Interview Cake

You wrote a trendy new messaging app, MeshMessage, to get around flaky cell phone coverage.

Instead of routing texts through cell towers, your app sends messages via the phones of nearby users, passing each message along from one phone to the next until it reaches the intended recipient. (Don't worry—the messages are encrypted while they're in transit.)

Some friends have been using your service, and they're complaining that it takes a long time for messages to get delivered. After some preliminary debugging, you suspect messages might not be taking the most direct route from the sender to the recipient.

Given information about active users on the network, find the shortest route for a message from one user (the sender) to another (the recipient). Return a list of users that make up this route.

> There might be a *few* shortest delivery routes, all with the same length. For now, let's just return *any* shortest route.

Your network information takes the form of a dictionary mapping username strings to a list of other users nearby:

Python 3.6 ▾

```
network = {
    'Min'     : ['William', 'Jayden', 'Omar'],
    'William' : ['Min', 'Noam'],
    'Jayden'  : ['Min', 'Amelia', 'Ren', 'Noam'],
    'Ren'     : ['Jayden', 'Omar'],
    'Amelia'  : ['Jayden', 'Adam', 'Miguel'],
    'Adam'    : ['Amelia', 'Miguel', 'Sofia', 'Lucas'],
    'Miguel'  : ['Amelia', 'Adam', 'Liam', 'Nathan'],
    'Noam'    : ['Nathan', 'Jayden', 'William'],
    'Omar'    : ['Ren', 'Min', 'Scott'],
    ...
}
```

For the network above, a message from Jayden to Adam should have this route:

Python 3.6 ▾

```
['Jayden', 'Amelia', 'Adam']
```

Gotchas

We can find the shortest route in $O(N + M)$ time, where N is the number of users and M is the number of connections between them.

It's easy to write code that can get caught in an infinite loop for some inputs! Does your code always finish running?

What happens if there's no way for messages to get to the recipient?

What happens if the sender tries to send a message to themselves?

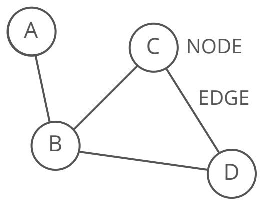
Breakdown

Users? Connections? Routes? What data structures can we build out of that? Let's run through some common ones and see if anything fits here.

- Lists? Nope—those are a bit too simple to express our network of users.
- Dictionaries? Maybeee.

- Graphs?⤵

A **graph** is an abstract data structure with **nodes** (or **vertices**) that are connected by **edges**.



They're useful in cases where you have *things that connect to other things*. Nodes and edges could, for example, respectively represent cities and highways, routers and ethernet cables, or Facebook users and their friendships.

Yeah, that seems like it could work!

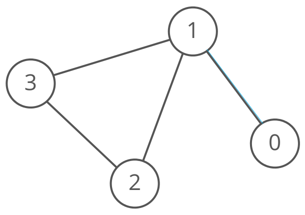
Let's run with graphs for a bit and see how things go. Users will be nodes in our graph, and we'll draw edges between users who are close enough to message each other.

Our input dictionary already represents the graph we want in adjacency list⤵

One common graph storage format is called an **adjacency list**.

In this format, every node has a list of connected neighbors: an adjacency list. We could store these lists in a dictionary where the keys represent the node and the values are the adjacency lists.

For instance, here's one graph:



And, here's how we'd represent it using a dictionary:

Python 3.6 ▾

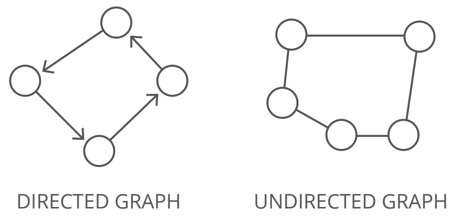
```
graph = {
    0: [1],
    1: [0, 2, 3],
    2: [1, 3],
    3: [1, 2],
}
```

Since node `3` has edges to nodes `1` and `2`, `graph[3]` has the adjacency list `[1, 2]`.

format. Each key in the dictionary is a node, and the associated value—a list of connected nodes—is an adjacency list.

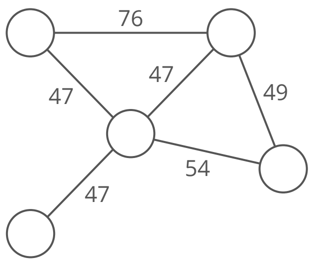
Is our graph directed or undirected?⤵

Graphs can be **directed** or **undirected**. In directed graphs, edges point from the node at one end to the node at the other end. In undirected graphs, the edges simply connect the nodes at each end.



Weighted or unweighted?⤵

If a graph is **weighted**, each edge has a "weight." The weights could, for example, represent the distance between two locations, or the cost or time it takes to travel between the locations.

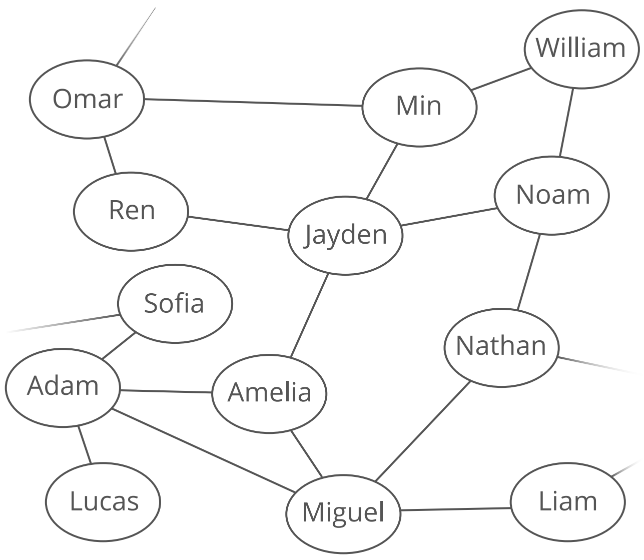


For directed vs. undirected, we'll assume that if Min can transmit a message to Jayden, then Jayden can also transmit a message to Min. Our sample input definitely suggests this is the case. And it makes sense—they're the same distance from each other, after all. That means our graph is **undirected**.

What about weighted? We're not given any information suggesting that some transmissions are more expensive than others, so let's say our graph is **unweighted**.

> These assumptions seem pretty reasonable, so we'll go with them here. But, this is a great place to step back and check in with your interviewer to make sure they agree with what you've decided so far.

Here's what our user network looks like as a graph:



Okay, how do we start looking around our graph to find the shortest route from one user to another?

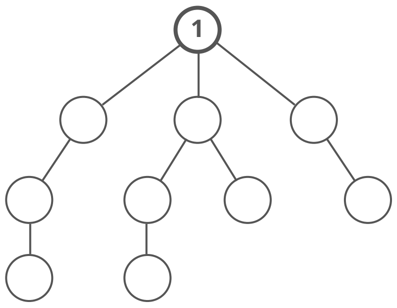
Or, more generally, **how do we find the shortest path from a start node to an end node in an unweighted, undirected graph?**

There are two common ways to explore undirected graphs: depth-first search (DFS)↓

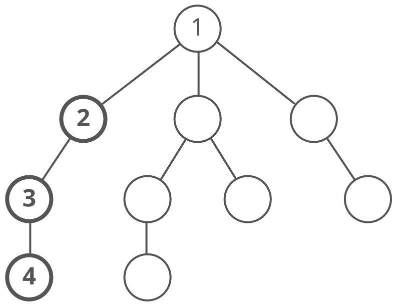
Depth-first search (DFS) is a method for exploring a tree or graph. In a DFS, you go as deep as possible down one path before backing up and trying a different one.

Depth-first search is like walking through a corn maze. You explore one path, hit a dead end, and go back and try a different one.

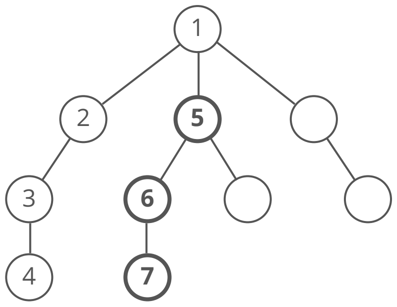
Here's a how a DFS would traverse this tree, starting with the root:



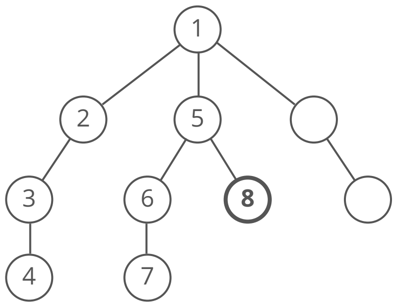
We'd go down the first path we find until we hit a dead end:



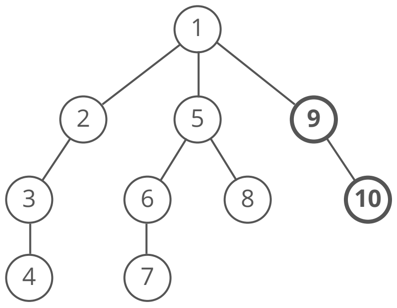
Then we'd do the same thing again—go down a path until we hit a dead end:



And again:



And again:



Until we reach the end.

Depth-first search is often compared with **breadth-first search**.

Advantages:

- Depth-first search on a binary tree *generally* requires less memory than breadth-first.
- Depth-first search can be easily implemented with recursion.

Disadvantages

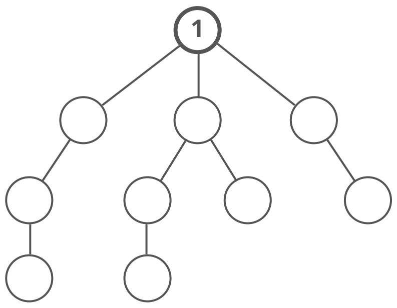
- A DFS doesn't necessarily find the shortest path to a node, while breadth-first search does.

and breadth-first search (BFS).⤵

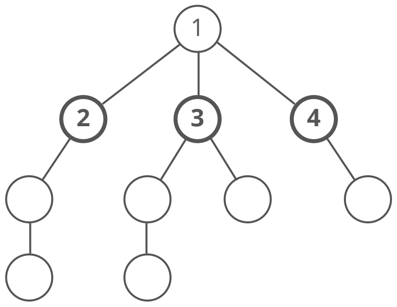
Breadth-first search (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.

Breadth-first search is like throwing a stone in the center of a pond. The nodes you explore "ripple out" from the starting point.

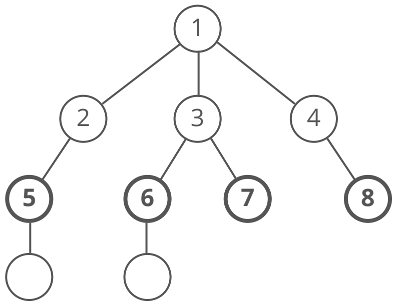
Here's a how a BFS would traverse this tree, starting with the root:



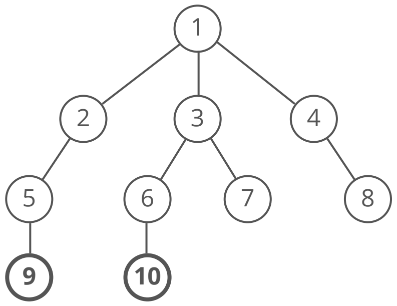
We'd visit all the immediate children (all the nodes that're one step away from our starting node):



Then we'd move on to all *those* nodes' children (all the nodes that're *two steps* away from our starting node):



And so on:



Until we reach the end.

Breadth-first search is often compared with **depth-first search**.

Advantages:

- A BFS will find the **shortest path** between the starting point and any other reachable node. A depth-first search will not necessarily find the shortest path.

Disadvantages

- A BFS on a binary tree *generally* requires more memory than a DFS.

Which do we want here?

Since we're interested in finding the *shortest* path, BFS is the way to go.

> Remember: both BFS and DFS will eventually find a path if one exists. The difference between the two is:
>
> - BFS *always* finds the shortest path.
> - DFS *usually* uses less space.

Okay, so let's do a breadth-first search of our graph starting from the sender and stopping when we find the recipient. Since we're using breadth-first search, we know that the first time we see the recipient, we'll have traveled to them along the shortest path.

To code this up, let's start with a standard implementation of breadth-first search:

> It's a good idea to know breadth-first and depth-first search well enough to quickly write them out. They show up in a *lot* of graph problems.

Python 3.6

```
from collections import deque


def bfs(graph, start_node, end_node):
    nodes_to_visit = deque()
    nodes_to_visit.append(start_node)

    # Keep track of what nodes we've already seen
    # so we don't process them twice
    nodes_already_seen = set([start_node])

    while len(nodes_to_visit) > 0:
        current_node = nodes_to_visit.popleft()

        # Stop when we reach the end node
        if current_node == end_node:
            # Found it!
            break

        for neighbor in graph[current_node]:
            if neighbor not in nodes_already_seen:
                nodes_already_seen.add(neighbor)
                nodes_to_visit.append(neighbor)
```

> Look at the `nodes_already_seen` set—that's really important and easy to forget. If we didn't have it, our algorithm would be slower (since we'd be revisiting tons of nodes) *and* it might never finish (if there's no path to the end node).

> We're using a queue instead of a list because we want an efficient first-in-first-out (FIFO) structure with $O(1)$ inserts and removes. If we used a list, appending would be $O(1)$, but removing elements from the front would be $O(n)$.

This seems like we're on the right track: we're doing a breadth-first search, which gets us from the start node to the end node along the shortest path.

But we're still missing an important piece: we *didn't actually store our path anywhere*. We need to *reconstruct the path we took*. How do we do that?

Well, we know that the *first* node in the path is `start_node`. And the next node in the path is ... well ... hmm.

Maybe we can start from the end and work backward? We know that the *last* node in the path is `end_node`. And the node before that is ... hmm.

We don't have enough information to actually reconstruct the path.

What additional information can we store to help us?

Well, to reconstruct our path, we'll need to *somehow* recover how we found each node. When do we find new nodes?

We find new nodes when iterating through `current_node`'s neighbors.

So, each time we find a new node, let's jot down what `current_node` was when we found it. Like this:

Python 3.6 ▾

```
from collections import deque


def bfs_get_path(graph, start_node, end_node):
    nodes_to_visit = deque()
    nodes_to_visit.append(start_node)

    # Keep track of what nodes we've already seen
    # so we don't process them twice
    nodes_already_seen = set([start_node])

    # Keep track of how we got to each node
    # we'll use this to reconstruct the shortest path at the end
    how_we_reached_nodes = {start_node: None}

    while len(nodes_to_visit) > 0:
        current_node = nodes_to_visit.popleft()

        # Stop when we reach the end node
        if current_node == end_node:
            # Somehow reconstruct the path here
            return path

        for neighbor in graph[current_node]:
            if neighbor not in nodes_already_seen:
                nodes_already_seen.add(neighbor)
                nodes_to_visit.append(neighbor)
                # Keep track of how we got to this node
                how_we_reached_nodes[neighbor] = current_node
```

Great. Now we just have to take that bookkeeping and use it to reconstruct our path! How do we do that?

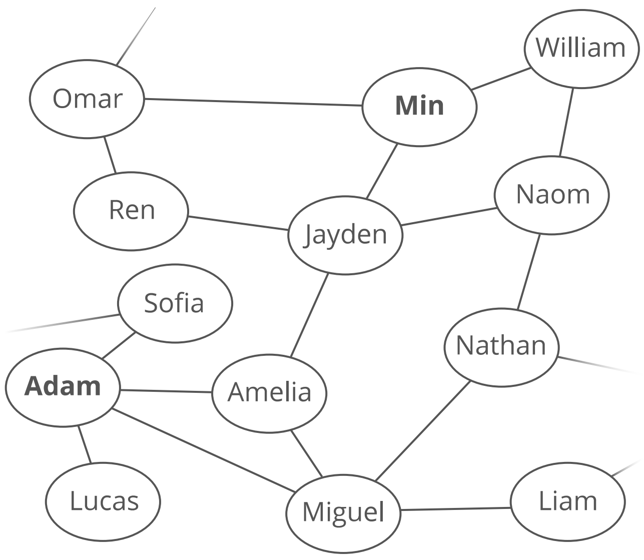
Let's start by looking up `start_node` in our dictionary. Oops, it's just `None`.

Oh. Right. Our dictionary tells us which node comes *before* a given node on the shortest path. And nothing comes before the start node.

What about the `end_node`? If we look up the end node in `how_we_reached_nodes`, we'll get the node we were visiting when we found `end_node`. That's the node that comes right *before* the end node along the shortest path!

So, we'll actually be building our path *backward* from `end_node` to `start_node`.

Going back to our example, to recover the shortest path from Min to Adam:



We'd take this dictionary we built up during our search:

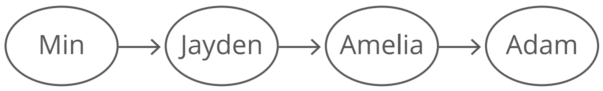
Python 3.6 ▾

```
{'Min'     : None,
 'Jayden'  : 'Min',
 'Ren'     : 'Jayden',
 'Amelia'  : 'Jayden',
 'Adam'    : 'Amelia',
 'Miguel'  : 'Amelia',
 'William' : 'Min'}
```

And, we'd use it to backtrack from the end node to the start node, recovering our path:

- To get to Adam, we went through Amelia.
- To get to Amelia, we went through Jayden.
- To get to Jayden, we went through Min.
- Min is the start node.

Chaining this together, the shortest path from Min to Adam is



Here's what this could look like in code:

Python 3.6 ▾

```
def reconstruct_path(how_we_reached_nodes, start_node, end_node):
    shortest_path = []

    # Start from the end of the path and work backwards
    current_node = end_node

    while current_node:
        shortest_path.append(current_node)
        current_node = how_we_reached_nodes[current_node]

    return shortest_path
```

One small thing though. Won't this return a path that has the recipient at the beginning?

Oh. Since we started our backtracking at the recipient's node, our path is going to come out *backward*. So, let's reverse it before returning it:

Python 3.6 ▾

```
shortest_path.reverse()  # now from start_node to end_node
return shortest_path  # return in the right order
```

Okay. That'll work!

But, before we're done, let's think about edge cases and optimizations.

What are our edge cases, and how should we handle them?

What happens if there *isn't* a route from the sender to the recipient?

If that happens, then we'll finish searching the graph without ever reconstructing and returning the path. That's a valid outcome—it just means we can't deliver the message right now. So, if we finish our search and haven't returned yet, let's return `None` to indicate that no path was found.

What about if either the sender or recipient aren't in our user network?

That's invalid input, so we should raise an exception.

Any other edge cases?

Those two should be it. So, let's talk about optimizations. Can we make our algorithm run faster or take less space?

One thing that stands out is that we have two data structures— `nodes_already_seen` and `how_we_reached_nodes`—that are updated in similar ways. In fact, every time we add a node to `nodes_already_seen`, we also add it to `how_we_reached_nodes`. Do we need both of them?

We definitely need `how_we_reached_nodes` in order to reconstruct our path. What about `nodes_already_seen`?

Every node that appears in `nodes_already_seen` *also* appears in our dictionary. So, instead of keeping a separate set tracking nodes we've already seen, we can just use the keys in `how_we_reached_nodes`. This lets us get rid of `nodes_already_seen`, which saves us $O(n)$ space.

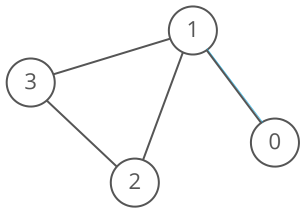
Solution

We treat the input user network as a graph in adjacency list↓

One common graph storage format is called an **adjacency list**.

In this format, every node has a list of connected neighbors: an adjacency list. We could store these lists in a dictionary where the keys represent the node and the values are the adjacency lists.

For instance, here's one graph:



And, here's how we'd represent it using a dictionary:

Python 3.6 ▾

```
graph = {
    0: [1],
    1: [0, 2, 3],
    2: [1, 3],
    3: [1, 2],
}
```

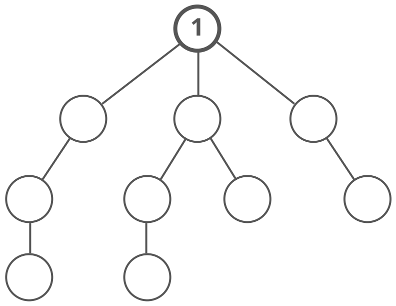
Since node `3` has edges to nodes `1` and `2`, `graph[3]` has the adjacency list `[1, 2]`.

format. Then we do a breadth-first search↴

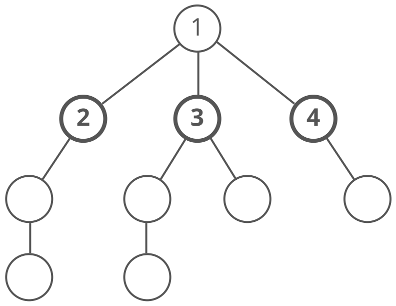
Breadth-first search (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.

Breadth-first search is like throwing a stone in the center of a pond. The nodes you explore "ripple out" from the starting point.

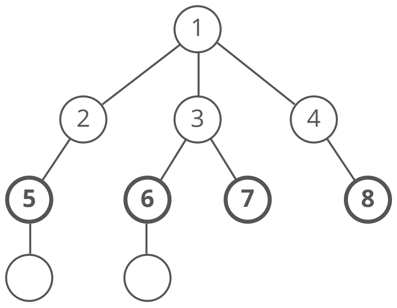
Here's a how a BFS would traverse this tree, starting with the root:



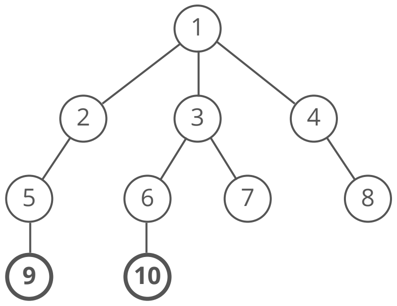
We'd visit all the immediate children (all the nodes that're one step away from our starting node):



Then we'd move on to all *those* nodes' children (all the nodes that're *two steps* away from our starting node):



And so on:



Until we reach the end.

Breadth-first search is often compared with **depth-first search**.

Advantages:

- A BFS will find the **shortest path** between the starting point and any other reachable node. A depth-first search will not necessarily find the shortest path.

Disadvantages

- A BFS on a binary tree *generally* requires more memory than a DFS.

from the sender, stopping once we reach the recipient.

In order to recover the actual shortest path from the sender to the recipient, we do two things:

1. *during* our breadth-first search, we keep track of how we reached each node, and
2. *after* our breadth-first search reaches the end node, we use our dictionary to *backtrack* from the recipient to the sender.

To make sure our breadth-first search terminates, we're careful to avoid visiting any node twice. We could keep track of the nodes we've already seen in a set, but, to save space, we reuse the dictionary we've already set up for recovering the path.

Python 3.6 ▾

```
from collections import deque


def reconstruct_path(previous_nodes, start_node, end_node):
    reversed_shortest_path = []

    # Start from the end of the path and work backwards
    current_node = end_node
    while current_node:
        reversed_shortest_path.append(current_node)
        current_node = previous_nodes[current_node]

    # Reverse our path to get the right order
    reversed_shortest_path.reverse()  # flip it around, in place
    return reversed_shortest_path  # no longer reversed


def bfs_get_path(graph, start_node, end_node):
    if start_node not in graph:
        raise Exception('Start node not in graph')
    if end_node not in graph:
        raise Exception('End node not in graph')

    nodes_to_visit = deque()
    nodes_to_visit.append(start_node)

    # Keep track of how we got to each node
    # We'll use this to reconstruct the shortest path at the end
    # We'll ALSO use this to keep track of which nodes we've
    # already visited
    how_we_reached_nodes = {start_node: None}

    while len(nodes_to_visit) > 0:
        current_node = nodes_to_visit.popleft()

        # Stop when we reach the end node
        if current_node == end_node:
            return reconstruct_path(how_we_reached_nodes, start_node, end_node)

        for neighbor in graph[current_node]:
            if neighbor not in how_we_reached_nodes:
                nodes_to_visit.append(neighbor)
                how_we_reached_nodes[neighbor] = current_node

    # If we get here, then we never found the end node
    # so there's NO path from start_node to end_node
    return None
```

Complexity

Our solution has two main steps. First, we do a breadth-first search of the user network starting from the sender. Then, we use the results of our search to backtrack and find the shortest path.

How much work is a breadth-first search?

In the worst case, we'll go through the BFS loop once for every node in the graph, since we only ever add each node to `nodes_to_visit` once (we check `how_we_reached_nodes` to see if we've already added a node before). Each loop iteration involves a constant amount of work to dequeue the node and check if it's our end node. If we have n nodes, then *this* portion of the loop is $O(N)$.

But there's more to each loop iteration: we also look at the current node's *neighbors*. Over all of the nodes in the graph, checking the neighbors is $O(M)$, since it involves crossing each edge twice: once for each node at either end.

Putting this together, the complexity of the breadth-first search is $O(N + M)$.

> BFS and DFS are common enough that it's often acceptable to just state their complexity as $O(N + M)$. Some interviewers might want you to derive it though, so definitely be ready in case they ask.

What about backtracking to determine the shortest path? Handling each node in the path is $O(1)$, and we could have at most N nodes in our shortest path. So, that's $O(N)$ for building up the path. Then, it's another $O(N)$ to reverse it. So, the total time complexity of our backtracking step is $O(N)$.

Putting these together, the time complexity of our entire algorithm is $O(N + M)$.

What about space complexity? The queue of nodes to visit, the mapping of nodes to previous nodes, and the final path ... they all store a *constant* amount of information *per node*. So, each data structure could take up to $O(N)$ space if it stored information about all of our nodes. That means our overall space complexity is $O(N)$.

Bonus

1. In our solution, we assumed that if one user (Min) could transmit a message to another (Jayden), then Jayden would also be able to transmit a message to Min. Suppose this wasn't guaranteed—maybe Min's cell phone transmits over shorter distances than Jayden's. How would our graph change to represent this? Could we still use BFS?
2. What if we wanted to find the *shortest* path? Assume we're given a GPS location for each user. How could we incorporate the distance between users into our graph? How would our algorithm change?
3. In our solution, we assumed that users never moved around. How could we extend our algorithm to handle the graph changing over time?

> Our app's design has a formal name: a **mesh network**. In a mesh network, data is sent from one node (here, a phone) to another *directly*, rather than through intermediate devices (here, cell towers). Assuming enough devices are in range, mesh networks provide multiple possible transmission paths, making them reliable even if some devices have failed.

What We Learned

The tricky part was *backtracking* to assemble the path we used to reach our `end_node`. In general, it's helpful to think of backtracking as two steps:

1. Figuring out *what additional information* we need to store in order to rebuild our path at the end (`how_we_reached_nodes`, in this case).
2. Figuring out how to reconstruct the path from that information.

And in this case, something interesting happened after we added `how_we_reached_nodes`—it made `nodes_already_seen` redundant! So we were able to remove it. A good reminder to always look through your variables at the end and see if there are any you can cut out.

Ready for more?

Check out our full course ➡

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.