COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 14: CS Job Hunting (Interview Training)

Lecturer: Chris Lucas (cflucas@ncat.edu)

Midterm by EoW!

HW4 due tonight!

10/13 @ 11:59PM ET

HW5 released by EoD!

Zoom Office Hours!

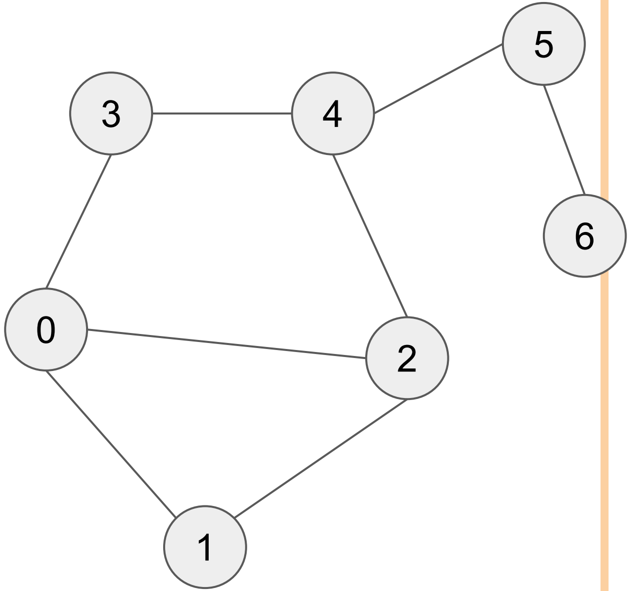
Google STEP program! (link)

Recall where we ended last lecture…

Graphs

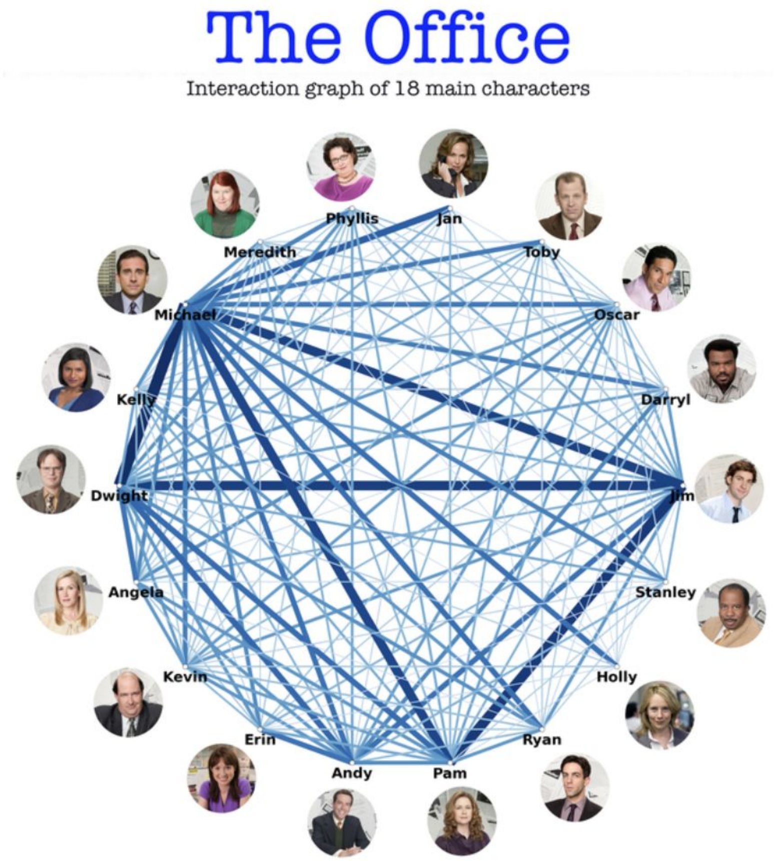
G = (V, E) is an undirected graph where
V = {0, 1, 2, 3, 4, 5, 6}
E = {(0,1),(0,2),(0,3),(1,2),(2,4),(3,4),(4,5),(5,6)}

- Graphs are composed of **vertices** (AKA nodes) and **edges**.
- Each vertex is connected to other vertices with edges. Each vertex has a **degree**.
- The edges can be **directed** or **undirected**.
- Graphs may contain cycles or otherwise be acyclic.
- When two vertices are connected, we can say they are **neighbors** and that they are **adjacent** to one another.
- The notation G = (V, E) stands for “Graph G has a set of vertices V = {v_1, v_2, …} and set of edges E = {e_1, e_2, ... }”

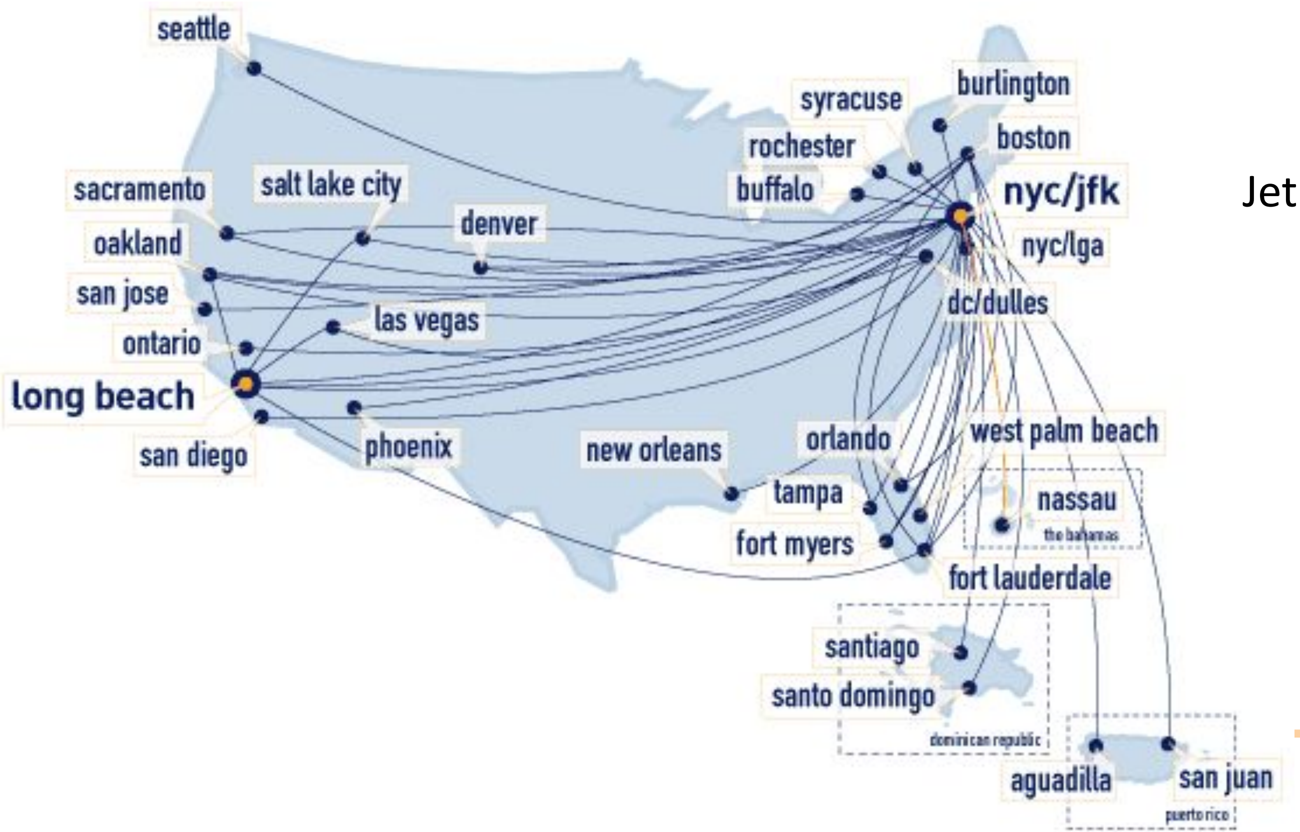


Graphs

The Office characters interaction network
why the different thicknesses?



Graphs



JetBlue flight route

Graph Representations

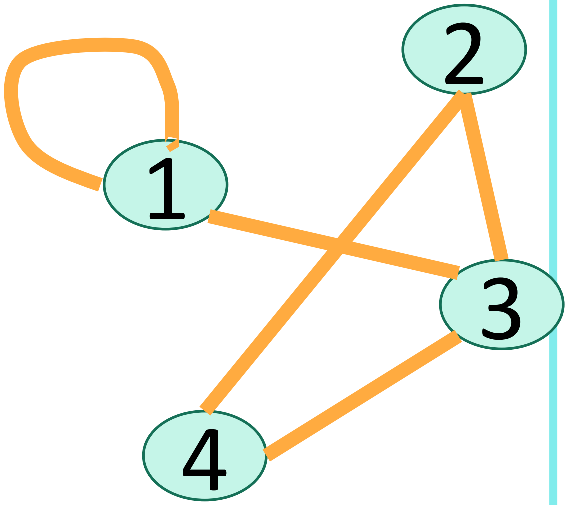
The two most common representations of Graphs:

- **Adjacency Matrix:** a 2D array (table) where the indices (rows and columns) represent vertices, and each entry at (i, j) will have a 1 if an edge is present (and have a 0 if no edge is present).
- **Adjacency List:** A list of lists, where each list at index i represents all neighbors for a vertex i.

How do we represent graphs?

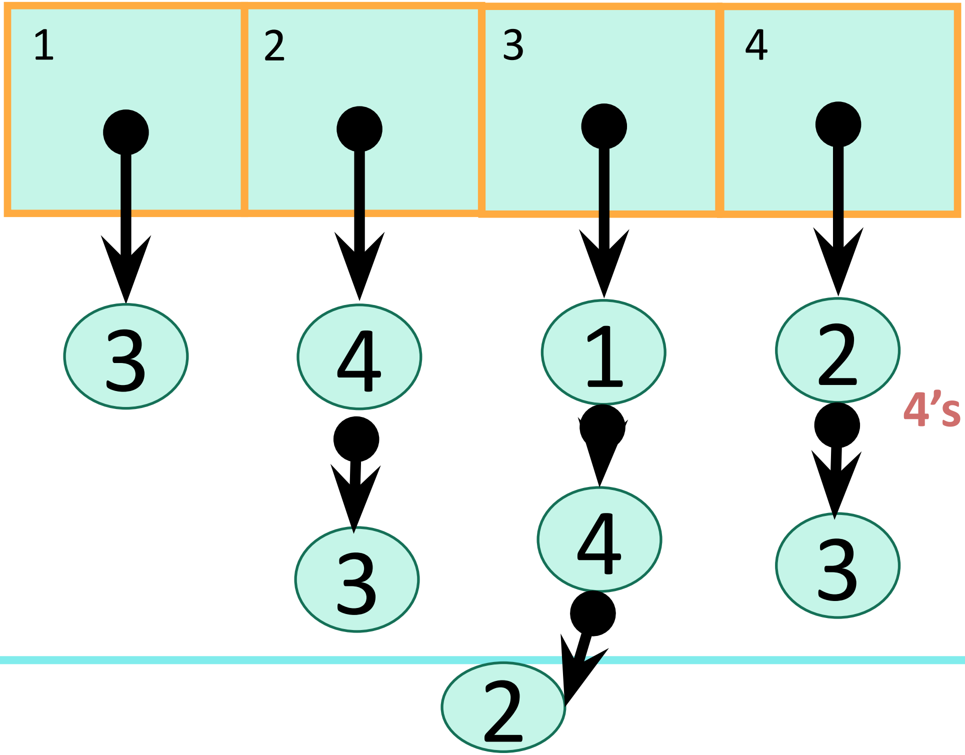
- Option 1: adjacency matrix

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 1 & 1 & 0 \end{array}$$



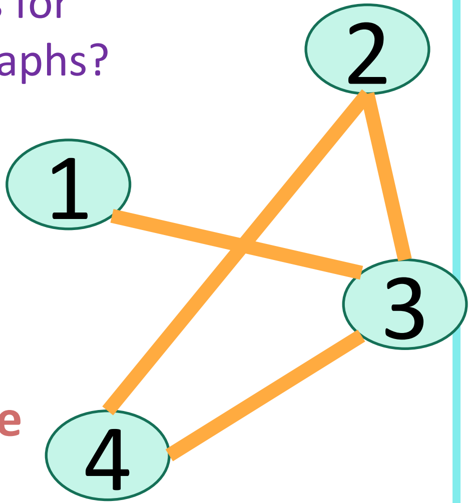
How do we represent graphs?

- Option 2: Adjacency lists

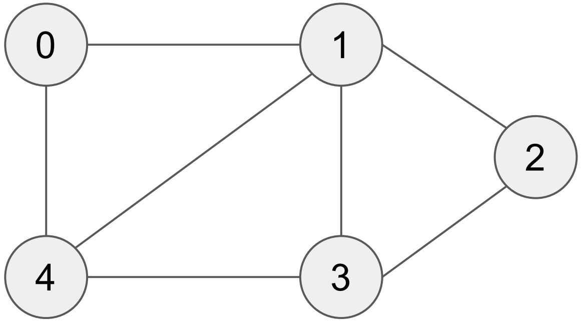


How would you modify this for directed graphs?

4's neighbors are 2 and 3



Graph Representation Examples



Graph

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

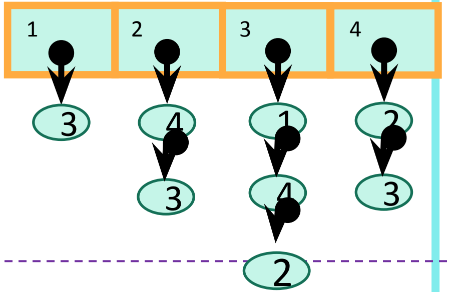
Adjacency Matrix

0: [1, 4]
1: [0, 4, 2, 3]
2: [1, 3]
3: [1, 4, 2]
4: [3, 0, 1]

Adjacency List

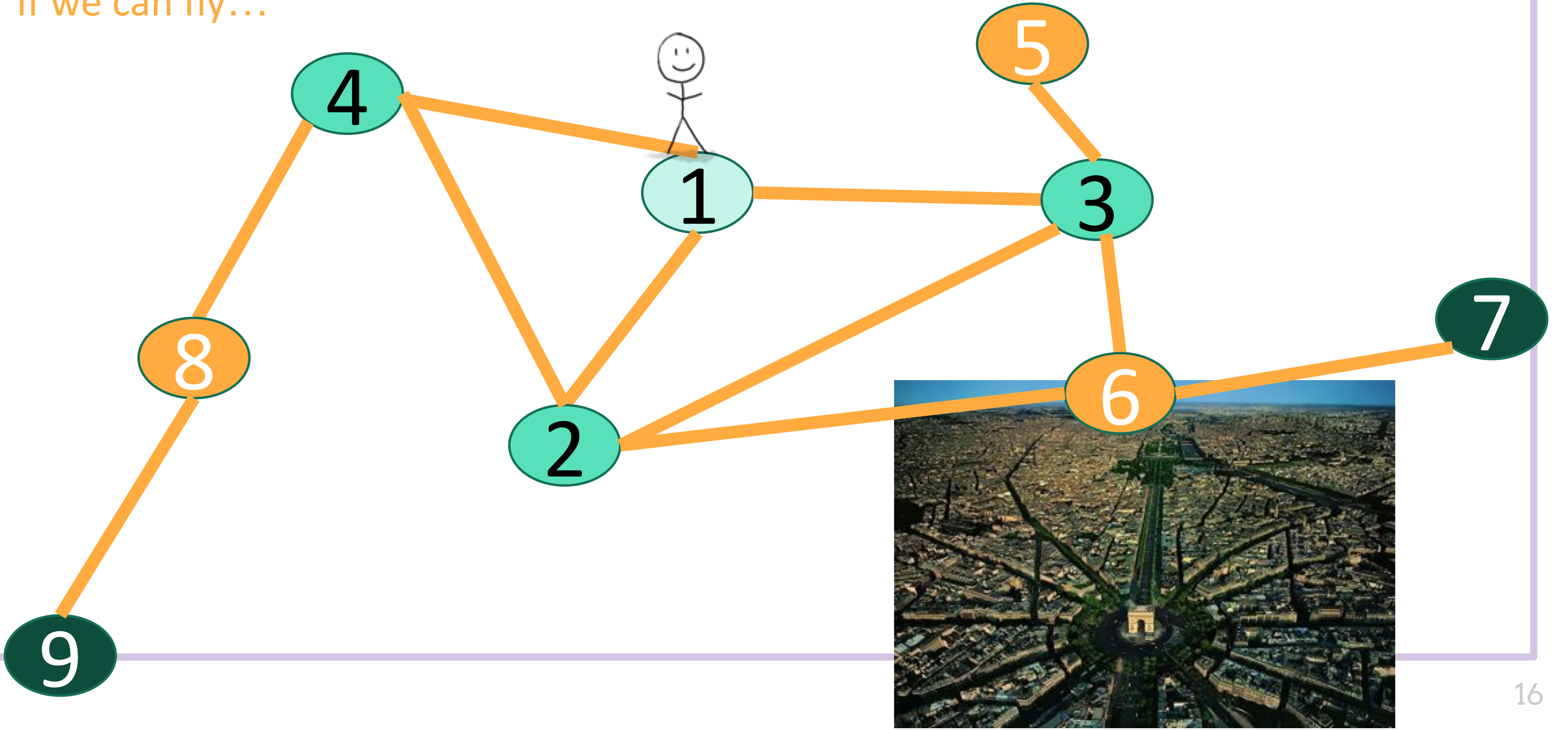
Generally better for **sparse** graphs (where $m \ll n^2$)

Trade-offs

Say there are V vertices and E edges.	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$	
Edge membership Is e = {v,w} in E?	O(1)	O(deg(V)) or O(deg(W))
Neighbor query Find all of v's neighbors.	O(V)	O(1)
Space requirements	$O(V^2)$	O(V + E)

How do we explore a graph?

If we can fly…



Big Questions!

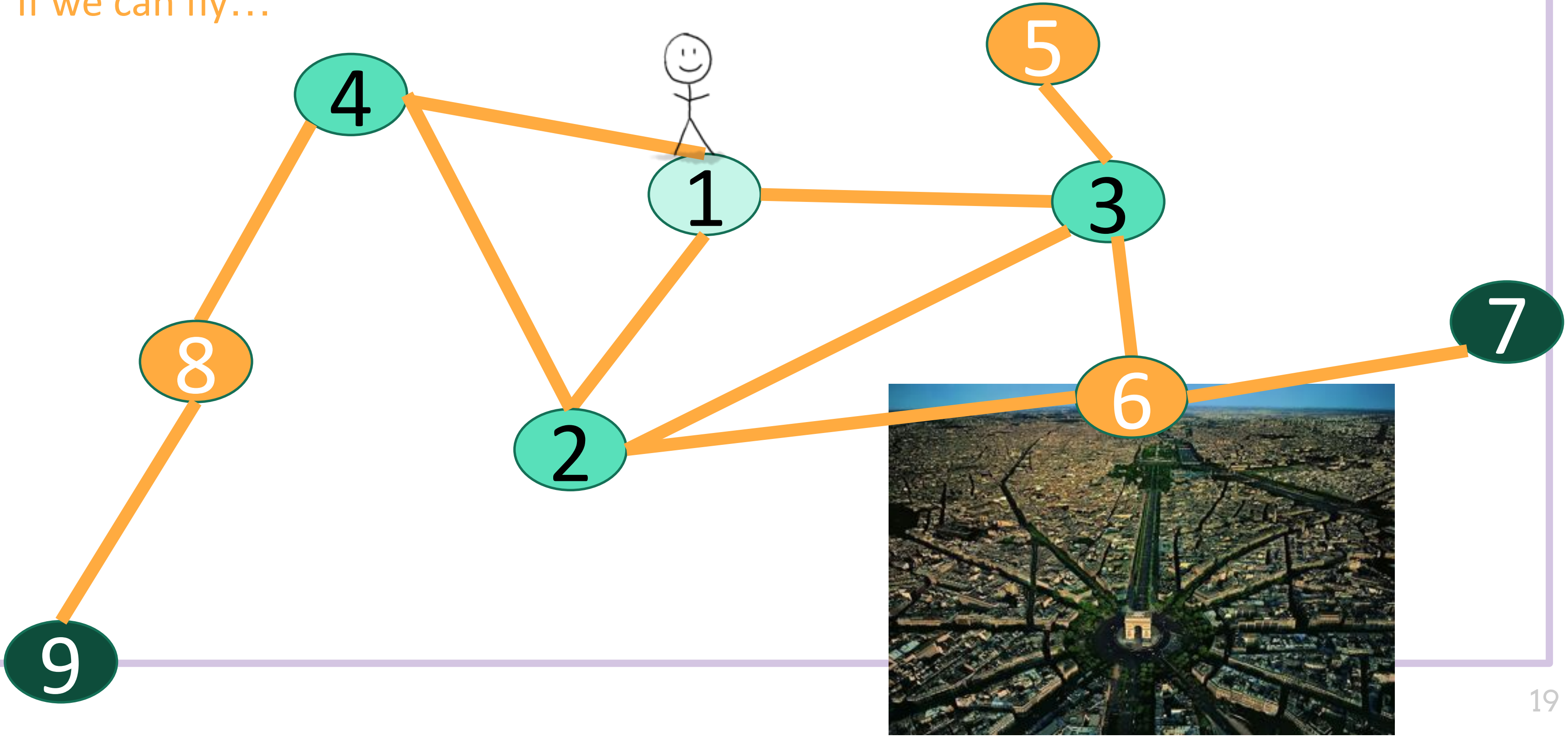
- How to find a viable path in a graph?
- How to have a successful software technical interview?
- How to have a successful hardware technical interview?

Big Questions!

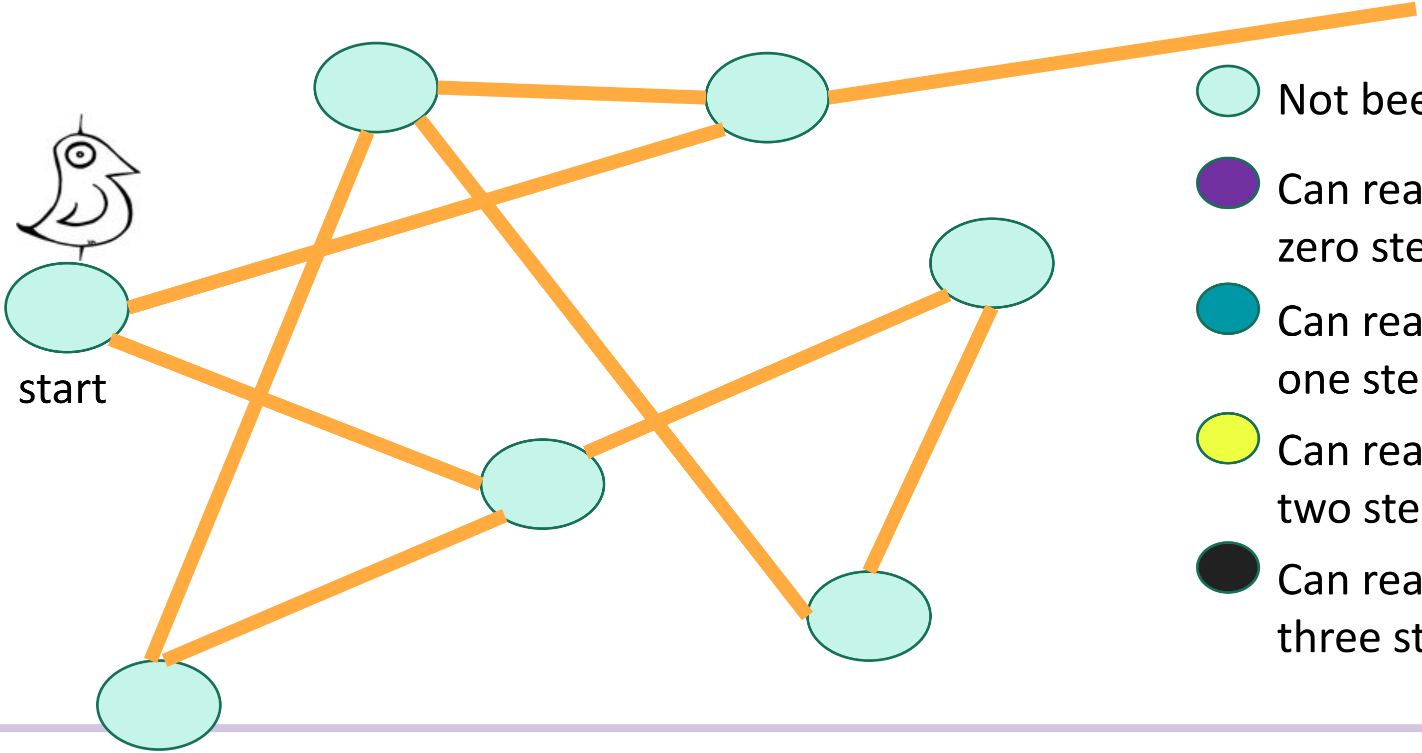
- How to find a viable path in a graph?
- How to have a successful software technical interview?
- How to have a successful hardware technical interview?

How do we explore a graph?

If we can fly…



Breadth-First Search: Exploring the world with a bird’s-eye view



○ Not been there yet

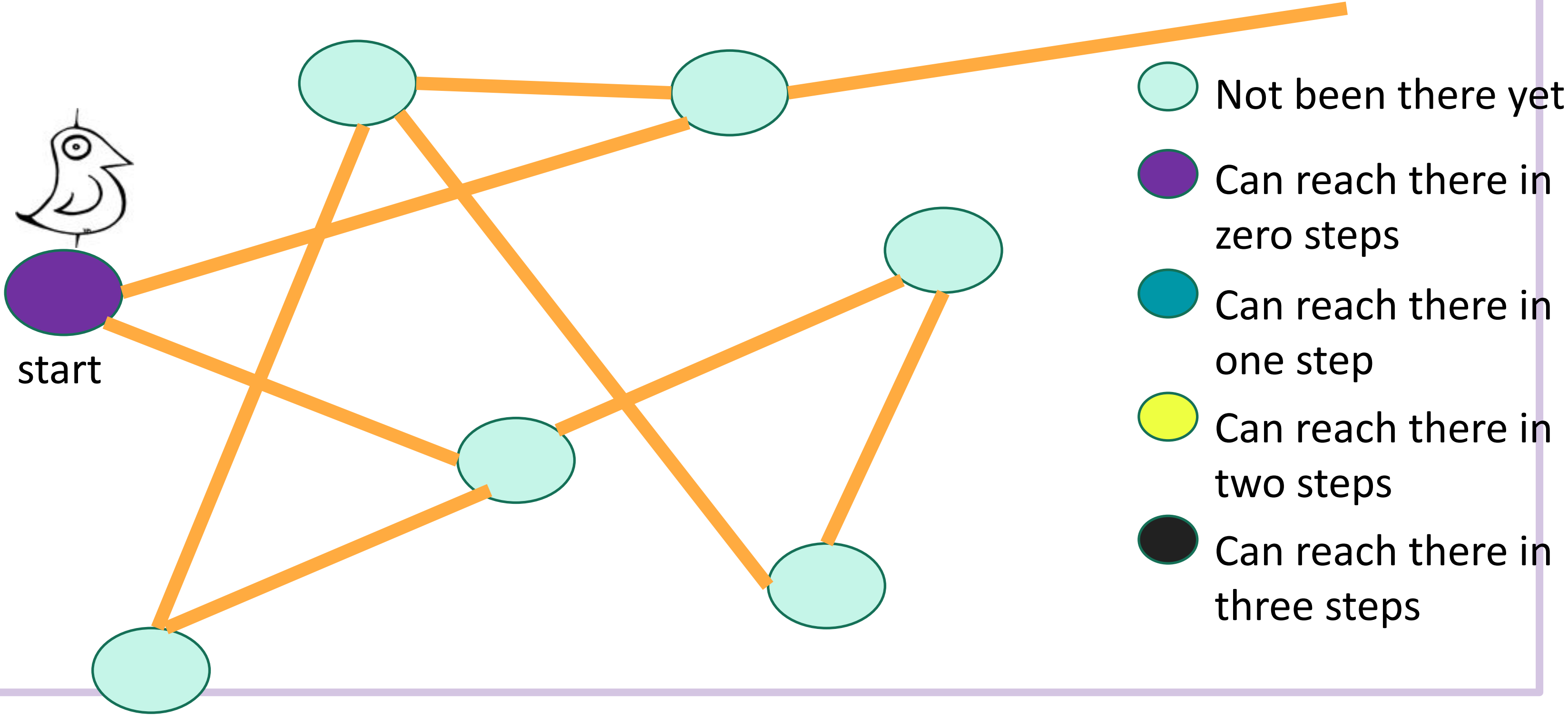
🟣 Can reach there in zero steps

🔵 Can reach there in one step

🟡 Can reach there in two steps

⚫ Can reach there in three steps

Breadth-First Search: Exploring the world with a bird's-eye view



start

○ Not been there yet

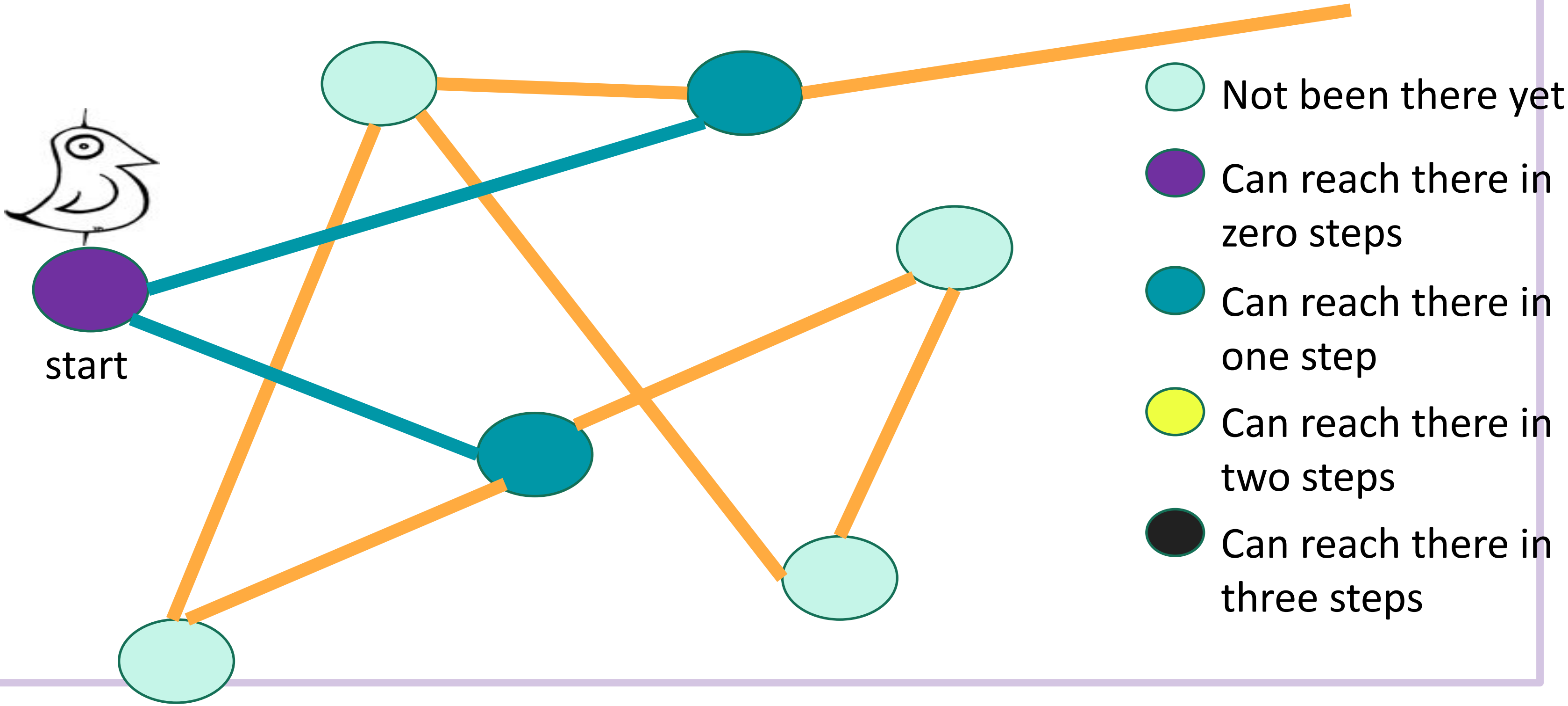
🟣 Can reach there in zero steps

🔵 Can reach there in one step

🟡 Can reach there in two steps

⚫ Can reach there in three steps

Breadth-First Search: Exploring the world with a bird's-eye view



○ Not been there yet

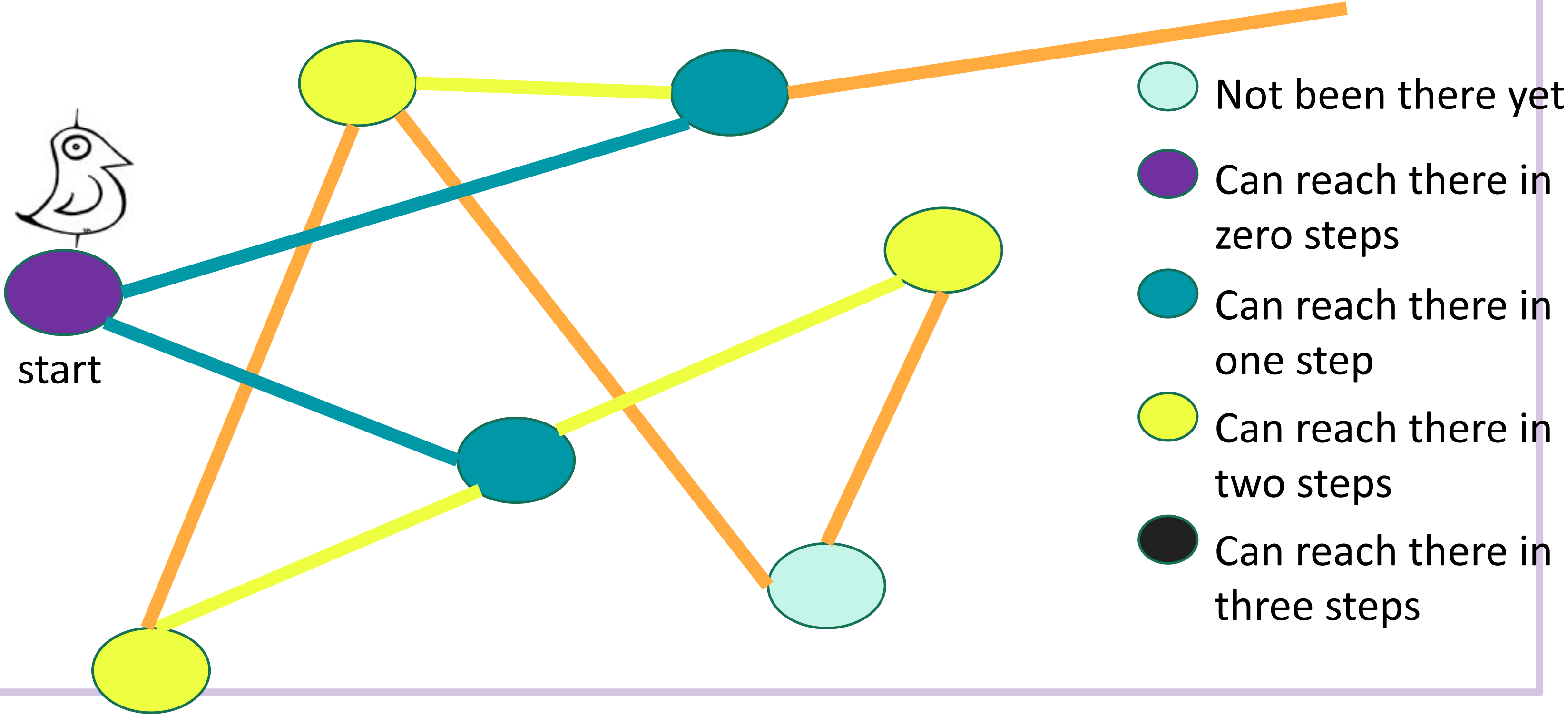
● Can reach there in zero steps

● Can reach there in one step

● Can reach there in two steps

● Can reach there in three steps

Breadth-First Search: Exploring the world with a bird’s-eye view



start

Not been there yet

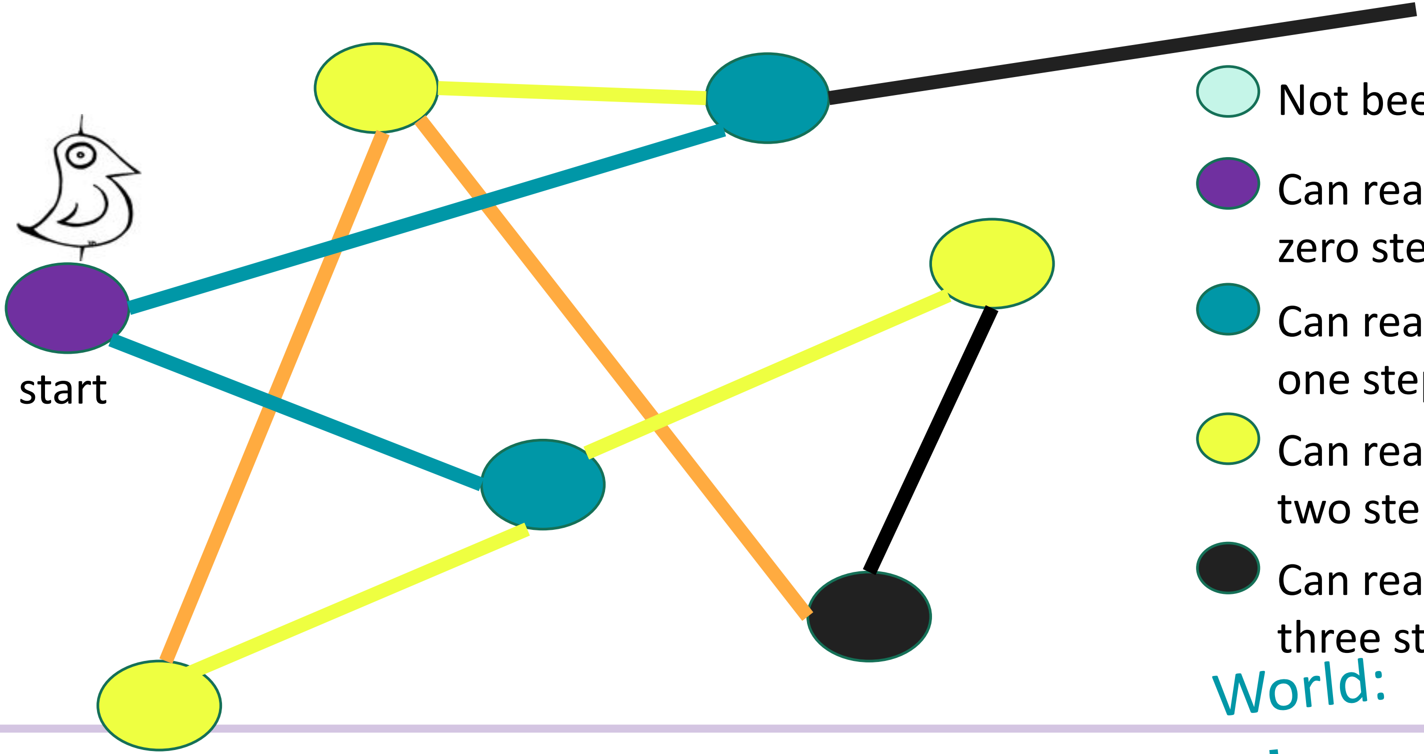
Can reach there in zero steps

Can reach there in one step

Can reach there in two steps

Can reach there in three steps

Breadth-First Search: Exploring the world with a bird's-eye view



○ Not been there yet

🟣 Can reach there in zero steps

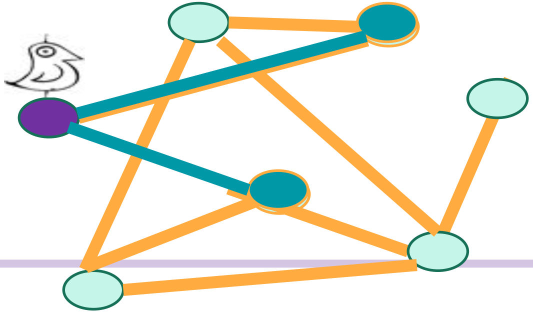
🔵 Can reach there in one step

🟡 Can reach there in two steps

⚫ Can reach there in three steps

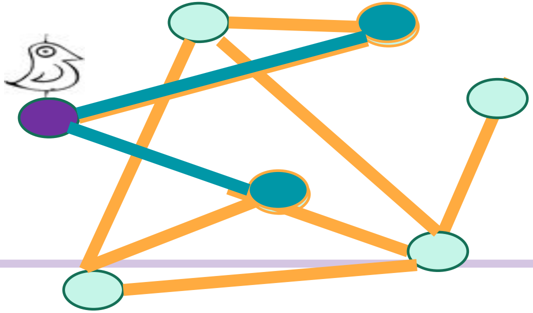
World: explored!

Breadth-First Search Pseudocode



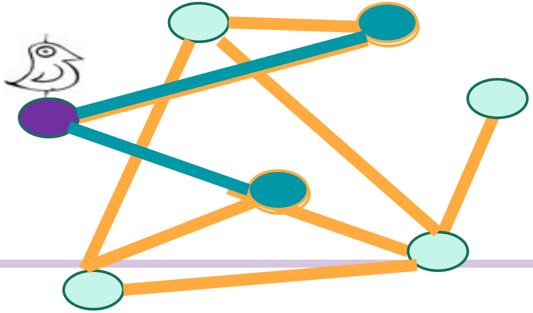
Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise
```



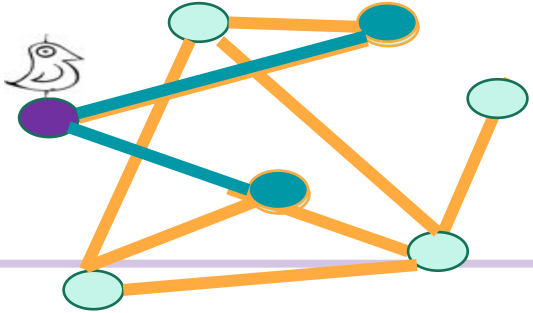
Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise

    frontier = Queue of integers
    visited = {} // empty hash set
```



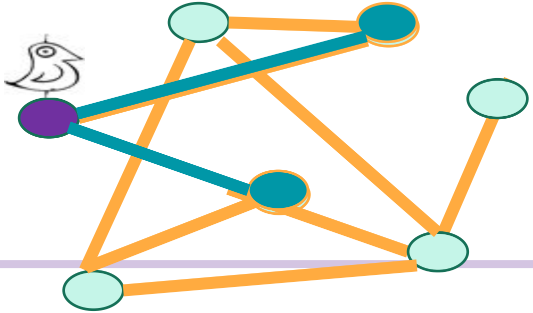
Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
```



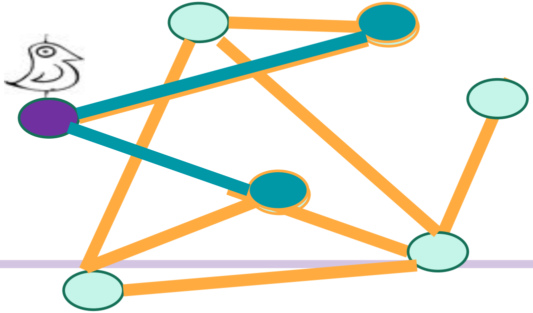
Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
```



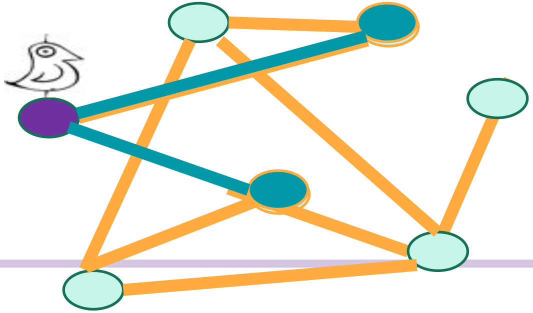
Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise

    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
```



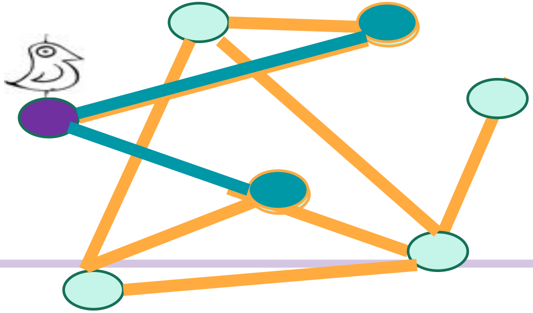
Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise

    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
        // ???
          visited.insert(neighbor)
          frontier.add(neighbor)
```



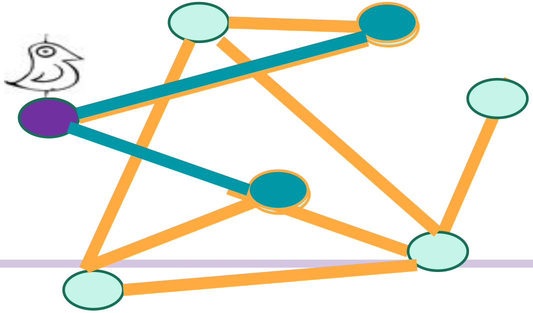
Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise

    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)
```



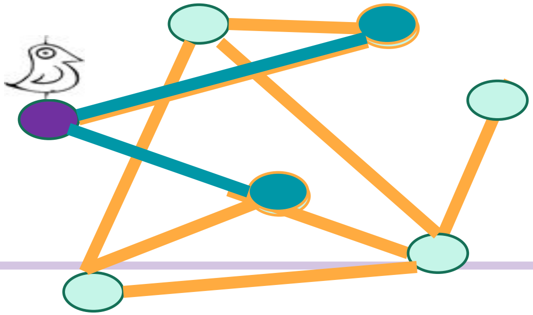
Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise

    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
```



Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), int s and int d
    Output: true if there's a path from s to d; false otherwise

    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
```

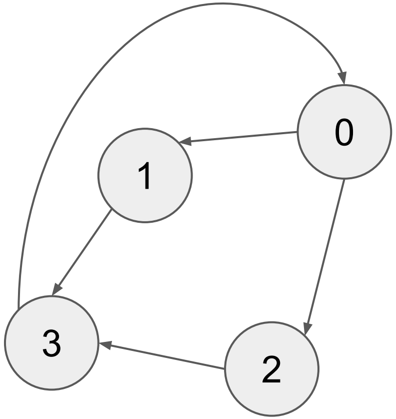


Running time and extension to directed graphs

- To explore the whole graph, explore the connected components one-by-one.
  - BFS running time is O(V + E)
- BFS also works fine on directed graphs.

Graph BFS

- Breadth-first search works on both Graphs and Trees. We'll be using a queue to store our pending nodes.
- Note that for graphs:
    - We may not reach every node.
    - We must keep track of **visited** nodes so you don't go in circles.
- Can be used to **find every node reachable from a source**, or **find the shortest paths to other nodes**.
- Demo: https://visualgo.net/en/dfsbfs



Big Questions!

- ○ How to find a viable path in a graph?
- ○ How to have a successful software technical interview?
- ○ How to have a successful hardware technical interview?

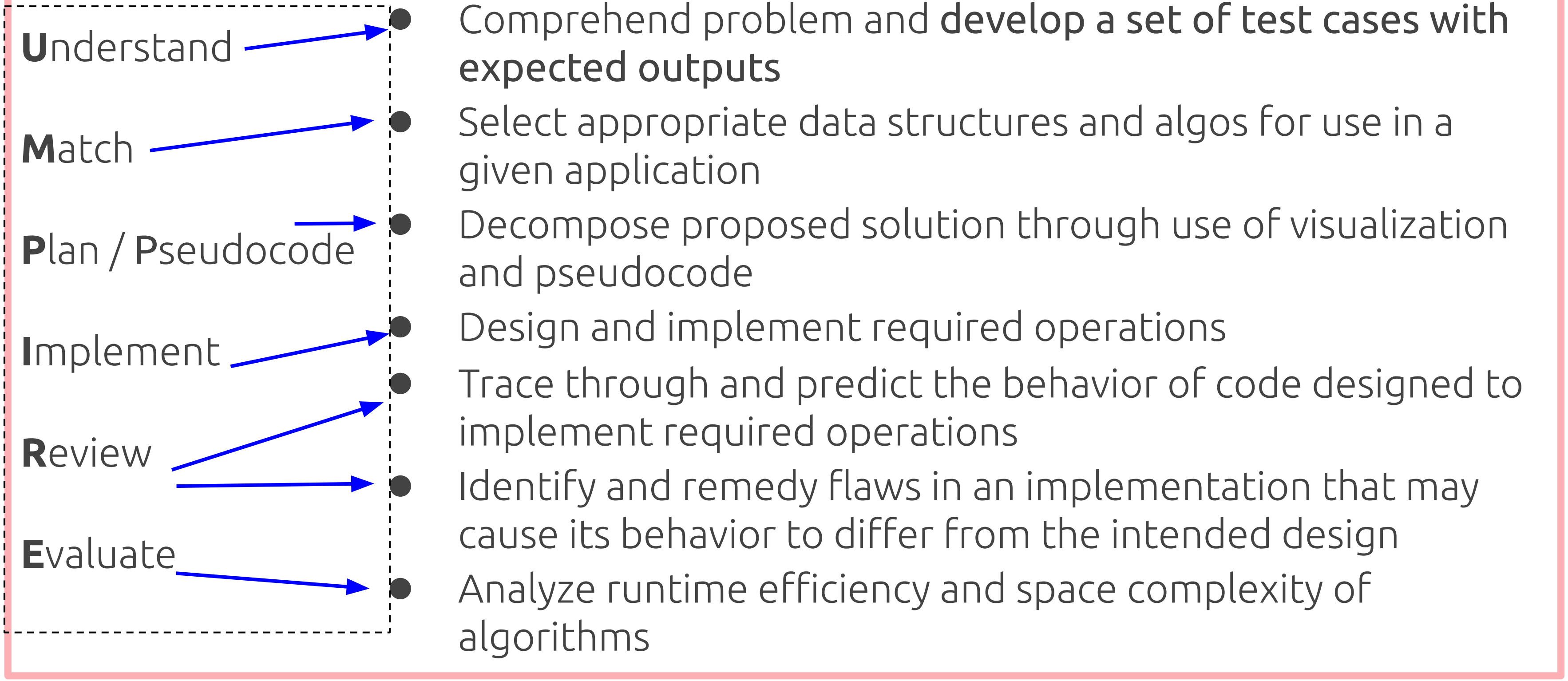
Motivation

- Knowing the sub-skills of “algorithmic problem-solving” allows us to practice metacognition, a fancy word for thinking about thinking.
    - Breaking down the process of “How do we think through solving problems with algorithms?”
    - This awareness gives us a way to more accurately evaluate our own strengths and weaknesses, and improve as needed.
- Knowing the sub-skills that exist can also help nudge the relevant muscle if we ever find ourselves hitting a wall / drawing a blank on a challenging problem.
- What are the sub-steps / sub-skills of algorithmic problem-solving?

UMPIRE METHOD

- Comprehend problem and **develop a set of test cases with expected outputs**
- Select appropriate data structures and algos for use in a given application
- Decompose proposed solution through use of visualization and pseudocode
- Design and implement required operations
- Trace through and predict the behavior of code designed to implement required operations
- Identify and remedy flaws in an implementation that may cause its behavior to differ from the intended design
- Analyze runtime efficiency and space complexity of algorithms

UMPIRE METHOD



Understand

Match

Plan / Pseudocode

Implement

Review

Evaluate

- Comprehend problem and **develop a set of test cases with expected outputs**
- Select appropriate data structures and algos for use in a given application
- Decompose proposed solution through use of visualization and pseudocode
- Design and implement required operations
- Trace through and predict the behavior of code designed to implement required operations
- Identify and remedy flaws in an implementation that may cause its behavior to differ from the intended design
- Analyze runtime efficiency and space complexity of algorithms

*UMPIRE pneumonic created by CodePath (www.codepath.org)

Example: return whether or not an array contains duplicates

Example: return whether or not an array contains duplicates

1. **Understand**: So we want to find whether or not there are duplicates in the array. That means if I pass in {4, 1, 3, 89} I'll return false, but if there were an extra 3, for example, I'd return true.
2. **Match**: I think I might be able to use a hashset here to do this efficiently.
3. **Plan**: I'll insert each element into a hashset, but before doing so, check if I've already seen it before. If I have, return true. If we get to the end, we can return False.

Example: return whether or not an array contains duplicates

1. **Understand**: So we want to find whether or not there are duplicates in the array. That means if I pass in {4, 1, 3, 89} I'll return false, but if there were an extra 3, for example, I'd return true.
2. **Match**: I think I might be able to use a hashset here to do this efficiently.
3. **Plan**: I'll insert each element into a hashset, but before doing so, check if I've already seen it before. If I have, return true. If we get to the end, we can return False.

4. Implement

```
#include <iostream>
#include <unordered_set>
#include <vector>

bool containsDuplicates(const std::vector<int>& arr)
{
 std::unordered_set<int> seenBefore;
 for (int i = 0; i < arr.size(); i++) {
   if (seenBefore.contains(arr[i])) {
     return true;
   } else {
     seenBefore.insert(arr[i]);
   }
 }
 return false;
}
```

5. **Review:** Line-by-line, everything looks good
6. **Evaluate:** We are iterating through our list of size n and doing O(1) work in each iteration,so our runtime is O(n)

Poll: Which skill in UMPIRE is being used?

"It sounds like we need to find if our input number is prime or not. That means if I input 7, we want to return true. But if we input 10, we would return false."

1. Understand
2. Match
3. Evaluate

Poll: Which skill in UMPIRE is being used?

“It sounds like we need to find if our input number is prime or not. That means if I input 7, we want to return true. But if we input 10, we would return false.”

1. [✓ Understand]
2. Match
3. Evaluate

Poll: Which skill in UMPIRE is being used?

“This algorithm should take $O(n^2)$ time, because for each element in the vector I am checking every element to the right of it.”

1. Understand
2. Match
3. Evaluate

Poll: Which skill in UMPIRE is used?

“This algorithm should take O(n^2) time, because for each element in the vector I am checking every element to the right of it.”

1. Understand
2. Match
3. [✓ Evaluate]

Poll: Which skill in UMPIRE is being used?

“I think a stack might be useful here”

1. Understand
2. Match
3. Evaluate

Poll: Which skill in UMPIRE is being used?

“I think a stack might be useful here”

1. Understand
2. [✓ Match]
3. Evaluate

Poll: Which skill in UMPIRE is being used?

“This sorting question is interesting because the input is mostly sorted. Maybe insertion sort makes sense.”

1. Understand
2. Match
3. Evaluate

Poll: Which skill in UMPIRE is being used?

“This sorting question is interesting because the input is mostly sorted. Maybe insertion sort makes sense.”

1. Understand
2. [✓ Match]
3. Evaluate

General Tips

- **Communicate. Communicate. Communicate.**
  - A perfect solution without communication would not pass.
  - You can communicate that you need a few minutes to think in silence.
  - If you get stuck or are going down the wrong track, your interviewer can identify that upfront rather than when you're 15 minutes into a solution.
- **Listen to your interviewer**
  - If they give you a hint or steer you in a different direction, listen to them and explore where they're taking you.
- **Remain positive**
  - Questions are meant to be challenging; they're pushing the boundaries of your problem-solving skills. (Breezing through an interview won't give them signal either).
  - Needing hints does not automatically disqualify you; it's all relative!
  - Good for your own well-being

General Tips (pt. 2)

- **Time is the Most Precious Resource**
  - 1-2 questions about your resume aren't atypical… focus on CONCISION (so you can maximize time spent on problem).
  - Don't utilize bathroom/water/snack breaks at beginning of interview, save for designated breaks in schedule.
- **Quickly Give a Solution (if you can)**
  - Duh!
  - Kidding lol, if you can only think of a brute force solution - that's still good! You can "get points" for verbalizing the solution, acknowledging the runtime is inefficient and start a conversation about how to optimize (ie. duplicated work)
- **Use Every Part of the Problem**
  - Especially if you're stumped or at suboptimal/inefficient solution, there's a reason for **every** part of the problem.
  - Even if you can't piece it together, acknowledge it: "I notice that I'm not using the fact that the array is sorted", "I know my algorithm ignores that the input won't contain duplicates…"

General Tips (pt. 3)

- **Trust your Instincts**
  - How are you able to solve the problem by hand/mentally?
  - Break down the steps you are performing in your head - usually a great starting point!
- **Code style matters!**
  - Write clean, concise, well-formatted code… (e.g. good variable names, helper methods, etc.)
  - Often times your code is photographed and sent to hiring committees
- **ALWAYS Prepare Questions**
  - Show enthusiasm for the role, no generic questions (ie. What are languages most used? How many hours do you work per day? What are the benefits? What’s the best/worst thing about your job?)
- **Notebook (Optional)**
  - It catches people off guard – in a good way!
  - Follow up interview email is unique, specific, personalized, differentiates you/demonstrates passion. Mention interviewers by name and include anecdotes from conversations.
  - Can be strategically used as a buffer while you think of an answer!
  - Follow your interviewer’s guidance if they object, (hasn’t happened to me though)

General Tips (pt. 3)

Understand

Match

Plan / Pseudocode

Implement

Review

Evaluate

- Communicate. Communicate. Communicate.
- Listen to your interviewer
- Remain positive
- Time is the Most Precious Resource
- Quickly Give a Solution (if you can)
- Use Every Part of the Problem
- Trust your Instincts
- Code style matters!
- ALWAYS Prepare Questions
- Notebook (Optional)

General Tips (pt. 3)

Understand

aate

- e.
- e
- can)
- he Problem
- your instincts.
- Code style matters!
- ALWAYS Prepare Questions
- Notebook (Optional)

How can I practice?

Simulating Interview Environment

- Talk to yourself and walk through each of the letters in UMPIRE.
- Write code by hand on whiteboard (or paper if you don't have a whiteboard)
- Record yourself explaining your approach as you go. Listen back, are you able to follow your own explanations?
- **Repetition is the largest contributor to success**. It's a sport that requires practice, a muscle that requires exercise.
  - We'll be doing something like this for HW5
- Practice Problems:
  - Cracking the Coding Interview here!
  - Leetcode
- Sign up with me here!

Big Questions!

- How to find a viable path in a graph?
- How to have a successful software technical interview?
- How to have a successful hardware technical interview?

What about Hardware?

- **More behavioral and knowledge-based questions**
  - Behavioral: Tell me about a time when … STAR method!
  - Knowledge-based: Specific questions about hardware engineering concepts (circuitry, memory, cache, capacitance, inductors, resistors, etc.)
  - More directly related to what you would study for an exam
- **Occasional software/coding questions**
  - So still should practice!
- **Resources:**
  - Prepfully has round-by-round breakdowns of interviews (here)
  - Recent interview questions asked at your company (here)
  - Laundry list of interview questions (here)

COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 14: CS Job Hunting (Interview Training)

Lecturer: Chris Lucas (cflucas@ncat.edu)