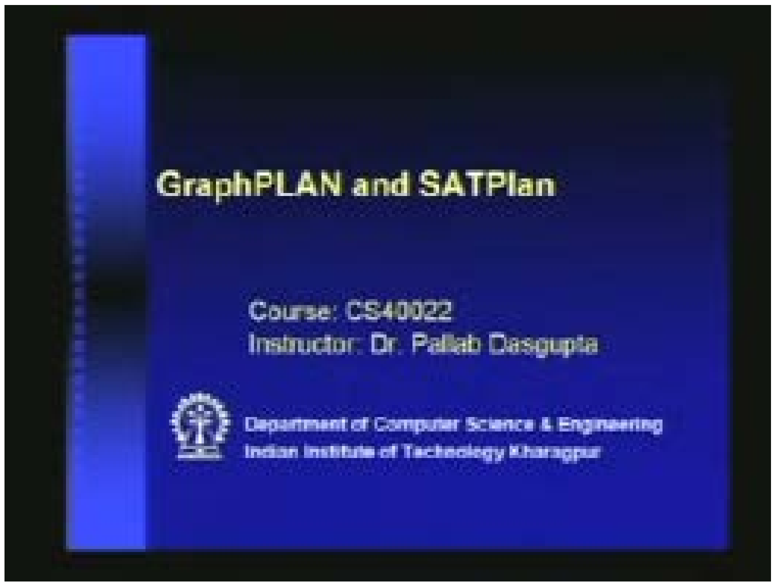
Artificial Intelligence
Prof. P. Dasgupta
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture No - 19
Graphplan and SAT plan

In the last lecture, we had seen the algorithm partial order planning- the POP algorithm. Today, we will study 2 other algorithms, namely graph plan and SAT plan, which have evolved much more recently. They have evolved in the 90s and they were actually brought about because planning was being used in a wide variety of practical domains.

(Refer Slide Time: 01:20)



GraphPLAN and SATPlan

Course: CS40022
Instructor: Dr. Pallab Dasgupta

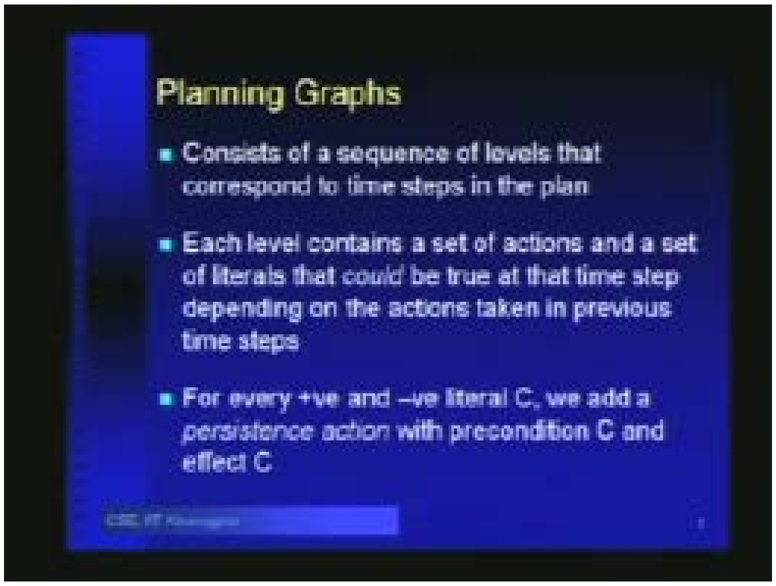
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

For example, 1 of the major breakthroughs in planning was when a complete satellite launch by NASA was done with a help of a planner. Why do we need the planner? Because during the actual launching, things can go wrong. Things will never go exactly as you want and so immediately, you need an alternative plan. You need to know what is the alternative sequence of actions that has to be taken, so that your launch vehicle comes back into the correct path. That is why and also in a wide variety of other kinds of control applications, planning was being used and people found that for problem domains which were NP complete or harder, it was being very difficult to use the partial order planning algorithm.

That is why new class of algorithms were born and these class of algorithms made use of the fact that the more recent computers that we had, had more amount of space and had more computation power. So, we could have algorithms which were infeasible 20 years back, but are perfectly feasible today. I will talk about 2 such algorithms in this lecture. To start with, we will look at the graph plan algorithm. Firstly, let us see what is a planning graph. A planning graph will consist of a sequence of levels that correspond to time steps in the planning. I will divide the planning into sequence of time steps.

Each time step or each level contains a set of actions and a set of literals that could be true at that time step, depending on the actions taken in previous time steps. It is like unfolding the possible set of actions and the set of initial facts that we have: describe a search problem, so we can unfold it in to a state spaced graph. This is the state space graph which is levelized depending on the distance from the start state. But we will see that it is not so simple, because there will be conflicting set of events in the graph. For example, if you take this action, then you will not be able to take another action, so, if you have achieved this, then you cannot achieve that at the same time. We will see things like that.

(Refer Slide Time: 04:22)



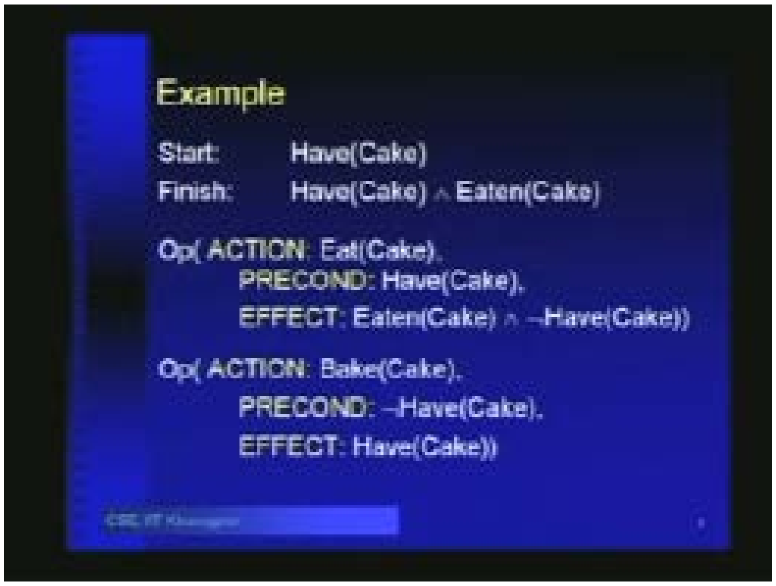
Planning Graphs

- Consists of a sequence of levels that correspond to time steps in the plan
- Each level contains a set of actions and a set of literals that *could* be true at that time step depending on the actions taken in previous time steps
- For every +ve and –ve literal C, we add a *persistence action* with precondition C and effect C

And then, for every positive and negative literal C, we add a persistence action with pre-condition C and effect C. I will shortly explain why we need this. This is the no action action- the action which describes that there is no change in C, in the truth of C. So, whatever we had as the truth of C in time step k, will be the same as the truth of C in time step k plus 1. In order to achieve that, we need some action which will tell us that action is this persistence action. It just retains the state of that literal up to the next cycle.

And we need to define that, because when we say that whichever things that we are saying will change and others will remain same. In a logical framework, that connotation has to be formally expressed. That is why we need this persistence action. Let us look at this example, that the start- so, this is the famous question about whether you can have the cake and eat it too.

(Refer Slide Time: 05:18)



Example

Start: Have(Cake)

Finish: Have(Cake) $\wedge$ Eaten(Cake)

Op(ACTION: Eat(Cake),
 PRECOND: Have(Cake),
 EFFECT: Eaten(Cake) $\wedge$ $\neg$Have(Cake))

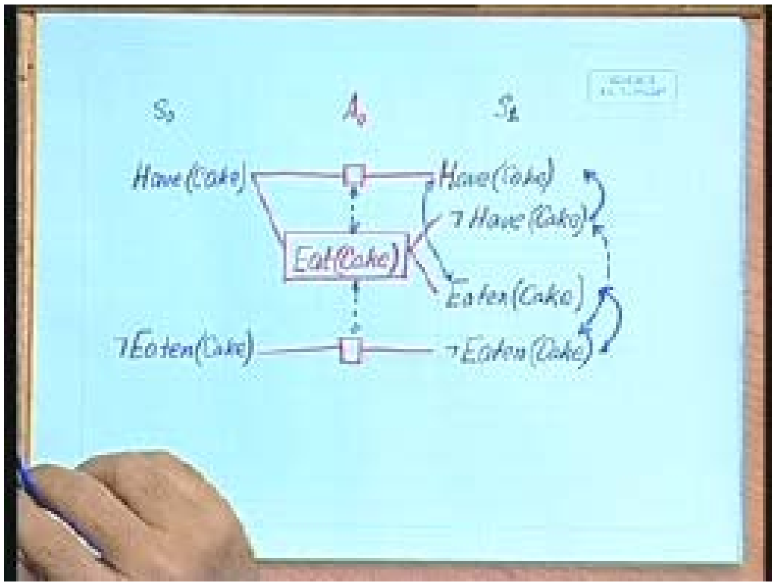
Op(ACTION: Bake(Cake),
 PRECOND: $\neg$Have(Cake),
 EFFECT: Have(Cake))

The start is that we have cake and in the finish, we have have cake and eaten cake, so we have eaten it and we still have the cake. Obviously, we request some actions- the first action is eat cake, so I have written down the actions in the STRIPS format. The action name is eat cake, the pre-condition is have cake, effect is eaten cake and not have cake. And the other action is bake cake, which has a pre-condition not have cake and the effect is have cake. Now, let us see how a planning graph for this is going to look like. We will start with the step S0. Initially, we are in the step S0, so, in this initial step, what we have are the following:

We have have and we have- right. What are the actions that we can have here? Let me enumerate the actions. 1 is the persistence of this, which means that I will just retain this. If I retain this, then I will have- so, this is a persistence action, which says that just let us keep the cake for the next time step, right? An alternative action is eat cake. See, at this, in this time step, bake cake is not a possible action, because bake cake has the pre-condition not have cake, but since we have cake, so, we cannot use that. If we apply the action eat cake, then, this is going to produce 2 effects. They will have not have cake and we have eaten cake.

The third action is the persistence of not eaten cake. If we use that action, then, we will have not of eaten cake. So, this is what we have in the step S1. This describes all possible things that we could have in the step S1 in the next time step. But note that we cannot have all of them, because some of them are conflicting with each other.

(Refer Slide Time: 11:59)



Let me note down what conflicts with each other. have cake and not have cake, obviously they conflict, so I cannot have the 2 of them. Similarly, eaten cake and not eaten cake cannot be there together. There is also the case that have cake and eaten cake have a conflict. Why do they conflict? (Students speaking). Let us see. These 2 actions- this persistence of this and this; these 2 actions are mutually exclusive actions. Why? Because 1 produces have cake as the result, the other produces not have cake as the result. This is 1 reason why these 2 are exclusive. They are also exclusive for another reason; the reason is that the effect of eat cake negates the pre-condition of this action.

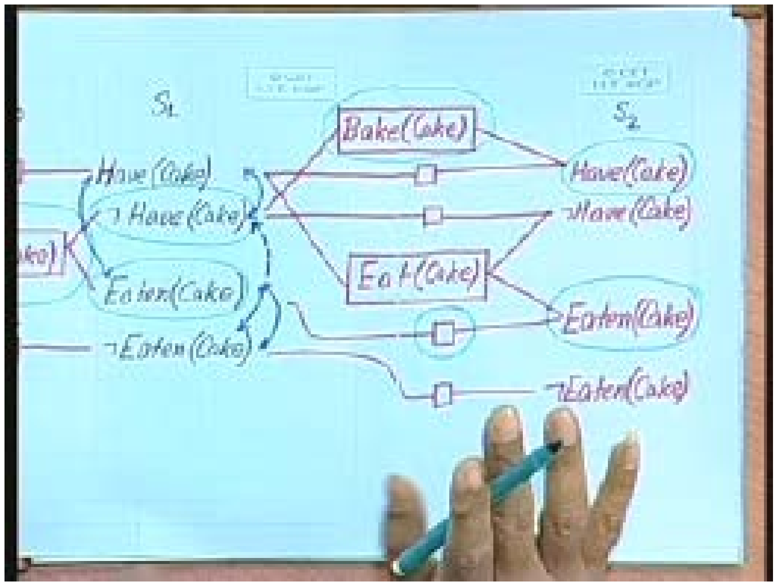
See, this is going to produce not have cake and then this cannot be applied, because you do not have cake anymore. Remember that in partial order planning also, we had this problem, that is why we were promoting or demoting in it. But here we want that these 2 actions both to be applicable in the same time step, which is not possible, because we have to either demote or promote 1 of them. They cannot be serialized in any order. If we put eat cake before the persistence of this, then, the eat cake will produce not have cake and then this action cannot be executed anymore. So, the ordering between these is important; we cannot have them in the same time step.

That is another reason why there is a conflict between these 2. Also, we have a conflict between not have cake and not eaten cake and the reason is that this is producing again, for the same reason, this is producing eaten cake and this is producing not eaten cake. And also, this is producing eaten cake, which negates this. So, in the planning graph, what we will do is, we will maintain the links between the actions, indicating that which pairs of actions are incompatible, which are exclusive and also we will maintain links between the propositions that are created as the effect of these actions.

Some of them can be mutually exclusive, like the ones that we have drawn here. So, we will maintain all these links in the planning graph. Let us look at 1 more level of this- what happens after this? We have have cake, not have cake, eaten cake and this. Let us apply 1 more round of the actions. Now, I can also use the bake cake option. Let us say that I try bake cake. The pre-condition of bake cake is not have cake, so, not have cake and the effect of this is to have cake. And the persistence actions of have cake is also going to produce the same result; the persistence action of not have cake will produce not have cake; then, we can use eat cake here as another action.

What we are doing is, we are applying all possible actions from 1 level to another. This action can also be used, because I have cake, so, I can eat cake and the effect of that is not have cake and eaten cake. And the persistence action of eaten cake is also going to produce this, and then we have not eaten cake and- (Students speaking). Yes. (Students speaking). It does not do anything, but it tells you that you can- in the next step also- you can have cake. See, the use of- suppose you did not use this persistence action in this level; suppose we did not use the persistence actions. Then, the only action that you can apply is eat cake. So, in the second time step, you will not have cake.

(Refer Slide Time: 20:54)



But you may also choose to have cake in the next time step. Not clear? Okay. (Students speaking). No, what we are trying to figure out is that in a given time step, what is the set of propositions that can be true? What kind of- what is the set of propositions that can be true, okay? Now, after time step 1, we found that we have got a bunch of propositions, but there are- mutex is between them. If I have this, I cannot have that. So, this is a set of propositions which have certain links between them, indicating that which pairs are compatible and which pairs are incompatible. Now, I have a goal in mind. In this case, my goal is have cake and eaten cake, but after the first step, you see I have have cake, I have eaten cake, but there is a mutex between them.

So, my planning problem is not yet solved. My planning problem is not yet solved, so, I apply another step of this. On this set, I again apply the set of actions and get a new set of propositions, which is what I get after the at time step S2, right? What we try to check there is that, do I now have have cake and eaten cake without a mutex? Now, the point is that as you progress through the steps of this algorithm of building the graph, the number of mutexes is going to decrease as you go ahead. Why is that so? Because we are finding out alternative ways of achieving that proposition. In the first time step, I could not produce have cake and eaten cake at the same time, but then after baking another cake, so, if I eat the cake in the first time step and then bake a cake in the next time step, then, I will have have cake and eaten cake.

Now, to answer your question, that why do we need the persistence action- we need the persistence action to maintain that I have already eaten a cake, otherwise, your eaten cake will not appear here and then, you will not be able to have the cake and have eaten the cake. Understood? If we do not have the persistence of the eaten cake, then, in every alternate step, you will eat the cake, then, do not have a cake, so you will bake one. You will again have the cake, but you still do not have eaten cake, because if you do not have the persistence, so, you will have it like this. Then, again, he will eat cake in the next cycle; you will have eaten cake, but you will have not have cake, right? Could I make myself clear?

You have to maintain also the persistence actions for those propositions, which do not change into the next time step. Now, let us see. Yes- (Students speaking). We are not applying 2 actions at a time; we are just enumerating all possible actions that I could have taken in that time step. I am just enumerating all those and I am clubbing them together, but I am also maintaining these mutex links, which tell me that which pairs are compatible and which pairs are incompatible. So, the planning problem is not solved by creating this graph, but the point is that when I finally have have cake and eaten cake in a non-mutex way, then most probably, there is a subset of this graph which is the plan. And indeed, if you look at this case, then, what is that subset of the graph?

Let us start from here- the subset of the graph which achieves the desired plan is to first eat cake. Then you will have not have cake and you will have eaten cake. Then, you can bake cake, bake the cake and then you have have cake, because of the bake cake and because of the persistence of not eaten cake or eaten cake. So, this 1 in blue represents the actual plan that I want, so, just unfolding the planning graph is not sufficient. But 1 thing I am certain is that as long as I do not have the goal propositions in a non-mutex way, I know that my plan is not yet complete. Whatever I have seen so far in the plan graph does not contain a plan, right? The moment I find that they are appearing in a non-mutex way, then, I know that I possibly have a plan, but even then, I may not yet have a plan. I may still have to unfold it for a few more cycles.

Now- (Students speaking). No. Okay, this is an interesting thing- just take this 1 as an exercise, that after we do all this, I explain the algorithm, etc., in details. But whenever we have the set of goal propositions in a non-mutex fashion, it is a necessary condition to have the plan, but it is not a sufficient condition to have the plan. Now, try to get this into your mind and try to figure out a counter example which establishes this result. Could I make the problem clear? The problem is that I start with the initial set of propositions and the set of actions and then progressively, at each time step, I am taking my entire set of propositions and applying the set of actions to get the set of propositions for the next time step.

In each time step, I am maintaining the mutex links between the propositions and then, I am continuing this until I reach a point where the set of goal propositions appear in a non-mutex link, so, there is no mutex links between the set of goal proposition. At that point of time, there is a possibility of having a plan in the set of steps that have unfolded so far, but it is not sufficient, so, try to find an example where it is not sufficient. But there are some other issues which we also have to understand. We have to understand that is this graph- the planning graph that we are describing like this- is it finite or could be for ever keep on unfolding, right?

Now, note 1 thing, that if we have a set of propositions as non-mutex in any time step, then, they will remain non-mutex for every time step in the future. Why? Because the individual persistence of those propositions will carry them over into the next time steps. So, at if at time step k, I am able to achieve 2 propositions without a mutex, then, in all time steps greater than k, I will be able to achieve them without a mutex. And that I can do it by just carrying them over through the persistence actions into the next time steps. How the mutex is going to change- so, anything which we have without a mutex will remain without a mutex forever. But those which currently are with a mutex, can become independent in future time steps. What we can say is that these mutex links can only disappear as we go ahead. Now, since we are clubbing all the propositions that we could possibly achieve in a time step, so, suppose we end up in a scenario where in 2 successive time steps, we have the same set of propositions and the same set of mutex links.

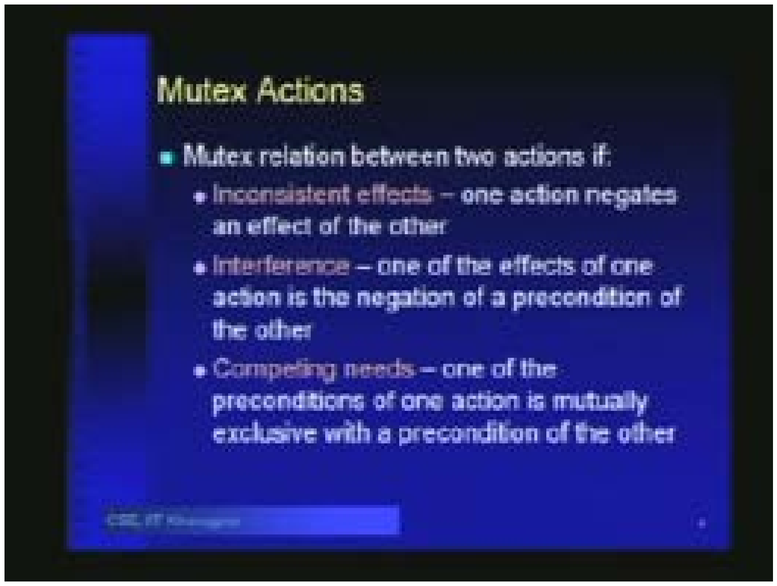
Then, we can say that any future steps will produce the same thing, so, I will have reached what we call a fix point. A fix point is the core thing here and we have to realize that a fix point will always exist for this and the reason why it will always exist is that these mutexes can only disappear. The set of propositions that we have here is final, so, the 2 things that can change over time is new actions can become applicable, new propositions can generated and mutex links can disappear. But each of these will taper off at the end, because the set of propositions is finite, so you cannot keep on adding propositions forever. And see that a proposition only appears once in this thing because we are maintaining the links between them.

A proposition does not appear multiple number of times in a single time step, so, we have 1 have cake, 1 not have cake, 1 eaten cake and not eaten cake. But they can be achieved in different ways, and the mutex links will also taper off in the end; they are going to reduce reduce reduce, and then, after some time, they will not reduce any more. So, that fix point establishes that this planning graph will have finite depth. Now, when we reach the end of the planning graph and if we find that we still do not have the set of goal propositions in a non-mutex way, then, we can say that the plan does not exist; then, we can say that the plan does not exist, otherwise, a plan may exist and we have to find it out. Now, how do we find out whether there is a valid plan? We have to find out whether we are able to achieve this at each step through a set of non-mutex actions.

One option is that you unfold the graph right up to the fix point and then check whether, in the fix point, you are able to find all the goal propositions in a non-mutex, their all independent, they achieved.

If that happens, then, you start walking backwards from there, trying to pick up actions which are non-mutex in nature, and go back right up to the start state to trace out the plan. We will see more examples to understand this.

(Refer Slide Time: 30:54)



Mutex Actions

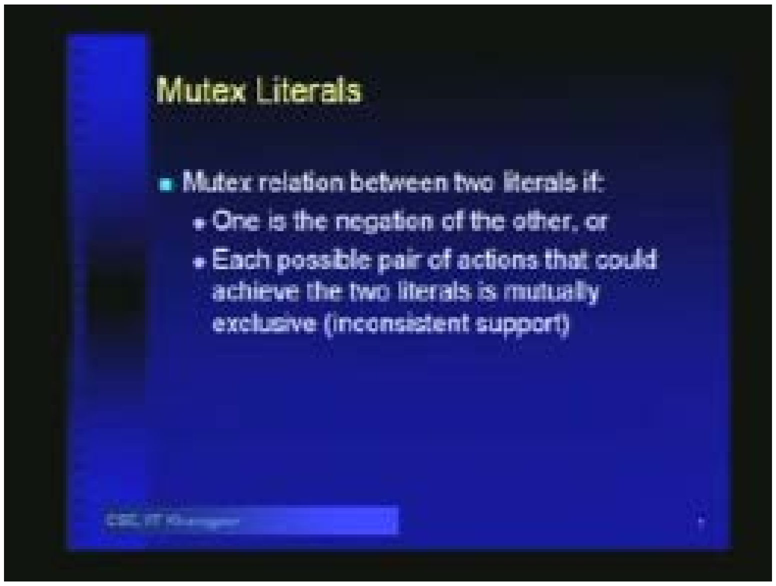
- Mutex relation between two actions if:
  - Inconsistent effects – one action negates an effect of the other
  - Interference – one of the effects of one action is the negation of a precondition of the other
  - Competing needs – one of the preconditions of one action is mutually exclusive with a precondition of the other

Once we trace out the plan, then, we say that this is the plan. Fine, but we need not actually unfold the graph right up to the end, because the set of goal propositions may have appeared much earlier than the entire depth of the planning graph. We can do a kind of iterative deepening and that is exactly what graph plan will do. Let us go forward and take a look at this. To summarize what we have discussed so far- mutex relation exist between 2 actions, if 1 action negates the effect of the other, like 1 is producing have cake and the other is producing not have cake.

These 2 actions are obviously mutually exclusive actions, then, interference: 1 of the effects of 1 action is the negation of a pre-condition of the other. For example, if cake produces not have cake, so, the persistence of have cake will conflict with this. Anything which requires have cake will conflict with this. The third is competing needs: 1 of the pre-conditions of 1 action is mutually exclusive with a pre-condition of the other. For example, if you look at bake cake and eat cake- eat cake requires have cake and bake cake requires not have cake, so, they have competing needs. They are also mutually exclusive actions. These are the 3 rules which define which actions are inconsistent, which pairs of actions are inconsistent.

And then, a mutex relation exists between 2 literals- literals means the propositions that we have- if 1 is the negation of the other, like have cake and not have cake. So, previously, in the last slide, what I discussed was the mutex between the actions and this is the mutex between the literals.

(Refer Slide Time: 31:31)



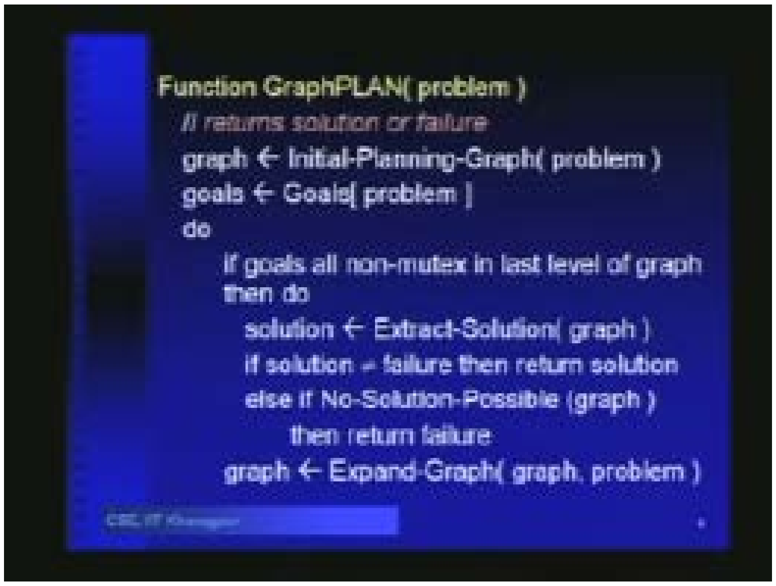
Mutex Literals

- Mutex relation between two literals if:
  - One is the negation of the other, or
  - Each possible pair of actions that could achieve the two literals is mutually exclusive (inconsistent support)

And the second 1 is each possible pair of actions that could achieve the 2 literals is mutually exclusive, which means we have inconsistent support. For example, in our example of the cakes, see, we have have cake here, we have eaten cake here; so, we have both of them, but they have inconsistent support, because eat cake can be achieved by this action and have cake can be achieved by this action, and these 2 actions are mutually exclusive. So, we will have have cake and eaten cake in the next time step and then, you see that I am having have cake by virtue of baking cake and I will have eaten cake by virtue of eating the cake in the previous time step.

These 2 actions are non-mutex; they can both take place, and why are these 2 non-mutex? Because these 2 actions, in turn, use 2 pre-conditions which are not conflicting, so, it is not have cake and this use eaten cake; these 2 where not conflicting. So therefore, I could use these 2 actions; so, there is no mutex between these 2 actions and they achieve this have cake and eaten cake, so, there is no mutex between these 2 again. Clear, right? Then, let us have a look at the function graph plan. The graph initially is the initial planning graph of the problem. Initial planning graph consists of the initial set of literals and we are given the goals, which is a conjunction of literals. Then, we do the following:

(Refer Slide Time: 34:57)



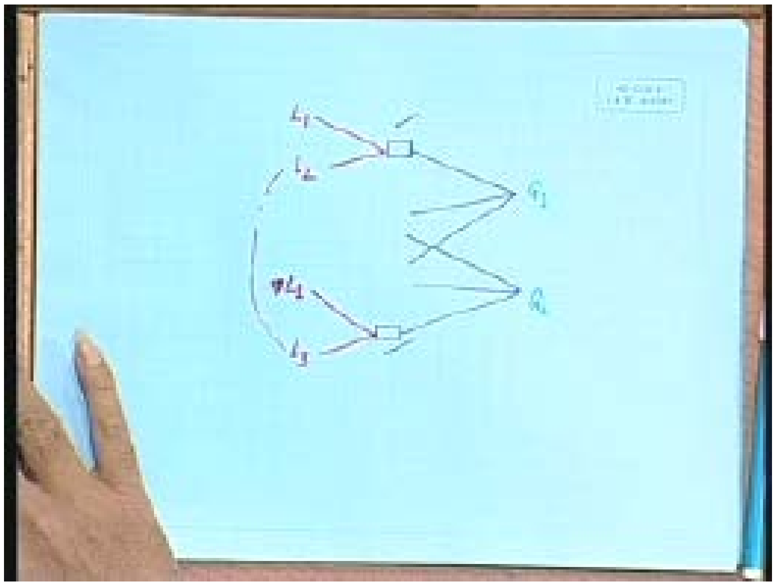
```
Function GraphPLAN( problem )
  // returns solution or failure
  graph ← Initial-Planning-Graph( problem )
  goals ← Goals[ problem ]
  do
      if goals all non-mutex in last level of graph
      then do
         solution ← Extract-Solution( graph )
         if solution ≠ failure then return solution
         else if No-Solution-Possible (graph )
               then return failure
      graph ← Expand-Graph( graph, problem )
```

If goals are all non-mutex in the last level of the graph- by last level, means that current last level- so, I am unfolding it 1 step at a time. If I find that the goals are all non-mutex in the last level of the graph, then, there is the possibility that I have a plan. So, we try to extract solution from the graph. If solution is not failure, which means that if we are able to find the solution, then, we return solution. I have not yet explained what is there inside this extract solution. But the idea is that we try to extract the solution from the graph and if we succeed, then, we return that as the plan.

Otherwise, we check whether no solution possible in the graph, and when can we say that? If we have already reached the fix point, then, we will say that no solution possible in the graph, then, return failure. Otherwise, we expand the graph to 1 more level; an expansion of the graph means that you take the set of current set of literal, and use the set of actions to deduce the next set of literals. This is the graph plan algorithm- it is a variant of iterative deepening, which you can see, that we are expanding 1 level at a time, then checking whether we have the goals in non-mutex way, and if so, then, we try to extract the solution, otherwise we expand another level of that.

Now, what does this extract solution do? It tries to work backwards from the set of goal propositions, which we have aceheived in the last level of the graph to determine that which is the set of independent actions which were able to produce these literals. I start with the set of goal propositions, so, suppose I have goal proposition G1 and I have goal proposition G2, and I have them in the last level of the graph in non-mutex fashion. So, I try to figure out that which are the set of actions. There can be many actions which have achieved this.

(Refer Slide Time: 37:46)



Now, I look at these actions and then, I have to find out a pair of non-mutex actions which are able to achieve this. Now, see that there can be more than 1 non-mutex actions pairs that achieve this. Which pair will eventually succeed all the way backwards we do not know. Suppose I pick this action and this action, then, this action has some pre-conditions, so, suppose this action has pre-conditions L1 and L2 and this has pre-conditions not of L1 and L3, right, or it still has L1 and L3 but then L2 and L3 are mutex. In this way, you will have to work backward and be able to determine the plan.

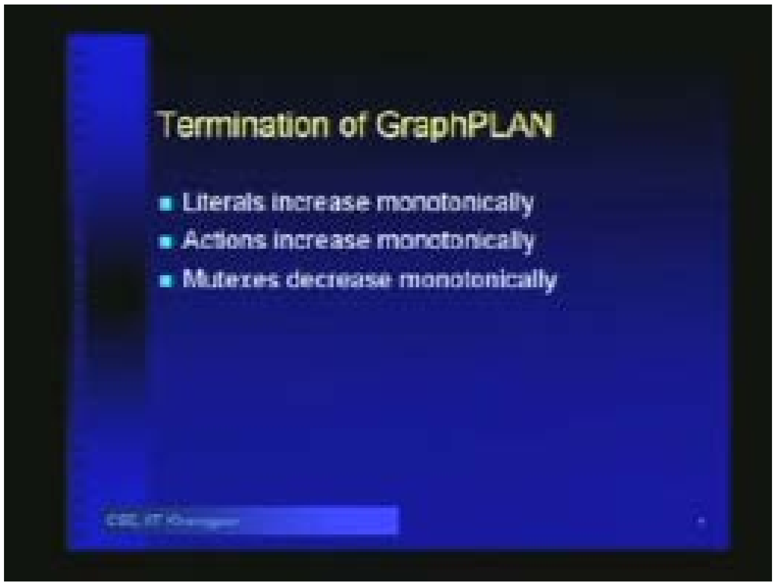
There is a search involved in this also. What I want you to do is to figure out that how this search will work. How will extract solution graph work? And then, I want you to create an example, where I have the goal propositions in a non-mutex way here, but the planning graph unfolded so far does not have a solution. This is the same exercise that I was talking about previously; what I want you to do is to figure out, that suppose I have the goal literals in a non-mutex way, and I want to figure out whether there exist a plan in the graph that I have unfolded so far. Try to think of an algorithm for doing this- that is part one; read the book Russell-Norvig or any other book which has this planning algorithm and try to figure out that what will be the algorithm of extracting this plan out of the unfolded state, unfolded plan graph.

Then, if you can figure that out properly, then, you will also understand how to create an example where the planning graph that you have so far still does not have a plan. It still does not have a plan, but the goal propositions are appearing in a non-mutex plan. See, this is very important to understand, because that is going to determine the complexity of the algorithm, so, you cannot stop the moment you find that the goal propositions have appeared in a non-mutex way. That is not sufficient for stopping the planning algorithm; you may still have to unfold a few more levels. (Students speaking).

You apply all the action you- (Students speaking). Yes. (Students speaking). Yes, the problem is, we are taking it pair-wise; we are just looking at it pair wise, not set wise. We are not considering subsets of actions which are independent; we just looking at pairs. This is just a hint that where the problem lies, but I want you to figure it out. I want you to look at some examples, try to work it out in your mind. But the main problem here is that we have just looking at pairs of actions which are incompatible.

Just try to think whether transitivity will hold in this inconsistency relation, if a and b are consistent, b and c are consistent, then, are a and c consistent? If a and b are inconsistent, if b and c are inconsistent, then, can a and c ever be consistent? Try to answer these questions, then you will get a flavor of what you need to do. The termination of graph plan depends on a set of things. Firstly, is literals increase monotonically, so, once a literal has arrived in the plan, it will always be there. Because the persistence actions will ensure that they will always be there.

(Refer Slide Time: 42:45)



Termination of GraphPLAN

- Literals increase monotonically
- Actions increase monotonically
- Mutexes decrease monotonically

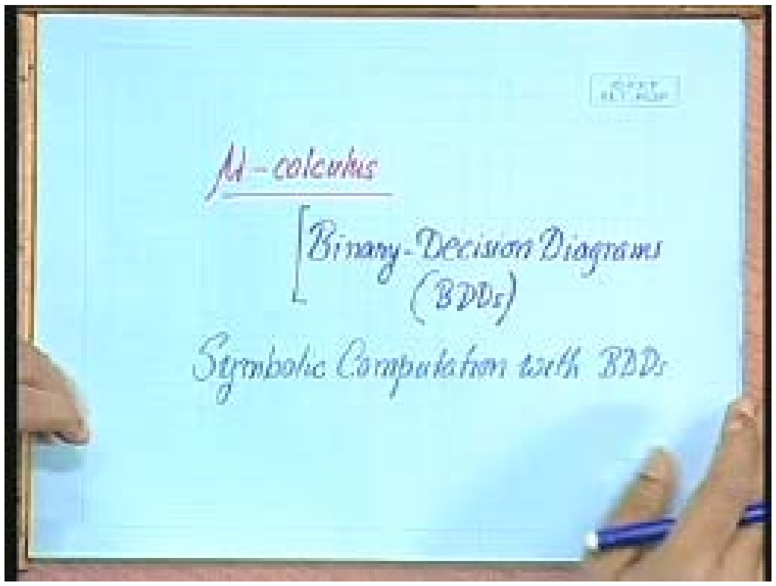
New actions can bring in new literals, but this cannot happen forever, because the set of literals is infinite. Actions increase monotonically- some action which was not applicable because the set of literals did not contain its pre-condition, can later on become applicable, because those pre-conditions have now coming to- those literals have now coming to the new state. So therefore, new actions can become applicable, but again, the set of actions that we have is finite, so, again, that also cannot happen forever. And the third thing is that mutex is decreased monotonically, so, something which is currently non-mutex can never become mutex literal. So, these are the 3 points which ensure that the graph plan algorithm will terminate.

Now, I am in a peculiar problem, because I do not have time to clean up SAT plan within the remaining time. So, let us do this- we will address SAT plan in the next lecture and any questions on this graph plan? (Students speaking). Yes. (Students speaking). L2 L3 are mutually exclusive, then, these 2 actions will also be mutually exclusive. (Students speaking). Yes it does, provided that- no, it could be that you are- that G1 and G2 is non-mutex, because there might be another pair of actions which maintain non-mutex. See, if you have any pair of non-mutex actions which are able to achieve this, then, the mutex link will not be there. So, my suggestion is that read through this algorithm in the book and then try to figure out, that how we are going to extract the solution from a partial planning graph.

(Student speaking). You have to apply it on everyone, on all literals. (Students speaking). Yes, for all literals and their negations; literals means that the propositions and their negations- you will have to have the persistence actions for all of them, right? That actually represents the fact that you can carry over these 2 the next time step, but new things will also come. Well, people actually devised this algorithm for several reasons; 1 was that we were at that point of time in the early 90s- new data structures were coming up which made this fix point computation quite convenient.

There is a branch of calculus called mu calculus. This mu calculus talks about computations in terms of fix points and tells you what kind of things you can compute and what you cannot. I am not going to go into the details of this, but then, there was a data structure called a binary decision diagram. Are you familiar with binary decision diagrams? (Students speaking). No, binary decision diagrams, which are alternatively called BDDs?

(Refer Slide Time: 48:31)



μ-calculus

[Binary-Decision Diagrams (BDDs)

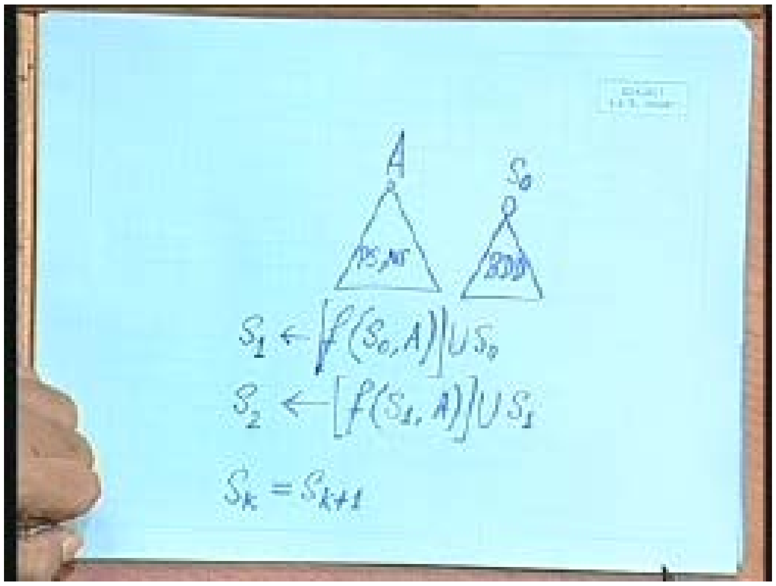
Symbolic Computation with BDDs

Well, BDDs are a kind of representation of Boolean functions. They are canonical and they are reasonably compact also, not for all functions, but given binary decision diagrams, once you construct the BDD, then, you can solve the SAT and the unSAT problems, just by traversing the BDD; say, traversing a path in the BDD. So, 1 subject that came up was that whether we could use these BDDs for fix point computations. And from a entirely different problem, there came up the notion of symbolic computation with these BDDs. What do these BDDs do?

Suppose we have a set of states and a set of states can be represented in binary; each state can have a binary encoding, fix length binary encoding, then, a set of states is nothing but a Boolean function which is true, if you present it with 1 of the valid states and is false for others. Any Boolean function can be thought of as a representation of a set which contains the valid min terms and does not contain the min terms which do not belong to that function. So, it is a set of min terms, so any Boolean set of items which can be encoded in Boolean in binary strings- so, that can be represented by a logic function. Is that clear?

Yes or no? Right. So, what we can do is, we can start with an initial set of states and we can have a BDD for that initial set of states. This is our S0, right, and suppose we have a set of actions A- so, what does A do? A checks whether the pre-condition of every action is there in S0 and if so, then the post condition of a gets added to this BDD. What happens is, S1 is actually some Boolean function of S0 and A, now- what this A? Is this A another BDD which has present state next, state pairs right, so, it has present state next state pairs.

(Refer Slide Time: 51:52)


A
S_0
PS, NS
BDD
$S_1 \leftarrow [f(S_0, A)] \cup S_0$
$S_2 \leftarrow [f(S_1, A)] \cup S_1$
$S_k = S_{k+1}$

If your present state is in the BDD, then, your pre-condition is satisfied and then, the next state is what? Is going to get produced, right? So, what we can do is, we can produce S1, which is basically the set of sates which are reachable by 1 action from S0. Now, which action- say, this A is the set of all actions, so, what we will have in S1 is the set of all states that are reachable from S0 by applying any of these actions. What we do is, we take not only this, but take the union of this with S0, so, this is our S0.

Then, our S2 is f of S1 with A, union with S1, right, and we continue to do this until we find that for some step, Sk is equal to Sk plus 1. Now, what BDD is helping us in doing is, it helps us doing these computations quite easily and the advantage of using this is that we are applying all the actions with a single Boolean operation. When we are doing this f S0 A, it is 1 single BDD operation which is going to have the effect of applying all the actions on the given state and to produce the entire collection of next states that these set of actions can produce and the BDD will be a compact representation for this next set of things. This is why now, in more recent times, people have attempted to solve the graph plan kind of problems with these kind of data structures.

And more recently, people find out that symbolic computation with BDDs has certain problems, like the BDD sizes can combinatorially explode for large number of variables. On the other hand, Boolean satisfiability is something which people have studied from ages. Boolean satisfiability or cnf satisfiability is something which people have studied for nearly, you know, 6-7 decades now, if not more. So therefore, the kind of solvers that we have for solving SAT problems are very rich.

So, what people found out is that rather than representing the Boolean functions has BDDs, if you just represent them as a set of classes, then, the existing SAT solvers are able to solve the problem much more quickly than you can do with BDD, then, the space requirement is much lesser. So, the SAT plan algorithm attempts to harness the power of the currently available sat solvers. In the next lecture, we will talk about SAT plan.