SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINERING

# UNIT- 1- ARTIFICIAL INTELLIGENCE- SECA3011

**COURSE OBJECTIVES**

 To understand the various characteristics of Intelligent agents.

 To learn the different search strategies in AI.

 To learn to represent knowledge in solving AI problems.

 To know about the various applications of AI.

**UNIT 1 INTRODUCTION TO ARTIFICIAL INTELLIGENCE 9 Hrs.**

Introduction–Definition – Future of Artificial Intelligence – Characteristics of Intelligent Agents–Typical Intelligent Agents – Problem Solving Approach to Typical AI problems.

**UNIT 2 PROBLEM SOLVING METHODS 9 Hrs.**

Problem solving Methods – Search Strategies- Uninformed – Informed – Heuristics – Local Search Algorithms and Optimization Problems – Searching with Partial Observations – Constraint Satisfaction Problems – Constraint Propagation – Backtracking Search – Game Playing – Optimal Decisions in Games – Alpha – Beta Pruning – Stochastic Games.

**UNIT 3 KNOWLEDGE REPRESENTATION 9 Hrs.**

First Order Predicate Logic – Prolog Programming – Unification – Forward Chaining-Backward Chaining – Resolution – Knowledge Representation – Ontological Engineering-Categories and Objects – Events – Mental Events and Mental Objects – Reasoning Systems for Categories – Reasoning with Default Information.

**UNIT 4 SOFTWARE AGENTS 9 Hrs.**

Architecture for Intelligent Agents – Agent communication – Negotiation and Bargaining – Argumentation among Agents – Trust and Reputation in Multi-agent systems.

**UNIT 5 APPLICATIONS 9 Hrs.**

AI applications – Language Models – Information Retrieval- Information Extraction – Natural Language Processing – Machine Translation – Speech Recognition.

Max. 45 Hrs.

**COURSE OUTCOMES**

On completion of the course, student will be able to

CO1 - Understand the principles of Artificial Intelligence.

CO2 - Use appropriate search algorithms for any AI problem.

CO3 - Represent a problem using first order and predicate logic.

CO4 - Provide the apt agent strategy to solve a given problem.

CO5 - Design software agents to solve a problem.

CO6 - Design applications for NLP that use Artificial Intelligence.

# UNIT 1

# INTRODUCTIONTO ARTIFICIAL INTELLIGENCE

Introduction–Definition –Future of Artificial Intelligence –Characteristics of Intelligent Agents– Typical Intelligent Agents –Problem Solving Approach to Typical AI problems.

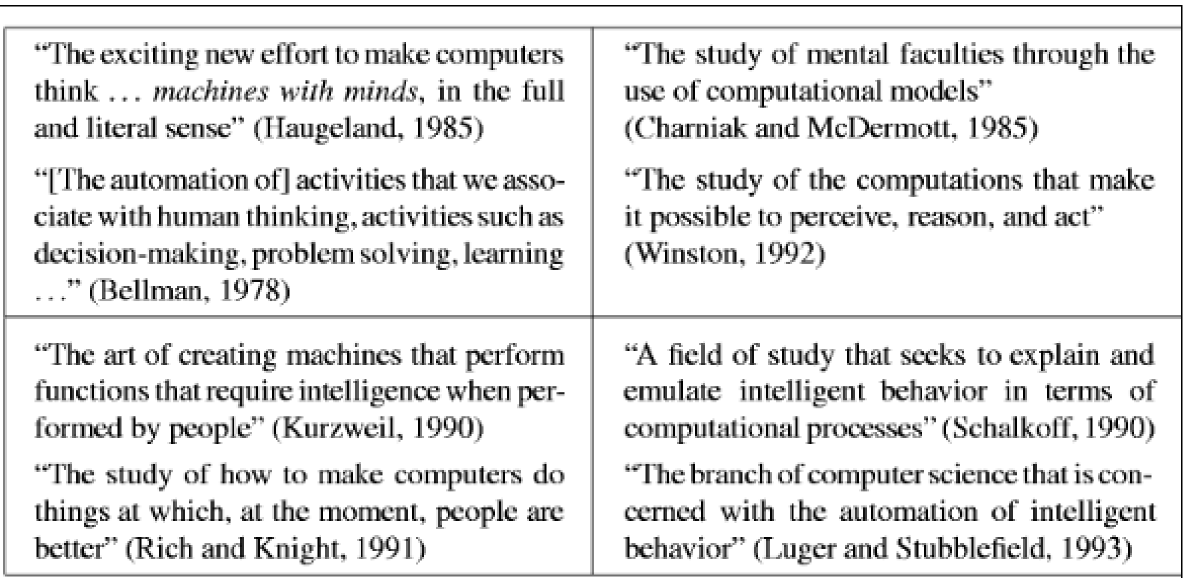
## 1.1 INTRODUCTION

| INTELLIGENCE | ARTIFICIAL INTELLIGENCE |
|---|---|
| It is a natural process. | It is programmed by humans. |
| It is actually hereditary. | It is not hereditary. |
| Knowledge is required for intelligence. | KB and electricity are required to generate output. |
| No human is an expert. We may get better solutions from other humans. | Expert systems are made which aggregate many person's experience and ideas. |

## 1.2 DEFINITION

The study of how to make computers do things at which at the moment, people are better. **"Artificial Intelligence is the ability of a computer to act like a human being".**

- Systems that think like humans
- Systems that act like humans
- Systems that think rationally. Systems that act rationally**.**



| | |
|---|---|
| "The exciting new effort to make computers think … *machines with minds*, in the full and literal sense" (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning …" (Bellman, 1978) | "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992) |
| "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991) | "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993) |

**Figure 1.1 Some definitions of artificial intelligence, organized into four categories**

**(a)** **Intelligence -** Ability to apply knowledge in order to perform better in an environment.

**(b)** **Artificial Intelligence -** Study and construction of agent programs that perform well in a given environment, for a given agent architecture**.**

**(c)** **Agent -** An entity that takes action in response to precepts from an environment**.**

**(d)** **Rationality** - property of a system which does the "right thing" given what it knows.

**(e)** **Logical Reasoning -** A process of deriving new sentences from old, such that the new sentences are necessarily true if the old ones are true.

Four Approaches of Artificial Intelligence:

- ➢ Acting humanly: The Turing test approach.
- ➢ Thinking humanly: The cognitive modelling approach.
- ➢ Thinking rationally: The laws of thought approach.
- ➢ Acting rationally: The rational agent approach.

## 1.3 ACTING HUMANLY: THE TURING TEST APPROACH

The Turing Test, proposed by Alan Turing (1950), was designed to provide a satisfactory operational definition of intelligence. A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer.

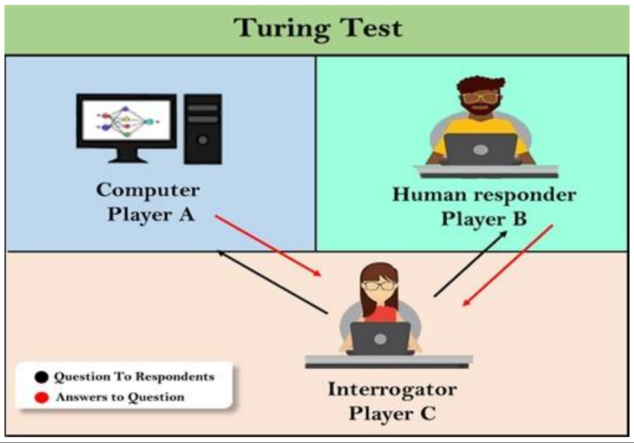


**Figure 1.2 Turing Test**

- **natural language processing** to enable it to communicate successfully in English;
- **knowledge representation** to store what it knows or hears;
- **automated reasoning** to use the stored information to answer questions and to draw new conclusions
- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

**Total Turing Test** includes a video signal so that the interrogator can test the subject’s perceptual abilities, as well as the opportunity for the interrogator to pass physical objects “through the hatch.” To pass the total Turing Test, the computer will need

- **computer vision** to perceive objects, and **robotics** to manipulate objects and move about**.**

Thinking humanly: The cognitive modelling approach

Analyse how a given program thinks like a human, we must have some way of determining how humans think. The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind.

Although cognitive science is a fascinating field in itself, we are not going to be discussing it all that much in this book. We will occasionally comment on similarities or differences between AI techniques and human cognition. Real cognitive science, however, is necessarily based on experimental investigation of actual humans or animals, and we assume that the reader only has access to a computer for experimentation. We will simply note that AI and cognitive science continue to fertilize each other, especially in the areas of vision, natural language, and learning.

Thinking rationally: The “laws of thought” approach

The Greek philosopher Aristotle was one of the first to attempt to codify ``right thinking," that is, irrefutable reasoning processes. His famous syllogisms provided patterns for argument structures that always gave correct conclusions given correct premises.

For example, ``Socrates is a man; all men are mortal; therefore Socrates is mortal."

These laws of thought were supposed to govern the operation of the mind, and initiated the field of ***logic.***

Acting rationally: The rational agent approach

Acting rationally means acting so as to achieve one's goals, given one's beliefs. An agent is just something that perceives and acts.

The right thing: that which is expected to maximize goal achievement, given the available information

Does not necessary involve thinking.

For Example - blinking reflex- but should be in the service of rational action.

## 1.4 FUTURE OF ARTIFICIAL INTELLIGENCE

- **Transportation:** Although it could take a decade or more to perfect them, autonomous cars will one day ferry us from place to place.
- **Manufacturing:** AI powered robots work alongside humans to perform a limited range of tasks like assembly and stacking, and predictive analysis sensors keep equipment running smoothly.
- **Healthcare:** In the comparatively AI-nascent field of healthcare, diseases are more quickly and accurately diagnosed, drug discovery is sped up and streamlined, virtual nursing assistants monitor patients and big data analysis helps to create a more personalized patient experience.
- **Education:** Textbooks are digitized with the help of AI, early-stage virtual tutors assist human instructors and facial analysis gauges the emotions of students to help determine who's struggling or bored and better tailor the experience to their individual needs.
- **Media:** Journalism is harnessing AI, too, and will continue to benefit from it. Bloomberg uses Cyborg technology to help make quick sense of complex financial reports. The Associated Press employs the natural language abilities of Automated Insights to produce 3,700 earning reports stories per year — nearly four times more than in the recent past
- **Customer Service:** Last but hardly least, Google is working on an AI assistant that can place human-like calls to make appointments at, say, your neighborhood hair salon. In addition to words, the system understands context and nuance.

## 1.5 CHARACTERISTICS OF INTELLIGENT AGENTS

**Situatedness**

The agent receives some form of sensory input from its environment, and it performs some action that changes its environment in some way.

Examples of environments: the physical world and the Internet.

- Autonomy

The agent can act without direct intervention by humans or other agents and that it has control over its own actions and internal state.

- Adaptivity

The agent is capable of

(1) reacting flexibly to changes in its environment;

(2) taking goal-directed initiative (i.e., is pro-active), when appropriate; and

(3) Learning from its own experience, its environment, and interactions with others.

- Sociability

  The agent is capable of interacting in a peer-to-peer manner with other agents or humans

## 1.6 AGENTS AND ITS TYPES

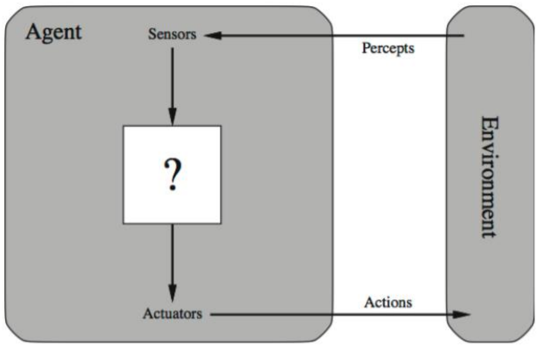


**Figure 1.3 Agent types**

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

- Human Sensors:
- Eyes, ears, and other organs for sensors.
- Human Actuators:
- Hands, legs, mouth, and other body parts.
- Robotic Sensors:
- Mic, cameras and infrared range finders for sensors
- Robotic Actuators:
- Motors, Display, speakers etc An agent can be:

**Human-Agent:** A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.

**Robotic Agent:** A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.

**Software Agent:** Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

Hence the world around us is full of agents such as thermostat, cell phone, camera, and even we are also agents. Before moving forward, we should first know about sensors, effectors, and actuators.

**Sensor:** Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.

**Actuators:** Actuators are the component of machines that converts energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.

**Effectors:** Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.

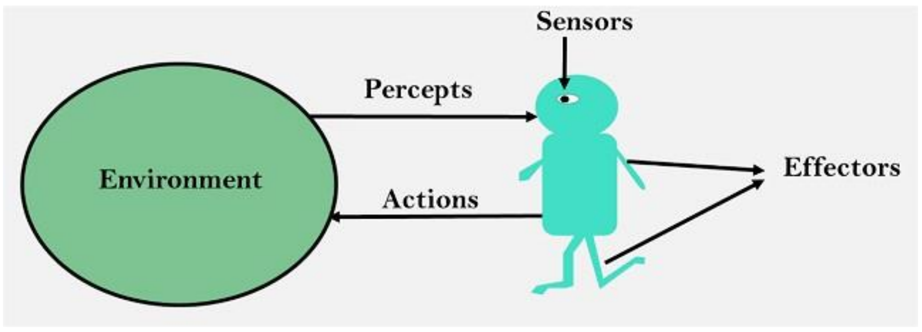


**Figure 1.4 Effectors**

## 1.7 PROPERTIES OF ENVIRONMENT

An **environment** is everything in the world which surrounds the agent, but it is not a part of an agent itself. An environment can be described as a situation in which an agent is present.

The environment is where agent lives, operate and provide the agent with something to sense and act upon it.

Fully observable vs Partially Observable:

If an agent sensor can sense or access the complete state of an environment at each point of time then it is **a fully observable** environment, else it is **partially observable**.

A fully observable environment is easy as there is no need to maintain the internal state to keep track history of the world.

An agent with no sensors in all environments then such an environment is called as unobservable.

**Example:** chess – the board is fully observable, as are opponent’s moves. Driving – what is around the next bend is not observable and hence partially observable.

**1. Deterministic vs Stochastic**

- If an agent's current state and selected action can completely determine the next state of the environment, then such environment is called a deterministic environment.

- A stochastic environment is random in nature and cannot be determined completely by an agent.
- In a deterministic, fully observable environment, agent does not need to worry about uncertainty.

2. **Episodic vs Sequential**

- In an episodic environment, there is a series of one-shot actions, and only the current percept is required for the action.
- However, in Sequential environment, an agent requires memory of past actions to determine the next best actions.

3. **Single-agent vs Multi-agent**

- If only one agent is involved in an environment, and operating by itself then such an environment is called single agent environment.
- However, if multiple agents are operating in an environment, then such an environment is called a multi-agent environment.
- The agent design problems in the multi-agent environment are different from single agent environment.

4. **Static vs Dynamic**

- If the environment can change itself while an agent is deliberating then such environment is called a dynamic environment else it is called a static environment.
- Static environments are easy to deal because an agent does not need to continue looking at the world while deciding for an action.
- However for dynamic environment, agents need to keep looking at the world at each action.
- Taxi driving is an example of a dynamic environment whereas Crossword puzzles are an example of a static environment.

5. **Discrete vs Continuous**

- If in an environment there are a finite number of precepts and actions that can be performed within it, then such an environment is called a discrete environment else it is called continuous environment.
- A chess game comes under discrete environment as there is a finite number of moves that can be performed.
- A self-driving car is an example of a continuous environment.

6. **Known vs Unknown**

- Known and unknown are not actually a feature of an environment, but it is an agent's state of knowledge to perform an action.
- In a known environment, the results for all actions are known to the agent. While in unknown environment, agent needs to learn how it works in order to perform an action.
- It is quite possible that a known environment to be partially observable and an Unknown environment to be fully observable.

7. **Accessible vs. Inaccessible**

- If an agent can obtain complete and accurate information about the state's environment, then such an environment is called an Accessible environment else it is called inaccessible.
- An empty room whose state can be defined by its temperature is an example of an accessible environment.
- Information about an event on earth is an example of Inaccessible environment.

**Task environments**, which are essentially the "problems" to which rational agents are the "solutions."

**PEAS:** Performance Measure, Environment, Actuators, Sensors

***Performance***

The output which we get from the agent. All the necessary results that an agent gives after processing comes under its performance.

***Environment***

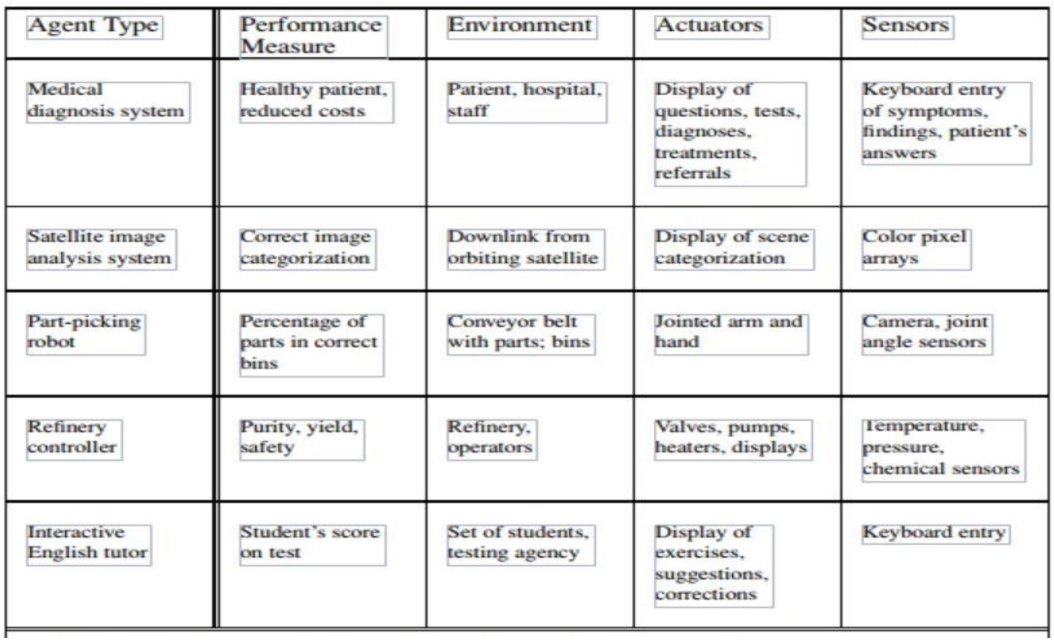
All the surrounding things and conditions of an agent fall in this section. It basically consists of all the things under which the agents work.

***Actuators***

The devices, hardware or software through which the agent performs any actions or processes any information to produce a result are the actuators of the agent.

***Sensors***

The devices through which the agent observes and perceives its environment are the sensors of the agent.



| Agent Type | Performance Measure | Environment | Actuators | Sensors |
|---|---|---|---|---|
| Medical diagnosis system | Healthy patient, reduced costs | Patient, hospital, staff | Display of questions, tests, diagnoses, treatments, referrals | Keyboard entry of symptoms, findings, patient's answers |
| Satellite image analysis system | Correct image categorization | Downlink from orbiting satellite | Display of scene categorization | Color pixel arrays |
| Part-picking robot | Percentage of parts in correct bins | Conveyor belt with parts; bins | Jointed arm and hand | Camera, joint angle sensors |
| Refinery controller | Purity, yield, safety | Refinery, operators | Valves, pumps, heaters, displays | Temperature, pressure, chemical sensors |
| Interactive English tutor | Student's score on test | Set of students, testing agency | Display of exercises, suggestions, corrections | Keyboard entry |

**Figure 1.5 Examples of agent types and their PEAS descriptions**

**Rational Agent -** A system is rational if it does the "right thing". Given what it knows.

**Characteristic of Rational Agent**

- The agent's prior knowledge of the environment.
- The performance measure that defines the criterion of success.
- The actions that the agent can perform.
- The agent's percept sequence to date.

For every possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

- An **omniscient agent** knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.

- **Ideal Rational Agent** precepts and does things. It has a greater performance measure.

  Eg. Crossing road. Here first perception occurs on both sides and then only action. No perception occurs in **Degenerate Agent**.

  Eg. Clock. It does not view the surroundings. No matter what happens outside. The clock works based on inbuilt program.

- **Ideal Agent** describes by ideal mappings. "Specifying which action an agent ought to take in response to any given percept sequence provides a design for ideal agent".

- **Eg.** SQRT function calculation in calculator.
- Doing actions in order to modify future precepts-sometimes called **information gathering**- is an important part of rationality.
- A rational agent should be **autonomous**-it should learn from its own prior knowledge (experience).

**The Structure of Intelligent Agents**

***Agent = Architecture + Agent Program***

Architecture = the machinery that an agent executes on. (Hardware)

Agent Program = an implementation of an agent function. (Algorithm, Logic – Software)

## 1.8 TYPES OF AGENTS

Agents can be grouped into four classes based on their degree of perceived intelligence and capability :

- Simple Reflex Agents
- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents
- Learning Agent

**The Simple reflex agents**

- The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history (**past State).**
- These agents only succeed in the fully observable environment.
- The Simple reflex agent does not consider any part of percepts history during their decision and action process.
- The Simple reflex agent works on Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, it works only if there is dirt in the room.
- Problems for the simple reflex agent design approach:
    - They have very limited intelligence
    - They do not have knowledge of non-perceptual parts of the current state

- Mostly too big to generate and to store.
- Not adaptive to changes in the environment.

**Condition-Action Rule** − It is a rule that maps a state (condition) to an action.

**Ex:** if car-in-front-is-braking then initiate- braking.

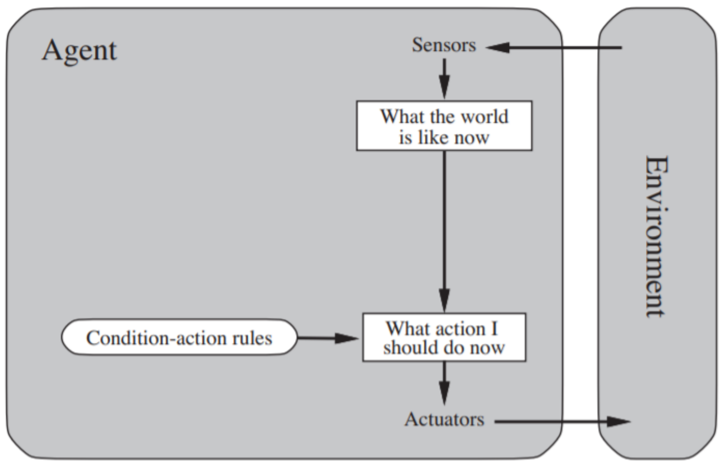


**Figure 1.6 A simple reflex agent**

**Model Based Reflex Agents**

- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
  - **Model:** It is knowledge about "how things happen in the world," so it is called a Model-based agent.
  - **Internal State:** It is a representation of the current state based on percept history.
- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.
- Updating the agent state requires information about:
  - How the world evolves
  - How the agent's action affects the world.

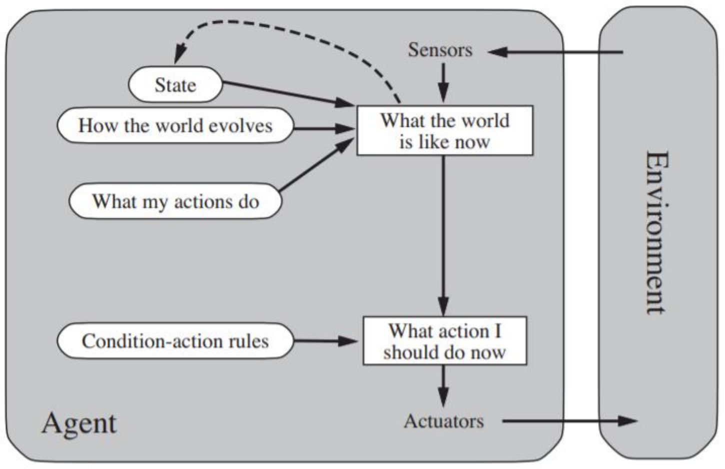


**Figure 1.7 A model-based reflex agent**

## Goal Based Agents

- The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.
- They choose an action, so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.

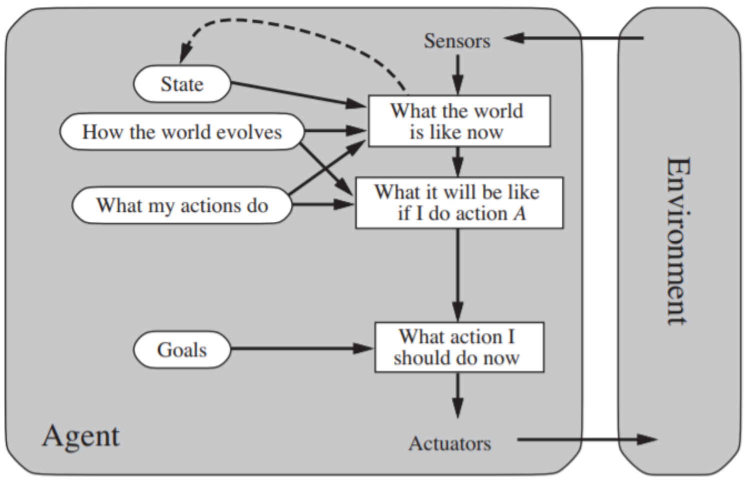

Sensors
State
How the world evolves
What the world is like now
What it will be like if I do action A
What my actions do
Environment
Goals
What action I should do now
Agent
Actuators

**Figure 1.8 A goal-based agent**

**Utility Based Agents**

- These agents are similar to the goal-based agent but provide an extra component of utility measurement **("Level of Happiness")** which makes them different by providing a measure of success at a given state.
- Utility-based agent act based not only goals but also the best way to achieve the goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The utility function maps each state to a real number to check how efficiently each action achieves the goals.

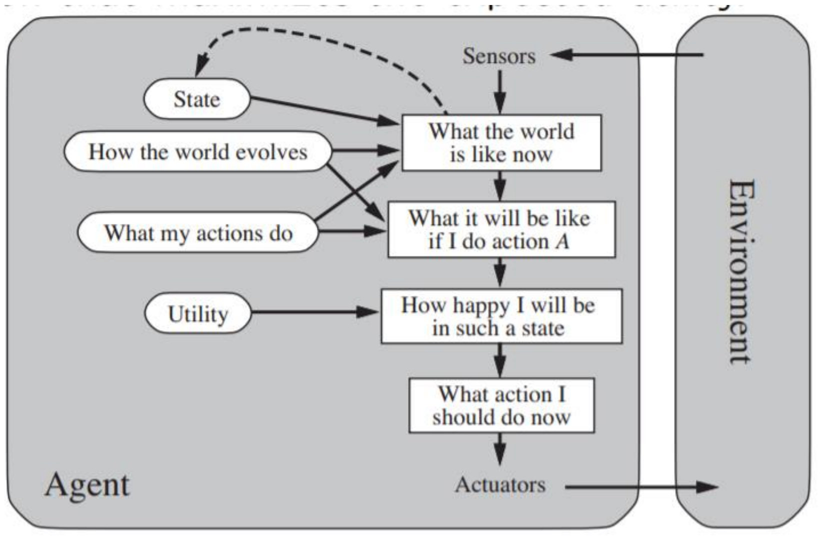

Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action *A*
How happy I will be in such a state
What action I should do now
Actuators
Agent
Environment

**Figure 1.9 A utility-based agent**

**Learning Agents**

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- A learning agent has mainly four conceptual components, which are:

    a. **Learning element:** It is responsible for making improvements by learning from environment

    b. **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.

    c. **Performance element:** It is responsible for selecting external action

d. **Problem generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.

- Hence, learning agents are able to learn, analyze performance, and look for new ways to improve the performance.

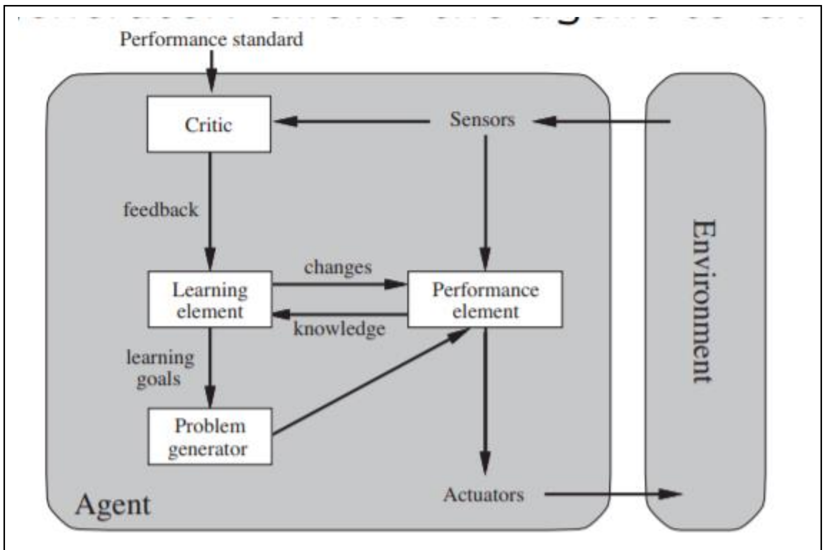


**Figure 1.10 Learning Agents**

## 1.9 PROBLEM SOLVING APPROACH TO TYPICAL AI PROBLEMS

### Problem-solving agents

In Artificial Intelligence, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem- solving agents are the goal-based agents and use atomic representation. In this topic, wewill learn various problem-solving search algorithms.

- Some of the most popularly used problem solving with the help of artificial intelligence are:

1. Chess.
2. Travelling Salesman Problem.
3. Tower of Hanoi Problem.
4. Water-Jug Problem.
5. N-Queen Problem.

### Problem Searching

- In general, searching refers to as finding information one needs.

- Searching is the most commonly used technique of problem solving in artificial intelligence.
- The searching algorithm helps us to search for solution of particular problem.

**Problem:** Problems are the issues which comes across any system. A solution is needed to solve that particular problem.

**Steps : Solve Problem Using Artificial Intelligence**

- The process of solving a problem consists of five steps. These are:

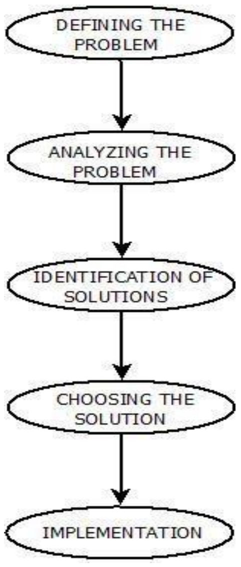


**Figure 1.11 Problem Solving in Artificial Intelligence**

**Defining The Problem**: The definition of the problem must be included precisely. It should contain the possible initial as well as final situations which should result in acceptable solution.

1. **Analyzing The Problem**: Analyzing the problem and its requirement must be done as few features can have immense impact on the resulting solution.
2. **Identification Of Solutions**: This phase generates reasonable amount of solutions to the given problem in a particular range.
3. **Choosing a Solution**: From all the identified solutions, the best solution is chosen basis on the results produced by respective solutions.
4. **Implementation**: After choosing the best solution, its implementation is done.

**Measuring problem-solving performance**

We can evaluate an algorithm's performance in four ways:

**Completeness:** Is the algorithm guaranteed to find a solution when there is one?

**Optimality:** Does the strategy find the optimal solution?

**Time complexity**: How long does it take to find a solution?

**Space complexity:** How much memory is needed to perform the search?

**Search Algorithm Terminologies**

- Search: Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
    1. Search Space: Search space represents a set of possible solutions, which a system may have.
    2. Start State: It is a state from where agent begins the search.
    3. Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.
- Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- Actions: It gives the description of all the available actions to the agent.
- Transition model: A description of what each action do, can be represented as a transition model.
- Path Cost: It is a function which assigns a numeric cost to each path.
- Solution: It is an action sequence which leads from the start node to the goal node. Optimal Solution: If a solution has the lowest cost among all solutions.

**Example Problems**

A **Toy Problem** is intended to illustrate or exercise various problem-solving methods. A**real- world problem** is one whose solutions people actually care about.

**Toy Problems**

Vacuum World

**States:** The state is determined by both the agent location and the dirt locations. The agent is in one of the 2 locations, each of which might or might not contain dirt. Thus there are 2*2^2=8 possible world states.

**Initial state**: Any state can be designated as the initial state.

**Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.

**Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost squ are, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect. The complete state space is shown in Figure.

**Goal test**: This checks whether all the squares are clean.

**Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

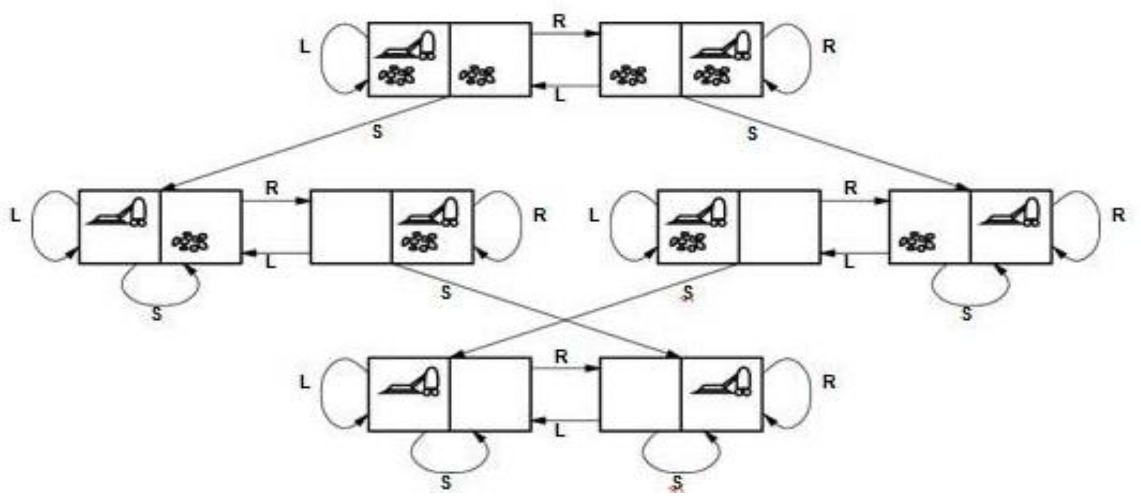


**Figure 1.12 Vacuum World State Space Graph**

**1) 8- Puzzle Problem**

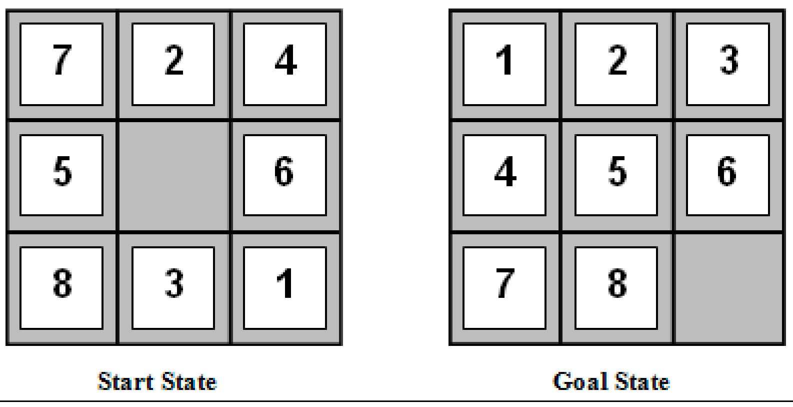


**Figure 1.13 8- Puzzle Problem**

**States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

**Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.

The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.

**Transition model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.

**Goal test**: This checks whether the state matches the goal configuration shown in Figure. **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

**Queens Problem**

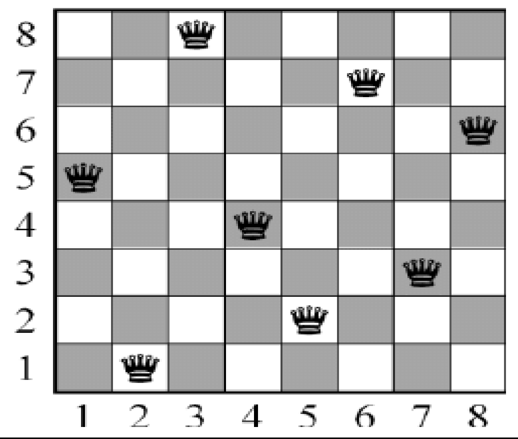


**Figure 1.14 Queens Problem**

- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

Consider the given problem. Describe the operator involved in it. Consider the water jug problem: You are given two jugs, a 4-gallon one and 3-gallon one. Neither has any measuring marker on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallon of water from the 4-gallon jug ?

Explicit Assumptions: A jug can be filled from the pump, water can be poured out of a jug on to the ground, water can be poured from one jug to another and that there are no other measuring devices available.

Here the initial state is (0, 0). The goal state is (2, n) for any value of n.

State Space Representation: we will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note that $0 \leq x \leq 4$, and $0 \leq y \leq 3$.

To solve this we have to make some assumptions not mentioned in the problem. They are:

- We can fill a jug from the pump.
- We can pour water out of a jug to the ground.
- We can pour water from one jug to another.
- There is no measuring device available.

Operators - we must define a set of operators that will take us from one state to another.

**Table 1.1**

| **Sr.** | **Current State** | **Next State** | **Descriptions** |
|---|---|---|---|
| 1 | (x,y) if x < 4 | (4,y) | Fill the 4 gallon jug |
| 2 | (x,y) if x < 3 | (x,3) | Fill the 3 gallon jug |
| 3 | (x,y) if x > 0 | (x – d, y) | Pour some water out of the 4 gallon jug |
| 4 | (x,y) if y > 0 | (x, y – d) | Pour some water out of the 3 gallon jug |
| 5 | (x,y) if y > 0 | (0, y) | Empty the 4 gallon jug |
| 6 | (x,y) if y > 0 | (x 0) | Empty the 3 gallon jug on the ground |
| 7 | (x,y) if x + y > = 4 and y > 0 | (4, y – (4 – x)) | Pour water from the 3 gallon jug into the 4 gallon jug until the 4 gallon jug is full |
| 8 | (x,y) if x + y > = 3 and x > 0 | (x – (3 – x), 3) | Pour water from the 4 gallon jug into the 3 gallon jug until the 3 gallon jug is full |
| 9 | (x,y) if x + y < = 4 and y > 0 | (x + y, 0) | Pour all the water from the 3 gallon jug into the 4 gallon jug |
| 10 | (x,y) if x + y < = 3 and x > 0 | (0, x + y) | Pour all the water from the 4 gallon jug into the 3 gallon jug |
| 11 | (0, 2) | (2, 0) | Pour the 2 gallons from 3 gallon jug into the 4 gallon jug |
| 12 | (2, y) | (0, y) | Empty the 2 gallons in the 4 gallon jug on the ground |

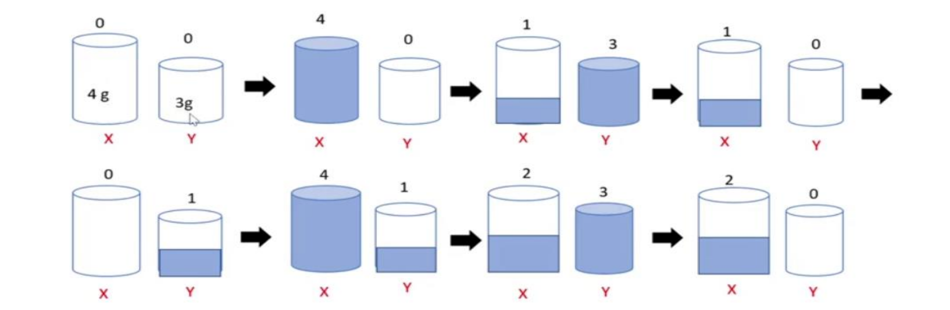


**Figure 1.15 Solution**

**Table 1.2**

**Solution**

| S.No. | Gallons in 4-gel jug(x) | Gallons in 3-gel jug (y) | Rule Applied |
|---|---|---|---|
| 1. | 0 | 0 | Initial state |
| 2.. | 4 | 0 | 1. Fill 4 |
| 3 | 1 | 3 | 6. Poor 4 into 3 to fill |
| 4. | 1 | 0 | 4. Empty 3 |
| 5. | 0 | 1 | 8. Poor all of 4 into 3 |
| 6. | 4 | 1 | 1. Fill 4 |
| 7. | 2 | 3 | 6. Poor 4 into 3 |

➢ 4-gallon one and a 3-gallon Jug

➢ No measuring mark on the jug.

➢ There is a pump to fill the jugs with water.

➢ How can you get exactly 2 gallon of water into the 4-gallon jug?

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINERING

# UNIT- II- ARTIFICIAL INTELLIGENCE- SECA3011

# UNIT 2

Problem solving Methods – Search Strategies- Uninformed – Informed – Heuristics – Local Search Algorithms and Optimization Problems - Searching with Partial Observations – Constraint Satisfaction Problems – Constraint Propagation - Backtracking Search – Game Playing – Optimal Decisions in Games – Alpha – Beta Pruning – Stochastic Games

## 2.1 PROBLEM SOLVING BY SEARCH

An important aspect of intelligence is ***goal-based*** problem solving.

The **solution** of many **problems** can be described by finding a **sequence of actions** that lead to a desirable **goal.** Each action changes the ***state*** and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

A well-defined problem can be described by:

- Initial state
  - **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
  - **State space** - all states reachable from initial by any sequence of actions
  - **Path** - sequence through state space
  - **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
  - **Goal test** - test to determine if at goal state

**What is Search?**

**Search** is the systematic examination of **states** to find path from the **start/root state** to the **goal state.**

The set of possible states, together with *operators* defining their connectivity constitute the *search space*.

The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

**Problem-solving agents**

A Problem solving agent is a **goal-based** agent. It decide what to do by finding sequence of actions that lead to desirable states. The agent can adopt a goal and aim at satisfying it.

To illustrate the agent's behavior, let us take an example where our agent is in the city of Arad, which is in Romania. The agent has to adopt a **goal** of getting to Bucharest.

**Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.

The agent's task is to find out which sequence of actions will get to a goal state.

**Problem formulation** is the process of deciding what actions and states to consider given a goal.

Example: Route finding problem

Referring to figure

On holiday in Romania : currently in Arad. Flight leaves tomorrow from Bucharest
**Formulate goal**: be in Bucharest

**Formulate problem**: **states**: various cities

**actions**: drive between cities

**Find solution**:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem formulation

A **problem** is defined by four items:

**initial state** e.g., "at Arad"

**successor function** S(x) = set of action-state pairs e.g., S(Arad) = {[Arad ->Zerind;Zerind],….} **goal test**, can be

explicit, e.g., x = at Bucharest" implicit, e.g., NoDirt(x)

**path cost** (additive)

e.g., sum of distances, number of actions executed, etc. c(x; a; y) is the step cost, assumed to be >= 0

A **solution** is a sequence of actions leading from the initial state to a goal state.

Goal formulation and problem formulation

## 2.2 EXAMPLE PROBLEMS

The problem solving approach has been applied to a vast array of task environments. Some best known problems are summarized below. They are distinguished as toy or real-world problems

A **toy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.

A **real world problem** is one whose solutions people actually care about.

## 2.3 TOY PROBLEMS

Vacuum World Example

- **States**: The agent is in one of two locations, each of which might or might not contain dirt. Thus there are 2 x $2^2$ = 8 possible world states.
- **Initial state**: Any state can be designated as initial state.
- **Successor function**: This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure
- **Goal Test**: This tests whether all the squares are clean.
- **Path test**: Each step costs one, so that the path cost is the number of steps in the path.

**Vacuum World State Space**

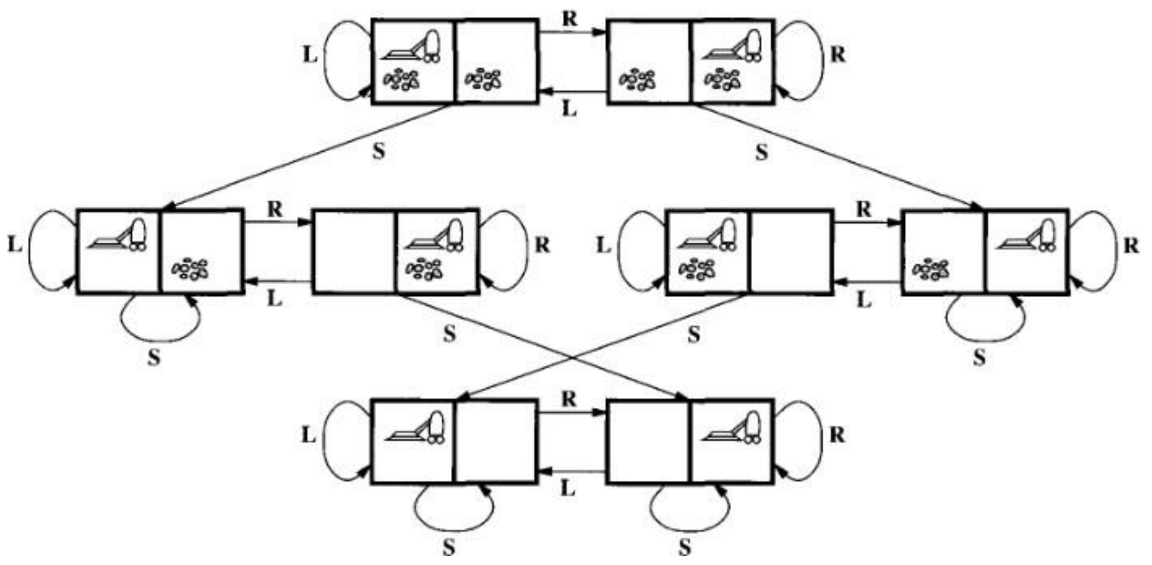


**Figure 2.1 The state space for the vacuum world.**
**Arcs denote actions: L = Left, R = Right**

**The 8-puzzle**

An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the balank space can slide into the space. The object is to reach the goal state, as shown in Figure 2.4

**Example: The 8-puzzle**

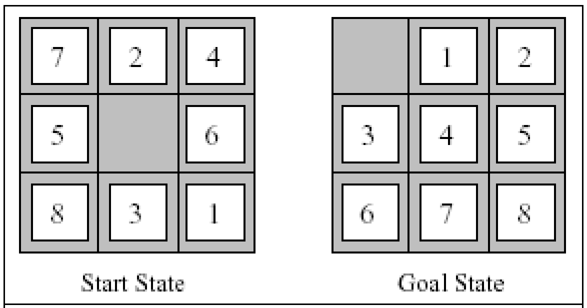


**Figure 2.2 A typical instance of 8-puzzle**

The problem formulation is as follows:

- **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- **Successor function** : This generates the legal states that result from trying the four actions(blank moves Left, Right, Up or down).
- **Goal Test** : This checks whether the state matches the goal configuration shown in Figure(Other goal configurations are possible)
- **Path cost** : Each step costs 1,so the path cost is the number of steps in the path.

**The 8-puzzle** belongs to the family **of sliding-block puzzles**, which are often used as test problems for new search algorithms in AI. This general class is known as NP-complete. The **8-puzzle** has 9!/2 = 181,440 reachable states and is easily solved.

The **15 puzzle** ( 4 x 4 board ) has around 1.3 trillion states, an the random instances can be solved optimally in few milli seconds by the best search algorithms.

The **24-puzzle** (on a 5 x 5 board) has around $10^{25}$ states and random instances are still quite difficult to solve optimally with current machines and algorithms.

**8-Queens problem**

The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other.(A queen attacks any piece in the same row, column or diagonal).

Figure 2.3 shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.

An **Incremental formulation** involves operators that augments the state description, starting with an empty state. For 8-queens problem, this means each action adds a queen to the state. A **complete-state formulation** starts with all 8 queens on the board and move them around. In either case the path cost is of no interest because only the final state counts.

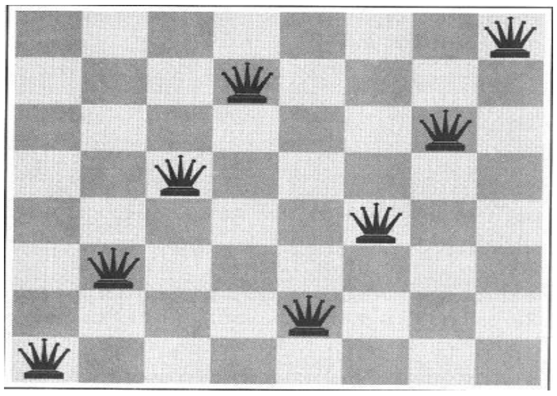


**Figure 2.3 8-queens problem**

The first incremental formulation one might try is the following:

- **States**: Any arrangement of 0 to 8 queens on board is a state.
- **Initial state**: No queen on the board.
- **Successor function**: Add a queen to any empty square.
- **Goal Test**: 8 queens are on the board, none attacked.

In this formulation, we have 64.63…57 = 3 x $10^{14}$ possible sequences to investigate.

A better formulation would prohibit placing a queen in any square that is already attacked.

- **States** : Arrangements of n queens ( 0 <= n < = 8 ),one per column in the left most columns, with no queen attacking another are states.
- **Successor function** : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queen state space from 3 x $10^{14}$ to just 2057,and solutions are easy to find.

For the 100 queens the initial formulation has roughly $10^{400}$ states whereas the improved formulation has about $10^{52}$ states. This is a huge reduction, but the improved state space is still too big for the algorithms to handle.

## 2.4 REAL-WORLD PROBLEMS

### ROUTE-FINDING PROBLEM

Route-finding problem is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and airline travel planning systems.

## 2.5 AIRLINE TRAVEL PROBLEM

The **airline travel problem** is specifies as follows**:**

- **States:** Each is represented by a location (e.g., an airport) and the current time.
- **Initial state:** This is specified by the problem.
- **Successor function:** This returns the states resulting from taking any scheduled flight (further specified by seat class and location),leaving later than the current time plus the within-airport transit time, from the current airport to another.
- **Goal Test:** Are we at the destination by some prespecified time?
- **Path cost:** This depends upon the monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of date, type of air plane, frequent-flyer mileage awards, and so on.

## 2.6 TOURING PROBLEMS

**Touring problems** are closely related to route-finding problems, but with an important difference. Consider for example, the problem, “Visit every city at least once” as shown in Romania map.

As with route-finding the actions correspond to trips between adjacent cities. The state space, however, is quite different.

The initial state would be “In Bucharest; visited{Bucharest}”.

A typical intermediate state would be “In Vaslui;visited {Bucharest, Urziceni,Vaslui}”.

The goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

## 2.7 THE TRAVELLING SALESPERSON PROBLEM(TSP)

Is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour. The problem is known to be **NP-hard**. Enormous efforts have been expended to improve the capabilities of TSP algorithms. These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.

### VLSI layout

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem is split into two parts: **cell layout** and **channel routing**.

**ROBOT navigation**

**ROBOT navigation** is a generalization of the route-finding problem. Rather than a discrete set of routes, a robot can move in a continuous space with an infinite set of possible actions and states. For a circular Robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that also must be controlled, the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

## 2.8 AUTOMATIC ASSEMBLY SEQUENCING

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen, there will be no way to add some part later without undoing some work already done. Another important assembly problem is protein design, in which the goal is to find a sequence of Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

## 2.9 INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching, looking for answers to questions, for related information, or for shopping deals. The searching techniques consider internet as a graph of nodes(pages) connected by links.

**Different Search Algorithm**

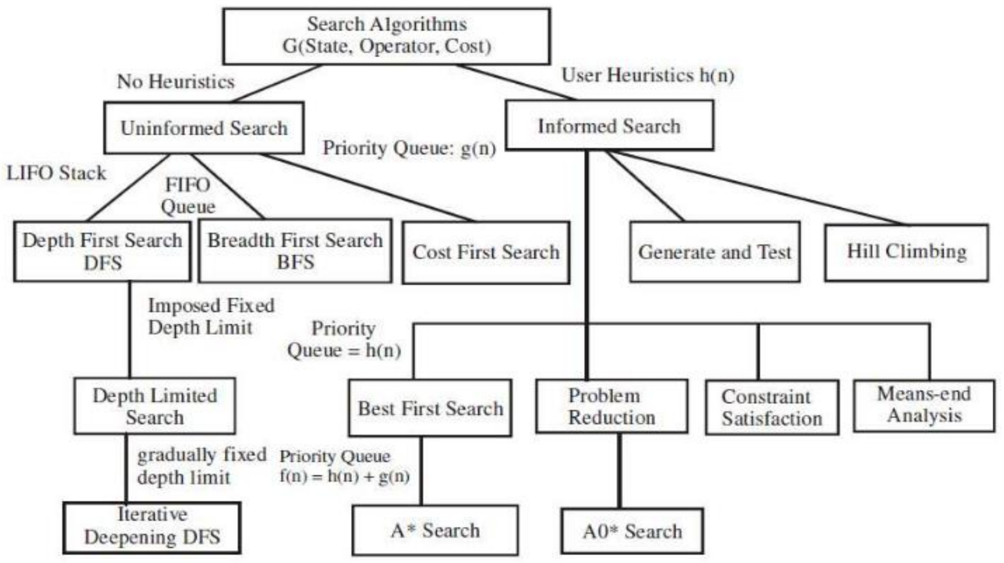


**Figure 2.4 Different Search Algorithms**

## 2.10 UNINFORMED SEARCH STRATGES

**Uninformed Search Strategies** have no additional information about states beyond that provided in the **problem definition**.

**Strategies** that know whether one non goal state is "more promising" than another are called

**Informed search or heuristic search** strategies.

There are five uninformed search strategies as given below.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

**Breadth-first search**

- Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In otherwards, calling TREE-SEARCH (problem, FIFO-QUEUE()) results in breadth-first-search. The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.

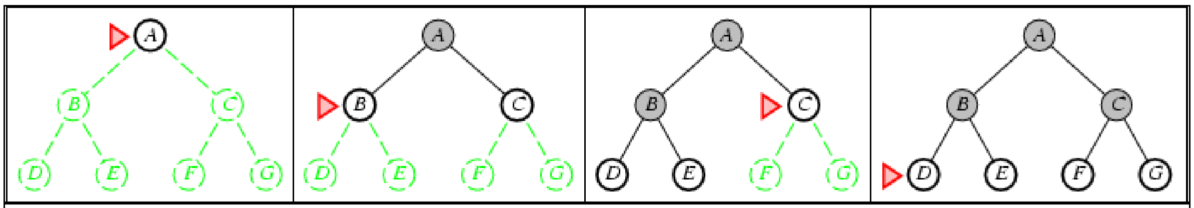
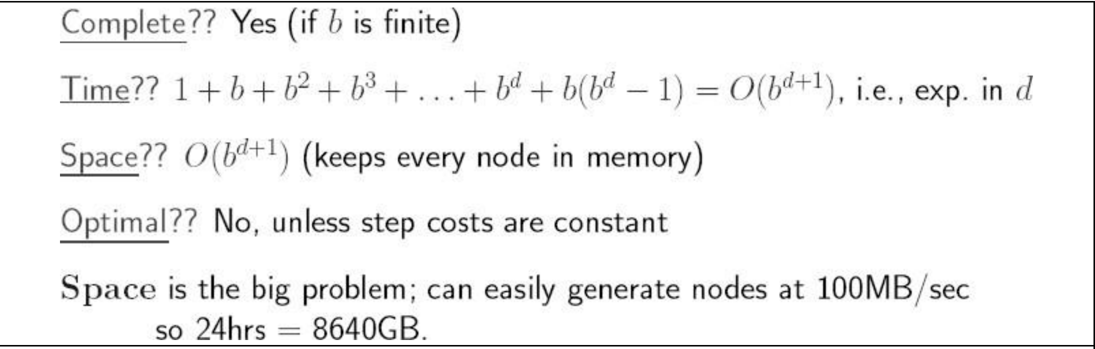


**Figure 2.5 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.**

**Properties of breadth-first-search**



Complete?? Yes (if $b$ is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

**Figure 2.6 Breadth-first-search properties**

**Time complexity for BFS**

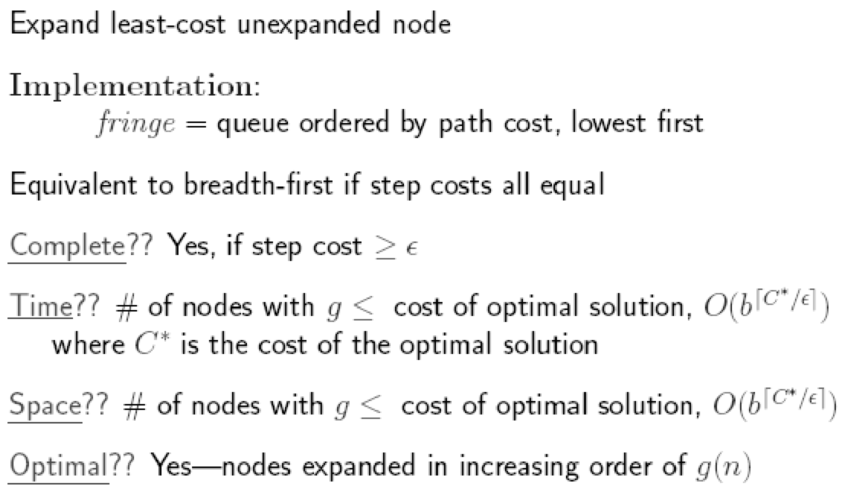
Assume every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of $b^2$ at the second level. Each of these generates b more nodes, yielding $b^3$ nodes at the third level, and so on. Now suppose, that the solution is at depth d. In the worst case, we would expand all but the last node at level d, generating $b^{d+1}$ - b nodes at level d+1.

Then the total number of nodes generated is $b + b^2 + b^3 + \ldots + b^d + (b^{d+1} + b) = O(b^{d+1}).$

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. The space compleity is, therefore, the same as the time complexity

## 2.11 UNIFORM-COST SEARCH

Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost. Uniform-cost search does not care about the number of steps a path has, but only about their total cost.



Expand least-cost unexpanded node

Implementation:
*fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where $C^*$ is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

## 2.12 DEPTH-FIRST-SEARCH

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in Figure 1.31. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.

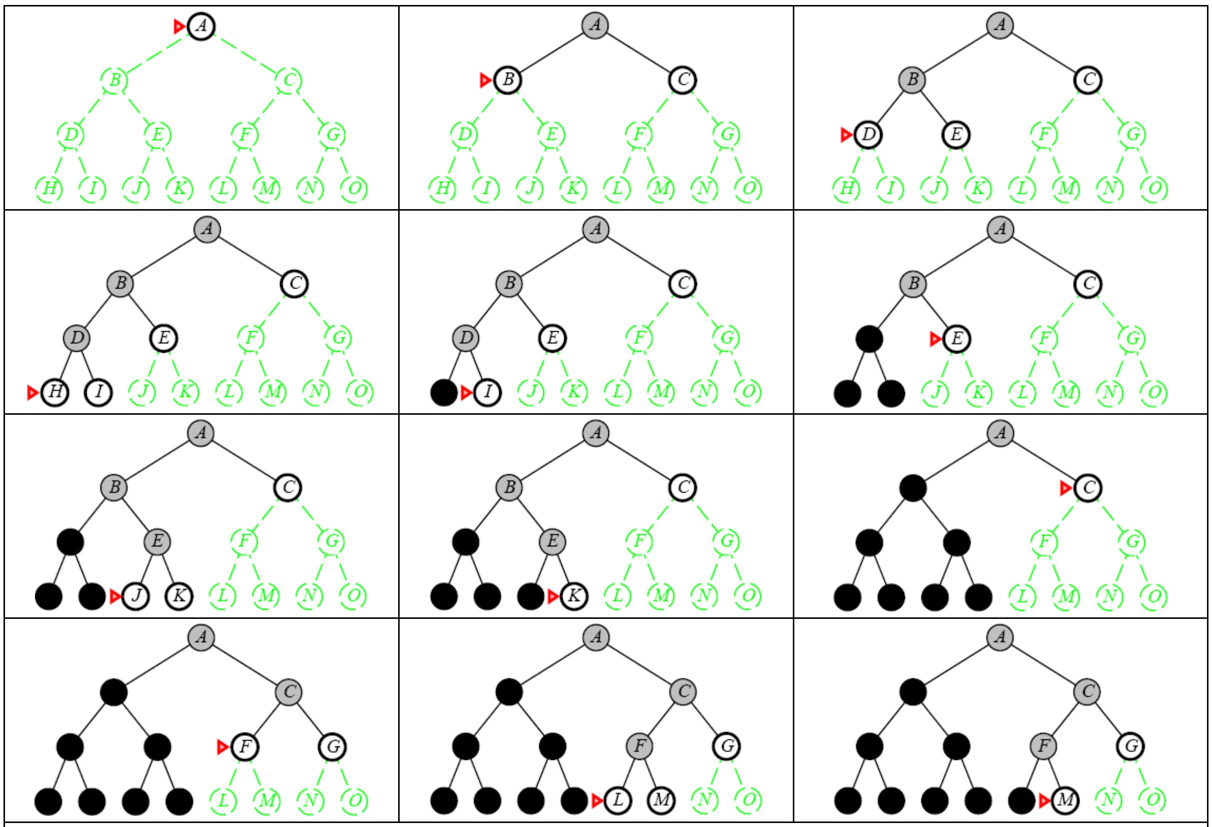


**Figure 2.7 Depth-first-search on a binary tree. Nodes that have been expanded and have node scendants in the fringe can be removed from the memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.**

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.

Depth-first-search has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded, it can be removed from the memory, as soon as its descendants have been fully explored (Refer Figure 2.7).

For a state space with a branching factor b and maximum depth m, depth-first-search requires storage of only bm + 1 nodes.

Using the same assumptions as Figure, and assuming that nodes at the same depth as the goal node have no successors, we find the depth-first-search would require 118 kilobytes instead of 10 petabytes, a factor of 10 billion times less space.

**Drawback of Depth-first-search**

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example, depth-first-search will explore the entire left subtree even if node C is a goal node.

## 2.12 BACKTRACKING SEARCH

A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm)

**DEPTH-LIMITED-SEARCH**

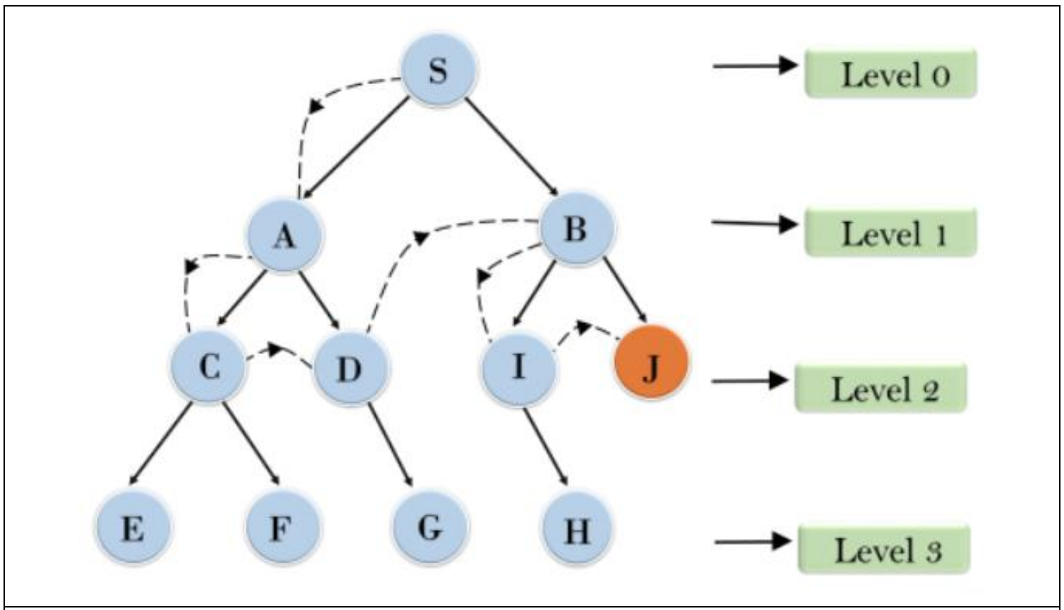


**Figure 2.8 Depth-limited-search**

The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre- determined depth limit l. That is, nodes at depth l are treated as if they have no successors. This approach is called **depth-limited-search**. The depth limit soves the infinite path problem.

Depth limited search will be nonoptimal if we choose l > d. Its time complexity is O($b^l$) and its space complete is O(bl). Depth-first-search can be viewed as a special case of depth-limited search with l = oo Sometimes, depth limits can be based on knowledge of the problem. For, example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, So l = 10 is a possible choice. However, it can be shown that any city can be reached from any other city in at most 9 steps. This number known as the **diameter** of the state space, gives us a better depth limit.

Depth-limited-search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first-search algorithm. The pseudocode for recursive depth- limited-search is shown in Figure.

It can be noted that the above algorithm can terminate with two kinds of failure : the standard *failure* value indicates no solution; the *cutoff*value indicates no solution within the depth limit. Depth-limited search = depth-first search with depth limit l,returns cut off if any path is cut off by depth limit

```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff return
Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit) function Recursive-
DLS(node, problem, limit) returns solution/fail/cutoff cutoff-occurred?      false
if Goal-Test(problem,State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do result
Recursive-DLS(successor, problem, limit)  if result = cutoff then cutoff_occurred?true
else if result not = failure then return result
ifcutoff_occurred? then return cutoff else return failure
```

**Figure 2.9 Recursive implementation of Depth-limited-search**

## 2.13 ITERATIVE DEEPENING DEPTH-FIRST SEARCH

Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search, that finds the better depth limit. It does this by gradually increasing the limit – first 0,then 1,then 2, and so on – until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node. The algorithm is shown in Figure.

Iterative deepening combines the benefits of depth-first and breadth-first-search Like depth-first-search, its memory requirements are modest; O(bd) to be precise.

Like Breadth-first-search, it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.

Figure shows the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree, where the solution is found on the fourth iteration.

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

**Figure 2.10 The iterative deepening search algorithm, which repeatedly applies depth-limited- search with increasing limits. It terminates when a solution is found or if the depth limited search returns *failure,* meaning that no solution exists.**

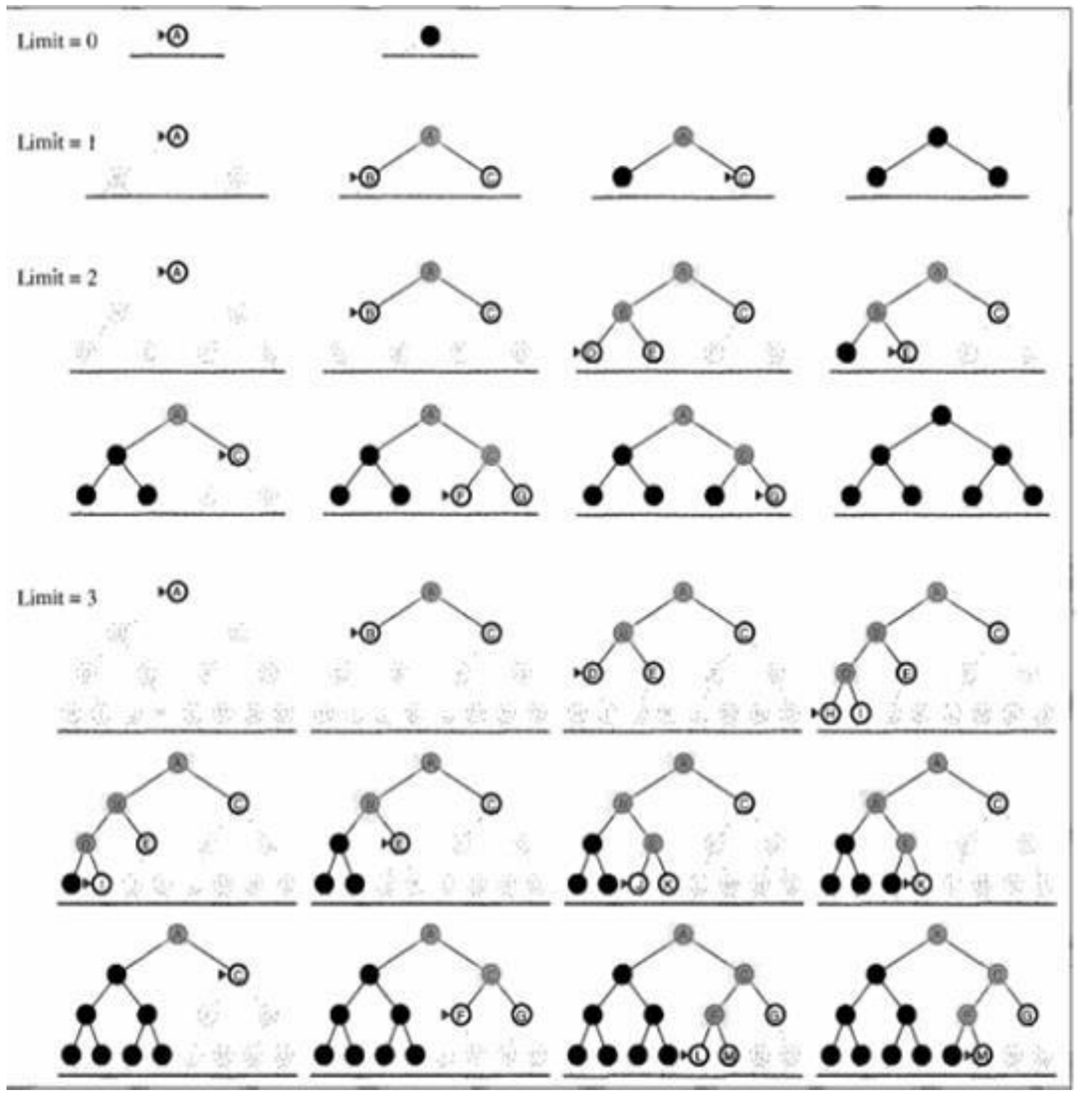


**Figure 2.11 Four iterations of iterative deepening search on a binary tree**

Iterative search is not as wasteful as it might seem

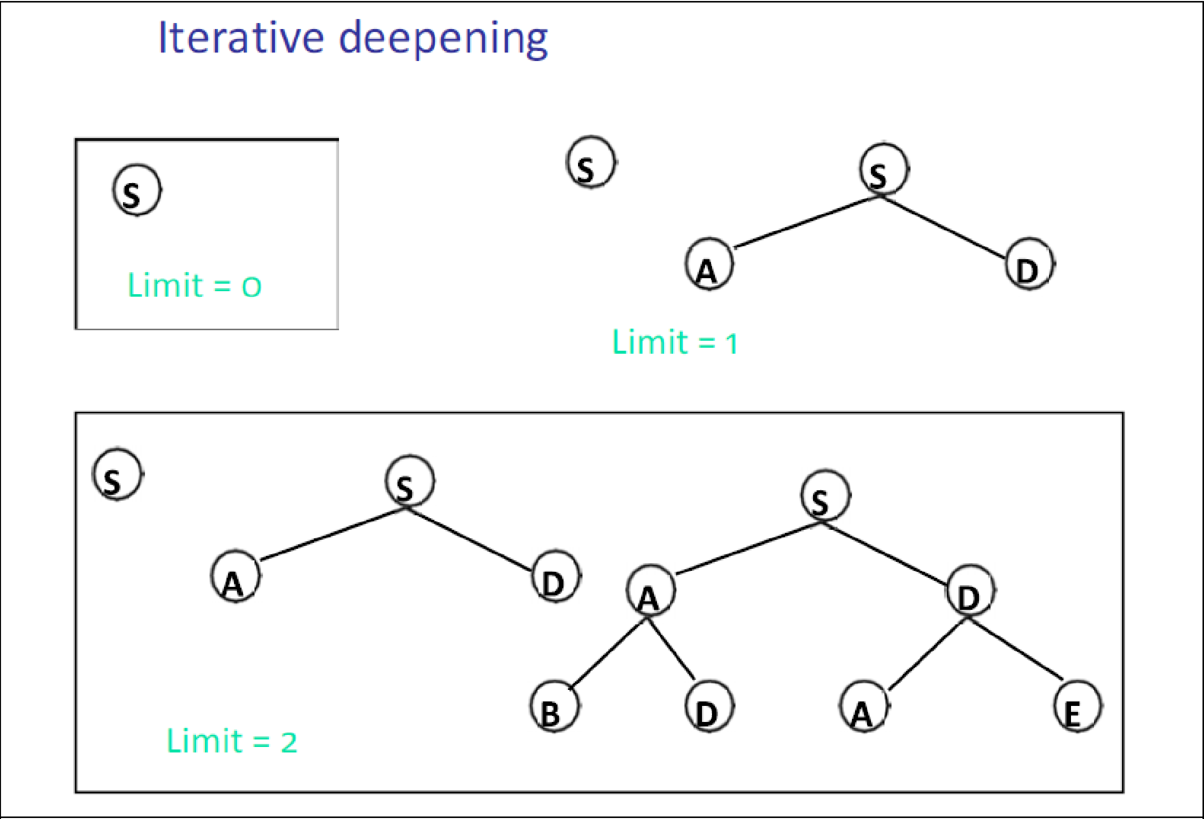
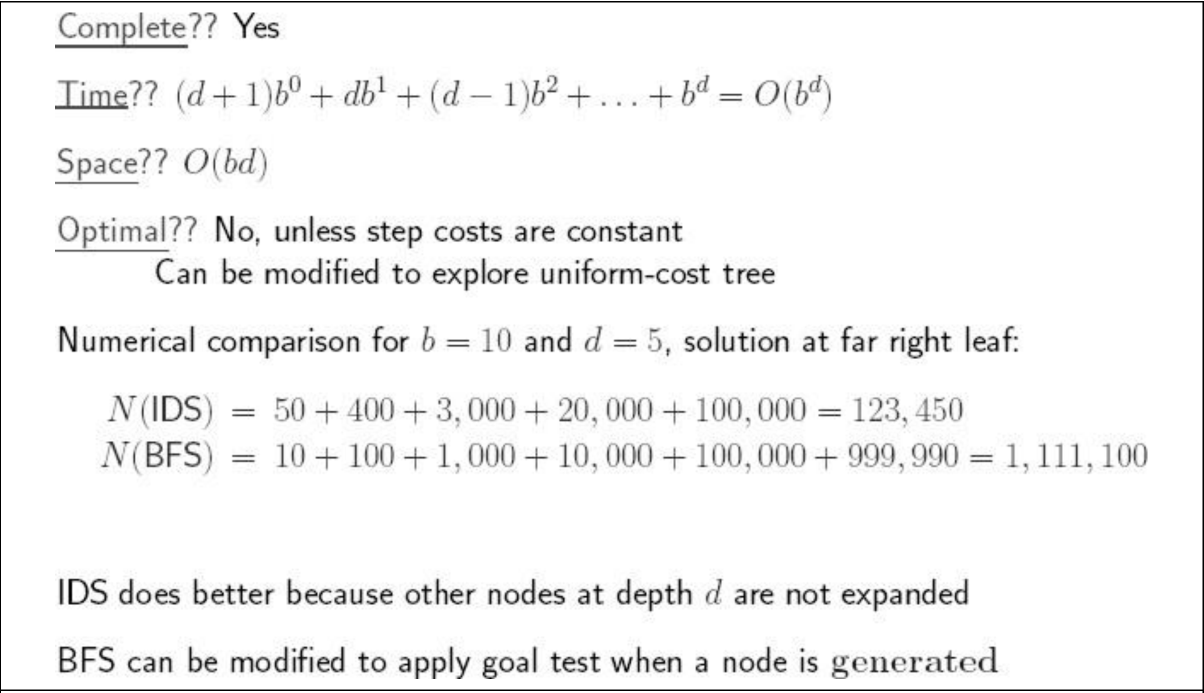


**Figure 2.12 Iterative search is not as wasteful as it might seem**

Properties of iterative deepening search



Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth $d$ are not expanded

BFS can be modified to apply goal test when a node is generated

**Figure 2.13 Properties of iterative deepening search**

**Bidirectional Search**

The idea behind bidirectional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle

The motivation is that $b^{d/2} + b^{d/2}$ much less than, or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

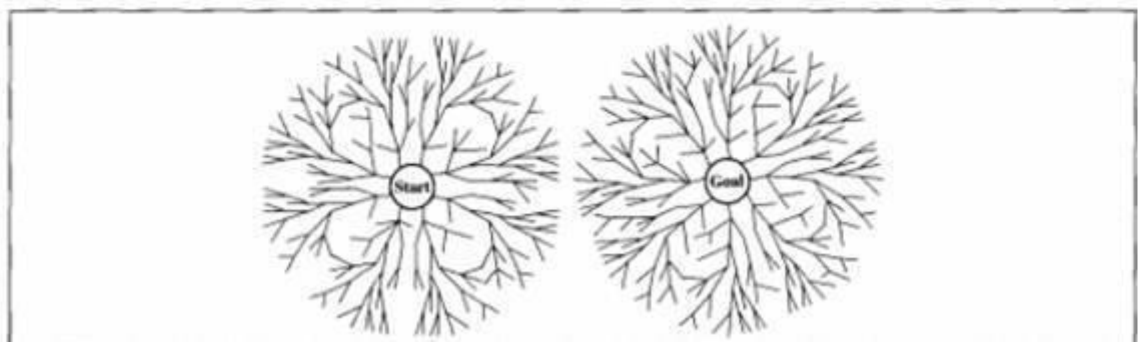


**Figure 2.14 A schematic view of a bidirectional search that is about to succeed, when a Branch from the Start node meets a Branch from the goal node.**

- Before moving into bidirectional search let’s first understand a few terms.
- Forward Search: Looking in-front of the end from start.
- Backward Search: Looking from end to the start back-wards.
- So Bidirectional Search as the name suggests is a combination of forwarding and backward search. Basically, if the average branching factor going out of node / fan-out, if fan-out is less, prefer forward search. Else if the average branching factor is going into a node/fan in is less (i.e. fan-out is more), prefer backward search.
- We must traverse the tree from the start node and the goal node and wherever they meet the path from the start node to the goal through the intersection is the optimal solution. The BS Algorithm is applicable when generating predecessors is easy in both forward and backward directions and there exist only 1 or fewer goal states.

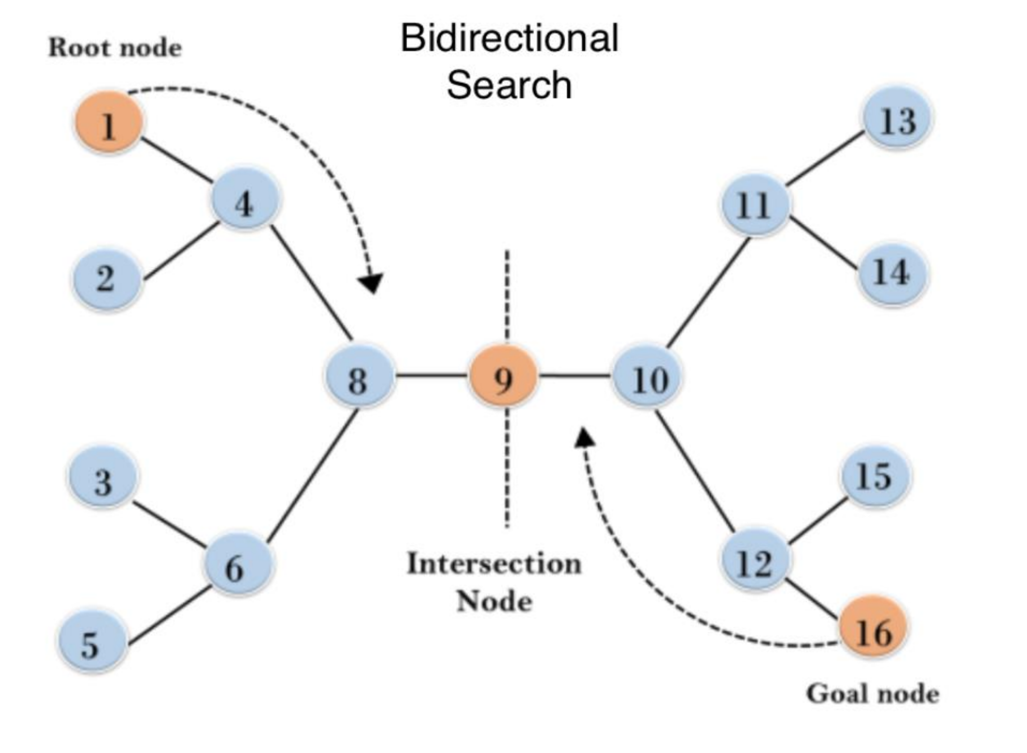
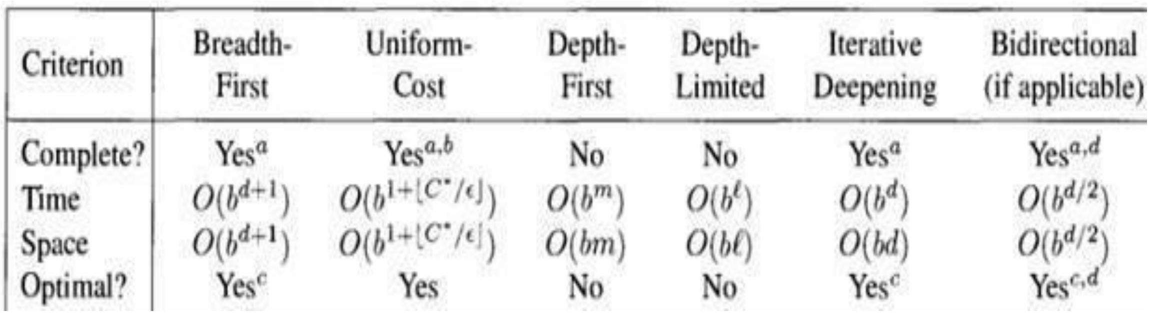


**Figure 2.15 Comparing Uninformed Search Strategies**



| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Complete? | Yes[a] | Yes[a,b] | No | No | Yes[a] | Yes[a,d] |
| Time | $O(b^{d+1})$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^\ell)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^{d+1})$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(b\ell)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal? | Yes[c] | Yes | No | No | Yes[c] | Yes[c,d] |

**Figure 2.16 Evaluation of search strategies, b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs >= E for positive E; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.**

## 2.14 SEARCHING WITH PARTIAL INFORMATION

- Different types of incompleteness lead to three distinct problem types:
- **Sensorless problems** (conformant): If the agent has no sensors at all
- **Contingency problem**: if the environment if partially observable or if action are uncertain (adversarial)
- **Exploration problems**: When the states and actions of the environment are unknown.

- No sensor
- Initial State(1,2,3,4,5,6,7,8)
- After action [Right] the state (2,4,6,8)
- After action [Suck] the state (4, 8)
- After action [Left] the state (3,7)
- After action [Suck] the state (8)
- Answer : [Right, Suck, Left, Suck] coerce the world into state 7 without any sensor
- Belief State: Such state that agent belief to be there

Partial knowledge of states and actions:

- *sensorless or conformant problem*
  - Agent may have no idea where it is; solution (if any) is a sequence.
- *contingency problem*
  - Percepts provide *new* information about current state; solution is a tree or policy; often interleave search and execution.
  - If uncertainty is caused by actions of another agent: *adversarial problem*
- *exploration problem*
  - When states and actions of the environment are unknown.

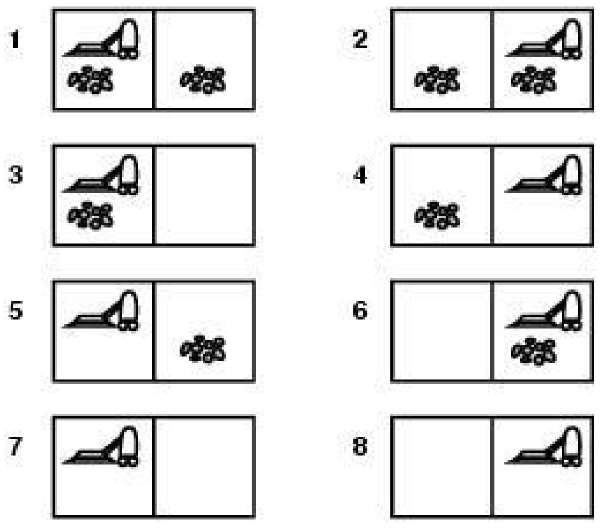


**Figure 2.17 states and actions of the environment are unknown**

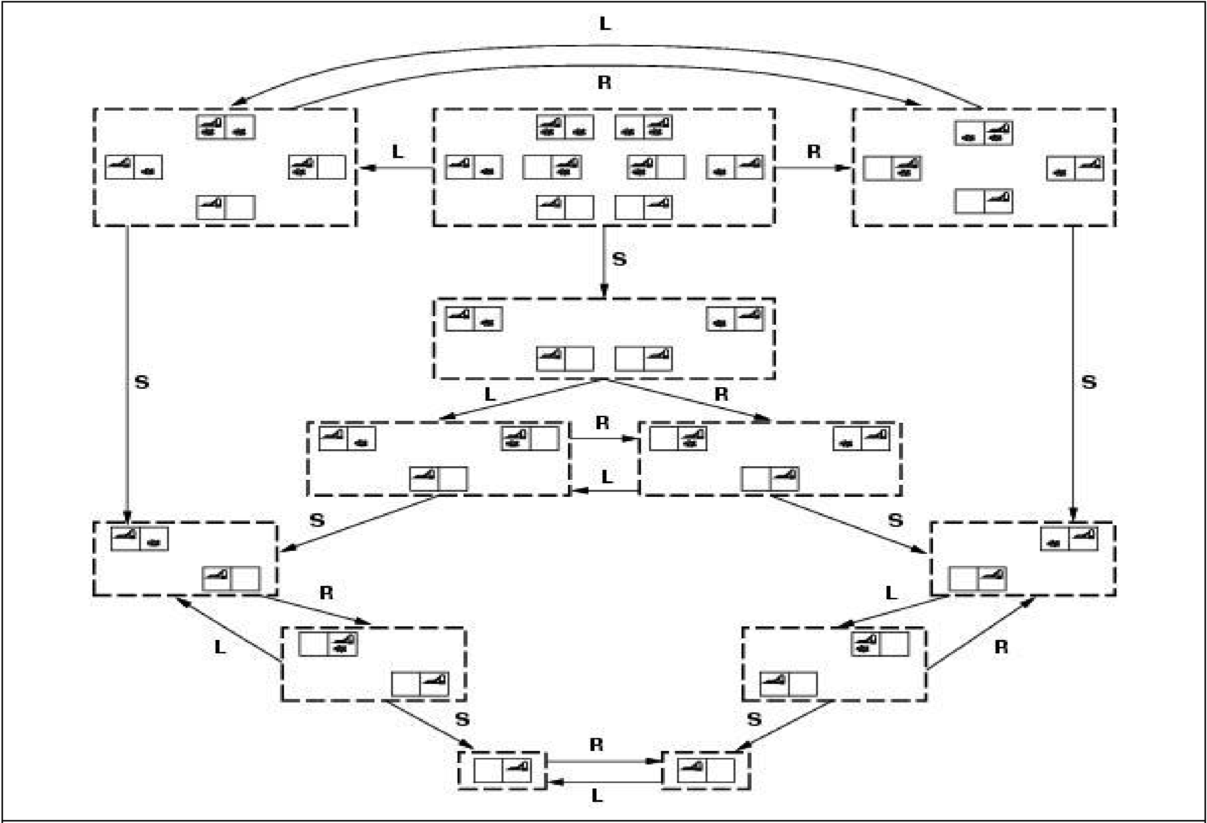


**Figure 2.18 states and actions**

Contingency, start in {1,3}.

Murphy's law, Suck *can* dirty a clean carpet. Local sensing: dirt, location only.

- Percept = [L,Dirty] ={1,3}
- [Suck] = {5,7}
- [Right] ={6,8}
- [Suck] in {6}={8} (Success)
- BUT [Suck] in {8} = failure Solution??
- Belief-state: no fixed action sequence guarantees solution

Relax requirement:

- [*Suck*, *Right*, if *[R,dirty]* then *Suck*]
- Select actions based on contingencies arising during execution.

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space.

In AI, where the graph is represented implicitly by the initial state and successor function, the complexity is expressed in terms of three quantities:

**b**, the **branching factor** or maximum number of successors of any node;

**d**, the **depth** of the **shallowest goal node**; and

**m**, the **maximum length** of any path in the state space.

**Search-cost -** typically depends upon the time complexity but can also include the term for memory usage.

**Total–cost –** It combines the search-cost and the path cost of the solution found.

## 2.15 INFORMED SEARCH AND EXPLORATION

### Informed (Heuristic) Search Strategies

**Informed search strategy** is one that uses problem-specific knowledge beyond the definition of the problem itself. It can find solutions more efficiently than uninformed strategy.

### Best-first search

**Best-first search** is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function** f(n). The node with lowest evaluation is selected for expansion, because the evaluation measures the distance to the goal.

This can be implemented using a priority-queue, a data structure that will maintain the fringe in ascending order of f-values.

## 2.16 HEURISTIC FUNCTIONS

A **heuristic function** or simply a **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.

The key component of Best-first search algorithm is a **heuristic function**, denoted by h(n): h(n) = estimated cost of the **cheapest path** from node n to a **goal node**.

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via a **straight-line distance** from Arad to Bucharest (Figure 2.19).

Heuristic function are the most common form in which additional knowledge is imparted to the search algorithm.

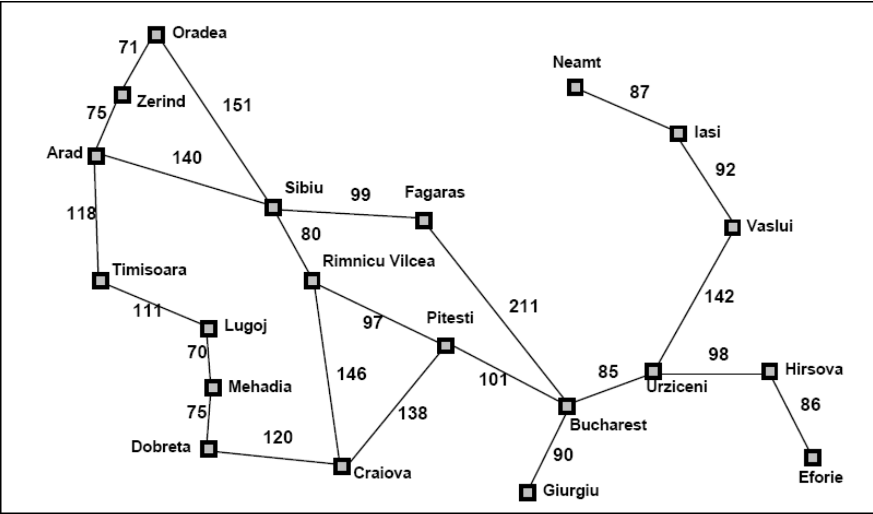
### Greedy Best-first search

**Greedy best-first search** tries to expand the node that is closest to the goal, on the grounds that this is likely to a solution quickly.

It evaluates the nodes by using the heuristic function f(n) = h(n).

Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in Figure. For example, the initial state is In(Arad),and the straight line distance heuristic hSLD (In(Arad)) is found to be 366.

Using the **straight-line distance** heuristic **hSLD,** the goal state can be reached faster.



| Straight−line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 178 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 98 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

**Figure 2.19 Values of $h_{SLD}$ - straight line distances to Bucharest**

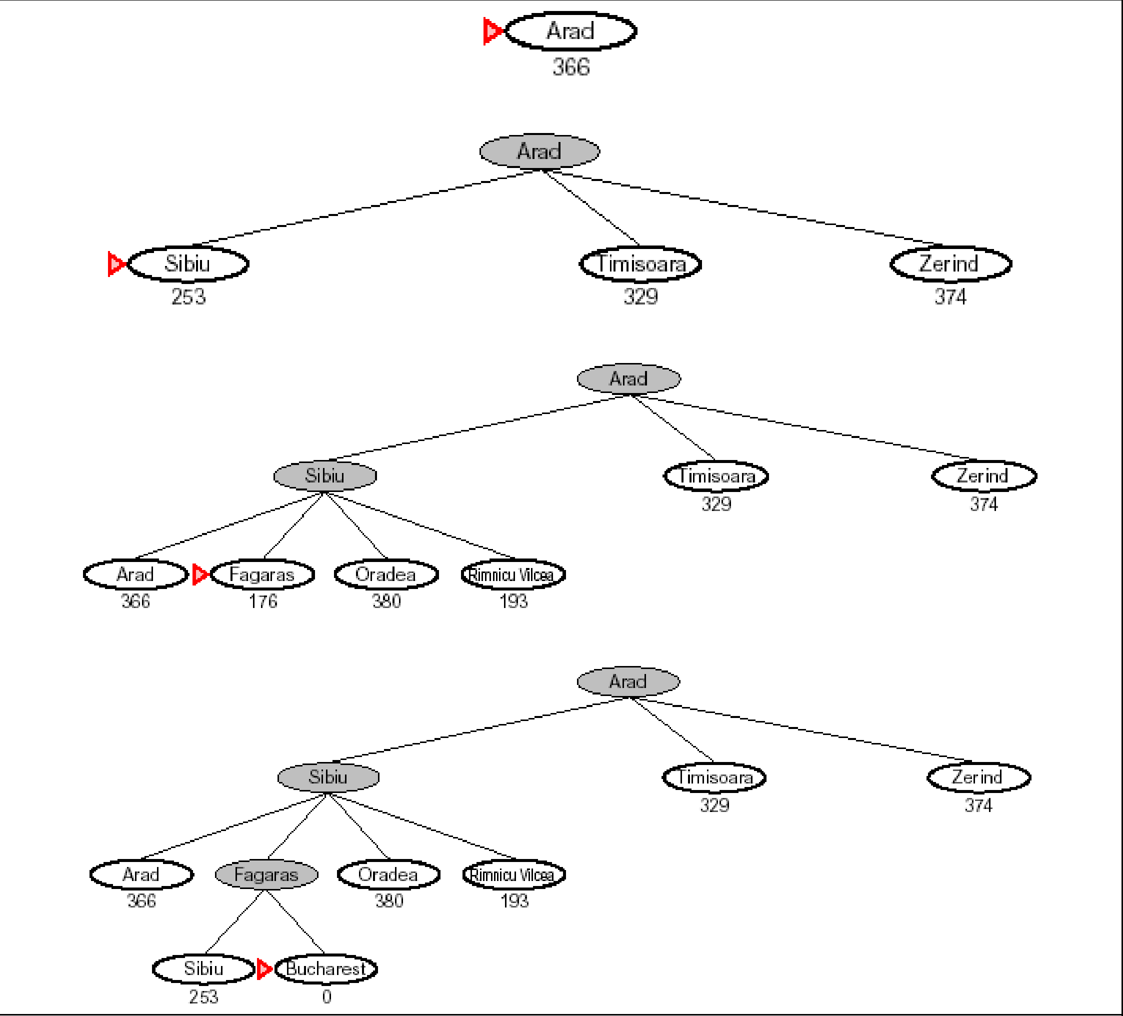


**Figure 2.20 progress of greedy best-first search**

Figure shows the progress of greedy best-first search using hSLD to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest. Fagaras in turn generates Bucharest, which is the goal.

**Properties of greedy search**

- **Complete:** No–can get stuck in loops, e.g., Iasi !Neamt !Iasi !Neamt !

  Complete in finite space with repeated-state checking
- **Time:** O(bm), but a good heuristic can give dramatic improvement
- **Space:** O(bm) - keeps all nodes in memory
- Optimal: No

Greedy best-first search is not optimal, and it is incomplete.

The worst-case time and space complexity is $O(b^m)$,where m is the maximum depth of the search space.

## A$^*$ SEARCH

**A* Search** is the most widely used form of best-first search. The evaluation function f(n) is obtained by combining

(1) **g(n) =** the cost to reach the node, and

(2) **h(n) =** the cost to get from the node to the **goal** :

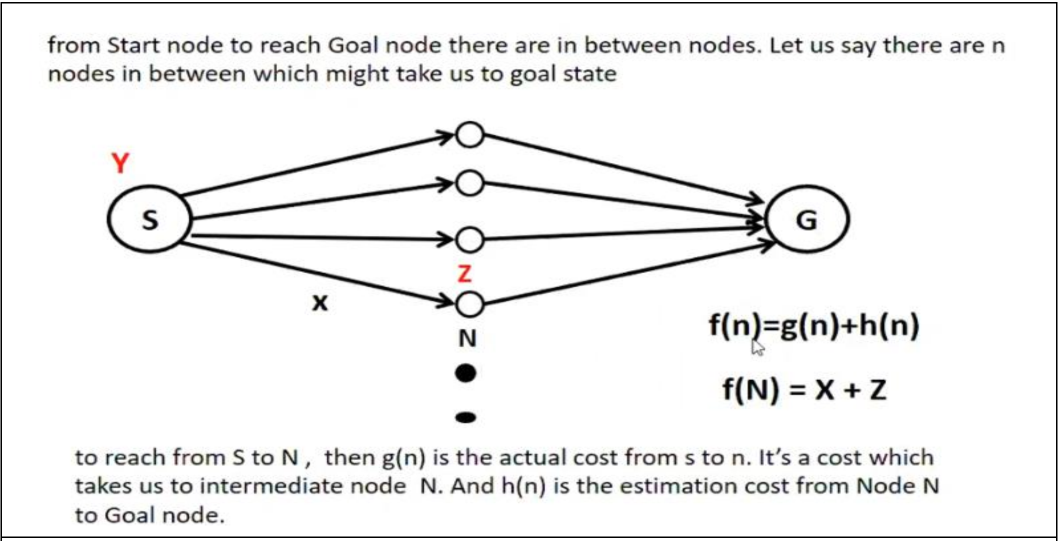
$$f(n) = g(n) + h(n).$$

A$^*$ Search is both optimal and complete. A$^*$ is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance hSLD. It cannot be an overestimate.

A$^*$ Search is optimal if h(n) is an admissible heuristic – that is, provided that h(n) never overestimates the cost to reach the goal.

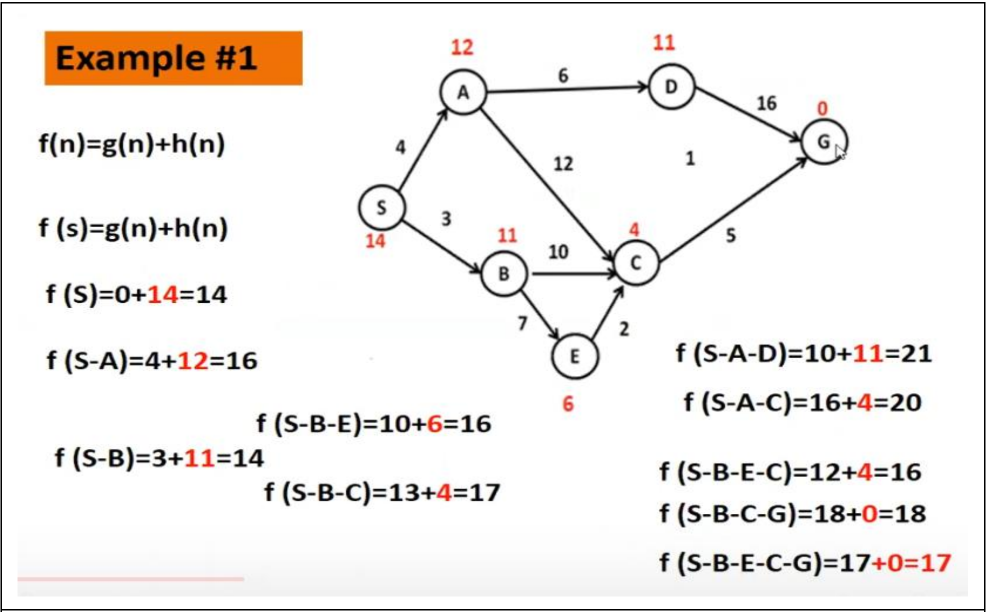
An obvious example of an admissible heuristic is the straight-line distance hSLD that we used in getting to Bucharest. The progress of an A$^*$ tree search for Bucharest is shown in Figure

The values of ‘g ‘ are computed from the step costs shown in the Romania map(figure). Also the values of hSLD are given in Figure


from Start node to reach Goal node there are in between nodes. Let us say there are n nodes in between which might take us to goal state

to reach from S to N , then g(n) is the actual cost from s to n. It's a cost which takes us to intermediate node N. And h(n) is the estimation cost from Node N to Goal node.

**Figure 2.21 A* Search**


Example #1

f(n)=g(n)+h(n)

f (s)=g(n)+h(n)

f (S)=0+14=14

f (S-A)=4+12=16

f (S-B)=3+11=14

f (S-B-E)=10+6=16

f (S-B-C)=13+4=17

f (S-A-D)=10+11=21

f (S-A-C)=16+4=20

f (S-B-E-C)=12+4=16

f (S-B-C-G)=18+0=18

f (S-B-E-C-G)=17+0=17

**Figure 2.22 Example A* Search**

## 2.17 LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.
- In such cases, we can **use local search algorithms.** They operate using a **single current state** (rather than multiple paths) and generally move only to neighbors of that state.

- The important applications of these class of problems are (a) integrated-circuit design, (b) Factory-floor layout, (c) job-shop scheduling, (d) automatic programming, (e) telecommunications network optimization, (f) Vehicle routing, and (g) portfolio management.

Key advantages of Local Search Algorithms

(1) They use very little memory – usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic algorithms are unsuitable.

## 2.18 OPTIMIZATION PROBLEMS

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the **best state** according to an **objective function**.

### State Space Landscape

To understand local search, it is better explained using **state space landscape** as shown in Figure.

A landscape has both "**location**" (defined by the state) and "**elevation**" (defined by the value of the heuristic cost function or objective function).

If elevation corresponds to **cost**, then the aim is to find the **lowest valley** – a **global minimum**; if elevation corresponds to an **objective function**, then the aim is to find the **highest peak** – a **global maximum.**

Local search algorithms explore this landscape. A complete local search algorithm always finds a **goal** if one exists; an **optimal** algorithm always finds a **global minimum/maximum**.

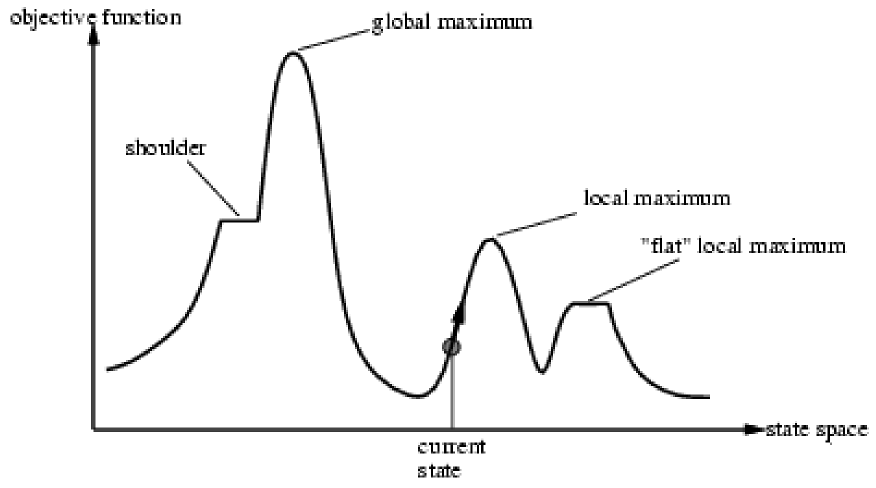


**Figure 2.23 A one dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text**

**Hill-climbing search**

The **hill-climbing** search algorithm as shown in figure, is simply a loop that continually moves in the direction of increasing value – that is, **uphill**. It terminates when it reaches a "**peak**" where no neighbor has a higher value.

```
function HILL-CLIMBING( problem) return a state that is a local maximum
        input: problem, a problem
  local variables: current, a
                   node.
          neighbor, a node.

        current ←MAKE-NODE(INITIAL-STATE[problem])
        loop do
                neighbor ← a highest valued successor of current
                if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
                current ←neighbor
```

**Figure 2.24 The hill-climbing search algorithm (steepest ascent version), which is the most basic local search technique. At each step the current node is replaced by the best neighbor; the neighbor with the highest VALUE. If the heuristic cost estimate h is used, we could find the neighbor with the lowest h.**

Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Greedy algorithms often perform quite well. **Problems with hill-climbing**

Hill-climbing often gets stuck for the following reasons :

- **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go
- **Ridges**: A ridge is shown in Figure 2.10. Ridges results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux**: A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which it is possible to make progress.

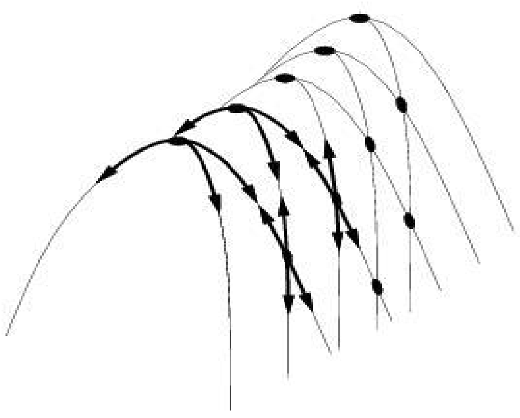


**Figure 2.25 Illustration of why ridges cause difficulties for hill-climbing. The grid of states(dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available options point downhill.**

Hill-climbing variations

- Stochastic hill-climbing
    - Random selection among the uphill moves.
    - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
    - cfr. stochastic hill climbing by generating successors randomly until a better one is found.
- Random-restart hill-climbing
    - Tries to avoid getting stuck in local maxima.

**Simulated annealing search**

A hill-climbing algorithm that never makes “downhill” moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor choosen uniformly at random from the set of successors – is complete, but extremely inefficient.

Simulated annealing is an algorithm that combines hill-climbing with a random walk in someway that yields both efficiency and completeness.

Figure shows simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the “badness” of the move – the amount  E by which the evaluation is worsened.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability $e^{\Delta E/T}$
```

**Figure 2.26 The simulated annealing search algorithm, a version of stochastic hill climbing where some downhill moves are allowed.**

## Genetic algorithms

A Genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by modifying a single state

Like beam search, Gas begin with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet – most commonly, a string of 0s and 1s. For example, an 8 8-quuens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 x log2 8 = 24 bits.

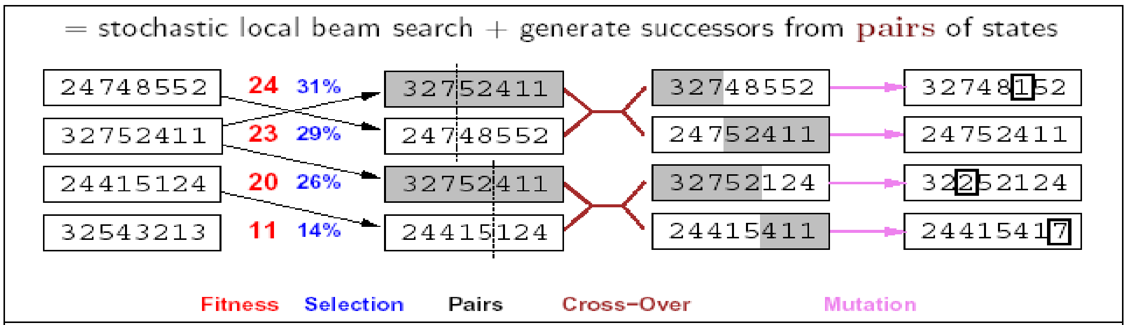


**Figure 2.27 Genetic algorithm**

Figure shows a population of four 8-digit strings representing 8-queen states. The production of the next generation of states is shown in Figure

In (b) each state is rated by the evaluation function or the **fitness function**.

In (c),a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b).

Figure describes the algorithm that implements all these steps.

```
function GENETIC_ALGORITHM( population, FITNESS-FN) return an individual
    input: population, a set of individuals
        FITNESS-FN, a function which determines the quality of the individual
    repeat
        new_population←empty set
        loop for ifrom 1 to SIZE(population) do
            x ←RANDOM_SELECTION(population, FITNESS_FN)
            y ←RANDOM_SELECTION(population,
                FITNESS_FN)
            child ←REPRODUCE(x,y)
                if (small random probability) then child □
                MUTATE(child ) add child to new_population
        population ←new_population
    until some individual is fit enough or enough time has elapsed
    return the best individual
```

**Figure 2.28 A genetic algorithm.**

## 2.29 CONSTRAINT SATISFACTION PROBLEMS(CSP)

A **Constraint Satisfaction Problem**(or CSP) is defined by a set of **variables,**X1,X2,….Xn, and a set of constraints C1,C2,…,Cm. Each variable Xi has a nonempty **domain** D,of possible **values**.

Each constraint Ci involves some subset of variables and specifies the allowable combinations of values for that subset.

A **State** of the problem is defined by an **assignment** of values to some or all of the variables,{Xi = vi,Xj = vj,…}. An assignment that does not violate any constraints is called a **consistent** or **legal assignment.** A complete assignment is one in which every variable is mentioned, and a **solution** to a CSP is a complete assignment that satisfies all the constraints.

Some CSPs also require a solution that maximizes an **objective function**.

**Example for Constraint Satisfaction Problem**

Figure shows the map of Australia showing each of its states and territories. We are given the task of coloring each region either red, green, or blue in such a way that the neighboring regions have the same color. To formulate this as CSP, we define the variable to be the regions

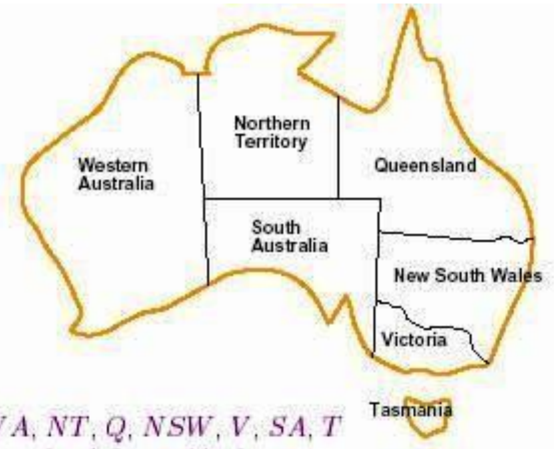
:WA,NT,Q,NSW,V,SA, and T. The domain of each variable is the set {red,green,blue}.The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for WA and NT are the pairs

{(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.

The constraint can also be represented more succinctly as the inequality WA not = NT, provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions such as

{ WA = red, NT = green, Q = red, NSW = green, V = red,SA = blue,T = red}.

It is helpful to visualize a CSP as a constraint graph, as shown in Figure 2.29. The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.



Variables $WA, NT, Q, NSW, V, SA, T$
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

**Figure 2.29 Principle states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.**

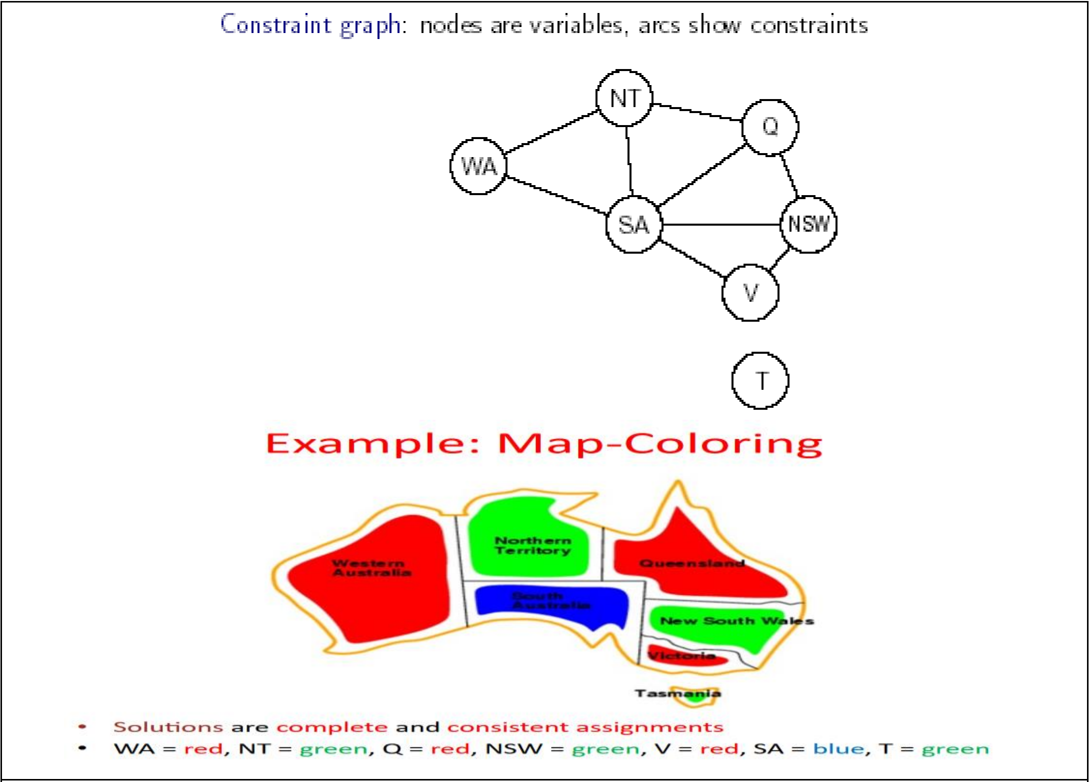

Constraint graph: nodes are variables, arcs show constraints
NT
Q
WA
SA
NSW
V
T
Example: Map-Coloring
Western Australia
Northern Territory
Queensland
South Australia
New South Wales
Victoria
Tasmania
Solutions are complete and consistent assignments
WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

**Figure 2.30 Mapping Problem**

CSP can be viewed as a standard search problem as follows:

- **Initial state**: the empty assignment {},in which all variables are unassigned.
- **Successor function**: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
- **Goal test**: the current assignment is complete.
- **Path cost**: a constant cost(E.g.,1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables.

Depth first search algorithms are popular for CSPs

**Varieties of CSPs**

**(i) Discrete variables Finite domains**

The simplest kind of CSP involves variables that are **discrete** and have **finite domains.** Map coloring problems are of this kind. The 8-queens problem can also be viewed as finite-domain

CSP, where the variables Q1,Q2,…..Q8 are the positions each queen in columns 1,….8 and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any variable in a CSP is d, then the number of possible complete assignments is O($d^n$) – that is, exponential in the number of variables. Finite domain CSPs include **Boolean CSPs**, whose variables can be either *true* or *false*. **Infinite domains**

Discrete variables can also have **infinite domains** – for example, the set of integers or the set of strings. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combination of values. Instead a constraint language of algebric inequalities such as Startjob1 + 5 <= Startjob3.

**(ii) CSPs with continuous domains**

CSPs with continuous domains are very common in real world. For example in operation research field, the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and manoeuvre are continuous-valued variables that must obey a variety of astronomical, precedence and power constraints. The best known category of continuous-domain CSPs is that of **linear programming** problems, where the constraints must be linear inequalities forming a *convex* region. Linear programming problems can be solved in time polynomial in the number of variables.

**Varieties of constraints**

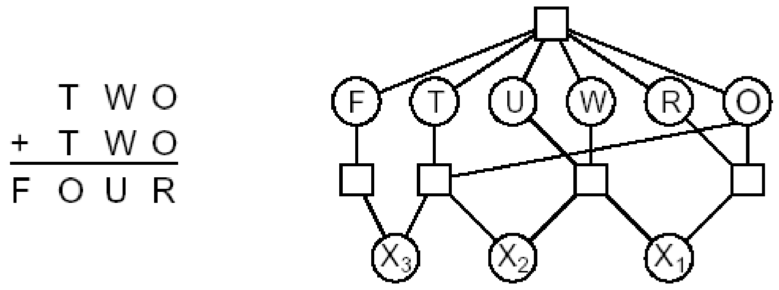
**(i)** **unary constraints** involve a single variable.

Example : SA # green

**(ii)** Binary constraints involve paris of variables.

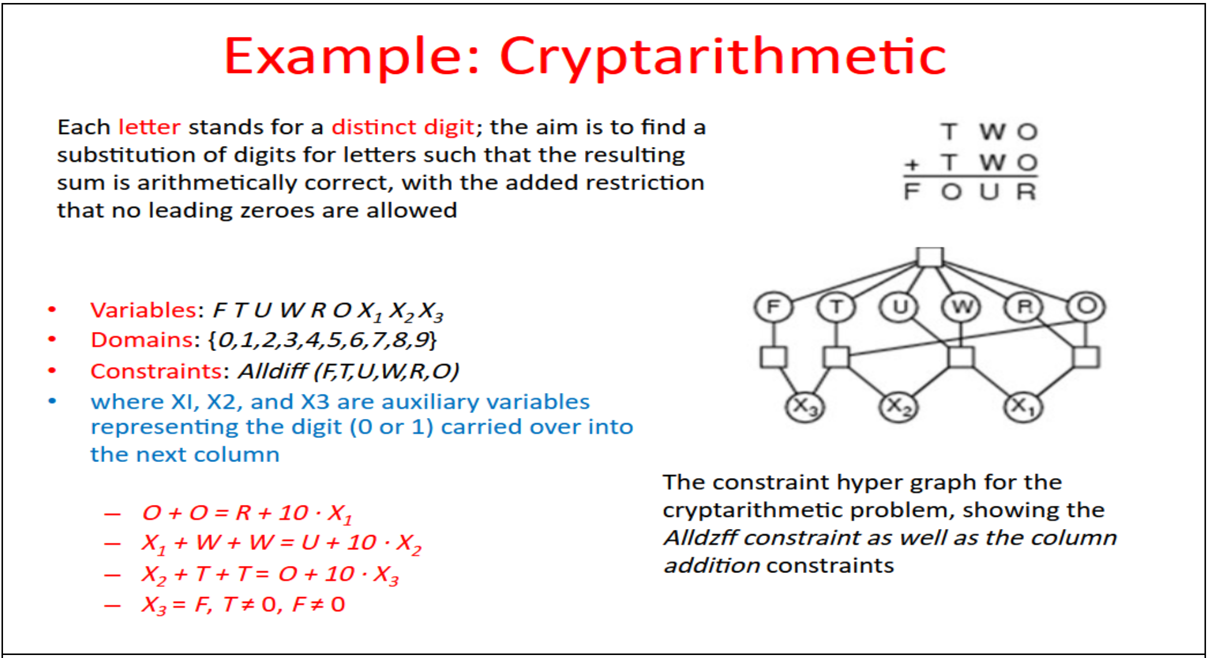
Example : SA # WA

**(iii)** Higher order constraints involve 3 or more variables. Example :cryptarithmetic puzzles.



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0,1,2,3,4,5,6,7,8,9\}$
Constraints
*alldiff*$(F,T,U,W,R,O)$
S $O+O=R+10\cdot X_1$, etc.

**Figure 2.31 cryptarithmetic puzzles**.

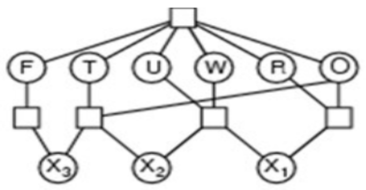


## Example: Cryptarithmetic

Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed

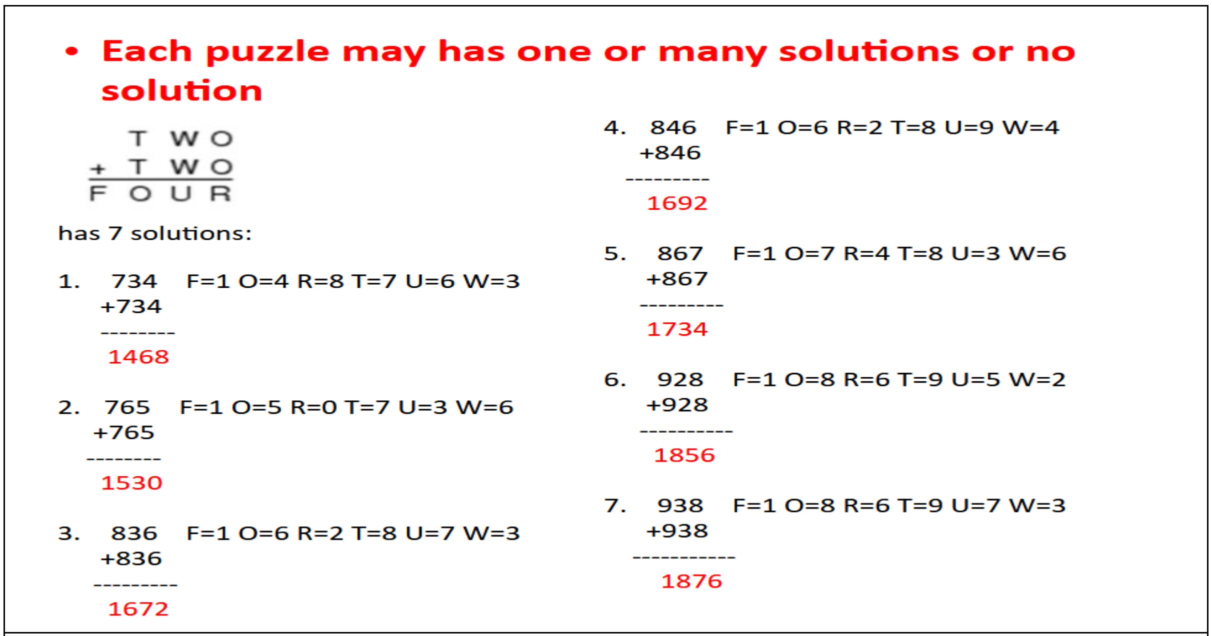
```
   T W O
 + T W O
 F O U R
```

- Variables: *F T U W R O $X_1$ $X_2$$X_3$*
- Domains: {*0,1,2,3,4,5,6,7,8,9*}
- Constraints: *Alldiff (F,T,U,W,R,O)*
- where XI, X2, and X3 are auxiliary variables representing the digit (0 or 1) carried over into the next column
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$



The constraint hyper graph for the cryptarithmetic problem, showing the *Alldzff constraint as well as the column addition* constraints

**Figure 2.32 Cryptarithmetic puzzles-Solution**



- **Each puzzle may has one or many solutions or no solution**

```
   T W O
 + T W O
 F O U R
```

has 7 solutions:

```
1.   734   F=1 O=4 R=8 T=7 U=6 W=3
    +734
    --------
     1468

2.  765   F=1 O=5 R=0 T=7 U=3 W=6
   +765
   --------
    1530

3.   836   F=1 O=6 R=2 T=8 U=7 W=3
    +836
   ---------
     1672

4.  846   F=1 O=6 R=2 T=8 U=9 W=4
   +846
  ---------
    1692

5.   867   F=1 O=7 R=4 T=8 U=3 W=6
    +867
   ---------
    1734

6.   928   F=1 O=8 R=6 T=9 U=5 W=2
    +928
   ----------
     1856

7.   938   F=1 O=8 R=6 T=9 U=7 W=3
    +938
  -----------
     1876
```

**Figure 2.33 Numerical Solution**

### Backtracking Search for CSPs

The term **backtracking search** is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure

```
function BACKTRACKING-SEARCH($csp$) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, $csp$)

function RECURSIVE-BACKTRACKING($assignment$, $csp$) returns soln/failure
   if $assignment$ is complete then return $assignment$
   $var$ ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[$csp$], $assignment$, $csp$)
   for each $value$ in ORDER-DOMAIN-VALUES($var$, $assignment$, $csp$) do
       if $value$ is consistent with $assignment$ given CONSTRAINTS[$csp$] then
           add $\{var = value\}$ to $assignment$
           $result$ ← RECURSIVE-BACKTRACKING($assignment$, $csp$)
           if $result \neq failure$ then return $result$
           remove $\{var = value\}$ from $assignment$
   return $failure$
```

**Figure 2.34 A simple backtracking algorithm for constraint satisfaction problem. The algorithm is modeled on the recursive depth-first search**

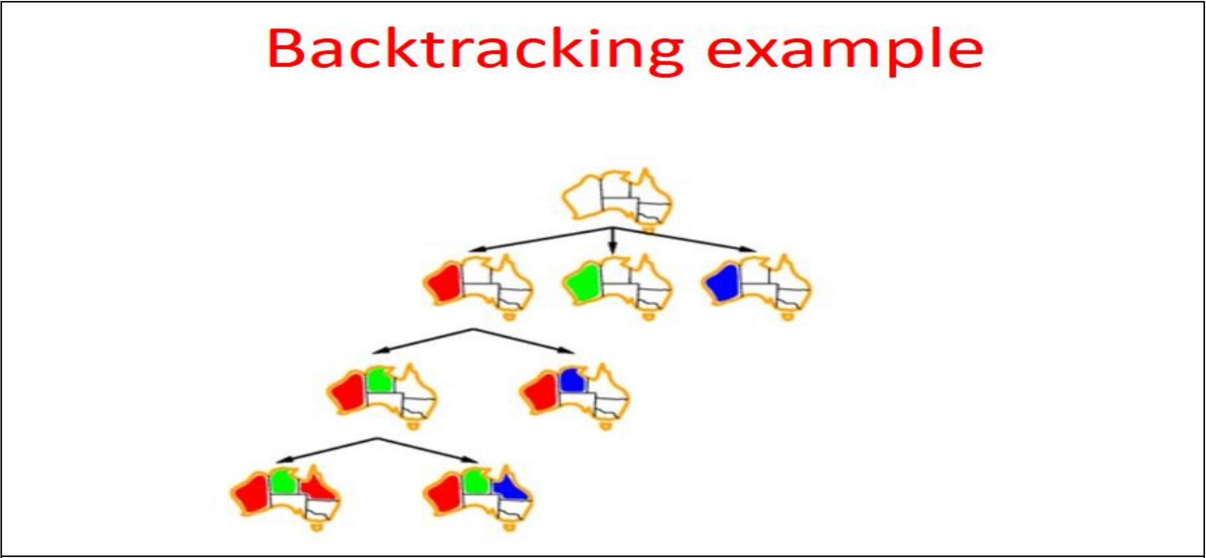


**Figure 2.34 Part of the search tree generated by simple backtracking for the map-coloring problem**

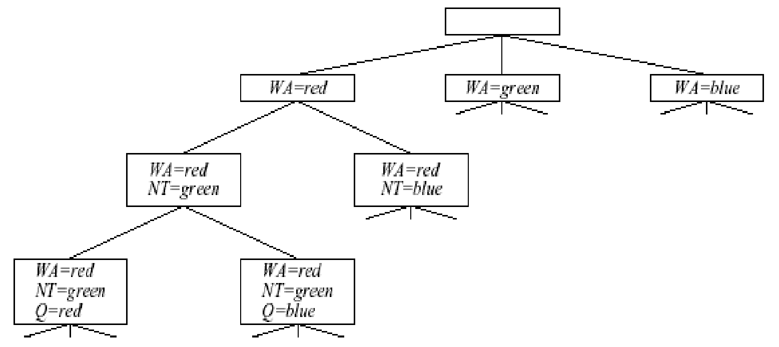
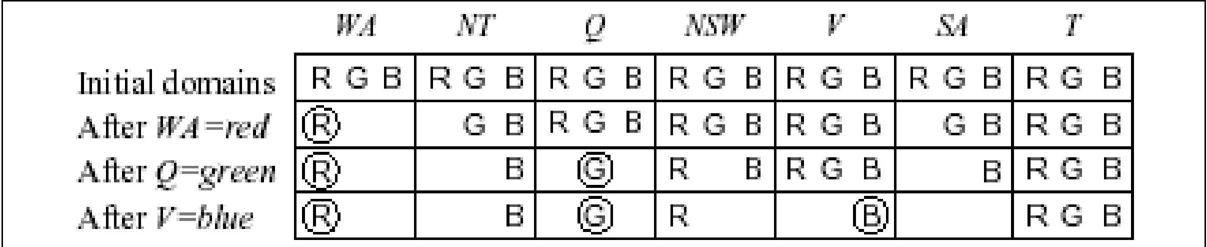


**Figure 2.35 Part of search tree generated by simple backtracking for the map coloring problem.**

## Forward checking

One way to make better use of constraints during search is called forward checking. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X. Figure 5.6 shows the progress of a map-coloring search with forward checking.



| | *WA* | *NT* | *Q* | *NSW* | *V* | *SA* | *T* |
|---|---|---|---|---|---|---|---|
| Initial domains | R G B | R G B | R G B | R G B | R G B | R G B | R G B |
| After *WA*=*red* | R | G B | R G B | R G B | R G B | G B | R G B |
| After *Q*=*green* | R | B | G | R B | R G B | B | R G B |
| After *V*=*blue* | R | B | G | R | B | | R G B |

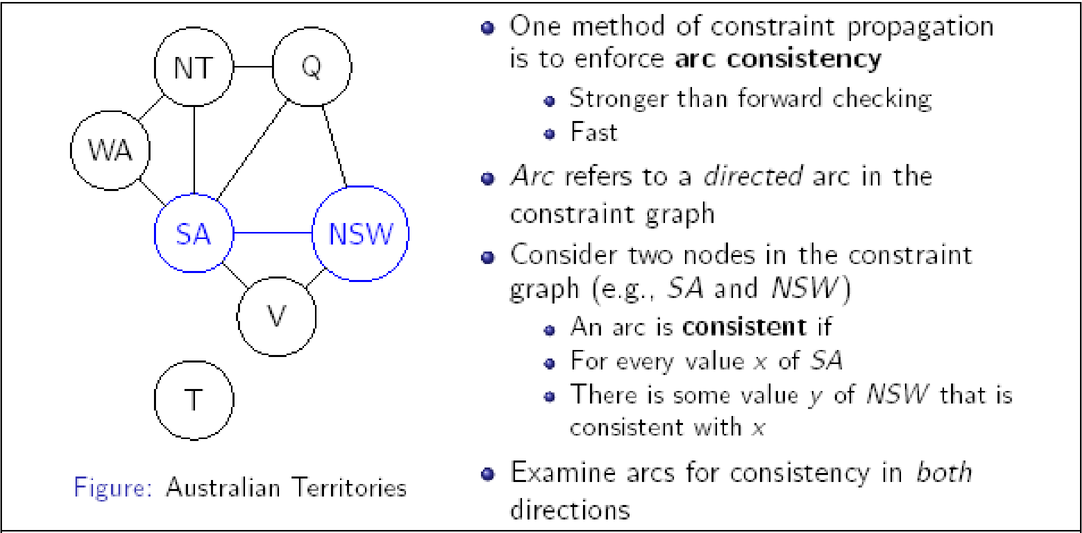
**Figure 2.36 The progress of a map-coloring search with forward checking. WA = *red* is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After *Q* = *green*, *green* is deleted from the domain of *NT, SA*, and *NSW*. After *V* = *blue, blue,* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.**

## Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them.

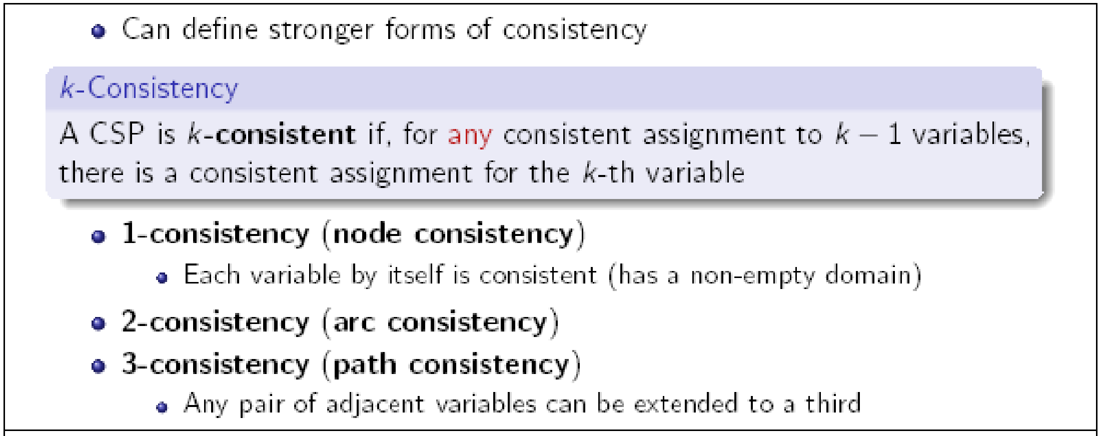
**Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables.

## Arc Consistency



- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value *x* of *SA*
  - There is some value *y* of *NSW* that is consistent with *x*
- Examine arcs for consistency in *both* directions

**Figure 2.37 Arc Consistency**



- Can define stronger forms of consistency

**$k$-Consistency**

A CSP is **$k$-consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the $k$-th variable

- **1-consistency** (**node consistency**)
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency** (**arc consistency**)
- **3-consistency** (**path consistency**)
  - Any pair of adjacent variables can be extended to a third

**Figure 2.38 Arc Consistency –CSP**

### k-Consistency

### Local Search for CSPs

- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

### The Structure of Problems Problem Structure

- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

### Independent Subproblems

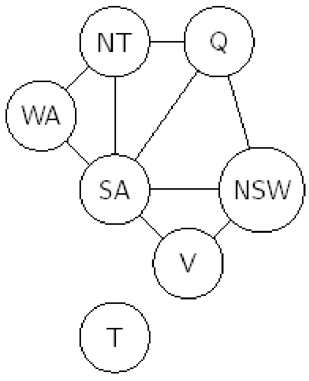


Figure: Australian Territories

- $T$ is not connected
- Coloring $T$ and coloring remaining nodes are **independent subproblems**
- *Any* solution for $T$ combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

**Figure 2.39 Independent Subproblems**

**Tree-Structured CSPs**

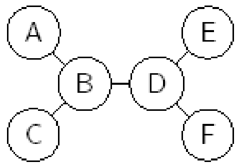


Figure: Tree-Structured CSP

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

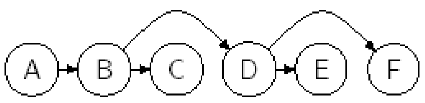


Figure: Linear ordering

**Figure 2.40 Tree-Structured CSPs**

## 2.30 ADVERSARIAL SEARCH

Competitive environments, in which the agent's goals are in conflict, give rise to **adversarial search** problems – often known as **games.**

### Games

Mathematical **Game Theory**, a branch of economics, views any **multiagent environment** as a **game** provided that the impact of each agent on the other is "significant", regardless of whether the agents are cooperative or competitive. In, **AI**, "games" are deterministic, turn-taking, two-player, zero-sum games of perfect information. This means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the **utility values** at the end of the game are always equal and opposite. For example, if one player wins the game of chess(+1),the other player necessarily loses(-1). It is this opposition between the agents' utility functions that makes the situation **adversarial.**

### Formal Definition of Game

We will consider games with two players, whom we will call **MAX** and **MIN**. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A **game** can be formally defined as a **search problem** with the following components:

- The **initial state**, which includes the board position and identifies the player to move.
- A **successor function**, which returns a list of (*move*, *state*) pairs, each indicating a legal move and the resulting state.

- A **terminal test**, which describes when the game is over. States where the game has ended are called **terminal states**.
- A **utility function** (also called an objective function or payoff function), which give a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values+1,-1, or 0. he payoffs in backgammon range from +192 to -192.

**Game Tree**

The **initial state** and **legal moves** for each side define the **game tree** for the game. Figure 2.18 shows the part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing a 0 until we reach leaf nodes corresponding to the terminal states such that one player has three in a row or all the squares are filled. He number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN. It is the MAX's job to use the search tree (particularly the utility of terminal states) to determine the best move.

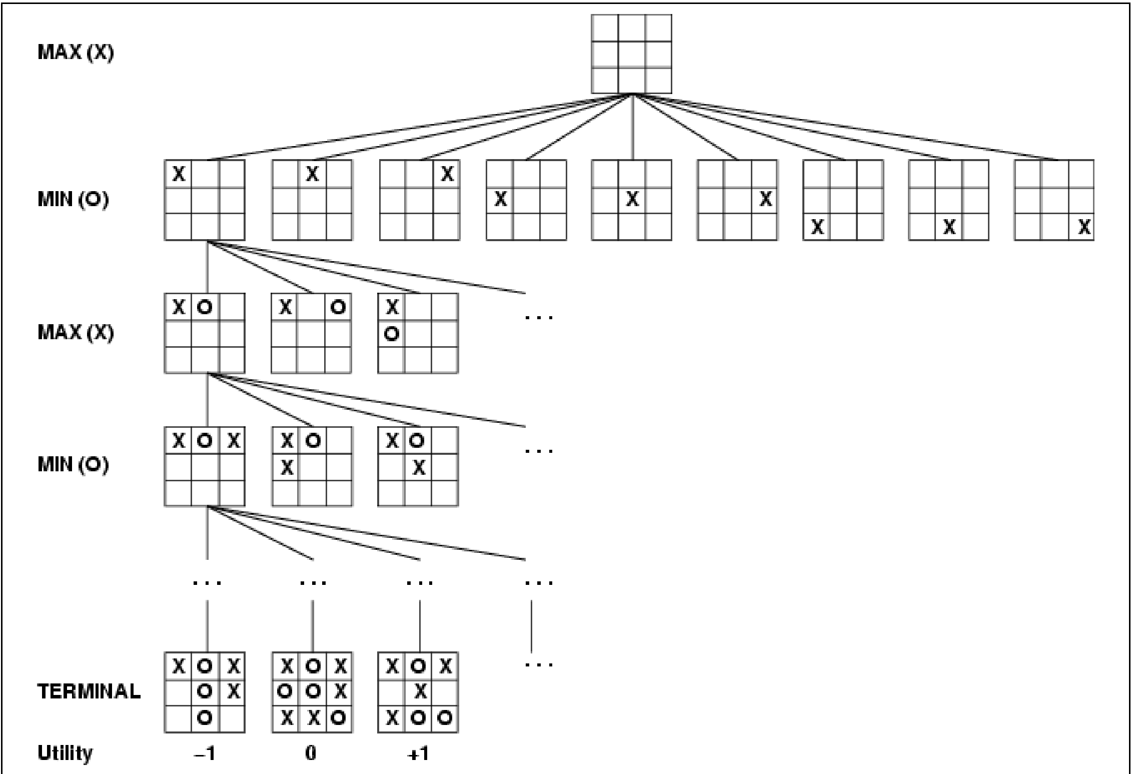


**Figure 2.41 A partial search tree. The top node is the initial state, and MAX move first, placing an X in an empty square.**

**Optimal Decisions in Games**

In normal search problem, the **optimal solution** would be a sequence of move leading to a **goal state** – a terminal state that is a win. In a game, on the other hand, MIN has something to say about it, MAX therefore must find a contingent **strategy**, which specifies MAX's move in the **initial state**, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN those moves, and so on. An **optimal strategy** leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

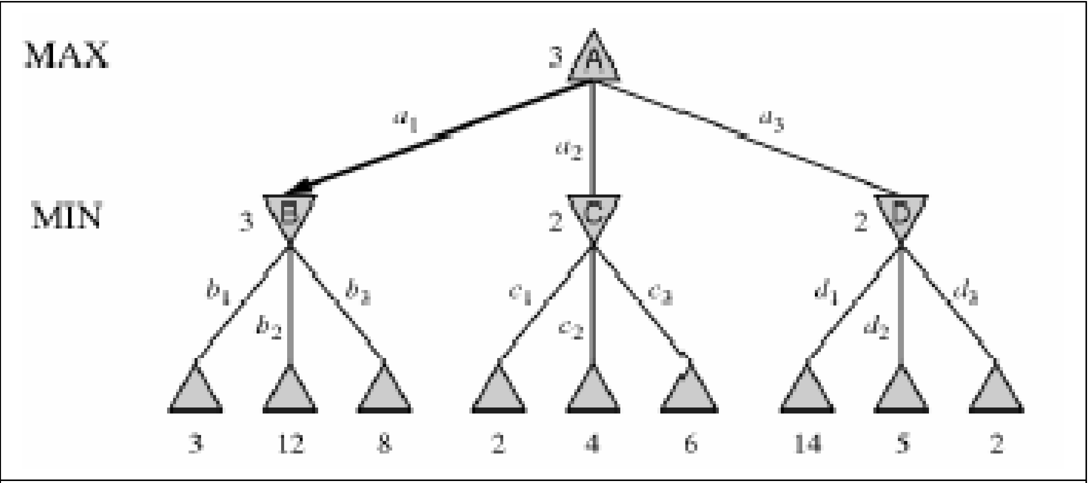


**Figure 2.42 Optimal Decisions in Games**

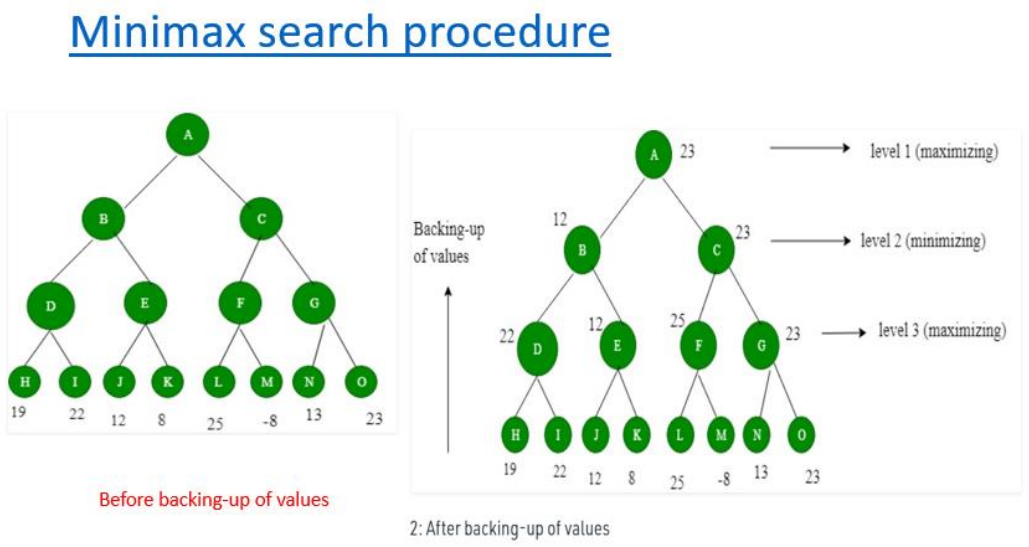
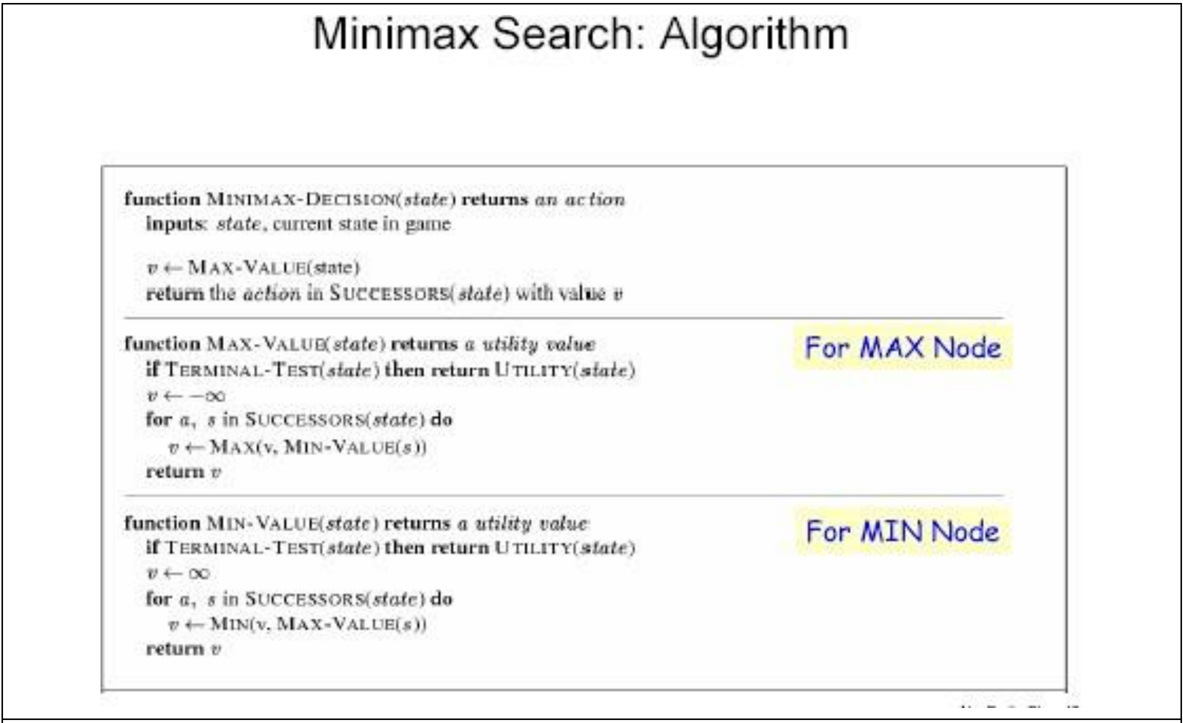


**Figure 2.43 MAX-VALUE and MIN-VALUE**



## Minimax Search: Algorithm

```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game

   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v
```

For MAX Node

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s))
   return v
```

For MIN Node

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s))
   return v
```

**Figure 2.44 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.**

**The minimax Algorithm**

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds**.** For example in Figure 2.19,the algorithm first recourses down to the three bottom left nodes, and uses the utility function on them to discover that their values are 3, 12, and 8 respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 at the root node. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is *m*, and there are b legal moves at each point, then the time complexity of the minimax algorithm is O($b^m$). The space complexity is O(bm) for an algorithm that generates successors at once.

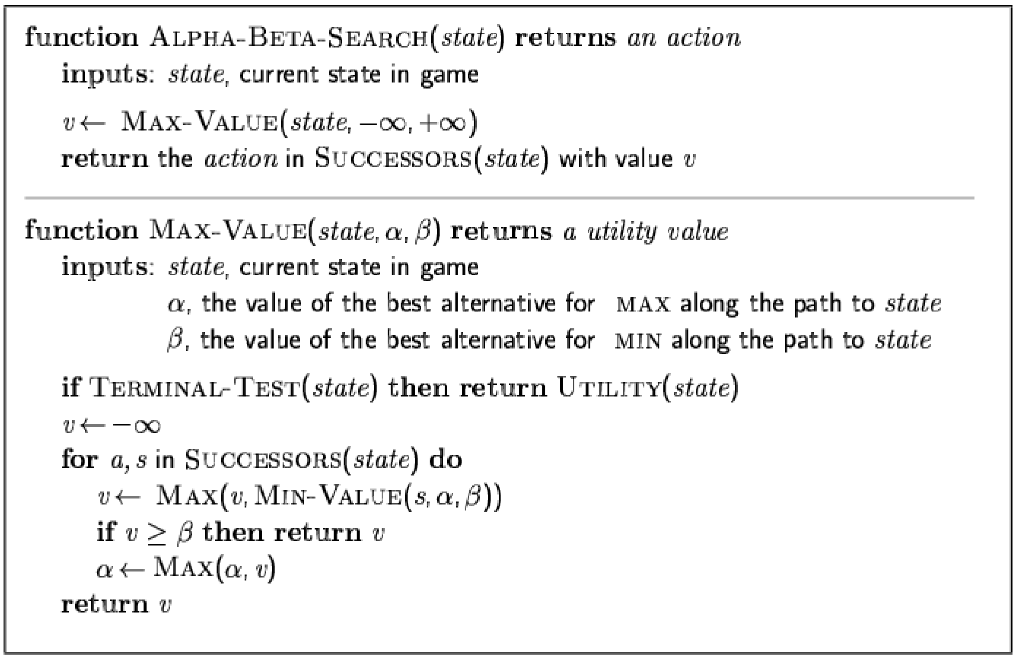
**Alpha-Beta Pruning**

The problem with minimax search is that the number of game states it has to examine is **exponential** in the number of moves. Unfortunately, we can't eliminate the exponent, but we can effectively cut it in half. By performing **pruning,** we can eliminate large part of the tree from consideration. We can apply the technique known as **alpha beta pruning**, when applied to a minimax tree, it returns the same move as **minimax** would, but **prunes away** branches that cannot possibly influence the final decision.

Alpha Beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

- o α : the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path of MAX.
- o β: the value of best (i.e., lowest-value) choice we have found so far at any choice point along the path of MIN.

Alpha Beta search updates the values of α and β as it goes along and prunes the remaining branches at anode(i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α and β value for MAX and MIN, respectively. The complete algorithm is given in Figure. The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. It might be worthwhile to try to examine first the successors that are likely to be the best. In such case, it turns out that alpha-beta needs to examine only O($b^{d/2}$) nodes to pick the best move, instead of O($b^d$) for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess,6 instead of 35. Put an other way alpha-beta cab look ahead roughly twice as far as minimax in the same amount of time.



```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    $v \leftarrow$ MAX-VALUE(state, $-\infty, +\infty$)
    return the action in SUCCESSORS(state) with value $v$

function MAX-VALUE(state, $\alpha, \beta$) returns a utility value
    inputs: state, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to state
            $\beta$, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MAX($v$, MIN-VALUE($s, \alpha, \beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha, v$)
    return $v$
```

```
function MIN-VALUE($state, \alpha, \beta$) returns a utility value
    inputs: state, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to state
            $\beta$, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow +\infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MIN($v$, MAX-VALUE($s, \alpha, \beta$))
        if $v \leq \alpha$ then return $v$
        $\beta \leftarrow$ MIN($\beta, v$)
    return $v$
```

**Figure 2.45 The alpha beta search algorithm. These routines are the same as the minimax routines in figure 2.20,except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β**

**Key points in Alpha-beta Pruning**

- Alpha: Alpha is the best choice or the highest value that we have found at any instance along the path of Maximizer. The initial value for alpha is – ∞.
- Beta: Beta is the best choice or the lowest value that we have found at any instance along the path of Minimizer. The initial value for alpha is + ∞.
- The condition for Alpha-beta Pruning is that α >= β.
- Each node has to keep track of its alpha and beta values. Alpha can be updated only when it's MAX's turn and, similarly, beta can be updated only when it's MIN's chance.
- MAX will update only alpha values and MIN player will update only beta values.
- The node values will be passed to upper nodes instead of values of alpha and beta during go into reverse of tree.
- Alpha and Beta values only be passed to child nodes.

**Working of Alpha-beta Pruning**

1. We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. α = -∞ and β= +∞. We will prune the node only when alpha becomes greater than or equal to beta.

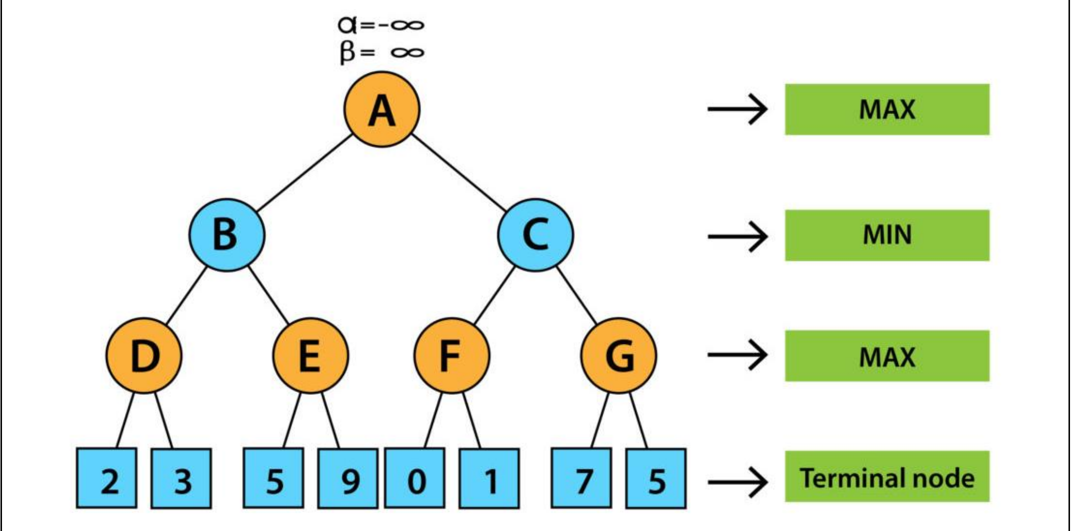


**Figure 2.46 Step 1 Alpha-beta Pruning**

2. Since the initial value of alpha is less than beta so we didn't prune it. Now it's turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.

3. Now the next move will be on node B and its turn for MIN now. So, at node B, the value of alpha beta will be min (3, ∞). So, at node B values will be alpha= – ∞ and beta will be 3.

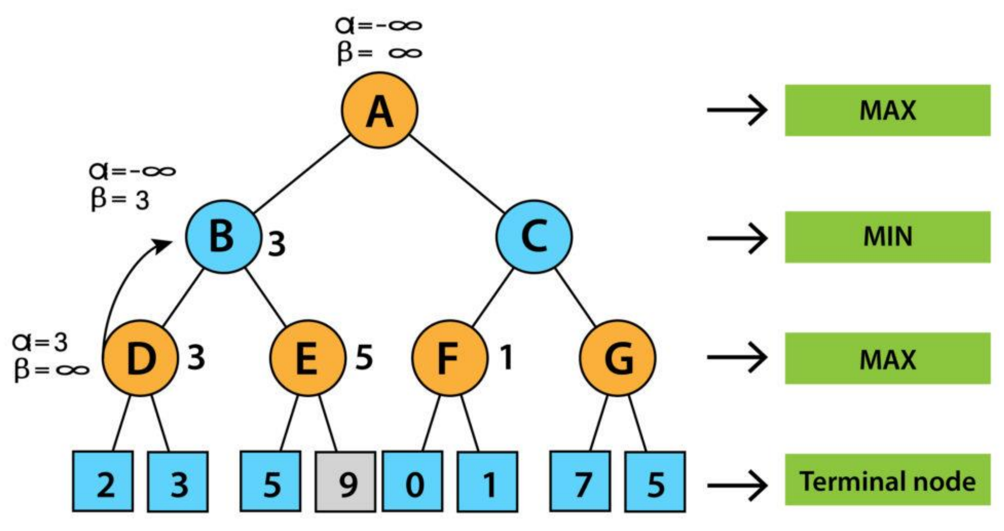


**Figure 2.47 Step 2 Alpha-beta Pruning**

In the next step, algorithms traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

4. Now it's turn for MAX. So, at node E we will look for MAX. The current value of alpha at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5. So, at node E, alpha = 5, Beta = 5. Now as we can see that alpha is greater than beta which is satisfying the pruning condition so we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.

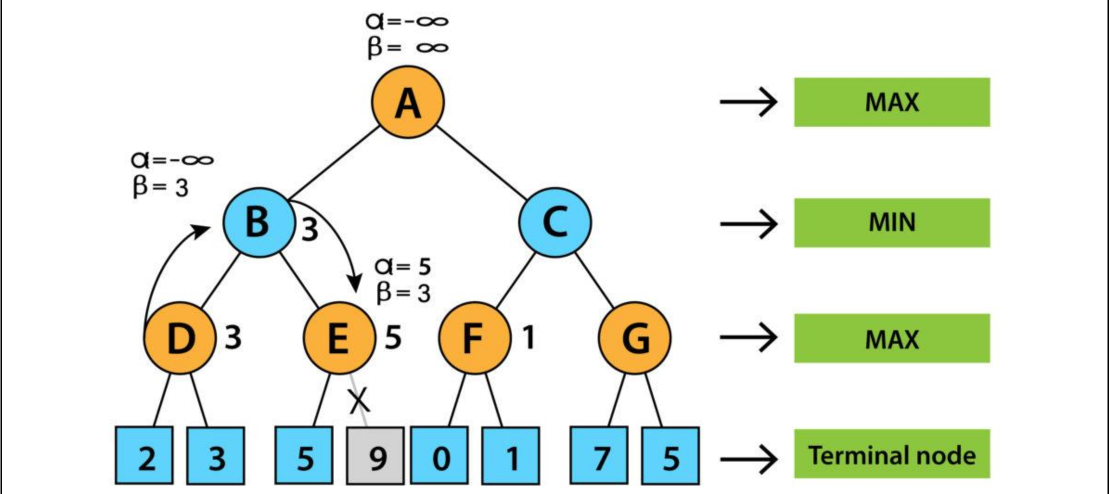


**Figure 2.48 Step 3 Alpha-beta Pruning**

6. In the next step the algorithm again comes to node A from node B. At node A alpha will be changed to maximum value as MAX (- ∞, 3). So now the value of alpha and beta at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.

7. At node F the value of alpha will be compared to the left branch which is 0. So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3 still α remains 3, but the node value of F will become 1.

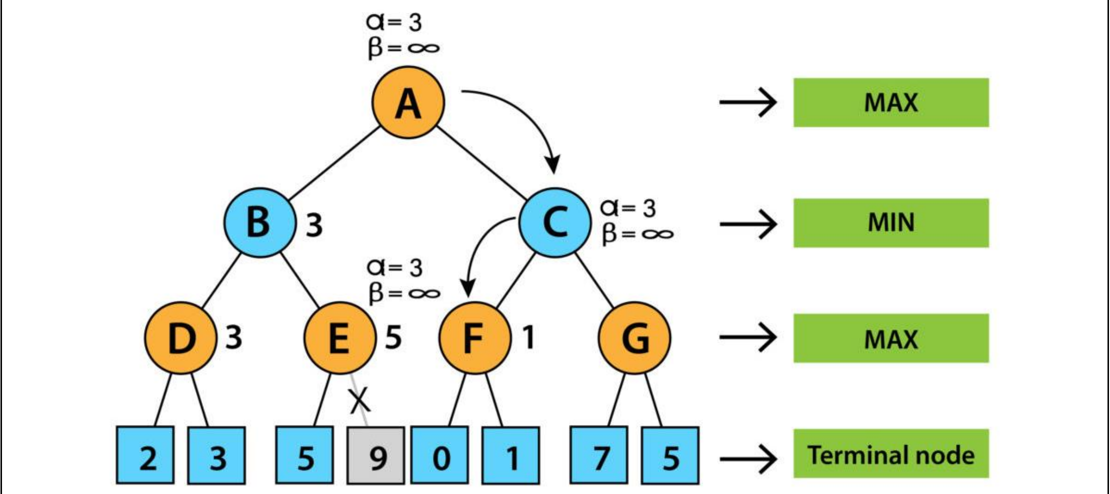


**Figure 2.49 Step 4 Alpha-beta Pruning**

8. Now node F will return the node value 1 to C and will compare to beta value at C. Now its turn for MIN. So, MIN (+ ∞, 1) will be 1. Now at node C, α= 3, and β= 1 and alpha is greater than beta which again satisfies the pruning condition. So, the next successor of node C i.e. G will be pruned and the algorithm didn't compute the entire subtree G.

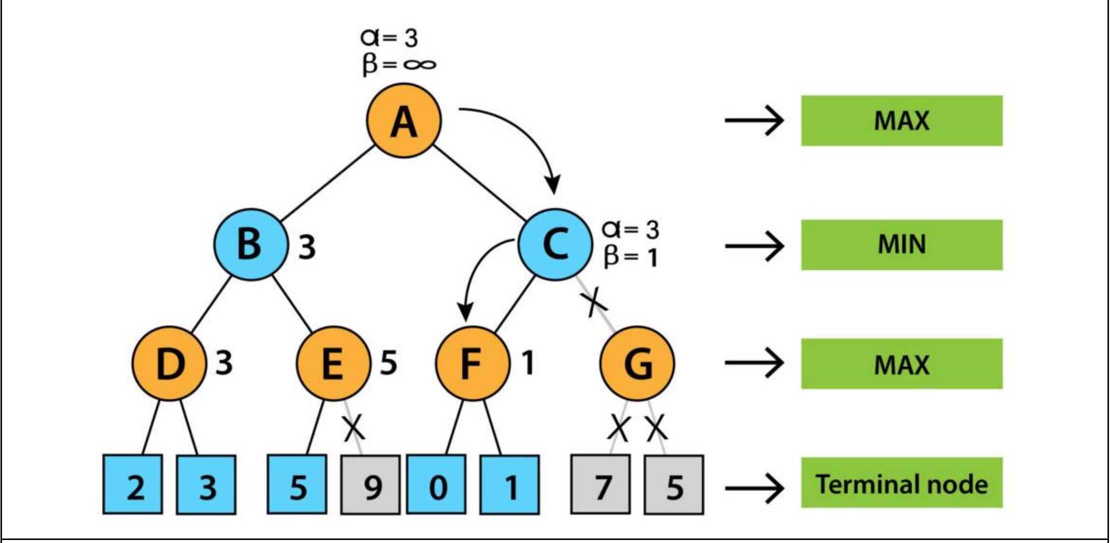


**Figure 2.50 Step 5 Alpha-beta Pruning**

Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3.

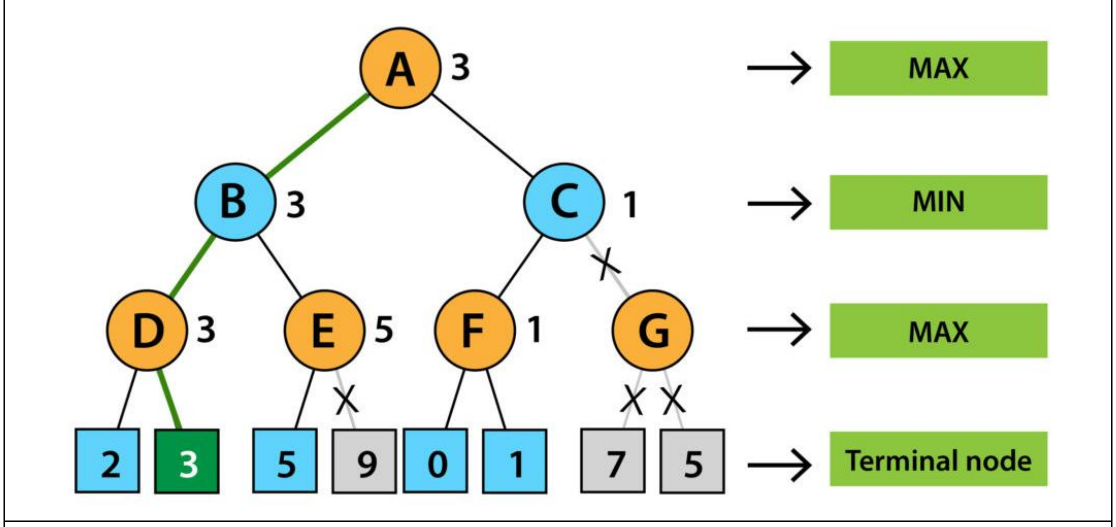


**Figure 2.51 Step 6 Alpha-beta Pruning**

The above represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINERING

# UNIT- III- ARTIFICIAL INTELLIGENCE- SECA3011

# UNIT 3

# KNOWLEDGE REPRESENTATION

First Order Predicate Logic – Prolog Programming – Unification – Forward Chaining-Backward Chaining – Resolution – Knowledge Representation – Ontological Engineering-Categories and Objects – Events – Mental Events and Mental Objects – Reasoning Systems for Categories – Reasoning with Default Information.

## First order Logic

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation.

**First-Order Logic** is a logic which is sufficiently expressive to represent a good deal of our common sense knowledge.

- It is also either includes or forms the foundation of many other representation languages.
- It is also called as **First-Order Predicate calculus.**
- It is abbreviated as **FOL** or **FOPC**

**FOL** adopts the foundation of propositional logic with all its advantages to build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks.

The Syntax of natural language contains elements such as,

1. Nouns and noun phrases that refer to objects (Squares, pits, rumpuses)
2. Verbs and verb phrases that refer to among objects ( is breezy, is adjacent to)

Some of these relations are functions-relations in which there is only one “Value” for a given “input”. Whereas propositional logic assumes the world contains facts, first-order logic (like natural language) assumes the world contains Objects: people, houses, numbers, colors, baseball games, wars, …

Relations: red, round, prime, brother of, bigger than, part of, comes between,…

Functions: father of, best friend, one more than, plus, …

## 3.1 SPECIFY THE SYNTAX OF FIRST-ORDER LOGIC IN BNF FORM

The domain of a model is DOMAIN the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object. Figure 8.2 shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown. The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation TUPLE is just the set of tuples of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

*{ ⟨Richard the Lionheart, King John>, ⟨ King John, Richard the Lionheart⟩ }*

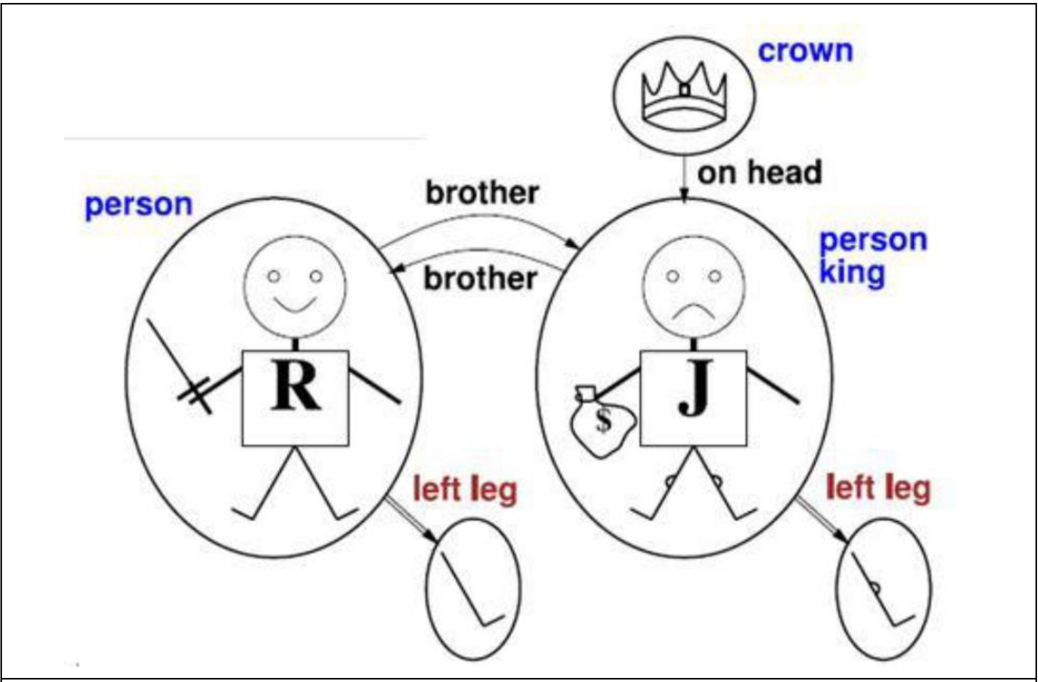


**Figure 3.1 First-Order Logic in BNF Form**

The crown is on King John's head, so the "on head" relation contains just one tuple, _ the crown, King John_. The "brother" and "on head" relations are binary relations — that is, they relate pairs of objects. Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

*⟨Richard the Lionheart⟩ → Richard's left leg*
*⟨King John⟩ → John's left leg*

The five objects are,

- Richard the Lionheart
- His younger brother
- The evil King John
- The left legs of Richard and John
- A crown

- The objects in the model may be related in various ways, In the figure Richard and John are brothers.
- Formally speaking, a relation is just the set of tuples of objects that are related.
- A tuple is a collection of Objects arranged in a fixed order and is written with angle brackets surrounding the objects.
- Thus, the brotherhood relation in this model is the set **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
- The crown is on King John's head, so the "on head" relation contains just one tuple, (the crown, King John).
    - The relation can be binary relation relating pairs of objects (Ex:- "Brother") or unary relation representing a common object (Ex:- "Person" representing both Richard and John)

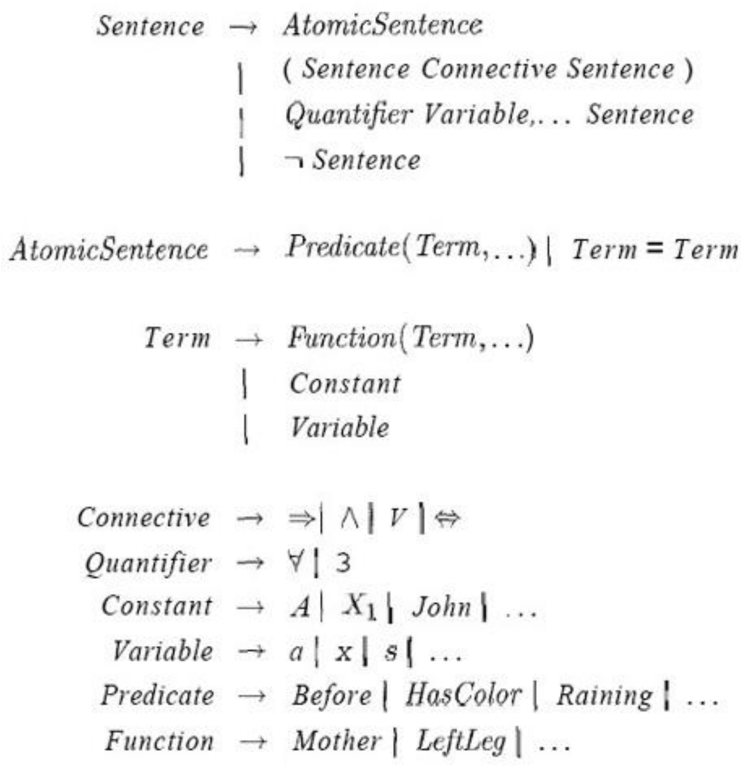
Certain kinds of relationships are best considered as functions that relates an object to exactly one object.

- For Example:- each person has one left leg, so the model has a unary "left leg" function that includes the following mappings (Richard the Lionheart) ----> Richard's left leg (King John) ----> John's left leg
- **Symbols and Interpretations:**
- The basic syntactic elements of first-order logic are the symbols that stand for **objects, relations** and **functions**
- **Kinds of Symbols**
- The symbols come in three kinds namely,
- Constant Symbols standing for **Objects** (Ex:- Richard)
- Predicate Symbols standing for **Relations** (Ex:- King)
- Function Symbols stands for **functions** (Ex:-Left Leg)

o Symbols will begin with uppercase letters
o The choice of names is entirely up to the user
o Each predicate and function symbol comes with an arity
o Arity fixes the number of arguments.

- The semantics must relate sentences to models in order to determine truth.
- To do this, an interpretation is needed specifying exactly which **objects, relations** and **functions** are referred to by the **constant, predicate and function symbols**.
- One possible interpretation called as the intended interpretation- is as follows;
- **Richard** refers to **Richard the Lion heart** and **John** refers to the **evil King John.**
- **Brother** refers to the brotherhood relation, that is the set of tuples of objects given in equation **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
- **On Head** refers to the “on head” relation that holds between the crown and King John; **Person, King** and **Crown** refer to the set of objects that are persons, kings and crowns.
- **Left leg** refers to the “left leg” function, that is, the mapping given in **{(Richard the Lion heart, King John), (King John, Richard the Lionheart)}**

A complete description from the formal grammar is as follows



```
      Sentence → AtomicSentence
               | ( Sentence Connective Sentence )
               | Quantifier Variable,... Sentence
               | ¬ Sentence

AtomicSentence → Predicate(Term,...) | Term = Term

          Term → Function(Term,...)
               | Constant
               | Variable

    Connective → ⇒ | ∧ | V | ⇔
    Quantifier → ∀ | 3
      Constant → A | X1 | John | ...
      Variable → a | x | s | ...
     Predicate → Before | HasColor | Raining | ...
      Function → Mother | LeftLeg | ...
```

Term A term is a logical expression that refers TERM to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression “King John’s left leg” rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use Left Leg (John). The formal semantics of terms is straightforward. Consider a term f(t1, . . . , tn). The function symbol f refers to some function in the model. Atomic sentences Atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as Brother (Richard, John). Atomic sentences can have complex terms as arguments. Thus, Married (Father (Richard),Mother (John)) states that Richard the Lionheart’s father is married to King John’s mother.

**Complex Sentences**

We can use logical connectives to construct more complex sentences, with the same syntax and semantics as in propositional calculus

¬Brother (LeftLeg (Richard), John)

Brother (Richard, John) ∧ Brother (John, Richard)

King(Richard ) ∨ King(John)

¬King(Richard) ⇒ King(John).

**Quantifiers**

Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name. First-order logic contains two standard quantifiers, called universal and existential

**Universal quantification (∀)**

“All kings are persons,” is written in first-order logic as

**∀x King(x) ⇒ Person(x)**

∀ is usually pronounced “For all. . .” Thus, the sentence says, “For all x, if x is a king, then x is a person.” The symbol x is called a variable. A term with no variables is called a **ground term.**

Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

**x → Richard the Lionheart,**

**x → King John, x → Richard’s left leg,**

**x → John’s left leg,**

**x → the crown**.

The universally quantified sentence ∀ x King(x) ⇒ Person(x) is true in the original model if the sentence King(x) ⇒ Person(x) is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

**Richard the Lionheart is a king ⇒ Richard the Lionheart is a person.**

**King John is a king ⇒ King John is a person.**

**Richard's left leg is a king ⇒ Richard's left leg is a person.**

**John's left leg is a king ⇒ John's left leg is a person.**

**The crown is a king ⇒ the crown is a person.**

**Existential quantification (∃)**

Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

***∃x Crown(x) ∧OnHead(x, John)***

∃x is pronounced "There exists an x such that . . ." or "For some x . . ." More precisely, ∃x P is true in a given model if P is true in at least one extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

***Richard the Lionheart is a crown ∧ Richard the Lionheart is on John's head;***

***King John is a crown ∧ King John is on John's head;***

***Richard's left leg is a crown ∧ Richard's left leg is on John's head;***

***John's left leg is a crown ∧ John's left leg is on John's head;***

***The crown is a crown ∧ the crown is on John's head.***

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Just as ⇒ appears to be the natural connective to use with ∀, ∧ is the natural connective to use with ∃.

Using ∧ as the main connective with ∀ led to an overly strong statement in the example in the previous section; using ⇒ with ∃ usually leads to a very weak statement, indeed. Consider the following sentence:

***∃x Crown(x) ⇒OnHead(x, John)***

Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

***Richard the Lionheart is a crown ⇒ Richard the Lionheart is on John's head;***

***King John is a crown ⇒ King John is on John's head;***

***Richard's left leg is a crown ⇒ Richard's left leg is on John's head;***

and so on. Now an implication is true if both premise and conclusion are true, or if its premise is false. So if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true whenever any object fails to satisfy the premise

**Nested quantifiers**

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, “Brothers are siblings” can be written as

***∀x ∀y Brother (x, y) ⇒ Sibling(x, y)***

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

***∀x, y Sibling(x, y) ⇔ Sibling(y, x)***. In other cases we will have mixtures. “Everybody loves somebody” means that for every person, there is someone that person loves:

***∀x∃ y Loves(x, y).***

On the other hand, to say “There is someone who is loved by everyone,” we write

***∃y ∀ x Loves(x, y).***

The order of quantification is therefore very important. It becomes clearer if we insert parentheses.

***∀x (∃ y Loves(x, y))*** says that everyone has a particular property, namely, the property that they love someone. On the other hand,

***∃y (∀ x Loves(x, y))*** says that someone in the world has a particular property, namely the property of being loved by everybody.

***Connections between ∀ and ∃***

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

∀x¬Likes(x, Parsnips ) is equivalent to ¬

**∀x¬Likes(x, Parsnips ) is equivalent to ¬∃ x Likes(x, Parsnips)**. We can go one step further: “Everyone likes ice cream” means that there is no one who does not like ice cream:

**∀x Likes(x, IceCream) is equivalent to ¬∃ x ¬Likes(x, IceCream).**

**Equality**

We can use the equality symbol to signify that two terms refer to the same object. For example,

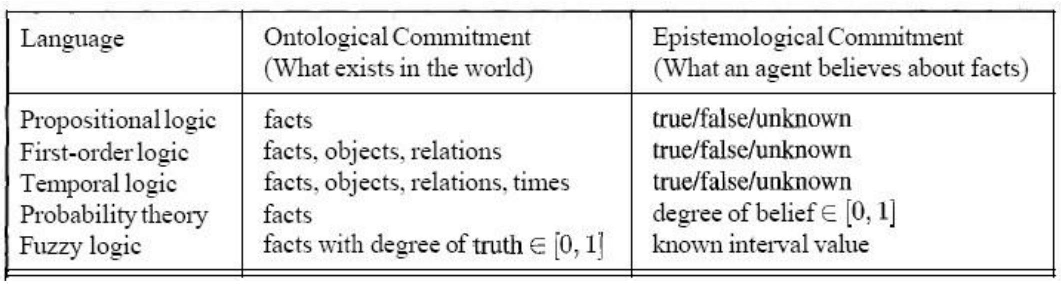
*Father (John)=Henry*

says that the object referred to by Father (John) and the object referred to by Henry are the same.

The equality symbol can be used to state facts about a given function, as we just did for the Father symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

***∃x, y Brother (x,Richard ) ∧ Brother (y,Richard ) ∧¬(x=y).***

**Compare different knowledge representation languages**



| Language | Ontological Commitment (What exists in the world) | Epistemological Commitment (What an agent believes about facts) |
|---|---|---|
| Propositional logic | facts | true/false/unknown |
| First-order logic | facts, objects, relations | true/false/unknown |
| Temporal logic | facts, objects, relations, times | true/false/unknown |
| Probability theory | facts | degree of belief $\in [0, 1]$ |
| Fuzzy logic | facts with degree of truth $\in [0, 1]$ | known interval value |

**Figure 3.2 Formal languages and their ontological and epistemological commitments**

**What are the syntactic elements of First Order Logic**?

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, come in three kinds:

a) constant symbols, which stand for objects;
b) predicate symbols, which stand for relations;
c) and function symbols, which stand for functions.

We adopt the convention that these symbols will begin with uppercase letters. Example:

Constant symbols :
Richard and John;
predicate symbols :
Brother, On Head, Person, King, and Crown; function symbol : LeftLeg.

**Quantifiers**

Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name if a logic that allows object is found.

It has two type,

The following are the types of standard quantifiers,

□ Universal

□ Existential

**Universal quantification**

Explain Universal Quantifiers with an example.

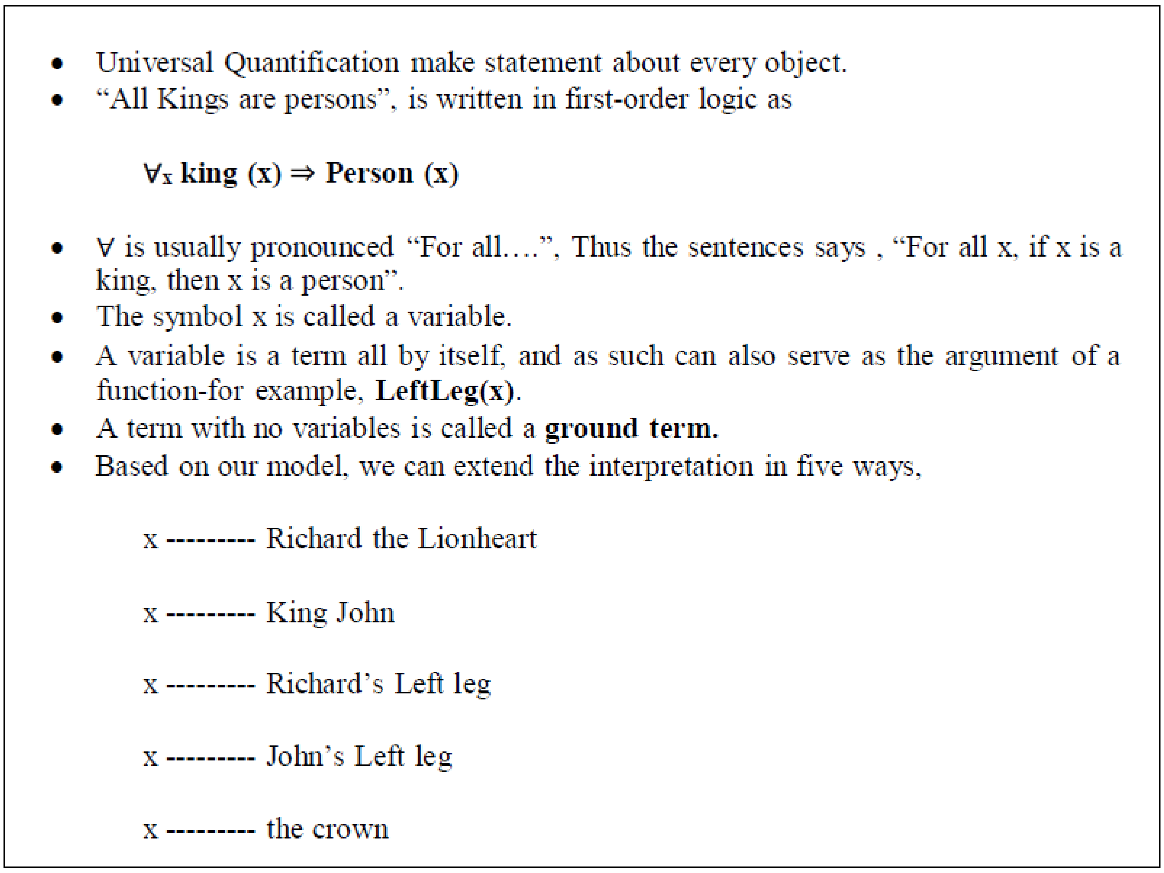
Rules such as "All kings are persons," is written in first-order logic as

□x King(x) => Person(x)

where□ is pronounced as " For all.."

Thus, the sentence says, "For all *x*, if *x* is a king, then is a person." The symbol x is called a variable(lower case letters)

The sentence □**x** P, where P is a logical expression says that P is true for every object x.



- Universal Quantification make statement about every object.
- "All Kings are persons", is written in first-order logic as

  $\forall_x$ **king (x) ⇒ Person (x)**

- ∀ is usually pronounced "For all….", Thus the sentences says , "For all x, if x is a king, then x is a person".
- The symbol x is called a variable.
- A variable is a term all by itself, and as such can also serve as the argument of a function-for example, **LeftLeg(x)**.
- A term with no variables is called a **ground term.**
- Based on our model, we can extend the interpretation in five ways,

  x --------- Richard the Lionheart

  x --------- King John

  x --------- Richard's Left leg

  x --------- John's Left leg

  x --------- the crown

The universally quantified sentence is equivalent to asserting the following five sentences

Richard the Lionheart ------ Richard the Lionheart is a person
King John is a King ------ King John is a Person
Richard’s left leg is King -------- Richard’s left leg is a person
John’s left leg is a King -------- John’s left leg is a person
The crown is a King -------- The crown is a Person

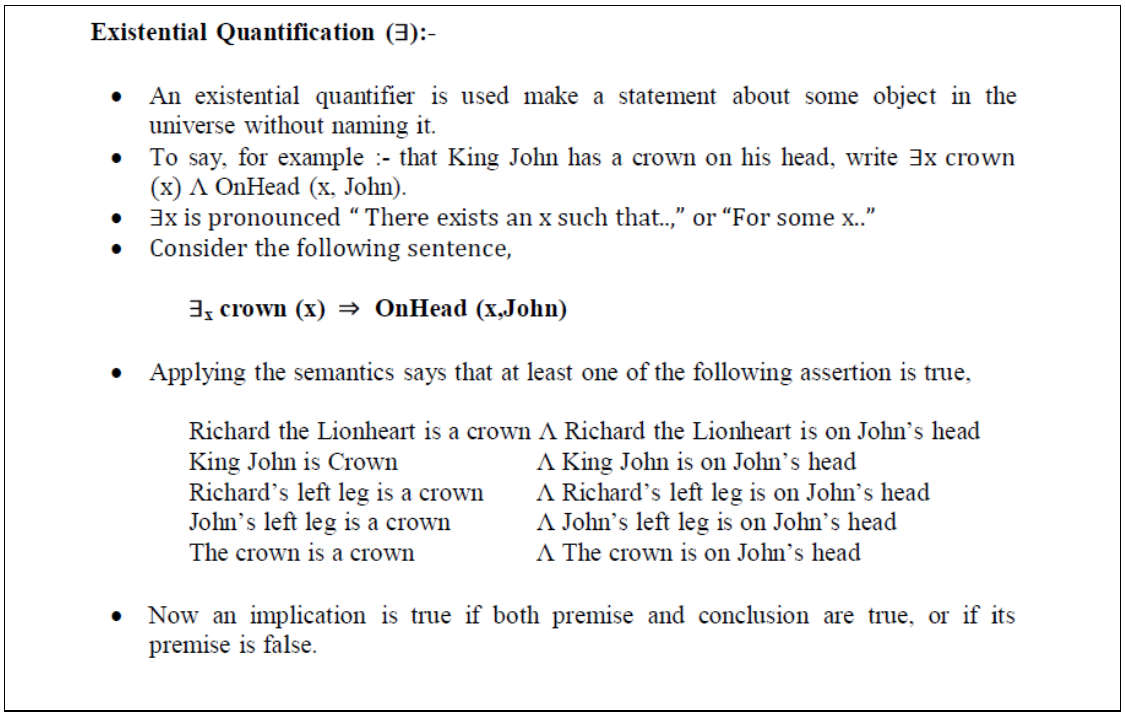
Existential quantification

Universal quantification makes statements about every object.

It is possible to make a statement about some object in the universe without naming it,by using an existential quantifier.

Example

“King John has a crown on his head”
□x Crown(x) ^ OnHead(x,John)
□x is pronounced “There exists an x such that..” or “ For some x ..”



**Existential Quantification (∃):-**

- An existential quantifier is used make a statement about some object in the universe without naming it.
- To say, for example :- that King John has a crown on his head, write ∃x crown (x) Λ OnHead (x, John).
- ∃x is pronounced “ There exists an x such that..,” or “For some x..”
- Consider the following sentence,

  $\exists_x$ **crown (x) ⇒ OnHead (x,John)**

- Applying the semantics says that at least one of the following assertion is true,

  Richard the Lionheart is a crown Λ Richard the Lionheart is on John’s head
  King John is Crown Λ King John is on John’s head
  Richard’s left leg is a crown Λ Richard’s left leg is on John’s head
  John’s left leg is a crown Λ John’s left leg is on John’s head
  The crown is a crown Λ The crown is on John’s head

- Now an implication is true if both premise and conclusion are true, or if its premise is false.

**Nested Quantifiers**

More complex sentences are expressed using multiple quantifiers.
The following are the some cases of multiple quantifiers,
The simplest case where the quantifiers are of the same type.

For Example:- "Brothers are Siblings" can be written as

$$\forall_x \forall_{y,} \text{Brother } (x,y) \Rightarrow \text{sibling } (x,y)$$

Consecutive quantifiers of the same type can be written as one quantifier with

several variables. For Example:- to say that siblinghood is a symmetric relationship as

$$\forall_{x\ ,y} \text{ sibling } (x,y) \Leftrightarrow \text{sibling } (y,z)$$

## 3.2 THE WUMPUS WORLD

The wumpus world is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence. A sample wumpus world is shown in Figure

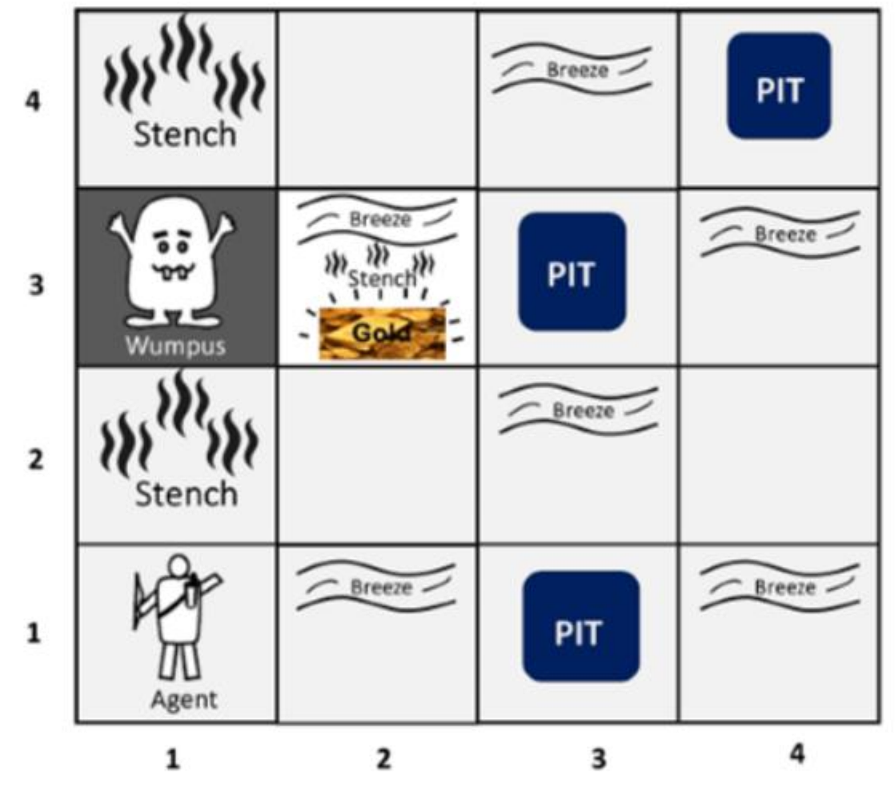


**Figure 3.3 wumpus world**

To specify the agent's task, we specify its percepts, actions, and goals. In the wumpus world, these are as follows:

- In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
- In the squares directly adjacent to a pit, the agent will perceive a breeze.
- In the square where the gold is, the agent will perceive a glitter.
- When an agent walks into a wall, it will perceive a bump.
- When the wumpus is killed, it gives out a woeful scream that can be perceived anywhere in the cave.
- The percepts will be given to the agent in the form of a list of five symbols; for example, if there is a stench, a breeze, and a glitter but no bump and no scream, the agent will receive the percept [Stench, Breeze, Glitter, None, None]. The agent cannot perceive its own location.
- Just as in the vacuum world, there are actions to go forward, turn right by 90°, and turn left by 90°. In addition, the action Grab can be used to pick up an object that is in the same square as the agent. The action Shoot can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits and kills the wumpus or hits the wall. The agent only has one arrow, so only the first Shoot action has any effect.

The wumpus agent receives a percept vector with five elements. The corresponding first order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be

***Percept ([Stench, Breeze, Glitter, None, None], 5)***.

Here, Percept is a binary predicate, and Stench and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

***Turn(Right ), Turn(Left ), Forward, Shoot, Grab, Climb.***

To determine which is best, the agent program executes the query

***ASKVARS(∃ a BestAction(a, 5))***,

which returns a binding list such as {a/Grab}. The agent program can then return Grab as the action to take. The raw percept data implies certain facts about the current state.

For example:

$\forall t,s,g,m,c$ ***Percept ([s,Breeze,g,m,c],t) ⇒ Breeze(t),***

$\forall t,s,b,m,c$ Percept ***([s,b,Glitter,m,c],t) ⇒ Glitter (t)***

These rules exhibit a trivial form of the reasoning process called perception. Simple "reflex" behavior can also be implemented by quantified implication sentences.

For example, we have ***∀t Glitter (t) ⇒BestAction(Grab, t)***.

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion

BestAction***(Grab, 5)***—that is, Grab is the right thing to do. For example, if the agent is at a square and perceives a breeze, then that square is breezy:

***∀s, t At(Agent, s, t) ∧ Breeze(t) ⇒ Breezy(s).***

It is useful to know that a square is breezy because we know that the pits cannot move about. Notice that Breezy has no time argument. Having discovered which places are breezy (or smelly) and, very important, not breezy (or not smelly), the agent can deduce where the pits are (and where the wumpus is). first-order logic just needs one axiom:

***∀s Breezy(s) ⇔∃r Adjacent (r, s) ∧ Pit(r).***

## 3.3 SUBSTITUTION

Let us begin with universal quantifiers

***∀x King(x) ∧ Greedy(x) ⇒ Evil(x)***.

Then it seems quite permissible to infer any of the following sentences:

***King(John) ∧ Greedy(John) ⇒ Evil(John)***
***King(Richard ) ∧ Greedy(Richard) ⇒ Evil(Richard)***
***King(Father (John)) ∧ Greedy(Father (John)) ⇒ Evil(Father (John)).***

The rule of **Universal Instantiation (UI for short)** says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.

Let ***SUBST(θ,α)*** denote the result of applying the substitution θ to the sentence α. Then the rule is written

***∀v α SUBST({v/g}, α)*** for any variable v and ground term g.
For example, the three sentences given earlier are obtained with the substitutions
***{x/John}, {x/Richard },*** and ***{x/Father (John)}.***

In the rule for Existential Instantiation, the variable is replaced by a single new constant symbol. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

***∃v α SUBST({v/k}, α)*** For example, from the sentence

***∃x Crown(x) ∧OnHead(x, John)*** we can infer the sentence

***Crown(C1) ∧OnHead(C1, John)***

**EXAMPLE**

Suppose our knowledge base contains just the sentences

***∀x King(x) ∧ Greedy(x) ⇒ Evil(x)***

***King(John)***

***Greedy(John) Brother (Richard, John)***

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case,

***{x/John} and {x/Richard }.*** We obtain

***King(John) ∧ Greedy(John) ⇒ Evil(John)***

***King(Richard ) ∧ Greedy(Richard) ⇒ Evil(Richard)***,

and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences

***King(John),***

***Greedy(John),*** and so on—as proposition symbols.

## 3.4 UNIFICATION

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called unification and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a unifier for them if one exists: ***UNIFY(p, q)=θ*** where ***SUBST(θ, p)= SUBST(θ, q)***.

Suppose we have a query ***AskVars(Knows(John, x)):*** whom does John know? Answers can be found by finding all sentences in the knowledge base that unify with ***Knows(John, x).***

Here are the results of unification with four different sentences that might be in the knowledge base: ***UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}***

***UNIFY(Knows(John, x), Knows(y, Bill )) = {x/Bill, y/John}***

***UNIFY(Knows(John, x), Knows(y, Mother (y))) = {y/John, x/Mother (John)}***
***UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail.***

The last unification fails because x cannot take on the values John and Elizabeth at the same time. Now, remember that ***Knows(x, Elizabeth)*** means "Everyone knows Elizabeth," so we should be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by standardizing apart one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in

***Knows(x, Elizabeth) to x17*** (a new variable name) without changing its meaning.

**Now the unification will work**

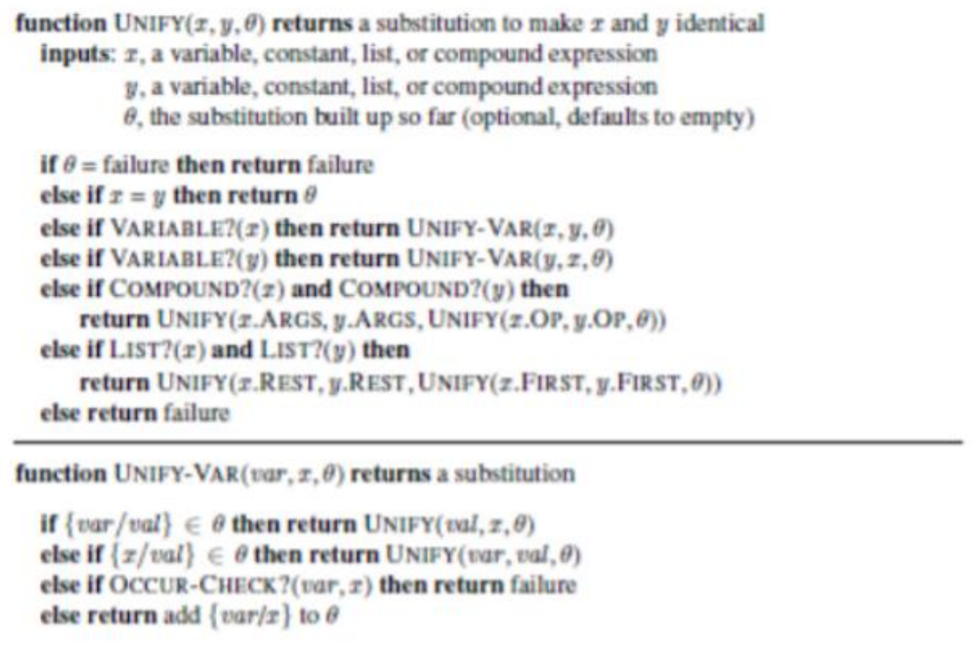
***UNIFY(Knows(John, x), Knows(x17, Elizabeth)) = {x/Elizabeth, x17/John} UNIFY*** should return a substitution that makes the two arguments look the same. But there could be more than one such unifier.

For example,

***UNIFY(Knows(John, x), Knows(y, z))*** could return
***{y/John, x/z} or {y/John, x/John, z/John}.***

The first unifier gives ***Knows(John, z)*** as the result of unification, whereas the second gives ***Knows(John, John).*** The second result could be obtained from the first by an additional substitution ***{z/John};*** we say that the first unifier is more general than the second, because it places fewer restrictions on the values of the variables. An algorithm for computing most general unifiers is shown in Figure.

The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed.



```
function UNIFY($x$, $y$, $\theta$) returns a substitution to make $x$ and $y$ identical
  inputs: $x$, a variable, constant, list, or compound expression
          $y$, a variable, constant, list, or compound expression
          $\theta$, the substitution built up so far (optional, defaults to empty)

  if $\theta$ = failure then return failure
  else if $x = y$ then return $\theta$
  else if VARIABLE?($x$) then return UNIFY-VAR($x$, $y$, $\theta$)
  else if VARIABLE?($y$) then return UNIFY-VAR($y$, $x$, $\theta$)
  else if COMPOUND?($x$) and COMPOUND?($y$) then
      return UNIFY($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
  else if LIST?($x$) and LIST?($y$) then
      return UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
  else return failure

function UNIFY-VAR($var$, $x$, $\theta$) returns a substitution

  if $\{var/val\} \in \theta$ then return UNIFY($val$, $x$, $\theta$)
  else if $\{x/val\} \in \theta$ then return UNIFY($var$, $val$, $\theta$)
  else if OCCUR-CHECK?($var$, $x$) then return failure
  else return add $\{var/x\}$ to $\theta$
```

**Figure 3.4 Recursively explore**

In artificial intelligence, forward and backward chaining is one of the important topics, but before understanding forward and backward chaining lets first understand that from where these two terms came.

**Inference engine**

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:

a. Forward chaining
b. Backward chaining

**Horn Clause and Definite clause**

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the first-order definite clause.nt

Definite clause: A clause which is a disjunction of literals with exactly one positive literal is known as a definite clause or strict horn clause.

Horn clause: A clause which is a disjunction of literals with at most one positive literal is known as horn clause. Hence all the definite clauses are horn clauses.

Example: $(\neg p \vee \neg q \vee k)$. It has only one positive literal k.
It is equivalent to $p \wedge q \rightarrow k$.

**A. Forward Chaining**

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 (Modus Ponens) 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.

Consider the following famous example which we will use in both approaches:

**Example**

"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, we will convert all the above facts into first-order definite clauses, and then we will 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.

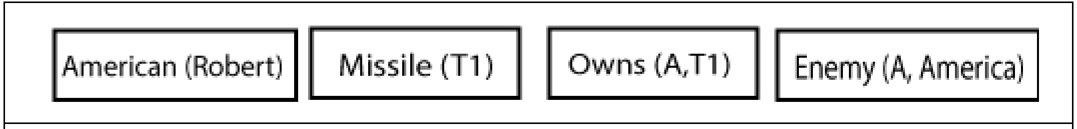


**Figure 3.5**

**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-(4) satisfy with the substitution {p/T1}, so Sells (Robert, T1, A) is added, which infers from the conjunction of Rule (2) and (3).

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

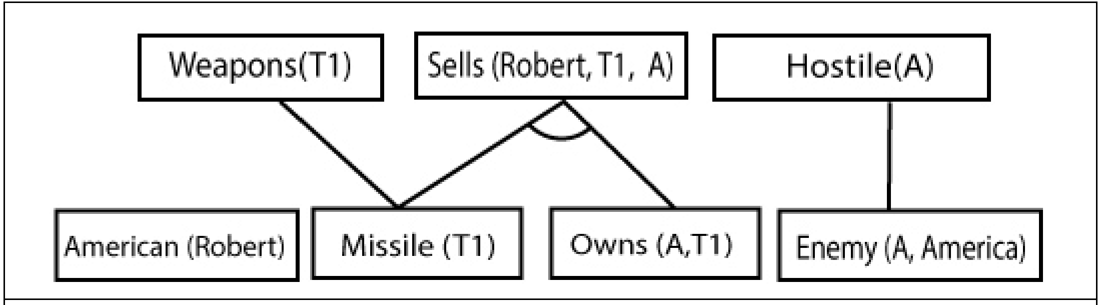


**Figure 3.6**

**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 we reached our goal statement.

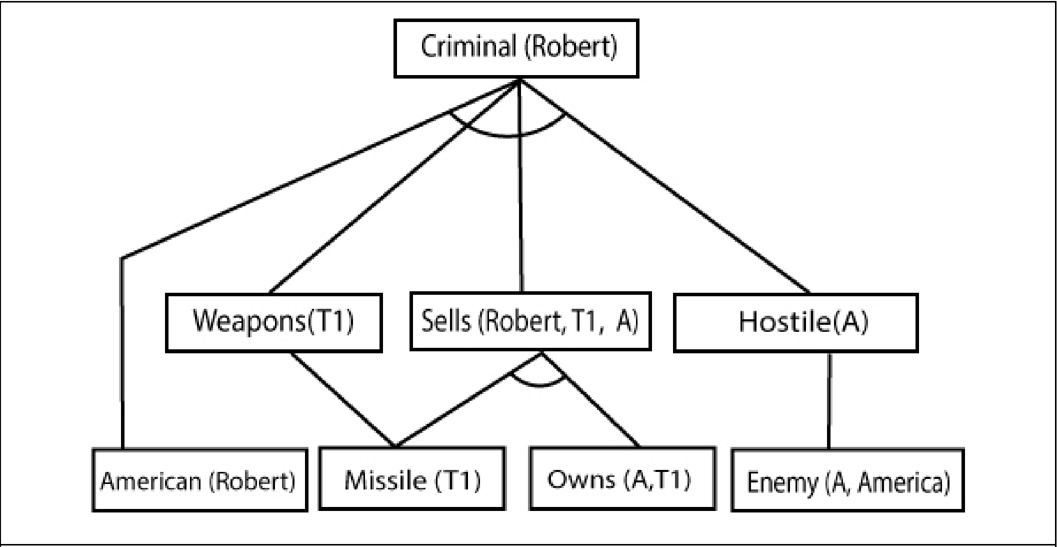


**Figure 3.7**

Hence it is proved that Robert is Criminal using forward chaining approach.

## B. Backward Chaining

Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

**Properties of backward chaining**

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a depth-first search strategy for proof.

**Example**

In backward-chaining, we will use the same above example, and will rewrite all the rules.

- American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) …(1)
  Owns(A, T1) …(2)

- Missile(T1)
- ?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) …(4)
- Missile(p) → Weapons (p) …(5)
- Enemy(p, America) →Hostile(p) …(6)
- Enemy (A, America) …(7)
- American (Robert). …(8)

**Backward-Chaining proof**

In Backward chaining, we will start with our goal predicate, which is Criminal (Robert), and then infer further rules.

**Step-1**

At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.

**Step-2**

At the second step, we will infer other facts form goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (Robert) is present with substitution {Robert/P}. So we will add all the conjunctive facts below the first level and will replace p with Robert.

Here we can see American (Robert) is a fact, so it is proved here.

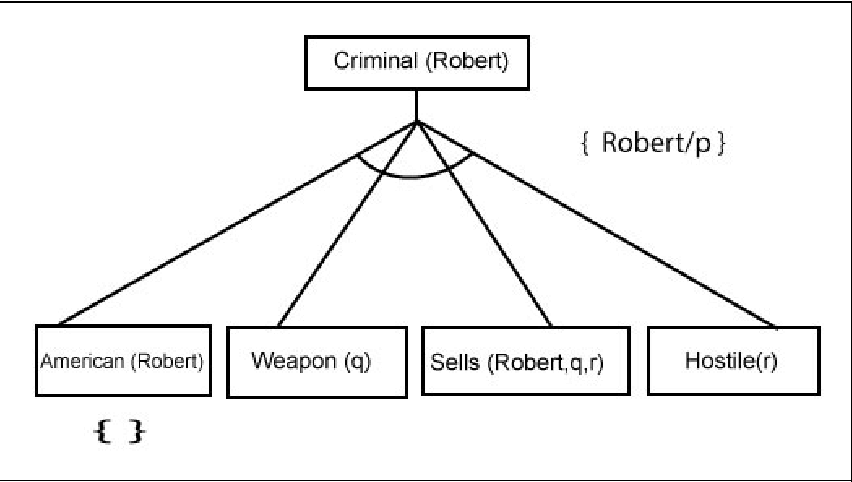


**Figure 3.8**

**Step-3**

t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.

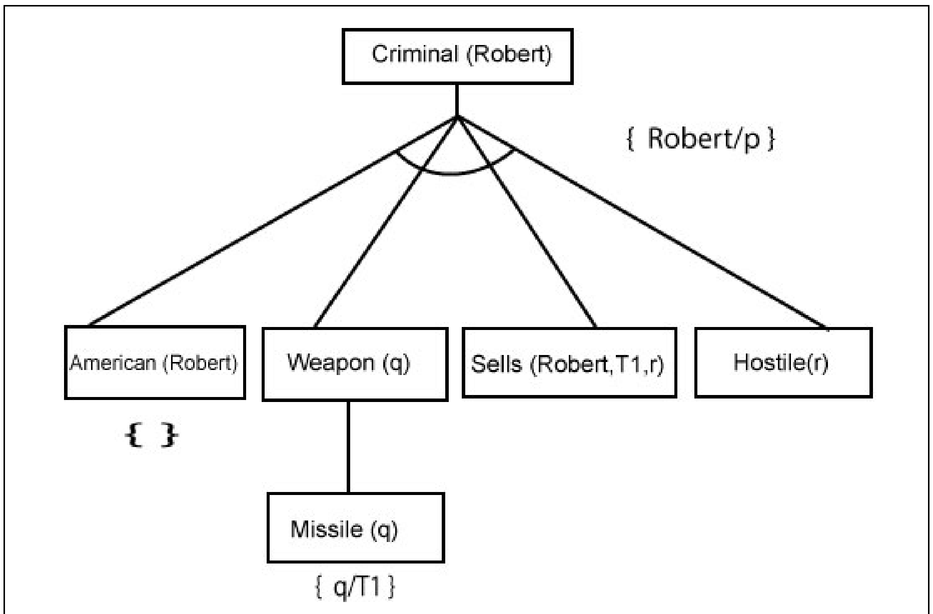


**Figure 3.9**

**Step-4**

At step-4, we can infer facts Missile(T1) and Owns(A, T1) form Sells(Robert, T1, r) which satisfies the Rule- 4, with the substitution of A in place of r. So these two statements are proved here.

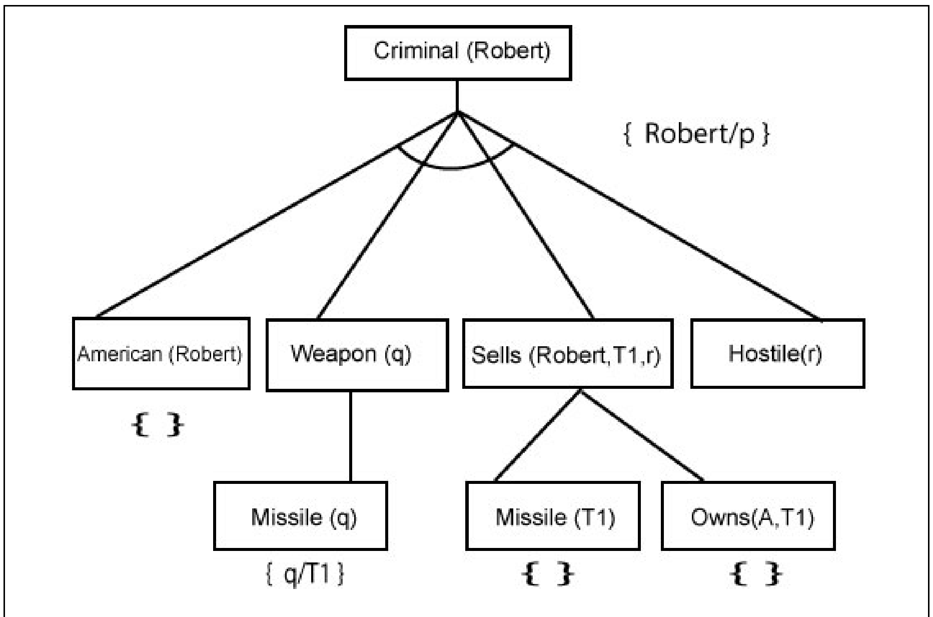


**Figure 3.10**

**Step-5**

At step-5, we can infer the fact Enemy(A, America) from Hostile(A) which satisfies Rule- 6. And hence all the statements are proved true using backward chaining.

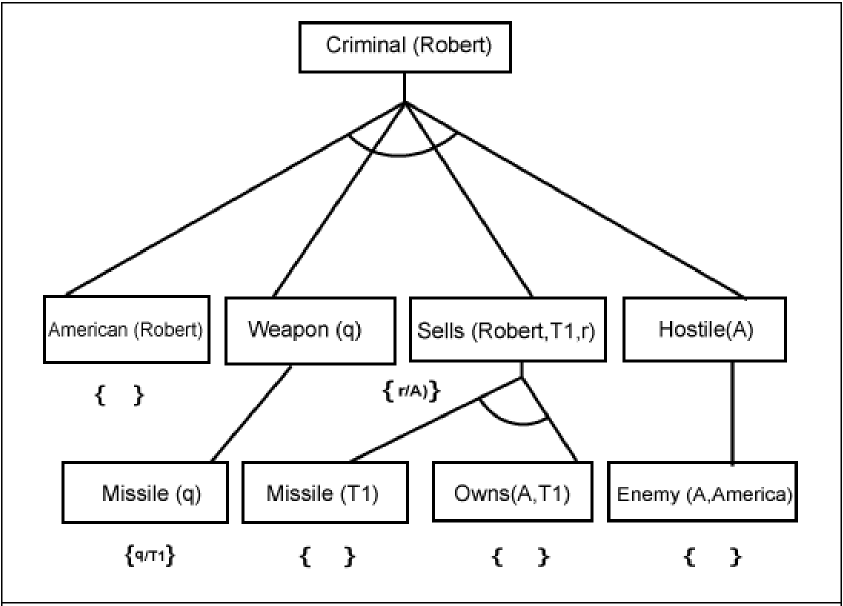


**Figure 3.11**

Suppose you have a production system with the FOUR rules: R1: IF A AND C then F R2: IF A AND E, THEN G R3: IF B, THEN E R4: R3: IF G, THEN D and you have four initial facts: A, B, C, D. PROVE A&B TRUE THEN D IS TRUE. Explain what is meant by “forward chaining”, and show explicitly how it can be used in this case to determine new facts.

## 3.5 RESOLUTION IN FOL

### Resolution

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the conjunctive normal form or clausal form.

Clause: Disjunction of literals (an atomic sentence) is called a clause. It is also known as a unit clause.

Conjunctive Normal Form: A sentence represented as a conjunction of clauses is said to be conjunctive normal form or CNF.

**Steps for Resolution**

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

To better understand all the above steps, we will take an example in which we will apply resolution.

**Example**

a. John likes all kind of food.
b. Apple and vegetable are food
c. Anything anyone eats and not killed is food.
d. Anil eats peanuts and still alive
e. Harry eats everything that Anil eats. Prove by resolution that:
f. John likes peanuts.

Step-1: Conversion of Facts into FOL

In the first step we will convert all the given statements into its first order logic.

a. ∀x: food(x) → likes(John, x)
b. food(Apple) Λ food(vegetables)
c. ∀x ∀y: eats(x, y) Λ ¬ killed(x) → food(y)
d. eats (Anil, Peanuts) Λ alive(Anil).
e. ∀x : eats(Anil, x) → eats(Harry, x)
f. ∀x: ¬ killed(x) → alive(x) } **added predicates.**
g. ∀x: alive(x) →¬ killed(x)
h. likes(John, Peanuts)

- **Eliminate all implication (→) and rewrite**

1. ∀x ¬ food(x) V likes(John, x)
2. food(Apple) Λ food(vegetables)
3. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
4. eats (Anil, Peanuts) Λ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
6. ∀x¬ [¬ killed(x) ] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(John, Peanuts).

- **Move negation (¬)inwards and rewrite**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  6. ∀x ¬killed(x) ] V alive(x)
  7. ∀x ¬ alive(x) V ¬ killed(x)
  8. likes(John, Peanuts).
- **Rename variables or standardize variables**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀w¬ eats(Anil, w) V eats(Harry, w)
  6. ∀g ¬killed(g) ] V alive(g)
  7. ∀k ¬ alive(k) V ¬ killed(k)
  8. likes(John, Peanuts).
- **Eliminate existential instantiation quantifier by elimination.** In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.
- **Drop Universal quantifiers.** In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.

  1. ¬ food(x) V likes(John, x)
  2. food(Apple)
  3. food(vegetables)
  4. ¬ eats(y, z) V killed(y) V food(z)
  5. eats (Anil, Peanuts)
  6. alive(Anil)
  7. ¬ eats(Anil, w) V eats(Harry, w)
  8. killed(g) V alive(g)
  9. ¬ alive(k) V ¬ killed(k)
  10. likes(John, Peanuts).
- **Distribute conjunction ∧ over disjunction ¬.** This step will not make any change in this problem.

**Step-3: Negate the statement to be proved**

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, Peanuts)

**Step-4: Draw Resolution graph**

- Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:

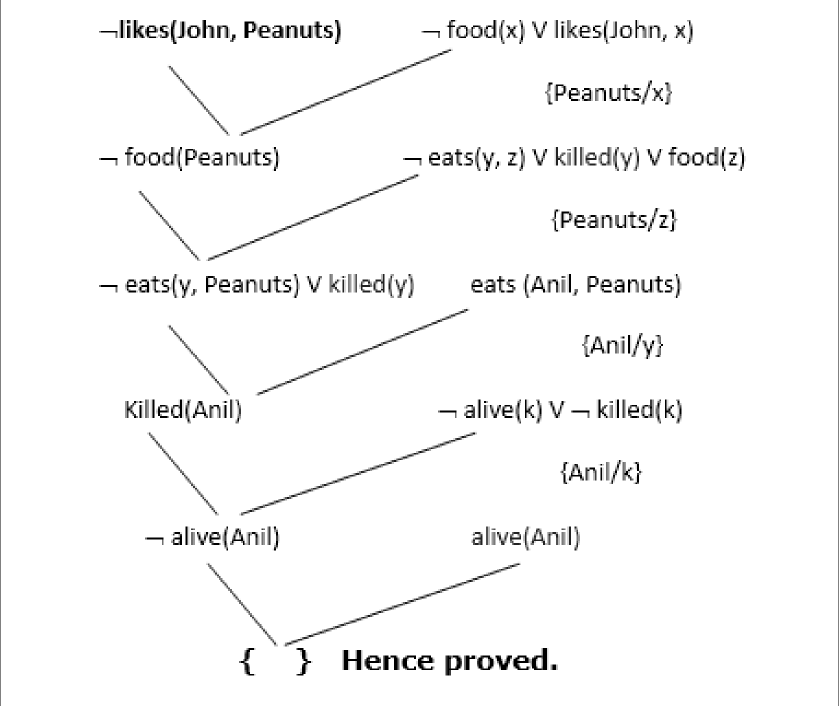


**Figure 3.12**

Explanation of Resolution graph

- In the first step of resolution graph, ¬likes(John, Peanuts), and likes(John, x) get resolved (canceled) by substitution of {Peanuts/x}, and we are left with ¬ food(Peanuts)
- In the second step of the resolution graph, ¬ food (Peanuts), and food(z) get resolved (canceled) by substitution of { Peanuts/z}, and we are left with ¬ eats(y, Peanuts) V killed(y).
- In the third step of the resolution graph, ¬ eats(y, Peanuts) and eats (Anil, Peanuts) get resolved by substitution {Anil/y}, and we are left with Killed(Anil).
- In the fourth step of the resolution graph, Killed(Anil) and ¬ killed(k) get resolve by substitution {Anil/k}, and we are left with ¬ alive(Anil).

o In the last step of the resolution graph ¬ alive(Anil) and alive(Anil) get resolved.

Difference between Predicate Logic and Propositional Logic

**Table 3.1**

| S. No. | Predicate logic | Propositional logic |
|---|---|---|
| 1. | Predicate logic is a generalization of propositional logic that allows us to express and infer arguments in infinite models. | A preposition is a declarative statement that's either TRUE or FALSE (but not both). |
| 2. | Predicate logic (also called predicate calculus and first-order logic) is an extension of propositional logic to formulas involving terms and predicates. The full predicate logic is undecidable | Propositional logic is an axiomatization of Boolean logic. Propositional logic is decidable, for example by the method of truth table. |
| 3. | Predicate logic have variables | Propositional logic has variables. Parameters are all constant. |
| 4. | A predicate is a logical statement that depends on one or more variables (not necessarily Boolean variables) | Propositional logic deals solely with propositions and logical connectives. |
| 5. | Predicate logic there are objects, properties, functions (relations) are involved. | Proposition logic is represented in terms of Boolean variables and logical connectives. |
| 6. | In predicate logic, we symbolize subject and predicate separately. Logicians often use lowercase letters to symbolize subjects (or objects) and upper case letter to symbolize predicates. | In propositional logic, we use letters to symbolize entire propositions. Propositions are statements of the form "x is y" where x is a subject and y is a predicate. |
| 7. | Predicate logic uses quantifies such as universal quantifier ("∀"), the existential quantifier ("∃"). | Prepositional logic has not qualifiers. |
| 8. | **Example** Everything is a green as "∀x Green(x)" or "Something is blue as "∃ x Blue(x)". | **Example** Everything is a green as "G" or "Something is blue as "B(x)". |

## 3.6 KNOWLEDGE REPRESENTATION ONTOLOGICAL ENGINEERING

Concepts such as Events, Time, Physical Objects, and Beliefs— that occur in many different domains. Representing these abstract concepts is sometimes called ontological engineering.

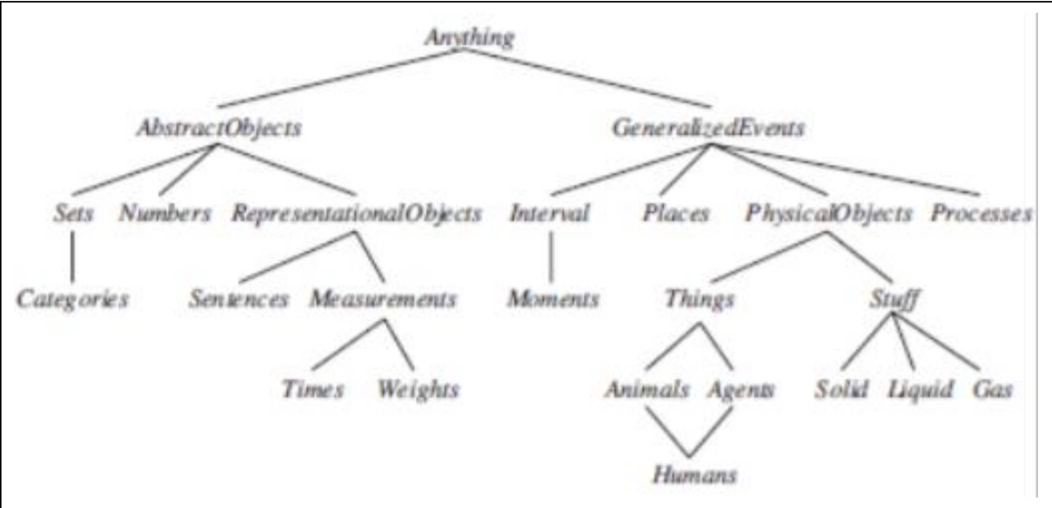


**Figure 3.13 The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example.**

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

**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 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 reify1 the category as an object, Basketballs.

We could then say ***Member(b, Basketballs*** ), which we will abbreviate as ***b∈ Basketballs***, to say that b is a member of the category of basketballs. We say ***Subset(Basketballs, Balls),*** abbreviated as ***Basketballs ⊂ Balls***, to say that Basketballs is a subcategory of Balls. Categories serve to organize and simplify the knowledge base through inheritance. If we say that all instances of the category Food are edible, and if we assert that Fruit is a subclass of Food and ***Apples*** is a subclass of Fruit, then we can infer that every apple is edible. We say that the individual apples inherit the property of edibility, in this case from their membership in the Food category. 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 types of facts, with examples of each:

- An object is a member of a category.

***BB9 ∈ Basketballs***

- A category is a subclass of another category. ***Basketballs ⊂ Balls***
- All members of a category have some properties.

***(x∈ Basketballs) ⇒ 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 ∈ Domesticated Species**

Notice that because Dogs is a category and is a member of Domesticated Species, the latter must be a category of categories. Categories can also be defined by providing necessary and sufficient conditions for membership. For example, a bachelor is an unmarried adult male:

***x∈ Bachelors ⇔ Unmarried(x) ∧ x∈ Adults ∧ x∈ Males***

**Physical Composition**

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

***PartOf (Bucharest, Romania)***
***PartOf (Romania, EasternEurope)***
***PartOf(EasternEurope, Europe)***
***PartOf (Europe, Earth)***
***The PartOf relation is transitive and reflexive; that is,***
***PartOf (x, y) ∧PartOf (y, z) ⇒PartOf (x, z)***
***PartOf (x, x)***
Therefore, we can conclude PartOf (Bucharest, Earth).
For example, if the apples are Apple1, Apple2, and Apple3, then
***BunchOf ({Apple1,Apple2,Apple3})***
denotes the composite object with the three apples as parts (not elements). We can define ***BunchOf in terms of the PartOf relation. Obviously, each element of s is part of***

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

***∀y [∀x x∈ s ⇒PartOf (x, y)] ⇒PartOf (BunchOf (s), y)***

**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. ***Length(L1)=Inches(1.5)=Centimeters(3.81)***

Conversion between units is done by equating multiples of one unit to another: ***Centimeters(2.54 ×d)=Inches(d)***

Similar axioms can be written for pounds and kilograms, seconds and days, and dollars and cents. Measures can be used to describe objects as follows:

***Diameter (Basketball12)=Inches(9.5)***

***ListPrice(Basketball12)=$(19)***

***d∈ Days ⇒ Duration(d)=Hours(24)***

**Time Intervals**

Event calculus opens us up to the possibility of talking about time, 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∈Moments⇔Duration(i)=Seconds(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.

***Interval (i) ⇒Duration(i)=(Time(End(i)) − Time(Begin(i)))***
***Time(Begin(AD1900))=Seconds(0)***

***Time(Begin(AD2001))=Seconds(3187324800)***

***Time(End(AD2001))=Seconds(3218860800)***
***Duration(AD2001)=Seconds(31536000)***

Two intervals Meet if the end time of the first equals the start time of the second. The complete set of interval relations, as proposed by Allen (1983), is shown graphically in Figure 12.2 and logically below:

***Meet(i,j) ⇔ End(i)=Begin(j)***

***Before(i,j) ⇔ End(i) < Begin(j)***

***After (j,i) ⇔ Before(i, j)***

***During(i,j) ⇔ Begin(j) < Begin(i) < End(i) < End(j)***

***Overlap(i,j) ⇔ Begin(i) < Begin(j) < End(i) < End(j)***

***Begins(i,j) ⇔ Begin(i) = Begin(j)***

***Finishes(i,j) ⇔ End(i) = End(j)***

***Equals(i,j) ⇔ Begin(i) = Begin(j) ∧ End(i) = End(j)***

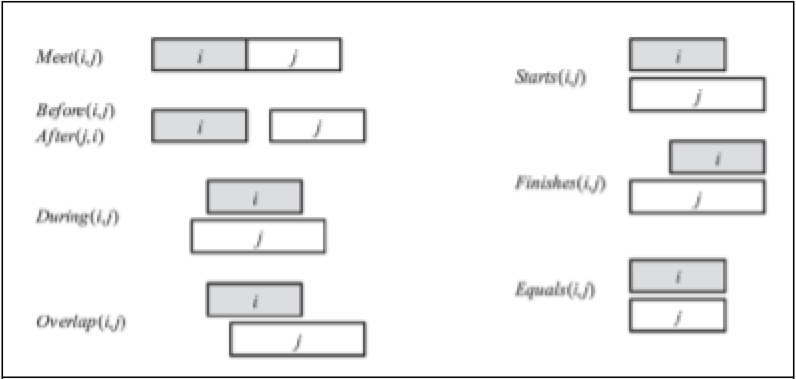


**Figure 3.14 Predicates on time intervals**

## 3.7 EVENTS

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 ∈Flyings∧ Flyer (E1, Shankar ) ∧ Origin(E1, SF) ∧ Destination (E1,DC)** we can define an alternative three-argument version of the category of flying events and say ***E1 ∈Flyings(Shankar, SF,DC)*** We then use ***Happens(E1, i)*** to say that the event E1 took place over the time interval i, and we say the same thing in functional form with Extent(***E1)=i.*** We represent time intervals by a (start, end) pair of times; that is, i = (t1, t2) is the time interval that starts at t1 and ends at t2. The complete set of predicates for one version of the event calculus is T(f, t) Fluent f is true at time t Happens(e, i) Event e happens over the time interval i Initiates(e, f, t) Event e causes fluent f to start to hold at time t Terminates(e, f, t) Event e causes fluent f to cease to hold at time t Clipped(f, i) Fluent f ceases to be true at some point during time interval i Restored (f, i) Fluent f becomes true sometime during time interval i We assume a distinguished event, Start, that describes the initial state by saying which fluents are initiated or terminated at the start time. We define T by saying that a fluent holds at a point in time if the fluent was initiated by an event at some time in the past and was not made false (clipped) by an intervening event. A fluent does not hold if it was terminated by an event and not made true (restored) by another event. Formally, the axioms are:

***Happens(e, (t1, t2)) ∧Initiates(e, f, t1) ∧¬Clipped(f, (t1, t)) ∧ t1 < t ⇒T(f, t)Happens(e, (t1, t2)) ∧ Terminates(e, f, t1)∧¬Restored (f, (t1, t)) ∧ t1 < t ⇒¬T(f, t)***

***where Clipped and Restored are defined by Clipped(f, (t1, t2)) ⇔∃ e, t, t3 Happens(e, (t, t3))∧ t1 ≤ t < t2 ∧ Terminates(e, f, t) Restored (f, (t1, t2)) ⇔∃ e, t, t3 Happens(e, (t, t3)) ∧ t1 ≤ t < t2 ∧ Initiates(e, f, t)***

## 3.8 MENTAL EVENTS AND MENTAL OBJECTS

What we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction. We will be happy just to be able to conclude that mother knows whether or not she is sitting.

We begin with the propositional attitudes that an agent can have toward mental objects: attitudes such as Believes, Knows, Wants, Intends, and Informs. The difficulty is that these attitudes do not behave like "normal" predicates.

For example, suppose we try to assert that Lois knows that Superman can fly: **Knows (Lois, CanFly(Superman))** One minor issue with this is that we normally think of CanFly(Superman) as a sentence, but here it appears as a term. That issue can be patched up just be reifying **CanFly(Superman);** making it a fluent. A more serious problem isthat, if it is true that Superman is Clark Kent, then we must conclude that Lois knows that Clark can fly: **(Superman = Clark) ∧Knows(Lois, CanFly(Superman)) |= Knows(Lois, CanFly (Clark))** Modal logic is designed to address this problem. Regular logic is concerned with a single modality, the modality of truth, allowing us to express "P is true." Modal logic includes special modal operators that take sentences (rather than terms) as arguments.

For example, "A knows P" is represented with the notation KAP, where K is the modal operator for knowledge. It takes two arguments, an agent (written as the subscript) and a sentence. The syntax of modal logic is the same as first-order logic, except that sentences can also be formed with modal operators. In first-order logic a model contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and that it isn't. Therefore, we will need a more complicated model, one that consists of a collection of possible worlds rather than just one true world. The worlds are connected in a graph by accessibility relations, one relation for each modal operator. We say that world w1 is accessible from world w0 with respect to the modal operator KA if everything in w1 is consistent with what A knows in w0, and we write this as **Acc(KA,w0,w1).** In diagrams such as Figure 12.4 we show accessibility as an arrow between possible worlds. In general, a knowledge atom KAP is true in world w if and only if P is true in every world accessible from w. The truth of more complex sentences is derived by recursive application of this rule and the normal rules of first-order logic. That means that modal logic can be used to reason about nested knowledge sentences: what one agent knows about another agent's knowledge. For example, we can say that, even though Lois doesn't know whether Superman's secret identity is Clark Kent, she does know that Clark knows: **KLois [KClark Identity(Superman, Clark )**

**∨KClark¬Identity(Superman, Clark )]** Figure 3.15 shows some possible worlds for this domain, with accessibility relations for Lois and Superman

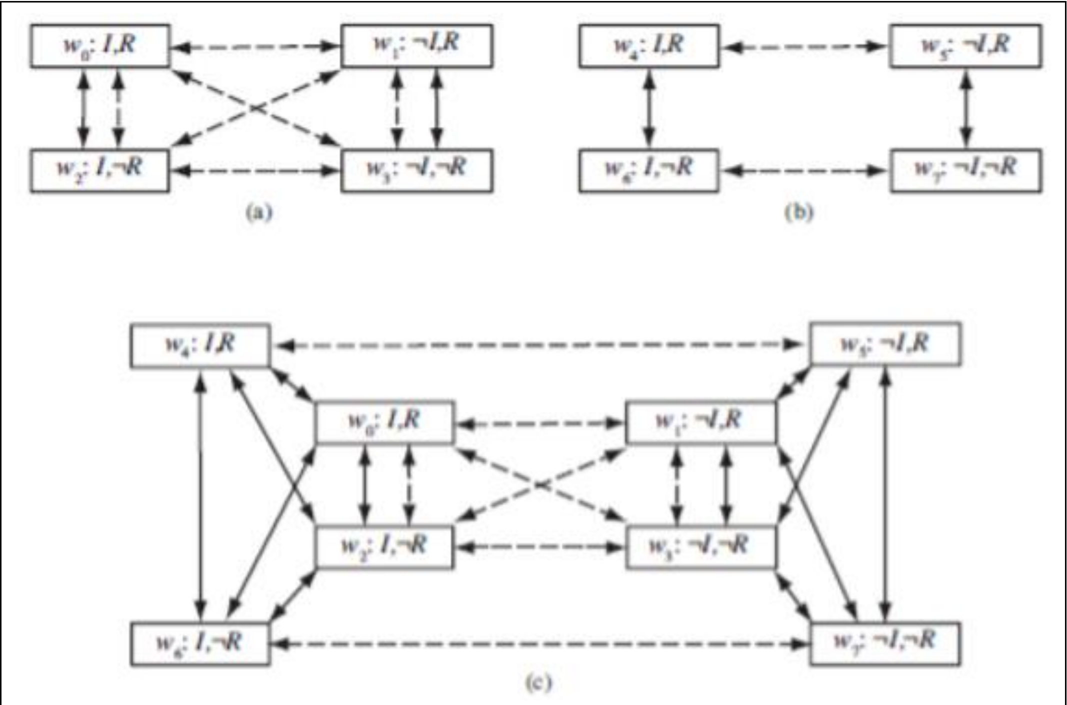


**Figure 3.15**

In the TOP-LEFT diagram, it is common knowledge that Superman knows his own identity, and neither he nor Lois has seen the weather report. So in w0 the worlds w0 and w2 are accessible to Superman; maybe rain is predicted, maybe not. For Lois all four worlds are accessible from each other; she doesn't know anything about the report or if Clark is Superman. But she does know that Superman knows whether he is Clark, because in every world that is accessible to Lois, either Superman knows I, or he knows ¬I. Lois does not know which is the case, but either way she knows Superman knows. In the TOP-RIGHT diagram it is common knowledge that Lois has seen the weather report. So in w4 she knows rain is predicted and in w6 she knows rain is not predicted. Superman does not know the report, but he knows that Lois knows, because in every world that is accessible to him, either she knows R or she knows ¬ R. In the BOTTOM diagram we represent the scenario where it is common knowledge that Superman knows his identity, and Lois might or might not have seen the weather report. We represent this by combining the two top scenarios, and adding arrows to show that Superman does not know which scenario actually holds. Lois does know, so we don't need to add any arrows for her. In w0 Superman still knows I but not R, and now he does not know whether Lois knows R. From what Superman knows, he might be in w0 or w2, in which case Lois does not know whether R is true, or he could be in w4, in which case she knows R, or w6, in which case she knows ¬R.

## 3.9 REASONING SYSTEMS FOR CATEGORIES

This section describes systems specially designed for organizing and reasoning with categories. There are two closely related families of systems: semantic networks provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties of an object on the basis of its category membership; and description logics provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.

## 3.10 SEMANTIC NETWORKS

There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects. A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled links. For example, Figure 12.5 has a Member Of link between Mary and Female Persons, corresponding to the logical assertion Mary ∈FemalePersons ; similarly, the SisterOf link between Mary and John corresponds to the assertion SisterOf (Mary, John). We can connect categories using SubsetOf links, and so on. We know that persons have female persons as mothers, so can we draw a HasMother link from Persons to FemalePersons? The answer is no, because HasMother is a relation between a person and his or her mother, and categories do not have mothers. For this reason, we have used a special notation—the double-boxed link—in Figure 12.5. This link asserts that ∀x x∈ Persons ⇒ [∀ y HasMother (x, y) ⇒ y ∈FemalePersons ] We might also want to assert that persons have two legs—that is, ∀x x∈ Persons ⇒ Legs(x, 2) The semantic network notation makes it convenient to perform inheritance reasoning. For example, by virtue of being a person, Mary inherits the property of having two legs. Thus, to find out how many legs Mary has, the inheritance algorithm followsthe MemberOf link from Mary to the category she belongs to, and then follows SubsetOf links up the hierarchy until it finds a category for which there is a boxed Legs link—in this case, the Persons category.

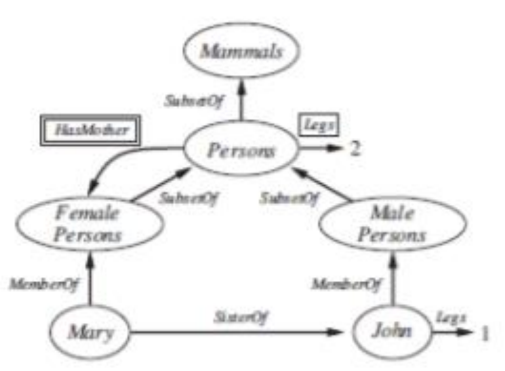


**Figure 3.16 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.**

Inheritance becomes complicated when an object can belong to more than one category or when a category can be a subset of more than one other category; this is called multiple inheritance. The drawback of semantic network notation, compared to first-order logic: the fact that links between bubbles represent only binary relations. For example, the sentence Fly(Shankar, NewYork, NewDelhi, Yesterday) cannot be asserted directly in a semantic network. Nonetheless, we can obtain the effect of n-ary assertions by reifying the proposition itself as an event belonging to an appropriate event category. Figure 12.6 shows the semantic network structure for this particular event. Notice that the restriction to binary relations forces the creation of a rich ontology of reified concepts. One of the most important aspects of semantic networks is their ability to represent.

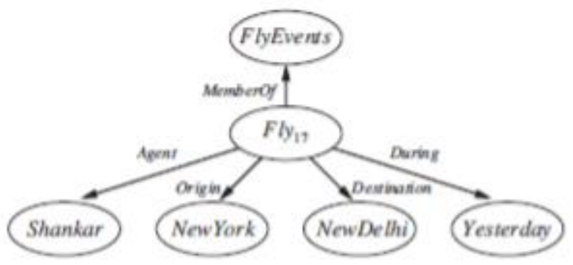
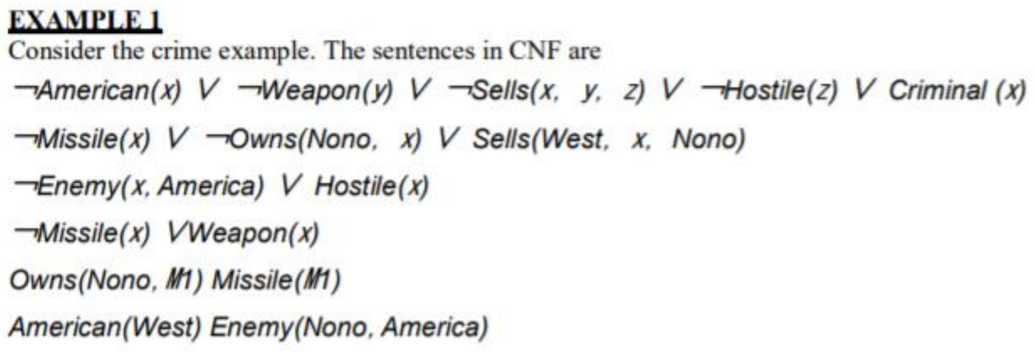


**Figure 3.17  A fragment of a semantic network showing the representation of the logical assertion *Fly(Shankar, NewYork, NewDelhi, Yesterday)***

One of the most important aspects of semantic networks is their ability to represent default values for categories. Examining Figure 3.6 carefully, one notices that John has one leg, despite the fact that he is a person and all persons have two legs. In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this is contradicted by more specific information



**EXAMPLE 1**

Consider the crime example. The sentences in CNF are

*¬American(x) V ¬Weapon(y) V ¬Sells(x, y, z) V ¬Hostile(z) V Criminal (x)*

*¬Missile(x) V ¬Owns(Nono, x) V Sells(West, x, Nono)*

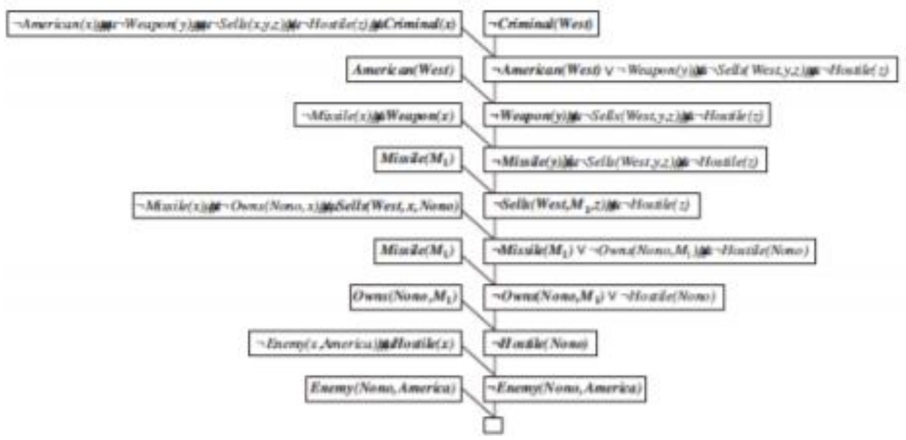
*¬Enemy(x, America) V Hostile(x)*

*¬Missile(x) VWeapon(x)*

*Owns(Nono, M1) Missile(M1)*

*American(West) Enemy(Nono, America)*

We also include the negated goal ¬ Criminal (West). The resolution proof is shown in Figure 3.18.



¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x,y,z) ∨ ¬Hostile(z) ∨ Criminal(x)
¬Criminal(West)
American(West)
¬American(West) ∨ ¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ Weapon(x)
¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ ¬Owns(Nono,x) ∨ Sells(West,x,Nono)
¬Sells(West,M1,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(M1) ∨ ¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
Owns(Nono,M1)
¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
¬Enemy(x,America) ∨ Hostile(x)
¬Hostile(Nono)
Enemy(Nono,America)
¬Enemy(Nono,America)

**Figure 3.18 A resolution proof that West is a criminal. At each step, the literals that unify are in bold.**

Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond exactly to the consecutive values of the goals variable in the backward-chaining algorithm of Figure. This is because we always choose to resolve with a clause whose positive literal unified with the left most literal of the "current" clause on the spine; this is exactly what happens in backward chaining. Thus, backward chaining is just a special case of resolution with a particular control strategy to decide which resolution to perform next.

**EXAMPLE 2**

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is a follows:

Everyone who love all animals is loved by someone.
Anyone who kills an animals is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.

Did Curiosity kill the cat? First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall$ x [$\forall$ y Animal(y) $\Rightarrow$ Loves(x,y) $\Rightarrow$ [$\exists$y Loves(y,x)]
B. $\forall$ x [$\exists$ x Animal(z) ^ Kills (x,z)] $\Rightarrow$ [$\forall$ y Loves(y,x)]
C. $\forall$ x Animals(x) $\Rightarrow$Loves(Jack, x)
D. Kills (Jack, Tuna) V Kills (Curiosity, Tuna)
E. Cat (Tuna)
F. $\forall$ x Cat(x) $\Rightarrow$ Animal (x)
¬G. ¬Kills(Curiosity, Tuna)

Now we apply the conversion procedure to convert each sentence to CNF:

| | |
|---|---|
| A1. | Animal(F(x)) v Loves (G(x),x) |
| A2. | ¬Loves(x,F(x)) v Loves (G(x),x) |
| B. | ¬Loves(y,x) v ¬ Animal(z) V ¬ Kills(x,z) |
| C. | ¬Animal(x) v Loves(Jack, x) |
| D. | Kills(Jack, Tuna) v Kills (Curiosity, Tuna) |
| E. | Cat (Tuna) |
| F. | ¬Cat(x) v Animal (x) |
| ¬G. | ¬Kills (Curiosity, Tuna) |

The resolution proof that Curiosity kills the cat is given in Figure. In English, the proof could be paraphrased as follows:

Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore Curiosity killed the cat.

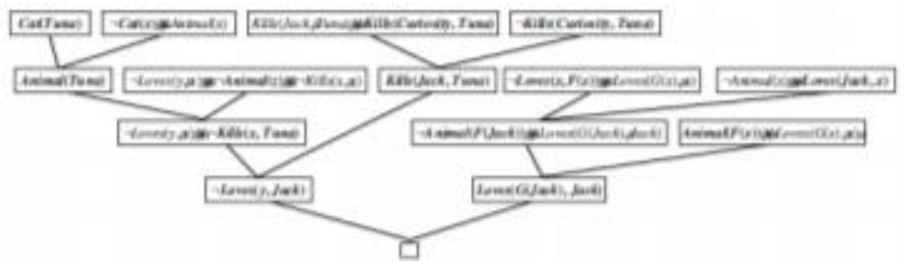


**Figure 3.19 A resolution proof that Curiosity killed that Cat. Notice then use of factoring in the derivation of the clause Loves(G(Jack), Jack). Notice also in the upper right, the unification of Loves(x,F(x)) and Loves(Jack,x) can only succeed after the variables have been standardized apart**

The proof answers the question “Did Curiosity kill the cat?” but often we want to pose more general questions, such as “Who killed the cat?” Resolution can do this, but it takes a little more work to obtain the answer. The goal is ∃w Kills (w, Tuna), which, when negated become ¬Kills (w, Tuna) in CNF, Repeating the proof in Figure with the new negated goal, we obtain a similar proof tree, but with the substitution {w/Curiosity} in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof.

**EXAMPLE 3**

1. All people who are graduating are happy.

2. All happy people smile.
3. Someone is graduating.
4. Conclusion: Is someone smiling?

**Solution**

**Convert the sentences into predicate Logic**

1. ∀x graduating(x) -> happy(x)
2. ∀x happy(x) -> smile(x)
3. ∃x graduating(x)
4. ∃x smile(x)

**Convert to clausal form**

(i) Eliminate the → sign

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(x) vsmile(x)
3. ∃x graduating(x)
4. -∃x smile(x)

**(ii) Reduce the scope of negation**

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(x) vsmile(x)
3. ∃x graduating(x)
4. ∀x¬ smile(x)

**(iii) Standardize variable apart**

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(y) vsmile(y)
3. ∃x graduating(z)
4. ∀x¬ smile(w)

**(iv) Move all quantifiers to the left**

1. ∀x¬ graduating(x) vhappy(x)
2. ∀x¬ happy(y) vsmile(y)
3. ∃x graduating(z)
4. ∀w¬ smile(w)

**(v) Eliminate ∃**

1. ∀x¬ graduating(x) vhappy(x)
2. ∀x¬ happy(y) vsmile(y)

3. graduating(name1)

4. ∀w¬ smile(w)

**(vi) Eliminate∀**

1. ¬graduating(x) vhappy(x)

2. ¬happy(y) vsmile(y)

3. graduating(name1)

4. ¬smile(w)

**(vii) Convert to conjunct of disjuncts form**

**(viii) Make each conjunct a separate clause.**

**(ix) Standardize variables apart again.**

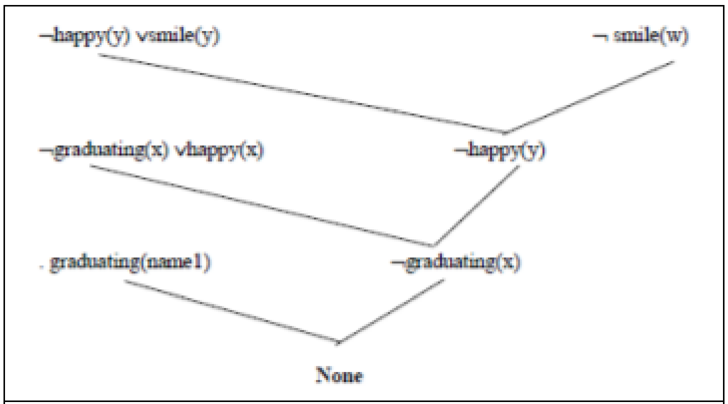


**Figure 3.20 Standardize variables**

Thus, we proved someone is smiling.

**EXAMPLE 4**

Explain the unification algorithm used for reasoning under predicate logic with an example. Consider the following facts

a. Team India

b. Team Australia

c. Final match between India and Australia

d. India scored 350 runs, Australia scored 350 runs, India lost 5 wickets, Australia lost 7 wickets.

f. If the scores are same the team which lost minimum wickets wins the match.

Represent the facts in predicate, convert to clause form and prove by resolution "India wins the match".

**Solution**

**Convert into predicate Logic**

(a) team(India)

(b) team(Australia)

(c) team(India) ^ team(Australia) → final_match(India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x team(x) ^ wins(x) → score(x,mat_runs)

(f) ∃xy score(x,equal(y)) ^ wicket(x,min) ^ final_match(x,y) → win(x)

**Convert to clausal form**

**(i) Eliminate the → sign**

(a) team(India)

(b) team(Australia)

(c) ¬(team(India) ^ team(Australia) v final_match(India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x¬ (team(x) ^ wins(x)) vscore(x,max_runs))

(f) ∃xy¬ (score(x,equal(y)) ^ wicket(x,min) ^final_match(x,y)) vwin(x)

**(ii) Reduce the scope of negation**

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match(India, Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x¬ team(x) v ¬wins(x) vscore(x,max_runs))

(f) ∃xy¬ (score(x,equal(y)) v ¬ wicket(x,min_wicket) v¬final_match(x,y)) vwin(x)

**(iii) Standardize variables apart**

**(iv) Move all quantifiers to the left**

**(v) Eliminate ∃**

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match (India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ¬team(x) v ¬wins(x) vscore(x, max_runs))

(f) ¬score(x,equal(y)) v¬wicket(x,min_wicket) v-final_match(x,y)) vwin(x)

**(vi) Eliminate∀**

**(vii) Convert to conjunct of disjuncts form.**

**(viii) Make each conjunct a separate clause.**

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match (India,Australia)

(d) score(India,350)
Score(Australia,350)
Wicket(India,5)
Wicket(Austrialia,7)

(e) ¬team(x) v ¬wins(x) vscore(x,max_runs))

(f) ¬score(x,equal(y)) v¬wicket(x,min_wicket) v¬final_match(x,y)) vwin(x)

**(ix) Standardize variables apart again**

**To prove:**win(India)

**Disprove:**¬win(India)

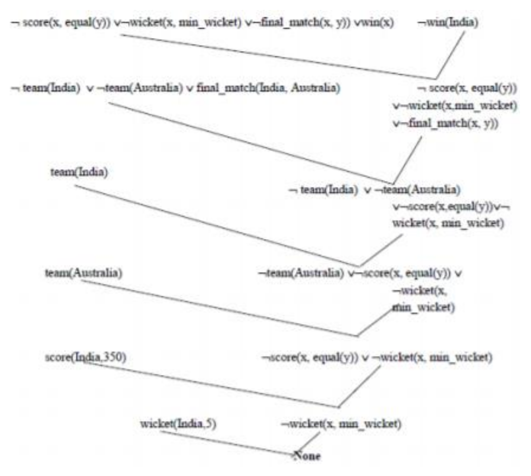

¬ score(x, equal(y)) v¬wicket(x, min_wicket) v¬final_match(x, y)) vwin(x)
¬win(India)
¬ team(India) v ¬team(Australia) v final_match(India, Australia)
¬ score(x, equal(y)) v¬wicket(x,min_wicket) v¬final_match(x, y))
team(India)
¬ team(India) v ¬team(Australia) v¬score(x,equal(y))v¬ wicket(x, min_wicket)
team(Australia)
¬team(Australia) v¬score(x, equal(y)) v ¬wicket(x, min_wicket)
score(India,350)
¬score(x, equal(y)) v ¬wicket(x, min_wicket)
wicket(India,5)
¬wicket(x, min_wicket)
None

**Figure 3.21 Standardize variables**

Thus, proved India wins match.

**EXAMPLE 5**

**Problem 3**

Consider the following facts and represent them in predicate form:

F1. There are 500 employees in ABC company.

F2. Employees earning more than Rs. 5000 per tax.

F3. John is a manger in ABC company.

F4. Manger earns Rs. 10,000.

Convert the facts in predicate form to clauses and then prove by resolution: “John pays tax”.

**Solution**

**Convert into predicate Logic**

1. company(ABC) ^employee(500,ABC)
2. ∃x company(ABC) ^employee(x,ABC) ^ earns(x,5000)→pays(x,tax)
3. manager(John,ABC)
4. ∃x manager(x, ABC)→earns(x,10000)

**Convert to clausal form**

**(i) Eliminate the → sign**

1. company(ABC) ^employee(500,ABC)
2. ∃x¬(company(ABC) ^employee(x,ABC) ^ earns(x,5000)) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

**(ii) Reduce the scope of negation**

1. company(ABC) ^employee(500,ABC)
2. ∃x¬ company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

**(iii) Standardize variables apart**

1. company(ABC) ^employee(500,ABC)
2. ∃x¬ company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

**(iv) Move all quantifiers to the left**

**(v) Eliminate ∃**

1. company(ABC) ^employee(500,ABC)
2. ¬company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ¬manager(x, ABC) v earns(x,10000)

**(vi) Eliminate∀**

**(vii) Convert to conjunct of disjuncts form**

**(viii) Make each conjunct a separate clause.**

1. (a) company(ABC)

   (b) employee(500,ABC)
2. ¬company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ¬manager(x, ABC) v earns(x,10000)

**(ix) Standardize variables apart again.**

**Prove:**pays(John,tax)

**Disprove:**¬pays(John,tax)

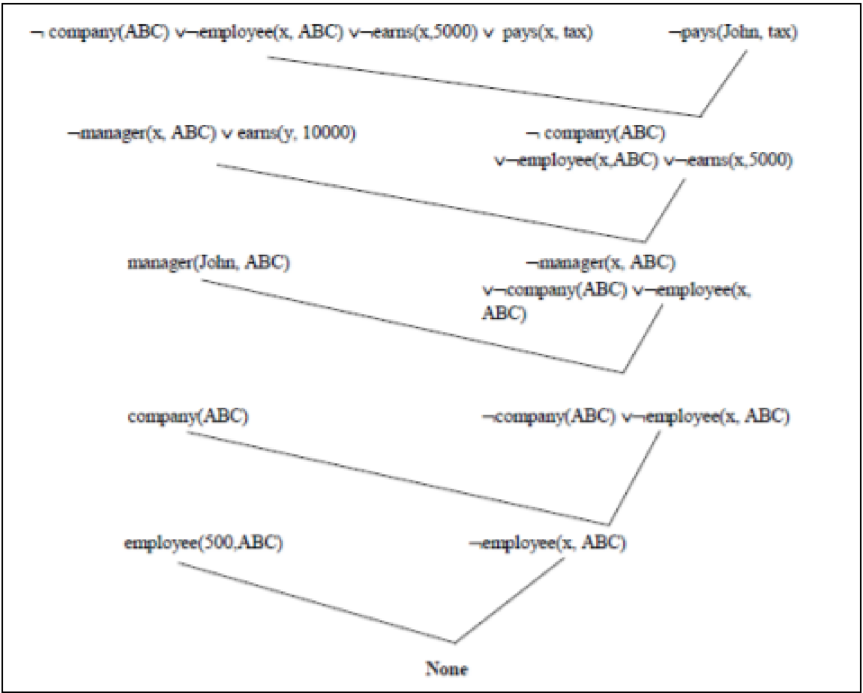

¬ company(ABC) v¬employee(x, ABC) v¬earns(x,5000) v pays(x, tax)
¬pays(John, tax)
¬manager(x, ABC) v earns(y, 10000)
¬ company(ABC) v¬employee(x,ABC) v¬earns(x,5000)
manager(John, ABC)
¬manager(x, ABC) v¬company(ABC) v¬employee(x, ABC)
company(ABC)
¬company(ABC) v¬employee(x, ABC)
employee(500,ABC)
¬employee(x, ABC)
None

**Figure 3.22 Standardize variables**

Thus, proved john pays tax.

**EXAMPLE 6 & EXAMPLE 7**

**Problem 4**

If a perfect square is divisible by a prime p then it is also divisible by square of p.
Every perfect square is divisible by some prime.
36 is a perfect square.

**Convert into predicate Logic**

1. ∀xyperfect_sq(x) ^ prime(h) ^divideds(x,y) → divides(x,square(y))

2. $\forall x \exists y$ perfect)sq(x) ^prime(y) ^divides(x,y)
3. perfect_sq(36)

**Problem 5**

1. Marcus was a a man

   man(Marcus)
2. Marcus was a Pompeian

   Pompeian(Marcus)
3. All Pompeians were Romans

   ∀x (Pompeians(x) → Roman(x))
4. Caesar was ruler

   Ruler(Caesar)
5. All Romans were either loyal to Caesar or hated him.

   ∃x (Roman(x) → loyalto(x,Caesar) v hate(x,Caesar))
6. Everyone is loyal to someone

   ∀x ∃y (person(x) → person(y) ^ loyalto(x,y))
7. People only try to assassinate rulers they are not loyal to

   ∀x ∃y (person(x) ^ ruler(y)^ tryassassinate(x,y) → -loyalto(x,y))
8. Marcus tried to assassinate Caesar

   tryassassinate(Marcus, Caesar)
9. All men are persons

   ∀x (max(x) → person(x))

**Example**

**Trace the operation of the unification algorithm on each of the following pairs of literals:**

**i) f(Marcus) and f(Caesar)**

**ii) f(x) and f(g(y))**

**iii) f(marcus,g(x,y)) and f(x,g(Caesar, Marcus))**

In propositional logic it is easy to determine that two literals can not both be true at the same time. Simply look for L and ~L. In predicate logic, this matching process is more complicated, since bindings of variables must be considered.

For example man (john) and man(john) is a contradiction while man (john) and man(Himalayas) is not. Thus in order to determine contradictions we need a matching procedure that compares two literals and discovers whether there exist a set of substitutions that makes them identical. There is a recursive procedure that does this matching. It is called Unification algorithm.

In Unification algorithm each literal is represented as a list, where first element is the name of a predicate and the remaining elements are arguments. The argument may be a single element (atom) or may be another list. For example we can have literals as

(tryassassinate Marcus Caesar)
(tryassassinate Marcus (ruler of Rome))

To unify two literals, first check if their first elements re same. If so proceed. Otherwise they can not be unified. For example the literals

(try assassinate Marcus Caesar)
(hate Marcus Caesar)

Can not be Unfied. The unification algorithm recursively matches pairs of elements, one pair at a time. The matching rules are :

i) Different constants, functions or predicates can not match, whereas identical ones can.

ii) A variable can match another variable, any constant or a function or predicate expression, subject to the condition that the function or [predicate expression must not contain any instance of the variable being matched (otherwise it will lead to infinite recursion).

iii) The substitution must be consistent. Substituting y for x now and then z for x later is inconsistent (a substitution y for x written as y/x).

The Unification algorithm is listed below as a procedure UNIFY (L1, L2). It returns a list representing the composition of the substitutions that were performed during the match. An empty list NIL indicates that a match was found without any substitutions. If the list contains a single value F, it indicates that the unification procedure failed.

**UNIFY (L1, L2)**

```
1. if L1 or L2 is an atom part of same thing do
(a) if L1 or L2 are identical then return NIL
(b) else if L1 is a variable then do
(i) if L1 occurs in L2 then return F else return (L2/L1)
© else if L2 is a variable then do
(i) if L2 occurs in L1 then return F else return (L1/L2)
else return F.
2. If length (L!) is not equal to length (L2) then return F.
3. Set SUBST to NIL
(at the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2).
```

```
4. For I = 1 to number of elements in L1 do
i) call UNIFY with the ith element of L1 and I'th element of L2, putting the result in S
ii) if S = F then return F
iii) if S is not equal to NIL then do
(A) apply S to the remainder of both L1 and L2
(B) SUBST := APPEND (S, SUBST) return SUBST.
```

Consider a knowledge base containing just two sentences: P(a) and P(b). Does this knowledge base entail ∀ x P(x)? Explain your answer in terms of models.

The knowledge base does not entail *∀x P*(*x*). To show this, we must give a model where *P*(*a*) and *P*(*b*) but *∀x P*(*x*) is false. Consider any model with three domain elements, where *a* and *b* refer to the first two elements and the relation referred to by *P* holds only for those two elements.

Is the sentence ∃ x, y x = y valid? Explain

The sentence *∃ x, y x*=*y* is valid. A sentence is valid if it is true in every model. An existentially quantified sentence is true in a model if it holds under any extended interpretation in which its variables are assigned to domain elements. According to the standard semantics of FOL as given in the chapter, every model contains at least one domain element, hence, for any model, there is an extended interpretation in which *x* and *y* are assigned to the first domain element. In such an interpretation, *x*=*y* is true.

What is ontological commitment (what exists in the world) of first order logic? Represent the sentence "Brothers are siblings" in first order logic?

Ontological commitment means what assumptions language makes about the nature if reality. Representation of "Brothers are siblings" in first order logic is ∀ x, y [Brother (x, y) ⇒ Siblings (x, y)]

Differentiate between propositional and first order predicate logic?

Following are the comparative differences versus first order logic and propositional logic.

1) Propositional logic is less expressive and do not reflect individual object`s properties explicitly. First order logic is more expressive and can represent individual object along with all its properties.

2) Propositional logic cannot represent relationship among objects whereas first order logic can represent relationship.

3) Propositional logic does not consider generalization of objects where as first order logic handles generalization.

4) Propositional logic includes sentence letters (A, B, and C) and logical connectives, but not quantifier. First order logic has the same connectives as

propositional logic, but it also has variables for individual objects, quantifier, symbols for functions and symbols for relations.

Represent the following sentence in predicate form:

"All the children like sweets"

∀x child(x) ∩ sweet(y) ∩ likes (x,y).

Illustrate the use of first order logic to represent knowledge. The best way to find usage of First order logic is through examples. The examples can be taken from some simple domains. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge. Assertions and queries in first-order logic Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called assertions. For example, we can assert that John is a king and that kings are persons: Where KB is knowledge base. TELL(KB, ∀x King(x) => Person(x)). We can ask questions of the knowledge base using AS K. For example, returns true. Questions asked using ASK are called queries or goals ASK(KB, Person(John)) Will return true. (ASK KBto find whether Jon is a king) ASK (KB, ∃x person(x)) The kinship domain The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William7' and rules such as "One's grandmother is the mother of one's parent." Clearly, the objects in our domain are people. We will have two unary predicates, Male and Female. Kinship relations-parenthood, brotherhood, marriage, and so on-will be represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle. We will use functions for Mother and Father.

**What are the characteristics of multi agent systems?**

Each agent has just incomplete information and is restricted in is capabilities. The system control is distributed. Data is decentralized. Computation is asynchronous Multi agent environments are typically open ad have no centralized designer. Multi agent environments provide an infrastructure specifying communication and interaction protocols. Multi agent environments have agents that are autonomous and distributed, and may be self interested or cooperative.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

## SCHOOL OF ELECTRICAL AND ELECTRONICS

### DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINERING

# UNIT- IV- ARTIFICIAL INTELLIGENCE- SECA3011

# UNIT 4

# SOFTWARE AGENTS

Architecture for Intelligent Agents – Agent communication – Negotiation and Bargaining – Argumentation among Agents – Trust and Reputation in Multi-agent systems.

## 4.1 DEFINITION

Agent architectures, like software architectures, are formally a description of the elements from which a system is built and the manner in which they communicate. Further, these elements can be defined from patterns with specific constraints. [Shaw/Garlin 1996]

1. A number of common architectures exist that go by the names pipe-and filter or layered architecture.
2. these define the interconnections between components.
3. Pipe-and-Filter defines a model where data is moved through a set of one or more objects that perform a transformation.
4. Layered simply means that the system is comprised of a set of layers that provide a specific set of logical functionality and that connectivity is commonly restricted to the layers contiguous to one another.

## 4.2 TYPES OF ARCHITECTURES

Based on the goals of the agent application, a variety of agent architectures exist to help. This section will introduce some of the major architecture types and applications for which they can be used.

1. Reactive architectures
2. Deliberative architectures
3. Blackboard architectures
4. Belief-desire-intention (BDI) architecture
5. Hybrid architectures
6. Mobile architectures

### 1. REACTIVE ARCHITECTURES

1. A reactive architecture is the simplest architecture for agents.
2. In this architecture, agent behaviors are simply a mapping between stimulus and response.
3. The agent has no decision-making skills, only reactions to the environment in which it exists.

4. The agent simply reads the environment and then maps the state of the environment to one or more actions. Given the environment, more than one action may be appropriate, and therefore the agent must choose.

5. The advantage of reactive architectures is that they are extremely fast.

6. This kind of architecture can be implemented easily in hardware, or fast in software lookup.

7. The disadvantage of reactive architectures is that they apply only to simple environments.

8. Sequences of actions require the presence of state, which is not encoded into the mapping function.

**2. DELIBERATIVE ARCHITECTURES**

1. A deliberative architecture, as the name implies, is one that includes some deliberation over the action to perform given the current set of inputs.

2. Instead of mapping the sensors directly to the actuators, the deliberative architecture considers the sensors, state, prior results of given actions, and other information in order to select the best action to perform.

3. The mechanism for action selection as is undefined. This is because it could be a variety of mechanisms including a production system, neural network, or any other intelligent algorithm.

4. The advantage of the deliberative architecture is that it can be used to solve much more complex problems than the reactive architecture.

5. It can perform planning, and perform sequences of actions to achieve a goal.

6. The disadvantage is that it is slower than the reactive architecture due to the deliberation for the action to select.

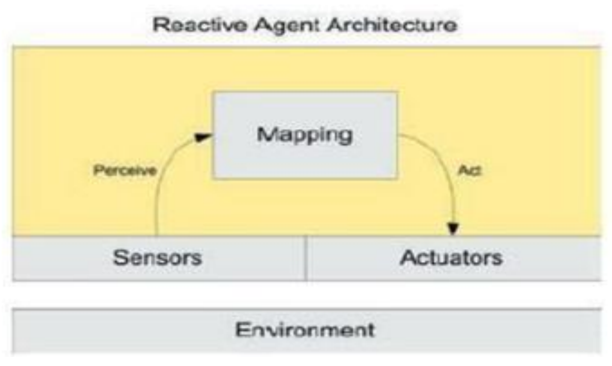


**Figure 4.1 Reactive architecture defines a simple agent**

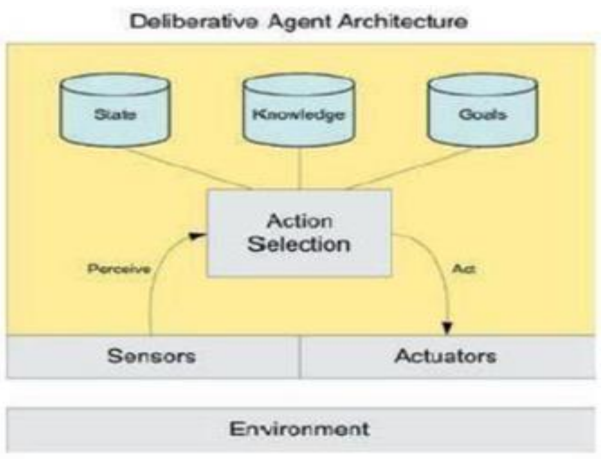


**Figure 4.2 A deliberative agent architecture considers its actions**

**3. BLACKBOARD ARCHITECTURES**

1. The blackboard architecture is a very common architecture that is also very interesting.
2. The first blackboard architecture was HEARSAY-II, which was a speech understanding system. This architecture operates around a global work area call the blackboard.
3. The blackboard is a common work area for a number of agents that work cooperatively to solve a given problem.
4. The blackboard therefore contains information about the environment, but also intermediate work results by the cooperative agents.
5. In this example, two separate agents are used to sample the environment through the available sensors (the sensor agent) and also through the available actuators (action agent).
6. The blackboard contains the current state of the environment that is constantly updated by the sensor agent, and when an action can be performed (as specified in the blackboard), the action agent translates this action into control of the actuators.
7. The control of the agent system is provided by one or more reasoning agents.
8. These agents work together to achieve the goals, which would also be contained in the blackboard.
9. In this example, the first reasoning agent could implement the goal definition behaviors, where the second reasoning agent could implement the planning portion (to translate goals into sequences of actions).
10. Since the blackboard is a common work area, coordination must be provided such that agents don’t step over one another.

11. For this reason, agents are scheduled based on their need. For example, agents can monitor the blackboard, and as information is added, they can request the ability to operate.

12. The scheduler can then identify which agents desire to operate on the blackboard, and then invoke them accordingly.

13. The blackboard architecture, with its globally available work area, is easily implemented with a multi-threading system.

14. Each agent becomes one or more system threads. From this perspective, the blackboard architecture is very common for agent and non-agent systems.

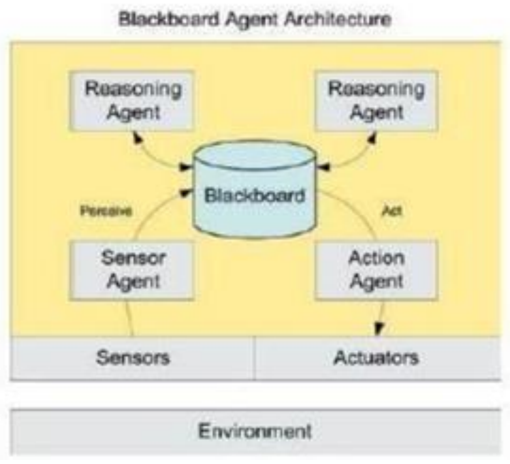


**Figure 4.3 The blackboard architecture supports multi-agent problem solving**

## 4. BELIEF-DESIRE-INTENTION (BDI) ARCHITECTURE

1. BDI, which stands for Belief-Desire-Intention, is an architecture that follows the theory of human reasoning as defined by Michael Bratman.

2. Belief represents the view of the world by the agent (what it believes to be the state of the environment in which it exists). Desires are the goals that define the motivation of the agent (what it wants to achieve).

3. The agent may have numerous desires, which must be consistent. Finally, Intentions specify that the agent uses the Beliefs and Desires in order to choose one or more actions in order to meet the desires.

4. As we described above, the BDI architecture defines the basic architecture of any deliberative agent. It stores a representation of the state of the environment (beliefs), maintains a set of goals (desires), and finally, an intentional element that maps desires to beliefs (to provide one or more actions that modify the state of the environment based on the agent's needs).

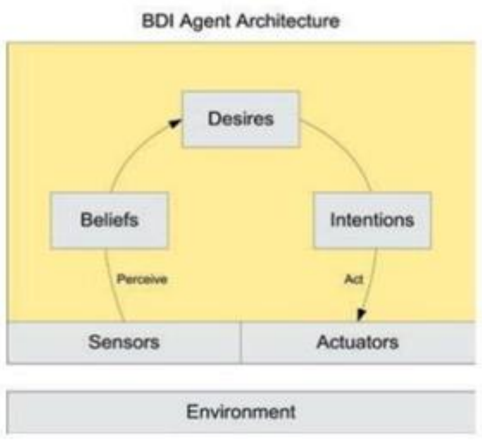


**Figure 4.4 The BDI architecture desires to model mental attributes**

## 5. HYBRID ARCHITECTURES

1. As is the case in traditional software architecture, most architectures are hybrids.
2. For example, the architecture of a network stack is made up of a pipe-and-filter architecture and a layered architecture.
3. This same stack also shares some elements of a blackboard architecture, as there are global elements that are visible and used by each component of the architecture.
4. The same is true for agent architectures. Based on the needs of the agent system, different architectural elements can be chosen to meet those needs.

## 6. MOBILE ARCHITECTURES

1. The final architectural pattern that we'l discuss is the mobile agent architecture.
2. This architectural pattern introduces the ability for agents to migrate themselves between hosts. The agent architecture includes the mobility element, which allows an agent to migrate from one host to another.
3. An agent can migrate to any host that implements the mobile framework.
4. The mobile agent framework provides a protocol that permits communication between hosts for agent migration.
5. This framework also requires some kind of authentication and security, to avoid a mobile agent framework from becoming a conduit for viruses. Also implicit in the mobile agent framework is a means for discovery.

6. For example, which hosts are available for migration, and what services do they provide? Communication is also implicit, as agents can communicate with one another on a host, or across hosts in preparation for migration.

7. The mobile agent architecture is advantageous as it supports the development of intelligent distributed systems. But a distributed system that is dynamic, and whose configuration and loading is defined by the agents themselves.

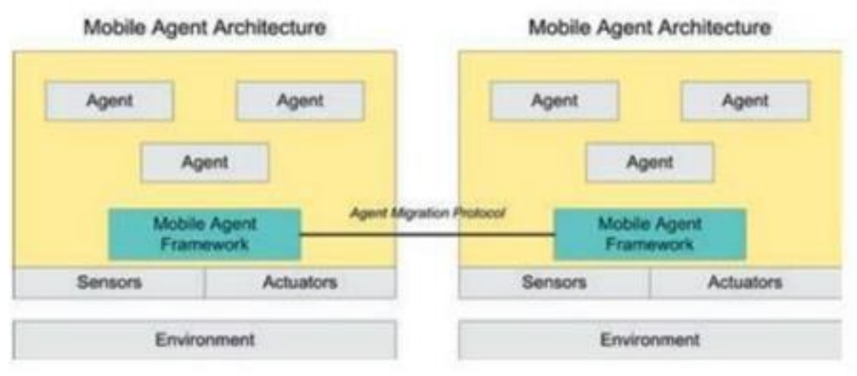


**Figure 4.5 The mobile agent framework supports agent mobility**

## 7. ARCHITECTURE DESCRIPTIONS

1. Subsumption Architecture (Reactive Architecture)
2. Behavior Networks (Reactive Architecture)
3. ATLANTIS (Deliberative Architecture)
4. Homer (Deliberative Arch)
5. BB1 (Blackboard)
6. Open Agent Architecture (Blackboard)
7. Procedural Reasoning System (BDI)
8. Aglets (Mobile)
9. Messengers (Mobile)
10. Soar (Hybrid)

### ➢ SUBSUMPTION ARCHITECTURE (REACTIVE ARCHITECTURE)

1. The Subsumption architecture, originated by Rodney Brooks in the late 1980s, was created out of research in behavior-based robotics.

2. The fundamental idea behind subsumption is that intelligent behavior can be created through a collection of simple behavior modules.

3. These behavior modules are collected into layers. At the bottom are behaviors that are reflexive in nature, and at the top, behaviors that are more complex. Consider the abstract model shown in Figure.

4. At the bottom (level 0) exist the reflexive behaviors (such as obstacle avoidance). If these behaviors are required, then level 0 consumes the inputs and provides an action at the output. But no obstacles exist, so the next layer up is permitted to subsume control.

5. At each level, a set of behaviors with different goals compete for control based on the state of the environment.

6. To support this capability levels can be inhibited (in other words, their outputs are disabled). Levels can also be suppressed such that sensor inputs are routed to higher layers. As shown in Figure.

7. Subsumption is a parallel and distributed architecture for managing sensors and actuators. The basic premise is that we begin with a simple set of behaviors, and once we've succeeded there, we extend with additional levels and higher- level behaviors.

8. For example, we begin with obstacle avoidance and then extend for object seeking. From this perspective, the architecture takes a more evolutionary design approach.

9. Subsumption does have its problems. It is simple, but it turns out not to be extremely extensible. As new layers are added, the layers tend to interfere with one another, and then the problem becomes how to layer the behaviors such that each has the opportunity to control when the time is right.

10. Subsumption is also reactive in nature, meaning that in the end, the architecture still simply maps inputs to behaviors (no planning occurs, for example). What subsumption does provide is a means to choose which behavior for a given environment.

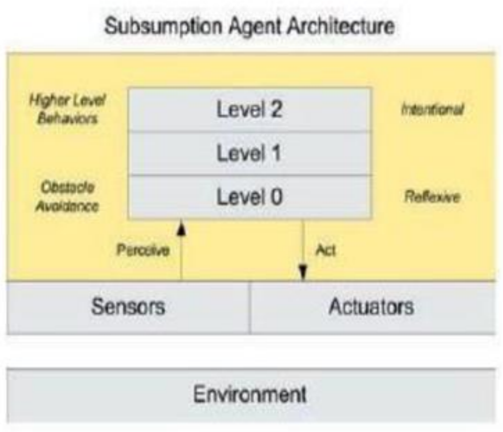


**Figure 4.6 Architectural view of the subsumption architecture**

## ➢ BEHAVIOR NETWORKS (REACTIVE ARCHITECTURE)

1. Behavior networks, created by Pattie Maes in the late 1980s, is another reactive architecture that is distributed in nature. Behavior networks attempt to answer the question, which action is best suited for a given situation.

2. As the name implies, behavior networks are networks of behaviors that include activation links and inhibition links.

3. An example behavior network for a game agent is shown in Figure. As shown in the legend, behaviors are rectangles and define the actions that the agent may take (attack, explore, reload, etc.).

4. The ovals specify the preconditions for actions to be selected, which are inputs from the environment.

5. Preconditions connect to behaviors through activation links (they promote the behavior to be performed) or inhibition links (that inhibit the behavior from being performed).

6. The environment is sampled, and then the behavior for the agent is selected based on the current state of the environment. The first thing to note is the activation and inhibition links. For example, when the agent's health is low, attack and exploration are inhibited, leaving the agent to find the nearest shelter. Also, while exploring, the agent may come across medkits or ammunition.

7. If a medkit or ammunition is found, it's used. Maes' algorithm referred to competence modules, which included preconditions (that must be fulfilled before the module can activate), actions to be performed, as well as a level of activation.

8. The activation level is a threshold that is used to determine when a competence module may activate.

9. The algorithm also includes decay, such that activiations dissipate over time. Like the subsumption architecture, behavior networks are instances of Behavior-Based Systems (BBS). The primitive actions produced by these systems are all behaviors, based on the state of the environment.

10. Behavior networks are not without problems. Being reactive, the architecture does not support planning or higher- level behaviors. The architecture can also suffer when behaviors are highly inter-dependent. With many competing goals, the behavior modules can grow dramatically in order to realize the intended behaviors. But for simpler architecture, such as the FPS game agent in Figure 4.7, this algorithm is ideal.

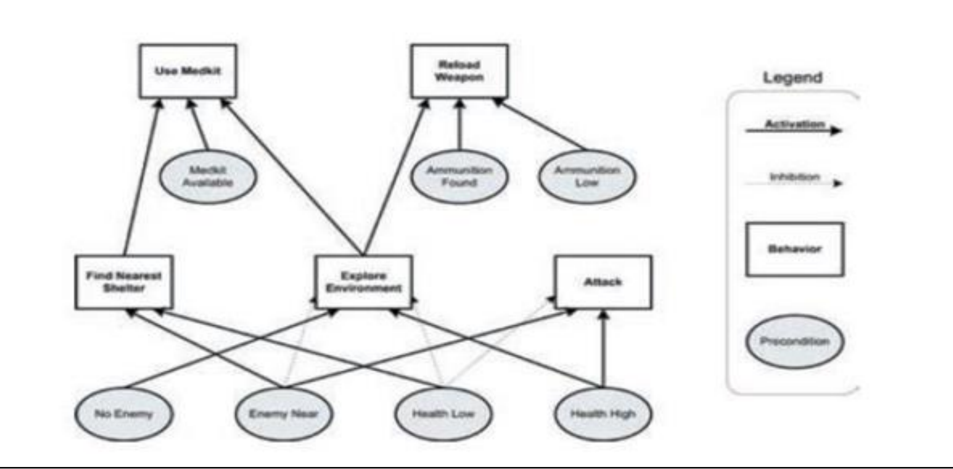


**Figure 4.7 Behavior network for a simple game agent**

➢ **ATLANTIS (Deliberative Architecture)**

1. The goal of ATLANTIS (A Three-Layer Architecture for Navigating Through Intricate Situations), was to create a robot that could navigate through dynamic and imperfect environments in pursuit of explicitly stated high-level goals.

2. ATLANTIS was to prove that a goal-oriented robot could be built from a hybrid architecture of lower-level reactive behaviors and higher- level deliberative behaviors.

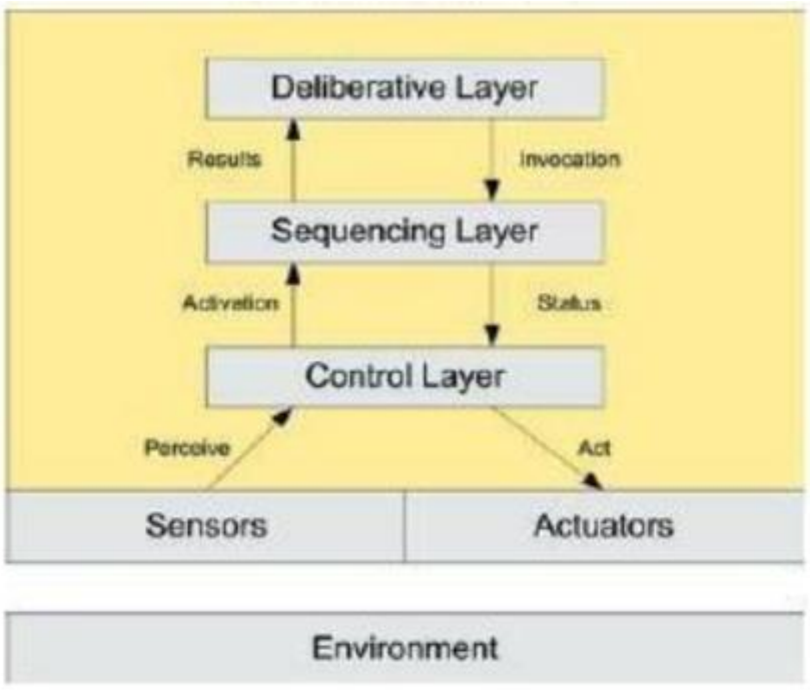


**Figure 4.8 ATLANTIS Architecture**

3. Where the subsumption architecture allows layers to subsume control, ATLANTIS operates on the assumption that these behaviors are not exclusive of one another. The lowest layer can operate in a reactive fashion to the immediate needs of the environment, while the uppermost layer can support planning and more goal-oriented behaviors.

4. In ATLANTIS, control is performed from the bottom- up. At the lowest level (the control layer) are the reactive behaviors.

5. These primitive level actions are capable of being executed first, based on the state of the environment. At the next layer is the sequencing layer. This layer is responsible for executing plans created by the deliberative layer.

6. The deliberative layer maintains an internal model of the environment and creates plans to satisfy goals.

7. The sequencing layer may or may not complete the plan, based on the state of the environment. This leaves the deliberation layer to perform the computationally expensive tasks. This is another place that the architecture is a hybrid.

8. The lower- level behavior-based methods (in the controller layer) are integrated with higher- level classical AI mechanisms (in the deliberative layer). Interestingly, the deliberative layer does not control the sequencing layer, but instead simply advises on sequences of actions that it can perform.

9. The advantage of this architecture is that the low- level reactive layer and higher- level intentional layers are asynchronous. This means that while deliberative plans are under construction, the agent is not susceptible to the dynamic environment. This is because even though planning can take time at the deliberative layer, the controller can deal with random events in the environment.

➢ **HOMER (DELIBERATIVE ARCH)**

1. Homer is another interesting deliberative architecture that is both modular and integrated. Homer was created by Vere and Bickmore in 1990 as a deliberative architecture with some very distinct differences to other architectures.

2. At the core of the Homer architecture is a memory that is divided into two parts. The first part contains general knowledge (such as knowledge about the environment). The second part is called episodic knowledge, which is used to record experiences in the environment (perceptions and actions taken).

3. The natural language processor accepts human input via a keyboard, and parses and responds using a sentence generator. The temporal planner creates dynamic plans to satisfy predefined goals, and is capable of replanning if the environment requires.

4. The architecture also includes a plan executor (or interpreter), which is used to execute the plan at the actuators. The architecture also included a variety of monitor processes. The basic idea behind Homer was an architecture for general intelligence.

5. The keyboard would allow regular English language input, and a terminal would display generated English language sentences. The user could therefore communicate with Homer to specify goals and receive feedback via the terminal.

6. Homer could log perceptions of the world, with timestamps, to allow dialogue with the user and rational answers to questions. Reflective (monitor) processes allow Homer to add or remove knowledge from the episodic memory.

7. Homer is an interesting architecture implementing a number of interesting ideas, from natural language processing to planning and reasoning. One issue found in Homer is that when the episodic memory grows large, it tends to slow down the overall operation of the agent.

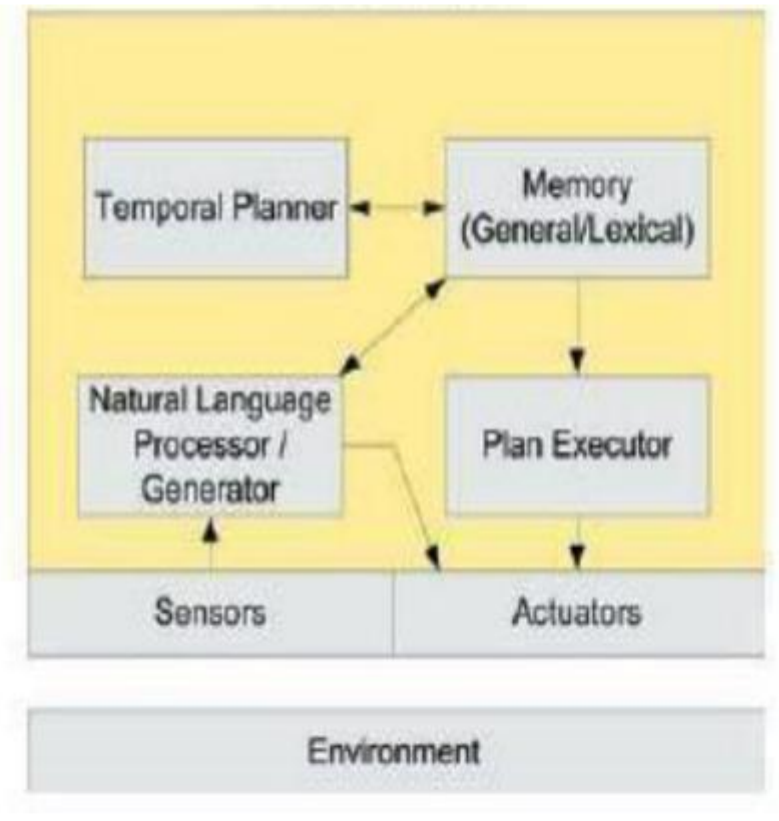


**Figure 4.9 Homer Architecture**

➢ **BB1 (BLACKBOARD)**

1. BB1 is a domain- independent blackboard architecture for AI systems created by Barbara Hayes- Roth. The architecture supports control over problem solving as well as explaining its actions. The architecture is also able to learn new domain knowledge.

2. BB1 includes two blackboards; a domain blackboard which acts as the global database and a control blackboard, which is used for generating a solution to the given control problem.

3. The key behind BB1 is its ability to incrementally plan. Instead of defining a complete plan for a given goal, and then executing that plan, BB1 dynamically develops the plan and adapts to the changes in the environment. This is key for dynamic environments, where unanticipated changes can lead to brittle plans that eventually fail.

➢ **PROCEDURAL REASONING SYSTEM (BDI)**

1. The Procedural Reasoning System (PRS) is a general-purpose architecture that's ideal for reasoning environments where actions can be defined by predetermined procedures (action sequences).

2. PRS is also a BDI architecture, mimicking the theory on human reasoning. PRS integrates both reactive and goal-directed deliberative processing in a distributed architecture.

3. The architecture is able to build a world-model of the environment (beliefs) through interacting with environment sensors.

4. Actions can also be taken through an intentions module. At the core is an interpreter (or reasoner) which selects a goal to meet (given the current set of beliefs) and then retrieves a plan to execute to achieve that goal. PRS iteratively tests the assumptions of the plan during its execution. This means that it can operate in dynamic environments where classical planners are doomed to fail.

5. Plans in PRS (also called knowledge areas) are predefined for the actions that are possible in the environment. This simplifies the architecture because it isn't required to generate plans, only select them based on the environment and the goals that must be met.

6. While planning is more about selection than search or generation, the interpreter ensures that changes to the environment do not result in inconsistencies in the plan. Instead, a new plan is selected to achieve the specific goals.

7. PRS is a useful architecture when all necessary operations can be predefined. It's also very efficient due to lack of plan generation. This makes PRS an ideal agent architecture for building agents such as those to control mobile robots.

➢ **AGLETS (MOBILE)**

1. Aglets is a mobile agent framework designed by IBM Tokyo in the 1990s. Aglets is based on the Java programming language, as it is well suited for a mobile agents framework. First, the applications are portable to any system (both homogeneous and

heterogeneous) that is capable of running a Java Virtual Machine (JVM). Second, a JVM is an ideal platform for migration services.

2. Java supports serialization, which is the aggregation of a Java application's program and data into a single object that is restart able.

3. In this case, the Java application is restarted on a new JVM. Java also provides a secure environment (sandbox) to ensure that a mobile agent framework doesn't become a virus distribution system. The Aglets framework is shown in Figure 4.9. At the bottom of the framework is the JVM (the virtual machine that interprets the Java byte codes). The agent runtime environment and mobility protocol are next. The mobility protocol, called Aglet Transport Protocol (or ATP), provides the means to serialize agents and then transport them to a host previously defined by the agent.

4. The agent API is at the top of the stack, which in usual Java fashion, provides a number of API classes that focus on agent operation. Finally, there are the various agents that operate on the framework.

5. The agent API and runtime environment provide a number of services that are central to a mobile agent framework. Some of the more important functions are agent management, communication, and security. Agents must be able to register themselves on a given host to enable communication from outside agents.

6. In order to support communication, security features must be implemented to ensure that the agent has the authority to execute on the framework.

7. Aglets provides a number of necessary characteristics for a mobile agent framework, including mobility, communication, security, and confidentiality. Aglets provide weak migration, in that the agents can only migrate at arbitrary points within the code (such as with the dispatch method).

➢ **MESSENGERS (MOBILE)**

1. Messengers is a runtime environment that provides a form of process migration (mobile agency).

2. One distinct strength of the messengers environment is that it supports strong migration, or the ability to migrate at arbitrary points within the mobile application.

3. The messengers environment provides the hop statement which defines when and where to migrate to a new destination.

4. After migration is complete, the messengers agent restarts in the application at the point after the previous hop statement. The end result is that the application moves to the data, rather than using a messaging protocol to move the data to the agent.

5. There are obvious advantages to this when the data set is large and the migration links are slow. The messengers model provides what the authors call Navigational Programming, and also Distributed Sequential Computing (DSC).

6. What makes these concepts interesting is that they support the common model of programming that is identical to the traditional flow of sequential programs. This makes them easier to develop and understand.

7. Let’s now look at an example of DSC using the messengers environment. Listing 11.5 provides a simple program. Consider an application where on a series of hosts, we manipulate large matrices which are held in their memory.

➢ **SOAR (HYBRID)**

1. Soar, which originally was an acronym for State-Operator-And-Result, is a symbolic cognitive architecture.

2. Soar provides a model of cognition along with an implementation of that model for building general-purpose AI systems.

3. The idea behind Soar is from Newell’s unified theories of cognition. Soar is one of the most widely used architectures, from research into aspects of human behavior to the design of game agents for first person- shooter games.

4. The goal of the Soar architecture is to build systems that embody general intelligence. While Soar includes many elements that support this goal (for example, representing knowledge using procedural, episodic, and declarative forms), but Soar lacks some important aspects. These include episodic memories and also a model for emotion. Soar’s underlying problem-solving mechanism is based on a production system (expert system).

5. Behavior is encoded in rules similar to the if-then form. Solving problems in Soar can be most simply described as problem space search (to a goal node). If this model of problem solving fails, other methods are used, such as hill climbing.

6. When a solution is found, Soar uses a method called chunking to learn a new rule based on this discovery. If the agent encounters the problem again, it can use the rule to select the action to take instead of performing problem solving again.

## 4.3 AGENT COMMUNICATION

In the domain of multi-agent systems, communication is an important characteristic to support both coordination and the transfer of information. Agents also require the ability to communicate actions or plans. But how the communication takes place is a function of its purpose.

1. Agents communicate in order to achieve better the goals of themselves or of the society/ system in which they exist.

2. Communication can enable the agents to coordinate their actions and behavior, resulting in systems that are more coherent.

3. Coordination is a property of a system of agents performing some activity in a shared environment.

4. The degree of coordination is the extent to which they avoid extraneous activity by reducing resource contention, avoiding live lock and deadlock, and maintaining applicable safety conditions.

5. Cooperation is coordination among non-antagonistic agents, while negotiation is coordination among competitive or simply self-interested agents.

6. Typically, to cooperate successfully, each agent must maintain a model of the other agents, and also develop a model of future interactions. This presupposes sociability Coherence is how well a system behaves as a unit. A problem for a multiagent system is how it can maintain global coherence without explicit global control. In this case, the agents must be able on their own to determine goals they share with other agents, determine common tasks, avoid unnecessary conflicts, and pool knowledge and evidence. It is helpful if there is some form of organization among the agents.

**Dimensions of Meaning**

There are three aspects to the formal study of communication: syntax (how the symbols of communication are structured), semantics (what the symbols denote), and pragmatics (how the symbols are interpreted). Meaning is a combination of semantics and pragmatics. Agents communicate in order to understand and be understood, so it is important to consider the different dimensions of meaning that are associated with communication.

1. Descriptive vs. Prescriptive. Some messages describe phenomena, while others prescribe behaviour. Descriptions are important for human comprehension, but are difficult for agents to mimic. Appropriately, then, most agent communication languages are designed for the exchange of information about activities and behaviour.

2. Personal vs. Conventional Meaning. An agent might have its own meaning for a message, but this might differ from the meaning conventionally accepted by the other agents with which the agent communicates. To the greatest extent possible, multiagent systems should opt for conventional meanings, especially since these systems are typically open environments in which new agents might be introduced at any time.

3. Subjective vs. Objective Meaning Similar to conventional meaning, where meaning is determined external to an agent, a message often has an explicit effect on the

environment, which can be perceived objectively. The effect might be different than that understood internally, i.e., subjectively, by the sender or receiver of the message.

4. Speaker's vs. Hearer's vs. Society's Perspective Independent of the conventional or objective meaning of a message, the message can be expressed according to the viewpoint of the speaker or hearer or other observers.

5. Semantics vs. Pragmatics The pragmatics of a communication are concerned with how the communicators use the communication. This includes considerations of the mental states of the communicators and the environment in which they exist, considerations that are external to the syntax and semantics of the communication.

6. Contextuality Messages cannot be understood in isolation, but must be interpreted in terms of the mental states of the agents, the present state of the environment, and the environment's history: how it arrived at its present state. Interpretations are directly affected by previous messages and actions of the agents.

7. Coverage Smaller languages are more manageable, but they must be large enough so that an agent can convey the meanings it intends.

8. Identity When a communication occurs among agents, its meaning is dependent on the identities and roles of the agents involved, and on how the involved agents are specified. A message might be sent to a particular agent, or to just any agent satisfying a specified criterion.

9. Cardinality A message sent privately to one agent would be understood differently than the same message broadcast publicly.

## 4.4 MESSAGE TYPES

1. It is important for agents of different capabilities to be able to communicate. Communication must therefore be defined at several levels, with communication at the lowest level used for communication with the least capable agent.

2. In order to be of interest to each other, the agents must be able to participate in a dialogue. Their role in this dialogue may be either active, passive, or both, allowing them to function as a master, slave, or peer, respectively.

3. In keeping with the above definition for and assumptions about an agent, we assume that an agent can send and receive messages through a communication network.

4. The messages can be of several types, as defined next.

5. There are two basic message types: assertions and queries. Every agent, whether active or passive, must have the ability to accept information. In its simplest form, this information is communicated to the agent from an external source by means of an assertion. In order to assume a passive role in a dialog, an agent must additionally be

able to answer questions, i.e., it must be able to 1) accept a query from an external source and 2) send a reply to the source by making an assertion. Note that from the standpoint of the communication network, there is no distinction between an unsolicited assertion and an assertion made in reply to a query.

6. In order to assume an active role in a dialog, an agent must be able to issue queries and make assertions. With these capabilities, the agent then can potentially control another agent by causing it to respond to the query or to accept the information asserted. This means of control can be extended to the control of subagents, such as neural networks and databases.

7. An agent functioning as a peer with another agent can assume both active and passive roles in a dialog. It must be able to make and accept both assertions and queries.

## 4.5 SPEECH ACTS

Spoken human communication is used as the model for communication among computational agents. A popular basis for analyzing human communication is speech act theory [1, 39]. Speech act theory views human natural language as actions, such as requests, suggestions, commitments, and replies. For example, when you request something, you are not simply making a statement, but creating the request itself. When a jury declares a defendant guilty, there is an action taken: the defendant's social status is changed.

A speech act has three aspects:

1. Locution, the physical utterance by the speaker

2. Illocution, the intended meaning of the utterance by the speaker

3. Perlocution, the action that results from the locution. KQML (Knowledge Query and Manipulation Language)

1. The KQML is an interesting example of communication from a number of facets. For example, communication requires the ability to locate and engage a peer in a conversation (communication layer).

2. A method for packaging the messages is then necessary (messaging layer), and finally an internal format that represents the messages and is sufficiently expressive to convey not only information but requests, responses, and plans (content layer).

3. In a network of KQML-speaking agents, there exists programs to support communication. These consist of facilitators that can serve as name servers to KQML components, and help find other agents that can satisfy a given agent’s request.

4. A KQML router supports the routing of messages and is a front-end to a specific KQML agent. As KQML was originally written in Common LISP, it’s message representation follows the LISP example (balanced parentheses).

5. A KQML message can be transferred to any particular transport (such as sockets) and has a format that consists of a performative and a set of arguments for that performative. The performative defines the speech act which defines the purpose of the message (assertion, command, request, etc.).

6. The performative- name defines the particular message type to be communicated (evaluate, ask- if, stream-about, reply, tell, deny, standby, advertise, etc.). The sender and receiver define the unique names of the agents in the dialogue. The content is information specific to the performative being performed.

7. This content is defined in a language (how to represent the content), and an ontology that describes the vocabulary (and meaning) of the content. Finally, the agent can attach a context which the response will contain (in-reply-to) in order to correlate the request with the response. The structure of a KQML message.

```
(performative-name
: sender X
: receiver Y
: content Z
: language L
: ontology Y
: reply-with R
: in-reply-to Q
)
```

Let's now look at an example conversation between two KQML agents. In this example, an agent requests the current value of a temperature sensor in a system. The request is for the temperature of TEMP_SENSOR_1A that's sampled at the temperature-server agent. The content is the request, defined in the prolog language. Our agent making the request is called thermal-control-appl.

```
(ask-one
:sender thermal-control-appl
:receiver temperature-server
:languageprolog
:ontology CELSIUS-DEGREES
:content "temperature(TEMP_SENSOR_1A ?temperature)"
:reply-with request-102
)
```

Our agent would then receive a response from the temperature-server, defining the temperature of the sensor of interest.

```
(reply
:sender temperature-server
:receiver thermal-control-appl
:languageprolog
:ontology CELSIUS-DEGREES
:content “temperature(TEMP_SENSOR_1A 45.2)”
:in-reply-to request-102
)
```

Our agent would then receive a response from the temperature-server, defining the temperature of the sensor of interest,

```
(reply
:sender temperature-server
:receiver thermal-control-appl
:languageprolog
:ontology CELSIUS-DEGREES
:content “temperature(TEMP_SENSOR_1A 45.2_”
:in-reply-to request1102
)
```

KQML is very rich in its ability to communicate information as well higher-level request that address the communication layer. Table 4.1 provides a short list of some of the other KQML per formatives.

## 4.6 KQML PERFORMATIVES

**Table 4.1**

| **Performatives** | **Description** |
|---|---|
| Evaluate | Evaluate the content of the message |
| ask-one | Request for the answer to the question |
| reply | Communicate a reply to a question |
| stream-about | Provide multiple response to a question |
| sorry | Return an error (can’t respond) |
| tell | Inform an agent of sentence |
| achieve | A request of something to achieve by the receiver |
| advertise | Advertise the ability to process a performative |
| subscribe | Subscribe to changes of information |
| forward | Route a message |

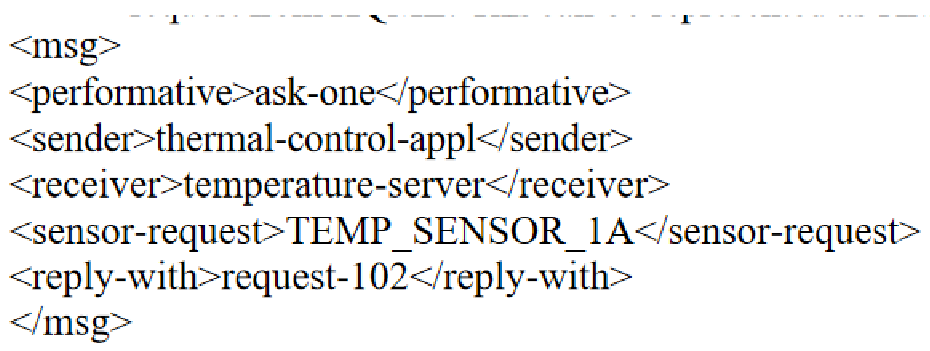
- KQML is a useful language to communicate not only data, but the meaning of the data (in terms of a language and ontology).
- KQML provides a rich set of capabilities that cover basic speed acts, and more complex acts including data streaming and control of information transfer.

## 4.7 ACL (FIPA AGENT COMMUNICATION LANGUAGE)

1. Where KQML is a language defined in the context of a university, the FIPA ACL is a consortium-based language for agent communication.
2. ACL simply means Agent Communication Language and it was standardized through the Foundation for Intelligent Physical Agents consortium. As with KQML, ACL is a speech- act language defined by a set of per formatives.
3. The FIPA, or Foundation for Intelligent Physical Agents, is a non-profit organization that promotes the development of agent-based systems. It develops specifications to maximize the portability of agent systems (including their ability to communicate using the ACL).
4. The FIPA ACL is very similar to the KQML, even adopting the inner and outer content layering for message construction (meaning and content).
5. The ACL also clarifies certain speech-acts, or performatives. For example, communication primitives are called communicative acts, which are separate from the performative acts.
6. The FIPA ACL also uses the Semantic Language, or SL, as the formal language to define ACL semantics. This provides the means to support BDI themes (beliefs, desires, intentions). In other words, SL allows the representation of persistent goals (intentions), as well as propositions and objects. Each agent language has its use, and while both have their differences, they can also be viewed as complementary.

**XML**

1. XML is the Extensible Markup Language and is an encoding that represents data and meta- data (meaning of the data). It does this with a representation that includes tags that encapsulate the data.
2. The tags explicitly define what the data represents. For example, consider the ask-one request from KQML. This can be represented as XML as shown below:



```
<msg>
<performative>ask-one</performative>
<sender>thermal-control-appl</sender>
<receiver>temperature-server</receiver>
<sensor-request>TEMP_SENSOR_1A</sensor-request>
<reply-with>request-102</reply-with>
</msg>
```

1. There are some obvious similarities to XML and KQML. In KQML, the tags exist, but use different syntax than is defined for XML. One significant difference is that KQML permits the layering of tags.

2. Note here that the <msg> tag is the outer layer of the performative and its arguments. XML is very flexible in its format and permits very complex arrangements of both data and meta-data.

3. XML is used in a number of protocols, including XML-RPC (Remote Procedure Call) and also SOAP (Simple Object Access Protocol). Each of these use the Hyper Text Transport Protocol (HTTP) as its transport.

**TRUST AND REPUTATION**

It depends on the level we apply it:

1. User confidence
- Can we trust the user behind the agent?
  - Is he/she a trustworthy source of some kind of knowledge? (e.g. an expert in a field)
  - Does he/she acts in the agent system (through his agents in a trustworthy way?

2. Trust of users in agents
- Issues of autonomy: the more autonomy, less trust
- How to create trust?
  - Reliability testing for agents
  - Formal methods for open MAS
  - Security and verifiability

3. Trust of agents in agents
- Reputation mechanisms
- Contracts
- Norms and Social Structures

What is Trust?

1. In closed environments, cooperation among agents is included as part of the designing process:

2. The multi- agent system is usually built by a single developer or a single team of developers and the chosen developers, option to reduce complexity is to ensure cooperation among the agents they build including it as an important system requirement.

3. Benevolence assumption: an agent AI requesting information or a certain service from agent aj can be sure that such agent will answer him if AI has the capabilities and the resources needed, otherwise aj will inform AI that it cannot perform the action requested.

4. It can be said that in closed environments trust is implicit.

Trust can be computed as

1. A binary value (1='I do trust this agent', 0='I don't trust this agent')

2. A set of qualitative values or a discrete set of numerical values (e g 'trust always' 'trust conditional to X' 'no trust') e.g. always, X, trust ) (e.g. '2', '1', '0', '-1', '-2')

3. A continuous numerical value (e.g. [-300..300])

4. A probability distribution

5. Degrees over underlying beliefs and intentions (cognitive approach)

**HOW TO COMPUTE TRUST**

1. Trust values can be externally defined

- by the system designer: the trust values are pre-defined
- by the human user: he can introduce his trust values about the humans behind the other agents

2. Trust values can be inferred from some existing representation about the interrelations between the agents

- Communication patterns, cooperation history logs, e-mails, webpage connectivity mapping...

3. Trust values can be learnt from current and past experiences

- Increase trust value for agent AI if behaves properly with us
- Decrease trust value for agent AI if it fails/defects us

4. Trust values can be propagated or shared through a MAS

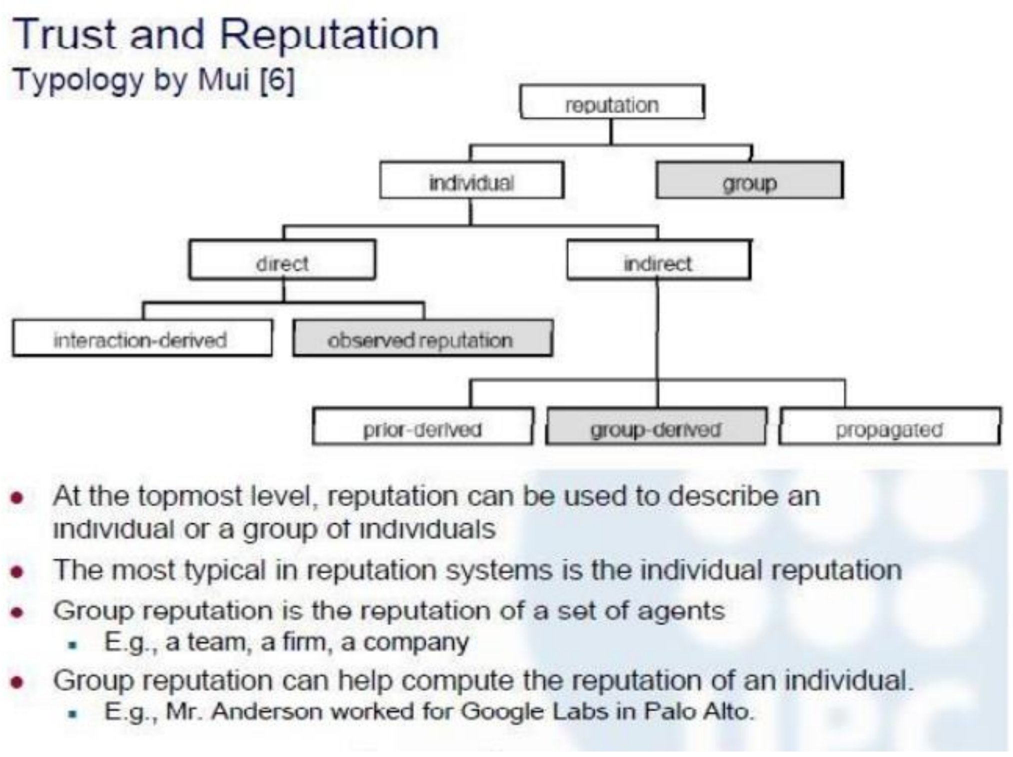
- Recommender systems, Reputation mechanisms.

**TRUST AND REPUTATION**

1. Most authors in literature make a mix between trust and reputation
2. Some authors make a distinction between them
3. Trust is an individual measure of confidence that a given agent has over other agent(s)
4. Reputation is a social measure of confidence that a group of agents or a society has over agents or groups. Reputation is one mechanism to compute (individual) Trust

- I will trust more an agent that has good reputation
- My reputation clearly affects the amount of trust that others have towards me.
- Reputation can have a sanctioning role in social groups: a bad reputation can be very costly to one's future transactions.

5. Most authors combine (individual) Trust with some form of (social) Reputation in their models
6. Recommender systems, Reputation mechanisms.



## Trust and Reputation
### Typology by Mui [6]

- At the topmost level, reputation can be used to describe an individual or a group of individuals
- The most typical in reputation systems is the individual reputation
- Group reputation is the reputation of a set of agents
  - E.g., a team, a firm, a company
- Group reputation can help compute the reputation of an individual.
  - E.g., Mr. Anderson worked for Google Labs in Palo Alto.

**Figure 4.10 TRUST AND REPUTATION**

Direct experiences are the most relevant and reliable information source for individual trust/reputation

1. Type 1: Experience based on direct interaction with the
2. Type 1: Experience with the partner
1. Used by almost all models
2. How to:
• trust value about that partner increases with good experiences,
• it decreases with bad ones
3. Problem: how to compute trust if there is no previous interaction?
3. Type 2: Experience based on observed interaction of other members
1. Used only in scenarios prepared for this.
2. How to: depends on what an agent can observe
• agents can access to the log of past interactions of other agents
• agents can access some feedback from agents about their past interactions (e.g., in eBay)
3. Problem: one has to introduce some noise handling or
4. confidence level on this information
4. Prior-derived: agents bring with them prior beliefs about strangers

   Used by some models to initialize trust/reputation values

   How-to:

   • designer or human user assigns prior values

   • a uniform distribution for reputation priors is set

Give new agents the lowest possible reputation value there is no incentive to throw away a cyber-identity when an agent's reputation falls below a starting point.

• Assume neither good nor bad reputation for unknown agents.
• Avoid lowest reputation for new, valid agents as an obstacle for other agents to realize that they are valid.
5. Group-derived:
• Models for groups can been extended to provide prior reputation estimates for agents in social groups.
• Mapping between the initial individual reputation of a stranger and the group from which he or she comes from.
• Problem: highly domain-dependent and model-dependent.
6. Propagated:
• Agent can attempt to estimate the stranger's reputation based on information garnered from others in the environment Also called word of mouth.

- Problem: The combination of the different reputation values tends to be an ad-hoc solution with no social basis

**TRUST AND REPUTATION MODELS**

1. Not really for MAS, but can be applied to MAS
2. Idea: For serious life / business decisions, you want the
- opinion of a trusted e pert trusted expert
3. If an expert not personally known, then want to find a reference to one via a chain of friends and colleagues
4. Referral-chain provides:
- Way to judge quality of expert's advice
- Reason for the expert to respond in a trustworthy manner
- Finding good referral-chains is slow, time-consuming, but vital business gurus on "networking"
- Set of all possible referral-chains = a social network
5. Model integrates information from
- Official organizational charts (online)
- Personal web pages (+ crawling)
- External publication databases
- Internal technical document databases
6. Builds a social network based in referral chains
- Each node is a recommender agent
- Each node provides reputation values for specific areas
  - o E.g. Frieze is good in mathematics
- Searches in the referral network are
- made by areas

o E.g. browsing the network's "mathematics" recommendation chains

7. Trust Model Overview
- 1-to-1 asymmetric trust relationships.
- Direct trust and recommender trust.
- Trust categories and trust values
- [-1,0,1,2,3,4].
8. Conditional transitivity.

Alice trusts Bob.&. Bob trusts Cathy

Alice trusts Cathy

Alice trusts.rec Bob.&. Bob says Bob trusts Cathy

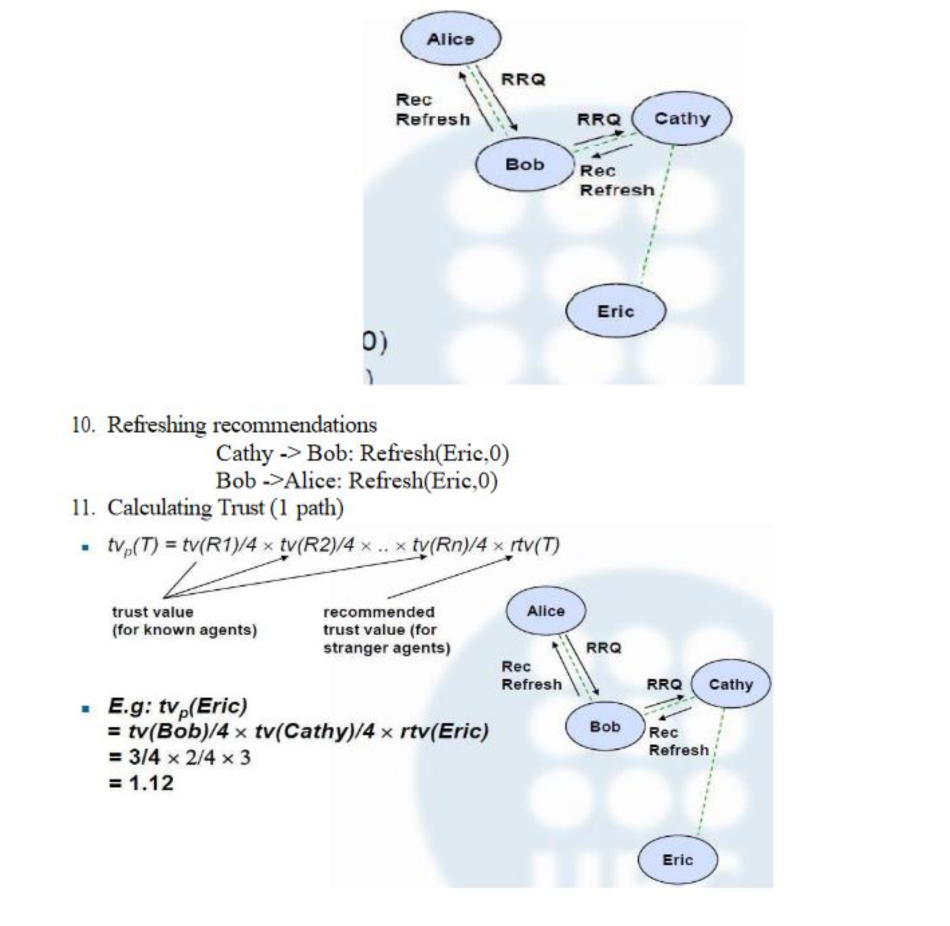
Alice may trust Cathy

Alice trusts.rec Bob value X. &. Bob says Bob trusts Cathy value Y

Alice may trust Cathy value f(X, Y)

9. Recommendation protocol

1. Alice ->Bob: RRQ(Eric)
2. Bob ->Cathy: RRQ(Eric)
3. Cathy -> Bob: Rec(Eric,3)
4. Bob ->Alice: Rec(Eric,3)



10. Refreshing recommendations

Cathy -> Bob: Refresh(Eric,0)

Bob ->Alice: Refresh(Eric,0)

11. Calculating Trust (1 path)

- $tv_p(T) = tv(R1)/4 \times tv(R2)/4 \times .. \times tv(Rn)/4 \times rtv(T)$
- **E.g: $tv_p(Eric)$**
  **$= tv(Bob)/4 \times tv(Cathy)/4 \times rtv(Eric)$**
  **$= 3/4 \times 2/4 \times 3$**
  **$= 1.12$**

Figure 4.11 Procedure

12. Direct Trust:

■ ReGreT assumes that there is no difference between direct interaction and direct observation in terms of reliability of the information. It talks about direct experiences.

■ The basic element to calculate a direct trust is the outcome.

■ An outcome of a dialog between two agents can be either:

• An initial contract to take a particular course of action and the actual result of the actions taken, or

• An initial contract to x the terms and conditions of a transaction and the actual values of the terms of the transaction.

13. Reputation Model: Witness reputation

a. First step to calculate a witness reputation is to identify the set of witnesses that will be taken into account by the agent to perform the calculation.

b. The initial set of potential witnesses might be

i. the set of all agents that have interacted with the target agent in the past.

ii. This set, however, can be very big and the information provided by its members probably suffer from the correlated evidence problem.

c. Next step is to aggregate these values to obtain a single value for the witness reputation. The importance of each piece of information in the final reputation value will be proportional to the witness credibility

14. Reputation Model: Witness reputation

a. Two methods to evaluate witness credibility:

i. ReGreT uses fuzzy rules to calculate how the structure of social relations influences the credibility on the information. The antecedent of each rule is the type and degree of a social relation (the edges in a sociogram) and the consequent is the credibility of the witness from the point of view of that social relation.

ii The second method used in the ReGreT system to calculate the credibility of a witness is to evaluate the accuracy of previous pieces of information sent by that witness to the agent. The agent is using the direct trust value to measure the truthfulness of the information received from witnesses.

15. Reputation Model: Neighbourhood Reputation

a. Neighbourhood in a MAS is not related with the physical location of the agents but with the links created through interaction.

b. The main idea is that the behaviour of these neighbours and the kind of relation they have with the target agent can give some clues about the behaviour of the target agent.

c. To calculate a Neighbourhood Reputation the ReGreT system uses fuzzy rules.

i. The antecedents of these rules are one or several direct trusts associated to different behavioural aspects and the relation between the target agent and the neighbour.

ii. The consequent is the value for a concrete reputation (that can be associated to the same behavioural aspect of the trust values or not).

16. Reputation Model: System Reputation

a. To use the common knowledge about social groups and the role that the agent is playing in the society as a mechanism to assign default reputations to the agents.

b. ReGreT assumes that the members of these groups have one or several observable features that unambiguously identify their membership.

c. Each time an agent performs an action we consider that it is playing a single role.

i. E.g. an agent can play the role of buyer and seller but when it is selling a product only the role of seller is relevant.

17. System reputations are calculated using a table for each social group where the rows are the roles the agent can play for that group, and the columns the behavioural aspects.

18. Reputation Model: Default Reputation

a. To the previous reputation types we have to add a fourth one, the reputation assigned to a third party agent when there is no information at all: the default reputation.

b. Usually this will be a fixed value

19. Reputation Model: Combining reputations

a. Each reputation type has different characteristics and there are a lot of heuristics that can be used to aggregate the four reputation values to obtain a single and representative reputation value.

b. In ReGreT this heuristic is based on the default and calculated reliability assigned to each type.

c. Assuming we have enough information to calculate all the reputation types, we have the stance that

a. witness reputation is the first type that should be considered, followed by

b. the neighbourhood reputation,

c. system reputation

d. the default reputation.

20. Main criticism to Trust and Reputation research:

a. Proliferation of ad-hoc models weakly grounded in social theory

b. No general, cross-domain model for reputation

c. Lack of integration between models

i. Comparison between models unfeasible

ii. Researchers are trying to solve this by, e.g. the ART competition

## NEGOTIATION

1. A frequent form of interaction that occurs among agents with different goals is termed negotiation.

2. Negotiation is a process by which a joint decision is reached by two or more agents, each trying to reach an individual goal or objective. The agents first communicate their positions, which might conflict, and then try to move towards agreement by making concessions or searching for alternatives.

3. The major features of negotiation are (1) the language used by the participating agents, (2) the protocol followed by the agents as they negotiate, and (3) the decision process that each agent uses to determine its positions, concessions, and criteria for agreement.

4. Many groups have developed systems and techniques for negotiation. These can be either environment-centered or agent-centered. Developers of environment-centered techniques focus on the following problem: "How can the rules of the environment be

designed so that the agents in it, regardless of their origin, capabilities, or intentions, will interact productively and fairly?"

The resultant negotiation mechanism should ideally have the following attributes:

- Efficiency: the agents should not waste resources in coming to an agreement. Stability: no agent should have an incentive to deviate from agreed- upon strategies.
- Simplicity: the negotiation mechanism should impose low computational and bandwidth demands on the agents.
- Distribution: the mechanism should not require a central decision maker.
- Symmetry: the mechanism should not be biased against any agent for arbitrary or inappropriate reasons.

5. An articulate and entertaining treatment of these concepts is found in [36]. In particular, three types of environments have been identified: worth-oriented domains, state-oriented domains, and task-oriented domains.

6. A task-oriented domain is one where agents have a set of tasks to achieve, all resources needed to achieve the tasks are available, and the agents can achieve the tasks without help or interference from each other. However, the agents can benefit by sharing some of the tasks. An example is the "Internet downloading domain," where each agent is given a list of documents that it must access over the Internet. There is a cost associated with downloading, which each agent would like to minimize. If a document is common to several agents, then they can save downloading cost by accessing the document once and then sharing it.

7. The environment might provide the following simple negotiation mechanism and constraints:
   (1) each agent declares the documents it wants
   (2) documents found to be common to two or more agents are assigned to agents based on the toss of a coin,
   (3) agents pay for the documents they download, and
   (4) agents are granted access to the documents they download. as well as any in their common sets. This mechanism is simple, symmetric, distributed, and efficient (no document is downloaded twice). To determine stability, the agents' strategies must be considered.

8. An optimal strategy is for an agent to declare the true set of documents that it needs, regardless of what strategy the other agents adopt or the documents they need. Because there is no incentive for an agent to diverge from this strategy, it is stable.

9. For the first approach, speech-act classifiers together with a possible world semantics are used to formalize negotiation protocols and their components. This clarifies the conditions of satisfaction for different kinds of messages. To provide a flavor of this approach, we show in the following example how the commitments that an agent might make as part of a negotiation are formalized [21]:

$$\begin{aligned}&\forall x(x \neq y) \wedge \\ &\neg(\mathit{Precommit}_a\ y\ x\ \phi) \wedge (\mathit{Goal}\ y\ \mathit{Eventually}(\mathit{Achieves}\ y\ \phi)) \wedge (\mathit{Willing}\ y\ \phi) \\ &\iff (\mathit{Intend}\ y\ \mathit{Eventually}(\mathit{Achieves}\ y\ \phi))\end{aligned}$$

10. This rule states that an agent forms and maintains its commitment to achieve ø individually iff (1) it has not precommitted itself to another agent to adopt and achieve ø, (2) it has a goal to achieve ø individually, and (3) it is willing to achieve ø individually. The chapter on "Formal Methods in DAI" provides more information on such descriptions.

11. The second approach is based on an assumption that the agents are economically rational. Further, the set of agents must be small, they must have a common language and common problem abstraction, and they must reach a common solution. Under these assumptions, Rosenschein and Zlotkin [37] developed a unified negotiation protocol. Agents that follow this protocol create a deal, that is, a joint plan between the agents that would satisfy all of their goals. The utility of a deal for an agent is the amount he is willing to pay minus the cost of the deal. Each agent wants to maximize its own utility.

The agents discuss a negotiation set, which is the set of all deals that have a positive utility for every agent.

In formal terms, a task-oriented domain under this approach becomes a tuple <T, A, c>

where T is the set of tasks, A is the set of agents, and c(X) is a monotonic function for the cost of executing the tasks X. A deal is a redistribution of tasks. The utility of deal d for agent k is Uk(d) = c(Tk) - c(dk)

The conflict deal D occurs when the agents cannot reach a deal. A deal d is individually rational if d > D. Deal d is pareto optimal if there is no deal d' > d. The set of all deals that are individually rational and pareto optimal is the negotiation set, NS. There are three possible situations:

1. conflict: the negotiation set is empty

2. compromise: agents prefer to be alone, but since they are not, they will agree to a negotiated deal

3. cooperative: all deals in the negotiation set are preferred by both agents over achieving their goals alone.

When there is a conflict, then the agents will not benefit by negotiating—they are better off acting alone. Alternatively, they can "flip a coin" to decide which agent gets to satisfy its goals.

Negotiation is the best alternative in the other two cases.

Since the agents have some execution autonomy, they can in principle deceive or mislead each other. Therefore, an interesting research problem is to develop protocols or societies in which the effects of deception and misinformation can be constrained. Another aspect of the research problem is to develop protocols under which it is rational for agents to be honest with each other. The connections of the economic approaches with human-oriented negotiation and argumentation have not yet been fully worked out.

## 4.7 BARGAINING

Link: http://www.cse.iitd.ernet.in/~rahul/cs905/lecture15/index.html (Refer this link for easy understanding) A bargaining problem is defined as a pair (S,d). A bargaining solution is a function f that maps every bargaining problem (S,d) to an outcome in S, i.e.,

$f : (S,d) \rightarrow S$

Thus the solution to a bargaining problem is a pair in R2. It gives the values of the game to the two players and is generated through the function called bargaining function. Bargaining function maps the set of possible outcomes to the set of acceptable ones.

**Bargaining Solution**

In a transaction when the seller and the buyer value a product differently, a surplus is created. A bargaining solution is then a way in which buyers and sellers agree to divide the surplus. For example, consider a house made by a builder A. It costed him Rs.10 Lacs. A potential buyer is interested in the house and values it at Rs.20 Lacs. This transaction can generate a surplus of Rs.10 Lacs. The builder and the buyer now need to trade at a price. The buyer knows that the cost is less than 20 Lacs and the seller knows that the value is greater than 10 Lacs. The two of them need to agree at a price. Both try to maximize their surplus. Buyer would want to buy it for 10 Lacs, while the seller would like to sell it for 20 Lacs. They bargain on the price, and either trade or dismiss. Trade would result in the generation of surplus, whereas no surplus is created in case of no-trade. Bargaining Solution provides an acceptable way to divide the surplus among the two parties. Formally, a Bargaining Solution is defined as, $F : (X,d) \rightarrow S$,

where $X \subseteq R2$ and $S,d \in R2$. X represents the utilities of the players in the set of possible bargaining agreements. d represents the point of disagreement. In the above example, price $\in$

[10,20], bargaining set is simply x + y ≤ 10, x ≥ 0, y ≥ 0. A point (x,y) in the bargaining set represents the case, when seller gets a surplus of x, and buyer gets a surplus of y, i.e. seller sells the house at 10 + x and the buyer pays 20 − y.

1. the set of payoff allocations that are jointly feasible for the two players in the process of negotiation or arbitration, and

2. the payoffs they would expect if negotiation or arbitration were to fail to reach a settlement.

Based on these assumptions, Nash generated a list of axioms that a reasonable solution ought to satisfy. These axioms are as follows:

Axiom 1 (Individual Rationality) This axiom asserts that the bargaining solution should give neither player less than what it would get from disagree ment, i.e., f(S,d) ≥ d.

Axiom 2 (Symmetry) As per this axiom, the solution should be independent of the names of the players, i.e., who is named a and who is named b. This means that when the players' utility functions and their disagreement utilities are the same, they receive equal shares. So any symmetries in the final payoff should only be due to the differences in their utility functions or their disagreement outcomes.

Axiom 3 (Strong Efficiency) This axiom asserts that the bargaining solution should be feasible and Pareto optimal.

Axiom 4 (Invariance) According to this axiom, the solution should not change as a result of linear changes to the utility of either player. So, for example, if a player's utility function is multiplied by 2, this should not change the solution. Only the player will value what it gets twice as much.

Axiom 5 (Independence of Irrelevant Alternatives) This axiom asserts that eliminating feasible alternatives (other than the disagreement point) that would not have been chosen should not affect the solution, i.e., for any closed convex set

Nash proved that the bargaining solution that satisfies the above five axioms is given by: $f(S,d) \in \underset{x \in S, x \geq d}{\operatorname{argmax}} \; (x^a - d^a)(x^b - d^b)$

## NON-COOPERATIVE MODELS OF SINGLE-ISSUE NEGOTATION

**Table 4.2**

| Cooperative Game | Non Cooperative Game |
|---|---|

| | |
|---|---|
| The players are allowed to communicate before choosing their strategies and playing the game. | Each player independently chooses its strategy. |
| The basic modeling unit is the group | The basic modeling unit is the individual. |
| Players can enforce cooperation in the group through a third party. | The cooperation between individuals is self-enforcing. |

## 4.8 GAME-THEORETIC APPROACHES FOR MULTI-ISSUE NEGOTIATION

The following are the four key procedures for bargaining over multiple issue:

1. Global bargaining: Here, the bargaining agents directly tackle the global problem in which all the issues are addressed at once. In the context of non-cooperative theory, the global bargaining procedure is also called the package deal procedure. In this procedure, an offer from one agent to the other would specify how each one of the issues is to be resolved.

2. Independent/separate bargaining: Here negotiations over the individual issues are totally separate and independent, with each having no effect on the other. This would be the case if each of the two parties employed m agents (for negotiating over m issues), with each agent in charge of negotiating one issue. For example, in negotiations between two countries, each issue may be resolved by representatives from the countries who care only about their individual issue.

3. Sequential bargaining with independent implementation: Here the two parties consider one issue at a time. For instance, they may negotiate over the first issue, and after reaching an agreement on it, move on to negotiate the second, and so on. Here, the parties may not negotiate an issue until the previous one is resolved. There are several forms of the sequential procedure. These are defined in terms of the agenda and the implementation rule. For sequential bargaining, the agenda3 specifies the order in which the issues will be bargained. The implementation rule specifies when an agreement on an individual issue goes into effect. There are two implementation rules: the rule of independent implementation and the rule of simultaneous implementation.

4. Sequential bargaining with simultaneous implementation: This is similar to the previous case except that now an agreement on an issue does not take effect until an agreement is reached on all the subsequent issues.

### Cooperative Models of Multi-Issue Negotiation

1. Simultaneous implementation agenda independence: This axiom states that global bargaining and sequential bargaining with simultaneous implementation yield the same agreement.

2. Independent implementation agenda independence: This axiom states that global bargaining and sequential bargaining with independent implementation yield the same agreement.

3. Separate/global equivalence: This axiom states that global bargaining and separate bargaining yield the same agreement.

**Non-Cooperative Models of Multi-Issue Negotiation**

An agent's cumulative utility is linear and additive. The functions $U^a$ and $U_b$ give the cumulative utilities for a and b respectively at time t and are defined as follows.

$$U^a((x^a,x^b),t) = \begin{cases} \Sigma_{c=1}^{m} w_c^a \delta^{t-1} x_c^a & \text{if } t \leq n \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$U^b((x^a,x^b),t) = \begin{cases} \Sigma_{c=1}^{m} w_c^b \delta^{t-1} x_c^b & \text{if } t \leq n \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

Where $w^a \in R^+_m$ denotes an *m* element vector of constants for agent a and $w^b \in R^+_m$ such a vector b. These vectors indicate how the agents prefer different issuers. For example, if $w^a_c > w^a_{c+1}$, then agent a values issue c more than issue c+ 1. Like for agent b.

## 4.9 ARGUMENTATION

➢ "A verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener or reader, by putting forward a constellation of propositions (i.e. arguments) intended to justify (or refute) the standpoint before a rational judge"

➢ Argumentation can be defined as an activity aimed at convincing of the acceptability of a standpoint by putting forward propositions justifying or refuting the standpoint.

➢ Argument: Reasons / justifications supporting a conclusion

➢ Represented as: support ->conclusion

– Informational arguments: Beliefs -> Belief e.g. If it is cloudy, it might rain.

– Motivational args: Beliefs, Desires ->Desire e.g. If it is cloudy and you want to get out then you don't want to get wet.

– Practical arguments: Belief, Sub-Goals -> Goal e.g. If it is cloudy and you own a raincoat then put the raincoat.

– Social arguments: Social commitments-> Goal, Desire e.g. I will stop at the corner because the law say so. e.g I can't do that, I promise to my mother that won't.

**Process of Argumentation**

1. Constructing arguments (in favor of / against a "statement") from available information.

A: "Tweety is a bird, so it flies"

B: "Tweety is just a cartoon!"

2. Determining the different conflicts among the arguments.

   "Since Tweety is a cartoon, it cannot fly!" (B attacks A)

   Evaluating the acceptability of the different arguments

   "Since we have no reason to believe otherwise, we'll assume Tweety is a cartoon." (accept B). "But then, this means despite being a bird he cannot fly." (reject A).

3. Concluding, or defining the justified conclusions.

   "We conclude that Tweety cannot fly!"

   Computational Models of Argumentation:

1. Given the definition of arguments over a content language (and its logic), the models allow to:

• Compute interactions between arguments: attacks, defeat, support,...

• Valuation of arguments: assign weights to arguments in order to compare them. Intrinsic value of an argument Interaction-based value of an argument

2. Selection of acceptable argument (conclusion)

• Individual acceptability

• Collective acceptability

## 4.10 OTHER ARCHITECTURES

### LAYERED ARCHITECTURES

Given the requirement that an agent be capable of reactive and pro-active behavior, an obvious decomposition involves creating separate subsystems to deal with these different types of behaviors. This idea leads naturally to a class of architectures in which the various subsystems are arranged into a hierarchy of interacting layers. In this section, we will consider some general aspects of layered architectures, and then go on to consider two examples of such architectures:

### INTERRAP and TOURINGMACHINES

Typically, there will be at least two layers, to deal with reactive and pro-active behaviors respectively. In principle, there is no reason why there should not be many more layers. However many layers there are, a useful typology for such architectures is by the information and control flows within them. Broadly speaking, we can identify two types of control flow within layered architectures (see Figure):

- Horizontal layering. In horizontally layered architectures (Figure (a)), the software layers are each directly connected to the sensory input and action output. In effect, each layer itself acts like an agent, producing suggestions as to what action to perform.
- Vertical layering. In vertically layered architectures (Figure (b) and (c)), sensory input and action output are each dealt with by at most one layer each. The great advantage of horizontally layered architectures is their conceptual simplicity: if we need an agent to exhibit n different types of behavior, then we implement n different layers.

However, because the layers are each in effect competing with one-another to generate action suggestions, there is a danger that the overall behavior of the agent will not be coherent. In order to ensure that horizontally layered architectures are consistent, they generally include a mediator function, which makes decisions about which layer has "control" of the agent at any given time.

The need for such central control is problematic: it means that the designer must potentially consider all possible interactions between layers. If there are n layers in the architecture, and each layer is capable of suggesting m possible actions, then this means there are mn such interactions to be considered. This is clearly difficult from a design point of view in any but the simplest system. The introduction of a central control system also introduces a bottleneck into the agent's decision making

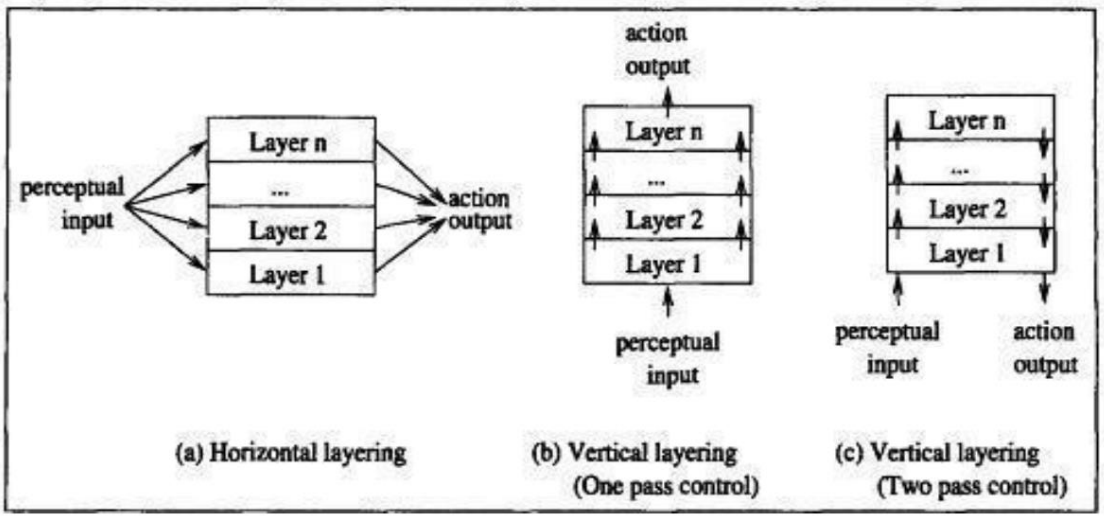


**Figure 4.12 INTERRAP and TOURINGMACHINES**

## ABSTRACT ARCHITECTURE

1. We can easily formalize the abstract view of agentgs presented so far. First, we will assume that the state of the agent’s environment can be characterized as a set $S = \{s_1, s_2, \ldots)$ of environment states.

2. At any given instant, the environment is assumed to be in one of these states. The effectoric capability of an agent is assumed to be represented by a set A = (a1, a2,…) of actions. Then abstractly, an agent can be viewed as a function

$$\text{Action } S^* \rightarrow A$$

3. Which maps sequences of enviornment states to actions. We will refer to an agent modelled by a function of this form as a standard agent. The intuition is that an agent decides what action to perform on the basis of its history – its experiences to date. These experiences are represented as a sequence of environment states – those that the agent has thus for encountered.

4. The (non-determinstic) behaviour of an environment can be modelled as a function env:

   $S \times A \rightarrow P(S)$

   Which takes the current state of the environment $s \in S$ and an action (perfrmed by the agent), and maps them to a set of environment state env(s,a) – those that could result from performing action a in state s. If all the sets in the rnage of env are all sigletons, (i.e., if the result of performing any action in any state in a set containing a single member), then the environment is deterministic, and its behaviour can be accrately predicted.

5. We can represented the interaction of agent and environment as a history. A history h is a sequence: h:: $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \ldots$

   where so is the initial state of the environment (i.e., its state when the agent starts executing), $a_u$ is the uth action that the agent chose to perform, and $s_u$ is the uth environment state (which is one of the possible results of executing action $a_{u-2}$ in state $s_{u-1}$). $S^* \rightarrow A$ is an agent, env: $S \times S \rightarrow p(S)$ is an environment, and so is the initial state of environment.

6. The characteristic behaviour of an agent action $S^* \rightarrow A$ is an envioronment env : is the set of all the histories that satisfy these properties. If some property $\phi$ holds of all these histories, this property can be regarded as an invariant property of the agent in the environment. For example, if our agent is a nuclear reactor controller, (i.e., the environment is a nuclear reactr), and in all possible histories of the contrller/reactor, the reactor does not below up, then this can be regarded as a (desirable) invariant property. We will denote by *hist(agent, environment)* the set of all histories of *agent in environment.* Two agentws *ag1* and *ag2* are said to be *behaviorally equivalent* with respect to enviroment *env* iff *hist(ag1, env) = hist(ag2, env)* and simply behaviorally equivalent iffr they are behaviorally equivalent with respect of all environments.

## 4.11 CONCRETE ARCHITECTURES FOR INTELLIGENT AGENTS

1. We have considered agents only in the abstract. So, while we have examined the properties of agents that do and do not maintain state, we have not stopped to consider what this state might look like. Similarly, we have modelled an agent's decision making as an abstract function *action*, which somehow manages to indicate which action to perform—but we have not discussed how this function might be implemented. In this section, we will rectify this omission. We will consider four classes of agents:

- ***logic based agents***—in which decision making is realized through logical deduction;
- ***reactive agents***—in which decision making is implemented in some form of direct mapping from situation to action;
- ***belief-desire-intention agents***—in which decision making depends upon the manipulation of data structures representing the beliefs, desires, and intentions of the agent; and finally,
- ***layered architectures***—in which decision making is realized via various software layers, each of which is more-or- less explicitly reasoning about the environment at different levels of abstraction.

In each of these cases, we are moving away from the abstract view of agents, and beginning to make quite specific commitments about the internal structure and operation of agents. Each section explains the nature of these commitments, the assumptions upon which the architectures depend, and the relative advantages and disadvantages of each.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

## SCHOOL OF ELECTRICAL AND ELECTRONICS

### DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINERING

# UNIT- V- ARTIFICIAL INTELLIGENCE- SECA3011

# UNIT 5

# APPLICATIONS

AI applications – Language Models – Information Retrieval- Information Extraction – Natural Language Processing – Machine Translation – Speech Recognition.

## 5.1 LANGUAGE MODELS

Language can be defined as a set of strings; “print(2+2)” is a legal program in the language Python, where “2) + (2 print” is not. Since the are an infinite number of legal programs, they cannot be enumerated; instead they are specified by a set of rules called a **grammar.** Formal languages also have rules that defined the meaning **semantics** of a program; for example, the rules say that the “meaning” of “2 + 2” is 4, and the meaning of “1/0” is that an error is signated.

1. Natural languages, such an English or Spanish, cannot be characterized as a definite set of sentences. **Example:** Everyone agrees that “Not to be invited is sad” is a sentence of English, but people disagree on the grammatically of “To be not invited is said”. Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set. That is, rather than asking if a string of words is or is not a member of the set defining the language, we instead ask for P(S = word) - what is the probability that a random sentence would to words. Natural languages are also **ambiguous**. “He saw her duck” can mean either that he saw a waterfowl belonging to her, or that he saw her move to evade something. Thus, again, we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings.

2. Finally, natural language are difficult to deal with because they are very large, and constantly changing. Thus, our language models are, at best, an approximation. We start with simplest possible approximation and move up from there.

## 5.2 N-GRAM CHARACTER MODELS

1. An n-gram model is defined as a **Markov chain** of order n – 1. In Markov chain the probability of character ci depends only on the immediately preceding characters, not on any other characters. So in a trigram model (Markov chain of order 2) we have

   $P(c_4 \mid c_{1:i-1}) = P(c_4 \mid c_{4-2:i-1})$

   We can define the probability of a sequence of characters $P(c_1:N)$ under the trigram model by first factoring with the chain rule and then using the Markov assumption:

$$P(C_1 : N) = \prod_{i=1}^{N} P(c_1 \mid c_{1:i-1}) = \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1})$$

For a trigram character model in a language with 100 characters, $P(C_i|C_{i-2:i=1})$ has a million entries, and can be accurately estimated by counting character sequences in a body of text of 10 million characters or more. We call a body of text a **corpus** (plural corpora), from the Latin word for body.

2. **Language identification:** Given a text, determine what natural language it is written in. This is a relatively easy task, even with short texts such as “Hello, world” or “Wiegehtesdir”, it is easy to identify the first as English and the second as German. Computer systems identify languages with greater than 99% accuracy; occasionally, closely related languages, such as Swedish and Norwegian, are confused.

3. One approach to language identification is to first build a trigram character model of each candidate language, P (ci | ci-2; i-1), where the variable_ranges over languages. For each_the model is built by counting trigrams in a corpus of that language. (About 100,000 characters of each language are needed). That gives us a model of P (Text | Language), but we want to select the most probable language given the text, so we apply Bayes’ rule followed by the Markov assumption to get the most probable language:

$$\ell^* = \underset{\ell}{\arg\max}\ \mathrm{argmaxP}(\ell \mid e_1 : N)$$

$$= \underset{\ell}{\mathrm{argmax}}\ P(\ell)O(C_{1:N}\ell)$$

$$= \underset{\ell}{\mathrm{argmax}}\ P(\ell)\prod_{i=1}^{N} P(c_4 \mid c_4 - 2 : i - 1, \ell)$$

4. Other tasks for character models include spelling correction, genre classification, and named-entity recognition. Genre classification means decided if a text is a new story, a legal document, a scientific article, etc. While many features help make this classification, counts of punctuation and other character n-gram features go a long way (Kessler et al., 1997).

5. **Named-entity recognition** is the task of finding names of things in a document and deciding what class they belong to. For example, in the text “MrSopersteen was prescribed aciphex”, we should recognize that “Mr. Sopersteen” is the name of a person and “aciphex” is the name of a drug. Character-level models are good for this task because they can associate the character sequence “ex” (“ex” followed by a space) with a drug name and “steen” with a person name, and thereby identify words that they have never seen before.

## 5.3 SMOOTHING N-GRAM MODELS

1. The major complication of n-gram models is that the training corpus provides only an estimate of the true probability distribution.

2. For common character sequence such as “th” any English corpus will give a good estimate: about 1.5% of all trigrams.

3. On the other hand, “ht” is very uncommon – no dictionary words start5 with ht. It is likely that the sequence would

4. Have a count of zero in a training corpus of standard English. Does that mean we should assign P(“th”) = 0? If we did, then the text “The program issues an http request” would have an English probability of zero, which seems wrong.

5. The process adjusting the probability of low-frequency counts is called smoothing.

6. A better approach is a **backoff model**, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to (n-1) grams. **Linear interpolation smoothing** is a backoff model that combines trigram, and unigram models by linear interpolation. It defines the probability estimate as

   * P(ci|ci-2:i=1) = λ3P(ci|ci-2:i-1) + λ2P(ci|ci-1) + λ1P(ci), where λ3 + λ2 + λ1 = 1.

   It is also possible to have the values of λI depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models.

## 5.4 MODEL EVALUATION

1. The evaluation can be a task-specific metric, such as meaning accuracy on language identification.

2. Alternatively, we can have a task-independent model of language quality: calculate the probability assigned to the validation corpus by the model; the higher the probability the better. This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue. A different way of describing the probability of a sequence is with a measure called **perplexity,** defined as

   Perplexity (c1:N) = P(c1:N)$^{-1/N}$

3. Perplexity can be thought of as the reciprocal of probability, normalized by sequence length.

4. It can also be thought of as the weighted average branching factor5 of a model. Suppose there are 100 characters in our language, and our model says they are all equally likely.

Then for a sequence of any length, the perplexity will be 100. If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

## 5.5 N-GRAM WORD MODEL

1. n-gram models over words rather than characters
2. All the same mechanism applies equally to word and character models.
3. The main difference is that the vocabulary— the set of symbols that make up the corpus and the model—is larger.
4. There are only about 100 characters in most languages, and sometimes we build character models that are even more restrictive, for example by treating “A” and “a” as the same symbol or by treating all punctuation as the same symbol. But with word models we have at least tens of thousands of symbols, and sometimes millions. The wide range is because it is not clear what constitutes a word.
5. Word n-gram models need to deal with out of vocabulary words.
6. With word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model.
7. This can be done by adding just one new word to the vocabulary: <UNK>, standing for the unknown word.
8. Sometimes multiple unknown-word symbols are used, for different classes. For example, any string of digits might be replaced with <NUM>, or any email address with <EMAIL>.

## 5.6 INFORMATION RETRIEVAL

DEFINITION: Information retrieval is the task of documents that are relevant to a user’s need for information.

The best-known examples of information retrieval systems are search engines on the WorldWideWeb. A Web user can type a query such as [AI book] into a search engine and see a list of relevant pages. In this section, we will see how such systems are built. An information retrieval (henceforth IR) system can be characterized by

1. A corpus of documents. Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.
2. Queries posed in a query language. A query specifies what the user wants to know. The query language can be just a list of words, such as [AI book]; or it can specify a phrase of words that must be adjacent, as in [“AI book”]; it can contain Boolean operators as

in [AI AND book]; it can include non-Boolean operators such as [AI NEAR book] or [AI book site:www.aaai.org].

3. A result set. This is the subset of documents that the IR system judges to be relevant to the query. By relevant, we mean likely to be of use to the person who posed the query, for the particular information need expressed in the query.

4. A presentation of the result set. This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three-dimensional space, rendered as a two-dimensional display.

The earliest IR systems worked on a Boolean keyword model. Each word in the document collection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not.

**Advantage**

1. Simple to explain and implement.

**Disadvantages**

1. The degree of relevance of a document is a single bit, so there is no guidance as to how to order the relevant documents for presentation.

2. Boolean expressions are unfamiliar to users who are not programmers or logicians. Users find it unintuitive that when they want to know about farming in the states of Kansas and Nebraska they need to issue the query [farming (Kansas OR Nebraska)].

3. It can be hard to formulate an appropriate query, even for a skilled user. Suppose we try [information AND retrieval AND models AND optimization] and get an empty result set. We could try [information OR retrieval OR models OR optimization], but if that returns too many results, it is difficult to know what to try next.

## 5.7 IR SCORING FUNCTIONS

1. Most IR systems have abandoned the Boolean model and use models based on the statistics of word counts.

2. A scoring function takes a document and a query and returns a numeric score; the most relevant documents have the highest scores

3. In the BM25 scoring function, the score is a linear weighted combination of scores for each of the words that make up the query

4. Three factors affect the weight of a query term:

- First, the frequency with which a query term appears in a document (also known as TF for term frequency). For the query [farming in Kansas], documents that mention "farming" frequently will have higher scores.
- Second, the inverse document frequency of the term, or IDF. The word "in" appears in almost every document,
- so it has a high document frequency, and thus a low inverse document frequency, and thus it is not as important to the query as "farming" or "Kansas."
- Third, the length of the document. A million-word document will probably mention all the query words, but may not actually be about the query. A short document that mentions all the words is a much better candidate.

5. The BM25 function takes all three of these into account

$$\mathrm{BM25}(d_j, q1:N) = \sum_{i=1}^{N} \mathrm{IDF}(q_i).\frac{\mathrm{TF}(q_i, d_j).(k+1)}{\mathrm{TF}(q_i, d_j) + k.(1 - b + a.\frac{|d_j|}{L}}$$

where $|d_j|$ is the length of document $d_j$ in words, and L is the average document length in the corpus, $L = \Sigma_i |d_i| N$. We have two parameters, k and b, that can be tuned by cross-validation; typical values are k = 2.0 and b = 0.75. IDF (qi) is the inverse document.

6. Systems create an index ahead of time that lists, for each vocabulary word, the documents that contain the word. This is called the hit list for the word. Then when given an query, we intersect the hit lists of the query words and only score the documents in the intersection.

## 5.8 IR SYSTEM EVALUATION

Imagine the an IR system has returned a result set for a single query, for which we know which documents are and are not relevant, out of a corpus of 100 documents. The document counts in each category are given in the following table.

**Table 5.1**

| | **In result set** | **Not in result set** |
|---|---|---|
| Relevant | 30 | 20 |
| Not relevant | 10 | 40 |

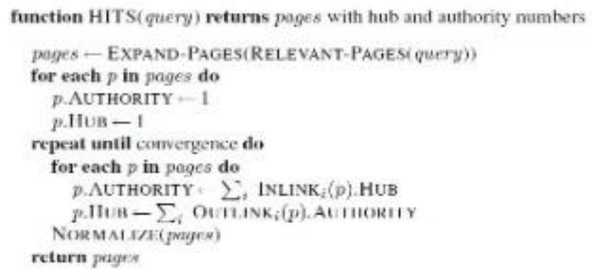
Precision measures the proportion of documents in the result set that are actually relevant. In our example, the precision is 30/(30 + 10)=.75. The false positive rate is 1 − .75=.25. 2. Recall measures the proportion of all the relevant documents in the collection that are in the result set. In our example, recall is 30/(30 + 20)=.60. The false negative rate is 1 −.60=.40. 3. In a very large document collection, such as the WorldWideWeb, recall is difficult to compute, because there is  no easy way to examine every page on the Web for relevance.

## 5.9 IR REFINEMENTS

1. One common refinement is a better model of the effect of document length on relevance.
2. The BM25 scoring function uses a word model that treats all words as completely independent, but we know that some words are correlated: “couch” is closely related to both “couches” and “sofa.” Many IR systems attempt to account for these correlations.
3. For example, if the query is [couch], it would be a shame to exclude from the result set those documents that mention “COUCH” or “couches” but not “couch.” Most IR systems do case folding of “COUCH” to “couch,” and some use a stemming algorithm to reduce “couches” to the stem form “couch,” both in the query and the documents.
4. The next step is to recognize synonyms, such as “sofa” for “couch.” As with stemming, this has the potential for small gains in recall, but can hurt precision. Synonyms and related words can be found in dictionaries or by looking for correlations in documents or in queries.
5. As a final refinement, IR can be improved by considering metadata—data outside of the text of the document.  Examples include human-supplied keywords and publication data. On the Web, hypertext links between documents are a crucial source of information.

## 5.10 PAGERANKALGORITHM

PageRank (PR) is an algorithm used by Google Search to rank web pages in their search engine results. Page Rank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages. According to Google: PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other website



```
function HITS(query) returns pages with hub and authority numbers
  pages ← EXPAND-PAGES(RELEVANT-PAGES(query))
  for each p in pages do
    p.AUTHORITY ← 1
    p.HUB ← 1
  repeat until convergence do
    for each p in pages do
      p.AUTHORITY ← $\sum_i$ INLINK$_i$(p).HUB
      p.HUB ← $\sum_i$ OUTLINK$_i$(p).AUTHORITY
    NORMALIZE(pages)
  return pages
```

**Figure 5.1 The HITS algorithm for computing hubs and authorities with respect to a query. REELVANT-PAGES fetches the pages that match the query and EXPAND-PAGES adds in every page that links to or is linked from one of the relevant pages. NORMALIZE divides each page's score by the sum of the sequence of all pages' scores (separately for both the authority and hubs scores)**

We will see that the recursion bottom out property. The PageRank for a page p is defined as

$$PR(p) = \frac{1-d}{N} + d\sum_i \frac{PR(in_i)}{C(in_i)}$$

where PR(p) is the PageRank of page p, N is the total number of pages in the corpus, $in_i$ are the pages that link into p, and $C(in_i)$ is the count of the total number of out-links on page $in_i$. The constant d is a damping factor. It can be understood through the **random surface model**: imagine a Web surface who starts at some random page and begins exploring.

## 5.11 THE HITS ALGORITHM

1. The Hyperlink-Induced Topic Search algorithm, also known as "Hubs and Authorities" or HITS, is another influential link-analysis algorithm.

2. HITS differ from PageRank in several ways. First, it is a query-dependent measure: it rates pages with respect to a query.

3. HITS first find a set of pages that are relevant to the query. It does that by intersecting hit lists of queries

4. Words, and then adding pages in the link neighborhood of these pages—pages that link to or are linked from one of the pages in the original relevant set.

5. Each page in this set is considered an authority on the query to the degree that other pages in the relevant set point to it. A page is considered a hub to the degree that it points to other authoritative pages in the relevant set.

6. Just as with PageRank, we don't want to merely count the number of links; we want to give more value to the high-quality hubs and authorities.

7. Thus, as with PageRank, we iterate a process that updates the authority score of a page to be the sum of the hub scores of the pages that point to it, and the hub score to be the sum of the authority scores of the pages it points to.

8. Both PageRank and HITS played important roles in developing our understanding of Web information retrieval. These algorithms and their extensions are used in ranking

billions of queries daily as search engines steadily develop better ways of extracting yet finer signals of search relevance.

## 5.12 IMAGE EXTRACTION

Information extraction is the process of acquiring INFORMATION knowledge by skimming a text and looking for occurrences of a particular class of object and for relationships among objects. A typical task is to extract instances of addresses from Web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation. In a limited domain, this can be done with high accuracy.

## 5.13 FINITE-STATE AUTOMATA FOR INFORMATION EXTRACTION

1. The simplest type of information extraction system is an attribute-based extraction system that assumes that the entire text refers to a single object and the task is to extract attributes of that object.

2. One step up from attribute-based extraction systems are relational extraction systems, which deal with multiple objects and the relations among them.

3. A typical relational-based extraction system is FASTUS, which handles news stories about corporate mergers and acquisitions.

4. A relational extraction system can be built as a series of cascaded finite-state transducers.

5. That is, the system consists of a series of small, efficient finite-state automata (FSAs), where each automaton receives text as input, transduces the text into a different format, and passes it along to the next automaton. FASTUS consists of five stages: 1. Tokenization 2. Complex-word handling 3. Basic-group handling 4. Complex-phrase handling 5. Structure merging

6. FASTUS's first stage is tokenization, which segments the stream of characters into tokens (words, numbers, and punctuation). For English, tokenization can be fairly simple; just separating characters at white space or punctuation does a fairly good job. Some tokenizers also deal with markup languages such as HTML, SGML, and XML.

7. The second stage handles complex words, including collocations such as "set up" and "joint venture," as well as proper names such as "Bridgestone Sports Co." These are recognized by a combination of lexical entries and finite- state grammar rules.

8. The third stage handles basic groups, meaning noun groups and verb groups. The idea is to chunk these into units that will be managed by the later stages.

9. The fourth stage combines the basic groups into complex phrases. Again, the aim is to have rules that are finite- state and thus can be processed quickly, and that result in

unambiguous (or nearly unambiguous) output phrases. One type of combination rule deals with domain-specific events.

10. The final stage merges structures that were built up in the previous step. If the next sentence says “The joint venture will start production in January,” then this step will notice that there are two references to a joint venture, and that they should be merged into one. This is an instance of the identity uncertainty problem.

## 5.14 PROBABILISTIC MODELS FOR INFORMATION EXTRACTION

1. The simplest probabilistic model for sequences with hidden state is the hidden Markov model, or HMM.

2. HMMs have two big advantages over FSAs for extraction.

- First, HMMs are probabilistic, and thus tolerant to noise. In a regular expression, if a single expected character is missing, the regex fails to match; with HMMs there is graceful degradation with missing characters/words, and we get a probability indicating the degree of match, not just a Boolean match/fail.

- Second, HMMs can be trained from data; they don’t require laborious engineering of templates, and thus they can more easily be kept up to date as text changes over time.

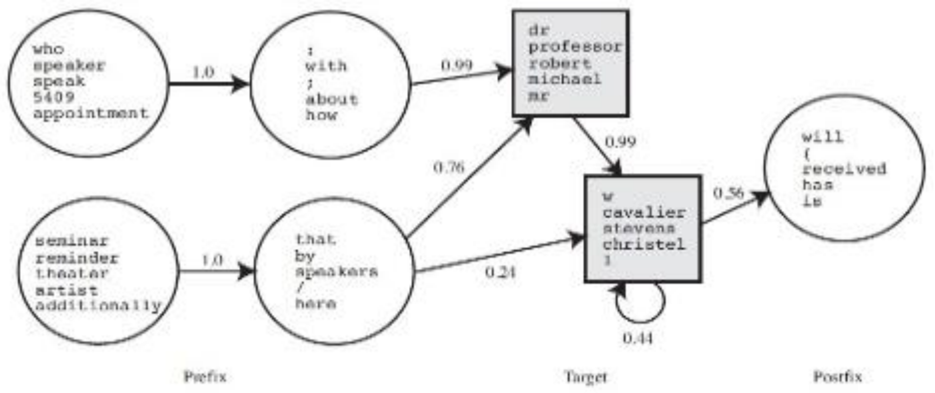


**Figure 5.2 Hidden Markov model for the speaker of a talk announcement. The two square states are the target (note the second target state has a self-loop, so the target can match a strain of any length), the four circles to the left are prefix, and the one of the right is the postfix. For each state, only a few of the high-probability words are shown. From Freitag and McCallum (2000)**

3. Once the HMMs have been learned, we can apply them to a text, using the Viterbi algorithm to find the most likely path through the HMM states. One approach is to apply each attribute HMM separately; in this case you would expect most of the HMMs to spend most of their time in background states. This is appropriate when the extraction is sparse - when the number of extracted words is small compared to the length of the text.

4. The other approach is to combine all the individual attributes into one big HMM, which would then find a path that wanders through different target attributes, first finding a speaker target, then a date target, etc. Separate HMMs are better when we expect just one of each attribute in a text and one big HMM is better when the texts are more free-form and dense with attributes.

5. HMMs have the advantage of supplying probability numbers that can help make the choice. If some targets are missing, we need to decide if this is an instance of the desired relation at all, or if the targets found are false positives. A machine learning algorithm can be trained to make this choice.

## 5.15 ONTOLOGY EXTRACTION FROM LARGE CORPORA

1. A different application of extraction technology is building a large knowledge base or ontology of facts from a corpus. This is different in three ways:

- First it is open-ended—we want to acquire facts about all types of domains, not just one specific domain.
- Second, with a large corpus, this task is dominated by precision, not recall—just as with question answering on the Web
- Third, the results can be statistical aggregates gathered from multiple sources, rather than being extracted from one specific text.

2. Here is one of the most productive templates: NP such as NP (, NP)* (,)? ((and | or) NP)?.

3. Here the bold words and commas must appear literally in the text, but the parentheses are for grouping, the asterisk means repetition of zero or more, and the question mark means optional.

4. NP is a variable standing for a noun phrase

5. This template matches the texts “diseases such as rabies affect your dog” and “supports network protocols such as DNS,” concluding that rabies is a disease and DNS is a network protocol.

6. Similar templates can be constructed with the key words “including,” “especially,” and “or other.” Of course these templates will fail to match many relevant passages, like “Rabies is a disease.” That is intentional.

7. The “NP is a NP” template does indeed sometimes denote a subcategory relation, but it often means something else, as in “There is a God” or “She is a little tired.” With a large corpus we can afford to be picky; to use only the high-precision templates.

8. We’ll miss many statements of a subcategory relationship, but most likely we’ll find a paraphrase of the statement somewhere else in the corpus in a form we can use.

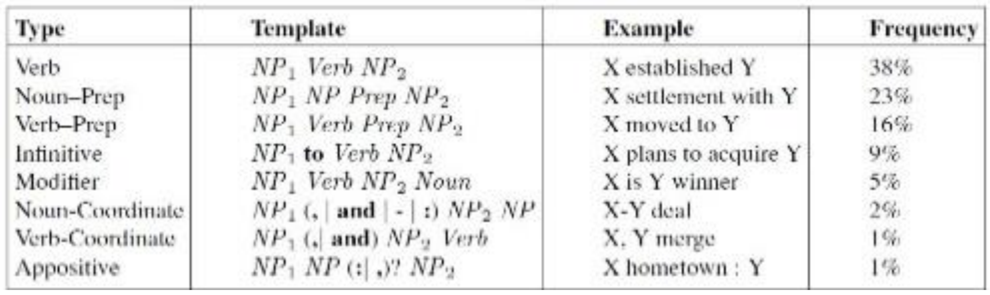
## 5.16 AUTOMATED TEMPLATE CONSTRUCTION

Clearly these are examples of the author–title relation, but the learning system had no knowledge of authors or titles. The words in these examples were used in a search over a Web corpus, resulting in 199 matches. Each match is defined as a tuple of seven strings, (Author, Title, Order, Prefix, Middle, Postfix, URL), where Order is true if the author came first and false if the title came first, Middle is the characters between the author and title, Prefix is the 10 characters before the match, Suffix is the 10 characters after the match, and URL is the Web address where the match was made.

1. Each template has the same seven components as a match.
2. The Author and Title are regexes consisting of any characters (but beginning and ending in letters) and constrained to have a length from half the minimum length of the examples to twice the maximum length.
3. The prefix, middle, and postfix are restricted to literal strings, not regexes.
4. The middle is the easiest to learn: each distinct middle string in the set of matches is a distinct candidate template. For each such candidate, the template's Prefix is then defined as the longest common suffix of all the prefixes in the matches, and the Postfix is defined as the longest common prefix of all the postfixes in the matches.
5. If either of these is of length zero, then the template is rejected.
6. The URL of the template is defined as the longest prefix of the URLs in the matches. The biggest weakness in this approach is the sensitivity to noise. If one of the first few templates is incorrect, errors can propagate quickly. One way to limit this problem is to not accept a new example unless it is verified by multiple templates, and not accept a new template unless it discovers multiple examples that are also found by other templates.

## 5.17 MACHINE READING

1. Traditional information extraction system that is targeted at a few relations and more like a human reader who learns from the text itself; because of this the field has been called machine reading.
2. A representative machine-reading system is TEXTRUNNER. TEXTRUNNER uses cotraining to boost its performance, but it needs something to bootstrap from.
3. Because TEXTRUNNER is domain-independent, it cannot rely on predefined lists of nouns and verbs.



| Type | Template | Example | Frequency |
|---|---|---|---|
| Verb | $NP_1\ Verb\ NP_2$ | X established Y | 38% |
| Noun–Prep | $NP_1\ NP\ Prep\ NP_2$ | X settlement with Y | 23% |
| Verb–Prep | $NP_1\ Verb\ Prep\ NP_2$ | X moved to Y | 16% |
| Infinitive | $NP_1$ **to** $Verb\ NP_2$ | X plans to acquire Y | 9% |
| Modifier | $NP_1\ Verb\ NP_2\ Noun$ | X is Y winner | 5% |
| Noun-Coordinate | $NP_1\ (, \mid \textbf{and} \mid - \mid :)\ NP_2\ NP$ | X-Y deal | 2% |
| Verb-Coordinate | $NP_1\ (, \mid \textbf{and})\ NP_2\ Verb$ | X, Y merge | 1% |
| Appositive | $NP_1\ NP\ (: \mid ,)?\ NP_2$ | X hometown : Y | 1% |

**Figure 5.3 Eight general templates that cover about 95% of the ways that relations are expressed in English**

TEXTRUNNER achieves a precision of 88% and recall of 45% (F1 of 60%) on a large Web corpus. TEXTRUNNER has extracted hundreds of millions of facts from a corpus of half-billion Web pages.

## 5.18 NATURAL LANGUAGE PROCESSING

Machine translation, sometimes referred to by the abbreviation MT (not to be confused with computer-aided translation, machine-aided human translation (MAHT) or interactive translation) is a sub-field of computational linguistics that investigates the use of software to translate text or speech from one language to another.

On a basic level, MT performs simple substitution of words in one language for words in another, but that alone usually cannot produce a good translation of a text because recognition of whole phrases and their closest counterparts in the target language is needed. Solving this problem with corpus statistical, and neural techniques is a rapidly growing field that is leading to better translations, handling differences in linguistic typology, translation of idioms and the isolation of anomalies.

### MACHINE TRANSLATION SYSTEMS

All translation systems must model the source and target languages, but systems vary in the type of models they use. Some systems, attempt to analyse the sourced language text all the way in to an interlingua knowledge representation and then generate sentences in the target language from that representation. This is difficult because it involves three unsolved problems: creating a complete knowledge representation of everything; parsing into that representation; and generating sentences from that representation. Other systems are based on a transfer model.

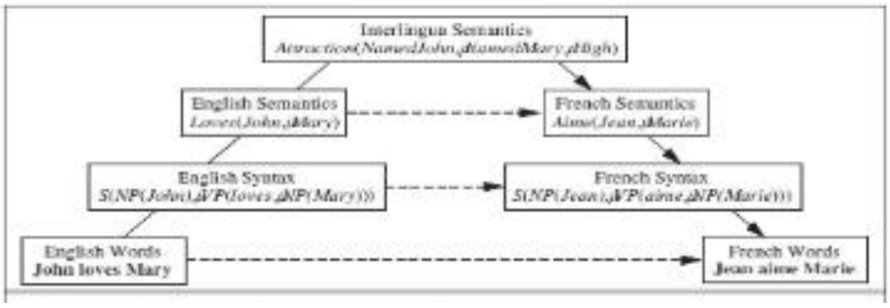


**Figure 5.4 The Vauquois triangle: schematic diagram of the choice for a machine translation system (Vauquois, 1968). We start with English test as the top. An interlingua-based system follows the solid lines, pursuing English first into a syntactic form, then into a semantic representation and an interlingua representation, and then through generation to a semantic, syntactic and lexical form in French. A transfer-based system uses the dashed lines as a shortcut. Different systems make the transfer at different points: some make it at multiple points.**

## STATISTICAL MACHINE TRANSLATION

1. **Find parallel texts:** First, gather a parallel bilingual corpus. For example, a **Hansard** is a record of parliamentary debate. Canada, Hong Kong, and other countries produce bilingual Hansards, the European Union publishes its official documents in 11 languages, and the United Nations publishes multilingual documents. Bilingual text is also available online; some Web sites publish parallel content with parallel URLs, for example, /en/ for the English page and /fr/ for the corresponding French page. The leading statistical translation systems train on hundreds of millions of words of parallel text and billions of words of monolingual text.

2. **Segment into sentences:** The unit of translation is a sentence, so we will have to break the corpus into sentences. Periods are strong indicators of the end of a sentences, but consider "Dr. J.R. Smith of Rodeo Dr. PAD $ 29.99 ON 9.9.09"; only the final period ends a sentence. One way to decide if a period ends a sentence is to train a model that takes as features the surrounding words and their parts of speech. This approach achieves about 98% accuracy.

3. **Align sentences:** For each sentence in the En1glish version, determine what sentence(s) it corresponds to in the French version. Usually, the next sentence of English corresponds to the next sentence of French in a 1:1 match, but sometimes there is variation: one sentence in one language will be split into a 2:1 match, or the order of two sentences will be swapped, resulting in a 2:2 match. By looking at the sentence lengths alone (i.e. short sentences should align with short sentences), it is possible to align (1:1, 1:2, or 2;2, etc) with accuracy in the 90% to 99% range using a variation on the Viterbi algorithm.

4. **Align phrases:** Within a sentence, phrases can be aligned by a process that is similar to that used for sentence alignment, but requiring iterative improvement. When we start, we have no way of knowing that "qui dort" aligns with "sleeping", but we can arrive at that alignment by a process of aggregation of evidence.

5. **Extract distortions:** Once we have an alignment of phrases, we can define distortion probabilities. Simply count how often distortion occurs in the corpus for each distance.

6. **Improve estimates with EM:** Use expectation – maximization to improve the estimate of P(f|e) and P(d) values. We compute the best alignments with the current values of these parameters in the E step, then update the estimates in the M step and iterate the process until convergence.

**SPEECH RECOGNITION**

**Definition:** Speech recognition is the task of identifying a sequence of SPEECH words uttered by a speaker, given the acoustic signal. It has become one of the mainstream applications of AI.

1. **Example:** The phrase "recognize speech" sounds almost the same as "wreak a nice beach" when spoken quickly. Even this short example shows several of the issues that make speech problematic.

2. First **segmentation:** written words in English have spaces between them, but in fast speech there are no pauses in "wreck a nice" that would distinguish it as a multiword phrase as opposed to the single word "recognize".

3. Second, **coarticulation**: when speaking quickly the "s" sound at the end of "nice" merges with the "b" sound at the beginning of "beach" yielding something that is close to a "sp". Another problem that does not show up in this example is **homophones** – words like "to", "too" and "two" that sound the same but differe in meaning

   $$\underset{word_{1:t}}{\operatorname{argmax}} P(word_{1:t} \mid sound_{1:t}) = \underset{word_{1:t}}{\operatorname{argmax}} P(sound_{1:t} \mid word_{1:t})\, P(word_{1:t}).$$

   Heere P ($sound_{1:t}$ |$sound_{1:t}$) is the **acrostic model.** It describes the sound of words – that "ceiling" begins with a soft "c" and sounds the same as "sealing". P ($word_{1:t}$) is known as the **language model**. It specifies the prior probability of each utterance – for example, that "ceiling fan" is about 500 times more likely as a word sequence than "sealing fan".

4. Once we define the acoustic and language models, we can solve for the most likely sequence of words using the Viterbi algorithm.

**Acoustic Model**

1. An analog-to-digital converter measures the size of the current – which approximates the amplitude of the sound wave at discrete intervals called as **sampling rate.**

2. The precision of each measurement is determined by the **quantization factor**, speech recognizers typically keep 8 to 12 bits. That means that a low-end system, sampling at 8 kHz with 8-bit quantization, would require nearly half a megabyte per minute of speech.

3. A **phoneme** is the smallest unit of sound that has a distinct meaning to speakers of a particular language.

For example the "t" in "stick" sounds similar enough to the "t" in "tick" that speakers of English consider them the same phoneme.

Each frame is summarized by a vector of **features. Below picture represents phone model.**

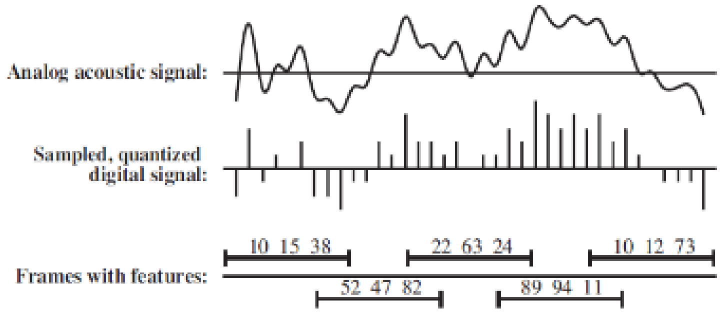


**Figure 5.5 Translating the acoustic signal into a sequence of frames. In this diagram each frame is described by the discretized values of three acoustic features; a real system would have dozens of features**

**Building a speech recognizer**

1. The quality of a speech recognition system depends on the quality of all of its components – the language model, the word-pronunciation models, the phone models, and the signal processing algorithms used to extract spectral features from the acoustic signals.

2. The systems with the highest accuracy work by training a different models for each speaker, thereby capturing differences in dialect as well as male / female and other variations. This training can require several hours of interaction with the speaker, so the systems with the most widespread adoption do not create speaker-specific models.

3. The accuracy of a system depends on a number of factors. First, the quality of the signal matters: a high-quality directional microphone aimed at a stationary mouth in a padded room will do much better than a cheap microphone transmitting a signal over phone lines from a car in traffic with the radio playing. The vocabulary size matters: when

recognizing digit strings with a vocabulary of 11 words (1-9 plus “oh” and “zero)”, the word  error rate will be below 0.5%, where as it rises to about 10% on news stories with a 20,000-word vocabulary, and 20% on a corpus with a 64,000-word vocabulary. The task matters too: when the system is trying to accomplish a specific task – book a flight or give directions to a restaurant – the task can often be a accomplished perfectly even with a word error rate of 109% or more.

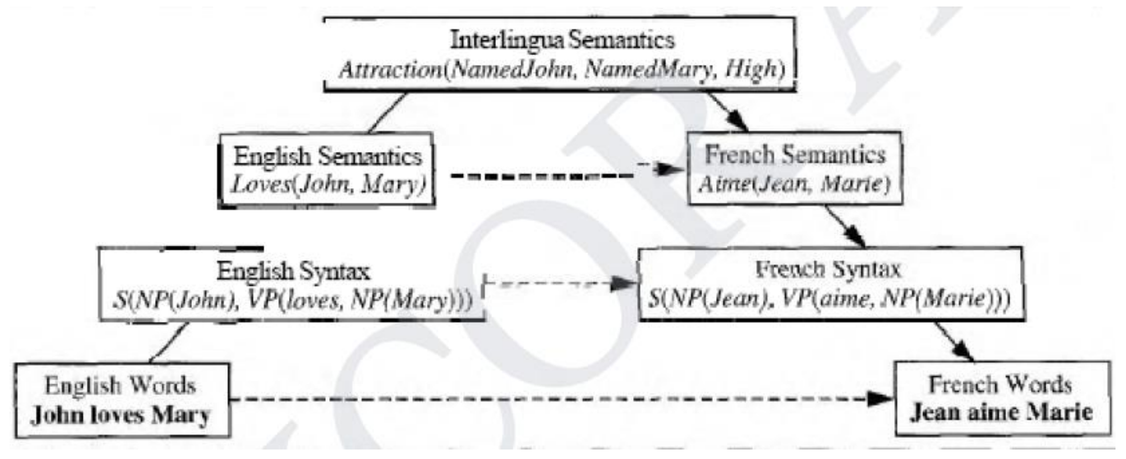


**Figure 5.6A schematic diagram of the choices for a machine translation system. We start with English text at the top. An interlingua-based system follows the solid lines, parsing English first into a syntactic form, then into a semantic representation and an interlingua representation, and then through generation to a semantic, syntactic, and lexical form in French. A transfer-based system uses the dashed lines as a shortcut. Different systems make the transfer at different points; some it at multiple points.**

## 5.19 ROBOT

1. Robots are physical agents that perform tasks by manipulating the physical world.
2. To do so, they are equipped with effectors such as legs, wheels, joints, and grippers.
3. Effectors have a single purpose: to assert physical forces on the environment.
4. Robots are also equipped with sensors, which allow them to perceive their environment.
5. Present day robotics employs a diverse set of sensors, including cameras and lasers to measure the environment, and gyroscopes and accelerometers to measure the robot’s own motion.
6. 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.

### Robot Hardware

1. Sensors are the perceptual interface between robot and environment.
2. Passive sensors, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment.
3. 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 increased power consumption and with a danger of interference when multiple active sensors are used at the same time. Whether active or passive, sensors can be divided into three types, depending on whether they sense the environment, the robot's location, or the robot's internal configuration.
4. Range finders are sensors that measure the distance to nearby objects. In the early days of robotics, robots were commonly equipped with sonar sensors. Sonar sensors emit directional sound waves, which are reflected by objects, with some of the sound making it.
5. Stereo vision relies on multiple cameras to image the environment from slightly different viewpoints, analyzing the resulting parallax in these images to compute the range of surrounding objects. For mobile ground robots, sonar and stereo vision are now rarely used, because they are not reliably accurate.
6. Other range sensors use laser beams and special 1-pixel cameras that can be directed using complex arrangements of mirrors or rotating elements. These sensors are called scanning lidars (short for light detection and ranging).
7. Other common range sensors include radar, which is often the sensor of choice for UAVs. Radar sensors can measure distances of multiple kilometers. On the other extreme end of range sensing are tactile sensors such as whiskers, bump panels, and touch-sensitive skin. These sensors measure range based on physical contact, and can be deployed only for sensing objects very close to the robot.
8. A second important class of sensors is location sensors. Most location sensors use range sensing as a primary component to determine location. Outdoors, the Global Positioning System (GPS) is the most common solution to the localization problem.
9. The third important class is proprioceptive sensors, which inform the robot of its own motion. To measure the exact configuration of a robotic joint, motors are often equipped with shaft decoders that count the revolution of motors in small increments.
10. Inertial sensors, such as gyroscopes, rely on the resistance of mass to the change of velocity. They can help reduce uncertainty.

11. Other important aspects of robot state are measured by force sensors and torque sensors. These are indispensable when robots handle fragile objects or objects whose exact shape and location is unknown.

**Robotic Perception**

1. Perception is the process by which robots map sensor measurements into internal representations of the environment. Perception is difficult because sensors are noisy, and the environment is partially observable, unpredictable, and often dynamic. In other words, robots have all the problems of state estimation (or filtering)

2. As a rule of thumb, good internal representations for robots have three properties: they contain enough information for the robot to make good decisions, they are structured so that they can be updated efficiently, and they are natural in the sense that internal variables correspond to natural state variables in the physical world.

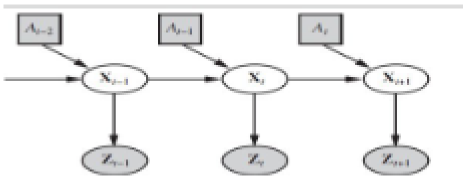


**Figure 5.7 Robot perception can be viewed as temporal inference from sequence of actions and measurements, as illustrated by this dynamic Bayers network**

2. Another machine learning technique enables robots to continuously adapt to broad changes in sensor measurements.

3. Adaptive perception techniques enable robots to adjust to such changes. Methods that make robots collect their own training data (with labels!) are called self-supervised. In this instance, the robot machine learning to leverage a short-range sensor that works well for terrain classification into a sensor that can see much farther.

## 5.20 PLANNING TO MOVE

1. All of a robot's deliberations ultimately come down to deciding how to move effectors.

2. The point-to-point motion problem is to deliver the robot or its end effector to a designated target location.

3. A greater challenge is the compliant motion problem, in which a robot moves while being in physical contact with an obstacle.

4. An example of compliant motion is a robot manipulator that screws in a light bulb, or a robot that pushes a box across a table top. We begin by finding a suitable representation in which motion-planning problems can be described and solved. It turns

out that the configuration space—the space of robot states defined by location, orientation, and joint angles—is a better place to work than the original 3D space.

5. The path planning problem is to find a path from one configuration to another in configuration space. We have already encountered various versions of the path-planning problem throughout this book; the complication added by robotics is that path planning involves continuous spaces. T

6. Here are two main approaches: cell decomposition and skeletonization. Each reduces the continuous path- planning problem to a discrete graph-search problem. In this section, we assume that motion is deterministic and that localization of the robot is exact. Subsequent sections will relax these assumptions.

7. The second major family of path-planning algorithms is based on the idea of skeletonization. These algorithms reduce the robot's free space to a one-dimensional representation, for which the planning problem is easier. This lower-dimensional representation is called a skeleton of the configuration space.

### 5.20.1 Configuration Space

1. It has two joints that move independently. Moving the joints alters the (x, y) coordinates of the elbow and the gripper. (The arm cannot move in the z direction.) This suggests that the robot's configuration can be described by a four-dimensional coordinate: (xe, ye) for the location of the elbow relative to the environment and (xg, yg) for the location of the gripper. Clearly, these four coordinates characterize the full state of the robot. They constitute what is known as workspace representation.

2. Configuration spaces have their own problems. The task of a robot is usually expressed in workspace coordinates, not in configuration space coordinates. This raises the question of how to map between workspace coordinates and configuration space.

3. These transformations are linear for prismatic joints and trigonometric for revolute joints. This chain of coordinate transformation is known as kinematics.

4. The inverse problem of calculating the configuration of a robot whose effector location is specified in workspace coordinates is known as inverse kinematics. The configuration space can be decomposed into two subspaces: the space of all configurations that a robot may attain, commonly called free space, and the space of unattainable configurations, called occupied space.

### 5.20.2 Cell Decomposition Methods

1. The simplest cell decomposition consists of a regularly spaced grid.

2. Grayscale shading indicates the value of each free-space grid cell—i.e., the cost of the shortest path from that cell to the goal.

3. Cell decomposition methods can be improved in a number of ways, to alleviate some of these problems. The first approach allows further subdivision of the mixed cells—perhaps using cells of half the original size. This can be continued recursively until a path is found that lies entirely within free cells. (Of course, the method only works if there is a way to decide if a given cell is a mixed cell, which is easy only if the configuration space boundaries

4. Have relatively simple mathematical descriptions.) This method is complete provided there is a bound on the smallest passageway through which a solution must pass. One HYBRID A* algorithm that implements this is hybrid A*.

### 5.20.3 Modified Cost Functions

1. A potential field is a function defined over state space, whose value grows with the distance to the closet obstacle.

2. The potential field can be used as an additional cost term in the shortest-path calculation.

3. This induces an interesting trade off. On the one hand, the robot seeks to minimize path length to the goal. On the other hand, it tries to stay away from obstacles by virtue to minimizing the potential function.

4. There exist many other ways to modify the cost function. For example, it may be desirable to smooth the control parameters over time.

### 5.20.4 Skeletonization methods

1. The second major family of path-planning algorithms is based on the idea of **skeletonization.**

2. These algorithms reduce the robot's free space to a one-dimensional representation, for which the planning problem is easier.

3. This lower-dimensional representation is called a **skeleton** of the configuration space.

4. It is a **Voronoi graph** of the free space - the set of all points that are equidistant to two or more obstacles. To do path planning with a Voronoi graph, the robot first changes its present configuration to a point on the Voronoi graph.

5. It is easy to show that this can always be achieved by a straight-line motion in configuration space. Second, the robot follows the Voronoi graph until it reaches the point nearest to the target configuration. Finally, the robot leaves the Voronoi graph

and moves to the target. Again, this final step involves straight-line motion in configuration space.

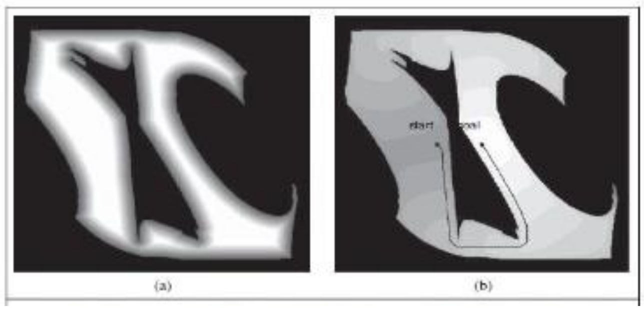


**Figure 5.8(a) A repelling potential field pushes the robot away from obstacles, (b) Path found by simultaneously minimizing path length and the potential**

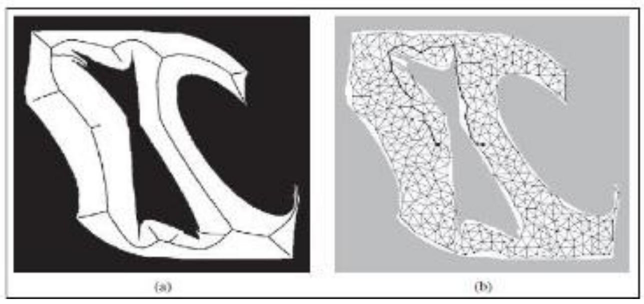


**Figure 5.9 (a) The Voronoi graph is the set of points equidistance to two or more obstacles in configuration space (b) A probabilistic moodmap, composed of 100 randomly chosen points in free space**.

**Robust methods**

1. A robust method is one that assumes a bounded amount of uncertainty in each aspect of a problem, but does not assign probabilities to values within the allowed interval.

2. A robust solution is one that works no matter what actual values occur, provided they are within the assumed intervals.

3. An extreme form of robust method is the **conformant planning** approach.

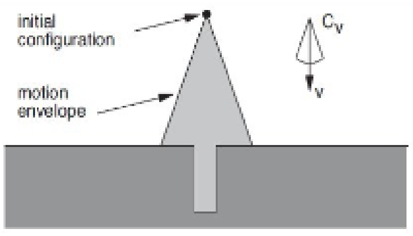


**Figure 5.10A two-dimensional environment, velocity uncertainty cone, and envelope of possible robot motions. The intended velocity is ν, but with uncertainty the actual velocity could be anywhere in Cν, resulting in a final configuration somewhere in the motion envelope, which means we wouldn't know if we hit the hole or not**

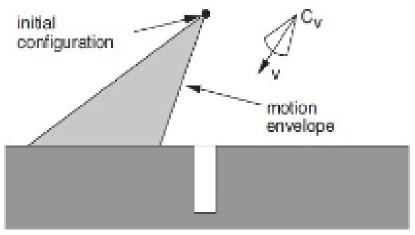


**Figure 5.11The first motion command and the resulting envelope of possible robot motions. No matter what the error, we know the final configuration will be to the left of the hole**

Define Speech Recognition. Explain different types of speech recognition systems.

Speech recognition is an interdisciplinary subfield of computer science and computational linguistics that develops methodologies and technologies that enable the recognition and translation of spoken language into text by computers. It is also known as automatic speech recognition (ASR), computer speech recognition or speech to text (STT). Speech recognition is the process by which computer maps an acoustic speech signal to some form of abstract meaning of the speech. The fundamental aspect of speech recognition is the translation of sound into text and commands.

This process is highly difficult since sound has to be matched with stored sound bites on which further analysis has to be done because sound bites do not match with pre-existing sound pieces. Feature extraction technique and pattern matching techniques plays important role in speech recognition system to maximize the rate of speech recognition of various persons.

Types of speech recognition system based on utterances

1. Isolated Words Isolated word recognition system which recognizes single utterances i.e. single word. Isolated word recognition is suitable for situations where the user is required to give only one word response or commands. It is simple and easiest for implementation because word boundaries are obvious and the words tend to be clearly pronounced which is the major advantage of this type. Eg. Yes/No, Listen / Not-Listen

2. Connected Words A connected words system is similar to isolated words, but it allows separate utterances to be - run-together‟ with a minimal pause between them.

Utterance is the vocalization of a word or words that represent a single meaning to the computer. Eg. drive car(action + object), baby book(agent + object) \

3. Continuous Speech Continuous speech recognition system allows users to speak almost naturally, while the computer determines its content. Continuous speech recognition system is difficult to develop. Eg. dictation

4. Spontaneous Speech Spontaneous speech recognition system recognizes the natural speech. Spontaneous speech is natural that comes suddenly through mouth. An ASR system with spontaneous speech is able to handle a variety of natural speech features such as words being run together. Spontaneous speech may include mispronunciation, false-starts and non words. Eg. Spontaneous speech may include mispronunciation, false-starts (will you – will you go to your college?) and non words("ums").

Explain about Automatic Speech Recognition System

Speech recognition is an interdisciplinary subfield of computer science and computational linguistics that develops methodologies and technologies that enable the recognition and translation of spoken language into text by computers. It is also known as automatic speech recognition (ASR), computer speech recognition or speech to text (STT).

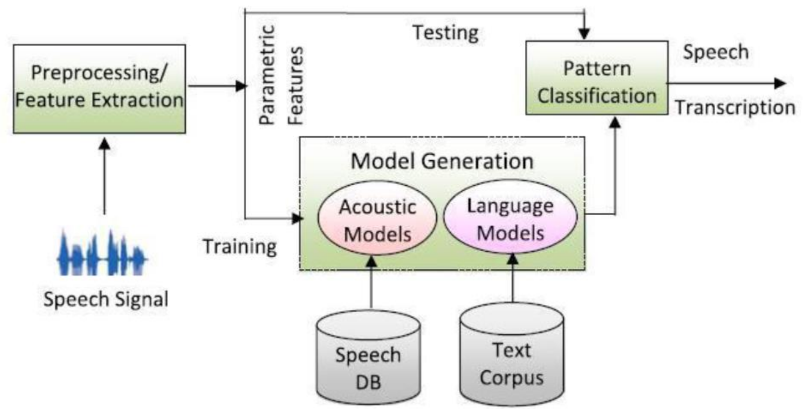


**Figure 5.12 System Architecture of for Automatic Speech Recognition System**

Pre-processing/Digital Processing The recorded acoustic signal is an analog signal. An analog signal cannot directly transfer to the ASR systems. So these speech signals need to transform in the form of digital signals and then only they can be processed. These digital signals are move to the first order filters to spectrally flatten the signals. This procedure increases the energy of signal at higher frequency.

Feature Extraction Feature extraction step finds the set of parameters of utterances that have acoustic correlation with speech signals and these parameters are computed through processing of the acoustic waveform. These parameters are known as features or feature vectors ($x_1$, $x_2$, $x_3$, ...). The main focus of feature extractor is to keep the relevant information and discard irrelevant one. Feature extractor divides the acoustic signal into 10-25 ms. Data acquired in these frames is multiplied by window function. There are many types of window functions that can be used such as hamming Rectangular, Blackman, Welch or Gaussian etc. Feature extraction methods: Principal component Analysis (PCA), Linear Discriminate Analysis (LDA), Wavelet, Independent component Analysis(ICA) etc.

## TEXT/REFERENCE BOOKS

1. S. Russell and P. Norvig, “Artificial Intelligence: A Modern Approach, Prentice Hall, 3rd Edition, 2009.
2. M. Tim Jones, “Artificial Intelligence: A Systems Approach (Computer Science)”, Jones and Bartlett Publishers, Inc.; 1st Edition, 2008.
3. Nils J. Nilsson, “The Quest for Artificial Intelligence”, Cambridge University Press, 2009.
4. William F. Clocksin and Christopher S. Mellish, “Programming in Prolog: Using the ISO Standard”, 5th Edition, Springer, 2003.
5. Gerhard Weiss, “Multi Agent Systems”, 2nd Edition, MIT Press, 2013.
6. David L. Poole and Alan K. Mackworth, “Artificial Intelligence: Foundations of Computational Agents”, Cambridge University Press, 2010