Lecture 11

Weighted Graphs: Dijkstra and Bellman-Ford

Announcements

- HW5 is out today!
- HW4 due FRIDAY.

- Lost and found from midterm:
  - Pencil case with a charger in it
  - Tote bag with a book in it
  - Email marykw@stanford.edu if either are yours.

Ed Heroes!

Top hearted

Name	Hearts
Shubham Anand Jain STAFF	157
Kevin Long Su STAFF	84
Rishu Garg STAFF	73
Jadon G	73
Monica H	68
Ingrid N	51
Gunnar H	50
Ruiqi Wang STAFF	48
Zach Wi	46
Aditi T	45

Top endorsed

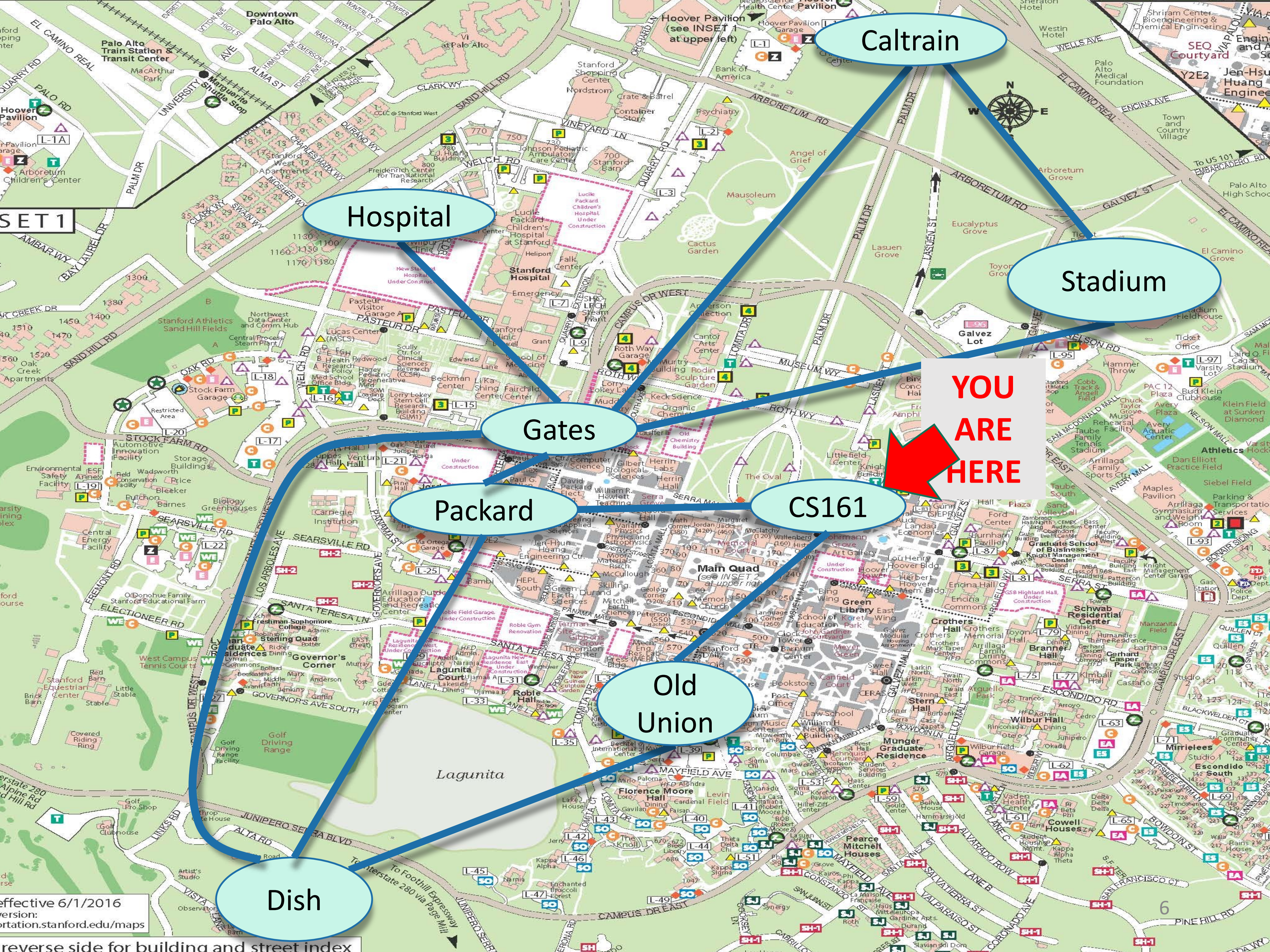
Name	Endorsements
Jadon G	79
Aditi T	36
Zach W	12
Thanawan A	11
Anna M	9
Giulia S	8
Maxton H	7
Rohan B	7
Claire M	6
Yasmine A	3

Previous two lectures

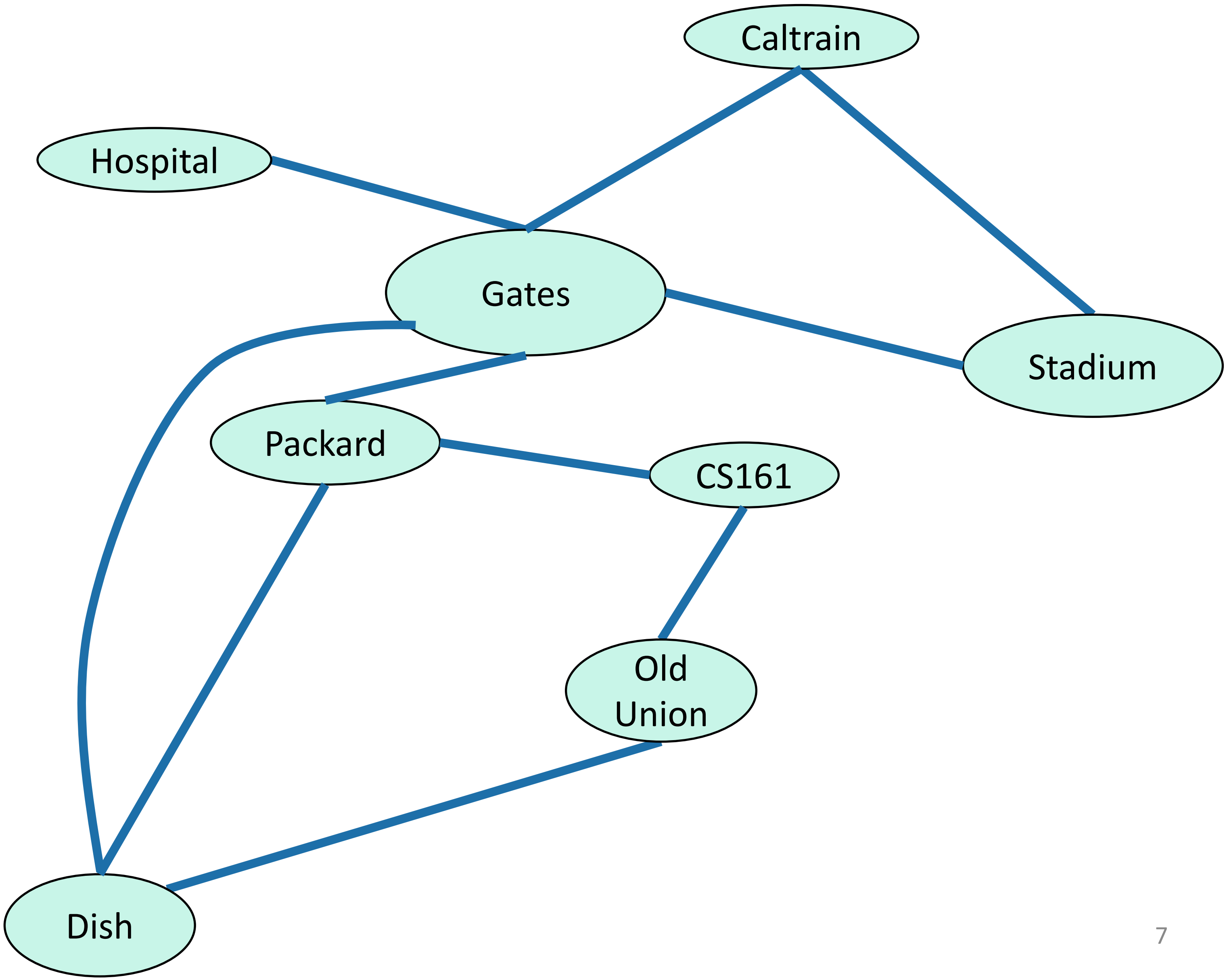
- Graphs!
- DFS
  - Topological Sorting
  - Strongly Connected Components
- BFS
  - Shortest Paths in unweighted graphs

Today

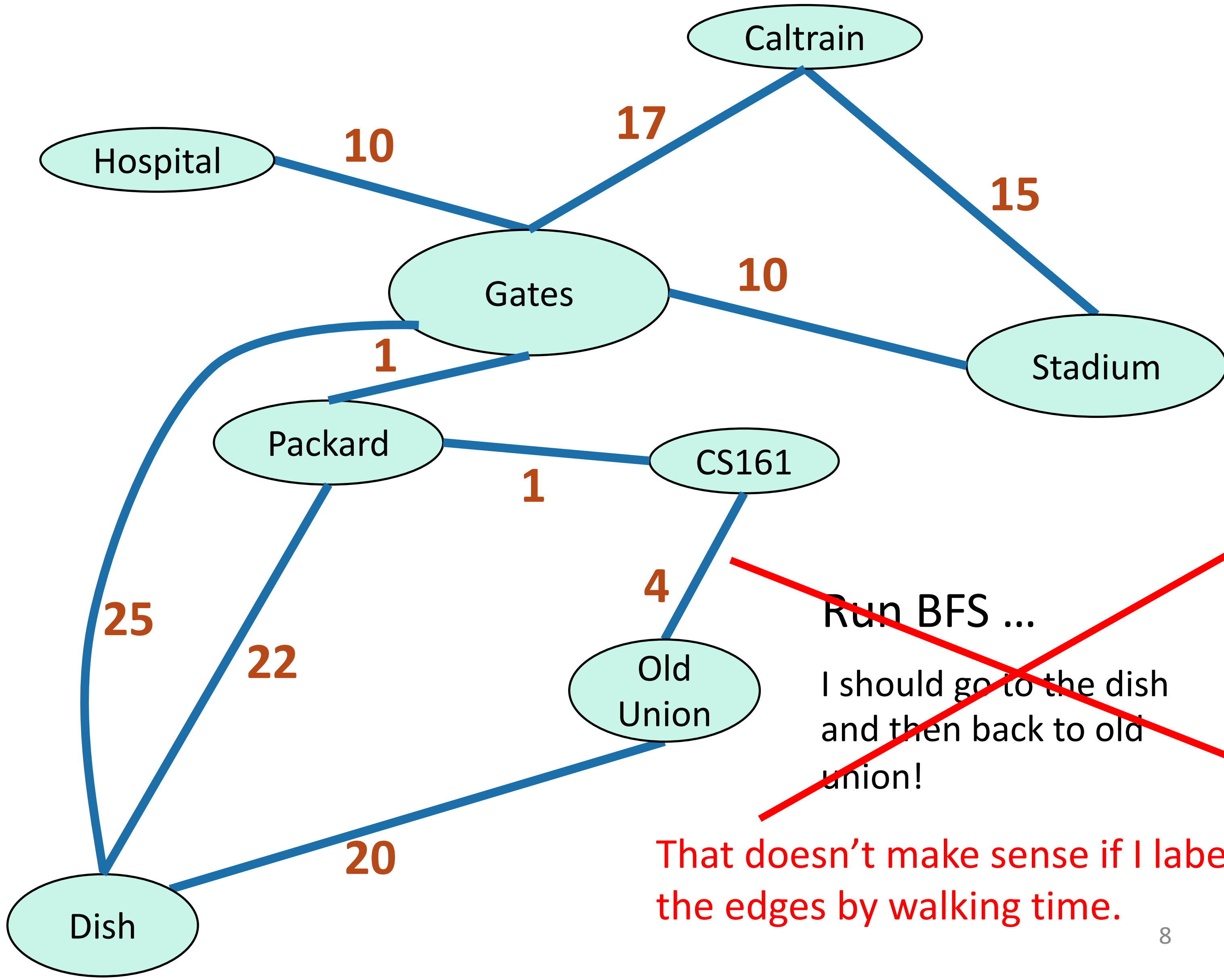
- What if the graphs are weighted?
- Part 1: Dijkstra!
  - This will take most of today's class
- Part 2: Bellman-Ford!
  - Real quick at the end **if we have time!**
  - We'll come back to Bellman-Ford in more detail, so today is just a taste.


Caltrain
Hospital
Stadium
YOU ARE HERE
Gates
Packard
CS161
Old Union
Dish
Lagunita
Main Quad

Just the graph



Shortest path from Gates to Old Union?



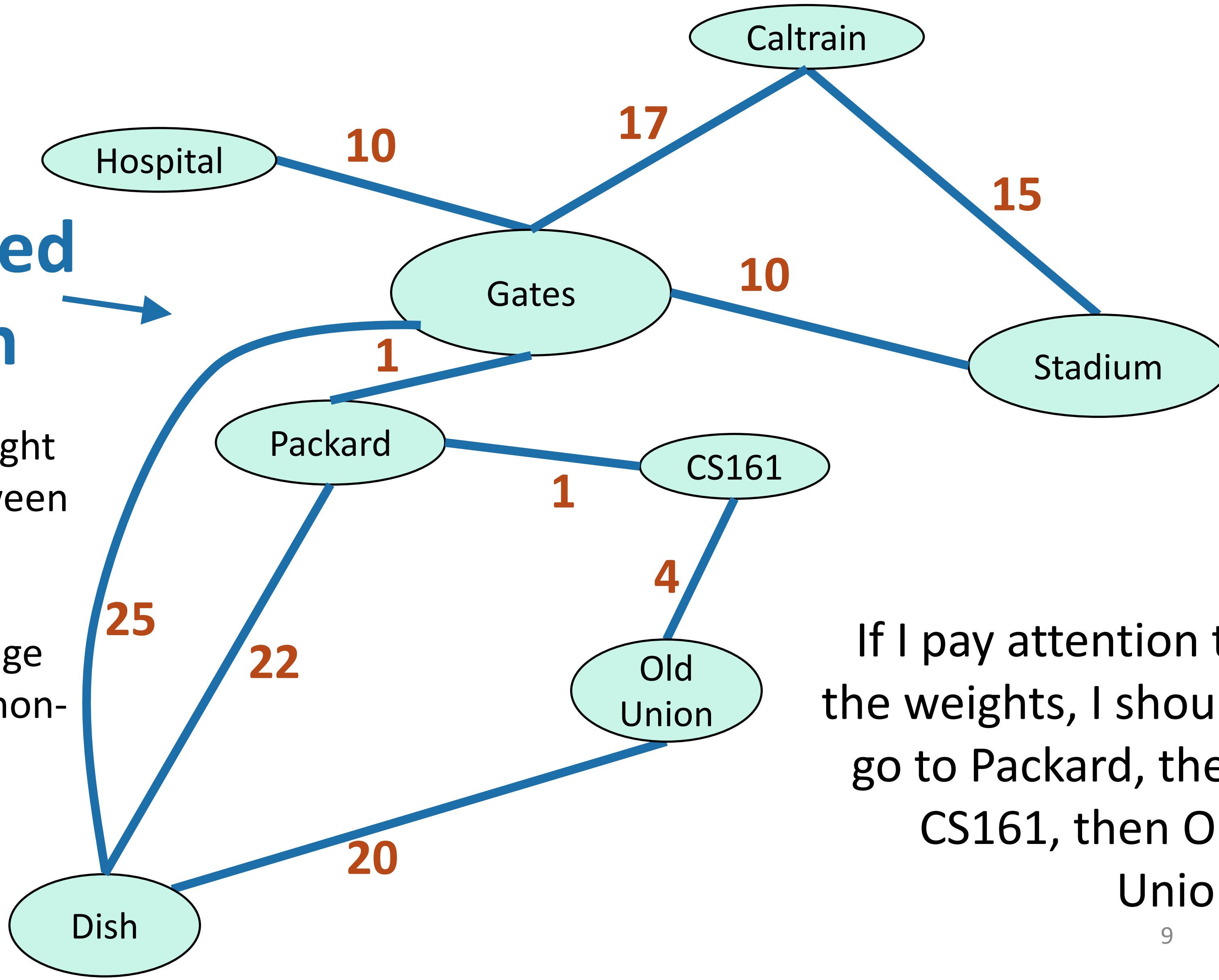
Run BFS …

I should go to the dish and then back to old union!

That doesn't make sense if I label the edges by walking time.

Shortest path from Gates to Old Union?

weighted graph



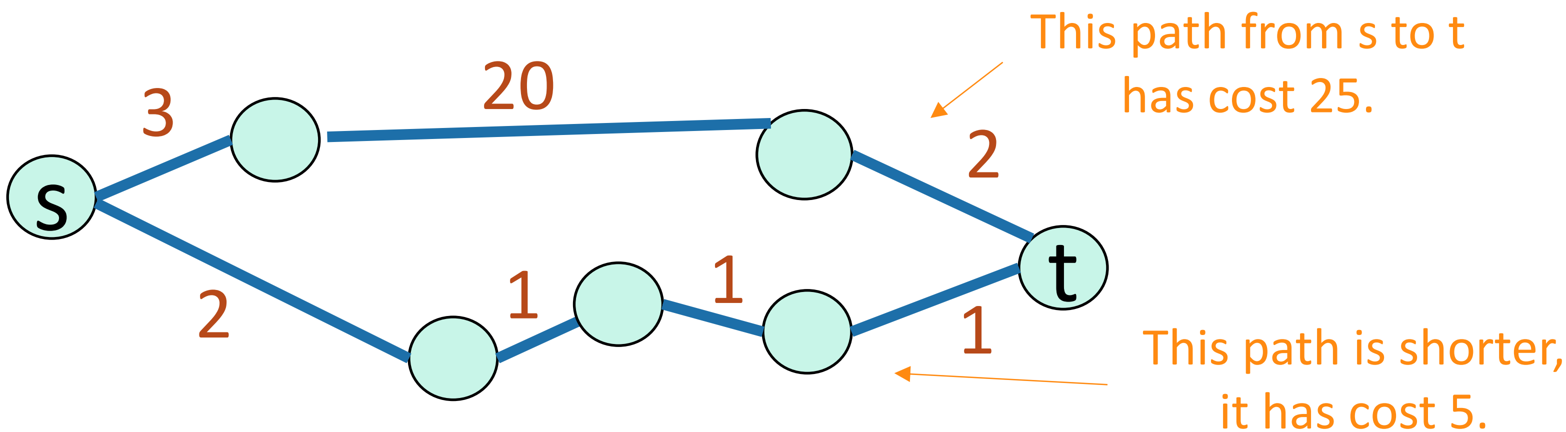
w(u,v) = weight of edge between u and v.

For now, edge weights are non-negative.

If I pay attention to the weights, I should go to Packard, then CS161, then Old Union.

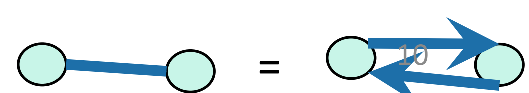
Shortest path problem

- 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.

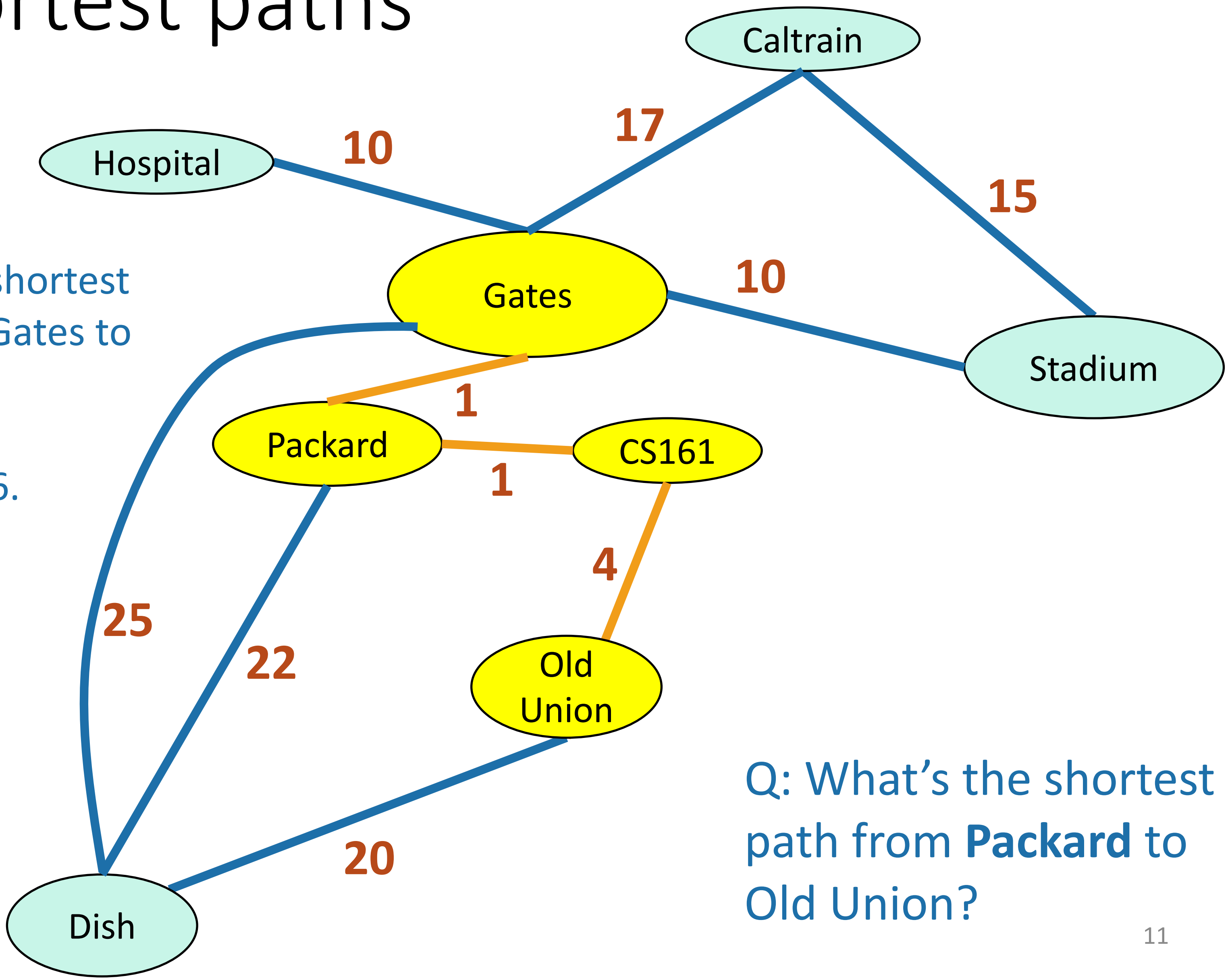


- The **distance** d(u,v) between two vertices u and v is the cost of the the shortest path between u and v.

Note: 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 Old Union.

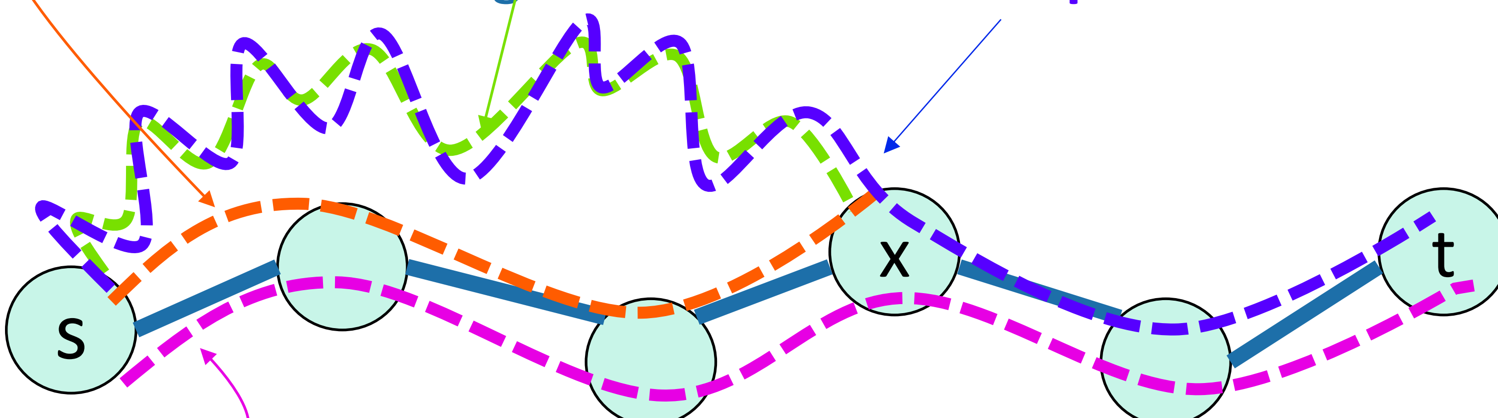
It has cost 6.

Q: What's the shortest path from **Packard** to Old Union?

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

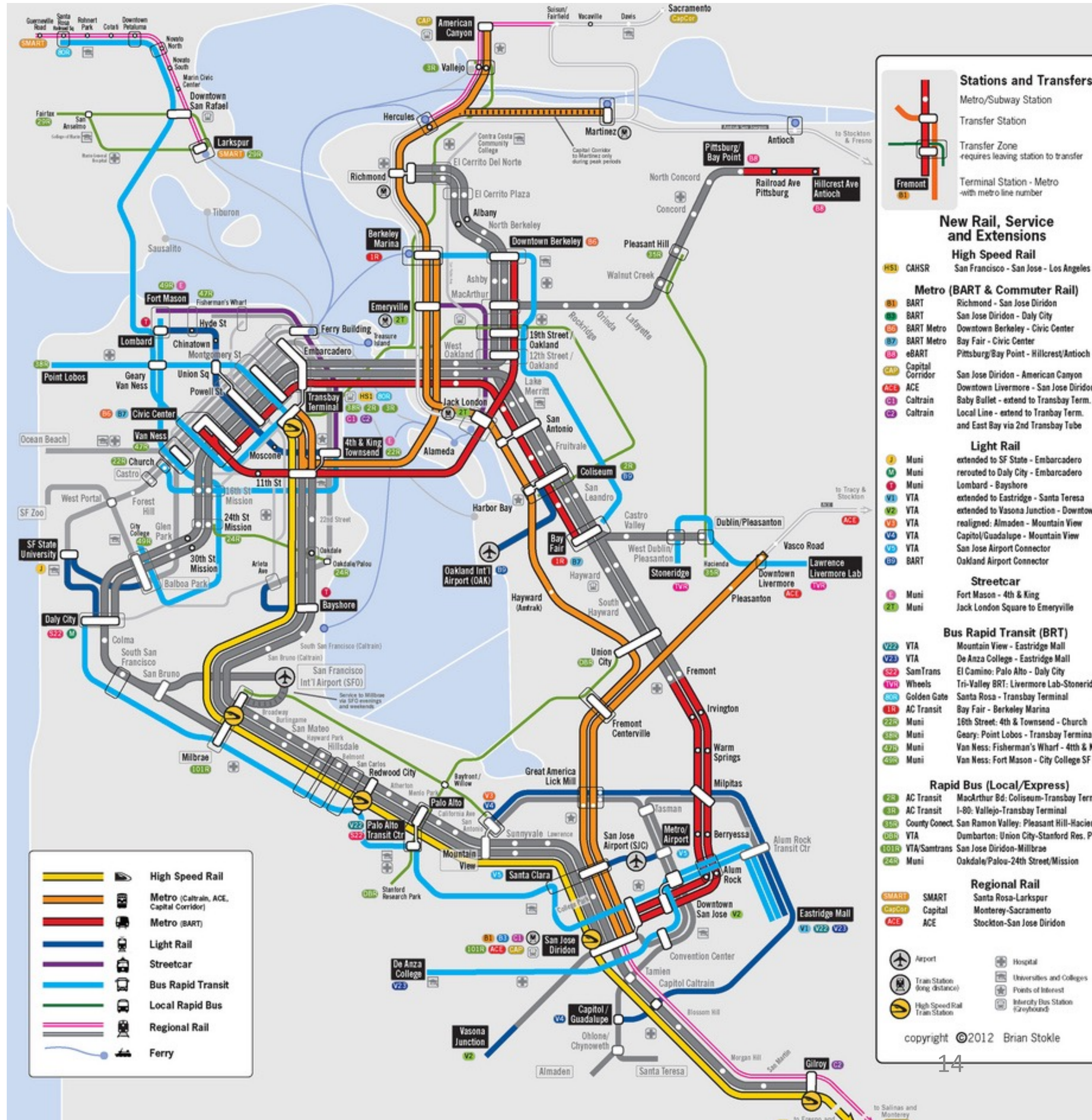
- What is the shortest path from one vertex (e.g. Gates) to all other vertices?

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

(The answer doesn't necessarily need to be stored as a table – how this information is represented will depend on the application)

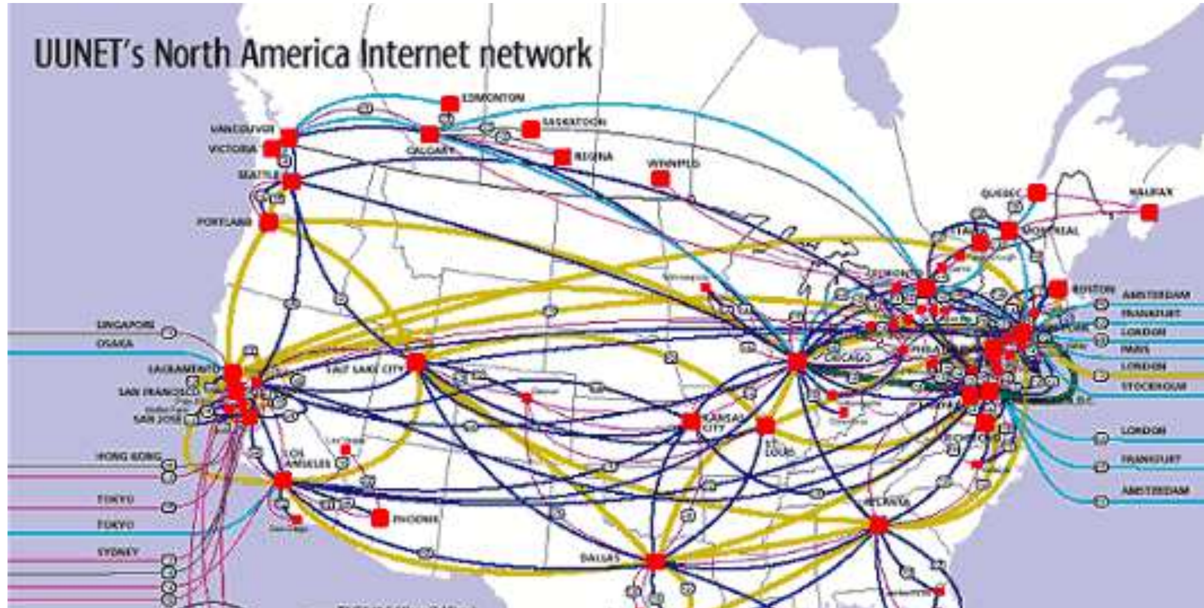
Example

- **"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.

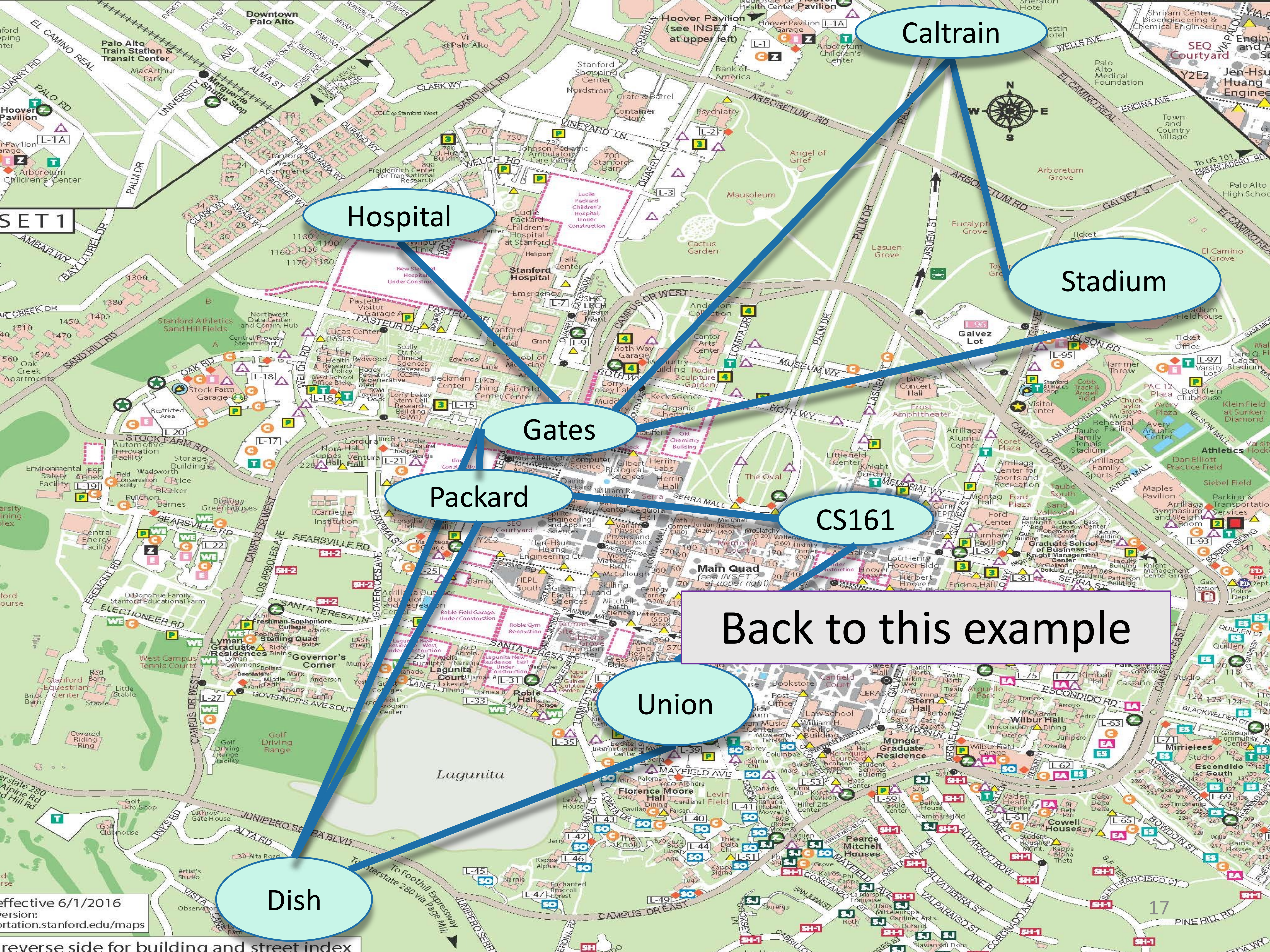


Example

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



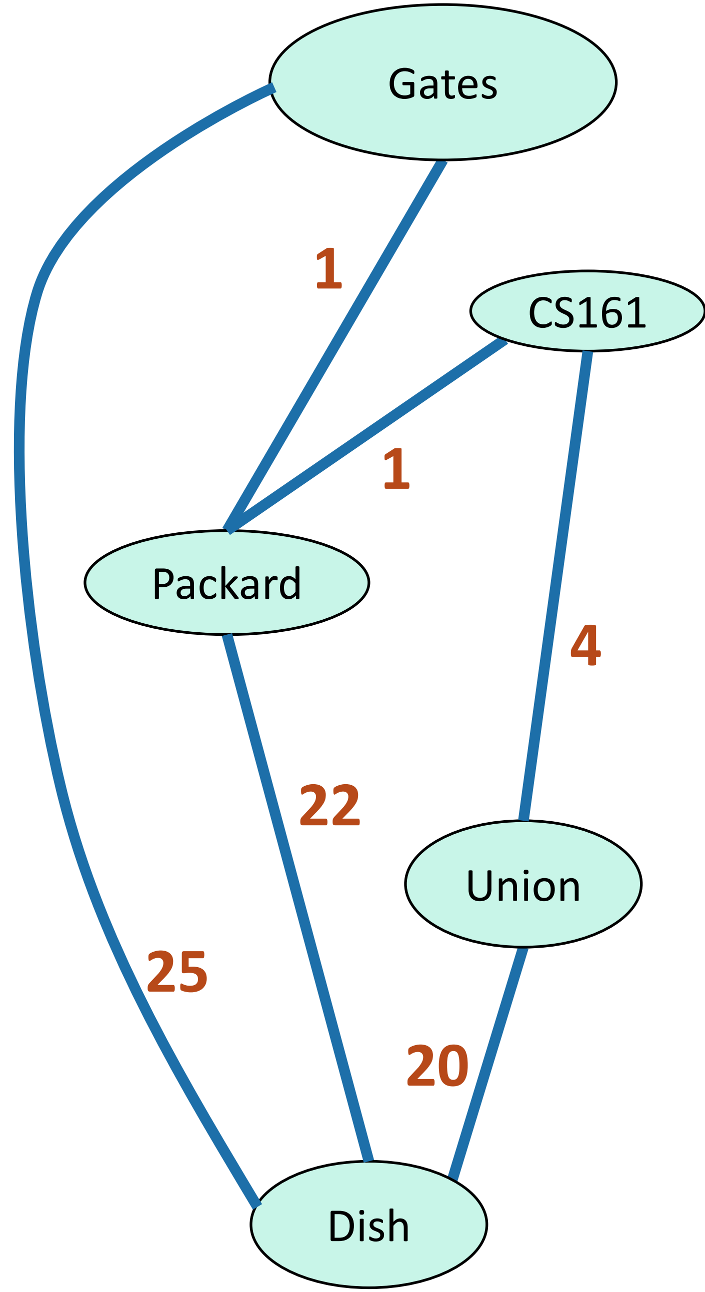

```
DN0a22a0e3:~ mary$ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1  [AS0] 10.34.160.2 (10.34.160.2)  38.168 ms  31.272 ms  28.841 ms
 2  [AS0] cwa-vrtr.sunet (10.21.196.28)  33.769 ms  28.245 ms  24.373 ms
 3  [AS32] 171.66.2.229 (171.66.2.229)  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  17
 5  [AS2152] hpr-svl-hpr2--stan-ge.cenic.net (137.164.27.161)  22.129 ms  4.
 6  [AS2152] hpr-lax-hpr3--svl-hpr3-100ge.cenic.net (137.164.25.73)  12.125
 7  [AS2152] hpr-i2--lax-hpr2-r&e.cenic.net (137.164.26.201)  40.174 ms  38.
 8  [AS0] et-4-0-0.4079.sdn-sw.lasv.net.internet2.edu (162.252.70.28)  46.57
 9  [AS0] et-5-1-0.4079.rtsw.salt.net.internet2.edu (162.252.70.31)  30.424
10  [AS0] et-4-0-0.4079.sdn-sw.denv.net.internet2.edu (162.252.70.8)  47.454
11  [AS0] et-4-1-0.4079.rtsw.kans.net.internet2.edu (162.252.70.11)  70.825
12  [AS0] et-4-1-0.4070.rtsw.chic.net.internet2.edu (198.71.47.206)  77.937
13  [AS0] et-0-1-0.4079.sdn-sw.ashb.net.internet2.edu (162.252.70.60)  77.68
14  [AS0] et-4-1-0.4079.rtsw.wash.net.internet2.edu (162.252.70.65)  71.565
15  [AS21320] internet2-gw.mx1.lon.uk.geant.net (62.40.124.44)  154.926 ms
16  [AS21320] ae0.mx1.lon2.uk.geant.net (62.40.98.79)  146.565 ms  146.604 m
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 m
19  [AS21320] swice1-100ge-0-3-0-1.switch.ch (62.40.124.22)  162.068 ms  160.
20  [AS559] swizh1-100ge-0-1-0-1.switch.ch (130.59.36.94)  165.824 ms  164.2
21  [AS559] swiez3-100ge-0-1-0-4.switch.ch (130.59.38.109)  164.269 ms  164.
22  [AS559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1)  164.082 ms  1
23  [AS559] rou-fw-rz-rz-gw.ethz.ch (192.33.92.169)  164.773 ms  165.193 ms
```



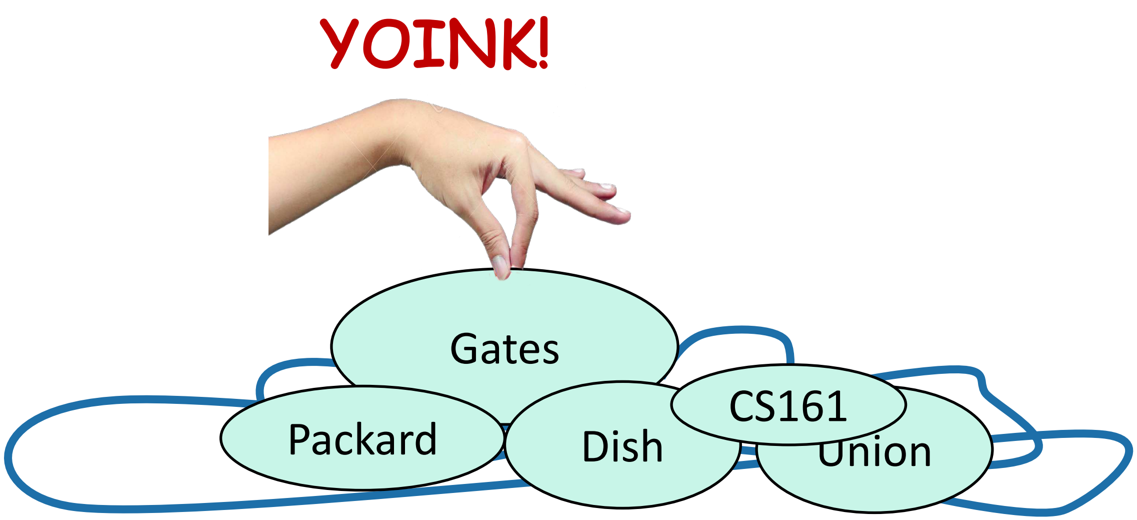
Back to this example

Dijkstra’s algorithm

- Finds shortest paths from Gates to everywhere else.

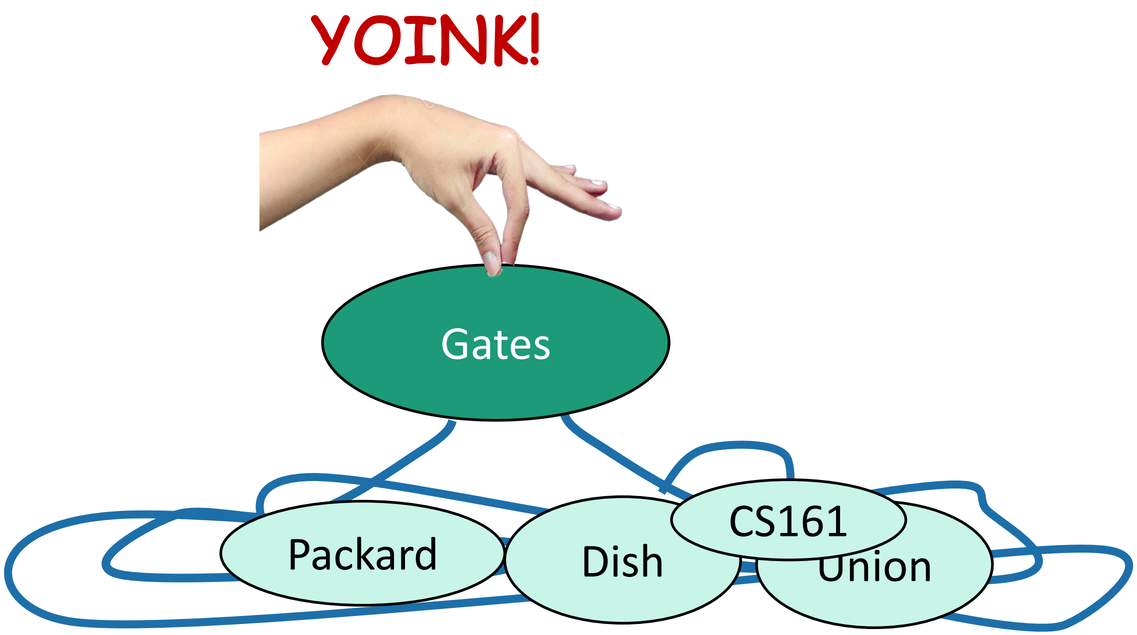


Dijkstra
intuition

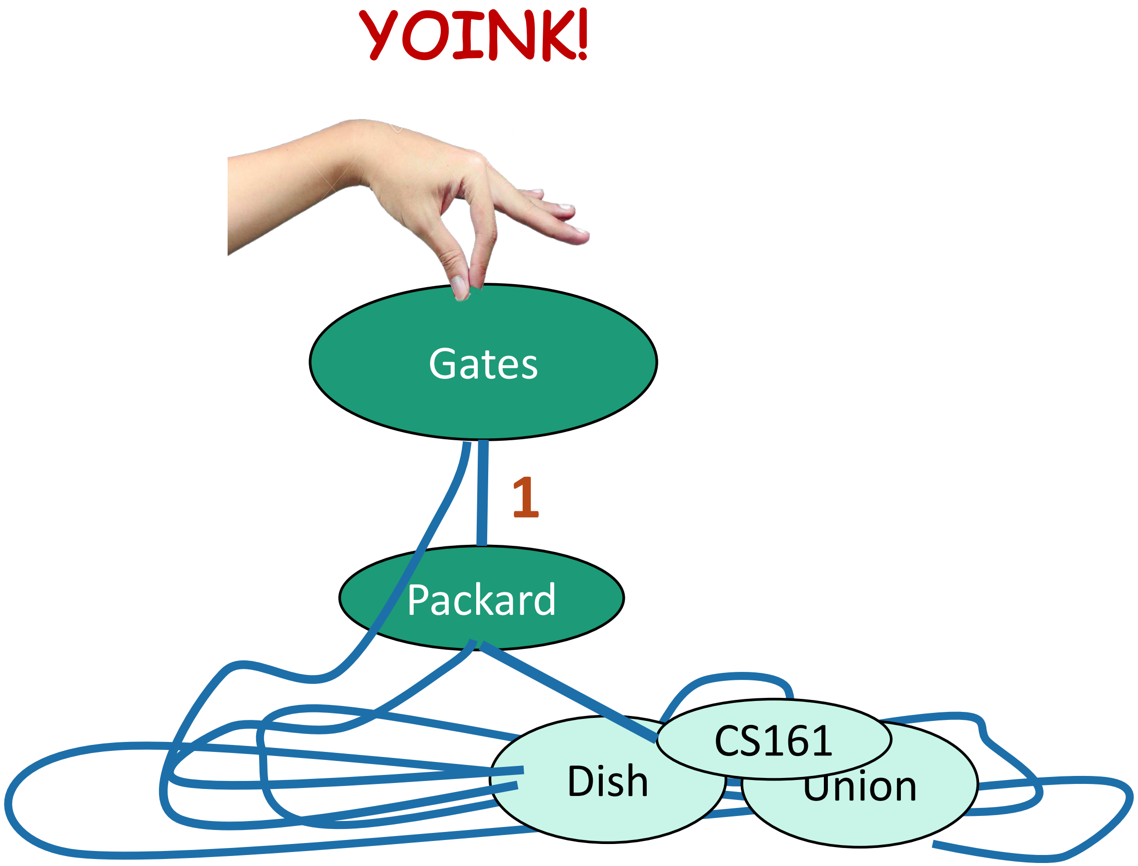


Dijkstra
intuition

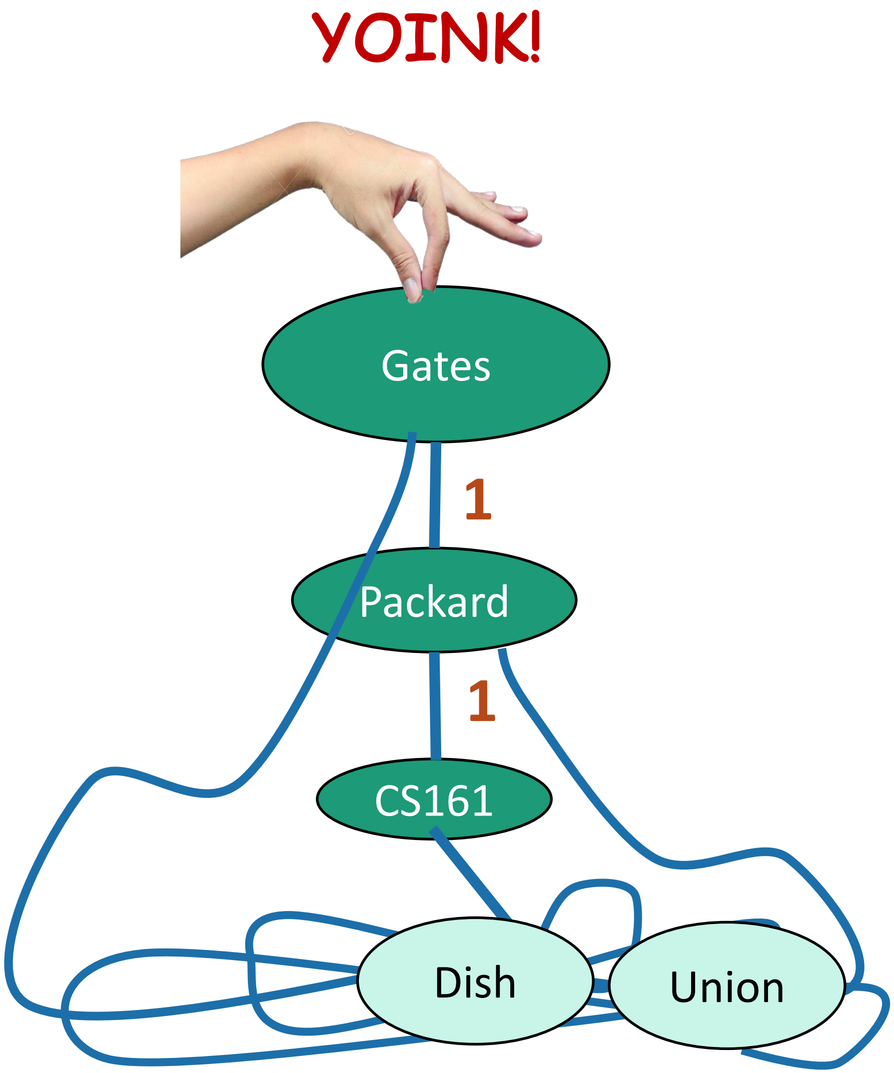
A vertex is done when it’s not on the ground anymore.



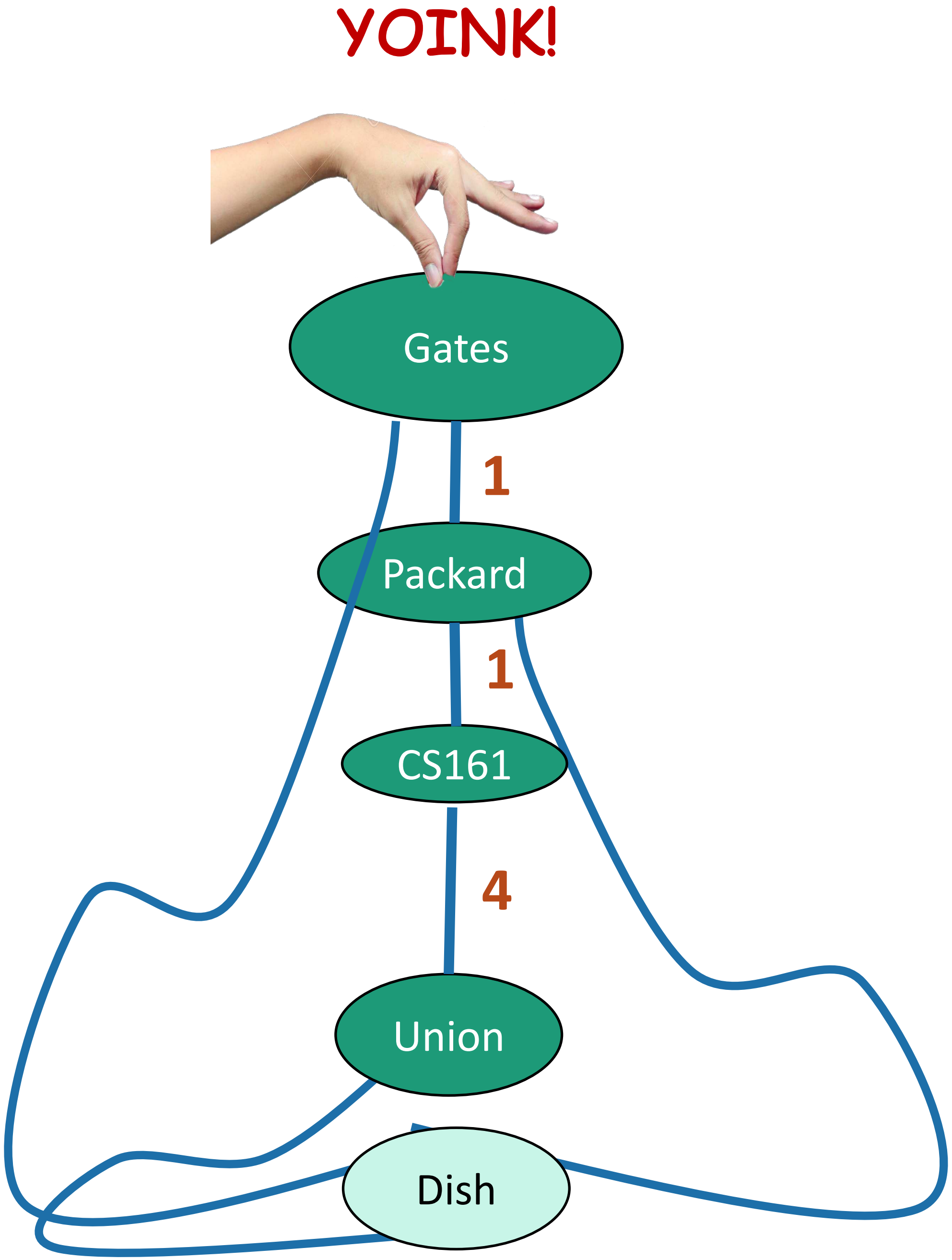
Dijkstra
intuition



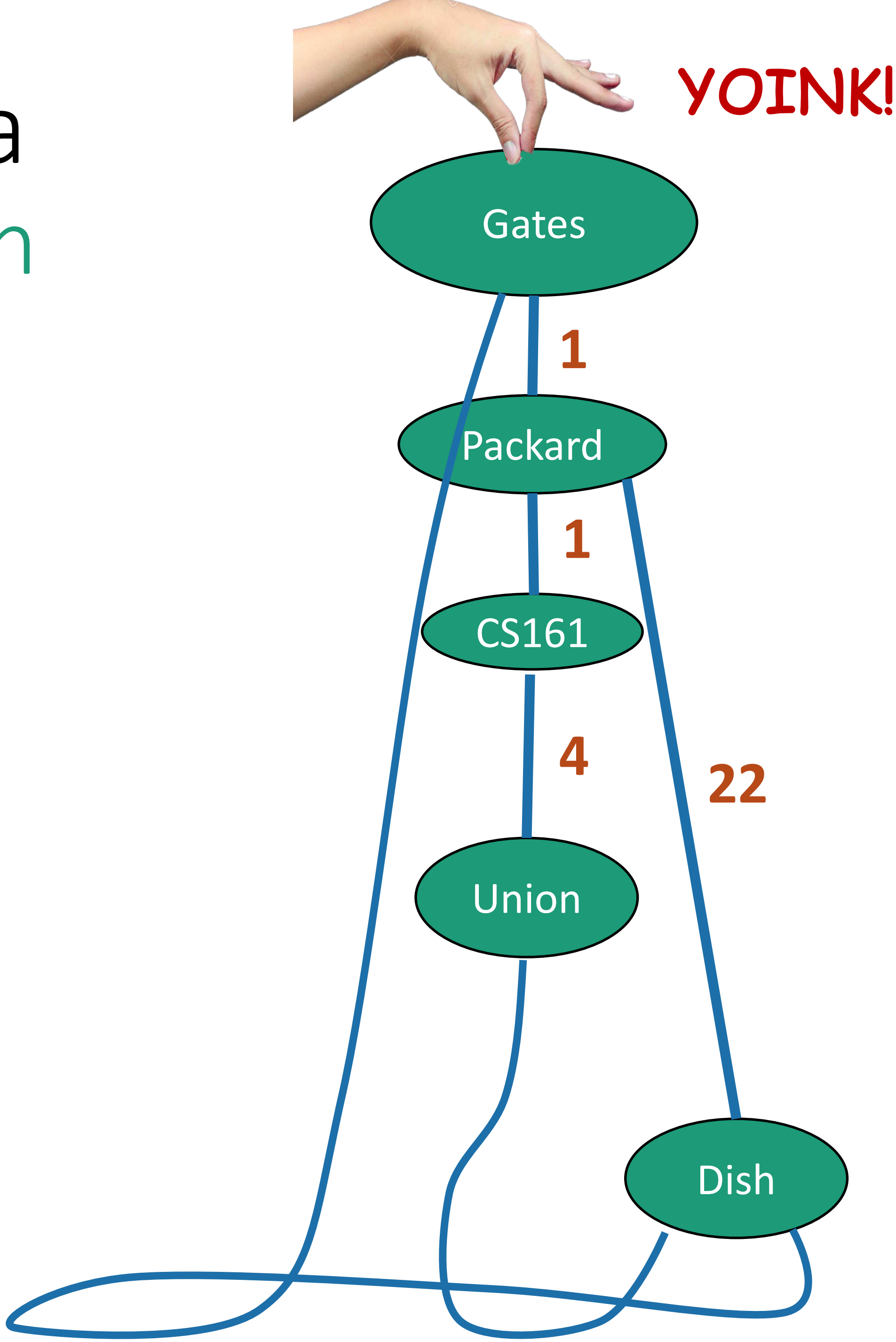
Dijkstra
intuition



Dijkstra
intuition



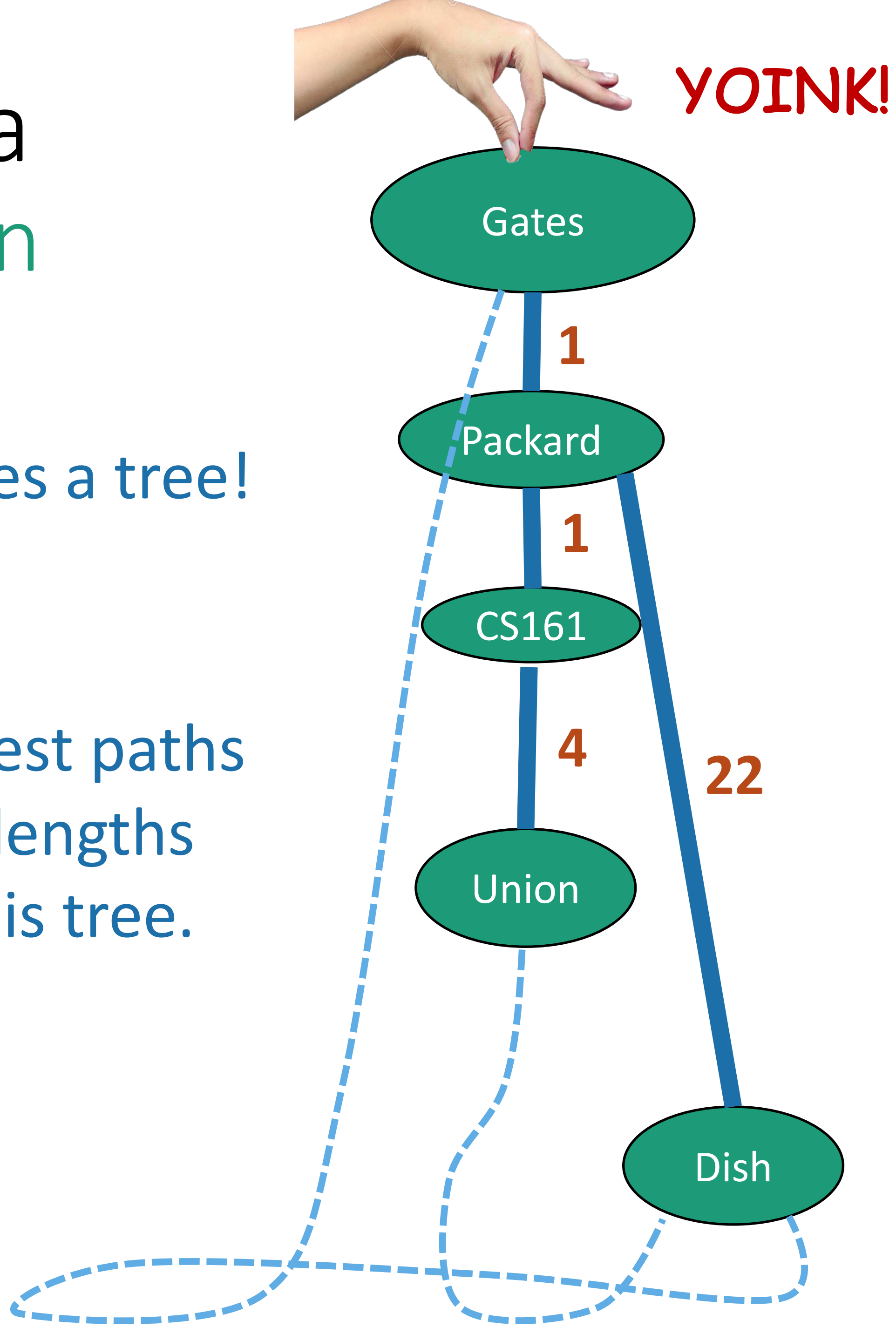
Dijkstra
intuition



Dijkstra
intuition

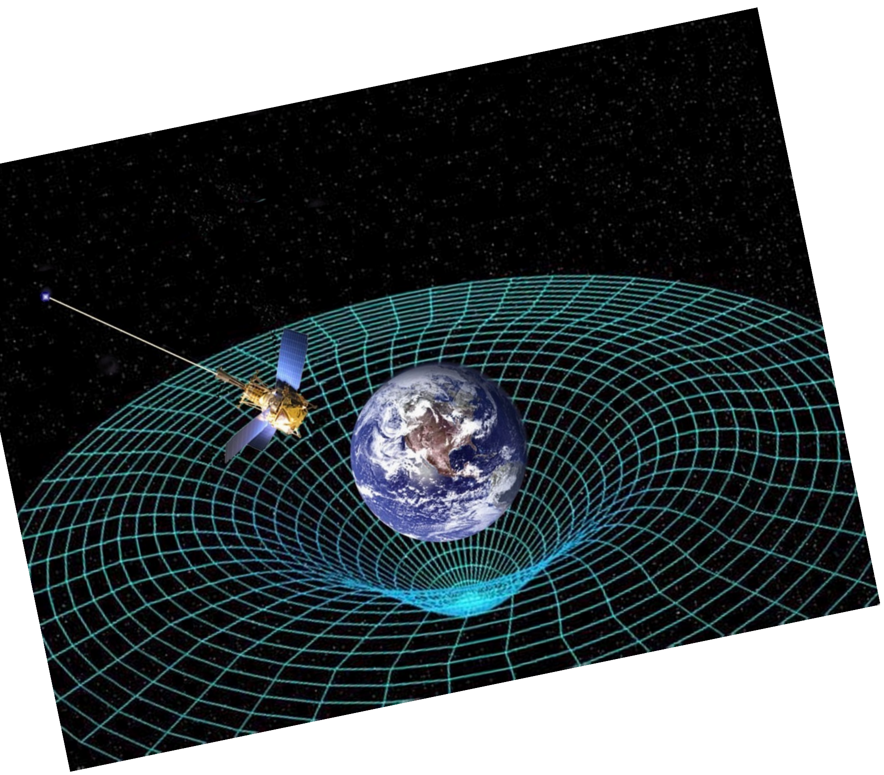
This creates a tree!

The shortest paths are the lengths along this tree.



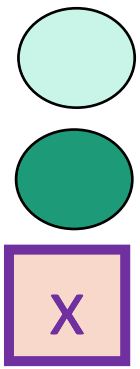
How do we actually implement this?

- **Without** string and gravity?



Dijkstra by example

How far is a node from Gates?



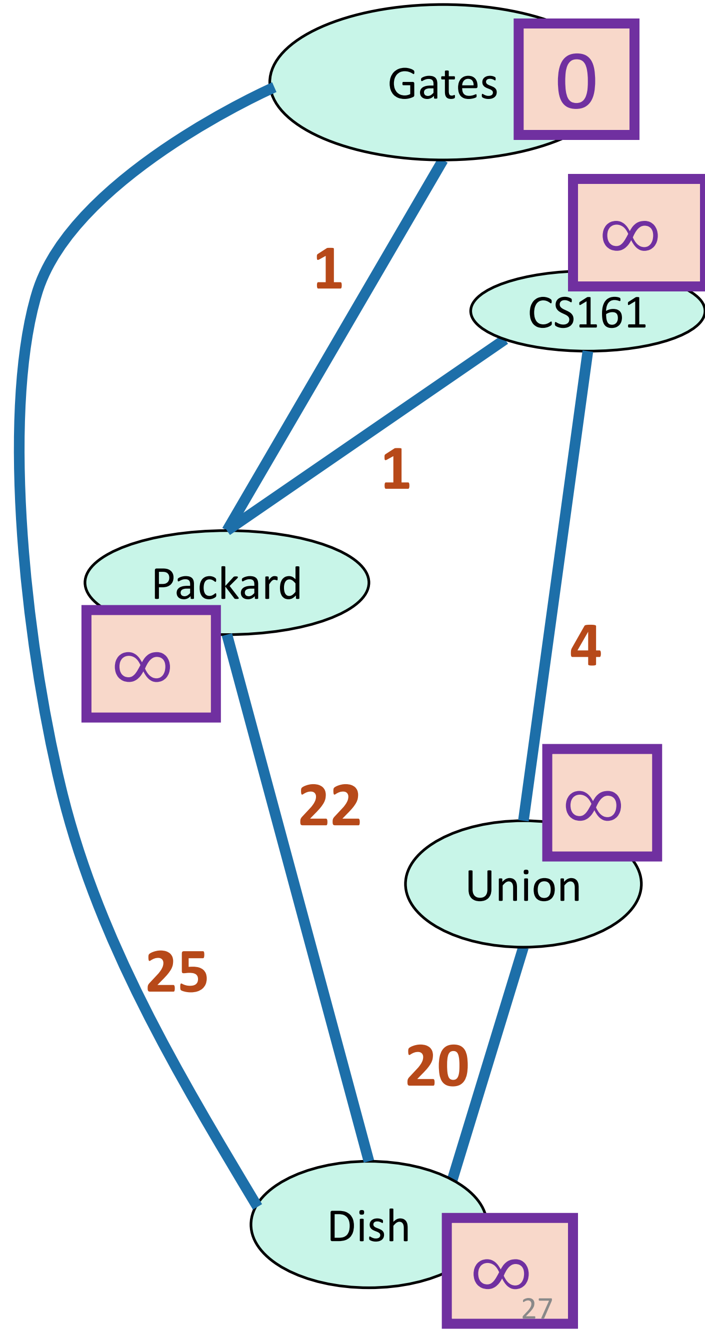
I'm not sure yet

I'm sure

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

Initialize d[v] = ∞
for all non-starting vertices v,
and d[Gates] = 0

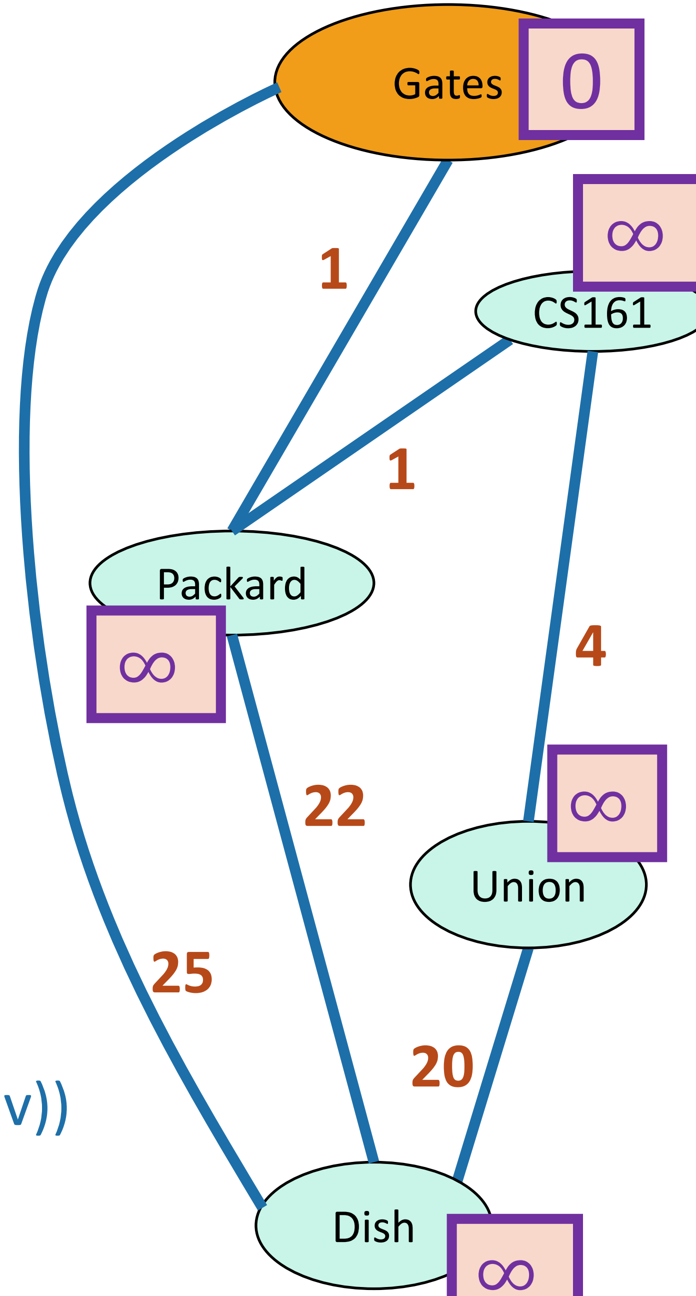
- Pick the **not-sure** node u with the smallest estimate **d[u].**



Dijkstra by example

How far is a node from Gates?

- I'm not sure yet
- I'm sure
- x: **x = d[v]** is my best **over-estimate** for dist(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] + edgeWeight(u,v))

Dijkstra by example

How far is a node from Gates?

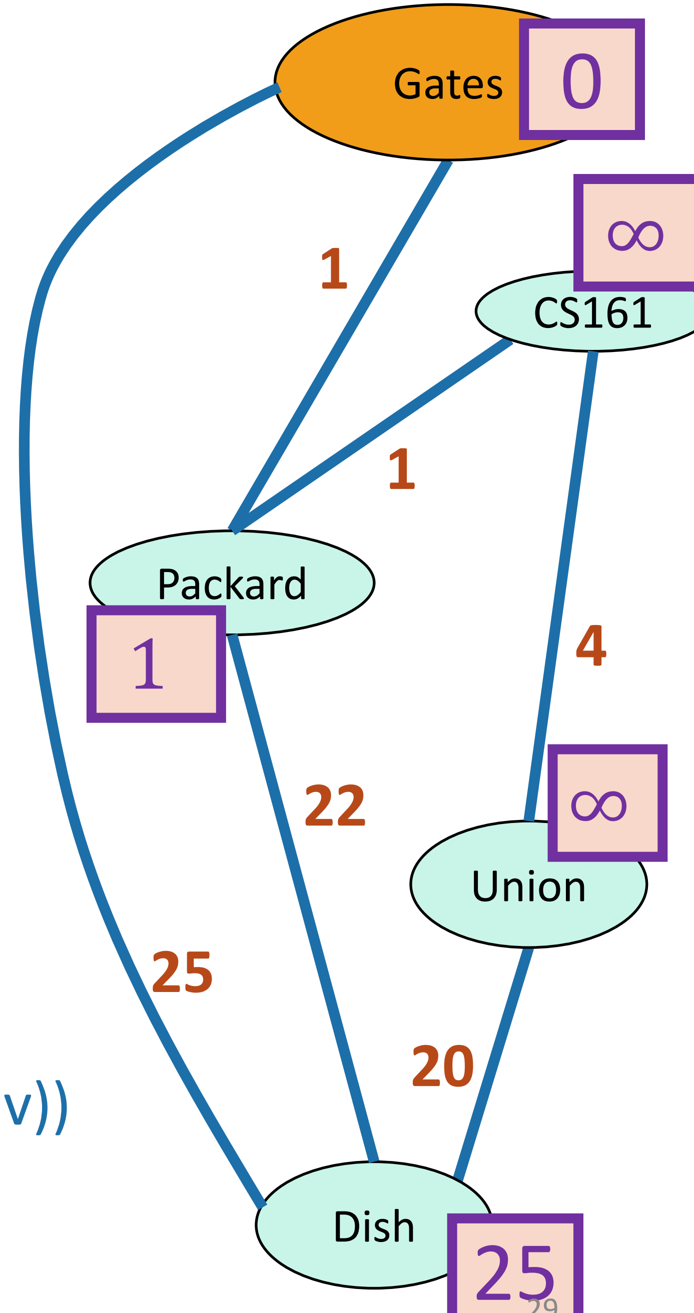
I'm not sure yet

I'm sure

x — **x = d[v]** is my best **over-estimate** for dist(Gates,v).

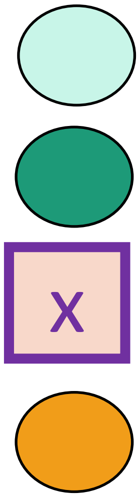
Current node u

- 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] + edgeWeight(u,v))
- Mark u as **sure**.



Dijkstra by example

How far is a node from Gates?



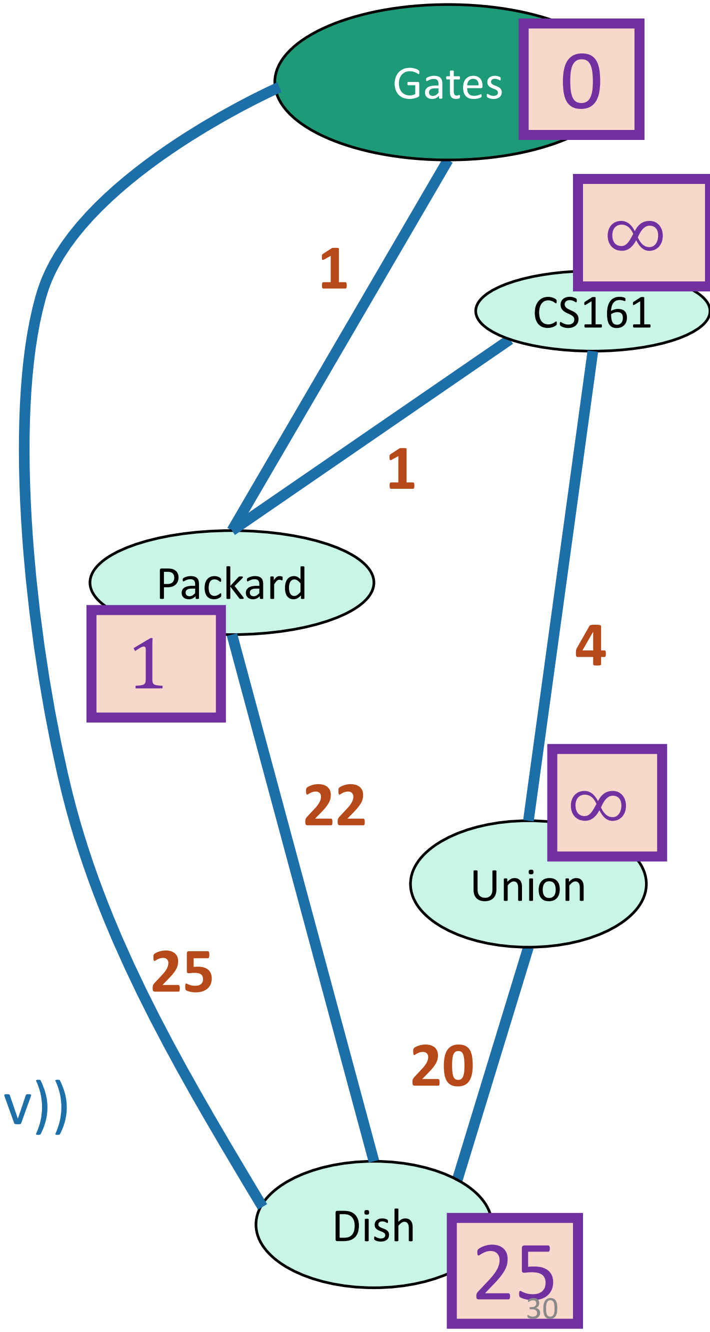
I'm not sure yet

I'm sure

x = d[v] is my best **over-estimate** for dist(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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

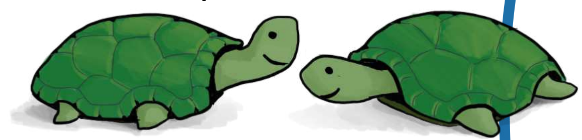


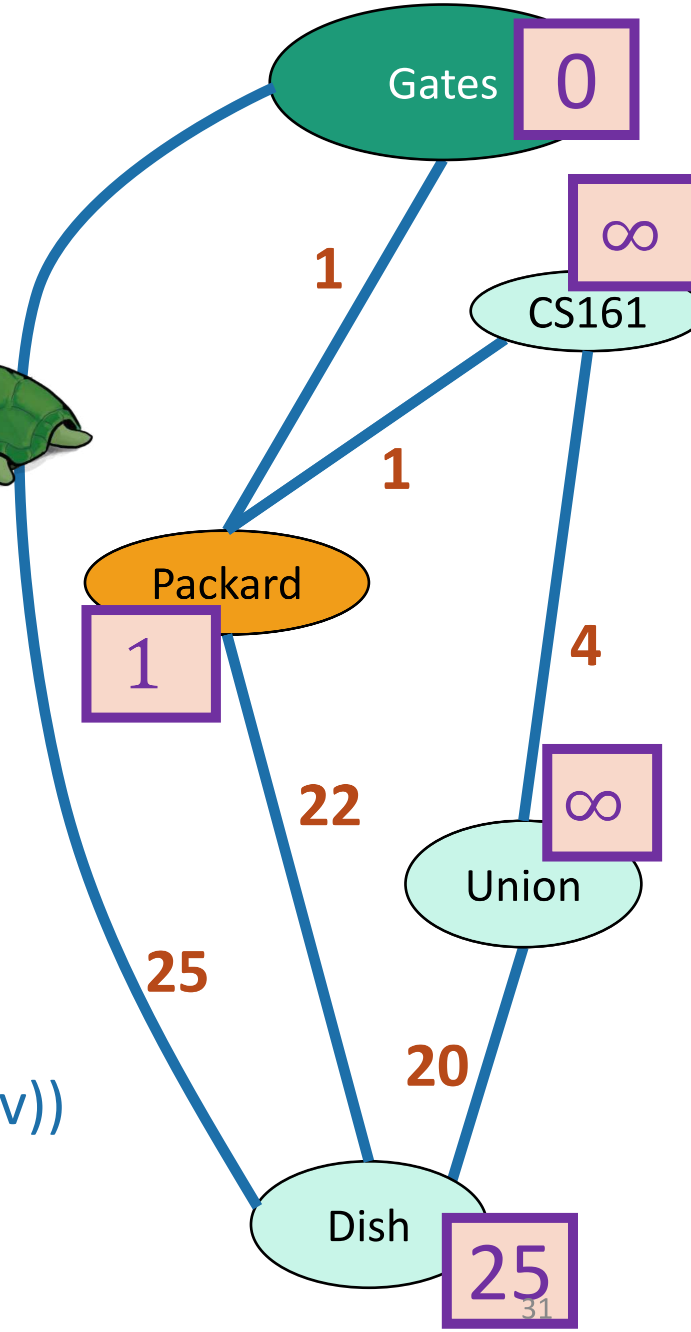
Dijkstra by example

How far is a node from Gates?

- I'm not sure yet
- I'm sure
- x: **x = d[v]** is my best **over-estimate** for dist(Gates,v).
- Current node u

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





- 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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from Gates?

I'm not sure yet

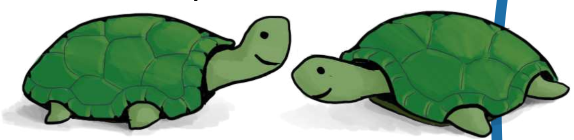
I'm sure

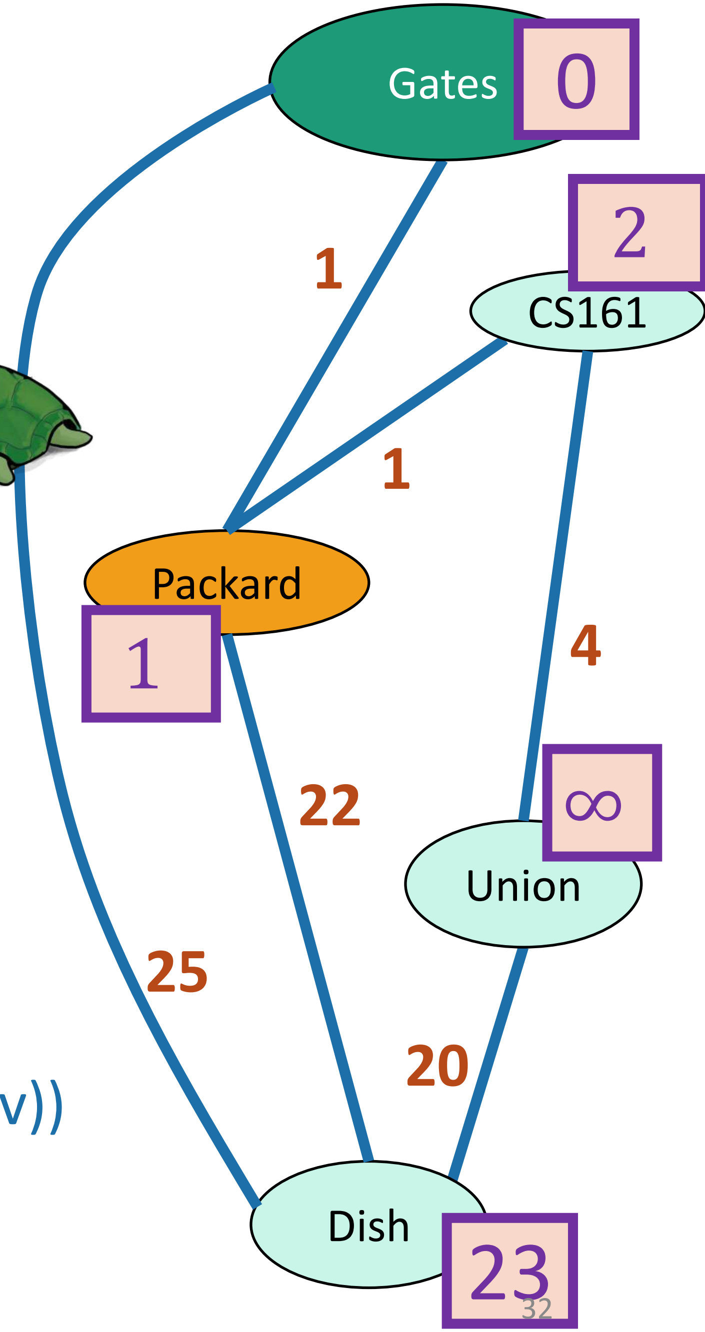
x

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

Current node u

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





- 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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from Gates?

I'm not sure yet

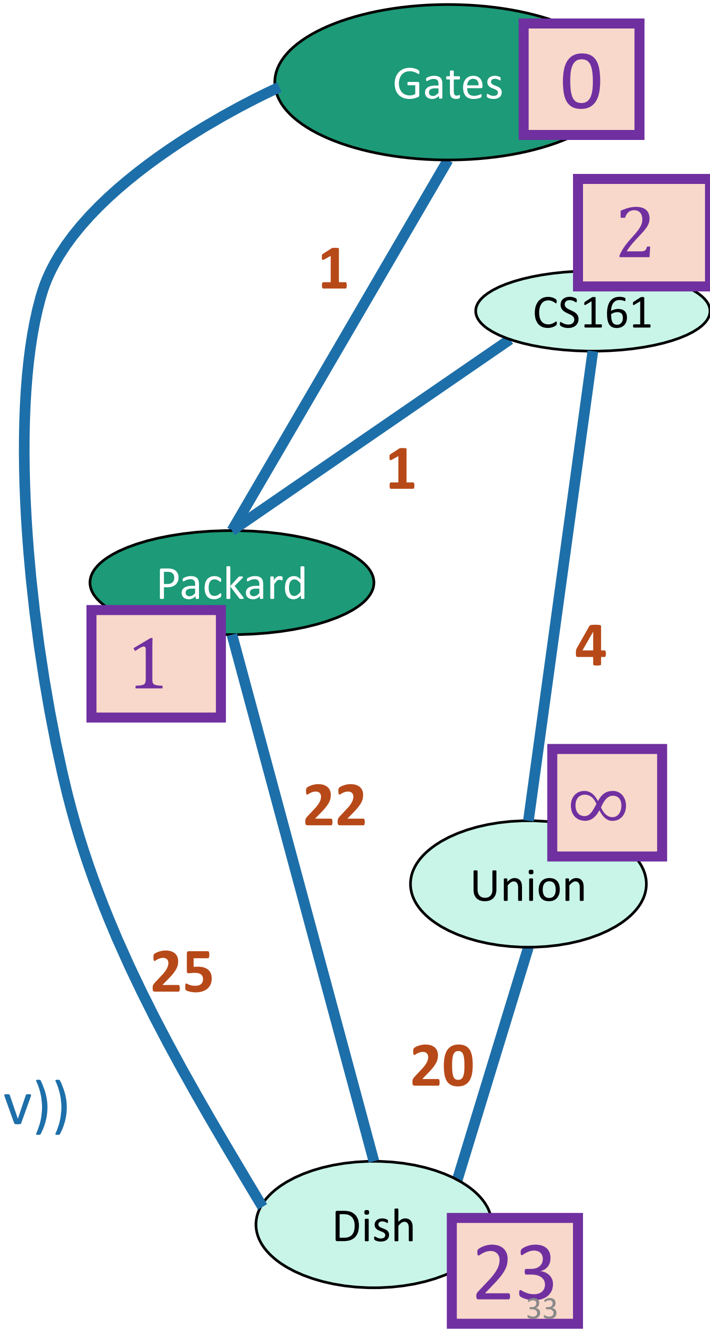
I'm sure

x

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

Current node u

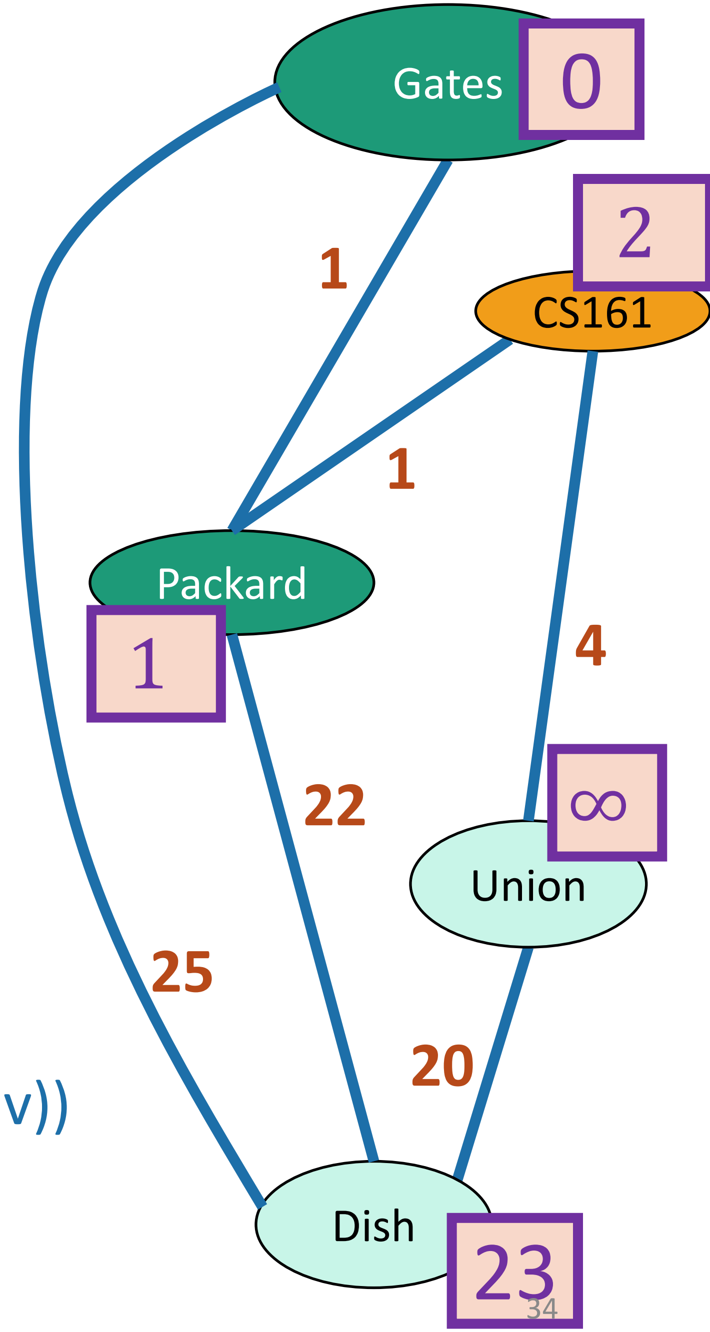
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra by example

How far is a node from Gates?

- I'm not sure yet
- I'm sure
- x: **x = d[v]** is my best **over-estimate** for dist(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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from Gates?

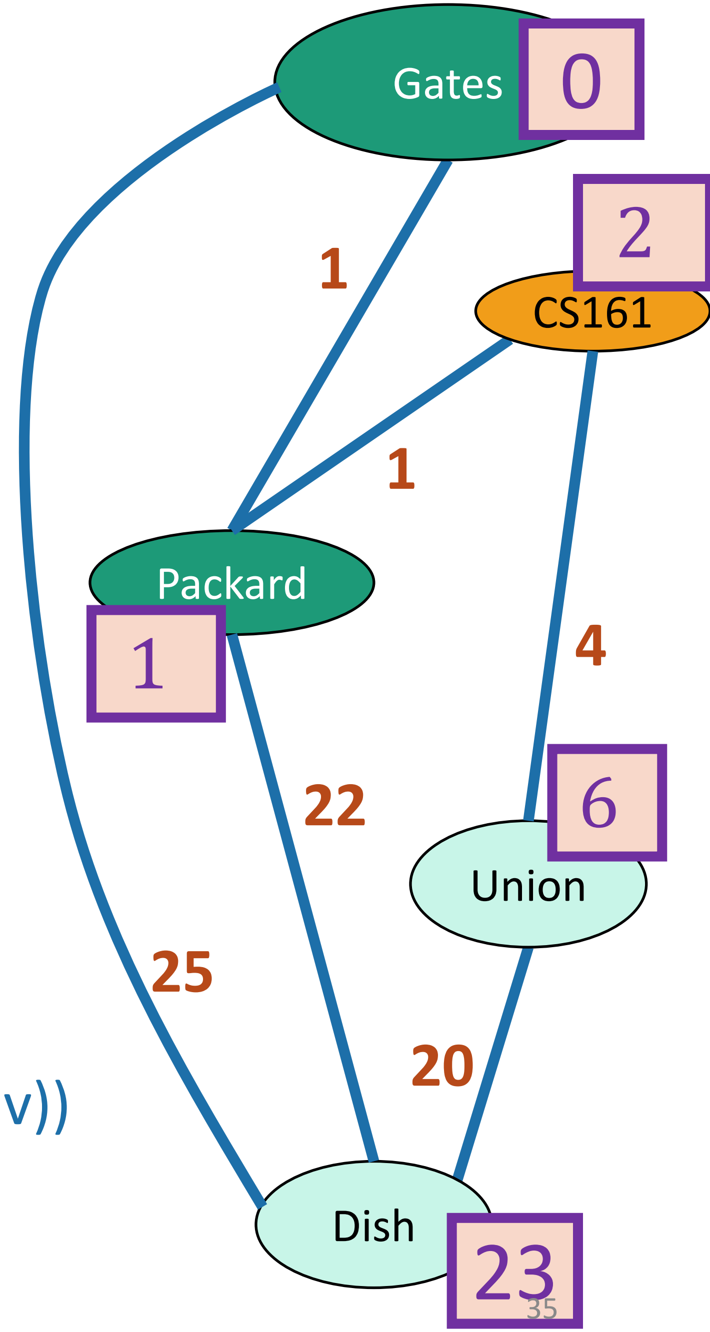
I'm not sure yet

I'm sure

x — **x = d[v]** is my best **over-estimate** for dist(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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

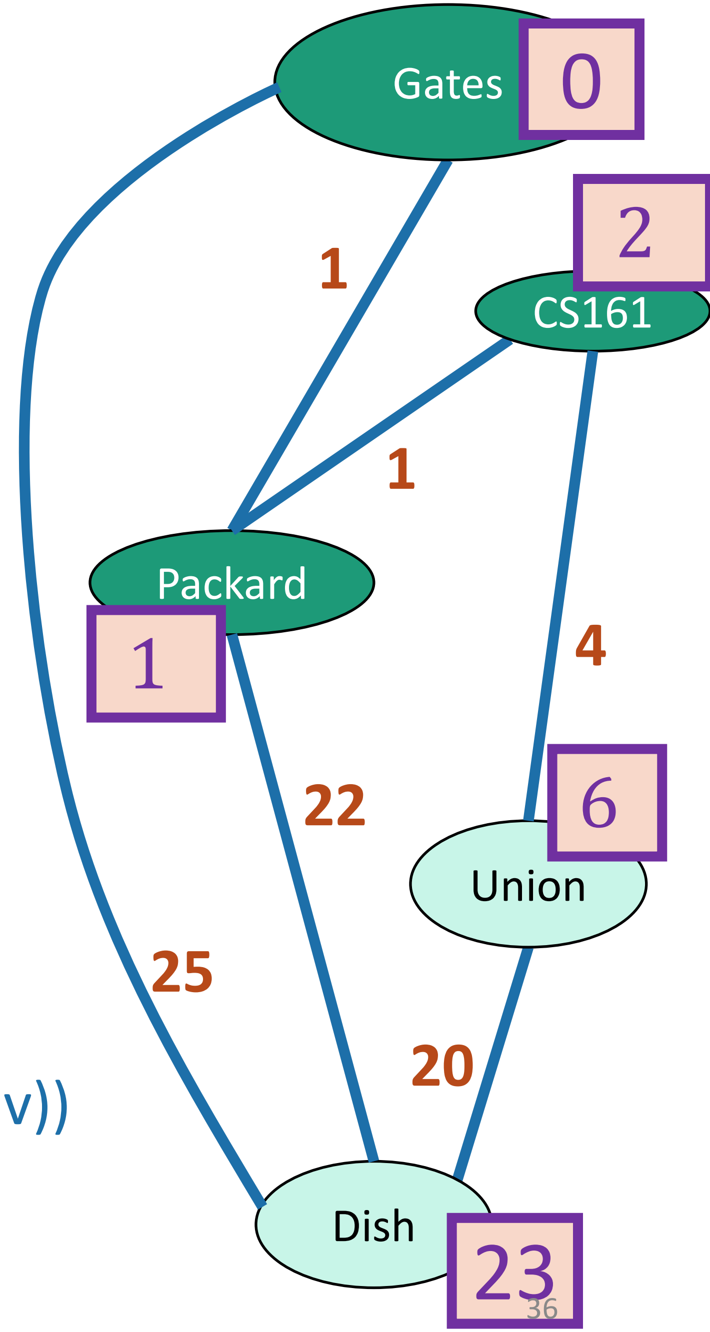


Dijkstra by example

How far is a node from Gates?

- I'm not sure yet
- I'm sure
- **x = d[v]** is my best **over-estimate** for dist(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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra by example

How far is a node from Gates?

I'm not sure yet

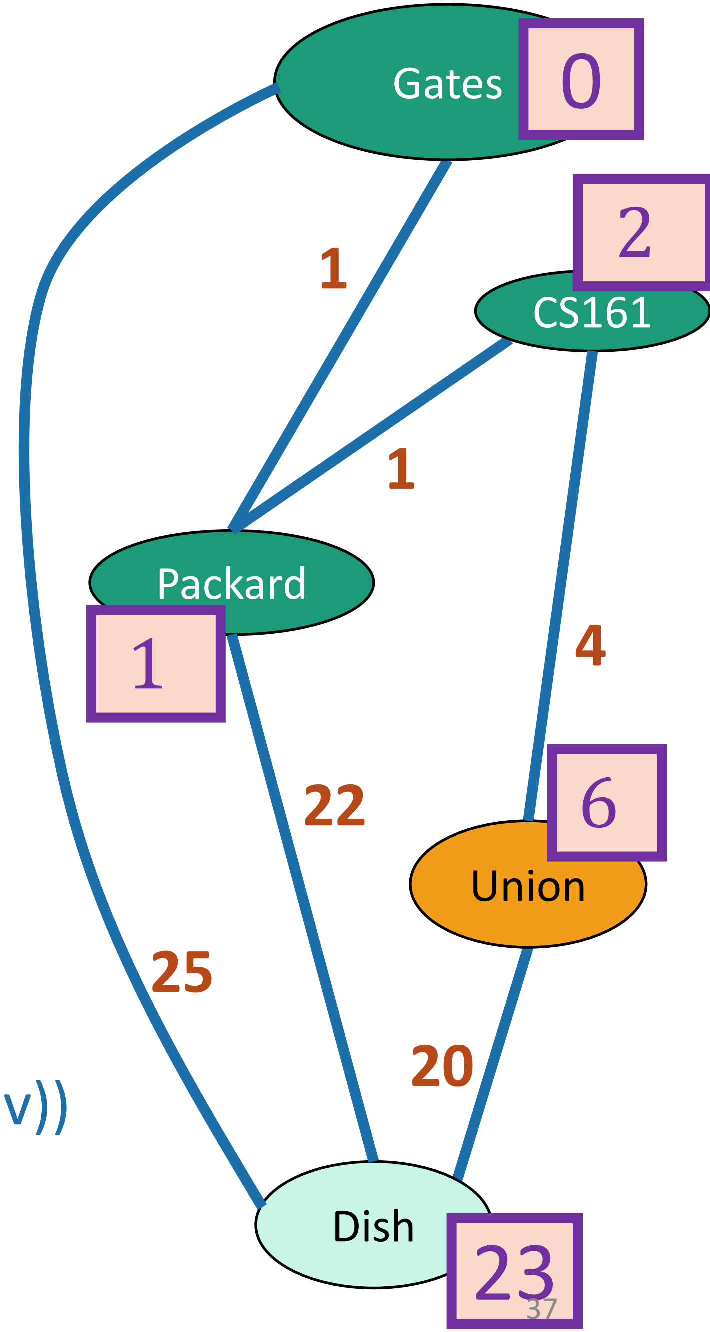
I'm sure

x

x = d[v] is my best **over-estimate** for dist(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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra by example

How far is a node from Gates?

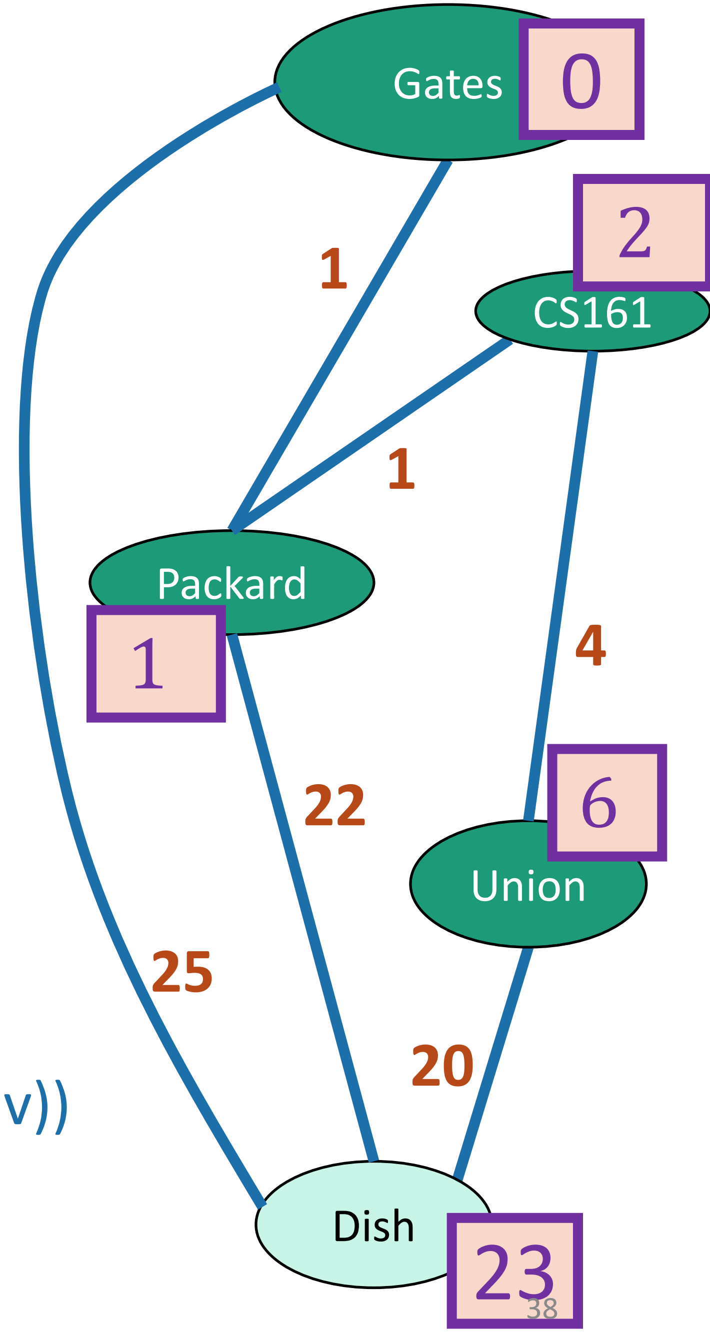
I'm not sure yet

I'm sure

x — **x = d[v]** is my best **over-estimate** for dist(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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra by example

How far is a node from Gates?

I'm not sure yet

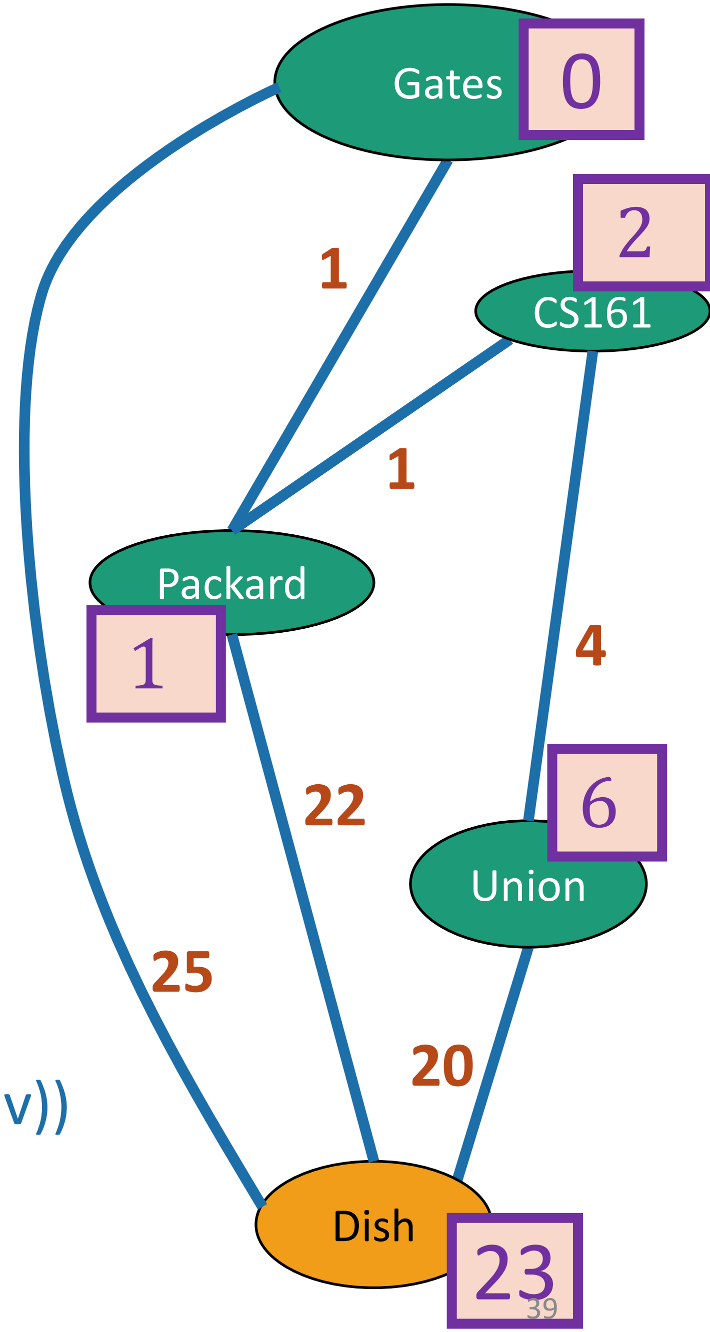
I'm sure

x

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

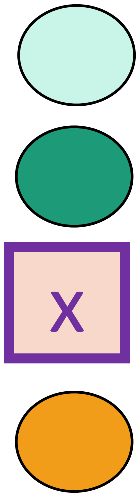
Current node u

- 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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra by example

How far is a node from Gates?



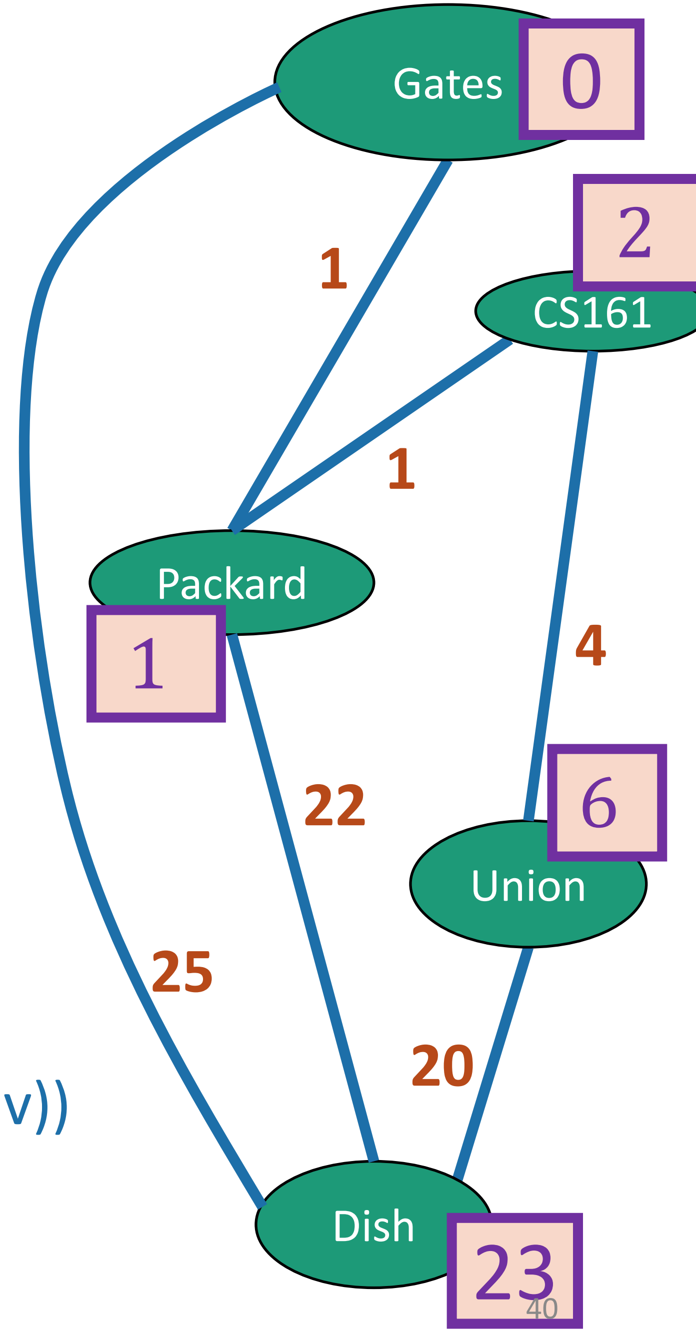
I'm not sure yet

I'm sure

x = d[v] is my best **over-estimate** for dist(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] + 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

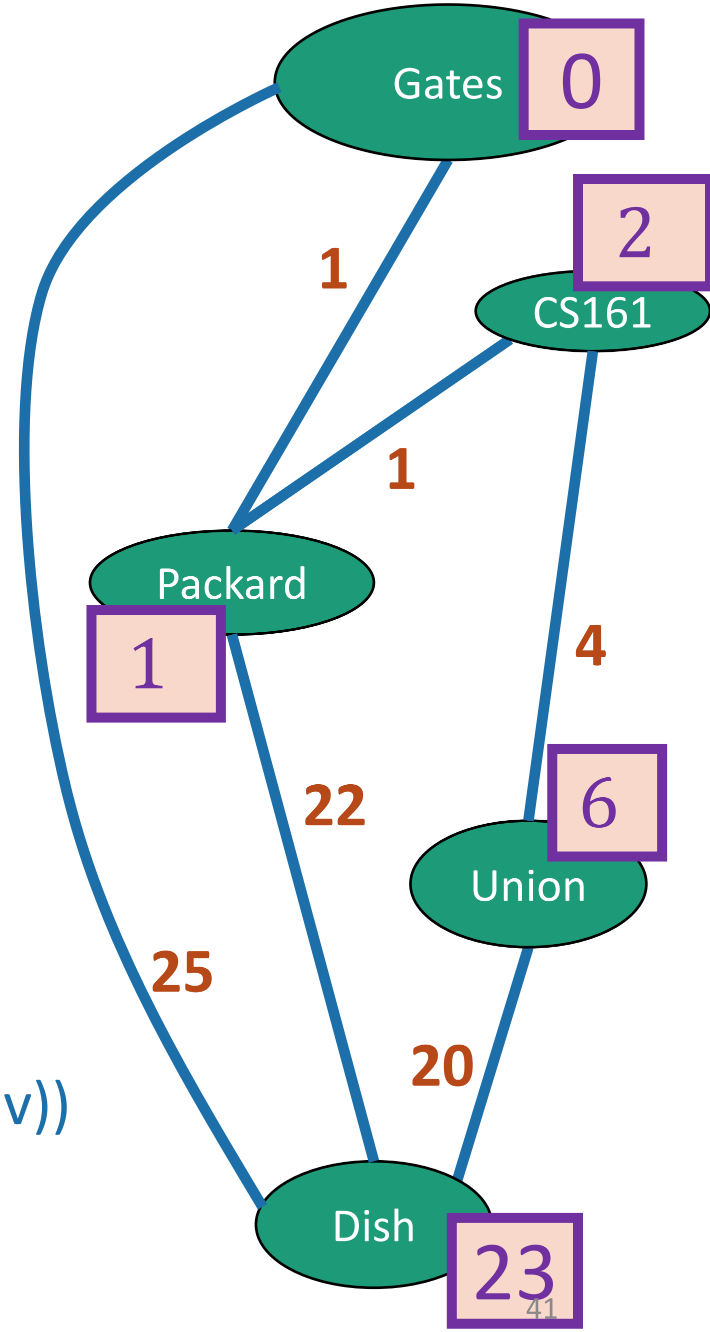
x

x = d[v] is my best **over-estimate** for dist(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] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

- After all nodes are **sure**, say that d(Gates, v) = d[v] for all v



Dijkstra's algorithm

Dijkstra(G,s):

- Set all vertices to **not-sure**
- d[v] = ∞ 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.neighbors:
    - d[v] ← min(d[v] , d[u] + 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:** Let G be a directed, weighted graph with non-negative edge weights.
  - 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).** ← Claim 2
  - **d[v] $\geq$ d(s,v)** and never increases, so after v is **sure**, **d[v]** stops changing. ← Claim 1 + def of algorithm
  - 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:

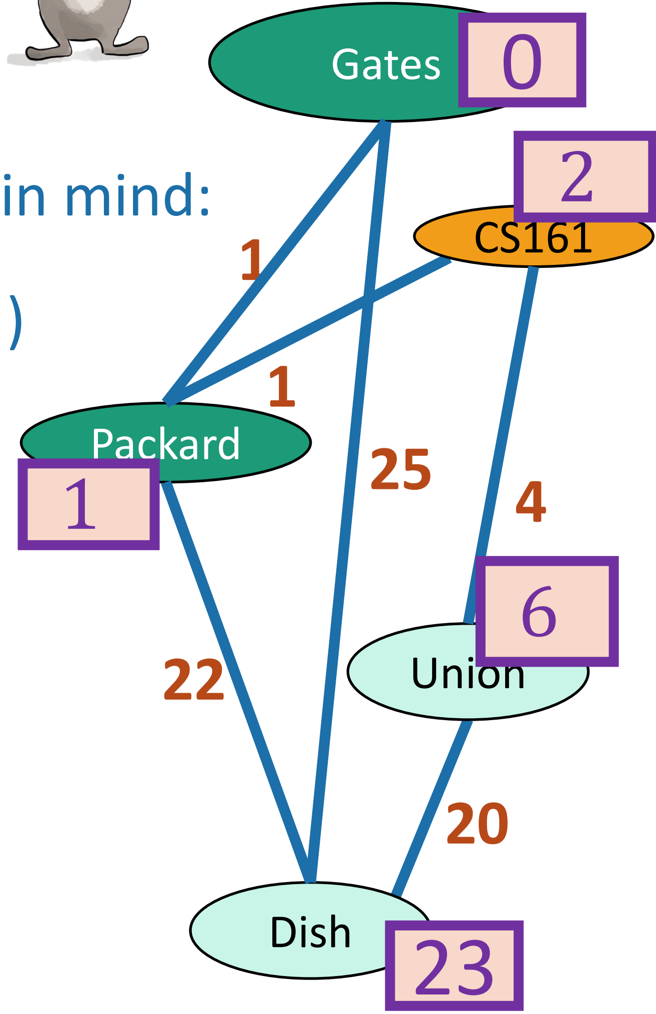
- Every time we update d[v], we have a path in mind:

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

Whatever path we had in mind before

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

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



Formally:

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

Claim 1

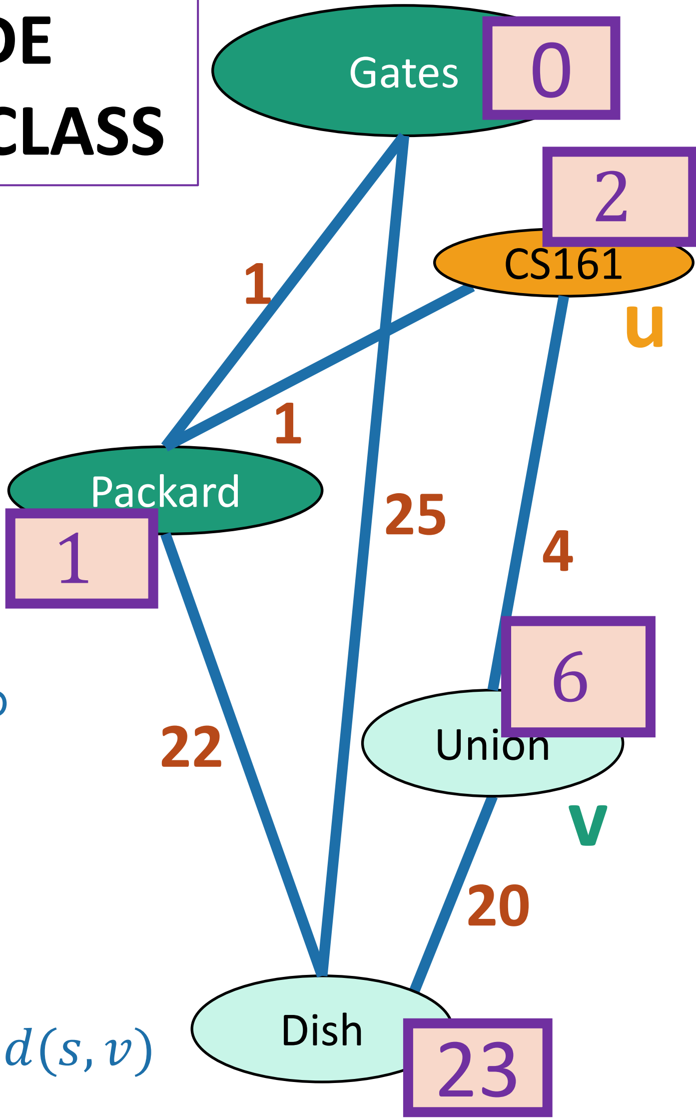
d[v] ≥ d(s,v) for all v.

THIS SLIDE SKIPPED IN CLASS

- Inductive hypothesis.
  - After t iterations of Dijkstra, d[v] ≥ d(s,v) for all v.
- Base case:
  - At step 0, $\mathrm{d}(\mathrm{s},\mathrm{s}) = 0$, 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] ← min(d[v] , d[u] + w(u,v)) $\geq d(s,v)$

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

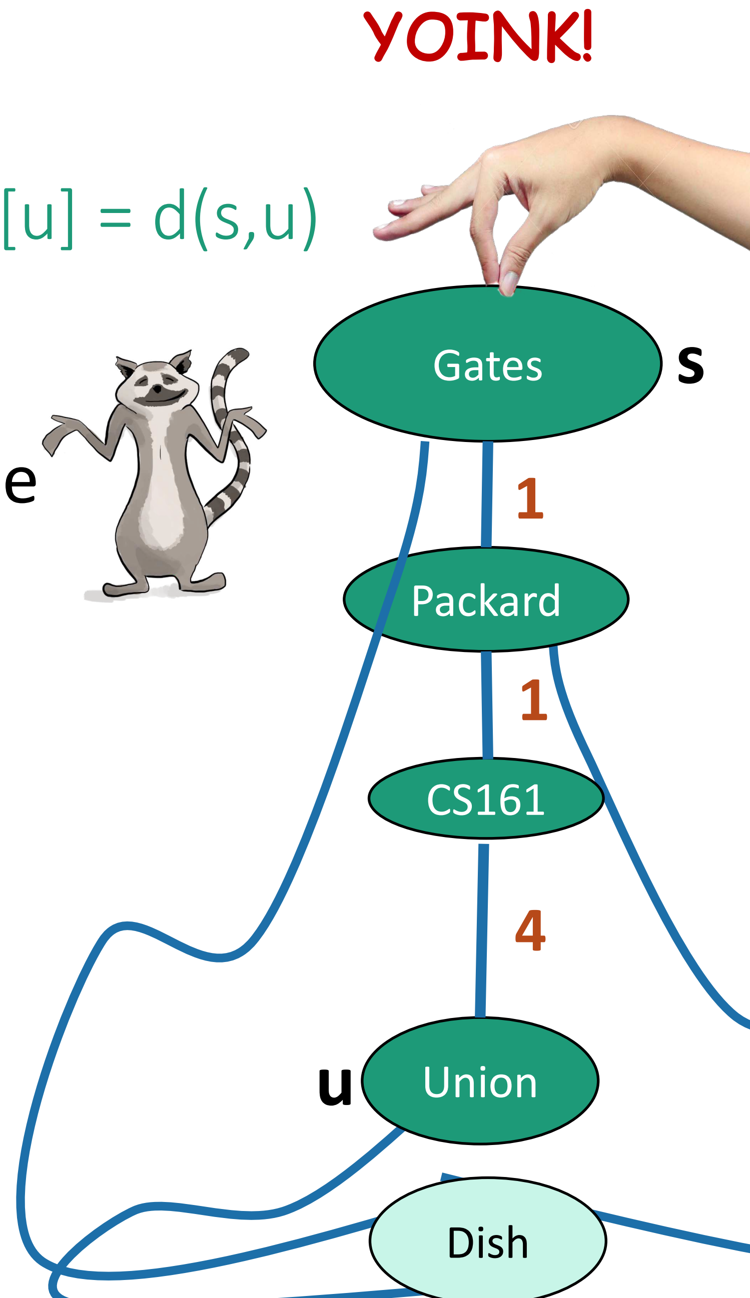
$\mathrm{d}(s,v) \leq d(s,u) + d(u,v)$
$\leq d[u] + w(u,v)$
using induction again for d[u]



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

Intuition for Claim 2

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.

- Let's prove it!
  - Or at least see a proof outline.

Informal outline!

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] = dist(s,v).
- Base case (t=1):
  - The first vertex marked **sure** is s, and d[s] = d(s,s) = 0.

(Assuming edge weights are non-negative!)

- Inductive step:
  - Assume by induction that every v already marked **sure** has d[v] = d(s,v).
  - 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] ← min(d[v] , d[u] + edgeWeight(u,v))
> - Mark u as **sure**.
> - Repeat

- Want to show that d[u] = d(s,u).

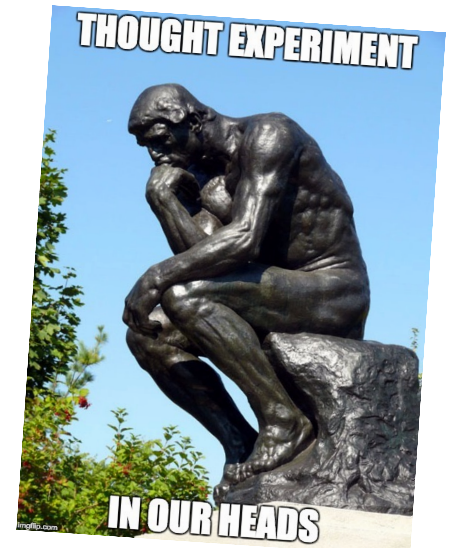
Recall that we picked u so that d[u] is smallest
(out of all not-sure vertices)

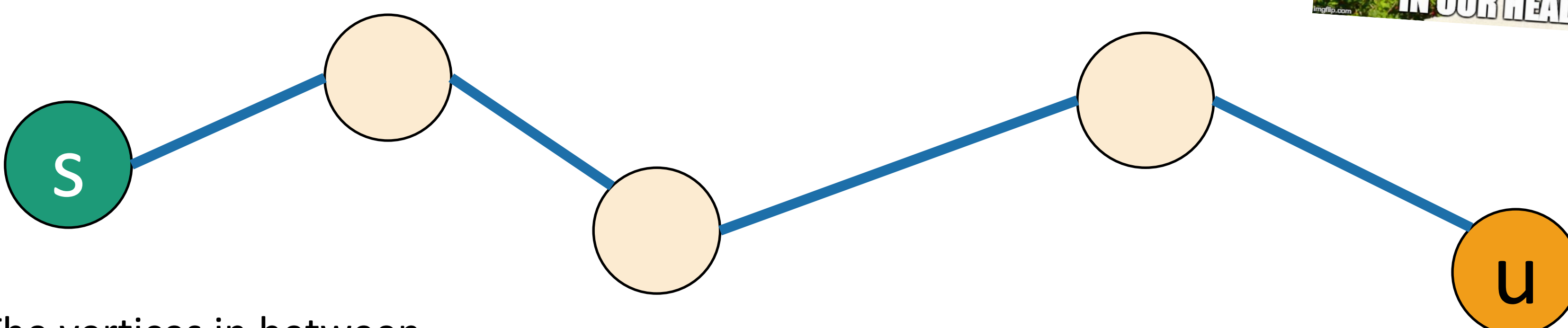
Temporary definition:
v is “good” means that d[v] = d(s,v)

Claim 2

Inductive step

- 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.

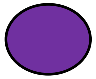
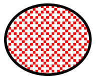
Recall that we picked u so that d[u] is smallest (out of all not-sure vertices)

Claim 2

Inductive step

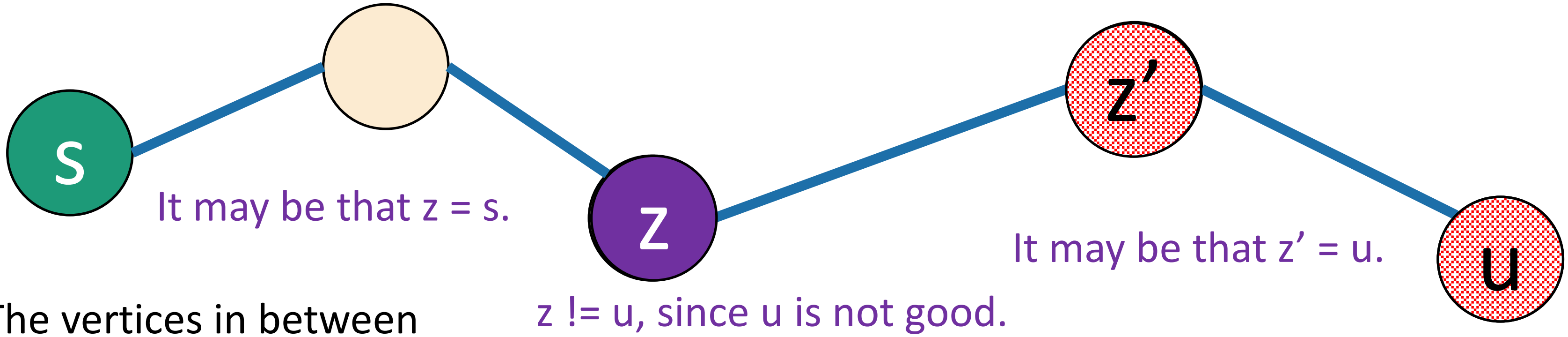
Temporary definition:

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.



It may be that z = s.

It may be that z’ = u.

z != u, since u is not good.

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

True shortest path.

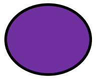
Recall that we picked u so that d[u] is smallest (out of all not-sure vertices)

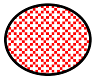
Claim 2

Inductive step

Temporary definition:

v is “good” means that d[v] = d(s,v)

 means good

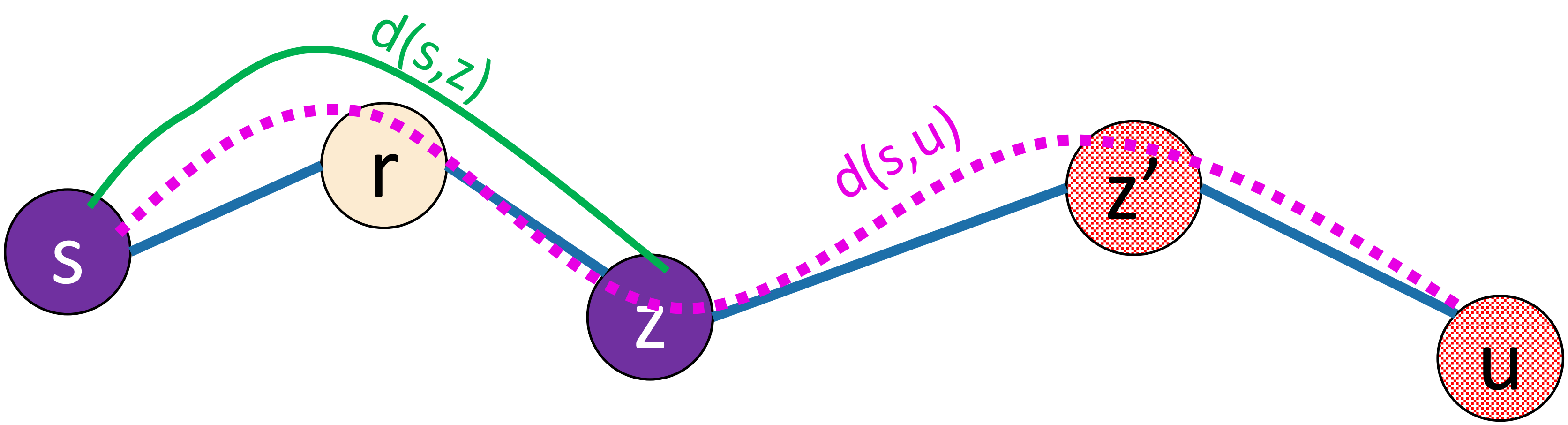
 means not good

- 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.
(We’re also using that the edge weights are non-negative).



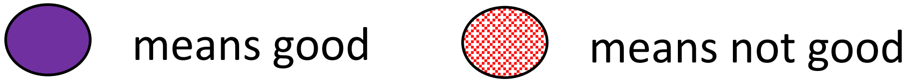
Recall that we picked u so that d[u] is smallest (out of all not-sure vertices)

Claim 2

Inductive step

Temporary definition:

v is "good" means that d[v] = d(s,v)



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

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

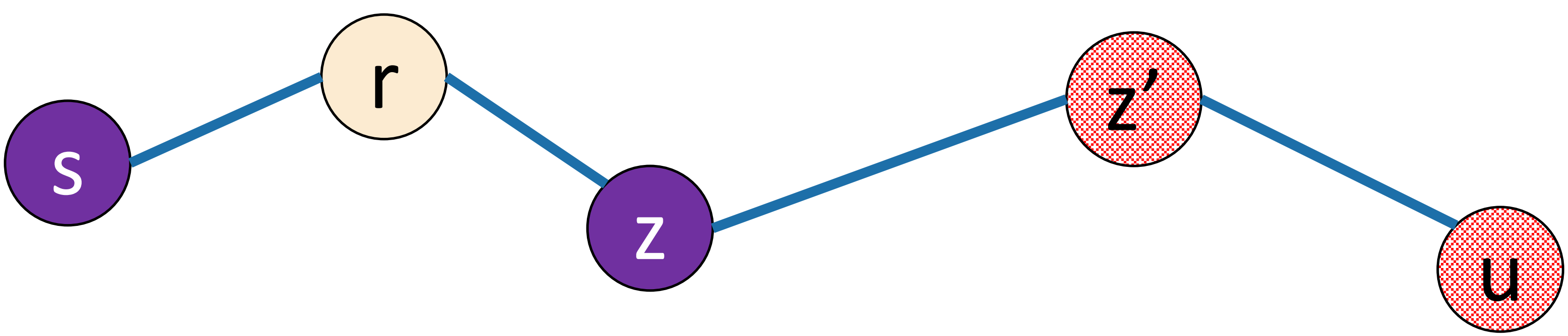
z is good

Subpaths of shortest paths are shortest paths.

Claim 1

- Since u is not good, $d[z] \neq d[u]$.
- So $d[z] < d[u]$, so z is **sure.**

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



Recall that we picked u so that d[u] is smallest (out of all not-sure vertices)

Claim 2

Inductive step

Temporary definition:

v is "good" means that d[v] = d(s,v)

(purple circle) means good (red dotted circle) means not good

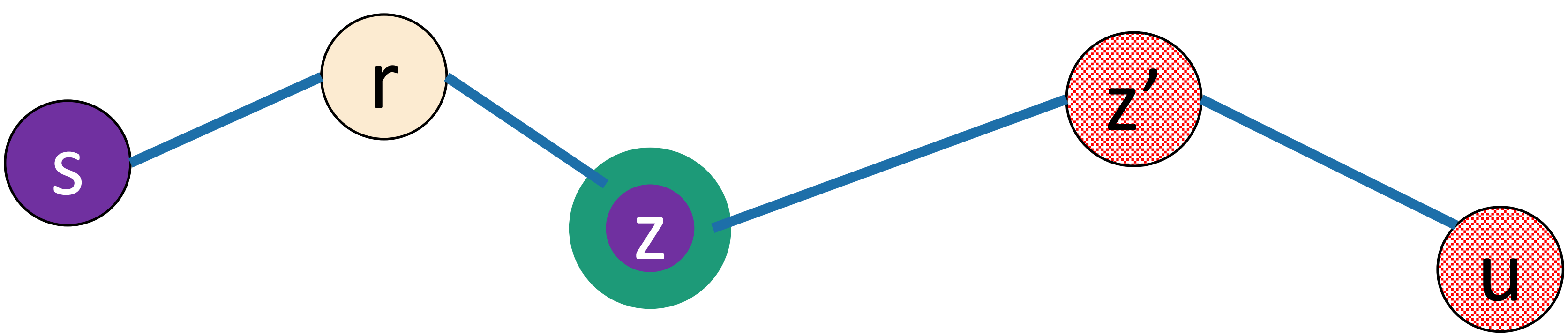
- Want to show that u is good. BWOC, suppose u isn't good.

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

z is good — Subpaths of shortest paths are shortest paths. — Claim 1

- Since u is not good, $d[z] \neq d[u]$.
- So $d[z] < d[u]$, so z is **sure.**

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



Recall that we picked u so that d[u] is smallest (out of all not-sure vertices)

Claim 2

Inductive step

Temporary definition:

v is "good" means that d[v] = d(s,v)

- Want to show that u is good. BWOC, suppose u isn't good.
- If z is sure then we've already updated z':

$d[z'] \leftarrow min\{\, d[z'], d[z] + w(z,z')\}$

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

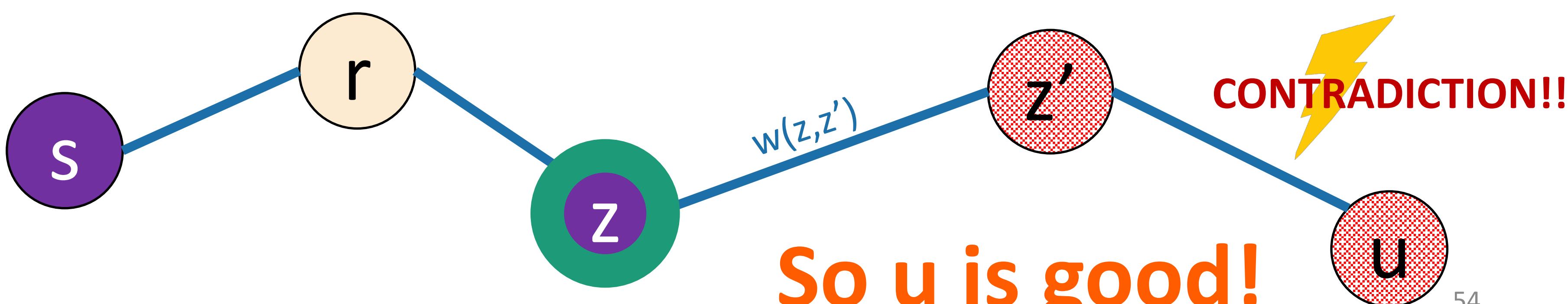
$= \; d(s,z) + w(z,z')$ By induction when z was added to the sure list it had d(s,z) = d[z]

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

$= \; d(s,z')$ sub-paths of shortest paths are shortest paths

$\leq d[z']$ Claim 1

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



So u is good!

Back to this slide

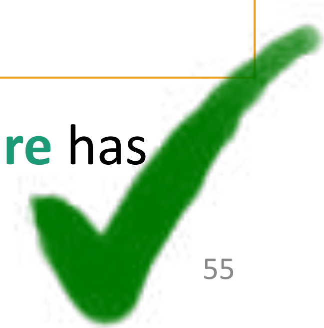
Claim 2

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] = 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] ← min(d[v] , d[u] + 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!

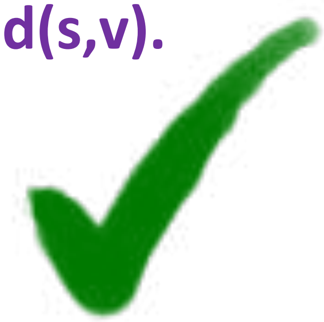
Why does this work?

Now back to this slide

- **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).

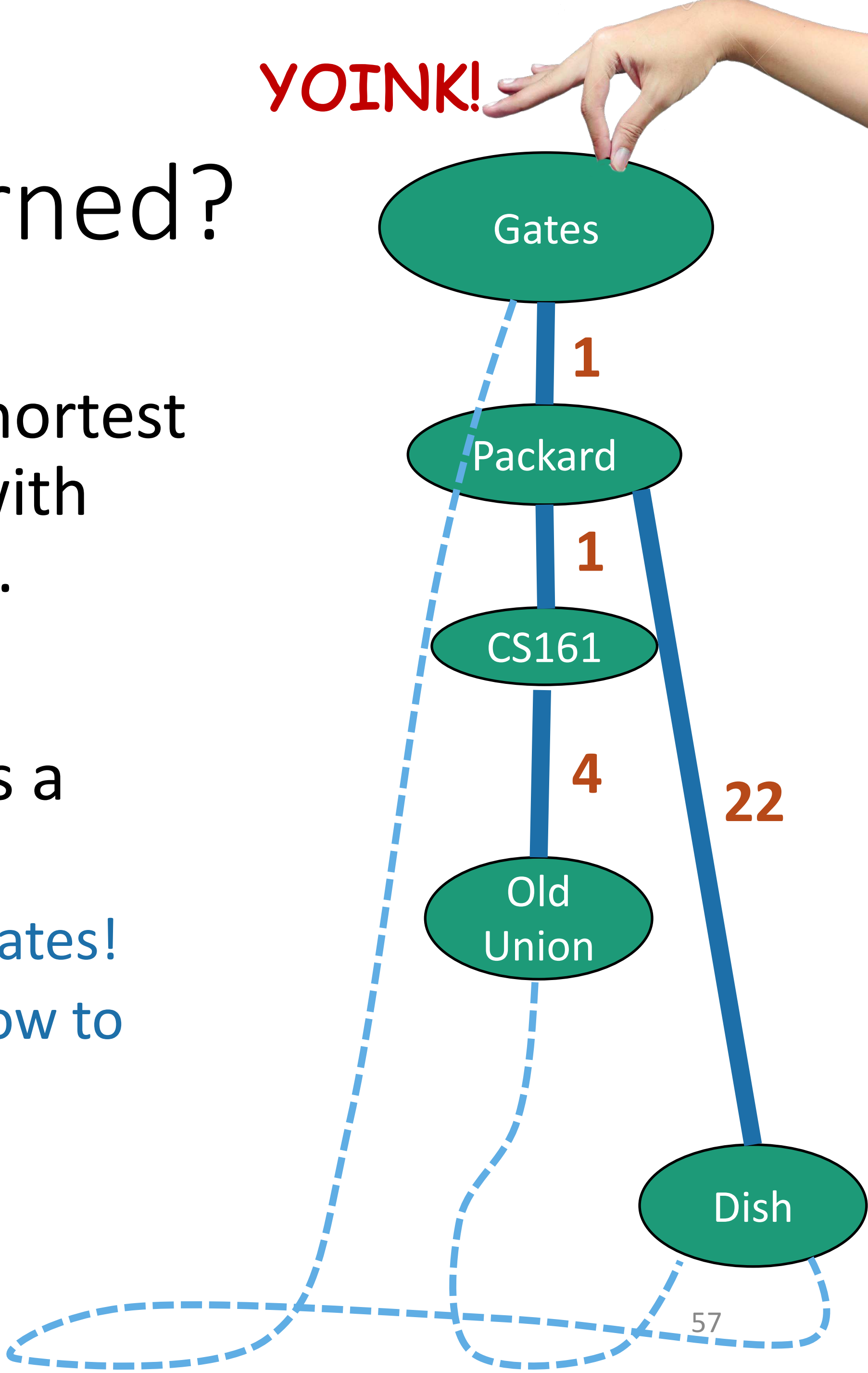
- Proof outline:
  - **Claim 1**: For all v, **d[v] ≥ d(s,v).**
  - **Claim 2**: When a vertex is marked **sure**, **d[v] = d(s,v).**

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



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] = ∞ 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.neighbors:
    - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**.
- Now 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:

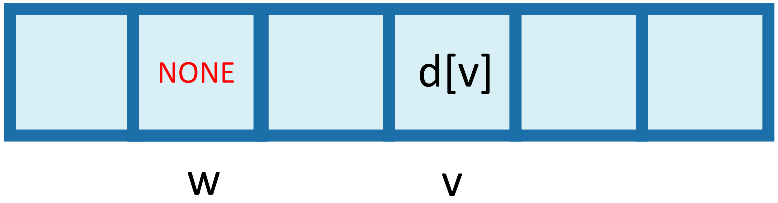
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.

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(`findMin`) + T(`removeMin`)) + m T(`updateKey`)

If we use an array

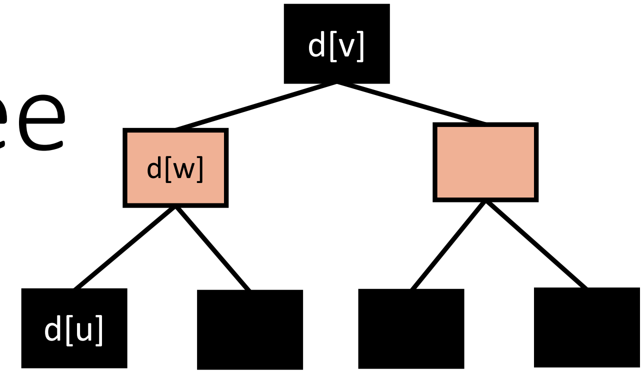


- T(findMin) = O(n)
- T(removeMin) = O(n)
- T(updateKey) = O(1)

- Running time of Dijkstra

$$= O(n(\,T(\texttt{findMin}) + T(\texttt{removeMin})\,) + m\,T(\texttt{updateKey}))$$
$$= O(n^2) + O(m)$$
$$= O(n^2)$$

If we use a red-black tree



- T(findMin) = O(log(n))
- T(removeMin) = O(log(n))
- T(updateKey) = O(log(n))

- Running time of Dijkstra

  =O(n(T(findMin) + T(removeMin)) + m T(updateKey))

  =O(nlog(n)) + O(mlog(n))

  =O((n + m)log(n))

Better than an array if the graph is sparse!

aka if m is much smaller than n^2

If we use a **Fibonacci Heap**

We won't cover heaps in this class! See CS166!
(You should know these supported operations and running times, but nothing else).

- T(findMin) = O(1) (amortized time*)
- T(removeMin) = O(log(n)) (amortized time*)
- T(updateKey) = O(1) (amortized time*)

- Running time of Dijkstra

  = O(n(T(`findMin`) + T(`removeMin`)) + m T(`updateKey`))

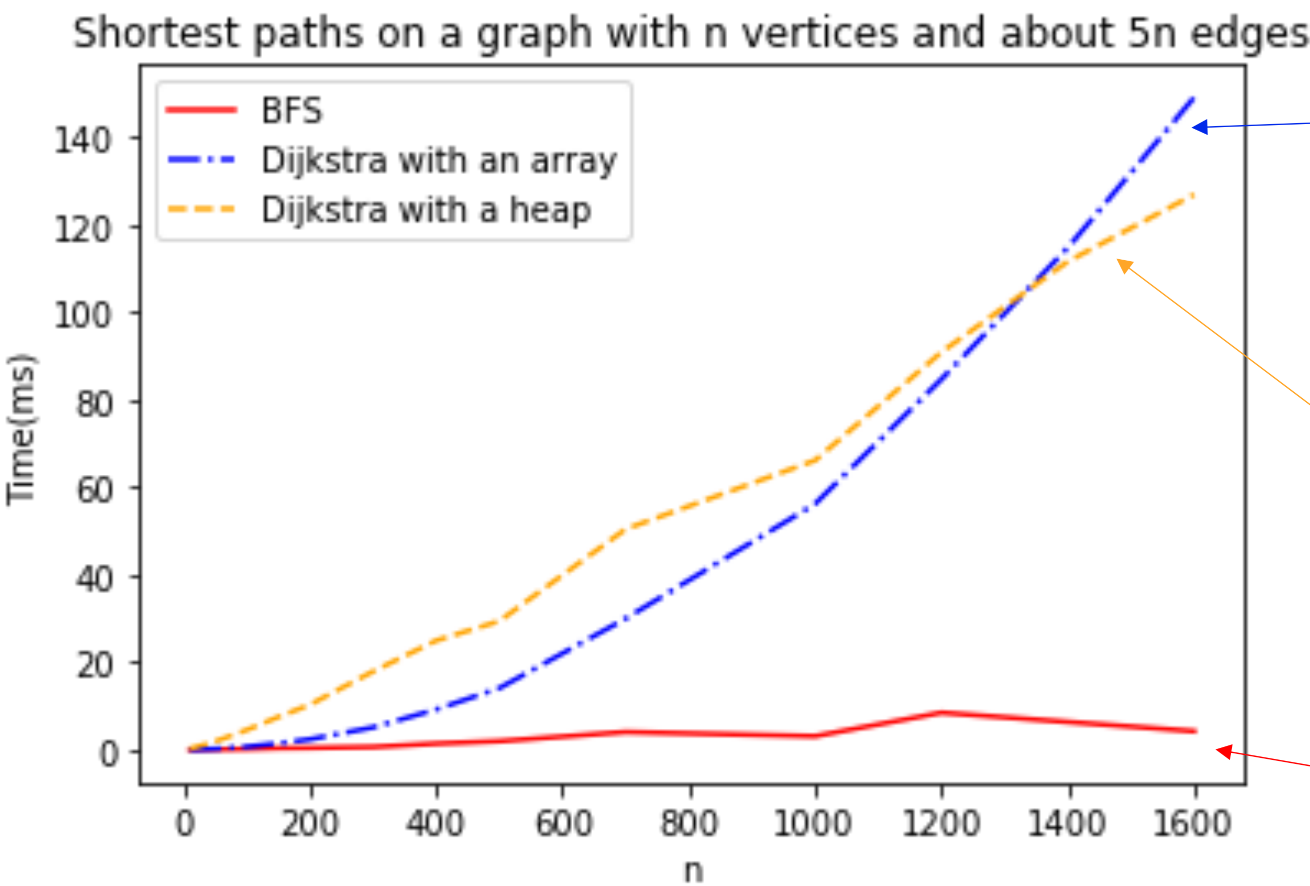
  = O(nlog(n) + m) (amortized time)

Compare:
Array: $O(n^2)$
RBTree: O((n+m)log n)

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

See IPython Notebook for Lecture 11
The heap is implemented using `heapdict`

In practice



Dijkstra using a Python list to keep track of vertices has quadratic runtime.

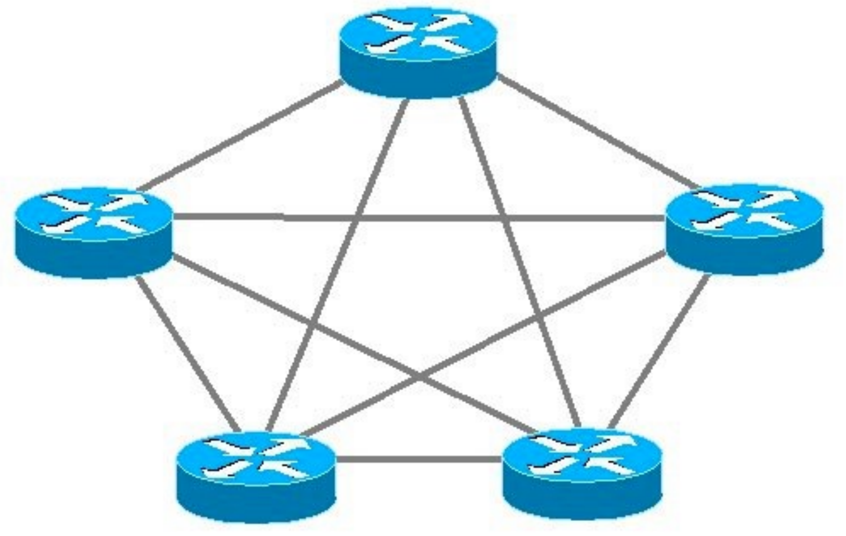
Dijkstra using a heap looks a bit more linear (actually nlog(n))

BFS is really fast by comparison! But it doesn't work on weighted graphs.

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

- Assumes 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.
  - We'll see formal definitions/pseudocode next time.

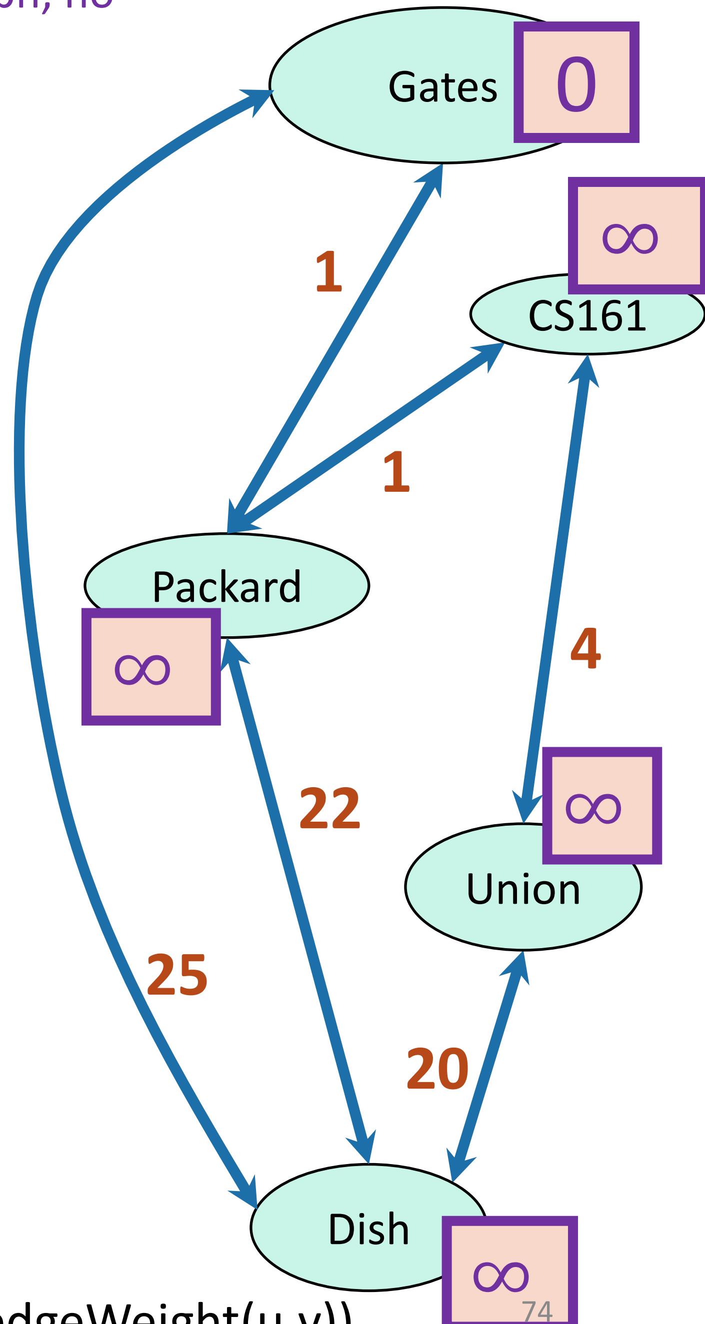
- Basic idea:
  - Instead of picking the u with the smallest d[u] to update, just update all of the u's simultaneously.

Bellman-Ford

Start with the same graph, no negative weights.

How far is a node from Gates?

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



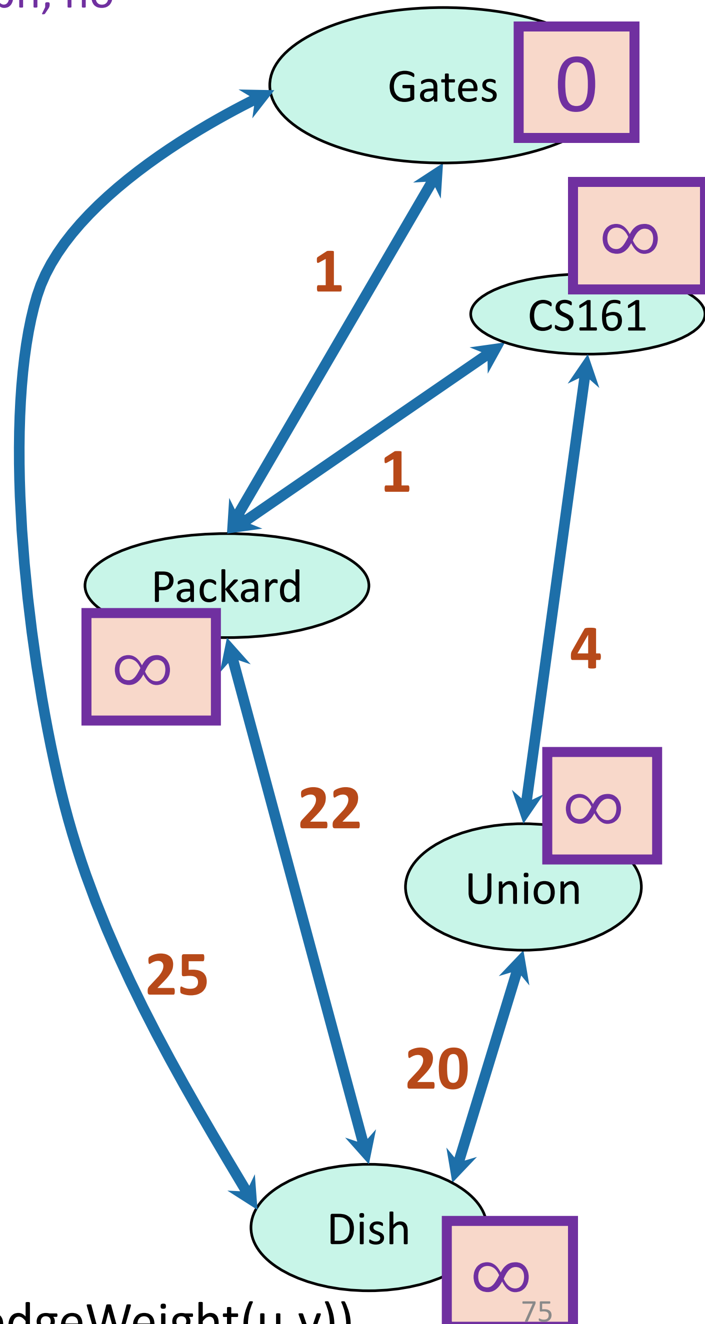
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

Start with the same graph, no negative weights.

How far is a node from Gates?

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



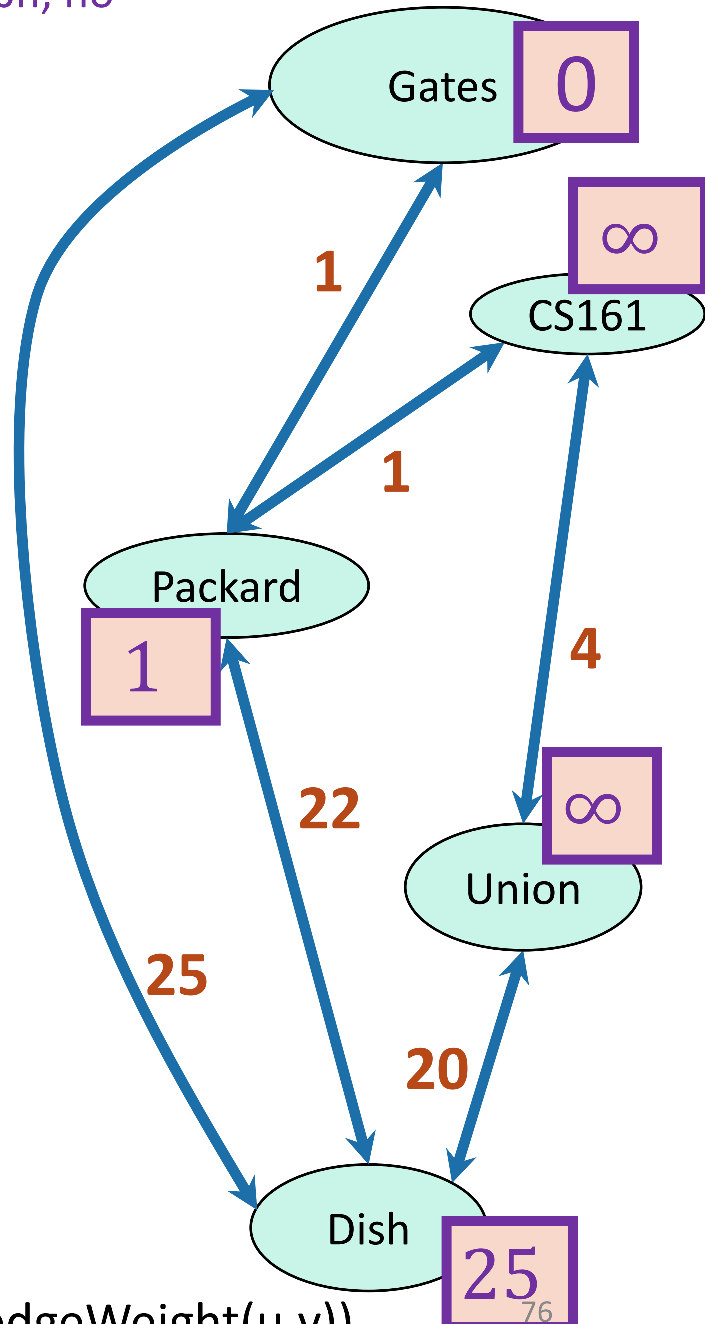
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

Start with the same graph, no negative weights.

How far is a node from Gates?

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



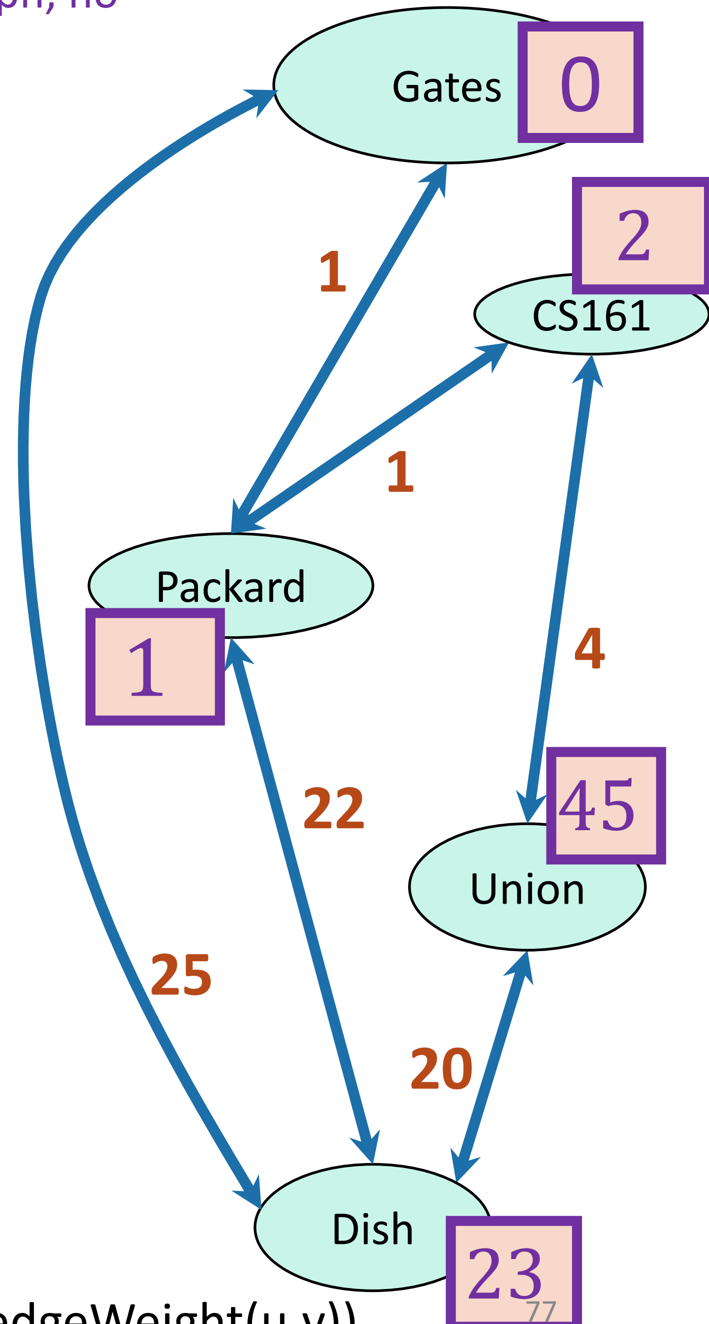
- **For** i=0,…,n-2:
    - **For** u in V:
        - **For** v in u.neighbors:
            - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

Start with the same graph, no negative weights.

How far is a node from Gates?

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



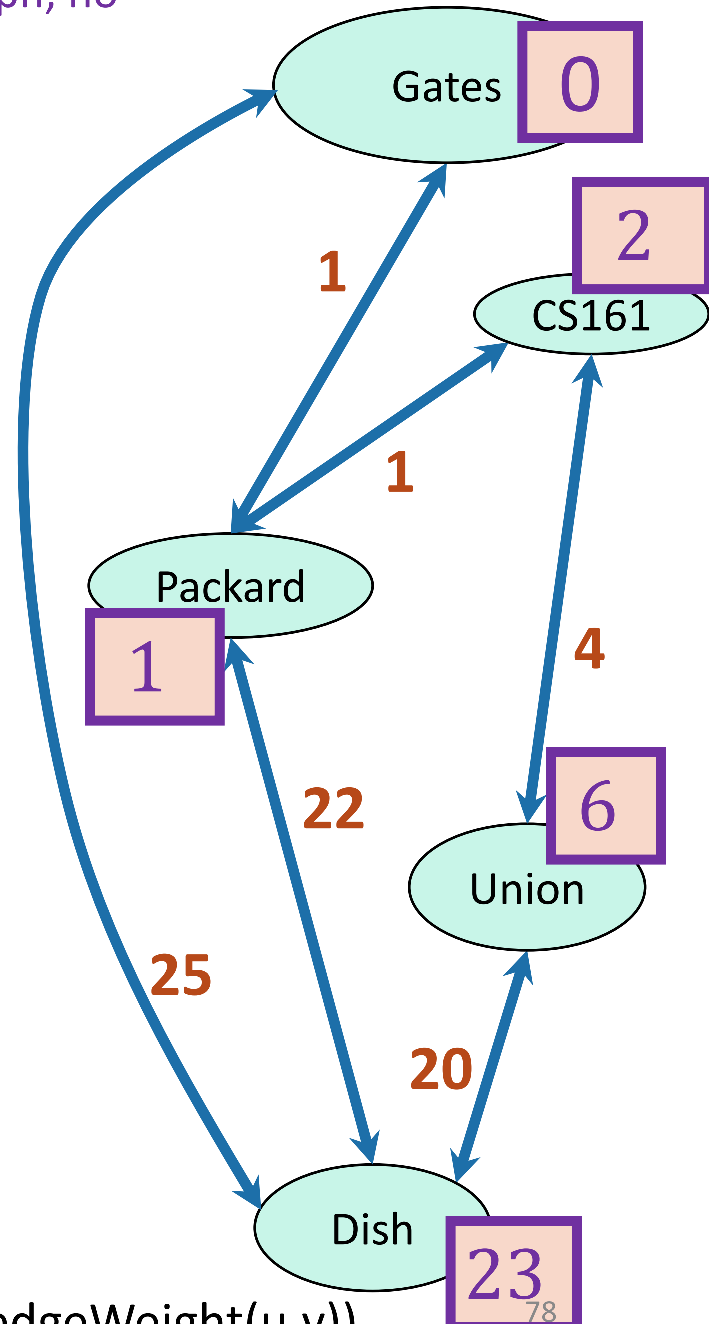
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

Start with the same graph, no negative weights.

How far is a node from Gates?

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



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

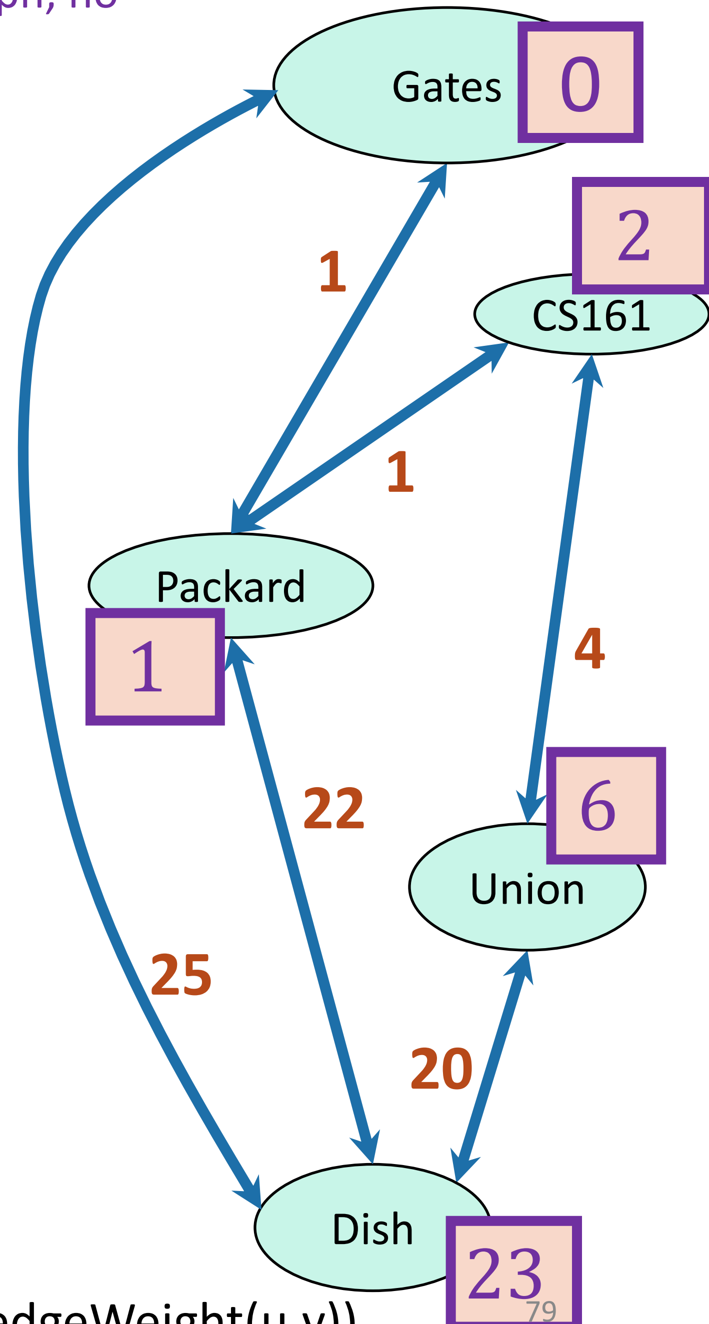
Bellman-Ford

Start with the same graph, no negative weights.

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**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!



- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.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)

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**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).**

- Is it fast?
  - Not really…
  - O(mn)

A **simple path** is a path with no cycles.

Skipped in class

Proof by induction

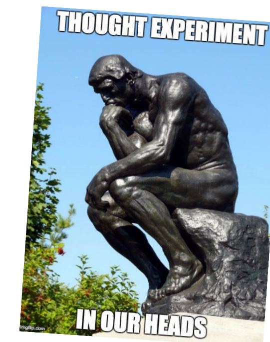
- **Inductive Hypothesis:**
  - After iteration i, for each v, $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v with at most i edges.
- **Base case:**
  - After iteration 0…
- **Inductive step:**

Skipped in class

Inductive step

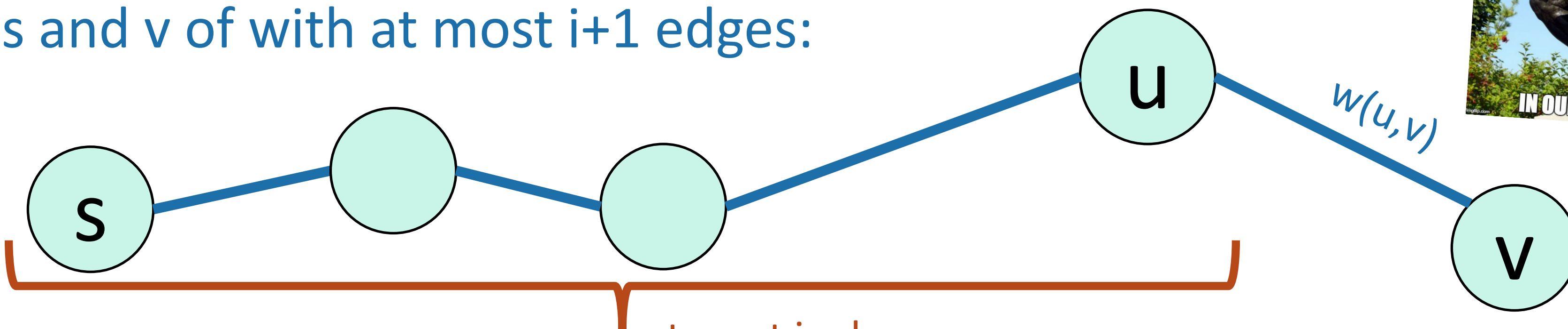
Hypothesis: After iteration i, for each v, $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v with at most i edges.

- 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:

Let u be the vertex right before v in this path.



- 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 $\mathbf{d^{(i+1)}[v] <= d^{(i)}[u] + w(u,v)}$.
- So $d^{(i+1)}[v]$ <= 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 >= cost of shortest path.
- So d[v] = cost of shortest path with at most i+1 edges.

Skipped in class

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 negative cycles!

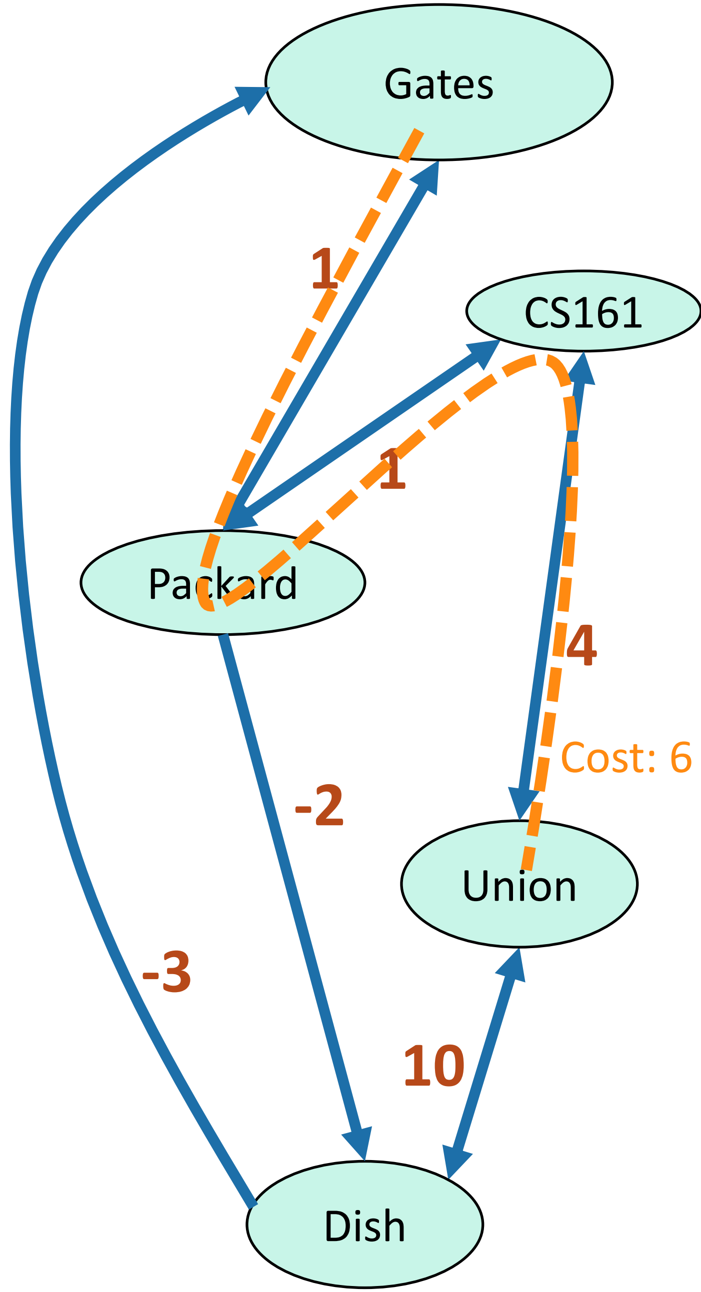
Nice things about Bellman-Ford

- Flexible if the weights change
  - Each node continuously updates itself by querying its neighbors, and changes in the network will eventually propagate through.
- Can handle negative edge weights*

*As long as there aren't negative cycles!

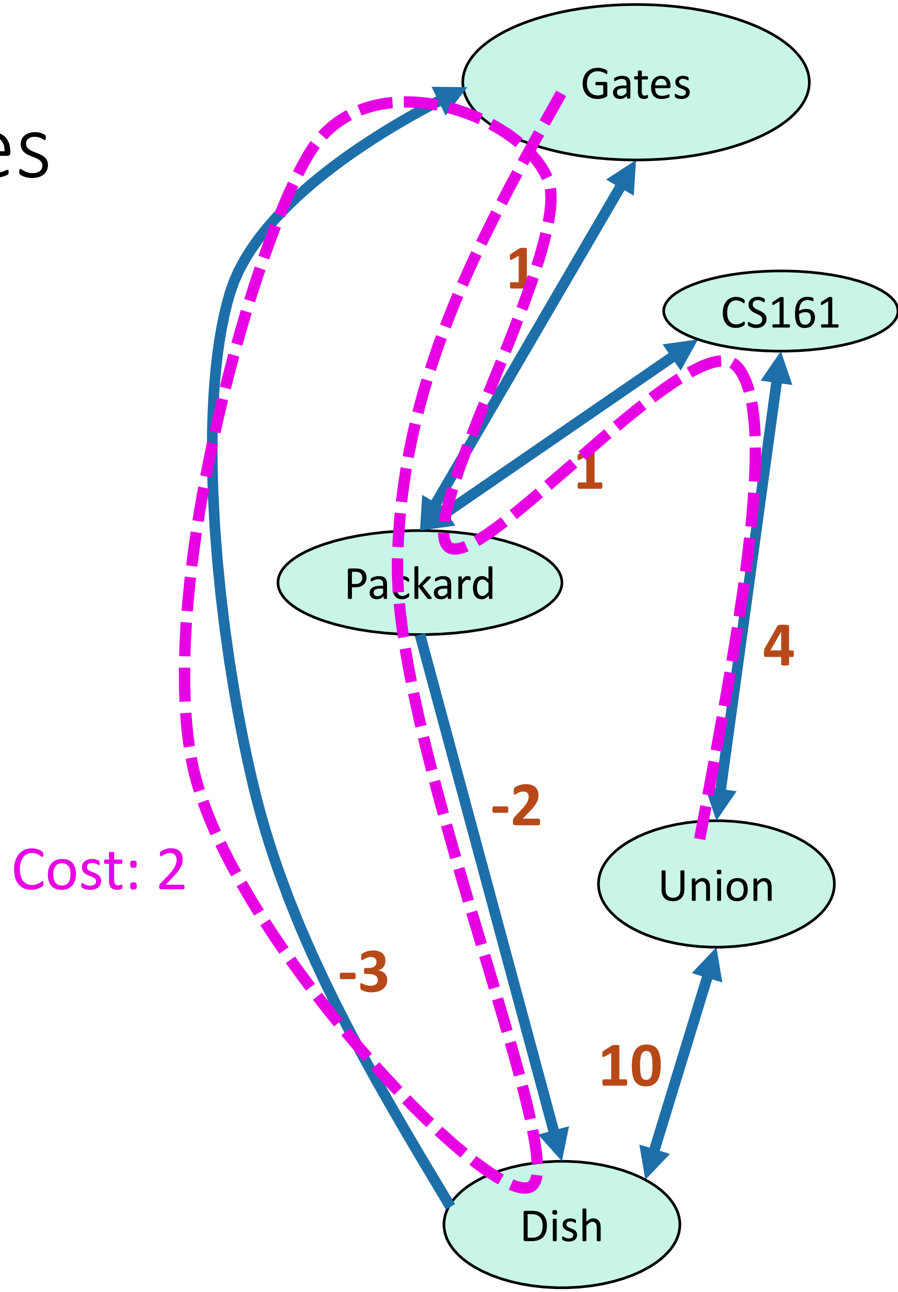
Caution: negative cycles

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



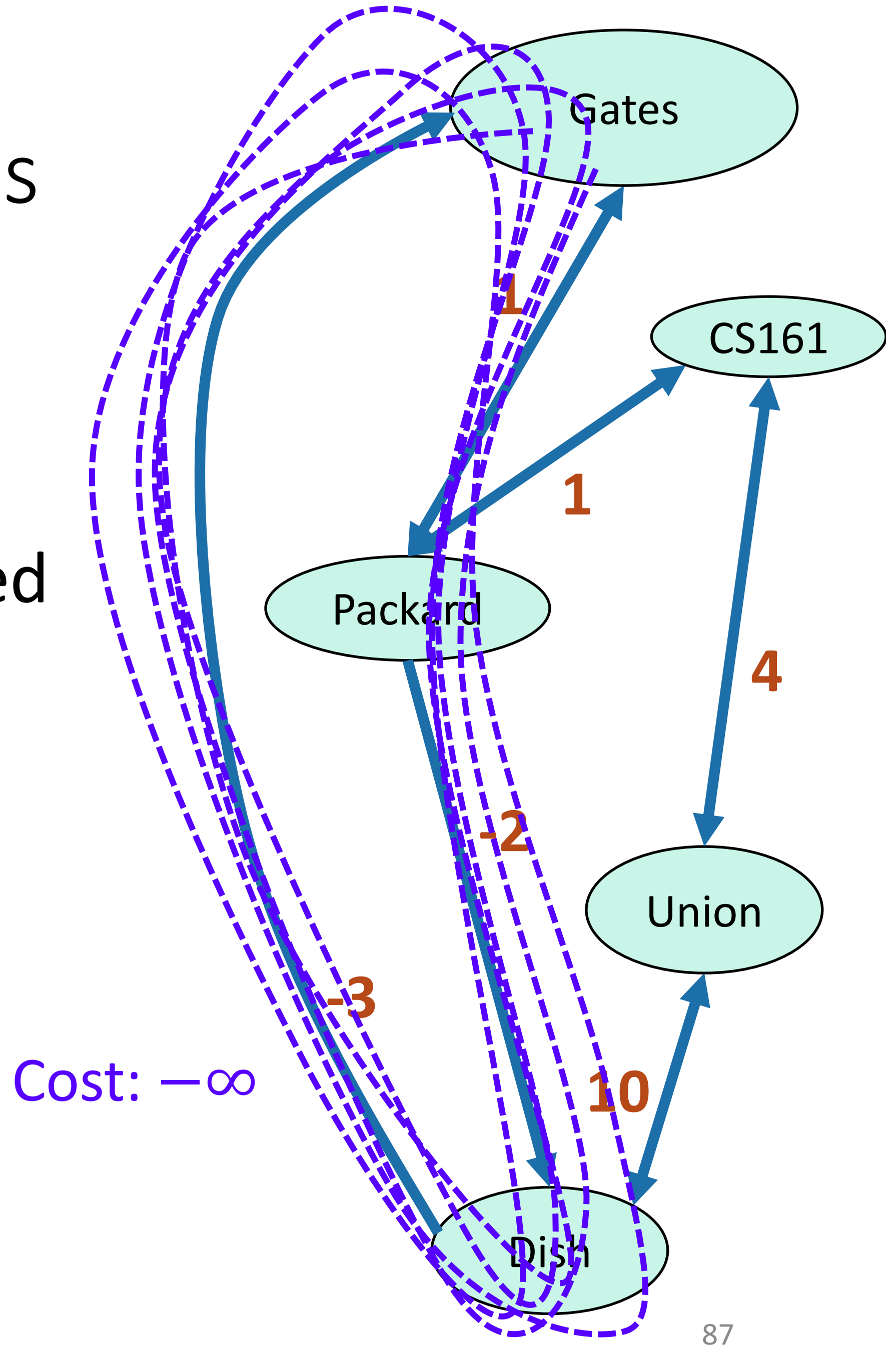
Caution: negative cycles

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



Caution: negative cycles

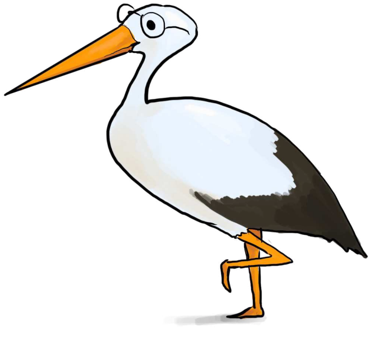
- What is the shortest path from Gates to Old Union?
- Shortest paths aren't defined if there are negative cycles!



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.

Figure out how! (Hint: if there are no negative cycles, the algorithm should stop updating after n-1 iterations. What happens if there are negative cycles?)



Siggi the Studious Stork

Summary

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

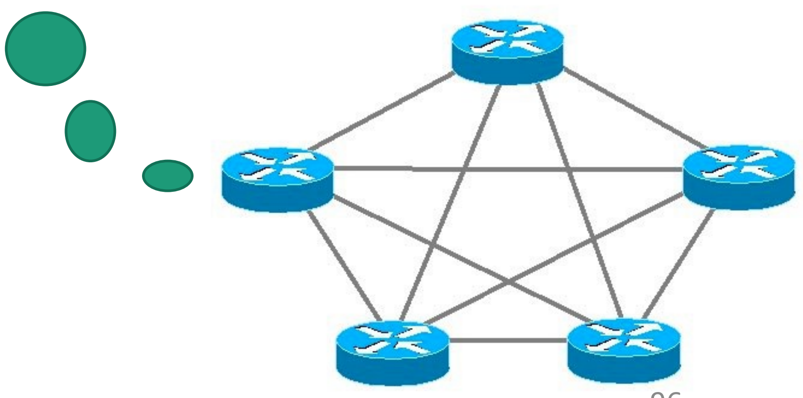
- The Bellman-Ford algorithm:
  - Finds shortest paths in weighted graphs, even with negative edge weights
  - runs in time O(nm) on a graph G with n vertices and m edges.
- If there are no negative cycles in G:
  - the BF algorithm terminates with $d^{(n-1)}[v] = d(s,v)$.
- If there are negative cycles in G:
  - the BF algorithm can be modified to return "`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(nlog(n) + m) if you implement it with a Fibonacci heap
  - (-) no negative edge weights
  - (-) very "centralized" (need to keep track of all the vertices to know which to update).
- 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

- Dynamic Programming!!!

Before next time

- Pre-lecture exercise for Lecture 12
  - Fibonacci numbers!