UNIT V KNOWLEDGE REPRESENTATION AND PLANNING

Ontological engineering –categories and objects –events –mental objects and modal logic reasoning systems for categories –reasoning with default information
Classical planning –algorithms for classical planning –heuristics for planning –hierarchical planning –non-deterministic domains –time, schedule, and resources-analysis

ONTOLOGICAL ENGINEERING

Representation of abstract concepts such as *events, time, physical objects,* and *beliefs that occur in many different domains* is sometimes called **ontological engineering**.
The general framework of concepts is called an **upper ontology** because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in Figure .

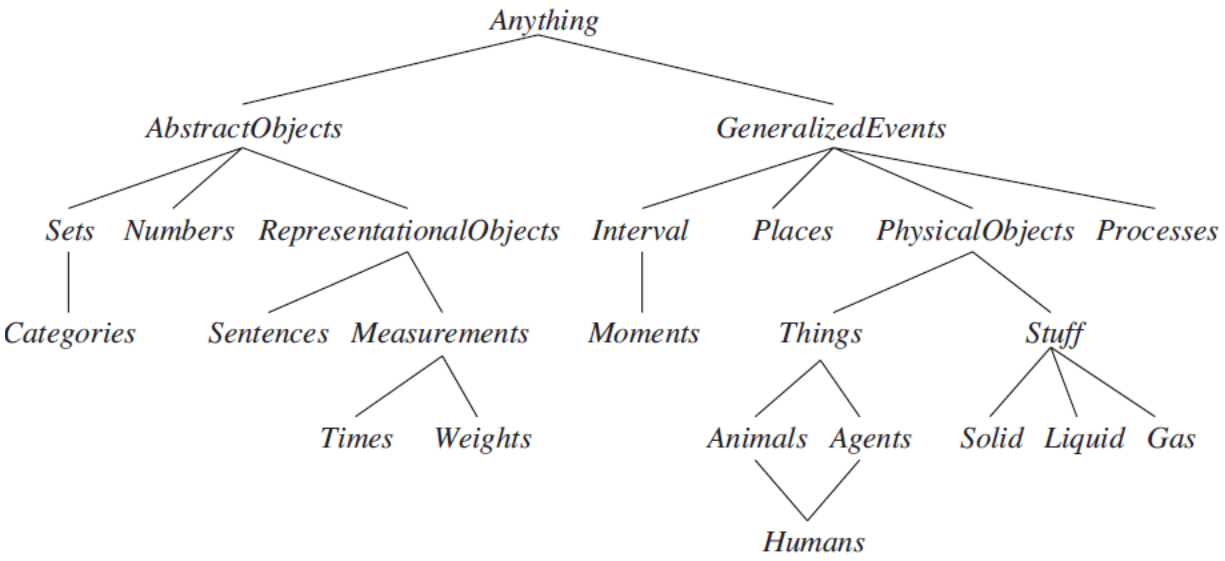


Figure shows the upper ontology of the world. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example.

A more general ontology, better suited for the real world, would allow for simultaneous changes extended over time. For any special-purpose ontology, it is possible to make changes like these to move toward greater generality.

Two major characteristics of general-purpose ontologies distinguish them from collections of special-purpose ontologies:

- A general-purpose ontology should be applicable in 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.
- In any sufficiently demanding domain, different areas of knowledge must be *unified*, because reasoning and problem solving could involve several areas simultaneously.

CATEGORIES AND OBJECTS

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*.

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.

There are two choices for representing categories in first-order logic: predicates and objects. Categories organize knowledge through **inheritance**. For example, all instances of the category *Food* are edible, and if *Fruit* is a subclass of *Food* and *Apples* is a subclass of *Fruit*, then every apple is edible can be inferred. Thus the individual apples **inherit** the property of edibility, in this case this is inherited from their membership in the *Food* category.

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. 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 makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members.
For example, a shopper would normally have the goal of buying a basketball, rather than a *particular* basketball such as BB9. 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.
$Orange(x) \land Round(x) \land Diameter(x) = 9.5" \land x \in Balls \Rightarrow x \in Basketballs$

A category as a whole has some properties.
$Dogs \in DomesticatedSpecies$

Notice that because *Dog* is a category and is a member of *DomesticatedSpecies*, the latter must be a category of categories.

Although subclass and member relations are the most important ones for categories, relations between categories are not subclasses of each other.

For example, if *Undergraduates* and *Graduate Students* are subclasses of *Students*, an undergraduate cannot also be a graduate student.
Two or more categories are **disjoint** if they have no members in common. The classes undergraduate and graduate student form an **exhaustive decomposition** of university students. A exhaustive decomposition of disjoint sets is known as a **partition**.

Here are some more examples of these three concepts:
Disjoint({Animals, V egetables})
ExhaustiveDecomposition({Americans,Canadians,Mexicans},NorthAmericans)
Partition({Animals, Plants, Fungi, Protista,Monera},LivingThings).

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) .

Categories can also be *defined* by providing necessary and sufficient conditions for membership. For example, a bachelor is an unmarried adult male:
x $\in$ Bachelors $\Leftrightarrow$ Unmarried(x) $\wedge$ x $\in$ Adults $\wedge$ x $\in$ Males .

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 each chapter is part of this book. The general *PartOf* relation states that one thing is part of another. Objects can be grouped into *part of* hierarchies, reminiscent(resemblance) of the *Subset* hierarchy:

Example
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, *PartOf (Bucharest,Earth) can be concluded*. Categories of **composite objects** are often characterized by structural relations among parts.

An object is composed of the parts in its *PartPartition* and can be viewed as deriving some properties from those parts. For example, the mass of a composite object is the sum of the masses of the parts.

It is also useful to define composite objects with definite parts but no particular structure.For example, "The apples in this bag weigh two pounds." The temptation would be to ascribe this weight to the *set* of apples in the bag, but this would be a mistake because the set is an abstract mathematical concept that has elements but does not have weight. Instead, a new concept, a **bunch can be used**. For example, if the apples are Apple1,Apple2, and Apple3 then

BunchOf({Apple1,Apple2,Apple3})denotes the composite object with the three apples as parts (not elements). Notice that BunchOf ({x})= x.

Furthermore, BunchOf (Apples) is the composite object consisting of all apples—not to be confused with Apples, the category or set of all apples.

The BunchOf can be defined in terms of the PartOf relation. Obviously, each element of s is part of BunchOf (s):
$\forall x\ x \in s \Rightarrow PartOf(x, BunchOf(s))$.

Furthermore, BunchOf (s) *is the smallest object satisfying this condition*. In other words, BunchOf (s) must be part of any object that has all the elements of s as parts:

$\forall y\ [\forall x\ x \in s \Rightarrow PartOf(x, y)] \Rightarrow PartOf(BunchOf(s), y)$.

These axioms are an example of a general technique called **logical minimization**, which means defining an object as the smallest one satisfying certain conditions.

Measurements

In both scientific and common sense theories of the world, objects have height, mass, cost, and so on. The values that are assigned for these properties are called **measures**. Ordinary quantitative measures are quite easy to represent. The universe includes abstract "measure objects," such as the *length* that is the length of this line segment : ——————————
This length is 1.5 inches or 3.81 centimeters. Thus, the same length has different names in language. The length with a **units function** can take a number as arguments.

If the line segment is called L1, then the length can be written as
Length(L1) = Inches (1.5) = Centimeters(3.81) .
Conversion between units is done by equating multiples of one unit to another:
Centimeters(2.54 × d)=Inches(d) .
Similar axioms can be written for pounds and kilograms, seconds and days, and dollars and cents.
Measures can be used to describe objects as follows:
Diameter (Basketball 12)=Inches(9.5) .
ListPrice(Basketball 12)=$(19) .
$d \in Days \Rightarrow Duration(d)=Hours(24)$.

Simple, quantitative measures are easy to represent. The most important aspect of measures is not the particular numerical values, but the fact that measures can be *ordered*. Although measures are not numbers, still it can be compared, using an ordering symbol such as >.

For example, Norvig's exercises are tougher than Russell's, and that one scores less on tougher exercises can be written as:

e1 ∈ Exercises ∧ e2 ∈Exercises ∧ Wrote(Norvig, e1) ∧ Wrote(Russell, e2) ⇒ Difficulty(e1) > Difficulty(e2) .

e1 ∈ Exercises ∧ e2 ∈Exercises ∧ Difficulty(e1) > Difficulty(e2) ⇒ ExpectedScore(e1) < ExpectedScore(e2) .

This is enough to allow one to decide which exercises to do, even though no numerical values for difficulty were ever used. These sorts of monotonic relationships among measures form the basis for the field of **qualitative physics**, a subfield of AI that investigates how to reason about physical systems without plunging into detailed equations and numerical simulations.

Objects: Things and 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, complexity involved in dealing with vast numbers of primitive objects individually can be reduced. There is, however, a significant portion of reality that seems to reject any obvious **individuation**—division into distinct objects. This generic name is called as **stuff**.

The English language distinguishes clearly between *stuff* and *things*. Linguists distinguish between **count nouns** such as apples, basket balls and **mass nouns**, such as butter, water, and energy. Several competing ontologies claim to handle this distinction.

Consider the example, butter is yellow, is less dense than water, is soft at room temperature, has a high fat content, and so on. On the other hand, butter has no particular size, shape, or weight. More specialized categories of butter can be defined such as UnsaltedButter , which is also a kind of *stuff*. Note that the category PoundOfButter , which includes as members all butter-objects weighing one pound, is not a kind of *stuff*.

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. On the other hand, their **extrinsic** properties—weight, length, shape, and so on—are not retained under subdivision. A category of objects that includes in its definition only *intrinsic* properties is then a substance, or mass noun; a class that includes *any* extrinsic properties in its definition is a count noun. The category Stuff is the most general substance category, specifying no intrinsic properties. The category Thing is the most general discrete object category, specifying no extrinsic properties.

EVENTS

Situation calculus represents actions and their effects. Situation calculus is limited in its applicability: it was designed to describe a world in which actions are discrete, instantaneous, and happen one at a time. Consider a continuous action, such as filling a bathtub. Situation calculus 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 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 alternative formalism known as **event calculus is introduced**, which is based on points of time rather than on situations.

Event calculus reifies(considers) fluents and events.
The fluent At(Shankar , Berkeley) is an object that refers to the fact of Shankar being in Berkeley, but does not by itself say anything about whether it is true.

To assert that a fluent is actually true at some point in time the predicate T is used , as
T(At(Shankar , Berkeley), t).

Events are described as instances of event categories. The event E1 of Shankar flying from San Francisco to Washington, D.C. is described as
$E1 \in Flyings \wedge Flyer(E1, Shankar) \wedge Origin(E1, SF) \wedge Destination(E1,DC)$.

An alternative three-argument version of the category of flying events can be described by
$E1 \in Flyings(Shankar , SF,DC)$.

Happens(E1, i) is used to describe that the event E1 took place over the time interval i, and this in functional form is given as Extent(E1)=i. Time intervals can be represented by a (start, end) pair of times; that is, i = (t1, t2) is the time interval that starts at t1 and ends at t2. The complete set of predicates for one version of the event calculus is

T(f, t) Fluent f is true at time t
Happens(e, i) Event e happens over the time interval i
Initiates(e, f, t) Event e causes fluent f to start to hold at time t
Terminates(e, f, t) Event e causes fluent f to cease to hold at time t
Clipped(f, i) Fluent f ceases to be true at some point during time interval i
Restored (f, i) Fluent f becomes true sometime during time interval i

A distinguished event, Start , describes the initial state by saying which fluents are initiated or terminated at the start time. T can be defined by saying that a fluent holds at a point in time if the fluent was initiated by an event at some time in the past and was not made (clipped)false by an intervening event. A fluent does not hold if it was terminated by an event and not made true (restored) by another event. Formally, the axioms are:

$Happens(e, (t1, t2)) \wedge Initiates(e, f, t1) \wedge \neg Clipped(f, (t1, t)) \wedge t1 < t \Rightarrow T(f, t)$

$Happens(e, (t1, t2)) \wedge Terminates(e, f, t1) \wedge \neg Restored(f, (t1, t)) \wedge t1 < t \Rightarrow \neg T(f, t)$

where Clipped and Restored are defined by

$Clipped(f, (t1, t2)) \Leftrightarrow \exists\ e, t, t3\ Happens(e, (t, t3)) \wedge t1 \leq t < t2 \wedge Terminates(e, f, t)$

Restored (f, (t1, t2)) ⇔ ∃ e, t, t3 Happens(e, (t, t3)) ∧ t1 ≤ t < t2 ∧ Initiates(e, f, t)

It is convenient to extend T to work over intervals as well as time points; a fluent holds over an interval if it holds on every point within the interval:
T(f, (t1, t2)) ⇔ [∀ t (t1 ≤ t < t2) ⇒ T(f, t)]

Fluents and actions are defined with domain-specific axioms that are similar to successor state axioms.
For example, In a wumpus-world the only way an agent gets an arrow is at the start, and the only way to use up an arrow is to shoot it:
Initiates(e, HaveArrow(a), t) ⇔ e = Start
Terminates(e, HaveArrow(a), t) ⇔ e ∈ Shootings(a)

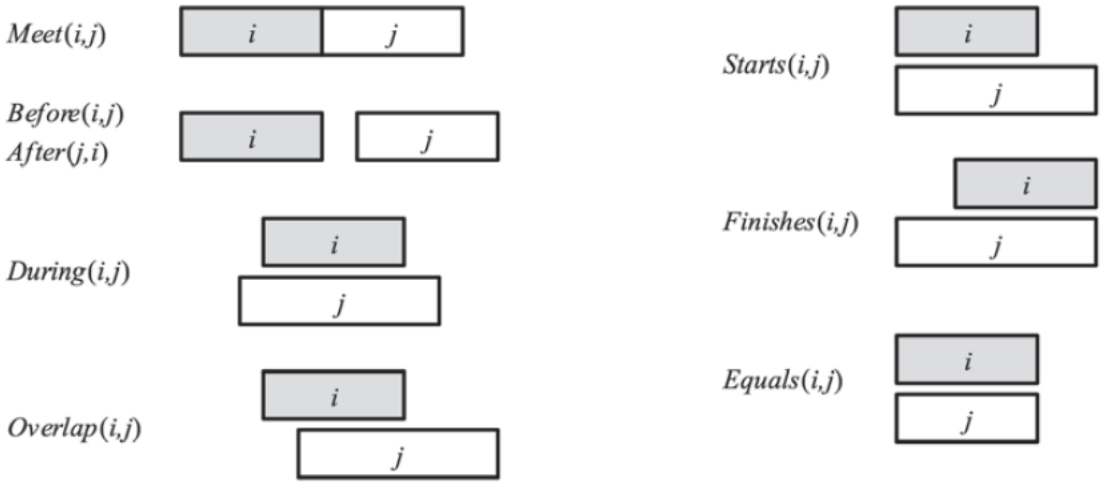
In an ontology where events are n-ary predicates, there would be no way to add extra information like this; moving to an n+ 1-ary predicate isn't a scalable solution.

Processes
Discrete events are those that have a definite structure. Shankar's trip has a beginning, middle, and end. If interrupted halfway, the event would be something different—it would not be a trip from San Francisco toWashington, but instead a trip from San Francisco to somewhere over Kansas. On the other hand, the category of events denoted by Flyings has a different quality.
Categories of events with this property are called **process** categories or **liquid event** categories. Any process e that happens over an interval also happens over any subinterval:
(e ∈ Processes) ∧ Happens(e, (t1, t4)) ∧ (t1 < t2 < t3 < t4) ⇒ Happens(e, (t2, t3)) .

The distinction between liquid and nonliquid events is exactly analogous to the difference between substances, or *stuff*, and individual objects, or *things*. In fact, some describe liquid events as **temporal substances**, whereas substances like butter are **spatial substances**.

Time intervals
Event calculus also includes time and time intervals. The two kinds of time intervals: moments and extended intervals. The distinction is that only moments have zero duration:
The function Duration gives the difference between the end time and the start time.
Interval (i) ⇒ Duration(i)=(Time(End(i)) − Time(Begin(i))) .

Two intervals Meet if the end time of the first equals the start time of the second. The complete set of interval relations, as proposed by Allen (1983), is shown graphically in Figure logically below:
Meet(i, j) ⇔ End(i)=Begin(j)
Before(i, j) ⇔ End(i) < Begin(j)
After (j, i) ⇔ Before(i, j)
During(i, j) ⇔ Begin(j) < Begin(i) < End(i) < End(j)
Overlap(i, j) ⇔ Begin(i) < Begin(j) < End(i) < End(j)
Begins(i, j) ⇔ Begin(i) = Begin(j)
Finishes(i, j) ⇔ End(i) = End(j)
Equals(i, j) ⇔ Begin(i) = Begin(j) ∧ End(i) = End(j)



These all have their intuitive meaning, with the exception of *Overlap*:
Overlap(i, j) only holds if *i* begins before *j*.

Fluents and objects

- Physical objects can be viewed as generalized events, it is a chunk of space–time.
- For example, USA can be thought of as an event that began in, say, 1776 as a union of 13 states and is still in progress today as a union of 50.
- The changing properties of USA can be described using state fluents, such as Population(USA).
- A property of the USA that changes every four or eight years, barring mishaps, is its president
- (The term President(USA, t) can denote different objects, depending on the value of t, but 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.

MENTAL EVENTS AND MENTAL OBJECTS

Knowledge about one's own knowledge and reasoning processes is useful for controlling inference.

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. On the other hand, if the question were "Is your mother sitting down right now?" then Bob should realize that thinking harder is unlikely to help.

Knowledge about the knowledge of other agents is also important; Bob should realize that his mother knows whether she is sitting or not, and asking her would be a way to find out.

A model of the mental objects is needed to analyze what is in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model does not have to be detailed.

This can be done with the **propositional attitudes** that an agent can have toward mental object attitudes such as Believes, Knows, Wants, Intends, and Informs. The difficulty is that these attitudes do not behave like "normal" predicates.

For example, Lois knows that Superman can fly: can be asserted as

Knows(Lois, CanFly(Superman)) .

One minor issue with this CanFly(Superman)appears as a term instead of a sentence. This issue can rectified by making CanFly(Superman); as a fluent.

A more serious problem is that, if it is true that Superman is Clark Kent, then the conclusion must be Lois knows that Clark can fly:

(Superman = Clark) ∧ Knows(Lois , CanFly(Superman)) |= Knows(Lois, CanFly(Clark)) .

This is a consequence of the fact that equality reasoning is built into logic.

Referential transparency is a property in which that doesn't matter what term a logic uses to refer to an object, what matters is the object that the term names.

Modal logic is designed to address this problem. Regular logic is concerned with a single modality, the modality of truth, that allows to express "*P* is true." Modal logic includes special modal operators that take sentences (rather than terms) as arguments.

For example, "*A* knows *P*" is represented with the notation $\mathbf{K}_A P$, 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.

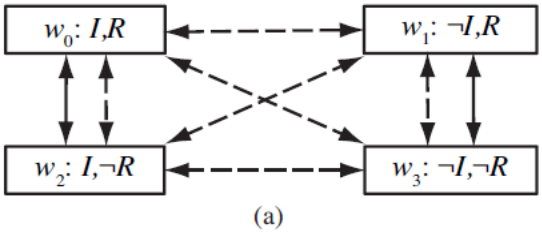
The semantics of modal logic is more complicated. In first-order logic a **model** contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In general, a knowledge atom $\mathbf{K}_A P$ is true in world w if and only if P is true in every world accessible from w.

The truth of more complex sentences is derived by recursive application of this rule and the normal rules of first-order logic. That means that modal logic can be used to reason about nested knowledge sentences: what one agent knows about another agent's knowledge.

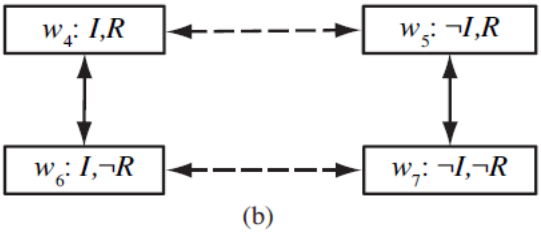
For example, even though Lois doesn't know whether Superman's secret identity is Clark Kent, she does know that Clark knows:

KLois [KClark Identity(Superman, Clark) ∨ KClark ¬ Identity(Superman, Clark)]

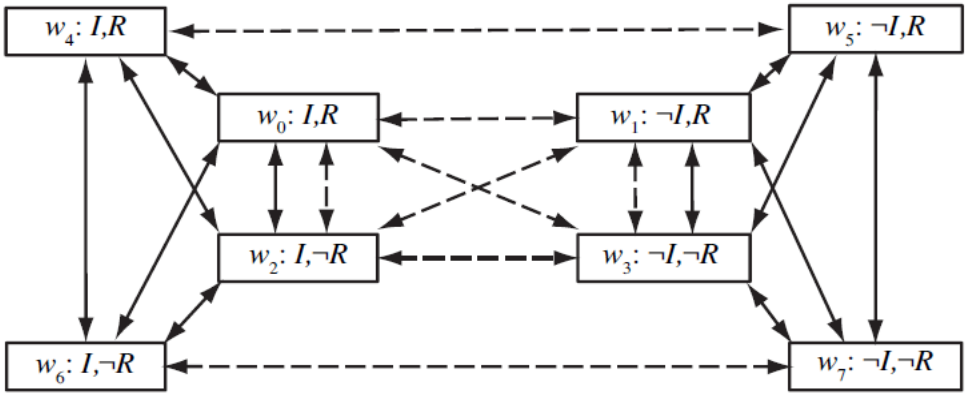
Figure shows some possible worlds for this domain, with accessibility relations for Lois and Superman.



(a)



(b)



(c)

In the TOP-LEFT diagram, it is common knowledge that Superman knows his own identity, and neither he nor Lois has seen the weather report. So in w0 the worlds w0 and w2 are accessible to Superman; maybe rain is predicted, maybe not. For Lois all four worlds are accessible from each other; she doesn't know anything about the report or if Clark is Superman. But she does know that Superman knows whether he is Clark, because in every world that is accessible to Lois, either Superman knows I, or he knows ¬ I. Lois does not know which is the case, but either way she knows Superman knows. In the TOP-RIGHT diagram it is common knowledge that Lois has seen the weather report. So in w4 she knows rain is predicted and in w6 she knows rain is not predicted.

Superman does not know the report, but he knows that Lois knows, because in every world that is accessible to him, either she knows R or she knows ¬ R.

In the BOTTOM diagram consider the scenario where it is common knowledge that Superman knows his identity, and Lois might or might not have seen the weather report. This can be represented by combining the two top scenarios, and adding arrows to show that Superman does not know which scenario actually holds.

Lois does know, hence no need to add any arrows for her. In w0 Superman still knows I but not R, and now he does not know whether Lois knows R. From what Superman knows, he might be in w0 or w2, in which case Lois does not know whether R is true, or he could be in w4, in which case she knows R, or w6, in which case she knows ¬ R.

Modal logic solves some tricky issues with the interplay of quantifiers and knowledge. The English sentence "Bond knows that someone is a spy" is ambiguous. This can be written as
$\exists$ x **K**BondSpy(x) ,which in modal logic means that there is an x that, in all accessible worlds, Bond knows to be a spy.
The second one is that Bond just knows that there is at least one spy:
KBond$\exists$ x Spy(x) .
The modal logic interpretation is that in each accessible world there is an x that is a spy, but it need not be the same x in each world.
A modal operator for knowledge can be written in the form of axiom as.
(**K**aP $\wedge$ **K**a(P $\Rightarrow$ Q)) $\Rightarrow$ **K**aQ.

Agents are able to draw deductions; if an agent knows P and knows that P implies Q, then the agent knows Q:

Thus **K**A(P $\vee$ $\neg$ P) is a tautology; every agent knows every proposition P is either true or false.

On the other hand, (**K**AP) $\vee$ (**K**A $\neg$ P) is not a tautology; in general, there will be lots of propositions that an agent does not know to be true and does not know to be false.

If something is known, it must be true, it can be written in axiom as
KaP $\Rightarrow$ P .

Furthermore, logical agents should be able to introspect on their own knowledge. If they know something, then they know that they know it:
KaP $\Rightarrow$ **K**a(**K**aP) .

Similar axioms for belief (often denoted by **B**) and other modalities can be defined. However, one problem with the modal logic approach is that it assumes **logical omniscience** on the part of agents. That is, if an agent knows a set of axioms, then it knows all consequences of those axioms.

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*:

In **linear temporal logic**, following modal operators can be added:
XP: " will be true in the next time step"
FP: " will eventually (**F**inally) be true in some future time step"
GP: " is always (**G**lobally) true"
P U Q: " remains true until occurs"

Sometimes there are additional operators that can be derived from these. Adding these modal operators makes the logic itself more complex (and thus makes it harder for a logical inference algorithm to find a proof). But the operators also allows to state certain facts in a more succinct form (which makes logical inference faster). The choice of which logic to use is similar to the choice of which programming language to use: pick one that is appropriate to your task, that is familiar to you and the others who will share your work, and that is efficient enough for your purposes.

REASONING SYSTEM FOR CATEGORIES

Categories are the primary building blocks of large-scale knowledge representation schemes. There are two closely related families of systems: **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; and **Description logics** provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.

Semantic networks

- Semantic networks are alternative of predicate logic for knowledge representation.
- In Semantic networks, knowledge can be represented in the form of graphical networks.
- This network consists of nodes representing objects and arcs which describe the relationship between those objects.
- Semantic networks can categorize the object in different forms and can also link those objects.
- Semantic networks are easy to understand and can be easily extended.

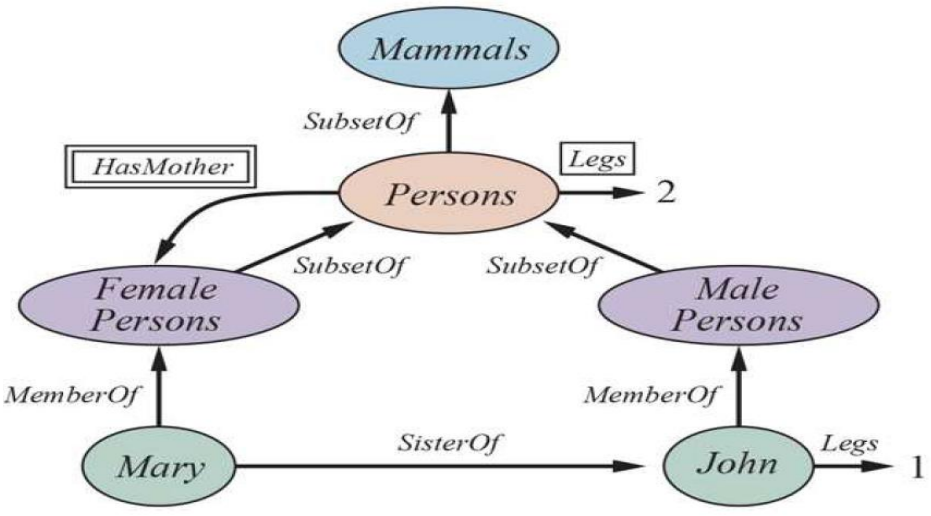
This representation consist of mainly two types of relations:

1. IS-A relation (Inheritance)

2. Kind-of-relation

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 labelled links.

For example, Figure has a MemberOf 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).Categories can be connected using SubsetOf links, and so on.



A special notation—the double-boxed link is used that represents categories do not have mothers

This link asserts that

$\forall x \; x \in \text{Persons} \Rightarrow [\forall y \; \text{HasMother}(x, y) \Rightarrow y \in \text{FemalePersons}]$.

The persons that has two legs can be asserted as

$\forall x \; x \in \text{Persons} \Rightarrow \text{Legs}(x, 2)$.

The semantic network notation makes it convenient to perform **inheritance.** 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

Designers can build a large network and still have a good idea about what queries will be efficient, because

(a) it is easy to visualize the steps that the inference procedure will go through and

(b) in some cases the query language is so simple that difficult queries cannot be posed.

In cases where the expressive power proves to be too limiting, many semantic network systems provide for **procedural attachment** to fill in the gaps. Procedural attachment is a technique whereby a query about (or sometimes an assertion of) a certain relation results in a call to a special procedure designed for that relation rather than a general inference algorithm.

One of the most important aspects of semantic networks is their ability to represent **default values** for categories. Examining Figure carefully, one notices that John has one 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 is contradicted 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.

Description logics

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.

The CLASSIC language is a typical description logic. The syntax of CLASSIC descriptions is shown in Figure

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

For example, to say that bachelors are unmarried adult males

Bachelor = And(Unmarried, Adult ,Male) .

The equivalent in first-order logic would be

Bachelor (x) ⇔ Unmarried(x) ∧ Adult(x) ∧ Male(x) .

Notice that the description logic has an an algebra of operations on predicates, which of course cant be used 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

And(Man, AtLeast(3, Son), AtMost(2, Daughter),

All(Son, And(Unemployed,Married, All(Spouse, Doctor))),

All(Daughter , And(Professor , Fills(Department , Physics,Math)))) .

Perhaps the most important aspect of description logics is their emphasis on tractability of inference. A problem instance is solved by describing it and then asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. The thrust in description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the descriptions.

REASONING WITH DEFAULT INFORMATION

This is a very common from of non-monotonic reasoning. Here the *conclusions are drawn based on what is most likely to be true.*

Two approaches are:

Non-Monotonic or Circumscription logic.

Default logic.

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 if the default rule that birds fly has to be asserted, introduce a predicate, say Abnormal 1(x), and write as

Bird(x) ∧ ¬ Abnormal 1(x) ⇒ Flies(x) .

Circumscription can be viewed as an example of a **model preference** logic. In such logics, a sentence is entailed (with default status) if it is true in all *preferred* models of the KB, as opposed to the requirement of truth in *all* models in classical logic. For circumscription, one model is preferred to another if it has fewer abnormal objects.

In addition, to assert that religious beliefs take precedence over political beliefs, a formalism called **prioritized circumscription** is used to give preference to models where Abnormal is minimized.

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.
The Nixon-diamond example can be represented in default logic with one fact and two default rules:
Republican(Nixon) ∧ Quaker(Nixon) .

Republican(x) : ¬ Pacifist (x)/ ¬ Pacifist (x) .

Quaker(x) : Pacifist (x)/Pacifist (x) .

To interpret what the default rules mean, the notion of an **extension** of a default theory is defined to be a maximal set of consequences of the theory. That is, an extension consists of the original known facts and a set of conclusions from the default rules, such that no additional conclusions can be drawn from S , and the justifications of every default conclusion in are consistent with S. As in the case of the preferred models in circumscription, the two possible extensions for the Nixon diamond: one where in he is a pacifist and one where in he is not. Prioritized schemes exist in which some default rules can be given precedence over others, allowing some ambiguities to be resolved.

Decisions often involve tradeoffs, and one therefore needs to compare the *strengths* of belief in the outcomes of different actions, and the *costs* of making a wrong decision. In cases where the

same kinds of decisions are being made repeatedly, it is possible to interpret default rules as "threshold probability" statements.

Truth maintenance systems

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**.

Suppose that a knowledge base KB contains a sentence P—perhaps a default conclusion recorded by a forward-chaining algorithm, or perhaps just an incorrect assertion—and if TELL(KB, ¬ P). has to be executed, To avoid creating a contradiction, first execute RETRACT(KB, P). This sounds easy enough.

Problems arise, however, if any *additional* sentences were inferred from P and asserted in the KB. For example, the implication P ⇒ Q might have been used to add Q. The obvious "solution"—retracting all sentences inferred from P—fails because such sentences may have other justifications besides P. For example, if R and R ⇒ Q are also in the KB, then Q does not have to be removed after all. **Truth maintenance systems**, or TMSs, are designed to handle exactly these kinds of complications.

One simple approach to truth maintenance is to keep track of the order in which sentences are told to the knowledge base by numbering them from P1 to Pn. When the call RETRACT(KB, Pi) is made, the system reverts to the state just before Pi was added, thereby removing both Pi and any inferences that were derived from Pi. The sentences Pi+1 through Pn can then be added again.

A more efficient approach JTMS 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. For example, if the knowledge base already contains P ⇒ Q, then TELL(P) will cause Q to be added with the justification {P, P ⇒ Q}. In general, a sentence can have any number of justifications. Justifications make retraction efficient.

The JTMS assumes that sentences that are considered once will probably be considered again, so rather than deleting a sentence from the knowledge base entirely when it loses all justifications, the sentence can be marked as being *out* of the knowledge base. If a subsequent assertion restores one of the justifications, then mark the sentence as being back *in*. In this way, the JTMS retains all the inference chains that it uses and need not rederive sentences when a justification becomes valid again.

In addition to handling the retraction of incorrect information, TMSs can be used to speed up the analysis of multiple hypothetical situations.

An assumption-based truth maintenance system, or **ATMS**, makes context switching between hypothetical worlds particularly efficient. In a JTMS, the maintenance of justifications allows 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.

Truth maintenance systems also provide a mechanism for generating **explanations**. Technically, an explanation of a sentence P is a set of sentences E such that E entails P. If the sentences in E are already known to be true, then E simply provides a sufficient basis for proving that P must be the case. But explanations can also include **assumptions**— sentences that are not known to be true, but would suffice to prove P if they were true.

For example, one might not have enough information to prove that one's car won't start, but a reasonable explanation might include the assumption that the battery is dead. This, combined with knowledge of how cars operate, explains the observed non behavior. In most cases, an explanation E is preferred that is minimal, meaning that there is no proper subset of E that is also an explanation. An ATMS can generate explanations for the "car won't start" problem by making assumptions (such as "gas in car" or "battery dead") in any order, even if some assumptions are contradictory.

The computational complexity of the truth maintenance problem is at least as great as that of propositional inference—that is, NP-hard. When used carefully, however, a TMS can provide a substantial increase in the ability of a logical system to handle complex environments and hypotheses.

DEFINITION OF 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.
The problem-solving agent 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 hybrid propositional logical agent can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. But it relies on ground (variable-free) propositional inference, which means that it may be swamped when there are many actions and states.

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, which allows us to express all actions with a single action schema, and does not need domain-specific knowledge. Basic PDDL can handle classical planning domains, and extensions can handle non-classical domains that are continuous, partially observable, concurrent, and multi-agent.

In PDDL, a **state** is represented as a conjunction of ground atomic fluents. Recall that "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. 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 they 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).

Action(Fly(P1, SFO, JFK),
PRECOND:At(P1, SFO) ∧ Plane(P1) ∧ Airport (SFO) ∧ Airport (JFK)
EFFECT: ¬ 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 (the **delete list** or DEL(a)), and adding the fluents that are positive literals in the action's effects (the **add list** or ADD(a))

RESULT (s, a) = (s − DEL(a)) ∪ ADD(a) .

For example, with the action Fly(P1, SFO, JFK), would remove At(P1, SFO) and add 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. As with all states, the closed-world assumption is used, which means that any atoms that are not mentioned are false. 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: Air cargo transport
Figure 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.

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

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 basic PDDL does not have a universal quantifier, hence a different solution is needed. The approach used says 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."

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)] .

The spare tire problem

Consider the problem of changing a flat tire as shown in figure.

$$
\begin{aligned}
&Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))\\
&Goal(At(Spare, Axle))\\
&Action(Remove(obj, loc),\\
&\quad \text{PRECOND: } At(obj, loc)\\
&\quad \text{EFFECT: } \neg At(obj, loc) \land At(obj, Ground))\\
&Action(PutOn(t, Axle),\\
&\quad \text{PRECOND: } Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle)\\
&\quad \text{EFFECT: } \neg At(t, Ground) \land At(t, Axle))\\
&Action(LeaveOvernight,\\
&\quad \text{PRECOND:}\\
&\quad \text{EFFECT: } \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)\\
&\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))
\end{aligned}
$$

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, version of the problem considered 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. Assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. [Remove(Flat , Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)] is a solution to the problem.

The blocks world

One of the most famous planning domains is known as the **blocks world**. This domain consists of a set of cube-shaped blocks sitting on a table.2 The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block B on C

Start State Goal State

On(b, x) is used to indicate that block b is on x, where x is either another block or the table. The action for moving block b from the top of x to the top of y will be Move(b, x, y). Now, one of the preconditions on moving b is that no other block be on it. In first-order logic, this would be $\neg \exists$ x On(x, b) or, alternatively, $\forall$ x $\neg$ On(x, b). Basic PDDL does not allow quantifiers, instead a predicate Clear (x) that is true is used when nothing is on x.(The complete problem description is in Figure.)

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

The action Move moves a block b from x to y if both b and y are clear. After the move is made, b is still clear but y is not. A first attempt at the Move schema is

Action(Move(b, x, y),

PRECOND:On(b, x) ∧ Clear (b) ∧ Clear (y),

EFFECT:On(b, y) ∧ Clear (x) ∧ ¬ On(b, x) ∧ ¬ Clear (y)) .

Unfortunately, this does not maintain Clear properly when x or y is the table. When x is the Table, this action has the effect Clear (Table), but the table should not become clear; and when y =Table, it has the precondition Clear (Table), but the table does not have to be clear to move a block onto it.

To fix this, the following two things are done. First, introduce another action to move a block b from x to the table:

Action(MoveToTable(b, x),

PRECOND:On(b, x) ∧ Clear (b),

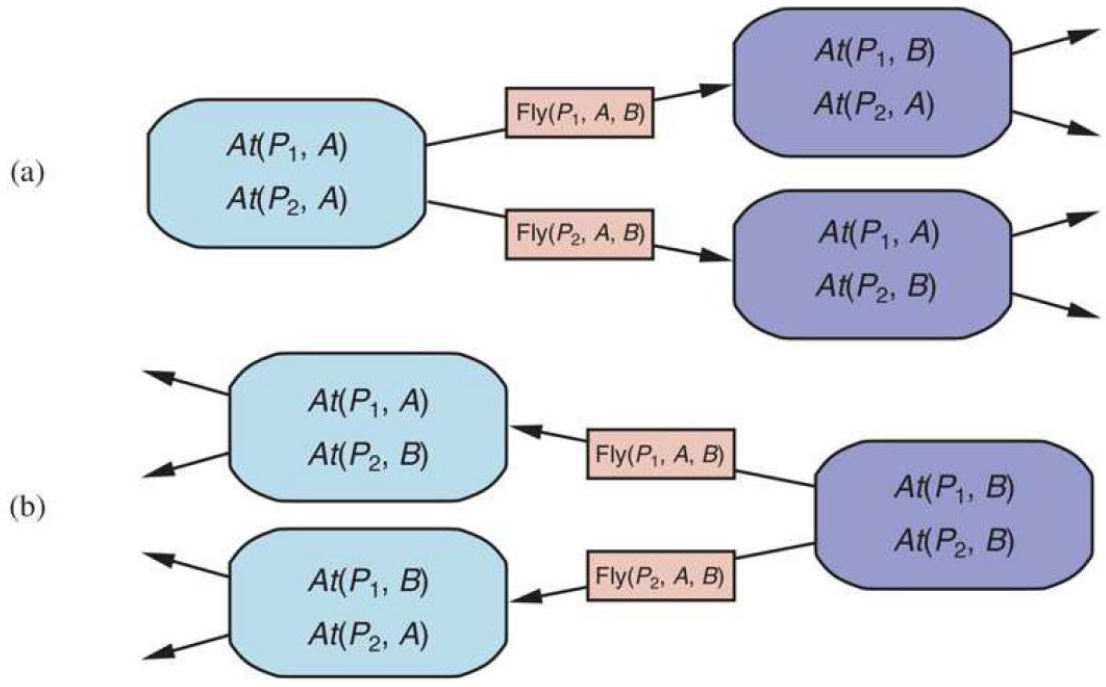
EFFECT:On(b, Table) ∧ Clear (x) ∧ ¬ On(b, x)) .

Second ,take the interpretation of Clear (x) to be "there is a clear space on x to hold a block." Under this interpretation, Clear (Table) will always be true. The only problem is that nothing prevents the planner from using Move(b, x, Table) instead of MoveToTable(b, x).

This lead to incorrect answers hence the predicate Block and add Block(b)∧ Block(y) is introduced for the precondition of Move

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 search can be applied backward from the goal, looking for the initial state . A third possibility is to translate the problem description into a set of logic sentences, to which a logical inference algorithm is applied to find a solution.



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.

Forward state-space search for planning

Planning problems can be solved by applying any of the heuristic search algorithms or local 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 the current state is unified against the preconditions of each action schema. For each unification that successfully results in a substitution, apply the substitution to the action schema to yield a ground action with no variables.

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/Truck}; 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.

Although many real-world applications of planning have relied on domain-specific heuristics, it turns out that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible

Backward search for planning

In backward search (also called **regression search**) start from the goal and apply the actions backward until a sequence of steps is found that reaches the initial state. At each step **relevant actions are considered.** 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.

Given a goal g and an action a, the **regression** from g over a gives a state description g' whose positive and negative literals are given by

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

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 this is unified with the goal, the substitution is {c/C2, a/SFO}got ; applying that substitution to the schema gives 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) ∧ ¬In (C2, p′)).

Here p is replaced 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.
The regressed state description gives the new goal:
g′ = In(C2, p′) ∧ At(p′, SFO) ∧ Cargo(C2) ∧ Plane(p′) ∧ Airport(SFO).

More formally, assume a goal description g that contains a goal literal g_i and an action Schema A . If A has an effect literal e′j where Unify(gi, e′j) = θ and where A′ = SUBST(θ,A) and if there is no effect in A' that is the negation of a literal in g , then A'is a relevant action towards g .

Other classical planning approaches
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** 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.

It is possible to encode a bounded planning problem (i.e., the problem of finding a plan of length k) as a constraint satisfaction problem (CSP). The encoding is similar to the encoding to a SAT problem , with one important simplification: at each time step only a single variable, is needed, whose domain is the set of possible actions. All the approaches construct *totally ordered* plans consisting of strictly linear sequences of actions.

An alternative called **partial-order planning** represents a plan as a graph rather than a 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. Partial-order planning is also often used in domains where it is important for humans to understand the plans. For example, 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 to verify that the plans are correct before they are executed.

HEURISTICS FOR PLANNING
Neither forward nor backward search is efficient without a good heuristic function. By definition, there is no way to analyze an atomic state, and thus it requires some ingenuity by an analyst (usually human) to define good domain-specific heuristics for search problems with atomic states. But planning uses a factored representation for states and actions, which makes it possible to define good domain-independent heuristics.

An admissible heuristic can be derived by defining a **relaxed problem** that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

A search problem is a graph where the nodes are states and the edges are actions. The problem is to find a path connecting the initial state to a goal state.
There are two main ways to relax the problems to make it easier:

- by adding more edges to the graph, making it strictly easier to find a path,
- or by grouping multiple nodes together, forming an abstraction of the state space that has fewer states, and thus is easier to search.

Consider the heuristics that add edges to the graph. Perhaps the simplest is the **ignore preconditions heuristic**, which drops all preconditions from actions. Every action becomes applicable in every state, and any single goal fluent can be achieved in one step .This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others.

For many problems an accurate heuristic is obtained by considering (1) and ignoring (2). First, the actions can be relaxed by removing all preconditions and all effects except those that are literals in the goal. Then, the minimum number of actions required is counted such that the union of those actions' effects satisfies the goal. This is an instance of the **set-cover problem**. Fortunately, a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of log n of the true minimum covering, where n is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

It is also possible to ignore only *selected* preconditions of actions. Consider the sliding block puzzle (8-puzzle or 15-puzzle) . This can be encoded as a planning problem involving tiles with a single schema *Slide*:

Action(Slide(t, s1, s2),
PRECOND:On(t, s1) ∧ Tile(t) ∧ Blank (s2) ∧ Adjacent (s1, s2)
EFFECT:On(t, s2) ∧ Blank(s1)∧ ¬ On(t, s1) ∧¬ Blank(s2))

If the preconditions Blank (s2) ∧ Adjacent (s1, s2) are removed then any tile can move in one action to any space and the number-of-misplaced-tiles heuristic is got. If Blank(s2) is removed the Manhattan-distance heuristic is got. The ease of manipulating the schemas is the great advantage of the factored representation of planning problems, as compared with the atomic representation of search problems.

Another possibility is the **ignore delete lists** heuristic. For example ,Assume that all goals and preconditions contain only positive literals. A relaxed version of the original problem is created that will be easier to solve,where the length of the solution will serve as a good heuristic. This can be done by removing the delete lists from all actions (i.e., removing all negative literals from effects). This makes it possible to make monotonic progress towards the goal—no action will ever undo progress made by another action. It turns out it is still NPhard to find the optimal solution to this relaxed problem, an approximate solution can be found in polynomial time by hill-climbing.

Domain-independent pruning

Factored representations make it obvious that many states are just variants of other states. For example, suppose a dozen blocks is placed on a table, and the goal is to have block A on top of a

three-block tower. The first step in a solution is to place some block x on top of block y (where x ,y ,A and are all different). After that, place A on top of x and hence the solution

There are 11 choices for x, and 10 choices for y, and thus 110 states to consider. But all these states are symmetric: choosing one over another makes no difference, and thus a planner should only consider one of them. This is the process of **symmetry reduction**: where all symmetric branches are pruned out of the search tree except for one. For many domains, this makes the difference between intractable and efficient solving.

Another possibility is to do forward pruning, accepting the risk that sometimes an optimal solution may be pruned out, in order to focus the search on promising branches. For this case define a **preferred action** as follows: First, define a relaxed version of the problem, and solve it to get a **relaxed plan**. Then a preferred action is either a step of the relaxed plan, or it achieves some precondition of the relaxed plan.

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. Such a problem has **serializable subgoals** if there exists an order of subgoals such that the planner can achieve them in that order without having to undo any of the previously achieved subgoals.

For example, in the blocks world, if the goal is to build a tower (e.g., A on B, which in turn is on C, which in turn is on the *Table,*), then the subgoals are serializable bottom to top: if C is first achieved on *Table,* then there is no need to undo it while achieving the other subgoals. A planner that uses the bottom-to-top trick can solve any problem in the blocks world without backtracking.

State abstraction in planning

A relaxed problem leaves with a simplified planning problem to calculate the value of the heuristic function. Many planning problems have 10^{100}states or more, and relaxing the *actions* does nothing to reduce the number of states, which means that it may still be expensive to compute the heuristic. Therefore, relaxations can be that decreases the number of states by forming a **state abstraction**—a many-to-one mapping from states in the ground representation of the problem to the abstract representation. The easiest form of state abstraction is to ignore some fluents.

A key idea in defining heuristics is **decomposition**: dividing a problem into parts, solving each part independently, and then combining the parts. The **subgoal independence** assumption is that the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal *independently*. The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each subgoal—for example, when an action in one subplan deletes a goal achieved by another subplan. It is pessimistic, and therefore inadmissible, when subplans contain redundant actions—for instance, two actions that could be replaced by a single action in the merged plan.

It is clear that there is great potential for cutting down the search space by forming abstractions. The trick is choosing the right abstractions and using them in a way that makes the total cost—defining an abstraction, doing an abstract search, and mapping the abstraction back to the original problem—less than the cost of solving the original problem. The techniques of **pattern databases**

can be useful, because the cost of creating the pattern database can be amortized over multiple problem instances.

A system that makes use of effective heuristics is FF, or FASTFORWARD , a forward state-space searcher uses the ignore-delete-lists heuristic, estimates the heuristic with the help of a planning graph. FF then uses hill climbing search (modified to keep track of the plan) with the heuristic to find a solution. FF's hill climbing algorithm is nonstandard: it avoids local maxima by running a breadth-first search from the current state until a better one is found. If this fails, FF switches to a greedy best-first search instead.

HIERARCHICAL PLANNING

The problem-solving and planning methods all operate with a fixed set of atomic actions. Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions.

The key benefit of hierarchical structure is that at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a *small* number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small.

High-level actions

The basic formalism to understand hierarchical decomposition comes from the area of **hierarchical task networks** or HTN planning. The key additional concept is the **high-level action** or HLA. Each HLA has one or more possible **refinements** that can be converted into a sequence of actions, each of which may be an HLA or a primitive action

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

$$
\begin{aligned}
&Refinement(Go(Home, SFO),\\
&\quad \text{STEPS: } [Drive(Home, SFOLongTermParking),\\
&\qquad\quad Shuttle(SFOLongTermParking, SFO)]\ )\\
&Refinement(Go(Home, SFO),\\
&\quad \text{STEPS: } [Taxi(Home, SFO)]\ )
\end{aligned}
$$

An HLA refinement that contains IMPLEMENTATION only primitive actions is called an **implementation** of the HLA. An implementation of a high-level plan (a sequence of HLAs) is the concatenation of implementations of each HLA in the sequence. Given the precondition–effect definitions of each primitive action, it is straightforward to determine whether any given implementation of a high-level plan achieves the goal. *A high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state.* The "at least one" in this definition is crucial —not *all* implementations need to achieve the goal, because the agent gets to decide which implementation it will execute. Thus, the set of possible implementations in HTN planning —each of which may have a different outcome—is not the same as the set of possible outcomes in nondeterministic planning.

The simplest case is an HLA that has exactly one implementation. When HLAs have multiple possible implementations, there are two options: one is to search among the implementations for one that works ; the other is to reason directly about the HLAs—despite the multiplicity of implementations. The latter method enables the derivation of provably correct abstract plans, without the need to consider their implementations.

Searching for primitive solutions

HTN planning is often formulated with a single "top level" action called *Act*, where the aim is to find an implementation of *Act* that achieves the goal. This approach is entirely general. For example, classical planning problems can be defined as follows: for each primitive action ai, provide one refinement of *Act* with steps [ai, *Act*]. That creates a recursive definition of *Act* that lets us add actions. Some method is needed to stop the recursion; this can be done by providing one more refinement for *Act*, one with an empty list of steps and with a precondition equal to the goal of the problem. This says that if the goal is already achieved, then the right implementation is to do nothing.

This 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 possible implementation based on breadth-first tree search is shown in Figure .

Figure : A breadth-first implementation of hierarchical forward planning search

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier ← a FIFO queue with [Act] as the only element
  loop do
    if EMPTY?(frontier) then return failure
    plan ← POP(frontier)  /* chooses the shallowest plan in frontier */
    hla ← the first HLA in plan, or null if none
    prefix,suffix ← the action subsequences before and after hla in plan
    outcome ← RESULT(problem.INITIAL-STATE, prefix)
    if hla is null then  /* so plan is primitive and outcome is its result */
      if outcome satisfies problem.GOAL then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

Plans are considered in order of depth of nesting of the refinements, rather than number of primitive steps. It is straightforward to design a graph-search version of the algorithm as well as depth-first and iterative deepening versions.

The key to HTN planning is a plan library containing known methods for implementing complex, high-level actions. One way to construct the library is to *learn* the methods from problem-solving experience. After gaining the experience of constructing a plan from scratch, the agent can save the plan in the library as a method for implementing the high level action defined by the task. In this way, the agent can become more and more competent over time as new methods are built on top of old methods. One important aspect of this learning process is the ability to *generalize* the methods that are constructed, eliminating detail that is specific to the problem instance and keeping just the key elements of the plan.

Searching for abstract solutions

The hierarchical search algorithm refines HLAs all the way to primitive action sequences to determine if a plan is workable. This contradicts common sense: one should be able to determine that the two-HLA high-level plan. For this to work, it has to be the case that every high-level plan that "claims" to achieve the goal (by virtue of the descriptions of its steps) does in fact achieve the goal in the sense defined earlier: it must have at least one implementation that does achieve the goal. This property has been called the **downward refinement property** for HLA descriptions.

Writing HLA descriptions that satisfy the downward refinement property is, in principle, easy: as long as the descriptions are *true*, then any high-level plan that claims to achieve the goal must in fact do so—otherwise, the descriptions are making some false claim about what the HLAs do.

One safe answer (at least for problems where all preconditions and goals are positive) is to include only the positive effects that are achieved by *every* implementation of the HLA and the negative effects of *any* implementation. Then the downward refinement property would be satisfied. Unfortunately, this semantics for HLAs is much too conservative.

The programming languages community has coined the term **demonic nondeterminism** for the case where an adversary makes the choices, contrasting this with **angelic nondeterminism**, where the agent itself makes the choices. This term is used to define **angelic semantics** for HLA descriptions. The basic concept required for understanding angelic semantics is the **reachable set** of an HLA: given a state , the reachable set for an HLA , written as REACH(s,h) , is the set of states reachable by any of the HLA's implementations

The key idea is that the agent can choose *which* element of the reachable set it ends up in when it executes the HLA; thus, an HLA with multiple refinements is more "powerful" than the same HLA with fewer refinements. The reachable set can be defined for a sequences of HLAs. For example, the reachable set of a sequence [h1, h2] is the union of all the reachable sets obtained by applying h2 in each state in the reachable set of h1:

REACH(s, [h1, h2]) = $\bigcup_{s' \in \text{REACH}(s',h1)} \text{REACH}(s', h2)$.

Given these definitions, a high-level plan—a sequence of HLAs—achieves the goal if its reachable set *intersects* the set of goal states. Conversely, if the reachable set doesn't intersect the goal, then the plan definitely doesn't work. **Figure illustrates these ideas.**

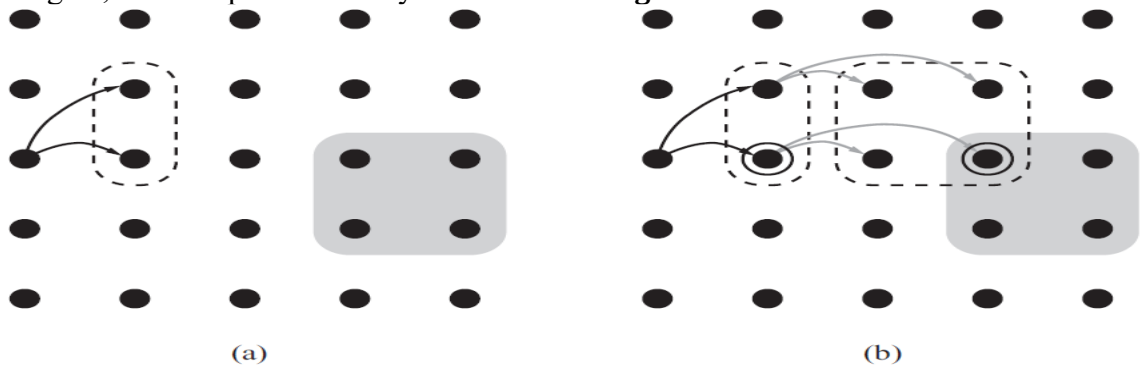


Figure: The set of goal states is shaded. Black and gray arrows indicate possible implementations of h1 and h2, respectively. (a) The reachable set of an HLA h1 in a state s. (b) The reachable set for the sequence [h1, h2]. Because this intersects the goal set, the sequence achieves the goal.

The notion of reachable sets yields a straightforward algorithm: search among high level plans, looking for one whose reachable set intersects the goal; once that happens, the algorithm can *commit* to that abstract plan, knowing that it works, and focus on refining the plan further.

As with the classical action schemas the following changes the *changes* made to each fluent. Think of a fluent as a state variable. A primitive action can *add* or *delete* a variable or leave it *unchanged*. An HLA under angelic semantics can do more: it can *control* the value of a variable, setting it to true or false depending on which implementation is chosen. In fact, an HLA can have nine different effects on a variable: if the variable starts out true, it can always keep it true, always make it false, or have a choice; if the variable starts out false, it can always keep it false, always make it true, or have a choice; and the three choices for each case can be combined arbitrarily, making nine.

Notationally, this is a bit challenging. The symbol ~ means "possibly, if the agent so chooses". Thus, an effect +A means "possibly add A," that is either leave A unchanged or make it true. Similarly -A means " possibly delete A" and $\widetilde{\pm}$ A means "possibly add or delete A."

Thus, the descriptions of HLAs are *derivable*, in principle, from the descriptions of their refinements—in fact, this is required if HLA descriptions are true , such that the downward refinement property holds. Now, consider the following schemas for the HLAs h1 and h2:

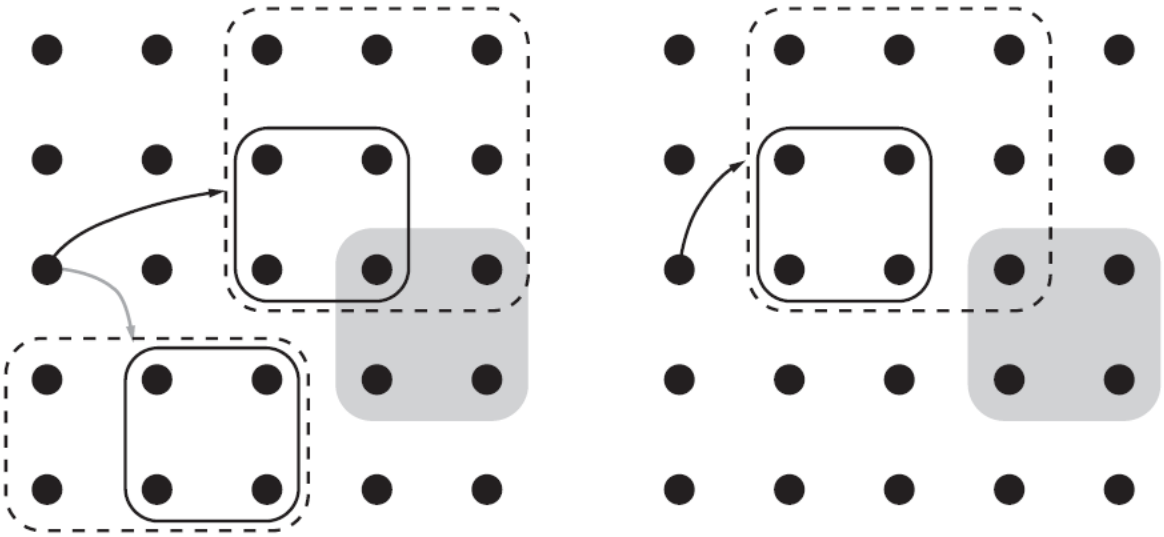
$$Action(h_1, \text{PRECOND:} \neg A, \text{EFFECT:} A \wedge \widetilde{-}B) ,$$
$$Action(h_2, \text{PRECOND:} \neg B, \text{EFFECT:} \widetilde{+}A \wedge \widetilde{\pm}C) .$$

That is, h1 adds A and possible deletes B, while h2 possibly adds A and has full control over C. Now, if only B is true in the initial state and the goal is A ∧ C then the sequence [h1, h2] achieves the goal: Thus an implementation of h1 is chosen that makes B false, then choose an implementation of h2 that leaves A true and makes C true.

HLA makes use of two kinds of approximation: an **optimistic description** $REACH^{+}(s,h)$ of an HLA h may overstate the reachable set, while a **pessimistic description** $REACH^{-}$PESSIMISTIC (s,h) may understate the reachable set. Thus,

$REACH^{-}(s, h) \subseteq REACH(s, h) \subseteq REACH^{+}(s, h)$.

Figure : Goal achievement for high-level plans with approximate descriptions. The set of goal states is shaded. For each plan, the pessimistic (solid lines) and optimistic (dashed lines) reachable sets are shown. (a) The plan indicated by the black arrow definitely achieves the goal, while the plan indicated by the gray arrow definitely doesn't. (b) A plan that would need to be refined further to determine if it really does achieve the goal.



An algorithm for hierarchical planning with approximate angelic descriptions is shown in Figure

function ANGELIC-SEARCH(*problem*, *hierarchy*, *initialPlan*) **returns** solution or *fail*

frontier ← a FIFO queue with *initialPlan* as the only element
loop do
  if EMPTY?(*frontier*) **then return** *fail*
  plan ← POP(*frontier*) /* chooses the shallowest node in *frontier* */
  if REACH$^+$(*problem*.INITIAL-STATE, *plan*) intersects *problem*.GOAL **then**
    if *plan* is primitive **then return** *plan* /* REACH$^+$ is exact for primitive plans */
    guaranteed ← REACH$^-$(*problem*.INITIAL-STATE, *plan*) ∩ *problem*.GOAL
    if *guaranteed*≠{ } and MAKING-PROGRESS(*plan*, *initialPlan*) **then**
      finalState ← any element of *guaranteed*
      return DECOMPOSE(*hierarchy*, *problem*.INITIAL-STATE, *plan*, *finalState*)
    hla ← some HLA in *plan*
    prefix,*suffix* ← the action subsequences before and after *hla* in *plan*
    for each *sequence* **in** REFINEMENTS(*hla*, *outcome*, *hierarchy*) **do**
      frontier ← INSERT(APPEND(*prefix*, *sequence*, *suffix*), *frontier*)

---

function DECOMPOSE(*hierarchy*, s_0, *plan*, s_f) **returns** a solution

solution ← an empty plan
while *plan* is not empty **do**
  action ← REMOVE-LAST(*plan*)
  s_i ← a state in REACH$^-$(s_0, *plan*) such that s_f ∈ REACH$^-$(s_i, *action*)
  problem ← a problem with INITIAL-STATE = s_i and GOAL = s_f
  solution ← APPEND(ANGELIC-SEARCH(*problem*, *hierarchy*, *action*), *solution*)
  s_f ← s_i
return *solution*

The algorithm can detect plans that will and won't work by checking the intersections of the optimistic and pessimistic reachable sets with the goal. When a workable abstract plan is found, the algorithm *decomposes* the original problem into subproblems, one for each step of the plan. The initial state and goal for each subproblem are obtained by regressing a guaranteed-reachable goal state through the action schemas for each step of the plan. The angelic approach can be extended to find least-cost solutions by generalizing the notion of reachable set.

NON-DETERMINISTIC DOMAINS

Planning can also be used to handle partially observable, nondeterministic, and unknown environments. **sensorless planning** (also known as **conformant planning**) for environments with no observations; **contingency planning** for partially observable and nondeterministic environments; and **online planning** and **replanning** for unknown environments.

Consider this problem: given a chair and a table, the goal is to have them match—have the same color. The initial state contains 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:
Init(Object(Table) ∧ Object(Chair) ∧ Can(C1) ∧ Can(C2) ∧ InView(Table))
Goal (Color (Chair, c) ∧ Color (Table, c))

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

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

To solve a partially observable problem, the agent will have to reason about the percepts it will obtain when it is executing the plan. The percept will be supplied by the agent's sensors when it is actually acting, but when it is planning it will need a model of its sensors.

For planning, PDDL is augmented with a new type of schema, the **percept schema**:
Percept (Color (x, c),
PRECOND:Object(x) ∧ InView(x)
Percept (Color (can, c),
PRECOND:Can(can) ∧ InView(can) ∧ Open(can)

The first schema says that whenever an object is in view, the agent. The first schema says that whenever an object is in view, the agent will perceive the color of 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. Because there are no external events in this world, the color of an object will remain the same, even if it is not being perceived, until the agent performs an action to change the object's color. Of course, the agent will need an action that causes objects (one at a time) to come into view:
Action(LookAt (x),
PRECOND:InView(y) ∧ (x _= y)
EFFECT:InView(x) ∧ ¬ InView(y))

For a fully observable environment, a Percept axiom is needed with no preconditions for each fluent. A sensorless agent, on the other hand, has no Percept axioms at all. Note that even a sensorless agent can solve the painting problem. One solution is to open any can of paint and apply it to both chair and table, thus **forcing** them to be the same color (even though the agent doesn't know what the color is)

Sensorless planning
The initial belief state for the sensorless painting problem can ignore InView fluents because the agent has no sensors. Furthermore, the unchanging facts Object(Table) ∧ Object(Chair) ∧Can(C1) ∧ Can(C2) because these hold in every belief state. The agent doesn't know the colors of the cans or the objects, or whether the cans are open or closed, but it does know that objects and cans have colors: ∀ x ∃ c Color (x, c).

After Skolemizing, the initial belief state obtained is:
b0 = Color (x,C(x)) .

In classical planning, where the **closed-world assumption** is made, assume that any fluent not mentioned in a state is false, but in sensorless (and partially observable) planning where the **open-world assumption is considered** in which states contain both positive and negative fluents, and if a fluent does not appear, its value is unknown. Thus, the belief state corresponds exactly to the set of possible worlds that satisfy the formula.
Given this initial belief state, the following action sequence is a solution:
[RemoveLid(Can1), Paint(Chair , Can1), Paint (Table, Can1)] .

In a given belief state b, the agent can consider any action whose preconditions are satisfied by b. According the general formula for updating the belief state b given an applicable action a in a deterministic world is as follows:
b'= RESULT(b, a) = {s : s'=RESULTP (s, a) and s ∈ b}
where RESULTP defines the physical transition model. Assume that the initial belief state is always a conjunction of literals, that is, a 1-CNF formula. To construct the new belief state b', consider what happens to each literal l in each physical state s in b when action a is applied. For literals whose truth value is already known in b, the truth value in b' is computed from the current value and the add list and delete list of the action.
(For example, if l is in the delete list of the action, then ¬ l is added to b'

The literal whose truth value is unknown in b can be found using the three cases:
1. If the action adds l, then l will be true in b' regardless of its initial value.

2. If the action deletes l, then l will be false in b' regardless of its initial value.
3. If the action does not affect l, then l will retain its initial value (which is unknown) and will not appear in b'.
Hence, the calculation of b' is almost identical to the observable case,
b'= RESULT(b, a) = (b − DEL(a))∪ ADD(a) .

The analysis of the update rule has shown a very important fact: *the family of belief states defined as conjunctions of literals is closed under updates defined by PDDL action schemas.* That is, if the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals. That means that in a world with n fluents, any belief state can be represented by a conjunction of size O(n).

For actions involving precondition, action schemas will need : a **conditional effect**. These have the syntax "**when** condition: " where *condition* is a logical formula to be compared against the current state, and *effect* is a formula describing the resulting state.

It is important to understand the difference between preconditions and conditional effects. *All* conditional effects whose conditions are satisfied have their effects applied to generate the resulting state; if none are satisfied, then the resulting state is unchanged. On the other hand, if a *precondition* is unsatisfied, then the action is inapplicable and the resulting state is undefined. From the point of view of sensorless planning, it is better to have conditional effects than an inapplicable action.

Contingent Planning
Contingent planning is the method of generation of plans with conditional branching based on percepts—is appropriate for environments with partial observability, nondeterminism, or both. For the partially observable painting problem with the percept axioms given earlier, one possible contingent solution is as follows:

[LookAt (Table), LookAt (Chair),
if Color (Table, c) ∧ Color (Chair, c) **then** NoOp
else [RemoveLid(Can1), LookAt (Can1),RemoveLid (Can2), LookAt (Can2),
if Color (Table, c) ∧ Color (can, c) **then** Paint(Chair , can)
else if Color (Chair, c) ∧ Color (can, c) **then** Paint(Table, can)
else [Paint(Chair , Can1), Paint (Table, Can1)]]]

Variables in this plan should be considered existentially quantified; the second line says that if there exists some color c that is the color of the table and the chair, then the agent need not do anything to achieve the goal. When executing this plan, a contingent-planning agent can maintain its belief state as a logical formula and evaluate each branch condition by determining if the belief state entails the condition formula or its negation.
The new belief state calculated after an action and subsequent percept is done in two stages. The first stage calculates the belief state after the action, just as for the sensorless agent:
$\hat{b}$= (b − DEL(a))∪ ADD(a)
where, the belief state represented as a conjunction of literals. The second stage is a little trickier. Suppose that percept literals p1, . . . , pk are received and the preconditions are satisfied. Now, if

a percept p has exactly one percept axiom, Percept (p, PRECOND:c), where c is a conjunction of literals, then those literals can be thrown into the belief state along with p. On the other hand, if p has more than one percept axiom whose preconditions might hold according to the predicted belief state $\hat{b}$, then the *disjunction* of the preconditions has to be added. Obviously, this takes the belief state outside 1-CNF and brings up the same complications as conditional effects, with much the same classes of solutions.

Given a mechanism for computing exact or approximate belief states, contingent plans can be with an extension of the AND–OR forward search over belief states. For the heuristic function, many of the methods suggested for sensorless planning are also applicable in the partially observable, nondeterministic case.

Online replanning

Replanning presupposes some form of **execution monitoring** to determine the need for a new plan. One such need arises when a contingent planning agent gets tired of planning for every little contingency. Some branches of a partially constructed contingent plan can simply say Replan; if such a branch is reached during execution, the agent reverts to planning mode.

Replanning may be needed if the agent's model of the world is incorrect. The model for an action may have a **missing precondition**—for example, the agent may not know that removing the lid of a paint can often requires a screwdriver. The model may have a **missing effect**—painting an object may get paint on the floor as well. Or the model may have a **missing fluent** that is simply absent from the representation altogether. The model may also lack provision for **exogenous events** such as someone knocking over the paint can. Exogenous events can also include changes in the goal, such as the addition of the requirement that the table and chair not be painted black. Without the ability to monitor and replan, an agent's behavior is likely to be fragile if it relies on absolute correctness of its model.

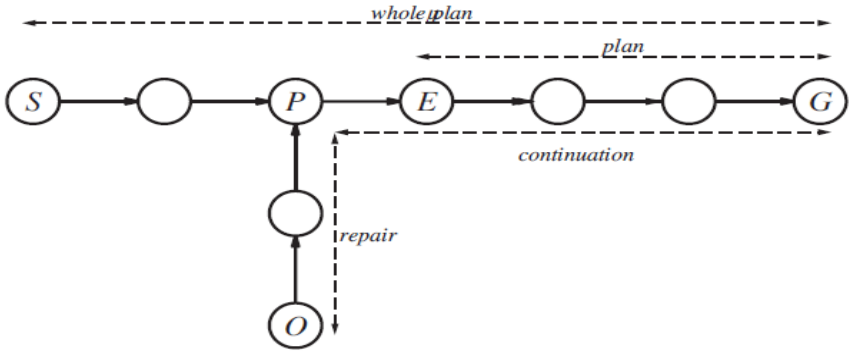
The online agent has a choice of how carefully to monitor the environment. The three levels are:

• **Action monitoring**: before executing an action, the agent verifies that all the preconditions still hold.

• **Plan monitoring**: before executing an action, the agent verifies that the remaining plan will still succeed.

• **Goal monitoring**: before executing an action, the agent checks to see if there is a better set of goals it could be trying to achieve.

Figure shows schematic of action monitoring.



The agent keeps track of both its original plan, whole plan, and the part of the plan that has not been executed yet, which is denoted by plan. After executing the first few steps of the plan, the agent expects to be in state E. But the agent observes it is actually in state O. It then needs to repair the plan by finding some point P on the original plan that it can get back to. (It may be that P is the goal state, G.) The agent tries to minimize the total cost of the plan: the repair part (from O to P) plus the continuation (from P to G).

Action monitoring is a simple method of execution monitoring, but it can sometimes lead to less than intelligent behavior. For example, suppose there is no black or white paint, and the agent constructs a plan to solve the painting problem by painting both the chair and table red. Suppose that there is only enough red paint for the chair. With action monitoring, the agent would go ahead and paint the chair red, then notice that it is out of paint and cannot paint the table, at which point it would replan a repair—perhaps painting both chair and table green. A plan-monitoring agent can detect failure whenever the current state is such that the remaining plan no longer works. Thus, it would not waste time painting the chair red.

Plan monitoring achieves this by checking the preconditions for success of the entire remaining plan—that is, the preconditions of each step in the plan, except those preconditions that are achieved by another step in the remaining plan. Plan monitoring cuts off execution of a doomed plan as soon as possible, rather than continuing until the failure actually occurs. Plan monitoring also allows for serendipity—accidental success. If someone comes along and paints the table red at the same time that the agent is painting the chair red, then the final plan preconditions are satisfied (the goal has been achieved), and the agent can go home early.

It is straightforward to modify a planning algorithm so that each action in the plan is annotated with the action's preconditions, thus enabling action monitoring. It is slightly more complex to enable plan monitoring. Partial-order planners have the advantage that they have already built up structures that contain the relations necessary for plan monitoring. Augmenting state-space planners with the necessary annotations can be done by careful bookkeeping as the goal fluents are regressed through the plan.

Trouble occurs when a seemingly-nondeterministic action is not actually random, but rather depends on some precondition that the agent does not know about. For example, sometimes a paint can may be empty, so painting from that can has no effect. No amount of retrying is going to change this. One solution is to choose randomly from among the set of possible repair plans, rather than to try the same one each time. In this case, the repair plan of opening another can might work. A better approach is to **learn** a better model. Every prediction failure is an opportunity for learning; an agent should be able to modify its model of the world to accord with its percepts. From then on, the replanner will be able to come up with a repair that gets at the root problem, rather than relying on luck to choose a good repair.

TIME, SCHEDULES, AND RESOURCES

Classical planning talks about *what to do*, in *what order*, but does not talk about time: *how long* an action takes and *when* it occurs. For example, in the airport domain a plan states which planes go where, carrying what, but could not specify departure and arrival times. This is the subject

matter of **scheduling**. The real world also imposes **resource constraints**: an airline has a limited number of staff, and staff who are on one flight cannot be on another at the same time. This section introduces techniques for planning and scheduling problems with resource constraints.

The approach considered is "plan first, schedule later": 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. This approach is common in real-world manufacturing and logistical settings, where the planning phase is sometimes automated, and sometimes performed by human experts.

Representing temporal and resource constraints

A typical **job-shop scheduling problem** , consists of a set of **jobs**, each of which has a collection of **actions** with ordering constraints among them. Each action has a **duration** and a set of resource constraints required by the action. A constraint specifies a *type* of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is **consumable** (e.g., the bolts are no longer available for use) or **reusable** (e.g., a pilot is occupied during a flight but is available again when the flight is over). Actions can also produce resources (e.g., manufacturing and resupply actions).

A solution to a job-shop scheduling problem specifies the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, the cost function is assumed as just the total duration of the plan, which is called the **makespan**.

Figure shows a simple example: a problem involving the assembly of two cars.

$Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},$
$\{AddEngine2 \prec AddWheels2 \prec Inspect2\})$

$Resources(EngineHoists(1), WheelStations(1), Inspectors(e2), LugNuts(500))$

$Action(AddEngine1,$ 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:$Inspectors(1))$

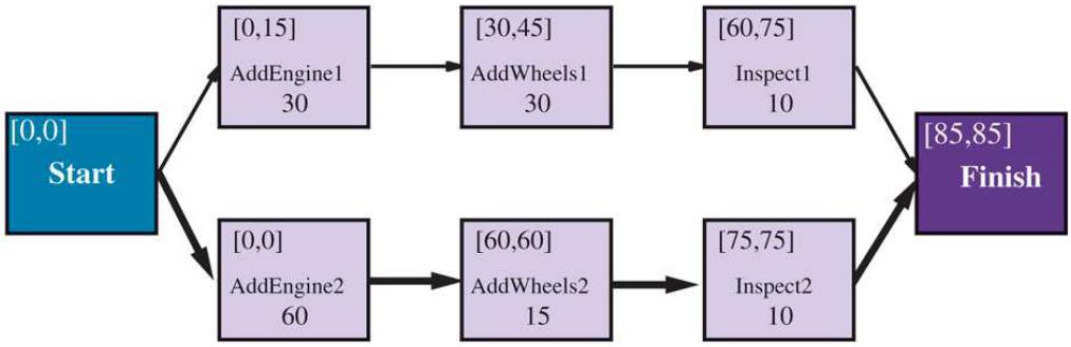
The problem consists of two jobs, each of the form [*AddEngine,AddWheels, Inspect*]. Then the *Resources* statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are *consumed* as wheels

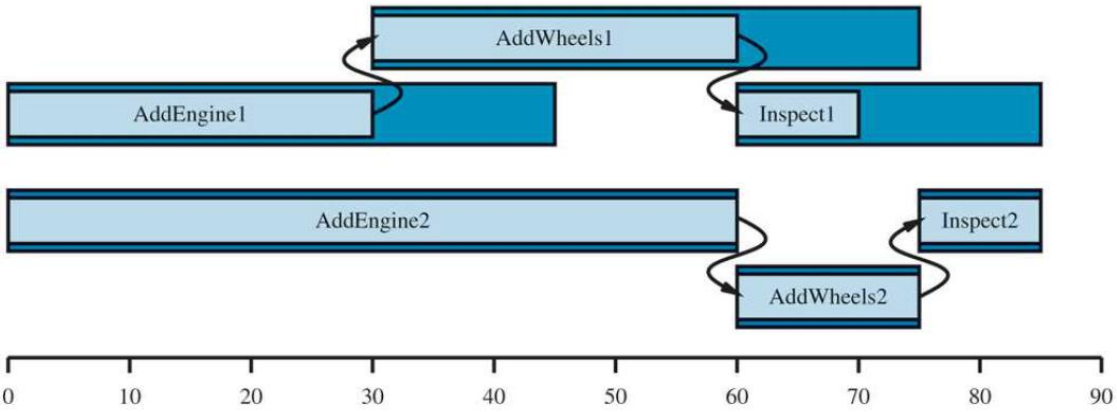
are added to the car, whereas the other resources are "borrowed" at the start of an action and released at the action's end.

The representation of resources as numerical quantities, such as Inspectors (2), rather than as named entities, such as Inspector (I1) and Inspector (I2), is an example of a very general technique called **aggregation**. The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand. In our assembly Aggregation is essential for reducing complexity. Consider what happens when a proposed schedule has 10 concurrent Inspect actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm backtracks to try all 10! ways of assigning inspectors to actions.

Solving scheduling problems

consider the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), the earliest start times has to be found 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.





The **critical path method** (CPM) can be applied to this graph to determine the possible start and end times of each action. A **path** through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with *Start* and ending with *Finish*. (For example, there are two paths in the partial-order plan in Figure.)

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. As in Figure the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition).Together the ES and LS times for all the actions constitute a **schedule** for the problem.

The following formulas serve as a definition for ES and LS and also as the outline of a dynamic-programming algorithm to compute them. A and B are actions, and $A \prec B$ means that A comes before B:

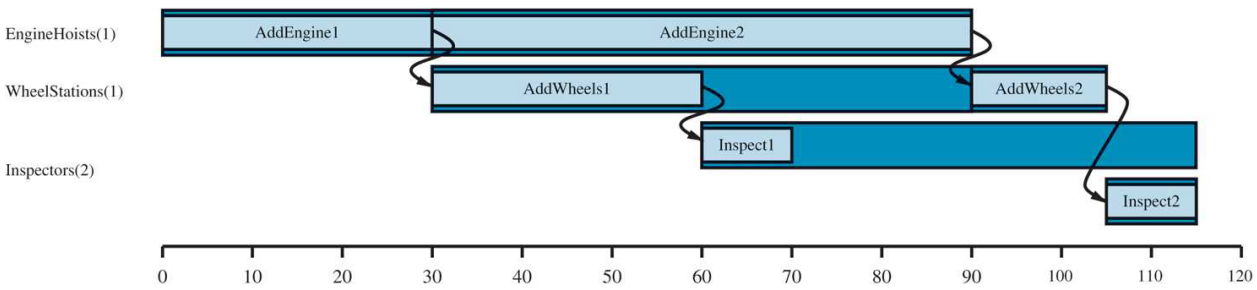
$ES(Start) = 0$
$ES(B) = \max_{A \prec B} ES(A) + Duration(A)$
$LS(Finish) = ES(Finish)$
$LS(A) = \min_{B \prec A} LS(B) - Duration(A)$.

The idea is that ES(Start) is assigned to be 0. Then, as soon as an action B is got such that all the actions that come immediately before B have ES values assigned, hence ES(B) is set to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an ES value. The LS values are computed in a similar manner, working backward from the Finish action.

The complexity of the critical path algorithm is just O(Nb), where N is the number of actions and b is the maximum branching factor into or out of an action. Therefore, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy. Mathematically speaking, critical-path problems are easy to solve because they are defined as a *conjunction* of *linear* inequalities on the start and end times.

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

Figure 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, hence immediately move one of the two inspectors to a more productive position.

The complexity of scheduling with resource constraints is often seen in practice as well as in theory. One simple but popular heuristic is the **minimum slack** algorithm: on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the ES and LS times for each affected action and repeat. The heuristic resembles the minimum-remaining-values (MRV) heuristic in constraint satisfaction.

If a scheduling problem is proving very difficult, however, it may not be a good idea to solve it this way—it may be better to reconsider the actions and constraints, in case that leads to a much easier scheduling problem. Thus, it makes sense to *integrate* planning and scheduling by taking into account durations and overlaps during the construction of a partial-order plan.

Analysis of Planning Approaches

Planning combines the two major areas of AI: *search* and *logic*. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has allowed planners to scale up from toy problems where the number of actions and states was limited to around a dozen, to real-world industrial applications with millions of states and thousands of actions.

Planning is foremost an exercise in controlling combinatorial explosion. If there are propositions in a domain, then there are states. Decomposability is destroyed, however, by negative interactions between actions. SATPLAN can encode logical relations between subproblems. Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent subproblems. Since this approach is heuristic, it can work even when the subproblems are not completely independent.

Unfortunately, there is no clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge, perhaps providing a synthesis of highly expressive first-order and hierarchical representations with the highly efficient factored and propositional representations that dominate today. There are examples of **portfolio** planning

systems, where a collection of algorithms are available to apply to any given problem. This can be done selectively (the system classifies each new problem to choose the best algorithm for it), or in parallel (all the algorithms run concurrently, each on a different CPU), or by interleaving the algorithms according to a schedule.