

# Lecture 11

Weighted Graphs: Dijkstra and Bellman-Ford

# Piazza Heroes

Name, Email	number of answers
Jabari Hastings	76
Ashish Paliwal	45
Michael Cooper	34
Adam Leon	26
Jiao Li	19
Logan Pearce	19
Pranav Jain	18
Xinlan Emily Hu	18
Trenton Chang	14
Xieyuan Zhang	14

# Midterm Wednesday 2/20!

- Organization will probably be something like:
  - 1. Multiple choice
  - 2. Short answer
  - 3. Algorithm analysis
  - 4. Algorithm design
  - 5. Challenge problem (not worth very many points)
- Promise: one of the questions will be very close to a homework question.
- This may be a hard exam.
  - **If it is, that means it's okay if you don't get all the questions.**
  - (Please don't freak out).

# Midterm Logistics

- Last names A-R: NVIDIA Auditorium (here)
- Last names S-Z: Oshman
  - McMurtry Art & Art History Building, Oshman Presentation Space, Room 102
- (Unless you have already arranged an alternate midterm).
- We will start PROMPTLY at 10:30, please be on time.
- We could not find enough space during this timeslot to seat everyone every-other seat.
  - Please be mindful of the honor code.

# More announcements

- HW4 due Friday.
- No new HW released Friday because of the midterm next week.
- Have a good long weekend!
  - Monday is President's day

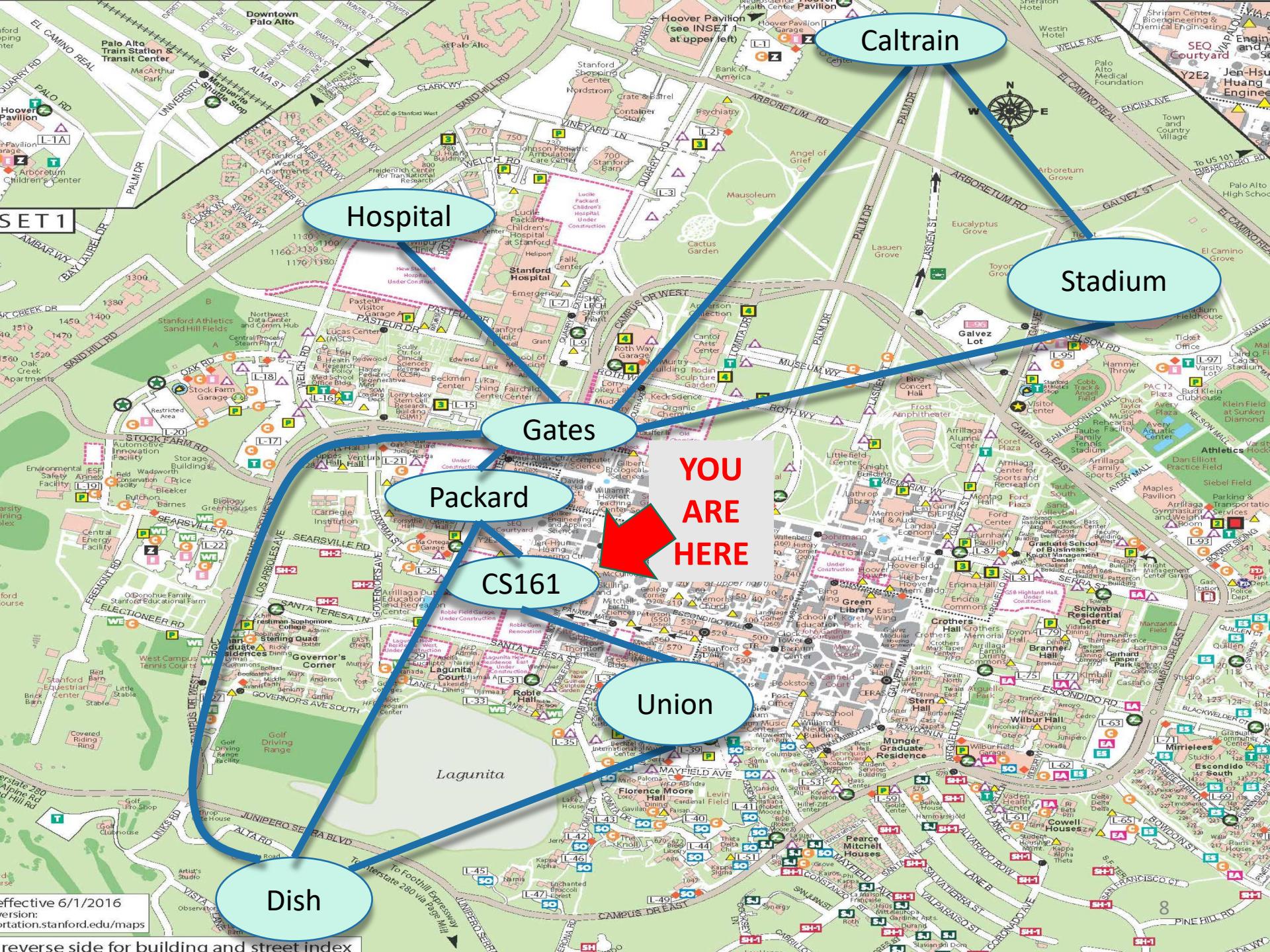
# Previous two lectures

- Graphs!
- DFS
  - Topological Sorting
  - Strongly Connected Components
- BFS
  - Shortest Paths in unweighted graphs

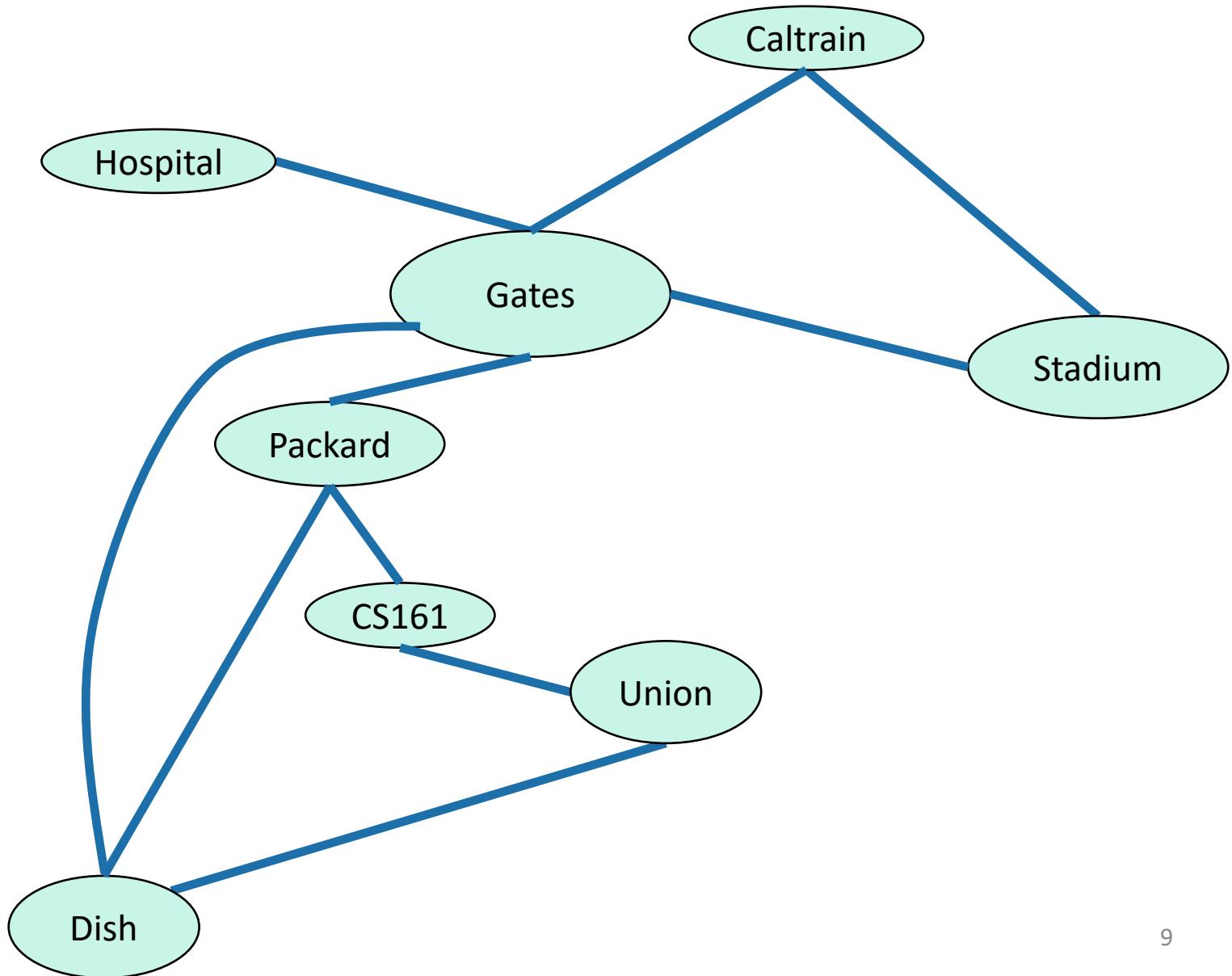
# Today

- What if the graphs are weighted?
- Part 1: Dijkstra!
  - This will take most of today's class
- Part 2: Bellman-Ford!
  - Real quick at the end!
  - We'll come back to Bellman-Ford in more detail, so today is just a taste.

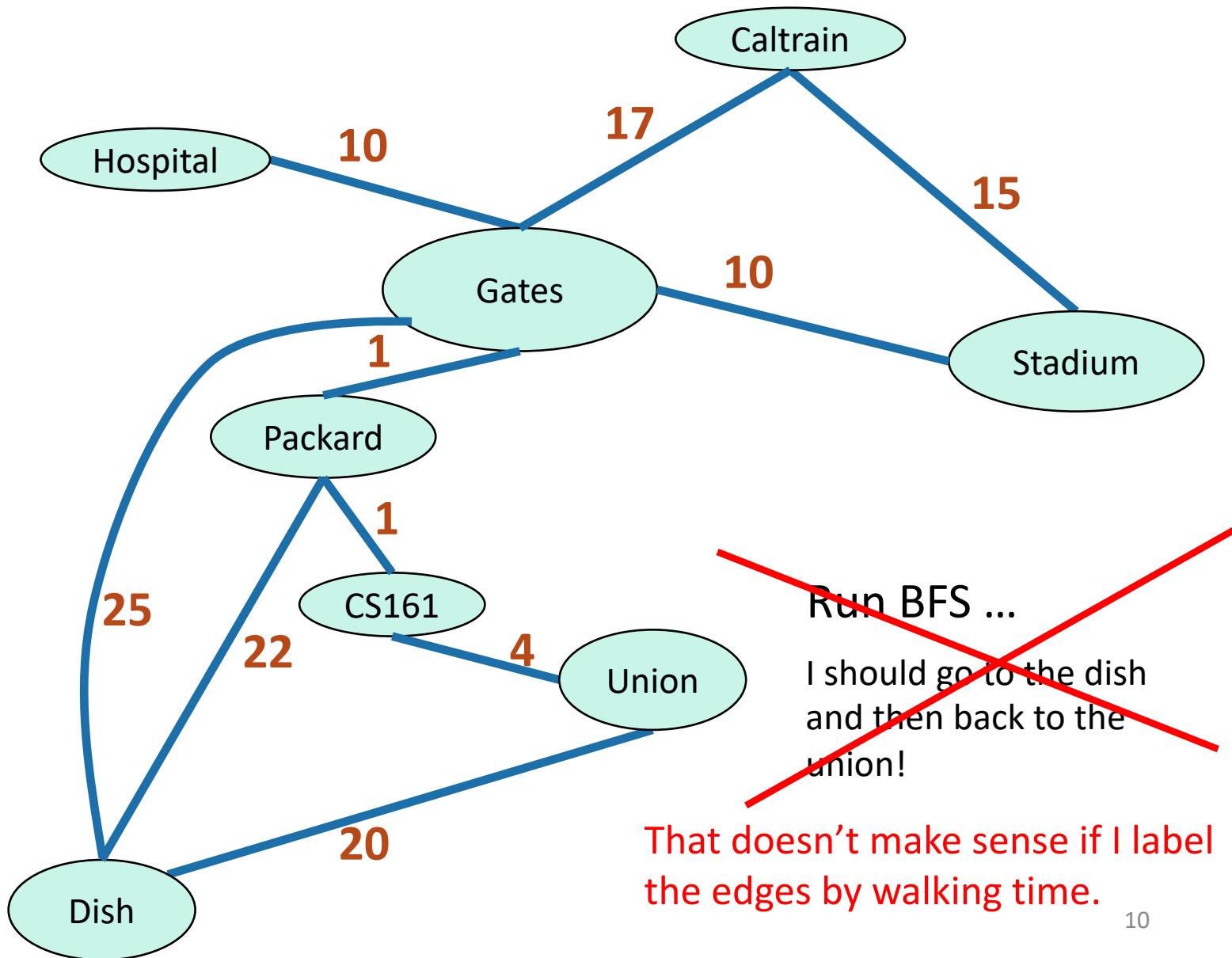




# Just the graph



# Shortest path from Gates to the Union?

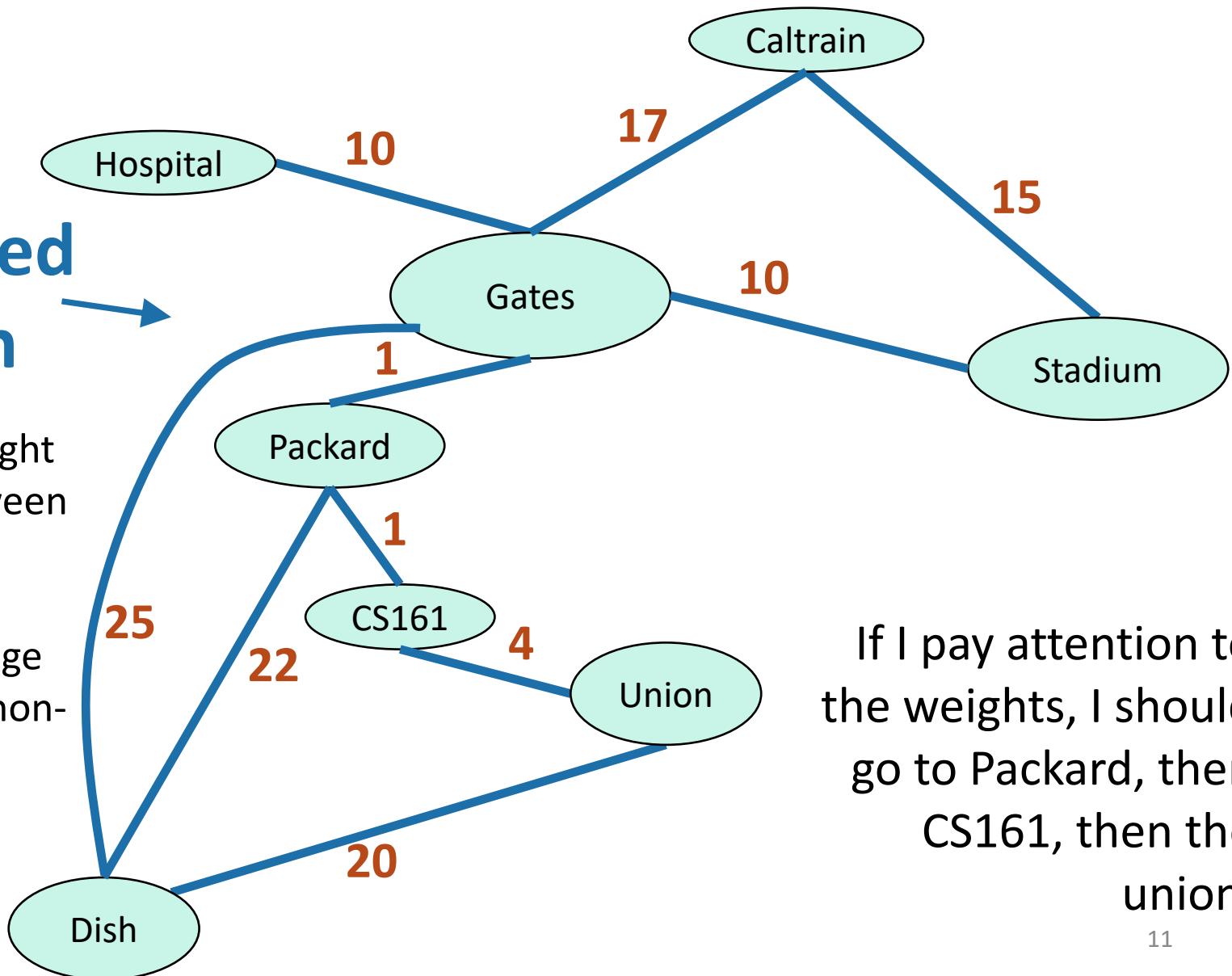


# Shortest path from Gates to the Union?

**weighted  
graph**

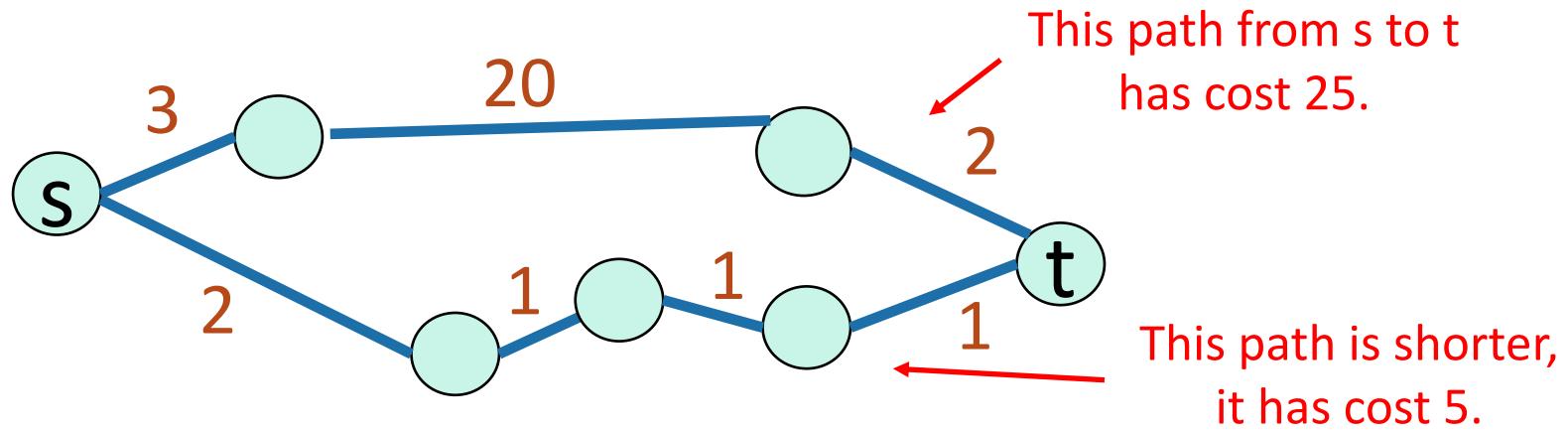
$w(u,v)$  = weight of edge between  $u$  and  $v$ .

For now, edge weights are non-negative.

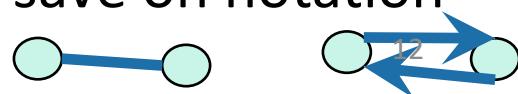


# Shortest path problem

- 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.



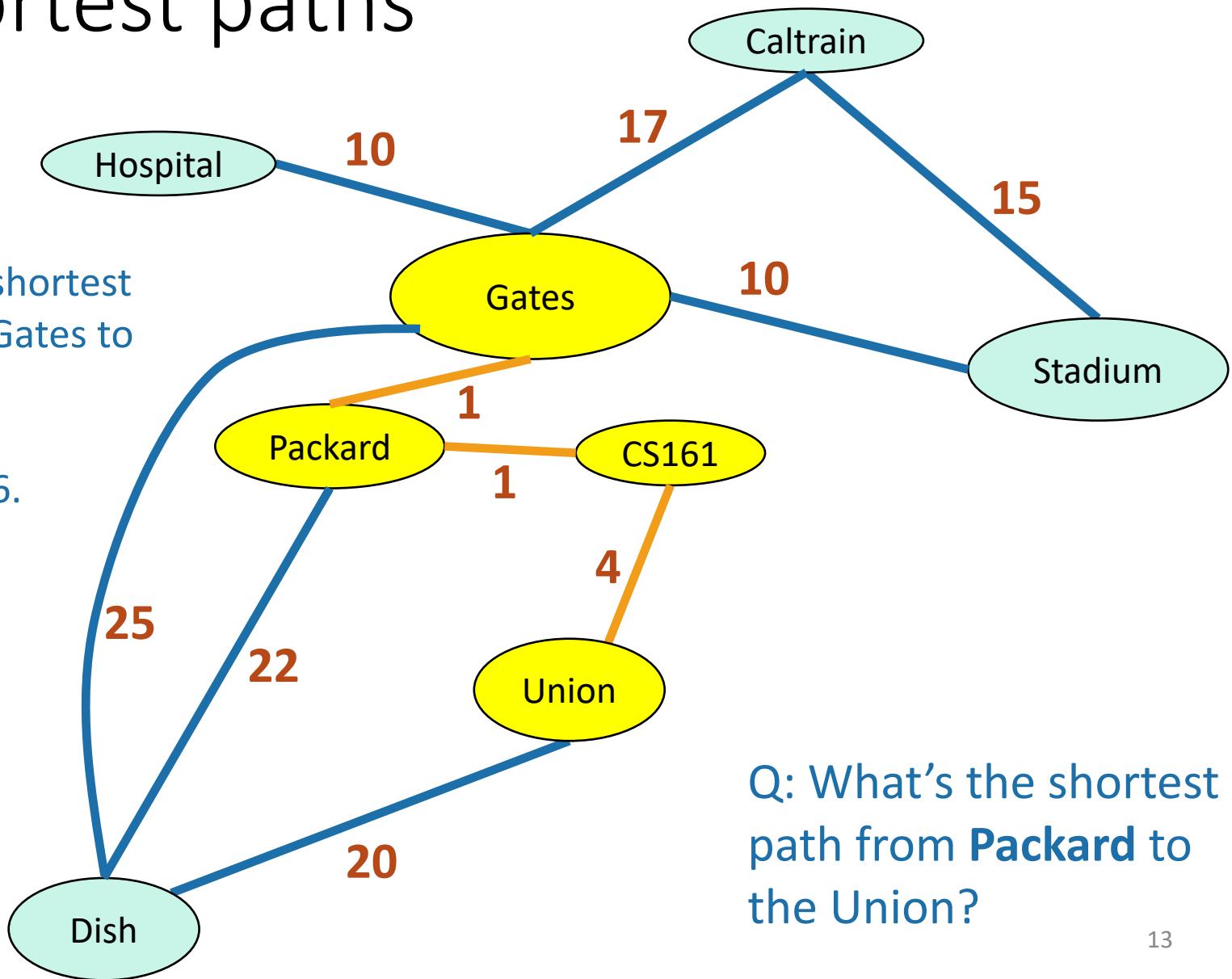
- The **distance**  $d(u,v)$  between two vertices  $u$  and  $v$  is the cost of the the shortest path between  $u$  and  $v$ .
- For this lecture **all graphs are directed**, but to save on notation I'm just going to draw undirected edges.



# Shortest paths

This is the shortest path from Gates to the Union.

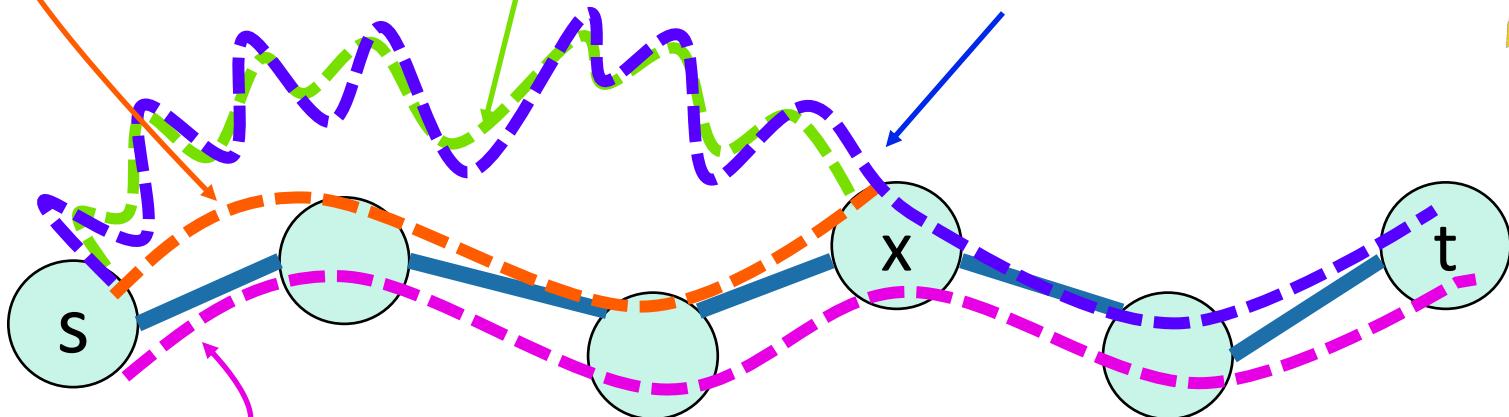
It has cost 6.



# Warm-up

- A sub-path of a shortest path is also a shortest path.

- Say **this** is a shortest path from  $s$  to  $t$ .
- Claim: **this** is a shortest path from  $s$  to  $x$ .
  - Suppose not, **this** one is a shorter path from  $s$  to  $x$ .
  - But then that gives an **even shorter path** from  $s$  to  $t$ !



# Single-source shortest-path problem

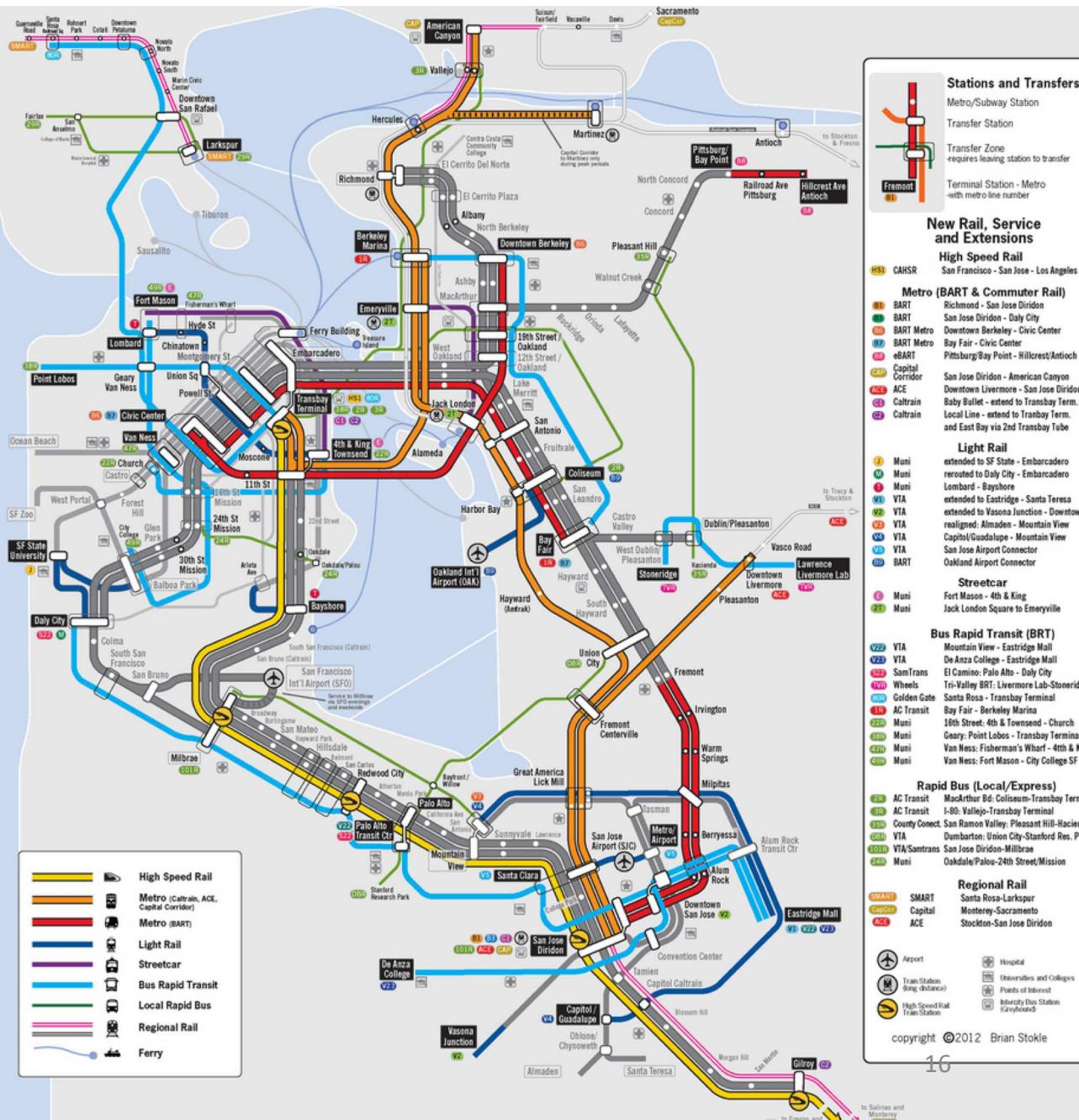
- I want to know the shortest path from one vertex (Gates) to all other vertices.

Destination	Cost	To get there
Packard	1	Packard
CS161	2	Packard-CS161
Hospital	10	Hospital
Caltrain	17	Caltrain
Union	6	Packard-CS161-Union
Stadium	10	Stadium
Dish	23	Packard-Dish

(Not necessarily stored as a table – how this information  
is represented will depend on the application)<sup>15</sup>

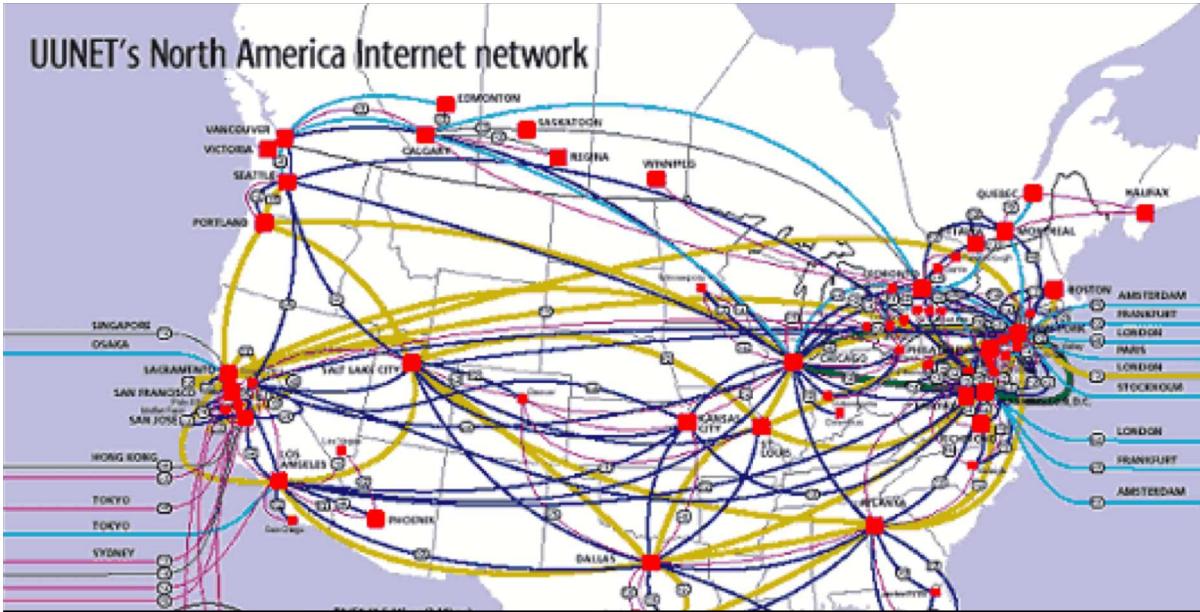
# Example

- “what is the shortest path from Palo Alto to [anywhere else]” using BART, Caltrain, lightrail, MUNI, bus, Amtrak, bike, walking, uber/lyft.
- Edge weights have something to do with time, money, hassle.

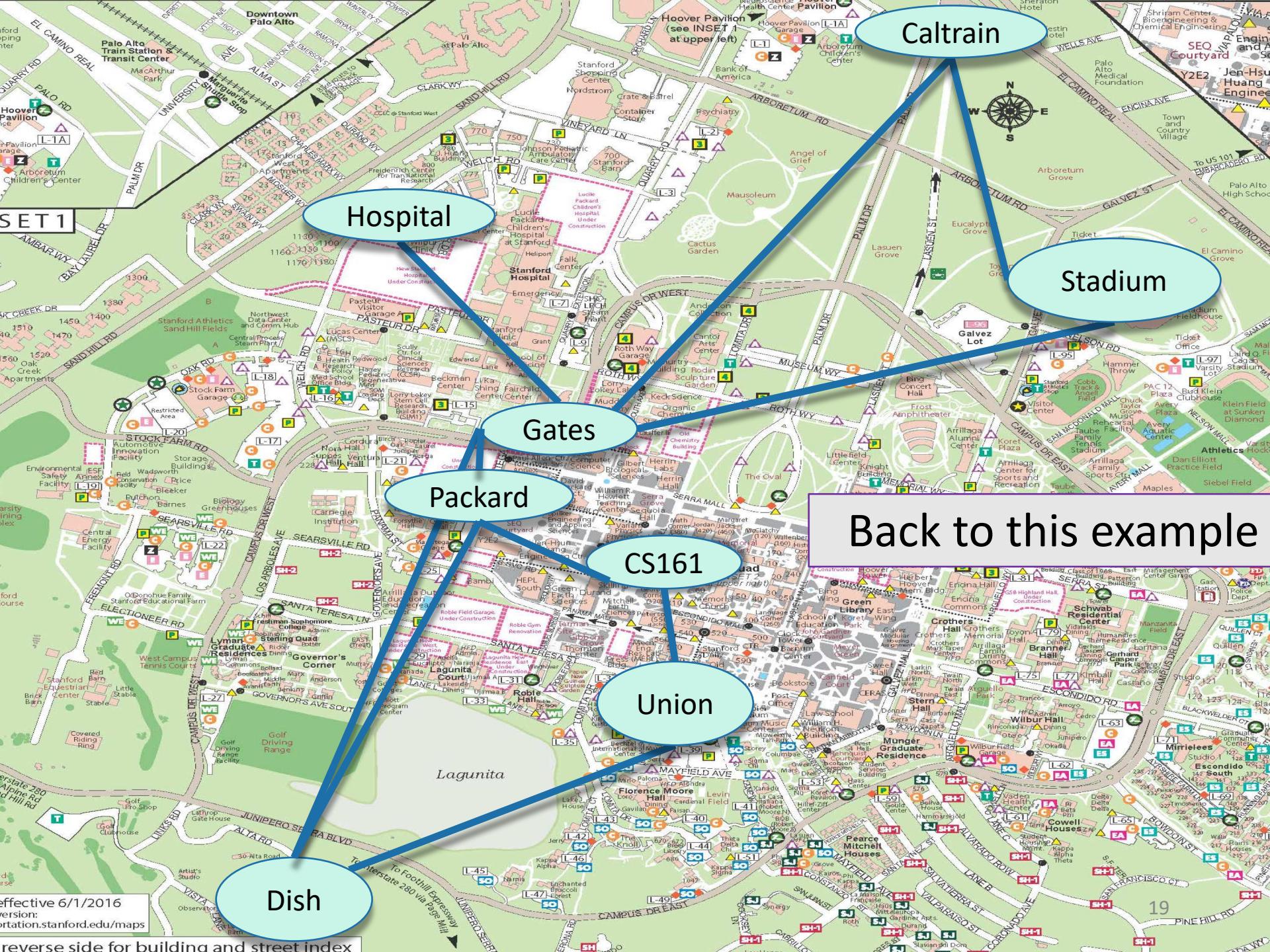


# Example

- **Network routing**
- I send information over the internet, from my computer to all over the world.
- Each path has a cost which depends on link length, traffic, other costs, etc..
- How should we send packets?

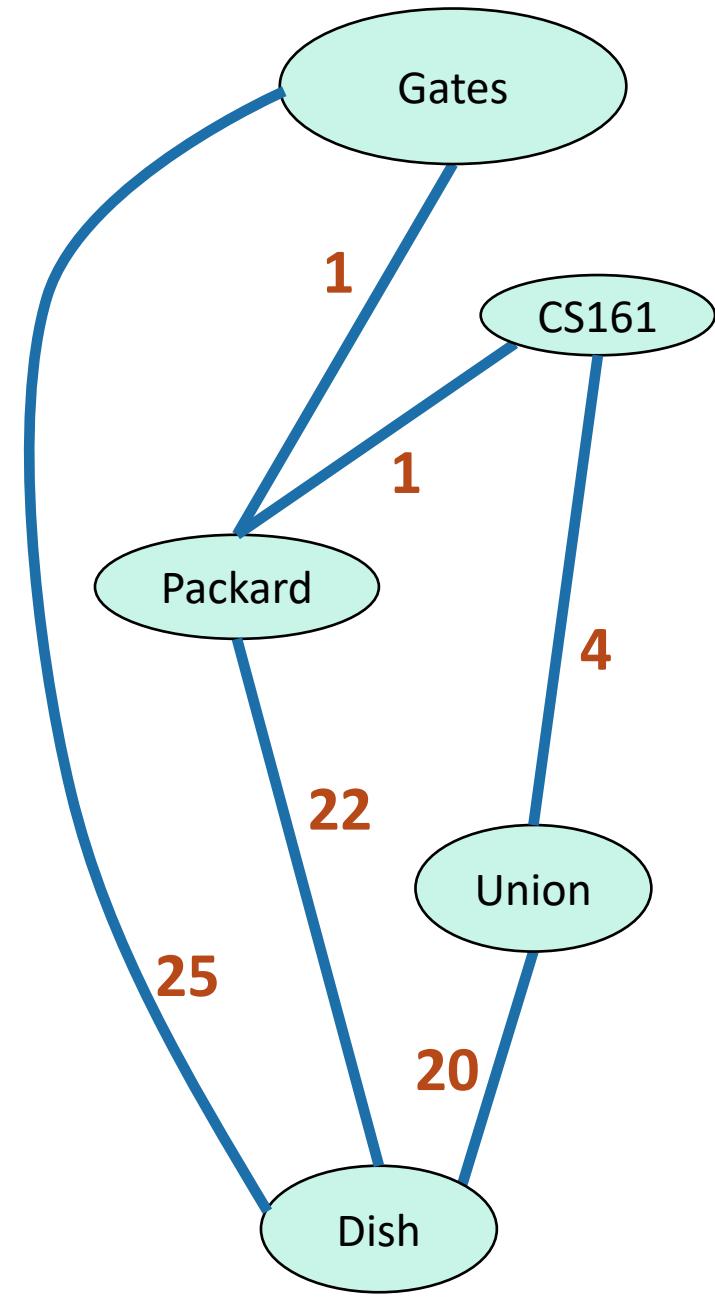


```
[DN0a22a0e3:~ mary$ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1 [AS0] 10.34.160.2 (10.34.160.2) 38.168 ms 31.272 ms 28.841 ms
 2 [AS0] cwa-vrtr.sunet (10.21.196.28) 33.769 ms 28.245 ms 24.373 ms
 3 [AS32] 171.66.2.229 (171.66.2.229) 24.468 ms 20.115 ms 23.223 ms
 4 [AS32] hpr-svl-rtr-vlan8.sunet (171.64.255.235) 24.644 ms 24.962 ms 11.111 ms
 5 [AS2152] hpr-svl-hpr2--stan-ge.cenic.net (137.164.27.161) 22.129 ms 4.916 ms
 6 [AS2152] hpr-lax-hpr3--svl-hpr3-100ge.cenic.net (137.164.25.73) 12.125 ms 1.111 ms
 7 [AS2152] hpr-i2--lax-hpr2-r&e.cenic.net (137.164.26.201) 40.174 ms 38.333 ms
 8 [AS0] et-4-0-0.4079.sdn-sw.lasv.net.internet2.edu (162.252.70.28) 46.573 ms
 9 [AS0] et-5-1-0.4079.rtsw.salt.net.internet2.edu (162.252.70.31) 30.424 ms
10 [AS0] et-4-0-0.4079.sdn-sw.denv.net.internet2.edu (162.252.70.8) 47.454 ms
11 [AS0] et-4-1-0.4079.rtsw.kans.net.internet2.edu (162.252.70.11) 70.825 ms
12 [AS0] et-4-1-0.4070.rtsw.chic.net.internet2.edu (198.71.47.206) 77.937 ms
13 [AS0] et-0-1-0.4079.sdn-sw.ashb.net.internet2.edu (162.252.70.60) 77.682 ms
14 [AS0] et-4-1-0.4079.rtsw.wash.net.internet2.edu (162.252.70.65) 71.565 ms
15 [AS21320] internet2-gw.mx1.lon.uk.geant.net (62.40.124.44) 154.926 ms 1.111 ms
16 [AS21320] ae0.mx1.lon2.uk.geant.net (62.40.98.79) 146.565 ms 146.604 ms
17 [AS21320] ae0.mx1.par.fr.geant.net (62.40.98.77) 153.289 ms 184.995 ms
18 [AS21320] ae2.mx1.gen.ch.geant.net (62.40.98.153) 160.283 ms 160.104 ms
19 [AS21320] swice1-100ge-0-3-0-1.switch.ch (62.40.124.22) 162.068 ms 160.104 ms
20 [AS559] swizh1-100ge-0-1-0-1.switch.ch (130.59.36.94) 165.824 ms 164.222 ms
21 [AS559] swiez3-100ge-0-1-0-4.switch.ch (130.59.38.109) 164.269 ms 164.222 ms
22 [AS559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1) 164.082 ms 164.222 ms
23 [AS559] rou-fw-rz-rz-gw.ethz.ch (192.33.92.169) 164.773 ms 165.193 ms
```

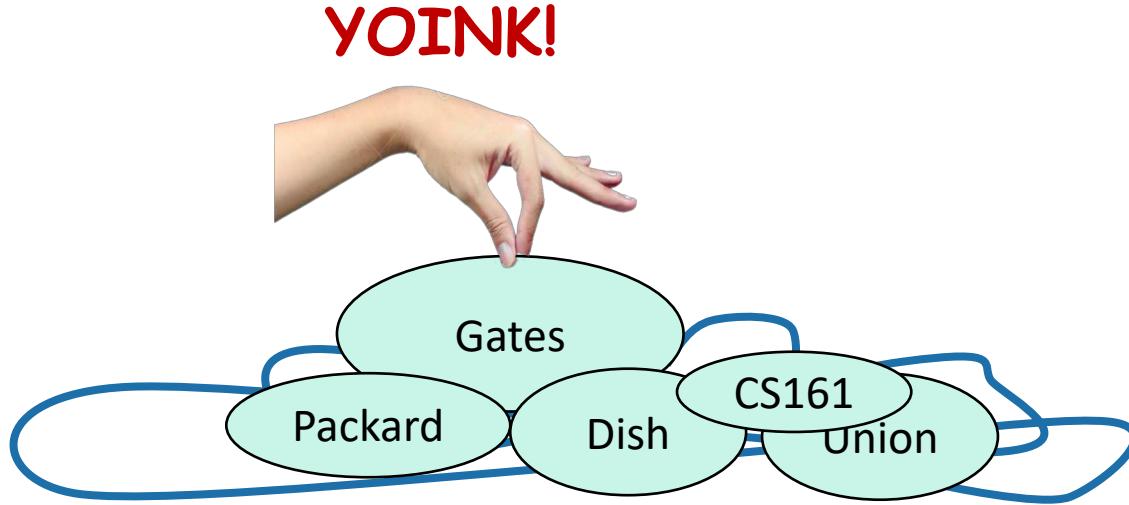


# Dijkstra's algorithm

- Finds shortest paths from Gates to everywhere else.



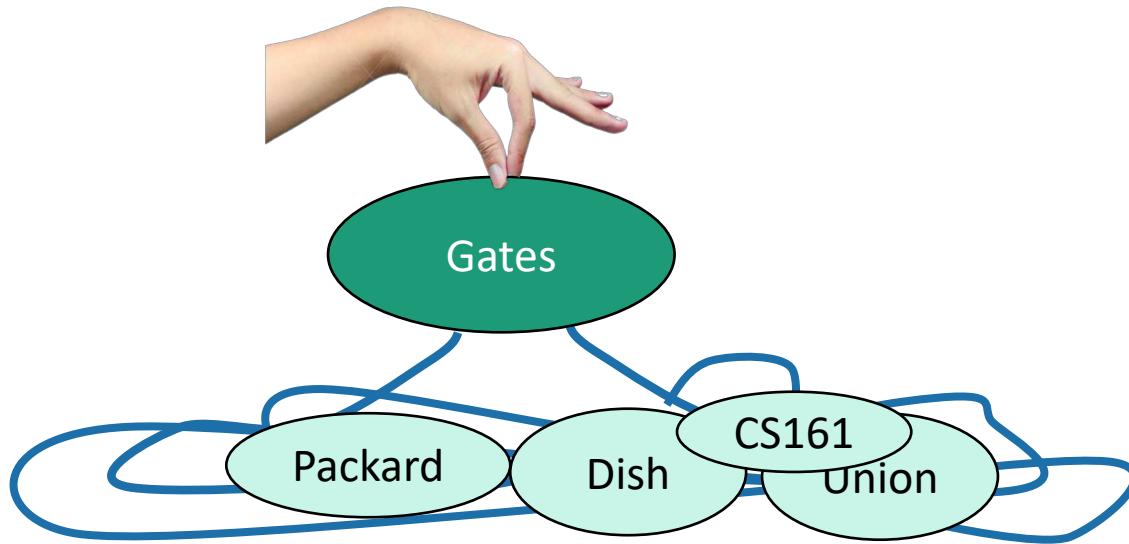
# Dijkstra intuition



# Dijkstra intuition

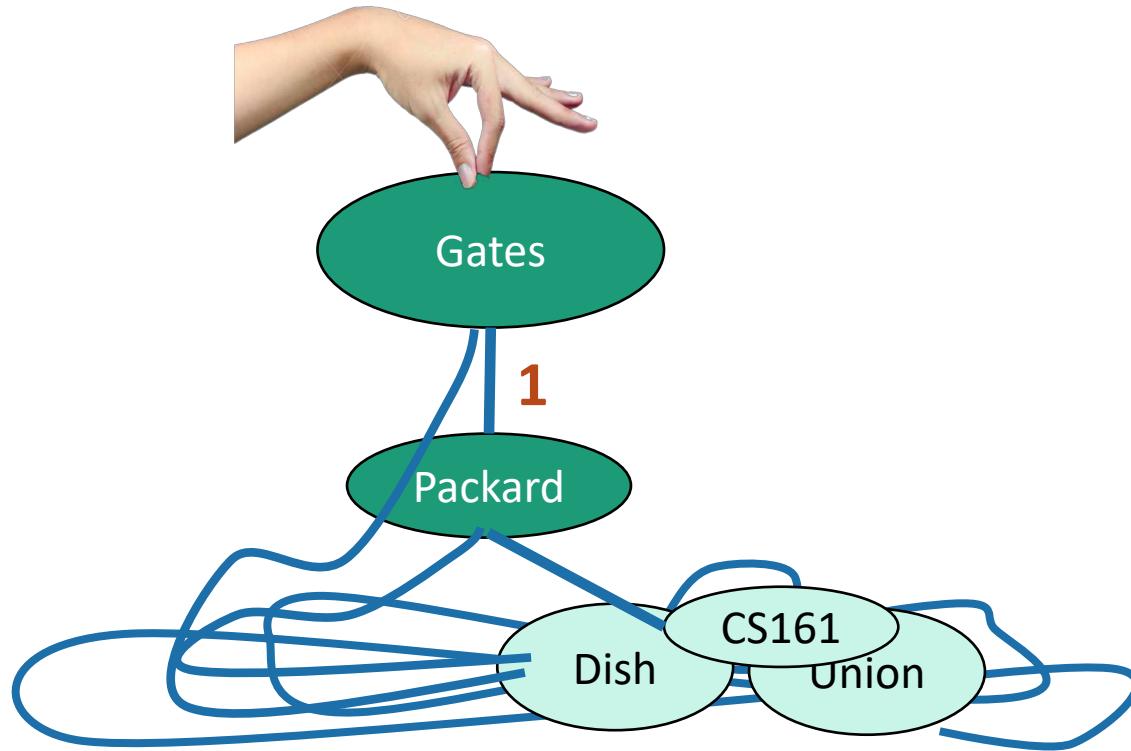
A vertex is done when it's not on the ground anymore.

YOINK!



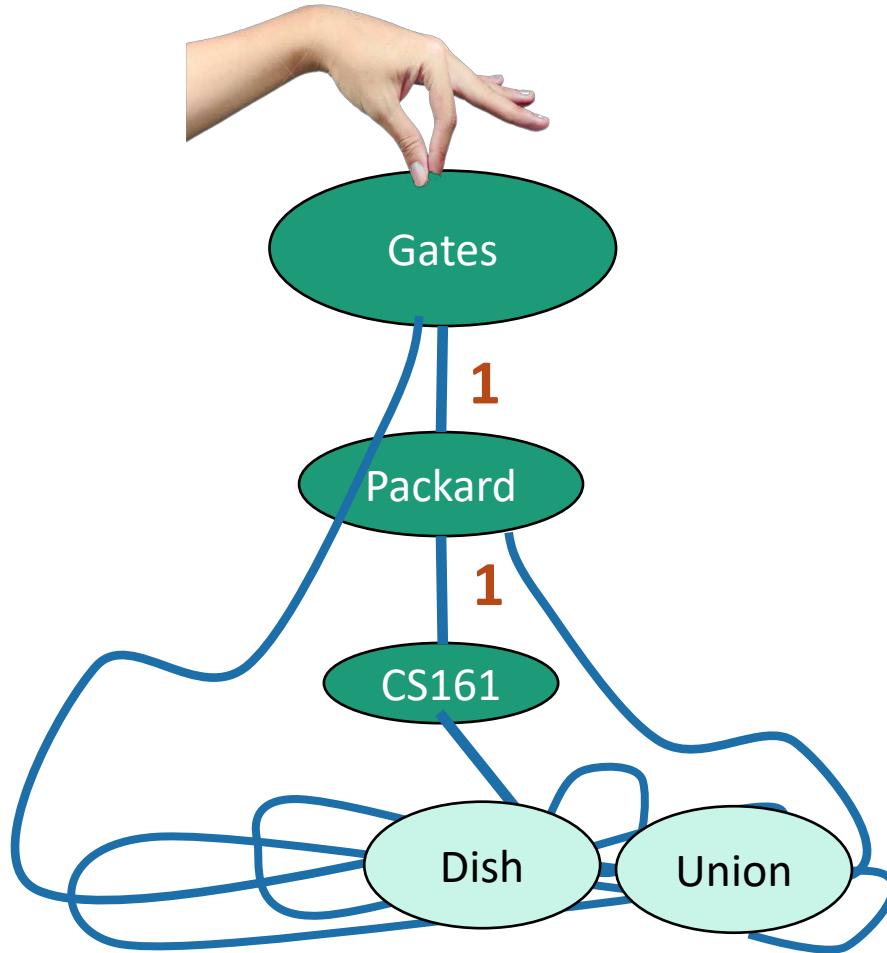
# Dijkstra intuition

YOINK!



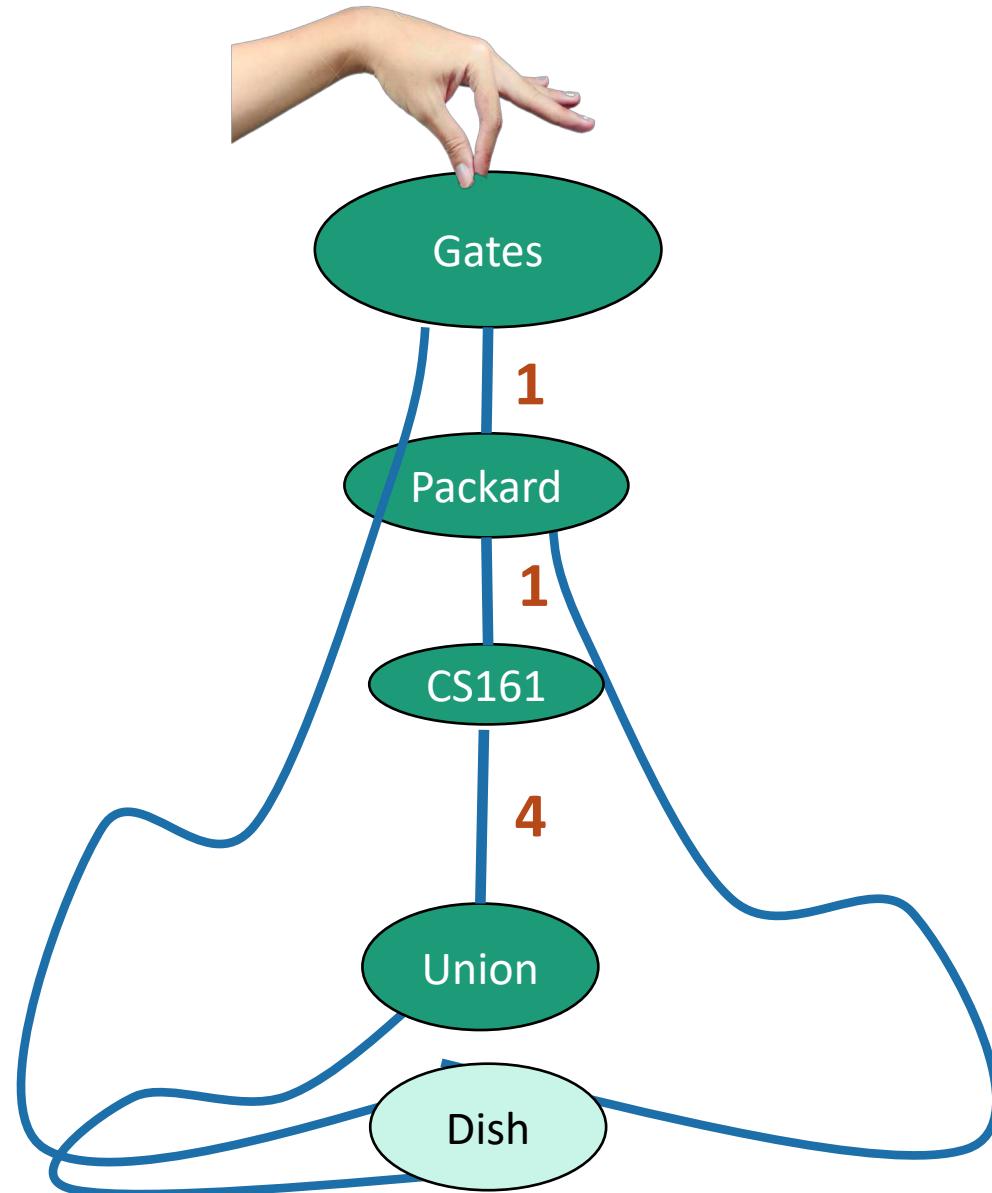
# Dijkstra intuition

YOINK!

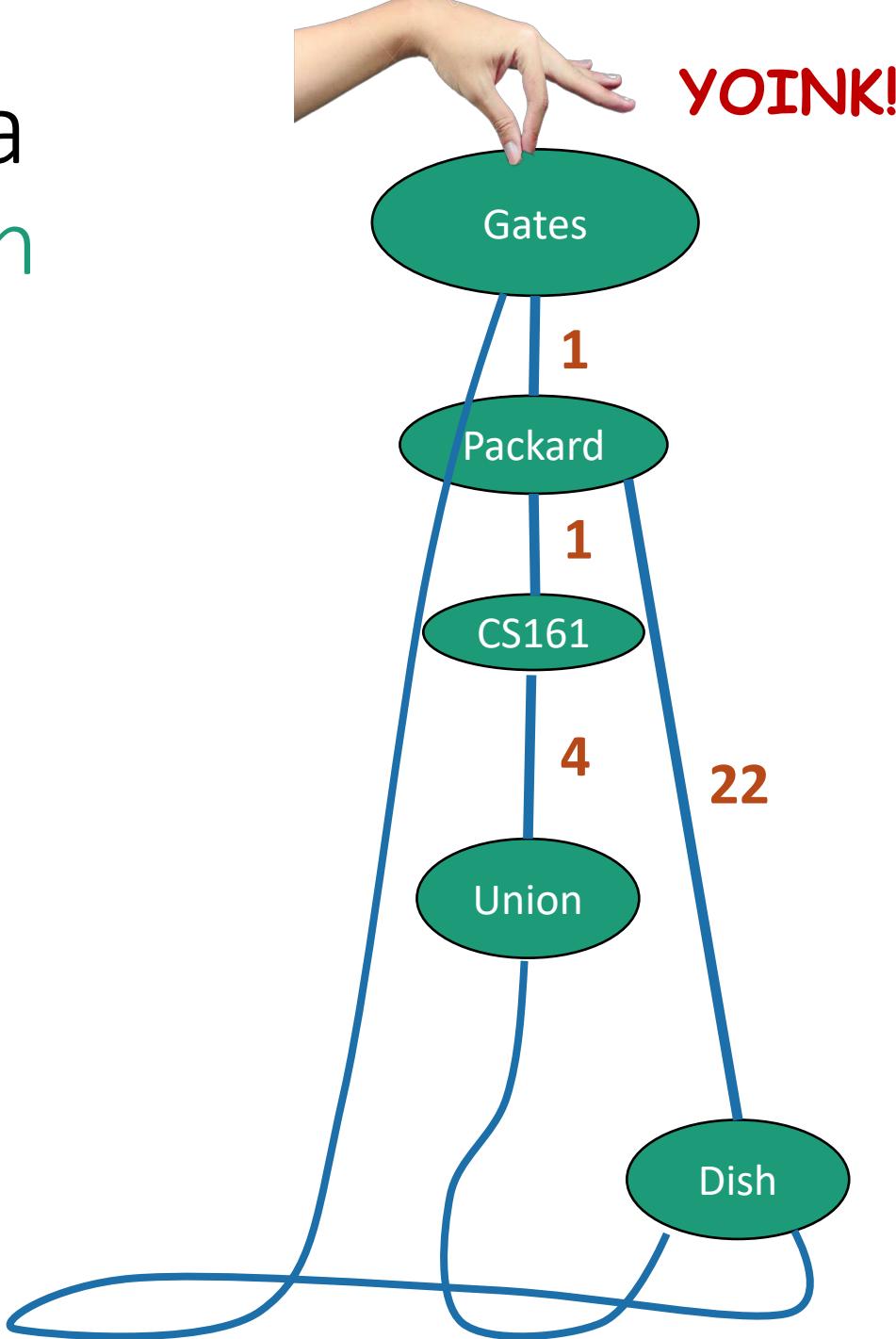


# Dijkstra intuition

YOINK!



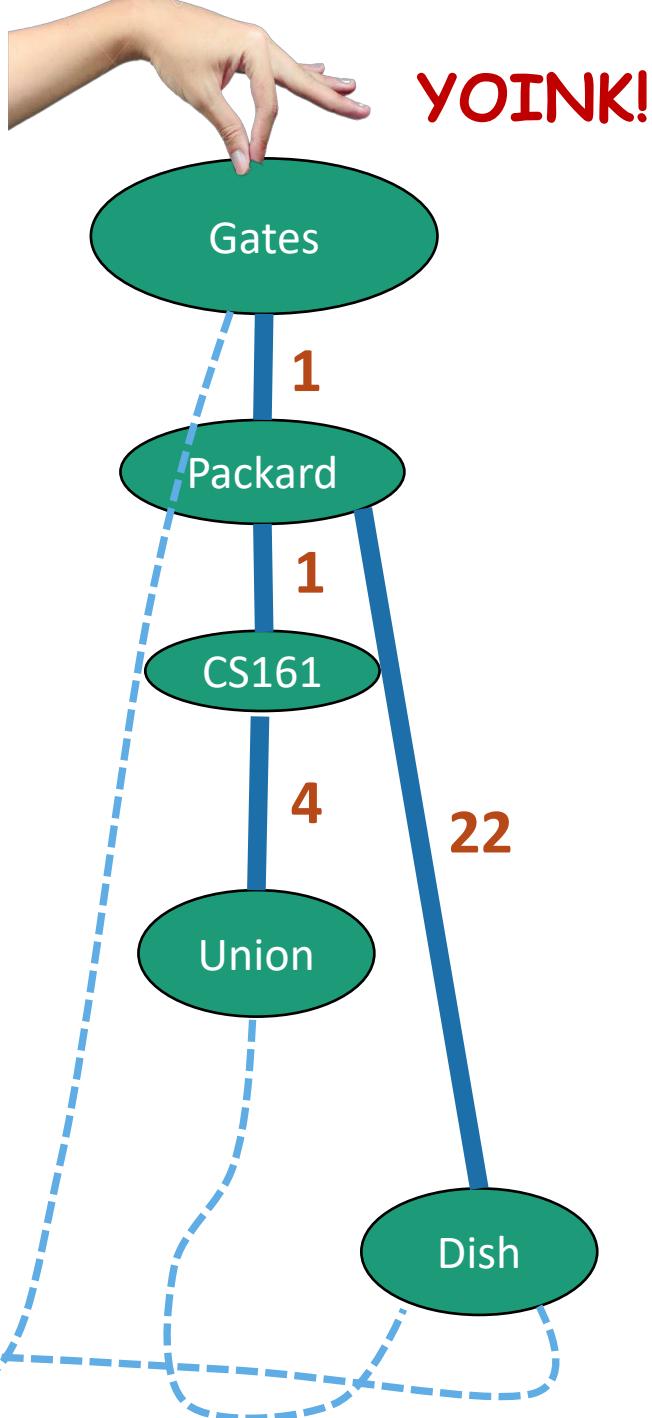
# Dijkstra intuition



# Dijkstra intuition

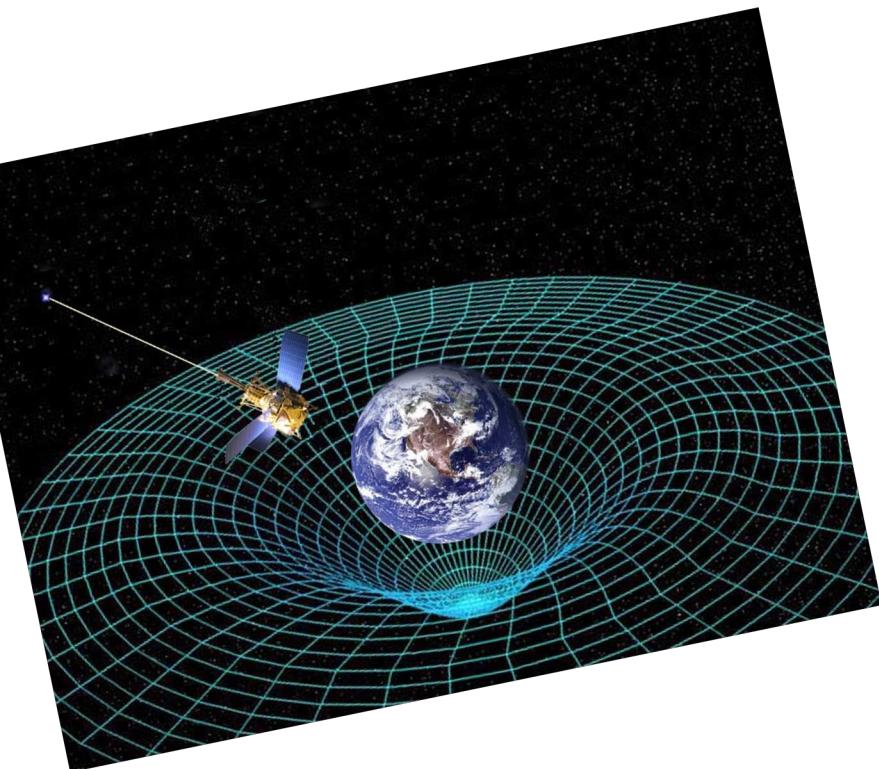
This creates a tree!

The shortest paths  
are the lengths  
along this tree.



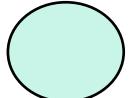
# How do we actually implement this?

- **Without** string and gravity?

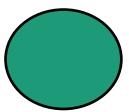


# Dijkstra by example

How far is a node from Gates?



I'm not sure yet



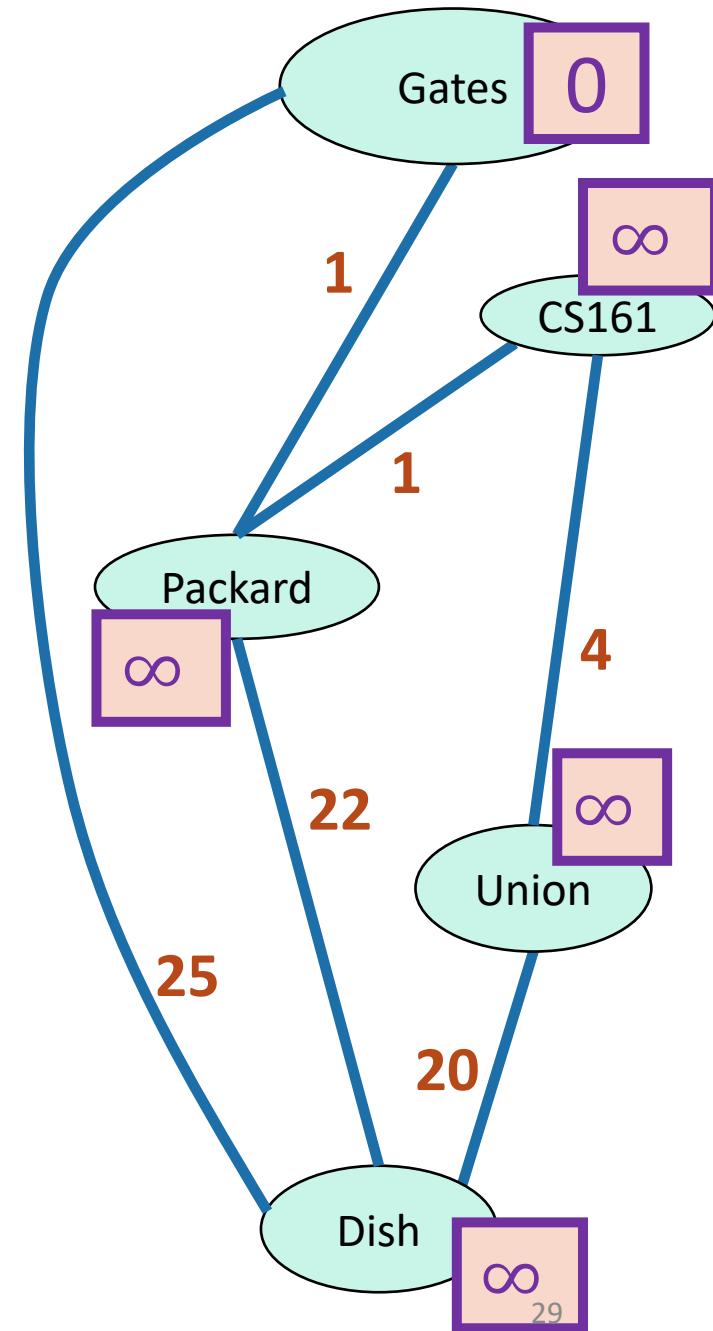
I'm sure



$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .

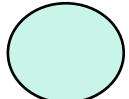
Initialize  $d[v] = \infty$  for all non-starting vertices  $v$ , and  $d[\text{Gates}] = 0$

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .

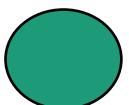


# Dijkstra by example

How far is a node from Gates?



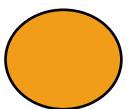
I'm not sure yet



I'm sure

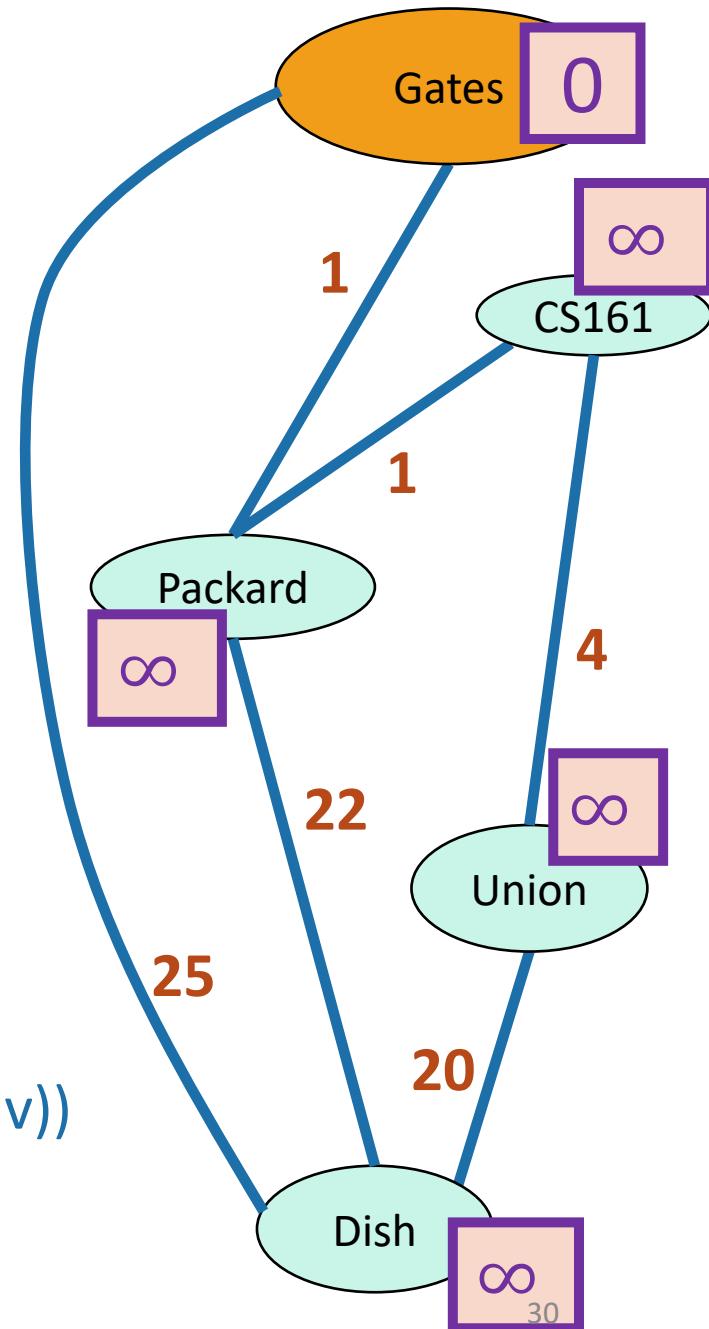


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



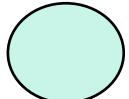
Current node u

- Pick the **not-sure** node u with the smallest estimate  $d[u]$ .
- Update all u's neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$

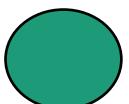


# Dijkstra by example

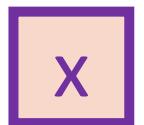
How far is a node from Gates?



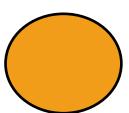
I'm not sure yet



I'm sure

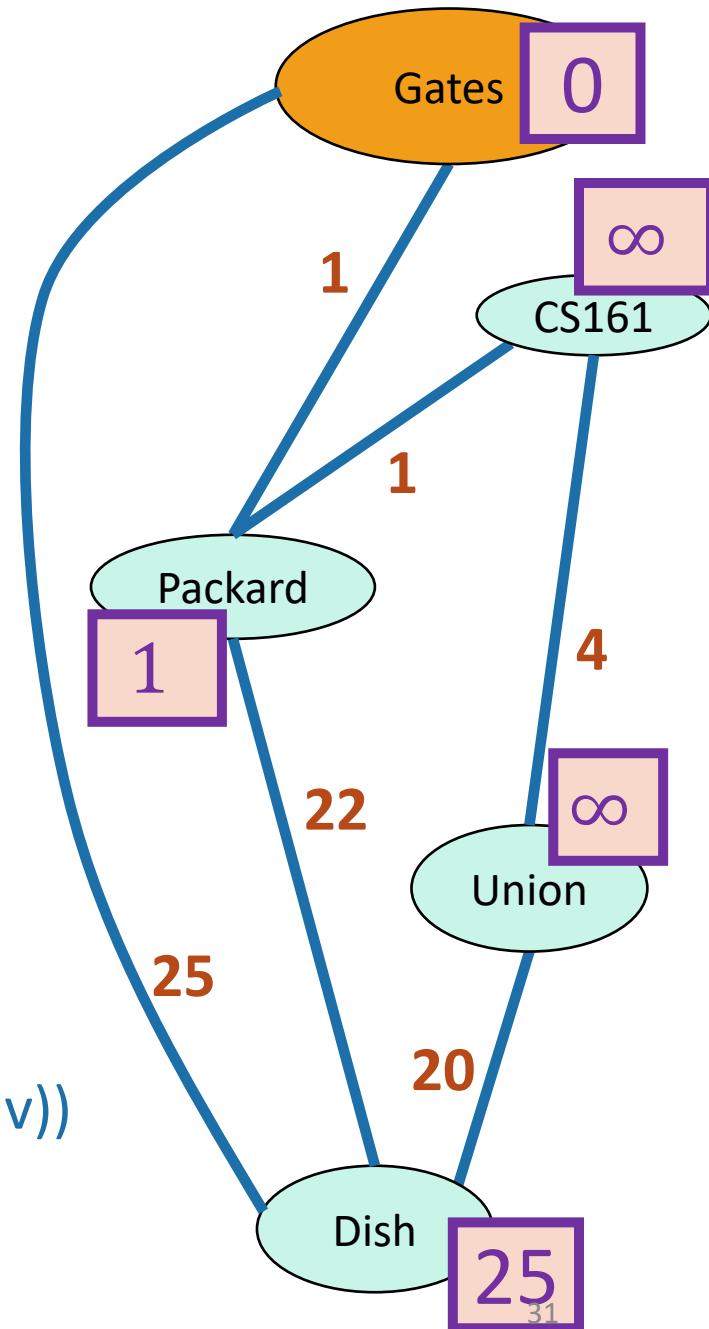


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



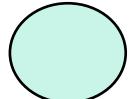
Current node u

- Pick the **not-sure** node u with the smallest estimate  $d[u]$ .
- Update all u's neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **Sure**.

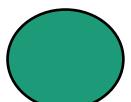


# Dijkstra by example

How far is a node from Gates?



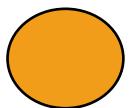
I'm not sure yet



I'm sure

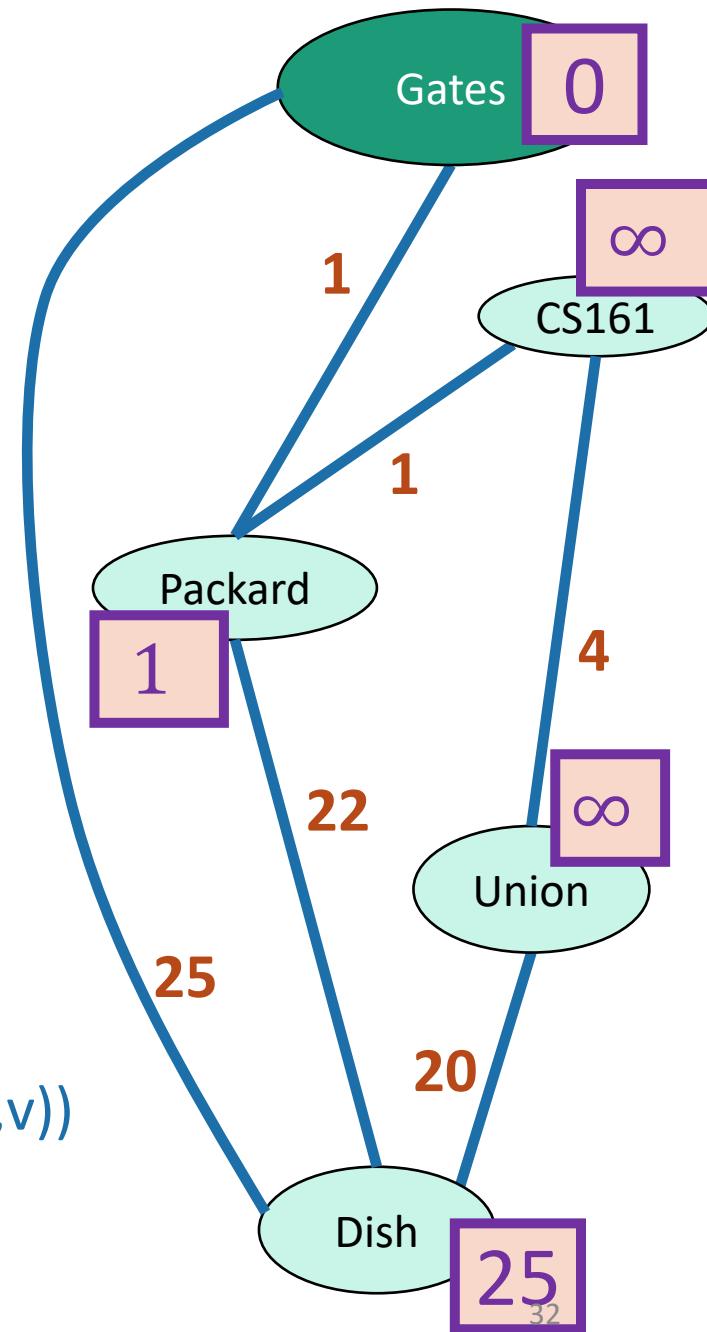


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



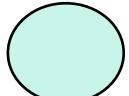
Current node u

- Pick the **not-sure** node u with the smallest estimate  $d[u]$ .
- Update all u's neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **SURE**.
- Repeat

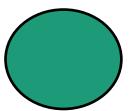


# Dijkstra by example

How far is a node from Gates?



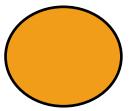
I'm not sure yet



I'm sure

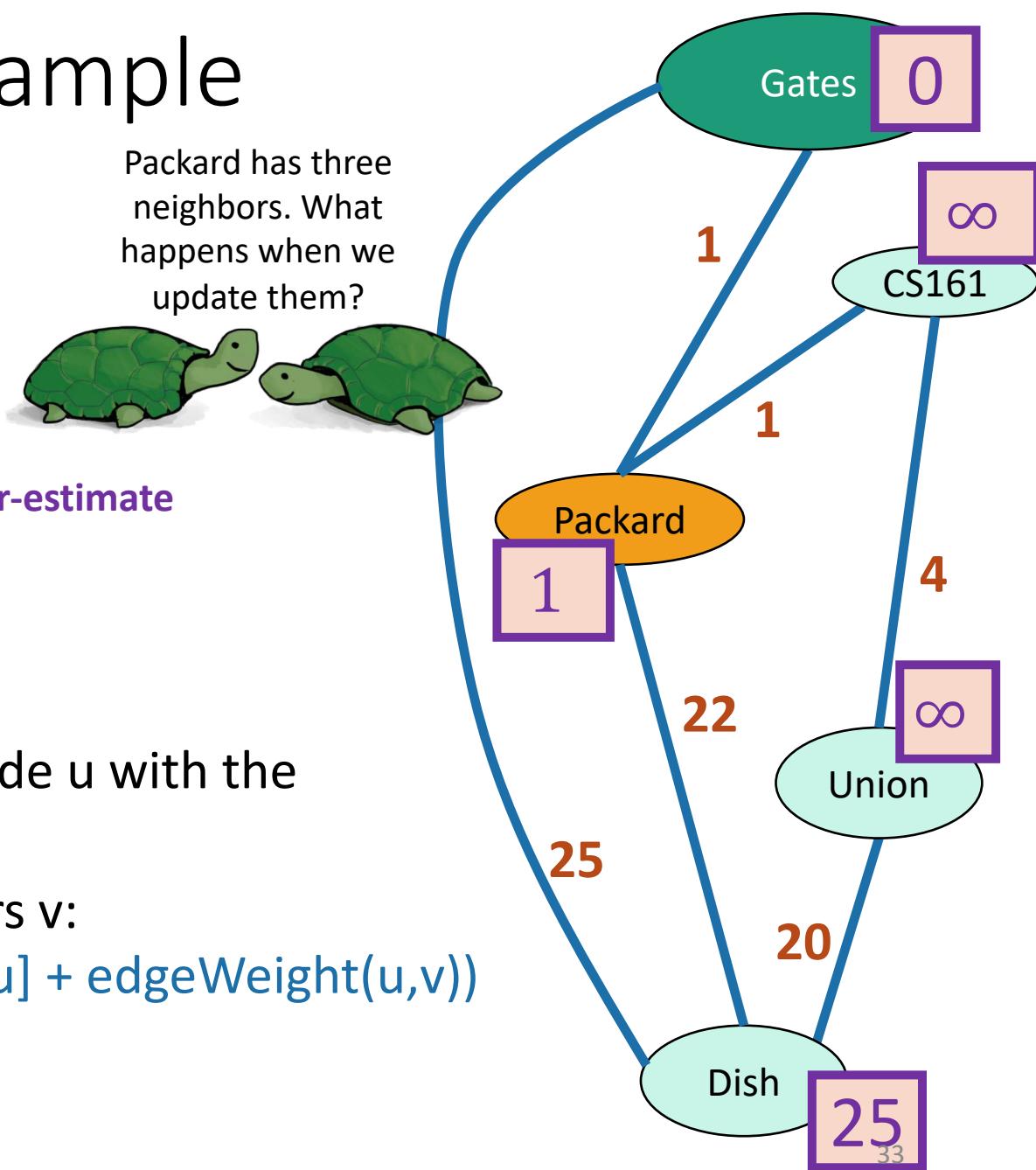


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



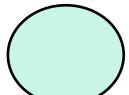
Current node u

- Pick the **not-sure** node u with the smallest estimate  $d[u]$ .
- Update all u's neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **SURE**.
- Repeat

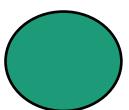


# Dijkstra by example

## How far is a node from Gates?



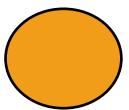
I'm not sure yet



I'm sure

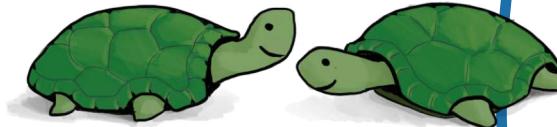


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .

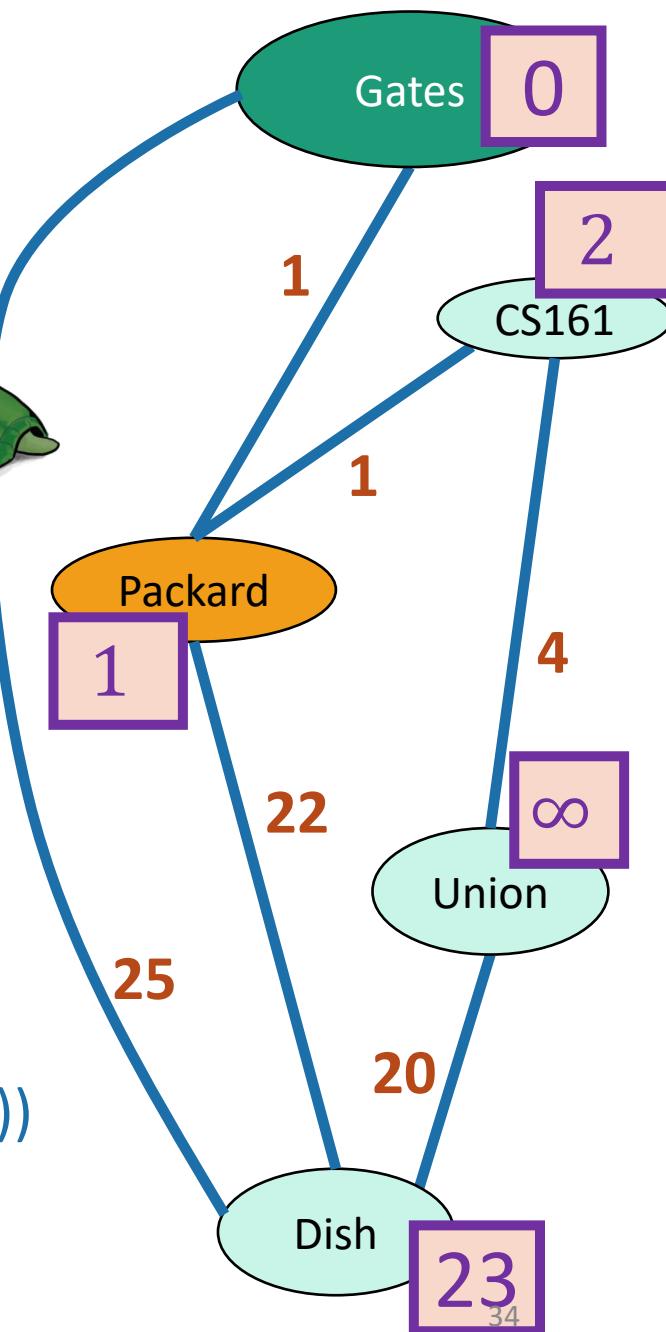


Current node u

- Pick the **not-sure** node u with the smallest estimate  $d[u]$ .
- Update all u's neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **SURE**.
- Repeat

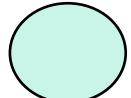


Packard has three neighbors. What happens when we update them?

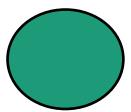


# Dijkstra by example

How far is a node from Gates?



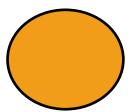
I'm not sure yet



I'm sure

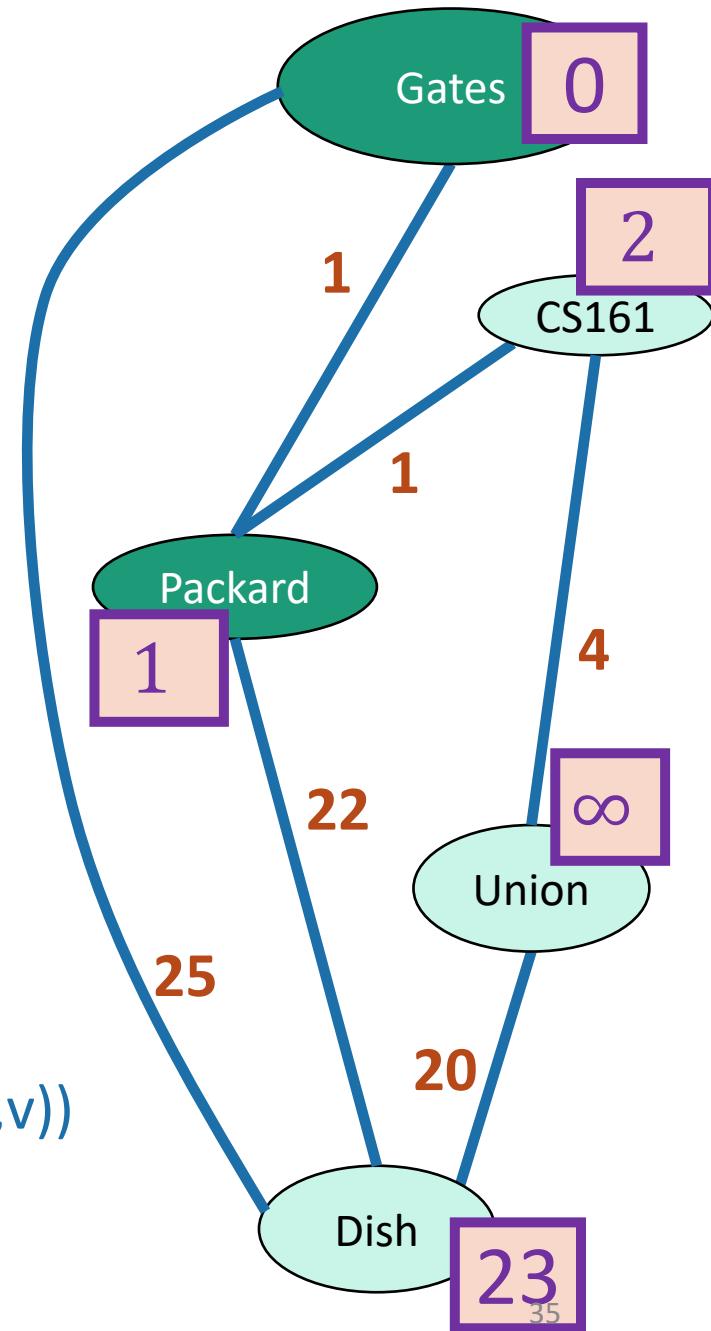


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



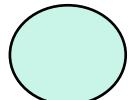
Current node u

- Pick the **not-sure** node u with the smallest estimate  $d[u]$ .
- Update all u's neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **SURE**.
- Repeat

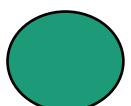


# Dijkstra by example

How far is a node from Gates?



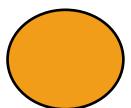
I'm not sure yet



I'm sure

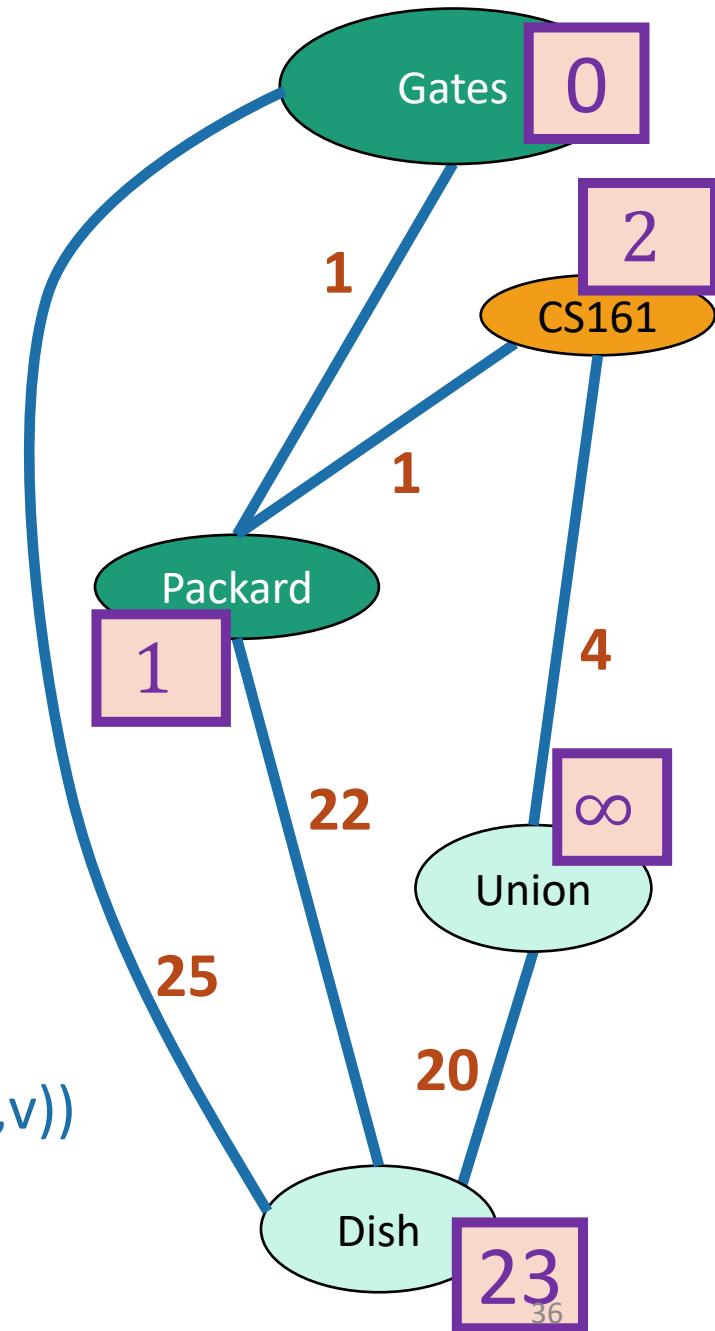


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



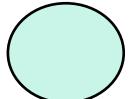
Current node  $u$

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u, v))$
- Mark  $u$  as **SURE**.
- Repeat

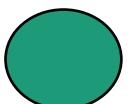


# Dijkstra by example

How far is a node from Gates?



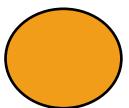
I'm not sure yet



I'm sure

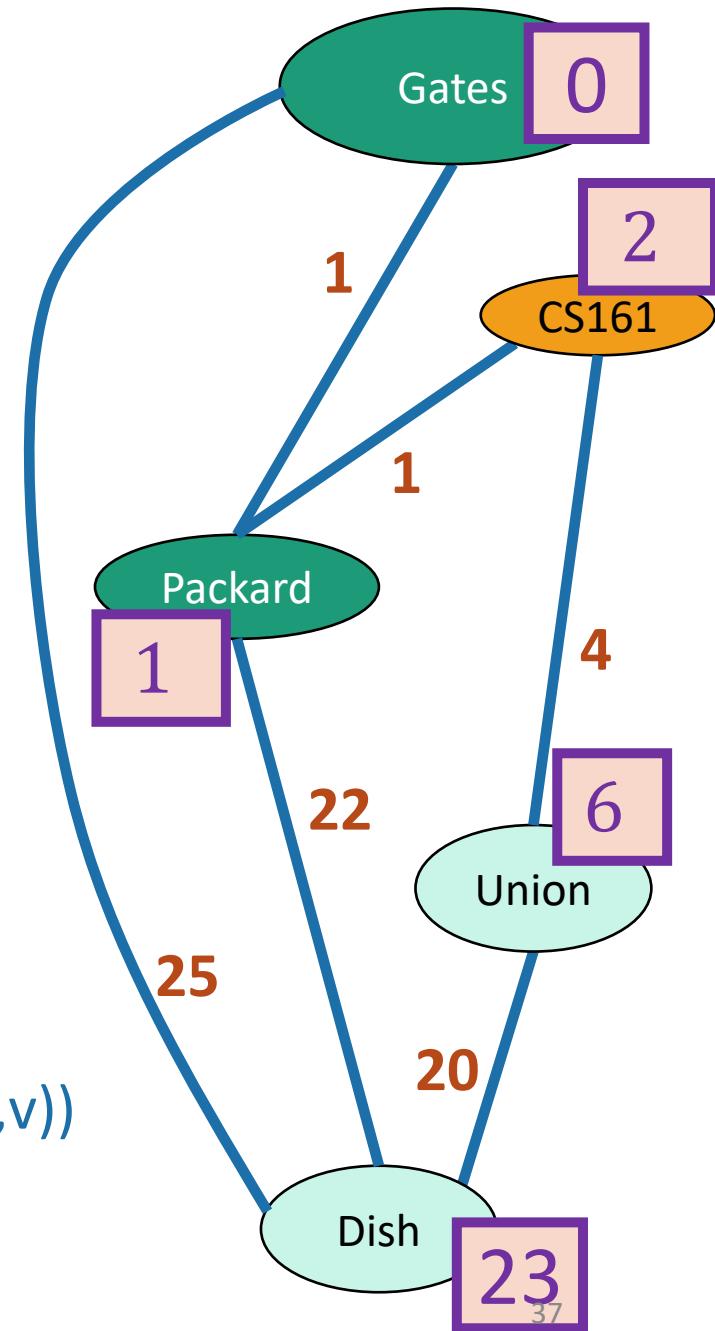


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



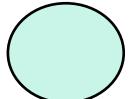
Current node  $u$

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u, v))$
- Mark  $u$  as **SURE**.
- Repeat

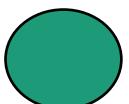


# Dijkstra by example

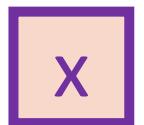
How far is a node from Gates?



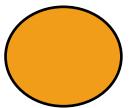
I'm not sure yet



I'm sure

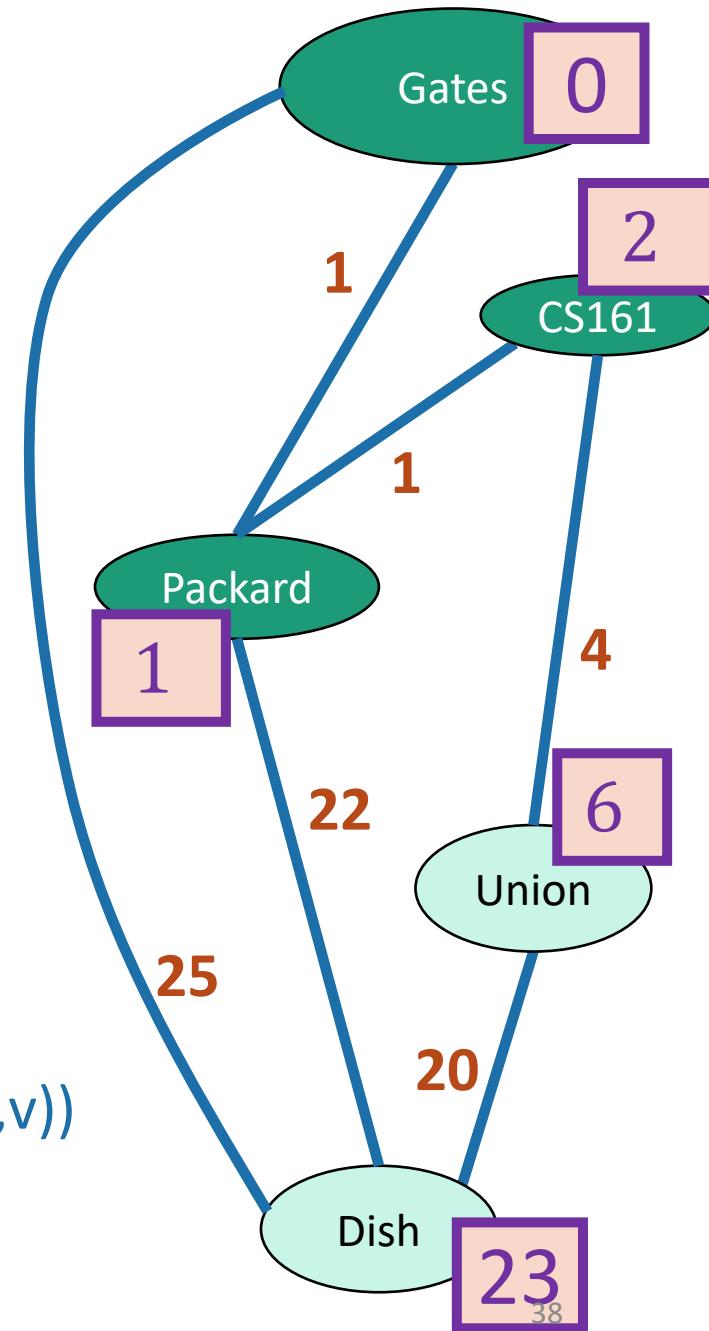


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



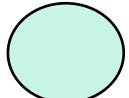
Current node u

- Pick the **not-sure** node u with the smallest estimate  $d[u]$ .
- Update all u's neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **SURE**.
- Repeat

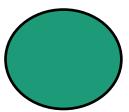


# Dijkstra by example

How far is a node from Gates?



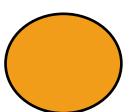
I'm not sure yet



I'm sure

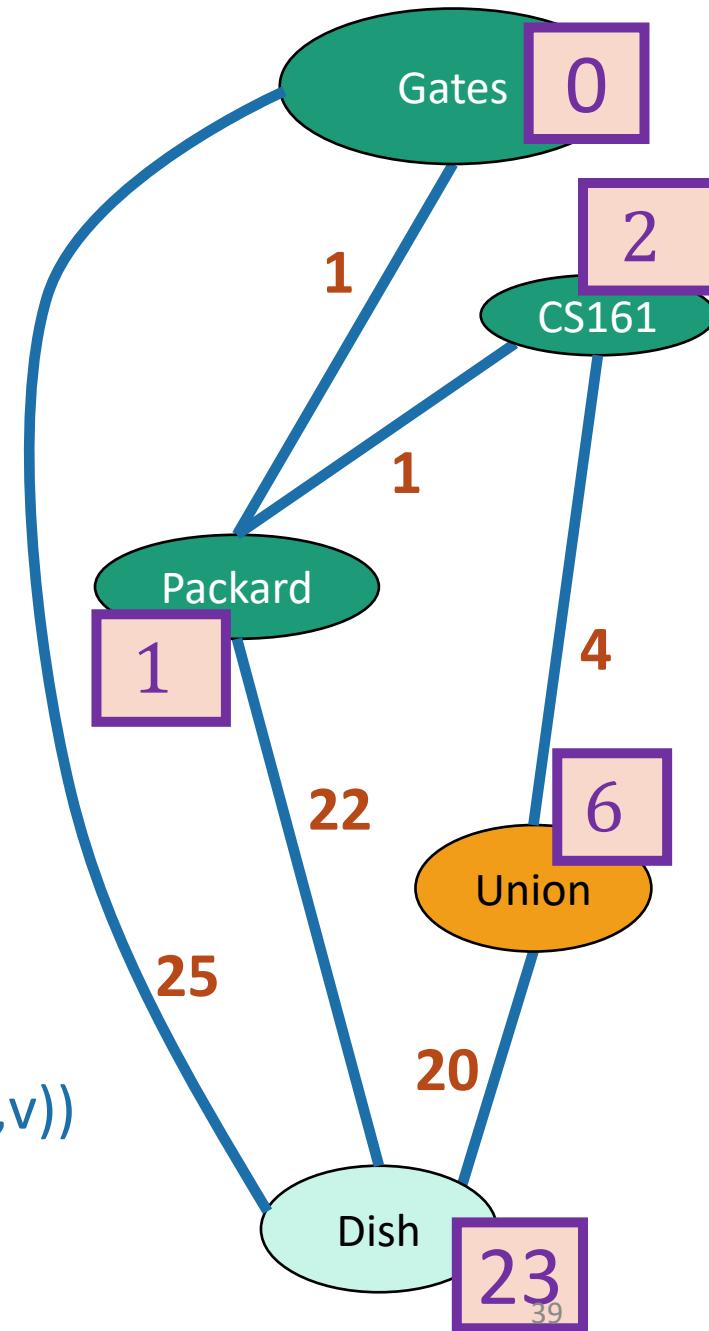


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



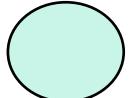
Current node  $u$

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u, v))$
- Mark  $u$  as **SURE**.
- Repeat

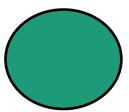


# Dijkstra by example

How far is a node from Gates?



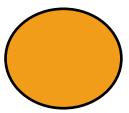
I'm not sure yet



I'm sure

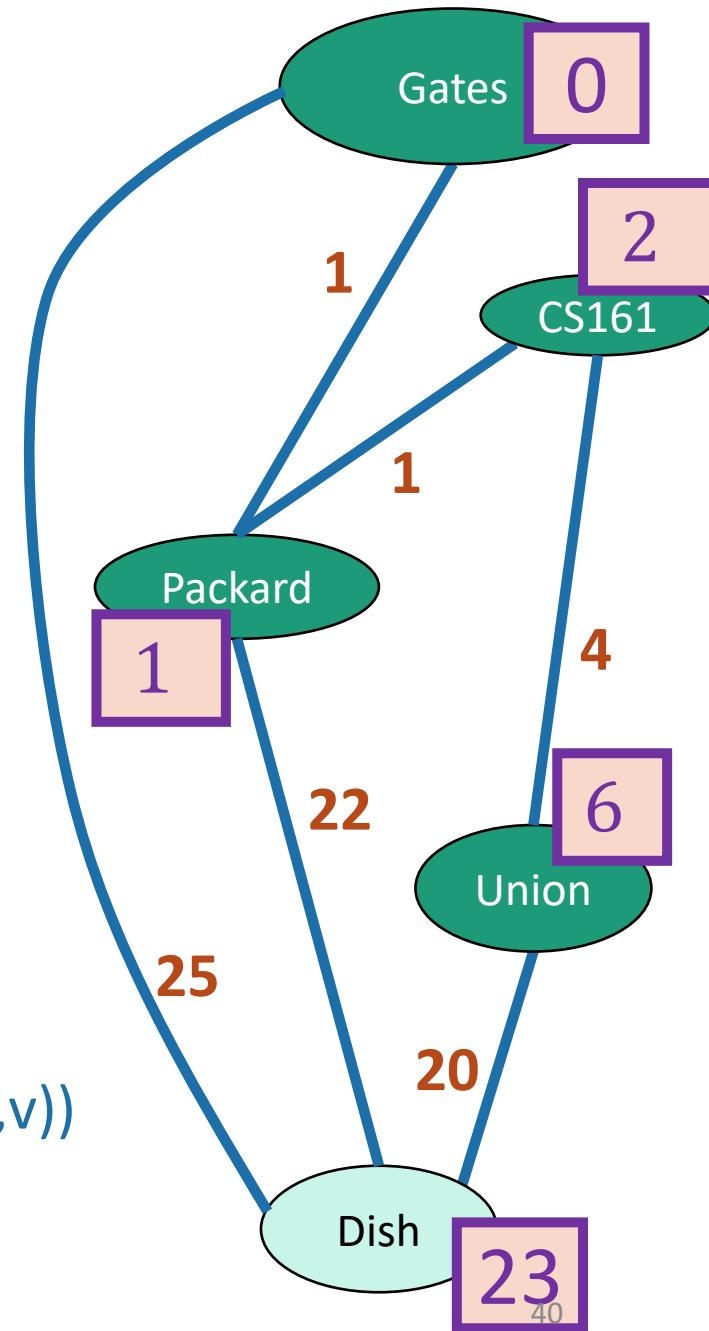


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



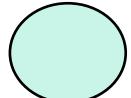
Current node u

- Pick the **not-sure** node u with the smallest estimate  $d[u]$ .
- Update all u's neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **SURE**.
- Repeat

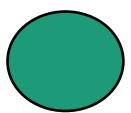


# Dijkstra by example

How far is a node from Gates?



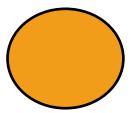
I'm not sure yet



I'm sure

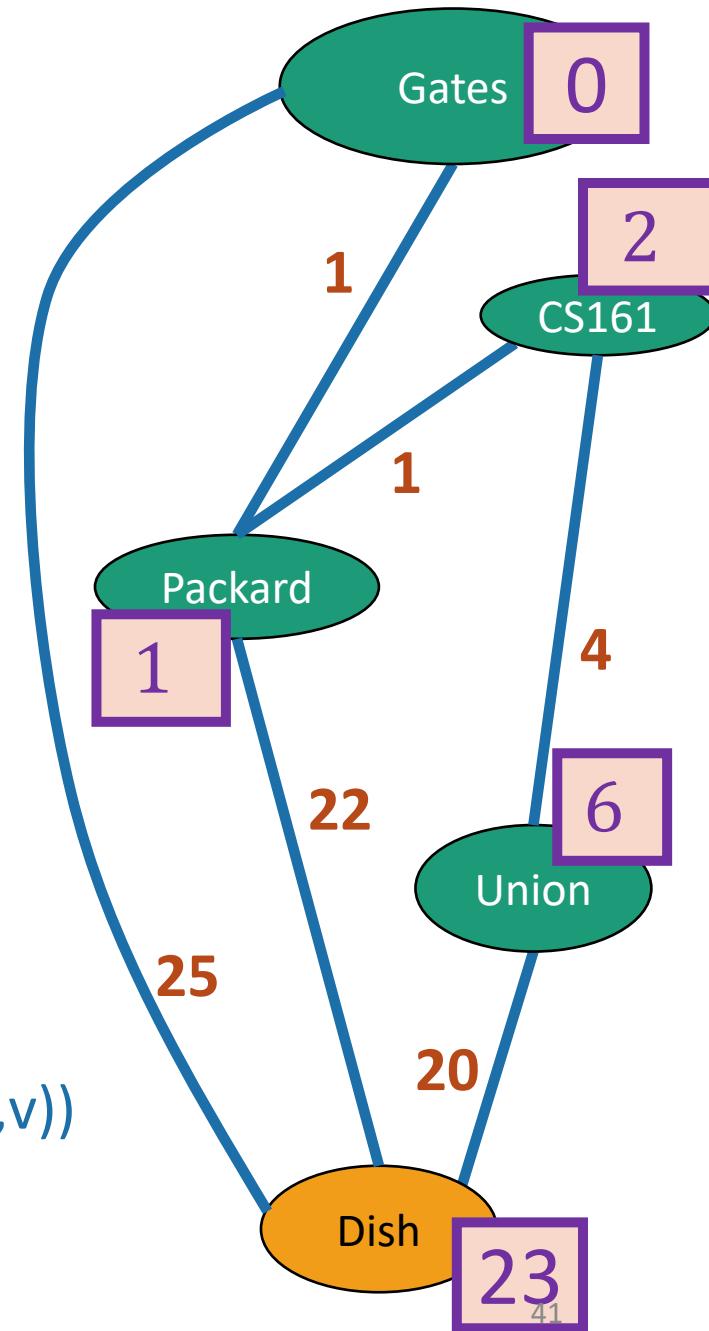


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



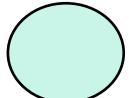
Current node u

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u, v))$
- Mark  $u$  as **SURE**.
- Repeat

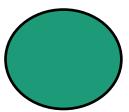


# Dijkstra by example

How far is a node from Gates?



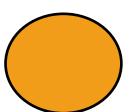
I'm not sure yet



I'm sure

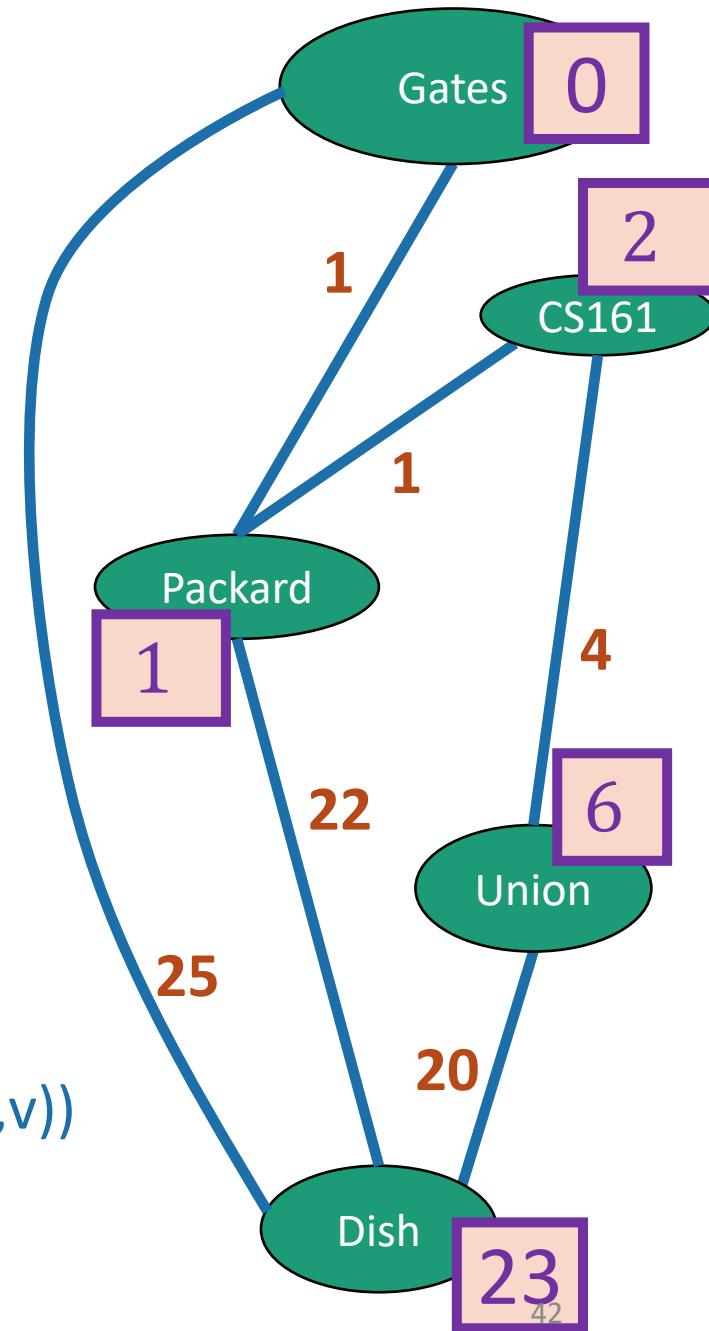


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



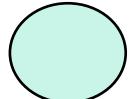
Current node u

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u, v))$
- Mark  $u$  as **SURE**.
- Repeat

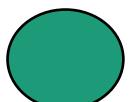


# Dijkstra by example

How far is a node from Gates?



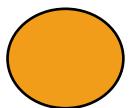
I'm not sure yet



I'm sure

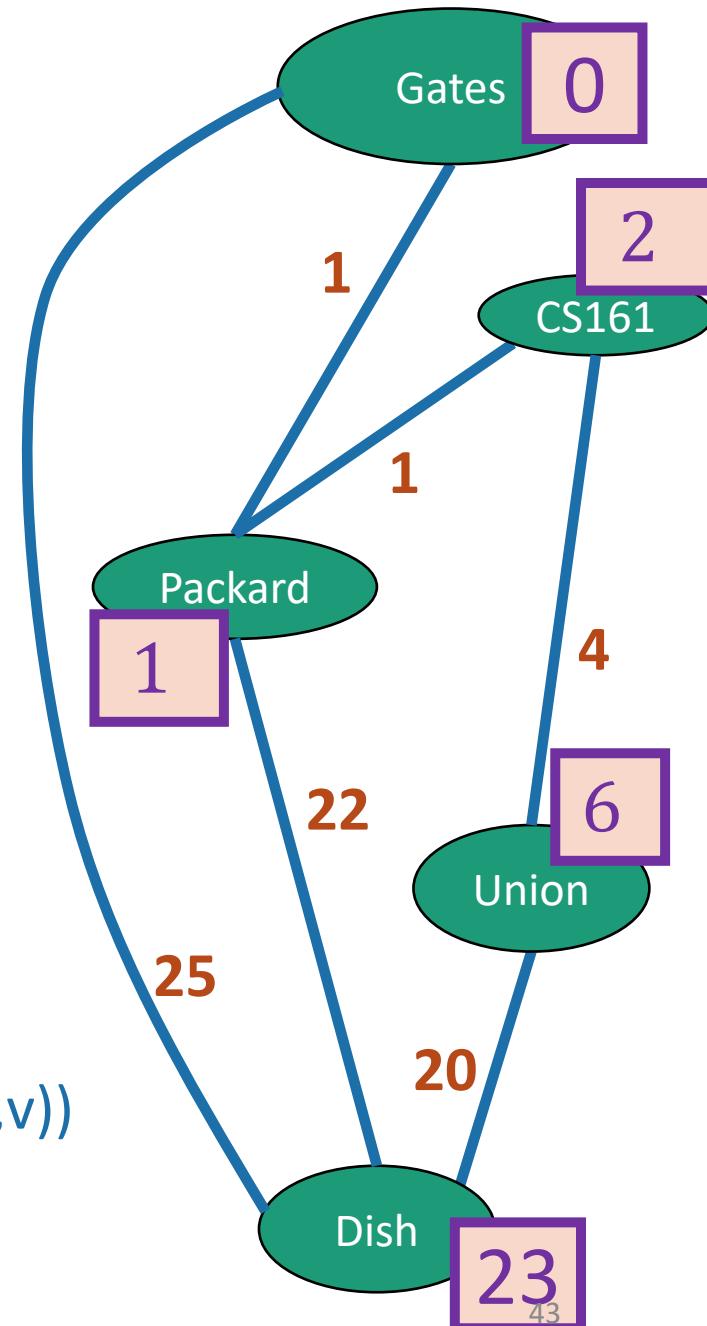


$x = d[v]$  is my best **over-estimate** for  $\text{dist}(\text{Gates}, v)$ .



Current node u

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u, v))$
- Mark  $u$  as **SURE**.
- Repeat
- After all nodes are **SURE**, say that  $d(\text{Gates}, v) = d[v]$  for all  $v$



# Dijkstra's algorithm

**Dijkstra( $G, s$ ):**

- Set all vertices to **not-sure**
- $d[v] = \infty$  for all  $v$  in  $V$
- $d[s] = 0$
- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
  - **For**  $v$  in  $u.\text{neighbors}$ :
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u, v))$
  - Mark  $u$  as **sure**.
- Now  $d(s, v) = d[v]$

Lots of implementation details left un-explained.  
We'll get to that!

See IPython Notebook for code!

# As usual



- Does it work?
  - Yes.
- Is it fast?
  - Depends on how you implement it.

# Why does this work?

- **Theorem:**

- Suppose we run Dijkstra on  $G = (V, E)$ , starting from  $s$ .
- At the end of the algorithm, the estimate  $d[v]$  is the actual distance  $d(s, v)$ .

Let's rename "Gates" to "s", our starting vertex.

- Proof outline:

- **Claim 1:** For all  $v$ ,  $d[v] \geq d(s, v)$ .
- **Claim 2:** When a vertex  $v$  is marked **sure**,  $d[v] = d(s, v)$ .

- **Claims 1 and 2 imply the theorem.**

- When  $v$  is marked **sure**,  $d[v] = d(s, v)$ .  

- $d[v] \geq d(s, v)$  and never increases, so after  $v$  is **sure**,  $d[v]$  stops changing.
- This implies that at any time *after*  $v$  is marked **sure**,  $d[v] = d(s, v)$ .
- All vertices are **sure** at the end, so all vertices end up with  $d[v] = d(s, v)$ .

Next let's prove the claims!

# Claim 1

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

**Informally:**

- Every time we update  $d[v]$ , we have a path in mind:

$$d[v] \leftarrow \min( d[v] , d[u] + \text{edgeWeight}(u,v) )$$

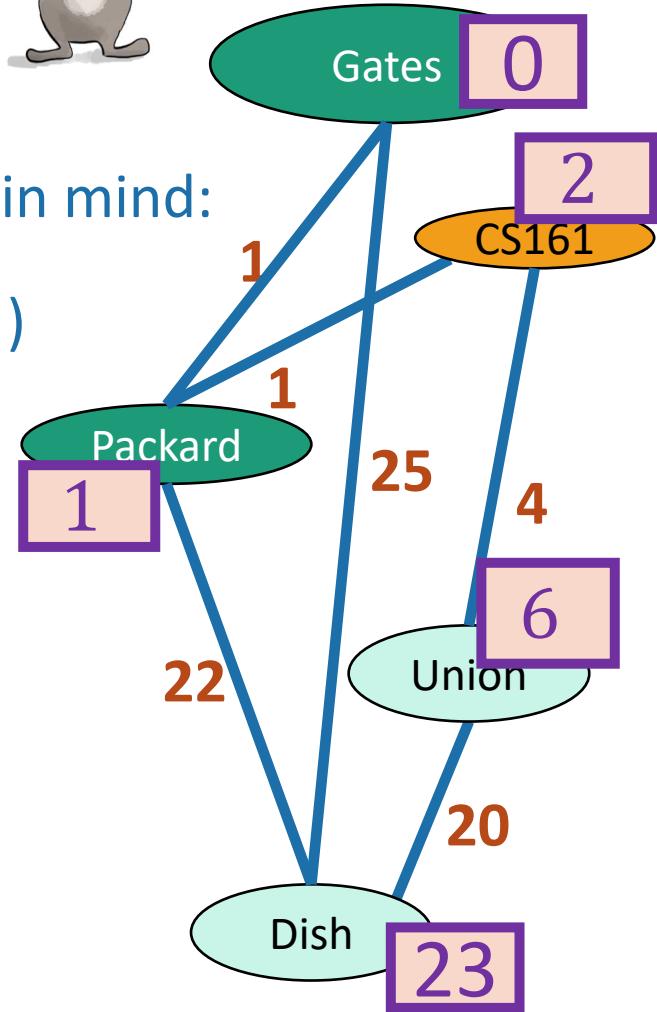
Whatever path we  
had in mind before

The shortest path to  $u$ , and  
then the edge from  $u$  to  $v$ .

- $d[v] = \text{length of the path we have in mind}$   
 $\geq \text{length of shortest path}$   
 $= d(s,v)$

**Formally:**

- We should prove this by induction.
  - (See skipped slide or do it yourself)



# Claim 1

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

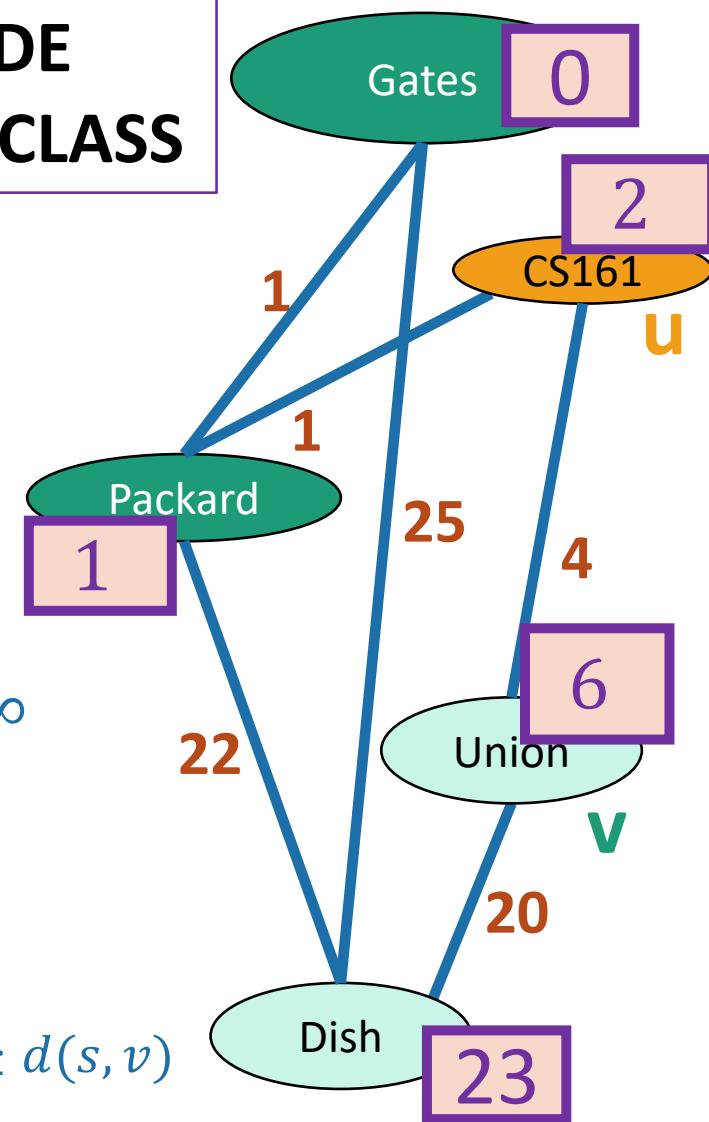
- Inductive hypothesis.
  - After  $t$  iterations of Dijkstra,  
 $d[v] \geq d(s, v)$  for all  $v$ .
- Base case:
  - At step 0,  $d(s, s) = 0$ , and  $d(s, v) \leq \infty$
- Inductive step: say hypothesis holds for  $t$ .
  - At step  $t+1$ :
    - Pick  $u$ ; for each neighbor  $v$ :
    - $d[v] \leftarrow \min( d[v] , d[u] + w(u,v) ) \geq d(s, v)$

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

$$\begin{aligned} d(s, v) &\leq d(s, u) + d(u, v) \\ &\leq d[u] + w(u, v) \end{aligned}$$

using induction again for  $d[u]$

THIS SLIDE  
SKIPPED IN CLASS



So the inductive hypothesis holds for  $t+1$ , and Claim 1 follows.

Note, this slide changed since lecture: I proved the inductive step but didn't state the rest of the induction argument!

## Claim 2

When a vertex  $u$  is marked **sure**,  $d[u] = d(s,u)$

- **Inductive Hypothesis:**

- When we mark the  $t$ 'th vertex  $v$  as **sure**,  $d[v] = \text{dist}(s,v)$ .

- **Base case:**

- The first vertex marked **sure** is  $s$ , and  $d[s] = d(s,s) = 0$ .

(Note: we are assuming here that the edge weights are non-negative, so there's no way to sneakily get from  $s$  to  $s$  with cost less than zero!)

- **Inductive step:**

- Suppose that we are about to add  $u$  to the **sure** list.
- That is, we picked  $u$  in the first line here:

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
- Mark  $u$  as **sure**.
- Repeat

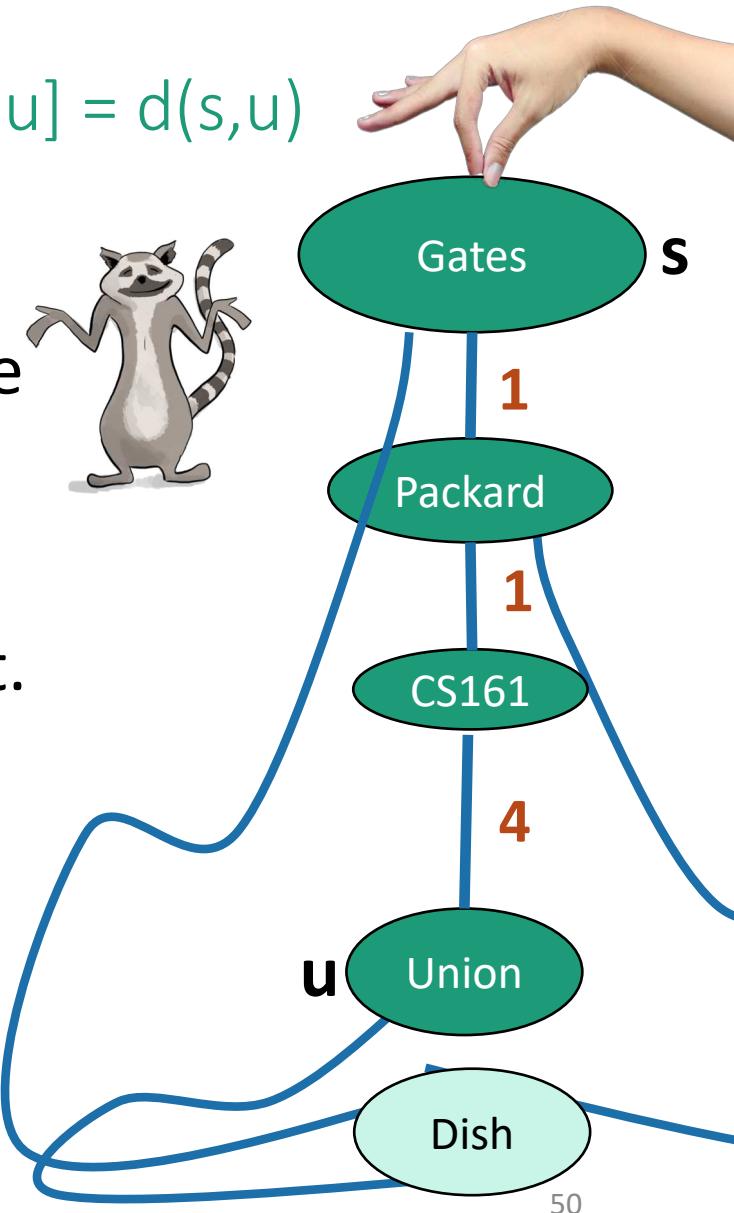
- Assume by induction that every  $v$  already marked **sure** has  $d[v] = d(s,v)$ .
- Want to show that  $d[u] = d(s,u)$ .

YOINK!

# Intuition

When a vertex  $u$  is marked sure,  $d[u] = d(s,u)$

- The first path that lifts  $u$  off the ground is the shortest one.
- But we should actually prove it.

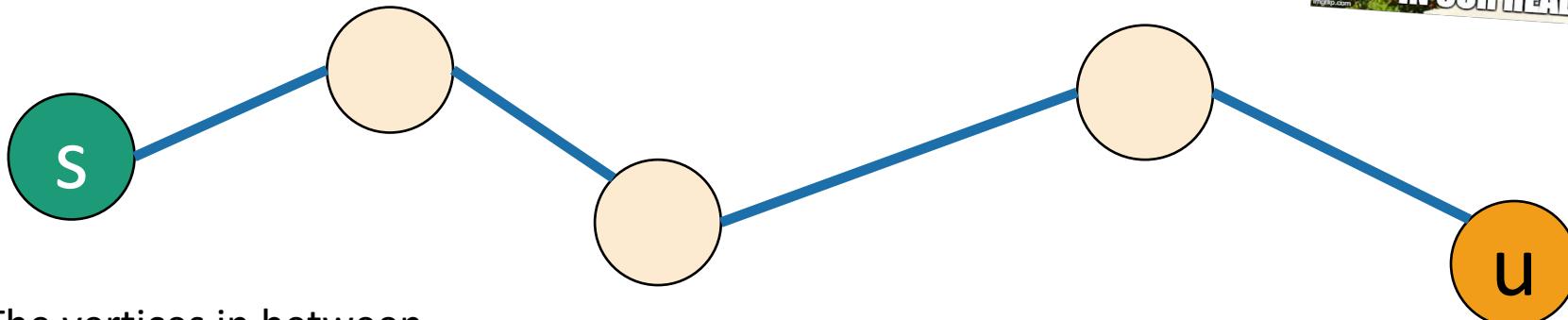


# Claim 2

Inductive step

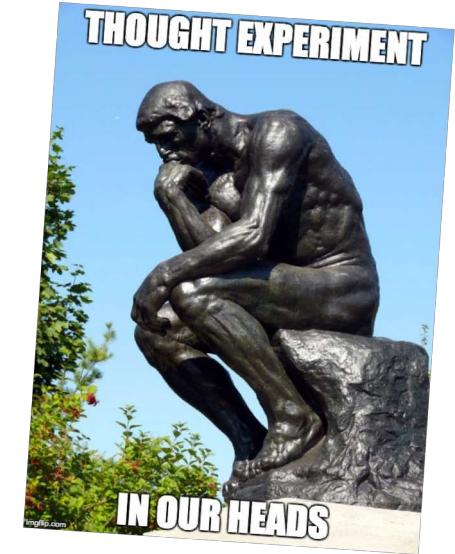
**Temporary definition:**  
 $v$  is “good” means that  $d[v] = d(s,v)$

- Want to show that  $u$  is good.
- Consider a **true** shortest path from  $s$  to  $u$ :



The vertices in between  
are beige because they  
may or may not be **sure**.

True shortest path.



# Claim 2

Inductive step

**Temporary definition:**

$v$  is “good” means that  $d[v] = d(s,v)$



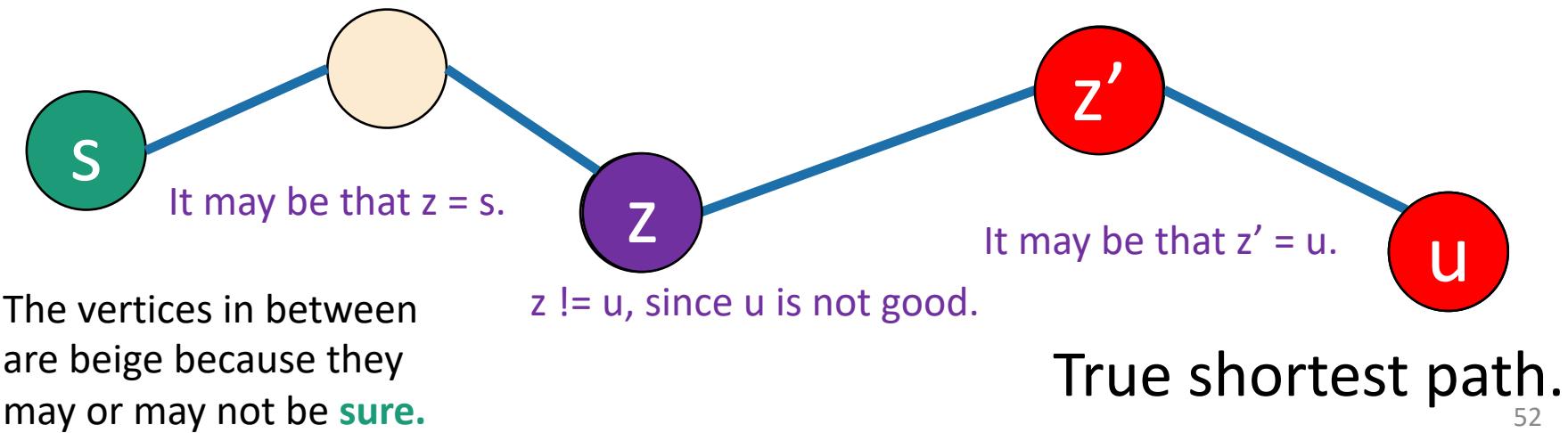
means good



means not good

“by way of contradiction”

- Want to show that  $u$  is good. **BWOC, suppose  $u$  isn't good.**
- Say  $z$  is the last good vertex before  $u$ .
- $z'$  is the vertex after  $z$ .



The vertices in between are beige because they may or may not be **sure**.

$z \neq u$ , since  $u$  is not good.

True shortest path.

# Claim 2

Inductive step

**Temporary definition:**

$v$  is “good” means that  $d[v] = d(s,v)$



means good



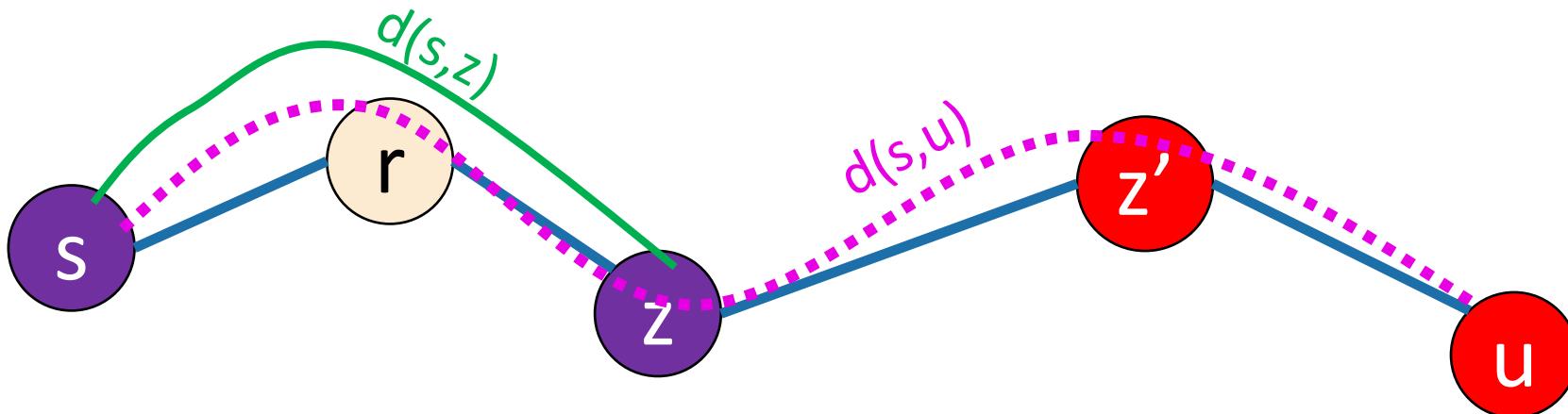
means not good

- Want to show that  $u$  is good. BWOC, suppose  $u$  isn't good.

$$d[z] = d(s,z) \leq d(s,u) \leq d[u]$$

$z$  is good

Subpaths of  
shortest paths are  
shortest paths. AND, also that  $d(z,u) \geq 0$ ,  
since all of the edge-weights  
are non-negative!



# Claim 2

Inductive step

**Temporary definition:**

$v$  is “good” means that  $d[v] = d(s, v)$



means good



means not good

- Want to show that  $u$  is good. BWOC, suppose  $u$  isn't good.

$$d[z] = d(s, z) \leq d(s, u) \leq d[u]$$

$z$  is good

Subpaths of  
shortest paths are  
shortest paths.

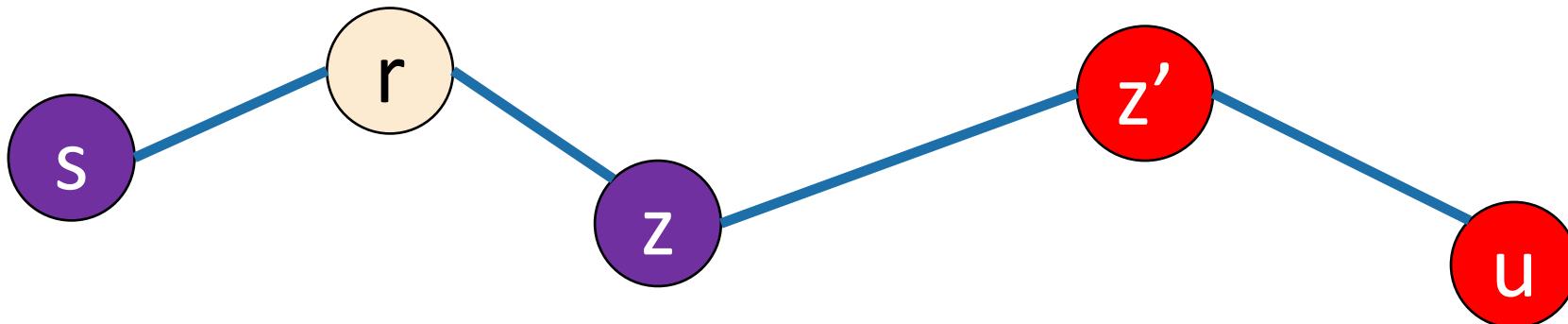
Claim 1

- If  $d[z] = d[u]$ , then  $u$  is good.
- So  $d[z] < d[u]$ , so  $z$  is **sure**.



But  $u$  is not good!

We chose  $u$  so that  $d[u]$  was  
smallest of the unsure vertices.



# Claim 2

Inductive step

**Temporary definition:**

$v$  is “good” means that  $d[v] = d(s,v)$



means good



means not good

- Want to show that  $u$  is good. BWOC, suppose  $u$  isn't good.

$$d[z] = d(s,z) \leq d(s,u) \leq d[u]$$

$z$  is good

Subpaths of  
shortest paths are  
shortest paths.

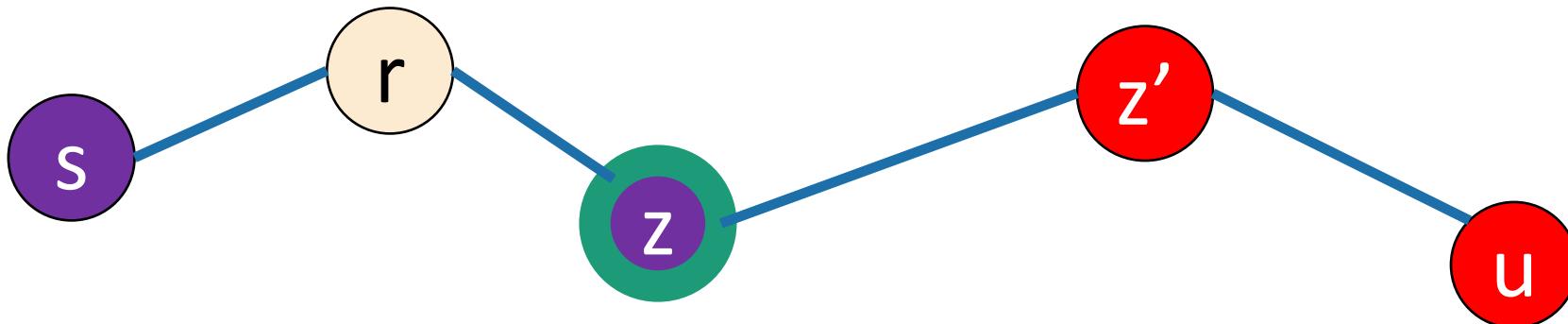
Claim 1

- If  $d[z] = d[u]$ , then  $u$  is good.
- So  $d[z] < d[u]$ , so  $z$  is **sure**.



But  $u$  is not good!

We chose  $u$  so that  $d[u]$  was  
smallest of the unsure vertices.



# Claim 2

Inductive step

**Temporary definition:**

$v$  is “good” means that  $d[v] = d(s, v)$



means good



means not good

- Want to show that  $u$  is good. BWOC, suppose  $u$  isn't good.

- If  $z$  is **sure** then we've already updated  $z'$ :

- $d[z'] \leq d[z] + w(z, z')$  def of update

$$= d(s, z) + w(z, z')$$

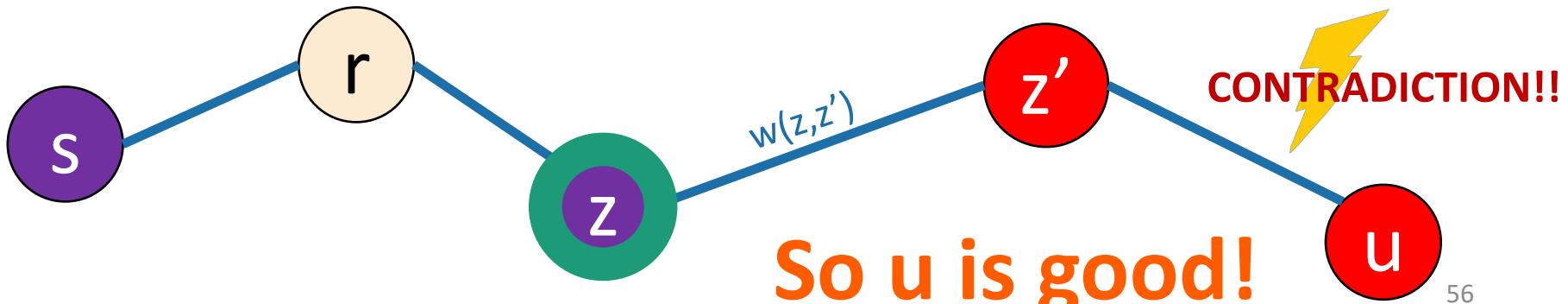
By induction when  $z$  was added to  
the sure list it had  $d(s, z) = d[z]$

That is, the value of  
 $d[z]$  when  $z$  was  
marked sure...

$$= d(s, z') \quad \text{sub-paths of shortest paths are shortest paths}$$

$$\leq d[z'] \quad \text{Claim 1}$$

So  $d(s, z') = d[z']$  and so  $z'$  is good.



## Claim 2

When a vertex  $u$  is marked sure,  $d[u] = d(s,u)$

- **Inductive Hypothesis:**

- When we mark the  $t$ 'th vertex  $v$  as sure,  $d[v] = \text{dist}(s,v)$ .

- **Base case:**

- The first vertex marked **sure** is  $s$ , and  $d[s] = d(s,s) = 0$ .

- **Inductive step:**

- Suppose that we are about to add  $u$  to the **sure** list.
- That is, we picked  $u$  in the first line here:

- Pick the **not-sure** node  $u$  with the smallest estimate  **$d[u]$** .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
- Mark  $u$  as **sure**.
- Repeat

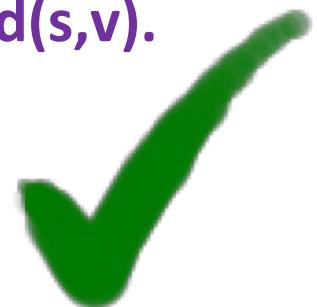
- Assume by induction that every  $v$  already marked **sure** has  $d[v] = d(s,v)$ .
- Want to show that  $d[u] = d(s,u)$ .

**Conclusion:** Claim 2 holds!

*Now back to  
this slide*

# Why does this work?

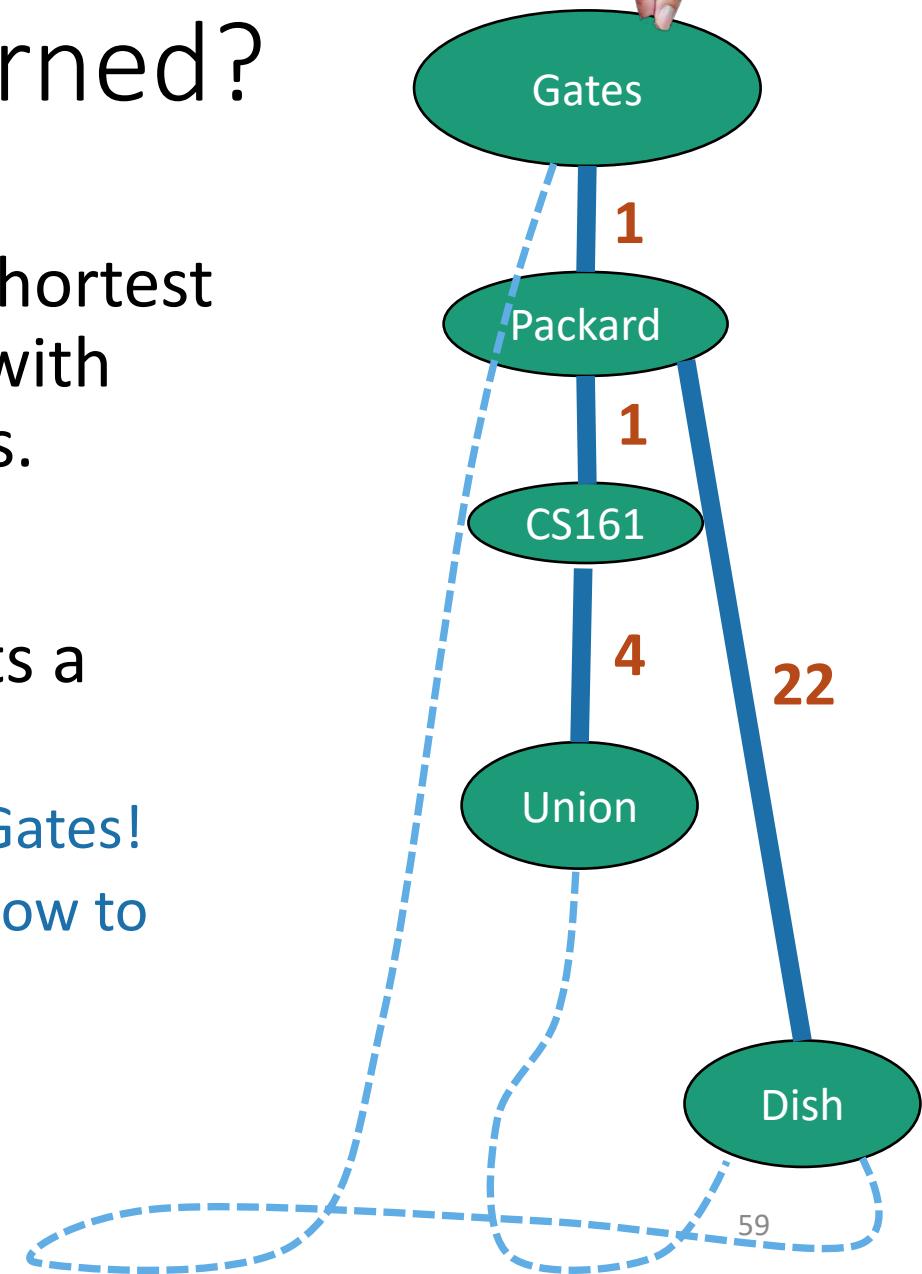
- **Theorem:**
  - Run Dijkstra on  $G = (V, E)$  starting from  $s$ .
  - At the end of the algorithm, the estimate  $d[v]$  is the actual distance  $d(s, v)$ .
- Proof outline:
  - **Claim 1:** For all  $v$ ,  $d[v] \geq d(s, v)$ .
  - **Claim 2:** When a vertex is marked **sure**,  $d[v] = d(s, v)$ .
- **Claims 1 and 2 imply the theorem.**



YOINK!

# What have we learned?

- Dijkstra's algorithm finds shortest paths in weighted graphs with non-negative edge weights.
- Along the way, it constructs a nice tree.
  - We could post this tree in Gates!
  - Then people would know how to get places quickly.



# As usual

- Does it work?
  - Yes.
- Is it fast?
  - Depends on how you implement it.



# Running time?

## Dijkstra( $G, s$ ):

- Set all vertices to **not-sure**
- $d[v] = \infty$  for all  $v$  in  $V$
- $d[s] = 0$
- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
  - **For**  $v$  in  $u.\text{neighbors}$ :
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u, v))$
    - Mark  $u$  as **sure**.
  - Now  $\text{dist}(s, v) = d[v]$
- $n$  iterations (one per vertex)
- How long does one iteration take?

Depends on how we implement it...

# We need a data structure that:

- Stores unsure vertices  $v$
- Keeps track of  $d[v]$
- Can find  $u$  with minimum  $d[u]$ 
  - `findMin()`
- Can remove that  $u$ 
  - `removeMin(u)`
- Can update (decrease)  $d[v]$ 
  - `updateKey(v, d)`

Just the inner loop:

- Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
- Update all  $u$ 's neighbors  $v$ :
  - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
- Mark  $u$  as **sure**.

Total running time is big-oh of:

$$\sum_{u \in V} \left( T(\text{findMin}) + \left( \sum_{v \in u.\text{neighbors}} T(\text{updateKey}) \right) + T(\text{removeMin}) \right)$$

$$= n(T(\text{findMin}) + T(\text{removeMin})) + m T(\text{updateKey})$$

# If we use an array

- $T(\text{findMin}) = O(n)$
- $T(\text{removeMin}) = O(n)$
- $T(\text{updateKey}) = O(1)$
- Running time of Dijkstra
  - $= O(n(T(\text{findMin}) + T(\text{removeMin})) + m T(\text{updateKey}))$
  - $= O(n^2) + O(m)$
  - $= O(n^2)$

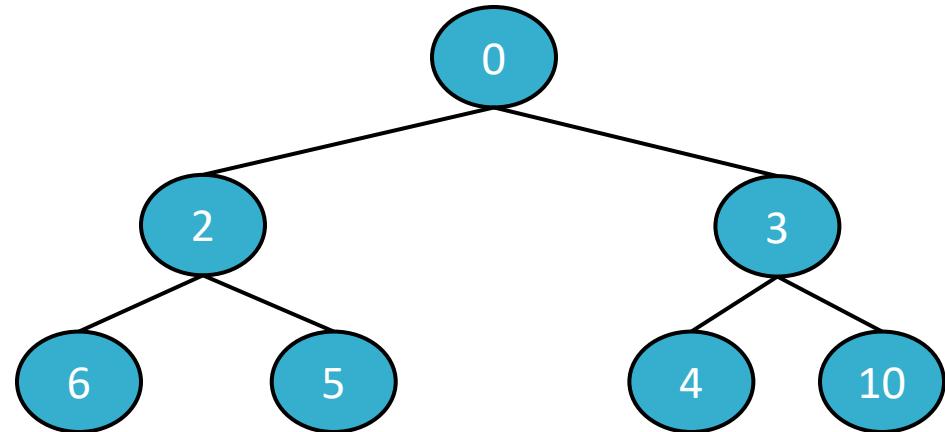
# If we use a red-black tree

- $T(\text{findMin}) = O(\log(n))$
- $T(\text{removeMin}) = O(\log(n))$
- $T(\text{updateKey}) = O(\log(n))$
- Running time of Dijkstra
  - $= O(n(T(\text{findMin}) + T(\text{removeMin})) + m T(\text{updateKey}))$
  - $= O(n\log(n)) + O(m\log(n))$
  - $= O((n + m)\log(n))$

Better than an array if the graph is sparse!  
aka if  $m$  is much smaller than  $n^2$

# Heaps support these operations

- T(findMin)
- T(removeMin)
- T(updateKey)



- A **heap** is a tree-based data structure that has the property that **every node has a smaller key than its children.**
- Not covered in this class – see CS166! (Or CLRS).
- But! We will use them.

# Many heap implementations

Nice chart on Wikipedia:

Operation	Binary <sup>[7]</sup>	Leftist	Binomial <sup>[7]</sup>	Fibonacci <sup>[7][8]</sup>	Pairing <sup>[9]</sup>	Brodal <sup>[10][b]</sup>	Rank-pairing <sup>[12]</sup>	Strict Fibonacci <sup>[13]</sup>
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)^{[c]}$	$O(\log n)^{[c]}$	$O(\log n)$	$O(\log n)^{[c]}$	$O(\log n)$
insert	$O(\log n)$	$\Theta(\log n)$	$\Theta(1)^{[c]}$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)^{[c]}$	$O(\log n)^{[c][d]}$	$\Theta(1)$	$\Theta(1)^{[c]}$	$\Theta(1)$
merge	$\Theta(n)$	$\Theta(\log n)$	$O(\log n)^{[e]}$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

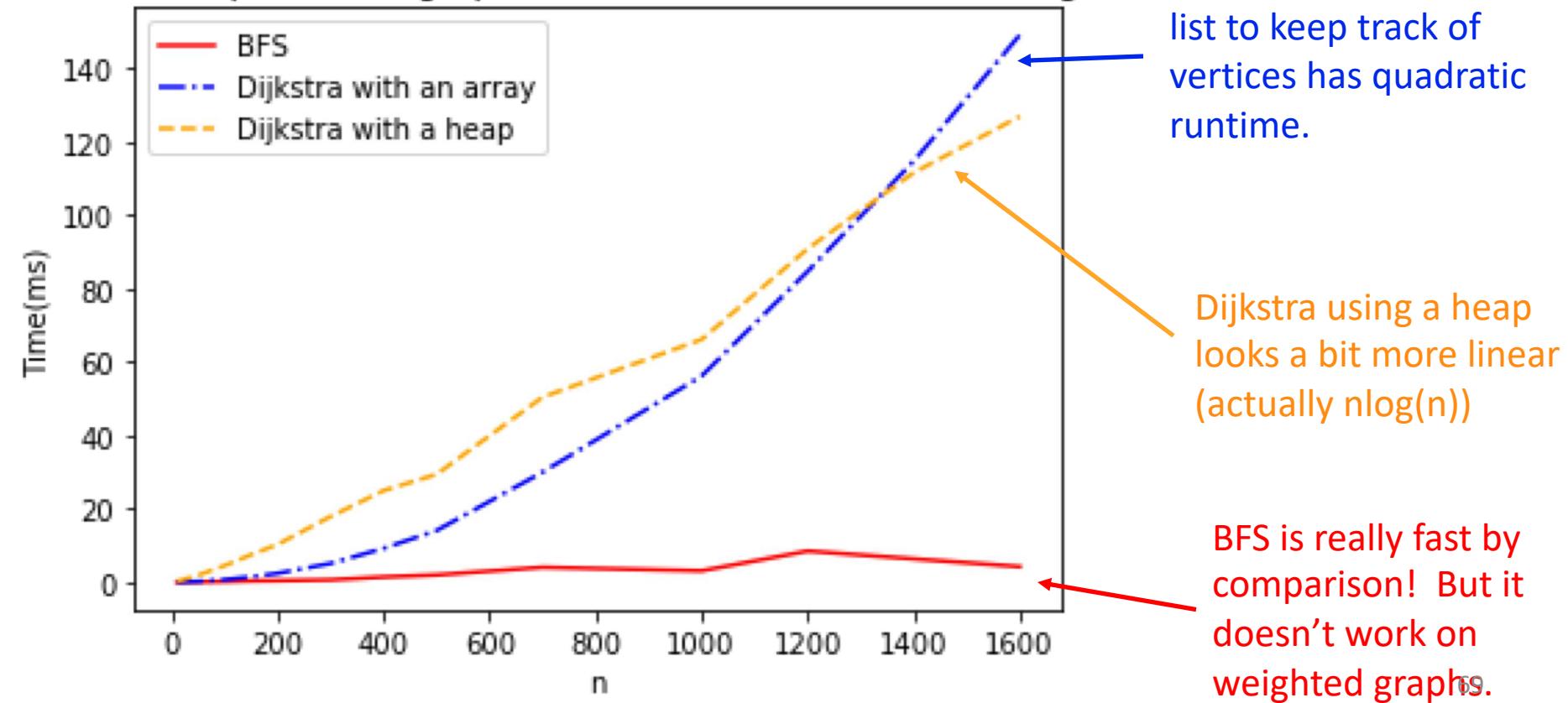
# Say we use a Fibonacci Heap

- $T(\text{findMin}) = O(1)$  (amortized time\*)
- $T(\text{removeMin}) = O(\log(n))$  (amortized time\*)
- $T(\text{updateKey}) = O(1)$  (amortized time\*)
- See CS166 for more! (or CLRS)
- Running time of Dijkstra  
 $= O(n(T(\text{findMin}) + T(\text{removeMin})) + m T(\text{updateKey}))$   
 $= O(n \log(n) + m)$  (amortized time)

\*This means that any sequence of  $d$  `removeMin` calls takes time at most  $O(d \log(n))$ .  
But a few of the  $d$  may take longer than  $O(\log(n))$  and some may take less time..<sup>68</sup>

# In practice

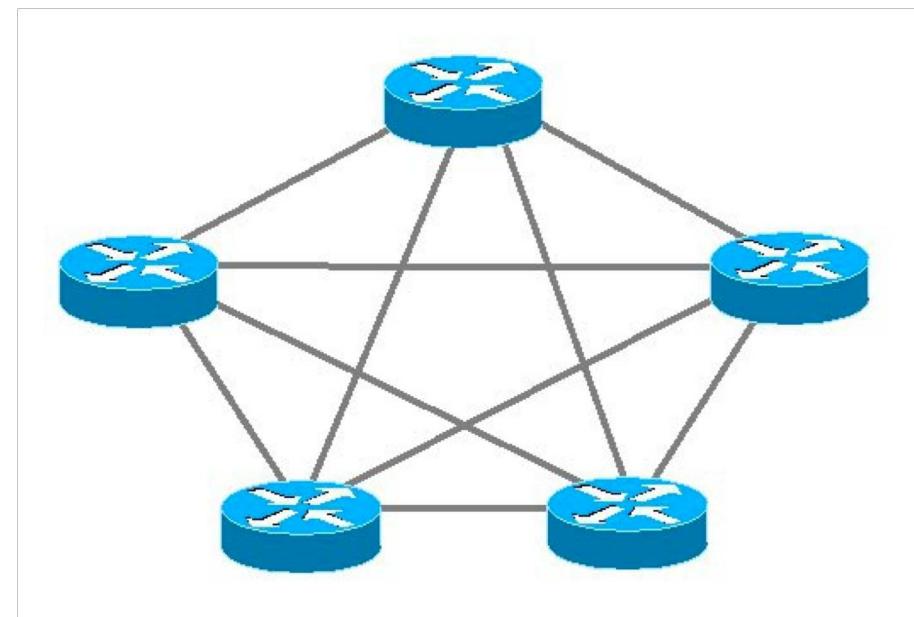
Shortest paths on a graph with  $n$  vertices and about  $5n$  edges



# Dijkstra is used in practice

- eg, [OSPF \(Open Shortest Path First\)](#), a routing protocol for IP networks, uses Dijkstra.

But there are  
some things it's  
not so good at.



# Dijkstra Drawbacks

- Needs **non-negative edge weights**.
- If the weights change, we need to re-run the whole thing.
  - in OSPF, a vertex broadcasts any changes to the network, and then every vertex re-runs Dijkstra's algorithm from scratch.

# Bellman-Ford algorithm

- (-) Slower than Dijkstra's algorithm
- (+) Can handle negative edge weights.
  - Can be useful if you want to say that some edges are actively good to take, rather than costly.
  - Can be useful as a building block in other algorithms.
- (+) Allows for some flexibility if the weights change.
  - We'll see what this means later

# Today: *intro* to Bellman-Ford

- We'll see a definition by example.
- We'll come back to it next lecture with more rigor.
  - Don't worry if it goes by quickly today.
  - There are some skipped slides with pseudocode, but we'll see them again next lecture.
- Basic idea:
  - Instead of picking the  $u$  with the smallest  $d[u]$  to update, just update all of the  $u$ 's simultaneously.

# Bellman-Ford algorithm

SLIDE SKIPPED IN  
CLASS

**Bellman-Ford(G,s):**

- $d[v] = \infty$  for all  $v$  in  $V$
  - $d[s] = 0$
  - **For**  $i=0,\dots,n-1$ :
    - **For**  $u$  in  $V$ :
      - **For**  $v$  in  $u.\text{neighbors}$ :
        - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
- Instead of picking  $u$  cleverly,  
just update for all of the  $u$ 's.

Compare to Dijkstra:

- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node  $u$  with the smallest estimate  $d[u]$ .
  - **For**  $v$  in  $u.\text{neighbors}$ :
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
  - Mark  $u$  as **sure**.

# For pedagogical reasons which we will see next lecture

- We are actually going to change this to be less smart.
- Keep  $n$  arrays:  $d^{(0)}, d^{(1)}, \dots, d^{(n-1)}$

## Bellman-Ford\*(G,s):

- $d^{(0)}[v] = \infty$  for all  $v$  in  $V$
- $d^{(0)}[s] = 0$
- **For**  $i=0, \dots, n-2$ :
  - **For**  $u$  in  $V$ :
    - **For**  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
  - Then  $\text{dist}(s,v) = d^{(n-1)}[v]$

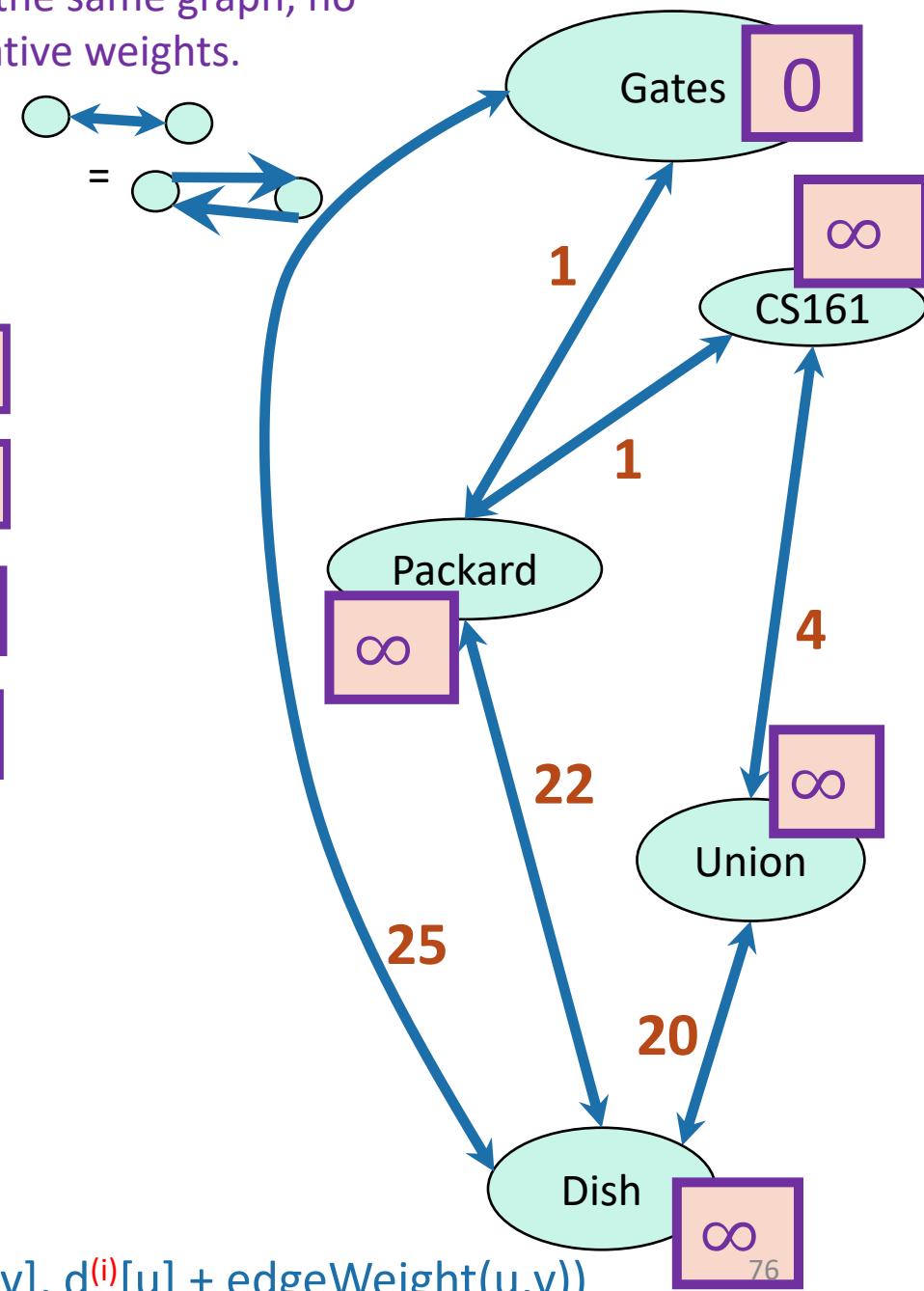
Slightly different than the original Bellman-Ford algorithm, but the analysis is basically the same.

# Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(2)}$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(3)}$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(4)}$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$

Start with the same graph, no negative weights.



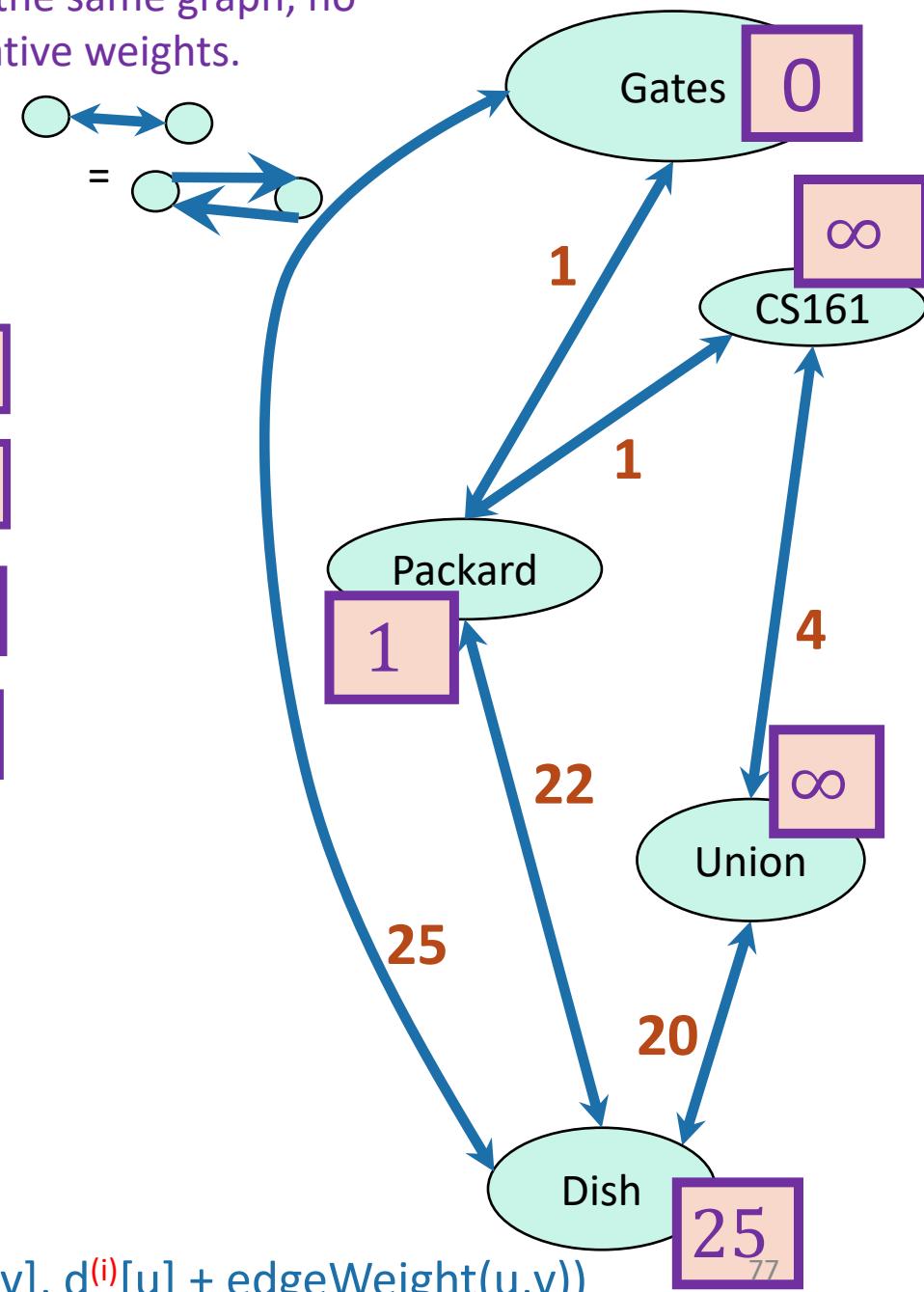
- For  $i=0, \dots, n-2$ :
  - For  $u$  in  $V$ :
    - For  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

# Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	0	1	$\infty$	$\infty$	25
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					

Start with the same graph, no negative weights.



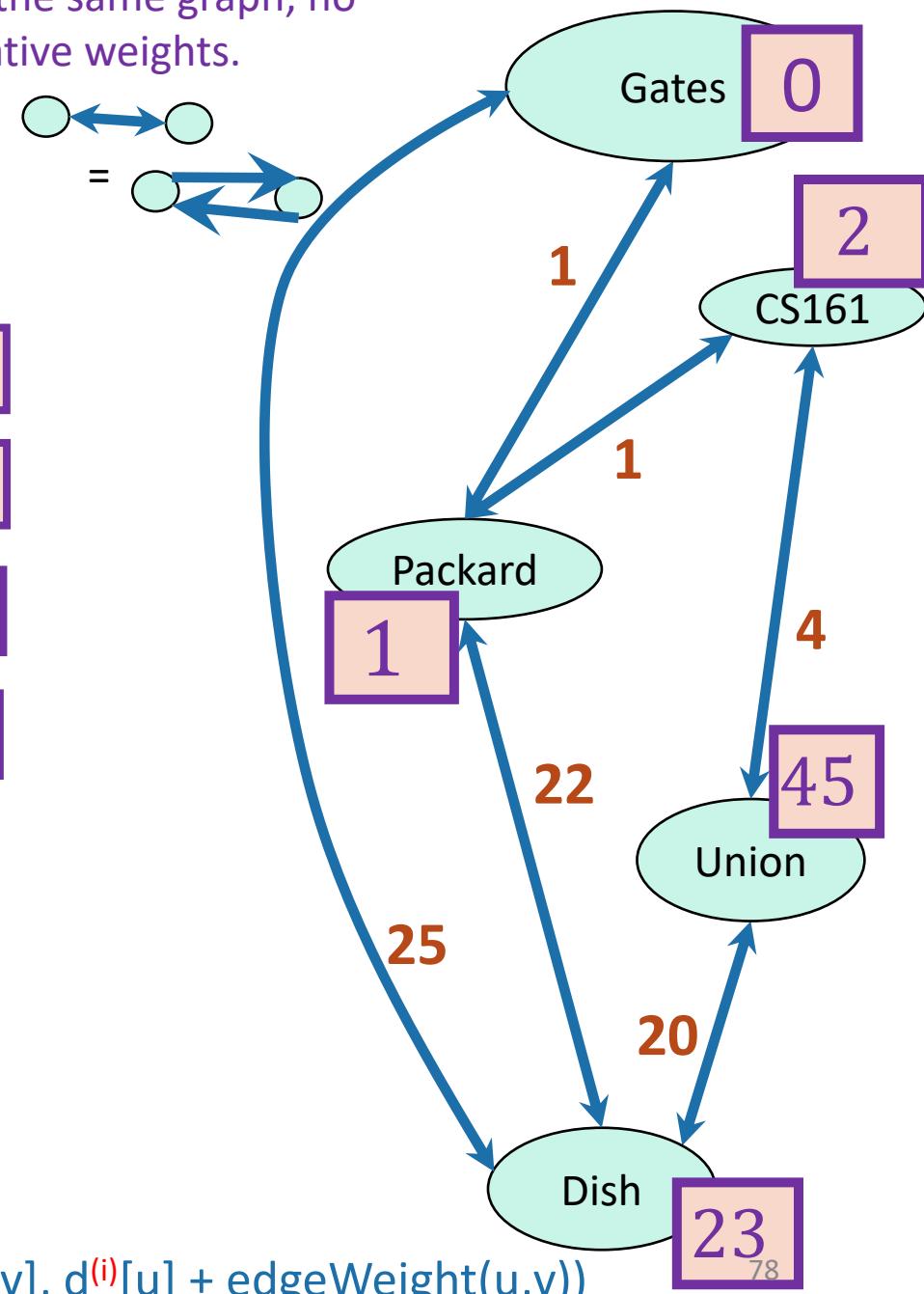
- For  $i=0, \dots, n-2$ :
  - For  $u$  in  $V$ :
    - For  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

# Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	0	1	$\infty$	$\infty$	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$					
$d^{(4)}$					

Start with the same graph, no negative weights.



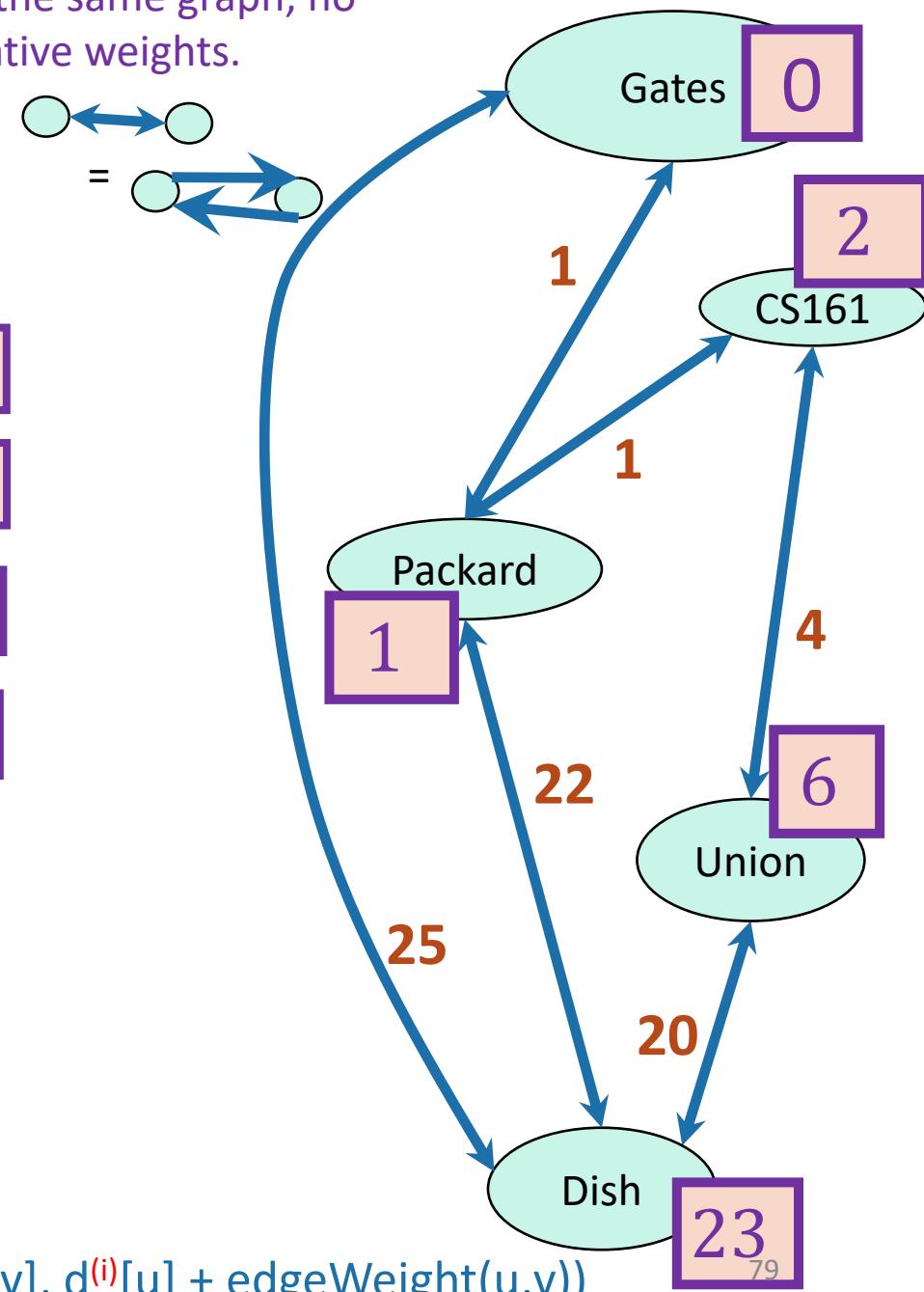
- For  $i=0, \dots, n-2$ :
  - For  $u$  in  $V$ :
    - For  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

# Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	0	1	$\infty$	$\infty$	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$	0	1	2	6	23
$d^{(4)}$					

Start with the same graph, no negative weights.



- For  $i=0, \dots, n-2$ :
  - For  $u$  in  $V$ :
    - For  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

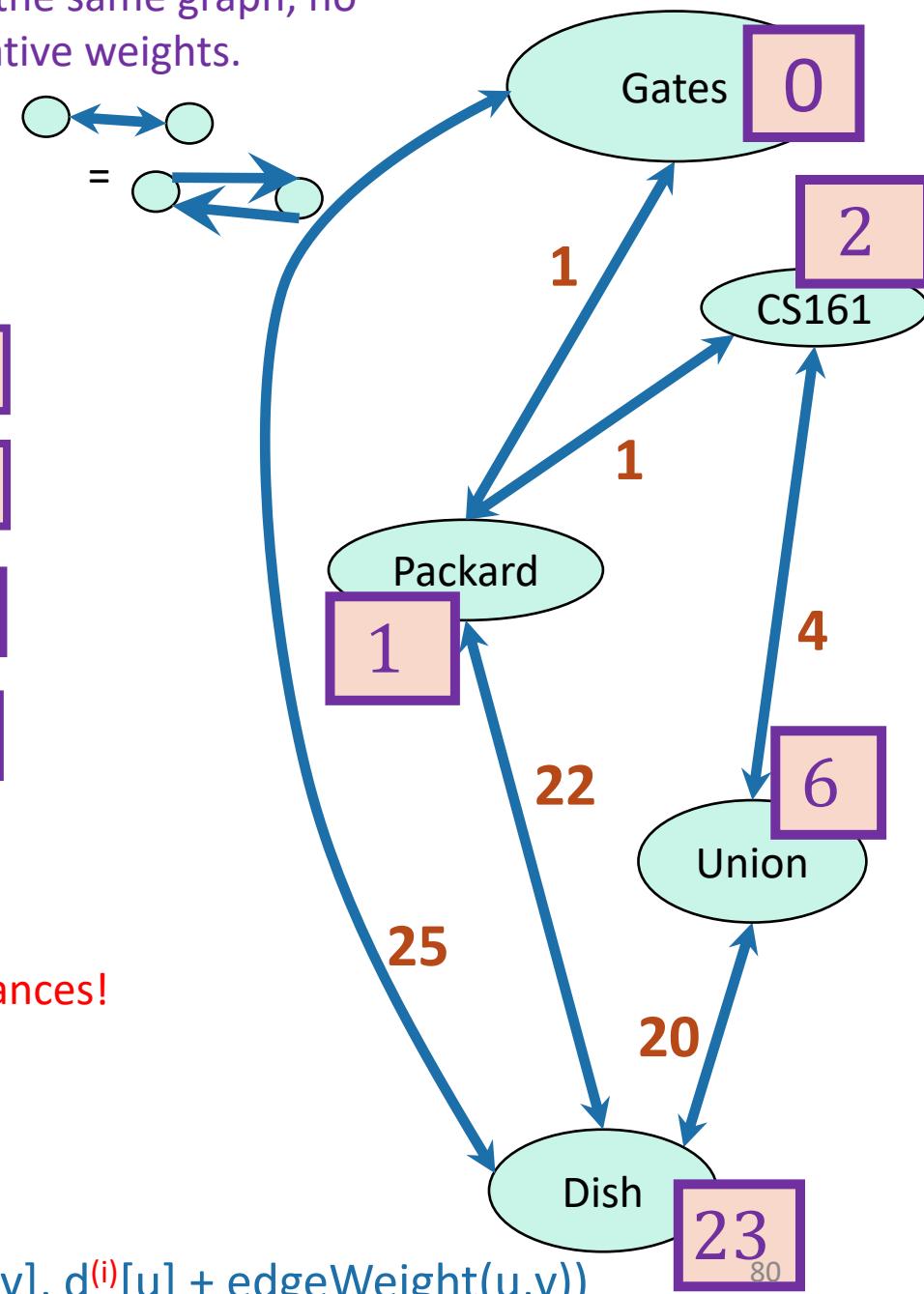
# Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	0	1	$\infty$	$\infty$	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$	0	1	2	6	23
$d^{(4)}$	0	1	2	6	23

These are the final distances!

Start with the same graph, no negative weights.



- For  $i=0, \dots, n-2$ :
  - For  $u$  in  $V$ :
    - For  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

# As usual

- Does it work?
  - Yes
  - Idea to the right.
  - (See hidden slides for details)
- Is it fast?
  - Not really...

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	0	1	$\infty$	$\infty$	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$	0	1	2	6	23
$d^{(4)}$	0	1	2	6	23

**Idea:** proof by induction.

**Inductive Hypothesis:**

$d^{(i)}[v]$  is equal to the cost of the shortest path between  $s$  and  $v$  **with at most  $i$  edges**.

**Conclusion:**

$d^{(n-1)}[v]$  is equal to the cost of the shortest simple path between  $s$  and  $v$ . **(Since all simple paths have at most  $n-1$  edges).**

# Proof by induction

- **Inductive Hypothesis:**

- After iteration  $i$ , for each  $v$ ,  $d^{(i)}[v]$  is equal to the cost of the shortest path between  $s$  and  $v$  **with at most  $i$  edges**.

- **Base case:**

- After iteration 0...



- **Inductive step:**

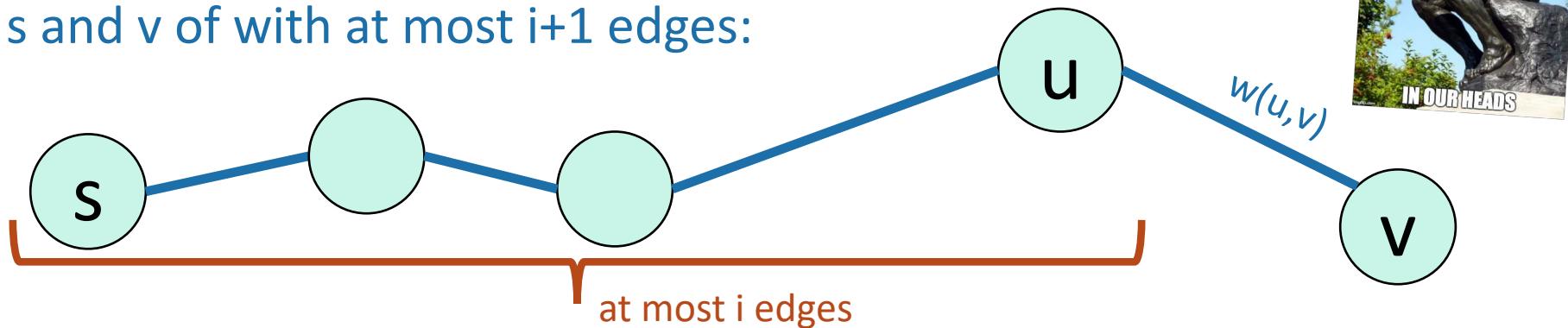
# Skipped in class

# Inductive step

**Hypothesis:** After iteration  $i$ , for each  $v$ ,  $d^{(i)}[v]$  is equal to the cost of the shortest path between  $s$  and  $v$  with at most  $i$  edges.

- Suppose the inductive hypothesis holds for  $i$ .
- We want to establish it for  $i+1$ .

Say this is the shortest path between  $s$  and  $v$  of with at most  $i+1$  edges:



- By induction,  $d^{(i)}[u]$  is the cost of a shortest path between  $s$  and  $u$  of  $i$  edges.
- By setup,  $d^{(i)}[u] + w(u,v)$  is the cost of a shortest path between  $s$  and  $v$  of  $i+1$  edges.
- In the  $i+1$ 'st iteration, we ensure  $d^{(i+1)}[v] \leq d^{(i)}[u] + w(u,v)$ .
- So  $d^{(i+1)}[v] \leq$  cost of shortest path between  $s$  and  $v$  with  $i+1$  edges.
- But  $d^{(i+1)}[v] =$  cost of a particular path of at most  $i+1$  edges  $\geq$  cost of shortest path.
- So  $d[v] =$  cost of shortest path with at most  $i+1$  edges.

# Proof by induction

- **Inductive Hypothesis:**

- After iteration  $i$ , for each  $v$ ,  $d^{(i)}[v]$  is equal to the cost of the shortest path between  $s$  and  $v$  **of length at most  $i$  edges**.

- **Base case:**

- After iteration 0...



- **Inductive step:**



- **Conclusion:**

- After iteration  $n-1$ , for each  $v$ ,  $d[v]$  is equal to the cost of the shortest path between  $s$  and  $v$  **of length at most  $n-1$  edges**.
- **Aka,  $d[v] = d(s,v)$  for all  $v$  as long as there are no cycles!**

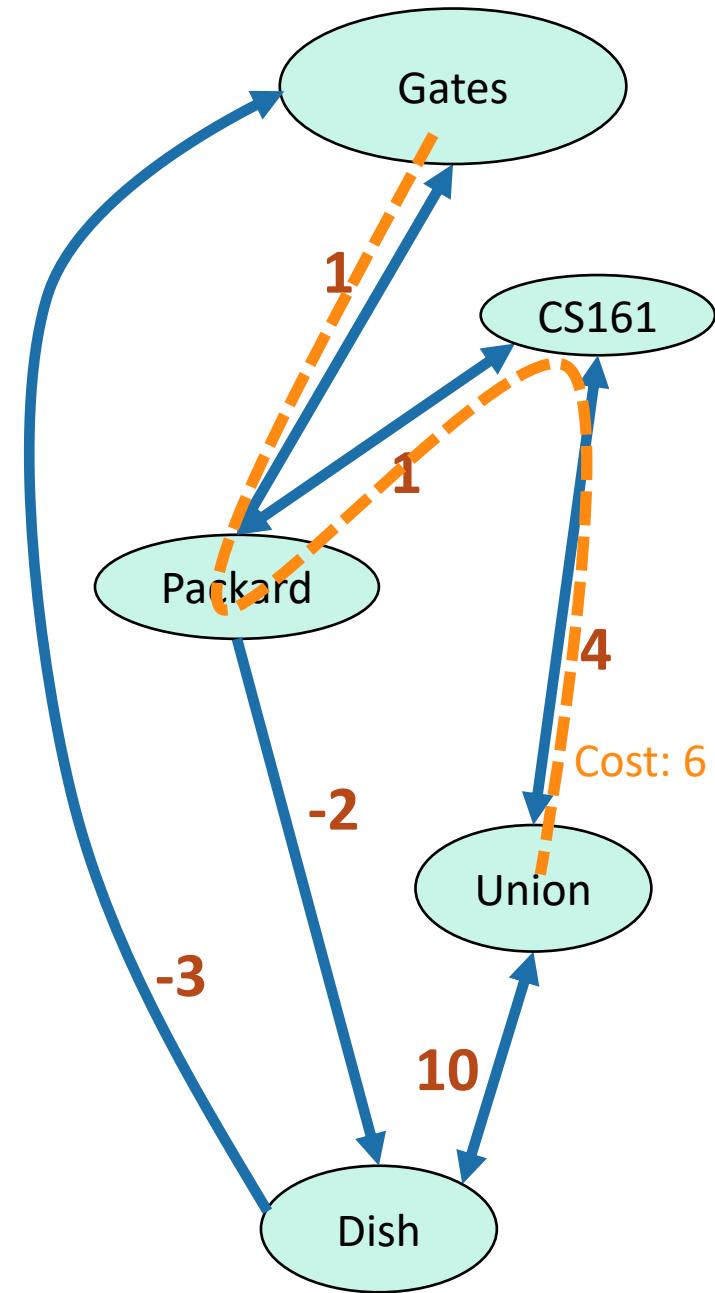


# Pros and cons of Bellman-Ford

- Running time:  $O(mn)$  running time
  - For each of  $n$  steps we update  $m$  edges
  - Slower than Dijkstra
- However, it's also more flexible in a few ways.
  - Can handle negative edges
  - If we constantly do these iterations, any changes in the network will eventually propagate through.

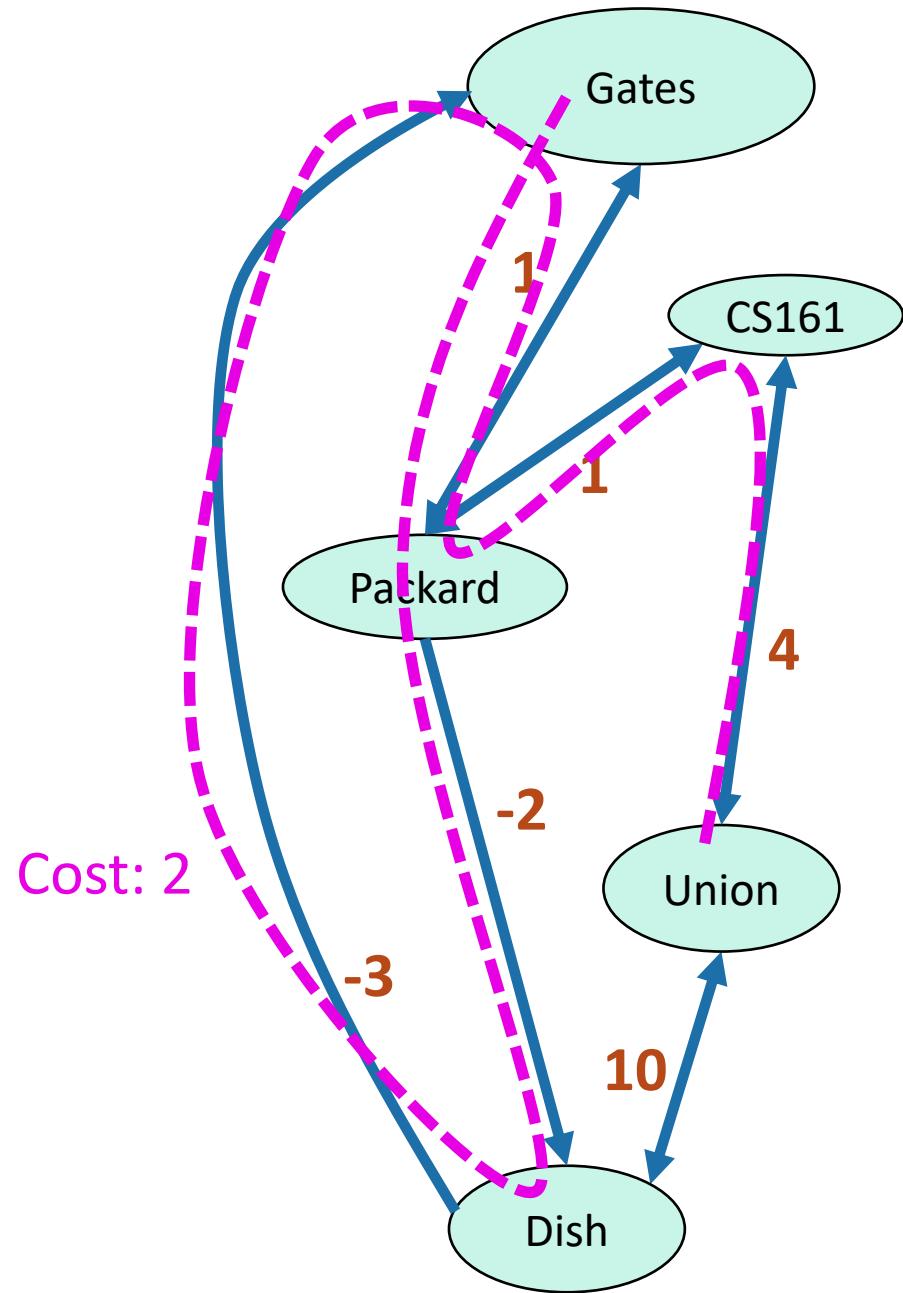
Wait a second...

- What is the shortest path from Gates to the Union?



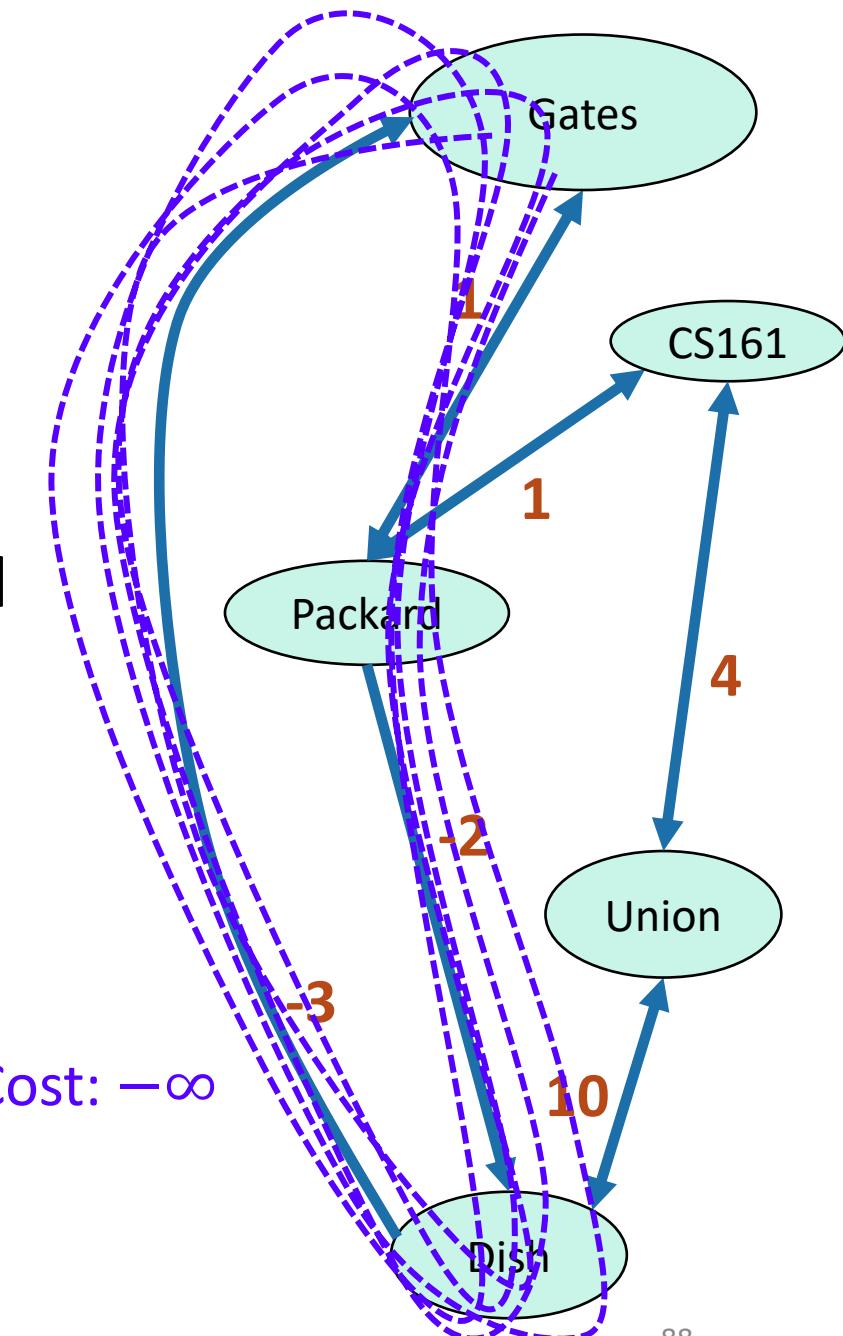
Wait a second...

- What is the shortest path from Gates to the Union?



# Negative edge weights?

- What is the shortest path from Gates to the Union?
- Shortest paths aren't defined if there are negative cycles!



# Bellman-Ford and negative edge weights

- B-F works with negative edge weights...as long as there are not negative cycles.
  - A negative cycle is a path with the same start and end vertex whose cost is negative.
- However, B-F can **detect** negative cycles.

# Back to the correctness

- Does it work?
  - Yes
  - Idea to the right.

If there are negative cycles,  
then non-simple paths matter!

So the proof breaks for  
negative cycles.



	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	0	1	$\infty$	$\infty$	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$	0	1	2	6	23
$d^{(4)}$	0	1	2	6	23

Idea: proof by induction.

**Inductive Hypothesis:**

$d^{(i)}[v]$  is equal to the cost of the shortest path between  $s$  and  $v$  **with at most  $i$  edges**.

**Conclusion:**

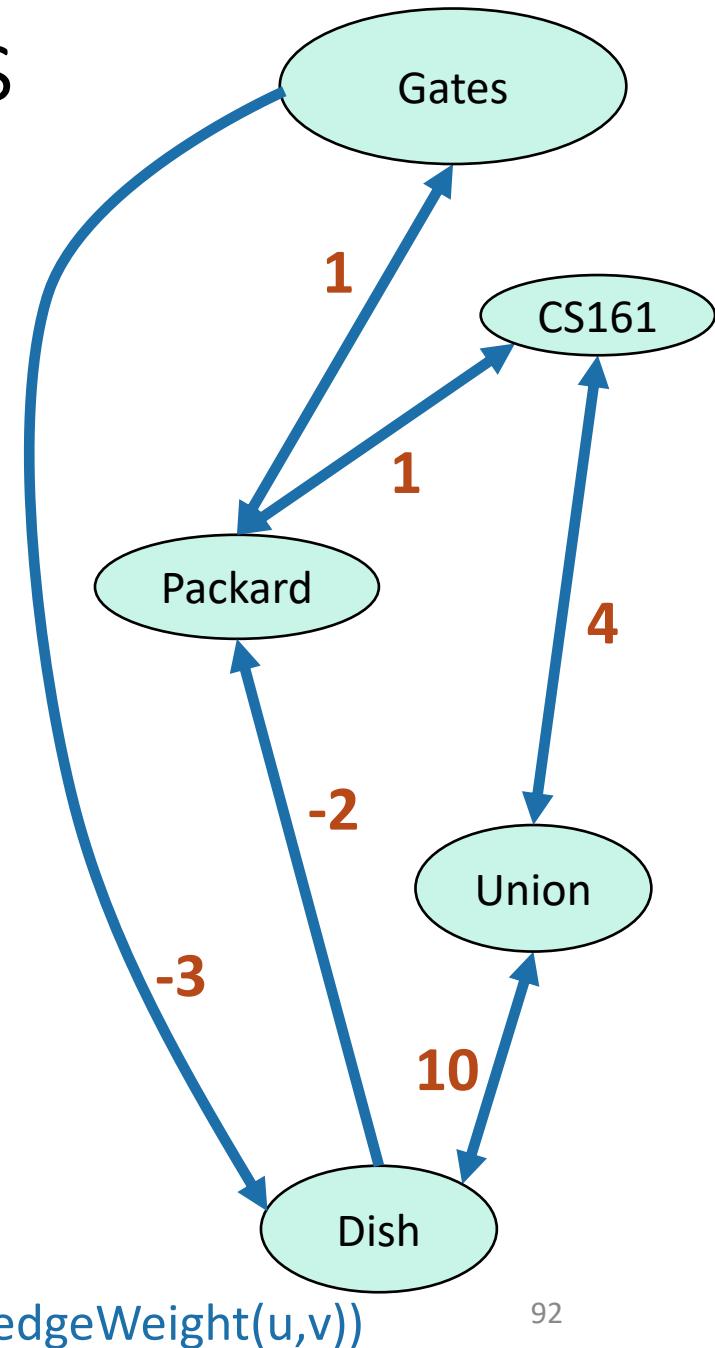
$d^{(n-1)}[v]$  is equal to the cost of the shortest simple path between  $s$  and  $v$ . (**Since all simple paths have at most  $n-1$  edges**).

# B-F with negative cycles

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	0	1	$\infty$	$\infty$	-3
$d^{(2)}$	0	-5	2	7	-3
$d^{(3)}$	-4	-5	-4	6	-3

This is not looking good!

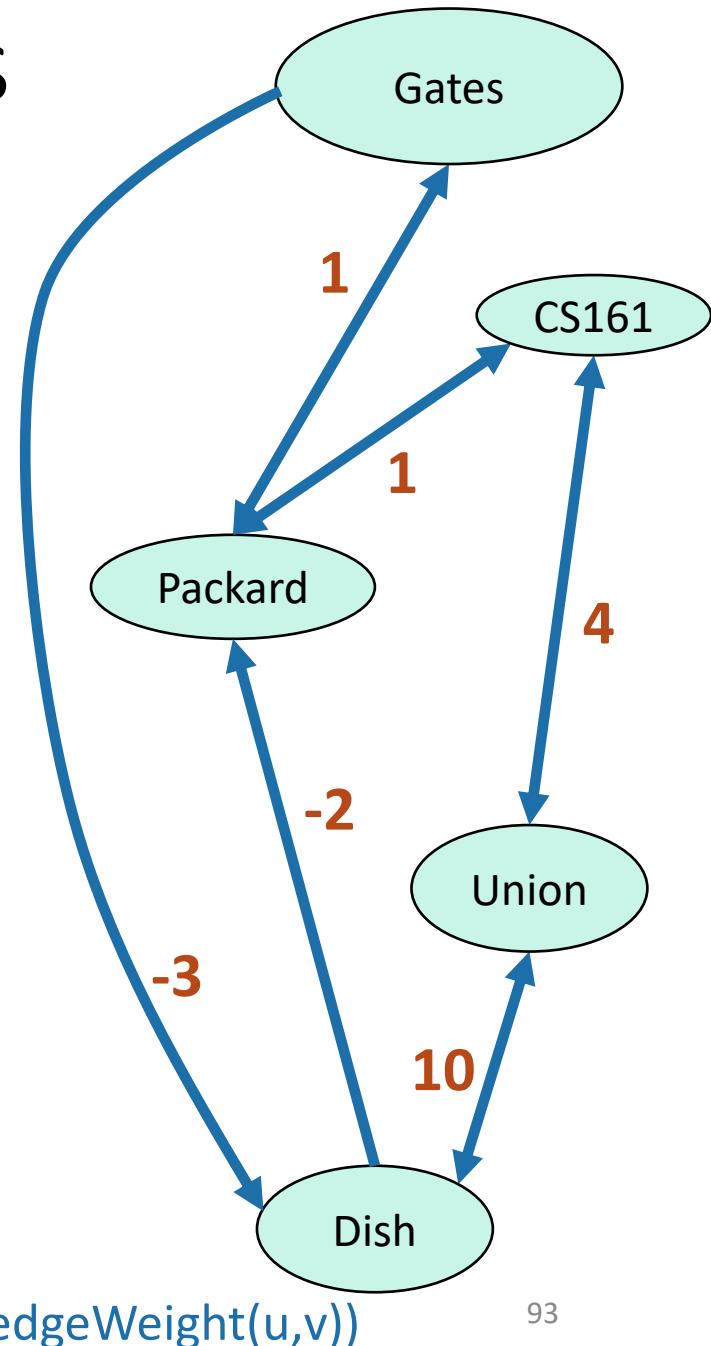
- For  $i=0, \dots, n-2$ :
  - For  $u$  in  $V$ :
    - For  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$



# B-F with negative cycles

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	$\infty$	$\infty$	$\infty$	$\infty$
$d^{(1)}$	0	1	$\infty$	$\infty$	-3
$d^{(2)}$	0	-5	2	7	-3
$d^{(3)}$	-4	-5	-4	6	-3
$d^{(4)}$	-4	-5	-4	6	-7
$d^{(5)}$	-4	-9	-4	3	-7

But we can tell that it's not looking good:



Some stuff changed!

- For  $i=0, \dots, n-1$ :
  - For  $u$  in  $V$ :
    - For  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

# How Bellman-Ford deals with negative cycles

- If there are no negative cycles:
  - Everything works as it should.
  - The algorithm stabilizes after  $n-1$  rounds.
  - Note: Negative **edges** are okay!!
- If there are negative cycles:
  - Not everything works as it should...
    - Note: it couldn't possibly work, since shortest paths aren't well-defined if there are negative cycles.
  - The  $d[v]$  values will keep changing.
- Solution:
  - Go one round more and see if things change.
    - If so, return NEGATIVE CYCLE ☹
  - (Pseudocode on skipped slide)

# Bellman-Ford algorithm

SLIDE SKIPPED IN CLASS

**Bellman-Ford\*(G,s):**

- $d^{(0)}[v] = \infty$  for all  $v$  in  $V$
- $d^{(0)}[s] = 0$
- **For**  $i=0, \dots, n-1$ :
  - **For**  $u$  in  $V$ :
    - **For**  $v$  in  $u.\text{neighbors}$ :
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
  - If  $d^{(n-1)} \neq d^{(n)}$  :
    - **Return NEGATIVE CYCLE** ☹
  - Otherwise,  $\text{dist}(s,v) = d^{(n-1)}[v]$

# Summary

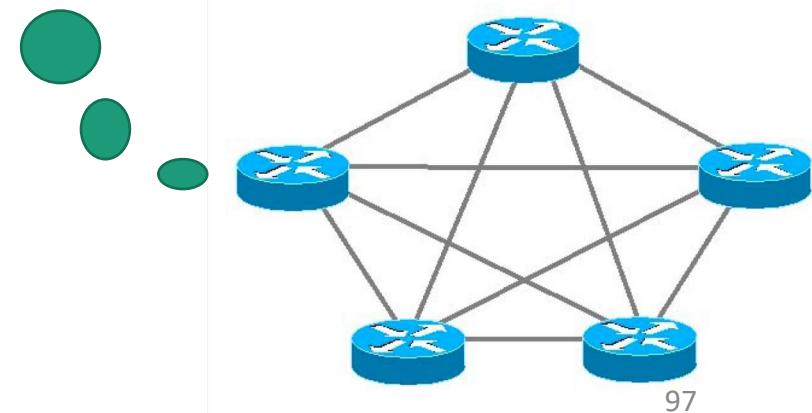
It's okay if that went by fast, we'll come back to Bellman-Ford

- The Bellman-Ford algorithm:
  - Finds shortest paths in weighted graphs with negative edge weights
  - runs in time  $O(nm)$  on a graph  $G$  with  $n$  vertices and  $m$  edges.
- If there are no negative cycles in  $G$ :
  - the BF algorithm terminates with  $d^{(n-1)}[v] = d(s,v)$ .
- If there are negative cycles in  $G$ :
  - the BF algorithm returns **negative cycle**.

# Bellman-Ford is also used in practice.

- eg, Routing Information Protocol (RIP) uses something like Bellman-Ford.
  - Older protocol, not used as much anymore.
- Each router keeps a **table** of distances to every other router.
- Periodically we do a Bellman-Ford update.
- This means that if there are changes in the network, this will propagate. (maybe slowly...)

Destination	Cost to get there	Send to whom?
172.16.1.0	34	172.16.1.1
10.20.40.1	10	192.168.1.2
10.155.120.1	9	10.13.50.0



# Recap: shortest paths

- **BFS:**
  - (+)  $O(n+m)$
  - (-) only unweighted graphs
- **Dijkstra's algorithm:**
  - (+) weighted graphs
  - (+)  $O(n\log(n) + m)$  if you implement it right.
  - (-) no negative edge weights
  - (-) very “centralized” (need to keep track of all the vertices to know which to update).
- **The Bellman-Ford algorithm:**
  - (+) weighted graphs, even with negative weights
  - (+) can be done in a distributed fashion, every vertex using only information from its neighbors.
  - (-)  $O(nm)$

# Next Time

- Midterm!

## Before next time

- Study for midterm! ☺