Let's talk about graphs as a data structure. Graphs are bunch of nodes that are going to be connected with edges. And we are going to have bidirectional graphs where you can go back and forth that the edge is bi directional or directed graphs where we can only go one direction from a node to a given node. Let's start with bidirectional graphs. And within this graph structure we are going to explore a couple of different algorithms.

So our first graph is made up of some nodes that are integers 0, 1, 2, 3, and they're connected in a circle 0 to 1, 1 to 2, 2 to 3, 3 to 0. So we have our set of edges that show the connections and we want to see if we can get given the set of edges from the starting node 0 to an ending node 7. In this graph there is no 7. So we are going to fail. But this is a good demonstration of the algorithm.

So let's go to is reachable function. So is reachable function. We want to create an adjacency list and we are going to assume that our node values are in the range 0 to 99. And so we are going to have a vector, and for each index 0 to 99, we are going to have a vector of integers that show what that index is connected to. An alternative would be to use some kind of an unordered map that goes from an integer to a vector of integers.

If we had more than 100 or if they were integers, we would use an unordered map. But for now, let's stick to the vector data structure. So for the vector data structure, we are going to go through all of our edges and we'll make sure that we are adding our edges so that we create our adjacency data structure. So for example, we have our first is 2, second is 3. So 2 is connected to 3.

So at index 2 of the graph, we are going to put 3 into the vector of all the nodes that 2 is connected to. And similarly for graph at index 3, we are going to put it into the vector of all the things that tree is connected to. Once we have done our connections, we are going to test out two different functions, breadth first search and depth first search. So our breadth first search function uses a queue data structure and we can think about breadth first search as throwing a rock into a pond. When you see the ripples, the set of circles getting bigger and bigger and bigger.

Breadth first search works in a similar way. It's going to check if we can find the node that's at a distance of 1 and then at a distance of 2, and then at a distance of 3 and at a distance of 4 and so on. So the cube is going to hold all of our neighboring vertices, all of our neighboring nodes that are just a set distance away. And because graphs can have circularities, we are also going to keep an unordered set of visited vertices so we don't visit them again. And we are going to put into our queue our starting node.

And we can think of it as the kind of the middle of our circle. And we put our starting node saying, hey, that has been visited. We no longer have to visit visited again. So we'll process our queue. And so we start with node zero.

If this is the node that we want, if this is our target, then we can just return true. If not, we are going to look through all of our neighbors, all of the ones that zero is connected to. And in this case, because we have a circular structure, 0 is connected to 1 and 0 is connected to 3. So for example, for neighbor 1, we'll see if it has been visited. If it hasn't been visited, we'll say, hey, we are going to be visiting that neighbor and push that into our queue structure.

Similarly, for our node 3, we are going to check if it's been visited, it hasn't. So we'll put it into our queue and push it into our queue. So now we have our queue. We have just put 1 and then 3 in there, so we'll continue with 1. So we are looking at node 1.

That's not the node we want. So now we'll look at all of the nodes that one is going to be connected to. One is connected to zero and two. So first we look at our neighbor zero and it turns out neighbor zero is visited, so we don't put it back in the queue. And then our next neighbor is going to be two, and two has not been visited.

So we put it into our queue, so it's gone to the back of the queue. Next we are going to look at our node3.3, just like 1 is 1 distance away from 0.

And since we haven't gotten to our target, which is 7, and we skipped it, and then we look at all of the neighbors of our neighbor. Sorry, all of the neighbors of our node 3, 3 has the neighbors 0 and 2 in the in the case of 2, it has been visited. In the case of 0, it has been visited. So nothing goes into the queue. Next we look at our node 3.

That's not the node we are looking for look at our neighbors. One has been visited, three has been visited. So nothing to put. And now our queue is empty. As a result, we are going to return false from breadth first search.

Depth first search works in a similar way, but instead of using a cube data structure, it uses a stack type data structure. So what that means is that we are going to go to the one of the neighbors and then keep going to the next neighbor and the next neighbor keep following that until we can't. And if we need to, then we'll go back and look at the other neighbors. So let's look through that one, so we have a stack. Once again, we need to keep track of visited vertices so we don't have loops.

Starts from zero, our node is currently zero. That's not the target we are looking for. We look at our neighbors of zero, neighbors of zero are going to be one and three. So one hasn't been visited, so we put it into our stack. Next we look at three.

Three hasn't been visited, so we put it into our stack as well. Now we have one in the stack and then three in the stack. So we'll take the top node out of it. So from the three we'll look at three's neighbors. Three neighbors are going to be two which is going to get put into the stack as well as zero which doesn't get put into the stack.

So we have gone from zero to three and now we are going to and then we put two into the stack. So two comes out of the stack and we look at two's neighbors and we continue until we can we run out of everything in the stack. And once we have exhausted it, we end up returning false. So in this case we did a PFS where we looked at the nodes at the distance 1, this is 2, distance 3. And then we looked at DFS where we started from a node and kept following the set of nodes.

And then if we get stuck, we would go back and look at the other nodes that were in the stack. We can also do DFS recursively. So to do DFS recursively we are going to press the graph the starting nodes zero that are target node seven. And this time, because it's a recursive function, we are going to define our unordered set of visited nodes outside of the function and kind of pass it in. So DFS recursive is going to check if we are at the target.

If we are not at the target, say that we have visited that node and for all of the Neighbors call DFS recursive again. And if the DFS recursive returns true, then we'll return true. If not, at the very end we'll end up returning false. So in this example we looked at PFS. PFS didn't find the target 7.

DFS didn't find the target 7. DFS recursive didn't find the 7. So none of them were able to find the 7. And we got the same result on all of the graph algorithms as expected.

So in terms of our reachability, well, from zero we really can't reach seven. So let's also look at something that is reachable in this case 0 to 3 and Chase and kind of go through the algorithm. So once again we have putting the edges into an adjacency list. Then we are going to go to bfs.

We are starting from zero and we are searching for three. We initialized, we are starting from zero. Zero is not our target. We look at all of our neighbors. Our neighbors from zero are going to be one and three.

So those are going to get pushed into our queue.

And then we continue processing the queue. And now we took out one. One is not our target. We put one's neighbors into our queue and next we are going to take three out and three is our targets. So we end up returning true from BFS in terms of dfs.

Once again, the same kind of algorithm, but instead of a queue structure, we are using a stack structure.

We are starting from zero and zero is not our target. We look at our neighbors. Our neighbors are going to be one and three, so we push them onto the stack. So next we look at the node 3 from the stack and hey, we found it as our target and we can return true in terms of DFS recursive. We are going to have the same process, but in a recursive version.

So once again all of our graph algorithms return the same results. We are able to get from 0 to 3. When we are using breadth first search or depth first search. Our edges don't have any weights to them. They might have directionality, but they don't have any weights.

Later we are going to see edges that have a certain length, such as a road as part of it.

So we have reached three. Let's look at one more graph to get a better understanding of breadth first search and depth first search. So in this case we have a tree like structure. We have 0 at the top, 1 and 2 as the children. 1 has 3 and 4 and then 2 has 5 and 6.

Then 3 has 7 and then 4 has 8 as its as its child. So we provide these as edges. Once again we look at if it's reachable, we create our adjacency list.

Let's put a breakpoint here and just continue. And then let's start from bfs. We are going to use a queue structure. So we are going to start from zero is our node zero we said had one and two. So we are trying to get to a target of of 7.

So in this case our neighbors are 1 and 2. So we are going to put our neighbor 1 into the queue. And next we are putting our neighbor two into the queue. And now we go back and process the queue again. So we are now looking at our node1 that's in the queue.

And node1 has neighbors0, so it can go up to the parents since it's direct directional. But since it's been visited, we don't want to go to zero from node one. We can also go to three. So we put that one into our queue and four into our queue. Next we are going to look at the node 2.

So 0 adds 1 and 2 as its children. We looked at all of the children of one and we put them in the queue. Now we are going to look at two and put all of its children in the queue as well. So we go through two's two's neighbors. Zero is a neighbor we don't put but also has five we put into the queue and six into the queue.

Our queue is not empty. So now we are looking at node 3.

Node 3 has node 1 which is the parent as a neighbor. We don't put it there. It also has seven as one of its neighbor. Although we are finding seven, we don't immediately return. We are going to put it into the queue and wait for the queue to do its its thing.

So we put seven into the queue and now we are going to look at fours neighbors. We put them in the queue. Next we are looking at five neighbors, we put them into the queue and then we are looking at six neighbors, we put them into the queue. And finally we are looking at seven and say hey, seven is our target so we can return true as part of it. So we covered it using breadth first search.

So if it was a tree like structure, we can think of it going level by level by level, extending each level at a time.

Oh, I skipped the. Let's see if I can go back.

Let's see. We want to go through graph 1. We don't care about the graph 1 right now. I want to go to graph 2s reach reachability for 7. So let's go there.

And then I want to go to the DFs for that one. So let's quickly go through the adjacency.

Okay, so let's look at depth first search going from zero to seven.

So we are putting a stack like structure. We are starting from zero, we put it onto the stack, then we pop it from the stack. It's not what we are looking for. So we look at its neighbors. 0 has the neighbors 1 and 2.

So we put 1 into the queue and then 2 into the queue. Now we are going to look at two that's on the stack. Look at its neighbors zero which is the parent of two or is is a neighbor.

We don't put it onto the queue. 2 has additionally 5 and 6 as part of it. So they go, they go onto the stack. Now then we look at six. Next we look at five and then we are going to look at one which is going to have neighbors three and four.

Then we are going to look at four and four heads neighbor eight. So finally we look at eight. Eight doesn't have it and we are still looking. And finally we are pushing seven as a neighbor and we get seven out and we have done it. So maybe it will be helpful to kind of have a little graph of what we have been looking at.

So let's have a look. So we started with 0 and 0 had 1 and 2. 1 had 3 and 4, 2 had 5 and 6 and then 3 had 7 and 4 head 8. So based on that we started from 0. Let me see if I can pick another color.

So let's pick another color. So we started from zero, we put one on the stack and then two on the stack. Since two was the top of the stack, we went this way. From 2, we looked at its neighbors 5 on this we put 5 on the stack and then 6 on the stack and then we visited 6. 6.

Well its only neighbor is 2 which has already been visited. So we didn't visit that. So we came back to 2 and visited its neighbor 5. From 5 it had no other neighbors to add. So we went back to two.

Then we went back to zero. So now we have processed all of zero's neighbors that go from two. Now we went to one, we looked at its neighbors in the depth first search 3 and 4. 3 went on the stack 1, then 4 went on the stack. So we explored 4.

Then we explored 8. Then having nothing else at those levels we went back and then we explored three. And finally we explored seven to find it. So depth first search chooses a direction and keeps going as much as possible. And when it hits a dead end, it goes back up and then it goes down as part of it.

Let's see if we can clear this annotation so we don't have to suffer from it clearing all drawings. Let's continue.

And if we wanted to do DFS recursive going from 0 to 7, we'll have a similar structure. But now for each neighbor we are kind of trying it out first. So this time we are trying out if one, if we can get to seven from one. So if we go into there, one is not our node. So we put it into our visited.

Then we look, zero is visited. The next one that we are going to look at is from three. So we are going zero, one, three and then it's going to be seven. So we go here.

So one has already been visited, now it's going to be looking at 7. We go into recursive. Our node 7 is equal to our target. So we return true, which returns true from the recursive loop, which also returns true, which also returns true. So our recursive DFS ends up returning true for all of them.

So in this case we can use the same algorithm to check using BFS and dfs, if something is reachable. So breadth first search and depth first search are both good algorithms. One thing to remember is is breadth first search is going to find the shortest path if we are keeping track of the path, because it's going to explore everything at that distance of one, then at a distance of two, then at a distance of three. So it's going to find it at the shortest distance possible. Whereas DFS might take a very long way to to get to its destination.

And that's breadth first search and depth first search algorithms.