## Lecture 11

Weighted Graphs: Dijkstra and (intro to) Bellman-Ford

### Announcements

- Midterm a week from today!
  - Thursday 11/6
  - Covers material through today
    - **Dijkstra's Algorithm**; not Bellman-Ford, which we'll only get to quickly at the end, if at all.
  - Keep an eye out for logistics post
  - (See announcements from last time for more...)

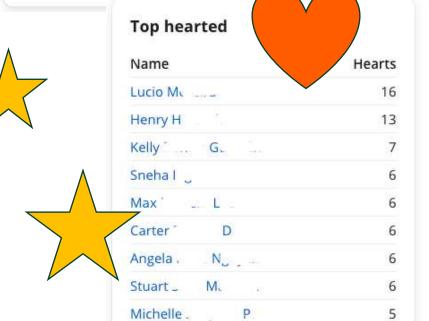
No class Tuesday 11/4: Democracy Day!

## Ed Heroes!

### Top answerers

| Name       | Answers |
|------------|---------|
| Henry H.   | 11      |
| Lucio M    | 9       |
| Kelly I G  | 4       |
| Sneha Ir   | 3       |
| Peter , C. | 3       |
| Samantha L | 1       |
| Chloe T    | 1       |
| Britney B  | 1       |
| Marta V    | 1       |
| Ethan H.   | 1       |

#### Top askers Name Questions Lucio M\_ 15 Zeynep Y 13 Sneha li 11 Ellie L 11 Annabelle 5 11 Cristofer . A 10 Sayuri ' 9 Y Samantha L 8 Henry H 8 Jonathan G 8



5

June Z

### Also

It's that time in the quarter...

...take care of yourselves!

Mental health resources on campus:

• List of resources via student affairs: studentaffairs.stanford.edu/mhrs

Resources at Vaden:

medicalservices.stanford.edu/medical-services-resources/mental-health

### Previous two lectures

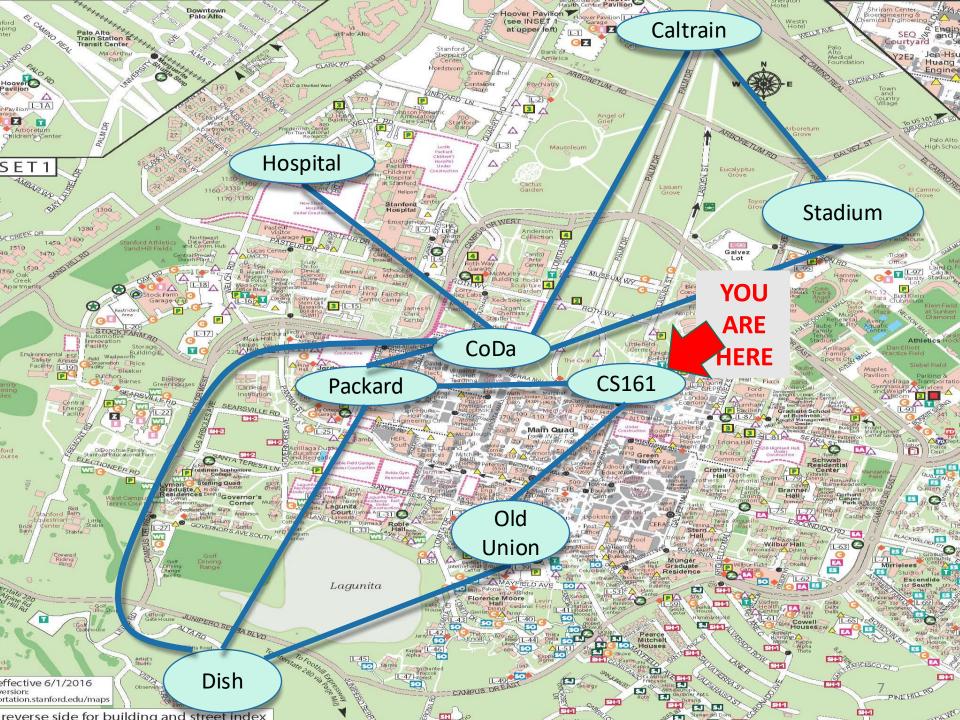
- Graphs!
- DFS
  - Topological Sorting
  - Strongly Connected Components
- BFS
  - Shortest Paths in unweighted graphs

### Today

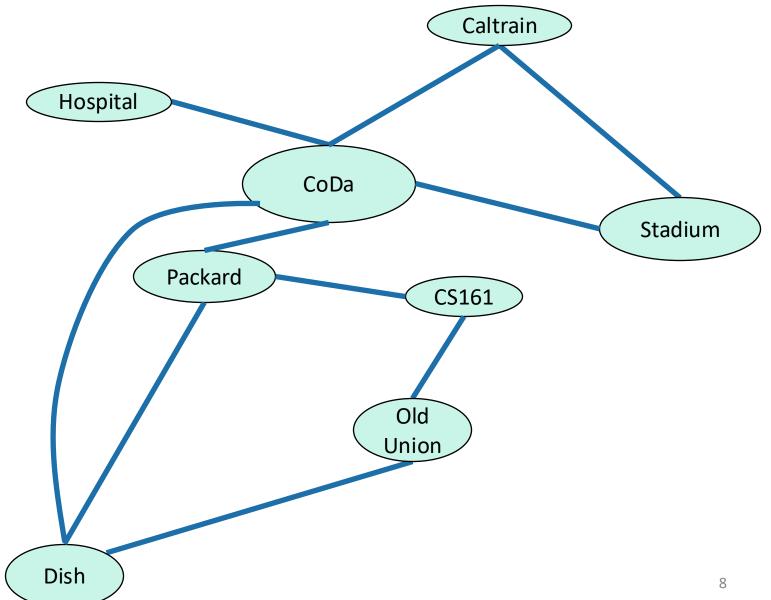
What if the graphs are weighted?



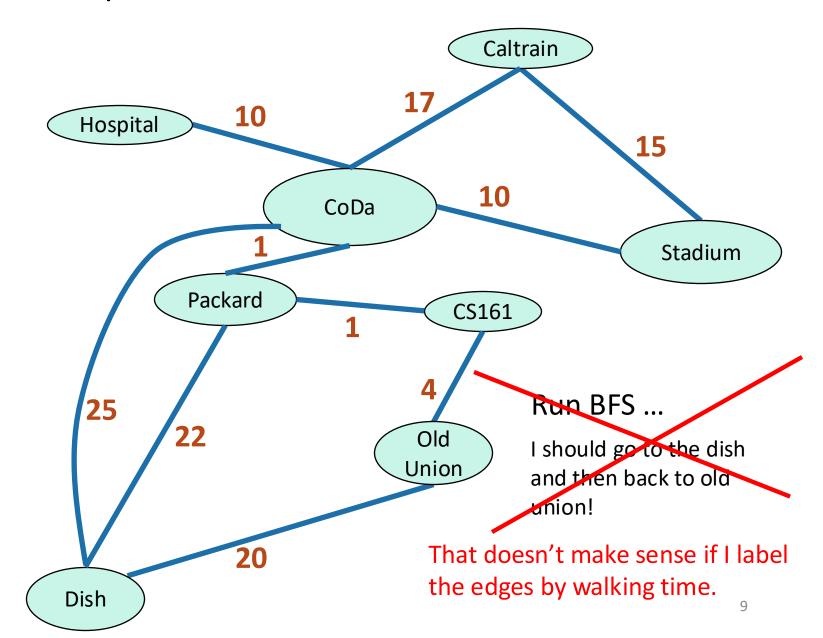
- Part 1: Dijkstra!
  - This will take most of today's class
- Part 2: Bellman-Ford!
  - Real quick at the end if we have time!
  - We'll come back to Bellman-Ford in more detail, so today is just a taste, if we get to it at all.



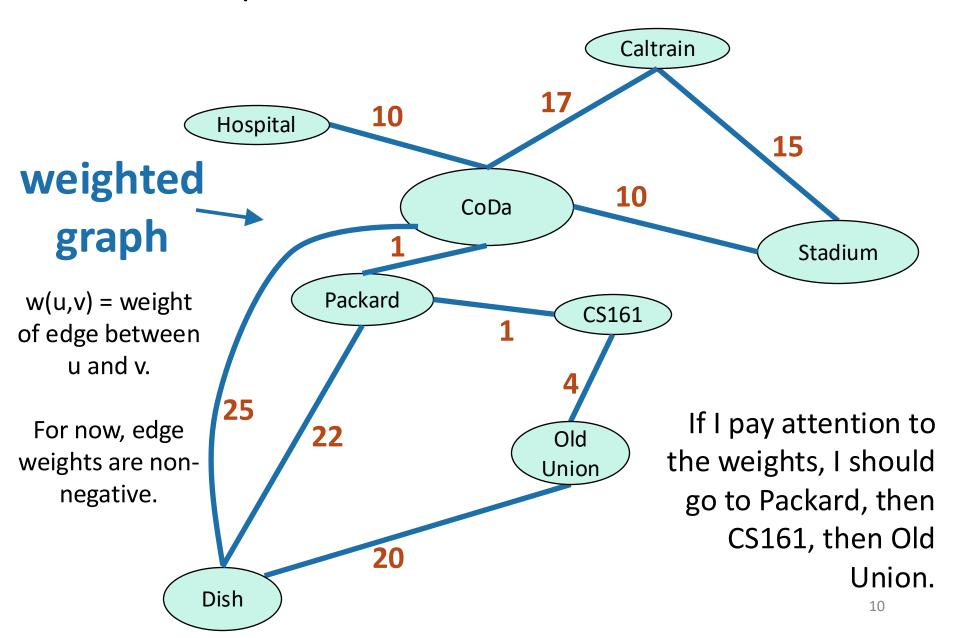
## Just the graph



### Shortest path from CoDa to Old Union?

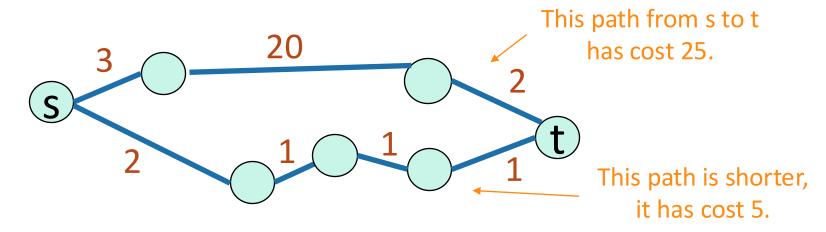


### Shortest path from CoDa to Old Union?

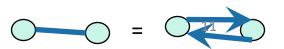


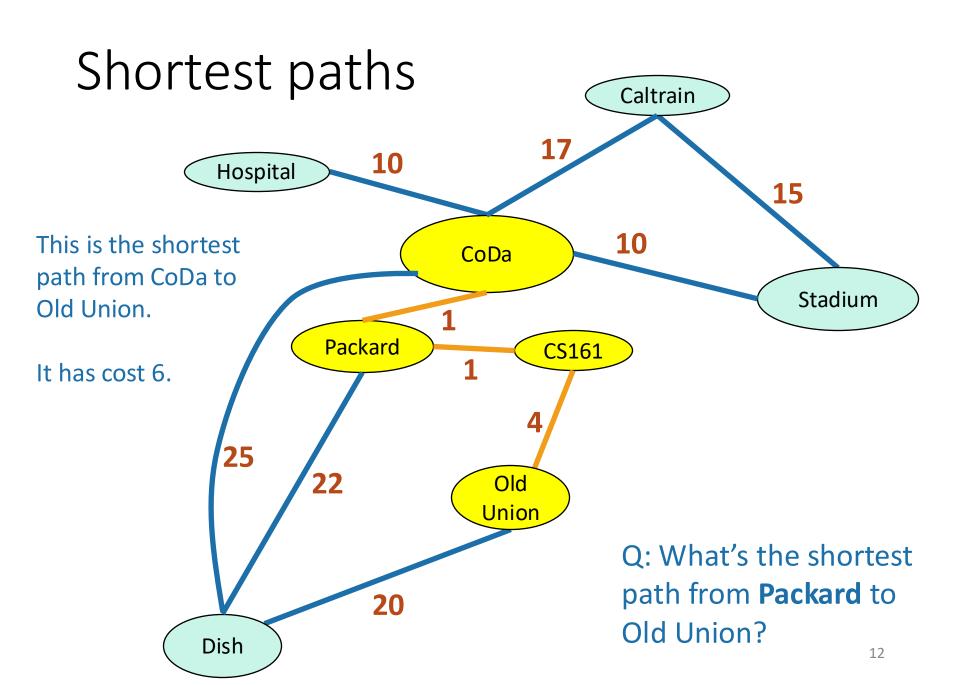
### Shortest path problem

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



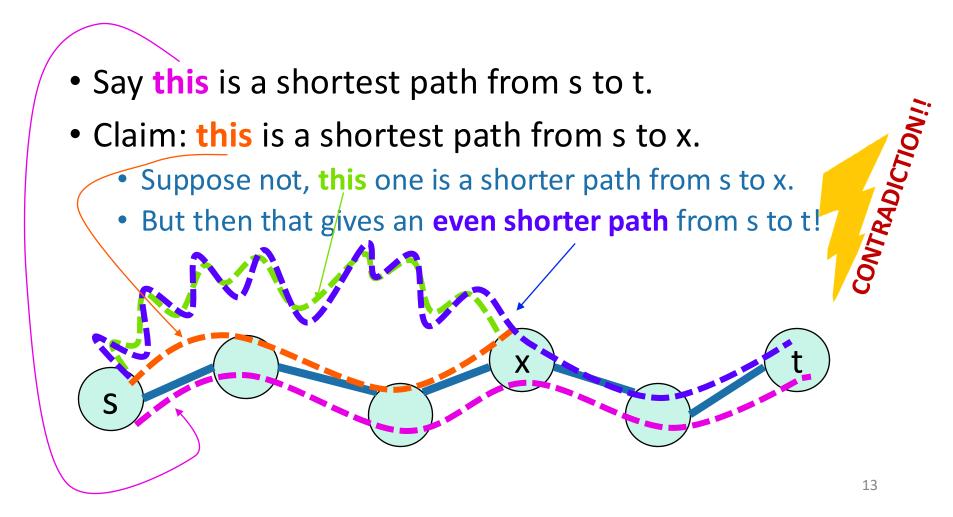
• The **distance** d(u,v) between two vertices u and v is the cost of the the shortest path between u and v.





### Warm-up

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



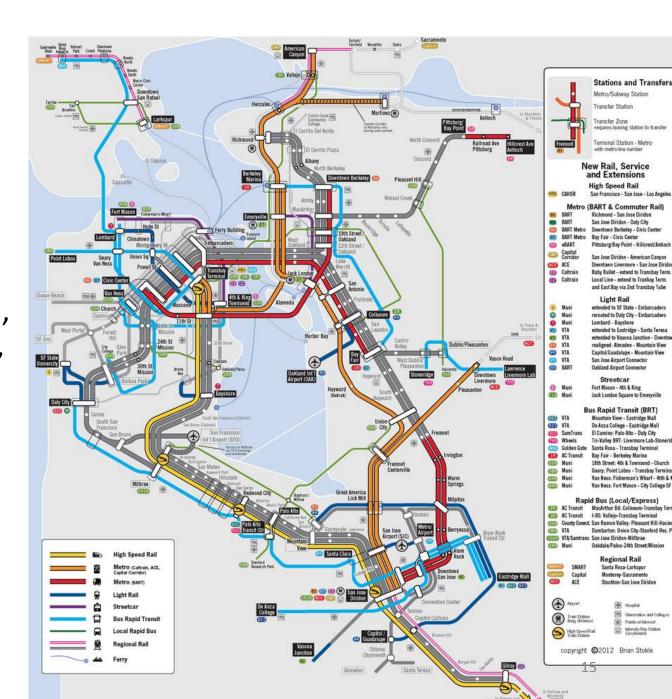
### Single-source shortest-path problem

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

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

### Example

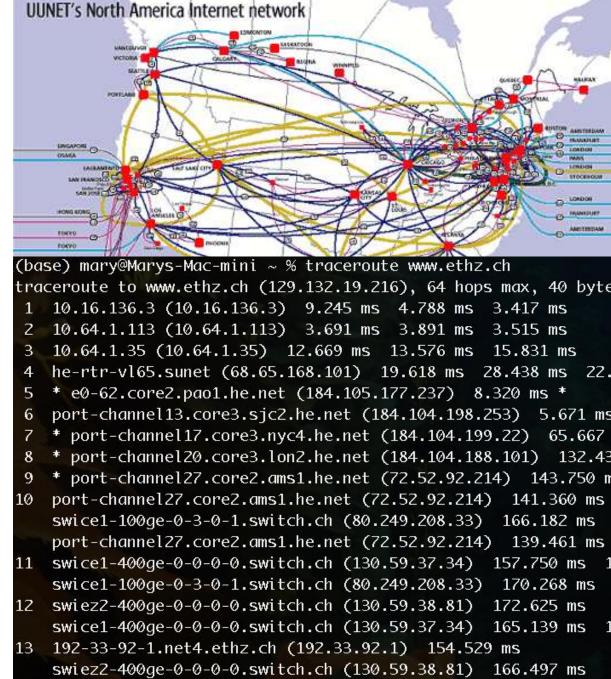
- what is the shortest path from Palo Alto to [anywhere else]" using BART, Caltrain, lightrail, MUNI, bus, Amtrak, bike, walking, uber/lyft.
- Edge weights have something to do with time, money, hassle.



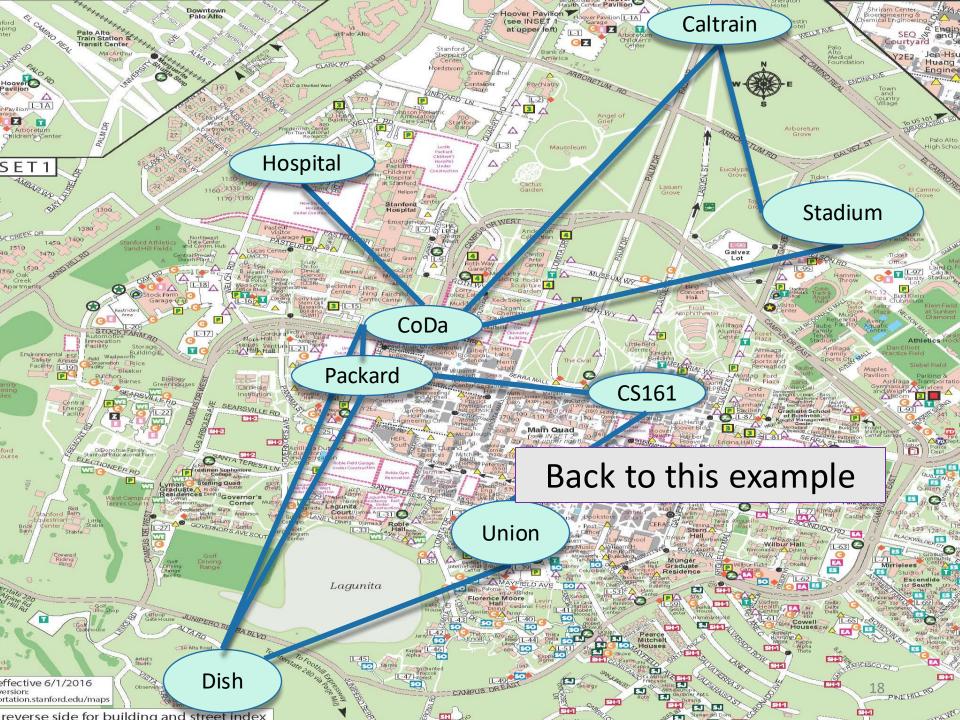
### Example

### 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?

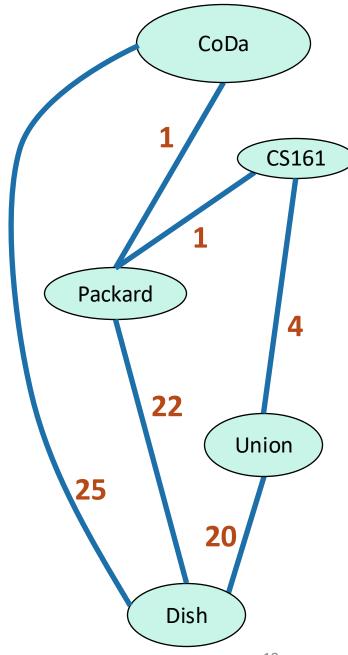


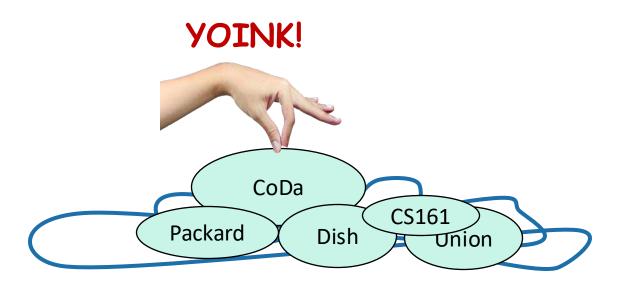
192-33-92-1.net4.ethz.ch (192.33.92.1) 157.398 ms



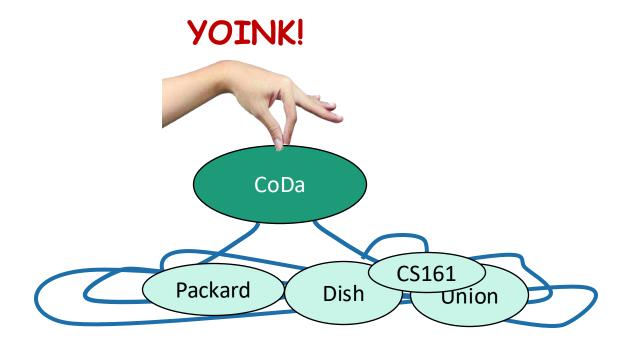
## Dijkstra's algorithm

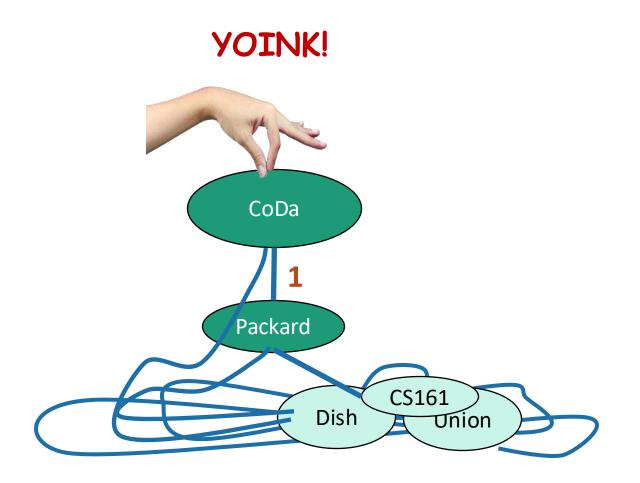
• Finds shortest paths from CoDa to everywhere else.



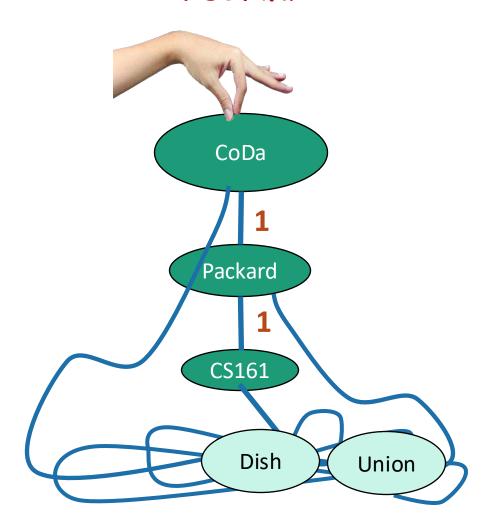


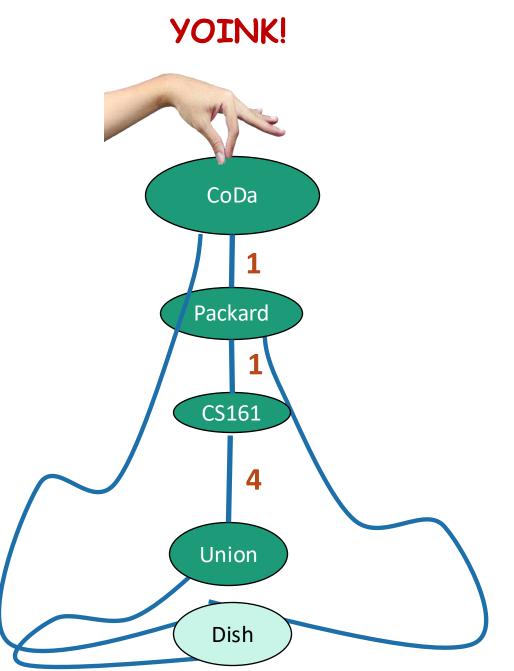
A vertex is done when it's not on the ground anymore.

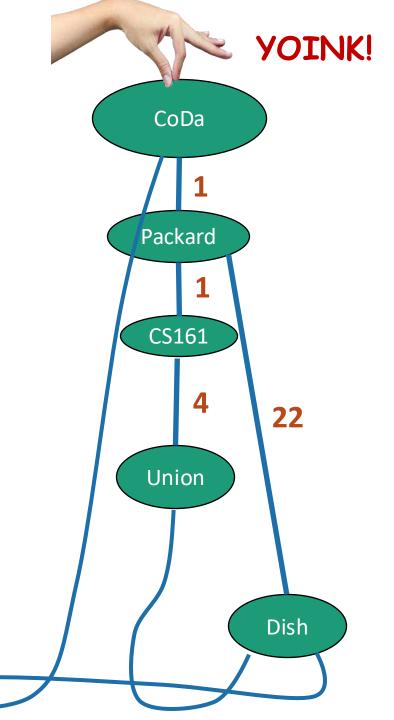




### YOINK!

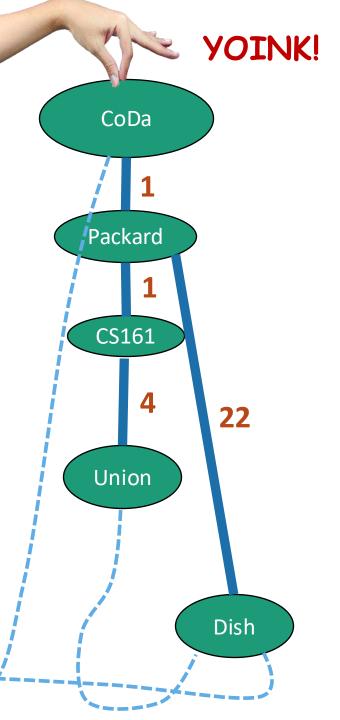






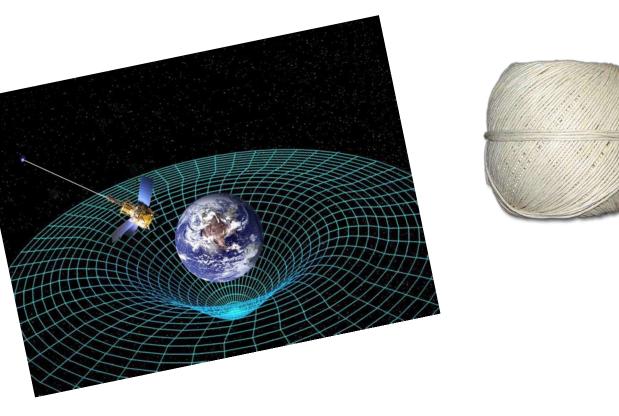
This creates a tree!

The shortest paths are the lengths along this tree.



### How do we actually implement this?

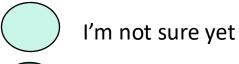
Without string and gravity?

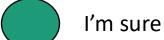






### How far is a node from CoDa?



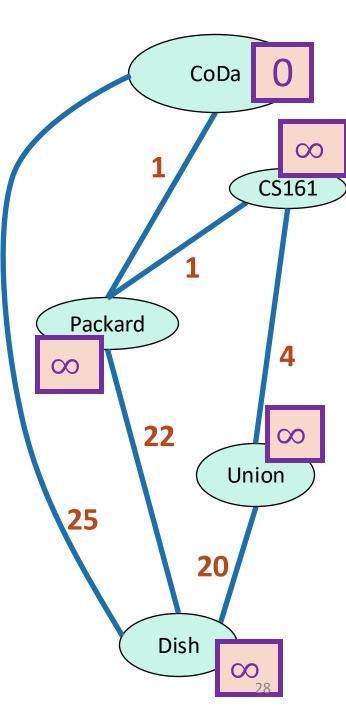


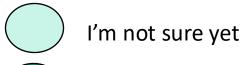


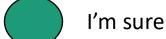
x = d[v] is my best over-estimate
for dist(CoDa,v).

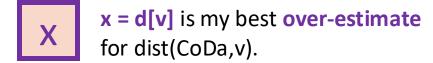
Initialize  $d[v] = \infty$ for all non-starting vertices v, and d[CoDa] = 0

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



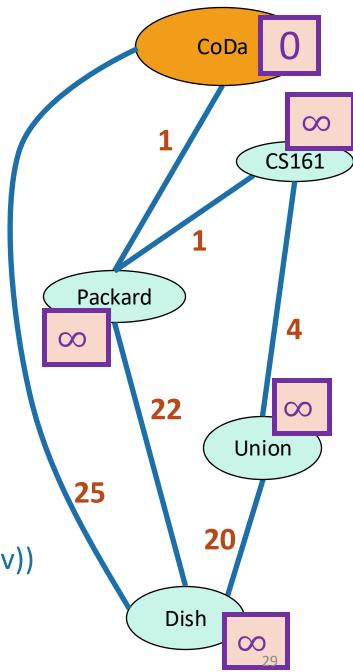


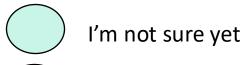


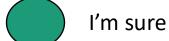


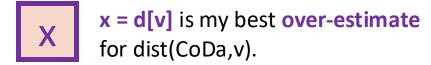


- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))



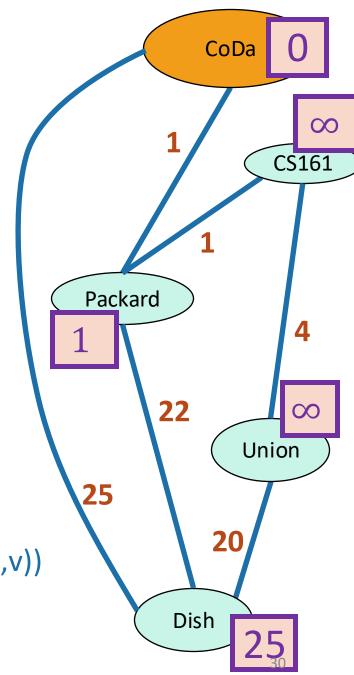


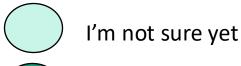


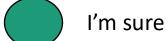


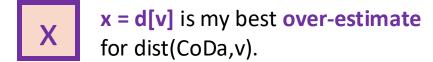


- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min( d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.



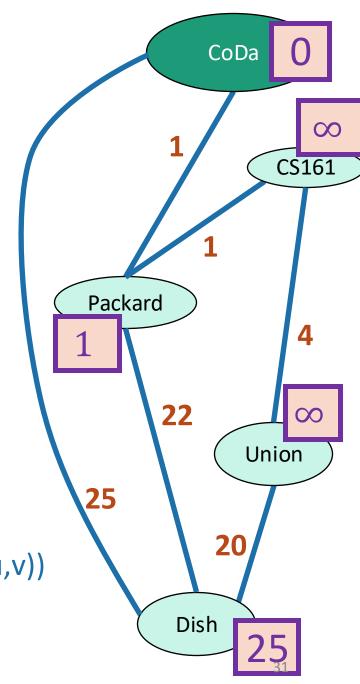








- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.
- Repeat



How far is a node from CoDa?

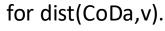
I'm not sure yet



I'm sure



x = d[v] is my best over-estimate

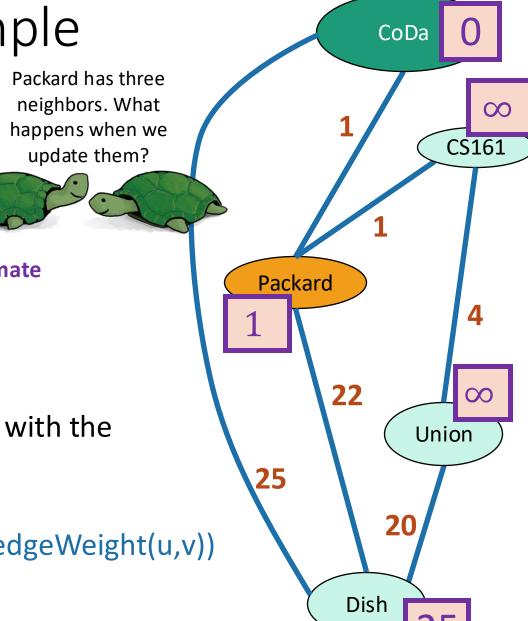




Current node u



- Update all u's neighbors v:
  - d[v] = min( d[v] , d[u] + edgeWeight(u,v))
- Mark u as **SUCC**.
- Repeat



How far is a node from CoDa?

I'm not sure yet



I'm sure



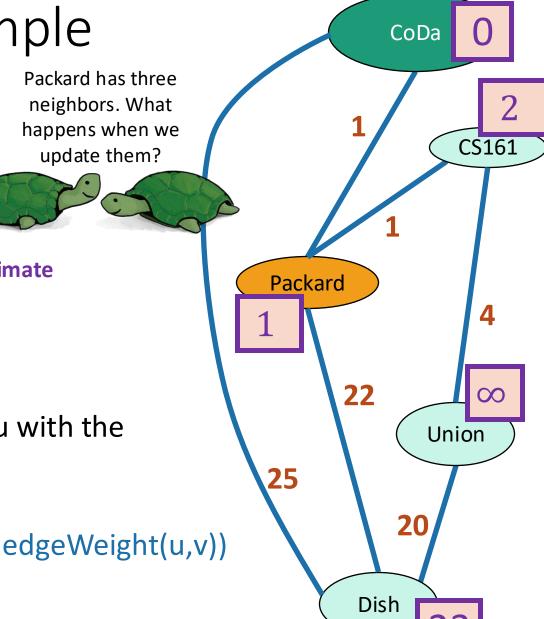
x = d[v] is my best over-estimate for dist(CoDa,v).

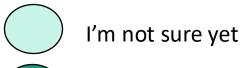


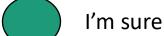
Current node u

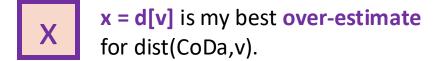
 Pick the not-sure node u with the smallest estimate d[u].

- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.
- Repeat



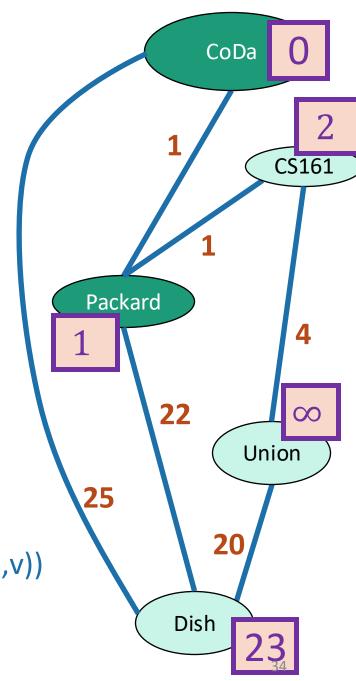


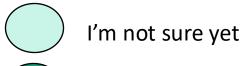


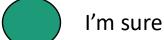


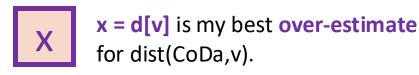


- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.
- Repeat



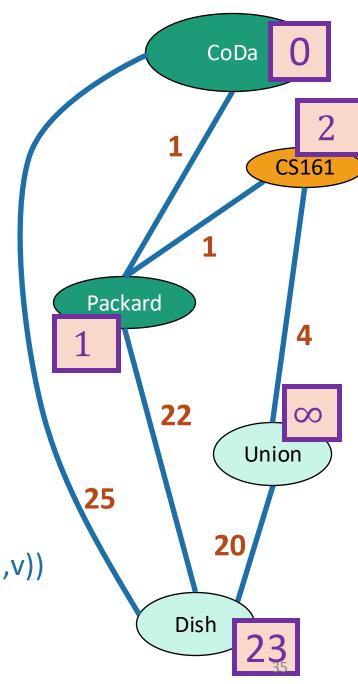


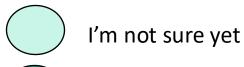


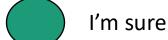


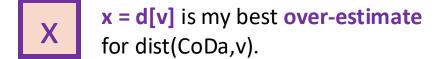


- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.
- Repeat



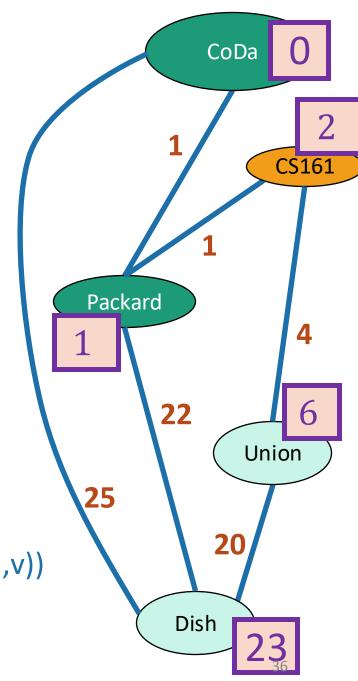








- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.
- Repeat



#### How far is a node from CoDa?



I'm not sure yet



I'm sure

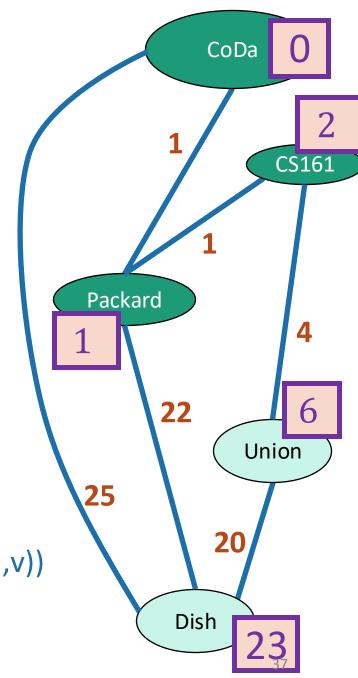


x = d[v] is my best over-estimate
for dist(CoDa,v).

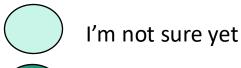


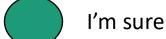
Current node u

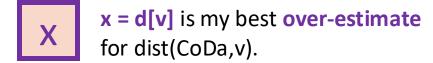
- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.
- Repeat



#### How far is a node from CoDa?

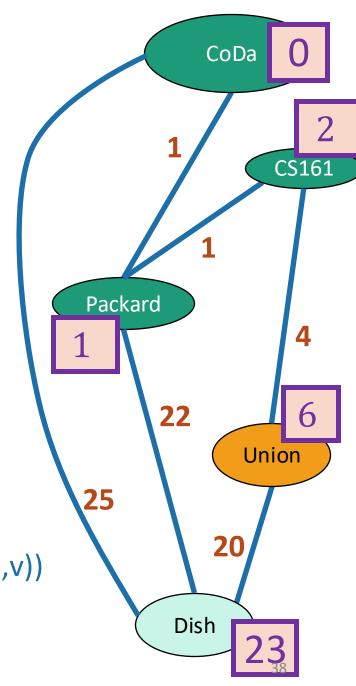




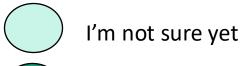


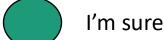


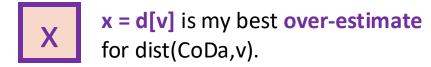
- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.
- Repeat



#### How far is a node from CoDa?

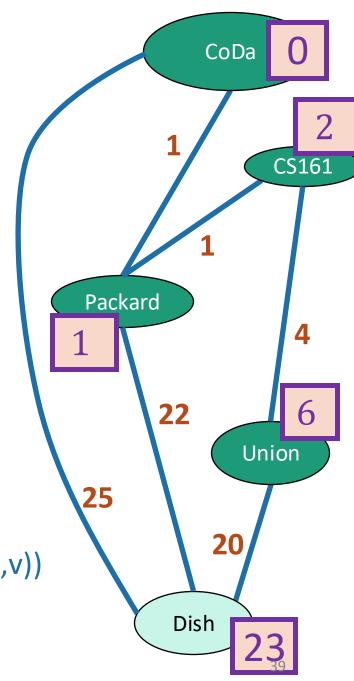








- Pick the not-sure node u with the smallest estimate d[u].
- Update all u's neighbors v:
  - d[v] = min(d[v], d[u] + edgeWeight(u,v))
- Mark u as Sure.
- Repeat



#### How far is a node from CoDa?



I'm not sure yet



I'm sure

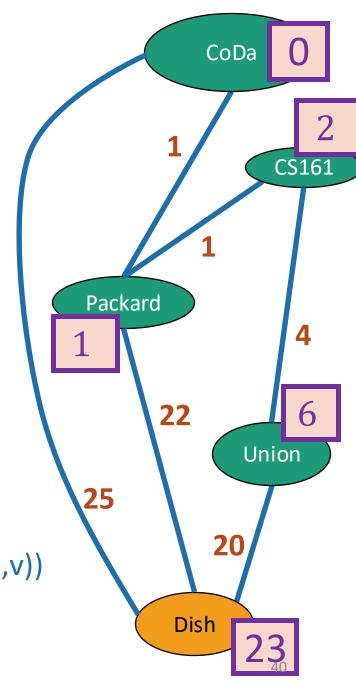


x = d[v] is my best over-estimate
for dist(CoDa,v).



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



#### How far is a node from CoDa?



I'm not sure yet



I'm sure

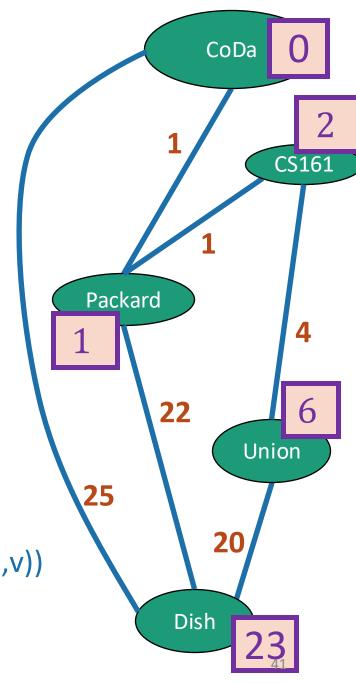


x = d[v] is my best over-estimate
for dist(CoDa,v).

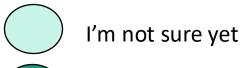


Current node u

- 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



#### How far is a node from CoDa?



I'm sure

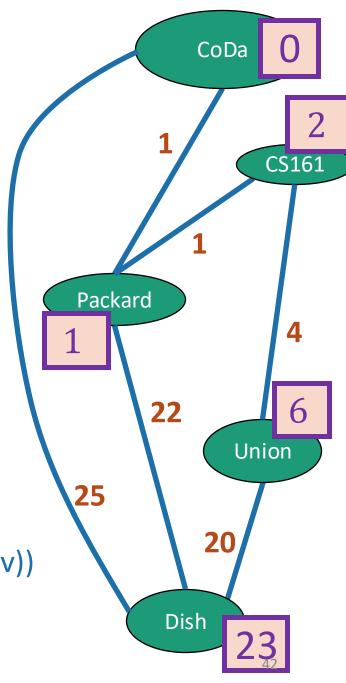


x = d[v] is my best over-estimate
for dist(CoDa,v).



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(CoDa, 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: Let G be a directed, weighted graph with non-negative edge weights.
  - Suppose we run Dijkstra on G = (V,E), starting from s.
  - At the end of the algorithm, the estimate d[v] is the actual distance d(s,v).

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

- Proof outline:
  - Claim 1: For all v,  $d[v] \ge d(s,v)$ .
  - Claim 2: When a vertex v is marked sure, d[v] = d(s,v).
- Claims 1 and 2 imply the theorem.
  - When v is marked sure, d[v] = d(s,v).

Claim 1 + def of algorithm

Claim 2

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

Next let's prove the claims!

### Claim 1

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

#### Informally:

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

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

Whatever path we had in mind before

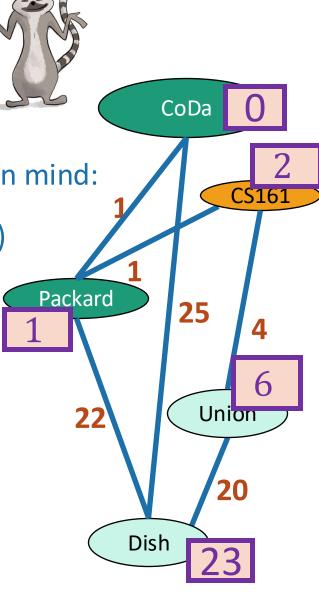
The shortest path to u, and then the edge from u to v.

d[v] = length of the path we have in mind
 ≥ length of shortest path

= d(s,v)

#### Formally:

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



Intuition!

### Claim 1

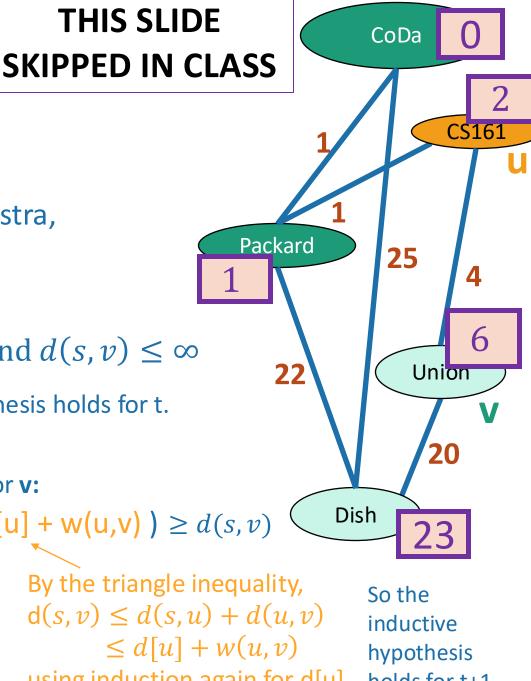
 $d[v] \ge d(s,v)$  for all v.

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

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

By the triangle inequality,  $d(s, v) \le d(s, u) + d(u, v)$  $\leq d[u] + w(u, v)$ using induction again for d[u]

So the inductive hypothesis holds for t+1.



**Conclusion:** After Dijkstra is done,  $d[v] \ge d(s,v)$ , so Claim 1 holds!

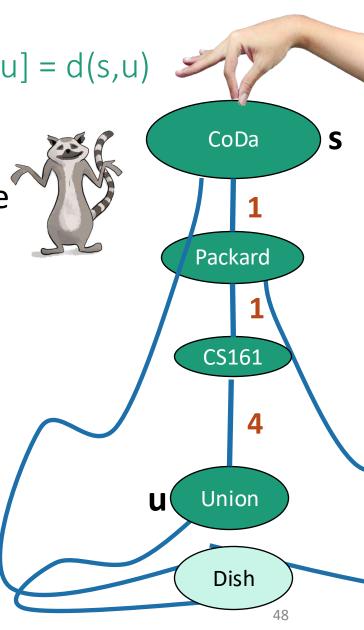
### Intuition for Claim 2

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).

Claim 2
Inductive step

**Temporary definition:** 

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

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

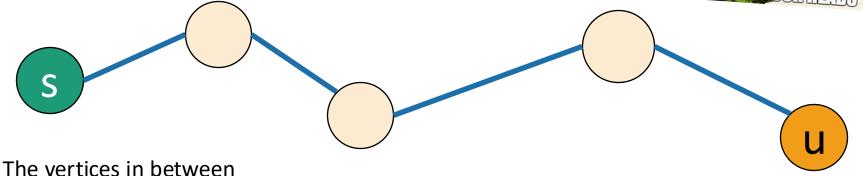
Claim 2
Inductive step

#### **Temporary definition:**

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

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





are beige because they may or may not be sure.

True shortest path.

> Claim 2 Inductive step

may or may not be sure.

#### **Temporary definition:**

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

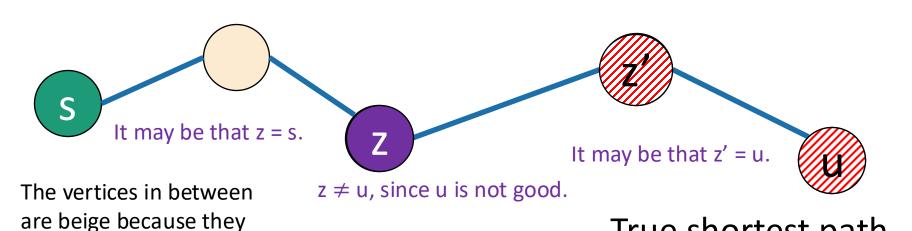




True shortest path.

"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.



# 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.

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

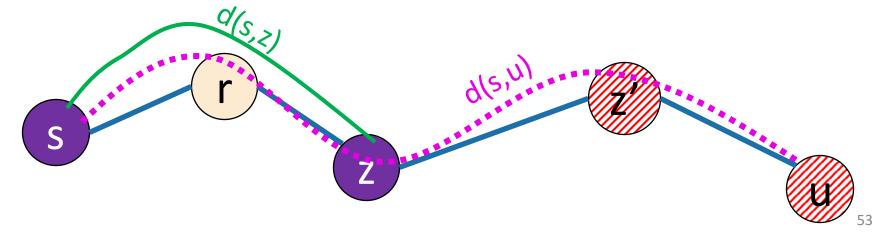
z is good

Subpaths of

shortest paths are

shortest paths.

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



Claim 2
Inductive step

**Temporary definition:** 

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



means good



means not good

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

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

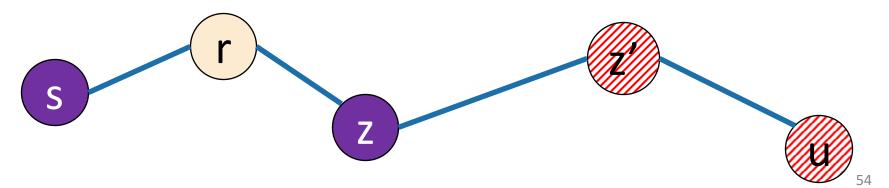
z is good

Subpaths of shortest paths are shortest paths.

Claim 1 says that  $d(s, u) \le d[u]$ And they aren't equal since 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.

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

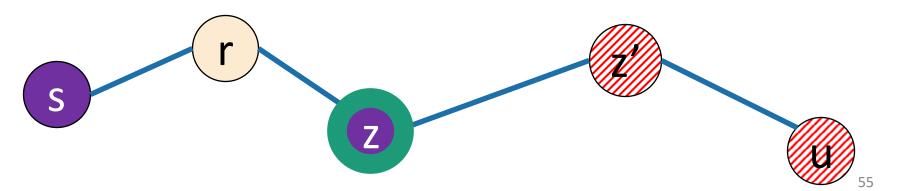
z is good

Subpaths of shortest paths are shortest paths.

Claim 1 says that  $d(s, u) \le d[u]$ And they aren't equal since 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.
- Since z is sure then we've already updated z':

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

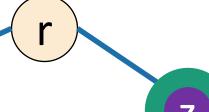
•  $d[z'] \le d[z] + w(z, z')$  def of update

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

That is, the value of d[z] when z was = d(s, z') sub-paths of shortest paths are shortest paths marked sure...

$$\leq d[z']$$
 Claim 1

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



N(2,2')

CONTRADICTION!!

So u is good!



# Back to this slide

#### Claim 2

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

- Inductive Hypothesis:
  - When we mark the t'th vertex v as sure, d[v] = dist(s,v).
- Base case:
  - The first vertex marked sure is s, and d[s] = d(s,s) = 0.
- Inductive step:
  - Suppose that we are about to add u to the sure list.
  - That is, we picked u in the first line here:
    - Pick the not-sure node u with the smallest estimate d[u].
    - Update all u's neighbors v:
      - d[v] ← min( d[v] , d[u] + edgeWeight(u,v))
    - Mark u as sure.
    - Repeat
  - Assume by induction that every v already marked sure has d[v] = d(s,v).
  - Want to show that d[u] = d(s,u).

# Why does this work?



#### Theorem:

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

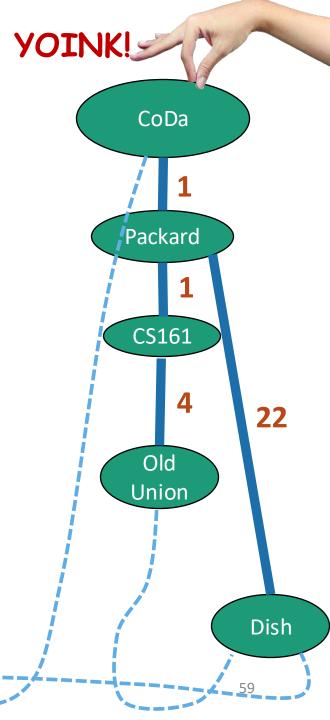
#### Proof outline:

- Claim 1: For all v,  $d[v] \ge d(s,v)$ .
- 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 CoDa!
  - 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?

### We need a data structure that:

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

#### Just the inner loop:

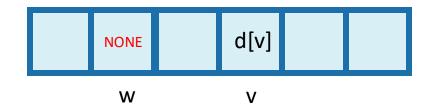
- Pick the **not-sure** node u with the smallest estimate **d[u]**.
- Update all u's neighbors v:
  - d[v] ← min( d[v] , d[u] + edgeWeight(u,v))
- Mark u as sure.

Total running time is big-oh of:

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

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

# If we use an array



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

time to find and also remove the min

– the time to remove the min once
you've found it is just O(1), but the
difference doesn't matter in the
running time for Dijkstra

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

Running time of Dijkstra

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

- =O(nlog(n)) + O(mlog(n))
- $=O((n + m)\log(n))$

Better than an array if the graph is sparse!

aka if m is much smaller than n<sup>2</sup>

d[v]

d[w]

d[u]

### If we use a Fibonacci Heap

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

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

Running time of Dijkstra

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

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

Compare:

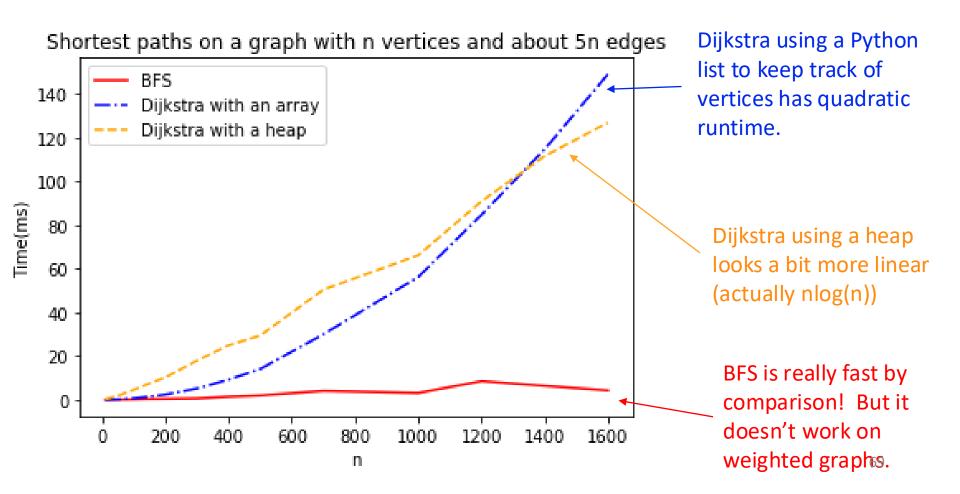
Array: O(n<sup>2</sup>)

RBTree: O((n+m)log n)

\*This means that any sequence of d removeMin calls takes time at most O(dlog(n)).

But a few of the d may take longer than O(log(n)) and some may take less time..

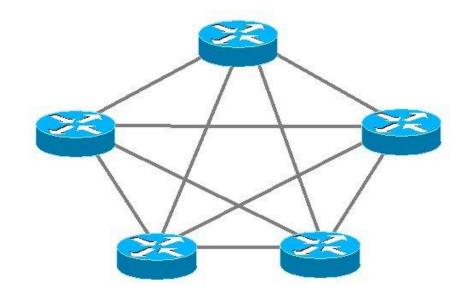
# In practice



### Dijkstra is used in practice

• eg, OSPF (Open Shortest Path First), a routing protocol for IP networks, uses Dijkstra.

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



# Dijkstra Drawbacks

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

### Bellman-Ford algorithm

• (-) Slower than Dijkstra's algorithm

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

### How are we doing on time?

• If we're out of time, we'll see Bellman-Ford in the next lecture!

#### **UPDATE** (added after class):

Indeed we were out of time! Don't worry about the slides after this point (especially for the midterm, which won't cover Bellman-Ford), we'll cover them next time.



### Today: *intro* to Bellman-Ford

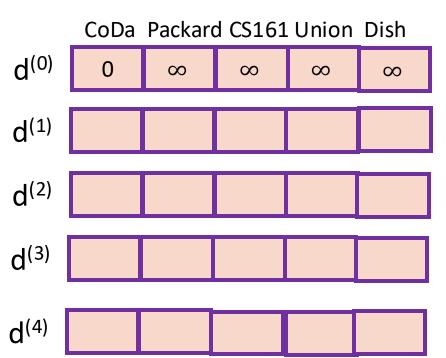
- We'll see a definition by example.
- We'll come back to it next lecture with more rigor.
  - Don't worry if it goes by quickly today.
  - We'll see formal definitions/pseudocode next time.

#### Basic idea:

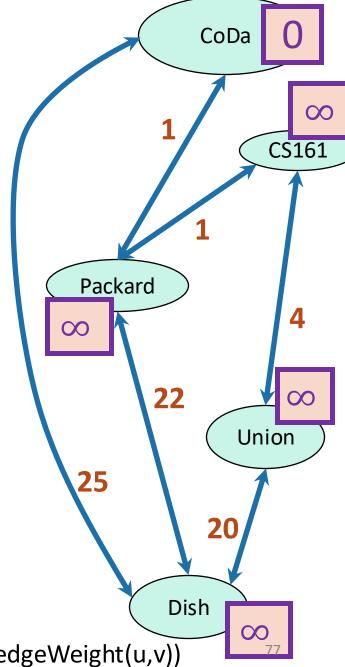
 Instead of picking the u with the smallest d[u] to update, just update all of the u's simultaneously. Bellman-Ford

Start with the same graph, no negative weights.

#### How far is a node from CoDa?



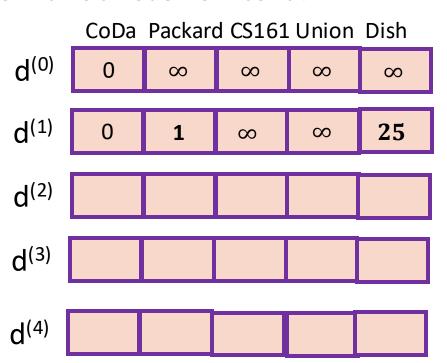
- **For** i=0,...,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + edgeWeight(u,v))$



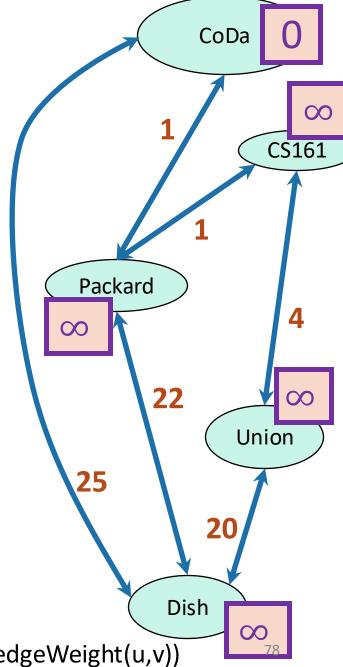
Bellman-Ford

Start with the same graph, no negative weights.

#### How far is a node from CoDa?

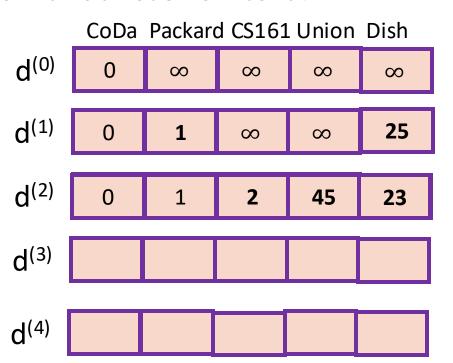


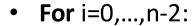
- **For** i=0,...,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + edgeWeight(u,v))$



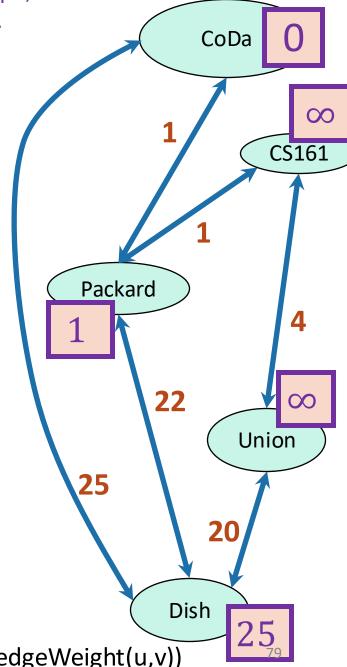
Start with the same graph, no negative weights.

#### How far is a node from CoDa?



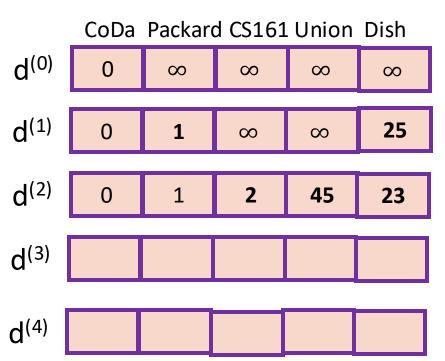


- **For** u in V:
  - **For** v in u.neighbors:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + edgeWeight(u,v))$

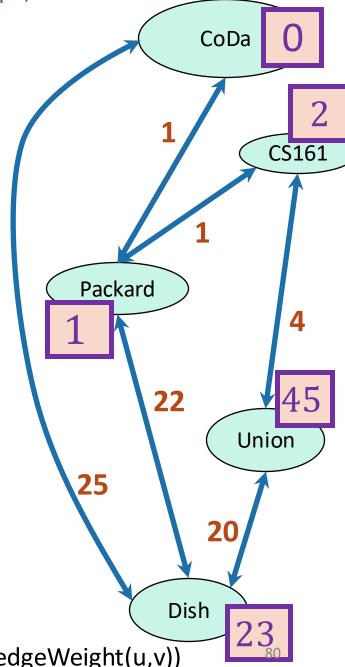


Start with the same graph, no negative weights.

#### How far is a node from CoDa?

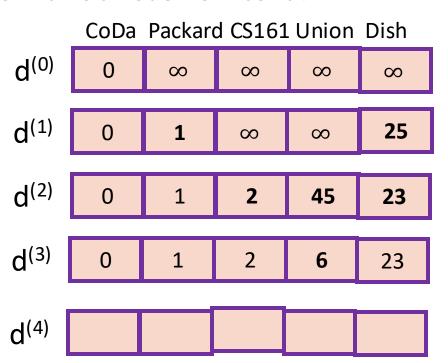


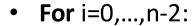
- **For** i=0,...,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + edgeWeight(u,v))$



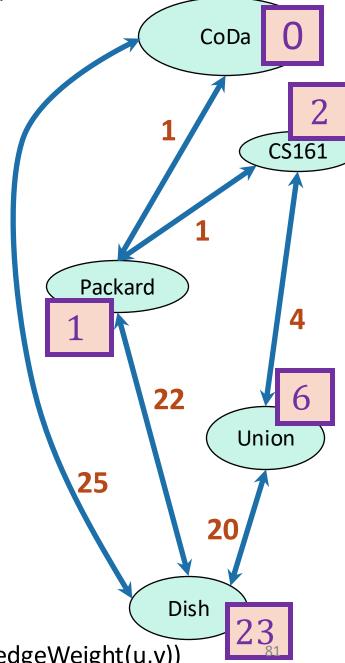
Start with the same graph, no negative weights.

#### How far is a node from CoDa?



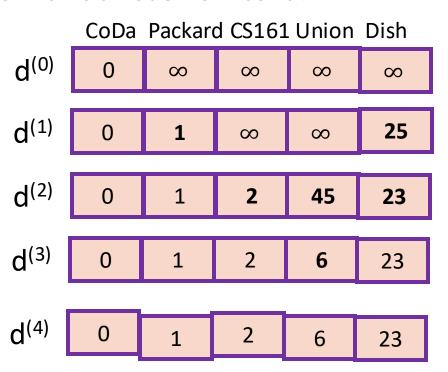


- **For** u in V:
  - **For** v in u.neighbors:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + edgeWeight(u,v))$



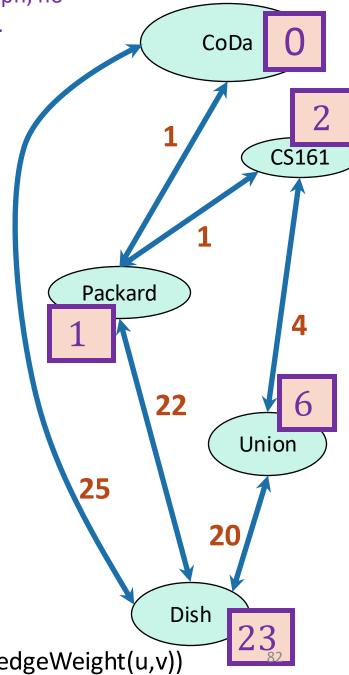
Start with the same graph, no negative weights.

#### How far is a node from CoDa?



These are the final distances!

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



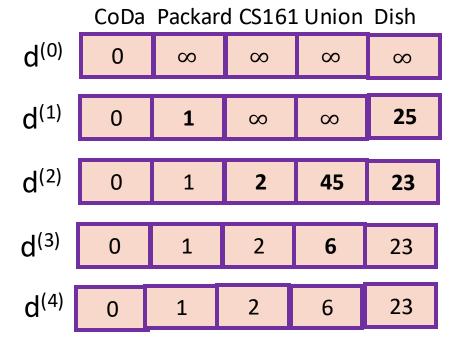
## As usual

- Does it work?
  - Yes
  - Idea to the right.
  - (See skipped slides for details)

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

A simple path is a path with no cycles.





**Idea:** proof by induction.

#### **Inductive Hypothesis:**

d<sup>(i)</sup>[v] is equal to the cost of the shortest path between s and v with at most i edges.

#### **Conclusion:**

d<sup>(n-1)</sup>[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).

## **Skipped in class**

# Proof by induction

- Inductive Hypothesis:
  - After iteration i, for each v, d<sup>(i)</sup>[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<sup>(i)</sup> [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.

  U

  V

  at most i edges
- By induction, d<sup>(i)</sup>[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  $d^{(i+1)}[v] \le d^{(i)}[u] + w(u,v)$ .
- So d<sup>(i+1)</sup>[v] <= cost of shortest path between s and v with i+1 edges.</li>
- But  $d^{(i+1)}[v] = \cos t$  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<sup>(i)</sup>[v] is equal to the cost of the shortest path between s and v of length at most i edges.

#### Base case:

• After iteration 0...

## • Inductive step:

#### Conclusion:

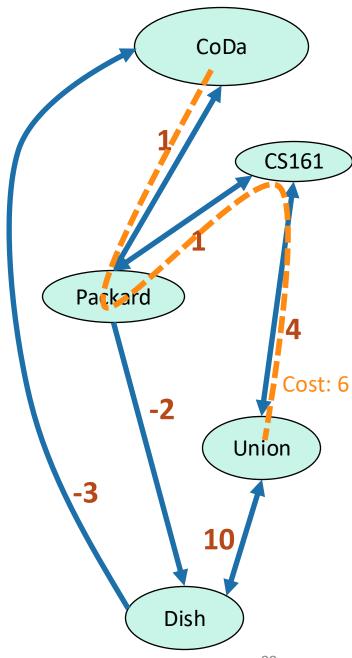
- After iteration n-1, for each v, d[v] is equal to the cost of the shortest path between s and v of length at most n-1 edges.
- Aka, d[v] = d(s,v) for all v as long as there are no negative cycles!

# Nice things about Bellman-Ford

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

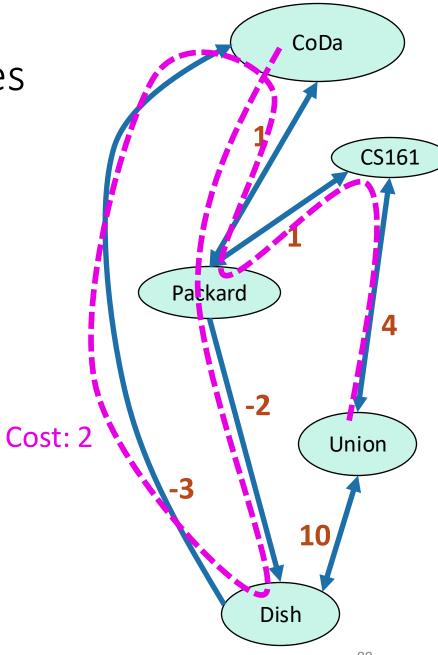
## Caution: negative cycles

 What is the shortest path from CoDa to Old Union?



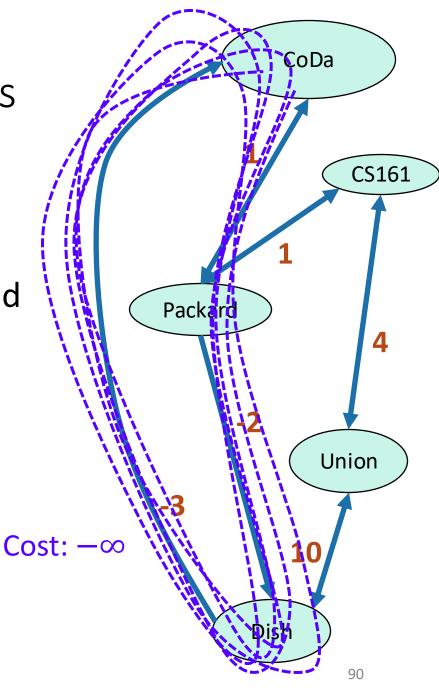
Caution: negative cycles

 What is the shortest path from CoDa to Old Union?



Caution: negative cycles

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



# Bellman-Ford and negative edge weights

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

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



# Summary

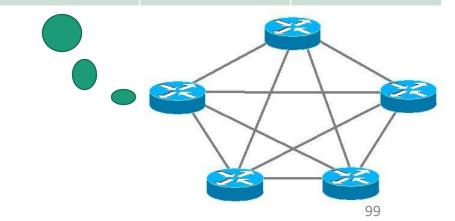
It's okay if that went by fast, we'll come back to Bellman-Ford

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

# Bellman-Ford is also used in practice.

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

| Destination  | Cost to get<br>there | Send to whom? |
|--------------|----------------------|---------------|
| 172.16.1.0   | 34                   | 172.16.1.1    |
| 10.20.40.1   | 10                   | 192.168.1.2   |
| 10.155.120.1 | 9                    | 10.13.50.0    |



# Recap: shortest paths

#### • BFS:

- (+) O(n+m)
- (-) only unweighted graphs

## Dijkstra's algorithm:

- (+) weighted graphs
- (+) O(nlog(n) + m) if you implement it with a Fibonacci heap
- (-) no negative edge weights
- (-) very "centralized" (need to keep track of all the vertices to know which to update).

## Bellman-Ford algorithm:

- (+) weighted graphs, even with negative weights
- (+) can be done in a distributed fashion, every vertex using only information from its neighbors.
- (-) O(nm)

## **Next Time**

- No class Tuesday (Democracy Day)
- On Thursday: Midterm 2!
- (and after that, Dynamic Programming!!!)

## Before next time

- Study for the midterm!
- And after that, pre-lecture exercise for Lecture 12
  - Fibonacci numbers!