# ProgrammingAbstractions-Lecture23

**Instructor (Julie Zelenski)**:Okay, we got volume and everything. All right, here we go. Lots of you think your learning things that aren't gonna actually someday help you in your future career. Let's say you plan to be president. You think, "Should I take 106b; should I not take 106b?" Watch this interview and you will see. [Video]

We have questions, and we ask our candidates questions, and this one is from Larry Schwimmer. What – you guys think I'm kidding; it's right here. What is the most efficient way to sort a million 32-bit integers?

Well – I'm sorry. Maybe we should – that's not a – I think the bubble sort would be the wrong way to go. Come on; who told him this?

**Instructor (Julie Zelenski)**:There you go. So, apparently – the question was actually by a former student of mine, in fact, Larry Schwimmer. So apparently, he's been prepped well. As part of going on the campaign trail, not only do you have to know all your policy statements, you also have to have your N2 and login sorts kept straight in your mind, so – just so you know.

Okay. So let's talk about where we are administratively. PQ coming in, big milestone, right? So PQ, one of the rights of passage in terms of getting the pointers, and the link lists, and all that goopy, low level stuff, sort of, straight in your head. I expect this was, sort of, a bigger investment on your part, but let's do a little poll, and we'll find out how much joy there was in Mudville on this. How many people finished in under ten hours? I don't think anybody could/should. My gosh – well, okay. There you go – a few, a few. Ten to fifteen? Still, I think a very impressive, kind of, efficiency to getting that done in that. Fifteen to twenty? Clinton, you're not raising your hand; what are you telling me? Uh, oh.

**Student:**I had a late day.

**Instructor (Julie Zelenski)**:Late day – oh, okay. All right. More than 20? Okay, and this more than 20 people may still be in this late day camp. It is one of the heftier pieces of work that we've given you, so we, kind of, are expecting that, but hopefully, at this point, on of the things that should've been the mid-quarter eval is, right, was people saying, "Yeah, this pointer – " and a couple things they said pointers, still mystical, link lists feeling a little shaky on that, as well as, sort of, Big O not totally solid in their head, and I do think that with the sorting lab and the PQ work on the Big O, hopefully that last category has come together but also the pointers and link lists, right, you're starting to get real hands-on practice which helps to make the concepts, kind of, make more sense than they did when they're just me yakking about it. Your last assignment is pathfinder, which is gonna go out today. I'm gonna actually demo it just a little bit because I think this is an awesome, kind of, capstone to finish off the quarter with, and I just love playing with it. So I want to show it to you just to get you psyched for what's going on here. So what pathfinder is doing is a graph search problem, but it's, kind of, just a version of something that MapQuest, and Google Maps, and all those little GPS devices are doing is path finding, right? Given you're at this position, and you want to get at this address, what are the routes that take you from here to there, and which of those is the best on given the constraints you're working under? So maybe it's taken advantage of faster roads, freeway roads, or roads that are shorter, roads that are in better repair, roads where there's less traffic. In this case, we can, kind of, simplify this problem down to, well, if we just had the distance information and all this connecting data about which places you can be at – you can be at the golf course, and you can bike over to FroSoCo, or you could walk Robley Field, or over to the Gates Center, and things like that. It's, like, if you were at one of these places, what path could you take that would get you from one place to the other, in this case, minimizing overall distance. So it seems like a little bit of magic, right? Like, you go to those sites, you type in these things, and it finds you things, but, in fact, actually, at this point, you're capable of writing a program that can do exactly what those things do, and the key thing is having an efficient algorithm and a big pile of data to work on. We have some smaller datas that make it easier to test on but the same approach applies in the large as well as the small. And so, in the case of the program, what it does is I can pick a place that I am at. I can say, oh, I'm over here at Escondido at my kid's elementary school, and I need to get to Gates, and it'll tell me that I need to cut through campus going down by the east residences and then across through the quad, cutting off the corner and heading to Gates is my shortest path that connects up those two things. And so all the other things, kind of, out there, you know, it considered as part of its strategy here but was able to zero in on a path, in this case, is about a third of a mile between here and there, and then it actually tells me a little bit about how it did its work. It actually looked at 112 different paths before it concluded that was the best one, which is a fairly small number given all the paths that are out there. So it's actually doing some very efficient pruning to decide which ones to look at first. And so I can do that again with some more things, doing other ones. I'm over here at the stadium, and I'd like to get to Robley Field. It turns out, given the paths that it has, it doesn't have a direct path, so it has to, kind of, cut around and come back in that case, but kind of, cutting through the front of the Quad, and coming around by Lagunita. So a very neat, neat program, right, that makes use of, kind of, everything we've seen this quarter. So lots of uses for the ADTs, the set, and the vector, and stack, and queue have some roles to play. We'll see the priority queue come back. So the priority queue that you've just finished, right, is gonna have a role to play here. We're gonna take it and turn it into a template form as part of the algorithms that you're implementing need a priority queue, so you'll turn that integer-based priority queue into a template one, and then draw and manage this grafting. A little bit of graphics work, some of the stuff you saw in the chaos and the maze assignment's come back, a little bit of drawing to do here to get the visual results up. The other thing it does, actually it does another secondary graph calculation, which is the computation of what's called the Minimal Spanning Tree, which is if you took apart all the roads, you know, or paths that exist on campus, and you were interested in reinstalling them but choosing the minimal set that will connect up all the locations you have on campus so that you can get somewhere from everywhere, but without any redundancies or cycles, and choosing the smallest of your options. So if you were trying to wire up, let's say, internet across the campus, and you needed to hit each of those nodes, where are the places you could lay those cables to connect everyone onto the campus network using the minimal amount of wire? And that's the second algorithm that is what you're implementing in this project, so a nice thing to know how to do. So we're gonna talk about why does that – what does that work; what is it doing? Let's talk about graphs. Okay. So graphs and graphs algorithms are the topics for today which relate to that final assignment, and that final assignment's coming in on Friday of dead week, so that's a week and a few days from today. You can use one late day on it if you really need to which extends it into Monday of exam week, probably not a good choice though if you have early in the week exams. So finishing it on time, kind of, gets it behind you and lets you focus on the exam studying. Our exam is at the very end of exam week, right, so that Friday afternoon 12:00 to 3:00. I know being at the end, kind of, interferes with the rushing off to have a good time over spring break, but it also gives you extra time to study. So maybe you can think of it as, kind of, a tradeoff, and then I will put out a practice exam either later this week or next week that gives you and idea of, kind of, what to expect in terms of going into that exam. The last section of the reader that we're gonna be looking at on Friday is about hashing and hash tables as an alternate implementation for the map, and that's covered in Chapter 11, and that's actually, kind of, the final material coming out of the reader. What I'll be doing next week is just looking at some advanced topics, trying to, kind of, pull together some compare and contrast in the data structures and algorithms we've seen. So I'll talk a little bit about, kind of, the big picture view of how we make choices among these things, and where we go from here. All right, any administrative questions? So let me give you some examples of things that are graphs, just to get you thinking about the data structure that we're gonna be working with and playing around with today. Sometimes you think of graphs of being those, like, bar graphs, right, your graphing histograms. This is actually a different kind of graph. The computer scientist's form of graph is just a generalized, recursive data structure that, kind of, starts with something that looks a little bit like a tree and extends it to this notion that there are any number of nodes, in this case, represented as the cities in this airline route map, and then there are connections between them, sometimes called edges or arcs that wire up the connections that in the case, for example, of this graph are showing you which cities have connecting flights. You have a direct flight that takes you from Las Vegas to Phoenix, right, one from Phoenix to Oklahoma City. There is not a direct flight from Phoenix to Kansas City, but there's a path by which you could get there by taking a sequence of connecting flights, right, going from Phoenix, to Oklahoma City, to Kansas City is one way to get there. And so the entire, kind of, map of airline routes forms a graph of the city, and there are a lot of interesting questions that having represented that data you might want to answer about finding flights that are cheap, or meet your time specification, or get you where you want to go, and that this data would be, kind of, primary in solving that problem. Another one, right, another idea of something that's represented as a graph is something that would support the notion of word ladders, those little puzzles where you're given the word "chord," and you want to change it into the word "worm," and you have these rules that say, well, you can change one letter at a time. And so the connections in the case of the word letter have to do with words that are one character different from their neighbors, have direct connections or arcs between them. Each of the nodes themselves represents a word, and then paths between that represent word ladders. That if you can go from here through a sequence of steps directly connecting your way, each of those stepping stones is a word in the chain that makes a word ladder. So things you might want to solve there is what's the shortest word ladder? How many different word ladders – how many different ways can I go from this word to that one could be answered by searching around in a space like this. Any kind of prerequisite structure, such as the one for the major that might say you need to take this class before you take that class, and this class before that class, right, forms a structure that also fits into the main graph. In this case, there's a connection to those arcs. That the prerequisite is such that you take 106a, and it leads into 106b, and that connection doesn't really go in the reverse, right? You don't start in 106b and move backwards. Whereas in the case, for example, of the word ladder, all of these arcs are what we called undirected, right? You can traverse from wood to word and back again, right? They're equally visitable or neighbors, in this case. So there may be a situation where you have directed arcs where they really imply a one-way directionality through the paths. Another one, lots and lots of arrows that tell you about the lifecycle of Stanford relationships all captured on one slide in, sort of, the flowchart. So flowcharts have a lot of things about directionality. Like, you're in this state, and you can move to this state, so that's your neighboring state, and the arcs in there connect those things. And so, as you will see, like, you get to start over here and, like, be single and love it, and then you can, kind of, flirt with various degrees of seriousness, and then eventually there evolves a trombone and serenade, which, you know, you can't go back to flirting aimlessly after the trombone and serenade; there's no arc that direction. When somebody gets the trombone out, you know, you take them seriously, or you completely blow them off. There's the [inaudible] there. And then there's dating, and then there's two-phase commit, which is a very important part of any computer scientist's education, and then, potentially, other tracks of children and what not. But it does show you, kind of, places you can go, right? It turns out you actually don't get to go from flirt aimlessly to have baby, so write that down – not allowed in this graph. And then it turns out some things that you've already seen are actually graphs in disguise. I didn't tell you when I gave you the maze problem and its solution, the solving and building of a maze, that, in fact, what you're working on though is something that is a graph. That a perfect maze, right, so we started with this fully disconnected maze. If you remember how the Aldis/Broder algorithm worked, it's like you have all these nodes that have no connections between them, and then you went bopping around breaking down walls, which is effectively connecting up the nodes to each other, and we kept doing that until actually all of the nodes were connected. So, in effect, we were building a graph out of this big grid of cells by knocking down those walls to build the connections, and when we were done, then we made it so that it was possible to trace paths starting from that lower corner and working our way through the neighbors all the way over to there. So in the case of this, if you imagine that each of these cells was broken out in this little thing, there'd be these neighboring connections where this guy has this neighbor but no other ones because actually it has no other directions you can move from there, but some of them, for example like this one, has a full set of four neighbors you can reach depending on which walls are intact or not. And so the creation of that was creating a graph out of a set of disconnected nodes, and then solving it is searching that graph for a path. Here's one that is a little horrifying that came from the newspaper, which is social networking, kind of, a very big [inaudible] of, like, on the net who do you know, and who do they know, and how do those connections, like, linked in maybe help you get a job. This is one that was actually based on the liaisons, let’s say, of a group of high school students, right, and it turns out there was a lot of more interconnected in that graph than I would expect, or appreciate, or approve of but interesting to, kind of, look at. So let’s talk, like, concretely. How are we gonna make stuff work on graphs? Graphs turn out to actually – by showing you that, I’m trying to get you this idea that a lot of things are really just graphs in disguise. That a lot of the problems you may want to solve, if you can represent it as a graph, then things you learn about how to manipulate graphs will help you to solve that kind of problem. So the basic idea of a graph, it is a recursive data structure based on the node where nodes have connections to other nodes. Okay. So that’s, kind of, like a tree, but it actually is more freeform than a tree, right? In a tree, right, there’s that single root node that has, kind of, a special place, and there’s this restriction about the connections that there’s a single path from every other node in the tree starting from the root that leads to it. So you never can get there by multiple ways, and there’s no cycles, and it’s all connected, and things like that. So, in terms of a graph, it just doesn’t place those restrictions. It says there’s a node. It has any number of pointers to other nodes. It potentially could even have zero pointers to other’s nodes, so it could be on a little island of its own out here that has no incoming or outgoing flights, right? You have to swim if you want to get there.

And then there’s also not a rule that says that you can’t have multiple ways to get to the same node. So if you’re at B, and you’re interested in getting to C, you could take the direction connection here, but you could also take a longer path that bops through A and hits to C, and so there’s more than one way to get there, which would not be true in a tree structure where it has that constraint, but that adds some interesting challenges to solving problems in the graphs because there actually are so many different ways you can get to the same place. We have to be careful avoiding getting into circularities where we’re doing that, and also not redundantly doing work we’ve already done before because we’ve visited it previously. There can be cycles where you can completely come around. I this one, actually, doesn’t have a cycle in it. Nope, it doesn’t, but if I add it – I could add a link, for example, from C back to B, and then there would be this place you could just, kind of, keep rolling around in the B, A, C area. So we call each of those guys, the circles here, the nodes, right? The connection between them arcs. We will talk about the arcs as being directed and undirected in different situations where we imply that there’s a directionality; you can travel the arc only one-way, or in cases where it’s undirected, it means it goes both ways. We’ll talk about two nodes being connected, meaning there is a direct connection, an arc, between them, and if they are not directly connected, there may possibly be a path between them. A sequence of arcs forms a path that you can follow to get from one node to another, and then cycles just meaning a path that revisits one of the nodes that was previously traveled on that path. So I’ve got a little terminology. Let’s talk a little bit about how we’re gonna represent it.

So before we start making stuff happen on them, it’s, like, well, how does this – in C++, right, what’s the best way, or what are the reasonable ways you might represent this kind of connected structure? So if I have this four-node graph, A, B, C, D over where with the connections that are shown with direction, in this case, I have directed arcs. That one way to think of it is that what a graph really is is a set of nodes and a set of arcs, and by set, I mean, sort of, just the generalization. There might be a vector of arcs; there might be a vector of nodes, whatever, but some collection of arcs, some collection of nodes, and that the nodes might have information about what location they represent, or what word, or what entity, you know, in a social network that they are, and then the arcs would show who is directly connected to whom. And so one way to manage that, right, is just to have two independent collections – a collection of nodes and a collection of arcs, and that when I need to know things about connections, I'll have to, kind of, use both in tandem. So if I want to know if A is connected to B, I might actually have to just walk through the entire set of arcs trying to find one that connects A and B as its endpoints. If I want to see if C is connected to B, same thing, just walk down the whole set. That's probably the simplest thing to do, right, is to just, kind of, have them just be maintained as the two things that are used in conjunction. More likely, what you're gonna want to do is provide some access that when at a current node, you're currently trying to do some processing from the Node B, it would be convenient if you could easily access those nodes that emanate or outgo from the B node. That's typically the thing you're trying to do is at B try to visit its neighbors, and rather that having to, kind of, pluck them out of the entire collection, it might be handy if they were already, kind of, stored in such a way to make it easy for you to get to that. So the two ways that try to represent adjacency in a more efficient way are the adjacency list and the adjacency matrix. So in an adjacency list representation, we have a way of associating with each node, probably just by storing it in the node structure itself, those arcs that outgo from there. So, in this case, A has as direct connection to the B and C Nodes, and so I would have that information stored with the A Node, which is my vector or set of outgoing connections. Similarly, B has one outgoing connection, which goes to D. C has an outgoing connection to D, and then D has outgoing connections that lead back to A and to B. So at a particular node I could say, well, who's my neighbors? It would be easily accessible and not have to be retrieved, or searched, or filtered.

Another way of doing that is to realize that, in a sense, what we have is this, kind of, end by end grid where each node is potentially connected to the other N -1 nodes in that graph. And so if I just build a 2x2 matrix that's labeled with all the node names on this side and all the node names across the top, that the intersection of this says is there an arc that leads away from A and ends at B? Yes, there is. Is there one from A to C? And in the places where there is not an existing connection, a self loop or just a connection that doesn't exist, right, I have an empty slot. So maybe these would be Booleans true and false or something, or maybe there'd be some more information here about the distance or other associated arc properties for that particular connection.

In this case, it's actually very easy then to actually do the really quick search of I'm at A and I want to know if I can get to B, then it's really just a matter of reaching right into the slot in constant time in that matrix to pull it out, and so this one involves a lot of searching to find things. This one involves, perhaps, a little bit of searching. If I want to know if A is connected to B, I might have to look through all its outgoing connections, right? This one gives me that direct access, but the tradeoffs here has to do with where the space starts building up, right? A full NxN matrix could be very big, and sometimes we're allocating space as though there are a full set of connections between all the nodes, every node connected to every other, and in the cases where a lot of those things are false, there’s a lot of capacity that we have set aside that we’re not really tapping into.

And so in the case of a graph that we call dense, where there’s a lot of connections, it will use a lot of that capacity, and it might make sense in a graph that’s more sparse, where there’s a few connections. So you have thousands of nodes, but maybe only three or four on average are connected to one. Then having allocated 1,000 slots of which to only fill in three might become rather space inefficient. You might prefer an adjacency list representation that can get you to the ones you’re actually using versus the other, but in terms of the adjacency matrices is very fast, right? A lot of space thrown at this gives you this immediate 0,1 access to know who your neighbors are. So we’re gonna use the adjacency list, kind of, strategy here for the code we’re gonna do today, and it’s, kind of, a good compromise between the two alternatives that we’ve seen there. So I have a struct node. It has some information for the node. Given that I’m not being very specific about what the graph is, I’m just gonna, kind of, leave that unspecified. It might be that it’s the city name. It might be that it’s a word. It might be that it’s a person in a social network, you know, some information that represents what this node is an entity, and then there’s gonna be a set of arcs or a set of nodes it’s connected to, and so I’m using vector here. I could be using set. I could be using a raw array, or a link list, or any of these things. I need to use some unbounded collection though because there’s no guarantee that they will be 0, or 1, or 2 the way there is in a link list or a tree where there’s a specified number of outgoing links. There can be any number of them, so I’m just leaving a variable size collection here to do that work for me. The graph itself, so I have this idea of all the nodes in the graph, right, would each get a new node structure and then be wired up to the other nodes that they are connected to, but then when I’m trying to operate on that graph, I can’t just take one pointer and say here’s a pointer to some node in the graph and say this is – and from here you have accessed everything. In the way that a tree, the only thing we need to keep track of is the pointer to the root, or a link list, the only thing we keep a pointer to, typically, is that frontmost cell, and from there, we can reach everything else. There is no special head or root cell in a graph. A graph is this being, kind of, a crazy collection without a lot of rules about how they are connected. In fact, it’s not even guaranteed that they totally are connected. That I can’t guarantee that if I had a pointer, for example, to C, right, I may or may not be able to reach all the nodes from there. If I had a pointer to A, right, A can get to C and B, and then also down to D, but, for example, there could just be an E node over here that was only connected to C or not connected to anything at all for that matter, connected to F in their own little island.

That there is no way to identify some special node from which you could reach all the nodes. There isn’t a special root node, and you can’t even just arbitrarily pick one to be the root because actually there’s no guarantee that it will be connected to all of the others. So it would not give you access to the full entirety of your graph if you just picked on and said I want anything reachable from here, it might not get you the whole thing. So, typically, what you need to really operate on is the entire collection of node stars. You’ll have a set or a vector that contains all of the nodes that are in the graph, and then within that, there’s a bunch of other connections that are being made on the graph connectivity that you’re also exploring.

So let’s make it a little bit more concrete and talk a little bit about what really is a node; what really is an arc? There may be, actually, be some more information I need to store than just a node is connected to other nodes. So in the case of a list or a tree, the next field and the left and the right field are really just pointers for the purpose of organization. That’s the pointer to the next cell. There’s no data that is really associated with that link itself.

That’s not true in the case of a graph. That often what the links that connect up the nodes actually do have a real role to play in terms of being data. It may be that what they tell you is what road you’re taking here, or how long this path is, or how much this flight costs, or what time this flight leaves, or how full this flight already is, or who knows, you know, just other information that is more than just this nodes connected to that one. It’s, like, how is it connected; what is it connected by, and what are the details of how that connection is being represented?

So it’s likely that what you really want is not just pointers to other nodes but information about that actual link stored with an arc structure. So, typically, we’re gonna have an arc which has information about that arc, how long, how much it costs, you know, what flight number it is, that will have pointers to the start and end node that it is connecting. The node then has a collection of arcs, and those arcs are expected to, in the adjacency list form, will all be arcs that start at this node. So their start node is equal to the one that’s holding onto to them, and then they have an end pointer that points to the node at the other end of the arc.

Well, this gets us into a little C++ bind because I’ve described these data structures in a way that they both depend on each other; that an arc has pointers to the start and the end node. A node has a collection of arcs, which is probably a vector or a set of arc pointers, and so I’m starting to define the structure for arc, and then I want to talk about node in it, and I think, oh, okay, well, then I better define node first because C++ always likes to see things in order. Remember, it always wants to see A, and then B if B uses A, and so in terms of how your functions and data structures work, you’ve always gotten this idea like, well, go do the one first.

Well, if I say, okay, arc’s gonna need node. I start to write arc, and I say, oh, I need to talk about node. Well, I better put node up here, right? But then when I’m starting to write node, I start talking about arc, and they have a circular reference to each other, right? They both want to depend on the other one before we’ve gotten around to telling the compiler about it. One of them has to go first, right? What are we gonna do?

What we need to do is use the C++ forward reference as a little hint to the compiler that we are gonna be defining a Node T structure in a little bit; we’re gonna get to it, but first we’re gonna define the Arc T structure that uses that Node T, and then we’re gonna get around to it. So we’re gonna give it, like, a little heads up that says there later will be an Act 2 of this play. We’re gonna introduce a character called Node T. So you can now start talking about Node T in some simple ways based on your agreement to the compiler that you plan on telling it more about Node T later.

So the struct Node T says there will be a struct called Node T. Okay, the compiler says, okay, I'll write that down. You start defining struct Arc T, and it says, oh, I see you have these pointers to the node T. Okay. Well, you told me there'd be a struct like that; I guess that'll work out. And now you told it what struct Node T is. It's, like, oh, I have a vector with these pointers to arc, and then it says, okay, now I see the whole picture, and it all worked out. So it's just a little bit of a requirement of how the C++ compiler likes to see stuff in order without ever having to go backwards to check anything out again.

Okay. So I got myself some node structures, got some arc structures; they're all working together. I'm gonna try to do some traversals. Try to do some operations that work their way around the graph. So tree traversals, right, the pre, and post, and in order are ways that you start at the root, and you visit all the nodes on the tree, a very common need to, kind of, process, to delete nodes, to print the nodes, to whatnot. So we might want to do the same thing on graphs. I've got a graph out there, and I'd like to go exploring it. Maybe I just want to print, and if I'm in the Gates building, what are all the places that I can reach from the Gates building? Where can I go to; what options do I have?

What I'm gonna do is a traversal starting at a node and, kind of, working my way outward to the visible, reachable nodes that I can follow those paths to get there. We're gonna look at two different ways to do this. One is Depth-first and one is Breadth-first, and I'll show you both algorithms, and they will visit the same nodes; when all is done and said, they will visit all the reachable nodes starting from a position, but they will visit them in different order. So just like the pre/post in order tree traversals, they visit all the same notes. It's just a matter of when they get around to doing it is how we distinguish depth and breadth-first traversals. Because we have this graph structure that has this loose connectivity and the possibility of cycles and multiple paths to the same node, we are gonna have to be a little bit careful at how we do our work to make sure that we don't end up getting stuck in some infinite loop when we keep going around the ABC cycle in a particular graph. That we need to realize when we're revisiting something we've seen before, and then not trigger, kind of, an exploration we've already done. So let's look at depth-first first. This is probably the simpler of the two. They're both, actually, pretty simple. Depth-first traversal uses recursion to do its work, and the idea is that you pick a starting node – let's say I want to start at Gates, and that maybe what I'm trying to find is all the reachable nodes from Gates. If I don't have a starting node, then I can just pick one arbitrarily from the graph because there is actually no root node of special status, and the idea is to go deep. That in the case of, for example, a maze, it might be that I choose to go north, and then I just keep going north. I just go north, north, north, north, north until I run into a dead wall, and then I say, oh, I have to go east, and I go east, east, east, east, east. The idea is to just go as far away from the original state as I can, just keep going outward, outward, outward, outward, outward, outward, outward, and eventually I'm gonna run into some dead end, and that dead end could be I get to a node that has no outgoing arcs, or it could be that I get to a node that whose only arcs come back to places I've already been, so it cycles back on itself, in which case, that also is really a dead end.

So you go deep. You go north, and you see as far as you can get, and only after you've, kind of, fully explored everything reachable from starting in the north direction do you backtrack, unmake that decision, and say, well, how about not north; how about east, right? And then find everything you can get to from the east, and after, kind of, going all through that, come back and try south, and so on. So all of the options you have exhaustively getting through all your neighbors in this, kind of, go-deep strategy. Well, the important thing is you realize if you've got this recursion going, so what you're actually doing, basically, is saying I'm gonna step to my neighbor to the right, and I'm gonna explore all the things, so do a depth-first search from there to find all the places you can reach, and that one says, well, I'm gonna step through this spot and go to one of my neighbors. And so we can't do that infinitely without getting into trouble. We need to have a base case that stops that recursion, and the two cases that I've talked about – one is just when you have no neighbors left to go. So when you get to a node that actually, kind of, is a dead end, or that neighbor only has visited neighbors that surround it.

So a very simple little piece of code that depends on recursion doing its job. In this case, we need to know which nodes we have visited, right? We're gonna have to have some kind of strategy for saying I have been here before. In this case, I'm using just a set because that's an easy thing to do. I make a set of node pointers, and I update it each time I see a new node; I add it to that set, and if I ever get to a node who is already previously visited from that, there's no reason to do any work from there; we can immediately stop.

So starting that step would be empty. If the visit already contains the node that we're currently trying to visit, then there's nothing to do. Otherwise, we go ahead and add it, and there's a little node here that says we'll do something would occur. What am I trying to do here? Am I trying to print those nodes? Am I trying to draw pictures on those nodes, highlight those nodes, you know, something I'm doing with them. I don't know what it is, but this is the structure for the depth-first search here, and then for all of the neighbors, using the vector of outgoing connections, then I run a depth-first search on each of those neighbors passing that visited set by reference through the whole operation. So I will always be adding to and modifying this one set, so I will be able to know where I am and where I still need to go.

So if we trace this guy in action – if I start arbitrarily from A, I say I'd like to begin with A. Then the depth-first search is gonna pick a neighbor, and let's say I happen to just work over my neighbors in alphabetical order. It would say okay, well, I've picked B. Let's go to everywhere we can get to from B. So A gets marked as visited. I move onto B; I say, well, B, search everywhere you can to from B, and B says okay, well, I need to pick a neighbor, how about I pick C? And says and now, having visited C, go everywhere you can get to from C. Well, C has no neighbors, right, no outgoing connections whatsoever. So C says, well, okay, everywhere you can get to from C is C, you know, I'm done.

That will cause it to backtrack to this place, to B, and B says okay, I explored everywhere I can get to from C; what other neighbors do I have? B has no other neighbors. So, in fact, B backtracks all the way back to A. So now A says okay, well, I went everywhere I can get to from B, right? Let me look at my next neighbor. My next neighbor is C. Ah, but C, already visited, so there's no reason to go crazy on that thing. So, in fact, I can immediately see that I’ve visited that, so I don’t have any further path to look at there. I’ll hit D then as the next one in sequence.

So at this point, I’ve done everything reachable from the B arm, everything reachable from the C arm, and now I’m starting on the D arm. I said okay, where can I get to from D? Well, I can get to E. All right, where can I get to from E? E goes to F, so the idea is you think of it going deep. It’s going as far away as it can, as deep as possible, before it, kind of, unwinds getting to the F that has no neighbors and saying okay. Well, how about – where can I get to also from E? I can get to G. From G where can I get to? Nowhere, so we, kind of, unwind the D arm, and then we will eventually get back to and hit the H.

So the order they actually were hit in this case happened to be alphabetical. I assigned them such that that would come out that way. That’s not really a property of what depth-first search does, but think of it as like it’s going as deep as possible, as far away as possible, and so it makes a choice – it’s pretty much like the recursive backtrackers that we’ve seen. In fact, they make a choice, commit to it, and just move forward on it, never, kind of, giving a backwards glance, and only if the whole process bottoms out does it come back and start unmaking decisions and try alternatives, eventually exploring everything reachable from A when all is done and said.

So if you, for example, use that on the maze, right, the depth-first search on a maze is very much the same strategy we’re talking about here. Instead of the way we did it was actually breadth-first search, if you remember back to that assignment, the depth-first search alternative is actually doing it the – just go deep, make a choice, go with it, run all the way until it bottoms out, and only then back up and try some other alternatives.

The breadth-first traversal, gonna hit the same nodes but just gonna hit them in a different way. If my goal were to actually hit all of them, then either of them is gonna be a fine strategy. There may be some reason why I’m hoping to stop the search early, and that actually might dictate why I would prefer which ones to look at first. The breadth-first traversal is going to explore things in terms of distance, in this case, expressed in terms of number of hops away from the starting node. So it’s gonna look at everything that’s one hop away, and then go back and look at everything two hops away, and then three hops away.

And so if the goal, for example, was to find a node in a shortest path connection between it, then breadth-first search, by looking at all the ones one hop away, if it was a one hop path, it’ll find it, and if it’s a two hop path, it’ll find it on that next round, and then the three hop path, and there might be paths that are 10, and 20, and 100 steps long, but if I find it earlier, I won’t go looking at them. It actually might prove to be a more efficient way to get to the shortest solution as opposed to depth-first search which go off and try all these deep paths that may not ever end up where I want it to be before it eventually made its way to the short solution I was wanting to find.

So the thing is we have the starting node. We’re gonna visit all the immediate neighbors. So, basically, a loop over the immediate one, and then while we’re doing that, we’re gonna actually, kind of, be gathering up that next generation. Sometimes that's called the Frontier, sort of, advancing the frontier of the nodes to explore next, so all the nodes that are two hops away, and then while we're processing the ones that are two hops away, we're gonna be gathering the frontier that is three hops away. And so at each stage we'll be, kind of, processing a generation but while assembling the generation N+1 in case we need to go further to find those things. We'll use an auxiliary data structure during this. That management of the frontier, keeping track of the nodes that are, kind of, staged for the subsequent iterations, the queue is a perfect data structure for that. If I put the initial neighbors into a queue, as I dequeue them, if I put their neighbors on the back of the queue, so enqueue them behind all the one hop neighbors, then if I do that with code, as I'm processing Generation 1 off the front of the queue, I'll be stacking Generation 2 in the back of the queue, and then when all of Generation 1 is exhausted, I'll be processing Generation 2, but, kind of, enqueuing Generation 3 behind it. Same deal that we had with depth-first search though is we will have to be careful about cycles will pull past that when there's more than one possibility of how we get to something, we don't want to, kind of, keep going around that cycle or do a lot of redundant work visiting nodes that we've already seen. So I have the same idea of keeping track of a set that knows what we've previously visited, and the initial node gets enqueued just, kind of, by itself, just like Generation 0 gets put into the queue, and then while the queue is not empty, so while there's still neighbors I haven't yet visited, I dequeue the frontmost one, if it hasn't already been visited on some previous iteration, then I mark it as visited, and then I enqueue all of its children or neighbors at the back of the queue.

And so, as of right now, I'm processing Generation 0, that means all of Generation 1, so the neighbors that are one hop away, get placed in the queue, and then the next iteration coming out here will pull off a Generation 1, and then enqueue all of these two hop neighbors. Come back and other Generation 1's will get pulled off, enqueue all their two hop neighbors, and so the two hop generation will, kind of, be built up behind, and then once the last of the one hop generation gets managed, start processing the Generation 2 and so on.

And so this will keep going while the queue has some contents in there. So that means some neighbors we have connections to that we haven't yet had a chance to explore, and either we will find out that they have all been visited, so this will end when all of the nodes that are in the queue have been visited or there are no more neighbors to enqueue. So we've gotten to those dead ends or those cycles that mean we've reach everything that was reachable from that start node.

And so I've traced this guy to see it doing its work. So if I start again with A, and, again, assuming that I'm gonna process my nodes in alphabetical order when I have children, so I will through and I will enqueue, so the queue right now has just node A in it. I pull it off. I say have I visited? No, I have not. So then I mark it as visited, and now for each of its neighboring nodes, B, C, D, H, I'm gonna put those into the queue, and so then they're loaded up, and then I come back around, I say do I still have stuff in the queue? I do, so let's take out the first thing that was put in; it was B. Okay, and I'll say okay, well, where can you get to from B? You can get to C, so I'm gonna put C in the queue.

Now, C is actually already in the queue, but we're gonna pull it out earlier in terms it'll get handled by the visiting status later. So, okay, we'll put C, D, and E in and then put C behind it. So right now, we'll have C, D, H, and C again. It'll find the C that's in the front there. It'll say where can you get to from C? Nowhere, so no additional Generation 2's are added by that one. I'll look at D, and I'll say okay, well, what Generation 2's do we have from D? It says, well, I can get to E. All right, so put E on the back of the queue. Look at H; where can H get to? H can get to G, so go ahead and put G on the back of the queue.

So now, it comes back around to the things that are two hops away. The first one it's gonna pull off is C. It's gonna say, well, C is already visited, so there's no need to redundantly visit that or do anything with that. It passes over C. It finds the E that's behind that, and then it finds the G that was behind the H. The E also enqueued the F that was four hops, and the very last one, right, output will be that F that was only reachable by taking three hops away from the thing.

So this one's, kind of, working radially. The idea that I'm looking at all the things that I can get to, kind of, in one hop are my first generation, then two hops, then three hops, and so it's like throwing a stone in the water and watching it ripple out that the visiting is managed in terms of, kind of, growing length of path, and number of hops that it took to get there.

So this is the strategy that you used for maze solving, if you remember. That what you were tracking was the queue of paths where the path was AB, or ADC, or ADE, and that they grew step wise. That, kind of, each iteration through the traversal there where you're saying, okay, well, I've seen all the paths that are one hop. Now I've seen all the ones of two, now the ones three, and four, and five, and when you keep doing that, eventually you get to the hop, you know, 75, which leads you all the way to the goal, and since you have looked at all the paths that were 74 and shorter, the first path that you dequeue that leads to your goal must be the best or the shortest overall because of the order you processed them.

That's not true in depth-first search, right? That depth-first search might find a much longer and more circuitous path that led to the goal when there is a shorter one that would be eventually found if I had a graph that looked, you know, had this long path, let's say, and so this is goal node, let's say, and this is the start node that there could be a two hop path, kind of, off to this angle, but if this was the first one explored in the depth-first, right, it could eventually, kind of, work its way all the way around through this eight hop path, and say, okay, well, I found it, right?

It would be the first one we found, but there's no guarantee that would be the shortest in depth-first search, right? It would just happen to be the one that based on its arbitrary choice of which neighbors to pursue it happened to get to that there could be a shorter one that would be found later in the traversal. That's not true in the breadth-first search strategy because it will be looking at this, and then these two, and then these two, and, kind of, working its way outward down those edges. So as soon as we get to one that hits the goal, we know we have found the shortest we can have. Question?

**Student:**In that case, you could return, okay, we need two jumps to get to the goal, like, as far winning the goal.

**Instructor (Julie Zelenski)**:Yeah.

**Student:**How do we remember which path led us there?

**Instructor (Julie Zelenski)**:So if you were really doing that, you know, in terms of depth-first search, what you'd probably be tracking is what's the path? Probably a stack that said here's the stack that led to that path. Let me hold onto this, right, and I will compare it to these alternatives. So I'll try it on Neighbor 1, get the stack back from that; it's the best. Try it on Neighbor 2, get that stack back, see which one is smaller and use that as the better choice, and so, eventually, when I tried it out this side, I'd get the stack back that was just two, and I could say, yeah, that was better than this ten step path I had over here, so I will prefer that. So just using your standard, kind of, backtracking with using your return values, incorporating them, and deciding which was better.

And so in this case, the depth-first search can't really be pruned very easily because it has to, kind of, explore the whole thing to know that this short little path right over here happened to be just arbitrarily examined last, whereas, the breadth-first search actually does, kind of, because it's prioritizing by order of length. If the goal was shortest path, it will find it sooner and be able to stop when it found it rather than look at these longer paths. So it won't end up – if there were hundreds of other paths over here, breadth-first search won't actually have to ever reach them or explore them.

So there are a lot of things that actually end up being graph search in disguise. That's why a, kind of, a graph is a great data structure to, kind of, get to in 106b because there are lots of problems that, in the end, if you can model them using the same setup of nodes, and arcs, and traversals, that you can answer interesting questions about that data by applying the same graph search techniques. So as long as you want to know things like, well, which nodes are reachable from this node at all? And then that just means, okay, I'm here and I want to get to these places; what places could I get to?

Or knowing what courses I could take that would lead me to a CS degree, it's, kind of, like well, looking for the ones that lead to having all your graduation requirements satisfied, and each of those arcs could be a course you could take, right, how close did that get you to the goal? How can I get to the place I'd like to be, or what are the places that this course satisfies requirements that lead to? Knowing things about connectivity is, kind of, useful in things like the, kind of, bacon numbers or erdish numbers where people want to know, well, how close am I to greatness by – have I been in a movie with somebody who was in a movie, who was in a movie with Kevin Bacon? And, for me, it turns out zero; I have been in no movies.

But have you written a paper with someone who then connects through a chain to some other famous person, right? Tells you about your, kind of, social network distance or something, or I'm linked in. You're trying to get a new job, and you want to find somebody to introduce you to the head of this cool company. It's, like, figuring who you know who knows them, right? It's finding paths through a social network graph.

Finding out things like are there paths with cycles? What's the shortest path through the cycle, longest path through the cycle often tells you about things about redundancies in that structure that, for example, when you're building a network infrastructure, sometimes you do have cycles in it because you actually want to have multiple ways to get to things in case there's a breakdown. Like if there's a freeway accident, you want to have another way to get around it, but you are interested in, maybe, trying to keep your cost down is trying to find out where are the right places to put in those redundancies, those cycles that give you the best benefit with the least cost.

Trying to find things like a continuous path that visits all the nodes once and exactly once, especially doing so efficiently, picking a short overall path for that. We talked a little bit about that in terms of what's called the traveling salesman problem. You are trying to get from – you have ten cities to visit on your book tour. You don't want to spend all your lifetime on a plane. What's the path of crisscrossing the country that gets you to all ten of those with the least time painfully spent on an airline? And so those are just graph search problems. You have these nodes. You have these arcs. How can you traverse them and visit and come up with a good solution.

There's other problems I think that are kind of neat. Word letters, we used to give out an assignment on word letters. We didn't this time, but I thought it was an interesting problem to think about. The idea of how it is you can transform one letter at a time from one word to another is very much a graph problem behind the scenes, and that largely a breadth-first traversal is exactly what you're looking for there. How can I transform this with the shortest number of steps is, kind of, working radially out from the starting word.

Your boggle board turns out, really, is just a graph. If you think of having this sequence of letters that you're trying to work through, that really each of these letters can actually just be thought of as a node, and then the connections it has are to its eight neighbors, and that finding words in there, especially finding paths, starting from this node that lead away, that don't revisit, so, in fact, doing breadth-first or depth-first search. You most likely did depth-first search when you were writing the implementation of boggle, but that's, kind of, a deep search on I found an A. I found a P. I found a P, and, kind of, keeps going as long as that path looks viable.

So, in this case, that the arc information we're using is having a sub of those letters, do they form a prefix that could be leading somewhere good, and then the, kind of, recursion bottoming out when we have run ourselves into a corner where we have no unvisited nodes that neighbor us, or only nodes that would extend to a prefix that no longer forms any viable words as a search problem. So it feels like everything you've done this quarter has actually been a graph in disguise; I just didn't tell you. The maze problem is very much a graph search problem, building a graph and searching it.

Another one that I think is, kind of, neat to think about is, kind of, related to the idea of word letters is that often when you mistype a word, a word processor will suggest to you here's a word that you might have meant instead. You type a word that it doesn't have in its lexicon, and it says, well, that could just be a word I don't know, but it also could mean that you actually mistyped something, and so what you want to look at is having neighboring nodes, sort of, diagramming a graph of all the words you know of and then the, kind of, permutations, the tiny little twiddles to that word that are neighbors to it so that you could describe a set of suggestions as being those things that are within a certain hop distance from the original mistyped word.

And so you could say, yeah, things that have two letters different are probably close enough, but things that have three or more letters different aren't actually likely to be the mistyped word, and then that, kind of, leads to supporting things like wildcard searches where you're trying to remember whether it's E-R-A-T or A-R-T-E at the end of desperate. Those things could be modeled as, kind of, like they search through a graph space where you're modeling those letters, and then there's, like, a branching where you try all of the letters coming out of D-E-S-P, seeing if any of them lead to an R-A-T-E.

So the last thing I wanted to just bring up, and this is the one that's going into this week's assignment. I'm just gonna actually show it to you to, kind of, get you thinking about it because it's actually the problem at hand that you're gonna be solving is that the one of where I'm interested in finding the shortest path, but I'm gonna add in this new idea that all hops are not created equal. That a flight from San Francisco to Seattle is a shorter distance than the one that goes from San Francisco to Boston, and if I'm interested in getting to someplace, it's not just that I would like to take a direct flight. I'd also like to take the set of flights that involves the least, kind of, going out of my way, and going through other cities, and other directions.

And so in this case, a breadth-first search is a little bit too simplistic. That if I just look at all the places I can get to in one hop, those are not really equally weighted in terms of my goal of minimizing my total path distance, but if I, instead, prioritize – prioritize in your mind, immediately conjure up images of the priority queue, to look at paths that are, if I'm trying to minimize total distance, the shortest path, even if they represent several hops, if they're still shorter – if I have three flights, and each are 100 miles, then they represent a flight of 300 total miles, and that's still preferred to one that actually is 2,000 miles or 7,000 miles going to Asia alternatively.

And so if I worked on a breadth-first that's been modified by a total distance, that it would be a way to work out the shortest path in terms of total weight from San Francisco to my destination by a modified breadth-first, kind of, weighted strategy, and that is exactly what you're gonna be doing in this assignment. I'm not gonna give away too much by, kind of, delving too much in it, but the handout does a pretty good job of talking you through what you're doing, but it's a, kind of, neat way to think about that's how the GPS, the MapQuest, and things like that are working is you’re at a site; you have a goal, and you have this information that you’re trying to guide that search, and you’re gonna use heuristics about appears to be the, kind of, optimal path to, kind of, pick that, and go with it, and see where it leads you. So that will be your task for this week. What we’ll talk about on Friday is caching.

[End of Audio]

Duration: 53 minutes