

wk5

Proj2-wk1



# 0wk5-Overview

WELCOME: Welcome to Part 2 of the Algorithms Specialization: Graph Search, Shortest Paths, and Data Structures! Like the previous part, the course will have four weeks of lectures and assignments, followed by a final exam. Here are the highlights of the course's first week.

SUMMARY: Week 1 is all about graph search and its applications (section X). We'll cover a selection of fundamental primitives for reasoning about graphs. One very cool aspect of this material is that all of the algorithms that we'll cover are insanely fast (linear time with small constants); and, it can be quite subtle to understand why they work! The culmination of these lectures --- computing the strongly connected components of a directed graph with just two passes of depth-first search --- vividly illustrates the point that fast algorithms often require deep insight into the structure of the problem that you're solving. There are also lecture notes for this last topic (available for download underneath the videos). (If you're feeling REALLY motivated, you might read up on Tarjan's earlier algorithm for computing SCCs that needs only a single DFS pass!)

THE VIDEOS: We begin with an overview video, which gives some reasons why you should care about graph search, a general strategy for searching a graph without doing any redundant work, and a high-level introduction to the two most important search strategies, breadth-first search (BFS) and depth-first search (DFS). The second video discusses BFS in more detail, including applications to computing shortest paths and the connected components of an undirected graph. The third video drills down on DFS, and shows how to use it to compute a toplogical ordering of a directed acyclic graph (i.e., to sequence tasks in a way that respects precedence constraints). The fourth and fifth videos cover the description and analysis, respectively, of the aforementioned algorithm for computing SCCs. A final optional video --- hopefully one of the more fun ones in the course --- discusses the structure of the Web, and lightly touches on topics like Google's PageRank algorithm and the "six degrees of separation" phenomenon in social networks.

DISCUSSION FORUMS: The discussion forums play a crucial role in massive online courses like this one. If you have trouble understanding a lecture or completing an assignment, you should turn to the forums for help. After you've mastered the lectures and assignments for a given week, I hope you'll contribute to the forums and help out your fellow students. While I won't have time to carefully monitor the discussion forums, I'll check in and answer questions whenever I find the time.

VIDEOS AND SLIDES: Videos can be streamed or downloaded and watched offline (recommended for commutes, etc.). We are also providing PDF lecture slides (typed versions of what's written in the lecture videos), as well as subtitle files (in English and in some cases other languages as well). And if you find yourself wishing that I spoke more quickly or more slowly, note that you can adjust the video speed to accommodate your preferred pace.

THE HOMEWORK: Problem Set #1 should help you solidify your understanding of graph representations and graph search. The programming assignment asks you to implement the SCC algorithm from the lectures, and report your findings about the SCCs of a large graph. Programming Assignment #1 is the most difficult one of the course (and one of the more difficult ones in the entire specialization); as always, I encourage you to use the discussion forums to exchange ideas, tips, and test cases. If you can get through the first week of the course, it should be all downhill from there!

CLRS Chapter 22

DPV Chapter 3

KT Chapter 3, Section 3.5, 3.6

SW Chapter 4, Section 4.1,4.2

# 0wk5-Overview2

Course overview

Algorithms are the heart of computer science, and the subject has countless practical applications as well as intellectual depth. This specialization is an introduction to algorithms for learners with at least a little programming experience. The specialization is rigorous but emphasizes the big picture and conceptual understanding over low-level implementation and mathematical details. After completing this specialization, you will have a greater mastery of algorithms than almost anyone without a graduate degree in the subject. Specific topics in Part 2 of the specialization include: data structures (heaps, balanced search trees, hash tables, bloom filters), graph primitives (applications of breadth-first and depth-first search, connectivity, shortest paths), and their applications (ranging from deduplication to social network analysis).

Video Lectures

Lectures will be made available weekly, for a total of four weeks. In a given week, there will be roughly three hours of material. Some weeks include extra optional material (some review, some advanced topics).

Below each lecture video there is a PDF of typed versions of the slides.

You can download the videos of the lectures for offline viewing.

The video player supports speeding up and slowing down the pace.

Weekly Programming Assignments and Problem Sets

Every week there will be a new problem set and a new programming assignment.

For each problem set you are allowed a maximum of two attempts in a 12-hour period (we'll use the best score).

For each programming assignment you're allowed a maximum of 10 attempts in a 12-hour period (we'll use the best score).

For the final exam, you're allowed one attempt per 24 hours.

Grading

To pass a problem set, you must get at least 4 of the 5 questions correct (80%).

To pass a programming assignment, you must get all of the answers correct (100%).

To pass the final exam, you must get at least 70% of the total points (7 out of 10).

To pass the course, you must pass all of the problem sets, all of the programming assignments, and the final exam.

Theory Problems

These are totally optional theory questions (no deadlines or credit given).

We encourage you to attempt these questions and discuss them in the forums to develop a deeper understanding of the design and analysis of algorithms.

# 10-1-Graph Search - Overview

So let's talk about the absolutely fundamental problem of searching a graph, and the very related problem of finding paths through graphs. So why would one be interested in searching a graph, or figuring out if there's a path from point A to point B? Well there's many, many reasons. I'm going to give you a highly non-exhaustive list on this slide. >>So let me begin with a very sorta obvious and literal example, which is if you have a physical network, then often you want to make sure that the network is fully connected in the sense that you can get from any starting point to any other point. So, for example, think back to the phone network. It would've been a disaster if callers from California could only reach callers in Nevada, but not their family members in Utah. So obviously a minimal condition for functionality in something like a phone network is that you can get from any one place to any other place, similarly for road networks within a given country, and so on. It can also be fun to think about other non-physical networks and ask if they're connected. So one network that's fun to play around with is the movie network. So this is the graph where the nodes correspond to actors and actresses, and you have an edge between two nodes, if they played a role in a common movie. So this is going to be an undirected graph, where the edges correspond to, not necessarily co-starring, but both the actors appearing at least some point in the same movie. So versions of this movie network you should be able to find publicly available on the web, and there's lots of fun questions you can ask about the movie network. Like, for example, what's the minimum number of hops, where a hop here again is the movie that two people both played a role in? The minimum number of hops or edges from one actor to another actor, so perhaps the most famous statistic that's been thought about with the movie is the Bacon Number. So this refers to the fairly ubiquitous actor Kevin Bacon, and the question the Bacon Number of an actor is defined as the minimum number of hops you need in this movie graph to get to Kevin Bacon. So, for example, you could ask about Jon Hamm, also known as Don Draper from "Mad Men". And you could ask how many edges do you need on a path through the movie graph to get to Kevin Bacon? And it turns out that the answer is 1, excuse me, 2 edges. You need one intermediate point, namely Colin Firth. And that's became, that's because Colin Firth and Kevin Bacon both starred in Atom Egoyan's movie, "Where "the Truth Lies", and Jon Hamm and Colin Firth were both in the movie "A Single Man". So that would give Jon Hamm a Bacon Number of 2. So, these are the kind of questions you're gonna ask about connectivity. Not just physical networks, like telephone and telecommunication networks, but also logical networks about parallel relationships between objects in general. So the Bacon Number is fundamentally not just about any path, but actually shortest paths, the minimum number of edges you need to traverse to get from one actor to Kevin Bacon. And shortest paths are also have a very practical use, that you might use yourself in the driving directions. So when you use a website or a phone app and you ask for the best way to get from where you are now to say some restaurant where you're gonna have dinner, obviously you're trying to find some kind of path through a network through a graph, and indeed often you want the, the shortest path, perhaps in mileage or perhaps in anticipated travel time. Now I realize that when you are thinking about paths and graphs, it's natural to focus on sort of very literal paths and quite literal physical networks. Things like routes through a road network or paths through the internet and so on. You should really think more abstractly as a path as just a sequence of decisions, taking you from some initial state to some final state. And it's this abstract mentality which is what makes graph search so ubiquitous, it feels like artificial intelligence, where you want to formulate a plan of how to get from an initial state to some goal state. So, to give a simple recreational example, you can imagine just trying to understand how to compute automatically a way to fill in a Sudoku puzzle so that you get to, so that you solve the puzzle correctly. So you might ask, you know, what is the graph that we're talking about, when we wanna solve a Sudoku puzzle. Well this is gonna be a directed graph, where here the nodes correspond to partially completed puzzles. So, for example, at one node of this extremely large graph, perhaps 40 out of the 81 cells are filled in with some kind of number, and now, again, remember a path is supposed to correspond to a sequence of decisions. So, what are the actions that you take in solving Sudoku? Well, you fill in a number into a square. So, an edge which here is going to be directed, is going to move you from one partially completed puzzle to another, where one previously empty square gets filled in with one number. And of course then the path is that you're interested in computing, or what your searching for when you search this graph. You begin with the initial state of the Sudoku puzzle and you want to reach some goal state where the Sudoku puzzle is completely filled in without any violations of the rules of Sudoku. And of course it's easy to imagine millions of other situations where you wanna formulate some kind of plan like this, for example if you have a robotic hand and you wanna grasp some object, you need to think about exactly how to approach the object with this robotic hand, so that you can grab it without, for example, first knocking it over, and you can think of millions of other examples. Another thing which turns out to be closely related to graph search, as we'll see, it has many applications in its own right, is that of computing connectivity information about graphs, in particular the connected components. So this, especially for undirected graphs, corresponds to the pieces of the graph. We'll talk about these topics in their own right, and I'll give you applications for them later. So for undirected graphs I'll briefly mention an easy clustering heuristic you can derive out of computing connected components. For directed graphs where the very definition of computing components is a bit more subtle, I'll show you applications to understanding the structure of the web. So these are a few of the reasons why it's important for you understand how to efficiently search graphs. It's a, a fundamental and widely applicable graph primitive. And I'm happy to report that in this section of the course, pretty much anything, any questions we wanna answer about graph search, computing connected components, and so on, there's gonna be really fast algorithms to do it. So, this will be the part of the course where there's lots of what I call for free primitives, processing steps, subroutines you can run without even thinking about it. All of these algorithms we're gonna discuss in the next several lectures, are gonna run in linear time, and they're gonna have quite reasonable constants. So, they're really barely slower than reading the input. So, if you have a graph and you're trying to reason about it and you're trying to make sense about it, you should in some sense feel free to apply any of these subroutines we're gonna discuss to try and glean some more information about what they look like, how you might use the network data. There's a lot of different approaches to systematically searching a graph. So, there's many methods. In this class we're gonna focus on two very important ones, mainly breadth first search and depth first search. But all of the graph search methods share some things in common. So, in this slide let me just tell you the high order bits of really any graph search algorithm. So graph search subroutines generally are passed as input a starting search vertex from which the search originates. So that's often called source vertex. And your goal then is to find everything findable from the search vertex and obviously you're not gonna find anything that you can't find that is not findable. What I mean by findable, I mean, there's a path from the starting point to this other node. So any other node to which you can get along on a path from the starting point you should discover. So, for example, if you're given an undirected graph that has three different pieces, like this one I'm drawing on the right, then perhaps S is this left most node here, then the findable vertices starting from S, i.e. the ones which you can reach from a path to S, is clearly precisely these four vertices. So, you would want graph search to automatically discover and efficiently discover these four vertices if you started at S. You can also think about a directed version of the exact same graph, where I'm gonna direct the vertices like so. So now the definition of the findable nodes is a little bit different. We're only expecting to follow arcs forward, along the forward direction. So we should only expect at best to find all of the nodes that you can reach, by following a succession of forward arcs, that is, any node that there's a path to from S. So in this case, these three nodes would be the ones we'd be hoping to find. This blue node to the right, we would no longer expect to find, because the only way to get there from S, is by going backward along arcs. And that's not what we're going to be thinking about in our graph searches. So we want to find everything findable, i.e. that we can get to along paths, and we want to do it efficiently. Efficiently means we don't want to explore anything twice. Right, so the graph has m arcs, m edges and n nodes or n vertices and really we wanna just look at either each piece of the graph only once for a small cost number of times. So looking for running time which is linear on the size of the graph that is big O of m plus n. Now when we were talking about representing graphs, I said that in many applications, it's natural to focus on connected graphs, in which case M is gonna dominate N, and you're gonna have at least as many edges as nodes, essentially. But connectivity is the classic case where you might have the number of edges of being much smaller than the number of nodes. There might be many pieces of the whole point of what you're trying to do is discover them. So, for this sequence of lectures where we talk about graph search and connectivity, we will usually write M plus N. We'll think that either one can be bigger or smaller than the other. So let me now give you a generic approach to graph search. It's gonna be under-specified, there'll be many different ways to instantiate it. Two particular instantiations will give us breadth first search and depth first search but here is just a general systematic method to finding everything findable without exploring anything more than once. So motivated by the second goal, the fact that we don't want to explore anything twice, with each node, with each vertex, we're gonna remember whether or not we explored it before. So we just need one Boolean per node and we will initialize it by having everything unexplored except S, our starting point we'll have it start off as explored. And it's useful to think of the nodes thus far as being in some sense territory conquered by the algorithm. And then there's going to be a frontier in between the conquered and unconquered territory. And the goal of the generic outcome is that each step we supplement the conquered territory by one new node, assuming that there is one adjacent to the territory you've already conquered. So for example in this top example with the undirected network, initially the only thing we've explored is the starting point S. So that's sort of our home base. That's all that we have conquered so far. And then in our main while loop, which we iterate as many times as we can until we don't have any edges meeting the following criterion, we look for an edge with one end point that we've already explored. One end point inside the conquered territory and then the other end point outside. So this is how we can in one hop supplement the number of nodes we've seen by one new one. If we can't find such an edge then this is where the search stops. If we can find such an edge, well then we suck V into the conquered territory. We think of it being explored. And we return to the main while loop. So, for example, in this example on the right, we start with the only explored node being S. Now, there's actually two edges that cross the frontier, in the sense one of the endpoints is explored, namely one of the endpoints is S, and the other one is some other vertex. Right? There's this there's these two vert, two edges to the left, two vertices adjacent to S. So, in this algorithm we pick either one. It's un, under-specified which one we pick. Maybe we pick the top one. And then all of the sudden, this second top vertex is also explored so the conquered territory is a union of them, and so now we have a new frontier. So now again we have two edges that cross from the explored nodes to the unexplored nodes. These are the edges that are in some sense going from northwest to southeast. Again, we pick one of them. It's not clear how. The algorithm doesn't tell us, we just pick any of them. So, maybe for example we pick this right most edge crossing the frontier. Now the right most edge of these-- right most vertex of these four is explored so our conquered territory is the top three vertices. And now again we have two edges crossing the frontier. The two edges that are incident to the bottom node, we pick one of them, not clear which one, maybe this one. And now the bottom node is also explored. And now there are no edges crossing the frontier. So there are no edges who, on the one hand, have one end-point being explored, and the other end-point being unexplored. So these will be the four vertices, as one would hope, that the search will explore started from S. Well generally the claim is that this generic graph search algorithm does what we wanted. It finds everything findable from the starting point and moreover it doesn't explore anything twice. I think that's fairly clear that it doesn't explore anything twice. Right? As soon as you look at a node for the first time, you suck it into the conquered territory never to look at it again. Similarly as soon as you look at an edge, you suck them in. But when we explore breadth and depth first search, I'll be more precise about the running time and exactly what I mean by you don't explore something twice. So, at this level of generality, I just wanna focus on the first point, that any way you instantiate this search algorithm, it's going to find everything findable. So, what do I really mean by that? The formal claim is that at the termination of this algorithm, the nodes that we've marked exp-, explored, are precisely the ones that can be reached via a path from S. That's the sense in which the algorithm explores everything that could potentially be findable from the starting point S. And one thing I wanna mention is that this claim and the proof I'm going to give of it, it holds whether or not G is an undirected graph or a directed graph. In fact, almost all of the things that I'm gonna say about graph search, and I'm talking about breadth first search and depth first search, work in essentially the same way, either in undirected graphs or directed graphs. The obvious difference being in an undirected graph you can traverse an edge in either direction. In a directed graph, we're only supposed to traverse it in a forward direction from the tail to the head. The one big difference between undirected and directed graphs is when we connectivity computations and I'll remind you when we get to that point which one we're talking about. Okay? But for the most part, when we just talk about basic graph search it works essentially the same way whether it's undirected or directed. So keep that in mind. Alright, so why is this true? Why are the nodes that get explored precisely the nodes for which there's a path to them from S? Well, one direction is easy. Which is, you can't find anything which is not findable, that is, if you wind up exploring a node, the only reason that can happen is because you traversed a sequence of edges that got you there. And that sequence of edges obviously defines a path from S to V. If you really want to be pedantic about the forward direction that explored nodes have to have paths from S. Then you can just do an easy induction. And I'll leave this for you to check, if you want, in the privacy of your own home. So the important direction of this claim is really the opposite. Why is it that no matter how we instantiate this generic graph search procedure, it's impossible for us to miss anything. That's the crucial point, we don't miss anything that we could, in principle, find via a path. But we're gonna proceed by contradiction. So, what does that mean, we're going to assume that, the statement that we want to prove is true, is not true. Which means that, it's possible that, G has a path from s to v and yet, somehow our algorithm misses it, doesn't mark it as explored. Alright, that's the thing we're really hoping doesn't happen. So let's suppose it does happen and then derive a contradiction. So suppose G does have a path from s to some vertex v. Call the path P. I'm gonna draw the picture for an undirected graph but the situation would be same in the, in the directed case. So there's a bunch of hops, there's a bunch of edges and then eventually this path ends at v. Now the bad situation, the situation from which we want to derive a contradiction is that V is unexplored at the end of this algorithm. So let's take stock of what we know. S for sure is explored, right. We initialized this search procedure so that S is marked as explored. V by hypothesis in this proof by contradiction is unexplored. So S is explored, V is unexplored. So now imagine we, just in our heads as a thought experiment which traverse this path P. We start at S and we know it's explored. We go the next vertex, it may or may not have been explored, we're not sure. We go to the third vertex, again who knows. Might be explored, might be unexplored and so on, but by time we get to V, we know it's unexplored. So we start at S, it's been explored, we get to V it's been unexplored. So at some point there's some hop, along this path P, from which we move from an explored vertex, to an unexplored vertex. There has to be a switch, at some point, cuz the end of the day at the end of the path we're at an unexplored node. So consider the first edge, and there must be one that we switch from being at an explore node to being at an unexplored node. So, I'm going to call the end points of this purported edge U and W. Where U is the explored one and W is the unexplored one. Now, for all we know U could be the same as S, that's a possibility, or for all we know, W could be same as V. That's also a possibility. In the picture, I'll draw it as if this edge UX was somewhere in the middle of this path. But, again it may be at one of the ends. That's totally fine. But now in this case, there's something I need you to explain to me. How is it possible that, on the one hand, our algorithm terminated. And on the other hand, there's this edge U comma X. Where U has been explored and X has not been explored. That, my friends, is impossible. Our generic search algorithm does not give up. It does not terminate, unless there are no edges where the one end point is explored and the other end point is unexplored. As long as there's such an edge, it has, is gonna suck in that unexplored vertex into the conquered territory, it's gonna keep going. So the upshot is there's no way that our algorithm terminated with this picture. With there being an edge U X, U explored, X unexplored. So, that's the contradiction. This contradicts the fact that our algorithm terminated with V unexplored. So that is a general approach to graph search. So that I hope gives you the flavor of how this is going to work. But now there's two particular instantiations of this generic method that are really important and have their own suites of applications. So we're gonna focus on breadth-first search and depth-first search. We'll cover them in detail in the next couple of videos. I wanna give you the highlights to conclude this video. Now let me just make sure it's clear where the ambiguity in our generic method is. Why we can have different instantiations of it that potentially have different properties and different applications. The question is at a given iteration of this while loop, what do you got? You've got your nodes that you've already explored, so that includes S plus probably some other stuff, and then you've got your nodes that are unexplored, and then you have your crossing edges. Right? So, there are edges with one point in each side. And for an undirected graph, there's no orientation to worry about. These crossing edges just have one endpoint on the explored side, one endpoint on the unexplored side. In the directed case, you focus on edges where the tail of the edge is on the explored side and the head of the edge is on the unexplored side. So, they go from the explored side to the unexplored side. And the question is, in general, in an iteration of this while loop there's gonna be many such crossing edges. There are many different unexplored nodes we could go to next, and different strategies for picking the unexplored node to explore next leads us to different graph search algorithms with different properties. So the first specific search strategy we're gonna study is breadth first search, colloquially known as BFS. So let me tell you sort of the high level idea and applications of bread first search. So, the goal is going to be to explore the nodes in what I call, layers. So, the starting point S will be in its own layer, Layer-0. The neighbors of S will constitute Layer-1, and then Layer-2 will be the nodes that are neighbors of Layer-1 but that are not already in layer zero or layer one, and so on. So layer i plus one, is the stuff next to layer i that you haven't already seen yet. You can think of this as a fairly cautious and tentative exploration of the graph. And it's gonna turn out that there's a close correspondence between these layers and shortest path distances. So if you wanna know the minimal number of hops, the minimal number of edges you need in a path to get from point A to point B in a graph. The way we wanted to know the fewest number of edges in the movie graph necessary to connect to John Hamm to Kevin Bacon. That corresponds directly to these layers. So if a node is in layer i, then you need i edges to get from S to i in the graph. Once we discuss breadth-first search, we'll also discuss how to compute the connected components, or the different pieces, of an undirected graph. Turns out this isn't that special to breadth-first search, you can use any number of graph search strategies to compute connected components in undirected graphs. But I'll show you how to do it using a simple looped version of breadth-first search. And we'll be able to do this stuff in the linear time that we want. The very ambitious goal of getting linear time. To get the linear time implementation, you do wanna use the right data structure, but it's a simple, simple data structure, something probably you've seen in the past. Namely a queue. So, something's that first in and first out. So, the second search strategy that's super important to know is depth first search, also known as DFS to its friends. Depth first search has a rather different feel than breadth first search. It's a much more aggressive search where you immediately try to plunge as deep as you can. It's very much in the spirit of how you might explore a maze, where you go as deeply as you can only backtracking when absolutely necessary. Depth first search will also have its own set of applications. It's not, for example, very useful for computing shortest path information, but especially in directed graphs it's going to do some remarkable things for us. So, in directed acyclic graphs, so a directed graph with no directed cycles it will give us what's called the topological ordering. So it'll sequence the nodes in a linear ordering from the first to the last, so that all of the arcs of the directed graph go forward. So this is useful for example if you have a number of tasks that need to get completed with certain precedence constraints. Like for example you have to take all of the classes in your undergraduate major, and there was certain prerequisites, topological ordering will give you a way in which to do it, respecting all of the prerequisites. And finally where for undirected graphs it doesn't really matter whether you use BFS or DFS to connect the components, in the directed graphs where even defining connected components is a little tricky it turns out depth first search is exactly what you want. That's what you're going to get a linear time implementation for computing the right notion of connected components in the directed graph case. Time-wise, both of these are superb strategies for exploring a graph. They're both linear time with very good constants. So depth-first search again, we're gonna get O of M plus N time in a graph with M edges and N vertices. You do wanna use a different data structure reflecting the different search strategy. So, here because you're exploring aggressively, as soon as you get to a node you'll meet and you start exploring its neighbors, you wanna last-in first-out data structure, also known as a stack. Depth first search also admits a very elegant recursive formulation, and in that formulation, you don't even need to maintain a stack data structure explicitly, the stack is implicitly taken care of in the recursion. So that concludes this overview of graph search. Both what it is, what our goals are, what kind of applications they have and two of the most common strategies. The next couple videos are going to explore these search strategies, as well as a couple of these applications in greater depth.

# 10-2-Breadth-First Search (BFS) - The Basics

So in this lecture we're going to drill down into our first specific, search strategy for graphs and also explore some applications. Namely, breadth first search. So let me remind you the intuition and applications of breath first search. The plan is to systematically explore the nodes of this graph beginning with the given starting vertex in layers. So let's think about the following example graph. Where S is the starting point for our breadth first search. So to start vertex S will constitute the first layer. So we'll call that L zero. And then the neighbors of S are going to be the first layer. And so those are the vertices that we explore just after S. So those are L one. Now the second layer is going to be the vertices that are neighboring vertices of L one but are not themselves in L one or for that matter L zero. So that's going to be C and D. That's going to be the second layer. Now you'll notice for example S is itself a neighbor of these nodes in layer one, but we've already counted that in a previous layer so we don't count S toward L two. And then finally the neighbors of L two, which are not already put in some layer is E. That will be layer three. Again notice C and D are neighbors of each other but, they've already been classified in layer two. So, that's where they belong, not in layer three. So that's the high level picture of breadth first search you should have. We'll talk about how to actually precisely implement it on the next slide. Again just a couple other things that you can do with breadth first search which we'll explore in this video is computing shortest paths. So it turns out shortest path distances correspond precisely to these layers. So, for example if you had that as S, you had that as the Kevin Bacon node in the movie graph, then Jon Hamm would pop up in the second layer from the breadth first search from Kevin Bacon. I'm also going to show you how to compute the connected components of an undirected graph. That is to compute its pieces. We'll do that in linear time. And for this entire sequence of videos on graph primitives, we will be satisfied with nothing less than the holy grail of linear time. And again, remember in a graph you have two different size parameters, the number of edges M and the number of nodes N. For these videos I'm not going to assume any relationship between M and N. Either one could be bigger. So linear time's gonna mean O of M plus N. So let's talk about how you'd actually implement breadth first search in linear time. So the sub routine is given as input both the graph G. I'm gonna explain this as if it's undirected, but this entire procedure will work in exactly the same way for a directed graph. Again, obviously in an undirected graph you can traverse an edge in either direction. For a directed graph, you have to be careful only to traverse arcs in the intended direction from the tail to the head, that is traverse them forward. So as we discussed when we talked about just generic strategies for graph search, we don't want to explore anything twice, that would certainly be inefficient. So we're going to keep a boolean at each node, marking whether we've already explored it or not. And by default, I'm just, we're just going to assume that nodes are unexplored. They're only explored if we explicitly mark them as such. So we're going to initialize the search with the starting vertex S. So we mark S as explored and then we're gonna put that in what I was previously calling conquered territory the nodes we have already started to explore. So to get linear time we are gonna have to manage those in a slightly non naive but, but pretty straightforward way namely via a queue, which is a first in first out data structure that I assume you have seen. If you have never seen a queue before, please look it up in a programming textbook or on the web. Basically a queue is just something where you can add stuff to the back in constant time and you can take stuff from the front in constant time. You can implement these, for example, using a doubly linked list. Now recall that in the general systematic approach to graph search, the trick was to, in each iteration of some while loop, to add one new vertex to the conquered territory. To identify one unexplored node that is now going to be explored. So that while loop's gonna translate into one in which we just check if the queue is non-empty. So we're assuming that the queue data structure supports that query in constant time which is easy to implement. And if the queue is not empty we remove a node from it. And because it's a queue, removing nodes from the front is what you can do in constant time. So call the node that you get out of the queue V. So, now we're going to look at V's neighbors, vertices with which it shares edges, and we're gonna see if any of them have not already been explored. So, if W's something we haven't seen before, that's unexplored, that means it's in the unconquered territory, which is great. So, we have a new victim. We can mark W as explored. We can put it in our queue and we've advanced the frontier and now we have one more explored node than we did previously. And again, a queue by construction, it supports adding constant time additions at the end of the queue, so it's where we put W. So, let's see how this code actually executes in this same graph that we were looking at in the previous slide. And what I'm gonna do is I'm gonna number the nodes in the order in which they are explored. So, obviously the first node to get explored is S. That's where the queue starts. So now, when we follow the code, what happens? Well in the first iteration of the while loop we ask is the queue empty? No it's not, because S is in it. So we remove in this case the only node of the queue. It's S. And then we iterate over the edges incident to S. Now there are two of them. There's the edge between S and A and there's the edge between S and B. And again this is still a little under specified. In the sense that the algorithm doesn't tell us which of those two edges we should look at. Turns out it doesn't matter. Each of those is a valid execution of breadth first search. But for concreteness, let's suppose that of the two possible edges, we look at the edge S comma A. So, then we ask, has A already been explored? No, it hasn't. This is the first time we've seen it, so we say, oh goody. This is sort of new grist for the mill. So, we can add A to the queue at the end and we mark W as, sorry mark A as explored. So, A is gonna be the second vertex that we mark. So, after marking A as explored and adding it to the queue, so now we go back to the for loop, and so now we move on to the second edge. It's into S, that's the edge between S and B. So, we ask, have we already explored B? Nope, this is the first time we've seen it. So, now we have the same thing with B. So, B gets marked as explored and gets added to the queue at the end. So the queue at this juncture has first a record for A, cause that was the first one we put in it after we took S out. And then B follows A in the queue. Again, depending on the execution this could go either way. But for concreteness, I've done it so that A got added before B. So at this point, this is what the queue looks like. So now we go back up to the while loop, we say is the queue empty? Certainly not. There's actually two elements. Now we remove the first node from queue, in this case, that's the node A that was the one we put in before the node B. And so now we say, well, let's look at all the edges incident to A. And in this case A has two two incident edges. It has one that it shares with S and it has one that it shares with C. And so, if we look at the edge between A and S, then we'd be asking an if statement. Has S already been explored? Yes it has, that's where we started. So, there's no reason to do anything with S. That's already been taken out of the queue. So, in this for loop for A, there's two iterations. One involves the edge with S, and that one we completely ignore. But then there's the other edge that A shares with C, and C we haven't seen yet. So, at that part of the for loop, we say ahah. C is a new thing, new node we can mark as explored and put in the queue. So, that's gonna be our number four. So now how has the queue changed. Well, we got rid of A. And so now B is in the front and we added C at the end. And so now the same thing happens. We go back to the while loop, the queue is not empty, we take off the first vertex, in this case that's gonna be B. B has three incident edges, it has one incident S but that's irrelevant, we've already seen S. It has one incident to C, that's also irrelevant, that's also irrelevant, because we've already seen C. True, we just saw it very recently, but we've already seen it. But the edge between B and D is new, and so that means we can take the node D, mark it as explored and add it to the queue. So D is going to be the fifth one that we see. And now the queue has the element C followed by D. So now we go back to the while loop and we take C off of the queue. It again has four now edges. The one with A is irrelevant, we've already seen A. The one with B is irrelevant, we've already seen B. The one with D is irrelevant, we've already seen D. But we haven't seen E yet. So, when we get to the part of the for loop, or the edge between C and E, we say, aha, E is new. So E will be the sixth and final vertex to be marked as explored. And that will get added at the end of the queue. So then in the final two iterations of the while loop the D is going to be removed, we'll iterate through its three edges, none of those will be relevant because we've seen the three endpoints. And then we'll go back to the while loop and we'll get rid of the E. E is irrelevant cause it has two edges we've already seen the other endpoints. Now we go back to the while loop. The queue is empty. And we stop. That is breadth-first search. And to see how this simulates the notion of the layers that we were discussing in the previous slide notice that the nodes are numbered according to the layer that they're in, so S was layer zero. And then the two nodes that S caused to get added to the queue, the A and the B, are number two and three, and the edges of layer three are precisely the ones, sorry the edges of layer two are precisely the ones that got added to the queue, while we were processing the nodes from layer one. That is, C and D are precisely the nodes that got added to the queue while we were processing A and B. So, this is level zero, level one, and level two. E is the only node that got added to the queue while we were processing level, layer two. The vertices C and D. So E will be the third layer. So, in that sense, by using a first in first out data structure, this queue, we do wind up kinda processing the nodes according to the layers that we discussed earlier. So, the claim that breadth first search is a good way to explore a graph, in the sense that it meets the two high level goals that we delineated in the previous video. First of all it finds everything findable, and obviously nothing else, and second of all, it does it without redundancy. It does it without exploring anything twice, which is the key to its linear time implementation. So a little bit more formally, claim number one. At the end of the algorithm, the vertices that we've explored are precisely the ones such that there was a path from S to that vertex. Again this claim is equally valid, whether you're running BFS in an undirected graph or a directed graph. Of course in an undirected graph, meaning an undirected path from S to V, whereas a directed graph in a directed path from S to V. That means a path where every arc in the path gets traversed in the forward direction. So, why is this true? Well, this is true, we basically proved this more generally for any graph search strategy of a certain form of which breadth first search is one. If it's hard for you to see the right way to interpret breadth-first search as a special case of our generic search algorithm, you can also just look at our proof for the generic search algorithm and copy it down for breadth-first search. So it's clear that you're only gonna, again, the forward direction of this claim is clear. If you actually find something, if something's marked as explored, it's only because you found a sequence of edges that led you there. So the only way you mark something as explored is if there's a path from S to V. Conversely, to prove that anything with an S to V, for with a path from V will be found, you can proceed by contradiction: you can look at the part of the path from S to V that, that BFS does successfully explore, and then you gotta ask, why didn't it go one more hop? It never would've terminated before reaching all the way to V. So, you can also just copy that same proof that we had for the generic search strategy in the previous video. Okay? So, again, the upshot. Breadth first search finds everything you'd wanna find. Okay? So, it only traverses paths, so you're not gonna find anything where there isn't a path to it. But it never misses out. Okay? Anything where there's a path, BFS, guaranteed to find it. No problem. Claim number two is that the running time is exactly what we want and I am gonna state it in a form that will be useful later when we talk about connected components. So the running time of the main while loop, ignoring any kind of pre processing or initialization is proportional to what I am gonna call NS and MS which is the number of nodes that can be reached from S and number of edges that can be reached from S. And the reason for this claim it just becomes clear if you inspect the code which we'll do in a second. So let's return to the code and just tally up all the work that gets done. So I'm gonna ignore this initialization. I'm just gonna focus on the main while loop. So we can summarize the total work done in this while loop as follows. First we just think about the vertices so in this search we're only gonna deal, ever deal with the vertices that are findable from S, so that's NS. And what do we do for the given node, well we insert it into the queue and we delete it from the queue. Alright? So we're never gonna deal with a single node more than once. So that's constant time overhead per vertex that we ever see, so that's the proportion of the NS part. Now, a given edge, we might look at it twice. So, for an edge V W, we might consider it once when we first look at the vertex V, and we might consider it again when we look at the vertex W. Each time we look at an edge we do constant work. So that means we're only gonna do constant work per edge. Okay. So we look at each vertex at most once. We look at each edge findable from S at most twice. We do constant time, constant work when we look at something. So the overall running time is going to be proportional to the number of vertices findable from S plus the number of edges findable from S. So, that's really cool. We have a linear time of implementation of a really nice graph search strategy. Moreover we just need very basic data structures, a queue, to make it run fast with small constants. But it gets even better. We can use breadth first search as a work horse for some interesting applications. So, that's what we'll talk about in the rest of this video.

# 10-3-BFS and Shortest Paths

And let's begin with the idea of shortest paths. So, again I'll give you the movie graph. I'll give you Kevin Bacon as a starting point. What's the fewest number of hops, the fewest number of edges on a path that leads to, say, Jon Hamm? So some notation, I'm going to use DIST of V, to denote this shortest path distance. So with respect to a starting node S, the fewest number of hops or the fewest number of edges on a path that starts at S, and goes to V. And again you can define this in the same way for undirected graphs or directed graphs. In a directed graph, you always want to traverse arcs in the forward direction, in the correct direction. And to do this we just have to add a very small amount of extra code to the BFS code that I showed you earlier. It's just gonna be a very small constant overhead, and basically it just keeps track of what layer each node belongs to, and the layers are exactly tracking shortest path distances away from the starting point S. So what's the extra code. Well first in the initialization step, you set your preliminary estimate of the distance, the number of the shortest path distance from S to vertex V as well if V equals S, you know you can get from S to S on a path of length zero, the empty path. And if it's any other vertex all bets are off, you have no idea if there's a path to V at all. So let's just initially put plus infinity for all vertices other than the starting point. This is something we will of course revise once we actually discover a path to vertex V. And the only other extra code you have to add is, when you're considering, so when you take a vertex off of the front of the queue and then you iterate through its edges and you're considering one of those edges V, W, so your V would be the vertex that you just removed from the front of the queue. And as usual if the other end of the edge W has already been dealt with then, you know, you just throw it out. That would be redundant work to look at it again. But if this is the first time you've seen the vertex W. Then, in addition to what we did previously, in addition to marking it as explored and putting it in the queue at the back, we also compute its distance, and its distance is just going to be one more than the distance of the vertex V, responsible for W's addition to the queue, responsible for first discovering this vertex W. So, returning to our running example of breadth first search, let's see what happens. So, again, remember the way this worked is we start out with from the vertex S, and we set the distance, you know in our initialization equal to zero. We don't know what the distance is of anything else. So, then how did breadth first search work? So, we, in the initial step we put S in the queue. We go to the main while loop, and then the queue's not empty. We extract S from the queue. We look at its neighbors. Those neighbors are A and B. We handle them in some order. Let's again think of that we first handle the edge between S and A. So, then what do we do? We say we haven't seen A yet. So we mark A as explored. We put A in the queue at the front, and now we have this extra step. It's the first time we're seeing A, so we wanna compute its distance. And we compute its distance as one more than the vertex responsible for discovering A. And so in this case S was the vertex whose exploration unveiled the existence of the vertex A to us. S's distance is zero so we set A's distance to one. And that's tantamount to being a member of the ith layer. So what happens in the next iteration of the while loop. So now the queue contains Sorry, the next iteration of the for loop, excuse me. So after we've handled the edge S comma A, we're still dealing with S's edges, now we handle the edge S comma B. We put, it is the first time we've seen B. We put B at the end of the queue, we mark it as explored, and then we also execute this new step. We set B's distance to one more than the vertex responsible for discovering it. That would again be the vertex S. S led to B's discovery. And so we set B's distance to be one more than S's distance, also known as one. And that corresponds to being the other node in layer one. Now having handled all of S's adjacent arcs we go back to the while loop. We ask if the queue is empty. Certainly not. It takes two vertices, first A then B. We extract the first vertex cuz it's FIFO, that would be the vertex A. Now we look at A's incident edges. There's S comma A, which we ignore. There's A comma C. This is the first time we've seen C. So as before we mark C as explored. We add C to the end of the queue and now again we have this additional line. We set C's distance to be one more than the vertex responsible for its discovery. In this case it's A. That first discovered C. So we're gonna set C's distance to be one more than A's distance also known as two. So then having handled A we move on to the next vertex in the queue, which in this case is B. Again we can forget about the edge between S and V. We've already seen S, we can forget about the edge between B and C. We've already seen C but D is now discovered for the first time via B. It gets more as explored, it goes to the end of the queue and its distance is set equal to one more than B's distances which is two. So, then we deal with C. Again it has four arcs, four edges, three of them are irrelevant. The one to E is not irrelevant, cause this is the first time we've seen E. So, E's distance is computed as one more than C, cause C was the one who first found E, and so E gets a distance of three, and then the rest of the algorithm proceeds as before. And you will notice that the labelings, the shortest path labels, are exactly the layers as promised. I hope you find it very easy to believe at this point that, that claim is true in general. That the distance computed by breadth-first search for an arbitrary vertex V, that's reachable from S is, that's gonna be equal to i if and only if V is in the ith layer as we've been defining it previously. And what does it really mean to be in the ith layer? It means that the shortest path distance between V and S has i hops, i edges. So I don't wanna spend time giving a super rigorous proof of this claim but let me just give you the gist, the basic idea, and I encourage you to produce some formal proof at home if that is something that interests you. So one way to do it is you can do it by induction on the layer i. And so what you want to prove is that all of the nodes that belong to a given layer i do Indeed, breadth first search does indeed compute the distance of i for them. So what does it mean to be a node in layer i? Well, first of all, you can't have been seen in either of the, any of the previous layers; you weren't a member of layer zero through i minus one. And furthermore, you're a, a neighbor of somebody who's in layer i minus one. Right? You're seen for the first time once all of the layer i minus one nodes are processed. So the inductive hypothesis tells you that distances were correctly computed for everybody from the lower l, from the lower layers. So in particular, whoever this node V was from layer i minus one was responsible for discovering u, in layer i. It has a distance computed as i minus one. Yours is assigned to be one more than its, namely i. So that pushes through the inductive step everything in layer i indeed gets the correct label of a shortest path distance i away from S. So before we wrap up with this application, I do wanna emphasize, it is only breadth first search that gives us this guarantee of shortest paths. So, we have a wide family of graph search strategies, all of which find everything findable. Breadth-first search is one of those, but this is a special additional property that breadth-first search has: you get shortest path distances from it. So in particular depth-first search does not in general compute shortest path distances. This is really a special property of breadth-first search. By contrast in this next application, which is going to be computing the connected components of an undirected graph, this is not really fundamental to breadth first search. For example, you could use depth first search instead and that would work just as well.

# 10-4-BFS and Undirected Connectivity

So what's the problem? Well, I did say most of the stuff about graph search really doesn't matter, undirected or directed, it's pretty much cosmetic changes. But the big exception is when you're computing connectivity, when you're computing the pieces of the graph. So right now, I'm only going to talk about undirected graphs. The directed case, we can get very efficient algorithms for, but it's quite a bit harder work, so that's going to be covered in detail in a separate video. So for now, focus just on an undirected graph, G. And we're certainly not going to assume that G is connected, and the part of the point here is to figure out whether or not it's connected, i.e., in one piece. So maybe the graph looks like this. So for example maybe the graph has ten vertices and looks like this on the right. And intuitively, especially given that I've drawn it in such a clean way, it's clear that this graph has three pieces. And those are the things that we want to call the connected components. But we do want to sum up more formal definitions, something which is actually in math that we could say is true or false about a given graph. And roughly, we define the connected components of an undirected graph as the maximal regions that are connected. In the sense you can get from any vertex in the region from any other vertex in the region using a path. So, maximal connected regions in that sense. Now the slick way to do this is using an equivalence relation. And I'm going to do this here in part because it's really the right way to think about the directed graph case, which we'll talk about in some detail later. So from your [INAUDIBLE] graphs, so this isn't super important, but let me go ahead and state the formal definition just for completeness about what is a connecting component. What do I mean by a maximal region that's mutually connected. So a good formal definition is as the equivalence classes of the relation on vertices where we define by u being related to v if and only if there's a path between u and v in the graph. So I'll leave it for you to do the simple check that this squiggle is indeed an equivalence relation. I'm going to remind you what equivalence relations are. This is something you generally learn in your first class on proofs or your first class in discrete math. So it's just something which may or may not be true about pairs of objects. In an equivalence relation, you have to satisfy three properties. So first you have to be reflexive, meaning everything has to be related to itself, and indeed in a graph there is a path from any node to itself, namely the empty path. Also a couple of these relations have to be symmetric, meaning that if u and v are related then v and u are related. Because this is an undirected graph it's clear that this is symmetric. If there's a path from u to v in the graph there's also a path from v to u so no problem there. Finally equivalence classes have got to be transitive. So that means if u and v are related and so are v and w and so are u and wo. That is if u and v have a path, v and w have a path, then so does u and w and you can prove transtivity just by pasting the two paths together. And so the upshot is, when you want to say something like the maximal subset of something where everything is the same. The right way to make that mathematical is using equivalence relations. So over in this blue graph we want to say one, three, five, seven, and nine, which are all the same in the sense that they're mutually connected and so that's exactly what this relation is making precise. All five of those nodes are related to each other. 2 and 4 are related to each other. 6, 8, and 10, all pairs of them are related. So equivalence relations always have equivalence classes, the maximal mutual related stuff. And in this graph context, it's exactly these connected components, exactly what you want. So what I want to show you is you can use wrapped in an outer of the vertices to compute, to identify all of the connected components of a graph. In time linear in the graph in n plus n time. Now you might be wondering why do you want to do that. Well there's a lot of reasons, so an obvious one which is relevant for physical networks is to check if a network has broken into two pieces. So certainly if you're an Internet service provider, you want to make sure that from any point in your network, you can reach any other point in the network. And that boils down to just understanding whether the graph that represents you network, is a connected graph, that is if it's in one piece, or not in one piece. So obviously, you can ask this same question about recreational examples. So if you return to the movie graph, maybe you're wondering, can you get from every single actor in the IMDB database to Kevin Bacon? Or are there actors for which you cannot reach Kevin Bacon? Via a sequence of edges, a sequences of movies in which two actors have both played a role. So that's something that boils down to a connectivity computation. If you have networked data and you want to display it, you want to visualize it and show it to a group of people so that they can interpret it. Obviously one thing you want to do is you want to know if there's multiple pieces, and then you want to display the different pieces separately. So let me make sure that one probably a little less obvious application of undirected connectivity is that it gives us a nice quick and dirty heuristic for doing clustering if you have paralyzed information about objects. Let me be a little more concrete. Suppose you have a set of objects that you really care about. This could be documents, maybe web pages that you crawl, something like that. It could be a set of images, either your own or drawn from some data base. Or it could be, for example, a set of genomes. Suppose further, that you have a pairwise function, which for each pair of objects tells you whether they are very much like each other or very much different. So let's suppose that two objects are very similar to each other, like the two web pages that are almost the same. Or there are two genomes where you can get from one to the other with a small number of mutations. Then, they have a low score. So low numbers, close to zero, indicates that the objects are very similar to each other. High numbers, let's say, they can go up to a thousand or something, indicate that they are very different objects. Two webpages that have nothing to do with each other, two genomes for totally unrelated parts, or two images that seem to be of completely different people or even completely different objects. Now here's a graph you can construct using these objects and the similarity data that you have about them. So you can have a graph where the nodes are the objects. Okay, so for each image, for each document, whatever, you have a single node and then for a given pair of nodes, you put in an edge if and only if the two objects are very similar. So for example, you could put in an edge between two objects if and only if the score is at most ten. So remember, the more similar two objects are, the closer there score is to zero. So you're going to get an edge between very similar documents, very similar genomes, very similar images. Now in this graph you've constructed, you can find the connected components. So each of these connected components will be a group of objects, which more or less are all very similar to each other. So this would be a cluster of closely related objects in your database. And you can imagine a lot of reasons why, given a large set of unstructured data, just a bunch of pictures, a bunch of documents or whatever, you might want to find clusters of highly related objects. So we'll probably see more sophisticated heuristics for clustering in some SQL course. But already undirected connectivity gives you a super fast linear time quick and dirty heuristic for identifying clusters of similar objects, given pairwise data about similarity. So that's some reasons you might want to do it. Now let's actually talk about how to compute the components in the near time using just a simple for loop and breadth-first search as it's inner work horse. So here's the code to compute all the connected components of an undirected graph. So first we initialize all nodes as being unexplored. I'm also going to assume that the nodes have names. Let's say that the names are from 1 to n. So these names could just be the position in the node array that these nodes occupy. So this is going to be an outer for loop, which walks through the nodes in an arbitrary order, let's say from 1 to n. This outer for loop is to ensure that every single node of the graph will be inspected for sure at some point in the algorithm. Now, again, one of maxims is that we should never do redundant work, so before we start exploring from some node, we check if they've already been there. And if we haven't seen i before, then we invoke the breath-first search, a term we were talking about previously in the lecture, in the graph G, starting from the node I. So to make sure this is clear, let's just run this algorithm on this blue graph to the right. So we start in the outer for loop and we said i equal to 1. And we say have we explored node number 1 yet. And of course not, we haven't explored anything yet. So the first thing we're going to do is we're going to invoke BFS on node number 1 here. So now we start running the usual breadth for search subroutine starting from this node one and so we explore layer one here is going to be nodes 3 and 5. So we explore them in some order for example maybe node number 3 is what we explore second. Then node number five is what we explore third and then the second layer in this component is going to be the nodes 7 and 9. So we'll explore them in some order as well and say 7 first followed by 9. So after this BFS initiated from node number one completes, of course it will have found everything it could possibly find, namely the five nodes in the same connected component as node number 1. And of course, all of the five of these nodes will be marked as explored. So now we return once that exits we return to the outer for loop we increment I we go to I equal 2, and we say we already explored node number 2 no we have not. And so now we invoke BFS again from node number 2. So that will be the sixth node we explore. There's not much to do from two, all we can do is go to 4, so that's the seventh node we explore. That BFS terminates at finding the nodes in this connected component, then we go back to the outer for loop. We increment i to 3, we say we've already seen node number three. Yes we have, we saw that in the first breadth first search. So we certainly don't bother to BFS from node 3, then we increment item four. Have we seen 4? Yes we have, in the second called to BFS. Have we seen node 5? Yes we have, in the first call to BFS. Have we seen node 6? No, we have not. So the final indication of breadth-first search begins from node number 6. That's going to be the eighth node overall that we see. And then we're going to see the notes 8 and 10 in some order, so for example maybe we first explore note number 8. That's one of the first layer in this component, and then note number 10 is the other note of the first layer in this component. So in general, what's going on, well what we know will happen when we invoked that first search from a given node i. We're going to discover exactly the nodes in i's connected component. Right, anything where there's a path from i to that node, we'll find it. That's the BFS guarantee, that's also the definition of a connected component. All the other nodes which have a path to i. Another thing that I hope was clear from the example, but just to reiterate it, is every search call, when you explore a node, you remember that through the entire for loop. So when we invoke breadth-first search from node number 1, we explore nodes 1, 3, 5, 7 and 9, and we keep those marked as explored for the rest of this algorithm. And that's so we don't do redundant work when we get to later stages of the for loop. So as far as what does this algorithm accomplish, well, it certainly finds every connected component. There is absolutely no way it can miss a node because this for loop literally walks through the nodes, all of them, one at a time. So you're not going to miss a node. Moreover, we know that as soon as you hit a connected component for the first time, and you do breadth-first search from that node, you're going to find the whole thing. That the breadth-first a search guarantee. As far as what's the running time, well it's going to be exactly what we want. It's going to be linear time, which again means proportional to the number of edges plus the number of vertices. And again depending on the graph, one of these might be bigger that the other. So why is it O of m plus n? Well as far as the nodes, we have to do this initialization there where we mark them all as unexplored, so that takes constant time per node. We have just the basic overhead of a for loop, so that's constant time per node. And then we have this check, constant time per node, so that's O of n. And then recall we proved that within breath for research, you do a amount of work proportional. You do constant time for each node in that connected component. Now, each of the nodes of the graph is in exactly one of the connected components. So you'll do constant time for each node in the BFS in which you discover that node. So that's again, o of n over all of the connected components. And as far as the edges, note we don't even bother to look at edges until we're inside one of these breadth first search calls. They played no role in the outer for loop or in the pre-processing. And remember what we proved about an indication of breadth first search. The running time, you only do constant amount of work per edge in the connected component that you're exploring. In the worst case, you look at an edge once from either endpoint and each of that triggers a constant amount of work. So when you discover a given connected component, the edge work is proportional to the number of edges in that kind of component. Each edge of the graph is only in exactly one of the connect components, so over this entire for loop, over all of these BFS calls. For each edge of the graph, you'll only be responsible for a constant amount of work of the algorithm. So summarizing because breadth-first search from a given starting node. That is, works in time proportional to the size of that component, piggybacking on that sub routine and looping over all of the nodes of the graph. We find all of the connecting components in time proportional to the amount of edges and nodes in the entire graph.

# 10-5-Depth-First Search (DFS) - The Basics

Let's explore our second strategy for graph search, namely depth-first search. And again, like with breadth-first search, I'll open by just reminding you what depth-first search is good for and we'll trace through it in a particular example, and then we'll tell you what the actual code is. So if breadth-first search is the cautious and tentative exploration strategy, then depth-first search or DFS for short is its more aggressive cousin. So the plan is to explore aggressively and only back track when necessary. And this is very much the strategy one often uses when trying to solve a maze. To explain what I mean let me show you how this would work in the same running example we used when we discussed breadth-first search. So here if we invoke depth-first search from the node number S, here's what's going to happen. So obviously we start at S and obviously there's two places where we can go next. We can go to A or to B. And depth-first search is under determine like breadth-first search we can pick either one. Select with a breadth-first search example let's go to A first. So A will be the second one that we explore. But now, unlike breadth first search where we automatically went to node B next, since that was the other layer one node. Here, the only rule is that we have to go next to one of A's immediate neighbors. We might go to B, but we're not going to B because it is one of the neighbor's of S, we go because it is one of the neighbors of A. And actually to make sure the difference is clear let's assume that we aggressively pursue deeper when we go from A to C and now the depth first search strategy is again to just pursue deeper, so you go to one of C's immediate neighbors, so maybe we go to E next, so E is going to be the fourth one visited. Now from E there's only one neighbor not counting the one that we came in on so from E we go to D. And D is the fifth one we see. Now from D we have a choice, we can either go to B or we could go to C. So let's suppose we go to C from D. Well then we get to a node number three where we've been before. And as usual we're going to keep track of where we've already been. So at this point we have to back track from C back to D. We retreat to D. Now, there's still another outgoing edge from D to explore and then they'll be the one to B. And so what happens is we actually wind up wrapping all the way around this outer cycle and we could B sixth. And now, of course, anywhere we try to explore we see somewhere we've already been. So, from B we try to go to S, but we've been there so we retreat to B. We can try to go to A but we've been there so we retreat to B. Now we've explored all of the options out of B. So we have to retreat from B, we have to go back to D. Now from D we've explored both B and C so we have to retreat back to E. We've explored the only outgoing arc D, so we have to retreat to C. We retreat to A. From A we actually haven't yet looked along this arc, but that just sends us to B where we have been before. So then we retreat back to A. Finally we retreat back to S and S, even at S there's still an extra edge to explore. At S we say, we haven't tried this as B-edge yet. But of course, when we look across we get the B where we've been before and then we backtrack to S. Then we've looked at every edge once, and so we stop. So that's how depth-first search works. You just pursue your path, you go to an immediate neighbor as long as you can until you hit somewhere you've been before. And then you retreat. So you might be wondering why bother with another graph search strategy? After all we have breadth-first search, which seemed pretty awesome, right. It runs in linear time. It's guaranteed to find everything you might want to find, it computes shortest paths, it computes connected components if you embed it in a foreloop. It kind of seems like, what else would you want? Well, it turns out, depth-first search is going to have its own impressive catalogue of applications, which you can't necessarily replicate with breadth-first search. And I'm going to focus on applications in directed graphs. So there's going to be a simple one that we discuss in this video, and then there's going to be a more complicated one that has a separate video devoted to it. So in this video we're going to be discussing, computing topological orderings of directed acyclic graphs. That is, directed graphs that have no directed cycle. The more complicated application is computing strongly connected components in directed graphs. The run time will be, essentially, the same as it was for breadth-first search, and the best we could hope for, which is linear time. And again, we're not assuming that there's necessarily that many edges. There may be much fewer edges than vertices. So linear time and these connectivity applications means O of M plus n. So let's not talk about the actual code of depth for search. There's a couple of ways to do it. One way to do it is to just make some minor modifications to the code for breadth for a search. The primary difference being instead of using a queue in its first in first out behavior, you swap in a stack with its last in first out behavior. Again, if you don't know what a stack is you should read about that in the program textbook or on the web. It's something that supports constant time insertions to the front and constant time deletions from the front, unlike a queue which is meant to support constant time deletions to the back. Okay so stack that operates just like those cafeteria trays that you know where you put in a tray and the last one that got pushed in when you take the first one out that's the last one that got put in. So these are called push and pop, in a stack context both are constant time. So if you swap out the queue you swap in the stack, make a couple other minor modifications. Breadth-first search turns into depth-first search. For the sake of both variety and elegance, I'm instead going to show you a recursive version. So depth-first search is very naturally phrased as a recursive algorithm, and that's what we'll discuss here. So, depth-first search, of course, takes as input a graph g and again it could be undirected or directed. It doesn't matter, just with a directed graph be sure that you only follow arcs in the appropriate direction, which should be automatically handled in the adjacency lists of your graph data structure anyways. So as always we keep a Boolean local to each vertex of the graph, remembering whether we've been there before or not. And of course, as soon as we start exploring from S we better make a note that now we have been there. We better plant a flag, as it were. And remember, depth-first search is an aggressive search, so we immediately try to recursively search from any of S's neighbors that we haven't already been to. And now, if we find such a vertex, if we find somewhere we've never been, we recursively call depth-first search from that node. The basic guarantees of depth-first search are exactly the same as they were for breath-first search. We find everything we could possibly hope to find and we do it in linear time. And once again, the reason is this is simply a special case of the generic stretch procedure that we started this sequence of videos about. So it just corresponds to a particular way of choosing amongst multiple crossing edges between the region of explored nodes, and between the region of unexplored nodes. Essentially always being biased toward the most recently discovered explored nodes. And just like, breadth for search, the running time is going to be proportional to the size of the component that you're discovering. And the basic reason is that each node is looked at only once, right? This boolean makes sure that we don't ever explore a node more than once, and then for each edge, we look at it at most twice, once from each endpoint. And given that these exact same two claims hold for depth-first search as for breadth-first search, that means if we wanted to compute connected components in an undirected graph, we could equally well use an outer for loop with depth-first search as our workhorse in the inner loop. It wouldn't matter. Either of those for undirected graphs, depth-first search, breadth-first search, is going to find all the connected components in O of n plus m time, in linear time. So instead, I want to focus on an application in particular to depth-first search, and this is about finding a topological ordering of a directed acyclic graph.

# 10-6-Topological Sort

Let me begin by telling you what a topological ordering of a directed graph is. Essentially, it's an ordering of the vertices of a graph so that all of the arcs, the directed edges of the graph, only go forward in the ordering. So let me encode an ordering by a labeling of the vertices with the numbers one through n. This is just to encode the position of each vertex in this ordering. So formally there's going to be a function which takes vertices of G and maps them to integers between 1 and n. Each of the numbers 1 through n should be taken on by exactly one vertex. Here n is the number of vertices of G. So that's just a way to encode an ordering, and then here's really the important property that every directed edge of G goes forward in the ordering. That is if u v is directed edge of the directed graph G, then it should be that the f value of the tail is less than the f value of the head. That is this directed edge has the higher f value as traverse in the correct direction. Let me give you an example just to make this more clear. So suppose we have this very simple directed graph, with four vertices. Let me show you two different, totally legitimate topological orderings of this graph. So the first thing you could do, is you could label s1, v2, w3 and t4. Another option would be to label them the same way, except you can swap the labels of v and w. So if you want, you can label v 3 and w 2. So again, what these labelings are really meant to encode is an ordering of the vertices. So the blue labeling, you can think of as encoding the ordering in which we put s first then v then w and then t. Whereas the green labeling can be thought of as the same ordering of the nodes except with w coming before v. What's important is that the pattern of the edge is exactly the same in both cases, and in particular all of the edges go forward in this ordering. So in either case we have s with edges from s to v, and s to w. So that looks the same way pictorially, whichever order v or w are in, and then symmetrically there are edges from v and w to t. So you'll notice that no matter which order that we put v and w in, all four of these edges go forward in each of these orderings. Now if you try to put v before s it wouldn't work because the edge from s to v would be going backward if v preceded s. Similarly, if you put t anywhere other than the final position, you would not have a topological ordering. So in fact, these are the only two topological orderings of this directed graph. I encourage you to convince yourself of that. Now, who cares about topological orderings? Well, this is actually a very useful subroutine. This has been come up in all kinds of applications. Really, whenever you want to sequence a bunch of tasks when there's precedent constraints among them. By precedence constraint I mean one task has to be finished before another. You can think, for example, about the courses in some kind of undergraduate major like a computer science major. Here the vertices are going to correspond to all of the course and there's a directed edge from course A to course B if course A is a prerequisite for course B, if you have to take it first. So then of course, you'd like to know a sequence in which you can take these courses so that you always take a course after you've taken its pre-requisite. And that's exactly what a topological ordering will accomplish. So it's reasonable to ask the question when does a directed graph have a topological ordering and when a graph does have such an ordering, how do we get our grubby little hands on it? Well there's a very clear necessary condition for a graph to have a topological ordering, which is it had better be a cyclic. Put differently, if a directed graph has a directed cycle then there is certainly no way there is going to be a topological ordering. So I hope the reason for this is fairly clear. Consider any directed graph which does have a directed cycle and consider any purported way of ordering the vertices. Well, now just traverse the edges of the cycle, one by one. So you start somewhere of the cycle, and if the first edge goes backward, well, you're already screwed. You already know that this ordering is not topological. No edges can go backward. So evidently, the first edge of this cycle has to go forward. But now, you have to traverse the rest of the edges on this cycle, and eventually you come back to where you started. So if you started out by going forward, at some point you have to go backward. So that edge goes backward in the ordering, violating the property of the topological ordering. That's true for every ordering, so directed cycles exclude the possibility of topological ordering. Now the question is what if you don't have a cycle? Is that a strong enough condition that you're guaranteed to have a topological ordering? Is the only obstruction to sequencing jobs without conflicts the obvious one of having circular precedence constraints? So it turns out not only is the answer yes, as long as you don't have any directed cycles, you're guaranteed a topological ordering. But we can even compute on in linear time no less via depth-first search. So before I show you the super slick and super efficient reduction of computing topological orderings of depth-first search. Let me first show over a pretty good but slightly less slick and slightly less efficient solution to help build up your intuition about directed acyclic graphs and their topological orderings. So for the straightforward solution, we're going to begin with a simple observation. Every directed acyclic graph has what I'm going to call a sink vertex. That is a vertex without any outgoing arcs. So in the forenode, directed acyclic graph we were exploring on the last slide there's exactly one source of sink vertex, and that's this right-most vertex here. That has no outgoing arcs, the other three vertices all have at least one outgoing arc. Now why is it the case that a directed acyclic graph has to have a sink vertex? Well, suppose it didn't, suppose it had no sink vertex. That would mean every single vertex has at least one outgoing arc. So what could we do if every vertex has one outgoing arc? Well, we can start in an arbitrary node. We know it's not a sink vertex, because we're assuming there aren't any. So there's an outgoing arc, so let's follow it. We get to some other node. By assumption there's no sink vertex, so this isn't a sink vertex, so there's an outgoing arc, so let's follow it. We get to some other node. That also has an outgoing arc, let's follow that, and so on. So we just keep following outgoing arcs, and we do this as long as we want because every vertex has at least one outgoing arc. Well there's a finite number of vertices, right this graph has say N vertices. So if we follow N arcs, we are going to see N+1 vertices. So by the pigeon-hole principle, we're going to have to see a repeat. Right so, if N+1 vertices, is only N distinct vertices, we're going to see some vertex twice. So for example, maybe after I take the outgoing arc from this vertex, I get back to this one that I saw previously. Well, what have we done? What happens when we get a repeated vertex? By tracing these outgoing arcs and repeating a vertex, we have exhibited a directed cycle. And that's exactly what we're assuming doesn't exist. We're talking about directed acyclic graphs. So, put differently, we just prove that a vertex with no sink vertex has to have a directed cycle. So a directed acyclic graph therefore has to have at least one sink vertex. So here's how we use this one very simple observation now to compute a topological ordering of a directed acyclic graph. Well let's do a little thought experiment. Suppose in fact this graph did have a topological order. Let's think about the vertex which goes last in this topological ordering. Remember, any arc which goes backward in the ordering is a violation. So we have to avoid that. We have to make sure every arc goes forward in the ordering. Now any vertex which has an outgoing arc, we better put somewhere other than in the final position, right? So the node that we put in the final position, all of its outgoing arcs are going to wind up going backward in the topological ordering. There's no where else they can go, this vertex is last. So in other words, if we plan to successfully compute a topological ordering, the only candidate vertices for that final position in the ordering are the sink vertices. That's all that's going to work. We put a non-sink vertex there, we're toast, it's not going to happen. Fortunately, if it's directed acyclic, we know there is a sink vertex. So let v be a sink vertex of g, if there's many sink vertices, we pick one arbitrarily. We set v's label to be the maximum possible, so if there's n vertices we're going to put that in the nth position. And now we just recurse on the rest of the graph, which has only n-1 vertices. So how would this work in the example on the right? Well in the first iteration, or the first outermost recursive call, the only sink vertex is this right most one, circled in green. So there's four vertices, so we're going to give that the label 4. So, then having labeled that 4, we delete that vertex and all the edges incident to it. And we recurse on what's left of the graph, so that would be the left-most three vertices plus the left-most two edges. Now, this graph has two sink vertices, after we've deleted 4 and everything from it. So both this top vertex and this bottom vertex are sinks in the residual graph. So now in the next recursive call, we can choose either of those as our sink vertex. Because we have two choices, that generates two topological orderings. Those are exactly the ones that we saw in the example. But if, for example, we choose this one to be our sink vertex, then that gets the label 3. Then we recurse just on the northwestern most two edges. This vertex is the unique sink in that graph, that gets the label 2. And then it recurs on the one node that we graph, and that gets the label 1. So, why is this algorithm work? Well, there's just two quick observations we need. So, first of all, we need to argue that it makes sense that in every iteration or in every recursive call, we can indeed find the sink vertex, that we can assign in the final position that's still unfilled. And the reason for that is just if you take a directed acyclic graph and you delete one or more vertices from it, you're still going to have a directed acyclic graph, right? You can't create cycles by just getting rid of stuff. You can only destroy cycles, and we started with no cycles. So through all the intermediate recursive calls we have no cycles by our first observation is always the sink. So the second that that we have to argue is that we really do produce a topological ordering. So remember what that means, that means for every edge of the graph, it goes forward in the ordering. That is the head of the arc is given a position later than the tail of the arc. And this simply follows because we always use sink vertices. So consider the vertex v which is assigned to the position i. This means then, that when we're down to a graph that only has i vertices remaining, v is the sink vertex. If v is the sink vertex when only the first i vertices remain, what property does it have in the original graph? Well, it means all of outgoing arcs that it has have to go to vertices that were already deleted and assigned higher positions. So for every vertex, by the time it actually gets assigned a position, it's a sink and it only has incoming arcs from the as yet unsigned vertices. It's outgoing arcs all go forward to vertices that were already assigned higher positions, and got deleted previously from the graph. So now we have under our belt a pretty reasonable solution for computing a topological ordering of a directed acyclic graph. In particular, remember we observed that if a graph does have a directed cycle, then of course there's no way there's a topological ordering. However, you order the vertices summage of the side that's going to have to go backward. And the solution on the previous slide shows that, as long as you don't have a cycle, it guarantees the topological ordering does indeed exist. And in fact, it's a constructive proof, constructive argument, that gives an algorithm. What you do is you would keep plucking off sinks, sink vertices one at a time, and populating the ordering from right to left, as you keep peeling off these sinks. So that's a pretty good algorithm, it's not too slow, and actually if you implement it just so, you can even get it to run in linear time. But I want to conclude this video with an application of depth first search, which is a very slick, very efficient computation of a topological ordering of a directed acyclic graph. So, we're just going to make two really quite minor modifications to our previous depth first search subroutine. The first thing is we have to embed it in a for loop, just like we did with breadth first search when we were computing the connected components of an undirected graph. That's because, in completing a topological ordering, we better give every single vertex a label, we'd better look at every vertex at least once. So to do that, we'll just make sure there's an outer for loop and then if we have multiple components, we'll just make sure to invoke DFS as often as we need to. The second thing we'll do is we'll add a little bit of bookkeeping, and this will make sure that every node gets a label. And in fact, these labels will define a topological order. So let's not forget the code for depth first search. This is where you're given a graph, G, in this case we're interested in a directed acyclic graph, and you're given a start vertex, S. And what you do is, as soon as you get to S you very aggressively start trying to explore its neighbors. Of course, you don't visit any vertex you've already been to. You keep track of who you visited. And if you find any vertex that you haven't seen before, you immediately start recursing on that node. So I said the first modification we need is to embed this into an outer for loop to ensure that every single node gets labeled. So I'm going to call that subroutine DFS-Loop. It does not take a start vertex. Initialization, all nodes start out on an explorative course. And we're also going to keep track of a global variable, which I'll call current\_label. This is going to be initialized to n, and we're going to count down each time we finish exploring a new node. And these will be precisely the f values. These will be exactly the positions of the vertices in the topological ordering that we output. In the main loop we're going to iterate over all of the nodes of the graph. So for example, we just do a scan through the node array. As usual, we don't want to do any work twice, so if a vertex has already been explored in some previous invocation of DFS, we don't search from it. This should all be familiar from our embedding of breadth first search in a for loop when we computed the connected components of an undirected graph. And if we get to a vertex v of the graph, that we haven't explored yet, then we just invoke DFS in the graph, with that vertex as the starting point. So, the final thing I need to add is I need to tell you what the f values are, what the actual assignments of vertices to positions are. And as I foreshadowed, we're going to use this global current\_label variable. And that'll have us assign vertices to positions from right to the left. Very much mimicking what was going on in our recursive solution, where we plucked off sink vertices one at a time. So, when's the right time to assign a vertex its position? Well, it turns out, the right time is when we've completely finished with that vertex. So we're about to pop the recursive call from the stack corresponding to that vertex. So, after we've gone through the for loop of all the edges outgoing from a given vertex, we set f (s) = to whatever the current\_label is and then we decrement the current\_label. And that's it, that is the entire algorithm. So, the claim is going to be that the f values produced, which you'll notice are going to be the integers between n through 1, because DFS will be called eventually once on every vertex, and it will get some integer assignment at the end. And everybody is going to get a distinct value, and the largest one is n and the smallest one is 1. The claim is that is a topological ordering. Clearly this algorithm is just as blazingly fast as DFS itself, with just a trivial amount of extra bookkeeping. Lets see how it works on our running example. So lets just say we have this four node directed graph that we're getting quite used to. So this has four vertices, so we initialize the current label variable to be equal to 4. So let's say that in the outer DFS loop, let's say we start somewhere like the vertex v. So notice in this outer for loop, we wind up considering the vertices in a totally arbitrary order. So let's say we first call DFS from this vertex v. So what happens, well, the only place you can go from v is to t, and then at t, there's nowhere to go. So we recursively call DFS at t, there's no edges to go through, we finish the for loop, and so t is going to be assigned an f value equal to the current label, which is n, and here, n is the number of vertices, which is 4. So f(t) is going to get, so our t is going to get the assignment, the label 4. So then now we're done with t, we backtrack back to v. We decrement the current label as we finish up with t, we get to v, and now there's no more outgoing arcs to explore, so for loops finish. So we're done with it in depth-first search. So it gets what's the new current label, which is now 3, and again, having finished with v, we decrement the current label, which is now down to 2. So now we go back to the outer for loop, maybe the next vertex we consider is the vertex t. But we've already been there, so we don't bother to DFS on t. And then maybe after that, we try it on s. So maybe s is the third vertex that the for loop considers. We haven't seen s yet, so we invoke DFS, starting from the vertex s. From s, there's two arcs to explore, the one with v, v we've already seen, so nothing's going to happen with the arc sv. But on the other hand, the arc sW will cause us to recursively call DFS on w. From w, we try to look at the arc from w to t, but we've already been to t, so we don't do anything. That finishes up with w, so depth-first search then finishes up at the vertex w, w gets the assignment of the current label. So f(w) = 2. We decrement current label, now its value is 1. Now we backtrack to s, so we've already considered all of s's outgoing arcs, so we're done with s. It gets the current label, which is 1. And this is indeed one of the two topological orderings of this graph that we exhibited a couple slides ago. So that's the full description of the algorithm and how it works in a concrete example. Let's just discuss what are its key properties, its running time and its correctness. So, as far as the running time of this algorithm the running time is linear. It's exactly what you'd want it to be. And the reason the running time is linear is for the usual reasons that these search algorithms run in linear time. You're explicitly keeping track of which nodes you've been to so that you don't visit them twice, so you only do a constant amount of work for each of the n nodes. And each edge, in a directed graph, you actually only look at each edge once, when you visit the tail of that edge. So you only do a constant amount of work per edge as well. Of course, the other key property is correctness. That is, we need to show that you are guaranteed to get a topological ordering. So what does that mean? That means every arc travels forward in the ordering. So if (u,v) is an edge, then f(u), the label assigned to u in this algorithm is less than the label assigned to v. The proof of correctness splits into two cases, depending on which of the vertices u or v is visited first by depth-first search. Because of our for loop, which iterates over all of the vertices of the graph g, depth-first search is going to be invoked exactly once from each of the vertices. Either u or v could be first, both are possible. So first let's assume that u was visited by DFS before v, so then what happens? Well, remember what depth first search does, when you invoke it from a node, it's going to find everything findable from that node. So if u is visited before v, that means v isn't getting explored, so it's a candidate for being discovered. Moreover, there's a an arc straight from u to v, so certainly DFS invoked at u is going to discover v. Furthermore, the recursive call corresponding to the node v is going to finish, it's going to get popped off the program stack before that of u. The easiest way to see this is just to think about the recursive structure of depth-first search. So when you call depth-first search from u, that recursive call, that's going to make further recursive calls to all of the relevant neighbors including v, and u's call is not going to get popped off the stack until v's does beforehand. That's because of the last in, first out nature of a stack or of a recursive algorithm. So because v's recursive call finishes before that of u, that means it will be assigned a larger label than u. Remember, the labels keep decreasing as more and more recursive calls get popped off the stack. So that's exactly what we wanted. Now, what's up in the second case, case two? So this is where v is visited before u. And here's where we use the fact that the graph has no cycles. So there's a direct arc from u to v. That means there cannot be any directed path from v all the way back to u. That would create a directed cycle. Therefore, DFS invoked from v is not going to discover u. There's no directed path from v to u, again, if there was, there'd be a directed cycle. So it doesn't find u at all. So the recursive call of v again is going to get popped before u's is even pushed onto the stack. So we're totally done with v before we even start to consider u. So therefore, for the same reasons, since v's recursive call finishes first, its label is going to be larger, which is exactly what we wanted to prove. So that concludes the first quite interesting application of depth-first search. In the next video, we'll look at an even more interesting one which computes the strongly connected components of a directed graph. This time, we can't do it in one depth-first search, we'll need two.

# 10-7-Computing Strong Components - The Algorithm

Having mastered computing the connecting components of an undirected graph in linear time, let's now turn our attention to directed graphs, which also arise in all kinds of different applications. Now the good news is, is we'll be able to get just a blazingly fast primitive for computing connectivity information for directed graphs. The bad news, if you want to call it hat, is we'll have to think a little bit harder. So it won't be as obvious how to do it. But by the end of this lecture will you know linear time algorithms, a very good concept. It's really just based on depth first search for computing all of the pieces of a directed graph. In fact, it's not even so obvious how to define pieces, how to define connected components in a directed graph. Certainly not as obvious as it was with undirected graphs. So, see what I mean, let's consider the following four node directed graph. So on the one hand, this graph in some sense in one piece. If this was an actual physical object, made say of a bunch of strings connected to each other and we picked it up with our hands, it wouldn't fall apart into two pieces, it would hang together in one piece. On the other hand, when you think about moving around this network, it's not connected in the sense that we might think about. You cannot get from any one point a, to any other point b. For example, if you started the right-most node in this graph, certainly there's no directed path that will get you to the left-most node. So what's typically studied and most useful for directed graphs is what you call strong connectivity. And a graph is strongly connected if you can get from any one point to any other point and vice versa. And the components then informally are the maximal portions of this graph, the maximal regions, which are internally strongly connected. So the maximum regions from within which you can get from any one ,point a to any other point b along a directed graph. For the record, let me go ahead and give you a formal definition, although this intuition is perfectly valid. Just regions where you can get from anywhere to anywhere else. So we say that the strongly connected components of directed graph, or the SCCs for short. And as in the undirected case, we're going to give a somewhat slick definition rather than talking about maximal region satisfying some property. We're going to talk about the equivalence classes of a particular equivalence relation. But really it means the same thing. This is just sort of the more mathematically more mature way of writing it down. So the equivalence relation we're going to use, if it's on nodes of the graph, and we'll say that u and node is related to a node v if you can get from u to v via directed path, and also from v to u in some other directed path. I'm not going to bore you with the verification that this is an equivalence relation. That's something you can work out in the privacy of your home. So remember what it means to be an equivalence relation, that's reflexive, that is everybody's related to itself. But of course there is a path from every node to itself. It also means that it's symmetric, so if u is related to v, then v is related to u. Well again, by definition we're saying that the vertices are mutually reachable from each other. So that's clearly symmetric by definition. And then it has to be transitive. And the way you prove it's transitive is you just paste paths together, and it just works the way you'd expect it to. So let's illustrate this concretely with a somewhat more complicated directed graph. So here's a directed graph, and I claim that it has exactly four strongly connecting components. There's a triangle on the left. A triangle on the right. It has this single node on top. And then it has this directed forecycle with a diagonal at the bottom. So what I hope is pretty clear is that each of these circled regions is indeed strongly connected. That is, if you start from one node in one of these circled regions, you can have a directed path to any other node. So that's symmetric, because on a directed cycle you can get from any one starting point to anyother place. And all of these have directed cycles. And then there's the one strong component that just has the one node, which is obviously strongly connected. What I also claim is true is that all of these regions are maximal. Subject to being strongly connected. That's why they're the strongly connected components. That's why they're the equivalence classes of this equivalence relation we just defined. So if you take any two pairs of nodes which lie in two different circles, you either won't have a path from the first one to the second, or you won't have a directed path from the second one back to the first one. In fact, the structure of the strong components in this black graph exactly mirrors the directed acyclic graph that we started in red. So in the same way in the red four-node graph, you can't move from right to left. Here in this bigger graph, you can't move from any of the circled SCCs to the right to any of the circled SCCs to the left. So for example, from the right-most nodes, there are no directed paths to the left-most nodes. So that's a recap of the definition of the strongly connected components. I've motivated in a separate video some reasons why you might care about computing strongly connected components. And in particular, on extremely large graphs which motivates the need for blazingly fast sub routines, so four free primitives that will let you compute connectivity information. So you'd love to be able to just know the pieces of a directed graph. Maybe you don't even know why they're going to be useful but you just compute them because why not? It's four free primitive. So, that's what I'm going to give you on this lecture. So the algorithm that we're going to present is based on depth-first search. And that's going to be the main reason why it's going to be so blazingly fast, is because depth-first search is blazingly fast. Now, you might be wondering, what on earth does graph search have to do with computing components? They don't seem obviously related. So let's return to the same directed graph that I shows you on the previous slide. And to see why something like depth-first search might conceivably have some use for computing strong components. Suppose we called depth-first search starting from this red node as a starting point. What would it would explore? So remember what the guarantee of something like depth-first search or breath first search for that matter is. You find everything that findable, but naturally nothing else. So what is findable from this red node? Where by findable I mean, you can reach it from a directed path emanating from this red node. Well, there's not much you can do. So from here you can explore this arc. And you can explore this arc. And then you can go backward. And so, if you do DFS or BFS from this node, you're going to find precisely the nodes in this triangle. All of the other arcs involved go the wrong direction and they won't be traversed by, say, a depth-first search call. So, why is that interesting? What's interesting is that if we invoke DFS from this red node, or any of the three nodes from this triangle, then it's going to discover precisely this strongly connected component, precisely the three nodes in this circled SCC. So that seems really cool, seems like maybe we just do a DFS, and boom we get an SCC. So maybe if we can do that over and over again we'll get all the SCCs. So that's a good initial intuition, but something can go wrong. Suppose that instead of initiating DFS from one of these three nodes on the triangle, we say, initiated from this bottom most node in green. So remember, what is the guarantee of a graph search subroutine like DFS? It will find everything findable but of course, nothing more. So what's findable from this green node? Well, naturally everything in its own SCC, right? So the four nodes here, it'll certainly find those four nodes. On the other hand, if we start from this green node, since there are arcs that go from this bottom-most SCC to the right-most the SCC. Not only will this DFS call find the four nodes in the green node's strong component, but it will also traverse these blue arcs and discover the three nodes in the red triangle. So, if we call DFS from this green node, we'll capture all seven of these. So the point is, if we call DFS, it looks like we're going to get a union of possibly multiple SCCs. In fact, in the worst case, if we invoke DFS from the leftmost node, what's it going to discover? It's going to discover the entire graph. And that didn't give us any insight into the strong component structure at all. So, what's the takeaway point is, the takeaway point is if you call DFS from just the right place, you'll actually uncover an SCC. If you call it from the wrong place, it will give you no information at all. So the magic of the algorithm that we're going to discuss next is we'll show having this super slick pre-processing step which ironically is itself is called a depth-first search. We can in linear time compute exactly where we want to start the subsequent depth-first searches from, so that each indication gets us exactly one strongly connected component and nothing more. So the algorithm that I'm going to show you is due to Kosaraju and it will show the following theorem, that the strongly connected components of a directed graph can be computed in linear time. And as we'll see, the constants are also very small. It's really just going to be two passes of depth first search. And again I'm going to remind you that for many problems, it's natural to assume that the number of edges is at least the number of nodes because you're only interested in cases where the graph is connected. Of course when you're computing connected components, that's one of the most natural cases where you might have a super sparse broken up graph. So we're not going to assume M is at least N so that's why linear time is going to be M plus N because that's the size of the input. And we don't know either m could be bigger than n or n could be bigger than m. We have no idea. Kosaraju's algorithm is shocking in its simplicity. It has three steps. Let me tell you what they are. First very mysteriously we are going to reverse all of the archs of the given graph. Totally not clear why that would be an interesting thing to do yet. Then we're going to do our first pass, our first depth first search, and we're going to do it on the reverse graph. Now the naive way to implement this would be to literally construct a new copy of the input graph with all the the arcs in the reverse direction, and then just run depth first search on it. Of course, the sophisticated, the sort of obvious optimization would be to just run DFS on the original graph, but going across arcs backwards. So I'll let you think through the details of how you'd do that, but that just works. You run DFS, and instead of going forward along edges, you go backward along edges, that simulates depth first search on the reverse graph. Now I've written here DFS loop and that just means the user will check more to make sure that you see all of the nodes of the graph even if it's disconnected you have an outer loop where you just try each starting point separately. If you haven't already seen it then you run DFS from that given node. I'll be more detailed on the next slide. And the third step is just you run depth-first search again, but this time on the original graph. Now at this point you should be thinking that I'm totally crazy, right, so what are we trying to do? We're trying to compute these strongly connected components. We're trying to actually compute real objects, these maximal regions and all I'm doing is searching the graph. I do it once forward. I do it once backward. I mean it doesn't seem like I'm computing anything. So here's the catch and it's a very minor catch. So we're going to get you to do a little bit of bookkeeping, it's going to be very little overhead so we'll still have a blazingly fast algorithm. So but with a little bit of bookkeeping, here's what's going to happen. The second depth first search. Which searches the graph, will in it's search process discover the components one at a time in a very natural way. And that will be really obvious when we do an example which we'll do in just a second. Now, for the second depth first search to work as magical way where it just discovers the connective component one at a time, it's really important that executes the depth first searches in a particular order, that it goes to the nodes of the graph in a particular order. And that is exactly the job of the first pass. The depth first search on reverse graph is going to compute an ordering of the nodes which, when the second depth first search goes through them in that order, it will just discover the SCCs one at a time. In linear time. So let me say a little bit more about the form of the bookkeeping and then I'll show you how that bookkeeping is kept in as we do depth-first search. So we're going to have a notion of a finishing time of a vertex. And that's going to be computed in the first pass when we do depth-first search in the reverse graph. And we're going to make use of this data in the second pass. So rather than just going through the nodes of the graph in an arbitrary order, like we usually do when we sort of have a loop to depth first search. We're going to make sure that we go through the vertices in decreasing order of these finishing times. Now there's still the question of what sense doe this second depth first search discover and report to the strong connected components that it finds? So we're going to label each node in the second pass with what we call a leader. And the idea is that the nodes in the same strong connected component will be labeled with exactly the same leader node. And again, all of this will be much more clear once we do a concrete example, but I want to have it down for the record right now. So that's the algorithm at a high level. It's really just two passes of DFS with some bookkeeping, but this is under specify. You really shouldn't understand how to implement the algorithm just yet. So what do I owe you? I owe you exactly what I mean by the DFS-loop, although this is seen more or less in the past. It's just a loop over all the vertices of the graph, and if you haven't seen something yet in DFS from that starting point, I need to tell you what finishing times are and how they get computed. They're just going to be integers 1 to n, which is basically when depth first search gets finished with one of the nodes, and then I need to tell you how you compute these leaders. So, let me tell you all three of those things on the next slot. So the work course for Kosaraju's strongly connected components algorithm is this DFS-loop subroutine, and it takes, as input, a graph. So it does not take as input a starting node, it's going to loop over possible starting nodes. Now for the bookkeeping to compute finishing nodes, we're going to keep track of a global variable of it all called T. Which we initialize to zero. The point of T is to count how many nodes we've totally finished exploring at this point. So this is the variable we use to compute those finishing times in the first pass, that magical ordering that I was talking about. Now we're also going to have a second global variable to compute these things I was calling leaders, and these are only relevant for the second pass. So what S is going to keep track of is the most recent vertex from which a DFS was initiated. So to keep the code simple, I'm just going to do all of the bookkeeping in the DFS-Loop, so really DFS-Loop gets called twice, once in a reverse graph, once in the forward graph. And we only need to compute these finishing times in the first pass on the reverse graph and we only need to compute these leaders on the second pass for the forward graph. But let's just keep them both in there just for kicks. Now we're going to need to loop through the vertices. And so the question is in what order are we going to loop through the vertices? And that's going to happen differently in the two different passes, but let me just use some common notation. Let's just assume, in this sub-routine, that the nodes are somehow labeled from 1 to n. In our first depth first search it's going to be labeled totally arbitrary, so these are basically just the names of the node or their position in the node array, whatever, you just do it in some arbitrary order. Now the second time we run DFS loop, as indicated on the previous slide, we're going to use the finishing times as the labeling. As we'll see, the finishing times are indeed numbers between 1 in it, so now what do we do is we just iterate through the nodes in decreasing order. And if we haven't already seen node i, then we initiate a DFS from it, so as usual we're going to be maintaining a local boolean to keep track of what we had already seen a node yet in one of the DFS passes. Now remember, the global variable s is responsible for keeping track of the most recent node from which Depth First Search had been initiated, so if i's not explored and we initiate a Depth First Search from it, we better reset s. And then we do the usual DFS ng starting from the source node i. So for completeness let me just remind you what the depth first search sub-routine looks like, so now we're given a graph and a starting node. So the first time we see a node we mark it as explored. And just a side note that once a node is marked explored, it's explored for this entire indication of DFS loop. Okay so even if this DFS from a given node i finishes, and then the outer for loop marches on, and encounters i again, it's still going to be marked as explored. Now one of our bookkeeping jobs is to keep track of from which vertex did the DFS that discovered i get called. So when i is first encountered, we remember that s was the node from which this DFS originated. And that by definition is the leader of i. And then we do what we always do with depth first search, we immediately look at the arcs going our of i and we try recursively DFS on any of those. Although we don't bother to do it if we've already seen those nodes. Now once this for loop has completed, once we've examined every outgoing arc from i and for each node j. Either we already saw it in the past or we've recursively explored from j and have returned. At the point, we call ourselves done with node i, there's no more outgoing arc to explore. We think of it being finished, remember t is the global variable that's keeping track of how many nodes we're done with, so we increment t because now we're done with i. And we also remember that i was the t-th vertex with which we finished. That is, we said i's finishing time to be t. Because depth first search is guaranteed to visit every node exactly once, and that therefore finish with every node exactly once. This global counter t, well when the first node is finished it'll be value 1, then when the next node gets finished I'll have value 2, then it'll have value 3 and so on. When the final node gets finished with it'll have value n. So the finishing times of the nodes are going to be exactly the integers from 1 to n. Let's make this much more concrete by going through a careful example. In fact, I think it'll be better for everybody if you, yourself, traced through part of this algorithm on a concrete example. So let me draw a nine node graph for you. So to be clear, let's assume that we've already executed step one of the algorithm, and we've already reversed the graph. So that is, this blue graph that I've drawn on the slide, this is the reversal. We've already reversed the arcs. Moreover the nodes are labeled in some arbitrary way from 1 to 9. Just assume these are how they show up in the node array for example and remember in the DFS loop routine you're supposed to process the nodes from top to bottom from n down to 1. So my question for you then is in the second step of the algorithm when we run DFS-Loop, and we process the nodes from the highest name 9 in order down to the lowest name 1. What are the finishing times that we're going to compute as we run DFS-Loop? Now, it is true that you can get different finishing times depending on the different choices that the DFS-Loop has to make about which outgoing arc to explore next. But I've given you four options for what the finishing times of the nodes 1,2,3, all the way up to 9, respectively, might be. And only one of these four could conceivably be an output of the finishing time of DFS loop on this graph, so which one is it? All right so the answer is the fourth option, that is the only one of these four sets of finishing times that you might see being computed by DFS-Loop on this blue graph. So let's go ahead and trace through DFS-Loop and see how we might get this set of finishing times. So remember in the main loop we start from the highest node 9 and then we descend down to the lowest node 1. So we start by invoking DFS from the node 9. So now from here there's only one outgoing arc, we have to go to so we mark 9 as explored. And then there's only one place we can go, we can go to 6. So we mark 6 as explored. Now there's two places we can go next, we can either go to 3 or we can go to 8 and in general DFS could do either one. Now to generate this fourth set of finishing times I'm going to need to assume that I go to 3 first okay? So again, what DFS does, what we're assuming it does, it starts at 9, and it has to go to 6, it marks those as explored, then it goes to 3. It does not go to 8 first it goes to 3 first. Now, from 3, there's only one outgoing arc which goes to 9, but 9, we've already marked as explored. So it's not going to re-explore 9, it's going to skip that arc. Since that's 3's only outgoing arc, then that for loop completes, and then 3 is the first node to finish. So when we finish with 3, we increment t, it started at 0, now it's 1, and we set the finishing time of 3 to be 1. Just like we said it was in the example. So, now we backtrack to 6. Now we have another outgoing arc from 6 to explore, so now we go to 8. From 8 we have to go to 2, from 2 we have to go to 5, from 5 we have to go 8. 8 we've already seen, so then we're going to be done with 5, because that was its only outgoing arc. So then we increment t, now it's 2, and the finishing time of 5 is going to be 2 as promised. So now we've backtracked to 2, there's no more outgoing arcs from 2. So 2 is going to be the third one that we finish as promised. Then we finish with 8, so the finishing time for 8 is going to be the fourth node to be done as promised. And now I back track back to 6, now at 6, that's the fifth node to be completed as promised. And finally we got all the way back to where we started at 9 and 9 is the sixth node to be completed as promised. Now if we were computing those leaders all of these nodes would get the leader 9, but again the leaders are only relevant for the second pass. So we're just going to ignore the leaders as we're doing this tree so we're just going to keep track of finishing times. So now we're not done so all we did is we finished with the DFS that is invoked from the node 9 and we found 6 of the nodes total in that depth first search. So now we return to the outer for loop and we decrement i. So it started at 9, we're done with that, now we go down to 8. We say, have we already seen 8, yes 8's already explored so we skip it. We go, we decrement i down to 7, we say have we already seen node 7? No we have not okay? 7 is not yet explored. So we invoke DFS now from node sever 7 has two outgoing arcs, it can either go to 4 or it can go to 9. Let's say it checks the outgoing arc to 9 first. Now 9 we already explored. Granted, that was an earlier part of the for loop, but we remember that. We're going to keep track of who got explored on previous iterations of the for loop so we don't bother to re-explore 9, so we skip that. So now from 7 we have to go to 4, from 4 we have to go to 1, from 1 we have to go back to 7. 7's already been exploratory backtrack and now we're done with 1. So 1 is the next one we're completed with and the finishing count of 1 is going to be 7 as promised. We backtrack to 4, there's no more outgoing arcs from 4 to explore, so that's going to be the eighth one to finish. As promised, and the last one to finish is poor node 7. It is last. So that would be an example of how the DFS-Loop subroutine computes finishing times on a reversed graph. So now, let's work through the second pass on the forward version of the graph using the same example. Now remember, the point of the first pass is to compute a magical ordering, and the magical ordering is these finishing times. So now we're going to throw out the original node names, and we're going to replace the node names in blue by the finishing times in red. We're also going to work with the original graphs, which means we have to reverse the arcs back to where they were originally. So those are the two changes you're going to see when I redraw this graph. First of all, all the arcs were reverse orientation. Second of all, all of nodes will change names from their original ones to the finishing times that we just computed. So here's our new graph with the new node names and all of the arcs with their orientation reversed. And now we run DFS again on this graph. And again we're going to process the nodes in order from the highest label 9 down to the lowest label 1. Moreover, we don't need to compute finishing times in the second pass, we only need to do that in the first pass. In the second pass we have to keep track of the leaders, and remember the leader of a vertex Is the vertex from which DFS was called that first discovered that node. All right, so what's going to happen? Well, in the outer for loop, again, we start with i equal to nine, and we invoke DFS from the node 9. So, that's going to be the current leader because that's where the current DFS got initiated. Now, from 9, there's only one choice. We have to go to 7. From 7, there's only one choice, we have to go to 8. From 8, there's only one choice, we have to go back to 9. And then, 9's already been seen, so we backtrack. We go back to 8, we go back to 7, we go back to 9, and that's it. So, when we invoke DFS for node 9, the only things that we encounter are, the nodes 7, 8, and 9. And these are all going to be given the leader vertex 9. You will notice that this is indeed one of the strongly connected components of the graph. We just sort of found it with this indication of DFS from the node 9. So, now we go back to the outer for loop. And we say, okay, let's go to node 8, have we already seen 8? Yes. What about 7, have we already seen 7? Yes. What about 6? Have we have already seen 6? We have not, we have not yet discovered 6, so we invoke DFS from node 6, we reset the global source vortex s to 6. From 6, we can go to 9, we can go to 1. So, let's say we explore 9 first. Well, we already saw 9 in an earlier iteration in the for loop, so we don't explore it again, so, we don't discover 9 now, so we backtrack to 6. We go to 1, from 1, we have to go to 5, and 5 we have to go to 6, and then we start backtracking again. So, the only new nodes that we encounter when we invoke DFS from the node 6 are the vertices 6, 1, and 5. And all of these will have a leader vertex of 6 because that's where we called DFS from when we first discovered these 3 nodes. And you'll notice, this is another FCC of this directed graph. So, we invoke DFS again, not from a new node, the new node 6. And what it discovered, the new nodes it discovered was exactly an FCC of the graph. Nothing more, nothing less. So, now we return to the outer for loop, we go to node 5, have we already seen 5? Yes. Have we already seen 4? No, we haven't seen 4 yet. So, now we invoke DFS from 4. Again, we could try to explore 5, but we've seen that before, we're not going to explore it again. So from 4 then, we have to go to 2, from 2 we have to go to 3, from 3 have to go back to 4, and then, after all the backtracking, we're done. So, the final call to DFS will be from the node 4. And, that DFS will discover precisely, newly discover, precisely the nodes 2, 3 and 4. They will all have the leader vertex 4 because that was where this DFS was called from. It's true we'll go back to the for loop and we'll check have we seen 3 yet? Yes. Have we seen 2 yet? Yes. Have we seen 1 yet? Yes, and then the whole thing completes. And, what we see is that, using the finishing times computed from that first depth first search pass, somehow the strongly connected components of this graph just showed up and presented themselves to us and one at a time on a silver platter. Every time we invoke DFS, the nodes we discovered newly were precisely one of the FCCs, nothing more, nothing less. And, that's really what's going on in this algorithm, turns out this was true in general. The first pass, DFS on the reverse graph, computes finishing times so that if you then process nodes according to decreased order in finishing times. In the second pass, each invocation to DFS will discover one new FCC and exactly one FCC. So, they'll just present themselves to you, one per DFS call in that second pass' for loop. This is, of course, merely an example. You should not just take a single example as proof that this algorithm always works. I will give you a general argument in the next video. But hopefully there is at least a plausibility arcing. No longer does this three-step algorithm seem totally insane. And maybe you could imagine, perhaps it works. At least there's some principles going on. Where you first compute the right ordering to process the nodes, and then the second pass peels off FCCs one at a time like layers from an onion. One thing that I hope is pretty clear is that this algorithm, correct or not, is blazingly fast. Pretty much all you do is two depth per searches. And, since depth per search as we've seen in the past, runs in time linear in the size of the graph, so does Kosaraju's two-pass algorithm. There are a couple subtleties and I encourage you to think about this and you'll be forced to think about this in the program and project for week four. So, for example, in the second pass, how do you process the nodes in decreasing order of finishing time? You don't want to sort the nodes by their finishing time, because that would take n log and time. So, you need to make sure that you remember in the first pass, that you sort of remember the nodes in a way that you can just do a linear scan through them in a second pass. So, there are some details, but, if your intuition is that this is really just double DFS properly implemented, that's pretty much exactly right. So, having spelled out the full implementation, argued that it's definitely a linear time algorithm and given at least a plausibility argument via an example that it might conceivably be correct, let's now turn to the general argument.

# 10-8-Computing Strong Components- The Analysis

So the goal of this video is to prove the correctness of Kasaraju two-pass, depth-first-search based, linear time algorithm that computes the strongly connected components of a directed graph. So I've given you the full specification of the algorithm. I've also given you a plausibility argument of why it might work, in that at least it does something sensible on an example. Namely, it first does a pass of depth first search on the reverse graph. It computes this magical ordering. And what's so special about this ordering is then when we do a depth first search using this ordering on the forward graph; it seems to do exactly what we want. Every indication of depth first search to some new node discovers exactly the nodes of the strong component and no extra stuff. Remember that was our first observation, but that was unclear whether depth for search would be useful or not for computing strong components. If you call depth first search from just the right place, you're gonna get exactly the nodes of an SCC and nothing more. If you call it from the wrong place, you might get all of the nodes of the graph, and get no information at all about the structure of the strong components and at least in this example this first pass with the finishing time seems to be accomplishing seems to be leading us to invoking DFS from exactly the right places. So remember how this worked in the example so in the top graph, I have shown you the graph with the arch reversed. This is where we first invoked DFS loop with the loop over the nodes going from the highest node name nine all the way down to the node name one. And here we compute finishing time that's the bookkeeping that we do in the first pass so we just keep the running count of how many nodes we've finished processing. That is how many we've both explored that node as well as explore all of the outgoing arches and so that gives us these numbers in the red, these finishing times between one and nine for the various nodes. Those became the new node names in the second graph and then we reverse the arches again and get the original graphs back and then we saw that every time we invoked DFS in our second pass we uncovered exactly the nodes of an SCC. So when we invoked it from the node 9 we discovered that 9, 8 and 7 those have a leader vortex 9. Then when we next invoked DFS from 6, we discovered 6, 5, and 1, and nothing else. And then finally we invoked it from 4, and we discovered 2, 3, and 4, and nothing else. And those are exactly the three, SCCs of this graph. So let's now understand why this works in any directed graph, not just this, in this one example. So let's begin with a simple observation about directed graphs, which is actually interesting in its own right. The claim is that every directed graph has two levels of granularity. If you squint, if you sort of zoom out, then what you see is a directed acyclic graph, of course comprising its strongly connective components. And if you want, you can zoom in and focus on the fine grain structure with one SCC. A little bit more precisely. The claim is of the strongly connected components of a directed graph induce in a natural way an acyclic metagraph. So what is this metagraph? What are the nodes and what are the arcs, what are the edges? Well, the metanodes are just the SCCs, so we think of every strong connected component as being a single node in this metagraph. So call them say 'C1' up to 'Ck'. So what are the arcs in this metagraph? Well, they're basically just the ones corresponding to the arcs between SCCs in the original graph. That is, we include in the meta graph an arc from the strong component 'C' to 'C-hat' if and only if there's an arc from a node in 'C' to a node in 'C-hat' in the original graph 'G'. So for example if this is your 'C' and so the triangle is your 'C-hat' and you have one or maybe multiple edges going from 'C' to 'C-hat', then in the corresponding metagraph your just gonna have a node for 'C', a node for 'C-hat' and the directed arch from 'C' to 'C-hat'. So if we go back to some of the directed graphs that we've used as running examples so we go back to the beginning of the previous video and it's look maybe something like this the corresponding directed acyclic graph has four nodes and four arches. And for the running example we used to illustrate Kosaraju's algorithm with the three triangles, the corresponding metagraph would just be a path with three nodes. So why is this meta-graph guaranteed to be acyclic? Well, remember metanodes correspond to strong components, and in a strong component you can get from anywhere to anywhere else. So, if you had a cycle that involved two different metanodes, that is two different strong connected components, remember on a directed cycle you can also get from anywhere to anywhere else. So if you had two supposedly distinct SCCs, that you could get from the one to the other and vice versa, they would collapse into a single SCC. You can get from anywhere to anywhere in one, anywhere from anywhere in the other one, and you can also go between them at will, so you can get from anywhere in this union to anywhere in the union. So not just in this context of competing strong components but also just more generally, this is a useful fact to know about directed graphs. On the one hand, they can have very complex structure within a strong components. You have paths going from everywhere to everywhere else, and it may be sort of complicated looking. But at a higher level, if you abstract out to the level of SCCs, you are guaranteed to have this simple DAG, this simple directed acyclic graph structure. So, to reinforce these concepts, and also segue into thinking about Kosaraju's algorithm in particular, let me ask you a question about how reversing arcs affects the strong components of a directed graph. So the correct answer to this quiz is the fourth one. The strong components are exactly the same as they were before, in fact the relation that we described is exactly the same as it was before so therefore the equivalence classes of the strong components is exactly the same. So if two nodes were related in the original graph, that is a path from U to V and a path from V to U, that's still true after you reverse all the arcs, you just use the reversal of the two paths that you had before. Similarly if the two nodes weren't related before, for example because you could not get from U to V, well that after you reverse everything, then you can't get from V to U, so again you don't have this relation holding, so the SCCs are exactly the same in the forward or the backward graph. So in particular in Kazarogi's algorithm, the strong component structure is exactly the same in the first pass of DFS and in the second pass of DFS. So now that we understand how every directed graph has a meta graph with the nodes correspond to a strong connected components, and you have an arch from one SCC to another if there's any arch from any node in that SCC to the other SCC in the original graph, I'm in a position to state what's the key lemma. That drives the correctness of Kosaraju's two pass algorithm for computing the strong connected component of a directed graph. So here's the lemma statement. It considers two strongly connecting components that are adjacent, in the sense that there's an arc from one node in one of them to one node in the other one. So let's say we have one SCC - 'C1', with a node I, and another, SCC 'C2' with a node J, and that in G, in the graph, there's an arc directly from I to J. So in this sense we say that these SCCs are adjacent, with the second one being in some sense after the first one. Now let's suppose we've already run the first pass of the DFS loop subroutine. And remember that works on the reverse graph. So we've invoked it on the reverse graph. We've computed these finishing times. As usual we'll let f(v) denote the finishing times computed in that depth first search subroutine on the reverse graph. The lemma then asserts the following. It says first, amongst all the nodes in 'C1' look at the one with the largest finishing time. Similarly amongst all nodes in 'C2' look at the one with the biggest finishing time. Amongst all of these the claim is that the biggest finishing time will be in 'C2' not in 'C1'. So what I wanna do next is I wanna assume that this lemma is true temporarily. I wanna explore the consequences of that assumption and in particular what I wanna show you is that if this lemma holds, then we can complete the proof of correctness of Kosaraju's two-pass SCC computation algorithm. Okay, so if the lemma is true then after... I'll give you the argument about why we're done. About why we just peel off the SCC one at a time with the second pass of depth first search. Now of course a proof with a hole in it, isn't a proof. So at the end of the lecture I'm gonna fill in the hole. That is, I'm gonna supply a proof of this key lemma. But for now, as a working hypothesis, let's assume that it's true. Let's begin with a corollary, that is a statement which follows essentially immediately, from the statement of a lema. So for the corollary, let's forget about just trying to find the maximum maximum finishing time in a single SCC. Let's think about the maximum finishing time in the entire graph. Now, why do we care about the maximum finishing time in the entire graph? Well, notice that's exactly where the second pass of DFS is going to begin. Right, so it processes nodes in order from largest finishing time to smallest finishing time. So equivalently, let's think about the node at which the second pass of depth first search is going to begin, i.e., the node with the maximum finishing time. Where could it be? Well, the corollary is that it has to be in what I'm gonna call a sink, a strongly connected component, that is a strongly connected component without any outgoing arcs. So for example let's go back to the, meta graph of SCCs for the very first directed graph we looked at. You recall in the very first direc ted graph we looked at in when we started talking about this algorithm there were four SCCs. So there was a 'C1', a 'C2', a 'C3', and a 'C4'. And of course within each of these components, there could be multiple nodes but they are all strongly connected to each other. Now, let's use F1, F2, F3 and F4 to denote the maximum finishing time in each of these SCCs. So we have F1, F2, F3 and F4. So, now we have four different opportunities to apply this lemma. Right? Those four different pairs of adjacent SCCs. And so, what do we find? We find that well, comparing F1 and F2, because C2 comes after C1, that is there's an arc from C1 to C2, the max finishing time in C2 has to be bigger than that in C1. That is F2 is bigger than F1. For the same reasoning F3 has to be bigger than F1. Symmetrically we can apply the limit to the pair C2, C4 and C3, C4 and we get that F4 has to dominate both of them. Now notice we actually have no idea whether F2 or F3 is bigger. So that pair we can't resolver. But we do know these relationships. Okay F1 is the smallest and F4 is the smallest [biggest!!]. And you also notice that C4 is a sink SCC and the sink has no outgoing arches and you think about it that's a totally general consequence of this lema. So in a simple group of contradiction will go as follows. Consider this SCC with the maximum F value. Suppose it was not a sink SCC that it has an outgoing arch, follow that outgoing arch to get some other SCC by the lema the SCC you've got into has even bigger maximum finishing time. So that contradicts the fact that you started in the SCC with a maximum finishing time. Okay. So just like in this cartoon, where the unique sink SCC has to have the largest finishing time, that's totally general. As another sanity check, we might return to the nine node graph, where we actually ran Kasaraja's algorithm and looking at the ford version of the graph, which is the one on the bottom, we see that the maximum finishing times in the three FCC are 4,6 and 9. And it turns out that the same as the leader nodes which is not an accident if you think about it for a little while and again you'll observe the maximum finishing time in this graph namely 9 is indeed in the left most SCC which is the only SCC with no outgoing arks. Okay but it's totally general basically you can keep following arks and you keep seeing bigger and bigger finishing times so the biggest one of all it has to be somewhere where you get stuck where you can't go forward but there's no outgoing arks and that's what I'm calling a sink SCC. Okay. So assuming the lemma is true we know that the corollary is true. Now using this corollary let's finish the proof of correctness, of Kasaraja's algorithm, module over proof of the key lima. So I'm not going to do this super rigorously, although everything I say is correct and can be made, made rigorous. And if you want a more rigorous version I'll post some notes on the course website which you can consult for more details. So what the previous corollary accomplished, it allows us to locate, the node with maximum finishing time. We can locate it in somewhere in some sink SCC. Let me remind you about the discussion we had at the very beginning of talking about computing strong components. We're tryna understand depth-first search would be a useful workhorse for finding the strong components. And the key observation was that it depends, where you begin that depth-first search. So for example in this, graph with four SCC's shown in blue on the right. A really bad place to start. DFS called Depth For Search would be somewhere in C1. Somewhere in this source SCC, so this is a bad DFS. Why is it bad? Well remember what Depth For Search does; it finds everything findable from its starting point. And from C1 you can get to the entire world, you can get to all the nodes in the entire graph. So you can discover everything. And this is totally useless because we wanted to discover much more fine-grain structure. We wanted to discover C1, C2, C3 and C4 individually. So that would be an disaster if we invoked depth first search somewhere from C1. Fortunately that's not what's going to happen, right? We computed this magical ordering in the first pass to insure that we look at the node with the maximum finishing time and, by the corollary, the maximum finishing time is going to be somewhere in C4. That's gonna be a good DFS, in the sense that, when we start exploring from anywhere in C4, there's no outgoing arcs. So, of course, we're gonna find everything in C4. Everything in C4's strongly connected to each other. But we can't get out. We will not have the option of trespassing on other strong components, and we're not gonna find'em. So we're only gonna find C4, nothing more. Now, here's where I'm gonna be a little informal. Although, again, everything I'm gonna say is gonna be correct. So what happens now, once we've discovered everything in C4? Well, all the nodes in C4 get marked as explored, as we're doing depth first search. And then they're basically dead to us, right? The rest of our depth first search loop will never explore them again. They're already marked as explored. If we ever see'em, we don't even go there. So the way to think about that is when we proceed with the rest of our for loop in DFS loop it's as if we're starting afresh. We're doing depth first search from scratch on a smaller graph, on the residual graph. The graph G with this newly discovered strong component 'C<i>'</i> deleted. So in this example on the right, all of the nodes in C4 are dead to us and it's as if we run DFS anew, just on the graph containing the strong components C1, C2 and C3. So in particular, where is the next indication of depth first search going to come from? It's going to come from some sink SCC in the residual graph, right? It's going to start at the node that remains and that has the largest finishing time left. So there's some ambiguity in this picture. Again recall we don't know whether F2 is bigger or F3 is bigger. It could be either one. Maybe F2 is the largest remaining finishing time in which case the next DFS indication's gonna begin somewhere more from C2. Again, the only things outgoing from C2 are these already explored nodes. Their effectively deleted. We're not gonna go there again. So this is essentially a sink FCC. We discover, we newly discover the nodes in C2 and nothing else. Those are now effectively deleted. Now, the next indication of DFS will come from somewhere in F3, somewhere in C3. That's the only remaining sink SCC in the residual graph. So the third call, the DFS will discover this stuff. And now, of course, we're left only with C1. And so the final indication of DFS will emerge from and discover the nodes in C1. And in this sense because we've ordered the nodes by finishing times when DFS was reverse graph, that ordering has this incredible property that when you process the nodes in the second pass we'll just peel off the strongly connected components one at a time. If you think about it, it's in reverse topological order with respect to the directed asypric graph of the strongly connected components. So we've constructed a proof of correctness of Kosaraju's, algorithm for computing strongly connected components. But again, there's a hole in it. So we completed the argument assuming a statement that we haven't proved. So let's fill in that last gap in the proof, and we'll we done. So what we need to do is prove the key lemma. Let me remind you what it says. It says if you have two adjacent SCCs, C1 and C2 and is an arc from a node in C1, call it 'I' to a node in C2, say J. Then the max finishing time in C2 is bigger than the max finishing time in C1. Where, as always, these finishing times are computed in that first pass of depth-first search loop in the reversed graph. All right, now the finishing times are computed in the reversed graph, so let's actually reverse all the arcs and reason about what's happening there. We still have C1. It still contains the node I. We still have C2, which still contains the node J. But now of course the orientation of the arc has reversed. So the arc now points from J to I. Recall we had a quiz which said, asked you to understand the effect of reversing all arcs on the SCC's and in particular there is no effect. So the SCC's in the reverse graph are exactly the same as in the forward graph. So now we're going to have two cases in this proof and the cases correspond to where we first encounter a node of C1 and union C2. Now remember, when we do this DFS loop, this second pass, because we have this outer four loop that iterates over all of the nodes we're guaranteed to explore every single node of the graph at some point. So in particular we're gonna have to explore at some point every node in C1 and C2. What I want you to do is pause the algorithm. When it first, for the first time, explores some node that's in either C1 or C2. There's going to be two cases, of course, because that node might be in C1, you might see that first. Or it might be in C2, you might see something from C2 first. So our case one is going to be when the first node that we see from either one happens to lie in C1. And the second case is where the first node V that we see happens to lie in C2. So clearly exactly one of these will occur. So let's think about case one. When we see a node of C1 before we see any nodes of C2. So in this case where we encounter a node in C1 before we encounter any node in C2, the claim is that we're going to explore everything in C1 before we ever see anything in C2. Why is that true? The reason is there cannot be a path that starts somewhere in C1, for example, like the vertex V, and reaches C2. This is where we are using the fact that the meta-graph on SCC is a cyclic. Right C1 is strong connected, C2 is strong connected, you can get from C2 to C1 and, if you can also get from C1 back to C2 this all collapses into a single strongly connected component. But that would be a contraction, we're assuming C1 and C2 are distinct strongly connected components, therefore you can't have paths in both directions. We already have a path from right to left, via JI, so there's no path from left to right. That's why if you originate a depth first search from somewhere inside C1 like this vertex V, you would finish exploring all of C1 before you ever are going to see C2, you're. Only gonna see C2 at some later point in the outer for loop. So, what's the consequence that you completely finish with C1 before you ever see C2? Well it means every single finishing time in C1 is going to be smaller than every single finishing time in C2. So that's even stronger that what we're claiming, we're just claiming that the biggest thing in C2 is bigger than the biggest of C1. But actually finishing times in C2 totally dominate those in C1, because you finish C1 before you ever see C2. So let's now have a look at case one actually in action. Let's return to the nine-node graph, the one that we actually ran Kosaraju's algorithm to completion. So if we go back to this graph which has the three connected components, then remember that the bottom version is the forward version, the top version is the reversed version. So if, if you think about the middle SCC as being c1, pulling the row of c1 and the left most. Scc playing the role of C2, then what we have exactly is case one of the key lemma. So, which was the first of these six vertices visited during the DFS loop in the reversed graph? Well that would just be the node with the highest name, so the node nine. So this was the first of these six vertices that depth first search looked at in the first pass, that lies in what we're calling C1. And indeed everything in C1 was discovered in that pass before anything in C2 and that's why all of the finishing times in C2, the 7,8,9 are bigger than all of the finishing times in C1 - the 1,5, and 6. So we're good to go in case two. We've proven sorry, in case one, we've proven the lemma. When it's the case that, amongst the vertices in C1 and C2, depth first search in the first pass sees something from C1 first. So now, let's look at this other case, this grey case, which could also happen, totally possible. Well, the first thing we see when depth first searching in the first pass, is something from C2. And here now is where we truly use the fact that we're using a depth first search rather than some other graph search algorithm like breadth first search. There's a lot of places in this algorithm you could swap in breadth first search but in this case two, you'll see why it's important we're using depth first search to compute the finishing times. And what's the key point? The key point is that, when we invoke depth first search beginning from this node V, which is now assuming the line C2. Remember depth first search will not complete. We won't be done with V until we've found everything there is to find from it, right? So we recursively explore all of the outgoing arcs. They recursively explore all of the outgoing arcs, and so on. It's only when all paths going out of V have been totally explored and exhausted, that we finally backtrack all the way to V, and we consider ourselves done with it. That is, depth first search. In the reverse graph initiated at v. Won't finish until everything findable has been completely explored. Because there's an arc from C2 to C1, obviously everything to C2 is findable from V, that's strongly connected. We can from C2 to C1 just using this arc from J to I. C1 being strongly connected we can then find all of that. Maybe we can find other strongly connected components, as well, but for sure depth-first search starting from V will find everything in C1 to C2, maybe some other things. And we won't finish with V until we finish with everything else, that's the depth-first search property. For that reason the finishing time of this vertex V will be the largest of anything reachable from it. So in particular it'll be larger than everything in C two but more to the point, it'll be larger than everything in C1 which is what we are trying to prove. Again let's just see this quickly in action in the nine node network on which we traced through Kosaraju's algorithm. So to show the rule that case two is playing in this concrete example let's think of the right most strongly connected component as being C1. And let's think of the middle SCC as being C.2. Now the last time. We called the middle one C1 and the leftmost one C2. Now we're calling the rightmost one C1 and the middle one C2. So again, we have to ask the question, you know, of the six nodes in C1 and in C2, what is the first one encountered in the depth first search that we do in the first pass. And then again, is the node nine? The, the node which is originally labeled not. So it's the same node that was relevant in the previous case, but now with this relabeling of the components, nine appears in the strongly connected component C-2, not in the one labeled C-1. So that's the reason now we're in case two, not in case one. And what you'll see is, what is the finishing time that this originally labeled nine node gets. It gets the finishing time six. And you'll notice six is bigger than any of the other finishing times of any of the other nodes in C1 or C2. All, the other five nodes have the finishing times one through five. And that's exactly because when we ran depth first search in the first pass, and we started it at the node originally labeled nine, it discovered these other five nodes and finished exploring them first before finally back tracking all the way back to nine, and deeming nine fully explored. And it was only at that point that nine got its finishing time after everything reachable from it had gotten there lower finishing times. So that wraps it up. We had two cases depending on whether in these two adjacent SCC's, the first vertex encountered was in the C1, or in C2. Either way it doesn't matter, the largest finishing time has to be in C2. Sometimes it's bigger than everything, sometimes it's just bigger than the biggest in C-1, but it's all the same to us. And to re cap how the rest of the proof goes, we have a corollary based on this lemma, which is maximum finishing time have to lie in sink SCC And that's exactly where we want our depth first search to initiate. If you're initiated in a strong component with no outgoing arcs, you do DFS. The stuff you find is just the stuff and that strongly connected component. You do not have any avenues by which to trespass on other strong components. So you find exactly one SCC. In effect, you can peel that off and recurse on the rest of the graph. And our slick way of implementing this recursion is to just do the single second, DFS pass, where you just treat the nodes in decreasing order of finishing times, that in effect, unveil, unveils all of the SCCs in reverse topological ordering. So that's it, Kosaraju's algorithm, and the complete proof of correctness. A blazingly fast graph primitive that, in any directed graph, will tell you its strong components.

# 10-9-Structure of the Web [Optional]

So, we've now put in a lot of work designing and analyzing super fast algorithms for reasoning about graphs. So in this optional video, what I want to do is show you why you might want such a primitive, especially for computation on extremely large graphs. Specifically, we're going to look at the results of a famous study that computes the strongly connected components of the web graph. So what is the web graph? Well it's the graph in which the vertices correspond to webpages. So for example I have my own webpage where I list my research papers and also links to courses, such as this one. And the edges are going to be directed and they correspond precisely to hyperlinks. So the links that bring you from one web page to another. Note, of course, these are directed edges, where the tail is the page that contains the hyperlink. And the head is the page that you go to if you click the hyperlink. And so, this is a directed graph. So from my home page you can get to my papers, you can get to my courses. Sometimes I have random links up to things I like, say, my favorite record store. And of course for many of these webpages, there are additional links going out or going in. So for example, from my papers I might link to some of my co-authors. Some of my co-authors might be linking from their homepages to me. Or of course, there's webpages out there which list the currently available free online courses and so on. So obviously, this is just part of a massive web graph, just a tiny, tiny piece of it. So the origins of the web were probably around 1990 or so, but it started to really explode in the mid 90s. And by the year 2000, it was sort of already beyond comprehension. Even though, in Internet years, the year 2000 is sort of the stone age, relative to right now, relative to 2012. But still, even by 2000 people were so overwhelmed with the massive scale of the web graph, they wanted to understand anything about it, even the most basic things. Now of course, one issue with understanding what the graph looks like is you don't even have it locally, right? It's distributed over all of these different servers over the entire world. So the first thing people really focused on, when the wanted to answer this question, was on techniques for crawling. So having software which just follows lots of hyperlinks, reports back to the home base, from which you can assemble at least some kind of sketch of what this graph actually is. But then the question is, even once you have this crawled information, even once you've accessed a good chunk of the nodes and the edges of this network, what does it look like? So what makes this a difficult question, more difficult than, say, for any other directed graph you might encounter? Well, it's simply the massive scale of the web graph, it's just so big. So for the graph used in the particular study I'm going to discuss, like we said, it was in the year 2000, which is sort of the stone age compared to 2012. So the graph was small, relatively, but still the graph was really, really big. So it was something like 200 million nodes and one billion edges, really one and a half billion edges. So the reference for the work I'm going to discuss is a paper by a number of authors. The first author is Andre Broder and then he has many co-authors and this was a paper that appeared in the WWW Conference of the year 2000, that's the World Wide Web Conference. And I encourage to those of you who are interested to go track down the paper online and read the original source. So Andre Broder, the lead author, at this time he was at a company that was called Alta Vista. So how many of you remember a company called Alta Vista? Well, some of you, especially the youngest ones among you maybe have never heard of Alta Vista. And the youngest ones among you maybe can't even conceive of a world in which we didn't have Google. But in fact, there was a time when we had web search, but Google did not yet exist. That was sort of in the maybe '97 or so, and so this is in the very embryonic years of Google, and this data set actually came out of Alta Vista instead. So Broder et al wanted to get some answers to this question, what does this web graph look like? And they approached it in a few ways. But the one I'm going to focus on here is, they asked, well, what's the most detailed structure we can get about this web graph, without doing an infeasible amount of computation? Really just sticking to linear time algorithms, at the worst. And, what have we seen? We've seen that in a directed graph you can get full connectivity information just really using depth first search. That you can compute strongly connected components in linear time with small constants. And that's one of the major things that they did in this study. Now if you wanted to do the same computation today, you'd have one thing going against you and one thing going for you. The obvious thing that you'd have going against you is that the web is still very much bigger than it was here, certainly by an order of magnitude. The thing that you'd have going for you is now there's specialized systems which are meant to operate on massive data sets. And in particular, they can do things like compute connectivity information on graph data. So what you have to remember, for those of you who are aware of these terms, in 2000 there was no MapReduce, there was no Hadoop. There were no tools for automated processing large data sets, these guys really had to do it from scratch. So let me tell you about what Broder et al found when they did strong connectivity computations on the web graph. They explain their results in what they called the bow tie picture of the web. So let's begin with the center, or the knot of the bow tie. So in the middle we have what we're going to call a giant strongly connected component. With the interpretation being this is the core of the web in some sense. All right, so all of you know what an SCC is at this point. A strongly connected component is a region from which you can get from any point to any other point along a directed path. So in the context of the web graph, with this giant SCC, what this means is that from any webpage inside this blob, you can get to any other webpage inside this blob, just by traversing a sequence of hyperlinks. And hopefully it doesn't strike you as too surprising that a big chunk of the web is strongly connected, is well-connected in this sense, right? So if you think about all the different universities in the world, probably all of the webpages corresponding to all of the different universities, you can get from any one place to any other place. For example, from the homepage on which I put my papers, I often include links to my co-authors, which very commonly are at other universities. So that already provides a web link from some Stanford page to some page at say, Berkeley or Cornell or whatever. And of course, I'm just one person, I'm just one of many faculty members at Stanford. So you put all of these together, you would expect all of the different SCCs corresponding to different universities to collapse into a single one. And so on for other sectors as well. And then of course, if you knew that a huge chunk of the web was in the same strongly connected component, so let's say 10% of the web, which would be tens of millions of webpages. You wouldn't expect there to be a second one, right? It would be super weird if there were two different blobs, 10 million web pages each that somehow were not mutually reachable from each other. All it takes to collapse two SCCs into one is a lone arc going from one to the other and then a lone arc going in the reverse direction. And then those two SCCs collapse into one. So we do expect a giant SCC, just sort of thinking anecdotally about what the web looks like. And then once we realize there's one giant SCC, we don't expect there to be more than one. All right, so is that the whole story? Is the web graph just one big SCC? Well, one of the perhaps interesting findings of this Broder et al paper is that there is a giant SCC, but it doesn't actually take up the whole web, or anything really that close to the entire web. So what else would there be in such a picture? Well, there's the other two ends of the bow tie, which are called the in and the out regions. In the out regions you have a bunch of strongly connected components, not giant SCCs. We've established their shouldn't be any other giant SCCs. But small SCCs, which you can reach from the giant strongly connected component. But from which you cannot go back to the giant strongly connected component. I encourage you to think about what types of websites you would expect to see in this out part of the bow tie. I'll give you one example. Very often if you look at a corporate site, including those of well known corporations, which you would definitely expect to be reachable from the giant SCC. It's actually a corporate policy that no hyperlinks can go from something in the corporate site to something outside the corporate site. So that means the corporate site is going to be a collection of webpages, which is certainly strongly connected. Because it's a major corporation, you can certainly get there from the giant SCC. But because of its corporate policy, you can't get back out. Symmetrically, in the in part of the bow tie, you have strongly connected components, generally small ones. From which you can reach the giant SCC, but you cannot get to them from the giant SCC. Again, I encourage you to think about all the different types of webpages you might expect to see in this in part of the bow tie. Certainly I think one really obvious example would be new webpages. So if you just create something, and then if I just created a webpage and pointed it to Stanford University, that would immediately be in this in component or this in collection of components. Now, if you think about it, this does not exhaust all of the possibilities of where nodes can lie. There's a few other cases that frankly are pretty weird, but they're there. You can have passive hyperlinks, which bypass the giant SCC and go straight from the in part of the bow tie to the out part. So Broder et al suggested calling these tubes. And then there's also a kind of very curious outgrowths going out of the in component, but which don't make it all the way to the giant SCC. And similarly, there's stuff which goes into the out component. And Broder et al recommended calling these strange creatures tendrils. And then, in fact, you can just have some weird isolated islands of SCCs that are not connected, even weakly, to the giant SCC. So this is the picture that emerged from Broder et al's strong component computation on the web graph. And here's, qualitatively, some of the main findings that they came up with. So first of all, that picture on the previous slide I drew roughly to scale. In the sense that all four parts, so the giant SCC, the in part, the out part, and then the residual stuff, the tubes and tendrils, have roughly the same size, more or less 25% of the nodes in the graph. I think this is surprising people. I think some people might have thought that the core, that the giant SCC might have been a little bit bigger than just 25 or 28%. But it turns out there's a lot of other stuff outside of this strongly connected core. You might wonder if this is just an artifact of this data set being from the stone age, being from 2000 or so. But people have rerun this experiment on the web graph again in later years. And of course the numbers are changing because the graph is growing rapidly. But these qualitative findings have seemed pretty stable throughout subsequent reevaluations of the structure of the web. On the other hand, while the core of the web is not as big as you might have expected, it's extremely well connected. Perhaps better connected then you might have expected. Now you'd be right to ask the question, what could I mean by unusually well connected? We've already established that this giant core of the web is strongly connected. You can get from any one place to any other place via a sequence of hyperlinks. What else could you want? Well in fact, it has a very richer notion of connectivity called the small world property. So let me tell you about the small world property or the phenomenon colloquially known as six degrees of separation. So this is an idea that had been in the air at least since the early 20th century, but really kind of was studied in a major way and popularized by Stanley Milgram, who's a social scientist, back in 1967. So, Milgram was interested in understanding, are people at great distance, in fact, connected by a short change of intermediaries? So, the way he evaluated this, he ran the following experiment. He identified a friend in Boston, Massachusetts, a doctor, I believe. And so this was going to be the target. And then he identified a bunch of people who were thought to be far away, both culturally and geographically, specifically Omaha. So for those of you who don't live in the US, just take it on faith that many people in the US would regard Boston and Omaha as being fairly far apart, geographically and otherwise. And what did is he wrote each of these residents of Omaha the following letter. He said, look, here is the name and address of this doctor who lives in Boston. Your job is to get this letter to this doctor in Boston. Now, you're not allowed to mail the letter directly to the doctor. Instead, you need to mail it to an intermediary, someone who you know on a first name basis. So of course, if you knew the doctor on a first name basis, you could mail it straight to them, but that was very unlikely. So what people would do in Omaha is they'd say, well, I don't know any doctors or I don't know anyone in Boston, but at least I know somebody in Pittsburgh. And at least that's closer to Boston than Omaha, that's further eastward. Or maybe someone would say, well, I don't really know anyone on the east coast but at least I do know some doctors here in Omaha. And so they'd give the letter to somebody that they knew on a first name basis in Omaha. And then the situation would repeat. Whoever got the letter, again they'd be given the same instructions. If you know this doctor in Boston on a first name basis, send him the letter. Otherwise, pass the letter on to somebody who seems more likely closer to them than you are. Now, of course, many of these letters never reach their destination, but shocking, at least to me, is that a lot of them did. So something like 25%, at least, of the letters that they started with made it all the way to Boston. Which I think says something about people in the late 60s just having more free time on their hands than they do in the early 21st century. I find this hard to imagine, but it's a fact. So you had dozens and dozens of letters reaching this doctor in Boston. And they were able to trace exactly which path of individuals the letter went along before it eventually reached this doctor in Boston. And so then what they did is they looked at the distribution of chain links. So how many intermediaries were required to get from some random person in Omaha to this doctor in Boston? Some were as few as two, some were as big as nine but the average number of hops, the average number of intermediaries, was in the range of five and a half or six. And so this is what has given rise to the colloquialism, even the name of a popular play, the six degrees of separation. So that's the origin myth, that's where this phrase comes from, these sort of experiments with physical letters. But now in network science, the small world property is meant to be a network. Which on the one hand is richly connected, but also in some sense there are enough cues about which links are likely to get closer to some target. So that if you need to route information from point a to point b, not only is there a short path, but if you in some sense follow your nose, then you'll actually exhibit a short path. So in some sense, routing information is easy in small world networks. Now this is exactly the property that Broder et al identified within this giant SCC. Very rich with short paths, and if you want to get from point a to point b, just follow your nose and you'll do great. You don't need a very sophisticated shortest path algorithm to find a short path. Some of you may have heard of Sammy Milgram, not for the small world experiment, but for another famous, or maybe infamous experiment he did earlier in the 60s. Which consisted into tricking volunteers into thinking they are subjecting other human beings to massive doses of electric shocks. So that wound up causing a rewrite to certain standards of ethics in experimental psychology. You don't hear about that so much when people are talking about networks, but that's another reason why Milgram's work is well known. And just as a point of contrast, outside of this giant strongly connected component, which has this rich small world structure, very poor connectivity in the other parts of the web graph. So there's lots of cool research going on these days about the study of information networks like the web graph. So I don't want you to get the impression that the entire interaction between algorithms and thinking about information networks has just been this one strongly connected component computation in 2000. Of course, there's all kinds of interactions. I've just singled one out that was easy to explain and also highly intellectual and interesting back in the day. But these days, lots of stuff's going on. People are thinking about information networks in all kinds of different ways and of course algorithms, like in almost everything, is playing a very fundamental role. So let me just dash off sort of a few examples, maybe to whet your appetite. Maybe you want to go explore this topic in greater depth outside of this course. So one super interesting question is, rather than looking at a static snapshot of the web, like we were doing so far in this video, the web's changing all the time. New pages are getting created, new links are getting created and destroyed and so on. And how does this evolution proceed? Can we have a mathematical model, which faithfully reproduces the most important first order properties of this evolutionary process? So a second issue is to think not just about the dynamics of the graph itself, but the dynamics of information that gets carried by the graph. And you could ask this both about the web graph and about other social networks, like say, Facebook or Twitter. Another really important topic, which there's been a lot of work on, but we still don't fully understand by any means, is getting at the finer grain structure in networks including the web graph. In particular, what we'd really like to do is have foolproof methods for identifying communities. So groups of nodes, this could either be webpages in the web graph or individuals in a social network, which we should think of as grouped together. We discussed this a little bit when we talked about applications of cuts. One motivation for cuts is to identify communities, if you think of communities as being relatively densely connected inside and sparsely connected outside. But that's just a baby step. Really, we need much better techniques for both defining and computing communities in these kinds of networks. So I think these questions are super interesting, both from a mathematical/technical level, but also they're very timely. Answering them really helps us understand our world better. Unfortunately, these are going to be well outside the course of just the bread-and-butter design analysis of algorithms, which is what I'm tasked with covering here. But I will leave you with a reference book that I recommend if you want to read more about these topics. Namely, the quite recent book by David Easley and John Kleinberg called Networks, Crowds, and Markets.

# zwk5-Optional

In the 2SAT problem, you are given a set of clauses, where each clause is the disjunction of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value "true" or "false" to each of the variables so that all clauses are satisfied --- that is, there is at least one true literal in each clause. For this problem, design an algorithm that determines whether or not a given 2SAT instance has a satisfying assignment. (Your algorithm does not need to exhibit a satisfying assignment, just decide whether or not one exists.) Your algorithm should run in O(m+n) time, where m and n are the number of clauses and variables, respectively. [Hint: strongly connected components.]

# zzwk5-Prog

The file contains the edges of a directed graph. Vertices are labeled as positive integers from 1 to 875714. Every row indicates an edge, the vertex label in first column is the tail and the vertex label in second column is the head (recall the graph is directed, and the edges are directed from the first column vertex to the second column vertex). So for example, the 11th row looks liks : "2 47646". This just means that the vertex with label 2 has an outgoing edge to the vertex with label 47646 Your task is to code up the algorithm from the video lectures for computing strongly connected components (SCCs), and to run this algorithm on the given graph. Output Format: You should output the sizes of the 5 largest SCCs in the given graph, in decreasing order of sizes, separated by commas (avoid any spaces). So if your algorithm computes the sizes of the five largest SCCs to be 500, 400, 300, 200 and 100, then your answer should be "500,400,300,200,100" (without the quotes). If your algorithm finds less than 5 SCCs, then write 0 for the remaining terms. Thus, if your algorithm computes only 3 SCCs whose sizes are 400, 300, and 100, then your answer should be "400,300,100,0,0" (without the quotes). (Note also that your answer should not have any spaces in it.) WARNING: This is the most challenging programming assignment of the course. Because of the size of the graph you may have to manage memory carefully. The best way to do this depends on your programming language and environment, and we strongly suggest that you exchange tips for doing this on the discussion forums.