
  - z must have been marked as "done" since it's on the subpath to x and weights are non-negative.
  - Before z was marked as "done", d[z'] was updated to d[z] + w(z, z'), which must equal d(s, z') since s to z to z' is a shortest path.
- **Thus, contradiction!**

Bellman-Ford

Bellman-Ford Algorithm

- Dijkstra's algorithm solves the single-source shortest path problem in weighted graphs.
  - Sometimes it works on graphs with negative edge weights, but sometimes it doesn't work.
- Bellman-Ford also solves the SSSP problem in weighted graphs.
  - Always works on graphs with negative edge weights (when a solution exists).

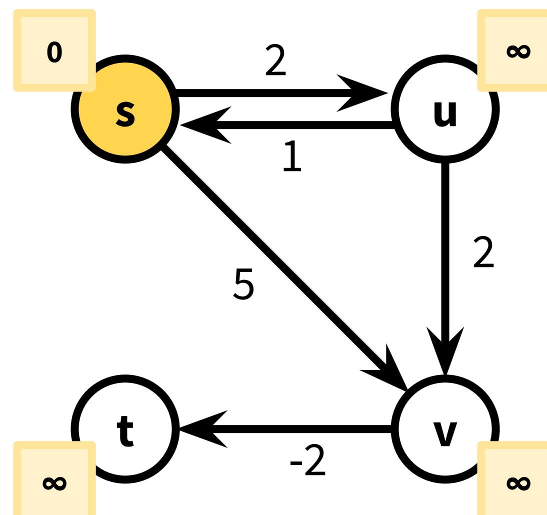
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each $k = 0, 1, \ldots, |V|-1$.

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

We know k = 0
i.e. shortest
paths to each
vertex with at
most 0 edges
in it.

	s	**u**	**v**	**t**
$\mathbf{d^{(0)}}$	**0**	**∞**	**∞**	**∞**
$\mathbf{d^{(1)}}$				
$\mathbf{d^{(2)}}$				
$\mathbf{d^{(3)}}$				



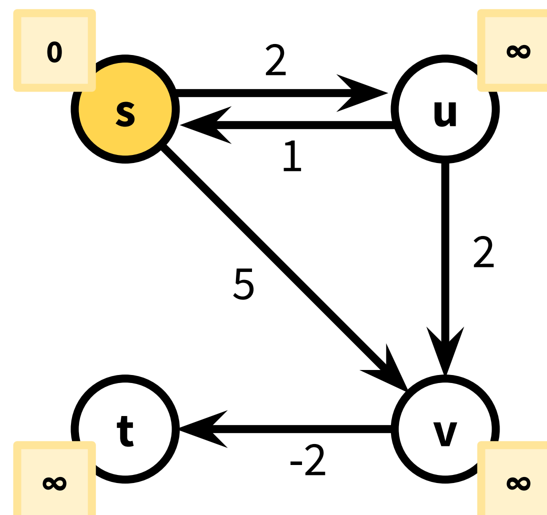
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				

We will use table $d^{(0)}$ to fill in $d^{(1)}$. More generally, we will use table $d^{(k-1)}$ to fill in $d^{(k)}$.

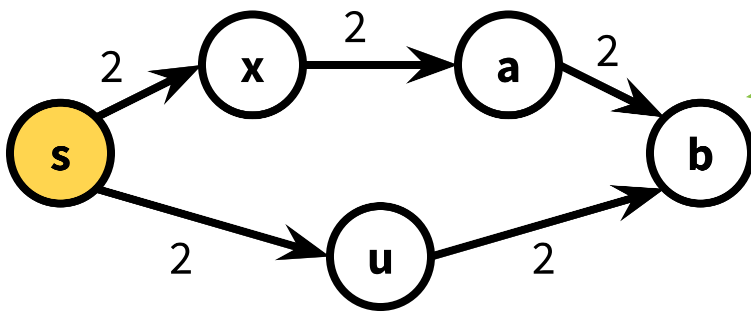


Bellman-Ford Algorithm

How do we use $d^{(k-1)}$ to fill in $d^{(k)}[b]$?

Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

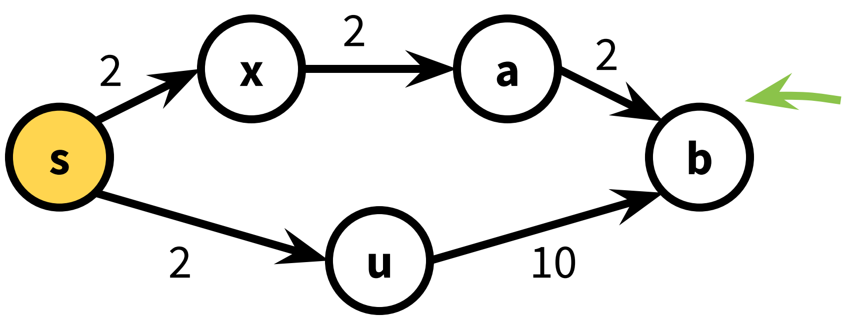
Case 1: the shortest path from s to b with at most k edges actually has at most k - 1 edges.



Suppose k = 3.

$d^{(k)}[b] = d^{(k-1)}[b]$ i.e. the shortest path of at most k - 1 edges is at least as short as any path of at most k edges.

Case 2: the shortest path from s to b with at most k edges really has k edges.



Suppose k = 3.

$d^{(k)}[b] = \min_a\{d^{(k-1)}[a] + w(a, b)\}$ i.e. the shortest path of at most k edges is shorter than any path of at most k - 1 edges.

Bellman-Ford Algorithm

```
def bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

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

Bellman-Ford Algorithm

```
def bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Runtime: O(|V||E|)

Bellman-Ford Algorithm

```
def bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Minimum over all a such that (a, b) ∈ E.

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

Bellman-Ford Algorithm

```
def bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

$$d^{(k)}[b] = \min\{\underbrace{d^{(k-1)}[b]}_{\text{Case 1}},\ \min_a\{d^{(k-1)}[a] + w(a,b)\}\ \}$$

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Minimum over all a such that (a, b) ∈ E.

Runtime: O(|V||E|)

Bellman-Ford Algorithm

```
def bellman_ford(G):
  d(k) = [] for k = 0 to |V|-1
  d(0)[v] = ∞ for all v ≠ s
  d(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d(k)[b] = min{d(k-1)[b], min_a{d(k-1)[a] + w(a,b)} }
  return d(|V|-1)
```

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Minimum over all a such that (a, b) ∈ E.

Case 1: $d^{(k-1)}[b]$

Case 2: $\min_a\{d^{(k-1)}[a] + w(a,b)\}$

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

Bellman-Ford Algorithm

```
def bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

$$d^{(k)}[b] = \min\{\underbrace{d^{(k-1)}[b]}_{\text{Case 1}},\ \underbrace{\min_a\{d^{(k-1)}[a] + w(a,b)\}}_{\text{Case 2}}\}$$

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Minimum over all a such that (a, b) ∈ E.

Case 1

Case 2

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

Slower than Dijkstra's $O(|E| + |V|\log(|V|))$

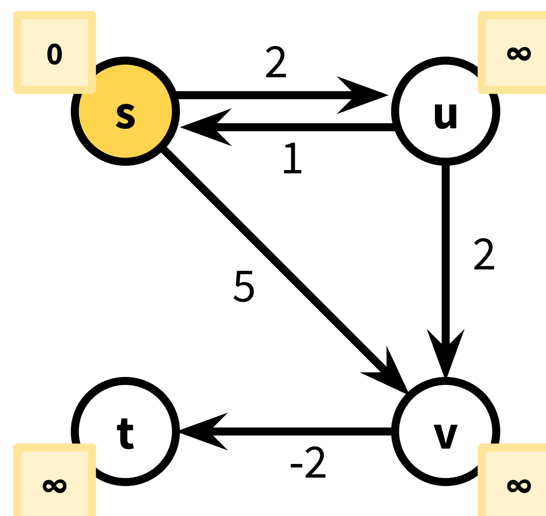
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				

We will use table $d^{(0)}$ to fill in $d^{(1)}$. More generally, we will use table $d^{(k-1)}$ to fill in $d^{(k)}$.



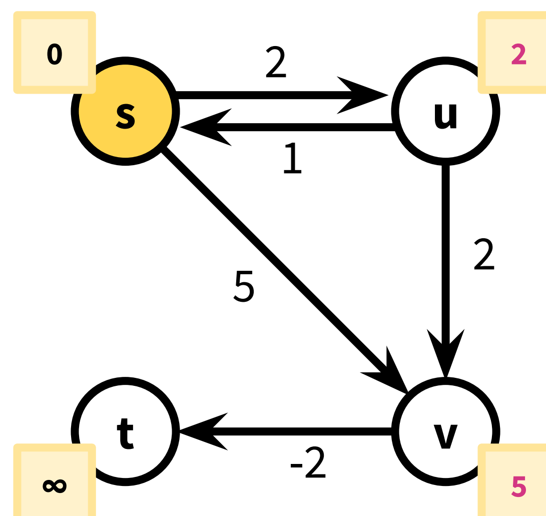
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$				
$d^{(3)}$				

We will use table $d^{(0)}$ to fill in $d^{(1)}$. More generally, we will use table $d^{(k-1)}$ to fill in $d^{(k)}$.

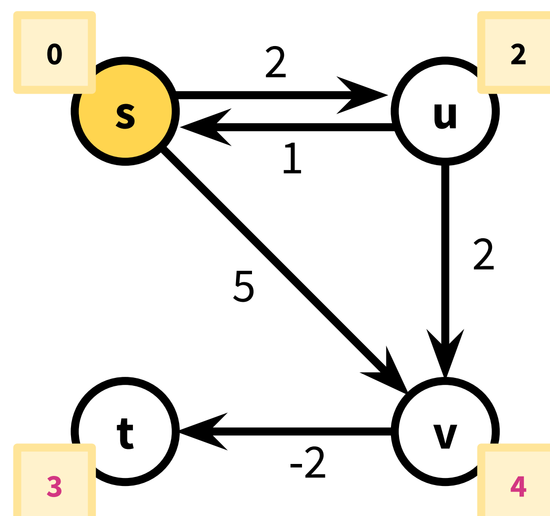


Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$				

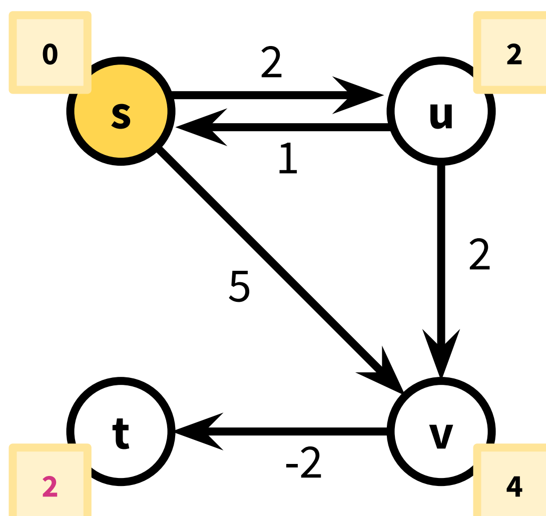


Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2

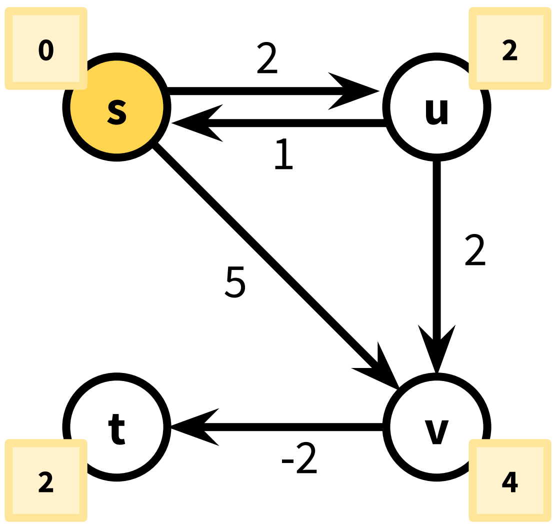


Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



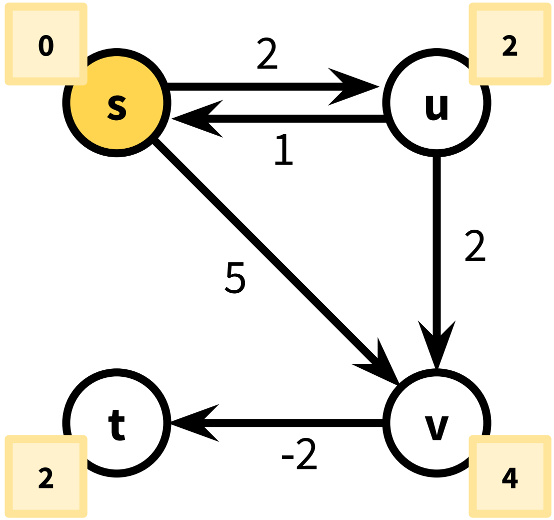
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with 1 edge has cost ∞ (no path exists).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



Bellman-Ford Algorithm

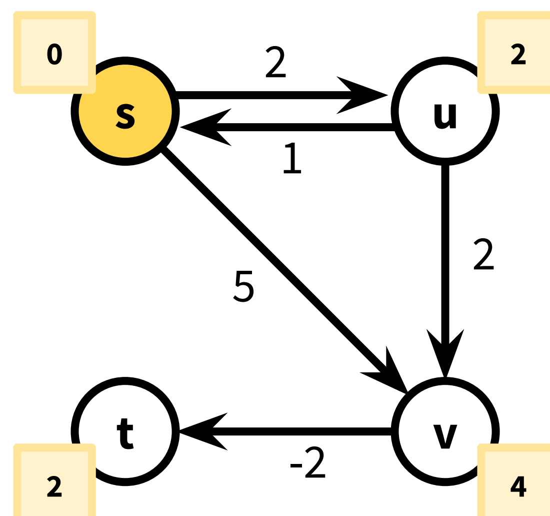
We maintain a list $d^{(k)}$ of length n for each $k = 0, 1, \ldots, |V|-1$.

Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with 1 edge has cost **∞** (no path exists).

The shortest path from s to t with 2 edges has cost **3** (s-v-t).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

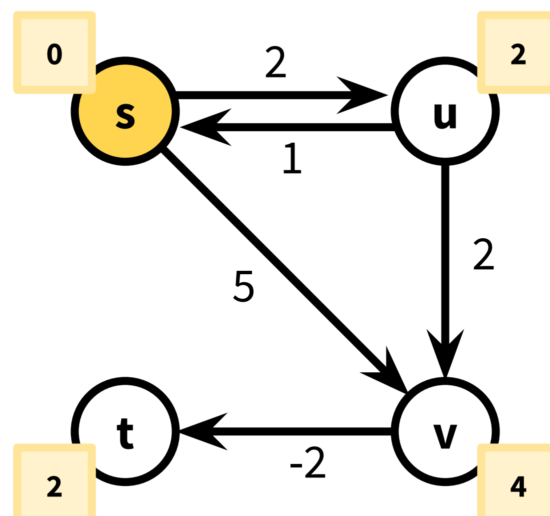
Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with 1 edge has cost **∞** (no path exists).

The shortest path from s to t with 2 edges has cost **3** (s-v-t).

The shortest path from s to t with 3 edges has cost **2** (s-u-v-t).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



BF Proof of Correctness

We need to prove our main argument.

- $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices $v \neq s$ contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices $v \neq s$ contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

For our inductive step, assume that at the start of iteration k, $d^{(k-1)}[b]$ is the cost of the shortest path from s to b with at most k - 1 edges. We consider two cases:

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices v ≠ s contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

For our inductive step, assume that at the start of iteration k, $d^{(k-1)}[b]$ is the cost of the shortest path from s to b with at most k - 1 edges. We consider two cases:

Case 1: $d^{(k-1)}[b] < \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains fewer than k edges. Then our algorithm correctly sets $d^{(k)}[b] = d^{(k-1)}[b]$.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices v ≠ s contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

For our inductive step, assume that at the start of iteration k, $d^{(k-1)}[b]$ is the cost of the shortest path from s to b with at most k - 1 edges. We consider two cases:

Case 1: $d^{(k-1)}[b] < \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains fewer than k edges. Then our algorithm correctly sets $d^{(k)}[b] = d^{(k-1)}[b]$.

Case 2: $d^{(k-1)}[b] \geq \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains exactly k edges. Then our algorithm correctly sets $d^{(k)}[b] = \min_a\{d^{(k-1)}[a] + w(a, b)\}$, which minimizes the sum of the shortest path with at most k-1 edges to an in-neighbor of b and the weight from a to b.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices $v \neq s$ contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

For our inductive step, assume that at the start of iteration k, $d^{(k-1)}[b]$ is the cost of the shortest path from s to b with at most k - 1 edges. We consider two cases:

Case 1: $d^{(k-1)}[b] < \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains fewer than k edges. Then our algorithm correctly sets $d^{(k)}[b] = d^{(k-1)}[b]$.

Case 2: $d^{(k-1)}[b] \geq \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains exactly k edges. Then our algorithm correctly sets $d^{(k)}[b] = \min_a\{d^{(k-1)}[a] + w(a, b)\}$, which minimizes the sum of the shortest path with at most k-1 edges to an in-neighbor of b and the weight from a to b.

At the start of iteration k = |V|, the algorithm terminates and $d^{(|V|-1)}$ is correct.

BF Proof of Correctness

We need to prove our main argument.

$d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges. 👌

What else to do? 🤔

BF Proof of Correctness

We need to prove our main argument.

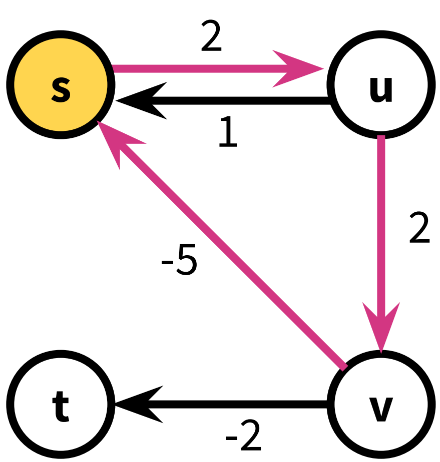
$d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges. 👌

What else to do? 🤔

We still need to prove that this argument implies `bellman_ford` is correct i.e. $d^{(|V|-1)}[a]$ = distance(s, a).

To show this, we'll prove that the shortest path with at most |V|-1 edges is the shortest path with any number of edges (if a shortest path exists).

If the graph has a negative cycle, a shortest path might not exist!

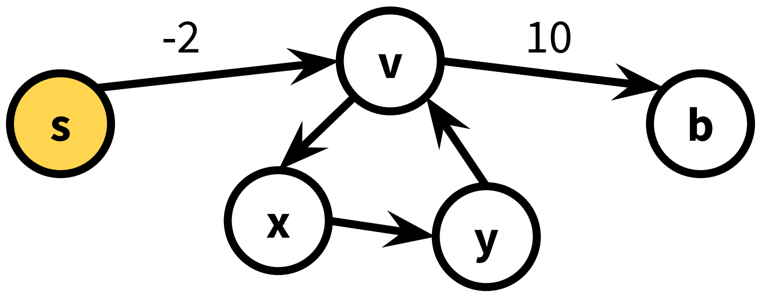


BF Proof of Correctness

But if there's no negative cycle.

There's always a simple shortest path.

A simple path has
no cycles.

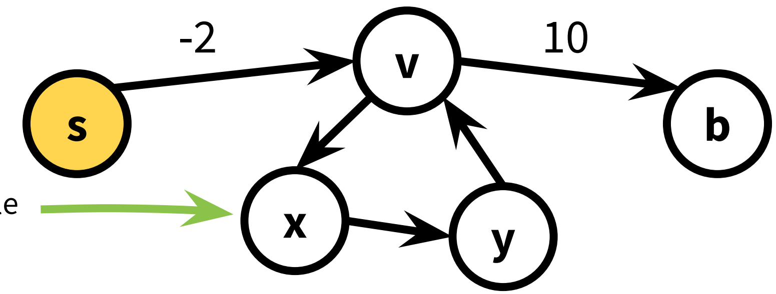


BF Proof of Correctness

But if there's no negative cycle.

There's always a simple shortest path.

A simple path has no cycles.



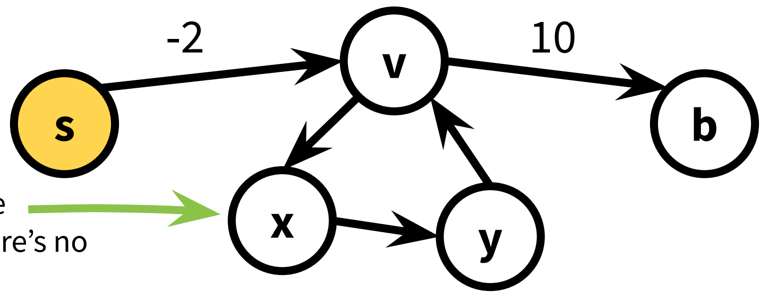
How do we know this cycle doesn't help? 🤔

BF Proof of Correctness

But if there’s no negative cycle.

There’s always a simple shortest path.

A simple path has
no cycles.



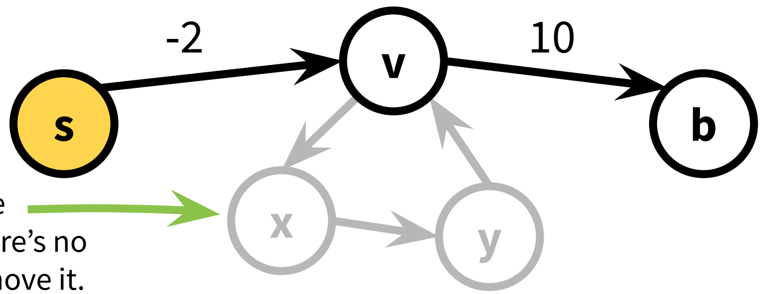
How do we know this cycle
doesn’t help? 🤔 Since there’s no
negative cycles!

BF Proof of Correctness

But if there's no negative cycle.

There's always a simple shortest path.

A simple path has no cycles.



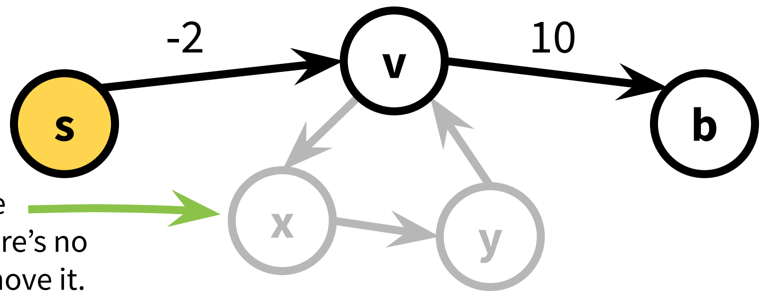
How do we know this cycle doesn't help? 🤔 Since there's no negative cycles! So we remove it.

BF Proof of Correctness

But if there’s no negative cycle.

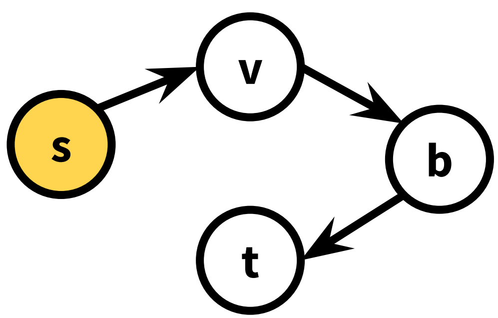
There’s always a simple shortest path.

A simple path has no cycles.



How do we know this cycle doesn’t help? 🤔 Since there’s no negative cycles! So we remove it.

A simple path in a graph with |V| vertices has at most |V|-1 edges in it.

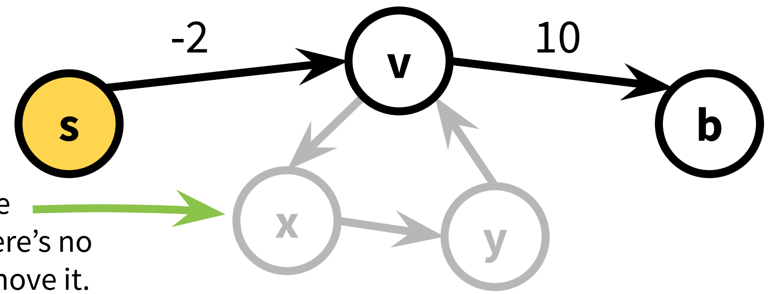


BF Proof of Correctness

But if there’s no negative cycle.

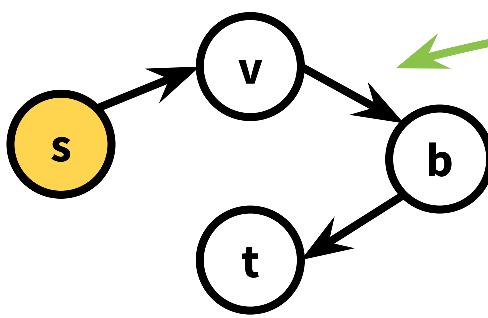
There’s always a simple shortest path.

A simple path has no cycles.



How do we know this cycle doesn’t help? 🤔 Since there’s no negative cycles! So we remove it.

A simple path in a graph with |V| vertices has at most |V|-1 edges in it.



We can’t add another edge to this s-t path without making a cycle (an edge from s to b wouldn’t be along the path).

BF Proof of Correctness

Theorem: `bellman_ford` is correct as long as the graph has no negative cycles.

Proof:

By our lemma, $d^{(|V|-1)}[b]$ contains the cost of the shortest path from s to b with at most |V|-1 edges. If there are no negative cycles, then the shortest path must be simple, and all simple paths have at most |V|-1 edges. Therefore, the value the algorithm returns, $d^{(|V|-1)}[b]$, is also the cost of the shortest path from s to b with any number of edges.

Bellman-Ford Algorithm

Bellman-Ford gets used in practice.

- e.g. Routing Information Protocol (RIP) uses it. Each router keeps a table of distances to every other router. Periodically, we do a Bellman-Ford update.

Dynamic Programming

Bellman-Ford is an example of **dynamic programming**!

Dynamic programming is an algorithm design paradigm.

- Often it's used to solve optimization problems e.g. **shortest** path.