Artificial Intelligence-BSCE-306L

Module 6

Planning

Dr . Saurabh Agrawal

Faculty Id: 20165

School of Computer Science and Engineering

VIT, Vellore-632014

Tamil Nadu, India

Outline

- ❑Classical Planning (RN_C_10.1)
- ❑Planning as State-space search (RN_C_10.2)
- ❑Forward search (RN_C_10.2.1)
- ❑Backward search (RN_C_10.2.2)
- ❑Planning graphs (RN_C_10.3)
- ❑Hierarchical Planning (RN_C_11.2)
- ❑Planning and acting in Nondeterministic domains (RN_C_11.3)
- ❑Sensor-less Planning (RN_C_11.3.1)
- ❑Multiagent planning (RN_C_11.4)

Classical Planning

- **What is planning?**
- “Devising a plan of action to achieve one’s goals”

Planning = How do I get from here to there?

- Planning systems are problem-solving algorithms that operate on explicit propositional or relational representations of states and actions
- Planning problem: find a plan that is guaranteed (from any of the initial states) to generate a sequence of actions that leads to one of the goal states
- Planning problems often have large state spaces

Classical Planning

❑ Automated Planning

❑ We will look at two popular and effective current approaches to automated classical planning:

- Forward state-space search with heuristics
- Translating to a Boolean satisfiability problem

❑ There are also other approaches

- e.g. planning graphs: data structures to give better heuristic estimates than other methods, and also used to search for a solution over the space formed by the planning graph

Classical Planning

❑**Representing Planning Problems**

❑Recall search based problem-solving agents

▪Find sequences of actions that result in a goal state BUT deal with *atomic states so need good domain specific* heuristics to perform well

❑Planning represented by factored representation

▪Represent a state by a collection of variables

❑Planning Domain Definition Language (PDDL)

▪Allows expression of all actions with one schema

▪Inspired by earlier STRIPS planning language

Classical Planning

- **Defining a Search Problem :** Define a search problem through:

1. Initial state
2. Actions available in a state
3. Result of action
4. Goal test

❑PDDL – Representing States (I)

❑A state is represented by a conjunction of fluents

❑These are ground, functionless atoms

- Example: At(Truck1,Manchester) ∧ At(Truck2,Warrington)

❑Closed world assumption (no facts = false)

❑Unique names assumption (Truck1 distinct from Truck2)

Classical Planning

❑PDDL – Representing States (II)

❑Not allowed:

❑At(x,y) non-ground (i.e. variables alone)

- ¬ Poor negation
- At(Father(Fred),Liverpool) uses function

❑A state is treated as either

- conjunction of fluents, manipulated by logical inference
- set of fluents, manipulated with set operations

❑PDDL – Representing Actions

❑Actions described by a set of action schemas that implicitly define Actions(s) and Result(s,a) functions

❑Classical planning: most actions leave most states unchanged

▪Relates to the Frame Problem: issue of what changes and what stays the same as a result of actions

❑PDDL specifies the result of an action in terms of *what changes – don't need to mention* everything that stays the same

❑Action Schema (I)

- Represents a set of ground actions
- Contains action name, list of variables used, precondition and effect
- Example: action schema for flying a plane from one location to another

Action(Fly(p,from,to),

PRECOND: At(p,from) ∧ Plane(p) ∧

Airport(from) ∧ Airport(to)

EFFECT: ¬ At(p,from)∧ At(p,to))

❑Action Schema (II)

❑Free to choose whatever values we want to instantiate variables

❑Precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences)

- Precondition defines states in which action can be executed
- Effect defines result of action

❑Sometimes we want to *propositionalise a PDDL problem* (replace each action schema with a set of ground actions) and use a propositional solver (e.g. SATPLAN) to find a solution

- More on this later…

❑Action Schema (III)

❑Action a can be executed in state s if s entails the precondition of a

$(a \in Actions(s)) \Leftrightarrow s \models Precond(a)$

where any variables in a are universally quantified

▪Example:

$\forall p, from, to\ (Fly(p, from, to) \in Actions(s)) \Leftrightarrow$

$s \models (At(p, from) \wedge Plane(p) \wedge Airport(from)$

$\wedge\ Airport(to))$

❑We say that a is applicable in s if the preconditions are satisfied by s

Action Schema (IV)

- Result of executing action *a in state s (s')*
  - Result(s,a)=(s-Del(a)) U Add(a)
- Delete list (Del(a)): fluents that appear as negative literals in action’s effect
- Add list (Add(a)): fluents that appear as positive literals in action’s effect
- Note that time is implicit: preconditions have time t, effects have t+1

Classical Planning

❑Planning Domain

❑A set of action schemas defines a planning domain

❑A specific problem within a domain is defined by adding initial state and goal

- Initial state: conjunction of ground atoms
- Goal: conjunction of literals (positive or negative) that may contain variables
- e.g. $At(p,LPL) \wedge Plane(p)$

❑Problem solved when we find sequence of actions that end in a state that entails the goal

- e.g. $Plane(Plane1) \wedge At(Plane1,LPL)$ entails the goal $At(p,LPL) \wedge Plane(p)$

Classical Planning

❑Example: Air Cargo Transport

Init$(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK) \wedge$
$Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge$
$Airport(JFK) \wedge Airport(SFO))$
Goal$(At(C_1, JFK) \wedge At(C_2, SFO))$

Action(Load(c,p,a),
PRECOND: $At(c,a) \wedge At(p,a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $\neg At(c,a) \wedge In(c,p))$

Action(Unload(c,p,a),
PRECOND: $In(c,p) \wedge At(p,a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $At(c,a) \wedge \neg In(c,p))$

Action(Fly(p,from,to),
PRECOND: $At(p,from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p,from) \wedge At(p,to))$

Classical Planning

- **Example: Air Cargo Transport**
- Problem defined with 3 actions
- Actions affect 2 predicates
- When a plane flies from one airport to another, all cargo inside goes too
  - in PDDL we have no explicit universal quantifier to say this as part of the Fly action
  - so instead we use the load/unload actions:
    - cargo seizes to be At the old airport when it is loaded
    - and only becomes At the new airport when it is unloaded
- A solution plan:

[Load(C1,P1,SFO),Fly(P1,SFO,JFK),Unload(C1,P1,JFK),
Load(C2.P2.JFK),Fly(P2,JFK,SFO),Unload(C2,P2,SFO)].

- Problem – spurious actions like *Fly(P1,JFK,JFK) have contradictory effects*
  - Add inequality preconditions ∧ *(from ≠ to)*

Planning as State-space search

- Now we turn our attention to planning algorithms.
- We saw how the description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal.
- One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state.
- Figure 10.5 compares forward and backward searches.

Planning as State-space search

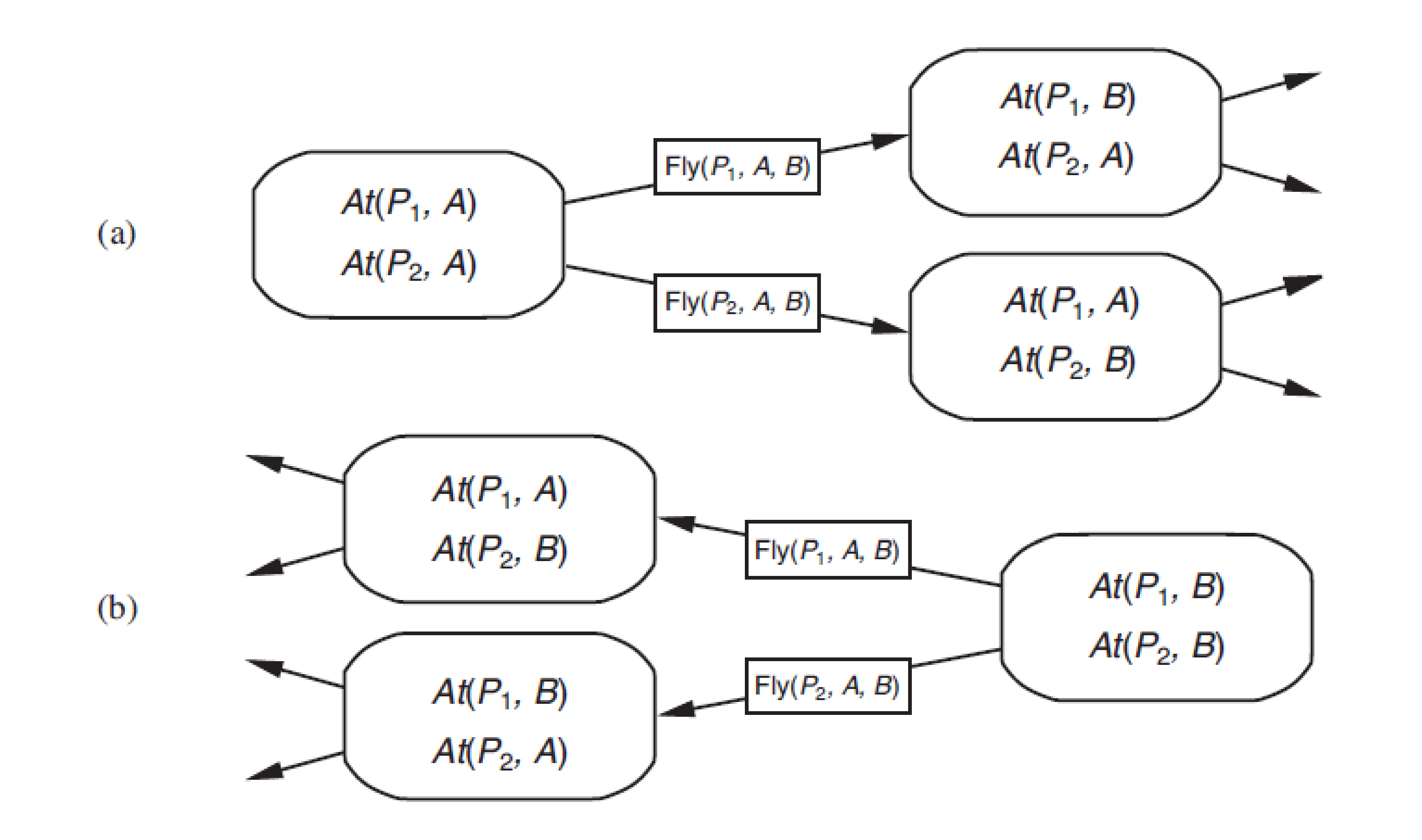

Figure 10.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

Forward (progression) State-space search

- Now that we have shown how a planning problem maps into a search problem, we can solve planning problems with any of the heuristic search algorithms or a local search algorithm (provided we keep track of the actions used to reach the goal).
- First, forward search is prone to exploring irrelevant actions.
- Consider the noble task of buying a copy of *AI: A Modern Approach from an online bookseller.*
- *Suppose there is an* action schema Buy(isbn) with effect Own(isbn).
- ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.

Forward (progression) State-space search

- Second, planning problems often have large state spaces.
- Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo.
- The goal is to move all the cargo at airport A to airport B.
- There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo.
- Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded).
- So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport).
- On average, let’s say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 2000^{41} nodes.

Forward (progression) State-space search

- Clearly, even this relatively small problem instance is hopeless without an accurate heuristic.
- Although many real-world applications of planning have relied on domain-specific heuristics, it turns out that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible.

Backward (regression) relevant states search

- In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state.
- It is called **relevant-states search because we** only consider actions that are relevant to the goal (or current state).
- As in belief-state search , there is a set of relevant states to consider at each step, not just a single state.
- We start with the goal, which is a conjunction of literals forming a description of a set of states—for example, the goal $\neg Poor \wedge Famous$ describes those states in which Poor is false, Famous is true, and any other fluent can have any value.
- If there are n ground fluents in a domain, then there are 2^n ground states (each fluent can be true or false), but 3^n descriptions of sets of goal states (each fluent can be positive, negative, or not mentioned).

Backward (regression) relevant states search

- In general, backward search works only when we know how to regress from a state description to the predecessor state description.
- For example, it is hard to search backwards for a solution to the n-queens problem because there is no easy way to describe the states that are one move away from the goal.
- Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it.
- Given a ground goal description g and a ground action a, the regression from g over a gives us a state description g’ defined by.

$$g' = (g - ADD(a)) \cup Precond\ (a)$$

Backward (regression) relevant states search

- That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed.
- Note that DEL(a) does not appear in the formula; that's because while we know the fluents in DEL(a) are no longer true after the action, we don't know whether or not they were true before, so there's nothing to be said about them.
- To get the full advantage of backward search, we need to deal with partially uninstantiated actions and states, not just ground ones.
- For example, suppose the goal is to deliver a specific piece of cargo to SFO: At(C_2, SFO). That suggests the action Unload(C_2, p', SFO):

$$
\begin{aligned}
&Action(Unload(C_2, p', SFO),\\
&\quad \text{PRECOND:} In(C_2, p') \land At(p', SFO) \land Cargo(C_2) \land Plane(p') \land Airport(SFO)\\
&\quad \text{EFFECT:} At(C_2, SFO) \land \neg In(C_2, p')\,.
\end{aligned}
$$

Backward (regression) relevant states search

- (Note that we have **standardized variable names (changing p to p'** in this case) so that there will be no confusion between variable names if we happen to use the same action schema twice in a plan.
- This represents unloading the package from an *unspecified plane at SFO; any plane will do,* but we need not say which one now.
- We can take advantage of the power of first-order representations: a single description summarizes the possibility of using *any of the planes by* implicitly quantifying over p'.
- The regressed state description is

$$g' = In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO) .$$

- The final issue is deciding which actions are candidates to regress over.
- In the forward direction we chose actions that were **applicable—those actions that could be the next step in the** plan.
- In backward search we want actions that are **relevant—those actions that could be the** *last step in a plan leading up to the current goal state.*

Backward (regression) relevant states search

- ❑For an action to be relevant to a goal it obviously must contribute to the goal: at least one of the action's effects (either positive or negative) must unify with an element of the goal.
- ❑What is less obvious is that the action must not have any effect (positive or negative) that negates an element of the goal.
- ❑Now, if the goal is $A \land B \land C$ and an action has the effect $A \land B \land \neg C$ then there is a colloquial sense in which that action is very relevant to the goal—it gets us two-thirds of the way there.
- ❑But it is not relevant in the technical sense defined here, because this action could not be the *final step of a solution—we would always need at least* one more step to achieve C.

Backward (regression) relevant states search

- Given the goal At(C_2, SFO), several instantiations of Unload are relevant: we could chose any specific plane to unload from, or we could leave the plane unspecified by using the action Unload(C_2, p', SFO).
- We can reduce the branching factor without ruling out any solutions by always using the action formed by substituting the most general unifier into the (standardized) action schema.
- As another example, consider the goal Own(0136042597), given an initial state with 10 billion ISBNs, and the single action schema

$$A = Action(Buy(i), \text{PRECOND:} ISBN(i), \text{EFFECT:} Own(i))$$

Backward (regression) relevant states search

- As we mentioned before, forward search without a heuristic would have to start enumerating the 10 billion ground Buy actions.
- But with backward search, we would unify the goal Own(0136042597) with the (standardized) effect Own(I'), yielding the substitution $\theta = \{i'/0136042597\}$.
- Then we would regress over the action Subst(θ, A') to yield the predecessor state description ISBN (0136042597).
- This is part of, and thus entailed by, the initial state, so we are done.

Backward (regression) relevant states search

We can make this more formal. Assume a goal description g which contains a goal literal g_i and an action schema A that is standardized to produce A'. If A' has an effect literal e'_j where $Unify(g_i, e'_j) = \theta$ and where we define $a' = \text{SUBST}(\theta, A')$ and if there is no effect in a' that is the negation of a literal in g, then a' is a relevant action towards g.

Backward search keeps the branching factor lower than forward search, for most problem domains. However, the fact that backward search uses state sets rather than individual states makes it harder to come up with good heuristics. That is the main reason why the majority of current systems favor forward search.

Planning Graphs

- ❑Planning graph can be used to give better heuristic estimates.
- ❑We can search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.
- ❑A planning problem asks if we can reach a goal state from the initial state.
- ❑Suppose we are given a tree of all possible actions from the initial state to successor states, and their successors, and so on.
- ❑If we indexed this tree appropriately, we could answer the planning question “can we reach state G from state S_0 ” immediately, just by looking it up.
- ❑Of course, the tree is of exponential size, so this approach is impractical.
- ❑A planning graph is polynomialsize approximation to this tree that can be constructed quickly.
- ❑The planning graph can’t answer definitively whether G is reachable from S_0, but it can *estimate how many steps it* takes to reach G.
- ❑The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.

Planning Graphs

- A planning graph is a directed graph organized into levels:

1. first a level S_0 for the initial state, consisting of nodes representing each fluent that holds in S_0;
2. then a level A_0 consisting of nodes for each ground action that might be applicable in S_0;
3. then alternating levels S_i followed by A_i;

- until we reach a termination condition.
- Si contains all the literals that could hold at time i, depending on the actions executed at preceding time steps.
- If it is possible that either P or $\neg$P could hold, then both will be represented in S_i .
- Also, A_i contains all the actions that could have their preconditions satisfied at time i.

Planning Graphs

- Planning graphs work only for propositional planning problems: ones with no variables.
- It is straightforward to propositionalize a set of action schemas.
- Despite the resulting increase in the size of the problem description, planning graphs have proved to be effective tools for solving hard planning problems.

Planning Graphs

Init(*Have*(*Cake*))
Goal(*Have*(*Cake*) ∧ *Eaten*(*Cake*))
Action(*Eat*(*Cake*)
  PRECOND: *Have*(*Cake*)
  EFFECT: ¬ *Have*(*Cake*) ∧ *Eaten*(*Cake*))
Action(*Bake*(*Cake*)
  PRECOND: ¬ *Have*(*Cake*)
  EFFECT: *Have*(*Cake*))

Figure 10.7 The "have cake and eat cake too" problem.

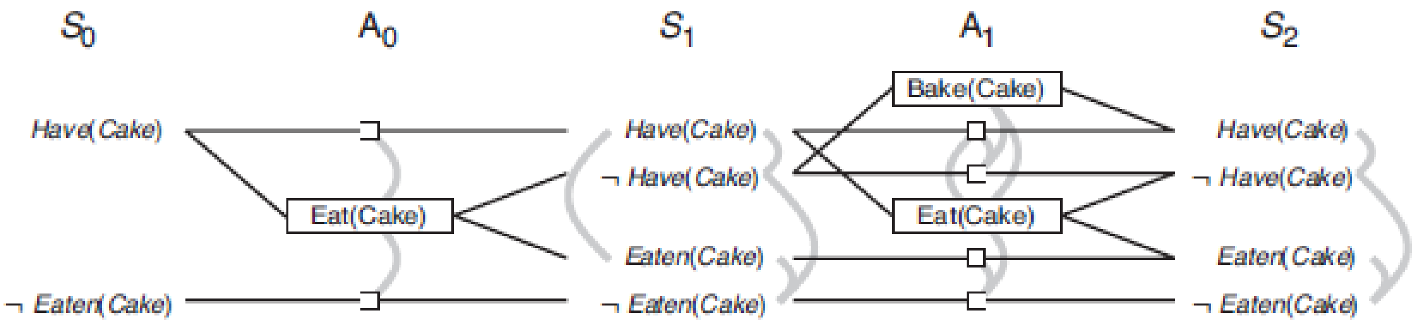

Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i and we need not draw that mutex link.

Planning Graphs

- Figure 10.7 shows a simple planning problem, and Figure 10.8 shows its planning graph.
- Each action at level A_i is connected to its preconditions at S_i and its effects at S_{i+1}.
- So a literal appears because an action caused it, but we also want to say that a literal can persist if no action negates it.
- This is represented by a **persistence action (sometimes called** a no-op).
- For every literal C, we add to the problem a persistence action with precondition C and effect C.
- Level A_0 in Figure 10.8 shows one “real” action, Eat (Cake), along with two persistence actions drawn as small square boxes.
- Level A_0 contains all the actions that could occur in state S_0, but just as important it records conflicts between actions that would prevent them from occurring together.
- The gray lines in Figure 10.8 indicate **mutual exclusion (or mutex) links.**
- **For example, Eat (Cake) is** mutually exclusive with the persistence of either Have(Cake) or ¬Eaten(Cake).
- We shall see shortly how mutex links are computed.

Planning Graphs

- Level S_1 contains all the literals that could result from picking any subset of the actions in A_0, as well as mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions.
- For example, Have(Cake) and Eaten(Cake) are mutex: depending on the choice of actions in A_0, either, but not both, could be the result.
- In other words, S_1 represents a belief state: a set of possible states.
- The members of this set are all subsets of the literals such that there is no mutex link between any members of the subset.
- We continue in this way, alternating between state level S_i and action level A_i until we reach a point where two consecutive levels are identical.
- At this point, we say that the graph has leveled off.
- The graph in Figure 10.8 levels off at S_2 .

Planning Graphs

- What we end up with is a structure where every A_i level contains all the actions that are applicable in S_i, along with constraints saying that two actions cannot both be executed at the same level.
- Every S_i level contains all the literals that could result from any possible choice of actions in A_{i-1}, along with constraints saying which pairs of literals are not possible.
- It is important to note that the process of constructing the planning graph does *not require* choosing among actions, which would entail combinatorial search.
- Instead, it just records the impossibility of certain choices using mutex links.

Planning Graphs

- We now define mutex links for both actions and literals.
- A mutex relation holds between two actions at a given level if any of the following three conditions holds:

1. **Inconsistent effects:** one action negates an effect of the other. For example, Eat (Cake) and the persistence of Have(Cake) have inconsistent effects because they disagree on the effect Have(Cake).
2. **Interference:** one of the effects of one action is the negation of a precondition of the other. For example Eat (Cake) interferes with the persistence of Have(Cake) by negating its precondition.
3. **Competing needs:** one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, Bake(Cake) and Eat (Cake) are mutex because they compete on the value of the Have(Cake) precondition.

Planning Graphs

- A mutex relation holds between two literals at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive.
- This condition is called inconsistent support.
- For example, Have(Cake) and Eaten(Cake) are mutex in S_1 because the only way of achieving Have(Cake), the persistence action, is mutex with the only way of achieving Eaten(Cake), namely Eat (Cake).
- In S_2 the two literals are not mutex, because there are new ways of achieving them, such as Bake(Cake) and the persistence of Eaten(Cake), that are not mutex.

Hierarchical Planning

- The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions.
- Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions.
- For plans executed by the human brain, atomic actions are muscle activations.
- In very round numbers, we have about 10^3 muscles to activate (639, by some counts, but many of them have multiple subunits); we can modulate their activation perhaps 10 times per second; and we are alive and awake for about 10^9 seconds in all.
- Thus, a human life contains about 10^{13} actions, give or take one or two orders of magnitude.
- Even if we restrict ourselves to planning over much shorter time horizons—for example, a two-week vacation in Hawaii—a detailed motor plan would contain around 10^{10} actions.
- This is a lot more than 1000.

Hierarchical Planning

- To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction.
- A reasonable plan for the Hawaii vacation might be “Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation stuff for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home.”
- Given such a plan, the action “Go to San Francisco airport” can be viewed as a planning task in itself, with a solution such as “Drive to the long-term parking lot; park; take the shuttle to the terminal.”
- Each of these actions, in turn, can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequences.

Hierarchical Planning

- In this example, we see that planning can occur both before and during the execution of the plan; for example, one would probably defer the problem of planning a route from a parking spot in long-term parking to the shuttle bus stop until a particular parking spot has been found during execution.
- Thus, that particular action will remain at an abstract level prior to the execution phase.
- Here, we concentrate on the aspect of **hierarchical decomposition, an idea that pervades almost all** attempts to manage complexity.

Hierarchical Planning

- For example, complex software is created from a hierarchy of subroutines or object classes; armies operate as a hierarchy of units; governments and corporations have hierarchies of departments, subsidiaries, and branch offices.
- The key benefit of hierarchical structure is that, at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a small number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small.
- Nonhierarchical methods, on the other hand, reduce a task to a large number of individual actions; for large-scale problems, this is completely impractical.

Hierarchical Planning

❑High Level Actions

❑The basic formalism we adopt to understand hierarchical decomposition comes from the area of hierarchical task networks or HTN planning.

❑As in classical planning , we assume full observability and determinism and the availability of a set of actions, now called primitive actions, with standard precondition–effect schemas.

❑The key additional concept is the high-level action or HLA—for example, the action “Go to San Francisco airport” in the example given earlier.

❑Each HLA has one or more possible refinements, into a sequence of actions, each of which may be an HLA or a primitive action (which has no refinements by definition).

❑For example, the action “Go to San Francisco airport,” represented formally as Go(Home, SFO), might have two possible refinements, as shown in Figure 11.4.

❑The same figure shows a recursive refinement for navigation in the vacuum world: to get to a destination, take a step, and then go to the destination.

Hierarchical Planning

$Refinement(Go(Home, SFO),$
STEPS: $[Drive(Home, SFOLongTermParking),$
$Shuttle(SFOLongTermParking, SFO)]\ )$
$Refinement(Go(Home, SFO),$
STEPS: $[Taxi(Home, SFO)]\ )$

$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $a = x \ \wedge \ b = y$
STEPS: $[\,]\ )$
$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $Connected([a, b], [a - 1, b])$
STEPS: $[Left, Navigate([a - 1, b], [x, y])]\ )$
$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $Connected([a, b], [a + 1, b])$
STEPS: $[Right, Navigate([a + 1, b], [x, y])]\ )$
...

Figure 11.4 Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.

❑High Level Actions

❑These examples show that high-level actions and their refinements embody knowledge about how to do things.

❑For instance, the refinements for Go(Home, SFO) say that to get to the airport you can drive or take a taxi; buying milk, sitting down, and moving the knight to e4 are not to be considered.

❑An HLA refinement that contains only primitive actions is called an **implementation** of the HLA.

❑For example, in the vacuum world, the sequences [Right , Right , Down] and [Down, Right , Right] both implement the HLA Navigate([1, 3], [3, 2]).

❑An implementation of a high-level plan (a sequence of HLAs) is the concatenation of implementations of each HLA in the sequence.

❑Given the precondition–effect definitions of each primitive action, it is straightforward to determine whether any given implementation of a high-level plan achieves the goal.

Hierarchical Planning

❑High Level Actions

❑We can say, then, that a high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state.

❑The “at least one” in this definition is crucial—not all implementations need to achieve the goal, because the agent gets to decide which implementation it will execute.

❑Thus, the set of possible implementations in HTN planning—each of which may have a different outcome—is not the same as the set of possible outcomes in nondeterministic planning.

❑There, we required that a plan work for all outcomes because the agent doesn’t get to choose the outcome; nature does.

Planning and Acting in Nondeterministic Domain

- We extend planning to handle partially observable, nondeterministic, and unknown environments.
- The methods here are:
  1. Sensorless planning (also known as conformant planning) for environments with no observations;
  2. Contingency planning for partially observable and nondeterministic environments;
  3. Online planning and replanning for unknown environments.
- While the basic concepts are the same, there are also significant differences.
- These arise because planners deal with factored representations rather than atomic representations.
- This affects the way we represent the agent's capability for action and observation and the way we represent belief states—the sets of possible physical states the agent might be in—for unobservable and partially observable environments.
- We can also take advantage of many of the domain-independent methods for calculating search heuristics.

Planning and Acting in Nondeterministic Domain

- Consider this problem: given a chair and a table, the goal is to have them match—have the same color.
- In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown.
- Only the table is initially in the agent's field of view:

$$
\begin{aligned}
&Init(Object(Table) \land Object(Chair) \land Can(C_1) \land Can(C_2) \land InView(Table)) \\
&Goal(Color(Chair, c) \land Color(Table, c))
\end{aligned}
$$

- There are two actions: removing the lid from a paint can and painting an object using the paint from an open can.
- The action schemas are straightforward, with one exception: we now allow preconditions and effects to contain variables that are not part of the action's variable list.
- That is, Paint(x, can) does not mention the variable c, representing the color of the paint in the can.
- In the fully observable case, this is not allowed—we would have to name the action Paint(x, can, c).
- But in the partially observable case, we might or might not know what color is in the can.
- (The variable c is universally quantified, just like all the other variables in an action schema.)

Planning and Acting in Nondeterministic Domain

$$
\begin{aligned}
&Action(RemoveLid(can),\\
&\quad \text{PRECOND: } Can(can)\\
&\quad \text{EFFECT: } Open(can))\\
&Action(Paint(x, can),\\
&\quad \text{PRECOND: } Object(x) \wedge Can(can) \wedge Color(can, c) \wedge Open(can)\\
&\quad \text{EFFECT: } Color(x, c))
\end{aligned}
$$

- To solve a partially observable problem, the agent will have to reason about the percepts it will obtain when it is executing the plan.
- The percept will be supplied by the agent's sensors when it is actually acting, but when it is planning it will need a model of its sensors.
- For planning, we augment PDDL with a new type of schema, the **percept schema:**

$$
\begin{aligned}
&Percept(Color(x, c),\\
&\quad \text{PRECOND: } Object(x) \wedge InView(x)\\
&Percept(Color(can, c),\\
&\quad \text{PRECOND: } Can(can) \wedge InView(can) \wedge Open(can)
\end{aligned}
$$

Planning and Acting in Nondeterministic Domain

- The first schema says that whenever an object is in view, the agent will perceive the color of the object (that is, for the object x, the agent will learn the truth value of Color (x, c) for all c).
- The second schema says that if an open can is in view, then the agent perceives the color of the paint in the can.
- Because there are no exogenous events in this world, the color of an object will remain the same, even if it is not being perceived, until the agent performs an action to change the object's color.
- Of course, the agent will need an action that causes objects (one at a time) to come into view:

$$\begin{aligned}&Action(LookAt(x),\\&\qquad \text{PRECOND:} InView(y) \land (x \neq y)\\&\qquad \text{EFFECT:} InView(x) \land \neg InView(y))\end{aligned}$$

Planning and Acting in Nondeterministic Domain

- For a fully observable environment, we would have a Percept axiom with no preconditions for each fluent.
- A sensorless agent, on the other hand, has no Percept axioms at all.
- Note that even a sensorless agent can solve the painting problem.
- One solution is to open any can of paint and apply it to both chair and table, thus **coercing them to be the same color (even** though the agent doesn't know what the color is).
- A contingent planning agent with sensors can generate a better plan. First, look at the table and chair to obtain their colors; if they are already the same then the plan is done.
- If not, look at the paint cans; if the paint in a can is the same color as one piece of furniture, then apply that paint to the other piece. Otherwise, paint both pieces with any color.

Planning and Acting in Nondeterministic Domain

- Finally, an online planning agent might generate a contingent plan with fewer branches at first—perhaps ignoring the possibility that no cans match any of the furniture—and deal with problems when they arise by replanning.
- It could also deal with incorrectness of its action schemas.
- Whereas a contingent planner simply assumes that the effects of an action always succeed—that painting the chair does the job—a replanning agent would check the result and make an additional plan to fix any unexpected failure, such as an unpainted area or the original color showing through.
- In the real world, agents use a combination of approaches.
- Car manufacturers sell spare tires and air bags, which are physical embodiments of contingent plan branches designed to handle punctures or crashes.
- On the other hand, most car drivers never consider these possibilities; when a problem arises they respond as replanning agents.

Planning and Acting in Nondeterministic Domain

- In general, agents plan only for contingencies that have important consequences and a nonnegligible chance of happening.
- Thus, a car driver contemplating a trip across the Sahara desert should make explicit contingency plans for breakdowns, whereas a trip to the supermarket requires less advance planning.
- We next look at each of the three approaches in more detail.

Sensor less Planning

❑Basic idea of searching in belief-state space to find a solution for sensorless problems.

❑Conversion of a sensorless planning problem to a beliefstate planning problem works much.

❑The main differences are that the underlying physical transition model is represented by a collection of action schemas and the belief state can be represented by a logical formula instead of an explicitly enumerated set of states.

❑For simplicity, we assume that the underlying planning problem is deterministic.

❑The initial belief state for the sensorless painting problem can ignore InView fluents because the agent has no sensors.

❑Furthermore, we take as given the unchanging facts Object(Table) ∧ Object(Chair) ∧ Can(C1) ∧ Can(C2) because these hold in every belief state.

❑The agent doesn't know the colors of the cans or the objects, or whether the cans are open or closed, but it does know that objects and cans have colors: ∀ x ∃ c Color (x, c).

❑After Skolemizing, we obtain the initial belief state:

$$b_0 = Color(x, C(x)) .$$

Sensor less Planning

- In classical planning, where the **closed-world assumption is made, we would assume that** any fluent not mentioned in a state is false, but in sensorless (and partially observable) planning we have to switch to an **open-world assumption in which states contain both positive** and negative fluents, and if a fluent does not appear, its value is unknown.
- Thus, the belief state corresponds exactly to the set of possible worlds that satisfy the formula.
- Given this initial belief state, the following action sequence is a solution:

$$[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)] .$$

- We now show how to progress the belief state through the action sequence to show that the final belief state satisfies the goal.

Sensor less Planning

- ❑First, note that in a given belief state b, the agent can consider any action whose preconditions are satisfied by b.
- ❑(The other actions cannot be used because the transition model doesn't define the effects of actions whose preconditions might be unsatisfied.)
- ❑The general formula for updating the belief state b given an applicable action a in a deterministic world is as follows:

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$$

where RESULT_P defines the physical transition model. For the time being, we assume that the initial belief state is always a conjunction of literals, that is, a 1-CNF formula. To construct the new belief state b', we must consider what happens to each literal ℓ in each physical state s in b when action a is applied. For literals whose truth value is already known in b, the truth value in b' is computed from the current value and the add list and delete list of the action. (For example, if ℓ is in the delete list of the action, then $\neg\ell$ is added to b'.) What about a literal whose truth value is unknown in b? There are three cases:

1. If the action adds ℓ, then ℓ will be true in b' regardless of its initial value.
2. If the action deletes ℓ, then ℓ will be false in b' regardless of its initial value.
3. If the action does not affect ℓ, then ℓ will retain its initial value (which is unknown) and will not appear in b'.

Sensor less Planning

Hence, we see that the calculation of b' is almost identical to the observable case, which was specified by Equation (10.1) on page 368:

$$b' = \text{RESULT}(b, a) = (b - \text{DEL}(a)) \cup \text{ADD}(a) \; .$$

We cannot quite use the set semantics because (1) we must make sure that b' does not contain both ℓ and $\neg\ell$, and (2) atoms may contain unbound variables. But it is still the case that $\text{RESULT}(b, a)$ is computed by starting with b, setting any atom that appears in $\text{DEL}(a)$ to false, and setting any atom that appears in $\text{ADD}(a)$ to true. For example, if we apply *RemoveLid*(Can_1) to the initial belief state b_0, we get

$$b_1 = Color(x, C(x)) \wedge Open(Can_1) \; .$$

When we apply the action $Paint(Chair, Can_1)$, the precondition $Color(Can_1, c)$ is satisfied by the known literal $Color(x, C(x))$ with binding $\{x/Can_1, c/C(Can_1)\}$ and the new belief state is

$$b_2 = Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)) \; .$$

Finally, we apply the action $Paint(Table, Can_1)$ to obtain

$$\begin{aligned} b_3 = \; & Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)) \\ & \wedge Color(Table, C(Can_1)) \; . \end{aligned}$$

The final belief state satisfies the goal, $Color(Table, c) \wedge Color(Chair, c)$, with the variable c bound to $C(Can_1)$.

Sensor less Planning

- The preceding analysis of the update rule has shown a very important fact: the family of belief states defined as conjunctions of literals is closed under updates defined by PDDL action schemas.
- That is, if the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals.
- That means that in a world with n fluents, any belief state can be represented by a conjunction of size O(n).
- This is a very comforting result, considering that there are 2^n states in the world.
- It says we can compactly represent all the subsets of those 2^n states that we will ever need.
- Moreover, the process of checking for belief states that are subsets or supersets of previously visited belief states is also easy, at least in the propositional case.

Multiagent Planning

- ❑Multiagent Planning is experiential learning.
- ❑ You can refer the section number **11.4** of the prescribed text book as well as online materials.
- ❑ If anyone have doubt then ask me on or before 24 April 2024.

Note for Students

- **This power point presentation is for lecture, therefore it is suggested that also utilize the text books and lecture notes.**
- **Also Refer the solved and unsolved examples of Text and Reference Books.**