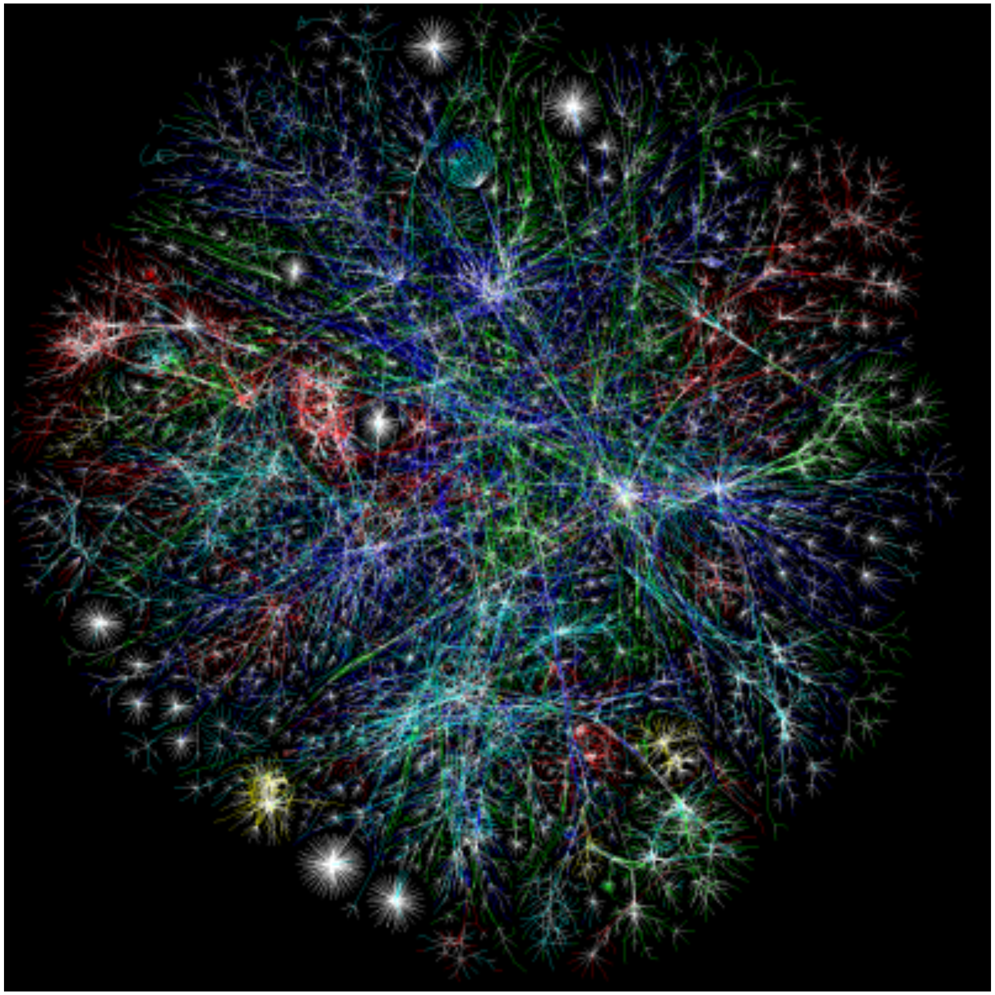
CS 106B

Lecture 27: A* Heuristics and Minimum Spanning Trees

Friday, December 2, 2016



Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, pp. 820-821

Today's Topics

- Logistics
  - We updated the Trailblazer alternate route description on the handout.

- Typedef definition
  - Real Graph: Internet routers and traceroute
- More on Trailblazer
- A* Heuristics
  - heuristic bounds, and why we want to underestimate
- Minimum Spanning Trees
  - Kruskal's algorithm
- Dijkstra and negative weights (extra slides)

C++: typedef

In Assignment 7, you will use a **`Vector<Vertex *>`** type for all your paths. Because paths are used so often in the assignment, we have defined a type called **`Path`**, which is simply a **`Vector<Vertex *>`**. To define a new type in C++, you can use the **`typedef`** keyword, as follows:

```
typedef Vector<Vertex *> Path;
```

Now, you can use Path as any other variable, and it is just a **`Vector<Vertex *>`**:

```
Path p;
if (p.size() > 0) {
    Vertex *v = p[0];
}
for (Vertex *v : p) {
  cout << "Next vertex name: " << v->name << endl;
}
```

Real Graphs!

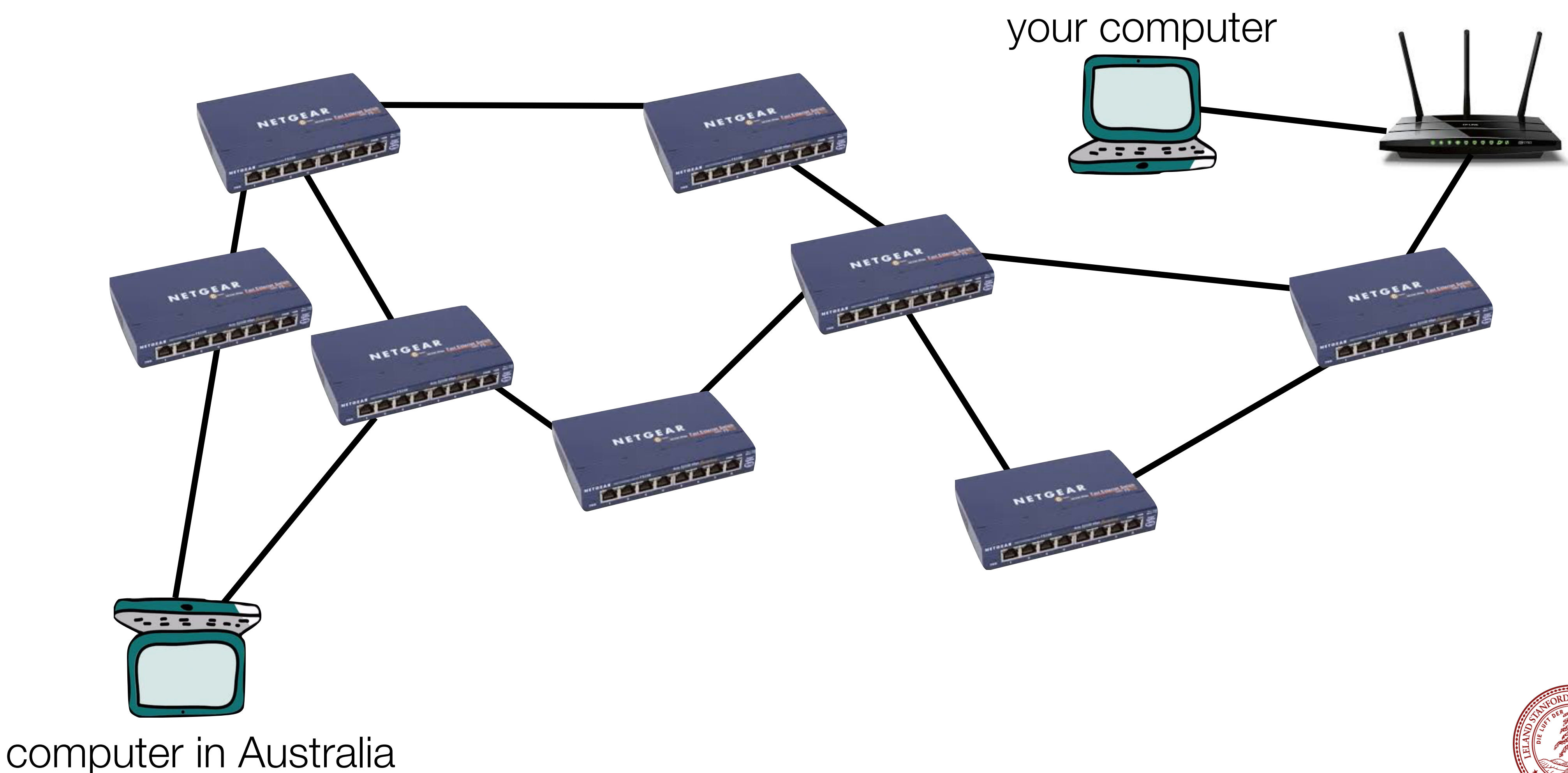
There was a Tiny Feedback from the last lecture that said,

> *❝Would love more stories about when you two use these different search algorithms more in real life.❞*

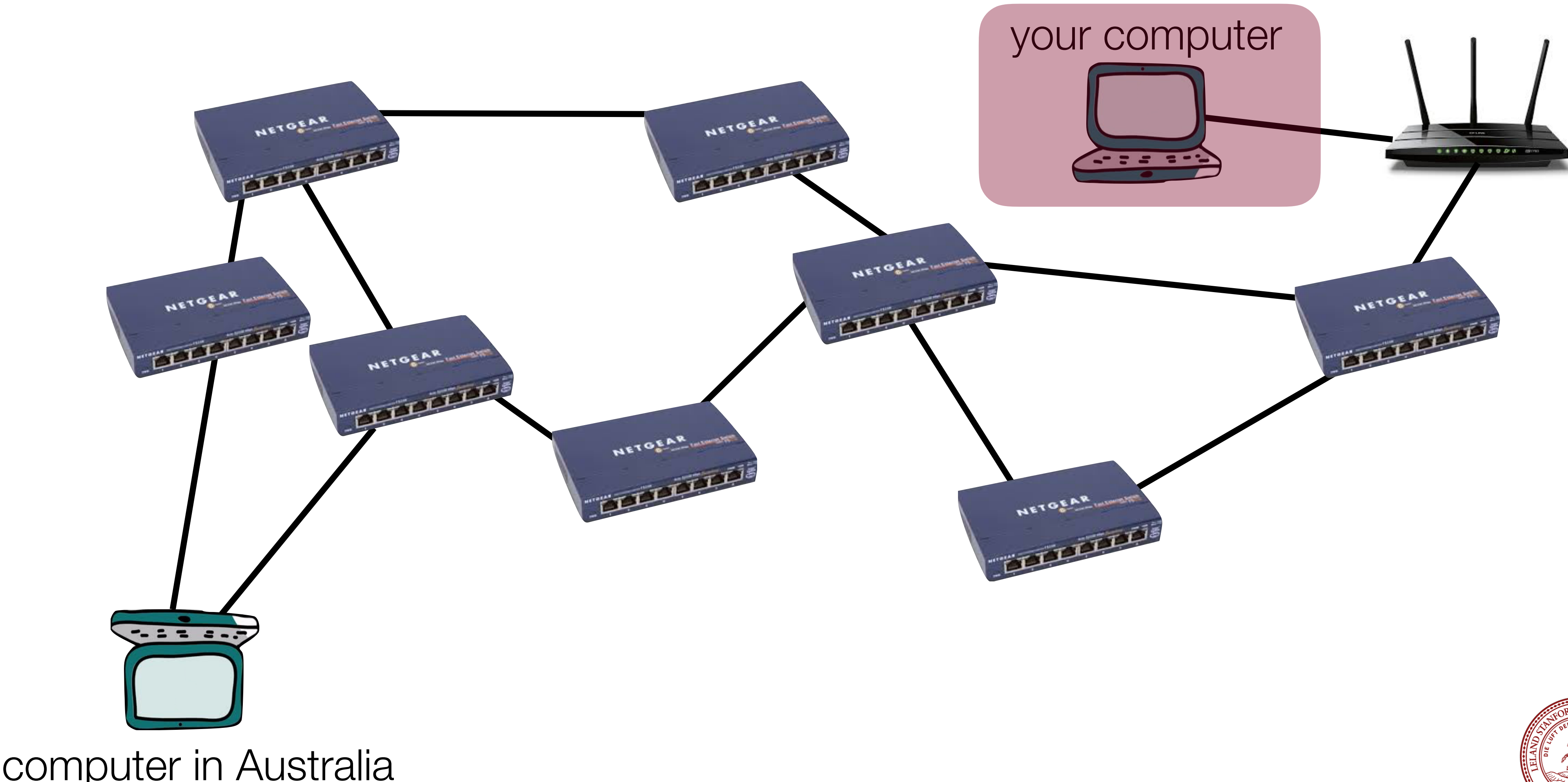
On Monday we played the "Wikipedia Get to Philosophy" game with the Internet, which have web pages with links that form a graph. Let's see another example of how the Internet is a real graph in a completely different way: Routers

How does a message get sent from your computer to another computer on the Internet, say in Australia?

The Internet: Computers connected through routers



The Internet: Computers connected through routers



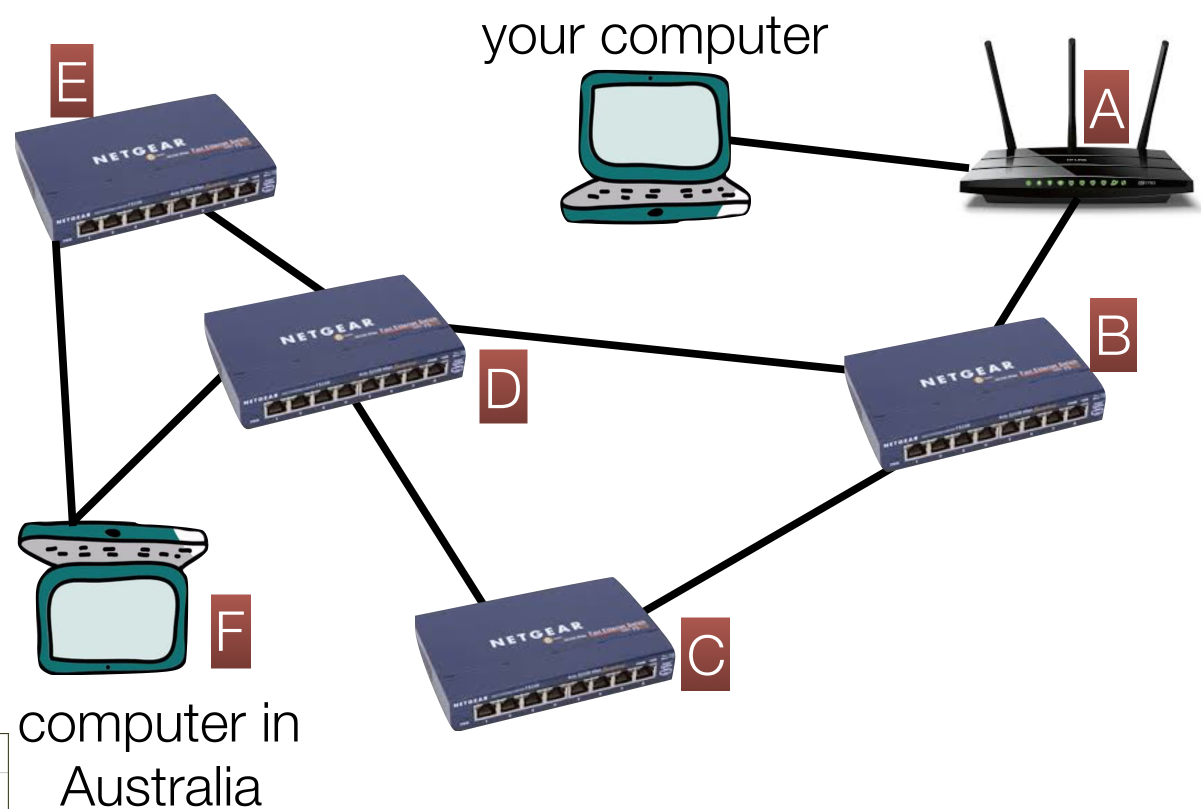
The Internet: Let's simplify a bit

The destination computer has a name and an IP address, like this:

`www.engineering.unsw.edu.au`
`IP address: 149.171.158.109`

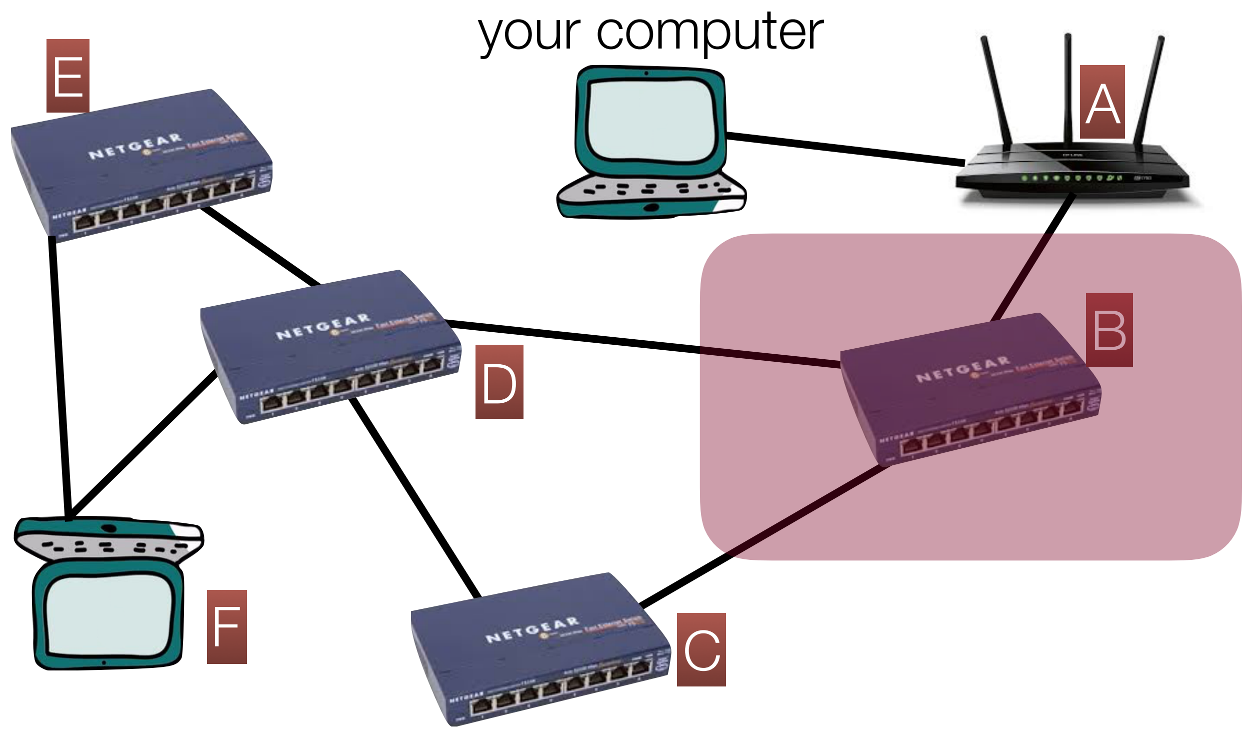
The first number denotes the "network address" and routers continually pass around information about how many "hops" they think it will take for them to get to all the networks. E.g., for router `C`:

router	hops
A	2
B	1
C	-
D	1
E	2
F	2



The Internet: Let's simplify a bit

Each router knows its neighbors, and it has a copy of its neighbors' tables. So, `B` would have the following tables:



A

router	hops
A	-
B	1
C	3
D	2
E	3
F	3

C

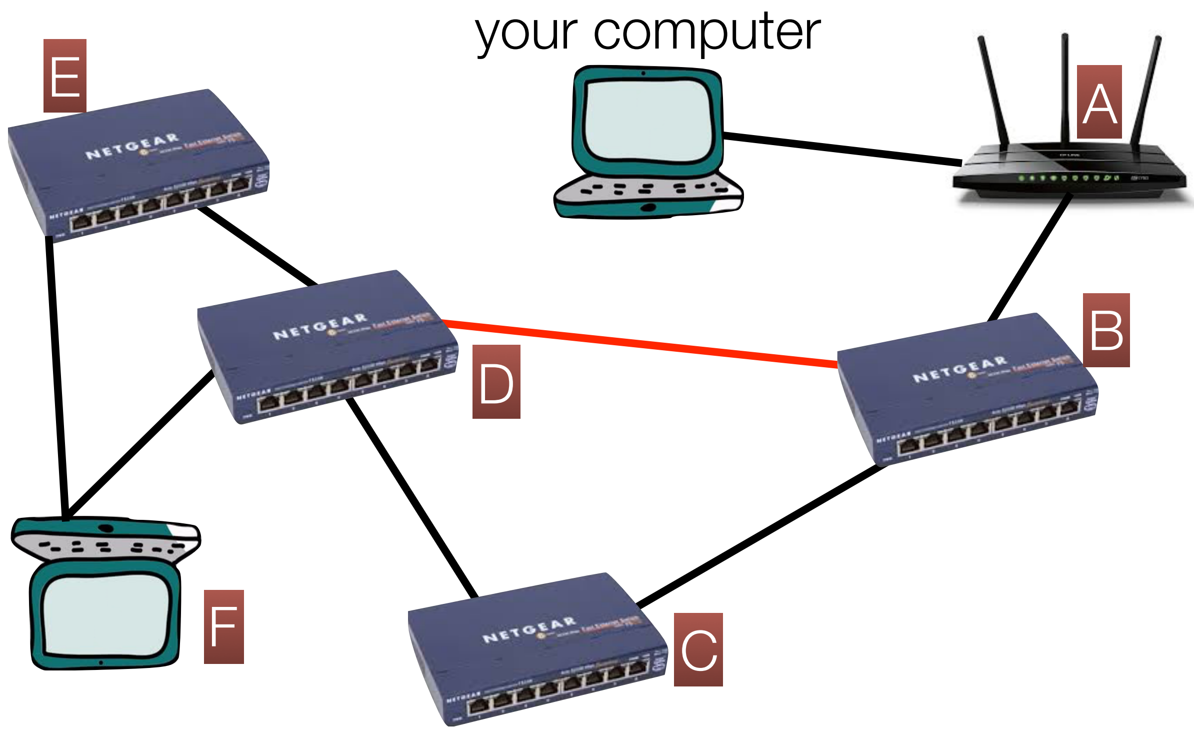
router	hops
A	2
B	1
C	-
D	1
E	2
F	2

D

router	hops
A	2
B	1
C	1
D	-
E	1
F	1

The Internet: Let's simplify a bit

If B wants to connect to F, it connects through its neighbor that reports the shortest path to F. Which router would it choose?



A

router	hops
A	-
B	1
C	3
D	2
E	3
F	3

C

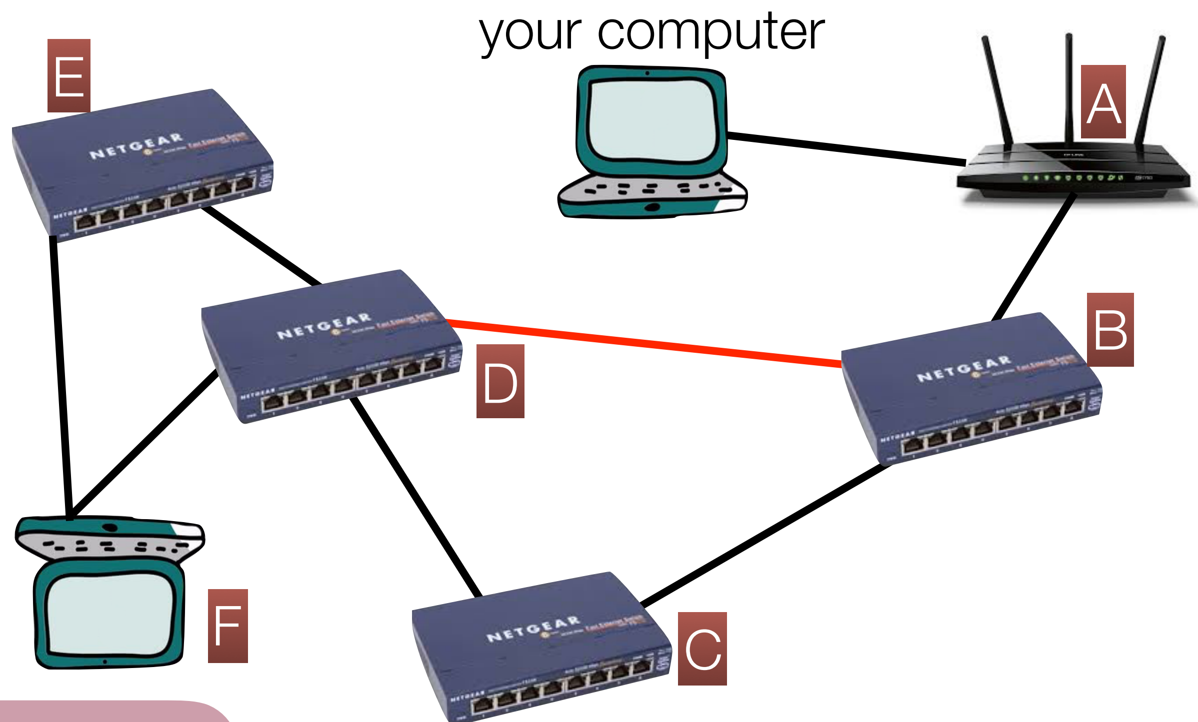
router	hops
A	2
B	1
C	-
D	1
E	2
F	2

D

router	hops
A	2
B	1
C	1
D	-
E	1
F	1

The Internet: Let's simplify a bit

If B wants to connect to F, it connects through its neighbor that reports the shortest path to F. Which router would it choose? D.



A

router	hops
A	-
B	1
C	3
D	2
E	3
F	3

C

router	hops
A	2
B	1
C	-
D	1
E	2
F	2

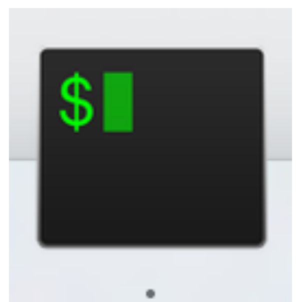
D

router	hops
A	2
B	1
C	1
D	-
E	1
F	1

Traceroute

We can use a program called "traceroute" to tell us the path between our computer and a different computer:

`traceroute -I -e www.engineering.unsw.edu.au`



Traceroute: Stanford Hops

```
traceroute -I -e www.engineering.unsw.edu.au
traceroute to www.engineering.unsw.edu.au (149.171.158.109), 64 hops max, 72 byte packets
 1  csmx-west-rtr.sunet (171.67.64.2)  7.414 ms  9.155 ms  8.288 ms
 2  gnat-2.sunet (172.24.70.12)  0.339 ms  1.532 ms  0.423 ms
 3  csmx-west-rtr-vl3866.sunet (171.64.66.2)  38.916 ms  10.506 ms  8.402 ms
 4  dca-rtr-vlan8.sunet (171.64.255.204)  0.530 ms  0.521 ms  0.713 ms
 5  dc-svl-agg4--stanford-10ge.cenic.net (137.164.50.157)  1.554 ms  1.653 ms  2.828 ms
 6  hpr-svl-hpr2--svl-agg4-10ge.cenic.net (137.164.26.249)  1.212 ms  1.161 ms  1.204 ms
 7  aarnet-2-is-jmb-778.sttlwa.pacificwave.net (207.231.245.4)  17.994 ms  17.998 ms  18.319 ms
 8  et-2-0-0.pe2.brwy.nsw.aarnet.net.au (113.197.15.98)  160.020 ms  160.234 ms  159.922 ms
 9  et-3-3-0.pe1.brwy.nsw.aarnet.net.au (113.197.15.148)  160.285 ms  160.076 ms  160.118 ms
10  138.44.5.1 (138.44.5.1)  160.124 ms  160.138 ms  160.068 ms
11  ombcr1-te-1-5.gw.unsw.edu.au (149.171.255.106)  160.090 ms  160.381 ms  160.185 ms
12  r1dcdnex1-po-2.gw.unsw.edu.au (149.171.255.178)  160.909 ms  160.847 ms  160.921 ms
13  dcfw1-ae-1-3049.gw.unsw.edu.au (129.94.254.60)  160.592 ms  160.558 ms  160.949 ms
14  www.engineering.unsw.edu.au (149.171.158.109)  160.978 ms  161.184 ms  160.987 ms
```

Traceroute: CENIC

```
traceroute -I -e www.engineering.unsw.edu.au
traceroute to www.engineering.unsw.edu.au (149.171.158.109), 64 hops max, 72 byte packets
 1  csmx-west-rtr.sunet (171.67.64.2)  7.414 ms  9.155 ms  8.288 ms
 2  gnat-2.sunet (172.24.70.12)  0.339 ms  1.532 ms  0.423 ms
 3  csmx-west-rtr-vl3866.sunet (171.64.66.2)  38.916 ms  10.506 ms  8.402 ms
 4  dca-rtr-vlan8.sunet (171.64.255.204)  0.530 ms  0.521 ms  0.713 ms
 5  dc-svl-agg4--stanford-10ge.cenic.net (137.164.50.157)  1.554 ms  1.653 ms  2.828 ms
 6  hpr-svl-hpr2--svl-agg4-10ge.cenic.net (137.164.26.249)  1.212 ms  1.161 ms  1.204 ms
 7  aarnet-2-is-jmb-778.sttlwa.pacificwave.net (207.231.245.4)  17.994 ms  17.998 ms  18.319 ms
 8  et-2-0-0.pe2.brwy.nsw.aarnet.net.au (113.197.15.98)  160.020 ms  160.234 ms  159.922 ms
 9  et-3-3-0.pe1.brwy.nsw.aarnet.net.au (113.197.15.148)  160.285 ms  160.076 ms  160.118 ms
10  138.44.5.1 (138.44.5.1)  160.124 ms  160.138 ms  160.068 ms
11  ombcr1-te-1-5.gw.unsw.edu.au (149.171.255.106)  160.090 ms  160.381 ms  160.185 ms
12  r1dcdnex1-po-2.gw.unsw.edu.au (149.171.255.178)  160.909 ms  160.847 ms  160.921 ms
13  dcfw1-ae-1-3049.gw.unsw.edu.au (129.94.254.60)  160.592 ms  160.558 ms  160.949 ms
14  www.engineering.unsw.edu.au (149.171.158.109)  160.978 ms  161.184 ms  160.987 ms
```

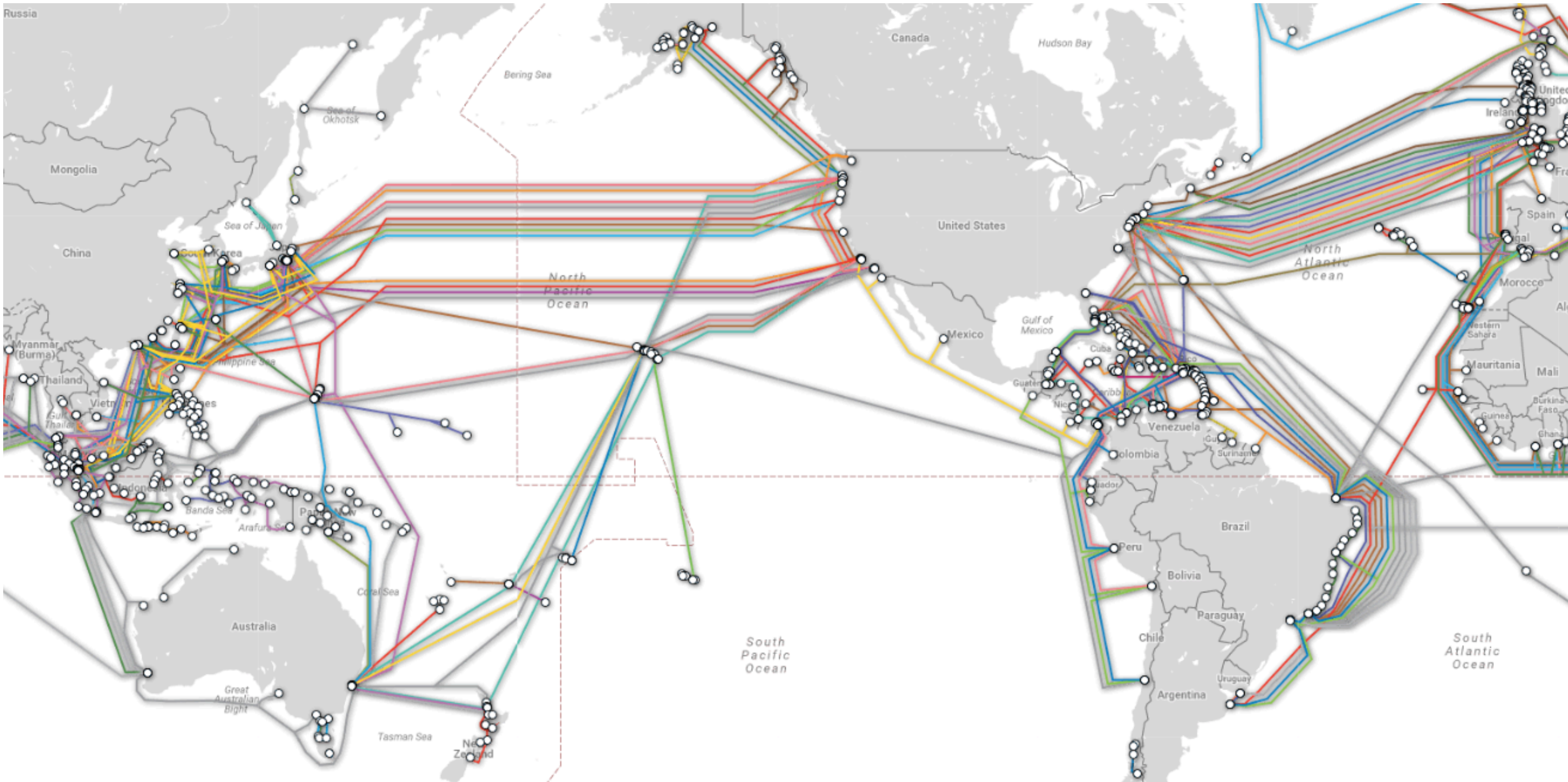
The **Corporation for Education Network Initiatives in California** (**CENIC**) is a nonprofit corporation formed in 1996 to provide high-performance, high-bandwidth networking services to California universities and research institutions (source: Wikipedia)

Traceroute: Pacificwave (Seattle)

```
traceroute -I -e www.engineering.unsw.edu.au
traceroute to www.engineering.unsw.edu.au (149.171.158.109), 64 hops max, 72 byte packets
 1  csmx-west-rtr.sunet (171.67.64.2)  7.414 ms  9.155 ms  8.288 ms
 2  gnat-2.sunet (172.24.70.12)  0.339 ms  1.532 ms  0.423 ms
 3  csmx-west-rtr-vl3866.sunet (171.64.66.2)  38.916 ms  10.506 ms  8.402 ms
 4  dca-rtr-vlan8.sunet (171.64.255.204)  0.530 ms  0.521 ms  0.713 ms
 5  dc-svl-agg4--stanford-10ge.cenic.net (137.164.50.157)  1.554 ms  1.653 ms  2.828 ms
 6  hpr-svl-hpr2--svl-agg4-10ge.cenic.net (137.164.26.249)  1.212 ms  1.161 ms  1.204 ms
 7  aarnet-2-is-jmb-778.sttlwa.pacificwave.net (207.231.245.4)  17.994 ms  17.998 ms  18.319 ms
 8  et-2-0-0.pe2.brwy.nsw.aarnet.net.au (113.197.15.98)  160.020 ms  160.234 ms  159.922 ms
 9  et-3-3-0.pe1.brwy.nsw.aarnet.net.au (113.197.15.148)  160.285 ms  160.076 ms  160.118 ms
10  138.44.5.1 (138.44.5.1)  160.124 ms  160.138 ms  160.068 ms
                   gw.unsw.edu.au (149.171.255.106)  160.090 ms  160.381 ms  160.185 ms
                   .gw.unsw.edu.au (149.171.255.178)  160.909 ms  160.847 ms  160.921 ms
                   9.gw.unsw.edu.au (129.94.254.60)  160.592 ms  160.558 ms  160.949 ms
                   g.unsw.edu.au (149.171.158.109)  160.978 ms  161.184 ms  160.987 ms
```

Pass Internet traffic directly with other major national and international networks, including U.S. federal agencies and many Pacific Rim R&E networks (source: http://www.pnwgp.net/services/pacific-wave-peering-exchange/)

Traceroute: Oregon to Australia - underwater!



http://www.submarinecablemap.com

Traceroute: Australia

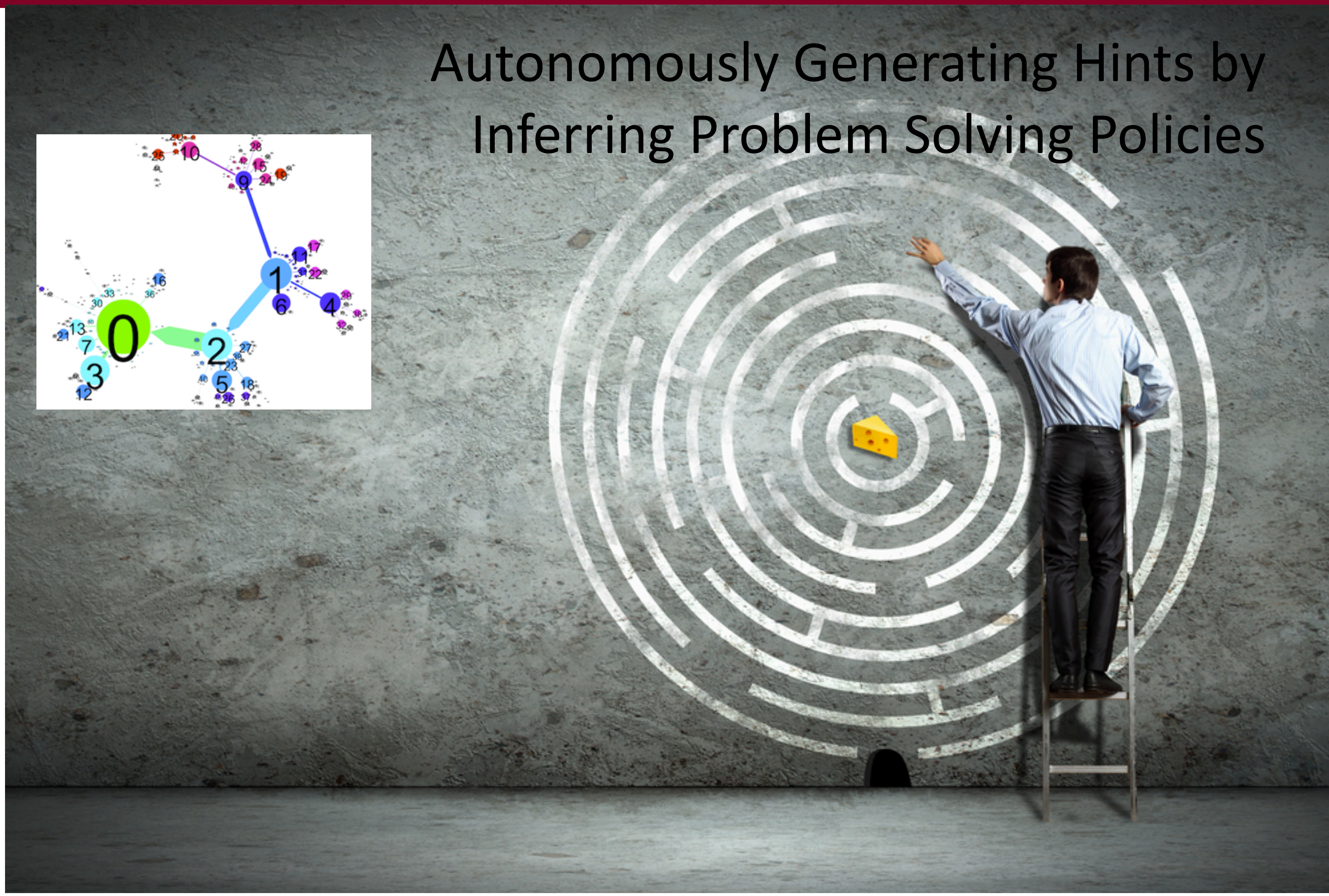
```
traceroute -I -e www.engineering.unsw.edu.au
traceroute to www.engineering.unsw.edu.au (149.171.158.109), 64 hops max, 72 byte packets
 1  csmx-west-rtr.sunet (171.67.64.2)  7.414 ms  9.155 ms  8.288 ms
 2  gnat-2.sunet (172.24.70.12)  0.339 ms  1.532 ms  0.423 ms
 3  csmx-west-rtr-vl3866.sunet (171.64.66.2)  38.916 ms  10.506 ms  8.402 ms
 4  dca-rtr-vlan8.sunet (171.64.255.204)  0.530 ms  0.521 ms  0.713 ms
 5  dc-svl-agg4--stanford-10ge.cenic.net (137.164.50.157)  1.554 ms  1.653 ms  2.828 ms
 6  hpr-svl-hpr2--svl-agg4-10ge.cenic.net (137.164.26.249)  1.212 ms  1.161 ms  1.204 ms
 7  aarnet-2-is-jmb-778.sttlwa.pacificwave.net (207.231.245.4)  17.994 ms  17.998 ms  18.319 ms
 8  et-2-0-0.pe2.brwy.nsw.aarnet.net.au (113.197.15.98)  160.020 ms  160.234 ms  159.922 ms
 9  et-3-3-0.pe1.brwy.nsw.aarnet.net.au (113.197.15.148)  160.285 ms  160.076 ms  160.118 ms
10  138.44.5.1 (138.44.5.1)  160.124 ms  160.138 ms  160.068 ms
11  ombcr1-te-1-5.gw.unsw.edu.au (149.171.255.106)  160.090 ms  160.381 ms  160.185 ms
12  r1dcdnex1-po-2.gw.unsw.edu.au (149.171.255.178)  160.909 ms  160.847 ms  160.921 ms
13  dcfw1-ae-1-3049.gw.unsw.edu.au (129.94.254.60)  160.592 ms  160.558 ms  160.949 ms
14  www.engineering.unsw.edu.au (149.171.158.109)  160.978 ms  161.184 ms  160.987 ms
```

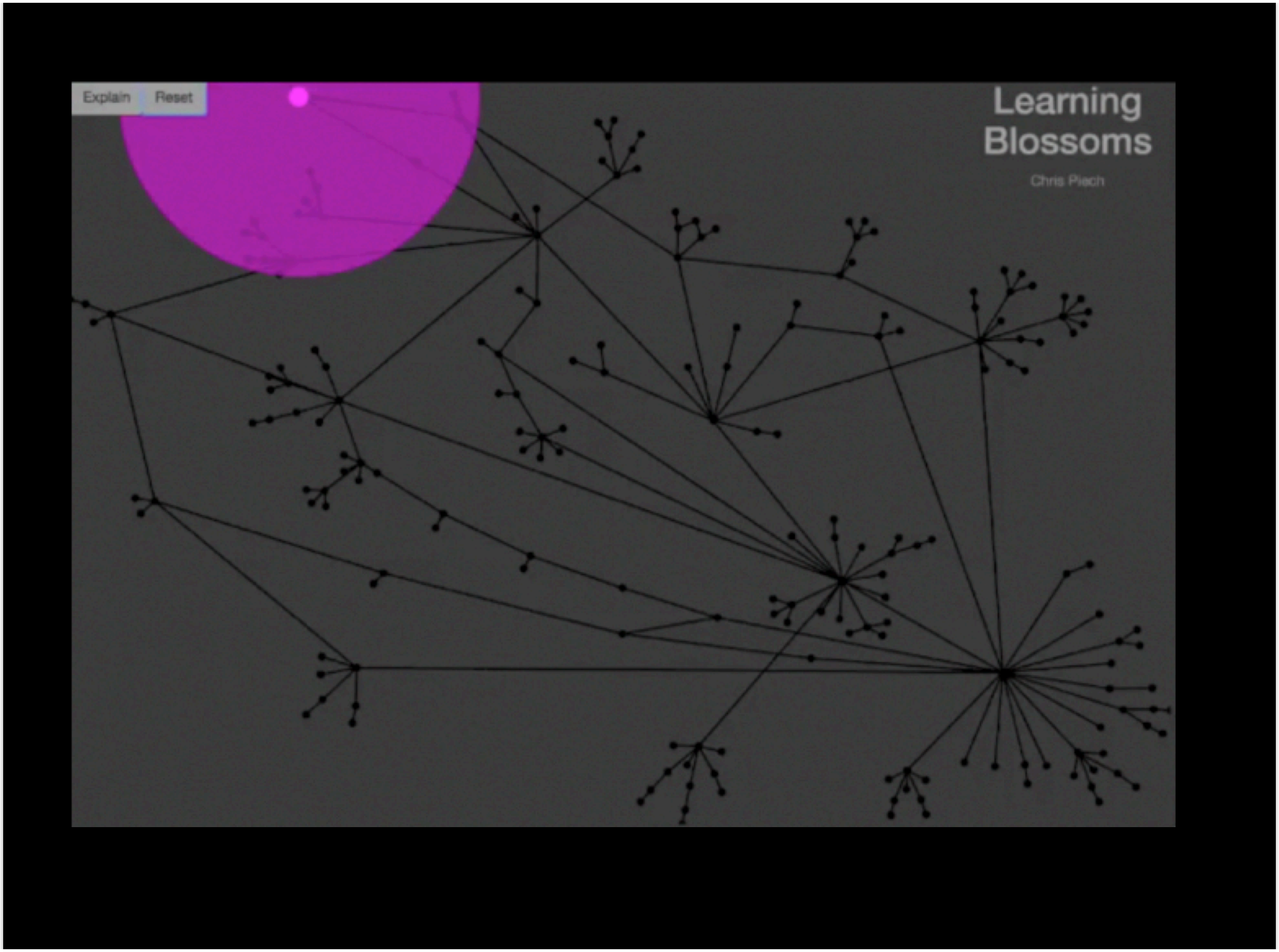
Traceroute: University of New South Wales

```
traceroute -I -e www.engineering.unsw.edu.au
traceroute to www.engineering.unsw.edu.au (149.171.158.109), 64 hops max, 72 byte packets
 1  csmx-west-rtr.sunet (171.67.64.2)  7.414 ms  9.155 ms  8.288 ms
 2  gnat-2.sunet (172.24.70.12)  0.339 ms  1.532 ms  0.423 ms
 3  csmx-west-rtr-vl3866.sunet (171.64.66.2)  38.916 ms  10.506 ms  8.402 ms
 4  dca-rtr-vlan8.sunet (171.64.255.204)  0.530 ms  0.521 ms  0.713 ms
 5  dc-svl-agg4--stanford-10ge.cenic.net (137.164.50.157)  1.554 ms  1.653 ms  2.828 ms
 6  hpr-svl-hpr2--svl-agg4-10ge.cenic.net (137.164.26.249)  1.212 ms  1.161 ms  1.204 ms
 7  aarnet-2-is-jmb-778.sttlwa.pacificwave.net (207.231.245.4)  17.994 ms  17.998 ms  18.319 ms
 8  et-2-0-0.pe2.brwy.nsw.aarnet.net.au (113.197.15.98)  160.020 ms  160.234 ms  159.922 ms
 9  et-3-3-0.pe1.brwy.nsw.aarnet.net.au (113.197.15.148)  160.285 ms  160.076 ms  160.118 ms
10  138.44.5.1 (138.44.5.1)  160.124 ms  160.138 ms  160.068 ms
11  ombcr1-te-1-5.gw.unsw.edu.au (149.171.255.106)  160.090 ms  160.381 ms  160.185 ms
12  r1dcdnex1-po-2.gw.unsw.edu.au (149.171.255.178)  160.909 ms  160.847 ms  160.921 ms
13  dcfw1-ae-1-3049.gw.unsw.edu.au (129.94.254.60)  160.592 ms  160.558 ms  160.949 ms
14  www.engineering.unsw.edu.au (149.171.158.109)  160.978 ms  161.184 ms  160.987 ms
```

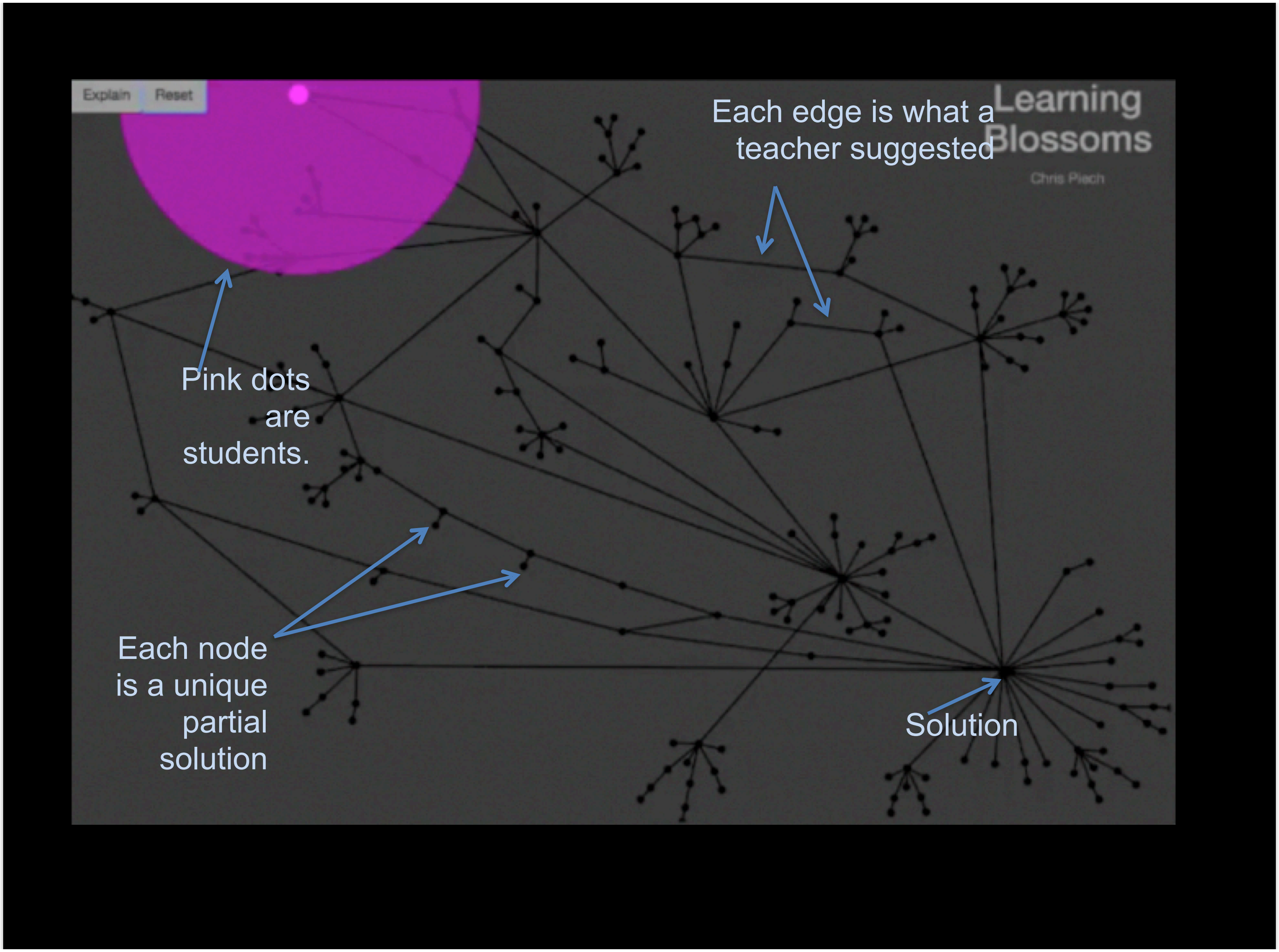
161 milliseconds to get to the final computer

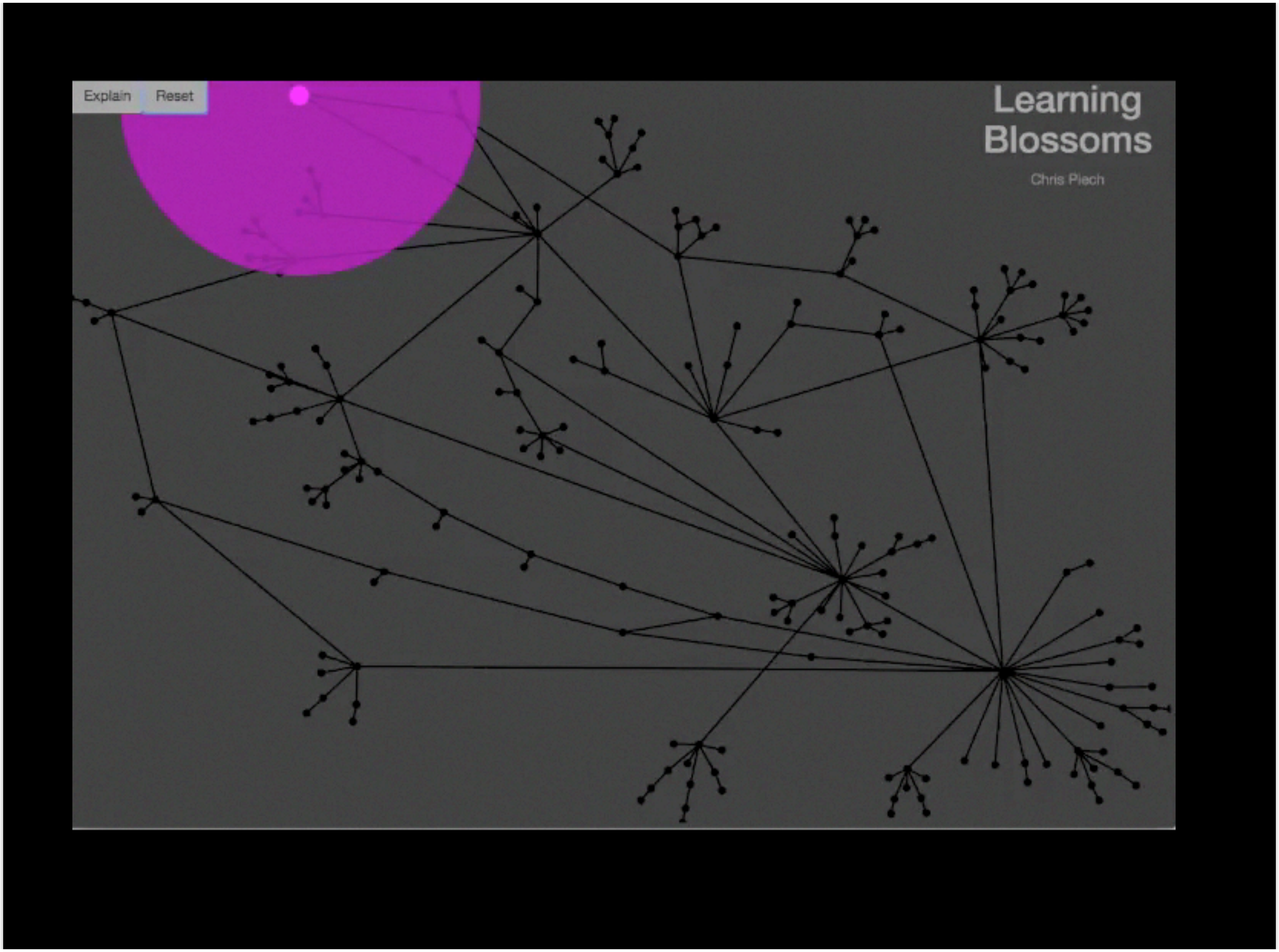
More Real Graphs: Chris Piech's Research!



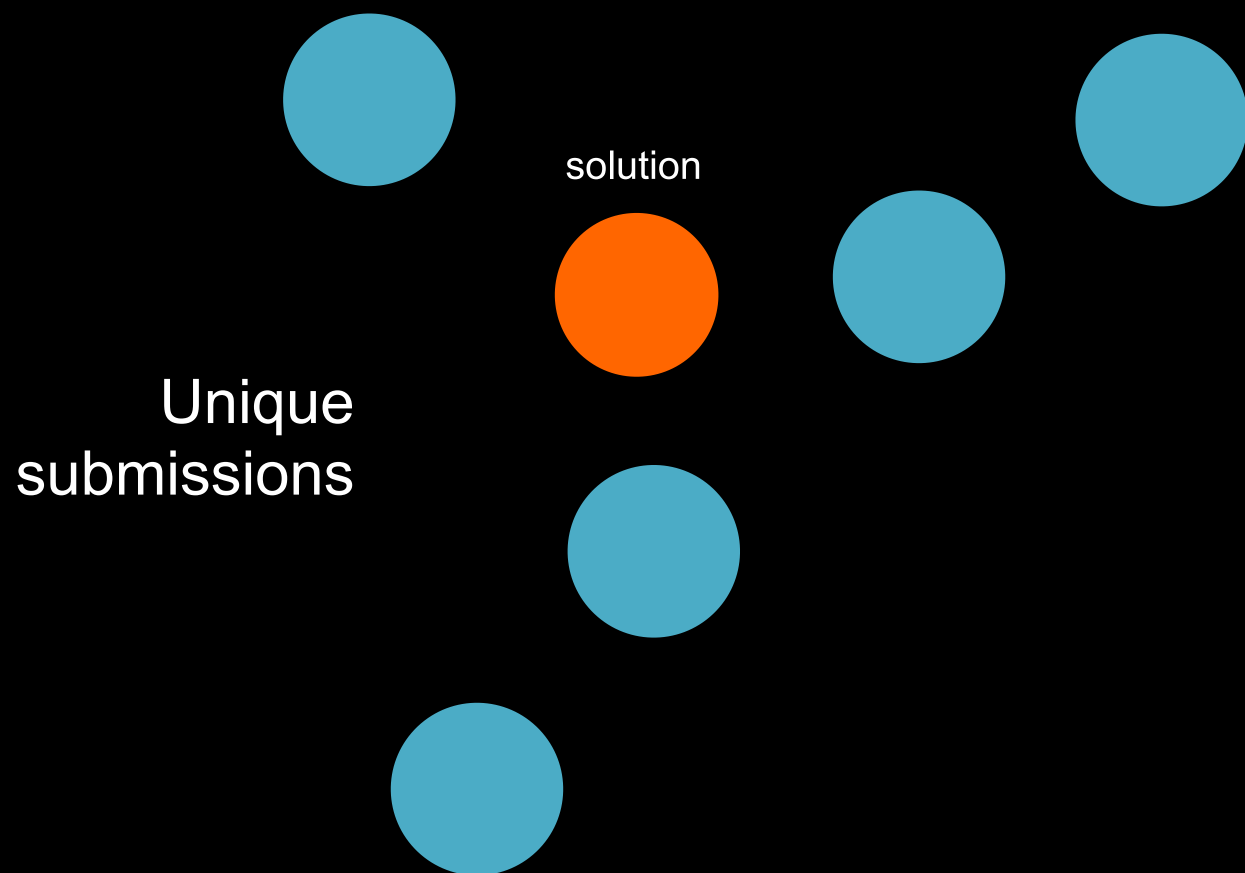


Explain
Reset
Learning Blossoms
Chris Piech

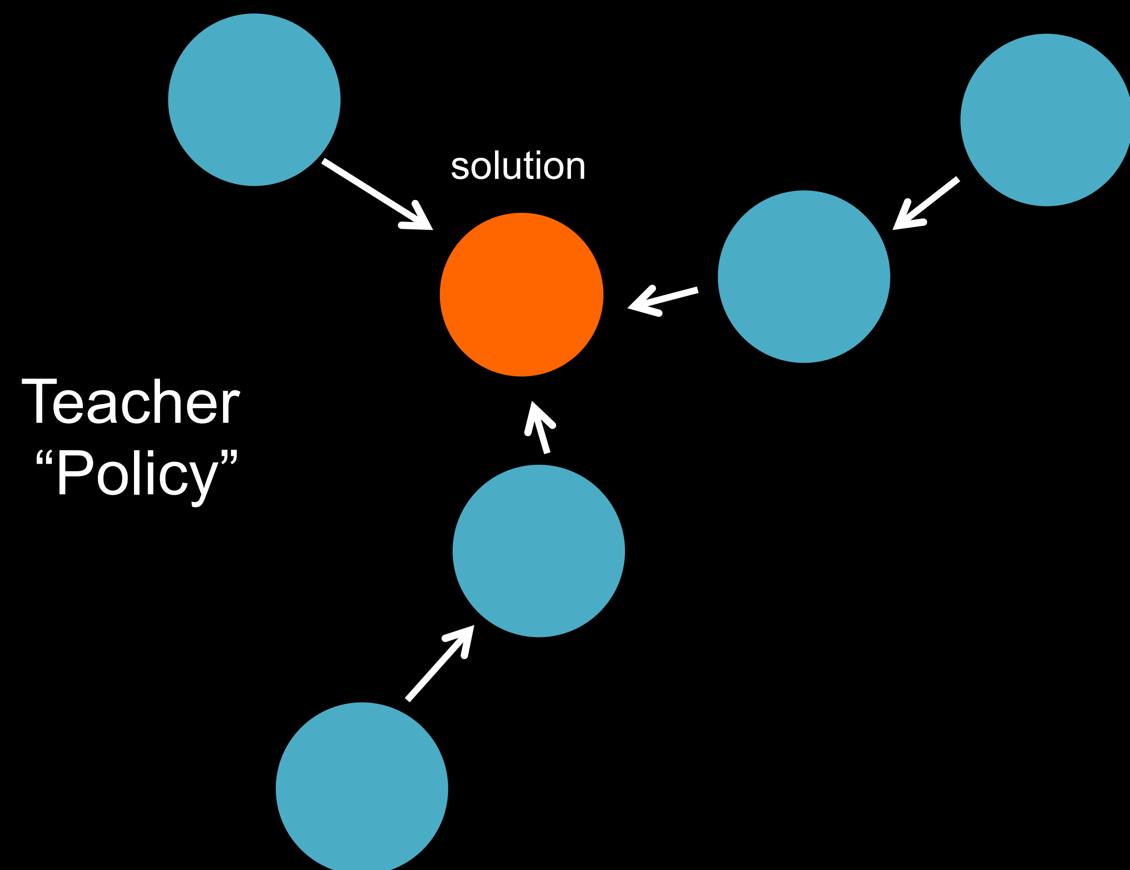

Explain
Reset
Learning Blossoms
Chris Piech
Each edge is what a teacher suggested
Pink dots are students.
Each node is a unique partial solution
Solution


Explain
Reset
Learning Blossoms
Chris Piech

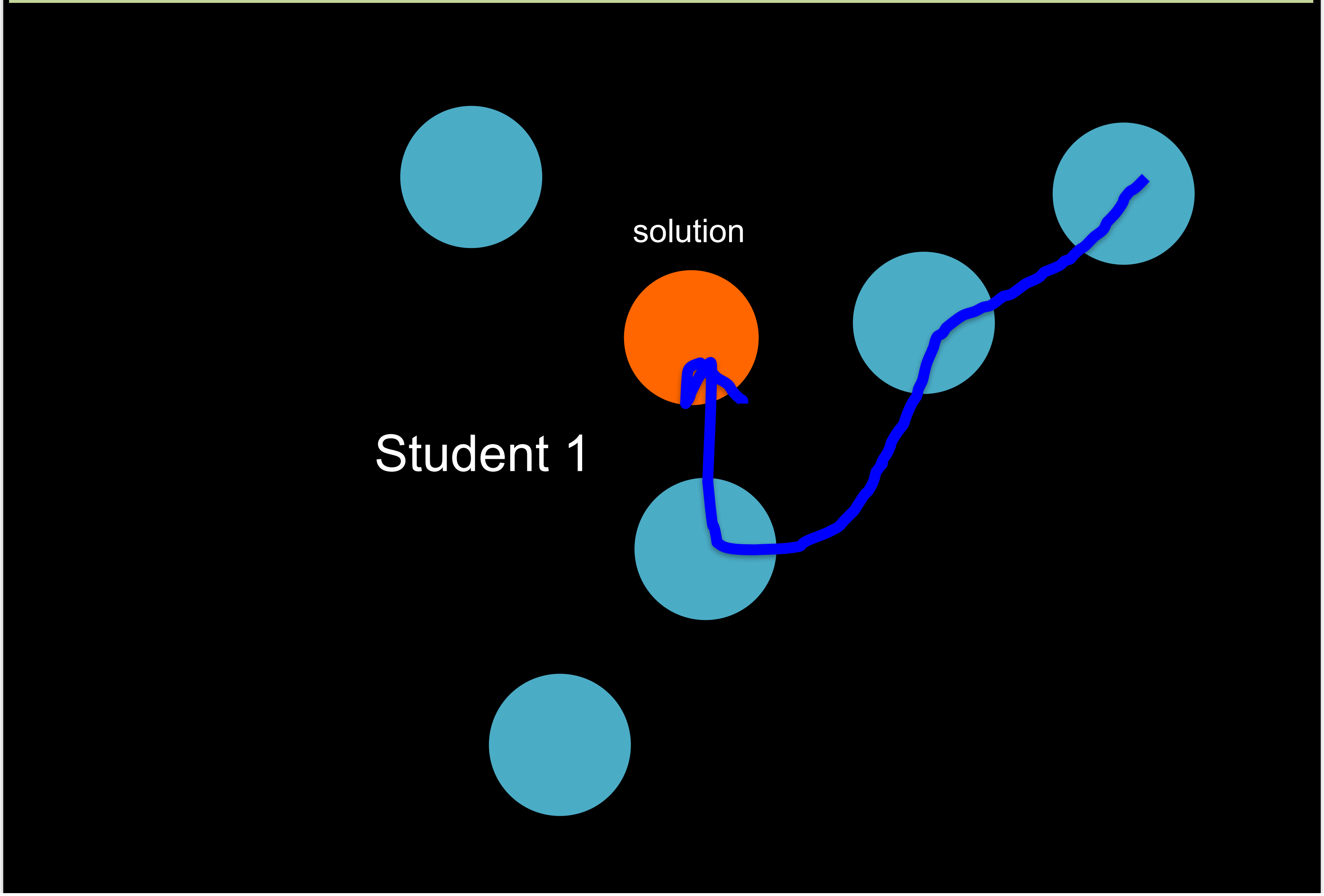
Machine Learning Problem



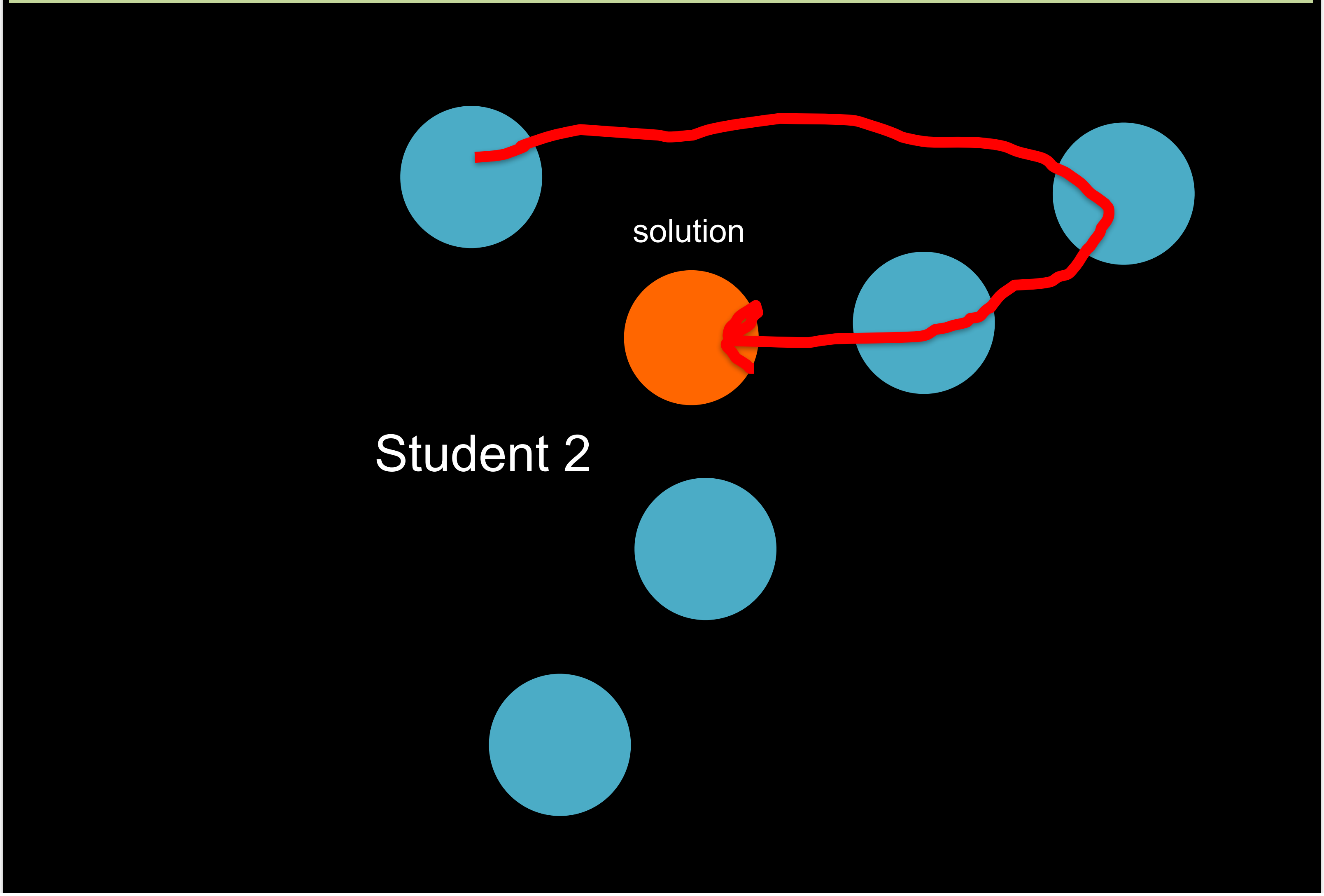
Machine Learning Problem



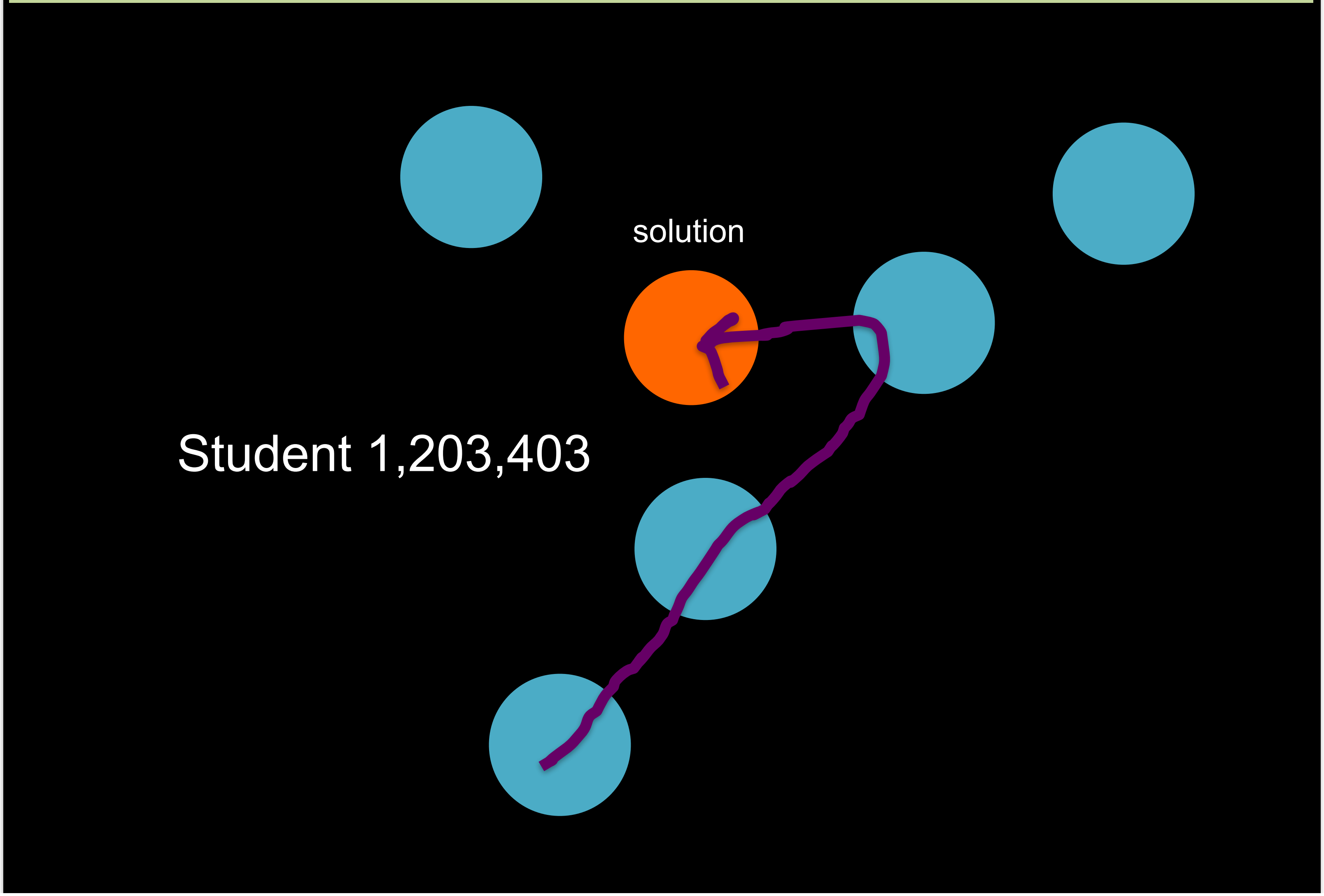
Example Student



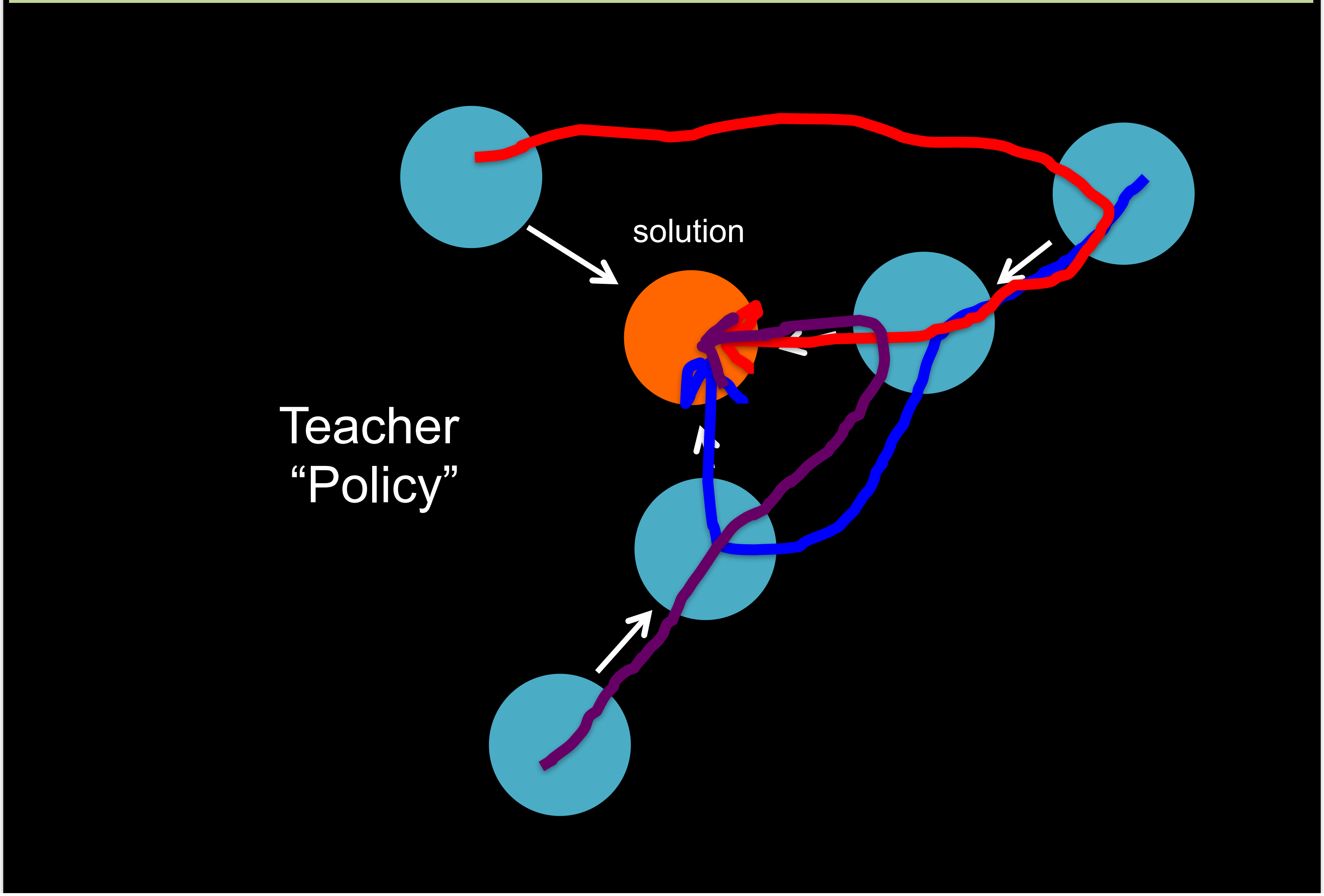
Example Student

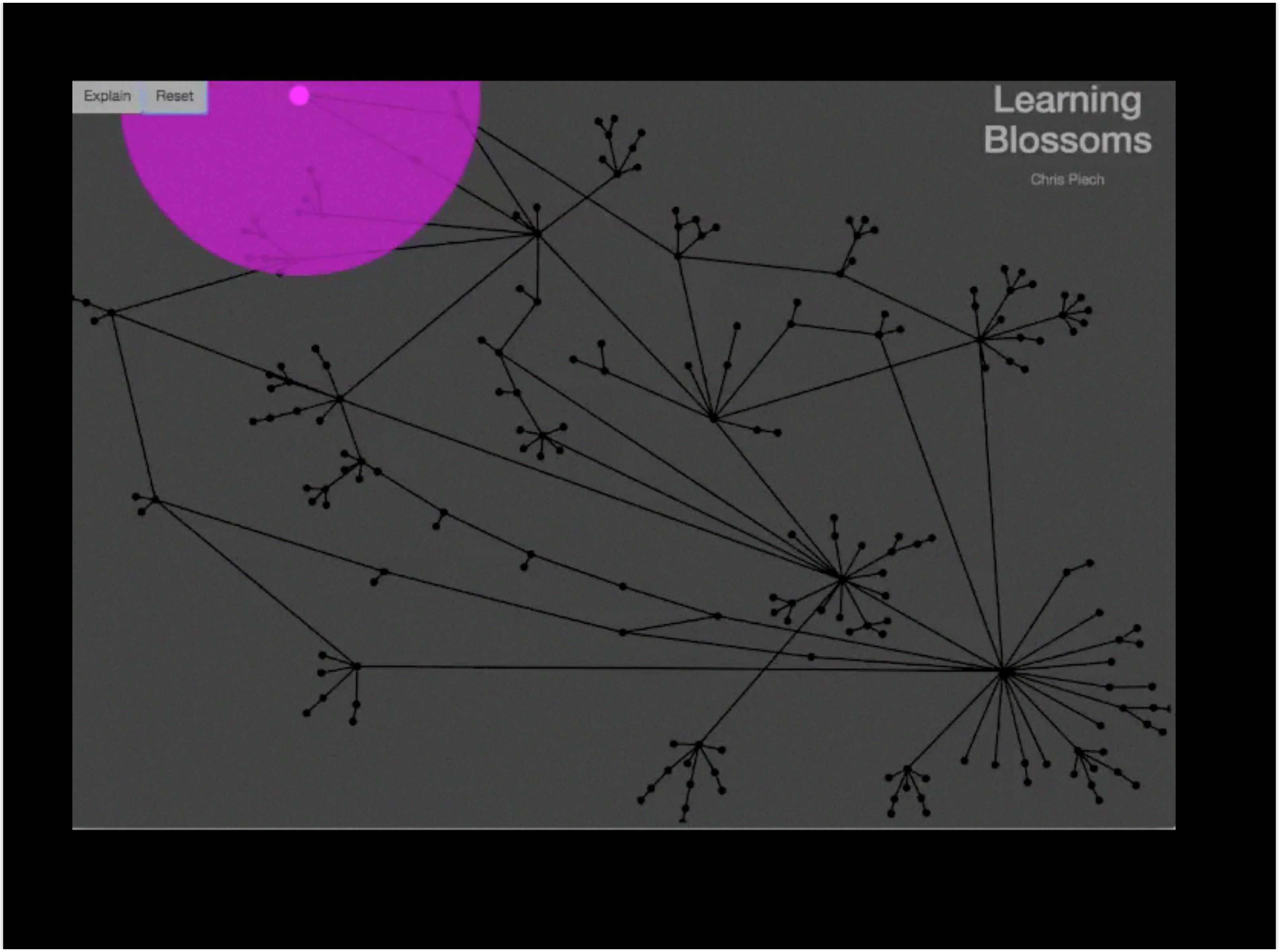


Example Student

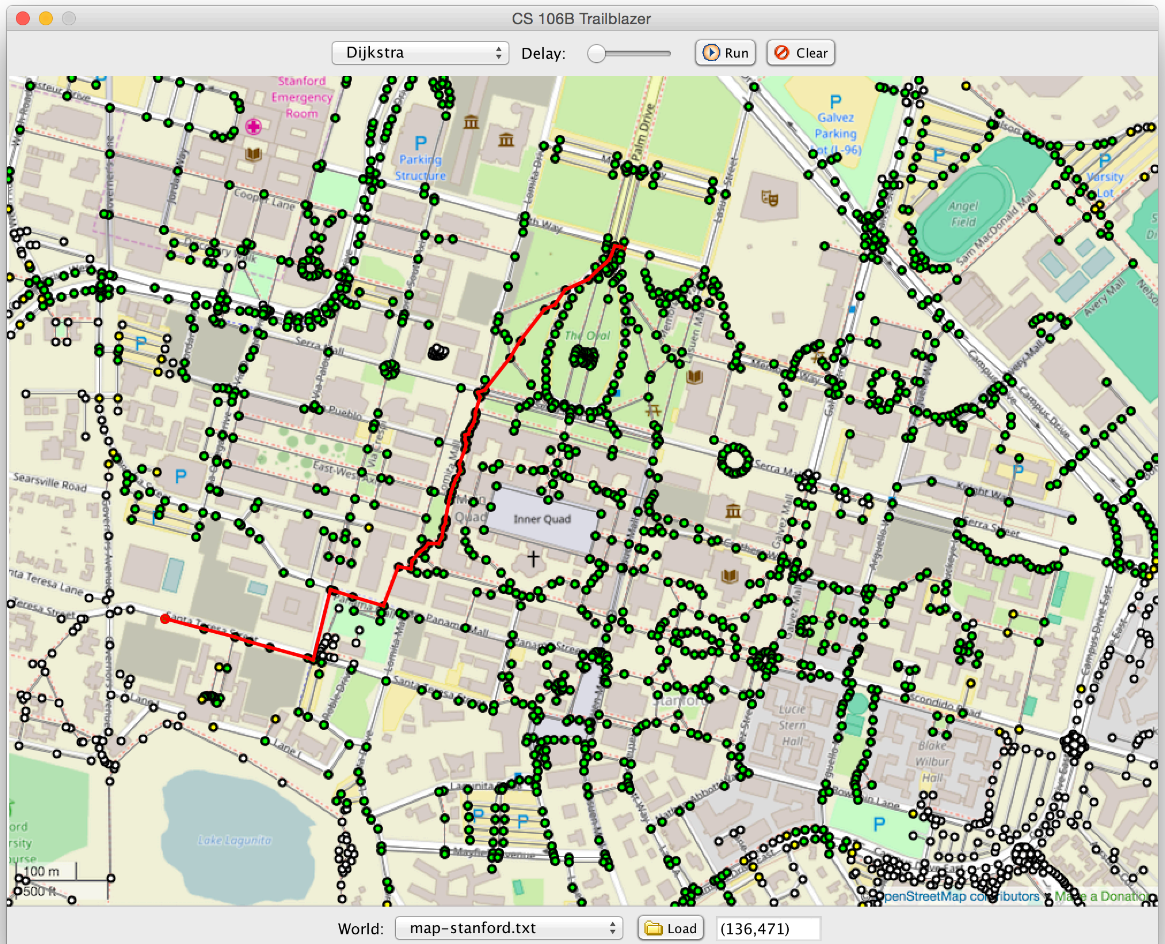


Machine Learning Problem

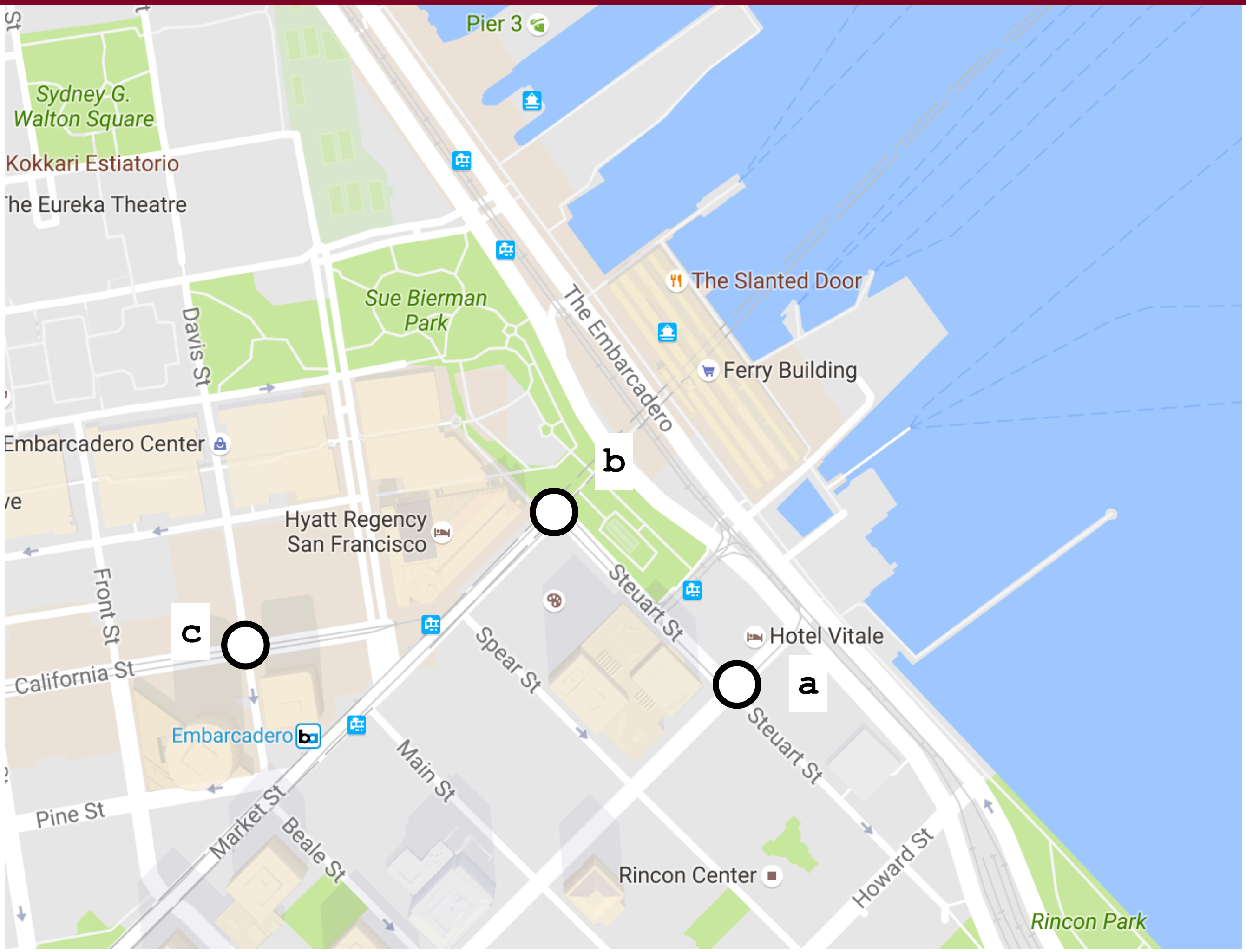



Explain
Reset
Learning Blossoms
Chris Piech

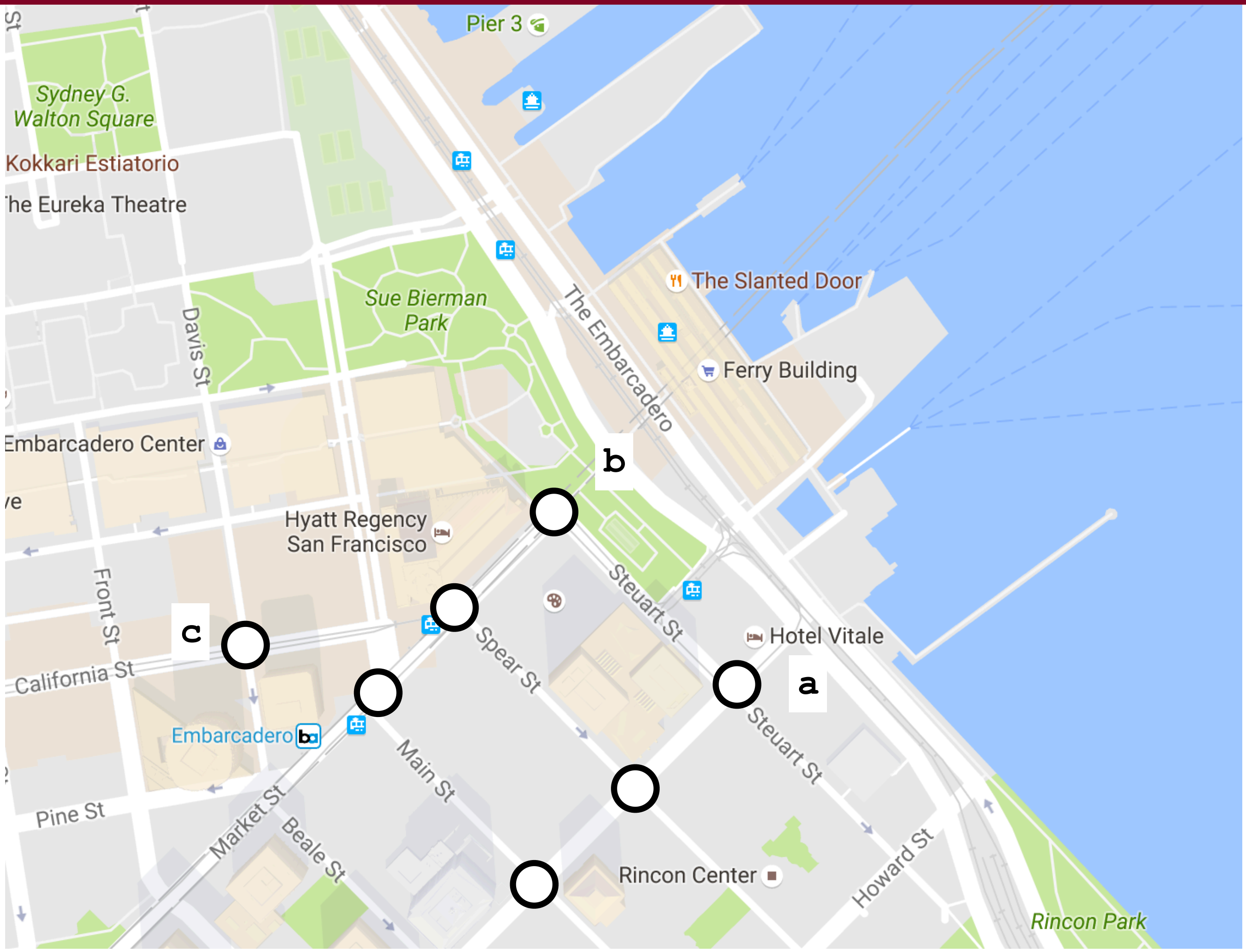
Back to Trailblazer



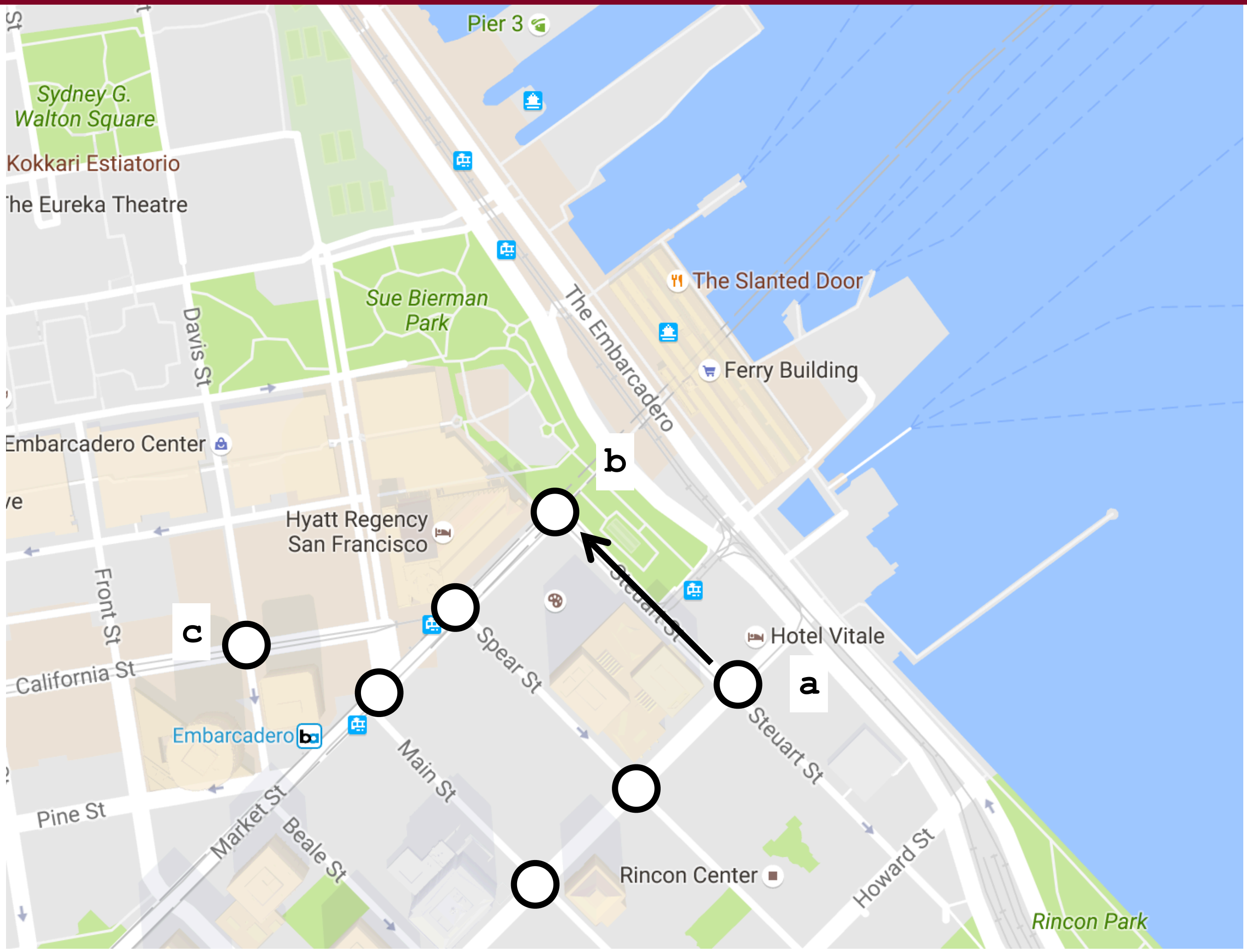
Road Map Node



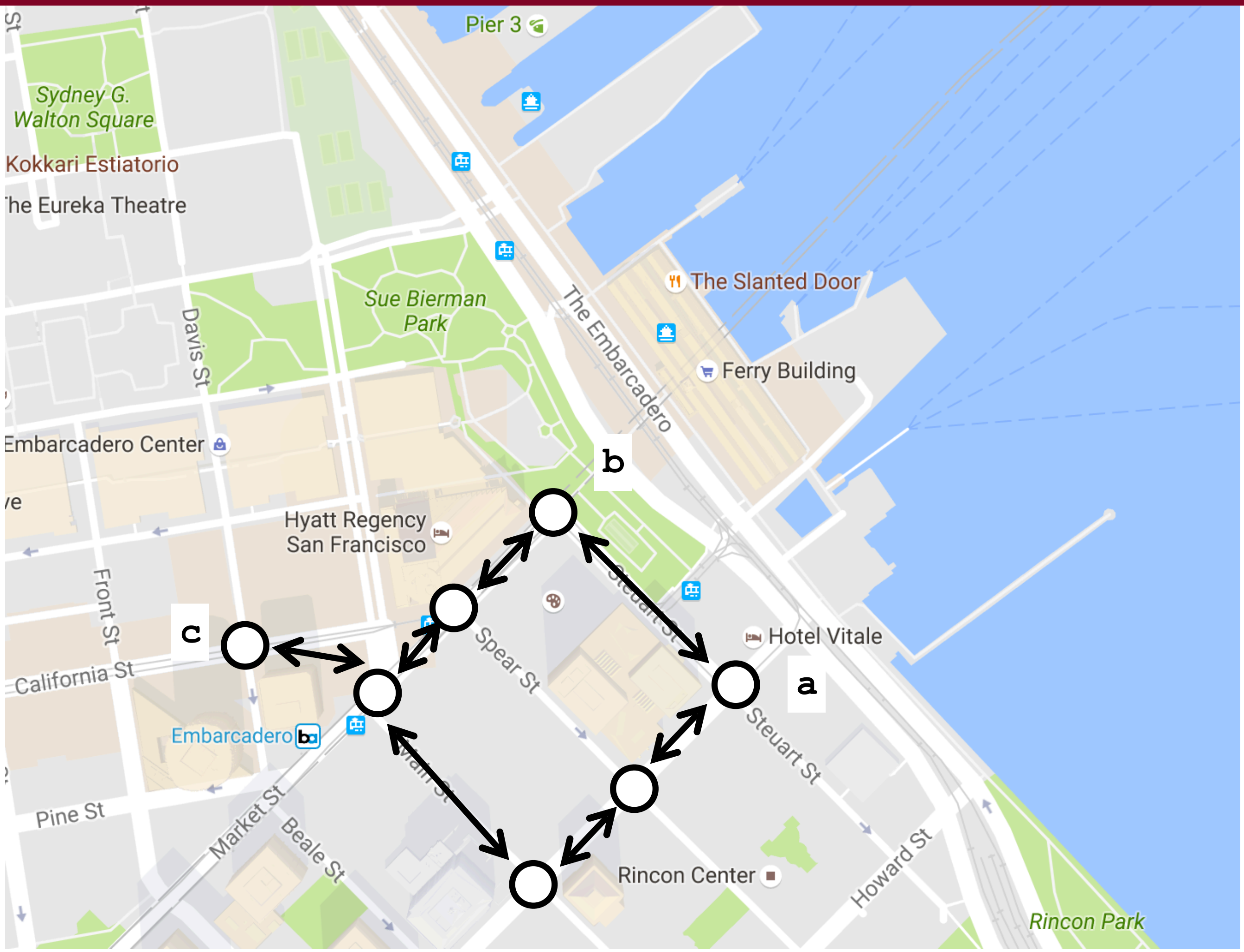
Road Map Node



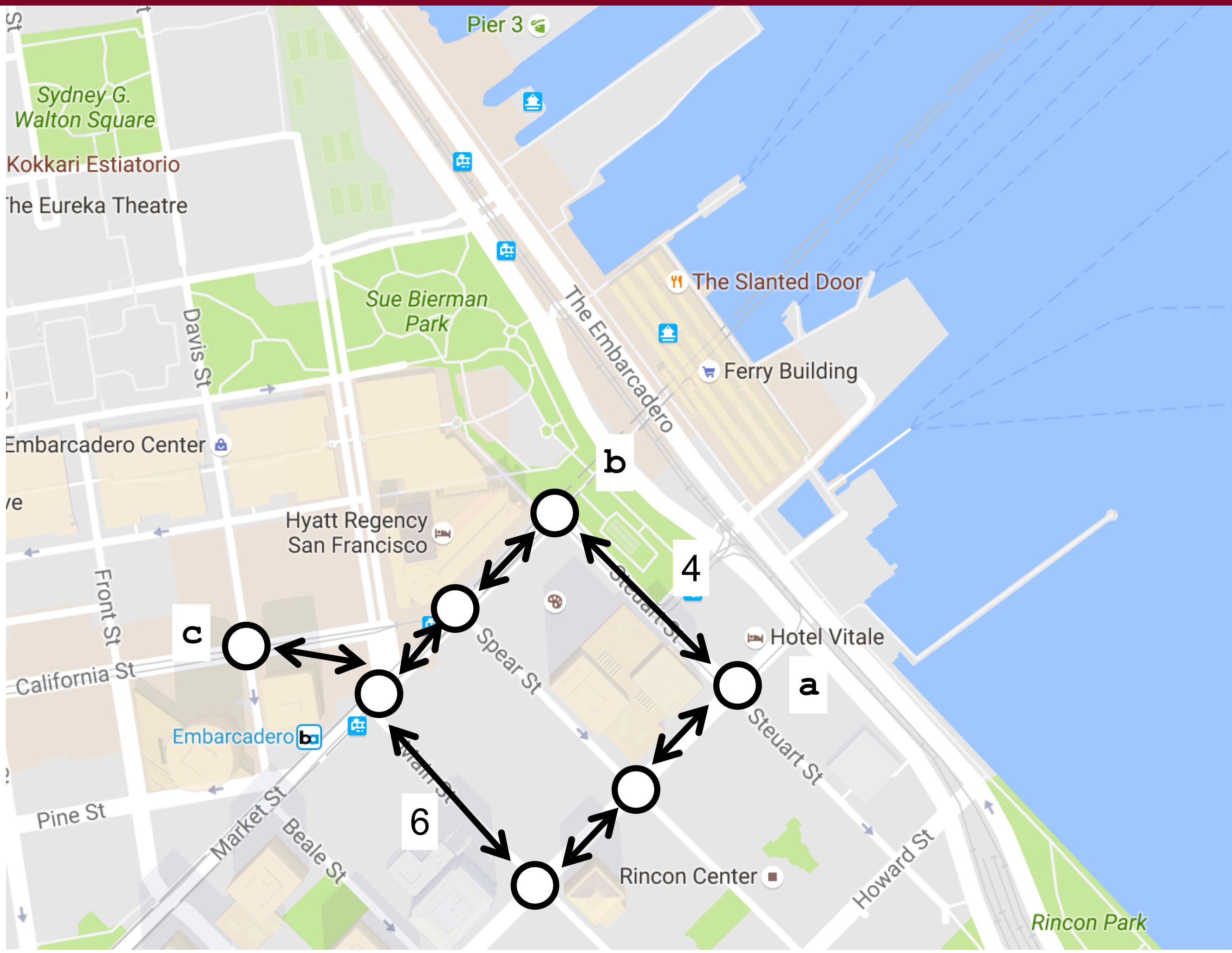
Road Map Edge



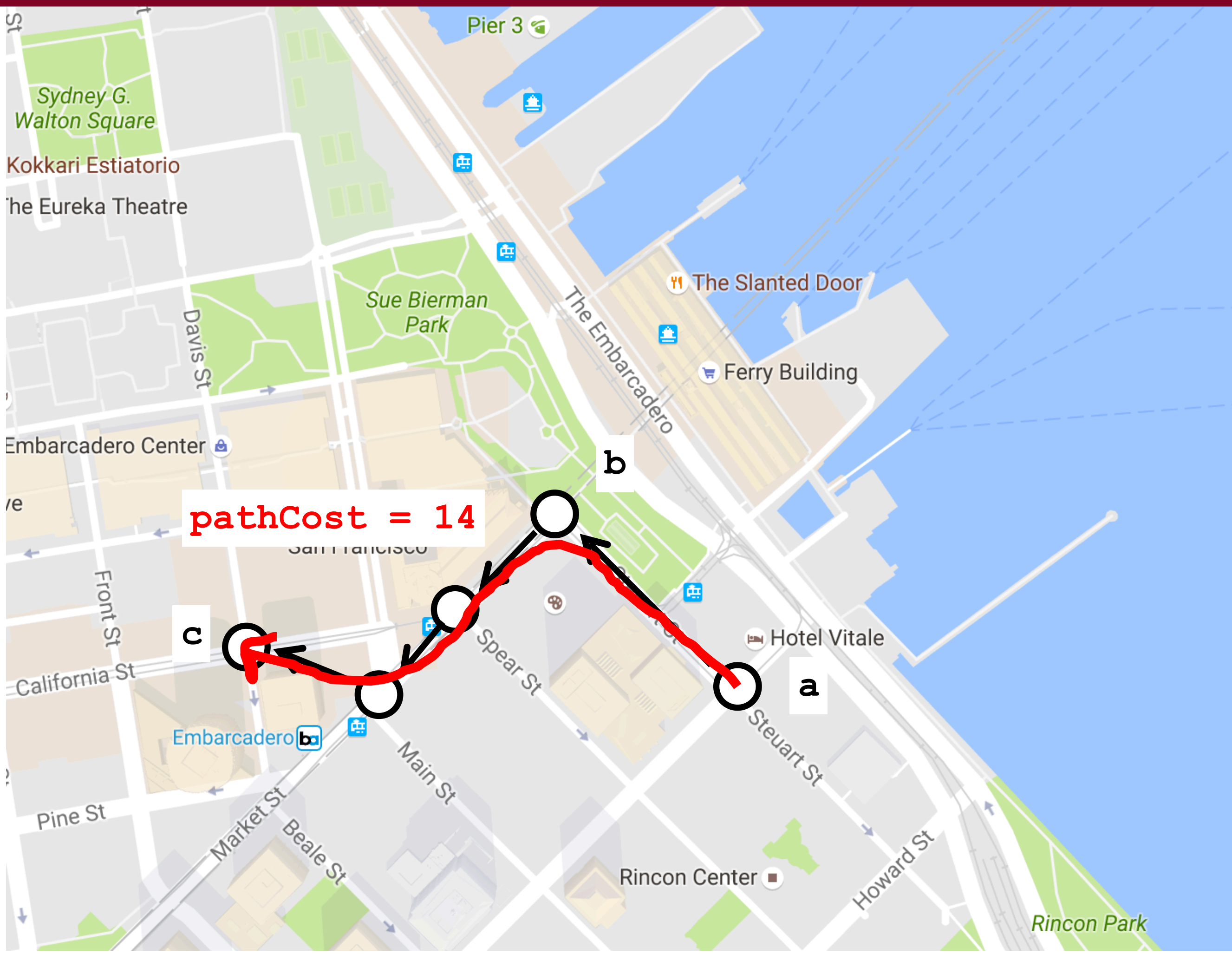
Road Map Edge



Road Map Edge Cost



Road Map Path Cost



Could Google Just Precompute?

How many nodes in google maps graph?

~ 75 million

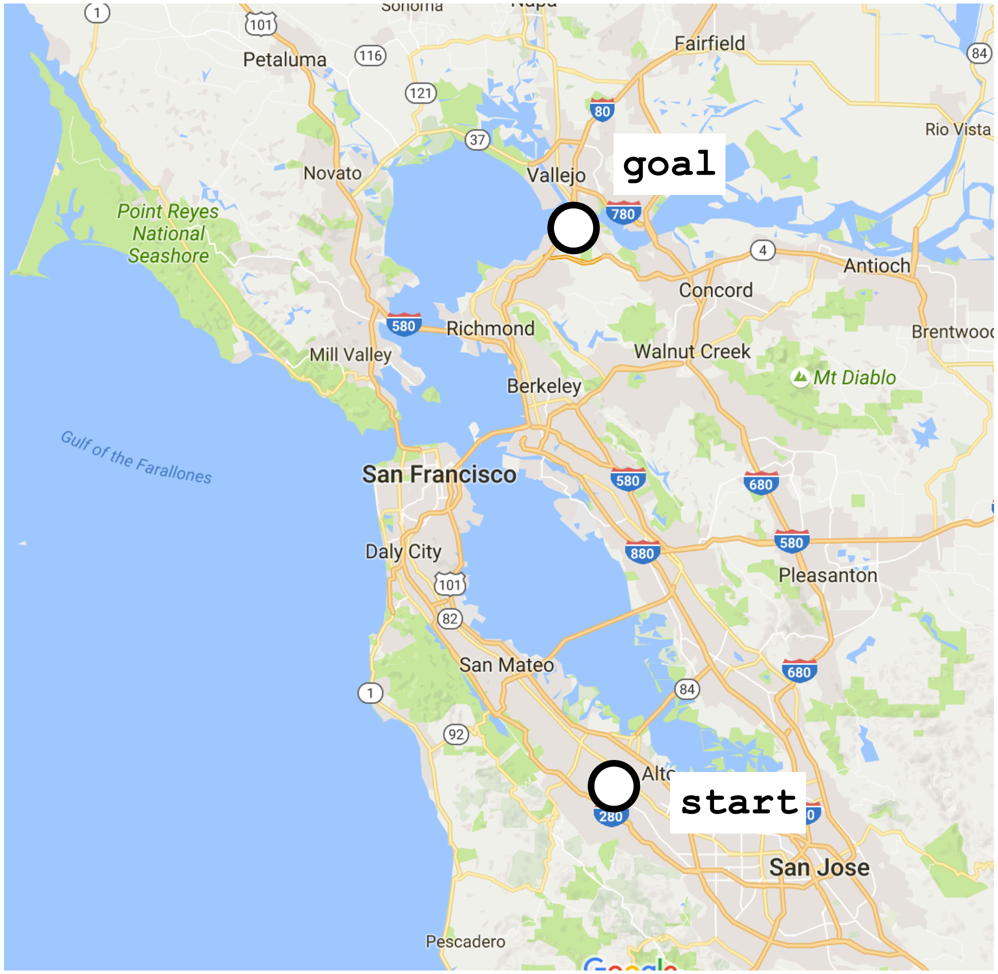
n^2

6×10^{15}

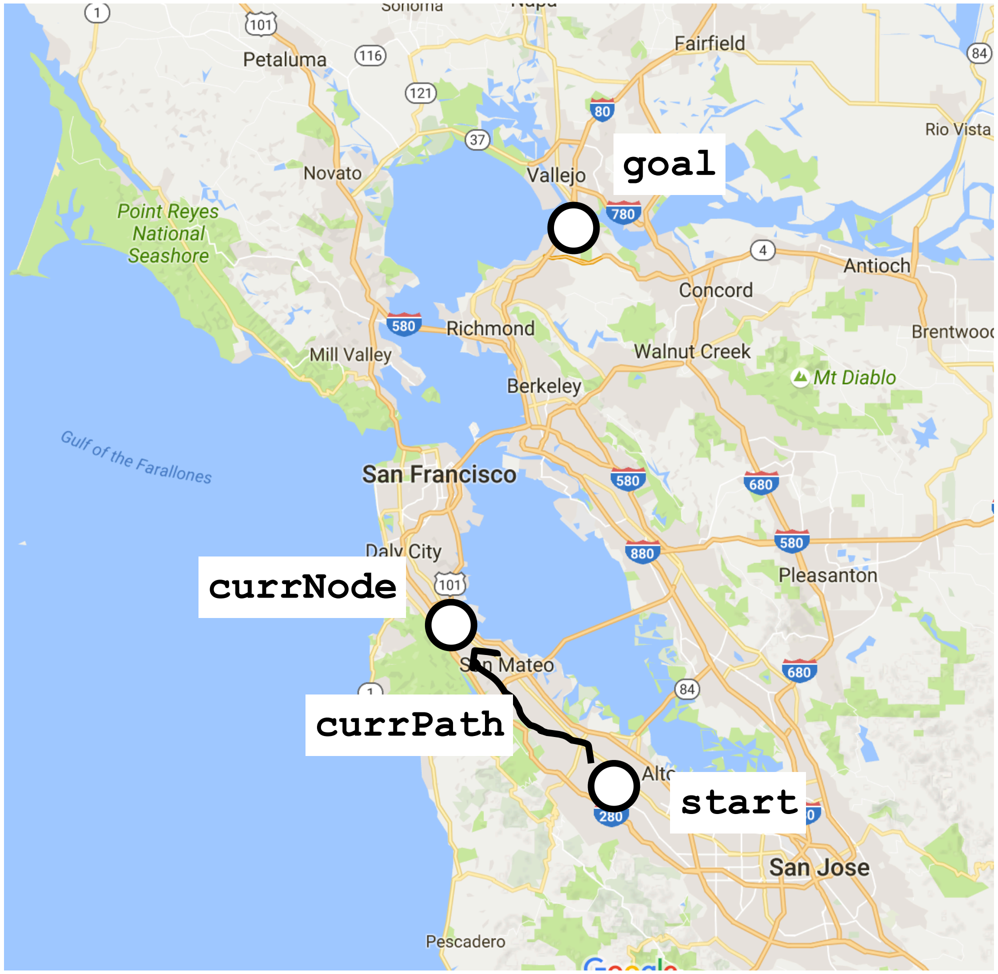
1 petasecond = 31.7 million years

Can you think of a heuristic?

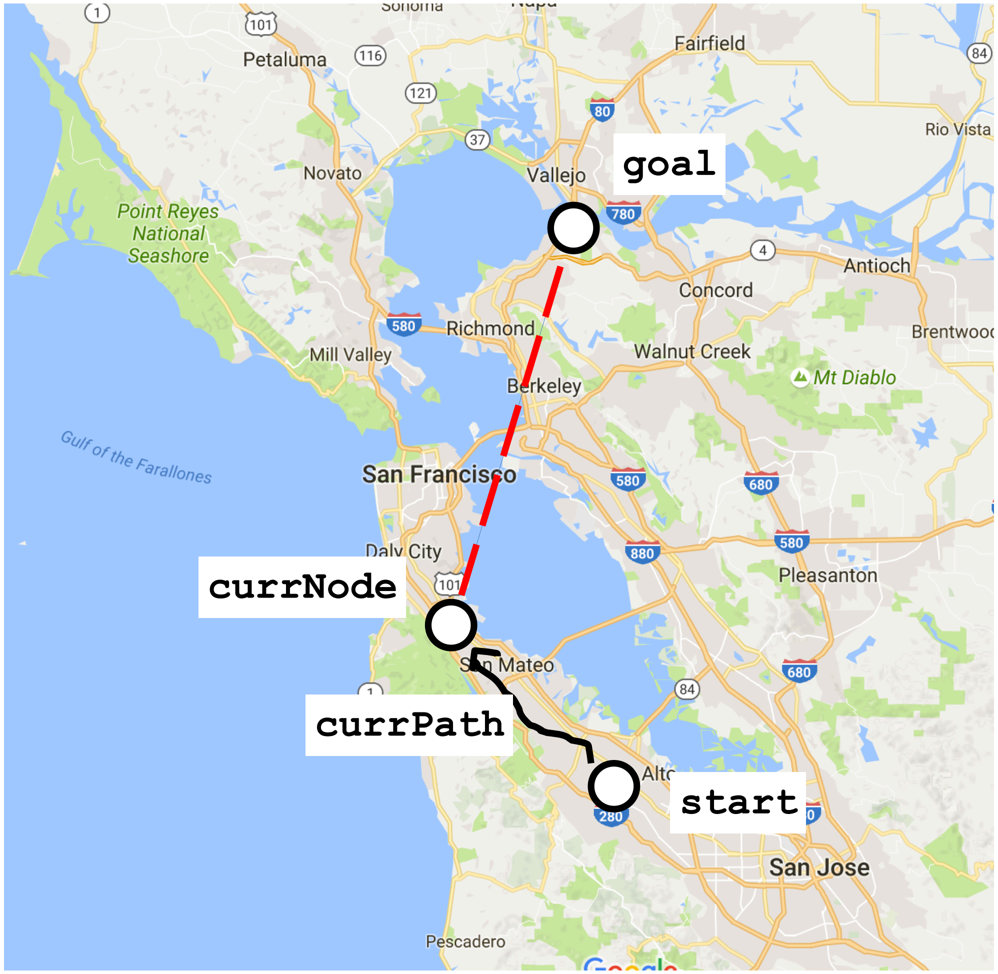
Road Map Heuristic



Road Map Heuristic

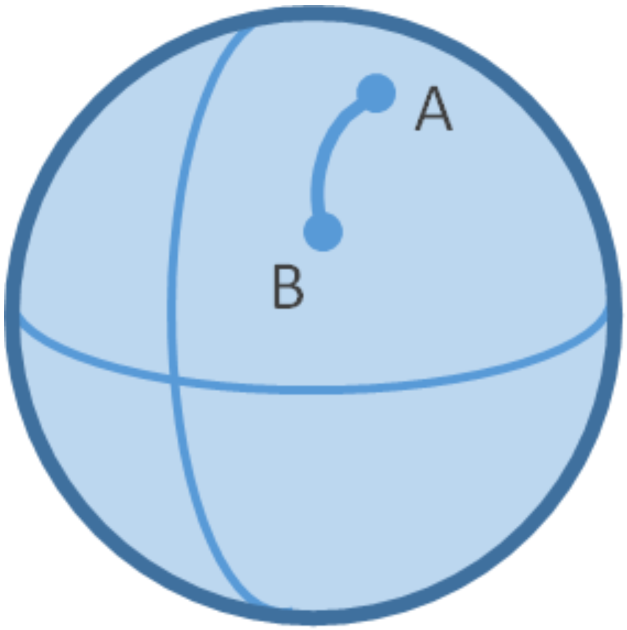


We must *underestimate* this time



Direct Highway

$$\text{Heuristic} = \frac{\text{Distance on surface of earth}}{\text{Speed on fastest highway}}$$

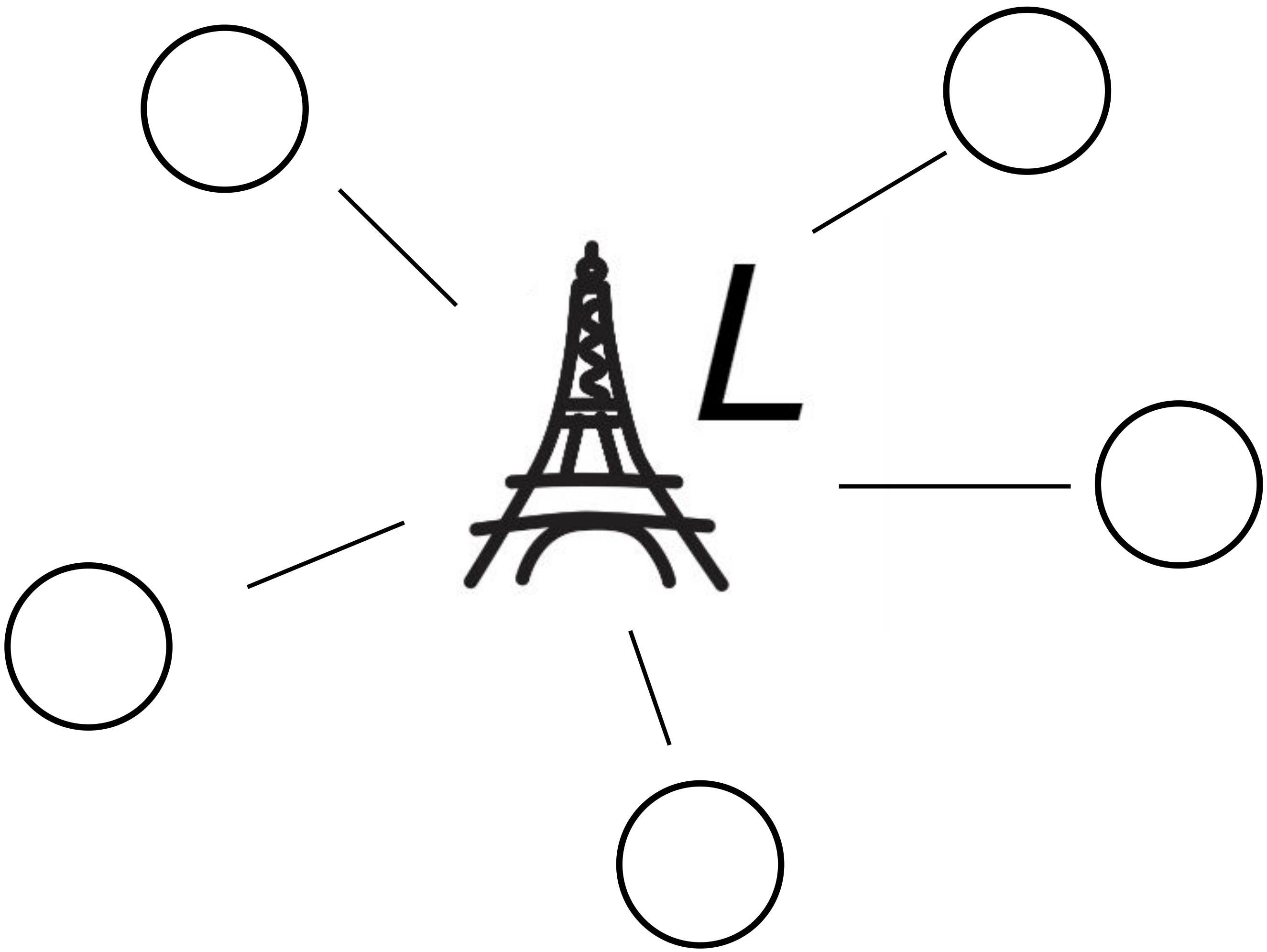


For Trailblazer:

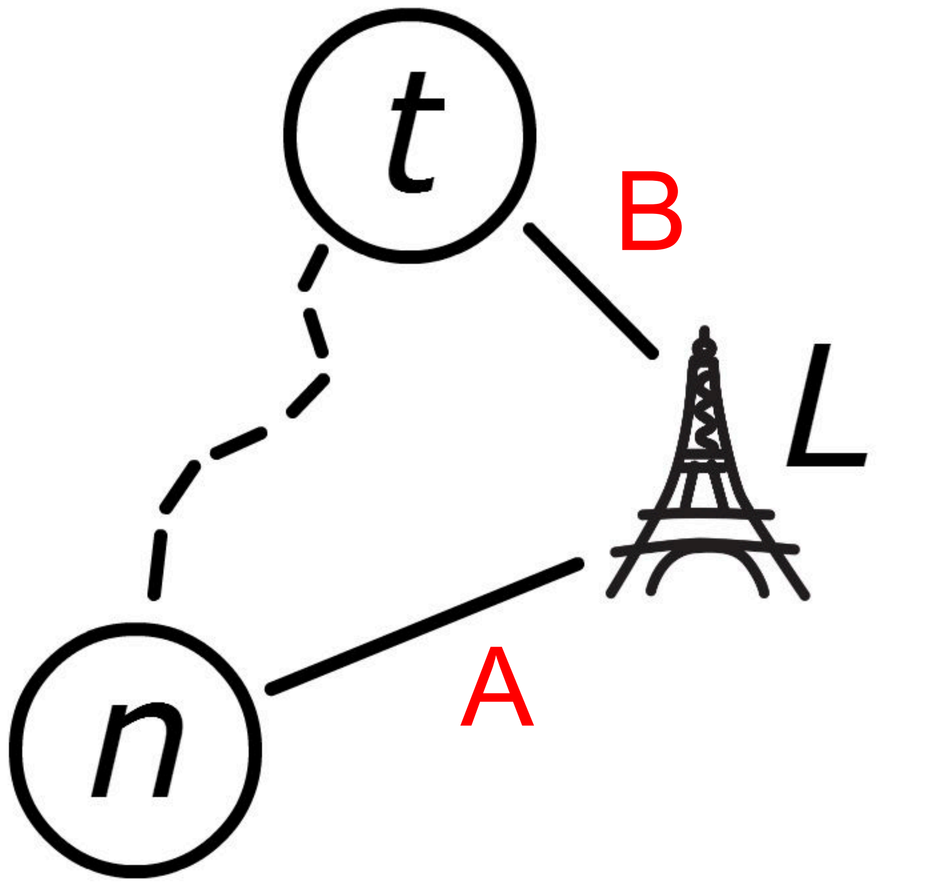
Distance on surface of earth is **`getCrowFlyDistance()`**

Speed on fastest highway is **`getMaxRoadSpeed()`**

Distance to Landmarks



Landmark Heuristic



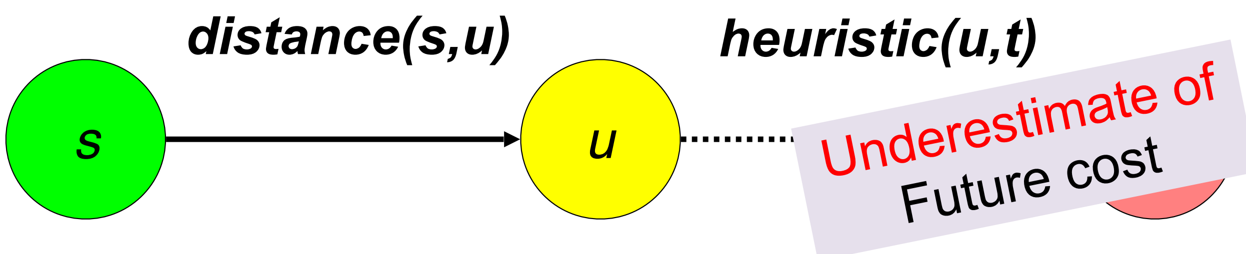
Distance < abs(A – B)

Best of All Heuristics

$$h = \max(h_1, h_2, ..., h_n)$$

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)

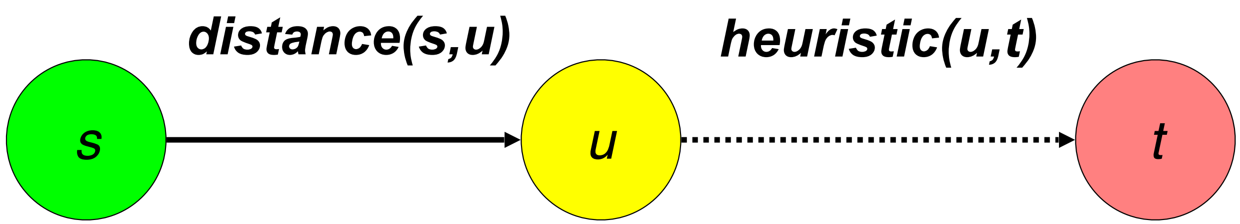


We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



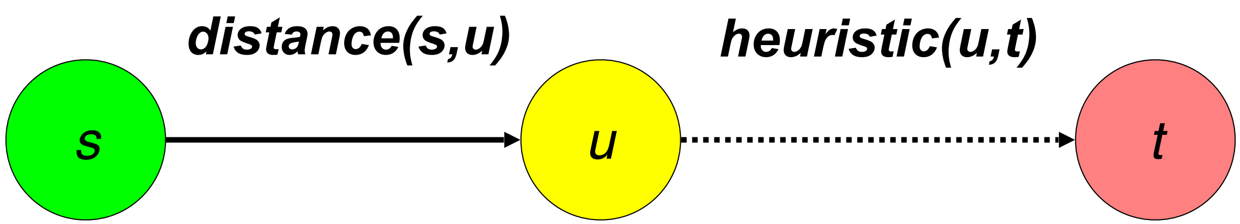
We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Same as Dijkstra

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



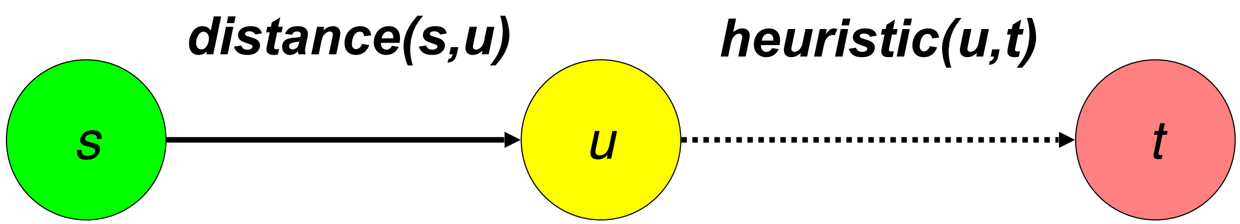
We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Will be the same or faster than Dijkstra, and will find the shortest path (this is the only "admissible" heuristic for A*.

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



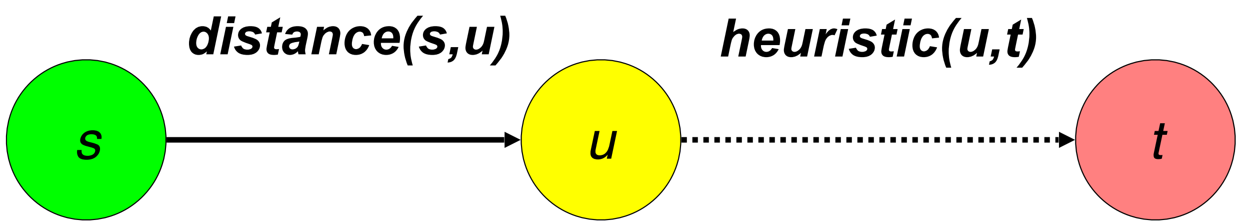
We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Will only follow the best path, and will find the best path fastest (but requires perfect knowledge)

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Won't necessarily find shortest path (but might run even faster)

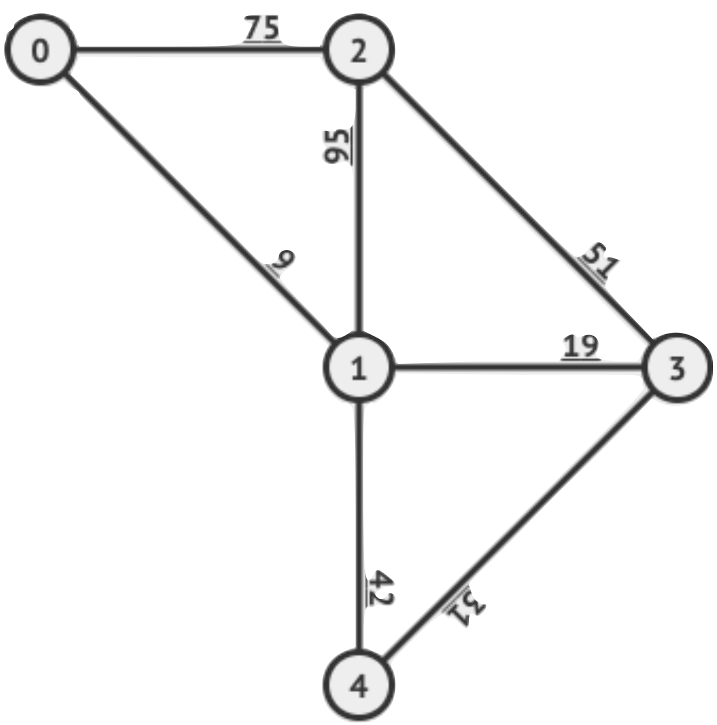
Admissible Heuristic

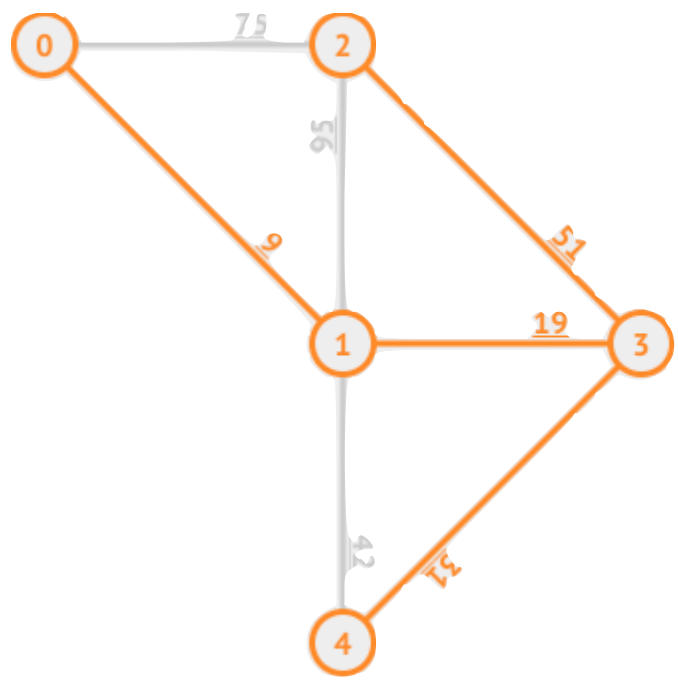
Definition: An admissible heuristic always underestimates the true cost.

Could you precompute this for all your vertices? Yes, but it would not be feasible.

Spanning Trees and Minimum Spanning Trees

Definition: A **Spanning Tree (ST)** of a connected undirected weighted graph **G** is a subgraph of **G** that is a **tree** and **connects (spans) all vertices of G**. A graph **G** can have multiple STs. A **Minimum Spanning Tree (MST)** of **G** is a ST of **G** that has the **smallest total weight** among the various STs. A graph **G** can have multiple MSTs but the MST weight is unique.





Minimum Spanning Tree

Kruskal's Algorithm to find a Minimum Spanning Tree

- **Kruskal's algorithm**: Finds a MST in a given graph.

function **kruskal**(graph):

Remove all edges from the graph.

Place all edges into a **priority queue** based on their weight (cost).

While the priority queue is not empty:

  Dequeue an edge *e* from the priority queue.

  If *e*'s endpoints aren't already connected to one another,

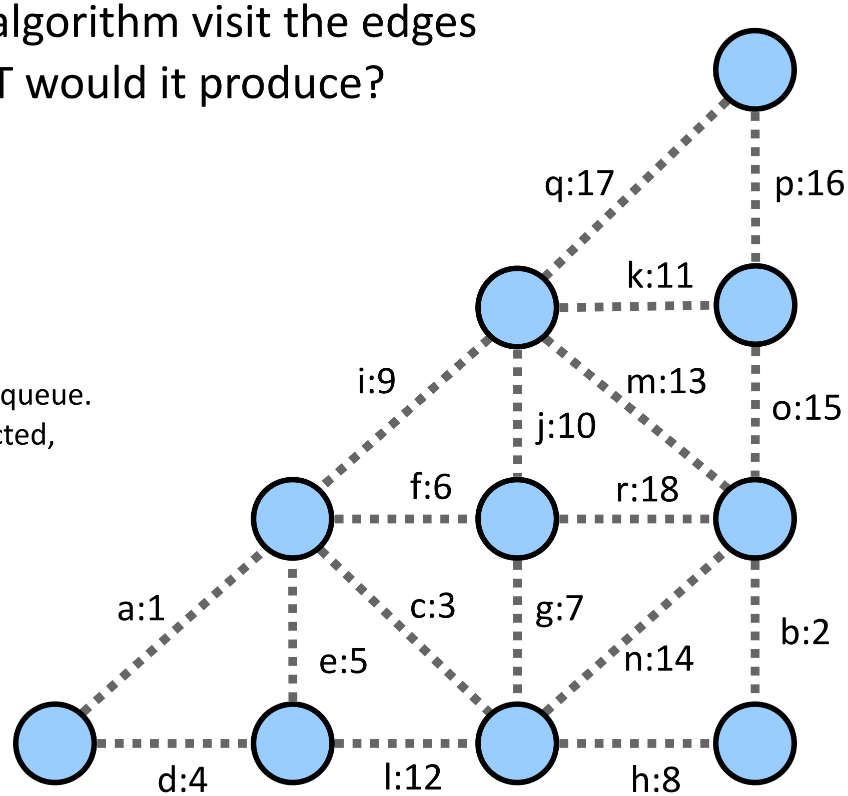
    add that edge into the graph.

  Otherwise, skip the edge.

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

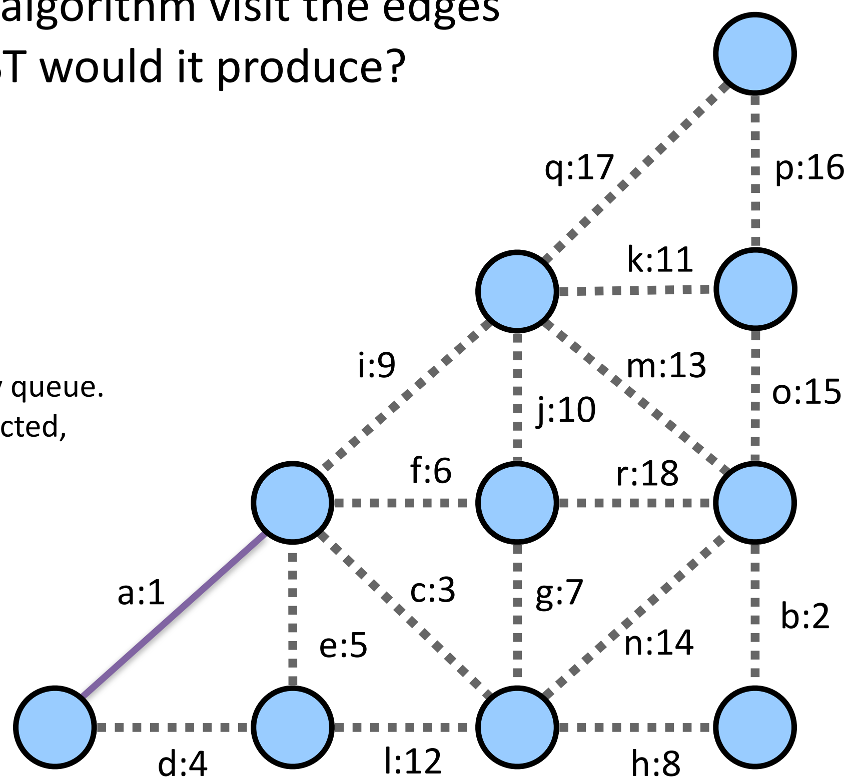


pq = {a:1, b:2, c:3, d:4, e:5, f:6, g:7, h:8, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

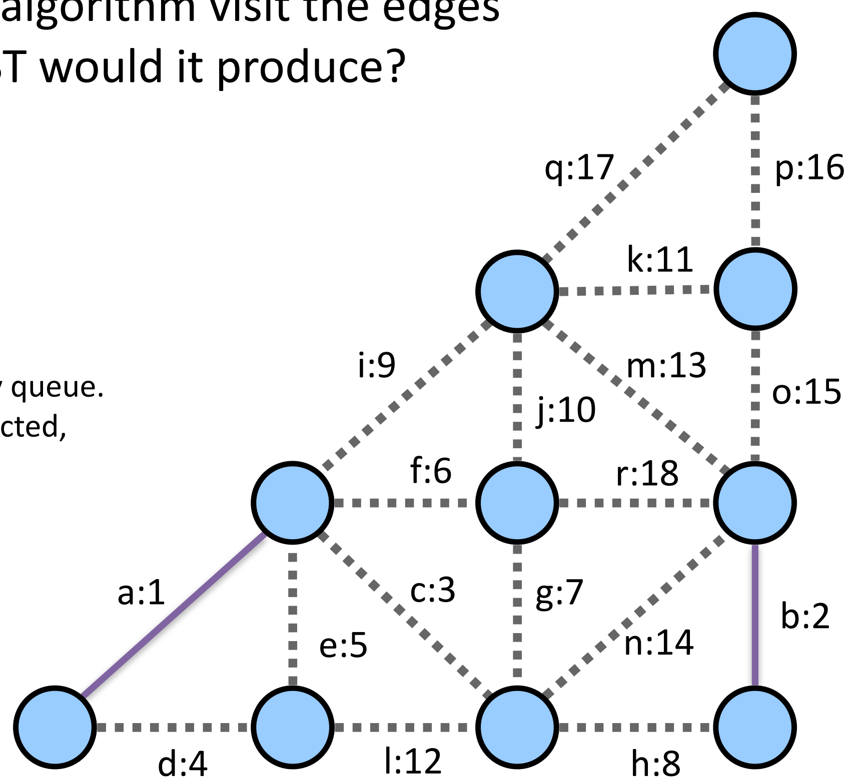


pq = {**a:1**, b:2, c:3, d:4, e:5, f:6, g:7, h:8, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
 based on their weight (cost).
 While the priority queue is not empty:
 Dequeue an edge *e* from the priority queue.
 If *e*'s endpoints aren't already connected,
 add that edge into the graph.
 Otherwise, skip the edge.

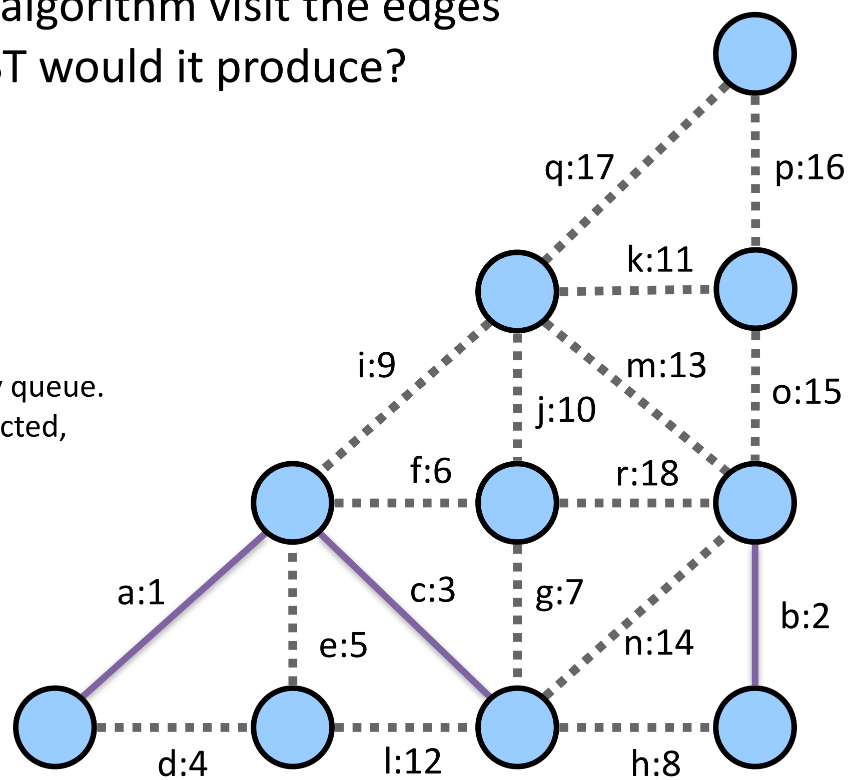


pq = {**b:2**, c:3, d:4, e:5, f:6, g:7, h:8, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
  Remove all edges from the graph.
  Place all edges into a priority queue based on their weight (cost).
  While the priority queue is not empty:
    Dequeue an edge *e* from the priority queue.
    If *e*'s endpoints aren't already connected, add that edge into the graph.
    Otherwise, skip the edge.

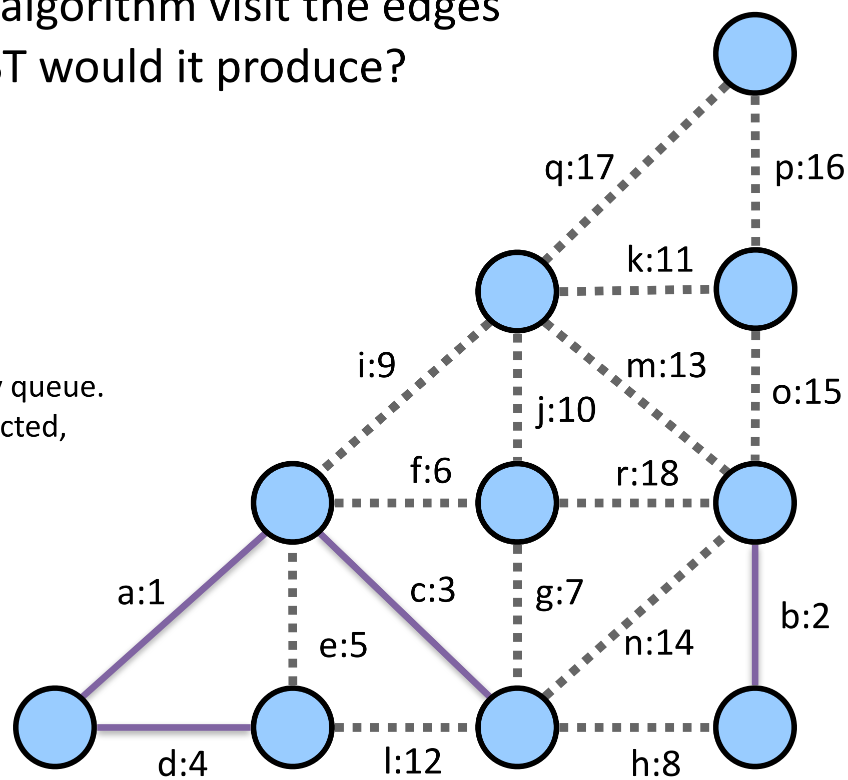


pq = {**c:3**, d:4, e:5, f:6, g:7, h:8, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

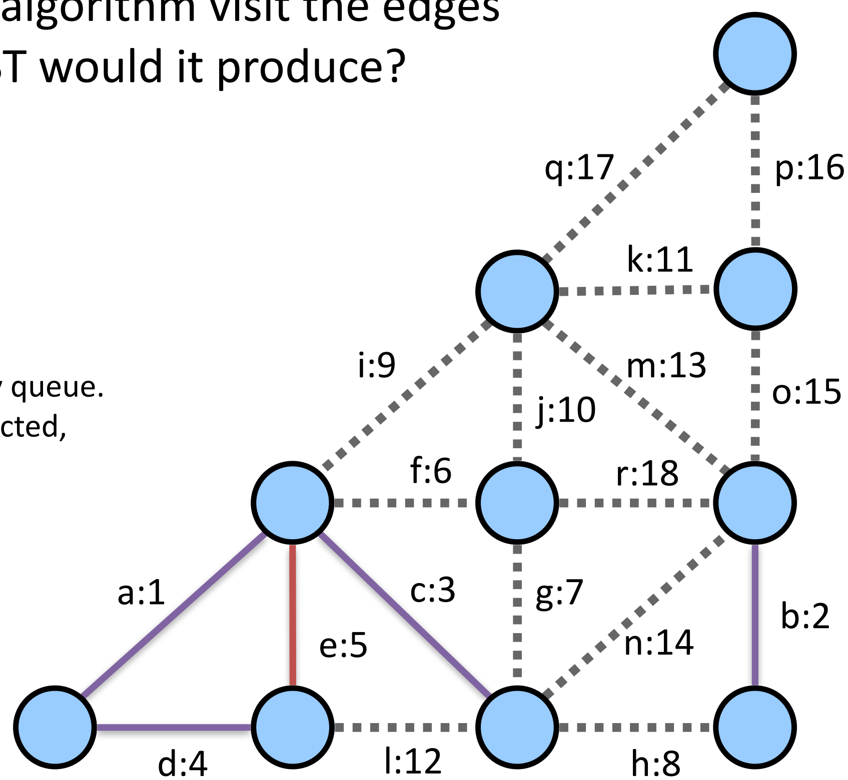


pq = {**d:4**, e:5, f:6, g:7, h:8, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
Remove all edges from the graph.
Place all edges into a priority queue based on their weight (cost).
While the priority queue is not empty:
Dequeue an edge *e* from the priority queue.
If *e*'s endpoints aren't already connected, add that edge into the graph.
Otherwise, skip the edge.

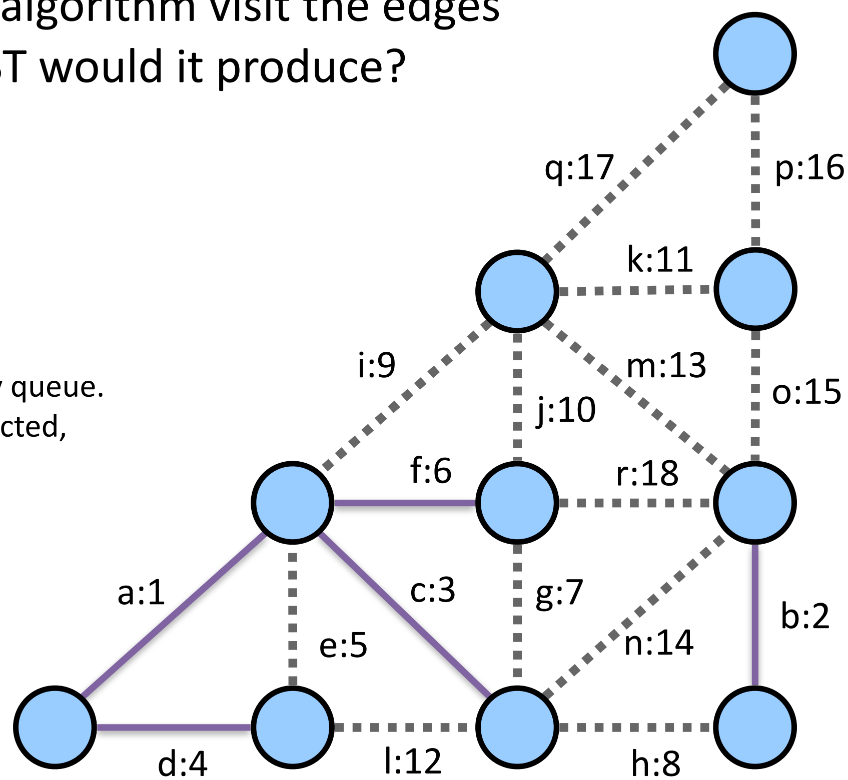


pq = {**e:5**, f:6, g:7, h:8, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected, add that edge into the graph.
  Otherwise, skip the edge.

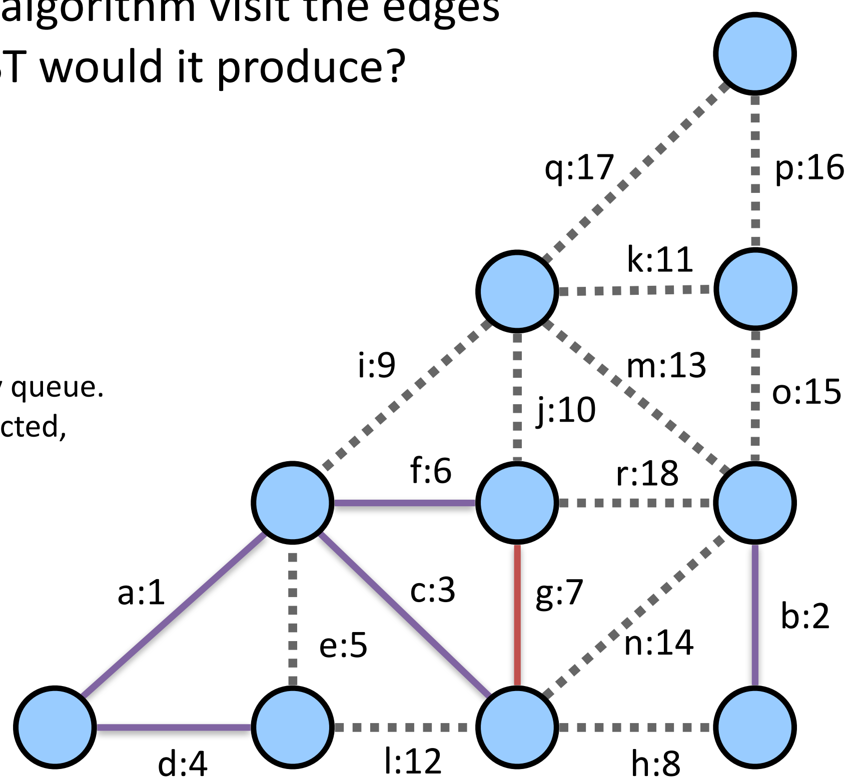


pq = {**f:6**, g:7, h:8, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
  Remove all edges from the graph.
  Place all edges into a priority queue
    based on their weight (cost).
  While the priority queue is not empty:
    Dequeue an edge *e* from the priority queue.
    If *e*'s endpoints aren't already connected,
     add that edge into the graph.
    Otherwise, skip the edge.

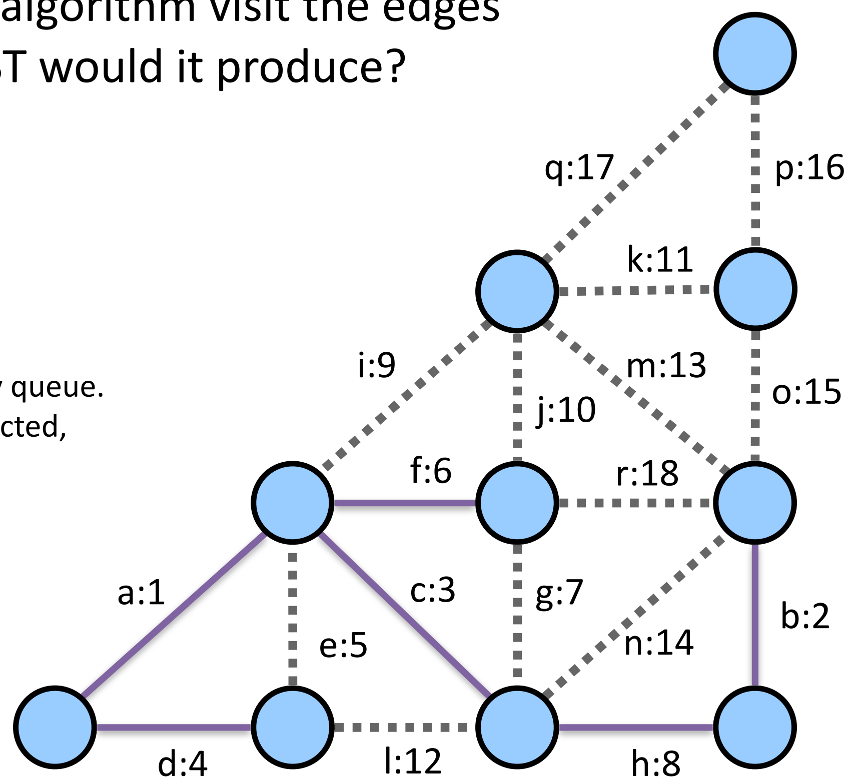


pq = {**g:7**, h:8, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

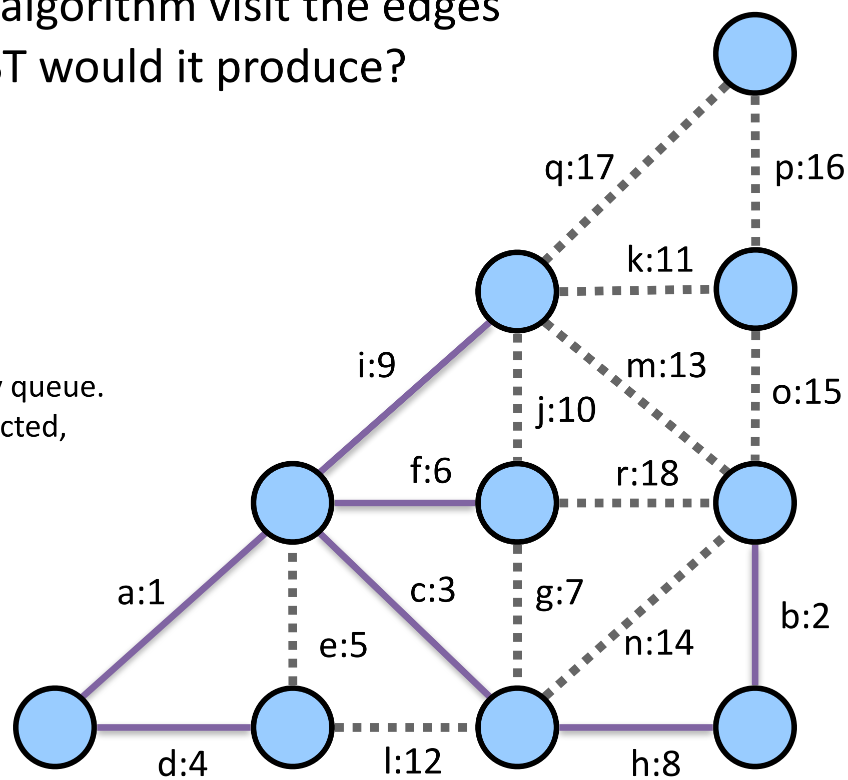


pq = {**h:8**, i:9, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

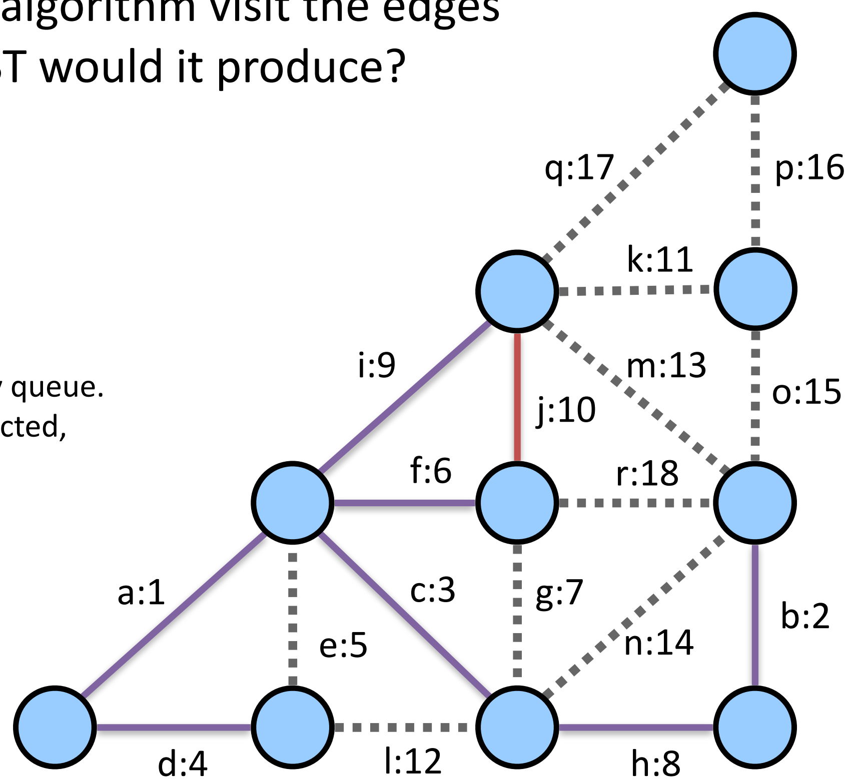


pq = {**i:9**, j:10, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
Remove all edges from the graph.
Place all edges into a priority queue based on their weight (cost).
While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected, add that edge into the graph.
  Otherwise, skip the edge.

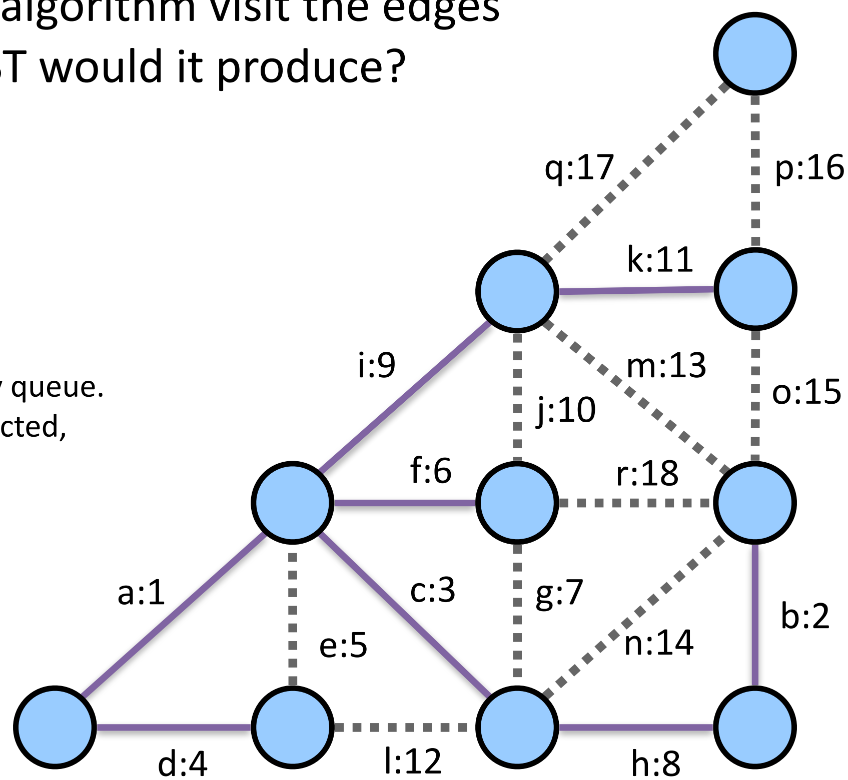


pq = {**j:10**, k:11, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

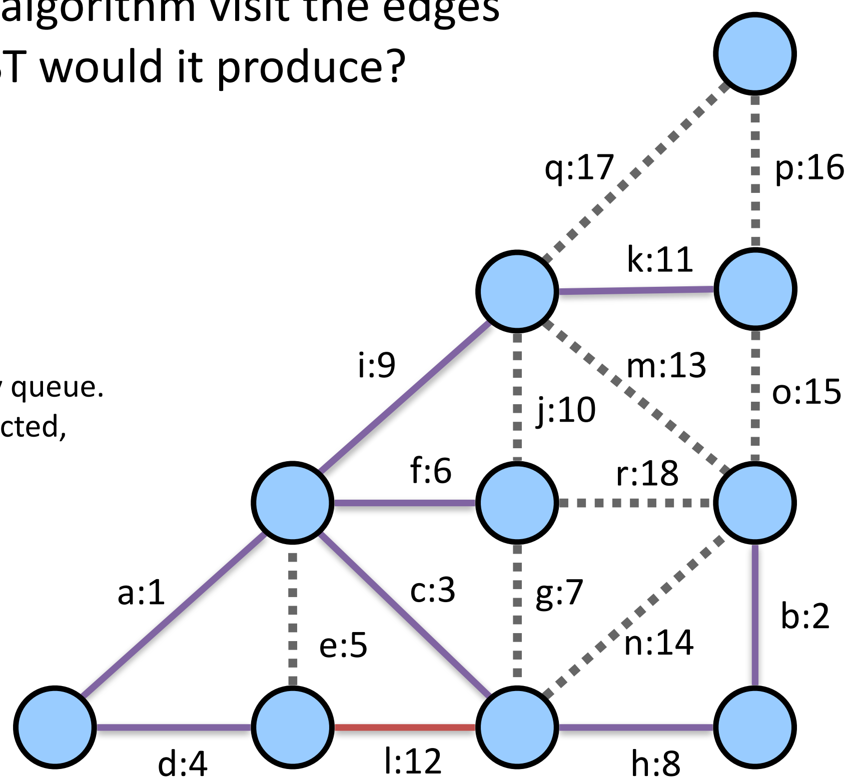


pq = {**k:11**, l:12, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

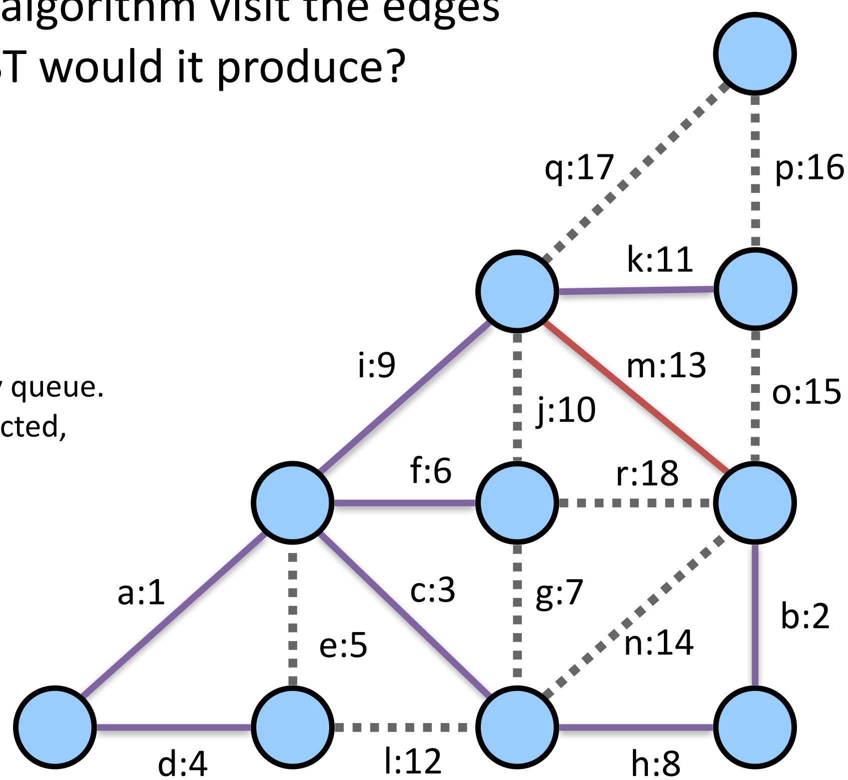


pq = {**l:12**, m:13, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

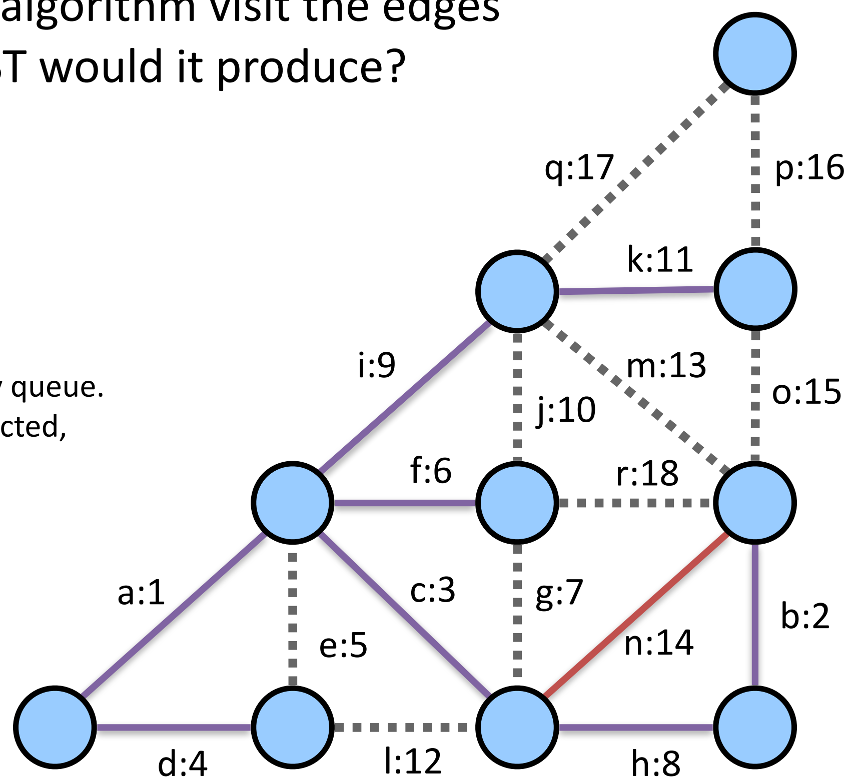


pq = {**m:13**, n:14, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected, add that edge into the graph.
  Otherwise, skip the edge.

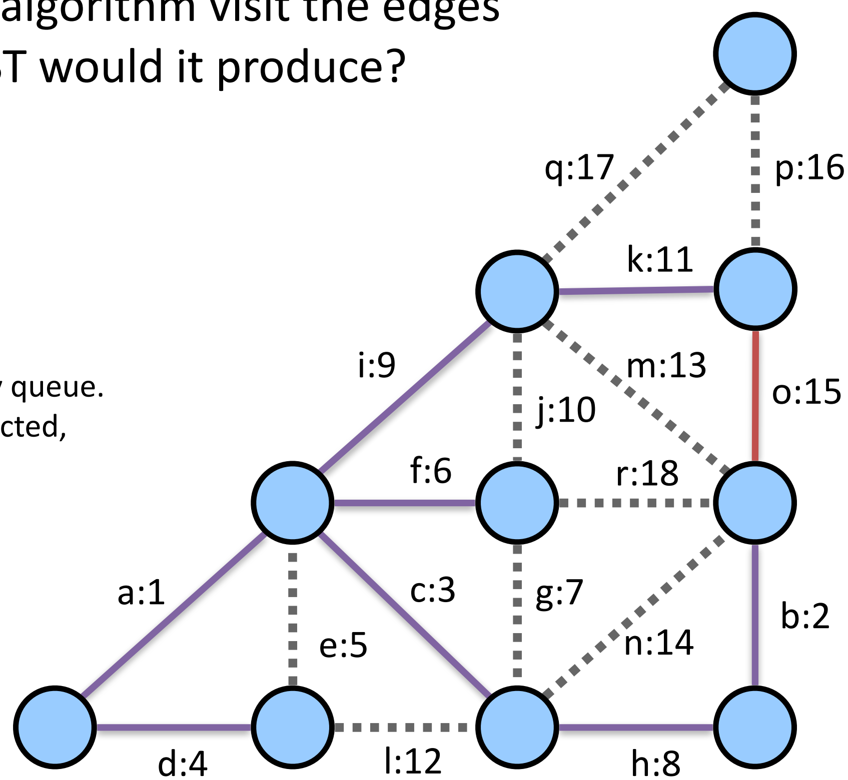


pq = {**n:14**, o:15, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

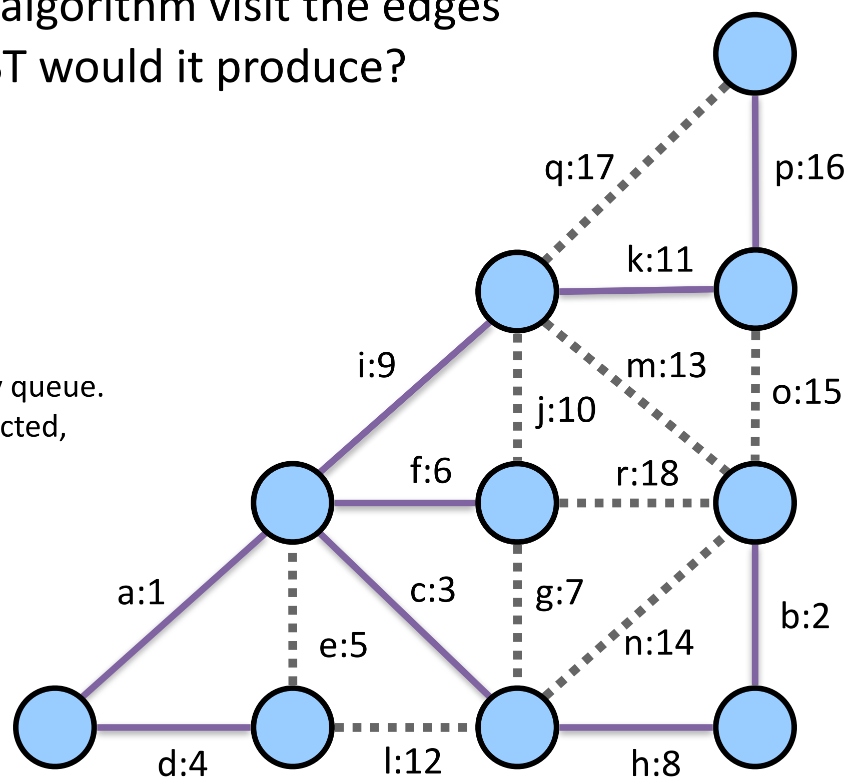


pq = {**o:15**, p:16, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

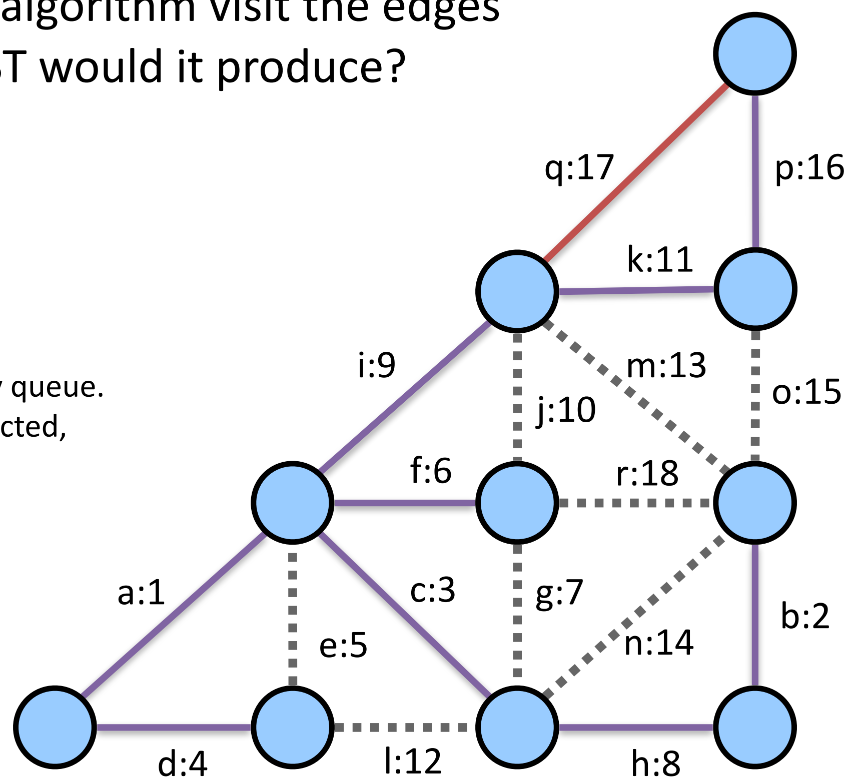


pq = {**p:16**, q:17, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

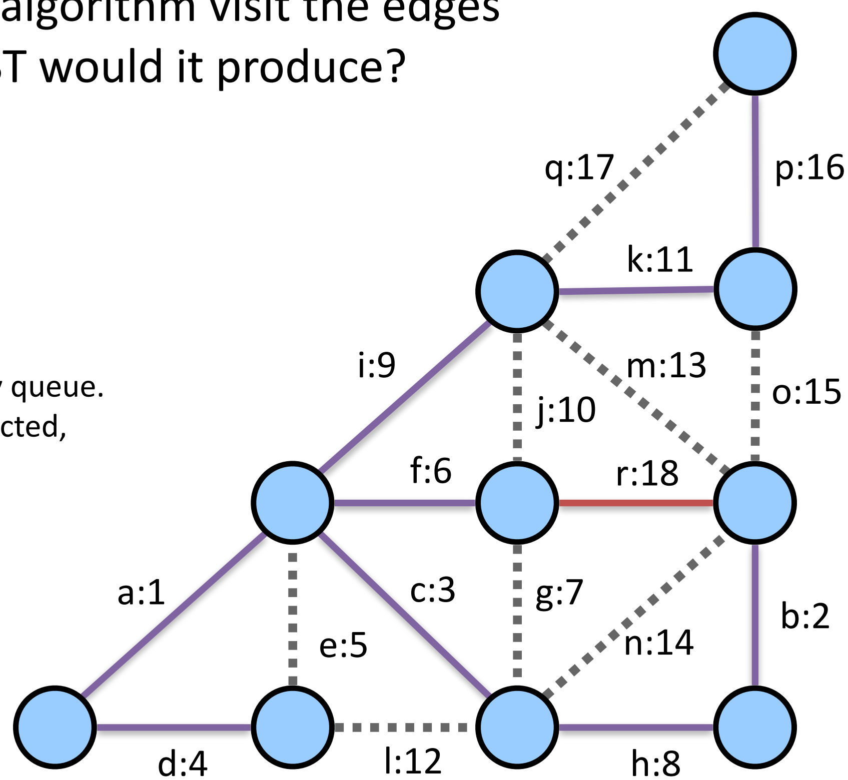


pq = {**q:17**, r:18}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

function **kruskal**(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
  based on their weight (cost).
 While the priority queue is not empty:
  Dequeue an edge *e* from the priority queue.
  If *e*'s endpoints aren't already connected,
   add that edge into the graph.
  Otherwise, skip the edge.

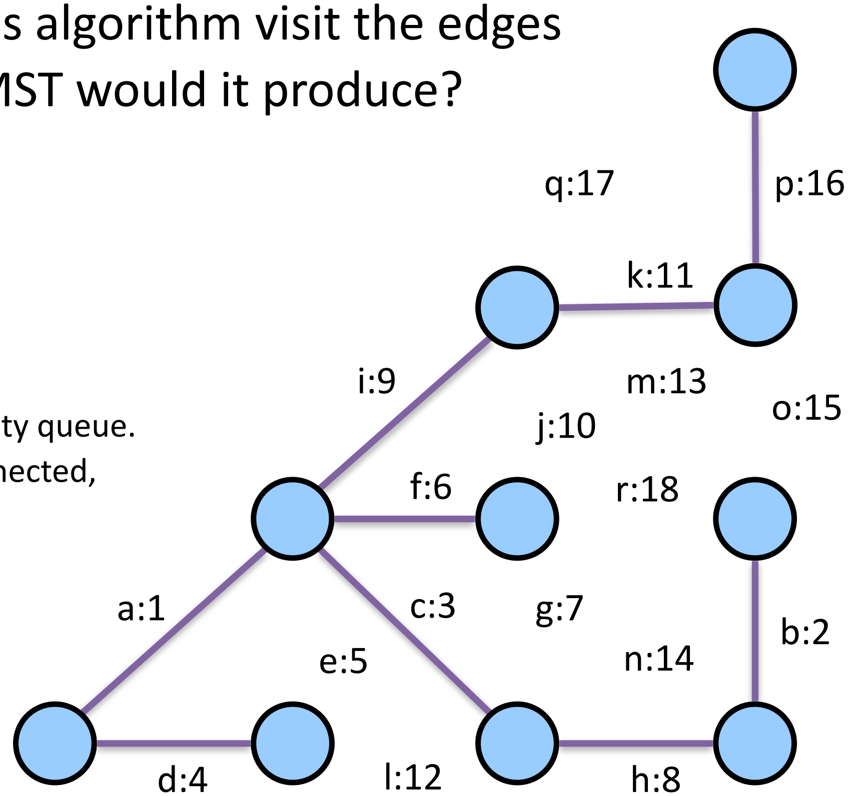


pq = {**r:18**}

Kruskal Example

- In what order would Kruskal's algorithm visit the edges in the graph below? What MST would it produce?

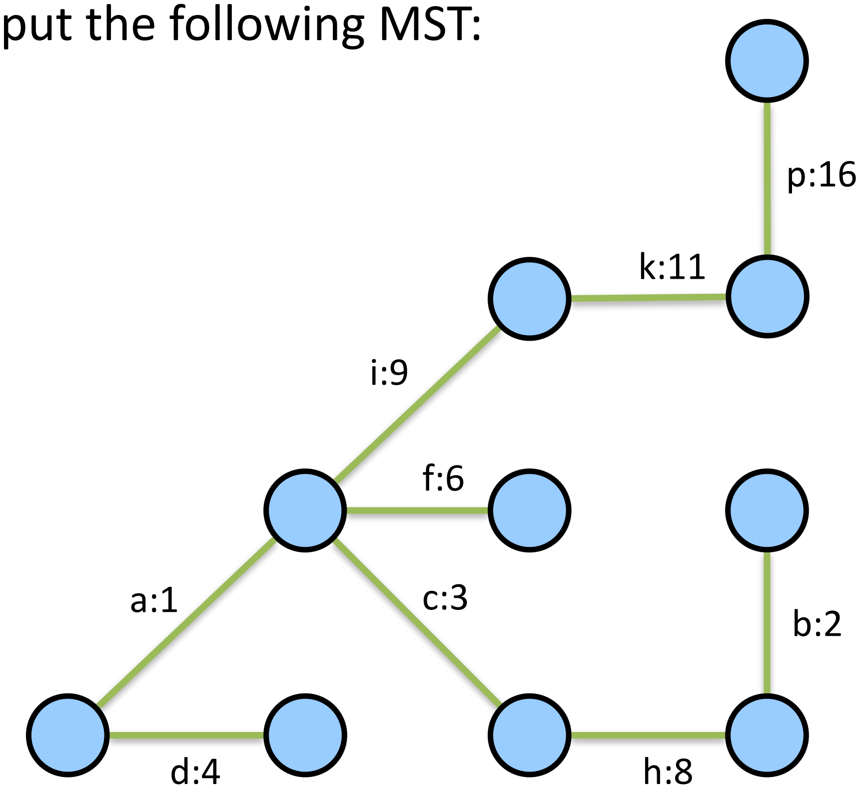
function **kruskal**(graph):
  Remove all edges from the graph.
  Place all edges into a priority queue
    based on their weight (cost).
  While the priority queue is not empty:
    Dequeue an edge *e* from the priority queue.
    If *e*'s endpoints aren't already connected,
      add that edge into the graph.
    Otherwise, skip the edge.



pq = {}

Kruskal Example

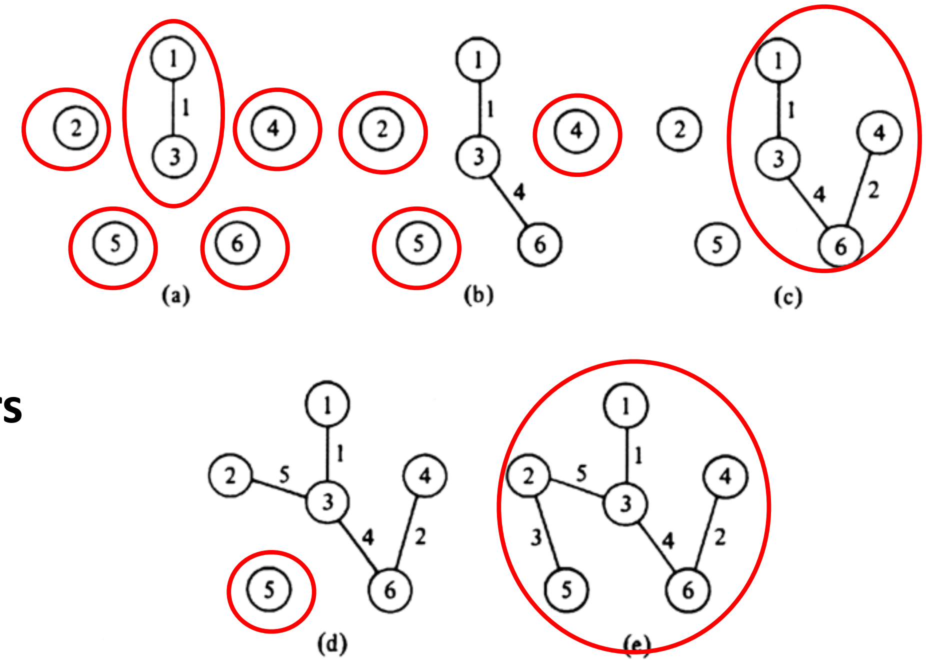
- Kruskal's algorithm would output the following MST:
  - {a, b, c, d, f, h, i, k, p}
- The MST's total cost is:
  1+2+3+4+6+8+9+11+16 = 60



- What data structures should we use to implement this algorithm?

```
function kruskal(graph):
 Remove all edges from the graph.
 Place all edges into a priority queue
     based on their weight (cost).
 While the priority queue is not empty:
         Dequeue an edge e from the priority queue.
         If e's endpoints aren't already connected,
            add that edge into the graph.
         Otherwise, skip the edge.
```

- Need some way to identify which vertexes are "connected" to which other ones
  - we call these "**clusters**" of vertices
- Also need an efficient way to figure out which cluster a given vertex is in.
- Also need to **merge clusters** when adding an edge.



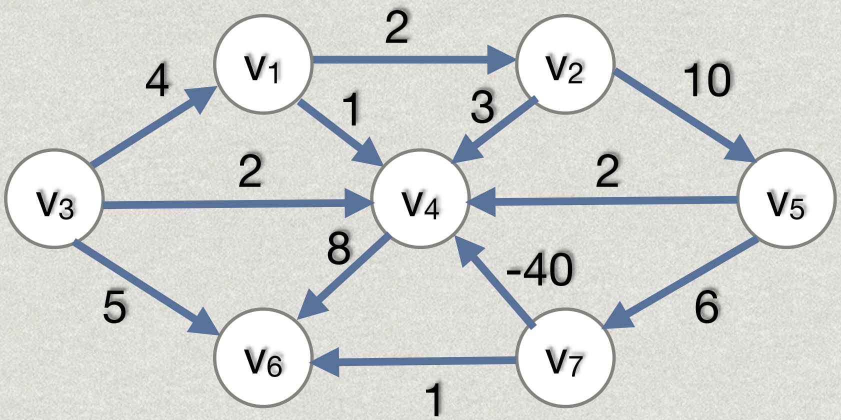
References and Advanced Reading

- **References:**
  - A* Heuristics: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
  - Minimum Spanning Tree visualization: https://visualgo.net/mst
  - Kruskal's Algorithm: https://en.wikipedia.org/wiki/Kruskal's_algorithm

- **Advanced Reading:**
  - How Internet Routing works: https://web.stanford.edu/class/msande91si/www-spr04/readings/week1/InternetWhitepaper.htm
  - http://www.explainthatstuff.com/internet.html

Extra Slides

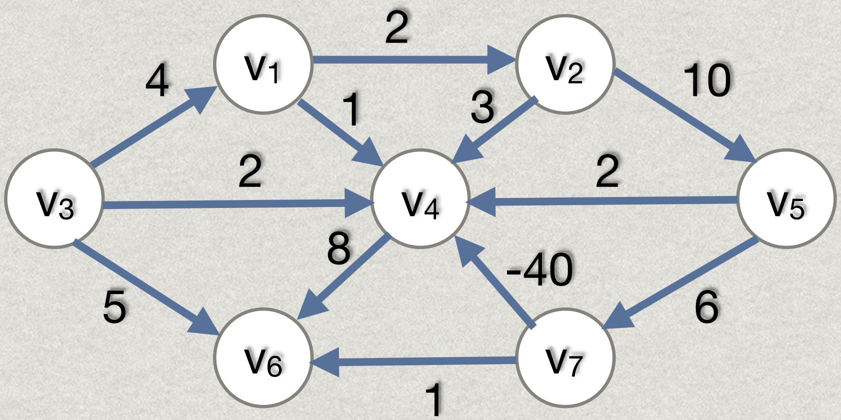
Dijkstra and negative edge costs.



Dijkstra fails with negative edge costs. Once a vertex is declared known (say,v_4), it is possible from some other unknown vertex to create a shorter path to the vertex (say, by eventually looking at $v_7 \rightarrow v_4$).

Is there an easy solution?

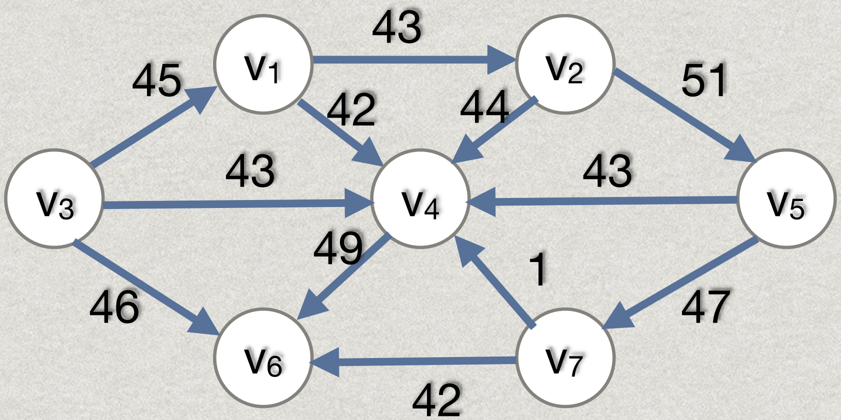
Dijkstra and negative edge costs.



Is there an easy solution?

A naïve approach might be to add a delta (in this case, 41) to all paths, and then apply Dijkstra's algorithm, but this fails because paths with many edges become more weighty than paths with few edges.

Dijkstra and negative edge costs.

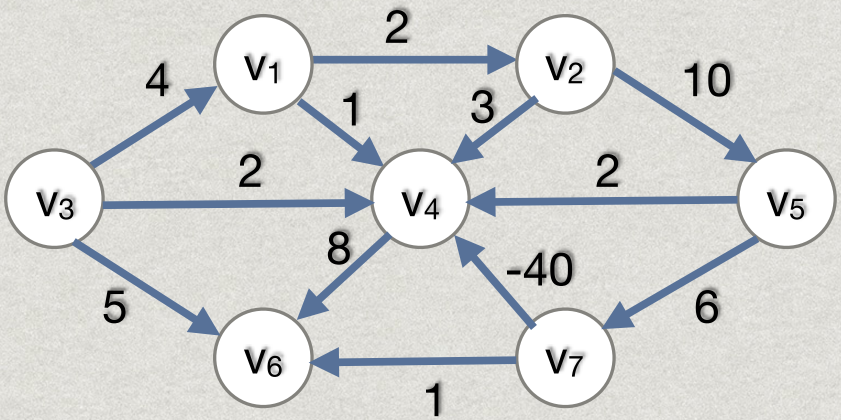


Is there an easy solution?

A naïve approach might be to add a delta (in this case, 41) to all paths, and then apply Dijkstra's algorithm, but this fails because paths with many edges become more weighty than paths with few edges.

[redacted]

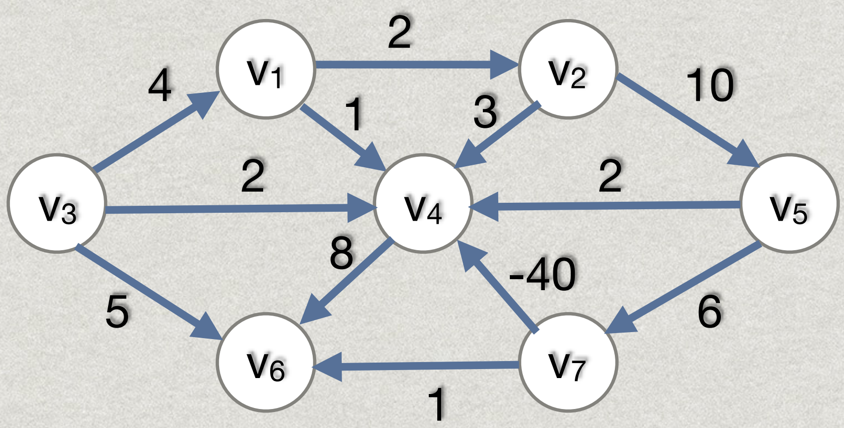
Dijkstra and negative edge costs.



So, there isn't a particularly easy solution. However, we can solve the problem with a combination of the weighted and unweighted algorithms, but at a drastically increased running time cost.

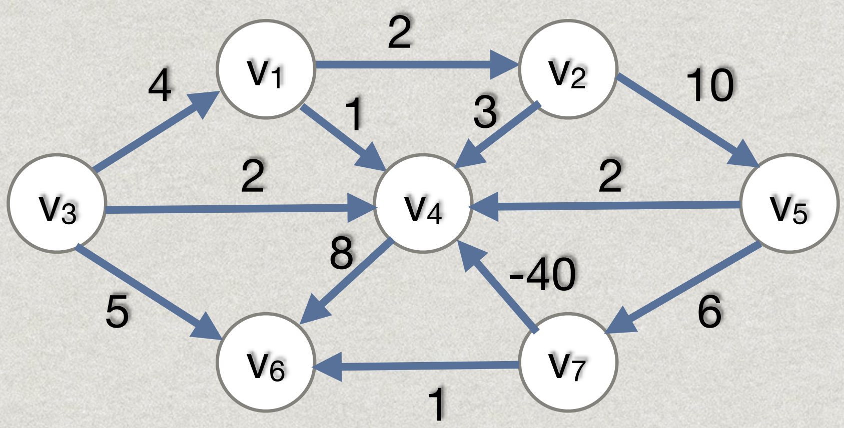
[redacted]

Dijkstra and negative edge costs.



We have to forget about the idea of "known" vertices, since we will have to be able to change our mind if necessary (the greedy algorithm doesn't work properly).

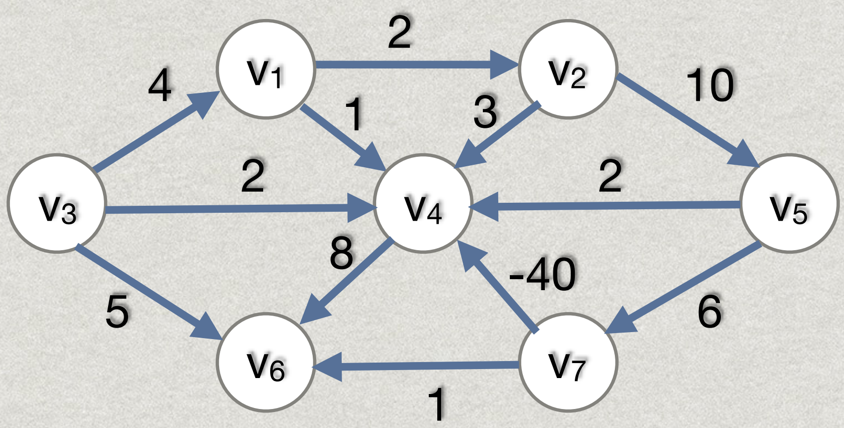
Dijkstra and negative edge costs.



Idea:

1. Place *s* on a queue
2. At each stage, dequeue a vertex, *v*. Then, find all vertices, *w*, adjacent to *v*, such that:
   $$d_w > d_v + cost_{v,w}$$
3. Update d_w and place *w* on the queue if it isn't already there (set a boolean to indicate presence in the queue)
4. Repeat until the queue is empty.

Dijkstra and negative edge costs.



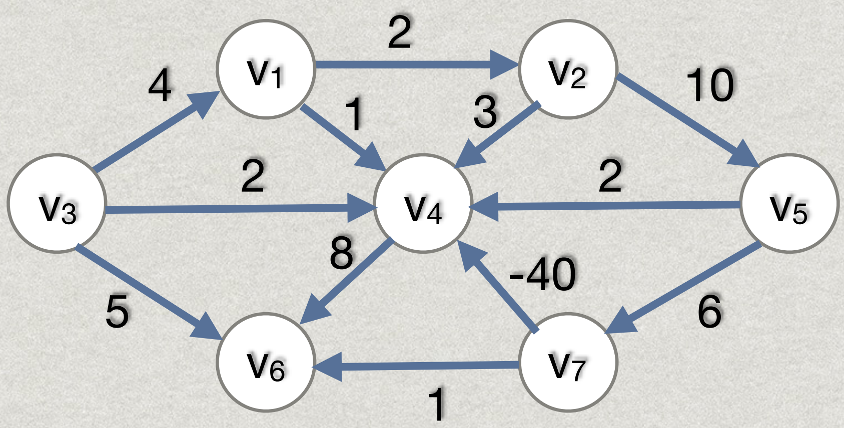
v	in queue?	d_v	p_v
v_1	Yes	0	0
v_2	No	INF	0
v_3	No	INF	0
v_4	No	INF	0
v_5	No	INF	0
v_6	No	INF	0
v_7	No	INF	0

queue

v_1					

Dequeue v_1 and check weights for v_2 and v_4. Update and place in queue.

Dijkstra and negative edge costs.



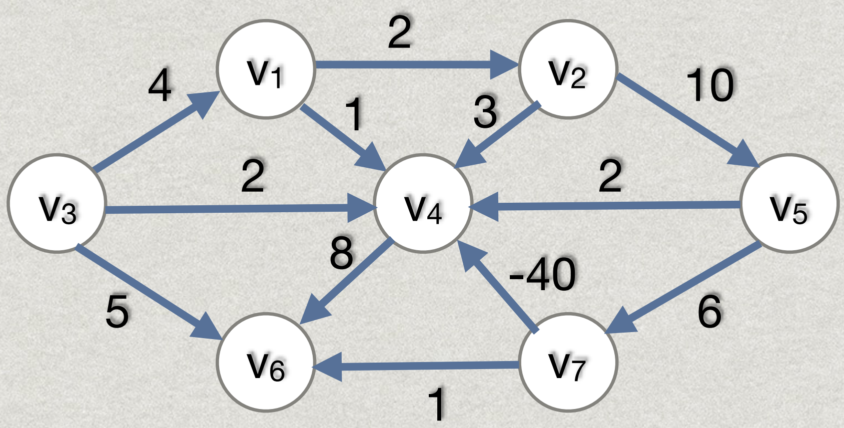
v	in queue?	d_v	p_v
v_1	No	0	0
v_2	Yes	2	v_1
v_3	No	INF	0
v_4	Yes	1	v_1
v_5	No	INF	0
v_6	No	INF	0
v_7	No	INF	0

queue

v_2	v_4				

Dequeue v_2 and check weights for v_4 and v_5. Update and place in queue.

Dijkstra and negative edge costs.



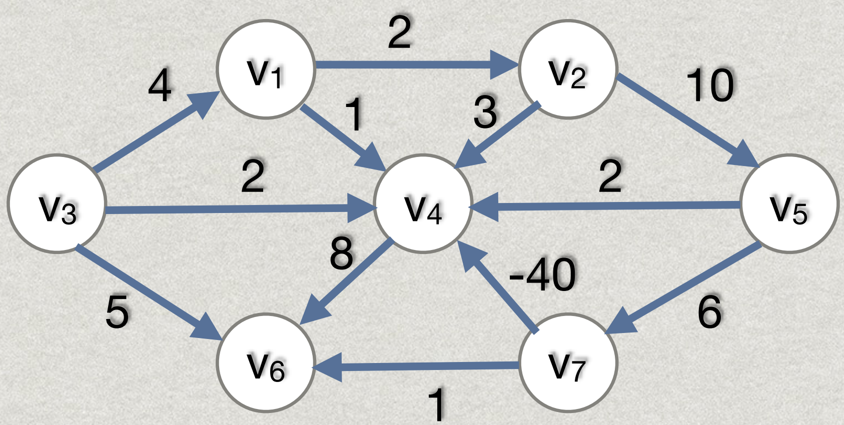
v	in queue?	d_v	p_v
v_1	No	0	0
v_2	No	2	v_1
v_3	No	INF	0
v_4	Yes	1	v_1
v_5	Yes	12	v_2
v_6	No	INF	0
v_7	No	INF	0

queue

v_4	v_5				

Dequeue v_4 and check weight for v_6. Update and place in queue.

Dijkstra and negative edge costs.



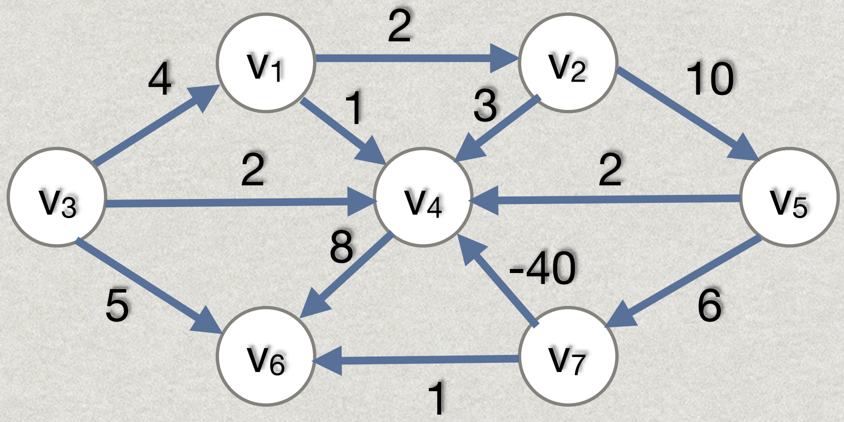
v	in queue?	d_v	p_v
v_1	No	0	0
v_2	No	2	v_1
v_3	No	INF	0
v_4	No	1	v_1
v_5	Yes	12	v_2
v_6	Yes	9	v_4
v_7	No	INF	0

queue

v_5	v_6				

Dequeue v_5 and check weights for v_4 and v_7. Update and place in queue.

Dijkstra and negative edge costs.



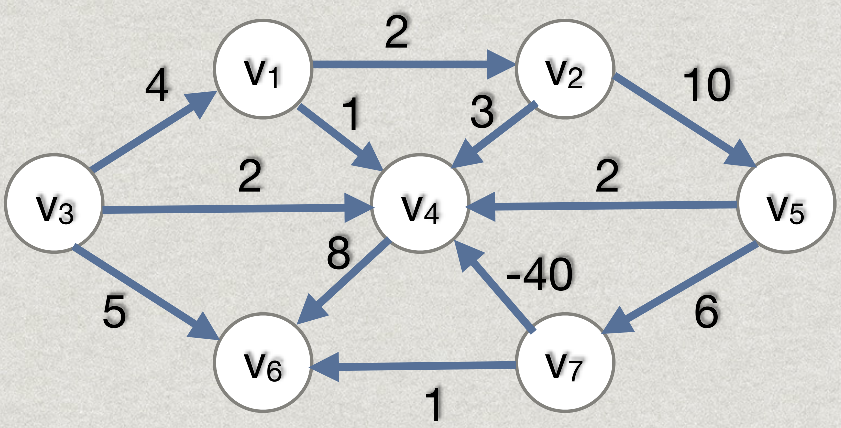
v	in queue?	d_v	p_v
v_1	No	0	0
v_2	No	2	v_1
v_3	No	INF	0
v_4	No	1	v_1
v_5	No	12	v_2
v_6	Yes	9	v_4
v_7	Yes	18	v_5

queue

v_6	v_7				

Dequeue v_6 and there aren't any weights to check (v_6 doesn't have any out-going edges).

Dijkstra and negative edge costs.



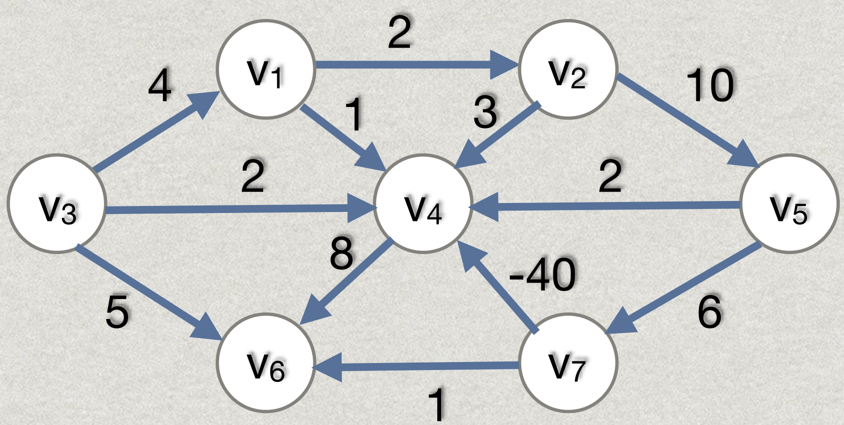
v	in queue?	d_v	p_v
v_1	No	0	0
v_2	No	2	v_1
v_3	No	INF	0
v_4	No	1	v_7
v_5	No	12	v_2
v_6	No	9	v_4
v_7	Yes	18	v_5

queue

v_7					

Dequeue v_7 and check weights for v_4 and v_6. v_4 will get updated and placed back in the queue.

Dijkstra and negative edge costs.



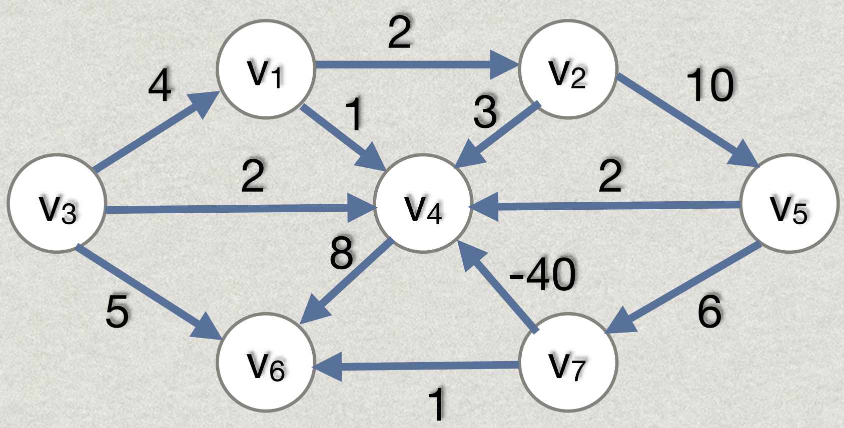
v	in queue?	d_v	p_v
v_1	No	0	0
v_2	No	2	v_1
v_3	No	INF	0
v_4	Yes	-22	v_7
v_5	No	12	v_2
v_6	No	9	v_4
v_7	No	18	v_5

queue

v_4					

Dequeue v_4 and check weight for v_6. v_6 will get updated and placed back in the queue.

Dijkstra and negative edge costs.



v	in queue?	d_v	p_v
v_1	No	0	0
v_2	No	2	v_1
v_3	No	INF	0
v_4	No	-22	v_7
v_5	No	12	v_2
v_6	Yes	-14	v_4
v_7	No	18	v_5

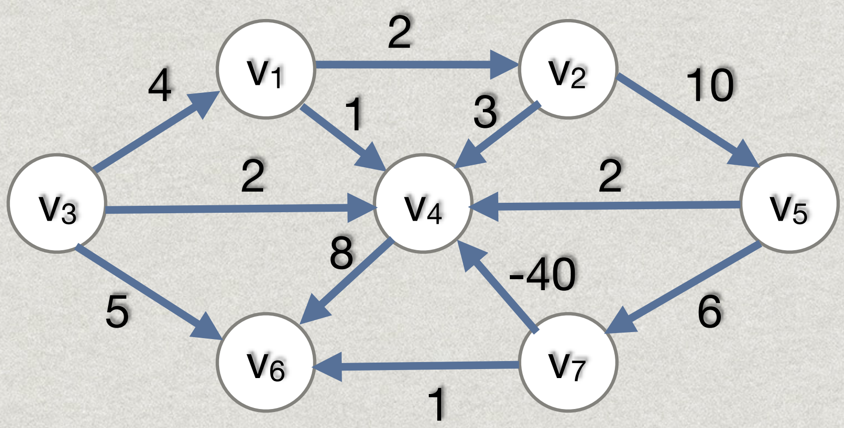
queue

v_6					

Finally, Dequeue v_6. There aren't any vertices to check (no out-going edges from v_6), and that will empty the queue.

[redacted]

Dijkstra and negative edge costs.

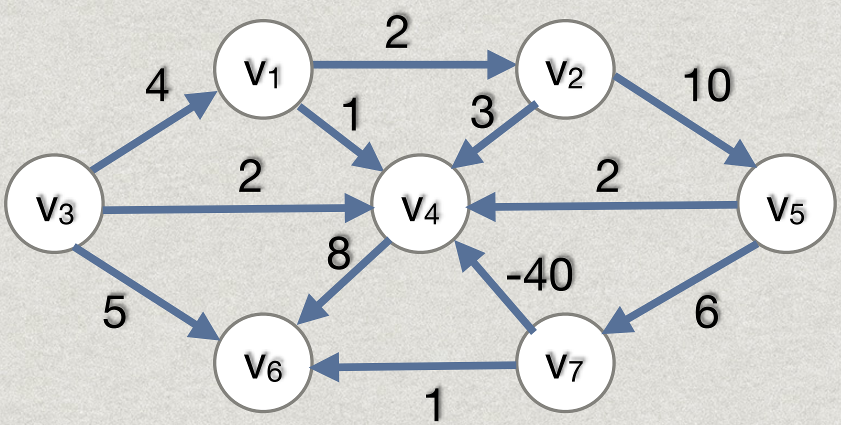


v	in queue?	d_v	p_v
v_1	No	0	0
v_2	No	2	v_1
v_3	No	INF	0
v_4	No	-22	v_7
v_5	No	12	v_2
v_6	No	-14	v_4
v_7	No	18	v_5

queue

Finally, Dequeue v_6. There aren't any vertices to check (no out-going edges from v_6), and that will empty the queue.

Dijkstra and negative edge costs.

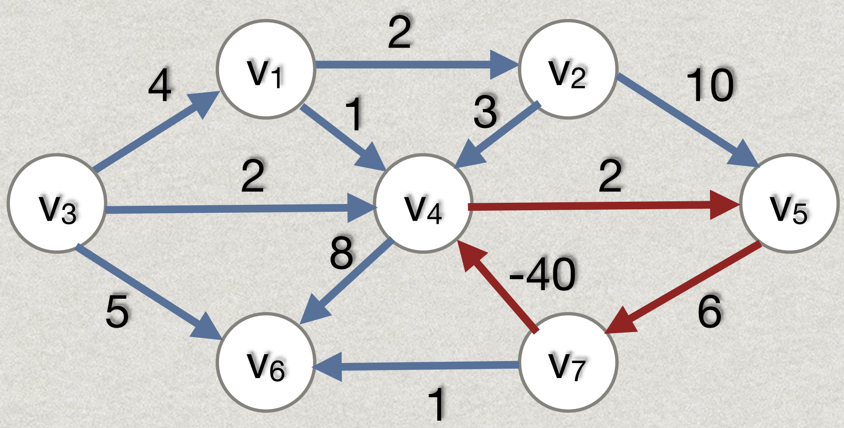


v	in queue?	d_v	p_v
v_1	No	0	0
v_2	No	2	v_1
v_3	No	INF	0
v_4	No	-22	v_7
v_5	No	12	v_2
v_6	No	-14	v_4
v_7	No	18	v_5

Running time?

Each vertex can be dequeued at most |V| times per edge, meaning that the running time is now O(|E| * |V|), which is significantly worse than for the algorithm without negative costs.

Dijkstra and negative edge costs.



What about negative cost cycles?

This will run our queue indefinitely, so we need to make a decision about when to stop.

If you stop after every vertex has been dequeued |V|+1 times, you will guarantee termination.