Unit IV

Knowledge Representation

It shows how to represent diverse facts about the real world in a form that can be used to reason and solve problems.

Knowledge Representation addresses the question of what content to put into such an agent's knowledge base—how to represent facts about the world. We can use first-order logic as the representation language. There are different representation formalisms such as hierarchical task networks for reasoning about plans, Bayesian networks for reasoning with uncertainty, Markov models for reasoning over time, and deep neural networks for reasoning about images, sounds, and other data.

Ontological Engineering:

In "toy" domains, the choice of representation is not that important; many choices will work. Complex domains such as shopping on the Internet or driving a car in traffic require more general and flexible representations concentrating on general concepts—such as Events, Time, Physical Objects, and Beliefs — that occur in many different domains. Representing these abstract concepts is sometimes called *ontological engineering*.

Upper ontology:

The general framework of concepts is called an upper ontology because of the convention of drawing Upper ontology graphs with the general concepts at the top and the more specific concepts below them, as in below figure.

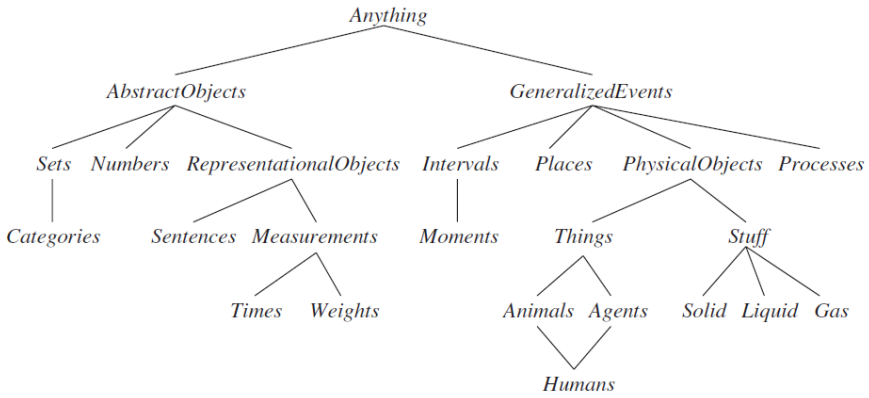


Figure 10.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint—a human is both an animal and an agent. We will see in Section 10.3.2 why physical objects come under generalized events.

Two major characteristics of general-purpose ontologies distinguish them from collections of special-purpose ontologies:
1. A general-purpose ontology should be applicable in more or less any special-purpose domain (with the addition of domain-specific axioms). This means that no representational issue can be finessed or swept under the carpet.
2. In any sufficiently demanding domain, different areas of knowledge must be unified, because reasoning and problem solving could involve several areas simultaneously. A robot circuit-repair system, for instance, needs to reason about circuits in terms of electrical connectivity and physical layout, and about time, both for circuit timing analysis and estimating labor costs. The sentences describing time therefore must be capable of being combined with those describing spatial layout and must work equally well for nanoseconds and minutes and for angstroms and meters.

Categories and Objects:

What are Categories and why they are important?

The organization of objects into categories is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, much reasoning takes place at the level of categories. For example, a shopper would normally have the goal of buying a basketball, rather than a particular basketball such as BB9.

Categories also serve to make predictions about objects once they are classified. One infers the presence of certain objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects. For example, from its green and yellow mottled skin, one-foot diameter, ovoid shape, red flesh, black seeds, and presence in the fruit aisle, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

How to represent Categories?

There are two choices for representing categories in first-order logic: **predicates and objects.** That is, we can use the predicate Basketball(b), or we can reify the category as an object, Basketballs. Turning a proposition into an object is called "**reification**". We could then say Member(b,Basketballs), which we will abbreviate as $b \in Basketballs$, to say that b is a member of the category of basketballs. We say Subset(Basketballs,Balls), abbreviated as $Basketballs \subset Balls$, to say that Basketballs is a subcategory of Balls. We can use subcategory, subclass, and subset interchangeably.

Inheritance:
Categories organize knowledge through inheritance. If we say that all instances of the category Food are edible, and if we assert that Fruit is a subclass of Food and Apples is a subclass of Fruit, then we can infer that every apple is edible. We say that the individual apples inherit the property of edibility, in this case from their membership in the Food category.

Taxonomic hierarchy:

Subclass relations organize categories into a taxonomic hierarchy or taxonomy.
Taxonomies have been used explicitly for centuries in technical fields. The largest such taxonomy organizes about 10 million living and extinct species, many of them beetles,2 into a single hierarchy; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system; and tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products.

First-order logic as a representation language for Categories:

First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members. Here are some example facts:
• An object is a member of a category.
$BB9 \in Basketballs$
• A category is a subclass of another category.
$Basketballs \subset Balls$
• All members of a category have some properties.
$(x \in Basketballs) \Rightarrow Spherical(x)$
• Members of a category can be recognized by some properties.
$Orange(x) \wedge Round(x) \wedge Diameter(x)=9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$
• A category as a whole has some properties.
$Dogs \in DomesticatedSpecies$

Although subclass and member relations are the most important ones for categories, we also want to be able to state relations between categories that are not subclasses of each other. For example, if we just say that Undergraduates and GraduateStudents are subclasses of Students, then we have not said that an undergraduate cannot also be a graduate student.

Disjoint categories:
We say that two or more categories are disjoint if they have no members in common.
Example: Disjoint({Animals,Vegetables})

Exhaustive decomposition:

We may also want to say that the classes undergrad and graduate student form an exhaustive
decomposition of university students.
Example: ExhaustiveDecomposition({Americans,Canadians,Mexicans},
NorthAmericans)

Partition:
A exhaustive decomposition of disjoint sets is known as a partition.
Example: Partition({Animals,Plants,Fungi,Protista,Monera},
LivingThings) .
(Note that the ExhaustiveDecomposition of NorthAmericans is not a Partition, because some
people have dual citizenship.)

The three predicates are defined as follows:
$Disjoint(s) \Leftrightarrow (\forall c1,c2\ c1 \in s \wedge c2 \in s \wedge c1 \neq c2 \Rightarrow Intersection(c1,c2)=\{\ \})$
$ExhaustiveDecomposition(s,c) \Leftrightarrow (\forall i\ i \in c \Leftrightarrow \exists c2\ c2 \in s \wedge i \in c2)$
$Partition(s,c) \Leftrightarrow Disjoint(s) \wedge ExhaustiveDecomposition(s,c)$.

Physical composition:

The idea that one object can be part of another is a familiar one. One's nose is part of one's head, Romania is part of Europe, and this chapter is part of this book. We use the general PartOf relation to say that one thing is part of another. Objects can be grouped into PartOf hierarchies, reminiscent of the Subset hierarchy:

PartOf (Bucharest,Romania)
PartOf (Romania,EasternEurope)
PartOf (EasternEurope,Europe)
PartOf (Europe,Earth) .
The PartOf relation is transitive and reflexive; that is,
$PartOf(x,y) \wedge PartOf(y, z) \Rightarrow PartOf(x, z)$
$PartOf(x,x)$.
Therefore, we can conclude PartOf (Bucharest,Earth).

Composite object:
Categories of composite objects are often characterized by structural relations among parts. For example, a biped is an object with exactly two legs attached to a body.

Measurements:
In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called "**measures**". Ordinary quantitative measures are quite easy to represent. We imagine that the universe includes abstract "measure objects," such as the length that is the length of

this line segment:|_____|. We can call this length 1.5 inches or 3.81 centimeters. Thus, the same length has different names in our language. We represent the length with a "**units function**" that takes a number as argument.

Natural Kind Categories:
Some categories have strict definitions: an object is a triangle if and only if it is a polygon with three sides. On the other hand, most categories in the real world have no clear-cut definition; these are called natural kind categories.

For example, tomatoes tend to be a dull scarlet; roughly spherical; with an indentation at the top where the stem was; about two to four inches in diameter; with a thin but tough skin; and with flesh, seeds, and juice inside. However, there is variation: some tomatoes are yellow or orange, unripe tomatoes are green, some are smaller or larger than average, and cherry tomatoes are uniformly small. Rather than having a complete definition of tomatoes, we have a set of features that serves to identify objects that are clearly typical tomatoes, but might not definitively identify other objects.

Objects: Things vesus stuff:

The real world can be seen as consisting of primitive objects (e.g., atomic particles) and composite objects built from them. By reasoning at the level of large objects such as apples and cars, we can overcome the complexity involved in dealing with vast numbers of primitive objects individually. There is, however, a significant portion of reality that seems to defy any obvious ***individuation***—division into distinct objects. We give this portion the generic name stuff. For example, suppose I have some butter and an aardvark in front of me. I can say there is one aardvark, but there is no obvious number of “butter-objects,” because any part of a butter-object is also a butter-object, at least until we get to very small parts indeed. This is the major distinction between stuff and things. If we cut an aardvark in half, we do not get two aardvarks (unfortunately).

Count nouns versus Mass nouns:
Linguists distinguish between count nouns, such as aardvarks, holes, and theorems, and mass nouns, such as butter, water, and energy.

Intrinsic versus Extrinsic properties:
Some properties are intrinsic: they belong to the very substance of the object, rather than to the object as a whole. When you cut an instance of stuff in half, the two pieces retain the intrinsic properties—things like density, boiling point, flavor, color, ownership, and so on. A category of objects that includes in
its definition only intrinsic properties is then a substance, or mass noun.

On the other hand, their extrinsic properties—weight, length, shape, and so on—are not retained under subdivision. A class that includes any extrinsic properties in its

definition is a count noun. Stuff and Thing are the most general substance and object categories, respectively.

3. Events:

Actions are things that happen, such as Shoot_t ; and fluents are aspects of the world that change, such as HaveArrow_t . In propositional logic, both can be represented as propositions, and successor-state axioms can be used to say that a fluent will be true at time t +1 if the action at time t caused it to be true, or if it was already true at time t and the action did not cause it to be false.

That was for a world in which actions are discrete, instantaneous, happen one at a time, and have no variation in how they are performed (that is, there is only one kind of Shoot action, there is no distinction between shooting quickly, slowly, nervously, etc.). But as we move from simplistic domains to the real world, there is a much richer range of actions or events to deal with. Consider a continuous action, such as filling a bathtub. A successor-state axiom can say that the tub is empty before the action and full when the action is done, but it can't talk about what happens during the action. It also can't easily describe two actions happening at the same time—such as brushing one's teeth while waiting for the tub to fill. To handle such cases an approach known as "event calculus" is used.

The objects of event calculus are events, fluents, and time points.

At(Shankar,Berkeley)

is a fluent: an object that refers to the fact of Shankar being in Berkeley. The event E1 of Shankar flying from San Francisco to Washington, D.C., is described as

$E1 \in Flyings \wedge Flyer(E1,Shankar) \wedge Origin(E1,SF) \wedge Destination(E1,DC)$.

where Flyings is the category of all flying events. By reifying events we make it possible to add any amount of arbitrary information about them. For example, we can say that Shankar's flight was bumpy with Bumpy(E1).

To assert that a fluent is actually true starting at some point in time t1 and continuing to time t2, we use the predicate T, as in T(At(Shankar,Berkeley), t1, t2).

The complete set of predicates for one version of the event calculus4 is:

T(f , t1, t2) Fluent f is true for all times between t1 and t2
Happens(e, t1, t2) Event e starts at time t1 and ends at t2
Initiates(e, f , t) Event e causes fluent f to become true at time t
Terminates(e, f , t) Event e causes fluent f to cease to be true at time t

Initiated(f , t1, t2) Fluent f become true at some point between t1 and t2
Terminated(f , t1, t2) Fluent f cease to be true at some point between t1 and t2
t1 < t2 Time point t1 occurs before time t2
We can describe the effects of a flying event:
E = Flyings(a,here, there)andHappens(E, t1, t2) ⇒
Terminates(E,At(a,here), t1) ∧ Initiates(E,At(a, there), t2)

Time:

Event calculus opens us up to the possibility of talking about time points and time intervals. There are two kinds of time intervals: moments and extended intervals. The distinction is that only moments have zero duration:

Partition({Moments,ExtendedIntervals}, Intervals)
i ∈ Moments ⇔ Duration(i)=Seconds(0) .

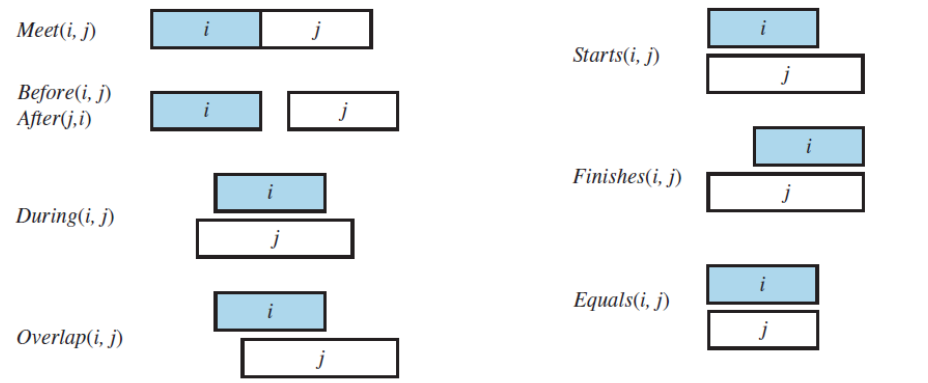


Figure 10.2 Predicates on time intervals.

Fl ue

Fluents and objects:

Physical objects can be viewed as generalized events, in the sense that a physical object is a chunk of space–time. For example, USA can be thought of as an event that began in 1776 as a union of 13 states and is still in progress today as a union of 50. We can describe the changing properties of USA using state fluents, such as Population(USA). A property of USA that changes every four or eight years, barring mishaps, is its president. One might propose that President(USA) is a logical term that denotes a different object at different times.

Unfortunately, this is not possible, because a term denotes exactly one object in a given model structure. (The term President(USA, t) can denote different objects,

depending on the value of t, but our ontology keeps time indices separate from fluents.) The only possibility is that President(USA) denotes a single object that consists of different people at different times. It is the object that is GeorgeWashington from 1789 to 1797, John Adams from 1797 to 1801, and so on, as in Figure 10.3. To say that George Washington was president throughout 1790, we can write

T(Equals(President(USA),GeorgeWashington),Begin(AD1790),End(AD1790)) .

4.

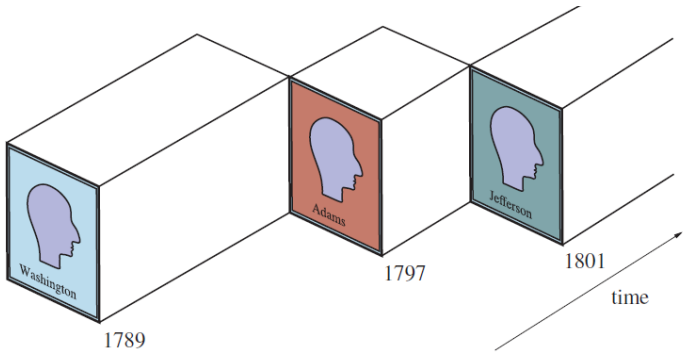


Figure 10.3 A schematic view of the object *President*(*USA*) for the early years.

Mental Objects and Modal Logic

The agents have beliefs and can deduce new beliefs. They may not have any knowledge about beliefs or about deduction. Knowledge about one's own knowledge and reasoning processes is useful for controlling inference. What we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects.

For example, suppose Alice asks "what is the square root of 1764" and Bob replies "I don't know." If Alice insists "think harder," Bob should realize that with some more thought, this question can in fact be answered.

Propositional attitude:
We begin with the propositional attitudes that an agent can have toward mental objects: attitudes such as Believes, Knows, Wants, and Informs. The difficulty is that these attitudes do not behave like "normal" predicates. For example, suppose we try to assert that Lois knows that Superman can fly:

Knows(Lois,CanFly(Superman)) .

One minor issue with this is that we normally think of CanFly(Superman) as a sentence, but here it appears as a term. That issue can be patched up by reifying CanFly(Superman); making it a fluent.

Modal logic:
Modal logic is designed to address this problem. Regular logic is concerned with a single modality, the modality of truth, allowing us to express "P is true" or "P is false." Modal logic includes special modal operators that take sentences (rather than terms) as arguments. For example, "A knows P" is represented with the notation KAP, where K is the modal operator for knowledge. It takes two arguments, an agent (written as the subscript) and a sentence. The syntax of modal logic is the same as first-order logic, except that sentences can also be formed with modal operators.

Semantics of Modal Logic:
The semantics of modal logic is more complicated. In first-order logic amodel contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and the possibility that it isn't.
Therefore, we will need a more complicated model, one that consists of a collection of possible worlds rather than just one true world. The worlds are connected in a graph by accessibility relations, one relation for each modal operator. We say that world w1 is accessible Accessibility relation from world w0 with respect to the modal operator K_A if everything in w1 is consistent with what A knows in w0.

If an agent knows P and knows that P implies Q, then the agent knows Q:
$(K_a\ P \wedge K_a(P \Rightarrow Q)) \Rightarrow K_a\ Q$.

If you know something, it must be true, and we have the axiom:
$K_a\ P \Rightarrow P$.
Furthermore, logical agents (but not all people) are able to introspect on their own knowledge.
If they know something, then they know that they know it:
K_a P ⇒ K_a(K_a P) .

Other Modal Logics:

Many modal logics have been proposed, for different modalities besides knowledge. One proposal is to add modal operators for possibility and necessity: it is possibly true that one of the authors of this book is sitting down right now, and it is necessarily true that 2+2 = 4. Some logicians favor modalities related to time. In linear temporal logic, we add the following modal operators:
• X P: "P will be true in the next time step"
• F P: "P will eventually (Finally) be true in some future time step"
• G P: "P is always (Globally) true"

• P U Q: “P remains true until Q occurs”

5.Reasoning Systems for Categories

Categories are the primary building blocks of large-scale knowledge representation schemes. We need systems specially designed for organizing and reasoning with categories. There are two closely related families of systems: "semantic networks" and "description logic".

Semantic Networks:
Semantic networks provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties of an object on the basis of its category membership.

There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects. A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled links. For example, below Figure has aMemberOf link between Mary and FemalePersons, corresponding to the logical assertion Mary ∈ FemalePersons; similarly, the SisterOf link between Mary and John corresponds to the assertion SisterOf (Mary,John).

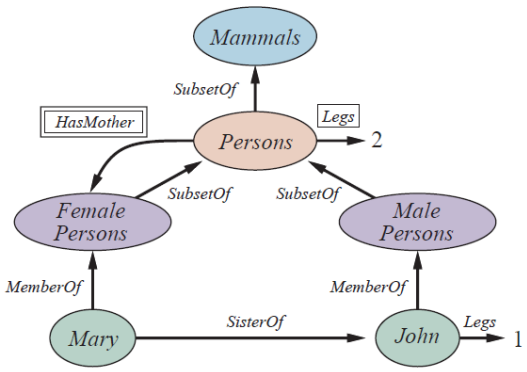


Figure 10.4 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

We can connect categories using SubsetOf links, and so on. It is such fun drawing bubbles and arrows that one can get carried away. For example, we know that persons have female persons as mothers, so can we draw a HasMother link from Persons to FemalePersons? The answer is no, because HasMother is a relation between a person and his or her mother, and categories do not have mothers.

For this reason, we have used a special notation—the double-boxed link—in above Figure .

This link asserts that ∀x x ∈ Persons ⇒ [∀y HasMother(x,y) ⇒ y ∈ FemalePersons] .
We might also want to assert that persons have two legs—that is,
∀x x ∈ Persons ⇒ Legs(x,2). As before, we need to be careful not to assert that a category has legs; the single-boxed link in above Figure is used to assert properties of every member of a category.

Inheritance becomes complicated when an object can belong to more than one category or when a category can be a subset of more than one other category; this is called multiple inheritance. In such cases, the inheritance algorithm might find two or more conflicting values answering the query. For this reason, multiple inheritance is banned in some object-oriented programming (OOP) languages, such as Java, that use inheritance in a class hierarchy. It is usually allowed in semantic networks.

An obvious drawback of semantic network notation, compared to first-order logic: the fact that links between bubbles represent only binary relations.

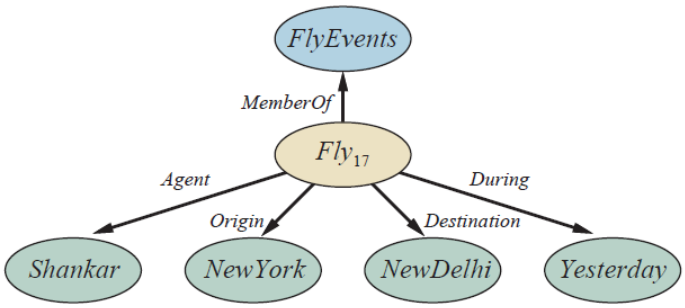


Figure 10.5 A fragment of a semantic network showing the representation of the logical assertion $Fly(Shankar, NewYork, NewDelhi, Yesterday)$.

For example, the sentence Fly(Shankar,NewYork,NewDelhi,Yesterday) cannot be asserted directly in a semantic network. Nonetheless, we can obtain the effect of n-ary assertions by reifying the proposition itself as an event belonging to an appropriate event category. Figure 10.5 shows the semantic network structure for this particular event. Notice that the restriction to binary relations forces the creation of a rich ontology of reified concepts.

Default values and overriding in Semantic Networks:

One of the most important aspects of semantic networks is their ability to represent default values for categories. Examining Figure 10.4 carefully, one notices that John has one Default value leg, despite the fact that he is a person and all persons have two legs.
In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this iscontradicted by more specific information. The default semantics is enforced naturally by the inheritance algorithm, because it follows links upwards from the object itself (John in this case) and stops as soon as it finds a value. We say that the default is overridden by the more specific value. Notice that we could also override the default number of legs by creating a category of OneLeggedPersons, a subset of Persons of which John is a member. We can retain a strictly logical semantics for the network if we say that the Legs assertion for Persons includes an exception for John:
$\forall x\ x \in \text{Persons} \wedge\ x \mathrel{!}= \text{John} \Rightarrow \text{Legs}(x,2)$.

Description logics:

Description logics provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.
The syntax of first-order logic is designed to make it easy to say things about objects. Description logics are notations that are designed to make it easier to describe definitions and properties of categories. Description logic systems evolved from semantic networks in response to pressure to formalize what the networks mean while retaining the emphasis on taxonomic structure as an organizing principle.
The principal inference tasks for description logics are subsumption (checking if one category is a subset of another by comparing their definitions) and classification (checking whether an object belongs to a category). Some systems also include consistency of a category definition—whether the membership criteria are logically satisfiable.

$$
\begin{aligned}
\textit{Concept} &\rightarrow \textbf{Thing} \mid \textit{ConceptName} \\
&\mid \textbf{And}(\textit{Concept}, \ldots) \\
&\mid \textbf{All}(\textit{RoleName}, \textit{Concept}) \\
&\mid \textbf{AtLeast}(\textit{Integer}, \textit{RoleName}) \\
&\mid \textbf{AtMost}(\textit{Integer}, \textit{RoleName}) \\
&\mid \textbf{Fills}(\textit{RoleName}, \textit{IndividualName}, \ldots) \\
&\mid \textbf{SameAs}(\textit{Path}, \textit{Path}) \\
&\mid \textbf{OneOf}(\textit{IndividualName}, \ldots) \\
\textit{Path} &\rightarrow [\textit{RoleName}, \ldots] \\
\textit{ConceptName} &\rightarrow \textit{Adult} \mid \textit{Female} \mid \textit{Male} \mid \ldots \\
\textit{RoleName} &\rightarrow \textit{Spouse} \mid \textit{Daughter} \mid \textit{Son} \mid \ldots
\end{aligned}
$$

Figure 10.6 The syntax of descriptions in a subset of the CLASSIC language.

The CLASSIC language (Borgida et al., 1989) is a typical description logic. The syntax of CLASSIC descriptions is shown in Figure 10.6.6 For example, to say that bachelors are unmarried adult males we would write

Bachelor = And(Unmarried,Adult,Male) .

The equivalent in first-order logic would be

$Bachelor(x) \Leftrightarrow Unmarried(x) \wedge Adult(x) \wedge Male(x)$.

Notice that the description logic has an algebra of operations on predicates, which of course we can't do in first-order logic. Any description in CLASSIC can be translated into an equivalent first-order sentence, but some descriptions are more straightforward in CLASSIC. For example, to describe the set of men with at least three sons who are all unemployed and married to doctors, and at most two daughters who are all professors in physics or math departments, we would use

And(Man,AtLeast(3,Son),AtMost(2,Daughter),
All(Son,And(Unemployed,Married,All(Spouse,Doctor))),
All(Daughter,And(Professor,Fills(Department,Physics,Math)))) .

6. Reasoning with Default Information

Consider a simple example of an assertion with default status: people have two legs. This default can be overridden by more specific information, such as that Long John Silver has one leg. The inheritance mechanism in semantic networks implements the overriding of defaults in a simple and natural way. There is a need to study defaults more generally, with a view toward understanding the semantics of defaults rather than just providing a procedural mechanism.

Circumscription and default logic:

Monotonicity requires all entailed sentences to remain entailed after new sentences are added to the KB. That is, if KB = A then $KB \wedge B = A$.

In general, humans often "jump to conclusions." For example, when one sees a car parked on the street, one is normally willing to believe that it has four wheels even though only three are visible. For most people, the possibility that the car does not have four wheels will not arise unless some new evidence presents itself. Thus, it seems that the four-wheel conclusion is reached by "**default**", in the absence of any reason to doubt it. If new evidence arrives—for example, if one sees the owner carrying a wheel and notices that the car is jacked up—then the conclusion can be retracted. This kind of reasoning is said to exhibit **nonmonotonicity**, because the set

of beliefs does not grow monotonically over time as new evidence arrives. Nonmonotonic logics have been devised with modified Nonmonotonic logic notions of truth and entailment in order to capture such behavior. There are two such logics that have been studied extensively: circumscription and default logic.

Circumscription: Circumscription can be seen as a more powerful and precise version of the closed-world assumption. The idea is to specify particular predicates that are assumed to be "as false as possible"—that is, false for every object except those for which they are known to be true.

For example, suppose we want to assert the default rule that birds fly. We would introduce a predicate, say Abnormal1(x), and write

$Bird(x) \land \neg Abnormal1(x) \Rightarrow Flies(x)$.

If we say that Abnormal1 is to be circumscribed, a circumscriptive reasoner is entitled to assume ¬Abnormal1(x) unless Abnormal1(x) is known to be true. This allows the conclusion Flies(Tweety) to be drawn from the premise Bird(Tweety), but the conclusion no longer holds if Abnormal1(Tweety) is asserted.

Default logic: Default logic is a formalism in which default rules can be written to generate contingent, nonmonotonic conclusions. A default rule looks like this:

Bird(x) : Flies(x)/Flies(x) .

This rule means that if Bird(x) is true, and if Flies(x) is consistent with the knowledge base, then Flies(x) may be concluded by default. In general, a default rule has the form

P : J1, . . . ,Jn/C

where P is called the prerequisite, C is the conclusion, and Ji are the justifications—if any one of them can be proven false, then the conclusion cannot be drawn. Any variable that appears in Ji or C must also appear in P.

Truth maintenance system:

Many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called **belief revision**. Truth maintenance systems, or TMSs, handle exactly these kinds of complications.

An efficient approach is the **justification-based truth maintenance system**, or **JTMS**. In a JTMS, each sentence in the knowledge base is annotated with a justification consisting of the set of sentences from which it was inferred.

An **assumption-based truth maintenance system, or ATMS**, makes this type of context switching between hypothetical worlds particularly efficient. In a JTMS, the maintenance of justifications allows you to move quickly from one state to another by making a few retractions and assertions, but at any time only one state is represented. An ATMS represents all the states that have ever been considered at the same time. Whereas a JTMS simply labels each sentence as being in or out, an ATMS keeps track, for each sentence, of which assumptions would cause the sentence to be true. In other words, each sentence has a label that consists of a set of assumption sets. The sentence is true just in those cases in which all the assumptions in one of the assumption sets are true.

Summary:

By delving into the details of how one represents a variety of knowledge, we hope we have given the reader a sense of how real knowledge bases are constructed and a feeling for the interesting philosophical issues that arise. The major points are as follows:

• Large-scale knowledge representation requires a general-purpose ontology to organize and tie together the various specific domains of knowledge.
• A general-purpose ontology needs to cover a wide variety of knowledge and should be capable, in principle, of handling any domain.
• Building a large, general-purpose ontology is a significant challenge that has yet to be fully realized, although current frameworks seem to be quite robust.
• We presented an upper ontology based on categories and the event calculus. We covered categories, subcategories, parts, structured objects, measurements, substances, events, time and space, change, and beliefs.
• Natural kinds cannot be defined completely in logic, but properties of natural kinds can be represented.
• Actions, events, and time can be represented with the event calculus. Such representations enable an agent to construct sequences of actions and make logical inferences about what will be true when these actions happen.
• Special-purpose representation systems, such as semantic networks and description logics, have been devised to help in organizing a hierarchy of categories. Inheritance is an important form of inference, allowing the properties of objects to be deduced from their membership in categories.

• The closed-world assumption, as implemented in logic programs, provides a simple

way to avoid having to specify lots of negative information. It is best interpreted as a default that can be overridden by additional information.
• Nonmonotonic logics, such as circumscription and default logic, are intended to capture default reasoning in general.
• Truth maintenance systems handle knowledge updates and revisions efficiently.
• It is difficult to construct large ontologies by hand; extracting knowledge from text makes the job easier.

Classical Planning

(PLAN II)

In which we see how an agent can take advantage of the structure of a problem to efficiently construct complex plans of action.

- By understanding these concepts and techniques, we can harness the power of AI planning to create intelligent systems that can solve complex problems and achieve goals in a variety of domains.

Planning in AI

Planning in Artificial Intelligence (AI) is a crucial process where AI systems devise a sequence of actions to achieve a specific goal. It's like creating a roadmap for an AI agent to navigate through a problem space, considering various possibilities and constraints.

Key Concepts in Planning:

- **State:** A representation of the current situation or configuration of the world.
- **Action:** An operation that can change the state of the world.
- **Goal:** The desired final state or condition to be achieved.
- **Plan:** A sequence of actions that, when executed, leads from the initial state to the goal state.

Planning Techniques:

- **Classical Planning:** Assumes a deterministic and fully observable environment. Techniques like forward and backward search, and Boolean satisfieability are used.
- **Heuristic Search Planning:** Employs heuristic functions to guide the search process, making it more efficient.
- **Hierarchical planning: It** is a powerful technique in artificial intelligence that involves breaking down complex tasks into smaller, more manageable subtasks, organized into different levels or hierarchies.

Applications of Planning in AI:

- **Robotics:** Planning robot movements, task sequences, and obstacle avoidance.
- **Autonomous Vehicles:** Planning driving routes, traffic navigation, and decision-making in complex scenarios.
- **Game AI:** Creating intelligent game agents that can plan strategies and make tactical decisions.
- **Logistics and Supply Chain Management:** Optimizing transportation routes, inventory management, and resource allocation.
- **Healthcare:** Planning treatment plans, scheduling appointments, and resource allocation in hospitals.

Challenges in Planning:

- **Scalability:** Planning problems can become computationally expensive as the complexity of the problem space increases.
- **Uncertainty:** Real-world environments are often uncertain, making it difficult to plan with perfect information.
- ***Dynamic Environments:*** *Environments can change rapidly, requiring replanning and adaptation.*

Planning a course of action is a key requirement for an intelligent agent. The right representation for actions and states and the right algorithms can make this easier. **A factored representation splits up each state into a fixed set of variables or attributes, each of which can have a value.** A general factored representation language for planning problems that can naturally and succinctly represent a wide variety of domains, can efficiently scale up to large *problems, and does not require ad hoc heuristics for a new domain.*

Classical planning:

Classical planning is defined as the task of finding a sequence of actions to accomplish a goal in a discrete, deterministic, static, fully observable environment. There are two approaches to this task: the problem-solving agent and the hybrid propositional logical agent. Both share two limitations.

- First, they both require ad hoc heuristics for each new domain: a heuristic evaluation function for search, and hand-written code for the hybrid wumpus agent.
- Second, they both need to explicitly represent an exponentially large state space.

For example, in the propositional logic model of the wumpus world, the axiom for moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations.

In response to these limitations, planning researchers have invested in a factored representation using a family of languages called PDDL, the Planning Domain Definition Language. In PDDL, a state is represented as a conjunction of ground atomic fluents. Here “ground” means no variables, “fluent” means an aspect of the world that changes over time, and “ground atomic” means there is a single predicate, and if there are any arguments, they must be constants.

For example, Poor ∧ Unknown might represent the state of a hapless agent, and At(Truck1,Melbourne) ∧ At(Truck2,Sydney) could represent a state in a package delivery problem. PDDL uses database semantics: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that Truck1 and Truck2 are distinct.

An **action schema** represents a family of ground actions. 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) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
EFFECT:¬At(p, from) ∧ At(p, to))

The schema consists of the action name, a list of all the variables used in the schema, a precondition and an effect. The precondition and the effect are each conjunctions of literals (positive or negated atomic sentences).

We can choose constants to instantiate the variables, yielding a ground (variable-free) action:
Action(Fly(P1,SFO,JFK),
PRECOND:At(P1,SFO) ∧ Plane(P1) ∧ Airport(SFO) ∧ Airport(JFK)
EFFECT:Â¬At(P1,SFO) ∧ At(P1,JFK))

A ground action a is applicable in state s if s entails the precondition of a; that is, if every positive literal in the precondition is in s and every negated literal is not. The result of executing applicable action a in state s is defined as a state s′ 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)):

RESULT(s, a) = (s−DEL(a)) ∪ ADD(a) .

For example, with the action Fly(P1,SFO,JFK), we would remove the fluent At(P1,SFO) and add the fluent At(P1,JFK).
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. The initial state is a conjunction of ground fluents.
The goal (introduced with Goal) is just like a precondition: a conjunction of literals (positive or negative) that may contain variables. For example, the goal At(C1,SFO) ∧ ¬At(C2,SFO) ∧ At(p,SFO), refers to any state in which cargo C1 is at SFO but C2 is not, and in which there is a plane at SFO.

Example domain: Air cargo transport

Figure 11.1 shows an air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: Load, Unload, and Fly. The actions affect two predicates: In(c, p) means that cargo c is inside plane p, and At(x,a) means that object x (either plane or cargo) is at airport a. Note that some care must be taken to make sure the At predicates are maintained properly. When a plane flies from one airport to another, all the cargo

inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be At anywhere when it is In a plane; the cargo only becomes At the new airport when it is unloaded. So At really means "available for use at a given location."

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\quad \land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
$\quad$ PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
$\quad$ PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
$\quad$ EFFECT: $\neg At(p, from) \land At(p, to))$

Figure 11.1 A PDDL description of an air cargo transportation planning problem.

The following plan is a solution to the problem:

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

Example domain: The spare tire problem

Consider the problem of changing a flat tire (Figure 11.2). 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.

$Init(Tire(Flat) \land Tire(Spare) \land At(Flat,Axle) \land At(Spare,Trunk))$
$Goal(At(Spare,Axle))$
$Action(Remove(obj,loc),$
PRECOND: $At(obj,loc)$
EFFECT: $\neg At(obj,loc) \land At(obj,Ground))$
$Action(PutOn(t, Axle),$
PRECOND: $Tire(t) \land At(t,Ground) \land \neg At(Flat,Axle) \land \neg At(Spare,Axle)$
EFFECT: $\neg At(t,Ground) \land At(t,Axle))$
$Action(LeaveOvernight,$
PRECOND:
EFFECT: $\neg At(Spare,Ground) \land \neg At(Spare,Axle) \land \neg At(Spare,Trunk)$
$\land \neg At(Flat,Ground) \land \neg At(Flat,Axle) \land \neg At(Flat, Trunk))$

Figure 11.2 The simple spare tire problem.

[Remove(Flat,Axle),Remove(Spare,Trunk),PutOn(Spare,Axle)] is a solution to the problem.

Example domain: The blocks world

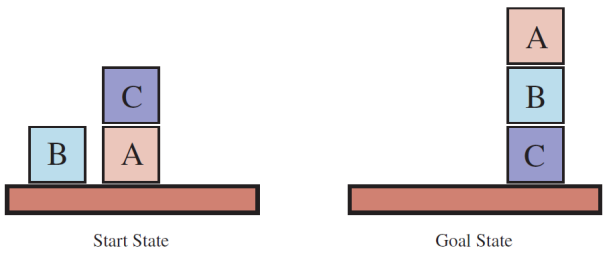


Figure 11.3 Diagram of the blocks-world problem in Figure 11.4.

One of the most famous planning domains is the blocks world. This domain consists of a set of cube-shaped blocks sitting on an arbitrarily-large table.1 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 top of it. A typical goal to get block A on B and block B on C (see Figure 11.3).

$Init(On(A,Table) \land On(B,Table) \land On(C,A)$
$\land Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C) \land Clear(Table))$
$Goal(On(A,B) \land On(B,C))$
$Action(Move(b,x,y),$
PRECOND: $On(b,x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land$
$(b \neq x) \land (b \neq y) \land (x \neq y),$
EFFECT: $On(b,y) \land Clear(x) \land \neg On(b,x) \land \neg Clear(y))$
$Action(MoveToTable(b,x),$
PRECOND: $On(b,x) \land Clear(b) \land Block(b) \land Block(x),$
EFFECT: $On(b,Table) \land Clear(x) \land \neg On(b,x))$

Figure 11.4 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C,A),Move(B,Table,C),Move(A,Table,B)]$.

2. Algorithms for Classical Planning

The description of a planning problem provides an obvious way to search from the initial state through the space of states, looking for a goal. A nice advantage of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state (Figure 11.5 compares forward and backward searches). A third possibility is to translate the problem description into a set of logic sentences, to which we can apply a logical inference algorithm to find a solution.

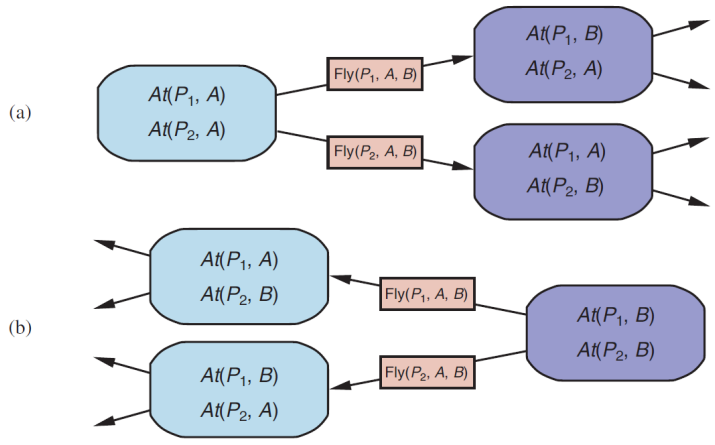


Figure 11.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of ground states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through state descriptions, starting at the goal and using the inverse of the actions to search backward for the initial state.

2.1 Forward state-space search for planning

We can solve planning problems by applying any of the heuristic search algorithms. The states in this search state space are ground states, where every fluent is either true or not. The goal is a state that has all the positive fluents in the problem's goal and none of the negative fluents. The applicable actions in a state, Actions(s), are grounded instantiations of the action schemas—that is, actions where the variables have all been replaced by constant values.

To determine the applicable actions we unify the current state against the preconditions of each action schema. For each unification that successfully results in a substitution, we apply the substitution to the action schema to yield a ground action with no variables. (It is a requirement of action schemas that any variable in the effect

must also appear in the precondition; that way, we are guaranteed that no variables remain after the substitution.)

Each schema may unify in multiple ways. In the spare tire example, the Remove action has the precondition At(obj, loc), which matches against the initial state in two ways, resulting in the two substitutions {obj/Flat, loc/Axle} and {obj/Spare, loc/Trunk}; applying these substitutions yields two ground actions. If an action has multiple literals in the precondition, then each of them can potentially be matched against the current state in multiple ways.

Andvantages and Disadvantages of forward search:

For most problem domains forward search the branching factor is much higher than backward search. However, the fact that forward search uses ground states makes which makes it easier to come up with good heuristics. That is the main reason why the majority of current systems favor forward search.

2.2 Backward search for planning

In backward search (also called regression search) we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. At each step we consider relevant actions (in contrast to forward search, which considers actions that are applicable). This reduces the branching factor significantly, particularly in domains with many possible actions.

A relevant action is one with an effect that unifies with one of the goal literals, but with no effect that negates any part of the goal. For example, with the goal ¬Poor ∧ Famous, an action with the sole effect Famous would be relevant, but one with the effect Poor ∧ Famous is not considered relevant: even though that action might be used at some point in the plan (to establish Famous), it cannot appear at this point in the plan because then Poor would appear in the final state.

What does it mean to apply an action in the backward direction? Given a goal g and an action a, the regression from g over a gives us a state description g′ whose positive and negative literals are given by

$$\mathbf{POS(g') = (POS(g) - ADD(a)) \cup POS(Precond(a))}$$
$$\mathbf{NEG(g') = (NEG(g) - DEL(a)) \cup NEG(Precond(a))\ .}$$

That is, the preconditions must have held before, or else the action could not have been executed, but the positive/negative literals that were added/deleted by the action need not have been true before.

These equations are straightforward for ground literals, but some care is required when there are variables in g and a. For example, suppose the goal is to deliver a

specific piece of cargo to SFO: At(C2,SFO). The Unload action schema has the effect At(c,a). When we unify that with the goal, we get the substitution {c/C2,a/SFO}; applying that substitution to the schema gives us a new schema which captures the idea of using any plane that is at SFO:

Action(Unload(C2, p′,SFO),
PRECOND:In(C2, p′) ∧ At(p′,SFO) ∧ Cargo(C2) ∧ Plane(p′) ∧ Airport(SFO)
EFFECT:At(C2,SFO) ∧ Â¬In(C2, p′)) .
Here we replaced p with a new variable named p′. This is an instance of standardizing apart variable names so there will be no conflict between different variables that happen to have the same name (see page 284). The regressed state description gives us a new goal:
g′ = In(C2, p′) ∧ At(p′,SFO) ∧ Cargo(C2) ∧ Plane(p′) ∧ Airport(SFO) .

Example:
As another example, consider the goal of owning a book with a specific ISBN number:
Own(9780134610993).

Given a trillion 13-digit ISBNs and the single action schema
A = Action(Buy(i),PRECOND:ISBN(i),EFFECT:Own(i)) .
a forward search without a heuristic would have to start enumerating the **10 billion** ground Buy actions.

But with backward search, we would unify the goal Own(9780134610993) with the effect Own(i′), yielding the substitution θ={i′/9780134610993}. Then we would regress over the action Subst(θ,A) to yield the predecessor state description ISBN(9780134610993).
This is part of the initial state, so we have a solution and we are done, having considered just **one action, not a trillion**.

Advantage and Disadvantage of Backword Search:

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

2.3 Planning as Boolean satisfiability

SAT-based planners such as SATPLAN operate by translating a PDDL problem description into propositional form. The translation involves a series of steps:

1. Propositionalize the actions: for each action schema, form ground propositions by substituting constants for each of the variables.
• Add action exclusion axioms saying that no two actions can occur at the same time,
• Add precondition axioms.
• Define the initial state.
• Propositionalize the goal: the goal becomes a disjunction over all of its ground instances, where variables are replaced by constants.
• Add successor-state axioms

The resulting translation is typically much larger than the original PDDL, but modern the efficiency of modern SAT solvers often more than makes up for this. Planning as SAT is a valuable approach for solving a variety of planning problems. By leveraging the power of SAT solvers, AI systems can efficiently find solutions to complex challenges.

2.4 Other classical planning approaches

Planning graph:

An approach called Graphplan uses a specialized data structure, a planning graph, to encode constraints on how actions are related to their preconditions and effects, and on which things are mutually exclusive.

Situation calculus:

Situation calculus is a method of describing planning problems in first-order logic. It uses successor-state axioms just as SATPLAN does, but first-order logic allows for more flexibility and more succinct axioms.

Partial-order planning:

All the approaches we have seen so far construct totally ordered plans consisting of strictly linear sequences of actions. But if an air cargo problem has 30 packages being loaded onto one plane and 50 packages being loaded onto another, it seems pointless to decree a specific linear ordering of the 80 load actions.

An alternative called partial-order planning represents a plan as a graph rather than a planning linear sequence: each action is a node in the graph, and for each precondition of the action there is an edge from another action (or from the initial state) that indicates that the predecessor action establishes the precondition. So we could have a partial-order plan that says that actions

Remove(Spare,Trunk) and Remove(Flat,Axle) must come before PutOn(Spare,Axle), but without saying which of the two Remove actions should come first. We search in the space of plans rather than world-states, inserting actions to satisfy conditions.

Summary:

This chapter described the **PDDL** representation for classical planning problems, and presented algorithmic approaches(**forward** and **backward search**) for finding solutions. The points to remember:

• Planning systems are problem-solving algorithms that **operate on** explicit factored representations of **states** and **actions**. These representations make possible the derivation of effective domain-independent heuristics and the development of powerful and flexible algorithms for solving problems.

• PDDL, the Planning Domain Definition Language, **describes** the initial and goal **states** as conjunctions of literals, and **actions** in terms of their preconditions and effects.

• State-space search can **operate** in the **forward direction (progression)** or the **backward direction (regression)**. Effective heuristics can be derived by subgoal independence assumptions and by various relaxations of the planning problem.

• Other approaches include encoding a planning problem as a Boolean satisfiability problem or as a constraint satisfaction problem; and explicitly searching through the space of partially ordered plans.