Artificial Intelligence: Constraint Satisfaction Problems
Module 7: Applications
Lecture 6: Planning as Satisfiability
Professor: Deepak Khemani
Department of Computer Science and Engineering, IIT Madras

Keywords: planning as satisfiability, SATPlan

Okay so in the last couple of classes we have seen how planning can be modeled as a CSP, you can convert a planning problem into a constraint satisfaction problem. The whole idea of doing this sort of a thing is that once you transform this problem into a well known class of problems then you can use off the shelf solvers for actually solving it. So your task is only to convert it into a CSP. We can also do the same for SAT. We can convert it into a SAT problem and that is in fact what was done by Kautz and Selman and they wrote a planner called SATPlan, as the name suggests, it converts the planning problem into a SAT problem. It is a hugely successful implementation in the planning competition. So this was done in 1992 or so. I think in 98 or something they won the International Planning Competition, which is a competition in which you have to write your domain independent planner and then they will give you domains and problems and then your planner should run on that and then they compare, which one finds better plans, which one solves bigger problems and things like that. Because planning is a hard problem and typically as the problem size grows, which means the number of blocks for example in blocksworld grows, it becomes harder and harder to solve this. Now, encoding planning as a SAT problem is similar to encoding it to a CSP and we look at a couple of ways to do that essentially. So instead of constraints, the problem is expressed as a formula in propositional logic, which is satisfiability. So essentially all the variables have values which are true or false and all the constraints are logical connectives and look for a satisfying assignment to the propositional variables. So the variables for example on(A,B) are called fluents which is also a standard term in AI nowadays, and by fluents we mean predicates that can change their value over time, and typically we use symbols like f_1, f_2, and f_3 and so on. The basic idea is called linear encoding that we want, we find linear plans into satisfiabilty is to add a time parameter to the fluents and express the relation between actions, preconditions, and effects essentially. Something similar to what we did in CSP. We had this $state_0$, $state_1$, and so on, here we are talking about it as time and then we are still trying to capture the relationship between actions and their preconditions, the actions and their effects, and also the frame axioms. Only thing is the formulation is different. The formulation is going to be a SAT problem.

So you have to sort of put together this SAT problem piece by piece and let's look at all the different influences on this thing. So in the initial state s_0 if a fluent is true in that state or if belongs to the state s_0, then we add f as a clause, so we assume it is in some CNF-like form or we can convert it to a CNF-like form. If it is not true, then we add negation of f_0 as a clause essentially. So for example in the problem that we were looking at if A is on B and, so if it is a very simple one in which A is on B then clear(A) is true and ¬clear(B) is true. So may be we should have also stated where is B, but that is something which is missing here. So maybe I can add that here, is that onTable(B,t_0) or something like that. Then the influences from the goal clause assuming that at most k steps and this is the same theme, it happens in graph plan, it happens in CSP, it happens in planning and satisfiability is that you try to pose it as a problem of a given length that the plan is of a given length and then you try to solve it essentially, and we can sort of iteratively increase the length so some kind of iterative depending algorithm can work here. So, if we assume that the plan is of at most k steps, then we add a fluent f_k for every goal proposition that is being specified in the problem. Then like in planning with CSPs, an action implies its precondition, so if an action holds at time t, then its preconditions must hold at time $t - 1$ essentially. So, you can see that earlier we talked of it as a constraint that an action for example unstack(A,B) in the state variable description said that it must be, it is related to the fact that A is on B in the previous state, so we had a binary constraint there essentially. So here we are stating it as a as a formula in logic and we are saying that if unstack(A,B) is true at time t, then the following three things should have been true at time t - 1, which is that the preconditions of this action, which is that A must have been on B, the arm must have been empty and A must have been clear. There was nothing on top of A essentially. So the difference is now we are writing this as a formula in propositional logic. An action implies its effects which is similar. So for the unstack action we have that if you unstack A from B then not A B will be true at time t, the action is done at time t, the preconditions hold at time t - 1, and the effects hold at time t, then arm is no longer empty at time t. You are holding A t, and B has become clear. So all the effects you add and then you add the classical frame axioms, that if the action A does not affect a fluent f then f remains unchanged after the action. So this is the generic form of stating the constraints that a fluent at time t and an action A time t - 1 and action A implies the fluent is true at time t also. So a fluent which was true remains true, a fluent if it was not true, not true meaning the fluent was not of something then it will remain not of something. So for example, if C was clear at time t - 1 and you are unstacking A from B then C must be clear at time t essentially. You have to state this for every action and for every fluent essentially. So of course, the good thing is that you do not have to do it manually, you can write a program to do this, but essentially all these connections must be established for every action, for every fluent. The original formulation of SAT plan, which is a program written by Kautz and Selman also had clauses that said that only one action occurs at a time because you were looking for linear actions at that time and that can be expressed as the constraint of the kind that one of them must be false at least. I mean both can be false, but one of them at least must be false. It is not that one of them has to happen. So for every pair of actions we have this constraint.

Okay, we can also like we did in the case of CSP, we can take a planning graph and then convert that into a SAT problem essentially. So if you remember this is what a planning graph looks like. The planning graph is constructed in a forward fashion and here we have constructed two stages of the planning graph and so we have for example the NO OP operation which says that OnT(B) was true and OnT(B) is true in the, it was true in the initial state and it is true in the first layer and then it gets carried forward essentially. Because you know you are taking a union of all possible actions essentially. So that is one property of the planning graph that it grows monotonically. Once a variable enters a layer it stays there. So there are of course, of course things like mutex relations that we had talked about, we had said that two actions are mutex or two variables are mutex essentially. What we are seeing here as a dot is a is a negative effect or in the STRIPS domain we sometimes call it as a delete effect essentially. So everything that can be possible is captured in the planning graph. So in that sense a planning graph is already like a specification of a constraint satisfaction problem. It says that if for example, if for example hold(C) has to be true in the proposition layer 1 then it must have been picked up in the first action layer essentially and if you want to pick up C in the first action layer, then clear(C) must have been true to start with onTable(C) must have been true and arm must have been empty essentially. So these are the three conditions. So essentially you are stating the conditions. The conditions between pickup and its preconditions, pickup and its post conditions, and now we are simply saying that the planning graph has already constructed this, and let's just convert it into a SAT problem essentially. So assuming that this planning graph is available to us and typically what you would do is you would grow a planning graph up to a certain length and that length is at least that till all the goal propositions appear in a mutex free fashion. So for example, if the goal here is to do let us say C on A and on B, oh, that's very easy. I hope it does not go off, C on B on A, then you want on(C,B), I don't have here, and on(B,A), I don't have here, because I have not drawn the full proposition layer, so on(C,B) should be here on(B,A) must be here and somewhere onTable(A). Okay, I don't even have that. onTable(A) must be true, and all of them must be non mutex. There must be no mutex relation between them essentially. It blows the planning graph at least till that level then you convert it into a SAT problem, which we will see in a moment, then try to solve it, if you cannot solve it then extend it by one more layer and then try to solve it, and there is a termination criteria, which will not go into, which will happen if the problem does not have a solution essentially.

So how do we encode this into a SAT problem. From the initial layer, this is like the straightforward application, the only difference is that we are saying that in the proposition layer zero, which is the initial layer, then you add f_0 as a clause, else you add negation of f_0 as a clause, which is what we did earlier also. Likewise for the goal state, we add a fluent f_k for every goal proposition that is in the goal state essentially. An action implies its precondition which is also the same as what we did a short while ago when we just directly formulated it as a SAT problem. If our action holds at true, so remember this is a SAT problem. You are saying that action happening at time t is a true statement at some time t then its preconditions must hold at time $t - 1$ and this is expressed as a clause of this kind that action implies its preconditions at time $t - 1$ and where this is conjunction of all these preconditions, so again the same example that we saw earlier essentially. If you are unstacking A from B, then A must have been on B at time $t - 1$, the arm must have been empty and A must have been clear, essentially. The effects are handled a little bit differently here. So what we say here is that for every fact in a proposition layer in some level t, a disjunction of the actions that could have created that fact is implied. So essentially you are saying that if this fact has come in the proposition layer, and remember that the proposition layer is constructed in a forward fashion is that whichever actions are applicable, you add their effects to the next proposition layer and you keep doing this essentially, and because now we are working with subset of all possible actions, we are only considering those actions which have appeared in the action layers. What we are saying is that if a proposition has appeared at level t then one of the, some action in the previous layer must have achieved it. So for example, if you are holding A at level t, then you must have either picked up A or you must have unstacked A from B for example or unstacked A from C or you would have done a no-op, which is the case that in the previous layer itself you are holding A essentially. So depending on what is true essentially in the previous layer, these are the actions, which may exist in the planning graph here essentially. So this implies that in the previous layer A was on the table and then pickup(A) action was applied. In the previous layer also A was on B, remember that you can have multiple things happening. There would be mutex relations between onTable(A) and on(A,B), both can be not true at the same time. So mutexes play a big role again. But one of the actions, or all actions in the previous layer, which could have achieved it at least one of them is implied essentially and then a mutex will take care of the fact that exactly one of them is implied that everything cannot happen. That is what happens in the backward phase of the planning graph. Of all the actions that could have achieved holding(A), in the final plan only one of them will exist and then you have to choose the one which is consistent with the rest of the planning graph and that is where this idea of constraints comes into play essentially. So this is determined by the planning graph. This is not a generic statement. The actions that are mutex can directly define clauses in the SAT and likewise for propositions you simply say that one of them must be false essentially that you can either pickup C or you can unstack A from B. You cannot make both true at the same time, at least one of them must be false and these are the binary mutexes that we were talking about essentially here. Okay, so I think with this we have seen two ways of converting a planning problem into a SAT problem and I do not think I have more details on this.

So we will leave it here and with this we will also end with this planning application to this constraint satisfaction course. We are seeing planning as an application of constraint satisfaction. Of course, planning needs a full course in itself and may be at some point we will have a course on planning as well. Okay, so we will stop here. In the next class, I will just wind up what we have studied in constraint satisfaction and I will try to also highlight things that we have not done and which we will not be doing in this course, but which would be part of constraint satisfaction. So there are certain things that we have not looked at. I will just try to point out list those things and then we will end there essentially.