

# 21PCA101: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

## UNIT – III

### ➤ **Planning:**

#### • **Definition of Classical Planning**

- planning researchers have settled on a factored representation one in which a state of the world is represented by a collection of variables. We use a language called PDDL, the Planning Domain Definition Language, that allows us to express all 4T n2 PDDL actions with one action schema. There have been several versions of PDDL.
- We now show how PDDL describes the four things we need to define a search problem the initial state, the actions that are available in a state, the result of applying an action, and the goal test.
- Each state is represented as a conjunction of fluents that are ground, functionless atoms. For example,  $\text{Poor} \wedge \text{Unknown}$  might represent the state of a hapless agent, and a state in a package delivery problem might be  $\text{at}(\text{Truck 1, Melbourne}) \wedge \text{At}(\text{Truck 2, Sydney})$
- Database semantics is used: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that Truck 1 and Truck 2 are distinct.
- The following fluents are not allowed in a state:  $\text{At}(x, y)$  (because it is non-ground),  $\neg\text{Poor}$  (because it is a negation), and  $\text{at}(\text{Father}(\text{Fred}), \text{Sydney})$  (because it uses a function symbol). The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a set SET SEMANTICS of fluents, which can be manipulated with set operations.
- Actions are described by a set of action schemas that implicitly define the  $\text{ACTIONS}(s)$  and  $\text{RESULT}(s, a)$  functions needed to do a problem-solving search.

That any system for action description needs to solve the frame problem to say what changes and what stays the same as the result of the action.

- A set of ground (variable-free) actions can be represented by a single action schema. The schema is a lifted representation it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic. For example, here is an action schema for flying a plane from one location to another:

*Action* (*Fly* (*p*, *from*, *to*),

PRECOND:  $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$

EFFECT:  $\neg At(p, from) \wedge At(p, to)$

✚ **For 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 action *a* is applicable in state *s* if the preconditions are satisfied by *s*. When an action schema *a* contains variables, it may have multiple applicable instantiations. For example, with the initial state defined in Figure 10.1, the *Fly* action can be instantiated as *Fly* (*P*<sub>1</sub>, *SFO*, *JFK*) or as *Fly* (*P*<sub>2</sub>, *JFK*, *SFO*), both of which are applicable in the initial state. If an action *a* has *v* variables, then, in a domain with *k* unique names of objects, it takes  $O(v^k)$  time in the worst case to find the applicable ground actions.
- Sometimes we want to propositionalize a PDDL problem—replace each action schema with a set of ground actions and then use a propositional solver such as SATPLAN to find a solution. However, this is impractical when *v* and *k* are large.
- The result of executing action *a* in state *s* is defined as a state *s*<sup>l</sup> which is represented by the set of fluents formed by starting with *s*, removing the fluents that appear as negative literals in the action's effects (what we call the delete list or  $DEL(a)$ ), and adding the fluents that are positive literals in the action's effects (what we call the add list or  $ADD(a)$ ):


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

- For example, with the action *Fly* ( $P_1, SFO, JFK$ ), we would remove  $At(P_1, SFO)$  and add  $At(P_1, JFK)$ . It is a requirement of action schemas that any variable in the effect must also appear in the precondition. That way, when the precondition is matched against the state  $s$ , all the variables will be bound, and  $\text{RESULT}(s, a)$  will therefore have only ground atoms. In other words, ground states are closed under the  $\text{RESULT}$  operation.
- Also note that the fluents do not explicitly refer to time, there we needed superscripts for time, and successor-state axioms of the form

$F^{t+1} \Leftrightarrow \text{ActionCauses}F^t \vee (F^t \wedge \neg \text{ActionCausesNot}F^t)$ .

- In PDDL the times and states are implicit in the action schemas: the precondition always refers to time  $t$  and the effect to time  $t + 1$ .
- A set of action schemas serves as a definition of a planning *domain*. A specific *problem*

within the domain is defined with the addition of an initial state and a goal.

 **Example:** The spare tire problem

Consider the problem of changing a flat tire. The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is an abstract one, with no sticky lug nuts or other complications. There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. A solution to the problem is [*Remove* (*Flat*, *Axle*), *Remove* (*Spare*, *Trunk*), *PutOn* (*Spare*, *Axle*)].

$Init(Tire(Flat) \wedge Tire(Spare) \wedge At(Flat, Axle) \wedge At(Spare, Trunk))$

$Goal(At(Spare, Axle))$  Action(Remove(obj, loc),

PRECOND:  $At(obj, loc)$

EFFECT:  $\neg At(obj, loc) \wedge At(obj, Ground)$ )

Action(PutOn(t, Axle),

PRECOND:  $Tire(t) \wedge At(t, Ground) \wedge \neg At(Flat, Axle)$

EFFECT:  $\neg At(t, Ground) \wedge At(t, Axle)$ ) Action(LeaveOvernight,

PRECOND:

EFFECT:  $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$

$\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk)$ )

### The simple spare tire problem.

✚ **Example:** The blocks world

One of the most famous planning domains is known as the block's world. This domain consists of a set of cube-shaped blocks sitting on a table.<sup>2</sup> The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it.

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A))$

$\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)$

$Goal(On(A, B) \wedge On(B, C))$

$Action(Move(b, x, y),$

PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$

$(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$

EFFECT:  $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$ )

$Action(MoveToTable(b, x),$

PRECOND:  $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),$  EFFECT:  $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$ )

A planning problem in the block's world: building a three-block tower. One solution is the sequence [ $MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)$ ].



Diagram of the blocks-world problem in Figure

The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks.

**Example:** A goal might be to get block  $A$  on  $B$  and block  $B$  on  $C$

- We use  $on(b, x)$  to indicate that block  $b$  is on  $x$ , where  $x$  is either another block or the table. The action for moving block  $b$  from the top of  $x$  to the top of  $y$  will be  $Move(b, x, y)$ . Now, one of the preconditions on moving  $b$  is that no other block be on it. In first-order logic, this would be  $\neg \exists x on(x, b)$  or, alternatively,  $\forall x \neg On(x, b)$ . Basic PDDL does not allow quantifiers, so instead we introduce a predicate  $Clear(x)$  that is true when nothing is on  $x$ .
- The action  $Move$  moves a block  $b$  from  $x$  to  $y$  if both  $b$  and  $y$  are clear. After the move is made,  $b$  is still clear but  $y$  is not. A first attempt at the  $Move$  schema is

*Action* ( $Move(b, x, y)$ ),

PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y)$ ,

EFFECT:  $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$ .

Unfortunately, this does not maintain  $Clear$  properly when  $x$  or  $y$  is the table.

- When  $x$  is the *Table*, this action has the effect  $Clear(Table)$ , but the table should not become clear; and when  $y = Table$ , it has the precondition  $Clear(Table)$ , but the table does not have to be clear

for us to move a block onto it. To fix this, we do two things.

**First:** we introduce another action to move a block  $b$  from  $x$  to the table:

*Action* ( $MoveToTable(b, x)$ ,

PRECOND:  $On(b, x) \wedge Clear(b)$ ,

EFFECT:  $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$ ).

**Second:** we take the interpretation of  $Clear(x)$  to be “there is a clear space on  $x$  to hold a block.” Under this interpretation,  $Clear(Table)$  will always be true.

- The only problem is that nothing prevents the planner from using  $Move(b, x, Table)$  instead of  $MoveToTable(b, x)$ . We could live with this problem it will lead to a larger-than-necessary search space, but will not lead to incorrect answers or we could introduce the predicate  $Block$  and add  $Block(b) \wedge Block(y)$  to the precondition of  $Move$ .

#### ✚ THE COMPLEXITY OF CLASSICAL PLANNING

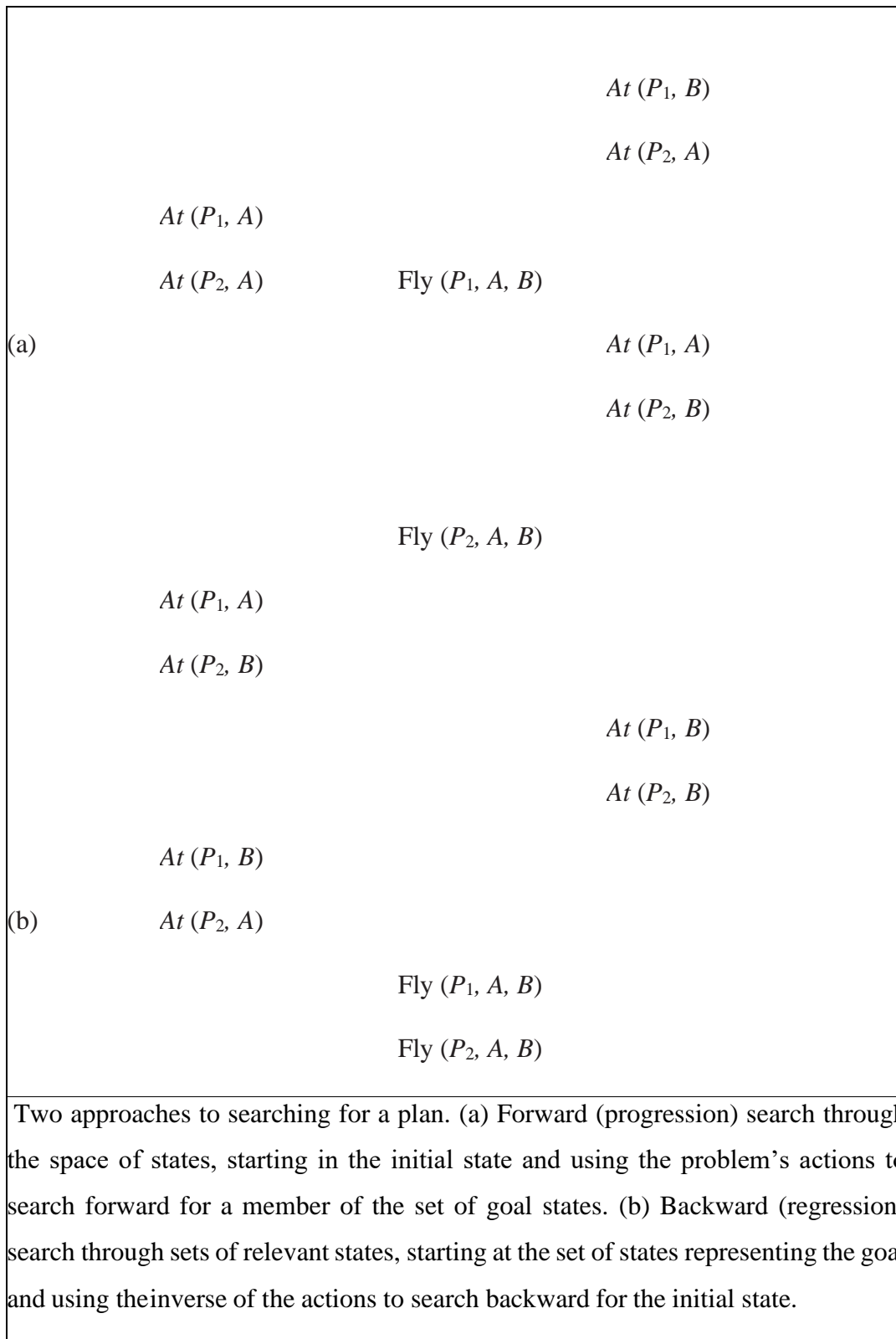
- we consider the theoretical complexity of planning and distinguish two decision problems. PlanSAT is the question of whether there exists any plan that solves a planning problem. Bounded PlanSAT asks whether there is a solution of length  $k$  or less; this can be used to find an optimal plan.
- The first result is that both decision problems are decidable for classical planning. The proof follows from the fact that the number of states is finite. But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semidecidable:
- An algorithm exists that will terminate with the correct answer for any solvable problem, but may not terminate on unsolvable problems. The Bounded PlanSAT problem remains decidable even in the presence of function symbols.
- These worst-case results may seem discouraging. We can take solace in the fact that agents are usually not asked to find plans for arbitrary worst-case problem instances, but rather are asked for plans in specific domains (such as blocks-world problems with  $n$  blocks), which can be much easier than the theoretical worst case.
- For many domains (including the blocks world and the air cargo world), Bounded PlanSAT is NP-complete while PlanSAT is in P; in other words, optimal planning is usually hard, but sub-optimal planning is sometimes easy. To do well on easier-than-worst-case problems, we will need good search heuristics.

- That's the true advantage of the classical planning formalism: it has facilitated the development of very accurate domain-independent heuristics, whereas systems based on successor-state axioms in first-order logic have had less success in coming up with good heuristics.

### ✚ PLANNING AS STATE-SPACE SEARCH

- 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. compares forward and backward searches.
  1. Forward (progression) state-space search
  2. Backward (regression) state-space search
- 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.





- 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

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).

- In general, backward search works only when we know how to regress from a state description to the predecessor state description.

**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  $a$  give us a state description  $g^j$  defined by

$$g^j = (g - \text{ADD}(a)) \cup \text{Precond}(a).$$

- 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  $\text{DEL}(a)$  does not appear in the formula; that's because while we know the fluents in  $\text{DEL}(a)$  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:  $\text{At}(C_2, \text{SFO})$ . That suggests the action

$\text{Unload}(C_2, p^j, \text{SFO})$ :

*Action* ( $\text{Unload}(C_2, p^j, \text{SFO})$ )

PRECOND:  $In(C_2, p^j) \wedge At(p^j, SFO) \wedge Cargo(C_2) \wedge Plane(p^j) \wedge Airport(SFO)$

EFFECT:  $At(C_2, SFO) \wedge \neg In(C_2, p^j)$ .

The regressed state description is

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

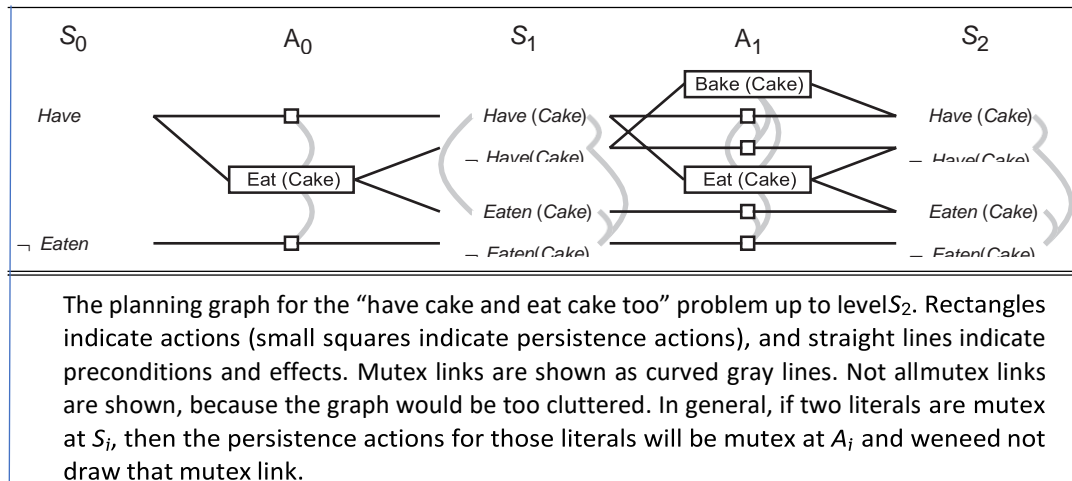
- The final issue is deciding which actions are candidates to regress over. In the forward direction we chose actions that were applicable those actions that could be the next step in the plan. In backward search we want actions that are relevant those actions that could be the *last* step in a plan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

### ➤ PLANNING GRAPHS

- All of the heuristics we have suggested can suffer from inaccuracies. This section shows how a special data structure called a planning graph can be used to give better heuristic estimates.
- These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, 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.
- A planning graph is polynomial-size 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.
- A planning graph is a directed graph organized into levels: first a level  $S_0$  for the initial state, consisting of nodes representing each fluent that holds in  $S_0$ ; then a level  $A_0$  consisting of nodes for each ground action that might be applicable in  $S_0$ ; then alternating levels  $S_i$  followed by  $A_i$ ; until we reach a termination condition
- $S_i$  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 roughly speaking,  $A_i$  contains all the actions that *could* have their preconditions satisfied at time  $i$ . We say "roughly speaking" because the planning

graph records only a restricted subset of the possible negative interactions among actions; therefore, a literal might show up at level  $S_j$  when actually it could not be true until a later level, if at all. (A literal will never show up too late.)

- Despite the possible error, the level  $j$  at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state.



- A mutex relation holds between two *actions* at a given level if any of the following three conditions holds:
  - **Inconsistent effects:** one action negates an effect of the other.
  - **Example:** *Eat (Cake)* and the persistence of *Have (Cake)* have inconsistent effects because they disagree on the effect *Have (Cake)*.
  - **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.
  - **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 for heuristic estimation**
- A planning graph, once constructed, is a rich source of information about the problem. First, if any goal literal fails to appear in the final level of the graph, then the problem is unsolvable. Second, we can estimate the cost of achieving any goal literal  $g_i$  from state  $s$  as the level at which  $g_i$  first appears in the planning graph constructed from initial state.
- it is common to use a serial planning graph for computing heuristics. A serial graph insists that only one action can actually occur at any given time step; this is done by adding mutex links between every pair of nonpersistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs
- To estimate the cost of a *conjunction* of goals, there are three simple approaches. The max-level heuristic simply takes the maximum level cost of any of the goal.
- The level sum heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this can be inadmissible but works well in practice for problems that are largely decomposable.
- The set-level heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. This heuristic gives the correct values of 2 for our original problem and infinity for the problem without *Bake (Cake)*. It is admissible, it dominates the max-level heuristic, and it works extremely well on tasks in which there is a good deal of interaction among subplans. It is not perfect, of course; for example, it ignores interactions among three or more literals.
- **The GRAPHPLAN algorithm**

This subsection shows how to extract a plan directly from the planning graph, rather than just using the graph to provide a heuristic. The GRAPHPLAN algorithm repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non-mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

```

function GRAPHPLAN (problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH (problem)
  goals ← CONJUNCTS (problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
    if goals all non-mutex in  $S_t$  of graph then
      solution ← EXTRACT-SOLUTION (graph, goals, NUMLEVELS
      (graph), nogoods)
      if solution ≠ failure then return solution
    if graph and nogoods have both leveled off then
      return failure
  graph ← EXPAND-GRAPH (graph,
  problem)

```

- The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.
- The goal *at (Spare, Axle)* is not present in  $S_0$ , so we need not call EXTRACT-SOLUTION — we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds into  $A_0$  the three actions whose preconditions exist at level  $S_0$  (i.e., all the actions except *PutOn (Spare, Axle)*), along with persistence actions for all the literals in  $S_0$ . The effects of the actions are added at level  $S_1$ . EXPAND-GRAPH then looks for mutex relations and adds them to the graph.
- *At (Spare, Axle)* is still not present in  $S_1$ , so again we do not call EXTRACT-SOLUTION. We call EXPAND-GRAPH again, adding  $A_1$  and  $S_1$  and giving us the planning graph shown in Figure. Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:
  - **Inconsistent effects:** *Remove (Spare, Trunk)* is mutex with *LeaveOvernight* because one has the effect *At (Spare, Ground)* and the other has its negation.
  - **Interference:** *Remove (Flat, Axle)* is mutex with *LeaveOvernight* because one has the precondition *At (Flat, Axle)* and the other has its negation as an effect.
  - **Competing needs:** *PutOn (Spare, Axle)* is mutex with *Remove (Flat, Axle)* because one has *At (Flat, Axle)* as a precondition and the other has its negation.
  - **Inconsistent support:** *At (Spare, Axle)* is mutex with *At (Flat, Axle)* in  $S_2$  because the

only way of achieving  $At(Spare, Axle)$  is by  $PutOn(Spare, Axle)$ , and that is mutex with the persistence action that is the only way of achieving  $At(Flat, Axle)$ . Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.

Alternatively, we can define EXTRACT-SOLUTION as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals. We define this search problem as follows:

- ✓ The initial state is the last level of the planning graph,  $S_n$ , along with the set of goals from the planning problem.
- ✓ The actions available in a state at level  $S_i$  are to select any conflict-free subset of the actions in  $A_{i-1}$  whose effects cover the goals in the state. The resulting state has level  $S_{i-1}$  and has as its set of goals the preconditions for the selected set of actions. By “conflict free,” we mean a set of actions such that no two of them are mutex and no two of their preconditions are mutex.
- ✓ The goal is to reach a state at level  $S_0$  such that all the goals are satisfied.
- ✓ The cost of each action is 1.

#### Termination of GRAPHPLAN

- The first thing to understand is why we can’t stop expanding the graph as soon as it has leveled off. Consider an air cargo domain with one plane and  $n$  pieces of cargo at airport  $A$ , all of which have airport  $B$  as their destination. In this version of the problem, only one piece of cargo can fit in the plane at a time.
- The graph will level off at level 4, reflecting the fact that for any single piece of cargo, we can load it, fly it, and unload it at the destination in three steps. But that does not mean that a solution can be extracted from the graph at level 4; in fact, a solution will require  $4n - 1$  steps: for each piece of cargo we load, fly, and unload, and for all but the last piece we need to fly back to airport  $A$  to get the next piece.
- ✓ *Literals increase monotonically:* Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; once a literal shows up, persistence actions cause it to stay forever.
- ✓ *Actions increase monotonically:* Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of the monotonic increase of literals; if

the preconditions of an action appear at one level, they will appear at subsequent levels, and thus so will the action.

- ✓ ***Mutexes decrease monotonically***: If two actions are mutex at a given level  $A_i$ , then they will also be mutex for all *previous* levels at which they both appear. The same holds for mutexes between literals. It might not always appear that way in the figures, because the figures have a simplification: they display neither literals that cannot hold at level  $S_i$  nor actions that cannot be executed at level  $A_i$ . We can see that “mutexes decrease monotonically” is true if you consider that these invisible literals and actions are mutex with everything.
- The proof can be handled by cases: if actions  $A$  and  $B$  are mutex at level  $A_i$ , it must be because of one of the three types of mutex. The first two, inconsistent effects and interference, are properties of the actions themselves, so if the actions are mutex at  $A_i$ , they will be mutex at every level.
- The third case, competing needs, depends on conditions at level  $S_i$ : that level must contain a precondition of  $A$  that is mutex with a precondition of  $B$ . Now, these two preconditions can be mutex if they are negations of each other (in which case they would be mutex in every level) or if all actions for achieving one are mutex with all actions for achieving the other. But we already know that the available actions are increasing monotonically, so, by induction, the mutexes must be decreasing.
- ***No-goods decrease monotonically***: If a set of goals is not achievable at a given level, then they are not achievable in any *previous* level. The proof is by contradiction: if they were achievable at some previous level, then we could just add persistence actions to make them achievable at a subsequent level.

➤ **Analysis of Planning approaches**

- Planning combines the two major areas of AI we have covered so far: *search* and *logic*. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution.
- The cross-fertilization of ideas from the two areas has led both to improvements in performance amounting to several orders of magnitude in the last decade and to an increased use of planners in industrial applications. Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge that dominate existing methods.



- Planning is foremost an exercise in controlling combinatorial explosion. If there are  $n$  propositions in a domain, then there are  $2^n$  states. As we have seen, planning is PSPACE-hard. Against such pessimism, the identification of independent subproblems can be a powerful weapon. In the best-case full decomposability of the problem, we get an exponential speedup.
- Decomposability is destroyed, however, by negative interactions between actions. GRAPHPLAN records mutexes to point out where the difficult interactions are. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure.
- Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent subproblems. Since this approach is heuristic, it can work even when the subproblems are not completely independent. Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out.
- We say that a problem has **serializable subgoals** if there exists an order of subgoals such that the planner can achieve them in that order without having to undo any of the previously achieved subgoals. For example, in the block's world, if the goal is to build a tower (e.g.,  $A$  on  $B$ , which in turn is on  $C$ ).
- Then the subgoals are serializable bottom to top: if we first achieve  $C$  on, we will never have to undo it while we are achieving the other subgoals. A planner that uses the bottom-to-top trick can solve any problem in the block's world without backtracking (although it might not always find the shortest plan).
- As a more complex example, for the Remote Agent planner that commanded NASA's Deep Space One spacecraft, it was determined that the propositions involved in commanding a spacecraft are serializable.
- This is perhaps not too surprising, because a spacecraft is *designed* by its engineers to be as easy as possible to control (subject to other constraints). Taking advantage of the serialized ordering of goals, the Remote Agent planner was able to eliminate most of the search. This meant that it was fast enough to control the spacecraft in real time, something previously considered impossible.
- Planners such as GRAPHPLAN, SATPLAN, and FF have moved the field of planning forward, by raising the level of performance of planning systems, by clarifying the representational and combinatorial issues involved, and by the

development of useful heuristics. However, there is a question of how far these techniques will scale. It seems likely that further progress on larger problems cannot rely only on factored and propositional representations, and will require some kind of synthesis of first-order and hierarchical representations with the efficient heuristics currently in use.

### ➤ PLANNING AND ACTING IN REAL WORLD

#### ✓ Time, Schedules and Resources

- The classical planning representation talks about *what to do*, and in *what order*, but the representation cannot talk about time: *how long* an action takes and *when* it occurs. **For Example**, the planners could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of **scheduling**. The real world also imposes many **resource constraints**.
- **For Example**, an airline has a limited number of staff and staff who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.
- The approach we take in this section is “plan first, schedule later”: that is, we divide the overall problem into **planning** phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later **scheduling** phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

```
Jobs ({AddEngine1 <AddWheels1 <Inspect1},
      {AddEngine2 <AddWheels2 <Inspect2})
Resources (EngineHoists (1), WheelStations (1), Inspectors (2), LugNuts
(500))
Action (AddEngine1, DURATION:30,
        USE: EngineHoists (1))
        Action (AddEngine2,
        DURATION:60, USE:
        EngineHoists (1))
        Action
(AddWheels1, DURATION:30,
        CONSUME: LugNuts (20), USE: WheelStations (1))
```

*Action (AddWheels2, DURATION:15,*  
*CONSUME: LugNuts (20), USE: WheelStations (1))*

*Action (Inspect i, DURATION:10,*  
*USE: Inspectors (1))*

**Figure A** job-shop scheduling problem for assembling two cars, with resource constraints. The notation  $A \prec B$  means that action  $A$  must precede action  $B$ .

- This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts.
- The automated methods can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness. GRAPHPLAN SATPLAN and partial-order planners can do this; search-based methods produce totally ordered plans, but these can easily be converted to plans with minimal ordering constraints.

#### ➤ **Representing temporal and resource constraints**

- A typical **job-shop scheduling problem**, as first introduced in consists of a set of **jobs**, each of which consists a collection of **actions** with ordering constraints among them. Each action has a **duration** and a set of resource constraints required by the action.
- Each constraint specifies a **type** of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is **consumable** (e.g., the bolts are no longer available for use) or **reusable** (e.g., a pilot is occupied during flight but is available again when the flight is over).
- Resources can also be **produced** by actions with negative consumption, including manufacturing, growing, and resupply actions. A solution to a job-shop scheduling problem must specify the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with

search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

- **RESOURCES** statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are *consumed* as wheels are added to the car, whereas the other resources are “borrowed” at the start of an action and released at the action’s end.
- The representation of resources as numerical quantities, such as *Inspectors (2)*, rather than as named entities, such as *Inspector (I<sub>1</sub>)* and *Inspector (I<sub>2</sub>)*, is an example of a very general technique called **aggregation**. The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand.
- ✓ **Solving scheduling problems**
  - We begin by considering just the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), we must find the earliest start times for all the actions consistent with the ordering constraints supplied with the problem. It is helpful to view these ordering constraints as a directed graph relating the actions, as we can apply the **critical path method** (CPM) to this graph to determine the possible start and end times of each action.
  - A **path** through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with *Start* and ending with *Finish*.
  - The **critical path** is that path whose total duration is longest; the path is “critical” because it determines the duration of the entire plan shortening other paths doesn’t shorten the plan as a whole, but delaying the start of any action on the critical path slows down the whole plan.
  - Actions that are off the critical path have a window of time in which they can be executed. The window is specified in terms of an earliest possible start time, *ES*, and a latest possible start time, *LS*.

- The quantity  $LS - ES$  is known as the **slack** of an action. that the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition). Together the  $ES$  and  $LS$  times for all the actions constitute a **schedule** for the problem.
- The following formulas serve as a definition for  $ES$  and  $LS$  and also as the outline of a dynamic-programming algorithm to compute them.  $A$  and  $B$  are actions, and  $A < B$  means that  $A$  comes before  $B$ :

$$ES(Start) = 0$$

$$ES(B) = \max_{A < B} ES(A) + Duration(A)$$

$$LS(Finish) = ES(Finish)$$

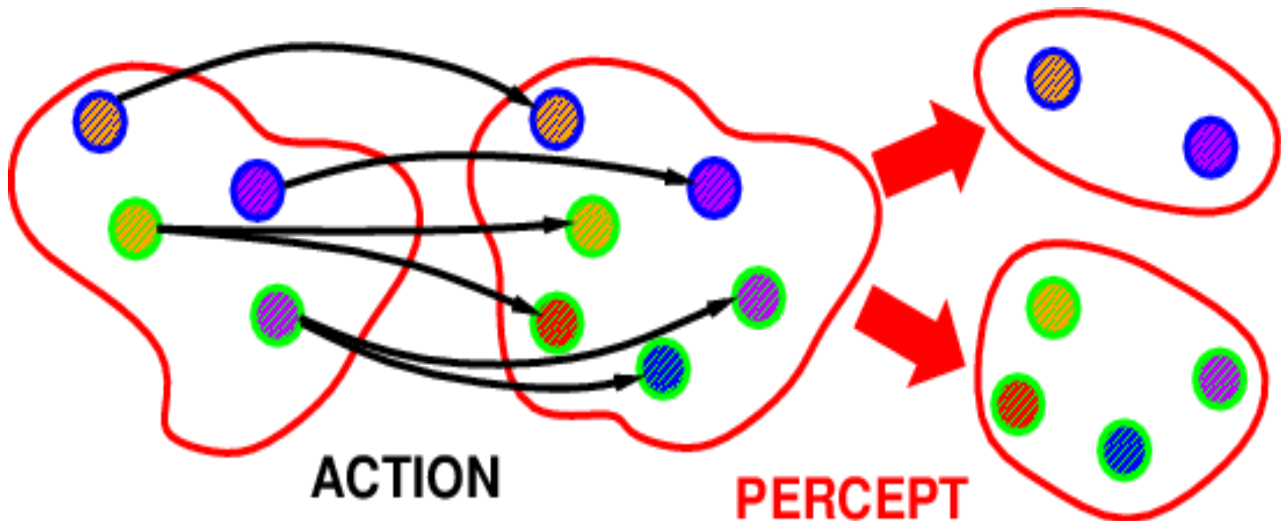
$$LS(A) = \min_{B > A} LS(B) - Duration(A).$$

- The idea is that we start by assigning  $ES(Start)$  to be 0. Then, as soon as we get an action  $B$  such that all the actions that come immediately before  $B$  have  $ES$  values assigned, we set  $ES(B)$  to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration.
- This process repeats until every action has been assigned an  $ES$  value. The  $LS$  values are computed in a similar manner, working backward from the *Finish* action
- The complexity of the critical path algorithm is just  $O(Nb)$ , where  $N$  is the number of actions and  $b$  is the maximum branching factor into or out of an action. (To see this, note that the  $LS$  and  $ES$  computations are done once for each action, and each computation iterates over at most  $b$  other actions.) Therefore, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy.
- Critical-path problems are easy to solve because they are de-fined as a **conjunction of linear** inequalities on the start and end times. When we introduce resource constraints, the resulting constraints on start and end times become more complicated. require the same *EngineHoist* and so cannot overlap.
- The “cannot overlap” constraint is a **disjunction** of two linear inequalities, one for each possible ordering. The introduction of disjunctions turns out to make scheduling

with resource constraints NP-hard.

### ✓ Conditional Planning

- If the world is nondeterministic or partially observable then percepts usually provide information, i.e., split up the belief state



Conditional plans check (any consequence of KB +) percept

...if  $C$  then  $Plan_A$  else  $Plan_B$ ..

Execution: check  $C$  against current KB, execute “then” or “else” Need

*some* plan for *every* possible percept

game playing: *some* response for *every* opponent moves

backward chaining: *some* rule such that *every* premise satisfied

### ✓ MONITORING AND REPLANNING

- Plan with Partially Ordered Plans algorithms
- Process plan, one step at a time
- Validate planned conditions against perceived reality
- “Failure” = preconditions of *remaining plan* not met

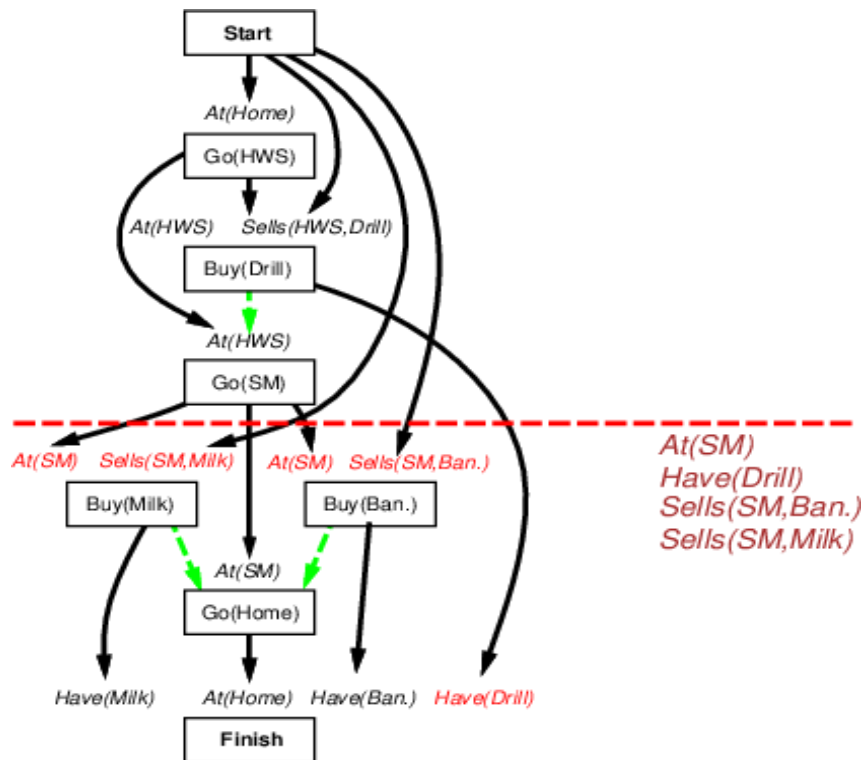
- Preconditions of remaining plan  
= all preconditions of remaining steps not achieved by remaining steps  
= all causal links ***crossing*** current time point
- Run Partially Ordered Plans algorithms again

Resume Partially Ordered Plans to achieve open conditions from current state IPEM

keep updating *Start* to match current state

links from actions replaced by links from *Start* when done

**Example:**



continuous planning proposes to replace the classic predefined and regular planning occasions with a continuous implementation of the planning in rapid parallel cycles. Planning is no longer triggered by a given date in the calendar, but by internal and external events as they occur.

- The plan stays more up-to-date: since the plan is updated every time a change occurs in the internal or external environment, the latest version of the plan automatically includes all changes up to that point.
- The plan is more accurate: because the plan can be updated at any time to incorporate new information, organizations do not need to project themselves into the future quite so strongly, and can afford to have a more simple, less detailed plan. Instead, the plan will be quickly updated as projects evolve and more specific and detailed data become available. This means less “guesstimates” and more accurate data.
- Top management (including financial and business people) get involved in the planning process more regularly and frequently. This in turn allows planning to become a tool that leaders can use to think about changes and the impact these will have on the business.
- Because re-planning occurs much more frequently, there is an additional motivation to improve and streamline the process (and make it more easily repeatable) switching to continuous planning requires a massive change in mindset, at all levels of the organization. Specifically, all the people involved in one way or another in the planning process (which is most of any organization) need to shift the way they look at planning:
- From a static, time-bound and ritual exercise to a dynamic, open-ended process that reacts to changes in the internal and external environment. This means that the way they work with the planning will change: instead of something constant that provides a fixed goalpost, the plan becomes something that can change from day to day, and those changes must be taken into account in the daily execution of the plan.
- From something whose value is in the end document (the plan) to something whose value is in the process, specifically in the way the activity of planning helps understand problems and risks.
- From something they get involved in maybe twice a year, to something they must engage with perhaps on a daily basis.



**✓ Considerations for implementing continuous planning**

- 1. Choosing the level at which to implement continuous planning:**  
Continuous planning is seldom applied at all levels of the organization. Companies wishing to implement it should therefore look at their planning practices, and identify at which levels implementing continuous planning would generate the greatest benefits.
  - 2. Ensuring that stakeholders at all levels of the organization make the leap to a mindset of continuous planning:**  
Regardless of the level at which continuous planning is implemented, it is important that all levels be aware of the principles of incremental development, and that they understand what continuous planning can and cannot do. There is no point in implementing continuous planning for instance at the product level if the business level is still going to insist on having a 1-year plan (and vice versa).
  - 3. Understanding of key business drivers:**  
Not all the changes to the internal or external environment will have the same impact on the organization's plan. Since planning become a much more frequent activity, it is important that the organization understand what its key business drivers are, and how to build (and easily retrieve) information about these drivers to make the process more fluid.
-

