Indian Institute of Information Technology Allahabad

Data Structures and Algorithms

Single Source Shortest Paths (SSSP): Dijkstra Algo

Dr. Shiv Ram Dubey
Assistant Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

Email: srdubey@iiita.ac.in Web: https://profile.iiita.ac.in/srdubey/

DISCLAIMER

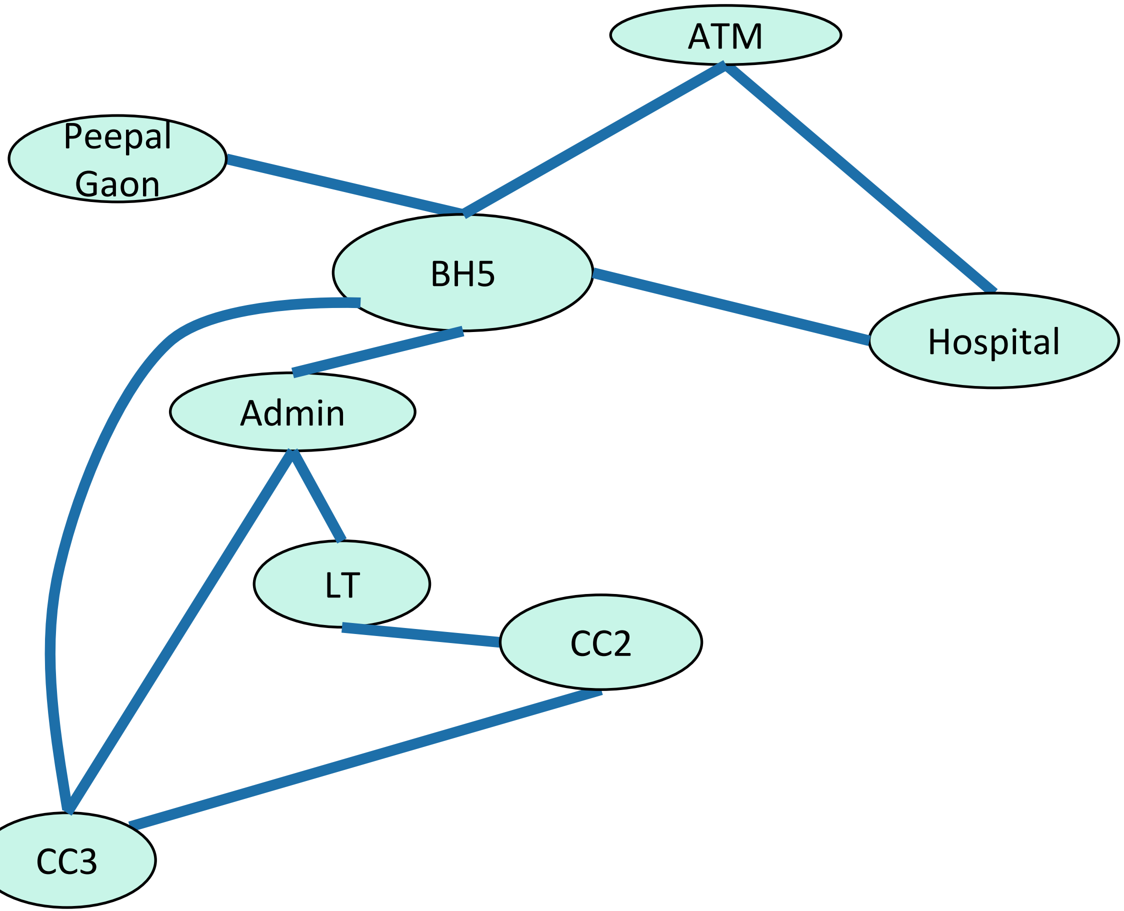
The content (text, image, and graphics) used in this slide are adopted from many sources for academic purposes. Broadly, the sources have been given due credit appropriately. However, there is a chance of missing out some original primary sources. The authors of this material do not claim any copyright of such material.

This Class

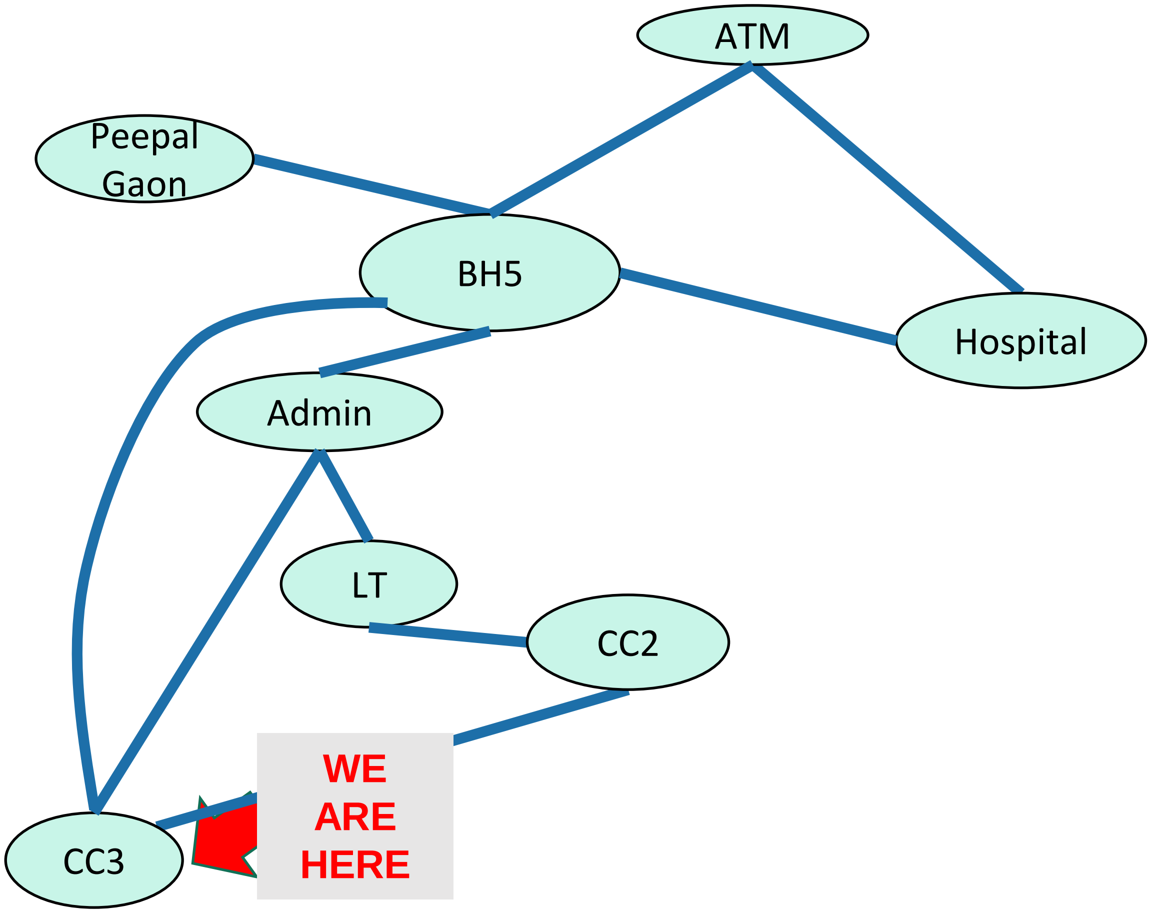
- Shortest Paths
  - BFS
  - What if the graphs are weighted?

- Single Source
  - Dijkstra!
  - Bellman-Ford!
- All Source
  - Floyd-Warshall

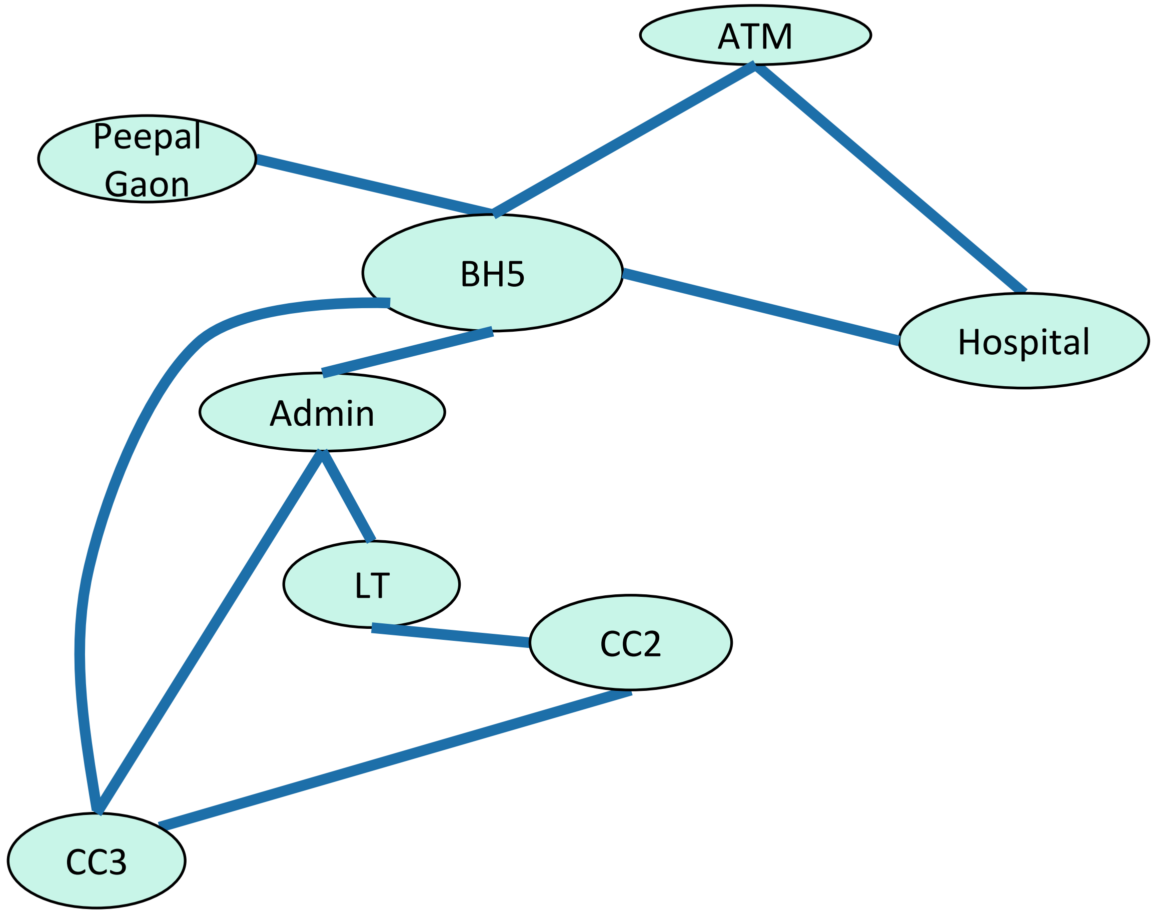
IIITA Graph



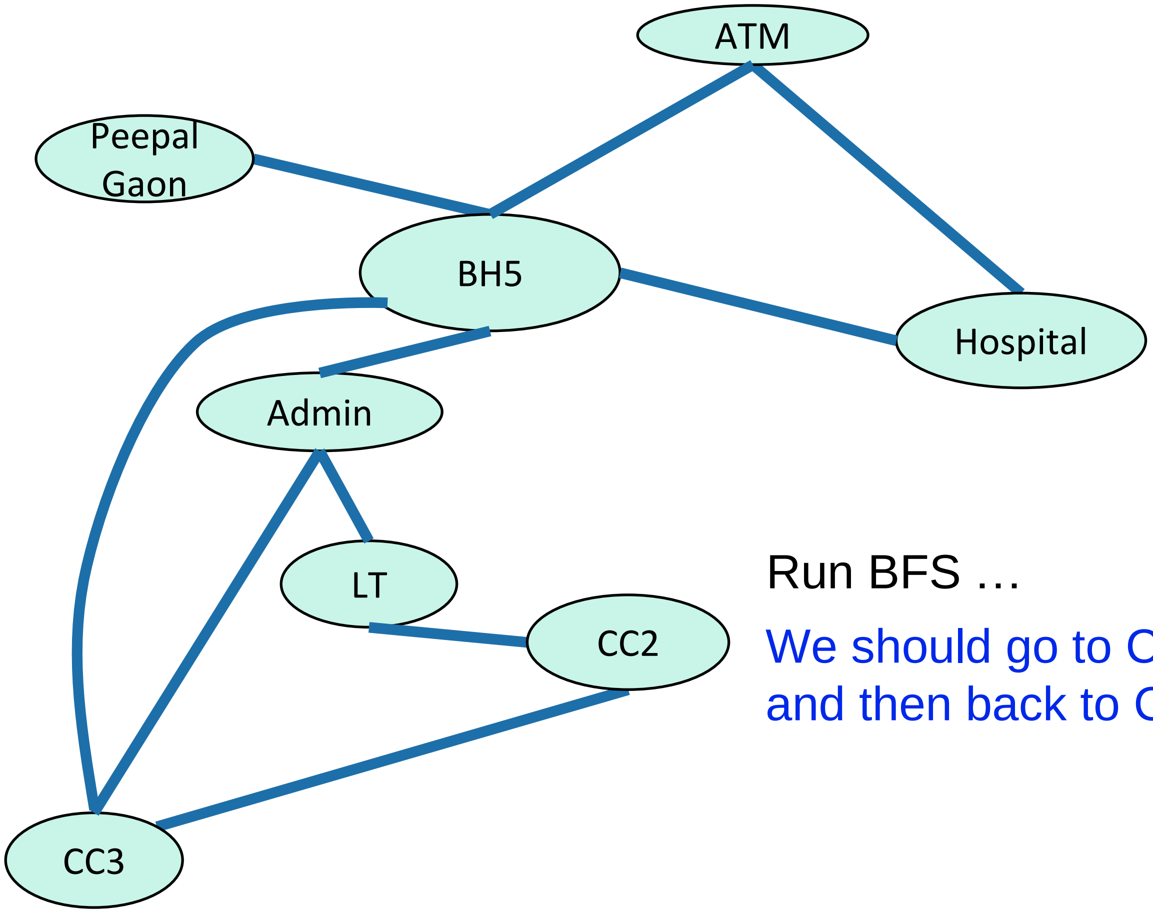
IIITA Graph



Shortest path from BH5 to CC2?



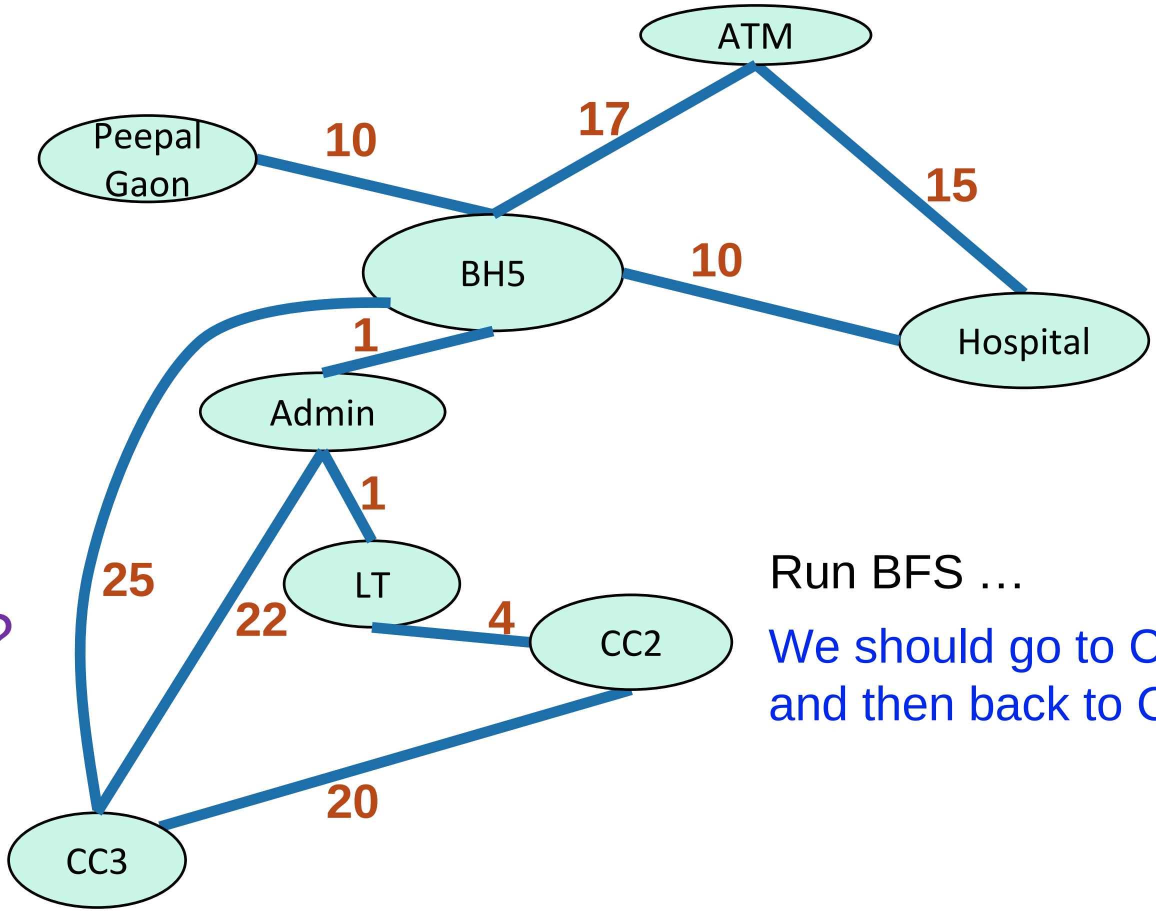
Shortest path from BH5 to CC2?



Run BFS …

We should go to CC3 and then back to CC2 !!!

Shortest path from BH5 to CC2?

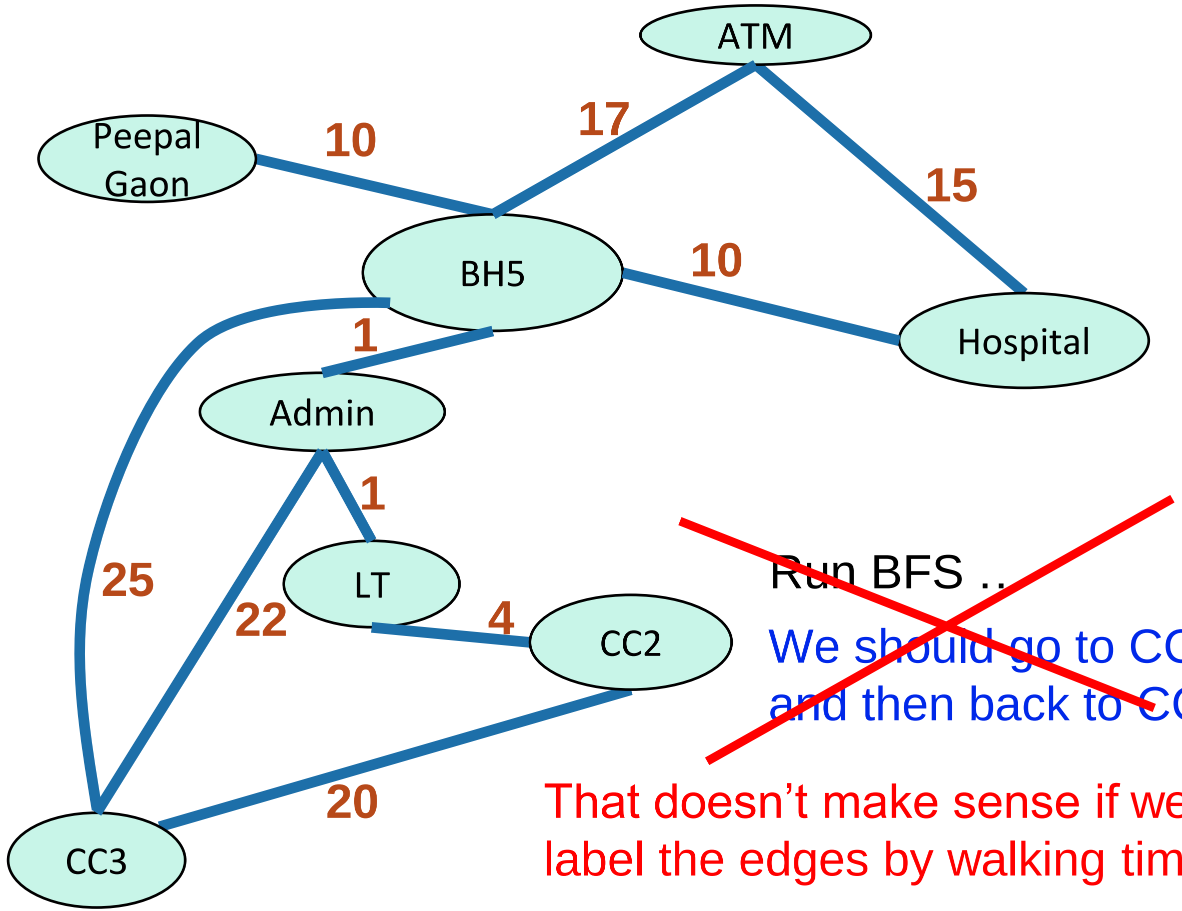


What if we label the edges by walking time ?

Run BFS …

We should go to CC3 and then back to CC2 !!!

Shortest path from BH5 to CC2?



~~Run BFS ..~~

~~We should go to CC3 and then back to CC2 !!!~~

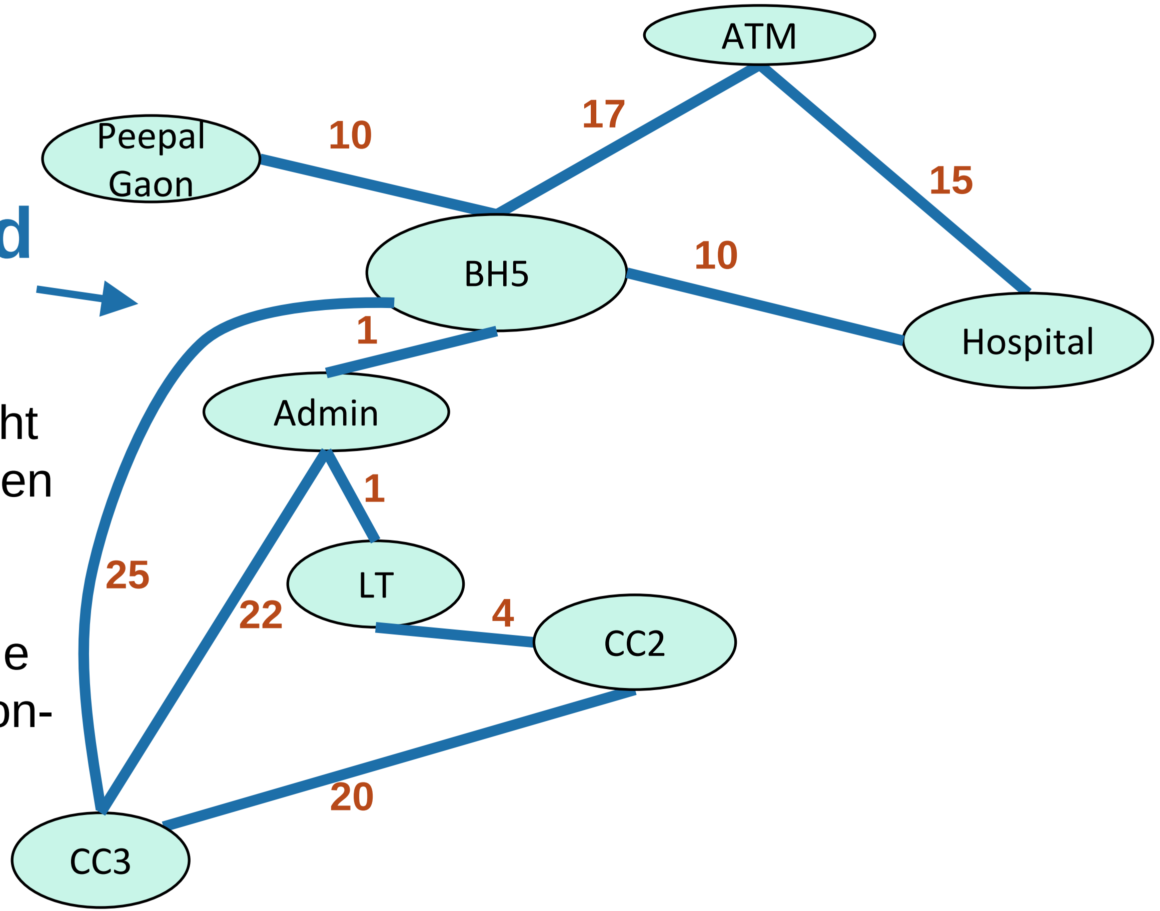
That doesn't make sense if we label the edges by walking time.

Shortest path from BH5 to CC2?

weighted graph

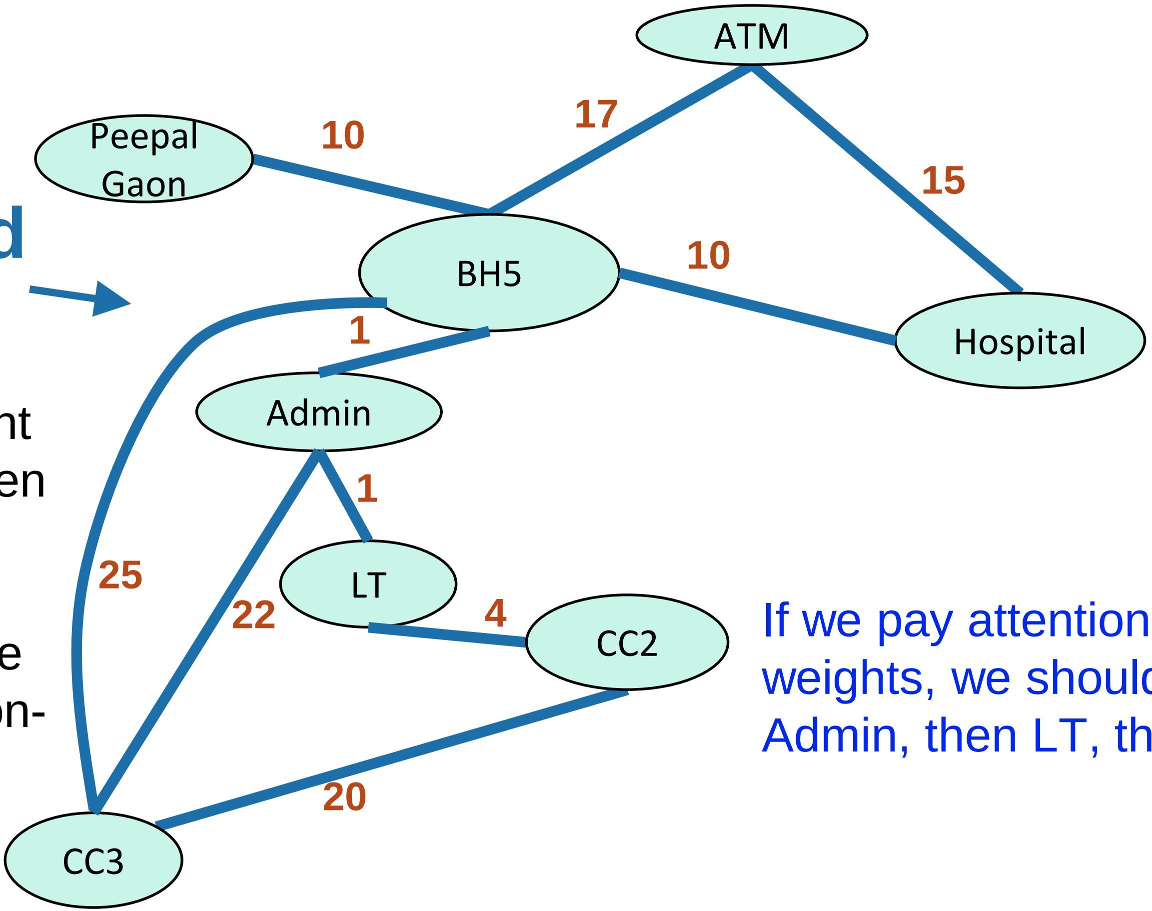
w(u,v) = weight of edge between u and v.

For now, edge weights are non-negative.



Shortest path from BH5 to CC2?

weighted graph



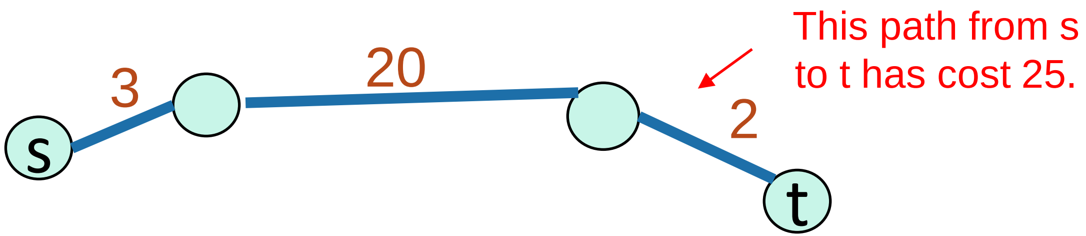
w(u,v) = weight of edge between u and v.

For now, edge weights are non-negative.

If we pay attention to the weights, we should go to the Admin, then LT, then CC2.

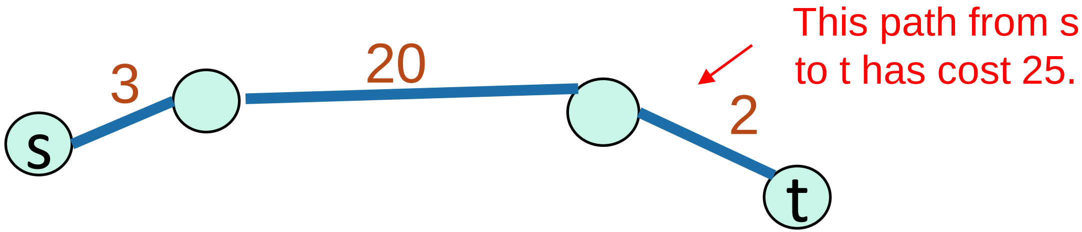
Shortest path problem

- 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



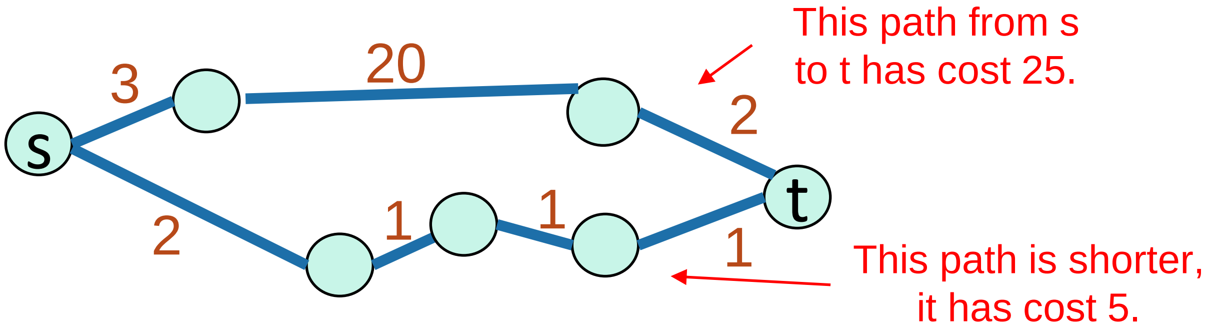
Shortest path problem

- 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.



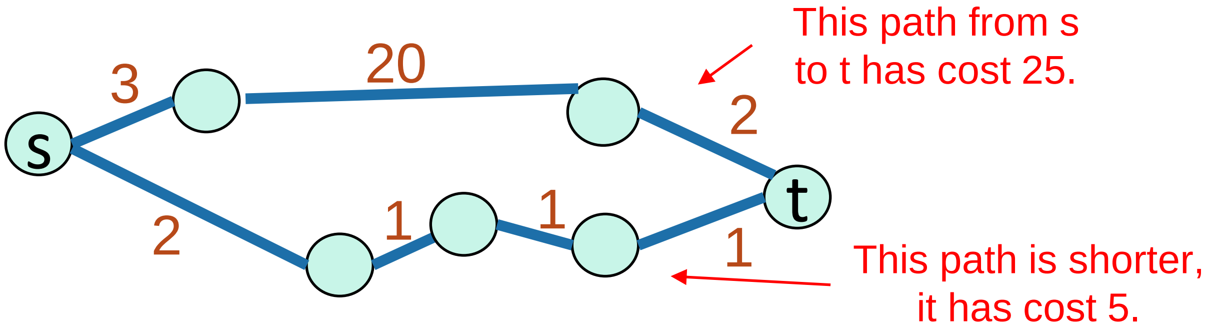
Shortest path problem

- 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.



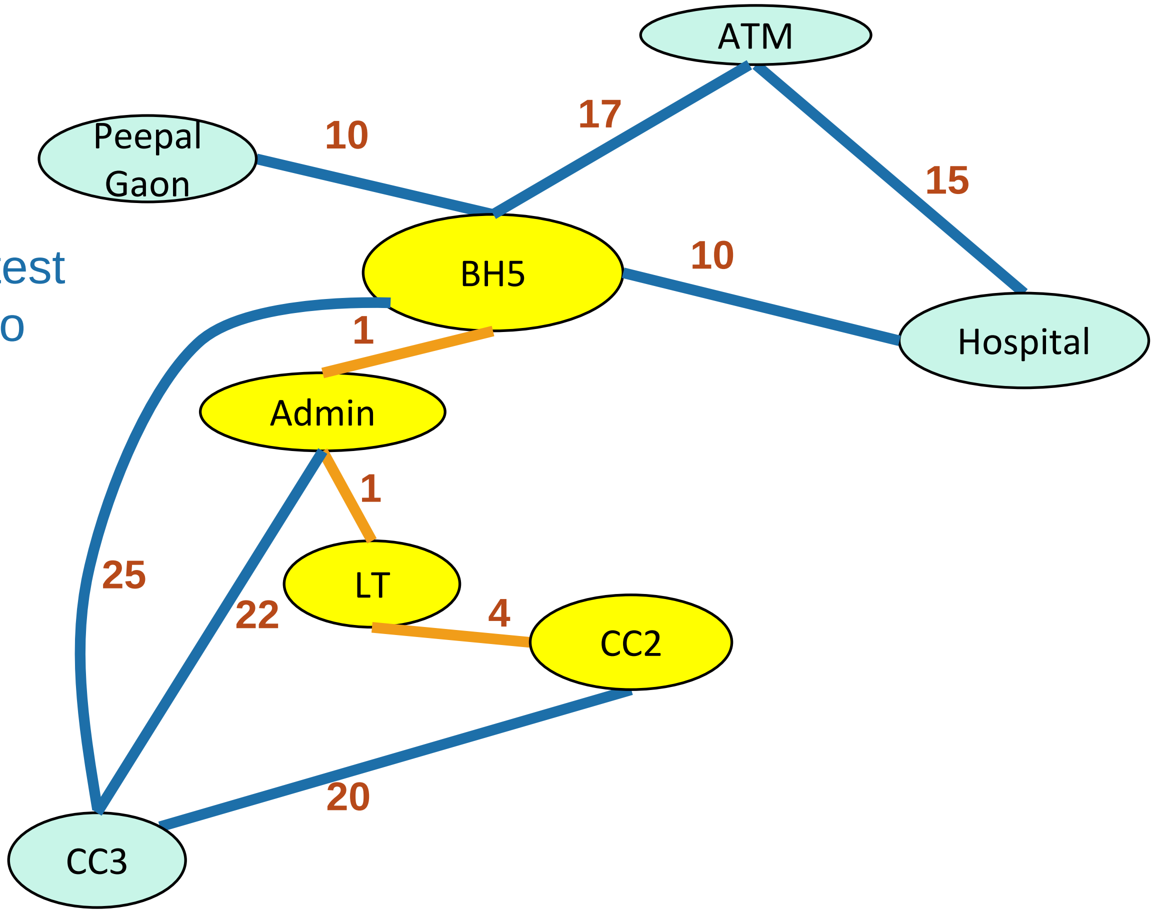
Shortest path problem

- 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 shortest path between u and v.

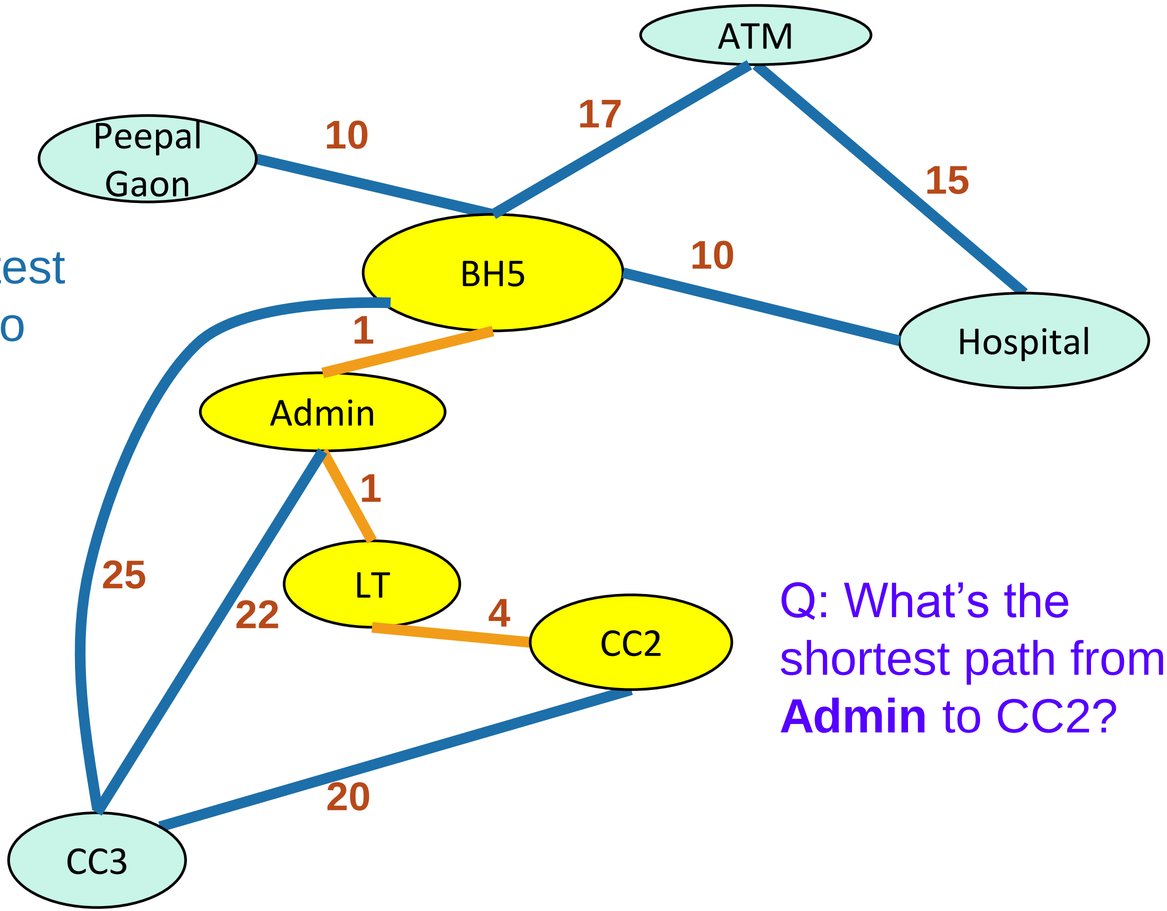
Shortest paths



This is the shortest path from BH5 to CC2.

It has cost 6.

Shortest paths

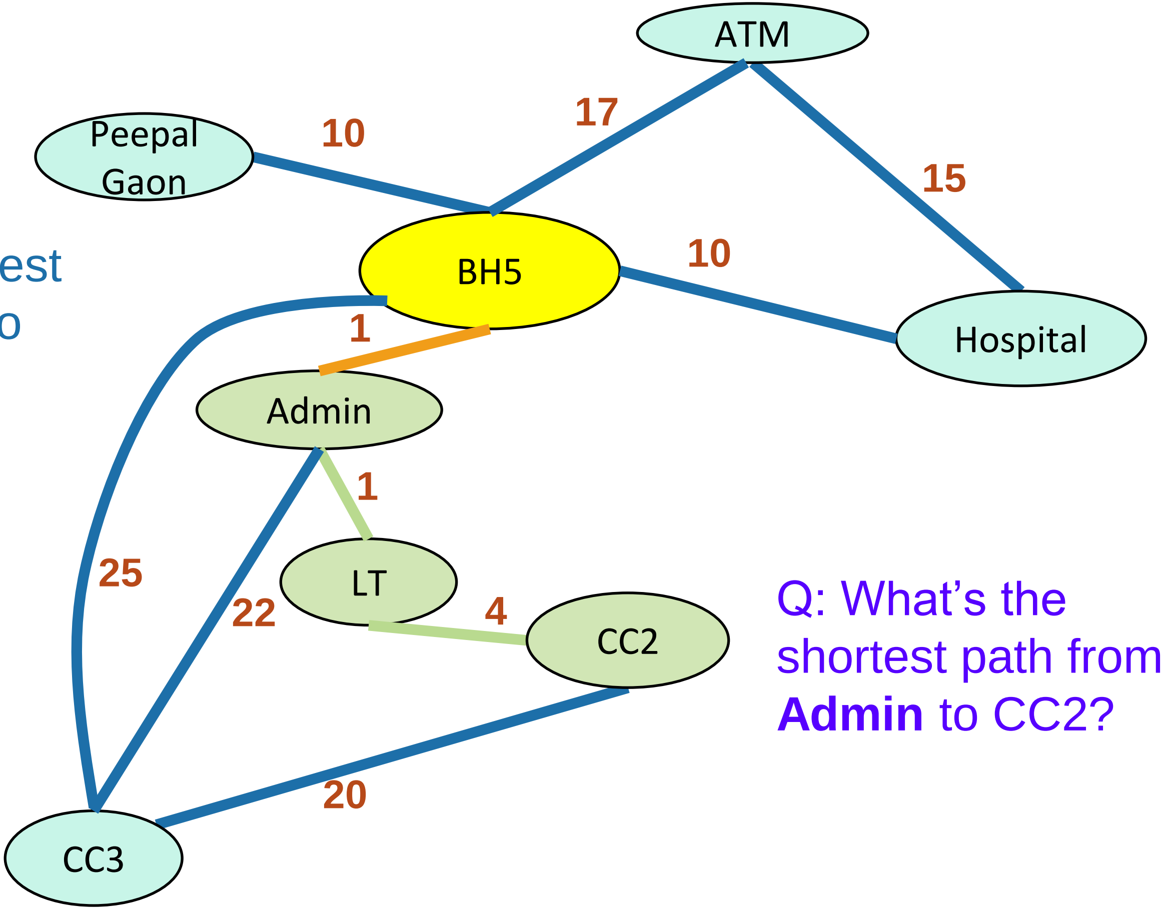


This is the shortest path from BH5 to CC2.

It has cost 6.

Q: What's the shortest path from **Admin** to CC2?

Shortest paths



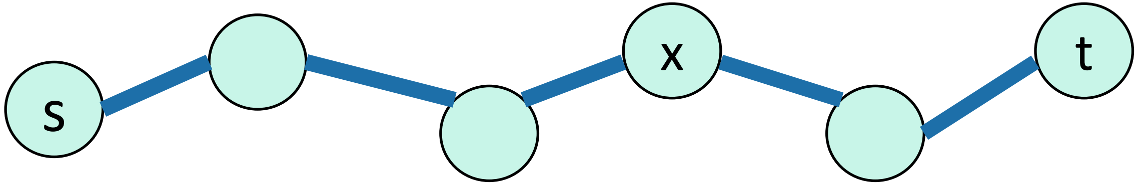
This is the shortest path from BH5 to CC2.

It has cost 6.

Q: What's the shortest path from **Admin** to CC2?

Warm-up

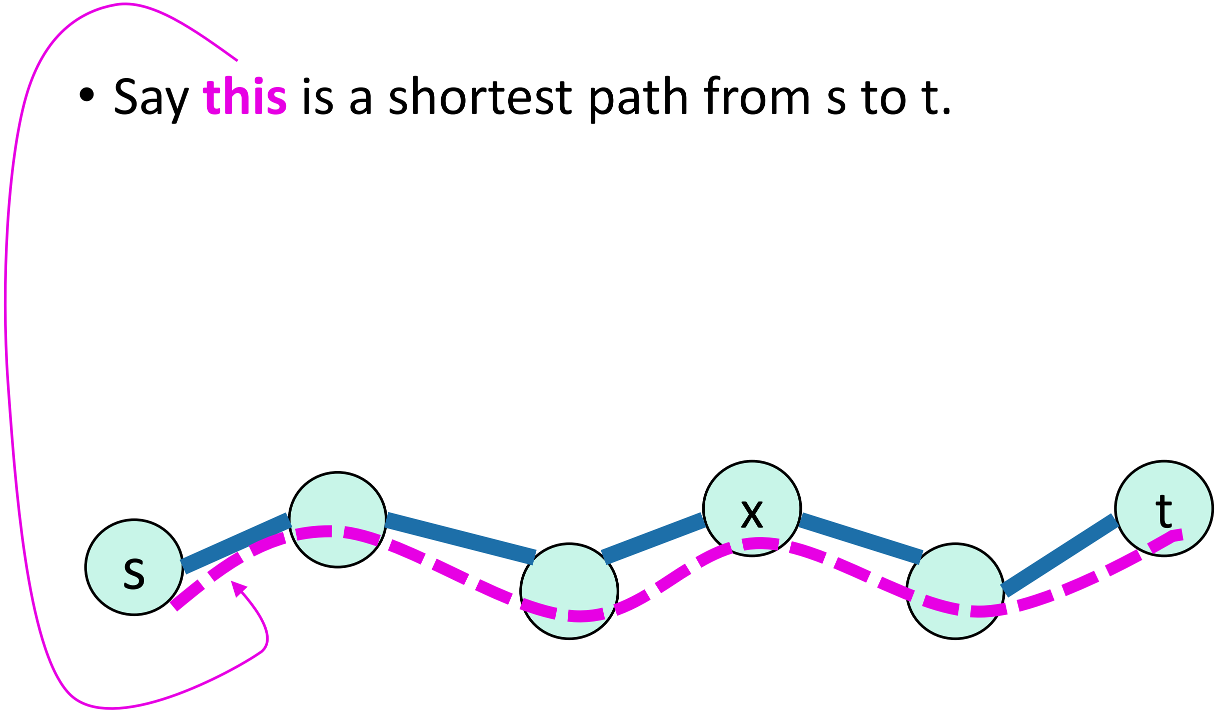
- A sub-path of a shortest path is also a shortest path.



Warm-up

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

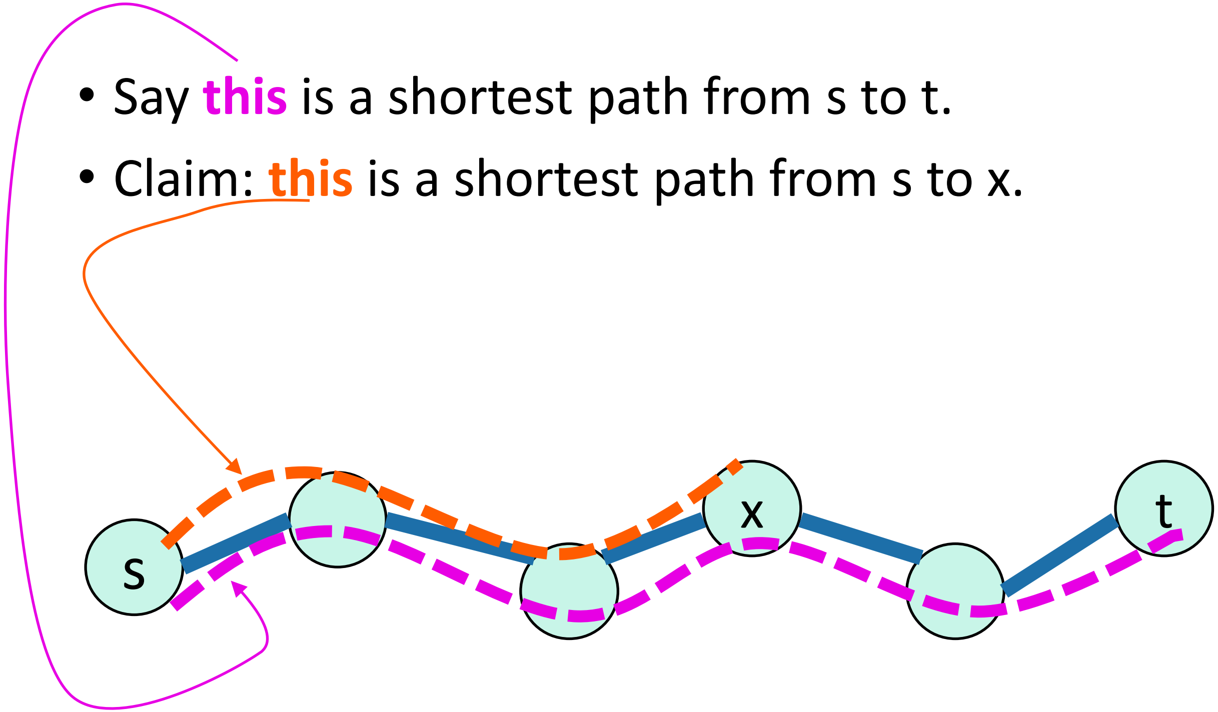
- Say **this** is a shortest path from s to t.



Warm-up

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

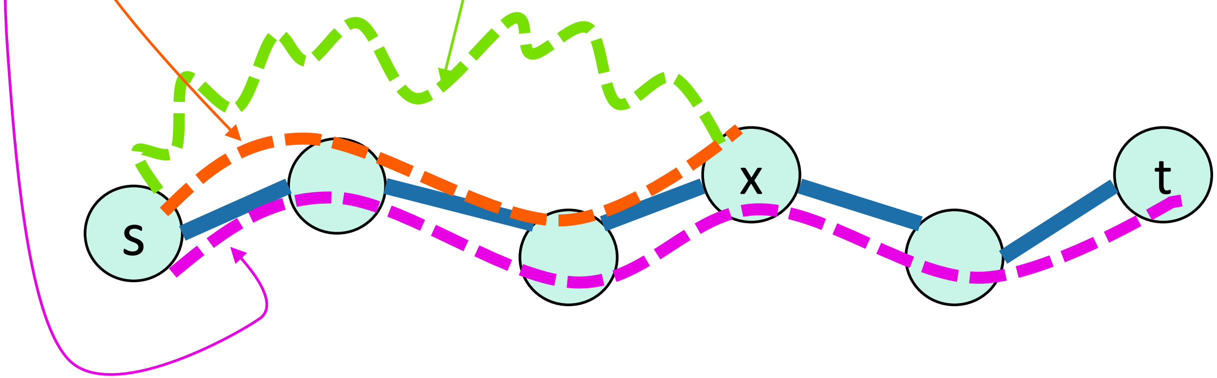
- Say **this** is a shortest path from s to t.
- Claim: **this** is a shortest path from s to x.



Warm-up

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

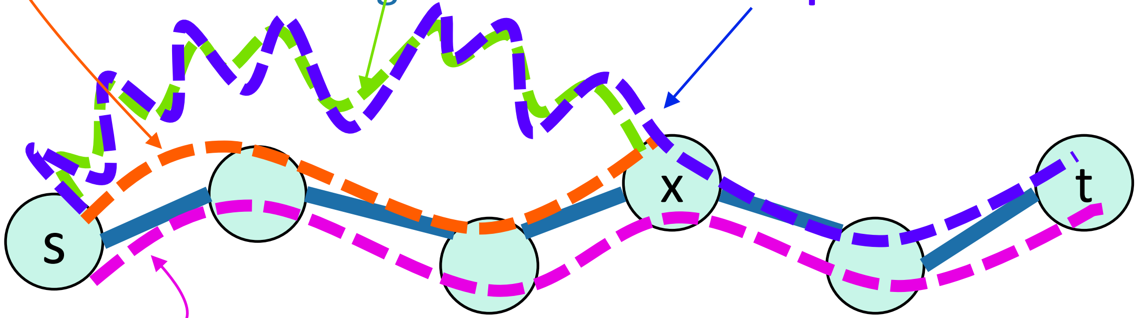
- Say **this** is a shortest path from s to t.
- Claim: **this** is a shortest path from s to x.
  - Suppose not, **this** one is a shorter path from s to x.



Warm-up

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

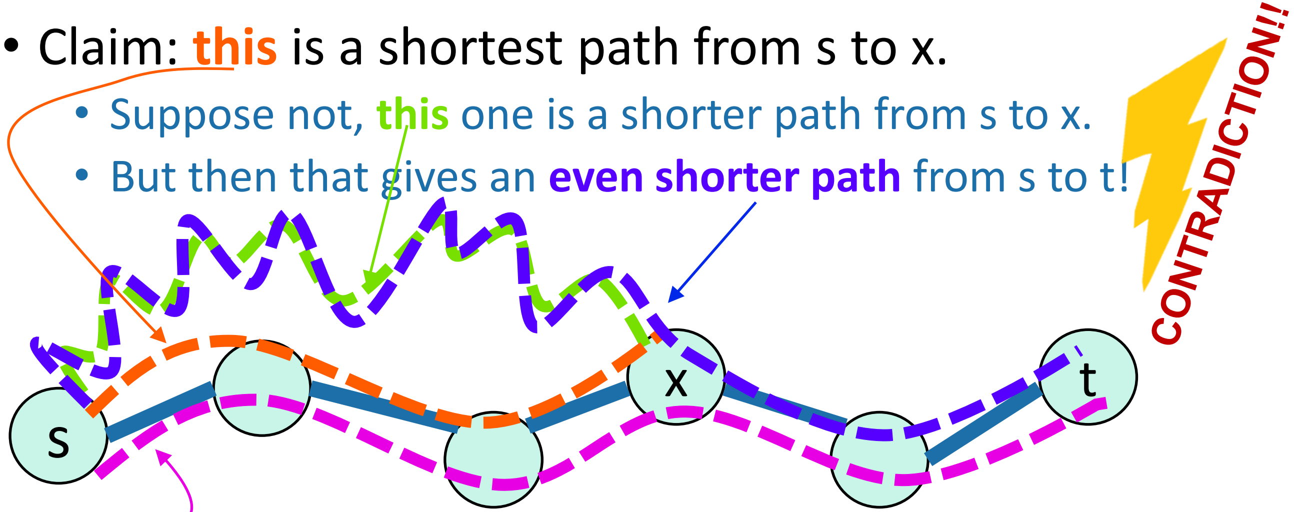
- Say **this** is a shortest path from s to t.
- Claim: **this** is a shortest path from s to x.
  - Suppose not, **this** one is a shorter path from s to x.
  - But then that gives an **even shorter path** from s to t!



Warm-up

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

- Say **this** is a shortest path from s to t.
- Claim: **this** is a shortest path from s to x.
  - Suppose not, **this** one is a shorter path from s to x.
  - 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 (BH5) to all other vertices.

Single-source shortest-path problem

- I want to know the shortest path from one vertex (BH5) to all other vertices.

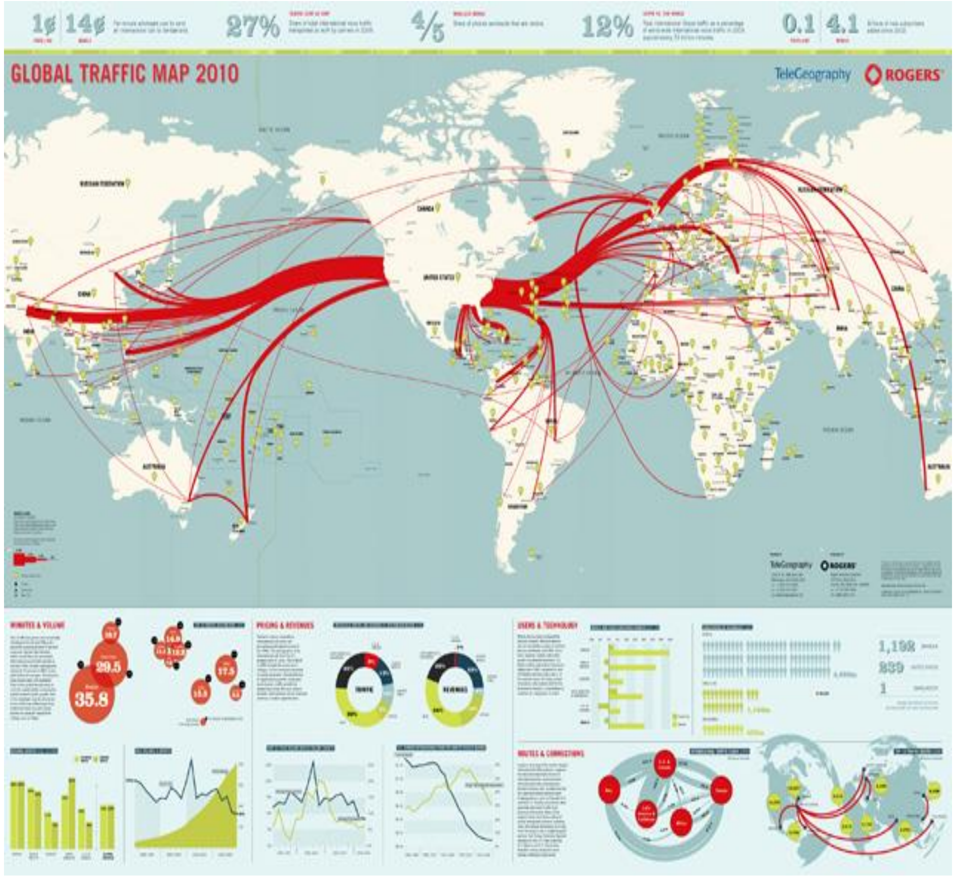
Destination	Cost	To get there
Admin	1	Admin
LT	2	Admin-LT
Peepal Gaon	10	Peepal Gaon
ATM	17	ATM
CC2	6	Admin-LT-CC2
Hospital	10	Hospital
CC3	23	Admin-CC3

Example

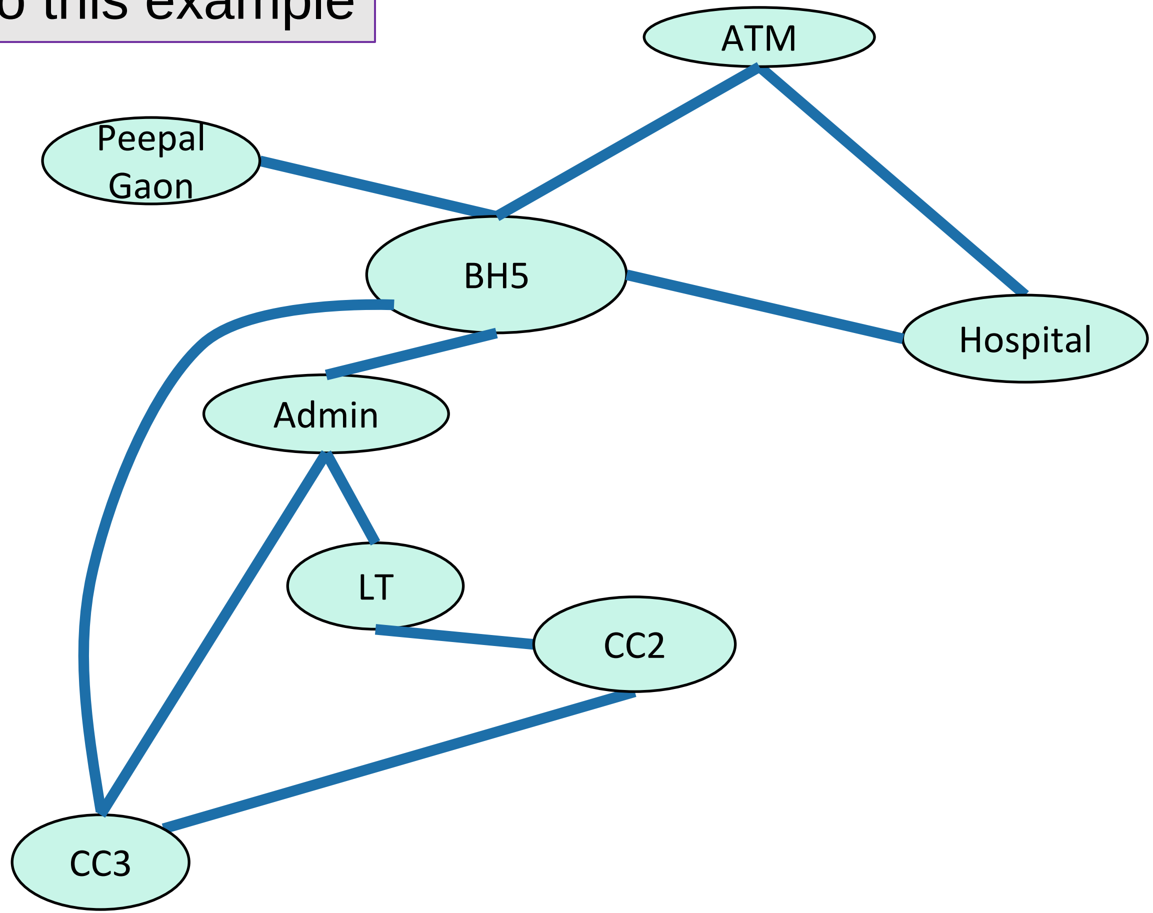
- **“what is the shortest path from IIITA to [anywhere else]”**
- Edge weights have something to do with time, money, hassle.

Example

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

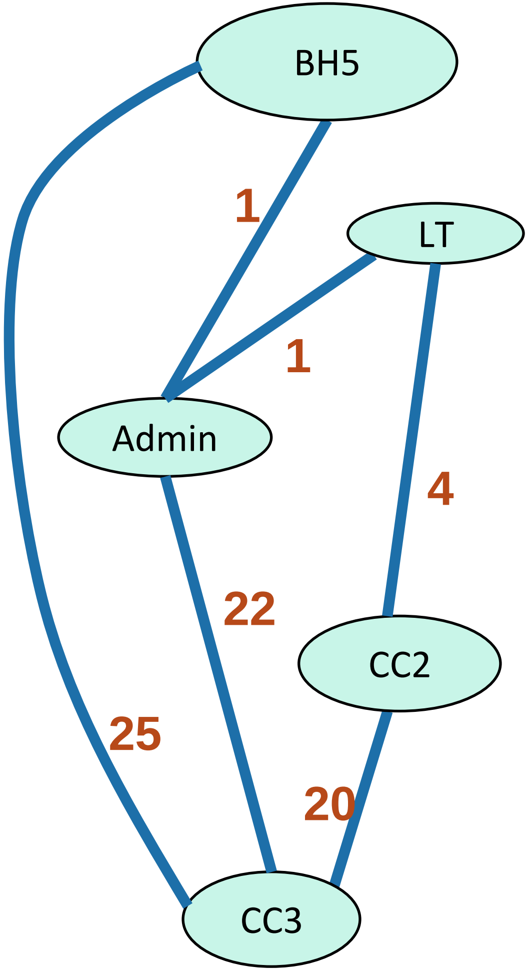


Back to this example



Dijkstra's algorithm

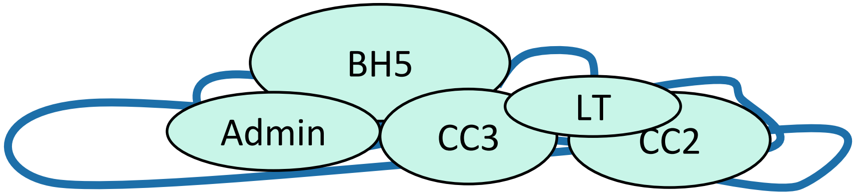
- Finds shortest paths from BH5 to everywhere else.



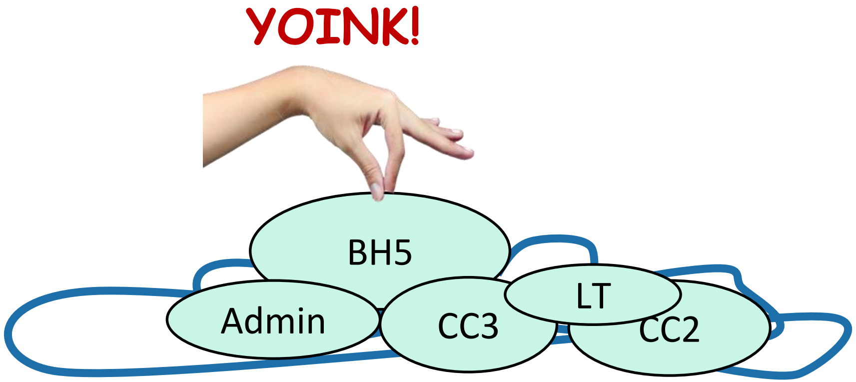
Dijkstra

intuition

All vertices are on ground initially.

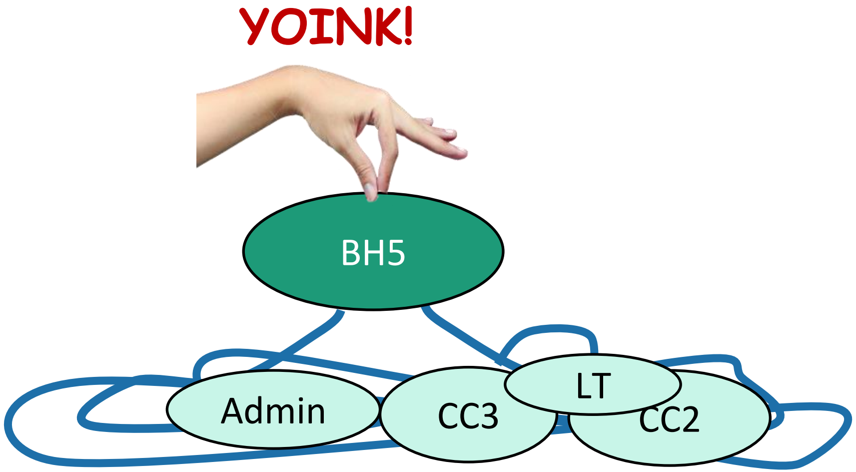


Dijkstra
intuition



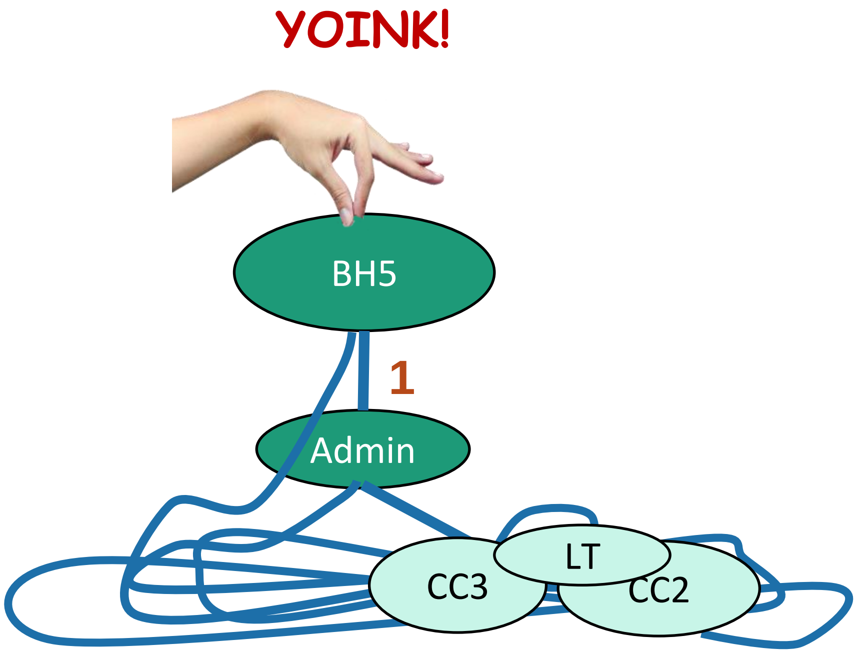
Dijkstra
intuition

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

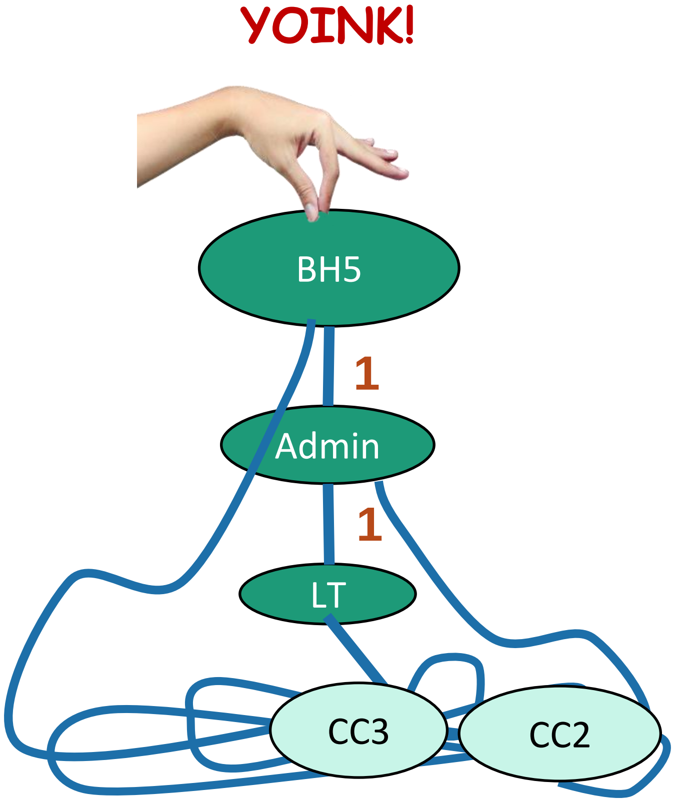


Dijkstra

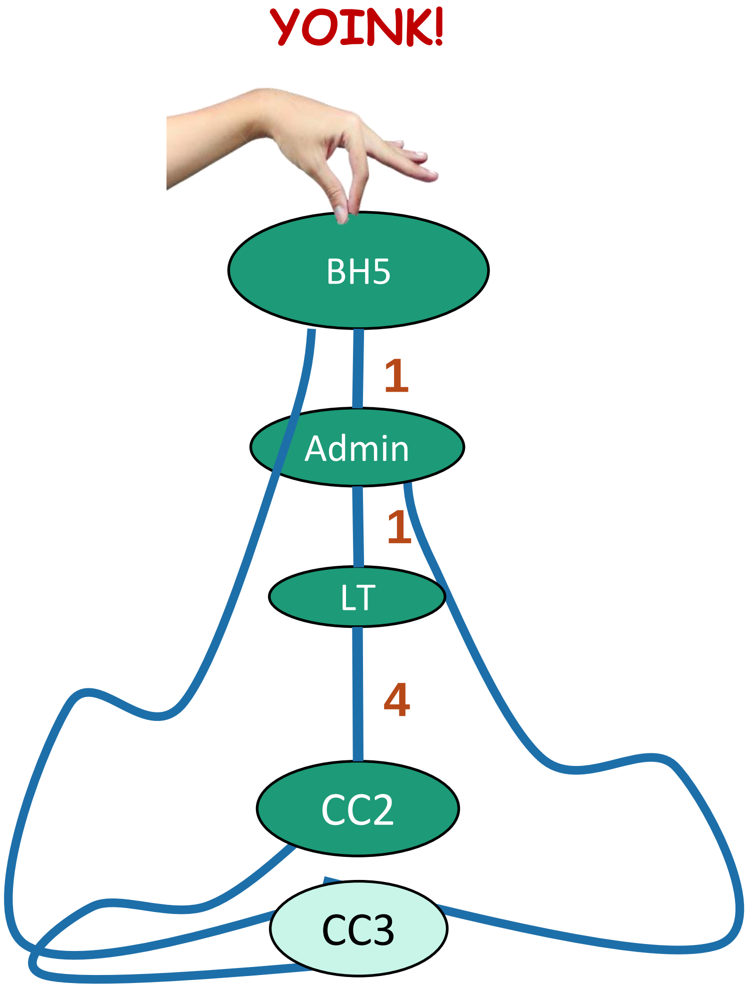
intuition



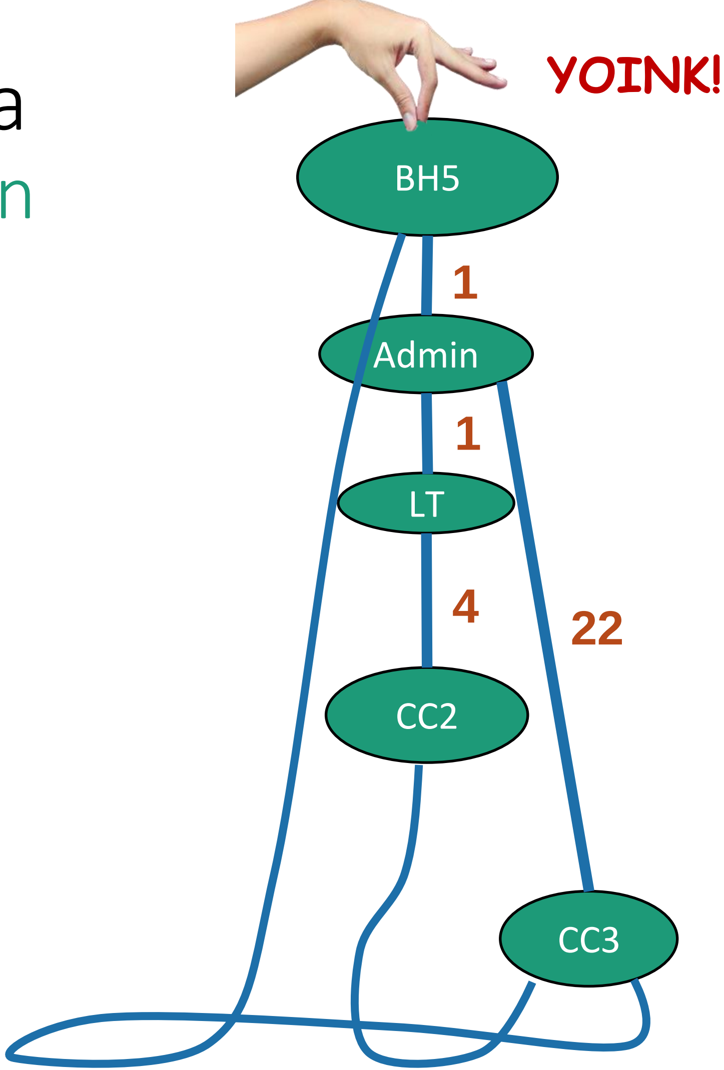
Dijkstra
intuition



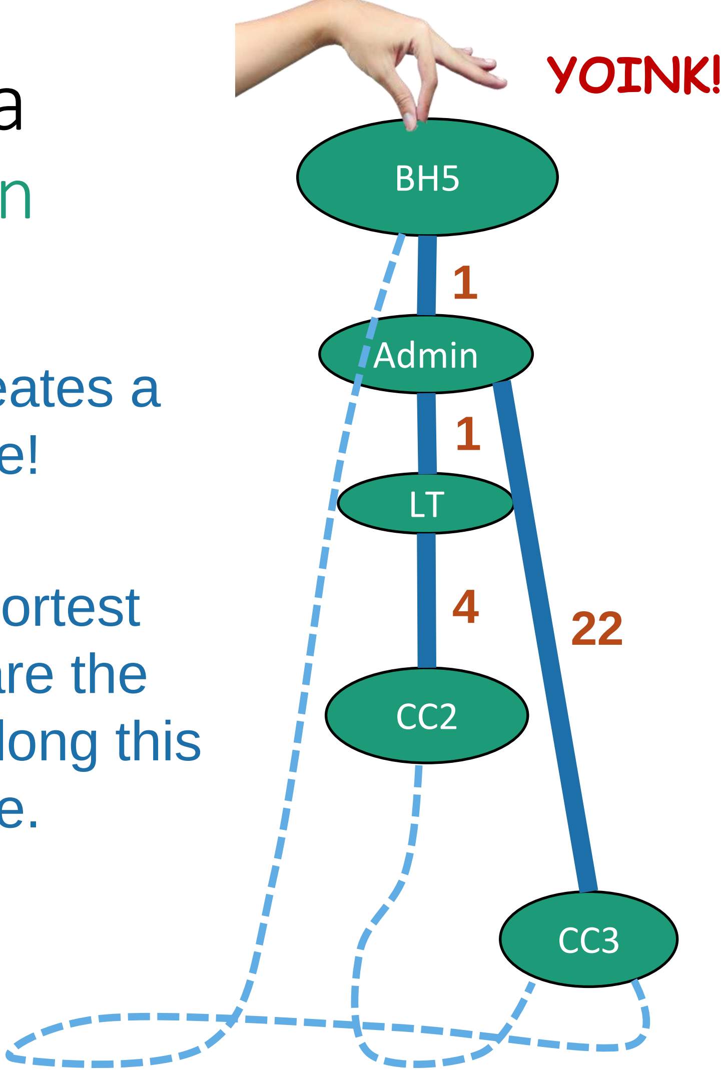
Dijkstra
intuition



Dijkstra
intuition



Dijkstra intuition

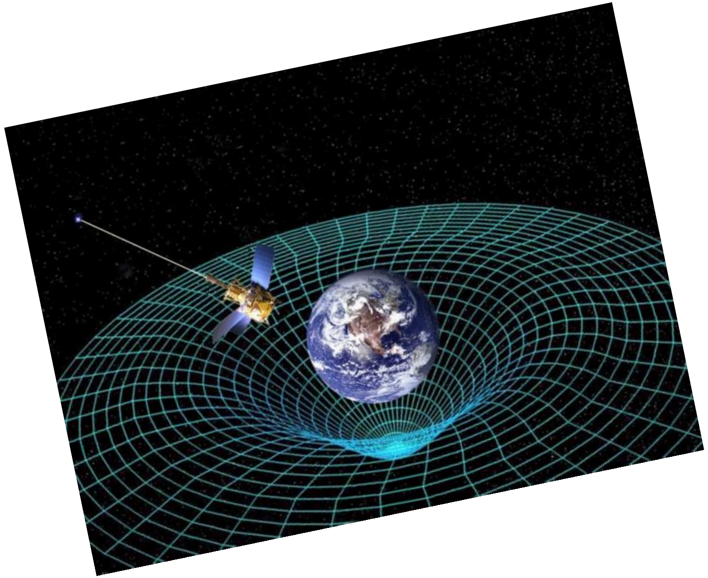


This creates a tree!

The shortest paths are the lengths along this tree.

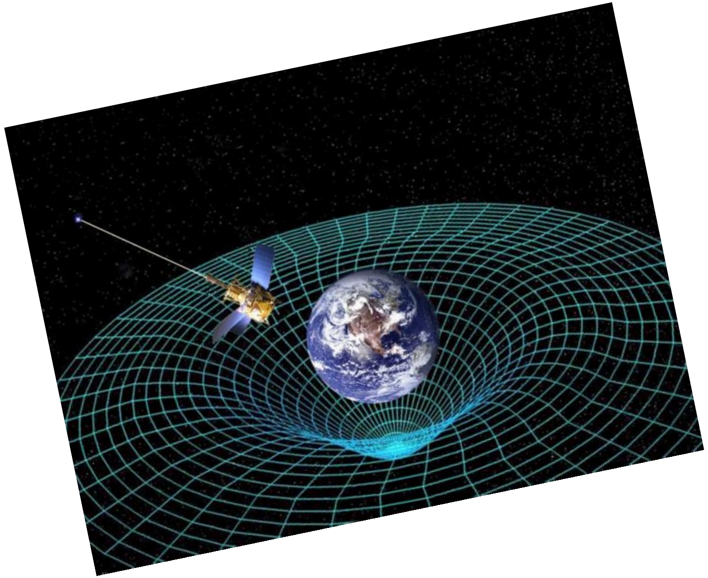
How do we actually implement this?

How do we actually implement this?



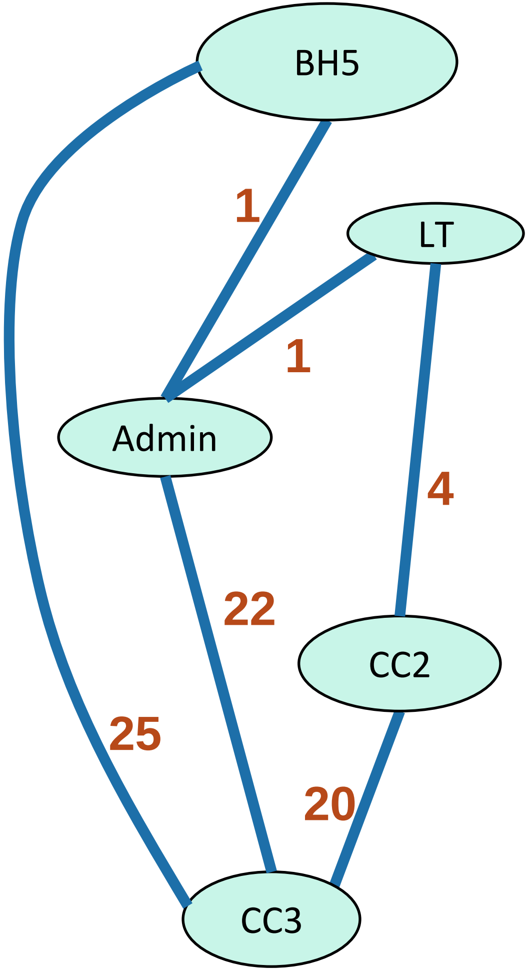
How do we actually implement this?

- **Without** string and gravity?



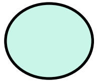
Dijkstra by example

How far is a node from BH5?

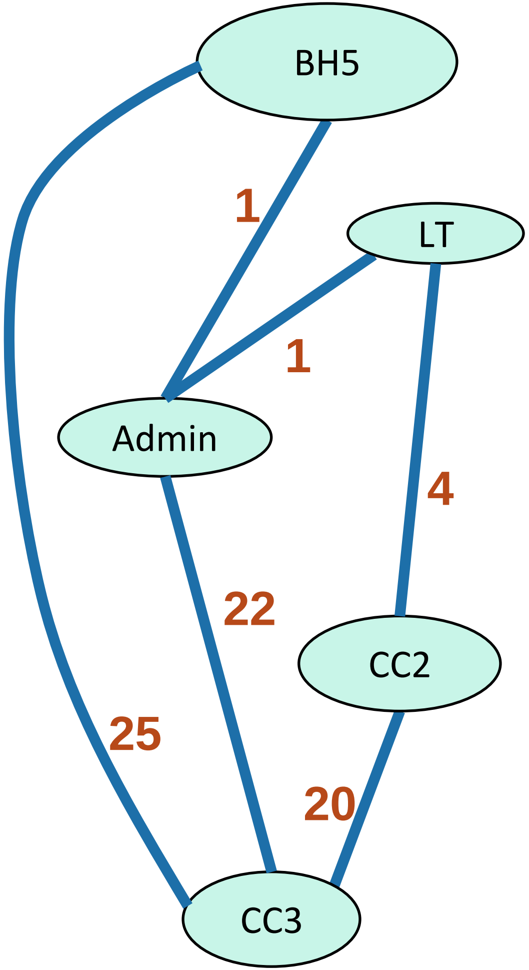


Dijkstra by example

How far is a node from BH5?

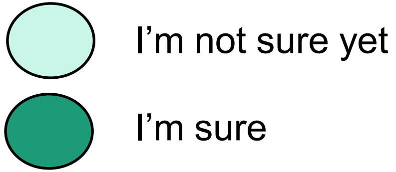


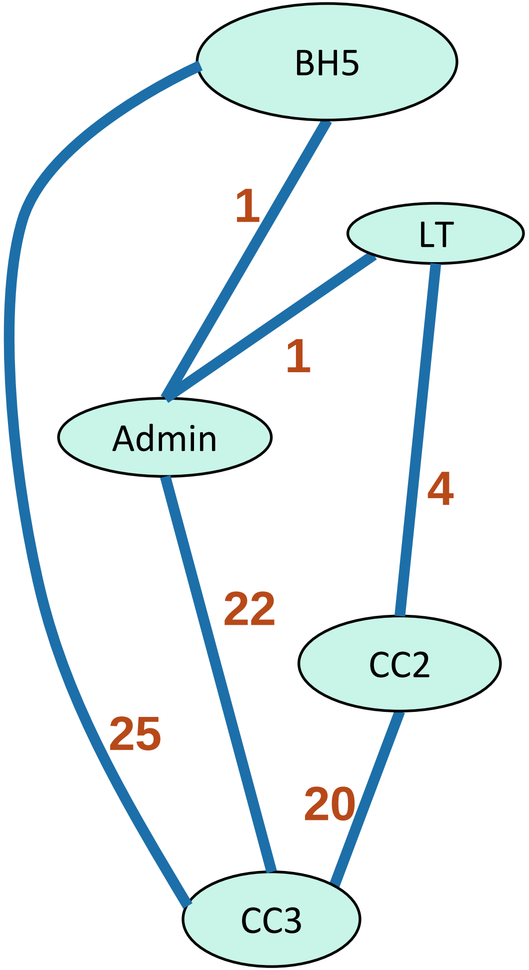
I'm not sure yet



Dijkstra by example

How far is a node from BH5?





Dijkstra by example

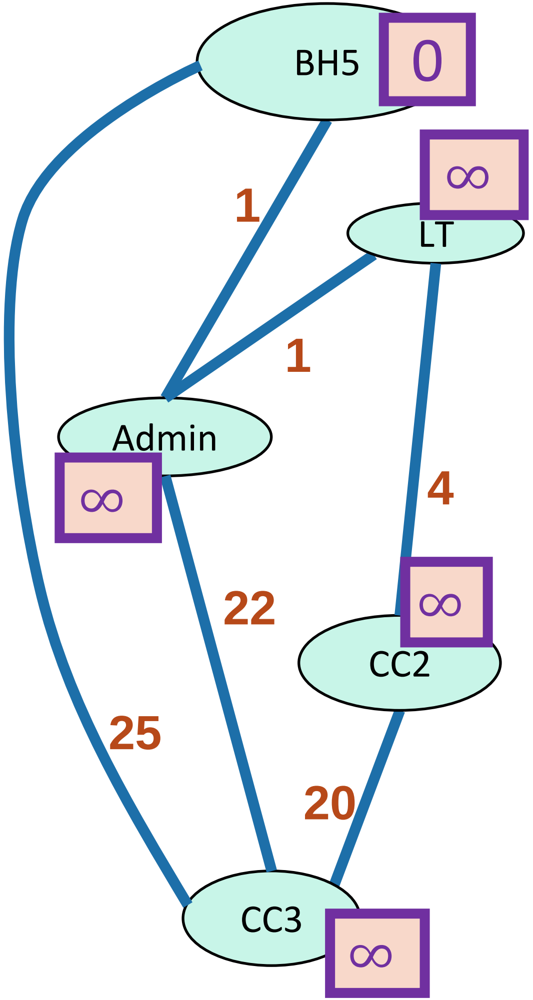
How far is a node from BH5?

I'm not sure yet

I'm sure

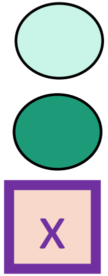
x **x = d[v]** is my best **over-estimate** for dist(BH5,v).

Initialize d[v] = ∞
for all non-starting vertices v,
and d[BH5] = 0



Dijkstra by example

How far is a node from BH5?



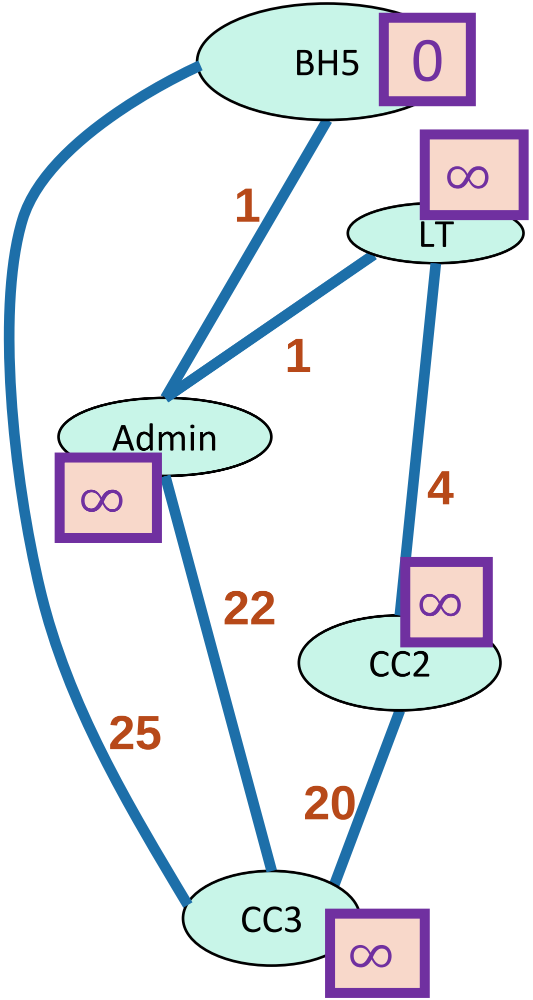
I'm not sure yet

I'm sure

x = d[v] is my best **over-estimate** for dist(BH5,v).

Initialize d[v] = ∞
for all non-starting vertices v,
and d[BH5] = 0

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



Dijkstra by example

How far is a node from BH5?

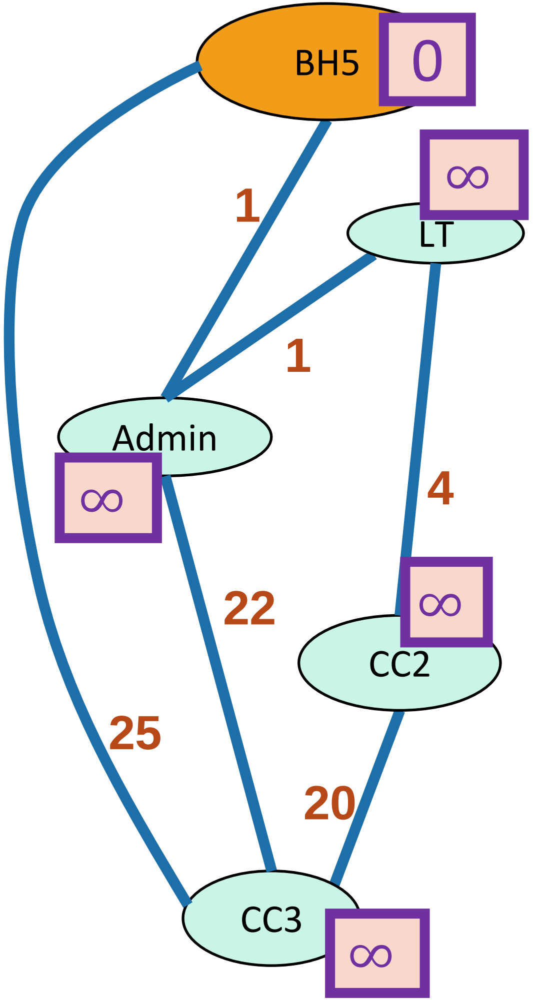
I'm not sure yet

I'm sure

x — **x = d[v]** is my best **over-estimate** for dist(BH5,v).

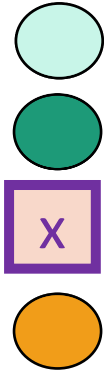
Current node u

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



Dijkstra by example

How far is a node from BH5?



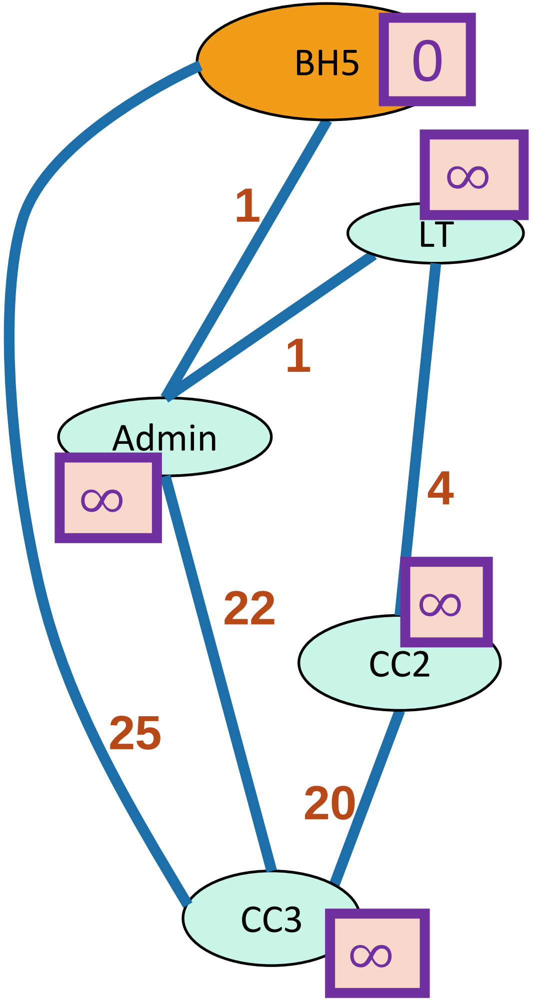
I'm not sure yet

I'm sure

x = d[v] is my best **over-estimate** for dist(BH5,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 BH5?

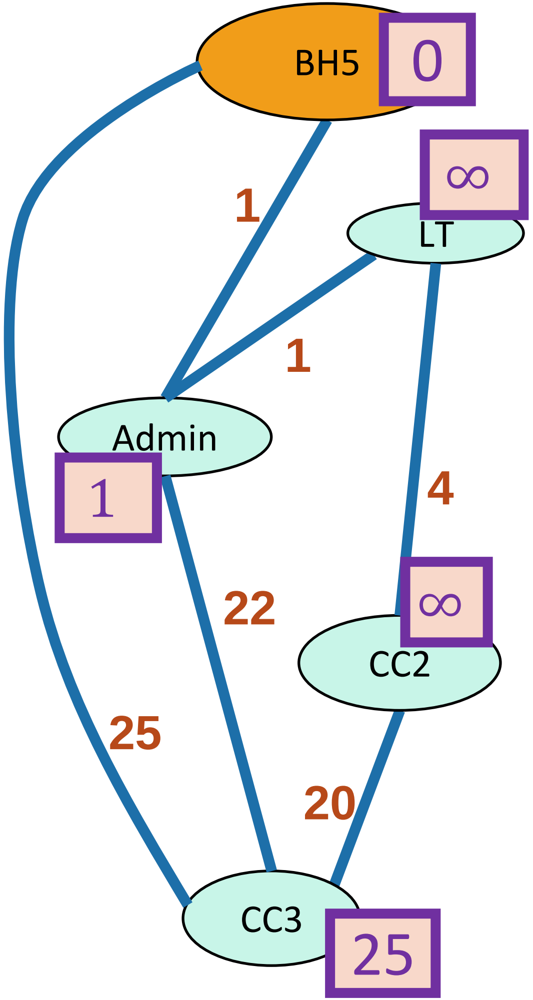
I'm not sure yet

I'm sure

x **x = d[v]** is my best **over-estimate** for dist(BH5,v).

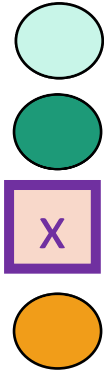
Current node u

- 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 BH5?



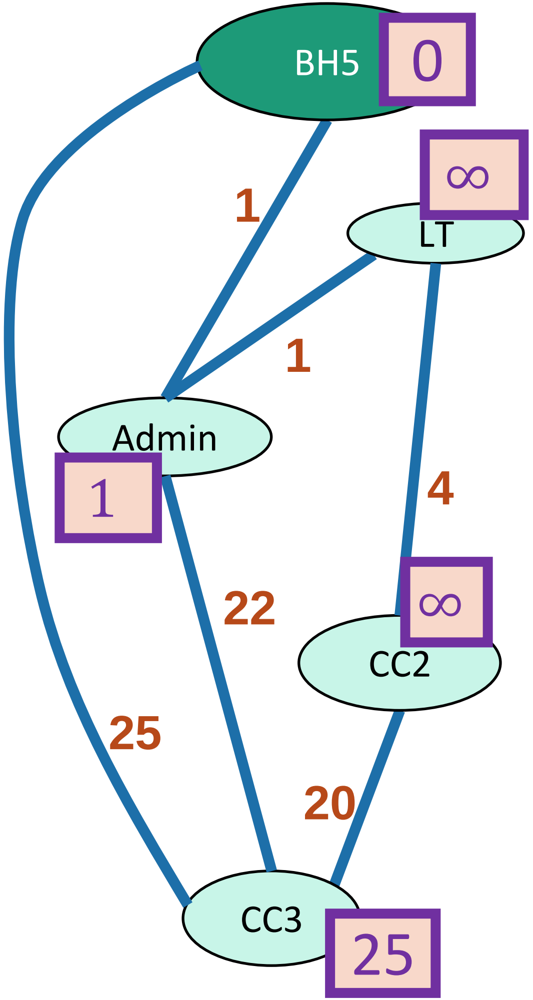
I'm not sure yet

I'm sure

x = d[v] is my best **over-estimate** for dist(BH5,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 BH5?

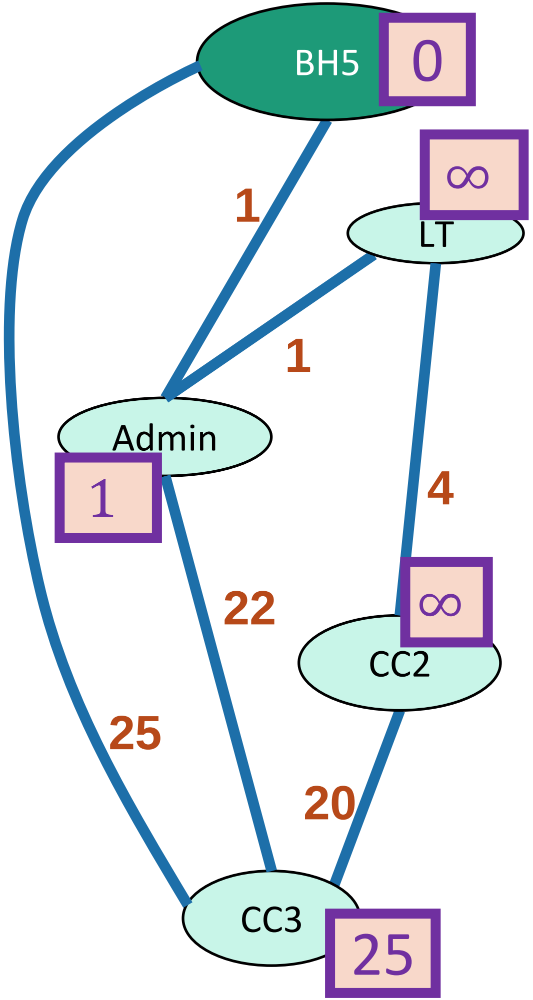
I'm not sure yet

I'm sure

x **x = d[v]** is my best **over-estimate** for dist(BH5,v).

Current node u

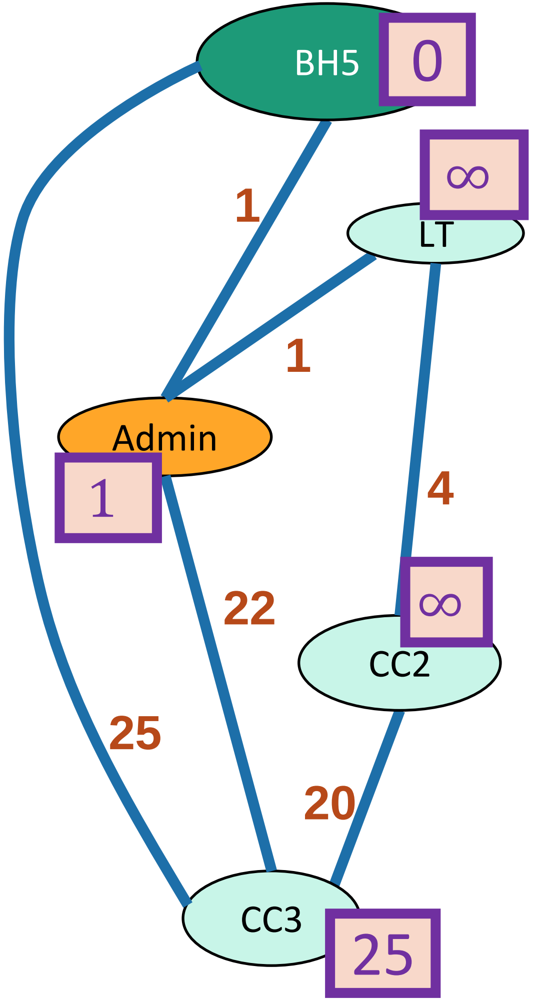
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra by example

How far is a node from BH5?

- I'm not sure yet
- I'm sure
- x — **x = d[v]** is my best **over-estimate** for dist(BH5,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

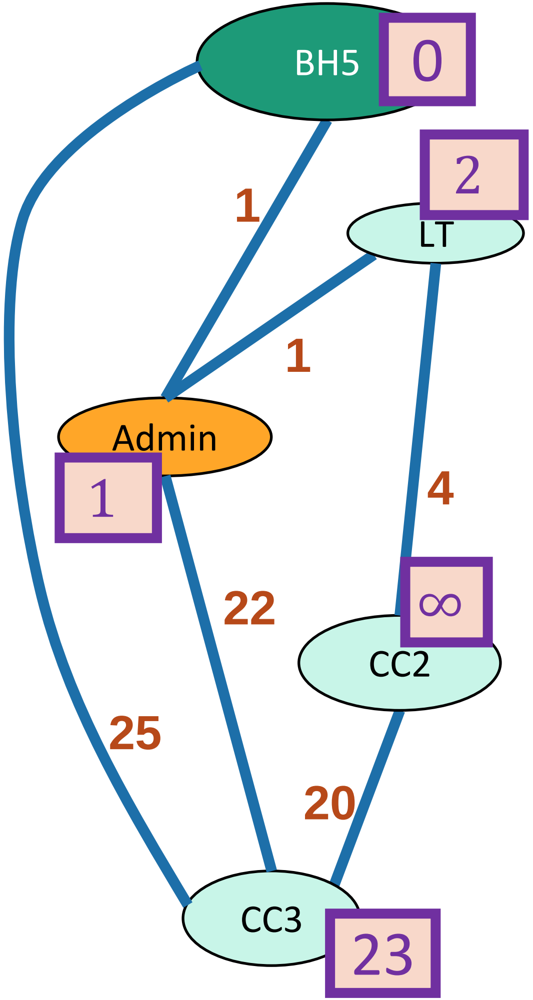
How far is a node from BH5?

I'm not sure yet

I'm sure

x **x = d[v]** is my best **over-estimate** for dist(BH5,v).

Current node u



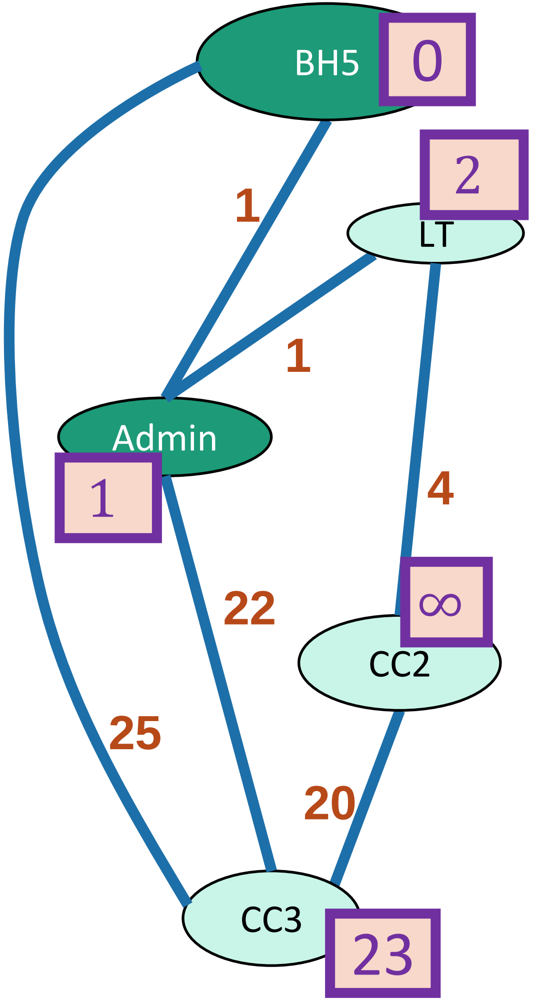
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from BH5?

- I'm not sure yet
- I'm sure
- **x = d[v]** is my best **over-estimate** for dist(BH5,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

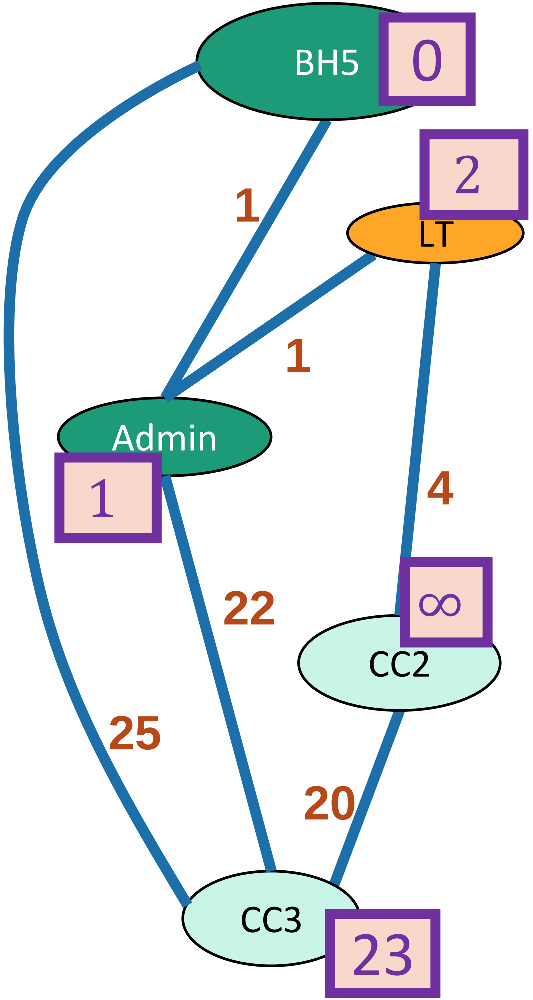
How far is a node from BH5?

I'm not sure yet

I'm sure

x — **x = d[v]** is my best **over-estimate** for dist(BH5,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 BH5?

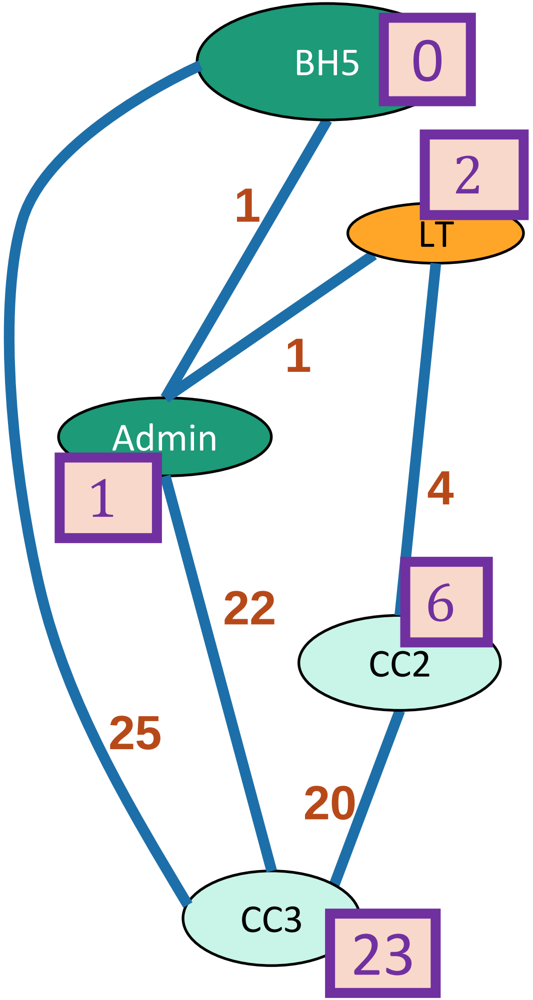
I'm not sure yet

I'm sure

x = **d[v]** is my best **over-estimate** for dist(BH5,v).

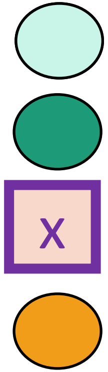
Current node u

- 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 BH5?

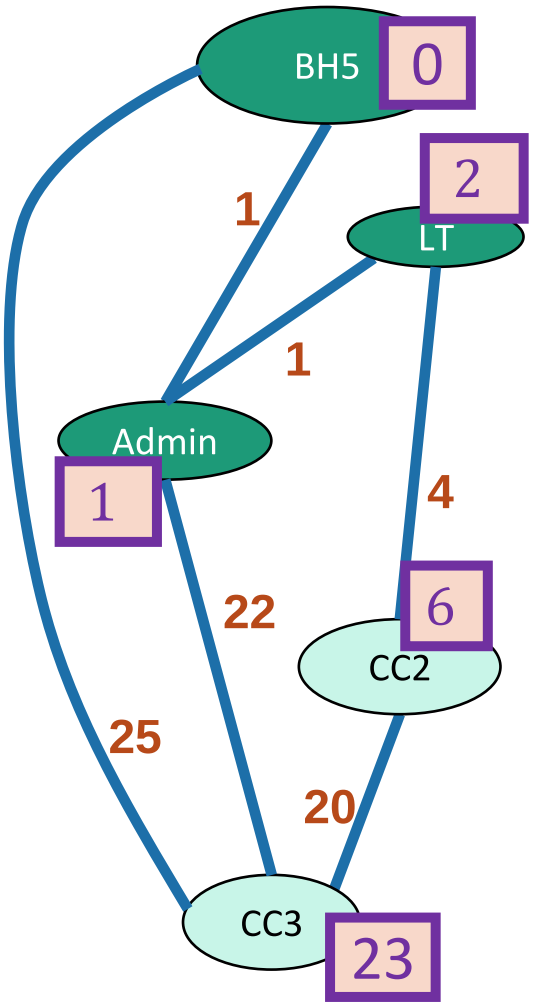


I'm not sure yet

I'm sure

x = d[v] is my best **over-estimate** for dist(BH5,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 BH5?

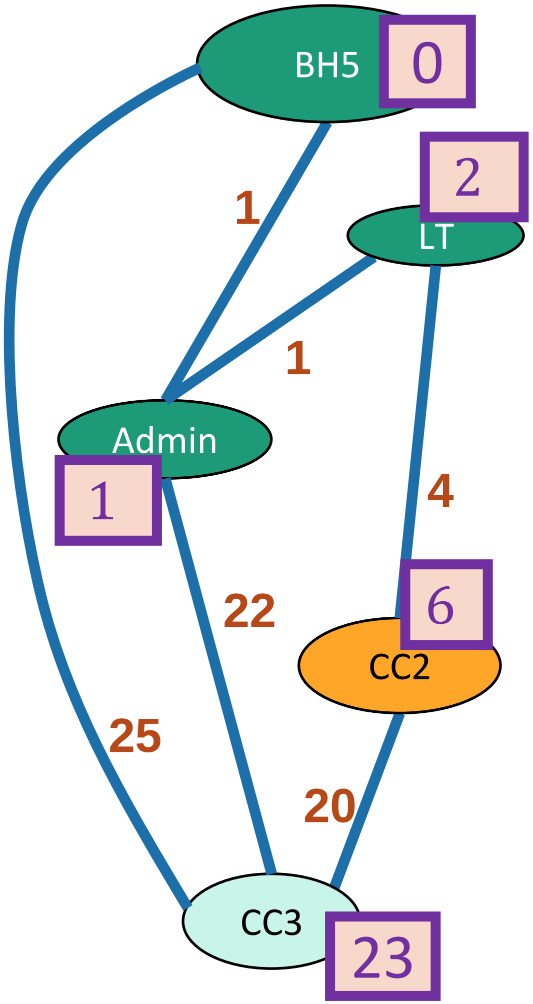
I'm not sure yet

I'm sure

x

x = d[v] is my best **over-estimate** for dist(BH5,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 BH5?

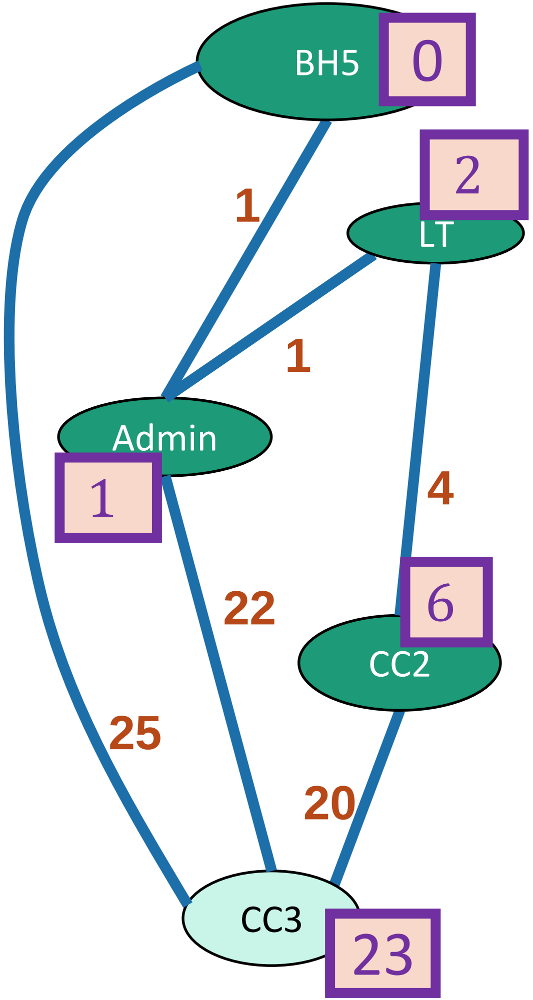
I'm not sure yet

I'm sure

x — **x = d[v]** is my best **over-estimate** for dist(BH5,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

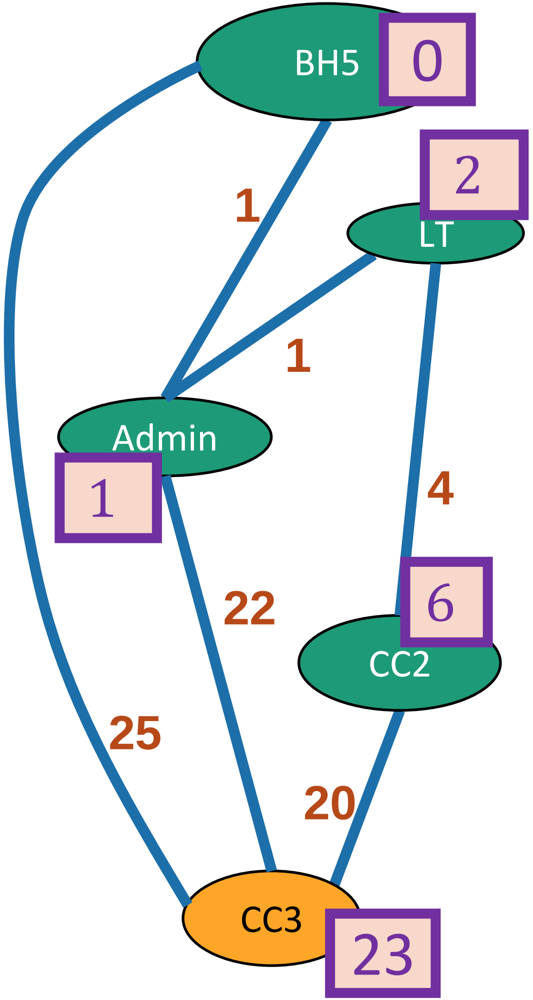
How far is a node from BH5?

I'm not sure yet

I'm sure

x **x = d[v]** is my best **over-estimate** for dist(BH5,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

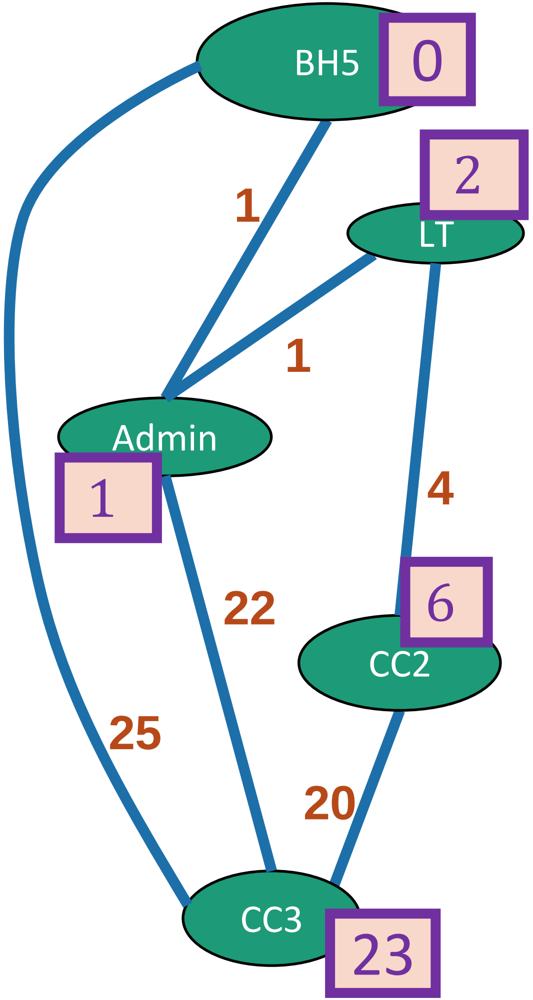
How far is a node from BH5?

I'm not sure yet

I'm sure

x — **x = d[v]** is my best **over-estimate** for dist(BH5,v).

Current node u

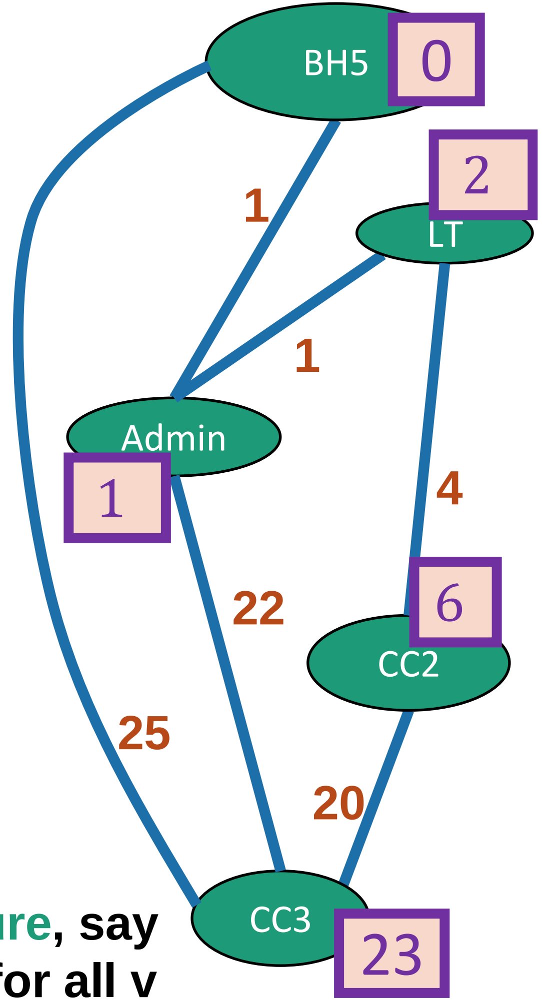


- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from BH5?

- I'm not sure yet
- I'm sure
- x — **x = d[v]** is my best **over-estimate** for dist(BH5,v).
- Current node u



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

- **After all nodes are sure, say that d(BH5, v) = d[v] for all v**

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 $d(s, v) = d[v]$

Lots of implementation details left un-explained.
We’ll get to that!

As usual

- Does it work?

- Is it fast?

As usual

- Does it work?
  - Yes.

- Is it fast?
  - Depends on how you implement it.

As usual

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

Why does this work?

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

Let's rename "BH5" to "s", our starting vertex.

Why does this work?

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

Let's rename "BH5" 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)**.

Why does this work?

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

Let's rename "BH5" 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.**
  - When v is marked **sure**, **d[v] = d(s,v).** ← Claim 2
  - **d[v] $\geq$ d(s,v)** and never increases, so after v is **sure**, **d[v]** stops changing. ← Claim 1 + def of algorithm
  - This implies that at any time *after* v is marked **sure**, **d[v] = d(s,v).**
  - All vertices are **sure** at the end, so all vertices end up with **d[v] = d(s,v).**

Why does this work?

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

Let's rename "BH5" to "s", our starting vertex.

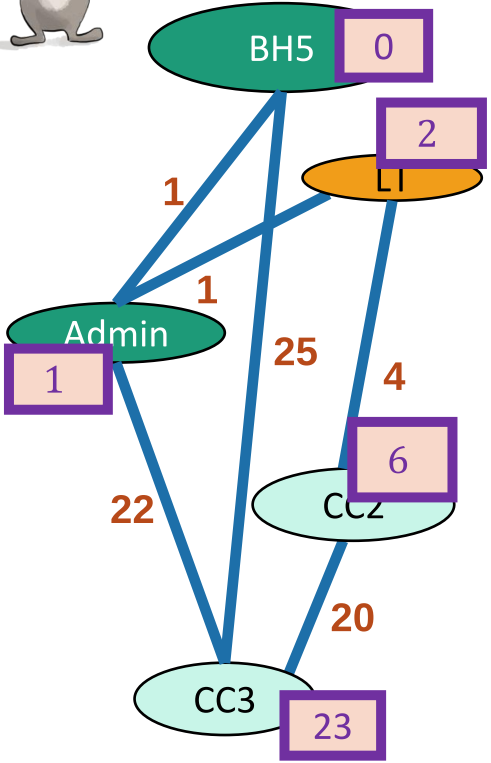
- Proof outline:
  - **Claim 1**: For all v, **d[v] $\geq$ d(s,v).**
  - **Claim 2**: When a vertex v is marked **sure**, **d[v] = d(s,v)**.

- **Claims 1 and 2** imply the **theorem.**
  - When v is marked **sure**, **d[v] = d(s,v).** ← Claim 2
  - **d[v] $\geq$ d(s,v)** and never increases, so after v is **sure**, **d[v]** stops changing. ← Claim 1 + def of algorithm
  - This implies that at any time *after* v is marked **sure**, **d[v] = d(s,v).**
  - All vertices are **sure** at the end, so all vertices end up with **d[v] = d(s,v).**

Next let's prove the claims!

Claim 1

d[v] ≥ d(s,v) for all v.

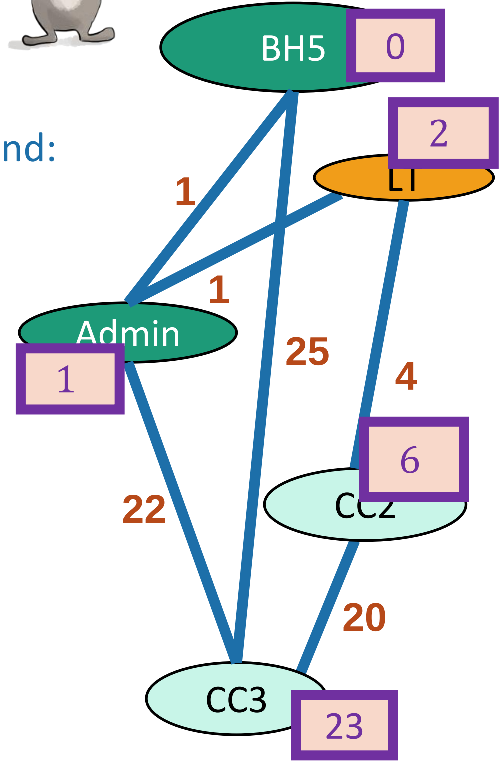


Claim 1

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

Informally:

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



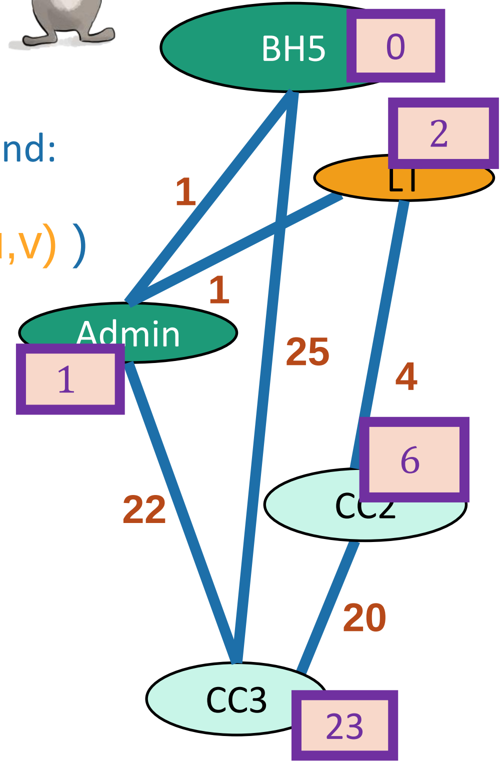
Claim 1

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

Informally:

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

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



Intuition!

Claim 1

d[v] ≥ d(s,v) for all v.

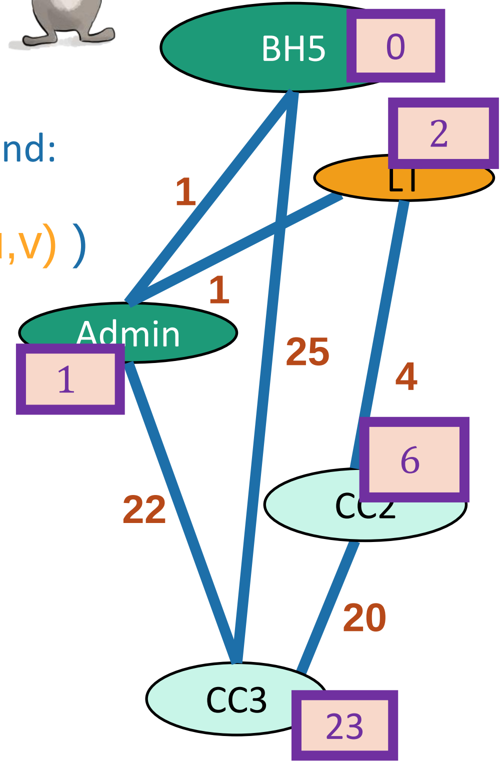
Informally:

- 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.



Claim 1

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

Informally:

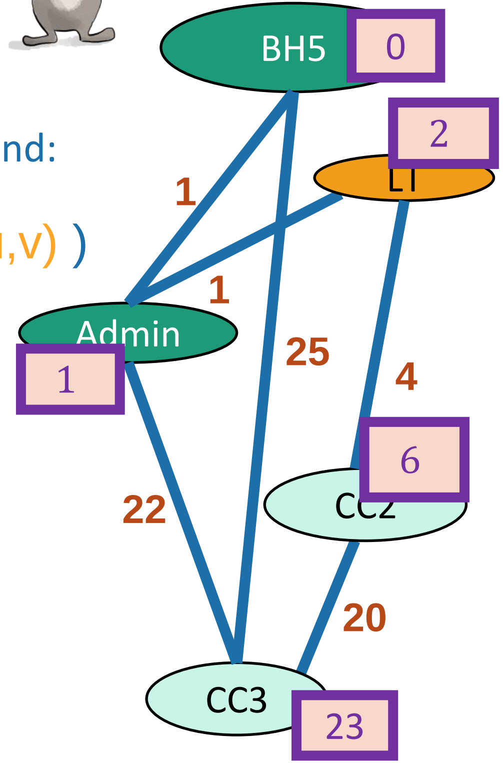
- Every time we update d[v], we have a path in mind:

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

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)



Claim 1

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

Informally:

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

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

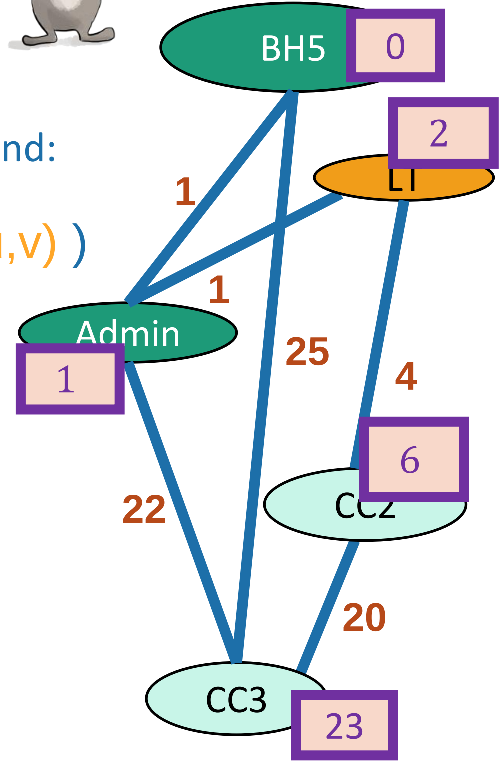
Whatever path we had in mind before

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)



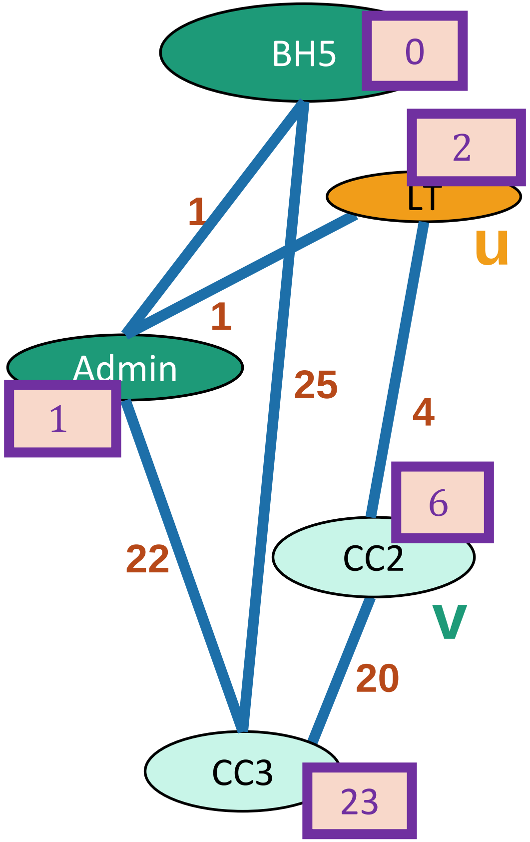
Formally:

- We should prove this by induction.

Claim 1

d[v] ≥ d(s,v) for all v.

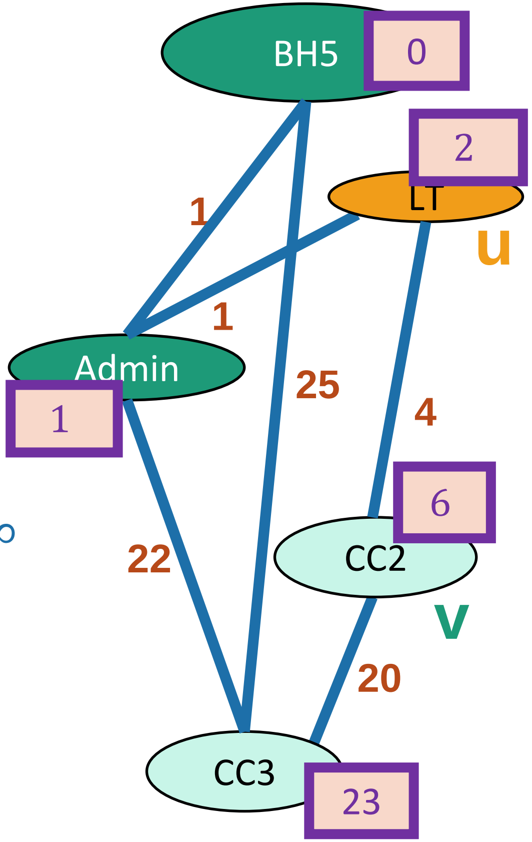
- Inductive hypothesis.
  - After t iterations of Dijkstra, d[v] ≥ d(s,v) for all v.



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, $\mathrm{d}(s,s) = 0$, and $d(s,v) \leq \infty$



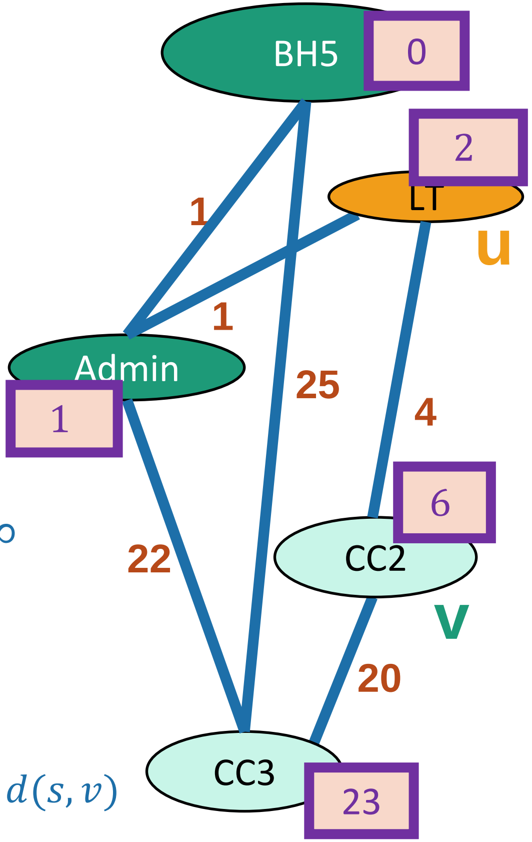
Claim 1

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

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

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

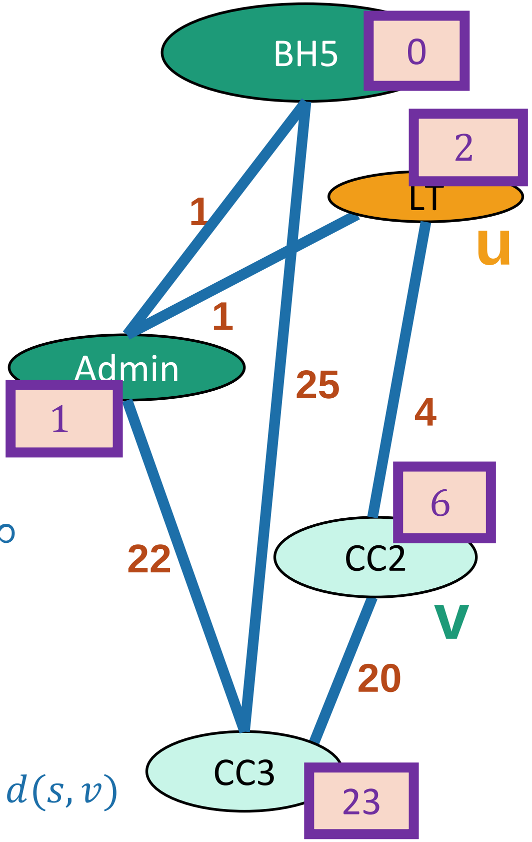
$d[v] = d[u] + w(u,v)$
$\geq d(s,u) + w(u,v) \geq d(s,v)$
using induction again for d[u]



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, $\mathrm{d}(\mathrm{s},\mathrm{s}) = 0$, and $d(s,v) \leq \infty$
- Inductive step: say hypothesis holds for t.
  - At step t+1:
    - Pick **u**; for each neighbor **v:**
    - d[v] ← min(d[v] , d[u] + w(u,v)) $\geq d(s,v)$



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

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

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

Claim 2

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

Claim 2

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

- Inductive Hypothesis:
  - When we mark the t^{th} vertex v as sure, d[v] = d(s,v).

Claim 2

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

- Inductive Hypothesis:
  - When we mark the t^{th} vertex v as sure, d[v] = d(s,v).
- Base case:
  - The first vertex marked **sure** is s, and d[s] = d(s,s) = 0.

Claim 2

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

- Inductive Hypothesis:
  - When we mark the t^{th} vertex v as sure, d[v] = d(s,v).
- Base case:
  - The first vertex marked **sure** is s, and d[s] = d(s,s) = 0.
- Inductive step:
  - Suppose that we are about to add u to the **sure** list.
  - That is, we picked u in the first line here:

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

Claim 2

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

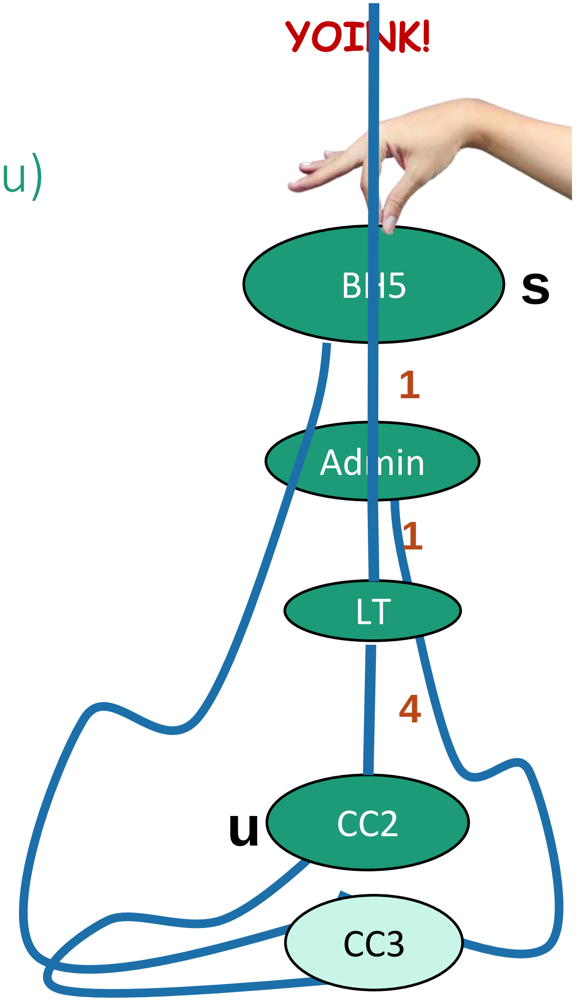
- Inductive Hypothesis:
  - When we mark the t^{th} vertex v as sure, d[v] = d(s,v).
- Base case:
  - The first vertex marked **sure** is s, and d[s] = d(s,s) = 0.
- Inductive step:
  - Suppose that we are about to add u to the **sure** list.
  - That is, we picked u in the first line here:

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

  - Assume by induction that every v already marked **sure** has d[v] = d(s,v).
  - Want to show that d[u] = d(s,u).

Intuition

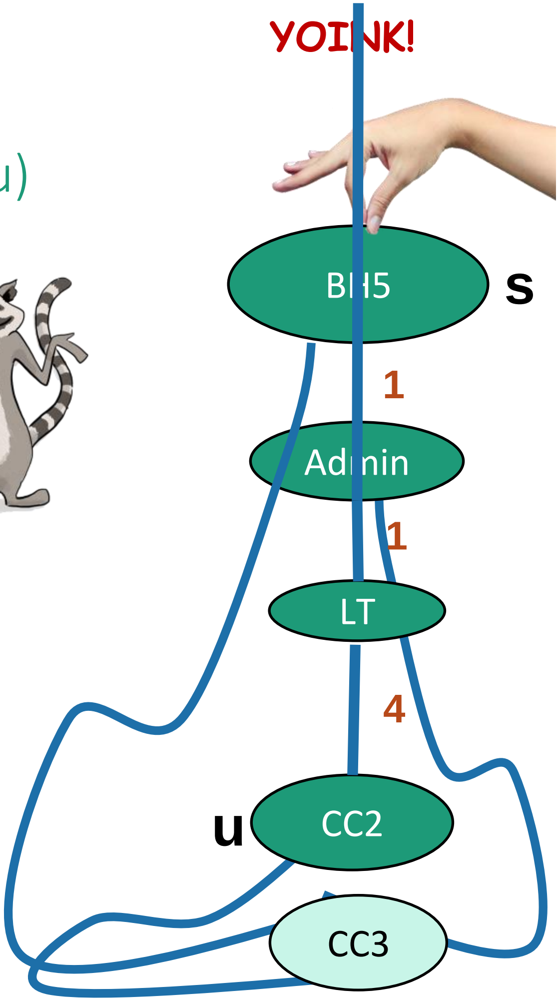
When a vertex u is marked sure, d[u] = d(s,u)



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.

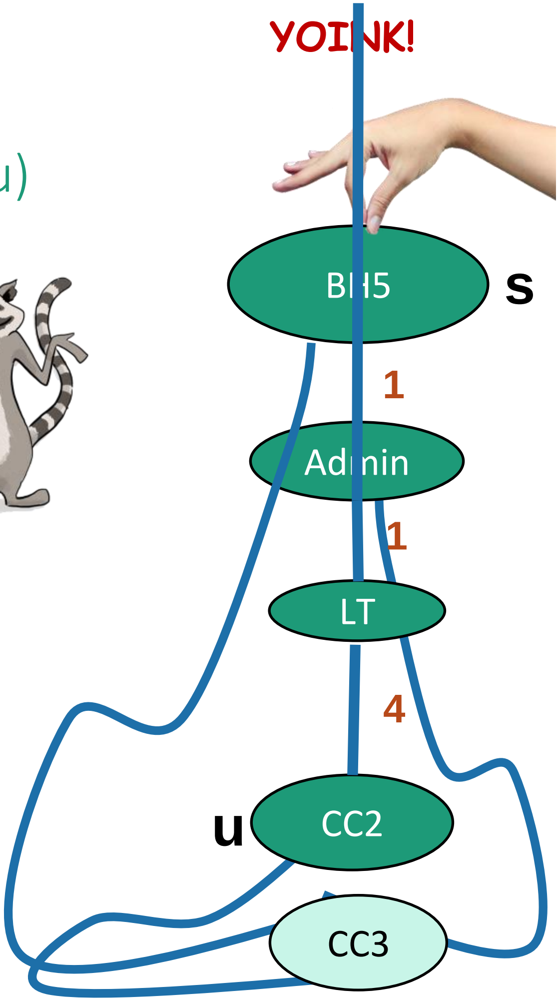


Intuition

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 we should actually prove it.

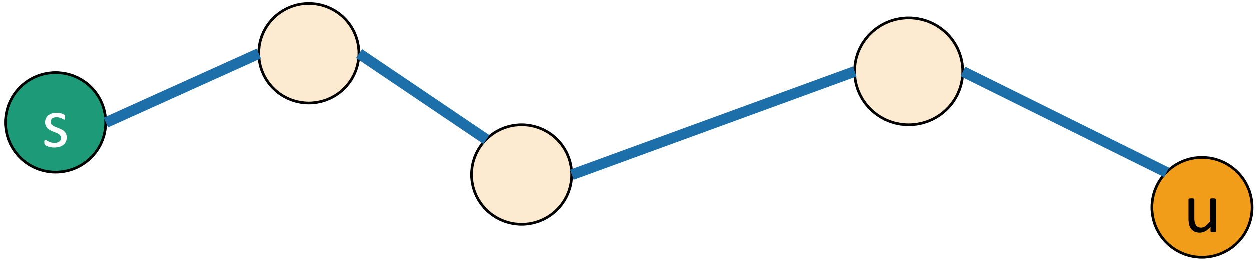


Claim 2

Inductive step

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

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



The vertices in between may or may not be **sure.**

True shortest path.

Claim 2

Inductive step

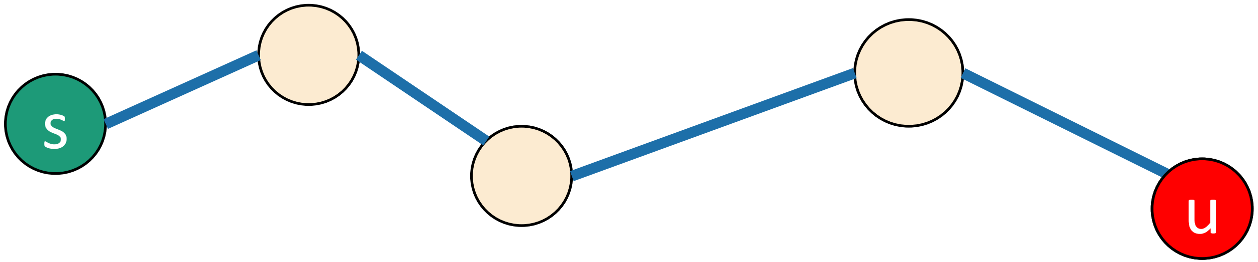
Temporary definition:

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

means good means not good

"by way of contradiction"

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



The vertices in between may or may not be **sure.**

True shortest path.

Claim 2

Inductive step

Temporary definition:

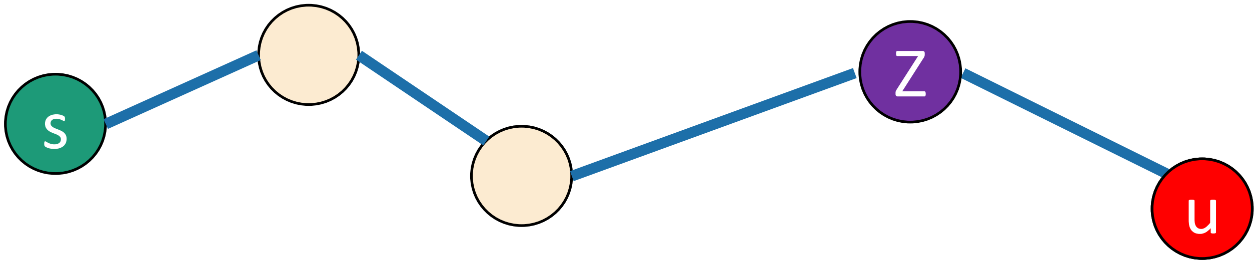
v is "good" means that d[v] = d(s,v)

means good

means not good

"by way of contradiction"

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



The vertices in between may or may not be **sure.**

True shortest path.

Claim 2

Inductive step

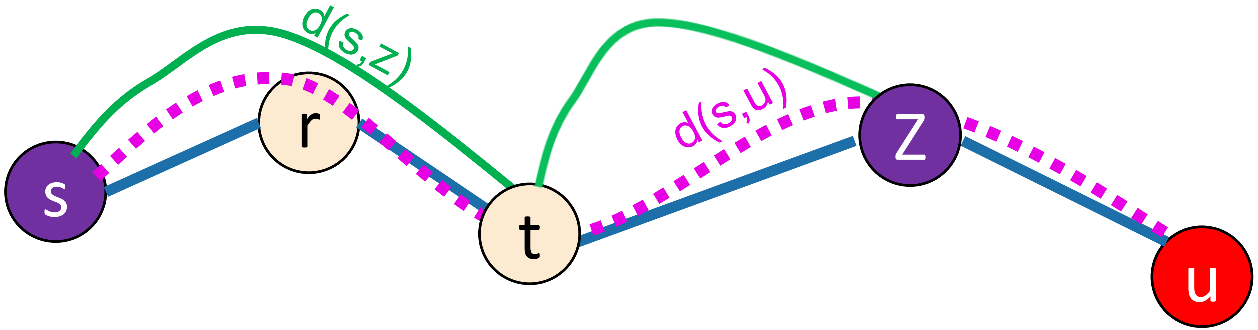
Temporary definition:
v is “good” means that d[v] = d(s,v)

(purple circle) means good (red circle) means not good

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

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

z is good | Subpaths of shortest paths are shortest paths. | Claim 1



Claim 2

Inductive step

Temporary definition:

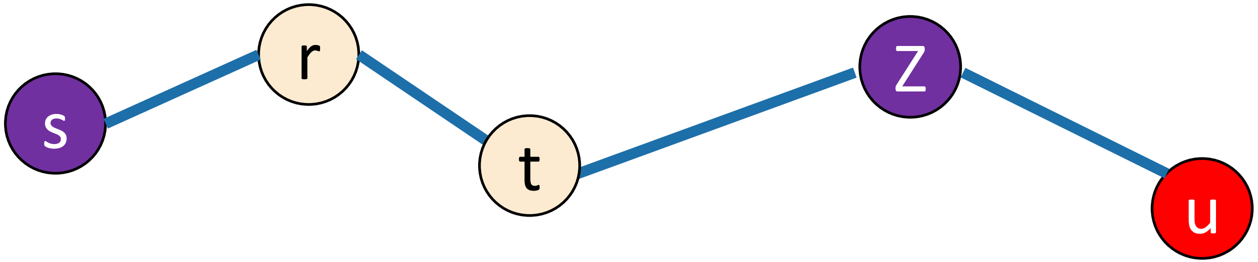
v is “good” means that d[v] = d(s,v)

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

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

z is good | Subpaths of shortest paths are shortest paths. | Claim 1

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



Claim 2

Inductive step

Temporary definition:

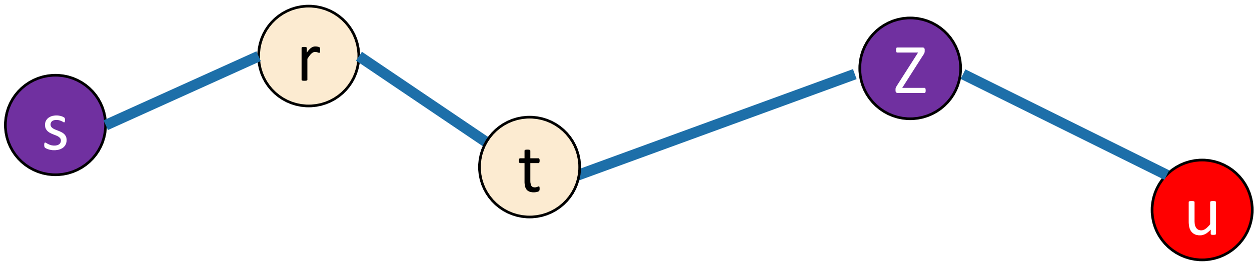
v is "good" means that d[v] = d(s,v)

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

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

z is good — Subpaths of shortest paths are shortest paths. — Claim 1

- If $d[z] = d[u]$, then u is good. But u is not good!



Claim 2

Inductive step

Temporary definition:

v is “good” means that d[v] = d(s,v)

(purple) means good (red) means not good

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

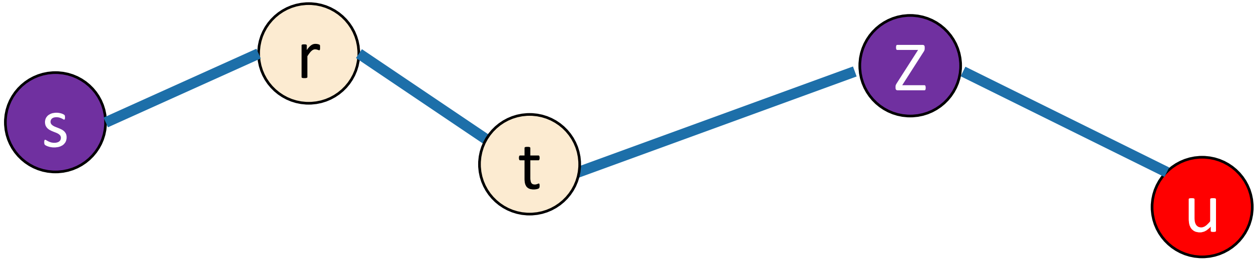
$$d[z] = d(s,z) \leq d(s,u) \leq d[u]$$

z is good — Subpaths of shortest paths are shortest paths. — Claim 1

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

But u is not good!

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



Claim 2

Inductive step

Temporary definition:

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

(purple) means good (red) means not good

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

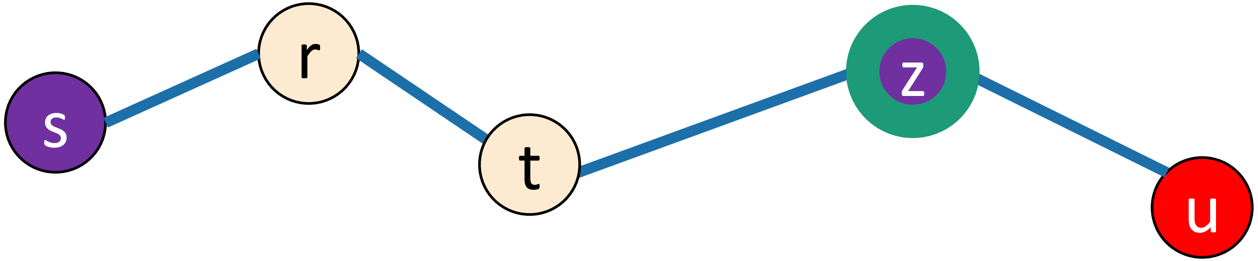
$$d[z] = d(s,z) \leq d(s,u) \leq d[u]$$

z is good — Subpaths of shortest paths are shortest paths. — Claim 1

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

But u is not good!

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



Claim 2

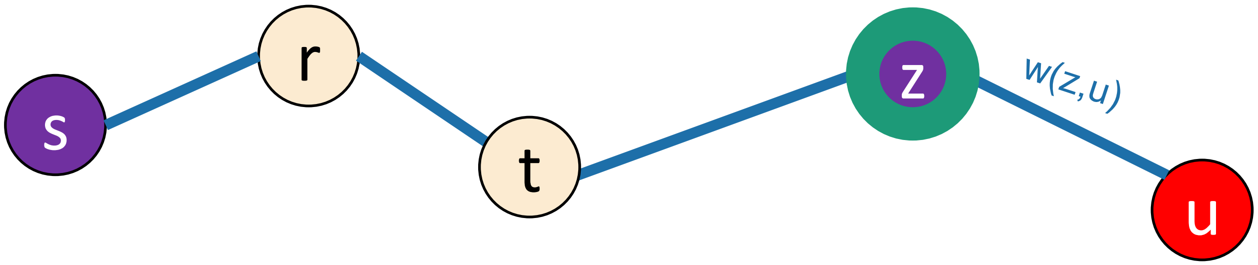
Inductive step

Temporary definition:

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

- Want to show that u is good. BWOC, suppose u isn't good.
- If z is sure then we've already updated u:

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



Claim 2

Inductive step

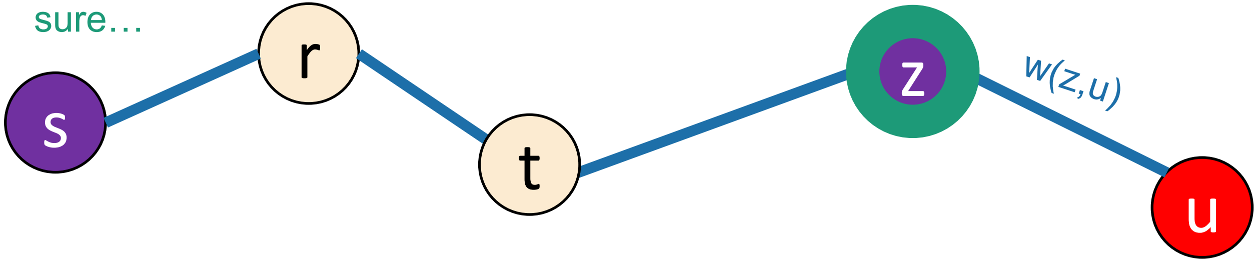
Temporary definition:
v is "good" means that d[v] = d(s,v)

means good

means not good

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

That is, the value of d[z] when z was marked sure…



Claim 2

Inductive step

Temporary definition:

v is “good” means that d[v] = d(s,v)

means good

means not good

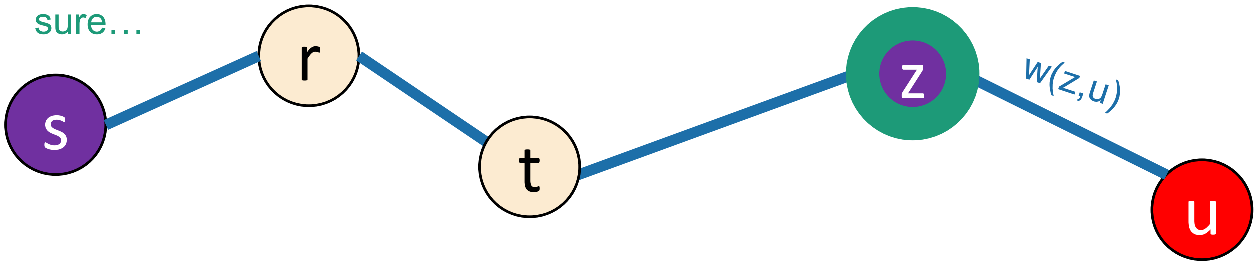
- Want to show that u is good. BWOC, suppose u isn’t good.
- If z is sure then we’ve already updated u:

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

- $d[u] \leq d[z] + w(z,u)$ def of update

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

That is, the value of d[z] when z was marked sure…



Claim 2

Inductive step

Temporary definition:

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

- Want to show that u is good. BWOC, suppose u isn't good.
- If z is sure then we've already updated u:

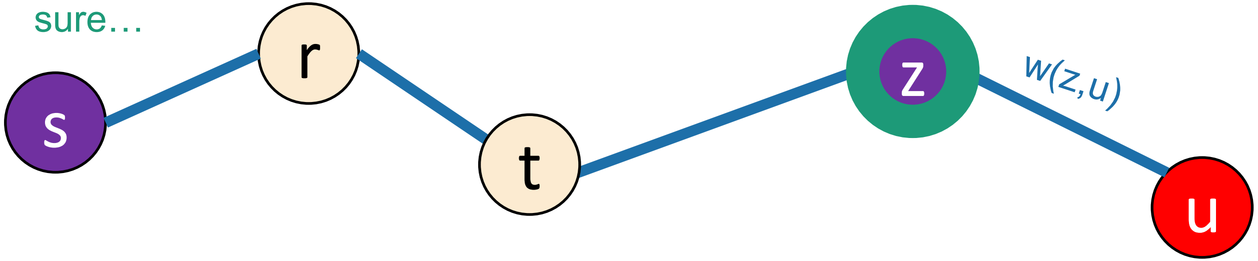
$d[u] \leftarrow min\{\, d[u], d[z] + w(z,u)\}$

- $d[u] \le d[z] + w(z,u)$ def of update

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

$= d(s,u)$ sub-paths of shortest paths are shortest paths

That is, the value of d[z] when z was marked sure…



Claim 2

Inductive step

Temporary definition:

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

- Want to show that u is good. BWOC, suppose u isn't good.
- If z is sure then we've already updated u:

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

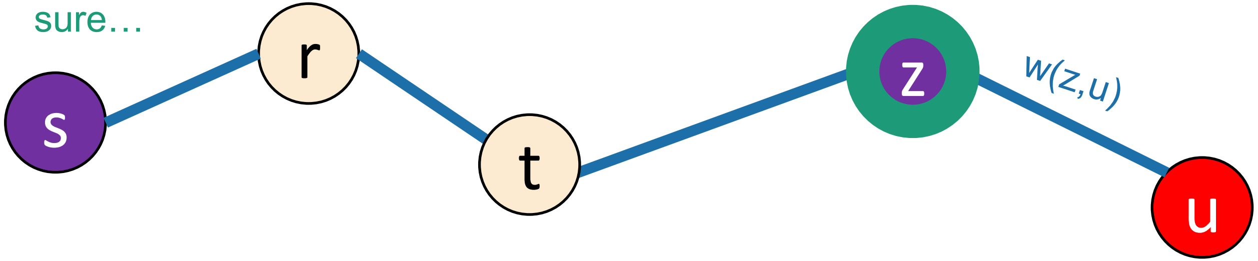
- $d[u] \leq d[z] + w(z,u)$ def of update

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

$= \; d(s,u)$ sub-paths of shortest paths are shortest paths

$\leq d[u]$ Claim 1

That is, the value of d[z] when z was marked sure…



Claim 2

Inductive step

Temporary definition:

v is “good” means that d[v] = d(s,v)

(purple circle) means good (red circle) 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 u:

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

- $d[u] \leq d[z] + w(z,u)$ def of update

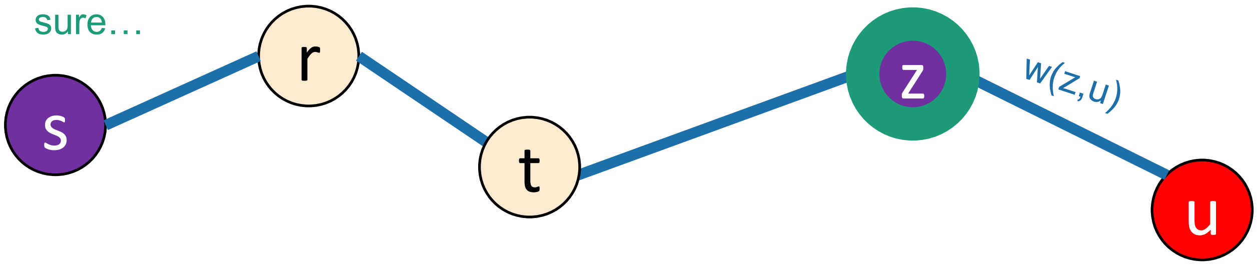
$= \; d(s,z) + w(z,u)$ By induction when z was added to the sure list it had d(s,z) = d[z]

$= \; d(s,u)$ sub-paths of shortest paths are shortest paths

$\leq d[u]$ Claim 1

So d(s, u) = d[u] and so u is good.

That is, the value of d[z] when z was marked sure…



Claim 2

Inductive step

Temporary definition:

v is “good” means that d[v] = d(s,v)

(purple circle) means good (red circle) 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 u:

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

- $d[u] \leq d[z] + w(z,u)$ def of update

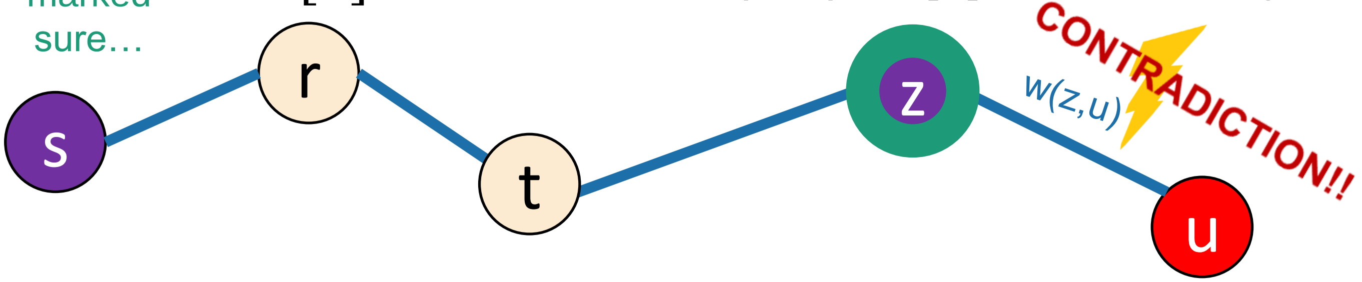
$= \; d(s,z) + w(z,u)$ By induction when z was added to the sure list it had d(s,z) = d[z]

$= \; d(s,u)$ sub-paths of shortest paths are shortest paths

$\leq d[u]$ Claim 1

That is, the value of d[z] when z was marked sure…

So d(s, u) = d[u] and so u is good.



Claim 2

Inductive step

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

means good   means not good

- Want to show that u is good. BWOC, suppose u isn't good.
- If z is sure then we've already updated u:

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

- $d[u] \leq d[z] + w(z,u)$ def of update

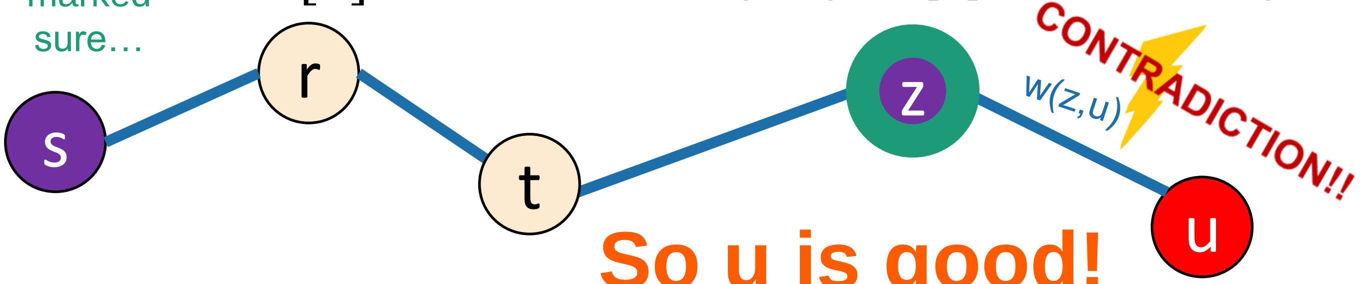
$= \ d(s,z) + w(z,u)$ By induction when z was added to the sure list it had d(s,z) = d[z]

That is, the value of d[z] when z was marked sure…

$= \ d(s,u)$ sub-paths of shortest paths are shortest paths

$\leq d[u]$ Claim 1

So d(s, u) = d[u] and so u is good.



So u is good!

Back to this slide

Claim 2

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

- Inductive Hypothesis:
  - When we mark the t^{th} vertex v as sure, d[v] = d(s,v).
- Base case:
  - The first vertex marked **sure** is s, and d[s] = d(s,s) = 0.
- Inductive step:
  - Suppose that we are about to add u to the **sure** list.
  - That is, we picked u in the first line here:

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

  - Assume by induction that every v already marked **sure** has d[v] = d(s,v).
  - Want to show that d[u] = d(s,u).

Conclusion: Claim 2 holds!

Why does this work?

Now back to this slide

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

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

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

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]

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:

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**.

We need a data structure that:

- Stores unsure vertices v

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**.

We need a data structure that:

- Stores unsure vertices v
- Keeps track of d[v]

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**.

We need a data structure that:

- Stores unsure vertices v
- Keeps track of d[v]
- Can find u with minimum d[u]
  - `findMin()`

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**.

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)`

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**.

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**.

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:

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

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

If we use an array

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

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

If we use a red-black tree

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

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

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

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

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

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

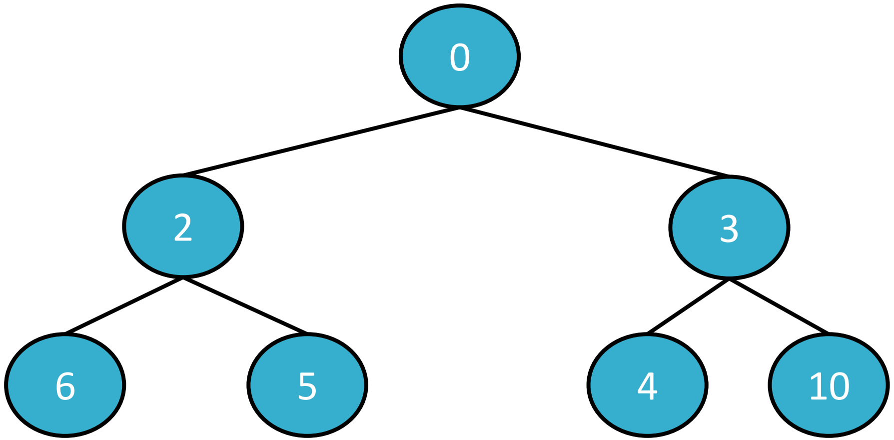
$$= O(n(T(\texttt{findMin}) + T(\texttt{removeMin})) + m\, T(\texttt{updateKey}))$$
$$= O(n\log(n)) + O(m\log(n))$$
$$= O((n + m)\log(n))$$

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.

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**

Say we use a **Fibonacci Heap**

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

Say we use a **Fibonacci Heap**

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

- Running time of Dijkstra

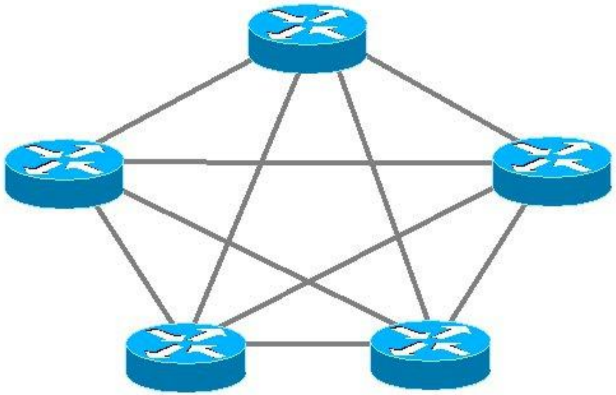
  =O(n(T(`findMin`) + T(`removeMin`)) + m T(`updateKey`))

  =O(nlog(n) + m)

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.

Summary

- 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).

Acknowledgement

- Stanford University

Thank You