Lecture 36:
Shortest Path First

December 16, 2021

SHORTEST PATH

- Finding the shortest path is a classical problem in graph theory.
- Edges are assigned certain weights representing, for example,
  - distances between cities,
  - times separating the execution of certain tasks,
  - costs of transmitting information between locations,
  - amounts of some substance transported from one place to another,
  - and so on.

SHORTEST PATH

- When determining the shortest path from vertex **v** to vertex **u**, information about distances between intermediate vertices w has to be recorded.
- This information can be recorded as a **label** associated with these vertices.
- Where the label is only the distance from **v** to **w** or the distance along with the predecessor of **w** in this path.
- The methods of finding the shortest path rely on these labels.

SHORTEST PATH

Depending upon how many times these labels are updated, we categorize the shortest path methods into two classes

- **Label Setting Methods**
- **Label Updating Methods**

SHORTEST PATH

- **Label Setting Methods**
  - In each pass through the vertices still to be processed, one vertex is set to a value that remains unchanged to the end of the execution.
  - This, however, limits such methods to processing graphs with only positive weights.
- **Label Updating Methods**
  - Allows for the changing of any label during application of the method. The latter methods can be applied to graphs with negative weights.
  - However, will not work negative cycle

GENERIC SHORTEST PATH

```
GenericShortestPathAlgorithm(weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    initialize  toBeChecked;
    while toBeChecked is not empty
        v = a vertex in toBeChecked;
        remove v from toBeChecked;
        for all vertices u adjacent to v
            if currDist(u) > currDist(v) + weight(edge(vu))
                currDist(u) = currDist(v) + weight(edge(vu));
                predecessor(u) = v;
                add u to toBeChecked if it is not there;
```

GENERIC SHORTEST PATH METHOD

- In generic algorithm label is
  - **label(v) = (currDist(v), predecessor(v))**
- Two open things
  - The organization of the set toBeChecked
  - The order of assigning new values to v in the assignment statement
    - v = a vertex in toBeChecked;
- It should be clear that the organization of toBeChecked can determine
  - the **order of choosing new values for v**,
  - but it also **determines the efficiency of the algorithm.**

LABEL SETTING Vs CORRECTING

What distinguishes label-setting methods from label-correcting methods is the **method of choosing the value for v**,

which is always a vertex in toBeChecked

with the smallest current distance.

Dijkstra's Algorithm

December 16, 2021

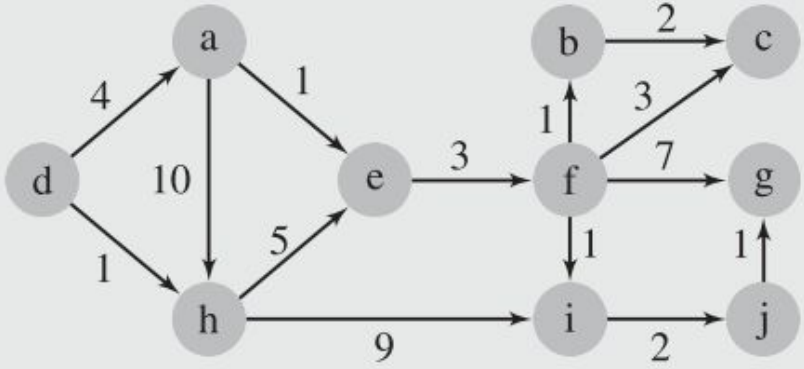
GENERIC SHORTEST PATH

```
DijkstraAlgorithm(weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    toBeChecked = all vertices;
    while toBeChecked is not empty
        v = a vertex in toBeChecked with minimal currDist(v);
        remove v from toBeChecked;
        for all vertices u adjacent to v and in toBeChecked
            if currDist(u) > currDist(v) + weight(edge(vu))
                currDist(u) = currDist(v) + weight(edge(vu));
                predecessor(u) = v;
```

GENERIC & DIJKSTRA

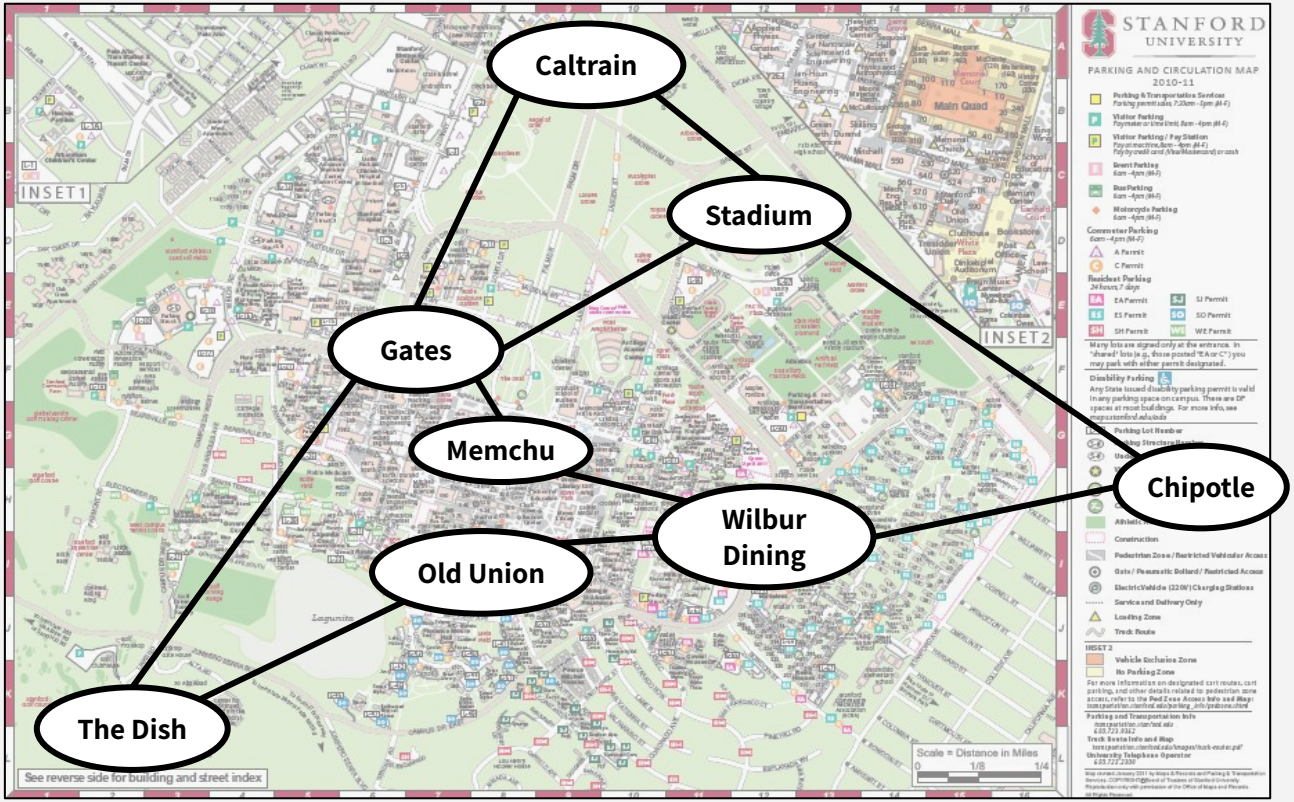
- Dijkstra is obtained from the generic method by being more specific
  - Which vertex is to be taken from toBeChecked so that the line
    - v = a vertex in toBeChecked; // this line is replaced
    - v = a vertex in toBeChecked with minimal currDist(v);
- By extending the condition in the if statement whereby the current distance of vertices eliminated from toBeChecked is set permanently.
- Note that the **structure of toBeChecked is not specified**, and the efficiency of the algorithms depends on the data type of toBeChecked, which determines how quickly a vertex with minimal distance can be retrieved

iteration:	init	1	2	3	4	5	6	7	8	9	10
active vertex:		d	h	a	e	f	b	i	c	j	g
a	∞	4	4								
b	∞	∞	∞	∞	∞	9					
c	∞	∞	∞	∞	∞	11	11	11			
d	0										
e	∞	∞	6	5							
f	∞	∞	∞	∞	8						
g	∞	∞	∞	∞	∞	15	15	15	15	12	
h	∞	1									
i	∞	∞	10	10	10	9	9				
j	∞	∞	∞	∞	∞	∞	∞	11	11		



SHORTEST PATHS IN WEIGHTED GRAPHS

Suppose you only know your way around campus via certain landmarks

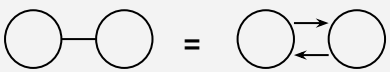


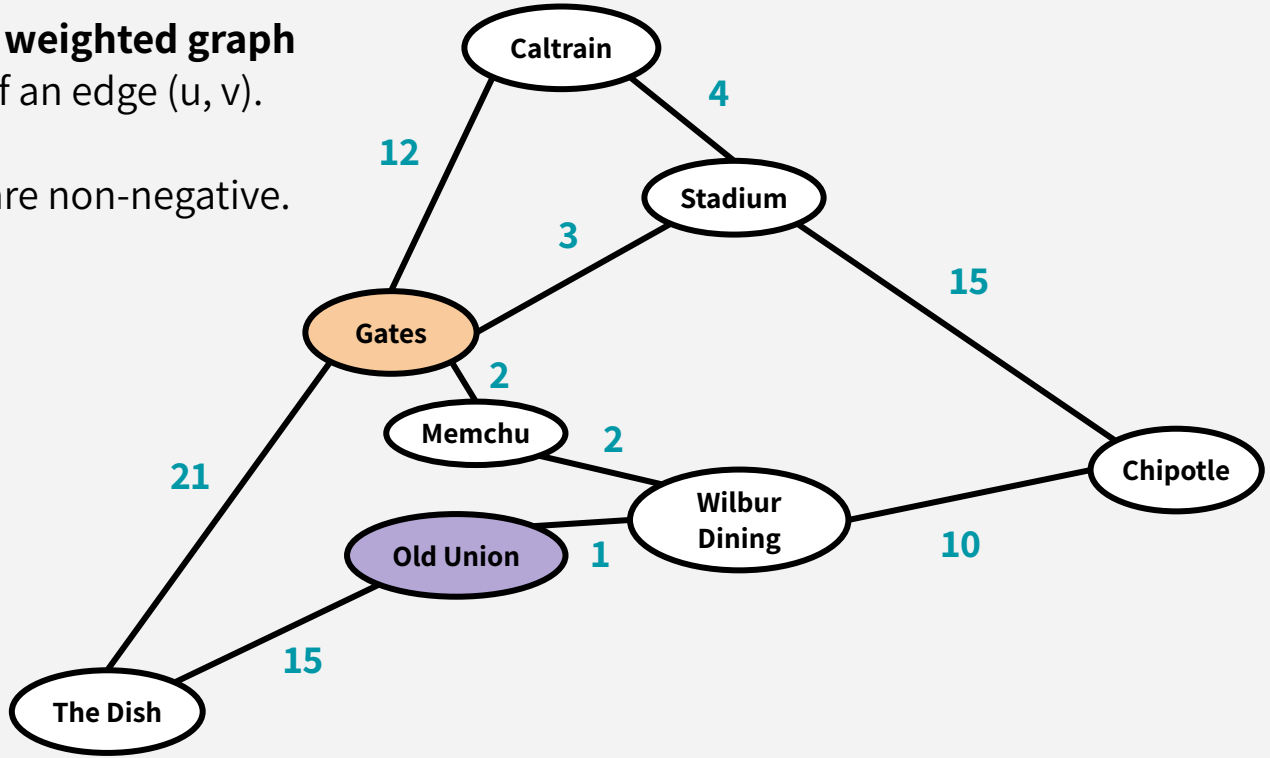
SHORTEST PATHS IN WEIGHTED GRAPHS

We can represent this as a **weighted graph** where **w(u,v)** = weight of an edge (u, v).

For today, these weights are non-negative.

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





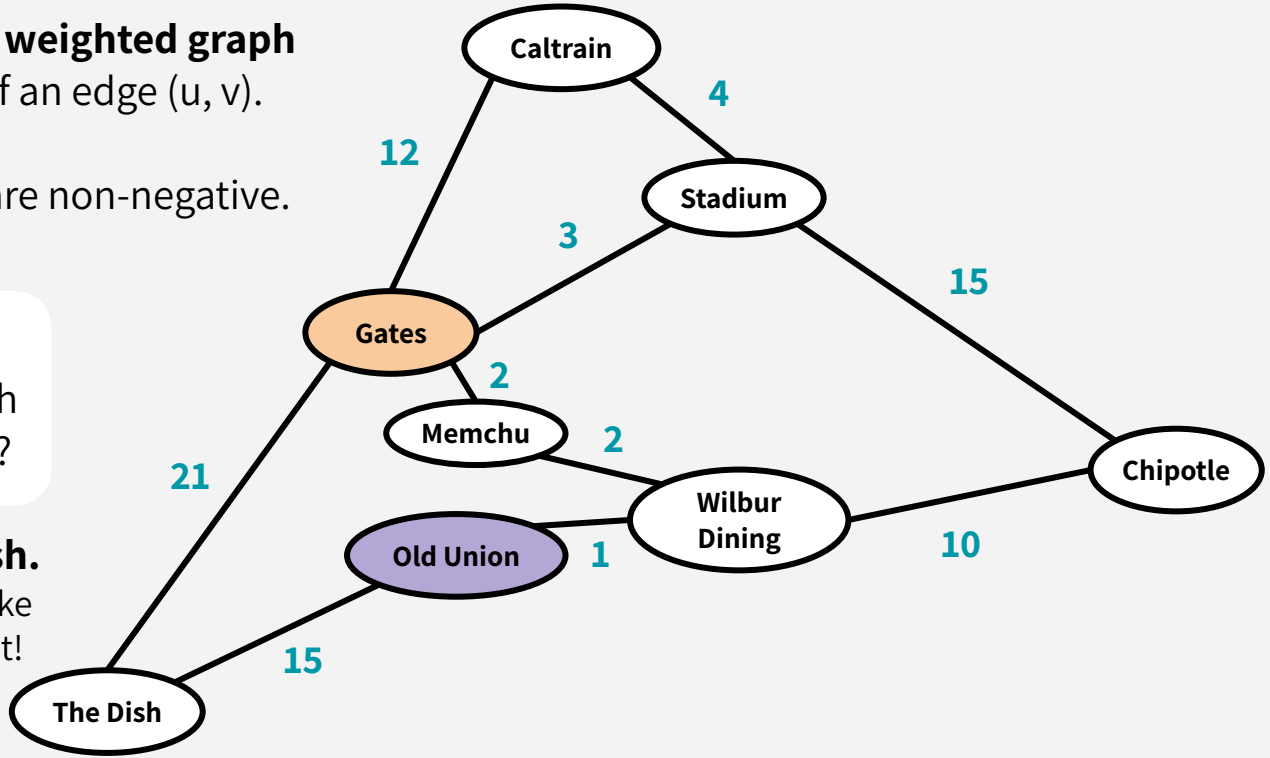
SHORTEST PATHS IN WEIGHTED GRAPHS

We can represent this as a **weighted graph** where **w(u,v)** = weight of an edge (u, v).

For today, these weights are non-negative.

What if we wanted to compute the shortest path from **Gates** to **Old Union**?

BFS would say via The Dish.
This is because BFS doesn't take any edge weights into account!



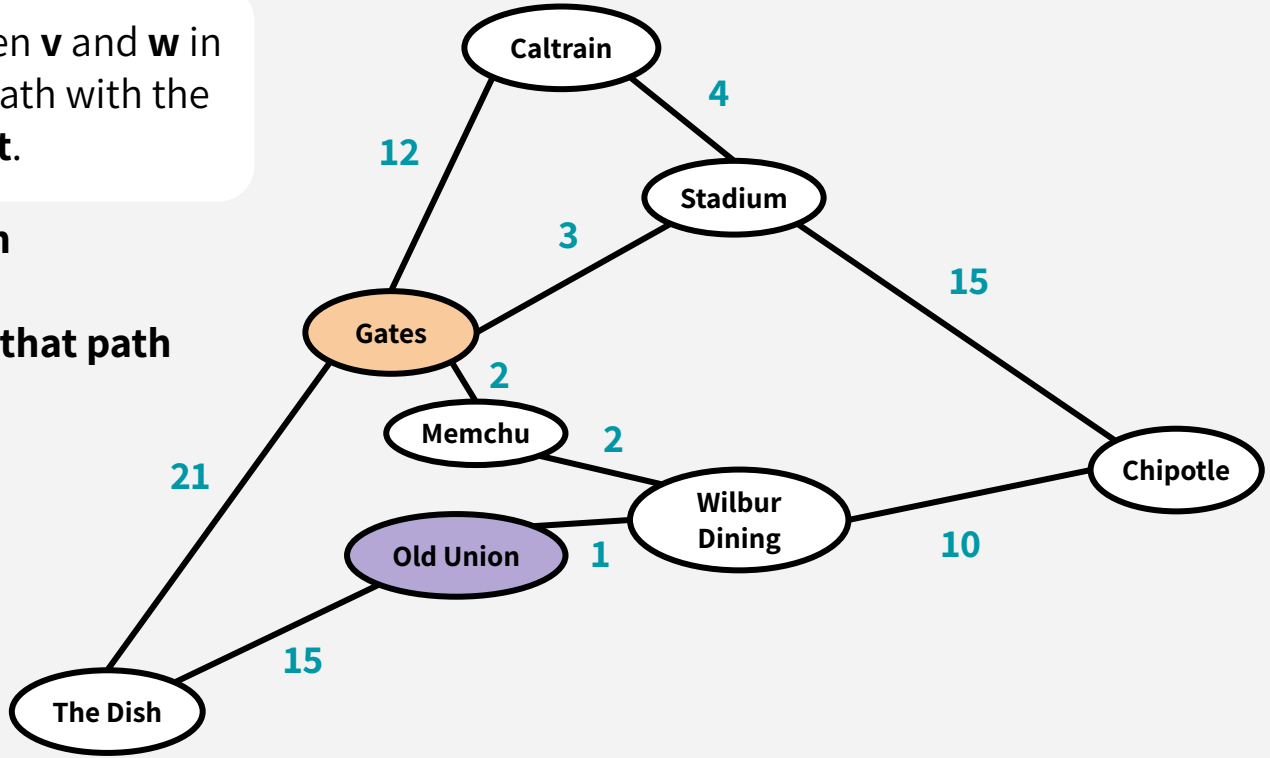
SHORTEST PATHS IN WEIGHTED GRAPHS

The **shortest path** between **v** and **w** in a weighted graph is the path with the **minimum cost**.

Cost of a path

=

sum of weights along that path



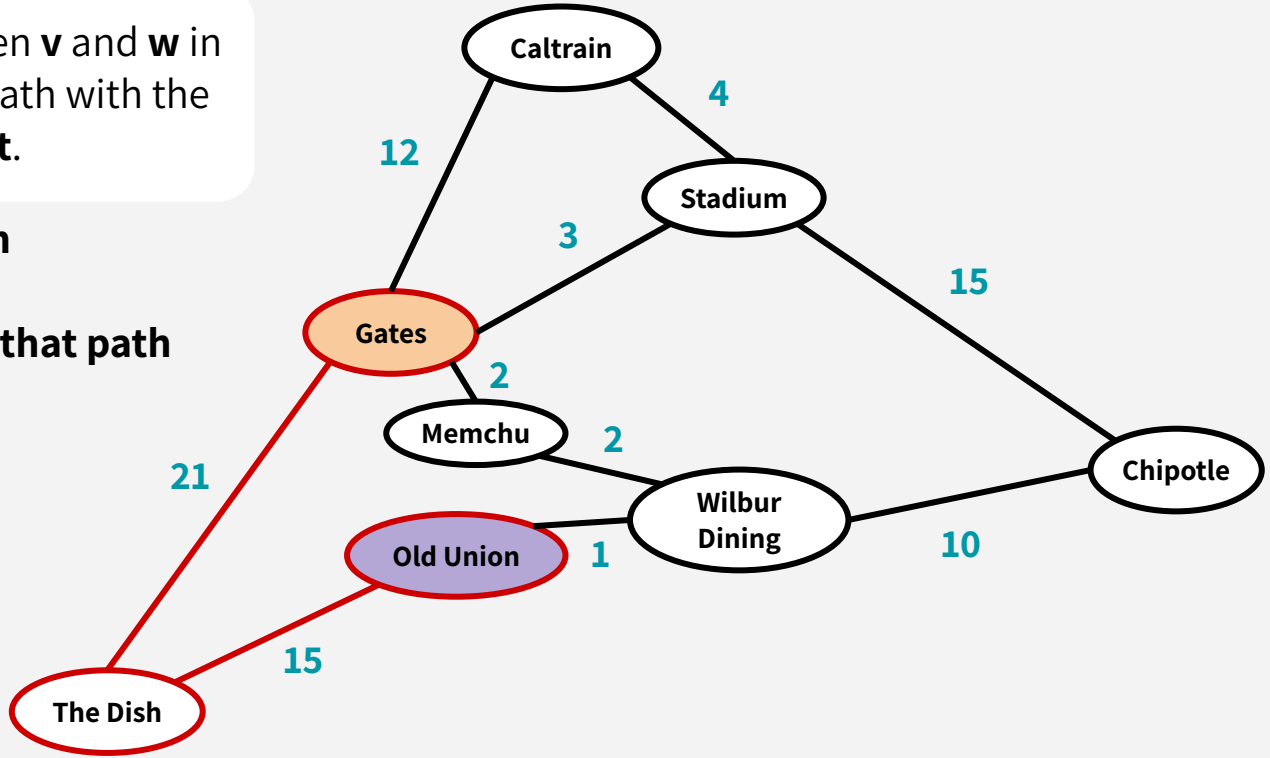
SHORTEST PATHS IN WEIGHTED GRAPHS

The **shortest path** between **v** and **w** in a weighted graph is the path with the **minimum cost**.

Cost of a path

=

sum of weights along that path



This path from Gates to Old Union has cost 36.

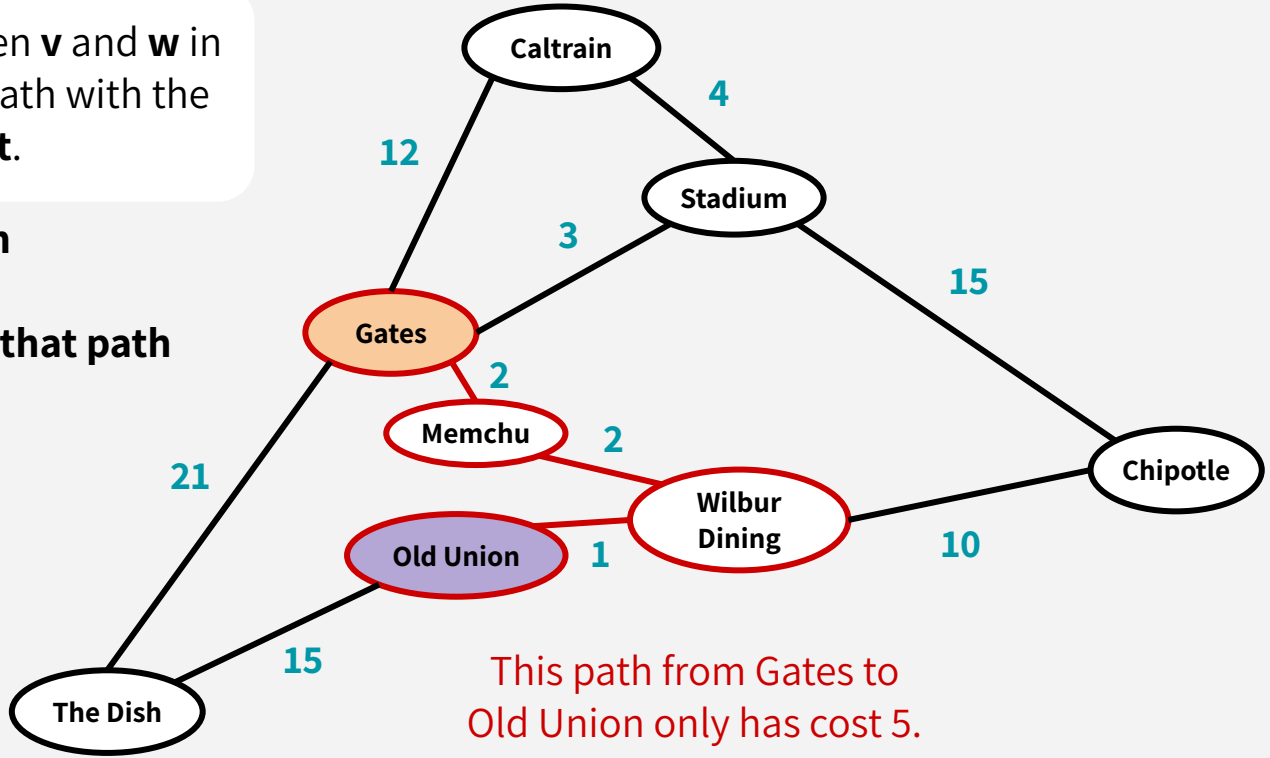
SHORTEST PATHS IN WEIGHTED GRAPHS

The **shortest path** between **v** and **w** in a weighted graph is the path with the **minimum cost**.

Cost of a path

=

sum of weights along that path

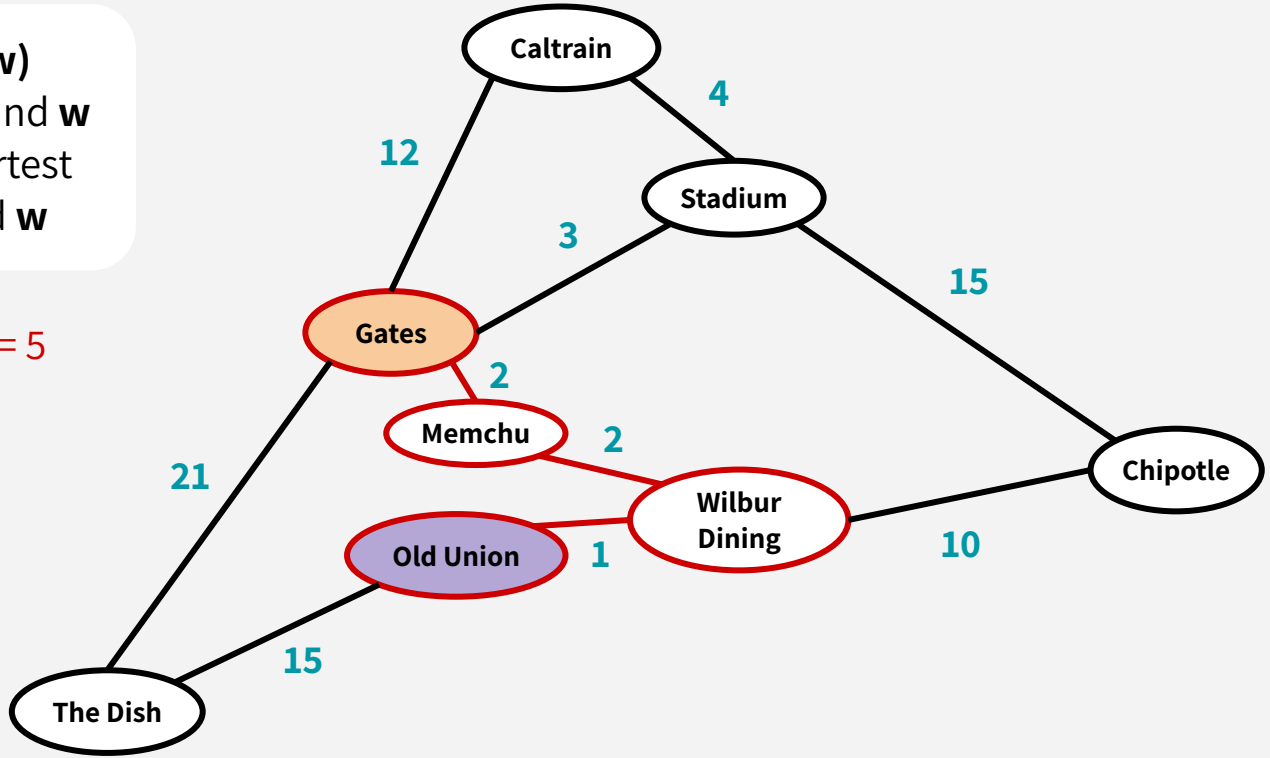


This path from Gates to Old Union only has cost 5.

SHORTEST PATHS IN WEIGHTED GRAPHS

The **distance d(v,w)** between 2 vertices **v** and **w** is the cost of the shortest path between **v** and **w**

d(Gates, Old Union) = 5



SINGLE-SOURCE SHORTEST PATH

How long is the shortest path between vertex v and *all other vertices* w?

On Monday, we saw that we can use BFS & the BFS layers to pretty easily solve this task for **unweighted graphs**.

Dijkstra's algorithm solves the single-source shortest path problem on **weighted graphs** with non-negative edge weights.

Applications:

Finding the shortest/most efficient path from point A to point B via bike, walking, Uber, Lyft, train, etc.
(Edge weights could be time, money, hassle, effort)

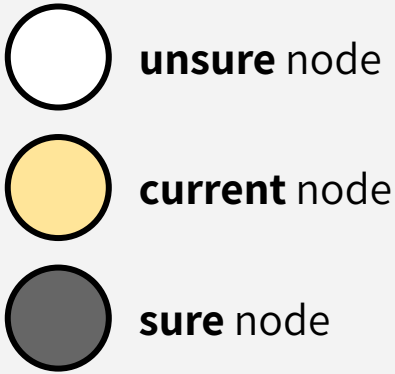
Finding the shortest path to send packets from my computer to some desired server using the Internet
(Edge weights could be link length, traffic, etc.)

DIJKSTRA'S ALGORITHM: THE IDEA

Source node is **s**.
For each node **v**, store an **over-estimate** of d(s, v).

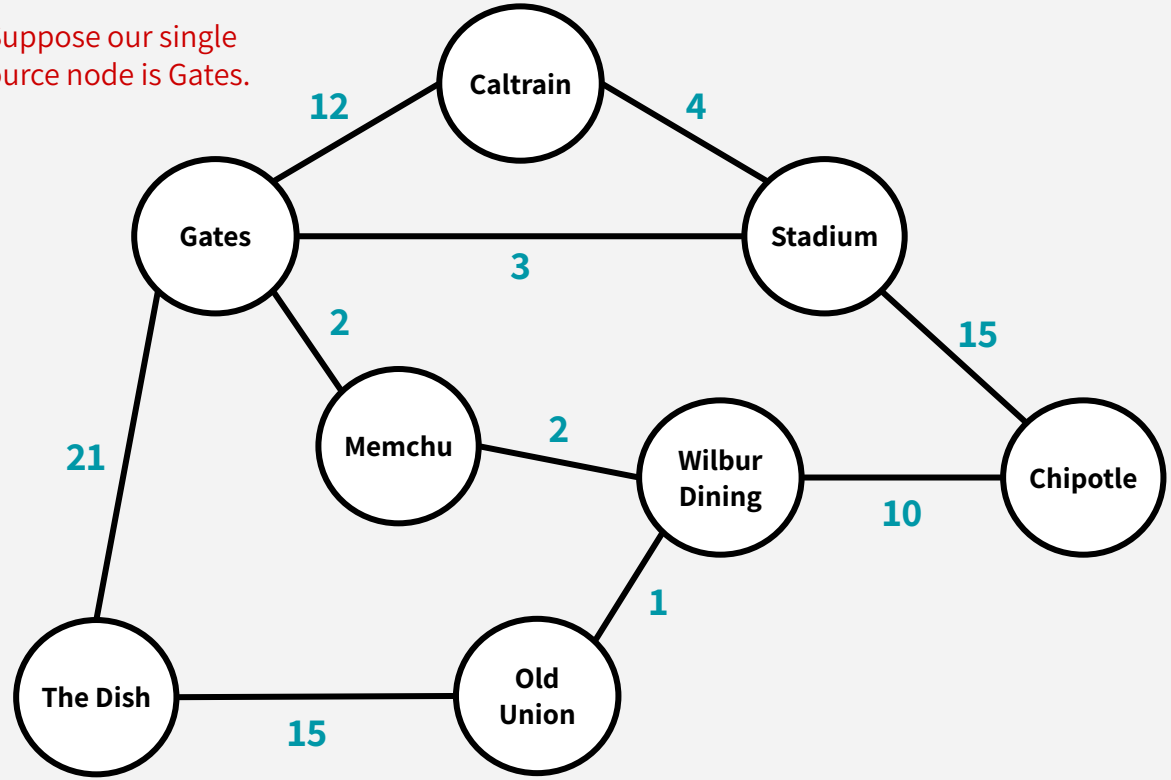
These over-estimates start off as ∞, and we'll iteratively work to make these over-estimates tighter and tighter, until they reach the true distance.

For demonstration purposes, we'll use three colors again to keep track of the state of each node:



DIJKSTRA BY EXAMPLE

Suppose our single source node is Gates.

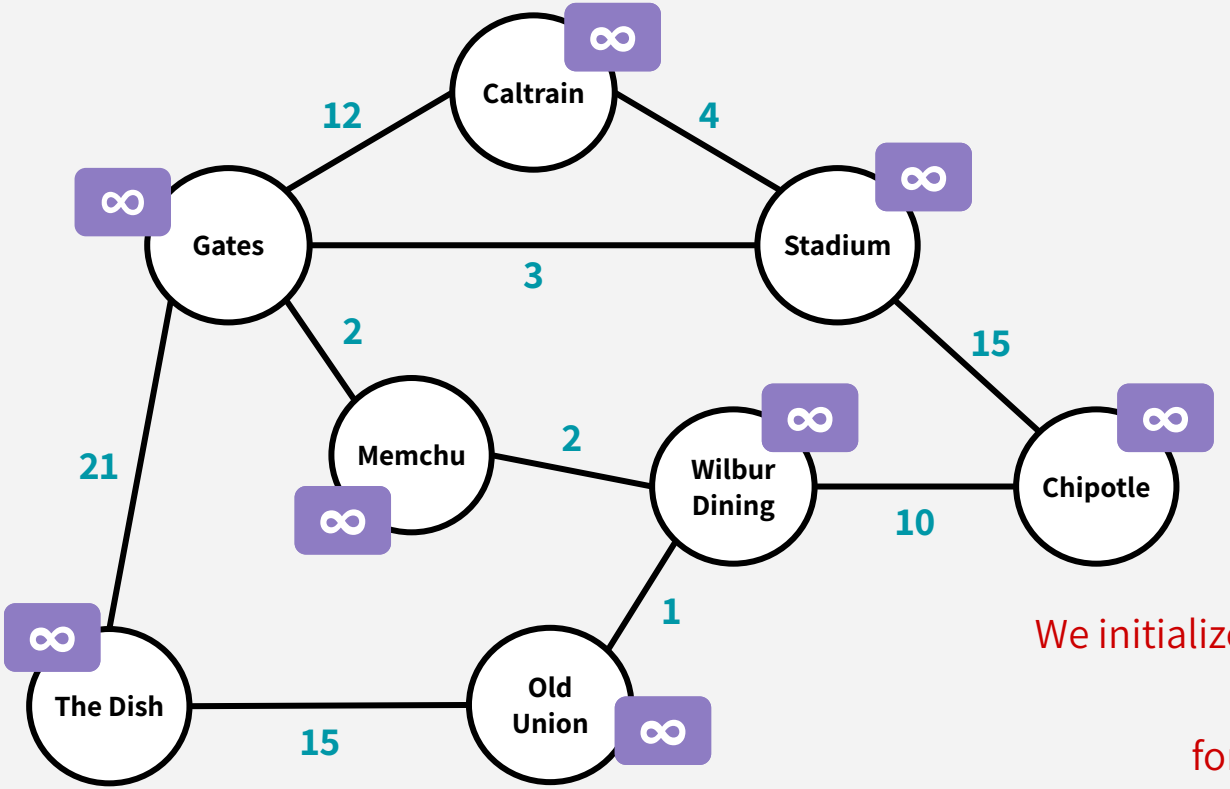


unsure node

current node

sure node

DIJKSTRA BY EXAMPLE



unsure node

current node

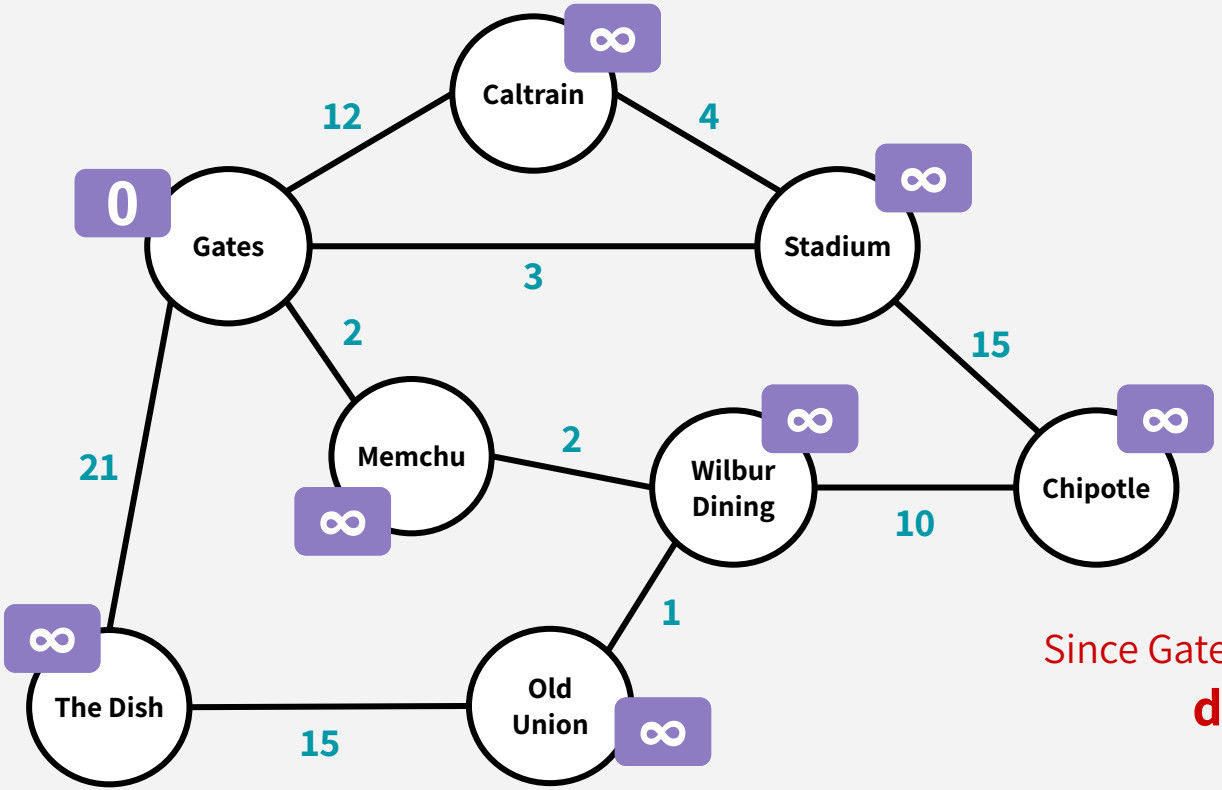
sure node

We initialize all our overestimates:

d[v] = ∞

for all vertices **v**.

DIJKSTRA BY EXAMPLE



unsure node

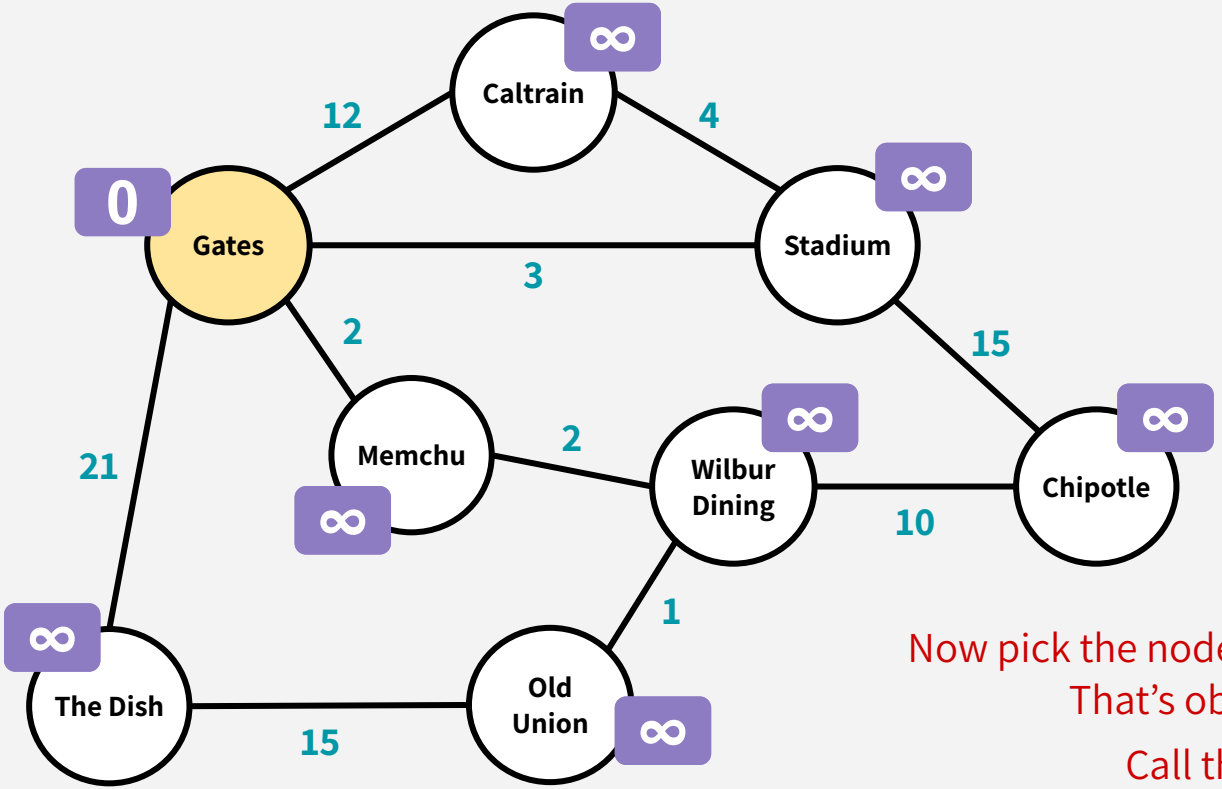
current node

sure node

Since Gates is our source, first set

d[Gates] = 0

DIJKSTRA BY EXAMPLE



unsure node

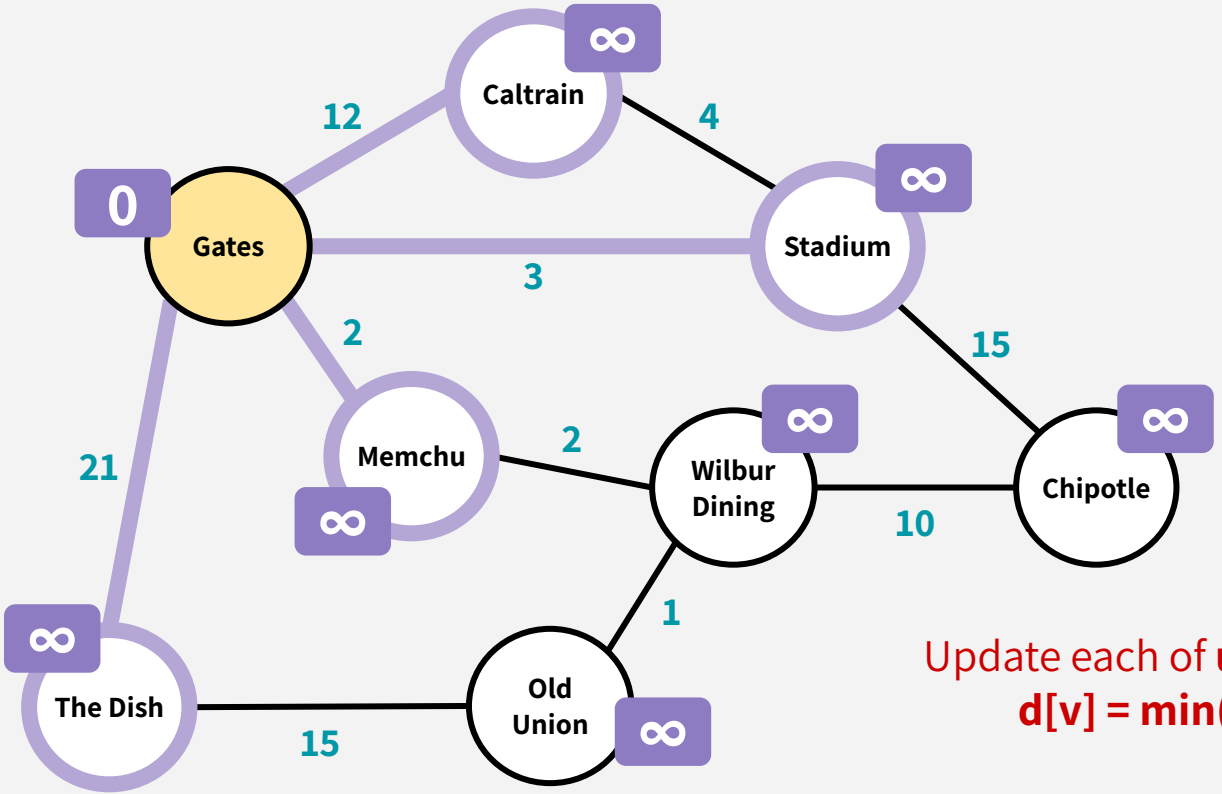
current node

sure node

Now pick the node with the smallest d[v].
That's obviously Gates.

Call this node **u**.

DIJKSTRA BY EXAMPLE



unsure node

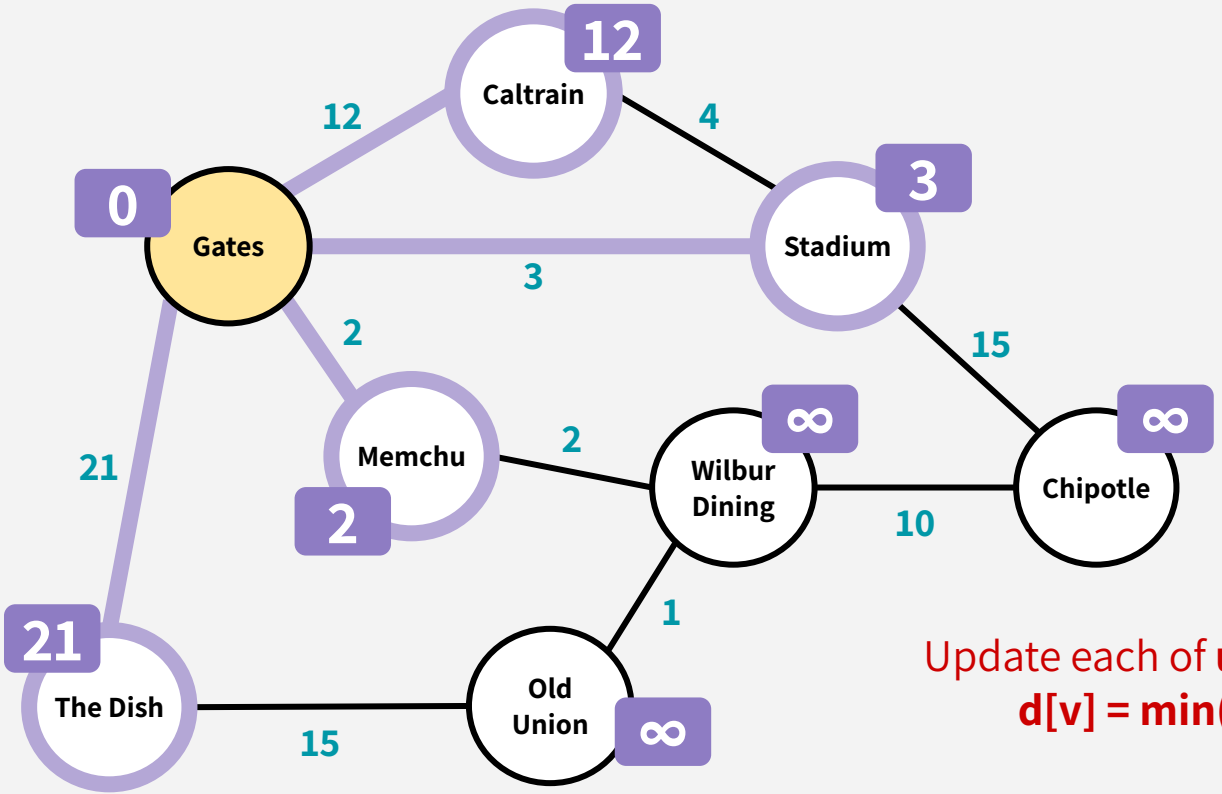
current node

sure node

Update each of **u**'s neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



unsure node

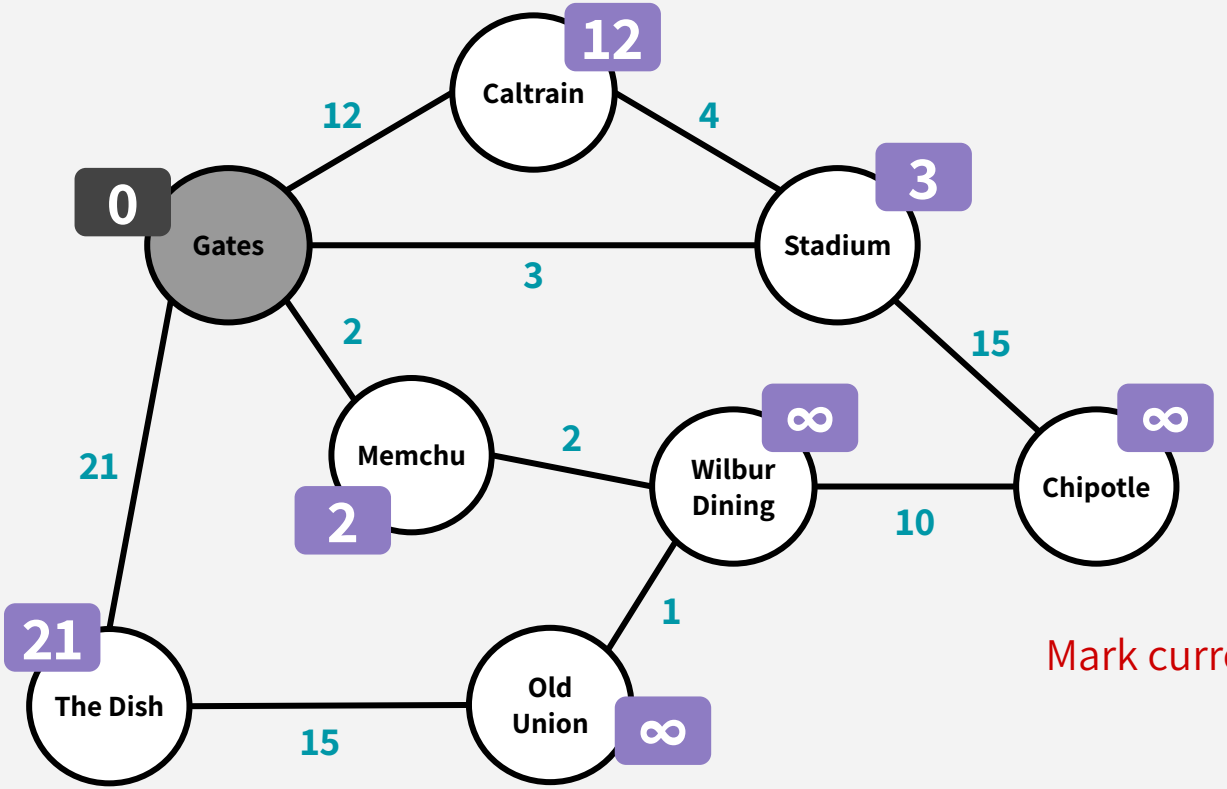
current node

sure node

Update each of **u**'s neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



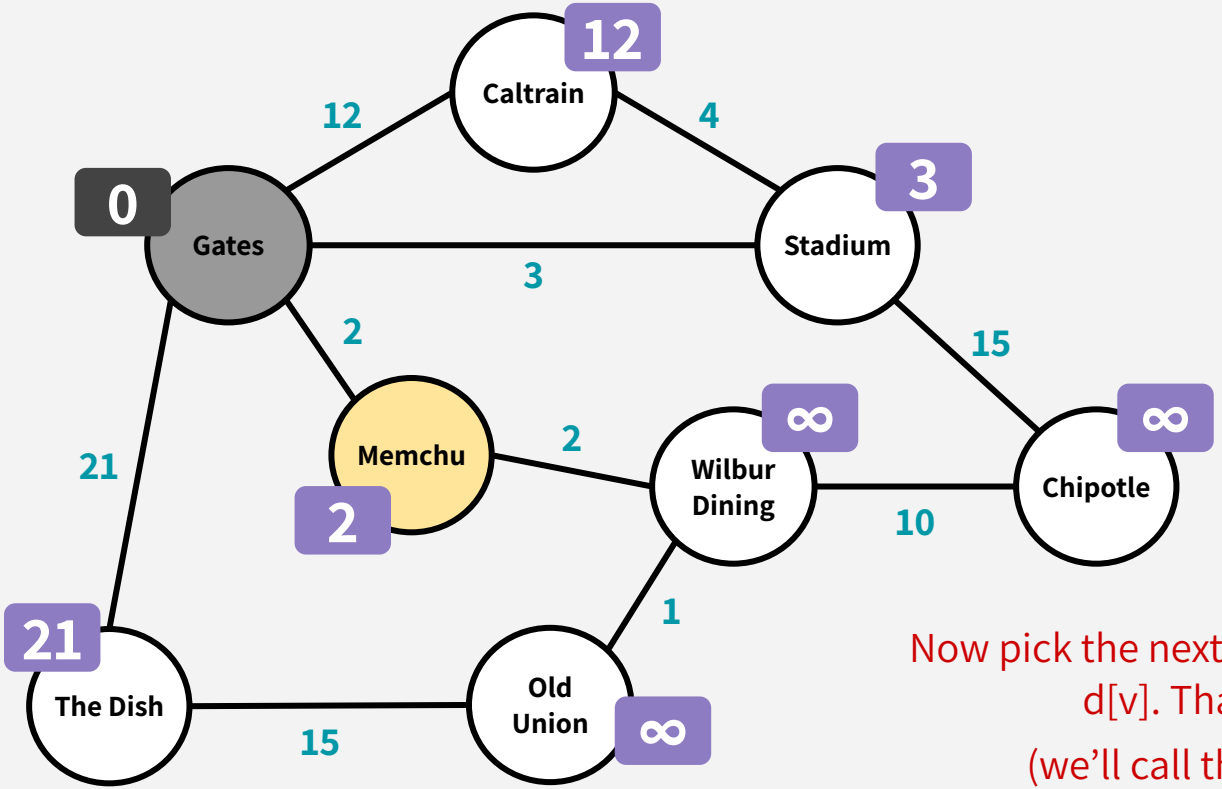
unsure node

current node

sure node

Mark current node **u** as **sure**.

DIJKSTRA BY EXAMPLE



unsure node

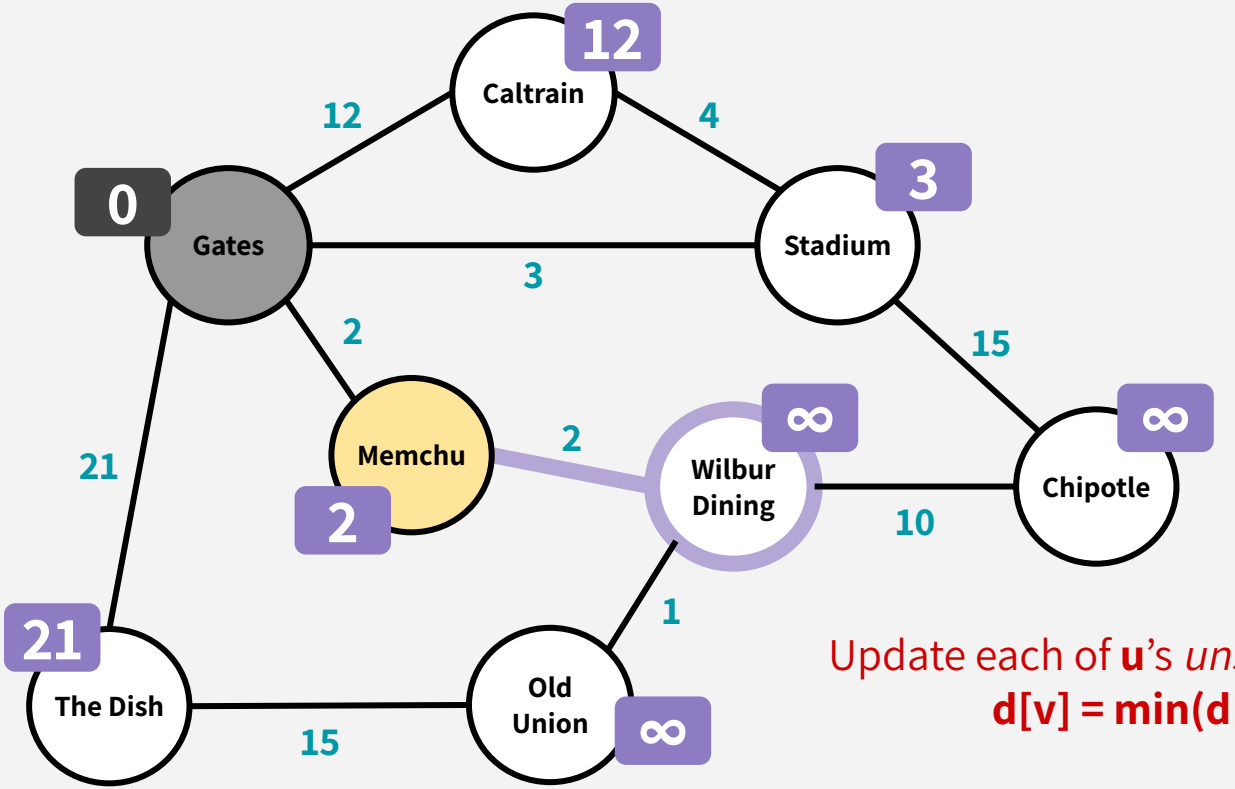
current node

sure node

Now pick the next node with the smallest d[v]. That's **MemChu**.

(we'll call this node **u** now).

DIJKSTRA BY EXAMPLE



unsure node

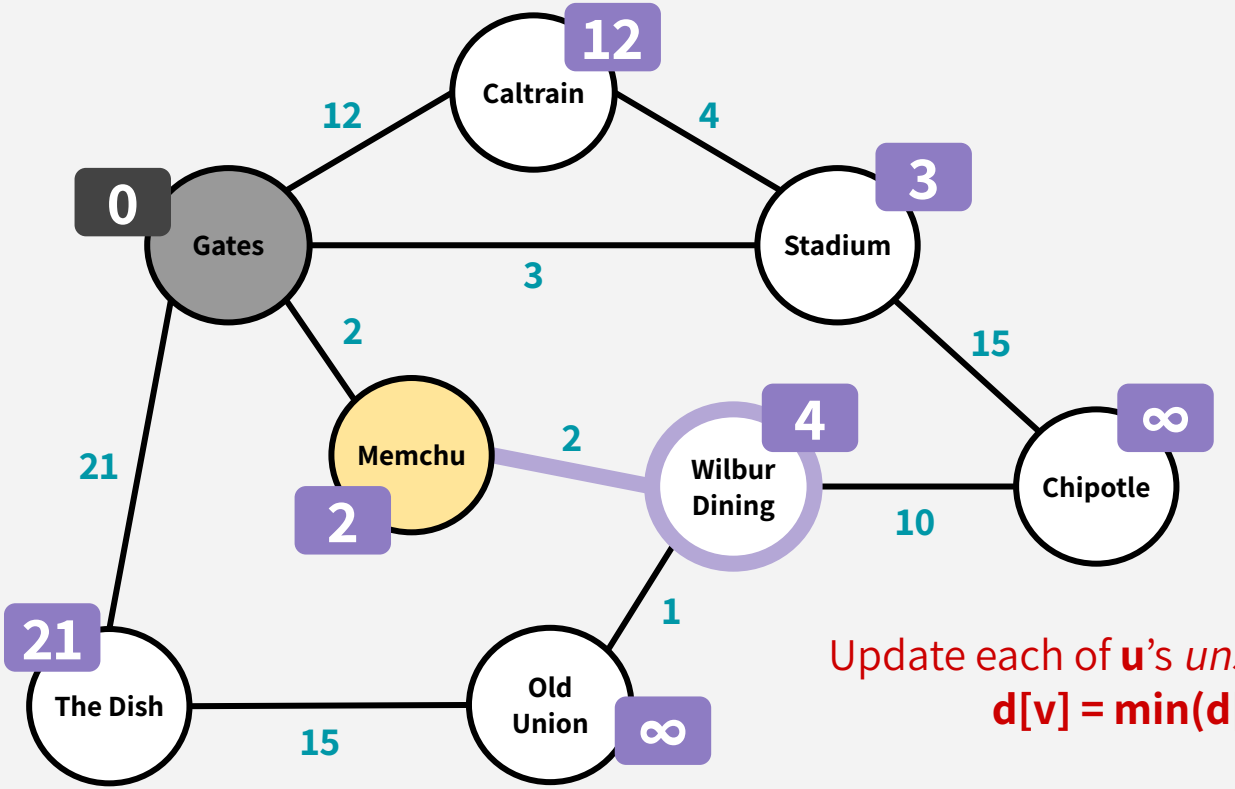
current node

sure node

Update each of **u**'s *unsure* neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



unsure node

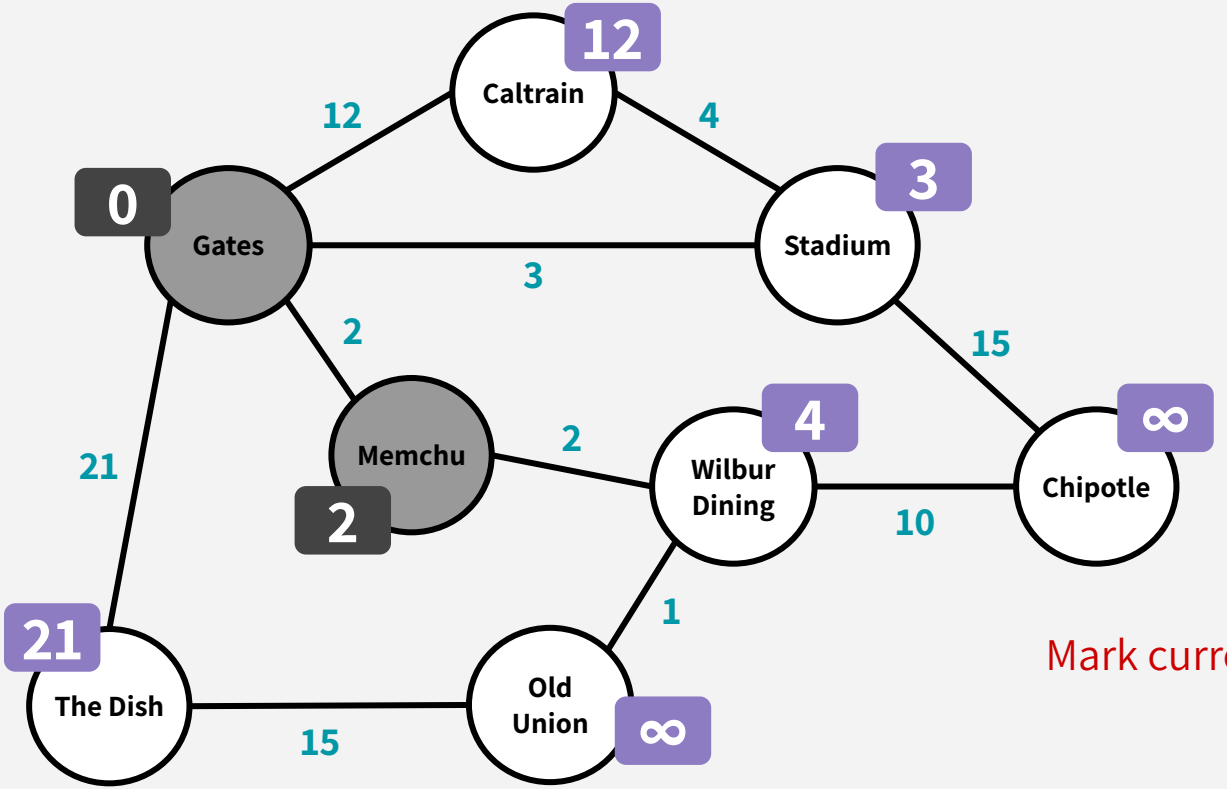
current node

sure node

Update each of **u**'s *unsure* neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



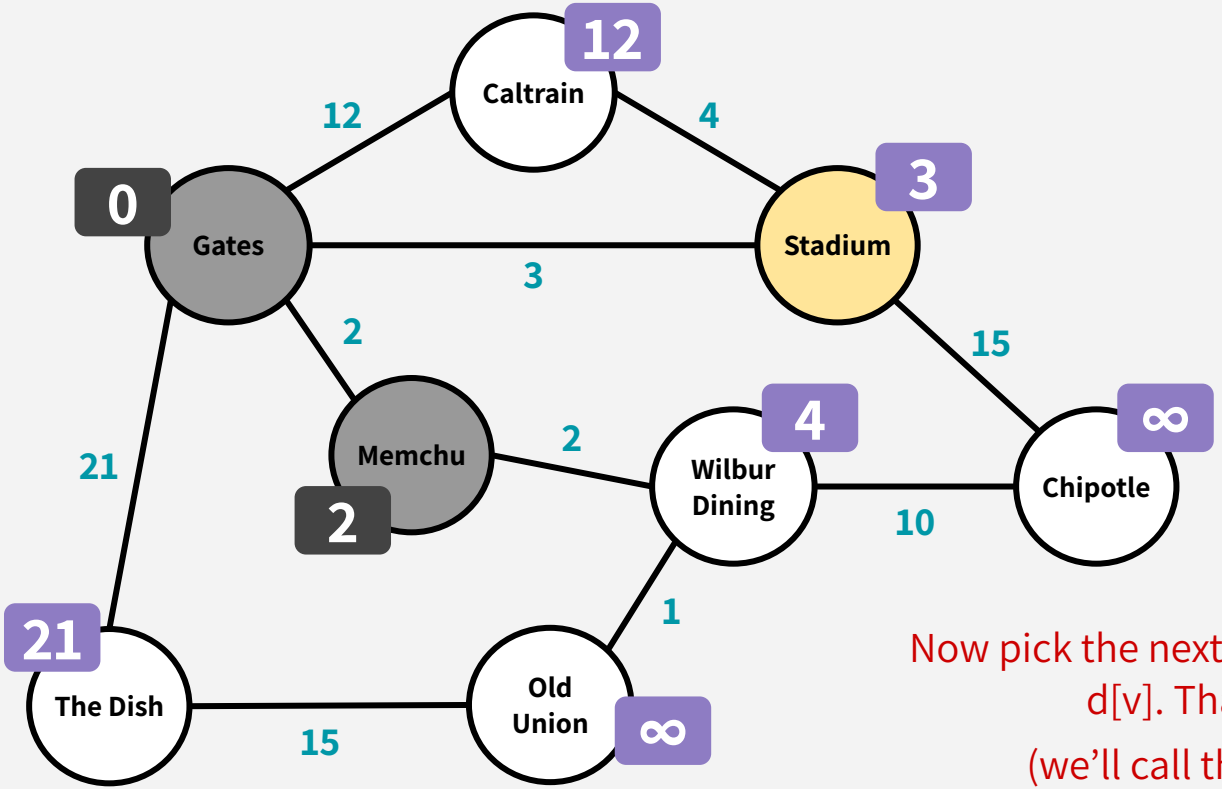
unsure node

current node

sure node

Mark current node **u** as **sure**.

DIJKSTRA BY EXAMPLE



unsure node

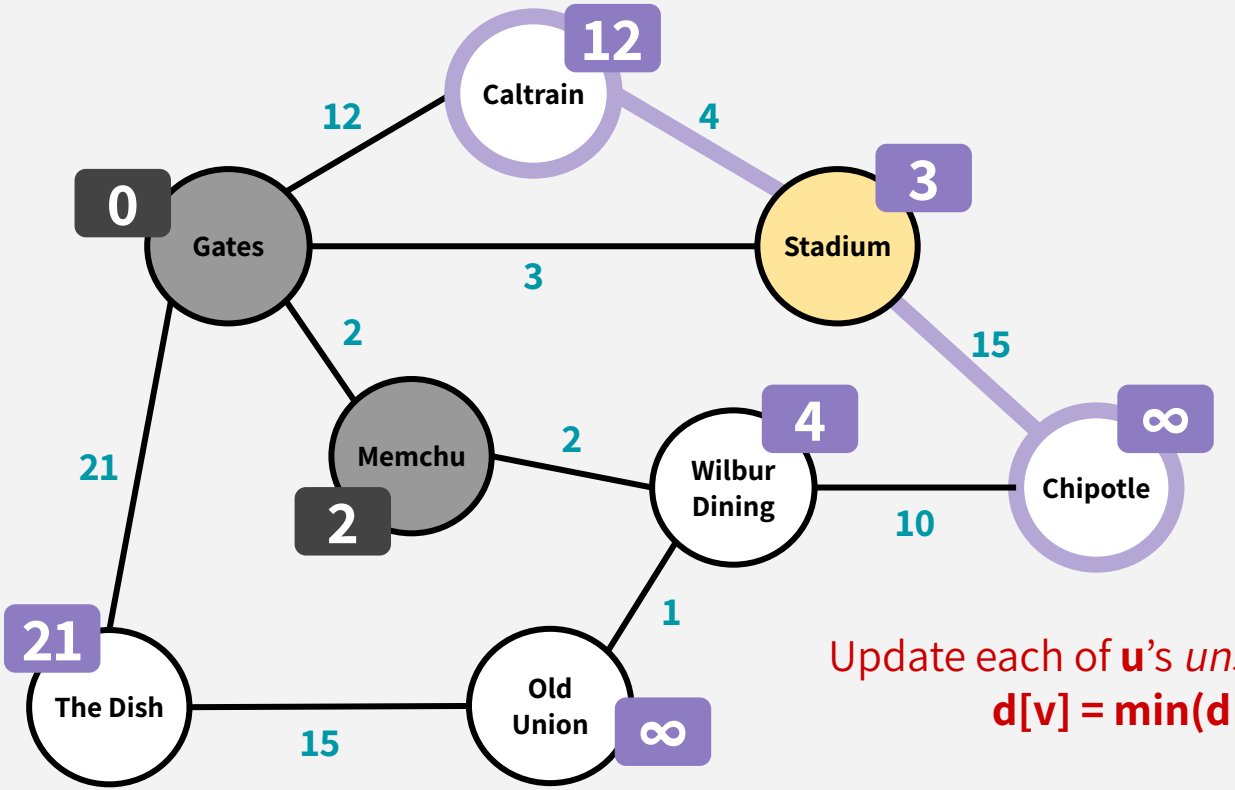
current node

sure node

Now pick the next node with the smallest d[v]. That's **Stadium**.

(we'll call this node **u** now).

DIJKSTRA BY EXAMPLE



unsure node

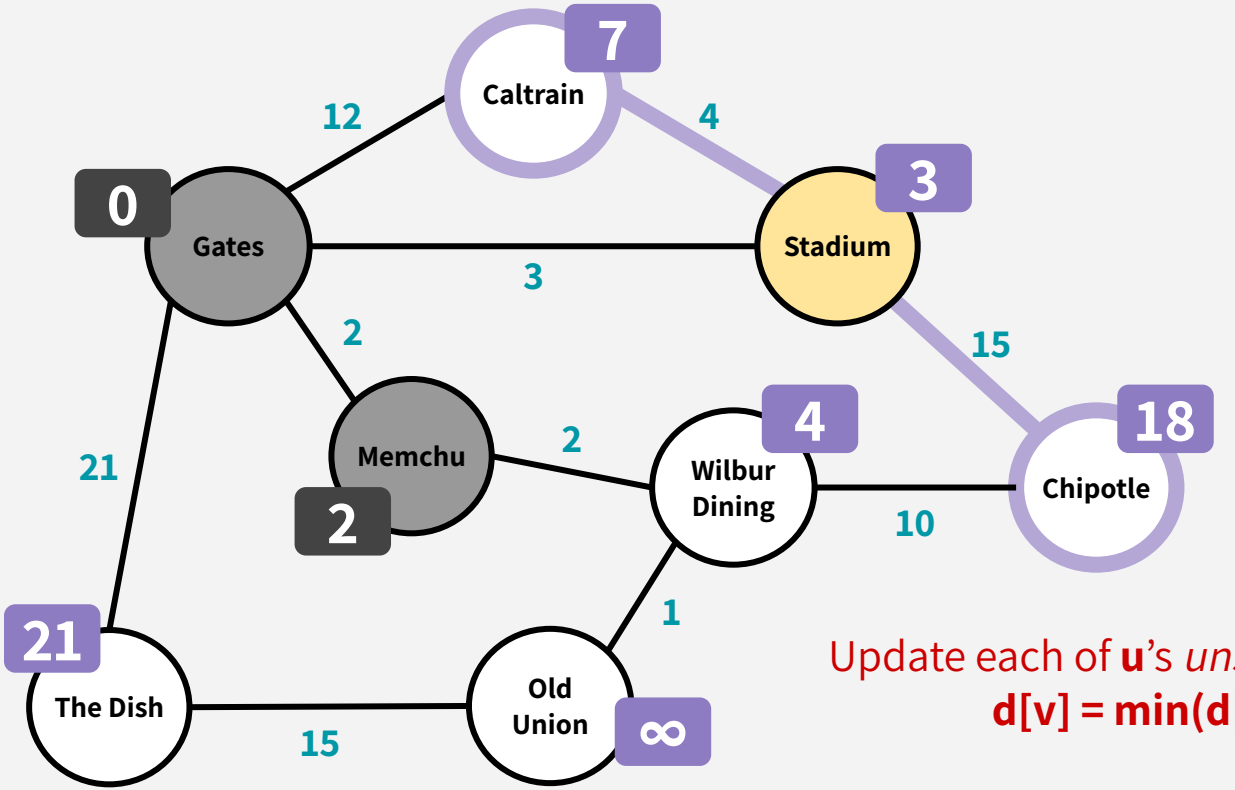
current node

sure node

Update each of **u**'s *unsure* neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



unsure node

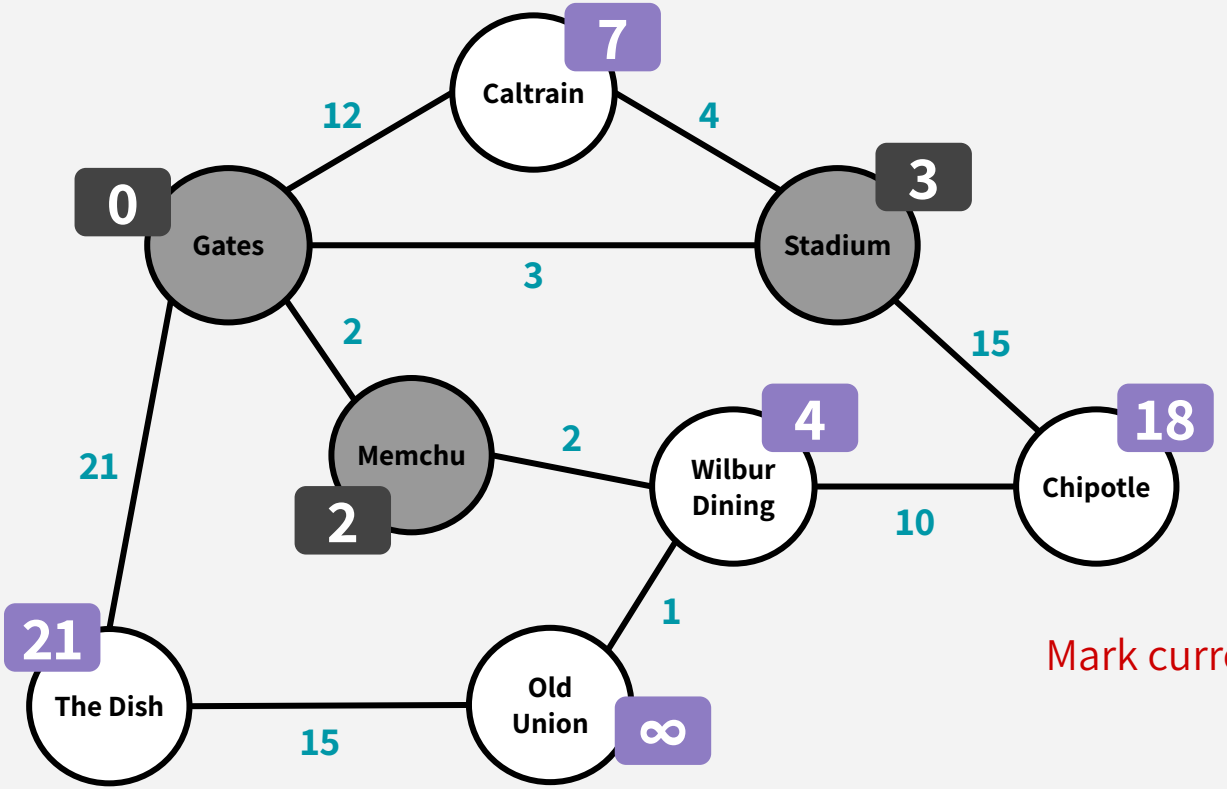
current node

sure node

Update each of **u**'s *unsure* neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



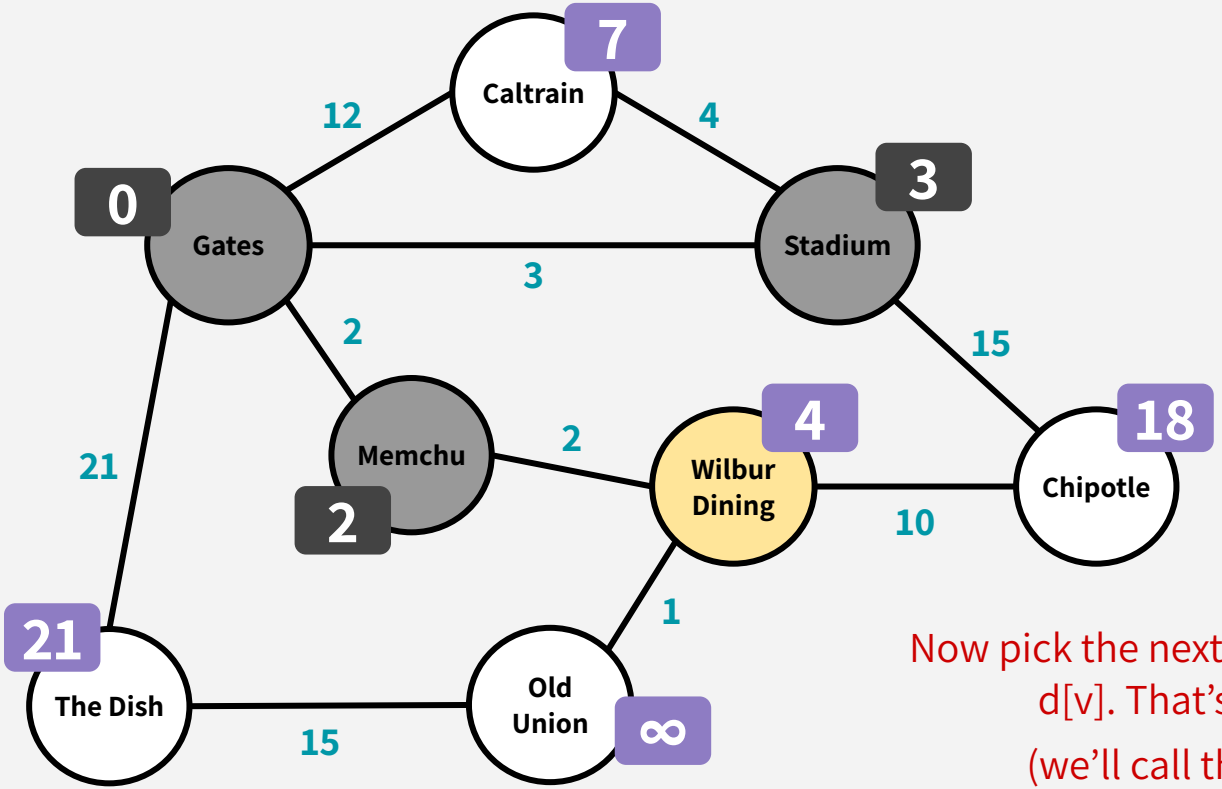
unsure node

current node

sure node

Mark current node **u** as **sure**.

DIJKSTRA BY EXAMPLE



unsure node

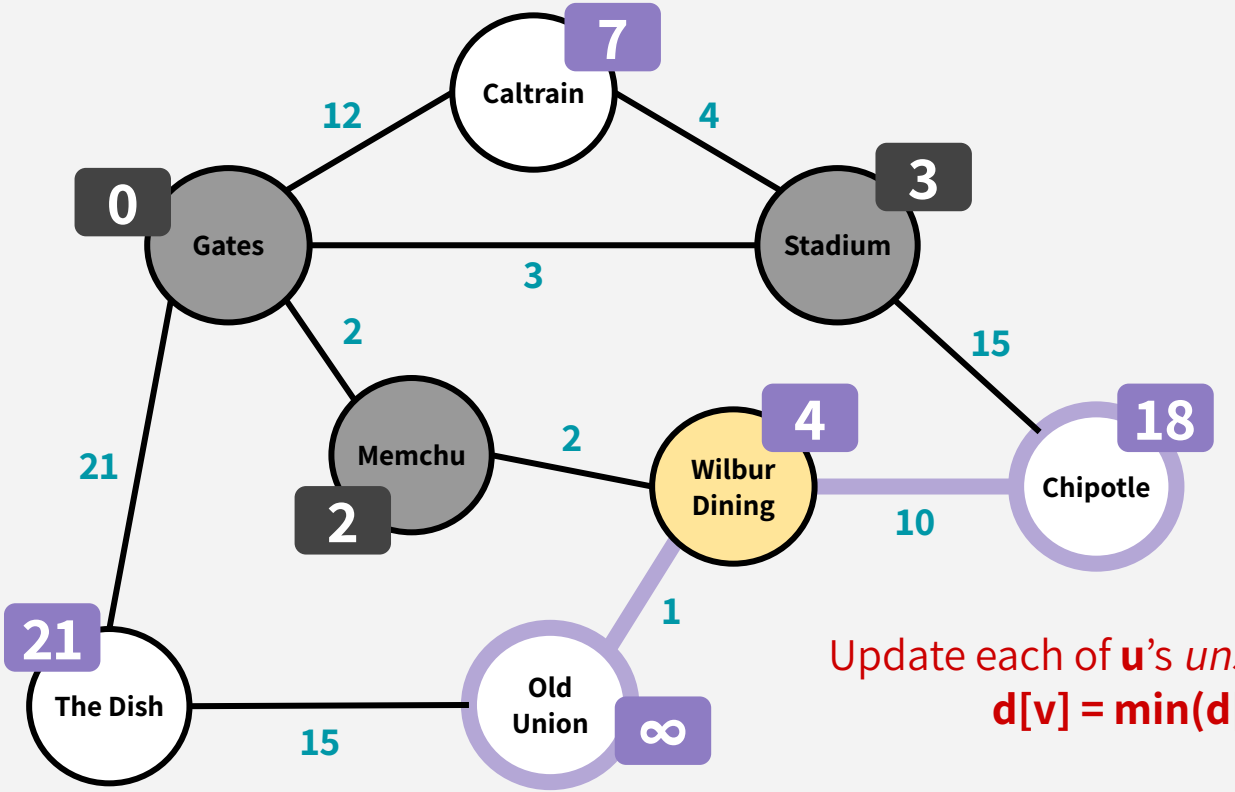
current node

sure node

Now pick the next node with the smallest d[v]. That's **Wilbur Dining**.

(we'll call this node **u** now).

DIJKSTRA BY EXAMPLE



unsure node

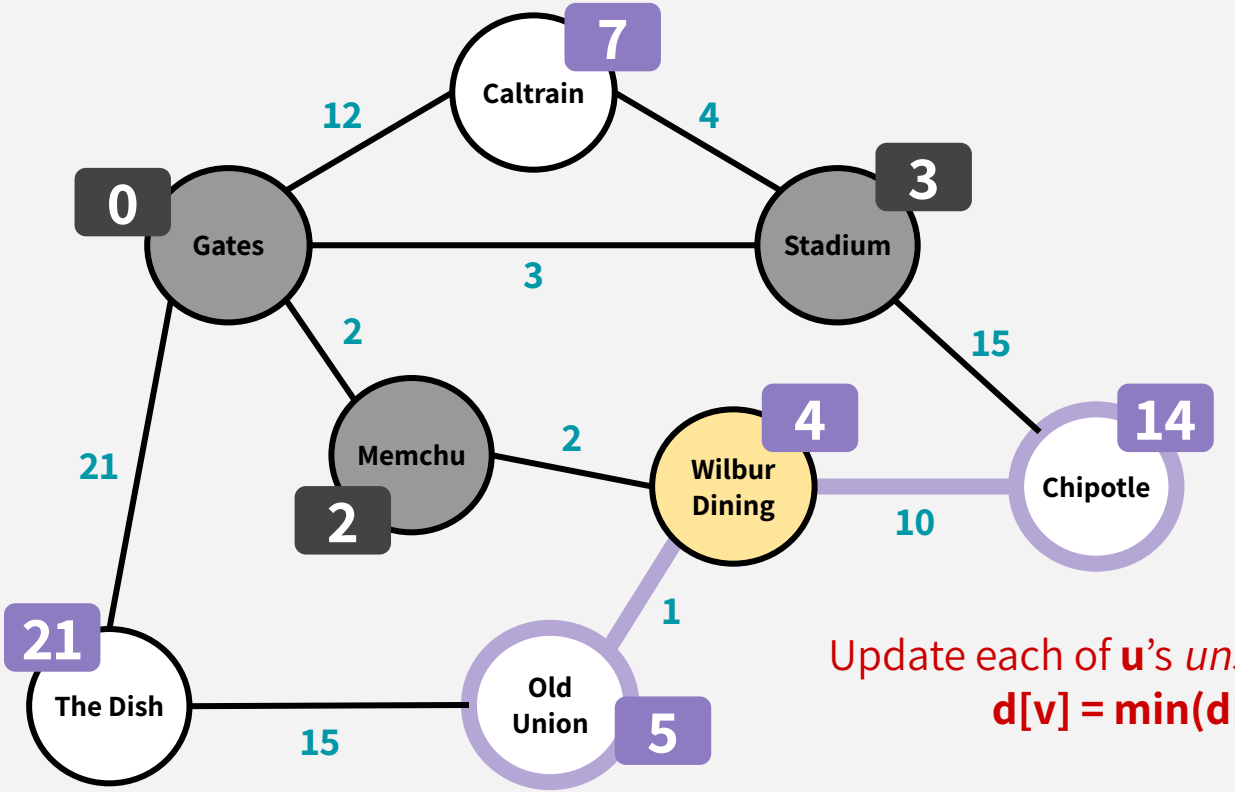
current node

sure node

Update each of **u**'s *unsure* neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



unsure node

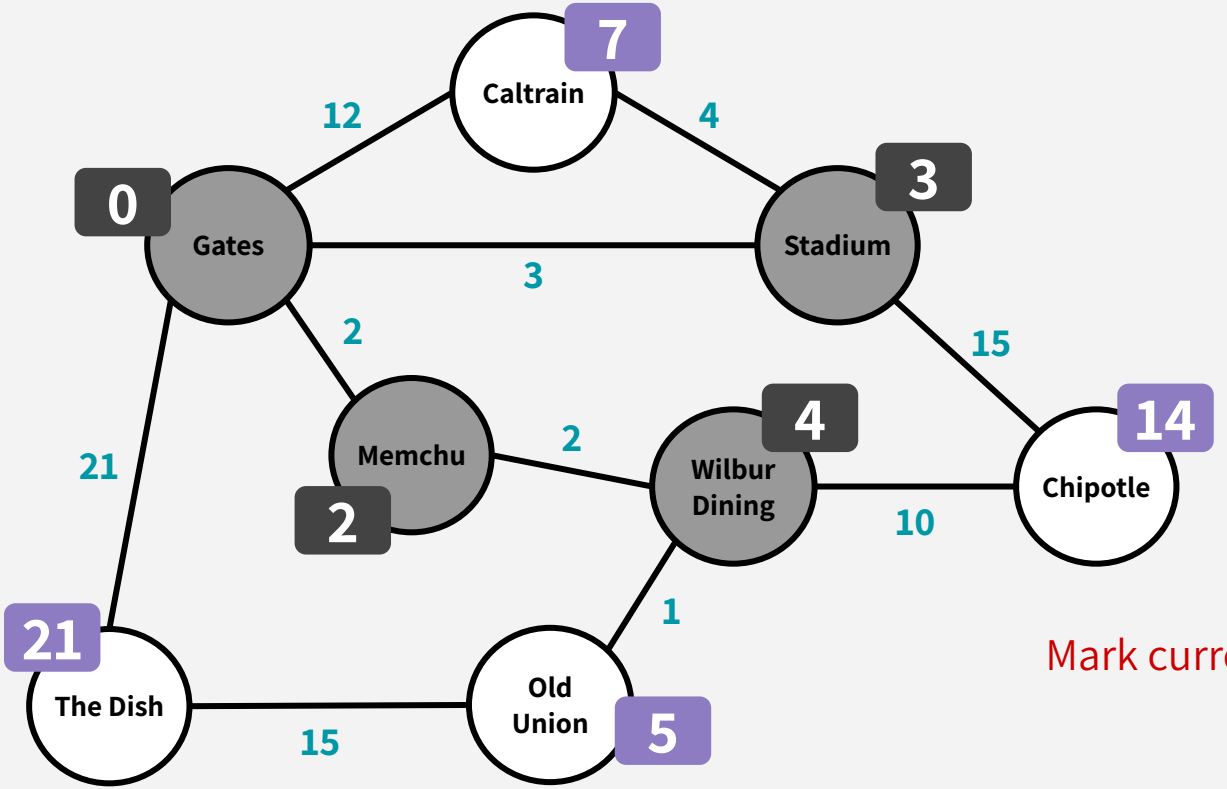
current node

sure node

Update each of **u**'s *unsure* neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



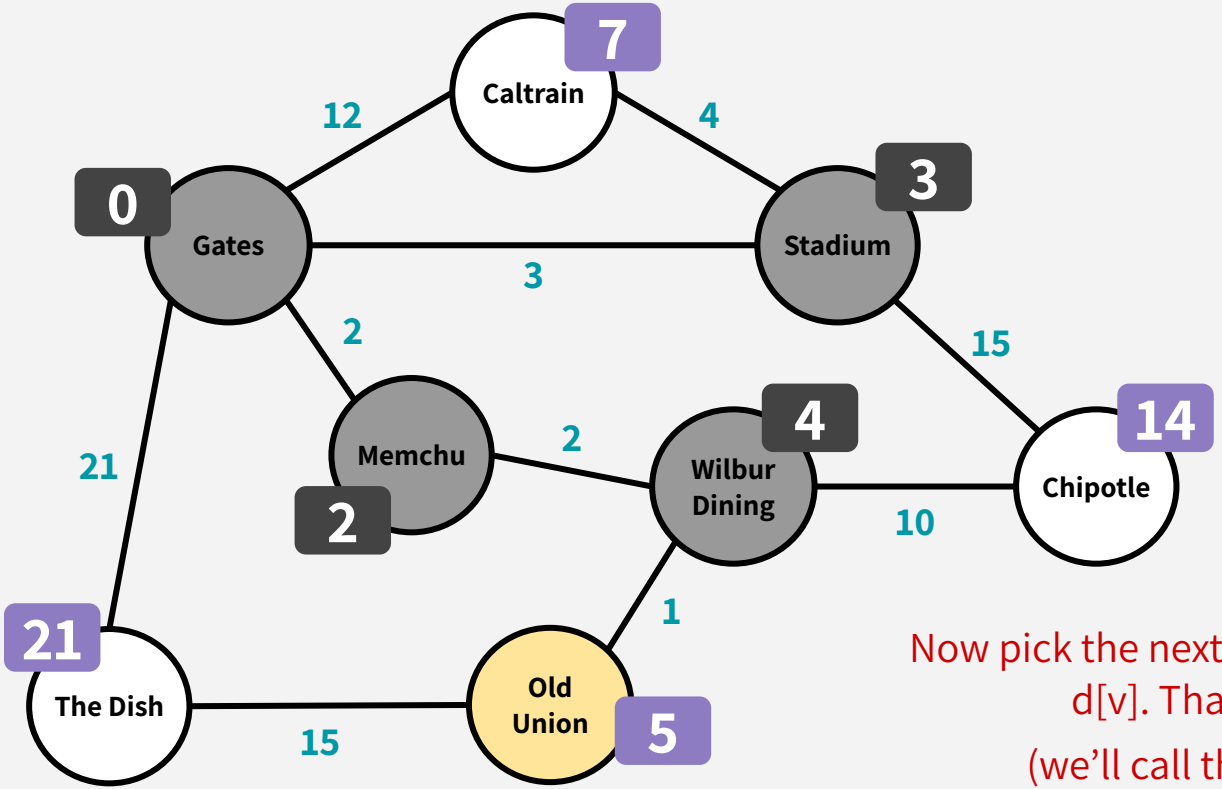
unsure node

current node

sure node

Mark current node **u** as **sure**.

DIJKSTRA BY EXAMPLE



unsure node

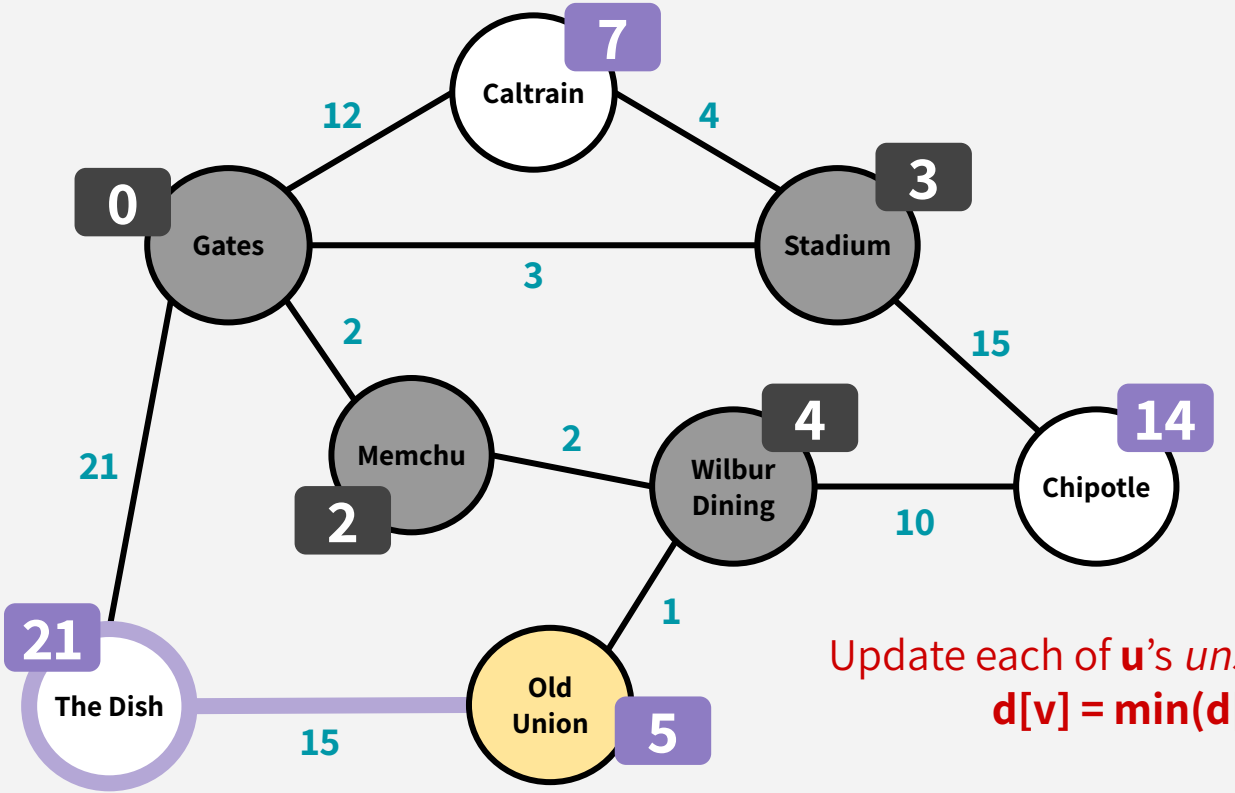
current node

sure node

Now pick the next node with the smallest d[v]. That's **Old Union**.

(we'll call this node **u** now).

DIJKSTRA BY EXAMPLE



unsure node

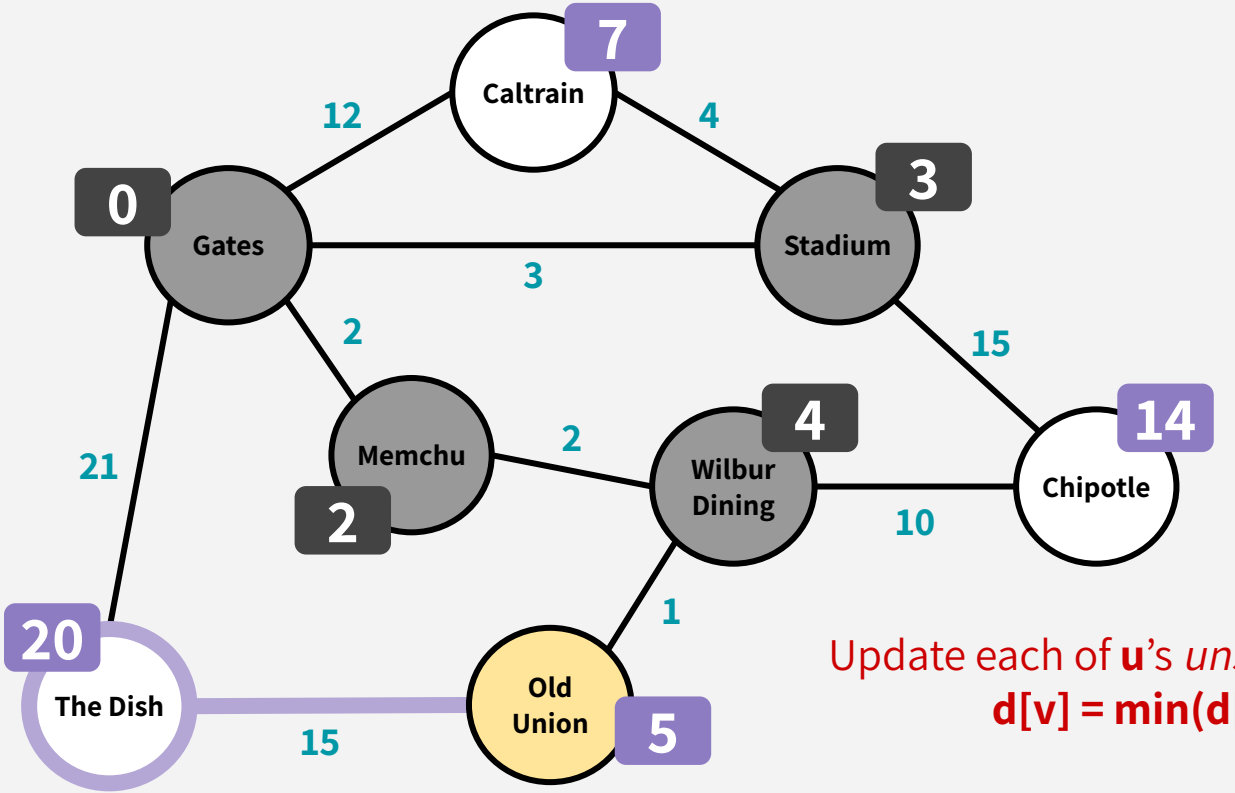
current node

sure node

Update each of **u**'s *unsure* neighbors **v** as follows:

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

DIJKSTRA BY EXAMPLE



unsure node

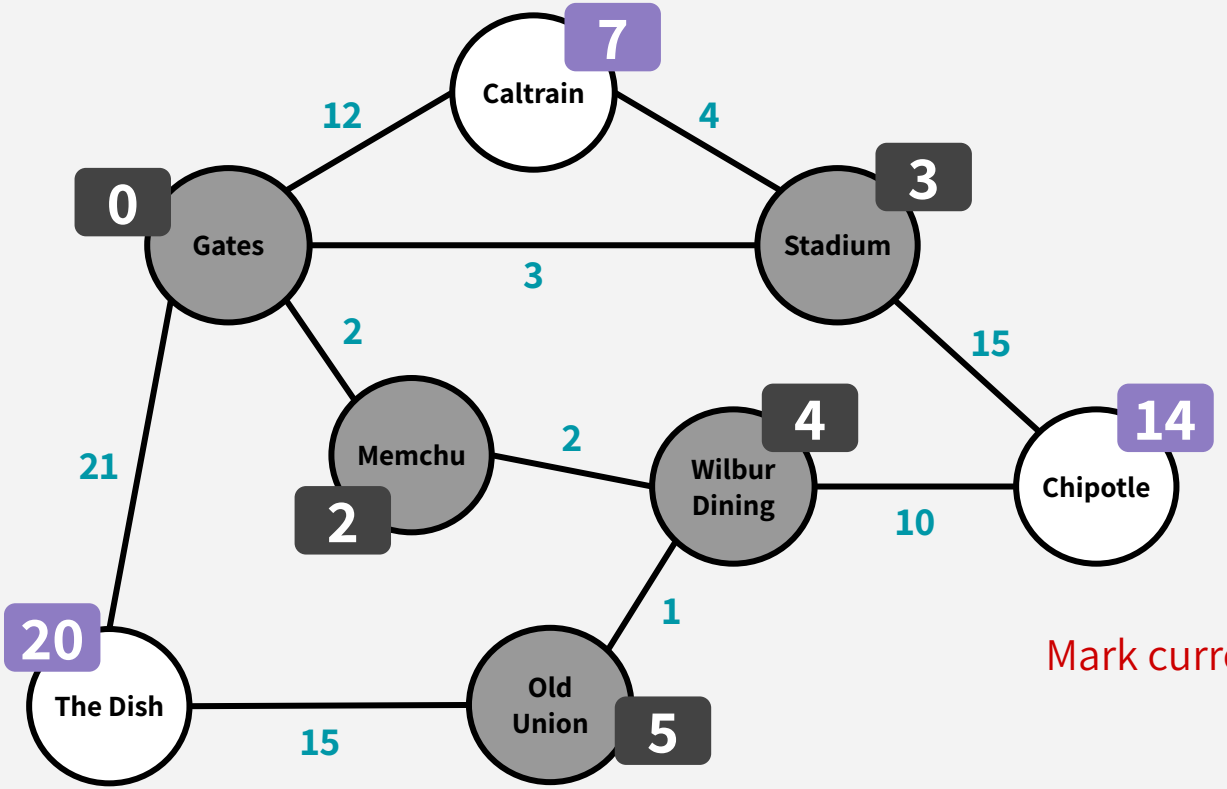
current node

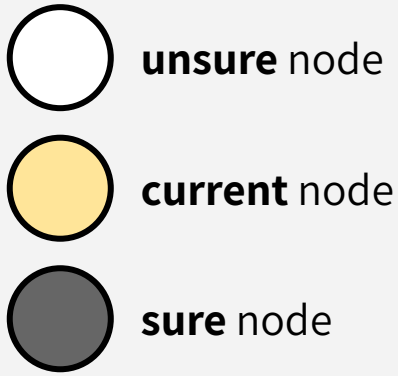
sure node

Update each of **u**'s *unsure* neighbors **v** as follows:

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

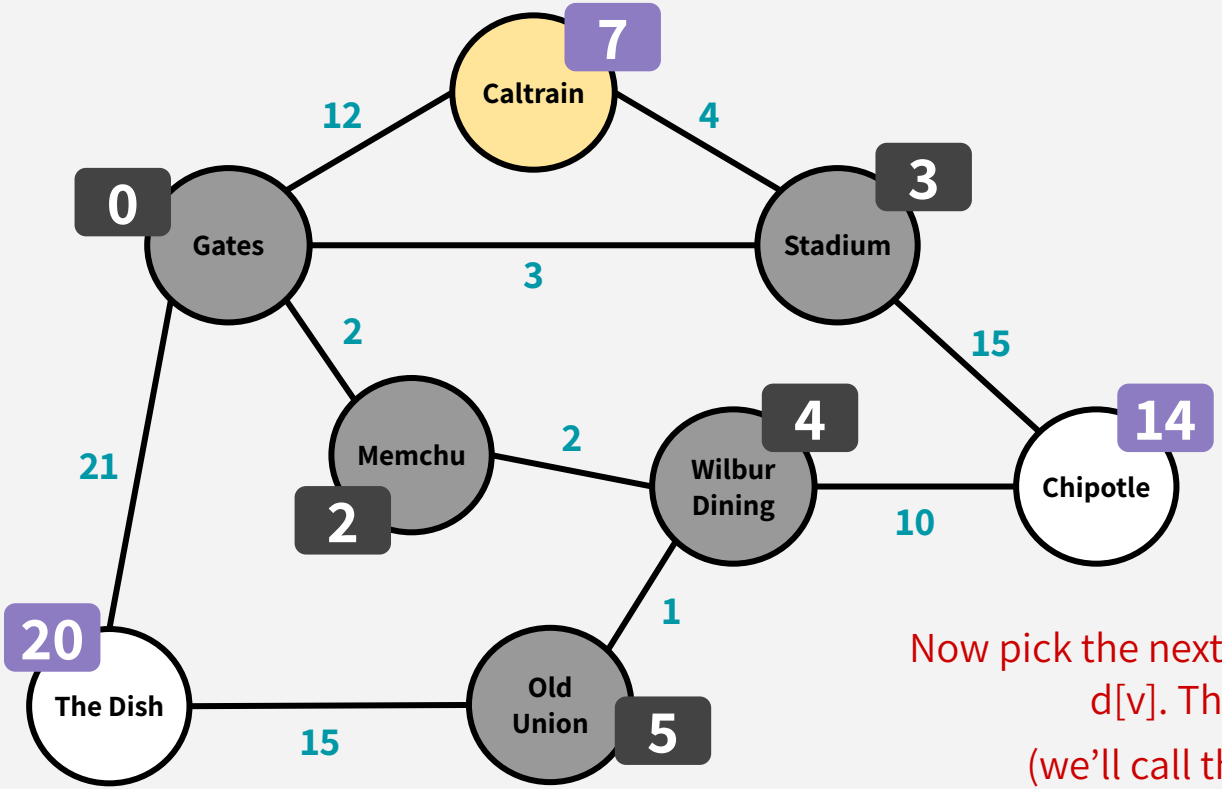
DIJKSTRA BY EXAMPLE

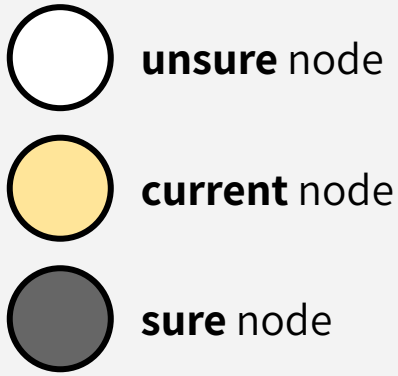




Mark current node **u** as **sure**.

DIJKSTRA BY EXAMPLE

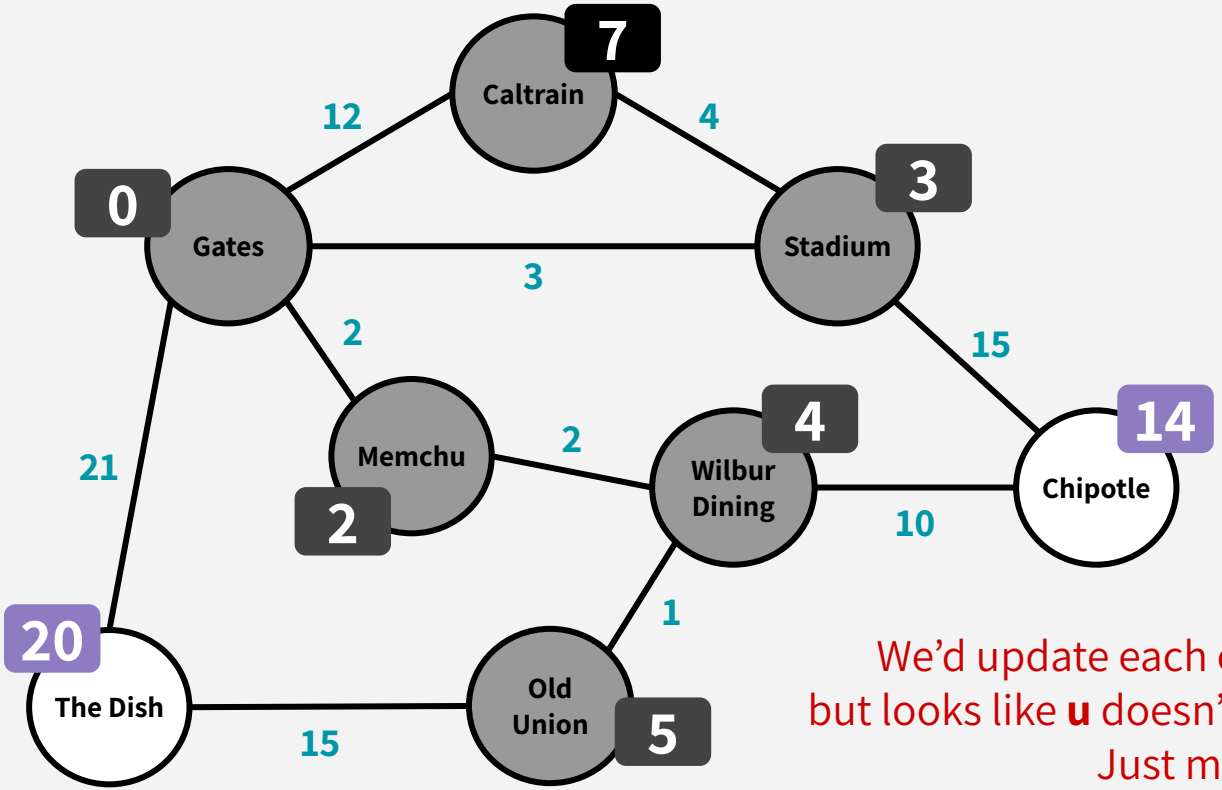




Now pick the next node with the smallest d[v]. That's **Caltrain**.

(we'll call this node **u** now).

DIJKSTRA BY EXAMPLE



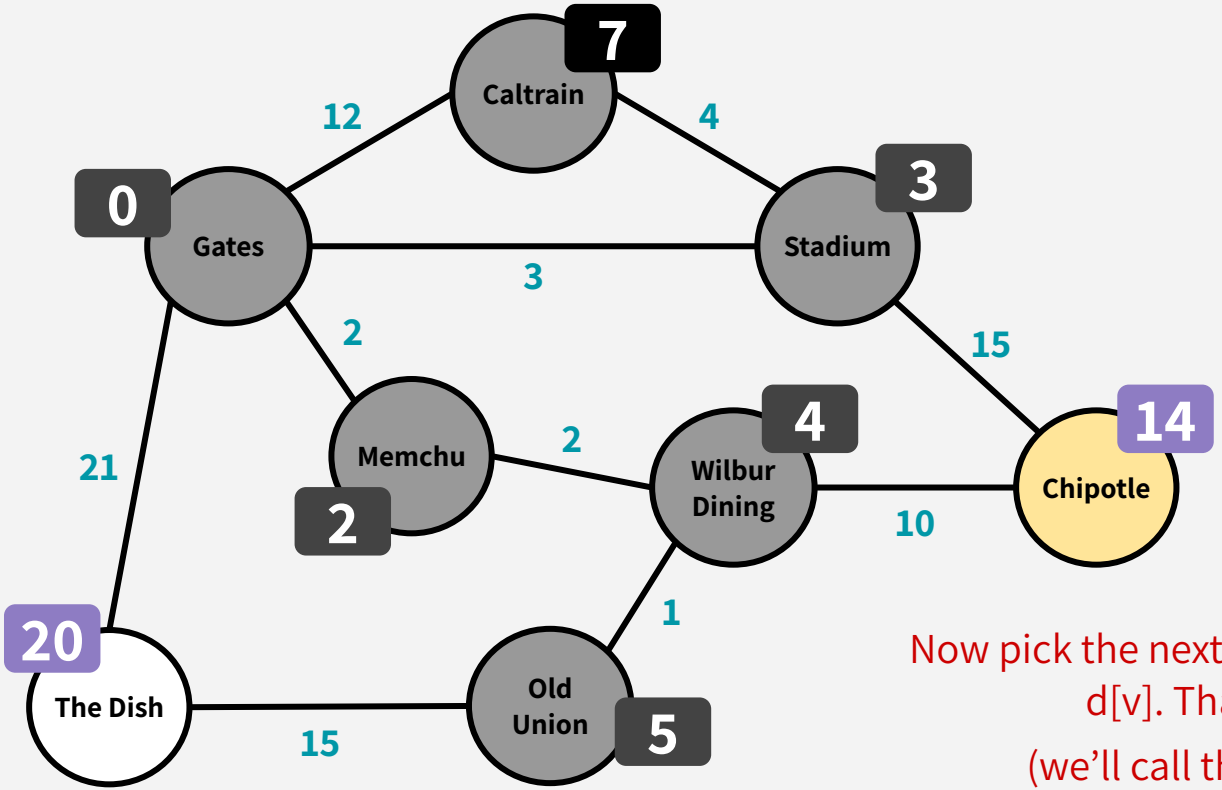
unsure node

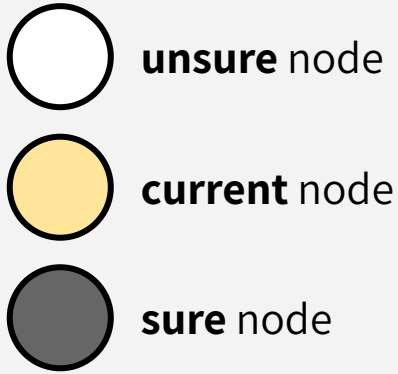
current node

sure node

We'd update each of **u**'s *unsure* neighbors **v**,
but looks like **u** doesn't have any unsure neighbors.
Just mark **u** as **sure.**

DIJKSTRA BY EXAMPLE

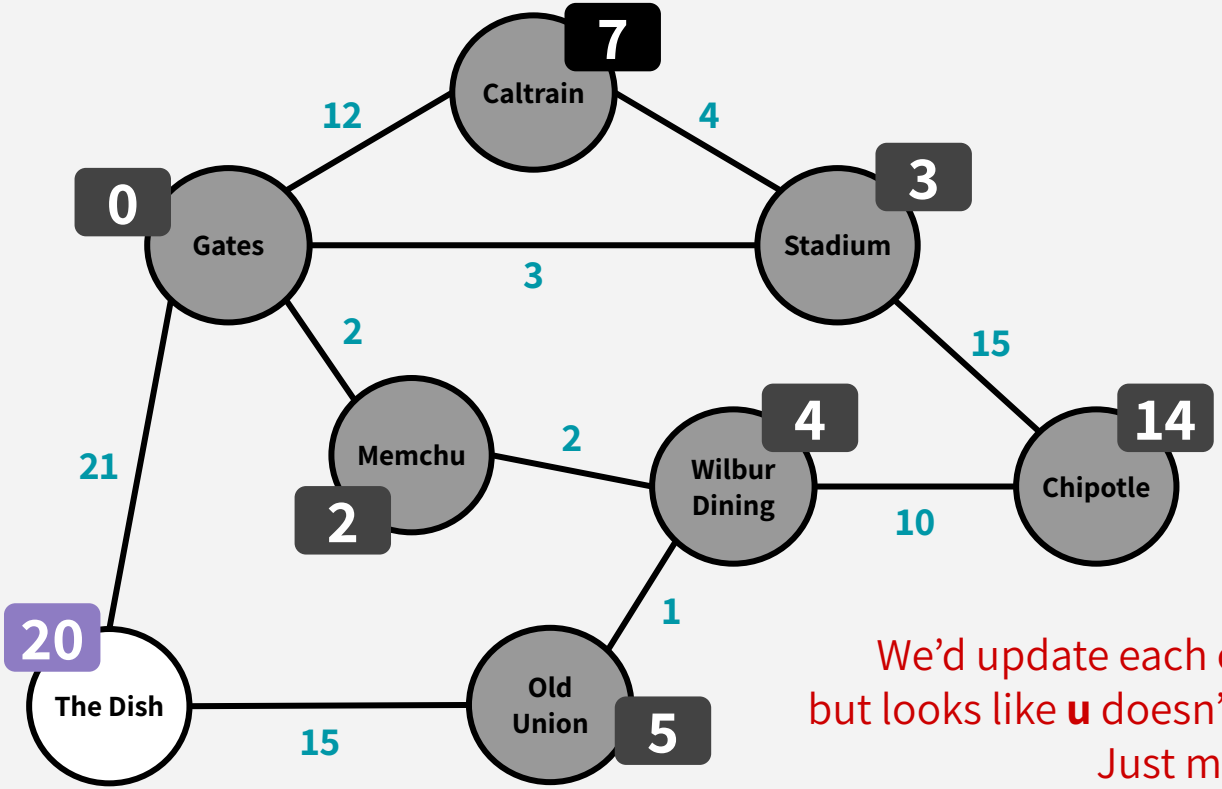




Now pick the next node with the smallest d[v]. That's **Chipotle**.

(we'll call this node **u** now).

DIJKSTRA BY EXAMPLE



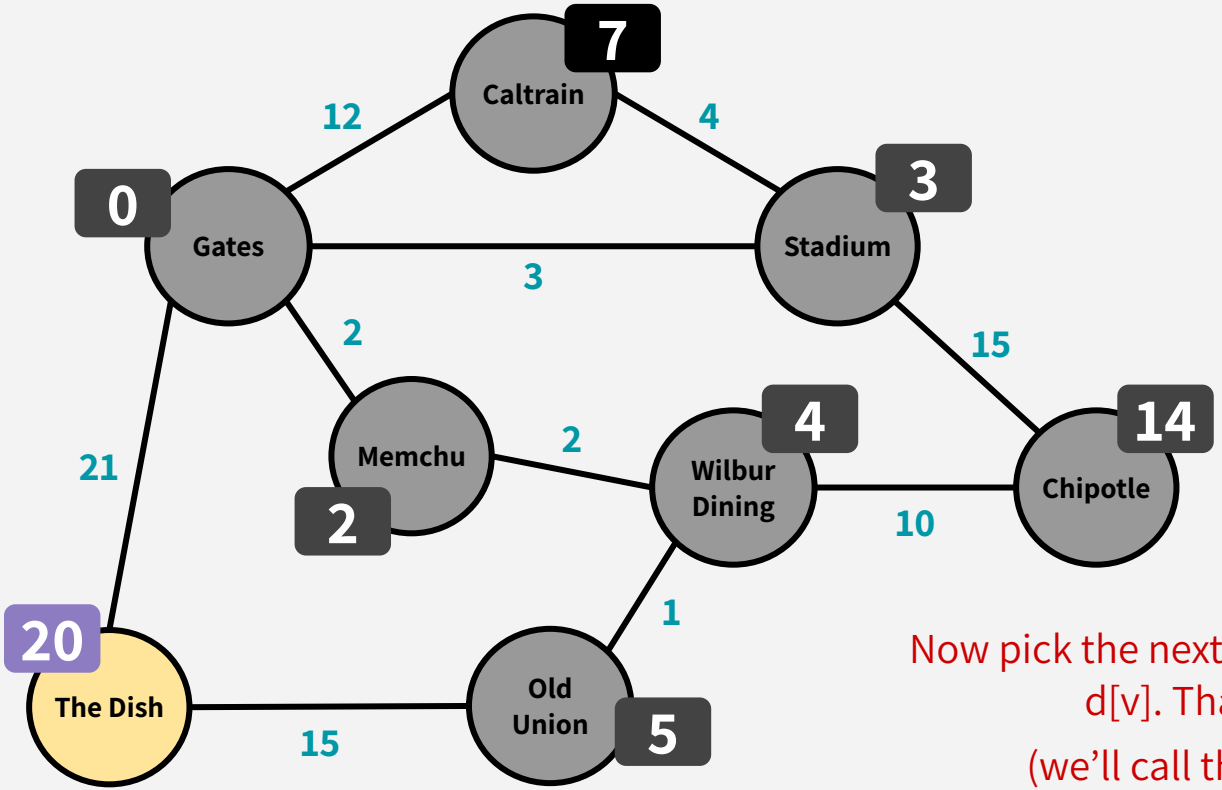
unsure node

current node

sure node

We'd update each of **u**'s *unsure* neighbors **v**,
but looks like **u** doesn't have any unsure neighbors.
Just mark **u** as **sure.**

DIJKSTRA BY EXAMPLE



unsure node

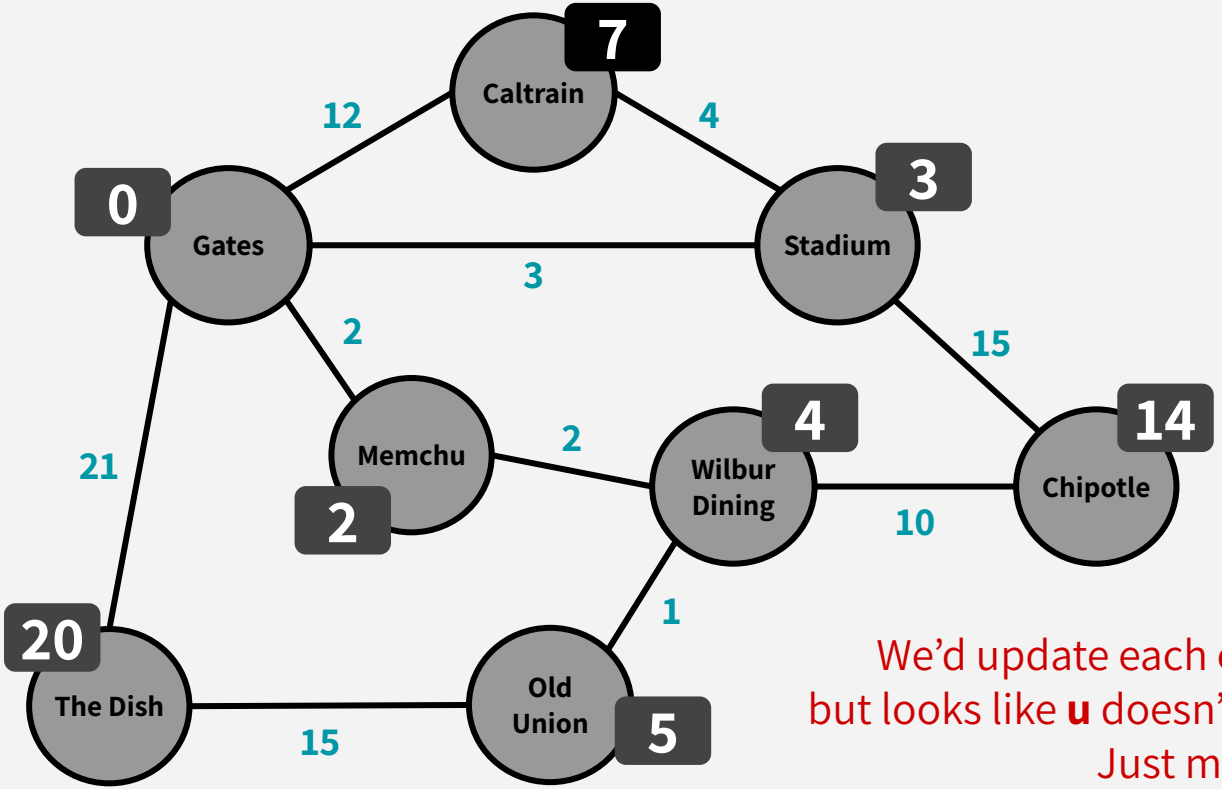
current node

sure node

Now pick the next node with the smallest d[v]. That's **The Dish**.

(we'll call this node **u** now).

DIJKSTRA BY EXAMPLE



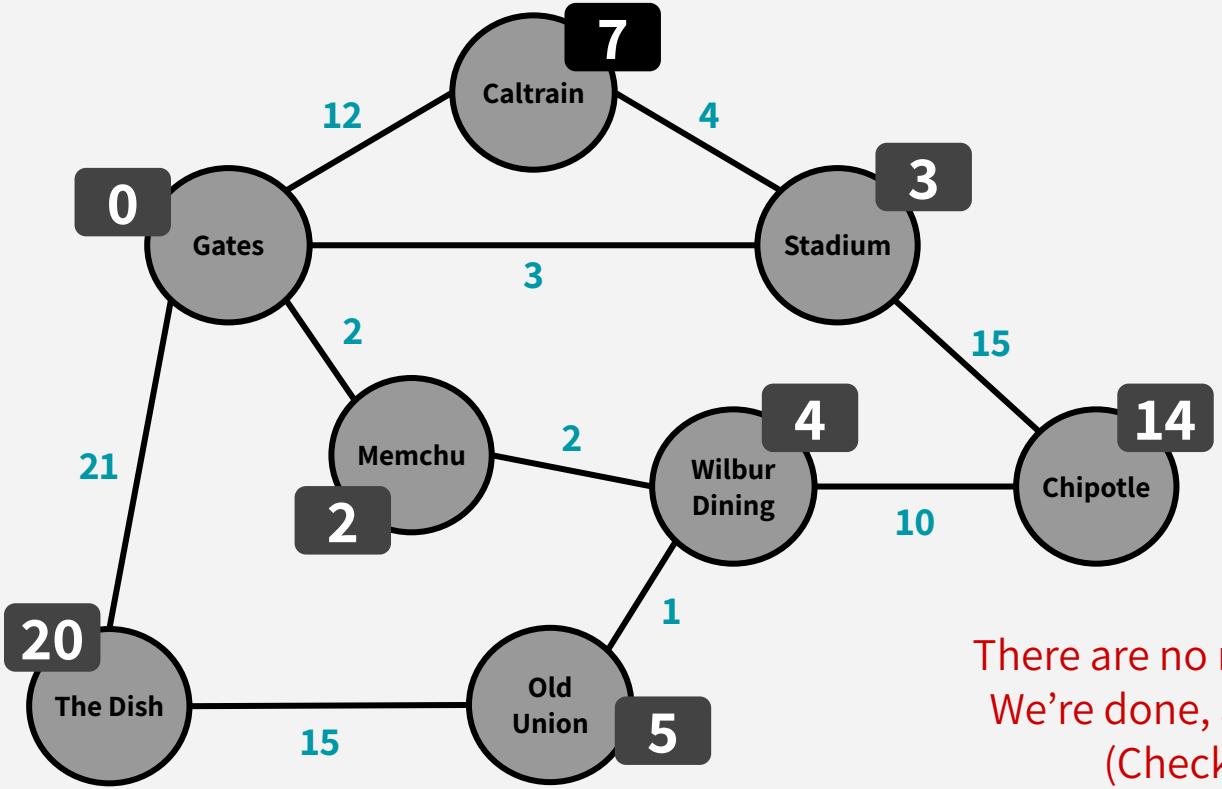
unsure node

current node

sure node

We'd update each of **u**'s *unsure* neighbors **v**,
but looks like **u** doesn't have any unsure neighbors.
Just mark **u** as **sure.**

DIJKSTRA BY EXAMPLE



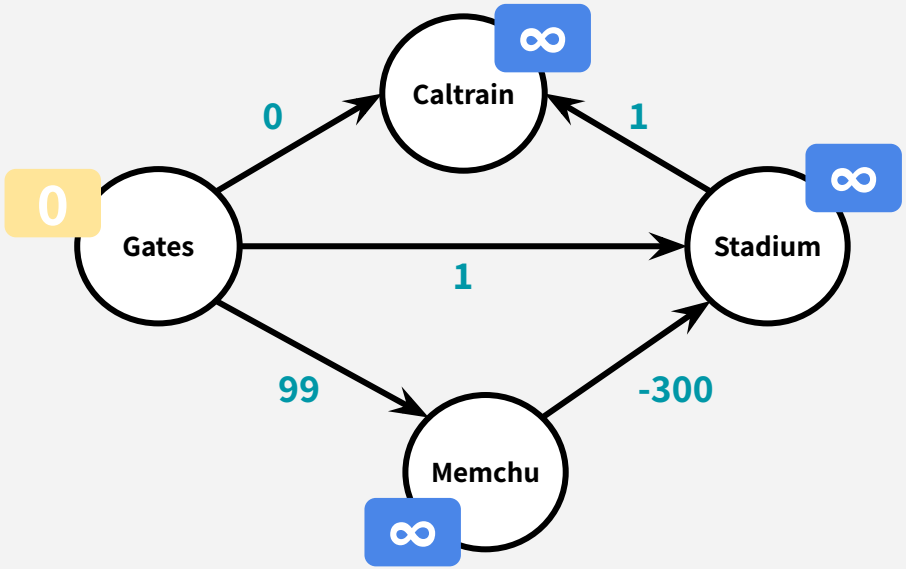
unsure node

current node

sure node

There are no more **unsure** nodes!
We're done, and all d[v] = d(s,v)!
(Check for yourself)

DIJKSTRA BY EXAMPLE



unsure node

current node

sure node

Apply Dijkstra Algorithm

Bellman Ford Algorithm

December 16, 2021

BELLMAN FORD ALGORITHM

- Like Dijkstra's algorithm, it uses the same method of setting current distances, but Ford's method does not permanently determine the shortest distance for any vertex until it processes the entire graph.
- It is more powerful than Dijkstra's method in that it can process graphs with negative weights (but not graphs with negative cycles).
- As required by the original form of the algorithm, all edges are monitored to find a possibility for an improvement of the current distance of vertices so that the algorithm can be presented in this pseudocode.

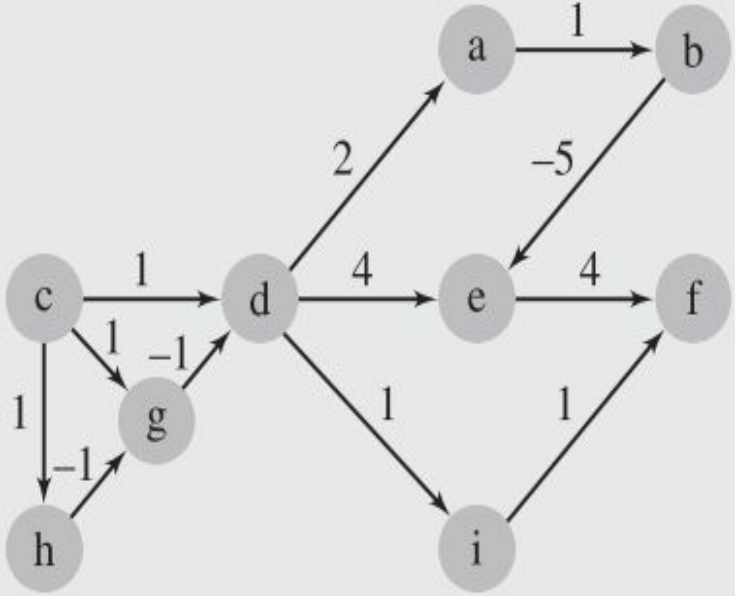
ORDER THE EDGES

- To impose a certain order on monitoring the edges, an alphabetically ordered sequence of edges can be used so that the algorithm can repeatedly go through the entire sequence and adjust the current distance of any vertex, if needed.

FORD ALGORITHM

FordAlgorithm(weighted simple digraph, vertex first)

```
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    while there is an edge(vu) such that currDist(u) > currDist(v)+ weight(edge(vu))
        currDist(u) = currDist(v)+ weight(edge(vu));
```



the order of edges: *ab be cd cg ch da de di ef gd hg if*

		iteration				
	init	**1**		**2**	**3**	**4**
a	∞	3		2	1	
b	∞			4	3	2
c	0					
d	∞	1	0	−1		
e	∞	5		−1	−2	−3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2		1	0	

Floyd-Warshall Algorithm

December 16, 2021

ORDER THE EDGES

- Although the task of finding all shortest paths from any vertex to any other vertex seems to be more complicated than the task of dealing with single source only.
- Floyd-Warshall Algorithm does it in surprisingly simple way
    - Stephen Warshall and implemented by Robert W. Floyd and P. Z. Ingerman

FORD ALGORITHM

```
WFIalgorithm(matrix weight)
        for i = 1 to |V|
                for j = 1 to |V|
                        for k = 1 to |V|
                                if weight[j][k] > weight[j][i] + weight[i][k]
                                        weight[j][k] = weight[j][i] + weight[i][k];
```

// The outermost loop refers to vertices that may be on a path between the vertex with index j and the vertex with index k.