Lec13: Graph Algorithms III

Algorithm I
COMP319-003
Spring 2023

Instructor: Jiyeon Lee

School of Computer Science and Engineering
Kyungpook National University **(KNU)**

Last time

- Graph representation
- depth-first search (DFS)
- Plus, applications!
  - Topological sorting
  - Strongly Connected Components

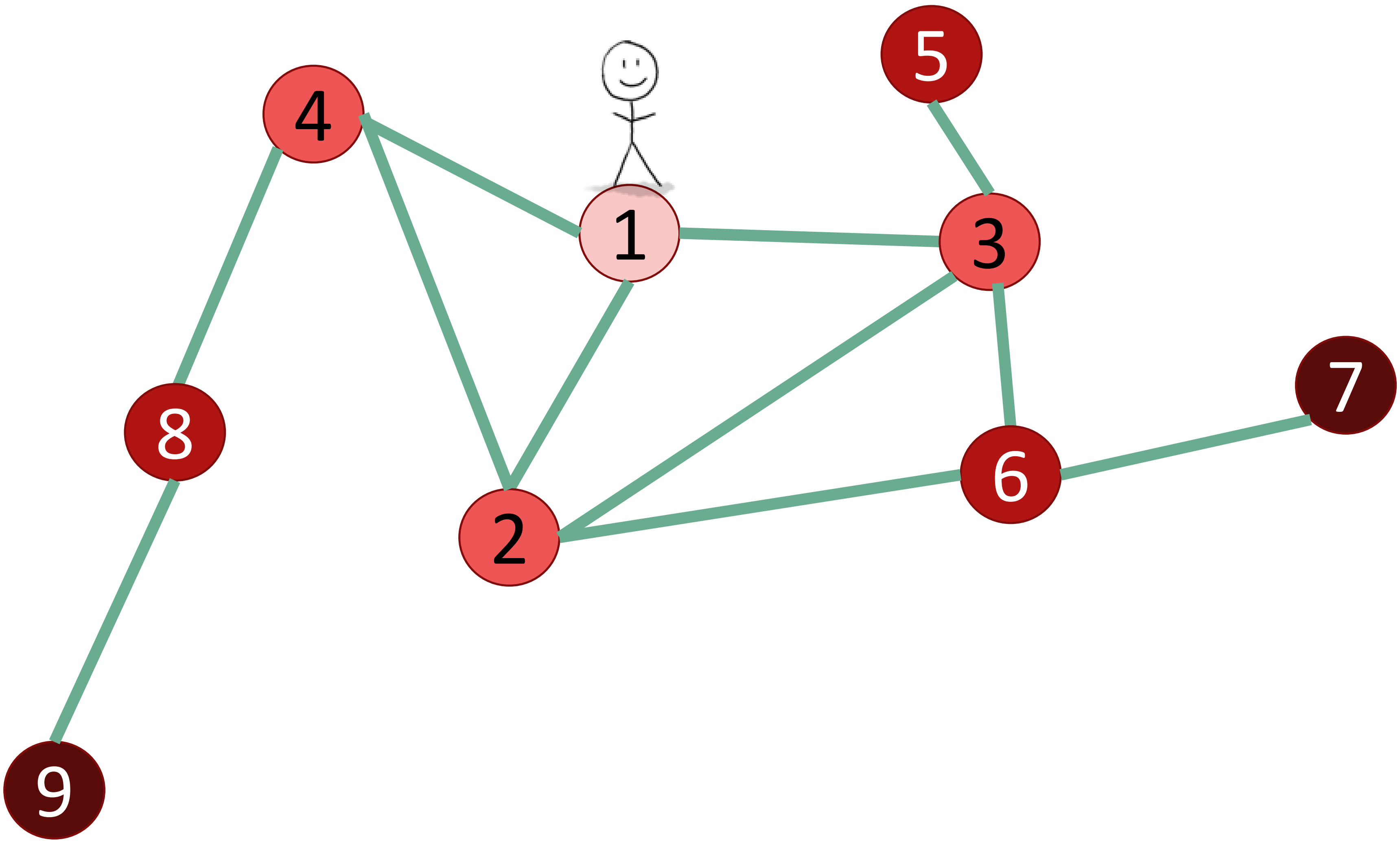
Outline

1. Graph Representations
2. DFS (Depth First Search)
   - Topological Ordering
   - Strongly Connected Components
3. **BFS (Breadth-First Search)**
   - **Dijkstra's Algorithm**

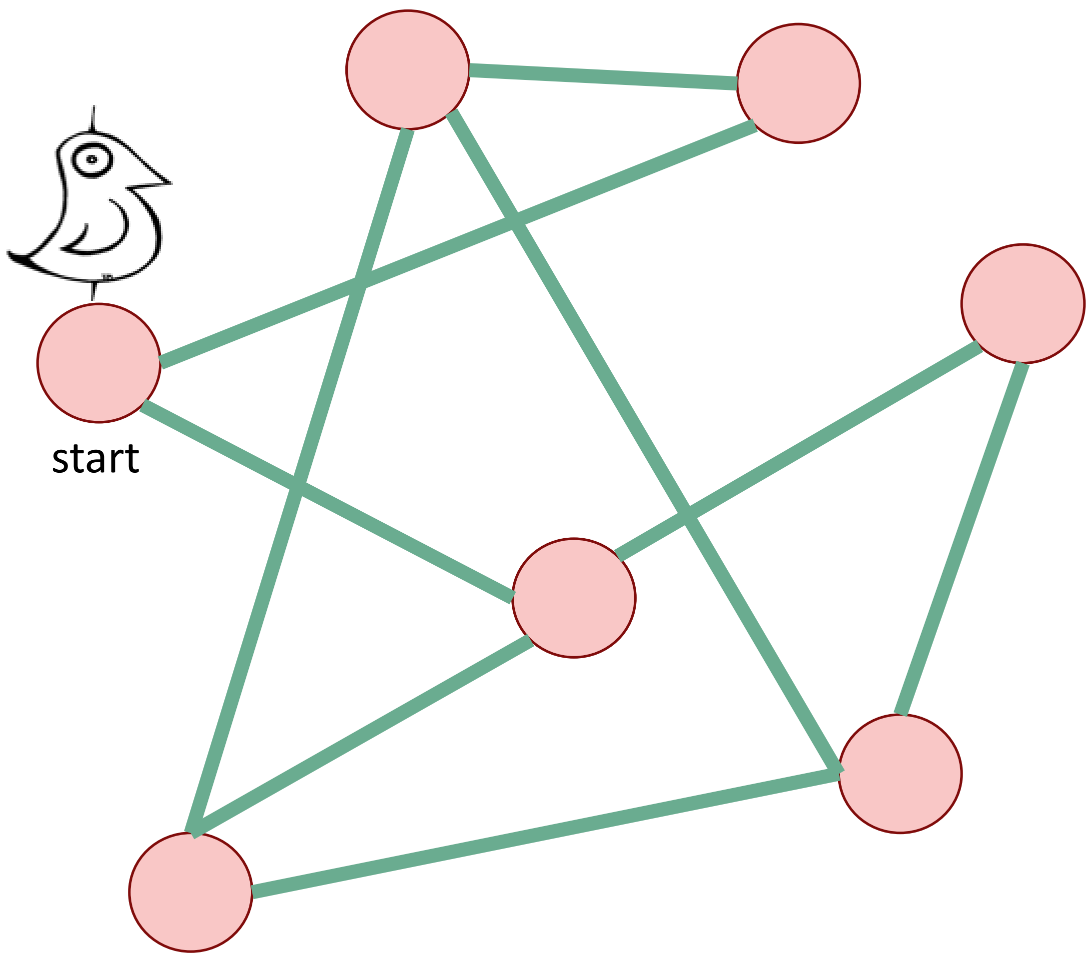
- *Reading: CLRS 22.1 – 22.4*

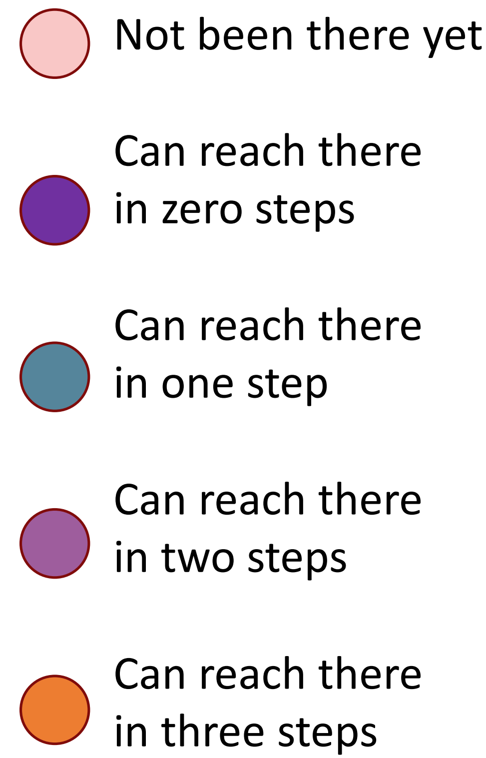
Breadth-First Search

How do we explore a graph?

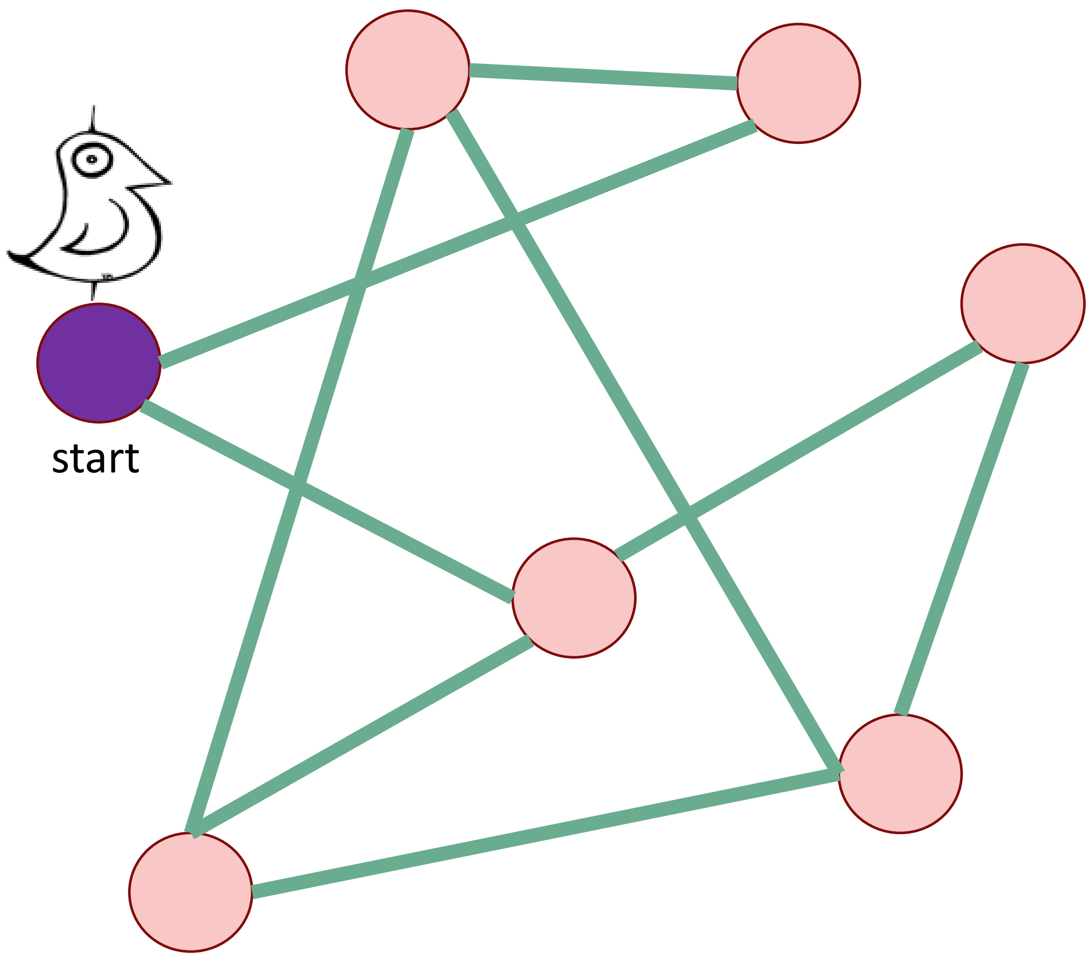


Breadth-First Search





Breadth-First Search



Not been there yet

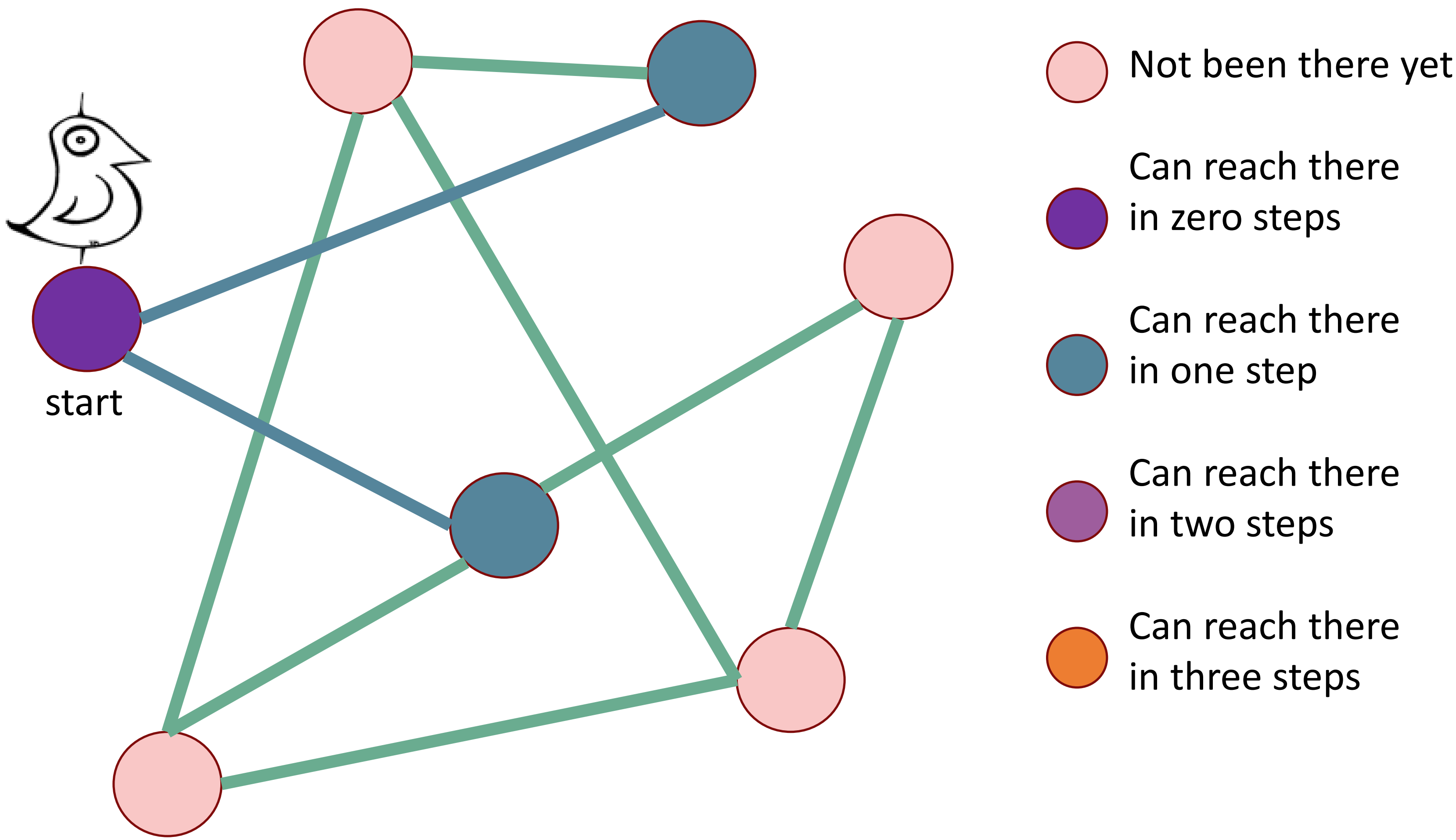
Can reach there in zero steps

Can reach there in one step

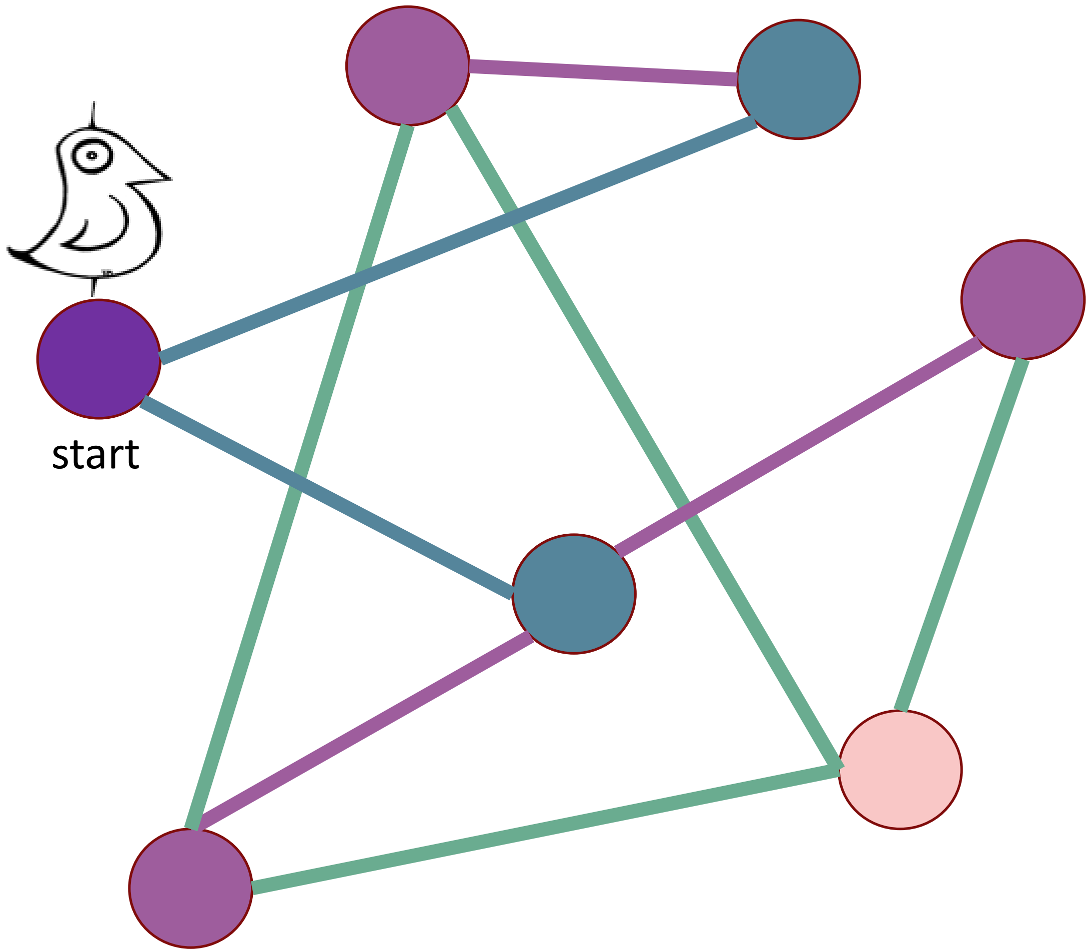
Can reach there in two steps

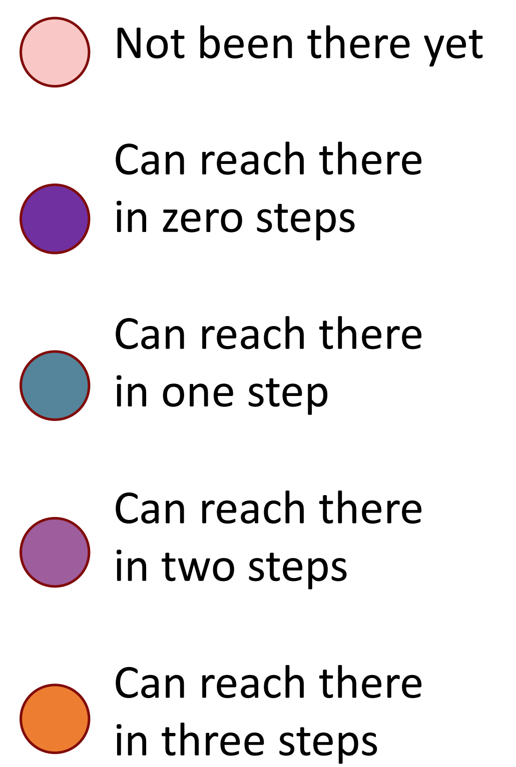
Can reach there in three steps

Breadth-First Search

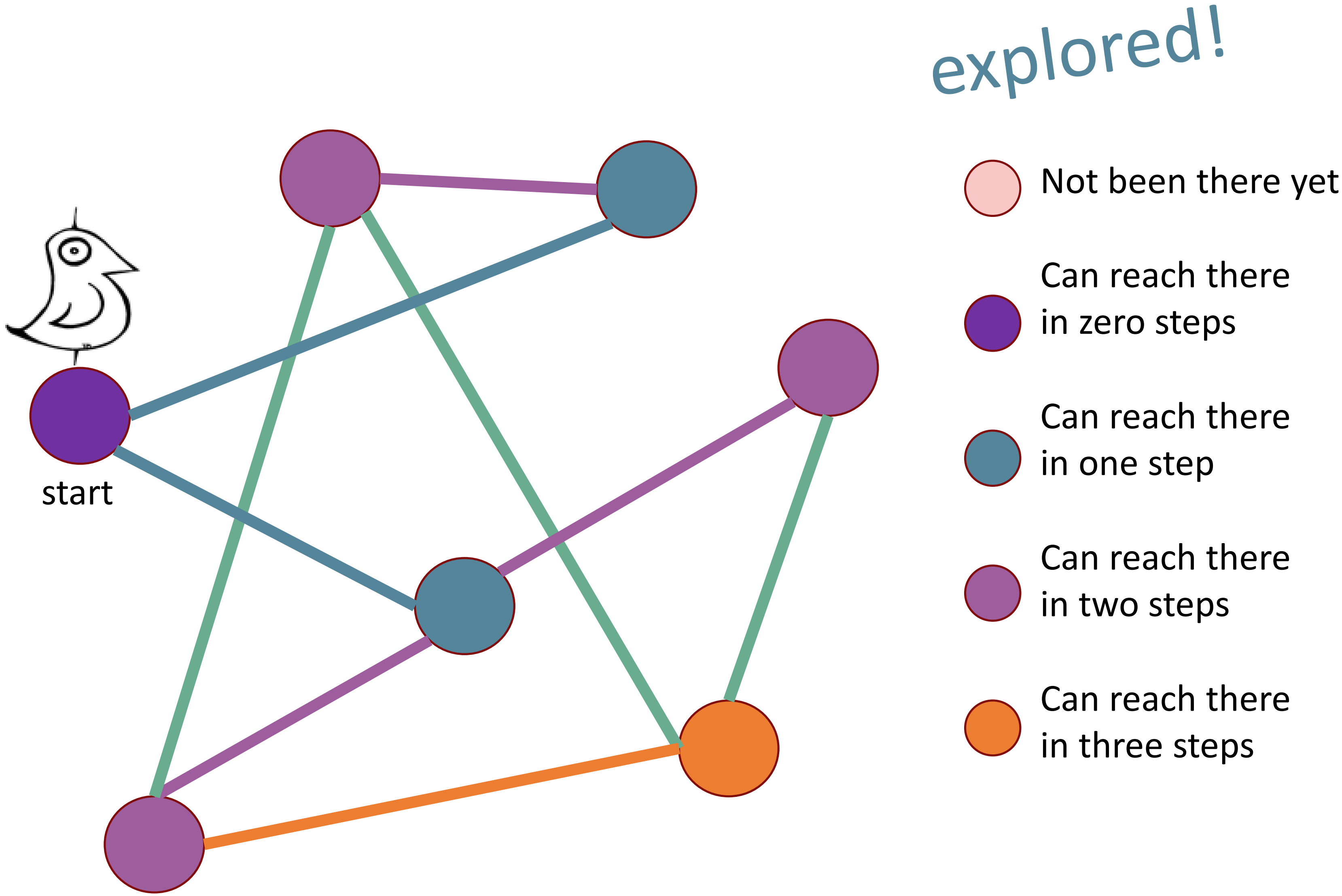


Breadth-First Search



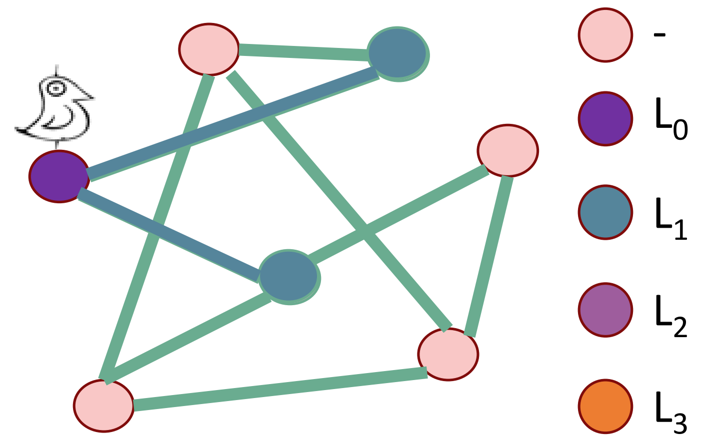


Breadth-First Search



Breadth-First Search

- Set L_i = [] for i=1,…,n
- L_0 = [w], where w is the start node
- Mark w as visited
- **For** i = 0, …, n-1:
  - **For** u in L_i:
    - **For** each v which is a neighbor of u:
      - **If** v isn't yet visited:
        - mark v as visited, and put it in L_{i+1}



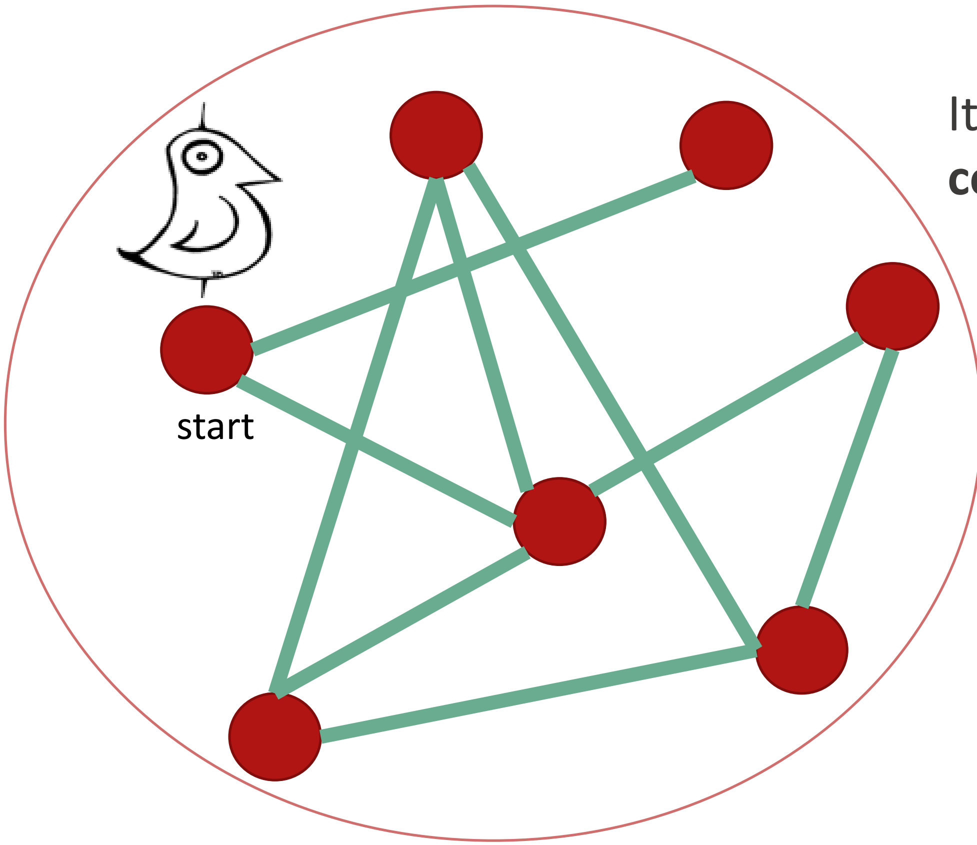
L_i is the set of nodes we can reach in i steps from w

Go through all the nodes in L_i and add their unvisited neighbors to L_{i+1}

- **Runtime?**
  - Same argument as DFS: BFS running time is **O(n + m)**

Breadth-First Search

- BFS also finds all the nodes reachable from the starting point

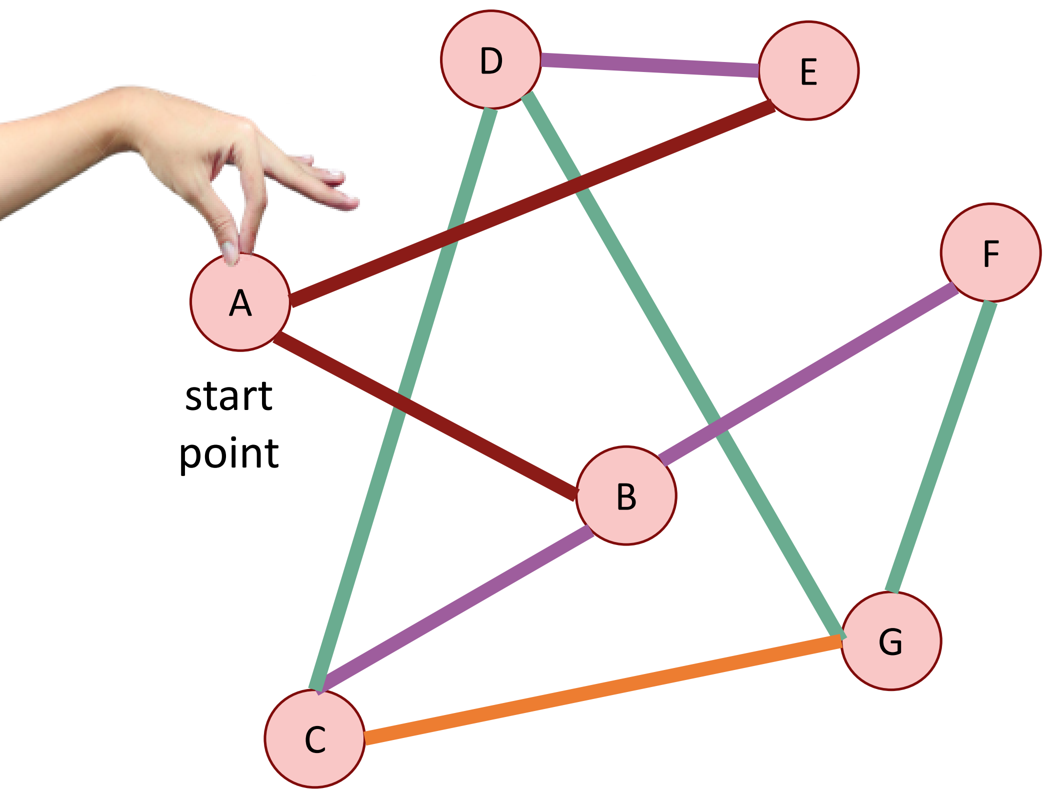


It is a good way to find all the **connected components.**

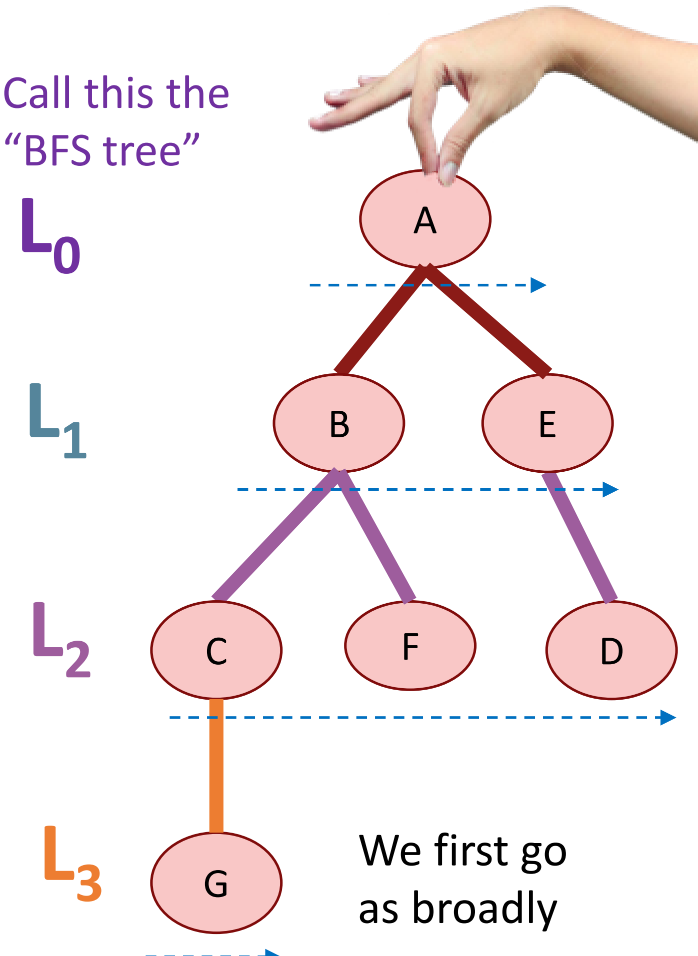
Like DFS, BFS also works fine on directed graphs.

Why is it called breadth-first?

- We are implicitly building a tree:



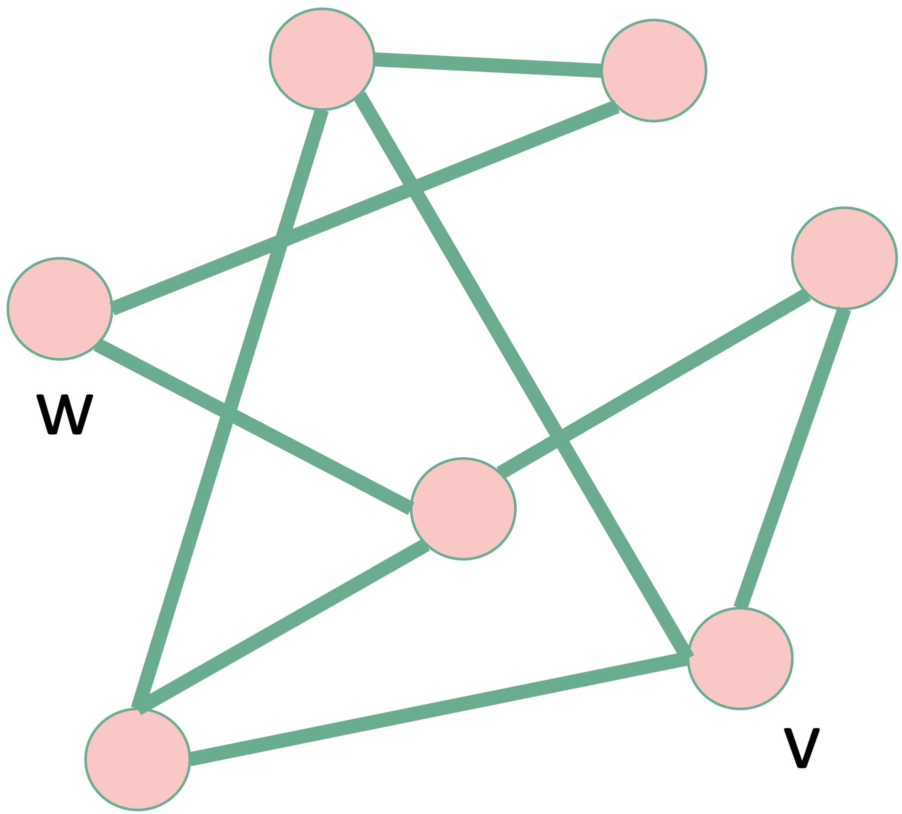
Call this the "BFS tree"



We first go as broadly as we can.

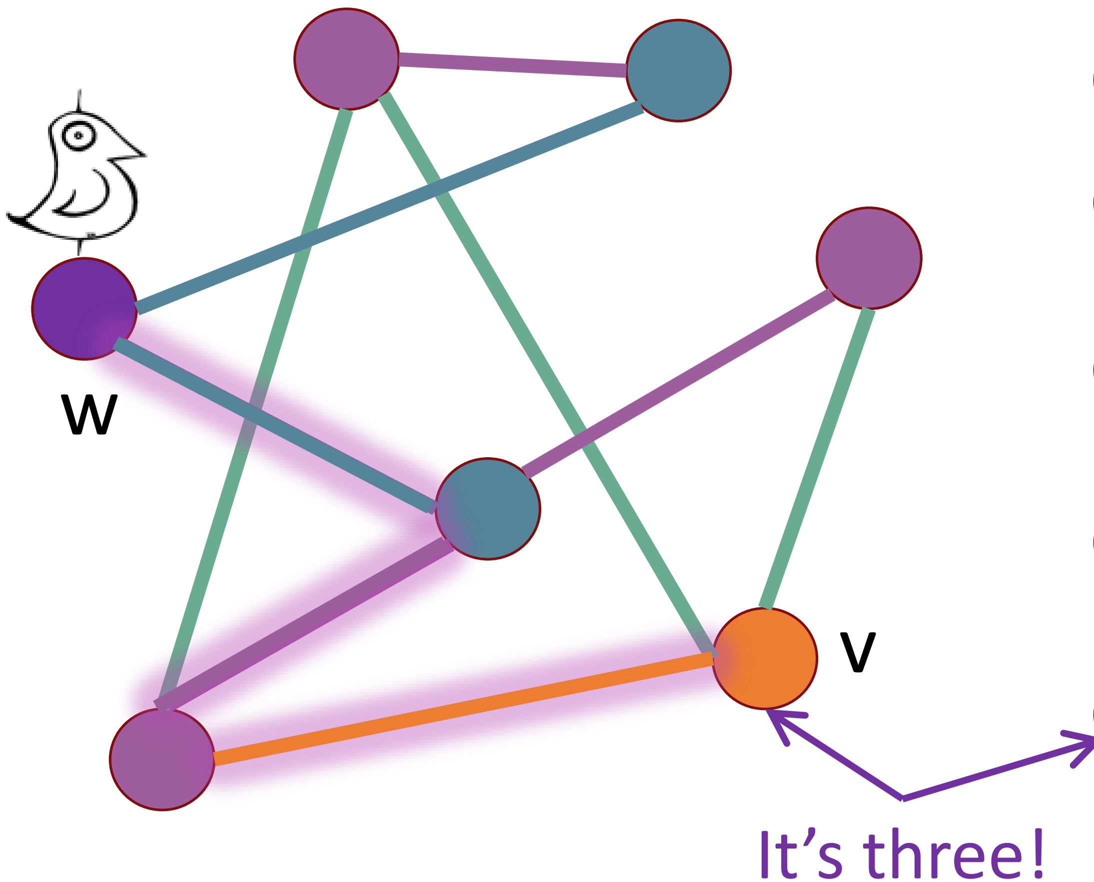
Application of BFS

- How long is the shortest path between w and v?



Application of BFS

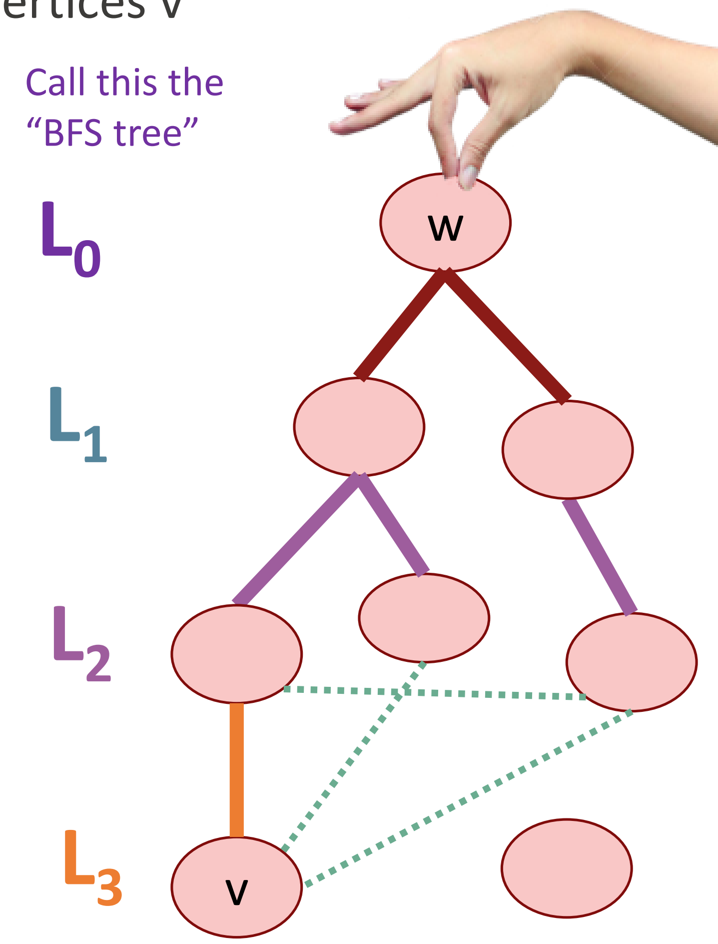
- How long is the shortest path between w and v?



- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

The Shortest Path

- To find the distance between w and all other vertices v
  - Do a **BFS** starting at w
  - For all v in L_i
    - The shortest path between w and v has length i. It is given by the path in the BFS tree.
  - If we never found v, the distance is infinite.



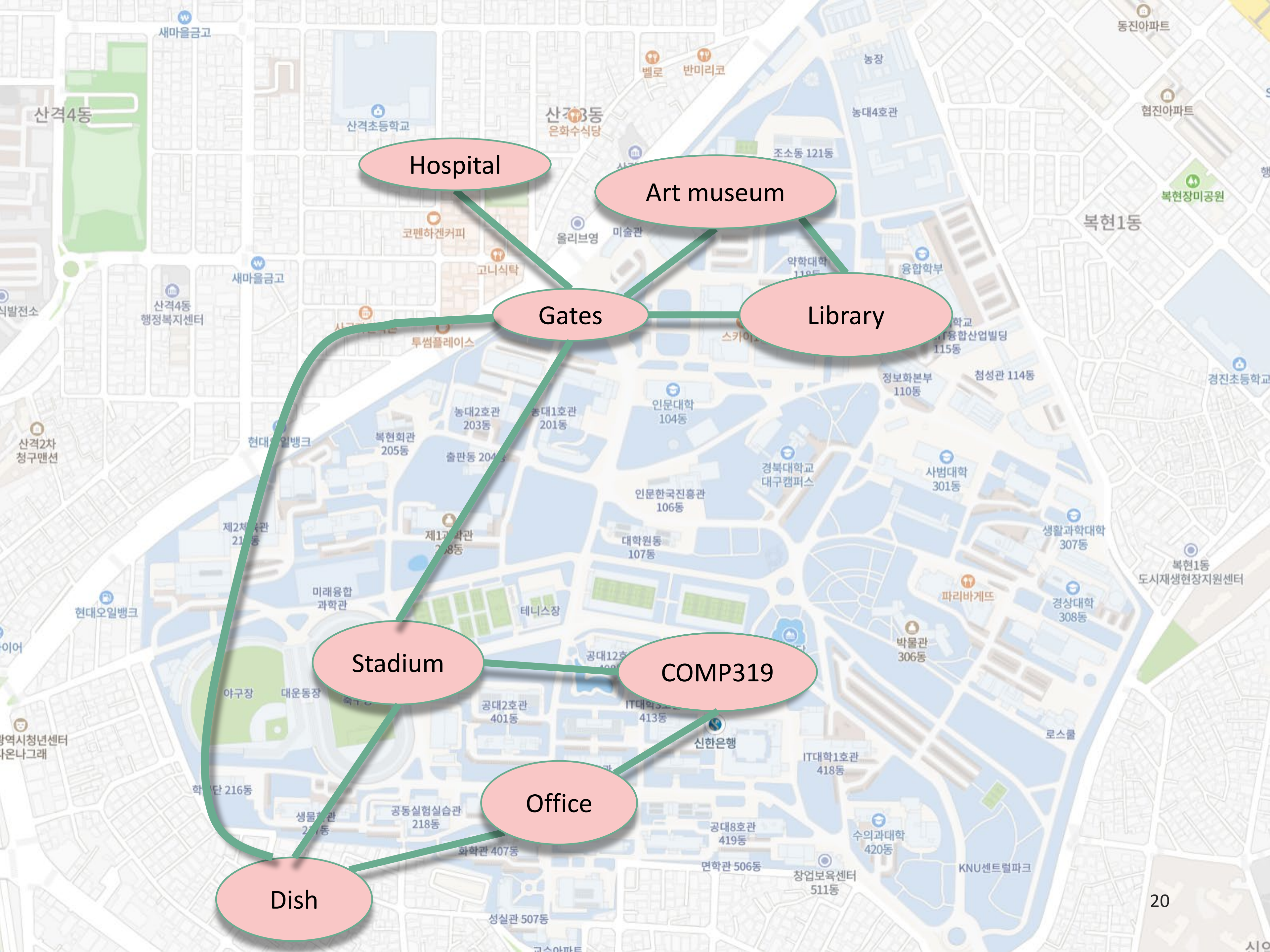
Summary

- The BFS tree is useful for computing distances between pairs of vertices.
- We can find the shortest path between u and v in time **O(n + m)**.

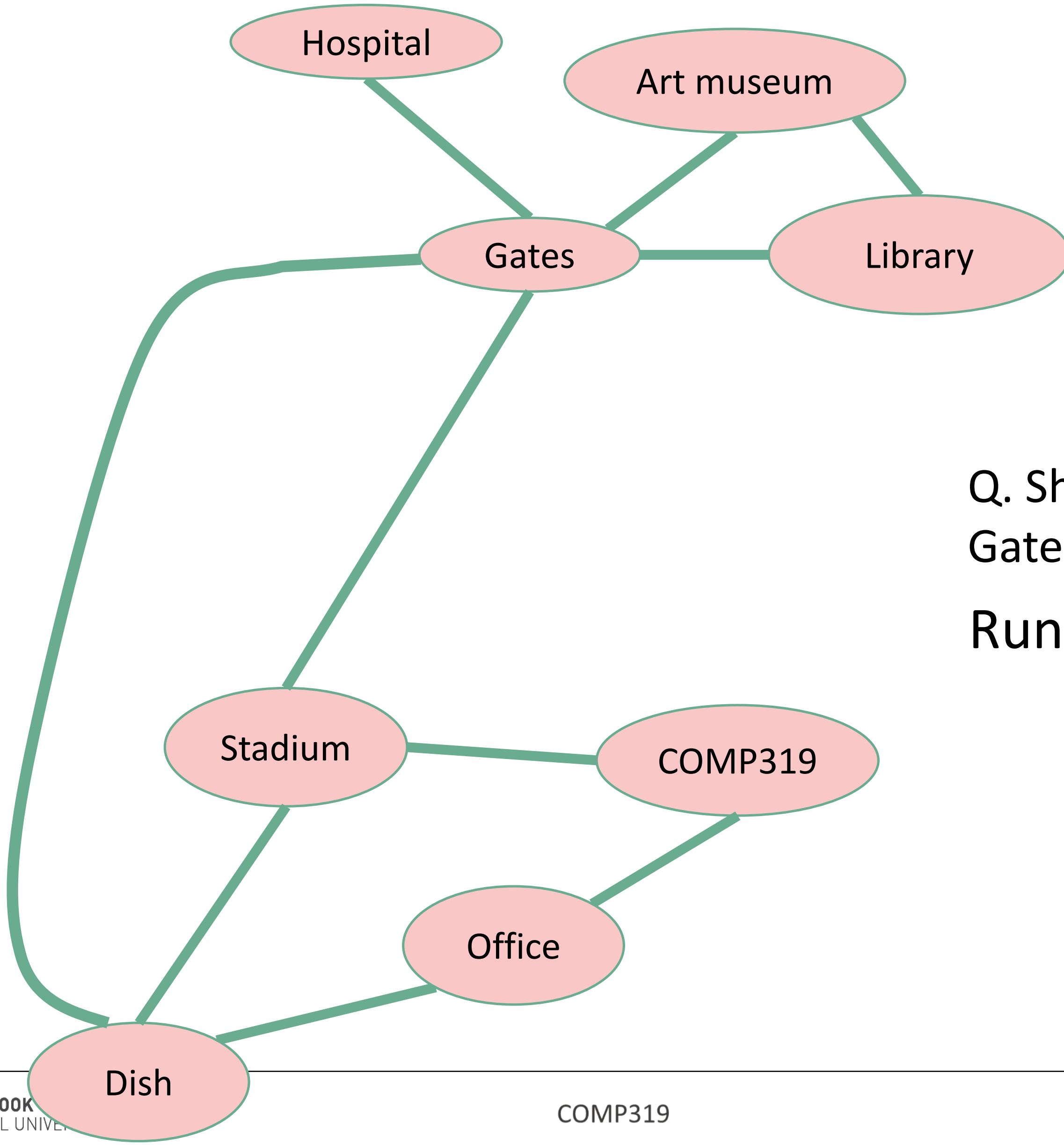
Dijkstra’s Algorithm

What if the graphs are weighted?

- Finding distance on a weighted graph
  - All nonnegative weights: Dijkstra
  - If there are negative weights: Bellman-Ford


Hospital
Art museum
Gates
Library
Stadium
COMP319
Office
Dish

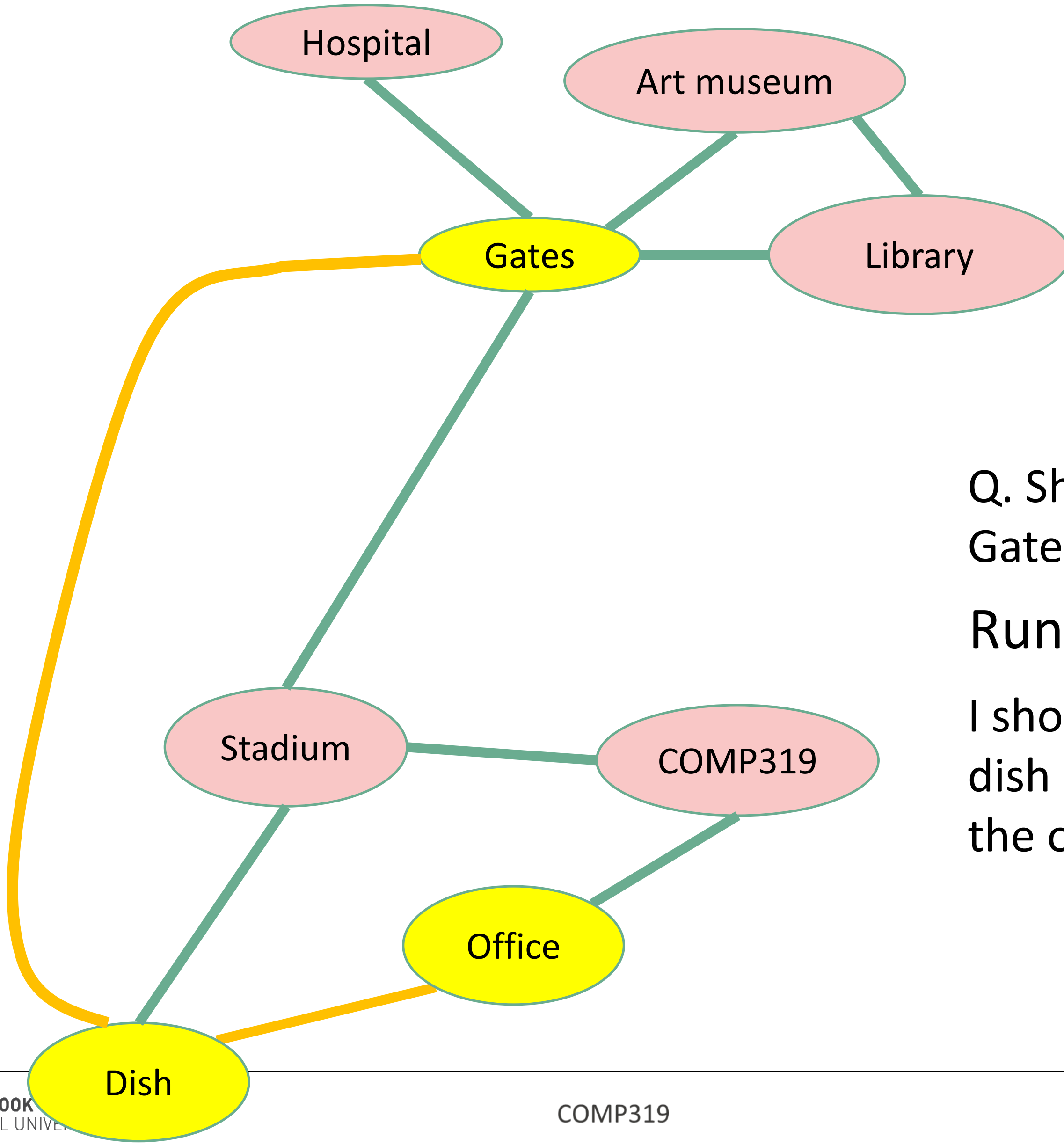
Just the graph



Q. Shortest path from Gates to the Office?

Run BFS …

Just the graph

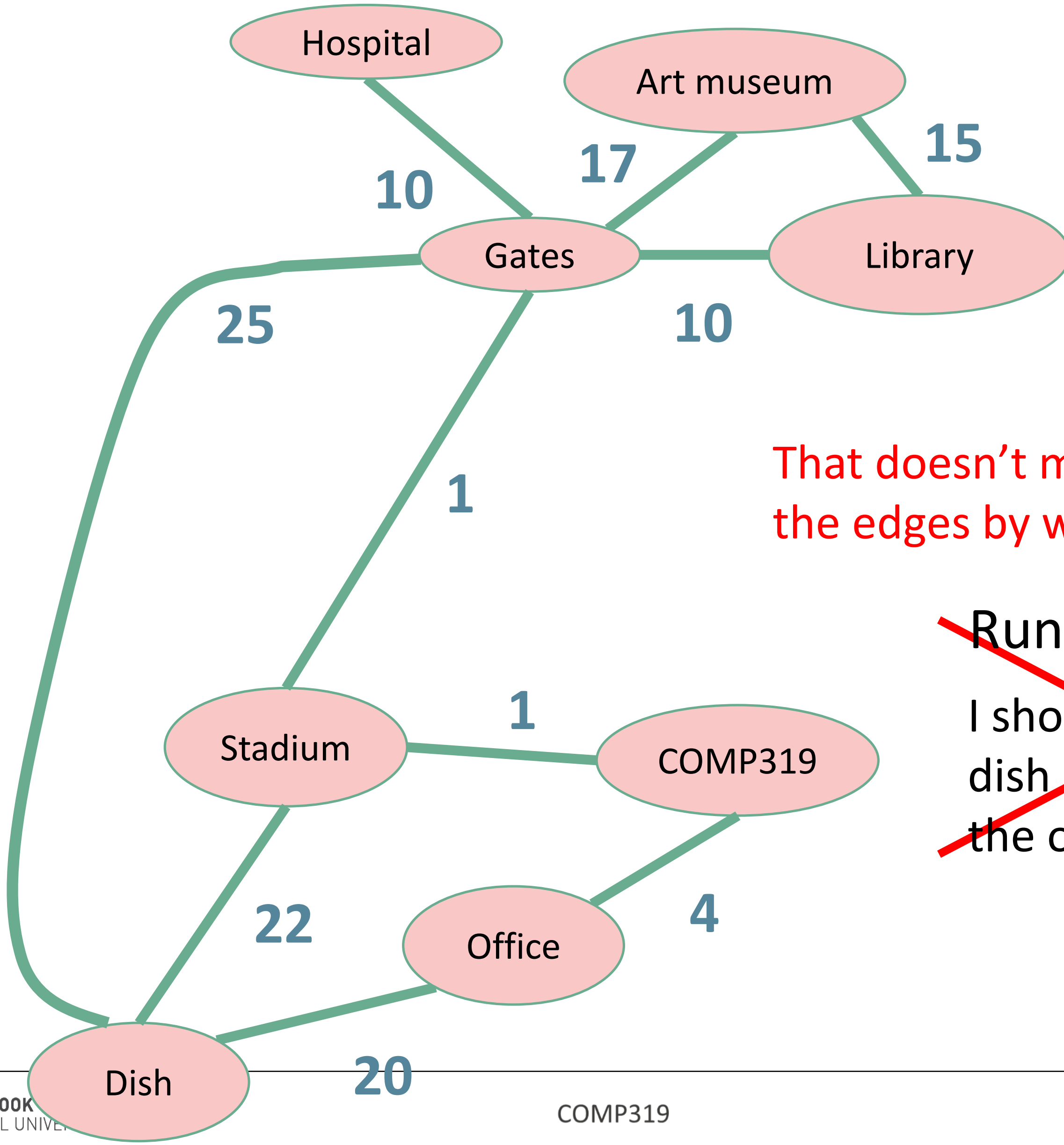


Q. Shortest path from Gates to the Office?

Run BFS …

I should go to the dish and then back to the office!

What if the graph has weights?



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

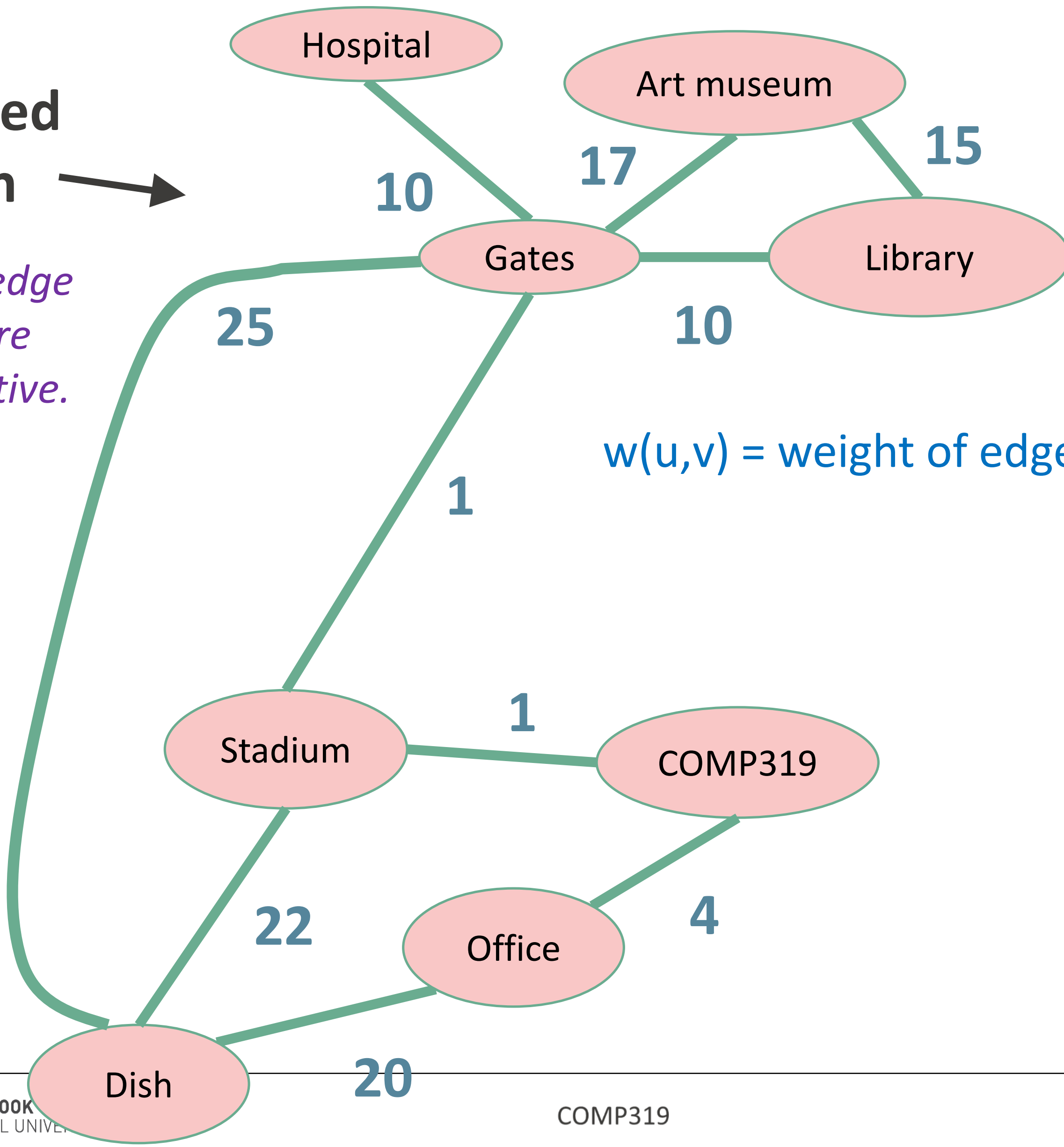
~~Run BFS …~~

~~I should go to the dish and then back to the office!~~

What if the graph has weights?

weighted graph

For now, edge weights are non-negative.

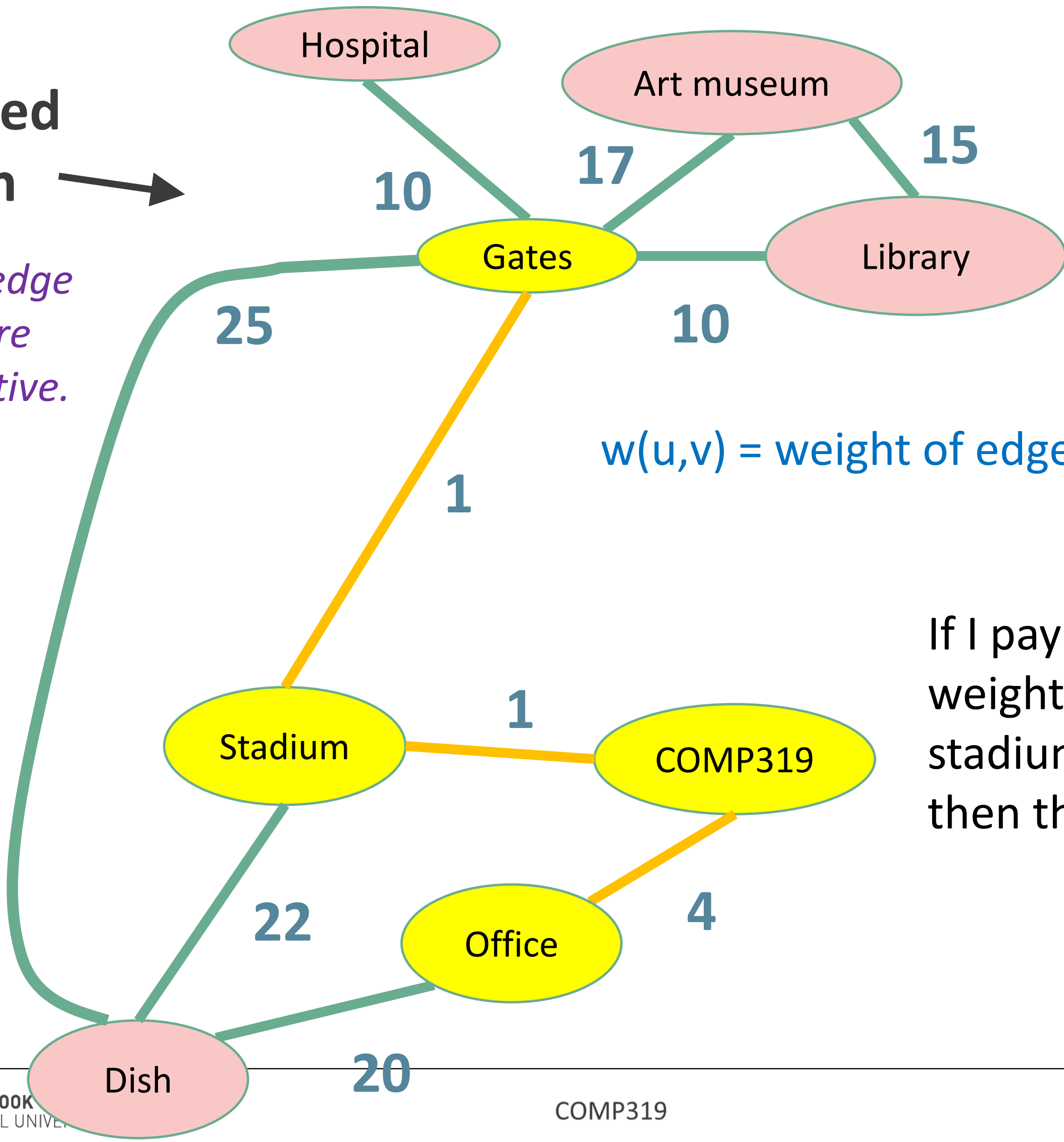


w(u,v) = weight of edge between u and v.

What if the graph has weights?

weighted graph

For now, edge weights are non-negative.

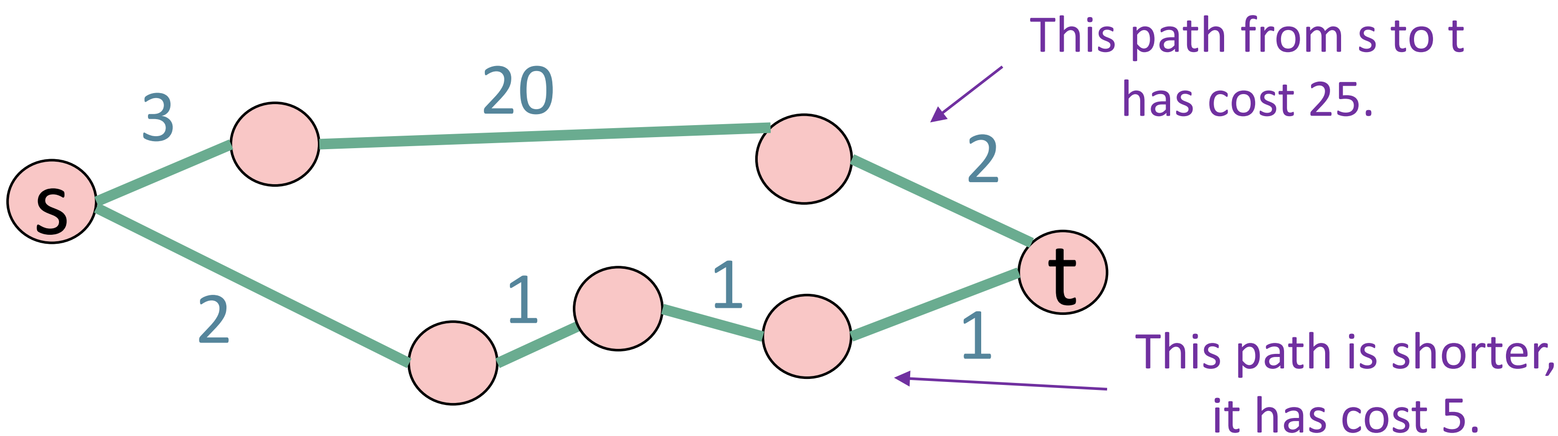


w(u,v) = weight of edge between u and v.

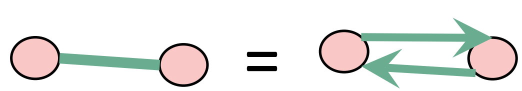
If I pay attention to the weights, I should go to stadium, then COMP319, then the Office.

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.

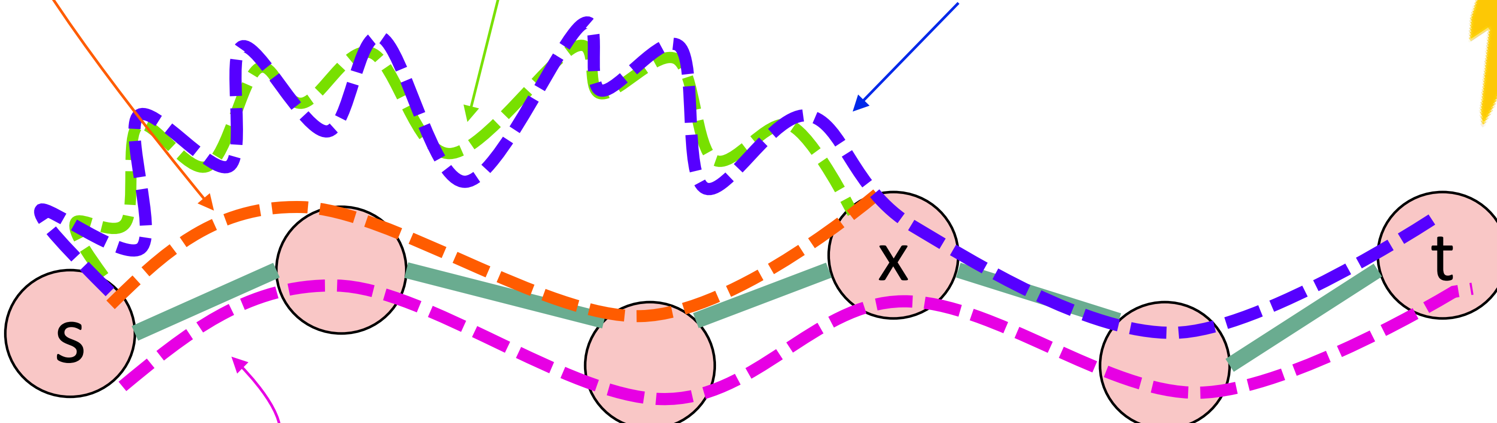


Note: For this lecture **all graphs are directed**, but to save on notation I'm just going to draw undirected edges.

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 (Gates) to all other vertices.

Destination	Cost	To get there
Stadium	1	Gates-Stadium
COMP319	2	Gates-Stadium-COMP319
Hospital	10	Gates-Hospital
Art museum	17	Gates-Art museum
Office	6	Gates-Stadium-COMP319-Office
Library	10	Gates-Stadium
Dish	23	Gates-Stadium-Dish

Example

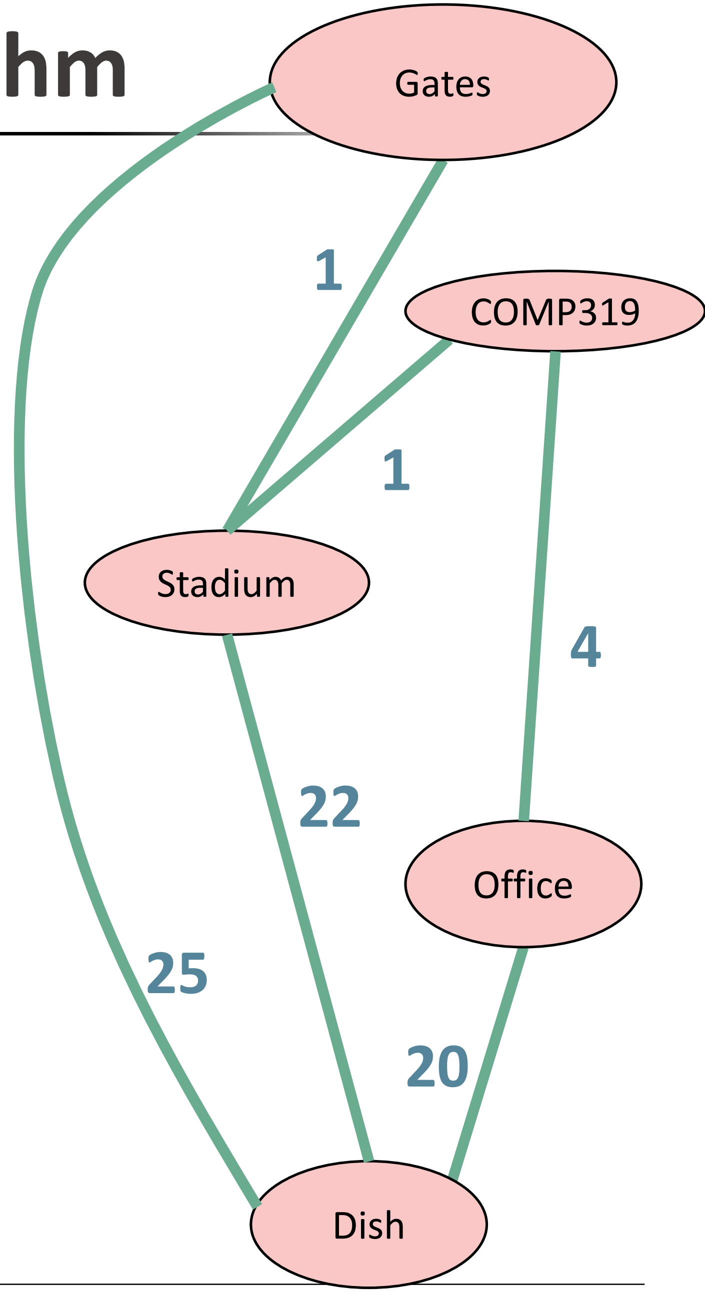
- I regularly have to solve **“what is the shortest path from home to [anywhere else]”** using Car, Bus, KTX, Flight, Taxi, Bike, Walking.
  - Edge weights have something to do with time, money, traffic, convenience.
- **Network routing**
  - I send information over the internet, from my computer to all over the world.
  - Each path has a cost which depends on link length, traffic, other costs, etc..

Back to this example

Dijkstra's algorithm

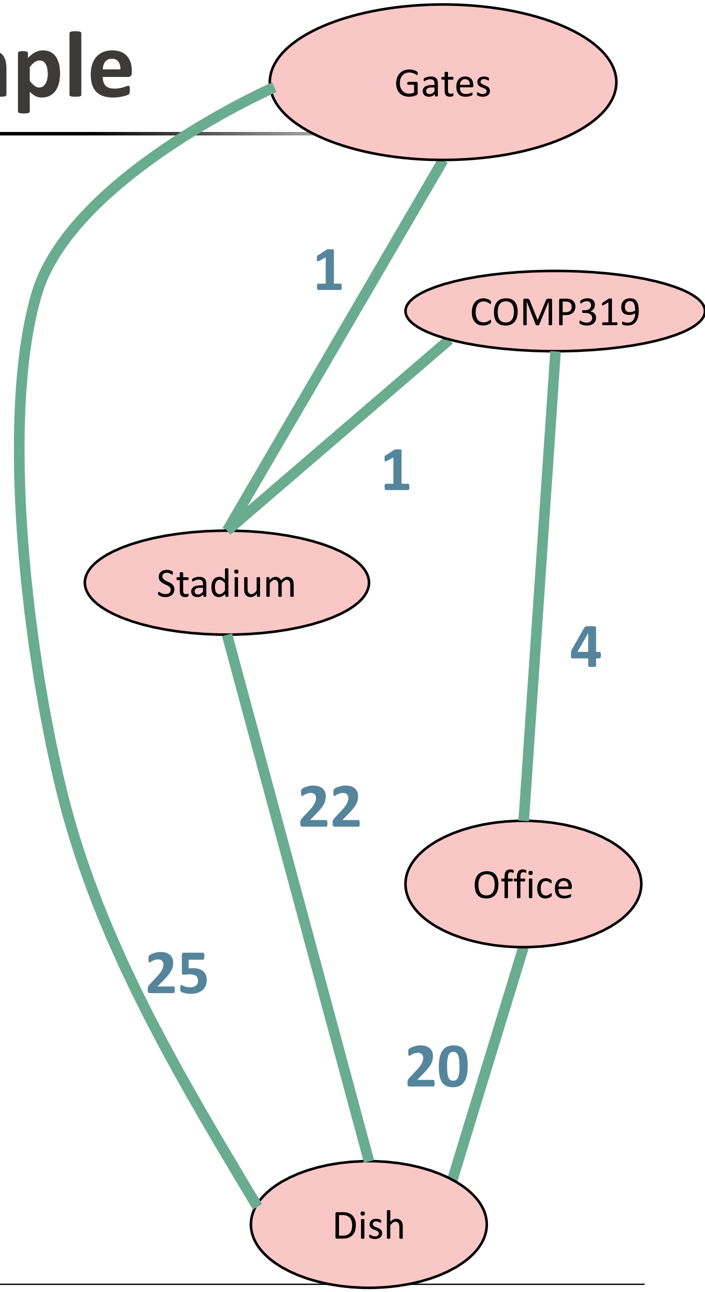
- Finds shortest paths from Gates to everywhere else.

- Dijkstra is used in practice
  - e.g., OSPF (Open Shortest Path First), a routing protocol for IP networks, uses Dijkstra.



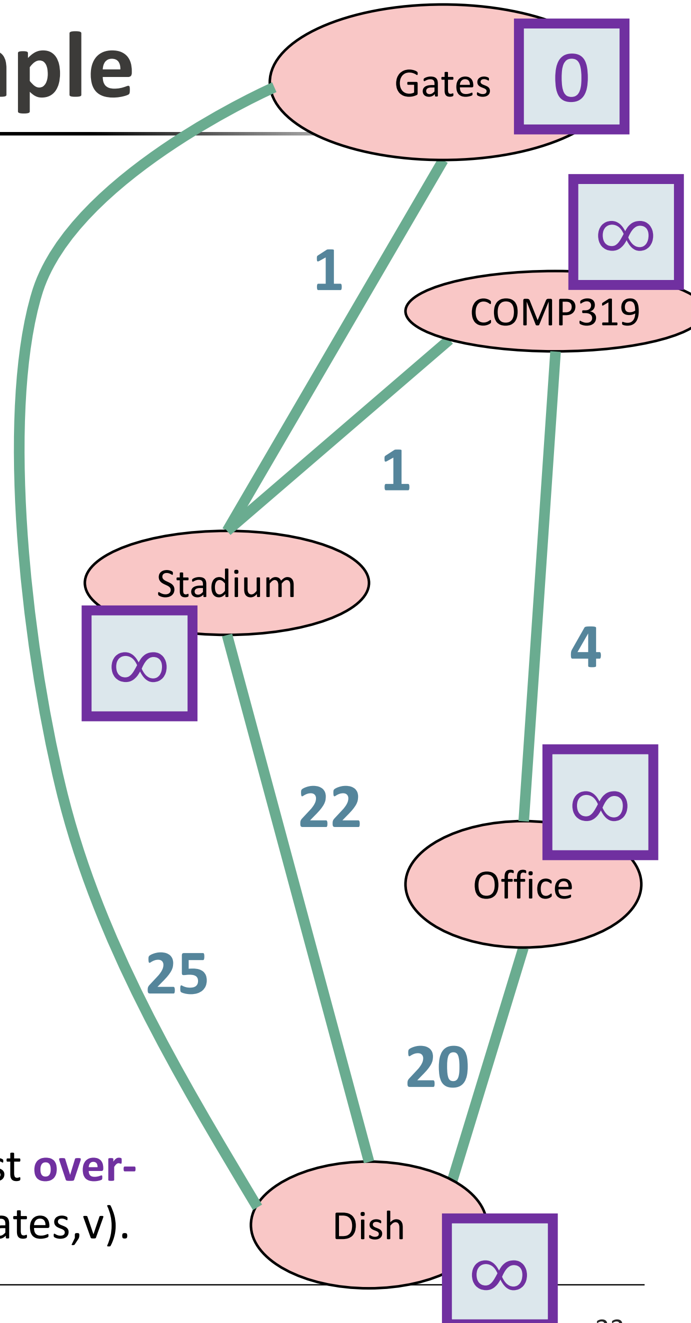
Dijkstra by example

- How far is a node from Gates?



Dijkstra by example

- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0

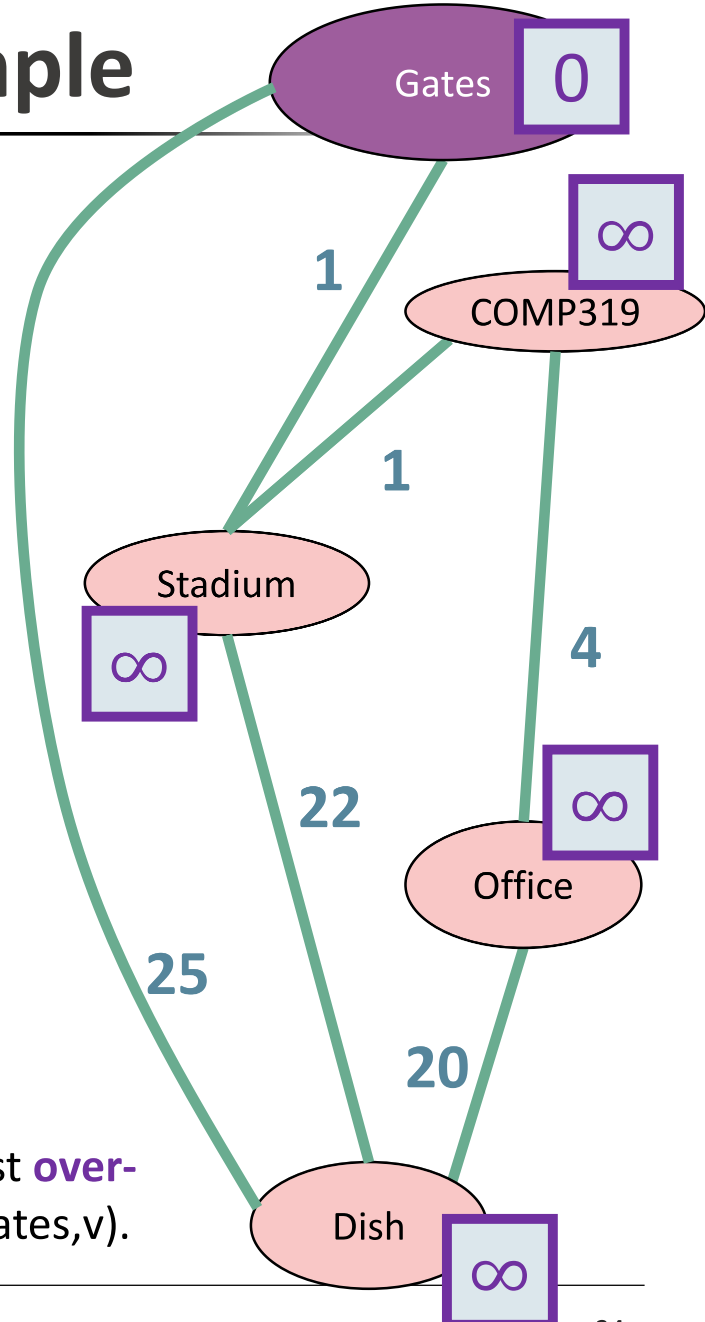


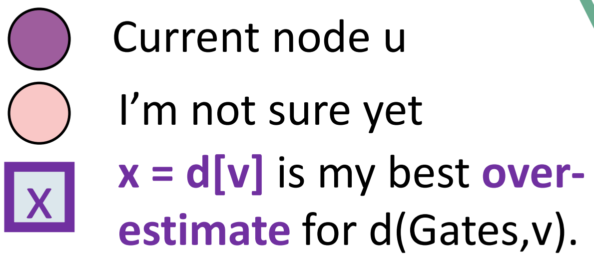
I'm not sure yet

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

Dijkstra by example

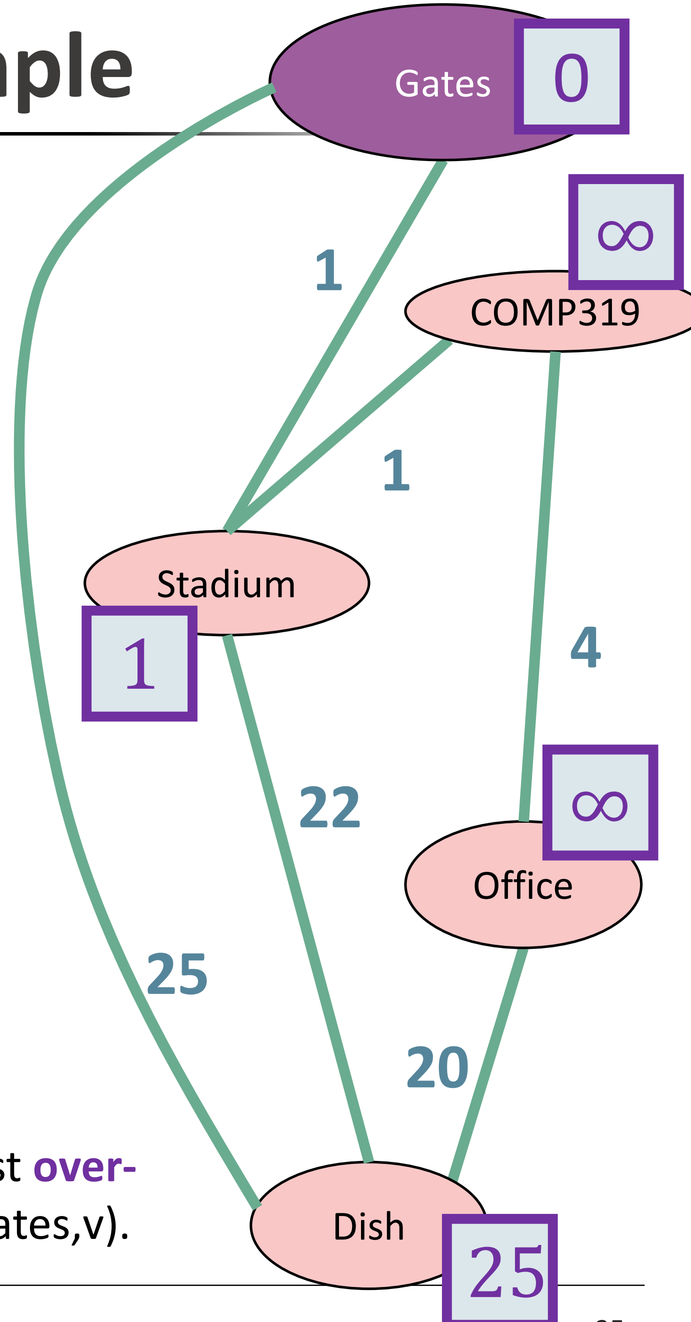
- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**





Dijkstra by example

- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))



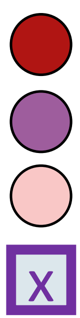
Current node u

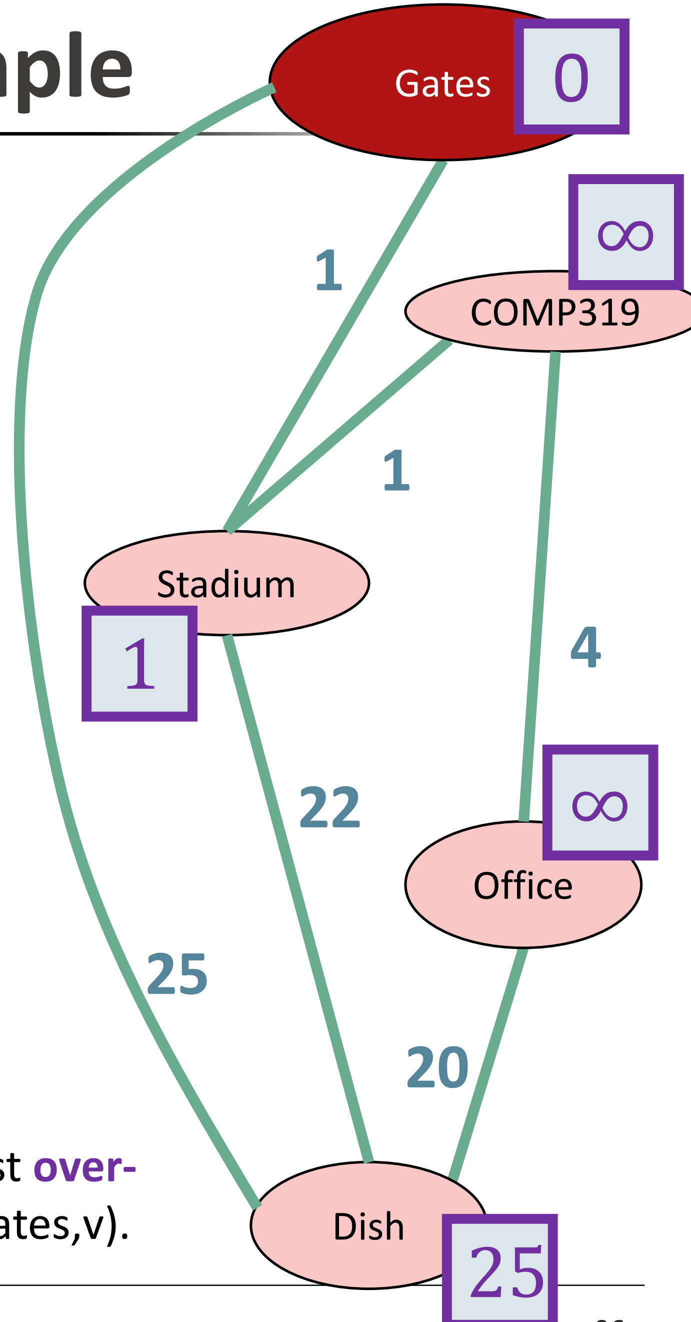
I'm not sure yet

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

Dijkstra by example

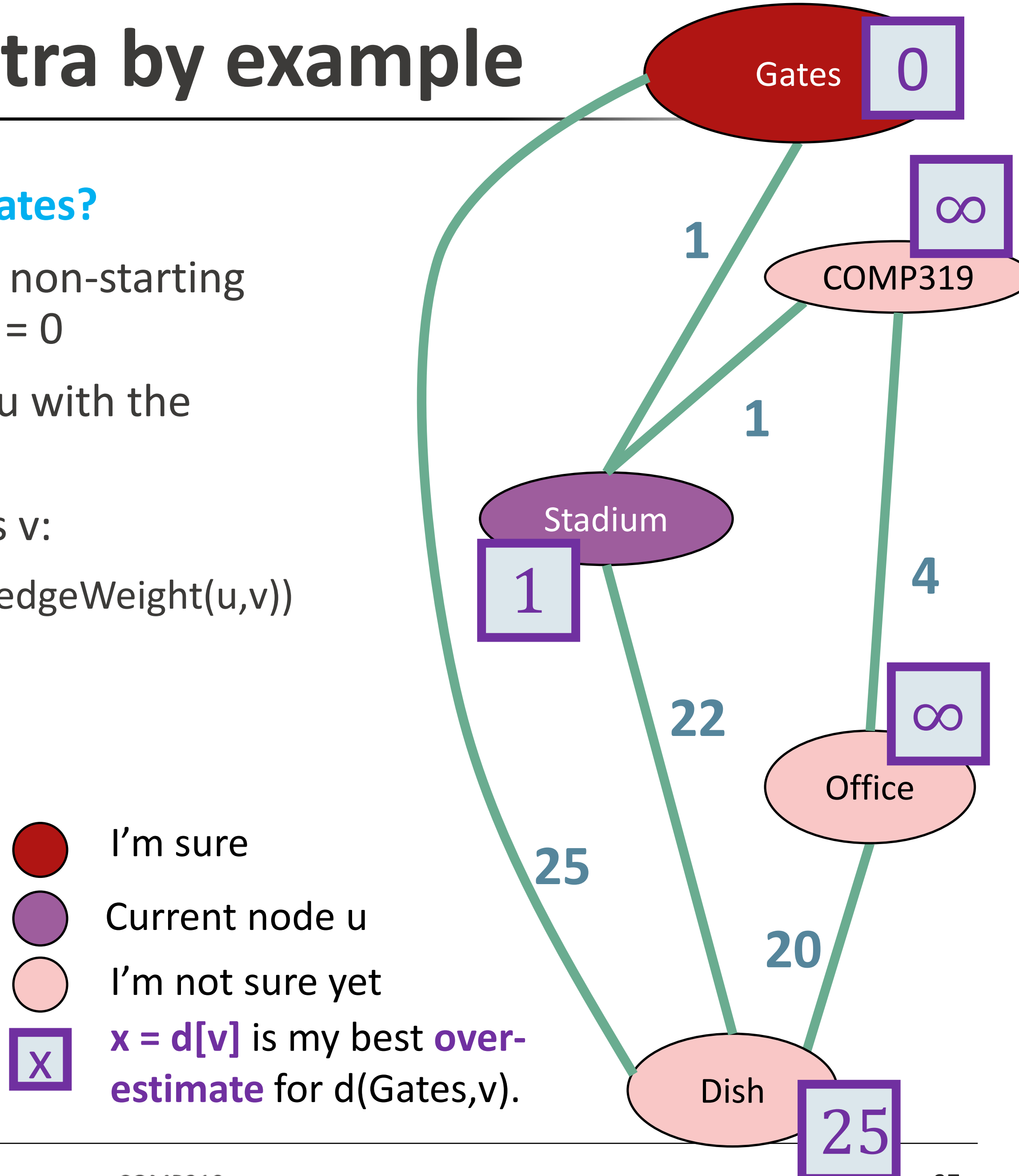
- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**





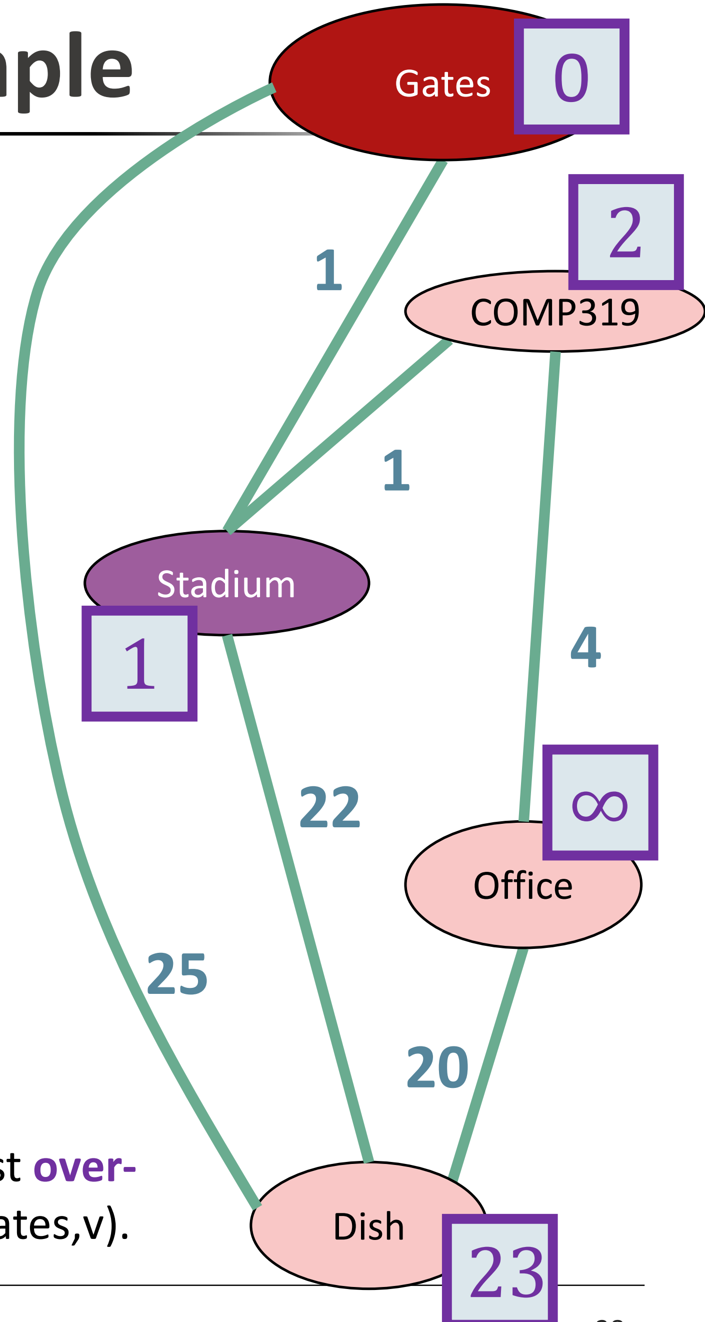
Dijkstra by example

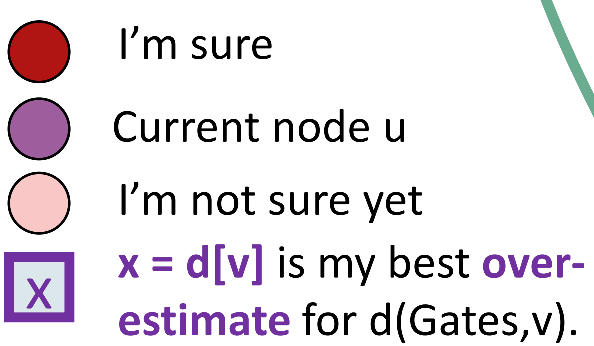
- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat



Dijkstra by example

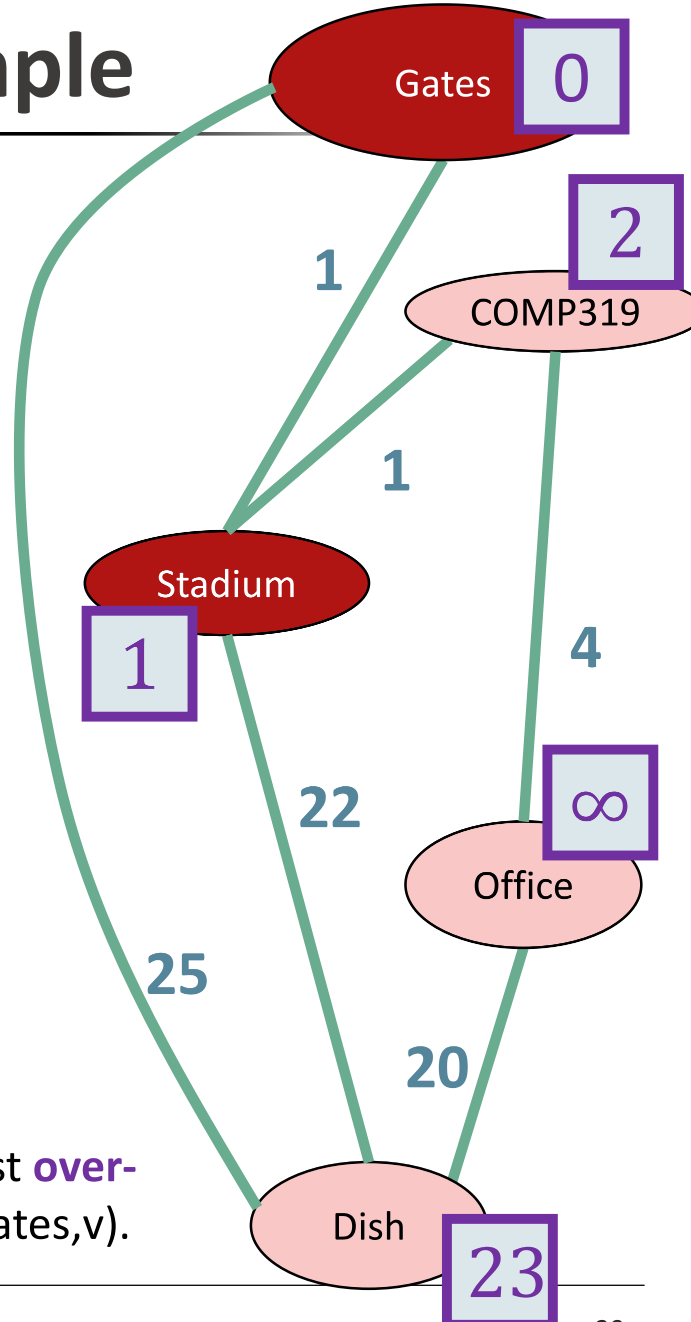
- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat





Dijkstra by example

- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat



I'm sure

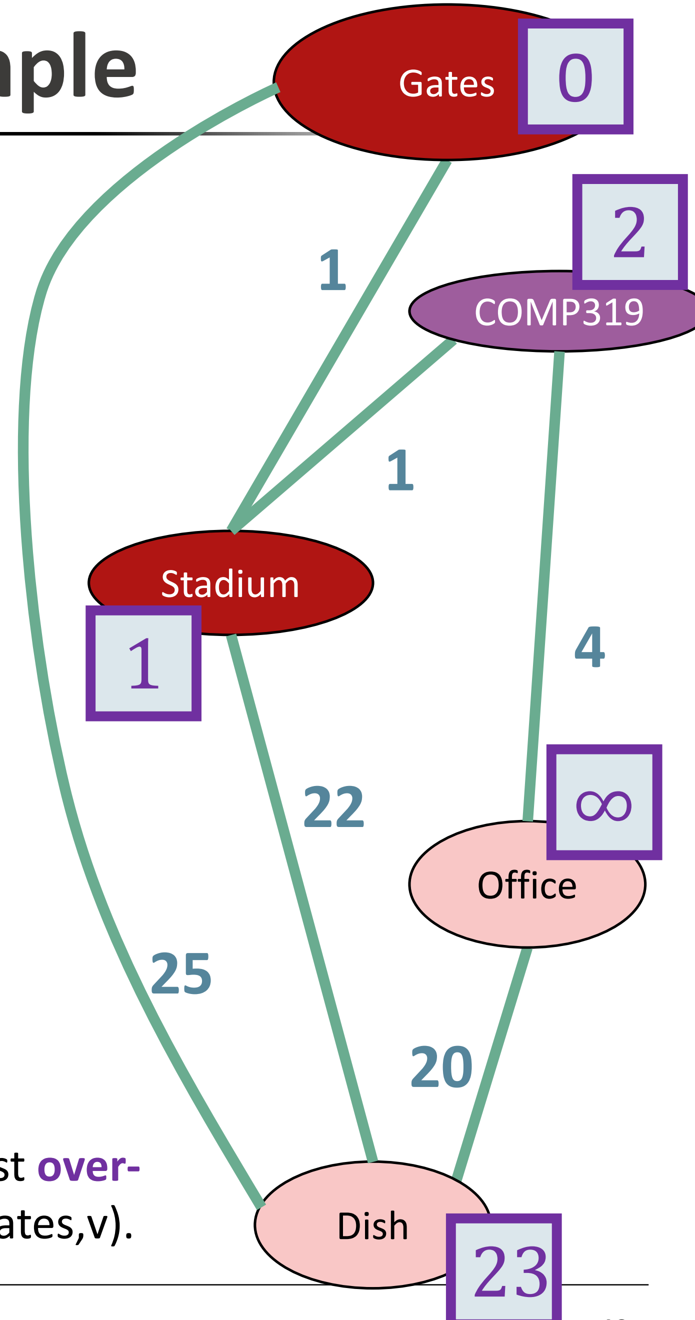
Current node u

I'm not sure yet

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

Dijkstra by example

- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat



I'm sure

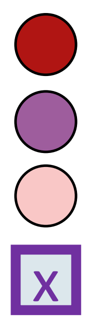
Current node u

I'm not sure yet

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

Dijkstra by example

- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u’s neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat

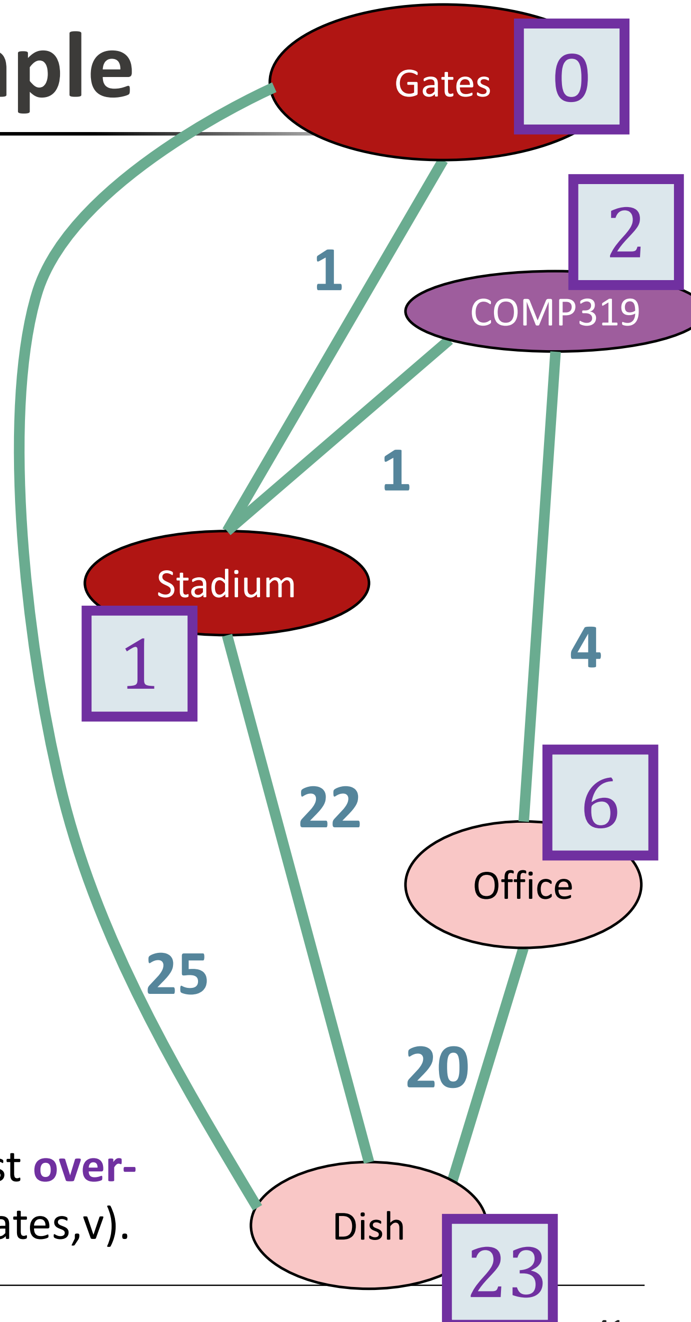


I’m sure

Current node u

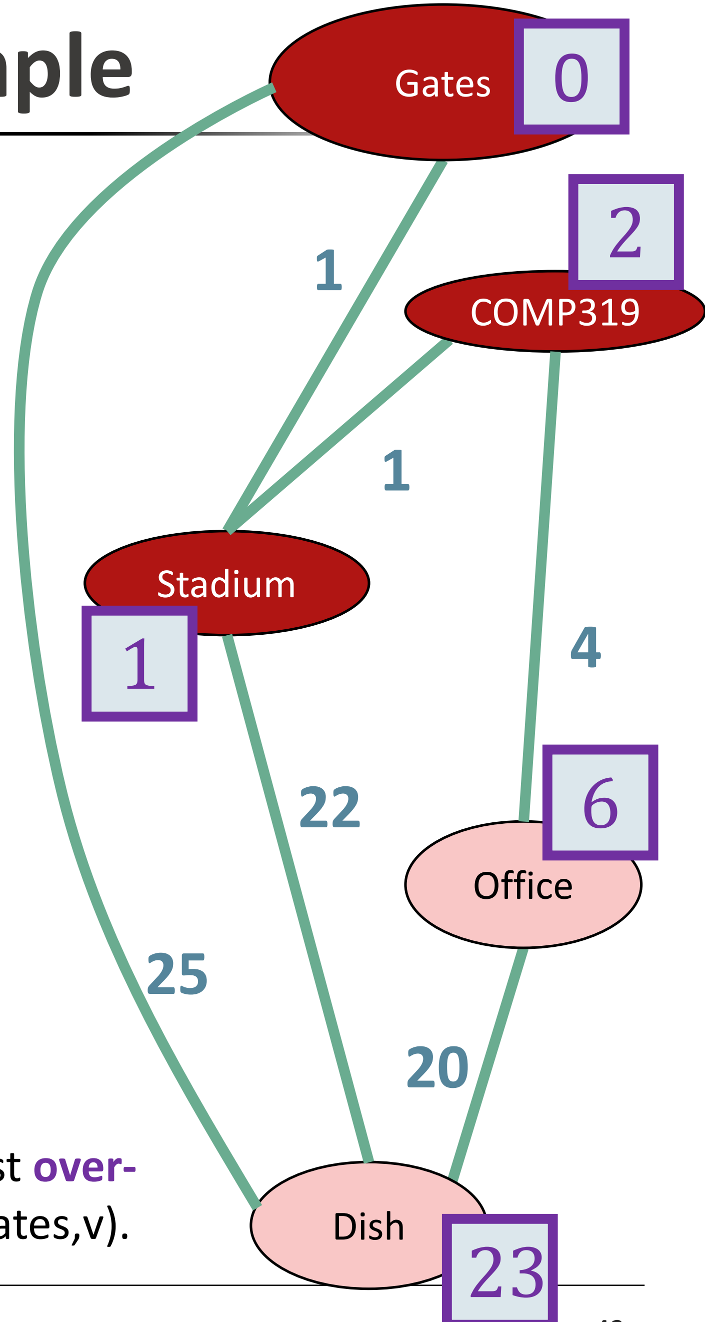
I’m not sure yet

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



Dijkstra by example

- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat



I'm sure

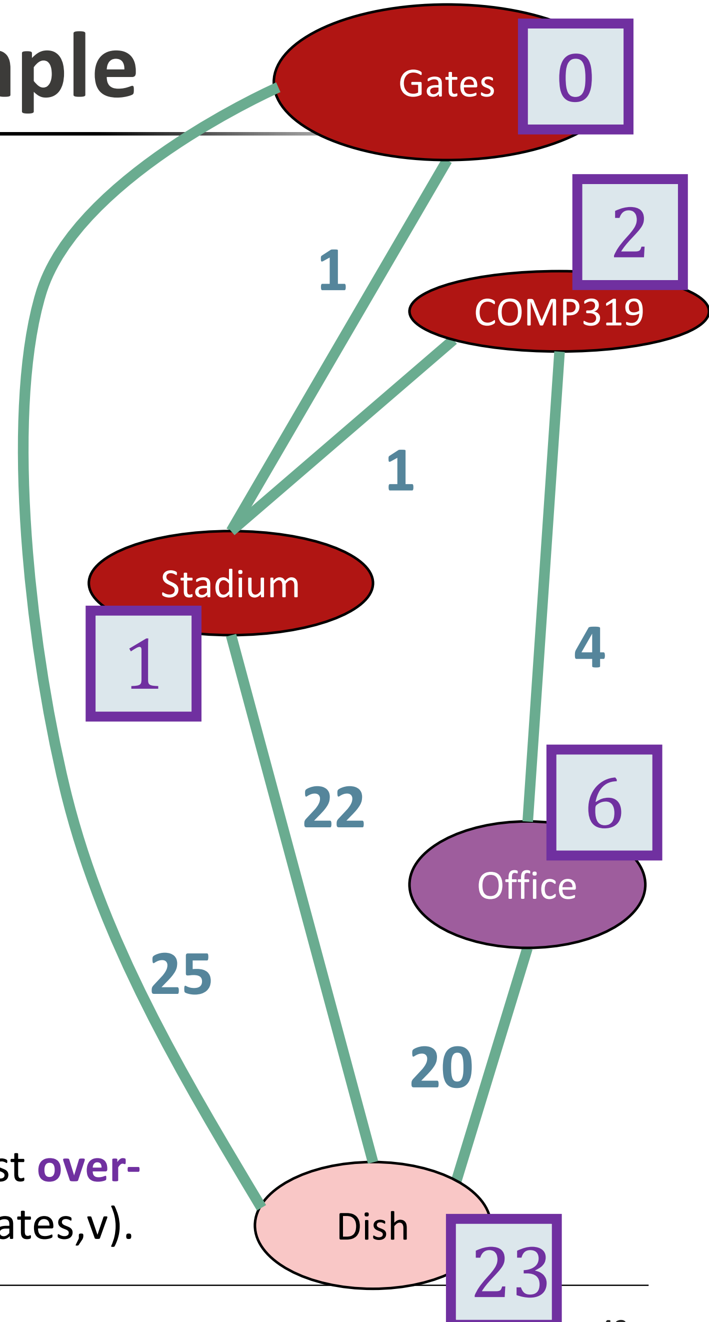
Current node u

I'm not sure yet

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

Dijkstra by example

- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat



I'm sure

Current node u

I'm not sure yet

$\boxed{x}$ **x = d[v]** is my best **over-estimate** for d(Gates,v).

Dijkstra by example

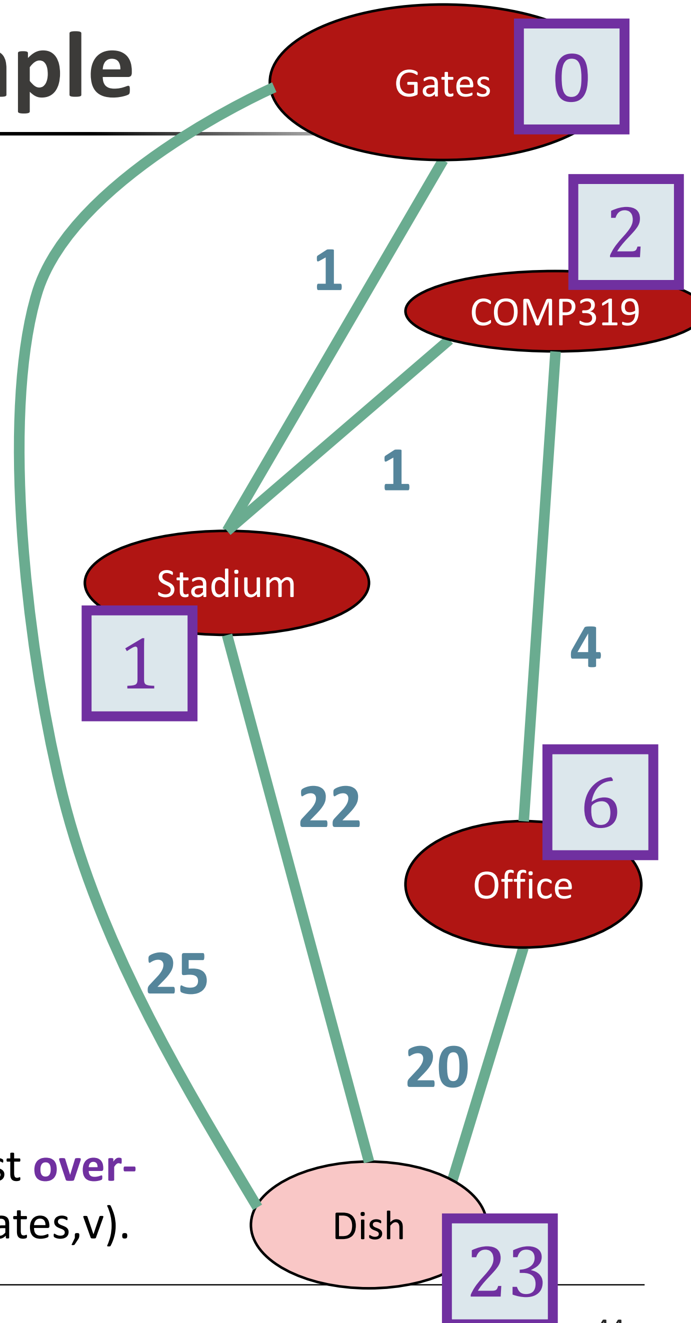
- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat

I'm sure

Current node u

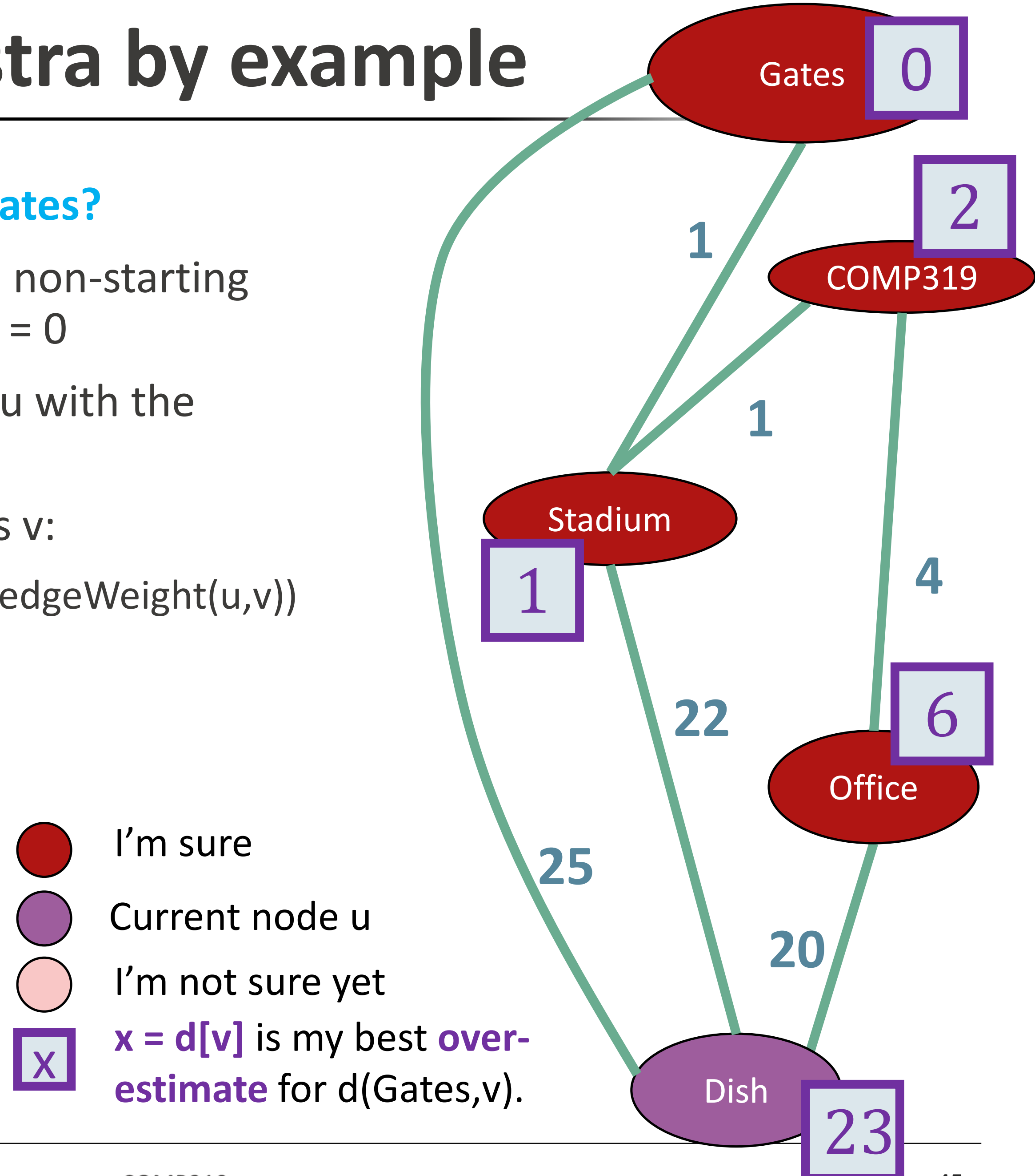
I'm not sure yet

$\boxed{x}$ **x = d[v]** is my best **over-estimate** for d(Gates,v).



Dijkstra by example

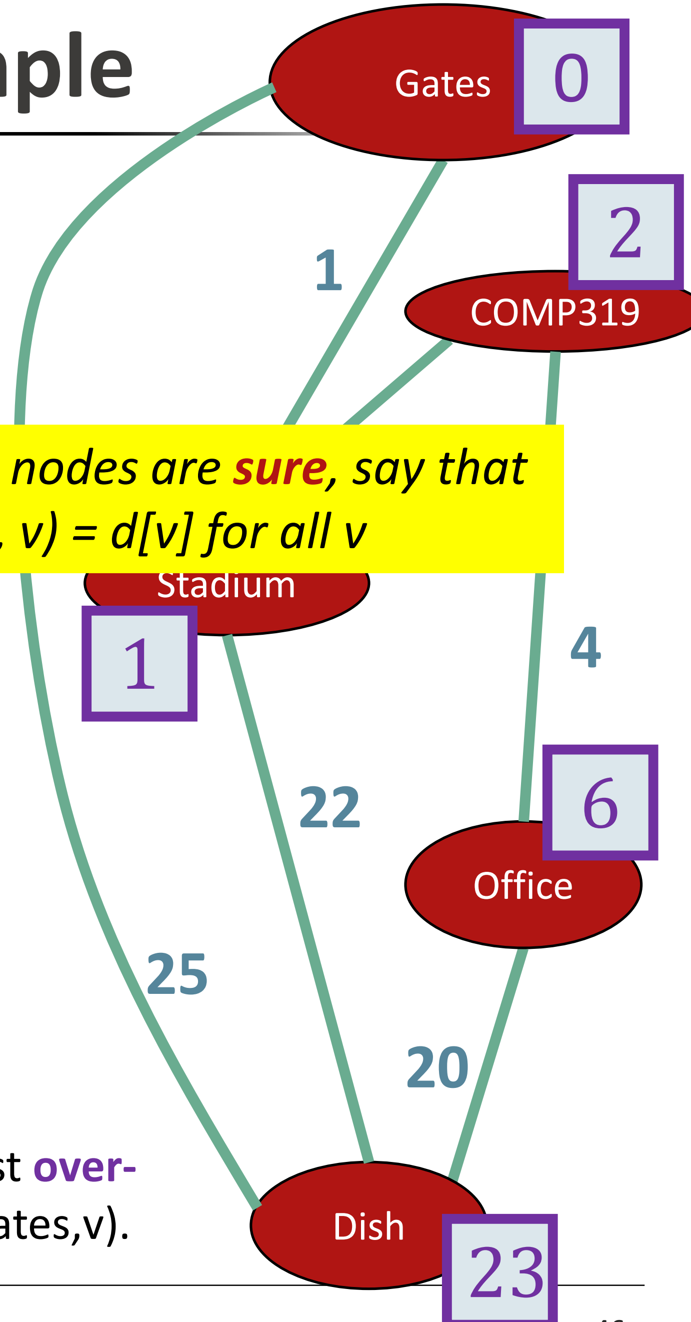
- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - Update all u's neighbors v:
    - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
  - Mark u as **sure**
  - Repeat



Dijkstra by example

- **How far is a node from Gates?**
  - Initialize d[v] = ∞ for all non-starting vertices v, and d[Gates] = 0
  - Pick the **not-sure** node u with the smallest estimate **d[u]**
  - 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(Gates, v) = d[v] for all v*



I'm sure

Current node u

I'm not sure yet

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

- As usual, we want to check:
  - Does it work?
  - Is it fast?

Several useful properties

- **Corollary 1**. If there is no path from s to v, then we have:

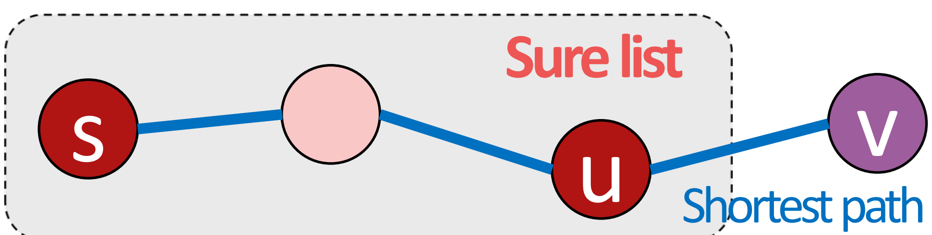
$$d[v] = d(s, v) = \infty$$

- **Lemma 1.** We always have $d[v] \geq d(s, v)$ for all v.

$$d[v] \leftarrow \min(\ \underbrace{d[v]}_{\text{Whatever path we had in mind before}},\ \underbrace{d[u] + \text{edgeWeight}(u,v)}_{\text{The shortest path to u, and then the edge from u to v.}}\ )$$

d[v] = length of the path we have in mind ≥ length of shortest path = d(s,v)

- **Lemma 2.** If s -> u -> v is a shortest path in G for some u, v in V, and if d[u] = d(s, u) at any time prior to update w(u, v),



Then,
$d[v] = d(s, u) + w(u, v) = d(s, v)$
at all time afterward.

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 equal to the shortest-path weight d(s, v) for all v.

- **Proof.** We use the following loop invariant:

> At the start of each iteration of the **while** loop, d[v] = d(s, v) for all v in **the sure list.**

  - **Initialization:** Initially, there is no v in **the sure list**, so the invariant is trivially true.

Proof cont'd – Maintenance

- Suppose that we are about to add u to **the sure list**.
- That is, we picked u in the first line here:

Recall:

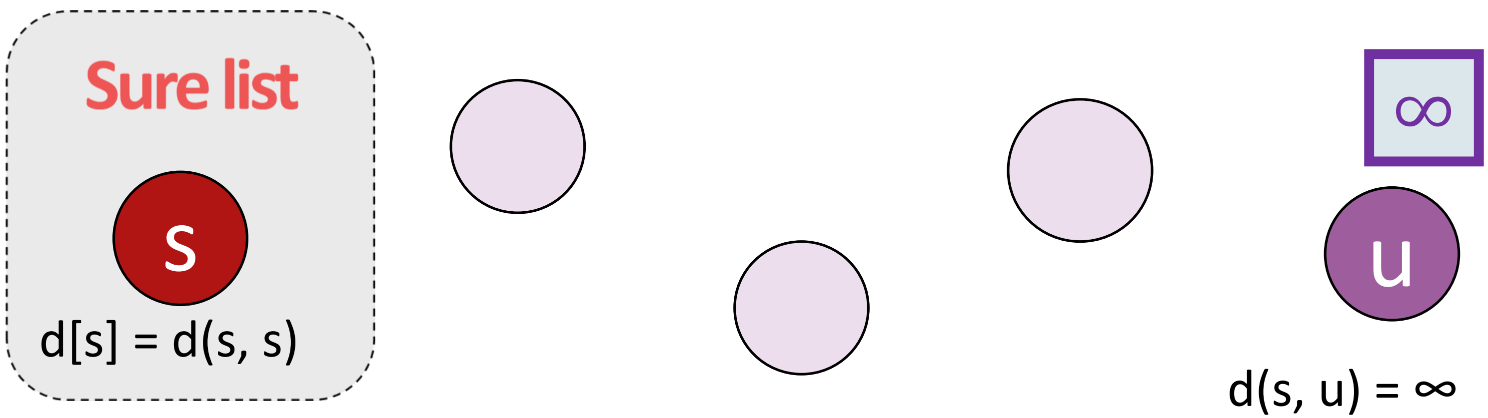
- Pick the **not-sure** node u with the smallest estimate **d[u]**
- Update all u's neighbors v:
  - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**
- Repeat

- Also suppose u is the first vertex that marked sure with d[u] != d(s, u)
  - (This is the way of contradiction.)

Proof cont'd – Maintenance

- s is the first vertex that is marked as sure,
  - At this time, d[s] = d(s, s) = 0. Thus, s ≠ u.

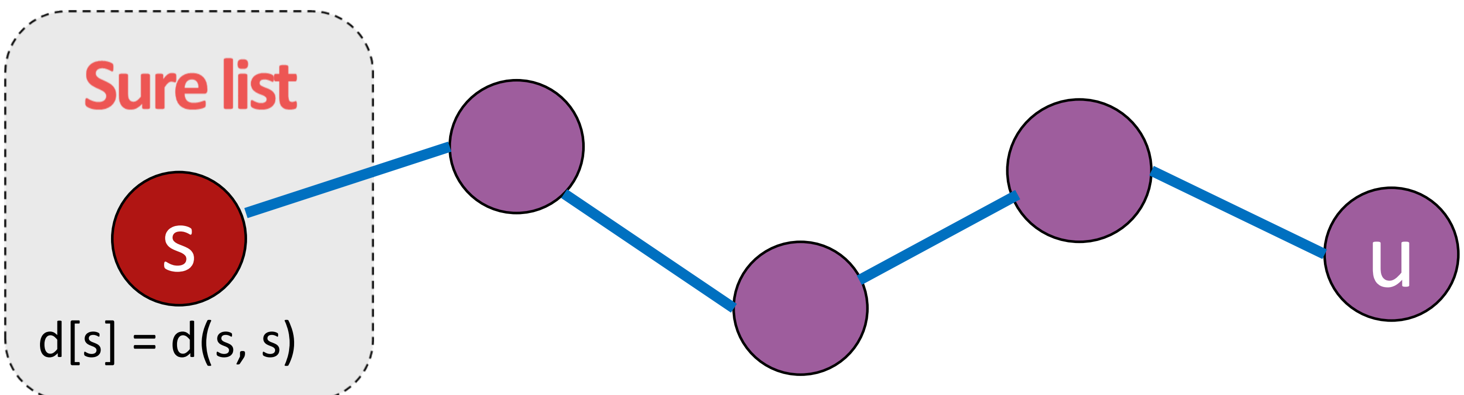
- If there is no path,
  - then, d[u] = d(s, u) = ∞, (by corollary 1) <-- violate the assumption!



Proof cont'd – Maintenance

- s is the first vertex that is marked as sure,
  - At this time, d[s] = d(s, s) = 0. Thus, s ≠ u.

- If there is no path,
  - then, d[u] = d(s, u) = ∞, (by corollary 1) <-- violate the assumption!
  - Therefore, there must exist one path and the shortest path too.

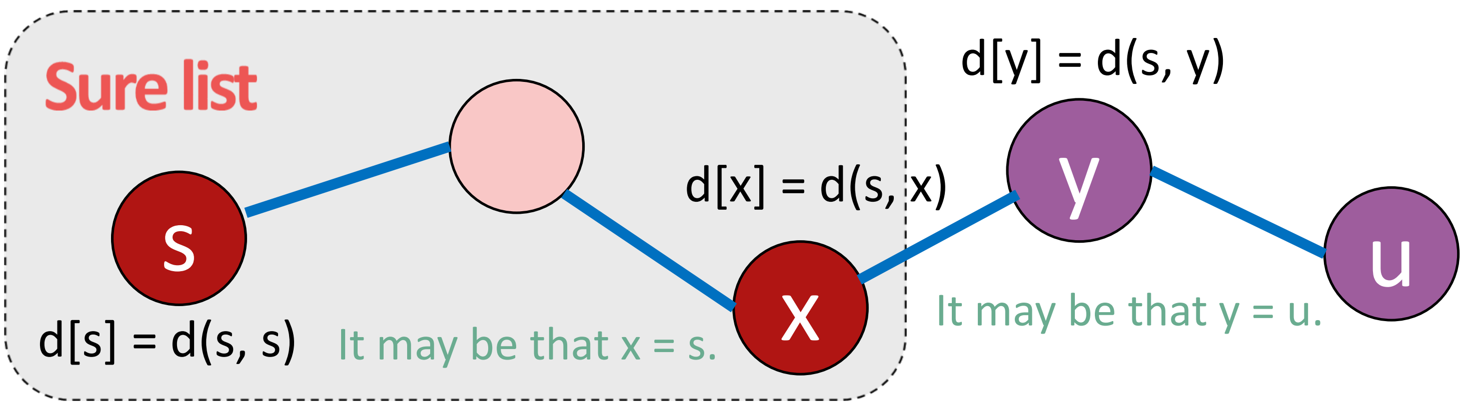


Consider this the shortest path *p*.

Proof cont'd – Maintenance

- let x, and y:
  - y be the first vertex along *p* such that y is in **the not-sure list**.
  - x be y's predecessor along *p*.
- Then,
  - d[x] = d(s, x). (By the hypothesis)
  - Also, d[y] = d(s, y). (By the Lemma 2)

u is the first vertex that marked sure with d[u] ≠ d(s, u))



Consider this the shortest path *p*.

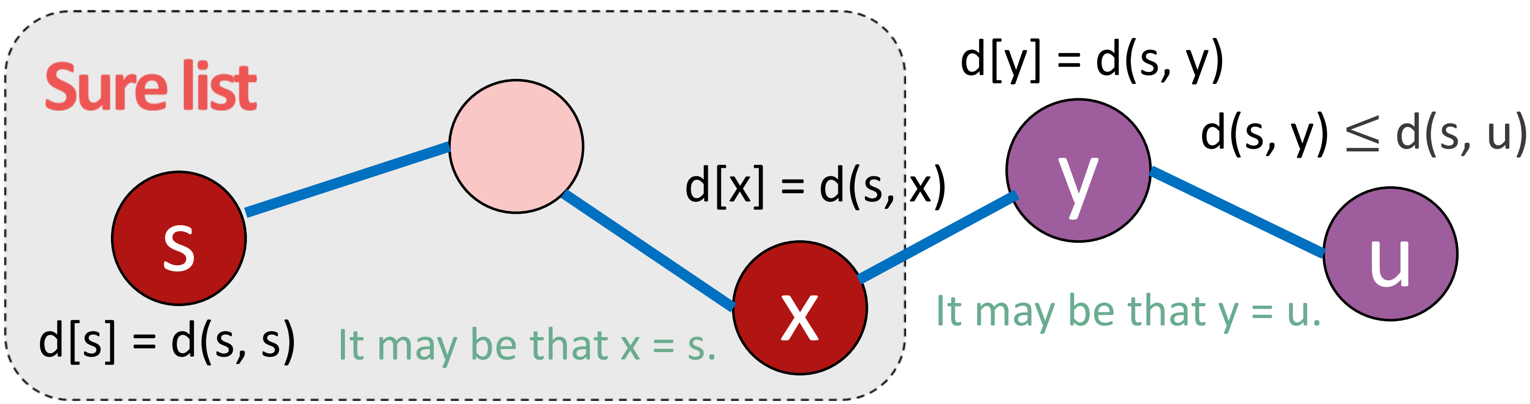
Proof cont'd – Maintenance

- Then, d[y] = d(s, y) ≤ d(s, u) ≤ d[u]

  (by Lemma 1)

  - But because both y, and u were in **the not-sure list**, when u was chosen, we have d[u] ≤ d[y].
  - Consequently, everything is equal. Thus d[u] = d(s, u). **CONTRADICTION!!**

- We conclude that d[u] = d(s, u) when u is added to **the sure list.**



Consider this the shortest path *p*.

Proof cont'd – Termination

- **Termination:**
  - At termination, **the not-sure list** = Ø which, along with our earlier invariant implies that **the sure list** is equal to V.
  - Thus, d[u] = d(s, u) for all u in V.

As usual

- Does it work?
  - Yes.

- **Is it fast?**

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 d(s, v) = d[v]

  - Operate n iterations (one per vertex)

- **How long does one iteration take?**
  - Depends on how you implement it..

We need a data structure that:

- find u with minimum d[u]
  - `findMin()`
- update (decrease) d[v]
  - `updateKey(v,d)`
- 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**

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`) + m T(`updateKey`) + T(`removeMin`))

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

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

- Running time of Dijkstra

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

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

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

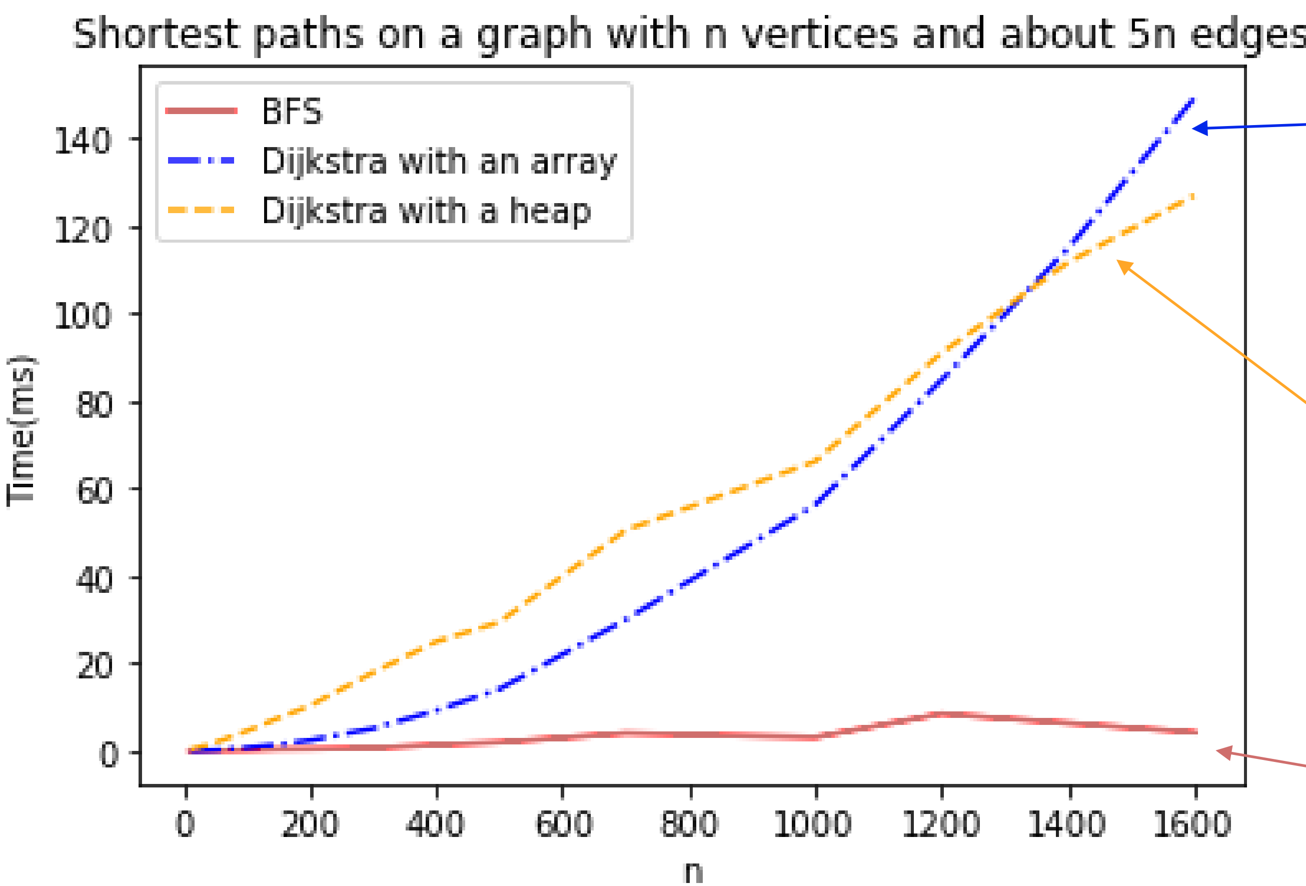
Better than an array if the graph is sparse!
aka if m is much smaller than n^2

Is a hash table a good idea here?

- **Not really**:
  - `Search(v)` is fast (in expectation)
  - But `findMin()` will still take time O(n) without more structure.

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

In practice



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

Dijkstra using a heap looks a bit more linear

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

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.

What have we learned?

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

- Along the way, it constructs a nice tree.
  - We could post this tree in Gates!
  - Then people would know how to get places quickly.


Any Question?