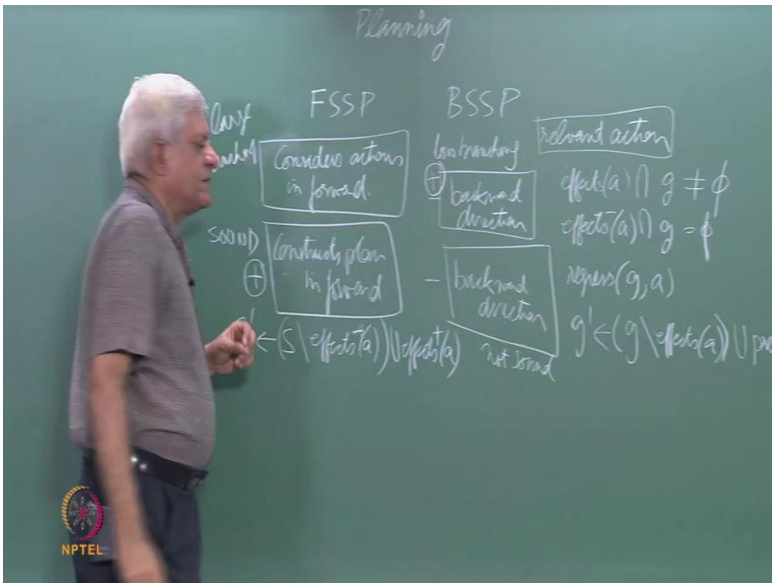
Artificial Intelligence
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 35
Goal Stack Planning Sussman's Anomaly

(Refer Slide Time: 00:14)



We are looking at planning. In the last class we saw two approaches; one was a forward state space planning, and the other was backward state space planning. The forward state space search; forward state space planning; thus, forward state space search, it starts from the start state and keeps applying actions, till it finds a goal state, considers actions in forward direction. This one considers actions in a backward direction. It constructs plan also, in the forward direction, and this one, constructs the plan in backward direction. So, in that sense, the two processes of looking for actions and constructing a plan, happens very in a closely coupled fashion. In forward state space search, we start looking for the first action and then, as soon as we pick a first action, we say this is the first action of our plan.

In this manner, we construct the plan also, in a forward direction. In backward state space search, in the likewise manner, we start looking at the last action, looking for the last action; what could be my last action, and then, also construct the plan in a backward

fashion, by saying that will be my last action of the plan, essentially. Now, if you remember, we had this notion of relevant action here, and the action was said to be relevant, if the effect of a; intersection goal was not empty, and if it has no negative effects, which kind of, distracted the goal. We had the notion of a relevant action and we had a notion of regression. We could regress a goal over an action. So, we would get a sub goal, g prime, if which is obtained as g minus the effects of a, because we expect that the actions will; actionable, produce the effects. The whole thing union p conditions of a. In the similar manner, we had for forward state space search; the notion of an applicability of an action and the notion of progress. So, a state could progress over another state. So, you could progress over it.

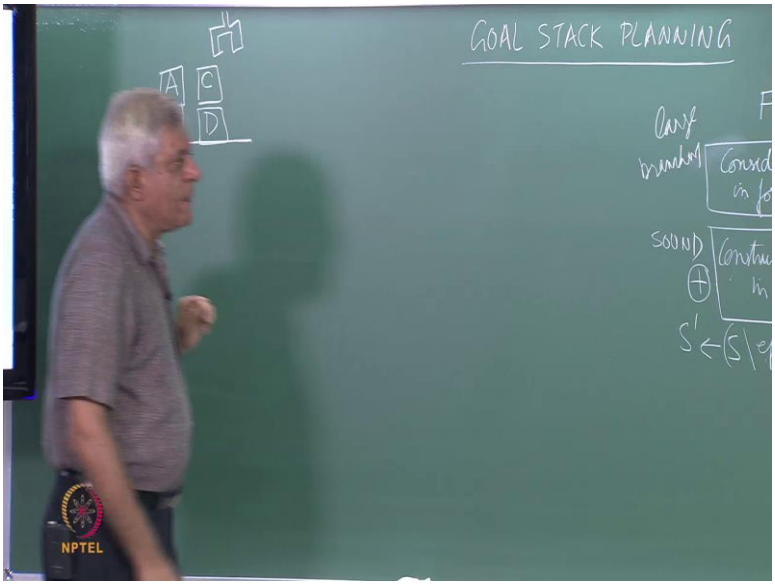
This regression progress and the notion of applicability and relevance was used basically, to do both these stars, that looking for actions and building the plan at the same time. So, the process of building the plan is that you move from one state to the next, and then, look for an applicable action; then, move to the next state, and look for an applicable action; move to the next state. Here, you are looking at a relevant action. So, you look at a goal, find a relevant action, regress to a goal g prime, and try to find a new action at that point. So, what we had observed then, was that this was a sound process that, when you progress from one state to another; what you get is a legal state, essentially. So, this was sound, but this was not sound.

We had seen that you could regress to a set of predicates, which could not have been part of a state. So, for example, you might have something, like holding a and holding b, at the same time. That, of course, not possible in a state in which, because we are considering one arm robot. So, you could; this process of regression was not sound, in the sense, it was not closed under the set of states. You could start with a possible state and you could end up with something, which is not a state, whereas, this was sound and you would always end up in states, which is why, when we did backward state space planning, we said one of things to do is that after you found a sequence of actions, check, whether it is a valid plan or not, before accepting it. So, this was a plus point of forward state space planning, and this was corresponding negative point of backward state space planning. On the other hand, in forward state space planning, we had large branching factor.

Because the state was a complete description; it may have hundreds of facts. There may be hundreds of applicable actions. Forward state space planning would consider all those hundreds actions, and choose one of them, essentially. Backward direction search had low branching, and the reason for that was that we were focusing on the goal; we are trying to see, what we need do to get the goal, predicates into our state, essentially. So, this was a plus point for backward state space search. Today, we want to look at an algorithm, which combines both these features. So, what do we mean by this? We want to look at an algorithm, which will consider actions from the goal point of view, in a goal directed fashion, but it will construct plans from the starting state to the goal state, essentially, which means that we will be benefiting from the low branching factor of doing goal directed search, and also, the soundness of constructing a plan in a forward manner, essentially.

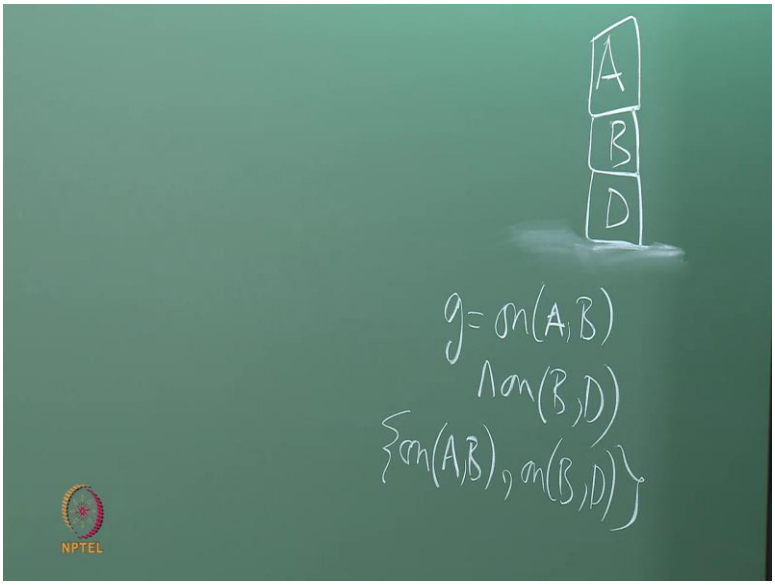
You should ponder by little bit over this, as to why is the progress action sound, and the regress action not sound, essentially. So, the actions are not symmetric in that sense, essentially; you cannot prove both ways. These are sort of an arrow of time, which says this is a precondition, and this is a post condition. So, you can only construct plans by looking at pre conditions and making post conditions.

(Refer Slide Time: 07:38)



The algorithm that you want to look at today is called goal stack planning. It is actually, one of the earliest planning algorithms devised, and was in fact, used in the skips program, which was used to control the robot in Stanford that we have spoken about, essentially. The general idea of goal stack planning is the following. What I will do is I will give a high level description of the planner, and then, we will look at an example in a little bit more detailed, essentially. So, as the name suggests, this uses a stack to do the reasoning, and we do the following that; let us also consider an example, along the same, at the same time. So, let us say that this is an example. Again, we have resorting to the blocks while, because we are familiar with it, but you must keep in mind that these are general domain independent algorithms that we are considering. This is a starting state and I am not drawing anything, which is relevant. You can imagine that there are 50 blocks, which I have not drawn here, which will not interfere with our plan. So, we just want to focus on the planning actions today.

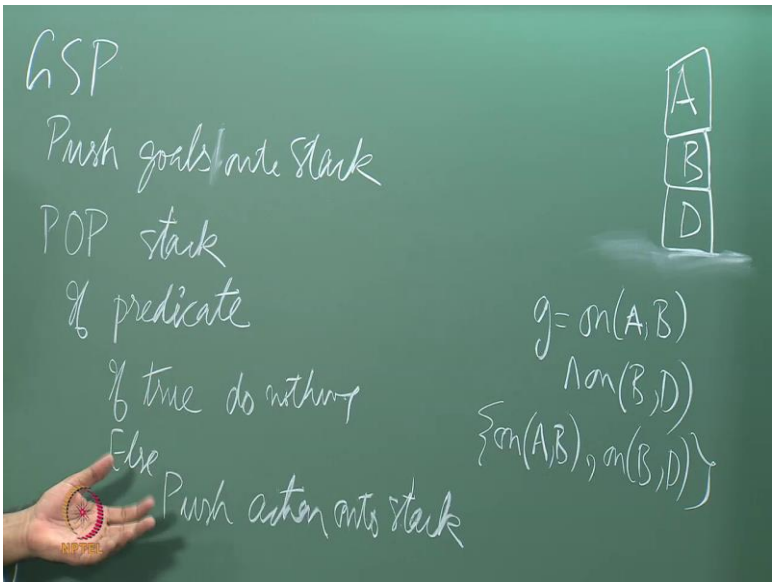
(Refer Slide Time: 09:19)



The goal state is that; let us say is that you want a on b, and you want b on d, essentially. You actually, do not care what else is true, essentially. So, the goal is on a b, and on b d. As long as these two predicate are there in my state description, I would call that a goal state, essentially, which means, as long as a is on b, and b is on d, that whichever state it is, is a goal state. You can think of this as a set of states in which, this part is common.

Everything else can be in some manner, essentially. So, this is basically, a set, and the algorithm that we are looking at, will do the following.

(Refer Slide Time: 10:22)



It pushes; you start of a pushing the goals that you want to achieve on the stack. So, the top of the stack will always, contain the goals that you want to achieve, essentially. When I say goals, I basically, mean the predicates of the goal, essentially. Now, let me do this here, and let me write the algorithm, here. So, goal stack planning; push goals on to stack; then, you pop the stack. There are various things that can come out of the stack. If it is a predicate; I will use very loose language here. When I say, if predicate, I mean a statement like; on a b, or on b c; or holding a; or some such things. Then, there are two possibilities; one if true, which means, it already holds in the world; then do nothing. Else, it is not true; push an action on to the stack. So, this is a basic process that this algorithm follows. The stack has, you are pushing these goal predicates on to the stack.

In our example, we push these two things on to the stack; then, you pop. So, it is the alternates between push and pop; you pop the stack. If it is a predicate that comes out, then you check, whether the predicate is true. If it is true, then you do not have to do anything. If it is not true, you push an action on to the stack, essentially. What action? You should say the element action; here, define the notion of relevant action.

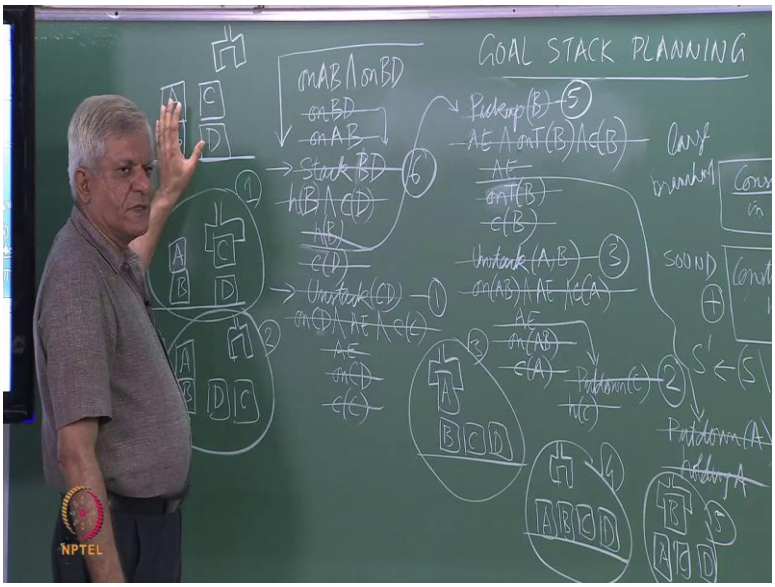
If it is not a predicate, it must be an action there. Only two kinds of objects in a domain, either predicates or actions; if it is an action; I forgot one thing. You push the action on to the stack and push. So, let us say this action is a; push pre conditions on to stack. You first, push the action. Then, you push the pre conditions of the action, and you can get some intuition here, that you are pushing an action. Then you are pushing the pre conditions, and then, you will look at the top of the stack, and there will be the pre conditions of the actions. If they are true, then this thing will happen; you will not do anything; you just remove them. Eventually, if the action comes to the top, then you will say, yes, I have found one action, essentially.

There is an extra step, which is, push each predicate on to the stack, essentially. So, this part is that, if then part, then for this if, we have these else, pop, action, or we have already done the pop. So, add action to the plan, and by this, we mean that a plan becomes plan followed by dot where, the dot operator is a concordinate operator, which says that you take the plan and add the action at the end of it. So, you found the next action, essentially. So, this part, that you are talking about, it constructs plan in a forward direction, is taken care of by this operator, essentially. The new plan is an old plan with the action at the end. Initially, of course, old plan will be empty. The moment you find the first action that will go into it, and then you find the next action that will go into it, and so on and so forth. So, this is a high level algorithm for goal stack planning. Let us see how it actually, executes this. We will, sort of, try to simulate for this small problem; what goal stack planning does? Before I do the simulation, let us make an observation as to what this is really, doing. It is taking a set of goals; the pre conditions. Every action has a few pre conditions. So, it is a set of goals, or set of sub goals, you might want to say, and push it on to the stack.

Then, it pushes each predicate of the pre conditions on to the stack. We should do the same thing here, essentially; push each predicate. We will see the usefulness of this step in the example that we see, but the important thing to note here, is that it is taking a set of goals, or a set of sub goals; we use the term goals and sub goals, interchangeably. The initial goal is the only final goal. Everything else is a sub goal, but we tend to use the term goals, also for that, essentially. When we have a set of goals to solve, for example, in the pre conditions of an action, we put them one by one into the stack, which means

we have serializing the goals, sub goals, essentially. So, this is saying; serializing the sub goals. In effect, we are saying, we will first achieve one sub goal. Then, we will achieve the next sub goal. Then, we will achieve the next sub goal, and in that fashion, essentially. So, we have, in some sense, if you look at what A star did, it also said, I am breaking up a goal into sub goals, and I will solve each of them independently. This is doing that, but it is also imposing an order in which, you will solve them. So, that is why we use a term; it is serializing the sub goals, essentially.

(Refer Slide Time: 17:44)



So, our initial sub goals are those two, on a b. I will just use, I will not use brackets, just to make it short; and on b d, and I will go by stack downwards, and I hope that you get used to that idea. When we pop stack, the stack we will just put a line across, to say, that element has been popped. So, you must visualize this stack going down. Let us say that we put these in this order that we say, you achieve on b d, and you achieve on a b. So, this is the bottom of my stack, and my stack is going like this. So, whatever done, I have pushed the two, I have pushed the goal, which is these two elements on to my stack, and then, I pushed each predicate in some order. So, we are not saying in what order; we saying in some order; push them in to this stack. This is a place where, you can, sort of, try to think of heuristics; what is the good order of pushing things. So, this is a goal given to us.

I want to emphasize again, that considering of actions is done in a backward fashion. So, we are now, only trying to see what actions will achieve these goals, which is exactly like, what backward state space does; except for that backward state space says that moment, for example, if you look at on a b, it will say the last action must be stack a b; stack a on b. It starts constructing the plan also, in a backward fashion. We will not do that here; we will wait a little bit more patiently, till we are sure that whenever, we add an action to a plan, its pre conditions are true. Backward state space planning does not look at pre conditions at all. It only looks at the relevance of an action. It says if an action is relevant, it could be the last action, and we saw that this leads to the trouble that the plan construction process is not sound. So, we have on a b, now, and we go to the pop cycle.

So, we push this out. This is gone and then, we have this condition. It is a predicate and it also, happens to be true, in my given state; you look at the value in a given state. It is true so, you do not do anything. Then, you pop the next thing out on b d. Remember, this was popped out first, and now, we are talking about on b d. So, let me, sort of, use an arrow to denote that we are considering this; just for our sake. That is not true; on b d is not true in my, this state, and therefore, I push and action, which will make this true. So, the action that makes on b d true is stack BD. So, let us say we use this arrow to depict the fact that we have pushing an action, essentially.

So, you push an action and we push the pre conditions of the action. What are the pre conditions of stack? I will you short forms; h for holding; holding b, and clear d, anything else; you remember these preconditions for stack. You must be holding b, and d must be clear, I think that is about it. Then, I have pushed the individual actions. While, we are doing this example, we will use a simple heuristic, we will assume that the holding action is the last action we want to do; last goal we want to achieve. Remember, these are two goals. If you can, just to recall, this is a, let me put brackets here; that we want holding b to be true, and we want to check, whether d is clear to be true, and we will push each individual action.

So, the first action, the first predicate we push, will be the last predicate we will check, and let us use this heuristic between ourselves. In practice, of course, an algorithm may

have to back track and try the other option or something like that. That we will check for holding b later. First, worry about; let us worry about clear d, essentially. So, this is a push space. In a push space, everything gets pushed; the action and its pre conditions, and individual goals in the pre conditions. We will refer to them also, as goals, because this is a goal stack now, essentially. So, we know push, we know pop this c d out.

That clear d is not true, but we must insert; we must push an action, which will make clear d true. So, this is a situation; c is on d. So, we can use an action unstack. So far, we are doing backward search c d, and then, the preconditions of unstack c d, which are that; on c d must be true; and arm empty must be true; and one more, clear c must be true. Then, these individually, again, in some order, let me choose arm empty as the last predicate. Intuitively, I just want to reduce some amount of extra work we want to do here, but this is a matter of choosing heuristics. So, everything is pushed here, like this. Basically, this is a cycle; I have not mentioned it here, but this whole thing is in to a cycle here. Then, you go and pop this clear c; now, clear c happens to be true in our world. So, we do not do anything. On c d also, happens to be true; on a also, happens true, and it is not a surprise in particular case; that the conjunct of all three on CD, an AE, an CC happens to be true.

So, we remove this from the stack, and now, in the next pop, an action comes out, which is this last part of the algorithm, which says, if it is not predicate, it must be an action, and add action to the plan. So, this becomes our first action; unstack c from d. The world has changed now. The world is, I am holding d. Whenever, I look at a predicate, I must look at this world. Now, you will notice that when I am talking about actions, I am going in the forward direction. This was the given start state, and this is the first action that will be there; part of my plan. The first action will be unstack c from d, essentially. So, everything that we do here will be sound, essentially.

That also, does not lose sight of the fact, that we are considering the actions in a goal directed fashion. We started off by saying that what is necessary for making on a b, on a b true, on b d true, and then, we said to make on b d true, you must do stack b d, and then, we discovered that to do stack b d, we need to do clear d, and to do clear d, we can do unstack c d, and we find that we are able to unstack cd, and so, we put that as a first

action. So, this signifies the plan. So, that is gone now, from the stack. Then, it has got holding b as a next action. Holding b is not true in this world; you are holding c, essentially. Let me grow the stack from here, that when I pop holding b out, I am forced to insert an action. So, I have a choice here. Notice that to make holding b true, I can use an action; unstack b from something, or I can use the action; pick up b, essentially. We will assume that we have some non determinism going on here, or you could look at the state and try to decide, which of those two actions is a relevant action? So, we will assume that somehow, we are used pick up b.

So, the stack is now going like that, and along with pick up b, the actions, which are arm empty, and on table b, and clear b. So, let us say I look at them in this order or in the same order; on table b and clear b. So, I have pushed this action and it is preconditions. Then, I pop the top of the stack. Remember, the top of the stack is actually, at the lower end of our list; I have popped this. Clear b is not true in the world that I have here. So, I must insert an action, which is unstack something from b, but we will assume that we have figured out that it has got to be a from b, and the preconditions for that are on a b, and arm empty and clear a. So, let us say arm empty, on a b and clear a; clear a is true. So, I can remove it from the, pop it from the stack. On a b is also true. So, I can pop that from the stack, but arm empty is not true, because this is the world that I am looking at. I am moving forward from here. I am holding c.

So, I must make arm empty to make arm empty, I insert an action, put down c and the preconditions for that are holding c, and that is all. So, you pop this and you pop this, and this becomes a second action; that you have put down c. So, now, the world looks like a, which is the world. I have done two actions. One action I have done is unstack c from d, and then, I was in this state. Then, I have put down c, then I am in this state; that is a second action. Then comes this conjunct, on a b is true here, arm empty is true here, clear a is true here. So, I can remove this, and then, I can pop this; this becomes the third action. Now, the world looks like, you are holding a. The rest is all on the table. So, this is that.

Let me label these states. This is a state after action one. This is a state after action two. This is a state after action three, which is unstacked a b. Then, the next thing on top of

stack is, on table b. I pop that and I see that is true in this state. Then, arm empty is not true. So, I have to achieve arm empty. Let me start here. I need, I remove arm empty of course, and then, say put down. So, I am holding a. I need put down a. Again, there is a choice, which I am, sort of, skinning over here. The choice is really, that either, I put a down, or I put it on b, or put it on c, or put it on d, essentially. May be, you can do a little bit more sophisticated reasoning here, but I am, sort of, just to illustrate this, I am just saying that we have something, like a nondeterministic choice happening, which means magically, we are making the correct choice; this is to put down.

For which, you must be holding a. So, you can do this, and this becomes the fourth action. After the fourth action, everything is on the table and the arm empty. So, I must go back by a stack I have; I find this conjunct here; arm is empty, on table t b, and clear b; everything is true. So, I pop that, then this becomes my fifth action; pickup b. So, at the end of fifth action, I am holding b, and a c d are on the table. So, this is gone from my stack, and this is where, we had taken off. Now, you are holding b and clear d. You can see that in that fifth state, both are true; you are holding b and clear d is true. This goes off. Now, we have the sixth action coming out. The moment an action comes out of the stack, we know that it is applicable. Why, because we have just popped their preconditions; pre conditions must be true, essentially. So, it must be applicable. So, this is the sixth action; stack b on d. So, this is how it looks and arm is empty.

Now, observe that in a manner of speaking, we started off with two sub goals; on b d and on a b. We decided to do on a b first, and in this case, it was already true in this state. So, we do not have to do anything, but as you can see that was a right choice, essentially. If I have to achieve, if we look at the goal state, which is that a must be on b, b must be on d; you can see that the way to achieve the goal is to first, achieve on b d, and then, put a on top of the stack that you have, tower that have constructed, essentially. We choose an opposite order, and we ended up finding a plan, which is the six step plan, which says that you unstack c from d. So, this is the state. Then, you put down c. Then, you unstack a from b. Then, put down a. Then, you pick up b, and stack it on to d, which is what, this did, essentially.

So, on the surface, it looks like we have achieved both these goals, but if you look at this

state, when we achieved the second goal, which is on b d, which is what we were doing all this while, and as a result of which, on b d is true here. We have undone the first goal that we had done, essentially. My first goal was that a should be on b. We started off with a on b, but by the time, we finished on b d that, a now, lying on the table. So, you can see this is the reason why, we have added both the conjunct of the goals as well as individual goals. So, we are saying we want achieve this conjunct, but we will do it individually, will serialize the sub goals; we did this; then, we did this. Then, we found that in this sequence, we, somehow ended up, undoing some of the goals. So, when I popped this out, I will find that this is not true.

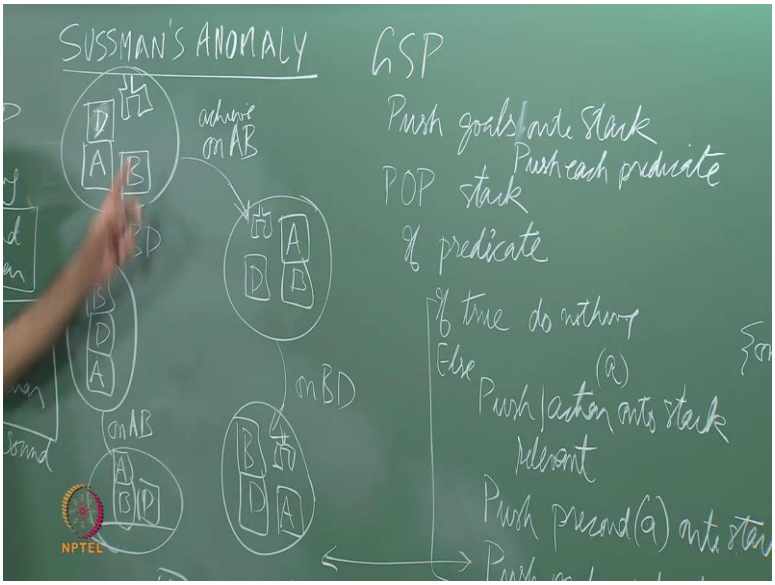
So, I will insert both the goals again, into the stack. So, let us say if we inserted in the same order here; that I insert on b d and on a b first; first, on b d, then on a b, which means I am first doing on a b and then, I am doing on b d, as I did in the last time, essentially, but now my starting state has changed. That is my starting state. So, I will not go into the stack, because we do not have a space left on the board, but you can imagine that to achieve on a b, we will do the same thing; stack a on b. To stack a on b, you must be holding a. To be holding a, you must pick up a. So, you pick up a and stack a on b; these two actions, you will end up doing. So, you will achieve on a b.

Once you achieved on b, you will go back to on b d, but this time, on b d is already true, because in that state, as you can see, it is already true. Only thing you are doing is in the seventh and the eight step, you are picking up a from here, and stacking it on to b. So, this is a final state that you are looking at; a is on b and b is on d, essentially. So, both the sub goals are true, and then, I am finally, able to pop the goal, and that is a terminating criteria. If I can pop the goal and come up an empty stack; that means, I found a plan for solving my problem, essentially. To emphasize what goal stack planning does, it does, it considers plans in a backward fashion. It looks for actions in a backward fashion by putting the goals that you want achieve, on to the top of stack, starting with an empty stack of push, and it always, looks at the goals set on the top of the stack, which means, it is doing backward reasoning, but when it comes to constructing a plan, when it comes to saying that this is my first, these are my actions; it starts off by choosing the first action first.

So, if you look at this plan, this is a first action. Even, when you want to actually, implement the plan, you want to first, unstack a from b, sorry, unstack c from d; put it down on the table. So, it is doing, it is constructing the plan like a forward state space planner. It is looking for a plan like a backward state space planner. So, it is taking the advantage of both the things. It is only focusing on the goal by looking for a plan, or it is making sure that when it is construct a sequence of actions, it is a valid plan, because it is doing it in the forward fashion. In the process, it ends up serializing the goals, but we have to be extra sure that we do not disturb the goal; so that, we add this whole thing or doing this extra thing, all over again, essentially.

Now, it turns out, and I will leave this as a small excise for you, is that if I had considered them in the opposite order, which says that first, do on b d, then, on a b, which inside inverted the order in which, I push up in to the stack. First, I would have done on b d, which amounts to everything that we have done here, and I would have ended up in that state. Then, I would have done on a b, and I would have just picked up the a; this a, and put it on to b. So, there is an order I can choose in which, I am not undoing the work done for the solving the previous goal. This particular order, I am undoing the work. Of course, I did not have to do any work to achieve on ab, because it was already true, but imagine that, a was on the table here, or something like this, and then, I picked up a, and put on to b. Now, I would have undone the work that I have, I am doing, essentially. So, there is an order in some cases, essentially.

(Refer Slide Time: 40:46)



Now, interestingly, it was shown by a guy, called Sussman, that it is not always possible that such an order may be found. What order am I talking about? I am talking about an order of serializing sub goals; so that, there is no disruption of previously achieved sub goals, essentially. So, this particular example is known as Sussman's anomaly. If you just search on web, you will find this example.

The interesting thing about this is that he shows that, this kind of planning will not always work; well, work in the sense, without doing this extra work, essentially; because we are serializing the sub goals, we also call this as linear planning. I will achieve one goal, then I will achieve the second goal, then I will achieve the third goal, and so on; I will solve goals in a linear fashion, essentially. So, I serialize the goals, essentially. What Sussman showed was that there are examples where, you just cannot serialize the sub goals. The example is quite a simple one. This is a start state; c is on a and a is on b. The goal state is a on b on c, which is very similar to that, essentially. Let me just, for the sake of illustration, call this d, to make it identical to the goal state that we just looked at, which means, this whole exercise that we did, will also hold for this. Of course, except for the start state is different, but the main point is that I cannot think of two goals, achieve on a b, and achieve on b d. Goal stack planning is forcing me to serialize the sub goals in some order, and what sussmans showed was that you cannot serialize the sub

goals.

Let us see what happens. So, let us say you first, achieved on a b. I am not going to the process, but we are just. To achieve on a b, what will you have to do? You will have to pick up unstack this d from a, put it down somewhere, then we will have pick up a, and stack it on to b; these four actions will achieve on a b, and goal stack planning will do that. You should try it as an exercise. So, a will be on b, and d will be on the; and arm is empty, and then; that means, you have first, done on a b, then you have to do on b d. Now, if you do achieve on b d, you can see something very similarly, happening. You will unstack a from a; unstack a from this stack, put it down from the table, pickup b and stack it on to d.

So, what would you get is d. What we have shown in this example? As an exercise, you should fill in the details and show how block, this goal stack planning will actually, do this? When you first, achieve this, then you achieved this. So, when you achieved this, this is true. When you achieved this, this is true, but this is a goal, and this is not a goal state, which means, to achieve this goal on a b and on b d, I cannot, at least, this order is not correct of doing things. Of course, I can do extra work; pick up this a and put it on d, but that means, I am somehow missing the correct order, if there is one. What Sussman shows was that there is no correct order. So, let us try the other order.

You can achieve on b d first, which is very simple. You just pick up b and stack on to d. So, you have achieved on b d. Then, you achieve on a b. What happens; you have to unstack b, put it down; unstack d, put it down; pick up a, put it on to b. So, you would get a b d. Again, you can see the other order also, does not do the task. None of these two paths leads to the goal, essentially. Of course, you can do extra work; that is a different matter, but we cannot take these two goals individually, and say, I will solve the first one, then I will solve the second one, and my task is done. I could do it here, if I change the order of this goal. If I have done on b d first, and then, on a b, then I would have solved the task in a serial order, essentially. What Sussman showed was that there are these non-serializable sub goals, essentially. That in many problems, goals are not serialized with.

So, that is a problem with this kind of planning, which we will also, call linear planning, because we are serializing the goals and saying, I will do this first; and I will do this first; and so on. Of course, this is something that we have observed earlier, in other situations. For example, when we talk about solving the rubrics cube, then if you say I will do the top surface first, and then, the middle layer and then, the lower surface and then, by the time you finished the top surface and while, you are doing the middle layer, in the middle, you upset the top layer.

Of course, those who know the solution know, how to get it back, but that is like doing an extra work, essentially. So, rubrics cube is the typical example of a goal, which is fundamentally, not serializable, like this problem, which means that there is no way that you can achieve the first goal, and not have to achieve it later again, essentially. Such problems are called non serializable sub goals, essentially. So, in the next class, we will look at an approach, which some people call as non-linear planning, which allows us the possibility of solving this kind of a problem, optimally. What do I mean by that? That, if you just think about this problem, this Sussman's anomaly; the best way to solve it follows.

You unstack d, put it on the table; that is two actions. Then you pick up b, put it, stack it on to d; that is two more actions. Then, you pick up a, stack it on to b; that is six actions, but neither of these paths is going to give a plan with six actions. Of course, they will even eventually, achieve the goal, but this will have to do two more actions here, and this will have to do at least four more actions here, essentially. So, I cannot find optimal plan, essentially. In the next class, we will look at an approach where, the possibility of finding a optimal plan is kept open, essentially. You can see that; to find an optimal plan, you have switch between goals, in some sense, that when you start by putting d on top of a b. For example, when you start to do this, then when you put d; what you do? You unstack d, and put it on the table, and you want to achieve a on ab. Then, you want to certainly, realize that if you stack a on to b, you would not be able stack b on to d. So, you abandon that goal of achieving on a b, and switched to the other goal of achieving on b d, in which case, of course, you will find optimal plan, but goal stack planning, because it serializes the sub goals. It says, I will completely solve my first goal and then, go to the second goal, is not able to do that, essentially.

So, we will stop here, and in the next class, we will take up this non-linear planning.