Lecture 11

Weighted Graphs: Dijkstra and Bellman-Ford

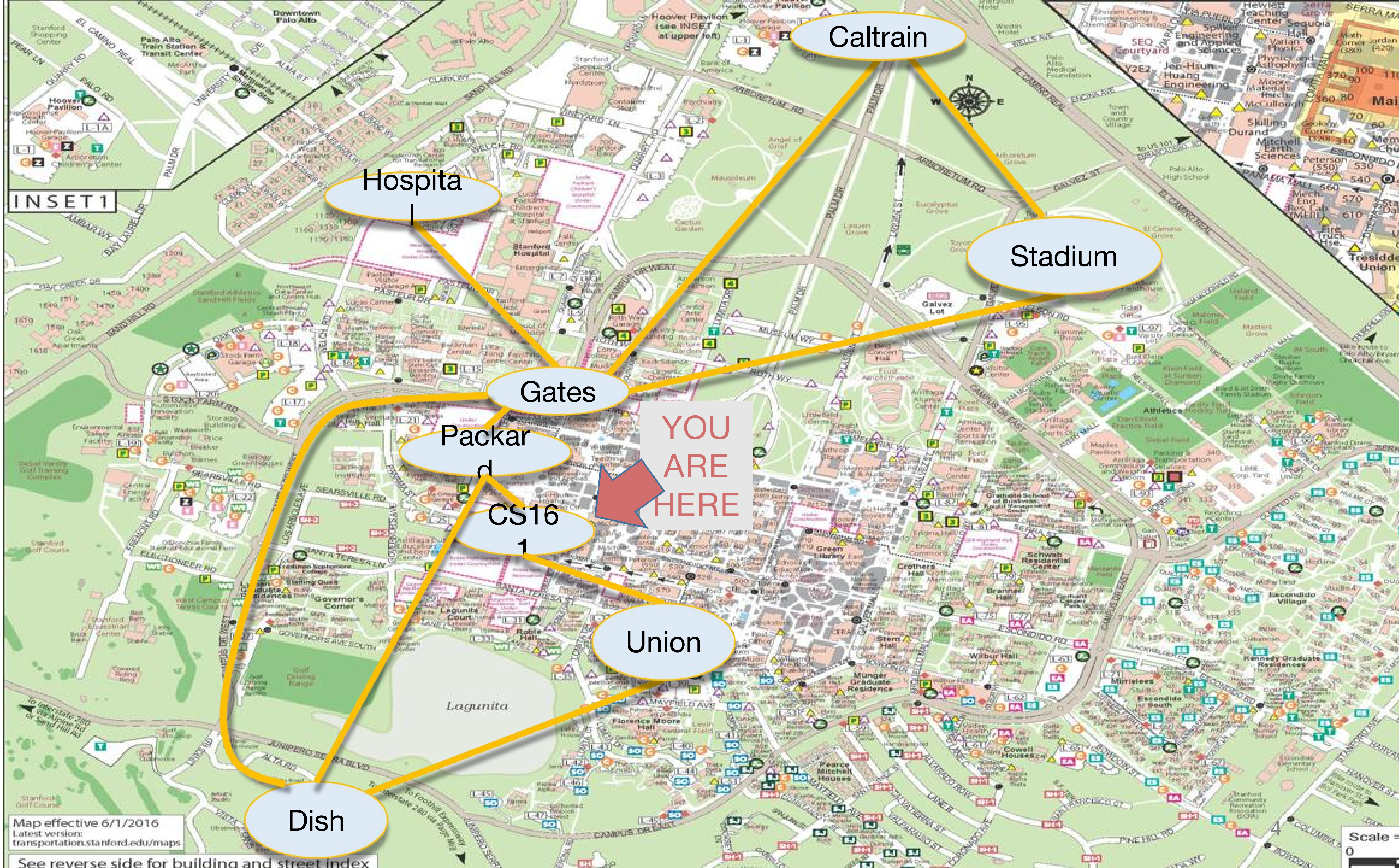
Previous two lectures

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

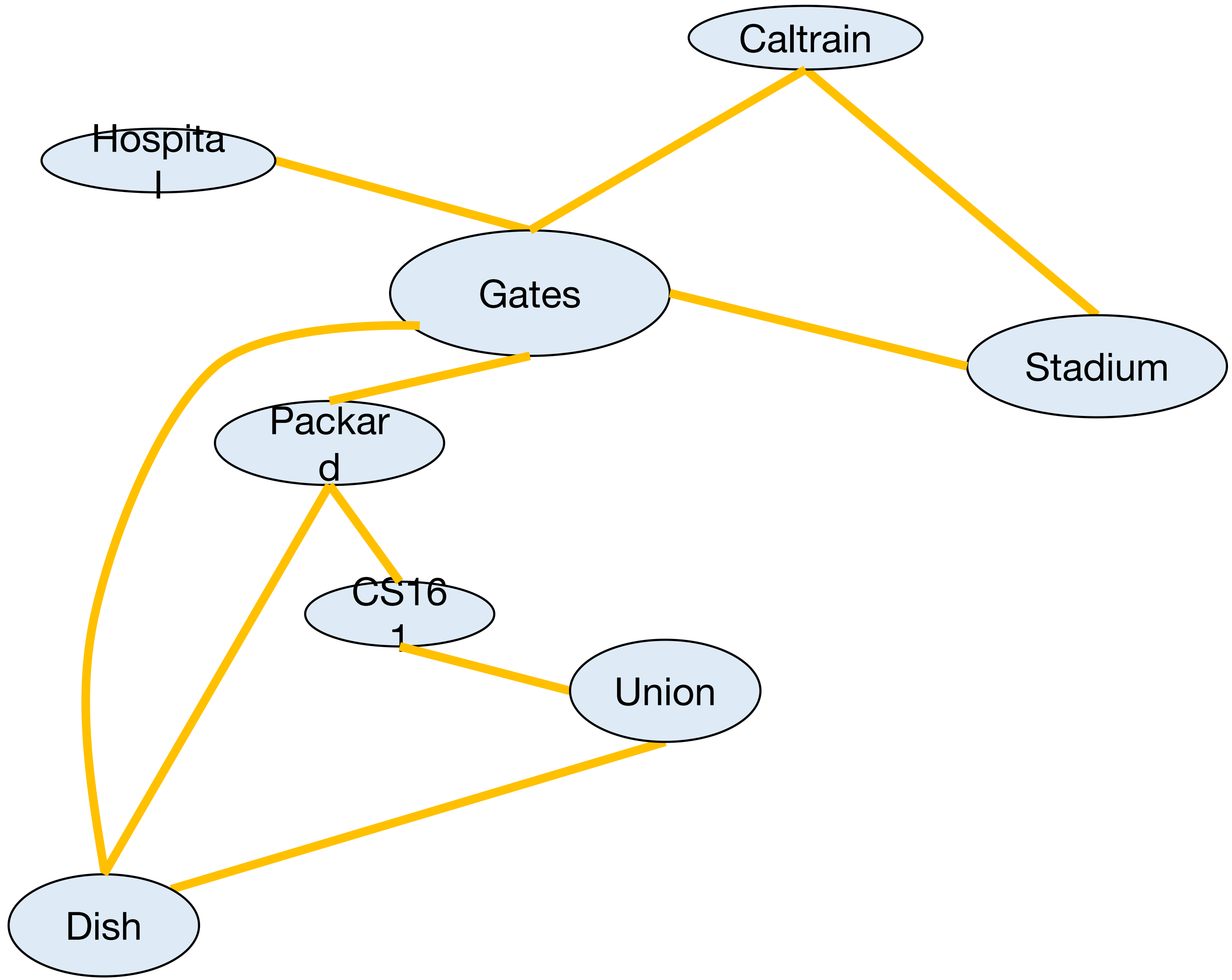
Today

- What if the graphs are weighted?

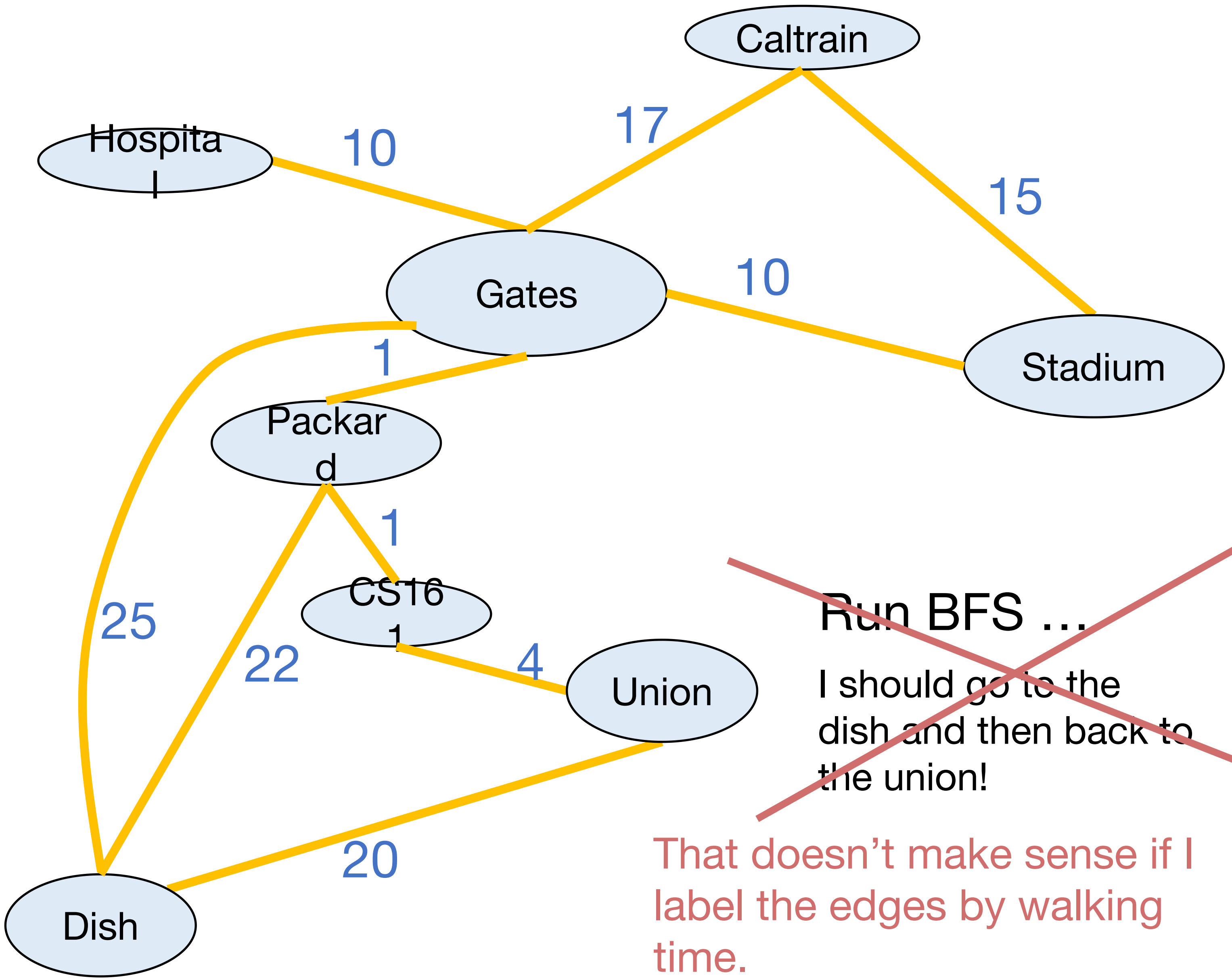
- Part 1: Dijkstra!
  - This will take most of today's class
- Part 2: Bellman-Ford!
  - Real quick at the end if we have time!
  - We'll come back to Bellman-Ford in more detail, so today is just a taste.


Caltrain
Hospita
l
Stadium
Gates
Packar
d
CS16
1
YOU
ARE
HERE
Union
Dish
INSET 1
Lagunita
Map effective 6/1/2016
Latest version:
transportation.stanford.edu/maps
See reverse side for building and street index

Just the graph



Shortest path from Gates to the Union?



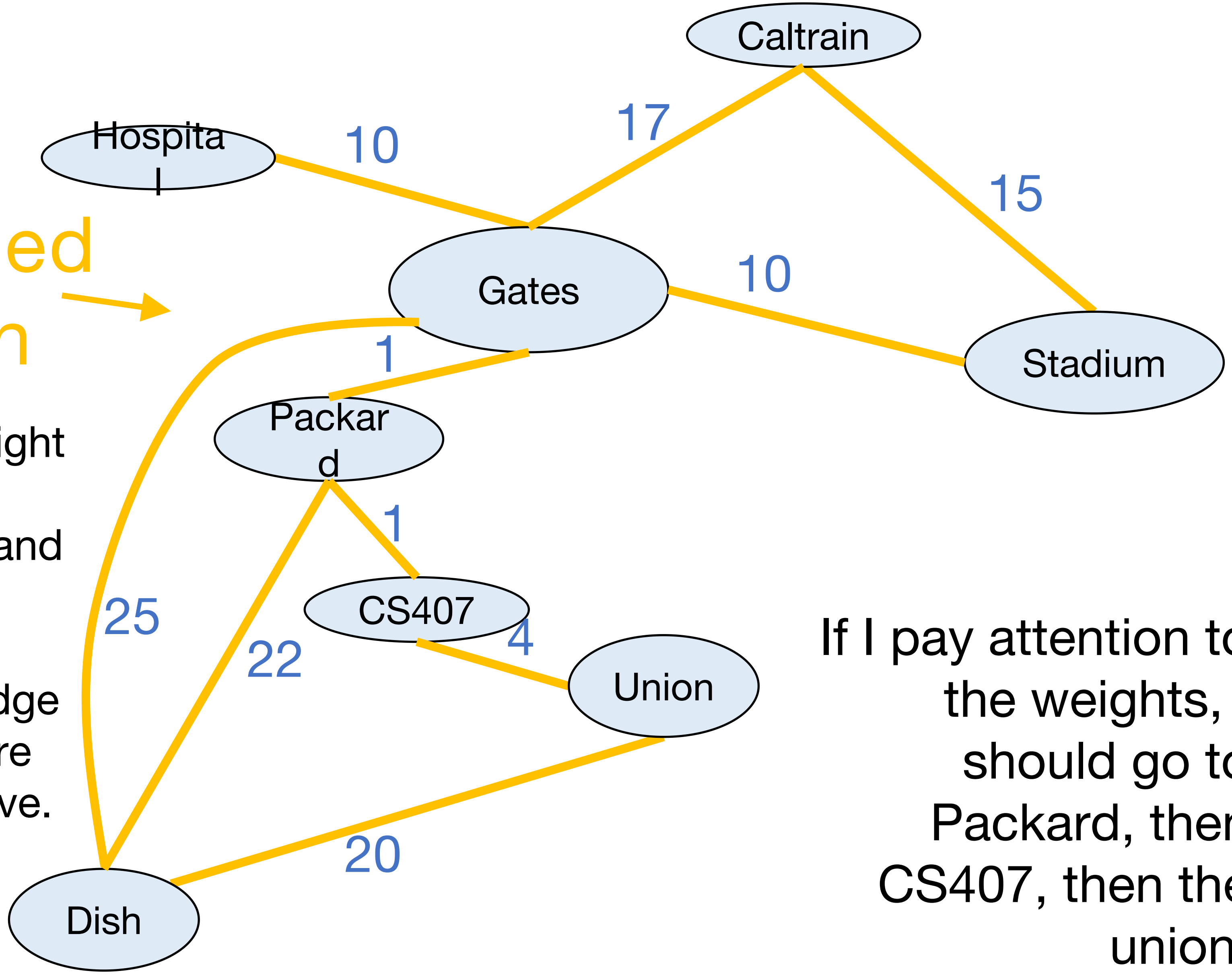
~~Run BFS …~~

~~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.

Shortest path from Gates to the Union?

weighted graph



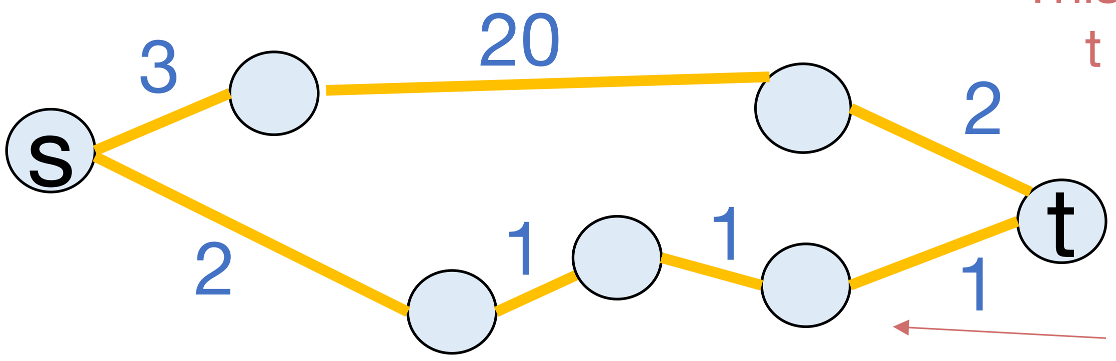
w(u,v) = weight of edge between u and v.

For now, edge weights are non-negative.

If I pay attention to the weights, I should go to Packard, then CS407, then the union.

Shortest path problem

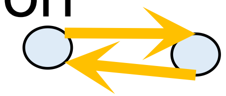
- What is the shortest path between u and v in a weighted graph?
  - the cost of a path is the sum of the weights along that path
  - The shortest path is the one with the minimum cost.



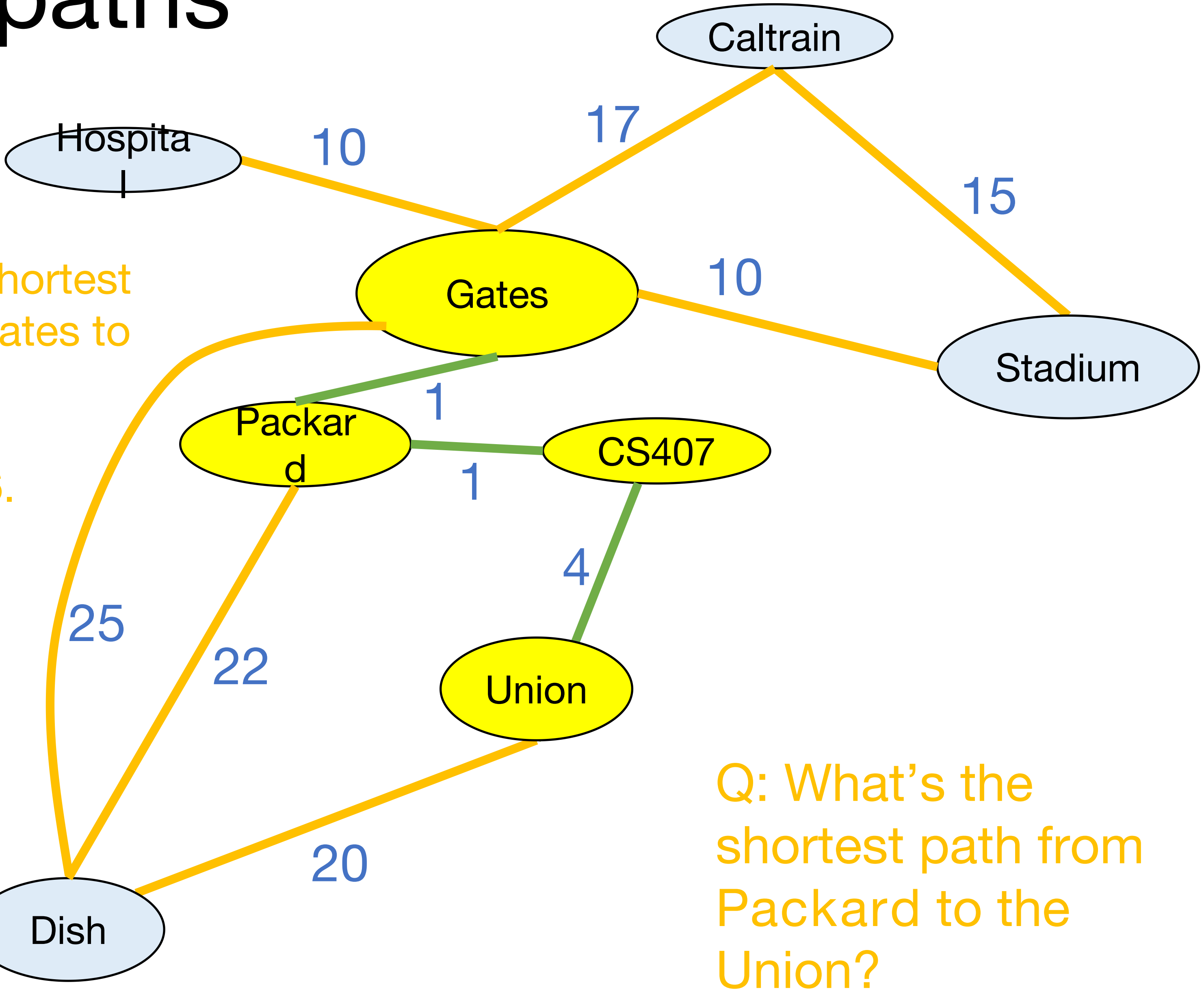
This path from s to t has cost 25.

This path is shorter, it has cost 5.

- 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.

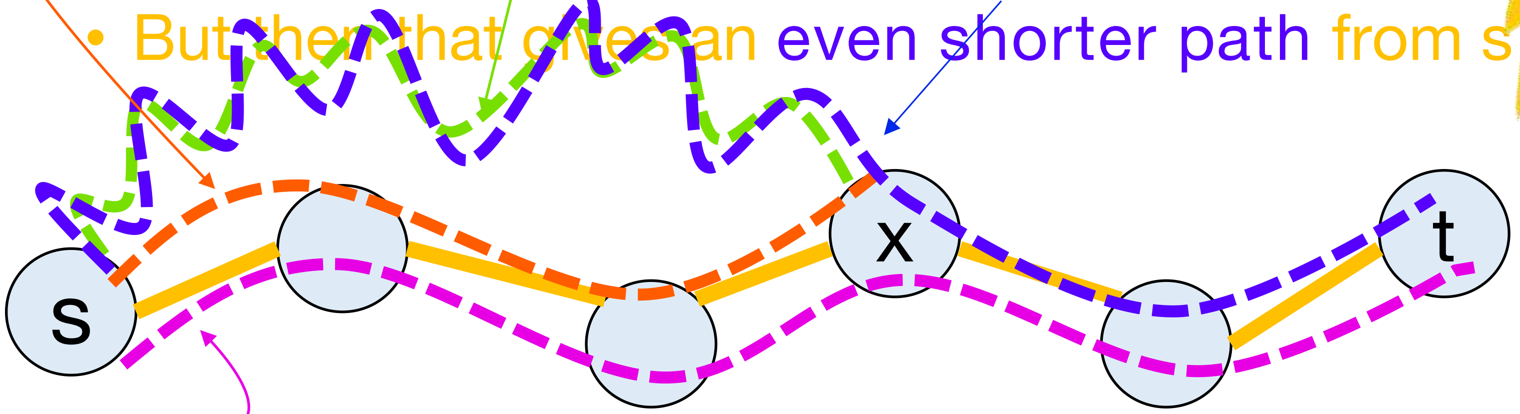
Q: What's the shortest path from Packard to the Union?

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

CONTRADICTIO
N!!



Single-source shortest-path problem

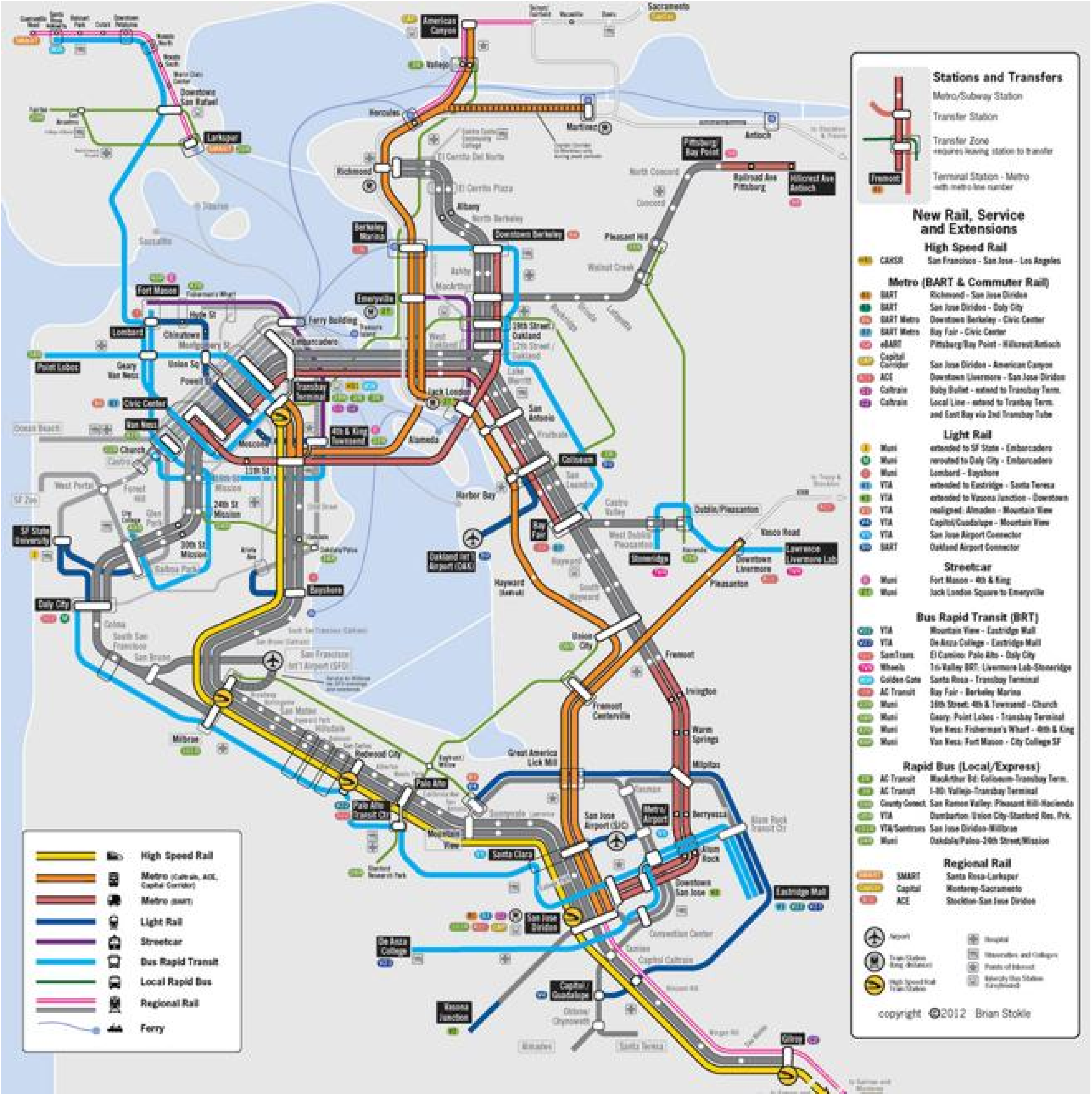
- I want to know the shortest path from one vertex (Gates) to all other vertices.

Destination	Cost	To get there
Packard	1	Packard
CS407	2	Packard-CS407
Hospital	10	Hospital
Caltrain	17	Caltrain
Union	6	Packard-CS407-Union
Stadium	10	Stadium
Dish	23	Packard-Dish

(Not necessarily stored as a table – how this information is represented will depend on the

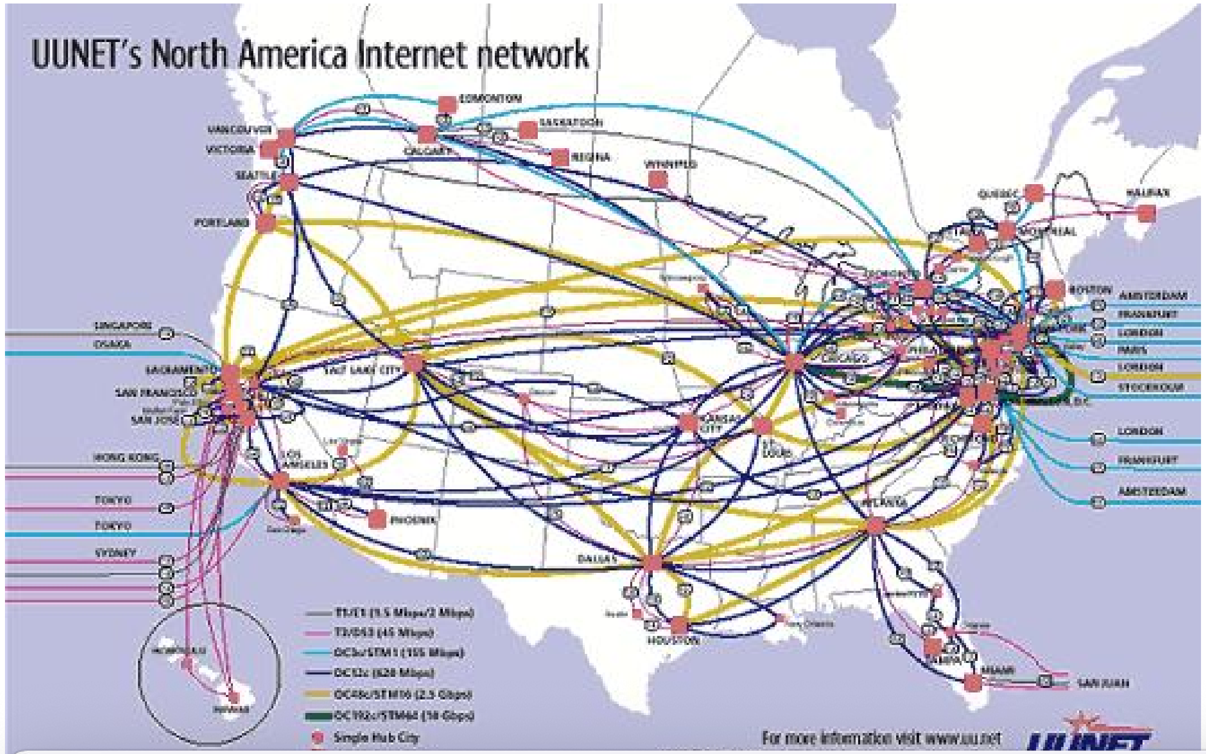
Example

- “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?



moses — traceroute -a www.ethz.ch — 103×19

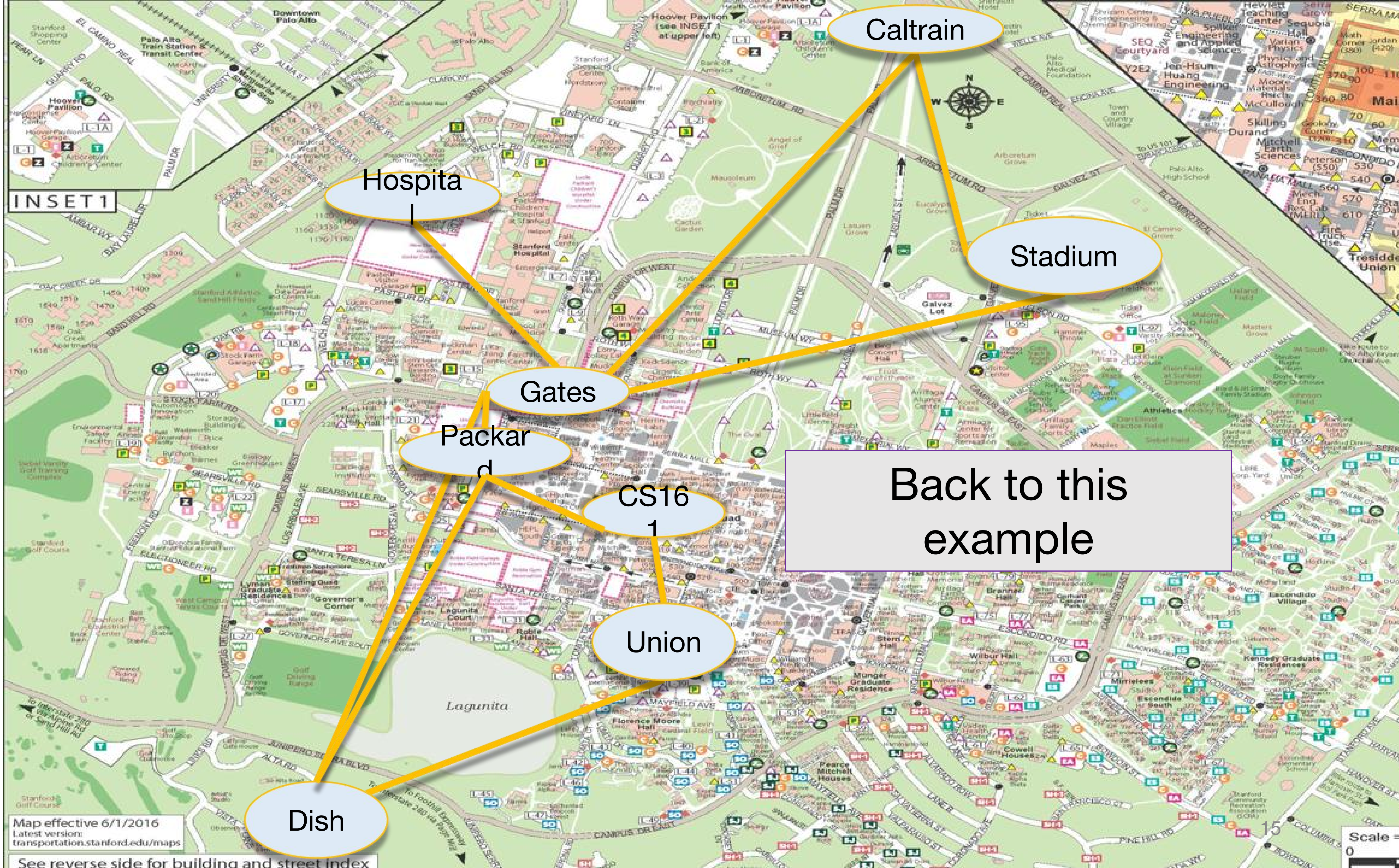
```
Last login: Mon Feb  7 09:27:47 on ttys003
moses@Mosess-MacBook-Pro ~ % traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1  [AS0] 192.168.7.1 (192.168.7.1)  3.898 ms  2.066 ms  2.881 ms
 2  [AS0] 192.168.0.1 (192.168.0.1)  2.897 ms  4.720 ms  3.108 ms
 3  [AS0] 10.127.252.2 (10.127.252.2)  57.256 ms  5.571 ms  4.268 ms
 4  [AS32] he-rtr.stanford.edu (128.12.0.209)  4.039 ms  11.471 ms  4.628 ms
 5  [AS6939] 100gigabitethernet5-1.core1.pao1.he.net (184.105.177.237)  4.648 ms  3.839 ms  4.300 ms
 6  [AS6939] 100ge9-2.core1.sjc2.he.net (72.52.92.157)  5.949 ms  5.291 ms  4.980 ms
 7  [AS6939] 100ge10-2.core1.nyc4.he.net (184.105.81.217)  69.007 ms  66.575 ms  67.516 ms
 8  [AS6939] 100ge7-1.core1.lon2.he.net (72.52.92.165)  268.329 ms  191.401 ms  203.048 ms
 9  [AS6939] port-channel2.core3.lon2.he.net (184.105.64.2)  205.515 ms  350.183 ms  206.164 ms
10  [AS6939] port-channel12.core2.ams1.he.net (72.52.92.214)  144.263 ms  143.638 ms  146.034 ms
11  [AS1200] swice1-100ge-0-3-0-1.switch.ch (80.249.208.33)  161.119 ms  208.169 ms  185.621 ms
12  [AS559] swice4-b4.switch.ch (130.59.36.70)  219.228 ms  203.833 ms  204.402 ms
13  [AS559] swibf1-b2.switch.ch (130.59.36.113)  184.671 ms  204.955 ms  204.671 ms
14  [AS559] swiez3-b5.switch.ch (130.59.37.6)  205.079 ms  164.116 ms  245.086 ms
15  [AS559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1)  204.296 ms  164.770 ms  162.913 ms
16  [AS559] rou-fw-rz-rz-gw.ethz.ch (192.33.92.169)  165.148 ms  322.839 ms  204.627 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 think about whether I should bike to Berkeley…
  - More seriously, the internet.

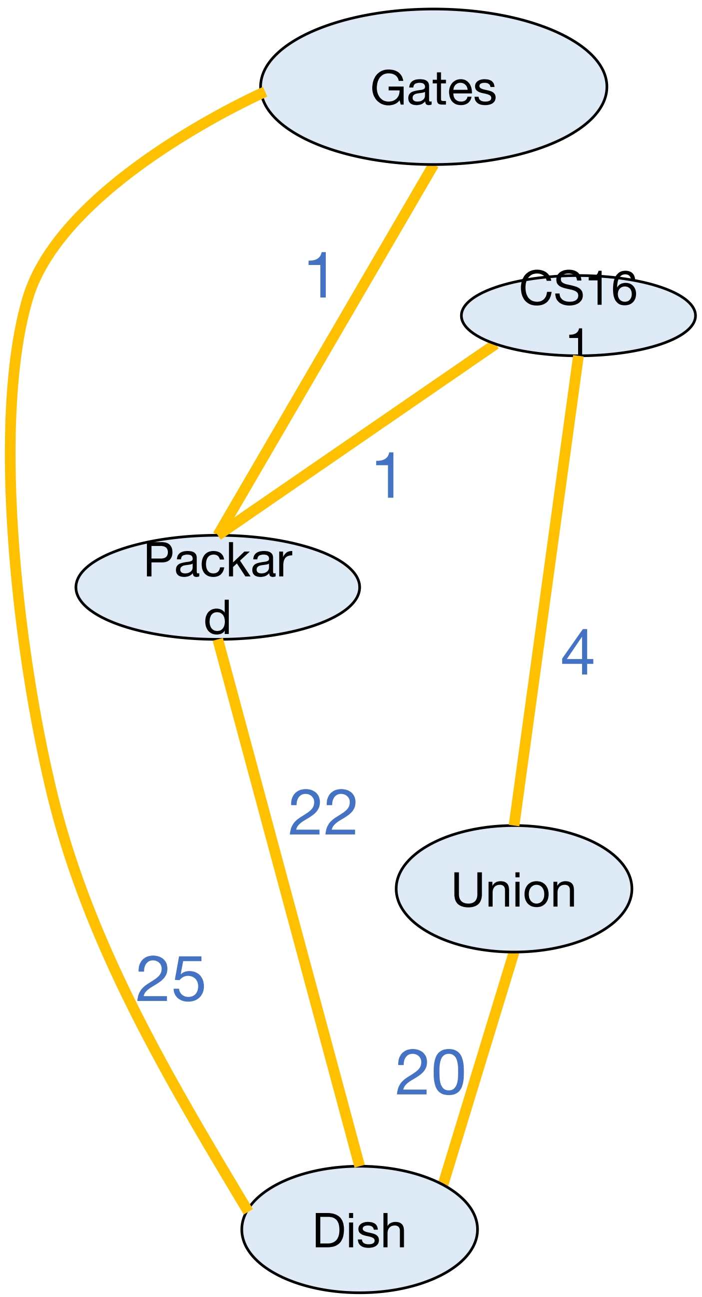
This is a joke.

But let's ignore them for now.

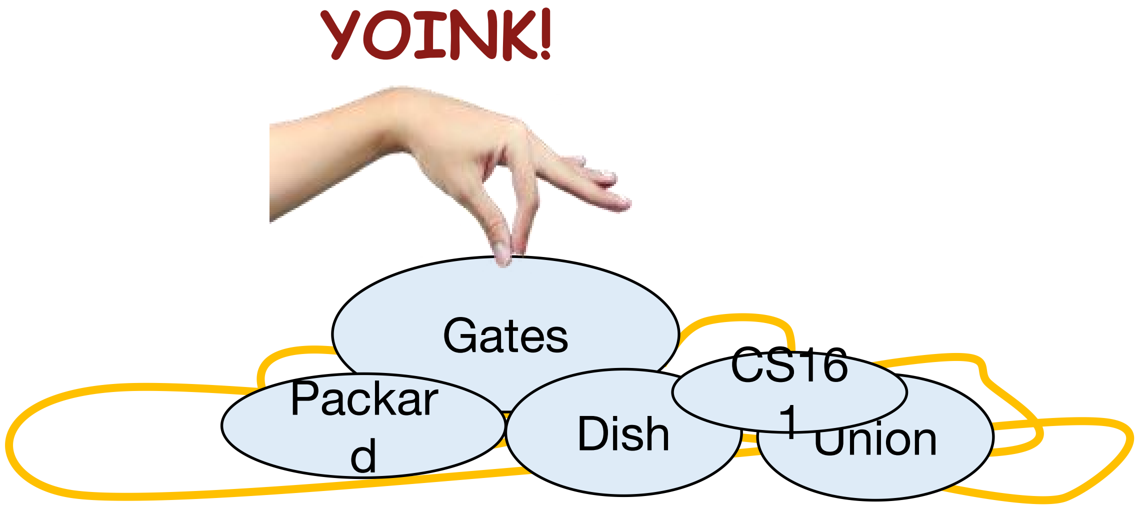

Caltrain
Hospital
Stadium
Gates
Packard
CS161
Union
Dish
Back to this example
INSET 1
Lagunita
Map effective 6/1/2016
Latest version:
transportation.stanford.edu/maps
See reverse side for building and street index

Dijkstra’s algorithm

- Finds shortest paths from Gates to everywhere else.

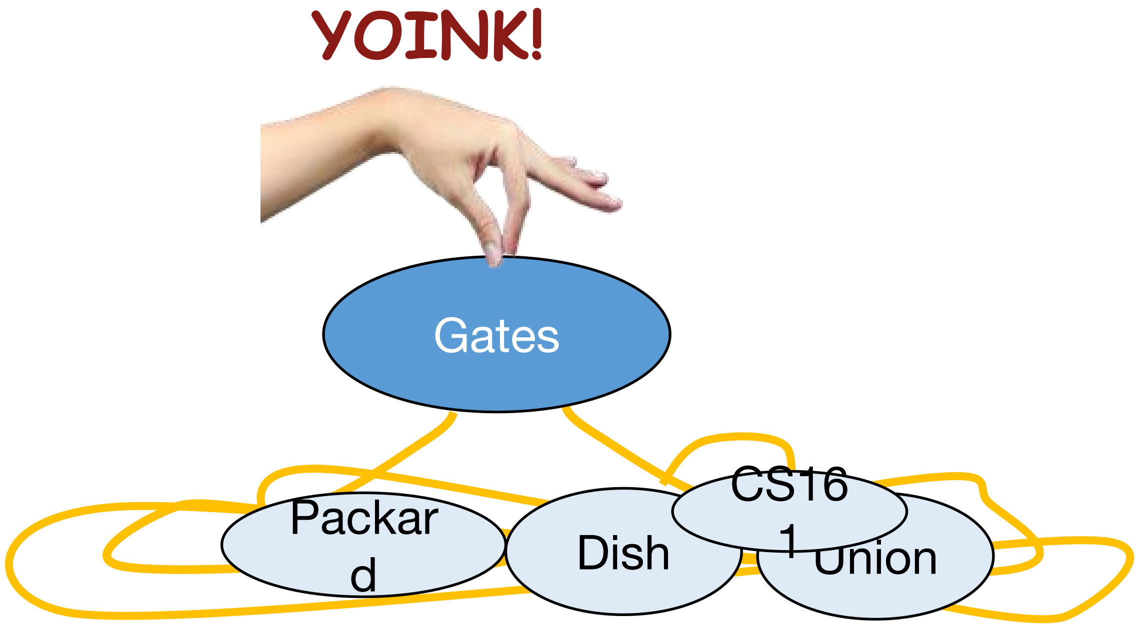


Dijkstra
intuition

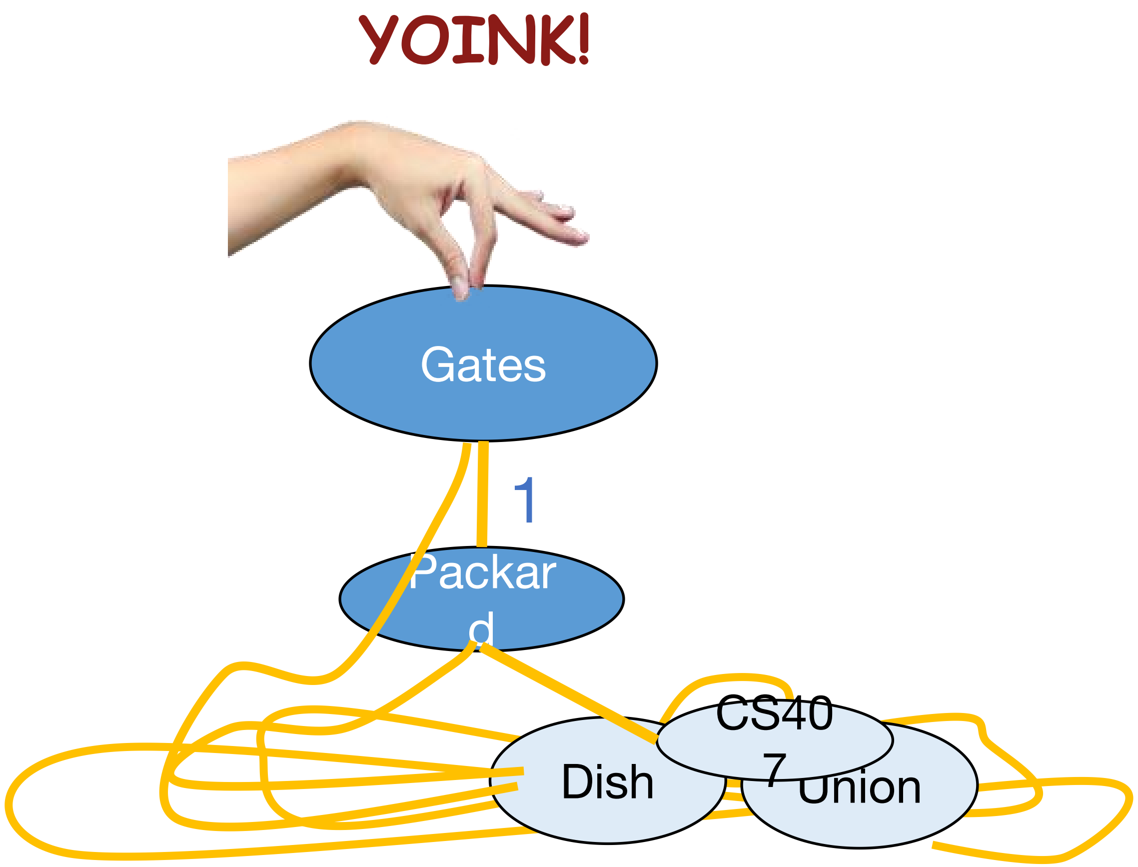


Dijkstra
intuition

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

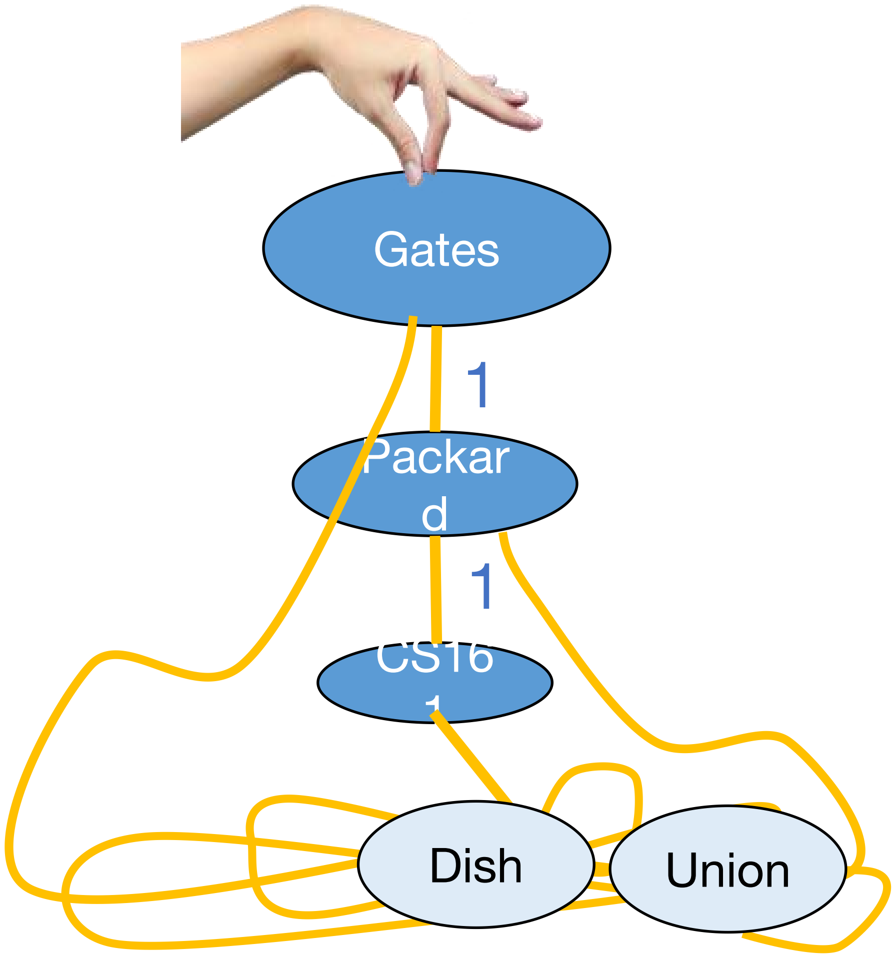


Dijkstra
intuition

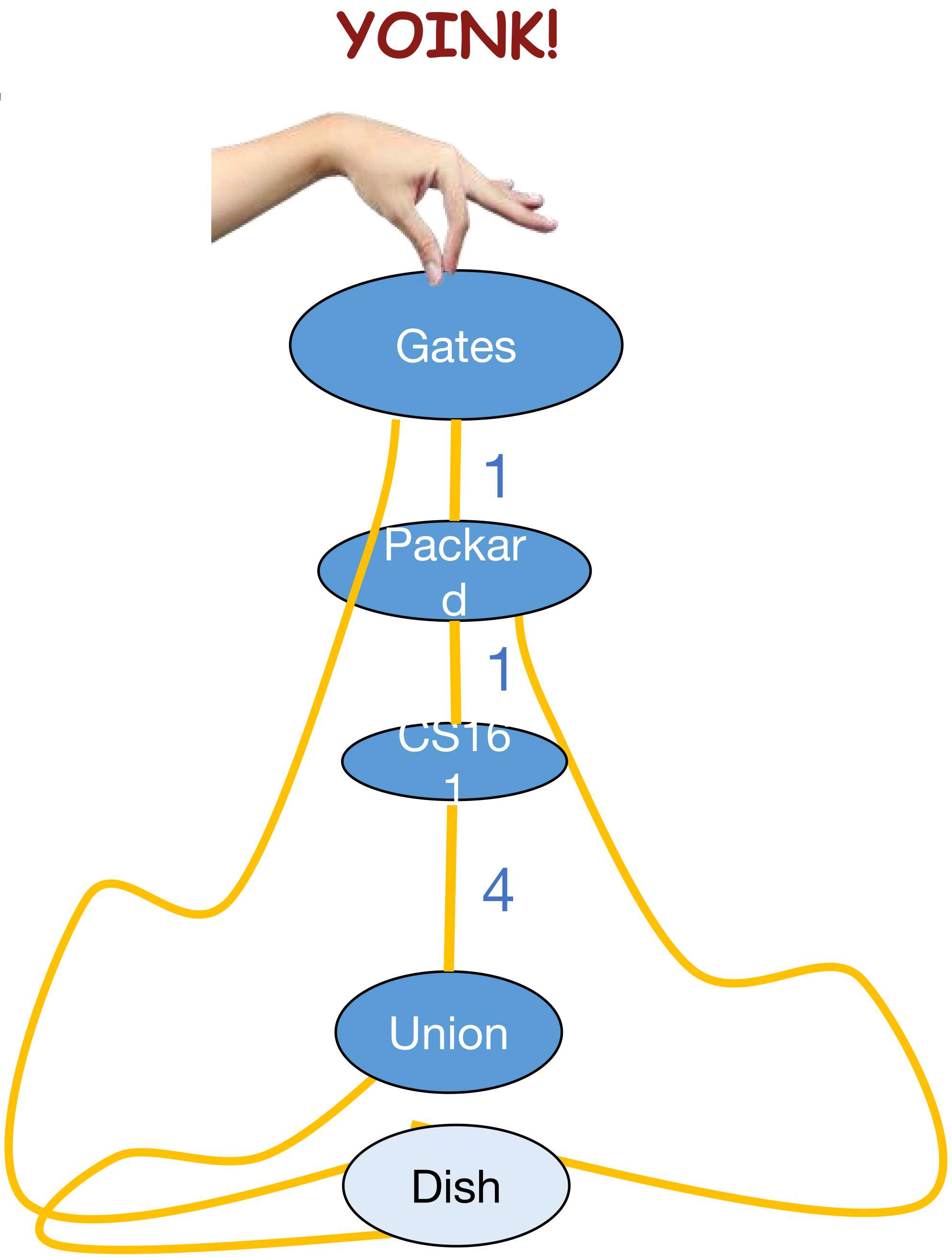


Dijkstra
intuition

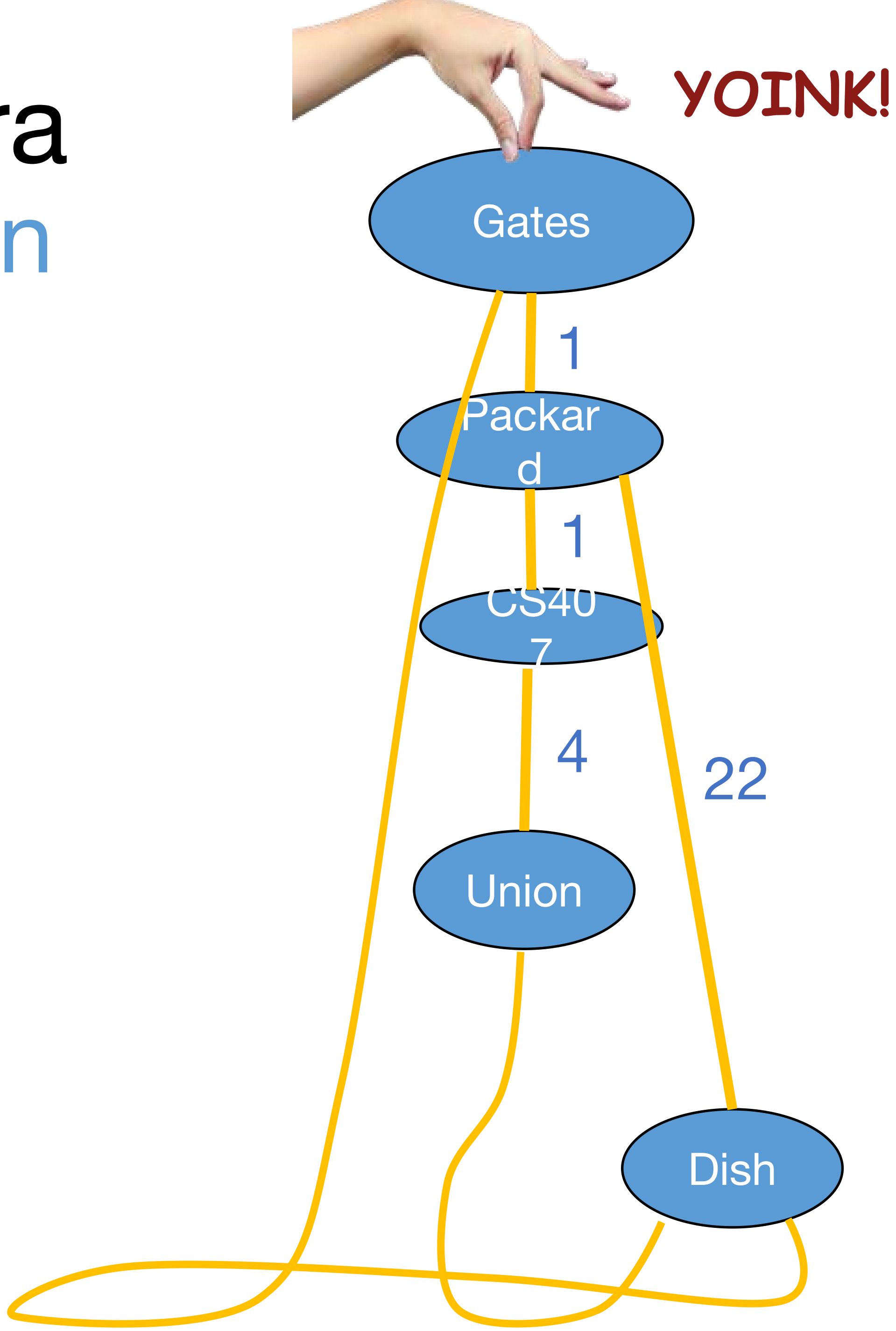
YOINK!



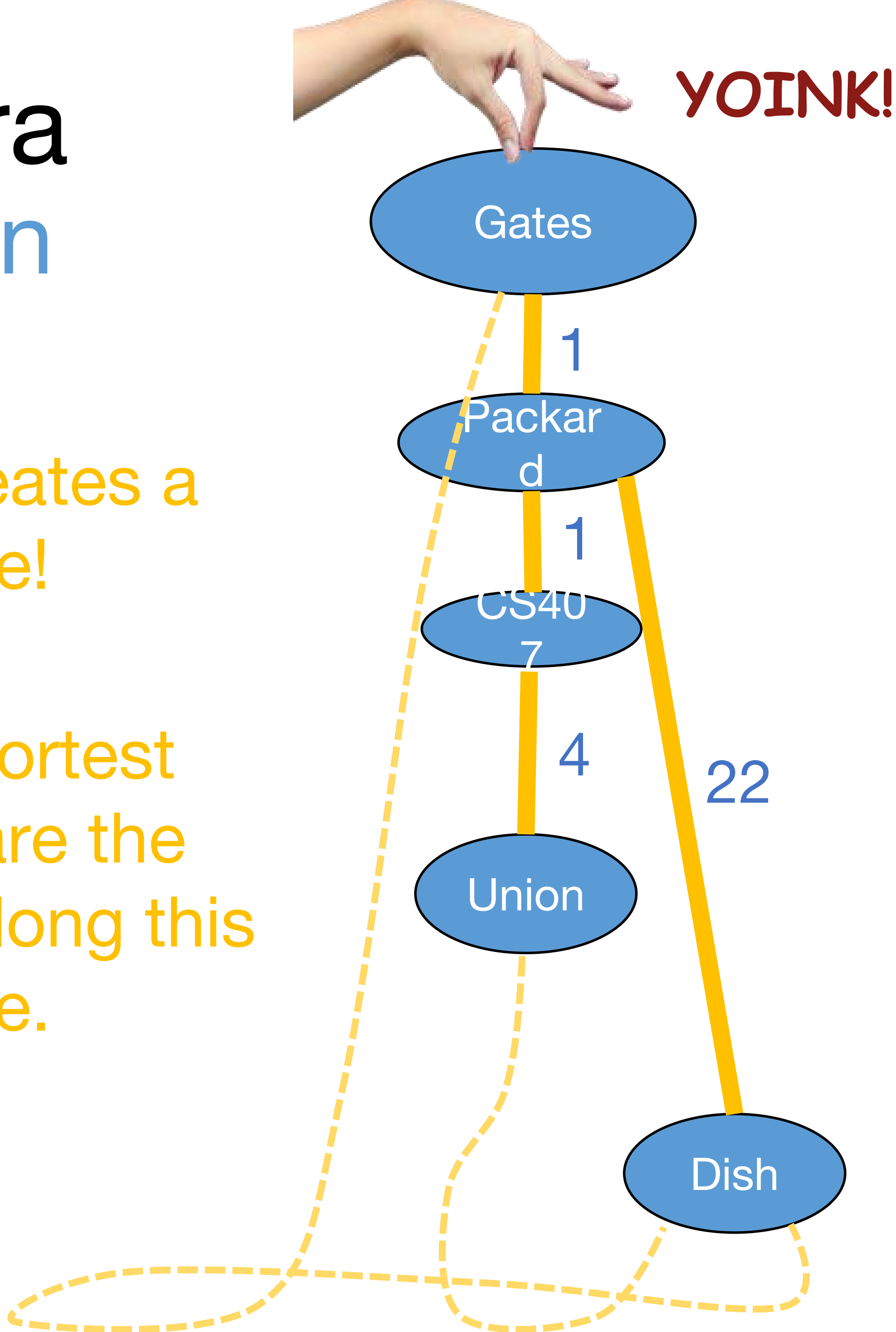
Dijkstra
intuition



Dijkstra
intuition



Dijkstra intuition

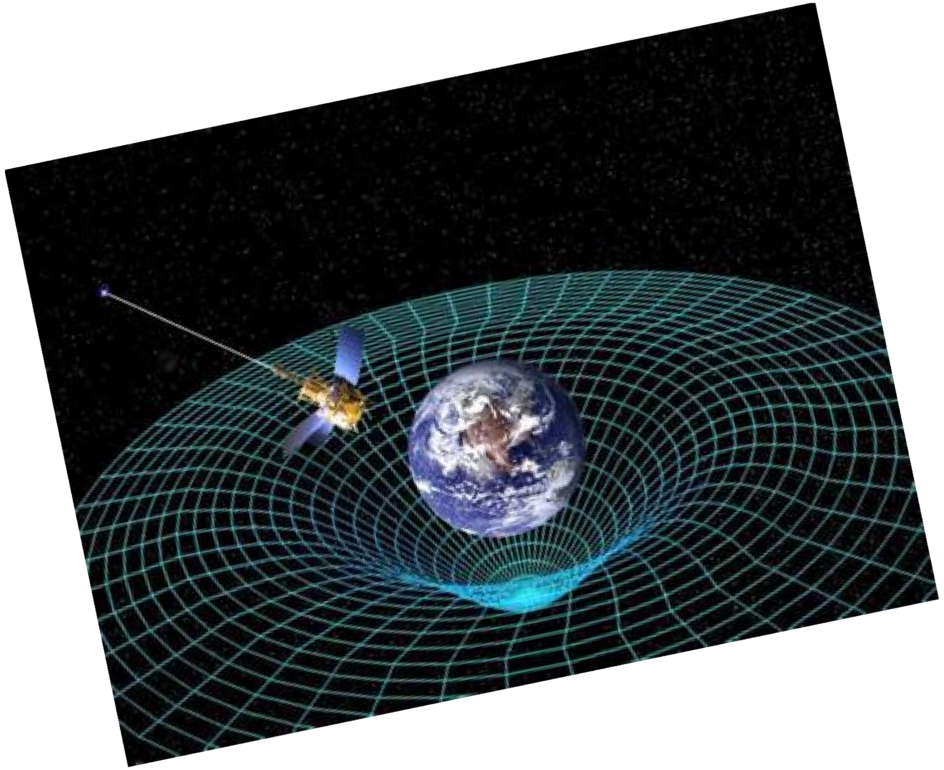


This creates a tree!

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

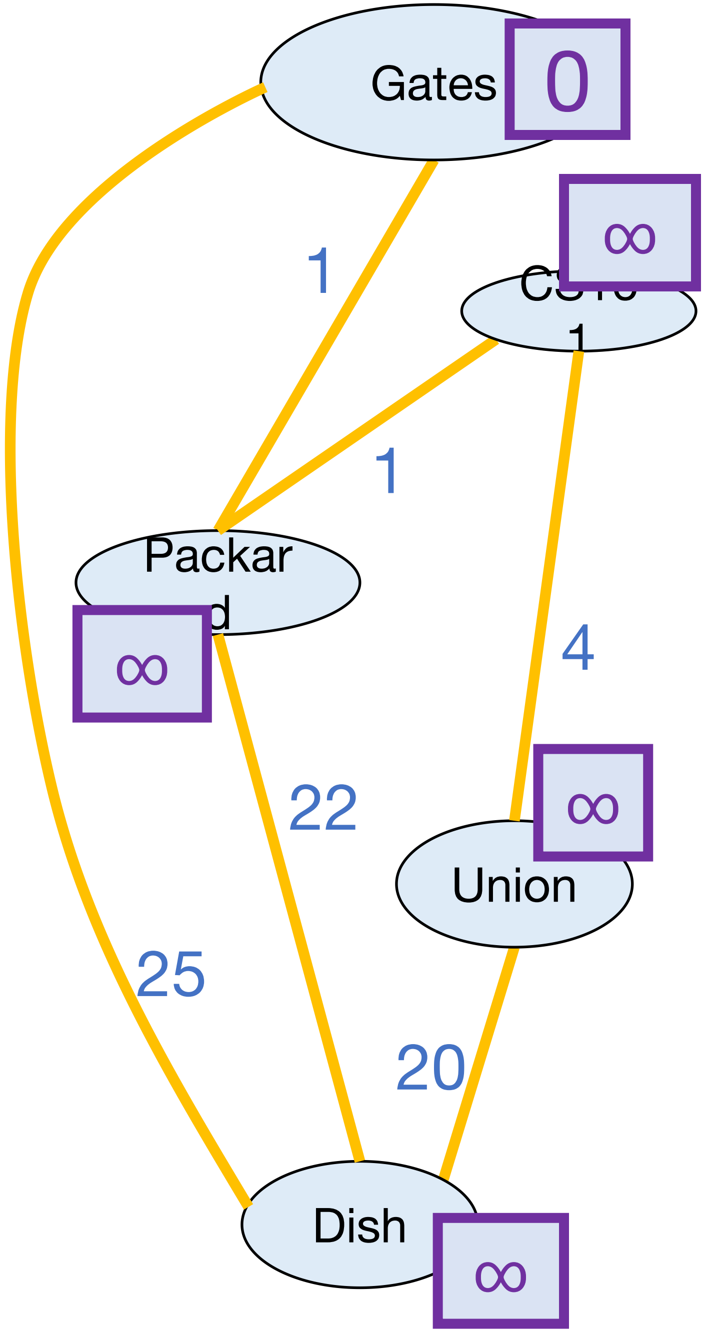
I'm sure

x

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?

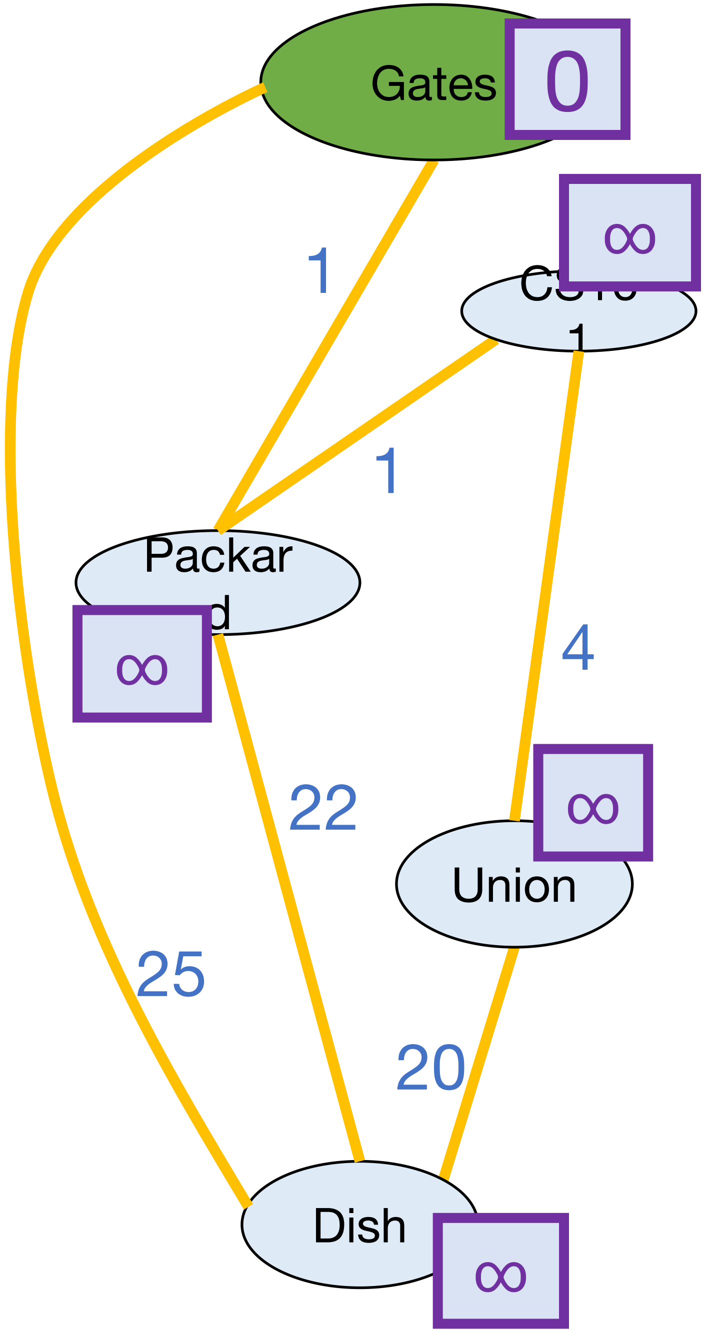
I'm not sure yet

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?

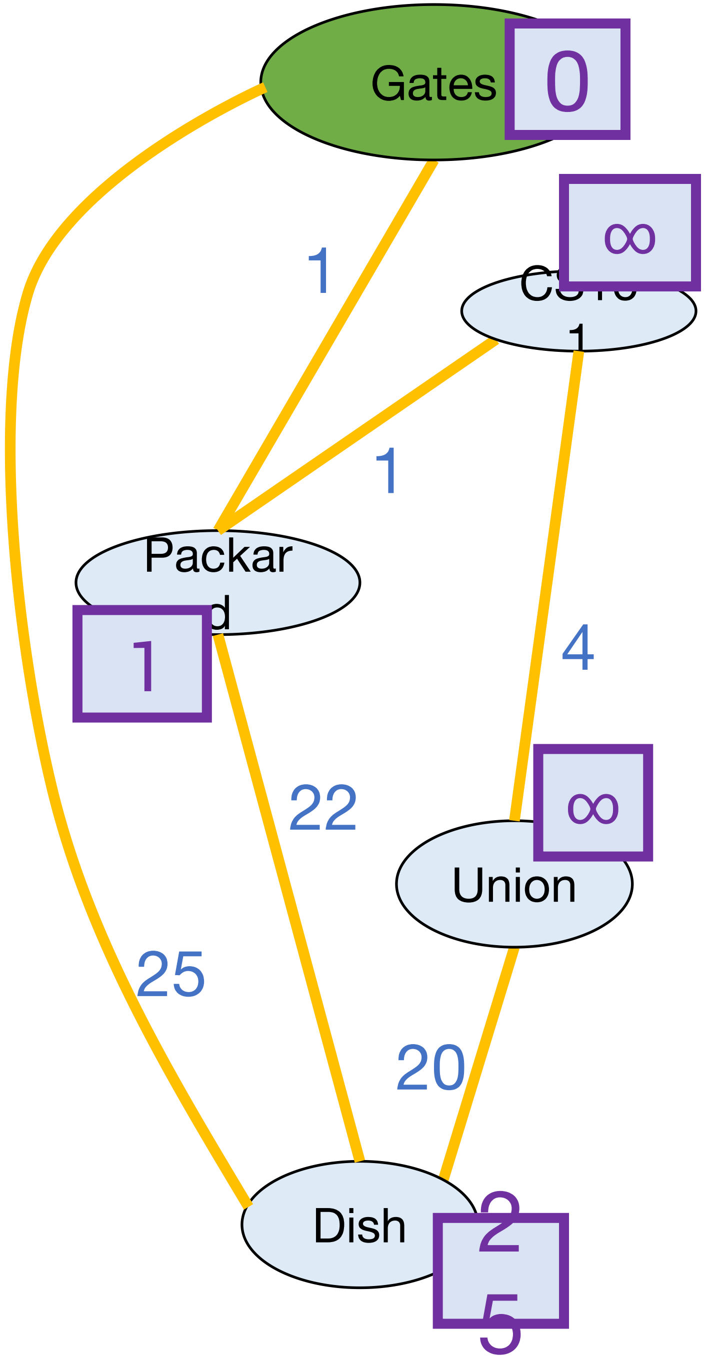
I'm not sure yet

I'm sure

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

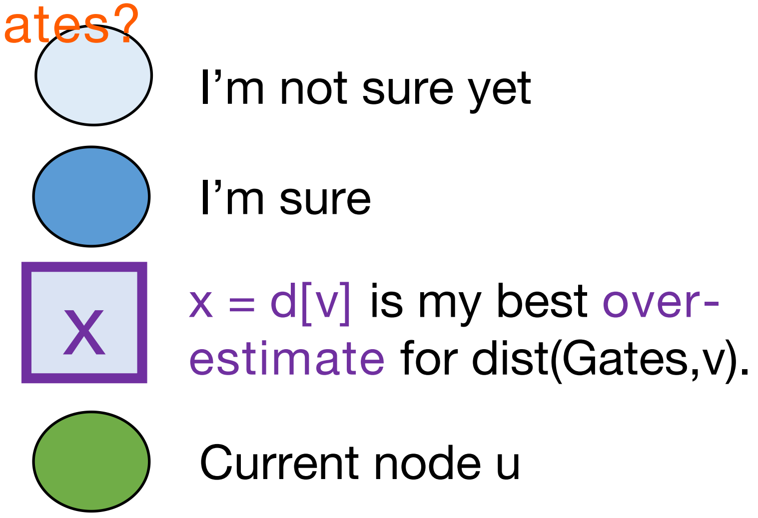
Current node u

- 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.

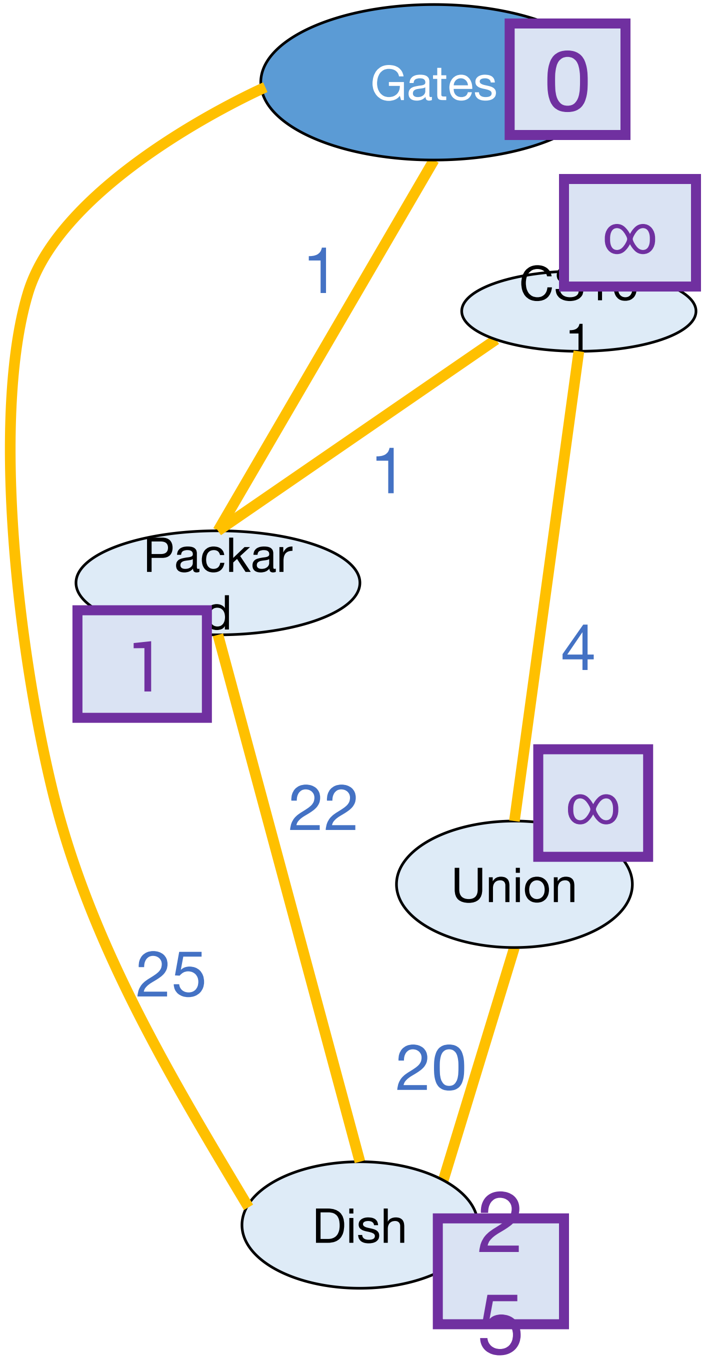


Dijkstra by example

How far is a node from Gates?

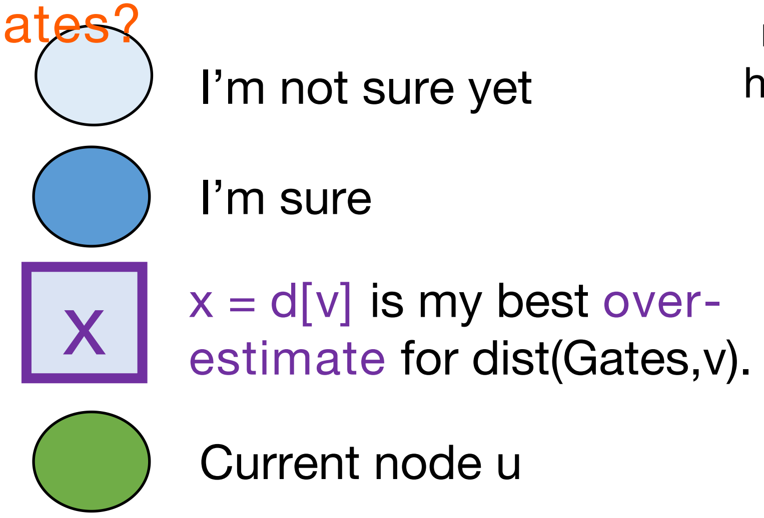


- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as sure.
- Repeat



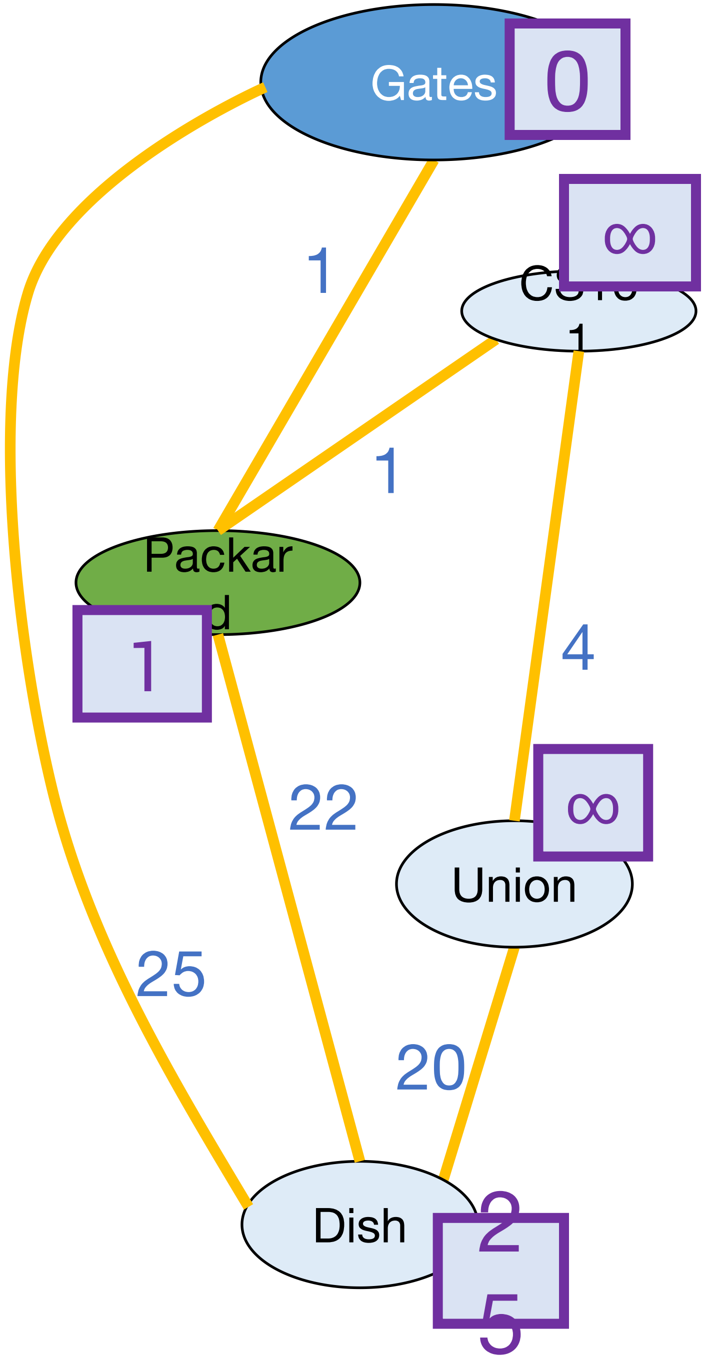
Dijkstra by example

How far is a node from Gates?



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?

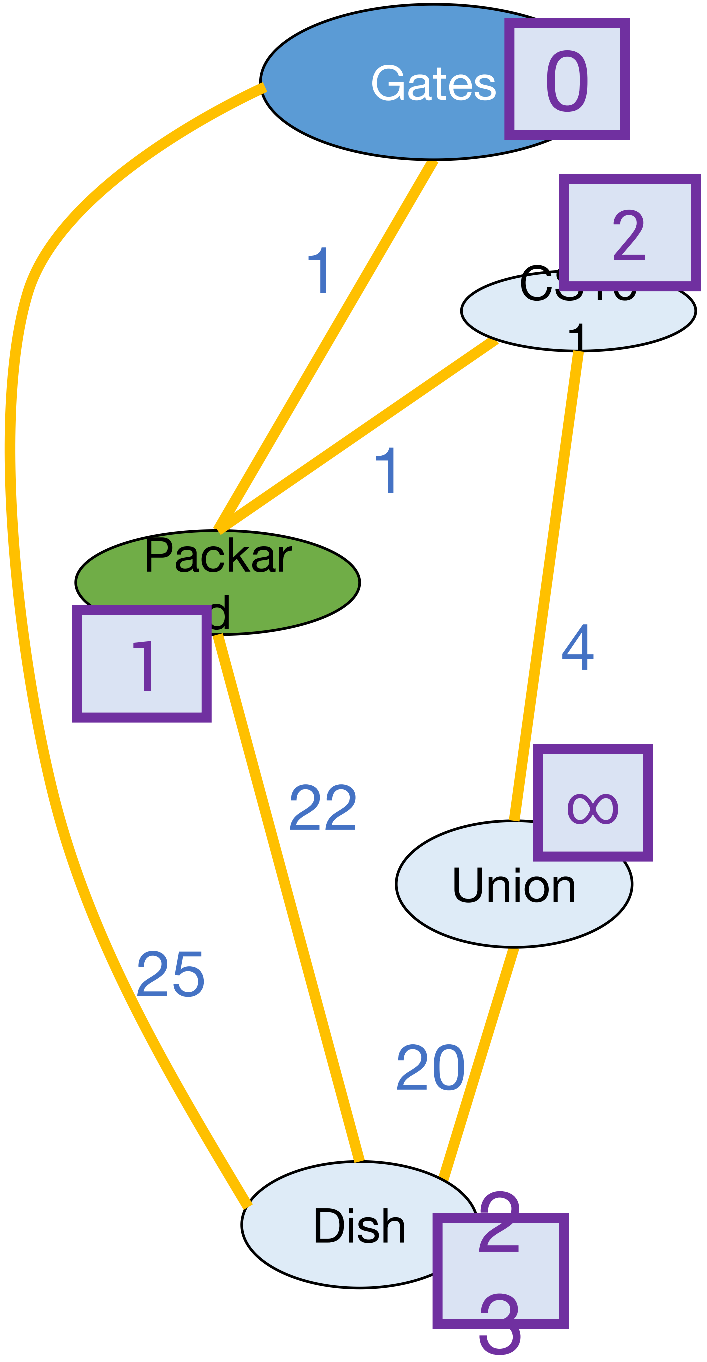
I'm not sure yet

I'm sure

$\boxed{x}$ 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?

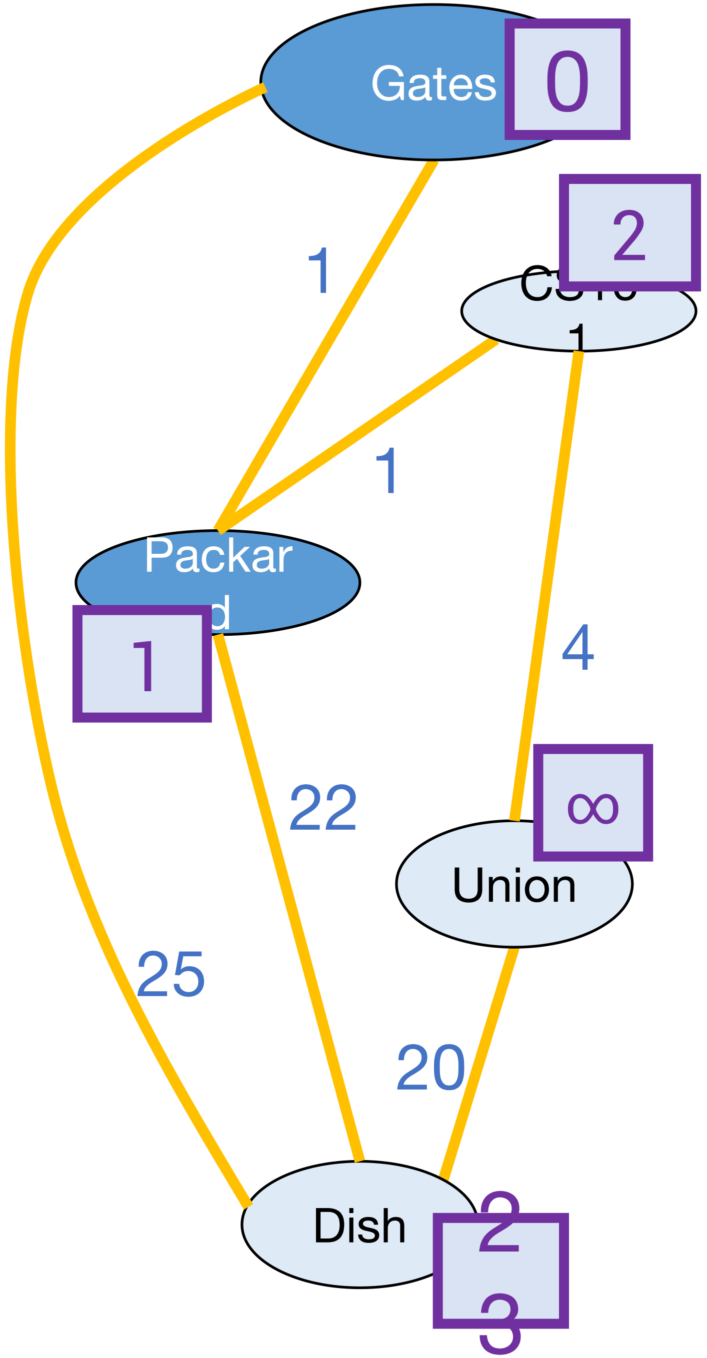
I'm not sure yet

I'm sure

x = 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?

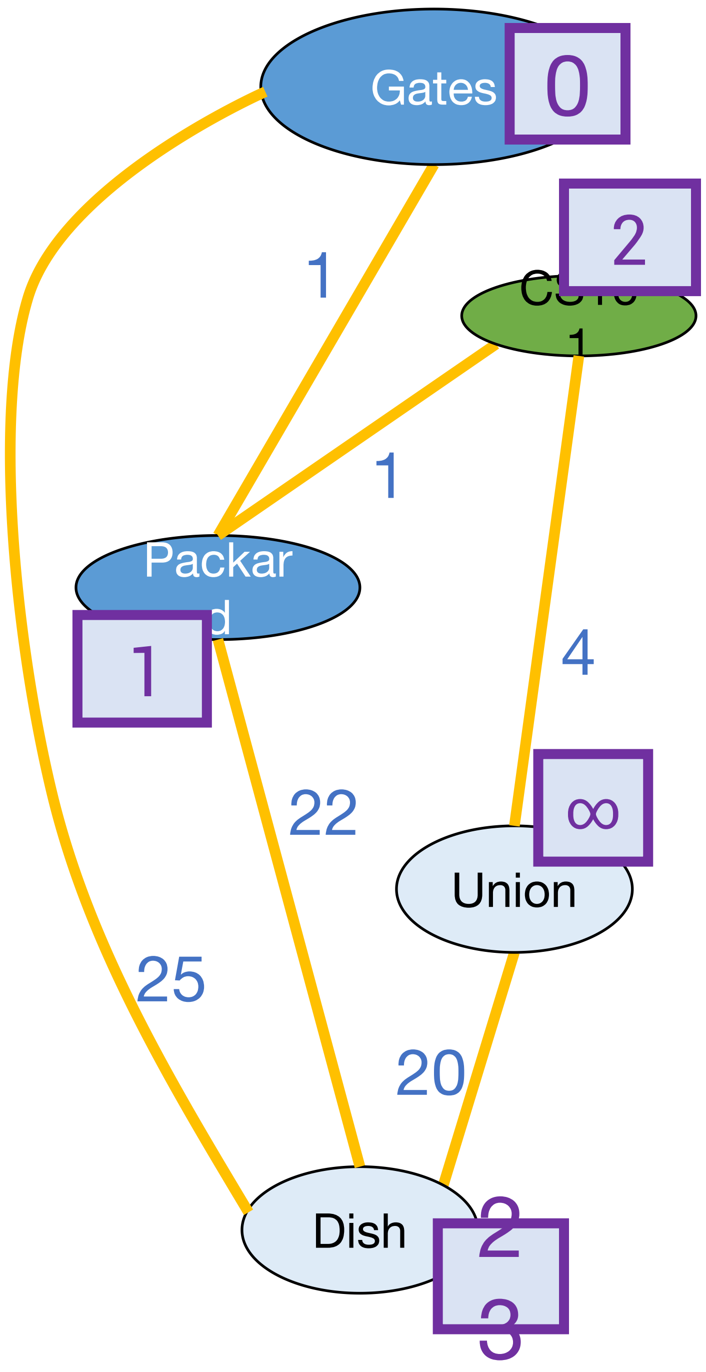
I'm not sure yet

I'm sure

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

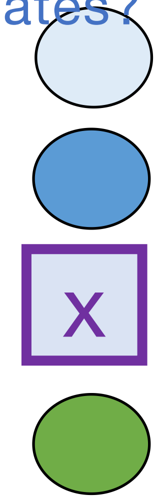
Current node u

- 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?



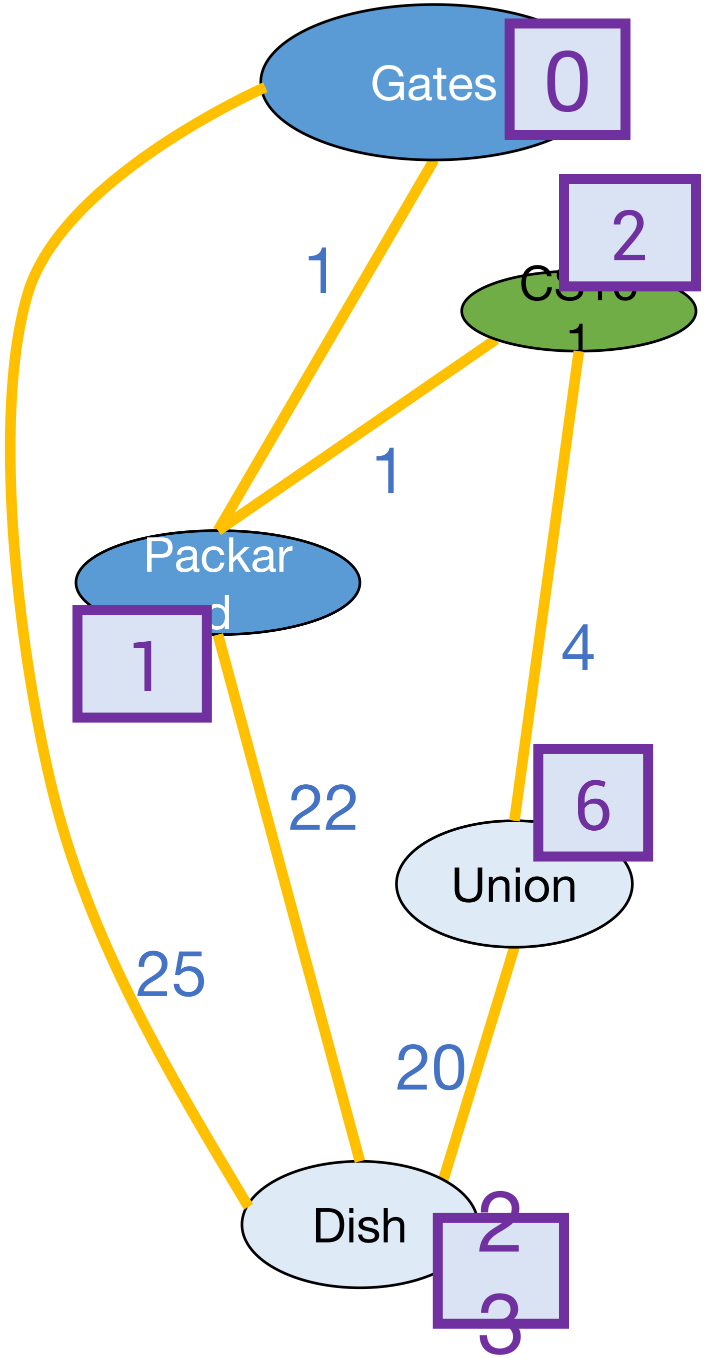
I'm not sure yet

I'm sure

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

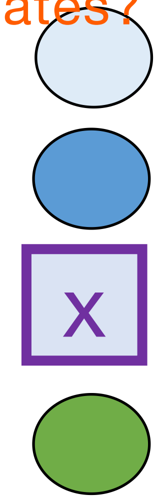
Current node u

- 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?



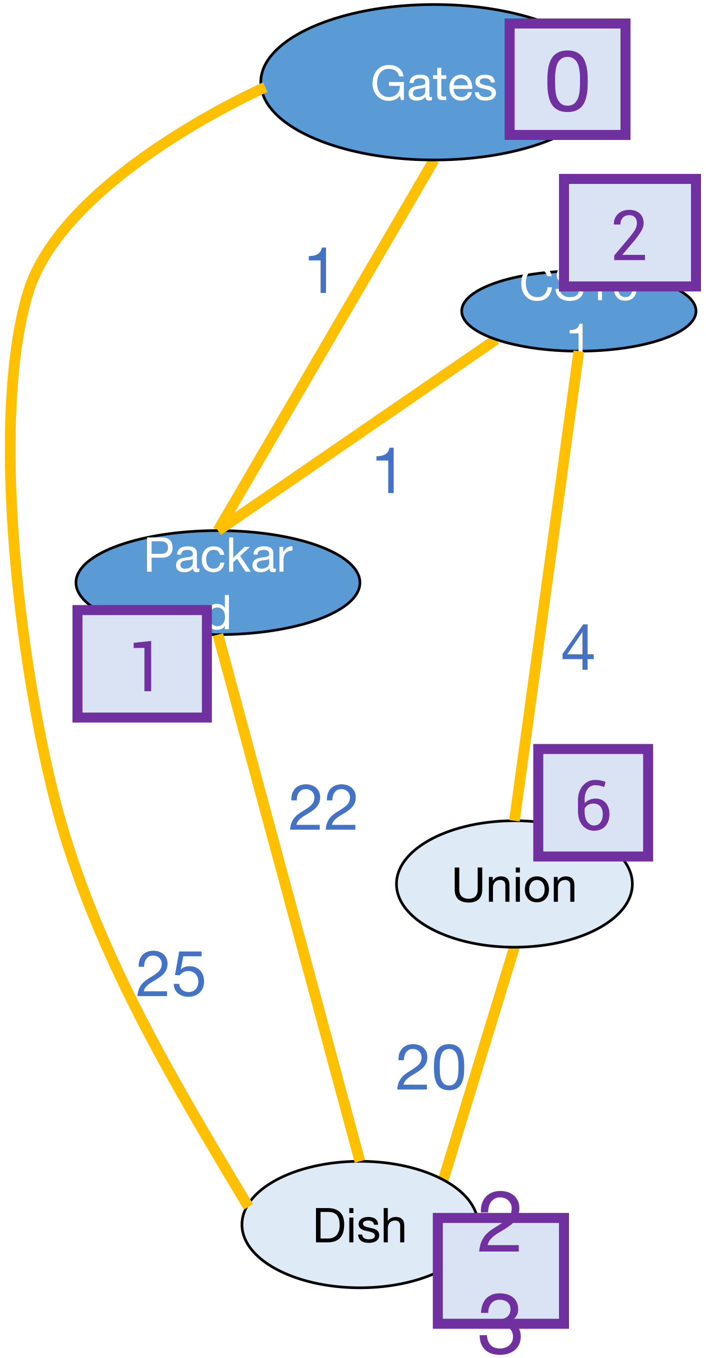
I'm not sure yet

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?

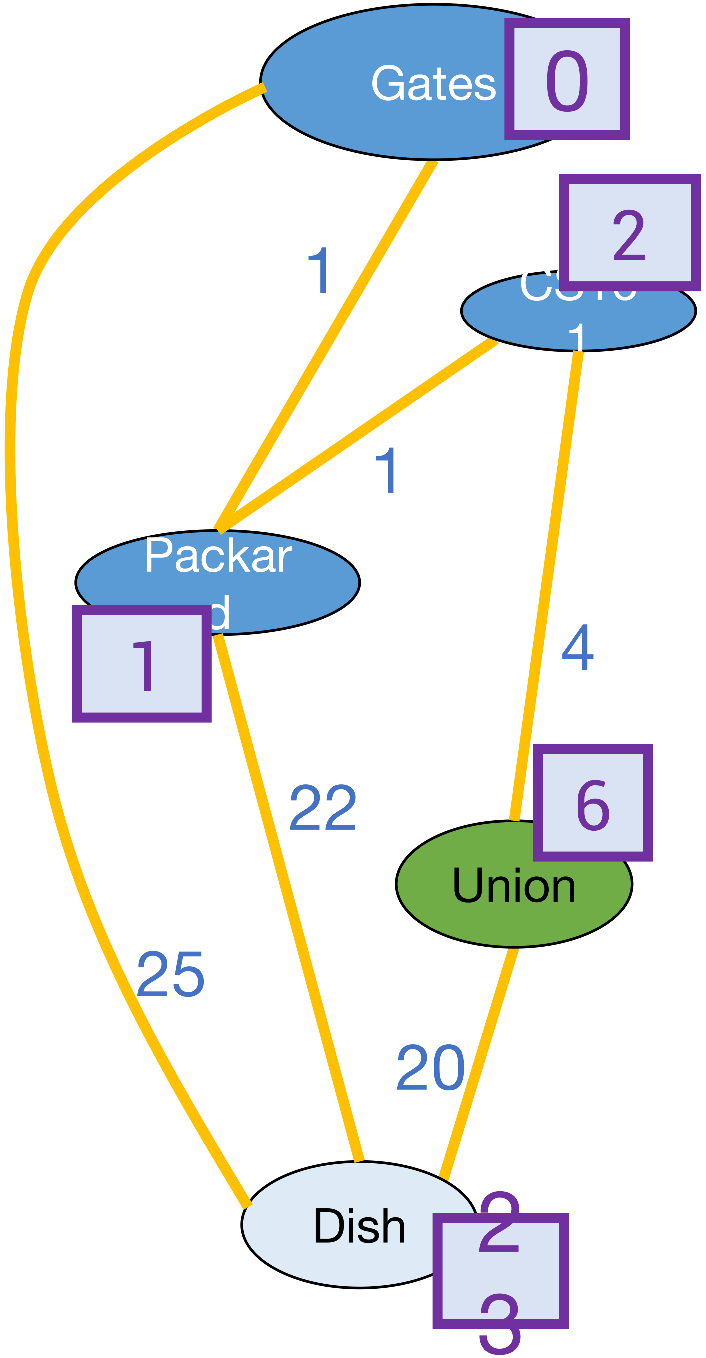
I'm not sure yet

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?

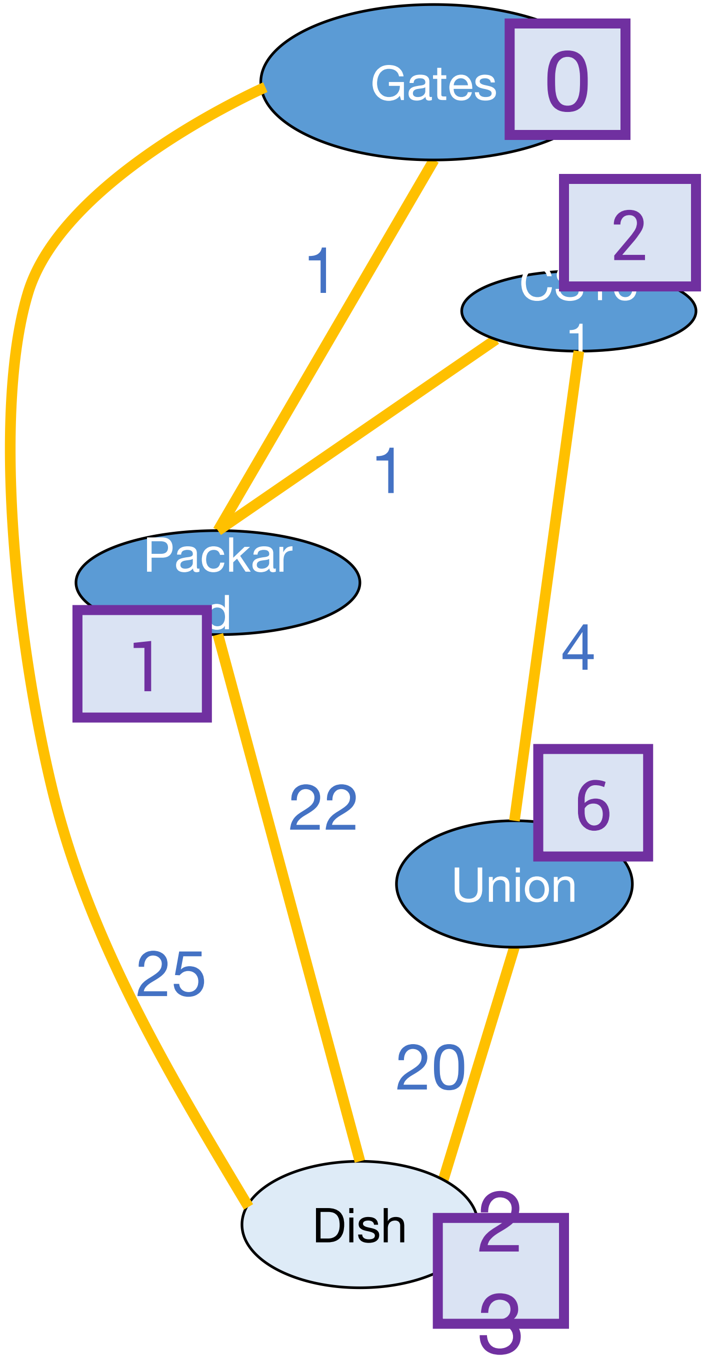
I'm not sure yet

I'm sure

x = 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?

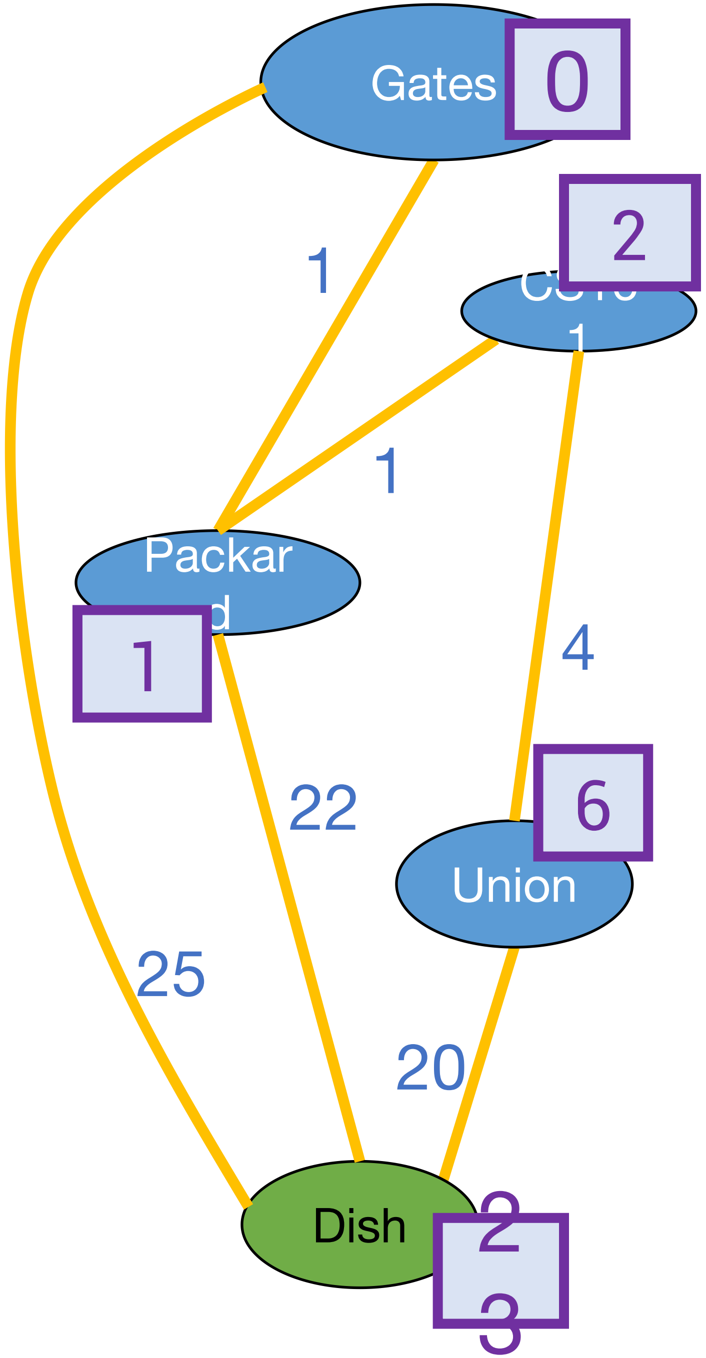
I'm not sure yet

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?

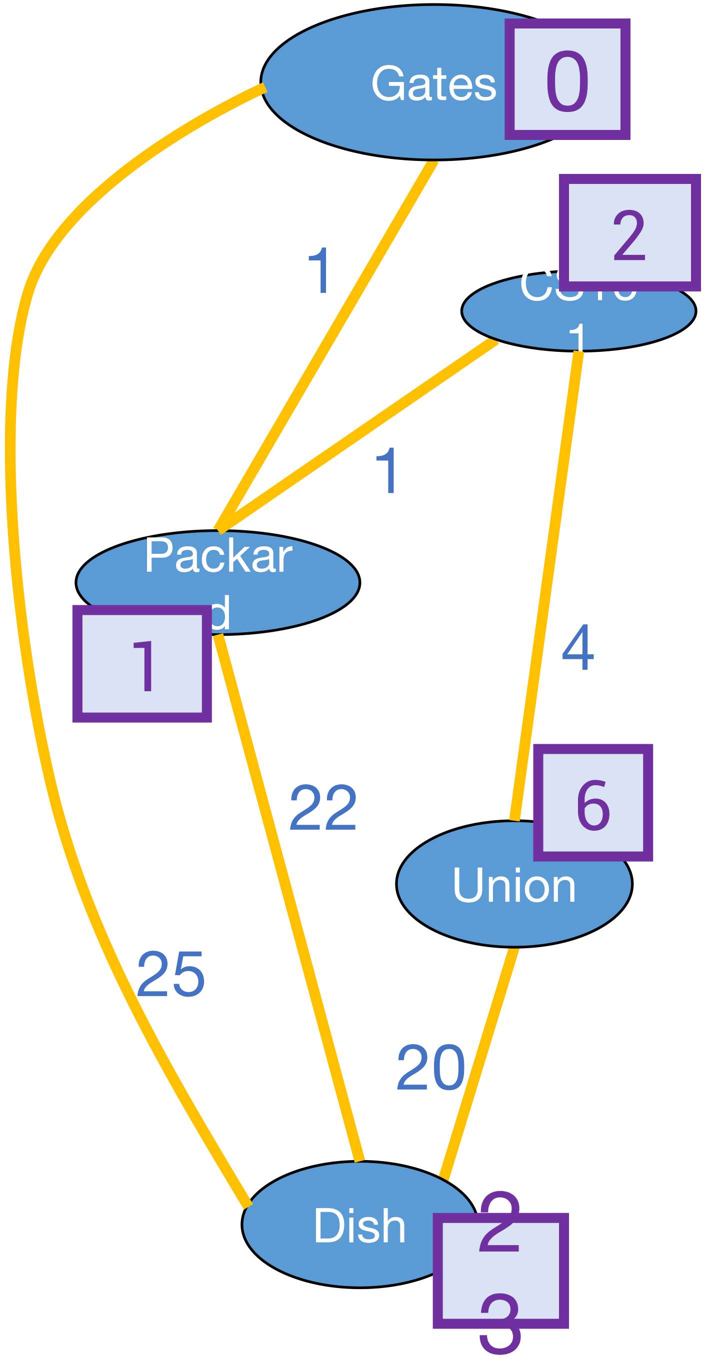
I'm not sure yet

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

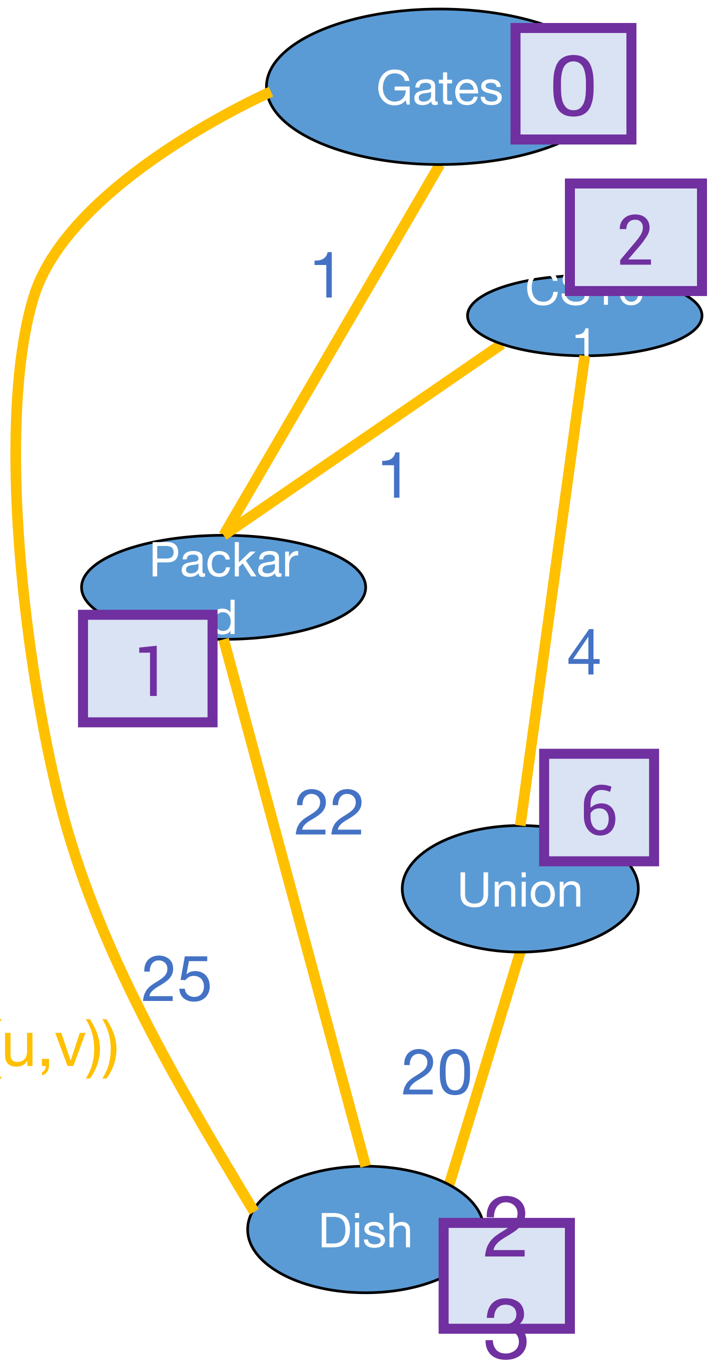
I'm sure

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!

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] ≥ 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] ≥ 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).

Claim 2

Claim 1 + def of algorithm

Next let's prove the claims!

Claim 1

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

Intuition!

Informally:

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

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

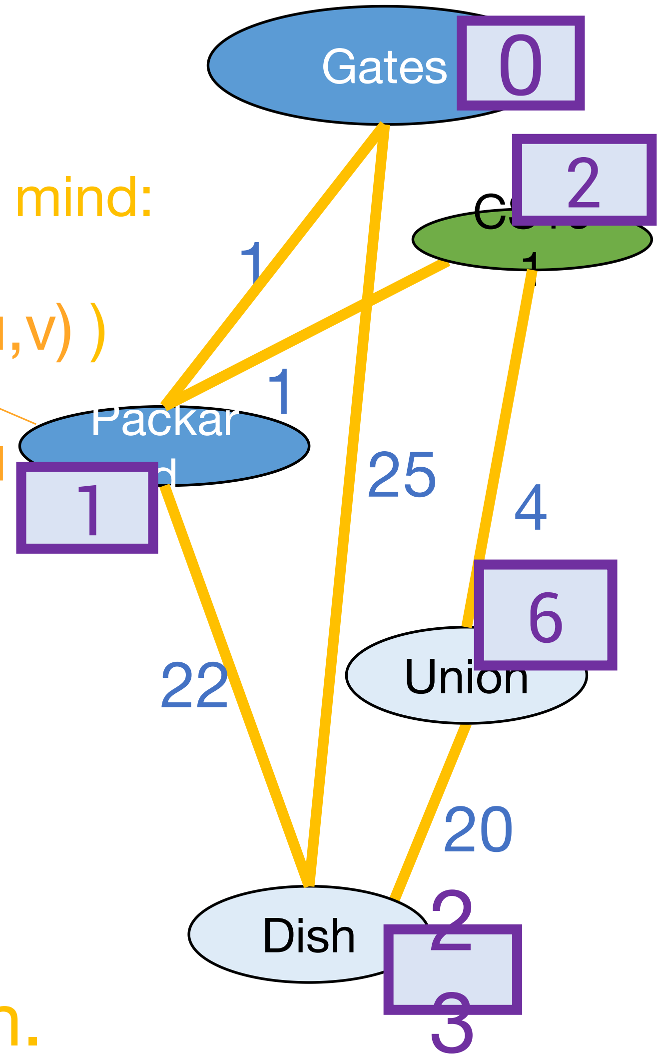
Whatever path we had in mind before

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)

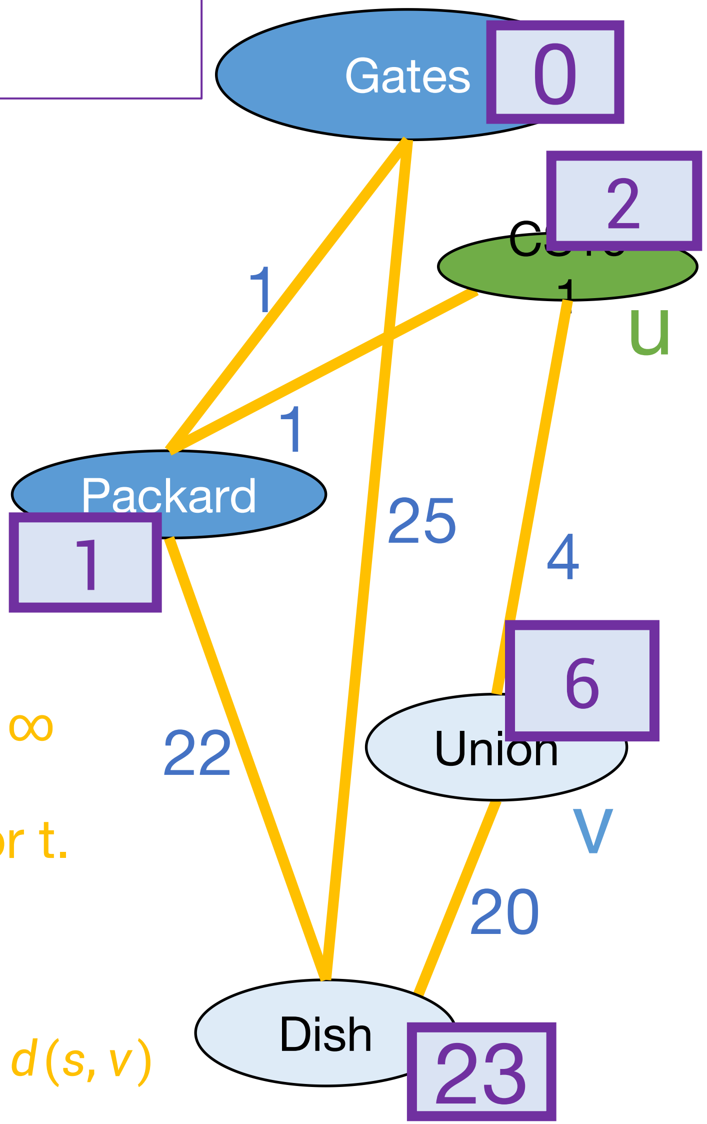
Claim 1

d[v] ≥ d(s,v) for all v.

- Inductive hypothesis.
  - After t iterations of Dijkstra, d[v] ≥ 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] ← 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]

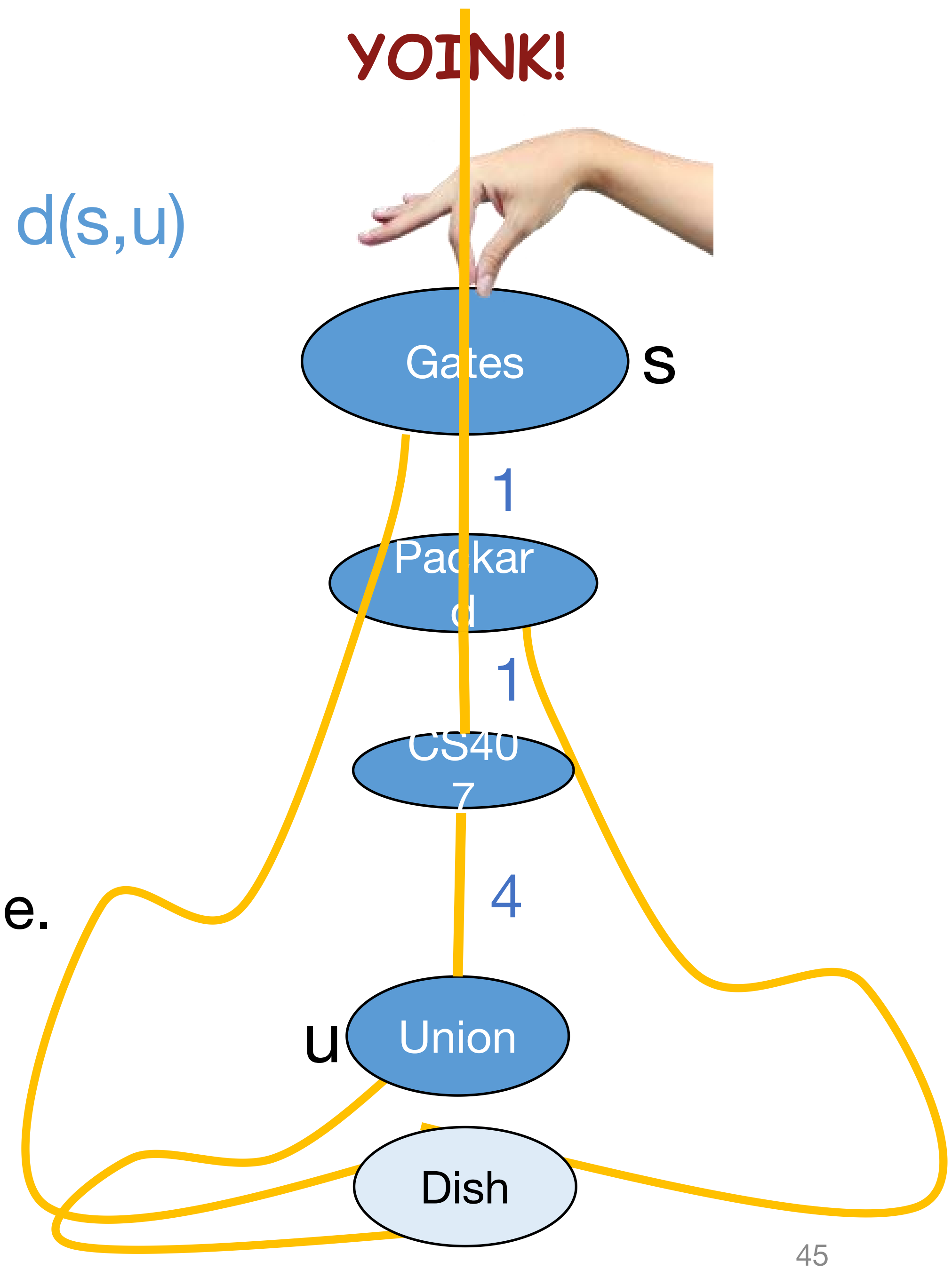


So the inductive hypothesis holds for t+1, and Claim 1 follows.

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:

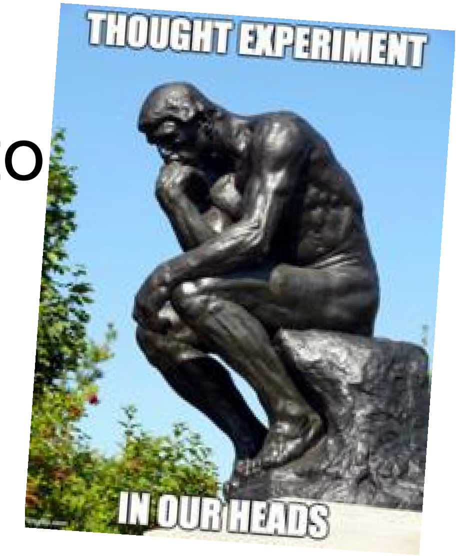
> - Pick the not-sure node u with the smallest estimate d[u].
> - Update all u's neighbors v:
>   - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
> - Mark u as sure.
> - Repeat

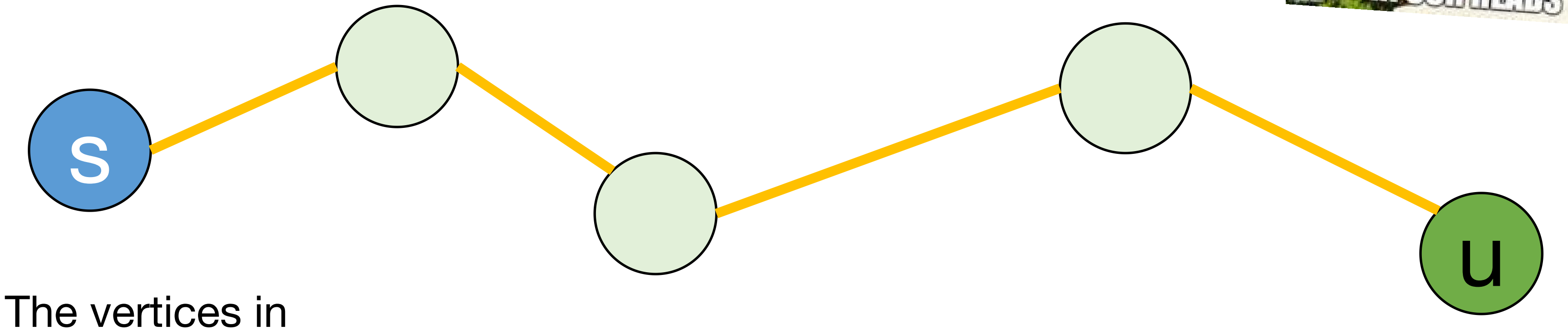
- Want to show that d[u] = d(s,u).

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





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

True shortest path.

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

means good

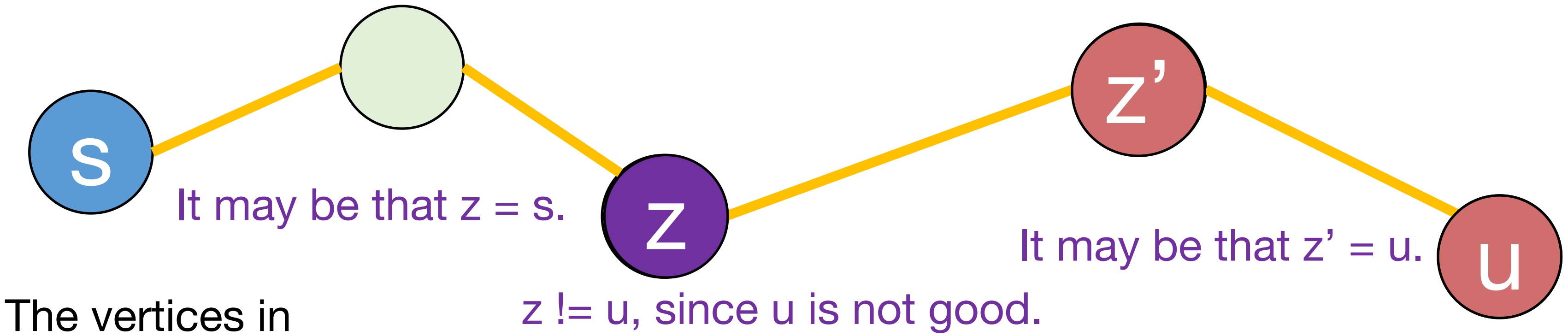
means not good

Claim 2

Inductive step

“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 (on shortest path to 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.

Claim 2

Inductive step

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

means good

means not good

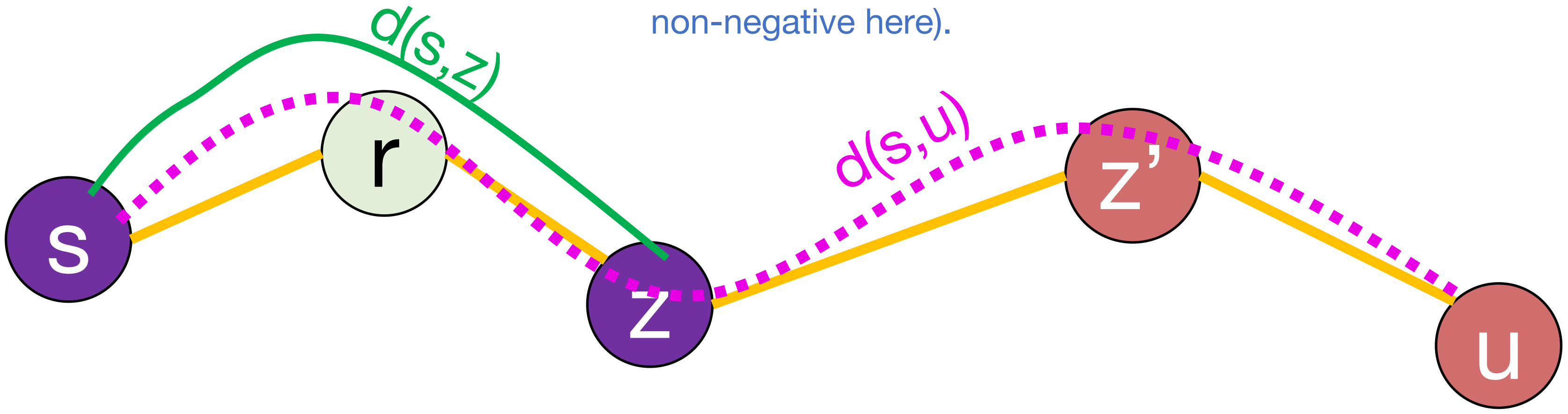
- Want to show that u is good.

BWOC, suppose u isn’t good.

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

z is good

Subpaths of shortest paths are shortest paths.
(We’re also using that the edge weights are non-negative here).



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.

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

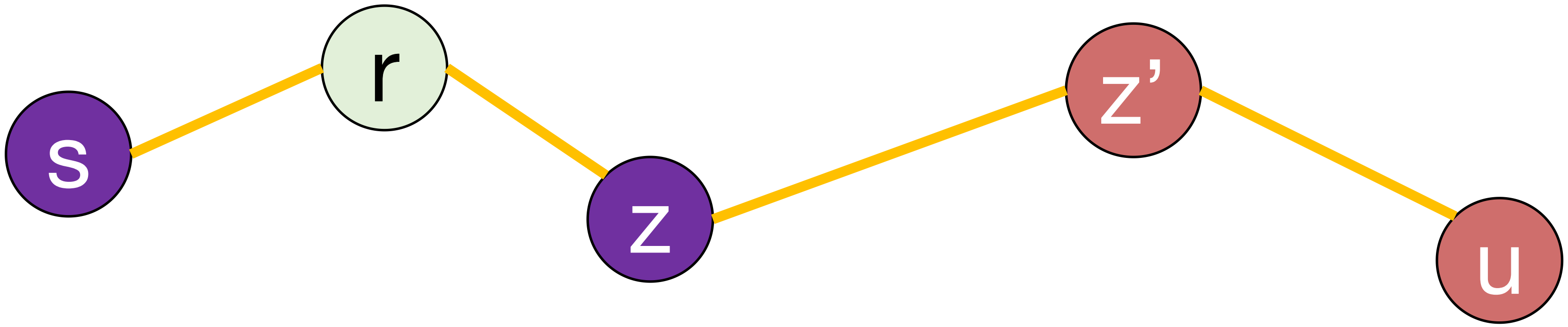
z is good

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.



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.

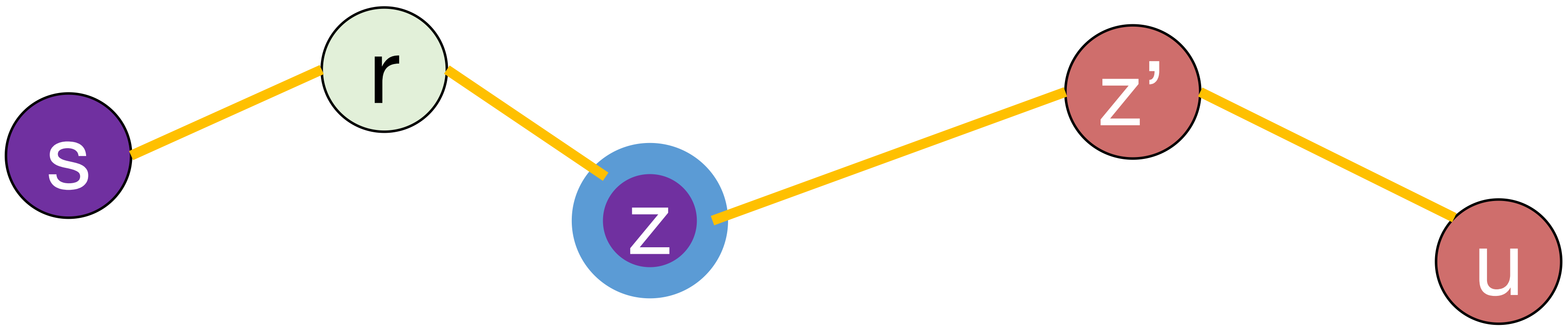
Claim 1

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

But u is not good!

- So $d[z] < d[u]$, so z is sure.

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

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

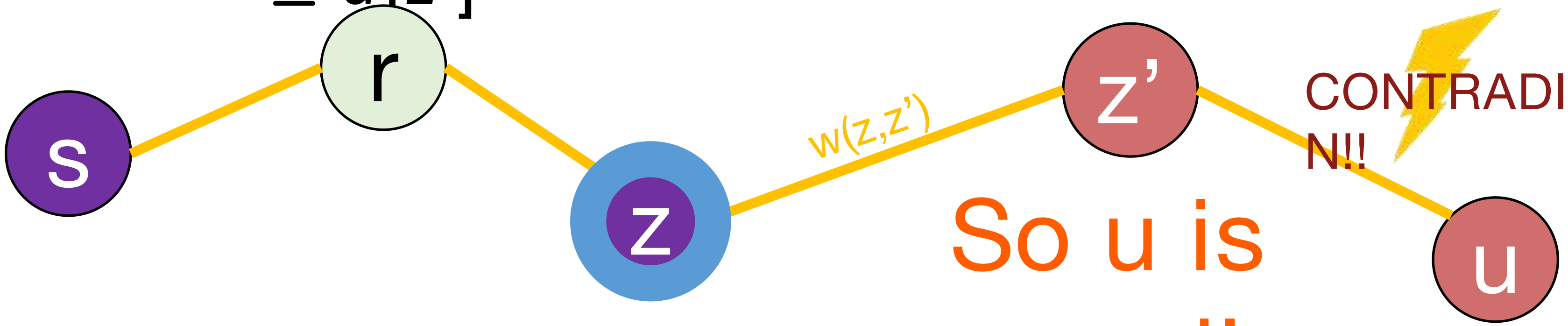
$= d(s,z) + w(z,z')$ By induction when z was added to the sure list it had d(s,z) = d[z]

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.



CONTRADICTION!!

So u is good!

Claim 2

Back to this slide

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).

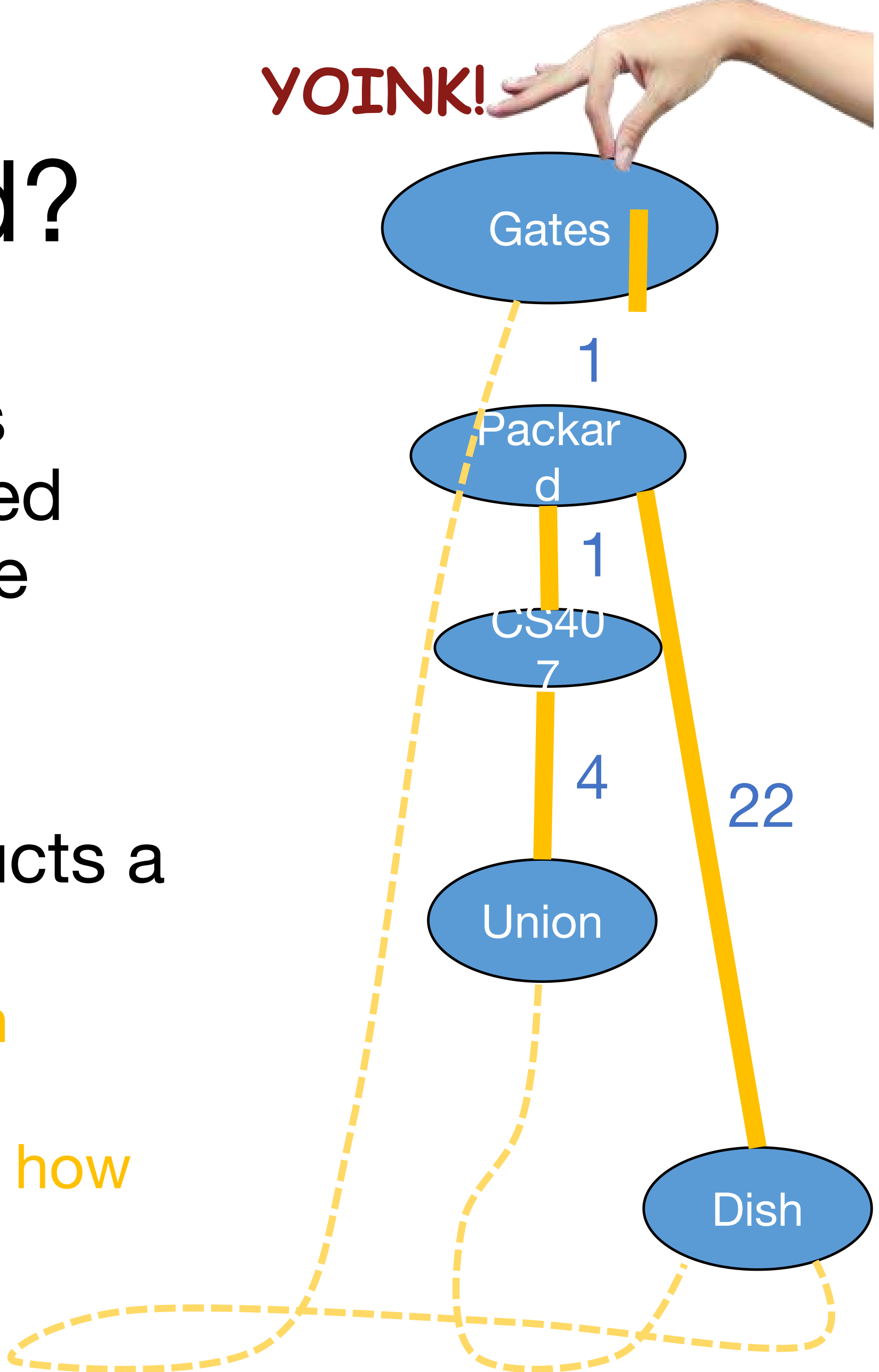
- Proof outline:
  - Claim 1: For all v, d[v] $\geq$ d(s,v).
  - Claim 2: When a vertex is marked sure, d[v] = d(s,v).

- Claims 1 and 2 imply the theorem.

What 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:

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

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

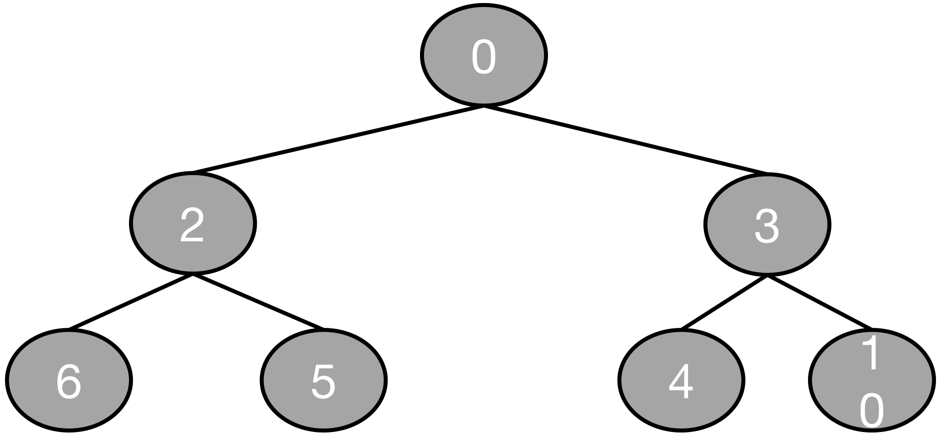
$O(n(\, T(\text{findMin}) + T(\text{removeMin})\,) + m\, T(\text{updateKey}))$

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.

Heaps support these operations

- findMin
- removeMin
- 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
- 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

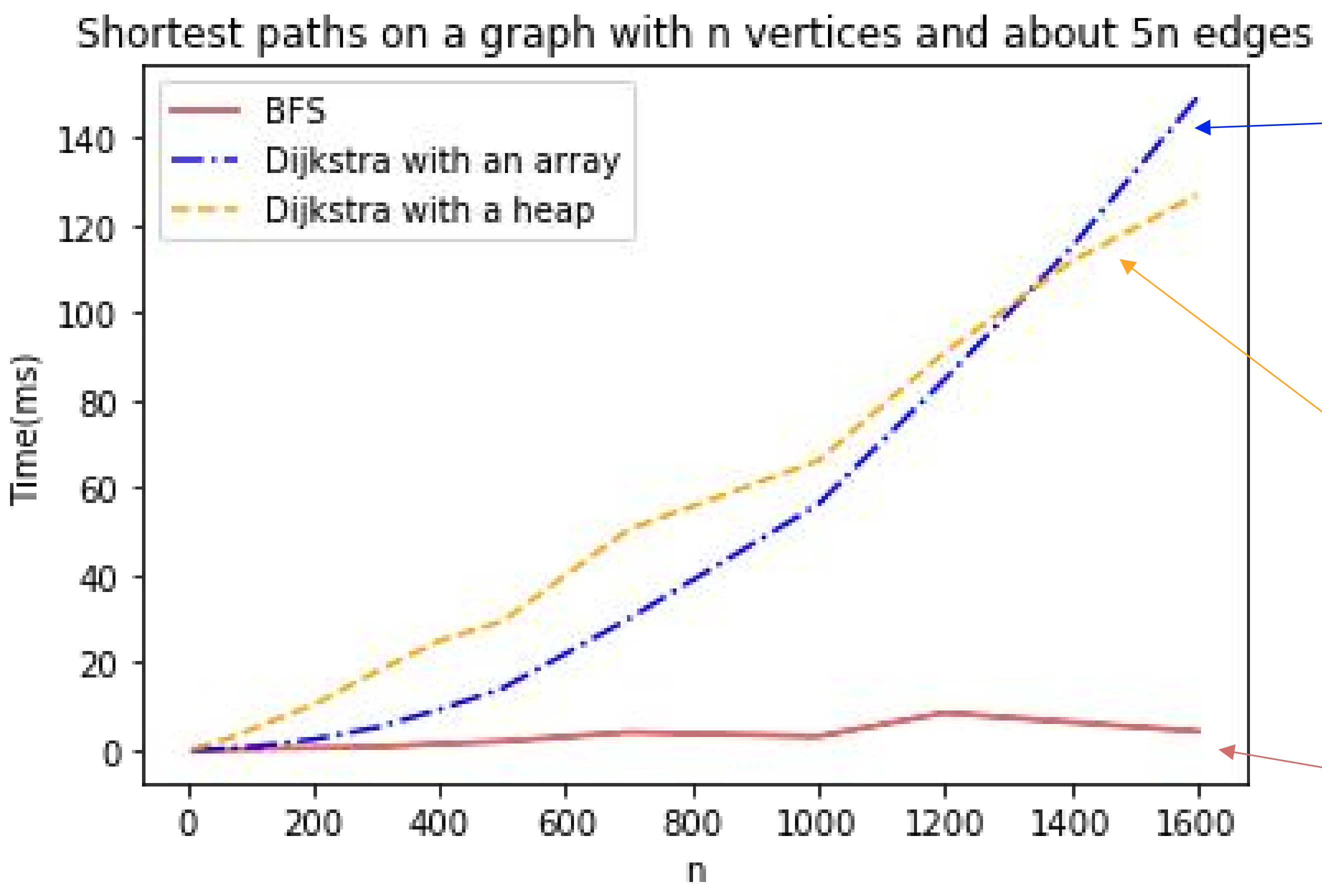
- $T(\text{findMin}) = O(1)$ (amortized time*)
- $T(\text{removeMin}) = O(\log(n))$ (amortized time*)
- $T(\text{updateKey}) = O(1)$ (amortized time*)
- See Data Sturcture for more!
- Running time of Dijkstra

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

$$= O(n\log(n) + m) \quad \text{(amortized time)}$$

*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..

In practice



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

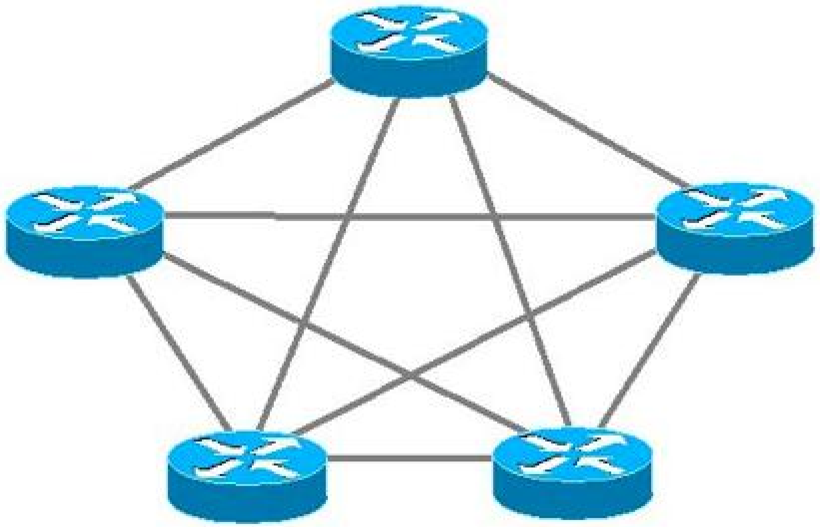
Dijkstra using a heap looks a bit more linear (actually nlog(n))

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.
  - 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

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 lecture

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

Bellman-Ford*(G,s):

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

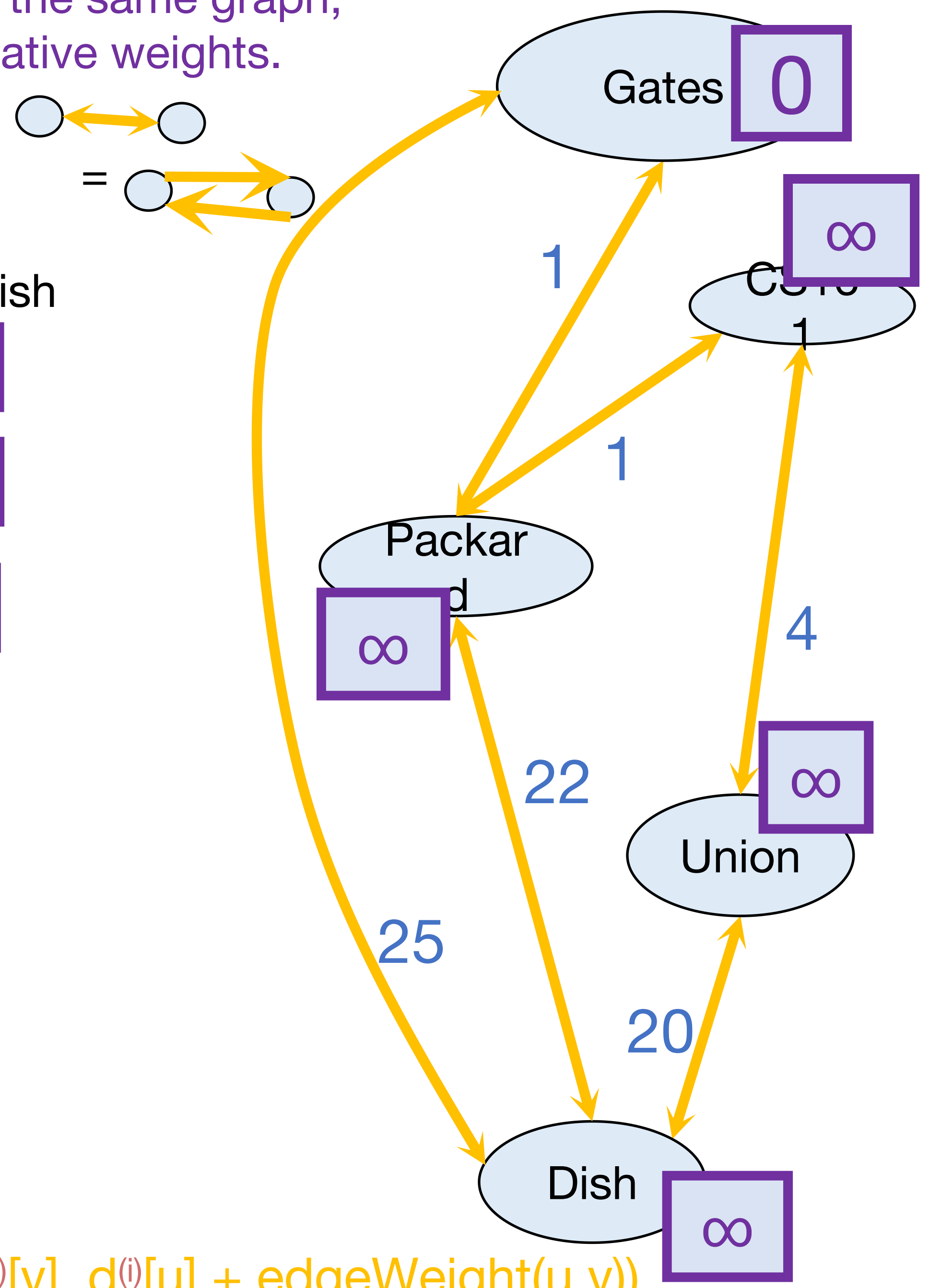
Slightly different than the original Bellman-Ford algorithm, but the analysis is basically the same.

Bellman-Ford

Start with the same graph, no negative weights.

How far is a node from Gates?

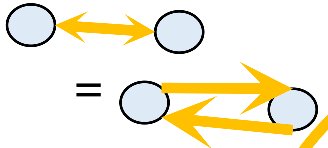
	Gates	Packard	CS407	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$					
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					



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

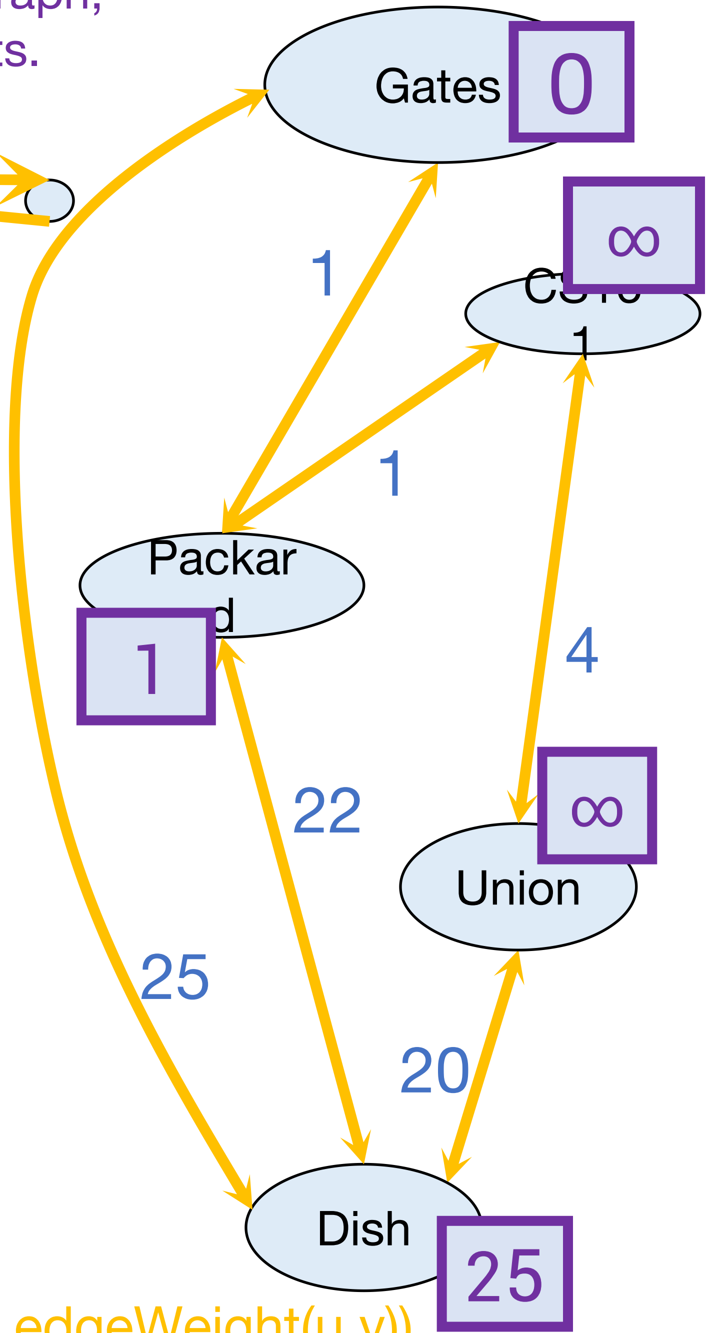
Bellman-Ford

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



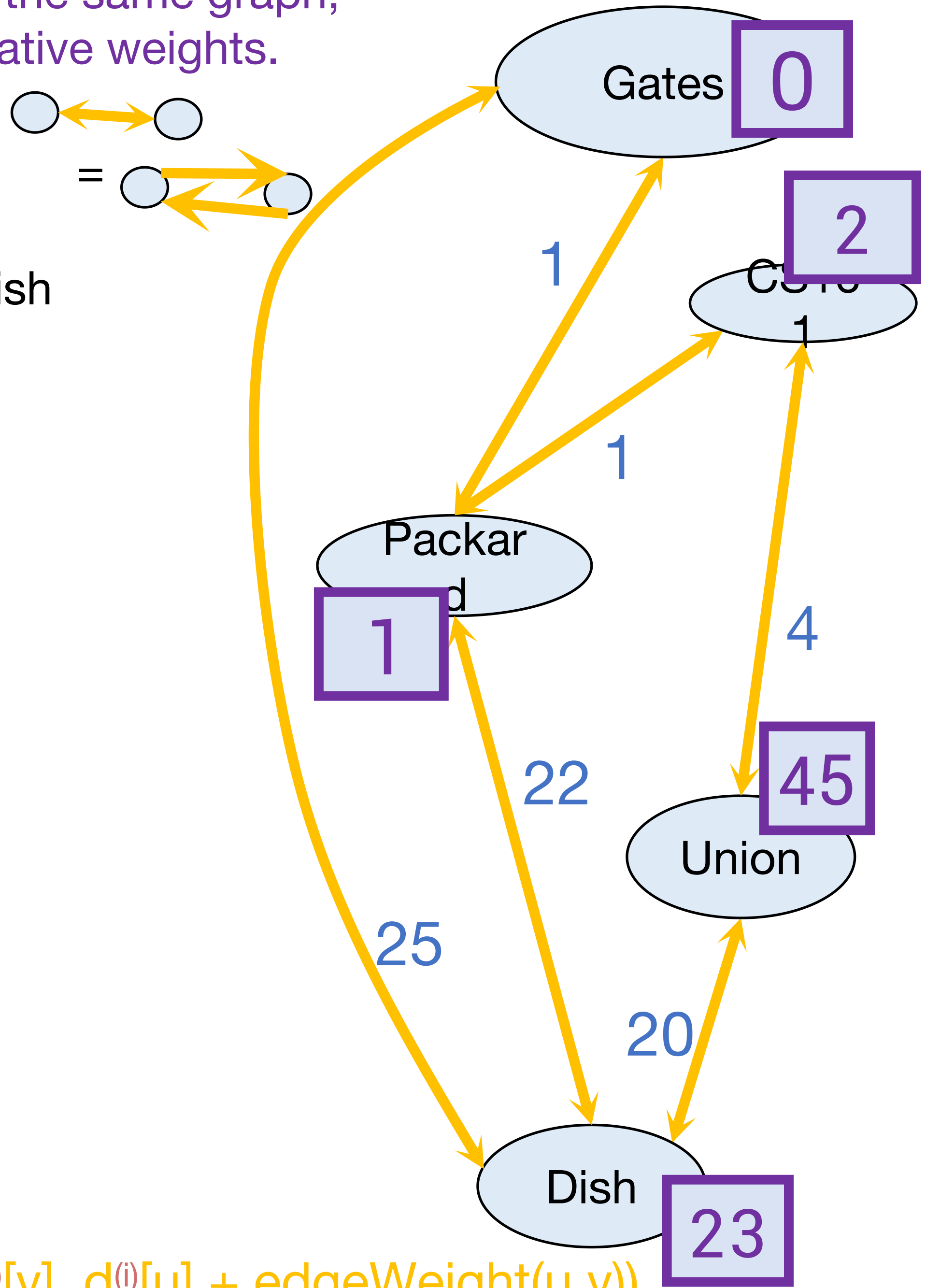
- For i=0,…,n-2:
  - For u in V:
    - For v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

Bellman-Ford

Start with the same graph, no negative weights.

How far is a node from Gates?

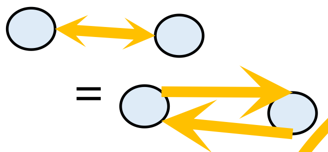
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	1	∞	∞	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$					
$d^{(4)}$					



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

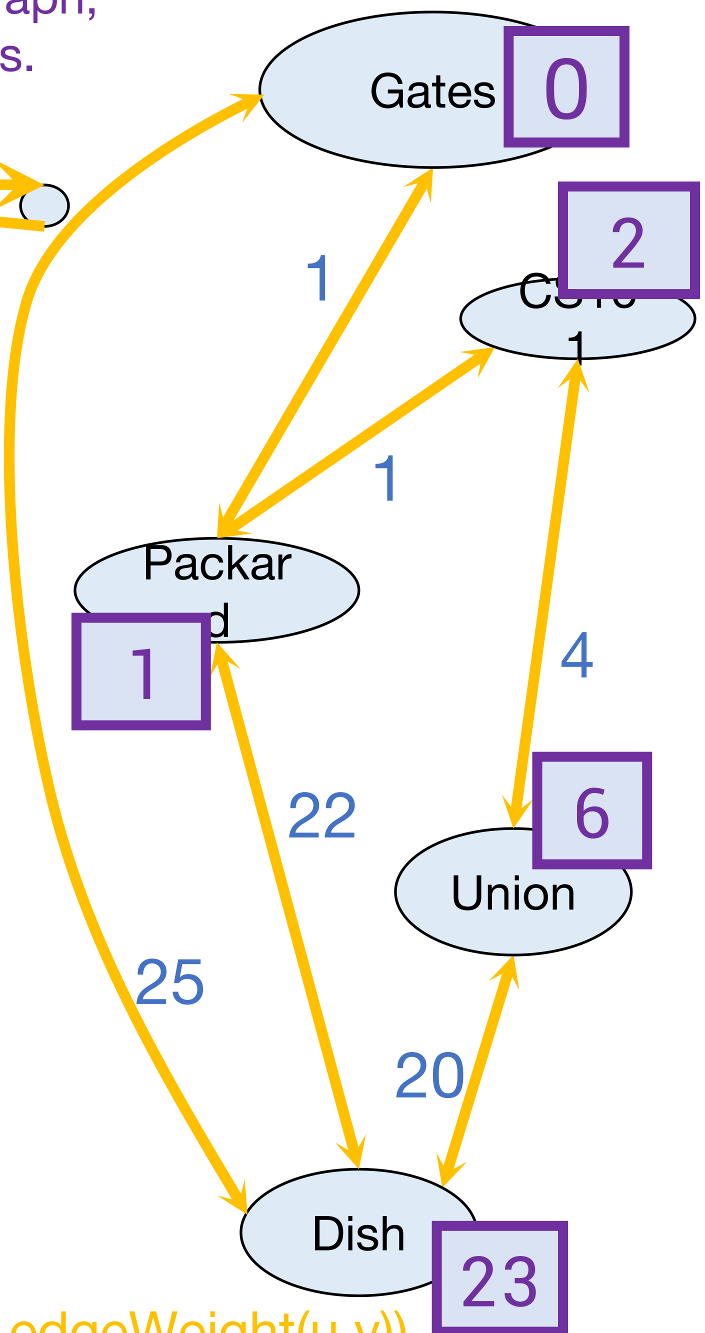
Bellman-Ford

Start with the same graph, no negative weights.



How far is a node from Gates?

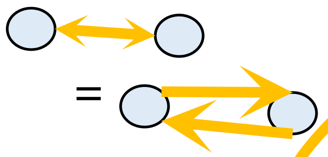
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	1	∞	∞	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$	0	1	2	6	23
$d^{(4)}$					



- 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

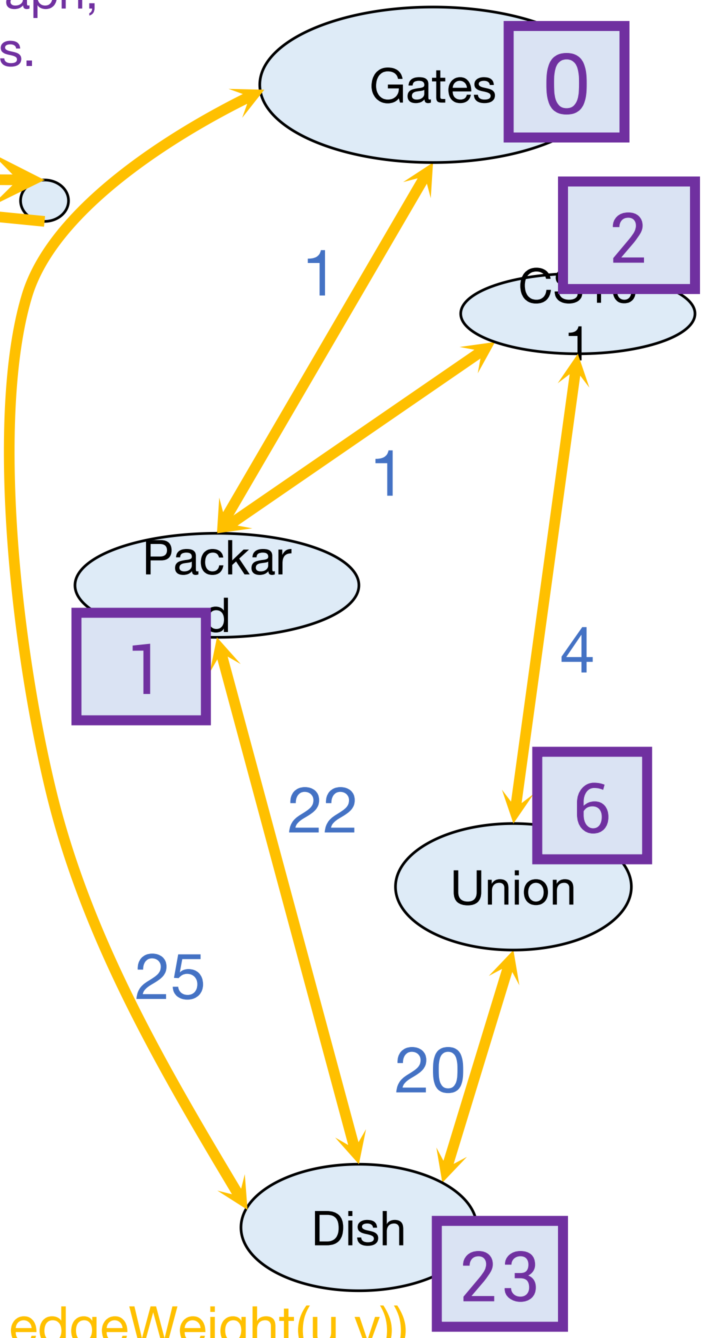
Start with the same graph, no negative weights.



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)

- Is it fast?
  - Not really…

A simple path is a path with no cycles.

	Gates	Packard	CS407	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).

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:

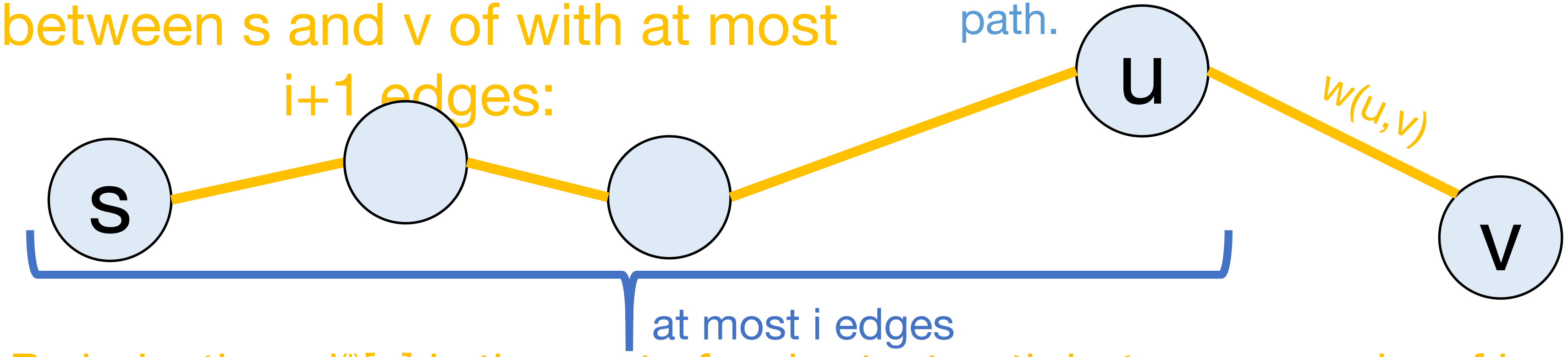
Inductive step

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 $d^{(i+1)}[v] <= d^{(i)}[u] + w(u,v)$.
- 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.

Proof by induction

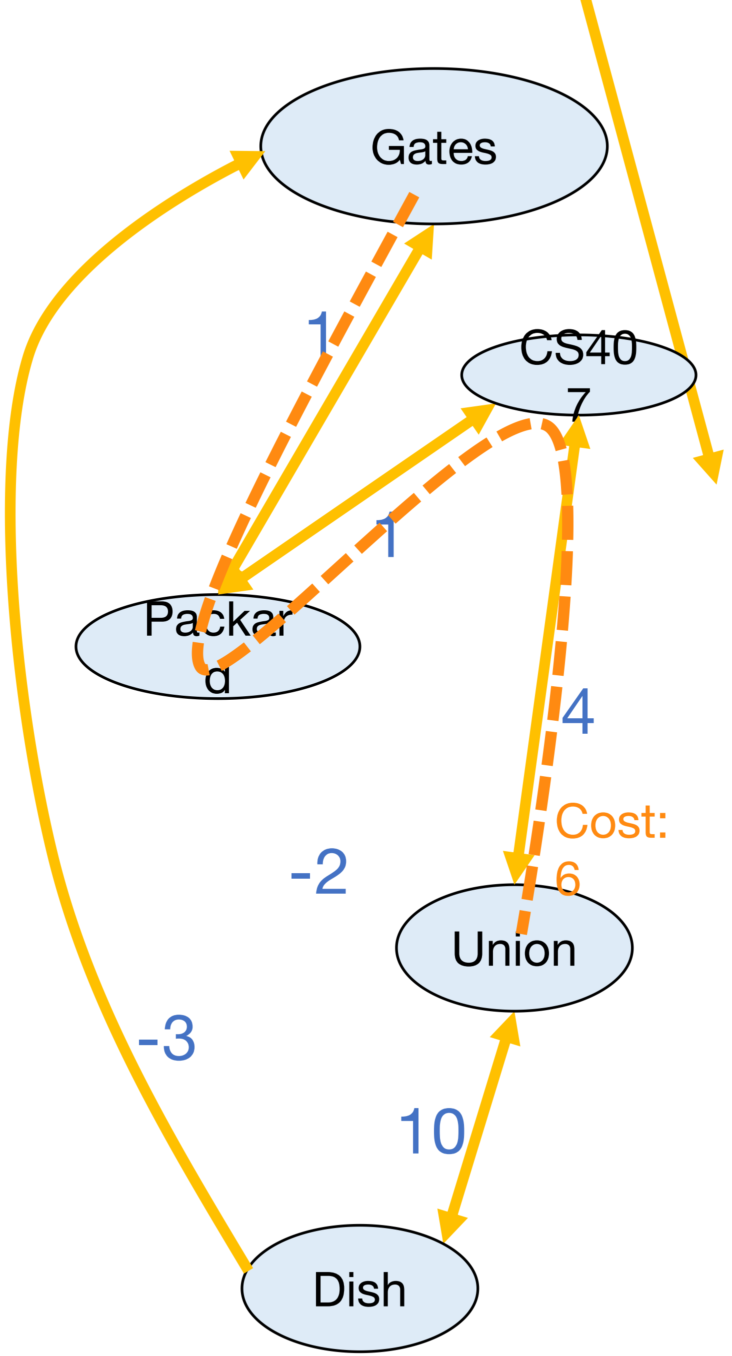
- Inductive Hypothesis:
  - After iteration i, for each v, $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v of length at most i edges.
- Base case:
  - After iteration 0…
- Inductive step:
- Conclusion:
  - After iteration n-1, for each v, d[v] is equal to the cost of the shortest 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!

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.

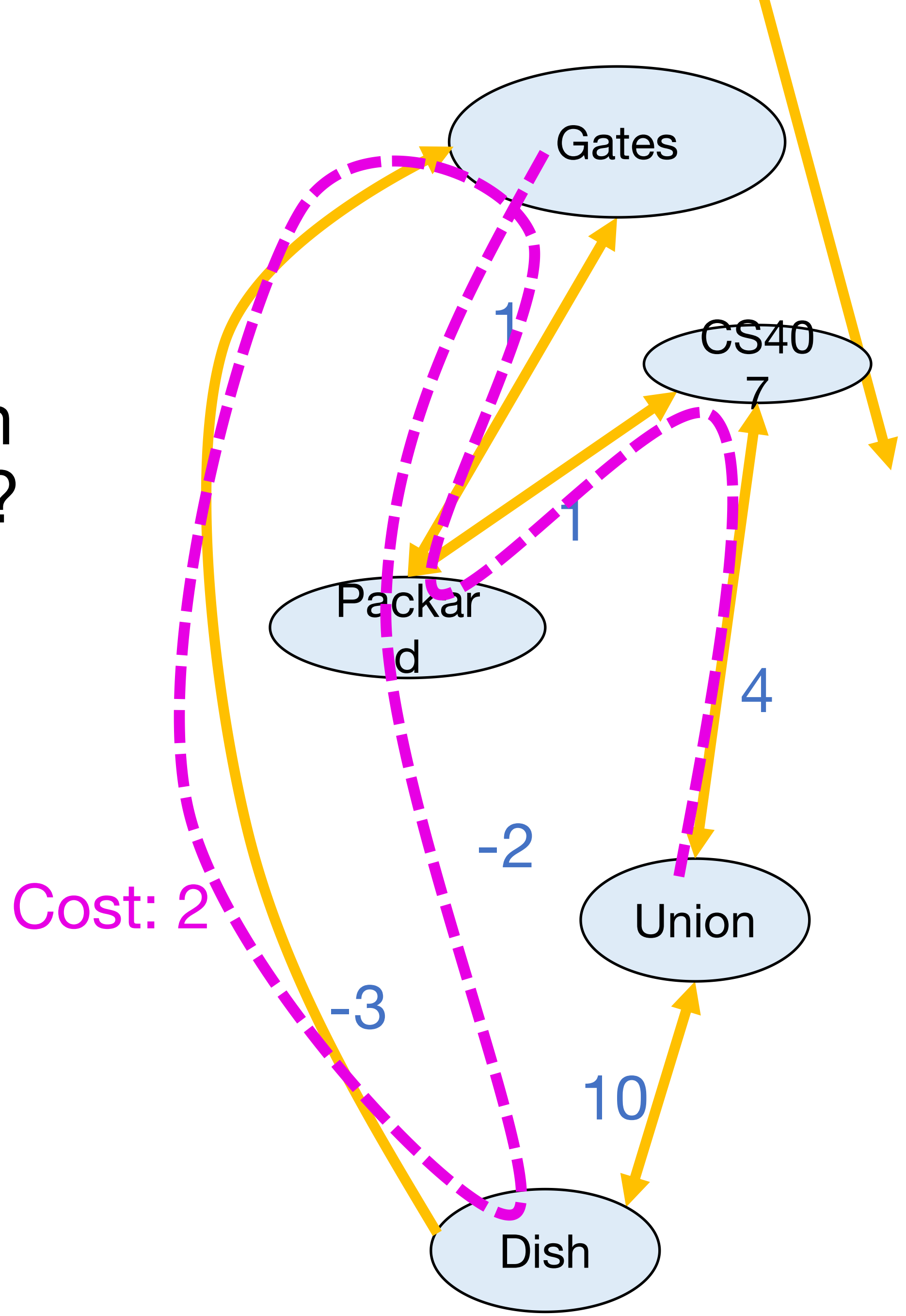
Wait a second…

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



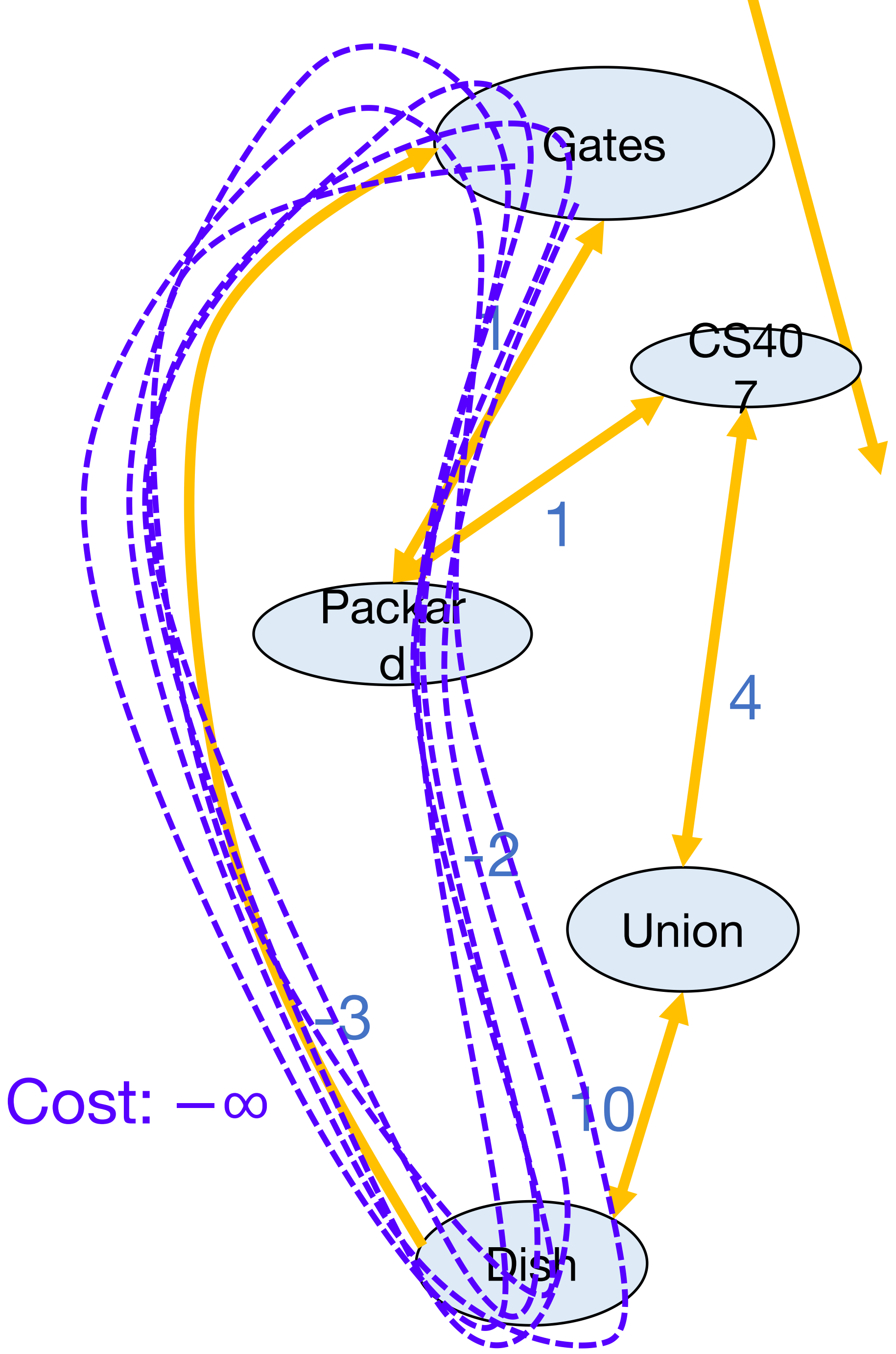
Wait a second…

- 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 no 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.

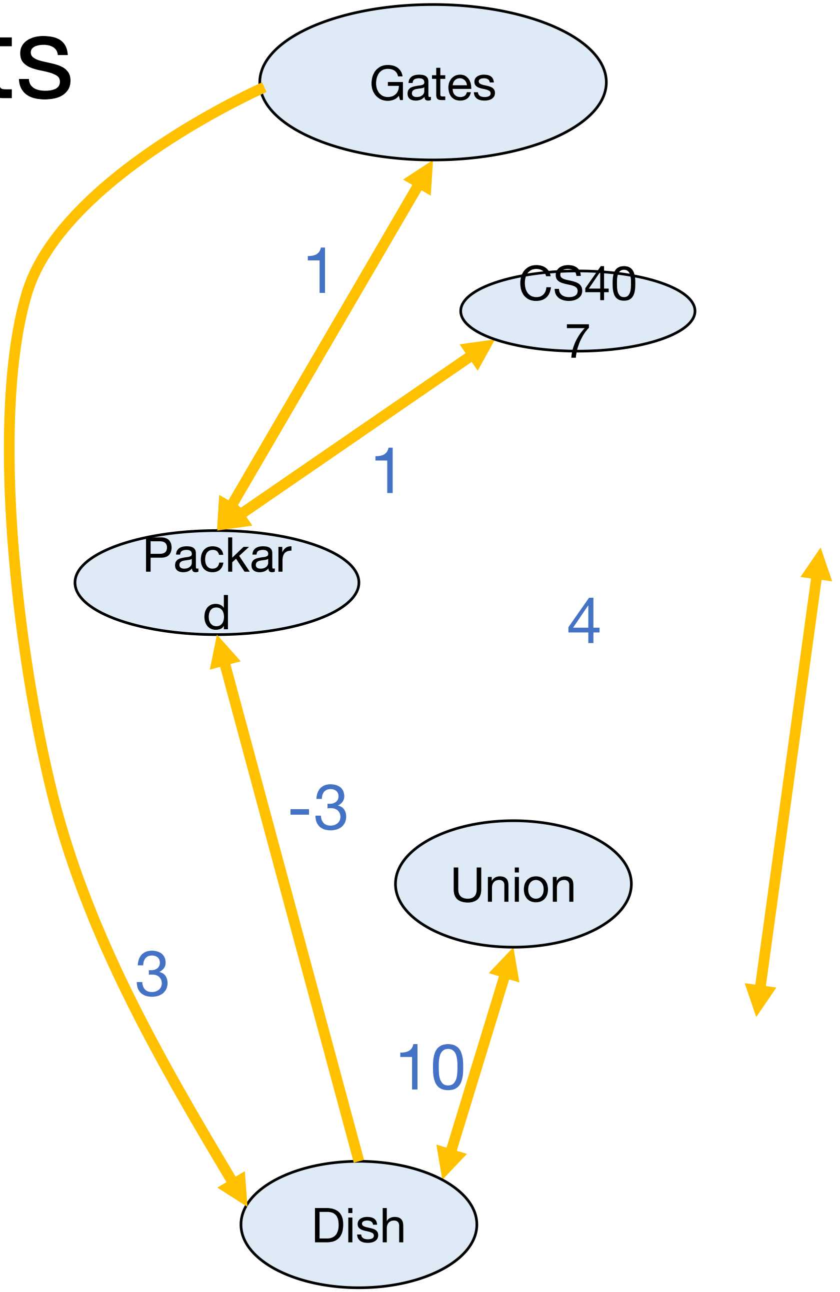
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	1	∞	∞	25
$d^{(2)}$	0	1	2	45	23
$d^{(3)}$	0	1	2	6	23
$d^{(4)}$	0	1	2	6	23

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).

If there are negative cycles, then non-simple paths matter!
So the proof breaks for negative cycles.

Negative edge weights

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	1	∞	∞	3
$d^{(2)}$	0	0	2	13	3
$d^{(3)}$	0	0	1	6	3
$d^{(4)}$	0	0	1	5	3

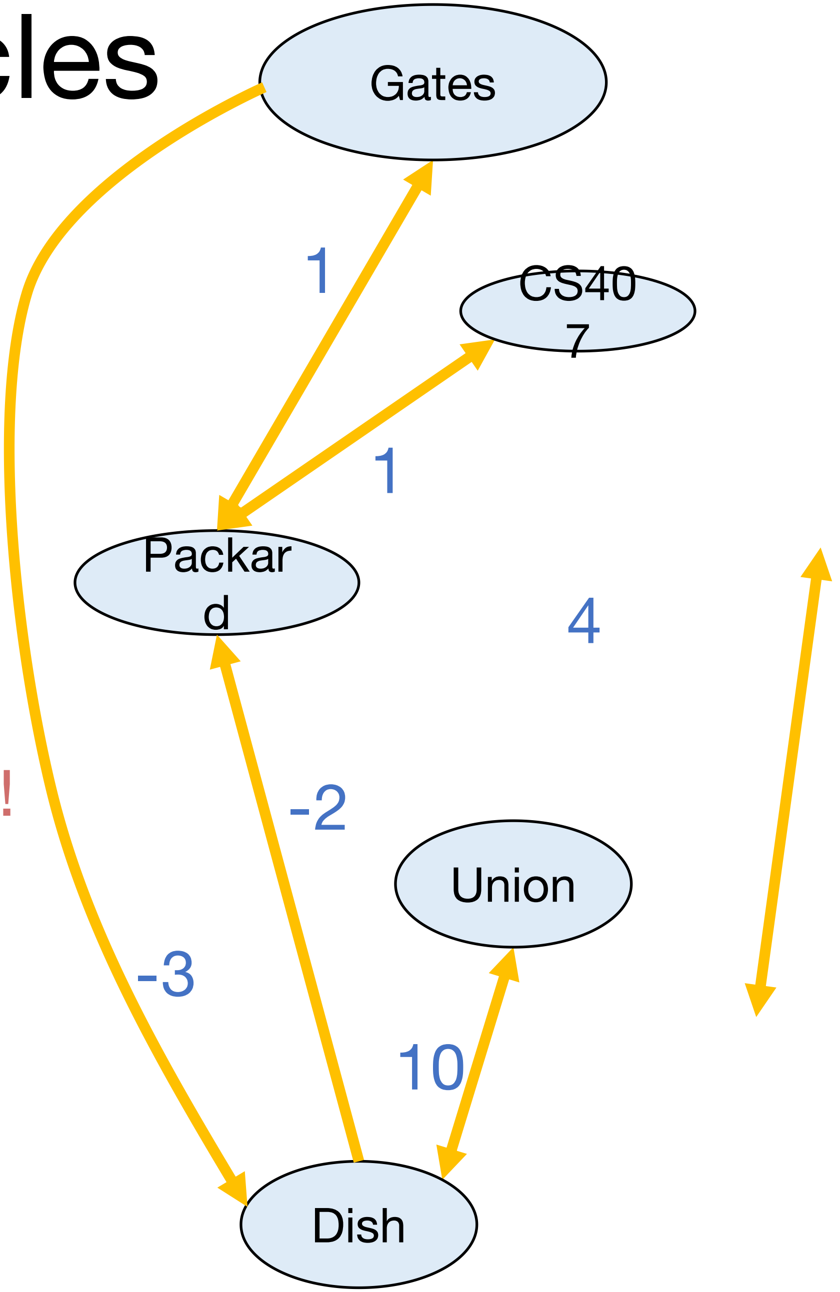


- For i=0,…,n-2:
    - For u in V:
        - For v in u.neighbors:
            - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + \text{edgeWeight}(u,v))$

B-F with negative cycles

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

B-F with negative cycles

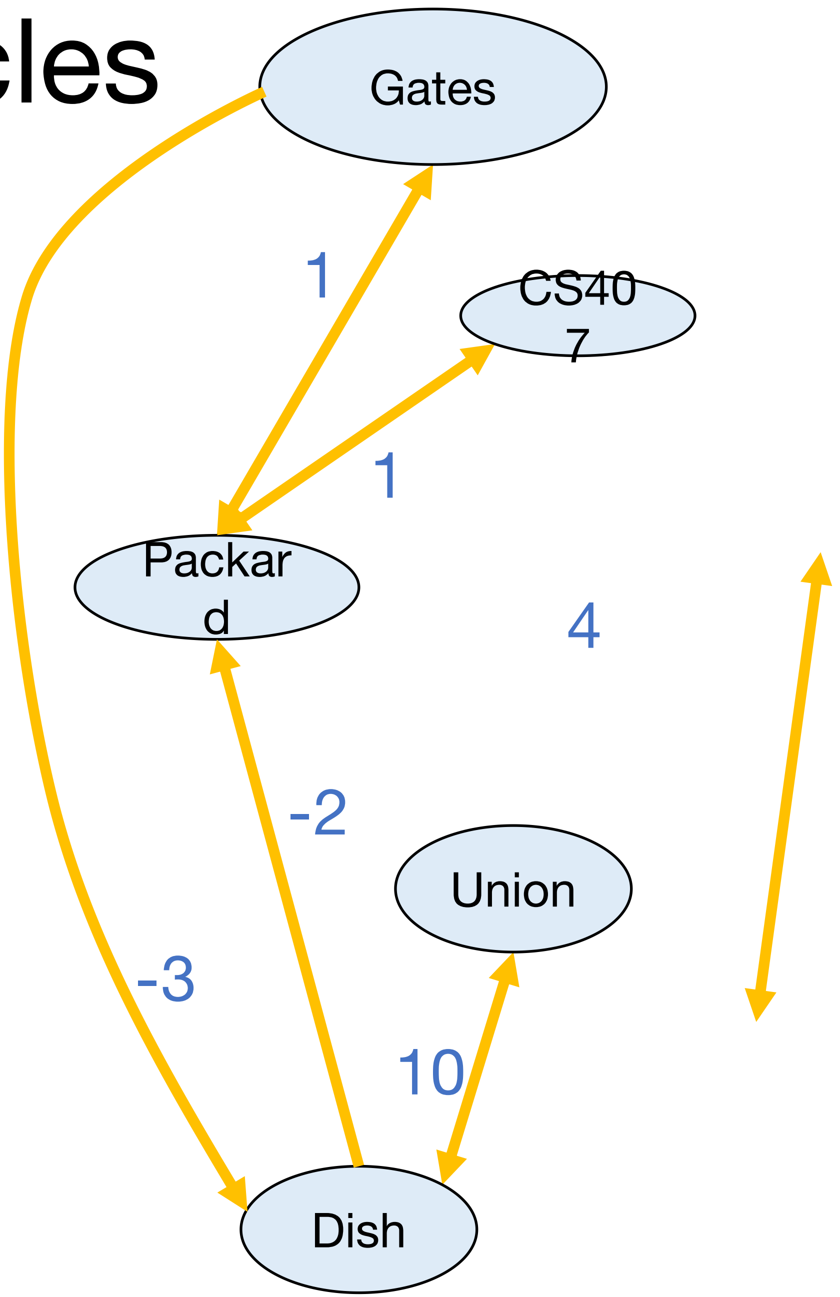
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	1	∞	∞	-3
$d^{(2)}$	0	-5	2	7	-3
$d^{(3)}$	-4	-5	-4	6	-3
$d^{(4)}$	-4	-5	-4	6	-7

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



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…
    - 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

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:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v]\ ,\ d^{(i+1)}[v],\ d^{(i)}[u] + \text{edgeWeight}(u,v))$
- If $d^{(n-1)} \mathrel{!}= d^{(n)}$:
  - Return NEGATIVE CYCLE ☹
- Otherwise, dist(s,v) = $d^{(n-1)}[v]$

Summary

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

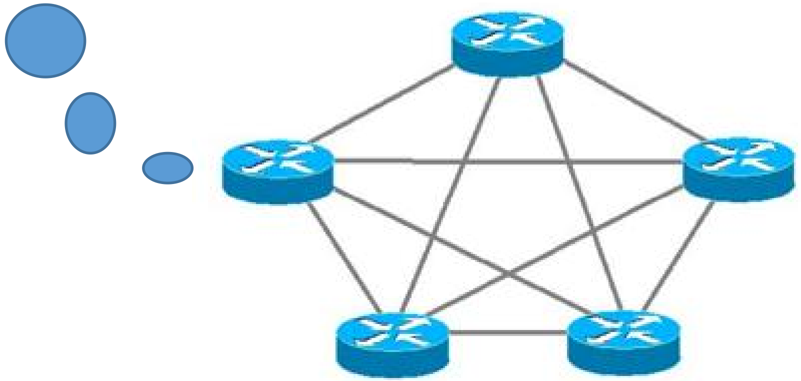
- The Bellman-Ford algorithm:
  - Finds shortest paths in weighted graphs with negative edge weights
  - runs in time O(nm) on a graph G with n vertices and m edges.
- If there are no negative cycles in G:
  - the BF algorithm terminates with $d^{(n-1)}[v] = d(s,v)$.
- If there are negative cycles in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

Next Time

- Dynamic Programming!!!