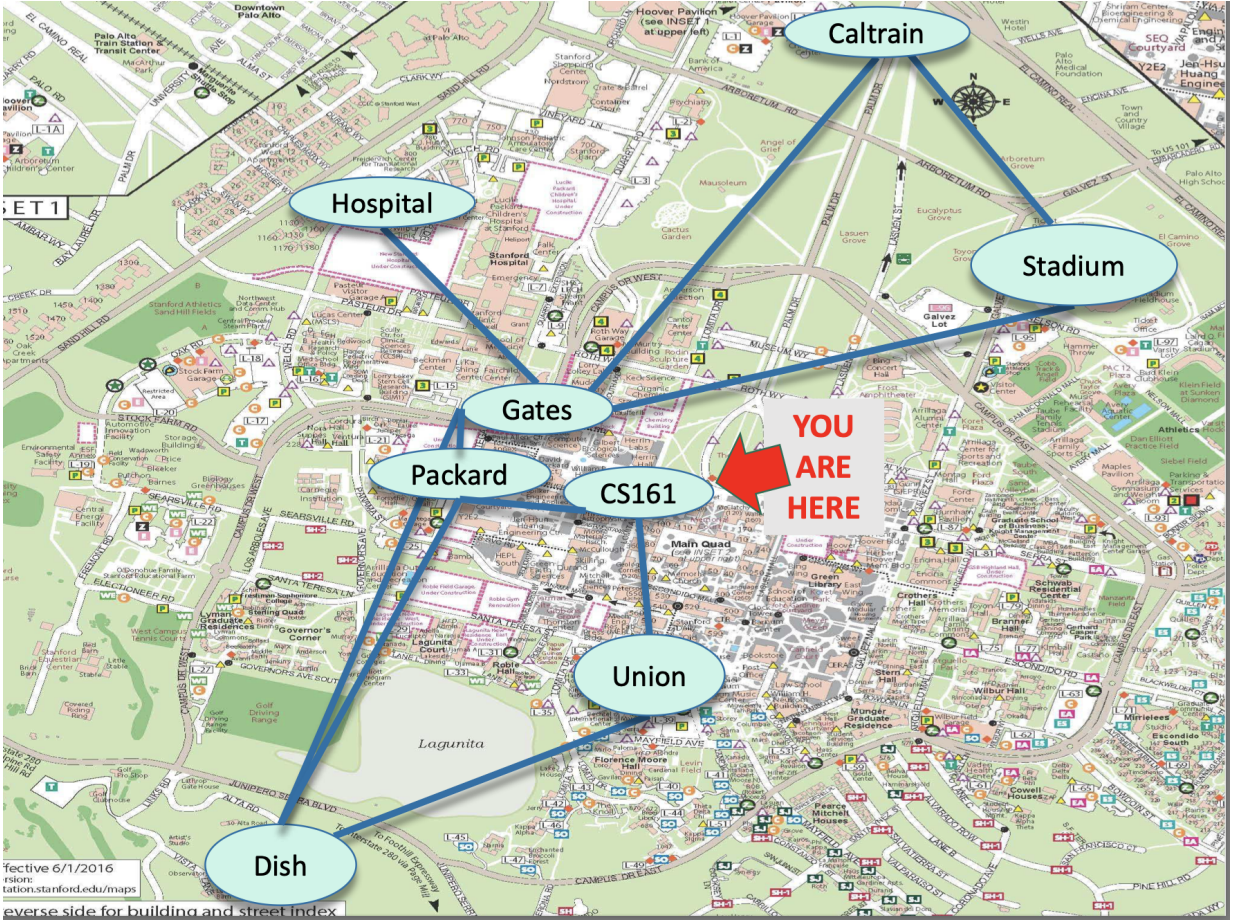
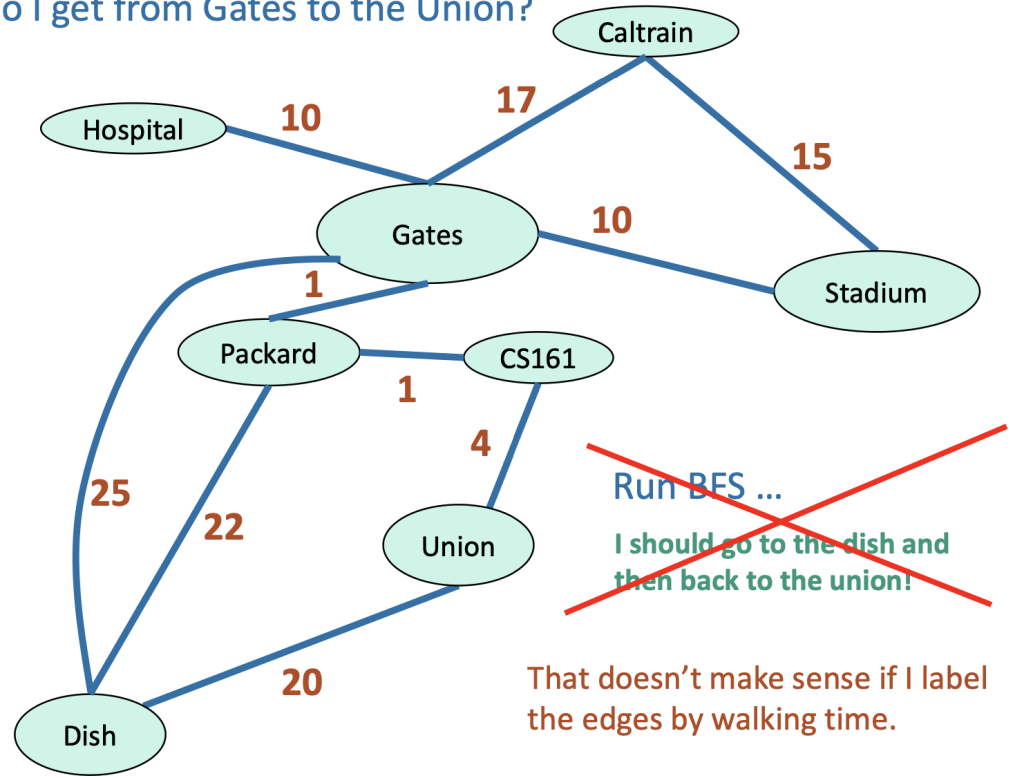
Dijkstra's Shortest Paths Algorithm


ANA
Inspiration of JAPAN
ANA


Caltrain
Hospital
Stadium
Gates
YOU ARE HERE
Packard
CS161
Union
Dish

Just the graph

How do I get from Gates to the Union?



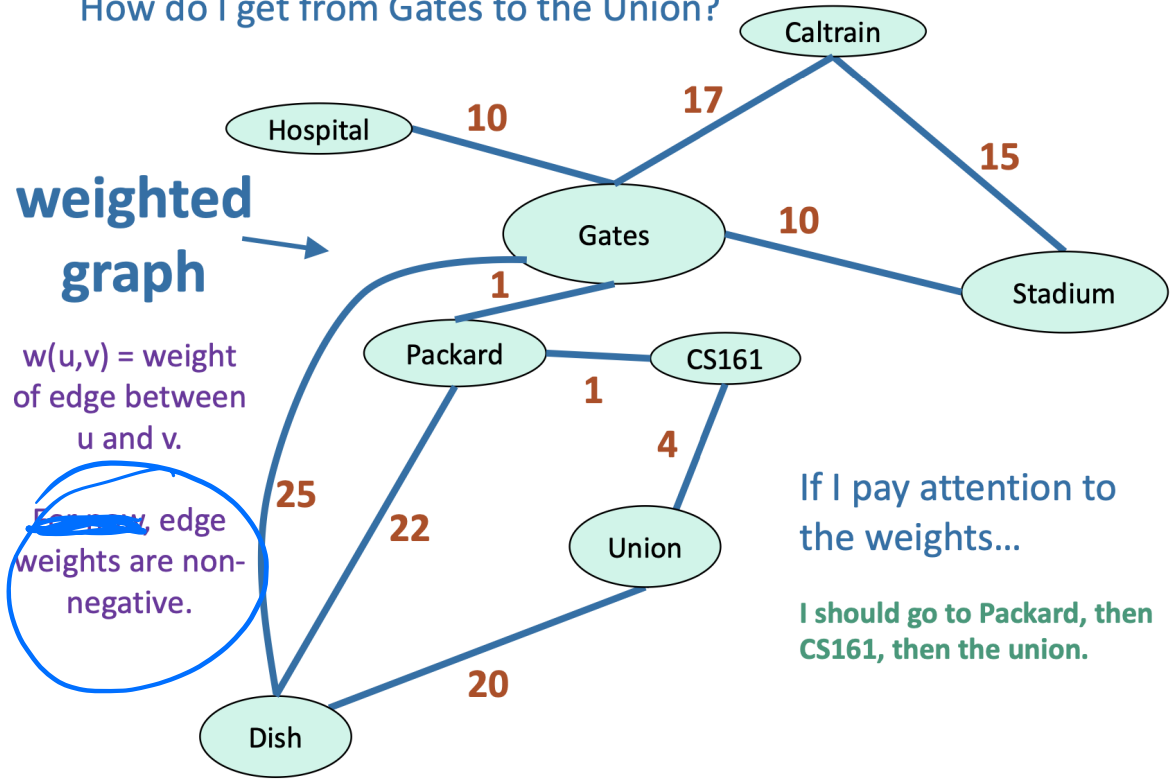
~~Run BFS …~~

~~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.

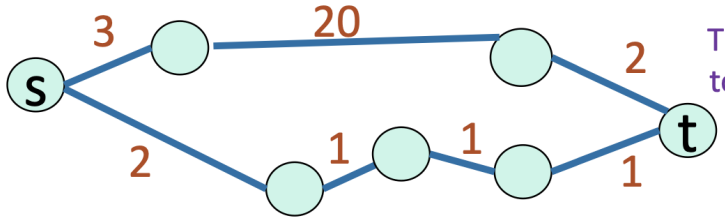
Just the graph

How do I get from Gates to the Union?



Shortest path problem

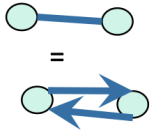
- What is the shortest path between u and v in a weighted graph?
  - the **cost** of a path is the sum of the weights along that path
  - The **shortest path** is the one with the minimum cost.



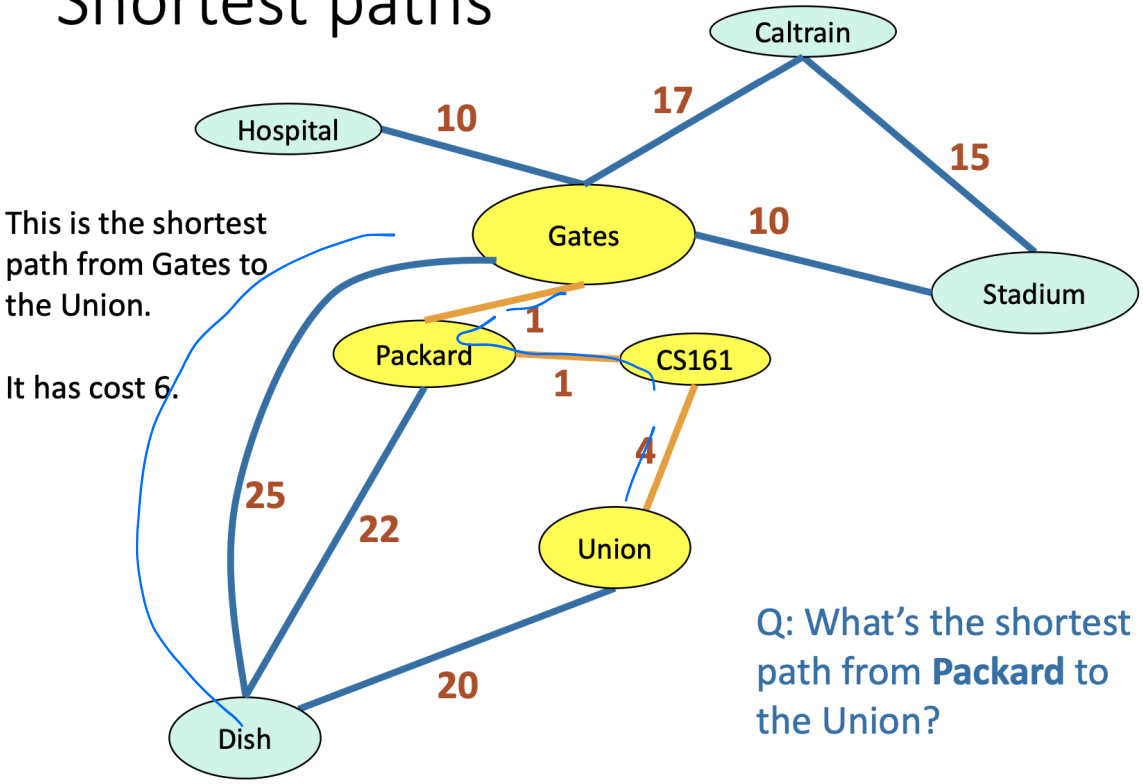
This path from s to t has cost 25.

This path is shorter, it has cost 5.

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



Shortest paths

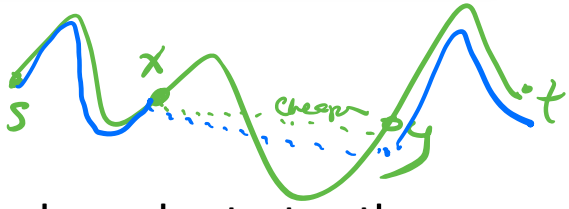


This is the shortest path from Gates to the Union.

It has cost 6.

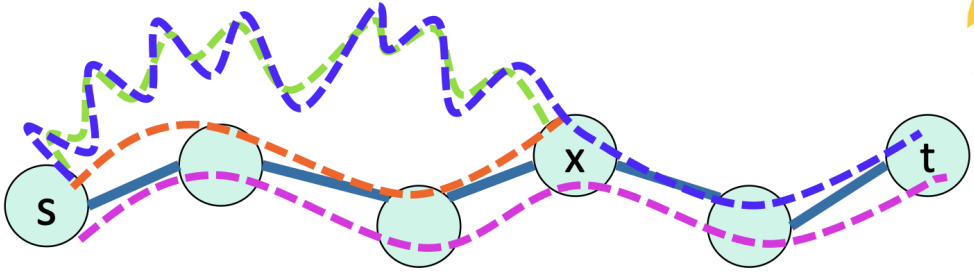
Q: What's the shortest path from **Packard** to the Union?

Warm-up



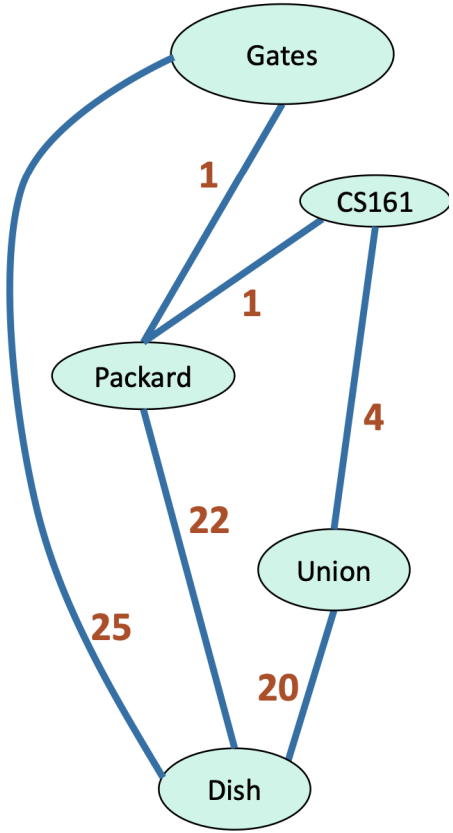
- A sub-path of a shortest path is also a shortest path.

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

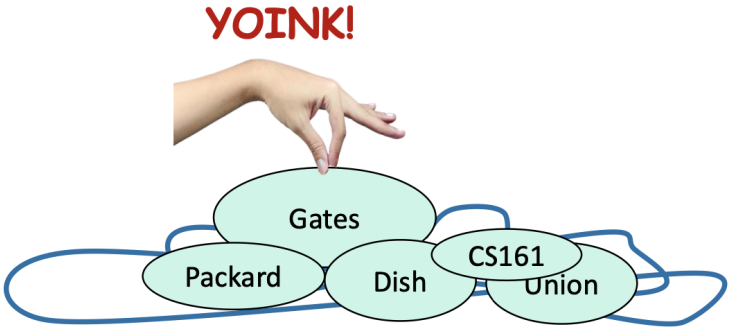


Dijkstra's algorithm

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

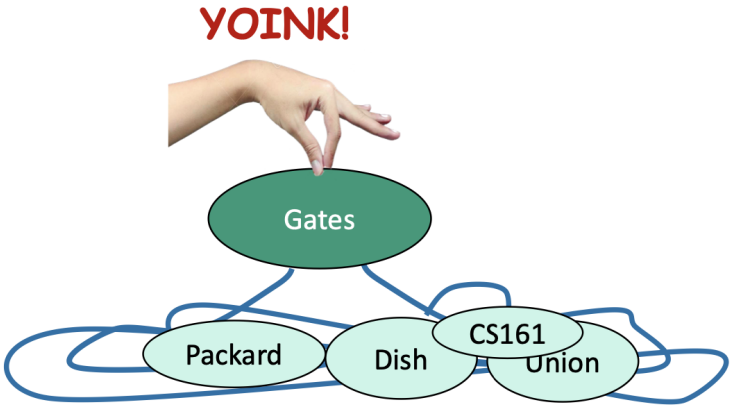


Dijkstra
intuition

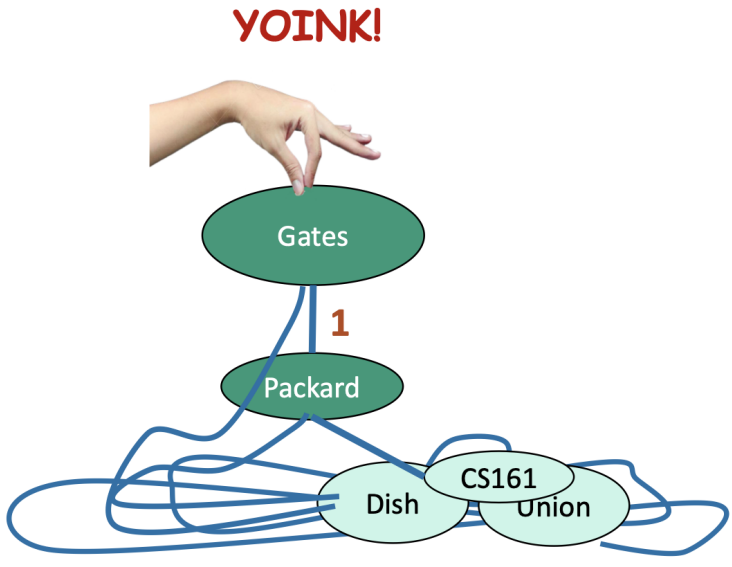


Dijkstra
intuition

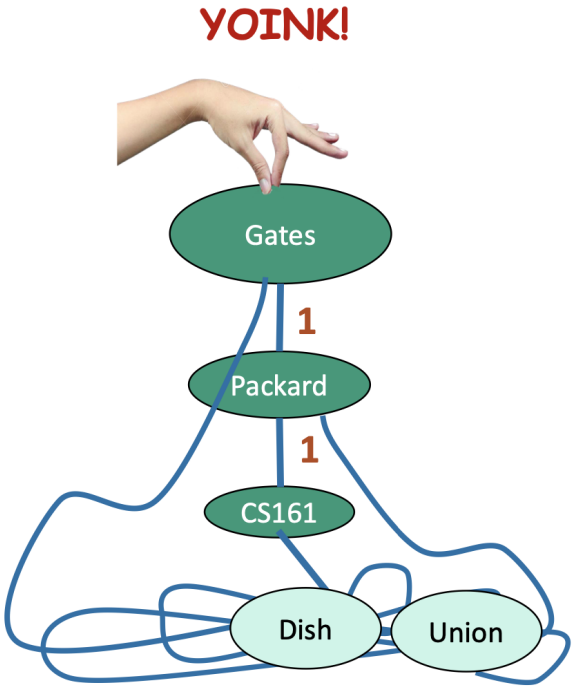
A vertex is done when it's not on the ground anymore.



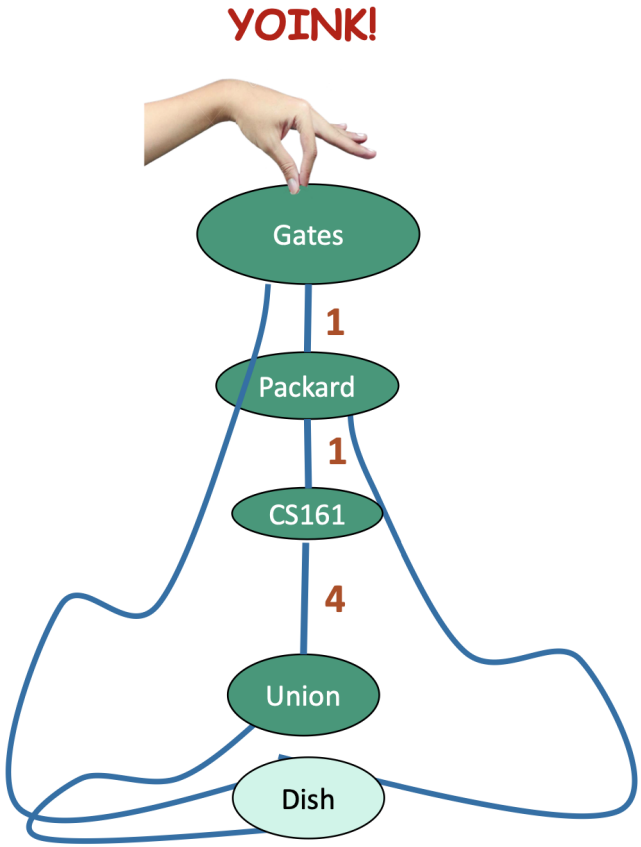
Dijkstra
intuition



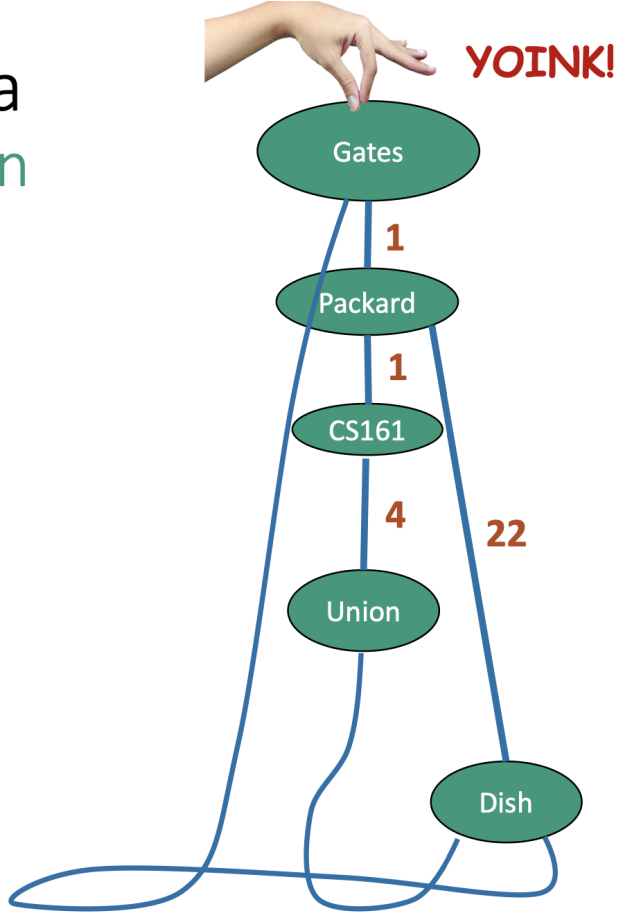
Dijkstra
intuition



Dijkstra
intuition



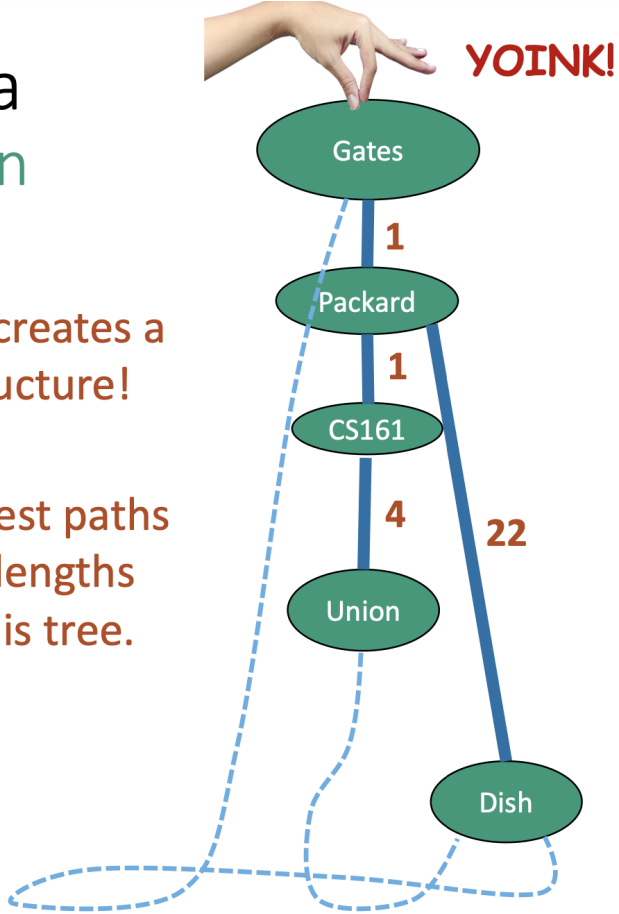
Dijkstra
intuition



Dijkstra
intuition

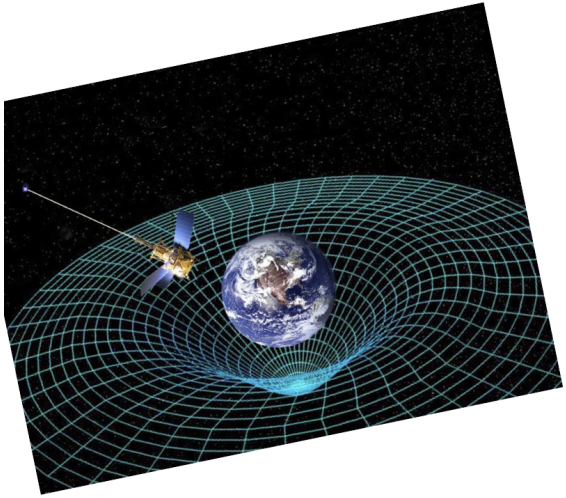
This also creates a tree structure!

The shortest paths are the lengths along this tree.



How do we actually implement this?

- **Without** string and gravity?



Dijkstra by example

Initialize:
d[Gates] = 0
dist[Allelse] = ∞
all light green

How far is a node from Gates?

I'm not sure yet

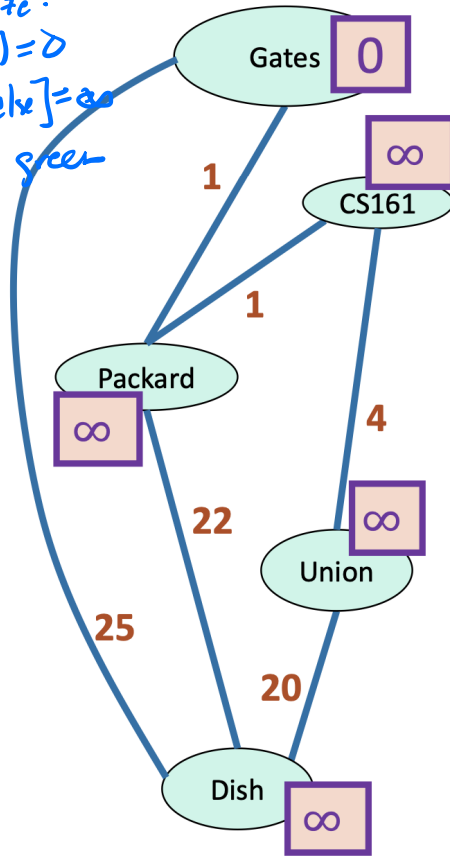
I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x

That is, an estimate of d(v,Gates).

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

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



Dijkstra by example

How far is a node from Gates?

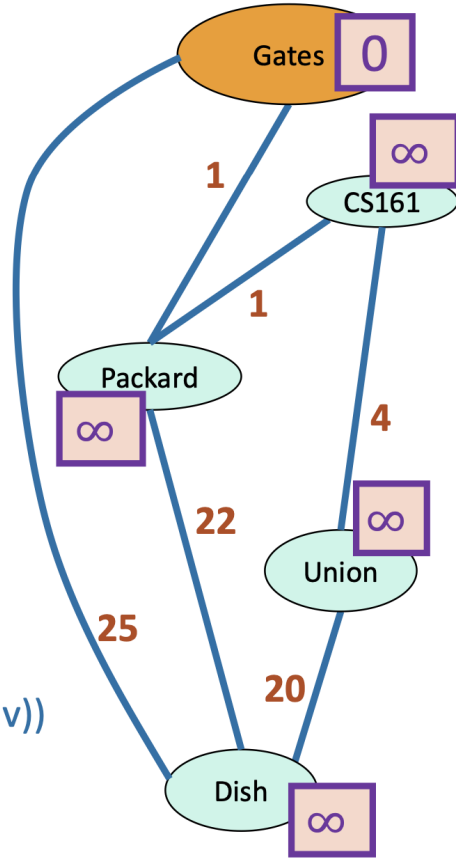

I'm not sure yet


I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x


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?

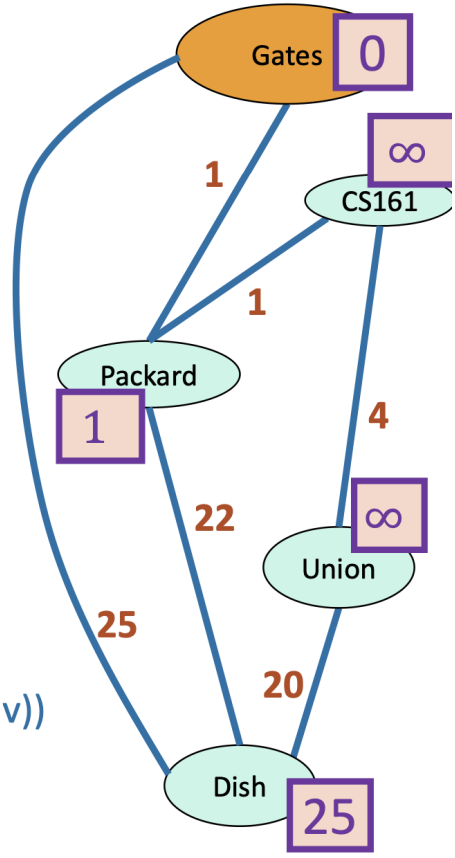

I'm not sure yet


I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x


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?

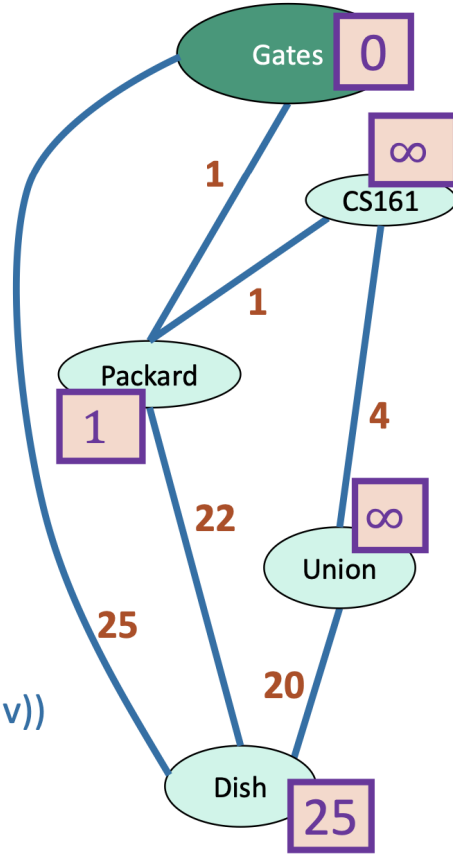
I'm not sure yet

I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x

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?

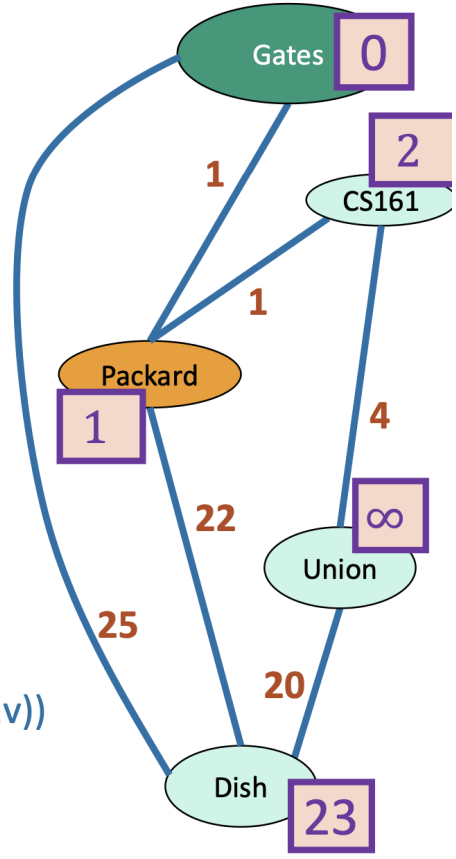
I'm not sure yet

I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x

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?

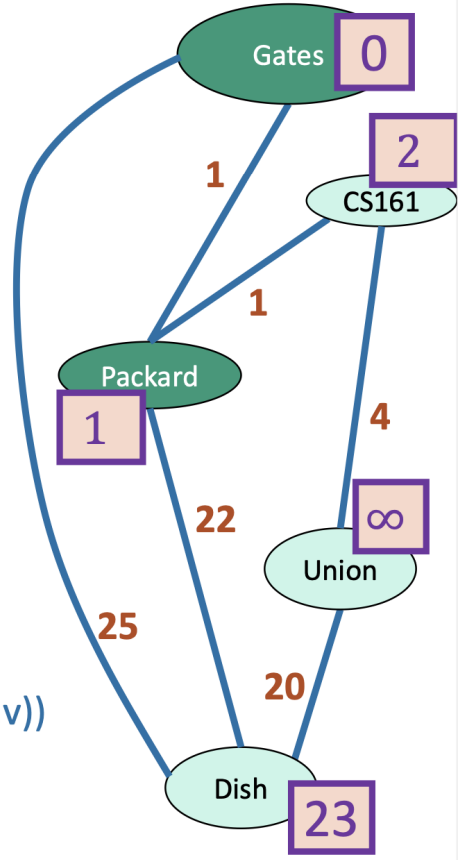

I'm not sure yet


I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x


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?

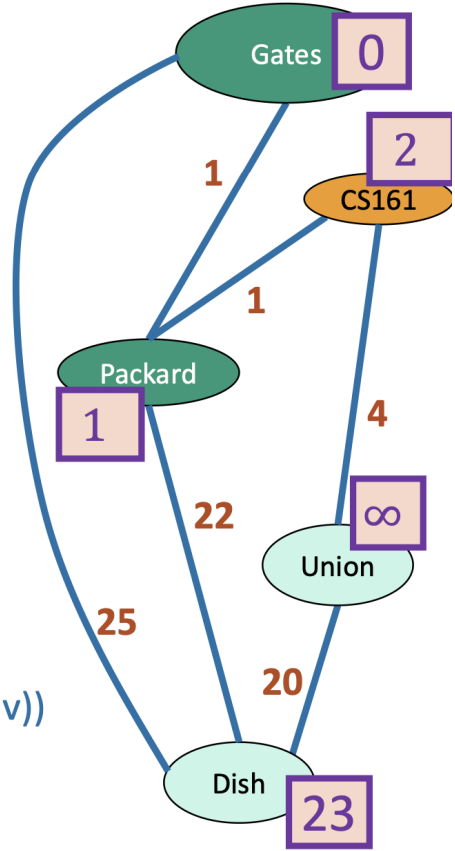

I'm not sure yet


I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x


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?

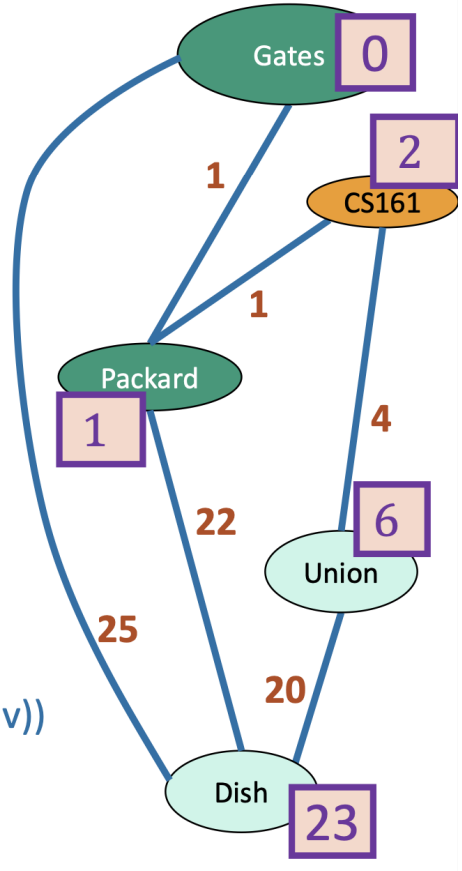

I'm not sure yet


I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x


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?

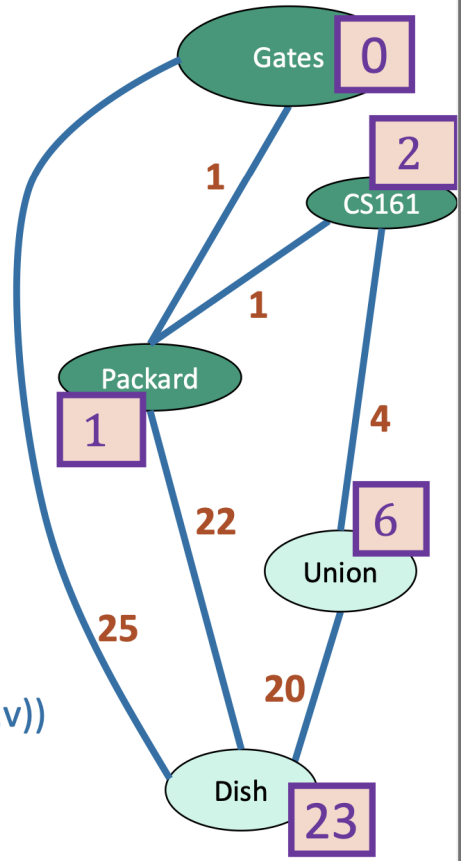
I'm not sure yet

I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x

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?

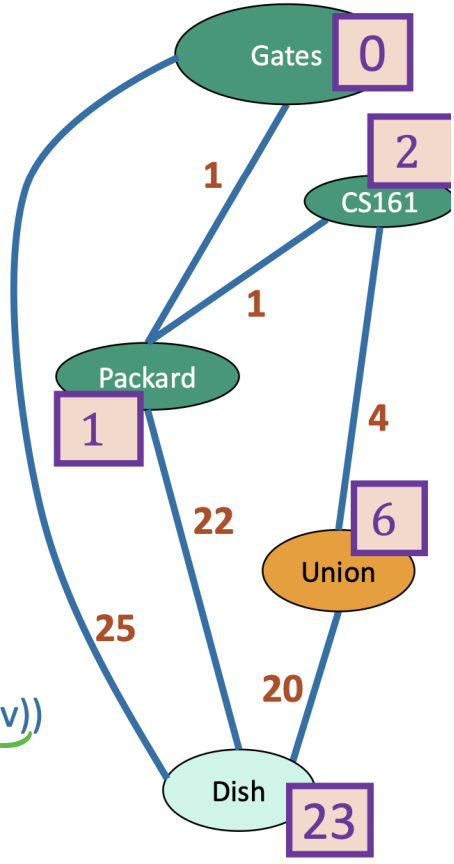
I'm not sure yet

I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x

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?

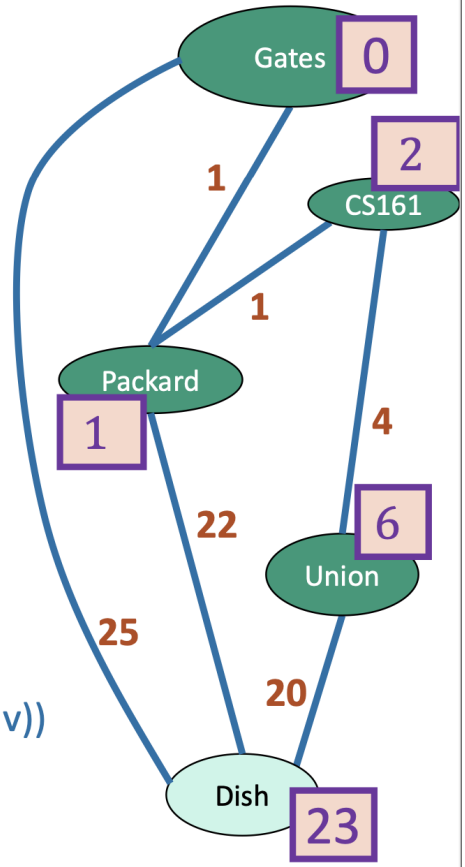

I'm not sure yet


I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x


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?

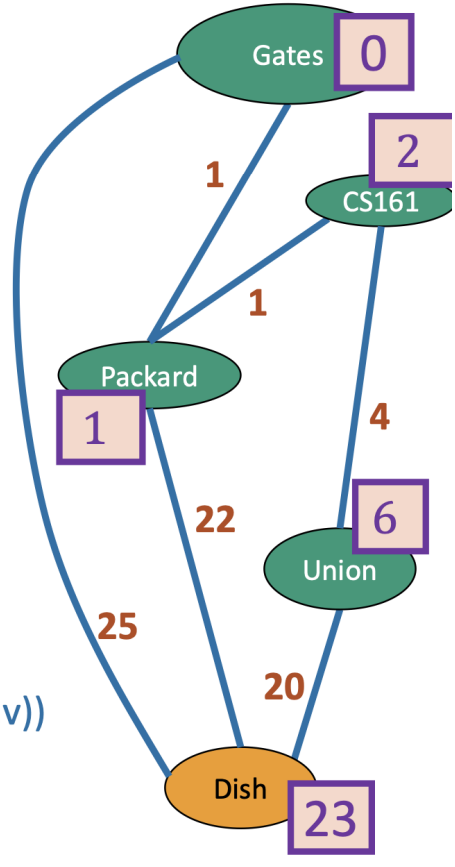
I'm not sure yet

I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x

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

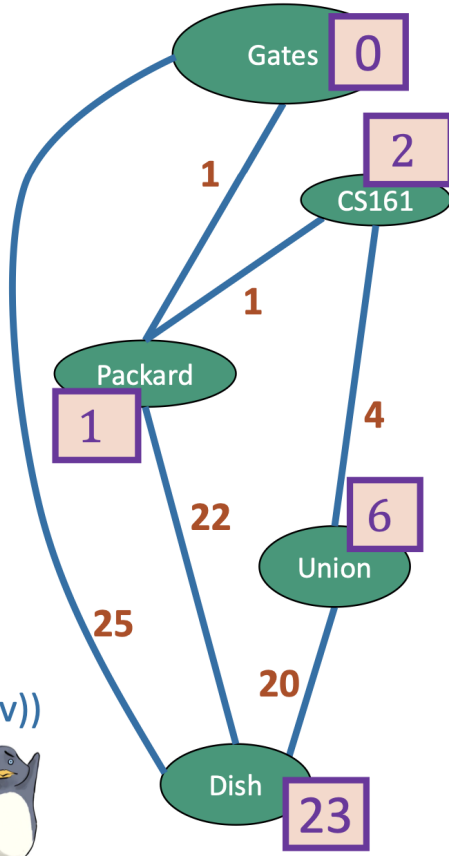
I'm sure

x is my best over-estimate for a vertex v.
We'll say d[v] = x

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

More formal pseudocode on board (or see CLRS)!



Why does this work?

Why does this work?

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

Next let's prove these!

- **Claims 1 and 2** imply the **theorem.**
  - d[v] never increases, so Claims 1 and 2 imply that **d[v] weakly decreases until d[v] = d(s,v), then never changes again.**
  - By the time we are **sure** about v, d[v] = d(s,v). (Claim 2 again)
  - All vertices are eventually **sure**. (Stopping condition in algorithm)
  - So all vertices end up with d[v] = d(s,v).

Claim 1

d[v] ≥ d(s,v) for all v.

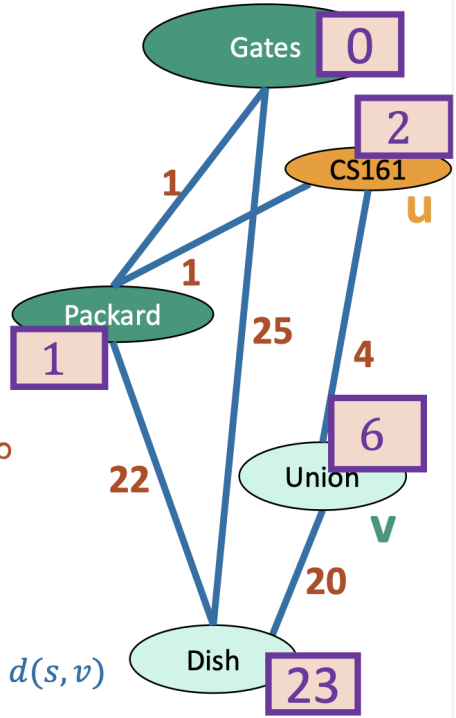
- Inductive hypothesis.
  - After t iterations of Dijkstra, d[v] ≥ d(s,v) for all v.
- Base case:
  - At step 0, $d(s,s) = 0$, and $d(s,v) \leq \infty$
- Inductive step: say hypothesis holds for t.
  - Then at step t+1:
    - We pick **u**; for each neighbor **v**:
    - d[v] ← min(d[v] , d[u] + w(u,v)) ≥ $d(s,v)$

(Details on board)

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

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

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



Claim 2

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

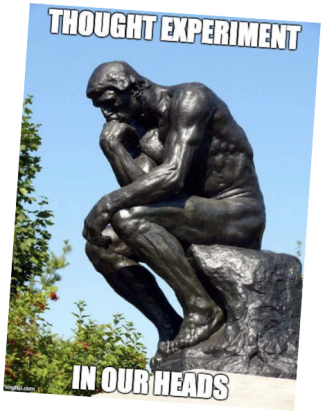
- To begin with:
  - The first vertex marked **sure** has d[s] = d(s,s) = 0.
- For t > 0:
  - Suppose that we are about to add u to the **sure** list.
  - That is, we picked u in the first line here:

- 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

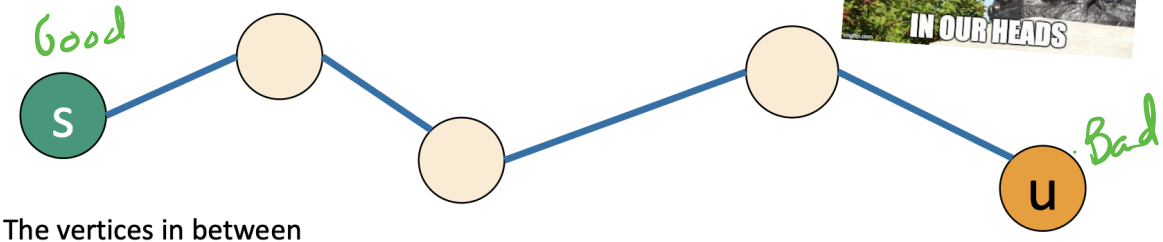
Claim 2

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

- Want to show that u is good.



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



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

True shortest path.

Claim 2

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

means good means not good

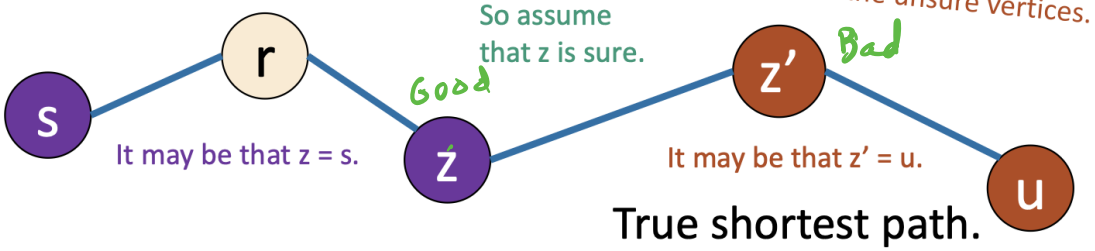
- Want to ~~show that u~~ is good. BWOC, suppose it's not.

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

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.



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 it's not.
- If z is **sure** then we've already updated z':
  - $d[z'] \leftarrow \min\{ d[z'], d[z] + w(z,z')\}$, so

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

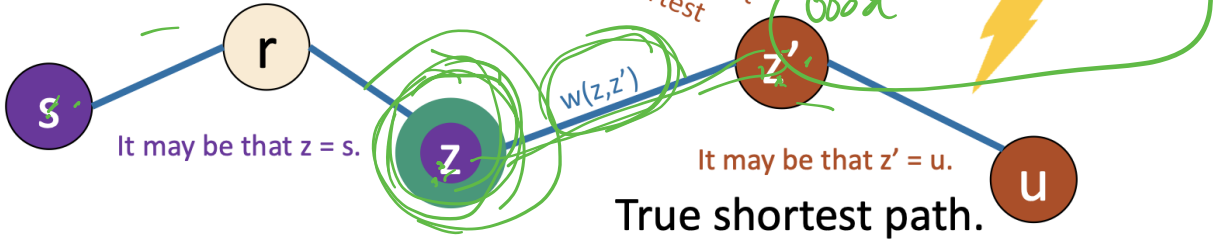
def of update

sub-paths of shortest paths are shortest paths

Claim 1

So everything is equal.
And z' is good.

CONTRADICTION!!



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 it's not.

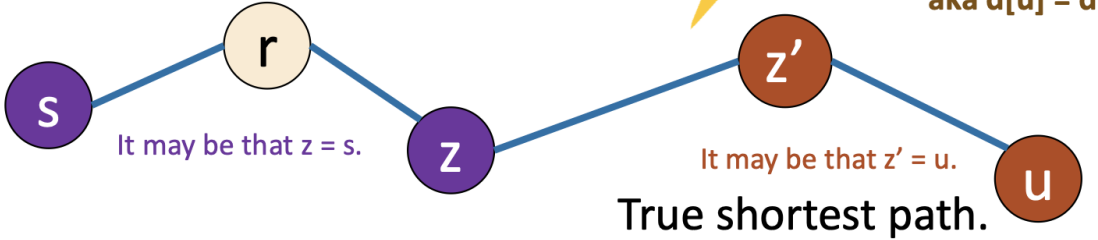
$$d[z] = d(s,z) \leq d(s,u) \leq d[u]$$

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)



Back to this slide

Claim 2

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

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

Then u is good!

aka d[u] = d(s,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

Why does this work?

Now back to this slide

- **Theorem**: 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.**
  - We will never mess up d[v] after v is marked **sure**, because d[v] is a decreasing over-estimate.

Why does this work?

Now back to this slide

- **Theorem**:
  - Run Dijkstra on G =(V,E).
  - 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] ≥ d(s,v).
  - **Claim 2**: When a vertex v is marked **sure**, d[v] = d(s,v).

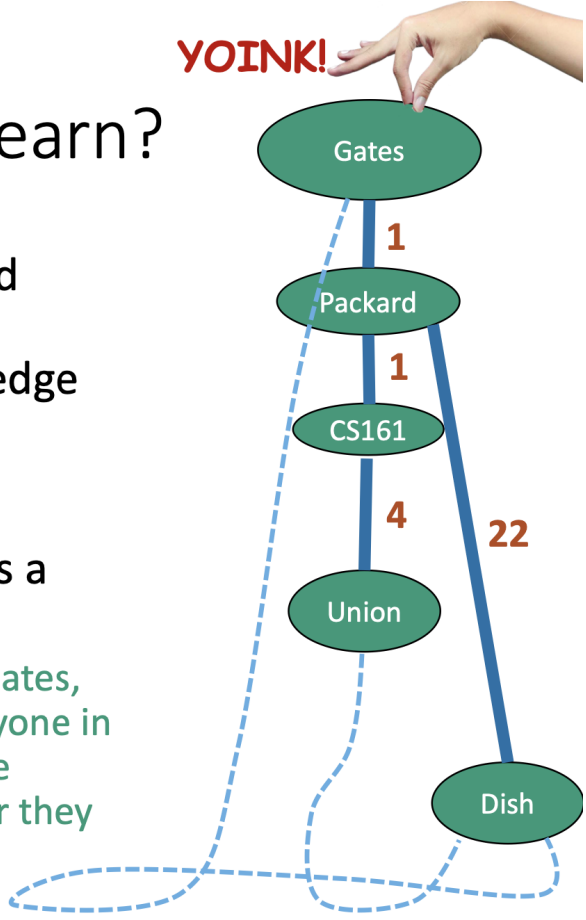
- **Claims 1 and 2** imply the **theorem.**

  - d[v] never increases, so Claims 1 and 2 imply that **d[v] weakly decreases until d[v] = d(s,v), then never changes again.**
  - By the time we are **sure** about v, d[v] = d(s,v). (Claim 1 again)
  - All vertices are eventually **sure.** (Stopping condition in algorithm)
  - So all vertices end up with d[v] = d(s,v).

What did we just learn?

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

- Along the way, it constructs a nice tree.
  - We could post this tree in Gates, and it would be easy for anyone in Gates to figure out what the shortest path is to wherever they want to go.



How long does it take?

Running time?

This is not very precise pseudocode (eg, initialization step is missing)…but it's good enough for this reasoning.

- 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

This will run for n iterations, since there's one iteration per vertex.

How long does an 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 v with minimum d[v]
  - `findMin()`
- Can remove that v
  - `removeMin(v)`
- Can update the d[v]
  - `updateKey(v,d)`

- 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

Total running time is big-oh of:

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

$$n(T(findMin) + T(removeMin)) + m\, T(updateKey)$$

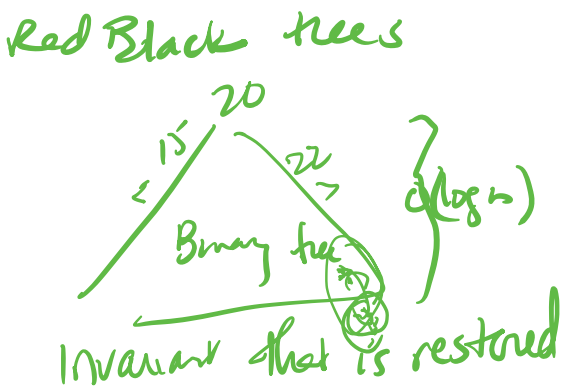
If we use an array

- 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 Priority Heap (or queue)

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



- Running time of Dijkstra

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

= O(nlog(n)) + O(mlog(n))

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

worse on dense graphs

$m = O(n^2)$

$O(n^2 \log n)$

Better than an array if the graph is sparse!

aka m is much smaller than n^2

Can also use a Fibonacci Heap

- This can do all operations in amortized time* O(1).
- Except `deleteMin` which takes amortized time* O(log(n)).
- See CS166 for more! (or CLRS)

- This gives (amortized) runtime O(m + nlog(n)) for Dijkstra's algorithm.

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