



Automated Planning

Fundamentals of Artificial Intelligence

MSc in Applied Artificial Intelligence, 2023-24

Contents

- Topics included
 - Knowledge Representation
 - Automated Planning

- These slides were based essentially on the following bibliography:
 - Norvig, P, Russell, S. (2021). Artificial Intelligence: A Modern Approach, 4th Edition. Pearson, ISBN-13: 978-1292401133

Knowledge Representation

How to represent facts about the world?

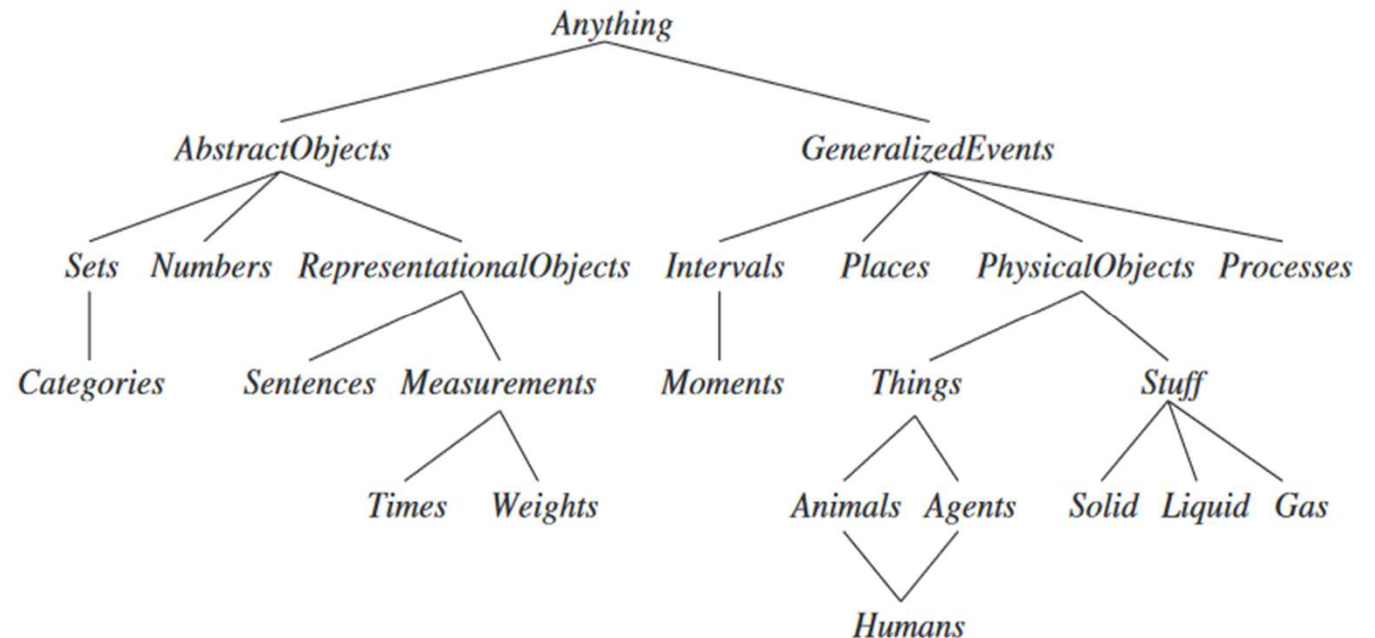
- The facts about the **real world need to be represented** in a form that can be used to reason and solve problems
 - We need to address the question of what content to put into the agent's knowledge base
 - To represent facts about the world we will use different representation formalisms
- For reasoning about plans we will use **first-order logic** as the representation language
 - We will introduce different representation formalisms such as hierarchical task networks
- There are many other representation formalisms such as
 - Bayesian networks for reasoning with uncertainty
 - Markov models for reasoning over time
 - Deep neural networks for reasoning about images, sounds, and other data

Ontological Engineering

- Complex domains such as shopping on the Internet or driving a car in traffic require more general and flexible representations
 - To create these representations, we will concentrate on general concepts, such as Events, Time, Physical Objects, and Beliefs, that occur in many different domains.
 - Representing these abstract concepts is sometimes called **ontological engineering**.
- We will leave placeholders where new knowledge for any domain can fit in
 - For example, we will define what it means to be a physical object, and the details of different types of objects—robots, televisions, books, or whatever—can be filled in later.
 - This is analogous to the way that designers of an object-oriented programming framework (such as the Java Swing graphical framework) define general concepts like *Window*, expecting users to use these to define more specific concepts like *SpreadsheetWindow*.
- 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

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



General-purpose ontology

- We have elected to use first-order logic (FOL) to discuss the content and organization of knowledge, although certain aspects of the real world are hard to capture in FOL.
- The principal difficulty is that most generalizations have exceptions or hold only to a degree.
 - For example, although “tomatoes are red” is a useful rule, some tomatoes are green, yellow, or orange.
 - Similar exceptions can be found to almost all the rules we will see
- The ability to handle exceptions and uncertainty is extremely important, but is orthogonal to the task of understanding the general ontology.
- Of what use is an upper ontology? For example, an ontology for circuits makes many simplifying assumptions:
 - time is omitted completely;
 - signals are fixed and do not propagate;
 - the structure of the circuit remains constant.
- A more general ontology would consider signals at particular times and would include the wire lengths and propagation delays. This would allow us to simulate the timing properties of the circuit, and indeed such simulations are often carried out by circuit designers.

General versus special-purpose ontology

- For any special-purpose ontology, it is possible to make changes like these to move toward greater generality.
- An obvious question then arises: do all these ontologies converge on a general-purpose ontology?
 - After centuries of philosophical and computational investigation, the answer is “Maybe.”
- We will present one general-purpose ontology that synthesizes ideas from those centuries.
- Two major characteristics of general-purpose ontologies distinguish them from collections of special-purpose ontologies:
 - A general-purpose ontology should be applicable in more or less any special-purpose domain. 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.
- A robot circuit-repair system, for instance, needs to reason about circuits in terms of electrical connectivity and physical layout, and about time, both for circuit timing analysis and estimating labor costs.
 - The sentences describing time therefore must be capable of being combined with those describing spatial layout and must work equally well for nanoseconds and minutes and for angstroms and meters

Barriers to ontology creation

- We should say up front that the enterprise of general ontological engineering has so far had only limited success. None of the top AI applications make use of a general ontology
- Social/political considerations can make it difficult for competing parties to agree on an ontology.
 - When competing concerns outweigh the motivation for sharing, there can be no common ontology.
 - The smaller the number of stakeholders, the easier it is to create an ontology, and thus it is harder to create a general-purpose ontology than a limited-purpose one
- Those ontologies that do exist have been created along four routes:
 1. By a team of trained ontologists or logicians, who architect the ontology and write axioms
 2. By importing categories, attributes, and values from an existing database or databases.
 3. By parsing text documents and extracting information from them.
 4. By enticing unskilled amateurs to enter commonsense knowledge.
- As an example, the Google Knowledge Graph uses semistructured content from Wikipedia, combining it with other content gathered from across the web under human curation. It contains over 70 billion facts and provides answers for about a third of Google searches.

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.
 - E.g., a shopper would have the goal of buying a basketball, rather than a particular one, such as BB.
- 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 objects' perceived properties, and then uses category information to make predictions about the objects.
 - E.g., from its green and yellow mottled skin, one-foot diameter, ovoid shape, red flesh, and black seeds, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.
- Turning a proposition into an object is called reification
 - There are two choices for representing categories in first-order logic: predicates and objects.
 - That is, we can use the predicate $\text{Basketball}(b)$, or we can reify the category as an object, Basketballs .
 - We could then say $\text{Member}(b, \text{Basketballs})$, which we will abbreviate as $b \in \text{Basketballs}$, to say that b is a member of the category of basketballs. We say $\text{Subset}(\text{Basketballs}, \text{Balls})$, abbreviated as $\text{Basketballs} \subset \text{Balls}$, to say that Basketballs is a subcategory of Balls .
- We will use subcategory, subclass, and subset interchangeably.

Knowledge organization through inheritance

- Categories organize knowledge through **inheritance**
 - If we say that all instances of the category Food are edible, and if we assert that Fruit is a subclass of Food and Apples is a subclass of Fruit, then we can infer that every apple is edible.
 - The individual apples inherit the property of edibility 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, many of them beetles, into a single hierarchy; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system;
 - Tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products



Facts about categories

- First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members. Here are some example facts
 - An object is a member of a category — $BB_9 \in Basketballs$
 - A category is a subclass of another category — $Basketballs \subset Balls$
 - All members of a category have some properties — $(x \in Basketballs) \Rightarrow Spherical(x)$
 - Members of a category can be recognized by some properties
$$Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$$
 - A category as a whole has some properties — $Dogs \in DomesticatedSpecies$
- Notice that because Dogs is a category and is a member of DomesticatedSpecies, the latter must be a category of categories. Of course, there are exceptions to many of the above rules (punctured basketballs are not spherical); we deal with these exceptions later.

Relations between categories

- Although subclass and member relations are the most important ones for categories, we also want to be able to state relations between categories that are not subclasses of each other.
 - For example, if we just say that Undergraduates and Graduate Students are subclasses of Students, then we have not said that an undergraduate cannot also be a graduate student.
- We say that two or more categories are **disjoint** if they have no members in common
- We may also want to say that the classes undergrad 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
 - $\text{Disjoint}(\{\text{Animals}, \text{Vegetables}\})$
 - $\text{ExhaustiveDecomposition}(\{\text{Americans}, \text{Canadians}, \text{Mexicans}\}, \text{NorthAmericans})$
 - $\text{Partition}(\{\text{Animals}, \text{Plants}, \text{Fungi}, \text{Protista}, \text{Monera}\}, \text{LivingThings})$
- The three predicates are defined as follows:
$$\text{Disjoint}(s) \Leftrightarrow (\forall c_1, c_2 \ c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow \text{Intersection}(c_1, c_2) = \{ \})$$
$$\text{ExhaustiveDecomposition}(s, c) \Leftrightarrow (\forall i \ i \in c \Leftrightarrow \exists c_2 \ c_2 \in s \wedge i \in c_2)$$
$$\text{Partition}(s, c) \Leftrightarrow \text{Disjoint}(s) \wedge \text{ExhaustiveDecomposition}(s, c).$$

Physical composition

- The idea that one object can be part of another is a familiar one. One's nose is part of one's head, Romania is part of Europe, and a chapter is part of a book. We use the general *PartOf* relation to say that one thing is part of another.
- Objects can be grouped into *PartOf* hierarchies, reminiscent of the Subset hierarchy
 - *PartOf*(Bucharest, Romania)
 - *PartOf*(Romania, EasternEurope)
 - *PartOf*(EasternEurope, Europe)
- The *PartOf* relation is transitive and reflexive; that is,
 - $\text{PartOf}(x, y) \wedge \text{PartOf}(y, z) \Rightarrow \text{PartOf}(x, z)$
 - $\text{PartOf}(x, x)$
- Therefore, we can conclude *PartOf*(Bucharest, Europe)
- Categories of composite objects are often characterized by structural relations among parts.
- For example, a biped is an object with exactly two legs attached to a body


$$\begin{aligned} \text{Biped}(a) \Rightarrow & \exists l_1, l_2, b \text{ Leg}(l_1) \wedge \text{Leg}(l_2) \wedge \text{Body}(b) \wedge \\ & \text{PartOf}(l_1, a) \wedge \text{PartOf}(l_2, a) \wedge \text{PartOf}(b, a) \wedge \\ & \text{Attached}(l_1, b) \wedge \text{Attached}(l_2, b) \wedge \\ & l_1 \neq l_2 \wedge [\forall l_3 \text{ Leg}(l_3) \wedge \text{PartOf}(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)] \end{aligned}$$

PartPartition relation and Bunch

- We can define a *PartPartition* relation analogous to the Partition relation for categories.
 - An object is composed of the parts in its *PartPartition* and can be viewed as deriving some properties from it
 - For example, the mass of a composite object is the sum of the masses of the parts.
 - Notice that this is not the case with categories, which have no mass, even though their elements might.
- It is also useful to define composite objects with definite parts but no particular structure.
 - For example, we might want to say “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, we need a new concept, which we will call a **bunch**.
- For example, if the apples are Apple 1 , Apple 2 , and Apple 3 , then $BunchOf(Apple_1, Apple_2, Apple_3)$ denotes the composite object with the three apples as parts (not elements)
 - We can then use the bunch as a normal, albeit unstructured, object. Notice that $BunchOf(\{x\}) = x$
 - Furthermore, $BunchOf(Apples)$ is the composite object consisting of all apples
- We can define BunchOf in terms of the PartOf relation. Obviously, each element of s is part of $BunchOf(s)$

$$\forall x \, x \in s \Rightarrow PartOf(x, BunchOf(s))$$

Measurements

- In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called measures
- Ordinary quantitative measures are quite easy to represent. We imagine that the universe includes abstract “measure objects,” such as the length that is the length of this line segment: 
- We can call this length 1.5 inches or 3.81 centimeters. Thus, the same length has different names in our language. We represent the length with a units function that takes a number as argument
 - If the line segment is called L_1 , we can write $Length(L_1) = Inches(1.5) = Centimeters(3.81)$
 - Conversion is done by equating multiples of one unit to another: $Centimeters(2.54 \times d) = Inches(d)$
- 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)
 - $Weight(BunchOf(Apple\ 1, Apple\ 2, Apple\ 3)) = Pounds(2)$
 - $d \in Days \Rightarrow Duration(d) = Hours(24)$

Objects: Things and stuff

- The real world can be seen as consisting of primitive/ atomic objects and composite objects built from them.
 - By reasoning at the level of large objects such as apples and cars, we can overcome the complexity involved in dealing with vast numbers of primitive objects individually.
 - A significant portion of reality that seems to defy any obvious individuation—division into distinct objects.
 - This portion has the generic name stuff.
- For example, suppose I have some butter and an aardvark in front of me.
 - I can say there is one aardvark, but there is no obvious number of “butter-objects”
 - any part of a butter-object is also a butter-object, at least until we get to very small parts indeed
- This is the major distinction between stuff and things
 - If we cut an aardvark in half, we do not get two aardvarks
- The English language distinguishes clearly between stuff and things
 - We say “an aardvark,” but one cannot say “a butter.”
 - Linguists distinguish between count nouns, such as aardvarks, holes, and theorems, and mass nouns, such as butter, water, and energy. Several competing ontologies claim to handle this distinction.



Events

- Actions, discussed in previous sections, were represented as propositions using successor-state axioms to say
 - a fluent will be true at time $t + 1$ if the action at time t caused it to be true, or
 - if it was already true at time t and the action did not cause it to be false.
- That was for a world in which actions are discrete, instantaneous, happen one at a time, and have no variation in how they are performed
 - there is only one kind of Shoot action, there is no distinction between shooting quickly, slowly, etc.
- But as we move from simplistic domains to the real world, there is a richer range of actions/events to deal with
- Consider a continuous action, such as filling a bathtub.
 - A successor-state axiom can say that the tub is empty before the action and full when the action is done,
 - But it can't talk about what happens during the action
- It also can't easily describe two actions happening at the same time — such as brushing one's teeth while waiting for the tub to fill. To handle such cases we introduce an approach known as event **calculus**

Event calculus

- The objects of event calculus are events, fluents, and time points
- $At(Shankar, Berkeley)$ is a fluent: an object that refers to the fact of Shankar being in Berkeley
- The event E_1 of Shankar flying from San Francisco to Washington, D.C., is described as

$$E_1 \in Flyings \wedge Flyer(E_1, Shankar) \wedge Origin(E_1, SF) \wedge Destination(E_1, DC)$$

where *Flyings* is the category of all flying events

- By reifying events we make it possible to add any amount of arbitrary information about them
 - For example, we can say that Shankar's flight was bumpy with $Bumpy(E_1)$
 - 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.
- To assert that a fluent is actually true starting at some point in time t_1 and continuing to time t_2 , we use the predicate T , as in $T(At(Shankar, Berkeley), t_1, t_2)$
- Similarly, we use $Happens(E_1, t_1, t_2)$ to say that the event E_1 actually happened, starting at time t_1 and ending at time t_2 .

Predicates for event calculus

- The complete set of predicates for one version of the event calculus is:
 - $T(f, t_1, t_2)$ Fluent f is true for all times between t_1 and t_2
 - $Happens(e, t_1, t_2)$ Event e starts at time t_1 and ends at t_2
 - $Initiates(e, f, t)$ Event e causes fluent f to become true at time t
 - $Terminates(e, f, t)$ Event e causes fluent f to cease to be true at time t
 - $Initiated(f, t_1, t_2)$ Fluent f become true at some point between t_1 and t_2
 - $Terminated(f, t_1, t_2)$ Fluent f cease to be true at some point between t_1 and t_2
 - $t_1 < t_2$ Time point t_1 occurs before time t_2

- We can describe the effects of a flying event:
 - $E = Flyings(a, here, there) \wedge Happens(E, t_1, t_2) \Rightarrow$
 $Terminates(E, At(a, here), t_1) \wedge Initiates(E, At(a, there), t_2)$

Time

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

Partition({Moments, ExtendedIntervals}, Intervals)

$i \in \text{Moments} \Leftrightarrow \text{Duration}(i) = \text{Seconds}(0)$

- Next, we invent a time scale and associate points on that scale with moments, giving us absolute times
 - The time scale is arbitrary; we will measure it in seconds and say that the moment at midnight (GMT) on January 1, 1900, has time 0.
 - The **functions Begin and End** pick out the earliest and latest moments in an interval, and the **function Time** delivers the point on the time scale for a moment
 - The **function Duration** gives the difference between the end time and the start time

$\text{Interval}(i) \Rightarrow \text{Duration}(i) = (\text{Time}(\text{End}(i)) - \text{Time}(\text{Begin}(i)))$

$\text{Time}(\text{Begin}(\text{AD1900})) = \text{Seconds}(0)$

$\text{Time}(\text{Begin}(\text{AD2001})) = \text{Seconds}(3187324800)$

$\text{Time}(\text{End}(\text{AD2001})) = \text{Seconds}(3218860800)$

$\text{Duration}(\text{AD2001}) = \text{Seconds}(31536000)$

Interval relations

- To make these numbers easier to read, we also introduce a function *Date*, which takes six arguments (hours, minutes, seconds, day, month, and year) and returns a time point:

$$Time(Begin(AD2001)) = Date(0, 0, 0, 1, Jan, 2001)$$

$$Date(0, 20, 21, 24, 1, 1995) = Seconds(30000000000)$$

- Two intervals **Meet** if the end time of the first equals the start time of the second
- The complete set of interval relations is shown below:

$$Meet(i, j) \Leftrightarrow End(i) = Begin(j)$$

$$Before(i, j) \Leftrightarrow End(i) < Begin(j)$$

$$After(j, i) \Leftrightarrow Before(i, j)$$

$$During(i, j) \Leftrightarrow Begin(j) < Begin(i) < End(i) < End(j)$$

$$Overlap(i, j) \Leftrightarrow Begin(i) < Begin(j) < End(i) < End(j)$$

$$Starts(i, j) \Leftrightarrow Begin(i) = Begin(j)$$

$$Finishes(i, j) \Leftrightarrow End(i) = End(j)$$

$$Equals(i, j) \Leftrightarrow Begin(i) = Begin(j) \wedge End(i) = End(j)$$

Interval relations representation

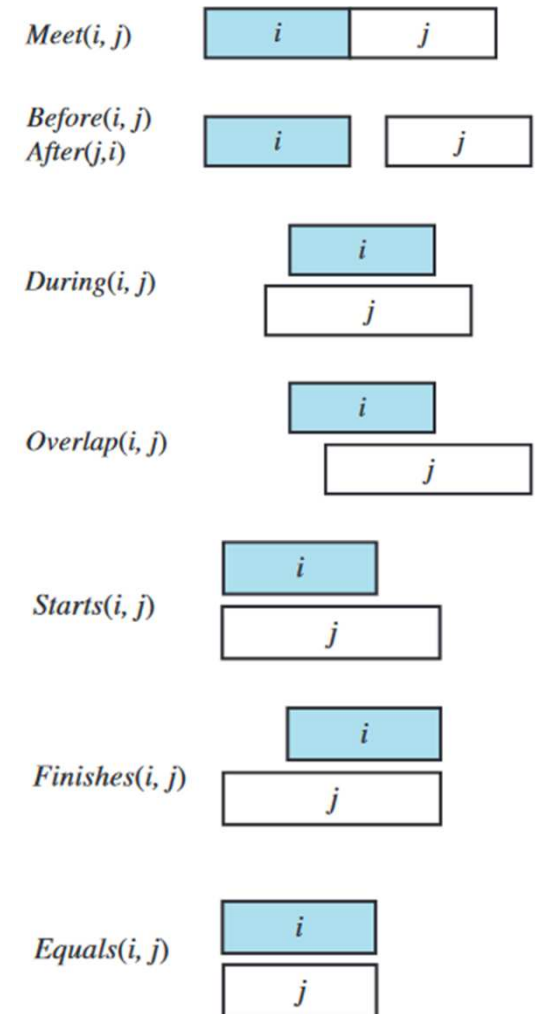
- All interval relations have their intuitive meaning, except for Overlap
- we tend to think of overlap as symmetric (if i overlaps j then j overlaps i), but in this definition, $Overlap(i, j)$ only is true if i begins before j
- Experience has shown that this definition is more useful for writing axioms.
- To say that the reign of Elizabeth II immediately followed that of George VI and the reign of Elvis overlapped with the 1950s, we can write the following

$Meets(ReignOf(GeorgeVI), ReignOf(ElizabethII))$

$Overlap(Fifties, ReignOf(Elvis))$

$Begin(Fifties) = Begin(AD1950)$

$End(Fifties) = End(AD1959)$



Fluents and objects

- Physical objects can be viewed as generalized events, in the sense that a physical object is a chunk of space-time
 - For example, USA can be thought of as an event that began in 1776 as a union of 13 states and is still in progress today as a union of 50.
- We can describe the changing properties of USA using state **fluents**, such as Population(USA)
 - A property of USA that changes every four or eight years, barring mishaps, is its president.
- One might propose that President(USA) is a logical term that denotes a different object at different times
 - This is not possible, because a term denotes exactly one object in a given model structure
 - The term President(USA,t) can denote different objects, depending on the value of t , but our ontology keeps time indices separate from fluents.
 - The only possibility is that President(USA) denotes a single object that consists of different people at different times.
 - It is the object that is George Washington from 1789 to 1797, John Adams from 1797 to 1801, and so on
- To say that George Washington was president during 1790, we can write (function *Equals* is not the same of =)
$$T(\text{Equals}(\text{President}(\text{USA}), \text{GeorgeWashington}), \text{Begin}(\text{AD1790}), \text{End}(\text{AD1790}))$$

Automated Planning

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
- We have seen two approaches: (a) the problem-solving agent, (b) the hybrid propositional logical agent
- Both share **two major limitations**
 - First, they both require ad hoc heuristics for each new domain: a heuristic evaluation function for search, and hand-written code for the hybrid wumpus agent
 - Second, they both need to explicitly represent an exponentially large state space
- For example, in the propositional logic model of the wumpus world, the axiom for moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations
- In response to these limitations, planning researchers have invested in a factored representation using a family of languages called **Planning Domain Definition Language (PDDL)**
 - PDDL allows us to express all $4T n^2$ 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
- The syntax of PDDL is based on Lisp, but we will translate it into a form that matches the notation used so far

PDDL

- In PDDL, a state is represented as a conjunction of ground atomic fluents
 - “ground” means no variables
 - “fluent” means an aspect of the world that changes over time
 - “ground atomic” means there is a single predicate, and if there are any arguments, they must be constants
- For example, $Poor \wedge Unknown$ might represent the state of a hapless agent, and $At(Truck1, Melbourne) \wedge At(Truck2, Sydney)$ could represent a state in a package delivery problem.
- PDDL uses **database semantics**
 - The closed-world assumption means that any fluents that are not mentioned are false
 - The unique names assumption means that *Truck1* and *Truck2* are distinct.
- The following fluents are not allowed in a state for this example:
 - $At(x, y)$, because it has variables
 - $\neg Poor$, because it is a negation
 - $At(Spouse(Ali), Sydney)$, because it uses a function symbol, *Spouse*
- When convenient, we can think of the conjunction of fluents as a set of fluents.

Action schema

- 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),

P RECOND: At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)

E FFECT: \neg At(p, from) \wedge 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 conjunctions of literals, i.e., positive or negated atomic sentences
- We can choose constants to instantiate the variables, yielding a ground (variable-free) action:

Action(Fly(P1, SFO, JFK),

P RECOND :At(P1, SFO) \wedge Plane(P1) \wedge Airport(SFO) \wedge Airport(JFK)

E FFECT : \neg At(P1, SFO) \wedge 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

- The result of executing applicable action a in state s is defined as a state s' . The states are represented by the set of fluents formed by starting with s and ...
 - removing the fluents that appear as negative literals in the action's effects, delete list or $DEL(a)$
 - adding the fluents that are positive literals in the action's effects, add list or $ADD(a)$
$$RESULT(s, a) = (s - DEL(a)) \cup ADD(a)$$
 - E.g., in action $Fly(P_1, SFO, JFK)$, we would remove the fluent $At(P_1, SFO)$ and add the fluent $At(P_1, JFK)$
- 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 $At(C_1, SFO) \wedge \neg At(C_2, SFO) \wedge At(p, SFO)$, refers to any state in which cargo C_1 is at *SFO* but C_2 is not, and in which there is a plane at *SFO*.

Example domain: Air cargo transport

- The right side shows an air cargo transport problem involving loading and unloading cargo and flying it from place to place
- There are three **actions**: Load, Unload, and Fly
- The actions affect two **predicates**:
 - In(c, p) means that cargo c is inside plane p,
 - 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.

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

Example domain: Air cargo transport (2)

- In first-order logic it would be easy to quantify over all objects that are inside the plane
 - But PDDL does not have a universal quantifier, so we need a different solution.
- The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane
 - the cargo only becomes *At* the new airport when it is unloaded
 - So, *At* really means “available for use at a given location.”
- The following plan is a solution to the problem:
*[Load(C1 , P1 , SFO), Fly(P1 , SFO, JFK), Unload(C1 , P1 , JFK),
Load(C2 , P2 , JFK), Fly(P2 , JFK, SFO), Unload(C2 , P2 , SFO)]*

Example domain: The spare tire problem

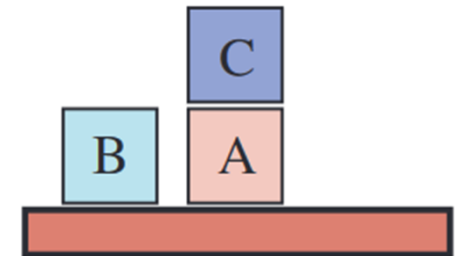
- The goal is to have a good spare tire properly mounted onto the car's axle,
- The initial state has a flat tire on the axle and a good spare tire in the trunk.
- To keep it simple, there are just four actions:
 - removing the spare from the trunk
 - removing the flat tire from the axle
 - putting the spare on the axle
 - leaving the car unattended overnight.
- We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.
- A possible solution to the problem
*[Remove(Flat, Axle),
Remove(Spare, Trunk),
PutOn(Spare, Axle)]*

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

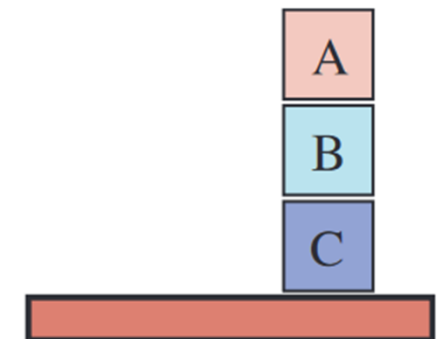
Example domain: The blocks world

- This domain consists of a set of cube-shaped blocks sitting on an arbitrarily-large table. The blocks can be stacked, but only one block can fit directly on top of another.
- A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on top of it.
- A typical goal to get block A on B and block B on C
- One solution is the seq. $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$

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



Start State



Goal State

Example domain: The blocks world (2)

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

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

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

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

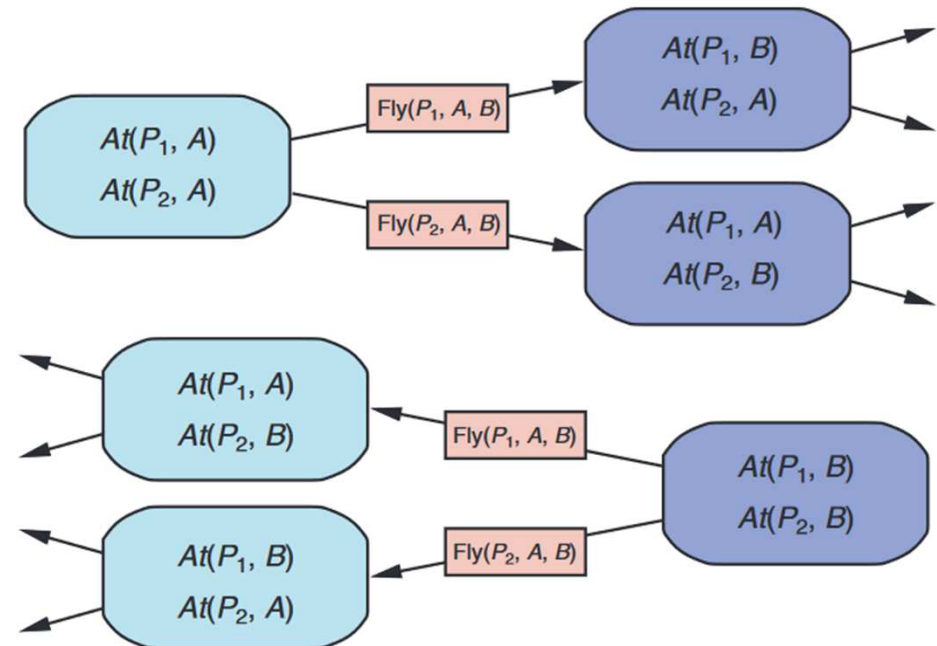
- Unfortunately, this does not maintain $Clear$ properly when x or y is the table

Example domain: The blocks world

- When x is the Table, this action has the effect ($Table$), but the table should not become clear
 - When $y = Table$, it has the precondition $Clear(Table)$
 - But the table does not have to be clear for us to move a block onto it. To fix this, we do two things
- First, we introduce another action to move a block b from x to the table:
 $Action(MoveToTable(b, x),$
 $P\ RECOND : On(b, x) \wedge Clear(b),$
 $E\ FFECT : On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$
- Second, we take the interpretation of $Clear(x)$ to be “there is a clear space on x to hold a block.”
 - Under this interpretation, $Clear(Table)$ will always be true
- However, nothing prevents the planner from using $Move(b, x, Table)$ instead of $MoveToTable(b, x)$
 - We could live with this problem or
 - We could introduce the predicate Block and add $Block(b) \wedge Block(y)$ to 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 we can also **search 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 we can apply a logical inference algorithm to find a solution.
- Figure compares forward and backward searches



Forward state-space search for planning

- We can solve planning problems by applying any of the heuristic search algorithms. The states in this search state space are **ground states**, where every fluent is either true or not.
 - The goal is a state that has all the positive fluents in the problem's goal and none of the negative fluents
 - The applicable actions in a state, Actions(s), are grounded instantiations of the action schemas, i.e., actions where the variables have all been replaced by constant values
- To decide the applicable actions, we unify the current state against the preconditions of each action schema
 - For each unification that successfully results in a substitution, we apply the substitution to the action schema to yield a ground action with no variables
 - It is a requirement of action schemas that any variable in the effect must also appear in the precondition, so we are guaranteed that no variables remain after the substitution
- Each schema may unify in multiple ways. In the spare tire example, the *Remove* action has the precondition *At(obj, loc)*, which matches against the initial state in two ways,
 - resulting in the two substitutions $\{obj|Flat, loc|Axle\}$ and $\{obj|Spare, loc|Trunk\}$
 - applying these substitutions yields two ground actions.
- If an action has multiple literals in the precondition, then each of them can potentially be matched against the current state in multiple ways.

Big state spaces

- Consider an air cargo problem with 10 airports, where each airport initially has 5 planes and 20 pieces of cargo.
 - The goal is to move all the cargo at airport A to airport B.
 - There is a 41-step solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the 20 pieces.
- The average branching factor is huge if you upgrow this problem to 50 planes and 200 pieces
 - each of the 50 planes can fly to 9 other airports, and
 - each of the 200 packages can be either unloaded or loaded into any plane at its airport (if it is unloaded)
- In any state there is a minimum of 450 actions, when all the packages are at airports with no planes, and a maximum of 10,450, when all packages and planes are at the same airport
 - On average, let's say there are about 2000 possible actions per state
 - so the search graph up to the depth of the **41-step solution has about 2000^{41} nodes**
- Clearly, even this relatively small problem instance is hopeless without an accurate heuristic
 - Although many real-world applications of planning have relied on domain-specific heuristics, it turns out that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible

Backward search for planning

- In backward search (also called regression search) we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state
- At each step we consider **relevant actions**
 - In contrast to forward search, which considers actions that are applicable
 - This reduces the branching factor significantly, particularly in domains with many possible actions
- A relevant action is one with an effect that unifies with one of the goal literals, but with no effect that negates any part of the goal. For example, with the goal $\neg \textit{Poor} \wedge \textit{Famous}$
 - An action with the sole effect *Famous* would be relevant
 - An action with the effect $\textit{Poor} \wedge \textit{Famous}$ is not considered relevant
- Even though the second action might be used at some point in the plan (to establish *Famous*), it cannot appear at *this* point in the plan because then *Poor* would appear in the final state

Apply an action in the backward

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

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

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

- 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.
- Some care is required in these equations when there are variables in g and a . For example ...
 - Suppose the goal is to deliver a specific piece of cargo to SFO: $At(C2, SFO)$
 - The Unload action schema has the effect $At(c, a)$
- When we unify that with the goal, we apply the substitution $\{c/C2, a/SFO\}$ to the schema
 - We get a new schema which captures the idea of using any plane that is at SFO

$Action(Unload(C2, p', SFO),$

$PRECOND : In(C2, p') \wedge At(p', SFO) \wedge Cargo(C2) \wedge Plane(p') \wedge Airport(SFO)$

$EFFECT : At(C2, SFO) \wedge \neg In(C2, p'))$

Standardizing apart

- Here we **replaced** p with a new variable named p' . This is an instance of **standardizing apart** variable names so there will be no conflict between different variables that happen to have the same name

- The regressed state description gives us a new goal:

$$g' = In(C2, p') \wedge At(p', SFO) \wedge Cargo(C2) \wedge Plane(p') \wedge Airport(SFO)$$

- As another example, consider the goal of owning a book with a specific ISBN number: $Own(9780134610993)$. Given a trillion 13-digit ISBNs and the single action schema

$$A = Action(Buy(i), PRECOND :ISBN(i), EFFECT :Own(i))$$

- A forward search without a heuristic would have to start enumerating the 10 billion ground Buy actions
- With backward search, we would unify the goal $Own(9780134610993)$ with the effect $Own(i')$, yielding the substitution $\theta = \{i' / 9780134610993\}$.
 - Then we would regress over the action $Subst(\theta, A)$ to yield the predecessor state description $ISBN(9780134610993)$.
 - This is part of the initial state, so we have a solution having considered just one action, not a trillion

Backward search summary

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



Thank you!