BLG 336E

Analysis of Algorithms II

Lecture 5:

Greedy Algorithms, Interval Scheduling, Interval Partitioning, Shortest Paths in a Graph (Dijkstra)

Graphs-Last week

- **There are a lot of graphs.**

- We want to answer questions about them.
  - Efficient routing?
  - Community detection/clustering?
    - Computing Bacon numbers
    - Signing up for classes without violating pre-req constraints
    - How to distribute fish in tanks so that none of them will fight.

Recap-Last week

- Depth-first search
  - Useful for topological sorting
  - Also in-order traversals of BSTs
- Breadth-first search
  - Useful for finding shortest paths
  - Also for testing bipartiteness
- Both DFS, BFS:
  - Useful for exploring graphs, finding connected components, etc

Greedy Algorithms

An algorithm is greedy if it builds a solution in small steps, choosing a decision at each step myopically [=locally, not considering what may happen ahead] to optimize some underlying criterion.

It is easy to design a greedy algorithm for a problem. There may be many different ways to choose the next step locally.

What is challenging is to produce an algorithm that produces
either an optimal solution,
or a solution close to the optimum.

Proving that the Greedy Solution is Optimal

Approaches to prove that the greedy solution is as good or better as any other solution:

1) prove that it stays ahead of any other algorithm
   e.g. Interval Scheduling

2) exchange argument (more general): consider any possible solution to the problem and gradually transform into the solution found by the greedy solution without hurting its quality.
   e.g. Scheduling to Minimize Lateness

Greedy Analysis Strategies

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Example Problems

Interval Scheduling

Interval Partitioning

Scheduling to Minimize Lateness

Shortest Paths in a Graph (Dijkstra)

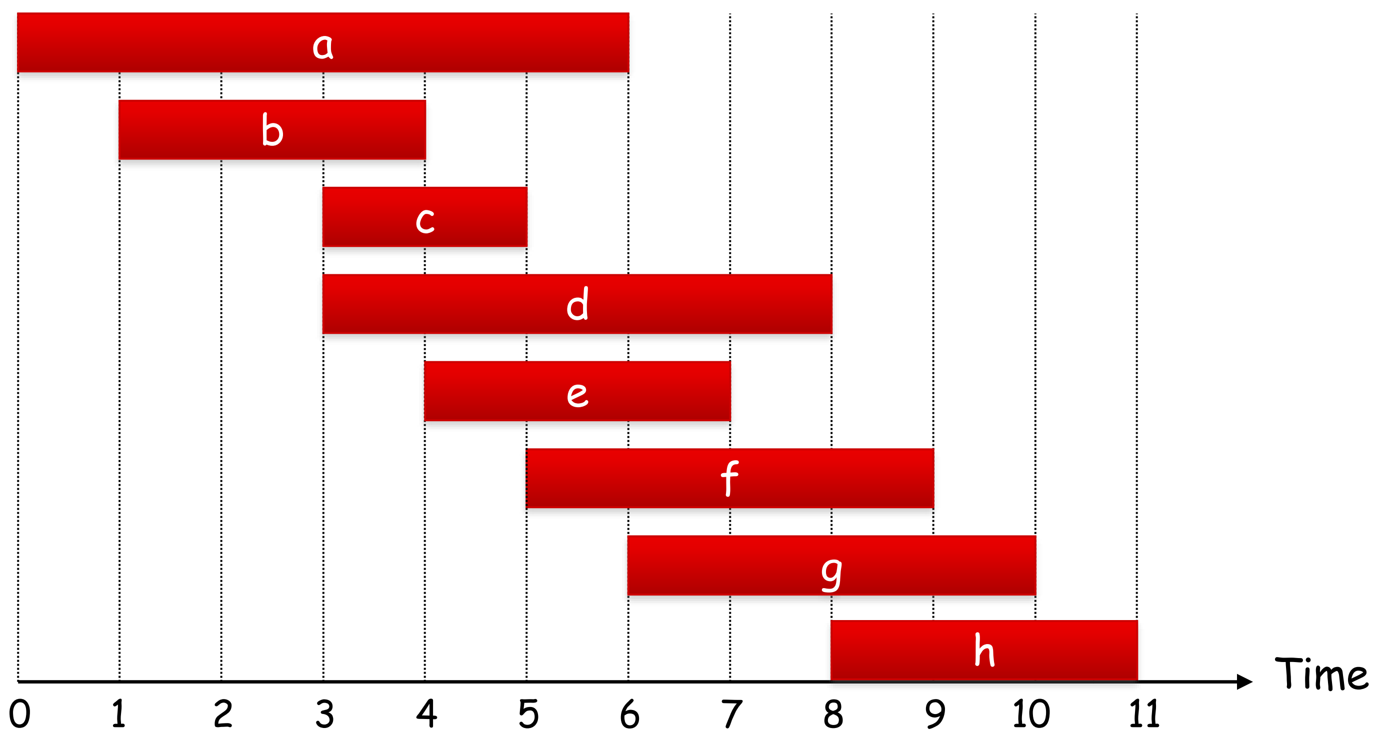
The Minimum Spanning Tree Problem
  Prim's Algorithm, Kruskal's Algorithm

Huffman Codes and Compression

Interval Scheduling

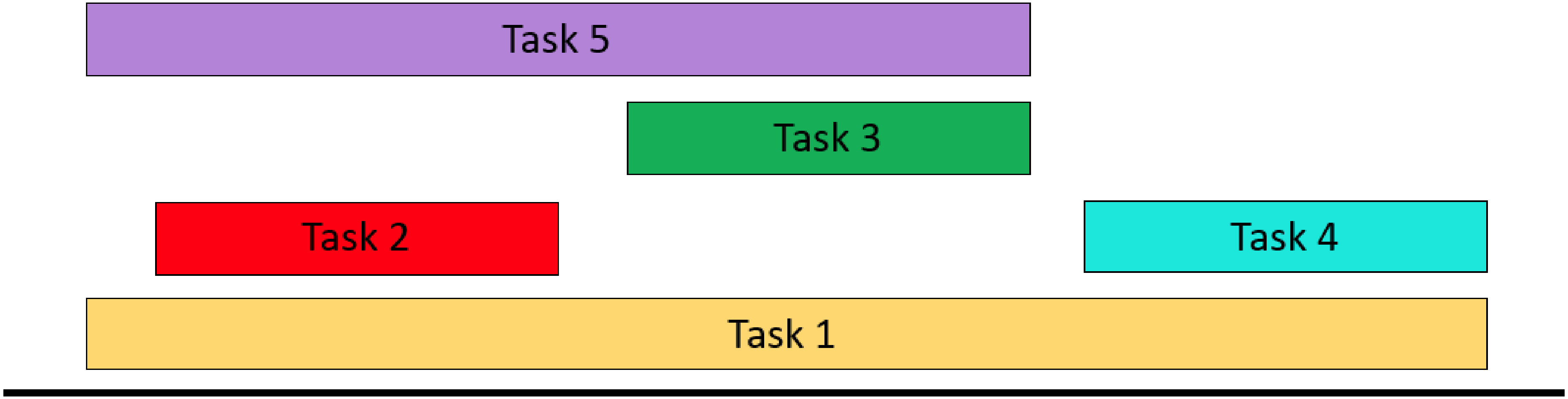
Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling Example 1

`Input:`



Optimal Solution ? Maximum number of jobs?

> `Output:` `[Task 2, Task 3, Task 4]`

Interval Scheduling Example 2

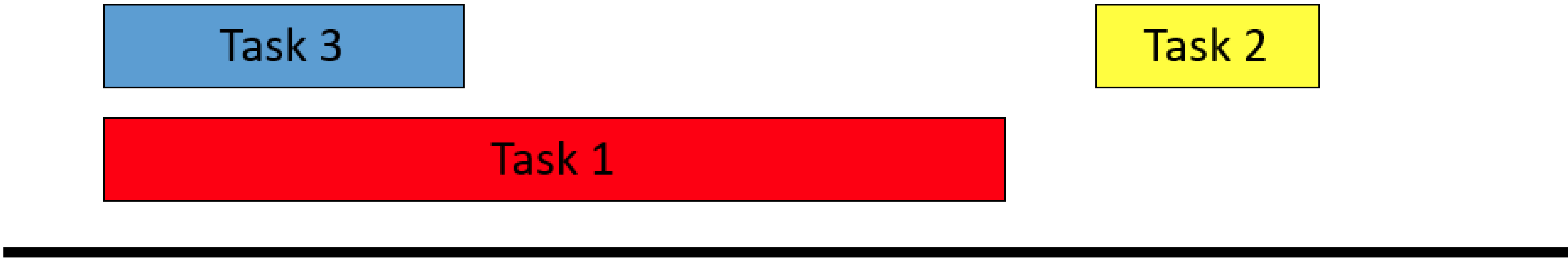
Question 1

What would the pseudocode for the above look like?

```
R: set of requests

Initialize S to be the empty set
While R is not empty
    Choose i in R
    Add i to S
Return S* = S
```

Interval Scheduling Example 3



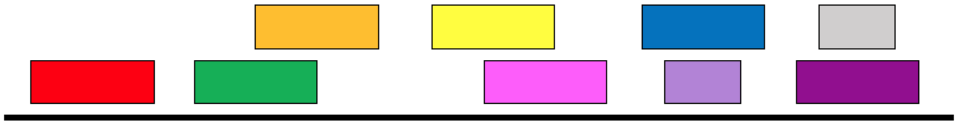
Question 2

How would you modify Algorithm draft 1 to handle this case?

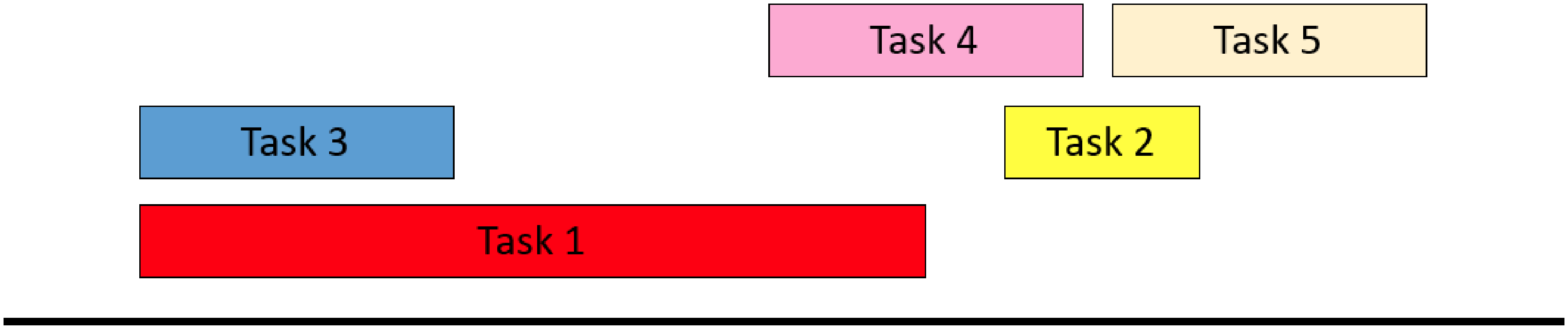
```
R: set of requests

Initialize S to be the empty set
While R is not empty
    Choose i in R
    Add i to S
    Remove all requests that conflict with i from R
Return S* = S
```

Or a more generally:



Interval Scheduling Example 4



Question 3

How should we update Algorithm draft 2 to handle this case?

```
R: set of requests

Initialize S to be the empty set
While R is not empty
    Choose i in R where v(i) is minimized
    Add i to S
    Remove all requests that conflict with i from R
Return S* = S
```

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

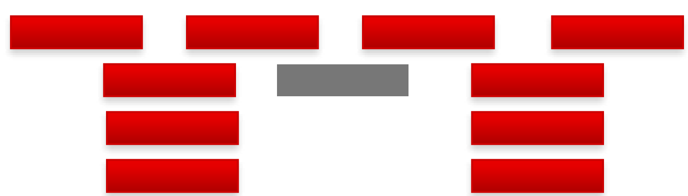
- [Earliest start time] Consider jobs in ascending order of start time s_j.
- [Earliest finish time] Consider jobs in ascending order of finish time f_j.
- [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

breaks earliest start time

breaks shortest interval



breaks fewest conflicts

Interval Scheduling: Greedy Algorithm

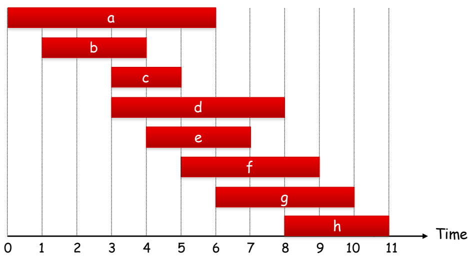
Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

  jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. $O(n \log n)$, due to the sorting operation

- Remember job j^* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

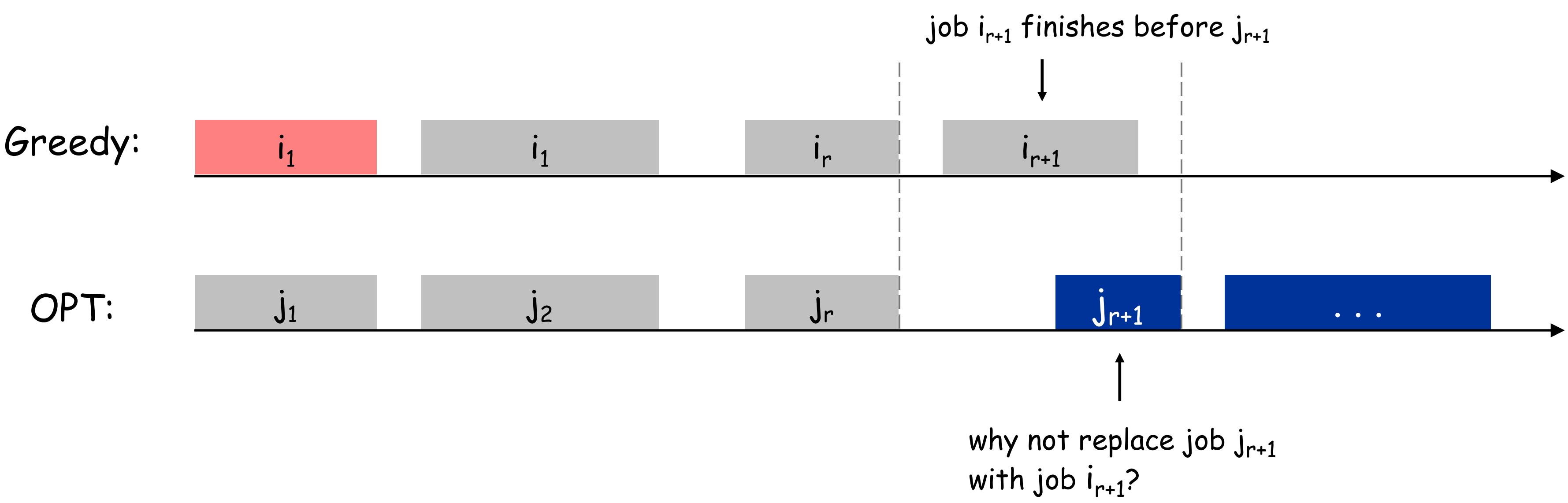


Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

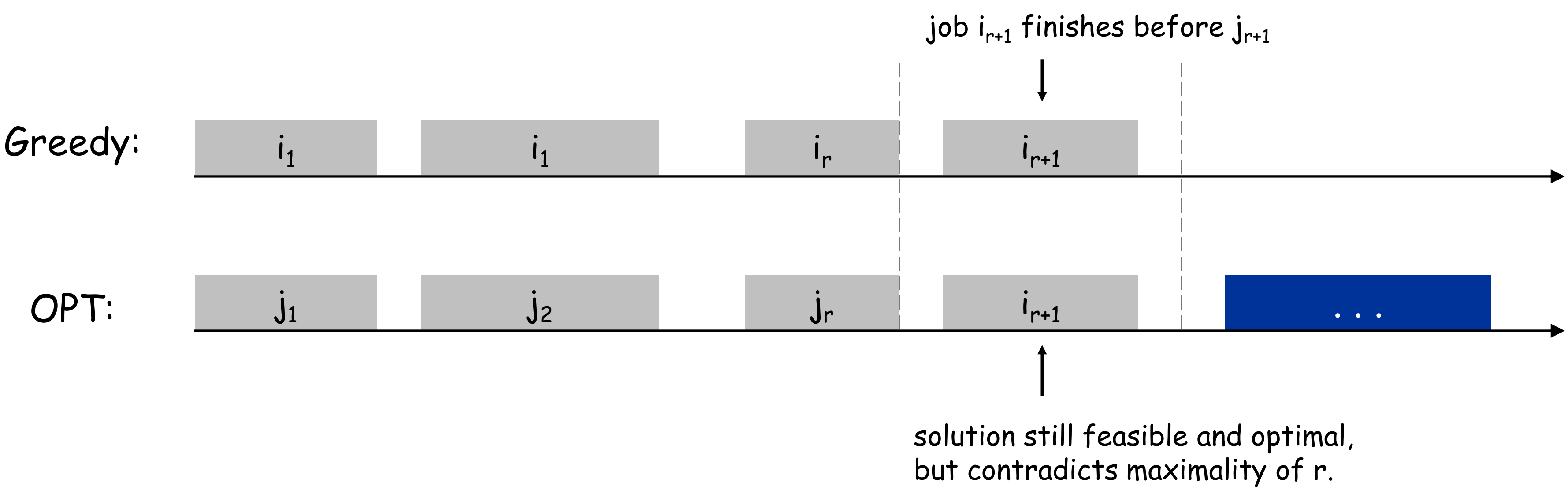


Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Interval Partitioning

Interval Partitioning

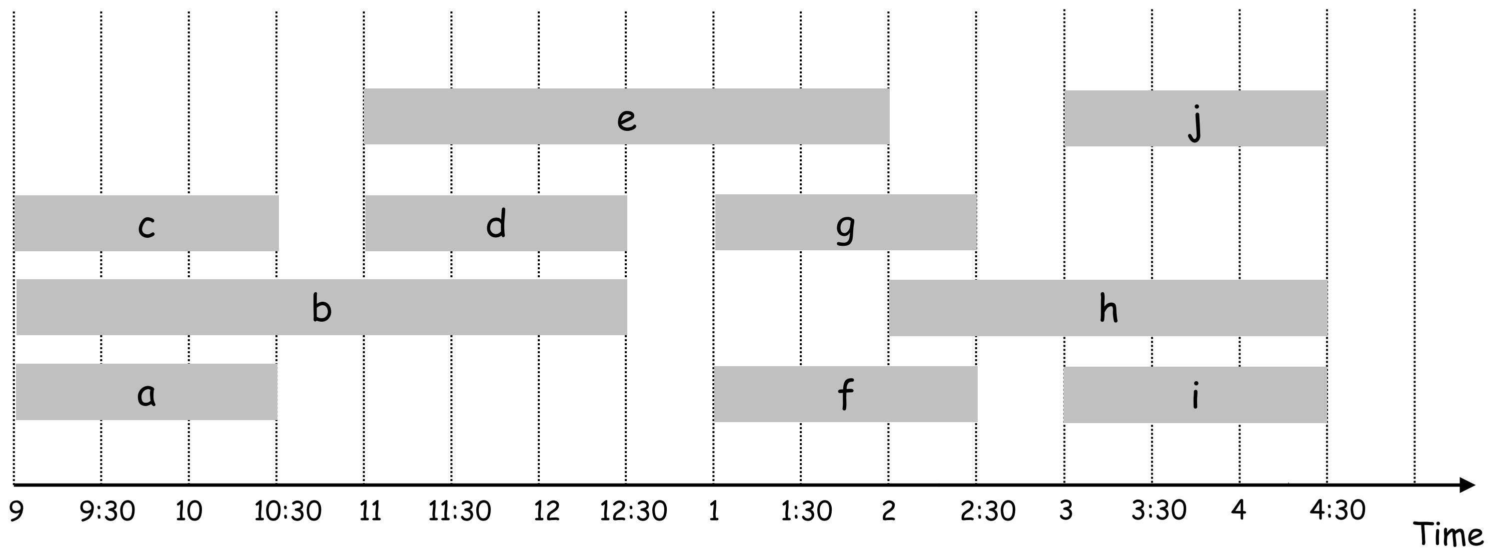
Interval partitioning.

Aim: Schedule all the requests by using as few resources as possible.

Example: Classroom Scheduling

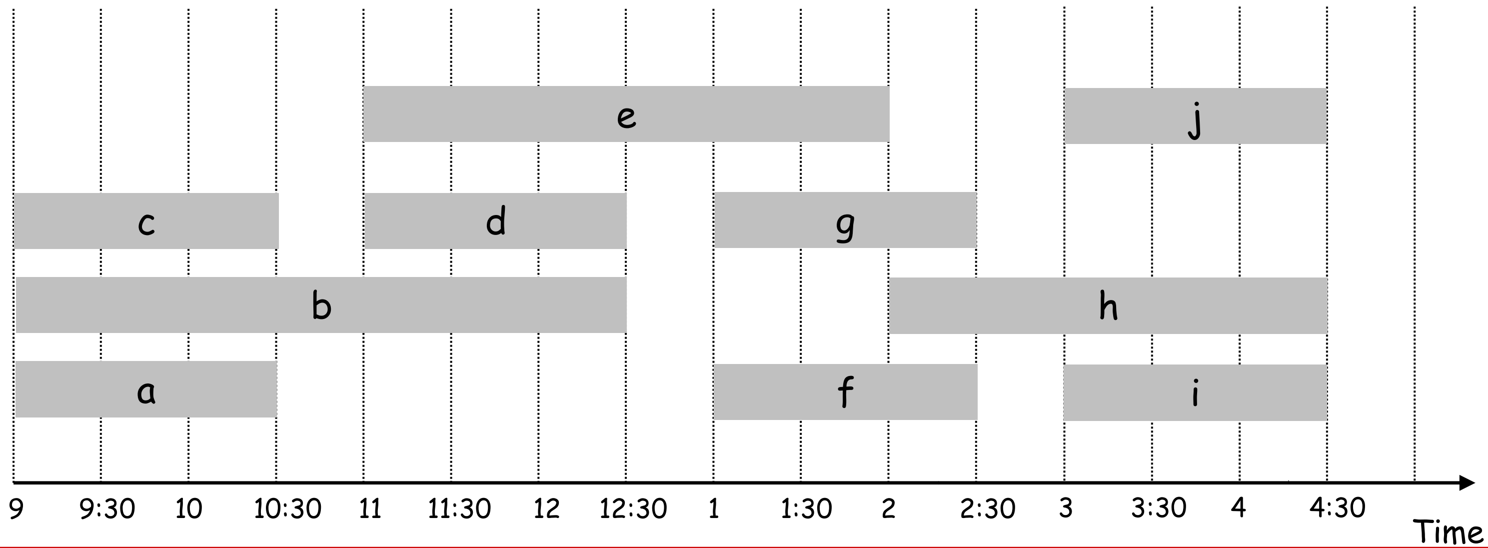
- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

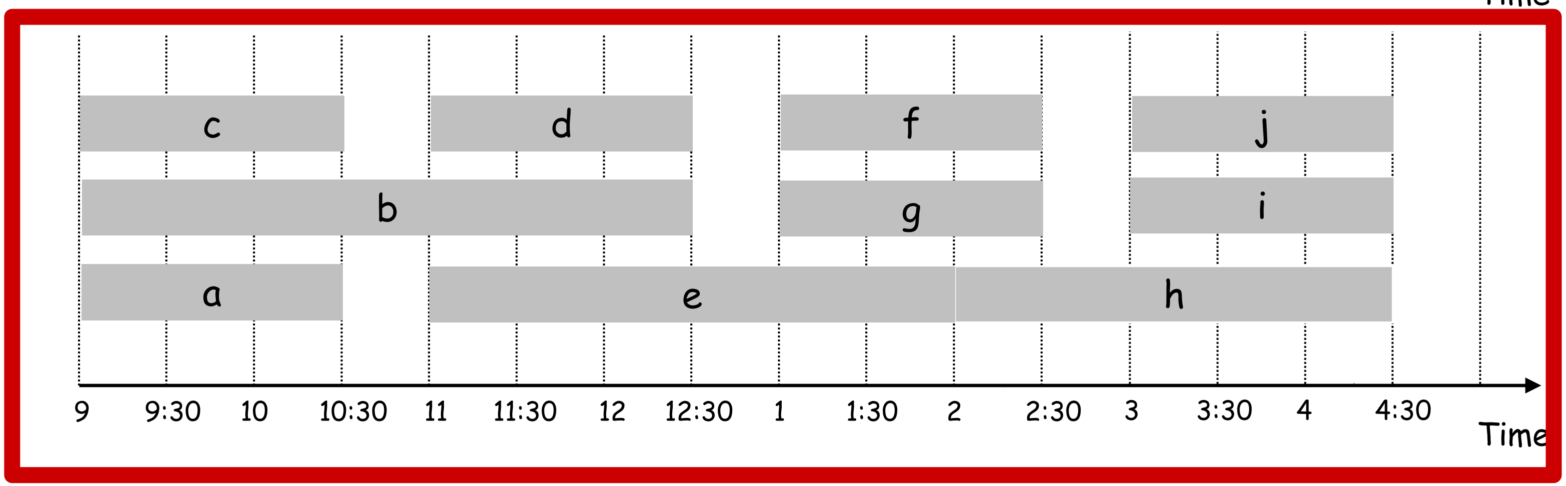
Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning

Ex: This schedule uses only 3.





Interval Partitioning: Lower Bound on Optimal Solution

Def. The depth of a set of intervals is the maximum number that pass over any single point on the time-line.

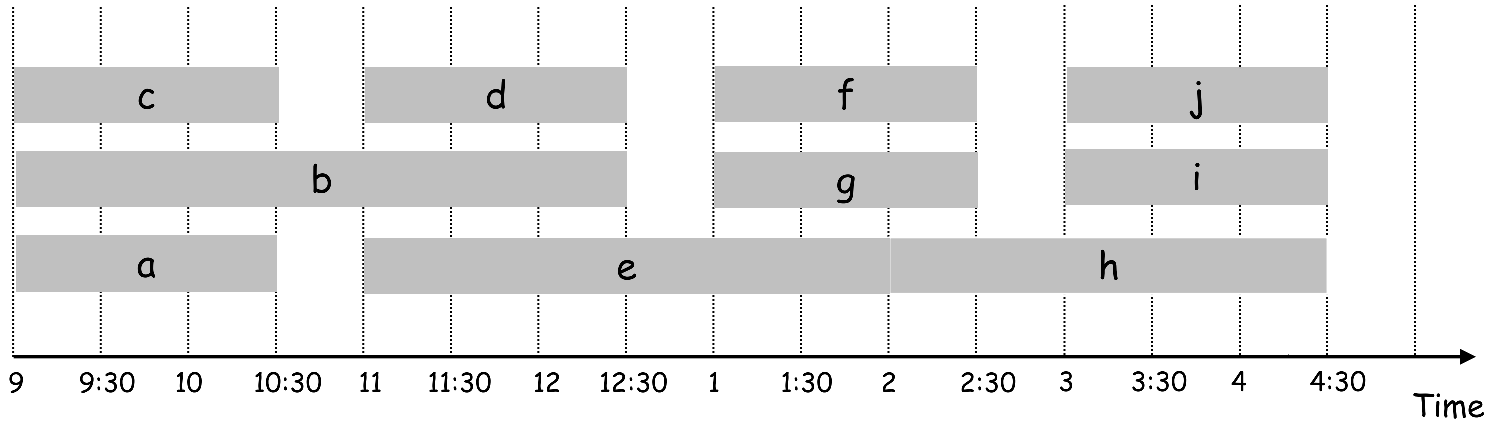
Key observation. Number of classrooms needed $\geq$ depth.

Ex: Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

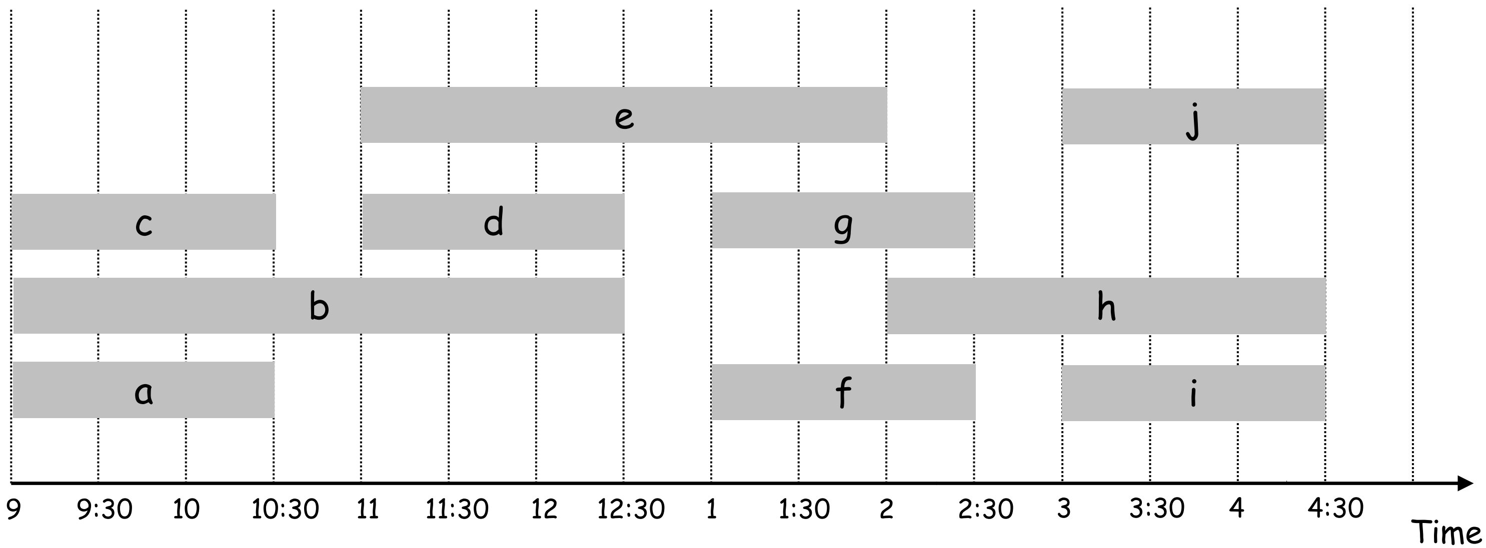
a, b, c all contain 9:30

Q. Does there always exist a schedule equal to depth of intervals?

R. May not be.



Depth of previous schedule



- Depth = 3 → Schedule is not optimal

Interval Partitioning: Greedy Algorithm

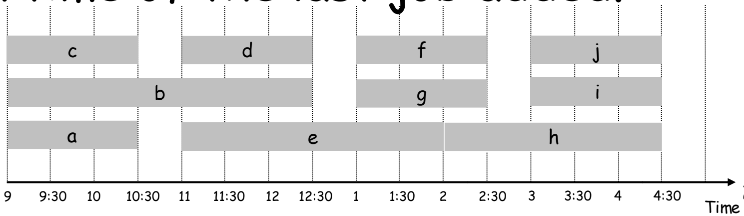
Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0    ← number of allocated classrooms

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Implementation. O(n log n).

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.



Interval Partitioning: Greedy Analysis

Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Greedy algorithm is optimal.

Pf.

- Let d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms. ▪

Scheduling to Minimize Lateness

We have a single resource and a set of n requests to use the resource for an interval of time. Each request has a deadline, *d*, and requires a contiguous time interval of length, *t*, but willing to be scheduled at any time before the deadline.

Aim: Minimizing the lateness

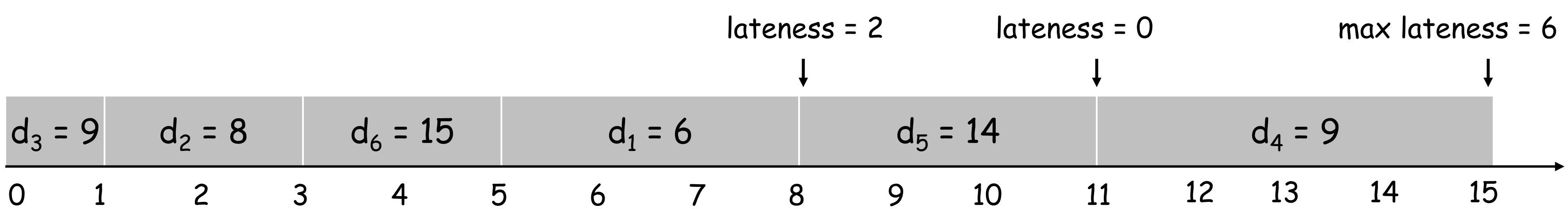
Scheduling to Minimizing Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, \; f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

counterexample

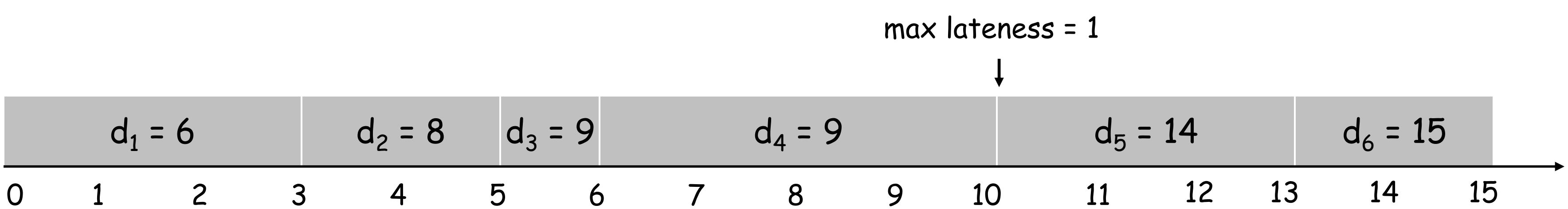
Minimizing Lateness: Greedy Algorithm

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

Greedy algorithm. Earliest deadline first.

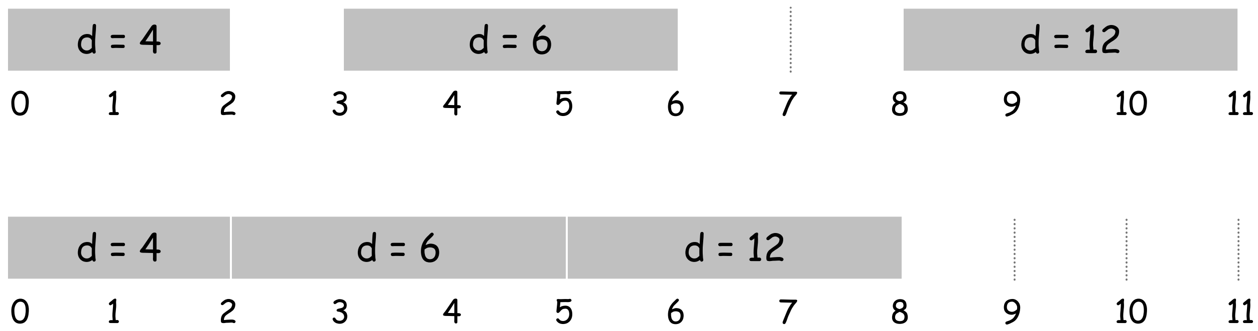
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + t_j]
   s_j ← t, f_j ← t + t_j
   t ← t + t_j
output intervals [s_j, f_j]
```



Minimizing Lateness: No Idle Time

Observation. There exists an optimal schedule with no idle time (no "gaps" between the scheduled jobs).

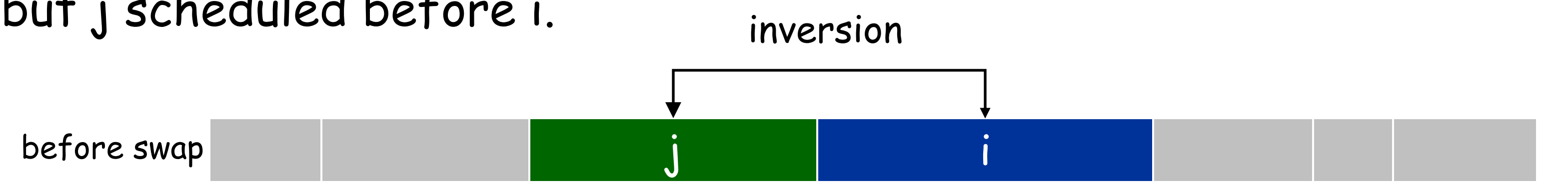


Observation. The greedy schedule has no idle time.

This is good since the aggregate execution time can not be smaller. We must check if it satisfies "minimum lateness."

Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.

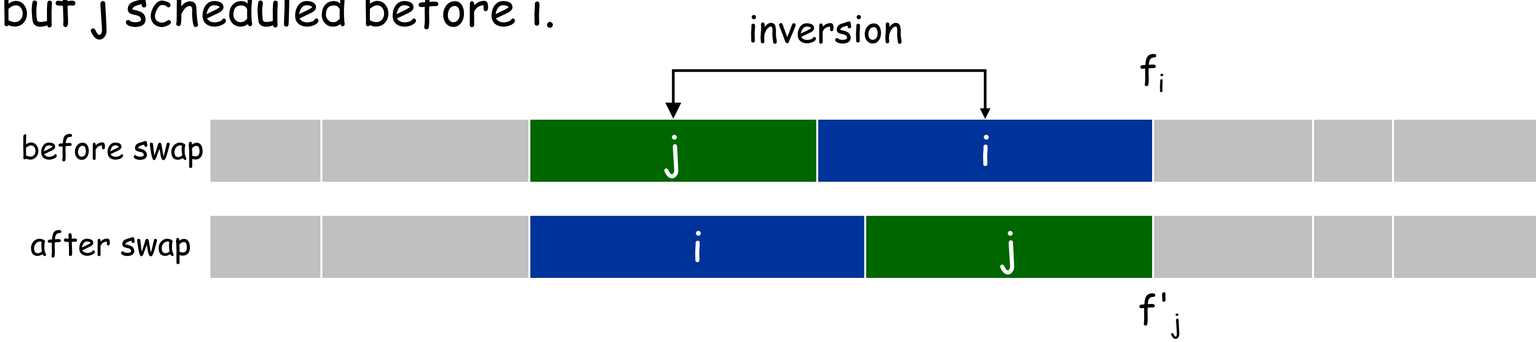


Observation. Greedy schedule has no inversions.

Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.



Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job j is late:

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (i < j) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

Minimizing Lateness-Example

- Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.

Pf. Let ℓ be the lateness before the swap, and let ℓ ' be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job j is late:

ℓ¢	=	f_j¢- d_j	(definition)
	=	f_i - d_j	(j finishes at time f_i)
	£	f_i - d_i	($i < j$)
	£	ℓ_i	(definition)

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

max lateness = 1

$d_1 = 6$	$d_2 = 8$	$d_3 = 9$	$d_4 = 9$	$d_5 = 14$	$d_6 = 15$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Minimizing Lateness: Analysis of Greedy Algorithm

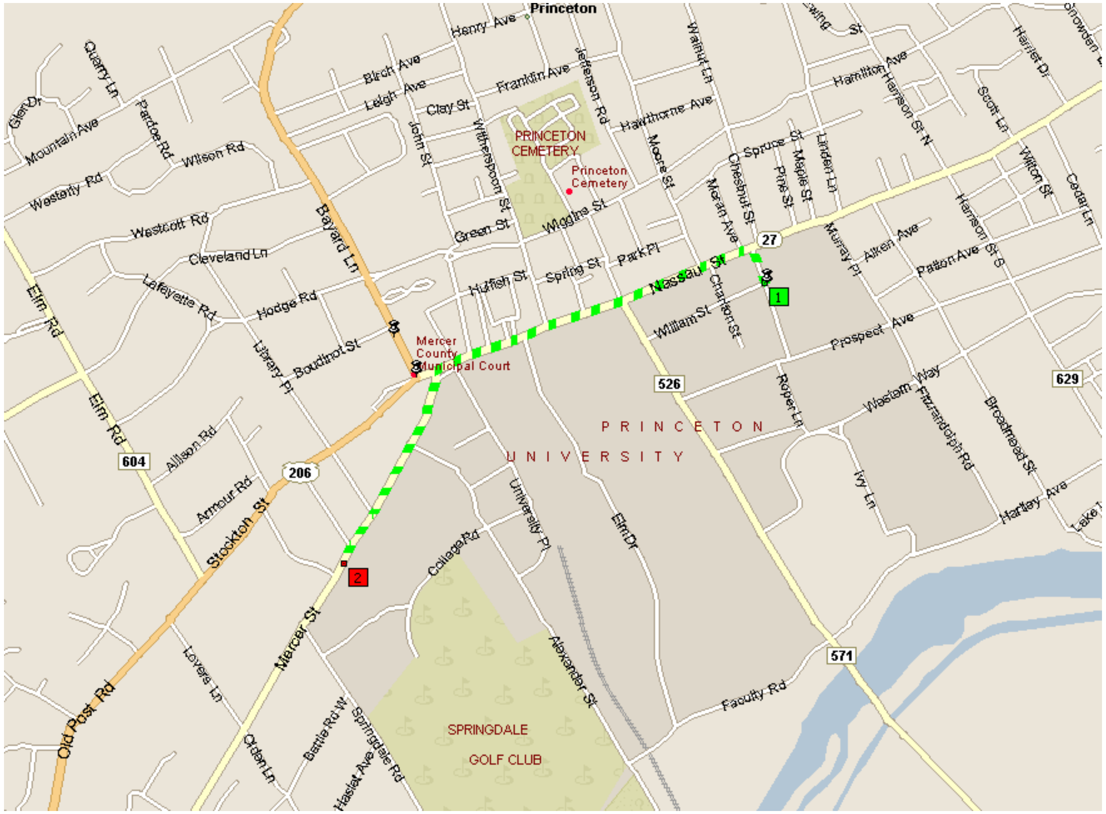
All schedules with no inversions and no idle time has the same maximum lateness.

Theorem. Greedy schedule S is optimal.

Pf. Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume S* has no idle time.
- If S* has no inversions, then S = S*.
- If S* has an inversion, let i-j be an adjacent inversion.
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts definition of S* ▪

Shortest Paths in a Graph

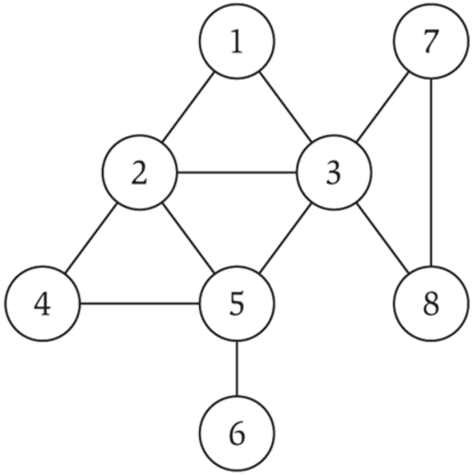


shortest path from Princeton CS department to Einstein's house

Graphs-Recap

Undirected graph. $G = (V, E)$

- V = nodes.
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|$, $m = |E|$.



$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

$E = \{ 1\text{-}2, 1\text{-}3, 2\text{-}3, 2\text{-}4, 2\text{-}5, 3\text{-}5, 3\text{-}7, 3\text{-}8, 4\text{-}5, 5\text{-}6 \}$

$n = 8$

$m = 11$

Single-Source Shortest Path

Input: directed graph G=(V, E). (m=|E|, n=|V|)
- each edge has non negative length l_e
- source vertex s

Length of path = sum of edge lengths

Path length = 6

Output: for each $v \in V$, compute
L(v) := length of a shortest s-v path in G

Assumption:
1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $l_e \geq 0 \;\; \forall e \in E$

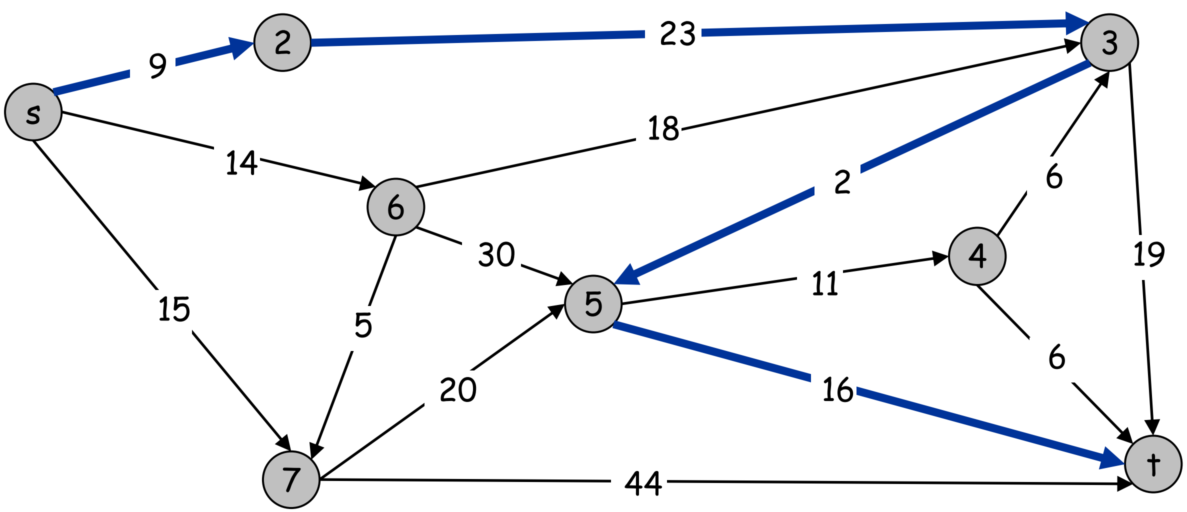
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

Example

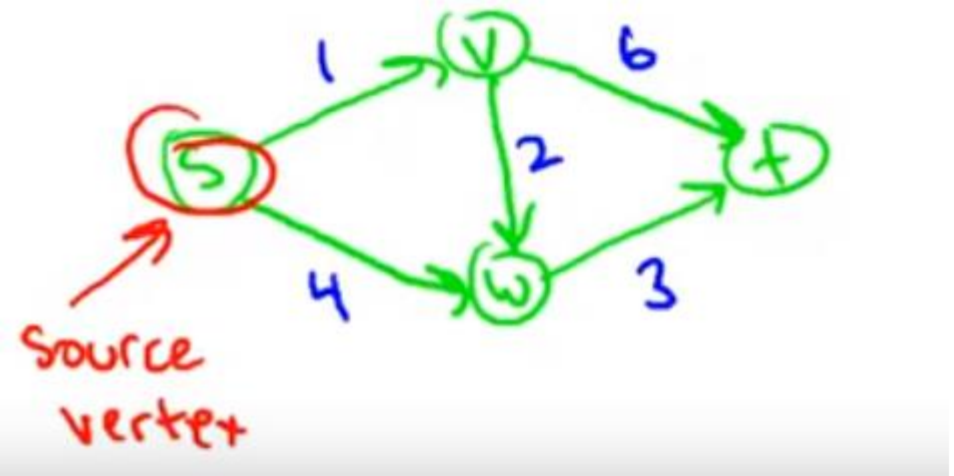
One of the following is the list of shortest-path distances for the nodes s,v,w,t, respectively. Which is it?

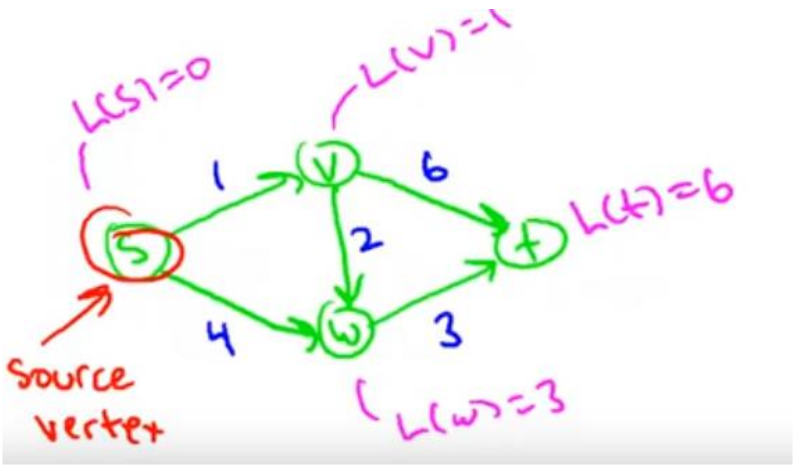
A. 0,1,2,3

B. 0,1,4,7

C. 0,1,4,6

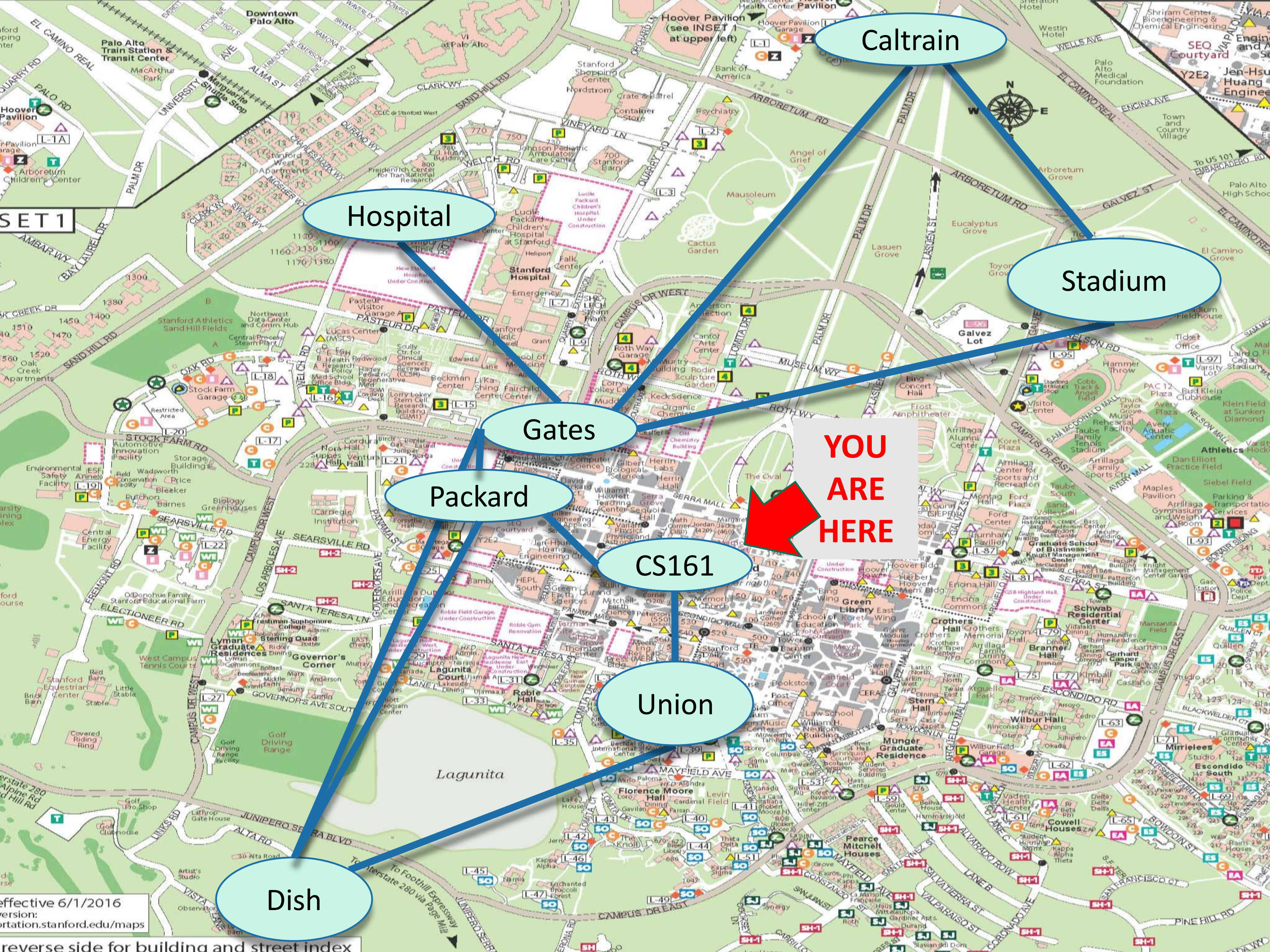
D. 0,1,3,6 ✓





Shortest Path

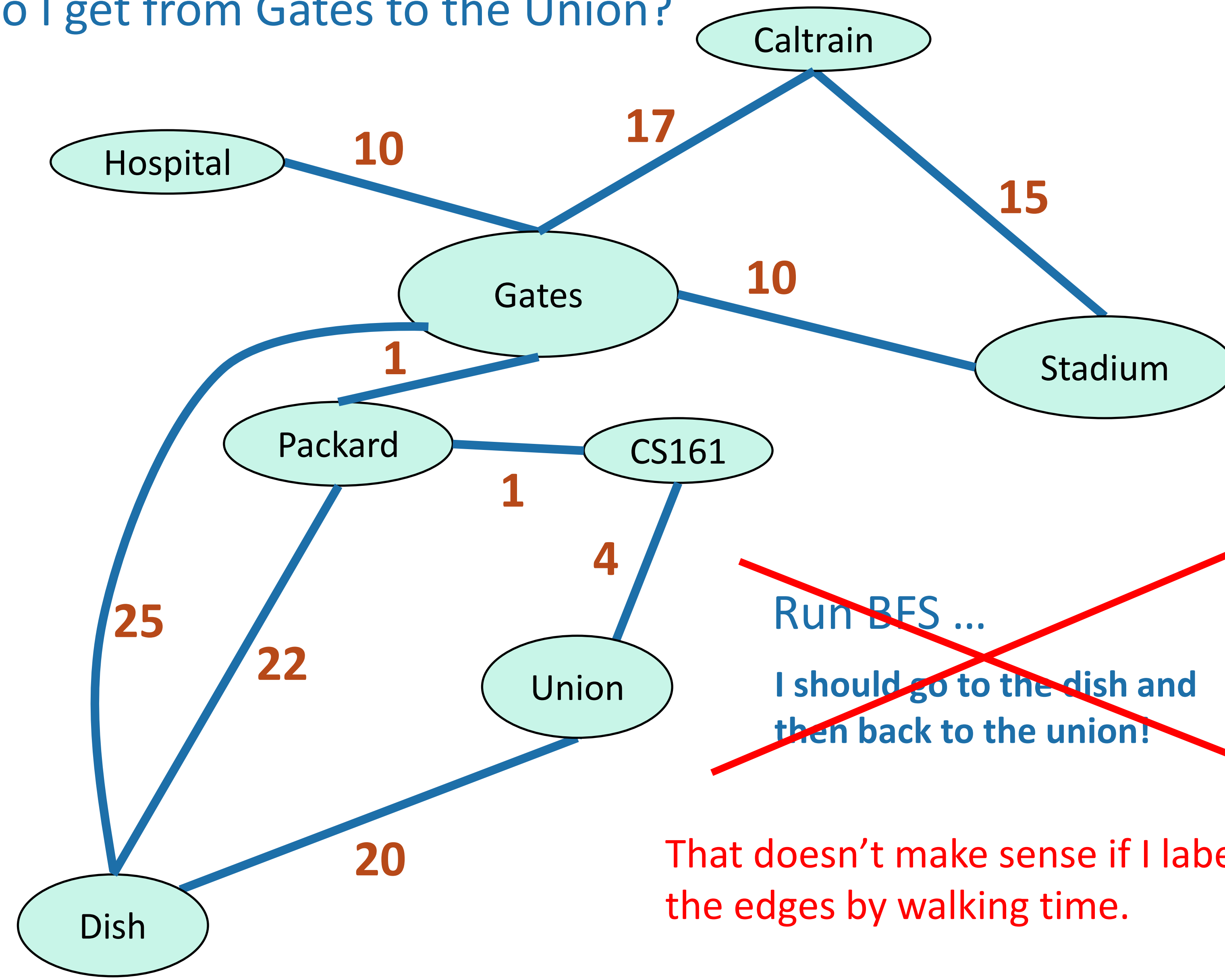
- What if the graphs are weighted?
  - All nonnegative weights: Dijkstra!
  - If there are negative weights: Bellman-Ford! (if time permits)



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

Just the graph

How do I get from Gates to the Union?



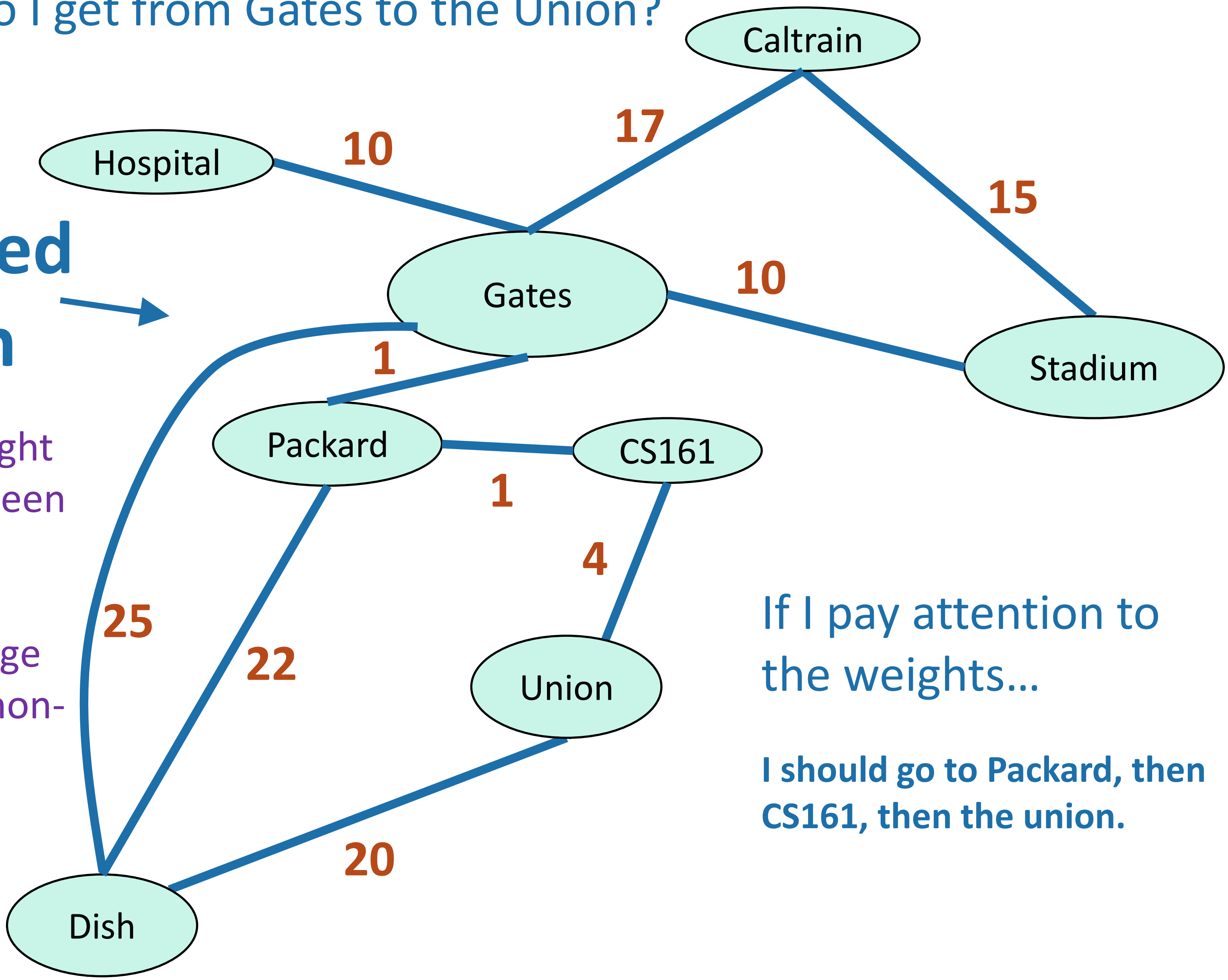
~~Run BFS …~~

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

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

Just the graph

How do I get from Gates to the Union?



weighted graph

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

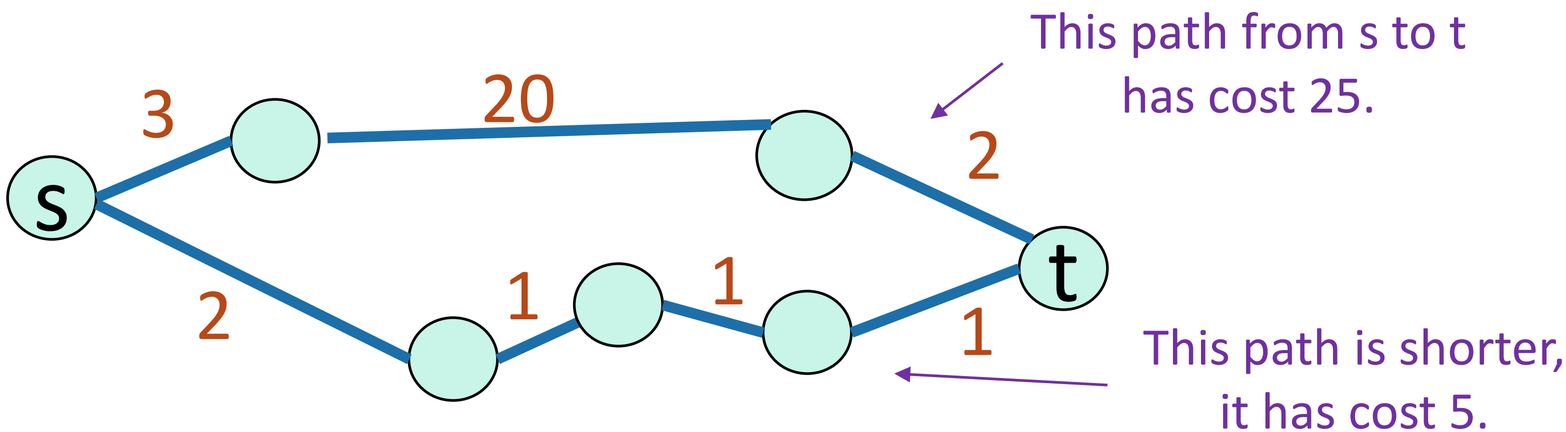
For now, edge weights are non-negative.

If I pay attention to the weights…

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

Shortest path problem

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

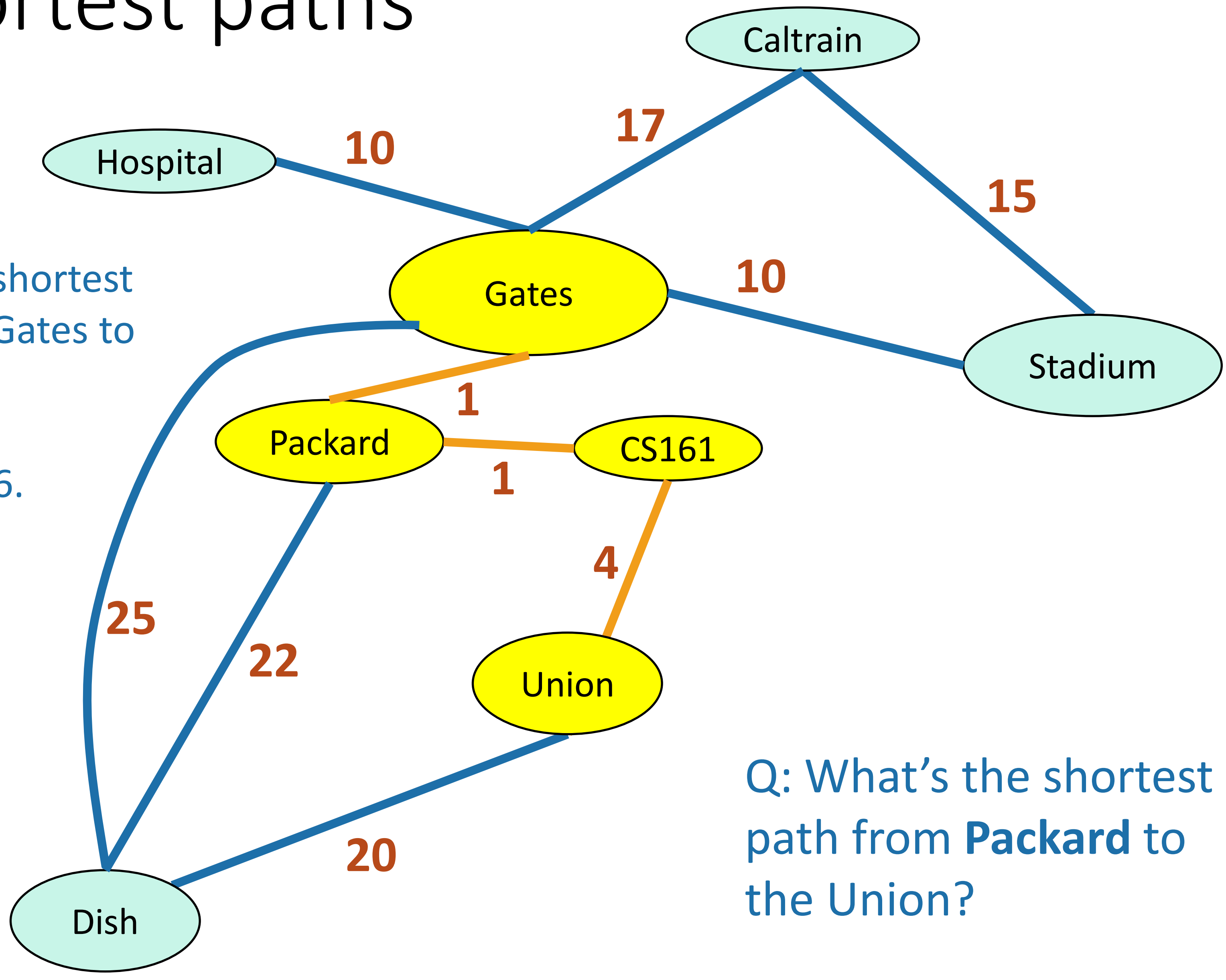


- The **distance** d(u,v) between two vertices u and v is the cost of the the shortest path between u and v.
- For this lecture **all graphs are directed**, but to save on notation I'm just going to draw undirected edges.

Shortest paths

This is the shortest path from Gates to the Union.

It has cost 6.

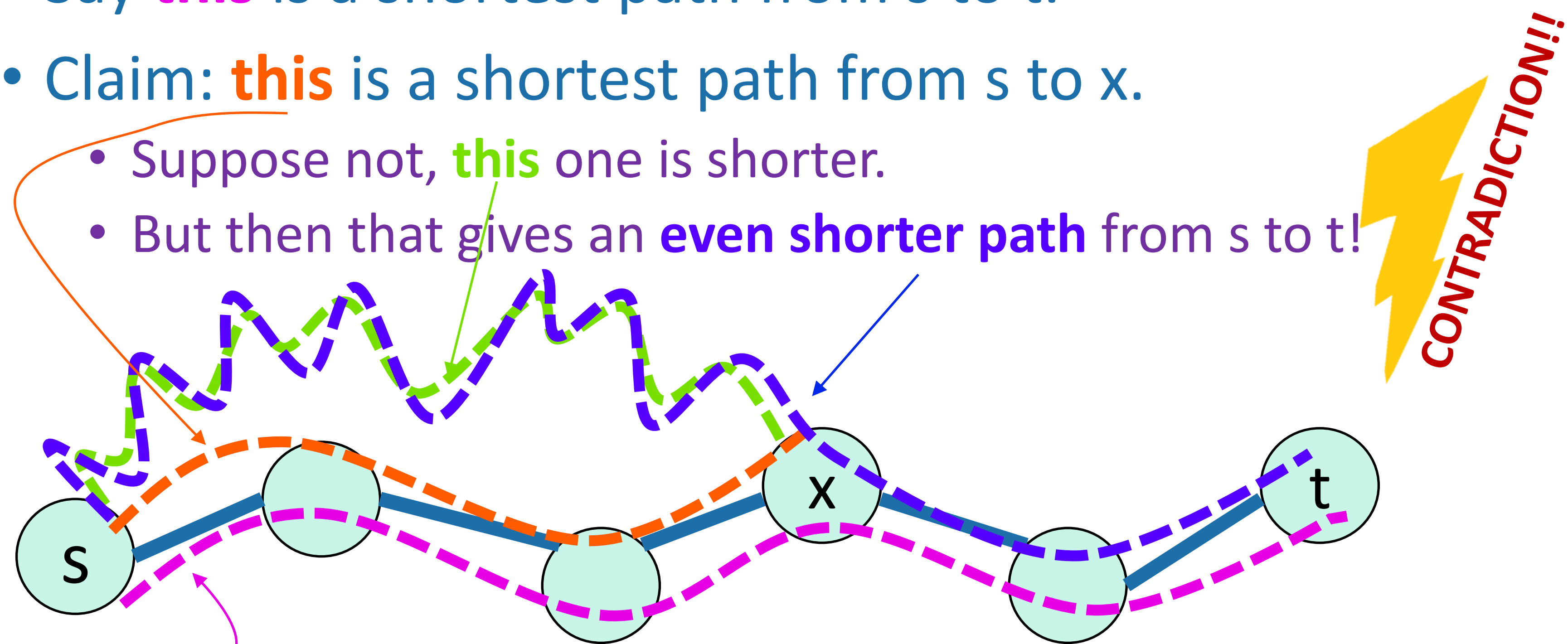


Q: What's the shortest path from **Packard** to the Union?

Warm-up

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

- Say **this** is a shortest path from s to t.
- Claim: **this** is a shortest path from s to x.
  - Suppose not, **this** one is shorter.
  - But then that gives an **even shorter path** from s to t!



Single-source shortest-path problem

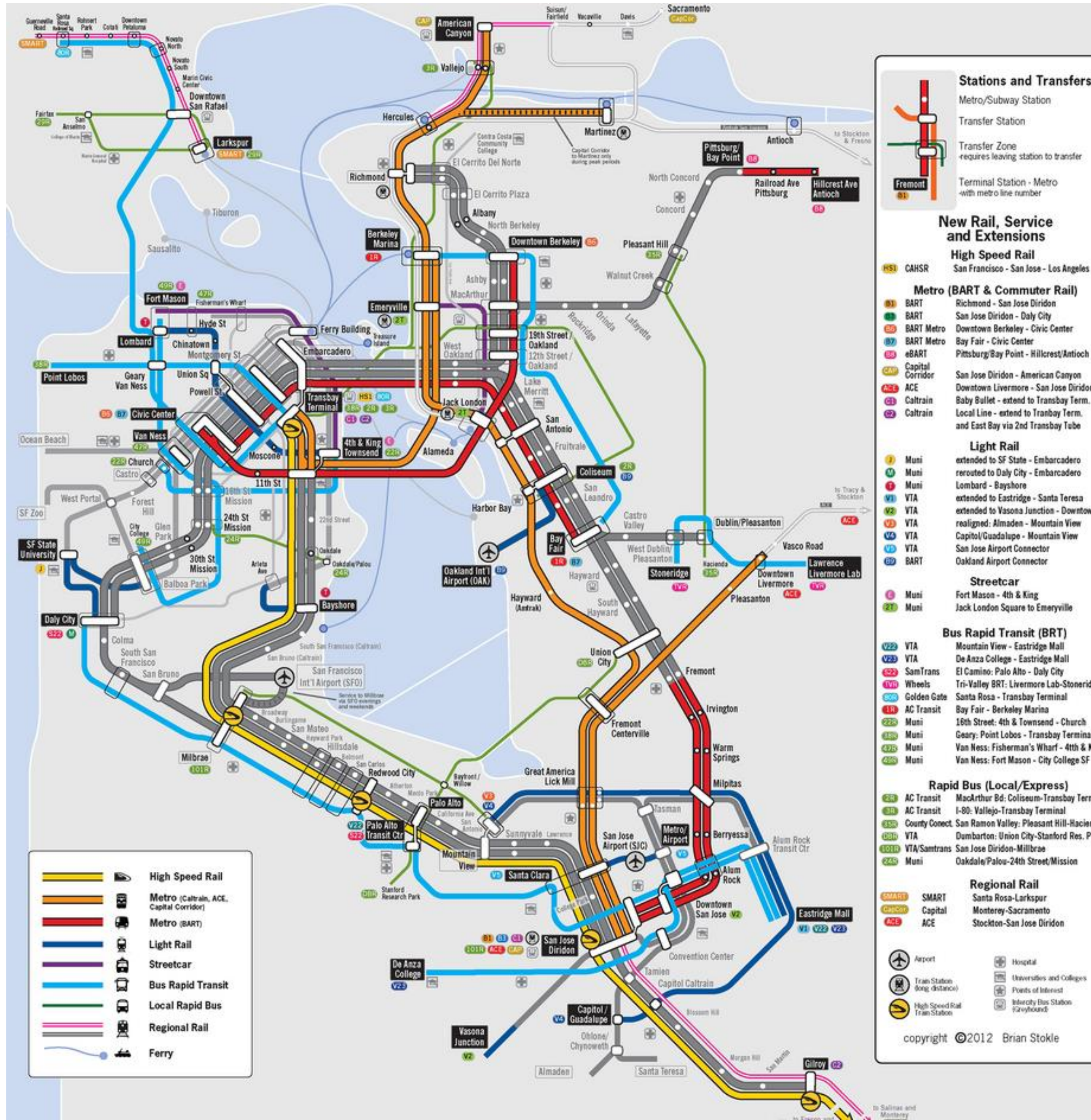
- I want to know the shortest path from one vertex (Gates) to all other vertices.

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

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

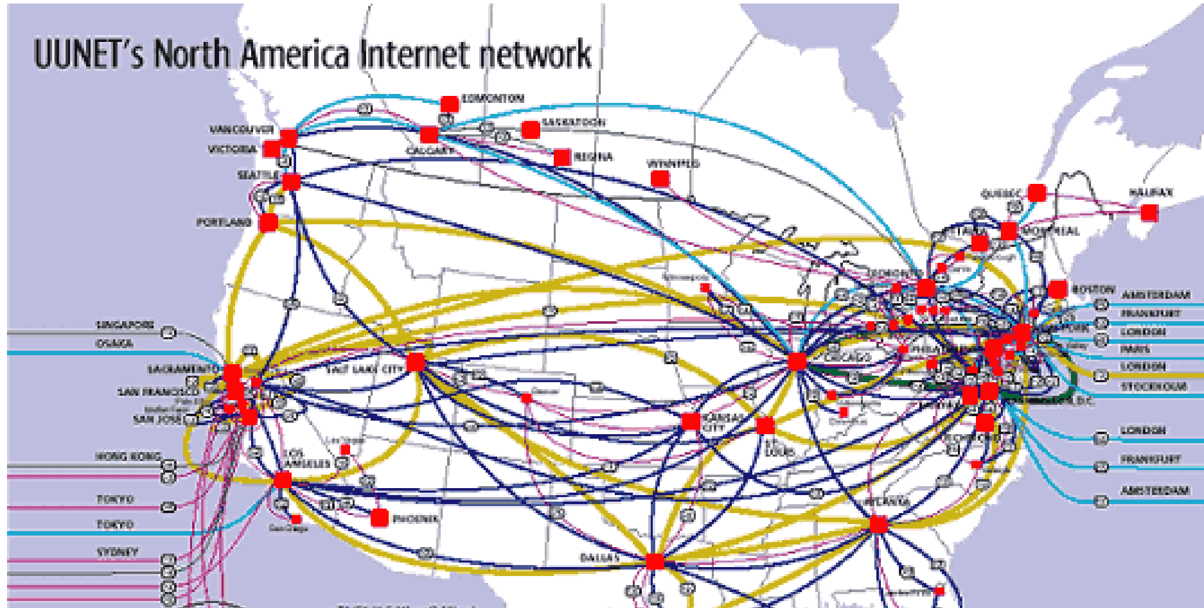
Example

- "**what is the shortest path from Palo Alto to [anywhere else]**" using BART, Caltrain, lightrail, MUNI, bus, Amtrak, bike, walking, uber/lyft.
- Edge weights have something to do with time, money, hassle. (They also change depending on my mood and traffic…).

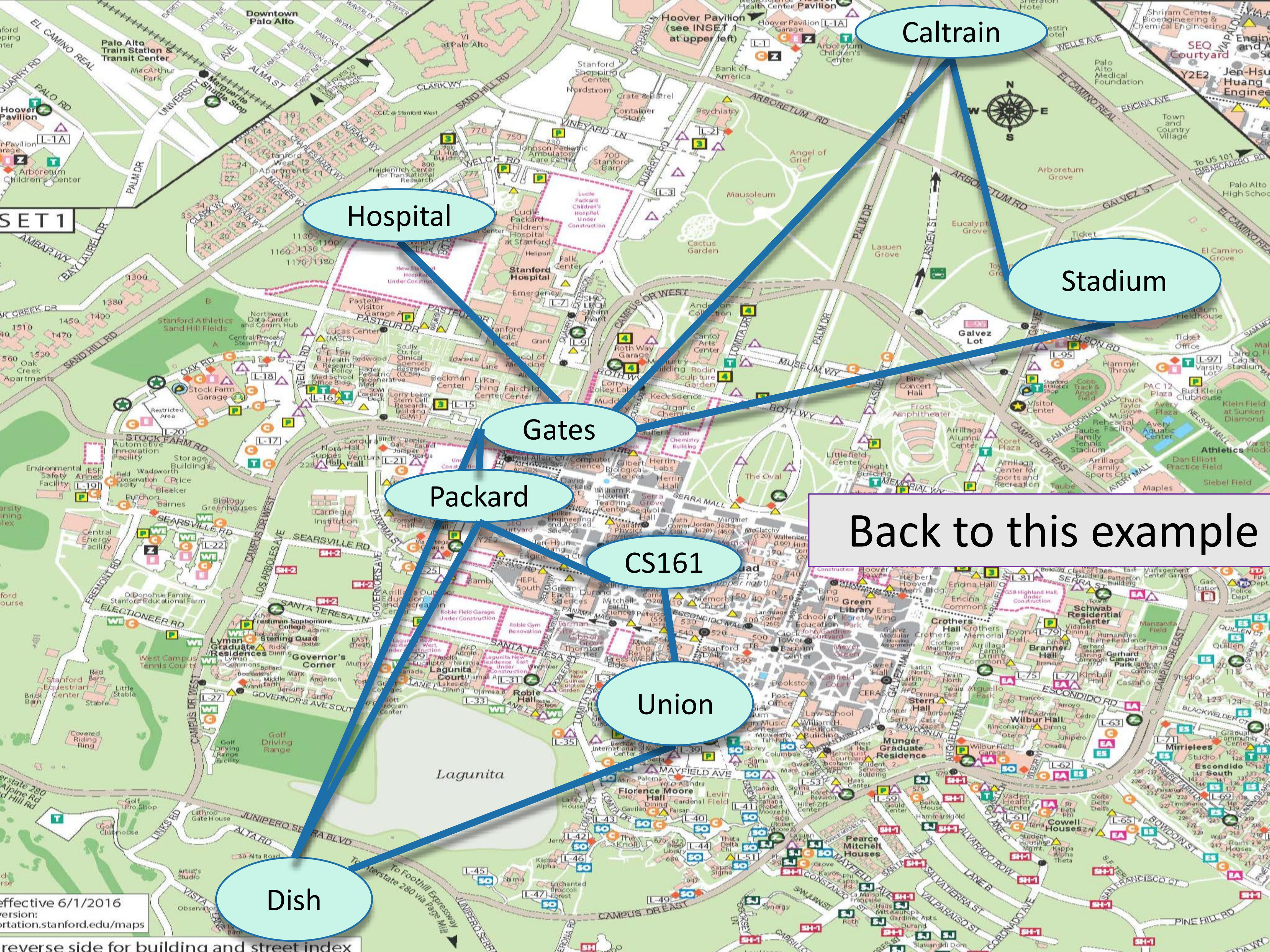


Example

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



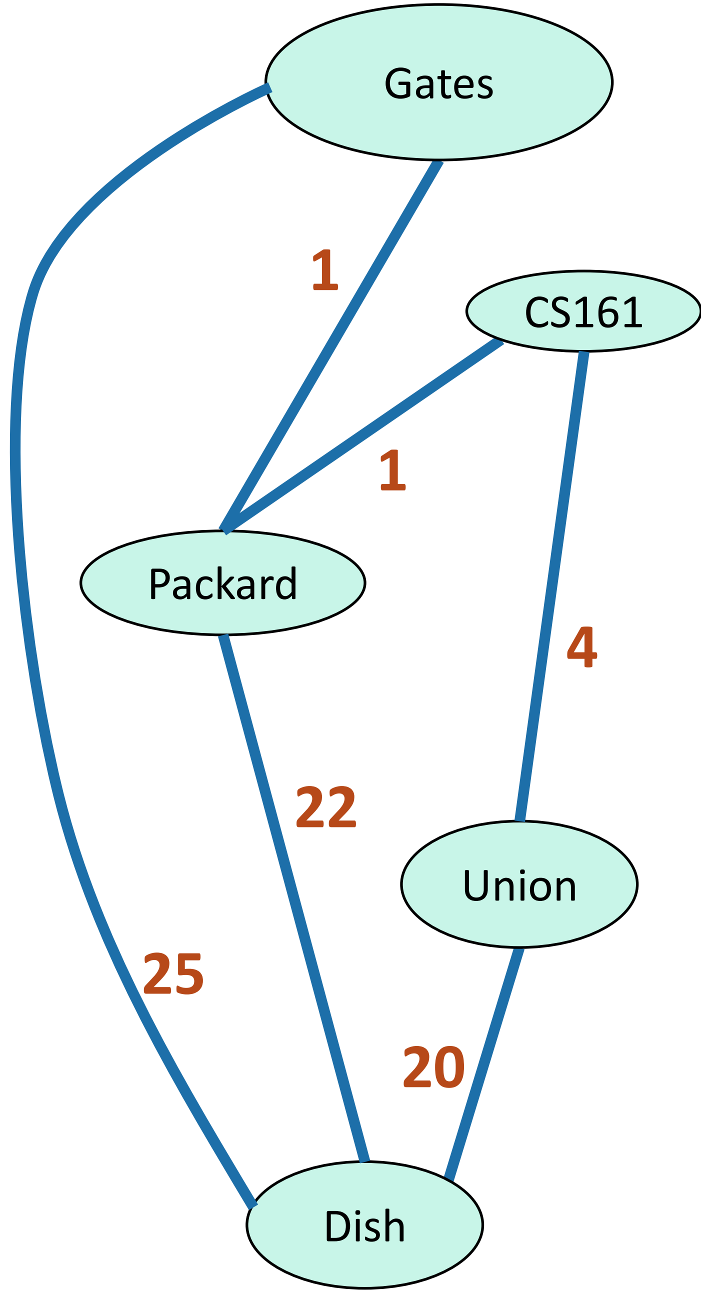

```
DN0a22a0e3:~ mary$ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1  [AS0] 10.34.160.2 (10.34.160.2)  38.168 ms  31.272 ms  28.841 ms
 2  [AS0] cwa-vrtr.sunet (10.21.196.28)  33.769 ms  28.245 ms  24.373 ms
 3  [AS32] 171.66.2.229 (171.66.2.229)  24.468 ms  20.115 ms  23.223 ms
 4  [AS32] hpr-svl-rtr-vlan8.sunet (171.64.255.235)  24.644 ms  24.962 ms  17
 5  [AS2152] hpr-svl-hpr2--stan-ge.cenic.net (137.164.27.161)  22.129 ms  4.9
 6  [AS2152] hpr-lax-hpr3--svl-hpr3-100ge.cenic.net (137.164.25.73)  12.125 m
 7  [AS2152] hpr-i2--lax-hpr2-r&e.cenic.net (137.164.26.201)  40.174 ms  38.3
 8  [AS0] et-4-0-0.4079.sdn-sw.lasv.net.internet2.edu (162.252.70.28)  46.573
 9  [AS0] et-5-1-0.4079.rtsw.salt.net.internet2.edu (162.252.70.31)  30.424 m
10  [AS0] et-4-0-0.4079.sdn-sw.denv.net.internet2.edu (162.252.70.8)  47.454
11  [AS0] et-4-1-0.4079.rtsw.kans.net.internet2.edu (162.252.70.11)  70.825 m
12  [AS0] et-4-1-0.4070.rtsw.chic.net.internet2.edu (198.71.47.206)  77.937 m
13  [AS0] et-0-1-0.4079.sdn-sw.ashb.net.internet2.edu (162.252.70.60)  77.682
14  [AS0] et-4-1-0.4079.rtsw.wash.net.internet2.edu (162.252.70.65)  71.565 m
15  [AS21320] internet2-gw.mx1.lon.uk.geant.net (62.40.124.44)  154.926 ms  1
16  [AS21320] ae0.mx1.lon2.uk.geant.net (62.40.98.79)  146.565 ms  146.604 ms
17  [AS21320] ae0.mx1.par.fr.geant.net (62.40.98.77)  153.289 ms  184.995 ms
18  [AS21320] ae2.mx1.gen.ch.geant.net (62.40.98.153)  160.283 ms  160.104 ms
19  [AS21320] swice1-100ge-0-3-0-1.switch.ch (62.40.124.22)  162.068 ms  160.
20  [AS559] swizh1-100ge-0-1-0-1.switch.ch (130.59.36.94)  165.824 ms  164.2
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
```



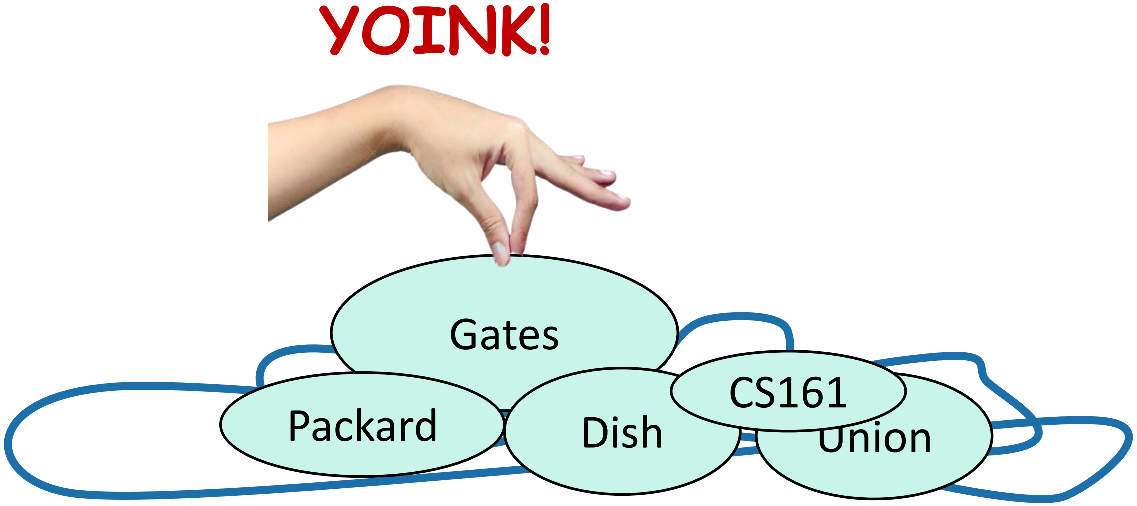
Back to this example

Dijkstra's algorithm

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

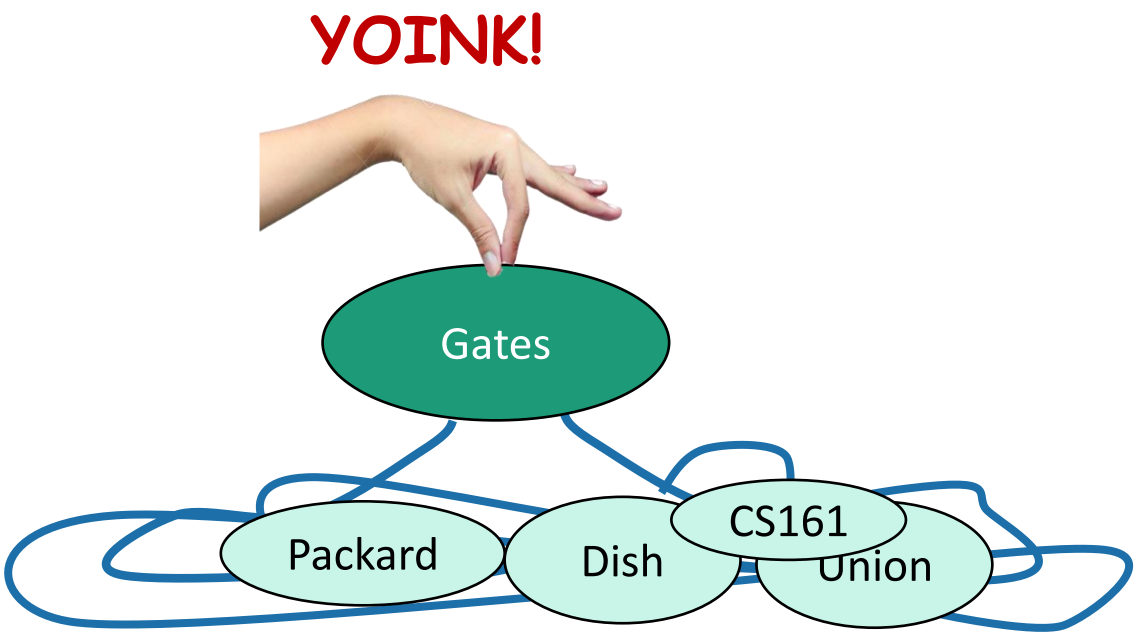


Dijkstra
intuition

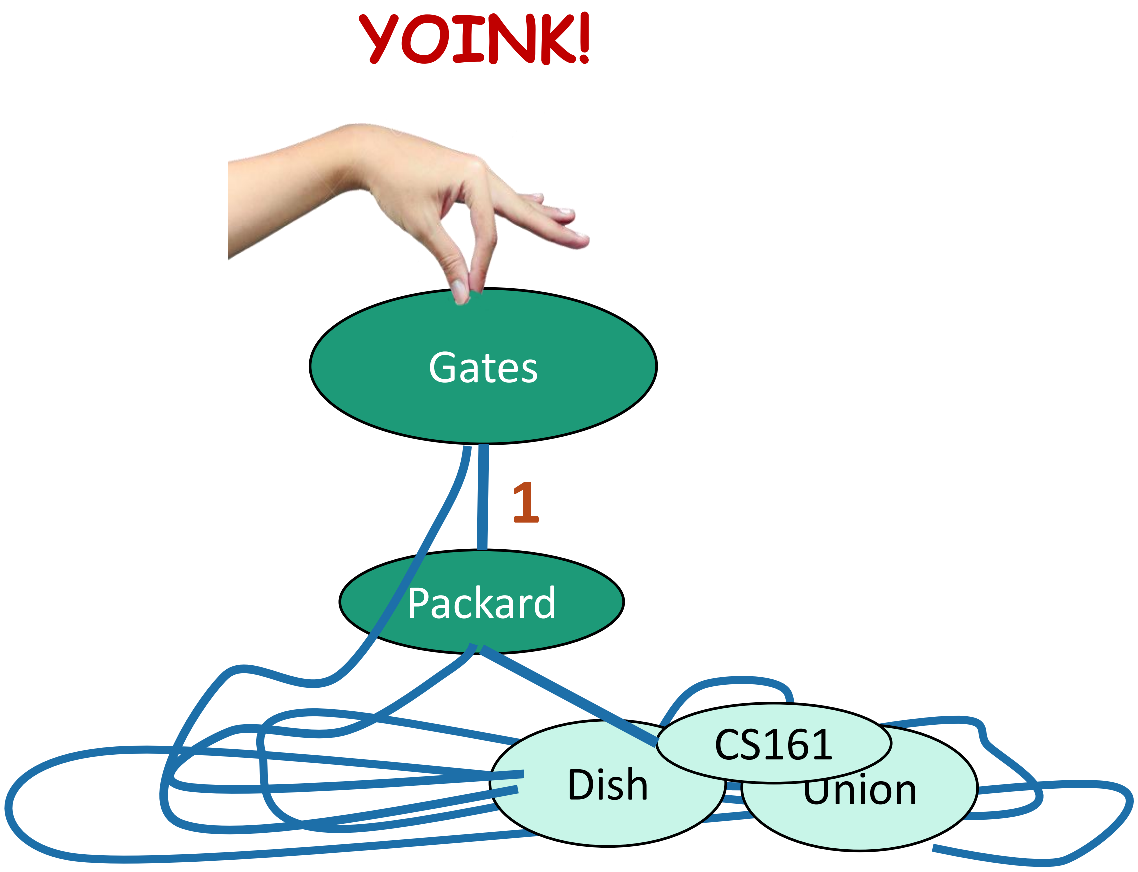


Dijkstra
intuition

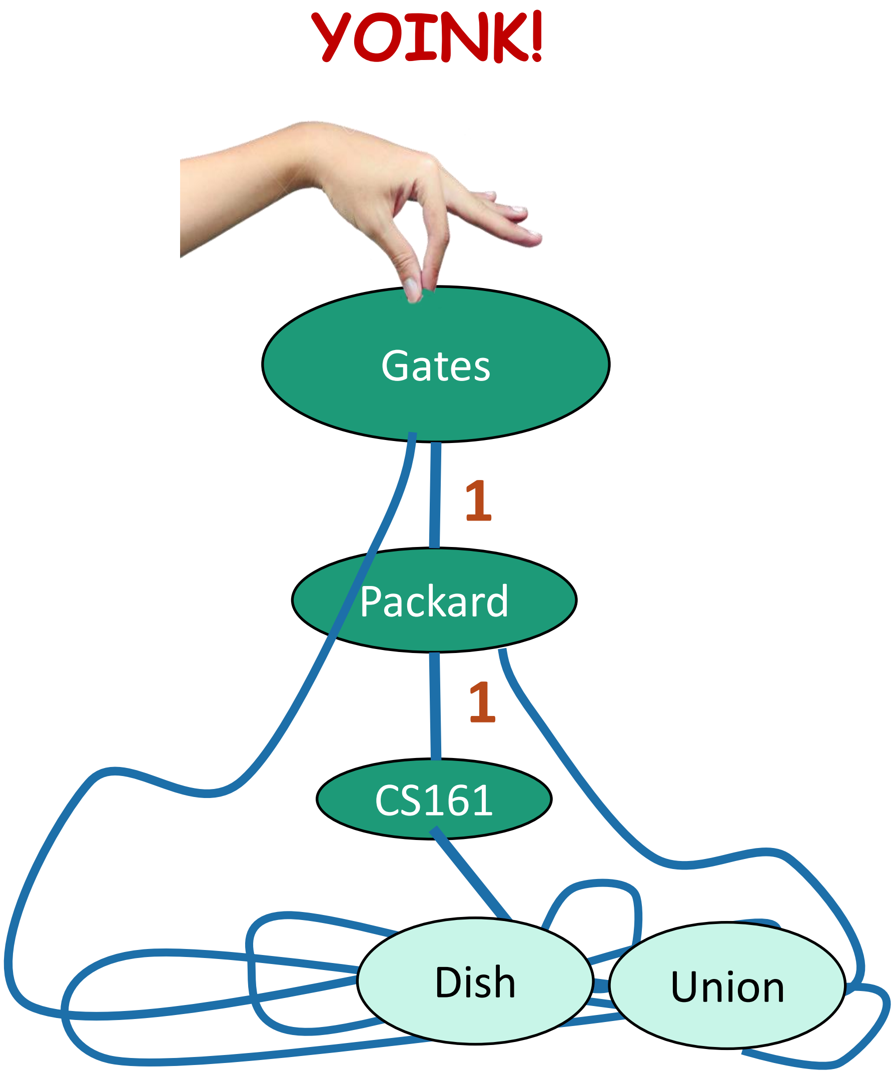
A vertex is done when it’s not on the ground anymore.



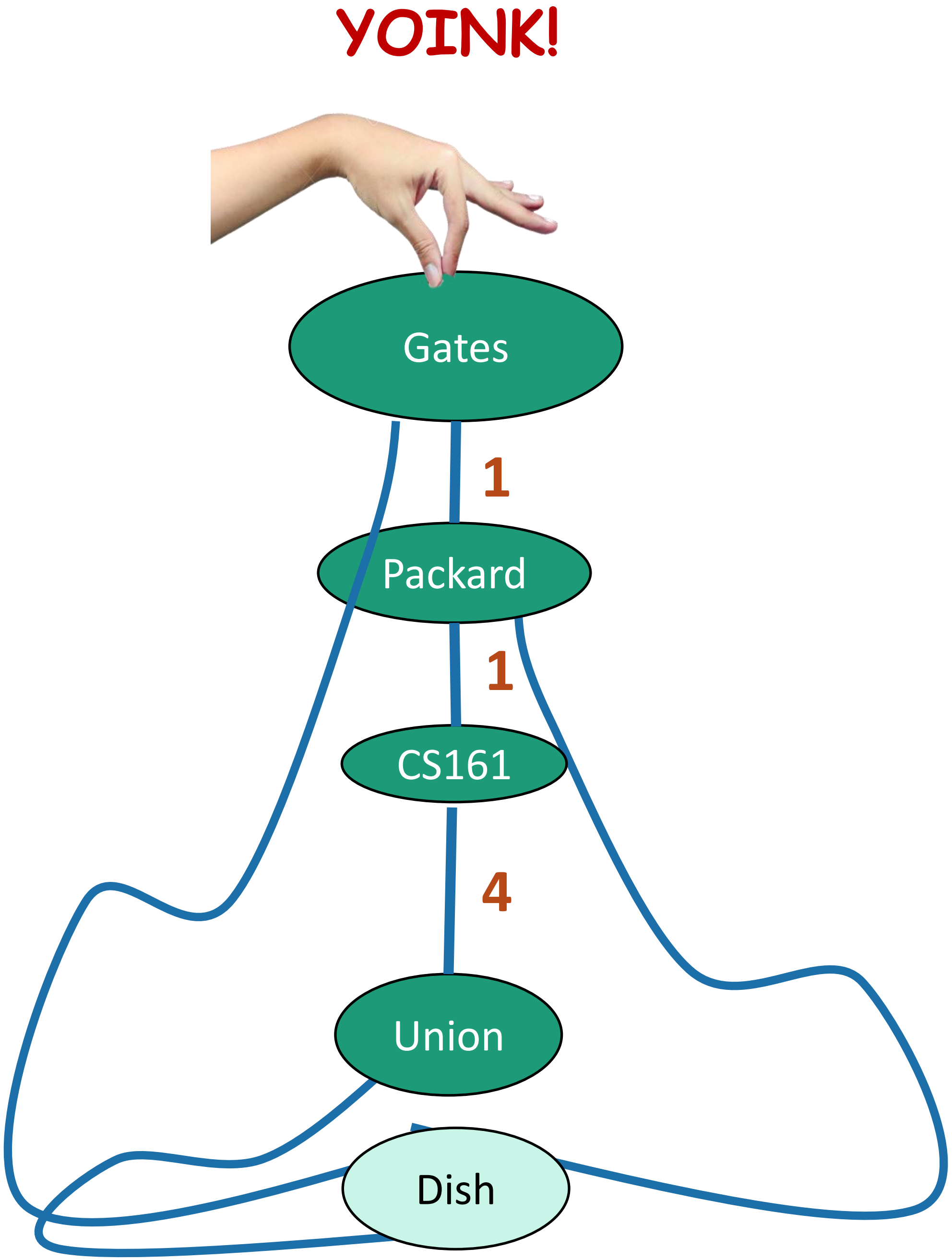
Dijkstra
intuition



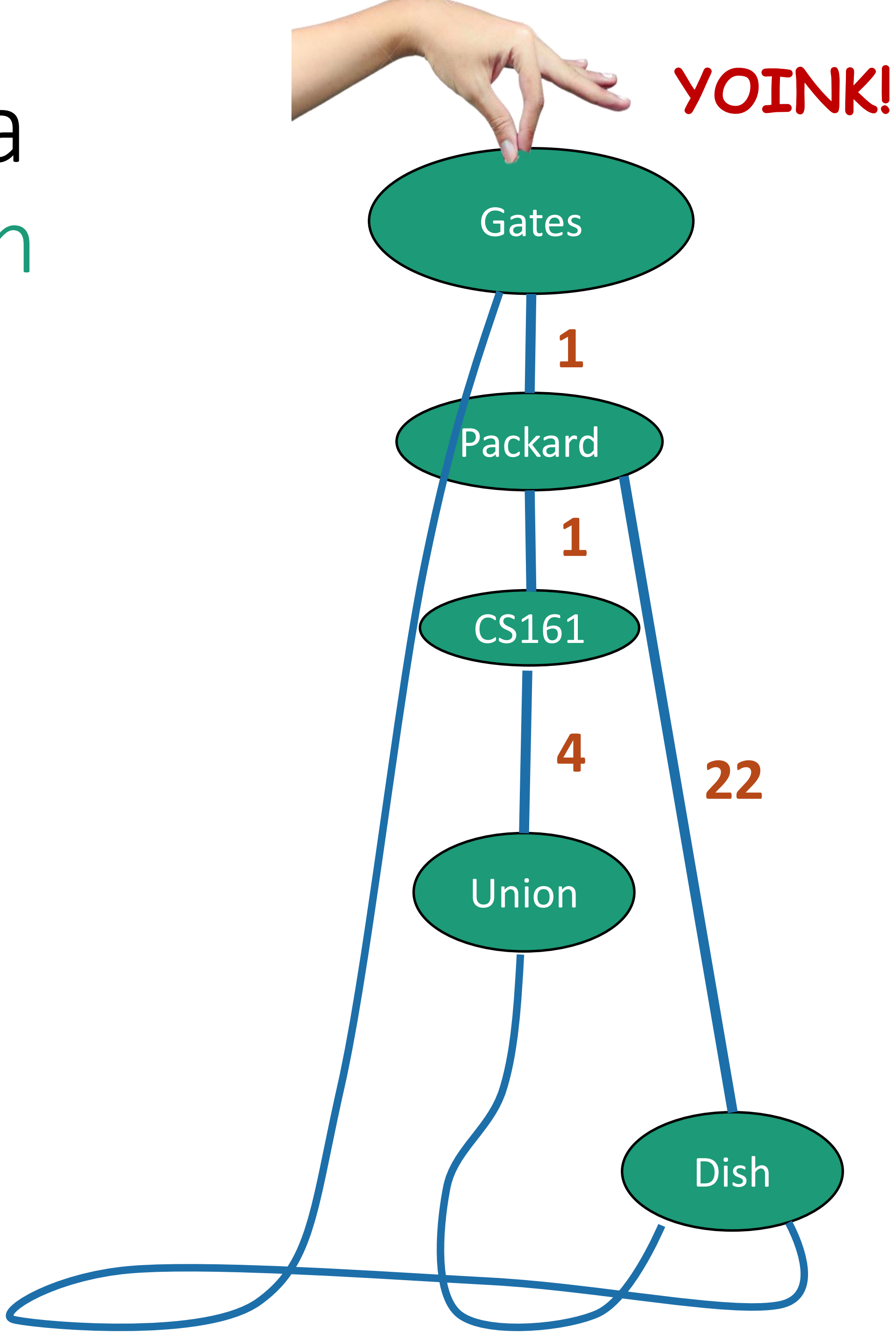
Dijkstra
intuition



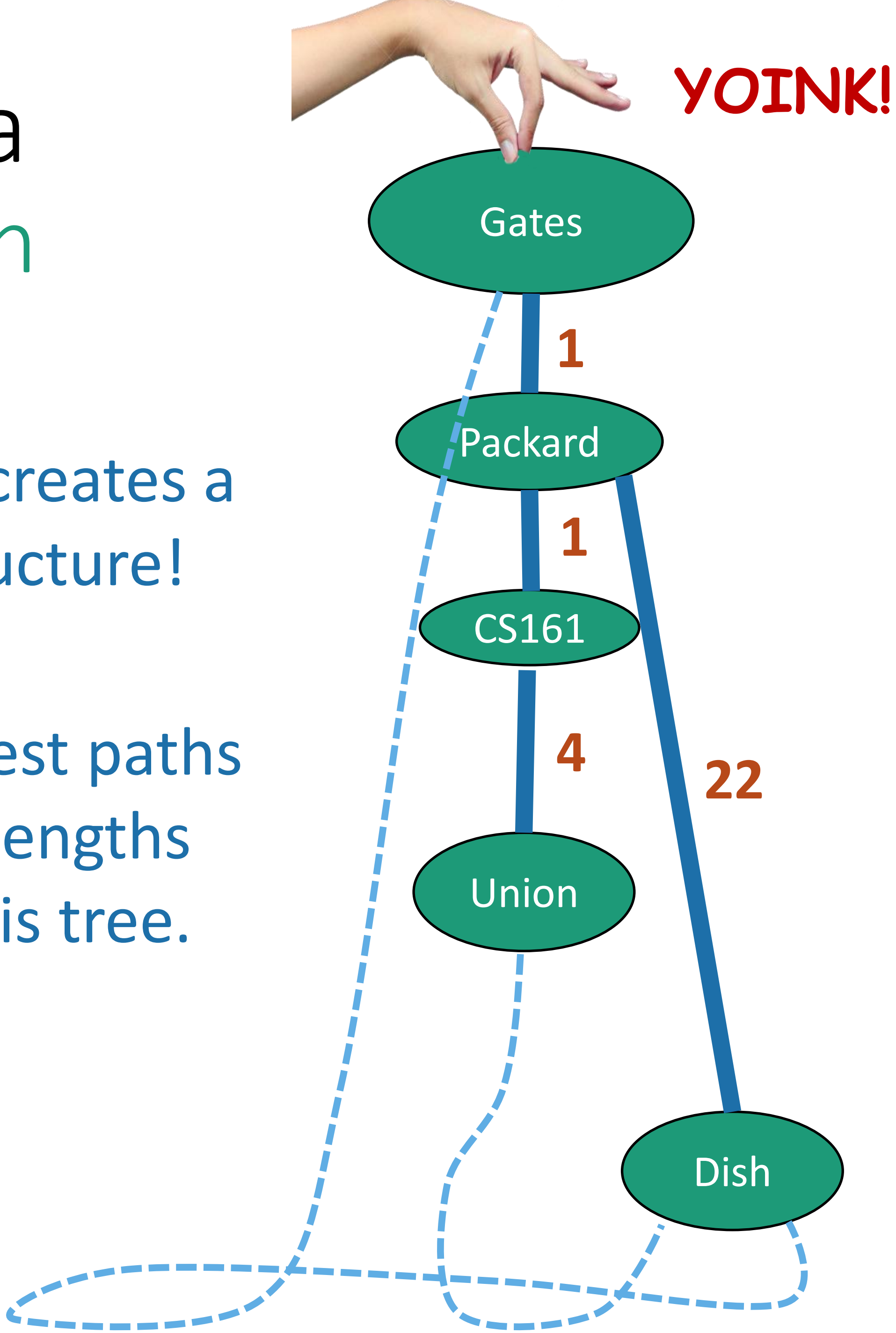
Dijkstra
intuition



Dijkstra
intuition



Dijkstra
intuition

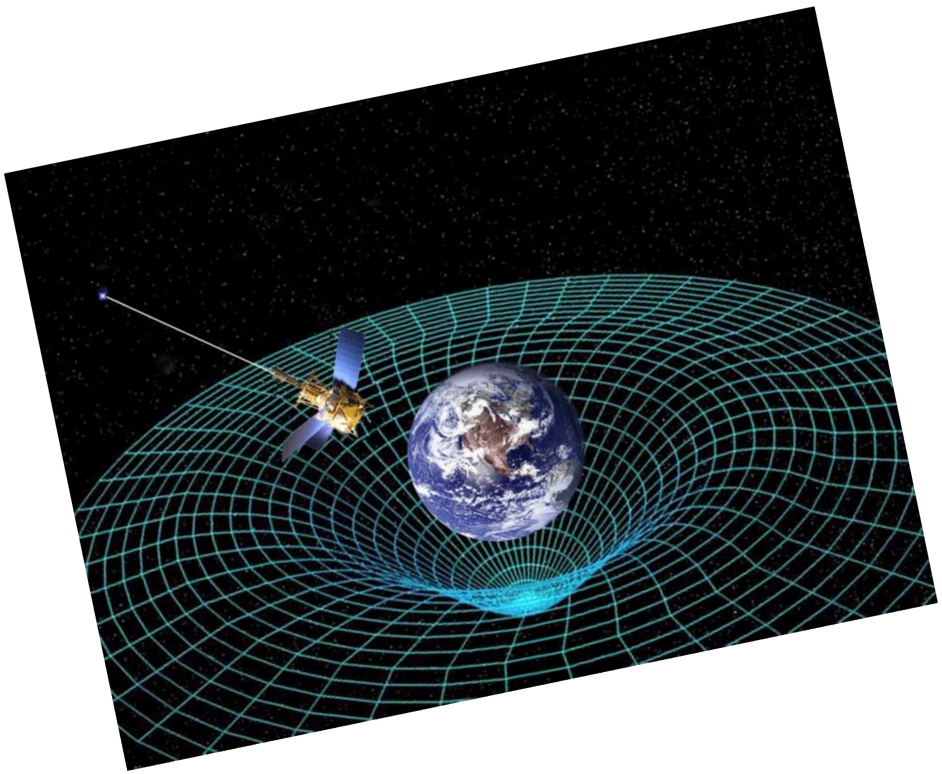


This also creates a tree structure!

The shortest paths are the lengths along this tree.

How do we actually implement this?

- **Without** string and gravity?



Dijkstra by example

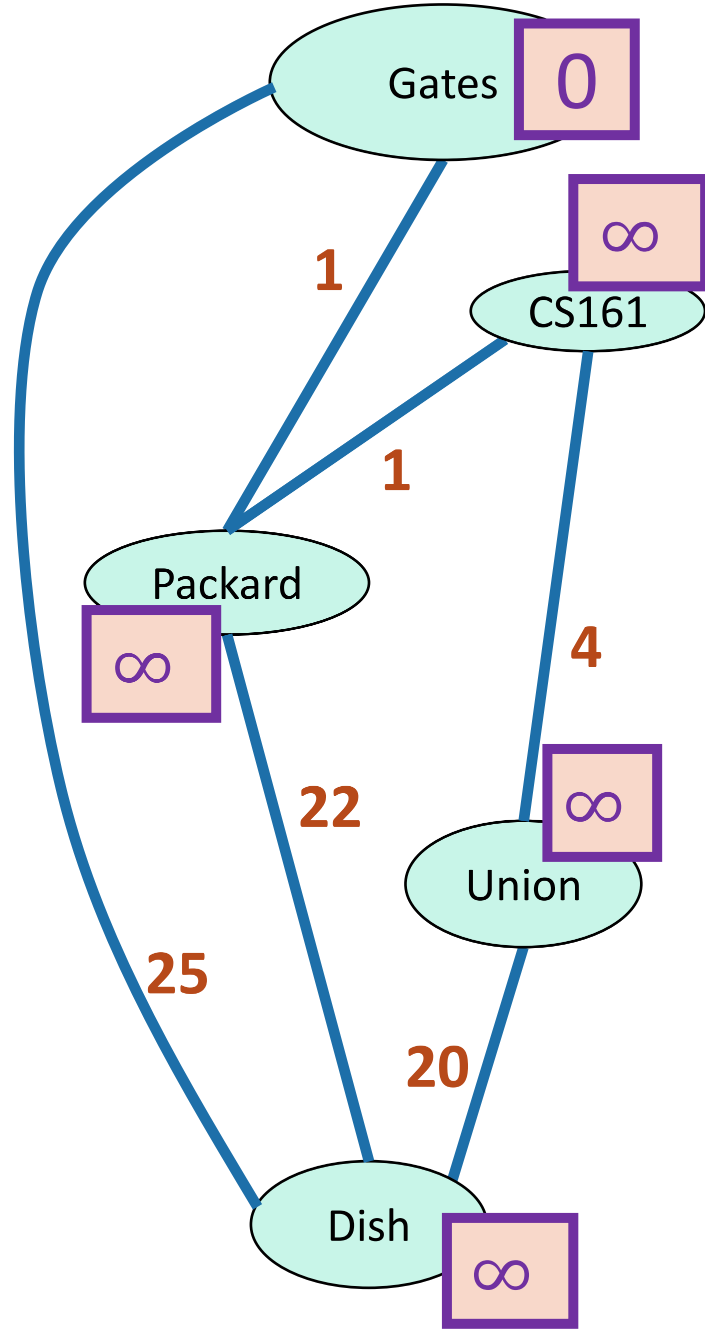
How far is a node from Gates?

I'm not sure yet

I'm sure

X **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?

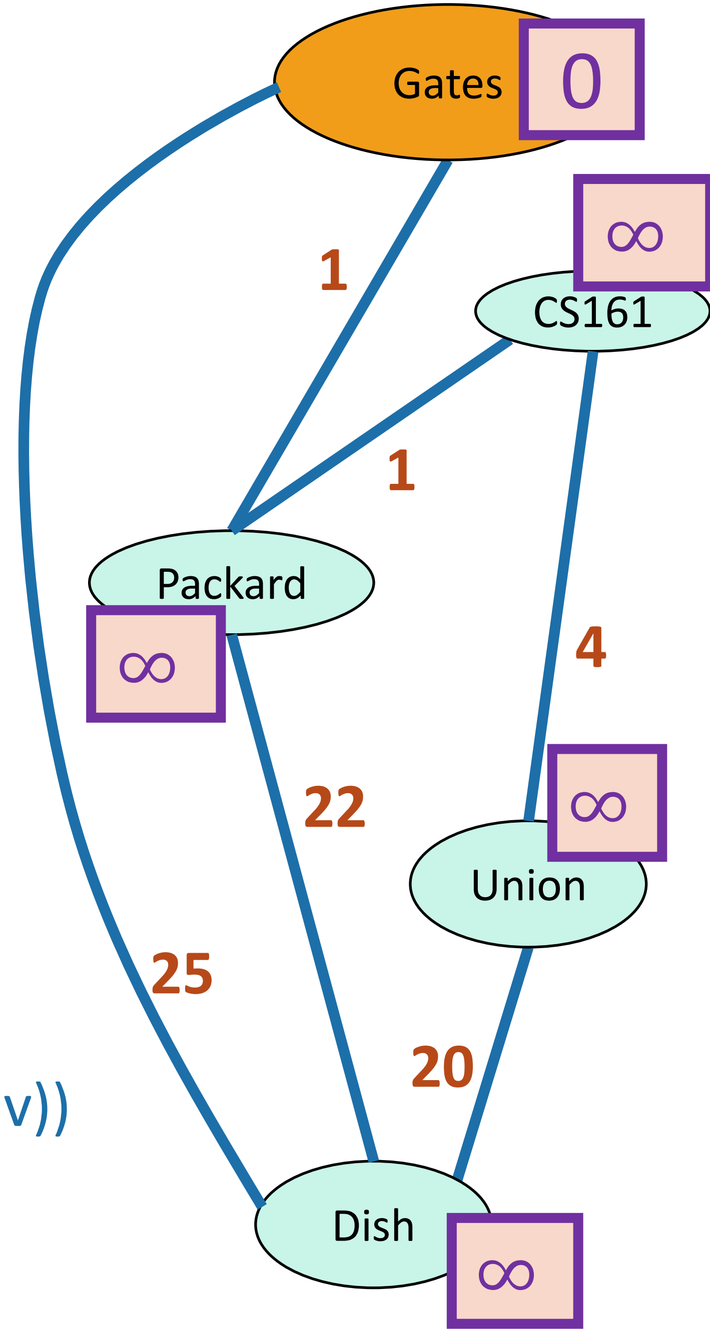
I'm not sure yet

I'm sure

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

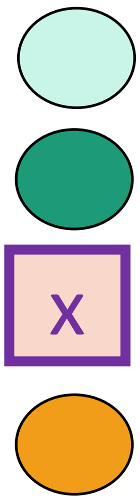
Current node u

- 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?



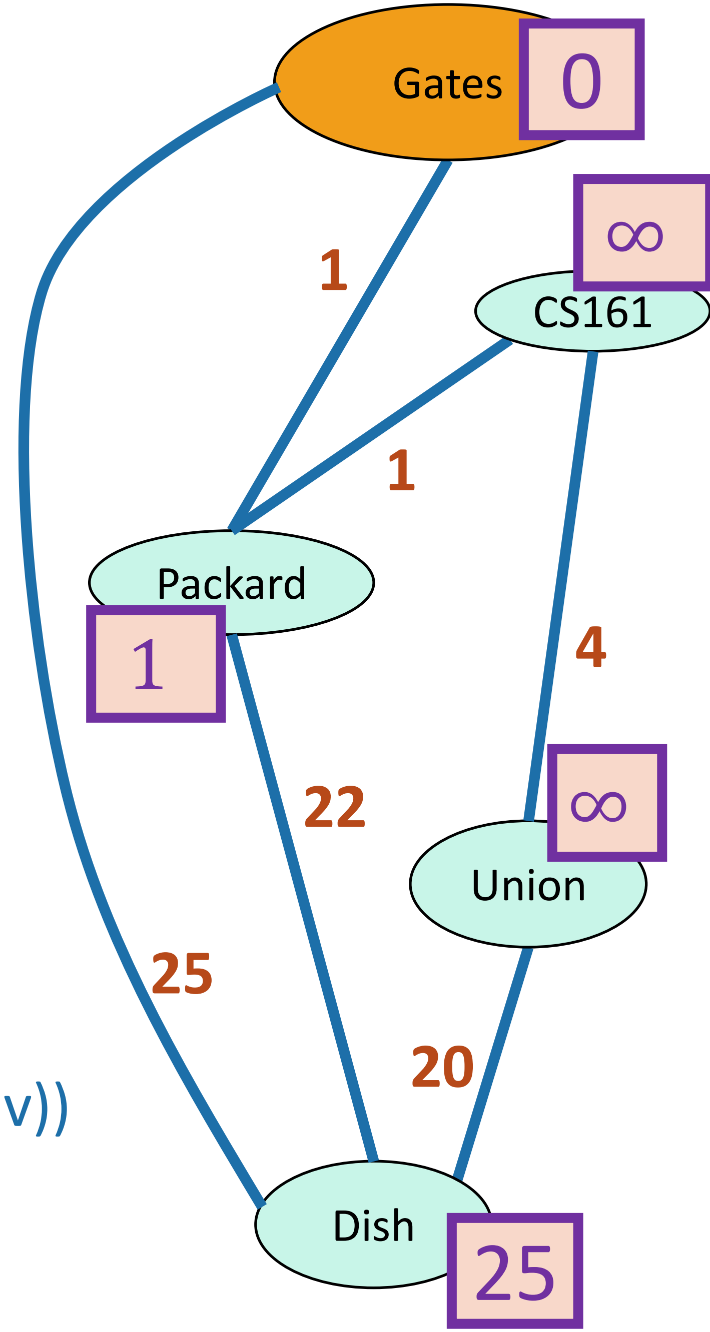
I'm not sure yet

I'm sure

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

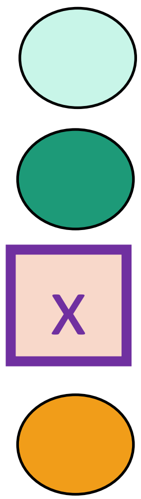
Current node u

- 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?



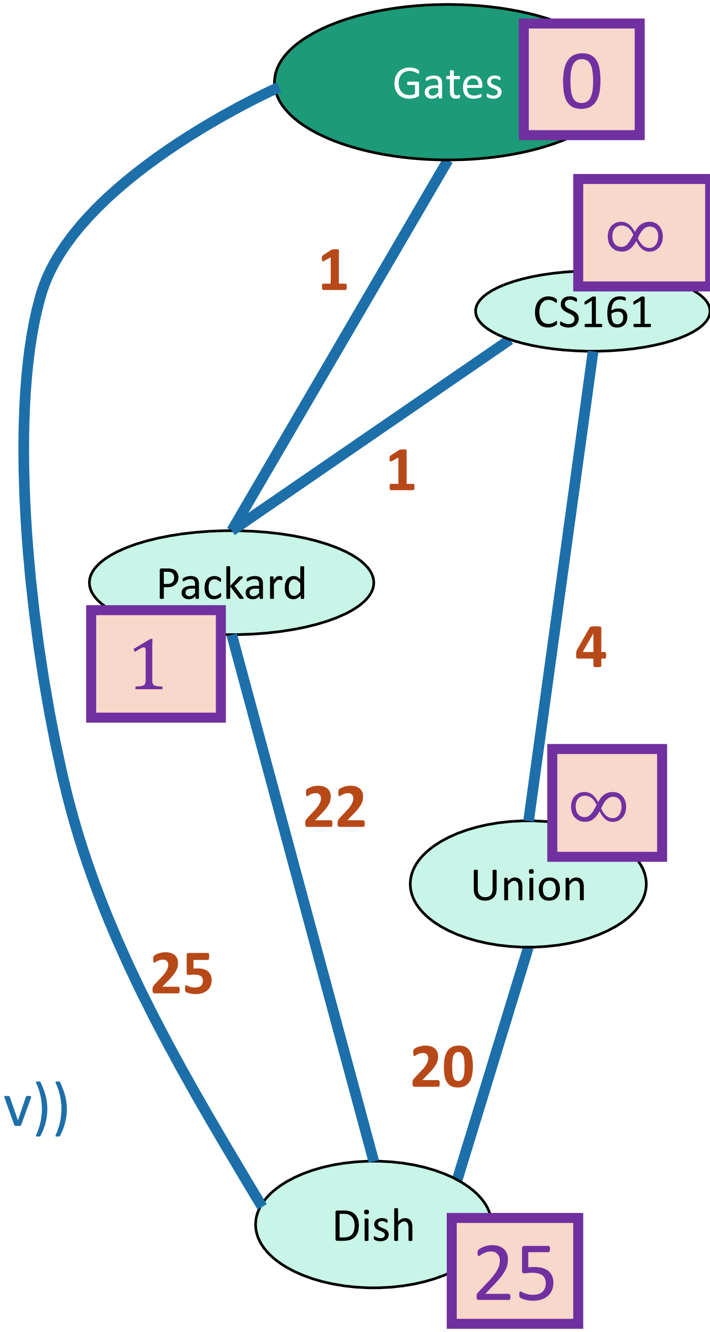
I'm not sure yet

I'm sure

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

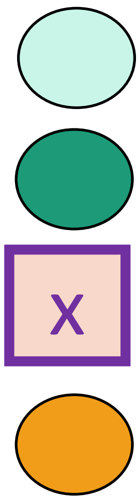
Current node u

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



Dijkstra by example

How far is a node from Gates?



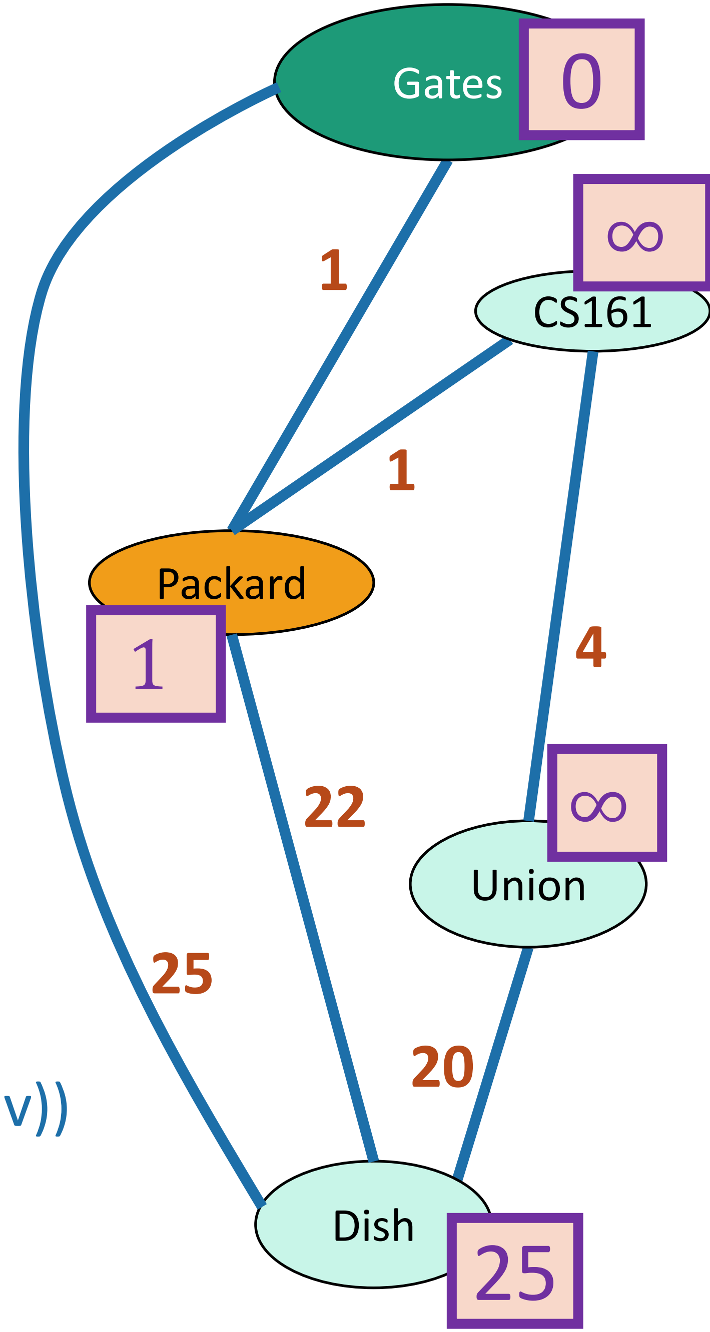
I'm not sure yet

I'm sure

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

Current node u

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



Dijkstra by example

How far is a node from Gates?

I’m not sure yet

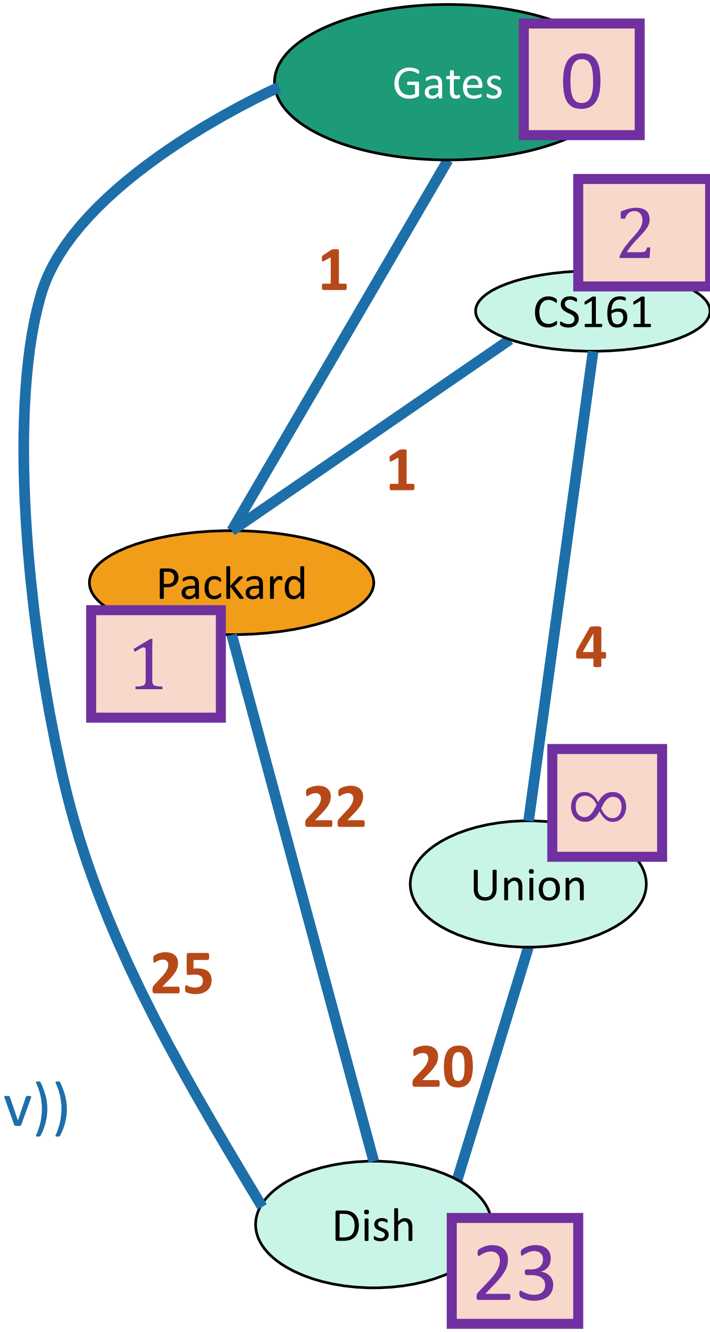
I’m sure

x

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

Current node u

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



Dijkstra by example

How far is a node from Gates?

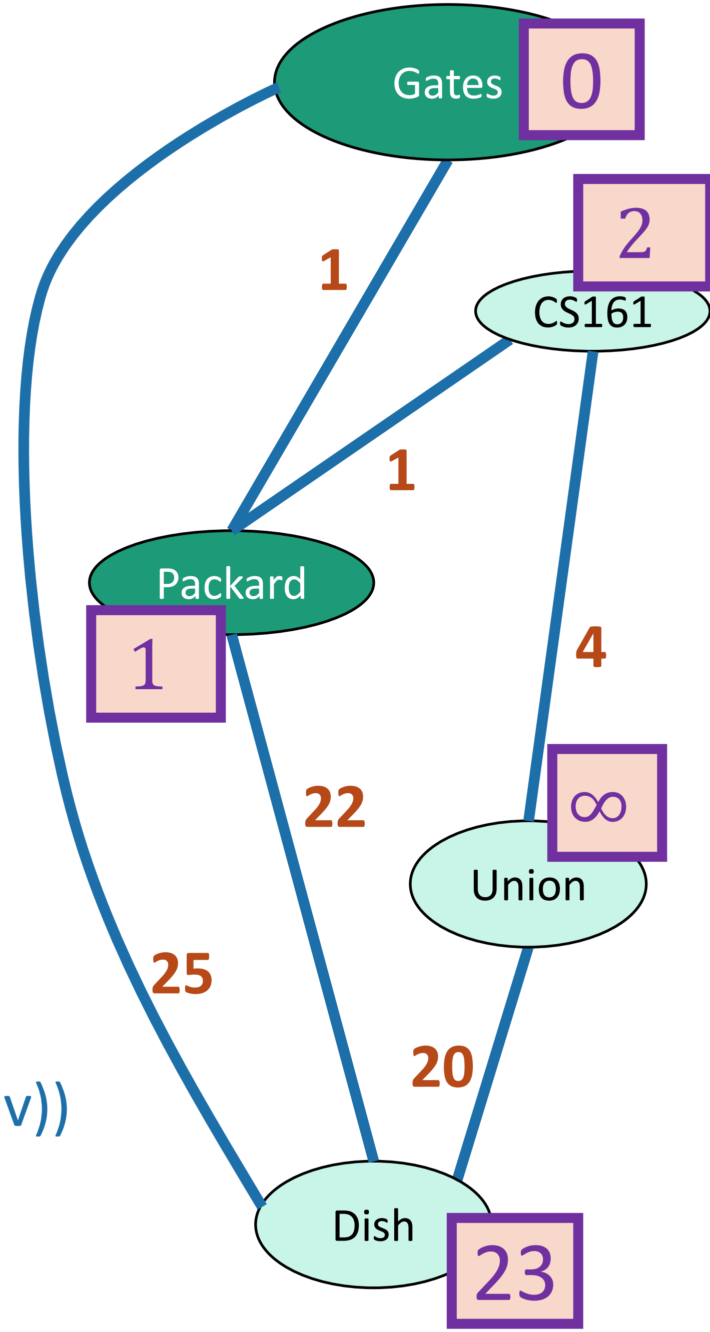
I'm not sure yet

I'm sure

x

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

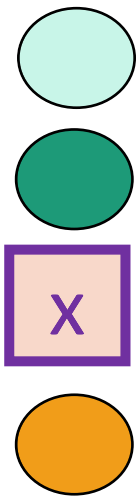
Current node u



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

Dijkstra by example

How far is a node from Gates?



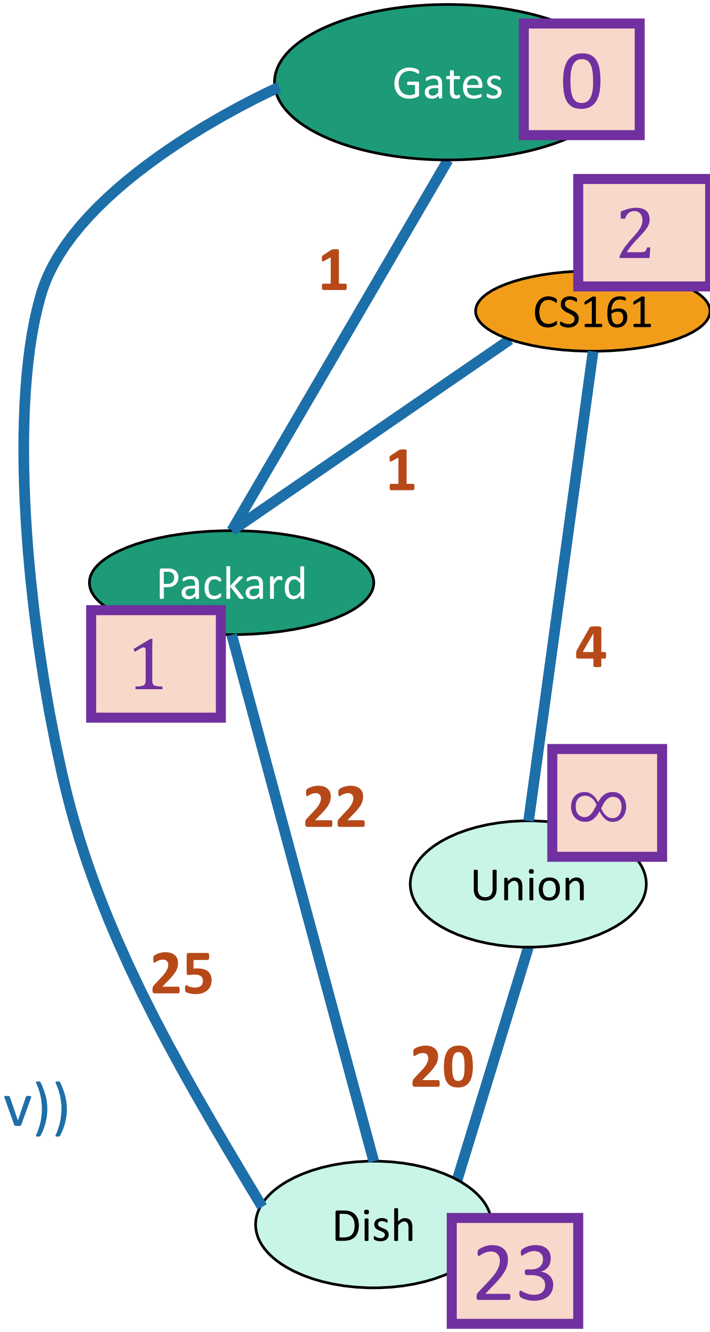
I'm not sure yet

I'm sure

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

Current node u

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



Dijkstra by example

How far is a node from Gates?

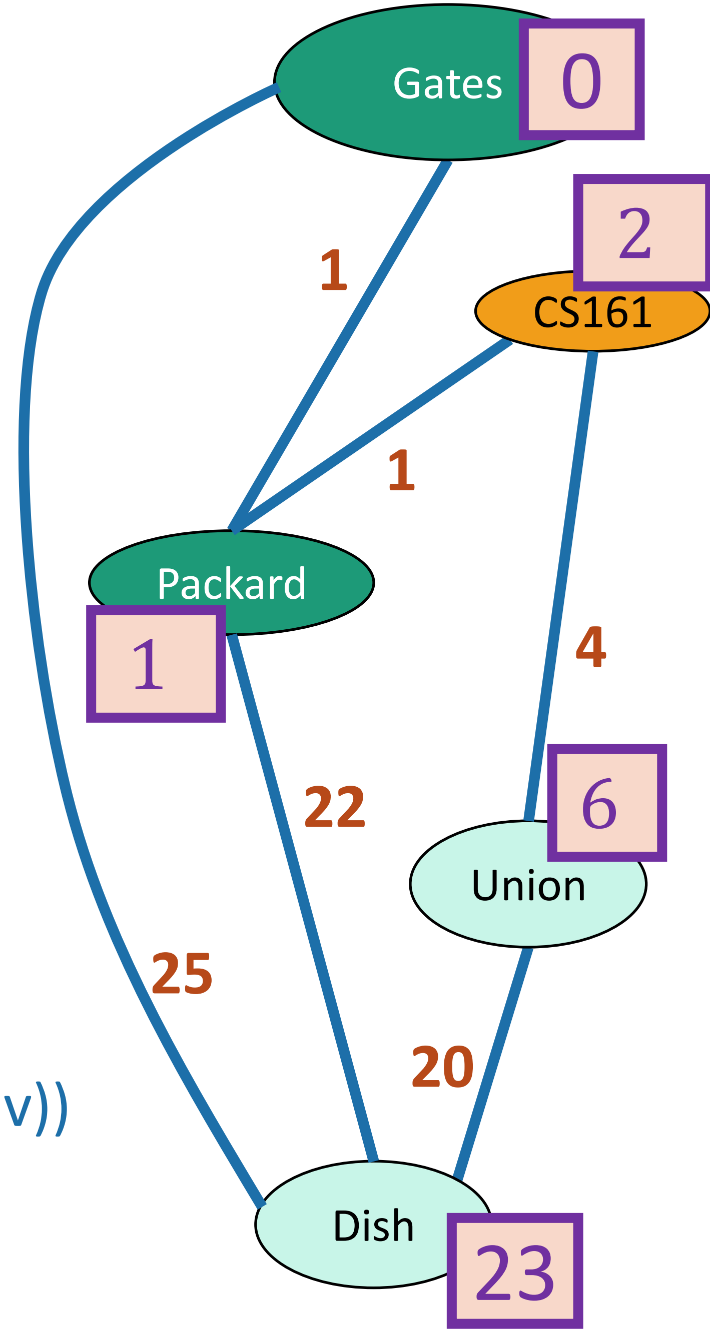
I'm not sure yet

I'm sure

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

Current node u

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



Dijkstra by example

How far is a node from Gates?

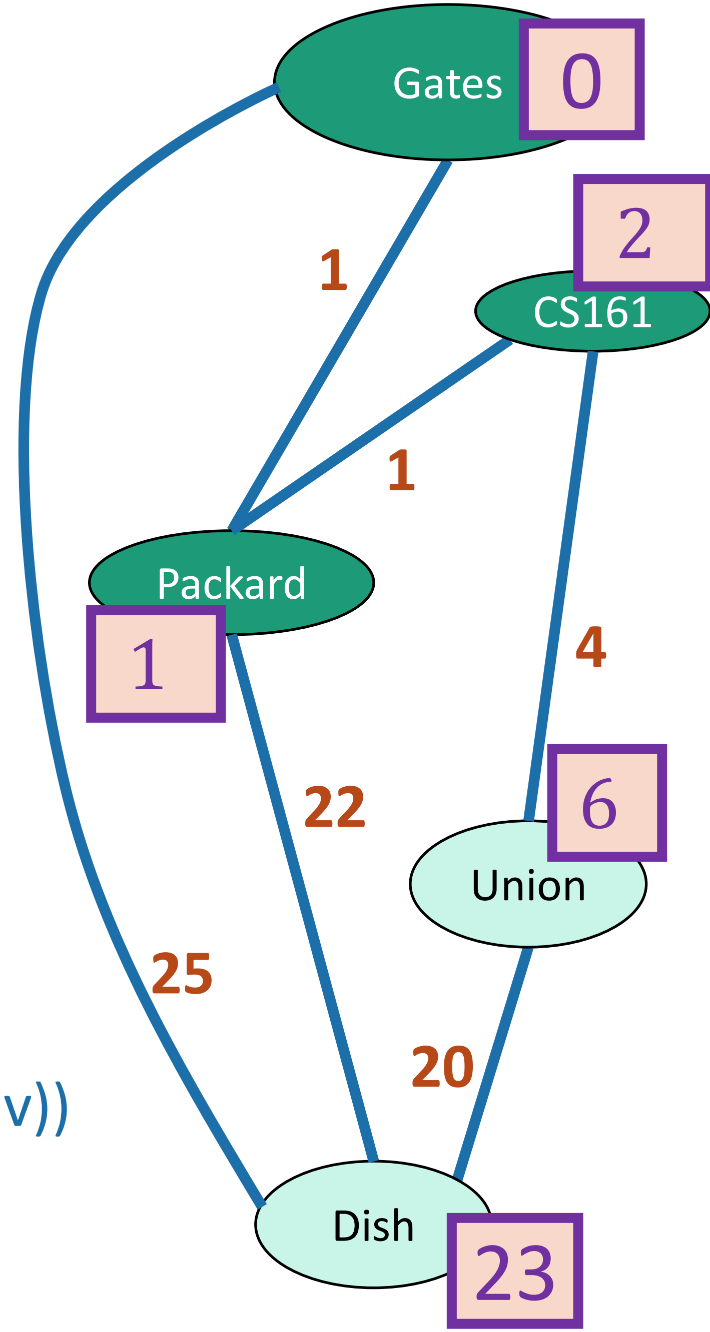
I'm not sure yet

I'm sure

x

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?

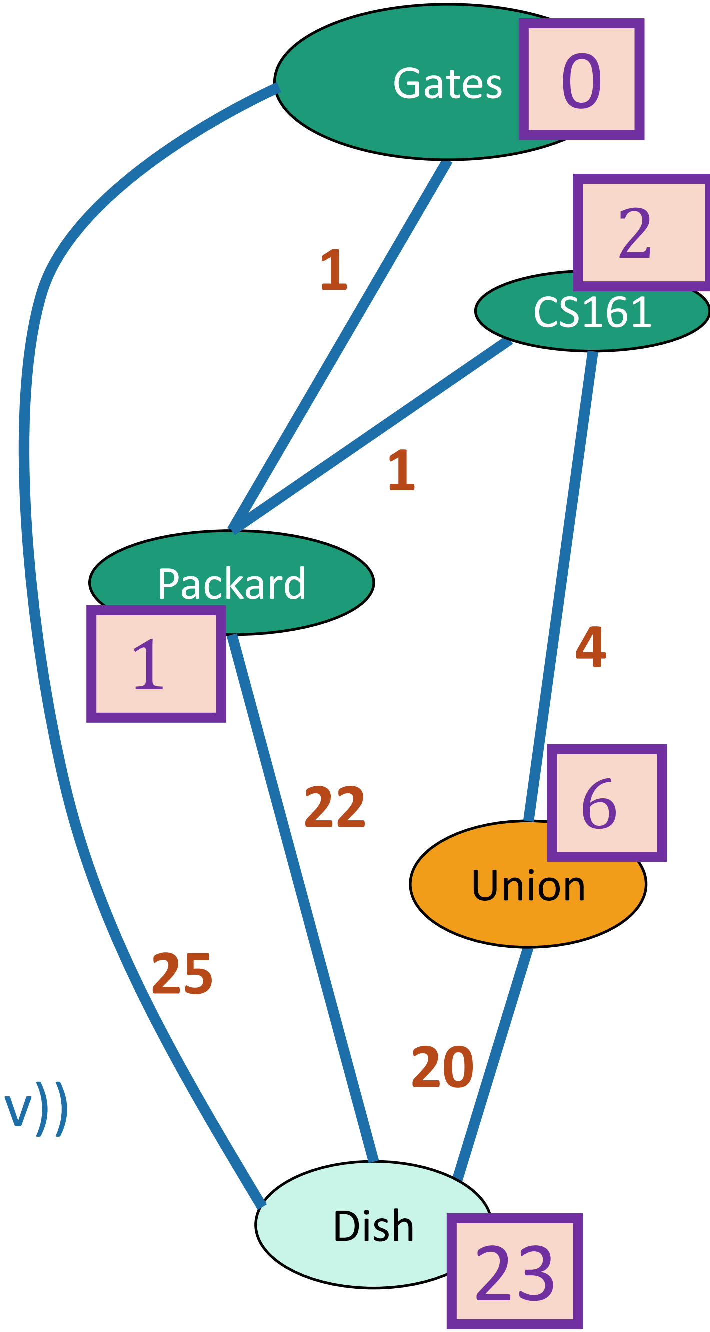
I'm not sure yet

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

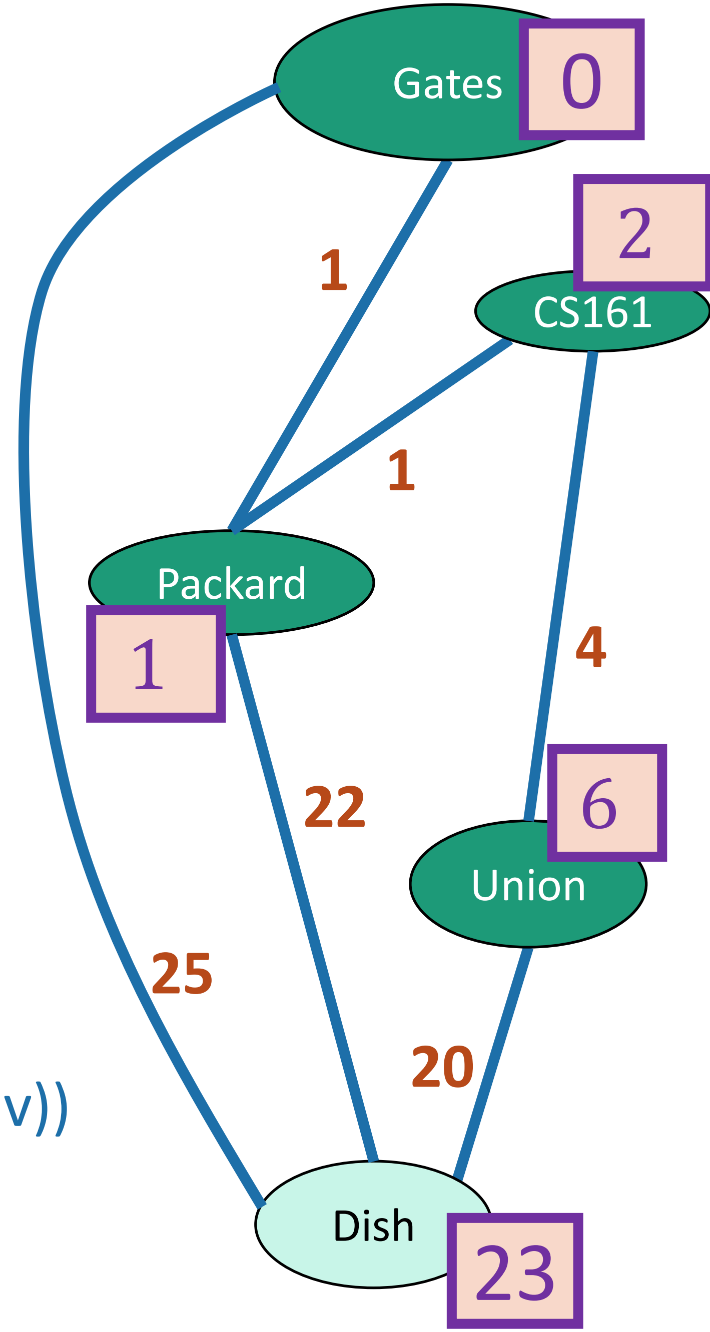
I'm sure

x

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

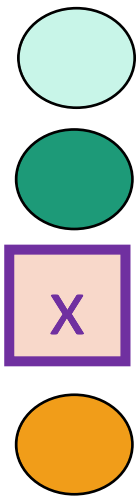
Current node u

- 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

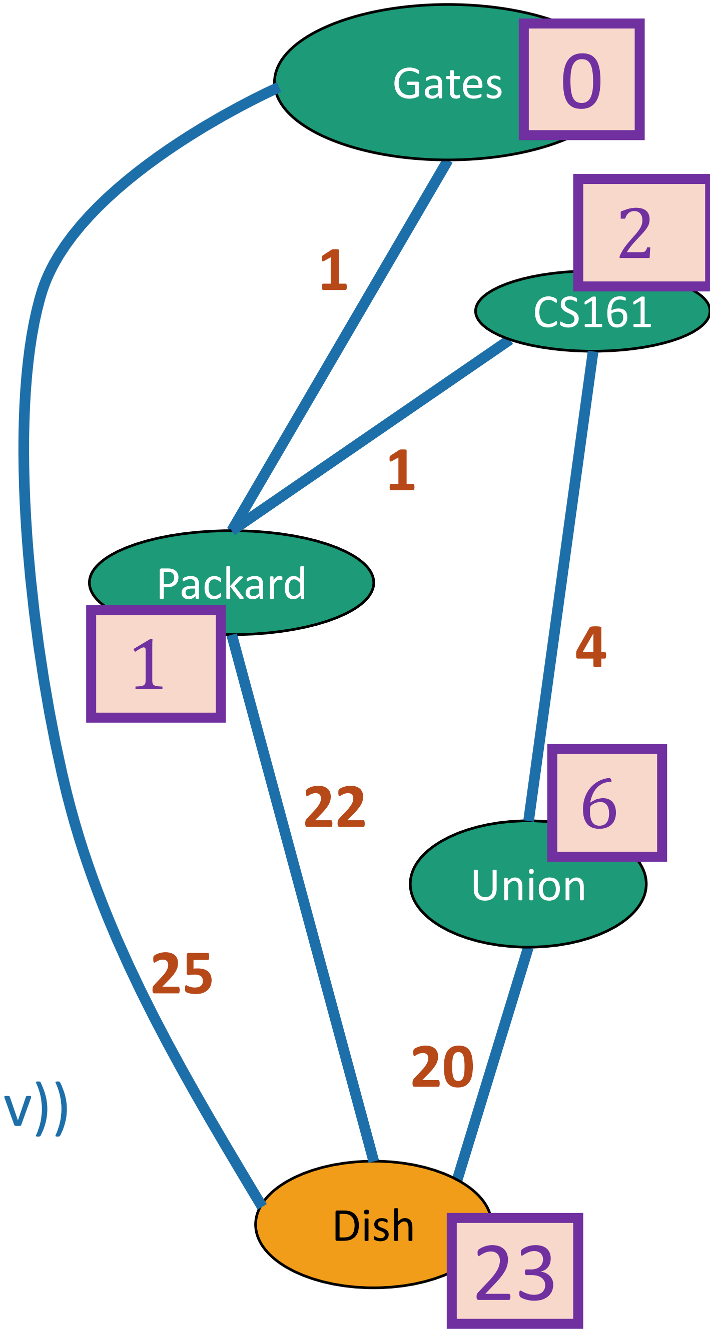
I'm sure

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

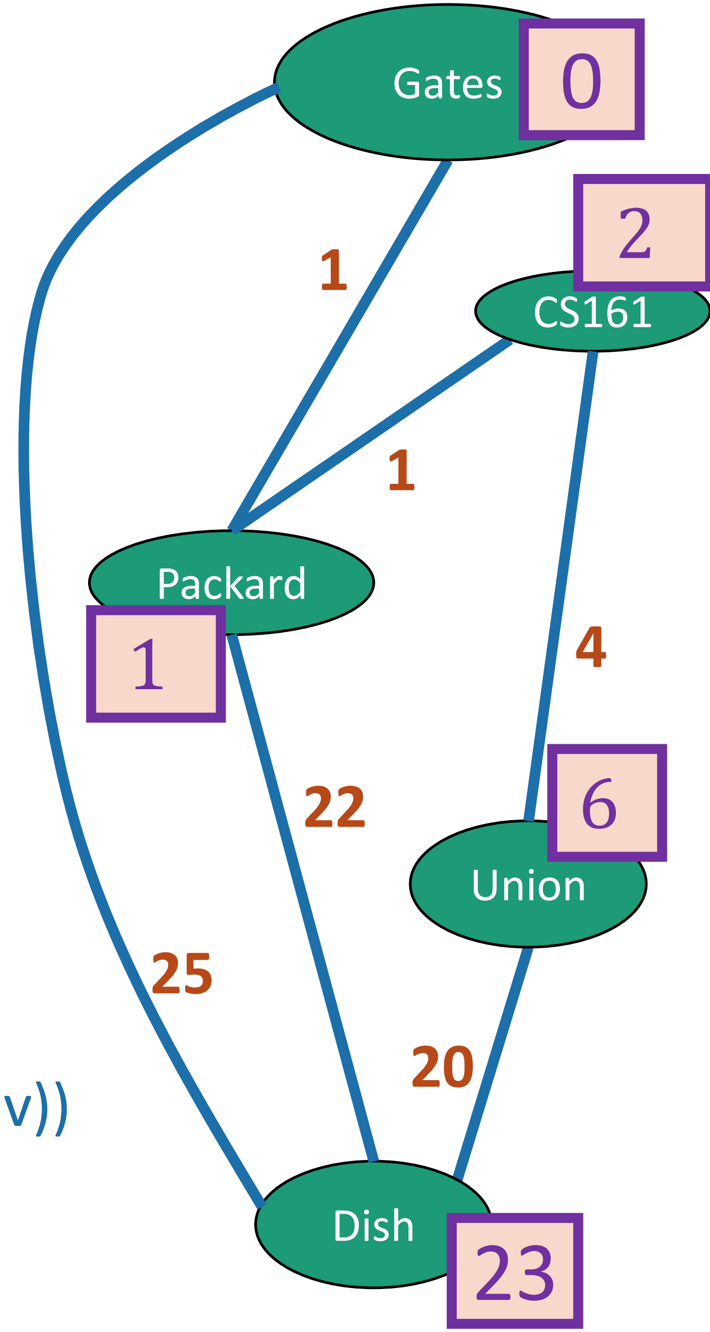
I'm sure

x

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**
- $d[v] = \infty$ 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:
    - $d[v] \leftarrow \min(d[v] , d[u] + \text{edgeWeight}(u,v))$
  - Mark u as **sure**.
- Now $d(s, v) = d[v]$

As usual

- Does it work?
  - Yes.
- Is it fast?
  - Depends on how you implement it.

Why does this work?

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

Let's rename "Gates" to "**s**", our starting vertex.

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

- **Claims 1 and 2** imply the **theorem.**
  - By the time we are **sure** about v, **d[v] = d(s,v).**
  - **d[v]** never increases, so after v is **sure**, **d[v]** stops changing.
  - All vertices are eventually **sure**. (Stopping condition in algorithm)
  - So all vertices end up with **d[v] = d(s,v).**

Next let's prove the claims!

Claim 1

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

Informally:

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

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

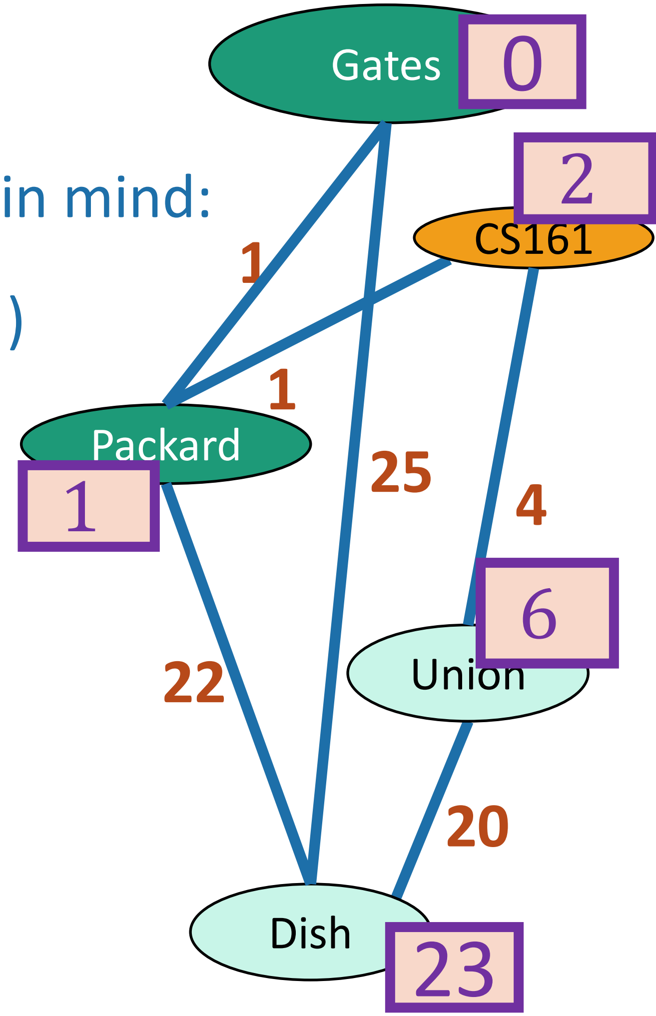
Whatever path we had in mind before

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

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

Formally:

- We should prove this by induction.



Claim 2

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

- For s (the start vertex):
  - The first vertex marked **sure** has d[s] = d(s,s) = 0.
- For all the other vertices:
  - Suppose that we are about to add u to the **sure** list.
  - That is, we picked u in the first line here:

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

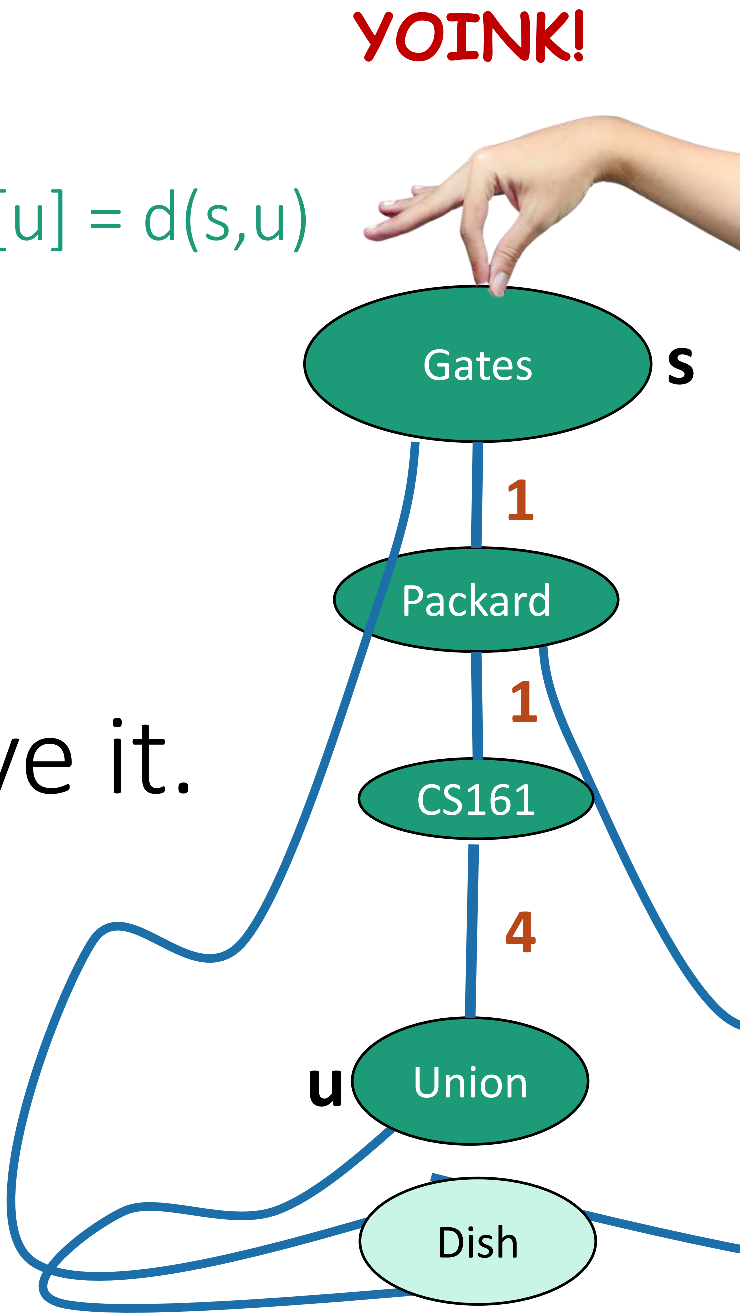
- Want to show that d[u] = d(s,u).

Intuition

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

- The first path that lifts **u** off the ground is the shortest one.

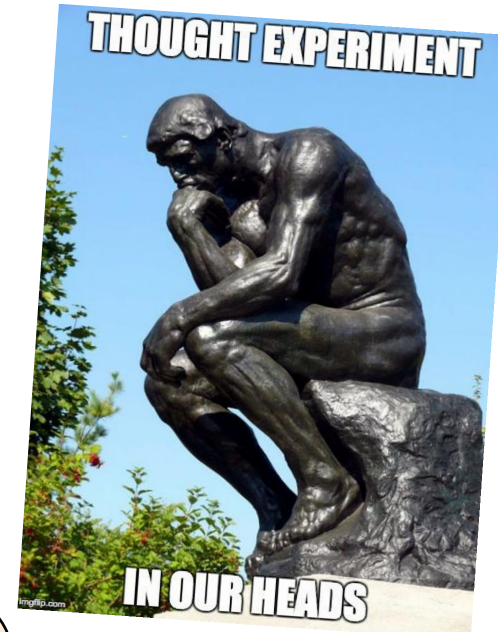
But let's actually prove it.



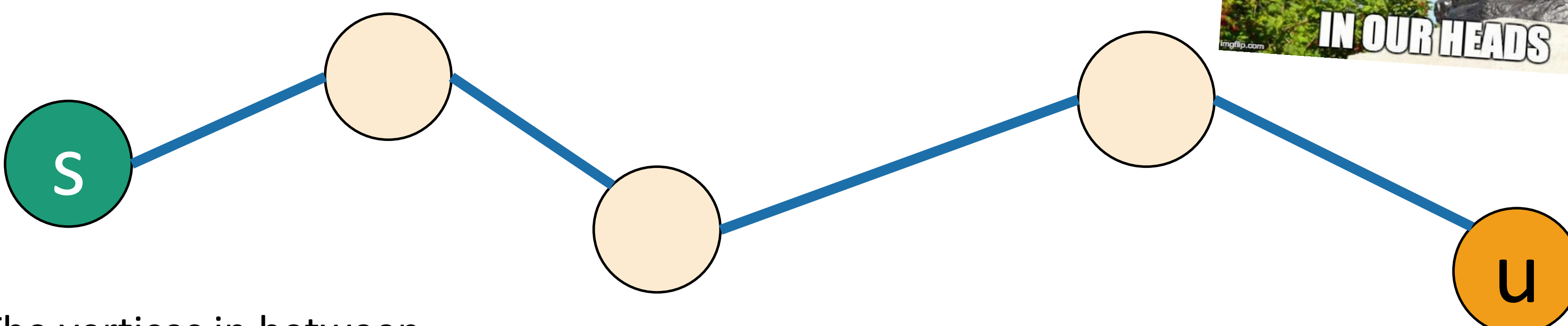
Temporary definition:
v is "good" means that d[v] = d(s,v)

Claim 2

- Want to show that u is good.



Consider a **true** shortest path from s to u:



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

True shortest path.

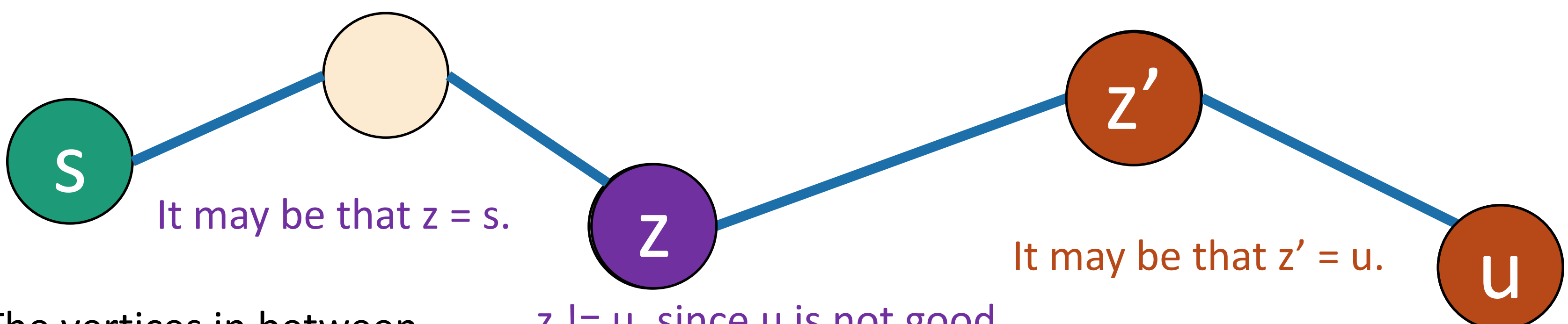
Claim 2

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

means good   means not good

"by way of contradiction"

- Want to show that u is good. BWOC, suppose u isn't good.
- Say z is the last good vertex before u.
- z' is the vertex after z.



It may be that z = s.

It may be that z' = u.

z != u, since u is not good.

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

True shortest path.

Claim 2

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

means good — means not good

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

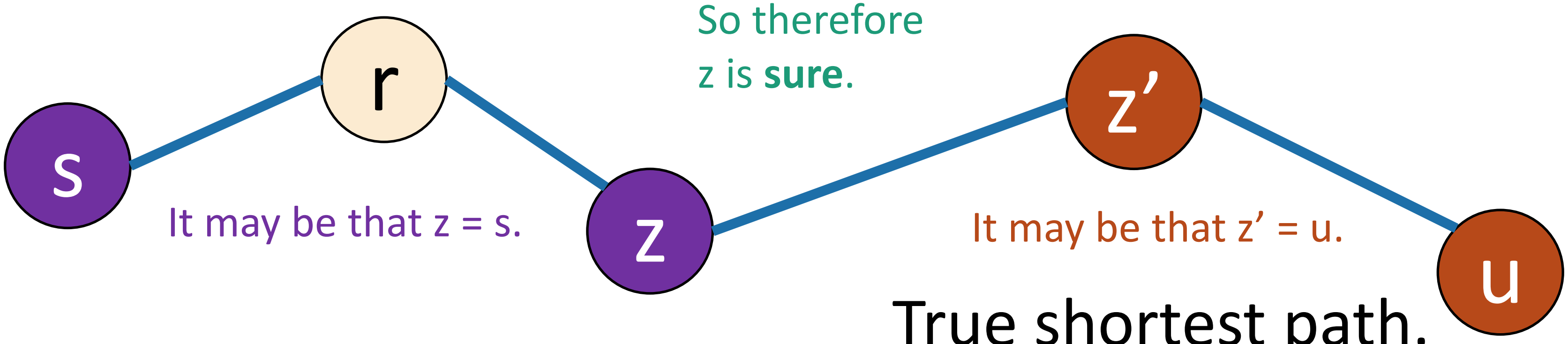
$$d[z] = d(s,z) \le d(s,u) \le d[u]$$

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

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

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

So therefore z is **sure**.



Claim 2

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

means good   means not good

- Want to show that u is good. BWOC, suppose u isn’t good.
- If z is **sure** then we’ve already updated z’:
  - $d[z'] \leftarrow \min\{\, d[z'], d[z] + w(z,z')\}$, so

$$\boldsymbol{d[z'] \leq d[z] + w(z,z') = d(s,z') \leq d[z']}$$

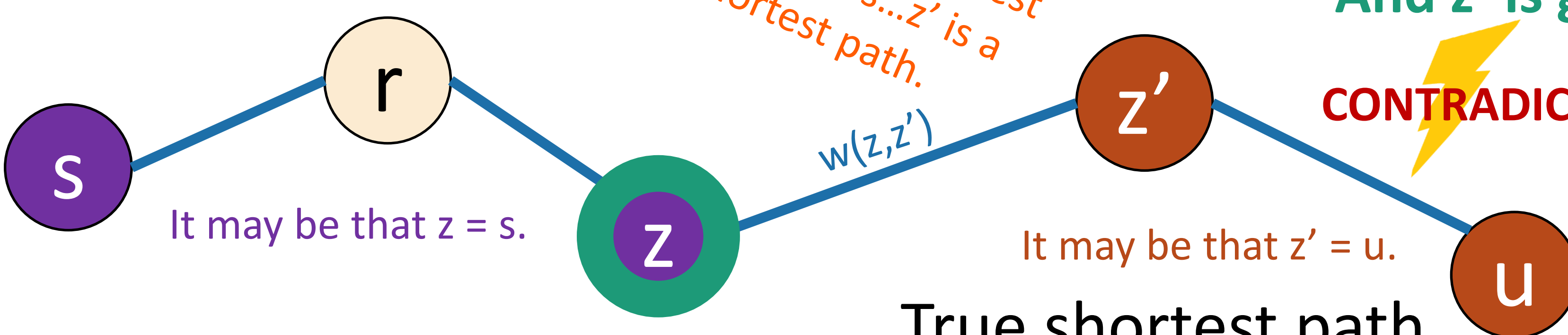
def of update

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

Claim 1

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

CONTRADICTION!!



True shortest path.

Back to this slide

Claim 2

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

means good

means not good

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

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

Def. of z

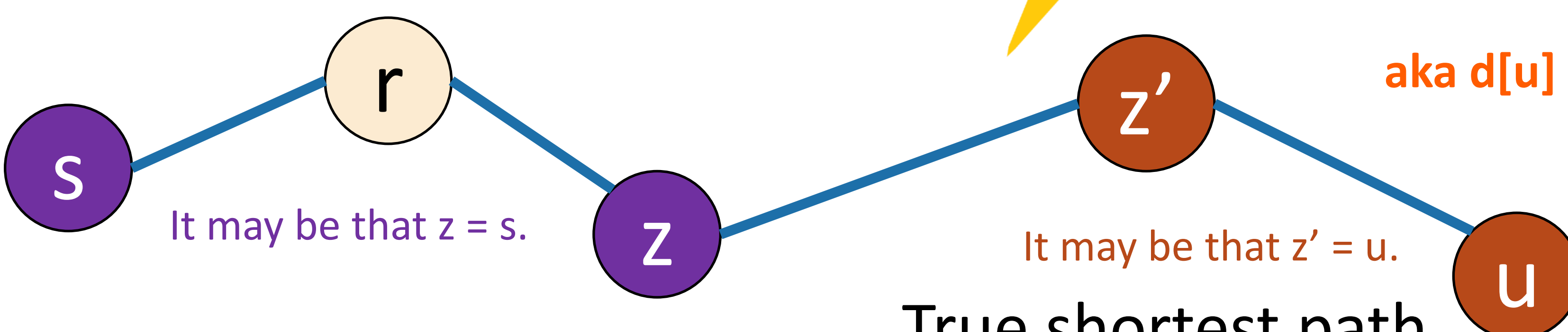
This is the shortest path from s to x

Claim 1

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

So u is good!

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



True shortest path.

Back to this slide

Claim 2

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

- For s (the starting vertex):
  - The first vertex marked **sure** has d[s] = d(s,s) = 0.
- For all other vertices:
  - Suppose that we are about to add u to the **sure** list.
  - That is, we picked u in the first line here:

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

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

Why does this work?

Now back to this slide

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

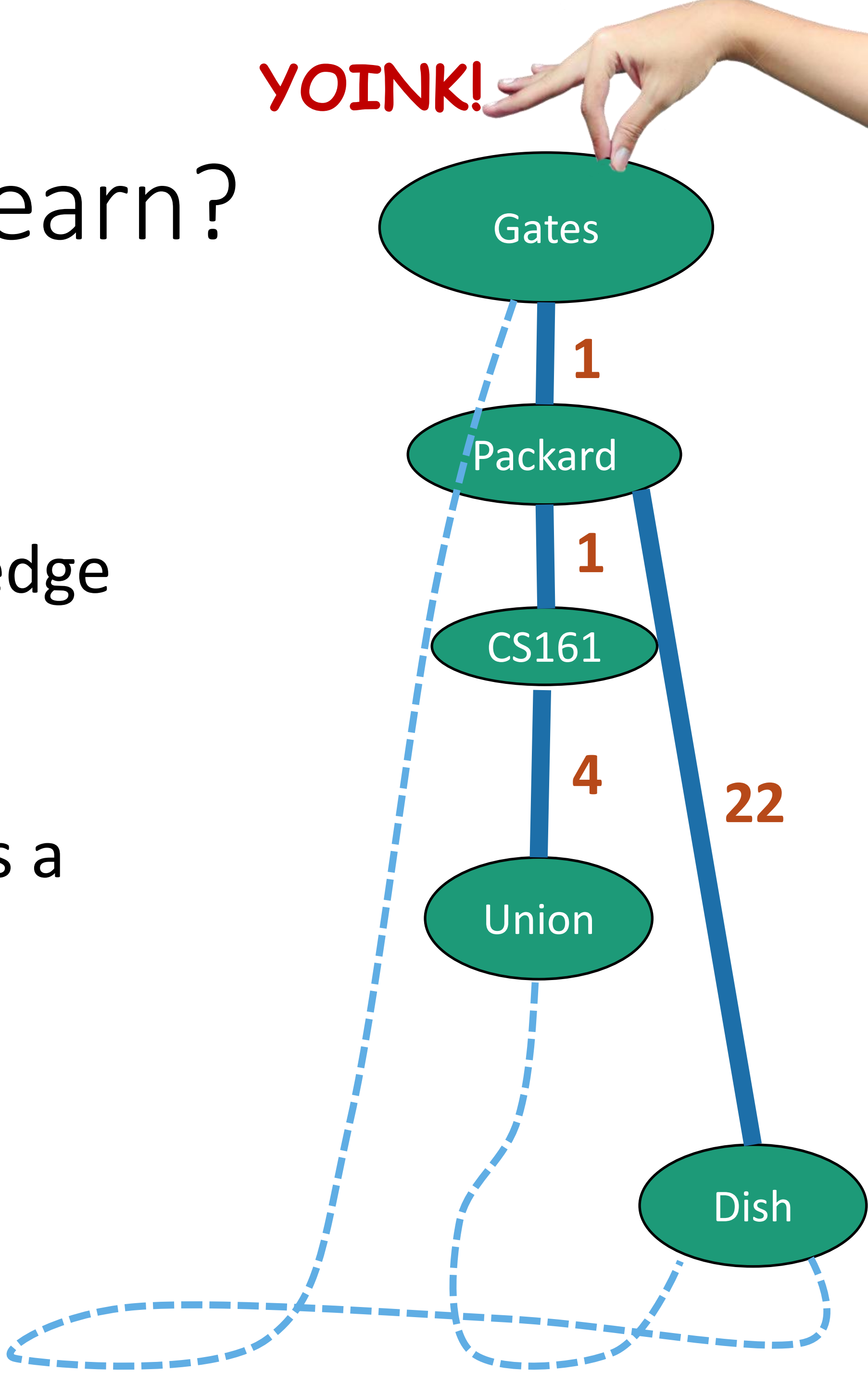
- Proof outline:
  - **Claim 1**: For all v, **d[v] $\geq$ d(s,v).**
  - **Claim 2**: When a vertex is marked **sure**, **d[v] = d(s,v).**

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

What did we just learn?

- **Dijkstra's algorithm finds shortest paths** in weighted graphs with non-negative edge weights.

- Along the way, it constructs a nice tree.



As usual

- Does it work?
  - Yes.
- Is it fast?
  - Depends on how you implement it.

Running time?

Dijkstra(G,s):

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

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

Depends on how we implement it…

We need a data structure that:

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

Just the inner loop:

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

Total running time is big-oh of:

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

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

If we use an array

- $T(\text{findMin}) = O(n)$
- $T(\text{removeMin}) = O(n)$
- $T(\text{updateKey}) = O(1)$

- Running time of Dijkstra

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

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

  $= O(n^2)$

If we use a red-black tree

- $T(\text{findMin}) = O(\log(n))$
- $T(\text{removeMin}) = O(\log(n))$
- $T(\text{updateKey}) = O(\log(n))$

- Running time of Dijkstra

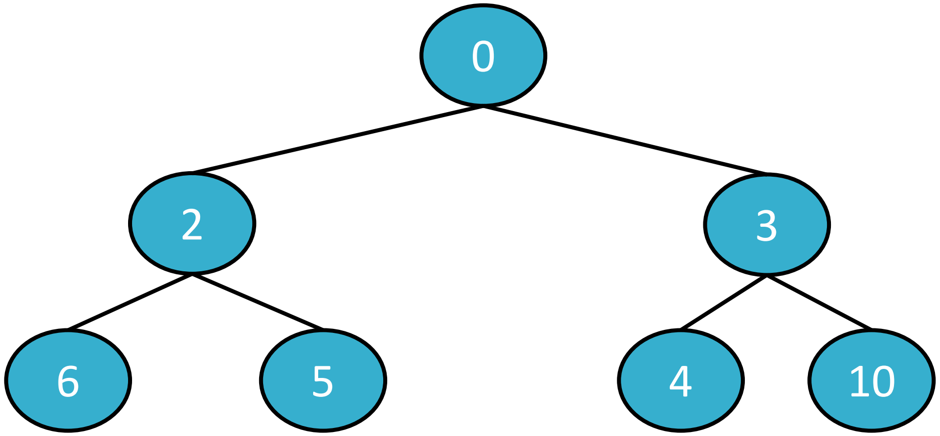
$$= O(n(\,T(\texttt{findMin}) + T(\texttt{removeMin})\,) + m\,T(\texttt{updateKey}))$$
$$= O(n\log(n)) + O(m\log(n))$$
$$= O((n + m)\log(n))$$

Better than an array if the graph is sparse!

aka if m is much smaller than n^2

Heaps support these operations

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



- A **heap** is a tree-based data structure that has the property that every node has a smaller key than its children.
- Not covered in this class – see AoA1!!! (Or CLRS).
- But! We will use them.

Many heap implementations

Nice chart on Wikipedia:

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

Say we use a **Fibonacci Heap**

- T(findMin) = O(1) (amortized time*)
- T(removeMin) = O(log(n)) (amortized time*)
- T(updateKey) = O(1) (amortized time*)
- See CS166 for more! (or CLRS)

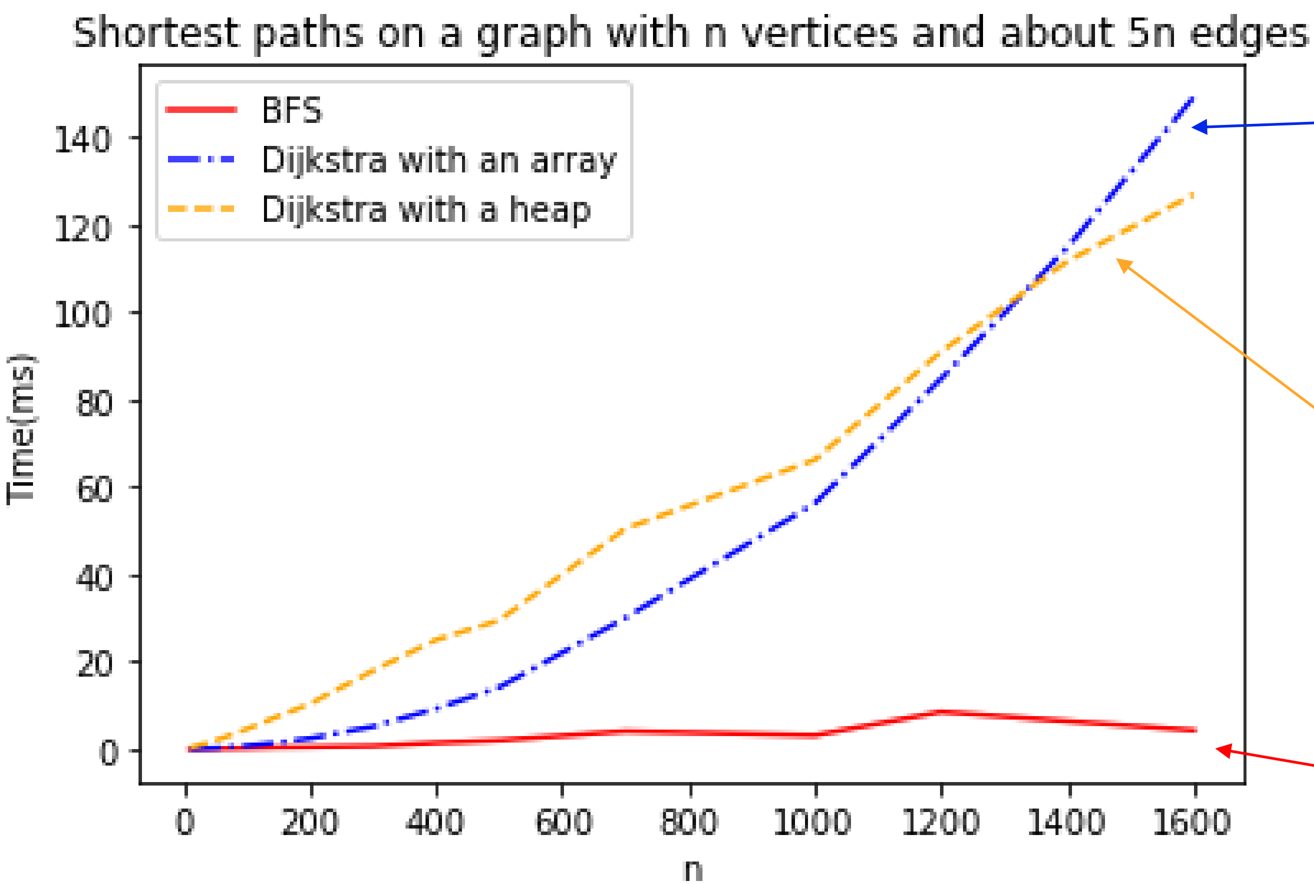
- Running time of Dijkstra

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

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

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

In practice



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

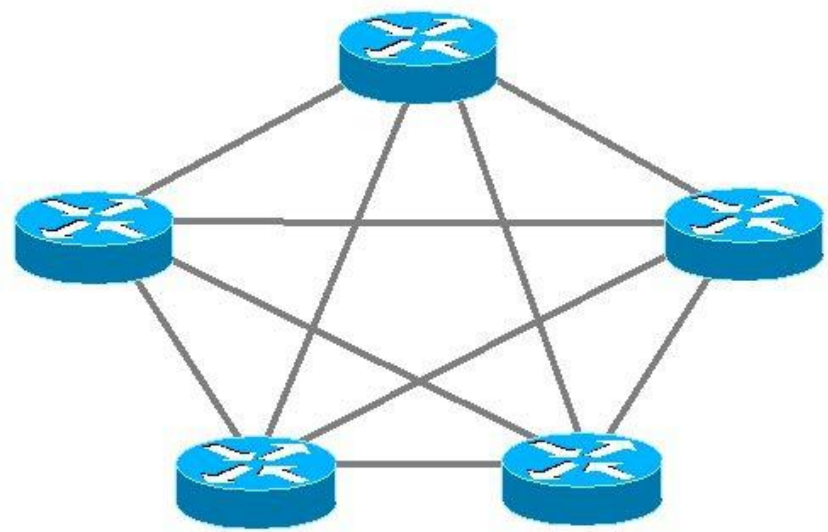
Dijkstra using a heap looks a bit more linear (actually nlog(n))

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.

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).

NEXT LECTURE

- The Minimum Spanning Tree
- Prim's Algorithm
- Kruskal's Algorithm