COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 16: Weighted Graphs, Dijkstra's, A*

Lecturer: Chris Lucas (cflucas@ncat.edu)

Office Hours

Today only -> 3:30pm-4:00pm

HW5 Due!

Tuesday 10/25 @ 11:59PM

HW5 Due!

Q2 (Alphabet Reconstruction) is optional!

HW4

Recall where we ended last lecture…

Analogy

BFS

DFS

Breadth/Depth-First Search Pseudocode

```
algorithm BFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

```
algorithm DFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Stack of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

https://visualgo.net/en/dfsbfs

Depth-First Search Pseudocode

```
algorithm DFS
 Input: undirected graph G, int s and int t
 Output: whether or not there's a path from s to t

 visited = new boolean array of size |V|
 return dfsHelper(G, s, t, visited)


algorithm DFSHelper
 Input: undirected graph G = (V, E), s, t, and
visited
 Output: whether or not there's a path from s to t

 visited[s] = true
 if s == t
   return true
 for each neighbor of s, starting from smallest
labeled neighbor
   if !visited[neighbor] and dfsHelper(G,
neighbor, t, visited)
     return true
 return false
```

Time Complexity? O(V + E)
Space Complexity? O(V)

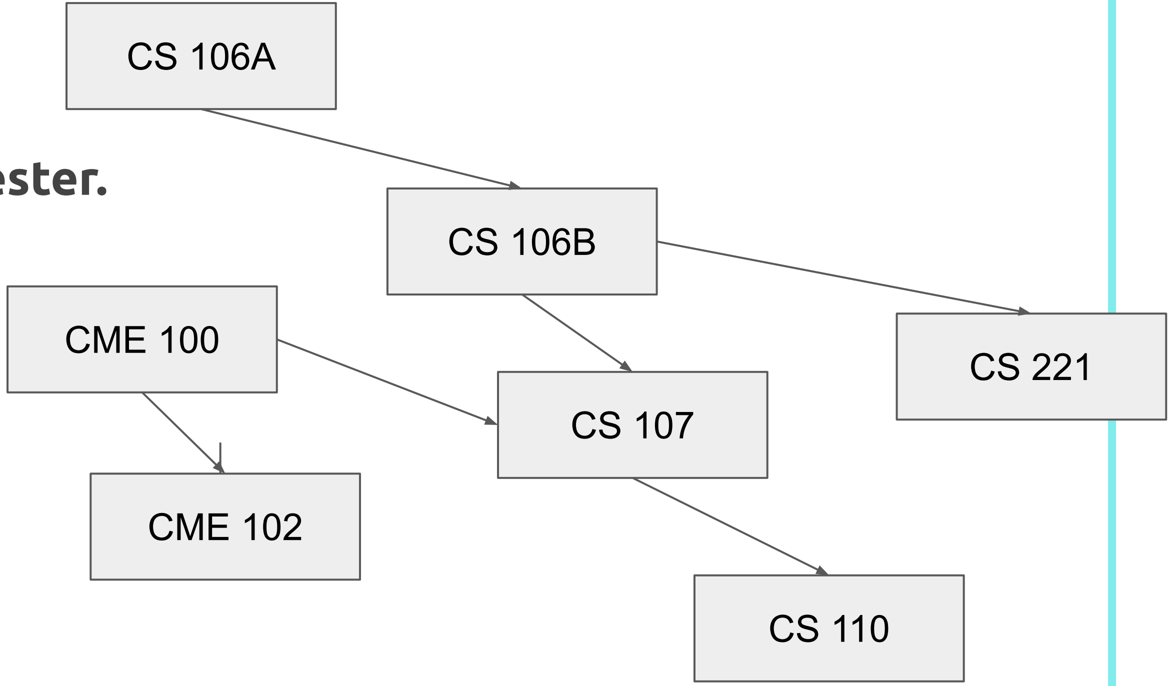
```
algorithm DFS
   Input: undirected graph G = (V,E), s and d
  Output: true/false if path from s to d

    frontier = Stack of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Time Complexity? O(V+E)
Space Complexity? O(V)

Topological Sort Example #1

What's one valid course ordering?
Assume 1 course per semester.

CME 100
CME 102
CS 106A
CS 106B
CS 107
CS 221
CS 110



Big Questions!

- How do we sort DAGs?
- What's a weighted graph?
- How can we traverse weighted graphs?

Big Questions!

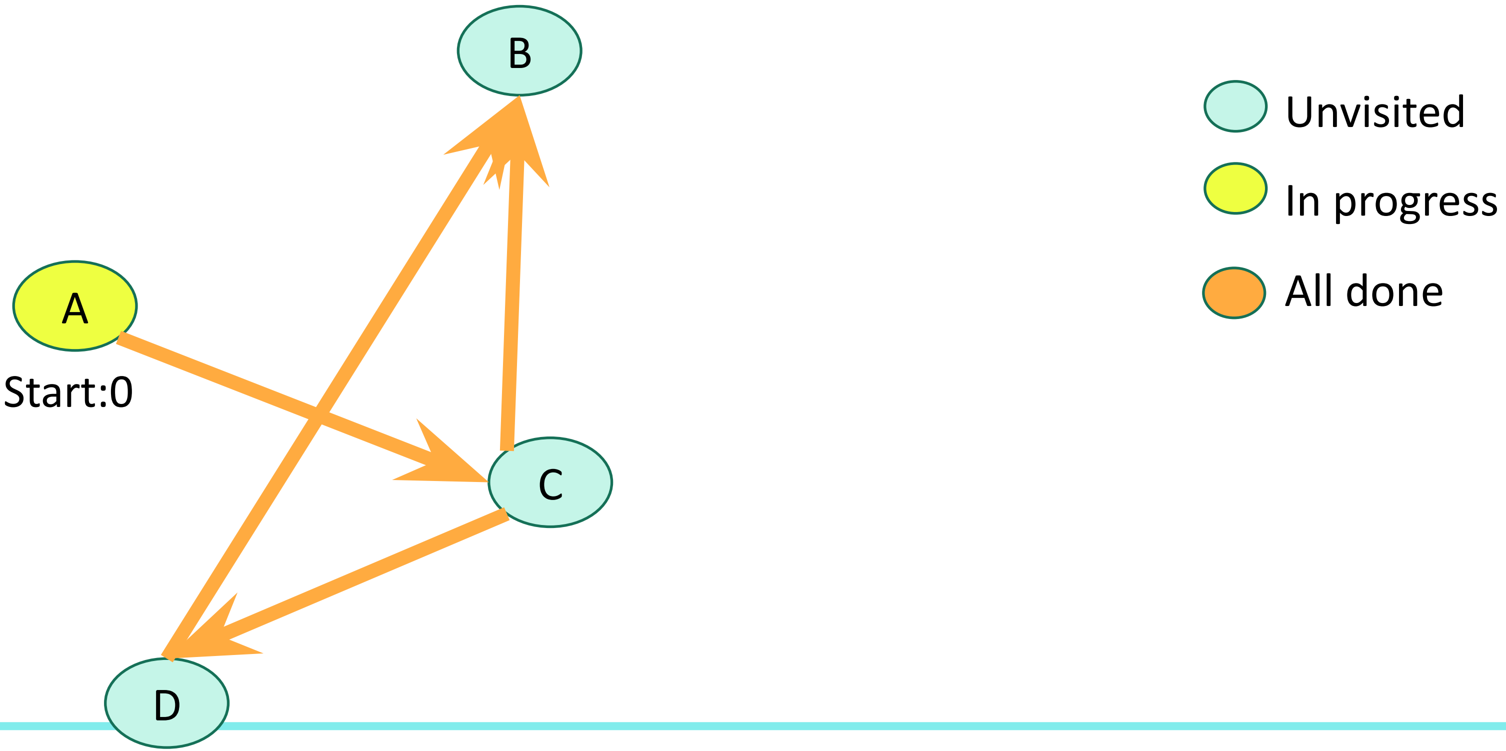
- How do we sort DAGs?
- What's a weighted graph?
- How can we traverse weighted graphs?

Topological Sort Algorithm

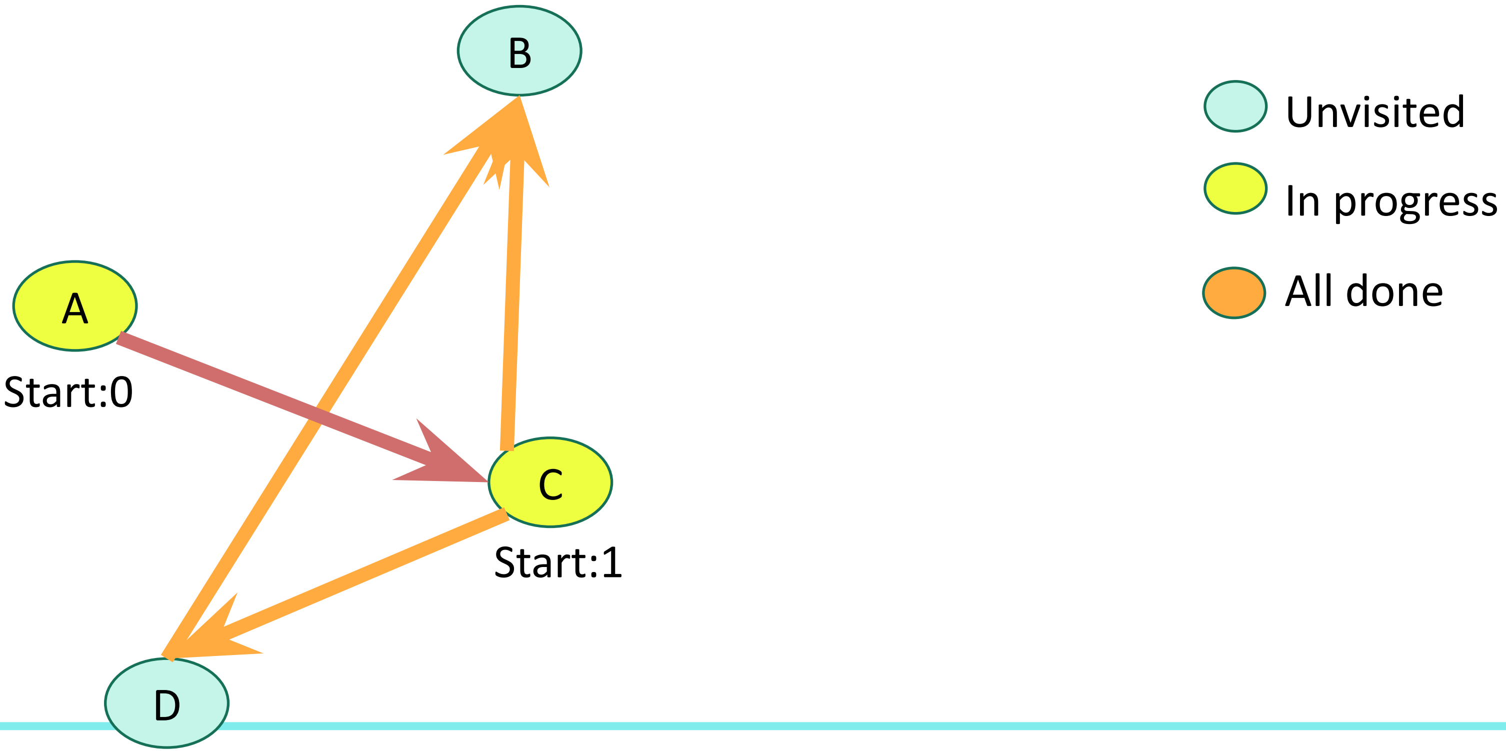
- Note that some valid orderings laid out each “child” until we reached a sink.

- We can actually perform a DFS, starting at source nodes and **reverse the DFS finishing times** to get a valid topological sort.

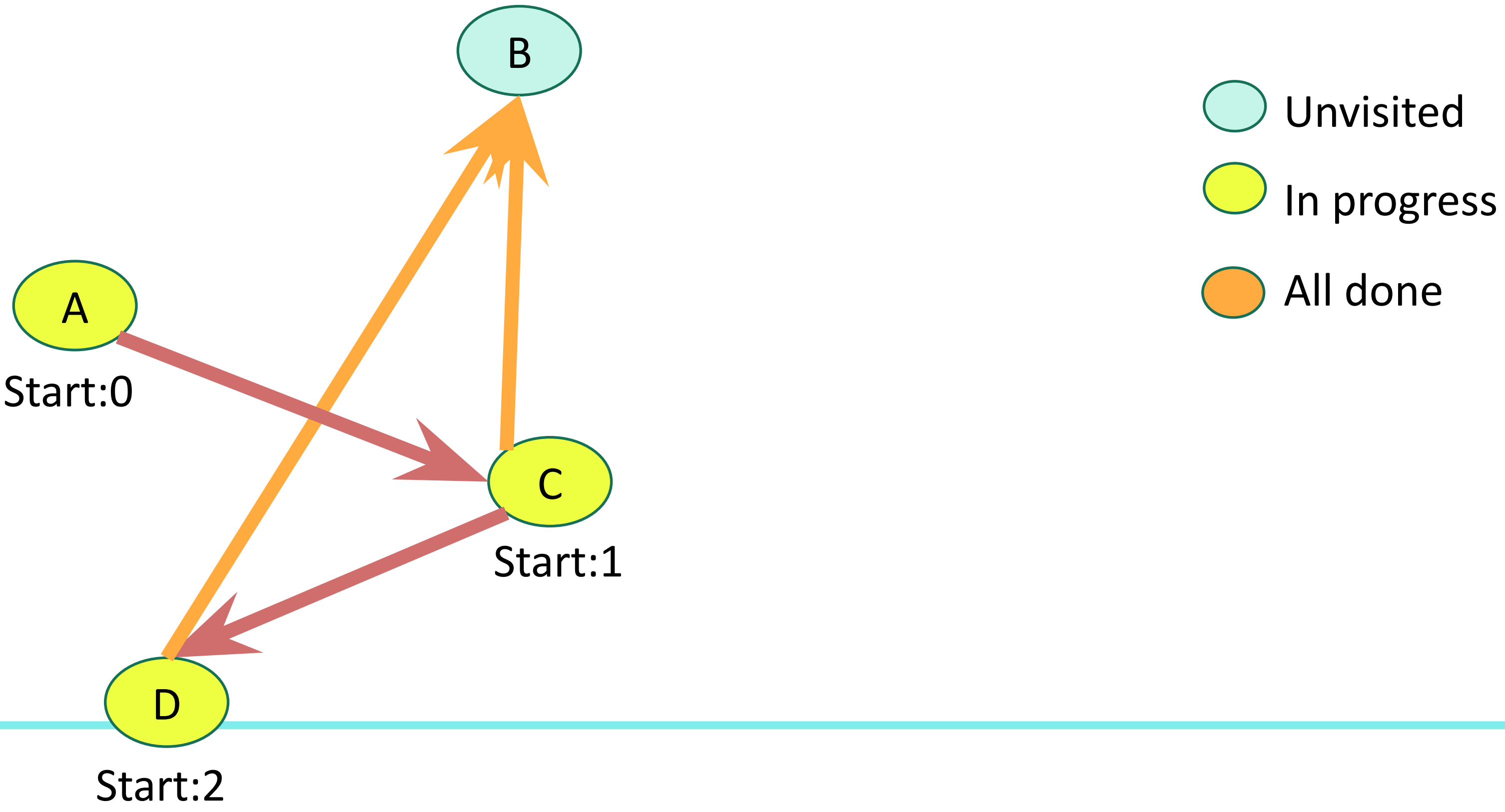
TopoSort Example



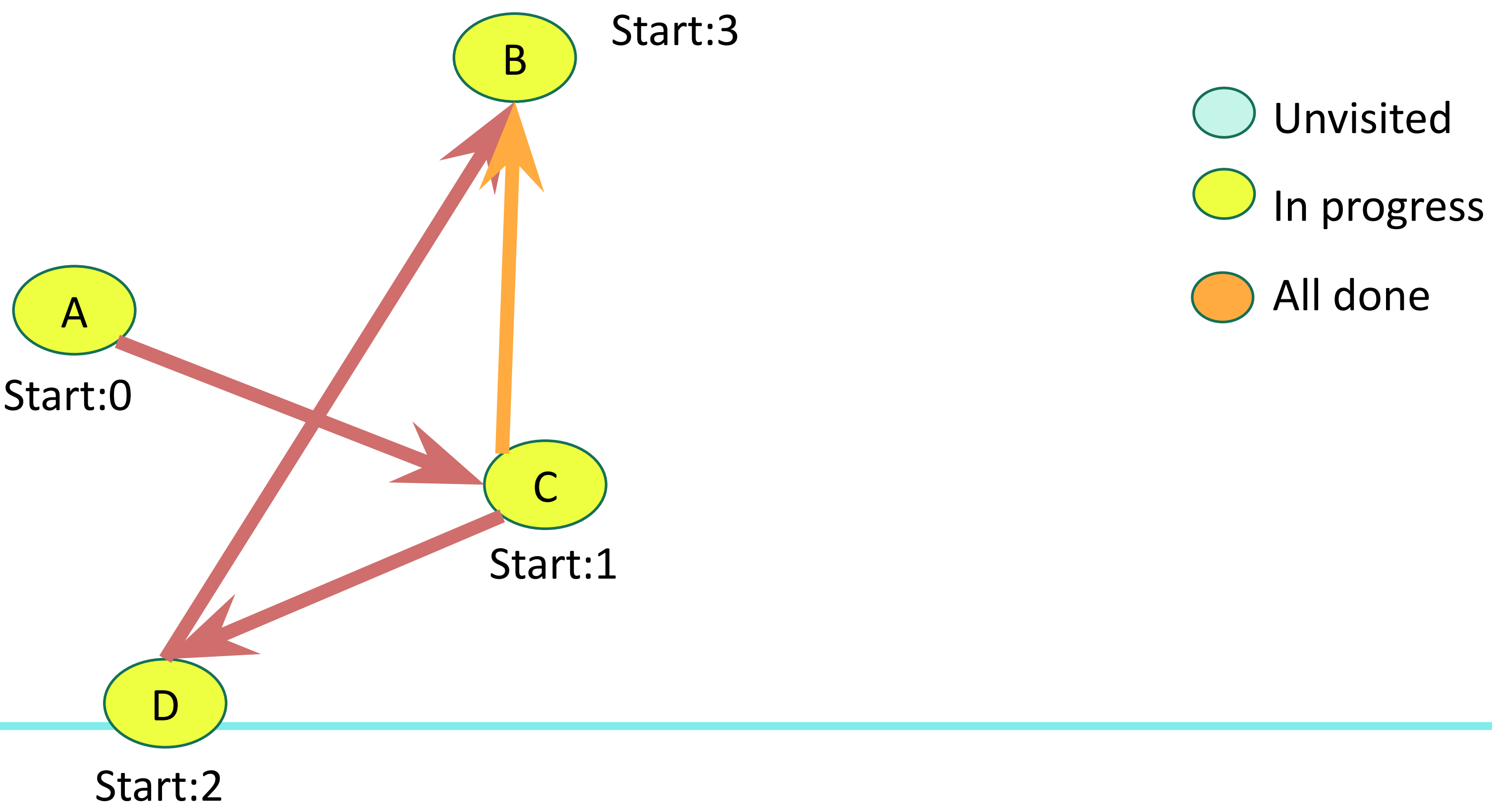
TopoSort Example



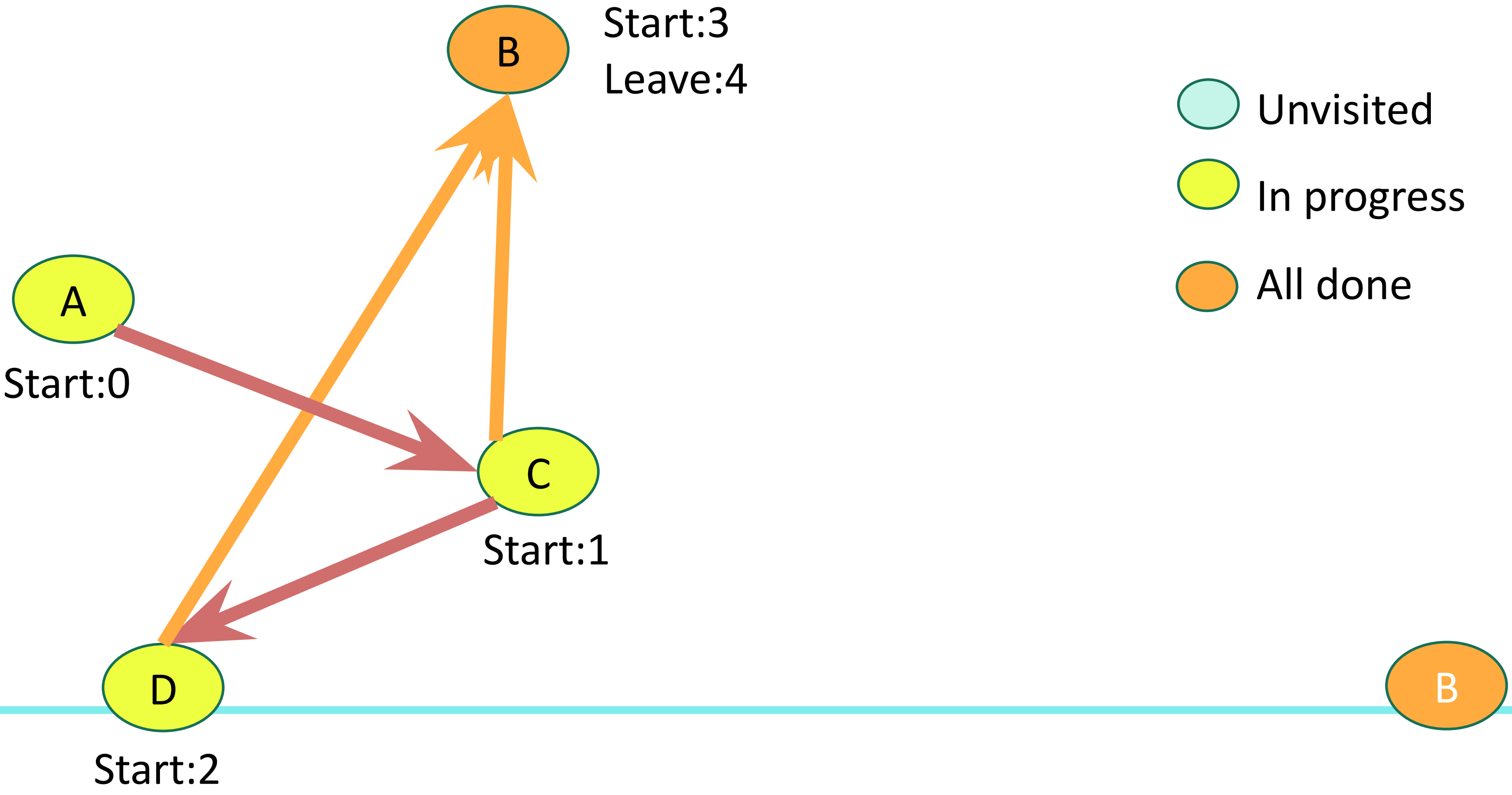
TopoSort Example



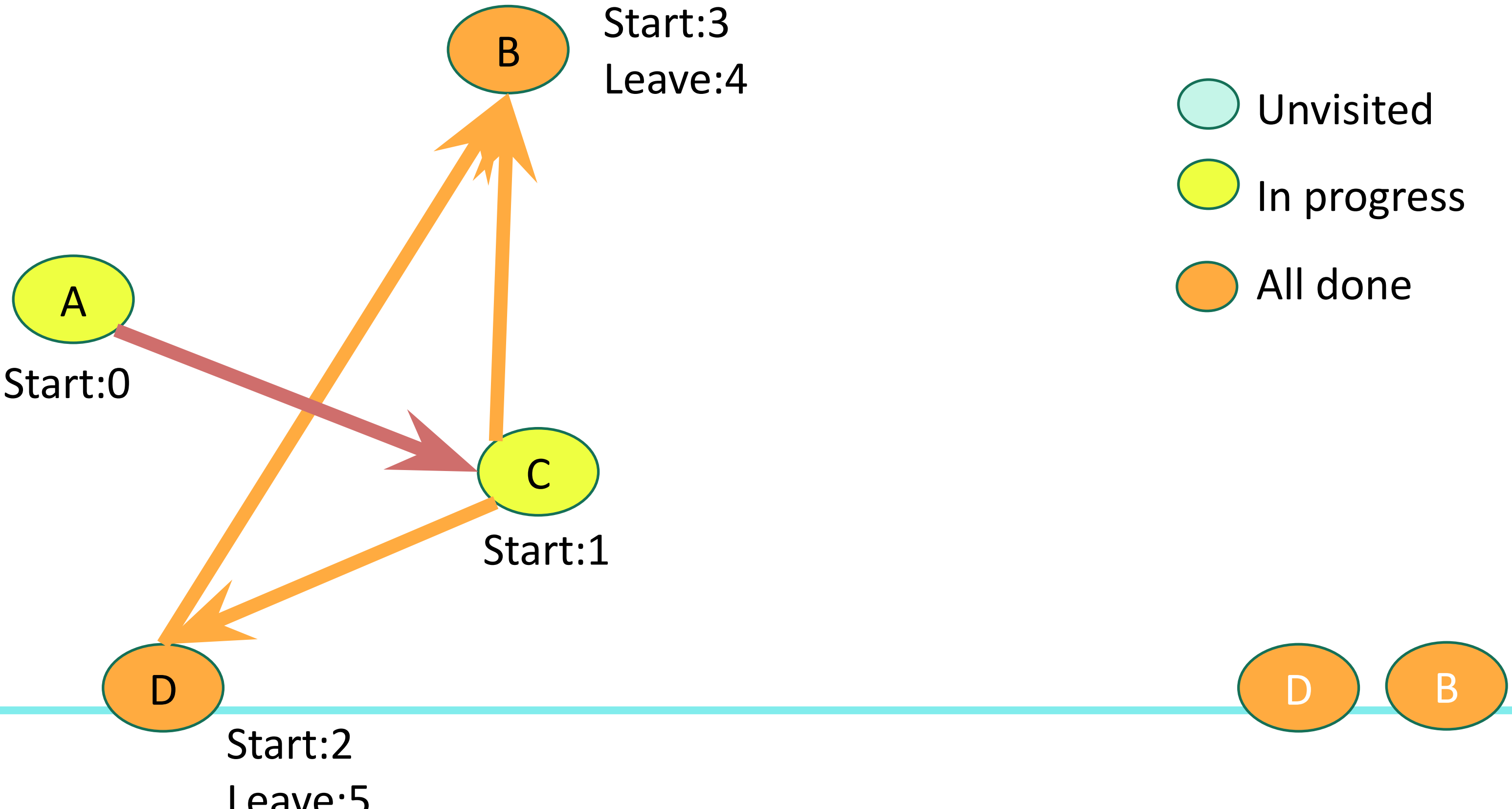
TopoSort Example



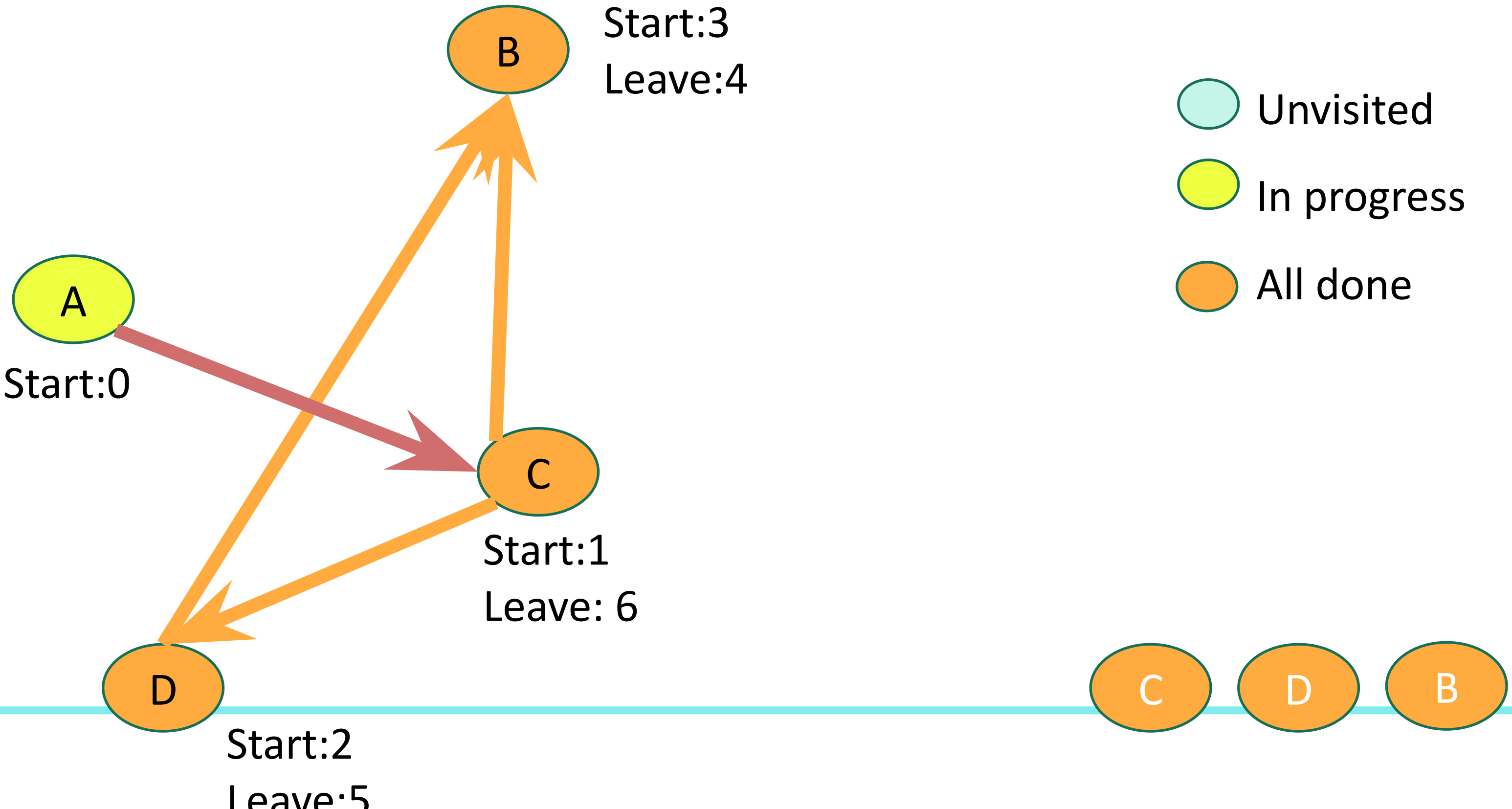
TopoSort Example



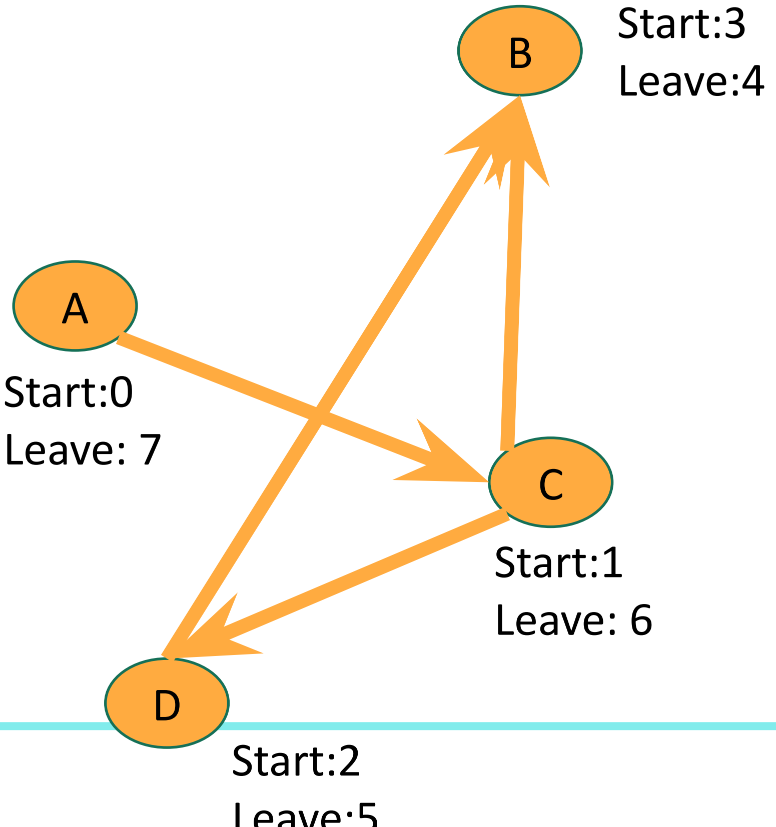
TopoSort Example



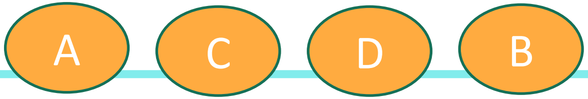
TopoSort Example



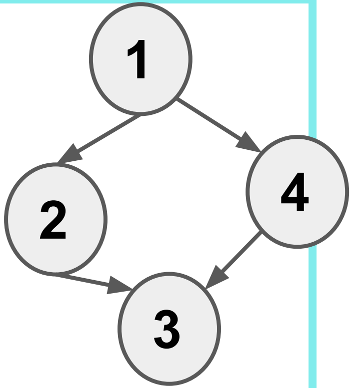
TopoSort Example



Do them in this order:

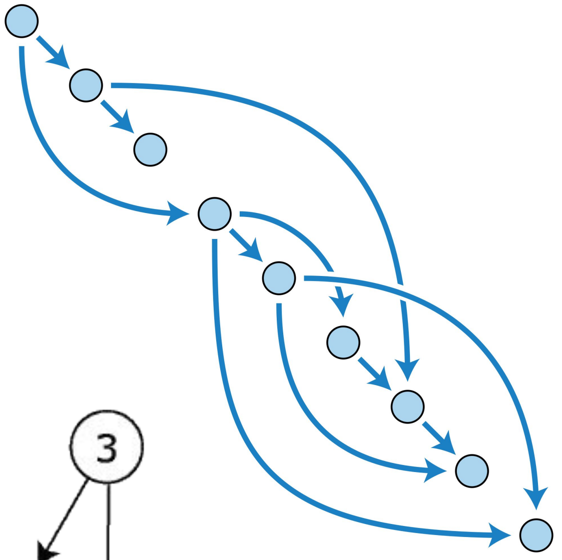


TopoSort Pseudocode



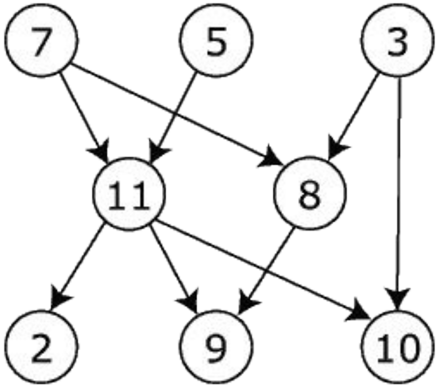

```
algorithm TopoSort
 Input: undirected graph G, int s and int t
 Output: whether or not there's a path from s to t

 visited = new boolean array of size |V|
 current_label = |V| // global variable
 For v in G:
   if !visited[v]:
     return topoSortHelper(G, v, visited)

algorithm topoSortHelper
 Input: undirected graph G, v, and visited
 visited[v] = true
 for each neighbor of v:
   if !visited[neighbor]:
      topoSortHelper(G, neighbor, visited)
   v.end = current_label
   current_label -= 1
```

TopoSort Pseudocode



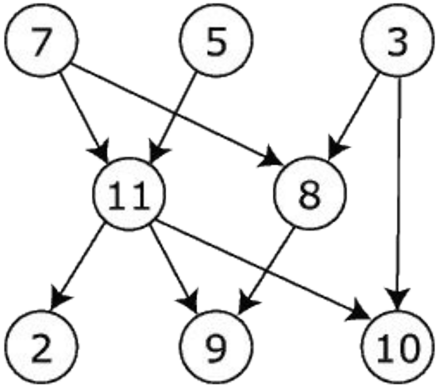
```
algorithm TopoSort
 Input: undirected graph G, int s and int t
 Output: whether or not there’s a path from s to t

 visited = new boolean array of size |V|
 current_label = |V| // global variable
 For v in G:
   if !visited[v]:
     return topoSortHelper(G, v, visited)
```

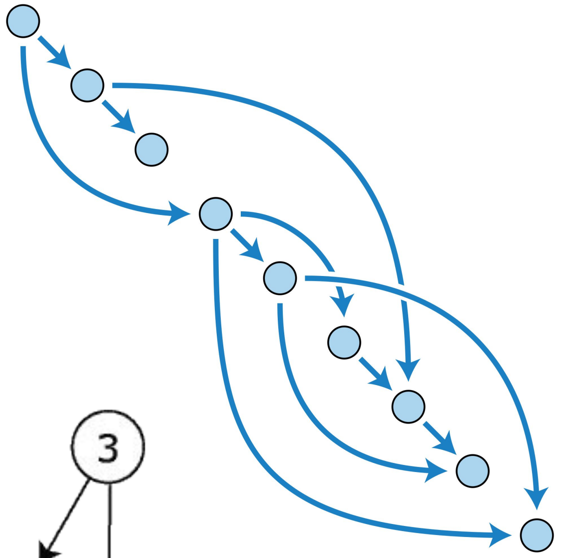
```
algorithm topoSortHelper
 Input: undirected graph G, v, and visited
 visited[v] = true
 for each neighbor of v:
   if !visited[neighbor]:
     topoSortHelper(G, neighbor, visited)
 v.end = current_label
 current_label -= 1
```



TopoSort Code

For implementation,
see [repl.it]





Big Questions!

- How do we sort DAGs?
- What’s a weighted graph?
- How can we traverse weighted graphs?

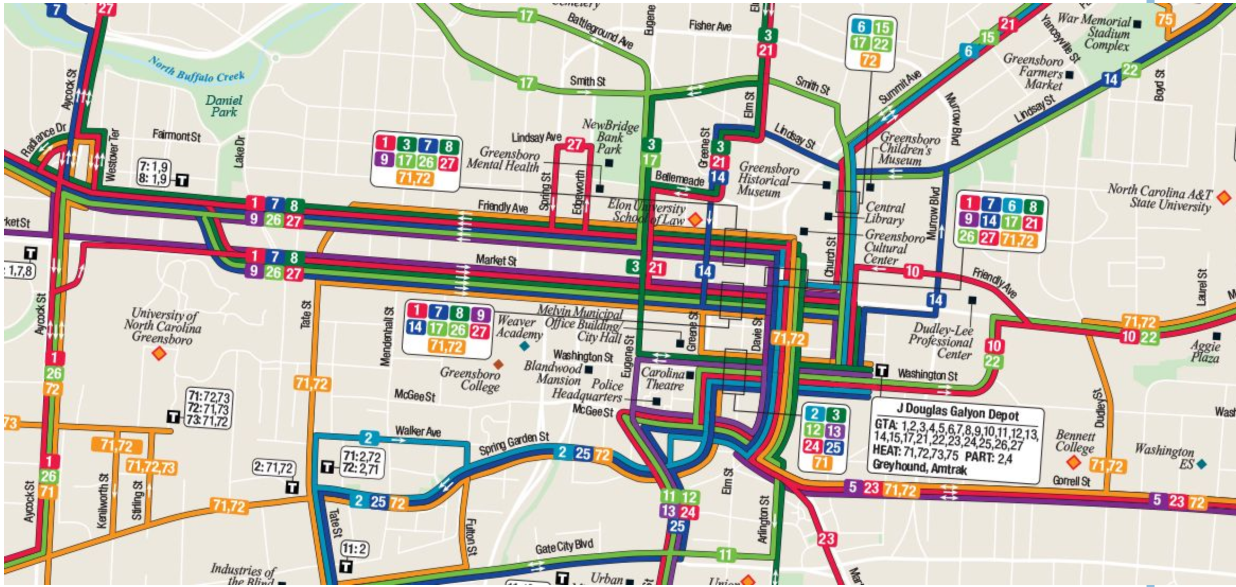
Single-source shortest-path problem

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

Destination	Cost	To get there
Clock Tower	1	Clock Tower
Health Center	2	Clock Tower-Health Center
Morrison Hall	10	Morrison Hall
COMP 285	17	COMP 285
Aggie Suites	6	Clock Tower-Health Center-Aggie Suites
Stadium	15	Stadium
Aggie Terrace	21	Clock Tower-Aggie Terrace

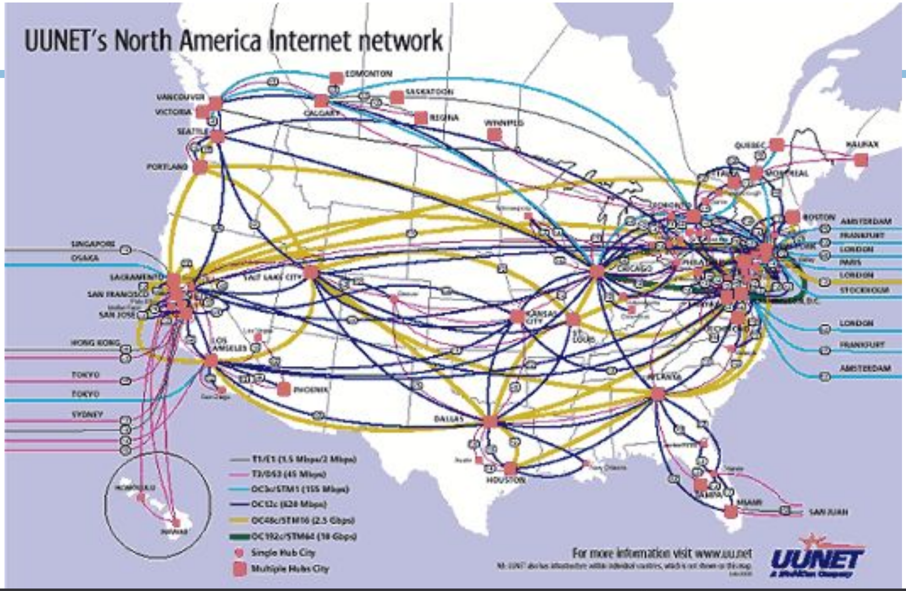
Example

- “what is the shortest path from NC A&T to [anywhere else]” using bus, bus, bike, walking, uber/lyft, driving
- Edge weights have something to do with time, money, hassle.



Example pt. 2

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




```
..rithms-course    traceroute -a www.ethz...    ..lopment/site2    ../hw-solutions
→  ~ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1  [AS0] 192.168.87.1 (192.168.87.1)  3.824 ms  2.489 ms  2.246 ms
 2  [AS0] 192.168.1.254 (192.168.1.254)  4.706 ms  2.512 ms  5.135 ms
 3  [AS0] 104.188.160.1 (104.188.160.1)  5.180 ms  4.191 ms  4.441 ms
 4  [AS0] 99.94.204.178 (99.94.204.178)  3.357 ms  3.007 ms  6.353 ms
 5  [AS7018] 12.242.111.38 (12.242.111.38)  16.523 ms  18.199 ms  14.701 ms
 6  [AS7018] attga402igs.ip.att.net (12.122.117.121)  14.785 ms  13.600 ms  14.220 ms
 7  [AS3356] 4.68.62.225 (4.68.62.225)  10.973 ms  13.979 ms  12.101 ms
 8  [AS3356] ae1.10.bar1.geneva1.level3.net (4.69.203.66)  107.572 ms  107.457 ms  113.640 ms
 9  [AS9057] dante.bar1.geneva1.level3.net (213.242.73.74)  108.367 ms  109.289 ms  109.347 ms
10  [AS559] swice4-b4.switch.ch (130.59.36.70)  109.503 ms  109.147 ms  108.886 ms
11  [AS559] swibf1-b2.switch.ch (130.59.36.113)  125.068 ms  125.046 ms  123.263 ms
12  [AS559] swiez3-b5.switch.ch (130.59.37.6)  121.484 ms  120.742 ms  117.935 ms
13  [AS559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1)  124.010 ms  124.115 ms  123.322
14  [AS559] rou-fw-rz-rz-gw.ethz.ch (192.33.92.169)  119.290 ms  121.185 ms *
15  * * *
```

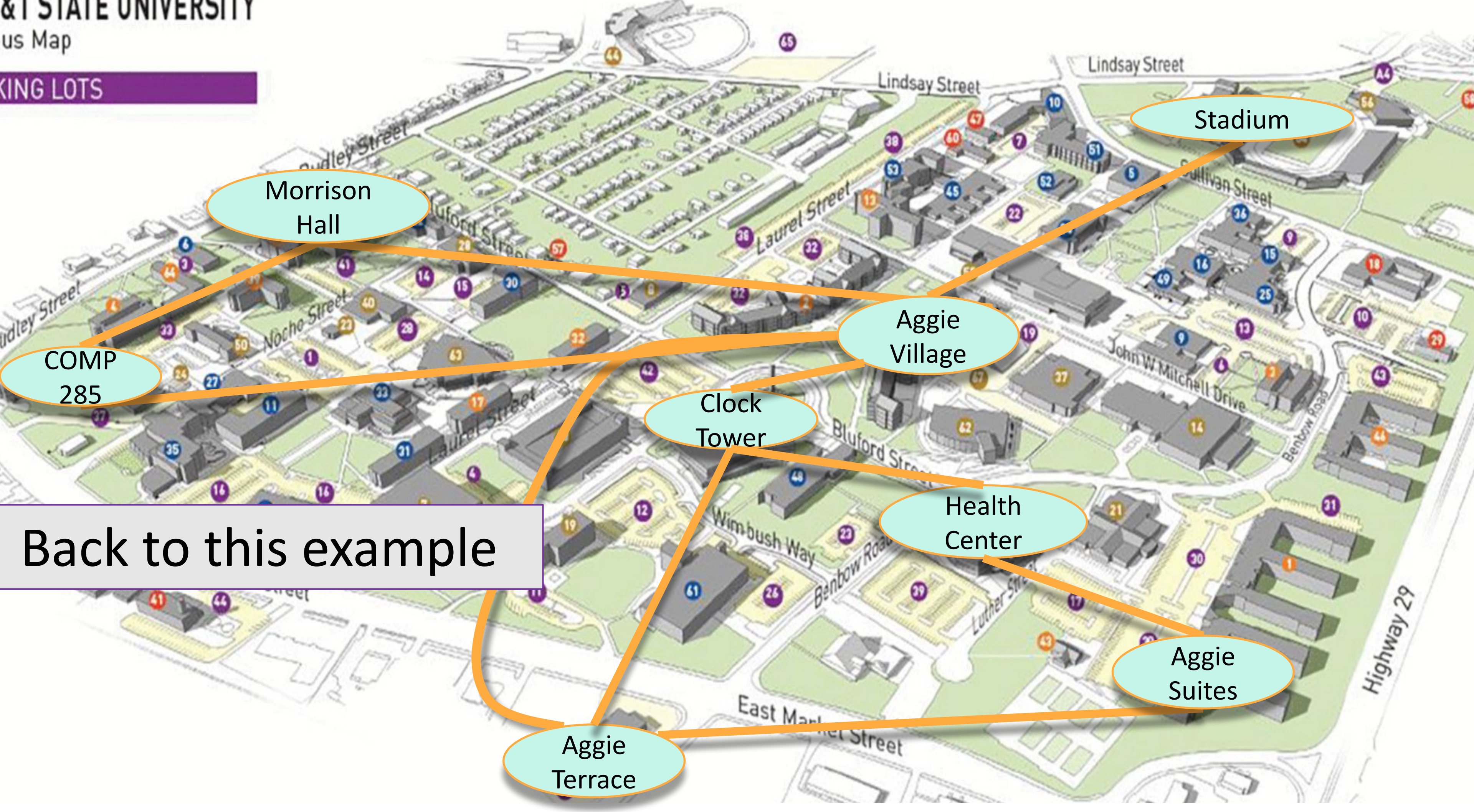
These are difficult, real-world problems

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

Big Questions!

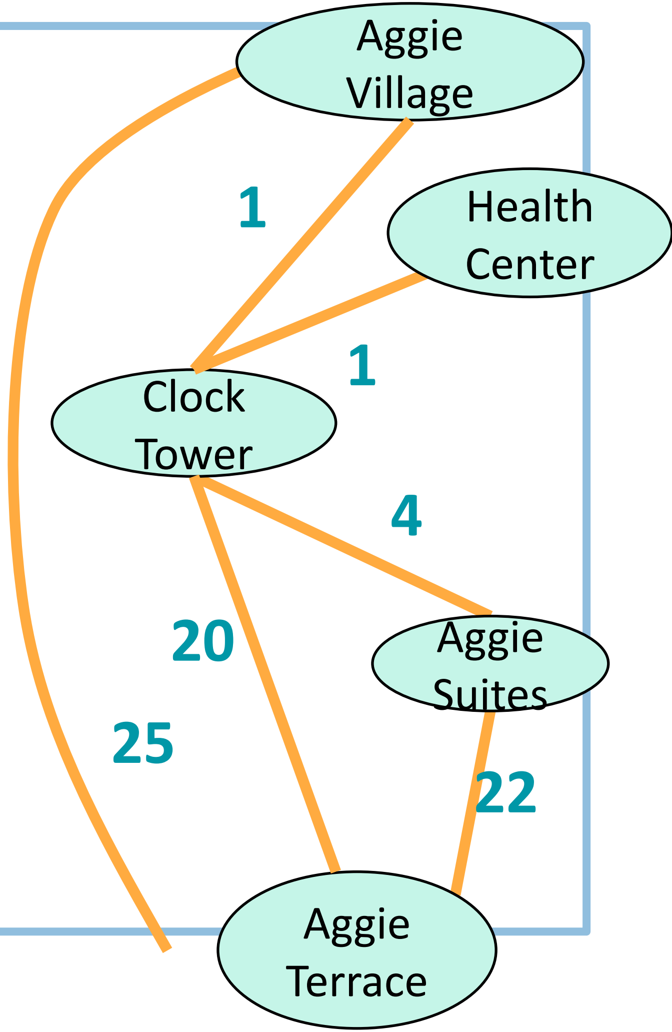
- How do we sort DAGs?
- What’s a weighted graph?
- How can we traverse weighted graphs?

Back to this example

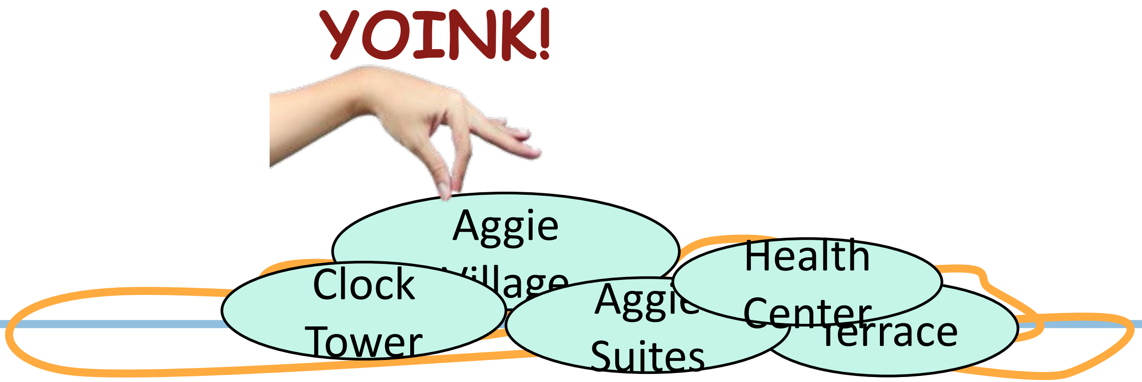


Dijkstra's algorithm

- Finds shortest paths from Aggie Village to everywhere else.

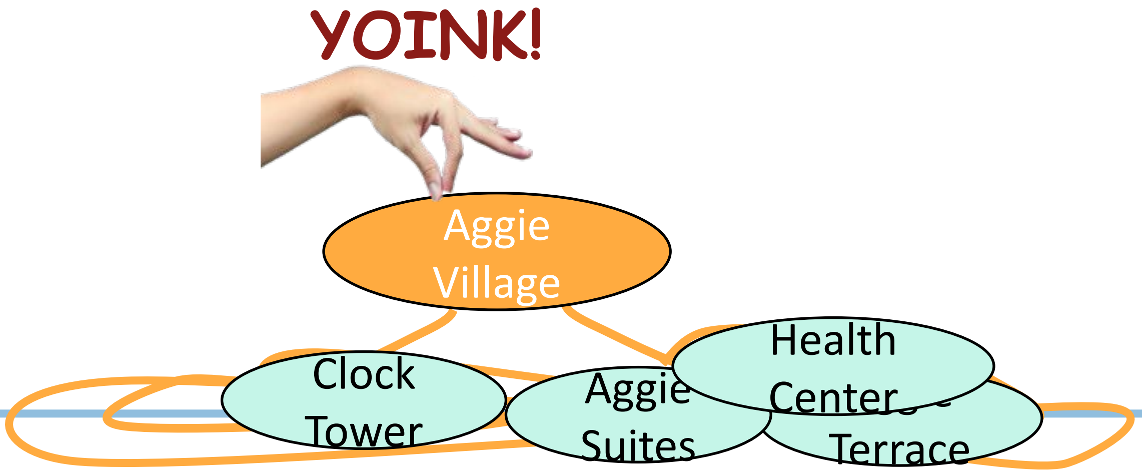


Dijkstra intuition

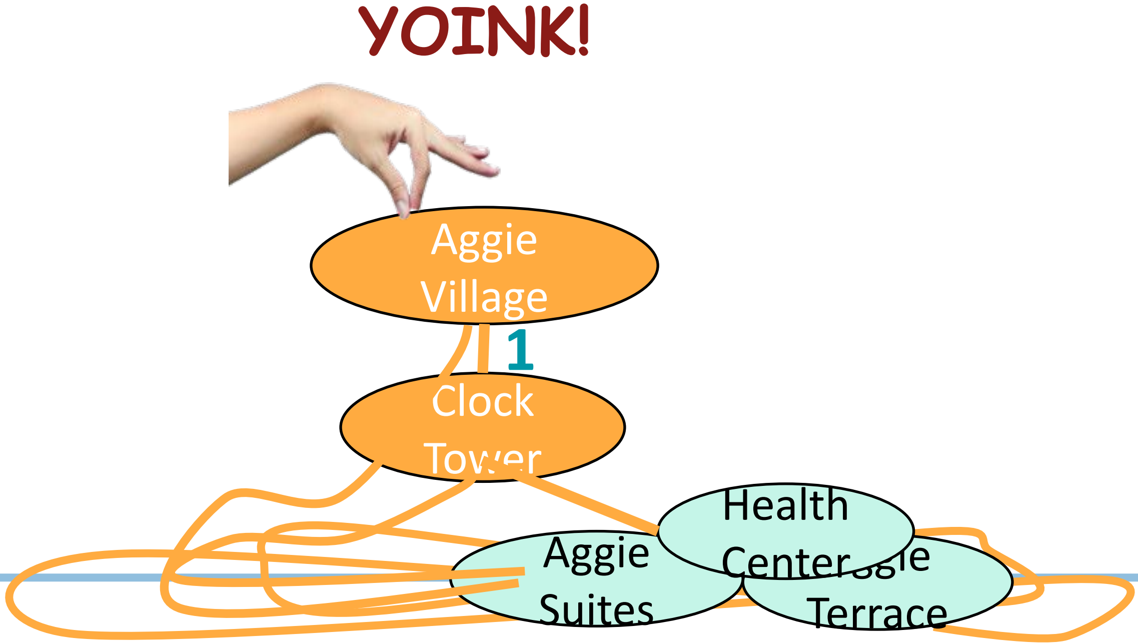


Dijkstra
intuition

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

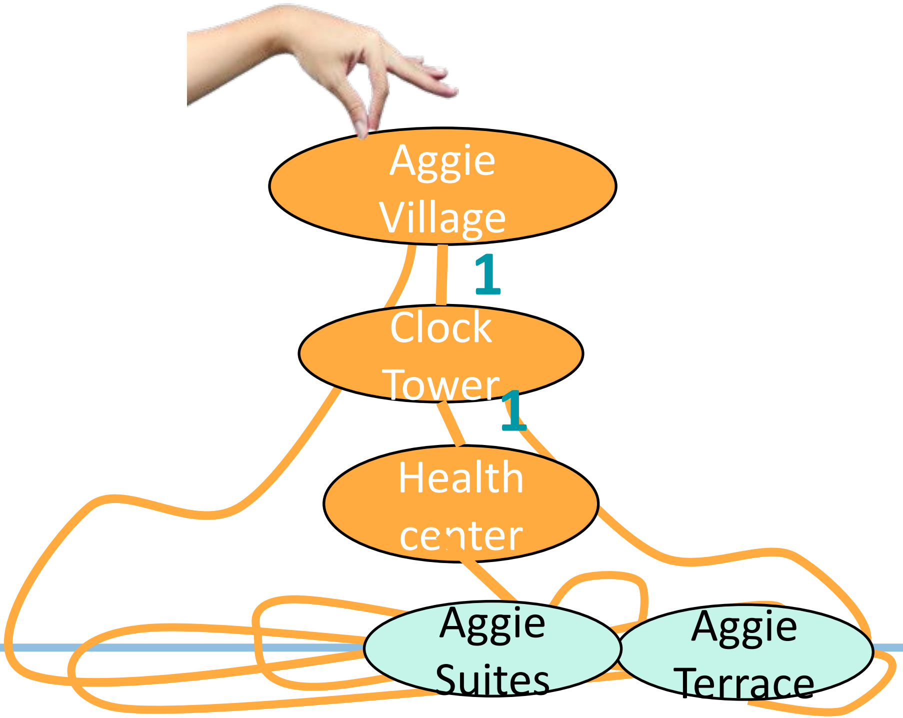


Dijkstra
intuition

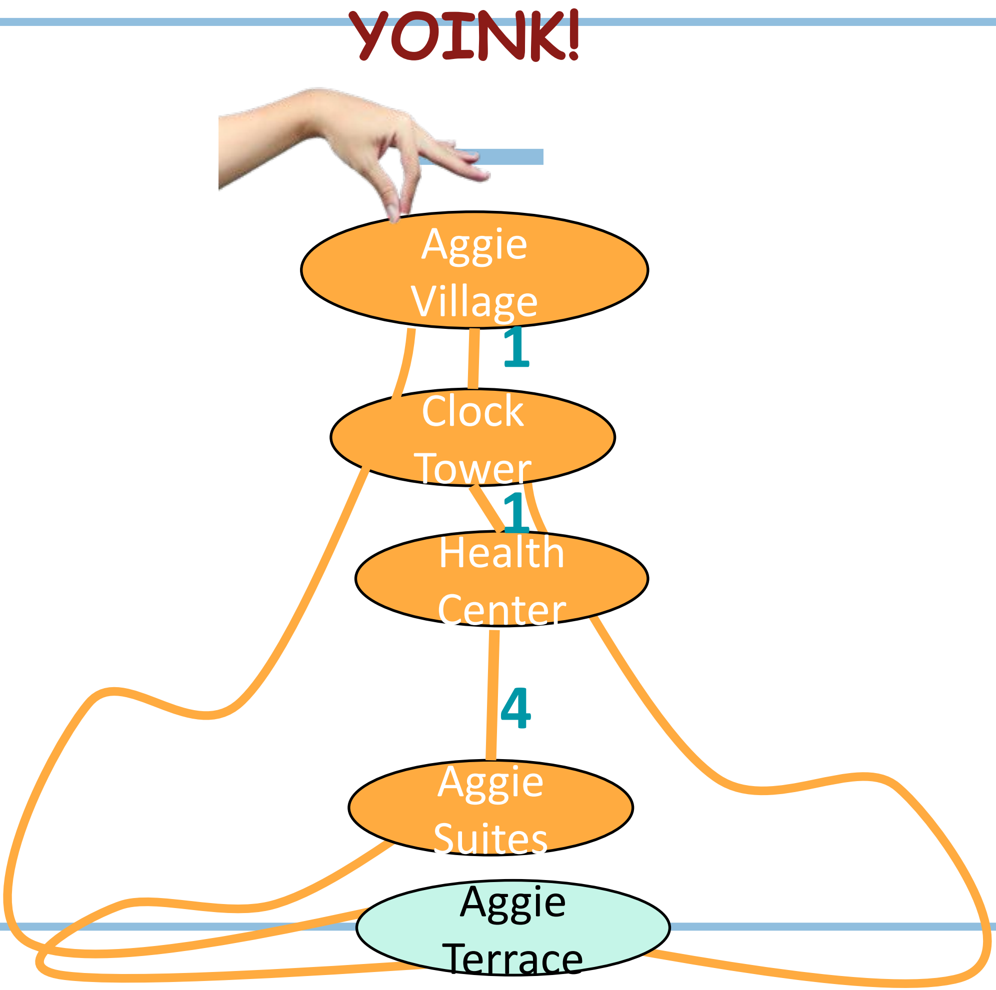


Dijkstra
intuition

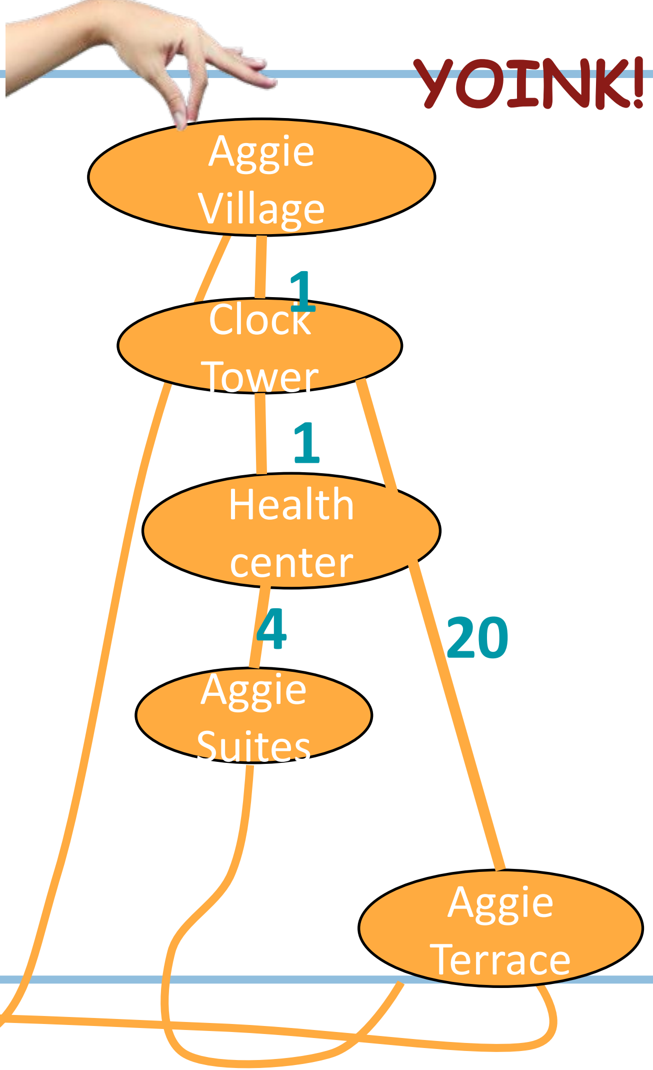
YOINK!



Dijkstra
intuition



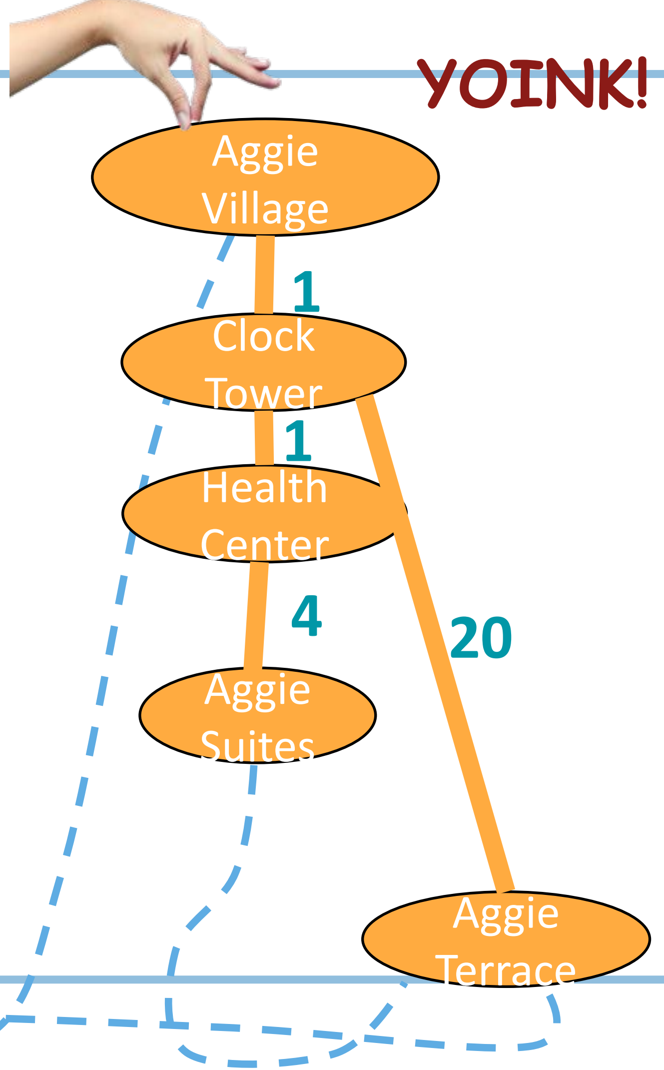
Dijkstra intuition



Dijkstra intuition

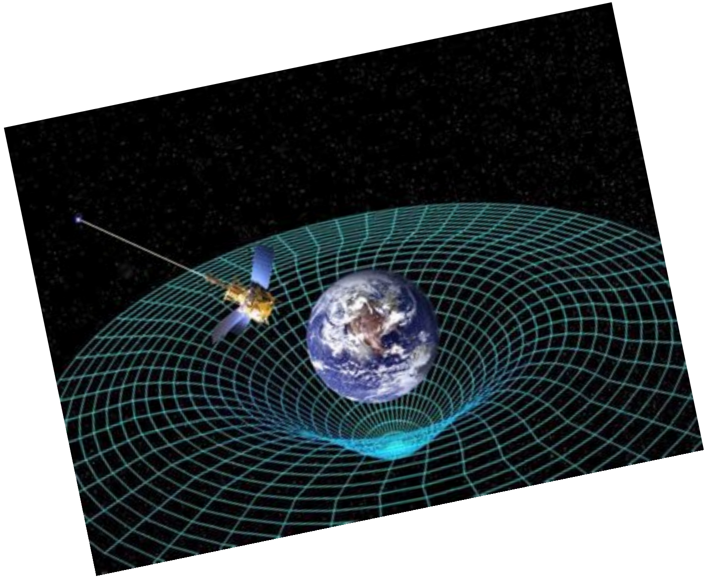
This creates a tree!

The shortest paths are the lengths along this tree.



How do we actually implement this?

- Without string and gravity?



Dijkstra by example

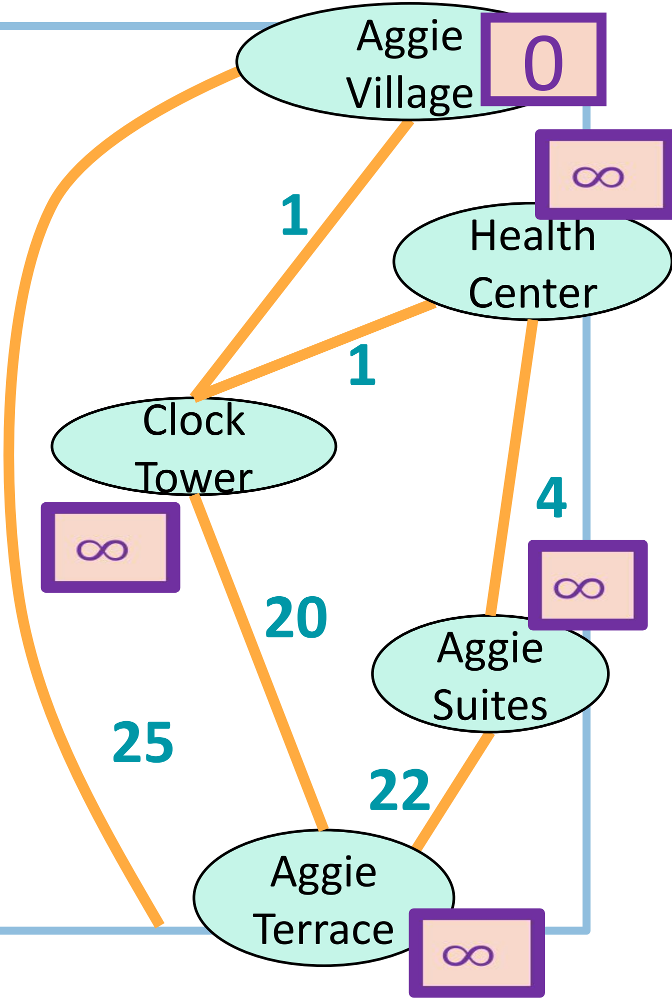
How far is a node from Aggie Village?

I'm not sure yet

I'm sure

X

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



Dijkstra by example

How far is a node from Aggie Village?

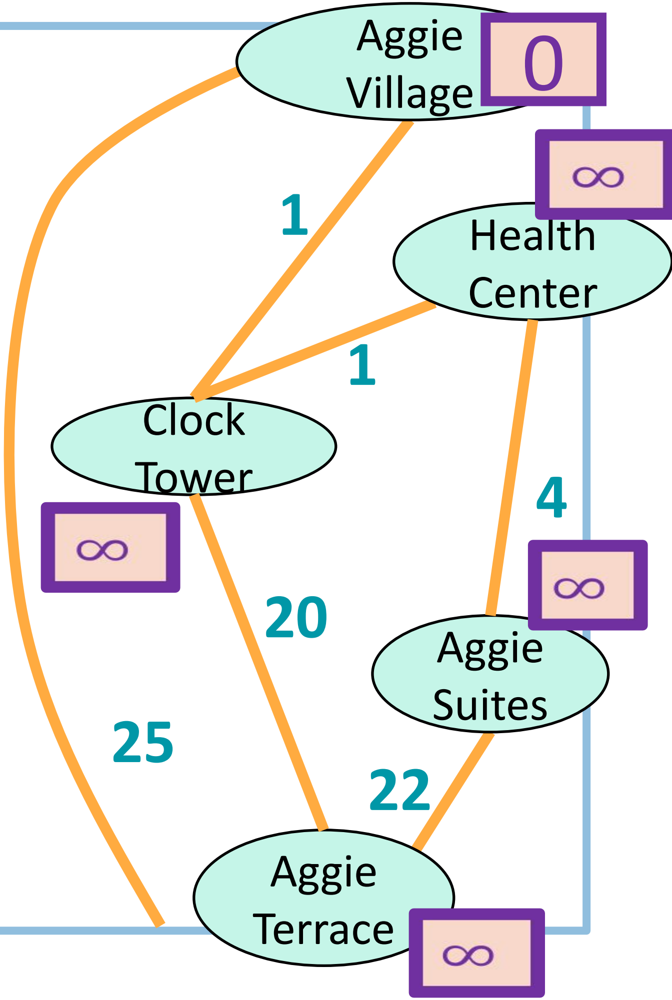
I'm not sure yet

I'm sure

X

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

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



Dijkstra by example

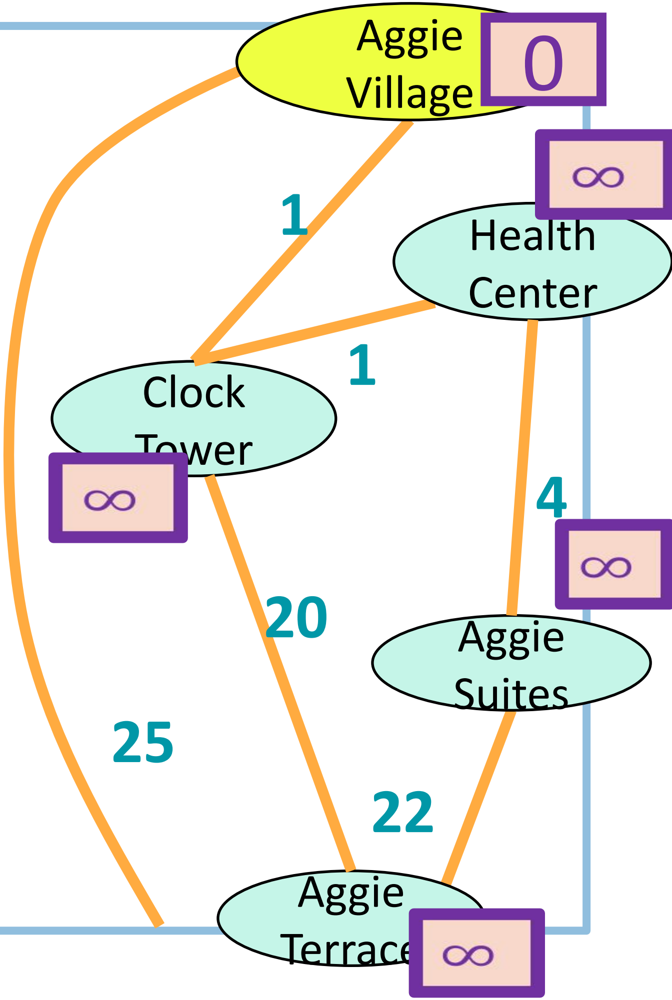
How far is a node from Aggie Village?

I'm not sure yet

I'm sure

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

- 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

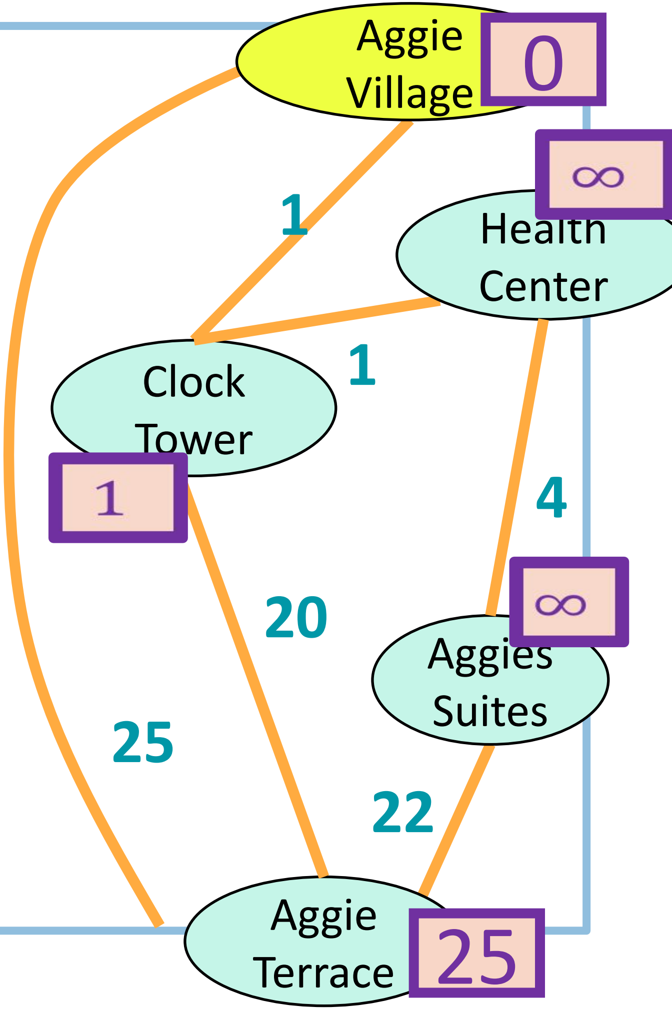
How far is a node from Aggie Village?

I'm not sure yet

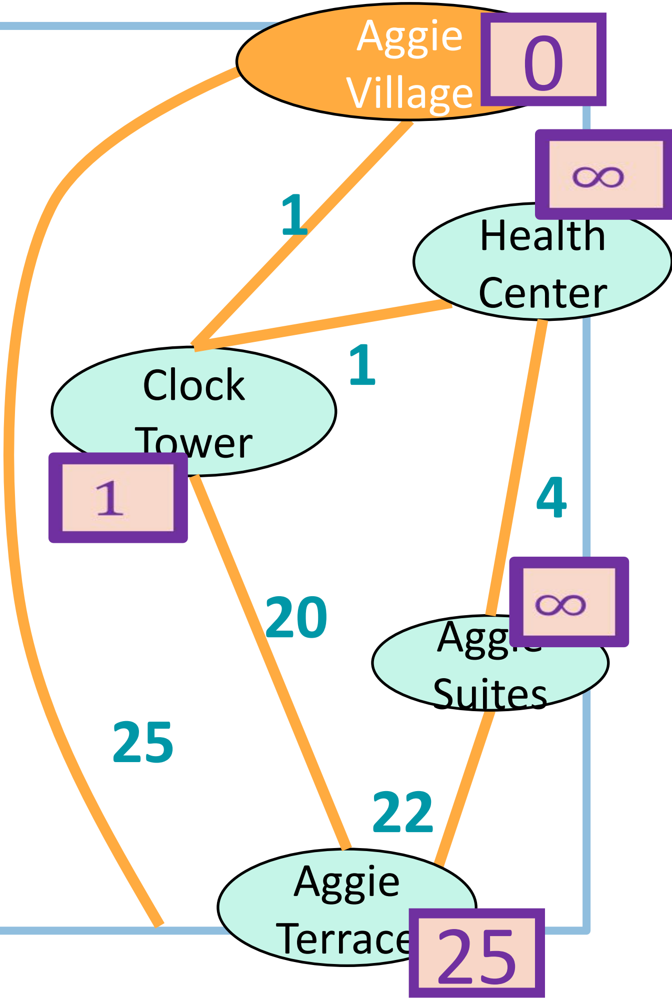
I'm sure

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

- 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 Aggie Village?

(not-sure node) I'm not sure yet

(sure node) I'm sure

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

- 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 Aggie Village?

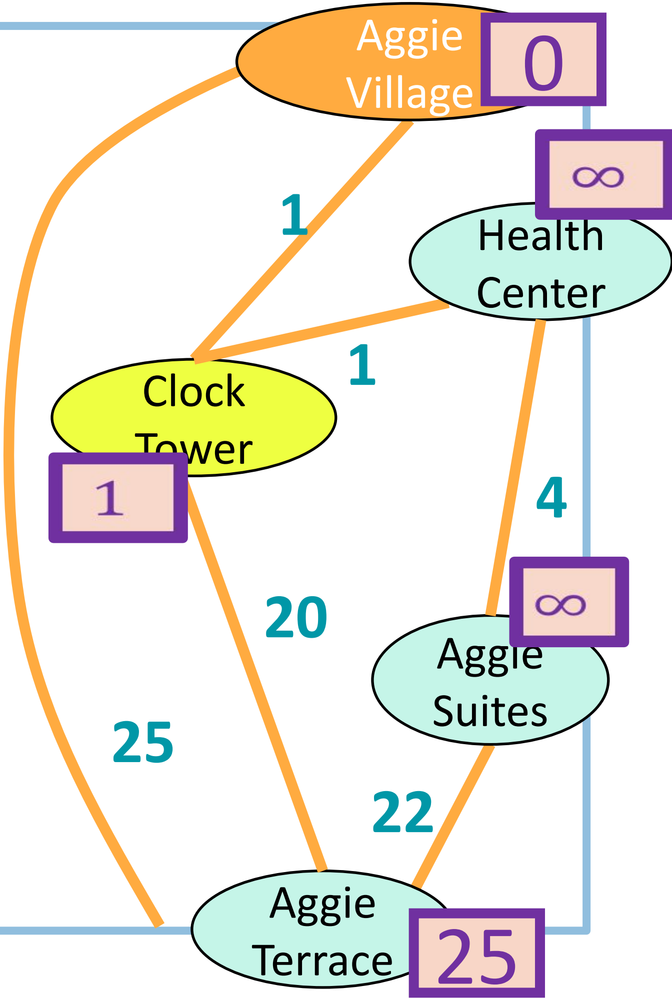
I’m not sure yet

I’m sure

x

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

- 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 Aggie Village?

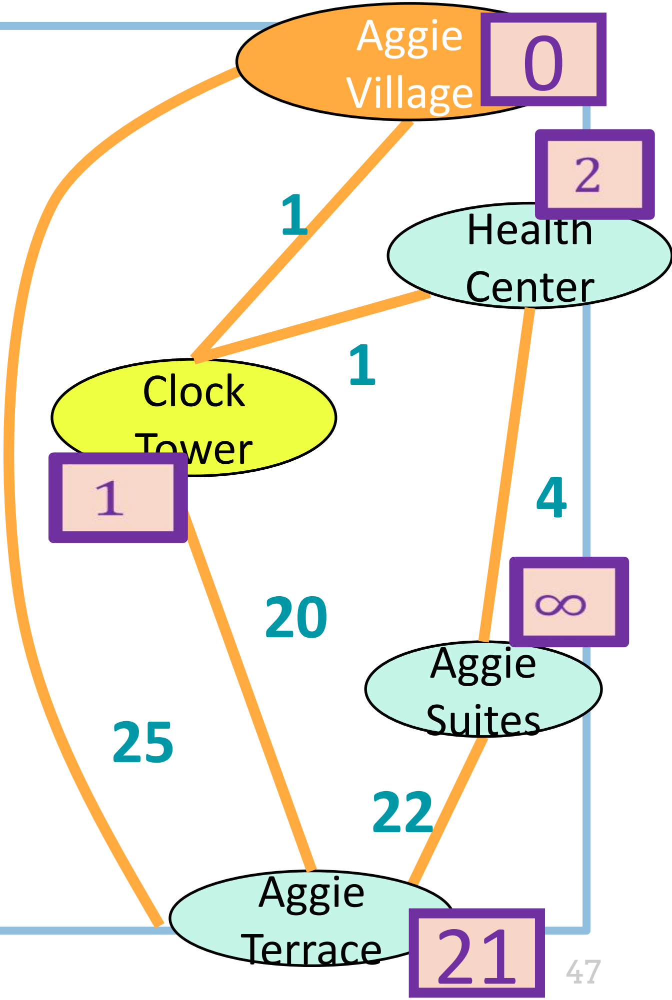
I'm not sure yet

I'm sure

x

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

- 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

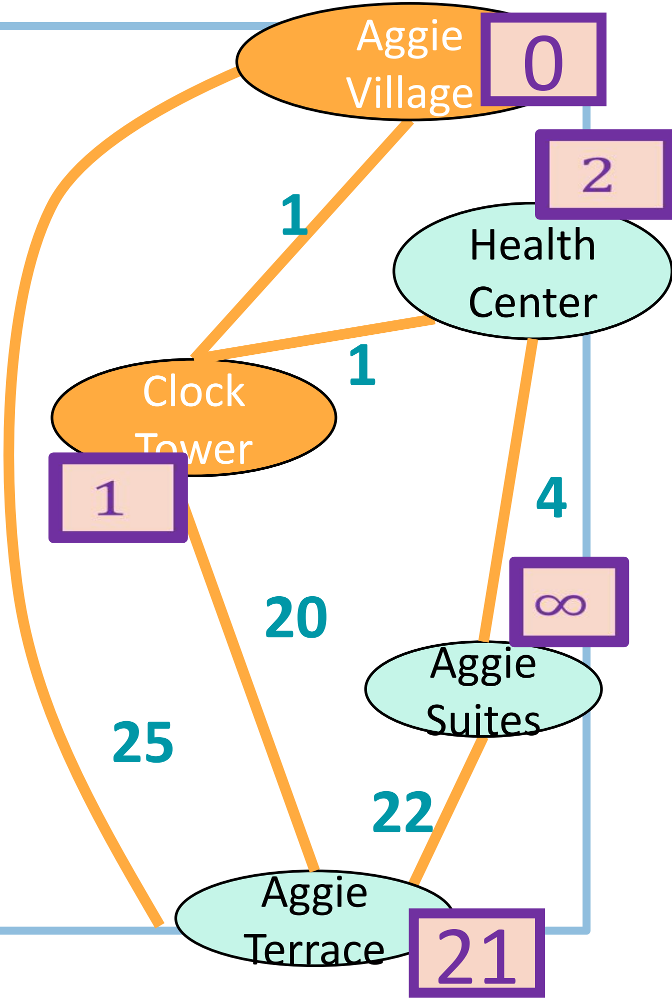
How far is a node from Aggie Village?

I'm not sure yet

I'm sure

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

- 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

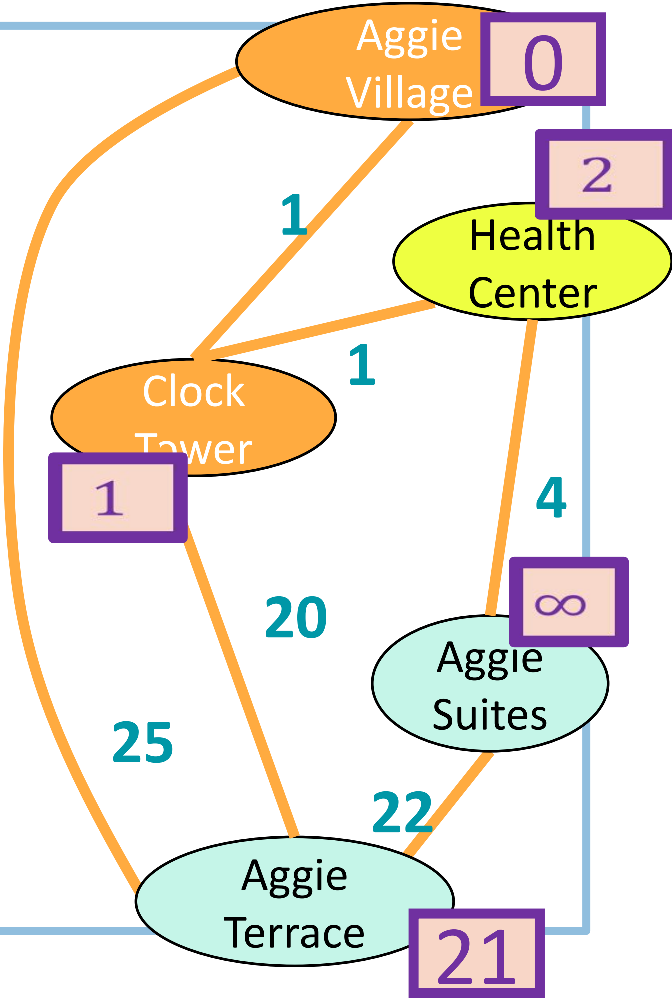
How far is a node from Aggie Village?

I'm not sure yet

I'm sure

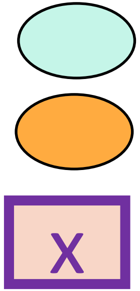
x = d[v] is my best **over-estimate** for dist(Aggie Village,v).

- 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 Aggie Village?

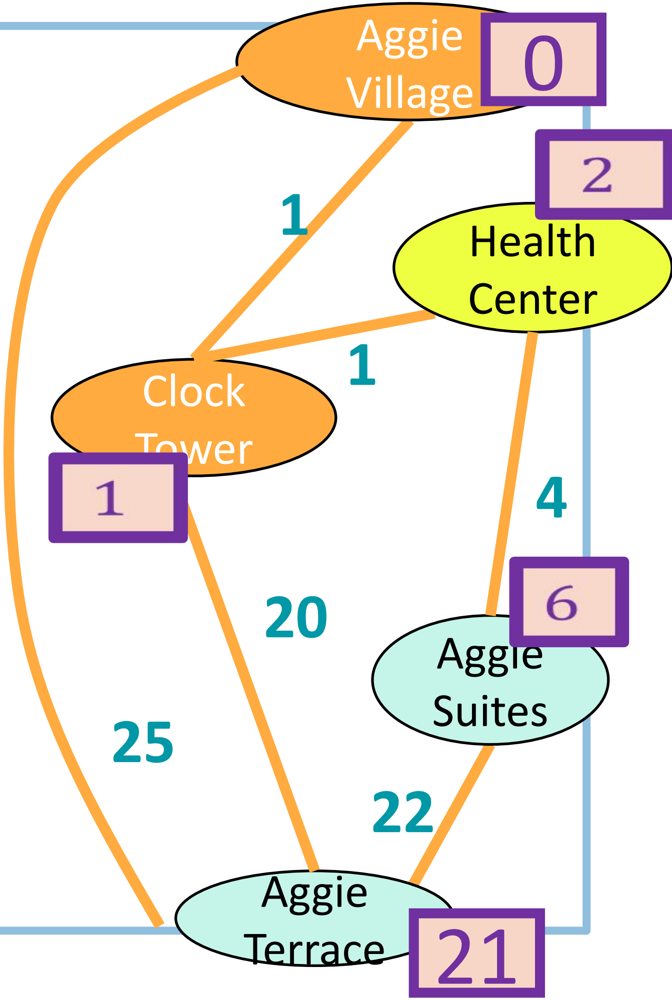


I'm not sure yet

I'm sure

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

- 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

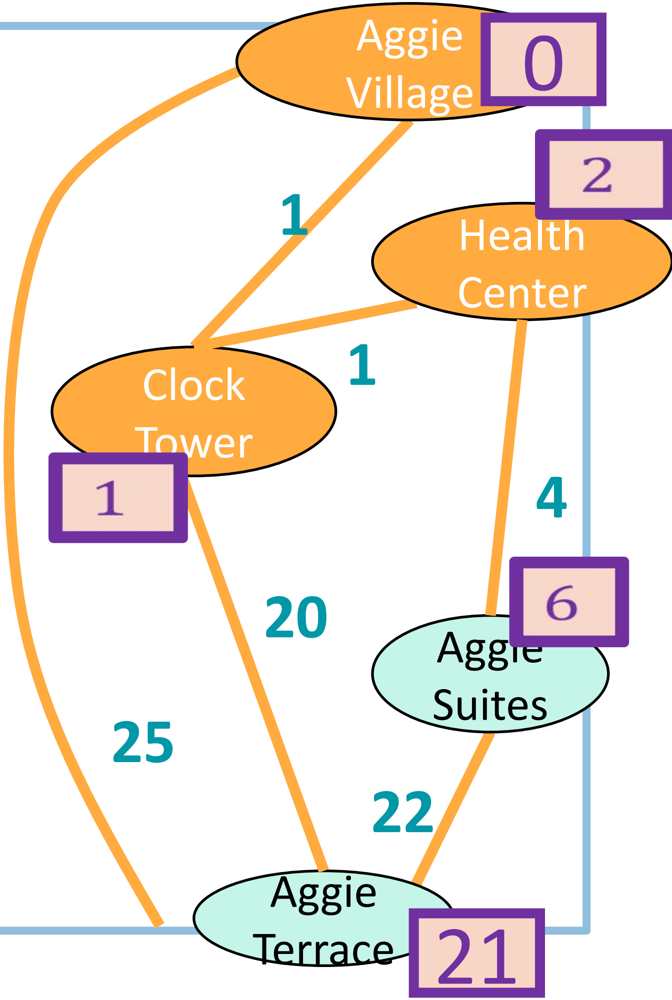
How far is a node from Aggie Village?

I'm not sure yet

I'm sure

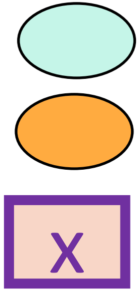
X **x = d[v]** is my best **over-estimate** for dist(Aggie Village,v).

- 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 Aggie Village?

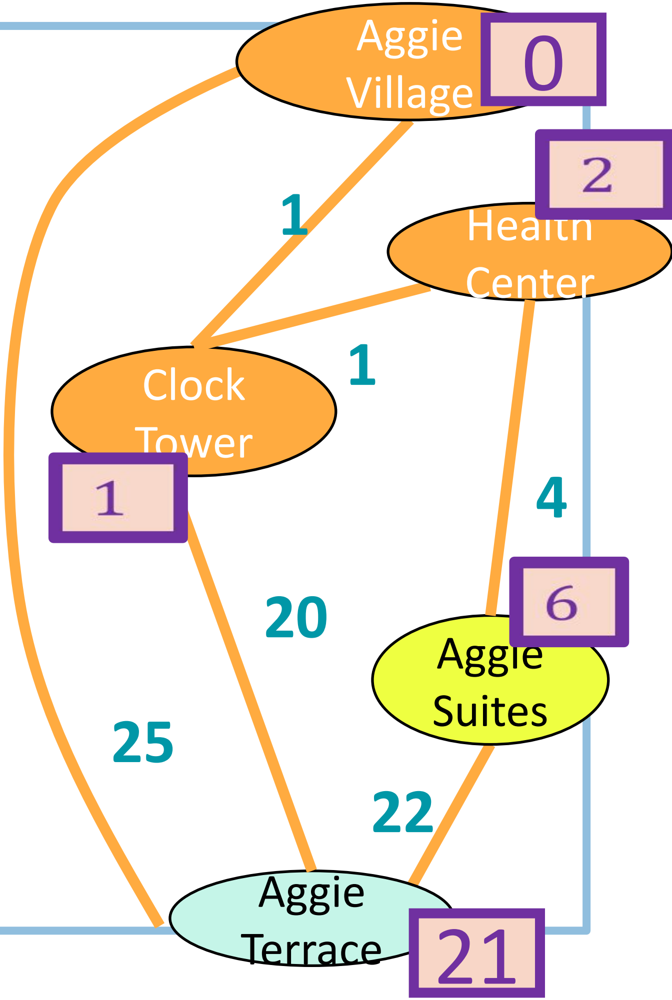


I'm not sure yet

I'm sure

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

- 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 Aggie Village?

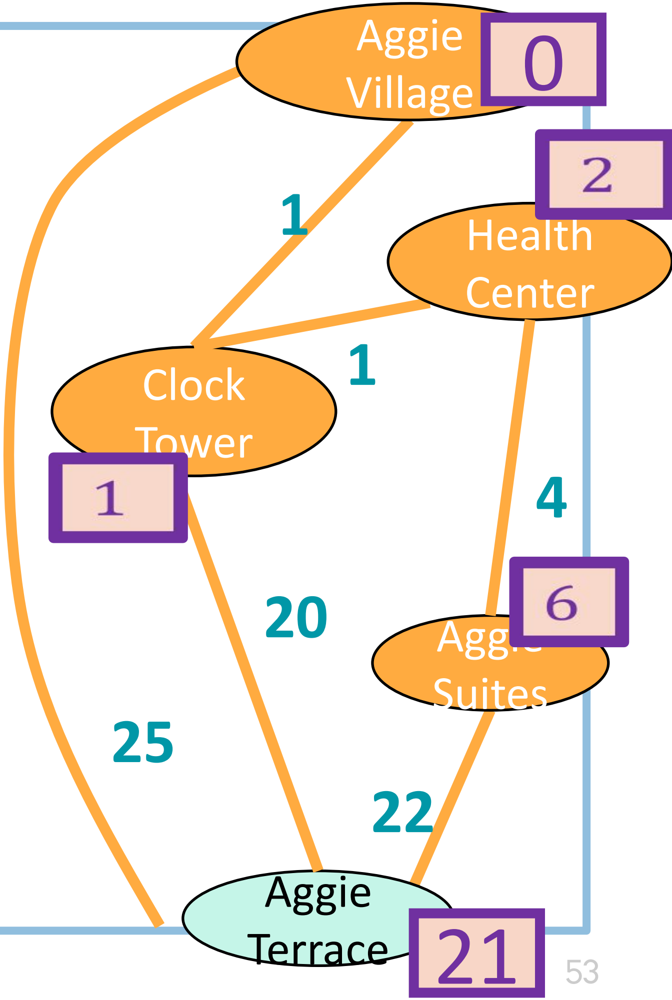
I'm not sure yet

I'm sure

x

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

- 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

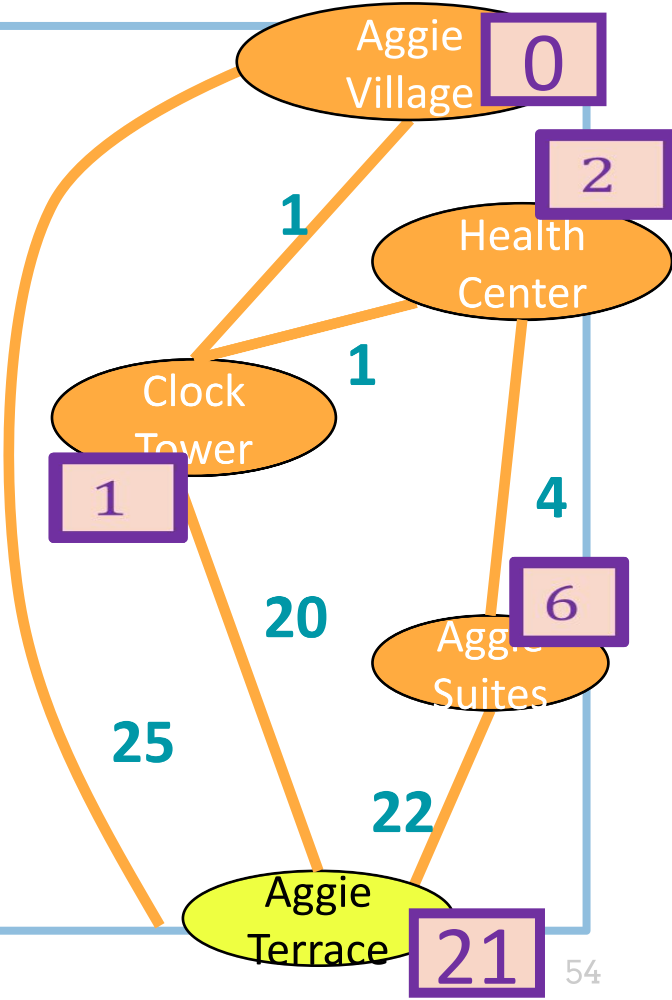


Dijkstra by example

How far is a node from Aggie Village?

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

- 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 Aggie Village?

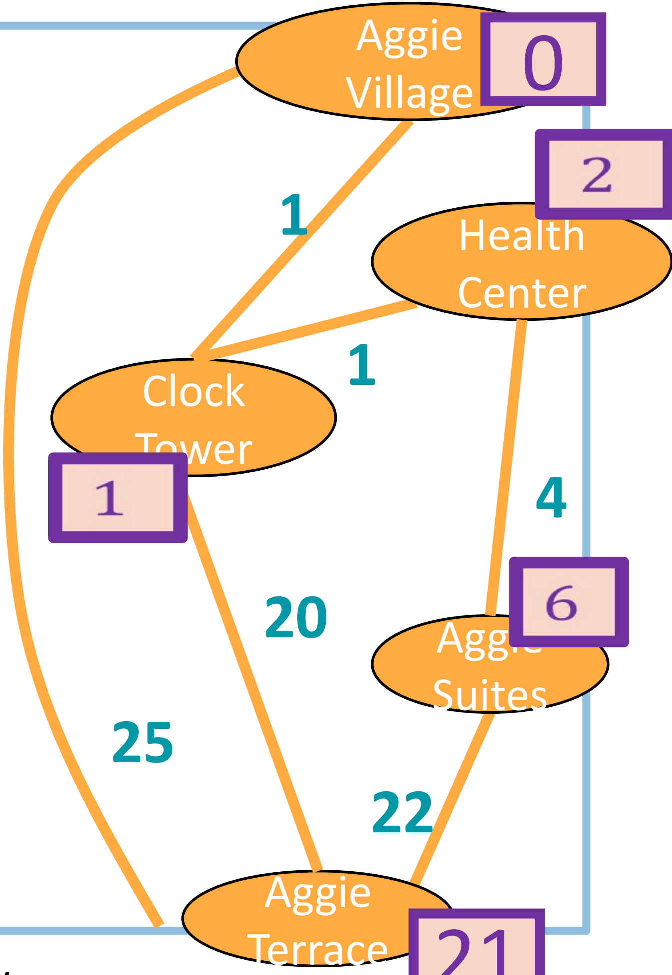
I'm not sure yet

I'm sure

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

- 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**, d(Aggie Village, v) = d[v] for all v



Dijkstra's Algorithm

Dijkstra(G,s):

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

Dijkstra's Running time?

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

Dijkstra's Running time?

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]

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

Dijkstra's Running time?

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

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

Depends on how we implement it…

We need a data structure that…

- Stores unsure vertices v
- … And 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
- … And 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
- … And 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**.

What's the runtime…?

How many times are we performing each operation?

We need a data structure that…

- Stores unsure vertices v
- … And 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**.

= V * (findMin + removeMin) + E * updateKey

If we use an array…

- findMin = ???
- removeMin = ???
- updateKey = ???

If we use an array...

- findMin = O(V)
- removeMin = O(V)
- updateKey = O(1)

Running time of Dijkstra
= O(V * (findMin + removeMin) + E * updateKey)
= O(V * (V + V) + E * 1)
= $O(V^2 + E)$

If we use a (balanced) BST...

- findMin = ???
- removeMin = ???
- updateKey = ???

If we use a (balanced) BST…

- findMin = O(log(V))
- removeMin = O(log(V))
- updateKey = O(log(V))

Running time of Dijkstra
= O(V * (findMin + removeMin) + E * updateKey)
= O(V * (log(V) + log(V)) + E * log(V))
= O(Vlog(V)+ Elog(V))
= O((V+E)log(V))

Better than an array if the graph is sparse!
aka if m is much smaller than n^2

If we use a hash table...

- findMin = ???
- removeMin = ???
- updateKey = ???

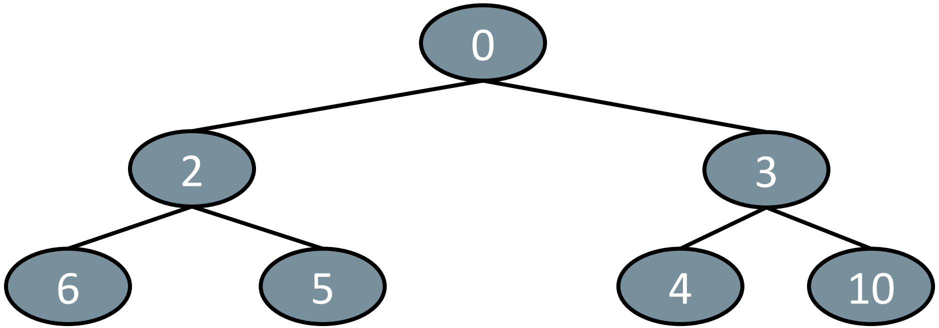
If we use a hash table...

- findMin = $O(V)$
- removeMin = $O(1)$
- updateKey = $O(1)$

Running time of Dijkstra
= $O(V * (\text{findMin} + \text{removeMin}) + E * \text{updateKey})$
= $O(V * (V + 1) + E * 1)$
= $O(V^2 + E)$

Heaps support these operations…

- findMin
- removeMin
- updateKey



- A **min-heap** is a tree-based data structure that has the property that every node has a smaller key than its children.
- A **max-heap** is similar, but every node has a larger key than its children.
- Not covered in this class – see CS166, but we will use them.

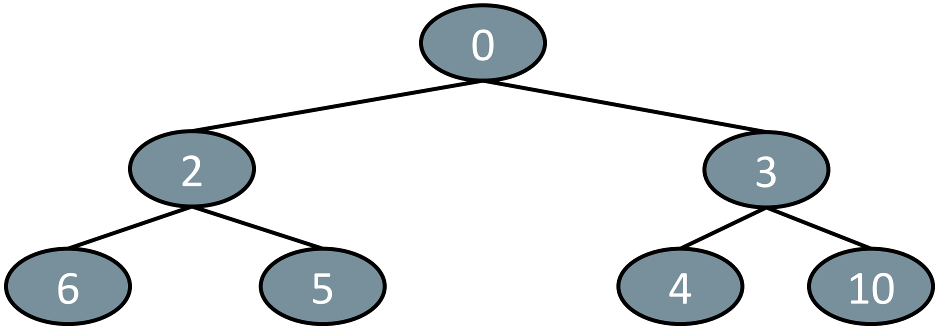
Many heap implementations

Nice chart on Wikipedia:

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

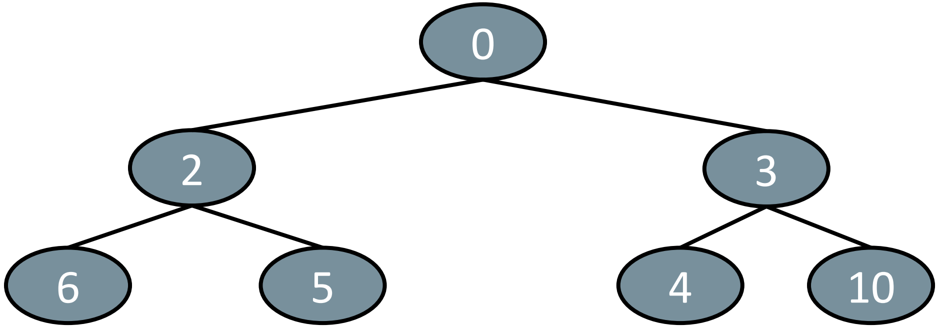
Say we use a binary min-heap...

- findMin = ???
- removeMin = ???
- updateKey = ???



Say we use a binary min-heap…

- findMin = O(1)
- removeMin = O(log(V)*)
- updateKey = O(log(V)*)



Running time of Dijkstra

= O(V * (findMin + removeMin) + E * updateKey)

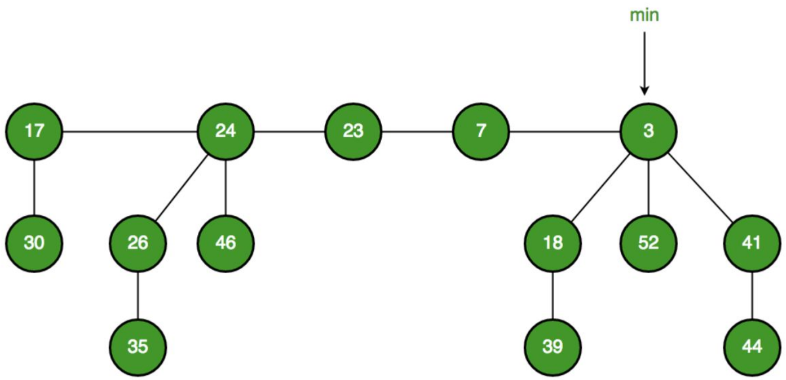
= O(V * (1 + log(V)) + E * log(V))

= O((V+E)log(V))

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

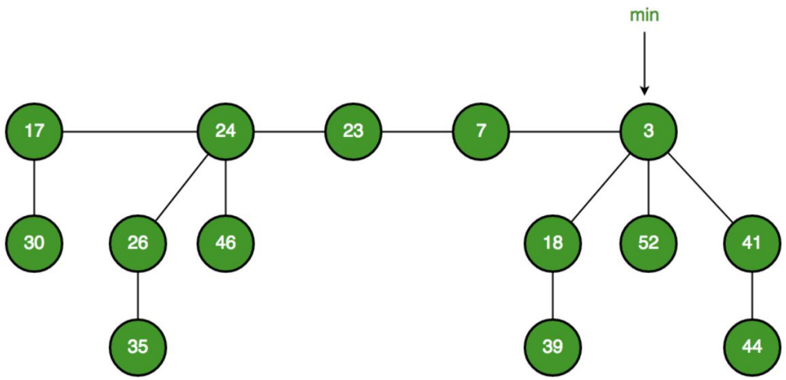
Say we use a fibonacci min-heap...

- findMin = ???
- removeMin = ???
- updateKey = ???



Say we use a fibonacci heap…

- findMin = O(1)
- removeMin = O(log(V)*)
- updateKey = O(1)



Running time of Dijkstra

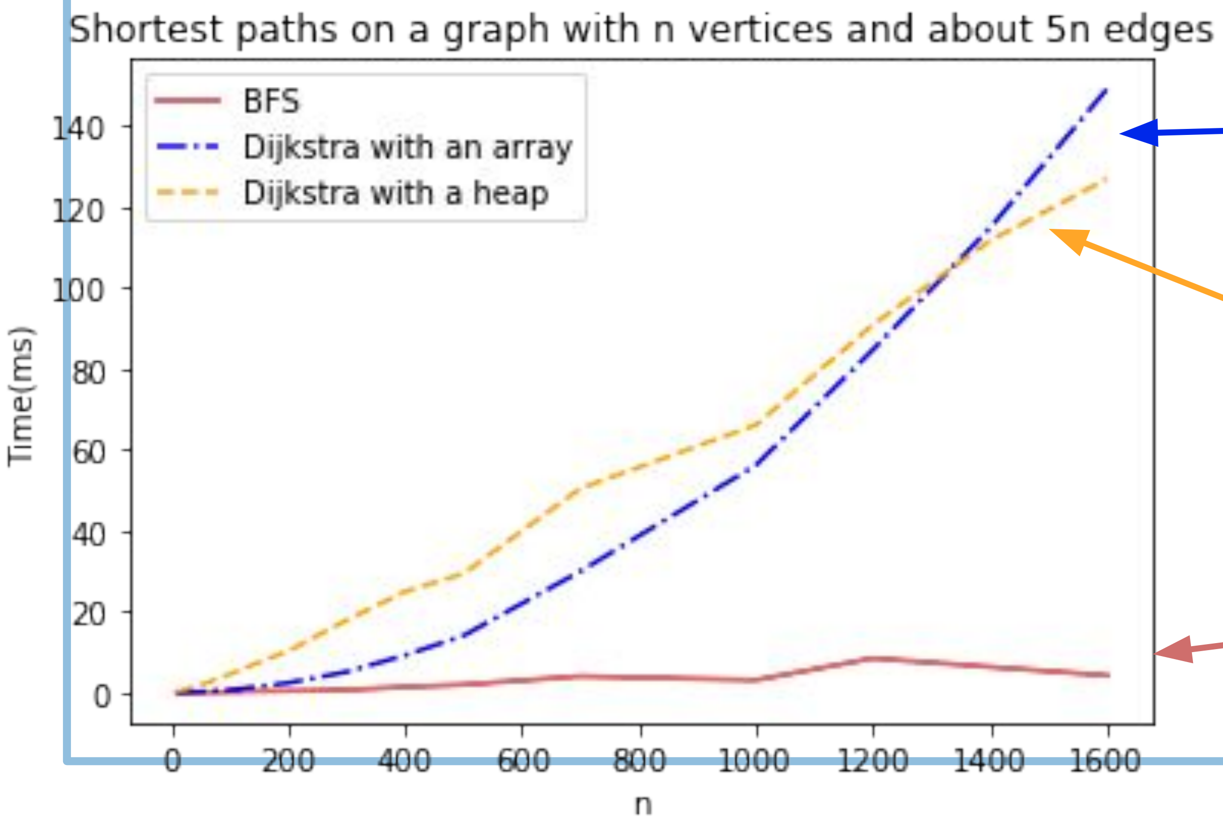
= O(V * (findMin + removeMin) + E * updateKey)

= O(V * (1 + log(V)) + E * 1)

= O(Vlog(V)+ E)

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

In practice



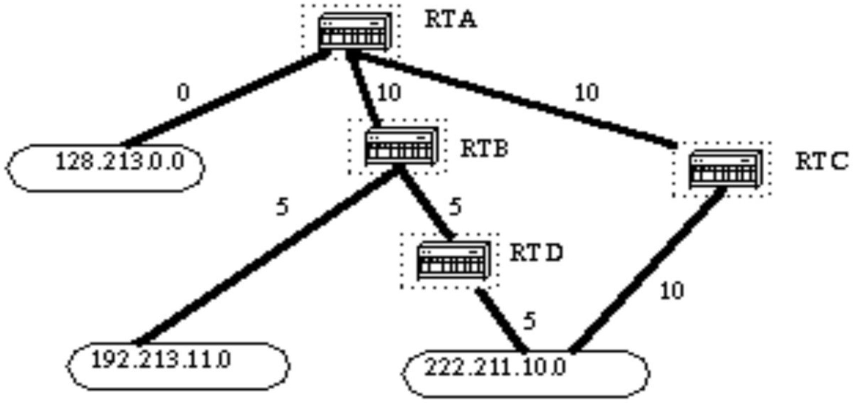
Dijkstra using a Python list to keep track of vertices has quadratic runtime.

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

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

Dijkstra is used in practice

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



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

Dijkstra Drawbacks

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

COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 16: Weighted Graphs, Dijkstra’s, A*

Lecturer: Chris Lucas (cflucas@ncat.edu)