in this lecture we're going to extend our discussion of search to methods that focus on depth and note some fairly counter-intuitive elements of that setting so just to remind you of what we've done we have sketched out a general method that we would approach search problem with and we'll start with one state that we're familiar with the initial state and we'll just repeatedly take the highest priority state we've encountered and the priority is vague here and we'll expand it and what expanding it means is adding every possible next state from that state into our space of consideration so we'll have this notion that a state can be have been encountered and be under consideration but not yet expanded and those are the ones that we will be selecting among and we'll be expanding one of those which means look at where we can get from that state and add it into consideration thus having a richer set to choose from when we do the next loop so by choosing the notion of priority we get different algorithms and we'll do this over and over until we expand a goal and this expand is in parentheses because we could wait we could notice a goal if we simply encounter it but haven't expanded it yet that's as soon as it comes into consideration for this step even though we haven't selected it yet we could notice it but optimality finding the shortest possible path or lowest cost path to the goal will usually require waiting until we choose to expand it because we might later find a cheaper path and we looked at those methods under breadth first focus so to get a death first focus what we have to do is prefer to keep expanding right where we just expanded like expand a child or a successor of the state we just expanded so to focus on depth we're going to give a notion of priority into this selection take the most recently encountered the one we most recently discovered and brought into consideration but haven't expanded yet of course there could be ties when we discover when we expand a node all of its children come into consideration sort of simultaneously and we'll pick one of those [Music] and so this means our q or our data structure for keeping track of the priority and all the all the encountered states is going to be what's called a stack you've all presumably encountered stacks it's a last in first out way of storing a collection so the last state we newly encountered and put into the stack comes back out first in this selection and that causes us to have a depth focus in contrast to this picture here where we expand this start state and all of these come into consideration and then we expand each one of them before we go out here and start expanding that's breadth focus and that's what will happen in a breadth first search but in a depth focused environment we'll expand the start state and all of these one step here come into consideration these four we'll pick one of them and expand it but then this is where it differs we won't go back and do these we'll say oh we want to expand a child of this one so we'll expand one of these let's say this one and this expansion will keep charging off down away and these green marked states have come into consideration a long time ago but have not been expanded yet because we're too busy diving off down here that's a depth focus now both of these methods keep considering new states so you might ask why would we prefer one versus the other and i like to do a little thought experiment that isn't a perfect metaphor so be cautious with it but you come to move into a new home in a brand new community where you know nothing and you come to your home you have no map you have no gps but you want to go to a grocery and you figure there must be a grocery somewhere nearby so you head off looking for one now do you sort of look in a breath first manner looking around your home or do you charge off and just head out go off like this head off further and further and further away from your home i presume your intuition like mine is that you'll do something a little more spirally now you can't simulate a random access memory and expand here and then expand there and then expand here and then expand there that's not something you can do in physical reality you'll try to stay near home while you're looking and then a little further from home and then a little further for a moment there's a reason for that and that's that you not only want to find a grocery you want to get back home afterwards so you'd like the grocer you find to be near your home and this method will explore just as much terrain i might even be exploring new terrain more quickly because there's some cost to this spirally pattern of you know retracing steps perhaps this doesn't retrace anything but it finds a grocery you know that's 60 miles away and that's not usually your first choice and so this is i'm trying to develop your intuition that a depth-first approach is sacrificing optimality we're not finding the optimal path to a goal if there's more than one goal there's only one goal there's you know we may we even then we might find a really long path maybe our long depth first focused path will wrap back around and come to this vertex we could have found it here but we found this really long path to it so even if there's

only one goal we may we may find a terrible path to it so that would seem like an enormous drawback and we're going to remedy that first let's let's first talk about some of the properties that this algorithm benefits from notice that at any given time the fringe this fringe that is what is the fringe it's the not yet expanded states that we have encountered so when we expand this the start state these once these four states become the fringe we go diving down here these three remain on the fringe and we get new fringe nodes like this one siblings that we didn't consider as we dive off down at any given time wherever we are wherever the last state we expanded is there's a path back to the root and the fringe all lies and siblings along that path these are siblings of this step in the path this is a sibling of this step and so the only unexplored vertices are one step off the current path back to the root so all the fringe nodes are one step off the path that goes from our currently expanded node the one we just expanded back to the root so we can remember just the path back to the root and at each step along the path to the root we'll have successors and we'll need to we'll need to know which ones we've done and which ones we haven't done so like here we know we are working on this one we haven't tried these yet but at some later point we might wrap this up we can't do anything further there then we'll come back and do one of these and let's say you know once we've explored everything you can reach down here if there's if we still haven't explored these we'll come back and and say explore this one we'll need to remember these two are still yet to go so basically we just need to remember the path back to the root and the natural way to do this is via recursion and the control stack so it's really common not to implement your own stack but to do a recursive algorithm which provides you a control stack which will keep track exactly of what you need so i think any computer engineer should be able to envision what the control stack is doing during the recursion and understand how it's working and it's very helpful in understanding depth first search in particular so here each rectangle is a invocation of the search function that's recursive so it's calling itself this call this search is calling that search is calling that search and what's the difference between these calls the current state the initial call is at the start state this center one that's this state but it makes a call it picks one of the nodes on the fringe which are these one step nodes it picks one and makes a recursive call on it and that's this one and then this one picks one of its children and makes a call and that's the third one and so this this this state the current state variable in these co in these uh environments for the search function each environment has a current state variable and they are this state this state this state going down the path so the path back to the root is is stored on the control stack in that sense and these black edges are shown just to dramatize for you that if i was to look on the control stack and identify these nodes they would all have edges between them the edges aren't on the stack the edges are in the graph but they would have edges between them in the graph and what we store as we've stated last lecture we store for each vertex the parent that led to it so the parent values for each of these nodes point back to the root this way along the stack now again neither the red arcs nor the black arcs are actually in the stack the only thing in the stack is the value of the ver the current state variable i'm just saying that those values form a pattern such that the parent arcs point up the stack and the graph arcs point down the stack and there's a graph path being traversed and we're currently expanding here so stack is sort of keeping track of this for us and this inside of this frame in each frame there'll be a loop that's looping over the children of that or the successors of that state where you can get in one action from that state and we try this one and if that ever returns we'll try the next one see if that's been explored yet and if not keep exploring there and then with that when that returns try the next one so this loop is keeping track of which of these children of the root well at the root the loop is keeping track of which of these children it's working on and which one's next so that's a way to envision the data structure inside of a executing depth first recursive search that can really help you understand what's going on so then you can think about the space usage how much space does it use well it uses a frame in the control stack for each step in the longest path that it goes to so whatever the depth it ends up going to that depth times the size of a frame in the control stack and you can imagine that maybe the frame in the control stack needs to have data structures as much as the branching factor it probably doesn't so we don't really even need to put the benching vector in but it doesn't really uh hurt our discussion to say well maybe we need a frame that uses data structures of size b branching factor and then we'll need d of those frames so that's at where's b of b d asymptotically branching factor times depth of the stack this d here is the deepest stack depth that it's going

to use so it goes charging off down it's building a large control stack and there's a serious question how deep does it get but however deep it gets it's basically linear in the depth it's no longer b to the d power remember that's what our priority q would grow to in this model as we went to depth d we would have b to the d vertices in the priority queue or states and so we've eliminated exponential space usage at what cost well in red here i've marked that we clearly don't find the shortest solution first so there's no point in waiting for the expansion of the goal if we encounter the goal we've got a path to the goal and waiting for it to be expanded it's not going to help it's not optimal anyways so a goal is found but this doesn't find the shortest solution first so that's a serious issue what about the run time we're talking about space has now become linear what's the run time well just like in the breadth first algorithms these algorithms are studied in the algorithms community and they're viewed as polynomial time because they run linearly in the number of vertices v and the number of edges e but an ai person will say well so what the number of vertices in the graph is like the size of the number of atoms in the universe or something astronomical numbers of vertices in a i problem search graph so the fact that we're linear in that we're linear and an extremely bad number this number of vertices is at least as many vertices as there are in the you know layer one particular layer of the graph and that's exponential as we move away from the source so this this is the algorithm's view the of v plus e and in the algorithms community we'll be analyzing graphs where the whole graph fits in memory and we can use this depth first search in ways that we haven't discussed for sorting the graph so that the sorted order of this the vertices is have all the edges running forward that's a topological sort or for finding strongly connected components in the graphs these are computer algorithms that are quite useful in many settings but they don't really apply in the ai setting where the graph won't fit remotely in memory even if we put all the memory on earth in one place so we don't have a runtime win here we still have a sort of b to the d um kind of run time not d as the depth we went to because we're never even necessarily exploring all the other notes but all the nodes in the graph is an astronomical number and this doesn't give you any particular reason to think you're going to find the goal soon so you may explore for who knows how long we've not won on time here although it is a fast algorithm per node it doesn't have to maintain some messy q data structure this the stack data structure is extremely efficient all the activity is at one end pushing and popping there could be viewed as some overhead and pushing and popping on the control stack but this is a fast and effective algorithm apart from the fact that it goes charging off doesn't find the shortest solution and has no particular guarantee of finding a solution soon so we have to overcome some of those issues so the key idea to overcome those issues is to bound the depth okay i'm going to go charging off looking for my grocery store but i'm only going to go so far and when i hit the depth bound i'm going to treat it as though that vertex that state doesn't have any um children it doesn't have any successors there's nothing i'm allowed to do in it because i've reached the depth bound and that will cause me to backtrack and try the next youngest vertex so before we consider exactly what that means i want to point out that the depth bound i'm talking about doesn't have to be number of hops like the number of edges that we've traversed could be a natural notion of depth but we could also use the g function which adds up the costs and bound how much cost we're willing to search down one path so if we look down a path and it has a really expensive edge we'll it will exceed the depth found even if it isn't very many hops and later when we do heuristic search we'll see we can even use the heuristically inspired function f to evaluate whether a vertex has gone too deep or not that's a four pointer but basically many different notions of what's too deep will work but when you're looking for your grocery store you're going to go so and so far whatever your notion of this is too far you'll cut off the search and back up and try a different path and and that looks like this so here we start at our start node start state home looking for the grocery store or whatever we we identify that we've seen that we can get to these places right but we choose one of them and pursue it so we pursue this one and then we see that we could get there but we choose one of the two options and pursue it and we keep doing that and then we hit the depth cut off we're going to stop at the depth cut off not continue because even though they're edges we'll pretend they're not there and say well okay we don't have any new nodes under consideration here so what's the least the most recently add added fringe node that'll be the sibling and once we this hits the cutoff what's the most recently added fringe note that'll be the you know the uncle over here and once once this goes down and pursues to the depth cut off it

will backtrack and we'll we'll pick one of these and so each of the pads will end up getting explored to the depth cutoff along the different paths has a kind of breadth-first flavor even though it does not need b to the d space it only keeps track of the path back to the root at any given time the depth cut off is what's giving it the breadthy flavor not the fact that we're keeping the whole layer in the fringe we're not we're just keeping the the nodes one step off the path to the root that's all that's in our fringe and so we're still using a control stack to store that and and we still get the linear space usage in the depth cut off just to be a little more precise the order of evaluation in this particular picture i've listed here in this diagram so this would be the first node we expand and then we'll expand this node and this node now anytime we have two successors we could have chosen either one i'm just showing one possible order so here and then here we hit the depth cut off well there are notes this in this diagram we're not showing any descendants at four but at five there are descendants across the depth cut off so we we don't we pretend they're not there then we backtrack what's the most recently brought into consideration vertex or node that hasn't yet been expanded it's this one so that becomes six and then at that point the most we can't go nothing's available here so we we have on the path back to the root we have these siblings are the only options left and we pick one of them um they're equally old and expand that one so we could go seven eight nine and then we come over here we pick one of the remaining two ten eleven twelve thirteen this is a possible order of expansion it's very different than the order that breadth first search would do which would expand each of these four before it went any further out what remains is to ask if we are actually going to find the best path to the goal and if we don't find the goal at all what the heck do we do so there's a natural idea here if we don't find the goal is to try a deeper depth right we obviously did explore everything within some depth cut off and we didn't find a goal at all so we must do this again but in fact we can systematically do that if we systematically do it we can actually not just remedy that we didn't find a goal we can actually ensure optimality and that idea is a critical idea in search it can be used to make a uniform cost search or a heuristic search space efficient by using a depth-based approach and that idea is called iterative deepening so this is a very important idea in search that can be used to adapt uniform cost search dijkstra's and it can be used to adapt a star which is our heuristic search that we'll meet later and this idea is to try each depth so do a depth first search to each depth one two three four when you finish the depth for search do a depth five search when you finish the depth five search do a depth sex search and so forth clearly if the goal is at a particular depth at its shortest we will find it with that depth cutoff and if there's a longer path of the goal we won't find it on that one because we are doing a depth cutoff equal to the shortest path of the goal first so because we try each depth we're guaranteed to find the goal at its shortest cost the shortest cost that we can find it at before we find it at longer costs and we get the linear space usage so it's a pretty effective algorithm and you you your intuition which i presume you may have that we're wasting a lot of time by doing first at depth one search then a dep two search then a depth three search then a depth for search and throwing it all the way and starting over each time that intuition is goes a little bit astray for two reasons one we get a big win on space like we don't save the whole fringe that can really speed things up to not have that big data structure around so we get this advantage but also the nature of exponential explosion tells us that almost all the nodes are in the fringe in the last layer like at in a depth d search almost all the nodes are at depth d so if you have a branching factor of say five there are five times as many nodes at depth 10 as there are at depth nine so even just looking at the depth 9 and depth 10 nodes only 16 of the nodes are at depth 9 and the other 84 are at depth at the depth center if you're doing a depth d search but you first did a depth if you're doing a depth 10 search but you first did a depth 9 search you will have only wasted 16 percent on the depth 9 search and unless you worry about the update search there's another factor of five there just generally what's happening here is is this geometric progression we're doing b to the d work at each depth so at depth d that's this one at t minus one it's one over b times b to the d at right at d minus two it's one over b squared times b to the d and if we imagine we've done all these previous searches one of these grounds out where d equals one and then the rest don't even exist but if we put them all in there at fractional costs this sum upper bounds the time we spent and you're going to see almost all of it's in this one times p to the d because the geometric series here has the closed form one over one minus p and we simplify that algebraically we get b over b minus one which means we've wasted one over b minus 1

fraction of the time if the branching factor is 2 that means we doubled the time so it took twice as long but we guaranteed optimality and that's the worst case when b is higher and it typically is higher in most ai search problems we have quite a branching factor we may have an enormous branching factor um as the branch factor goes higher the fraction wasted is is even less so the the general broad lesson of this math is that the overhead of trying all these depths is marginal to negligible and the guarantee you get is that you find the lowest cost whatever your notion of of depth was the lowest depth goal that there is if you use number of hops you're going to find the lowest number of hops goal if you know the g function you're going to find the goal at the lowest cost path and so forth so there's iteratively deepened uniform cost search and we will see there's a heuristic method called a star and we can do iteratively deepened a star and that's it for our discussion of depth first methods