UNIT - IV

Planning

Classical Planning: Definition of Classical Planning, Algorithms for Planning with State-Space Search,Planning Graphs, other Classical Planning Approaches, Analysis of Planning approaches.

Planning and Acting in the Real World: Time, Schedules, and Resources, Hierarchical Planning,

Planning and Acting in Nondeterministic Domains, Multi agent Planning.

We have defined AI as the study of rational action, which means that **planning**—devising a plan of action to achieve one's goals—is a critical part of AI. We have seen two examples of planning agents so far: the search-based agent and problem-solving agent

- The problem-solving agent of can find sequences of actions that result in a goal state. But it deals with atomic representations of states and thus needs good domain-specific heuristics to perform well.
- The search based agent can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. For example, in the vaccum world, the simple action of moving a step forward had to be repeated for all cleaning the surface.

Classical Planning is one of the classic AI problems, it has been used as the basis for applications like controlling robots and having conversations. In classical planning, we use a factored representation one in which a state of the world is represented by a collection of variables called **PDDL(**the Planning Domain Definition Language), that allows us to express all 4 *Thing* actions with one action schema.

The PDDL describes the four things to define a search problem. The things are

(1)The initial state

(2)T he actions that are available in a state

(3)The result of applying an action, and

(4) The goal test.

Each **state** is represented as a conjunction of fluents that are ground, functionless atoms. For example $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$.

Example: The blocks world

suppose you have three cubical blocks, A, B, and C, and a robot arm that can pick up and move one block at a time. suppose they are initially arranged on a table like this:

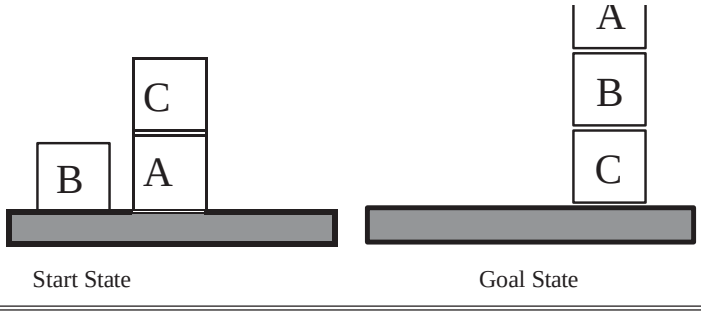


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

we need a precise representation of states, and also of actions that can be applied to states logic is often user for states, e.g. first guess you could represent a state like this:

- Block(x) means object x is a block
- On(x, y) means object x is on top of object y
- Clear(x) means there is nothing on top of object x

then we could model the initial state like this:

```
   C
 B A      initial state
-------

 On(B, Table) & On(A, Table) & On(C, A)
& Clear(B) & Clear(C)
& Block(A) & Block(B) & Block(C)
```

clear(x) seems a bit unusual, but is used because most planning languages (such as the one used in the textbook) don't support quantifiers in state descriptions

- i.e. if we had quantifiers we could say there is nothing on top of B with the sentence (for all blocks x).!on(x,B)
- but out planning language doesn't have quantifiers, so we instead use clear(B) to explicitly indicate that B can be picked up

now lets consider actions an action modifies a state, so we need to know at least two things for every action:

- whether or not the action can be applied to a given state; this is called the actions **pre-condition**
- how the state changes after the action is applied; this is called the action's **effect**

for example: (Note: the following steps doesn't belong to the above example.. this is another sample steps)

```
Move(b, x, y)
 Pre-condition: On(b,x) & Clear(b) & Clear(y)
      Effect: On(b,y) & Clear(x) & !On(b,x) & !Clear(y)
MoveToTable(b, x)
  Pre-condition: On(b, x) & Clear(b)
       Effect: On(b, Table) & Clear(x) & !On(b, x)
```

this is an **action schema**, i.e. a template that describes all possible move actions

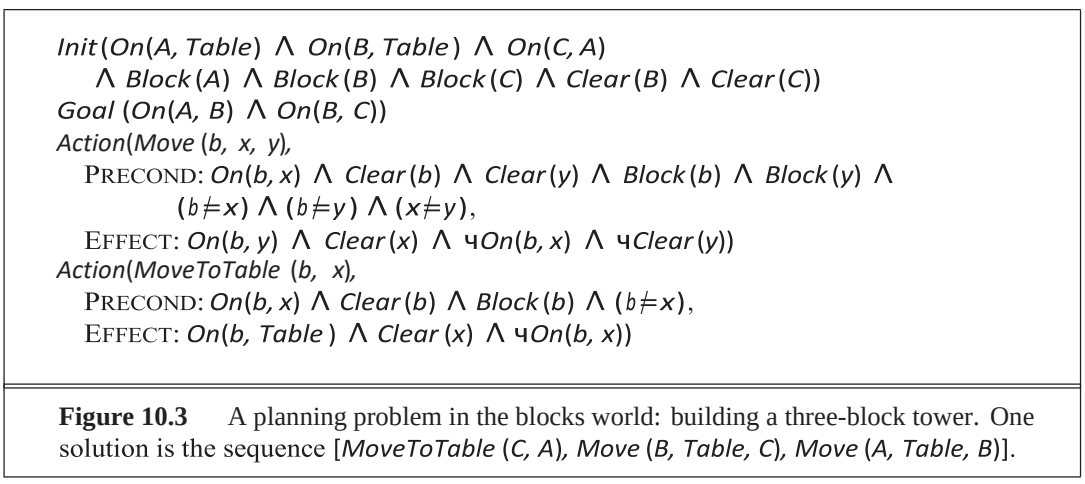


Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence [*MoveToTable* (*C, A*), *Move* (*B, Table, C*), *Move* (*A, Table, B*)].

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 followsfrom the fact that the number of states is finite. But if we add function symbols to the language, thenthe number of states becomes infinite, and PlanSAT becomes only semi decidable: 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.

Both PlanSAT and Bounded PlanSAT are in the complexity class PSPACE, a class that is larger (and hence more difficult) than NP and refers to problems that can be solved by a deterministic Turing machine witha polynomial amount of space. Even if we make some rather severe restrictions, the problems remain quite difficult.

Algorithms for Planning with State-Space Search

The most straight forward approach is to use state-space search. Because the descriptions of actions ina planning problem specify both preconditions and effects, it is possible to search in either direction: forward from the initial state or backward from the goal.

(a)Forward State-Space Search:

Example:1

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy ofAI: 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.

Suppose you want to buy a book with the ISBN 1234567890. i.e., own ISBN means our book number Here's how it would work:

1. **Action Schema**:
   - Buy(isbn) is the template action.
2. **Instantiating the Action**:
   - Replace isbn with 1234567890, creating a specific action: Buy(1234567890).
3. **Performing the Action**:
   - Execute Buy(1234567890), meaning you perform the buying process for the book with ISBN 1234567890.
4. **Effect**:
   - After the action is executed, you now own the book with ISBN 1234567890.
   - This new state of owning the book is represented as Own(1234567890).

In summary, performing an action from the schema Buy(isbn) results in a state where you own the book specified by the isbn, and this new ownership state is denoted as Own(isbn).

Example:2

> 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 theaverage 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 nodes.

Example Problem Description:

- **Air Cargo Problem**: Involves 10 airports, 5 planes at each airport, and 20 pieces of cargo at each airport.
- **Goal**: Move all cargo from airport A to airport B.
- **Simple Solution**: Load all 20 pieces of cargo into one plane at airport A, fly to airport B, and unload the cargo.

State Space Complexity:

- **State**: A specific configuration of all planes and cargo.
- **Branching Factor**: The number of possible actions from any given state.
- **Actions**:
  - Each of the 50 planes can fly to any of the 9 other airports.
  - Each of the 200 pieces of cargo can be either:
    - Unloaded (if it is currently loaded onto a plane).
    - Loaded into any plane at its current airport (if it is not already loaded).

Calculation of Actions:

- **Minimum Number of Actions**:
  - Occurs when all cargo is at airports with no planes.
  - Only actions are the possible flights of planes.
  - Each of the 50 planes can fly to 9 other airports: 50×9=45050 \times 9 = 45050×9=450 actions.
- **Maximum Number of Actions**:
  - Occurs when all cargo and planes are at the same airport.
  - Each plane can fly to 9 other airports: 50×9=45050 \times 9 = 45050×9=450 actions.
  - Each of the 200 pieces of cargo can be either loaded or unloaded into any of the 5 planes at that airport: 200×5=1000200 \times 5 = 1000200×5=1000 actions.

- Total maximum actions: 450+1000=1450450 + 1000 = 1450450+1000=1450 actions.

- **Average Number of Actions**:
  - Let's assume about 2000 possible actions per state as an average.

(b)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).
We start with the goal, which is a conjunction of literals forming a description of a set of states—for example, the goal ¬Poor A 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 flaunts in a domain, then there are 2n ground states (each fluent can be true or false), but 3n 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: Unloading Cargo

Goal: Deliver a specific piece of cargo (C2) to San Francisco International Airport (SFO):

- The goal is represented as At(C2, SFO), meaning cargo C2 should be at SFO.

Relevant Action: Unload(C2, p', SFO)

- This action involves unloading cargo C2 from an unspecified plane p′ at SFO.

conditions:

- In(C2, p'): Cargo C2 is in plane p′.
- At(p', SFO): Plane p′ is at SFO.
- Cargo(C2): C2 is a piece of cargo.
- Plane(p'): p′ is a plane.
- Airport(SFO): SFO is an airport.

Preconditions: In(C2,p′)∧At(p′,SFO)∧Cargo(C2)∧Plane(p′)∧Airport(SFO)

Effects: At(C2,SFO)∧¬In(C2,p′)

Standardizing Variable Names

- Variable names are standardized (e.g., changing p to p') to avoid confusion when the same action schema is used multiple times in a plan.

Example 2: Book example

- **Goal**: Own the book with ISBN 0136042597.

- **Action Schema**: A=Action(Buy(i), PRECOND: ISBN(i), EFFECT: Own(i))A = \text{Action(Buy(i), PRECOND: ISBN(i), EFFECT: Own(i))}A=Action(Buy(i), PRECOND: ISBN(i), EFFECT: Own(i))
- In forward search, the algorithm would enumerate all 10 billion possible Buy actions.
- In backward search:
    - Unify the goal Own(0136042597) with the effect Own(i'), yielding the substitution θ={i′/0136042597}\theta = \{i'/0136042597\}θ={i′/0136042597}.
    - Regress over the action with this substitution to get the predecessor state ISBN(0136042597).
    - If this state is part of the initial state, the goal is achieved.

Heuristics for planning:

Neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter3 that a heuristic function h(s) estimates the distance from a state s to the goal and that if we can derive an admissible heuristic for this distance—one that does not overestimate—then we can use A∗ search tofind optimal solutions. An admissible heuristic can be derived by defining a relaxed problem that iseasier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

By definition, there is no way to analyze an atomic state, and thus it it requires some ingenuity by a human analyst to define good domain-specific heuristics for search problems with atomic states. Example:

For example, consider an air cargo problem with 10 airports, 50 planes, and 200 pieces of cargo. Each plane can be at one of 10 airports and each package can be either in one of the planes or unloaded at
one of the airports. So there are $50^{10} \times 200^{50+10} \approx 10^{155}$ states. Now consider a particular problem in that domain in which it happens that all the packages are at just 5 of the airports,and all packages at a given airport have the same destination. Then a useful abstraction of the problem is to drop all the *At* fluents except for the ones involving one plane and one package at each of the 5 airports. Now there are only $5^{10} \times 5^{5+10} \approx 10^{17}$ states. A solution in this abstract state space will be shorter than a solution in the original space because the airports have less distance

Planning Graphs:

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 treeof 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 fromstate S0” immediately, just by looking it up.
- Of course, the **tree is of exponential size**, so this approach is impractical. 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 S0, but it can estimatehow 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 S0 for the initial state, consistingof nodes representing each fluent that holds in S0; then a level A0 consisting of nodes for each ground action that might be applicable in S0; then alternating levels Si followed by Ai; until we reach a termination condition .

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.

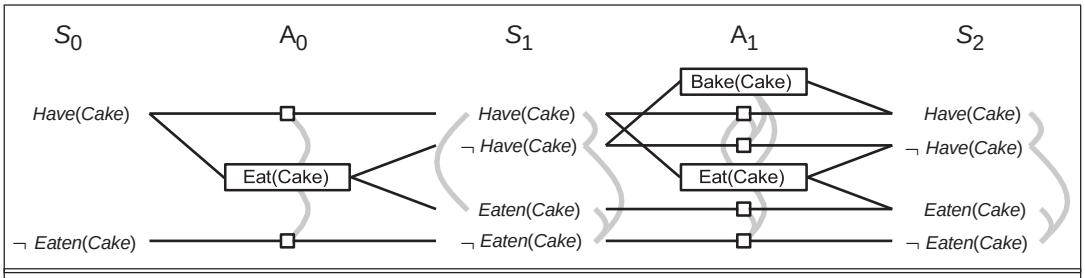


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because 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.

- 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}.
- 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.
- 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.

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:**

- **Definition:** This occurs when two actions produce effects that contradict each other. One action negates the effect of the other.
- **Example:** Consider the actions "Eat(Cake)" and "the persistence of Have(Cake)".
  - "Eat(Cake)" results in the effect of no longer having the cake, effectively negating "Have(Cake)".
  - If there is a plan or a state that involves maintaining "Have(Cake)" while also including the action "Eat(Cake)", the effects are inconsistent because "Eat(Cake)" directly negates "Have(Cake)".

2. **Interference:**
   - **Definition:** This occurs when one of the effects of an action negates a precondition required for another action to be performed.
   - **Example:** Again, using "Eat(Cake)" and "the persistence of Have(Cake)":
     - For the state "Have(Cake)" to persist, it must be true that the cake is not eaten.
     - "Eat(Cake)" negates "Have(Cake)" because once the cake is eaten, the precondition (having the cake) no longer holds.
     - Therefore, "Eat(Cake)" interferes with the persistence of "Have(Cake)" by negating its precondition.
3. **Competing Needs:**
   - **Definition:** This occurs when the preconditions of two actions are mutually exclusive, meaning they cannot both be true at the same time.
   - **Example:** Consider the actions "Bake(Cake)" and "Eat(Cake)":
     - "Bake(Cake)" has a precondition that there is no cake (since you bake a cake when there isn't one yet).
     - "Eat(Cake)" has a precondition that there is a cake to eat.
     - These preconditions are mutually exclusive because you cannot simultaneously be in a state where there is no cake (precondition for "Bake(Cake)") and there is a cake (precondition for "Eat(Cake)").
     - Therefore, these actions are mutex (mutually exclusive) because their preconditions cannot both be satisfied at the same time.

In summary:

- **Inconsistent effects** deal with contradictory outcomes of actions.
- **Interference** involves one action negating the necessary conditions for another.
- **Competing needs** concern mutually exclusive preconditions of actions.

OTHER CLASSICAL PLANNING APPROACHES

Currently the most popular and effective approaches to fully automated planning are:

- Forward state-space search with carefully crafted heuristics
- Search using a planning graph

These approaches are not the only ones tried in the 40-year history of automated planning. Figure 10.11 shows some of the top systems in the International Planning Competitions, which have been held every even year since 1998.

Year	Track	Winning Systems (approaches)
2008	Optimal	GAMER (model checking, bidirectional search)
2008	Satisficing	LAMA (fast downward search with FF heuristic)
2006	Optimal	SATPLAN, MAXPLAN (Boolean satisfiability)
2006	Satisficing	SGPLAN (forward search; partitions into independent subproblems)

2004	Optimal	SATPLAN (Boolean satisfiability)
2004	Satisficing	FAST DIAGONALLY DOWNWARD (forward search with causal graph)
2002	Automated	LPG (local search, planning graphs converted to CSPs)
2002	Hand-coded	TLPLAN (temporal action logic with control rules for forward search)
2000	Automated	FF (forward search)
2000	Hand-coded	TALPLANNER (temporal action logic with control rules for forward search)
1998	Automated	IPP (planning graphs); HSP (forward search)

Figure 10.11 Some of the top-performing systems in the International Planning Competition. Each year there are various tracks: "Optimal" means the planners must produce the shortest possible plan, while "Satisficing" means nonoptimal solutions are accepted. "Hand-coded" means domain-specific heuristics are allowed; "Automated" means they are not.

In this section we first describe the translation to a satisfiability problem and then describe other influential approaches: planning as first-order logical deduction; as constraint satisfaction; and as plan refinement.

(a)Classical planning as Boolean satisfiability

we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

1. **Propositionalize the Actions:**
   - **Definition:** This involves transforming the general action schemas into specific ground actions by substituting constants for each variable.
   - **Example:** If you have an action schema "Move(x, y)" and objects "A" and "B", you would create ground actions like "Move(A, B)", "Move(B, A)", etc.
   - **Purpose:** Although these ground actions are not directly part of the final translation, they are necessary for defining subsequent steps.
2. **Define the Initial State:**
   - **Definition:** Specify the truth value of each fluent (a proposition that can change over time) in the initial state.
   - **Example:** If the initial state of the problem specifies that "At(A, Location1)" is true and "At(B, Location1)" is not mentioned, then you assert "At(A, Location1)_0" as true and "¬At(B, Location1)_0" as true (since unmentioned fluents are assumed false).
   - **Purpose:** To establish the starting conditions from which the plan will begin.
3. **Propositionalize the Goal:**
   - **Definition:** Transform the goal into a propositional form by replacing variables with disjunctions over all possible constants.
   - **Example:** If the goal is "On(A, x) ∧ Block(x)" and you have objects "A", "B", and "C", this becomes: $(On(A, A) \land Block(A)) \lor (On(A, B) \land Block(B)) \lor (On(A, C) \land Block(C))$ **Purpose:** To express the goal in a way that can be evaluated using propositional logic.
4. **Add Successor-State Axioms:**
   - **Definition:** For each fluent FFF, add axioms that describe how the truth value of FFF changes from one time step to the next.
   - **Formula:** $F^{t+1} \Leftrightarrow ActionCausesF^t \lor (F^t \land \neg ActionCausesNotF^t)$,

where $ActionCausesF$ is a disjunction of all the ground actions that have F in their add list, and

$ActionCausesNotF$ is a disjunction of all the ground actions that have F in their delete list.

Purpose: To capture the dynamic behavior of the system in response to actions.

5. **Add Precondition Axioms:**
   - **Definition:** For each ground action AAA, add axioms that specify its preconditions must hold for the action to be executed.
   - **Formula:** For each ground action A, add the axiom $A^t \Rightarrow \text{PRE}(A)^t$, that is, if an action is taken at time t, then the preconditions must have been true.
   - **Explanation:** If action AAA is taken at time ttt, then its preconditions must have been true at ttt.
   - **Purpose:** To ensure that actions can only occur when their preconditions are satisfied.

6. **Add Action Exclusion Axioms:**
   - **Definition:** State that each action is distinct from every other action, ensuring that no two actions occur simultaneously.
   - **Formula:** If you have actions A and B, you add axioms like:

     ¬(At∧Bt)\neg (A_t \wedge B_t)¬(At∧Bt)

   - **Purpose:** To maintain the exclusivity of actions, ensuring only one action can occur at any given time.
7. **Translation for SATPLAN:**
   - **Definition:** Compile the problem into a propositional logic form that a SAT solver like SATPLAN can process to find a solution.
   - **Purpose:** To leverage SAT solvers' efficiency in solving complex planning problems by translating them into a suitable form.

These steps systematically convert a planning problem into a propositional logic representation, making it solvable by automated reasoning tools like SAT solvers.

(b) Planning as first-order logical deduction: Situation calculus

PDDL is a language that carefully balances the expressiveness of the language with the complexity of the algorithms that operate on it. But some problems remain difficult to express in PDDL. For example, we can't express the goal "move all the cargo from *A* to *B* regardless of how many pieces of cargo there are" in PDDL, but we can do it in first-order logic, using a universal quantifier. Likewise, first-order logic can concisely express global constraints such as "no more than four robots can be in the same place at the same time." PDDL can only say this with repetitious preconditions on every possible action that involves a move.

The propositional logic representation of planning problems also has limitations, such as the fact that the notion of time is tied directly to fluents. For example, $South^2$ means "the agent is facing south at time 2." With that representation, there is no way to say "the agent would be facing south at time 2 if it executed a right turn at time 1; otherwise it would be facing east." First-order logic lets us get around this limitation by replacing the notion of linear time with a notion of branching *situations*, using a representation called **situation calculus** that works like this:

Key Concepts:

1. **Situations:**
   - **Definition:** The initial state is called a situation. If sss is a situation and aaa is an action, then RESULT(s,a)\text{RESULT}(s, a)RESULT(s,a) is also a situation.
   - **Properties:** Two situations are the same if their start and actions are the same: (RESULT(s,a)=RESULT(s′,a′))⇔(s=s′∧a=a′)(\text{RESULT}(s, a) = \text{RESULT}(s', a')) \Leftrightarrow (s = s' \wedge a = a')(RESULT(s,a)=RESULT(s′,a′))⇔(s=s′∧a=a′)
2. **Fluents:**
   - **Definition:** Functions or relations that can vary from one situation to the next. The situation sss is always the last argument.
   - **Example:** At(x,l,s)\text{At}(x, l, s)At(x,l,s) Means object xxx is at location lll in situation sss.
3. **Possibility Axioms:**
   - **Definition:** Describe the preconditions of actions, indicating when an action can be taken.
   - **Example from the Wumpus world:**

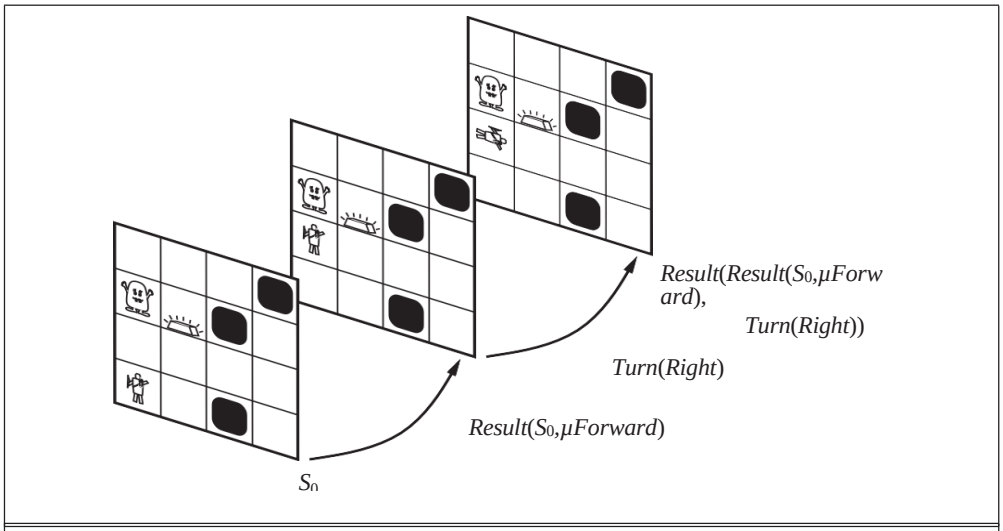


Figure 10.12 Situations as the results of actions in the wumpus world.

$$Alive(Agent, s) \wedge Have(Agent, Arrow, s) \Rightarrow Poss(Shoot, s)$$

   -
4. **Successor-State Axioms:**
   - **Definition:** Describe how fluents change as a result of actions.
   - **General Form:** that says what happens to the fluent, depending on what action is taken. This is similar to the approach we took for propositional logic. The axiom has the form

Action is possible ⇒

(*Fluent is true in result state* ⇔ *Action's effect made it true*

∨ *It was true before and action left it alone*) .

For example, the axiom for the relational fluent *Holding* says that the agent is holding some gold *g* after executing a possible action if and only if the action was a *Grab* of *g* or if the agent was already holding *g* and the action was not releasing it:

$Poss(a, s) \Rightarrow$

$$(Holding(Agent, g, Result(a, s)) \Leftrightarrow a = Grab(g) \vee (Holding(Agent, g, s) \wedge a \neq Release(g))) .$$

5. **Unique Action Axioms:**

- **Definition:** Ensure that actions are distinct from one another.
- **Example:** the agent can deduce that, for example, $a \neq Release(g)$. For each distinct pair of action names A_i and A_j we have an axiom that says the actions are different:

$$A_i(x, \ldots) \neq A_j(y, \ldots)$$

and for each action name A_i we have an axiom that says two uses of that action name are equal if and only if all their arguments are equal:

$$A_i(x_1, \ldots, x_n) = A_i(y_1, \ldots, y_n) \Leftrightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n .$$

Solution: A solution to a planning problem in situation calculus is a situation (a sequence of actions) that satisfies the goal.

Conclusion:

While PDDL offers a practical balance for many planning problems, its limitations in expressiveness make it less suitable for more complex goals and constraints. Situation calculus, with its greater expressiveness and ability to handle branching time, offers a more powerful framework but comes with significant practical challenges in terms of inference efficiency and the development of effective heuristics.

(c)Planning as constraint satisfaction:

The initial state can be used to prune what is not reachable and the goal to prune what is not useful. The CSP will be defined for a finite number of steps; the number of steps can be adjusted to find the shortest plan. One of the CSP methods can then be used to solve the CSP and thus find a plan.

To construct a CSP from a planning problem, first choose a fixed planning **horizon**, which is the number of time steps over which to plan. Suppose the horizon is k. The CSP has the following variables:

- a **state variable** for each feature and each time from 0 to k. If there are n features for a horizon of k, there are $n * (k+1)$ state variables. The domain of the state variable is the domain of the corresponding feature.
- an **action variable**, $Action_t$, for each time t in the range 0 to $k-1$. The domain of $Action_t$ is the set of all possible actions. The value of $Action_t$ represents the action that takes the agent from the state at time t to the state at time $t+1$.

There are several types of constraints:

- A **precondition constraint** between a state variable at time t and the variable $Action_t$ constrains what actions are legal at time t.
- An **effect constraint** between $Action_t$ and a state variable at time $t+1$ constrains the values of a state variable that is a direct effect of the action.
- A **frame constraint** among a state variable at time t, the variable $Action_t$, and the corresponding state variable at time $t+1$ specifies when the variable that does not change as a result of an action has the same value before and after the action.
- An **initial-state constraint** constrains a variable on the initial state (at time 0). The initial state is represented as a set of domain constraints on the state variables at time 0.
- A **goal constraint** constrains the final state to be a state that satisfies the achievement goal. These are domain constraints on the variables that appear in the goal.
- A **state constraint** is a constraint among variables at the same time step. These can include physical constraints on the state or can ensure that states that violate maintenance goals are forbidden. This is extra knowledge beyond the power of the feature-based or STRIPS representations of the action.

Example 6.14. *Consider finding a plan to get Sam coffee, where initially, Sam wants coffee but the robot does not have coffee. This can be represented as initial-state constraints: $SWC_0 = true$ and $RHC_0 = false$.*

With a planning horizon of 2, the goal is represented as the domain constraint $SWC_2 = false$, and there is no solution.

With a planning horizon of 3, the goal is represented as the domain constraint $SWC_3 = false$. This has many solutions, all with $RLoc_0 = cs$ (the robot has to start in the coffee shop), $Action_0 = puc$ (the robot has to pick up coffee initially), $Action_1 = mc$ (the robot has to move to the office), and $Action_2 = dc$ (the robot has to deliver coffee at time 2).

The CSP representation assumes a fixed planning horizon (i.e., a fixed number of steps). To find a plan over any number of steps, the algorithm can be run for a horizon of k=0, 11, 22,…until a solution is found. For the stochastic local search algorithm, it is possible to search multiple horizons at once, searching for all horizons, k from 0 to n, and allowing n to vary slowly.

(d)Planning as Refinement of Partially Ordered Plans

- Traditional planning approaches generate totally ordered plans, which means actions are strictly sequenced. However, in many situations, subproblems are independent and don't require a strict sequence.
- Partially ordered plans offer a more flexible approach by allowing some actions to be performed independently of others.
- Partially ordered plans are created by a *search through the space of plans* rather than through the state space.
- We start with the empty plan consisting of just the initial state and the goal, with no actions

in between, as in the top of Figure 10.13.

- The search procedure then looks for a **flaw** in the plan, and makes an addition to the plan to correct the flaw (or if no correction can be made, the search backtracks and tries something else).
- A flaw is anything that keeps the partial plan from being a solution. For example, one flaw in the empty plan is that no action achieves *At*(*Spare*, *Axle*).
- One way to correct the flaw is to insert into the plan the action *PutOn*(*Spare*, *Axle*).
- Of course that introduces some new flaws: the preconditions of the new action are not achieved.
- The search keeps adding to the plan (backtracking if necessary) until all flaws are resolved, as in the bottom of Figure 10.13.
- At every step, we make the **least commitment** possible to fix the flaw.
- For example, in adding the action *Remove* (*Spare*, *Trunk*) we need to commit to having it occur before *PutOn*(*Spare, Axle*), but we make no other commitment that places it before or after other actions. If there were a variable in the action schema that could be left unbound, we would do so.

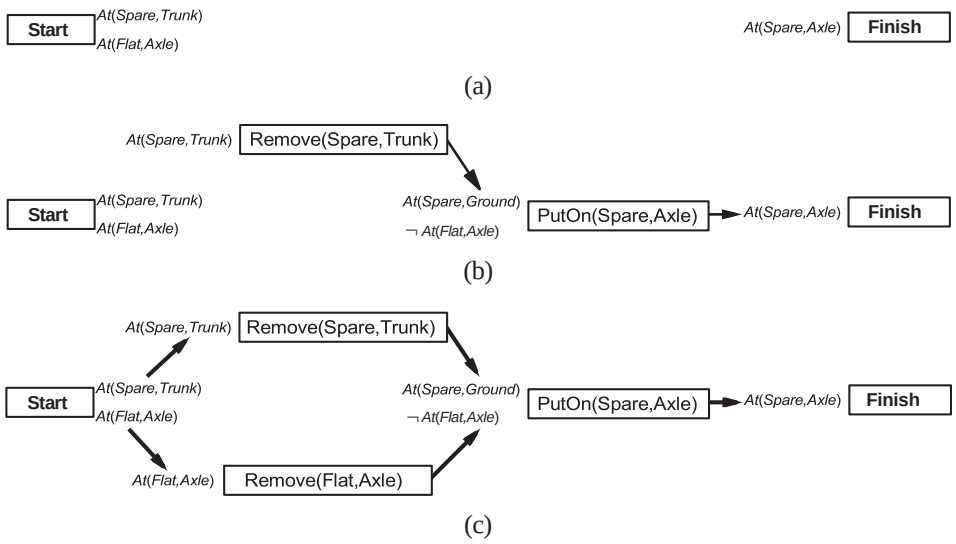


Figure 10.13 (a) the tire problem expressed as an empty plan. (b) an incomplete partially ordered plan for the tire problem. Boxes represent actions and arrows indicate that one action must occur before another. (c) a complete partially-ordered solution.

. Partial-order planning is also often used in domains where it is important for humans to understand the plans. Operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and verify that they are correct.

ANALYSIS OF PLANNING APPROACHES

Planning in artificial intelligence combines elements of search and logic. A planner can be seen either as a program that searches for a solution or one that proves the existence of a solution constructively.

Examples of Planning Systems

1. **GRAPHPLAN**:
    - Records mutexes (mutual exclusions) to highlight difficult interactions between actions.
    - These mutexes help identify where actions cannot occur simultaneously, aiding in managing the complexity of planning.
2. **SATPLAN**:
    - Represents mutex relations using the general CNF (Conjunctive Normal Form) rather than specific data structures.
    - Converts planning problems into Boolean satisfiability problems, allowing the use of SAT solvers to find solutions.

Serializable Subgoals in Practice

1. **Blocks World**:
    - Goal: Build a tower (e.g., A on B, B on C, C on the Table).
    - Subgoals are serializable bottom to top: once C is on the Table, it doesn't need to be moved again while achieving other subgoals.
    - This method avoids backtracking and can solve any problem in the blocks world efficiently, though it may not always find the shortest plan.
2. **Remote Agent Planner (NASA's Deep Space One)**:
    - The planner recognized that spacecraft commands have serializable propositions, simplifying control.
    - This approach enabled real-time control of the spacecraft by eliminating most of the search, previously thought impossible.

Future Directions in Planning

Planners like GRAPHPLAN, SATPLAN have advanced the field by improving performance, clarifying representation issues, and developing useful heuristics. However, there is a need for further progress in tackling larger problems. This may involve:

1. **Synthesis of Representations**:
    - Combining factored and propositional representations with first-order and hierarchical representations.
2. **Efficient Heuristics**:
    - Developing new heuristics that can efficiently handle larger and more complex planning problems.

Conclusion

Planning in AI is a dynamic field that leverages both search and logic to solve complex problems. By addressing combinatorial explosion, identifying independent subproblems, and recognizing serializable subgoals, planners can achieve significant performance improvements. Future advancements will likely require new techniques and representations to scale effectively to larger problems.

PLANNING AND ACTING IN THE REAL WORLD:

We have learnt the most basic concepts, representations, and algorithms forplanning. Planners that are are used in the real world for planning and scheduling the operations of spacecraft, factories, and military campaigns are more complex; they extend both the representation language and the way the planner interacts with the environment. This concepts extends the classical language for planning to talk about actions with durations and resource constraints, methods for constructing plans that are organized hierarchically. architectures that can handle uncertain environments and interleave deliberation with execution, and gives some examples of real-worldsystems.

Time, Schedules, And Resources:

- The **classical planning representation** talks about *what to do,* and in *what order,* but it 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 cannotbe 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 a *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.

Representing temporal and resource constraints

A typical **job-shop scheduling problem**, 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. . For simplicity, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

```
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 11.1 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.

Figure 11.1 shows a simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form [*AddEngine*, *AddWheels*, *Inspect*]. Then the *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.

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 shown in Figure 11.2.
- We can apply the **critical path method** (CPM) to this graph to determine the possible start and endtimes 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. We can see in Figure 11.2 that the whole plan will take 85 minutes, that each action in the top jobhas 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 $A \prec B$ means that *A* comes before *B*:

$$ES(Start) = 0$$
$$ES(B) = \max_{A \prec B} ES(A) + Duration(A)$$
$$LS(Finish) = ES(Finish)$$
$$LS(A) = \min_{B \succ A} LS(B) - Duration(A) .$$

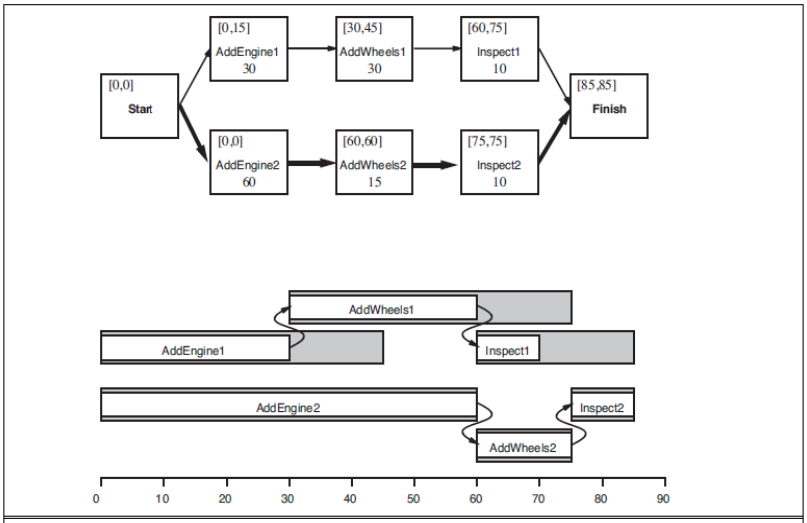


Figure 11.2 Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.1. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair $[ES, LS]$, displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Grey rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a gray rectangle indicates the slack.

When we introduce resource constraints, the resulting constraints on start and end times become more complicated. For example, the *AddEngine* actions, which begin at the same time in Figure 11.2, 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

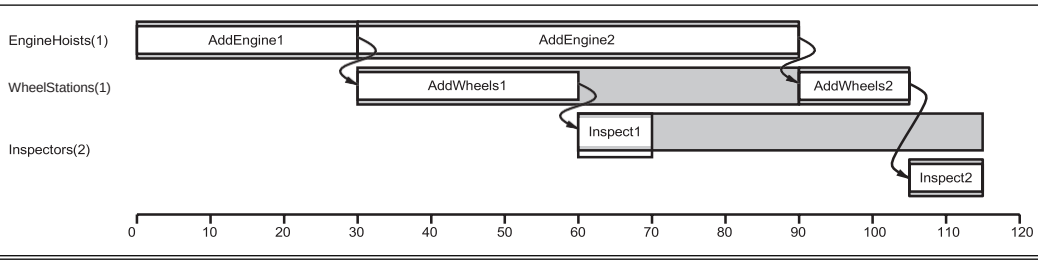


Figure 11.3 A solution to the job-shop scheduling problem from Figure 11.1, taking into account resource constraints. The left-hand margin lists the three reusable resources, and actions are shown aligned horizontally with the resources they use. There are two possi- ble schedules, depending on which assembly uses the engine hoist first; we've shown the shortest-duration solution, which takes 115 minutes.

.

Figure 11.3 shows the solution with the fastest completion time, 115 minutes. This is 30 minutes longer than the 85 minutes required for a schedule without resource constraints.Notice that there is no time at which both inspectors are required, so we can immediately move one of our two inspectors to a more productive position.

One simple called The minimum slack algorithm is a heuristic method used to solve scheduling problems with resource constraints. The steps of this heuristic, as you mentioned, are as follows:

1. **Identify Unscheduled Actions**:
    - At each iteration, identify the set of unscheduled actions that have all their predecessors scheduled.
2. **Select Action with Minimum Slack**:
    - From this set, select the action that has the least slack. Slack is the difference between the latest start time (LS) and the earliest start time (ES).
3. **Schedule the Selected Action**:
    - Schedule this selected action to start at its earliest possible start time (ES).
4. **Update Start and Finish Times**:
    - Update the ES and LS times for all affected actions. This involves recalculating the ES and LS times based on the newly scheduled action and its duration.
5. **Repeat**:
    - Repeat the process until all actions are scheduled.

It often works well in practice, but for our assembly problem it yields a 130–minute solution, not the 115–minute solution of Figure 11.3.

Hierarchical Planning:

The problem-solving and planning methods operate with a fixedset of atomic actions.For plans executed by the human brain, atomic actions are muscle activations. In veryround numbers, we have about 10^3 muscles to activate ,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—adetailed motor plan would contain around 10^{10} actions. This is a lot more than 1000.

To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction.

Example:
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 twoweeks; 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 actionsthat can be executed without deliberation to generate the required motor control sequences.

High-level actions

Hierarchical Task Network (HTN) planning is a method used in artificial intelligence for planning complex tasks. It involves breaking down a high-level task into simpler, more manageable sub-tasks (actions) until primitive actions (those that can be directly executed) are reached. Here's a detailed explanation of how HTN planning works and its benefits:

Key Concepts of HTN Planning

1. **Top-Level Action (Act)**:
   - The process starts with a single top-level action called Act.
   - The goal is to find an implementation of Act that achieves the desired goal.
2. **Refinement of Actions**:
   - Each high-level action (HLA) can be refined into a sequence of sub-actions.
   - This refinement process continues recursively until only primitive actions are left.
   - To prevent infinite recursion, a special refinement is provided: an empty list of steps with a precondition equal to the goal. This means that if the goal is already achieved, no further actions are needed.
3. **Algorithm for HTN Planning**:
   - Repeatedly choose an HLA in the current plan and replace it with one of its refinements.
   - Continue this process until the plan achieves the goal.
   - Breadth-first tree search can be used to explore plans, considering plans based on the depth of nesting of refinements rather than the number of primitive steps.
4. **Knowledge Encoding**:
   - HTN planning encodes a significant amount of domain knowledge in the refinements and their preconditions.
   - This allows HTN planners to generate large and complex plans efficiently, often with minimal search.
5. **Example of HTN Planning**:

plaintext Copy code

```
MakeBreakfast
├── CookOmelette
│   ├── CrackEggs
│   ├── BeatEggs
│   ├── PourEggsInPan
│   └── CookEggs
├── MakeToast
│   ├── GetBread
│   ├── PutBreadInToaster
│   ├── StartToaster
│   └── RemoveToast
└── BrewCoffee
    ├── GetCoffeeBeans
    ├── GrindCoffeeBeans
    ├── BoilWater
    └── BrewCoffeeInFrenchPress
```

Searching for primitive solutions in Hierarchical palnning

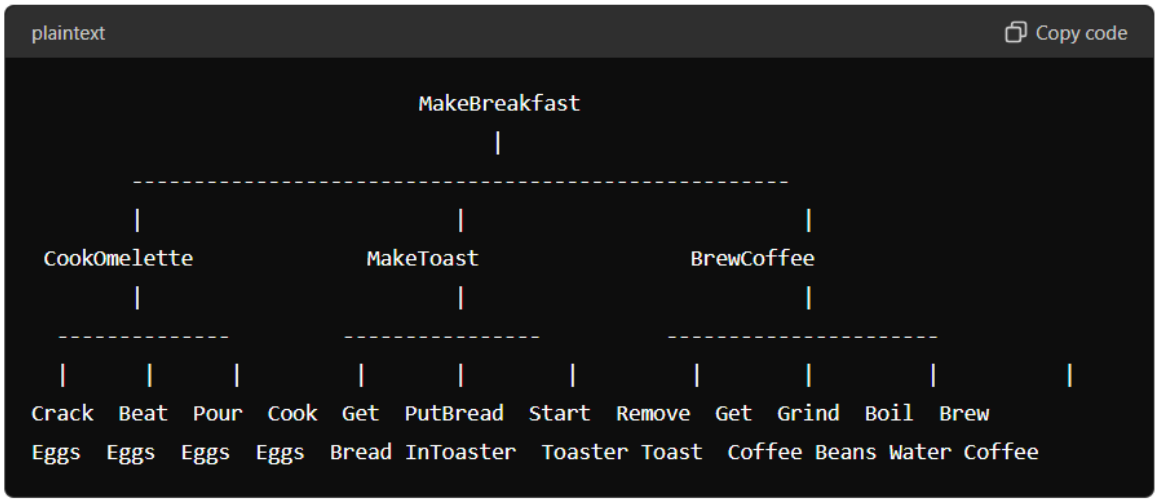
- The approach leads to a simple algorithm: repeatedly choose an HLA in the current plan and replace it with one of its refinements, until the plan achieves the goal.
- One possibleimplementation based on breadth-first tree search is shown in Figure 11.5. Plans are considered in order of depth of nesting of the refinements, rather than number of primitive steps.

Initial Plan:

- Start with the top-level action: MakeBreakfast

- **First Refinement**:
  - Refine MakeBreakfast into [CookOmelette, MakeToast, BrewCoffee]

- **Second Refinement**:
  - Refine CookOmelette into [CrackEggs, BeatEggs, PourEggsInPan, CookEggs]
  - Refine MakeToast into [GetBread, PutBreadInToaster, StartToaster, RemoveToast]
  - Refine BrewCoffee into [GetCoffeeBeans, GrindCoffeeBeans, BoilWater, BrewCoffeeInFrenchPress]



PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS:

we extend planning to handle partially observable, nondeterministic, and un-known environments. Extended search methods are :

(a) **Sensorless planning** (also known as **conformant planning**) for environments with no observations;

(b) **Contingency planning** for partially observable and nondeterministic environments;

(c) **Online planning** and **replanning** for unknown environments.

These agent's focus on **belief states**—the sets of possible physical states the agentmight be in—for unobservable and partially observable environments.

Example:
Consider this problem: given a chair and a table, the goal is to have them match—havethe 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:

Initial state: $Init(Object(Table) \land Object(Chair) \land Can(C_1) \land Can(C_2) \land InView(Table))$
Goal State: $Goal(Color(Chair, c) \land Color(Table, c))$

There are two actions:

(a)Removing the lid from a paint can and painting an object using the paint from an open can.

$Action(RemoveLid(can),$
PRECOND:$Can(can)$
EFFECT:$Open(can))$
$Action(Paint(x, can),$
PRECOND:$Object(x) \land Can(can) \land Color(can, c) \land Open(can)$
EFFECT:$Color(x, c))$

(2) To solve a partially observable problem, the agent will have to percepts , when it is executing the plan. The percept will be supplied by the agent's sensors whenit is actually acting, but when it is planning it will need a model of its sensors,this model was given by a function, PERCEPT(*s*). For planning, we augment PDDL with a new type of schema, the **percept schema**:

$Percept(Color(x, c),$
PRECOND:$Object(x) \land InView(x)$
$Percept(Color(can, c),$
PRECOND:$Can(can) \land InView(can) \land Open(can)$

The first schema says that whenever an object is in view, the agent will perceive the colorof the object The second schema says that if an open can is in view, then the agent perceives the color of the paint in the can. Of course, the agent will need an action that causes objects (one at a time) to come into view:

$Action(LookAt(x),$
PRECOND:$InView(y) \land (x \mathrel{/}= y)$
EFFECT:$InView(x) \land \neg InView(y))$

For a fully observable environment, we would have a *Percept* axiom with no preconditions for each fluent that is directly it percepts the colour and identify the colours of table and chair.

(a) **Sensorless planning**
- Now we search in a belief-state space
- Convert sensorless planning problem into a belief-state planning problem
- The belief state represented by a logical formula instead of explicitly enumerating a set of states
- Initial state can ignore the \(InView\) fluents
  - Agent has no sensors
- Can also take as given the unchanging facts
  - (Object(Table) and Object(Chair) and Can(C_1) and Can(C_2))
  - These hold in every belief state

- Agent doesn't know the color of cans or whether they are open or closed, it knows cans have colors.

$\forall x \exists c \; Color(x, c)$.

- After Skolemizing,(Note: Skolemization is a transformation on first-order logic formulae, which removes all existential quantifiers from a formula.). we obtain the initial belief state:
  $b_0 = Color(x, C(x))$.
- We do not follow the **closed-world assumption** because the fluent is mentioned as only positive i.e., true statement.
- We switch to an **open-world assumption**
  - States contain both, positive and negative fluents
  - If a fluent doesn't appear, its value is unknown
  - A possible solution
  - $[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)]$.

(b) **Contingent planning**

Contingent planning involves generating plans with conditional branches based on percepts, suitable for environments with partial observability, non-determinism, or both. This type of planning requires the agent to create a plan that includes different actions depending on the information received from the environment.

Consider a painting problem where the agent must paint objects based on their colors, but it cannot initially see the colors and must use percepts to gather information. A possible contingent plan is:

Initial Perception Actions **and** Conditional Branching**:**

$$
\begin{array}{l}
[LookAt(Table), LookAt(Chair), \\
\quad \textbf{if } Color(Table, c) \wedge Color(Chair, c) \textbf{ then } NoOp \\
\quad\quad \textbf{else } [RemoveLid(Can_1), LookAt(Can_1), RemoveLid(Can_2), LookAt(Can_2), \\
\quad\quad\quad \textbf{if } Color(Table, c) \wedge Color(can, c) \textbf{ then } Paint(Chair, can) \\
\quad\quad\quad \textbf{else if } Color(Chair, c) \wedge Color(can, c) \textbf{ then } Paint(Table, can) \\
\quad\quad\quad \textbf{else } [Paint(Chair, Can_1), Paint(Table, Can_1)]]]
\end{array}
$$

(c)Online replanning:

- A robot might seem to perform a repetitive task
- Replanning requires **execution monitoring** in order to know when a new plan is required
- Useful when a contingency planning is continuosly replanning
- Some branches of a partially constructed contingent plan could just say *Replan*
- Amount of planning in advance and how much planning is left for later is a tradeoff.
- Replanning may also be required when the agent's model of the world is incorrect

Model for an action may have:

- **Missing precondition:** i.e. Agent may not know it needs a screwdriver to open a paint can
- **A missing effect:** i.e. Floor may get paint when painting an object

- **A missing state variable:** i.e. Amount of paint in a can, the amount needed can't be zero

Online agent has three levels to monitor the environment and checks for it before executing an action

- **Action monitoring**: before executing an action, agent verifies that all preconditions still hold
- **Plan monitoring**: before executing an action, agent verifies that remaining plan will still succeed
- **Goal monitoring**: before executing an action, agent checks to see if there is a better set of goals it could be trying to achieve

    - Schematic of action monitoring

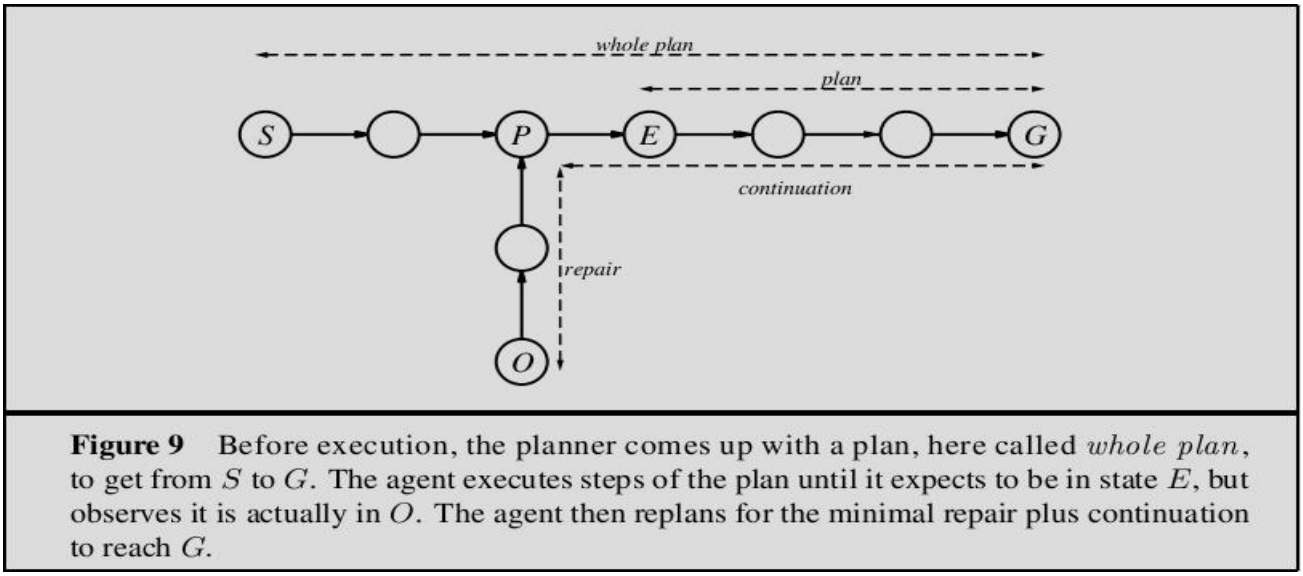


Figure 9 Before execution, the planner comes up with a plan, here called *whole plan*, to get from S to G. The agent executes steps of the plan until it expects to be in state E, but observes it is actually in O. The agent then replans for the minimal repair plus continuation to reach G.

$$[LookAt(Table), LookAt(Chair),$$
$$\textbf{if } Color(Table, c) \wedge Color(Chair, c) \textbf{ then } NoOp$$
$$\textbf{else } [RemoveLid(Can_1), LookAt(Can_1),$$
$$\textbf{if } Color(Table, c) \wedge Color(Can_1, c) \textbf{ then } Paint(Chair, Can_1)$$
$$\textbf{else } \text{REPLAN}]] \ .$$

Multi-Agent Planning in AI

- we have assumed that only one agent is doing the sensing, planning, and acting.
- When there are multiple agents in the environment, each agent faces a multiagent planning problem in which it tries to achieve its own goals with the help of others.
- An agent with multiple effectors that can operate concurrently—for example, a human who can type and speak at the same time—needs to do multi effector planning to manage each effector while handling positive and negative interactions among the effectors.
- When the effectors are physically decoupled into detached units—as in a fleet of delivery robots in a factory— multi effector planning becomes multibody planning.
- A multibody problem is still a "standard" single-agent problem as long as the relevant sensor information collected by each body can be pooled—either centrally or within each body—to form

a common estimate of the world state that then informs the execution of the overall plan; in this case, the multiple bodies act as a single body.

- When communication constraints make this impossible, we have what is sometimes called a decentralized planning problem; this is perhaps a misnomer, be- cause the planning phase is centralized but the execution phase is at least partially decoupled.
- Finally, some systems are a mixture of centralized and multiagent planning. For ex- ample, a delivery company may do centralized, offline planning for the routes of its trucks and planes each day, but leave some aspects open for autonomous decisions by drivers and pilots who can respond individually to traffic and weather situations. Also, the goals of the company and its employees are brought into alignment, to some extent, by the payment of **incentives** (salaries and bonuses)—a sure sign that this is a true multiagent system.
- The issues involved in multiagent planning can be divided roughly into two sets. The first, involves issues of representing and planning for multiple simultaneous actions; these issues occur in all settings from multieffector to multiagent planning. The second, involves issues of cooperation, coordination, and competition arising in true multiagent settings

(a)Planning with multiple simultaneous actions

For the time being, we will treat the multieffector, multibody, and multiagent settings in thesame way, labeling them generically as **multiactor** settings, using the generic term **actor** to cover effectors, bodies, and agents.

The goal of this section is to work out how to define transition models, correct plans, and efficient planning algorithms for the multiactor setting.A correct plan is one that, if executed by the actors, achieves the goal.

$Actors(A, B)$
$Init(At(A, LeftBaseline) \land At(B, RightNet) \land Approaching(Ball, RightBaseline)) \land Partner(A, B) \land Partner(B, A)$
$Goal(Returned(Ball) \land (At(a, RightNet) \lor At(a, LeftNet))$
$Action(Hit(actor, Ball),$
PRECOND:$Approaching(Ball, loc) \land At(actor, loc)$
EFFECT:$Returned(Ball))$
$Action(Go(actor, to),$
PRECOND:$At(actor, loc) \land to \neq loc,$
EFFECT:$At(actor, to) \land \neg At(actor, loc))$

Figure 11.10 The doubles tennis problem. Two actors A and B are playing together and can be in one of four locations: *LeftBaseline* , *RightBaseline* , *LeftNet* , and *RightNet* . The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

(b)Planning with multiple agents: Cooperation and coordination

- Now let us consider the true multiagent setting in which each agent makes its own plan.—each agent simply computes the joint solution and executes its own part of that solution.
- A first pass at a multiactor definition might look like Figure 11.10. With this definition, it is easy to see that the following joint plan plan works:

PLAN 1:

A : [Go(A, RightBaseline), Hit(A, Ball)]
B : [NoOp(B), NoOp(B)] .

Problems arise, however, when a plan has both agents hitting the ball at the same time. For example, the Hit action could be described as follows:

Action(Hit(a, Ball),
CONCURRENT:b /= a ⇒ ¬Hit(b, Ball)
PRECOND:Approaching (Ball, loc) ∧ At(a, loc)
EFFECT:Returned (Ball)) .

In other words, the Hit action has its stated effect only if no other Hit action by another agent occurs at the same time

PLAN 2:
A : [Go(A, LeftNet), NoOp(A)]
B : [Go(B, RightBaseline), Hit(B, Ball)] .

If both agents can agree on either plan 1 or plan 2, the goal will be achieved. But if A chooses plan 2 and B chooses plan 1, then nobody will return the ball. Conversely, if A chooses 1 and B chooses 2, then they will both try to hit the ball.