7/13 Lecture Agenda

- Announcements
- Part 4-1: Graphs and BFS
- 10 minute break!
- Part 4-2: Dijkstra's Algorithm

Announcements!

- Pre-HW3 due tonight!
- Pre-HW4 out tonight!
- HW3 templates and autograders
- HW2 solutions out soon, so you can study...
- HW1 grading continues, but solutions are out
- This is the last lecture in scope for the midterm!

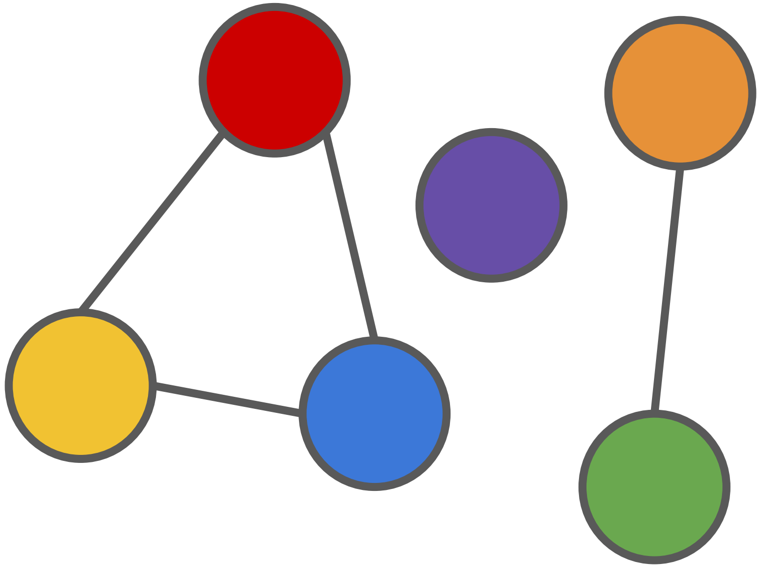
7/13 Lecture Agenda

- Announcements
- Part 4-1: Graphs and BFS
- 10 minute break!
- Part 4-2: Dijkstra's Algorithm

WORLD 4-1

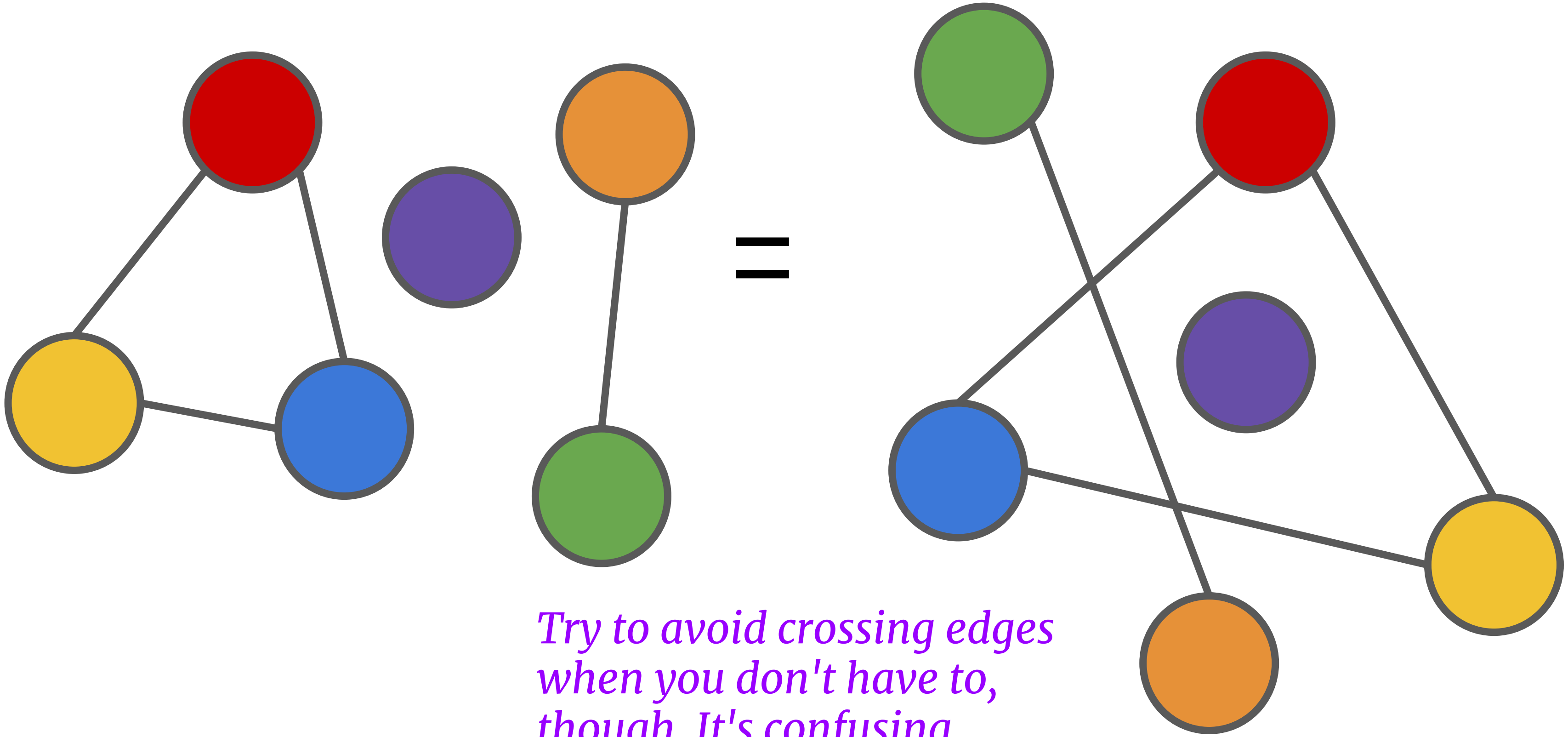
BFS: Steady And Not Slow

Graph terminology review



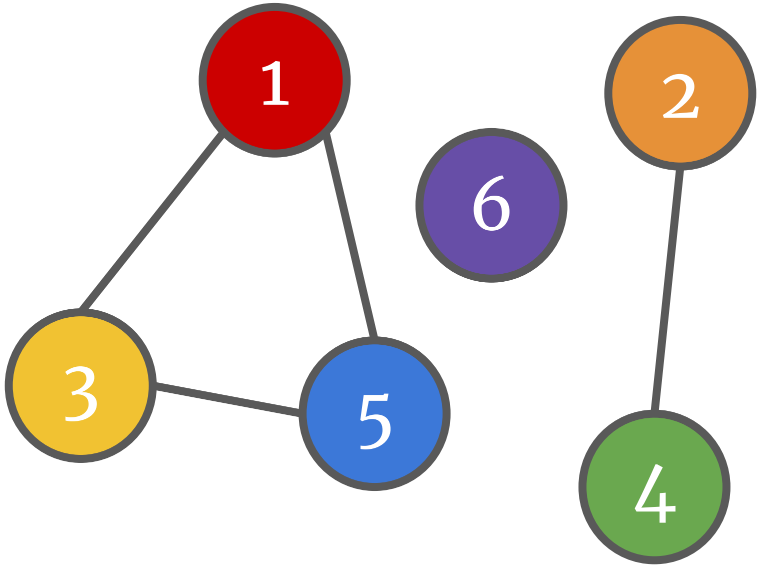
- *n* vertices (AKA *nodes*)
- *m* edges
  - There are (*n* choose 2) = *n*(*n*−1) / 2 pairs of vertices, so there can be up to that many edges as well!
  - But there could be as few as 0 edges...

The exact way a graph is drawn doesn't matter



Try to avoid crossing edges when you don't have to, though. It's confusing

Adjacency list representation

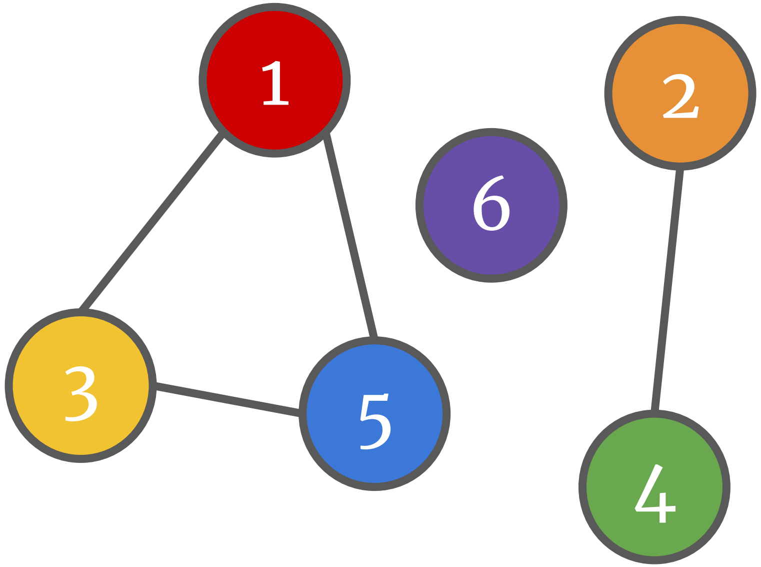


:

1: [3, 5]
2: [4]
3: [1, 5]
4: [2]
5: [1, 3]
6: []

and this could itself be an array with these six lists as values

Matrix representation (less common)

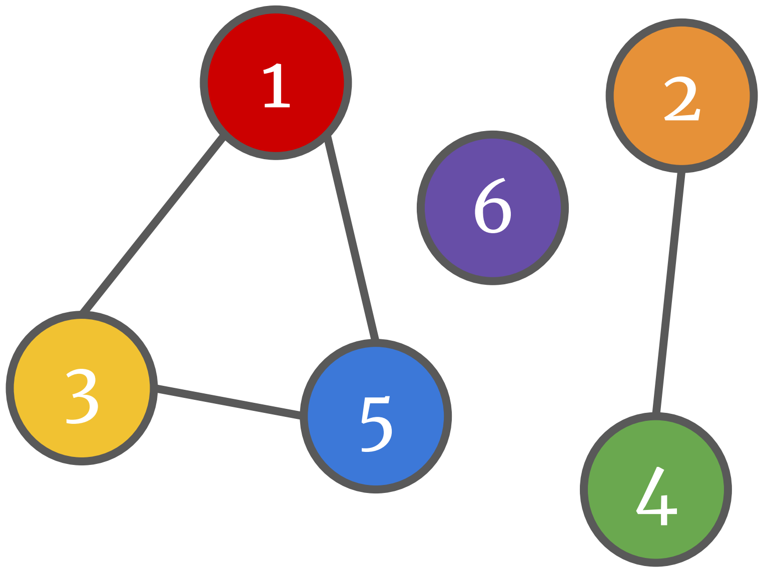


:

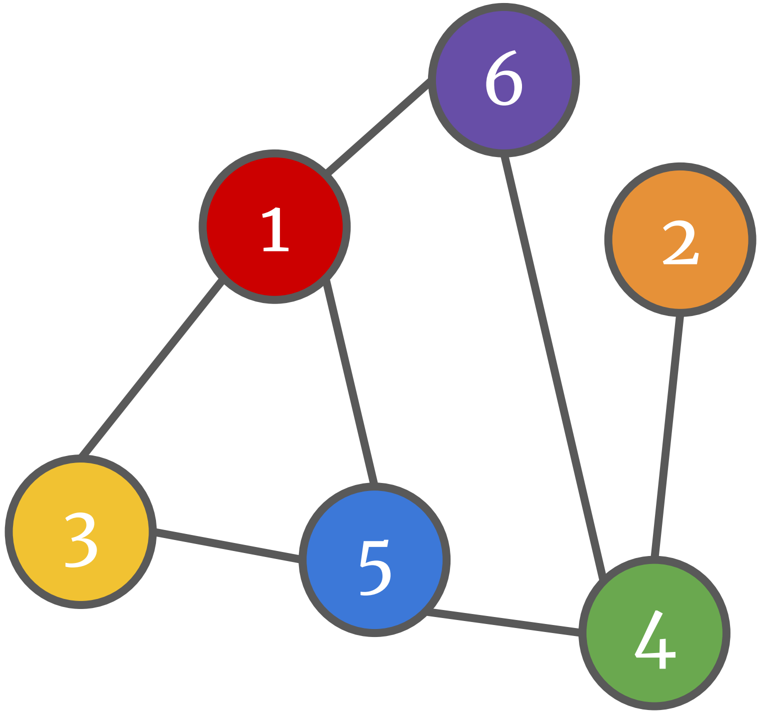
$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$O(n^2)$ storage space, though in practice we have good ways of storing "sparse" matrices

This graph has three connected components.

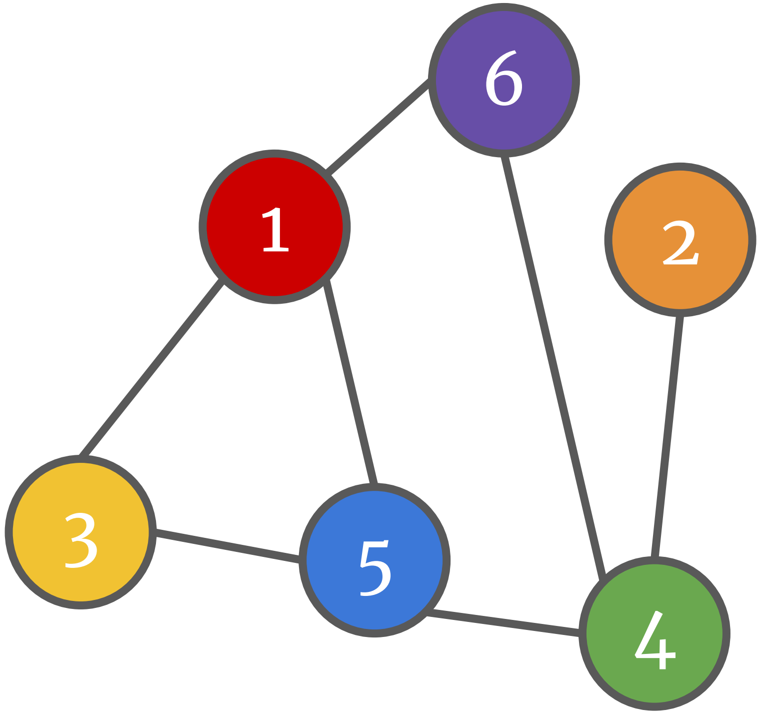


This graph is "connected" (i.e., has one connected component).



In a connected graph, there is (at least one) path between any two vertices.

This graph has three "distinct" cycles.



1-5-3-1

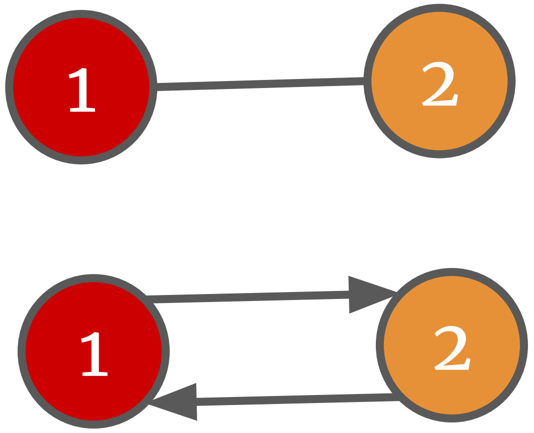
1-6-4-5-1

1-6-4-5-3-1

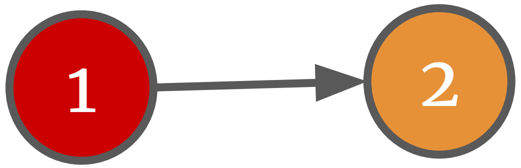
(and their reverses)

The exact definition of cycles, and counting cycles, generally won't be too important for us in 161.

So far all our edges have been undirected

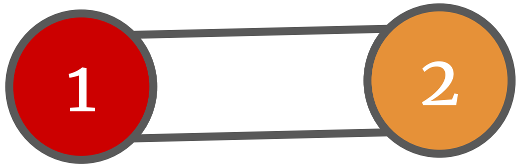


An undirected edge can be thought of as two directed edges.

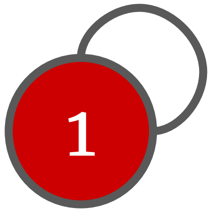


- Directed edges are one-way.
- They change the game somewhat! We'll meet them next lecture.

We'll usually pretend these don't exist

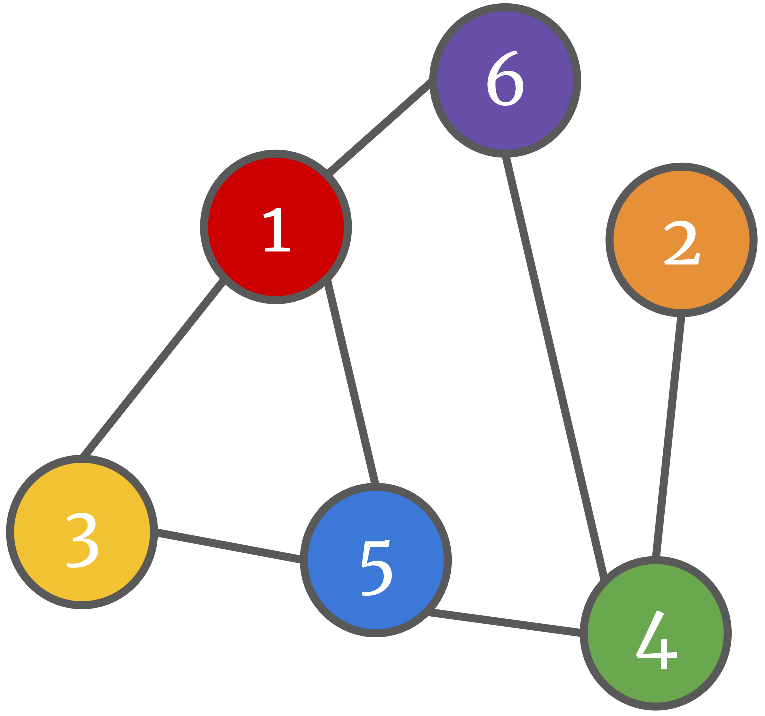


Allowing multiple edges between the same two vertices was important for Karger's Algorithm, but we generally don't think about these "multigraphs" when we talk about graphs.



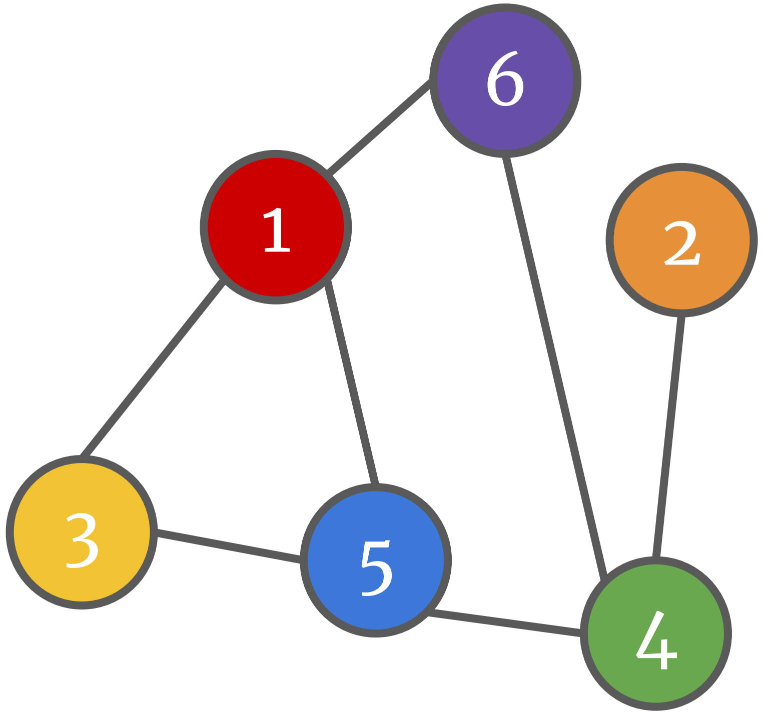
We also won't be doing this.
(i.e., no "self-loops")

How many steps apart are two vertices?



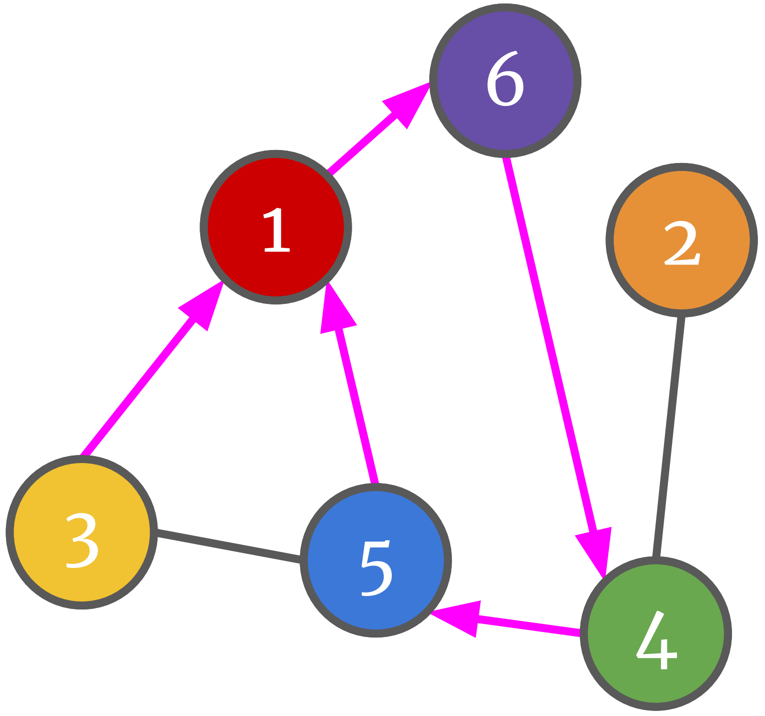
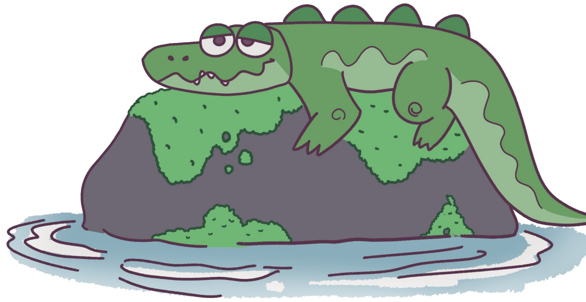
1 and 2 are at least three steps apart. (and there are two equally short paths)

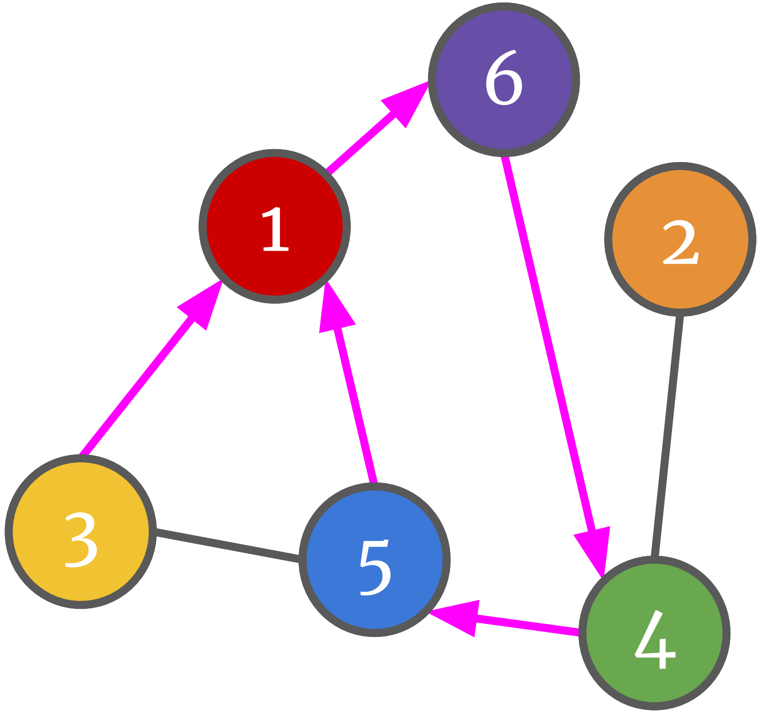
Let's start with a simpler problem: exploration



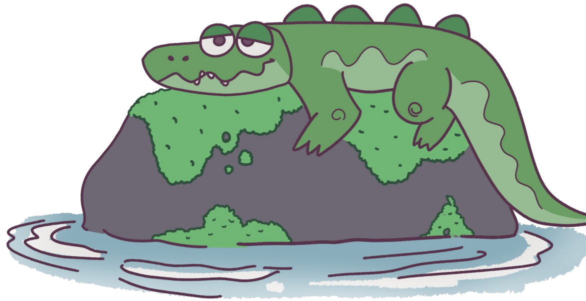
Can the yellow vertex 3 reach all other vertices?

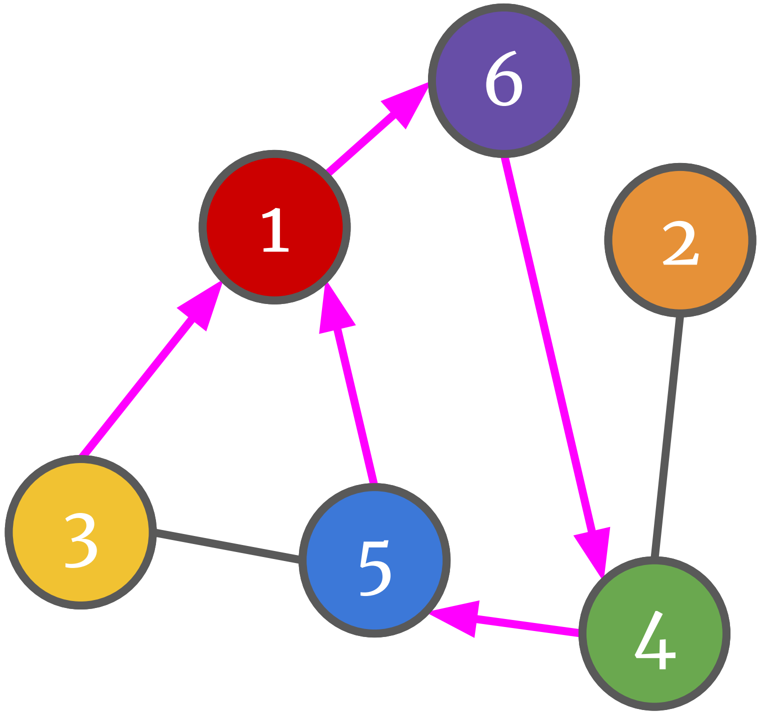
Sure, it looks visually obvious here. But imagine instead that you have a 1000000-vertex graph, as an adjacency list...


6
1
2
3
5
4


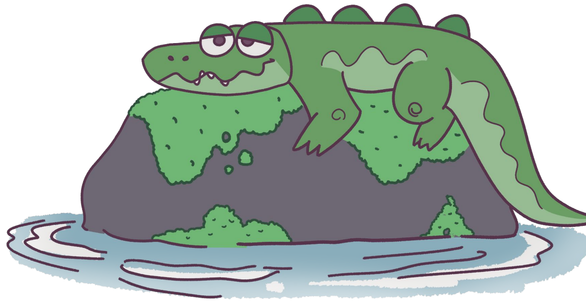


OK, I got stuck, but that's probably all of them!

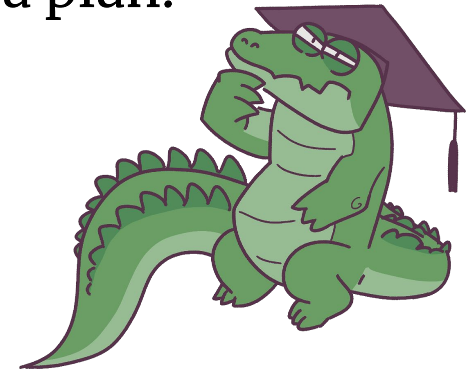




OK, I got stuck, but that's probably all of them!



If we just explore wildly, we might miss something! We need a plan.



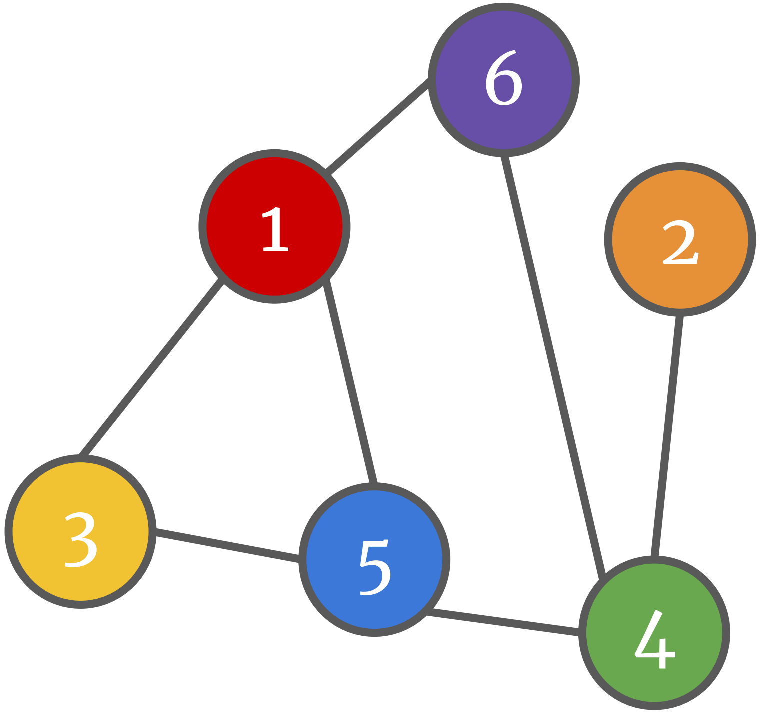
Breadth-First Traversal: The Idea

- Visit all vertices that are directly connected to the start vertex.

- Then visit all vertices that are directly connected to *those* vertices.

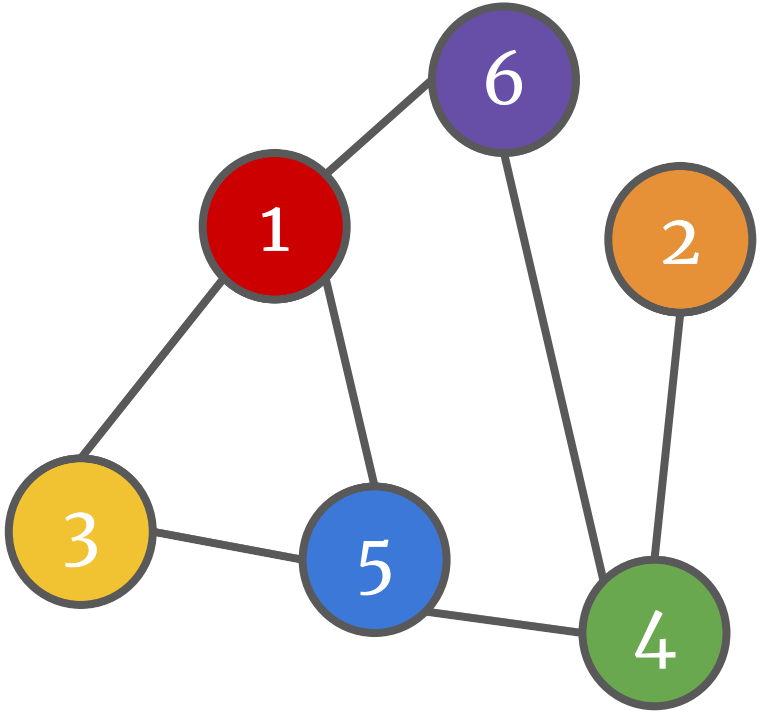
- And so on...

Breadth-First Traversal



- Create a queue of vertices to visit.
    - Initially, it just has the starting vertex.
- Repeat the following:
    - Pop off the first value in the queue.
    - Follow each of its edges (in some order), putting the vertices you find in the queue.

Breadth-First Traversal



Start: [3]

Pop 3: []

Inspect 3: it has edges to 1 and 5. Say we go in color order.

Add 1: [1]

Add 5: [1, 5]

Done with 3!

Pop 1: [5]

Inspect 1: it has edges to 3, 5, and 6.

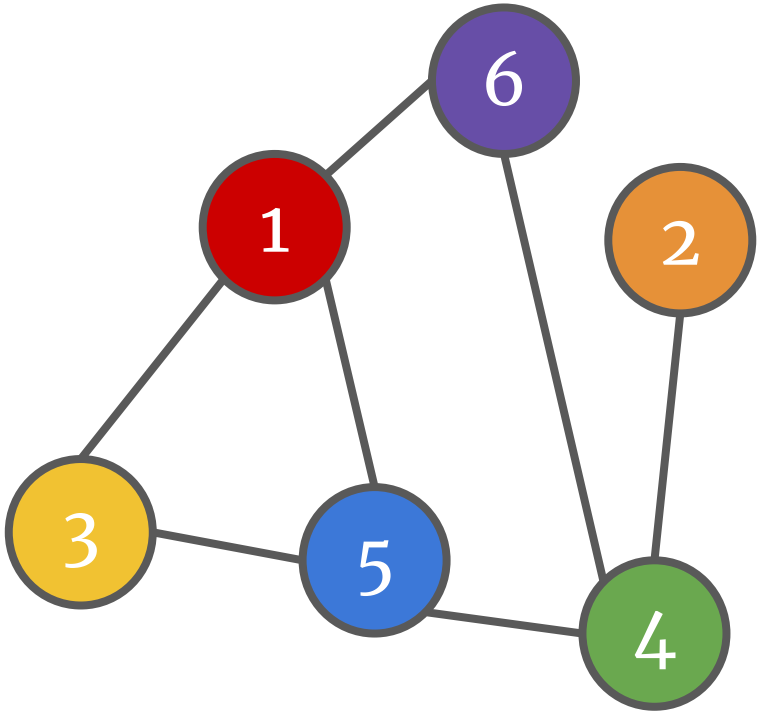
Add 3: [5, 3]... *wait a minute*

Avoiding Revisits!

- Once we've visited a vertex and followed all its edges, there is never a reason to go back.
  - If we already found it earlier, our new way of finding it is just a slower way of getting there!

- We can maintain a single Boolean variable per vertex to indicate whether we've visited it.

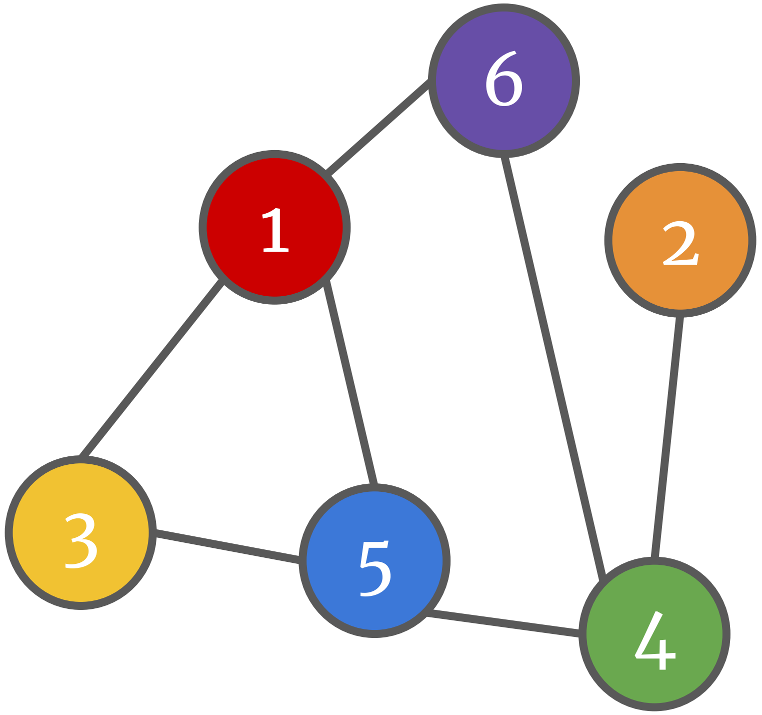
Breadth-First Traversal



- Create a queue of vertices to visit.
    - Initially, it just has the starting vertex.

- Repeat the following:
    - Pop off the first value in the queue.
    - Mark it as VISITED.
    - Follow each of its edges (in some order), putting the vertices you find in the queue as long as they are not VISITED.

Breadth-First Traversal



Start: [3]

Pop 3: []

Mark 3 as VISITED.

Inspect 3: it has edges to 1 and 5.

Add 1: [1]

Add 5: [1, 5]

Done with 3!

Pop 1: [5]

Mark 1 as VISITED.

Inspect 1: it has edges to 3, 5, and 6.

Do not add 3, since it is VISITED.

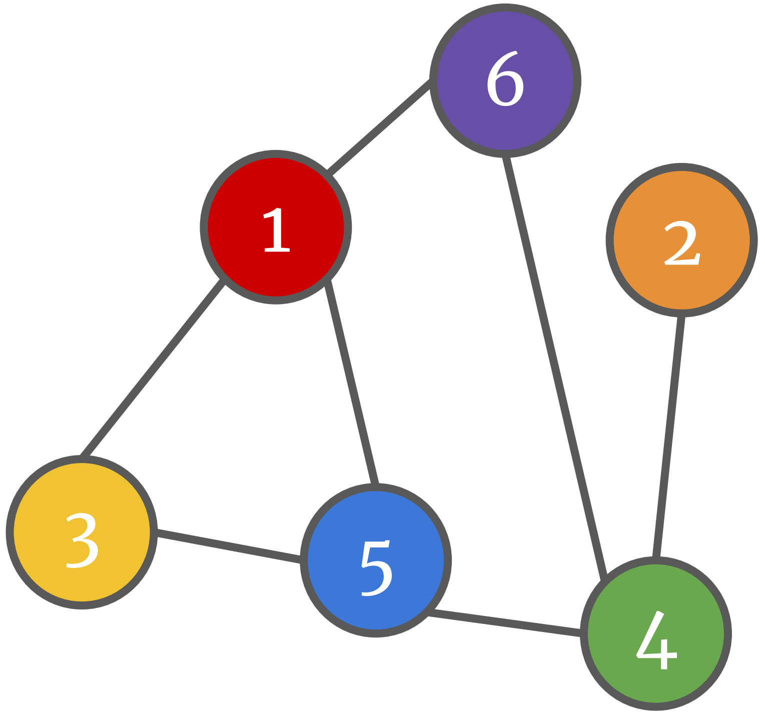
Add 5: [5, 5]... *wait a minute*

Avoiding Duplicates In The Queue!

- One idea: before putting something in the queue, check to see if it's already there...
    - But it would take time linear in the size of the queue to check!
    - OK, then let's make the queue a set.
        - But then we lose the order on the vertices!

- Better idea: think of the Boolean as marking a vertex as INSERTED instead of as VISITED.

Breadth-First Traversal



- Create a queue of vertices to visit.
  - Initially, it just has the starting vertex.
  - Mark the starting vertex as INSERTED.

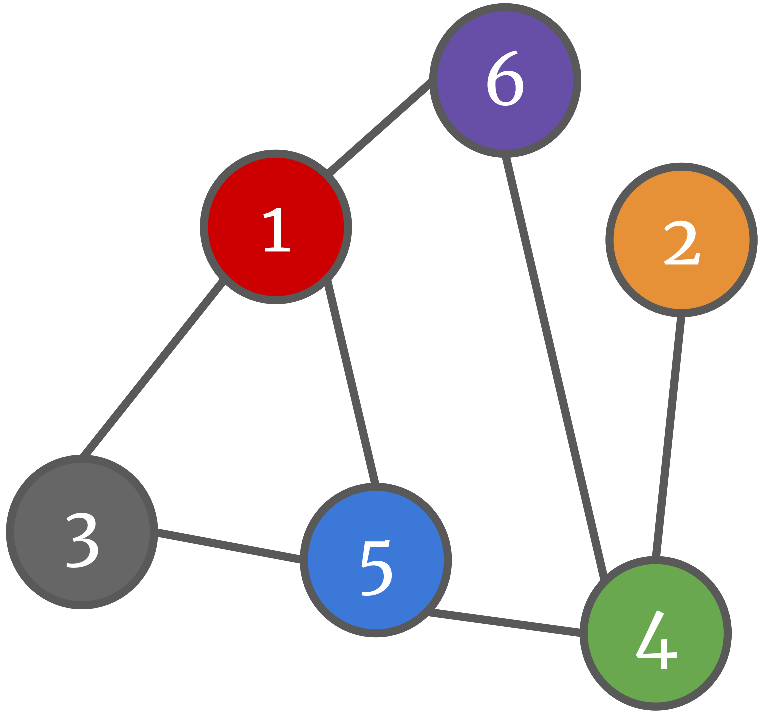
- Repeat the following:
  - Pop off the first value in the queue.
  - Follow each of its edges (in some order), putting the vertices you find in the queue (and marking them as INSERTED) as long as they are not INSERTED.

Breadth-First Traversal

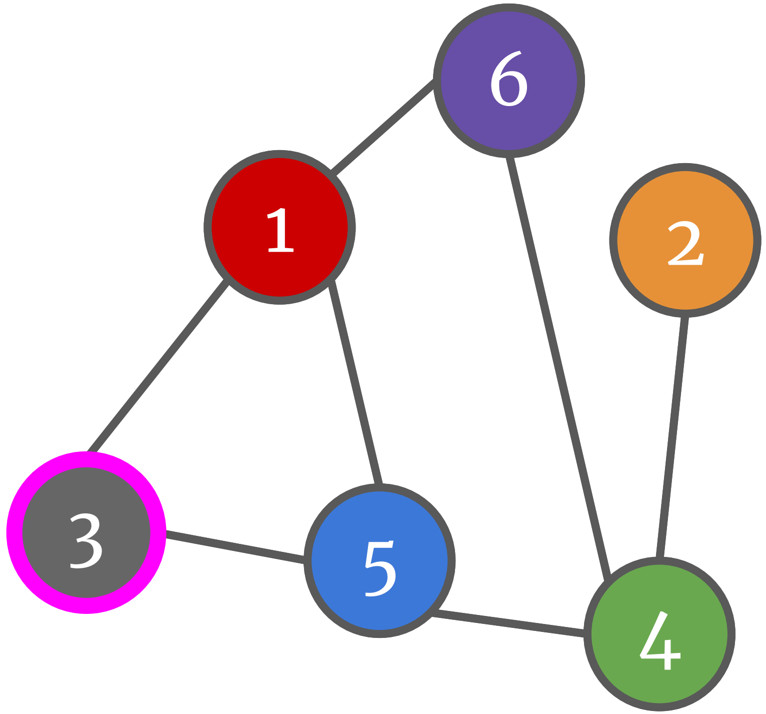
Queue: [3]

Inserted: [F, F, T, F, F, F]

Currently processing: None



Breadth-First Traversal

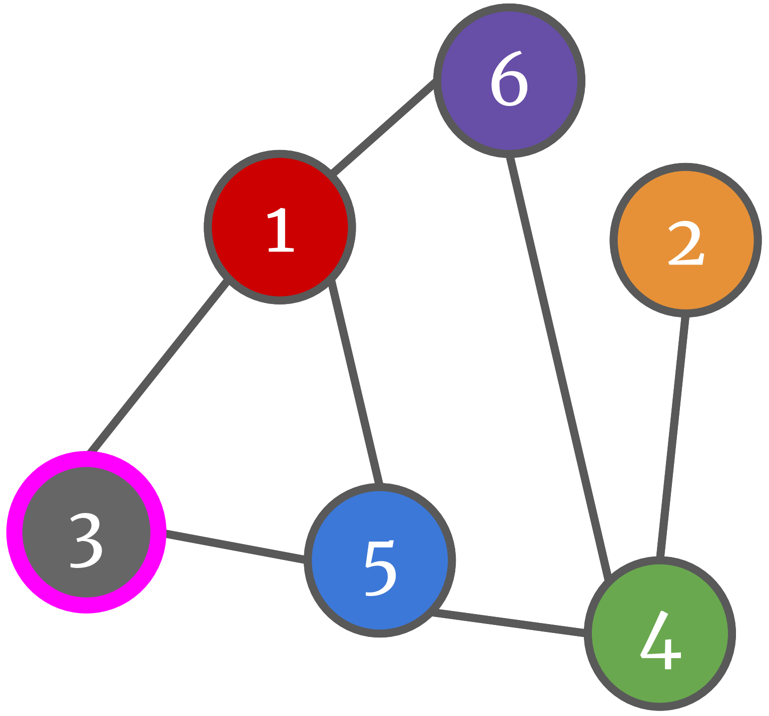


Queue: [ ]

Inserted: [F, F, T, F, F, F]

Currently processing: 3

Breadth-First Traversal



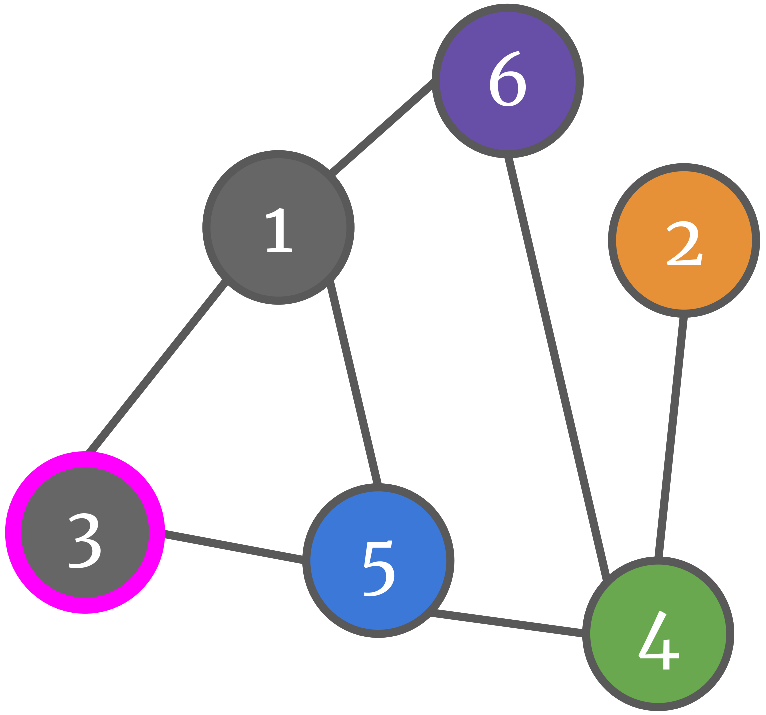
Queue: [ ]

Inserted: [F, F, T, F, F, F]

Currently processing: 3

- Neighbors: 1, 5

Breadth-First Traversal



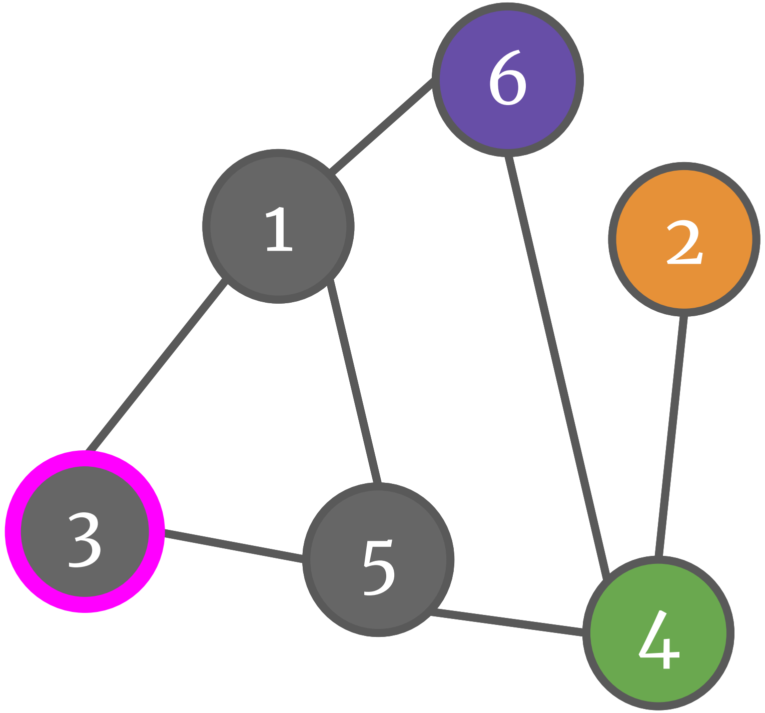
Queue: [1]

Inserted: [T, F, T, F, F, F]

Currently processing: 3

- Neighbors: 1, 5
- 1 is not inserted, so set it to inserted and add it to the queue

Breadth-First Traversal



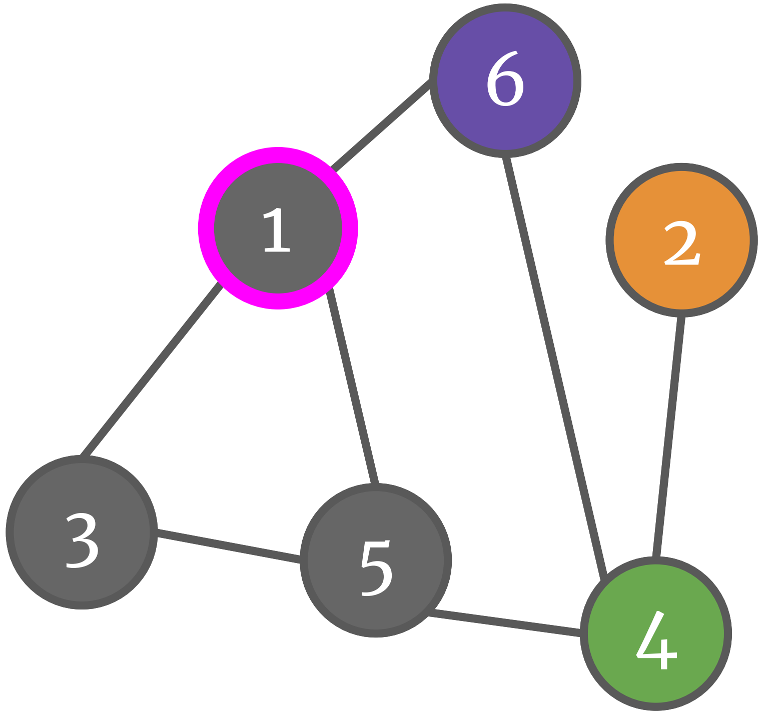
Queue: [1, 5]

Inserted: [T, F, T, F, T, F]

Currently processing: 3

- Neighbors: 1, 5
- 1 is not inserted, so set it to inserted and add it to the queue
- 5 is not inserted, so set it to inserted and add it to the queue
- Done processing 3!

Breadth-First Traversal



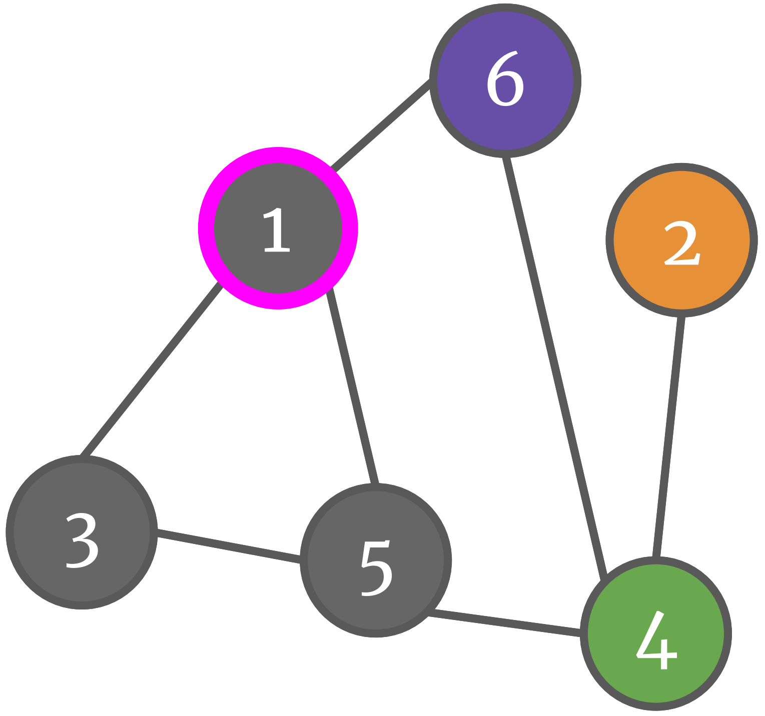
Queue: [5]

Inserted: [T, F, T, F, T, F]

Currently processing: 1

- Neighbors: 3, 5, 6

Breadth-First Traversal



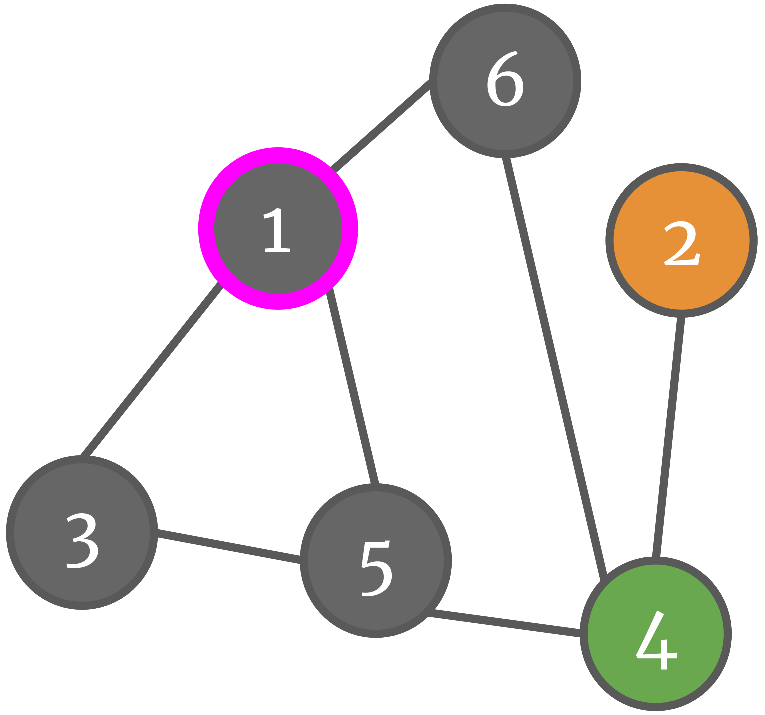
Queue: [5]

Inserted: [T, F, T, F, T, F]

Currently processing: 1

- Neighbors: 3, 5, 6
- 3 is inserted, so ignore it.
- 5 is inserted, so ignore it.

Breadth-First Traversal



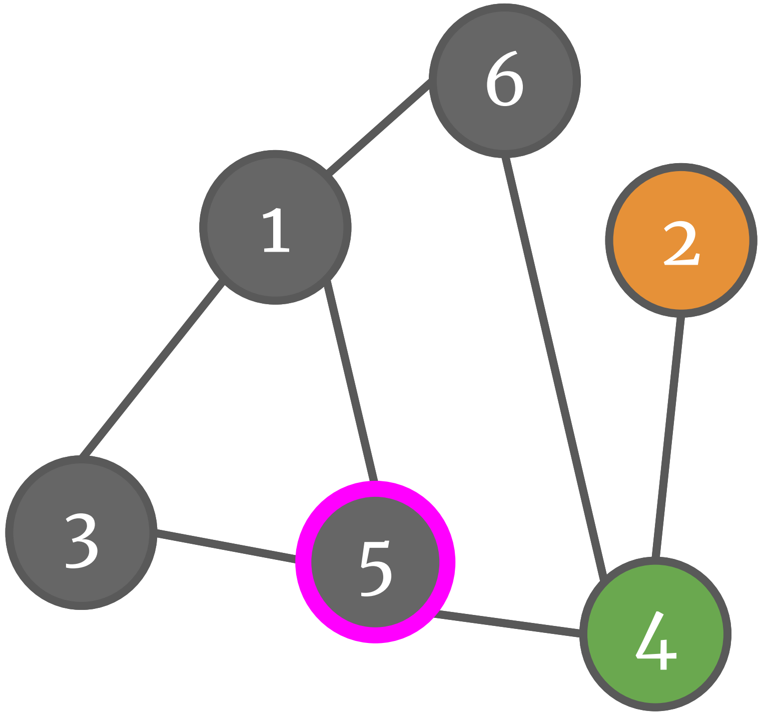
Queue: [5, 6]

Inserted: [T, F, T, F, T, T]

Currently processing: 1

- Neighbors: 3, 5, 6
- 3 is inserted, so ignore it.
- 5 is inserted, so ignore it.
- 6 is not inserted, so set it to inserted and add it to the queue
- Done processing 1!

Breadth-First Traversal

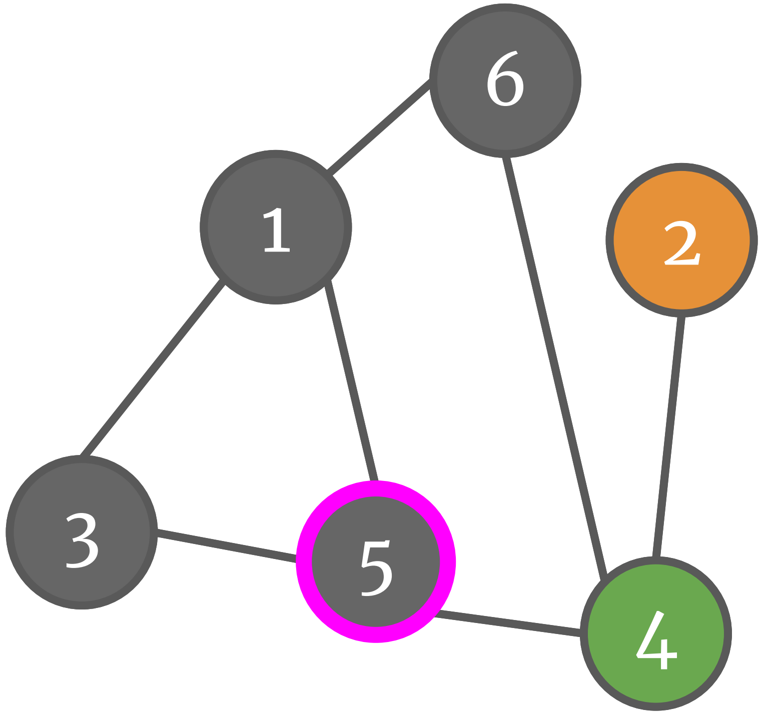


Queue: [6]

Inserted: [T, F, T, F, T, T]

Currently processing: 5

Breadth-First Traversal



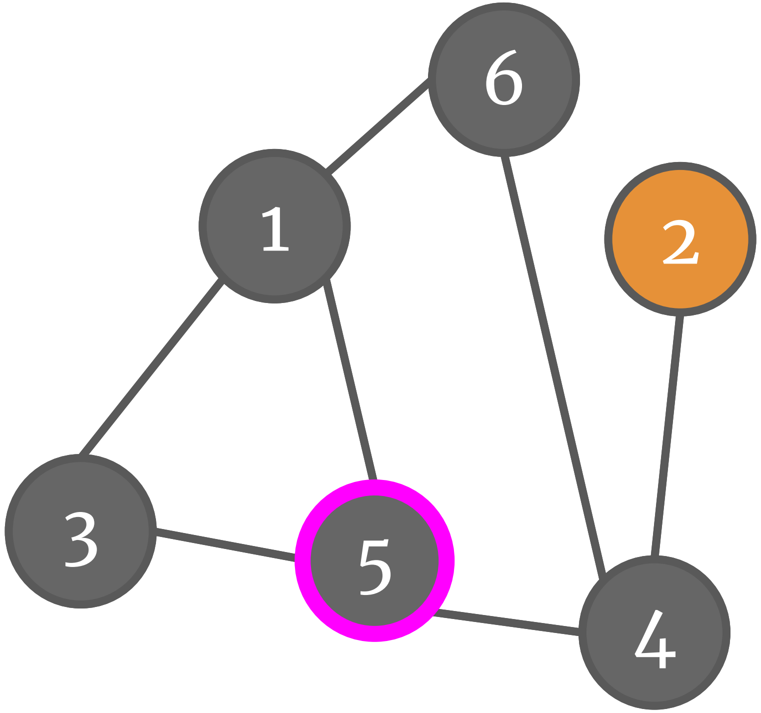
Queue: [6]

Inserted: [T, F, T, F, T, T]

Currently processing: 5

- Neighbors: 1, 3, 4
- 1 is inserted, so ignore it.
- 3 is inserted, so ignore it.

Breadth-First Traversal



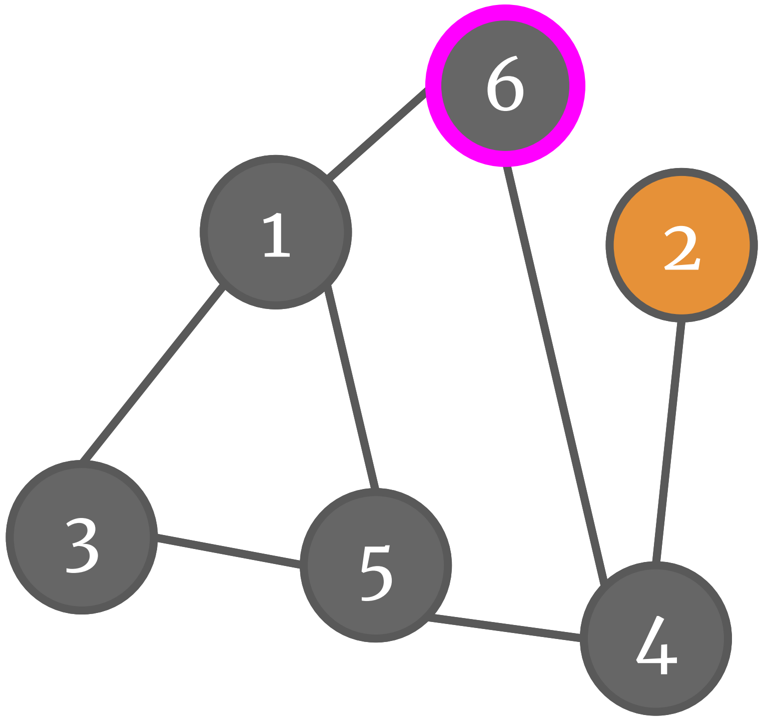
Queue: [6, 4]

Inserted: [T, F, T, T, T, T]

Currently processing: 5

- Neighbors: 1, 3, 4
- 1 is inserted, so ignore it.
- 3 is inserted, so ignore it.
- 4 is not inserted, so set it to inserted and add it to the queue
- Done processing 5!

Breadth-First Traversal

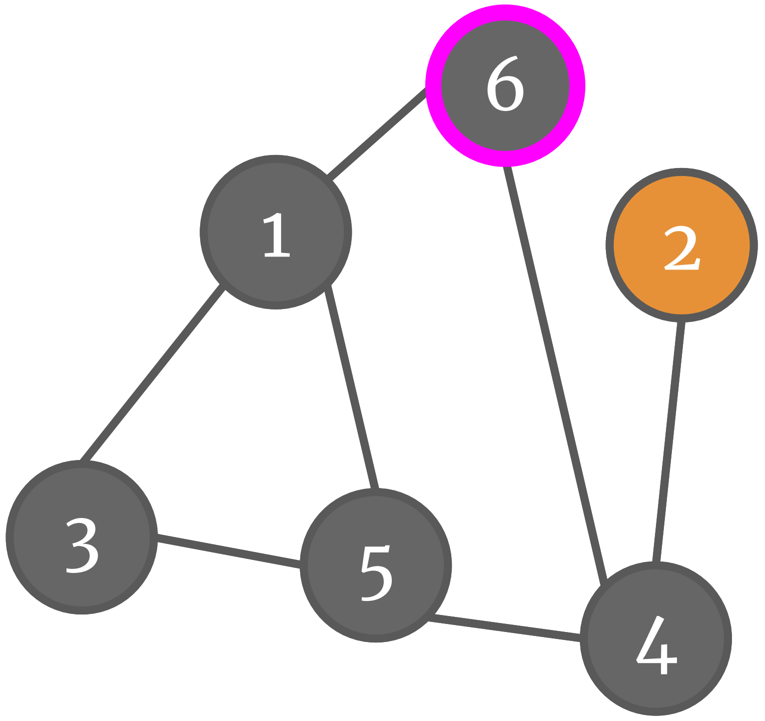


Queue: [4]

Inserted: [T, F, T, T, T, T]

Currently processing: 6

Breadth-First Traversal



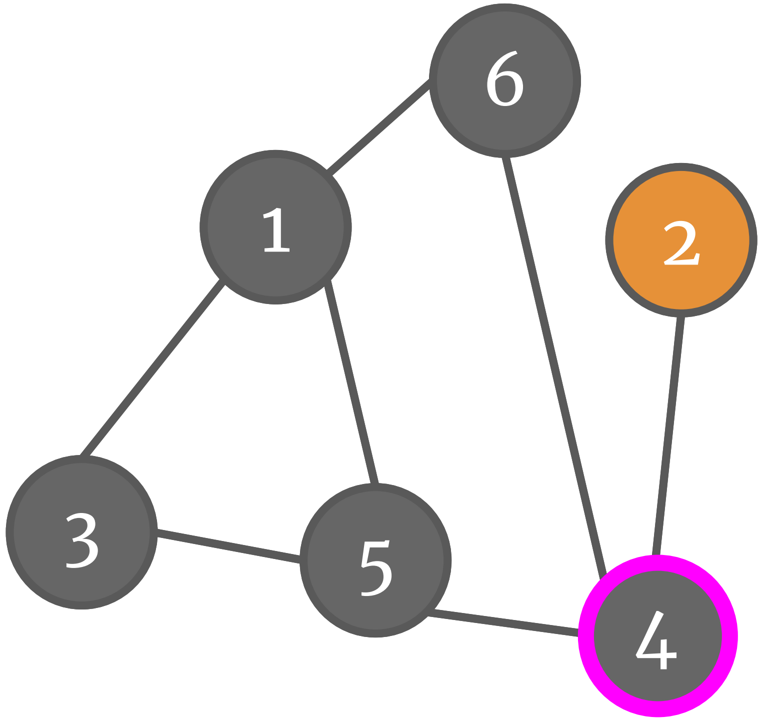
Queue: [4]

Inserted: [T, F, T, T, T, T]

Currently processing: 6

- Neighbors: 1, 4
- 1 is inserted, so ignore it.
- 4 is inserted, so ignore it.
- Done processing 6!

Breadth-First Traversal



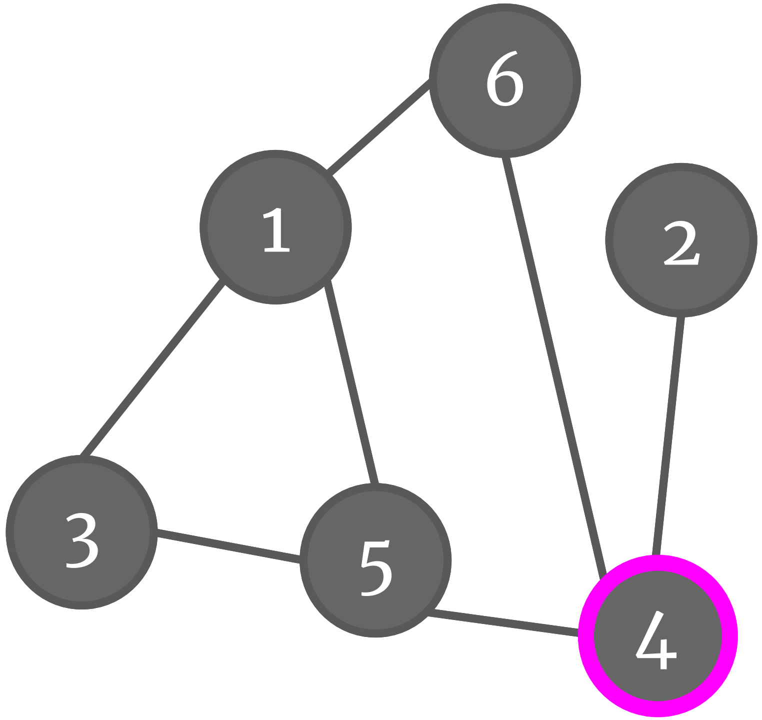
Queue: []

Inserted: [T, F, T, T, T, T]

Currently processing: 4

- Neighbors: 2, 5, 6

Breadth-First Traversal



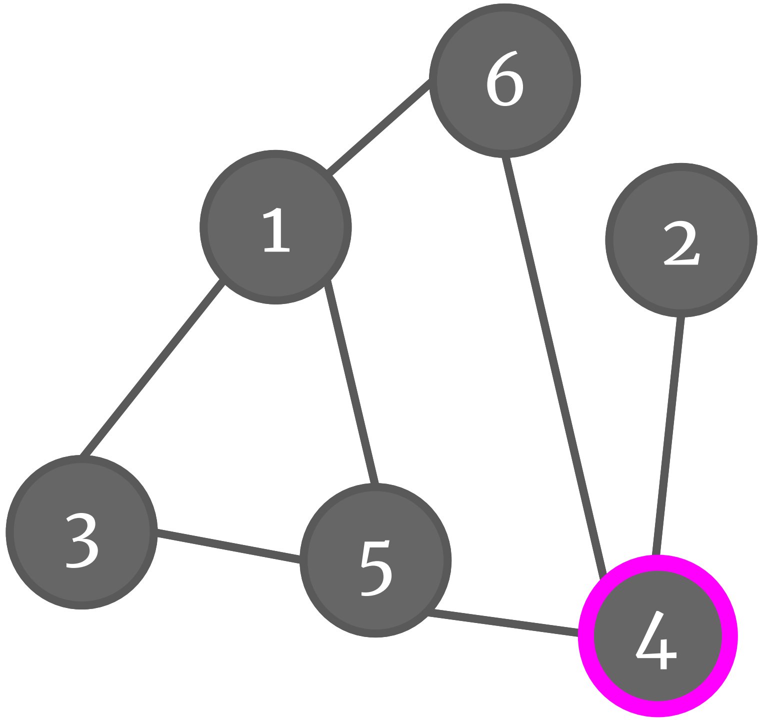
Queue: [2]

Inserted: [T, T, T, T, T, T]

Currently processing: 4

- Neighbors: 2, 5, 6
- 2 is not inserted, so set it to inserted and add it to the queue

Breadth-First Traversal



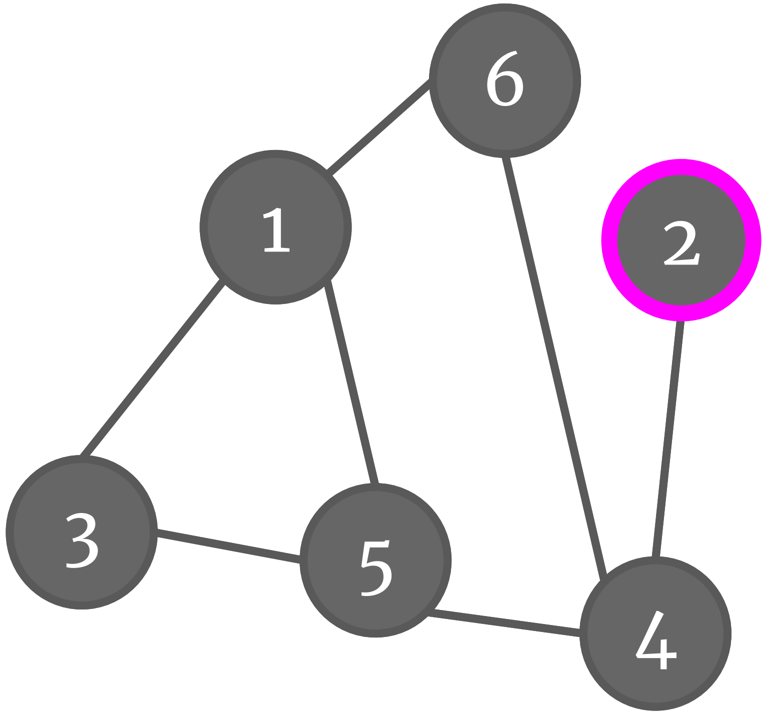
Queue: [2]

Inserted: [T, T, T, T, T, T]

Currently processing: 4

- Neighbors: 2, 5, 6
- 2 is not inserted, so set it to inserted and add it to the queue
- 5 is inserted, so ignore it.
- 6 is inserted, so ignore it.
- Done processing 4!

Breadth-First Traversal



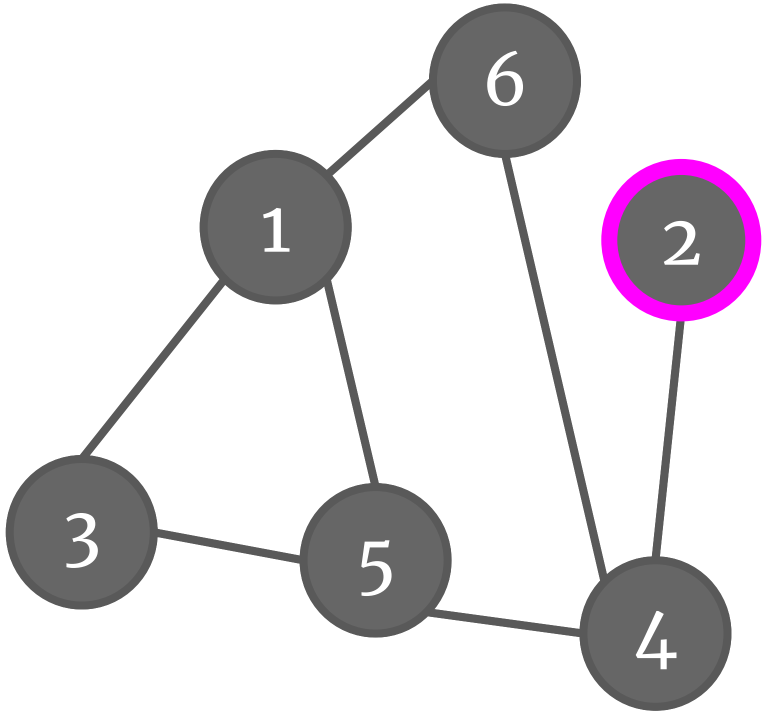
Queue: []

Inserted: [T, T, T, T, T, T]

Currently processing: 2

- Neighbors: 4
- 4 is inserted, so ignore it.
- Done processing 2!

Breadth-First Traversal



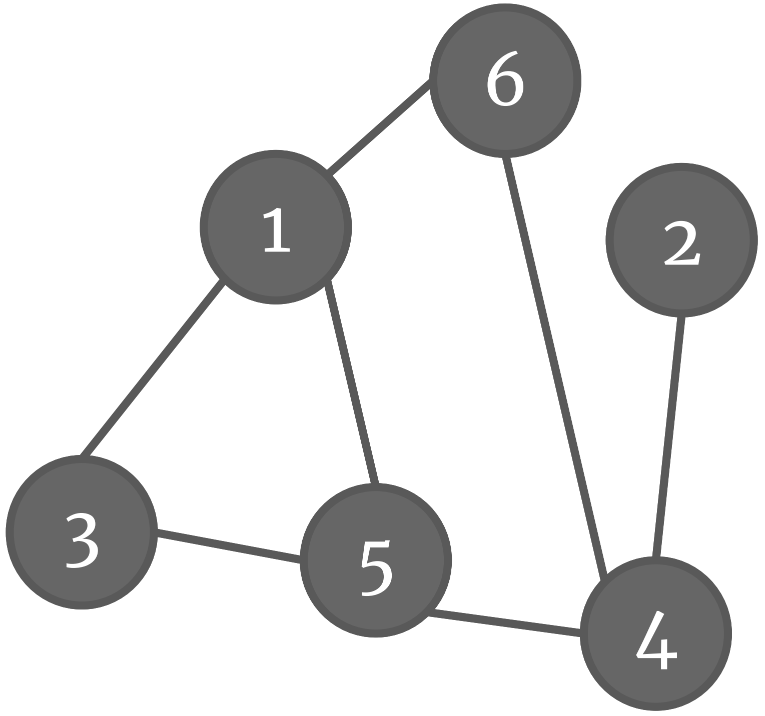
Queue: []

Inserted: [T, T, T, T, T, T]

Currently processing: 2
- Neighbors: 4
- 4 is inserted, so ignore it.
- Done processing 2!

Queue is empty, so we stop.

Breadth-First Traversal



Queue: []

Inserted: [T, T, T, T, T, T]

Currently processing: 2

- Neighbors: 4
- 4 is inserted, so ignore it.
- Done processing 2!

We reached every vertex!

Running Time

- We have to visit each of the n vertices at least once to look at its neighbors in the adjacency list.

- We consider each of the m edges at most twice (once each in its two vertices' adjacency lists)

- Running time: $O(n + m)$

Running Time

- We have to visit each of the n vertices at least once to look at its neighbors in the adjacency list.
- We consider each of the m edges at most twice (once each in its two vertices' adjacency lists)
- Running time: $O(n + m)$

But m can itself be $\Theta(n^2)$. Why not say $O(n^2)$?

Running Time

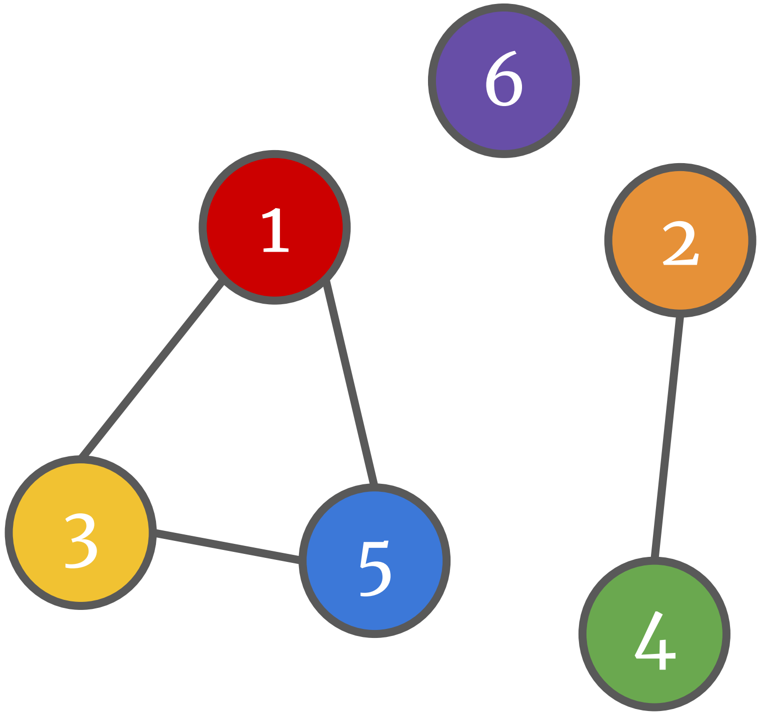
- We have to visit each of the *n* vertices at least once to look at its neighbors in the adjacency list.
- We consider each of the *m* edges at most twice (once each in its two vertices' adjacency lists)
- Running time: $O(n + m)$

But m can itself be $\Theta(n^2)$. Why not say $O(n^2)$?
True, but this is more informative / flexible the way it is. What if the graph is a tree with O(n) edges?

Counting connected components

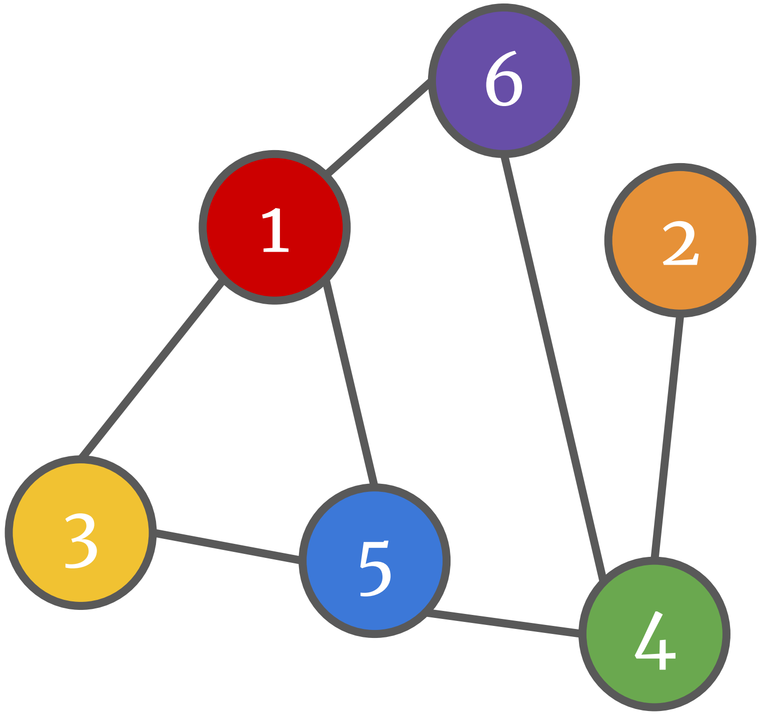
- Proceed through the vertices in order 1, 2, ..., doing the following:
    - If a vertex has not been SEEN:
        - Perform a breadth-first traversal on it and mark all encountered nodes as SEEN.
        - Increment the count of connected components.
    - Otherwise, do nothing.

Counting connected components



1: Mark 1, 3, 5 as SEEN. +1
2. Mark 2, 4 as SEEN. +1
3. Skip (3 is SEEN)
4. Skip (4 is SEEN)
5. Skip (5 is SEEN)
6. Mark 6 as SEEN. +1

How many steps apart are two vertices?



Oh, right, we wanted to solve this! For each other vertex, what's the fastest way to get there from 1?

How should we approach this?

Shortest Distance

- Do the same BFS traversal, but label each vertex with a distance, starting from 0 at the start vertex.

- Distance = 1 + the distance of the vertex that led us here.

- What if a vertex already has a distance at the time we try to set the distance? Can our new value be **better**?

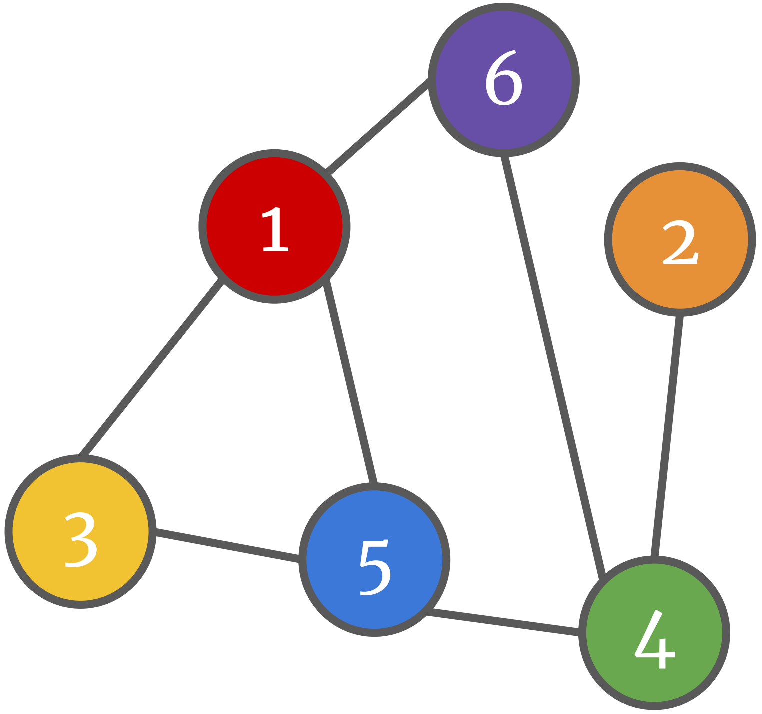
Shortest Distance

- Do the same BFS traversal, but label each vertex with a distance, starting from 0 at the start vertex.

- Distance = 1 + the distance of the vertex that led us here.

- What if a vertex already has a distance at the time we try to set the distance? Can our new value be **better**? No – let's see why.

Generations of the queue



Start: 3

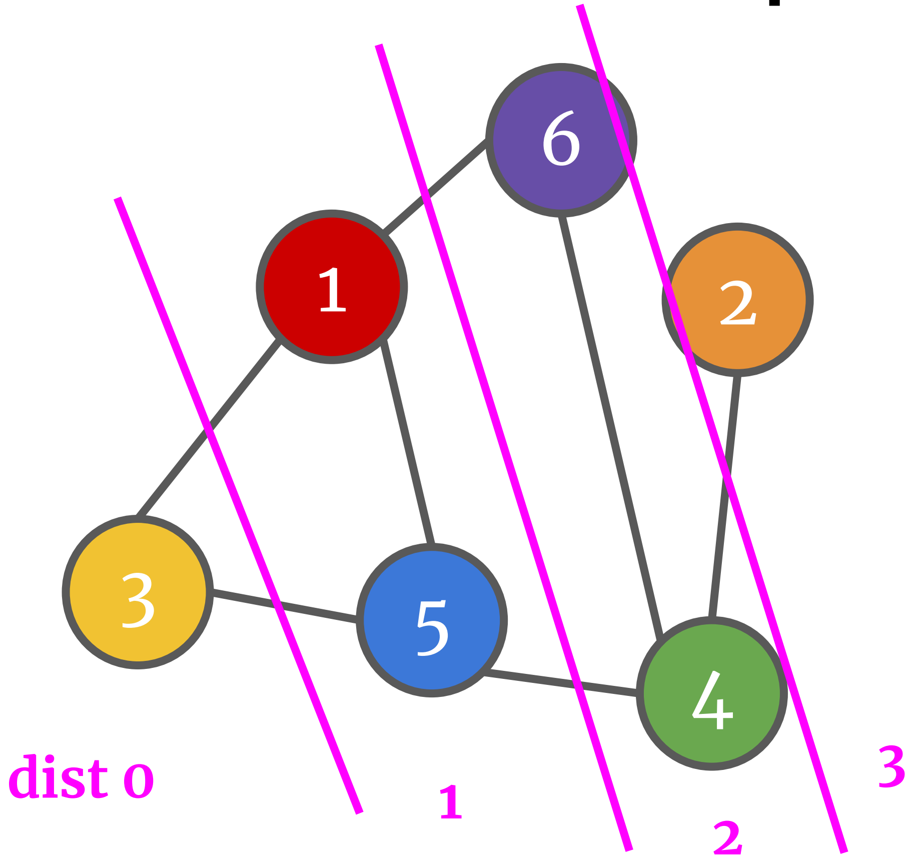
After fully processing 3, we had: 1, 5

After fully processing 1, 5, we had: 6, 4

After fully processing 6, 4, we had: 2

After fully processing 2: we were done

Generations of the queue



Start: 3

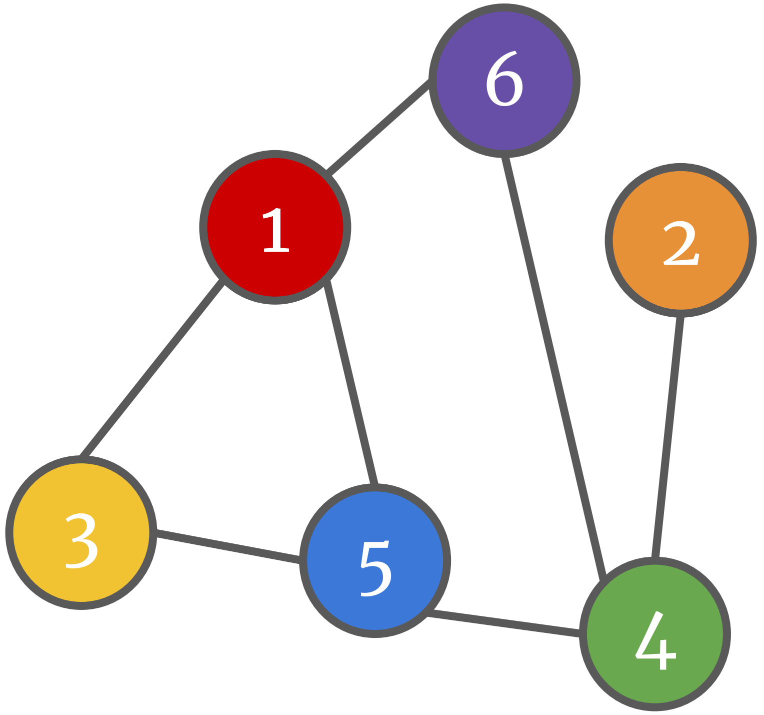
After fully processing 3, we had: 1, 5

After fully processing 1, 5, we had: 6, 4

After fully processing 6, 4, we had: 2

After fully processing 2: we were done

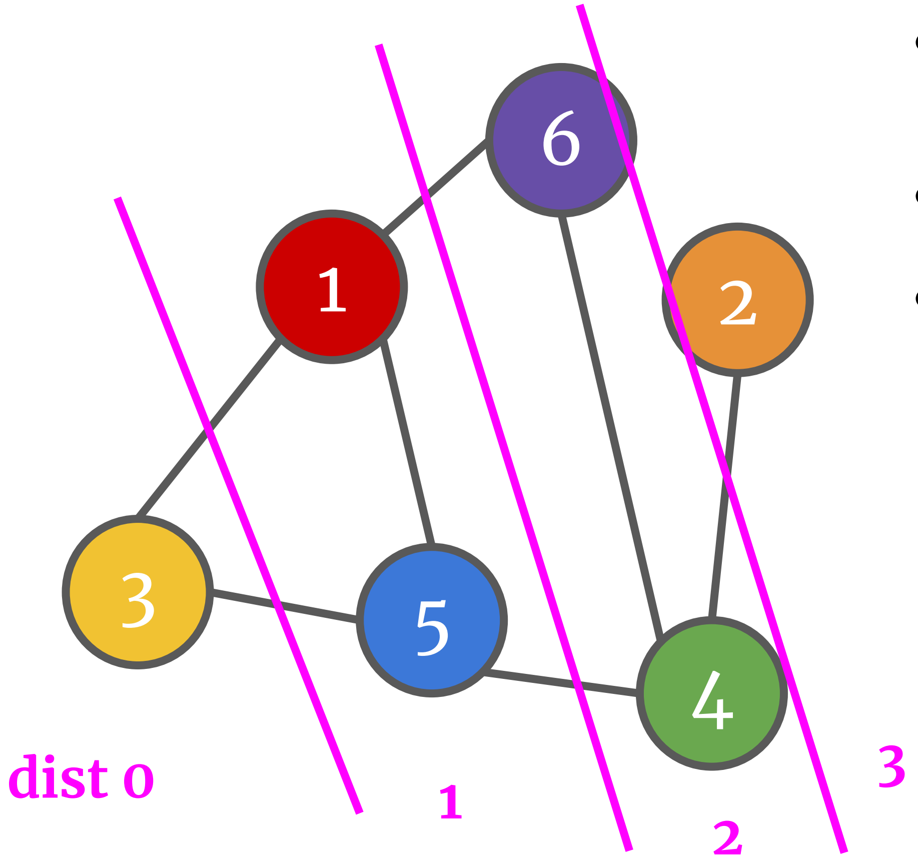
Shortest Distances



- Create a queue with just the starting vertex.
- Mark the starting vertex as INSERTED, and set its distance to 0.

- Set roundNum = 1.

Shortest Distances



- Create a queue with just the starting vertex.
- Mark the starting vertex as INSERTED, and set its distance to 0.

- Set roundNum = 1.

- Repeat the following:
    - Create a new empty queue.
    - For each value in the old queue, in order:
        - Process it as before, putting its non-INSERTED neighbors in the new queue.
        - Label its distance as roundNum.
    - Increment roundNum.
    - New queue = old queue.

Now you're equipped for

The Divergent Hunger Maze!

- You and your partner have been trapped in a maze with two floors. (Think of them as being superimposed, one atop the other.)

- You each start in the lower left corner, but on separate floors. You are each trying to get to the upper right corner of your floor…

- …but you always have to stay within *k* moves of each other so you can exchange awkward quips through the floor/ceiling.

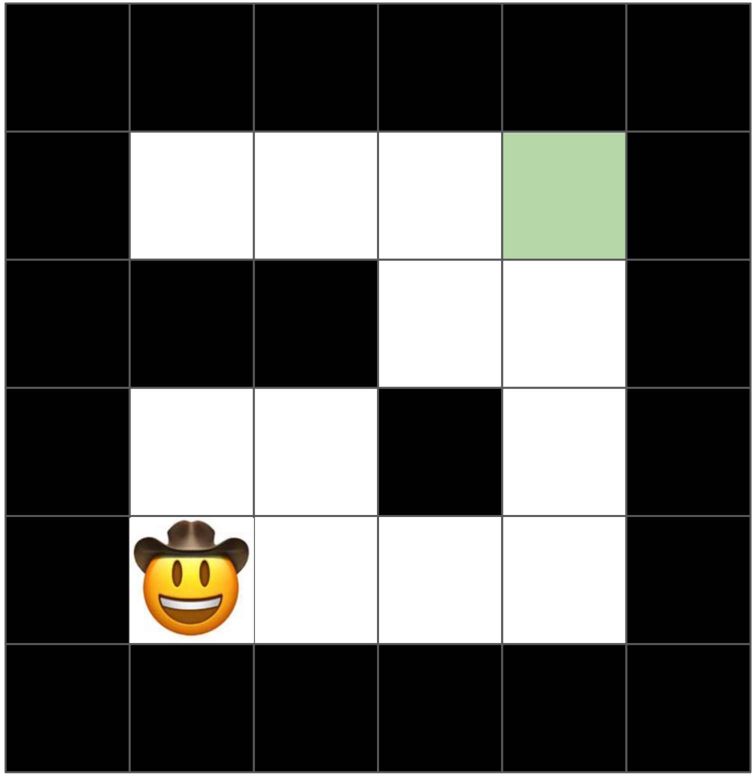
- You also want to use as few combined moves as possible.

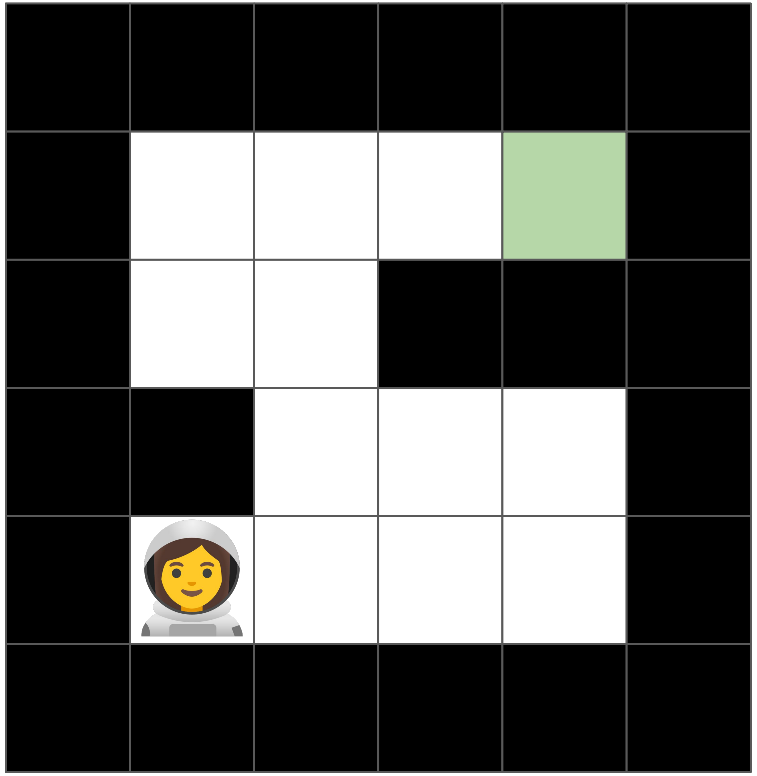
stay within 2 moves

Floor 1

0 moves so far

Floor 2



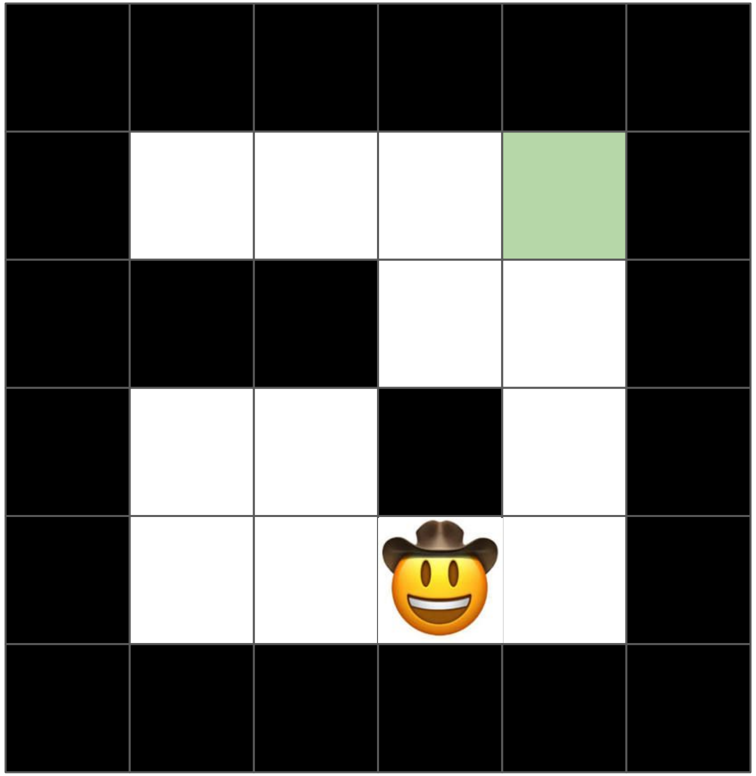


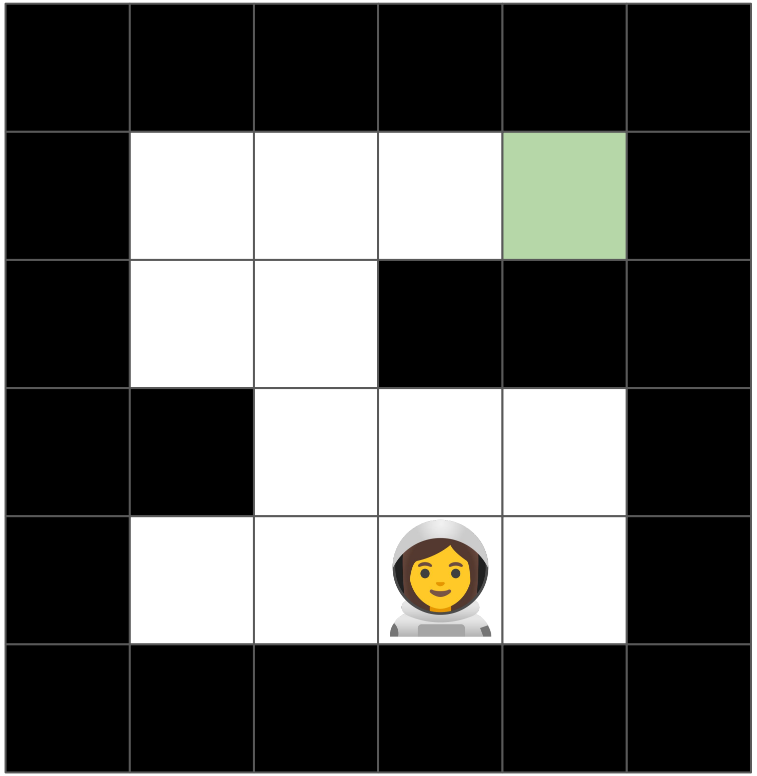
stay within 2 moves

Floor 1

4 moves so far

Floor 2

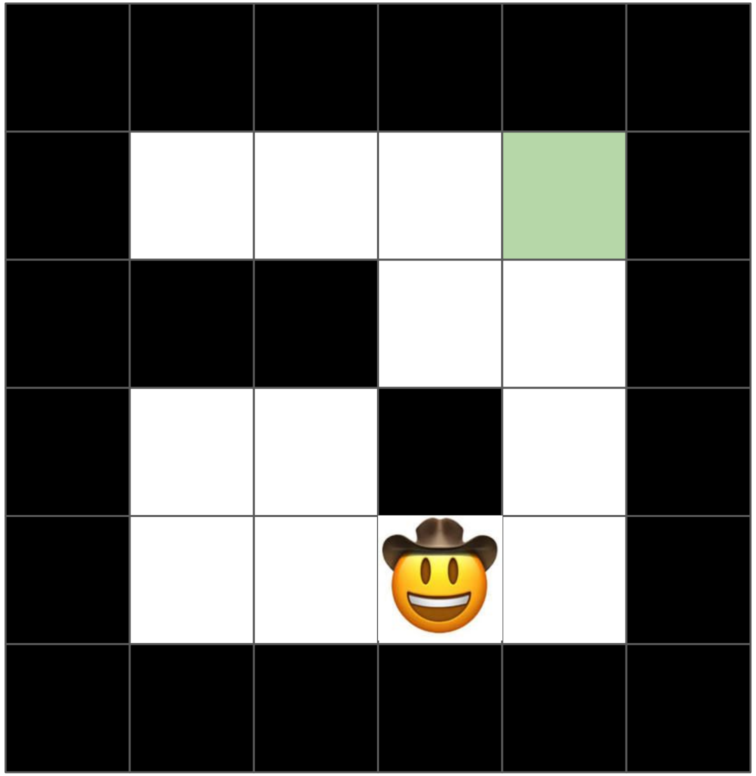




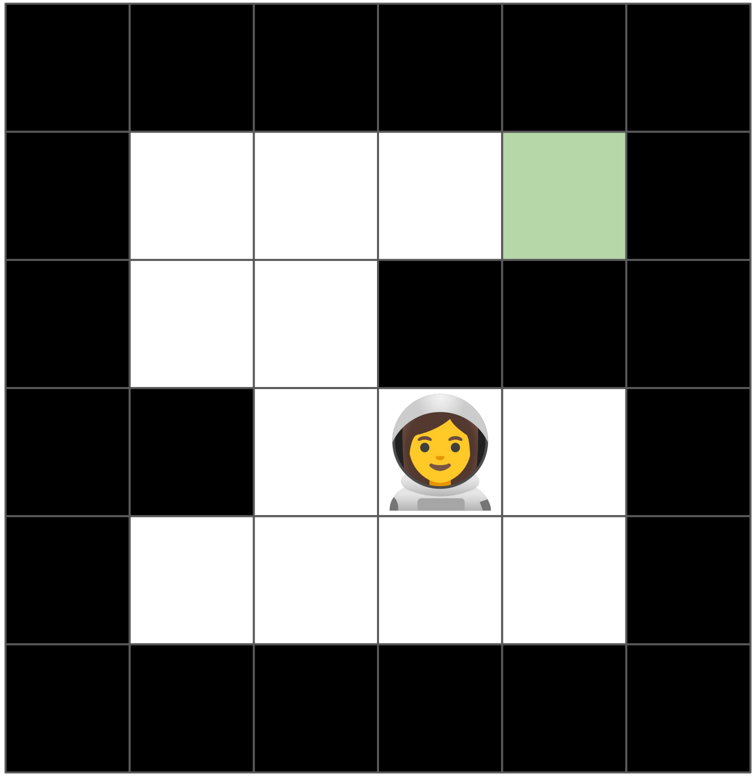
stay within 2 moves

5 moves so far

Floor 1



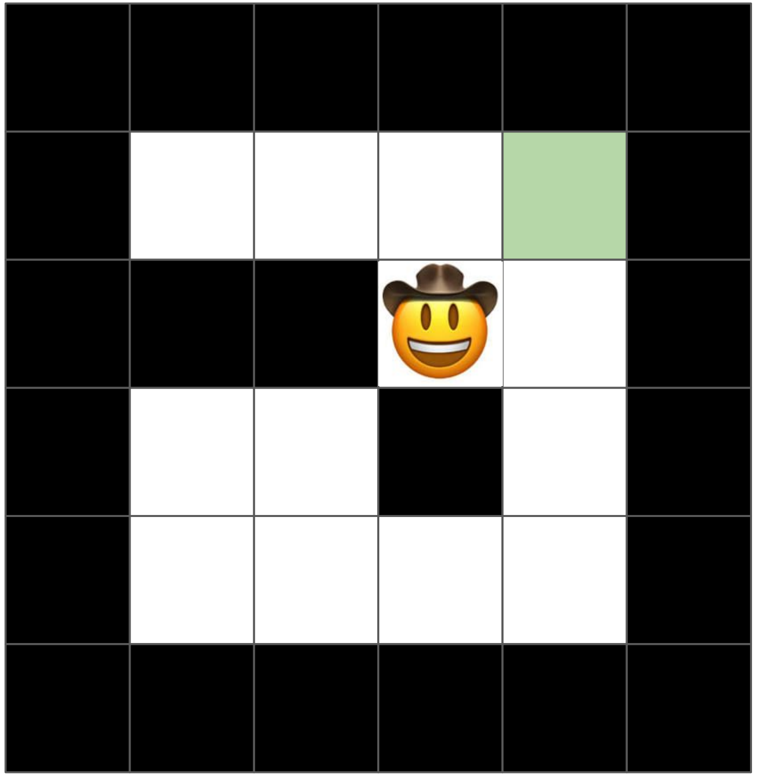
Floor 2



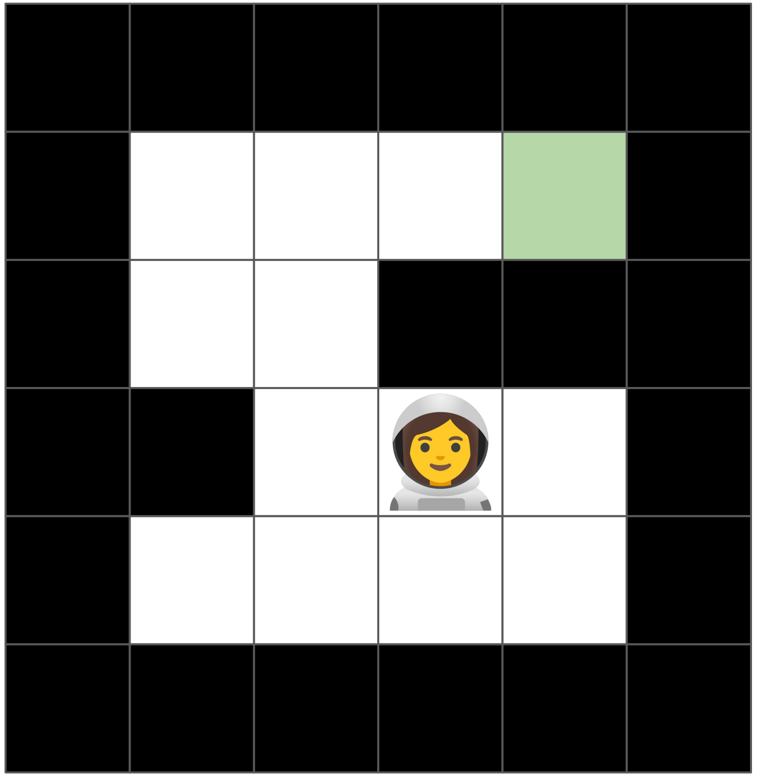
stay within 2 moves

9 moves so far

Floor 1



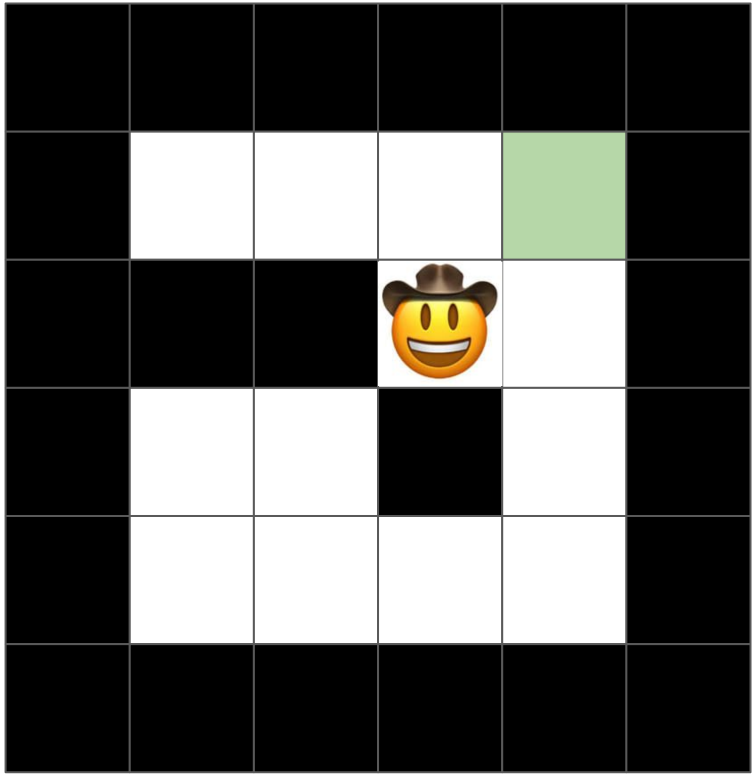
Floor 2



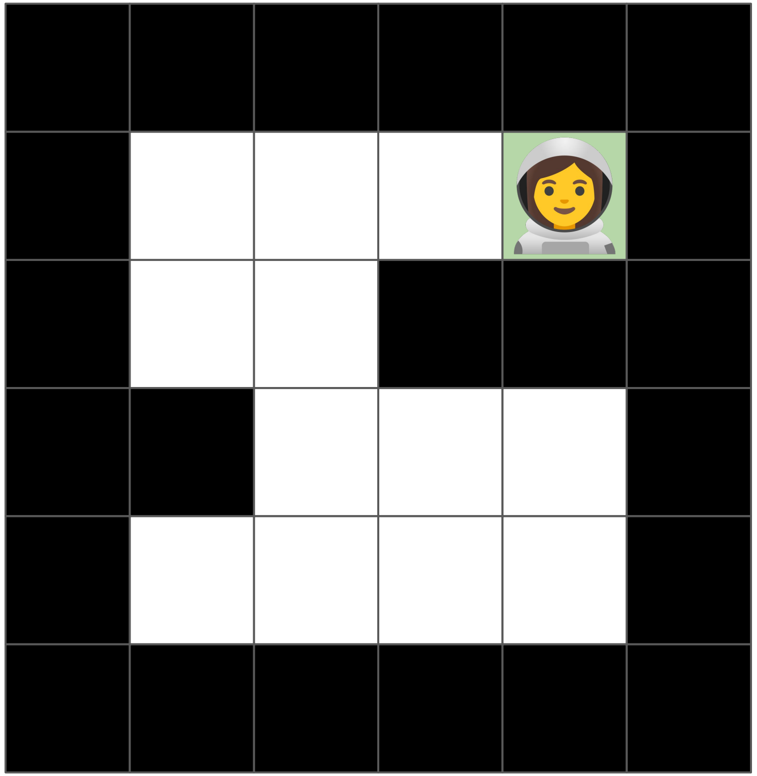
stay within 2 moves

14 moves so far

Floor 1



Floor 2

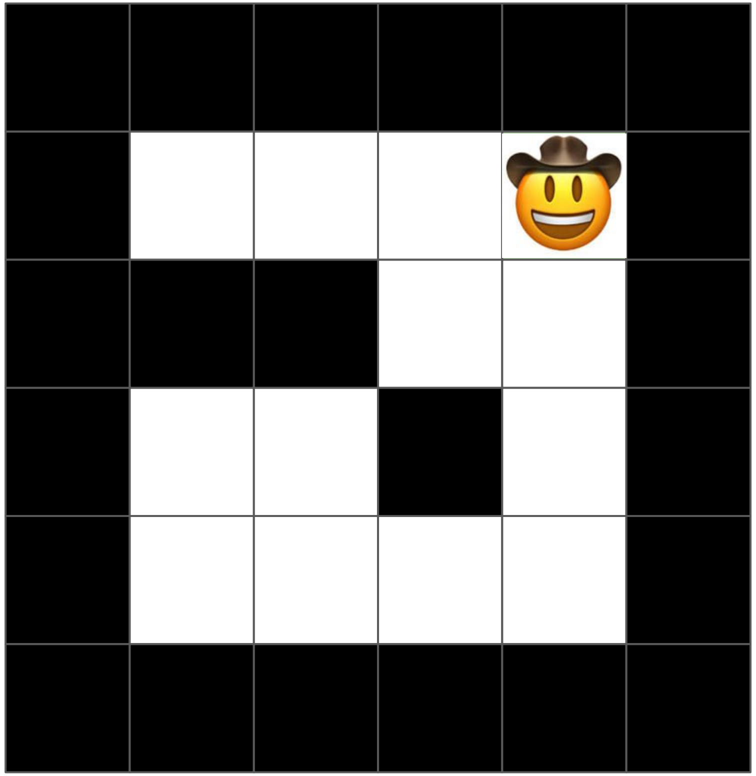


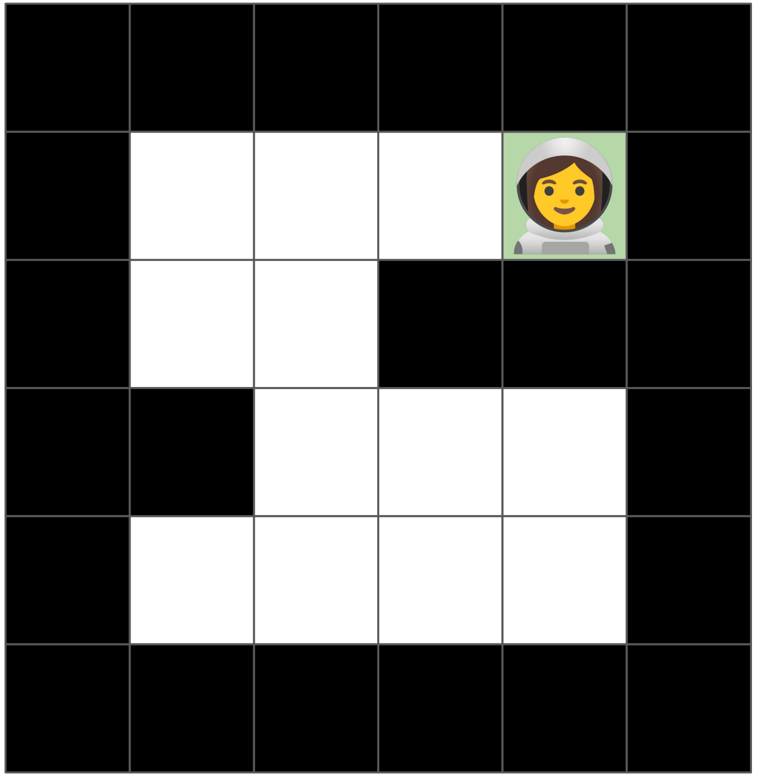
stay within 2 moves

Floor 1

16 moves so far

Floor 2





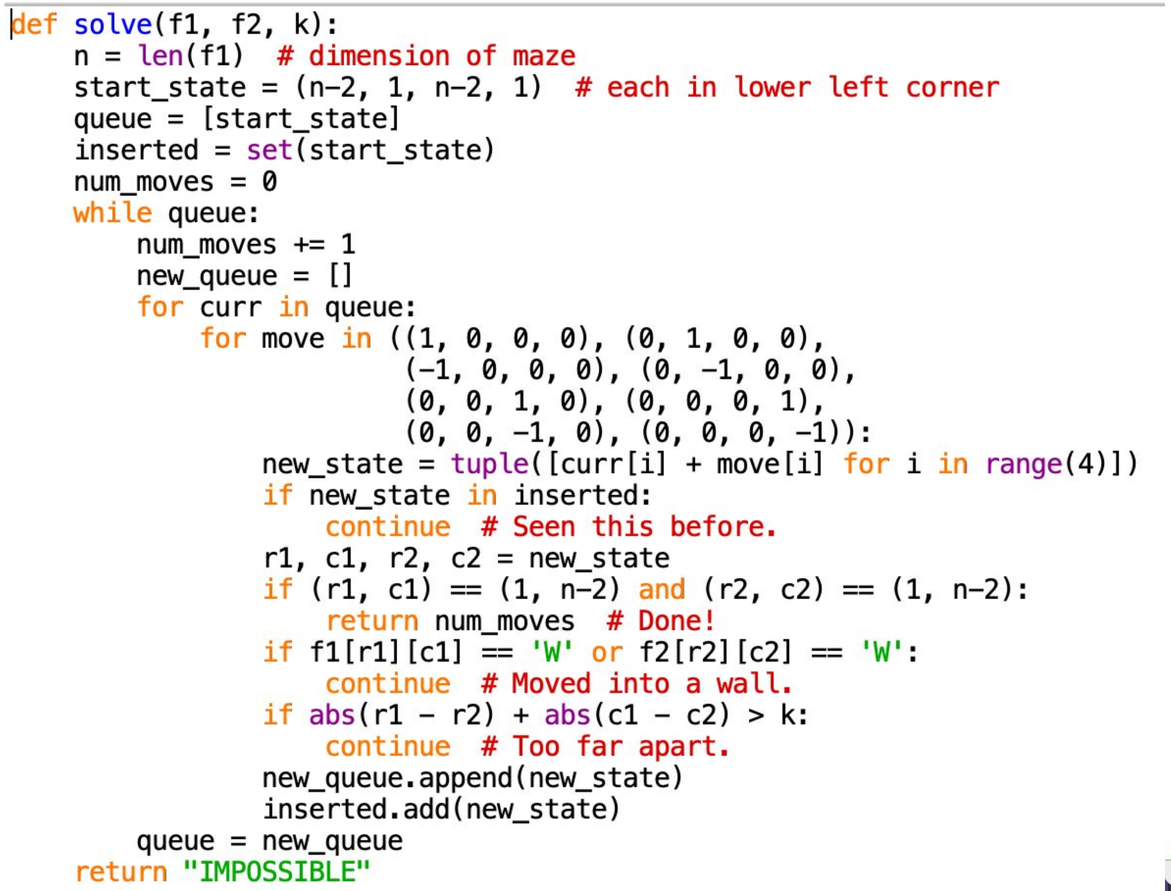
We know how to solve these mazes individually

- Just use BFS!

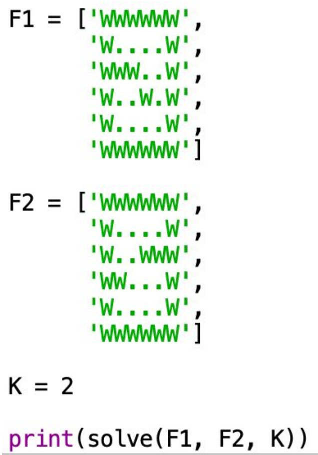
- But how do we solve them *at the same time*, given that they constrain each other?

An Algorithm

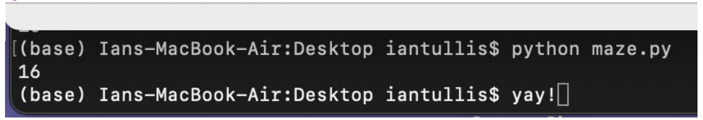
- Each state of this process can be described by the positions of the two people.
- Not all states are legal.
- To find the states reachable from a given state, try moving each of the people in all possible directions, and also checking that they don't get too far apart.
- Run BFS on this combined graph.



```
def solve(f1, f2, k):
    n = len(f1)  # dimension of maze
    start_state = (n-2, 1, n-2, 1)  # each in lower left corner
    queue = [start_state]
    inserted = set(start_state)
    num_moves = 0
    while queue:
        num_moves += 1
        new_queue = []
        for curr in queue:
            for move in ((1, 0, 0, 0), (0, 1, 0, 0),
                         (-1, 0, 0, 0), (0, -1, 0, 0),
                         (0, 0, 1, 0), (0, 0, 0, 1),
                         (0, 0, -1, 0), (0, 0, 0, -1)):
                new_state = tuple([curr[i] + move[i] for i in range(4)])
                if new_state in inserted:
                    continue  # Seen this before.
                r1, c1, r2, c2 = new_state
                if (r1, c1) == (1, n-2) and (r2, c2) == (1, n-2):
                    return num_moves  # Done!
                if f1[r1][c1] == 'W' or f2[r2][c2] == 'W':
                    continue  # Moved into a wall.
                if abs(r1 - r2) + abs(c1 - c2) > k:
                    continue  # Too far apart.
                new_queue.append(new_state)
                inserted.add(new_state)
        queue = new_queue
    return "IMPOSSIBLE"
```

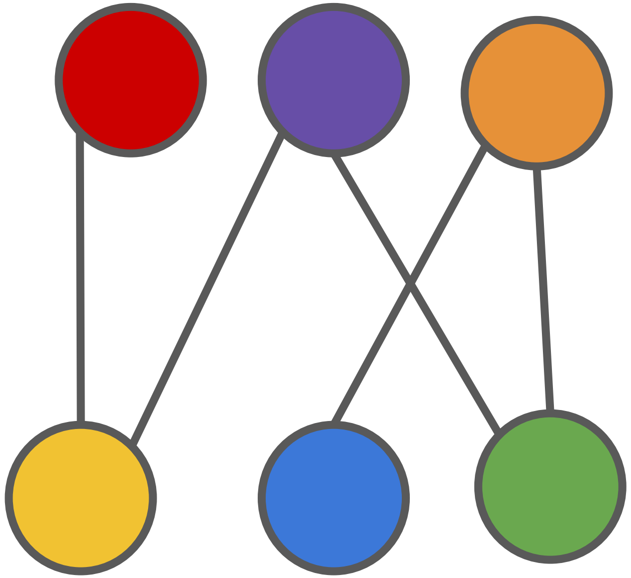


```
F1 = ['WWWWWW',
      'W....W',
      'WWW..W',
      'W..W.W',
      'W....W',
      'WWWWWW']

F2 = ['WWWWWW',
      'W....W',
      'W..WWW',
      'WW...W',
      'W....W',
      'WWWWWW']

K = 2

print(solve(F1, F2, K))
```



```
[(base) Ians-MacBook-Air:Desktop iantullis$ python maze.py
16
(base) Ians-MacBook-Air:Desktop iantullis$ yay!
```

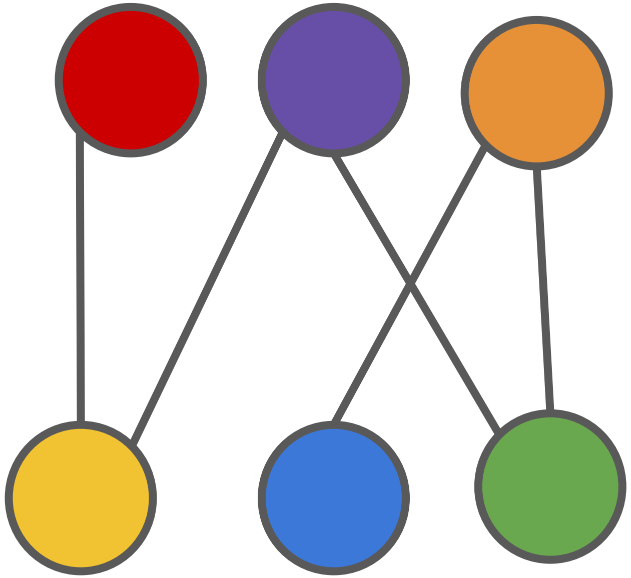
More graph terminology!



- A graph is **bipartite** if it can be divided into exactly two groups, such that every edge goes *between* the groups (i.e., no edges *within* groups).

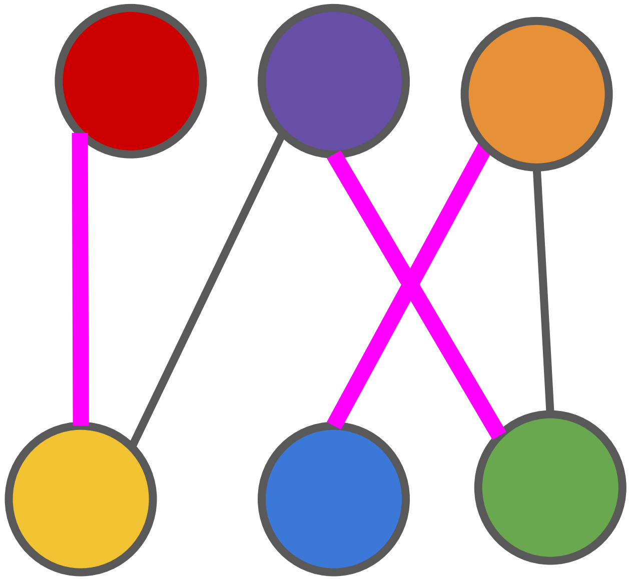
- This here graph is bipartite!

(Why do we care about bipartiteness?)



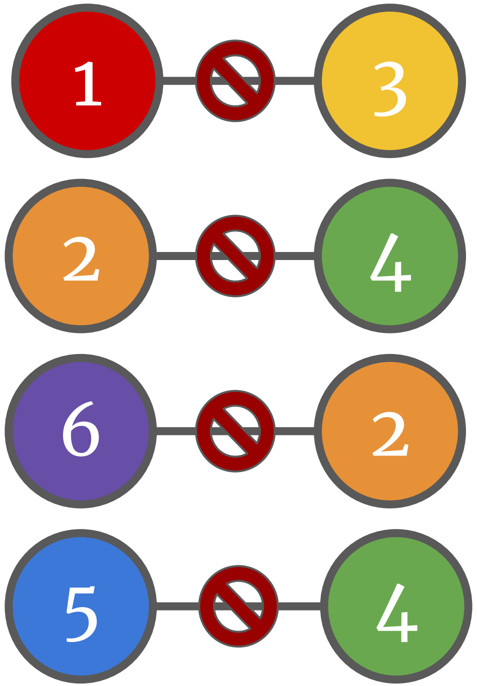
- This is the basis of **matching** problems. E.g., suppose that the vertices on the top are people, and the vertices on the bottom are jobs, and edges show who can do what. Can we give everyone exactly one job?
- We'll return to this in Unit 6 (Greed & Flow).

(Why do we care about bipartiteness?)



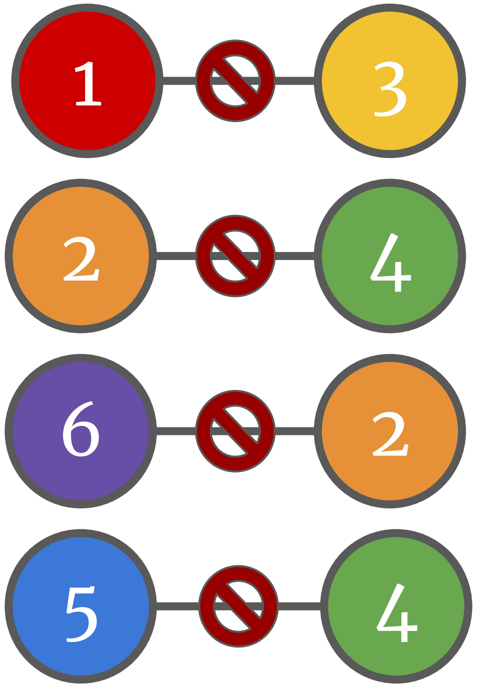
- This is the basis of **matching** problems. E.g., suppose that the vertices on the top are people, and the vertices on the bottom are jobs, and edges show who can do what. Can we give everyone exactly one job?
- We'll return to this in Unit 6 (Greed & Flow).

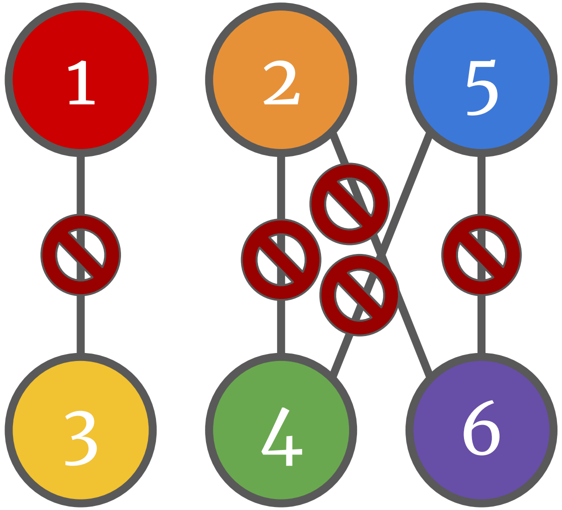
(Why do we care about bipartiteness?)



- This is also the basis of problems like: You know of a bunch of pairs of people who won't work together. Can you split them into two groups while honoring all of their requests?

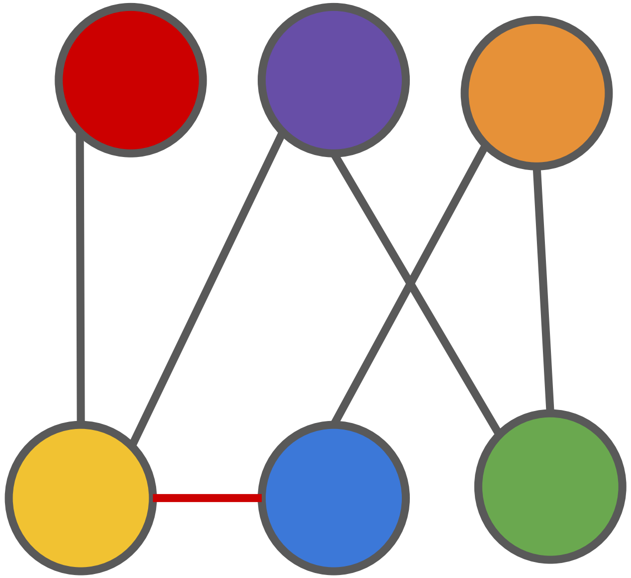
(Why do we care about bipartiteness?)





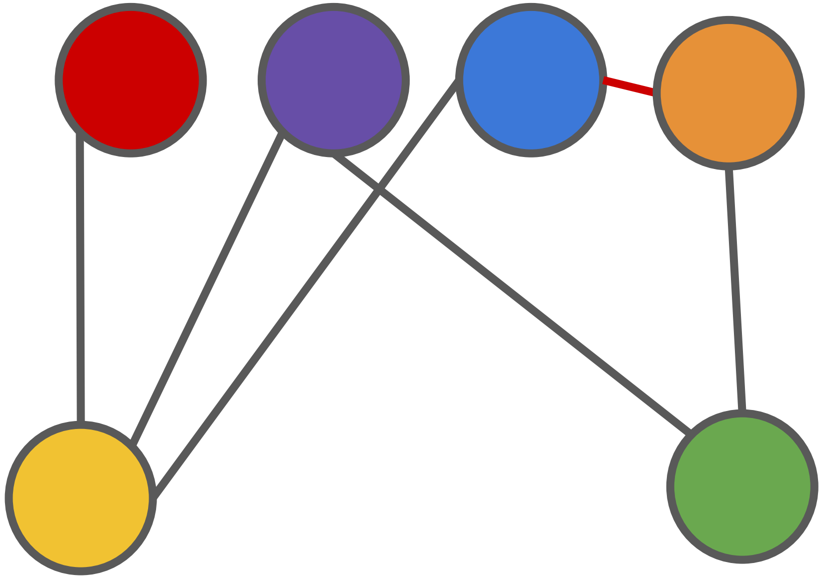
Success!

When is a graph not bipartite?



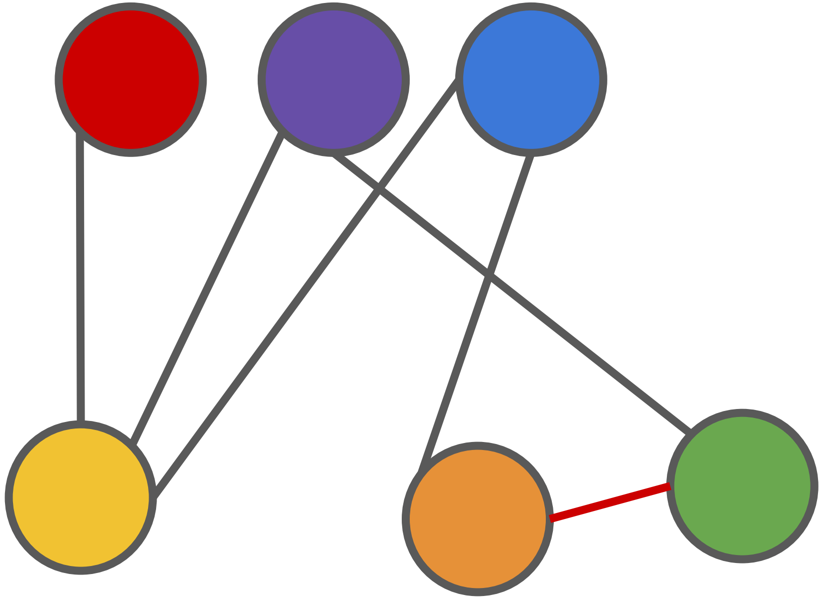
- This graph looks like it is not bipartite.
  - *How do we know? What if we just picked the wrong groups? Can we shift stuff around?*

When is a graph not bipartite?



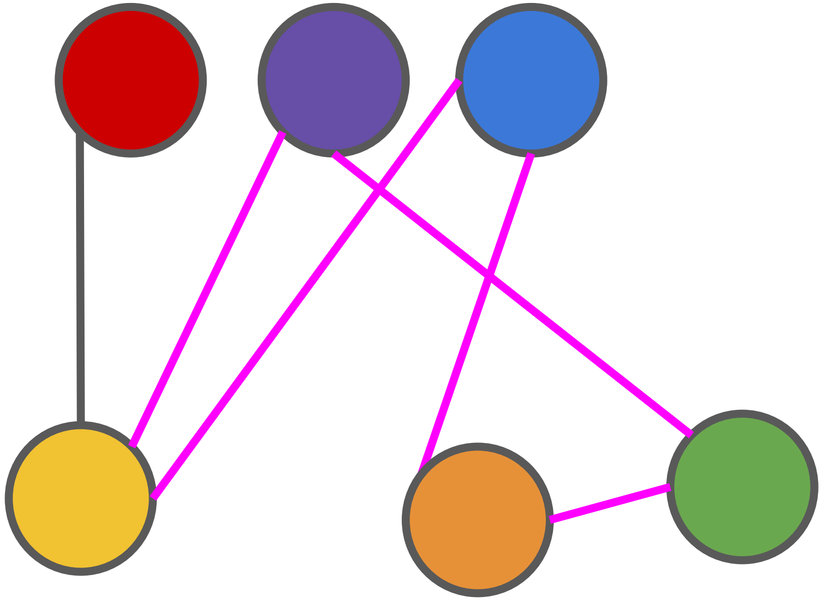
- This graph looks like it is not bipartite.
  - *How do we know? What if we just picked the wrong groups? Can we shift stuff around?*

When is a graph not bipartite?



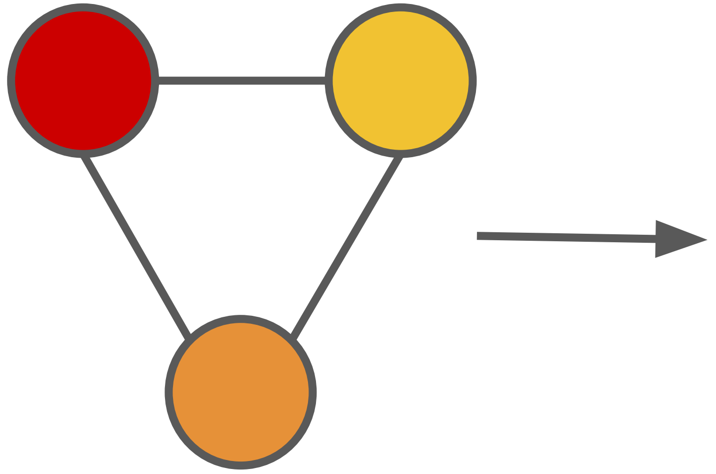
- This graph looks like it is not bipartite.
    - *How do we know? What if we just picked the wrong groups? Can we shift stuff around?*

When is a graph not bipartite?



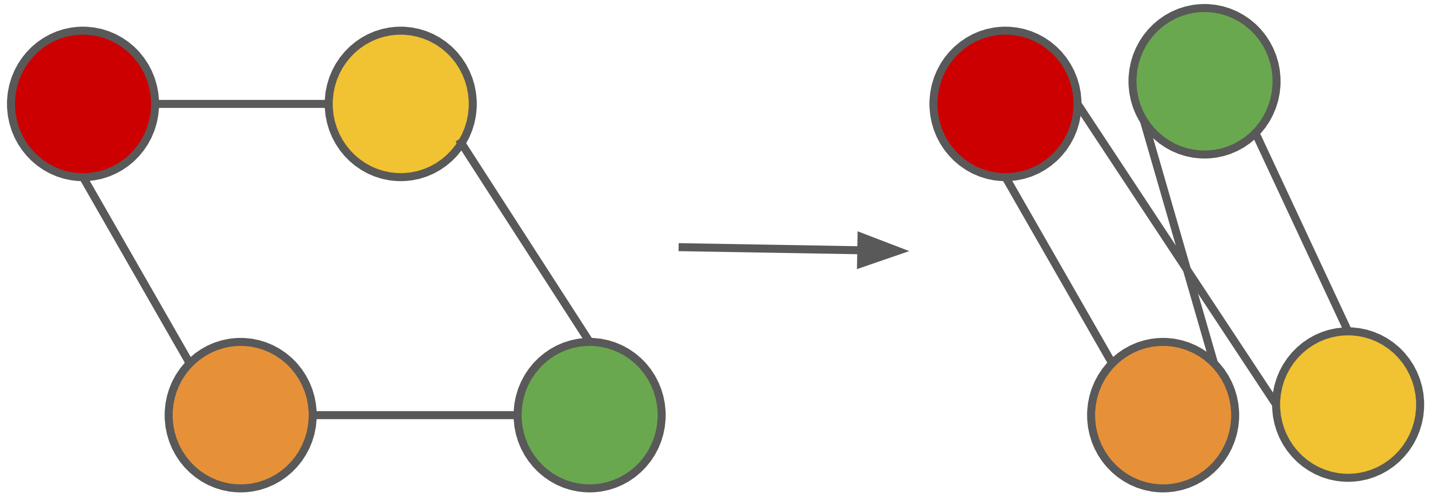
- This graph looks like it is not bipartite.
  - *How do we know? What if we just picked the wrong groups? Can we shift stuff around?* ***No!***
- Observe that it has a cycle of odd length. No amount of rearranging can change that!

A graph is bipartite if and only if it has no odd cycles.

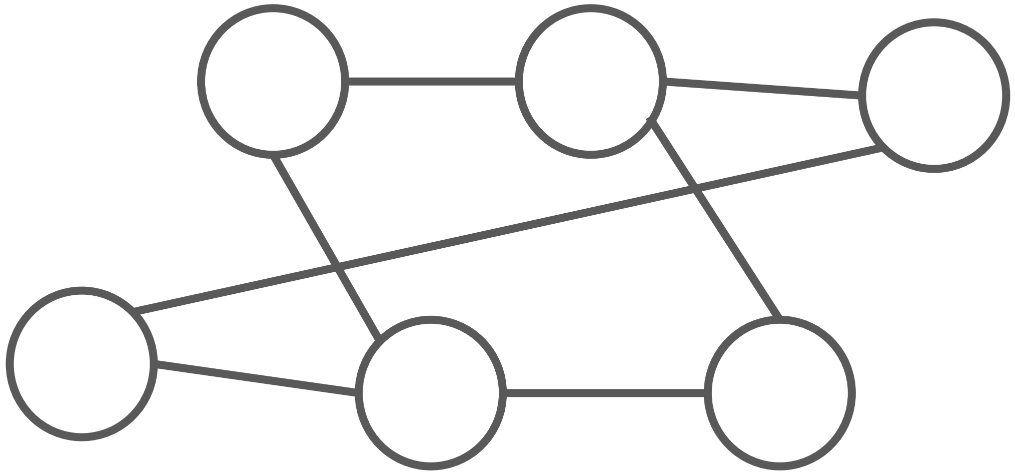


doomed!

A graph is bipartite if and only if it has no odd cycles.

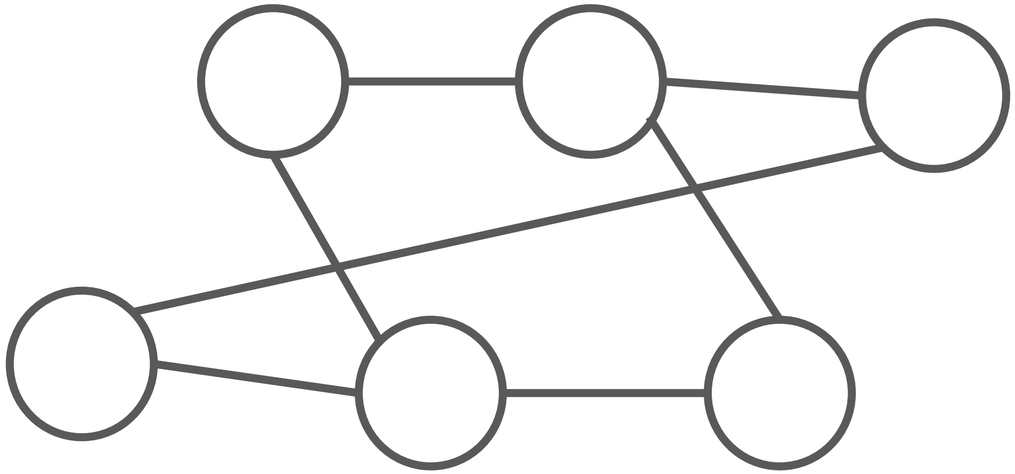


Is this graph bipartite?



Directly looking for odd cycles seems pretty inefficient!

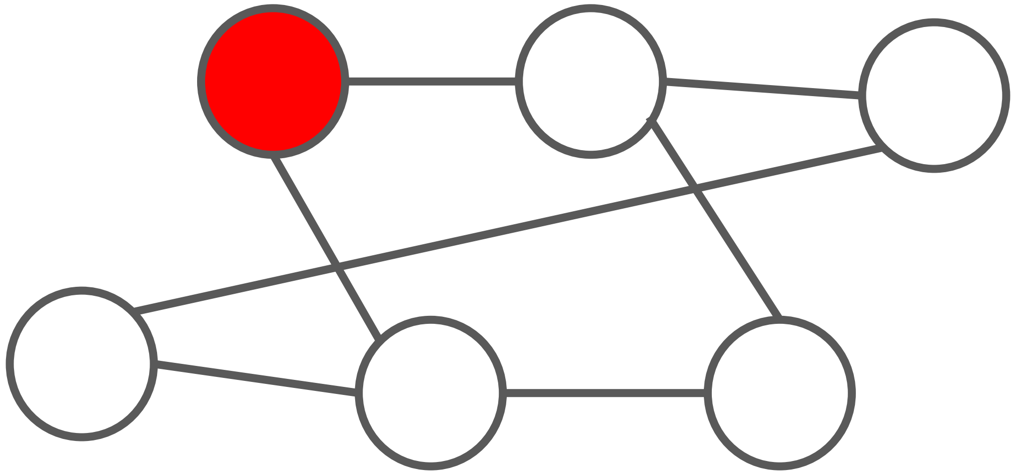
Is this graph bipartite?



Another way to check:
Try to color each vertex red or blue, such that no two edges share a color.

Is this graph bipartite?

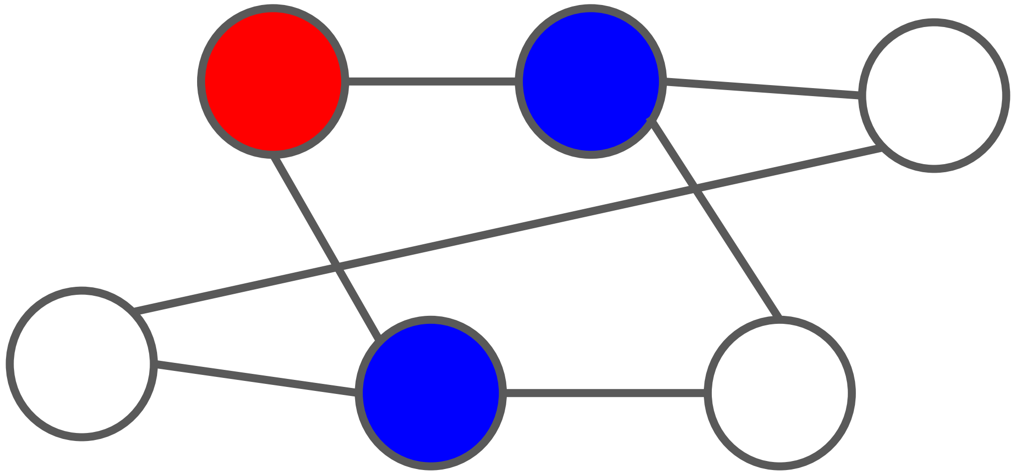
(arbitrary starting point and color)



Another way to check: Try to color each vertex red or blue, such that no two edges share a color.

Is this graph bipartite?

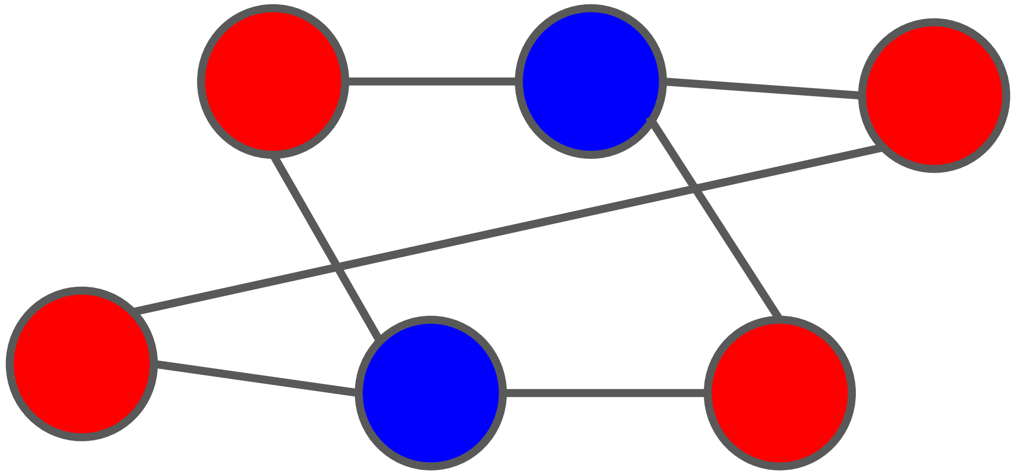
Its neighbors are forced to be blue.



Another way to check: Try to color each vertex red or blue, such that no two edges share a color.

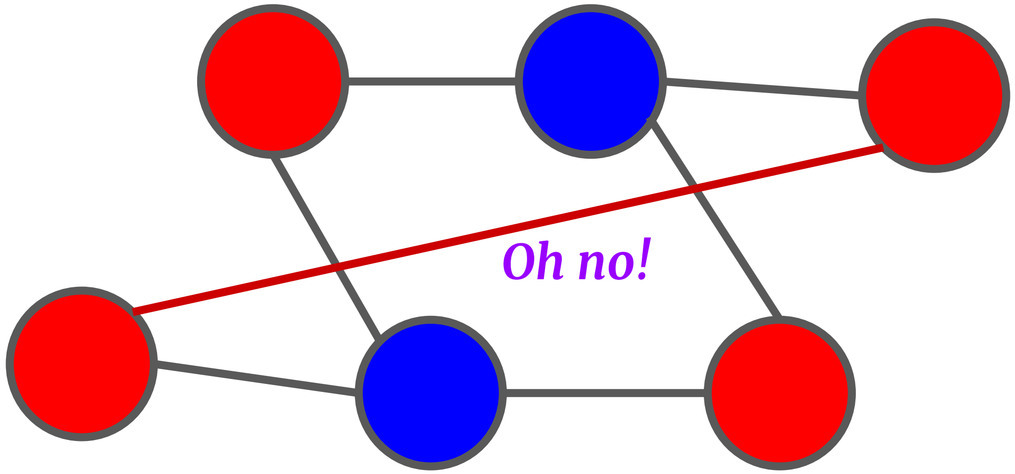
Is this graph bipartite?

And neighbors of those vertices are forced to be red...



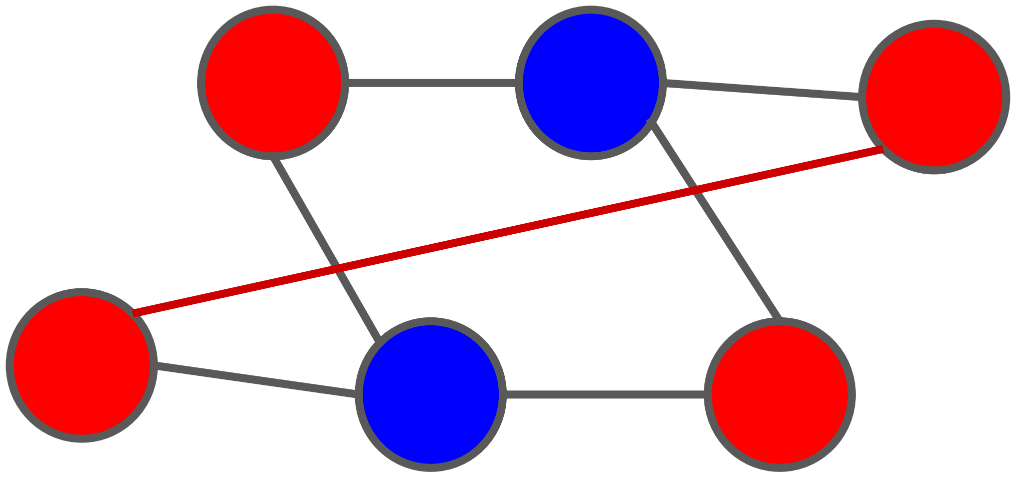
Another way to check: Try to color each vertex red or blue, such that no two edges share a color.

Is this graph bipartite? No!



Another way to check:
Try to color each
vertex red or blue,
such that no two
edges share a color.

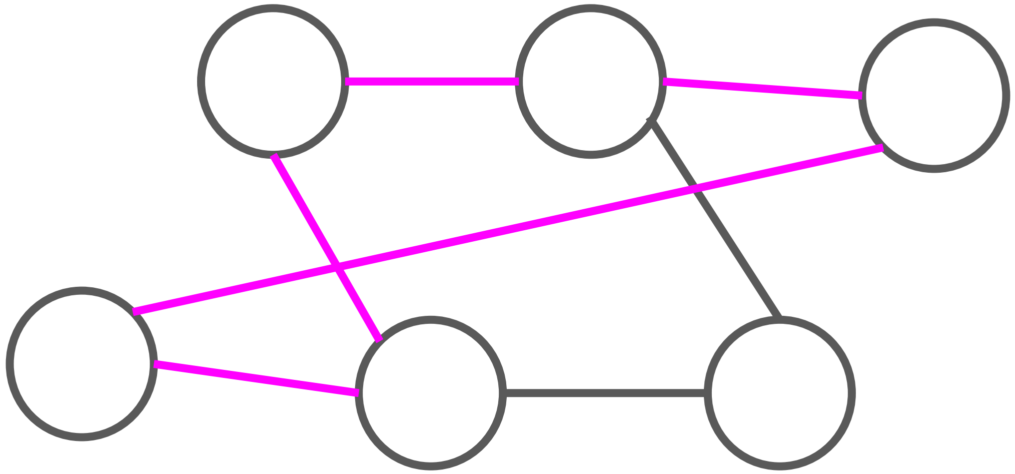
Is this graph bipartite? No!



Another way to check: Try to color each vertex red or blue, such that no two edges share a color.

We didn't even make any choices except where to start! And that doesn't matter...

Is this graph bipartite? No!



The culprit: a subtle odd cycle...

7/13 Lecture Agenda

- Announcements
- Part 4-1: Graphs and BFS
- 10 minute break!
- Part 4-2: Dijkstra's Algorithm

7/13 Lecture Agenda

- Announcements
- Part 4-1: Graphs and BFS
- 10 minute break!
- Part 4-2: Dijkstra's Algorithm

WORLD 4-2

The i, j, k In Dijkstra Are In Order

Divide and Conquer

Sorting & Randomization

Data Structures

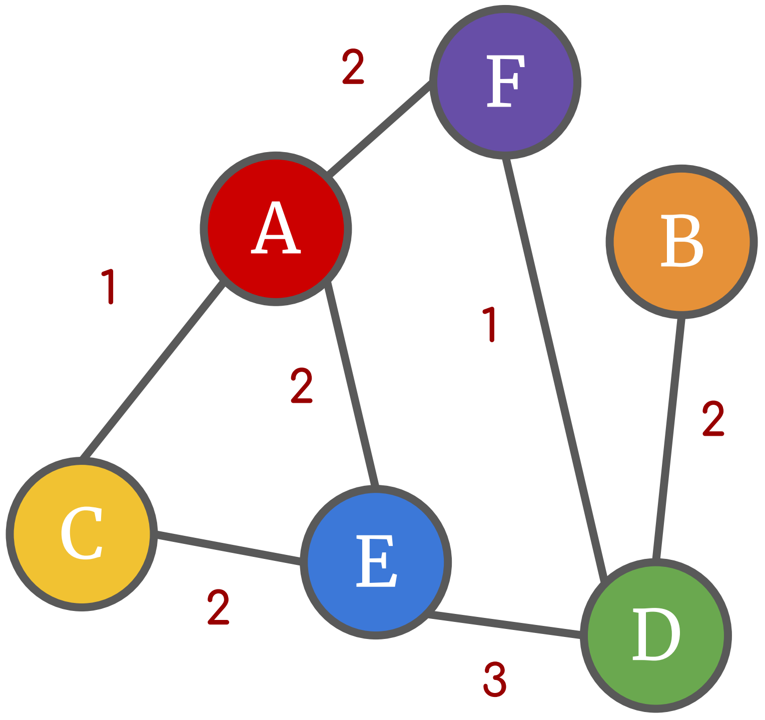
Graph Search

Dynamic Programming

Greed & Flow

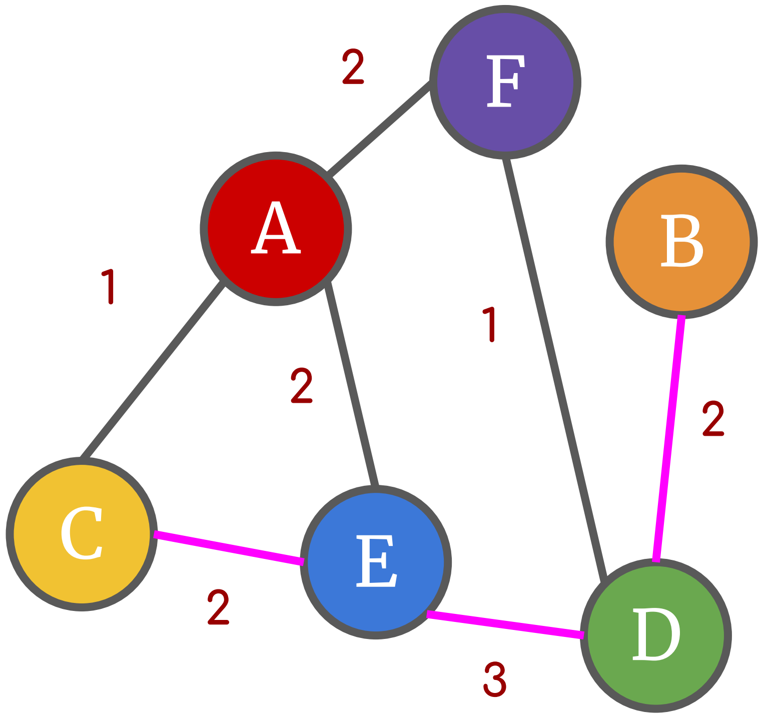
Special Topics

What if edges have weights?



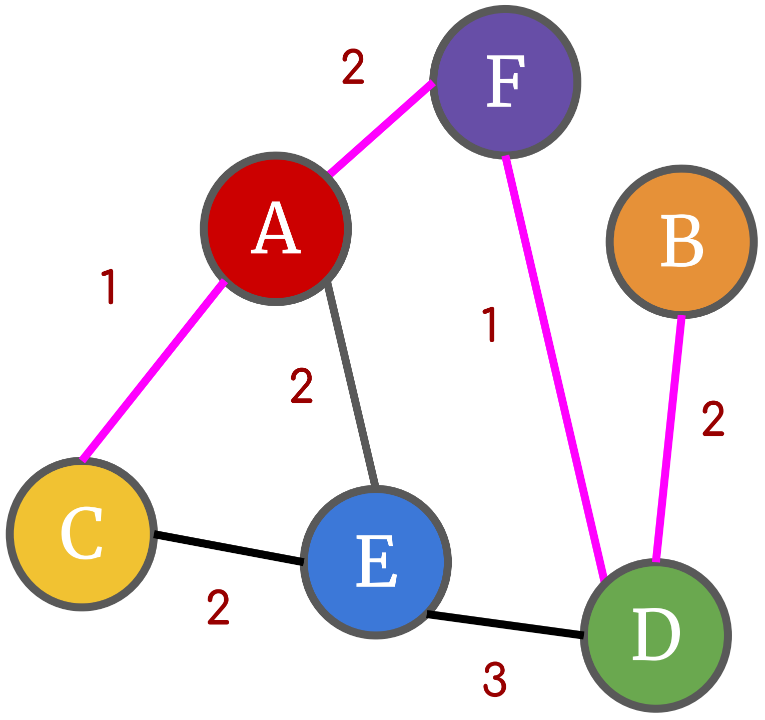
We'll give the vertices letters instead just so there aren't so many numbers flying around...

What's the lowest-cost path from C to D?



It's not necessarily the path with the fewest steps...

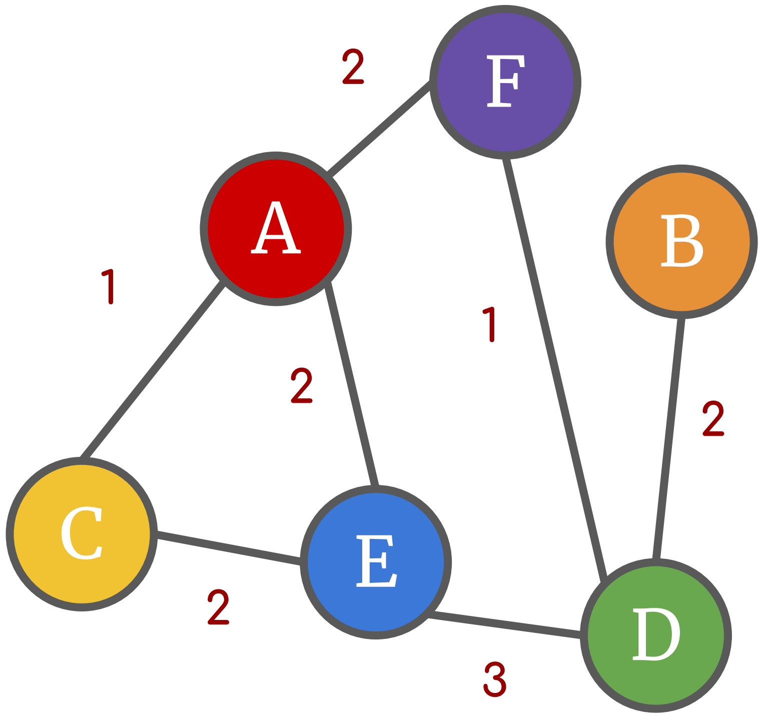
What's the lowest-cost path from C to D?



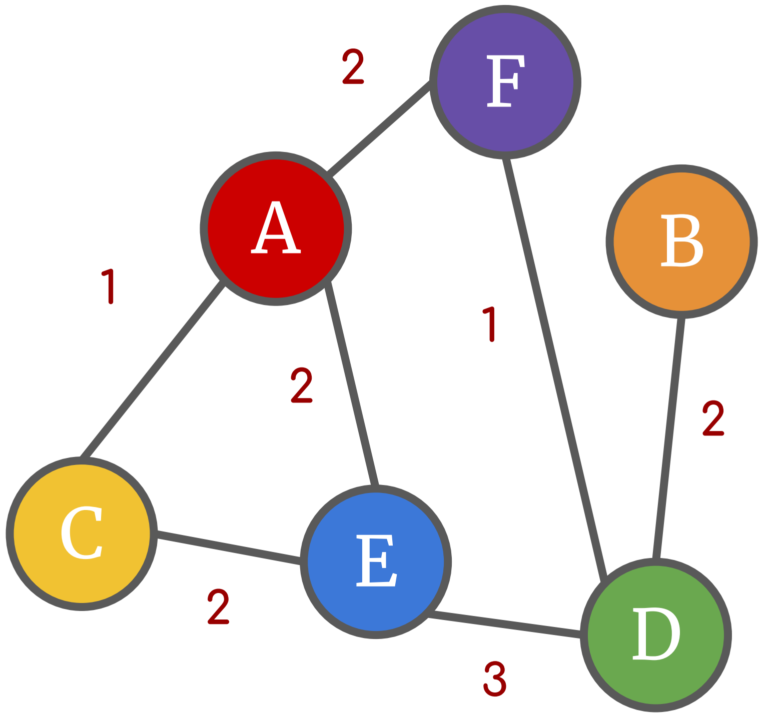
It's not necessarily the path with the fewest steps...

...this path with more steps is actually cheaper overall!

Can we adapt BFS to work?

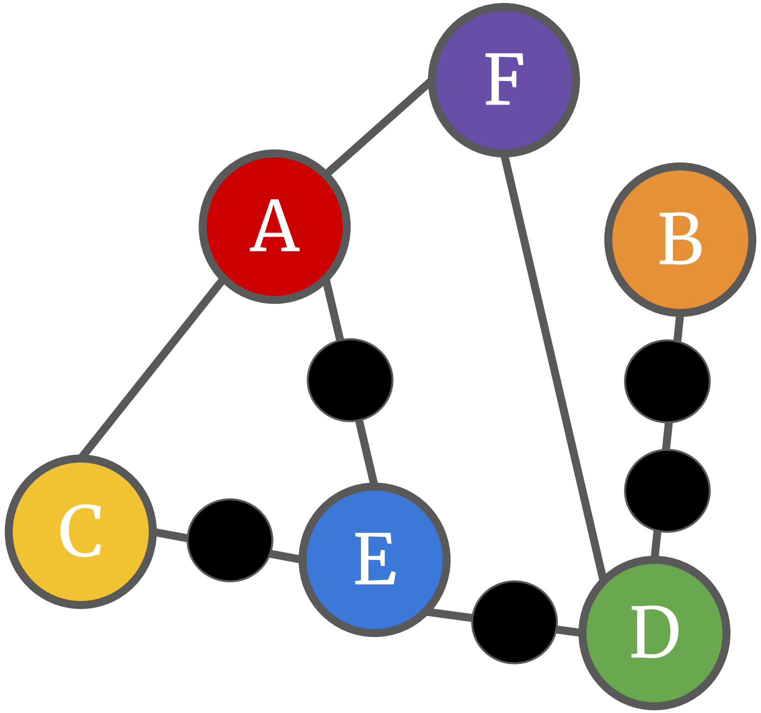


Can we adapt BFS to work?



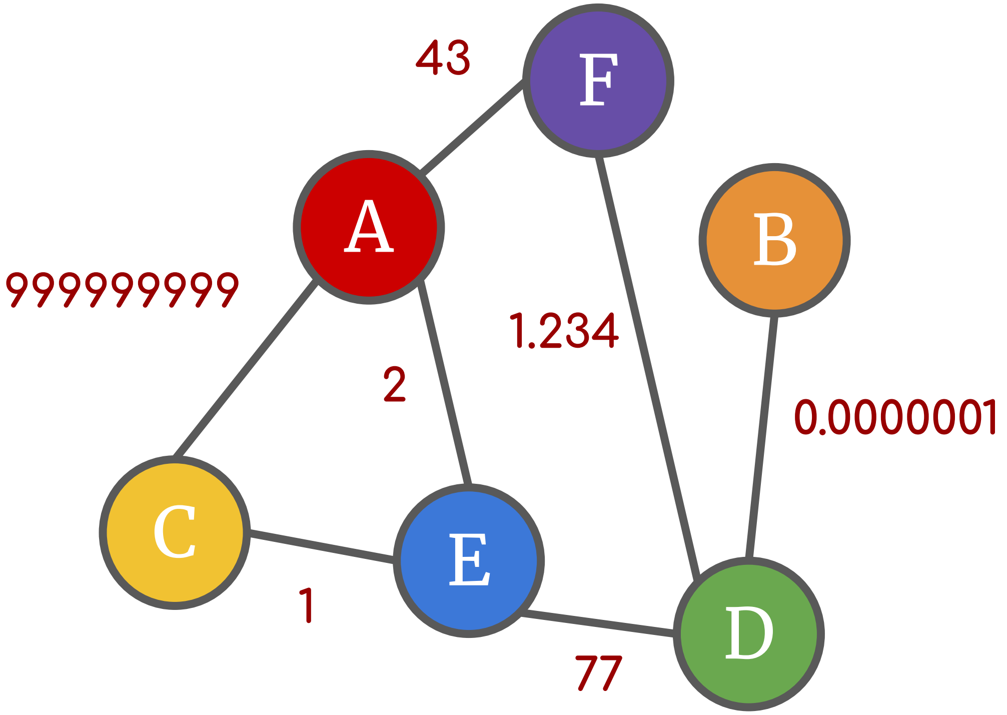
When the weights are very small positive integers, one hacky solution is to add fake nodes!

Can we adapt BFS to work? Kind of...



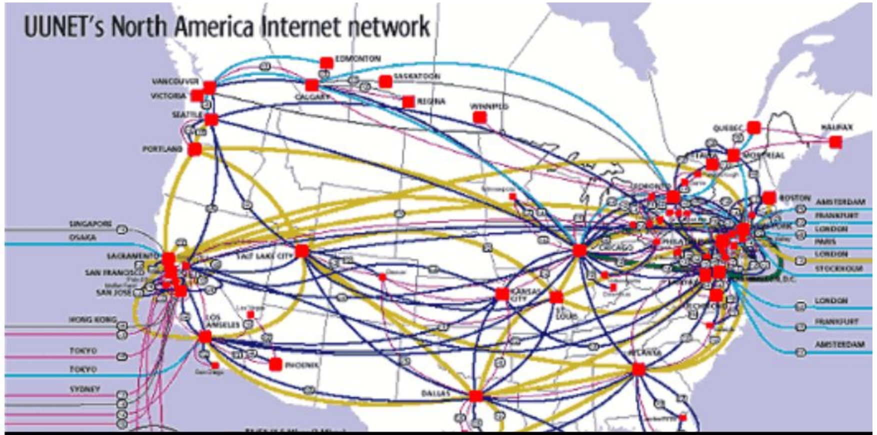
Now our BFS method for finding shortest distances works!

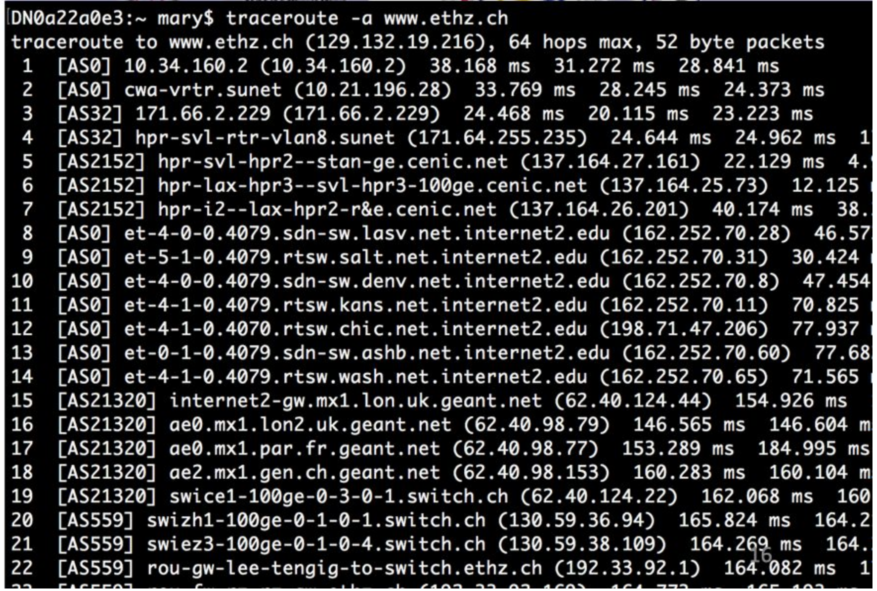
...but not always



Example

- **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?

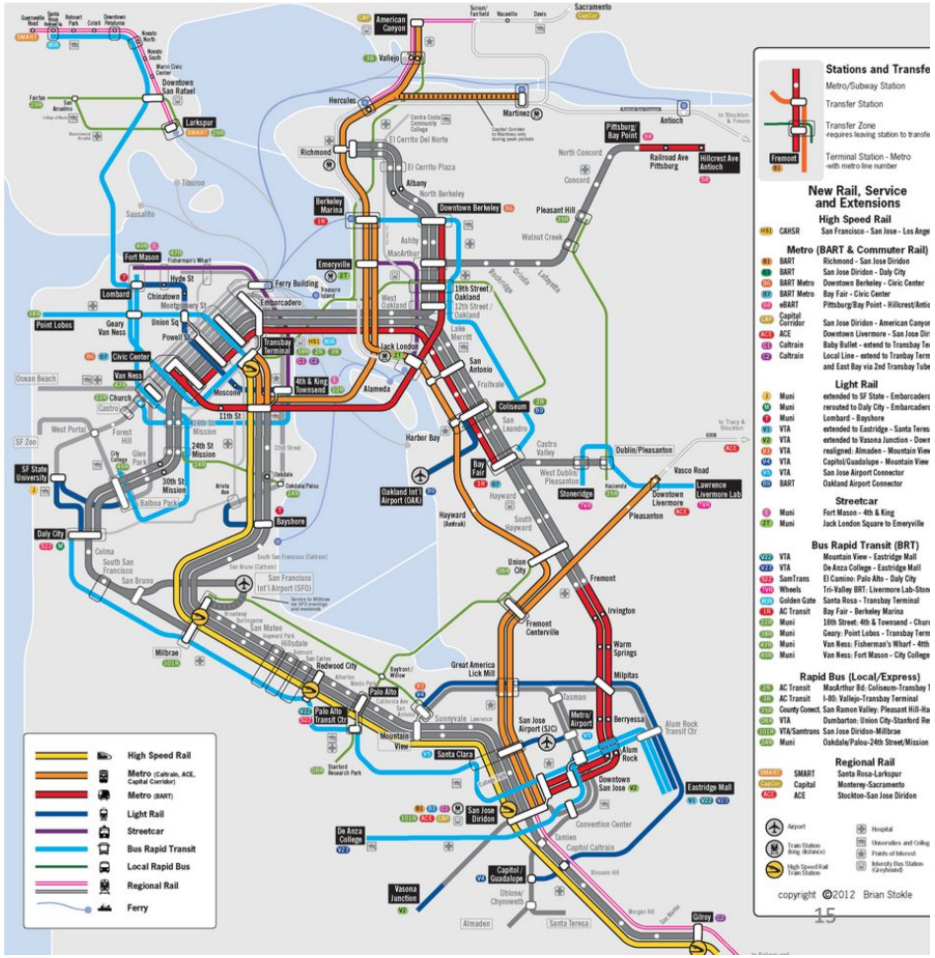




```
DN0a22a0e3:~ mary$ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1  [AS0] 10.34.160.2 (10.34.160.2)  38.168 ms  31.272 ms  28.841 ms
 2  [AS0] cwa-vrtr.sunet (10.21.196.28)  33.769 ms  28.245 ms  24.373 ms
 3  [AS32] 171.66.2.229 (171.66.2.229)  24.468 ms  20.115 ms  23.223 ms
 4  [AS32] hpr-svl-rtr-vlan8.sunet (171.64.255.235)  24.644 ms  24.962 ms  1
 5  [AS2152] hpr-svl-hpr2--stan-ge.cenic.net (137.164.27.161)  22.129 ms  4.
 6  [AS2152] hpr-lax-hpr3--svl-hpr3-100ge.cenic.net (137.164.25.73)  12.125
 7  [AS2152] hpr-i2--lax-hpr2-r&e.cenic.net (137.164.26.201)  40.174 ms  38.
 8  [AS0] et-4-0-0.4079.sdn-sw.lasv.net.internet2.edu (162.252.70.28)  46.57
 9  [AS0] et-5-1-0.4079.rtsw.salt.net.internet2.edu (162.252.70.31)  30.424
10  [AS0] et-4-0-0.4079.sdn-sw.denv.net.internet2.edu (162.252.70.8)  47.454
11  [AS0] et-4-1-0.4079.rtsw.kans.net.internet2.edu (162.252.70.11)  70.825
12  [AS0] et-4-1-0.4070.rtsw.chic.net.internet2.edu (198.71.47.206)  77.937
13  [AS0] et-0-1-0.4079.sdn-sw.ashb.net.internet2.edu (162.252.70.60)  77.68
14  [AS0] et-4-1-0.4079.rtsw.wash.net.internet2.edu (162.252.70.65)  71.565
15  [AS21320] internet2-gw.mx1.lon.uk.geant.net (62.40.124.44)  154.926 ms
16  [AS21320] ae0.mx1.lon2.uk.geant.net (62.40.98.79)  146.565 ms  146.604 m
17  [AS21320] ae0.mx1.par.fr.geant.net (62.40.98.77)  153.289 ms  184.995 ms
18  [AS21320] ae2.mx1.gen.ch.geant.net (62.40.98.153)  160.283 ms  160.104 m
19  [AS21320] swice1-100ge-0-3-0-1.switch.ch (62.40.124.22)  162.068 ms  160
20  [AS559] swizh1-100ge-0-1-0-1.switch.ch (130.59.36.94)  165.824 ms  164.2
21  [AS559] swiez3-100ge-0-1-0-4.switch.ch (130.59.38.109)  164.269 ms  164.
22  [AS559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1)  164.082 ms  1
```

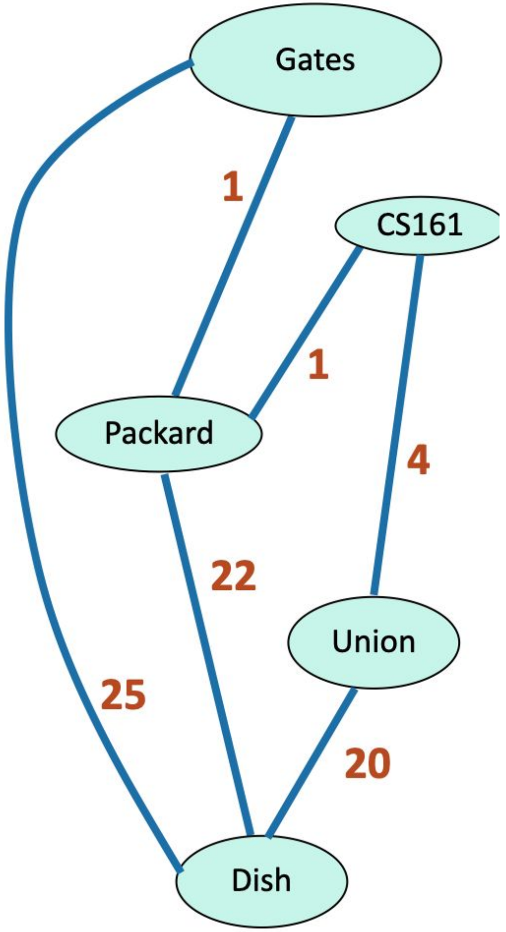
Example

- **"what is the shortest path from Palo Alto to [anywhere else]"** using BART, Caltrain, lightrail, MUNI, bus, Amtrak, bike, walking, uber/lyft.
- Edge weights have something to do with time, money, hassle.



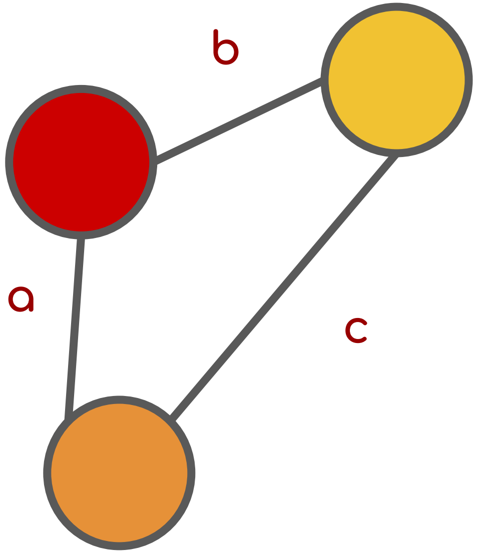
Dijkstra's algorithm

- Finds shortest paths from Gates to everywhere else.



A quick word on the triangle inequality

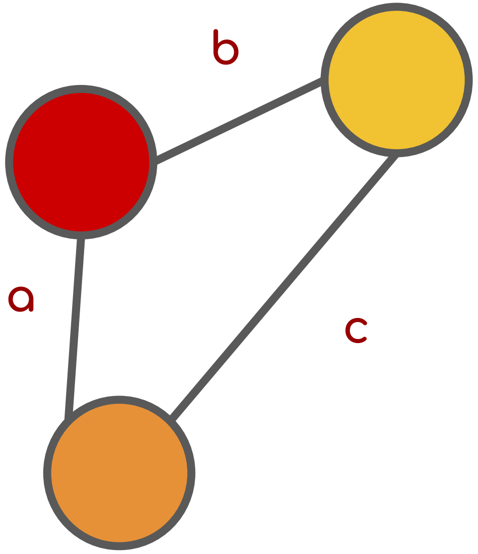
For pathfinding in the real world, it might seem like we can also use the triangle inequality:



$$a + b \geq c$$

A quick word on the triangle inequality

For pathfinding in the real world, it might seem like we can also use the triangle inequality:

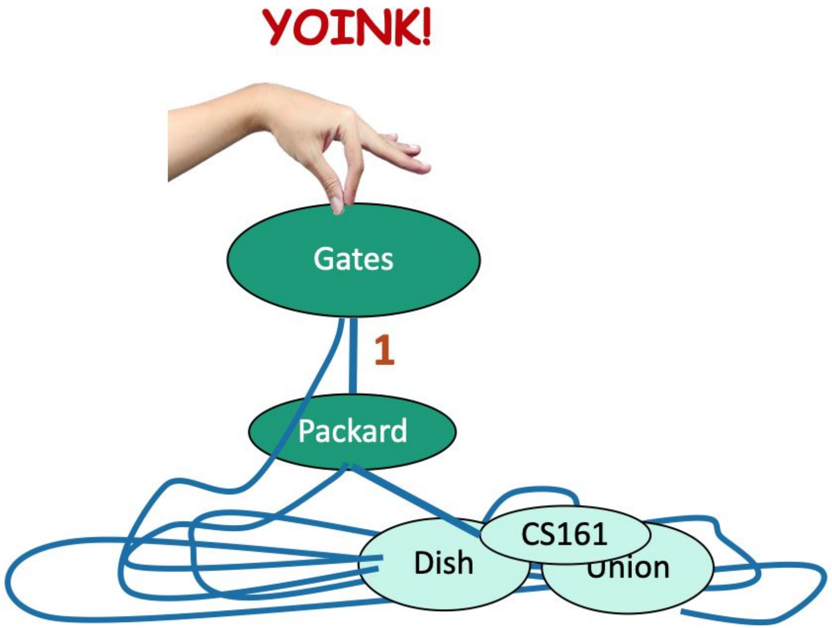


$a + b \geq c$

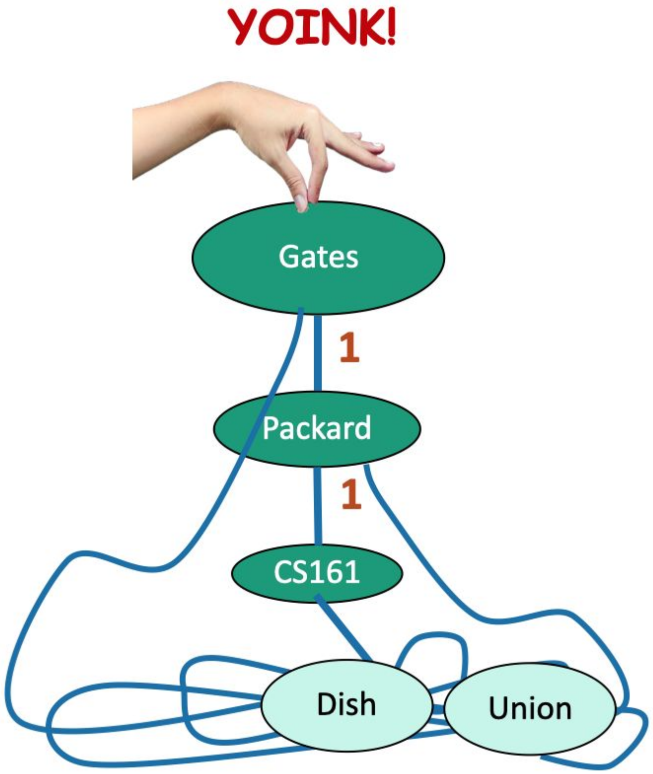
This would be true in a flat plane, but what if path *c* has a big hill in the middle that *a* and *b* avoid? Or what if one path is muddy?

Even the same path might take different amounts of time in different directions!

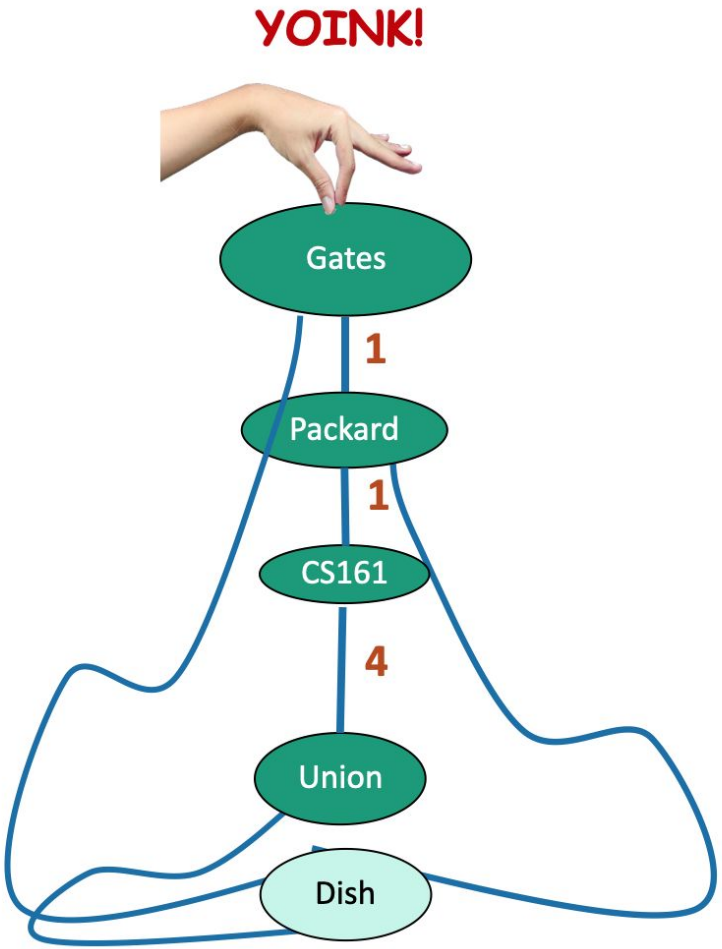
Dijkstra
intuition



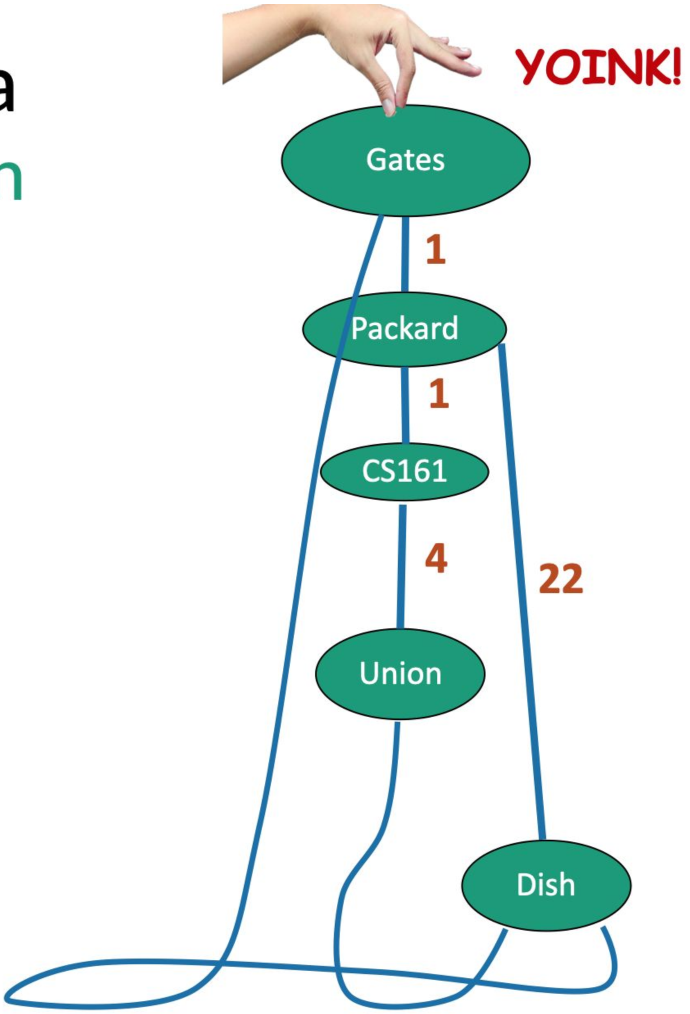
Dijkstra
intuition



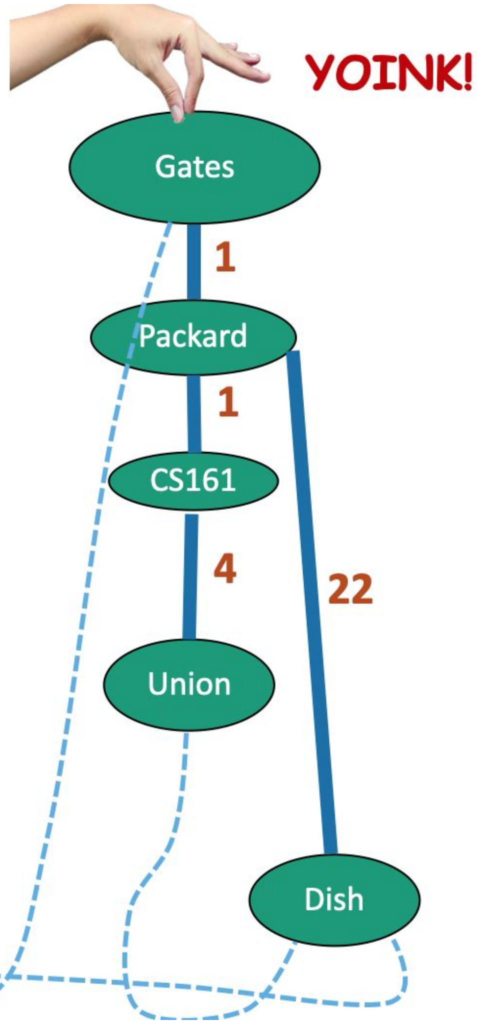
Dijkstra
intuition



Dijkstra
intuition



Dijkstra intuition



This creates a tree!

The shortest paths are the lengths along this tree.

Motion sickness warning

I used one of Mary's examples but had some trouble copying it in a nice way.

The blame for any shifting around, small size changes, cut-off edges, etc. rests entirely upon me!

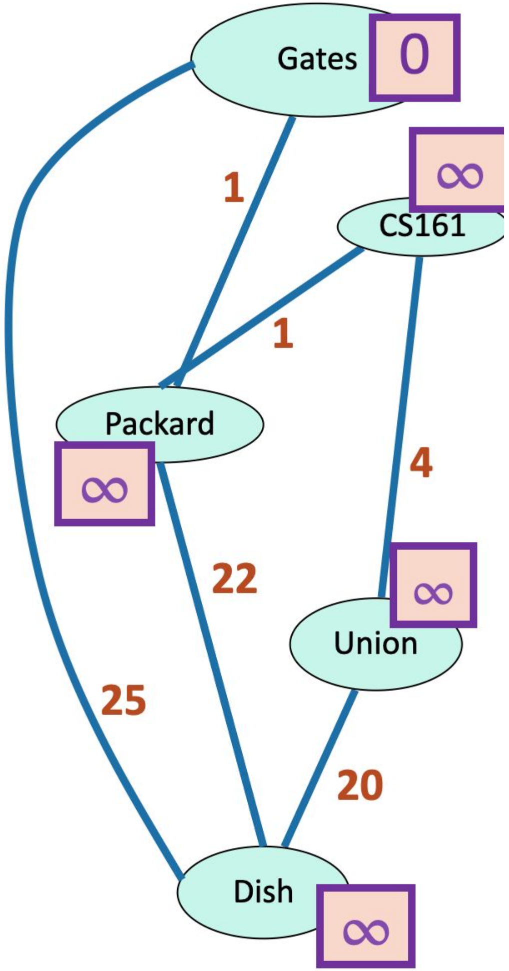
Dijkstra by example

How far is a node from Gates?

○	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).

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].**

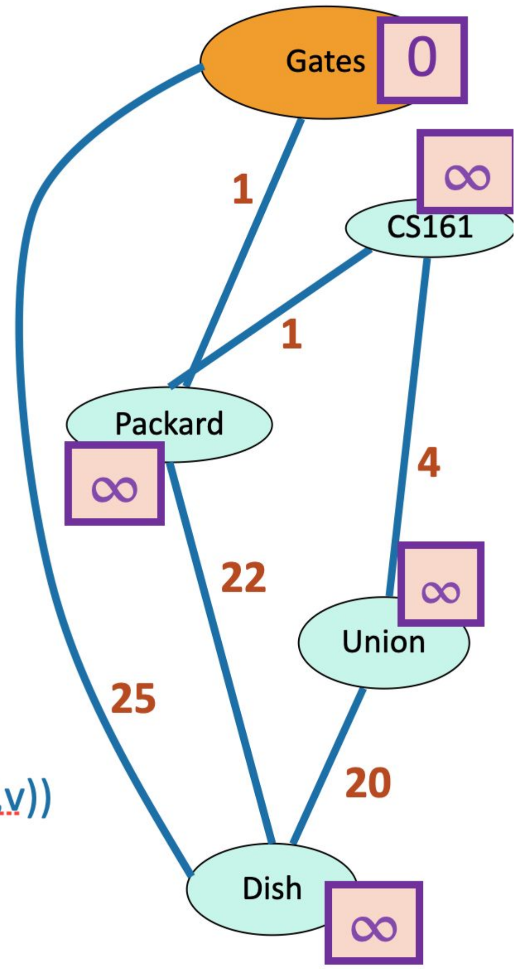


Dijkstra by example

How far is a node from Gates?

- ⬭ I'm not sure yet
- ⬬ I'm sure
- $\boxed{x}$ **x = d[v]** is my best **over-estimate** for dist(Gates,v).
- ⬬ Current node u

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

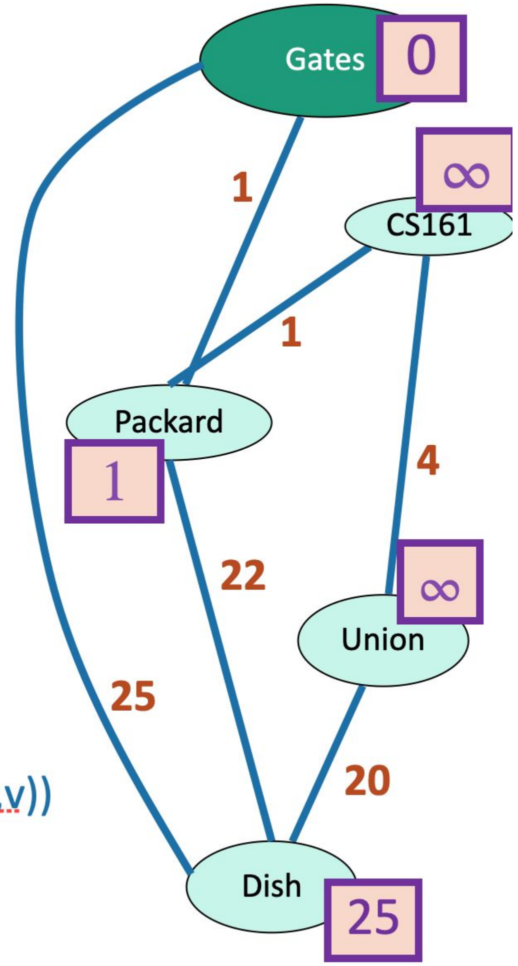


Dijkstra by example

How far is a node from Gates?

●	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
●	Current node u

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

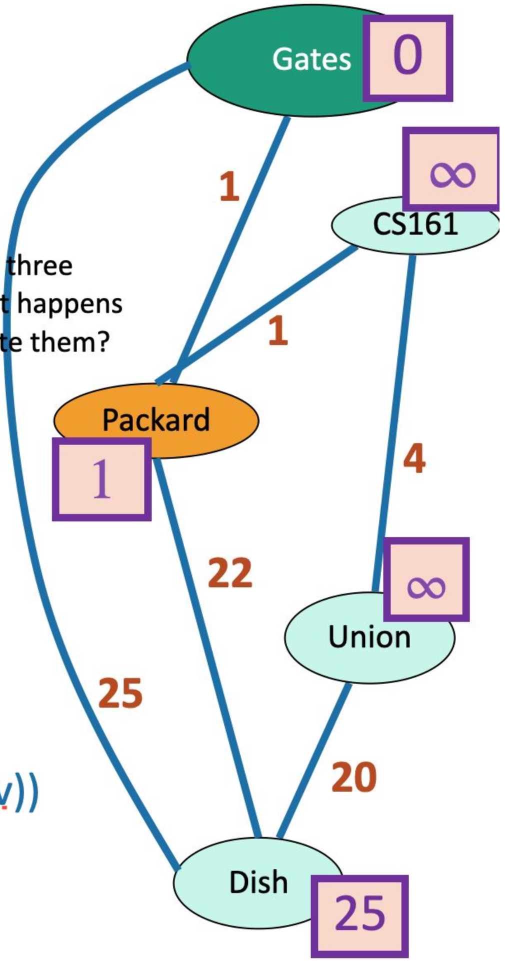


Dijkstra by example

How far is a node from Gates?

⬭	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
●	Current node u

Packard has three neighbors. What happens when we update them?



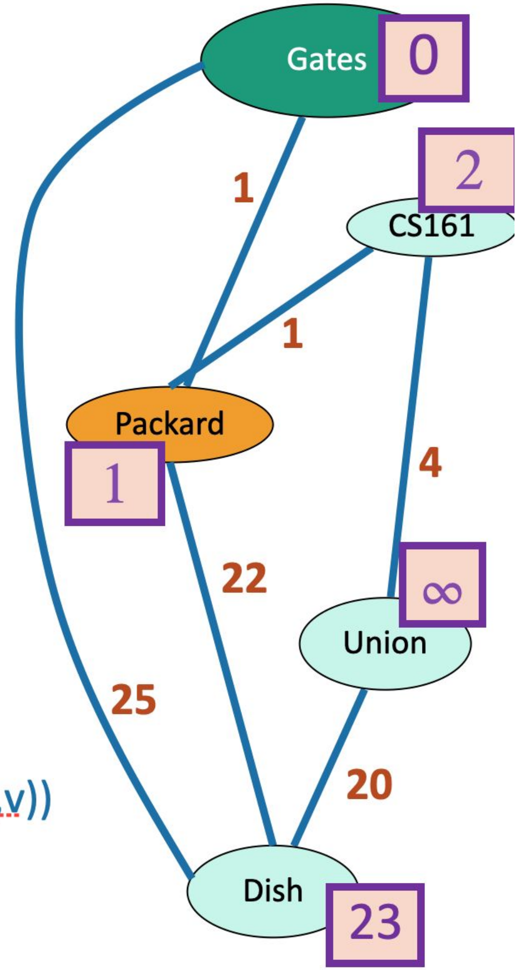
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from Gates?

◯	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
●	Current node u

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

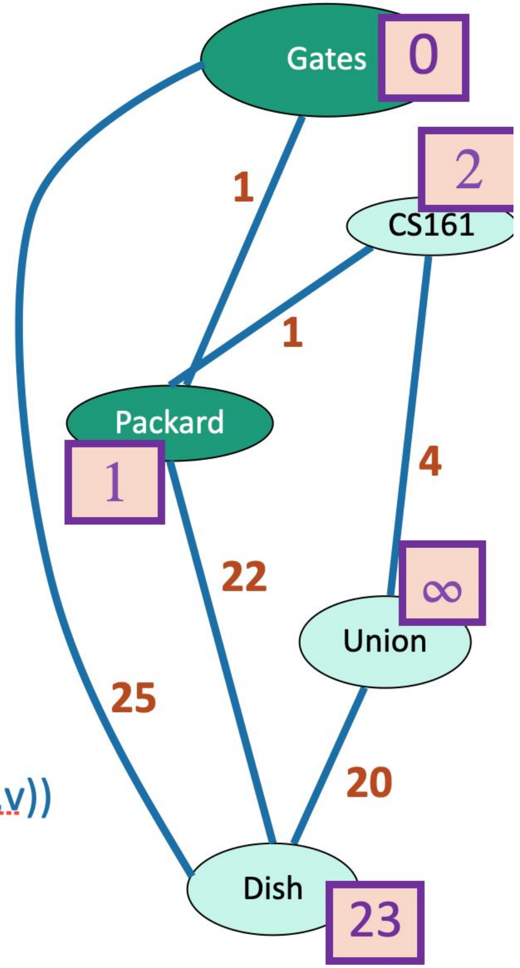


Dijkstra by example

How far is a node from Gates?

◯	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
●	Current node u

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

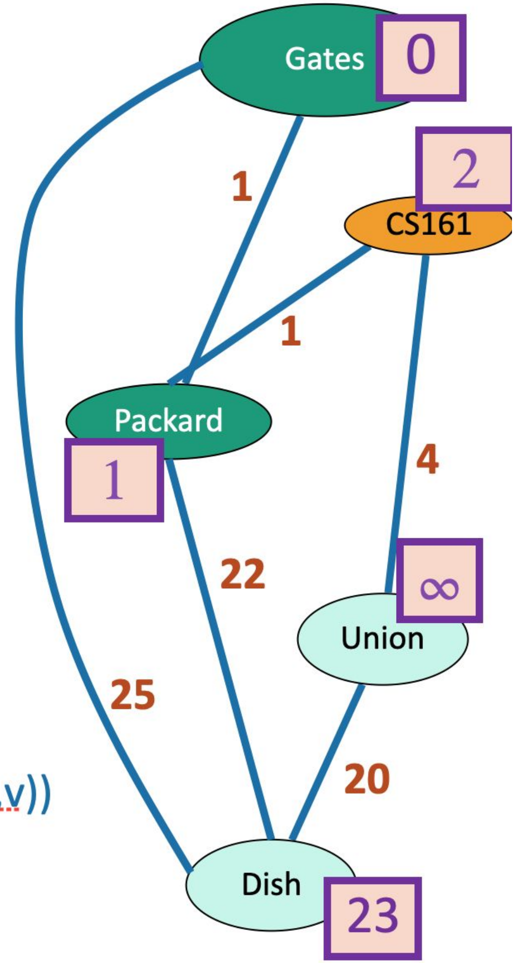


Dijkstra by example

How far is a node from Gates?

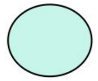
⬭	I'm not sure yet
⬬	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
⬬	Current node u

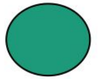
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



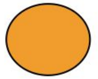
Dijkstra by example

How far is a node from Gates?

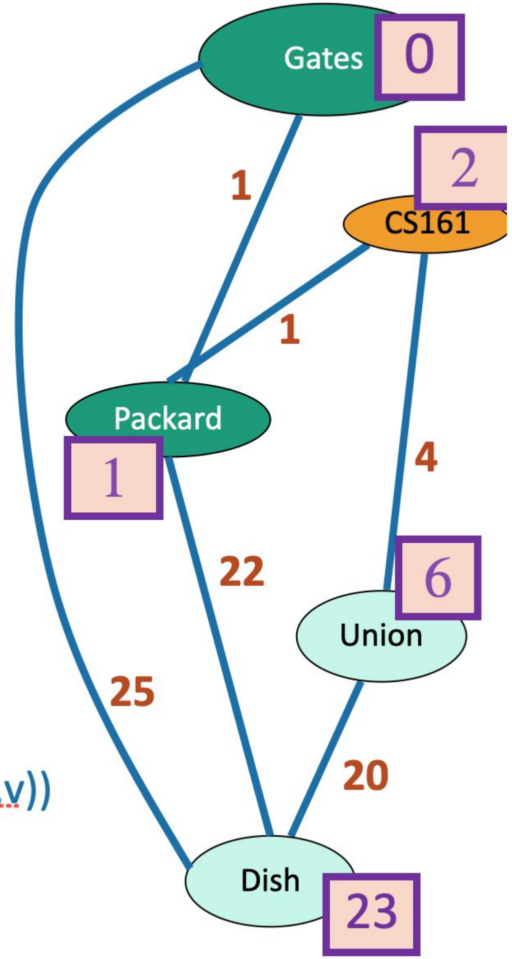

I'm not sure yet


I'm sure

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


Current node u

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

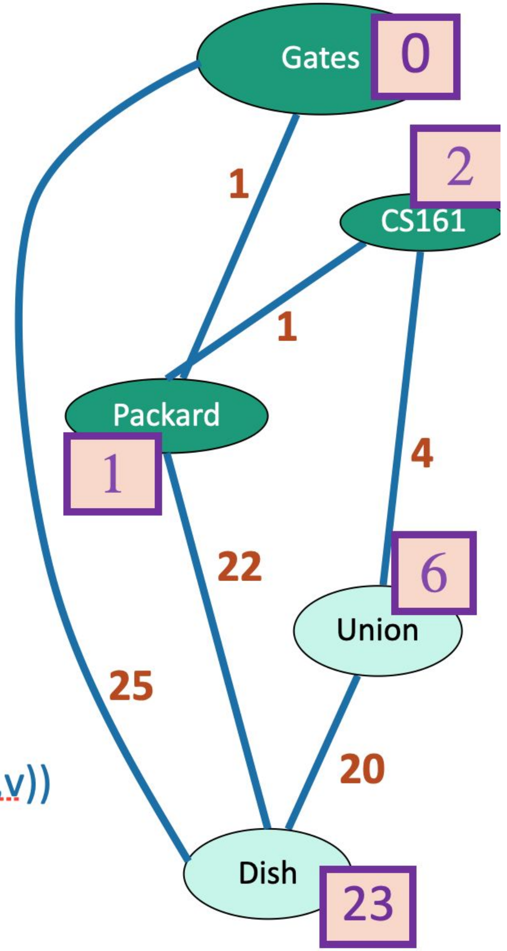


Dijkstra by example

How far is a node from Gates?

○	I’m not sure yet
●	I’m sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
●	Current node u

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

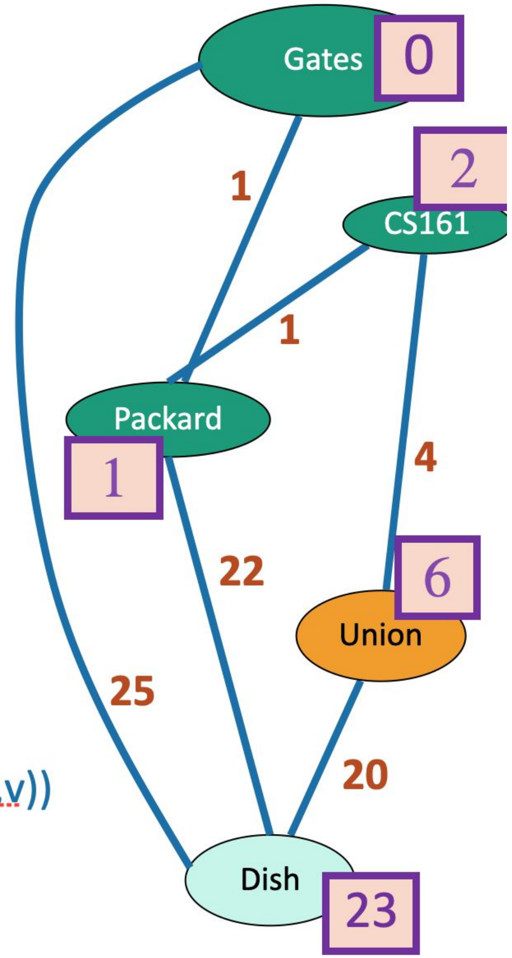


Dijkstra by example

How far is a node from Gates?

⬭	I'm not sure yet
⬬	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
⬬	Current node u

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

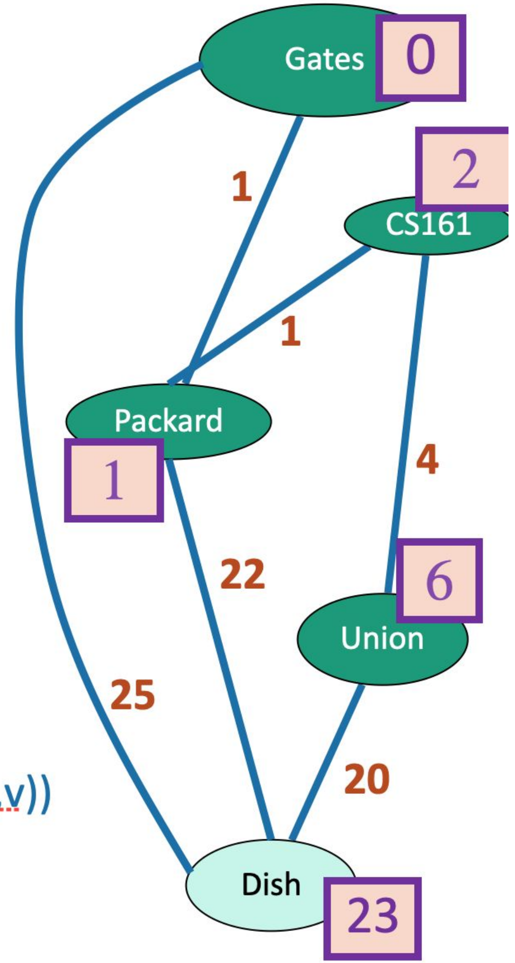


Dijkstra by example

How far is a node from Gates?

○	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
●	Current node u

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

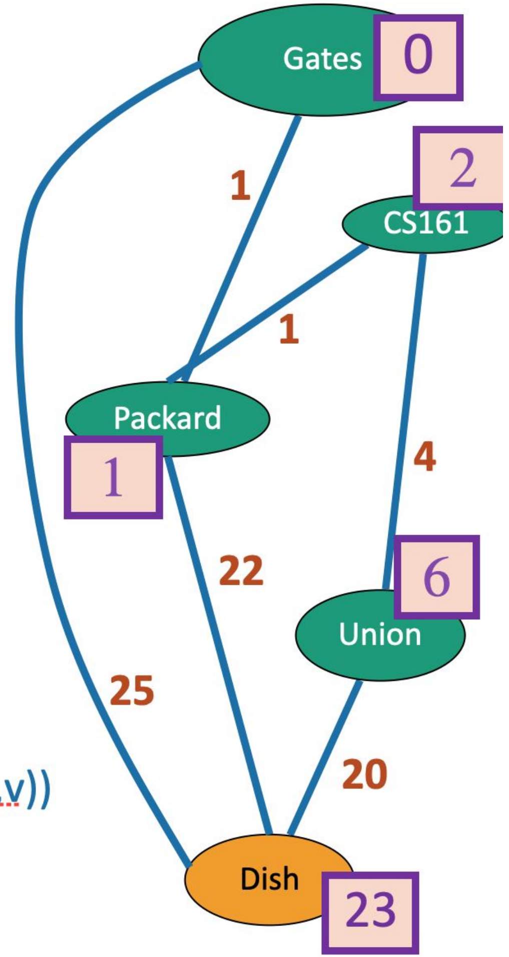


Dijkstra by example

How far is a node from Gates?

- ◯ I'm not sure yet
- 🟢 I'm sure
- $\boxed{x}$ **x = d[v]** is my best **over-estimate** for dist(Gates,v).
- 🟠 Current node u

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

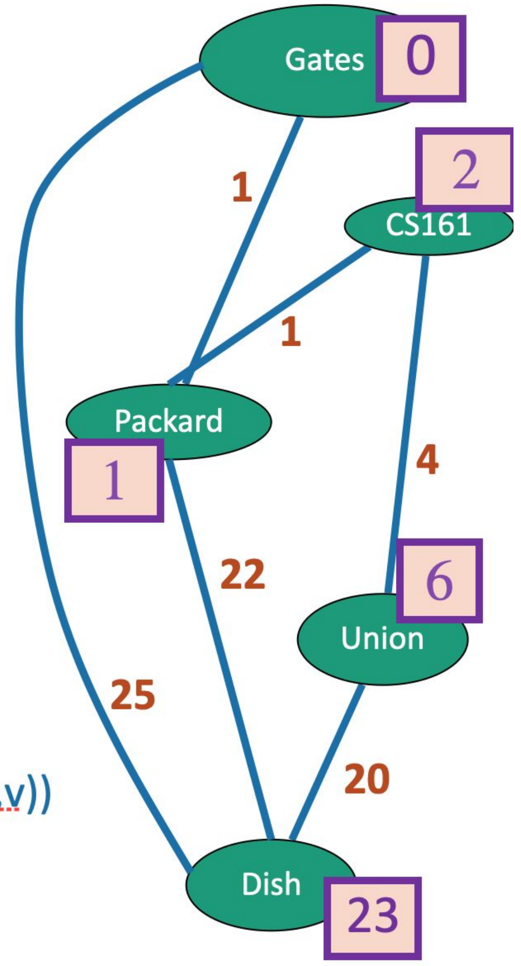


Dijkstra by example

How far is a node from Gates?

○	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
●	Current node u

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



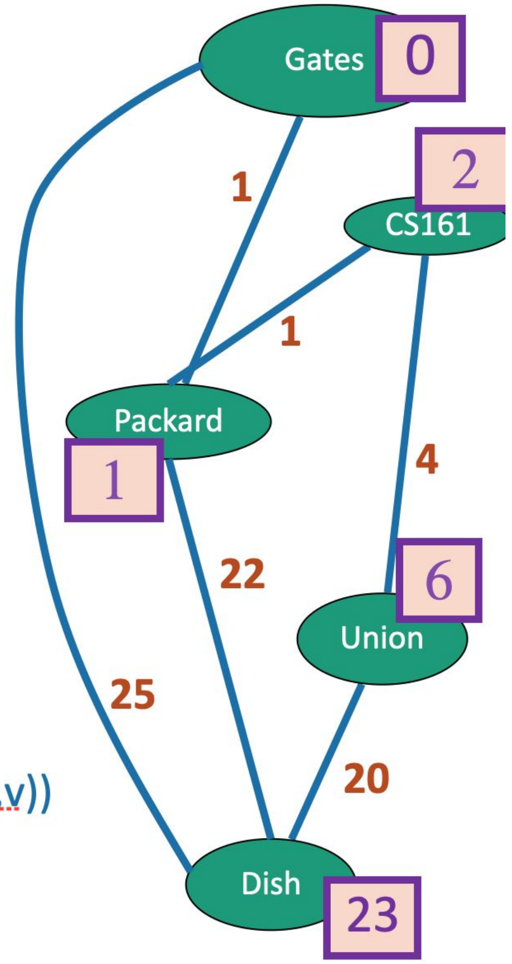
Dijkstra by example

How far is a node from Gates?

◯	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(Gates,v).
●	Current node u

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

After all nodes are **sure**, say that d(Gates, 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]
```

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

How can we be sure this works?

We'll prove it together on Homework 4!

(Note: that means you're not responsible for the details for the Midterm)

We need a data structure that:

- Stores unsure vertices v
- Keeps track of d[v]
- Can find u with minimum d[u]
    - `findMin()`
- Can remove that u
    - `removeMin(u)`
- Can update (decrease) d[v]
    - `updateKey(v,d)`

Just the inner loop:

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

We need a data structure that:

- Stores unsure vertices v
- Keeps track of d[v]
- Can find u with minimum d[u]
  - `findMin()`
- Can remove that u
  - `removeMin(u)`
- Can update (decrease) d[v]
  - `updateKey(v,d)`

Just the inner loop:

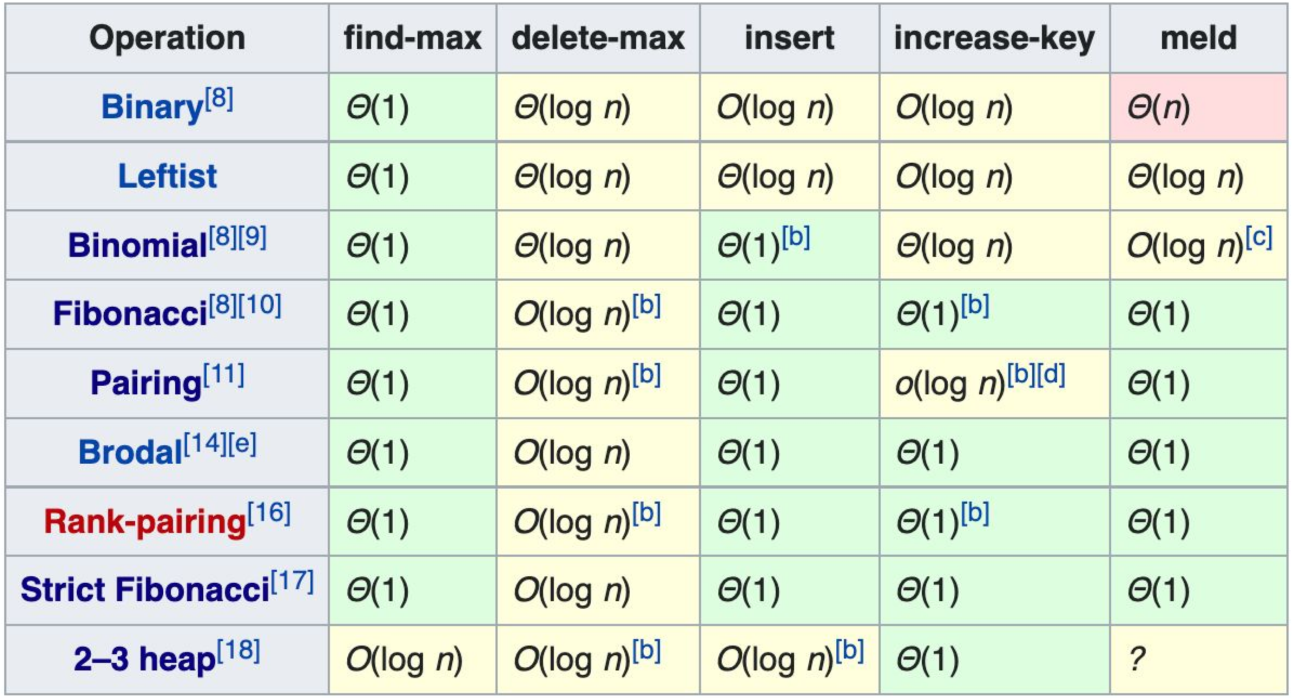
- Pick the not-sure node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] ← min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.

Total running time is big-O of:

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

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

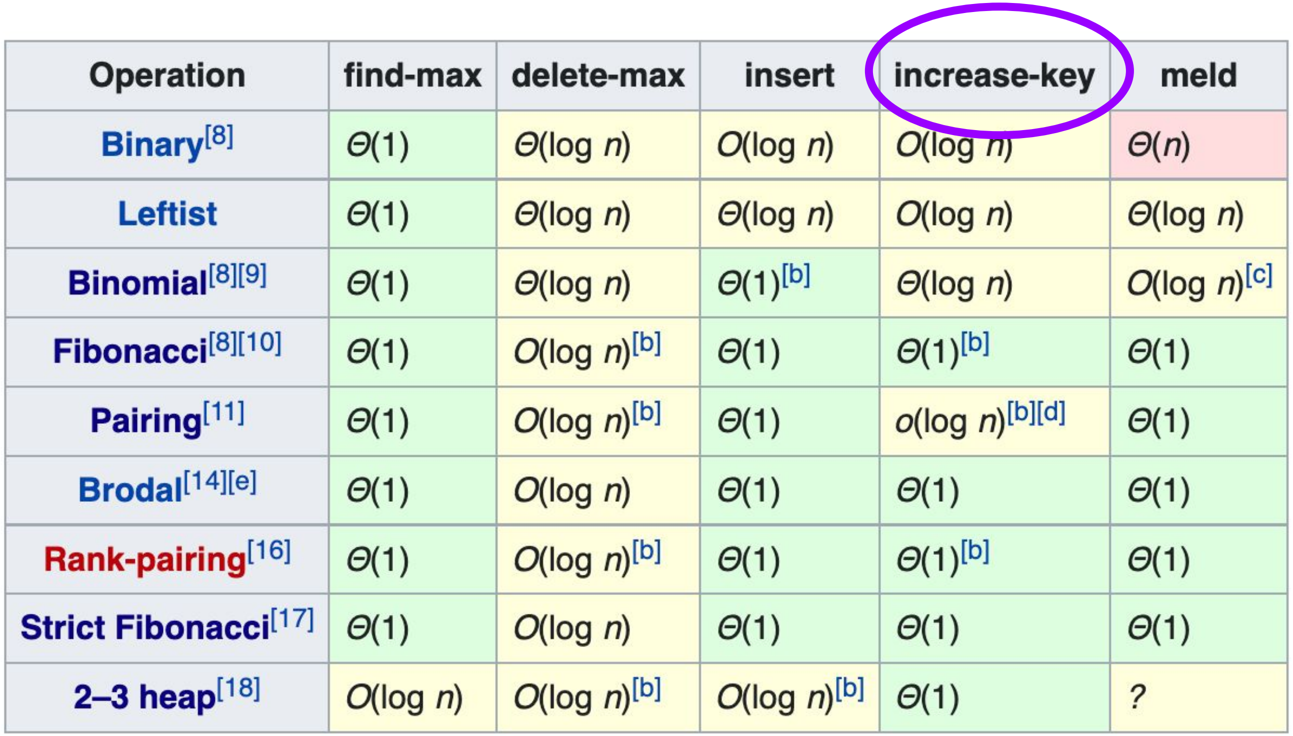
What about heaps?



Operation	find-max	delete-max	insert	increase-key	meld
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[8][9]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(\log n)$	$O(\log n)$[c]
Fibonacci[8][10]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Pairing[11]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$o(\log n)$[b][d]	$\Theta(1)$
Brodal[14][e]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[16]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Strict Fibonacci[17]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[18]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$\Theta(1)$	?

What about heaps?

This (well, decrease-key) is something we need to do a lot in Dijkstra's!

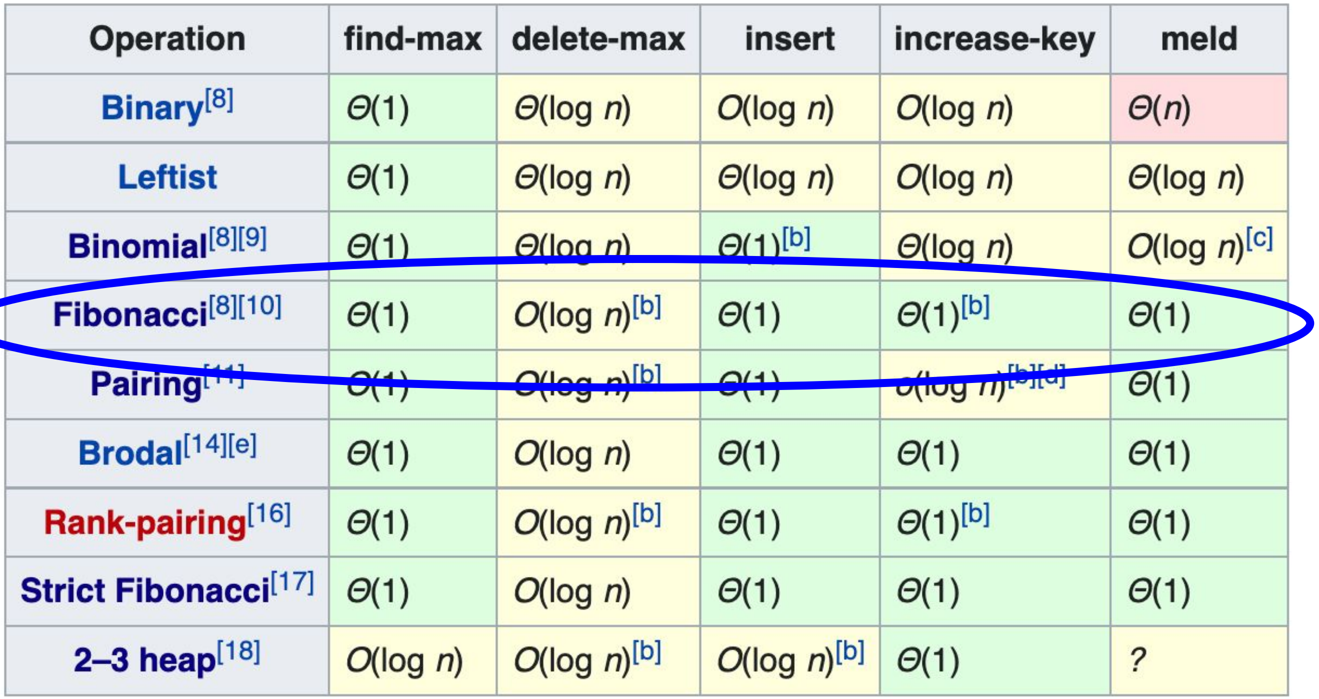


Operation	find-max	delete-max	insert	increase-key	meld
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[8][9]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(\log n)$	$O(\log n)$[c]
Fibonacci[8][10]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Pairing[11]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$o(\log n)$[b][d]	$\Theta(1)$
Brodal[14][e]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[16]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Strict Fibonacci[17]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[18]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$\Theta(1)$	?

What about heaps?

This (well, decrease-key) is something we need to do a lot in Dijkstra's!

Hmm...



Operation	find-max	delete-max	insert	increase-key	meld
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[8][9]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(\log n)$	$O(\log n)$[c]
Fibonacci[8][10]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Pairing[11]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$o(\log n)$[b][d]	$\Theta(1)$
Brodal[14][e]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[16]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Strict Fibonacci[17]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[18]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$\Theta(1)$	?

Fibonacci heaps!

Actually pretty simple! Here's how they work:

Fibonacci heaps!

Actually pretty simple! Here's how they work:

Fibonacci heaps!

~~Actually pretty simple! Here's how they work:~~ *j/k they're horrid*

Operation **extract minimum** (same as *delete minimum*) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was d, it takes time $O(d)$ to process all new roots and the potential increases by $d-1$. Therefore, the amortized running time of this phase is $O(d) = O(\log n)$.

However to complete the extract minimum operation, we need to update the pointer to the root with minimum key. Unfortunately there may be up to n roots we need to check. In the second phase we therefore decrease the number of roots by successively linking together roots of the same degree. When two roots u and v have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root. Its degree will increase by one. This is repeated until every root has a different degree. To find trees of the same degree efficiently we use an array of length $O(\log n)$ in which we keep a pointer to one root of each degree. When a second root is found of the same degree, the two are linked and the array is updated. The actual running time is $O(\log n + m)$ where m is the number of roots at the beginning of the second phase. At the end we will have at most $O(\log n)$ roots (because each has a different degree). Therefore, the difference in the potential function from before this phase to after it is: $O(\log n) - m$, and the amortized running time is then at most $O(\log n + m) + c(O(\log n) - m)$. With a sufficiently large choice of c, this simplifies to $O(\log n)$.

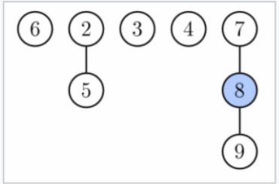


Figure 2. Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

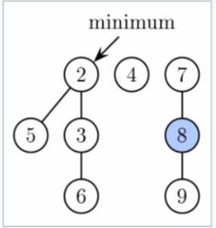


Figure 3. Fibonacci heap from Figure 1 after extract minimum is completed. First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

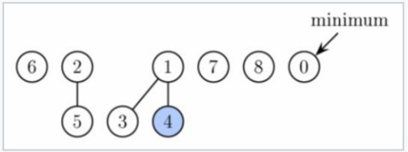


Figure 4. Fibonacci heap from Figure 1 after decreasing key of node 9 to 0. This node as well as its two marked ancestors are cut from the tree rooted at 1 and placed as new roots.

For this class, you don't need to know how they work, just the time guarantees...

Say we use a Fibonacci Heap

- T(findMin) = O(1) (amortized time*)
- T(removeMin) = O(log(n)) (amortized time*)
- T(updateKey) = O(1) (amortized time*)
- See CS166 for more!
- Running time of Dijkstra

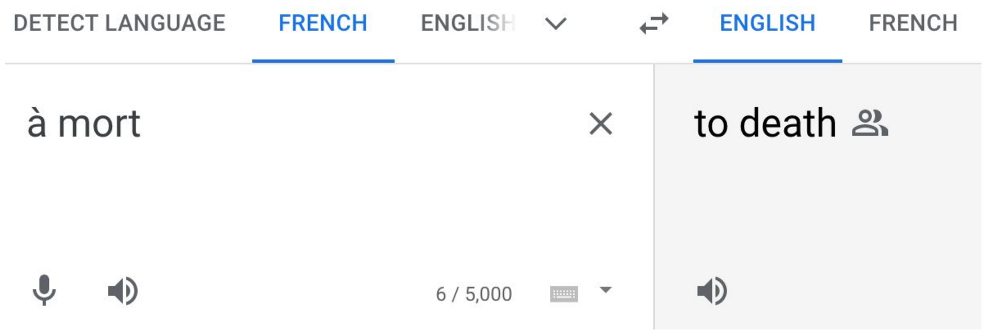
  = O(n(T(`findMin`) + T(`removeMin`)) + m T(`updateKey`))

  = O(nlog(n) + m) (amortized time)

*This means that 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..

A quick note on amortized analysis

- Accounting sense: this washing machine costs \$700 but we'll get to use it for 20 years, so \$35/yr!



à mort

to death

- CS sense: If we say something is O(1) amortized, then at **any** point, the average cost so far per operation is O(1). We can't take out a loan!

Example: inserting into an array and resizing

- Say that an array holds *n* elements.
- Insertion is O(1) since we know where the end of the array is.
- But when we run out of room, we have to create a new array with capacity 2*n*, and move all the existing elements there before inserting...
- Isn't this "exponential"? How could it be O(1) in any sense?

Initial size: 2

Insertion #	Cost	Average cost so far
1	1	1/1 = 1
2	1	2/2 = 1
3	4 (make new) + 2 (move) + 1	9/3 = 3
4	1	10/4 = 2.5
5	8 (make new) + 4 (move) + 1	23/5 = 4.6
6	1	24/6 = 4
7	1	25/7 = 3.6
8	1	26/8 = 3.3
9	16 (make new) + 8 (move) + 1	51/9 = 5.7

Insertion #	Cost	Average cost so far
1	1	1/1 = 1 <= 7
2	1	2/2 = 1 <= 7
3	4 (make new) + 2 (move) + 1	9/3 = 3 <= 7
4	1	10/4 = 2.5 <= 7
5	8 (make new) + 4 (move) + 1	23/5 = 4.6 <= 7
6	1	24/6 = 4 <= 7
7	1	25/7 = 3.6 <= 7
8	1	26/8 = 3.3 <= 7
9	16 (make new) + 8 (move) + 1	51/9 = 5.7 <= 7
65537	131072 (make new) + 65536 (move) + 1	458747/65537 = 6.9998 <= 7

No matter how long this goes on, it turns out the average can never exceed 7. So insertion is O(1) amortized.

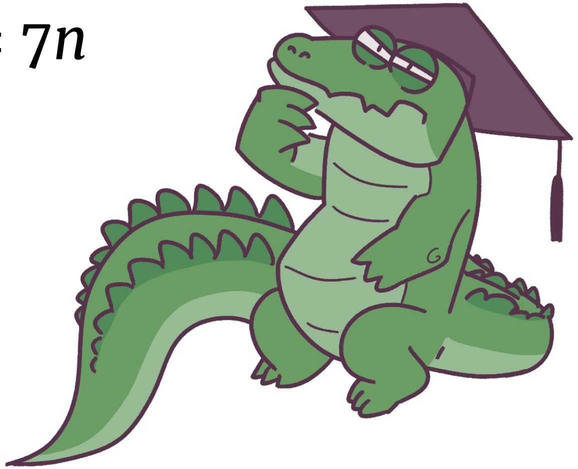
A little more rigor

Total cost after *n* inserts:

$n * 1 + (2^{\text{ceil}(\log n)} + 2^{\text{ceil}(\log n) - 1} + \ldots) + (2^{\text{ceil}(\log n) - 1} + 2^{\text{ceil}(\log n) - 2} + \ldots)$

$\leq n + 2^{\text{ceil}(\log n) + 1} + 2^{\text{ceil}(\log n)} \leq n + 4n + 2n = 7n$

On average, this is $\leq 7n / n = 7$



Amortization takeaway

- "O(1) amortized" does **not** mean that every instance of the operation is O(1).

- But it **does** mean – at any point in the process – that the average cost of all such operations so far is O(1).

- Again: no taking out loans! We can't start with one O(*n*) instance and then pay for it with a bunch of O(1) instances.

"I'll get the money, I swear!"

Say we use a Fibonacci Heap

- T(findMin) = O(1) (amortized time*)
- T(removeMin) = O(log(n)) (amortized time*)
- T(updateKey) = O(1) (amortized time*)
- See CS166 for more!
- Running time of Dijkstra

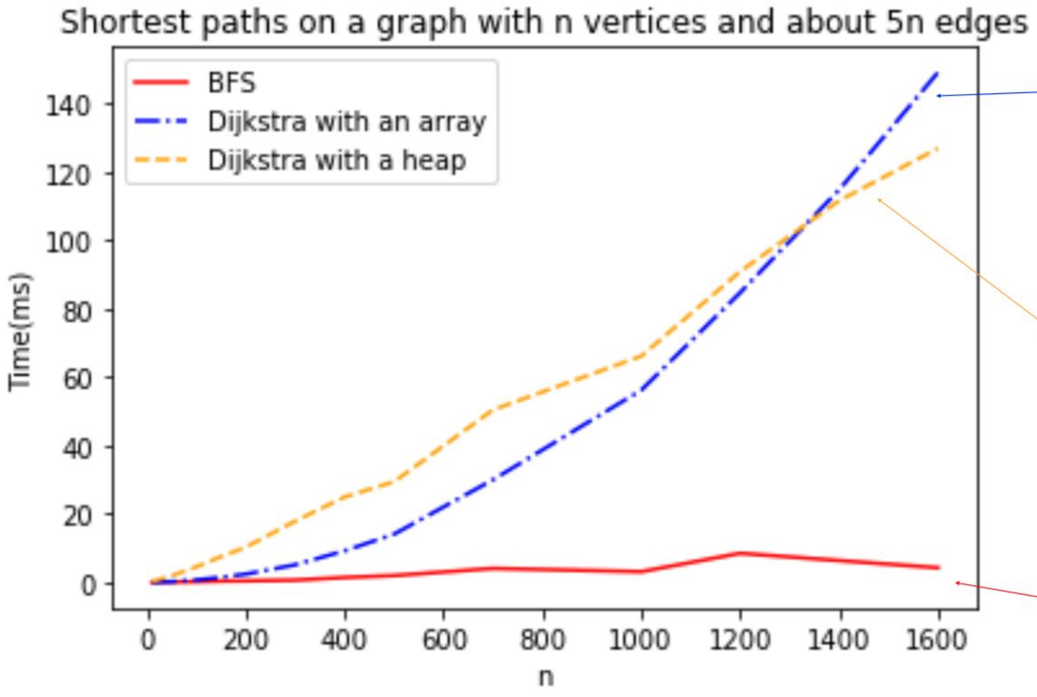
  = O(n(T(`findMin`) + T(`removeMin`)) + m T(`updateKey`))

  = O(nlog(n) + m) (amortized time)

*This means that 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

The heap is implemented using `heapdict`



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

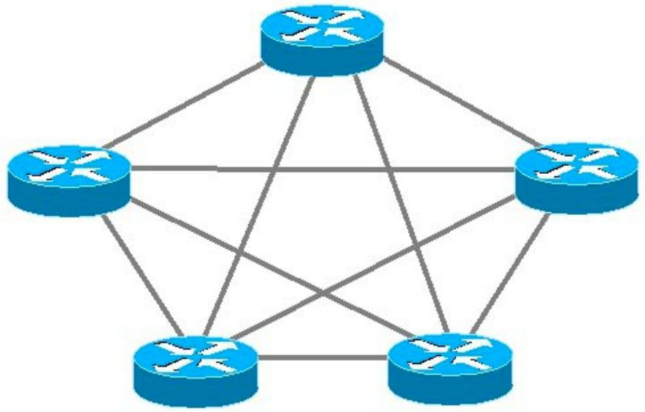
Dijkstra using a heap looks a bit more linear (actually nlog(n))

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.

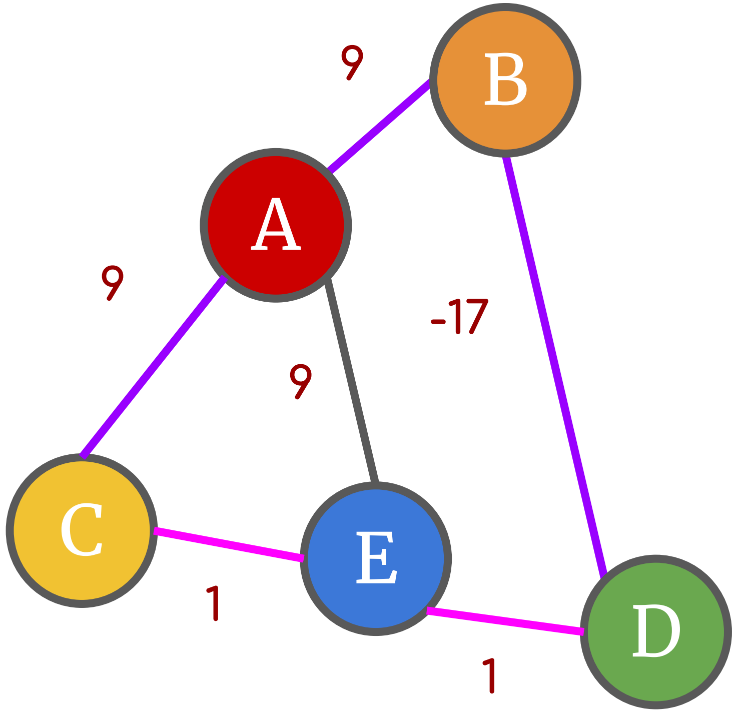
But there are some things it's not so good at.



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.

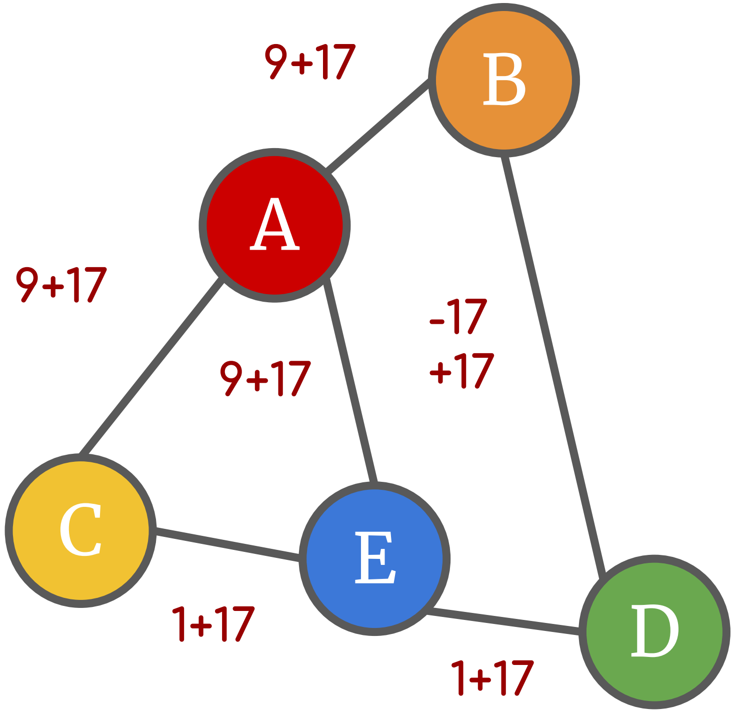
What's wrong with negative edge weights?



For one thing, we may stop too early, with an answer that is too big!

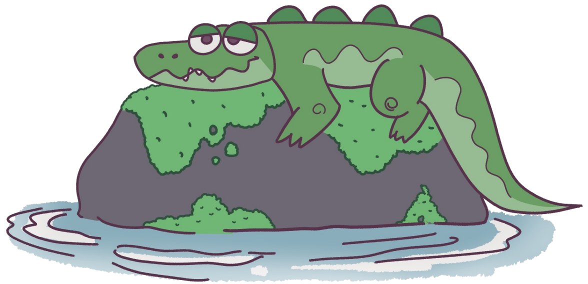
(Here we get 2, but the answer is 1)

What's wrong with negative edge weights?

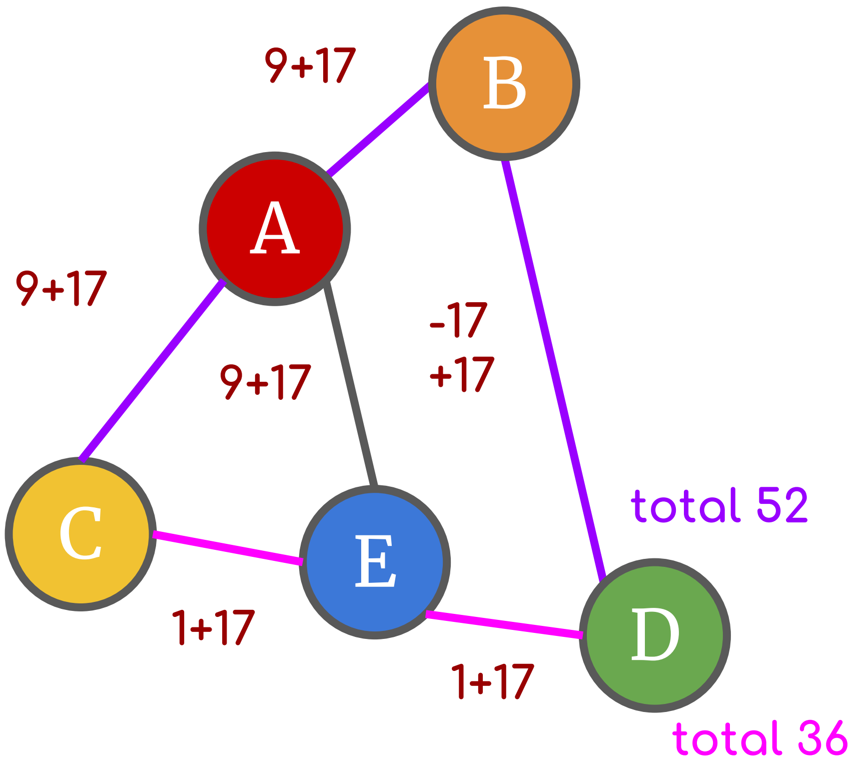


A possible fix:

- Add the most negative weight to every edge.
- Run the algorithm.
- Subtract off the added weights at the end.



What's wrong with negative edge weights?

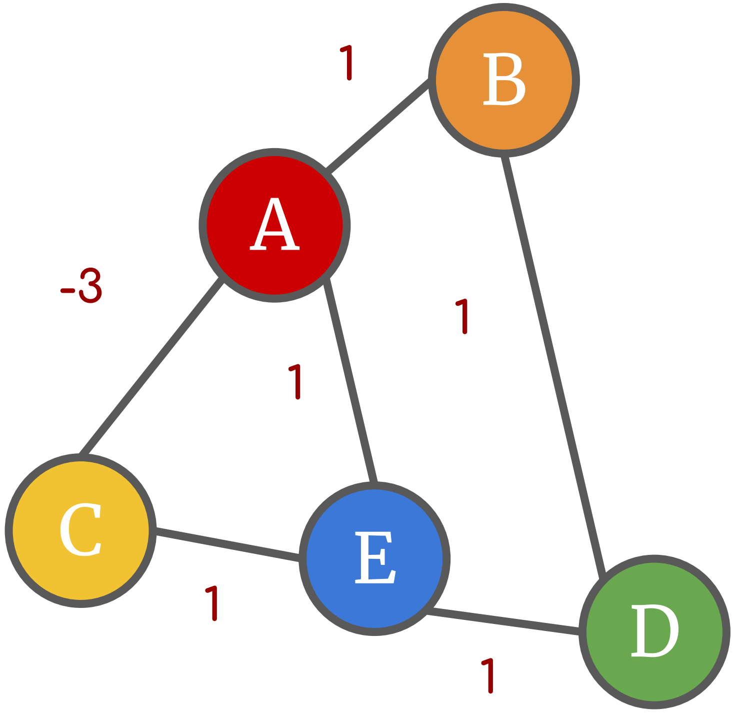


A possible fix:

- Add the most negative weight to every edge.
- Run the algorithm.
- Subtract off the added weights at the end.

Does this work? **No! We still end up picking the bad path because it has fewer steps.**

What's wrong with negative edge weights?



Even worse: negative cycles

Now the answer can be arbitrarily negative – we just go around CAEC... over and over! So the "shortest path" problem doesn't even make sense.

Next time:

- What if graphs can have directed edges?
- And what if we go deep rather than broad?