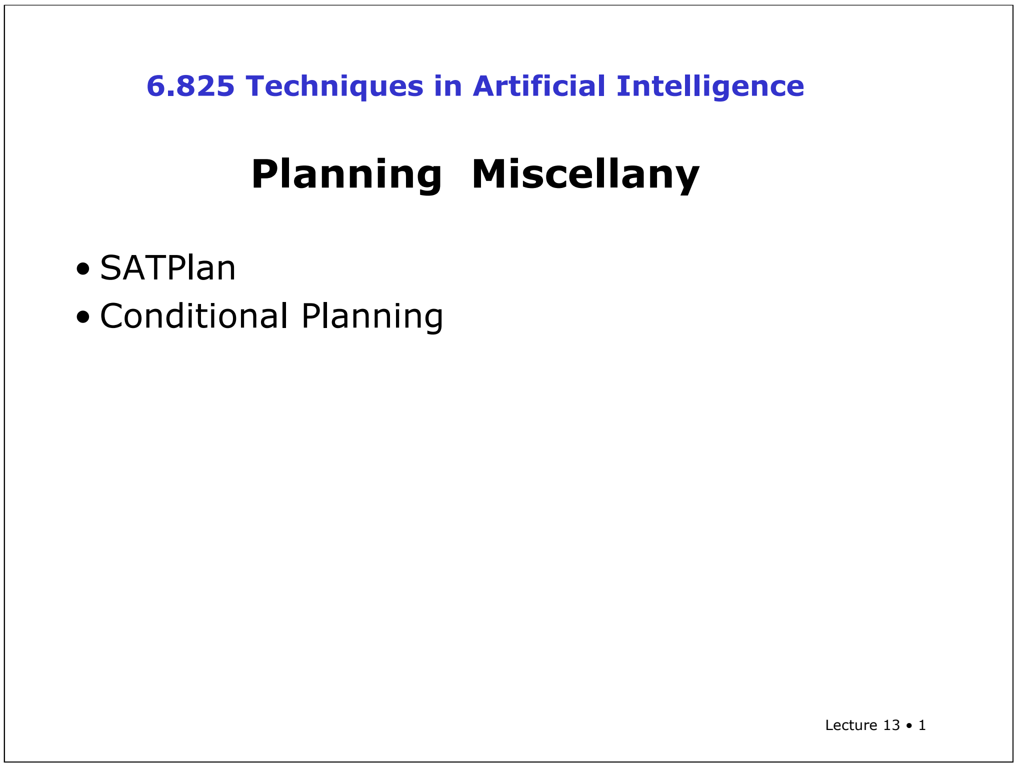


6.825 Techniques in Artificial Intelligence

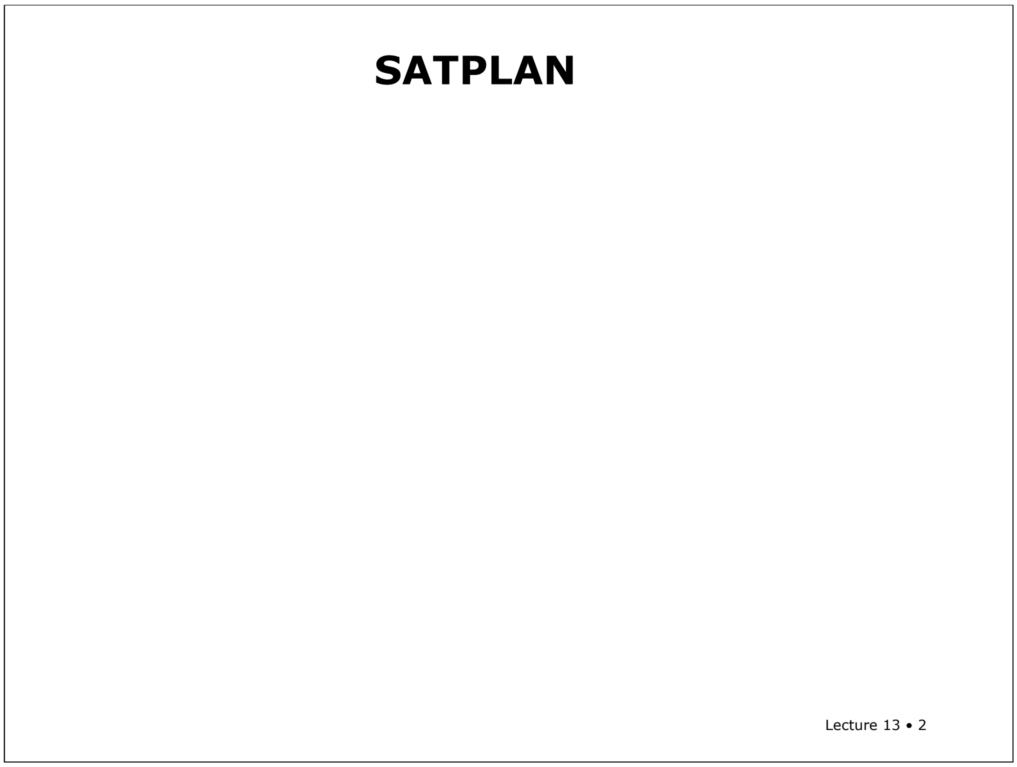
Planning Miscellany

- SATPlan
- Conditional Planning

Today we're going to cover a couple of left-over topics in planning, before we shift gears entirely into probability.

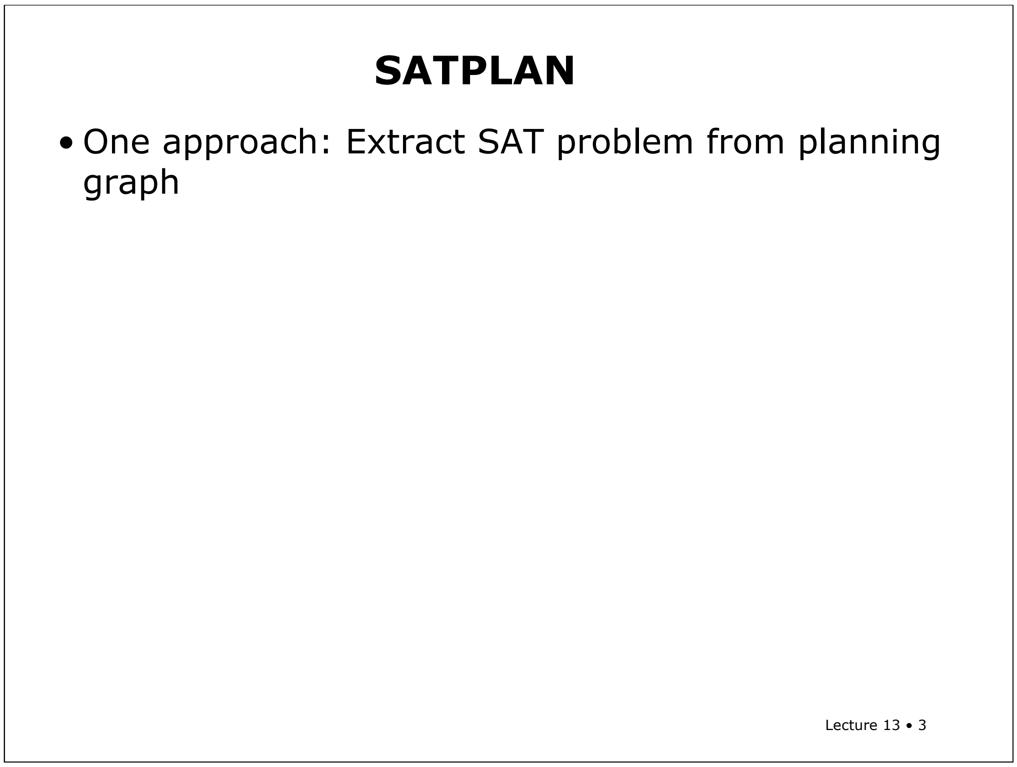
We'll start by looking at SAT Plan, which is a way of solving planning problems using a SAT solver.

Then we'll look at some strategies for handling uncertainty in planning, without moving all the way to probability.



SATPLAN

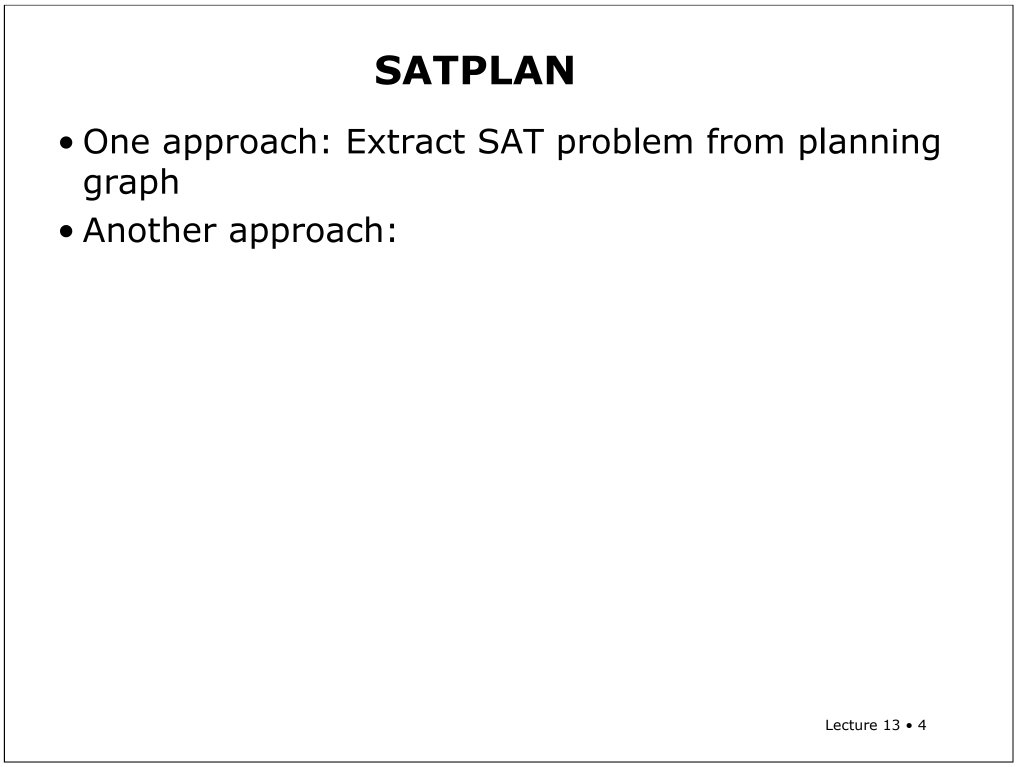
Very soon after Graphplan was developed, it was found to be quite successful. But at the same time, the randomized algorithms for satisfiability started to be working in other contexts, and so people said, "Hey! Well, if we can do this WalkSAT stuff for satisfiability problems in general, then maybe we could take these planning problems and make them into satisfiability problems. That idea leads to a method for planning called SATPlan.



SATPLAN

- One approach: Extract SAT problem from planning graph

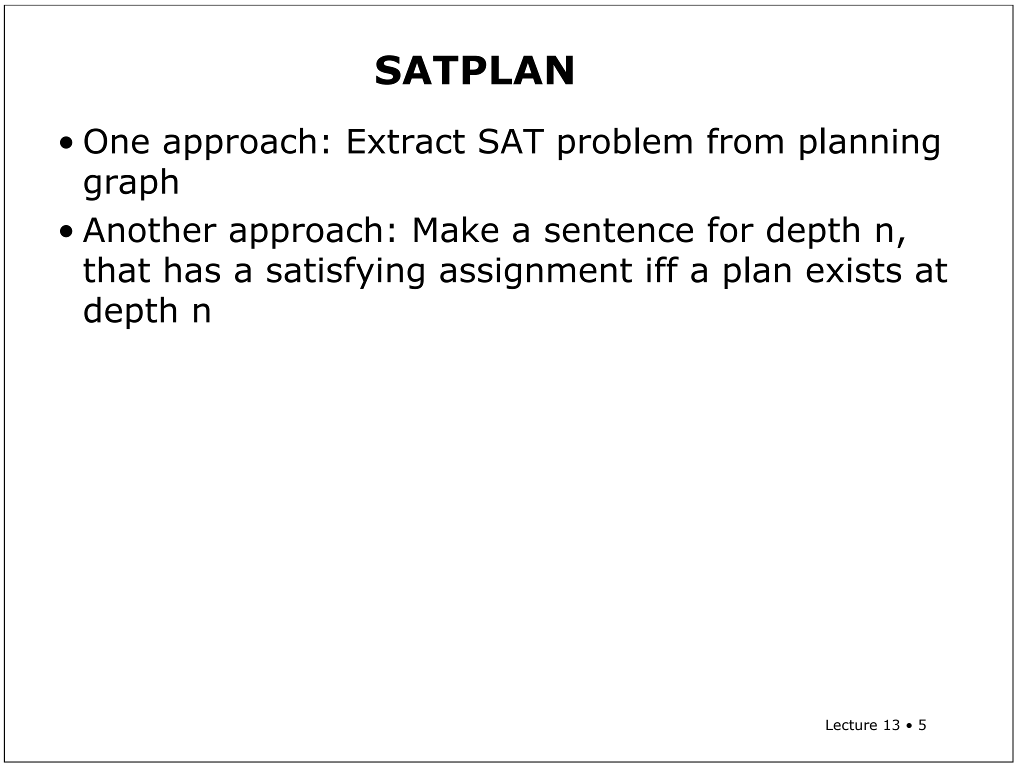
There's one way to convert a planning problem into a satisfiability problem that works by doing the GraphPlan stuff first. It makes the plan graph, and then extracts a satisfiability problem from the graph and tries to solve it. This approach is well described in the Weld paper.



SATPLAN

- One approach: Extract SAT problem from planning graph
- Another approach:

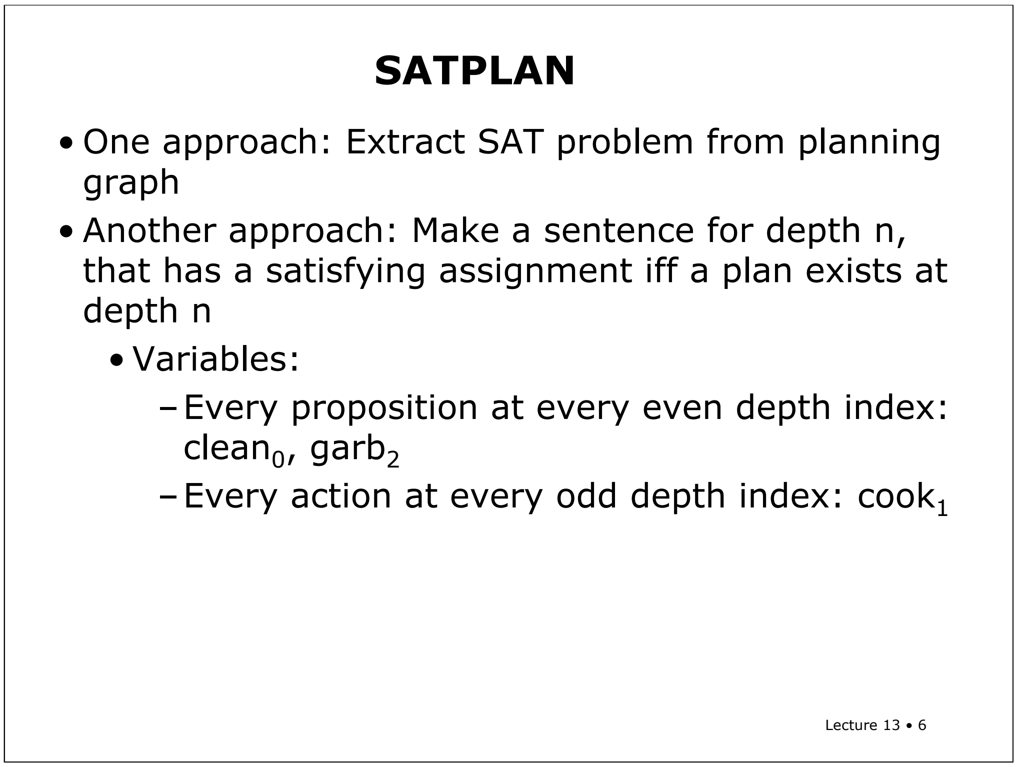
I'm going to talk for the remaining time today about a somewhat more direct way of describing a planning problem as a SAT problem. This is, again, an algorithm that only a computer could love; it's not very intuitive for humans, but it does seem to work pretty well.



SATPLAN

- One approach: Extract SAT problem from planning graph
- Another approach: Make a sentence for depth n, that has a satisfying assignment iff a plan exists at depth n

We'll pursue the same general methodology of considering increasing plan lengths. We'll try to use SAT to find a plan at a given length. If we do, great. If not, we'll increase the horizon and solve it again. We're going to make a particular SAT instance, a sentence that has a satisfying assignment if and only if there is a depth N plan to achieve the goal. And so then if you run a SAT solver on it, you get back a satisfying assignment (if there is one). The assignment will encode in it exactly which actions to take. And then if there is no satisfying assignment, that's a proof that there is no depth N plan.

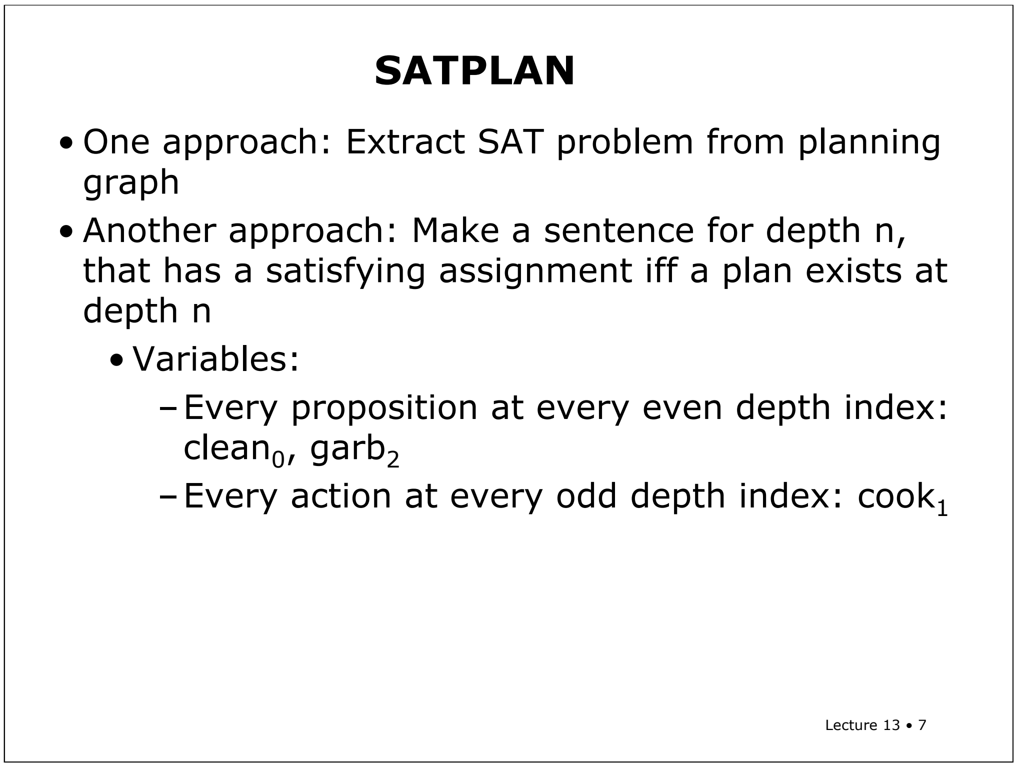


SATPLAN

- One approach: Extract SAT problem from planning graph
- Another approach: Make a sentence for depth n, that has a satisfying assignment iff a plan exists at depth n
  - Variables:
    - Every proposition at every even depth index: $clean_0$, $garb_2$
    - Every action at every odd depth index: $cook_1$

Lecture 13 • 6

We'll keep the indexing idea from GraphPlan, so we're going to have a variable for every proposition at every even step (time index). So we'll have variables like clean at zero or garbage at two. That means my hands are clean at step zero or there's garbage still in the kitchen at step two. So we have a variable for every proposition at every even time index, and we'll have a variable for every action at every odd time index.



SATPLAN

- One approach: Extract SAT problem from planning graph
- Another approach: Make a sentence for depth n, that has a satisfying assignment iff a plan exists at depth n
  - Variables:
    - Every proposition at every even depth index: $clean_0$, $garb_2$
    - Every action at every odd depth index: $cook_1$

This is exactly an instance of reducing your current problem to the previous one. I argued before that reducing planning to first-order logic and theorem proving wasn't such a good idea, because it’s computationally horrendous. It turns out that reducing planning to satisfiability isn't so bad. You get really big SAT problems but at least there's a fairly effective algorithmic crank that you can turn to try to solve the satisfiability problem.

Constructing SATPLAN sentence

Lecture 13 • 8

Remember that a satisfiability problem is a conjunction of propositional clauses. So somehow we have to turn this planning problem into a conjunction of clauses such that if there is a satisfying assignment, then there's a plan. We'll come up with a bunch of different kinds of clauses that we have to add to the satisfiability sentence in order to encode the whole planning problem. You can think of each clause, as before, as representing a constraint on the ultimate solution.

Constructing SATPLAN sentence

- Initial sentence (clauses): $garb_0$, $clean_0$, $quiet_0$

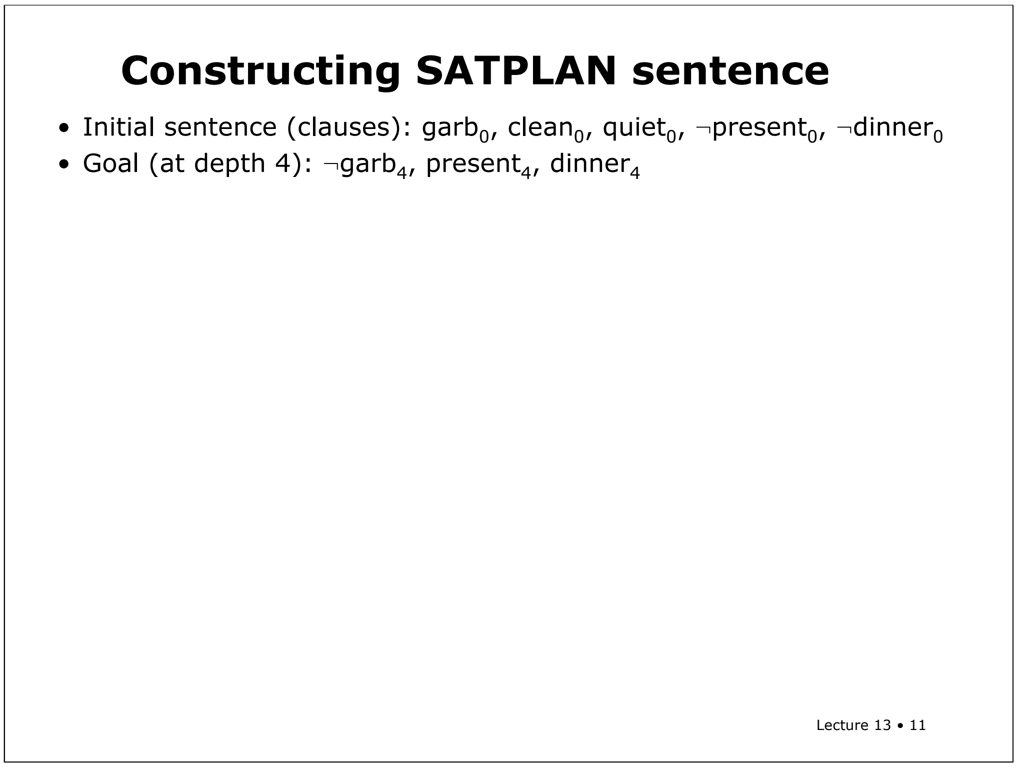
OK, so first, we have the initial sentence. So we're going to have one clause that says - - we'll do it by example -- garbage at zero and another that says clean at zero and another that says quiet at zero. Those are three things we know for sure. So we'll throw those clauses into our sentence.

Constructing SATPLAN sentence

- Initial sentence (clauses): $garb_0$, $clean_0$, $quiet_0$, $\neg present_0$, $\neg dinner_0$

Now, there's a further wrinkle here. In GraphPlan we were able to be sort of agnostic about the truth values of the things that weren't mentioned, right? So when we talked about what was true in the initial state, we just put down the things that were known to be true and we were OK with saying that we didn't know the truth values of everything else. Graphplan would come up with a plan that would work for **any** assignment of values to the unmentioned initial variables.

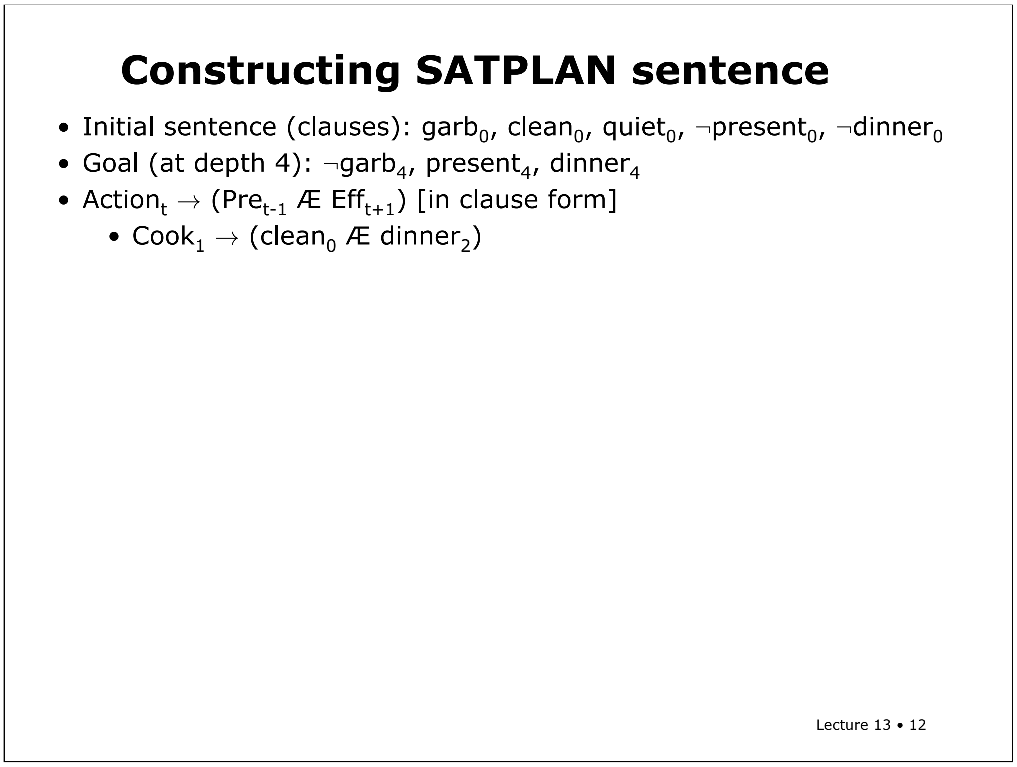
In SATPlan, we're going to have one variable for every single proposition at every single time step and we actually have to be committed about whether they're true or false. They can't just float around. So when we talk about the initial state, we're going to specify the whole initial state. We have to say that, initially present and dinner are false. If not, then it will be possible to come up with a plan that says those things were true initially and all we have to do is maintain them.



Constructing SATPLAN sentence

- Initial sentence (clauses): $garb_0$, $clean_0$, $quiet_0$, $\neg present_0$, $\neg dinner_0$
- Goal (at depth 4): $\neg garb_4$, $present_4$, $dinner_4$

Now we need a sentence that expresses our goal. Let's say we're going to look for a depth two plan. Then our goal would be that there is no garbage in step four, and that there's a present in step four and there's dinner at step four. So that nails down the initial and the final conditions of our planning problem.



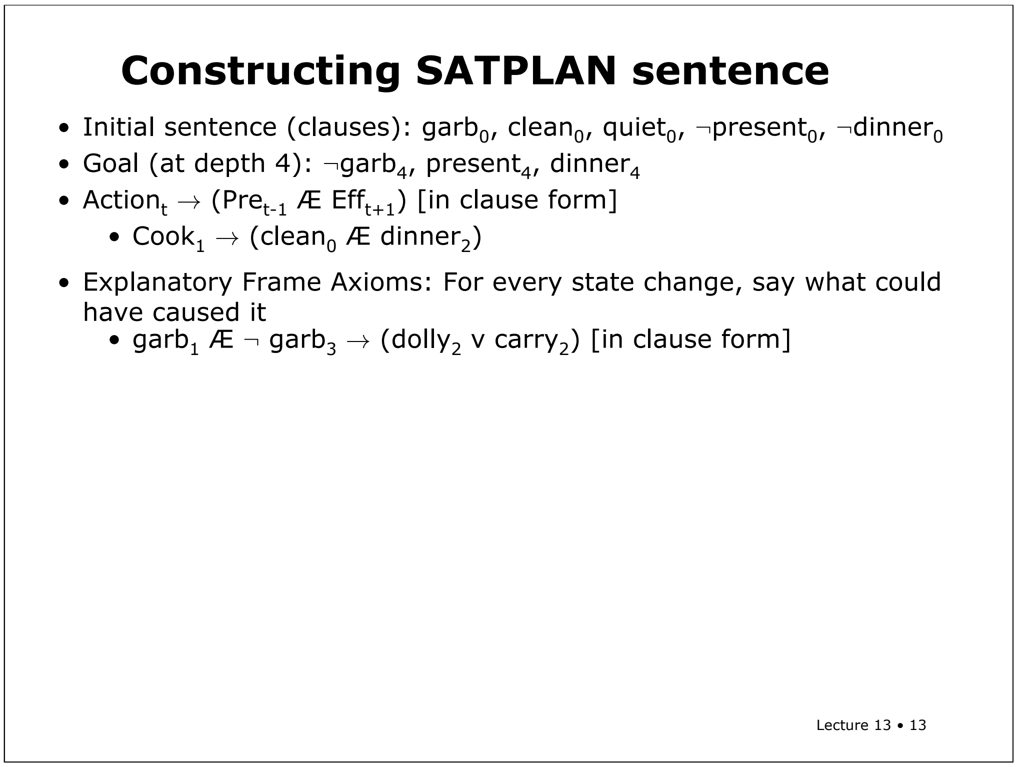
Constructing SATPLAN sentence

- Initial sentence (clauses): garb$_0$, clean$_0$, quiet$_0$, ¬present$_0$, ¬dinner$_0$
- Goal (at depth 4): ¬garb$_4$, present$_4$, dinner$_4$
- Action$_t$ → (Pre$_{t-1}$ Æ Eff$_{t+1}$) [in clause form]
    - Cook$_1$ → (clean$_0$ Æ dinner$_2$)

Lecture 13 • 12

Now we need to capture the information from the operator descriptions. You can think of it as a set of clauses, a set of constraints, that describe how the actions and their preconditions and their effects work. So we'll have a set of axioms that are of the form: an action at time T implies its preconditions at T-1 and its effects at T+1. So let’s think about the cook action. Doing cook at time step one implies clean at zero and dinner at two. If you're going to say that we're cooking in time step one, then you had better also say that you're clean at step zero and there's dinner at step two. It’s easy enough to turn that into clausal form.

For every possible action and every odd time step, you throw in one of these axioms. That's exactly like drawing the arcs between the actions and their preconditions and their effects, just hooking things up.



Constructing SATPLAN sentence

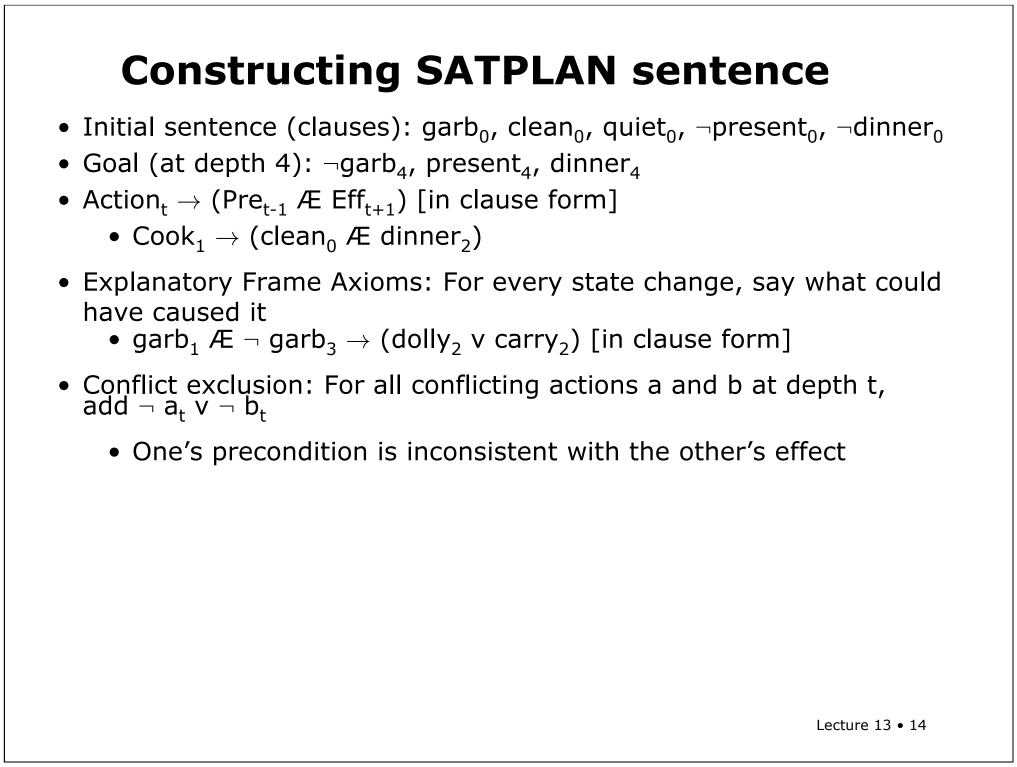
- Initial sentence (clauses): $garb_0$, $clean_0$, $quiet_0$, $\neg present_0$, $\neg dinner_0$
- Goal (at depth 4): $\neg garb_4$, $present_4$, $dinner_4$
- $Action_t \rightarrow (Pre_{t-1}$ Æ $Eff_{t+1})$ [in clause form]
    - $Cook_1 \rightarrow (clean_0$ Æ $dinner_2)$
- Explanatory Frame Axioms: For every state change, say what could have caused it
    - $garb_1$ Æ $\neg$ $garb_3 \rightarrow (dolly_2$ v $carry_2)$ [in clause form]

Lecture 13 • 13

Now we need frame axioms, and just going to talk about explanatory frame axioms, since they work out nicely. There are a number of other approaches described in the paper. Frame axioms say that if I haven't said that something changes then it doesn't, or they say explicitly if I paint something it doesn't move.

Explanatory frame axioms say, for every state change, what could possibly have caused it. So for instance, if I have garbage at time one, and !garbage at time three, then I either did a dolly at time two or a carry at time two. So, for every possible initial time and for each proposition you say what it is that could have caused the proposition to have changed truth values.

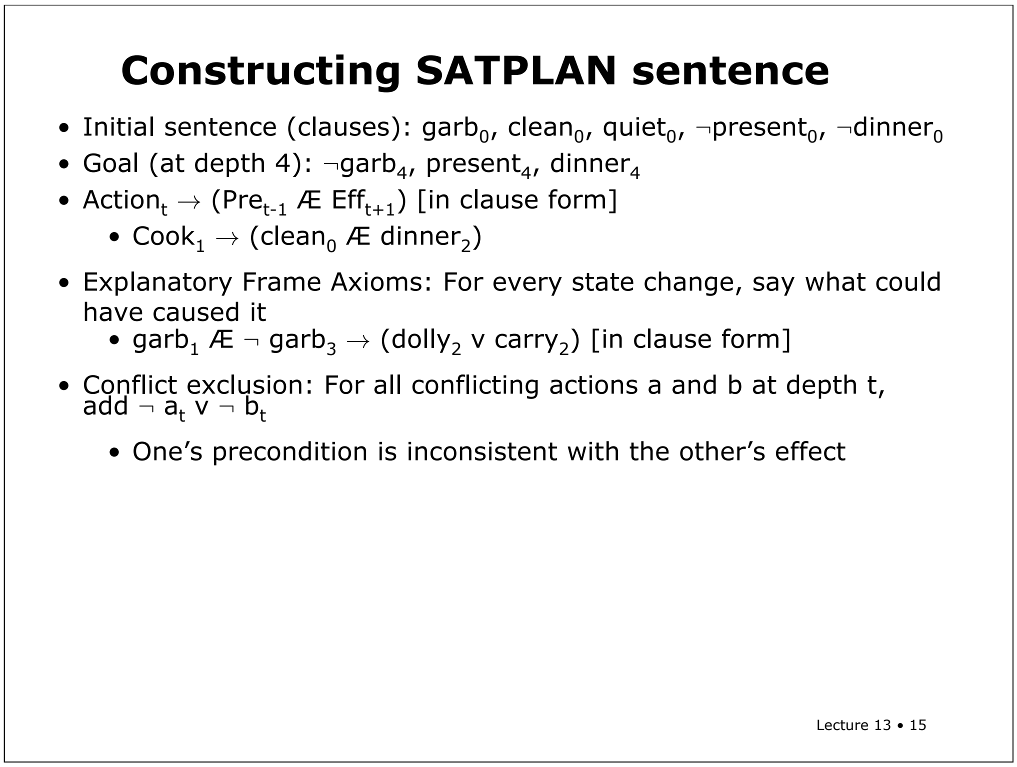
Now, you can do contrapositive reasoning. And SAT will do this in some sense implicitly for you, because, remember that if we know P implies Q, we know that !Q implies !P. Right? So for this axiom, you know that if you didn't do dolly and you didn't do carry, then it can't be that the garbage variable switched its sign. This is the only way that you could have done it. So if you didn't do one of these things, then it didn't change, and so there's your frame axiom.



Constructing SATPLAN sentence

- Initial sentence (clauses): $garb_0$, $clean_0$, $quiet_0$, $\neg present_0$, $\neg dinner_0$
- Goal (at depth 4): $\neg garb_4$, $present_4$, $dinner_4$
- $Action_t \rightarrow (Pre_{t-1}$ Æ $Eff_{t+1})$ [in clause form]
  - $Cook_1 \rightarrow (clean_0$ Æ $dinner_2)$
- Explanatory Frame Axioms: For every state change, say what could have caused it
  - $garb_1$ Æ $\neg garb_3 \rightarrow (dolly_2 \vee carry_2)$ [in clause form]
- Conflict exclusion: For all conflicting actions a and b at depth t, add $\neg a_t \vee \neg b_t$
  - One’s precondition is inconsistent with the other’s effect

There's one more set of axioms that we need to keep actions from conflicting, called conflict exclusion axioms, and then we'll be ready to go. For all conflicting actions A and B at step T we’ll add the clause !A at T or !B at T. So what's a conflicting action? Two actions conflict if one's preconditions are inconsistent with the other's effect. So if you have two actions and one's preconditions are inconsistent with another one's effect, they can't happen at the same time. It might look like they could happen exactly at the same time, but it would not be the case that you could do either linearization. So we have the same constraint that came up in GraphPlan.



Constructing SATPLAN sentence

- Initial sentence (clauses): garb$_0$, clean$_0$, quiet$_0$, $\neg$present$_0$, $\neg$dinner$_0$
- Goal (at depth 4): $\neg$garb$_4$, present$_4$, dinner$_4$
- Action$_t$ $\rightarrow$ (Pre$_{t-1}$ Æ Eff$_{t+1}$) [in clause form]
  - Cook$_1$ $\rightarrow$ (clean$_0$ Æ dinner$_2$)
- Explanatory Frame Axioms: For every state change, say what could have caused it
  - garb$_1$ Æ $\neg$ garb$_3$ $\rightarrow$ (dolly$_2$ v carry$_2$) [in clause form]
- Conflict exclusion: For all conflicting actions a and b at depth t, add $\neg$ a$_t$ v $\neg$ b$_t$
  - One’s precondition is inconsistent with the other’s effect

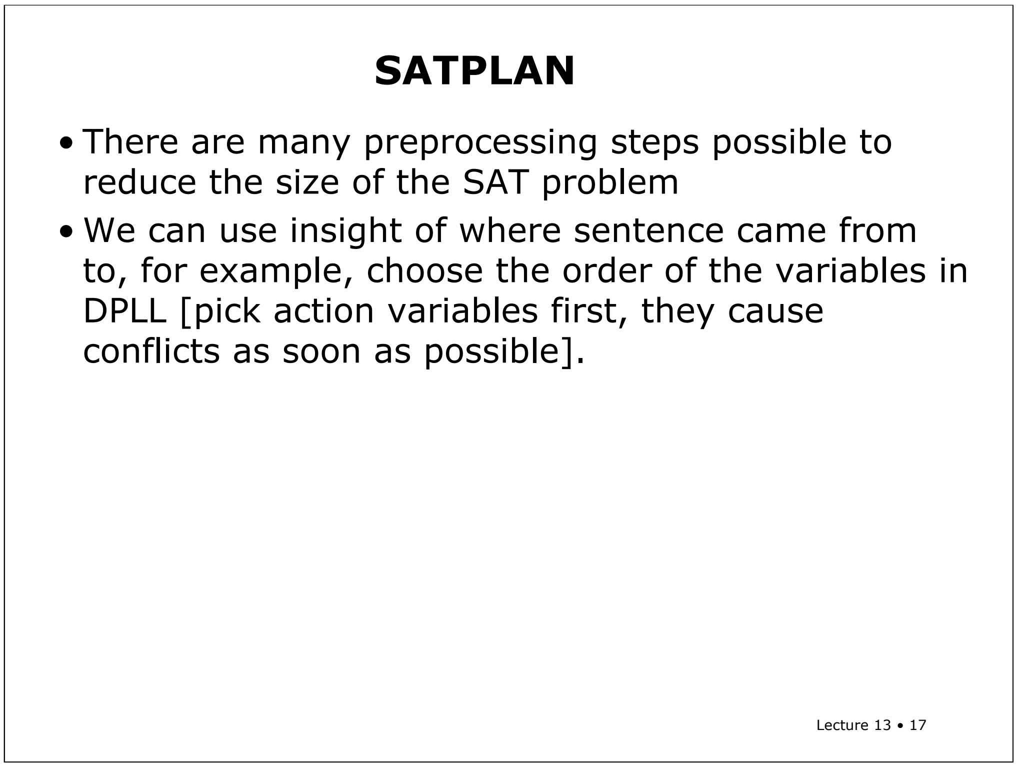
Lecture 13 • 15

Now we know how to take a planning problem and make it into a big sentence. You just take the conjunction of the unit clauses from the initial and goal conditions, and the conjunction of all the axioms that hook the actions up to their preconditions and effects and the conjunction of all the frame axioms and the conjunctions of these conflict exclusion axioms, and you clausify it all, and now you have a SAT sentence. You just feed it into DPLL or WalkSat and poof, out comes your answer. If an answer doesn't come out, you do it again for a bigger plan depth and eventually you'll get the answer out to your planning problem.

SATPLAN

- There are many preprocessing steps possible to reduce the size of the SAT problem

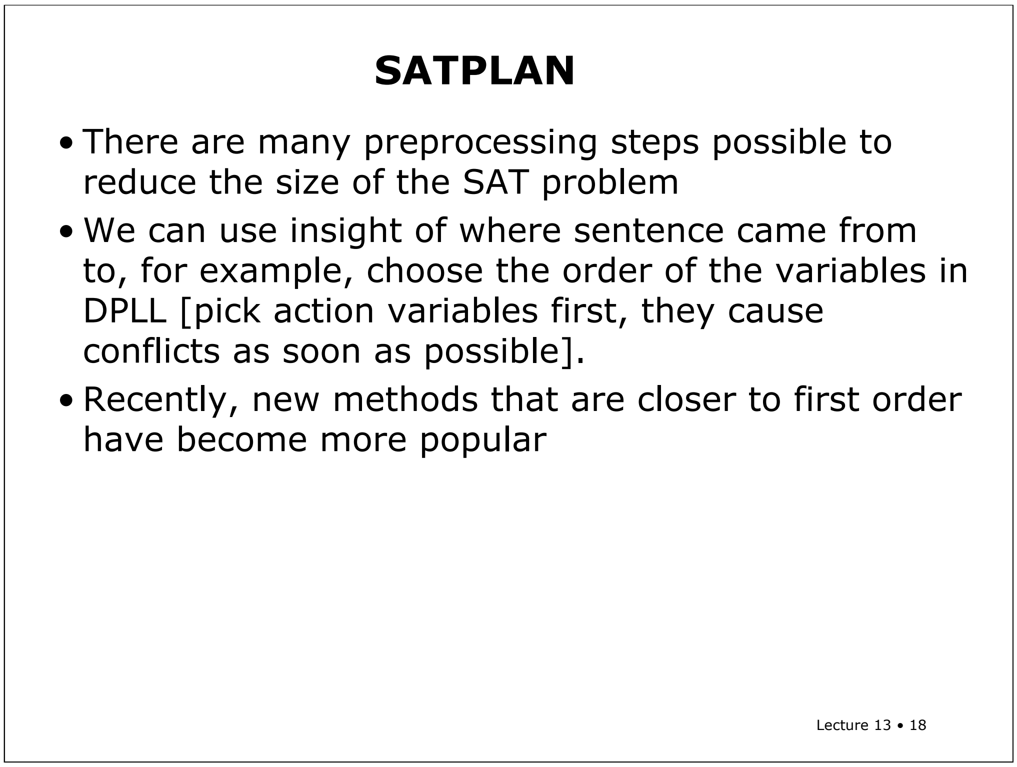
It turns out that there's room to be very clever in the construction of your SAT sentence. The method we just looked at is perhaps the most straightforward, but it isn’t the most efficient to solve using SAT. You can be much cleverer, and use a lot of preprocessing to shrink the size of the sentence.



SATPLAN

- There are many preprocessing steps possible to reduce the size of the SAT problem
- We can use insight of where sentence came from to, for example, choose the order of the variables in DPLL [pick action variables first, they cause conflicts as soon as possible].

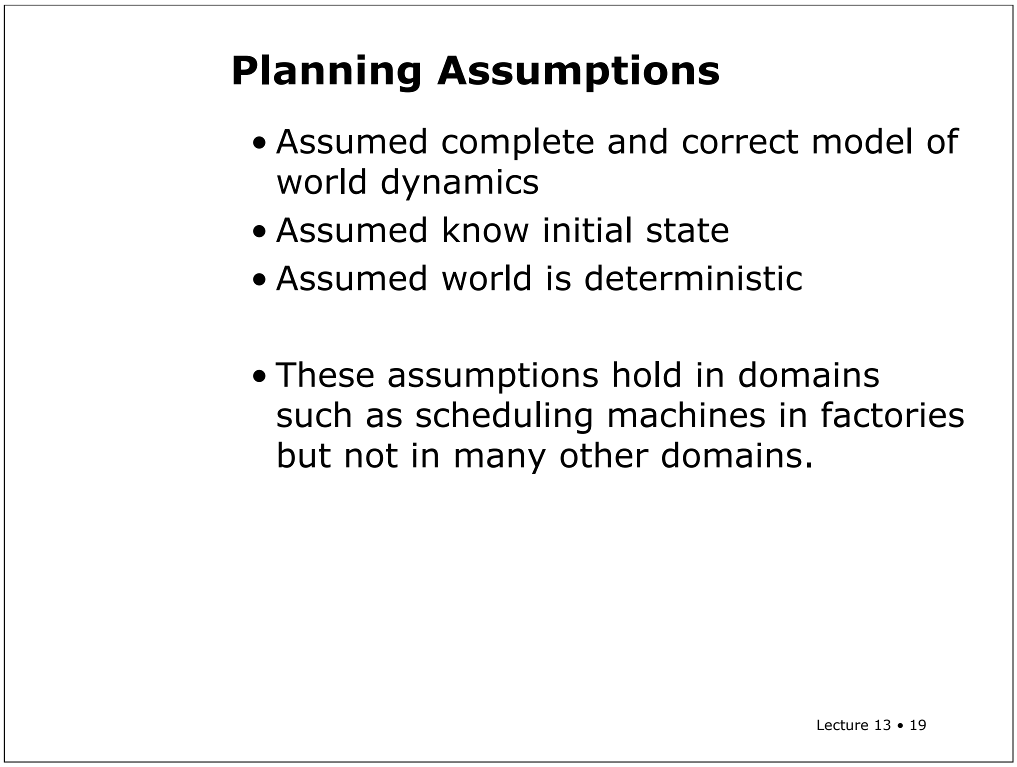
Also, if you're using DPLL, people have found that converting this to a DPLL sentence and forgetting where it came from isn't as effective as noticing that DPLL works by picking variables to assign in some order and it turns out that you can be cleverer about choosing the order of the variables to assign by knowing where they came from. So, in particular, the action variables are good ones to assign first in DPLL, because those are the things that really will cause the conflicts as quickly as possible. So you can use your insight about where this sentence came from in order to search the space more effectively.



SATPLAN

- There are many preprocessing steps possible to reduce the size of the SAT problem
- We can use insight of where sentence came from to, for example, choose the order of the variables in DPLL [pick action variables first, they cause conflicts as soon as possible].
- Recently, new methods that are closer to first order have become more popular

Up until about maybe three years ago or so, GraphPlan and SATPlan were the best things to use for hard planning problems, and they won all the planning contests. Now, more recently, people have gone back to these methods that are a little bit more first-orderish; they keep the structure of the original problem around and take advantage of it. They also use search heuristics to great effect.



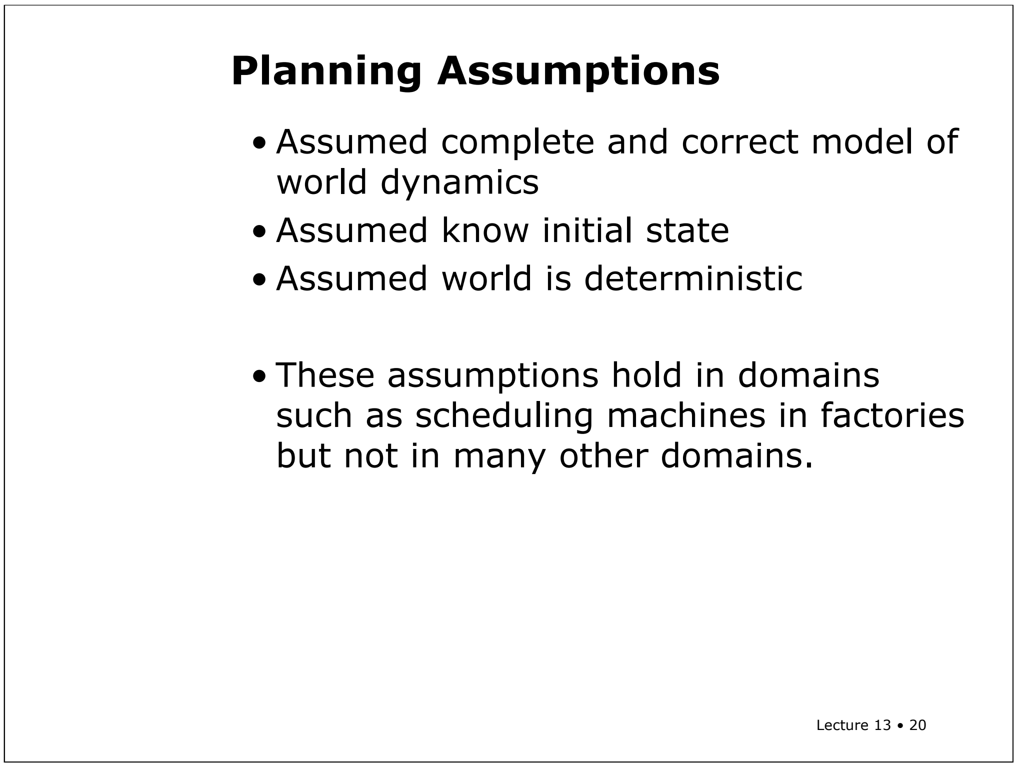
Planning Assumptions

- Assumed complete and correct model of world dynamics
- Assumed know initial state
- Assumed world is deterministic

- These assumptions hold in domains such as scheduling machines in factories but not in many other domains.

Lecture 13 • 19

In all the planning methods we've looked at so far, we've assumed a couple things. We've assumed that we know a complete and correct model of the world dynamics, which is encoded in the operator descriptions. And we've assumed that we know the initial state. And assumed that the world is deterministic. That is to say, whenever the world is in some state and we take an action, then it does whatever the operator description tells us it's going to do. So this is related to knowing a complete and correct model. But not only is it a complete and correct model, it's a deterministic model. Now, there are some kinds of sort of formal domains for which these kinds of assumptions are true. So, for instance, planning methods just like the ones we've been looking at have been applied in real application domains for things like scheduling, where the assumption is you're already working in an abstraction of a domain that satisfies these assumptions and is close enough to right. But for all kinds of other domains, these assumptions are completely untenable.

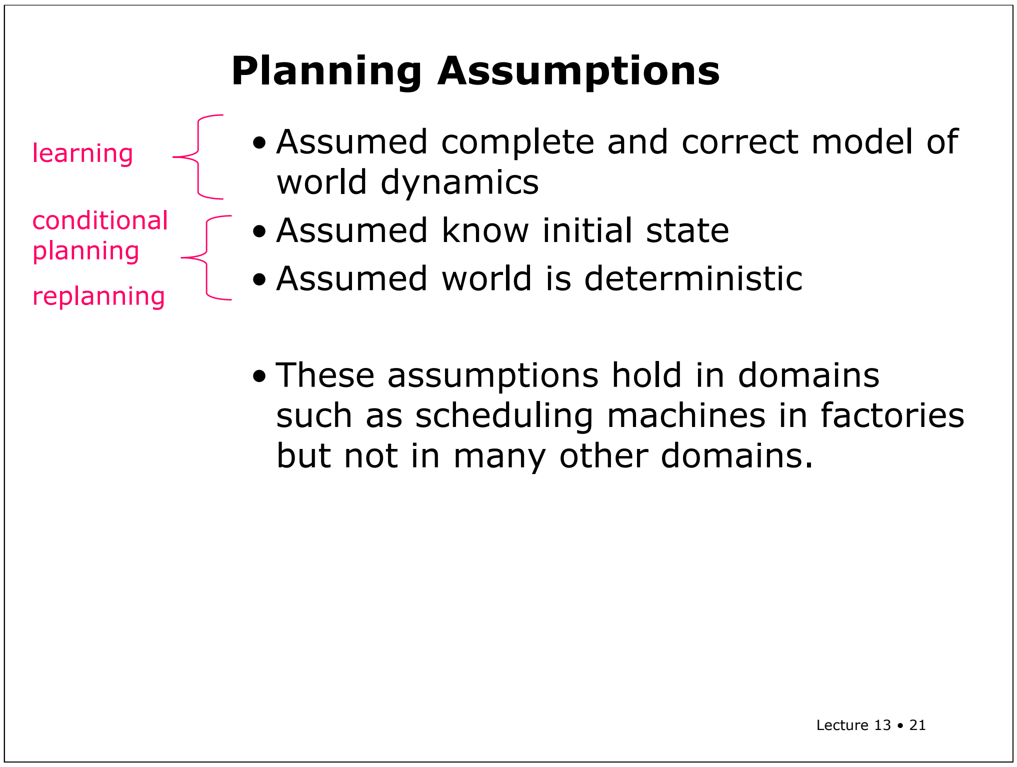


Planning Assumptions

- Assumed complete and correct model of world dynamics
- Assumed know initial state
- Assumed world is deterministic

- These assumptions hold in domains such as scheduling machines in factories but not in many other domains.

So, we'll finish this lecture by talking about ways of addressing some of these problems without moving directly yet to probabilistic representation. But it will be a motivation for going to probability pretty soon.



Planning Assumptions

learning

- Assumed complete and correct model of world dynamics

conditional planning

replanning

- Assumed know initial state
- Assumed world is deterministic

- These assumptions hold in domains such as scheduling machines in factories but not in many other domains.

The assumption of knowing a complete and correct model we'll address later on when we study learning. So we're not going to get to this problem today. But we can address, at least in a limited way, problem of not knowing the initial state, and the assumption that the world is deterministic. We're going to look at some conditional planning methods that are appropriate when you don't know everything about the state of the world. And we're going to look at re-planning, which is appropriate when the world is nondeterministic, when there can be errors in the execution of the actions, but you are not prepared or interested in modeling those errors in advance.

Conditional Planning Example

Let’s consider the following example to motivate conditional planning. You want to go to the airport and board your plane. But you don’t know, when you’re making your plan, which gate your flight leaves from. However, you feel confident that if you get to the airport lobby, you can read the display that will tell you what gate your airplane is leaving from.

Conditional Planning Example

Action	Preconditions	Effects

We can formalize this domain using simple operators, as follows. We're assuming that this is an incredibly small airport with only two gates. The variable Gate1 is true if your plane is leaving from gate 1; if your plane leaves from Gate 2, then it's false.

Conditional Planning Example

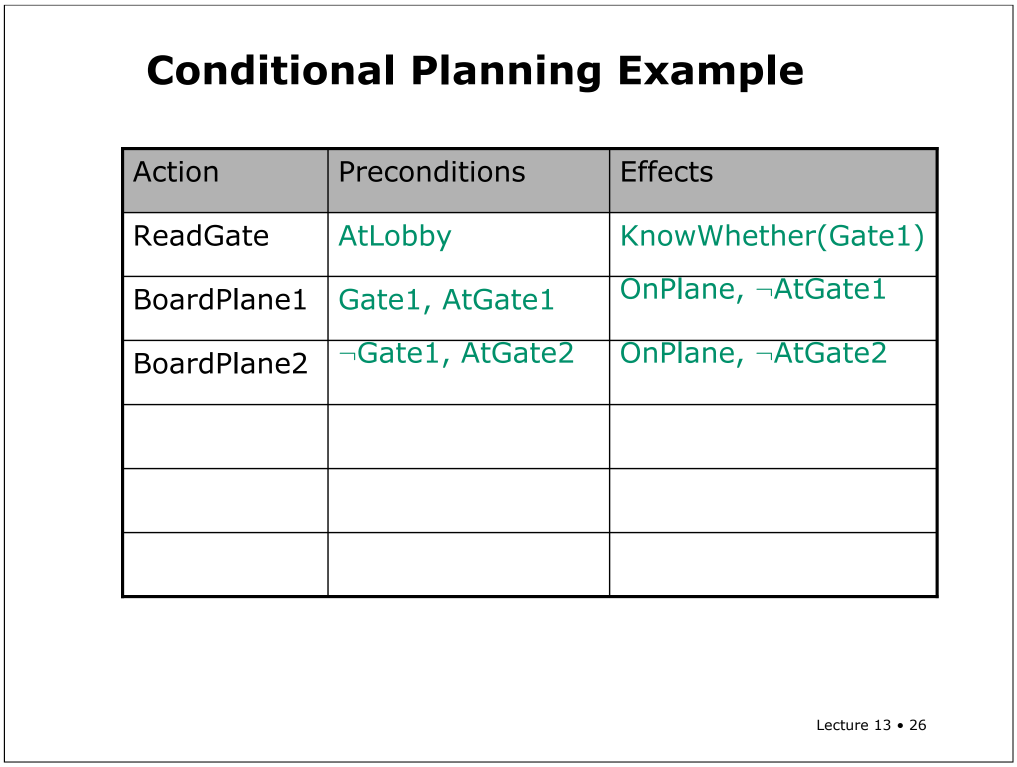
Action	Preconditions	Effects
ReadGate	AtLobby	KnowWhether(Gate1)

The **read gate** action has the precondition that you're at the lobby. And it has an interesting effect. It doesn't change the state of the world. It just changes the knowledge state of the agent. As a result of doing this action, the agent **knows whether** the variable Gate1 has value true or false; that is, the agent knows whether its plane is leaving from gate1 or gate 2.

Conditional Planning Example

Action	Preconditions	Effects
ReadGate	AtLobby	KnowWhether(Gate1)
BoardPlane1	Gate1, AtGate1	OnPlane, ¬AtGate1

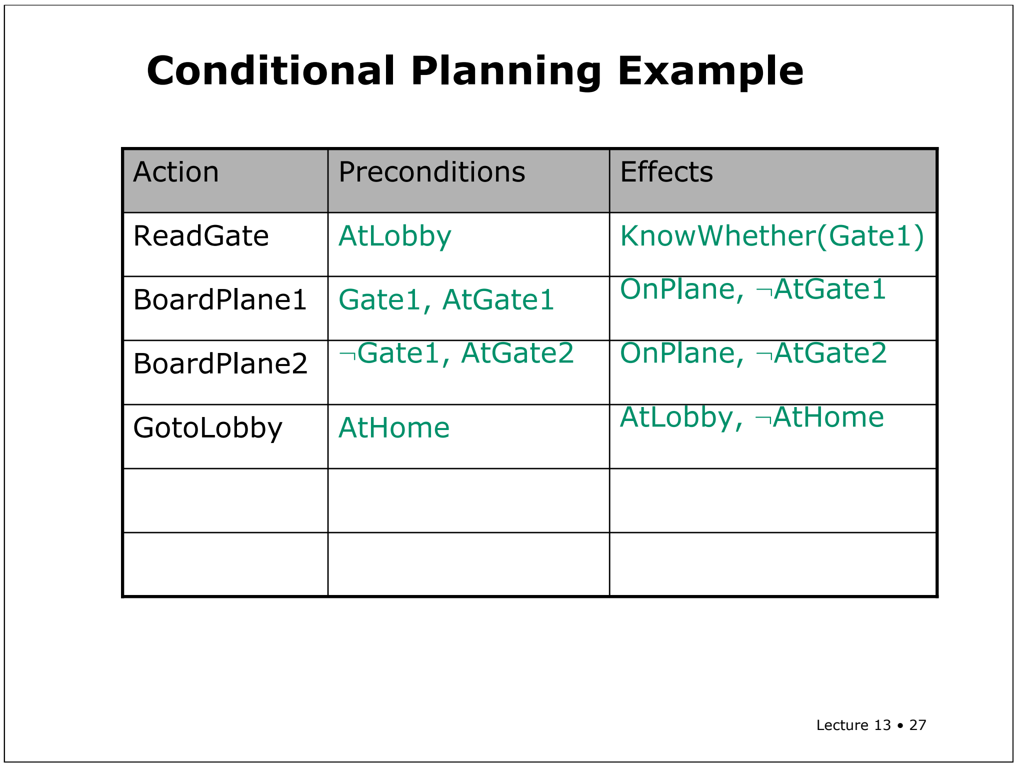
The other actions are what you might expect. You can board plane 1 if you're at gate 1 and if your plane is leaving from that gate.



Conditional Planning Example

Action	Preconditions	Effects
ReadGate	AtLobby	KnowWhether(Gate1)
BoardPlane1	Gate1, AtGate1	OnPlane, ¬AtGate1
BoardPlane2	¬Gate1, AtGate2	OnPlane, ¬AtGate2

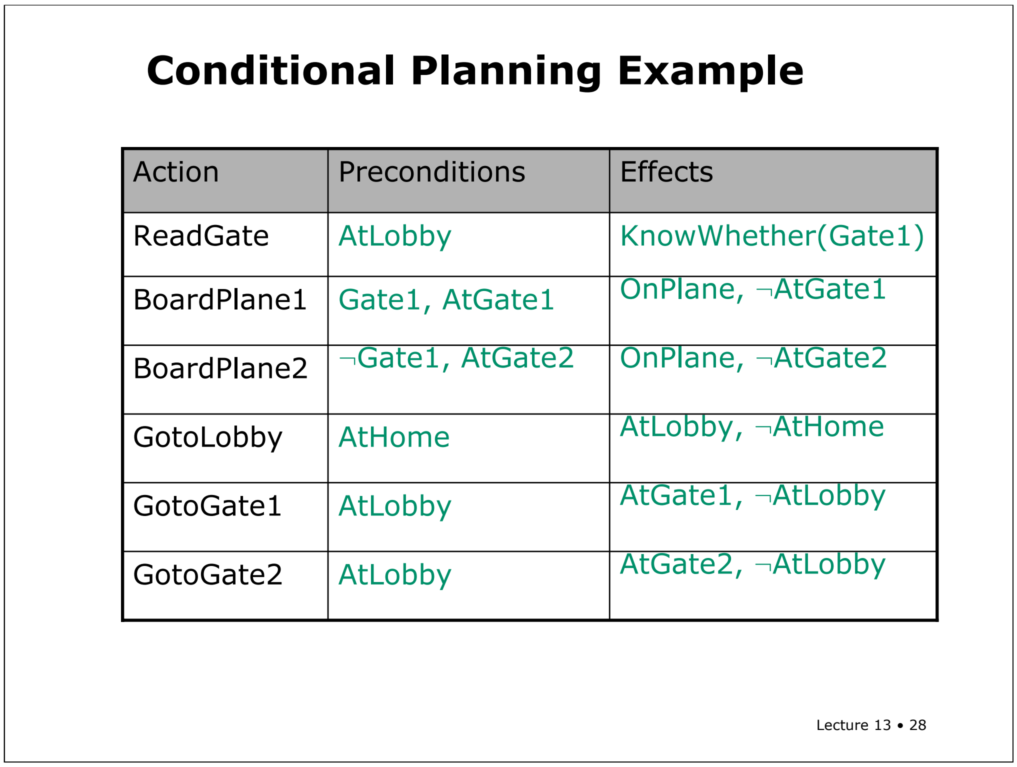
The same thing is true for the board plane 2 action.



Conditional Planning Example

Action	Preconditions	Effects
ReadGate	AtLobby	KnowWhether(Gate1)
BoardPlane1	Gate1, AtGate1	OnPlane, ¬AtGate1
BoardPlane2	¬Gate1, AtGate2	OnPlane, ¬AtGate2
GotoLobby	AtHome	AtLobby, ¬AtHome

If you're at home, you can go to the lobby.



Conditional Planning Example

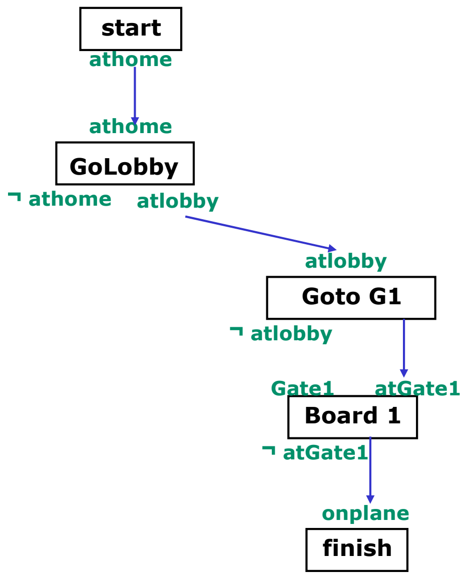
Action	Preconditions	Effects
ReadGate	AtLobby	KnowWhether(Gate1)
BoardPlane1	Gate1, AtGate1	OnPlane, ¬AtGate1
BoardPlane2	¬Gate1, AtGate2	OnPlane, ¬AtGate2
GotoLobby	AtHome	AtLobby, ¬AtHome
GotoGate1	AtLobby	AtGate1, ¬AtLobby
GotoGate2	AtLobby	AtGate2, ¬AtLobby

And, if you’re at the lobby, you can go to either gate. Note that knowing where your plane is leaving from isn’t a precondition for going to either gate. The assumption here is that you can wander around in confusion among the gates all you want; you just can’t board the wrong plane.

Partial Order Conditional Plan

Now let's look at how a conditional version of the POP algorithm might work. I'm not going to go through the algorithm in complete detail; I'll just sketch out an example of how it might work in this airport domain.

Partial Order Conditional Plan

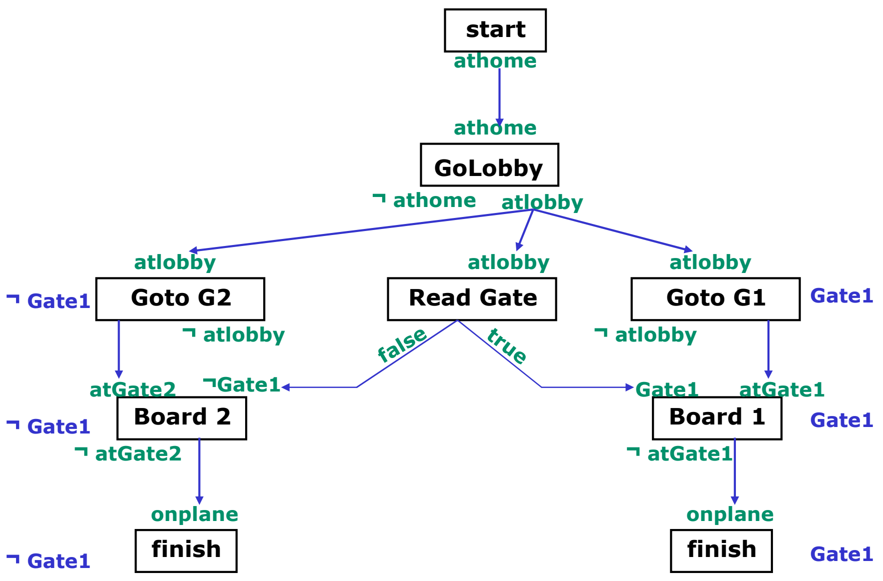


Given the initial condition at-home and goal condition on-plane, it's pretty easy to make this much of the plan.

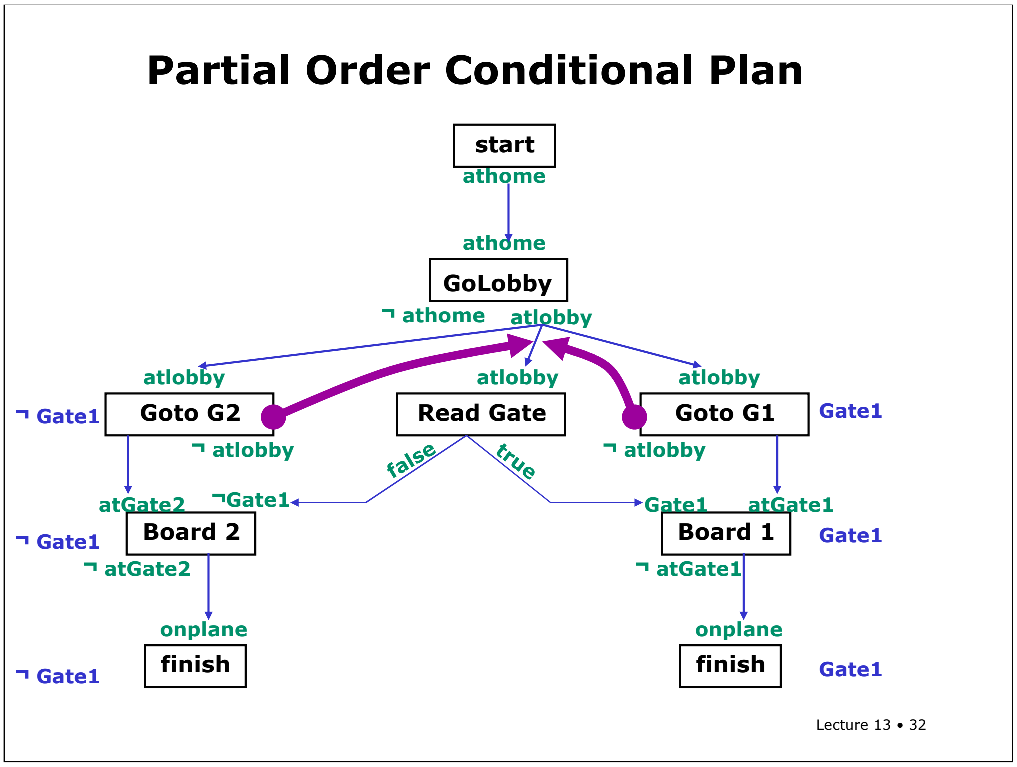
If you look carefully, you can see that all of the preconditions of all the actions are satisfied, except for the "Gate1" precondition of the Board1 action. Now we have a bit of a problem, because we don't have any actions that can cause Gate1 to be true. You can't in general influence the airport operations people to park your airplane wherever you want it!

The only thing we can do about Gate1 is to take the ReadGate action, and find out **whether** Gate1 is true. If we decide to add this step to our plan, then we have to divide the plan up into two separate contexts, one in which Gate1 is true and one in which it is false. And in future, when we check for conflicts, we only look **within** a context. Because we know, ultimately, that we'll only go down one branch of the plan.

Partial Order Conditional Plan



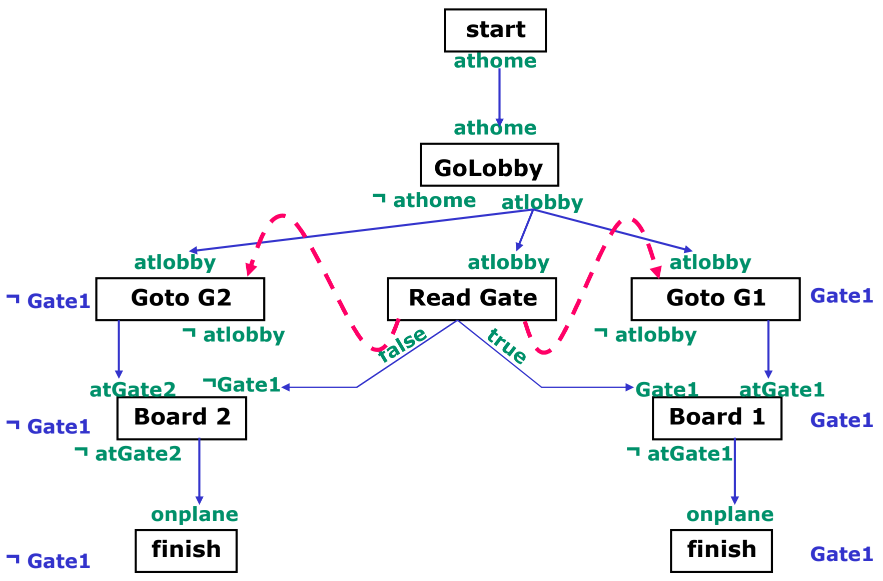
So, here’s a mostly worked-out plan that includes the read gate action, which divides the plan into two branches. The context conditions (Gate1 and not Gate1) are indicated in blue next to the relevant steps. It seems basically good, but we still have to watch out for threats.



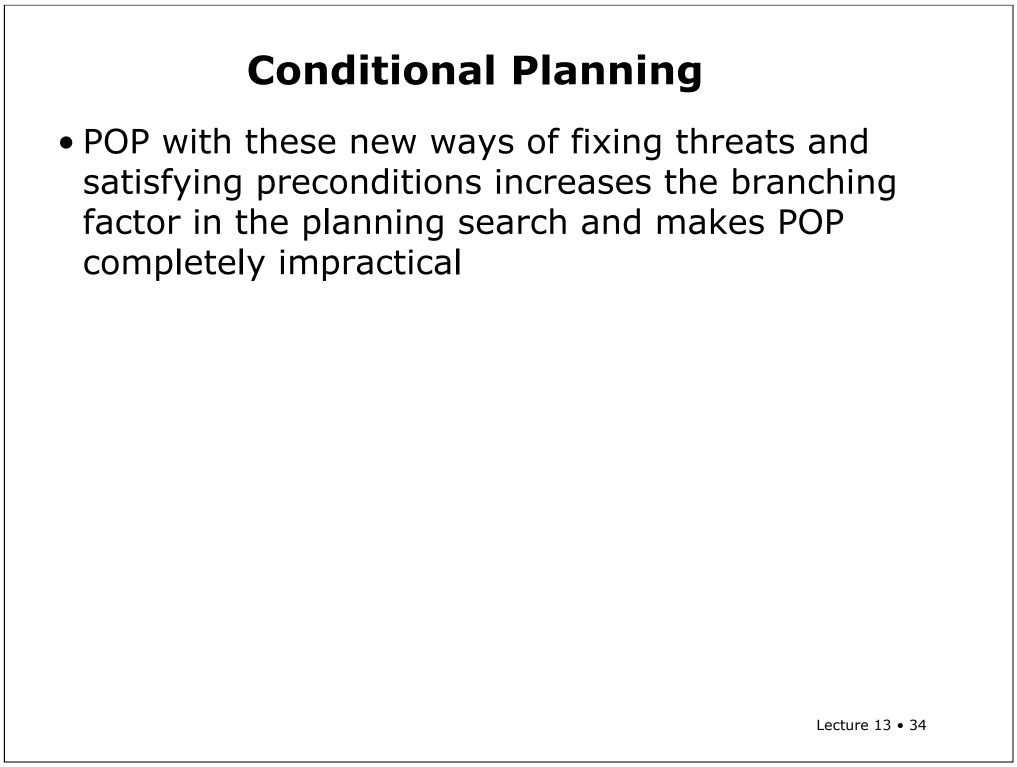
Partial Order Conditional Plan

We have two threats to deal with, one in each context of the plan. Since the Read Gate and Go Lobby actions are not in any context, they're considered to be part of both plans. And we can see the problem that GotoG1 or GotoG2 might execute before ReadGate, which would cause the deletion of atLobby, which is necessary for ReadGate to work.

Partial Order Conditional Plan



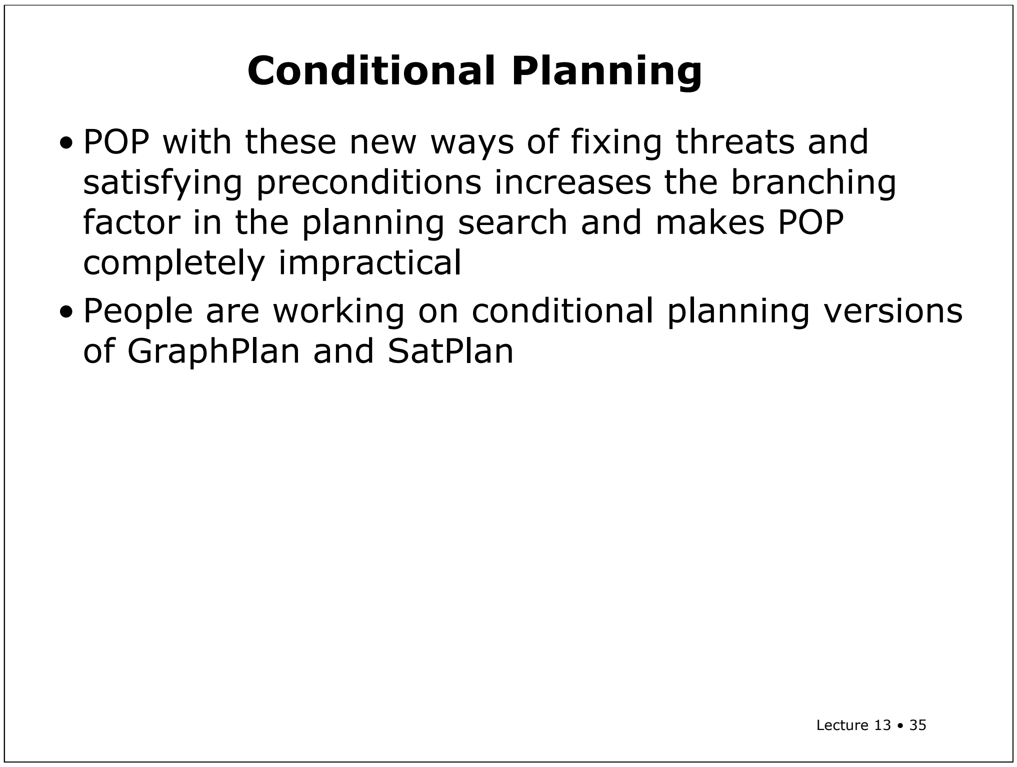
So, we fix the problem by adding a couple of temporal constraints, and we’re done.



Conditional Planning

- POP with these new ways of fixing threats and satisfying preconditions increases the branching factor in the planning search and makes POP completely impractical

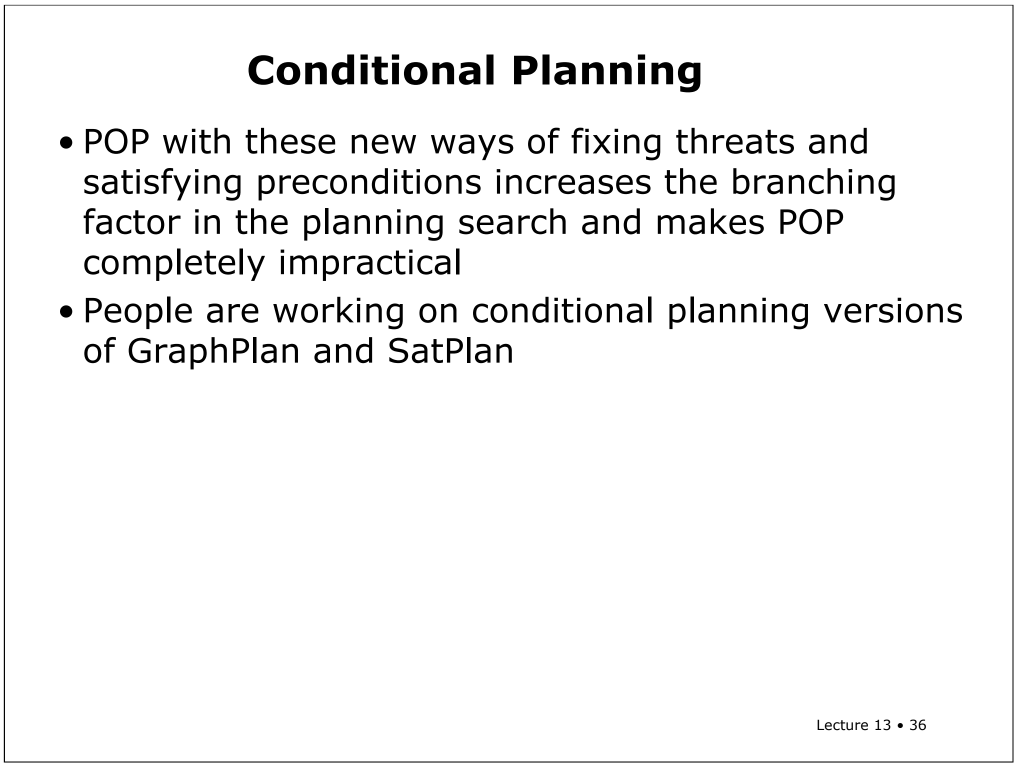
So, this is one way to do conditional planning. At some level it's not too hard to talk about, but at another level it makes POP, which is already not the most efficient planning method pretty much go out of control. So this is a case of something that you can write down, and you can kind of make it work, but adding these new ways of fixing threats and satisfying preconditions means that the branching factor gets really big. Now we're basically at frontier of classical planning research.



Conditional Planning

- POP with these new ways of fixing threats and satisfying preconditions increases the branching factor in the planning search and makes POP completely impractical
- People are working on conditional planning versions of GraphPlan and SatPlan

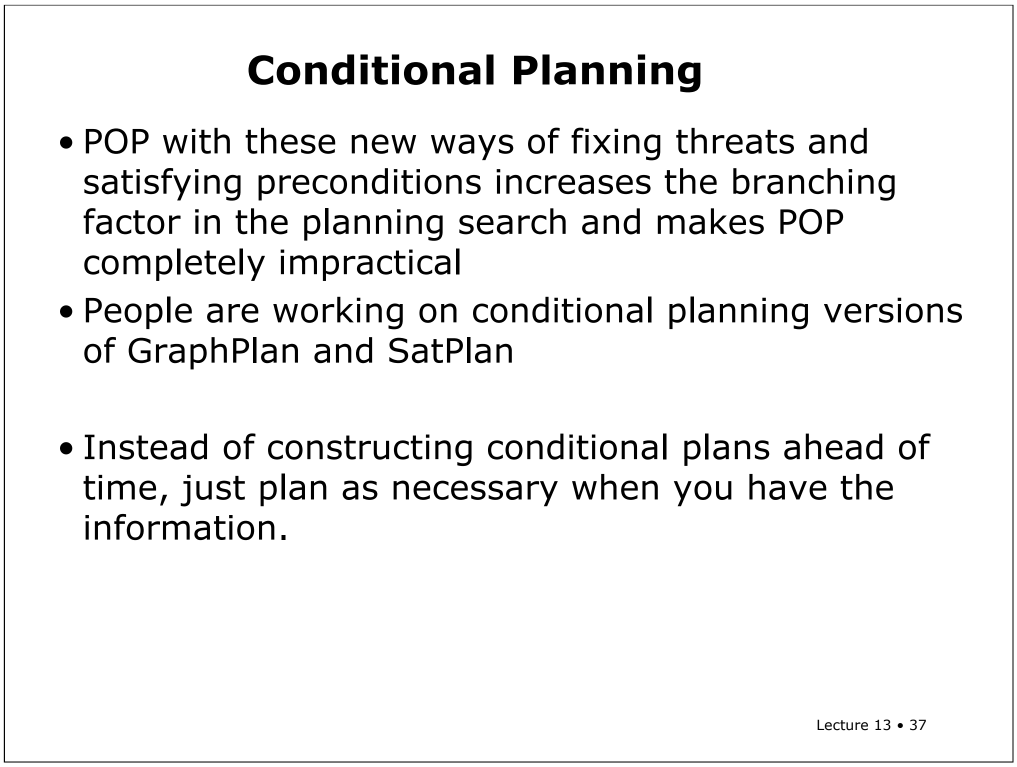
Another fairly current research topic is the development of methods for making GraphPlan and SatPlan build conditional plans. There are algorithms for this; I’m not sure anyone really understands how practical they are yet.



Conditional Planning

- POP with these new ways of fixing threats and satisfying preconditions increases the branching factor in the planning search and makes POP completely impractical
- People are working on conditional planning versions of GraphPlan and SatPlan

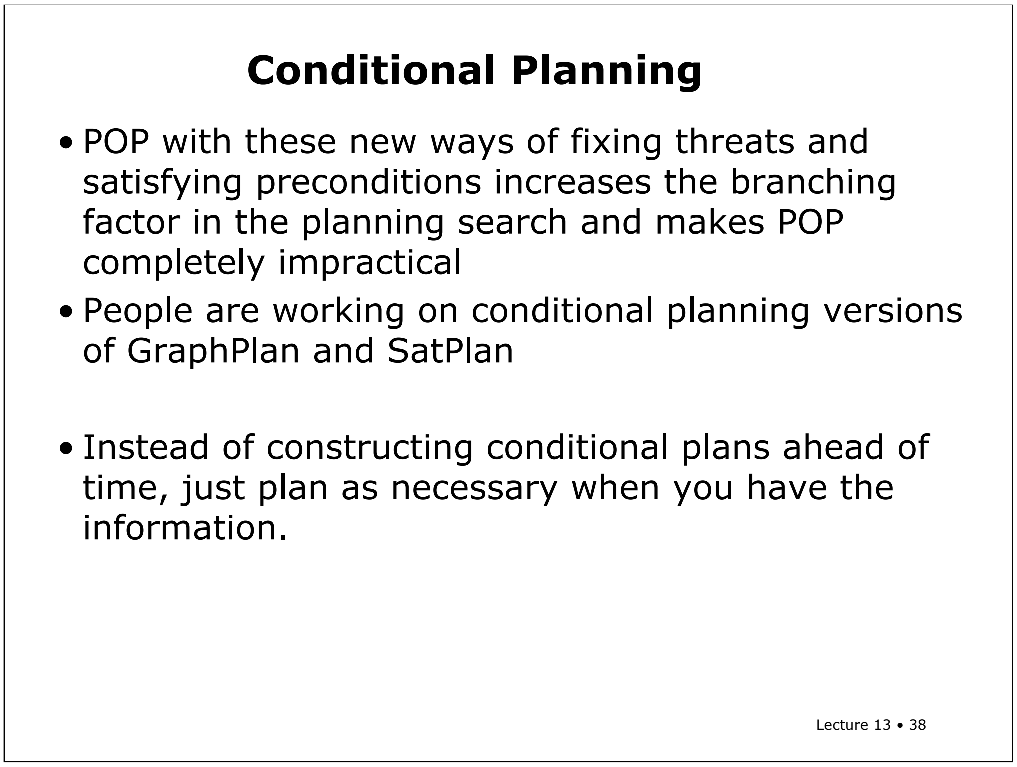
There are really two alternative stories that you could tell about what you do when you go to the airport. One is that, sitting at home or in the car or in the taxi or whatever, you make a conditional plan that says I'm going to go to the lobby, I'm going to look at the board, if it tells me I have to go somewhere that I go to by train then I'll go there by train and otherwise if it tells me to somewhere that I need to go by foot I'll go by foot.



Conditional Planning

- POP with these new ways of fixing threats and satisfying preconditions increases the branching factor in the planning search and makes POP completely impractical
- People are working on conditional planning versions of GraphPlan and SatPlan
- Instead of constructing conditional plans ahead of time, just plan as necessary when you have the information.

But that seems like a pretty unlikely story, right? All the time I go to airports where I have no idea whether they're going to involve taking trains to gates or not. Sometimes they do, sometimes they don't. So another story is that I shouldn’t worry too much in advance; I should just plan later on, when I have the information. Be a little more relaxed and plan on-line as the information becomes apparent to you.



Conditional Planning

- POP with these new ways of fixing threats and satisfying preconditions increases the branching factor in the planning search and makes POP completely impractical
- People are working on conditional planning versions of GraphPlan and SatPlan
- Instead of constructing conditional plans ahead of time, just plan as necessary when you have the information.

Now, that sort of approach doesn't suit NASA mission control. It doesn't suit people who want to have a theorem at the very beginning that they're going to for sure know exactly how to do what they're going to do. But for most of what we need to do in life, there are too many conditions to do conditional planning, and so a more relaxed approach often works better. So let's talk a little bit about that.

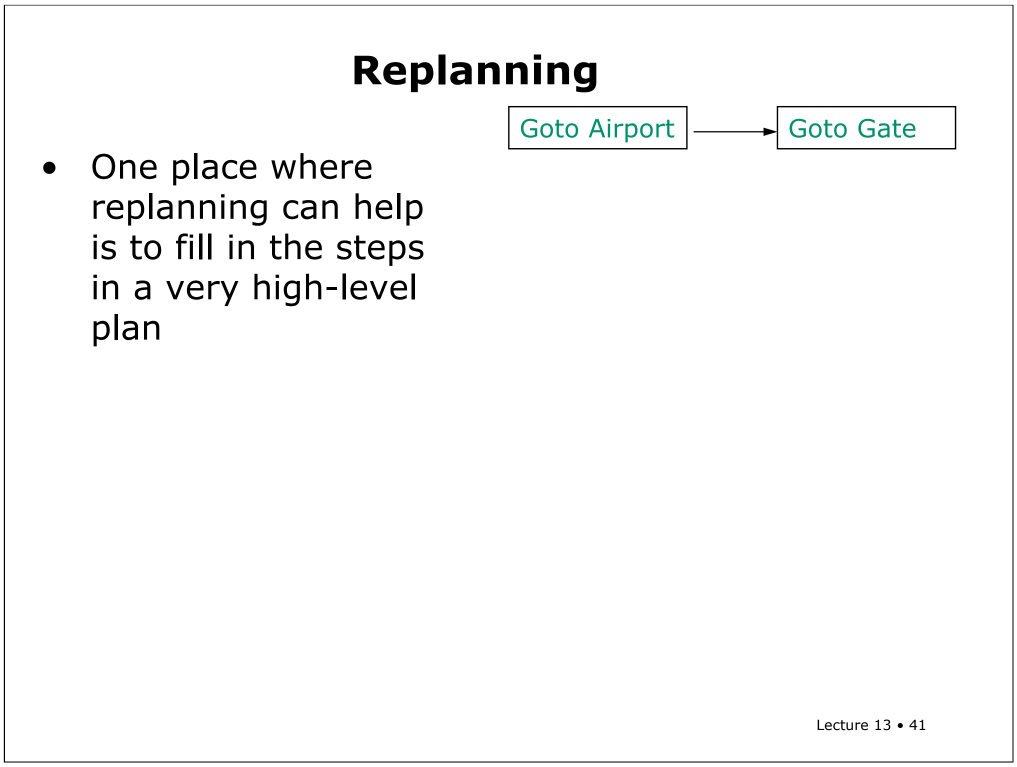
Replanning

There are (at least) two things that re-planning is good for.

Replanning

- One place where replanning can help is to fill in the steps in a very high-level plan

One is this case that we just talked about, where we don't really know yet how to do the thing that we're going to need to do. In that case, it's almost as if you can make a plan at a very high level of abstraction. You say, well, I'm going to go to the airport, and then I'm going to go to the gate, and then I'm going to do some other stuff. And you can go to the airport with a plan at this level of abstraction. So there's an idea that you might have a plan at a very high level of abstraction, but the details of how to go to the gate you don't know.



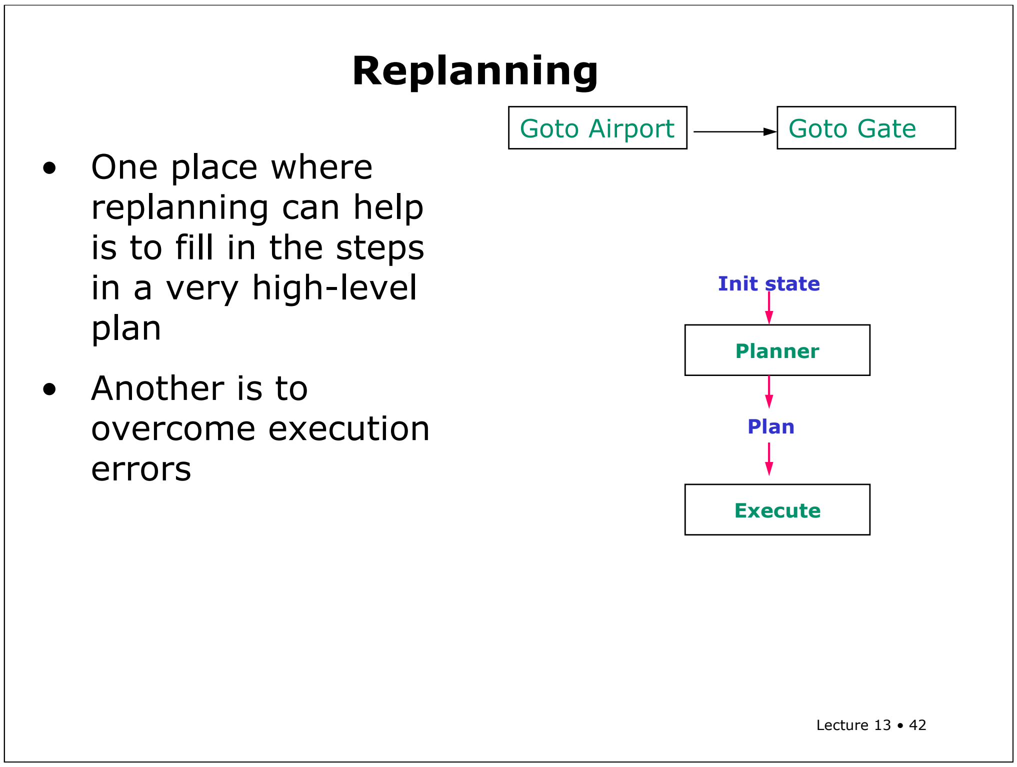
Replanning

$$\begin{tikzcd}\boxed{\text{Goto Airport}} \arrow[r] & \boxed{\text{Goto Gate}}\end{tikzcd}$$

- One place where replanning can help is to fill in the steps in a very high-level plan

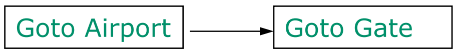
Lecture 13 • 41

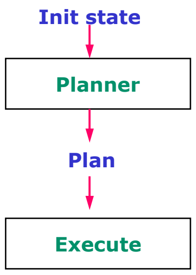
So, you’ll plan how to get to the airport, and you’ll start executing the plan. Once you get to the airport, you’ll get more relevant information, and you’ll call your planner again to figure out the rest of the plan (or maybe just another reasonable initial prefix, like what to do until you get to your destination airport, where you’ll probably have to do information gathering again).



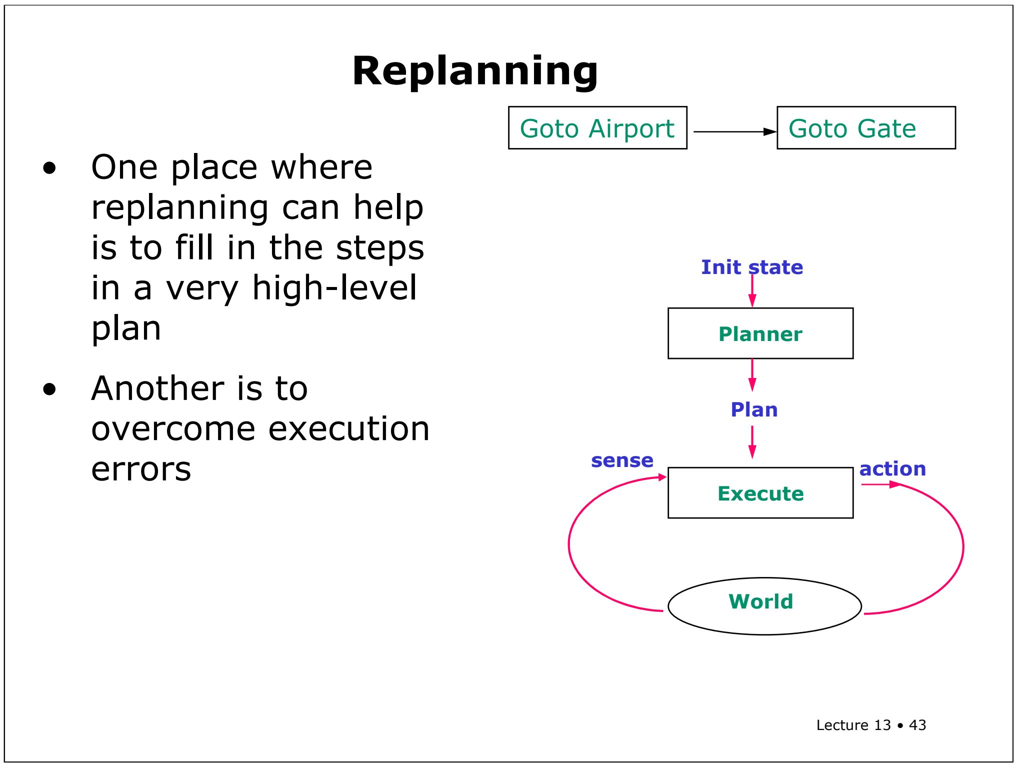
Replanning

- One place where replanning can help is to fill in the steps in a very high-level plan
- Another is to overcome execution errors



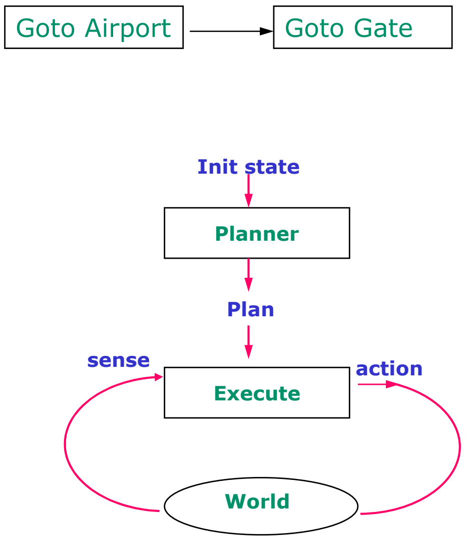


Another useful situation for re- planning is not a case of not having information in advance, but a case of having our model be not quite right, having execution errors happen. It’s easy to think of a situation in which you make a plan from your current state as the initial state. Then you start to execute the plan. If you made a plan that said to do four steps, then you could just execute them “open loop”, without looking at the world to see if things are going right.



Replanning

- One place where replanning can help is to fill in the steps in a very high-level plan
- Another is to overcome execution errors

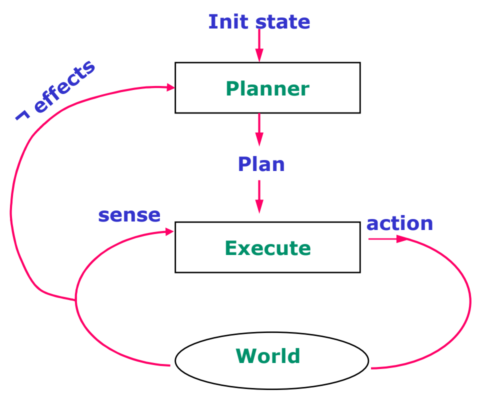


But it would be much more robust to execute them “closed loop”. When you were making your plan, you knew what the desired effects were of each step. When executing the plan, you can watch in the world to be sure that your actions are really having their desired effects.

Replanning

$$\begin{tikzcd}\boxed{\text{Goto Airport}} \arrow[r] & \boxed{\text{Goto Gate}}\end{tikzcd}$$

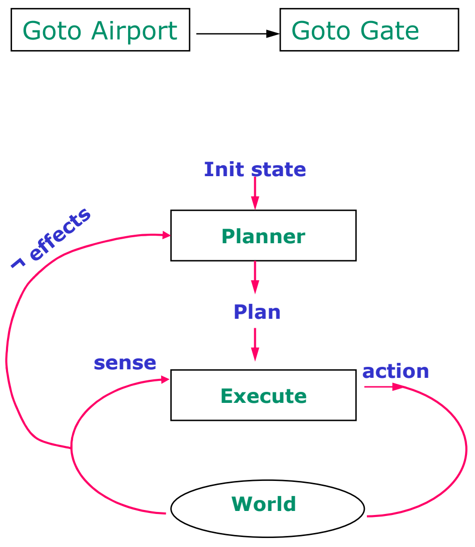
- One place where replanning can help is to fill in the steps in a very high-level plan
- Another is to overcome execution errors



If you find that they’re not working correctly, you can stop executing what is now probably a senseless sequence of actions and re-plan. You’d call your planner again with the current, unpredicted, state as the initial state. Then you’d get a new plan and start executing it, continuing to monitor the expected effects.

Replanning

- One place where replanning can help is to fill in the steps in a very high-level plan
- Another is to overcome execution errors



Replanning Cycle

That’s a moderately flexible way of dealing with the world messing up our plan. It doesn't try to anticipate anything. It doesn't try to think about what could go wrong and what to do if it did. It doesn't have a proof in its head that it can deal with all the things that might go wrong. It just says I'm going to pretend that the world is deterministic; I'm going to make a plan that's good. If it is, I'm going to execute it; but I'm actually going to keep my eye out. I'm actually going to pay attention to see if things start to go wrong; and if they do, I'll plan again.

Universal Plan

It might be that you're worried about computation time, that you're in a domain that has so much time pressure that you're worried that if you stop and re-plan, the bad guys will get you while you're thinking. Your race car will run into the wall, or some kind of bad time-critical thing will happen. In that case you might be worried about ever calling the planner because, as you know, these algorithms aren't always quick. And so there's a danger that you call the planner and it takes forever and there you are hung up and not knowing what to do.

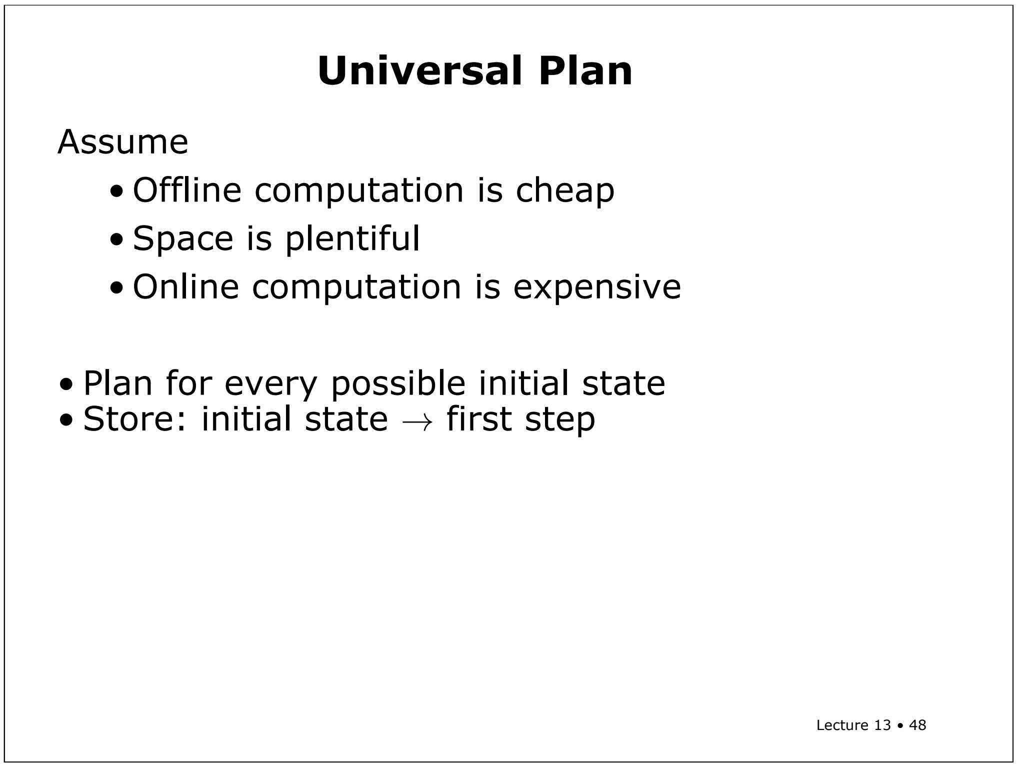
One way to handle this is with something called a “universal plan”.

Universal Plan

Assume

- Offline computation is cheap
- Space is plentiful
- Online computation is expensive

You've heard in other contexts about the idea of a time-space tradeoff. The idea of universal planning is at the opposite extreme from re-planning in the time-space tradeoff spectrum. In universal planning the idea is that off-line computation is cheap, that space is cheap and plentiful, and that on-line computation is expensive. Now, so, if those three conditions hold for your domain, then it might be worth thinking really hard in advance, really kind of preparing yourself for everything that could happen, so that when you go out there into the world you can just do it. You don't ever have to stop and think.



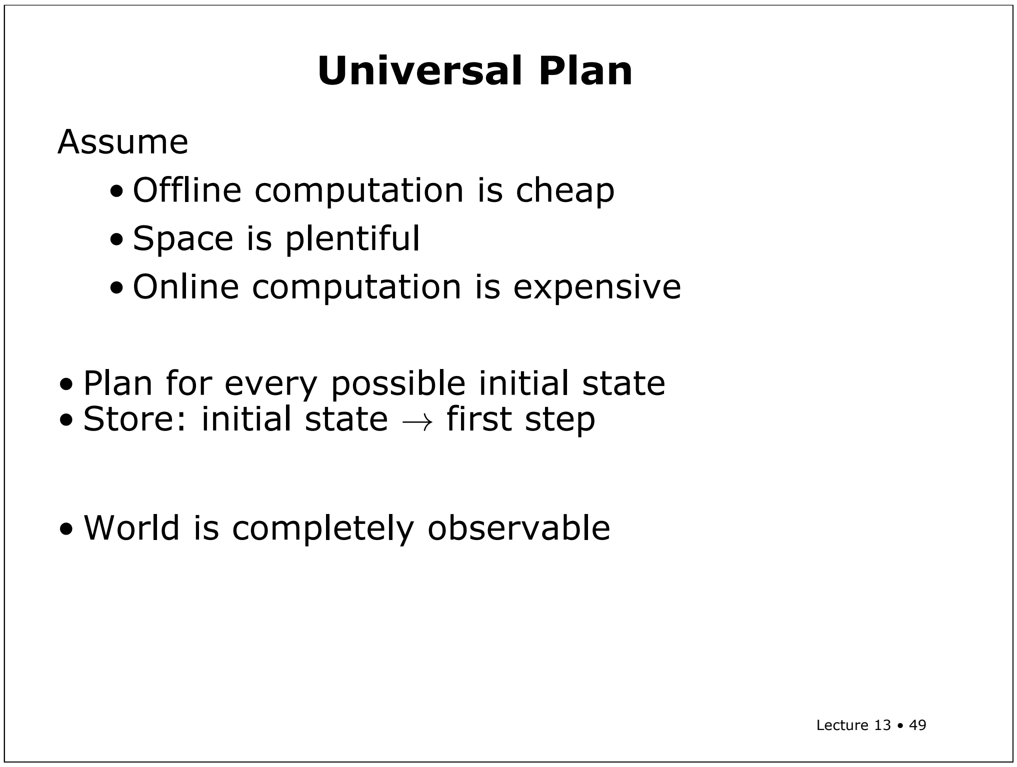
Universal Plan

Assume

- Offline computation is cheap
- Space is plentiful
- Online computation is expensive

- Plan for every possible initial state
- Store: initial state $\rightarrow$ first step

At some level this idea is crazy, but it's worth talking about because it's the extreme end of the spectrum, of which the intermediate points are interesting. So what do you do? You plan for every possible initial state, and you store a mapping from the initial state into the first action of the plan. You think really hard in advance of ever taking any actions and you say, if the world is like this, then I would have to do these ten actions in order to get to the goal. Now, you might think you would have to store all ten of those actions, but you don't. You just have to store the first one, because as long as you execute the first one, and assuming that you can see what the world is like after that, then you just go look the next state up in the table somewhere. Now, you could compute this table by dynamic programming. It's not as horrible as it seems, and we'll actually talk about doing something like this in the probabilistic case. I'm not going to go through it in the deterministic case.



Universal Plan

Assume

- Offline computation is cheap
- Space is plentiful
- Online computation is expensive

- Plan for every possible initial state
- Store: initial state $\rightarrow$ first step

- World is completely observable

There's one more assumption here, which is that the world is completely observable, meaning that we can really see what state the world is in. So, in every time step we would take an action, look to see what state the world is in, look it up in our table, do what action our table told us to do, see what state the world is in, and so on. OK, so that's one extreme.

Universal Plan

Assume

- Offline computation is cheap
- Space is plentiful
- Online computation is expensive

- Plan for every possible initial state
- Store: initial state $\rightarrow$ first step

- World is completely observable

The other extreme in some sense is replanning, where we don't store very much at all. All we store is this one little plan, but we might find ourselves having to think pretty hard on-line. I'm going to talk about one point that's in between these two things, mostly because I think it's neat and because it gives us some ideas about how to interpolate between these two approaches. In any interesting-sized domain, universal planning is way too expensive. Off-line computation can never be cheap enough, and space can never be plentiful enough, in a big domain - in the domain of your life. Why is your brain not simply a stored table of situations to actions? Well, the answer is that the table would just be way, way, way, way too big. So sometimes you have to stop and recompute.

Triangle Tables

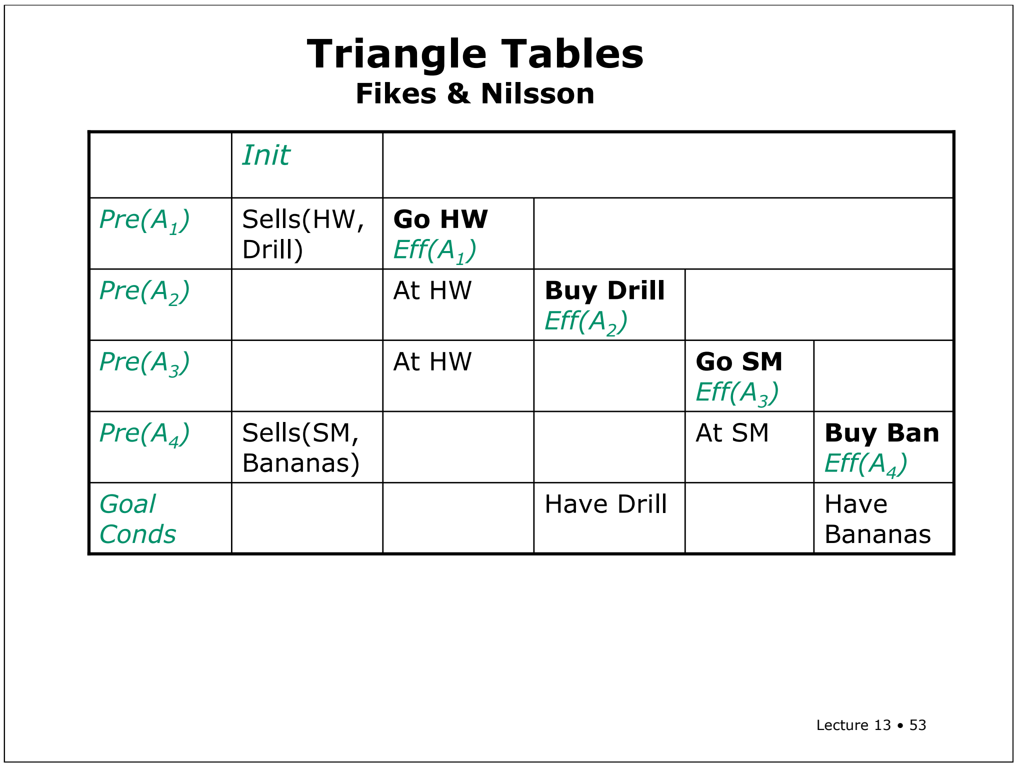
Fikes & Nilsson

So let's talk about an intermediate version, called a triangle table. These things were actually invented by Fikes and Nilsson as part of the original Strips planner. Shakey the robot really used Strips to figure out what to do, and Shakey was an actual robot. The people who worked on it invented all kinds of things that were really important and in many ways haven't been superseded.

Triangle Tables

Fikes & Nilsson

Let's go back to the hardware store, drill, bananas, milk, supermarket example. I'm just going to show you the triangle table, and explain how you might build one and what you'd do with it once you had it. A triangle table is a data structure that remembers the particular plan you made, but keeps some more information about why those steps are in the plan. In some sense the plan graph from GraphPlan encodes that information, as does the graph that you get from using POP. But in the triangle table it makes very vividly clear an execution strategy for the plan. We're going to make a plan in the ordinary way, but then we're going to develop an execution strategy that is a little bit more flexible and robust than simply emitting the actions in order.

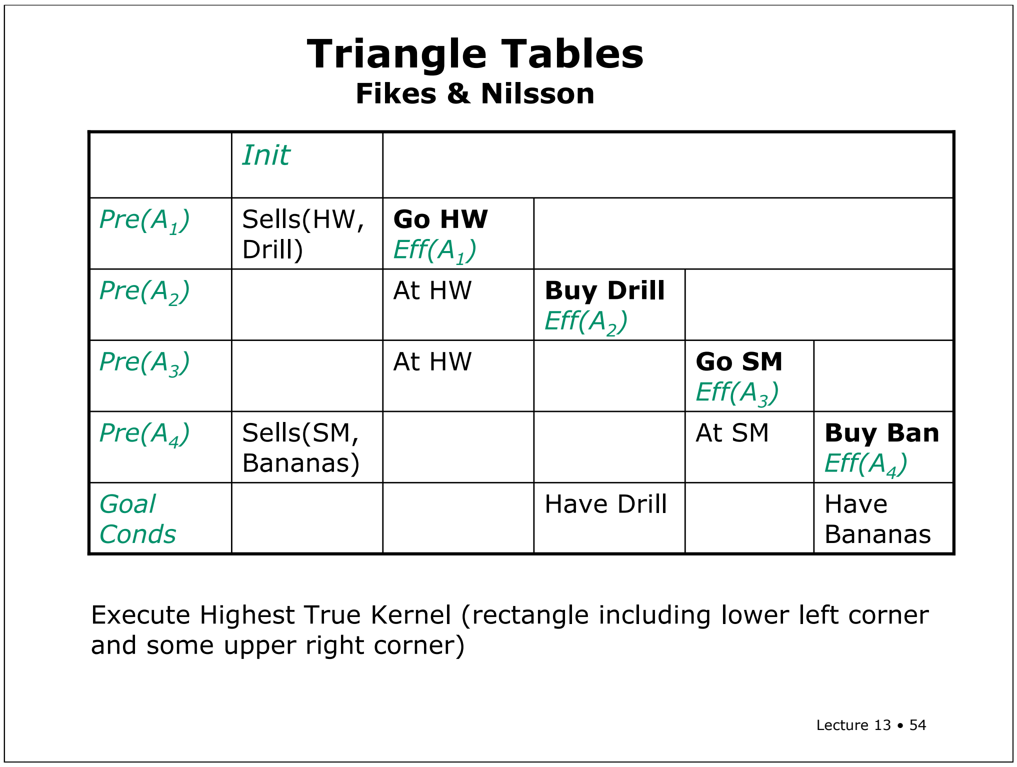


Triangle Tables
Fikes & Nilsson

	Init				
Pre(A_1)	Sells(HW, Drill)	**Go HW** *Eff(A_1)*			
Pre(A_2)		At HW	**Buy Drill** *Eff(A_2)*		
Pre(A_3)		At HW		**Go SM** *Eff(A_3)*	
Pre(A_4)	Sells(SM, Bananas)			At SM	**Buy Ban** *Eff(A_4)*
Goal Conds			Have Drill		Have Bananas

A triangle table is a big table. It’s really the diagonal and lower triangle of a matrix. Each row corresponds to one of the actions in the plan; the action is written at the end of its row. Each row contains the preconditions of the action. Each column contains the effects of the action above it.

So, for example, the At supermarket condition is a precondition of buy bananas, and an effect of go supermarket; so it’s in the column beneath go supermarket, and in the row associated with buy bananas. This is just another way of describing the information in a plan graph. The first column contains the initial conditions. And the bottom row contains the goal conditions.



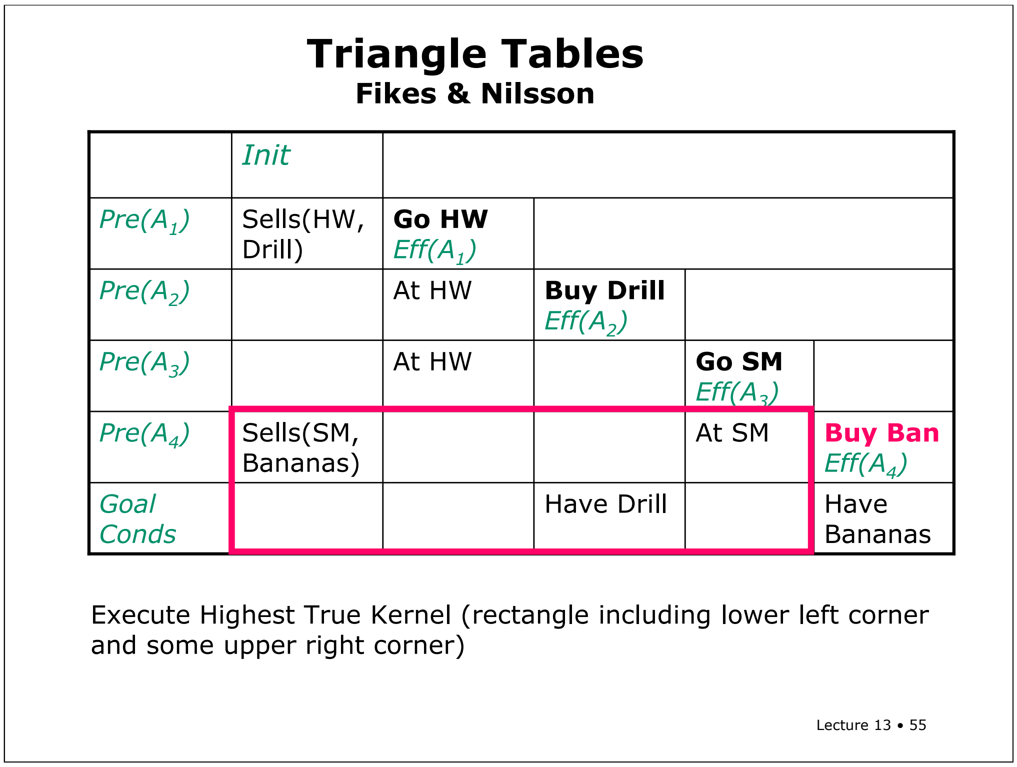
Triangle Tables

Fikes & Nilsson

	Init				
Pre(A_1)	Sells(HW, Drill)	**Go HW** *Eff(A_1)*			
Pre(A_2)		At HW	**Buy Drill** *Eff(A_2)*		
Pre(A_3)		At HW		**Go SM** *Eff(A_3)*	
Pre(A_4)	Sells(SM, Bananas)			At SM	**Buy Ban** *Eff(A_4)*
Goal Conds			Have Drill		Have Bananas

Execute Highest True Kernel (rectangle including lower left corner and some upper right corner)

Now, the rule for executing a triangle table is that you should execute the highest true kernel. A kernel is a rectangle that includes the lower left corner of the table and some upper right corner (not including an action).



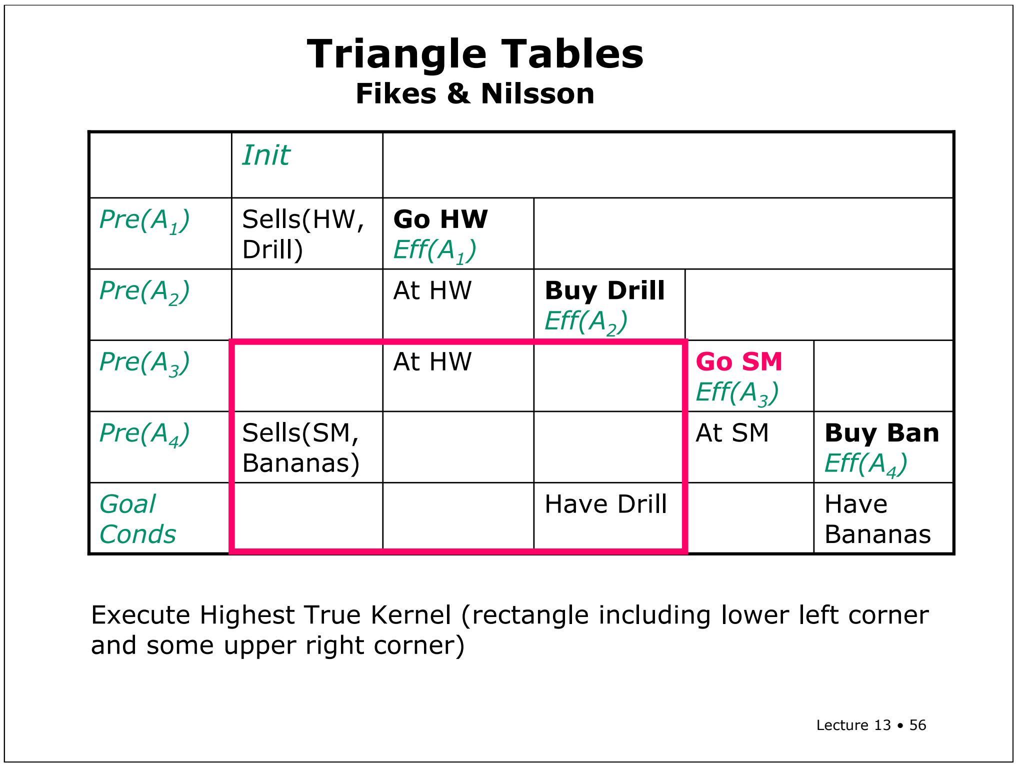
Triangle Tables

Fikes & Nilsson

	Init				
Pre(A_1)	Sells(HW, Drill)	**Go HW** *Eff(A_1)*			
Pre(A_2)		At HW	**Buy Drill** *Eff(A_2)*		
Pre(A_3)		At HW		**Go SM** *Eff(A_3)*	
Pre(A_4)	Sells(SM, Bananas)			At SM	**Buy Ban** *Eff(A_4)*
Goal Conds			Have Drill		Have Bananas

Execute Highest True Kernel (rectangle including lower left corner and some upper right corner)

Here is the highest kernel. It includes all of the preconditions of the last action, as well as the other conditions that we're depending on being maintained at this point (like have drill). The idea is that if we somehow find ourselves in a situation in which these three conditions are true, then we should execute the buy bananas action, no matter what sequence of actions we've done before.



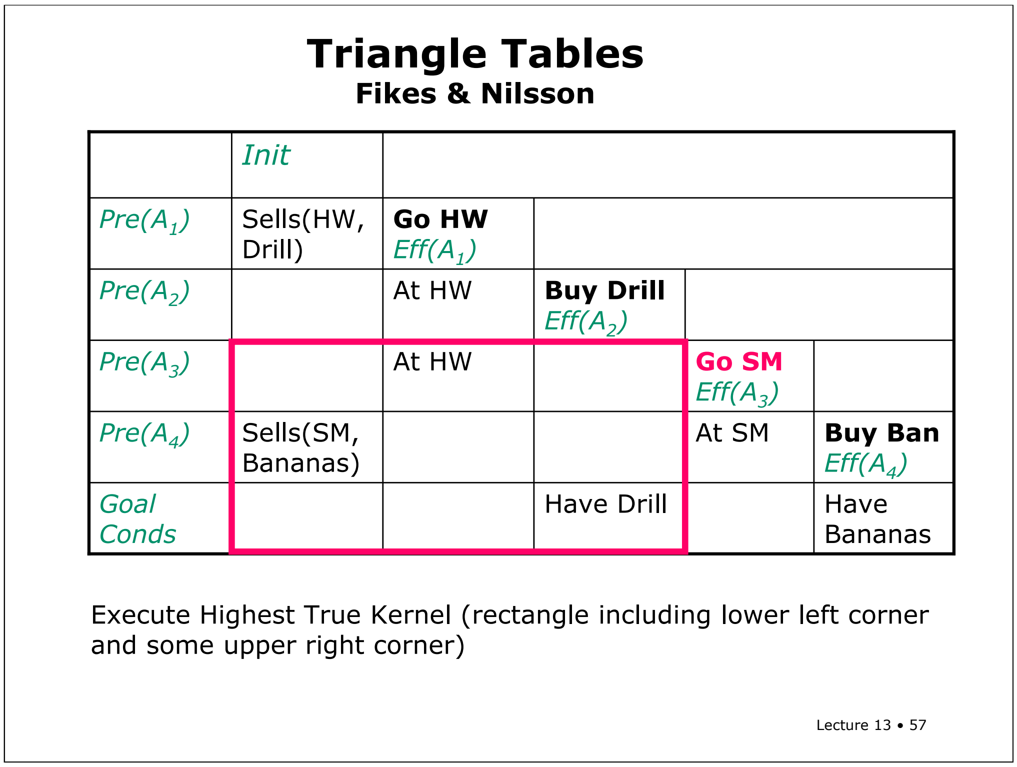
Triangle Tables

Fikes & Nilsson

	Init				
Pre(A_1)	Sells(HW, Drill)	**Go HW** *Eff(A_1)*			
Pre(A_2)		At HW	**Buy Drill** *Eff(A_2)*		
Pre(A_3)		At HW		**Go SM** *Eff(A_3)*	
Pre(A_4)	Sells(SM, Bananas)			At SM	**Buy Ban** *Eff(A_4)*
Goal Conds			Have Drill		Have Bananas

Execute Highest True Kernel (rectangle including lower left corner and some upper right corner)

If those conditions are not all satisfied, then we look for another true kernel. Here is the next highest one.

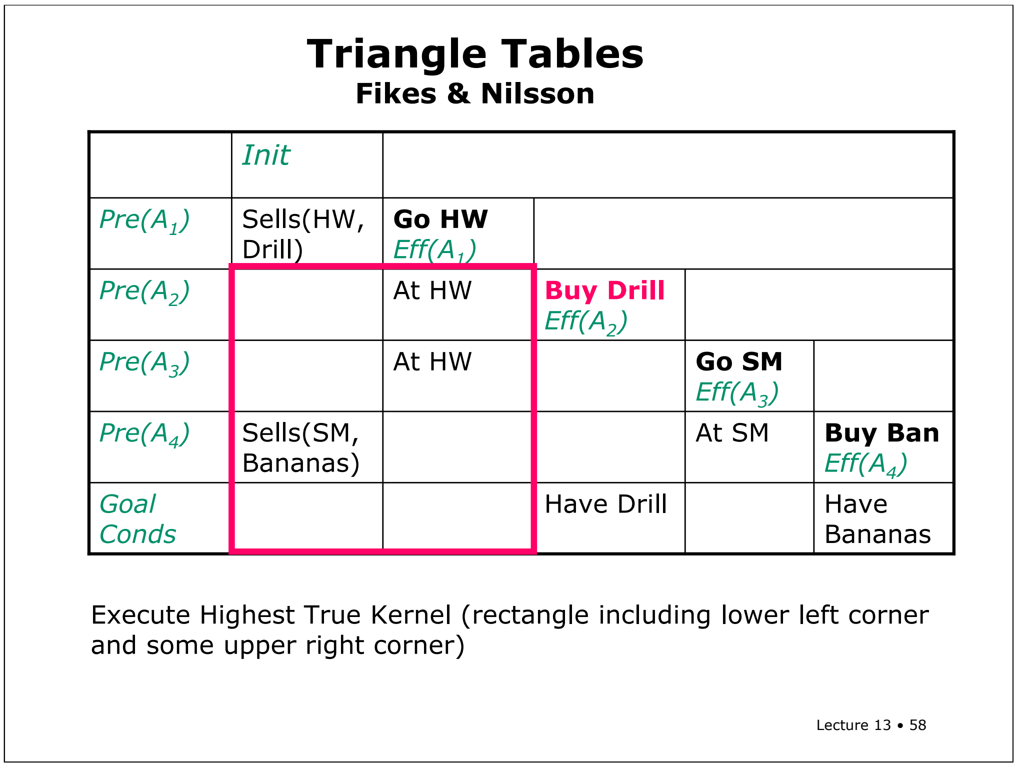


Triangle Tables
Fikes & Nilsson

	Init				
Pre(A_1)	Sells(HW, Drill)	**Go HW** *Eff(A_1)*			
Pre(A_2)		At HW	**Buy Drill** *Eff(A_2)*		
Pre(A_3)		At HW		**Go SM** *Eff(A_3)*	
Pre(A_4)	Sells(SM, Bananas)			At SM	**Buy Ban** *Eff(A_4)*
Goal Conds			Have Drill		Have Bananas

Execute Highest True Kernel (rectangle including lower left corner and some upper right corner)

If we have the drill but we're not at the supermarket, then we should go to the supermarket.



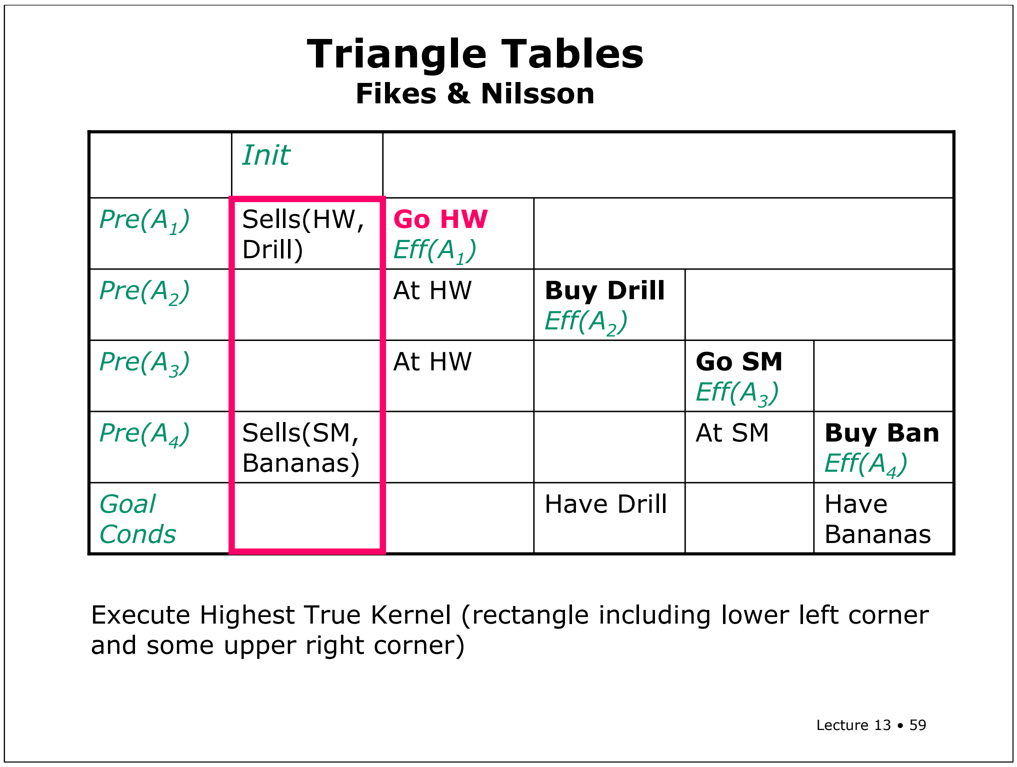
Triangle Tables

Fikes & Nilsson

	Init				
Pre(A_1)	Sells(HW, Drill)	**Go HW** *Eff(A_1)*			
Pre(A_2)		At HW	**Buy Drill** *Eff(A_2)*		
Pre(A_3)		At HW		**Go SM** *Eff(A_3)*	
Pre(A_4)	Sells(SM, Bananas)			At SM	**Buy Ban** *Eff(A_4)*
Goal Conds			Have Drill		Have Bananas

Execute Highest True Kernel (rectangle including lower left corner and some upper right corner)

If we don't have the drill yet, either, but we're at the hardware store, we should buy the drill.



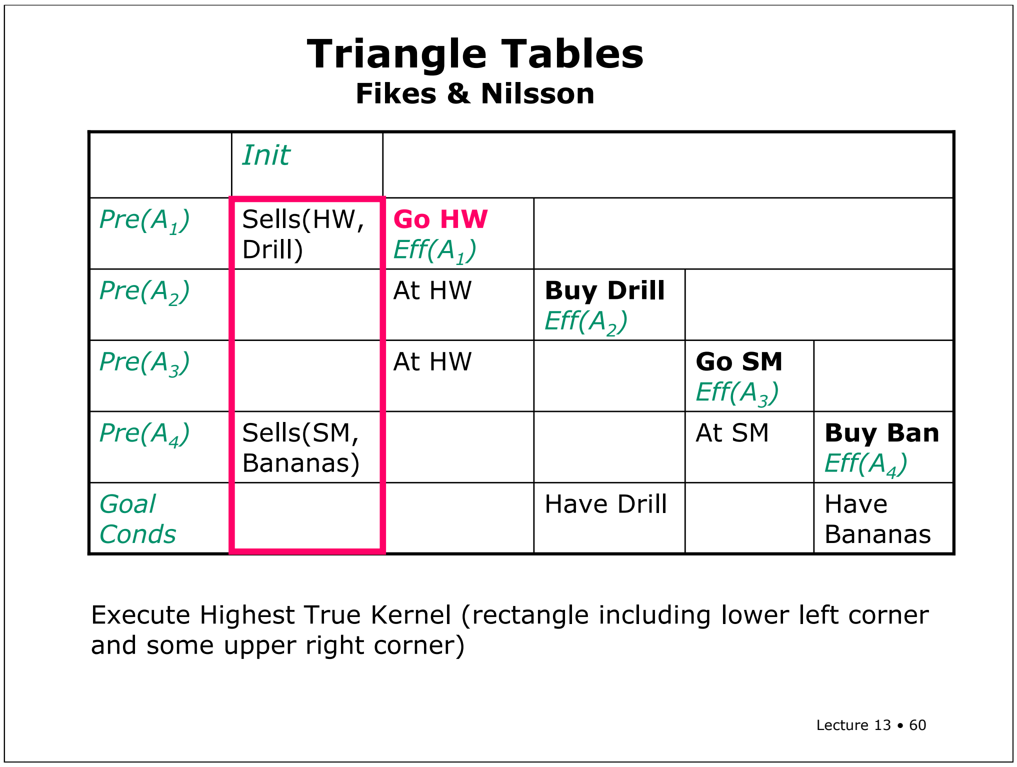
Triangle Tables

Fikes & Nilsson

	Init				
Pre(A_1)	Sells(HW, Drill)	**Go HW** *Eff(A_1)*			
Pre(A_2)		At HW	**Buy Drill** *Eff(A_2)*		
Pre(A_3)		At HW		**Go SM** *Eff(A_3)*	
Pre(A_4)	Sells(SM, Bananas)			At SM	**Buy Ban** *Eff(A_4)*
Goal Conds			Have Drill		Have Bananas

Execute Highest True Kernel (rectangle including lower left corner and some upper right corner)

Failing that, as long as the initial conditions are true, then we should go to the hardware store. If somehow even the initial conditions have become false, then the execution of the triangle table fails and the planner is called to re-plan.



Triangle Tables

Fikes & Nilsson

	Init				
Pre(A_1)	Sells(HW, Drill)	**Go HW** *Eff(A_1)*			
Pre(A_2)		At HW	**Buy Drill** *Eff(A_2)*		
Pre(A_3)		At HW		**Go SM** *Eff(A_3)*	
Pre(A_4)	Sells(SM, Bananas)			At SM	**Buy Ban** *Eff(A_4)*
Goal Conds			Have Drill		Have Bananas

Execute Highest True Kernel (rectangle including lower left corner and some upper right corner)

Thus, we get fairly robust execution of a plan, possibly repeating a step that didn't work, or skipping one that is serendipitously accomplished for us. But, we don't plan for every eventuality, and if things really go badly, we stop and replan.

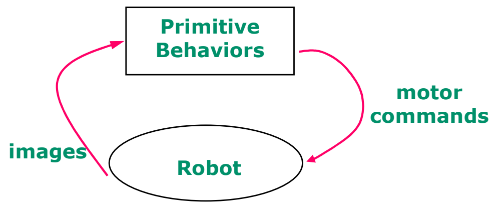
Hybrid Architectures

- Reactive lower level
- Deliberative higher level

Ultimately, in most systems, you want some combination of fast, “reactive” programs in the lowest layers with flexible “deliberative” systems in the higher layers.

Hybrid Architectures

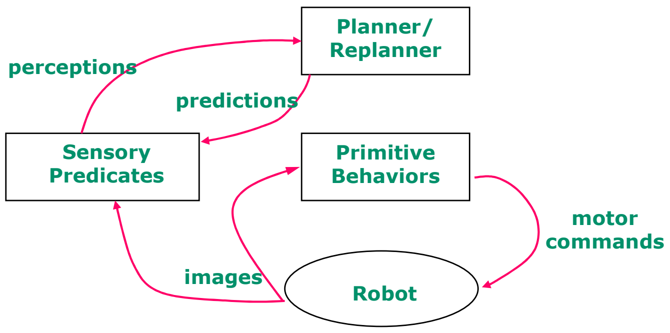
- Reactive lower level
- Deliberative higher level



A "reactive" program might be a universal plan, or a servo-loop that drives a mobile robot down a hallway. It typically has a quick cycle time, and it never really stops acting in order to think.

Hybrid Architectures

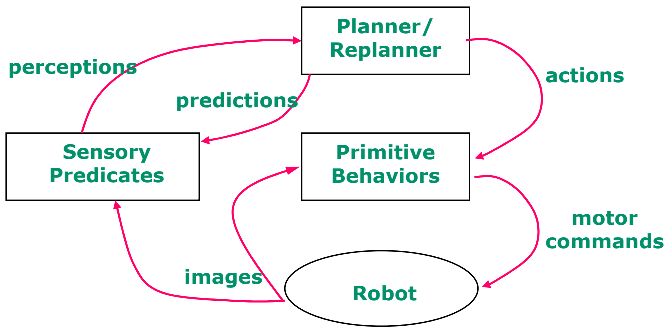
- Reactive lower level
- Deliberative higher level



In parallel with the reactive primitives, you might have a planning/replanning system that takes as its atomic actions things like driving across the room, which actually turn out to be pretty complex procedures from the perspective of the reactive layer.

Hybrid Architectures

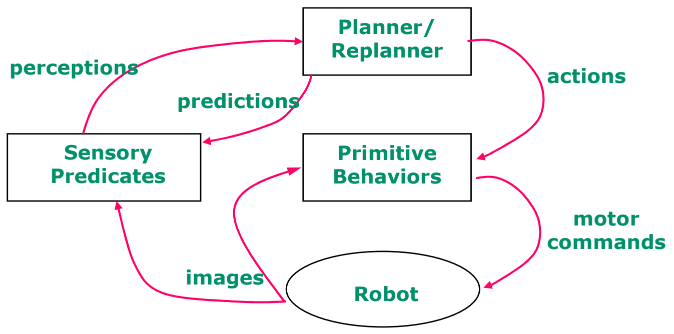
- Reactive lower level
- Deliberative higher level



The planner makes a plan, and feeds the actions, one by one, into the reactive layer, which executes the actions. Simultaneously, the world is monitored to see what effects are actually happening in the world. Often the planner can predict what ought to be happening in the world, which can make sensory processing easier.

Hybrid Architectures

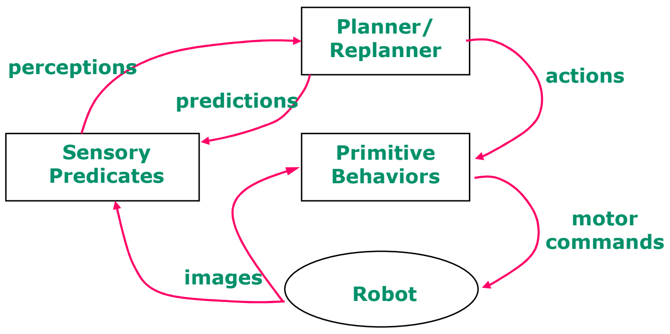
- Reactive lower level
- Deliberative higher level



If the plan-monitoring system detects that the plan is not having the expected effects, it replans and continues. An advantage of having a reactive lower level continuing in parallel with the planning and replanning is that , even when the “higher” brain is occupied with figuring out what to do next at the high level of abstraction, the lower layer is there to execute automatic reflex reactions; to keep the robot from running into things or the creature from being eaten.

Hybrid Architectures

- Reactive lower level
- Deliberative higher level



We don’t have any concrete recitation problems for this lecture, or the previous one.

Next time, we’ll start in on probability, and we’ll have a lot of exercises to do.