Lecture 11

Weighted Graphs: Dijkstra and Bellman-Ford

Announcements

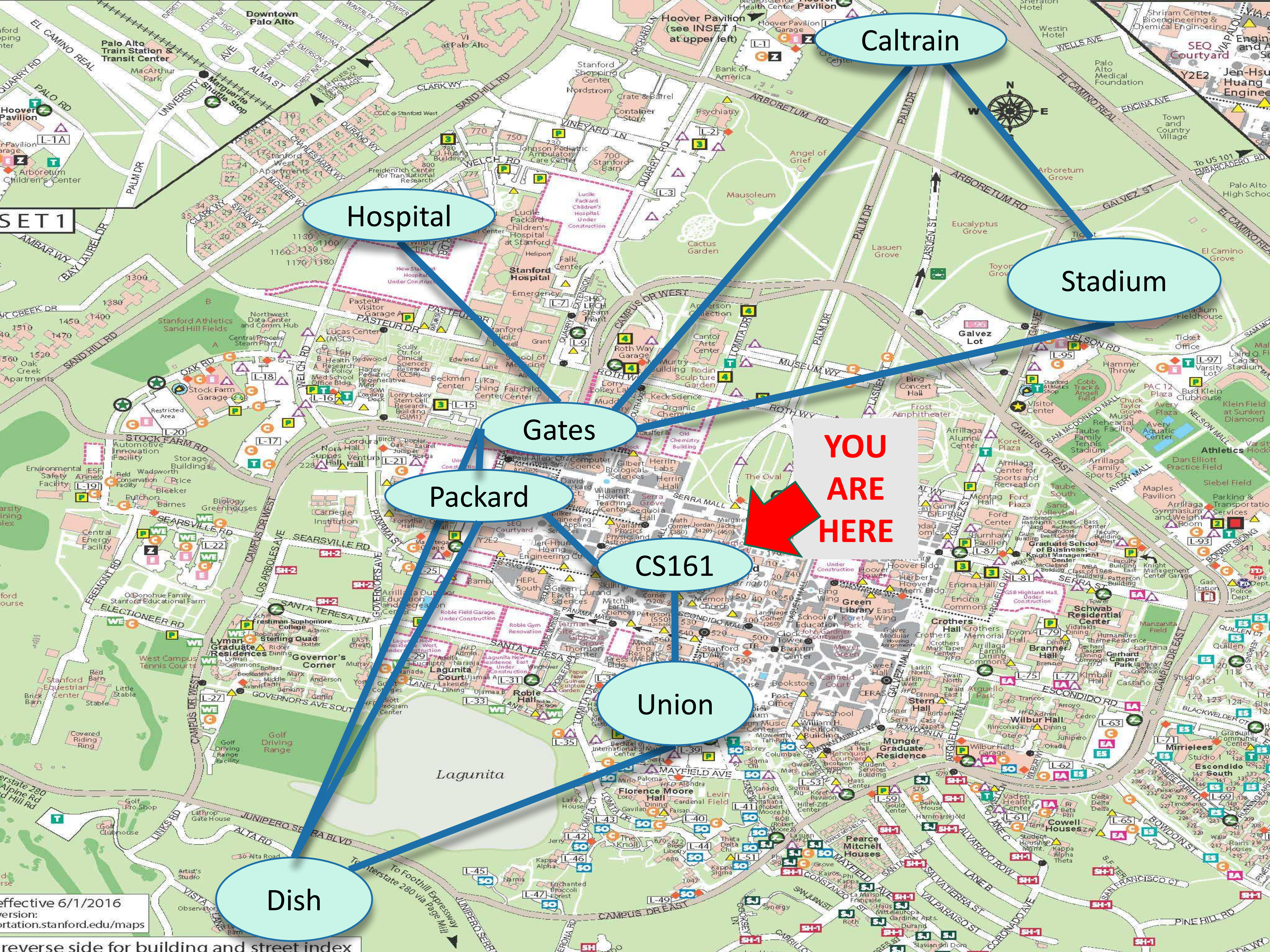
- HW5 will be posted Friday
- We will be doing midterm grading on Sunday.
  - Returned Monday (hopefully)
- The midterm was hard.
  - That’s okay, that’s what the curve is for.

Last week

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

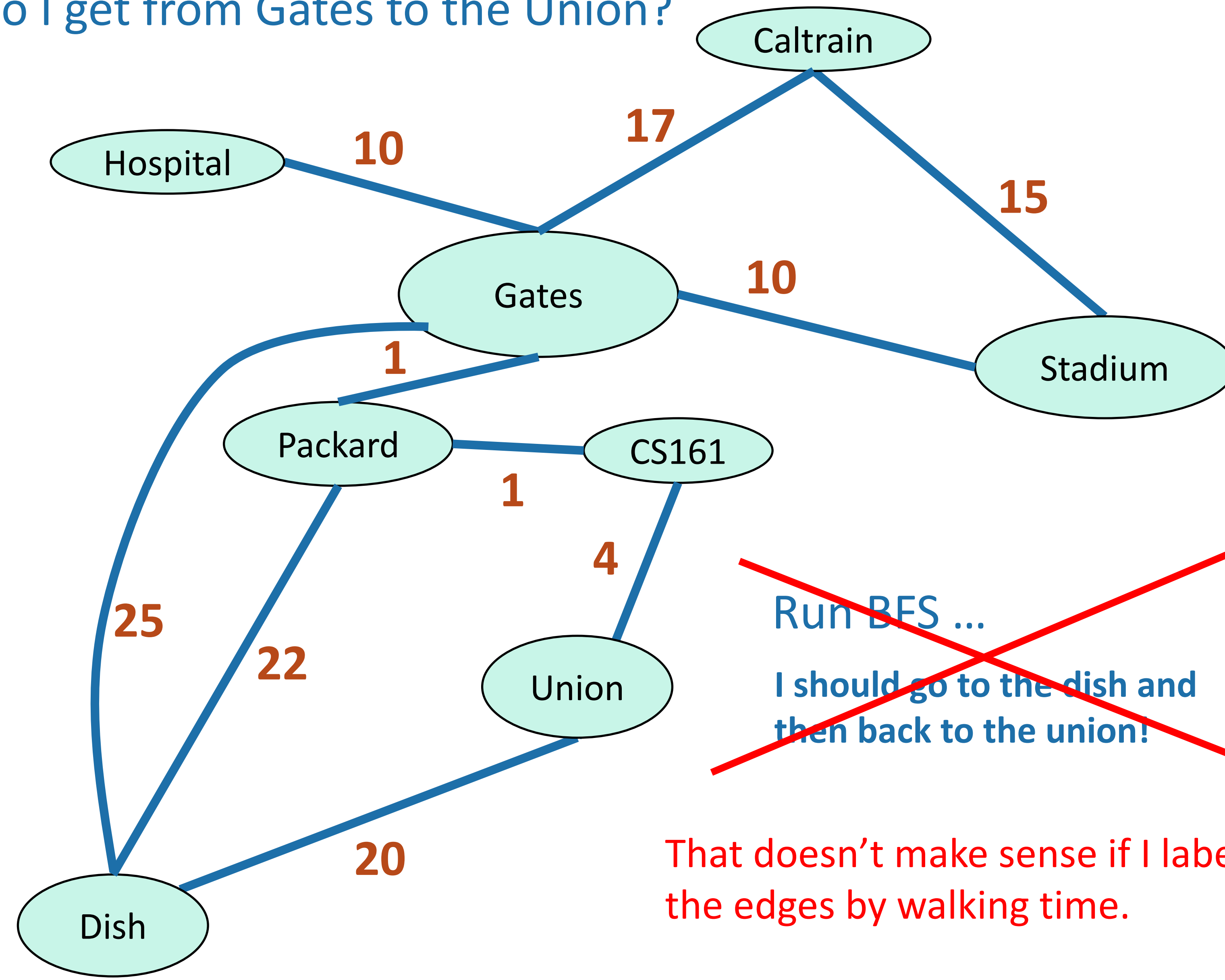
Today

- What if the graphs are weighted?
    - All nonnegative weights: Dijkstra!
    - If there are negative weights: Bellman-Ford!


Caltrain
Hospital
Stadium
Gates
Packard
CS161
Union
Dish
YOU ARE HERE

Just the graph

How do I get from Gates to the Union?



~~Run BFS …~~

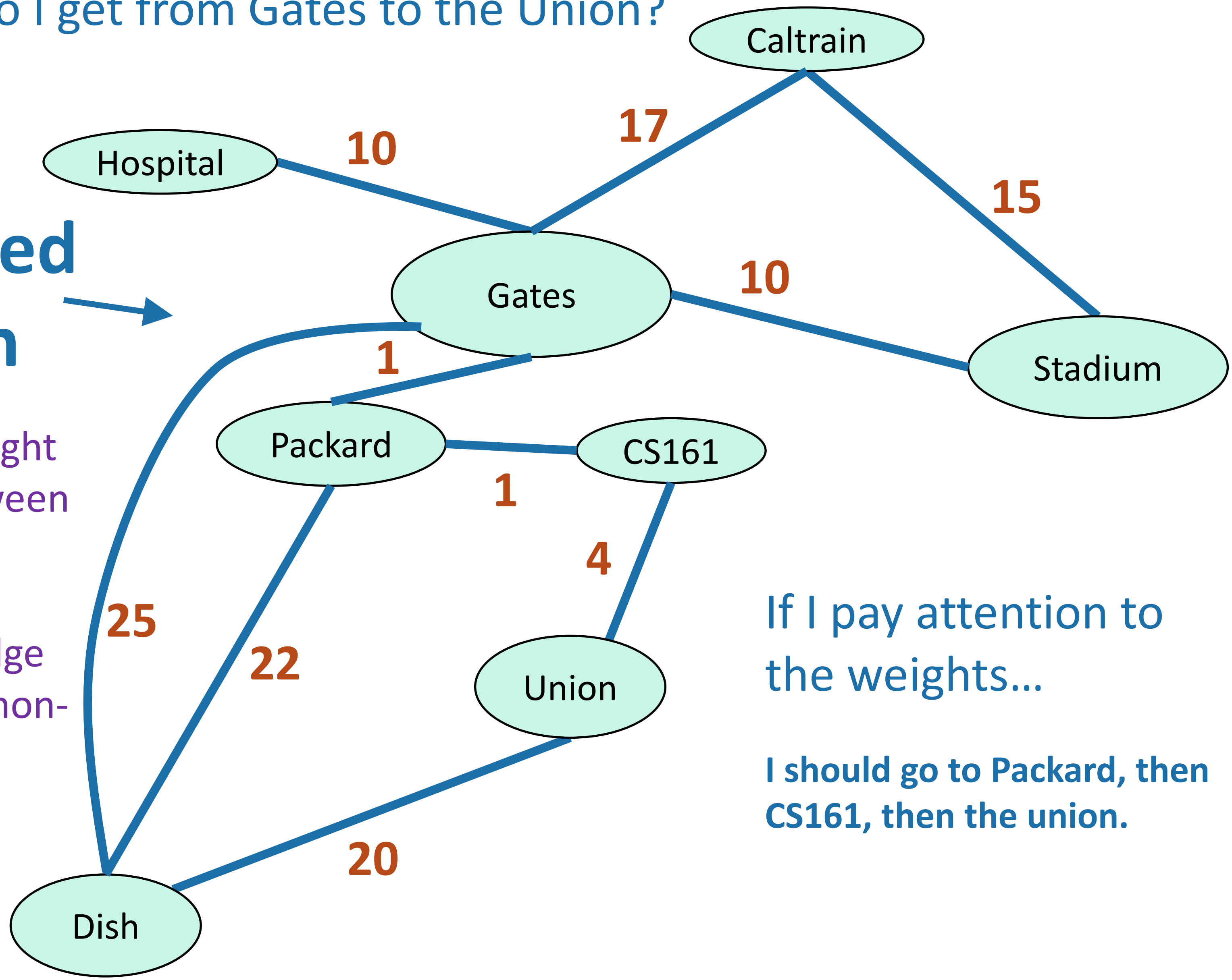
~~**I should go to the dish and then back to the union!**~~

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

Just the graph

How do I get from Gates to the Union?

weighted graph



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

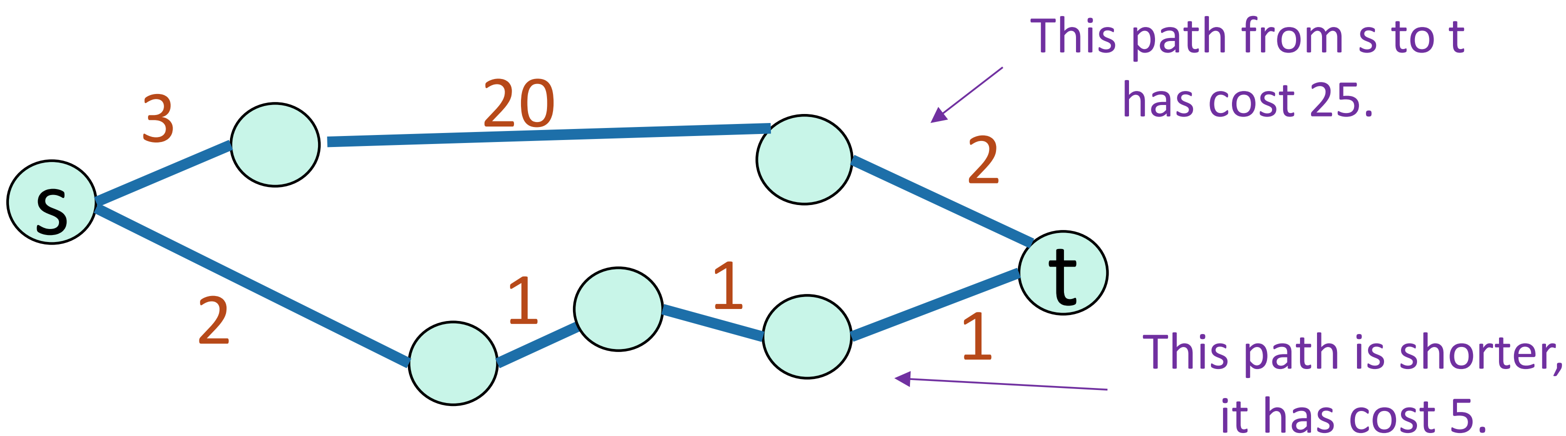
For now, edge weights are non-negative.

If I pay attention to the weights…

I should go to Packard, then CS161, then the union.

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.

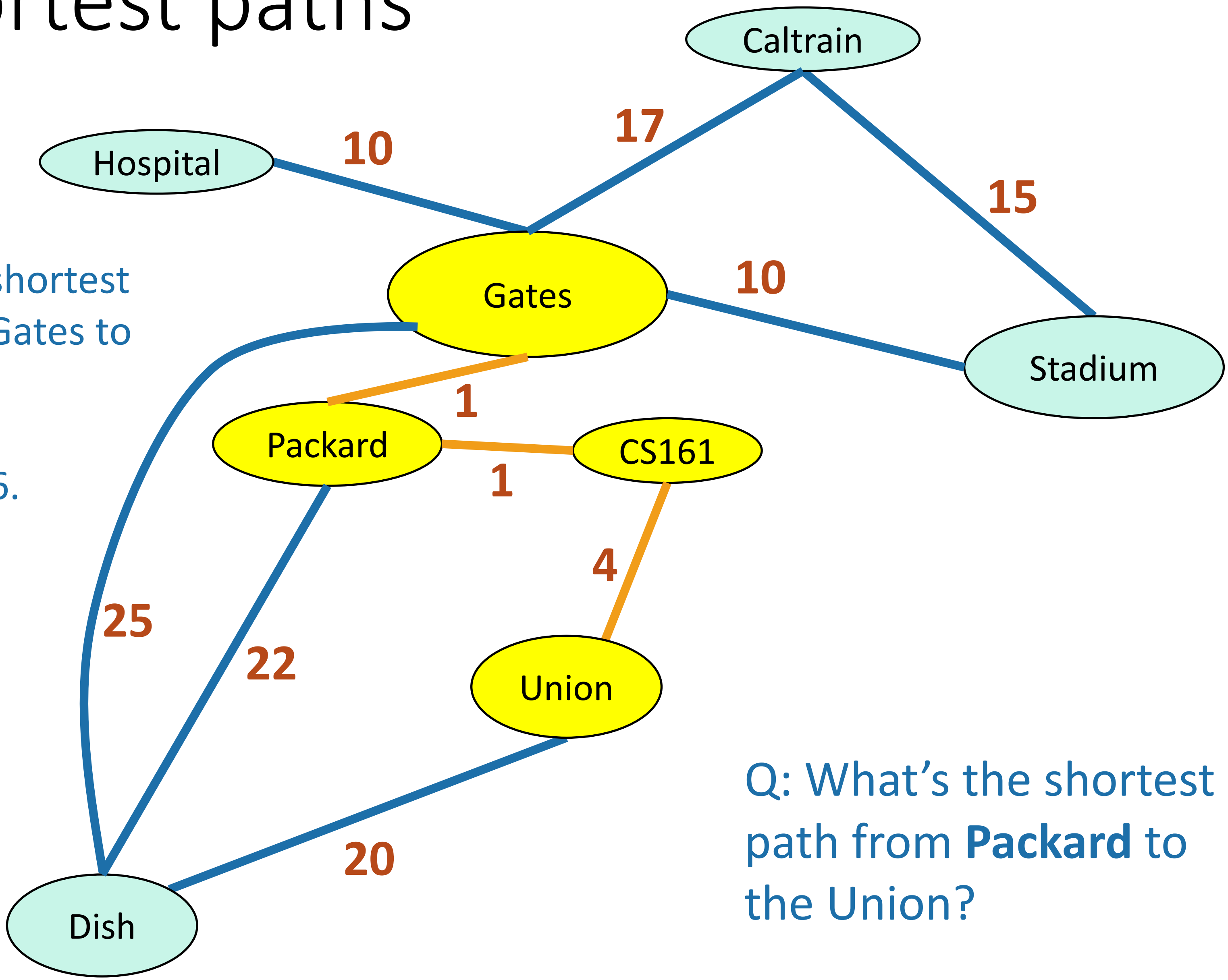


- The **distance** d(u,v) between two vertices u and v is the cost of the the shortest path between u and v.
- For this lecture **all graphs are directed**, but to save on notation I'm just going to draw undirected edges.

Shortest paths

This is the shortest path from Gates to the Union.

It has cost 6.

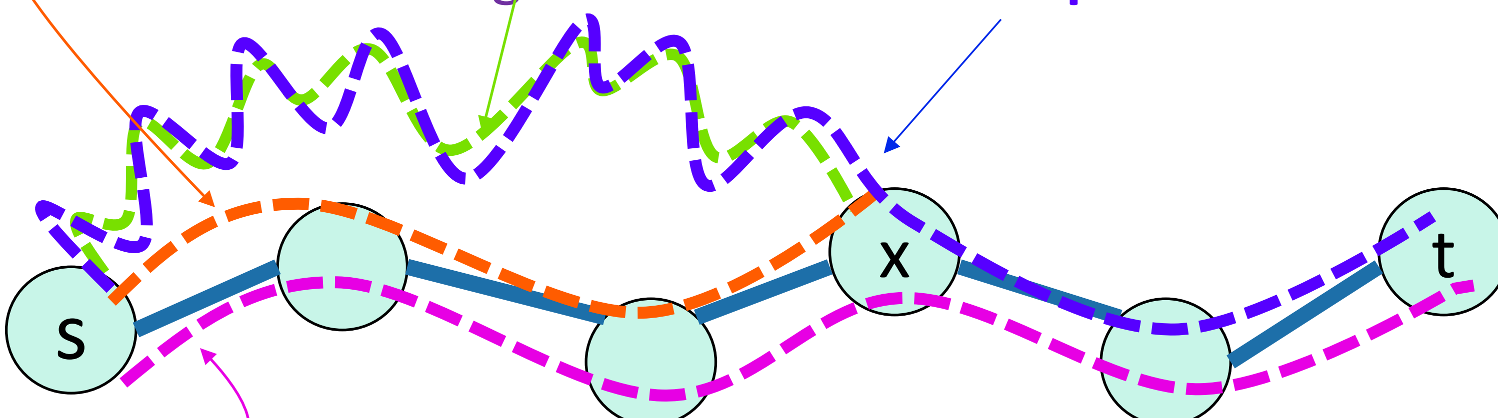


Q: What's the shortest path from **Packard** to the Union?

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 shorter.
  - But then that gives an **even shorter path** from s to t!



Single-source shortest-path problem

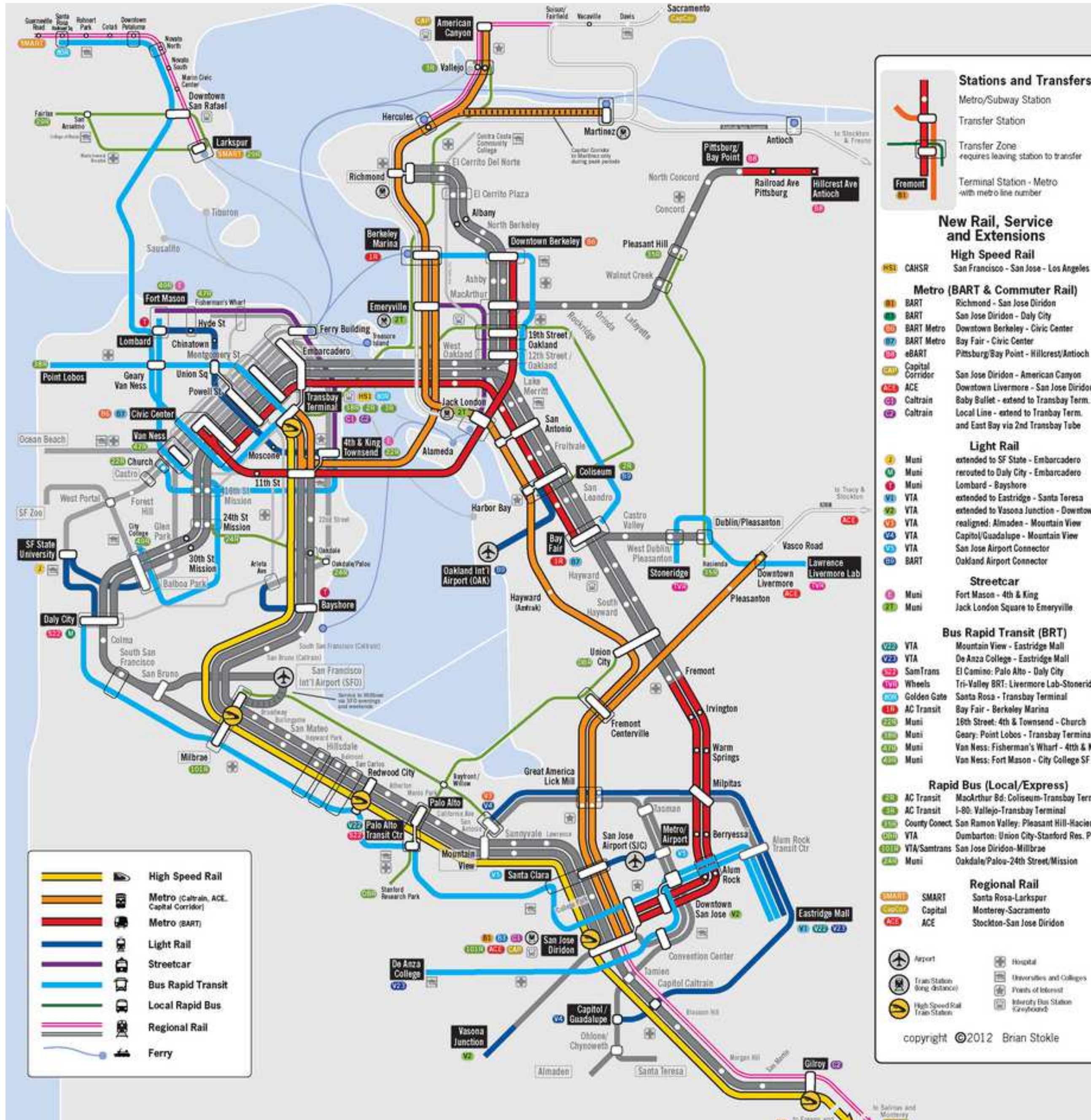
- I want to know the shortest path from one vertex (Gates) to all other vertices.

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)

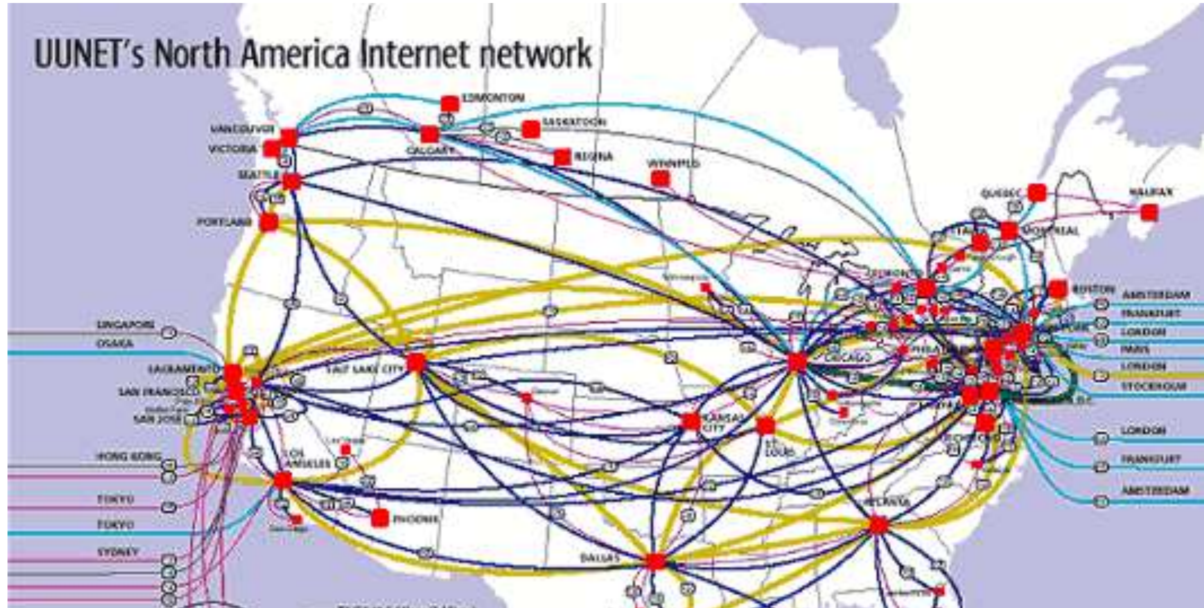
Example

- I regularly have to solve “**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. (They also change depending on my mood and traffic…).



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?



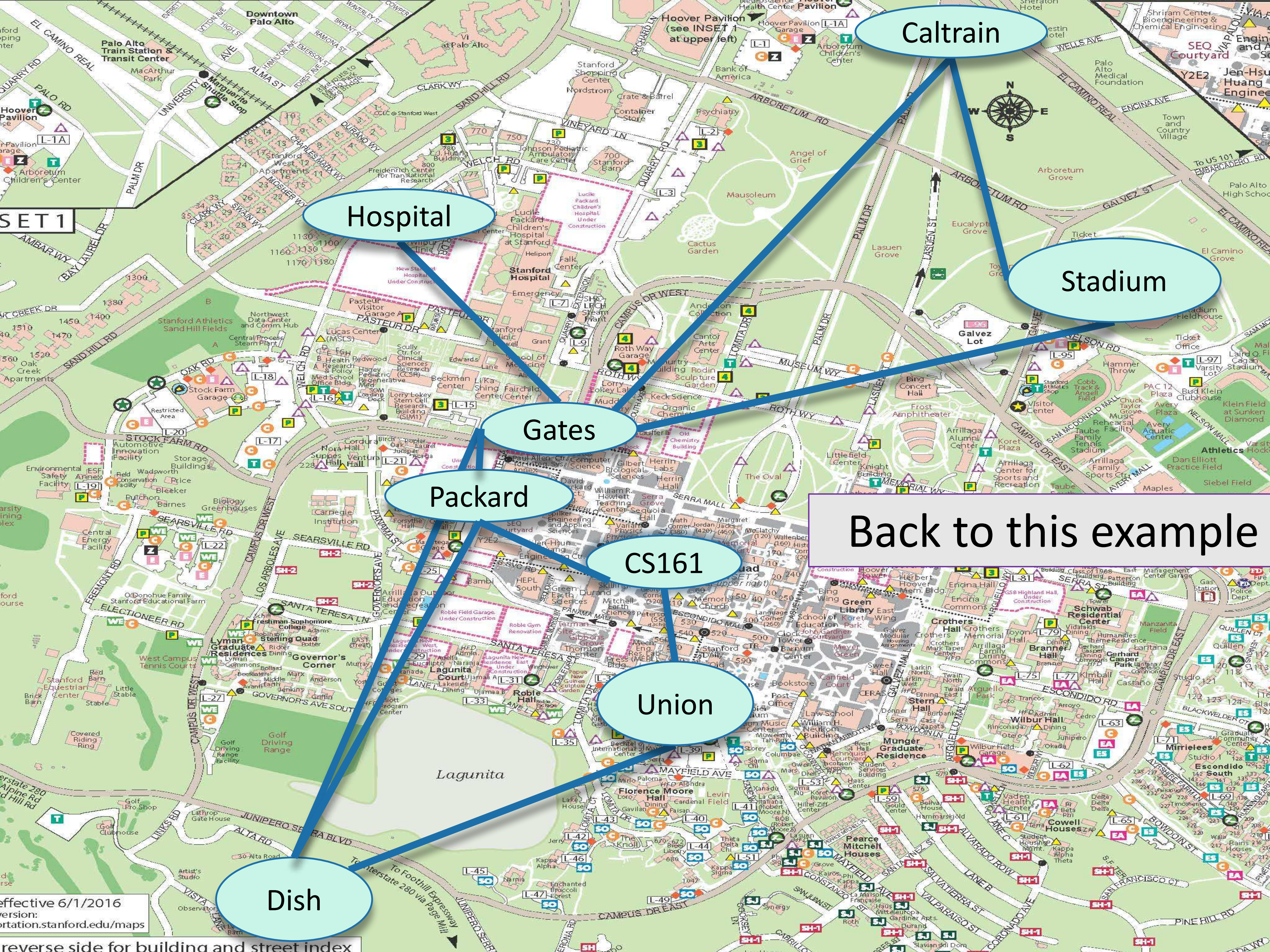

```
DN0a22a0e3:~ mary$ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1  [AS0] 10.34.160.2 (10.34.160.2)  38.168 ms  31.272 ms  28.841 ms
 2  [AS0] cwa-vrtr.sunet (10.21.196.28)  33.769 ms  28.245 ms  24.373 ms
 3  [AS32] 171.66.2.229 (171.66.2.229)  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  4.9
 6  [AS2152] hpr-lax-hpr3--svl-hpr3-100ge.cenic.net (137.164.25.73)  12.125 m
 7  [AS2152] hpr-i2--lax-hpr2-r&e.cenic.net (137.164.26.201)  40.174 ms  38.3
 8  [AS0] et-4-0-0.4079.sdn-sw.lasv.net.internet2.edu (162.252.70.28)  46.573
 9  [AS0] et-5-1-0.4079.rtsw.salt.net.internet2.edu (162.252.70.31)  30.424 m
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 m
12  [AS0] et-4-1-0.4070.rtsw.chic.net.internet2.edu (198.71.47.206)  77.937 m
13  [AS0] et-0-1-0.4079.sdn-sw.ashb.net.internet2.edu (162.252.70.60)  77.682
14  [AS0] et-4-1-0.4079.rtsw.wash.net.internet2.edu (162.252.70.65)  71.565 m
15  [AS21320] internet2-gw.mx1.lon.uk.geant.net (62.40.124.44)  154.926 ms  1
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.
20  [AS559] swizh1-100ge-0-1-0-1.switch.ch (130.59.36.94)  165.824 ms  164.21
21  [AS559] swiez3-100ge-0-1-0-4.switch.ch (130.59.38.109)  164.269 ms  164.3
22  [AS559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1)  164.082 ms  17
23  [AS559] rou-fw-rz-rz-gw.ethz.ch (192.33.92.169)  164.773 ms  165.193 ms
```

Aside: These are difficult problems

- Costs may change
  - If it's raining the cost of biking is higher
  - If a link is congested, the cost of routing a packet along it is higher
- The network might not be known
  - My computer doesn't store a map of the internet
- We want to do these tasks really quickly
  - I have time to bike to Berkeley, but not to *contemplate* biking to Berkeley…
  - More seriously, **the internet.**

This is a joke.

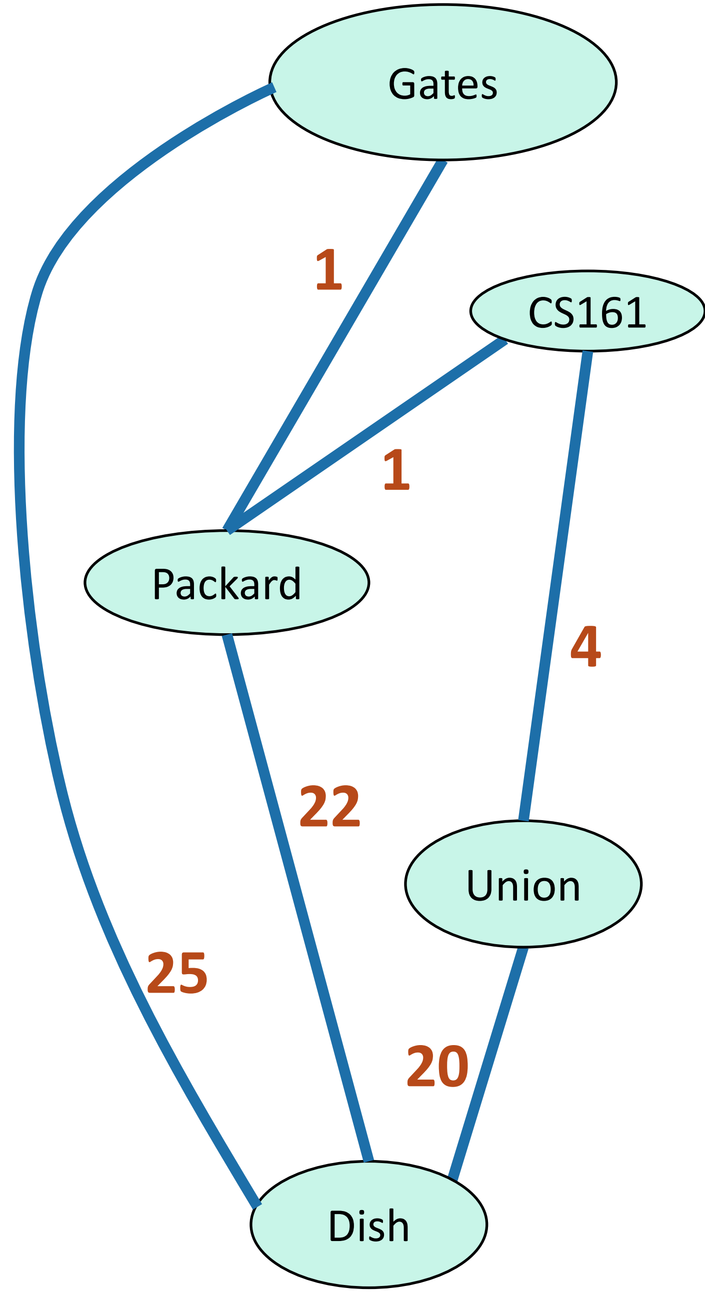
But let's ignore them for now.



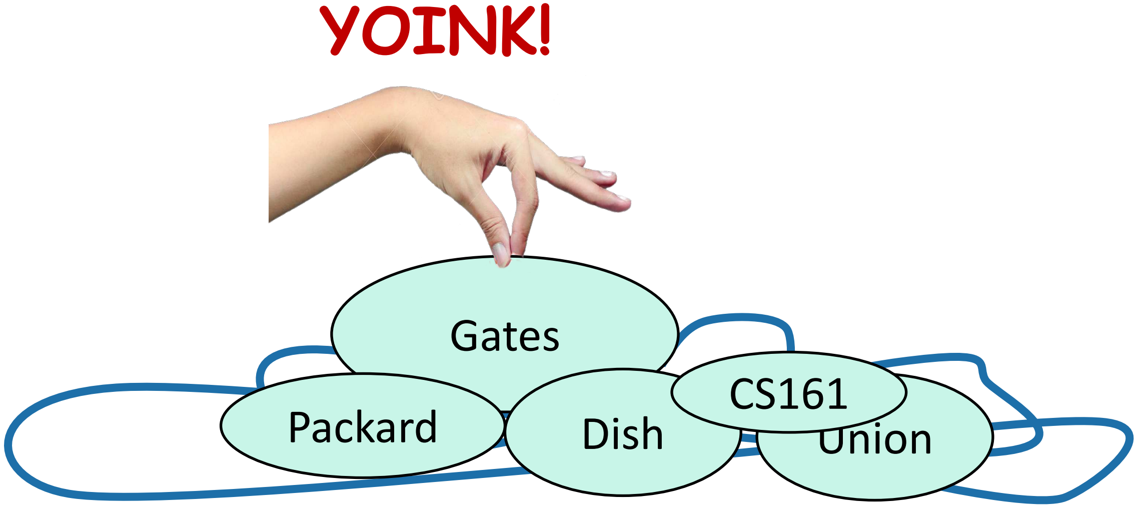
Back to this example

Dijkstra's algorithm

- What are the shortest paths from Gates to everywhere else?

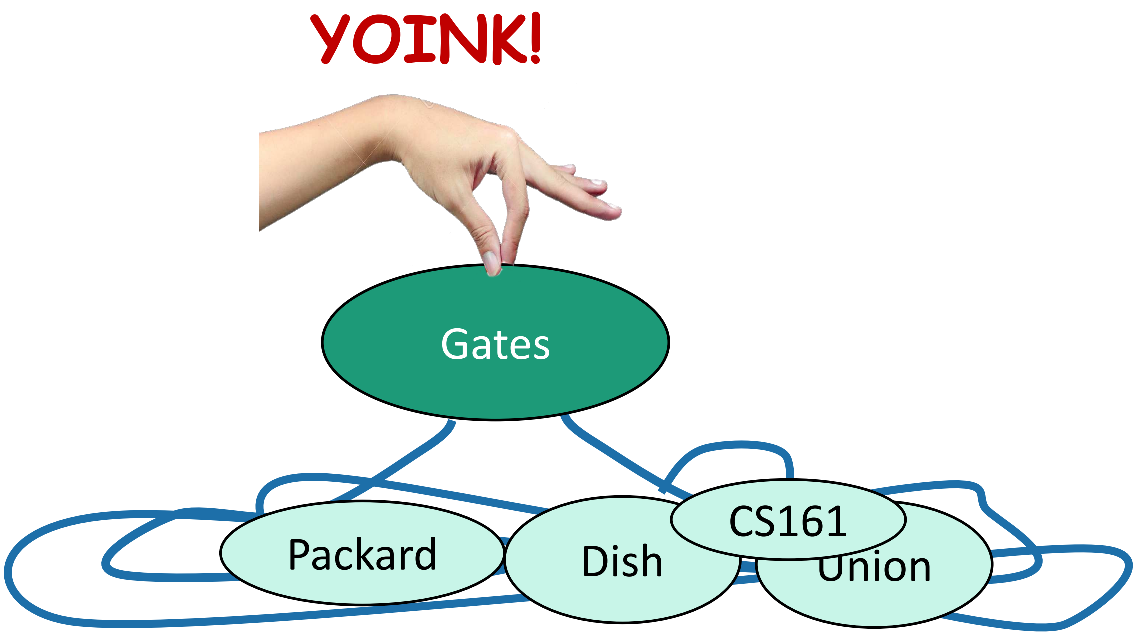


Dijkstra
intuition

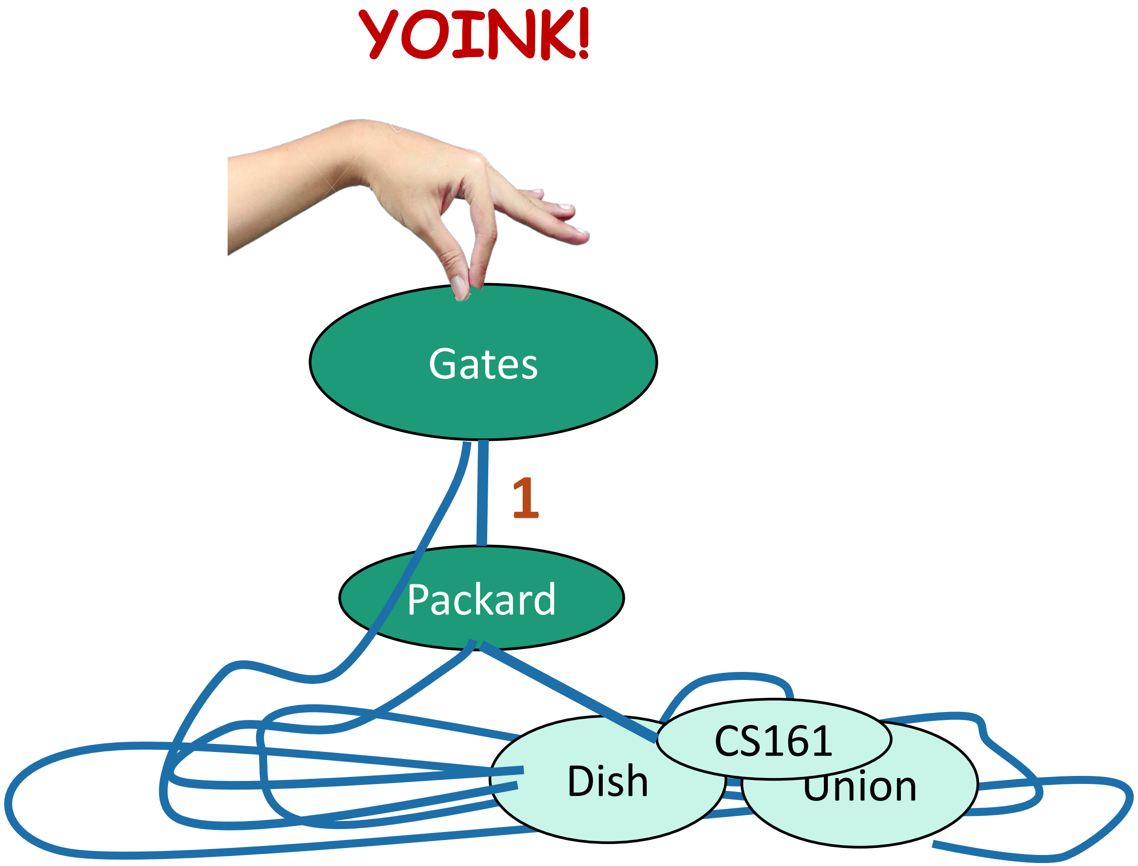


Dijkstra
intuition

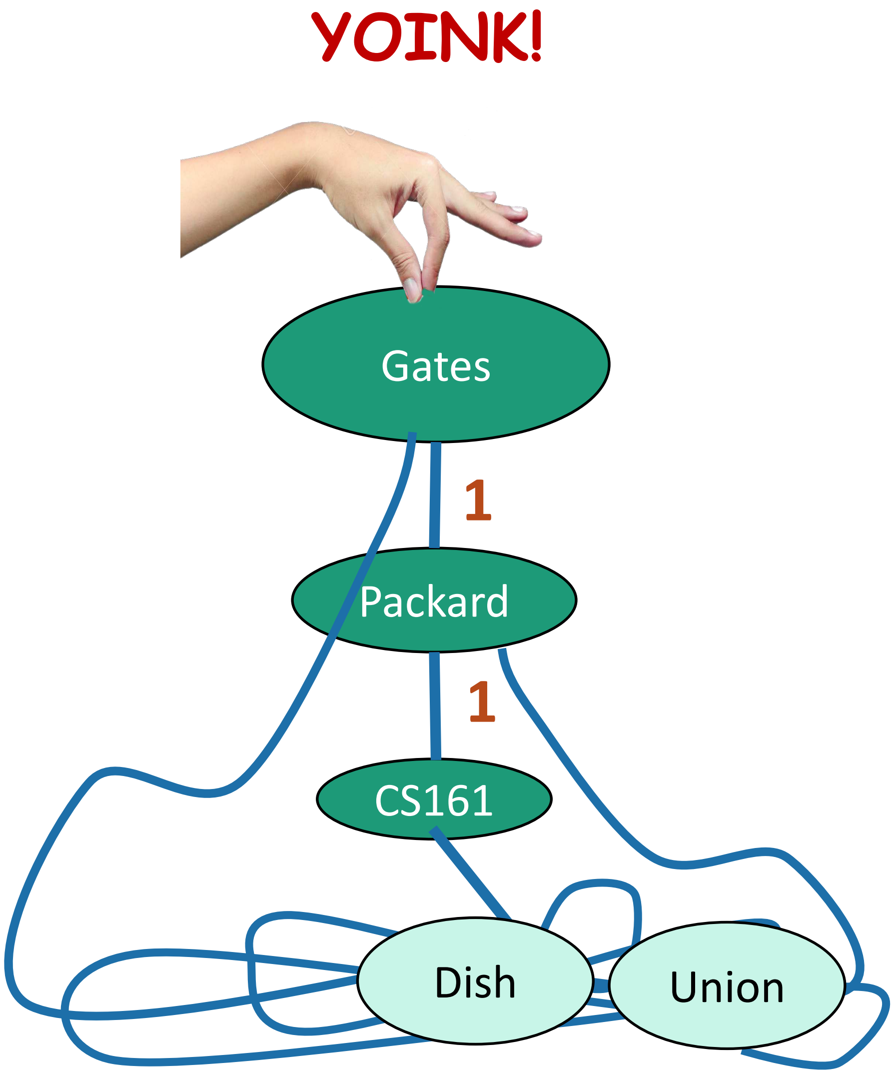
A vertex is done when it's not on the ground anymore.



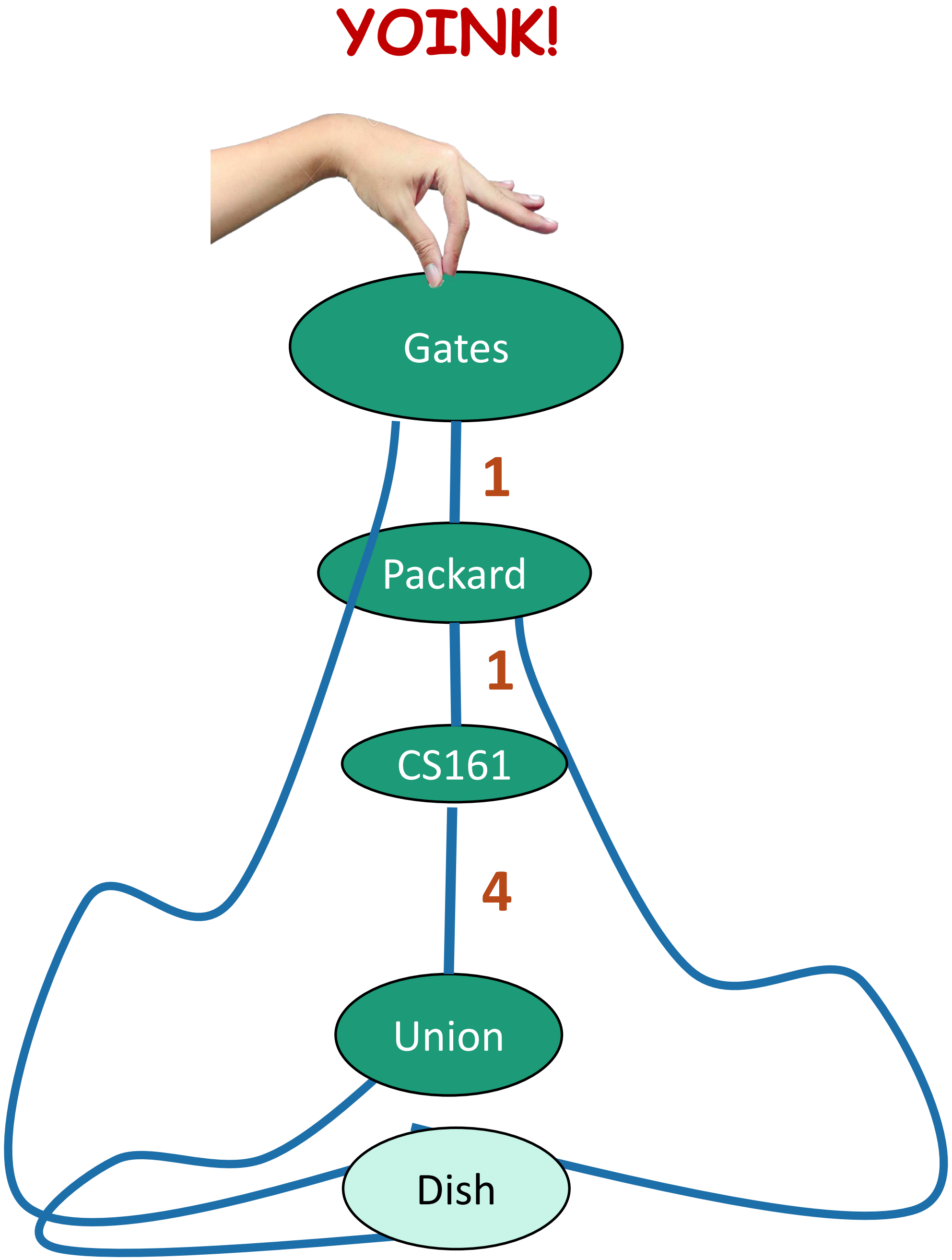
Dijkstra
intuition



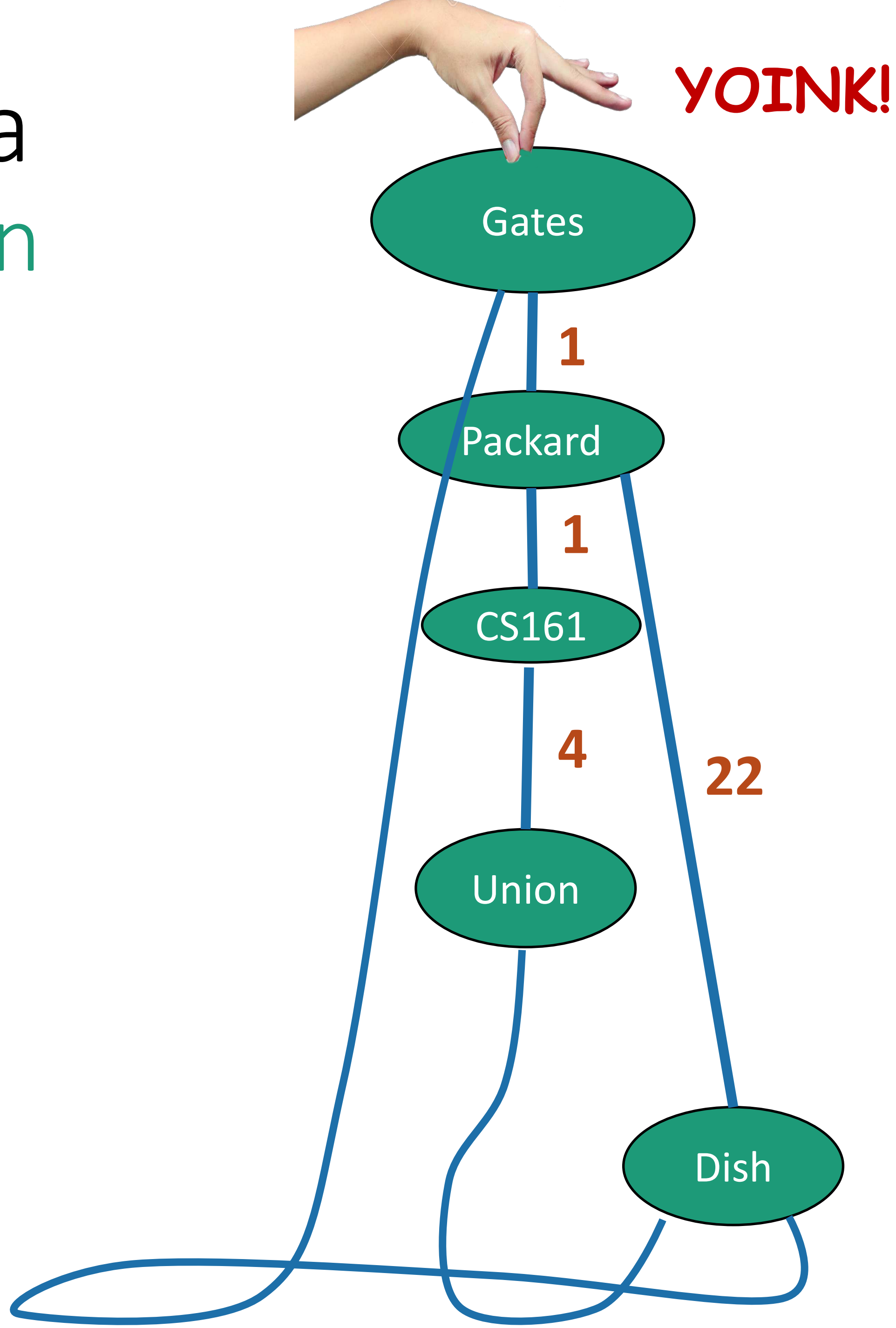
Dijkstra
intuition



Dijkstra
intuition



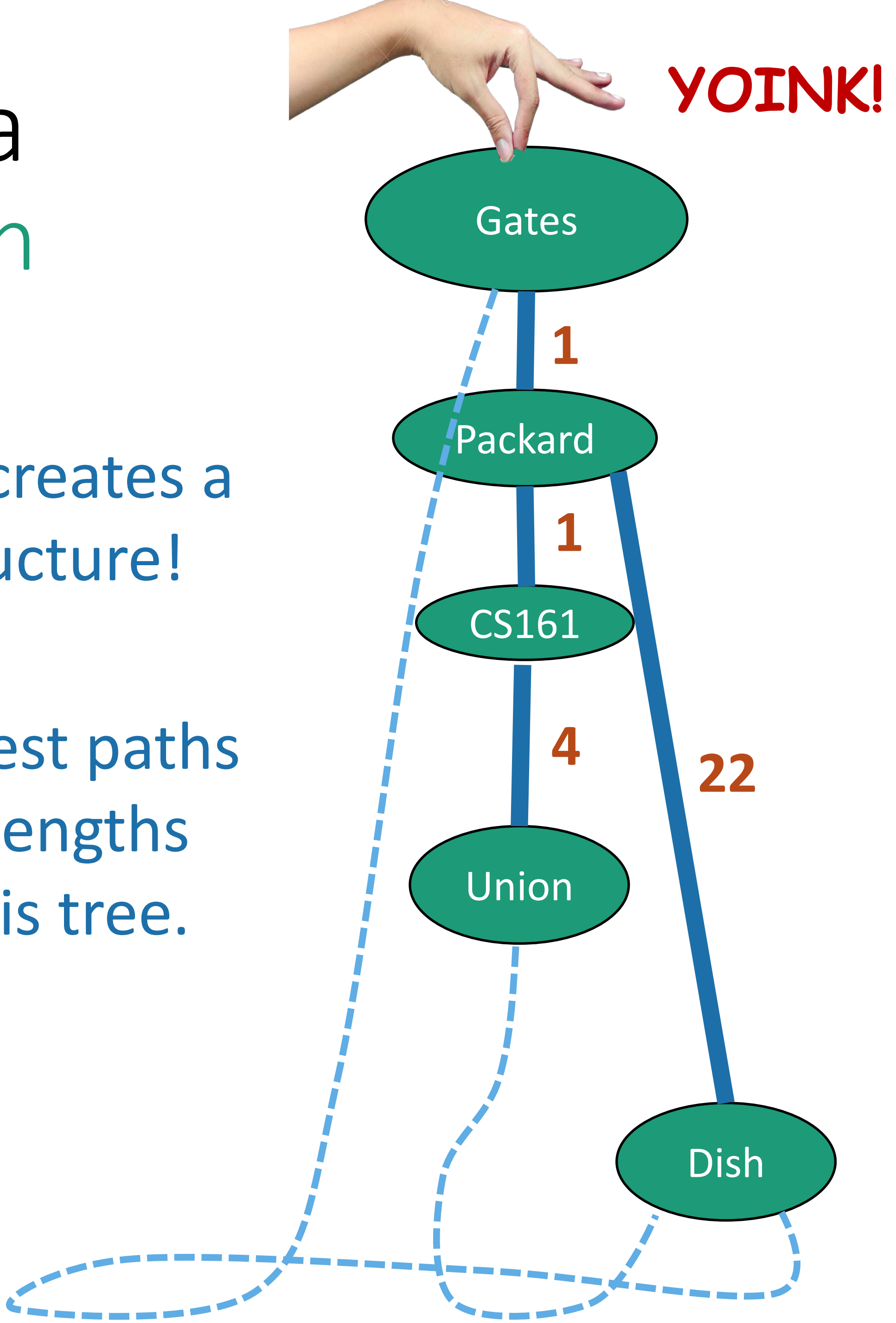
Dijkstra
intuition



Dijkstra intuition

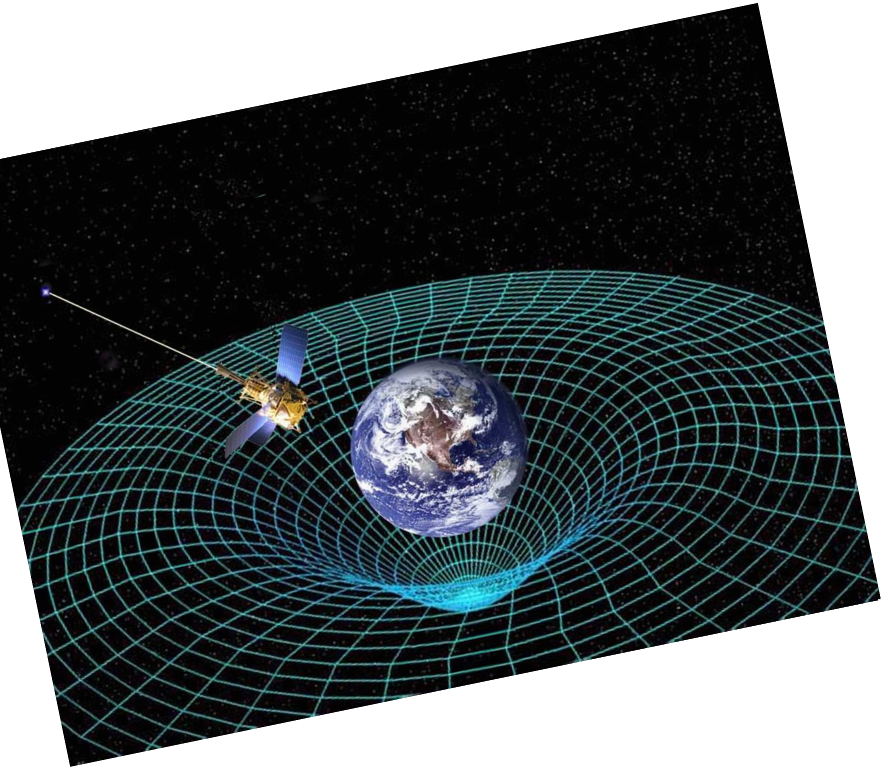
This also creates a tree structure!

The shortest paths are the lengths along this tree.



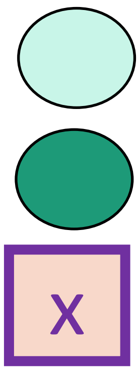
How do we actually implement this?

- **Without** string and gravity?



Dijkstra by example

How far is a node from Gates?



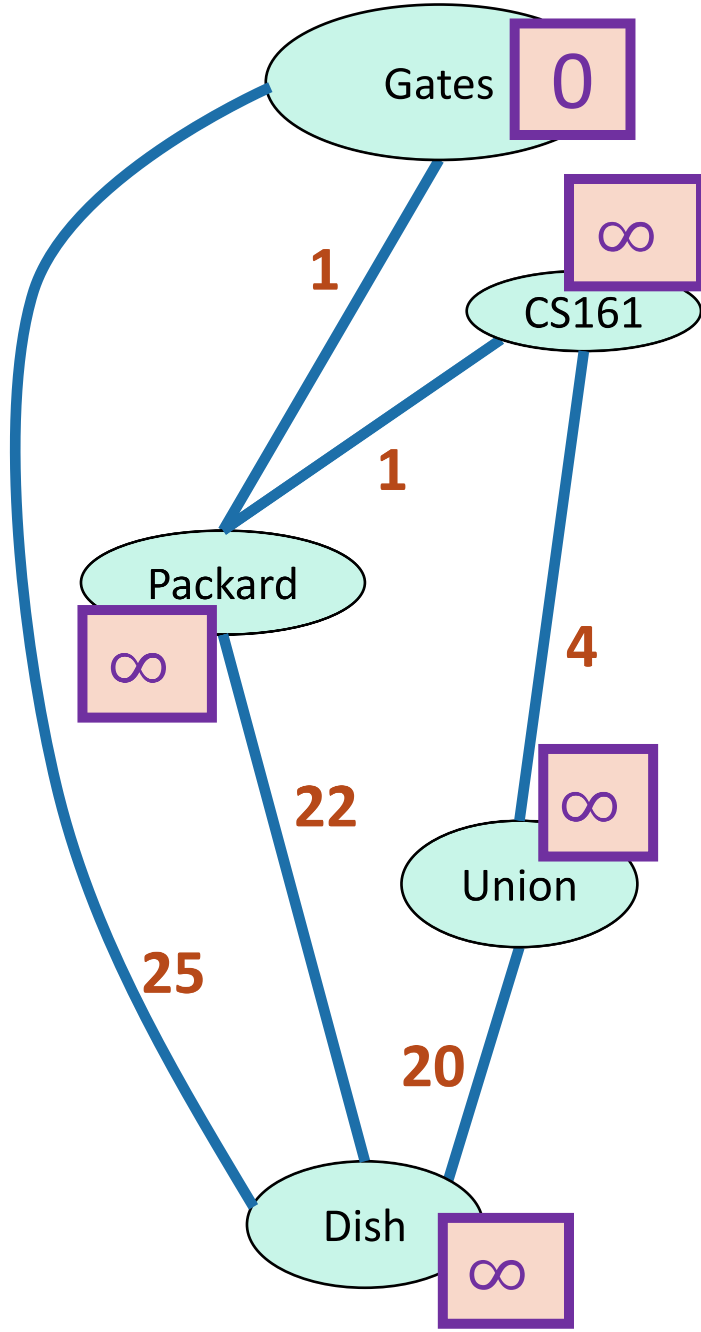
I’m not sure yet

I’m sure

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

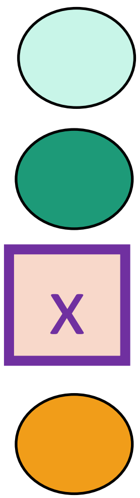
Initialize d[v] = ∞
for all non-starting vertices
v, and d[Gates] = 0

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



Dijkstra by example

How far is a node from Gates?



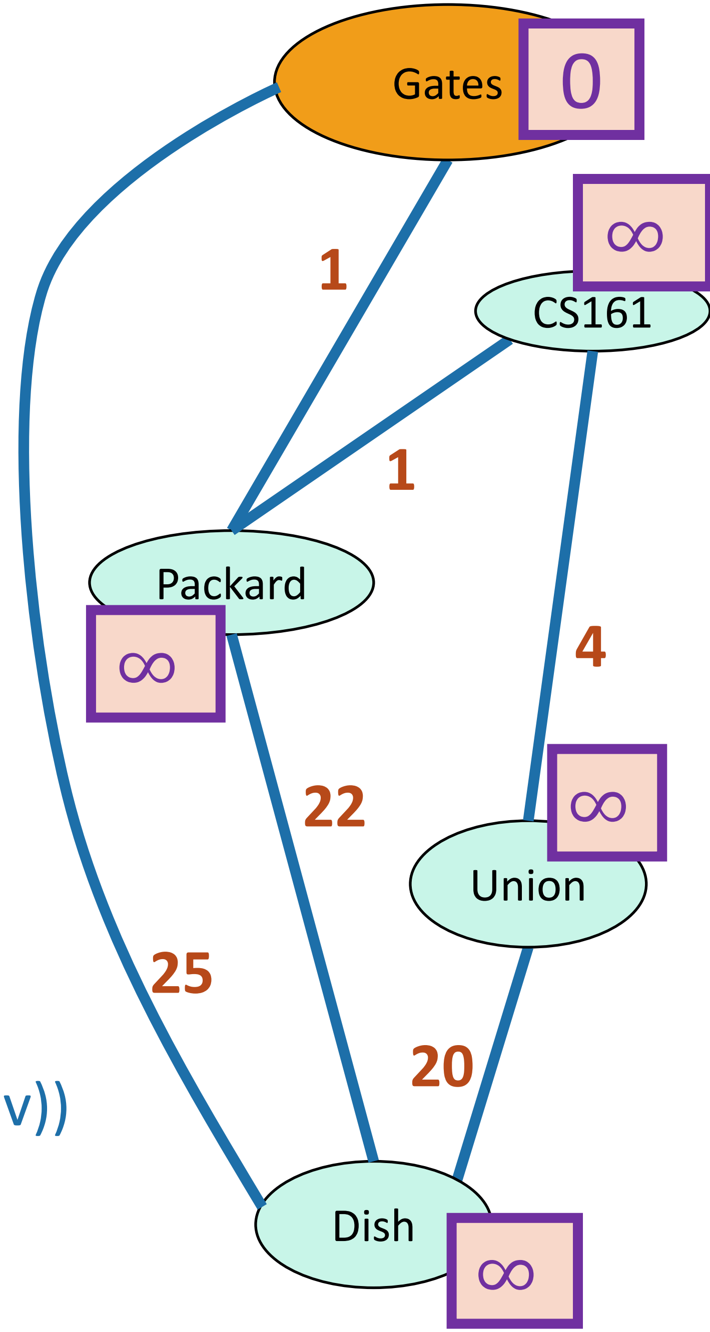
I'm not sure yet

I'm sure

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

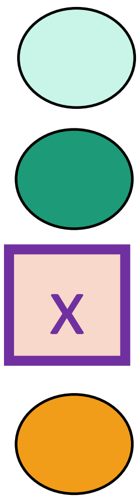
Current node u

- 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?



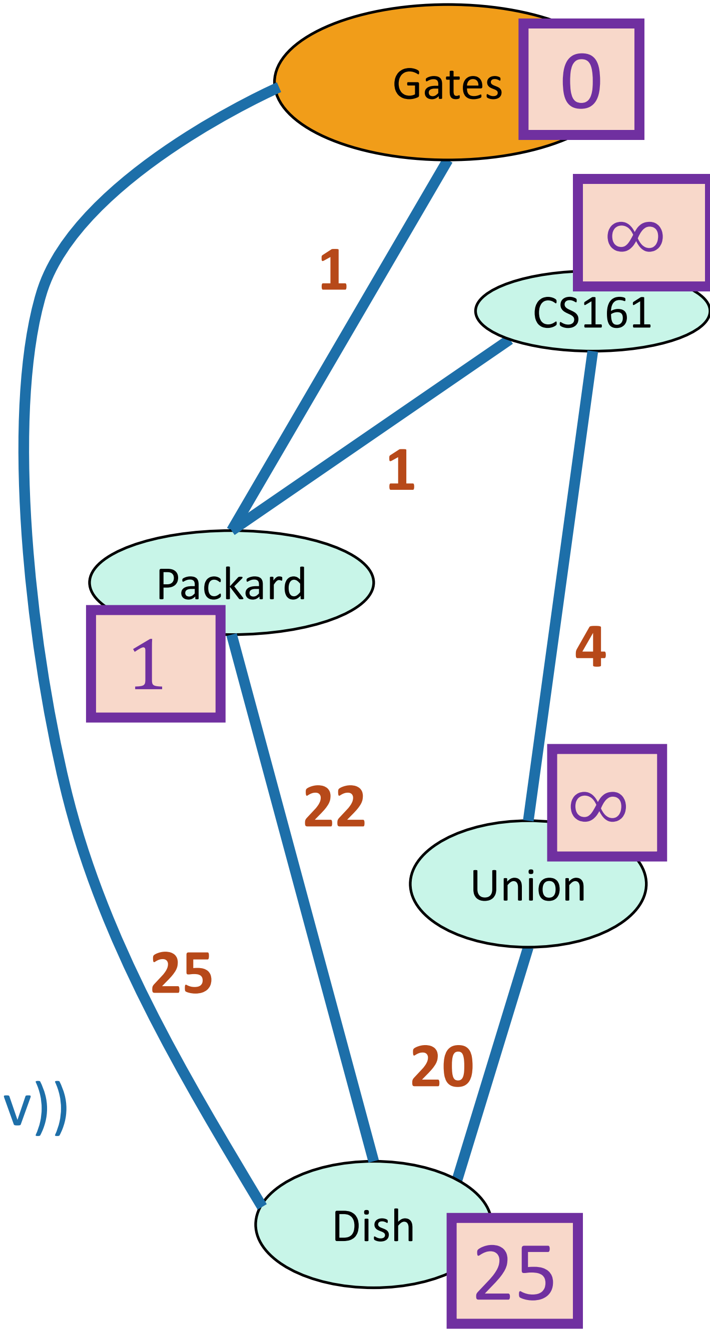
I'm not sure yet

I'm sure

x = 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

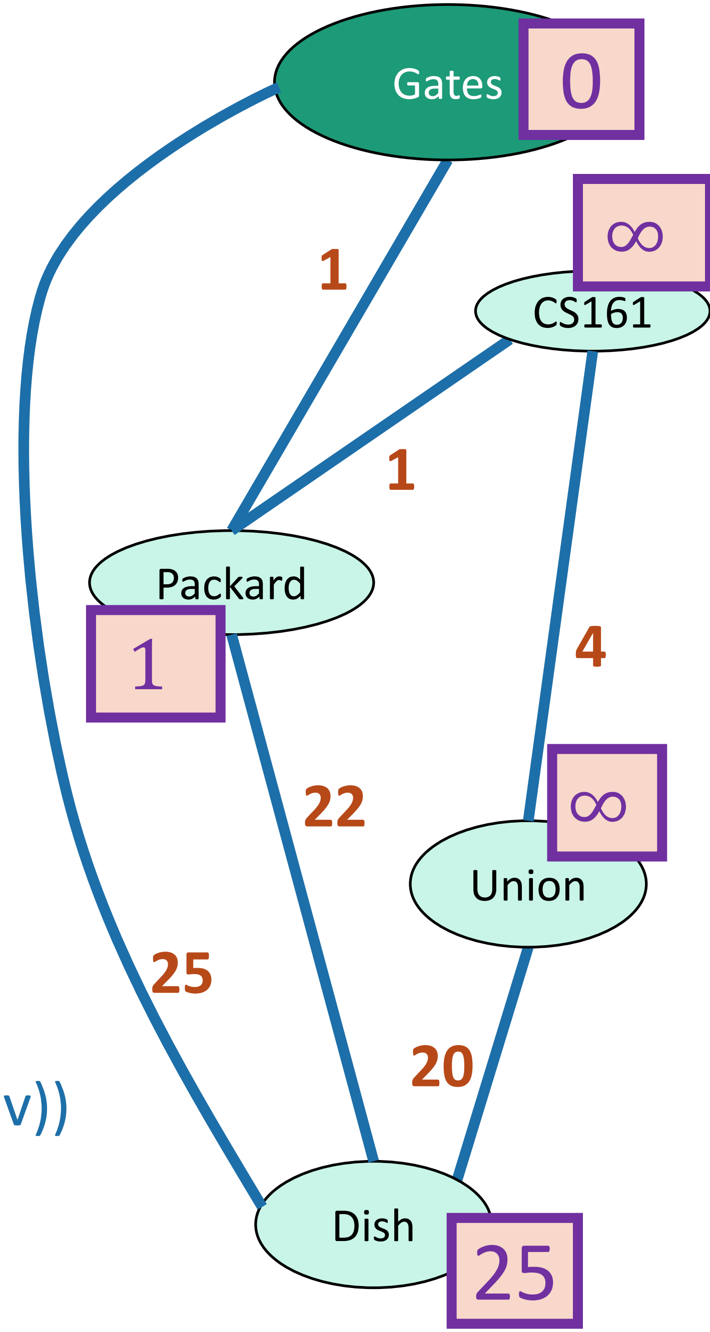
I'm sure

x

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

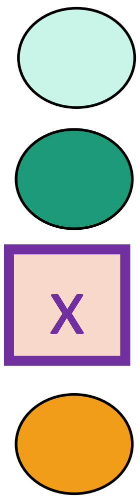
Current node u

- 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?



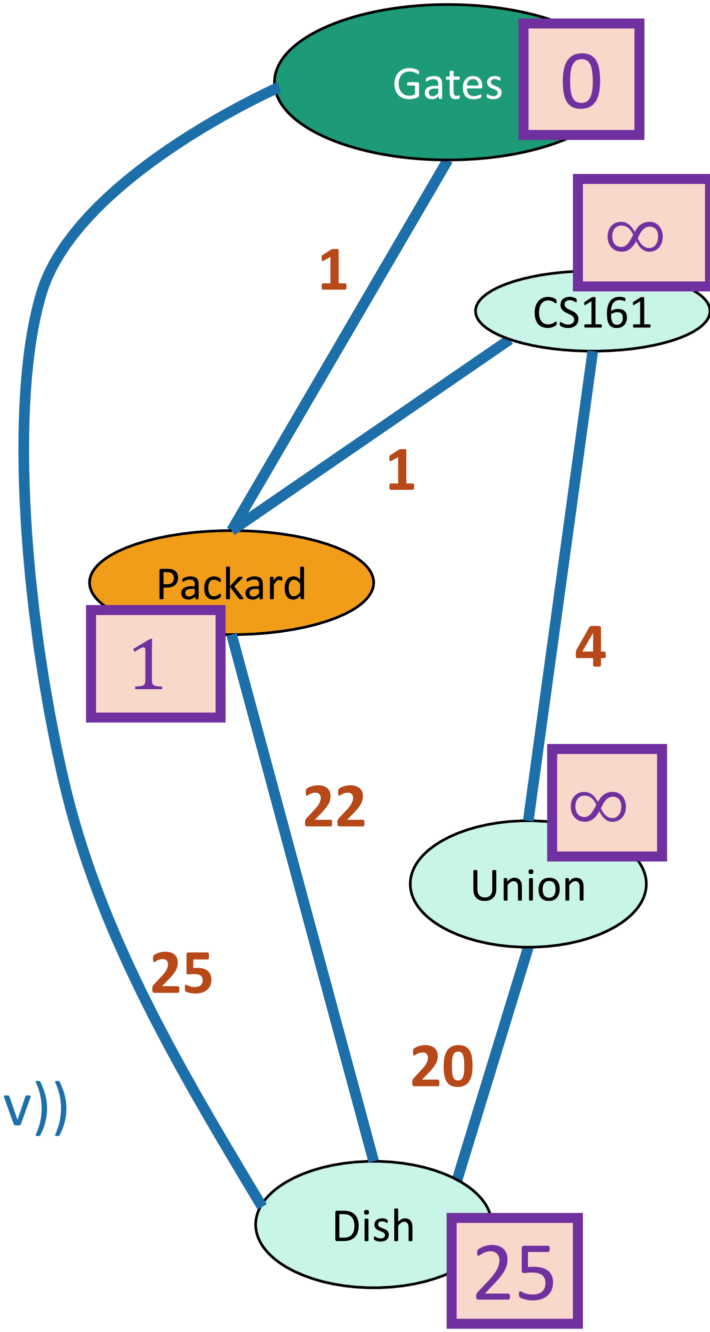
I'm not sure yet

I'm sure

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

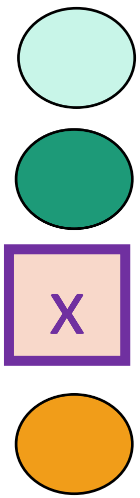
Current node u

- 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?



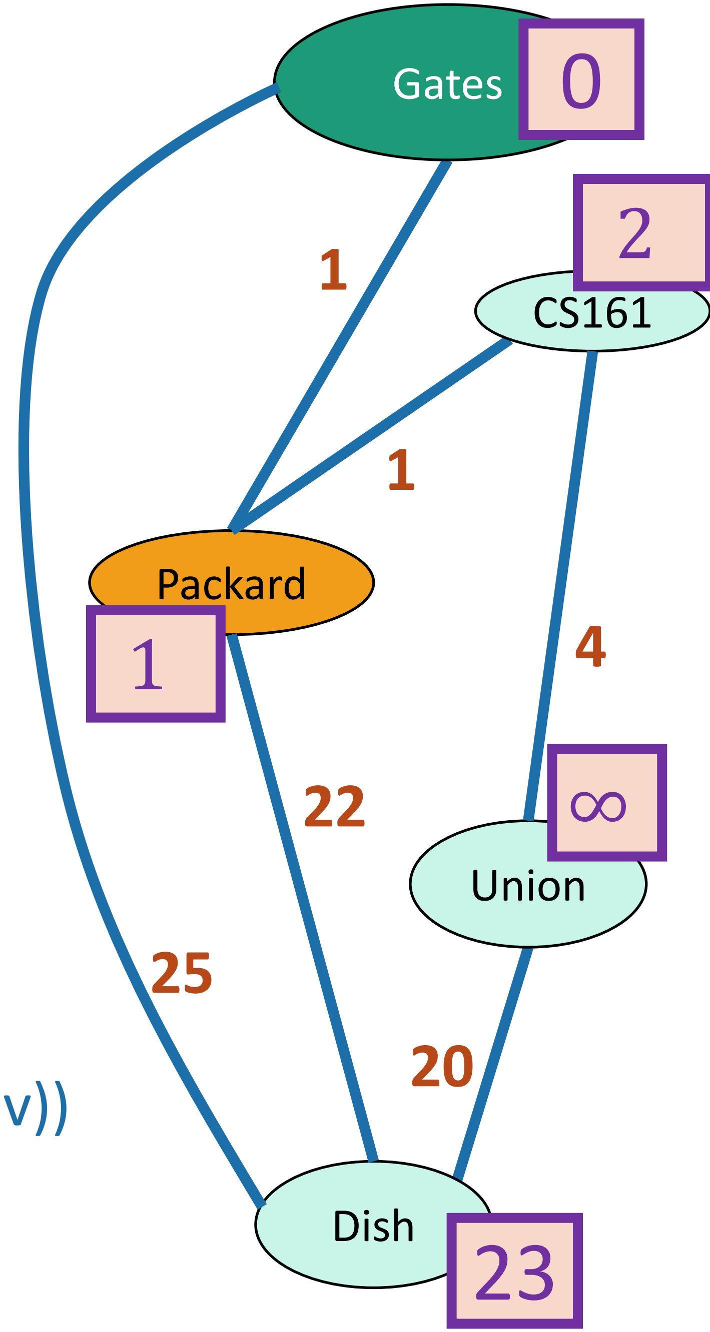
I'm not sure yet

I'm sure

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

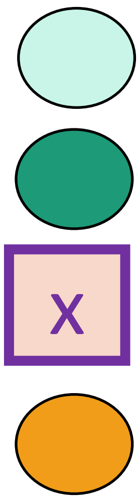
Current node u

- 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?



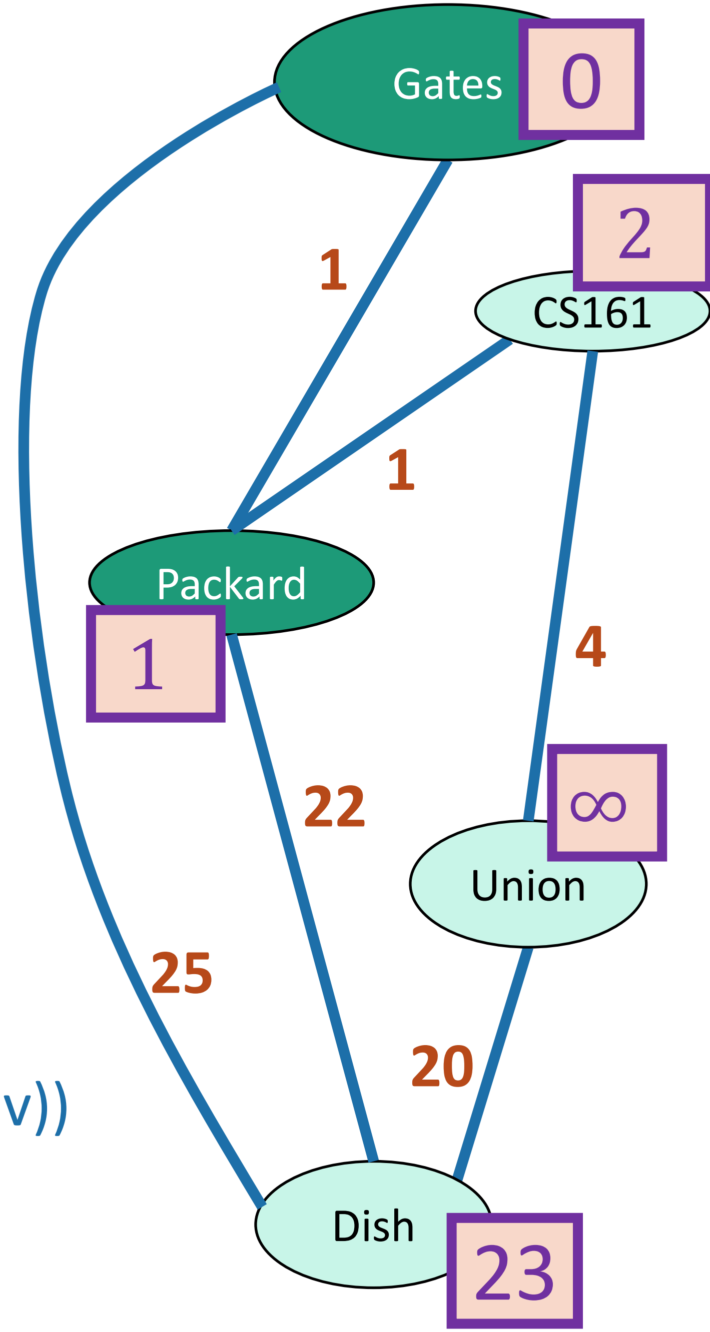
I'm not sure yet

I'm sure

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

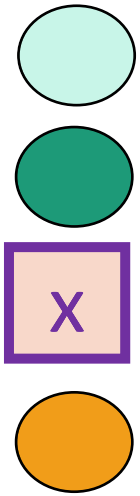
Current node u

- 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?



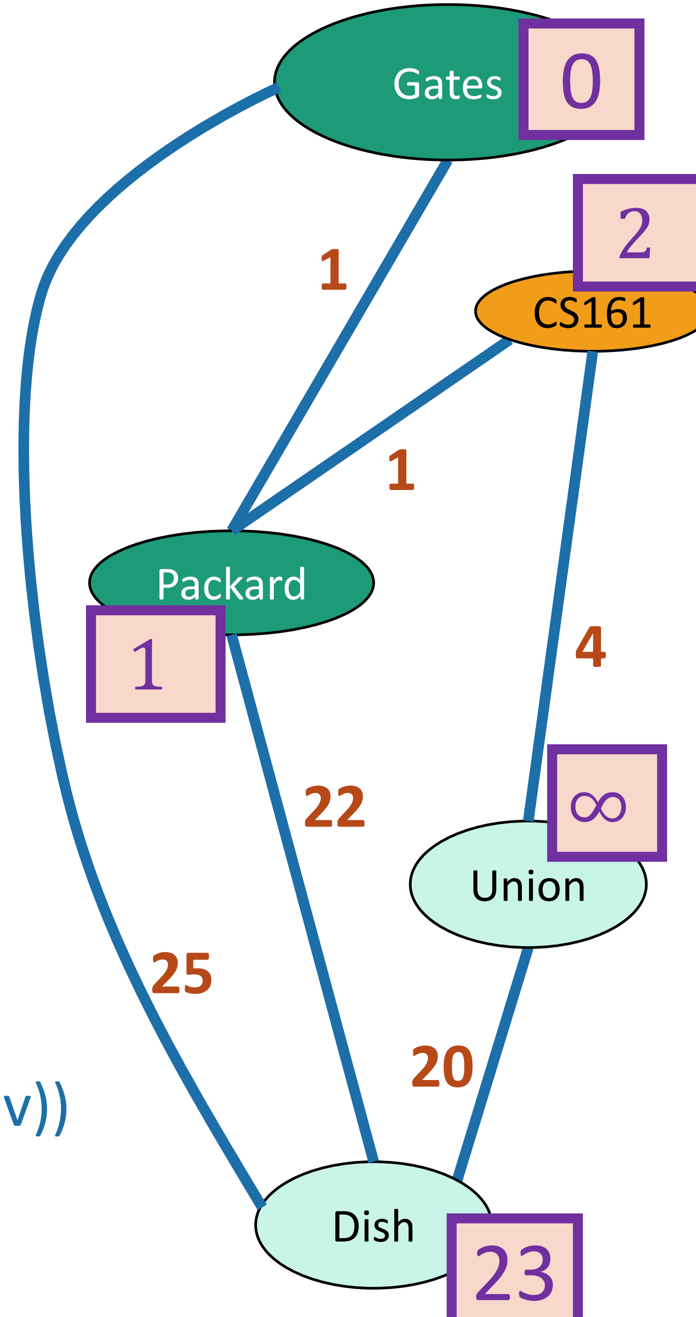
I'm not sure yet

I'm sure

x = 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?

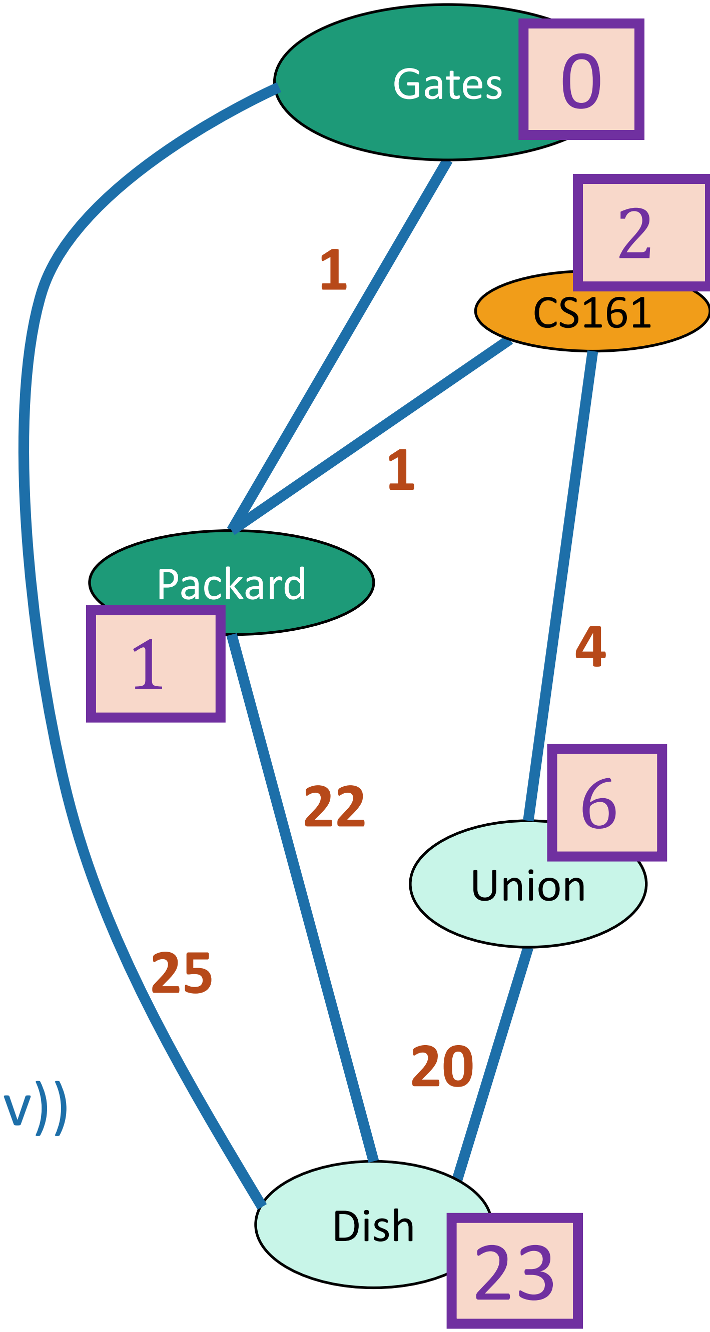
I'm not sure yet

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

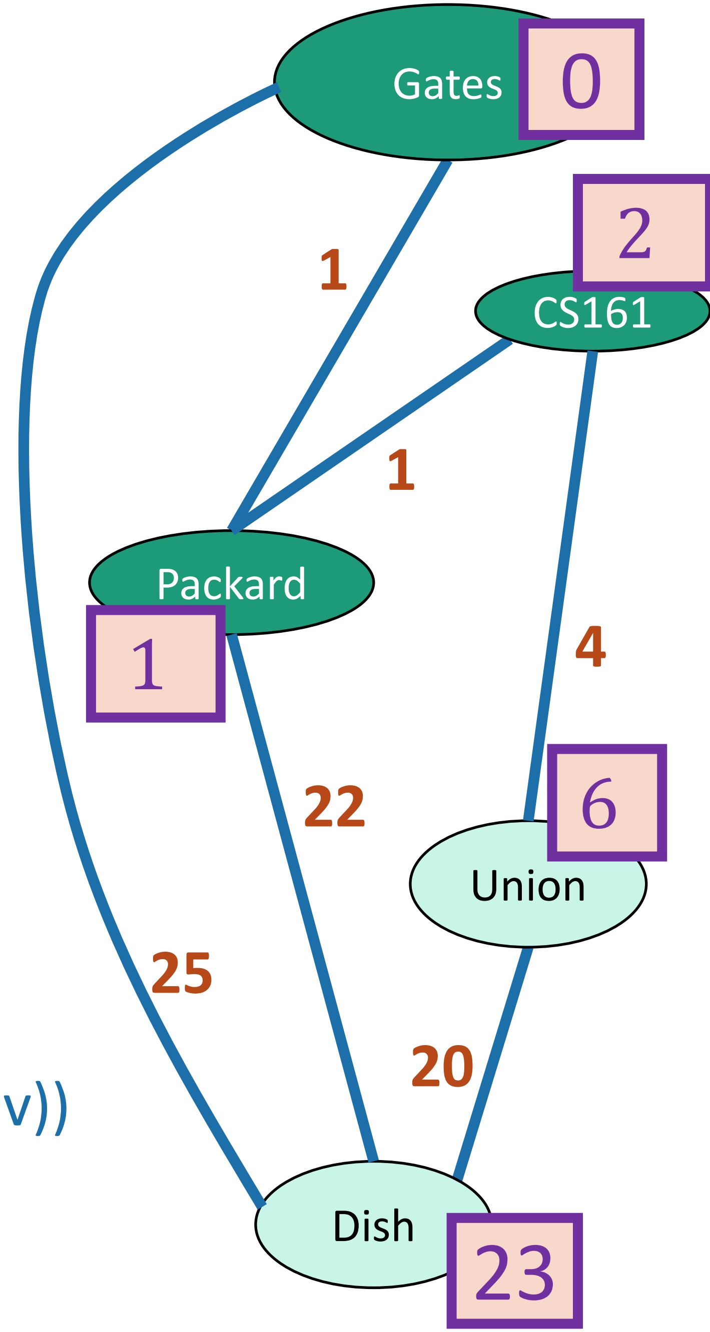
I'm sure

x

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

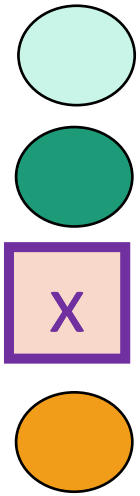
Current node u

- 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?



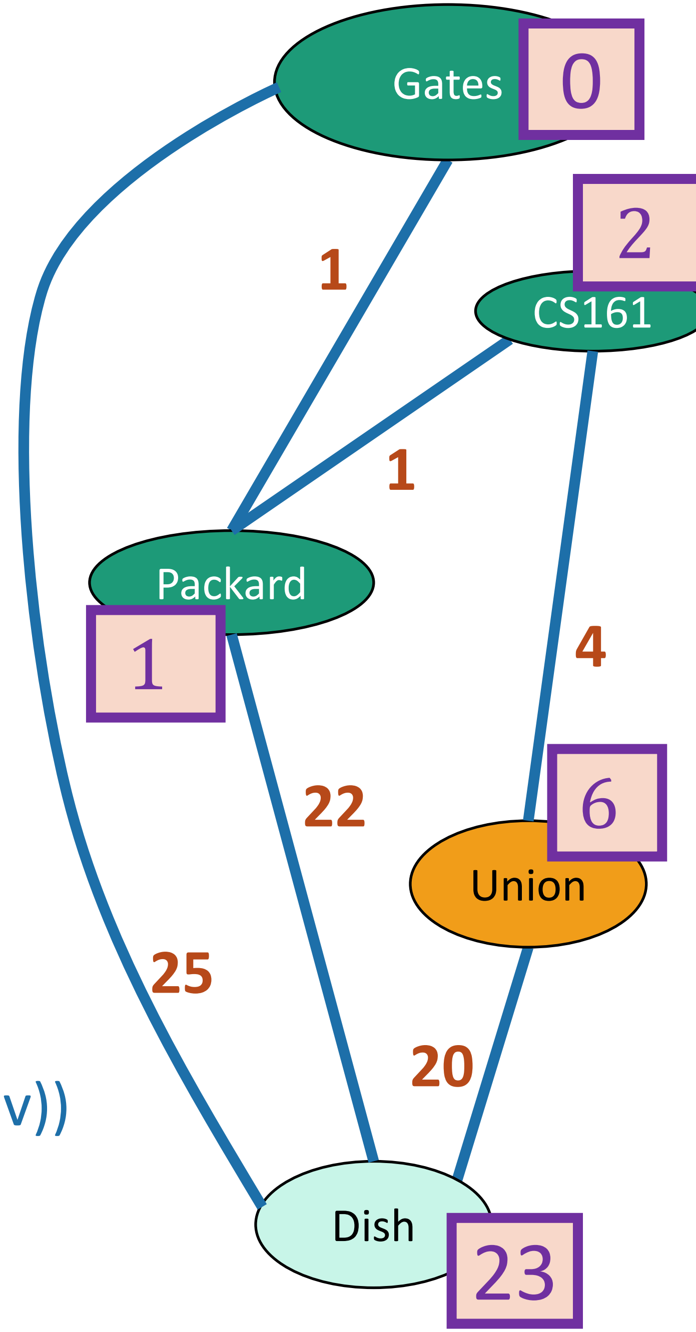
I'm not sure yet

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?

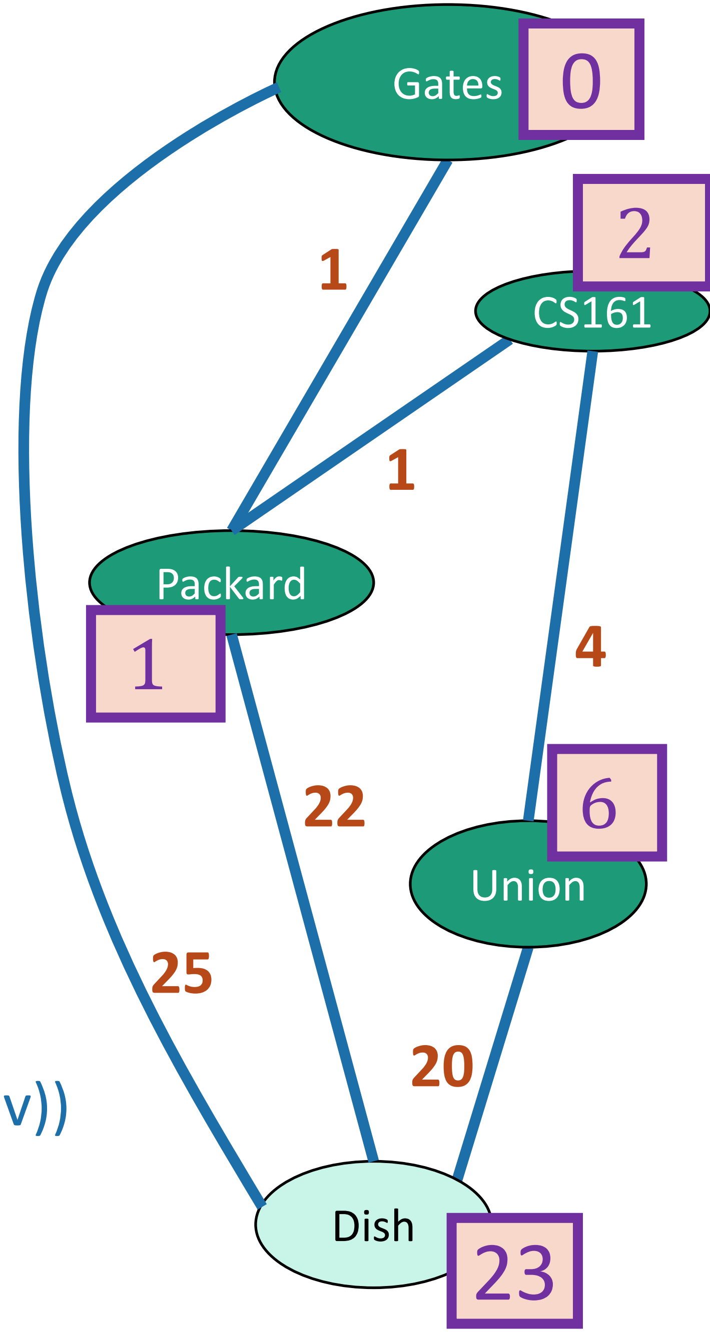
I'm not sure yet

I'm sure

x

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

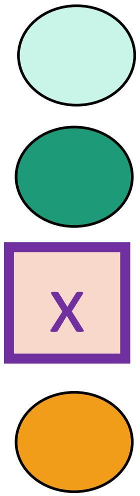
Current node u



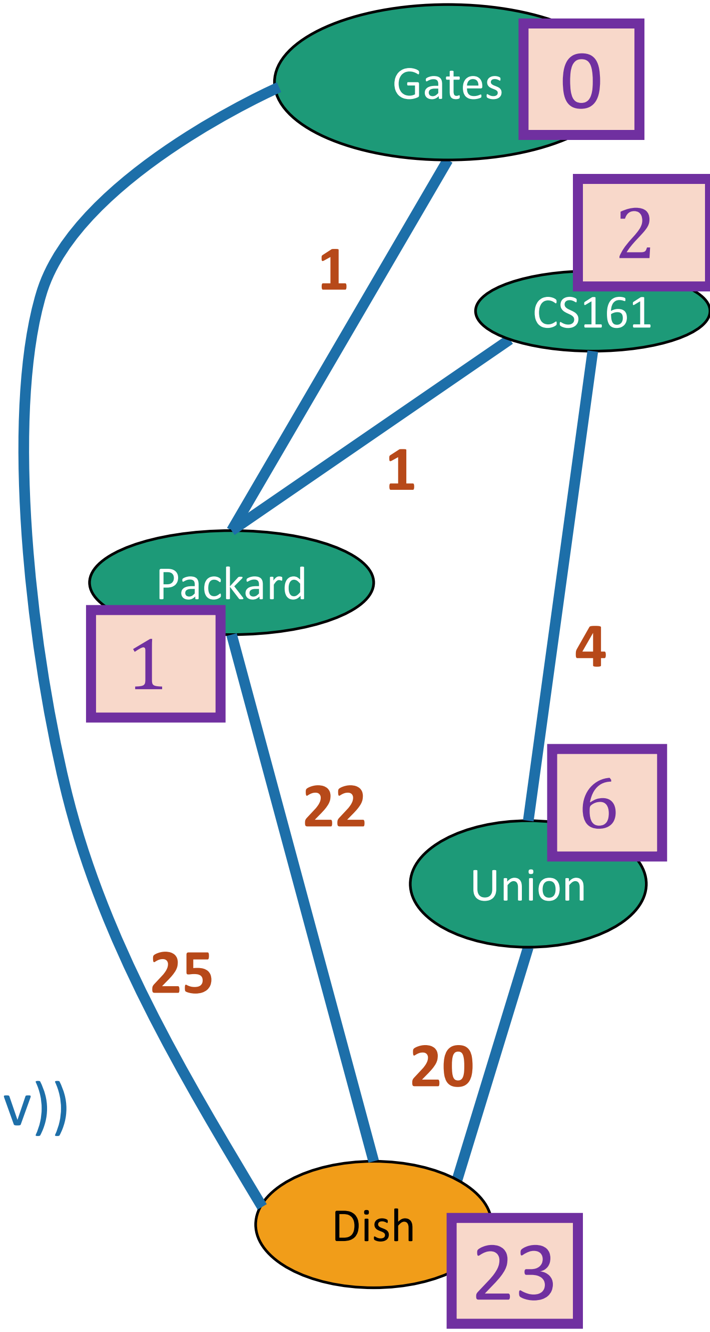
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from Gates?



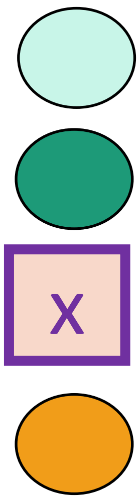
- I'm not sure yet
- I'm sure
- **x = d[v]** is my best **over-estimate** for dist(Gates,v).
- Current node u



- 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?



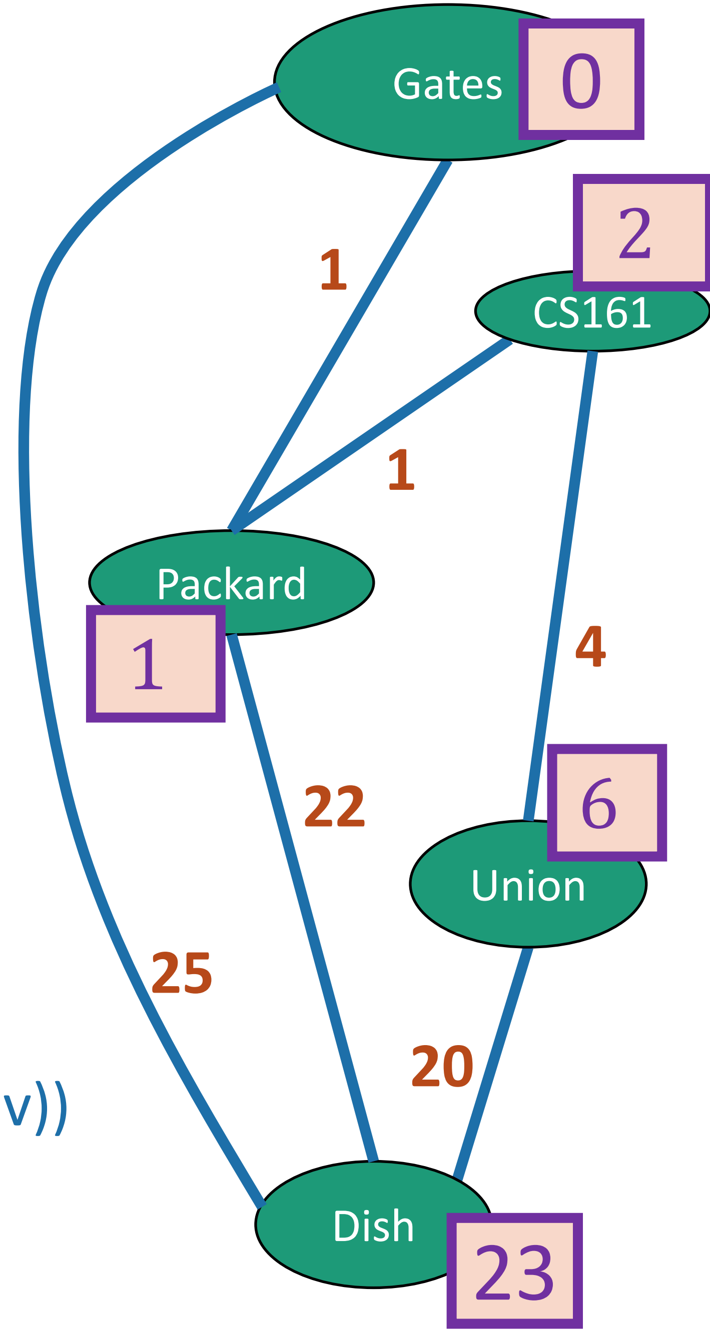
I'm not sure yet

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'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:**
  - 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.**
  - By the time we are **sure** about v, **d[v] = d(s,v).**
  - **d[v]** never increases, so after v is **sure**, **d[v]** stops changing.
  - All vertices are eventually **sure**. (Stopping condition in algorithm)
  - 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:

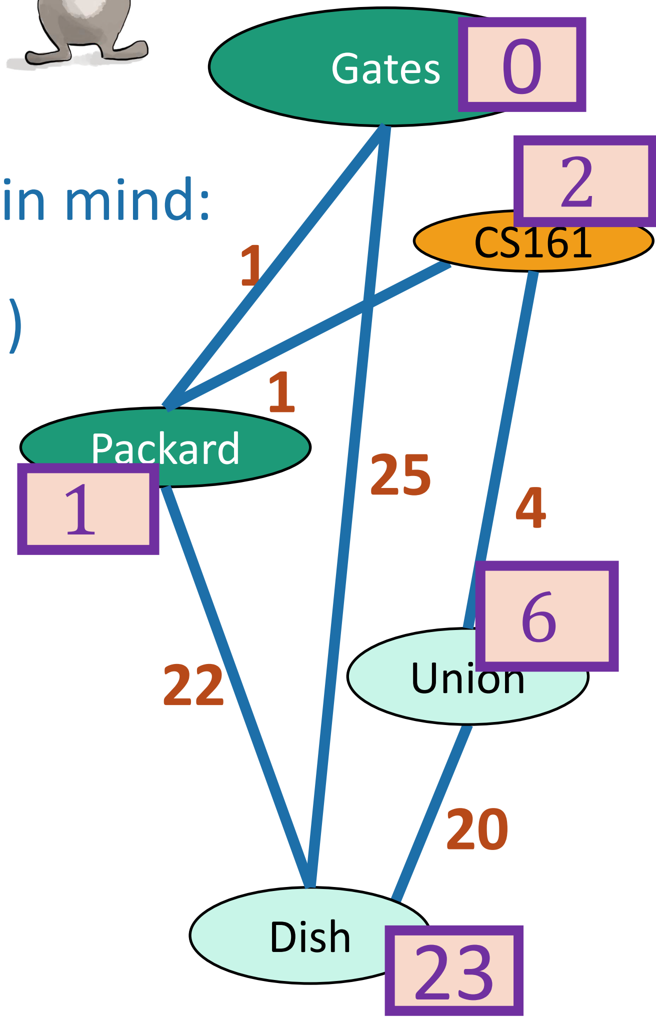
- Every time we update d[v], we have a path in mind:

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

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 hidden slide or do it yourself)

Claim 1

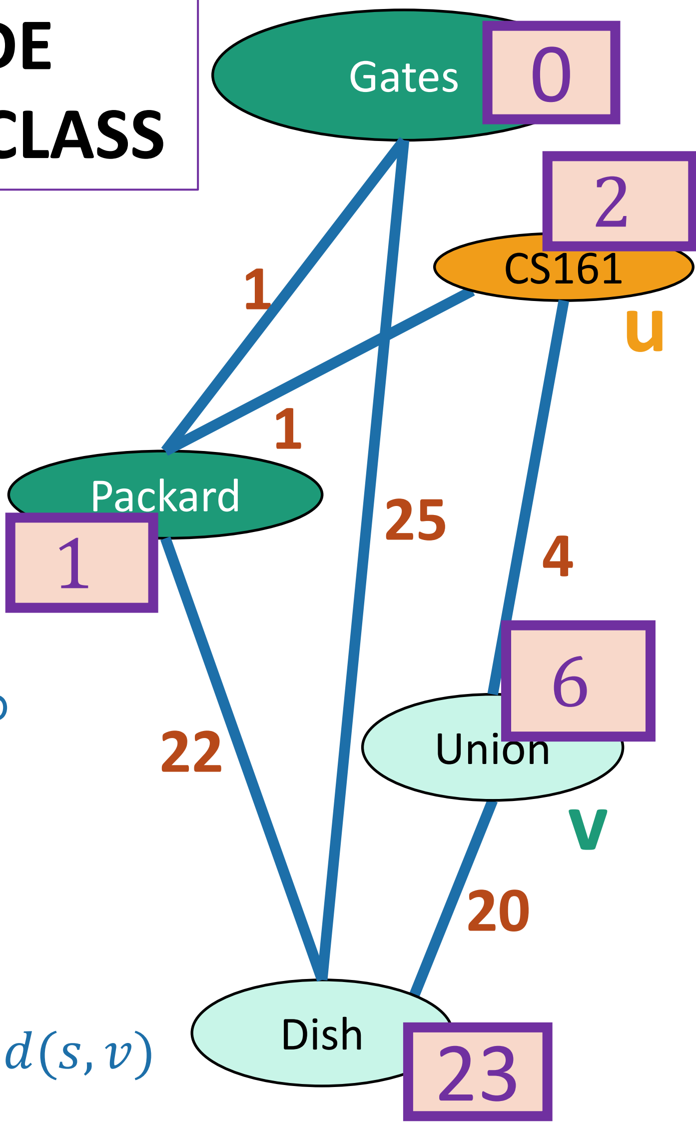
$d[v] \geq d(s,v)$ for all v.

THIS SLIDE SKIPPED IN CLASS

- 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,
$d(s,v) \leq d[v]$

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



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

Claim 2

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

- For s (the start vertex):
  - The first vertex marked **sure** has d[s] = d(s,s) = 0.
- For all the other vertices:
  - 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

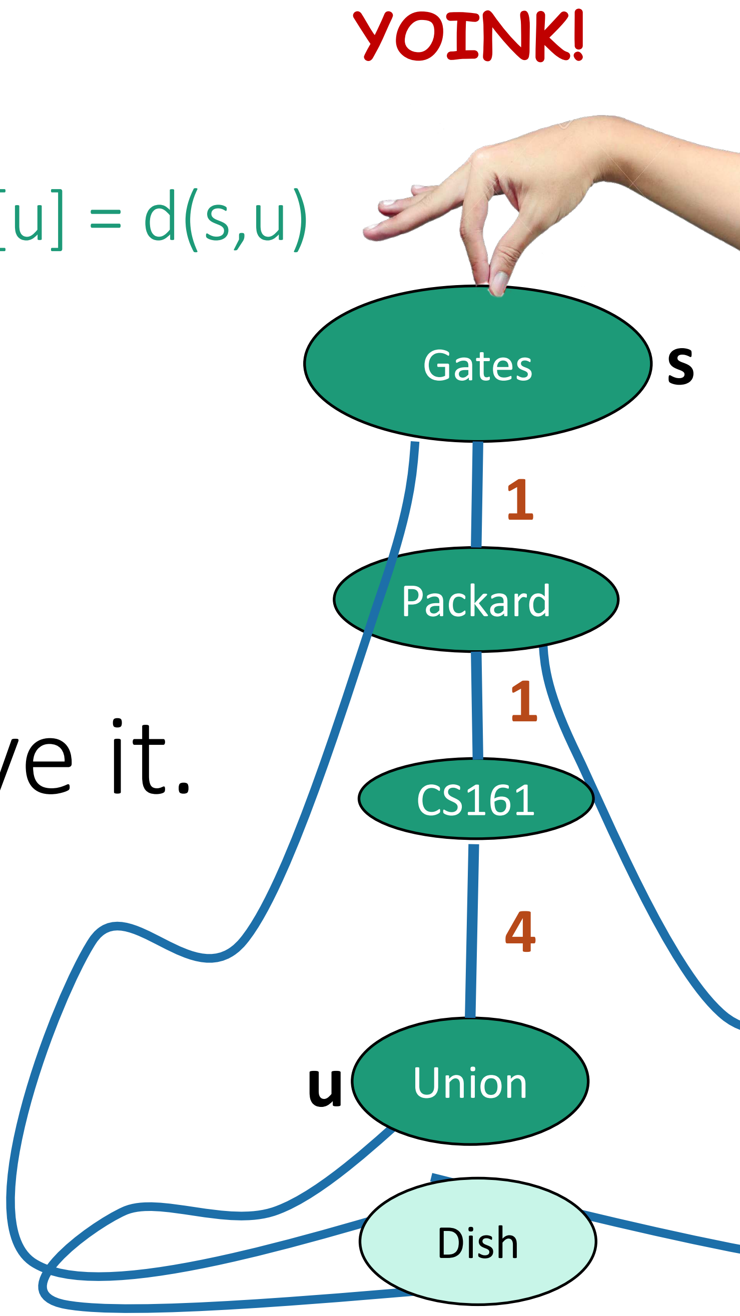
  - Want to show that d[u] = d(s,u).

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 let's actually prove it.

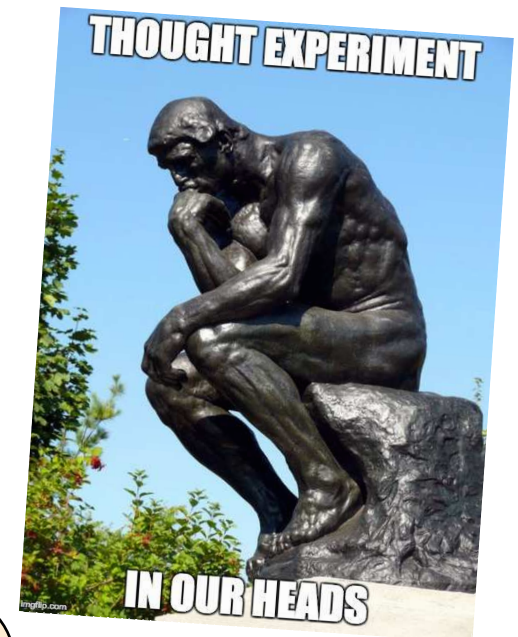


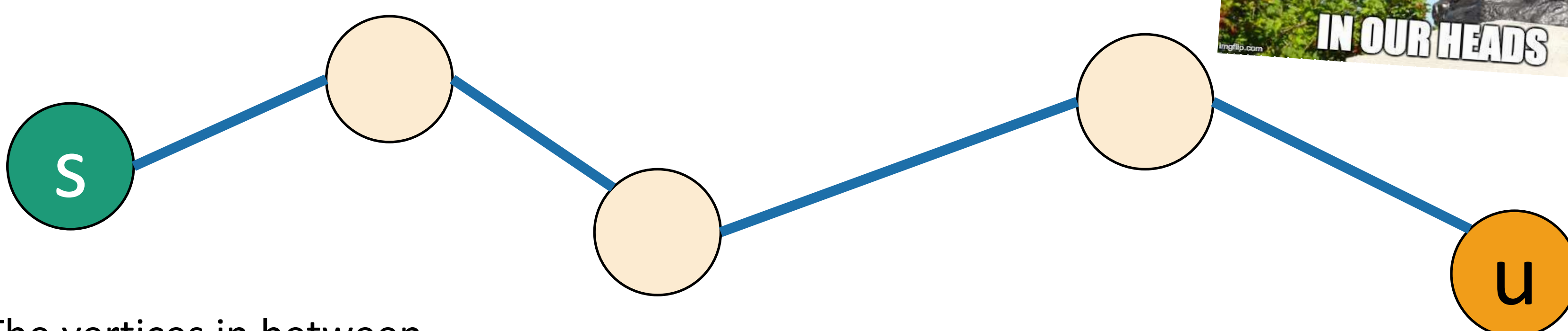
Claim 2

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

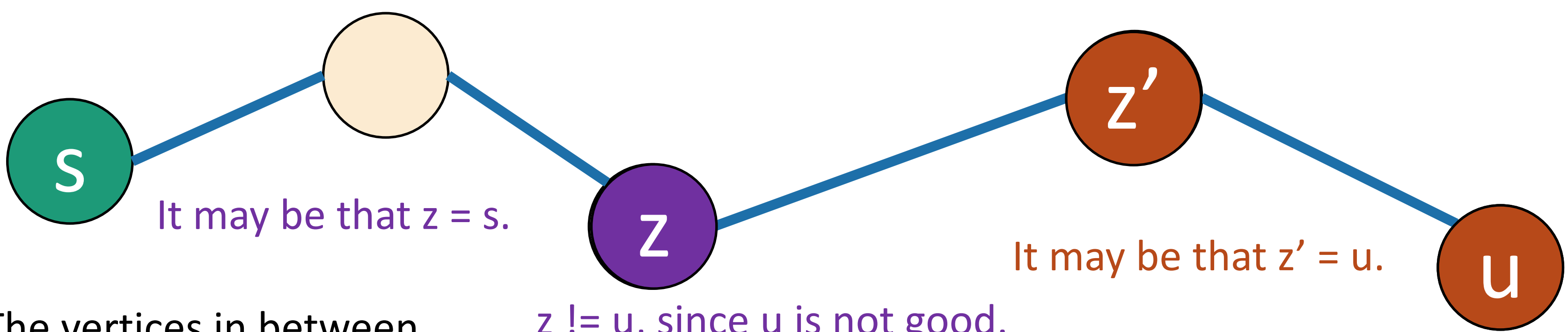
Temporary definition:

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.**

True shortest path.

Claim 2

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.

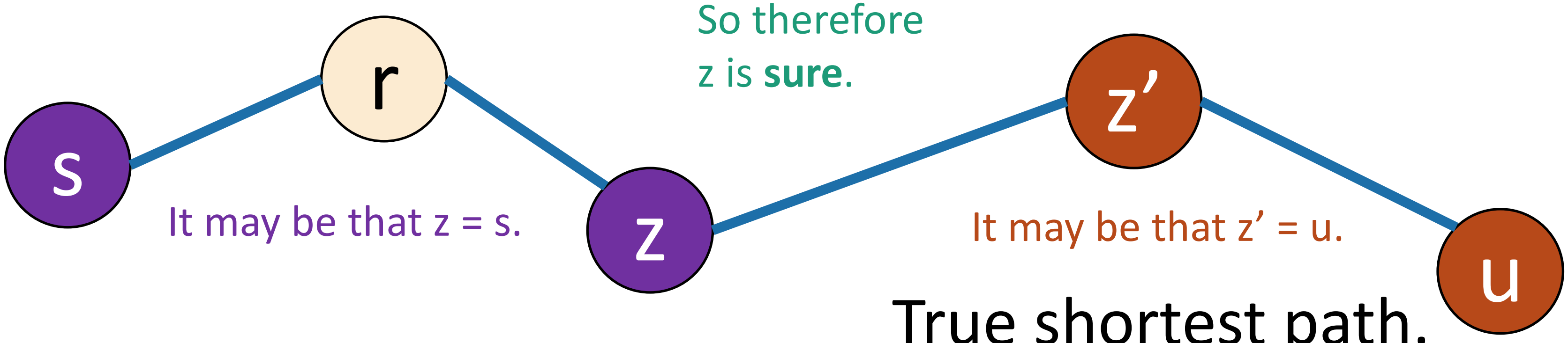
$$d[z] = d(s,z) \le d(s,u) \le d[u]$$

z is good   This is the shortest path from s to u.   Claim 1

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

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

So therefore z is **sure**.



Claim 2

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

$$d[z'] \le d[z] + w(z,z') = d(s,z') \le d[z']$$

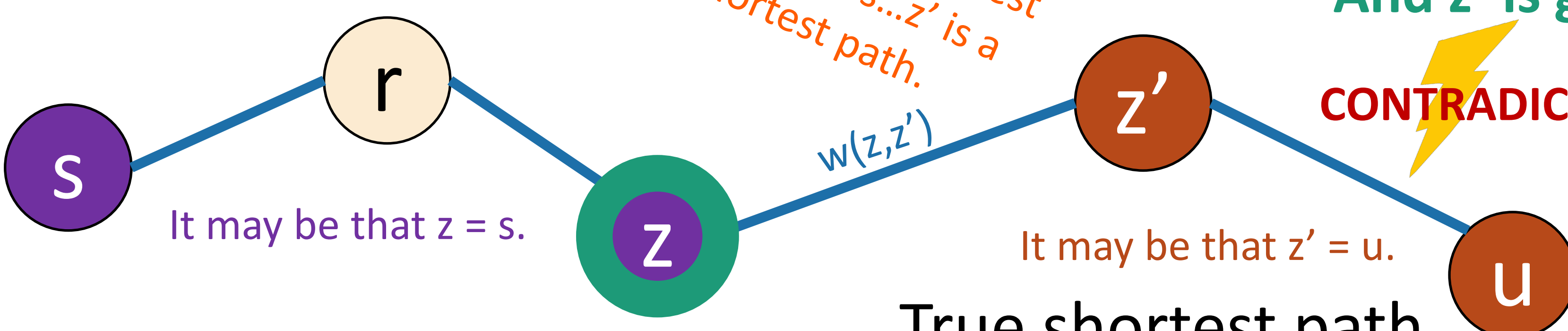
def of update

sub-paths of shortest paths are shortest paths so s…z' is a shortest path.

Claim 1

So everything is equal!
d(s,z') = d[z']
And z' is good.

CONTRADICTION!!



It may be that z = s.

It may be that z' = u.

True shortest path.

Back to this slide

Claim 2

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.

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

Def. of z

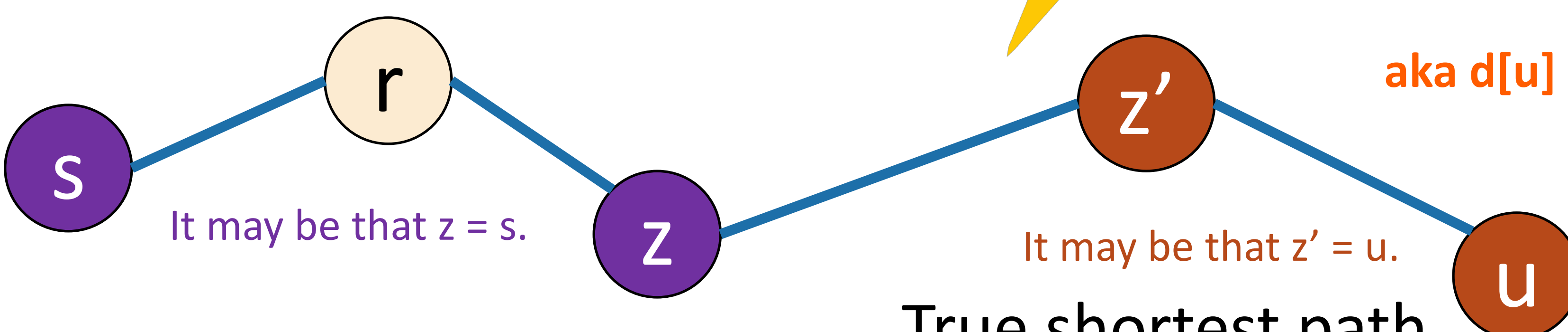
This is the shortest path from s to x

Claim 1

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

So u is good!

aka d[u] = d(s,v)



It may be that z = s.

It may be that z' = u.

True shortest path.

Back to this slide

Claim 2

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

- For s (the starting vertex):
  - The first vertex marked **sure** has d[s] = d(s,s) = 0.
- For all other vertices:
  - 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

Then u is good! aka d[u] = d(s,u)

Why does this work?

Now back to this slide

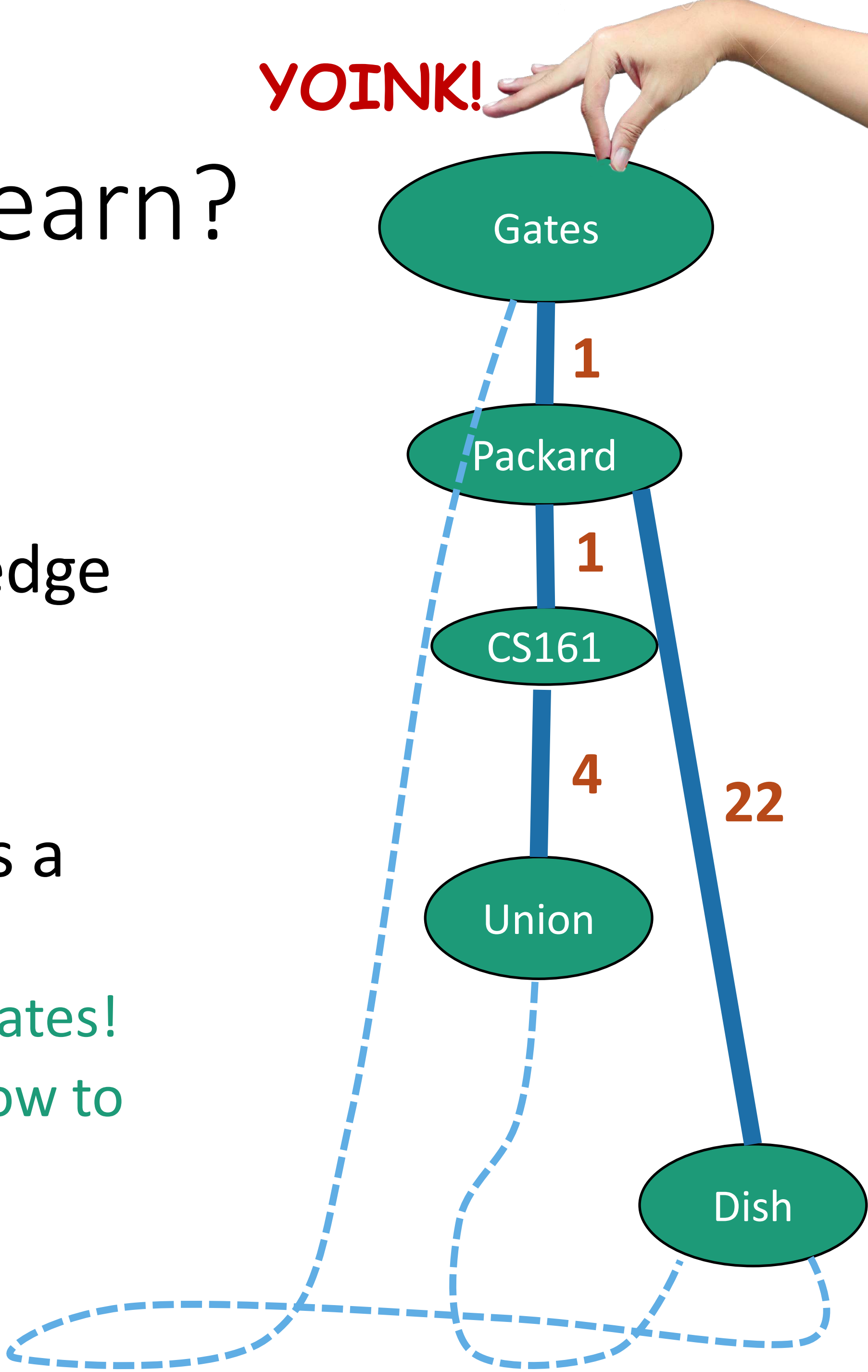
- **Theorem:**
  - Run Dijkstra on G =(V,E) starting from s.
  - At the end of the algorithm, the estimate **d[v]** is the actual distance d(s,v).

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

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

What did we just learn?

- **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:

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

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

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

Better than an array if the graph is sparse!

aka if m is much smaller than n^2

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

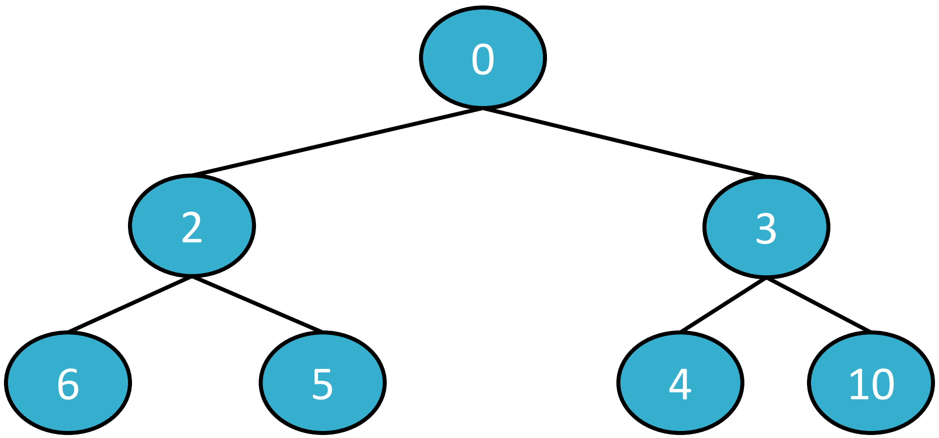
Is a hash table a good idea here?

- **Not really**:
  - `Search(v)` is fast (in expectation)
  - But `findMin()` will still take time O(n) without more structure.

Slide skipped in class

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[7]	Leftist	Binomial[7]	Fibonacci[7][8]	Pairing[9]	Brodal[10][b]	Rank-pairing[12]	Strict Fibonacci[13]
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(findMin) = O(1) (amortized time*)
- T(removeMin) = O(log(n)) (amortized time*)
- T(updateKey) = O(1) (amortized time*)
- See CS166 for more! (or CLRS)

- Running time of Dijkstra

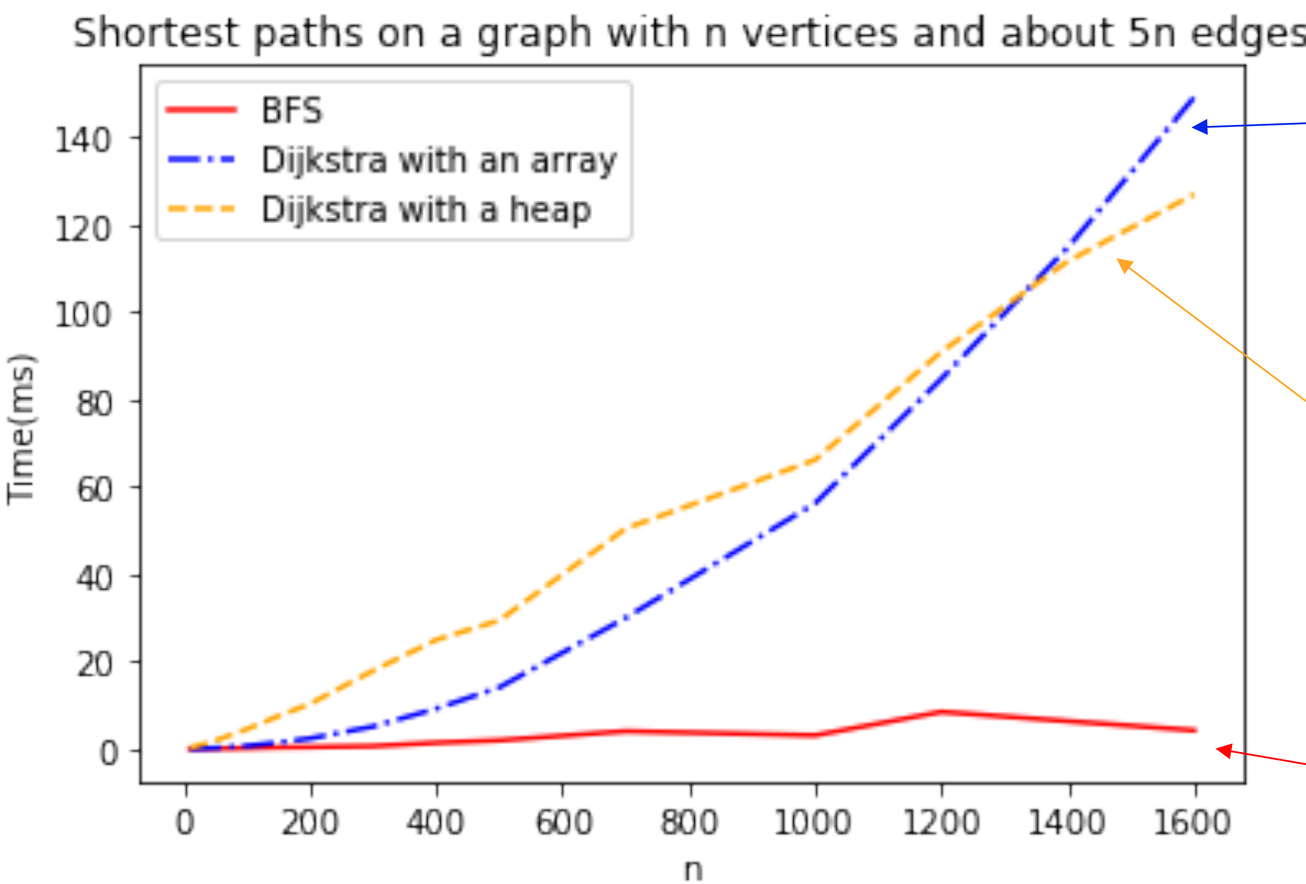
  = O(n(T(`findMin`) + T(`removeMin`)) + m T(`updateKey`))

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

*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.

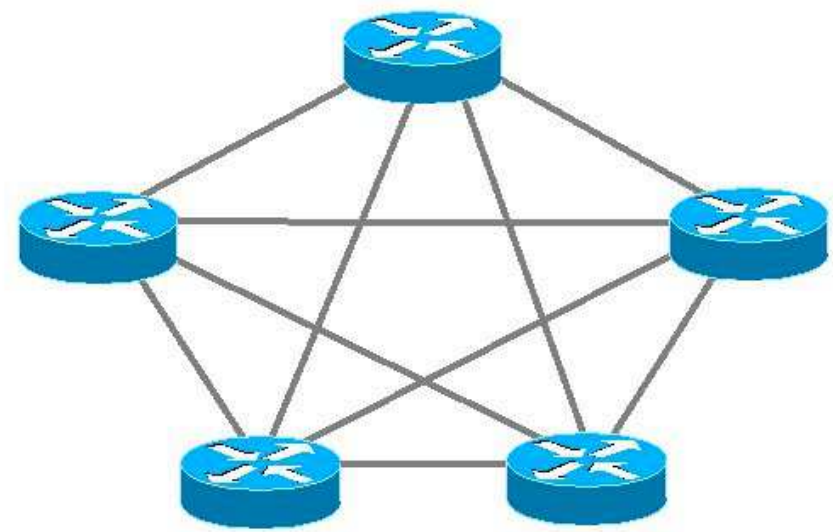
Dijkstra using a heap looks a bit more linear (actually nlog(n))

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.



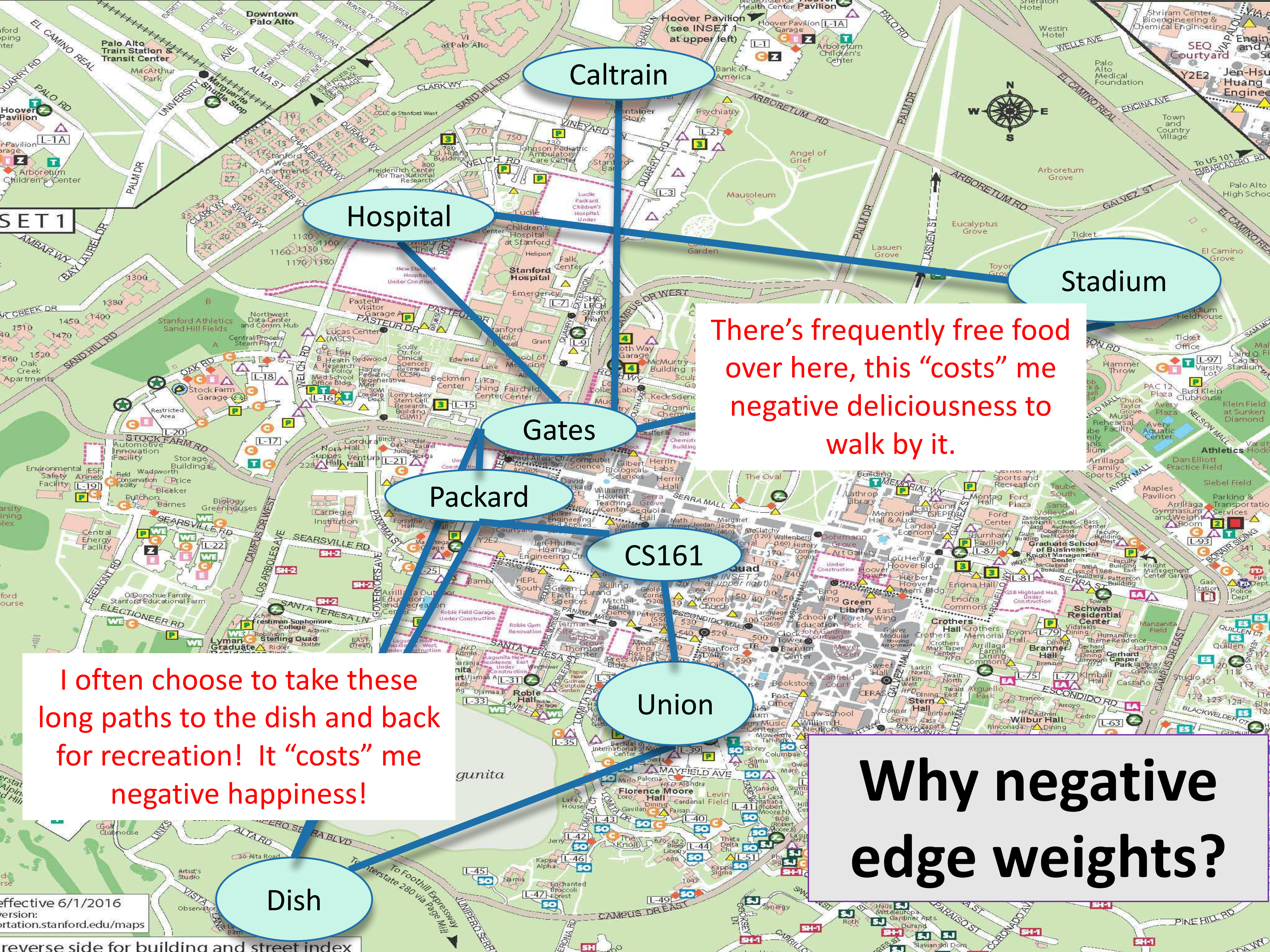
Dijkstra Drawbacks

- 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.
- (+) Allows for some flexibility if the weights change.
  - We'll see what this means later

Why negative edge weights?



There's frequently free food over here, this "costs" me negative deliciousness to walk by it.

I often choose to take these long paths to the dish and back for recreation! It "costs" me negative happiness!

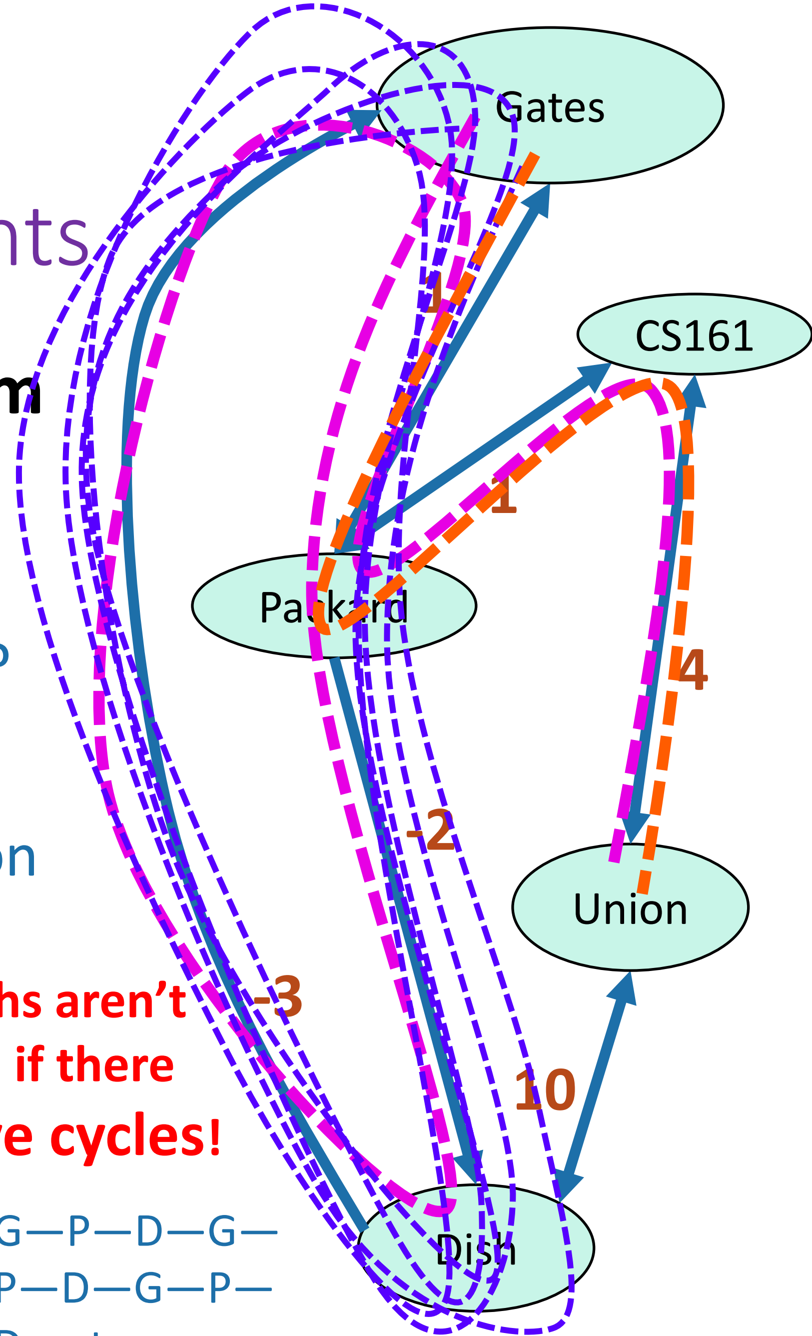
One problem

with negative edge weights

- **What is the shortest path from Gates to the Union?**
- Should it still be
  Gates—Packard—CS161—Union?
- But what about
  - G—P—D—G—P—CS161—Union
- That costs
  - 1-2-3+1+1+4 = 2.
- And why not

G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—G—P—D—etc....

Shortest Paths aren't well-defined if there are negative cycles!



Let’s put that aside for a moment

Onwards!

To the Bellman-Ford algorithm!

Bellman-Ford algorithm

Bellman-Ford(G,s):

- d[v] = ∞ for all v in V
- d[s] = 0
- **For** i=0,…,n-1:
  - **For** u in V:
    - **For** v in u.neighbors:
      - d[v] ← min(d[v] , d[u] + 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.neighbors:
    - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**.

For pedagogical reasons

which we will see next week

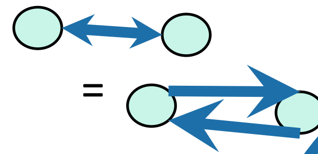
- We are actually going to change this to be dumber.
- Keep n arrays: $d^{(0)}, d^{(1)}, \ldots, d^{(n-1)}$

Bellman-Ford*(G,s):

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

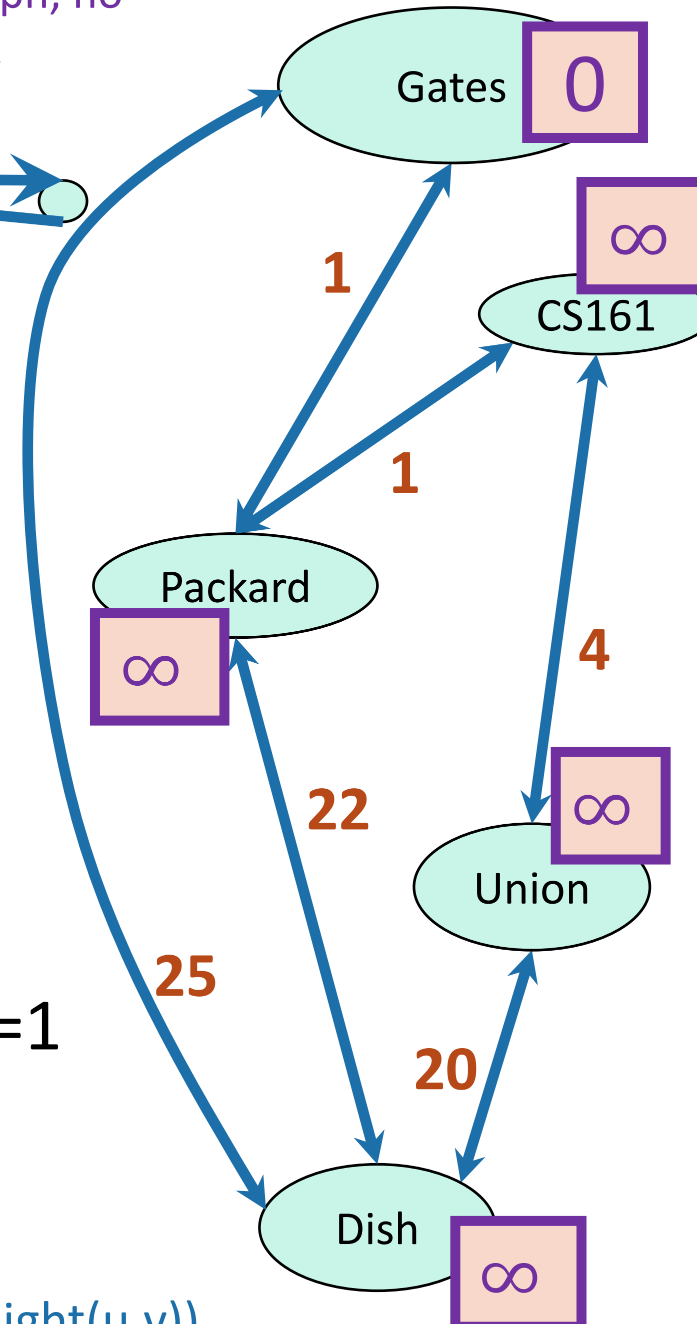
Bellman-Ford

Start with the same graph, no negative weights.



How far is a node from Gates?

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

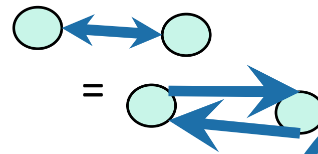


i=1

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

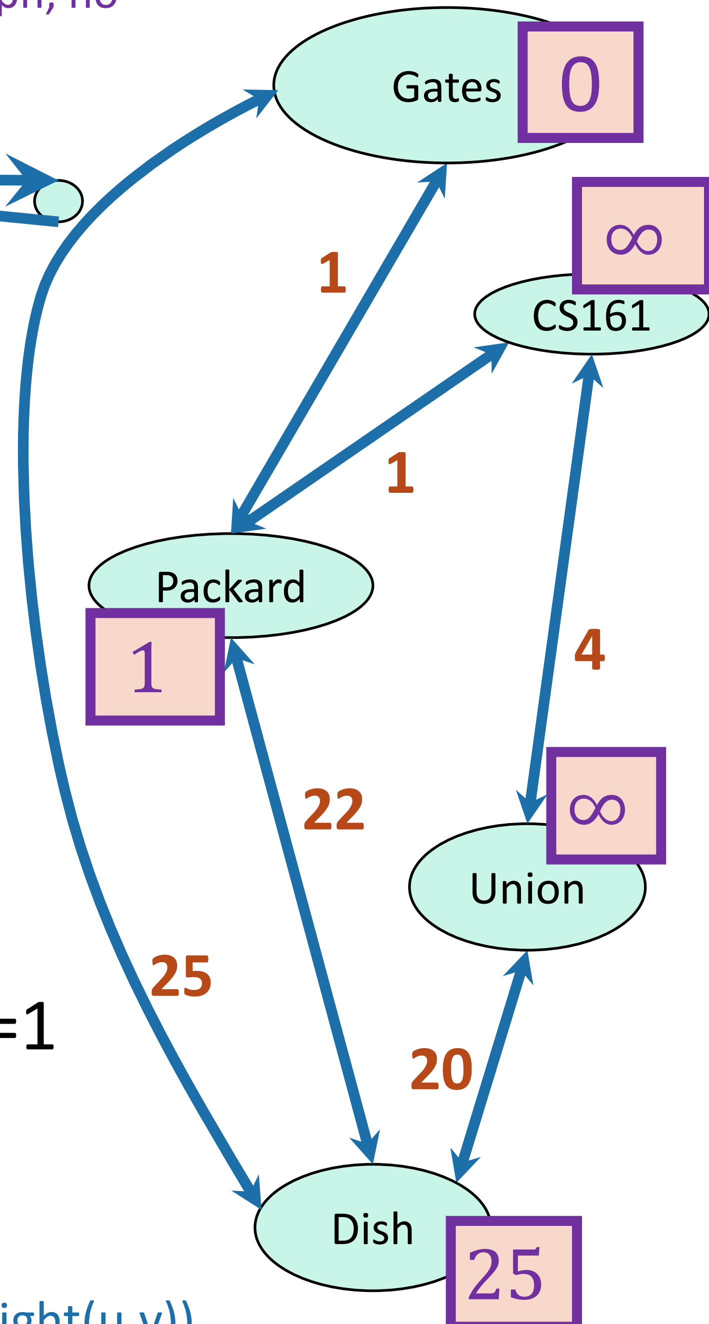
Bellman-Ford

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**
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					

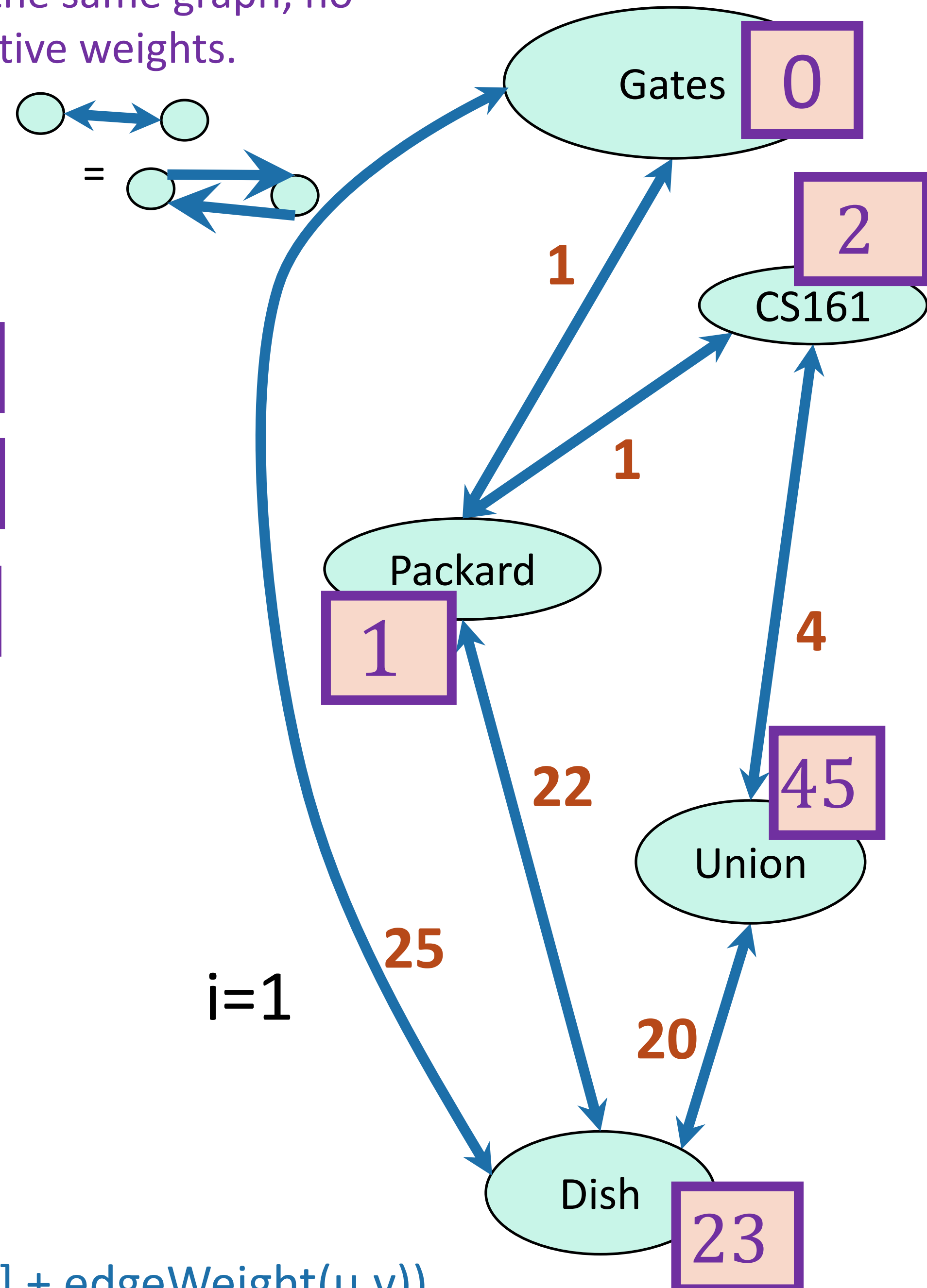


i=1

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

Bellman-Ford

Start with the same graph, no negative weights.



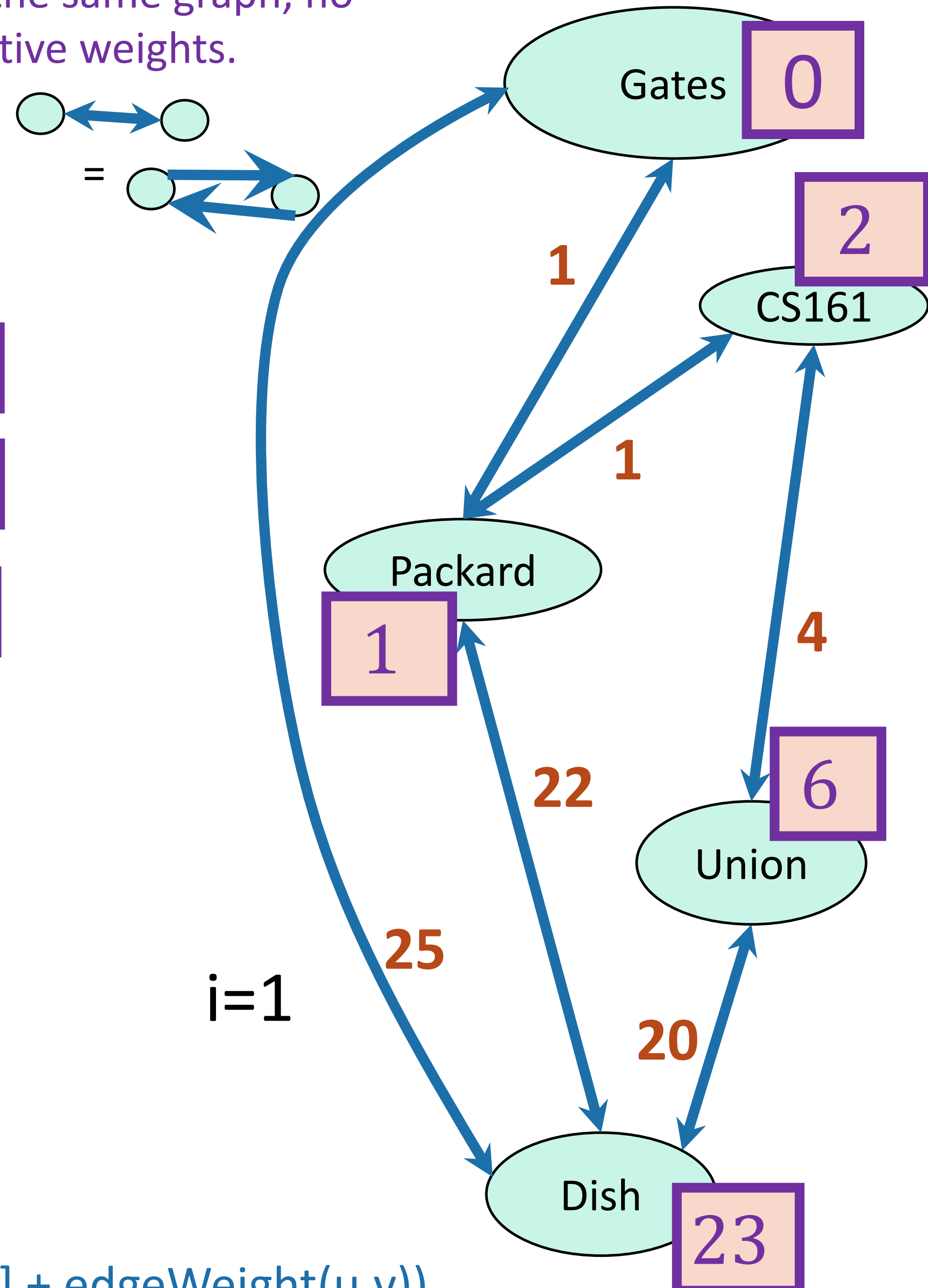
How far is a node from Gates?

	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:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v))$

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
$d^{(4)}$					

i=1

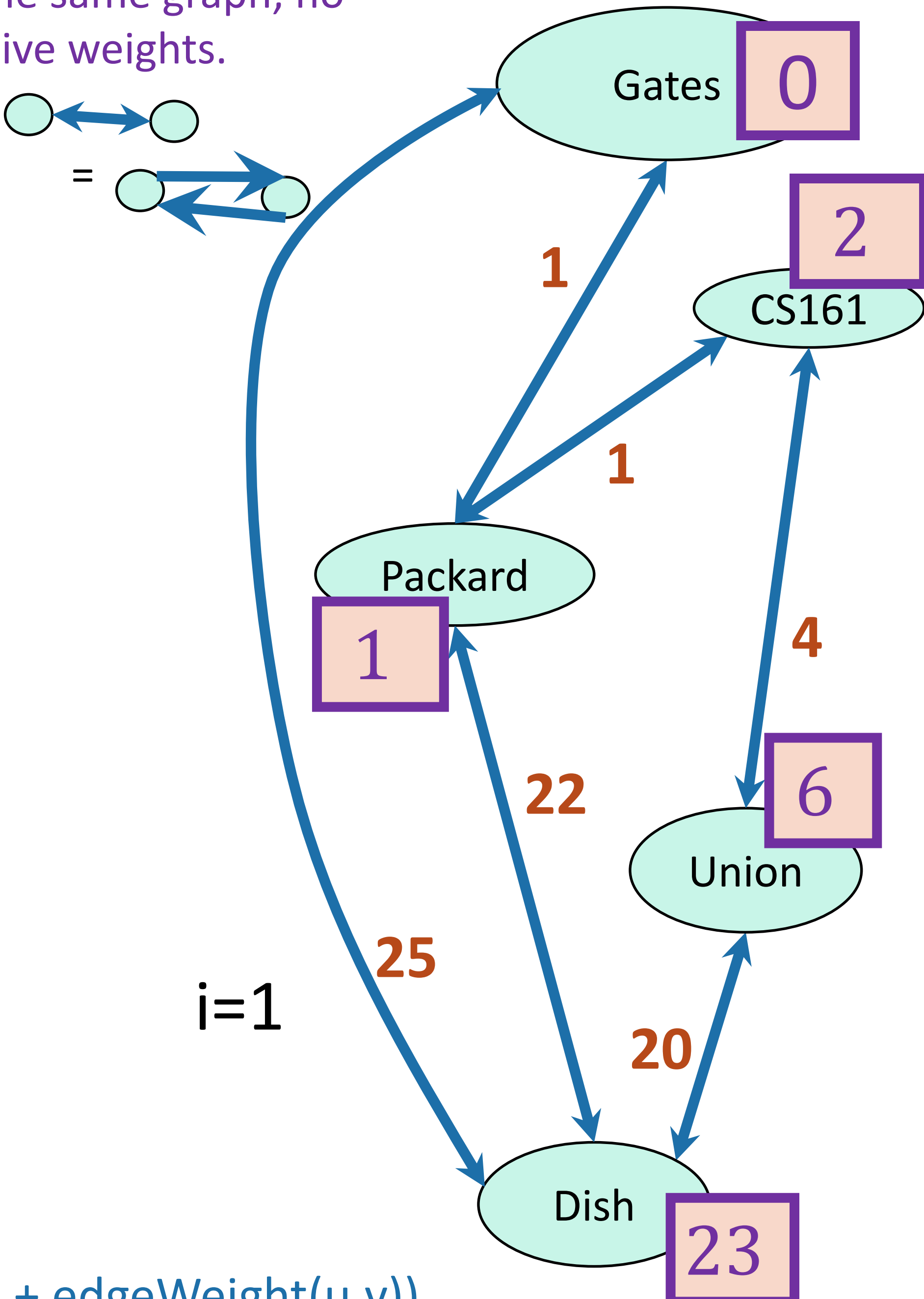
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v))$

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



i=1

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

As usual

- Does it work?
  - Yes
  - Idea to the right.
  - (Base case and inductive step similar to Dijkstra)
  - (See hidden slides for details)
- Is it fast?
  - Not really…

	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 path between s and v.
(Since all simple paths have at most n-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 with at most i edges.
- **Base case:**
  - After iteration 0…
- **Inductive step:**

Skipped in class

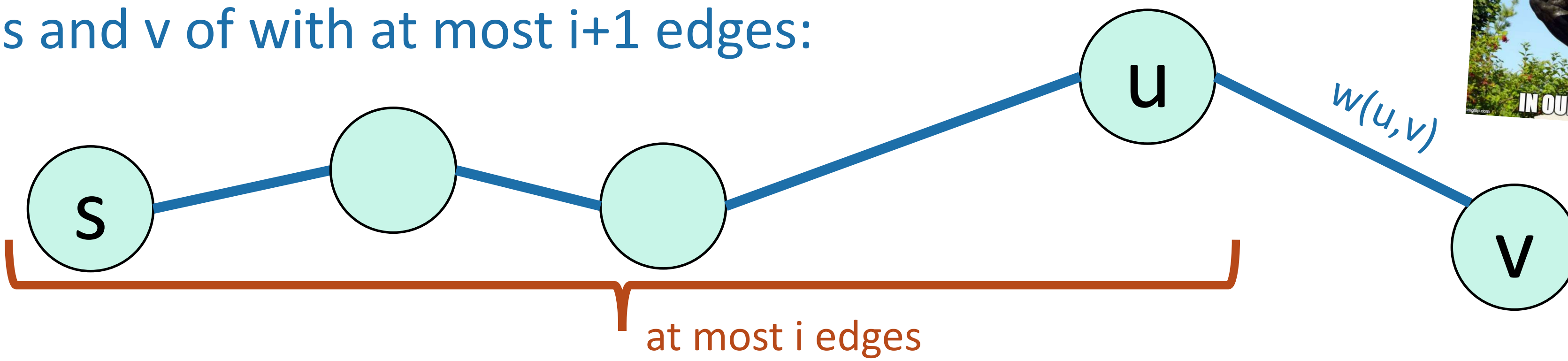
Inductive step

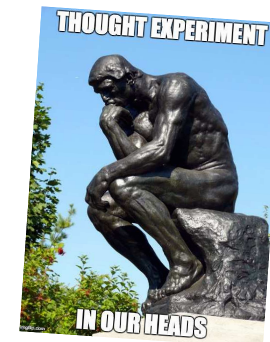
Hypothesis: After iteration i, for each v, $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v with at most i edges.

- 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 cycles!

This seems much slower than Dijkstra

- And it is:

Running time $O(mn)$

- However, it's also more flexible in a few ways.
  - Can handle negative edges
  - If we keep on doing these iterations, then changes in the network will propagate through.

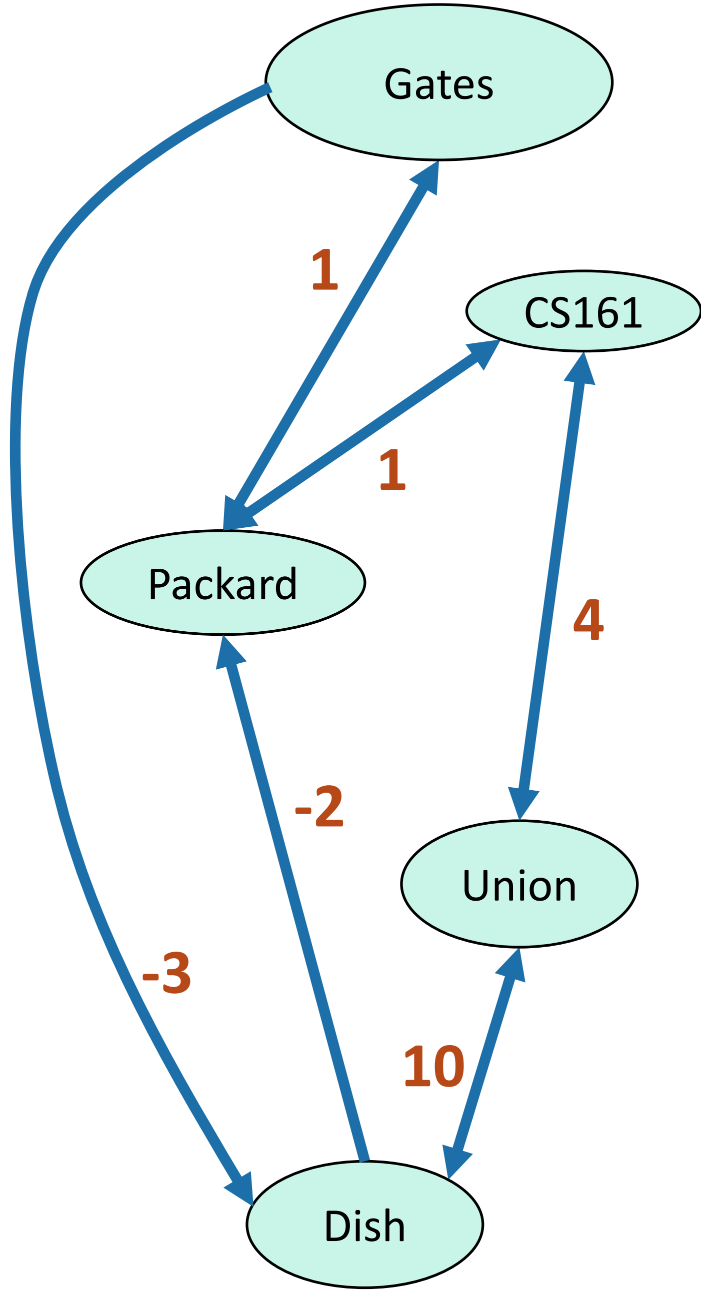
- **For** i=0,…,n-1:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v))$

Negative edge weights

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

This is not looking good!

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



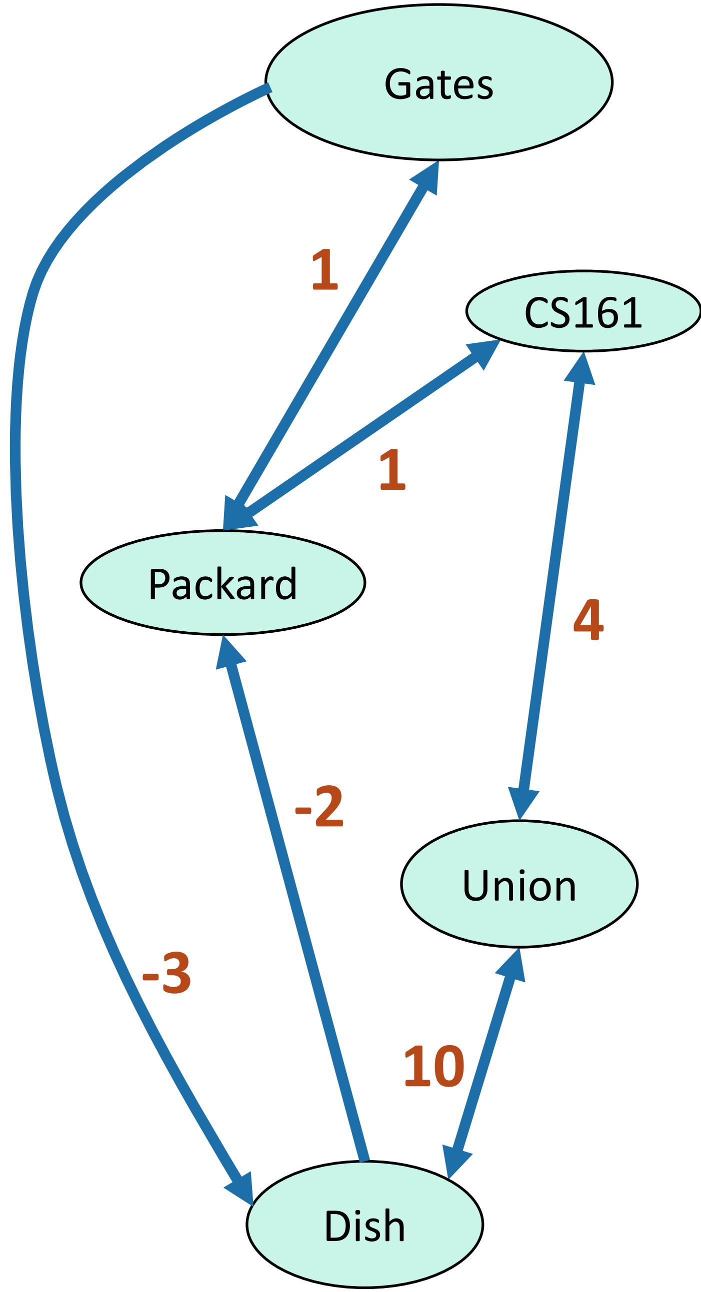
Negative edge weights

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

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

	Gates	Packard	CS161	Union	Dish
$d^{(5)}$	-4	**-9**	-4	**3**	-7

Some stuff changed!



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

Back to the correctness

- Does it work?
  - Yes
  - Idea to the right.
  - (Base case and inductive step similar to Dijkstra)

	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 path between s and v.
(Since all simple paths have at most n-1 edges).

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

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.

Bellman-Ford algorithm

Bellman-Ford*(G,s):

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

What have we learned?

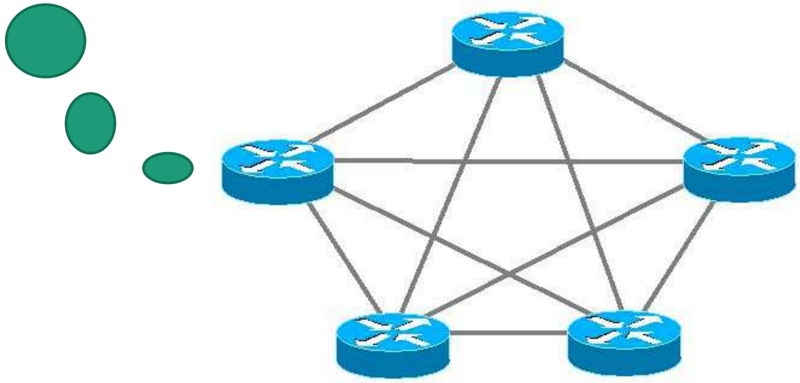
- The Bellman-Ford algorithm:
  - Finds shortest paths in weighted graphs with negative edge weights
  - runs in time O(nm) on a graph G with n vertices and m edges.
- If there are no negative cycles in G:
  - the BF algorithm terminates with $d^{(n-1)}[v] = d(s,v)$.
- If there are negative cycles in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

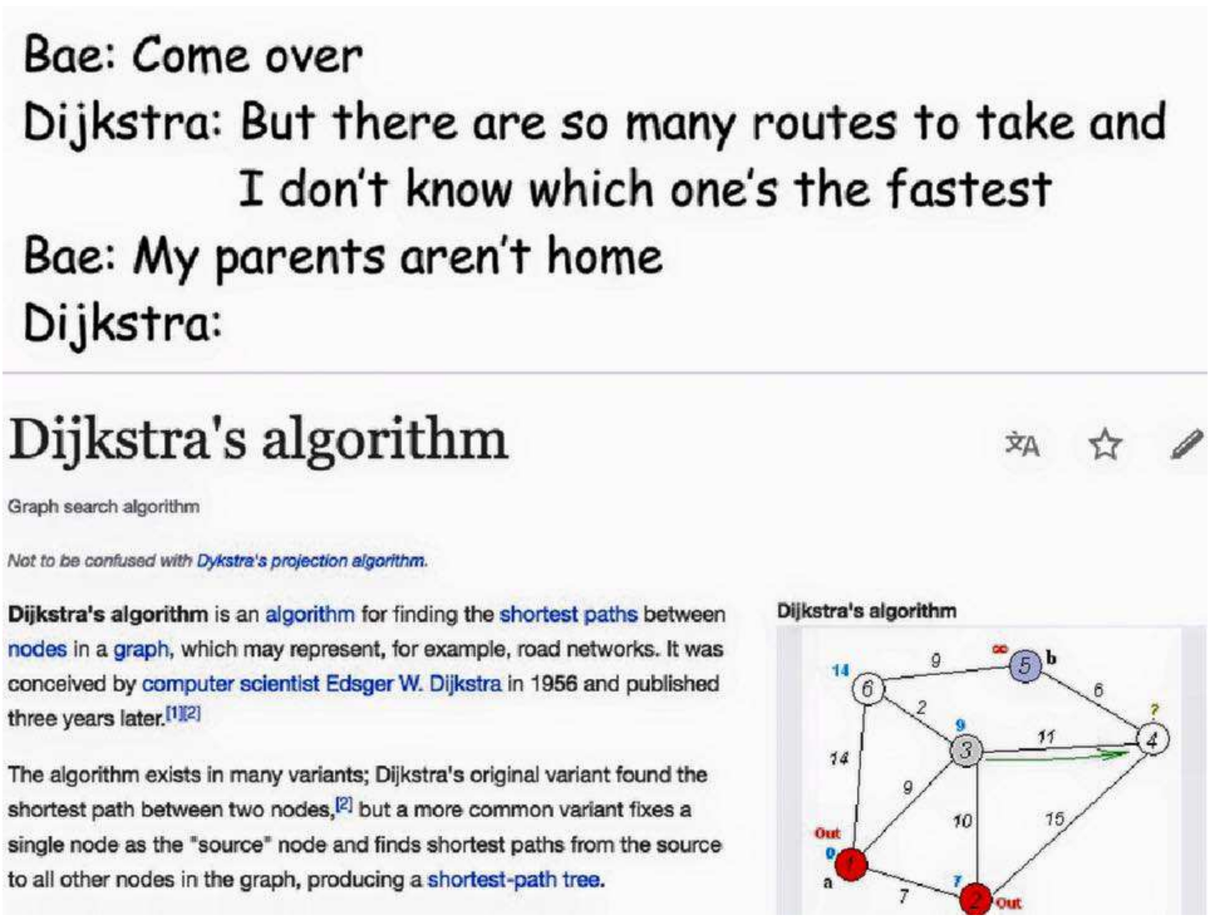
Destination	Cost to get there	Send to whom?
172.16.1.0	34	172.16.1.1
10.20.40.1	10	192.168.1.2
10.155.120.1	9	10.13.50.0



Recap: shortest paths

- BFS:
  - (+) O(n+m)
  - (-) only unweighted graphs
- Dijkstra's algorithm:
  - (+) weighted graphs
  - (+) O(nlog(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)

Andrés found a Dijkstra joke on the internets – thanks Andrés!



Bae: Come over

Dijkstra: But there are so many routes to take and I don't know which one's the fastest

Bae: My parents aren't home

Dijkstra:

Dijkstra's algorithm

Graph search algorithm

Not to be confused with Dykstra's projection algorithm.

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.[1][2]

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes,[2] but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

Dijkstra's algorithm

Perhaps this is why Dijkstra invented the algorithm?

Next Time

- More Bellman-Ford, plus Floyd-Warshall and dynamic programming!

Before next time

- Pre-lecture exercise:
  - How **NOT** to compute Fibonacci numbers.

Mini-topic (bonus slides; not on exam)

Amortized analysis!

- We mentioned this when we talked about implementing Dijkstra.

*Any sequence of d `deleteMin` calls takes time at most O(d log(n)). But some of the d may take longer and some may take less time.

- What's the difference between this notion and expected runtime?

Example

- Incrementing a binary counter n times.

0 1 10 11 100 101 110 111 1000 1001 1010 1011 1100 1101 1110 1111

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

- Say that flipping a bit is costly.
  - Above, we've noted the cost in terms of bit-flips.

Example

- Incrementing a binary counter n times.

0 1 10 11 100 101 110 111 1000 1001 1010 1011 1100 1101 1110 1111

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

- Say that flipping a bit is costly.
  - Some steps are very expensive.
  - Many are very cheap.
- ***Amortized*** over all the inputs, it turns out to be pretty cheap.
  - O(n) for all n increments.

This is different from expected runtime.

- The statement is deterministic, no randomness here.

- But it is still weaker than worst-case runtime.
    - We may need to wait for a while to start making it worth it.