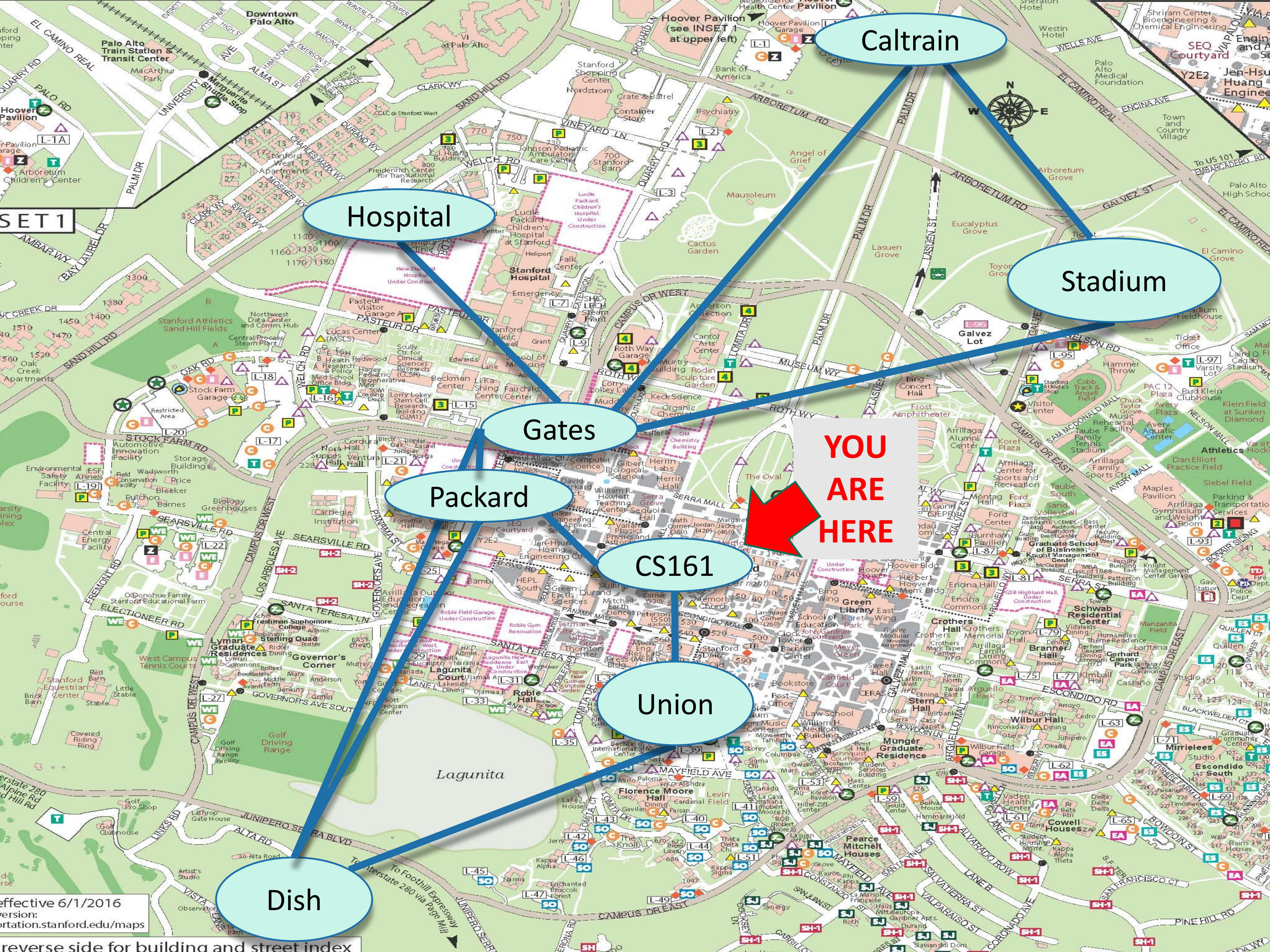
Module – Graphs II

Modified/Inspired from Stanford's CS161 by Nayyar Zaidi

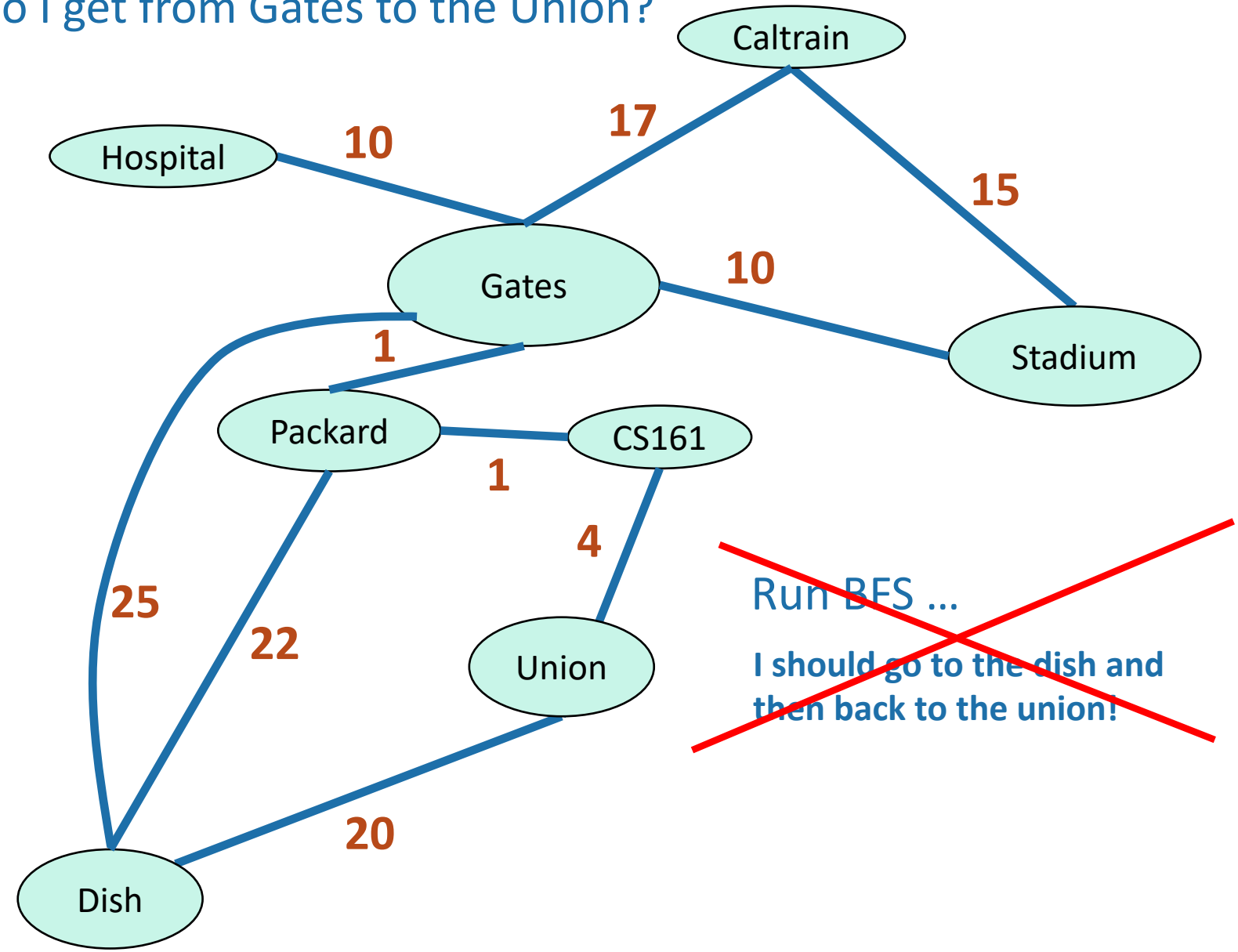
This module

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


Caltrain
Hospital
Stadium
Gates
Packard
CS161
YOU ARE HERE
Union
Dish
Lagunita
Downtown Palo Alto
effective 6/1/2016
reverse side for building and street index

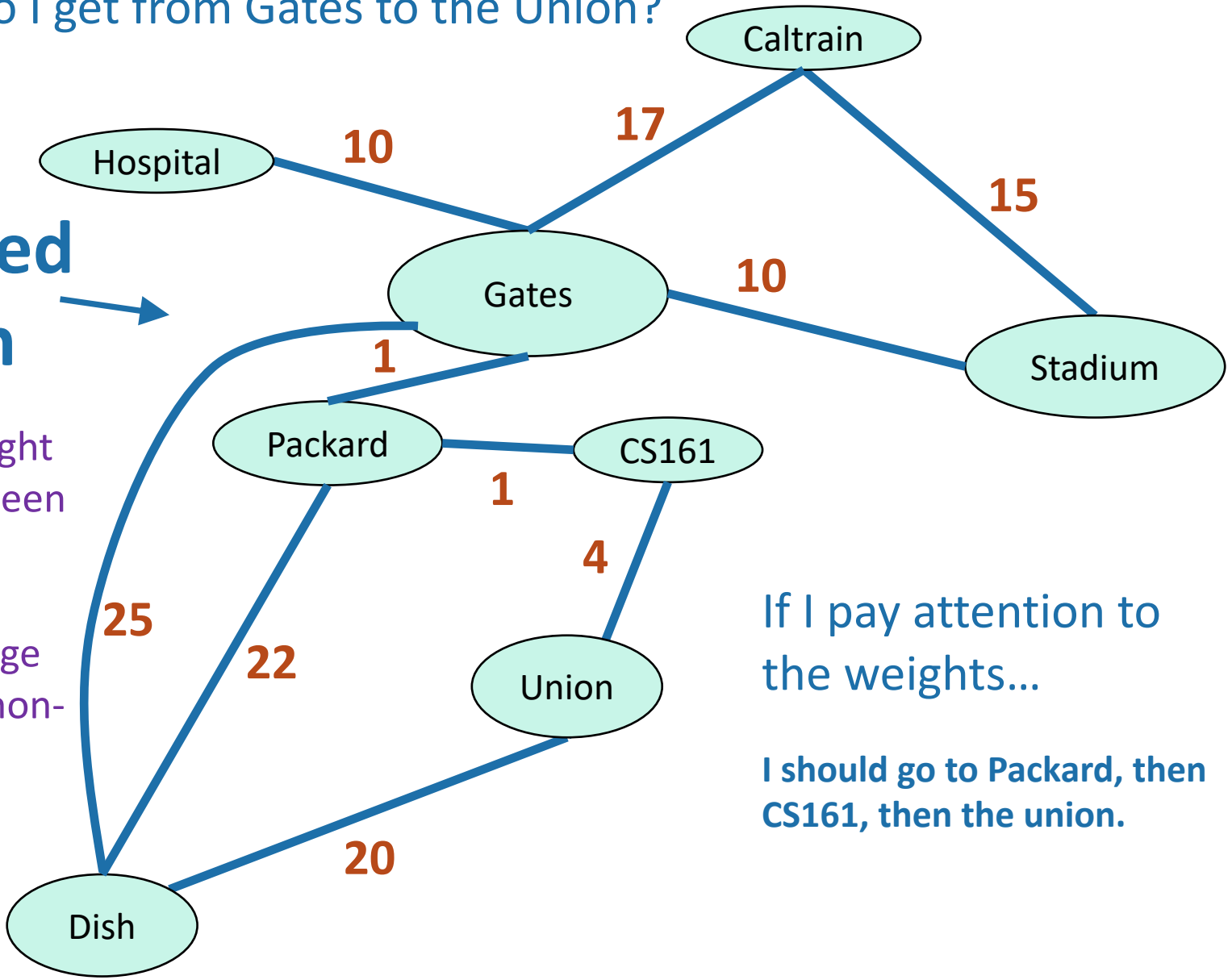
Just the graph

How do I get from Gates to the Union?



Just the graph

How do I get from Gates to the Union?



weighted graph

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

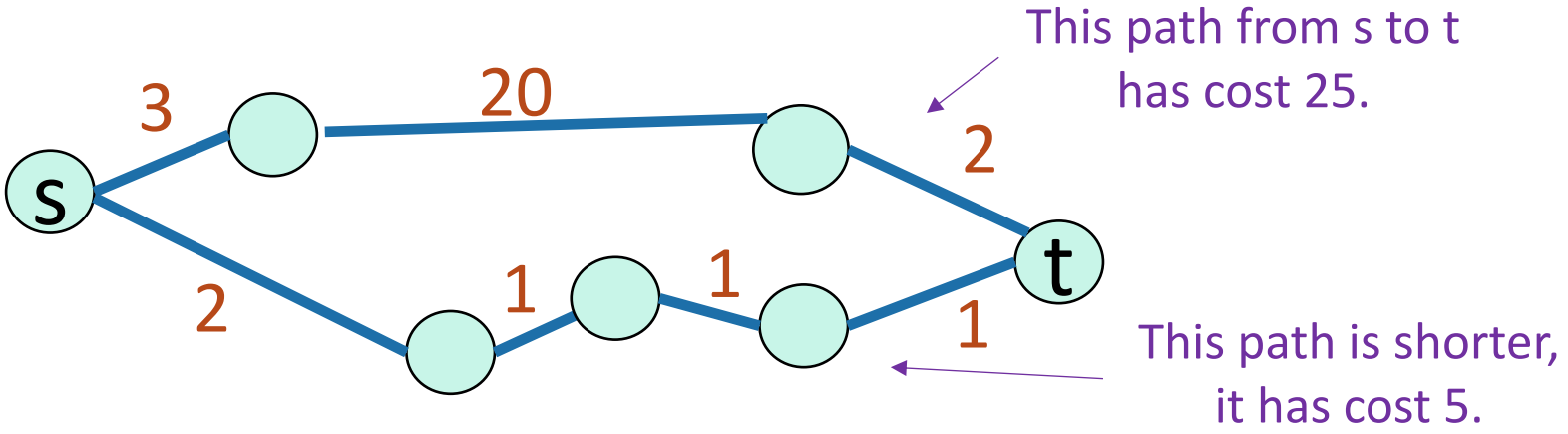
For now, edge weights are non-negative.

If I pay attention to the weights…

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

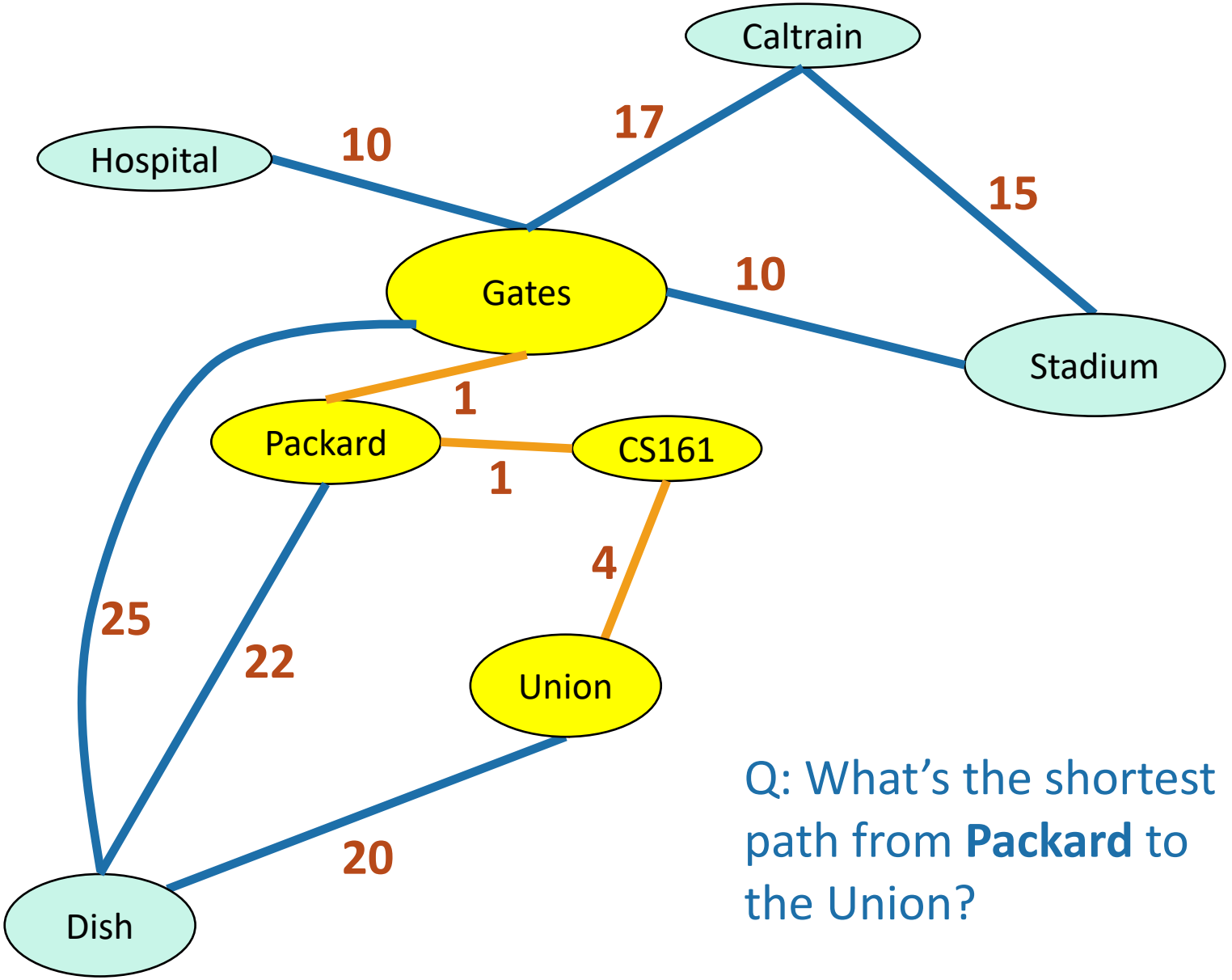
Shortest path problem

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



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

Shortest paths



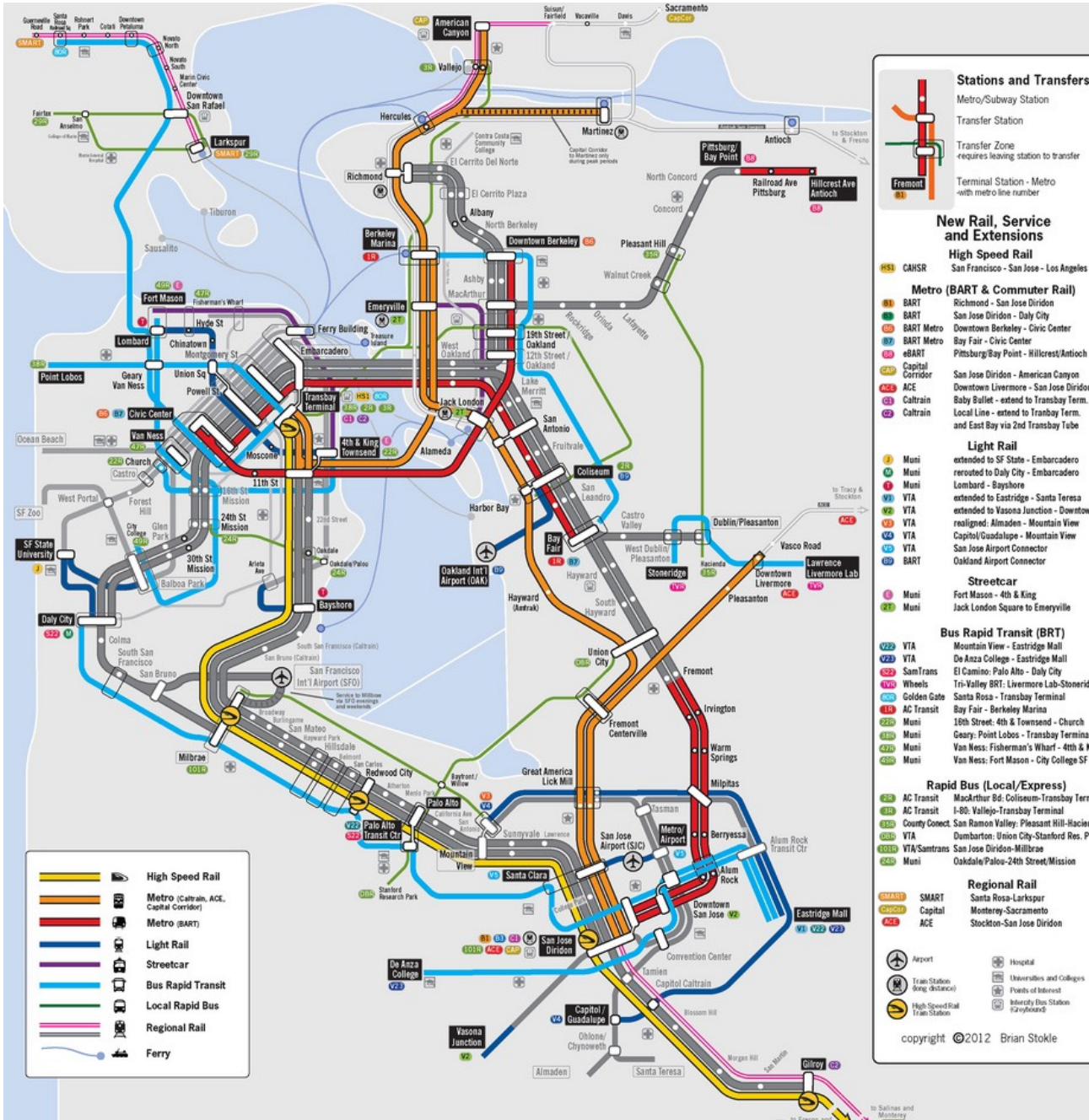
Q: What's the shortest path from **Packard** to the Union?

Single-source shortest-path problem

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

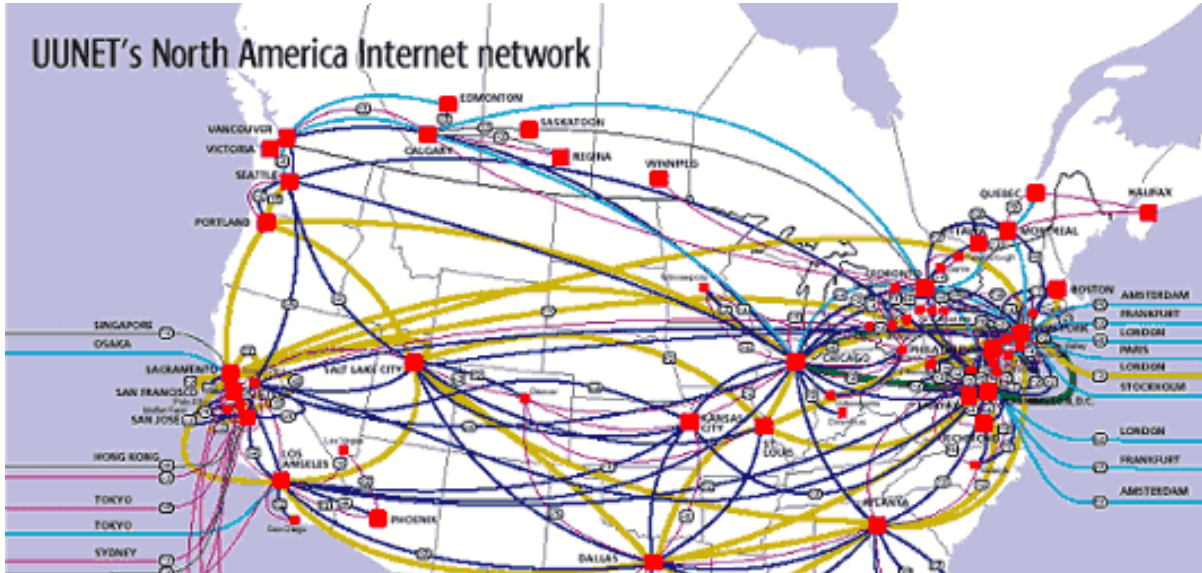
Destination	Cost	To get there
Packard	1	Packard
CS161	2	Packard-CS161
Hospital	10	Hospital
Caltrain	17	Caltrain
Union	6	Packard-CS161-Union
Stadium	10	Stadium
Dish	23	Packard-Dish

Example


Stations and Transfers
Metro/Subway Station
Transfer Station
Transfer Zone
-requires leaving station to transfer
Terminal Station - Metro
-with metro line number
New Rail, Service and Extensions
High Speed Rail
Metro (BART & Commuter Rail)
Light Rail
Streetcar
Bus Rapid Transit (BRT)
Rapid Bus (Local/Express)
Regional Rail
High Speed Rail
Metro (Caltrain, ACE, Capital Corridor)
Metro (BART)
Light Rail
Streetcar
Bus Rapid Transit
Local Rapid Bus
Regional Rail
Ferry
copyright ©2012 Brian Stokle

Example

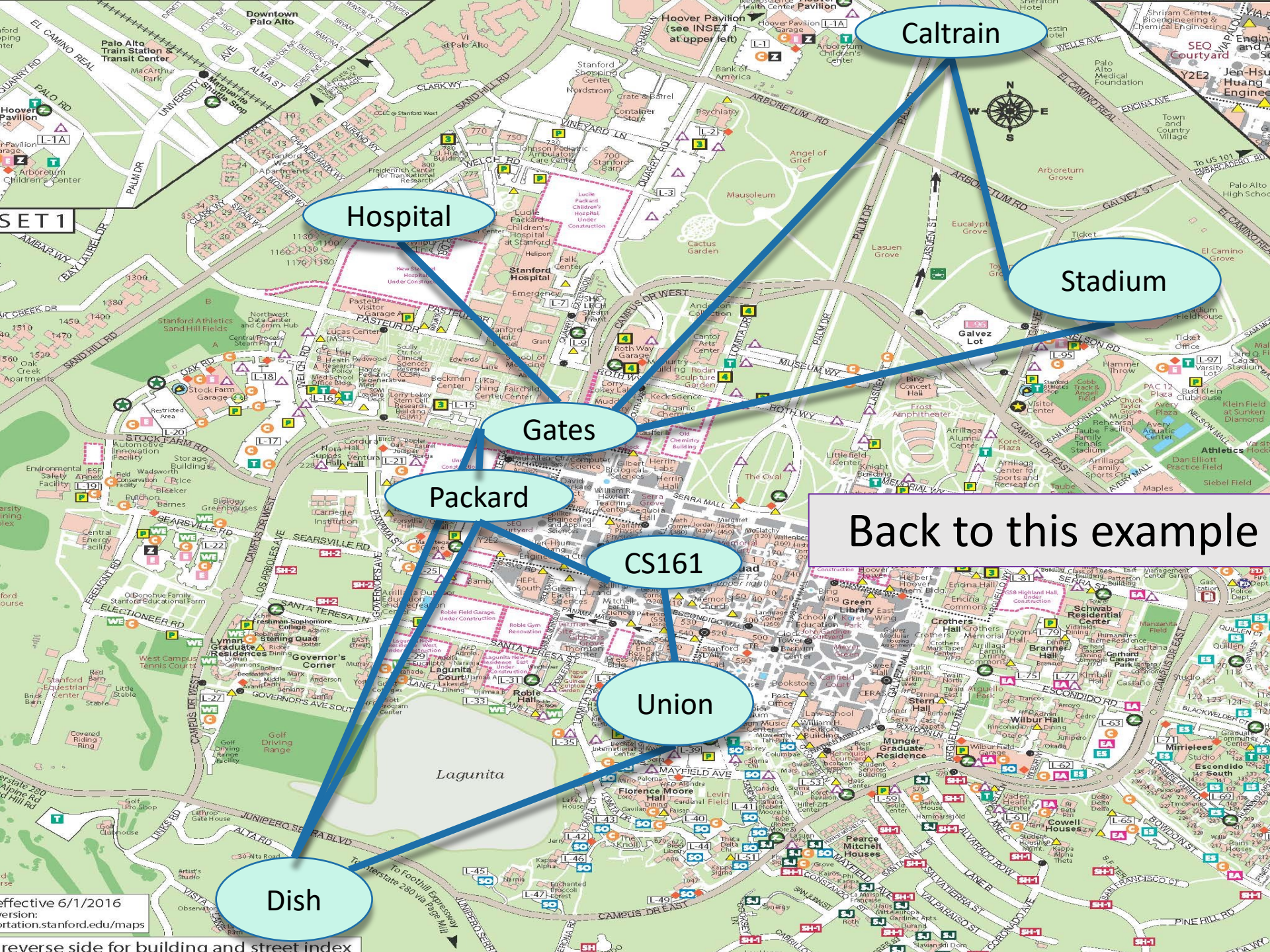
- **Network routing**
- I send information over the internet, from my computer to to all over the world.
- Each path has a cost which depends on link length, traffic, other costs, etc..
- How should we send packets?




```
DN0a22a0e3:~ mary$ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1  [AS0] 10.34.160.2 (10.34.160.2)  38.168 ms  31.272 ms  28.841 ms
 2  [AS0] cwa-vrtr.sunet (10.21.196.28)  33.769 ms  28.245 ms  24.373 ms
 3  [AS32] 171.66.2.229 (171.66.2.229)  24.468 ms  20.115 ms  23.223 ms
 4  [AS32] hpr-svl-rtr-vlan8.sunet (171.64.255.235)  24.644 ms  24.962 ms  1
 5  [AS2152] hpr-svl-hpr2--stan-ge.cenic.net (137.164.27.161)  22.129 ms  4.9
 6  [AS2152] hpr-lax-hpr3--svl-hpr3-100ge.cenic.net (137.164.25.73)  12.125 r
 7  [AS2152] hpr-i2--lax-hpr2-r&e.cenic.net (137.164.26.201)  40.174 ms  38.3
 8  [AS0] et-4-0-0.4079.sdn-sw.lasv.net.internet2.edu (162.252.70.28)  46.573
 9  [AS0] et-5-1-0.4079.rtsw.salt.net.internet2.edu (162.252.70.31)  30.424 r
10  [AS0] et-4-0-0.4079.sdn-sw.denv.net.internet2.edu (162.252.70.8)  47.454
11  [AS0] et-4-1-0.4079.rtsw.kans.net.internet2.edu (162.252.70.11)  70.825 r
12  [AS0] et-4-1-0.4070.rtsw.chic.net.internet2.edu (198.71.47.206)  77.937 r
13  [AS0] et-0-1-0.4079.sdn-sw.ashb.net.internet2.edu (162.252.70.60)  77.682
14  [AS0] et-4-1-0.4079.rtsw.wash.net.internet2.edu (162.252.70.65)  71.565 r
15  [AS21320] internet2-gw.mx1.lon.uk.geant.net (62.40.124.44)  154.926 ms  1
16  [AS21320] ae0.mx1.lon2.uk.geant.net (62.40.98.79)  146.565 ms  146.604 ms
17  [AS21320] ae0.mx1.par.fr.geant.net (62.40.98.77)  153.289 ms  184.995 ms
18  [AS21320] ae2.mx1.gen.ch.geant.net (62.40.98.153)  160.283 ms  160.104 ms
19  [AS21320] swice1-100ge-0-3-0-1.switch.ch (62.40.124.22)  162.068 ms  160.
20  [AS559] swizh1-100ge-0-1-0-1.switch.ch (130.59.36.94)  165.824 ms  164.21
21  [AS559] swiez3-100ge-0-1-0-4.switch.ch (130.59.38.109)  164.269 ms  164.3
22  [AS559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1)  164.082 ms  17
23  [AS559] rou-fw-rz-rz-gw.ethz.ch (192.33.92.169)  164.773 ms  165.193 ms
```

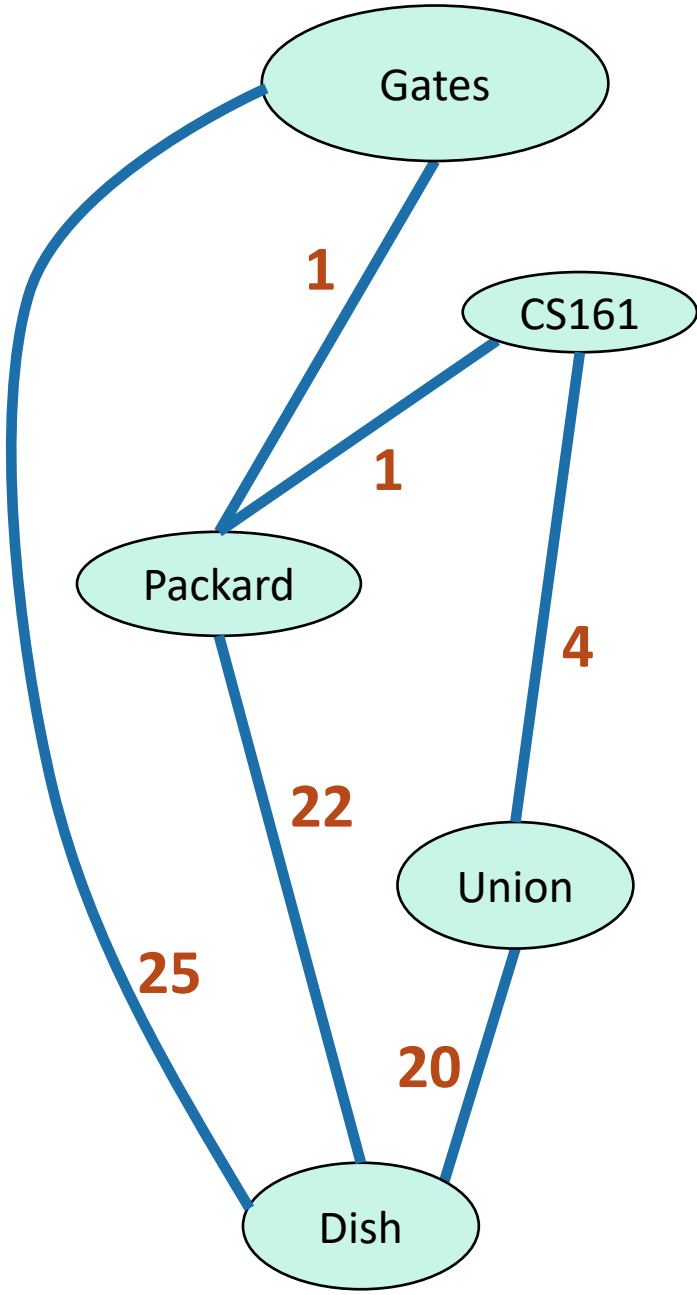
Aside: These are difficult problems

- Costs may change
  - If it's raining the cost of biking is higher
  - If a link is congested, the cost of routing a packet along it is higher
- The network might not be known
  - My computer doesn't store a map of the internet
- We want to do these tasks really quickly


Caltrain
Hospital
Stadium
Gates
Packard
CS161
Union
Dish
Back to this example

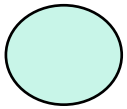
Dijkstra's algorithm

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

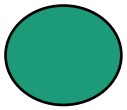


Dijkstra by example

How far is a node from Gates?



I'm not sure yet

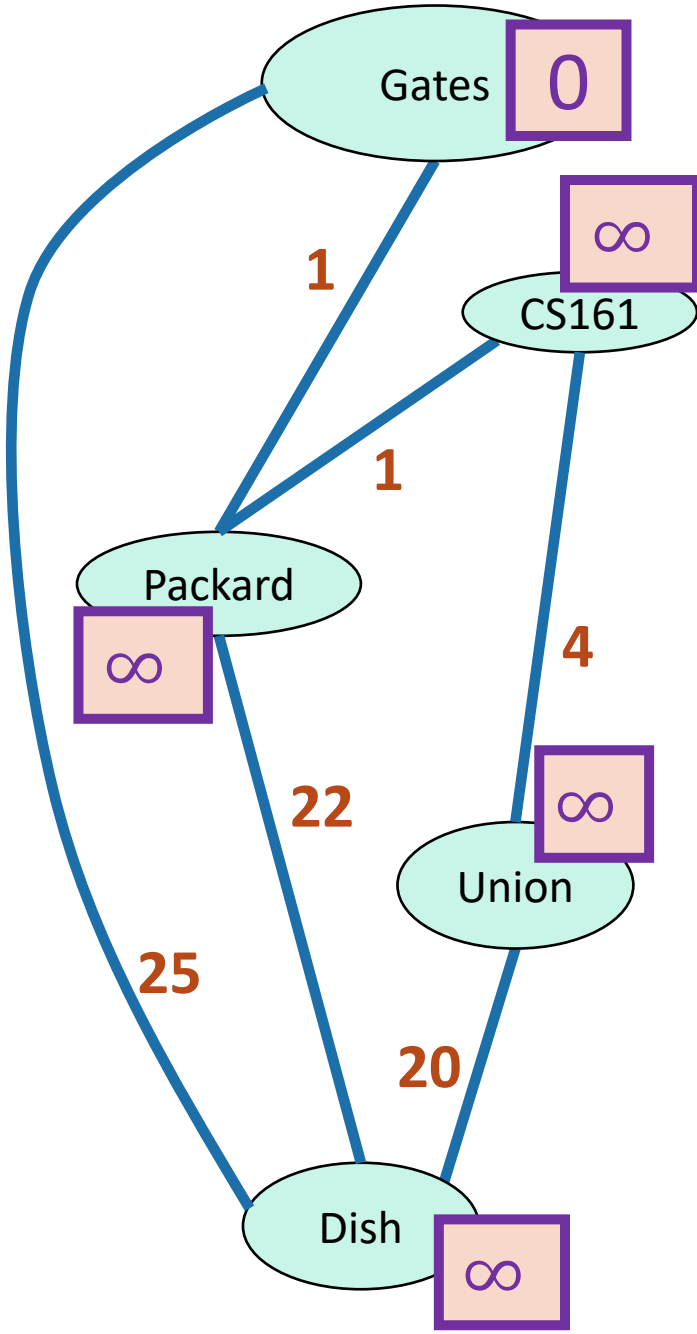


I'm sure

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

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

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



Dijkstra by example

How far is a node from Gates?

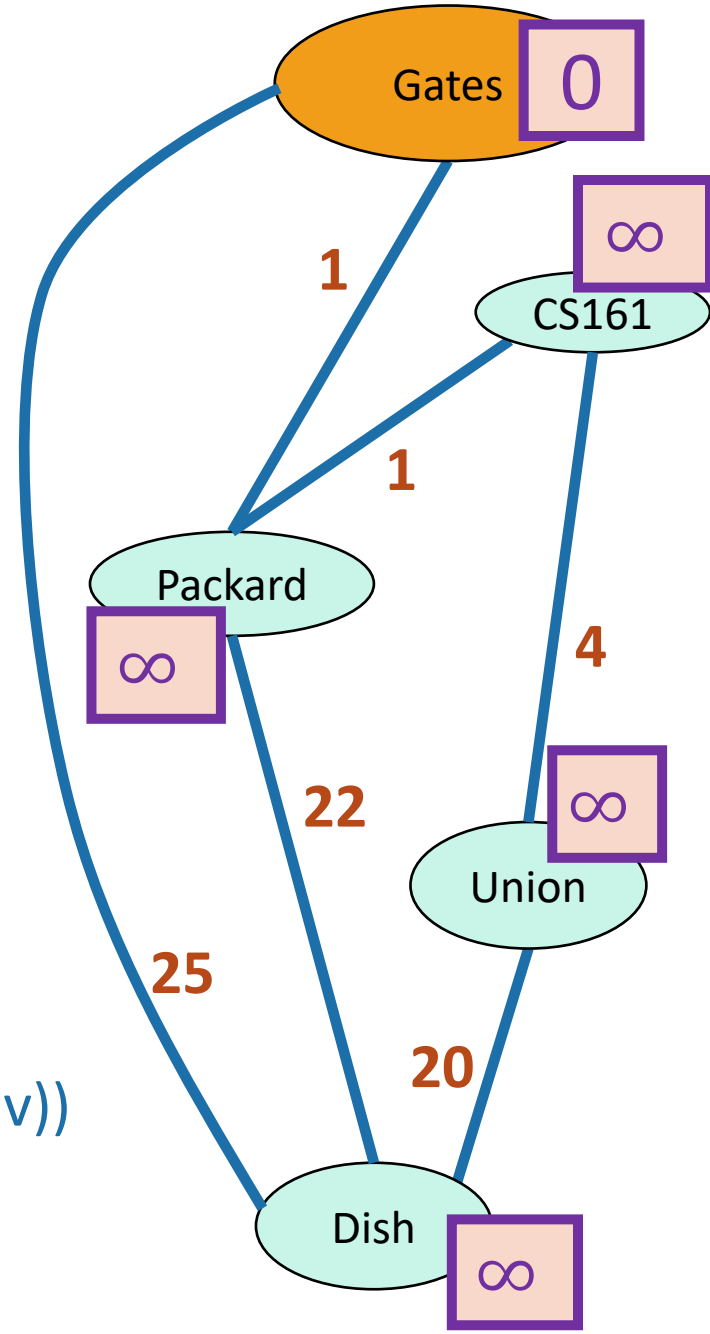
I'm not sure yet

I'm sure

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

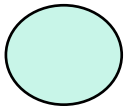
Current node u

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

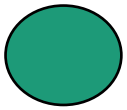


Dijkstra by example

How far is a node from Gates?

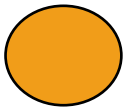


I'm not sure yet



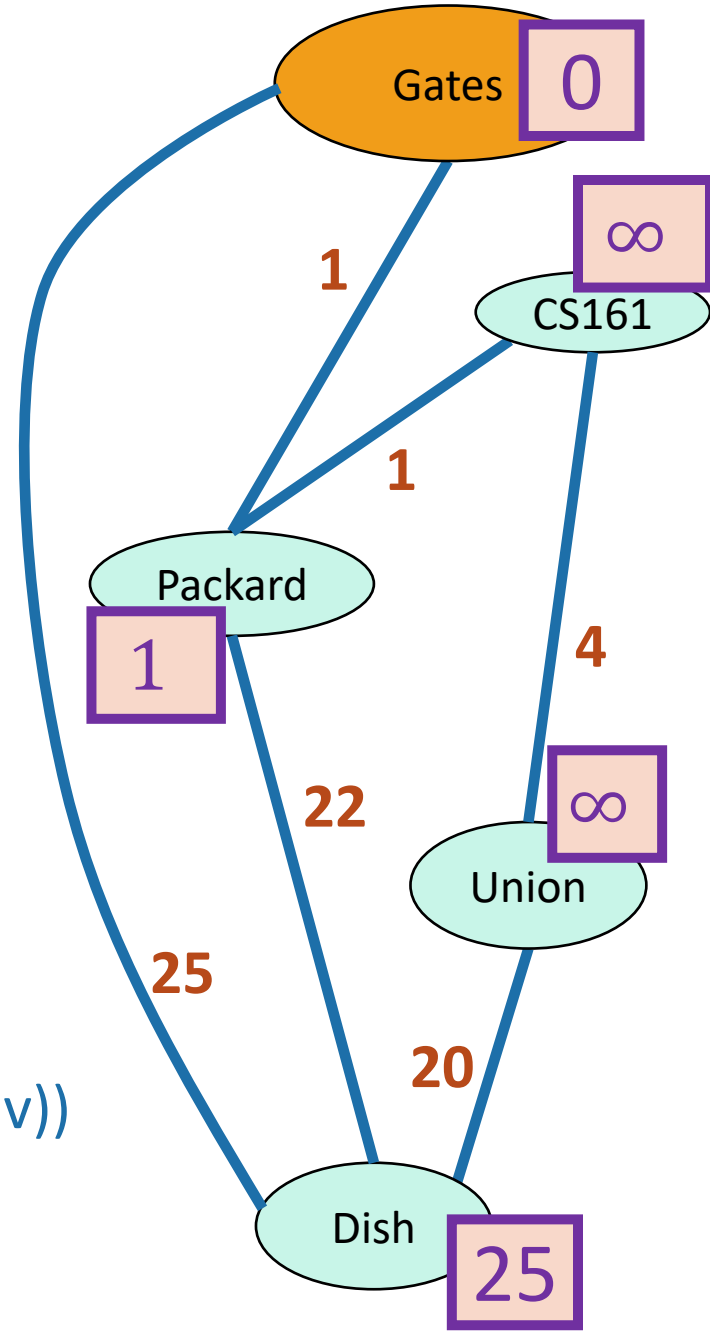
I'm sure

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



Current node u

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



Dijkstra by example

How far is a node from Gates?

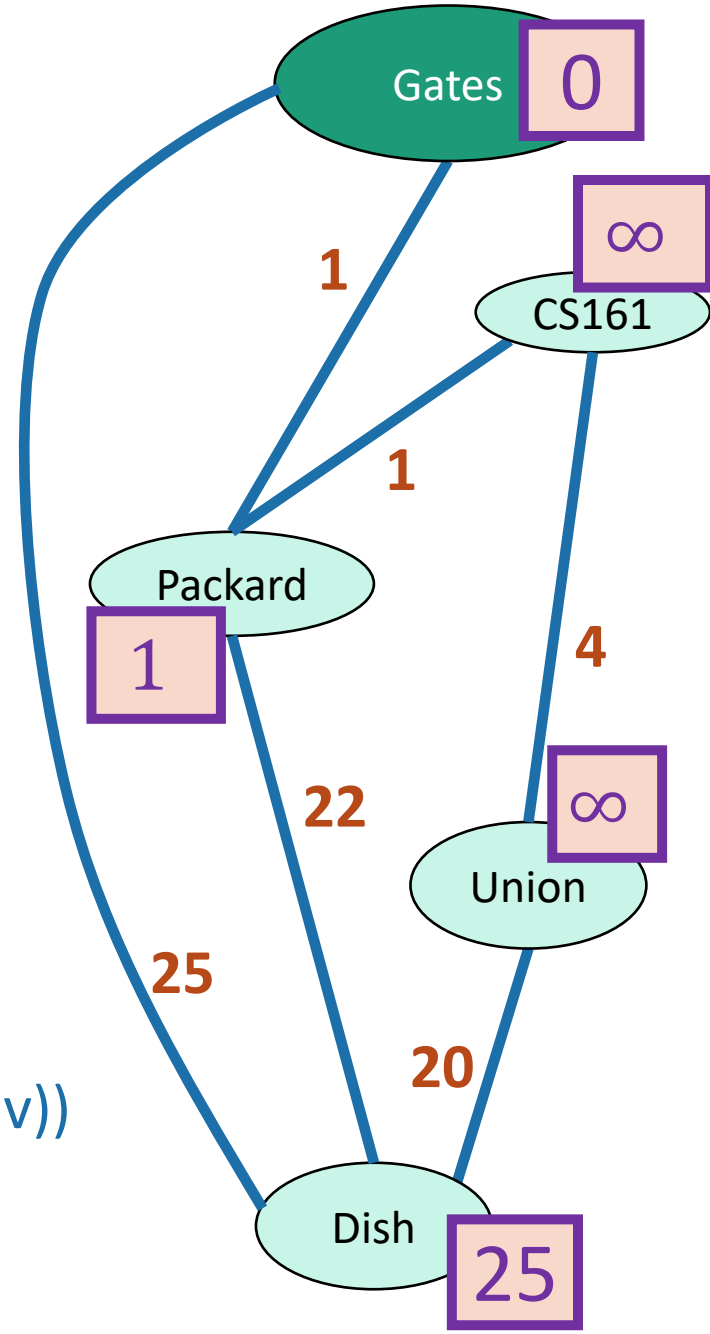
I'm not sure yet

I'm sure

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

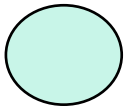
Current node u

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

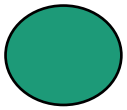


Dijkstra by example

How far is a node from Gates?

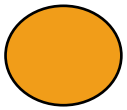


I'm not sure yet



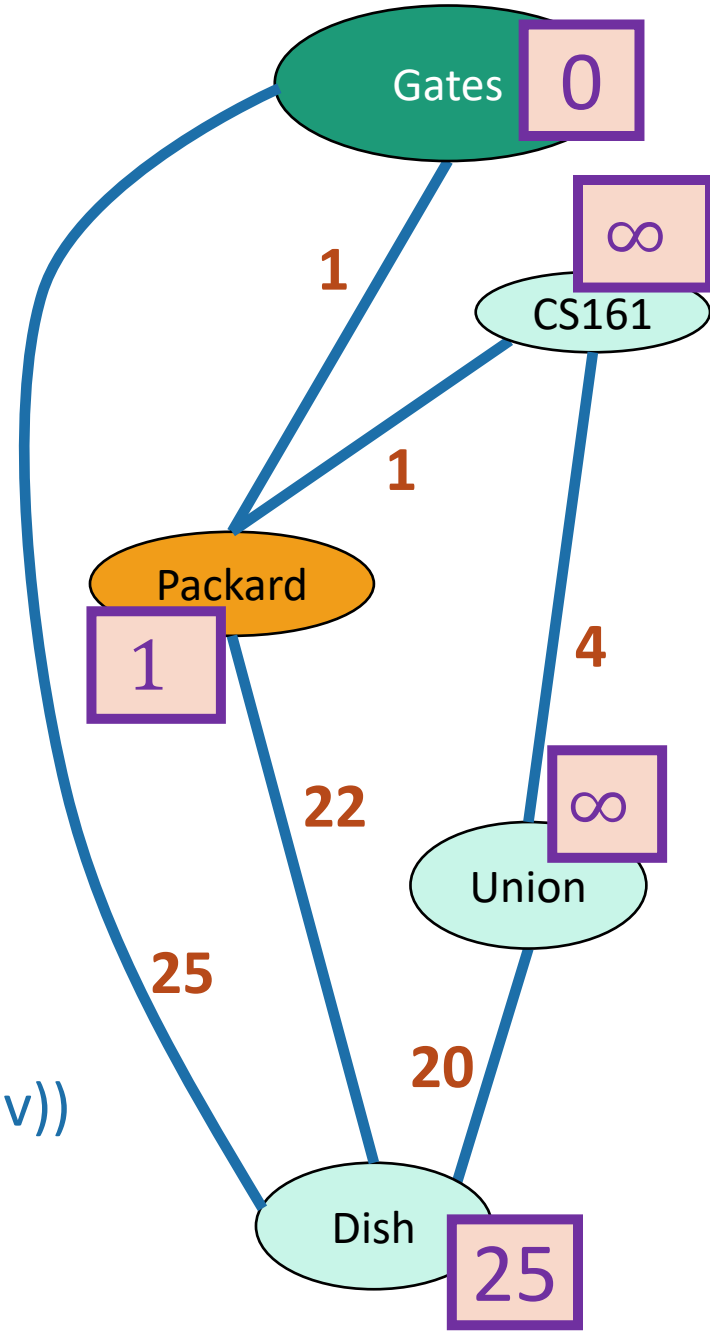
I'm sure

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



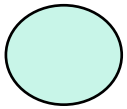
Current node u

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

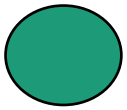


Dijkstra by example

How far is a node from Gates?

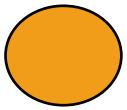


I'm not sure yet



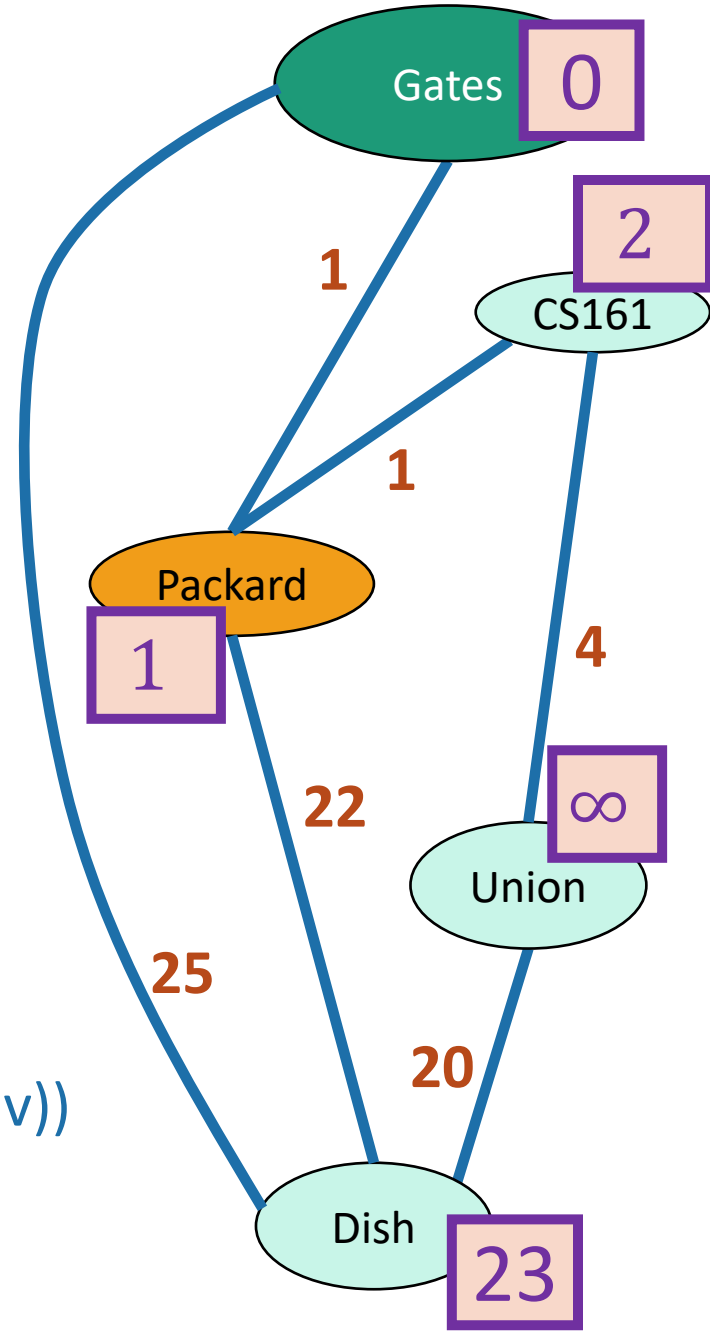
I'm sure

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



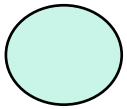
Current node u

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

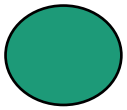


Dijkstra by example

How far is a node from Gates?

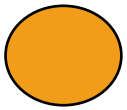


I'm not sure yet



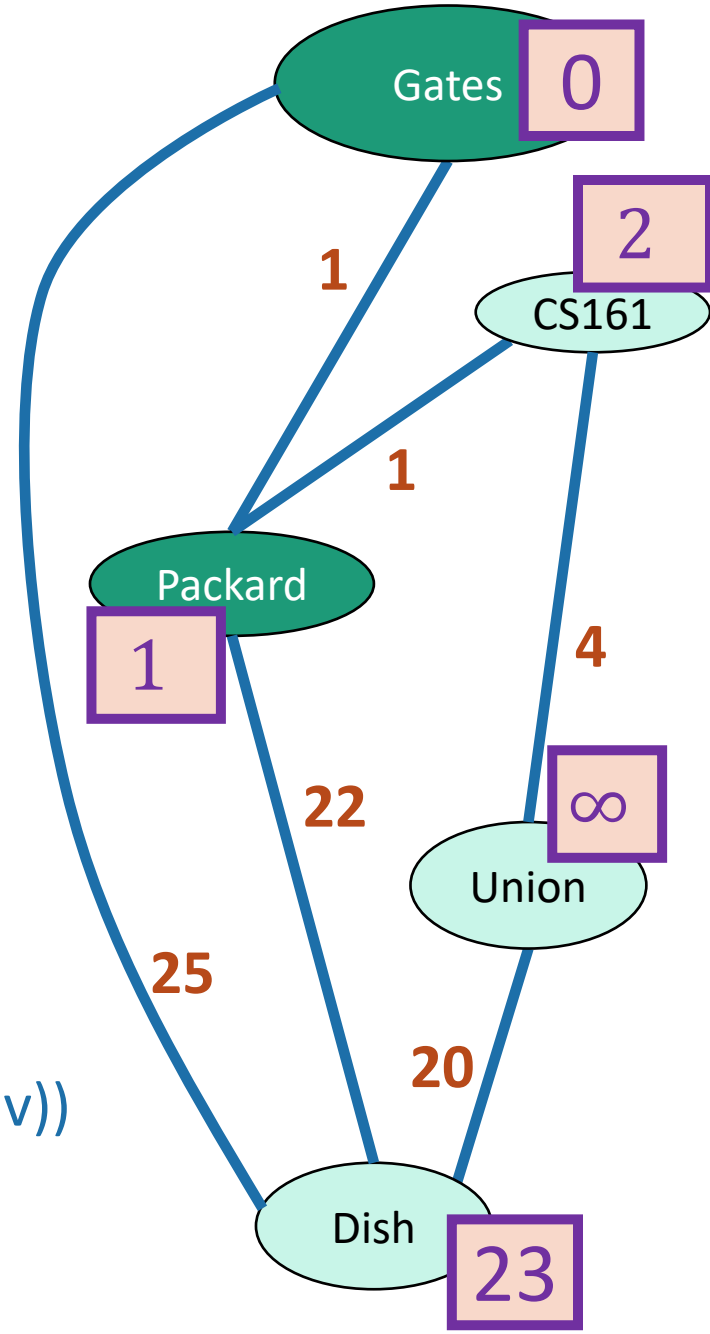
I'm sure

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



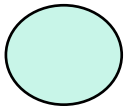
Current node u

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

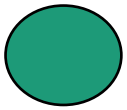


Dijkstra by example

How far is a node from Gates?

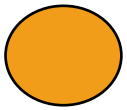


I'm not sure yet



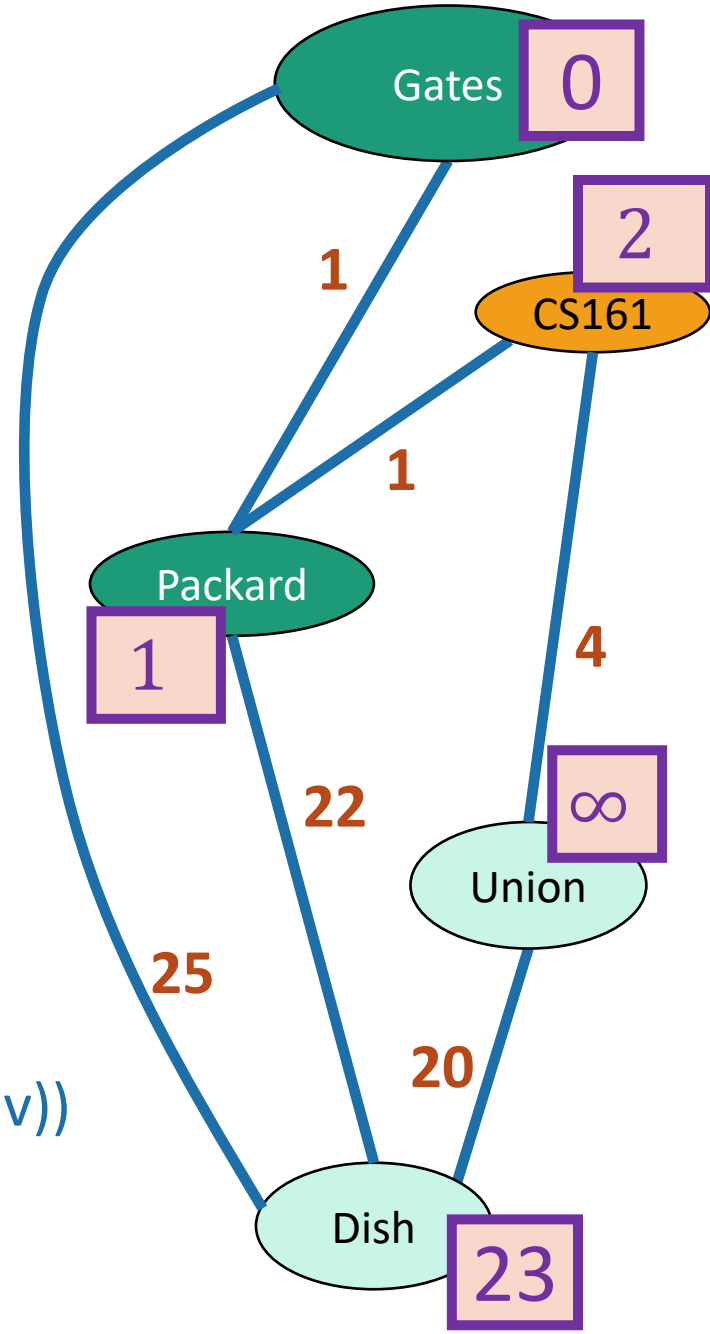
I'm sure

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



Current node u

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



Dijkstra by example

How far is a node from Gates?

I'm not sure yet

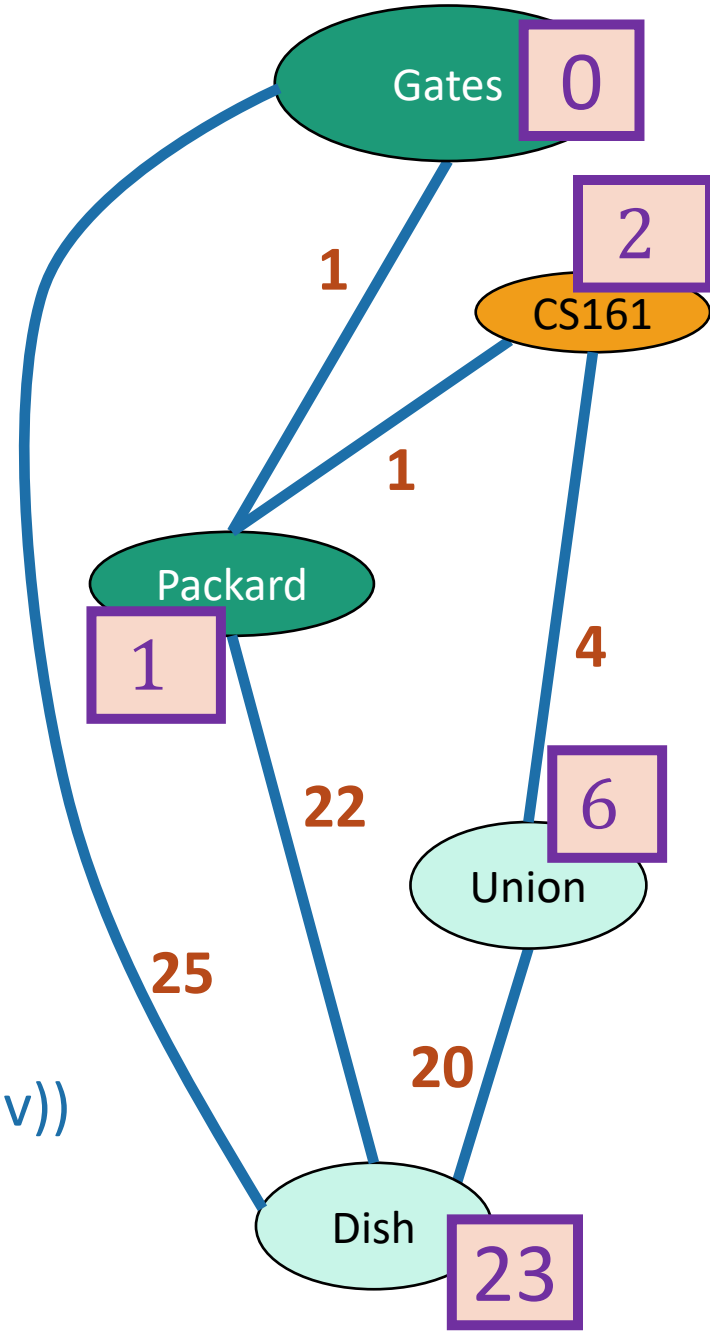
I'm sure

x

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

Current node u

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



Dijkstra by example

How far is a node from Gates?

I'm not sure yet

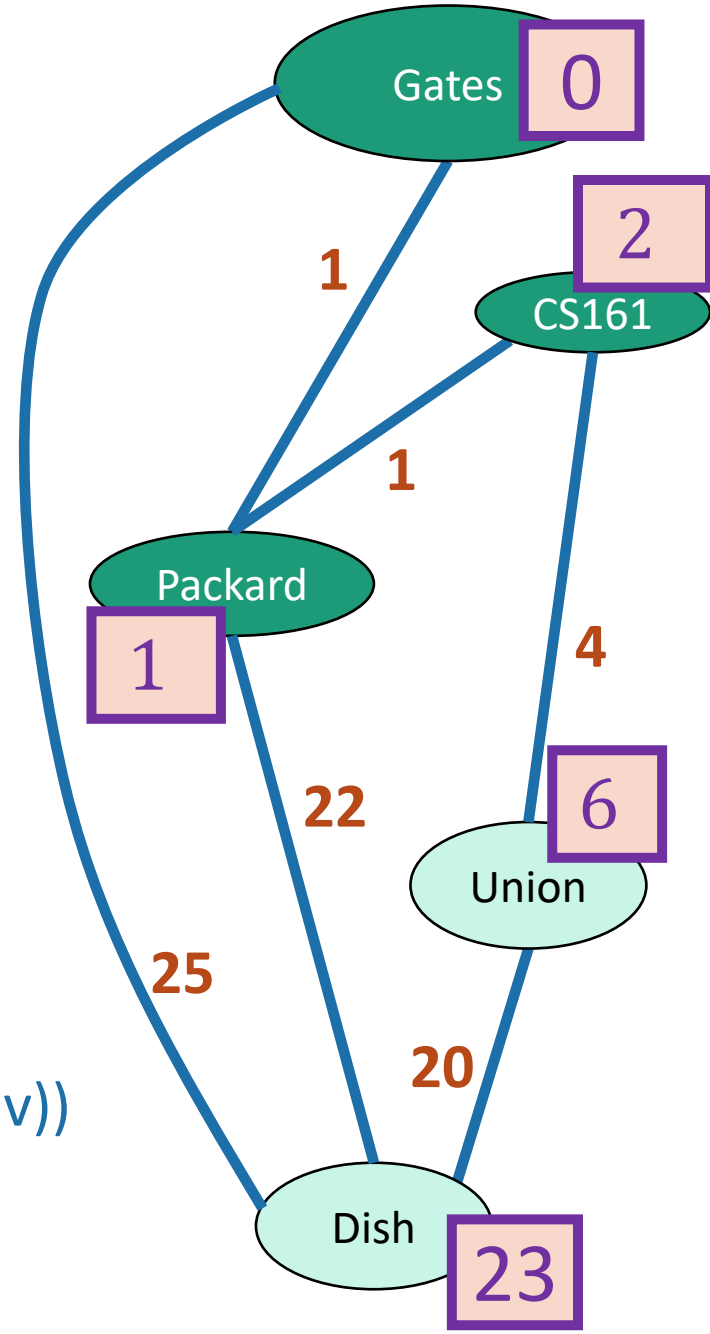
I'm sure

x

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

Current node u

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



Dijkstra by example

How far is a node from Gates?

I'm not sure yet

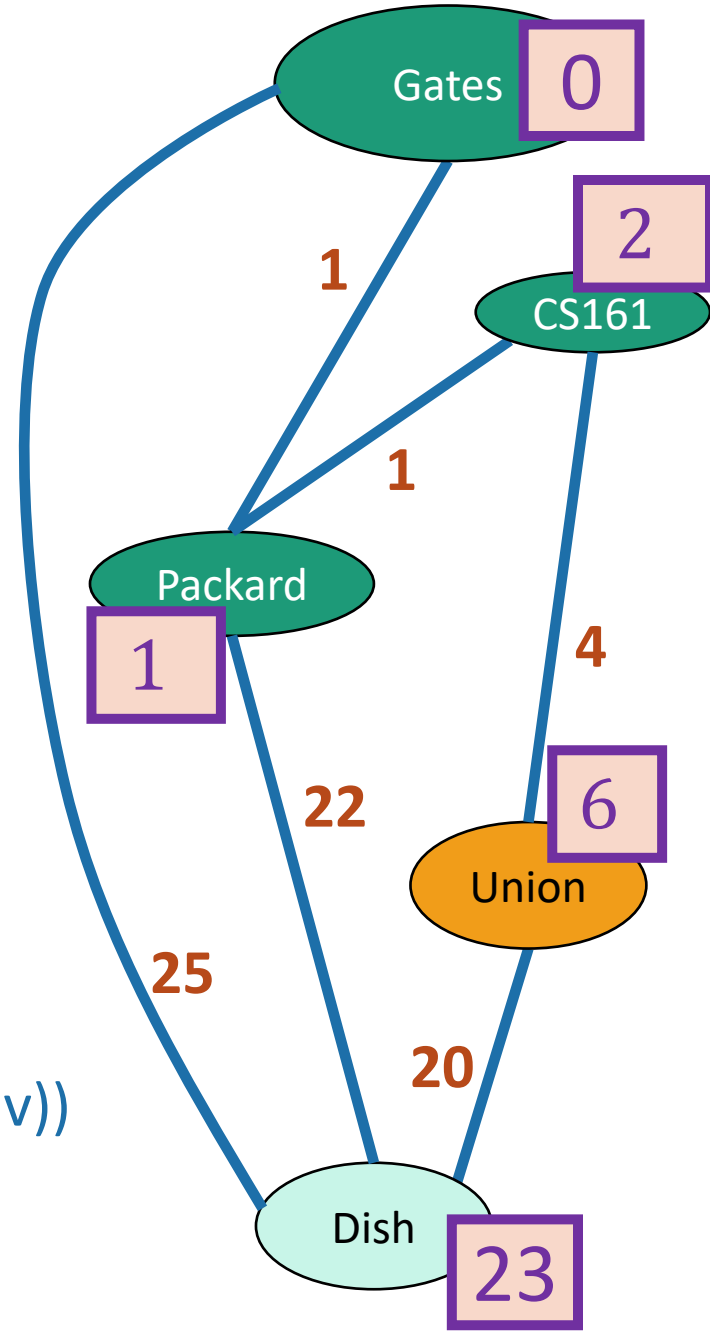
I'm sure

x

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

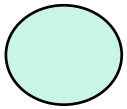
Current node u

- 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

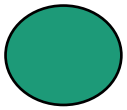


Dijkstra by example

How far is a node from Gates?

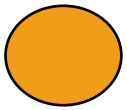


I'm not sure yet



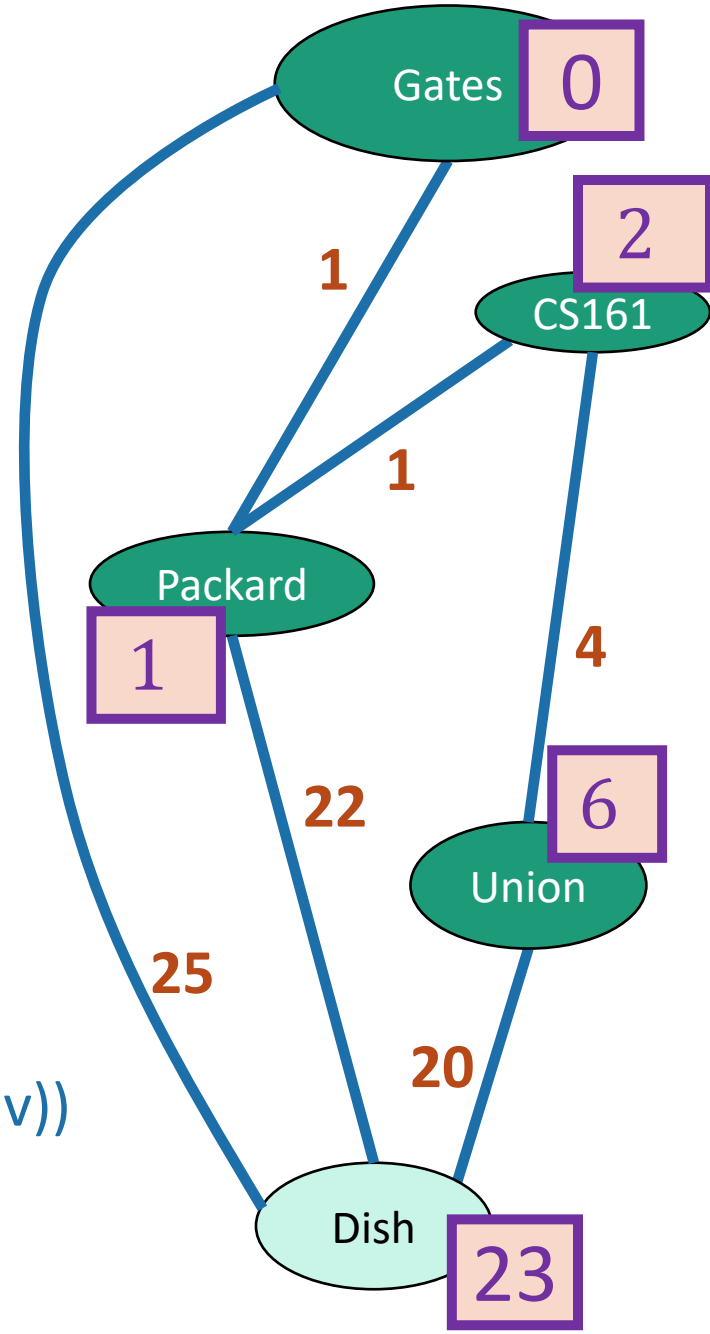
I'm sure

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



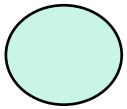
Current node u

- 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

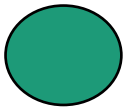


Dijkstra by example

How far is a node from Gates?

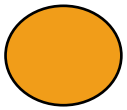


I'm not sure yet



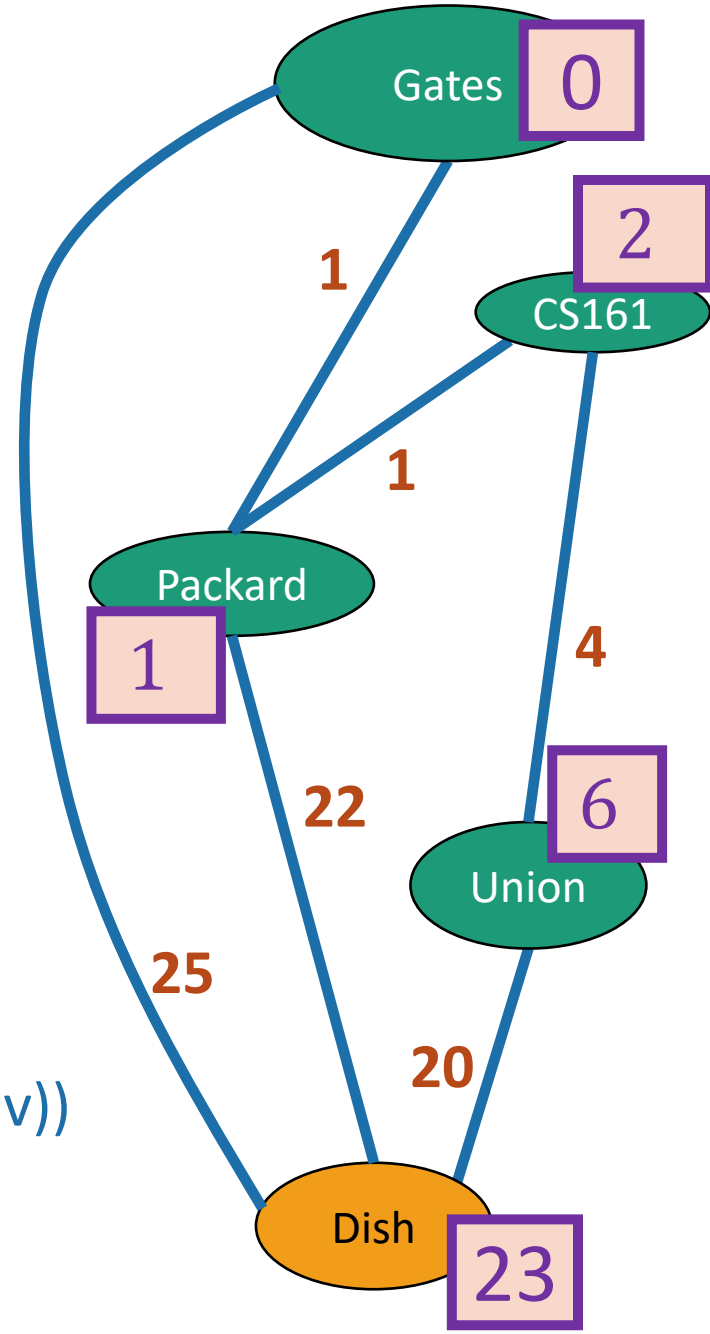
I'm sure

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?

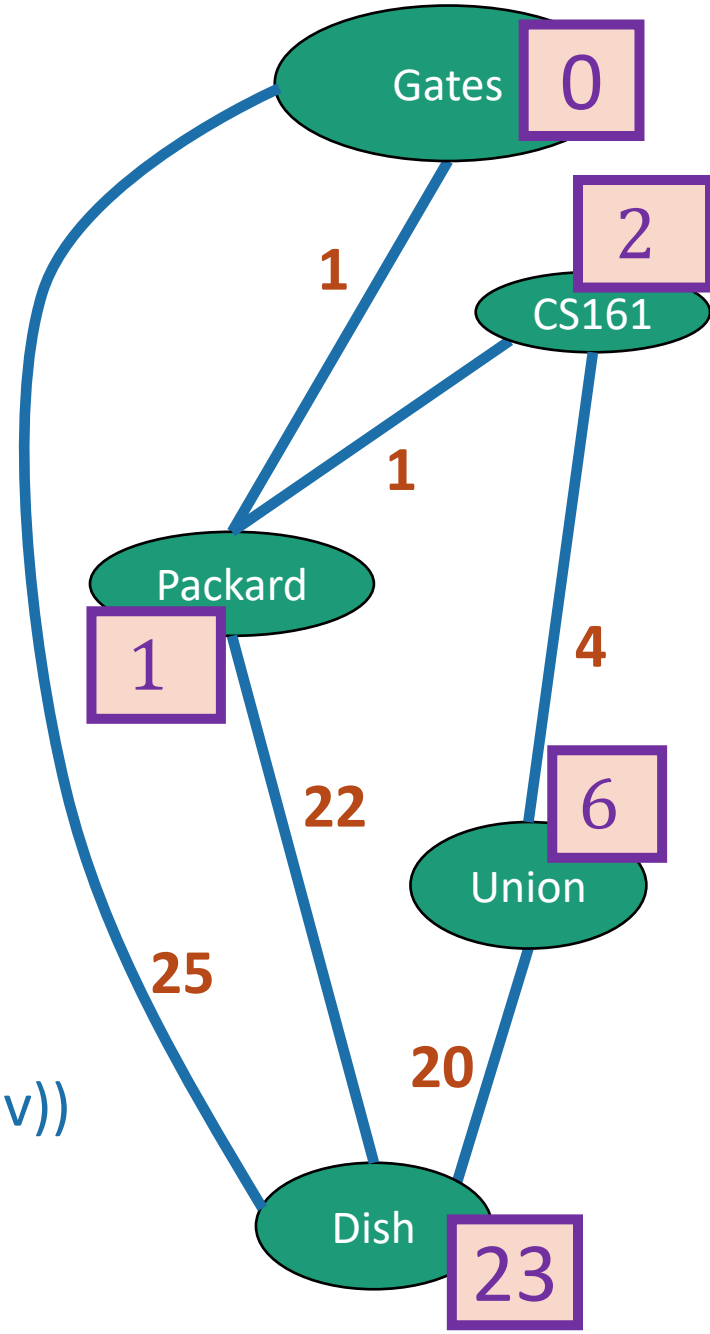
I'm not sure yet

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'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]

Running time?

Dijkstra(G,s):

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

- n iterations (one per vertex)
- How long does one iteration take?

Depends on how we implement it…

We need a data structure that:

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

Just the inner loop:

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

Total running time is big-oh of:

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

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

If we use an array

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

- Running time of Dijkstra

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

  =O(n^2) + O(m)

  =O(n^2)

If we use a red-black tree

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

- Running time of Dijkstra

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

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

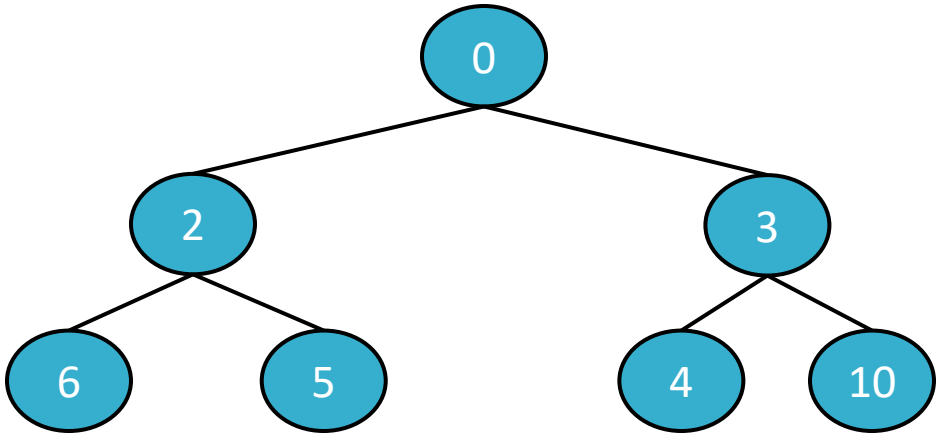
  $= O((n + m)\log(n))$

Better than an array if the graph is sparse!

aka if m is much smaller than n^2

Heaps support these operations

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



- A **heap** is a tree-based data structure that has the property that every node has a smaller key than its children.

Many heap implementations

Nice chart on Wikipedia:

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

Say we use a **Fibonacci Heap**

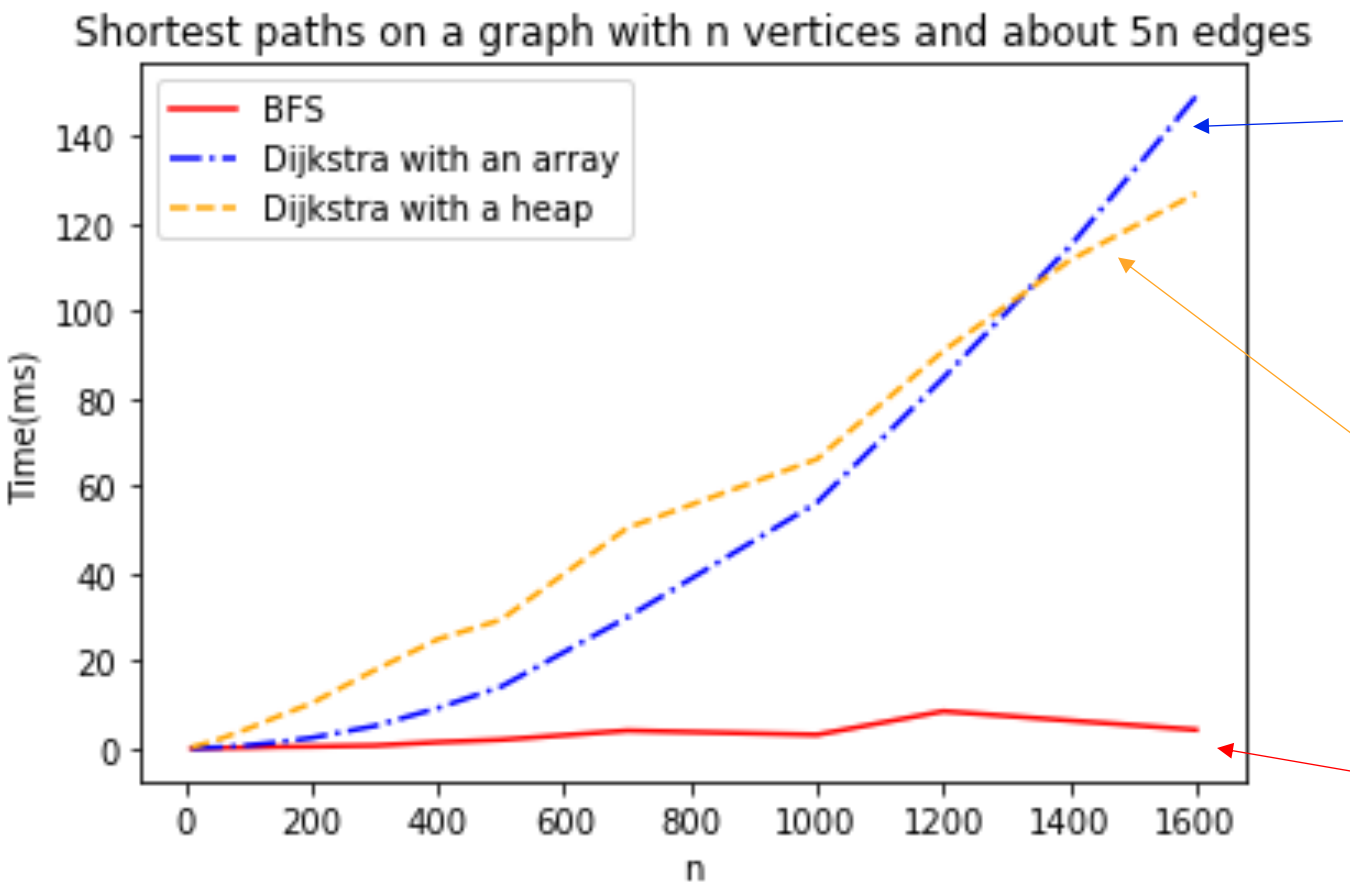
- T(findMin) = O(1) (amortized time*)
- T(removeMin) = O(log(n)) (amortized time*)
- T(updateKey) = O(1) (amortized time*)

- Running time of Dijkstra
  = O(n(T(`findMin`) + T(`removeMin`)) + m T(`updateKey`))
  = O(nlog(n) + m) (amortized time)

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

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

In practice



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

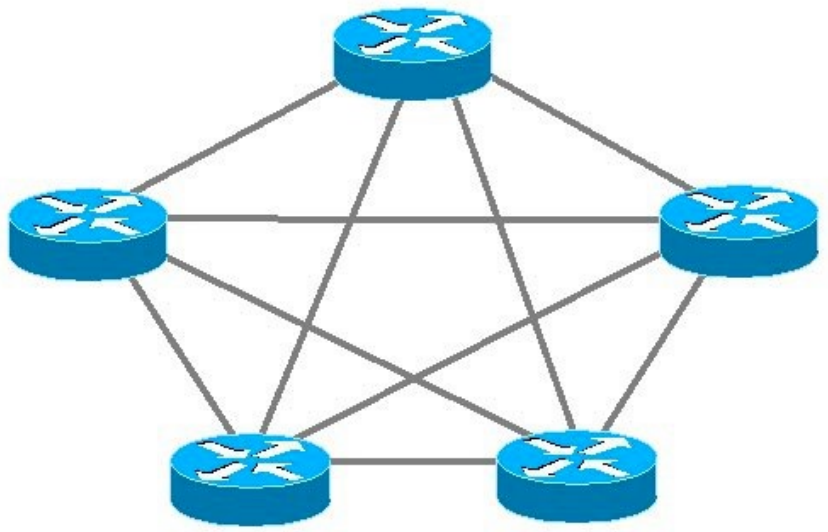
Dijkstra using a heap looks a bit more linear (actually nlog(n))

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

Dijkstra is used in practice

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

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

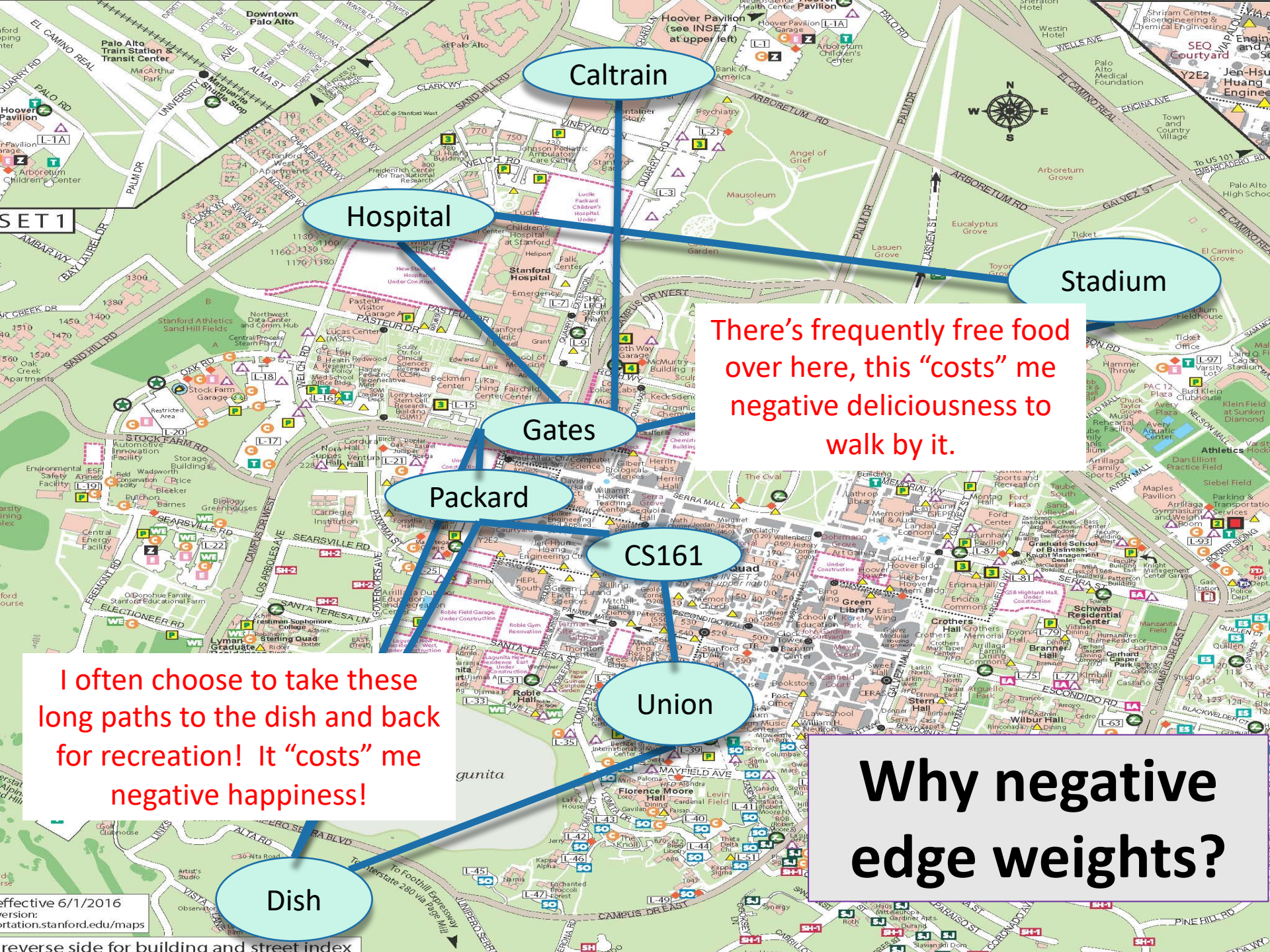


Dijkstra Drawbacks

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

Bellman-Ford algorithm

- (-) Slower than Dijkstra's algorithm
- (+) Can handle negative edge weights.
- (+) Allows for some flexibility if the weights change.
  - We'll see what this means later

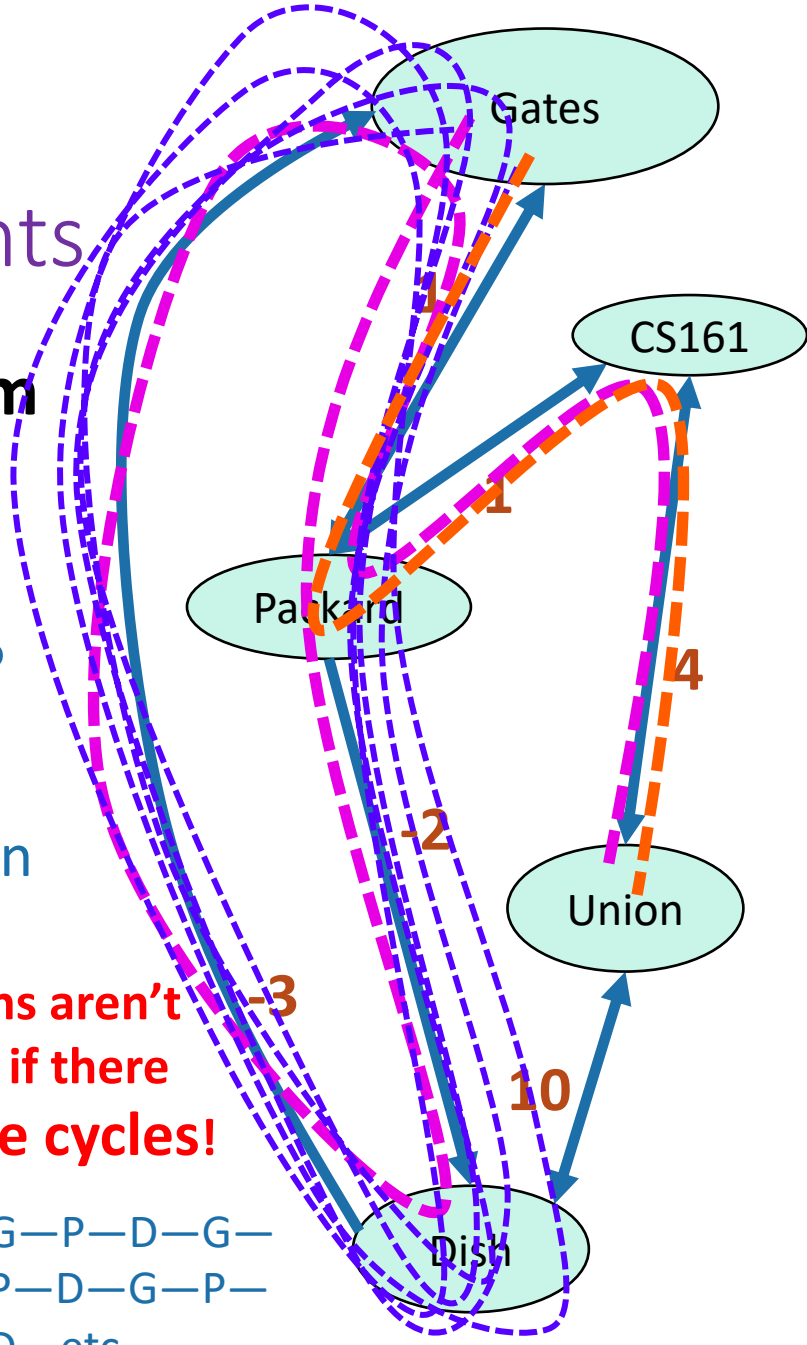

Caltrain
Hospital
Stadium
Gates
Packard
CS161
Union
Dish
There's frequently free food over here, this "costs" me negative deliciousness to walk by it.
I often choose to take these long paths to the dish and back for recreation! It "costs" me negative happiness!
Why negative edge weights?

One problem
with negative edge weights

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

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

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



Bellman-Ford algorithm

Bellman-Ford(G,s):

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

Instead of picking u cleverly, just update for all of the u's.

Compare to Dijkstra:

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

For pedagogical reasons

which we will see next week

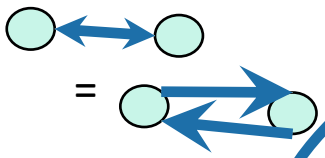
- We are actually going to change this to be dumber.
- Keep n arrays: $d^{(0)}$, $d^{(1)}$, …, $d^{(n-1)}$

Bellman-Ford*(G,s):

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

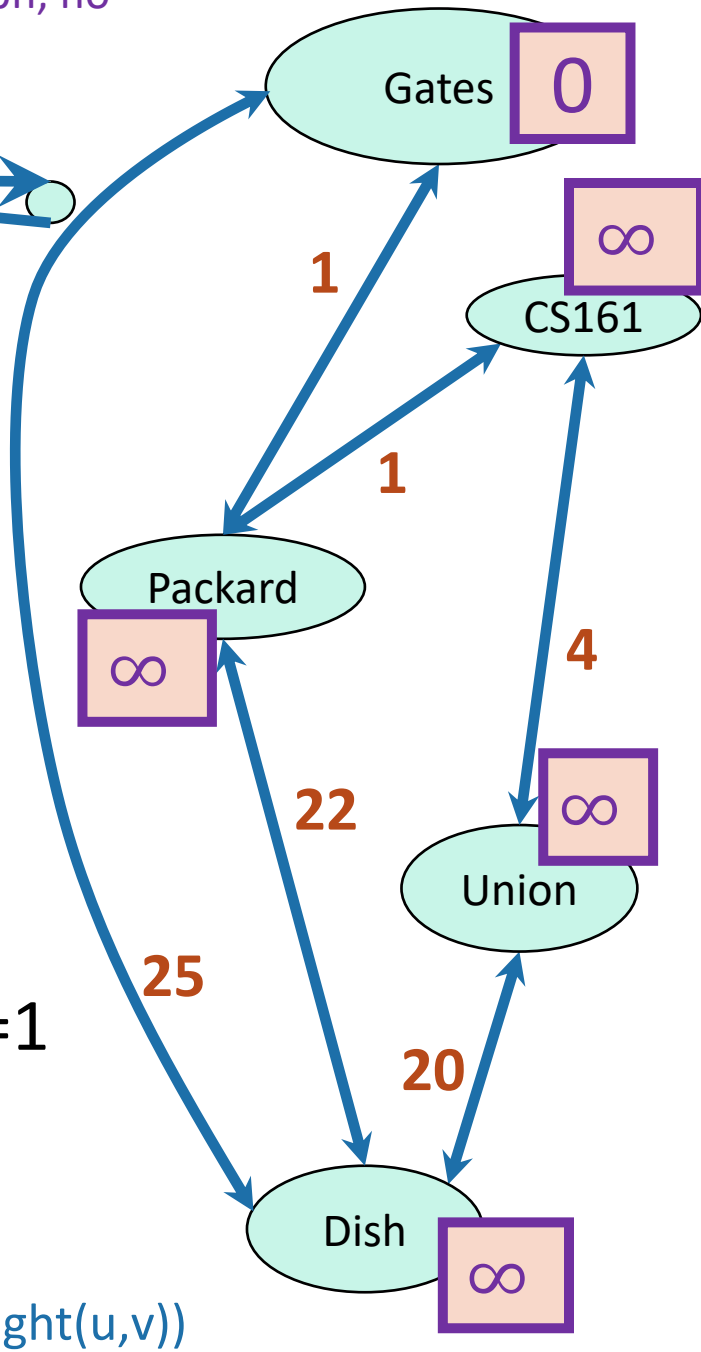
Bellman-Ford

Start with the same graph, no negative weights.



How far is a node from Gates?

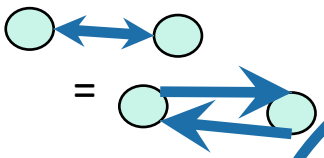
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$					
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					



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

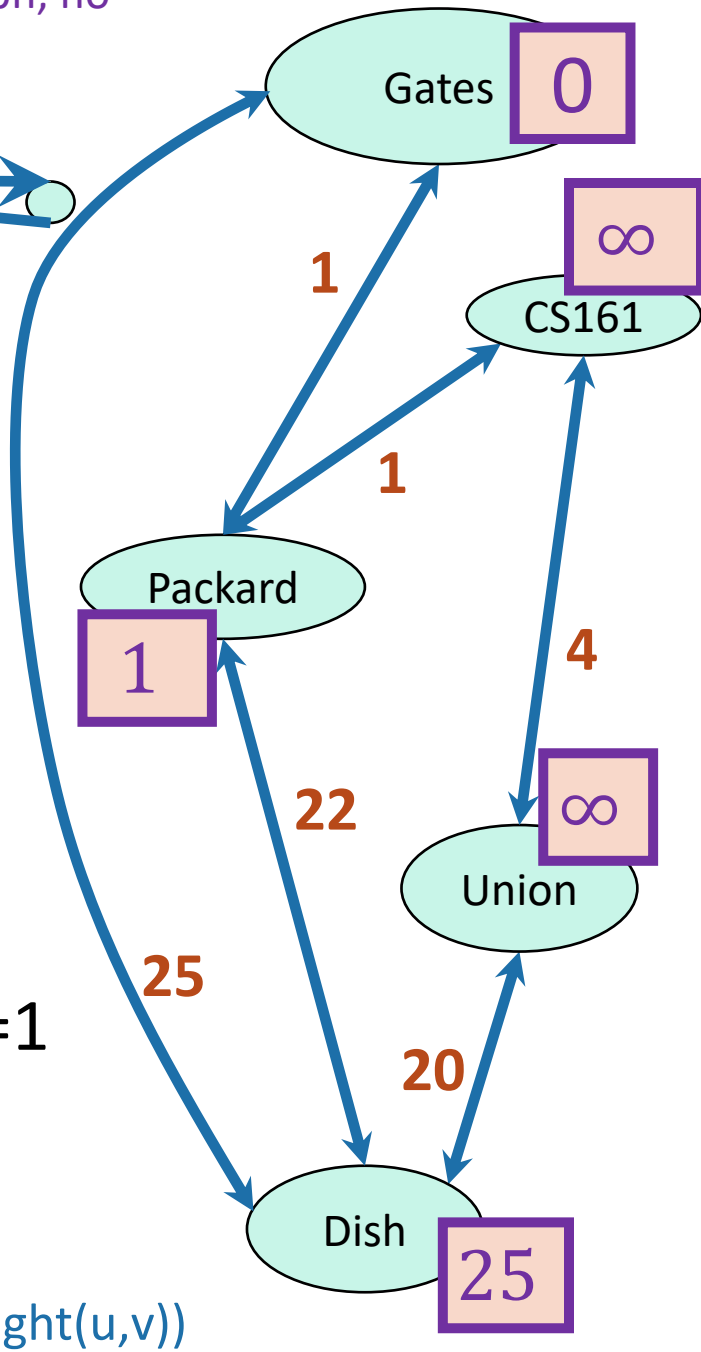
Bellman-Ford

Start with the same graph, no negative weights.



How far is a node from Gates?

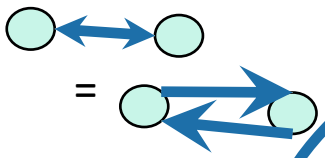
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					



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

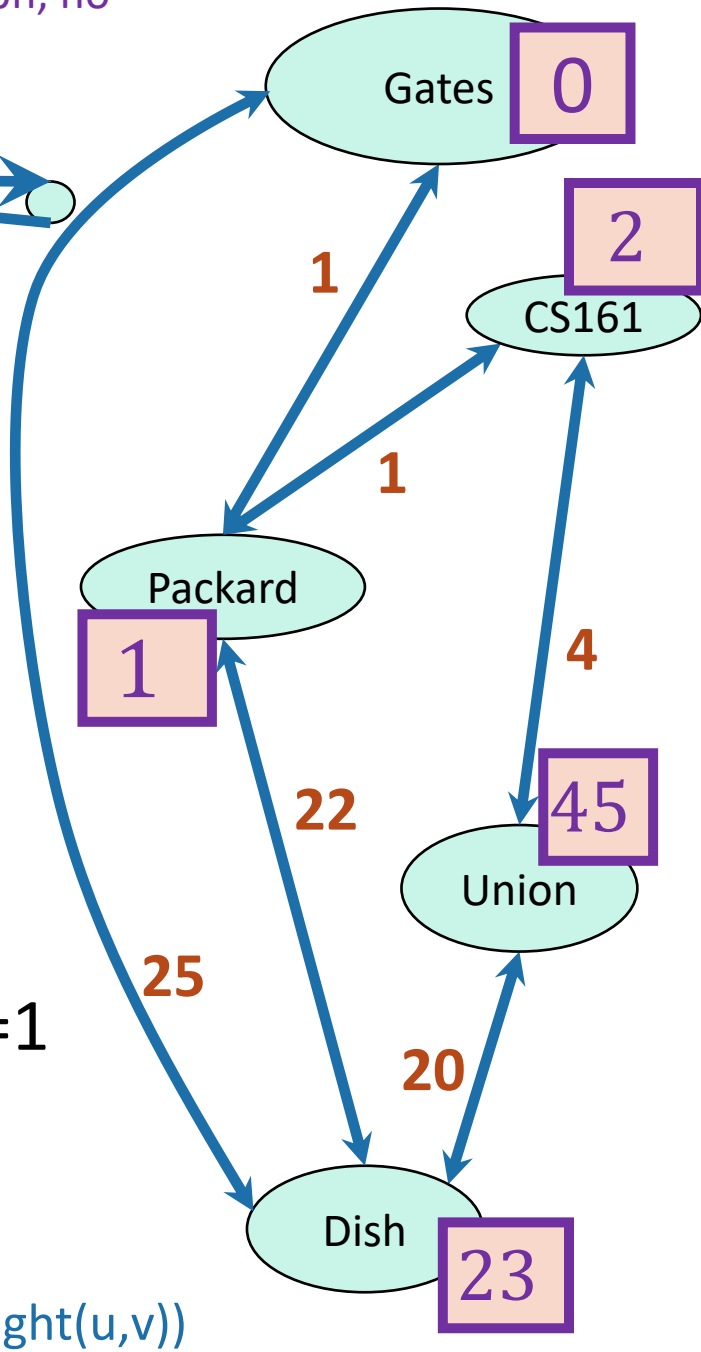
Bellman-Ford

Start with the same graph, no negative weights.



How far is a node from Gates?

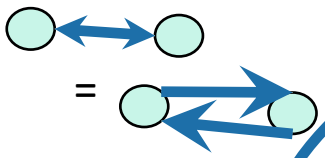
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$					
$d^{(4)}$					



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

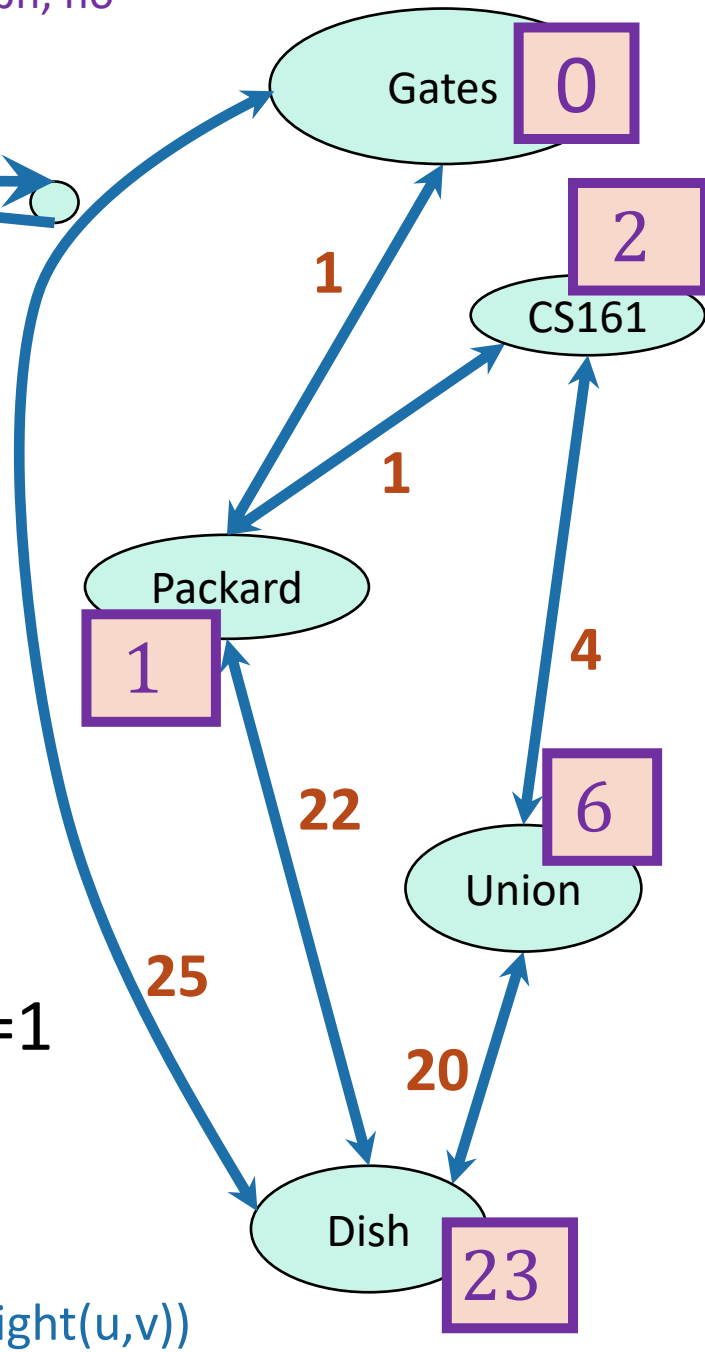
Bellman-Ford

Start with the same graph, no negative weights.



How far is a node from Gates?

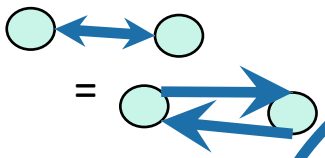
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$					



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

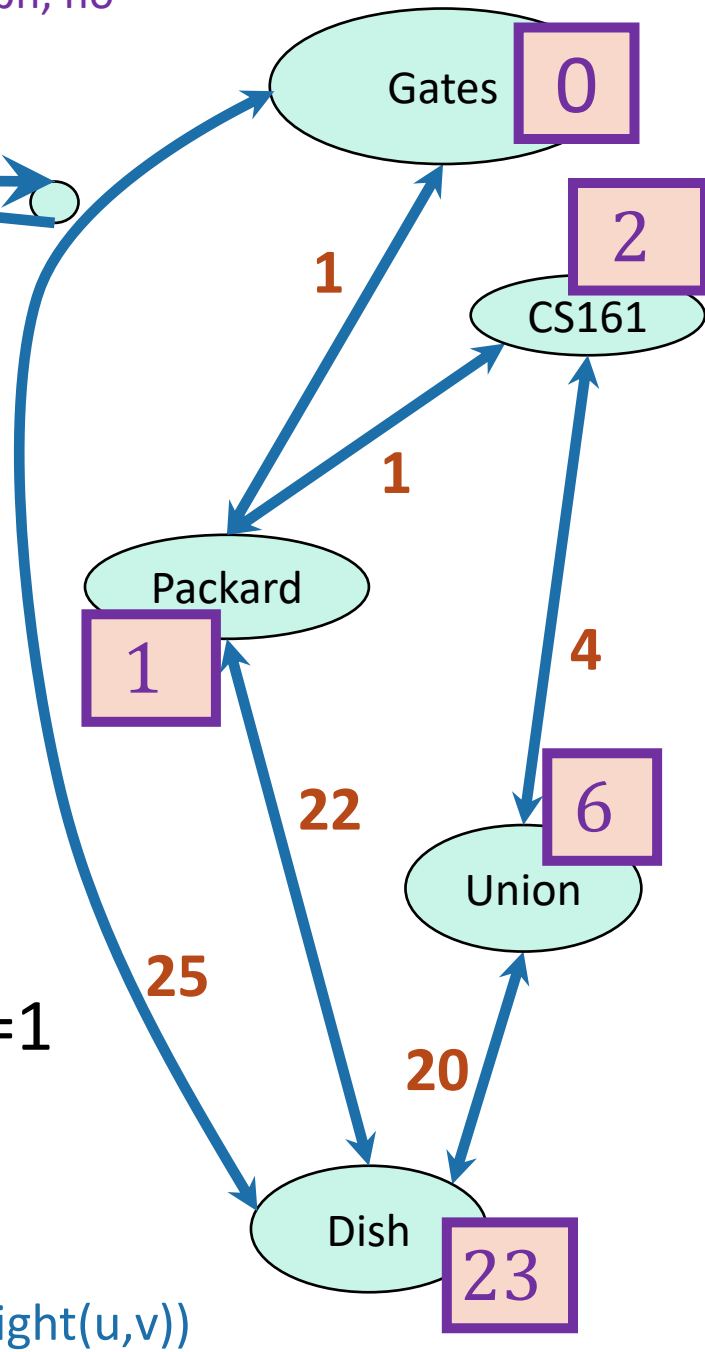
Bellman-Ford

Start with the same graph, no negative weights.



How far is a node from Gates?

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



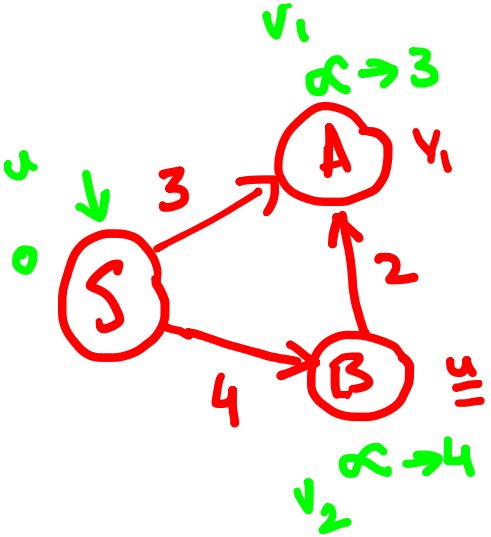
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v))$

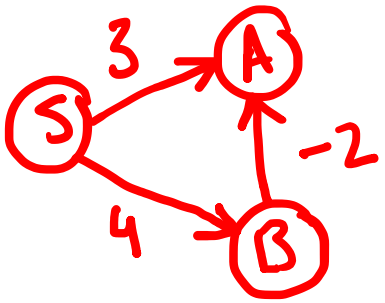
	S	A	B
S	0	∞	∞
	0	3	4

* ↳ $d[v] = \min(d[v], d[u] + w(u,v))$

∞ , 0+3

3 , 4+2





This seems much slower than Dijkstra

- And it is:

Running time O(mn)

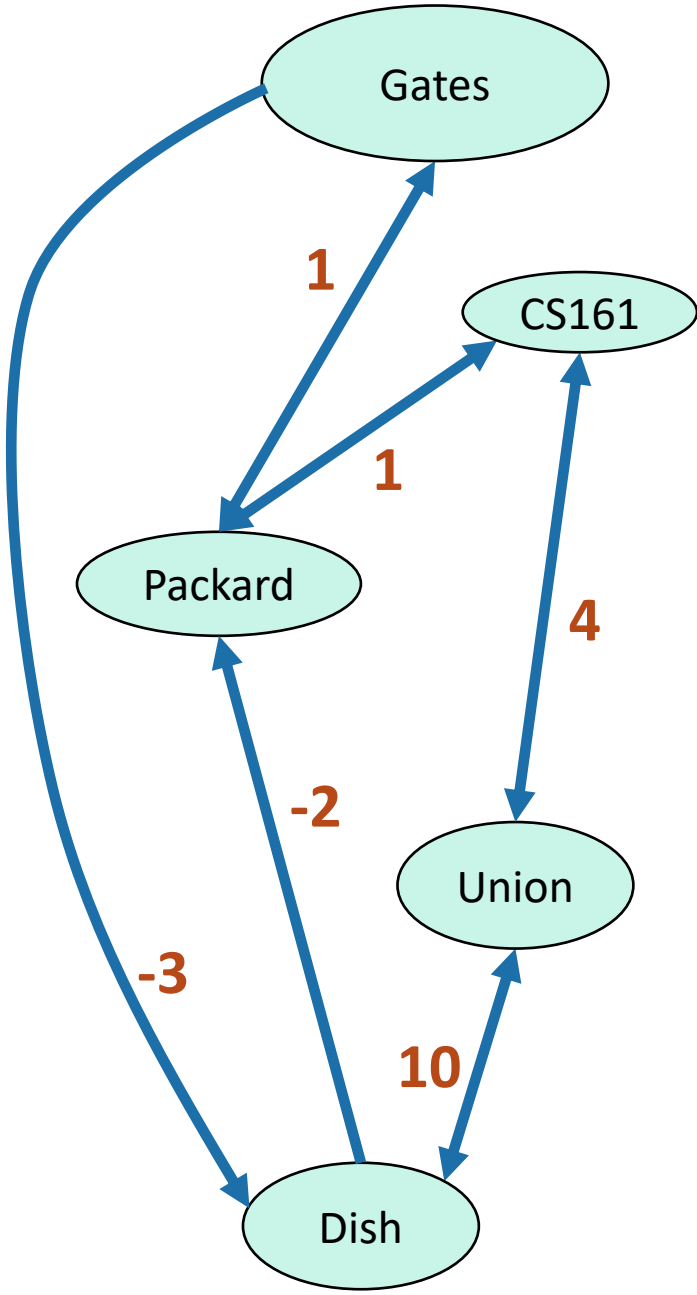
- However, it's also more flexible in a few ways.
  - Can handle negative edges
  - If we keep on doing these iterations, then changes in the network will propagate through.

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

Negative edge weights

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

This is not looking good!



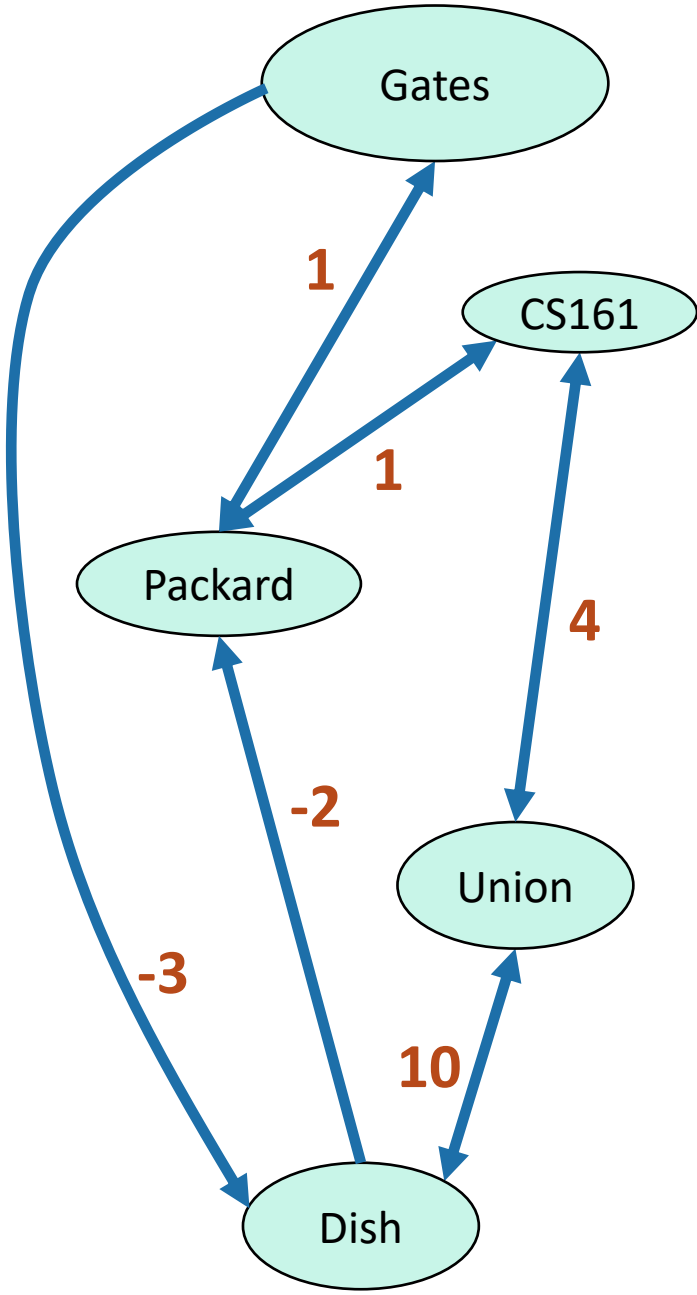
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v))$

Negative edge weights

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

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

$d^{(5)}$	-4	**-9**	-4	**3**	-7



Some stuff changed!

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

How Bellman-Ford deals with negative cycles

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

Bellman-Ford algorithm

Bellman-Ford*(G,s):

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

What have we learned?

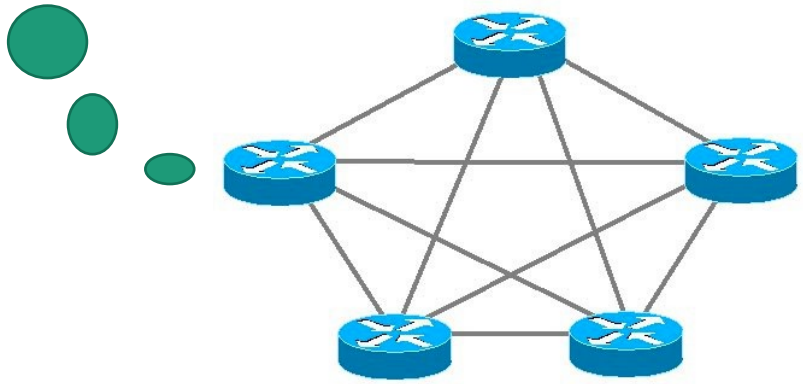
- The Bellman-Ford algorithm:
  - Finds shortest paths in weighted graphs with negative edge weights
  - runs in time O(nm) on a graph G with n vertices and m edges.
- If there are no negative cycles in G:
  - the BF algorithm terminates with $d^{(n-1)}[v] = d(s,v)$.
- If there are negative cycles in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

Bellman-Ford is also used in practice.

- eg, Routing Information Protocol (RIP) uses something like Bellman-Ford.
  - Older protocol, not used as much anymore.

- Each router keeps a **table** of distances to every other router.
- Periodically we do a Bellman-Ford update.
- This means that if there are changes in the network, this will propagate. (maybe slowly…)

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



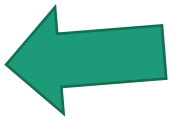
Recap: shortest paths

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

Bellman-Ford is an example of…

Dynamic Programming!

Today:



- Example of Dynamic programming:
  - Fibonacci numbers.
  - (And Bellman-Ford)
- What is dynamic programming, exactly?
  - And why is it called “dynamic programming”?
- Another example: Floyd-Warshall algorithm
  - An “all-pairs” shortest path algorithm

How not to compute Fibonacci Numbers

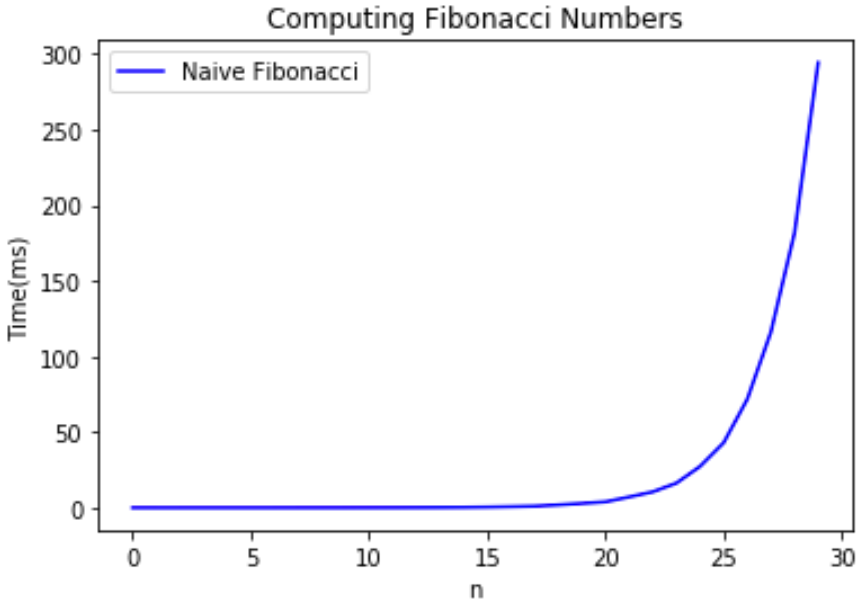
- Definition:
  - F(n) = F(n-1) + F(n-2), with F(0) = F(1) = 1.
  - The first several are:

    1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,…

- Question:
  - Given n, what is F(n)?

Candidate algorithm

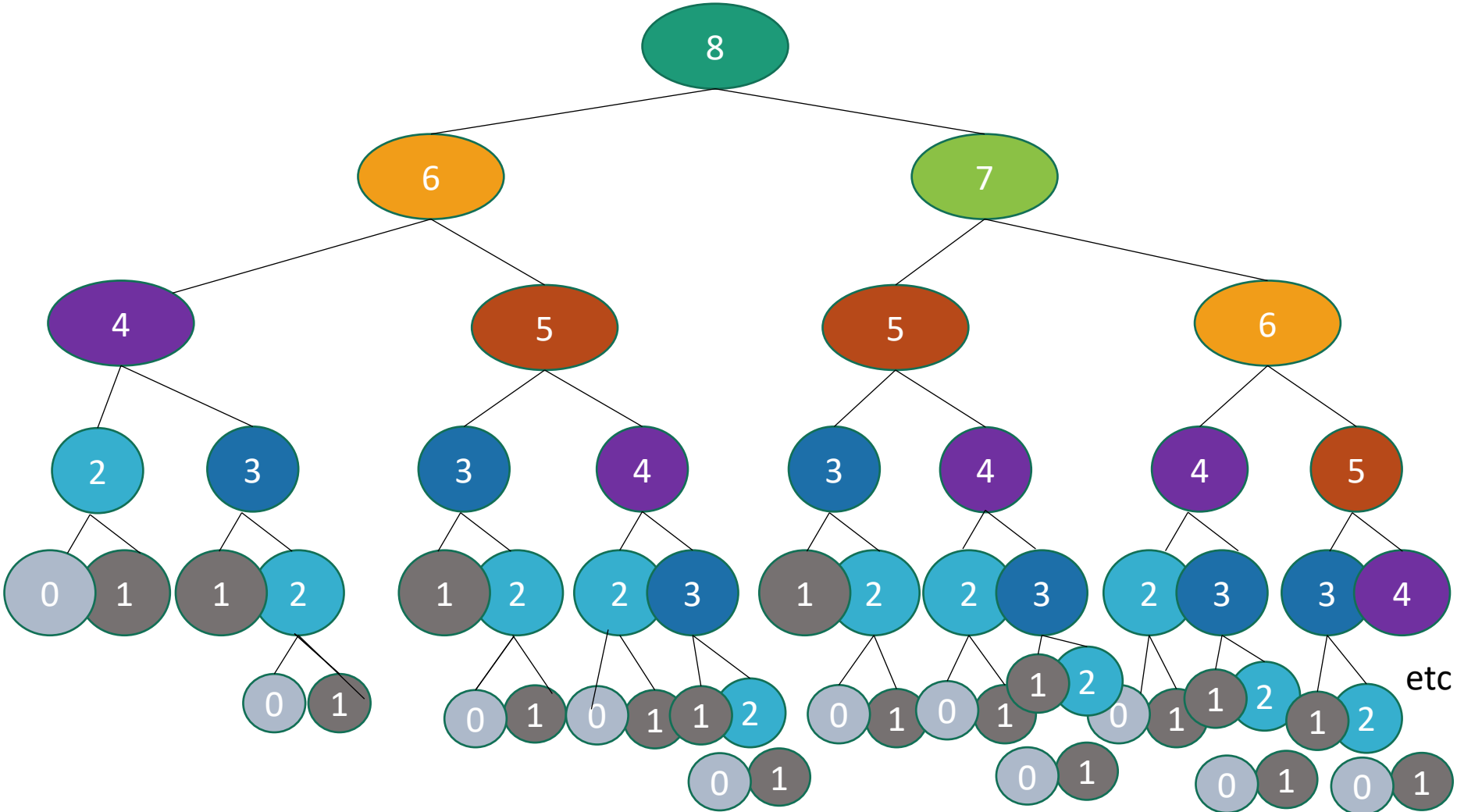
```
def Fibonacci(n):
    if n == 0 or n == 1:
        return 1
    return Fibonacci(n-1) + Fibonacci(n-2)
```



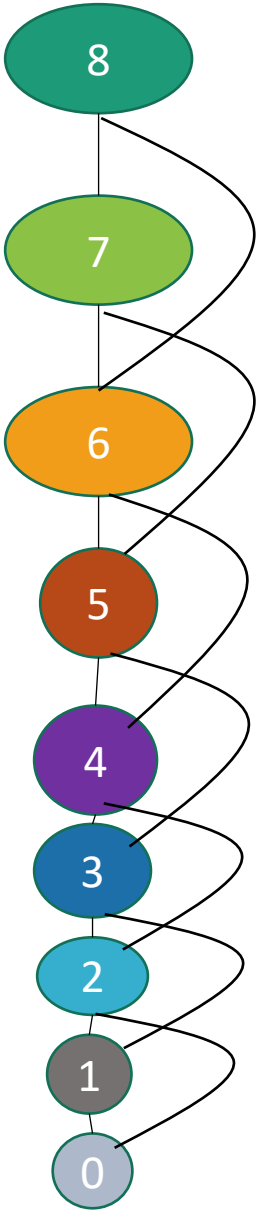
What's going on?

Consider Fib(8)

That's a lot of repeated computation!

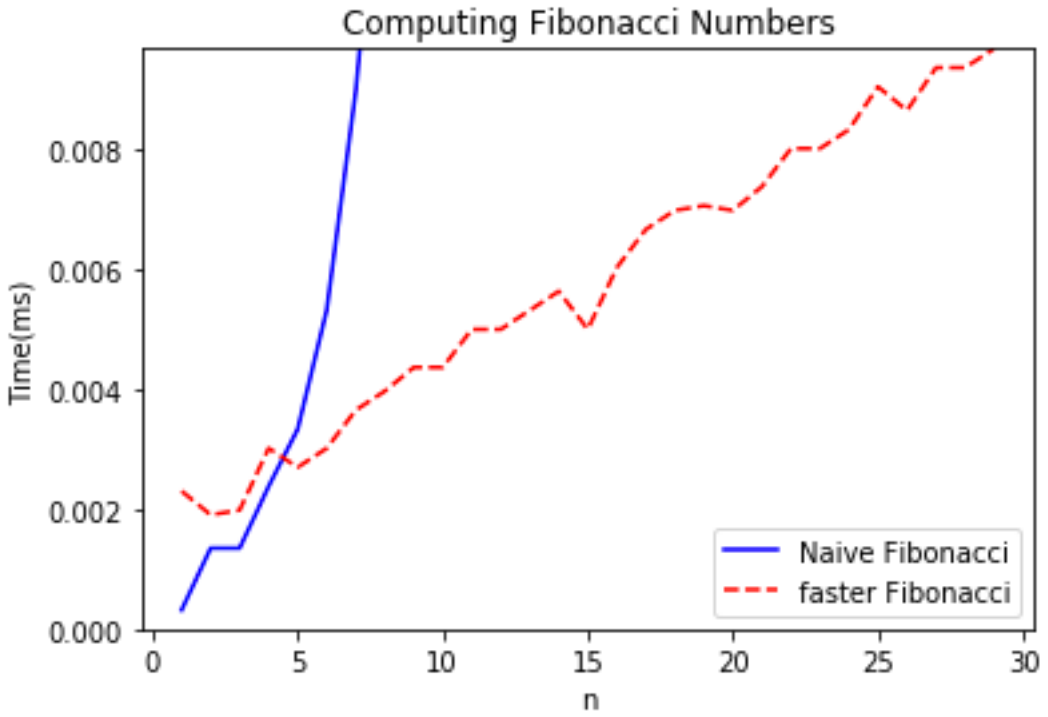


Maybe this would be better:




```
def fasterFibonacci(n):
    • F = [1, 1, None, None, …, None ]
        • \\ F has length n
    • for i = 2, …, n:
        • F[i] = F[i-1] + F[i-2]
    • return F[n]
```

Much better running time!



This was an example of…

What is *dynamic programming*?

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.
- Usually it is for solving **optimization problems**
  - eg, ***shortest*** path
  - (Fibonacci numbers aren't an optimization problem, but they are a good example…)

Elements of dynamic programming

1. Optimal sub-structure:

- Big problems break up into sub-problems.
  - Fibonacci: F(i) for $i \leq n$
  - Bellman-Ford: Shortest paths with at most i edges for $i \leq n$
- The solution to a problem can be expressed in terms of solutions to smaller sub-problems.
  - Fibonacci:

$$F(i+1) = F(i) + F(i-1)$$

  - Bellman-Ford:

$$d^{(i+1)}[v] \leftarrow \min\{\, d^{(i)}[v],\ \min_u \{d^{(i)}[u] + \text{weight}(u,v)\}\,\}$$

Shortest path with at most i edges from s to v

Shortest path with at most i edges from s to u.

Elements of dynamic programming

2. Overlapping sub-problems:

- The sub-problems overlap a lot.
  - Fibonacci:
    - Lots of different F[j] will use F[i].
  - Bellman-Ford:
    - Lots of different entries of $d^{(i+1)}$ will use $d^{(i)}[v]$.
  - This means that we can save time by solving a sub-problem just once and storing the answer.

Elements of dynamic programming

- Optimal substructure.
  - Optimal solutions to sub-problems are sub-solutions to the optimal solution of the original problem.
- Overlapping subproblems.
  - The subproblems show up again and again

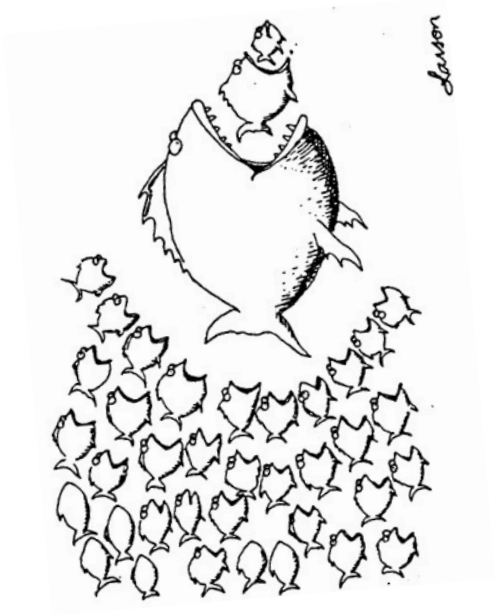
- Using these properties, we can design a ***dynamic programming*** algorithm:
  - Keep a table of solutions to the smaller problems.
  - Use the solutions in the table to solve bigger problems.
  - At the end we can use information we collected along the way to find the solution to the whole thing.

Two ways to think about and/or implement DP algorithms

- Top down
- Bottom up

Bottom up approach

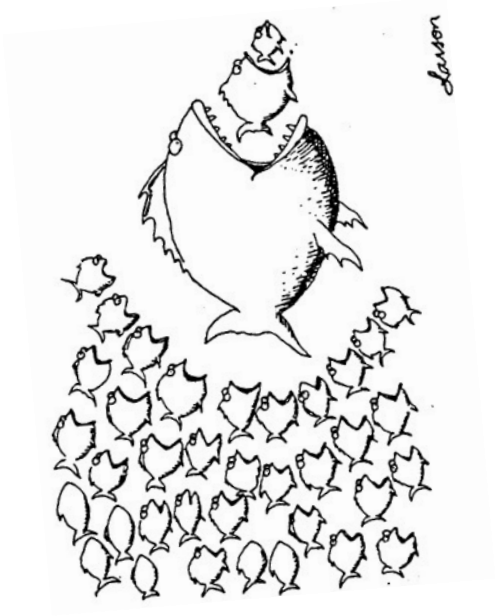
what we just saw.



- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]

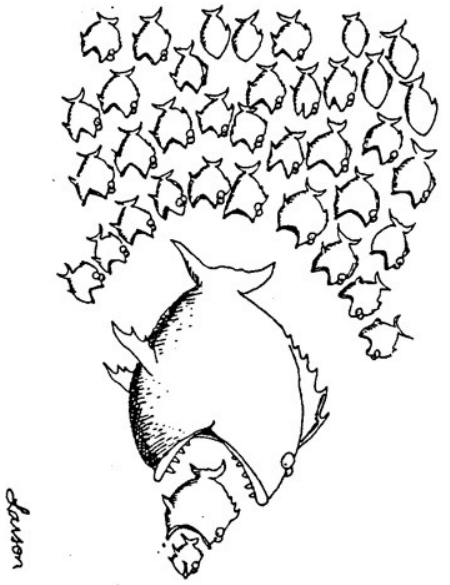
Bottom up approach

what we just saw.



- For Bellman-Ford:
- Solve the small problems first
  - fill in $d^{(0)}$
- Then bigger problems
  - fill in $d^{(1)}$
- …
- Then bigger problems
  - fill in $d^{(n-2)}$
- Then finally solve the real problem.
  - fill in $d^{(n-1)}$

Top down approach



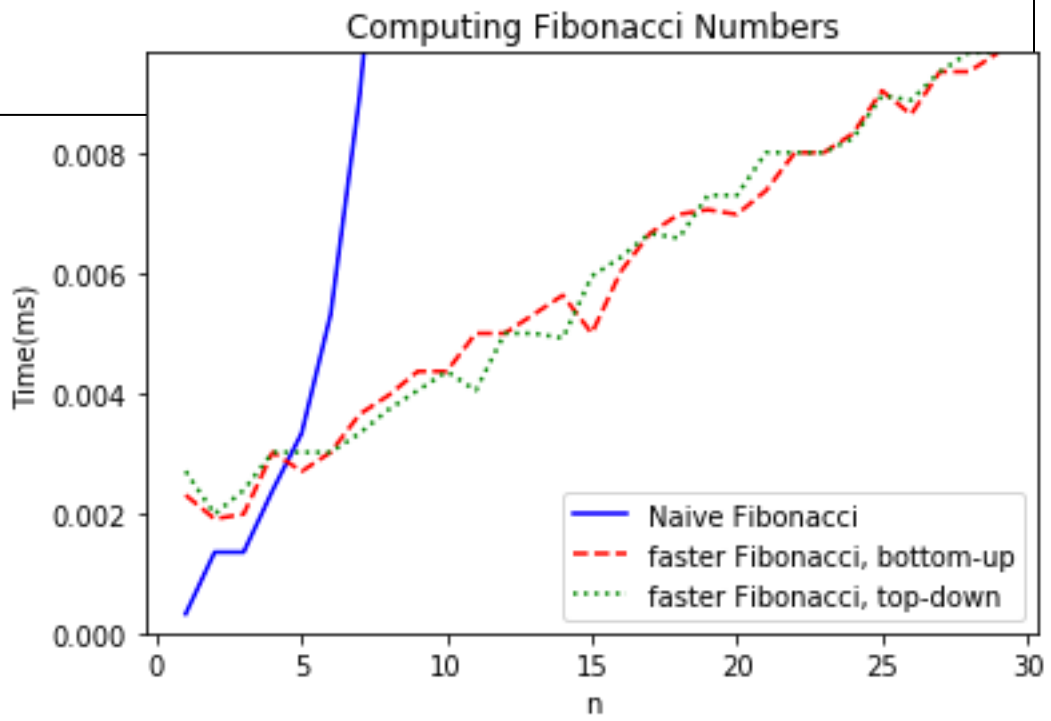
- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..

- The difference from divide and conquer:
  - **Memo-ization**
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.

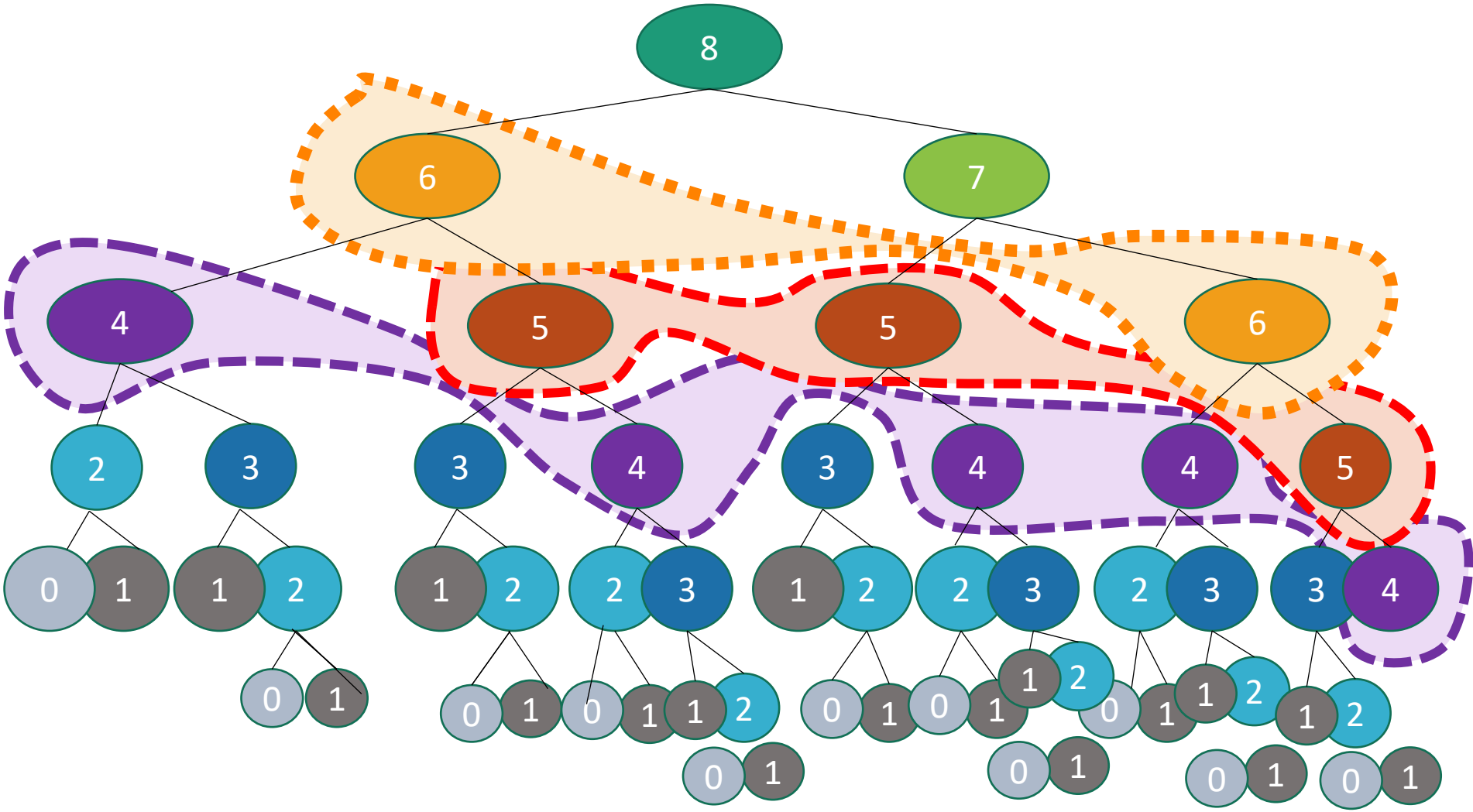
Example of top-down Fibonacci

- define a global list F = [1,1,None, None, …, None]
- **def** Fibonacci(n):
  - **if** F[n] != None:
    - **return** F[n]
  - **else:**
    - F[n] = Fibonacci(n-1) + Fibonacci(n-2)
  - **return** F[n]

Memo-ization: Keeps track (in F) of the stuff you've already done.



Memo-ization visualization



What have we learned?

- ***Dynamic programming:***
  - Paradigm in algorithm design.
  - Uses **optimal substructure**
  - Uses **overlapping subproblems**
  - Can be implemented **bottom-up** or **top-down**.
  - It's a fancy name for a pretty common-sense idea:

Why "*dynamic programming*" ?

- Programming refers to finding the optimal "program."
  - as in, a shortest route is a *plan* aka a *program*.
- Dynamic refers to the fact that it's multi-stage.
- But also it's just a fancy-sounding name.

Why "*dynamic programming*" ?

- Richard Bellman invented the name in the 1950's.
- At the time, he was working for the RAND Corporation, which was basically working for the Air Force, and government projects needed flashy names to get funded.
- From Bellman's autobiography:
  - "It's impossible to use the word, dynamic, in the pejorative sense…I thought dynamic programming was a good name. It was something not even a Congressman could object to."

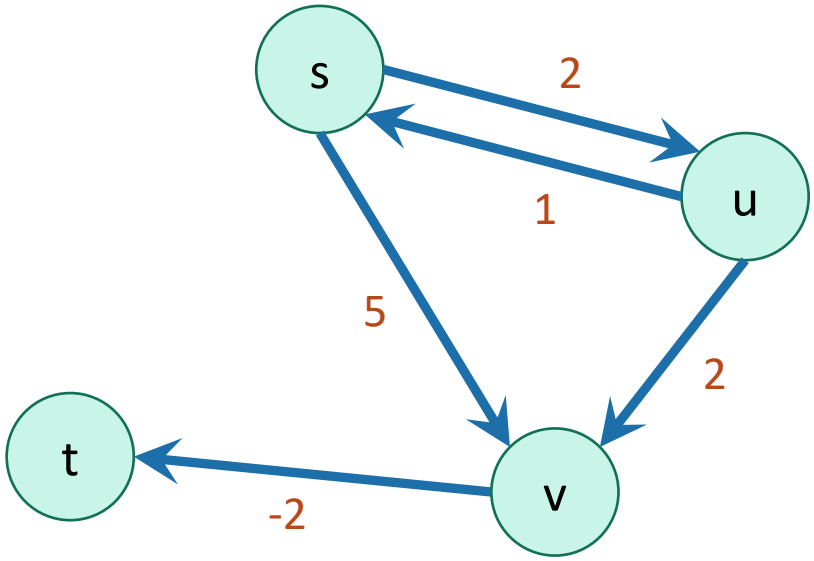
Floyd-Warshall Algorithm

Another example of DP

- This is an algorithm for **All-Pairs Shortest Paths** (APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.

Destination

Source	s	u	v	t
s	0	2	4	2
u	1	0	2	0
v	∞	∞	0	-2
t	∞	∞	∞	0



Floyd-Warshall Algorithm

Another example of DP

- This is an algorithm for **All-Pairs Shortest Paths (**APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.
- Naïve solution (if we want to handle negative edge weights):
  - For all s in G:
    - Run Bellman-Ford on G starting at s.
  - Time $O(n \cdot nm) = O(n^2m)$,
    - may be as bad as n^4 if $m=n^2$

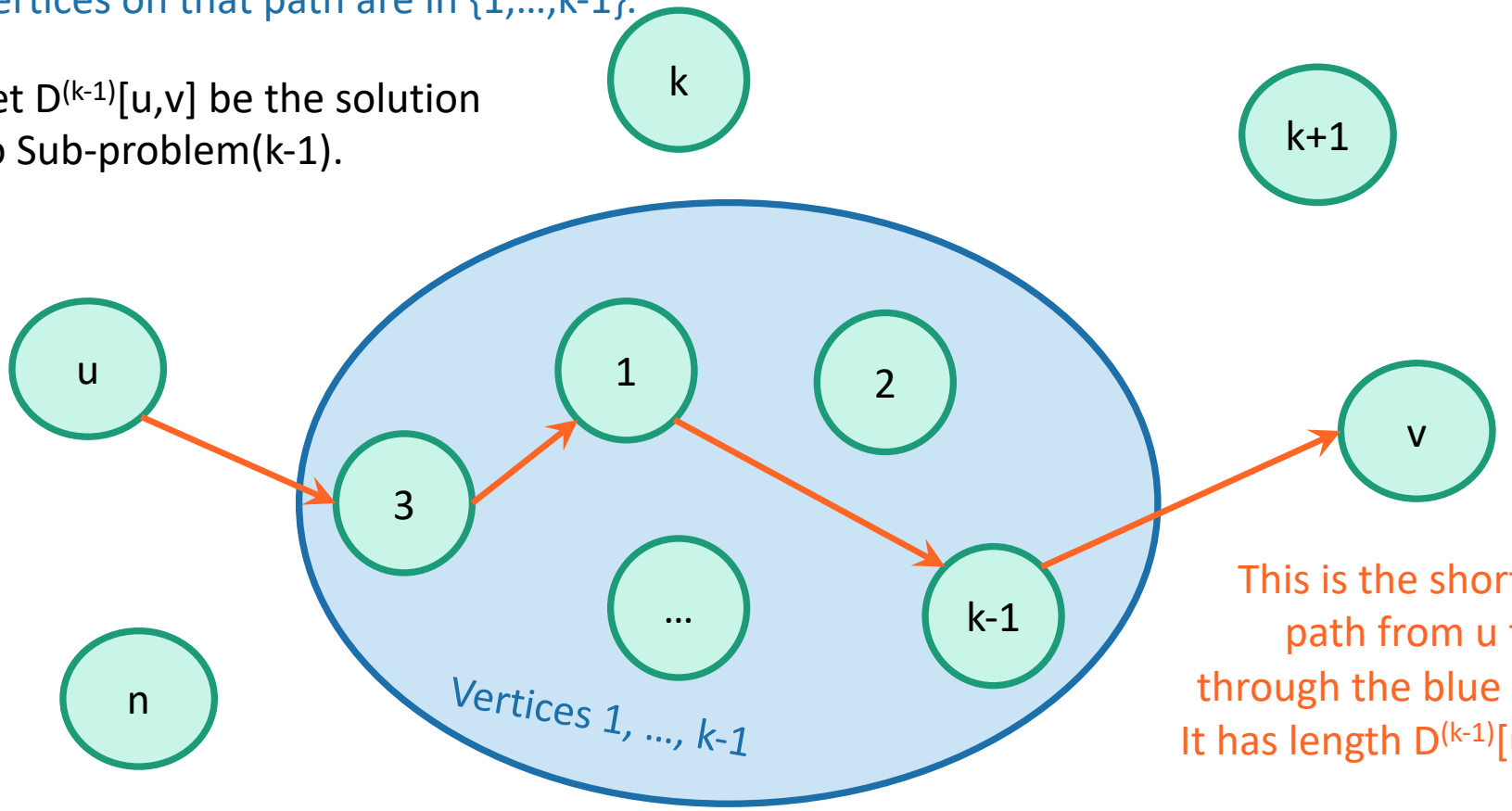
Can we do better?

Optimal substructure

Label the vertices 1,2,…,n
(We omit some edges in the picture below).

Sub-problem(k-1):
For all pairs, u,v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in {1,…,k-1}.

Let $D^{(k-1)}[u,v]$ be the solution to Sub-problem(k-1).



This is the shortest path from u to v through the blue set. It has length $D^{(k-1)}[u,v]$

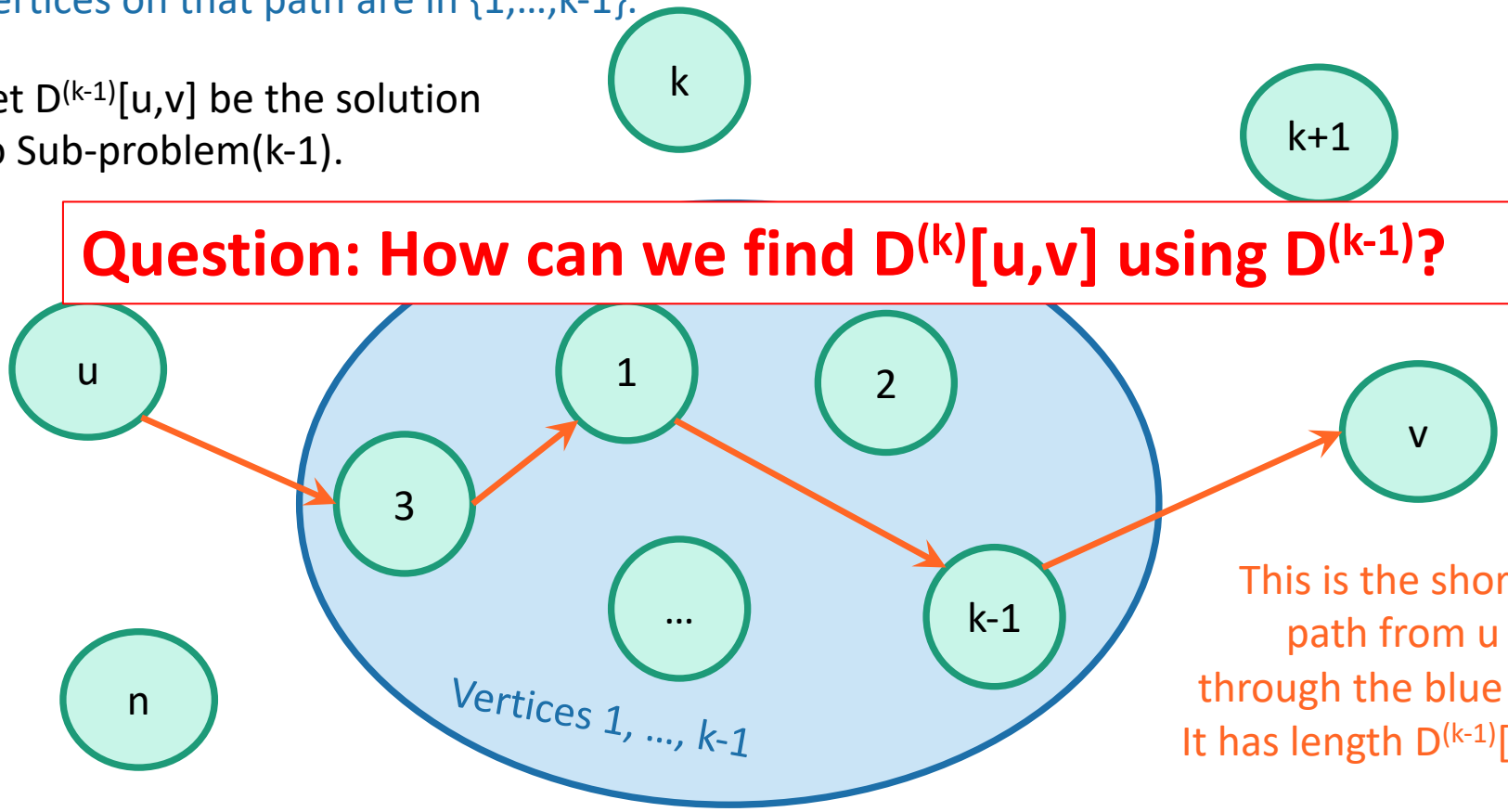
Optimal substructure

Label the vertices 1,2,…,n
(We omit some edges in the picture below).

Sub-problem(k-1):
For all pairs, u,v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in {1,…,k-1}.

Let $D^{(k-1)}[u,v]$ be the solution to Sub-problem(k-1).

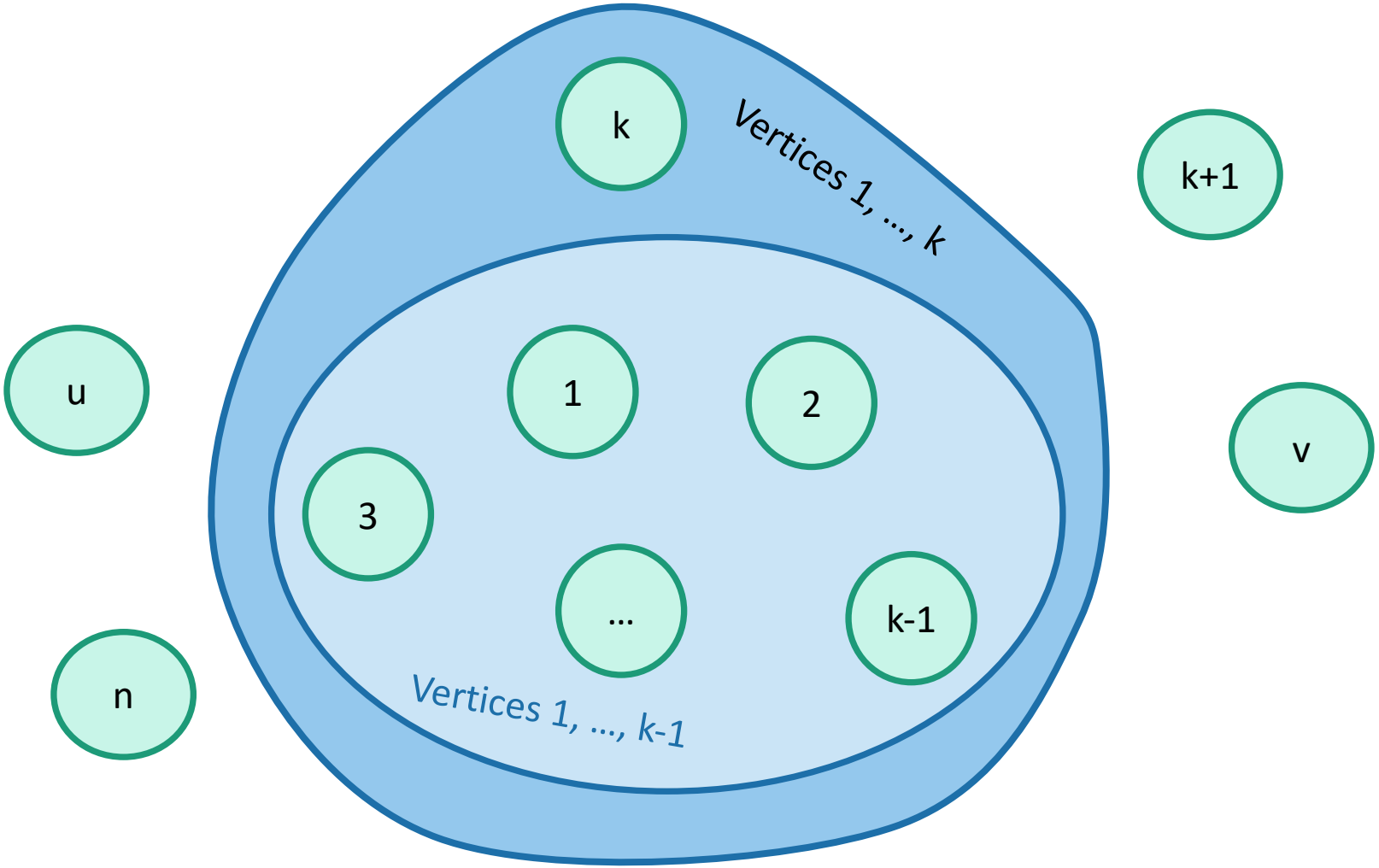
Question: How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?



This is the shortest path from u to v through the blue set. It has length $D^{(k-1)}[u,v]$

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

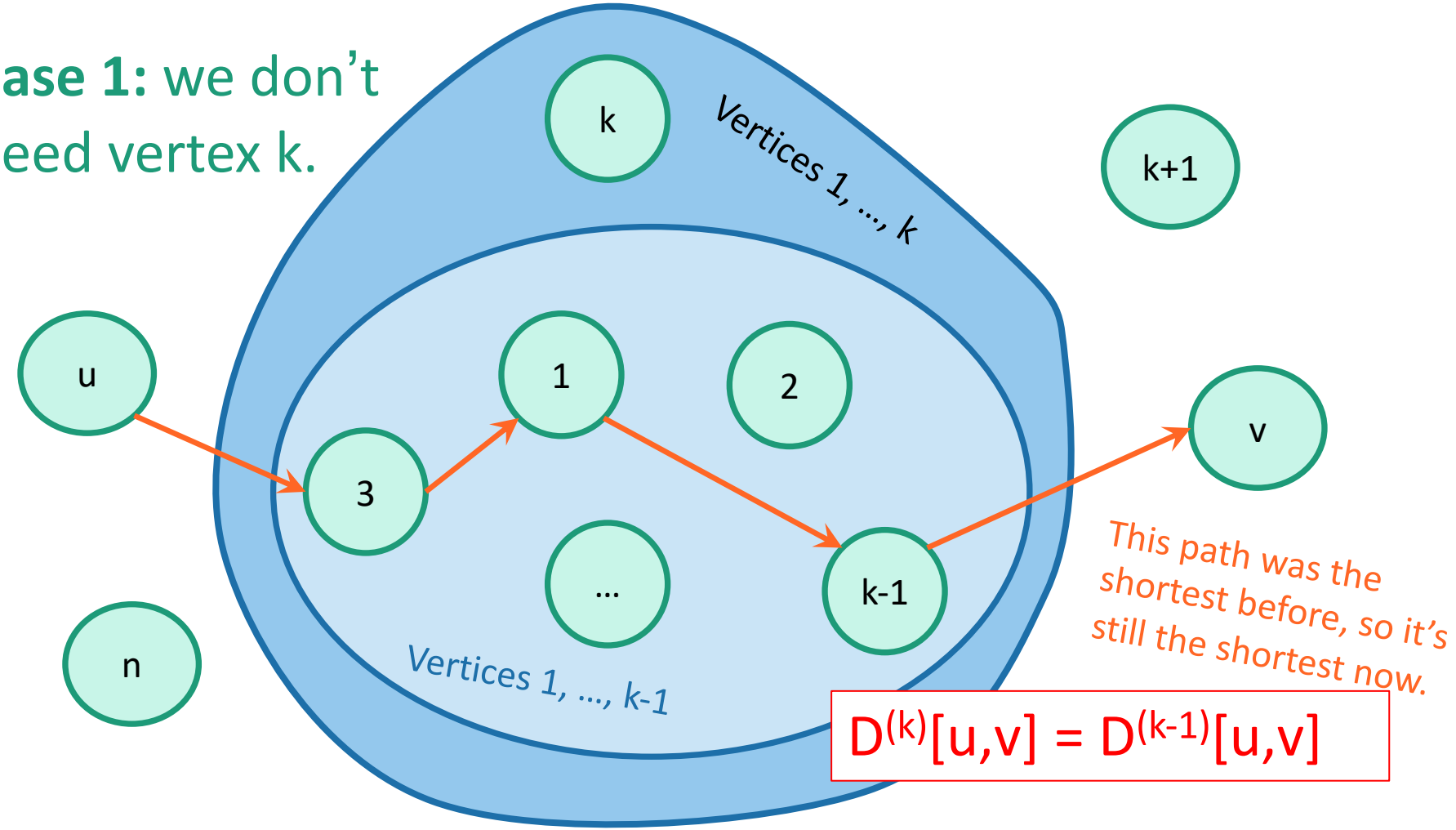
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.



How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.

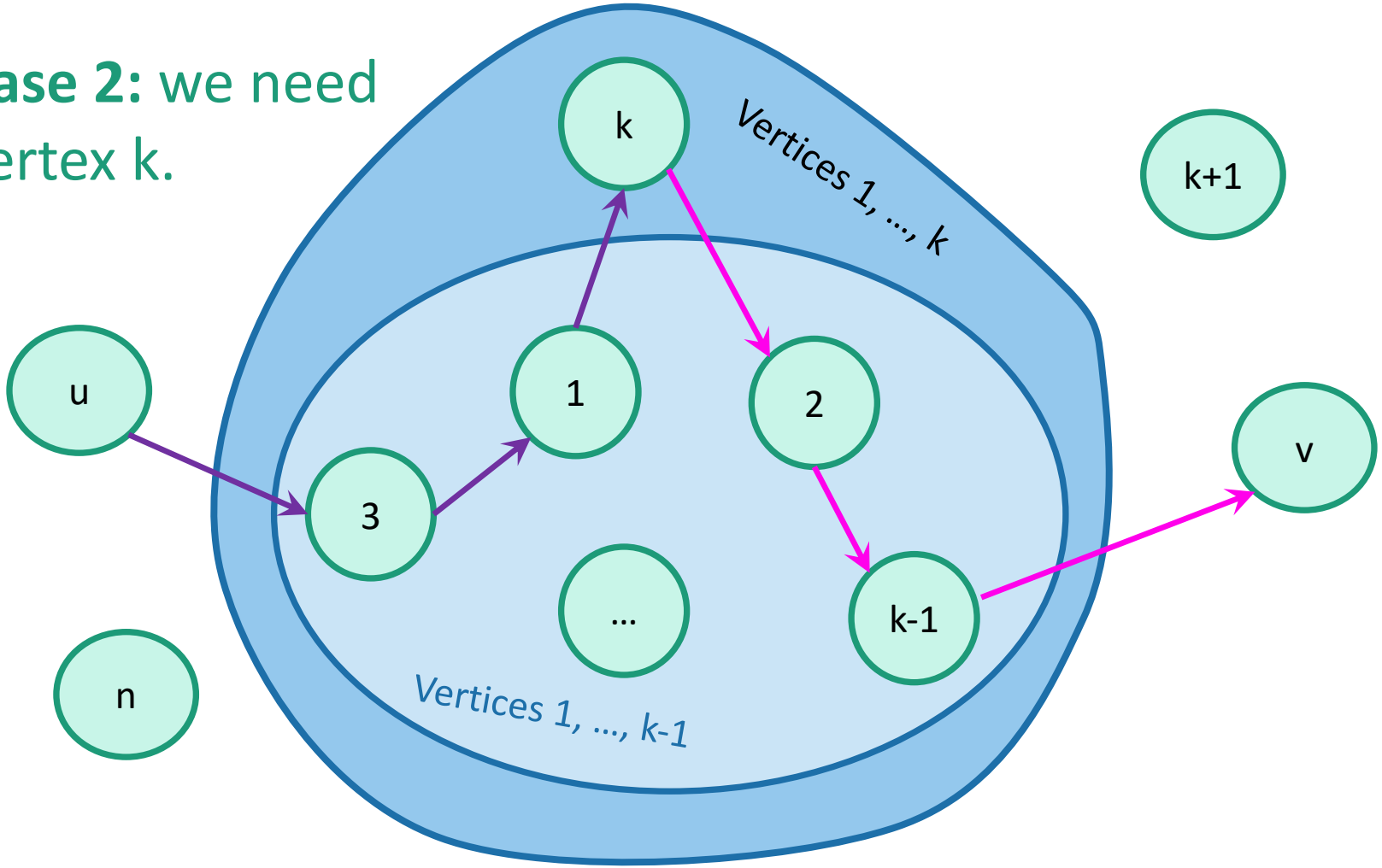
Case 1: we don't need vertex k.



How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.

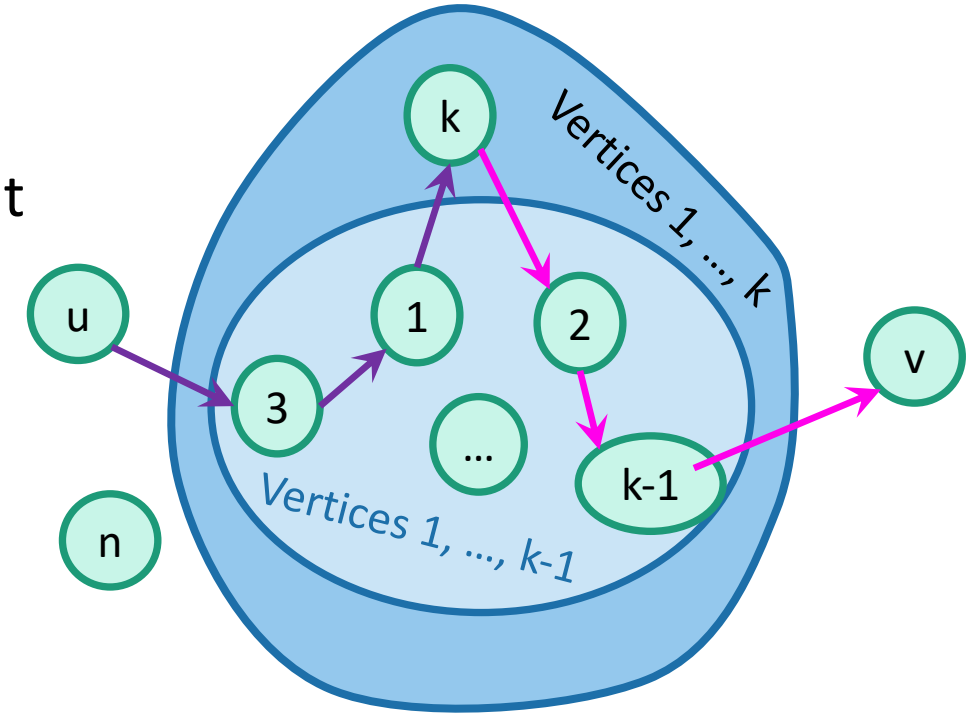
Case 2: we need vertex k.



Case 2 continued

Case 2: we need vertex k.

- If **that path** passes through k, it must look like this:
- **This path** is the shortest path from u to k through {1,…,k-1}.
  - sub-paths of shortest paths are shortest paths
- Same for **this path**.



$$D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]$$

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

Case 1: Cost of shortest path through {1,…,k-1}

Case 2: Cost of shortest path from **u to k** and then from **k to v** through {1,…,k-1}

- Optimal substructure:
  - We can solve the big problem using smaller problems.
- Overlapping sub-problems:
  - $D^{(k-1)}[k,v]$ can be used to help compute $D^{(k)}[u,v]$ for lots of different u's.

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

Case 1: Cost of shortest path through {1,…,k-1}

Case 2: Cost of shortest path from **u to k** and then from **k to v** through {1,…,k-1}

- Using our ***Dynamic programming*** paradigm, this immediately gives us an algorithm!

Floyd-Warshall algorithm

- Initialize n-by-n arrays $D^{(k)}$ for k = 0,…,n
  - $D^{(k)}[u,u] = 0$ for all u, for all k
  - $D^{(k)}[u,v] = \infty$ for all $u \neq v$, for all k
  - $D^{(0)}[u,v] = \text{weight}(u,v)$ for all (u,v) in E.
- **For** k = 1, …, n:
  - **For** pairs u,v in V^2:
    - $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$
- **Return** $D^{(n)}$

This is a bottom-up ***Dynamic programming*** algorithm.

We've basically just shown

- Theorem:

  If there are no negative cycles in a weighted directed graph G, then the Floyd-Warshall algorithm, running on G, returns a matrix $D^{(n)}$ so that:

  $D^{(n)}[u,v]$ = distance between u and v in G.

- Running time: $O(n^3)$
  - Better than running BF n times!
  - Not really better than running Dijkstra n times.
    - But it's simpler to implement and handles negative weights.
- Storage:
  - Need to store **two** n-by-n arrays, and the original graph.
    As with Bellman-Ford, we don't really need to store all n of the $D^{(k)}$.

What if there *are* negative cycles?

- Just like Bellman-Ford, Floyd-Warshall can detect negative cycles:
  - Negative cycle $\Leftrightarrow \exists$ v s.t. there is a path from v to v that goes through all n vertices that has cost < 0.
  - Negative cycle $\Leftrightarrow \exists$ v s.t. $D^{(n)}[v,v] < 0$.

- Algorithm:
  - Run Floyd-Warshall as before.
  - If there is some v so that $D^{(n)}[v,v] < 0$:
    - **return** `negative cycle.`

What have we learned?

- The Floyd-Warshall algorithm is another example of *dynamic programming*.
- It computes All Pairs Shortest Paths in a directed weighted graph in time $O(n^3)$.