UNIT 3

Inference in First-Order Logic

Inference in First-Order Logic is used to **deduce new facts or sentences** from **existing sentences**. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write F[a/x], so it refers to substitute a constant "a" in place of variable "x".

Equality:

First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use equality symbols which specify that the **two terms refer to the same object**.

Example: Brother (John) = Smith.

As in the above example, the object referred by the Brother (John) is similar to the object referred by Smith. The equality symbol can also be used with **negation to represent that two terms are not the same objects**.

Example: ¬(x=y) which is equivalent to x ≠y.

Inference rules for quantifiers

- Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:
- $\forall$ x King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x) .
- Then it seems quite permissible to infer any of the following sentences:
- King(John) $\wedge$ Greedy(John) $\Rightarrow$ Evil(John)
- King(Richard) $\wedge$ Greedy(Richard) $\Rightarrow$ Evil(Richard)
- King(Father (John)) $\wedge$ Greedy(Father (John)) $\Rightarrow$ Evil(Father (John))

- **FOL inference rules for quantifier:**
- As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

1. Universal Generalization
2. Universal Instantiation
3. Existential Instantiation
4. Existential introduction

1. Universal Generalization:

Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).

It can be represented as: $\frac{P(c)}{\forall x\, P(x)}$.

Inference in First-Order Logic.

This rule can be used if we want to show that every element has a similar property.

Example: Let's represent, P(c): "A byte contains 8 bits", so for ∀ x P(x) "All bytes contain 8 bits.", it will also be true.

2. Universal Instantiation:

Universal instantiation is also called as **universal elimination or UI** is a valid inference rule. It can be applied multiple times to add new sentences. The new KB is logically equivalent to old KB

The UI rule state that we can infer any sentence P(c) by substituting a ground term c (a constant within domain x) from ∀ x P(x) for any object in the universe of discourse.

It can be represented as: $\frac{\forall x\, P(x)}{P(c)}$

Inference in First-Order Logic.

Example: 2.

Let's take a famous example,

"All kings who are greedy are Evil."

So let our knowledge base contains this detail as in the form of FOL:

∀x king(x) ∧ greedy (x) → Evil (x),

So from this information, we can infer any of the following statements using Universal Instantiation:

King(John) ∧ Greedy (John) → Evil (John),
King(Richard) ∧ Greedy (Richard) → Evil (Richard),
King(Father(John)) ∧ Greedy (Father(John)) → Evil (Father(John))

3. Existential Instantiation:

Existential instantiation is also called as **Existential Elimination**, which is a valid inference rule in first-order logic. The new KB is not logically equivalent to old KB, but it will be satisfiable if old KB was satisfiable.

This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c. The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.

It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Inference in First-Order Logic

Example:

From the given sentence: ∃x Crown(x) ∧ OnHead(x, John),

So we can infer: Crown(K) ∧ OnHead(K, John), as long as K does not appear in the knowledge base.

The above used K is a constant symbol, which is called Skolem constant.

The Existential instantiation is a special case of Skolemization process.

4. Existential introduction

An existential introduction is also known as an **existential generalization**, which is a valid inference rule in first-order logic.

This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.

It can be represented as: $\frac{P(c)}{\exists x P(x)}$

Inference in First-Order Logic

Example: Let's say that,

"Priyanka got good marks in English."

"Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:

- For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.
- Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."
- According to Modus Ponens, for atomic sentences **pi, pi', q**. Where there is a substitution θ such that SUBST **(θ, pi',) = SUBST(θ, pi)**, it can be represented as:

$$\frac{p1', p2', \ldots, pn', (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}{SUBST(\theta, q)}$$

Ex: **We will use this rule for Kings are evil, so we will find some x such that x is king, and x is greedy so we can infer that x is evil.**

- Here let say, p1' is king(John) p1 is king(x)
- p2' is Greedy(y) p2 is Greedy(x)
- θ is {x/John, y/John} q is evil(x)
- SUBST(θ,q).

Unification:

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\boldsymbol{\Psi_1\sigma = \Psi_2\sigma}$, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**.

Ex: **Find the MGU for Unify{King(x), King(John)}**

Let Ψ_1 = King(x), Ψ_2 = King(John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences.
- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

Ex2: Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.

In this example, we need to make both above statements identical to each other.

P(x,y).....(i)
P(a, f(z))..... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: **[a/x, f(z)/y]**.

Following are some basic conditions for unification:

1. Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
2. Number of Arguments in both expressions must be identical.
3. Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ1, Ψ2)

Step. 1: If Ψ1 or Ψ2 is a variable or constant, then:

a) If Ψ1 or Ψ2 are identical, then return NIL.

b) Else if Ψ1 is a variable,

a. then if Ψ1 occurs in Ψ2, then return FAILURE

b. Else return { (Ψ2/ Ψ1)}.

c) Else if Ψ2 is a variable,

a. If Ψ2 occurs in Ψ1 then return FAILURE,

b. Else return {(Ψ1/ Ψ2)}.

d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return FAILURE.

Step. 3: IF Ψ1 and Ψ2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ1.

a) Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

Forward Chaining/ forward reasoning:

- Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine.
- Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules in the forward direction to extract more data until a goal is reached.
- The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved

Properties of Forward-Chaining:

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.
- Forward-chaining approach is also called as data-driven as we reach to the goal using available data.
- Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Example:

- Rule 1: If A and C then F
- Rule 2: If A and E then G
- Rule 3: If B and E
- Rule 4: If A and D

Prove if A and B is true, then D is true

RESOLUTION

- Steps:
  - Conversion of facts into FOL
  - Convert FOL statement into Normal Forms
  - Negate the statement (by contradiction)
  - Draw resolution graph (Unification – by substitution method)

Conjunctive normal form for first-order logic (Refer pg. 345-347)

- As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals. Literals can contain variables, which are assumed to be universally quantified. For example, the sentence
- ∀ x American(x) ∧Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal (x) becomes, in CNF,
- ¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x, y, z) ∨ ¬Hostile(z) ∨ Criminal (x) .
- *Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.* In particular, the CNF sentence will be unsatisfiable just when the original sentence is unsatisfiable, so we have a basis for doing proofs by contradiction on the CNF sentences.
- We illustrate the procedure by translating the sentence “Everyone who loves all animals is loved by someone,” or
- ∀ x [∀ y Animal(y) ⇒ Loves(x, y)] ⇒ [∃ y Loves(y, x)] .

- The steps are as follows:
- **Eliminate implications**:
- ∀ x [¬∀ y ¬Animal(y) ∨ Loves(x, y)] ∨ [∃ y Loves(y, x)] .
- **Move ¬ inwards**: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have
- ¬∀x p becomes ∃ x ¬p
- ¬∃x p becomes ∀ x ¬p .
- Our sentence goes through the following transformations:
- ∀ x [∃ y ¬(¬Animal(y) ∨ Loves(x, y))] ∨ [∃ y Loves(y, x)] .
- ∀ x [∃ y ¬¬Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ y Loves(y, x)] .
- ∀ x [∃ y Animal (y) ∧¬Loves(x, y)] ∨ [∃ y Loves(y, x)] .

Ex: "As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen**."**

- Prove that **"Robert is criminal.“**
- To solve the above problem, first, convert all the above facts into first-order definite clauses, and then use a forward-chaining algorithm to reach the goal.

Facts Conversion into FOL:

- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ….(1)
- Country A has some missiles. **∃p Owns(A, p) ∧ Missile(p)**. It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.
  Owns(A, T1) ……(2)
  Missile(T1) …….(3)

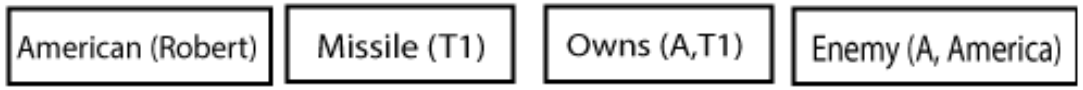
- All of the missiles were sold to country A by Robert.
  ∀p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) **.....(4)**
- Missiles are weapons.
  Missile(p) → Weapons (p) **.....(5)**
- Enemy of America is known as hostile.
  Enemy(p, America) →Hostile(p) **.....(6)**
- Country A is an enemy of America.
  Enemy (A, America) **.....(7)**
- Robert is American
  American(Robert). **.....(8)**

Forward chaining proof:

Step-1:

- In the first step we will start with the known facts and will choose the sentences which do not have implications, such as: **American(Robert), Enemy(A, America), Owns(A, T1), and Missile(T1)**.

All these facts will be represented as below:

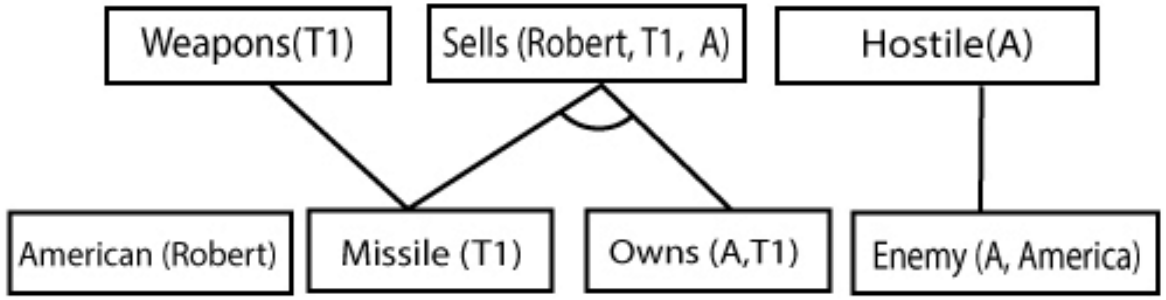


Step-2:

At the second step, we will see those facts which infer from available facts and with satisfied premises. Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

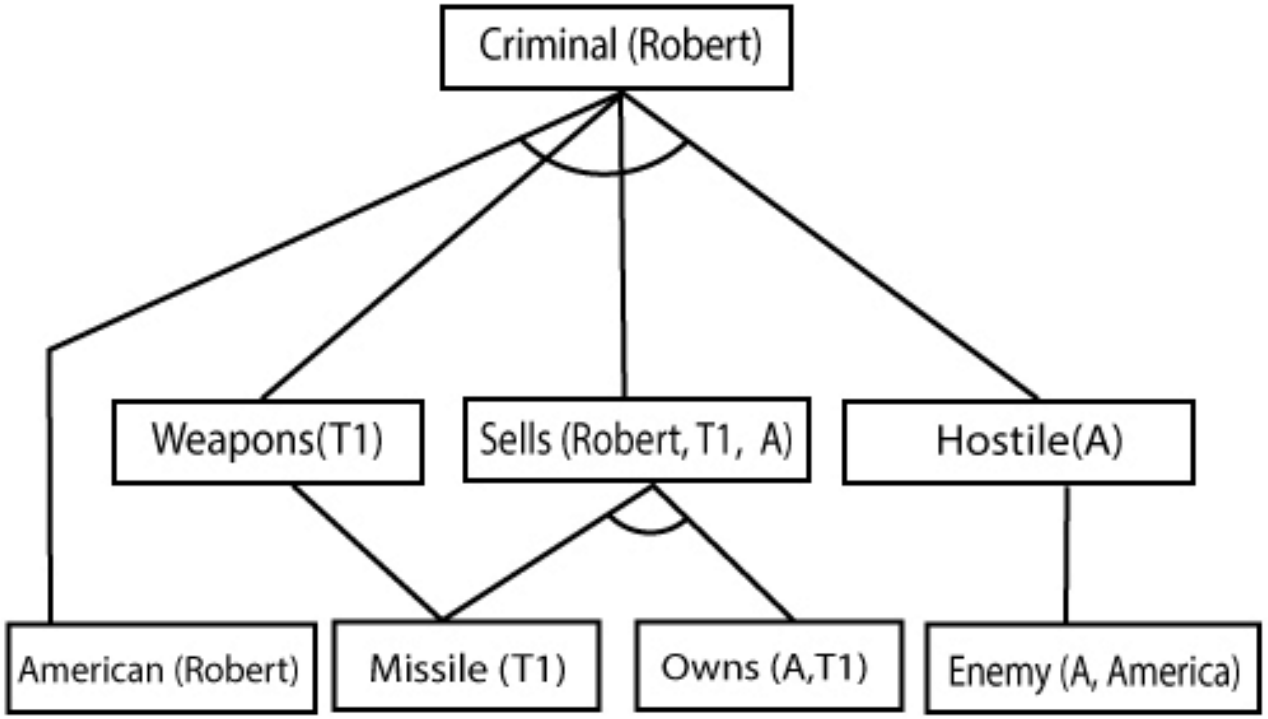
Rule-(2) and (3) are already added.

Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).



Step-3:

- At step-3, as we can check Rule-(1) is satisfied with the substitution **{p/Robert, q/T1, r/A}, so we can add Criminal(Robert)** which infers all the available facts and hence goal statement is reached.



Resolution in FOL

- It is a theorem-proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions.
- Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements.
- Resolution is a single inference rule which can efficiently operate on the **CNF or clausal form**.
- Resolution can resolve two clauses if they contain complementary literals, which are assumed to be standardized apart so that they share no variables.

$$\frac{l_1 \vee \ldots\ldots \vee l_k, \qquad m_1 \vee \ldots\ldots \vee m_n}{\text{SUBST}(\theta, l_1 \vee \ldots\ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots\ldots \vee l_k \vee m_1 \vee \ldots\ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots\ldots \vee m_n)}$$

- This rule is also called the **binary resolution rule** because it only resolves exactly two literals.

Classical planning:

- planning is one of the classic AI problems, study of rational action to achieve a goal.
- it has been used as the basis for applications like controlling robots and having conversations
- a plan is a sequence of actions
- an action is a transformation of a state
- we usually specify an initial state and a goal state, and then try to find (a short!) plan that transform the initial state state in the goal state.

Example Planning Problem: Making Tea

- suppose you have a robot that can serve tea
- think about what the robot must do to make tea, e.g.:
- it must put water in the kettle
- heat the kettle
- get a cup

- pour hot water into the cup (after the water is hot enough)
- get a tea bag
- leave the tea bag in the water for enough time
- remove the tea bag
- add milk
- add sugar
- mix the tea
- serve the tea

there are many, many actions to consider! and most actions consist of many smaller sub-actions and in some situations very long sub-plans might be triggered

- e.g. if there is no milk, the robot might go to the store, or ask if creme would be an acceptable substitute
- e.g. if the robot drops the spoon on the floor, then it must pick it up and clean it before continuing

time and sensing is important as well

- how long should the robot mix the tea for?

Example Planning Problem: Blocks World

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

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

- A and B are on the table, and C is on top of A
- suppose that we want to re-arrange them into this state:

```
 A
 B       goal state
 C
---
```

furthermore, suppose we re-arrange blocks by picking up one block at a time and moving it to the table, or on top of another block

- we can only pick up a block if there is nothing on top if it

it's easy to see that these actions will solve this particular problem:

- move C to the table
- move B on top of C
- move A on top of B

PDDL (Planning Domain Definition Language) is a standardized language used to model and solve automated planning problems in artificial intelligence. It provides a formal way to describe the components of a planning problem, including the **domain** (general rules and actions) and the **problem** (specific instance with initial state and goal).

Components of a PDDL planning task:

- Objects: Things in the world that interest us.
- Predicates: Properties of objects that we are interested in; can be true or false.
- Initial state: The state of the world that we start in.
- Goal specification: Things that we want to be true. .
- Actions/Operators: Ways of changing the state of the world.

A set of ground (variable-free) actions can be represented by a single **action schema.** 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 effect of an action are each conjunctions of literals (positive or negated atomic sentences).
  - The precondition defines the states in which the action can be executed,
  - Effect defines the result of executing the action

- State Representation:
- States are expressed as conjunctions of fluent—ground, functionless atoms (e.g., At(Truck1, Melbourne)).
- Uses database semantics:
    - Closed-world assumption: Any fluent not mentioned is false.
    - Unique names assumption: Different objects (e.g., Truck1, Truck2) are distinct.
- Action Representation:
- Actions are defined using schemas, which are high-level representations combining preconditions and effects.
- Example schema:

Action(Fly(p, from, to), PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to), EFFECT: ¬At(p, from) ∧ At(p, to))

- Preconditions must be satisfied for actions to execute. Effects describe changes in the state.

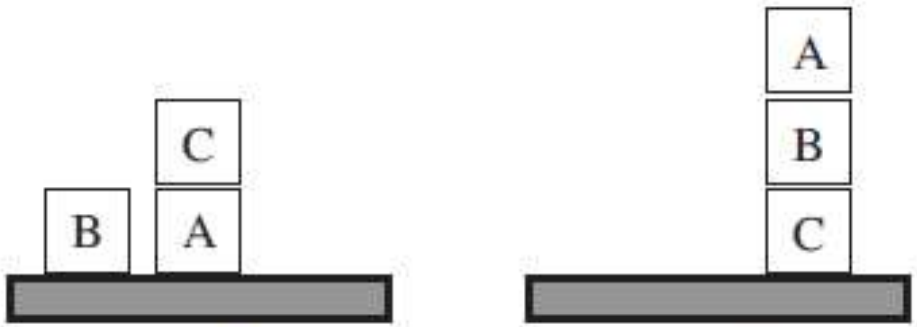
- Set Semantics:
- States can be treated either as logical conjunctions or sets for easier manipulation.
- Action Execution:
- The result of applying an action (RESULT(s, a)) involves:
    - Removing fluents in the action's delete list (DEL(a)).
    - Adding fluents in the action's add list (ADD(a)).
- Frame Problem:
- PDDL handles the frame problem by describing only what changes in an action; things that remain unchanged are left unmentioned.
- Planning Problem:
- Defined by:
    - Initial state: A conjunction of ground fluents.
    - Goal: A conjunction of literals (treated as existentially quantified).
- The objective is to find a sequence of actions that transitions from the initial state to a goal state.
- Propositionalization:
- Complex problems can be converted into propositional logic by grounding all action schemas. However, this is computationally impractical for large domains.
- Temporal Representation:
- PDDL implicitly handles time in action schemas, avoiding explicit temporal notations (e.g., superscripts for time).

Example: The blocks world

- The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it.

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

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



- On(b, x): To indicate that block b is on x, where x is either another block or the table.
- Move(b, x, y): action for moving block b from the top of x to the top of y.
- Clear(x): preconditions on moving b is that no other block be on it. In FOL, this would be ¬∃ x On(x, b) or, ∀ x ¬On(x, b). However, PDDL doesn't support the quantifiers. This is represented using clear(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)) .

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 fix this, we do two things. First, we introduce another action to move a block b from x to the table:

Action(MoveToTable(b, x),

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

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

Init (At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK)

∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)

∧ Airport (JFK) ∧ Airport (SFO))

Goal (At(C1, JFK) ∧ At(C2, SFO))

Action(Load (c, p, a),

PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport (a)

EFFECT: ¬ At(c, a) ∧ In(c, p))

Action(Unload(c, p, a),

PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport (a)

EFFECT: At(c, a) ∧ ¬In(c, p))

Action(Fly(p, from, to),

PRECOND: At(p, from) ∧ Plane(p) ∧ Airport (from) ∧ Airport (to)

EFFECT: ¬ At(p, from) ∧ At(p, to))

ALGORITHMS FOR PLANNING AS STATE-SPACE SEARCH:

Forward (progression) & backward state-space search:

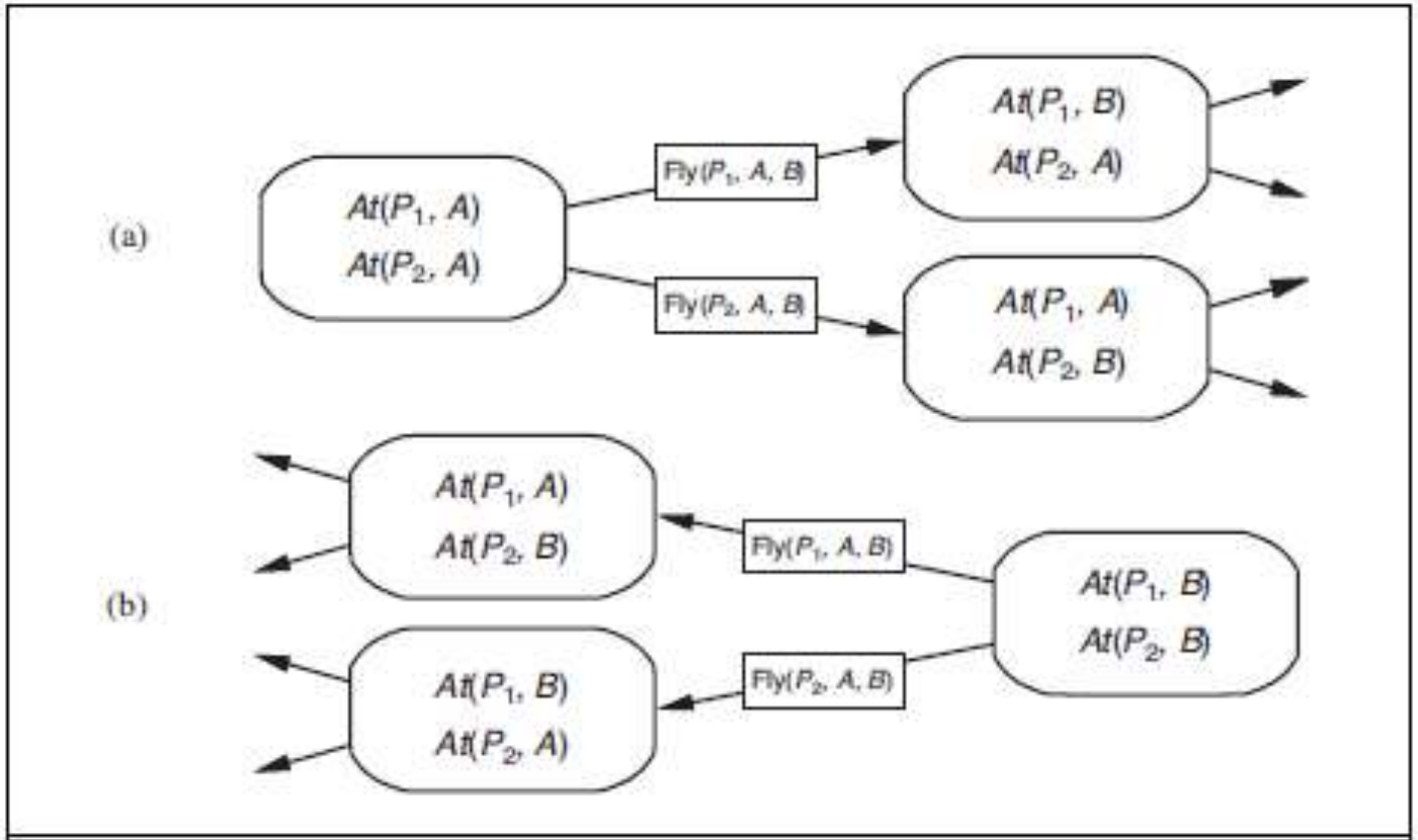


Figure 10.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

Forward search is prone to exploring irrelevant actions, ex:

- Consider the noble task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is an action schema Buy(isbn) with effect Own(isbn). ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.
- Second, planning problems often have large state spaces. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B.

Backward (regression) relevant-states search:

- In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial RELEVANT-STATES state. It is called **relevant-states** search because we only consider actions that are relevant to the goal.

- In general, backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the n-queens problem because there is no easy way to describe the states that are one move away from the goal.

Heuristics for planning:

- Neither forward nor backward search is efficient without a good heuristic function.
- a heuristic function h(s) estimates the distance from a state s to the goal and that if we can derive an **admissible** heuristic for this distance—one that does not overestimate.

PLANNING GRAPHS(PG):

- PG tells how special data structure called a **planning graph** can be used to give better heuristic estimates.
- This section shows how a special data structure called a **planning graph** can be used to give better heuristic estimates.
- The planning graph can’t answer definitively whether G is reachable from S0, but it can *estimate* how many steps it takes to reach G.
- A planning graph is a directed graph organized into **levels**: first a level S0 for the initial state, consisting of nodes representing each fluent that holds in S0;
- then a level A0 consisting of nodes for each ground action that might be applicable in S0; then alternating levels Si followed by Ai; until we reach a termination condition.

Ex:

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake))$

Figure 10.7 The "have cake and eat cake too" problem.

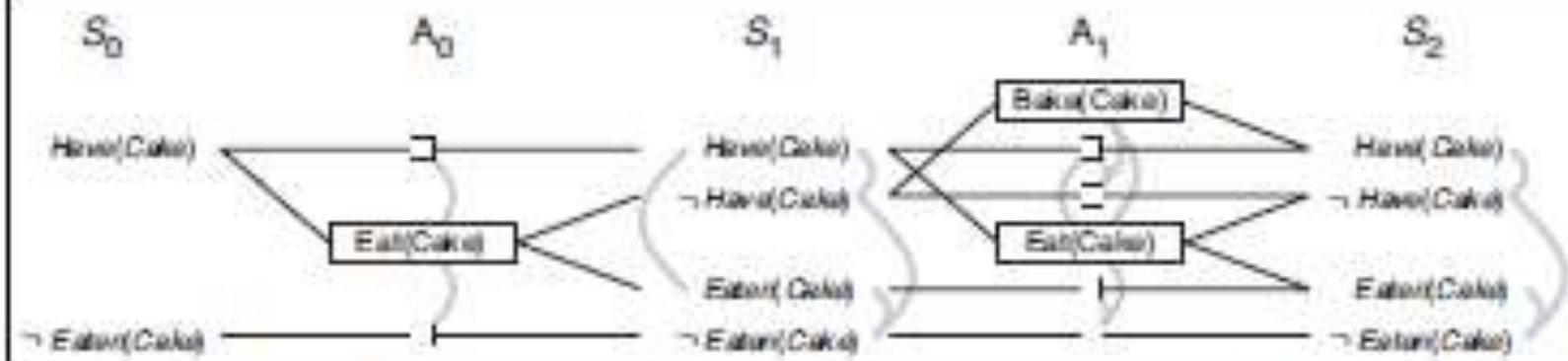


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i and we need not draw that mutex link.

UNIT IV

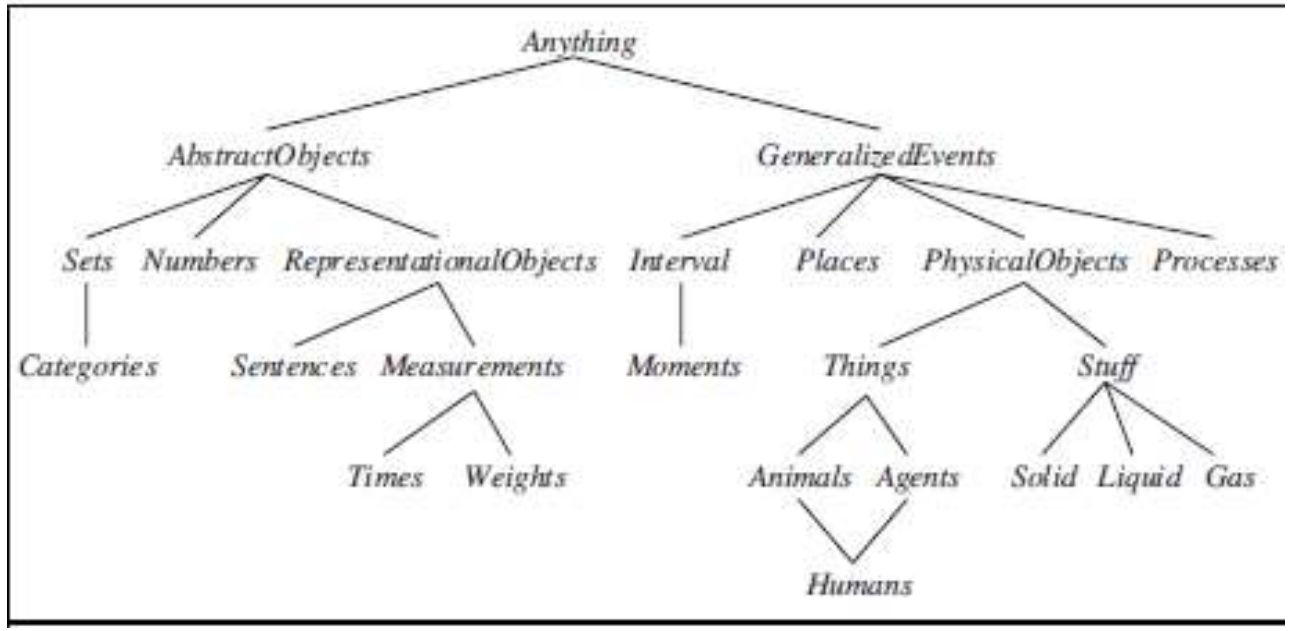
Knowledge Representation:

ONTOLOGICAL ENGINEERING:

- Complex domains such as shopping on the Internet or driving a car in traffic require more general and flexible representations.
- This chapter shows how to create these representations, concentrating on general concepts—such as *Events, Time, Physical Objects*, and *Beliefs*— that occur in many different domains.
- Representing these abstract concepts is sometimes called **ontological engineering**.

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 12.1

Ex: 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 an example



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 (with the addition of domain-specific axioms).
- 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.*
- For example, a shopper would normally have the goal of buying a basketball, rather than a *particular* basketball such as BB9.
- Categories also serve to make predictions about objects once they are classified.

There are two choices for representing categories in first-order logic: predicates and objects.

- That is, we can use the predicate Basketball (b), or we can **reify** the category as an object, Basketballs.
- Member(b, Basketballs), abbreviated as b∈ Basketballs - b is a member of the category of basketballs.

- Subset(Basketballs, Balls), abbreviated as Basketballs $\subset$ Balls - Basketballs is a **subcategory** of Balls.
- Categories serve to organize and simplify the knowledge base through **inheritance.**

  Ex: all instances of the category Food are edible
- Subclass relations organize categories into a **taxonomy**, or **taxonomic hierarchy.**
- FOL makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members. Here are some types of facts, with examples of each:
- 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) ∧ Round (x) ∧ Diameter(x)=9.5 ∧ x∈ Balls ⇒ x∈ Basketballs
- A category as a whole has some properties.
  - Dogs ∈DomesticatedSpecies
- We say that two or more categories are **disjoint** if they have no members in com mon.
  - Disjoint({Animals,Vegetables})
  - ExhaustiveDecomposition({Americans,Canadians, Mexicans},
  - NorthAmericans)
  - Partition({Males, Females}, Animals)

Physical composition:

General **PartOf** relation to say that one thing is part of another. Objects can be grouped into partOf hierarchies, reminiscent of the Subset hierarchy:

- PartOf (Bucharest , Romania)
- PartOf (Romania, EasternEurope)
- PartOf (EasternEurope, Europe)
- PartOf (Europe, Earth).

- The PartOf relation is transitive and reflexive; that is,

$$\text{PartOf}(x, y) \wedge \text{PartOf}(y, z) \Rightarrow \text{PartOf}(x, z) .$$

$$\text{PartOf}(x, x) .$$

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 MEASURE properties are called **measures**.

- If the line segment is called L1, we can write

  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).
- Measures can be used to describe objects as follows:
  - Diameter (Basketball 12)=Inches(9.5) .
  - ListPrice(Basketball 12)=$(19) .
  - d∈ Days ⇒ Duration(d)=Hours(24) .

EVENTS:

- To handle actions happening at the same time, we introduce an alternative formalism known as **event calculus**, which is based on points of time rather than on situations.

Event calculus reifies **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 we use the predicate T, as in 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

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

- If this is too verbose, alternative three-argument version of the category of flying events and say

  $E1 \in Flyings(Shankar , SF,DC)$.

- 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

Processes:

The events we have seen so far are what we call **discrete events**.

- Shankar's trip has a beginning, middle, and end. If interrupted halfway, the event would be something different.
- If we take a small interval of Shankar's flight, say, the third 20-minute segment, that event is still a member of Flyings. In fact, this is true for any subinterval.
- 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:

$(e \in \text{Processes}) \land \text{Happens}(e, (t1, t4)) \land (t1 < t2 < t3 < t4) \Rightarrow \text{Happens}(e, (t2, t3))$.

Acting under Certanity

- Agents may need to handle uncertainty, whether due to partial observability, nondeterminism, or a combination of the two.
- An agent may never know for certain what state it's in or where it will end up after a sequence of actions.
- Suppose, for example, that an automated taxi! automated has the goal of delivering a passenger to the airport on time. The agent forms a plan, A90, that involves leaving home 90 minutes before the flight departs and driving at a reasonable speed.
- Even though the airport is only about 5 miles away, a logical taxi agent will not be able to conclude with certainty that "Plan A90 will get us to the airport in time."

Summarizing uncertainty

Let's consider an example of uncertain reasoning: diagnosing a dental patient's toothache.

Let us try to write rules for dental diagnosis using propositional logic, so that we can see how the logical approach breaks down.

Consider the following simple rule:

$$\text{Toothache} \Rightarrow \text{Cavity}$$

The rule is wrong. Not all patients with toothaches have cavities; some of them have gum disease, an abscess, or one of several other problems:

$$\text{Toothache} \Rightarrow \text{Cavity} \vee \text{GumProblem} \vee \text{Abscess}$$

We could try turning the rule into a causal rule:

$$\text{Cavity} \Rightarrow \text{Toothache}$$

The rule is wrong.

Classical Planning:

- AI Classical planning is a key area in Artificial Intelligence to find a sequence of actions that will fulfil a specific goal from an exact beginning point.
- This process creates methods and algorithms that allow smart systems to explore systematically various actions and their outcomes which eventually lead to the desired result occasionally from the starting place.
- There are many different areas in which classical planning methods are used such as: robotics or other industries related to manufacturing, transportation services like supply chain management or project

Planning with State-Space Search

- The most straightforward approach of planning algorithm, is state-space search
  - Forward state-space search (Progression)
  - Backward state-space search (Regression)
- The descriptions of actions in a planning problem, and specify both preconditions and effects
- It is possible to search in both direction: either forward from the initial state or backward from the goal
- We can also use the explicit action and goal representations, to derive effective heuristics automatically.

Problem Formulation for Progression

- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

Progression

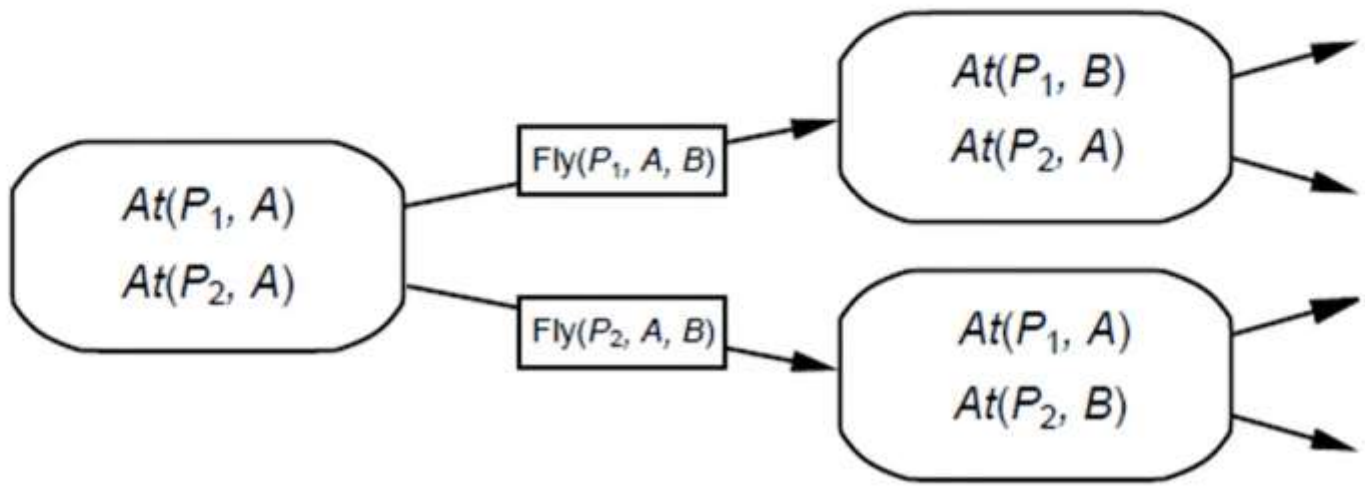
- From initial state, search forward by selecting operators whose preconditions can be unified with literals in the state
- New state includes positive literals of effect; the negated literals of effect are deleted
- Search forward until goal unifies with resulting state
- This is forward state-space search using STRIPS operators

Example : Transportation of air cargo between airports.

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

Forward (progression) state-space search

- Starting from the initial state and using the problem's actions to search forward for the goal state.



FSSS Algorithm

- (1) compute whether or not a state is a goal state,
- (2) find the set of all actions that are applicable to a state, and
- (3) compute a successor state, that is the result of applying an action to a state
- Algorithm takes as input the statement P = (O, s0, g) of a planning problem P. (O contains a list of actions)
- If P is solvable, then Forward-search(O, s0, g) returns a solution plan; otherwise it returns failure.

Algorithm for FSSS

1. Forward-search(O, s0, g)
2. $s \leftarrow s0$
3. $\pi \leftarrow$ the empty plan
4. loop
   1. if s satisfies g then return π
   2. applicable $\leftarrow$ {a | a is a ground instance of an operator in O, and precond(a) is true in s}
   3. if applicable = $\emptyset$ then return failure
   4. Non-deterministically choose an action a $\in$ applicable
   5. $s \leftarrow \gamma(s, a)$
   6. $\pi \leftarrow \pi.a$

Problem Formulation for Regression

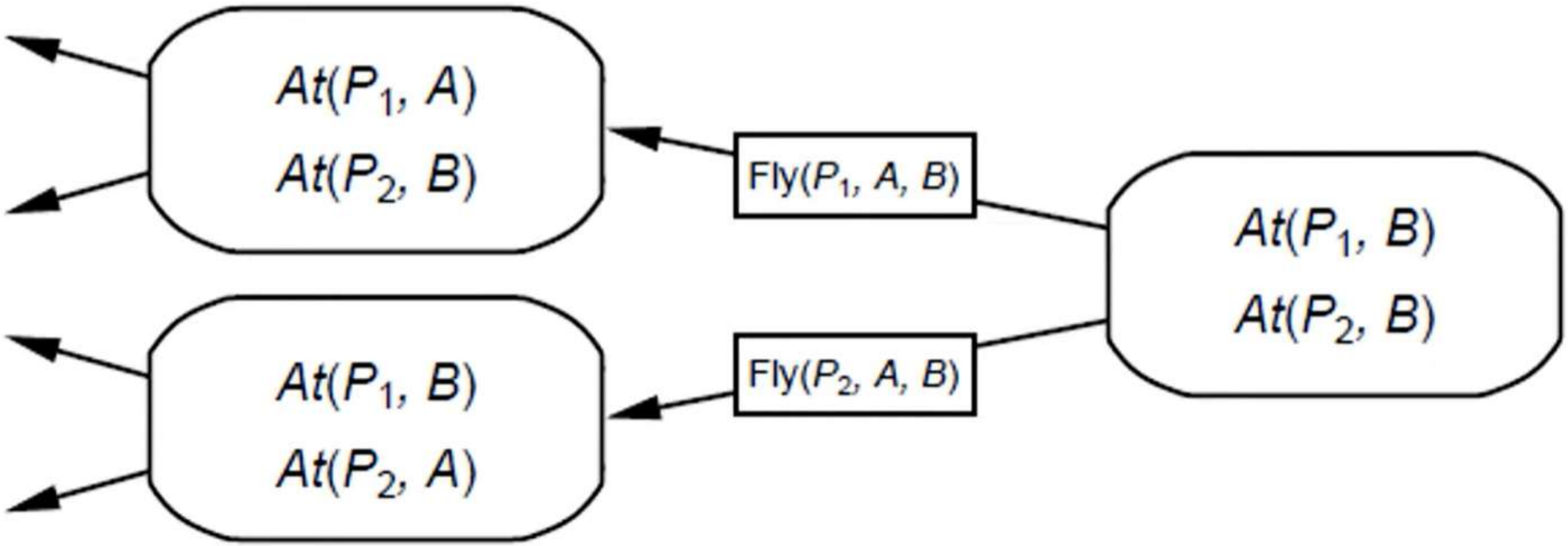
- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

Regression

- The goal state must unify with at least one of the positive literals in the operator's effect
- Its preconditions must hold in the previous situation, and these become subgoals which might be satisfied by the initial conditions
- Perform backward chaining from goal
- Again, this is just state-space search using STRIPS operators

Backward (regression) state-space search:

- Backward state search starting from the goal state(s)
- Using the inverse of the actions to search backward for the initial state.



BSSS Algorithm

- Start at the goal,
- Test the goal is initial state, otherwise
- Apply inverses of the planning operators to produce subgoals,
- The algorithm will stop, if we produce a set of subgoals that satisfies the initial state.

BSSS Algorithm

1. Backward-search(*O*, *s*0, *g*)
2. π the empty plan
3. loop
   1. if *s*0 satisfies *g* then return π
   2. *applicable* $\leftarrow$ {*a* | *a* is a ground instance of an operator in *O* that is relevant for *g*}
   3. if *applicable* = $\emptyset$ then return failure
   4. Non-deterministically choose an action $a \in applicable$
   5. $\pi \leftarrow a.\ \pi$
   6. $g \leftarrow \gamma^{-1}(g, a)$

Planning graph

- The Graph Plan's input is planning problem, expressed in STRIPS and produces a sequence of operations for reaching a goal state.
- Convert the planning problem structure into planning graph called as GRAPHPLAN, in the increment nature.
- It gives the relation between action and states, the precondition must be satisfy the action.
- The Planning graph is a layered graph, with alternate layers of propositions and actions.
  - Layer p0
  - Layer a1
  - Layer p1

Planning Graph…

- Propositional problem will look at, what the starting state, what the objects in the domains are, and it will produce all the possible actions, and works with those actions.
- We construct the planning graph from left to right,
  - we keep inserting actions and propositions, and
  - then actions and propositions
  - until we get the goal proposition appear on the proposition layer, and
  - they are not mutually exclusive.

Planning Graph…

- There are two states in the planning graph problem
  - Construct the planning graph
  - Search for solution
- If you cannot get solution, then extend the planning graph and search for solution, and keep doing that until you get the solution.

Example - The "have cake and eat cake too" problem.

$$Init(Have(Cake))$$
$$Goal(Have(Cake) \land Eaten(Cake))$$
$$Action(Eat(Cake)$$
$$\text{PRECOND: } Have(Cake)$$
$$\text{EFFECT: } \neg Have(Cake) \land Eaten(Cake))$$
$$Action(Bake(Cake)$$
$$\text{PRECOND: } \neg Have(Cake)$$
$$\text{EFFECT: } Have(Cake)$$

EVENT CALCULAS

- Event calculus is a powerful formalism for representing continuous actions and events in a more temporal context.
- Unlike situation calculus, which struggles with continuous or overlapping actions, event calculus uses time intervals to handle such complexities.
- Event calculus addresses these limitations by using time points instead of situations. It introduces fluents and events:
  - Fluents: Facts that can change over time (e.g., At(Shankar, Berkeley)), asserted true using T(fluent, time).
  - Events: Instances of event categories (e.g., Flying(Shankar, SF, DC)), which occur over time intervals.

Core Concepts

1. **Time Representation**:
   - Events occur over time intervals (e.g., (start, end) pairs).
   - Predicates like T, Happens, Initiates, and Terminates describe the state of fluents and events.
2. **Event Predicates:**
   - Happens(e, i): Event e occurs over interval i.
   - 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 become false at time t.
3. **Clipped and Restored**:
   - Clipped(f, i): Fluent f ceases to be true during interval i.
   - Restored(f, i): Fluent f becomes true during interval i.
4. **Fluent Truth Axioms**:
   - A fluent is true (T(f, t)) if it was initiated by an event and not later clipped.
   - A fluent is false if terminated and not restored.

Types of events:

1. Simultaneous Events (e.g., two people riding a seesaw).
2. Exogenous Events (e.g., wind affecting object locations).
3. Continuous Events (e.g., water level in a filling tub).

Mental Events and Mental Objects

- Agents need knowledge about their own reasoning processes and those of others to better control inference and interactions. For example:
  - **Self-knowledge**: Bob realizing he can compute a square root but not know his mother's current posture.
  - **Knowledge of others**: Understanding that Bob's mother knows whether she's sitting.
- To model this, we use **mental objects** (beliefs, knowledge) and **mental processes** (deduction, reasoning).

Propositional Attitudes

Propositional attitudes describe an agent's stance toward a statement, such as:

- **Believes**, **Knows**, **Wants**, **Intends**, **Informs**.

- However, these attitudes face challenges:

1. **Referential Opacity**: Terms used in statements matter because different agents may not know which terms are co-referential.
    1. Example: John knows "Superman can fly" but may not know "Clark Kent can fly," even if Superman = Clark Kent.

Modal Logic

Modal logic addresses referential opacity by introducing **modal operators** that describe an agent's mental state:

- Syntax: KAP means "Agent A knows proposition P."
- Semantics:
  - Truth is evaluated across **possible worlds**.
  - Accessibility relations define which worlds are consistent with an agent's knowledge.

Key Features of Modal Logic:

1.Possible Worlds: Each world represents a different scenario. Accessibility relations determine what an agent knows or does not know.

2.Nested Knowledge: Allows reasoning about what one agent knows about another's knowledge.

- Example: John knows that Superman knows his identity, even if she doesn't know the identity herself.

3.Ambiguity Resolution: Modal logic disambiguates statements like "Bond knows someone is a spy":

- ∃x KBondSpy(x): Bond knows a specific person is a spy.
- KBond∃x Spy(x): Bond knows there is at least one spy.

Reasoning Systems for Categories

1. **Categories in Knowledge Representation:**

Categories are foundational for structuring and reasoning in knowledge representation systems. Two prominent approaches for organizing and reasoning with categories are

- **semantic networks** and
- **description logics**.

Semantic Networks

- **Overview**: Semantic networks use nodes and edges to represent knowledge graphically. They express objects, categories, and relationships among them.
- **Structure**: Nodes represent objects or categories, and labeled edges indicate relationships
- (e.g., "Mary ∈ FemalePersons" or "SisterOf(Mary, John)"). Categories can be linked using "SubsetOf" or other relationships.

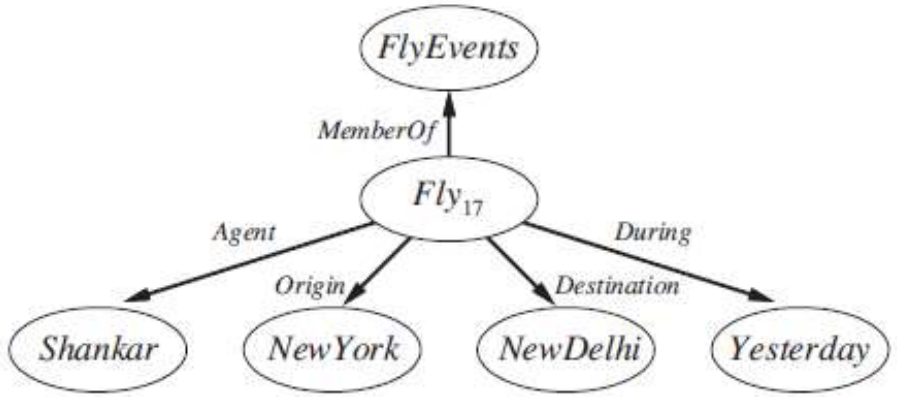


Figure 12.6 A fragment of a semantic network showing the representation of the logical assertion $Fly(Shankar, NewYork, NewDelhi, Yesterday)$.

Key Features

- **Inheritance Reasoning**: Properties are inherited by objects from their categories using links (e.g., "Persons" have two legs, so "Mary" inherits that property unless overridden).
- **Default Values**: Allows categories to have default properties that can be overridden by specific information (e.g., John has one leg, overriding the default property of "two legs").
- **Multiple Inheritance**: Objects or categories can belong to multiple categories, which might lead to conflicting values.
- **Limitations**: Semantic networks are limited to binary relations and cannot directly represent more complex (n-ary) assertions like "Fly(Shankar, NewYork, NewDelhi, Yesterday)."
- **Expressiveness**: While limited compared to first-order logic (e.g., missing negation, disjunction), semantic networks are simple, efficient, and allow for procedural attachments to handle specific relations.

Description Logics

- **Overview**: Description logics are formal systems for defining and reasoning about categories, emphasizing clarity and tractability.
- **Purpose**: They formalize the meaning of semantic networks while focusing on taxonomic structures and efficient reasoning.
- **Subsumption:** Checking if one category is a subset of another by comparing definitions.
- **Classification:** Determining if an object belongs to a specific category.
- **Consistency:** Ensuring that category definitions are logically satisfiable.

Syntax and Example:

- Syntax uses logical constructs like And, Or, AtLeast, AtMost, etc.
- Example: A "Bachelor" can be defined as And(Unmarried, Adult, Male).

ACTING UNDER UNCERTAINTY

- Uncertainty is a situation which involves imperfect and/or unknown information.
- However, "uncertainty is an impossible expression to understand without a straight forward description.
- Uncertainty can also arise because of incompleteness, incorrectness in agents understanding the properties of environment.
- So to represent uncertain knowledge, where we are not sure about the predicates, we need uncertain reasoning or probabilistic reasoning.

Characteristics of Uncertain Domains

1. **Incomplete Information:** The system does not have access to all the data required to make a fully informed decision.

2. **Ambiguity:** Information might be unclear or open to multiple interpretations.

3. **Noise:** Data might be corrupted or imprecise due to measurement errors or external factors.

4. **Stochastic Processes:** The environment might involve random processes or events.

Causes of uncertainty:

- Following are some leading causes of uncertainty to occur in the real world.

1. Information occurred from unreliable sources.
2. Experimental Errors
3. Equipment fault
4. Temperature variation
5. Climate change.

Probabilistic reasoning

- Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge.
- In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.
- We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.
- In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today," "behavior of someone for some situations," "A match between two teams or two players."
- These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Bayesian Networks

- Joint probability distribution
- Bayesian Networks with examples
- Semantics of Bayesian Networks
  - Representing the full joint distribution
    - A method for constructing Bayesian Network
    - Compactness and node ordering
  - Conditional independence relation in Bayesian networks.

Joint probability distribution

- The **full joint probability distribution** specifies the probability of values to random variables.
- It is usually too large to create or use in its explicit form.
- Joint probability distribution of two variables X and Y are

Joint probabilities	X	X’
Y	0.20	0.12
Y’	0.65	0.03

- Joint probability distribution for n variables require 2^n entries with all possible combination.

Drawbacks of joint Probability Distribution

- Large number of variables and grows rapidly
- Time and space complexity are huge
- Statistical estimation with probability is difficult
- Human tends signal out few propositions
- The alternative to this is Bayesian Networks.

Bayes' Theorem

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Ex You've been planning a picnic for your family. You're trying to decide whether to postpone due to rain. The chance of rain on any day is 15%. The morning of the picnic, it's cloudy. The prob. of it being cloudy is 25% and on days where it rains, it's cloudy in the morning 80% of the time. Should you postpone the picnic?

$$P(\text{rain}) = 0.15$$

$$P(\text{cloudy}) = 0.25$$

$$P(\text{cloudy}|\text{rain}) = 0.80$$

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

$$P(\text{rain}|\text{cloudy}) = \frac{P(\text{cloudy}|\text{rain}) \cdot P(\text{rain})}{P(\text{cloudy})}$$

$$= \frac{0.8 \cdot 0.15}{0.25}$$

$$\boxed{P(\text{rain}|\text{cloudy}) = 0.48}$$

Problem: Suppose a spam filter uses Bayes' Theorem to classify emails as spam or not spam. The prior probability that an email is spam is $P(\text{Spam}) = 0.3$, and the probability that an email is not spam is $P(\text{Not Spam}) = 0.7$. The word "free" appears in 40% of spam emails ($P(\text{Free} \mid \text{Spam}) = 0.4$) and in 10% of non-spam emails ($P(\text{Free} \mid \text{Not Spam}) = 0.1$).

Given that the word "free" appears in an email, what is the probability that the email is spam?

Solution: Using Bayes' Theorem:

$$P(\text{Spam} \mid \text{Free}) = \frac{P(\text{Free} \mid \text{Spam}) \cdot P(\text{Spam})}{P(\text{Free})}$$

where:

$$P(\text{Free}) = P(\text{Free} \mid \text{Spam}) \cdot P(\text{Spam}) + P(\text{Free} \mid \text{Not Spam}) \cdot P(\text{Not Spam})$$

Substitute the values:

$$P(\text{Free}) = (0.4 \cdot 0.3) + (0.1 \cdot 0.7) = 0.12 + 0.07 = 0.19$$

Now calculate:

$$P(\text{Spam} \mid \text{Free}) = \frac{0.4 \cdot 0.3}{0.19} = \frac{0.12}{0.19} \approx 0.63$$

So, the probability that the email is spam given that it contains the word "free" is approximately **63%**.

Problem: A new test for a rare disease has a 99% accuracy rate. The disease affects 1 in 1,000 people ($P(\text{Disease}) = 0.001$). If the test is positive, what is the probability that the person actually has the disease? The test has a false positive rate of 1% ($P(\text{Positive} \mid \text{No Disease}) = 0.01$).

Solution: Using Bayes' Theorem:

$$P(\text{Disease} \mid \text{Positive}) = \frac{P(\text{Positive} \mid \text{Disease}) \cdot P(\text{Disease})}{P(\text{Positive})}$$

Where:

$$P(\text{Positive}) = P(\text{Positive} \mid \text{Disease}) \cdot P(\text{Disease}) + P(\text{Positive} \mid \text{No Disease}) \cdot P(\text{No Disease})$$

Substitute the values:

$$P(\text{Positive}) = (0.99 \cdot 0.001) + (0.01 \cdot 0.999) = 0.00099 + 0.00999 = 0.01098$$

Now calculate:

$$P(\text{Disease} \mid \text{Positive}) = \frac{0.99 \cdot 0.001}{0.01098} = \frac{0.00099}{0.01098} \approx 0.09$$

So, the probability that the person has the disease given a positive test is approximately **9%**.

Problem: A weather forecasting system predicts the probability of rain and thunderstorm occurrences as follows:

- Probability of rain ($P(\text{Rain})$) = 0.4
- Probability of a thunderstorm ($P(\text{Thunderstorm})$) = 0.3
- Probability of both rain and a thunderstorm occurring ($P(\text{Rain and Thunderstorm})$) = 0.2

What is the probability that it rains, but no thunderstorm occurs?

Solution: The probability that it rains but no thunderstorm occurs is given by:

$$P(\text{Rain but no Thunderstorm}) = P(\text{Rain}) - P(\text{Rain and Thunderstorm})$$

Substitute the values:

$$P(\text{Rain but no Thunderstorm}) = 0.4 - 0.2 = 0.2$$

Answer: The probability that it rains but no thunderstorm occurs is **0.2 (or 20%)**.

Problem: A supermarket tracks customer behavior and finds the following probabilities:

- Probability a customer buys bread ($P(\text{Bread})$) = 0.6
- Probability a customer buys milk ($P(\text{Milk})$) = 0.5
- Probability a customer buys both bread and milk ($P(\text{Bread and Milk})$) = 0.3

What is the probability that a customer buys at least one of the two items (bread or milk)?

Solution: The probability of buying at least one of the two items is given by:

$$P(\text{Bread or Milk}) = P(\text{Bread}) + P(\text{Milk}) - P(\text{Bread and Milk})$$

Substitute the values:

$$P(\text{Bread or Milk}) = 0.6 + 0.5 - 0.3 = 0.8$$

Answer: The probability that a customer buys at least one of the two items is **0.8 (or 80%)**.

Problem: In a population, the probability that a person has a specific disease is $P(\text{Disease}) = 0.01$. The probability that a person tests positive for the disease is $P(\text{Positive}) = 0.05$. The probability that a person both has the disease and tests positive is $P(\text{Disease and Positive}) = 0.009$.

What is the probability that a person who tests positive does **not** have the disease?

Solution: The probability that a person tests positive and does not have the disease is given by:

$$P(\text{Not Disease and Positive}) = P(\text{Positive}) - P(\text{Disease and Positive})$$

Substitute the values:

$$P(\text{Not Disease and Positive}) = 0.05 - 0.009 = 0.041$$

Next, to find the probability that a person who tests positive does not have the disease, we use conditional probability:

$$P(\text{Not Disease} \mid \text{Positive}) = \frac{P(\text{Not Disease and Positive})}{P(\text{Positive})}$$

Substitute the values:

$$P(\text{Not Disease} \mid \text{Positive}) = \frac{0.041}{0.05} = 0.82$$

Answer: The probability that a person who tests positive does not have the disease is **0.82 (or 82%)**.

UNIT V: ROBOTICS

- Robots are physical agents that perform tasks by manipulating the physical world.
- To do so, they are equipped with effectors such as legs, wheels, joints, and grippers.
- Effectors have a single purpose: to assert physical forces on the environment.
- Robots are also equipped with sensors, which allow them to perceive their environment.
- Present day robotics employs a diverse set of sensors, including cameras and lasers to measure the environment, and gyroscopes and

Most of today's robots fall into one of **three primary categories**.

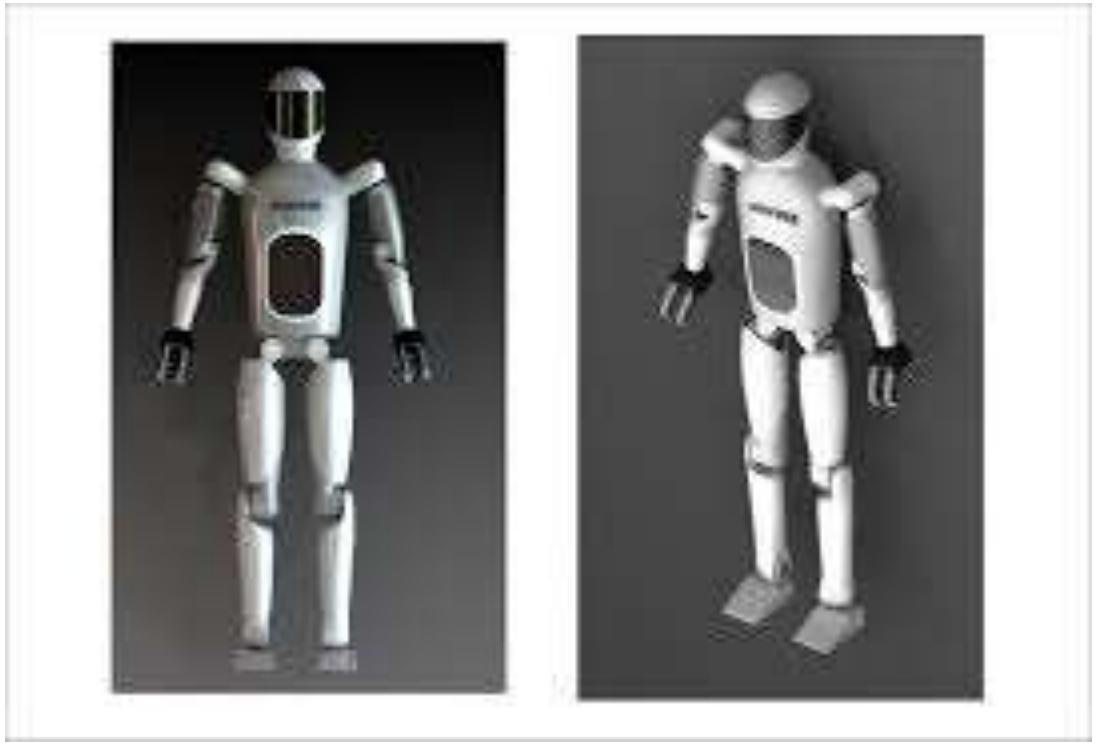
- **Manipulators, or robot arms** , are physically anchored to their workplace, for example in a factory assembly line or on the International Space Station.
- Manipulators are by far the most common type of industrial robots Some mobile manipulators are used in hospitals to assist surgeons.
- Few car manufacturers could survive without robotic manipulators, and some manipulators have even been used to generate original artwork

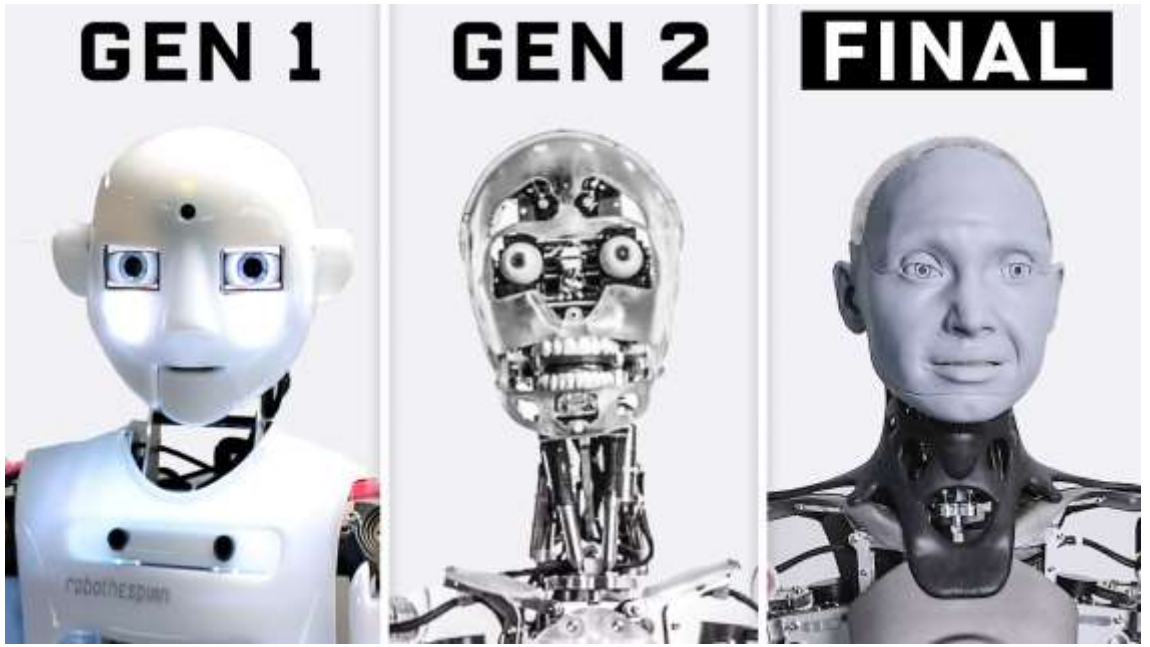
Industrial robotic manipulator

- The second category is the **mobile robot**.
- Mobile robots move about their environment using wheels, legs, or similar mechanisms.
- They have been put to use delivering food in hospitals, moving containers at loading docks, and similar tasks.
- Unmanned ground vehicles, or UGVs, drive autonomously on streets, highways, and off-road.

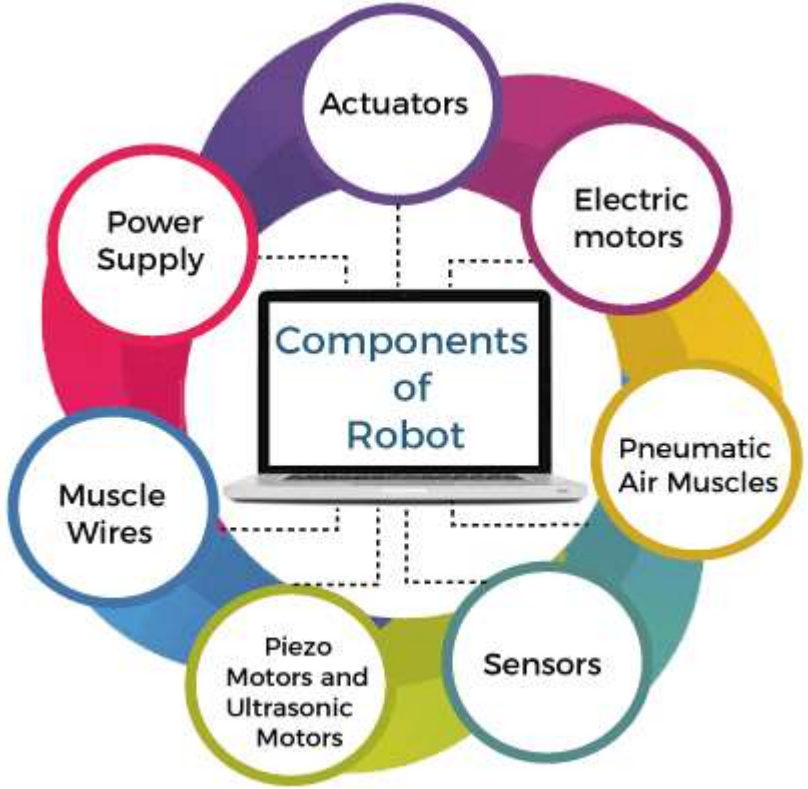
- Other types of mobile robots include **unmanned air vehicles (UAVs),** commonly used for surveillance, crop-spraying, and military operations.
- **Autonomous underwater vehicles** (AUVs) are used in deep sea exploration.
- Mobile robots deliver packages in the workplace and vacuum the floors at home.

- The third type of robot combines mobility with manipulation, and is often called a mobile manipulator.
- **Humanoid robots** mimic the human torso





COMPONENTS OF ROBOT



- Actuators are the devices that are responsible for moving and controlling a system or machine.
- Power Supply: It is an electrical device that supplies electrical power to an electrical load
- Electric Motors: These are the devices that convert electrical energy into mechanical energy and are required for the rotational motion of the machines.
- Ultrasonic Motors are the electrical devices that receive an electric signal and apply a directional force to an opposing ceramic plate. It helps a robot to move in the desired direction. These are the best suited electrical motors for industrial robots.
- Sensor: They provide the ability like see, hear, touch and movement like humans. Sensors are the devices or machines which help to detect the events or changes in the environment and send data to the computer processor.

ROBOT HARDWARE :

- Sensors: Sensors are the perceptual interface between robot and environment.
- Two types of sensors :
- **Passive sensors**, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment.
- **Active sensors**, such as sonar, send energy into the environment. They rely on the fact that this energy is reflected back to the sensor.
- Active sensors tend to provide more information than passive sensors, but at the expense of increasedpower consumption and with a danger of

- Whether active or passive, sensors can be divided into three types, depending on :
  - whether they sense the environment,
  - the robot’s location,
  - the robot’s internal configuration

Figure 25.3 (a) Time of flight camera; image courtesy of Mesa Imaging GmbH. (b) 3D range image obtained with this camera. The range image makes it possible to detect obstacles and objects in a robot's vicinity.

Effectors

- Effectors are the means by which robots move and change the shape of their bodies.
- To understand the design of effectors, it will help to talk about motion and shape in the abstract, using the concept of a degree of freedom (DOF)
- We count one degree of freedom for each independent direction in which a robot, or one of its effectors, can move
- The dynamic state of a robot includes these six plus an additional six dimensions for the rate of change of each kinematic dimension, that is, their velocities.
- For non rigid bodies, there are additional degrees of freedom within the robot itself.
- For example, the elbow of a human arm possesses two degree of freedom. It can flex the upper arm towards or away, and can rotate right or left.

- Robot joints also have one, two, or three degrees of freedom each. Six degrees of freedom are required to place an object, such as a hand, at a particular point in a particular orientation.
- The arm has exactly six degrees of freedom, created by five revolute joints that generate rotational motion and one prismatic joint that generates sliding motion

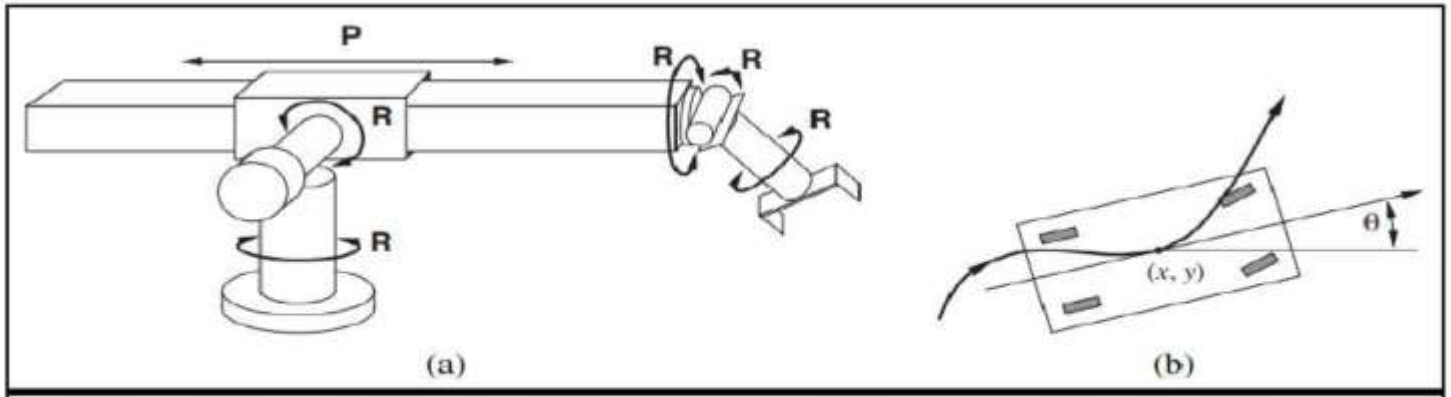


Figure 25.4 (a) The Stanford Manipulator, an early robot arm with five revolute joints (R) and one prismatic joint (P), for a total of six degrees of freedom. (b) Motion of a nonholonomic four-wheeled vehicle with front-wheel steering.