UNIT I

1

INTRODUCTION

Unit Structure

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Define Artificial Intelligence
- Understand foundations of AI
- Explain how the AI was evolved
- Explain history of AI.
- Describe the Today's AI, and Where and What research/ work is going in AI field.

1.2 INTRODUCTION

1.2.1 Introduction to Ai:

Artificial Intelligence is to make computers intelligent so that they can act intelligently as humans. It is the study of making computers does things which at the moment people are better at.

AI has made great effort in building intelligent systems and understanding them. Another reason to understand AI is that these systems are interesting and useful in their own right.

Many tools and techniques are used to construct AI systems. The AI encompasses a huge variety of fields like computer science, mathematics, logical reasoning, linguistics, neuro-science and psychology to perform specific tasks.

AI can be used in many areas like playing chess, proving mathematical theorems, writing poetry, diagnosing diseases, creating expert systems, speech recognition etc.

1.2.2 Objectives of Ai:

The field of artificial intelligence, or AI, attempts to understand intelligent entities. Thus, one reason to study it is to learn more about ourselves.

AI strives to build intelligent entities as well as understand them.

The study of intelligence is also one of the oldest disciplines. For over 2000 years, philosophers have tried to understand how seeing, learning, remembering, and reasoning could, or should, be done.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases.

Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavor. In this sense, it is truly a universal field.

1.3 ARTIFICIAL INTELLIGENCE/WHAT IS ARTIFICIAL INTELLIGENCE?

1.3.1 What Is Artificial Intelligence?:

Artificial Intelligence is the branch of computer science concerned with making computers behave like humans.

Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an intelligent agent is a system that perceives its environment and takes actions which maximize its chances of success. John McCarthy, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."

1.3.2 Definitions of AI:

Artificial intelligence is "The science and engineering of making intelligent machines, especially intelligent computer programs".

Artificial Intelligence is making a computer, a computer-controlled robot, or a software think intelligently, as the intelligent humans think. AI is accomplished by studying how the human brain thinks and how humans learn, decide, and work while trying to solve a problem, and then using the outcomes of this study as a basis of developing intelligent software and systems.

Some Definitions of AI:

There are various definitions of AI. They are basically categorized into four categories:

i) Systems that think like humans
ii) Systems that think rationally
iii) Systems that act like humans
iv) Systems that act rationally

Category 1) Systems that think like humans:

An electronic machine or computer thinks like a human being.

"The exciting new effort to make computers think … machines with minds, in the full literal sense" (Haugeland, 1985)

"The automation of activities that we associate with human thinking, activities such as decision making, problem solving, learning... " (Bellman, 1978)

Category 2) Systems that think rationally:

An electronic machine or computer thinks rationally. If system thinks the right thing, the system is rational.

"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).

Category 3) Systems that act like humans:

An electronic machine or computer acts like human being. It acts same as human being.

"The art of creating machines that performs 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)

Category 4) Systems that act rationally:

An electronic machine or computer acts rationally.

"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)

1.4 FOUNDATIONS OF AI

1.4.1 Introduction:

AI is a field which has inherited many ideas, viewpoints, and techniques from other disciplines like mathematics, formal theories of logic, probability, decision making, and computation.

From over 2000 years of tradition in philosophy, theories of reasoning and learning have emerged, along with the viewpoint that the mind is constituted by the operation of a physical system. From over 400 years of mathematics, we have formal theories of logic, probability, decision making, and computation. From psychology, we have the tools with which to investigate the human mind, and a scientific language within which to express the resulting theories. From linguistics, we have theories of the structure and meaning of language. Finally, from computer science, we have the tools with which to make AI a reality.

1.4.2 Turing Test:

Acting humanly: The Turing Test Approach

- Test proposed by Alan Turing in 1950
- The computer is asked questions by a human interrogator.

The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. Programming a computer to pass , the computer need to possess the following capabilities :

- 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

To pass the complete Turing Test, the computer will need

- Computer vision to perceive the objects, and

- Robotics to manipulate objects and move about.

Problem: Turing test is not reproducible, constructive, or amenable to mathematical analysis

1.4.3 Weak Ai Versus Strong Ai:

The definition of AI is along two dimensions, human vs. ideal and thought vs. action. But there are other dimensions that are worth considering. One dimension is whether we are interested in theoretical results or in practical applications. Another is whether we intend our intelligent computers to be conscious or not.. The claim that machines can be conscious is called the strong AI claim; the WEAK AI position makes no such claim. Artificial intelligence is …

a. "a collection of algorithms that are computationally tractable, adequate approximations of intractably specified problems" (Partridge, 1991)

b. "the enterprise of constructing a physical symbol system that can reliably pass the Turing Test" (Ginsberg, 1993)

c. "the field of computer science that studies how machines can be made to act intelligently" (Jackson, 1986)

d. "a field of study that encompasses computational techniques for performing tasks that apparently require intelligence when performed by humans" (Tanimoto, 1990)

e. "a very general investigation of the nature of intelligence and the principles and mechanisms required for understanding or replicating it" (Sharpies et ai, 1989)

f. "the getting of computers to do things that seems to be intelligent" (Rowe, 1988)

Philosophy (428 B.C.-Present):

Philosophers made AI conceivable by considering the ideas that the mind is in some ways like a machine that operates on knowledge encoded in some internal language and that thought can be used to choose what actions to take.

Philosophy is the implication of knowledge and understanding of intelligence , ethics, logic, methods of reasoning, mind as physical system, foundations of learning, language and rationality.

Mathematics:

Philosophers staked out most of the important ideas of AI, but to make the leap to a formal science required a level of mathematical formalization in three main areas: computation, logic,

Algorithm and Probability:

With logic a connection was made between probabilistic reasoning and action.! DECISION THEORY Decision theory, pioneered by John Von Neumann and Oskar Morgenstern (1944), combines! probability theory with utility theory to give the first general theory that can distinguish good! actions from bad ones. Decision theory is the mathematical successor to utilitarianism, and provides the theoretical basis for many of the agent designs.

Psychology (1879 – Present):

Psychology is to adopt the idea that humans and animals can be considered information processing machines.

It is all about adaptation, phenomena of perception and motor control, experimental techniques (psychophysics, etc.)

Computer Engineering (1940-Present):

For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been the artifact of choice. AI also owes a debt to the software side of computer science, which has supplied the operating systems, programming languages, and tools needed to write modern programs. Computer engineers provided the artifacts that make AI applications possible.

To succeed in artificial intelligence, two things are required: intelligence and an artifact. A computer machine is the best artifact to exhibit the intelligence. During World War II, Some innovations are done in AI field.

In 1940, Alan Turing's team had developed *Heath Robinson* to decode the German messages. Then in 1943, *Colossus* was built from vacuum tubes to decode complex code. In 1941, the first operational programmable computer *Z-3* was invented. In the US, ABC the first electronic computer was assembled between 1940 and 1942. Mark I, II, and III computers were developed at Harvard. ENIAC was developed; it is the first general-purpose, electronic, digital computer. Computing artillery firing tables is one of its application.

In 1952, IBM 701 was built. This was the first computer to yield a profit for its manufacturer. IBM went on to become one of the world's largest corporations, and sales of computers have grown.

Computer engineering has been remarkably successful, regularly doubling performance every two years.

Linguistics (1957-Present):

Linguists showed that language use fits into this model. Modern linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called computational linguistics or natural language processing**.**

In 1957. B. F. Skinner published *Verbal Behavior*. It is related to behaviorist approach to language learning. But curiously, a review of the book became as well-known as the book itself, and served to almost kill off interest in behaviorism. The author of the review was Noam Chomsky, who had just published a book on his own theory. Syntactic Structures. Chomsky showed how the behaviorist theory did not address the notion of creativity in language—it did not explain how a child could understand and make up sentences that he or she had never heard before. Chomsky's theory—based on syntactic models going back to the Indian linguist Panini (c. 350 B.C.)—could explain this, and unlike previous theories, it was formal enough that it could in principle be programmed.

1.5 HISTORY OF AI

The Gestation of Artificial Intelligence (1943-1955):

The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943). They drew on three sources: knowledge of the basic physiology and function of neurons in the brain; the formal analysis of propositional logic due to Russell and Whitehead; and Turing's theory of computation.

McCarthy convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence. They organized a two-month workshop at Dartmouth in the summer of 1956. Perhaps the longest-lasting thing to come out of the workshop was an agreement to adopt McCarthy's new name for the field: artificial intelligence.

Early Enthusiasm, Great Expectations (1952-1969):

The early years of A1 were full of successes-in a limited way.

General Problem Solver (GPS) was a computer program created in 1957 by Herbert Simon and Allen Newell to build a universal problem solver machine. The order in which the program considered sub goals and possible actions was similar to that in which humans approached the same problems. Thus, GPS was probably the first program to embody the "thinking humanly" approach.

At IBM, Nathaniel Rochester and his colleagues produced some of the first A1 programs. Herbert Gelernter (1959) constructed the Geometry Theorem Prover, which was able to prove theorems that many students of mathematics would find quite tricky.

Lisp was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT). In 1963, McCarthy started the AI lab at Stanford.

Tom Evans's ANALOGY program (1968) solved geometric analogy problems that appear in IQ tests.

A Dose of Reality (1966-1973):

From the beginning, AI researchers were making predictions of their coming successes. The following statement by Herbert Simon in 1957 is often quoted: “It is not my aim to surprise or shock you-but the simplest way I can summarize is to say that there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until-in a visible future-the range of problems they can handle will be coextensive with the range to which the human mind has been applied.

Knowledge-Based Systems: The Key To Power? (1969-1979):

Dendral was an influential pioneer project in artificial intelligence (AI) of the 1960s, and the computer software expert system that it produced. Its primary aim was to help organic chemists in identifying unknown organic molecules, by analyzing their mass spectra and using knowledge of chemistry.

A1 Becomes An Industry (1980-Present):

In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running Prolog. Overall, the A1 industry boomed from a few million dollars in 1980 to billions of dollars in 1988.

The Return of Neural Networks (1986-Present):

Although computer science had neglected the field of neural networks after Minsky and Papert's Perceptrons book, work had continued in other fields, particularly physics. Large collections of simple neurons could be understood in much the same way as large collections of atoms in solids.

Psychologists including David Rumelhart and Geoff Hinton continued the study of neural-net models of memory.

Recent Events:

In recent years, approaches based on hidden Markov models (HMMs) have come to dominate the area. Speech technology and the related field of handwritten character recognition are already making the transition to widespread industrial and consumer applications.

The Bayesian network formalism was invented to allow efficient representation of, and rigorous reasoning with, uncertain knowledge.

1.6 THE STATE OF ART AI TODAY

1.6.1 Current Ai Innovations:

Chatbots, smart cars, IoT devices, healthcare, banking, and logistics all use artificial intelligence to provide a superior experience. One AI that is quickly finding its way into most consumer's homes is the voice assistant,

such as Apple's Siri, Amazon's Alexa, Google's Assistant, and Microsoft's Cortana. Some of them are listed below in tabular format.

Area	Application/ Example/ Company Name
Autonomous cars (Driverless car)	Tesla Model S, Pony.ai, Waymo, Apple, Kia-Hyundai, Ford, Audi, Huawei.
Speech Recognition	Apple's Siri and Google's Alexa
Chatbot	Swelly, eBay, Lyft, Yes sire, 1-800-Flowers
Web search engines	ai.google
Translator	SYSTRAN
Natural Language Processing	IBM Watson API,
Medical Diagnosis, Imaging	Artificial Intelligence assistance in "keeping well"
Pattern Detection	RankBrain by Google

1.6.2 Practical Applications of Ai:

Autonomous Planning And Scheduling:

A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson et al., 2000). Remote Agent generated plans from high-level goals specified from the ground, and it monitored the operation of the spacecraft as the plans were executed-detecting, diagnosing, and recovering from problems as they occurred.

Game Playing:

IBM's Deep Blue became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997).

Autonomous Control:

The ALVINN computer vision system was trained to steer a car to keep it following a lane. It was placed in CMU's NAVLAB computer-controlled minivan and used to navigate across the United States-for 2850 miles it was in control of steering the vehicle 98% of the time.

Diagnosis:

Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine.

Logistics Planning:

During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to

account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques allowed a plan to be generated in hours that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

Robotics:

Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia et al., 1996) is a system that uses computer vision techniques to create a three-dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of a hip replacement prosthesis.

Language Understanding And Problem Solving:

PROVERB (Littman et al., 1999) is a computer program that solves crossword puzzles better than most humans, using constraints on possible word fillers, a large database of past puzzles, and a variety of information sources including dictionaries and online databases such as a list of movies and the actors that appear in them.

1.7 SUMMARY

Artificial intelligence is "The science and engineering of making intelligent machines, especially intelligent computer programs".

Turing test: To pass the complete Turing test, the computer will need Computer vision to perceive the objects, and Robotics to manipulate objects and move about.

Artificial intelligence is to make computers intelligent so that they can act intelligently as humans.

Categorical definitions of AI are: Systems that think like humans, Systems that think rationally, Systems that act like humans, Systems that act rationally.

AI comprises of Philosophy, Mathematics, Algorithm and Probability, Psychology, Computer Engineering, Linguistics etc.

Work on Machine Intelligence started early. But the period which is considered as History of AI started from The Gestation of Artificial Intelligence (1943-1955), till The Return of Neural Networks (1986-Present) and Recent Events.

There are various current AI innovations such as Autonomous cars (Driverless car), Speech Recognition, Chatbot, Web search engines, Translator, Natural Language Processing, Medical Diagnosis, Imaging Pattern Detection etc.

Various practical applications of AI such as Autonomous Planning And Scheduling, Game Playing, Autonomous Control, Diagnosis, Logistics

Planning, Robotics, Language Understanding And Problem Solving etc.

1.8 UNIT END QUESTIONS

1.1 There are well-known classes of problems that are intractably difficult for computers, and other classes that are provably undecidable by any computer. Does this mean that AI is impossible?

1.2 Suppose we extend Evans's ANALOGY program so that it can score 200 on a standard IQ test. Would we then have a program more intelligent than a human? Explain.

1.3 Examine the AI literature to discover whether or not the following tasks can currently be solved by computers:

a. Playing a decent game of table tennis (ping-pong).

b. Driving in the center of Cairo.

c. Playing a decent game of bridge at a competitive level.

d. Discovering and proving new mathematical theorems.

e. Writing an intentionally funny story.

f. Giving competent legal advice in a specialized area of law.

g. Translating spoken English into spoken Swedish in real time.

1.4 Find an article written by a lay person in a reputable newspaper or magazine claiming the achievement of some intelligent capacity by a machine, where the claim is either wildly exaggerated or false.

1.5 Fact, fiction, and forecast:

a. Find a claim in print by a reputable philosopher or scientist to the effect that a certain capacity will never be exhibited by computers, where that capacity has now been exhibited.

b. Find a claim by a reputable computer scientist to the effect that a certain capacity would be exhibited by a date that has since passed, without the appearance of that capacity.

c. Compare the accuracy of these predictions to predictions in other fields such as biomedicine, fusion power, nanotechnology, transportation, or home electronics.

1.6 Some authors have claimed that perception and motor skills are the most important part of intelligence, and that "higher-level" capacities are necessarily parasitic—simple add-ons to these underlying facilities. Certainly, most of evolution and a large part of the brain have been devoted to perception and motor skills, whereas AI has found tasks such as game playing and logical inference to be easier, in many ways, than perceiving and acting in the real world. Do you think

that AI's traditional focus on higher-level cognitive abilities is misplaced?

1.7 "Surely computers cannot be intelligent—they can only do what their programmers tell them." Is the latter statement true, and does it imply the former?

1.8 "Surely animals cannot be intelligent—they can only do what their genes tell them." Is the latter statement true, and does it imply the former?

1.9 REFERENCES

Artificial Intelligence: A Modern Approach, 4th US ed.by Stuart Russell and Peter Norvig.

Deepak Khemani, "A first course in Artificial Intelligence", McGraw Hill edition, 2013.

Patrick Henry Winston , "Artificial Intelligence", Addison-Wesley, Third Edition

*****

2

INTELLIGENT AGENTS

Unit Structure

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Define AI Agent,
- Understand the Agent and its Environment
- Understand the Rationality of agent, Performance measure
- Identify the PEAS, and PAGE
- Understand the nature of Environment
- Explain the Structure of Agents

2.2 INTRODUCTION

An agent is anything that can be viewed as capturing its environment through cameras, sensors or some other input devices and acting upon that environment through effectors. A human agent has eyes, ears, and other organs for sensors, and hands, legs, mouth, and other body parts for effectors. A robotic agent substitutes cameras and infrared range finders for the sensors and various motors for the effectors. A software agent has encoded bit strings as its percepts and actions.

The focus is on design agents that do a good job of acting on their environment,

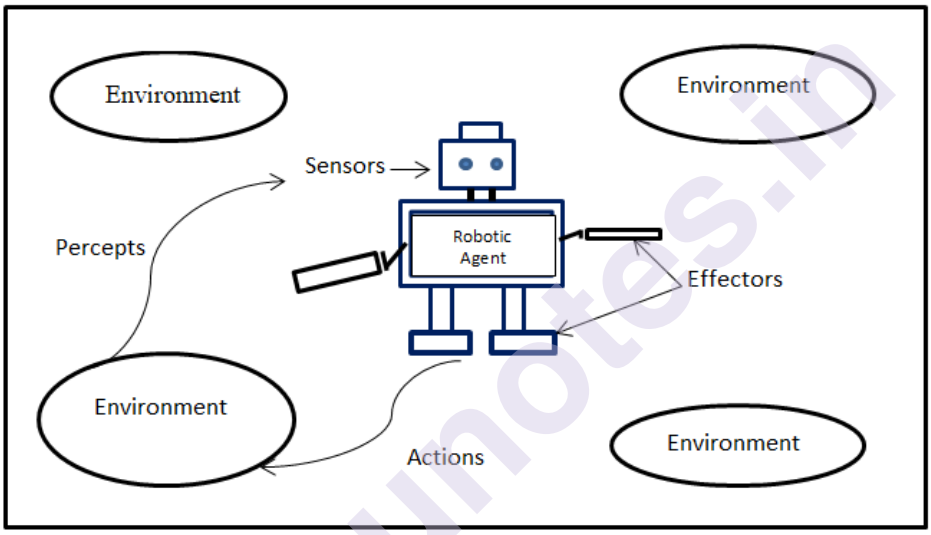
What we mean by a good job, rationality, performance measure, then different designs for successful agents, which things should be known to the agent and several kinds of environments.

2.3 AGENT AND ENVIRONMENT

Definition: An agent is anything that perceives its environment through sensors and acting upon that environment through effectors.

An agent is split into architecture and an agent program.

An intelligent agent is an agent capable of making decisions about how it acts based on experiences.



Sensors are used to receive percept signals. Percept signals can be anything like audio, video, image, etc.

Effectors are used to make action. We can also call effectors as actuators.

The following are the sensors and actuators for Robot / Robotic agent

Sensors: Cameras, infrared range finders, scanners, etc.

Actuators: Various motors, screen, printing devices.

A robotic agent substitutes cameras and infrared range finders for the sensors and various motors for the effectors.

Environment is the surrounding of the robotic agent. It is the area in which agent does the work, performs some actions. In a vacuum cleaner robotic agent, a room will be an environment. There are various types of environment fully observable or partially observable, Static or dynamic, Accessible or inaccessible, Known or unknown, Single agent or multi agent etc.

Agent perceives the signals of other robotic agent actions, temperature, humidity, pressure, air type, atmospheric conditions, climate, other surrounding conditions and many more. In addition to this, in taxi driving example – road conditions, traffic conditions, weather conditions, rain or smoke fog, wind, driving rules etc.

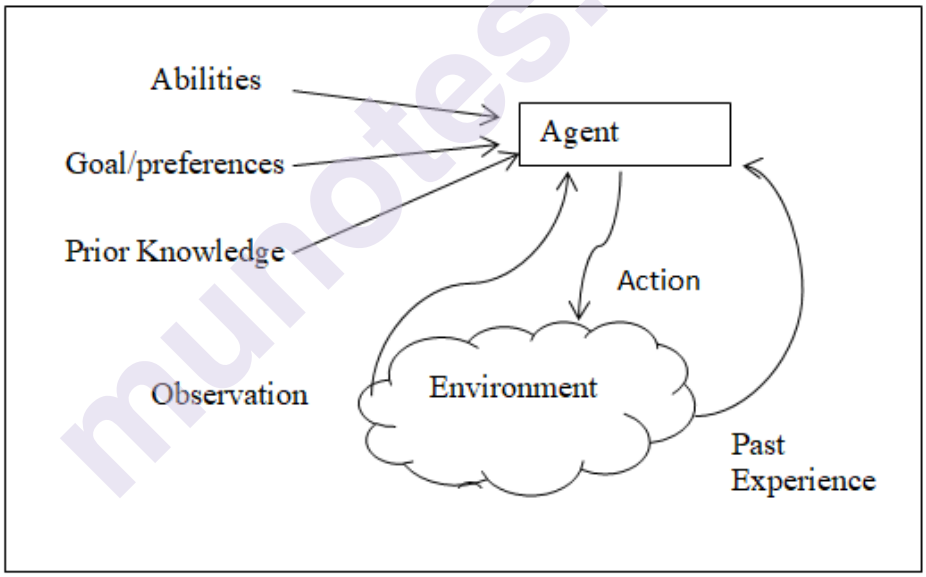
2.4 GOOD BEHAVIOR

Here, the behavior of the AI agent is checked. For this rationality, performance measure is considered.

2.4.1 Rational agent:

A rational agent is the one that does "right" things and acts rationally so as to achieve the best outcome even when there is uncertainty in knowledge.

Rational agent can be defined as an agent who makes use of its percept sequence, experience and knowledge to maximize the performance measures of an agent for every possible action. Performance measures can be any like Accuracy, Speed, Safety, total work done with respect to time, efficiency, saving money and time, following Legal rules, etc.



What is rational at any given time depends on four things:

1) The performance measure defines the degree of success. Performance is the measure of successful completion of any task.
2) Complete perceptual history is the percept sequence. This percept sequence is a sequence of signals or inputs that the agent has perceived so far.
3) What the agent knows about the environment. Especially about the environment dimensions, structure, basic laws, task, user, other agents, etc.
4) The actions that the agent can perform. It shows the capability of the agent.

Ideal rational agent:

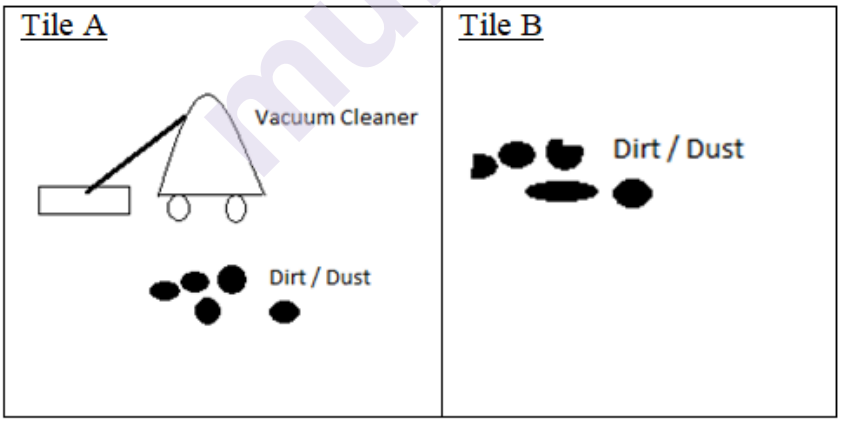
For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

2.4.2 The ideal mapping from percept sequences to actions:

For each percept signal what will be the associated action of the agent. This information can be described in tabular format.

Percept sequences / Input Signals sensed by sensor/ camera/ input device etc	Action
P1	A1
P2	A2
P3	A3
P4	A4
..	..
..	..
Pn	An

Let's Consider the Vacuum Cleaner robotic agent. It cleans the room. For simplicity two tiles are considered. The Vacuum cleaner agent is present on Tile A and observing the environment i.e. Tile A and Tile B both. Both the tiles have some dirt. This agent has to clean the dirt by sucking it.



The following table shows the percept sequence and its associated action.

Percept Sequence	Action
[A, Clean]	Move Right. i.e. Move to tile B
[A, Dirty]	Suck Dirt from tile A
[B, Clean]	Move Left. i.e. Move to tile A

Autonomous Agent:

An agent is autonomous to the extent that its action choices depend on its own experience, rather than on knowledge of the environment that has been built-in by the designer.

There are various autonomous agent are in development stage. Driverless car- Waymo, Google's Alexa, etc.

2.4.3 How and When to evaluate the agent's success:

Performance measure:

Performance measure is one of the major criteria for measuring success of an agent's performance.

As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.

Example: The performance measure of Vacuum-cleaner Robotic agent can depend upon various factors like it's dirt cleaning ability, time taken to clean that dirt, consumption of electricity, etc.

2.4.4 PEAS Properties of Agent:

What is PEAS?

PEAS: Performance, Environment, Actuators, Sensors.

Performance Measures - used to evaluate how well an agent solves the task at hand

Environment - surroundings beyond the control of the agent

Actuators - determine the actions the agent can perform

Sensors - provide information about the current state of the environment

Examples:

1) Automated Car Driving agent/Driverless Car:

Performance measures: Safety, Optimum speed, Comfortable journey, Maximize profits.

Environment: Roads, Traffic conditions, Clients.

Actuators: Steering wheel, Accelerator, Gear, Brake, Light signal, Horn.

Sensors: Cameras, Sonar system, Speedometer, GPS, Engine sensors, etc.

2) Medical Diagnosis system:

Performance measures: Healthy patient (Sterilized instruments), minimize costs.

Environment: Patients, Doctors, Hospital environment.

Actuators: Camera, Scanner (to scan patient's report).

Sensors: Body scanner, Organs scanner e.g. MRI, CT SCAN etc, Screen, Printer.

3) Soccer Player Robot:

Performance measures: Number of goals, speed, legal game.

Environment: Team players, opponent team players, playing ground, goal net.

Actuators: Joint angles, motors.

Sensors: Camera, proximity sensors, infrared sensors.

2.5 NATURE OF ENVIRONMENT OR PROPERTIES OF ENVIRONMENT

1) 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.

Example:

Accessible environment- An empty closed room whose state can be defined by its temperature, air pressure, humidity.

Inaccessible environment- Information about an event occurs in the atmosphere of the earth.

2) Deterministic vs. non deterministic:

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

In an accessible, deterministic environment an agent does not need to worry about uncertainty. If the environment is inaccessible, however, then it may appear to be nondeterministic. This is particularly true if the environment is complex, making it hard to keep track of all the inaccessible aspects. Thus, it is often better to think of an environment as deterministic or nondeterministic from the point of view of the agent.

3) Episodic vs. non episodic:

In an episodic environment, the agent's experience is divided into "episodes." Each episode consists of the agent perception and its corresponding action. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in

previous episodes. Episodic environments are much simpler because the agent does not need to think ahead. However, in a non-episodic environment, an agent requires memory of past actions to determine the next best actions.

4) Static vs. dynamic:

If the environment can change while an agent is deciding on an action, then we say the environment is dynamic for that agent; otherwise it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is semi dynamic.

Taxi driving is an example of a dynamic environment whereas Crossword puzzles are an example of a static environment.

5) Discrete vs. continuous:

If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Chess is discrete—there are a fixed number of possible moves on each turn. Taxi driving is continuous—the speed and location of the taxi and the other vehicles sweep through a range of continuous values.

Examples of environments and their nature or characteristics are as follows:

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive Maths tutor	No	No	No	No	Yes

2.6 STRUCTURE OF INTELLIGENT AGENT

2.6.1 Structure:

Structure of the agent describes how the agent works.

The job of AI is to design the agent program. Agent program is a function that implements the agent mapping from percepts to actions. An architecture or hardware is used to run this agent program.

The relationship among agents, architectures, and programs can be summed up as follows:

agent = architecture + program

2.6.2 Software Agents:

It is also referred to as softbots.

It lives in artificial environments where computers and networks provide the infrastructure.

It may be very complex with strong requirements on the agent

e.g. softbot designed to fly a flight simulator, softbot designed to scan online news sources and show the interesting items to its customers, World Wide Web, real-time constraints,

Natural and artificial environments may be merged user interaction sensors and actuators in the real world - camera, temperature, arms, wheels, etc.

2.6.3 Page:

PAGE is used for high-level characterization of agents.

P- Percepts - Information is acquired through the agent's sensory system.

A- Actions - Operations are performed by the agent on the environment through its actuators.

G- Goals - Desired outcome of the task with a measurable performance.

E- Environment - Surroundings beyond the control of the agent.

Following are some examples of agents and their PAGE descriptions.

Sr. No.	Agent Type	Percepts	Actions	Goals	Environment
1.	Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
2.	VacBot – Vacuum Cleaner Bot	tile properties like clean/dirt, empty/occupied movement and orientation	pick up dirt, move	desired outcome of the task with a measurable performance	surroundings beyond the control of the agent
3.	StudentBot	images (text,	comments, questions,	mastery of the material	classroom

		pictures, instructor, classmates) sound (language)	gestures note-taking (?)	performance measure: grade	
4.	Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
5.	Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
6.	Refinery controller	Temperature, pressure readings	Open, close valves; adjust temperature	Maximize purity, yield, safety	Refinery
7.	Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

2.6.4 Different types of Agent program:

Every agent program has some skeleton. Real agent program implements the mapping from percepts to action. There are four types of agent program as follows:

1) Simple reflex agents
2) Agents that keep track of the world
3) Goal-based agents
4) Utility-based agents

1) Type 1: Simple Reflex Agent:

Reflex agents respond immediately to percepts.

They choose actions only based on the current percept.

They are rational only if a correct decision is made only on the basis of current precept.

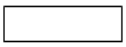
Their environment is completely observable.

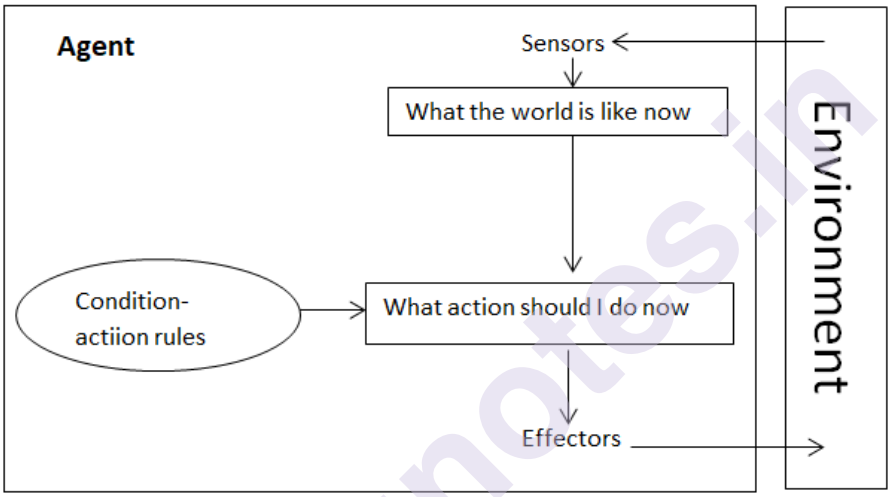
In the simple reflex agent, the correct decision can be made on the basis of the current percept.

Let's consider the driverless car – Car A. If the car B is running in front of car A, and suddenly if car B has applied brakes, and car B brake lights come ON, then driverless car agent should notice that and initiate breaking. Thus it can be specified in condition action rule and can be represented in the form as

"If condition Then Action".

"if car-in-front-is-breaking then initiate-braking".

Notation	Meaning
	the current internal state of the agent's decision process
	represent the background information used in the process



Schematic Diagram: Simple Reflex Agent

Following is the agent program basically it is a function for a simple reflex agent. It works by finding a rule whose condition matches the current situation (as defined by the percept) and then doing the action associated with that rule.

Function Simple_Reflex_Agent(percept) returns action

Static: rules, a set of condition-action rules

state ← Interpret_Input (percept)

rule ← Rule_Match (state, rules)

action ← Rule_Action [rule]

return action

This agent program calls the Interpret_Input() function that generates an abstracted description of the current state from the percept input, and the Rule_Match() function returns the first rule in the set of rules that matches the given state description. Such agents can be implemented very efficiently, but their range of applicability is very less.

2) Type 2: Agents that keep track of the world:

It is a reflex agent with internal state.

Internal State is a representation of unobserved aspects of current state depending on percept history.

Consider an example of a driverless car. If the car in front is a recent model, and has the centrally mounted brake light, then it will be possible to tell if it is braking from a single image. Unfortunately, older models have different configurations of tail lights, brake lights, and turn-signal lights, and it is not always possible to tell if the car is braking.

Thus, even for the simple braking rule, our driver will have to maintain some sort of internal state in order to choose an action. Here, the internal state is not too extensive—it just needs the previous frame from the camera to detect when two red lights at the edge of the vehicle go on or off simultaneously.

Consider one case in car driving: from time to time, the driver looks in the rear-view mirror to check on the locations of nearby vehicles. When the driver is not looking in the mirror, the vehicles in the next lane are invisible (i.e., the states in which they are present and absent are indistinguishable); but in order to decide on a lane-change maneuver, the driver needs to know whether or not they are there.

It happens because the sensors do not provide access to the complete state of the world. In such cases, the agent may need to maintain some internal state information in order to distinguish between world states that generate the same perceptual input but nonetheless are significantly different. Here, "significantly different" means that different actions are appropriate in the two states.

As time passes internal state information has to be updated. For this two kinds of knowledge have to be encoded in the agent program. First, we need some information about how the world evolves independently of the agent—for example, that an overtaking car generally will be closer behind than it was a moment ago. Second, we need some information about how the agent's own actions affect the world—for example, that when the agent changes lanes to the right, there is a gap (at least temporarily) in the lane it was in before.

Following Figure shows how the current percept is combined with the old internal state to generate the updated description of the current state.

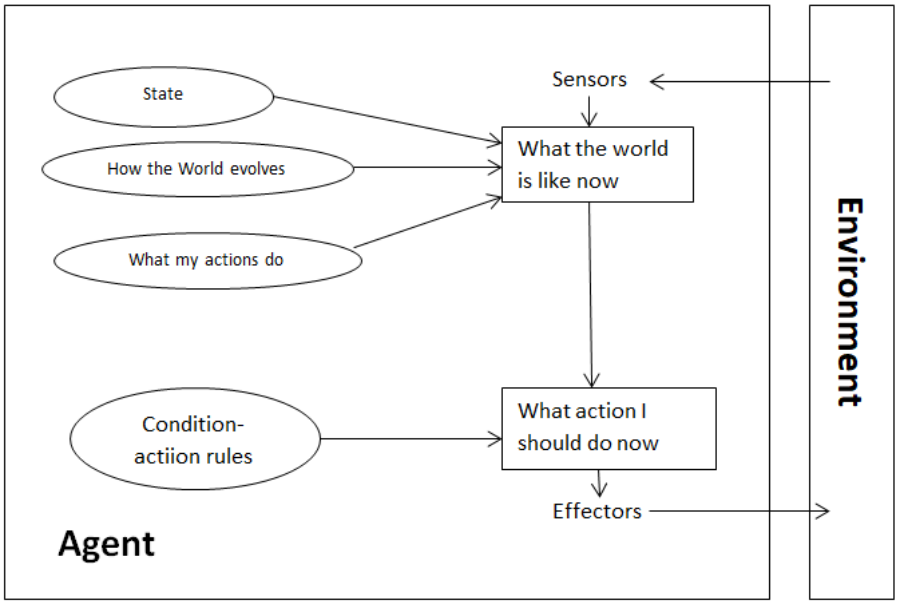


Diagram: Schematic Diagram for Agent that keep track of the world/ Reflex Agent with Internal State

Agent Program:

The interesting part is the function Update-State, which is responsible for creating the new internal state description. As well as interpreting the new percept in the light of existing knowledge about the state, it uses information about how the world evolves to keep track of the unseen parts of the world, and also must know about what the agent's actions do to the state of the world.

Following is the reflex agent with internal state. It is an agent that keeps track of the world. It works by finding a rule whose condition matches the current situation (as defined by the percept and the stored internal state) and then doing the action associated with that rule.

function Reflex-Agent-With-State(percept) returns action

static: state, a description of the current world state

rules, a set of condition-action rules

state ← Update-State (state, percept)

rule ← Rule-Match (state, rules)

action ← Rule-Action [rule]

state ← Update-State (state, action)

return action.

3) Type 3: Goal-based agents:

Goal-based agents act so that they will achieve their goal

If just the current state of the environment is known then it is not always enough to decide what to do.

The agent tries to reach a desirable state, the goal may be provided from the outside (user, designer, environment), or inherent to the agent itself

The agent performs the action by considering the Goal. To achieve the goal, agent has to consider long sequences of actions. Here, Search & Planning is required. Decision making is required. It involves consideration of the future—both "What will happen if I do such-and-such?" and "Will that make me happy?

Goal based agents can only differentiate between goal states and non goal states. Hence, their performance can be 100% or zero.

Goal-based agent is more flexible than reflex agent since the knowledge supporting a decision is explicitly modeled, thereby allowing for modifications.

Limitation: Once the goal is fixed, all the actions are taken to fulfill it. And the agent loses flexibility to change its actions according to the current situation.

Example: Vacuum cleaning Robot agent whose goal is to keep the house clean all the time. This agent will keep searching for dirt in the house and will keep the house clean all the time.

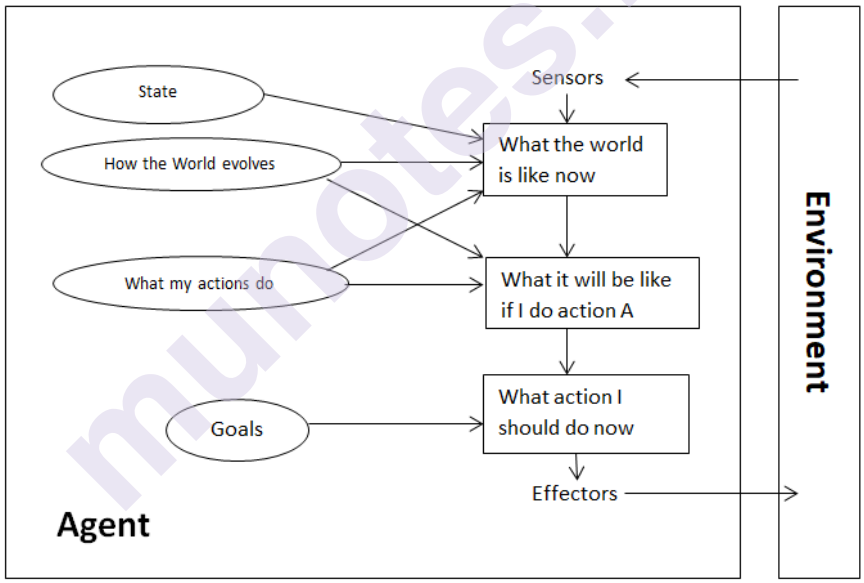


Diagram: Schematic Diagram for Goal Based Agent

4) Type 4: Utility-based agents:

Utility-based agents try to maximize their own "happiness."

There are conflicting goals, out of which only few can be achieved. Goals have some uncertainty of being achieved and you need to weigh likelihood of success against the importance of a goal.

Goals alone are not really enough to generate high-quality behavior. Here, the degree of happiness is considered.

For example, there are many action sequences that will get the taxi to its destination, thereby achieving the goal, but some are quicker, safer, more

reliable, or cheaper than others. Goals just provide a crude distinction between "happy" and "unhappy" states, whereas a more general performance measure should allow a comparison of different world states (or sequences of states) according to exactly how happy they would make the agent if they could be achieved. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher utility for the agent.

Utility is therefore a function that maps a state onto a real number, which describes the associated degree of happiness. A complete specification of the utility function allows rational decisions in two kinds of cases where goals have trouble. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate trade-off. Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed up against the importance of the goals

Utility function is used to map a state to a measure of utility of that state. We can define a measure for determining how advantageous a particular state is for an agent. To obtain this measure a utility function can be used. The term utility is used to depict how “HAPPY” the agent is to find out a generalization.

Examples: 1) Google Maps – A route which requires Least possible time.

1) Automatic Car: Should reach to the target location within least possible time safely and without consuming much fuel.

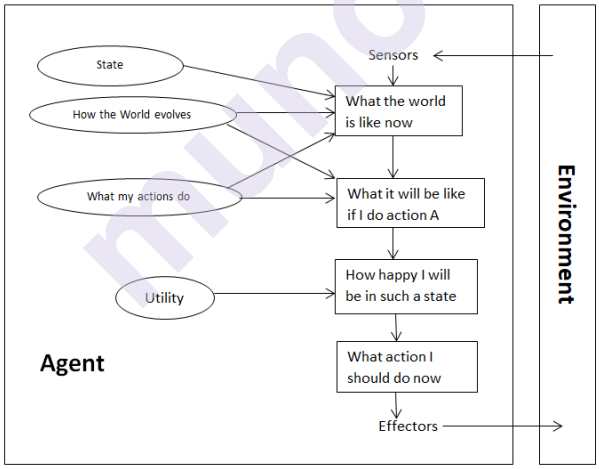


Diagram: Schematic Diagram for Utility Based Agent

2.7 SUMMARY

An agent is anything that perceives its environment through sensors and acting upon that environment through effectors.

A rational agent is the one that does “right” things and acts rationally so as to achieve the best outcome even when there is uncertainty in knowledge.

PEAS- Performance, Environment, Actuators, Sensors determines the behavior of an agent.

There are various nature of environment or properties of environment, they are as Accessible vs. inaccessible, Deterministic vs. non deterministic, Episodic vs. non episodic, Static vs. dynamic, Discrete vs. continuous, etc.

Structure of Intelligent Agent: agent = architecture + program.

PAGE is used for high-level characterization of agents.

Different types of Agent program are as Simple reflex agents, Agents that keep track of the world, Goal-based agents, Utility-based agents.

2.8 UNIT END QUESTIONS

1. Describe is an agent, rational Agent, autonomous agent.
2. Which are the four things that is considered to check the agent is rational at any given time?
3. What is the good behavior of agent. How performance measure is the key factor for
4. Which are the four things that is considered to check the rationality of agent at any given time.
5. How and When to evaluate the agent's success?
6. Describe Performance measure is one of the major criteria for measuring success of an agent's performance.
7. What is PEAS Properties of Agent? Explain with example the PEAS for any four agents.
8. Explain Nature of Environment or Properties of Environment
9. What is softbot?
10. What is PAGE? Describe with any five examples.
11. Explain the structure of Agent.
12. Which are the types of agent? Explain all the types with schematic diagram.
13. Write the Agent program for Simplex Reflex Agent and Reflex Agent with Internal State.

2.9 REFERENCES

Artificial Intelligence: A Modern Approach, 4th US ed.by Stuart Russell and Peter Norvig.

Deepak Khemani, “A first course in Artificial Intelligence”, McGraw Hill edition, 2013.

Patrick Henry Winston , “Artificial Intelligence”, Addison-Wesley, Third Edition.

*****

UNIT II

3

SOLVING PROBLEMS BY SEARCHING

Unit Structure

3.1 OBJECTIVES

After this chapter, you should be able to understand the following concepts:

1. Understand how to formulate a problem description.
2. Understand and solve some problems of Artificial Intelligence.
3. Know about uninformed search and based algorithms.
4. Understand informed search and based algorithms.
5. Know about heuristic functions and strategies.

3.2 INTRODUCTION

A problem-solving agent firstly formulates a goal and a problem to solve. Then the agent calls a search procedure to solve it. It then uses the solution to guide its actions. It will do whatever the solution recommends as the next thing to do and then remove that step from the sequence. Once the solution has been executed, the agent will formulate a new goal. Searching techniques like uninformed and informed or heuristic use in many areas like theorem proving, game playing, expert systems, natural language processing etc. In search technique, firstly select one option and leave other option if this option is our final goal, else we continue selecting, testing and expanding until either solution is found or there are no more states to be expanded.

3.3 PROBLEM SOLVING AGENTS

It is very important task of problem domain formulation. Problems are always dealt with an Artificial Intelligence. It is commonly known as state. For solving any type of task (problem) in real world one needs formal description of the problem. Each action changes the state and the goal is to find sequence of actions and states that lead from the initial start state to a final goal state.

Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider. Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

Problem formulation is the process of deciding what actions and states to consider, given a goal. the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action. If the agent has no additional information—i.e., if the environment is unknown.

An agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

The agent's task is to find out how to act, now and in the future so that is reaches a goal state. Before it can do this, it needs to decide what sorts of actions and states it should consider.

3.3.1 Well Defined problem:

A problem can be defined formally by four components.

1) The initial state that the agent starts in:

For example, Consider a agent program Indian Traveller developed for travelling **Hyderabad** to Mumbai travelling through different states. The initial state for this agent can be described as **In (Hyderabad).**

2) A description of the possible actions available to the agent:

Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is applicable in s. For example, from state In (Mumbai), the applicable actions are **{Go (Solapur), Go (Osmanabad), Go (Vijayawada)}**

3) The goal test:

Which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

For example, In Indian traveller problem the goal is to reach Mumbai i.e it is a singleton set. **{In (Mumbai)}**

Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape.

4) **A path cost function** that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Mumbai, time is of the essence, so the cost of a path might be its length in kilometres. The step cost of taking action a in state x to reach state y is denoted by c (x, a, y).

A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

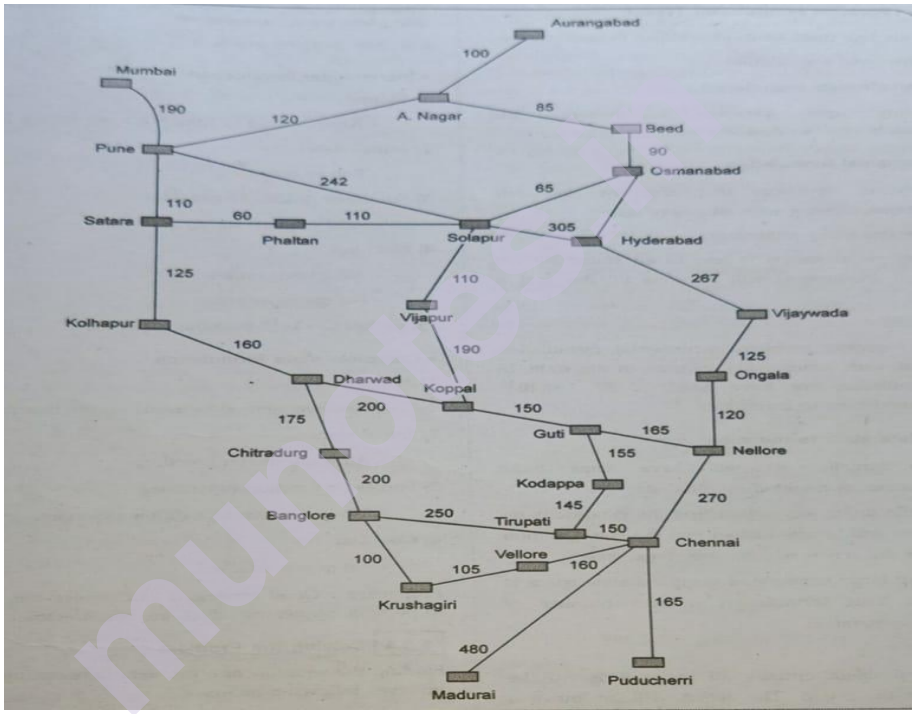


Fig. 3.2.1 Indian traveller map

3.4 EXAMPLES PROBLEMS

The problem-solving approach has been applied to a vast array of task environments. We list some known problems like Toy problem and real-world problem. A **toy problem** is intended to illustrate or exercise various problem-solving methods. This problem gives exact description so it is suitable for researchers to compare the performance of algorithms. **A real-world problem** is one whose solution people actually care about.

3.4.1 Toy Problems:

The first example we examine is the vacuum world.

This can be formulated as problem as follows:

1) States:

The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt.

2) Initial state:

Our vacuum can be in any state of the 8 states shown in the figure 3.2.

3) Actions:

Each state has just three actions: Left, Right, and Suck

4) Transition Model:

The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.

5) Goal test:

No dirt at all locations.

6) Path cost:

Each cost step is 1.

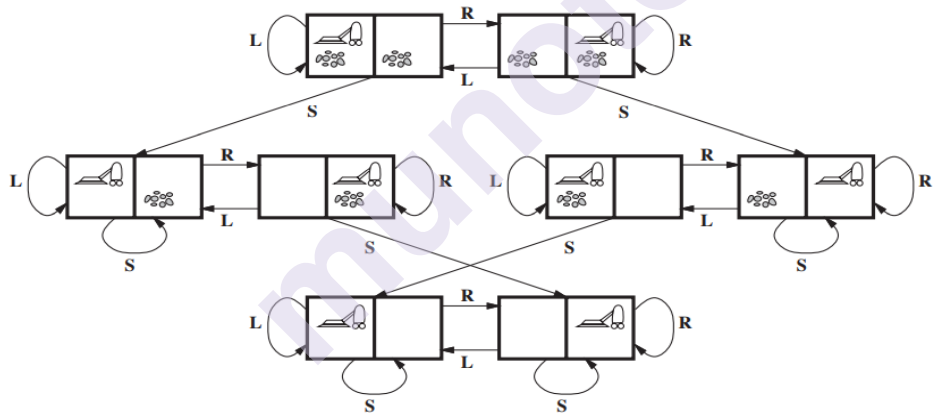


Fig. 3.2 The state space for the vacuum world. Links denote actions: L = Left, R= Right, S= Suck

The second example we examine is 8 puzzle problem.

The 8-puzzle consists of eight numbered, movable tiles set in a 3x3 frame. Each tile has a number on it. A tile that is adjacent to the blank space can slide into the space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.

This can be formulated as problem as follows:

1) States:

Location of eight tiles and blank.

2) Initial state:

Any state can be designed as the initial state.

3) Actions:

Move blank left, right, up or down.

4) Goal test:

This checks whether the states match the goal configuration.

5) Path cost:

Each step cost is 1.

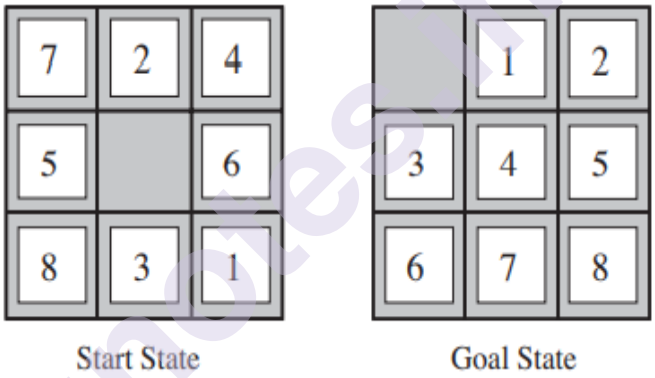


Fig.3.3 An instance of 8-puzzle problem

The third example is 8-Queen problem:

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

This can be formulated as problem as follows:

1) States:

Any arrangement of 0 to 8 queens on the board is a state

2) Initial state:

No queens on board.

3) Actions:

Add a queen to any empty square.

4) Transition Model:

Returns the board with a queen added to the specified square.

5) Goal Test:

8 queens are on the board, none attacked.

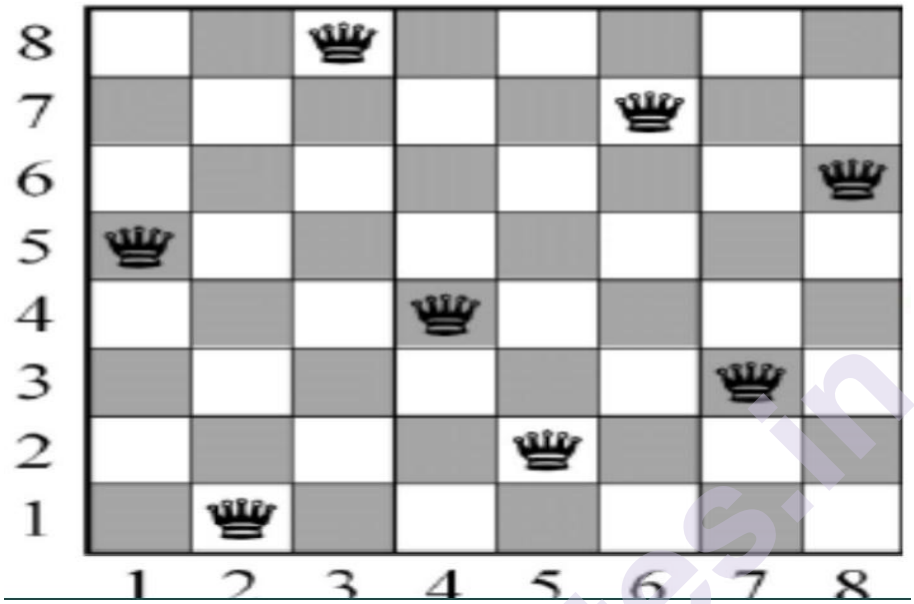


Fig. 3.4 Solution to the 8-queen problem

The fourth example is Water Jug problem:

You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured.

Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State – 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 0 d" x d" 4, and 0 d" y d" 3. Our initial state: (0,0). The goal state is à (2, n).

Rules:

1.	Fill the 4-Gallon Jug	(x,y)→ (4,y)
2.	Fill the 3-Gallon Jug.	(x,y)→ (x,3)
3.	Pour some water out of 4 Gallon jug	(x,y)→(x-d,y)
4.	Pour some water out of 3 Gallon jug	(x,y)→(x,y-d)
5.	Empty 4 Gallon jug on the ground.	(x,y)→(0,y)
6.	Empty 3 Gallon jug on the ground	(x,y)→(x,0)

7. Pour some water from 3 Gallon jug into the 4 Gallon jug until the 4 gallon jug is full. $(x,y) \rightarrow (4, y - (4 - x))$
8. Pour some water from 4 Gallon jug into the 3 Gallon jug until the 3 gallon jug is full. $(x,y) \rightarrow (x - (3-y), 3)$
9. Pour all water from 3 to 4 gallon. $(x,y) \rightarrow (x+y, 0)$
10. Pour all water from 4 to 3 Gallon jug. $(x,y) \rightarrow (0, x+y)$
11. Pour 2 gallon from 3 G to 4 G. $(0,2) \rightarrow (2,0)$ if $x<y$
12. Empty the 2 G in 4 G on the ground. $(2,y) \rightarrow (0,y)$ if $x<y$

Gallons in the 4-Gallon Jug	Gallons in 3-Gallon Jug	Rules Applied
0	0	-
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5
2	0	9

The fifth example is Missionaries and Cannibals:

Three missionaries and their three cannibals are present at one side of a river and need to cross the river. There is only one boat available.

At any point of time the number of cannibals should not outnumber the number of missionaries at that bank.

It is also known that only two persons can occupy the boat available at a time. Let Missionary is denoted by 'M' and cannibal by 'C'

Rules:

1. (0, M): One missionary sailing the boat from bank-1 to bank-2.
2. (M,0): One missionary sailing boat from bank-2 to bank-1.
3. (M, M): Two missionaries sailing the boat from bank 1 to bank 2.
4. (M, M): Two missionaries sailing the boat from bank 2 to bank 1.
5. (M, C): One missionary & one cannibal sailing the boat from bank 1 to bank-2.
6. (C, M): One missionary & one cannibal sailing the boat from bank 2 to bank-1.
7. (C, C): Two cannibals sailing the boat from bank-1 to bank-2.
8. (C, C): Two cannibals sailing the boat from bank-2 to bank-1.
9. (0, C): One cannibal sailing the boat from bank-1 to bank-2.

10. (C,0): One cannibal sailing the boat from bank-2 to bank-1.

SSS

After application of rules	Person in the river bank -1	Person in the river bank -2	Boat Position
Start state	M, M, M, C, C, C	0	Bank-1
5	M, M, C, C	C, M	Bank-2
2	M, M, M, C, C	C	Bank-1
7	M, M, M	C, C, C	Bank-2
10	M, M, M, C	C, C	Bank-1
3	M, C	C, C, M, M	Bank-2
6	M, C, M, C	C, M	Bank-1
3	C, C	C, M, M, M	Bank-2
10	C, C, C	M, M, M	Bank-1
7	C	C, C, M, M, M	Bank-2
10	C, C	C, M, M, M	Bank-1
7	0	M, M, M, C, C, C	Bank-2

3.4.2 Real-World Problems:

Example 1: Route finding problem:

A route-finding problem is defined in terms of specified locations and transitions along links between them, such as web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Indian map example. Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems.

Ex. Airline travel problem specified as follows:

1) States:

Each state obviously includes a location (e.g., an airport) and the current time.

2) Initial state:

This is specified by the user's query.

3) Actions:

Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.

4) Transition Model:

The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time

5) Goal test:

Are we at the final destination specified by the user?

6) Path cost:

This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

Touring Problem:

This is closely related to route-finding problems, but with an important difference. Consider, for example, the problem visits every city in Fig. 3.1 at least once. Starting and ending at Solapur. The actions correspond to trips between adjacent cities. Each state must include not just the current location but also the set of cities the agent has visited. So, the initial state would be In (Solapur), visited (Solapur). and the goal test would check whether the agent is in Bucharest and all 27 cities have been visited.

Example 2: Travelling salesman problem (TSP problem)

It is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.

1) States:

Cities or locations

Each city must be visited exactly once. Visited cities must be kept as state information.

2) Initial state:

Starting point, no cities visited.

3) Successor function:

Move from one city to another city.

4) Goal test:

All cities visited, Agent at the initial location.

5) Path cost:

Distance between cities.

Example 3: VLSI Layout Problem:

problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.

1) States:

Position of components, wires on a chip.

2) Initial State:

Incremental: no components placed

Complete-state: All components place (randomly, manually)

3) Successor Function:

Incremental: placed components, route wire

Complete-state: move components, move wire

4) Goal test:

All components placed; components connected as specified

5) Path cost:

Complex.

Example 4: Robot Navigation

It is a generalization of the route-finding problem described earlier. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) 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 must also be controlled, the search space becomes many-dimensional.

1) States:

Locations, Position of actuators.

2) Initial state:

Start position

3) Successor Function:

Movement, action of actuators

4) Goal Test:

Task-dependent

5) Path cost:

complex

Example 5: Automatic assembly sequencing

The aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation.

1) States:

Location of components

2) Initial state:

No components assembled

3) Successor function:

Place component

4) Goal test:

System fully assembled

5) Path cost:

Number of moves

3.5 SEARCHING FOR SOLUTIONS

A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem. A search problem where we aim not only at reaching our goal but also at doing so at minimal cost is an optimisation problem.

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

3.6 UNINFORMED SEARCH

This topic covers several search strategies that come under the heading of uninformed search or blind search. This search algorithm uses only initial state, search operators and a test for a solution. It proceeds systematically by exploring nodes either randomly or in some predetermined order.

3.6.1 Breadth First Search (BFS):

It is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.

All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

It starts at the tree root and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

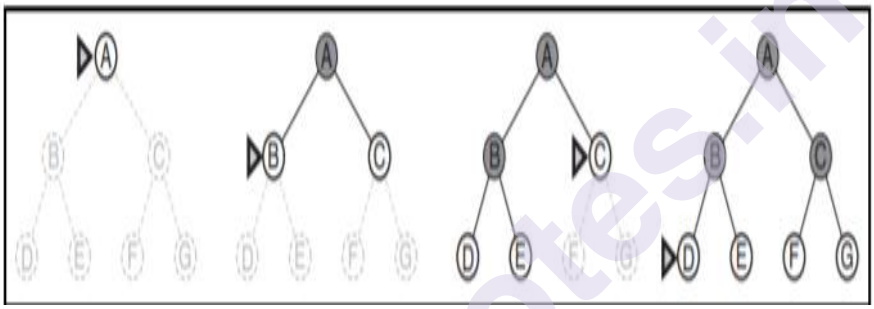


Fig. 3.5 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker

```
Function Breadth First Search (start)
Begins
Open ← start
Closed ←[ ];
While(open # [ ]) do
      Begin
            X ← remove first [open]
            If x is goal then return success
            Else begin
                  Right end (open) ← generated child(x)
                  Closed ← x;
                  End
                        End
            Return fail
            End
```

The BFS algorithm, presented above, explore the space in a level-by-level fashion. It maintains two lists namely open and closed. Open list contains states which have been generated but whose children have not been expanded and closed list records states that have already been examined. Open is maintained as a queue First in first out (FIFO).

Time complexity: Equivalent to the number of nodes traversed in BFS until the shallowest solution.

Space complexity: Equivalent to how large can the fringe get.

Completeness: BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.

Optimality: BFS is optimal as long as the costs of all edges are equal.

3.6.2 Depth First Search (DFS):

The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

DFS traverses the tree deepest node first. It would always pick the deeper branch until it reaches the solution. The strategy can be implemented by tree search with LIFO (Last in First Out) queue, most popularly known as a stack.

```
Function Breadth First Search (start)
Begins
Open ← start
Closed ←[ ];
While (open # [ ]) do
    Begin
        X ← remove first [open]
        If x is goal then return success
        Else begin
            Left end (open) ← generated child(x)
            Closed ← x;
            End
                End
        Return fail
        End
```

DFS always expands the deepest nodes in the current fringe of the search tree. The search proceeds immediately to the deepest level of the search tree.

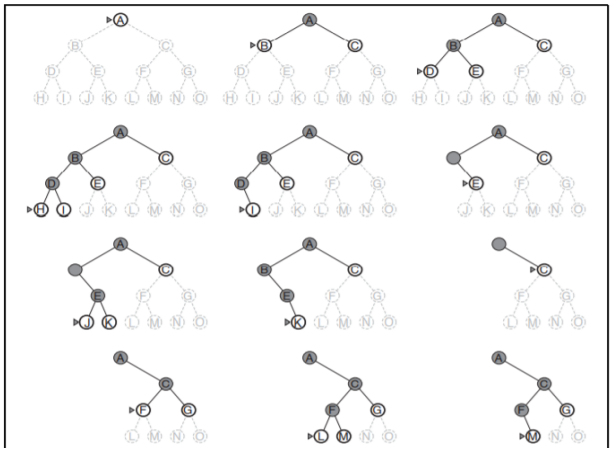


Fig. 3.6 Depth-first search on a binary tree

3.6.3 Uniform Cost Search:

When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n).

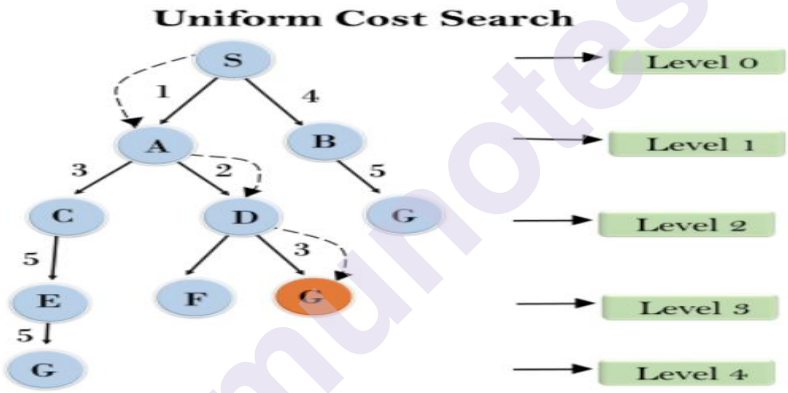


Fig. 3.7 Uniform cost search on a binary tree

Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions.

Uniform-cost search is guided by path costs rather than depths.

The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.

Uniform-cost search expands nodes according to their path costs form the root node.

It can be used to solve any graph/tree where the optimal cost is in demand.

A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost.

Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

3.6.4 Depth Limited Search:

The problem of Depth First Search can be solved by supplying DFS with a predetermined depth limit L. Nodes at depth L are treated as if they have no successors. It shows the infinite path problem. This approach introduces an additional source of incompleteness when L<d i.e. the shallowest goal is beyond the depth limit L.

Depth-limited search can be terminated with two Conditions of failure:

Standard failure value: It indicates that problem does not have any solution.

Cutoff failure value: It defines no solution for the problem within a given depth limit.

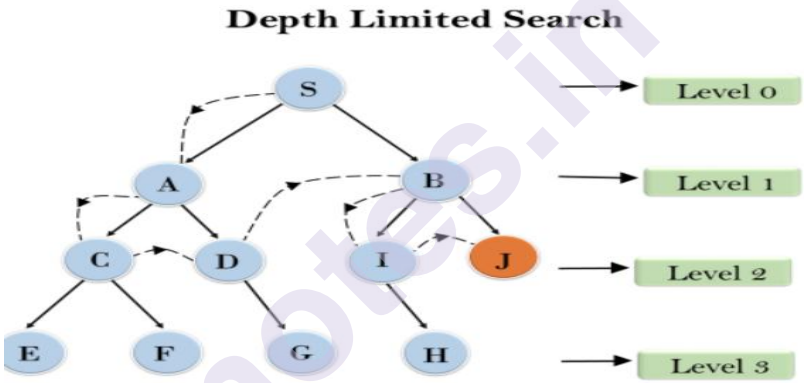


Fig. 3.8 Depth limited search tree

In the above example start node is S, Goal node is J and limit is 2.

Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is **O(b^{ℓ})**.

Space Complexity: Space complexity of DLS algorithm is O**(b×ℓ)**.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

3.6.9 Iterative Deepening Depth-First:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Function iterative deepening search (problem): returns solution or failure

Begin

Inputs, problem, depth

For depth 0 to ∞ do

Result <- depth – limited – search (problem, depth)

If result # cut-off then return result.

End.

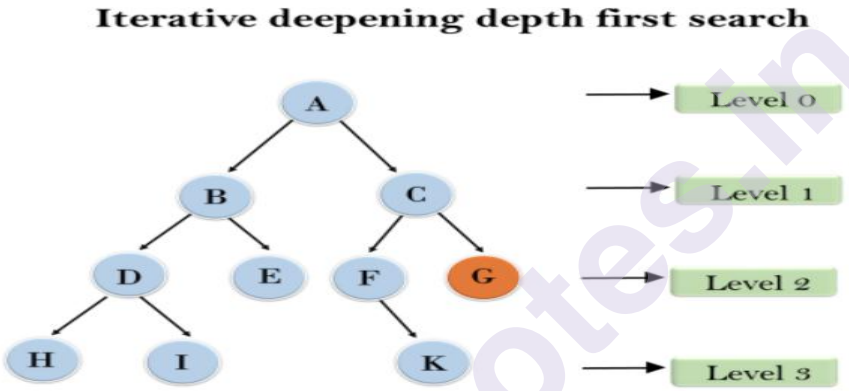


Fig. 3.9 Iterative deepening depth first search tree

1'st Iteration → A

2nd Iteration → A, B, C

3rd Iteration → A, B, D, E, C, F, G

4th Iteration → A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is if the branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.

Space Complexity:

The space complexity of IDDFS will be **O(bd)**.

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

3.6.10 Bidirectional Search:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.

Bidirectional search substitutes one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other. Bidirectional search can use search techniques such as BFS, DFS, DLS, etc

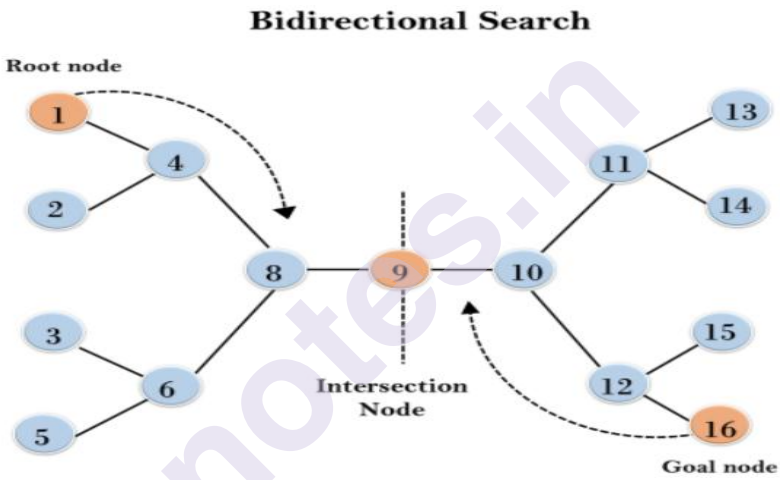


Fig. 3.10 Bidirectional search tree

In the above search tree, bidirectional search algorithm is applied.

This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction. The algorithm terminates at node 9 where two searches meet.

Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is **$O(b^d)$**.

Space Complexity: Space complexity of bidirectional search is **$O(b^d)$**.

Optimal: Bidirectional search is Optimal.

3.7 INFORMED SEARCH STRATEGIES

In this topic we will see more efficient search strategy, the informed search strategy or Heuristic Search. It is a search strategy that uses problem-specific knowledge beyond the definition of the problem itself. The term Heuristic is used for algorithms which find solutions among all possible ones. Heuristic is a rule of thumb or judgement technique that leads to a solution but provides no guarantee of success.

3.7.1 Greedy Best-First Search:

Best First Search is a combination of depth-first (DFS) and breadth-first (BFS) search methods. It is used to select the single path at a time which is more promising one than the current path. The most promising node is chosen for expansion. It evaluates nodes by using just the heuristic function; that is, f(n) = h(n).

Best first search algorithm:

Step 1: Put the initial node in OPEN.

Step 2: Pick the best node on OPEN, use heuristic method to find the promising node, generate its successor.

Follow the steps to each of its successor, do the step 2 till OPEN is not empty and the goal is not achieved.

Step 3: Evaluate the non-generated nodes so far add them to the list of OPEN and record their parents. If they have been generated before, change the parent if new path is better than previous one.

Advantages:

Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.

This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

It can behave as an unguided depth-first search in the worst-case scenario.

It can get stuck in a loop as DFS.

This algorithm is not optimal.

State	H(n)
S	13
A	12
B	4
C	7
D	3
E	8

F	2
H	4
I	9
G	0

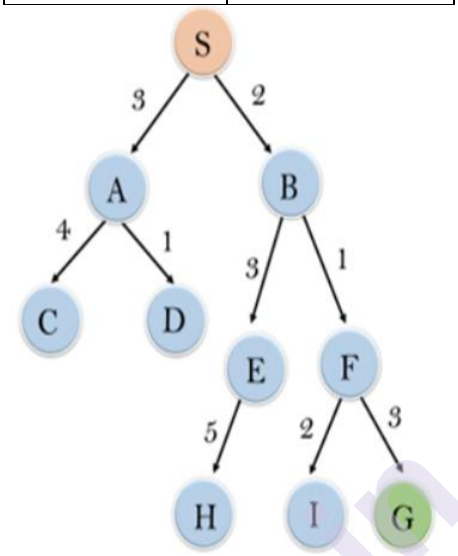


Fig. 3.11 Greedy best first search graph.

Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

Time Complexity: The worst-case time complexity of Greedy best first search is $O(b^m)$.

Space Complexity: The worst-case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

3.7.2 A* search:

A* combines the value of the heuristic function h(n) and the cost to reach the node n, g(n).

F(n)= g(n)+h(n)

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.

It has combined features of UCS and greedy best-first search, by which it solves the problem efficiently. A * search algorithm finds the shortest path through the search space using the heuristic function.

This search algorithm expands less search tree and provides optimal result faster.

A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

Algorithm of A* Algorithm:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node, then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Goto to **Step 2**.

Advantages:

A* search algorithm is the best algorithm than other search algorithms.

A* search algorithm is optimal and complete.

This algorithm can solve very complex problems.

Disadvantages:

It does not always produce the shortest path as it mostly based on heuristics and approximation.

A* search algorithm has some complexity issues.

The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

State	H(n)
S	5
A	3
B	4
C	2
D	6
G	0

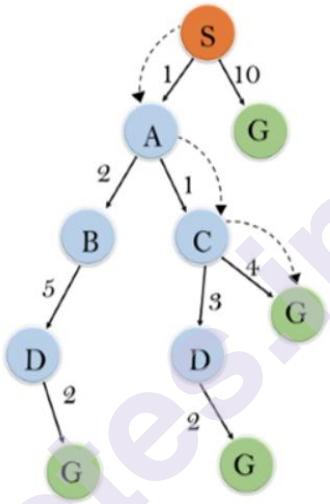


Fig. 3.12 A * search graph

Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

Complete: A* algorithm is complete as long as:

Branching factor is finite.

Cost at every action is fixed.

Time complexity: It depends upon heuristic function. the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O(b^d).**

Optimal: A* search algorithm is optimal.

1	2	3
8		4
7	6	5

3.8 HEURISTIC FUNCTIONS

Heuristic function estimates the cost of an optimal path between a pair of states in a single agent path-finding problem. Heuristic helps in solving problems, even though there is no guarantee that is will never lead in the wrong direction.

A good heuristic function is determined by its efficiency. More is the information about the problem, more is the processing time.

Some toy problems, such as 8-puzzle, 8-queen, tic-tac-toe, etc., can be solved more efficiently with the help of a heuristic function.

Consider the following 8-puzzle problem where we have a start state and a goal state. Our task is to slide the tiles of the current/start state and place it in an order followed in the goal state. There can be four moves either **left, right, up, or down**.

1	2	3
8	6	
7	5	4

Start state Goal state

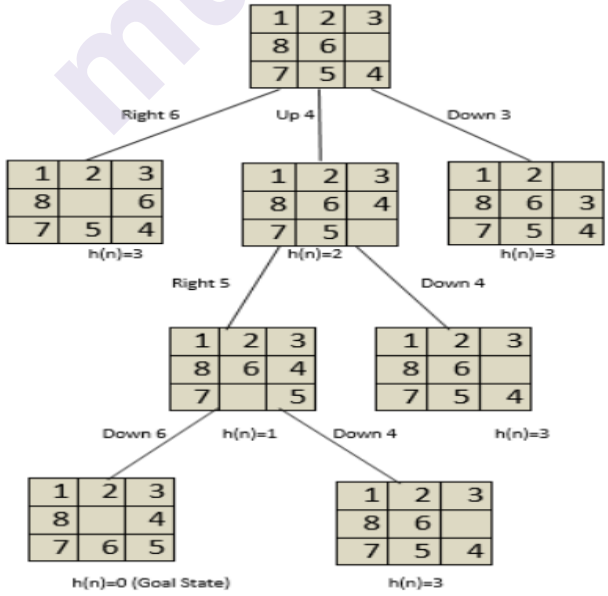


Fig. 3.13 8-puzzle solution

3.9 SUMMARY

To solve Artificial Intelligence problem, we need to follow certain steps. Firstly, define the problem precisely it means specify the problem space, the operators for moving within the space and the starting and goal state. Secondly, analyse the problem and finally, select one or more techniques for representing knowledge and for problem solving and apply it to the problem.

A problem consists of five parts: the initial state, a set of actions, a transition model describing the results of those actions, a goal test function, and a path cost function. The environment of the problem is represented by a state space. A path through the state space from the initial state to a goal state is a solution

Uninformed search methods have access only to the problem definition. The basic algorithms are as follows:

- Breadth-first search expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity.
- Uniform-cost search expands the node with lowest path cost, g(n), and is optimal for general step costs.
- Depth-first search expands the deepest unexpanded node first. It is neither complete nor optimal, but has linear space complexity. Depth-limited search adds a depth bound.
- Iterative deepening search calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal for unit step costs, has time complexity comparable to breadth-first search, and has linear space complexity.
- Bidirectional search can enormously reduce time complexity, but it is not always applicable and may require too much space.
- Informed search methods may have access to a heuristic function h(n) that estimates the cost of a solution from n.
- The generic best-first search algorithm selects a node for expansion according to an evaluation function.
- Greedy best-first search expands nodes with minimal h(n). It is not optimal but is often efficient.
- A∗ search expands nodes with minimal f(n) = g(n) + h(n). A∗ is complete and optimal, provided that h(n) is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH).

3.10 EXERCISES

1. Compare uniformed search with informed search.
2. What are the problems with Hill climbing and how can they be solved

3. State the best first search algorithm.
4. Explain advantages and disadvantages of DFS and BFS.
5. You have two jugs, measuring 8 gallons, 5 gallons and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly three gallons.
6. Three towers labelled A, B and C are given in which tower A has finite number of disks (n) with decreasing size. The task of the game is to more the disks from tower A to tower C using tower B as an auxiliary. The rules of the game are as follows:
   1. Only one disk can be moved among the towers at any given time.
   2. Only the "top" disk can be removed.
   3. No large disk can site over a small disk.

   Solve the problem and describe the following terms:

   a) State b) Initial state c) Goal state d) Operators e) Solution
7 Write down the heuristic function for 8-puzzle problem and solve the problem.
8 What is the use of online search agent in unknown environment?
9 Define the following terms in reference to Hill Climbing

   a) Local Maximum

   b) Plateau

   c) Ridge
10 Using suitable example, illustrate steps of A * search. Why is A* search better than best first search?
11 Describe A* search algorithm. Prove that A* is complete and optimal

3.11 BIBLIOGRAPHY

- Deva Rahul, Artificial Intelligence a Rational Approach, Shroff, 2014.
- Khemani Deepak, A First course in Artificial Intelligence, McGraw Hill, 2013.
- Chopra Rajiv, Artificial Intelligence a Practical Approach, S. Chand, 2012.
- Russell Stuart, and Peter Norvig Artificial Intelligence a Modern Approach, Pearson, 2010.
- Rich Elaine, et al. Artificial intelligence, Tata McGraw Hill, 2009.

*****

4

BEYOND CLASSICAL SEARCH

Unit Structure

4.1 OBJECTIVES

After this chapter, you should be able to:

- Understand the local search algorithms and optimization problems.
- Understand searching techniques of non-deterministic action.
- Familiar with partial observation.
- Understand online search agents and unknown environments.

4.2 INTRODUCTION

This chapter covers algorithms that perform purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it. The family of local search algorithms includes methods inspired by statistical physics (simulated annealing) and evolutionary biology (genetic algorithms).

The search algorithms we have seen so far, more often concentrate on path through which the goal is reached. But the problem does not demand the path of the solution and it expects only the final configuration of the solution then we have different types of problem to solve.

Local search algorithm operates using single path instead of multiple paths. They generally move to neighbours of the current state. Local algorithms use very little and constant amount of memory. Such kind of algorithms have ability to figure out reasonable solution for infinite state spaces.

They are useful for solving pure optimization problems. For search where path does not matter, Local search algorithms can be used, which operates

by using a single current state and generally move only to neighbours of the state.

4.3 LOCAL SEARCH ALGORITHMS

- Algorithms that perform purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state.
- These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it.
- The family of local search algorithms includes methods inspired by statistical physics **(simulated annealing)** and evolutionary biology **(genetic algorithms).**
- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path.
- When a goal is found, the path to that goal also constitutes a solution to the problem.
- In many problems, however, the path to the goal is irrelevant.
- Local search algorithms operate using CURRENT NODE a single current node (rather than multiple paths) and generally move only to neighbors of that node.
- Local Search Algorithm use very little memory—usually a constant amount; and they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- It is also useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.
- **Hill Climbing** is a technique to solve certain optimization problems.
- In these techniques, we start with a sub-optimal solution and the solution is improved repeatedly until some condition is maximized.
- The idea of starting with a sub-optimal solution is compared to walking up the hill, and finally maximizing some condition is compared to reaching the top of the hill.

Algorithm:

1) Evaluate the initial state. If it is goal state then return and quit.
2) Loop until a solution is found or there are no new operators left.
3) Select & apply new operator.
4) Evaluate new state

If it is goal state, then quit.

If it is better than current state then makes it new current state.

If it is not better than current then go to step 2.

4.3.1 Hill Climbing search:

It is so called because of the way the nodes are selected for expansion. At each point search path, successor node that appears to lead most quickly to the top of the hill is selected for exploration. It terminates when it reaches a "peak" where no neighbour has a higher value. This algorithm doesn't maintain a search tree so the data structure for the current node need only record the state and the value of the objective function. Hill Climbing Algorithm is sometimes called as a greedy local search because it grabs a good neighbour state without thinking ahead about where to go next. Unfortunately, hill climbing suffers from following problems:

This is shown in Figure 4.2.1

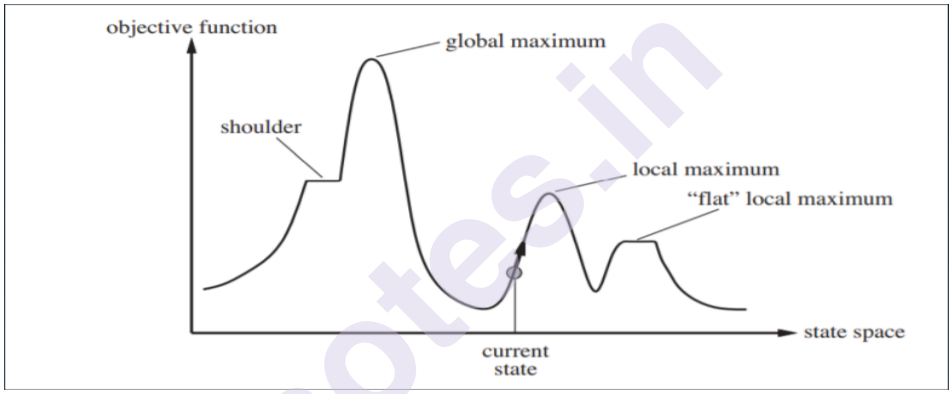


Fig. 4.1 Hill Climbing

- **Local maxima:**

Local maximum is a state which is better than its neighbour states, but there is also another state which is higher than it. They are also called as foot-hills. It is shown in Fig. 4.2.2

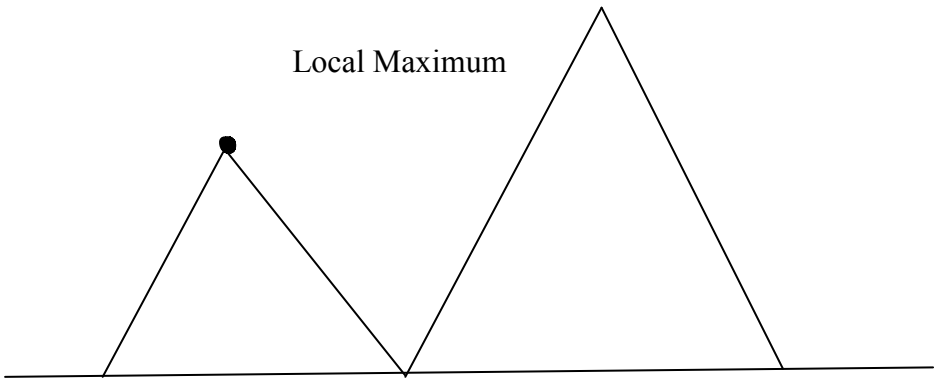


Fig. 4.2 Local maximum or Foot Hill

Solution to this problem:

Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

- **Plateau:**

It is a flat area of the search space in which a whole set of neighbouring states (nodes) have the same heuristic value. A plateau is shown in Fig. 4.3.

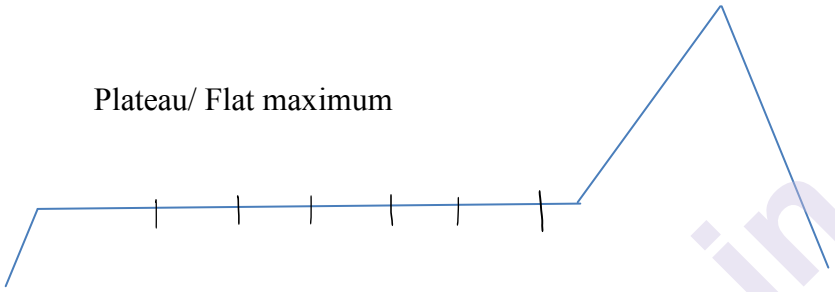


Fig 4.3 Plateau

Solution to this problem:

The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.

Another solution is to apply small steps several times in the same direction.

- **Ridge:**

It's an area which is higher than surrounding states but it cannot be reached in a single move. It is an area of the search space which is higher than the surrounding areas and that itself has a slope. Ridge is shown in Fig. 4.4.

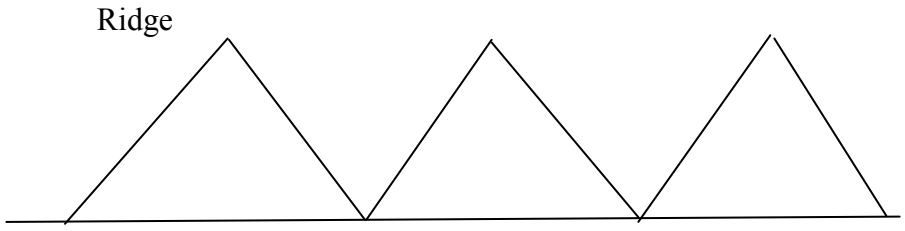


Fig. 4.4 Ridge

Solution to this problem:

Trying different paths at the same time is a solution. With the use of bidirectional search, or by moving in different directions, we can improve this problem.

Advantages of Hill Climbing:

It can be used in continuous as well as discrete domains.

Disadvantages of Hill Climbing:

It is not an efficient method.

It is not suitable for those problems whose value of heuristic function drops off suddenly when solution may be in sight.

It is local method as it looks at the immediate solution and decides about the next step to be taken rather than exploring all consequences before taking a move.

4.3.2 Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. Simulated Annealing is an algorithm which yields both efficiency and completeness. In mechanical term Annealing is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state.

The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

Algorithm:

1. Evaluate the initial state. If it is also goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialize BEST-SO-FAR to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state.
    a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
    b) Evaluate the new state. Compute

Δ E = (value of current)- (value of new state)

- If the new state is a goal state, then return it and quit.
- If it is not goal state but is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.
- If it is not better than current state, then make it the current state with probability p' as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0, 1]. If that number is less than p' then the move is accepted. Otherwise, do nothing.

c) Revise T as necessary according to the annealing schedule.

5. Return BEST-SO-FAR, as the answer.

To implement this revised algorithm, it is necessary to select an annealing schedule, which has three components. The first is the initial value to be used for temperature. The second is the criteria that will be used to decide when the temperature of the system should be reduced. The third is a amount by which the temperature will be reduced each time it is changed. There may also be a fourth component of the schedule, namely, when to quit. Simulated annealing is often used to solve problems in which the number of moves from a given sate is very large. For such problems, it may not make sense to try all possible moves.

4.3.3 Local beam search:

In this algorithm, it holds k number of states at any given time. At the start, these states are generated randomly. The successors of these k states are computed with the help of objective function. If any of these successors is the maximum value of the objective function, then the algorithm stops. Otherwise, the (initial k states and k number of successors of the states = 2k) states are placed in a pool. The pool is then sorted numerically. The highest k states are selected as new initial states. This process continues until a maximum value is reached. function Beam Search (problem, k), returns a solution state.

Start with k randomly generated states

Loop

Generate all successors of all k states

If any of the states =solution, then return the state

Else select the k best successors

End

4.3.4 Genetic algorithms:

Genetic Algorithms (GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are

intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate "survival of the fittest" among individual of consecutive generation for solving a problem. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Steps of genetic algorithms:

Initial Population:

The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution). In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

Fitness Function:

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Selection:

The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover:

It is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.

Mutation:

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.

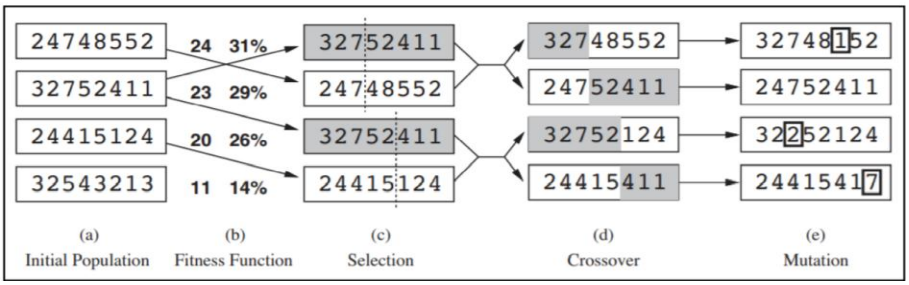


Fig. 4.5 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e)

4.4 SEARCHING WITH NON-DETERMINISTIC ACTION

When the environment is either partially observable or nondeterministic (or both), precepts become useful. When the environment is nondeterministic, precepts tell the agent which of the possible outcomes of its actions has actually occurred. In both cases, the future precepts cannot be determined in advance and the agent's future actions will depend on those future precepts. So, the solution to a problem is not a sequence but a contingency plan (also known as a strategy) that specifies what to do depending on what precepts are received.

4.4.1 The erratic vacuum world:

There are three actions- Left, right and Suck. And the goal is to clean up all the dirt.

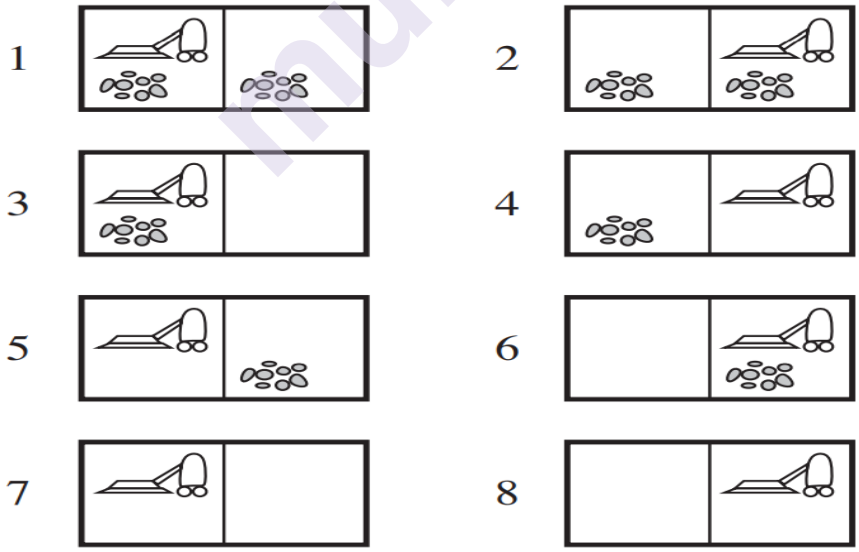


Fig. 4.6 The eight possible states of the vacuum world; states 7 and 8 are goal states.

In the erratic vacuum world, the Suck action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.
- a contingency plan such as the following:

[Suck, if State = 5 then [Right, Suck] else [ ]]

If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms.

For example, if the initial state is 1, then the action sequence [Suck, Right, Suck] will reach a goal state, 8. Thus, solutions for nondeterministic problems can contain nested if–then–else statements;

4.4.2 AND- OR search trees:

A solution to a problem can be obtained by decomposing it into smaller sub-problems. Each of this sub-problem can then be solved to get its solution. These sub-solutions can then be recombined to get a solution as a whole. OR graph finds a single path.

AND arc may point to any number of successor nodes, all of which must be solved in order for an arc to point a solution. This is actually called as an AND-OR graph or AND/OR tree.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Fig. 4.7 An algorithm for searching AND–OR graphs generated by nondeterministic environments.

Figure 4.8 gives a recursive, depth-first algorithm for AND–OR graph search. One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems (e.g., if an action sometimes has no effect or if an unintended effect can be corrected). If the current state is identical to a state on the path from the root, then it returns with failure. This doesn't mean that there is no solution from the current state; it simply means that if there is a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new

incarnation can be discarded. With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state.

Example.

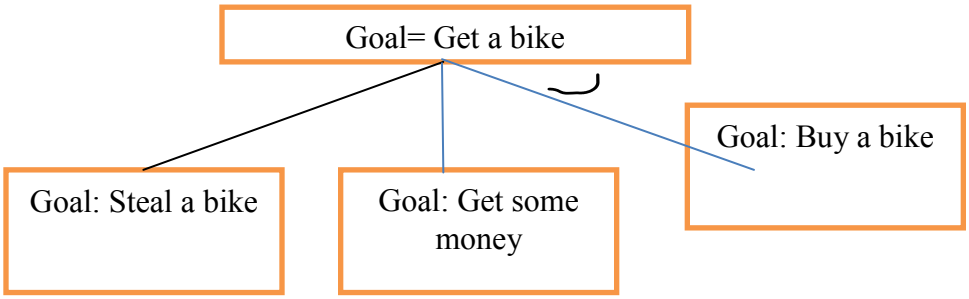


Fig. 4.8 AND/OR graph example

4.5 SEARCHING WITH PARTIAL OBSERVATIONS

When an environment is partially observable, an agent can be in one of several possible states. An action leads to one of several possible outcomes.

To solve these problems, an agent maintains a belief state that represent the agent's current belief about the possible physical state it might be in, given the sequence of actions and percepts up to that point.

Agents percepts cannot pin down the exact state the agent is in

Let Agents have **Belief** states

- Search for a sequence of belief states that leads to a goal
- Search for a plan that leads to a goal

Sensor-less vacuum world

Assume belief states are the same but no location or dust sensors

Initial state = {1, 2, 3, 4, 5, 6, 7, 8}

Action: Right

Result = {2, 4, 6, 8}

Right, Suck

Result = {4, 8}

Right, Suck, Left, Suck

Result = {7} guaranteed!

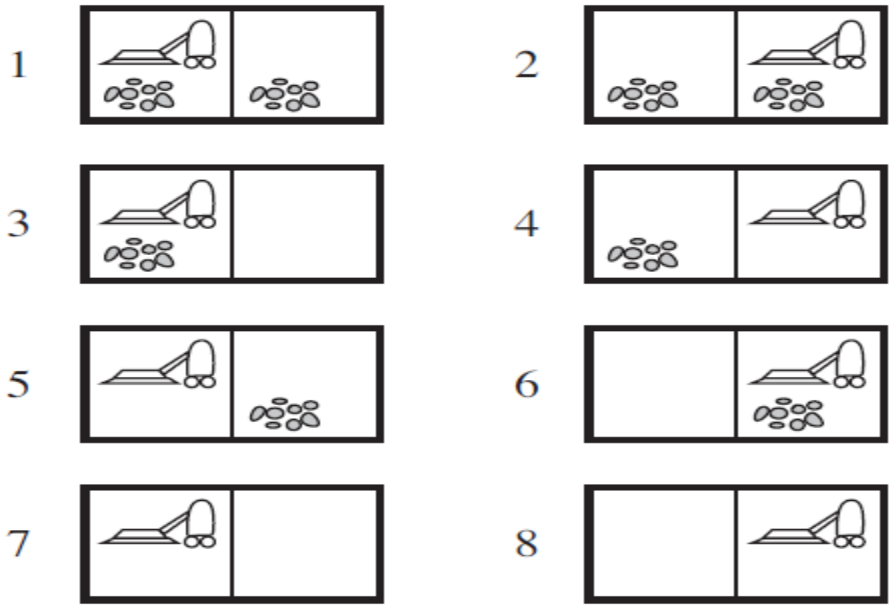


Fig. 4.2.9 Sensor less vacuum world

4.5.1 Searching with no observation:

It is instructive to see how the belief-state search problem is constructed. Suppose the underlying physical problem P is defined by ACTIONSP, RESULTP, GOAL-TESTP, and STEP-COSTP.

Then we can define the corresponding sensor less problem as follows:

- **Belief states:** The entire belief-state space contains every possible set of physical states. If P has N states, then the sensor less problem has up to 2N states, although many may be unreachable from the initial state.
- **Initial state:** Typically, the set of all states in P, although in some cases the agent will have more knowledge than this.
- **Actions:** This is slightly tricky. Suppose the agent is in belief state b = {s1, s2}, but ACTIONS P(s1) ≠ ACTIONS (s2); then the agent is unsure of which actions are legal.
- **Transition model:** The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state.
- **Goal test:** The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if all the physical states in it satisfy GOAL-TEST P. The agent may accidentally achieve the goal earlier, but it won't know that it has done so.
- **Path cost:** This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values.

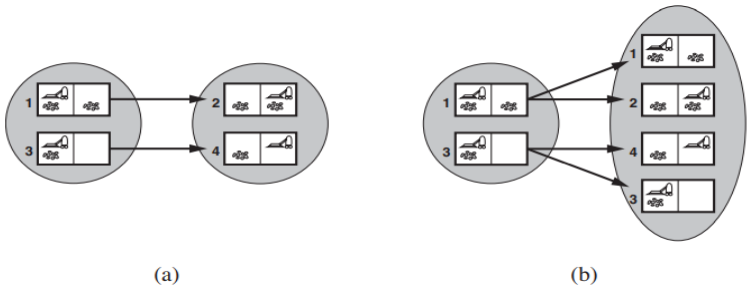


Fig. 4.10 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, right. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

4.5.2 Searching with observations:

For a general partially observable problem, we have to specify how the environment generates percepts for the agent. For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor but has no sensor capable of detecting dirt in other squares. The formal problem specification includes a PERCEPT(s) function that returns the percept received in a given state. (If sensing is nondeterministic, then we use a PERCEPTS function that returns a set of possible percepts.) For example, in the local-sensing vacuum world, the PERCEPT in state 1 is [A, Dirty]. Fully observable problems are a special case in which PERCEPT(s) = s for every state s, while sensorless problems are a special case in which PERCEPT(s) = null.

- The ACTIONS, STEP-COST, and GOAL-TEST are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated.
- The prediction stage is the same as for sensorless problems: given the action a in belief state b, the predicted belief state is $\hat{b}$ = PREDICT (b, a).
- The observation prediction stage determines the set of percepts o that could be observed in the predicted belief state: POSSIBLE-PERCEPTS ($\hat{b}$) = {o : o = PERCEPT(s) and s ∈ $\hat{b}$} .
- The update stage determines, for each possible percept, the belief state that would result from the percept. The new belief state bo is just the set of states in $\hat{b}$ that could have produced the percept: bo = UPDATE ($\hat{b}$, o) = {s : o = PERCEPT(s) and s ∈ $\hat{b}$}.
- Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts: RESULTS (b, a) = {bo : bo = UPDATE(PREDICT(b, a), o) and o ∈ POSSIBLE-PERCEPTS(PREDICT(b, a))} .

Fig.4.10 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, Right is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are [B, Dirty] and [B, Clean], leading to two belief states, each of which is a singleton. (b) In the slippery world, Right is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are [A, Dirty], [B, Dirty], and [B, Clean], leading to three belief states as shown.

4.5.3 Solving partially observable problems:

The preceding section showed how to derive the RESULTS function for a nondeterministic belief-state problem from an underlying physical problem and the PERCEPT function.

To solve these problems, an agent maintains a belief state that represent the agent's current belief about the possible physical state it might be in, given the sequence of actions and percepts up to that point.

Agent's percepts cannot pin down the exact state the agent is in

Let Agents have **Belief** states

- Search for a sequence of belief states that leads to a goal
- Search for a plan that leads to a goal

For such formulation, the AND-OR search algorithm can be applied directly to derive a solution. Figure 4.2.11 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept [A, Dirty]. The solution is the conditional plan [Suck, Right, if B state = {6} then Suck else [ ]] .

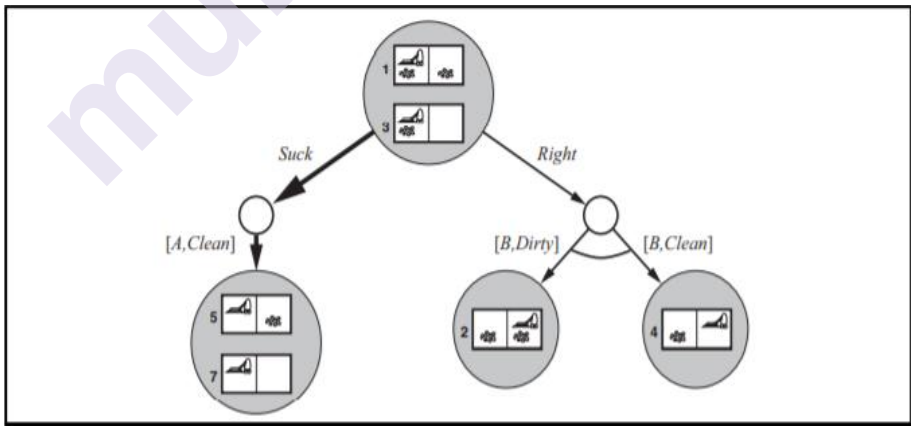


Fig. 4.11 AND -OR search tree

4.5.4 An agent for partially observable environments:

The design of a problem-solving agent for partially observable environments is quite similar to the simple problem-solving agent. the agent formulates a problem, calls a search algorithm (such as AND-OR-GRAPH-SEARCH) to solve it, and executes the solution. There are two main differences. First, the solution to a problem will be a conditional plan

rather than a sequence; if the first step is an if–then–else expression, the agent will need to test the condition in the if-part and execute the then-part or the else-part accordingly. Second, the agent will need to maintain its belief state as it performs actions and receives percepts.

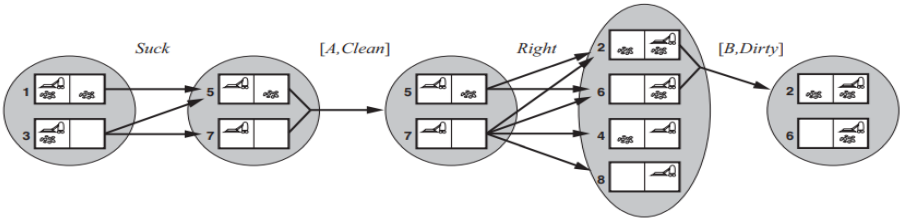


Fig. 4.12 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

4.6 ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

So far, we have concentrated on agents that use offline search algorithms. They compute a complete solution before setting foot in the real world and then execute the solution. In contrast, an online search agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi dynamic domains—domains where there is a penalty for sitting around and computing too long.

Online search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that might happen but probably won't. Of course, there is a trade-off: the more an agent plans ahead, the less often it will find itself up the creek without a paddle.

Online search is a necessary idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an exploration problem and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B.

4.6.1 Online search problem:

An online search problem must be solved by an agent executing actions, rather than by pure computation. online search agents can build a map and find a goal if one exists.

We assume a deterministic and fully observable environment but we stipulate that the agent knows only the following:

1. ACTIONS(s)
2. The step-cost function
3. GOAL-TEST(s).

For example, in the maze problem shown in Figure, the agent does not know that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,1).

- Necessary in unknown environments
- Robot localization in an unknown environment (no map)
- Does not know about obstacles, where the goal is, that UP from (1,1) goes to (1, 2)
- Once in (1, 2) does not know that down will go to (1, 1)
- Some knowledge might be available
- If location of goal is known, might use Manhattan distance heuristic
- Competitive Ratio = Cost of shortest path without exploration/Cost of actual agent path
- Irreversible actions can lead to dead ends and CR can become infinite.
- Hill- Climbing is already an online search algorithm but stops at local optimal.

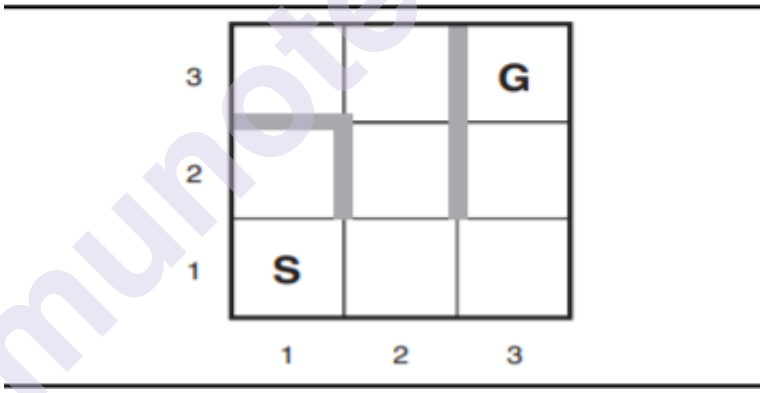


Fig. 4.13 A simple maze problem

4.6.2 Online search agents:

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. An online algorithm, on the other hand, can discover successors only for a node that it physically occupies.

To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a local order. Depth-first search has exactly this property because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure 4.2.14

```
function ONLINE-DFS-AGENT(s′) returns an action
  inputs: s′, a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s′) then return stop
  if s′ is a new state (not in untried) then untried[s′] ← ACTIONS(s′)
  if s is not null then
      result[s, a] ← s′
      add s to the front of unbacktracked[s′]
  if untried[s′] is empty then
      if unbacktracked[s′] is empty then return stop
      else a ← an action b such that result[s′, b] = POP(unbacktracked[s′])
  else a ← POP(untried[s′])
  s ← s′
  return a
```

Fig. 4.14 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action

It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

4.6.3 Online local search:

Like depth-first search, hill-climbing search has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

Instead of random restarts, one might consider using a random walk to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will eventually find a goal or complete its exploration, provided that the space is finite. On the other hand, the process can be very slow.

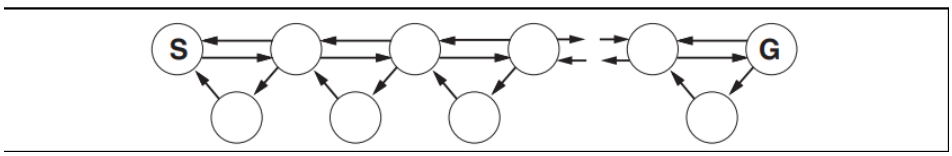


Fig. 4.15 An environment in which a random walk will take exponentially many steps to find the goal

4.7 SUMMARY

This chapter has examined search algorithms for problems beyond the "classical" case of finding the shortest path to a goal in an observable, deterministic, discrete environment. Local search methods such as hill climbing operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including simulated annealing, which returns optimal solutions when given an appropriate cooling schedule.

A genetic algorithm is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by mutation and by crossover, which combines pairs of states from the population.

In nondeterministic environments, agents can apply AND–OR search to generate contingent plans that reach the goal regardless of which outcomes occur during execution.

When the environment is partially observable, the belief state represents the set of possible states that the agent might be in. Standard search algorithms can be applied directly to belief-state space to solve sensor less problems, and belief-state AND–OR search can solve general partially observable problems.

4.8 UNIT END QUESTIONS

1 What do you mean by local maxima with respect to search technique?
2 Explain Hill Climbing Algorithm.
3 Explain AO* Algorithm.
4 Explain a searching with nondeterministic action.
5 Explain searching with partial observations.
6 Write a note on simulated annealing.
7 Explain a local search algorithm.
8 Write a note on online search agents.
9 Write a note on unknown environments.
10 Consider the sensorless version of the erratic vacuum world. Draw the belief-state space reachable from the initial belief state {1, 2, 3, 4, 5, 6, 7, 8}, and explain why the problem is unsolvable.

4.9 BIBLIOGRAPHY

- Deva Rahul, Artificial Intelligence a Rational Approach, Shroff, 2014.
- Khemani Deepak, A First course in Artificial Intelligence, McGraw Hill, 2013.
- Chopra Rajiv, Artificial Intelligence a Practical Approach, S. Chand, 2012.
- Russell Stuart, and Peter Norvig Artificial Intelligence a Modern Approach, Pearson, 2010.
- Rich Elaine, et al. Artificial intelligence, Tata McGraw Hill, 2009.

*****

UNIT III

5

ADVERSARIAL SEARCH

Unit Structure

5.0 OBJECTIVE

The objective of this chapter is to study the different strategies used by players to compete each other and try to win the game.

5.1 INTRODUCTION

In this chapter we will see Adversarial search method in which we examine the situation where agent compete with one another and try to defeat one another in order to win the game. Adversarial search is a game playing technique in which two or more players with conflicting goals are trying to explore the same search space for the solution.

5.2 GAMES

In single agent environment the solution is expressed in the form of sequence of actions. But there might be a situation occur in a game playing where more than one agent searching for a solution in the same search space. The environment with more than one agent is called as multi-agent environment, in which each agent has to consider the action of other agent and how they affect its own welfare. Such conflicting goal give rise to the adversarial search. So, searches in which two or more players with conflicting goals are trying to explore the same search space for the solution are known as Games.

Types of Games in AI:

Perfect Information Game: In this game agent act sequentially and observe the state of the world before acting. Each agent acts to maximize its own utility. The agent has all the information about the game and they can monitor each other movements as well. E.g. Chess, Checkers etc.

Imperfect information Game: In this type of game agent don't have all information about the game and not aware of what's going on. For e.g. tic-toc-toe, blind bridge, battleship etc.

Deterministic game- Deterministic games are follow a strict pattern and set of rules for the game. Change in state is fully determined by player move. For e.g. chess, Go, Checkers, tic-tac-toe etc.

Non-Deterministic games: non-deterministic games have various unpredictable events and it involves a factor of chance or luck. So, change in state is partially determineds by chance. In this type of game each action response is not fixed. Such type of game is also called as stochastic games. E.g. Poker, Monopoly, Backgammon etc.

Game Problem formulation:

A game can be formally defined s a kind of search problem with various elements as follows

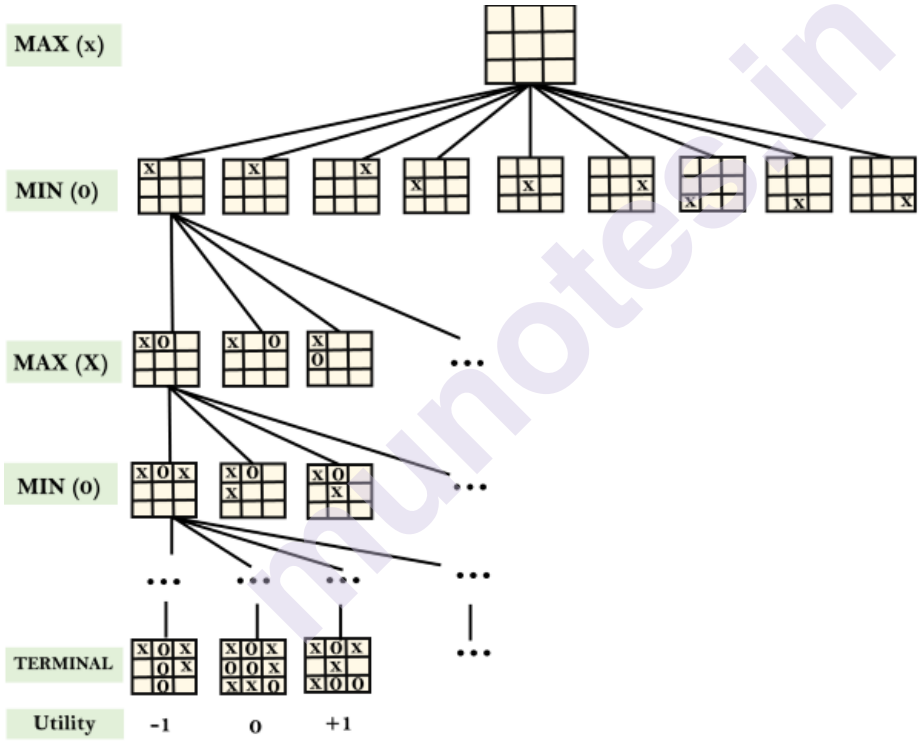
- **S_0**- initial state, which specifies how the game is set up at the start
- **Player(s)**-defines which player has the move in a state
- **Actions(s)**-returns the set of legal moves in a state.
- **Result (s, a)**-The transition model defines the result of a move.
- **Terminal -Test(s)**- A terminal test is true when the game is over and false otherwise. Terminal state is sate where game has ended.
- **Utility (s, p)**- Utility function defines the final numeric value for a game that ends in terminal state s for player p. Let's consider chess in which outcome is a win, lose or draw, with values +1, 0 or ½.

In **zero-sum game** the total payoff to all players is the same for every instance of the game. Chess is a zero some game because every game has payoff of either 0+1, 1+0 or ½ + ½.

Game Tree:

A game tree is defined by the initial state, ACTION and RESULT functions. A game tree is a tree where nodes indicate game states and edges indicate moves.

We consider tic-tac-toe game with two player, one player call MAX and another player call MIN. the following figure shows the part of game tree for tic-tac-toe. Max has nine possible moves from initial state. The player alternates between placing an X on MAX's turn and O on MIN's turn until reach up to leaf node. Leaf nodes corresponding to terminal state such that one plyer has three in a row or all the squares are filled. Both players will continue each node, and trying to keep each other from winning. In game tree there a layer for MIN and MAX called as ply. The numbers on leaf node indicate the utility value of the terminal state. By considering MAX point of view, high value is good for MAX and bad for MIN. In this either MAX win or MIN win or it's draw.



5.3 OPTIMAL DECISION IN GAMES

The optimal solution in a normal search problem would be a sequence of actions that leads to a goal state or a terminal state. In adversarial search, MAX must find a contingent strategy, which specifies MAX's move in the initial state, then as a result from every possible response by MIN, Max makes moves in the state, then MIN's moves in the state resulting from every possible response by MAX to those moves and so on.

Given a game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAXX(n). if both players play optimally from the start of the game until end of the

game, then the minimax value of a node is the utility of being in the corresponding state. The minimax value of a terminal state is just its utility. MAX prefers to move to a state of maximum value and MIN prefers a state of minimum value then

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN}. \end{cases}$$

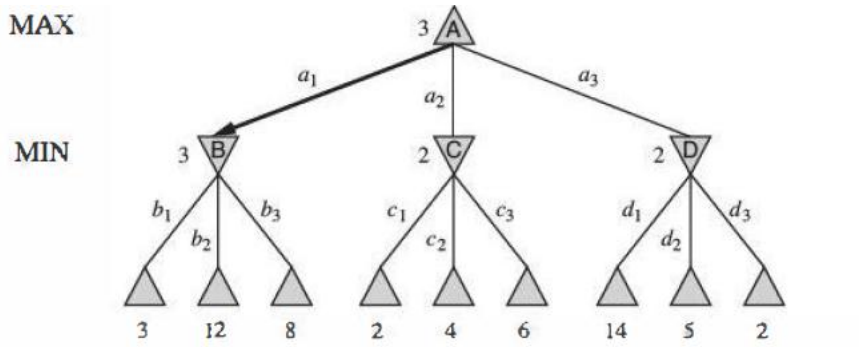
5.3.1 Minimax Algorithm:

Minimax is a barracking algorithm used in decision making and game theory. It is used to find the optimal move for a player by assuming the opponent is also playing optimally. It uses recursion to search through the game tree. A Minimax algorithm is often used to play games in AI such as Chess, tic-tac-toe, Go etc. Algorithm computes the minimax decision for the current state.

In this algorithm there are two players, MAX and MIN. Both players play the game as one tries to get the highest score while the opponent tries to get the lowest score. In the MINIMAX algorithm, the complete game tree is explored based on depth first search algorithm. A MINIMAX algorithm proceeds down the tree until the terminal node, then backtracks the tree in a recursive manner.

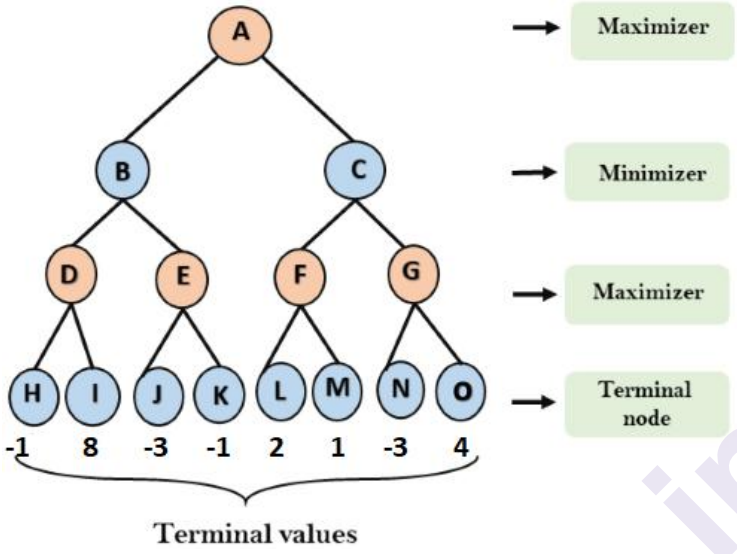
In the following diagram, the algorithm first recurses down to the three lowest left nodes and uses UTILITY function on them to find their values are 3,12 and 8 respectively. Then it takes the minimum vale 3 and return that value as the backup for node B. A same process gives backup value of 2 for C and 2 for D. Finally, to get the backed-up value of 3 for the root node, we take the maximum of 3,2, and 2.

If maximum depth of the tree is m and at each point legal moves are b then the time complexity of the minimax algorithm is $O(b^m)$. The algorithm that generates all action at once then, the space complexity is O(bm). The algorithm that generates actions one at a time then space complexity is O(m).



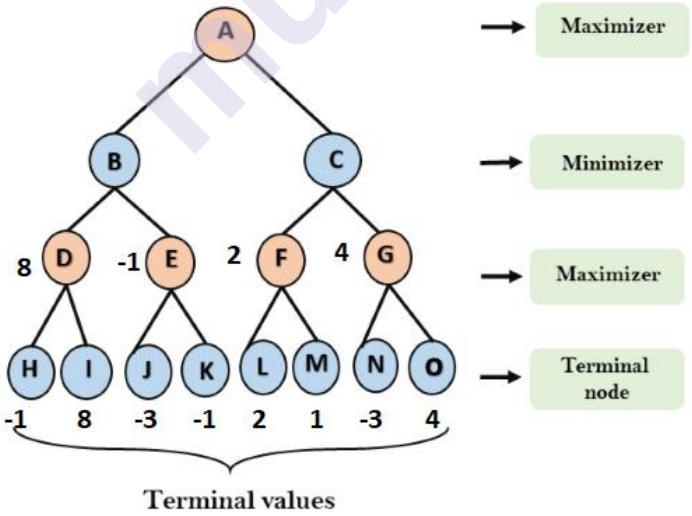
Example:

1 The algorithm generates the entire game tree and apply the utility function to get the utility values for the terminal state. Now consider in the following diagram A is the initial state.



1) Now, first find the utilities value for MAX, so we will compare each value in terminal state with initial value for MAX and determine the higher node values.

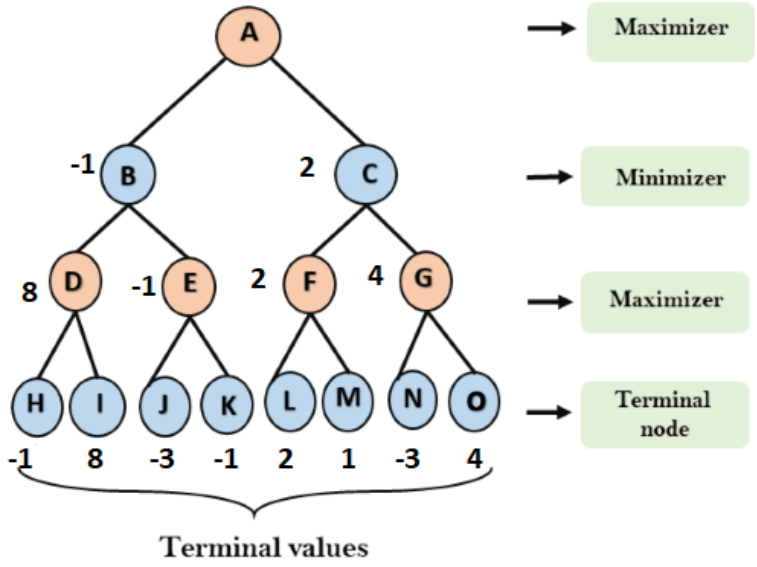
So, for node D max (-1, 8) => 8
for node E max (-3, -1) => -1
for node F max (2, 1) => 2
for node G max (-3, 4) => 4



2) Now, it's a turn for minimizer to minimize the MAX utility, so it will compare all node values and finds the third layer node values.

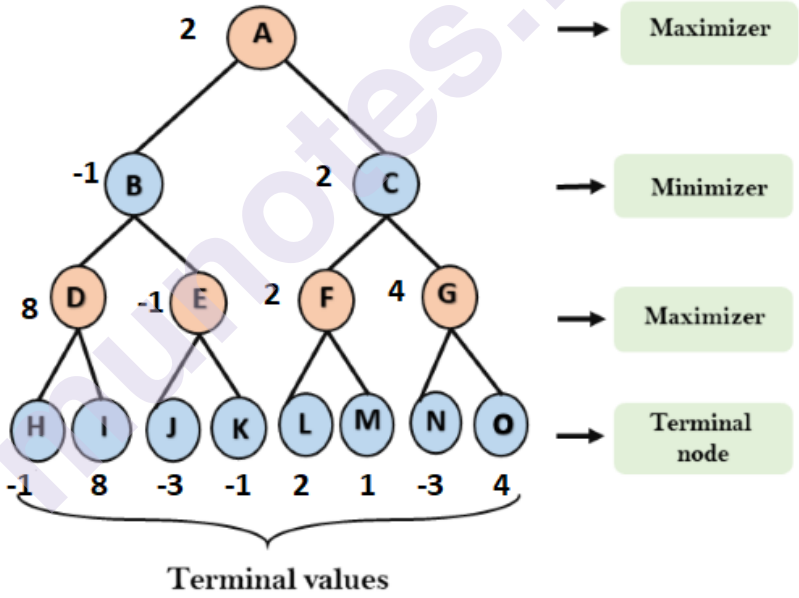
for node B min (8, -1) =>-1

for node C min (2, 4) => 2



3) Now its turn for MAX to choose the maximum of all node values and find the maximum value for the root node

for node A max (-1, 2) => 2



5.3.2 Optimal Decision in Multiplayer Game:

There are many popular games that allow more than two players. Let us examine how we can use the minimax concept for multiplayer games. From a technical perspective, this seems straightforward but it raises some interesting new conceptual issues. First, its necessary to replace single value of each node with a vector of value. Consider a three player game with players A, B and C having a vector (V_A,V_B,V_C) associated with each node. This vector gives the utility of the state from each player's point of view for terminal states. This can be implemented using utility functions that return a vector of utilities.

Now, for nonterminal states, consider the node X in the game tree as shown in below figure. In that state, player C decides what to do. The two choices lead to terminal states with utility vectors (V_A =1, V_B =2, V_C =6) and (V_A =4, V_B =2, V_C =3). C choose the first move because 6 is bigger than 3, means if state X is reached, subsequently play will lead to a terminal state with utilities (V_A =1, V_B =2, V_C =6). So, in this vector it backed-up the values of X.

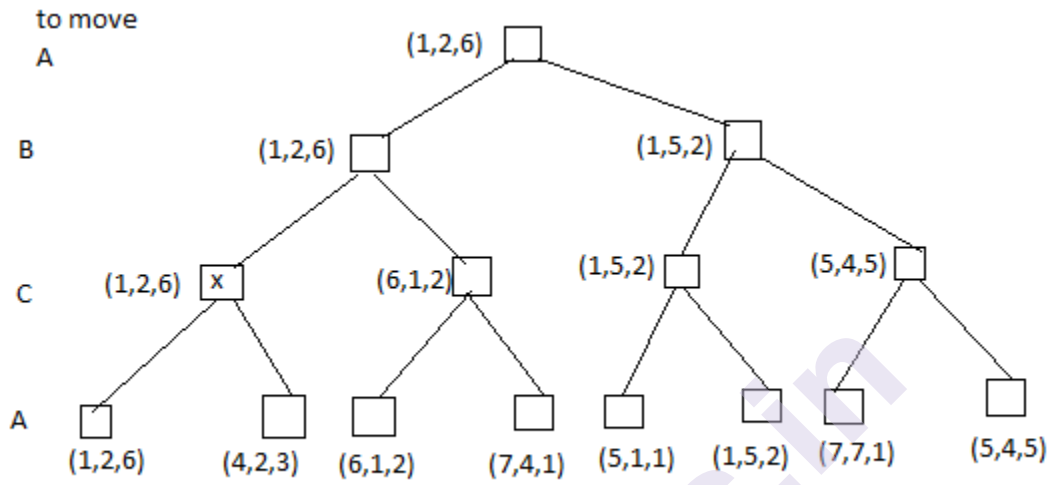


Fig. The first three piles of game tree with three players (A, B, C).

Algorithm for calculating minimax decision

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

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

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
```

5.4 ALPHA-BETA PRUNING

It is the modified version of minimax algorithm. In minimax search, the number of game states it has to examine is exponential in the depth of the tree. The exponent can't be eliminated but we can cut it to half. Therefore, there is a way to compute the correct minimax decision without checking every node in the game tree called as pruning. It involves two threshold

parameters such as Alpha and Beta for future expansion, so known as alpha-beta pruning or alpha-beta algorithm. This can be applied at any depth of a tree. The alpha-beta pruning to a standard minimax algorithm returns the same moves but it removes all the nodes which are not affecting the final decision. It makes the algorithm fast by pruning these nodes.

α=the best or the highest value choice found so far at any point along the path of maximizer. The initial value of beta is -∞.

β= the best or lowest value choice found so far at any point along the path of minimizer. The initial value of beta is +∞.

During alpha-beta search, the value of alpha and beta are updated as they go along, and prunes the remaining branches at a node as soon as the value of current node is known to be worse than the current alpha and beta values for MAX and MIN respectively. The main condition required for alpha-beta pruning is α >= β.

Alpha-Beta Search Algorithm:

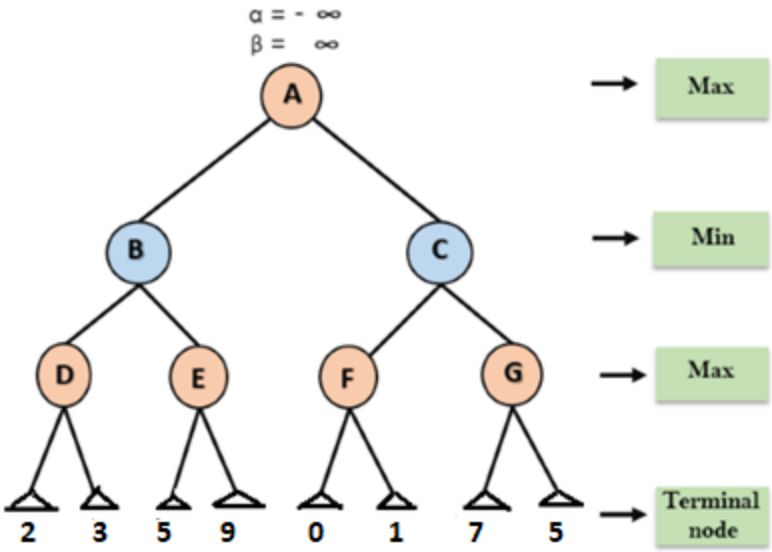
```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

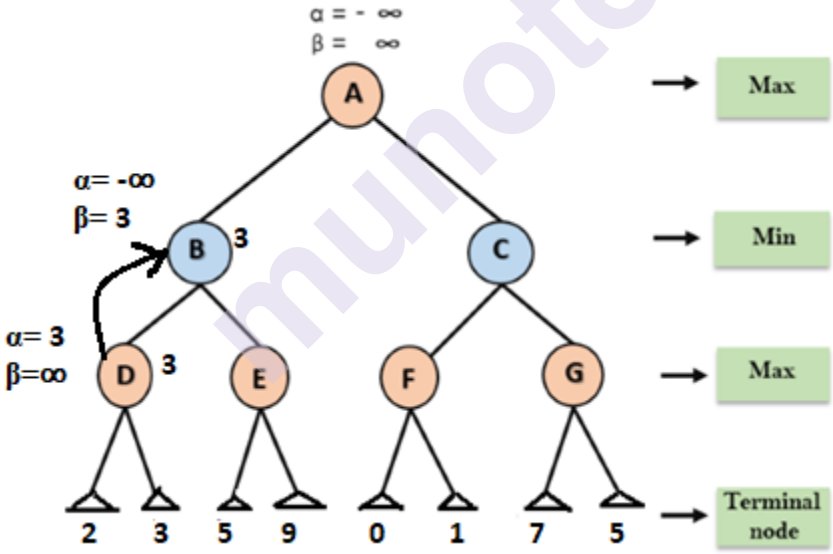
Example:

1) In the first step MAX player start with first move from node A where α= -∞ and β= +∞. Now these values of α and β are passed to node B and then node D.

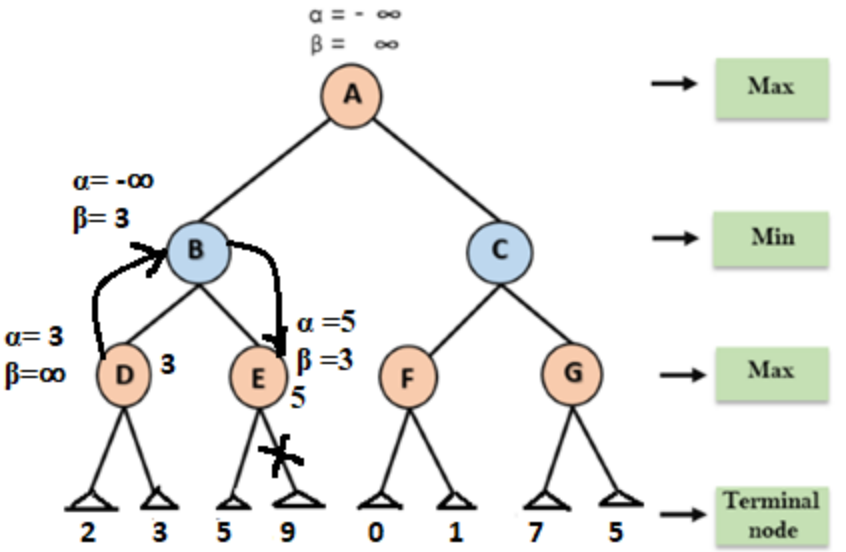


2) At node D the MAX player will play and calculate the α value. The value of α compare with 2 and 3. So max (2, 3) =3. So, 3 will be the value of α at node D.

3) Now backtracking is performed at node B. The value of β will change as MIN player will play. Now β= +∞, will compare with the available subsequent nodes value. So, min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

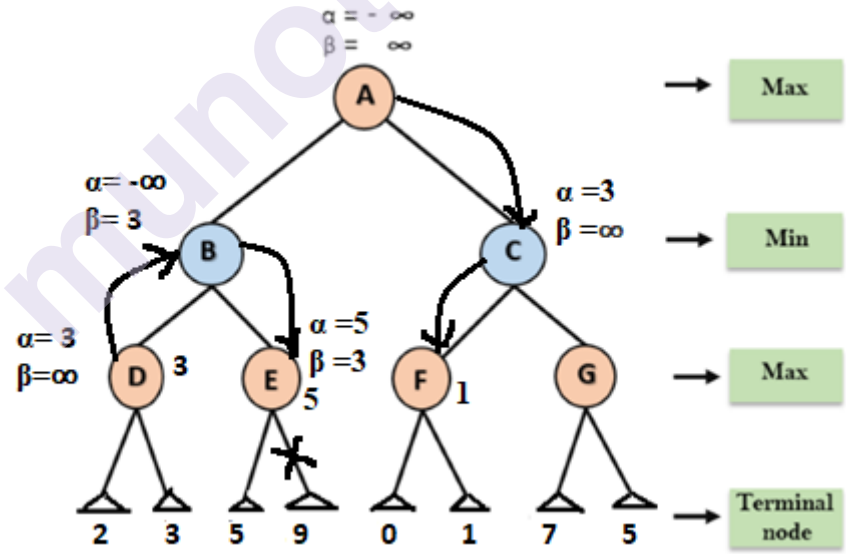


4) Algorithm traverse the next successor of node B which is nothing but node E. the value of α =-∞ and β =3. Now, MAX will play at node E and value of α will change. The current value of α will be compared with 5. So max (-∞, 5) =5. At node E, α =5 and β =3. But α>= β so the right successor of E will be pruned an algorithm stop traversing it. Now value of E node is 5.

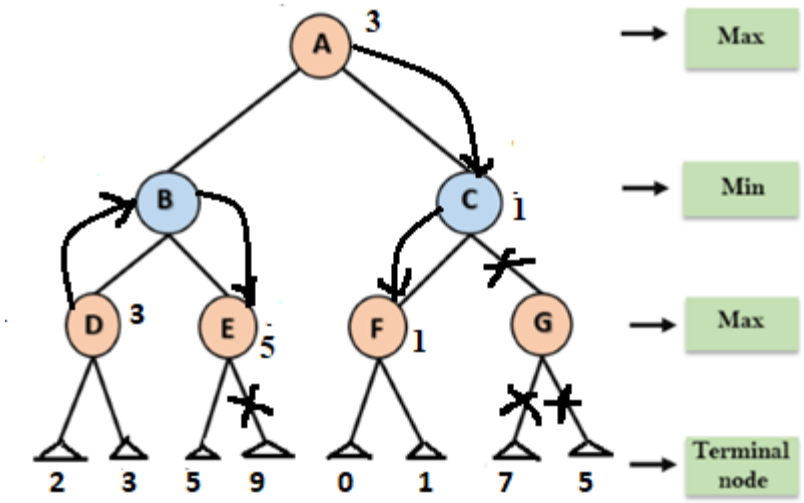


5) Algorithm again backtrack the tree from node B to A. The value of alpha will be changes at node A. max (- ∞,3) =3 and β=+∞. Both the values are passed to right successor of A which is C. Now, at node C, α=3 and β= +∞, and these same values will be passed on to node F

6) At node F value of alpha is compared with 0 so max (3,0)= 3 and compared with right child 1, so max (3,1) =3 but still α remains 3, but the node value of F will become 1.



7) Node F return the node value 1 to node C. At node C the value of beta is changed so min (∞, 1) =1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β. Now G node will be pruned and algorithm not traverse the entire sub tree of G. C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. And following is the final game tree.



5.4.1 Move Ordering:

The effectiveness of alpha-beta pruning is highly dependent on the order in which states are examined. It can be of two types

1) Worst ordering- Alpha beta pruning algorithm exactly work like minimax algorithm and it does not prune any leaves of the tree in some cases. So, in that case consumes more time because of alpha beta factors. Such a move pruning is known as worst ordering. the time complexity in for such case is $O(b^m)$.

2) Ideal ordering- it occurs when lots of pruning happens in the tree. Best move is occurred at the left side of the tree. The time complexity in ideal ordering is $O(b^{m/2})$.

5.5 IMPERFECT REAL-TIME DECISION

The minimax algorithm generates entire game search space while the alpha beta pruning algorithm allows to prune large path. The alpha beta has to search all the way to terminal state for at least a portion of search space. Due to the fact that moves have to be made within a reasonable amount of time, this depth is not usually practical. In order to make a move in reasonable amount of time, program should cut off the search earlier and heuristic evaluation function is applied. It effectively turns nonterminal nodes into terminal leaves.

Alter minimax or alpha beta in two ways, first is replace the utility function by heuristic evaluation function EVAL. This estimates the position's utility. And second, replace terminal test by a cutoff test that decides when to apply EVAL function. The following is the heuristic minimax for state s and maximum depth d

H-MINIMAX (s, d)=

EVAL(s) if CUTOFF-TEST(s,d)

$\max_{a \in Action(s)}$H-MINIMAX(RESULT(s,a),d+1 if Player(s)=MAX
H-MINIMAX (RESULT(s,a),d+1 if Player(s)=MIN $\min_{a \in Action(s)}$

5.5.1 Evaluation Function:

This function returns an estimate of the expected utility of the game from a given position. the performance of a game playing program strongly depends on the quality of its evaluation function. It is possible that the wrong evaluation function will be led an agent to position that turns out to be lost.

Design good evaluation function:

1) The evaluation function should order the terminal states same as true utility function: win states must better than draws, which in turn draw states must be better than loss states.

2) The computation must not take too long.

3) The evaluation function should be strongly correlated with the actual chances of winning for non-terminal states.

The majority of evaluation functions work by calculating various features of the state. For example, there are features for the number of white pawns, black pawn and so on in chess. This all features define various categories of state. The state in each category has same value for all the features. Consider one category includes two pawns vs. one pawn. Any given category contains some states that led to wins, some draw and some led to losses. The evaluation function cannot determine which states are which. It can return single value that reflect the proportion of states with each outcome.

Suppose in two-pawns vs. one-pawn 72% of the states led to win, 20% loss and 8% draw. Then expected value=(0.72* 1)+(0.20*0)+(0.08*0.5)=0.76. the expected value can be found for each category resulting in an evaluation function for each state. The evaluation function for terminal not return actual expected values as long as the ordering of the state is the same. Many evolutions function find separate numerical contributions from each feature and combine them to find total value. Weighted linear function is a kind of evaluation function that can be expressed as the following

$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$

Where w_i is a weight, fi is a feature of the position, the fi is the number of each kind of piece and the w_i is the value of the pieces in a chess.

5.5.2 Cutting off Search:

Alpha-beta search is modified so that it will call the heuristic EVAL function when it is appropriate to cut off the search. So, we can modify

alpha beta search algorithm and replace terminal state with cutoff-test and utility is replace by EVAL.

if Cutoff-Test(state, depth) then return Eval(state)

by setting a fixed depth limit is the easiest way to control the amount of searching so that cutoff return true for all depth greater than fixed depth d. In order to select a move within the allocated time, the depth d is chosen for it. Iterative deepening search can be applied and also helps with move ordering. The program returns the move selected by the deepest completed search as time runs out.

5.5.3 Forward Pruning:

Forward pruning is a technique in which some moves at a given node are pruned immediately without further consideration. It reduces number of nodes t be examined at each level in a search process. Beam search is a one approach for forward pruning. Consider a beam of the n best moves for each ply rather than considering all possible moves. But in this approach, there is no guarantee that the best move will not be pruned away.

The PROBCUT or probabilistic cut algorithm is a forward pruning version of alpha-beat search. Same as alpha-beta search PROBCUT also prunes node that are probably out side the window. This probability is computed by doing a shallow search and find the backed-up value v of a node. Based on past experience, estimate the probability that a score of v at depth d in the tree would be outside (α, β).

5.5.4 Search Versus Lookup:

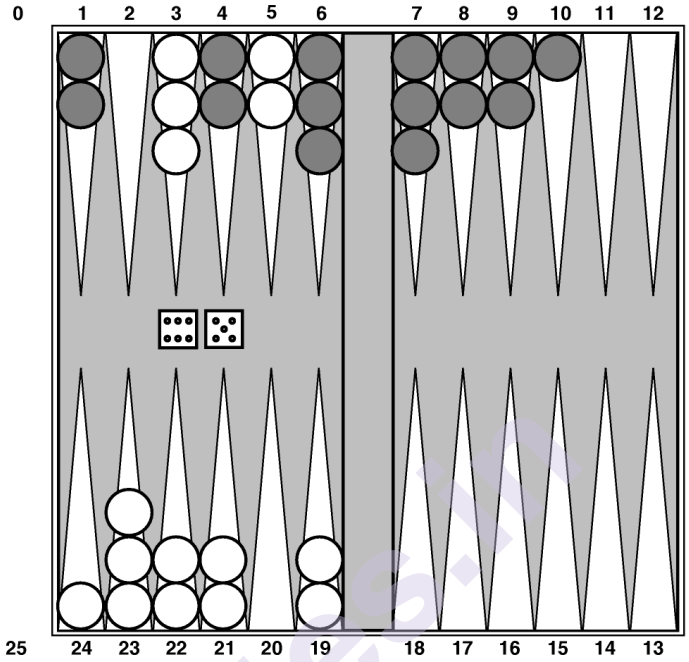
Many games plying programs us table lookup for the opening and ending of game rather than search. In order to play each opening, the best advice of human expert is copied from books and input it in a table for the computer's use. Also, computer can collect statistics from a database of previously played games to see which opening sequences are led to win. After starting the game, in the early moves there are few choices and thus usually after 10 moves the program must switch form table lookup to search. Near the end of the game there are few possible positions and thus more chance to do lookup.

A human can tell the general strategy for playing a game while computer on the other hand can completely solve the endgame by producing a policy, which is mapping from every possible state to the best move in that state.

5.6 STOCHASTIC GAMES

Many games are unpredictable in nature, such as dice throw., such type of games is called as stochastic games. The outcome of such games depends on skills as well as luck. For e.g., Gambling game Golf ball game, Backgammon game etc.

Backgammon game combine luck and skill of players. Dice are rolled at the starting of player's turn to find the legal moves. Each player moves the pieces according to the dice are thrown. For example, if a player who plays with black piece roll the dice and dice shown 6-5 and has four possible moves.



The goal of the game is to move all one's pieces off the board. White moves clockwise and black move anticlockwise. A piece can move to any position until multiple opponents are pieces are there. White knows his or her own legal moves but don't know about the black legal moves. So white cannot build a standard game tree. A game tee in a backgammon must include chance node in addition with MIN and MAX node. The possible dice node denoted by the branches leading from each chance nod. Brach is labeled with the roll and its probability. There are 36 ways to thrown the dice, but because a 6-5 is the same as 5-6, there are only 21 distinct rolls.

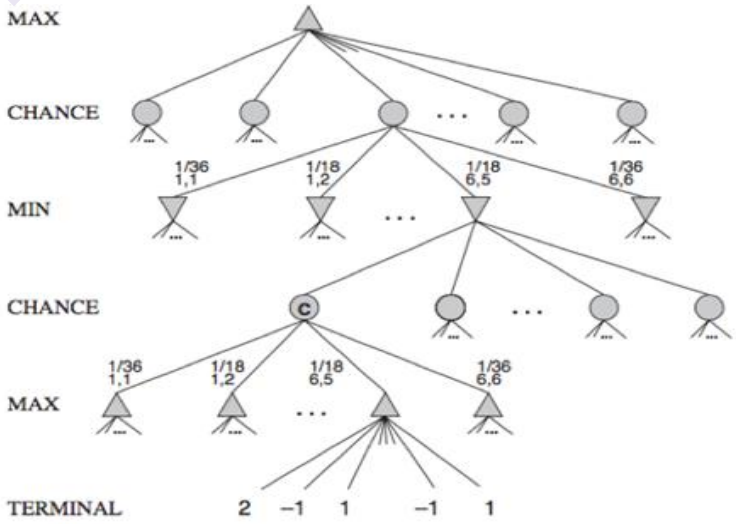


Fig. Game tree for Backgammon position

The next step is to understand how to make correct decision and pick up the move that leads to the best position. The position does not define minimax value, instead expected value of position has been calculated. For chance node expected value is calculated by using the sum of the value over all outcomes, weighted by the probability of each chance action

EXPECTIMINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX (RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX (RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX (RESULT}(s, r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

Where r is possible dice roll, (RESULT(s, r) same state as s

5.6.1 Evaluation functions for games of chance

In the same way that heuristic function returns an estimate of the distance to the goal, an evaluation function returns an estimate of the expected utility of the game from a given position. The performance of the game playing program depends strongly on the quality of its evaluation function. An inaccurate evaluation function will guide an agent towards position that turns out to be lost. There are some ways to design good Evaluation function.

First, the evaluation function should order the terminal states in the same way as the true utility function: state that are wins must evaluate better than draws, which in turn must be better than losses. Using the evaluation function would otherwise cause an agent to make mistakes even if it can see all the way to the end of the game. Second, the computation must not take too long. Third, the evaluation function must be strongly correlated with the actual chances of winning for nonterminal states.

Most evaluation function works by calculating various features of the state. For example, in chess, we would have features for the number of white pawns, black pawns, white queen, black queen and so on. These combine feature defines various categories or equivalent classes of states. The state in each category has the same values for all the features. Any given category will contain some state that lead to wins, some that lead to draws and some that lead to losses. The evaluation function cannot know which state are which, but it can return a single value that reflects the proportion of states with each outcome. For example, suppose our experience suggests that 72% of the states encountered in the two-pawns vs. one-pawn category lead to a win 20% to a loss (0), and 8% to a draw (1/2). Then a reasonable evaluation for states in the category is the expected value: $(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$. An evaluation function can be derived for every state by determining the expected value for each category. As with terminal states, the evaluation

function need not return actual expected values as long as the ordering of the states is the same.

This kind of analysis required too many categories and hence too much experience to estimate all the probabilities of winning. Instead, most evaluation functions compute separate numerical contributions for each feature and then combine them to calculate the total value. For example, introductory chess books give an approximate material value for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9. Other features such as "good pawn structure" and "king safety" might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position.

A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory Mathematically, this kind of evaluation function is called a weighted linear function and denoted as

$$EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s),$$

where each w_i is a weight and each f_i is a feature of the position.

5.7 PARTIALLY OBSERVABLE GAMES

Partially observable games are qualitatively different from other games. In these games various tricks are used to confused the opponent include the use of scouts and spies together the information and use of concealment and bluff too confuse the opponent etc. Kriegspiel is a partially observable chess.

5.7.1 Kriegspiel: Partially Observable Chess:

The uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent in deterministic partially observable games. For e.g., Battleships, Stratego etc. In partially observable chess the pieces can move but are completely invisible from the opponent. Means White can see only his pieces and black can see only his pieces. There is a referee who can see all the pieces of black and white and who can judge the game. It also periodically makes announcement that are heard by both players.

Any move that white chess piece would make if there were not any black piece is proposed to the referee. If move is not legal then referee announced illegal move. So, white piece may keep proposing until legal one is found and try to guess more about the location of black piece. If legal move is proposed then the referee announces the following "capture on square X", "check by D", Knight, Rank, File, Long Diagonal, Short Diagonal etc. If Black piece is checkmated the referee says so; otherwise, its Black's turn to play.

Kriegspiel state estimation directly map onto the partially observable, non-deterministic framework. For partially observable game the notion of strategy is altered. a move is making for every possible percept sequence that might receive. For Kriegspiel a winning strategy is one that, for each possible percept sequence leads to an actual checkmate foe every possible board state in the current belief state, regardless of how opponent moves.

5.7.2 Card Games:

Many examples of partial observability are provided by card game, where the missing information is generated randomly. In many games cards are distributed randomly at the beginning of the game, where each plyer not visible to opponent. For e.g., bridge, hearts etc.

It might seem that, these card games are the same as dice game. The cards are distributed randomly and find the moves available for each player. Consider all possible deals of invisible cards and solve each one as if it were a fully observable game. Then select the move that has best outcome. Suppose that each deal has probability P(s) then the moves are as follows

argmax $\sum_{sa}$ P(s) MINIMAX(RESULT(s, a))

In most of the card games the number of possible deals is larger. In bridge game, each player sees two hands out of the four so there are two unseen hands of 13 cards. Then the number of deals is 10,400,600. So instead of solving 10 million we resort to Monte Carlo approximation. In this, instead of taking all the deal, consider a random sample of N deals where probability of deals s appearing in sample is propositional to P(s). even for small N like 100 to 1000 the method gives good approximation.

5.8 STATE-OF-ART GAME PROGRAMMING

State of art game programs are blindly fast, highly optimized machine that incorporate the latest engineering. But this type of games is not much use for doing shopping or driving off-road

Chess:

it is well known for defeating world champion Garry Kasparov. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. The key to its success seems to have been its ability to generate singular extensions beyond the depth limit for sufficiently interesting lines of forcing/forced moves.

Checkers:

Jonathan Schaeffer and colleagues developed CHINOOK which runs on regular computers and it uses alpha beta search. It is used an endgame database defining perfect play for all positions.

Othello:

it is also called as Reversi. It is more popular as a computer game than a board game. It has a smaller search space than chess and 5 to 15 legal moves.

Backgammon:

The most of the work has gone into improving evaluation function. Gerry Tesauro combined reinforcement learning with neural network to develop accurate evaluator.

Go:

It is the most popular board game in Asia. It is 19 by 19 board game in which moves are allowed into every empty square. The branching factor starts at 361 which is too daunting for alpha-beta search method. The evaluation function for this game is difficult to write because control territory is often very unpredictable until the endgame.

Bridge:

It is multiplayer game in which cards are hidden from the other player. Players are paired into two teams for four player games.

5.9 SUMMARY

- A game has five components such as initial state, legal actions, result of action, terminal test and utility function.
- The minimax algorithm can select optimal moves by a depth-first enumeration of the game tree in two player zero sum game.
- The alpha-beta search algorithm achieves much greater efficiency by eliminating subtree that are irrelevant.
- Instead of considering whole game tree, cut the search off at some point and apply evaluation function to calculate the utility of state.
- Game of chance can be handled by an extension to minimax algorithm that calculate a chance node.
- Humans retain the edge in several games of imperfect information, such as poker, bridge, and Kriegspiel.

5.10 EXERCISE

1) Explain various strategies of game playing.
2) Explain the components to formally defined the game.
3) Explain minimax algorithm with example.
4) What is evaluation function?

5) Explain alpha-beta pruning

6) Explain alpha-beta cut-offs in game playing with example.

7) Describe and implement state description, move generators, terminal state, utility function and evaluation function for any of the following stochastic game: Bridge, Monopoly

8) Prove that alpha -beta pruning takes time $O(2^{m/2})$ with optimal move ordering, where m is maximum depth of the game tree.

5.11 BIBLIOGRAPHY

1) Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig

2) Artificial Intelligence: javapoint.com

*****

UNIT III

6

LOGICAL AGENT

Unit Structure

6.0 OBJECTIVE

In this chapter we design agent-based system that can represent a complex world. The new representation about the world can be derived by using process of inference and these new representations to deduce what to do. Also develop a logic as a general class of representation to support knowledge-based agent.

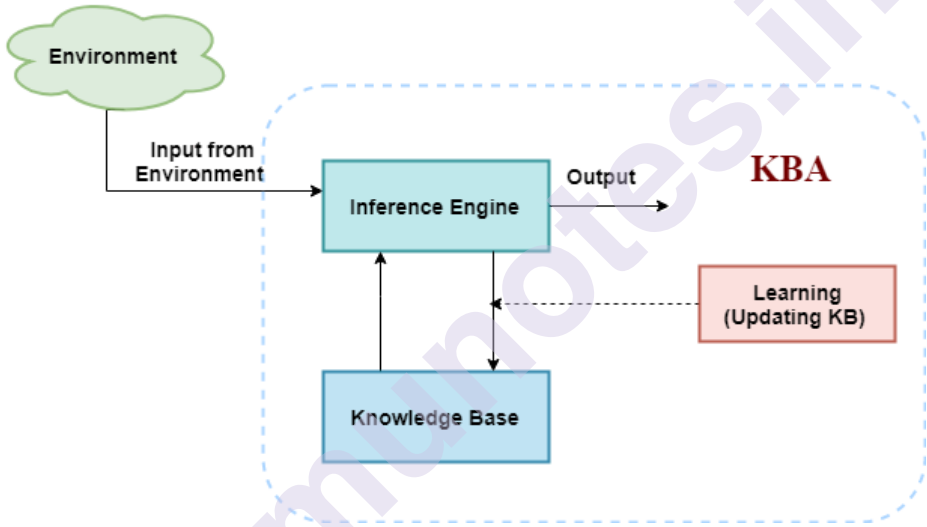
6.1 INTRODUCTION

Knowledge can be a particular path towards the solution. The knowledge must be represented in a particular format before using it. In 8 puzzle

transition model, the knowledge of what the actions do is hidden inside the domain specific code of the RESULT function. I observable environment as agent can represent what it knows about the current state is to list all possible concrete states. In this chapter logic is develop as a general class of representations to support knowledge-based agent. The knowledge-based agent can accept new tasks in the form of explicitly described gals. The agent can achieve ability quickly by learning new knowledge about the environment and also update the relevant knowledge so they can adapt to changes in the environment.

6.2 KNOWLEDGE-BASED AGENT

The main component of knowledge-based agent is its knowledge base. A knowledge base is a set of sentences and each sentence expressed in a language known as knowledge representation language.it is also known as KB. It represents some assertion about the world. Sometimes a sentence is dignified with axioms, taking the sentence as given without deriving it from another sentence.



The above diagram represents the architecture of knowledge based-agent. This agent takes the input by perceiving the environment. The knowledge base use TELL and ASK operation. TELL is add new sentence to the knowledge base perceive from the environment whereas, ASK is way to query what is known. The new sentence is derived from old is called as inference. Both ASK and TELL operation involve inference and it obey the requirements that when one ASKs a question of the knowledge base. The answer should follow from what has been told previously to knowledge base. The agent maintains a knowledge base which initially contain some background knowledge. The following figure shows a knowledge-based agent program.

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

MAKE-PERCEPT-SENTENCE generate a sentence asserting that the agent perceived the given percept at a given time. MAKE-ACTION-QUERY generate a sentence that asks what action should be taken at the current time. MAKE-ACTION-SENTENCE generate a sentence asserting that the selected action was executed.

The knowledge-based agent has three different levels:

1) **Knowledge level:** this is the first level of knowledge-based agent in which need to specify only what the agent knows and what its goals are, in order to fix its behavior. For example, an automated taxi might have to go from Location A to location B and one bridge is the only link between the two locations. An agent crosses the bridge because it knows that, that will achieve its goal.

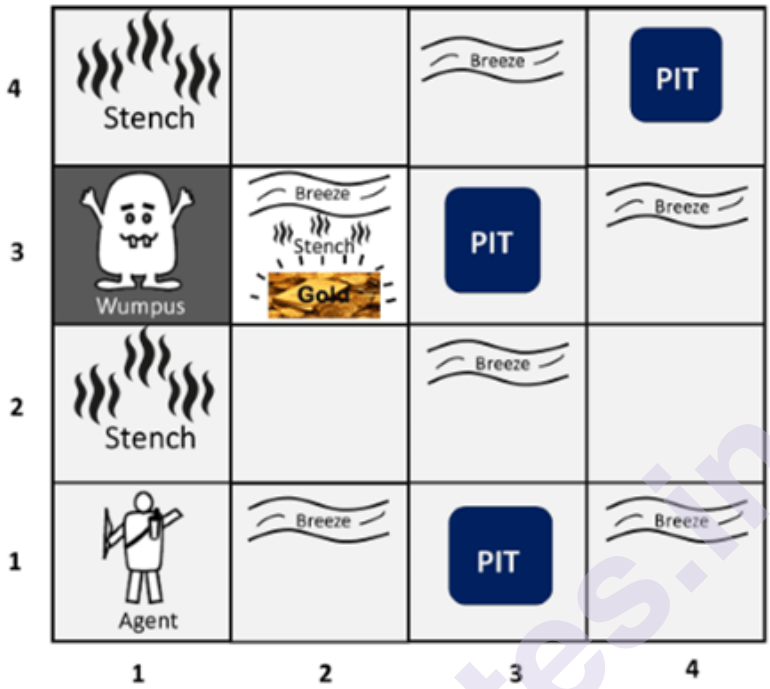
2) **Logical level:** At this level it is necessary to understand how knowledge representation of knowledge is stored. The sentences are encoded into different logics. For example, at logical level we expect that automatic taxi agent reach to the destination.

3) **Implementation level:** this level agent take action as per logic and knowledge level. So, it is representation of logic and knowledge. For example, at this level an automated taxi agent implements his knowledge and logic to reach to destination.

6.3 THE WUMPUS WORLD

The Wumpus world is an example of a world which describe an environment in which knowledge-based agent can show their worth. It is a cave consisting of 4X4 rooms connected by passageways. So, there are total 16 rooms connected with each other. The cave has a room contain beast and the beast can eat anyone who enter into the room. The Wumpus can be killed by the agent if he is facing it. There are some rooms contains pits. And if agent falls in pits, then he will be stuck there forever. In this cave, there is a possibility of finding a heap of gold. So, the goal of agent is to find the gold and climb out the cave without fallen into pits or eaten by Wumpus. The agent will get a reward if he come out with gold. If agent falls in the pit or eaten by Wumpus then he will get penalty.

There are some points which helps the agent to navigate through the cave safely such as the rooms adjacent to the Wumpus are smelly, so it would have some stench. The room adjacent to pits has a breeze, so when agent reach near pits, he feels the breeze. If the room is glitter, then it Contin gold. The Wumpus can be killed by the agent if is facing it.



The PEAS for the Wumpus world are described as follow

- **Performance measure**- +1000 if the agent come out from the cave with gold. -1000 if agent falls into the pits or eaten by the Wumpus. -1 for each action taken and -10 for using an arrow.

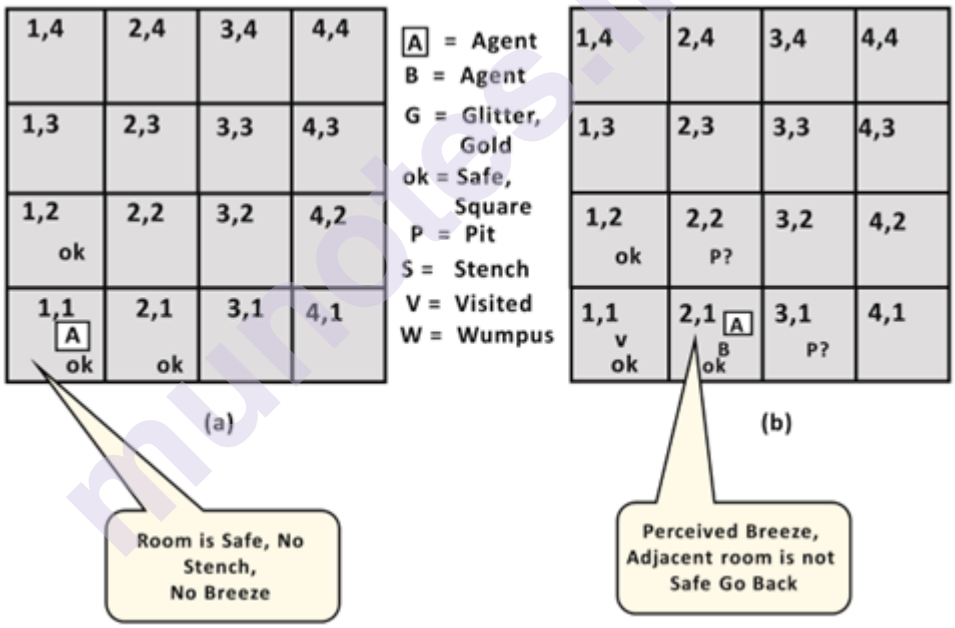
- **Environment:**
  - 4 X 4 grid of rooms connected to each other.
  - The agent always starts from the position [1,1] facing to right.
  - The location of gold and the Wumpus are selected randomly from the squares.
  - Each square of the cave other than the start can be pit with probability 0.2.

- **Actuators:**
  - the agent can move forward
  - turn left and turn right by 90^0.
  - The grab action can be used to pick up the gold.
  - The action shoot can be used to fire an arrow.
  - The arrow continues until it either hits the Wumpus or wall. The agent has only one arrow.
  - The action climb can be used to climb out of the cave.

- **Sensors:** The agent has five sensors as follows.
  - The agent will perceive the stench if he is in the room which is adjacent to Wumpus.
  - It perceives breeze if the room directly adjacent to pit.
  - The glitter is perceived by agent when room contains gold.
  - It perceives bump if agent walk into the wall.
  - When the Wumpus is shot, it releases a horrible scream which can be perceived anywhere in the cave.

The percept will be given to agent program in the form of five symbols if there is a stench and breeze but no glitter, bump or scream the agent program will get [Stench, Breeze, None, None, None]

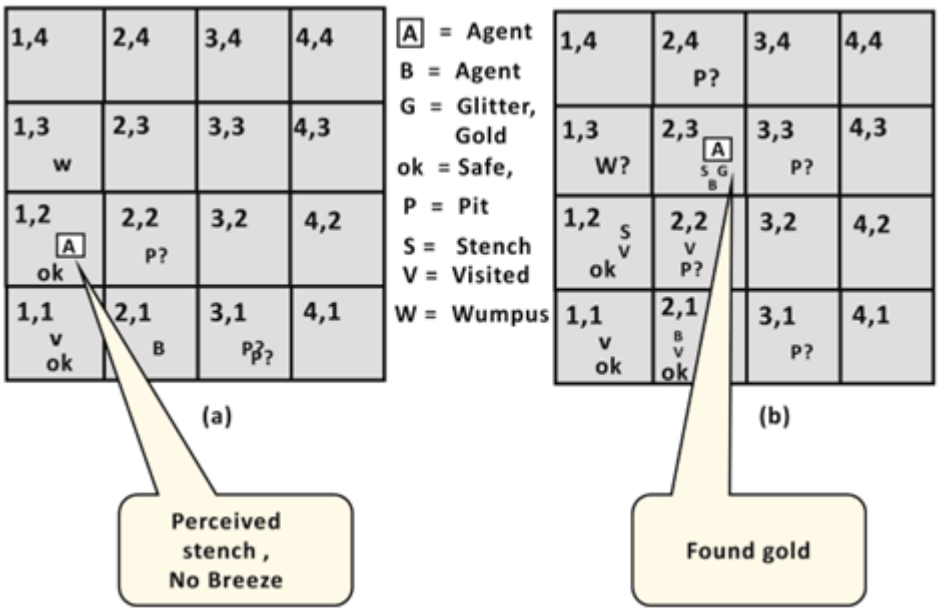
Consider the following example, the agent's initially starts from the location [1,1] and [1,1] is a safe square. Suppose it can be denoted by "A" or "OK" respectively in square [1,1]. So, the first percept is [None, None, None, None, None] from which agent can conclude about neighboring squares such as [1,2] and [2,1] are out of dangers. They are OK.



Let's suppose agent supposed to move in [2,1], so there must be a pit in [2,2] or [3,1] or both. Now at this point only one square is OK [1,2]. So, agent is turn around and go back to [1,1] and then [1,2]

The agent perceives a stench in [1,2] means there must be a Wumpus nearby. But nearby of [1,2] is [1,1] and [2,2] and both are OK. So, agent can guess that Wampus is in [1,3]. The agent feels that there is a lack of a breeze in [1,2] means there is no pit in [2,2]. The agent already inferred that there must be a pit in either [2,2] or [3,1]. So this means it must be in [3,1] .

Now at room [2,2] there is neither pit nor Wumpus so agent moves to [2,3] that detect a glitter. So, agent should grab the gold and return to home.



6.4 LOGIC

The proof or validation of a reason can be defined as Logic. Knowledge base consist of sentence and these sentences are expressed according to the syntax. Syntax is a formal description of the structure of program in a given language. The notion of syntax must be in well forms for example "x+y=10" is a well-formed but "xyz+=" is not in well formed.

A logic can also define the semantic or meaning of sentence. Semantic is a formal description of the programs in a given language. The truth of each sentence with respect to each possible world defined by semantics. For example, consider the sentence "x+y=10" is true in a world where possible value of x is 5 and y is 5, but false where x is 1 and y is1. In standard logic each sentence must be either true or false only. The term Model is used instead of possible world when need to be more precise.

If a sentence α is true in a model then m is a model of α and M(α) notation is used to mean the set of all models of α. Logical reasoning is the process through which any assertion is verified. It is relation of logical entailment between sentences. It can be denoted by α |=β and mean that the sentence α entails the sentence β. The entailment is defined as α |=β if and only if, in every model in which α is true then β is also true and can be written as

α |=β iff M(α) ⊆ M(β)

this analysis can be applied on Wumpus world reasoning example.

6.5 PROPOSITIONAL LOGIC: A VERY SIMPLE LOGIC

Propositional logic is a simplest form of logic in which knowledge is represented as proposition. A proposition is a declarative statement that's either true or false. It is also called as a Boolean logic because it works on 0 and 1. The symbolic variable is used to represent the logic such A, X etc.

Propositional logic consists of object, relation or functions and logical connectivity. In tautology propositional formula is always true whereas in contradiction propositional formula is always false.

6.5.1 Syntax:

Syntax of propositional logic defines allowable sentences in the model. The atomic sentences can be made up of single propositional symbol. Each symbol stands for proposition that can be either true or false. The uppercase symbols are used for example, A,P,R etc. Complex sentences are constructed from simple sentences by using parentheses and logical connectives. There are five connectives are as follows

1) Negation or not- A sentence such as ¬ P is called as negation of P. A literal is either a positive or negative literal.

For example, Sonu did not read the Novels can be represented as ¬READ (Sonu, Novels)

2) and – it is called as conjunction used to represent compound statement. Denoted by ∧.

For example, Sonu is intelligent and hardworking can be represented as follows

Suppose P- Sou is intelligent

Q- Sonu is Hardworking then P∧Q

3) or – A sentence such as P∨Q is called as disjunction.

For example, Sonu is Doctor or Engineer

Suppose P- Sou is Doctor

Q- Sonu is Engineer then P∨Q

4) implies- it is used to represent if then statement and denoted by P=>Q

for example, if it's raining, then streets are wet

Suppose P-if it's raining

Q-Street is wet then P=>Q

5) **if and only if- it is also called as bidirectional connective.** P ⇔ Q show that it is true whenever both P=>Q and Q=>P are true.

For example, Sonu eat the fruit if and only if it's an Apple

Suppose P-Sonu eats fruit

Q- it's an Apple then P ⇔ Q

P	Q	¬P	P ∧ Q	P ∨ Q	P ⇒ Q	P ⇔ Q
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

A BNF(Backus Naur-Form)- A BNF grammar of sentences in propositional logic is described as follows

Sentence->Atomic sentence| Complex sentence

Atomic sentence->True | False | Symbol

Symbol-> P | Q | R….

Complex sentence->¬ Sentence

|(Sentence ∧ Sentence)

|(Sentence ∨ Sentence)

|(Sentence => Sentence)

|(Sentence ⇔ Sentence)

Operator precedence- ¬, ∧, ∨, =>, ⇔

Every sentence constructed with binary connectives must be closed in parenthesis because grammar is very strict about parenthesis.

6.5.2 Semantics:

The semantic defines the rules for determining the truth of a sentence with respect to particular model. In a propositional logic, model fixes the truth value as true or false for every proposition symbol. Suppose if the sentence in the knowledge base makes use of the proposition symbols such as $P_{1,2}$, $P_{2,2}$ and $P_{3,1}$ then model is as follows

M1 = {$P_{1,2}$=false, $P_{2,2}$=false, $P_{3,1}$=false}

With these three symbols there are 2^3=8 possible models. The semantic must specify how to compute the truth value of any sentence and it done recursively. By using atomic sentences and five connectives all sentences are constructed. So, it's necessary to specify how to compute the truth of atomic sentence and sentence which are formed with each of the five connectives. The atomic sentences are easy

- True is true and False is false in every model.
- The truth value of every other proposition symbol must be specified directly in the model.

There are five rules for complex sentence, that hold any substances P and Q in any model m:

- ¬ P is true iff P is false in m.
- **P∧ Q is true iff both P and Q are true in m.**
- **P ∨ Q is true iff either P or Q is true in m.**
- P=>Q is true unless P is true and Q is false in m.
- P ⇔ Q is true iff P and Q are both true or both false in m.

6.5.3 A Simple Knowledge Base:

Now consider immutable aspects of the Wumpus world with the following symbol for each location [x,y] such as

$P_{x,y}$ is true if there is a pit in [x,y]

$W_{x,y}$ is true if there is Wumpus in [x,y] dead or alive

$B_{x,y}$ is true if the agent perceives a breeze in [x,y]

$S_{x,y}$ is true if the agent perceive a stench in {x,y\

Now consider the following situations and label each sentence R_i so it can be referred further.

1) there is no pit in [1,1]

R_1: ¬ $P_{1,1}$

2) A square is breeze if and only if there is pit in a neighboring square

R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \text{ V } P_{2,1})$

R_3: $B_{2,1} \Leftrightarrow (P_{1,1} \text{ V } P_{2,2} \text{ V } P_{3,1})$

3) Now the breeze percept has been included for the first two squares

R_4: ¬ $B_{1,1}$

R5: ¬ $B_{2,1}$

6.5.4 A Simple Inference Procedure:

The first algorithm for inference is model checking approach which is direct implementation of entailment. It checks that α is true in every model in which KB is true. Models are assignments of true or false to every symbol. Now consider Wumpus world example, the proposition symbols are $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $B_{2,1}$, $B_{2,2}$, an $B_{3,1}$ with these seven symbols there are 2^7=128 possible models.

A general algorithm for deciding entailment in propositional logic is as shown below. The algorithm directly implements the definition of entailments so algorithm is sound. The algorithm is complete because it works for any KB and α and always terminate. There are 2^n models if KB and α contains n symbols in all. The algorithm has time complexity $O(2^n)$.

function TT-ENTAILS?(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, *symbols*, { })

```
function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
      if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
      else return true // when KB is false, always return true
  else do
      P ← FIRST(symbols)
      rest ← REST(symbols)
      return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
              and
              TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

6.6 PROPOSITIONAL THEOREM PROVING

To construct the proof of the desired sentence the rules can be apply directly to the sentence in knowledge base without consulting the model. The entailment can be done by theorem proving. If the number of models is large but the length of the proof is short then theorem proving can be more efficient as compare to model checking.

Logical equivalence- two sentences α and β are logically equivalent if they re true in the same set of models. It can be denoted by α ≡ β. Or any two sentences α and β are equivalent only if each of them entails the other and denoted by α ≡ β if and only if α |= β and β |= α.

Validity- a sentence is true in all model then it is valid.

Tautology- A valid sentence is called as tautology.

Deduction theorem- it is derived from the definition of entailment, which was known to the ancient Greeks for any sentences α and β, α |= β if and only if the sentence (α ⇒ β) is valid

Satisfiability- A sentence is satisfiable if it is true in or satisfied by some model. It can be checked by enumerating the possible models until one is found that satisfies the sentence.

6.6.1 Inference and proofs:

Inference rules can be applied to derive a proof.

1) Modus Pones:

It is represented as follows

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$$

It indicates that any sentence of the form α ⇒ β and α are given, then the sentence β can be inferred

For example, if (Wumpus Ahead ∧ Wumpus Alive) => Shoot and (Wumpus Ahead ∧ Wumpus Alive) are given then shoot can be inferred.

2) **And-elimination:** It says that from conjunction any conjuncts can be inferred

$$\frac{\alpha \wedge \beta}{\alpha}$$

for example, from the sentence (Wumpus Ahead ∧ Wumpus Alive) the inference, "Wumpus Alive" can be drawn.

3) **Monotonicity:** it says that the set of entailed sentences can only increases as information is added to the knowledge base. For any sentence α and β

if KB|= α then KB ∧ β|= α

Monotonicity means inference rules can be applied whenever suitable premises are found in the knowledge base. the conclusion of the rule must follow regardless of what else is in the knowledge base.

6.6.2 Proof by Resolution:

The single inference rule called resolution that produce a complete inference algorithm when coupled with any complete search algorithm. Unit resolution rule takes a clause of a disjunction of literals and produce a new clause. A single literal can be viewed as a disjunction of one literal known as a unit clause. the unit resolution rule can be generalized to the full resolution rule

$$\frac{l_1 \vee \ldots \vee l_k \quad m_1 \vee \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n}$$

Where l_i and m_j are complementary literals. So, resolution takes two clauses and generate a new clause. The new clause containing all the literals of the two original clause except two complementary literals. The resulting clause should contain only one copy of each literal is the technical aspect of resolution rule. The removal of multiple copies of literals is called factoring.

Conjunctive Normal Form:

Every sentence of propositional logic is logically equivalent toa conjunction of clauses. A sentence can be expressed as a conjunction of clause is called as Conjunctive normal form or CNF. The following s procedure for converting sentence B1,1⇔(P1,2 ∨ P2,1) into CNF

1) Eliminate ⇔, by replacing α ⇔ β with (α ⇒ β) ∧ (β ⇒ α)

(B1,1⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .

2) Eliminate ⇒, by replacing α ⇒ β with ¬α ∨ β

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬(P1,2 ∨ P2,1) ∨ B1,1)

3) CNF requires ¬ to appears only in literals, so it is necessary to move ¬ inwards by repeated application of the following equivalence.

¬(¬α) ≡ α (double-negation elimination)

¬(α ∧ β) ≡ (¬α ∨ ¬β) (De Morgan)

¬(α ∨ β) ≡ (¬α ∧ ¬β) (De Morgan)

We require only one application of the last rule in the example.

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬(P1,2 ∧ P2,1) ∨ B1,1)

4) Now there is a sentence containing nested ∧ and ∨ operators applied to literals. Now by applying distributive law, distributing ∨ over ∧ whenever possible

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬(P1,2 ∨ B1,1) ∧ (¬P2,1 ∨ B1,1)

The original sentence is now converted in CNF as a conjunction of three clauses. It can be use as input to resolution procedure.

A resolution Algorithm:

It takes the input as a knowledge base and α and its output is either true or false. First (KB ∧ ¬ α) is converted into CNF. Then the resolution rule is applied to the resulting clauses. Each pair in which complementary literals are there is resolved to produce anew clause and it is added to the new set if it is not there. The procedure continues until one of the two things happens first, there are no new clauses that can be added in which KB does not entails α or second two clauses resolve to yield the empty clause in which case KB entails α. Th resolution algorithm is as shown belove.

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new ← { }
  loop do
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new ← *new* ∪ *resolvents*
    if *new* ⊆ *clauses* **then return** *false*
    clauses ← *clauses* ∪ *new*

6.6.3 Horn Clauses and Definite Clauses:

Some real-world knowledge base uses more restrictions on the form of sentences to enables them to use more restricted and efficient inference algorithm. The definite clause is one such restricted form, which is a disjunction of literals with exactly one positive literal.

For example, ($\neg L_{1,1} \vee \neg Breeze \vee B_{1,1}$) is a definite clause whereas ($\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$) is not definite clause.

Horn clause is a disjunction of literals with at most one positive literal. So, all definite clauses are horn clauses. Goal cluses are those in which there is no positive literals.

6.6.4 Forward and Backward Chaining:

Forward chaining is an example of the data driven reasoning concept that is reasoning in which the focus attention starts with the known data.it is called as data driven as it reach the goal state using available data. Forward chaining is the process of making conclusion based on known fact by starting from initial state to goal state. It can be used within an agent to derive conclusion from incoming percepts. While doing this, it cannot be kept any query in mind. The forward chaining is used in expert system. For example, the Wumpus agent might TELL its percept to knowledge base using an incremental forward chaining algorithm.

Now consider example following example,

As per the law, it is crime for B country to sell weapons to hostile nations. Country A is an enemy of Country B, has some missiles, and all the missiles were sold to it by John. John is a B country citizen now prove that John is criminal. Now convert all facts into first order definite clause.

It is a crime for a country B to sell weapons to hostile nation. Let's consider variables p, q and r.

Rules:

1) Country B(p) ∧ **weapon (q) ∧ sells (p,q,r) ∧ hostile (r)-> criminal (p)**
2) **Country A has some missiles. ?powns(A,p) ∧ missile(p) or Owns(A,T1) Missiles(T1)**
3) **All missiles are sold to country A by John. ?p Missiles(p) ∧ Owns(A, p)-> Sells (John, p, A)**
4) **Missiles are weapons. Missile (p) -> Weapons (p)**
5) **Enemy of country B is hostile. Enemy (p, B)-> Hostile(p)**
6) **Country A is an enemy of Country B. Enemy (A , B)**
7) **John is from country B. Country B(John)**

Steps:

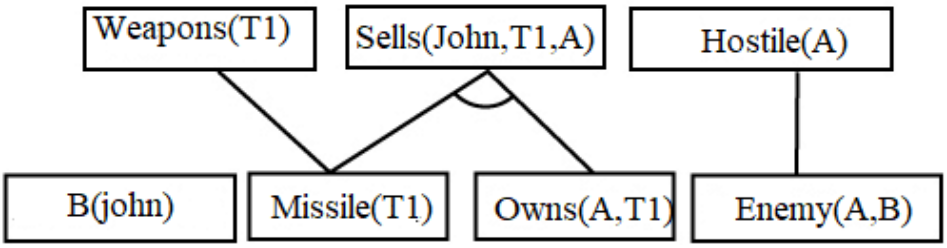
1) First start with the known facts and select the sentences which do not have implications such as B(Roberts), Enemy(A, B), Owns(A, T1) and Missile(T1) can be represented as below
2) The facts thar are infer from available facts and with satisfied premises

Rule 1- does not satisfy premises, so it will not be added in the first iteration

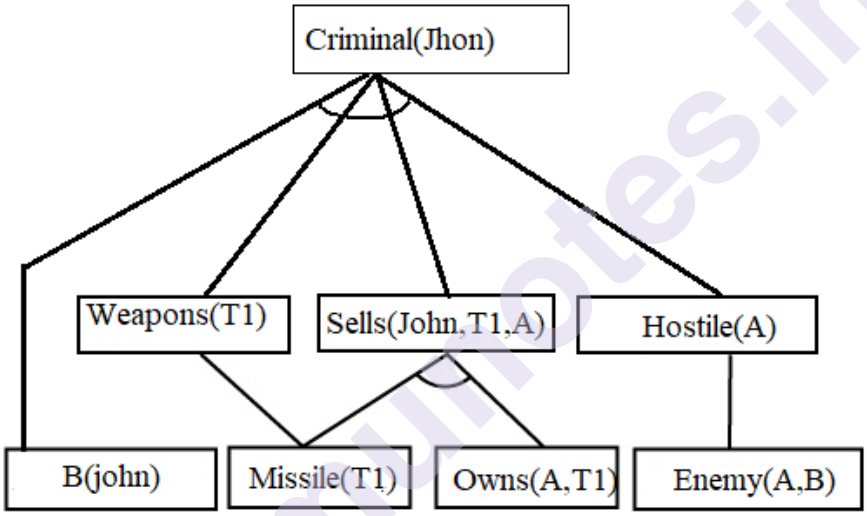
Rule 2 and 3 are already added.

Rule 4 -satisfy the substitution {p/T1}, so Sells (John, T1, A) is added. It infers from conjunction of rue 2 and 3

Rule 6- is satisfy with substitution (p/A) so Hostile (A) is added. It infers from rule 7.



3) Check rule 1 is satisfied with the substitution {p/John, q/T1, r/A}, so add Criminal(John). So, we reached at goal state and proved that Jhon is Criminal.



Backward Chaining: In backward chaining it starts with the goal and works backward from the query. If the query is known to be rule, then no work is needed. Otherwise, the algorithm finds implications that can be proved true then q is true. It is form of goal-directed reasoning. In this the goal is broken into sub-goals to prove the facts true. In backward chaining same example is consider.

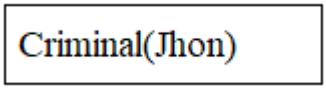
Rules:

1) Country B(p) **∧ weapon (q) ∧ sells (p,q,r) ∧ hostile (r)-> criminal (p)**
2) **Country A has some missiles. ?powns(A,p) ∧ missile(p) or Owns(A,T1) Missiles(T1)**
3) **All missiles are sold to country A by John. ?p Missiles(p) ∧ Owns(A, p)-> Sells (John, p, A)**
4) **Missiles are weapons. Missile (p) -> Weapons (p)**

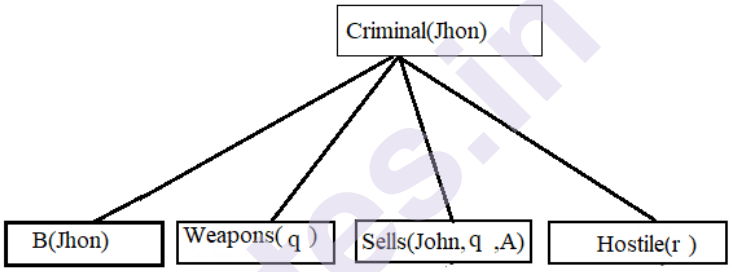
5) **Enemy of country B is hostile. Enemy (p, B)-> Hostile(p)**
6) **Country A is an enemy of Country B. Enemy (A , B)**
7) **John is from country B. Country B(John)**

Steps- in backward chaining start with goal state which is Criminal (Jhon)

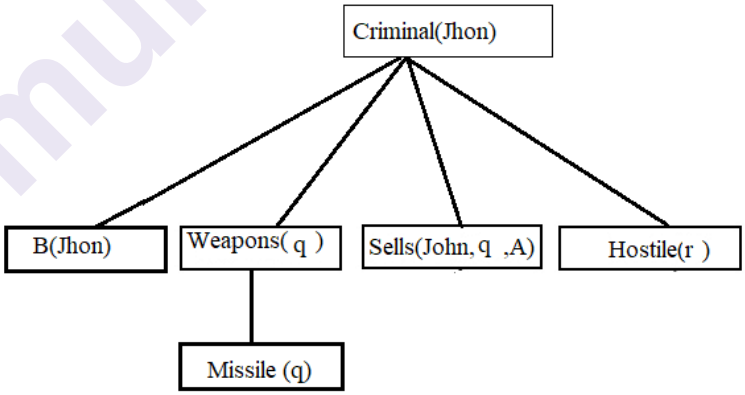
1) We will take the goal fact and from this infer other facts. At the end prove that those facts true. So, now goal fact is Jhon is Criminal as shown below



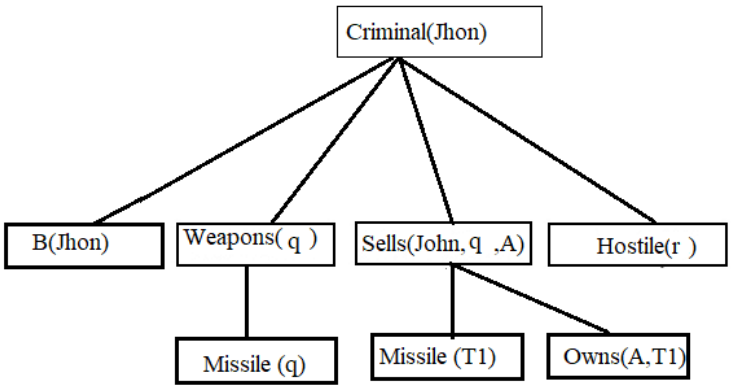
2) We will infer other facts from all fact which satisfies the rules. In rule 1 goal predicate is present with substitution {Jhon/p}. So, add all the conjunctive facts below the first level and replace p with Jhon.



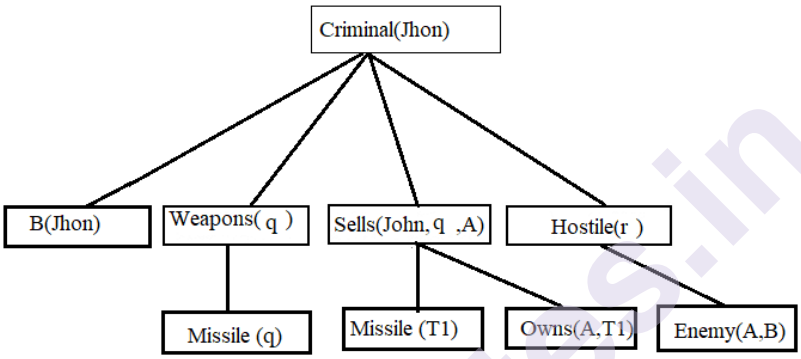
3) now extract further fact Missile(q) which infer from Weapon(q) as it satisfies rule 5 Weapon(q) is also true with the substitution of a constant T1 at q



4) now infer facts Missile(t1) and Owns (A, T1) from Sells (Jhon, T1, r) which satisfies the rule 4 with substitute of A in place of r. so two statements are proved here.



5) Infer the fact Enemy (A, B) from Hostile(A) which satisfies rule 6 an hence all the statements are proved true using backward chaining.



6.7 EFFECTIVE PROPOSITIONAL MODEL CHECKING

For better inferences efficient algorithm is needed that based on model checking. Algorithm approach can be backtracking search or hill-climbing search.

6.7.1 A Complete Backtracking Algorithm:

This algorithm is also called as Davis-Putnam algorithm or DPLL algorithm. It takes as input a sentence in conjunction normal form, which is set of clauses. It is recursive depth first enumeration of possible models. The algorithm is improvement over earlier general inferencing algorithm. There are three improvements over the simple scheme TT-ENTAILS.

1) **Early termination:** The algorithm finds whether the sentence must be true or false. A clause is true if any literal is true. A sentence is false if any clause is false, which occurs when each of its literal is false.

2) **Pure symbol heuristic:** it is a symbol that always appears with same "sign" in all clauses. If a sentence has a model, then it has a model with pure symbol assigned so as to make literal true.

3) **Unit clause heuristic:** it is clause with just one literal. One unit clause can create another unit clause. The only literal in unit clause is true.

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })
```

```
function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P=true}) or
        DPLL(clauses, rest, model ∪ {P=false}))
```

6.7.2 A Local Search Algorithm:

The local search algorithm like hill-climbing can be applied directly to satisfiable problem. These algorithms are worked properly if correct evaluation function is provided. The task of algorithm is to find an assignment that satisfies every clause. Therefore, those evaluation function counts the number of unsatisfied clauses will be a good.

WALKSAT algorithm picks up an unsatisfied clause and pick a symbol in the clause to flip. It randomly chooses one of the ways to pick which symbol to flip is min-conflict and random walk. Min- conflict is steps that minimizes the number of unsatisfied clauses in the new state. Random walks that pick the symbol randomly.

When the input sentence is indeed satisfiable WALKSAT returns a model. But when it is return failure then there are two possible causes. First, the sentence is unsatisfiable and second is need to give the algorithm more time. WALKSAT algorithm cannot always detect unsatisfiability, which is very necessary for deciding entailment.

```
function WALKSAT(clauses, p, max-flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move
          max-flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max-flips do
      if model satisfies clauses then return model
      clause ← a randomly selected clause from clauses that is false in model
      with probability p flip the value in model of a randomly selected symbol
          from clause
    else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

6.7.3 The Landscape of random SAT Problems

There are some SAT problems that are harder than others. Simple problems can be solved with any old algorithm. Because SAT is NP-complete, at least some problem instances will require exponential running time. The n-queen problem quite tricky for backtracking search algorithm but trivially easy for local search methods such as min conflict. This is because solutions are very densely distributed in the space of assignments, and any initial assignment is guaranteed to have a nearby solution. Therefore, n-queens is easy because it is underconstrained.

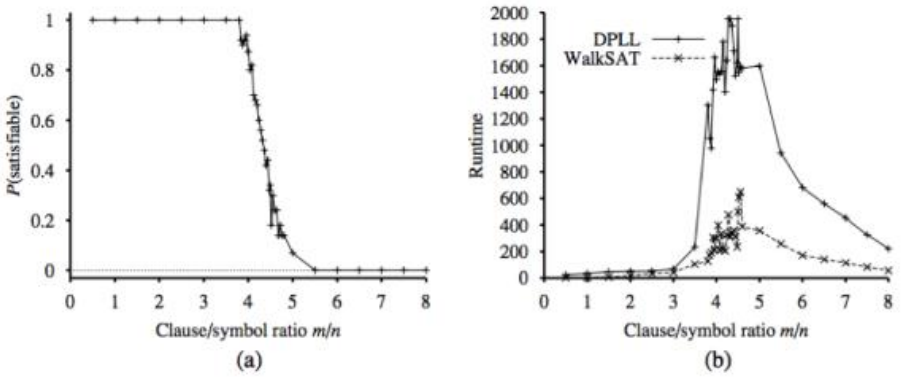
In the case of satisfiability problems in conjunctive normal form, an underconstrained problem is one with relatively few clauses constraining the variables. Consider the following example with a randomly generated 3-CNF sentence with five symbols and five clauses

$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$.

For this sentence sixteen of the 32 possible assignments are model, so, it would take just two random guesses to find a model. In general, underconstrained problems like this are easily satisfiable. In contrast, an overconstrained problem will have many clauses in relation to the number of variables and is likely to have no solution.

Beyond these basic intuitions, we have to define how random sentences are generated. The notation $CNF_k(m,n)$ denotes a k-CNF sentence with m cluses and n symbols, where the clauses are chosen uniformly, independently and without replacement from all clauses with k different literals, which are positive or negative at random.

The probability of satisfiability can be measured by examining the sources of random sentences. As shown in following figure (a), plots the probability for $CNF_3(m,50)$, that is, sentences with 50 variables and 3 literals per clause, as a function of clause/ symbol ratio, m/n. For small m/n the probability of satisfiability is close to 1, whereas for large m/n, the probability is close to 0. The probability drops fairly sharply around m/n=4.3. so, as n increases "cliff" gets sharper and sharper and stay in roughly the same place for k=3. Theoretically, the satisfiability threshold conjecture says that for every $k \geq 3$, there is a threshold ratio r_k such that, as n goes to infinity, the probability that $CNF_k(n, rn)$ is satisfiable becomes 1 for all values of r below the threshold, and 0 for all values above the threshold. The conjecture remains unproven.



(a) Graph showing the probability that a random 3-CNF sentence with n = 50 symbols is satisfiable, as a function of the clause/symbol ratio m/n. (b) Graph of the median run time on random 3-CNF sentences

6.8 AGENT BASED ON PROPOSITIONAL LOGIC

Inference based agent -it requires the separate copies of its knowledge base for every time step. Inference based agent takes exponential time for inferencing and huge time requirements for completeness. It is easy to represent in propositional logic language. It requires large memory as it stores previous percepts or states.

Circuit based agent- it doesn't require separate copy of its KB for every time step. Circuit based agent takes linear time for inferencing which is dependent on circuit size. For a complete circuit CBA will take exponential time with respect to circuit size so it is incomplete. It is easy to describe and construct knowledge base.

6.9 SUMMARY

- Intelligent agent require knowledge about the world in order to take good decision. Knowledge is contained in agent in the form of sentences.
- Sentences in a knowledge representation language are stored in a knowledge base.
- Basic concept of logic is represented by syntax, semantics, entailment, inference, soundness and completeness.
- Syntax is a formal structure of sentences. Semantics is truth of sentences based on models.
- Entailments is necessary truth of one sentence given another. Inference is deriving sentences from other sentences
- In sound inference algorithm, derivations produce only entailed sentences. In complete algorithm, derivations can produce all entailed sentences.
- Propositional logic isa simple language consisting of proposition symbols and logical connectives.
- Resolution is complete for propositional logic, forward, backward chaining are linear time, complete for Horn clauses.

6.11 UNIT END QUESTIONS

1) Explain knowledge-based agent? Explain role and importance.
2) What is knowledge based
3) Write a short note on Wumpus world problem.
4) Describe PEAS for Wumpus world problem.
5) What is logic?
6) Explain propositional logic.
7) Explain resolution algorithm.
8) Explain CNF
9) Write a short note on propositional theorem proving.
10) Write the connectives used to form complex sentence of propositional logic with example.
11) Explain backward and forward chaining with example.

6.10 BIBLIOGRAPHY

1) Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig
2) Artificial Intelligence: javapoint.com

*****

UNIT IV

7

FIRST-ORDER LOGIC

Unit Structure

7.0 OBJECTIVES

After going through this chapter, you will be able to:

- Represent knowledge in knowledge base using First-Order logic.
- Understand syntax and semantics of First-Order logic.
- Make use of Quantifiers for representation of knowledge.
- Learn use of First-Order logic in kinship domain, number and sets, and in Wumpus world.
- Understand Knowledge-Engineering process.
- Apply First-Order logic in electronic circuit domain.

7.1 INTRODUCTION

Statements can be represented using propositional logic. But unfortunately, in propositional logic, we can only represent the facts, which are either true or false. Propositional logic is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. Propositional logic lacks the expressive power to concisely describe an environment with many objects. Consider the following sentence, which cannot be represented using propositional logic.

- "Some humans are intelligent"
- "Sachin likes cricket."

To represent the above statements, propositional logic is not sufficient, so we need some more powerful logic such as first-order logic. We can adopt the foundation of propositional logic that is a declarative, compositional semantics which is context-independent and unambiguous and build a more expressive logic on that foundation, by borrowing representational ideas from natural language while avoiding its drawbacks. Important elements in natural language are:

- Objects (squares, pits, Wumpus)
- Relations among objects (is adjacent to, is bigger than) or unary relations or properties (is red, round)
- Functions (father of, best friend of)

First-order logic (FOL) is built around the above 3 elements that is objects, relations & functions.

7.2 FIRST-ORDER LOGIC

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
- First-Order logic is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as Predicate logic or First-order predicate logic. First-order logic is a powerful language that develops information about the objects in an easier way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:
  - **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, etc.
  - **Relations:** It can be unary relation such as: red, round, is adjacent, or n-any relation such as: the sister of, brother of, has color, comes between.

- **Function:** Father of, best friend, third inning of, end of, etc.
- As a natural language, first-order logic also has two main parts:
- Syntax
- Semantics

7.3 SYNTAX & SEMANTICS OF FIRST-ORDER LOGIC

We begin this section by specifying the way in which the possible worlds of first-order logic reflect the ontological commitment to objects and relations. Then we introduce the various elements of the language, explaining their semantics as we go along.

7.3.1 Models for First-Order Logic:

Models of a logical language are the formal structures that constitute the possible worlds under consideration. Models for propositional logic link proposition symbols to predefined truth values. Models for first-order logic have objects. Every possible world must contain at least one object.

Below figure shows a model with five objects:

1. Richard the Lionheart, King of England;
2. his younger brother, the evil King John;
3. the left legs of Richard;
4. the left legs of John;
5. and a crown.

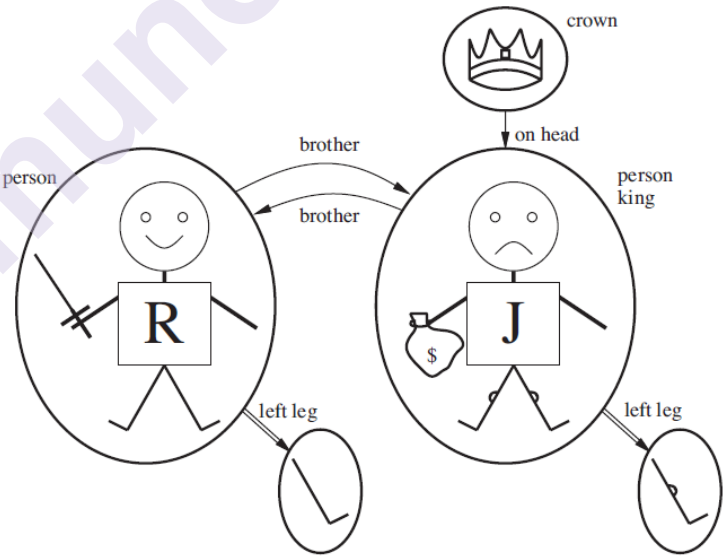


Figure: A model with five objects

The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Thus, the brotherhood relation in this model is the set

{ <Richard the Lionheart, King John>, <King John, Richard the Lionheart> }

The "brother" and "on head" relations are binary relations that is, they relate pairs of objects. The model also contains unary relations, or properties: the "person" property is true of both Richard and John; and the "crown" property is true only for the crown.

Certain kinds of relationships are considered as functions, in that a given object must be related 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.

7.3.2 Symbols and Interpretations:

The syntax determines which collections of symbols are legal expressions in first-order logic. The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds:

1. Constant symbols:

Constants symbols refer to objects in a universe of discourse. Objects can be anything like integers, people, real numbers and geometric shapes etc.

Example: Richard, John

2. Predicate symbols:

A predicate symbol represents a property of or relation between terms that can be true or false. Each predicate symbol has an associated arity indicating its number of arguments. Brother, OnHead, Person, King, and Crown, etc. are predicate symbols.

Example:

King(John) is a unary predicate which is having one arguments.

Brother(John, Richard) is a Binary predicate which is having two arguments.

3. Function symbols:

Function constants refer to functions like mother, age, LeftLeg, plus and times. Each function symbol has an associated arity indicating its number of arguments. Example:

mother(Jane) - here, mother has arity one (unary) while in times(3,2) - times has arity two(binary).

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

Term

ComplexSentence → (Sentence) | [Sentence]

| ¬Sentence

| Sentence ∧ Sentence

| Sentence ∨ Sentence

| Sentence ⇒ Sentence

| Sentence ⇔ Sentence

| Quantifier Variable, . . . Sentence

Term → Function(Term, . . .)

| Constant

| Variable

Quantifier → ∀| ∃

Constant → A | X_1 | John | · · ·

Variable → a | x | s | · · ·

Predicate → True | False | After | Loves | Raining | · · ·

Function → Mother | LeftLeg | · · ·

OPERATOR PRECEDENCE : ¬,=, ∧, ∨,⇒,⇔

Figure: The syntax of first-order logic, specified in Backus–Naur form and Operator precedences are specified from highest to lowest.

Like proposition symbols, the choice of names is entirely up to the user & every model must provide the information required to determine if any given sentence is true or false.

In addition to its objects, relations, and functions, each model includes an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.

One possible interpretation for our example is as follows:

Interpretation:

- Richard refers to Richard the Lionheart and John refers to the evil King John.
- Brother refers to the brotherhood relation; OnHead refers to the "on head" relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns.

- LeftLeg refers to the "left leg" function.

7.3.3 Terms:

Objects are represented by terms. Terms are simply names for objects. A term is a logical expression that refers 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. At that time instead of using a constant symbol, we use LeftLeg(John). A complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol.

Complex term is not a "subroutine call" that "returns a value." There is no LeftLeg subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing that John must have one) without ever providing a definition of LeftLeg.

For example, suppose the LeftLeg function symbol refers to the function and John refers to King John, then LeftLeg(John) refers to King John's left leg.

7.3.4 Atomic Sentences:

A sentence can either be an atomic sentence or a complex sentence. An atomic sentence is simply a predicate applied to a set of terms. Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms. We can represent atomic sentences as

Predicate (term1, term2, ..., term n)

Example:

1. Ravi and Ajay are brothers: Brothers(Ravi, Ajay).
2. Chinky is a cat: cat (Chinky).
3. John owns a car: Owns(John, Car)

Atomic sentences can also have complex terms as arguments. Such as,

Married(Father (Ravi),Mother (Ajay))

states that Ravi's father is married to Ajay's mother.

7.3.5 Complex Sentences:

Logical connectives can be used to construct more complex sentences, with the same syntax and semantics as in propositional calculus. Logical connectives like ¬, ∧, ∨, ⇒, ⇔ can be used.

(Sentence **)** | **[** Sentence **]**

¬Sentence

Sentence ∧ Sentence

Sentence ∨ Sentence

Sentence ⇒ Sentence

Sentence ⇔ Sentence

Quantifier Variable, . . . Sentence

Here we have four sentences that are true in the model that we have considered previously.

¬Brother (LeftLeg(Richard), John)

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

King(Richard) ∨ King(John)

¬King(Richard) ⇒ King(John)

7.3.6 QUANTIFIERS

A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse. These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of Quantifiers.

1. Universal Quantifier, (for all, everyone, everything)
2. Existential quantifier, (for some, at least one).

1. Universal Quantification (∀):

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing. The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.

Note: In universal quantifier we use implication "→".

If x is a variable, then ∀x is read as:

For all x

For each x

For every x.

Example 1:

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

∀ x King(x) ⇒ Person(x)

Thus, the sentence will be read as, "For all x, if x is a king, then x is a person." The symbol x is called variable. By convention, variables are lowercase letters.

Variable:

- A variable is a term by itself.
- It can also serve as the argument of a function for example, LeftLeg(x).
- A term with no variables is called a ground term.

Consider the model that we have used previously and the intended interpretation that goes with it. We can extend the interpretation in five ways by asserting all possible values of x:

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 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:

i. Richard the Lionheart is a king ⇒ Richard the Lionheart is a person.
ii. King John is a king ⇒ King John is a person.
iii. Richard's left leg is a king ⇒ Richard's left leg is a person.
iv. John's left leg is a king ⇒ John's left leg is a person.
v. The crown is a king ⇒ the crown is a person.

Since, in our model, King John is the only king, the second sentence asserts that he is a person and remaining four assertions are not valid because Richard is not king and left leg of a person and crown object cannot be a person.

Example 2:

All man drink coffee.

Let a variable x which refers to a man so all x can be represented as below:

x1 drinks coffee.

x2 drinks coffee.

x3 drinks coffee.

.

.xn drinks coffee.

We can write it as:

∀x man(x) → drink (x, coffee).

It will be read as: For all x, if x is a man, then x drinks coffee.

2. Existential Quantification:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something. It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier. The sentence ∃x P says that P is true for at least one object x.

Note: In Existential quantifier we always use AND or Conjunction symbol (∧).

If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

There exists a 'x.'

For some 'x.'

For at least one 'x.'

Example 1:

King John has a crown on his head, we write

∃ x Crown(x) ∧ OnHead(x, John)

That is, at least one of the following is true:

i. Richard the Lionheart is a crown ∧ Richard the Lionheart is on John's head;
ii. King John is a crown ∧ King John is on John's head;
iii. Richard's left leg is a crown ∧ Richard's left leg is on John's head;
iv. John's left leg is a crown ∧ John's left leg is on John's head;
v. The crown is a crown ∧ the crown is on John's head.

The fifth assertion is true in the model.

Example 2:

Some boys are intelligent.

∃x: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

Properties of Quantifiers:

- In universal quantifier, ∀x∀y is similar to ∀y∀x.
- In Existential quantifier, ∃x∃y is similar to ∃y∃x.
- ∃x∀y is not similar to ∀y∃x.

Nested quantifiers:

More complex sentences can be expressed 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)

Siblinghood is a symmetric relationship, we can write

∀ x, y Sibling(x, y) ⇔ Sibling(y, x)

In other cases, we will have mixtures. "Everybody read few books" means that for every person, there is some book that person read:

∀ x ∃ y Read(x, y)

On the other hand, to say "There is someone who is liked by everyone," we write

∃ y ∀ x Likes(x, y)

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

Connections between ∀ and ∃:

The two quantifiers are actually 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)

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)

Because ∀ is a conjunction over the universe of objects and ∃ is a disjunction, quantifiers obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

∀ x ¬P ≡ ¬∃x P ¬(P ∨ Q) ≡ ¬P ∧¬Q

¬∀x P ≡ ∃x ¬P ¬(P ∧ Q) ≡ ¬P ∨¬Q

∀x P ≡ ¬∃x ¬P P∧ Q ≡ ¬(¬P ∨¬Q)

∃x P ≡ ¬∀x ¬P P∨ Q ≡ ¬(¬P ∧¬Q)

Examples of First-Order Logic:

Some Examples of First-Order Logic using quantifiers:

1. All birds fly.

In this statement the predicate is fly(bird).

And since there are all birds who fly so it will be represented as follows.

$$\forall x \text{ bird}(x) \rightarrow \text{fly}(x)$$

2. Every man respects his parent.

In this statement, the predicate is respect(x, y), where x=man, and y= parent.

Since there is every man so will use ∀, and it will be represented as follows:

$\forall x \text{ man}(x) \rightarrow \text{respects}(x, \text{parent})$

3. Some boys play cricket.

In this statement, the predicate is play(x, y), where x= boys, and y= game. Since there are some boys so we will use ∃, and it will be represented as:

$\exists x \text{ boys}(x) \rightarrow \text{play}(x, \text{cricket})$

4. Not all students like both Mathematics and Science.

In this statement, the predicate is like(x, y), where x= student, and y= subject.

Since there are not all students, so we will use ∀ with negation, so following representation for this:

$\neg\forall (x) [\text{student}(x) \rightarrow \text{like}(x, \text{Mathematics}) \wedge \text{like}(x, \text{Science})]$

5. Only one student failed in Mathematics.

In this statement, the predicate is failed(x, y), where x= student, and y= subject.

Since there is only one student who failed in Mathematics, so we will use following representation for this:

$\exists(x) [\text{student}(x) \rightarrow \text{failed}(x, \text{Mathematics}) \wedge\forall (y) [\neg(x==y) \wedge \text{student}(y) \rightarrow \neg\text{failed}(x, \text{Mathematics})]$

7.3.7 Equality:

Equality symbol can be used 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)

The sentence ∃ x, y Brother (x,Richard) ∧ Brother (y,Richard) does not have the intended meaning. If x is John & y is also John the sentence will not be valid because Richard's both the brothers will not be having the same name as John, therefore x and y both should be different. For this purpose, ¬(x=y) is added. The notation x ≠ y is sometimes used as an abbreviation for ¬(x=y).

7.3.8 An Alternative Semantic:

Continuing the example from the previous section, suppose that we believe that Richard has two brothers, John and Geoffrey. If we assert

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

First, this assertion is true in a model where Richard has only one brother, we need to add John ≠ Geoffrey. Second, the sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey. Thus, the correct translation of "Richard's brothers are John and Geoffrey" is as follows:

Brother (John,Richard) ∧ Brother (Geoffrey,Richard) ∧ John ≠ Geoffrey ∧ ∀x Brother (x,Richard) ⇒ (x=John ∨ x=Geoffrey)

7.4 USING FIRST-ORDER LOGIC

Now we have defined an expressive logical language, it is time to learn how to use it. In this section, we provide representations of some simple domains. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge. We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of

1. Family relationships
2. Numbers
3. Sets and lists
4. The Wumpus world

7.4.1 Assertions and Queries In First-Order Logic:

1. Assertions:

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, Richard is a person, and all kings are persons:

TELL(KB, King(John)) .

TELL(KB, Person(Richard)) .

TELL(KB, ∀ x King(x) ⇒ Person(x)) .

2. Queries:

We can ask questions from the knowledge base using ASK. For example,

ASK(KB, King(John))

Above query returns true. Questions asked with ASK are called queries or goals.

ASK(KB, Person(John))

This query also returns true using above assertions. We can ask quantified queries, such as

ASK(KB, ∃ x Person(x)) .

The answer is true, but we do not know the value of x makes the sentence true. If we want to know what value of x makes the sentence true, we will use a different function which is called ASKVARS.

ASKVARS(KB, Person(x))

Above function yields a stream of answers. In this case there will be two answers: {x/John} and {x/Richard}. Such an answer is called a substitution or binding list. Which means in above query x can be substituted with John or Richard.

7.4.2 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 William" and rules such as "One's grandmother is the mother of one's parent". We have following in Kinship domain according to First-Order logic.

1. **Objects**: objects in kinship domain are people.
2. **Unary predicates:** Male and Female.
3. **Binary Predicates:** Kinship relations parenthood, brotherhood, marriage, and so on are represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle.
4. **Functions:** Mother and Father, because every person has exactly one of each of these (at least according to nature's design).

We can go through each function and predicate, writing down what we know in terms of the other symbols.

Example:

1. One's mother is one's female parent:

∀ m, c Mother (c)=m ⇔ Female(m) ∧ Parent(m, c)

2. One's husband is one's male spouse:

∀ w, h Husband(h,w) ⇔ Male(h) ∧ Spouse(h,w)

3. Male and female are disjoint categories:

∀ x Male(x) ⇔ ¬Female(x)

4. Parent and child are inverse relations:

∀ p, c Parent(p, c) ⇔ Child (c, p)

5. A grandparent is a parent of one's parent:

∀ g, c Grandparent (g, c) ⇔ ∃p Parent(g, p) ∧ Parent(p, c)

6. A sibling is another child of one's parents:

∀ x, y Sibling(x, y) ⇔ x = y ∧ ∃p Parent(p, x) ∧ Parent(p, y)

We could go on for several examples like this. Each of these sentences can be viewed as an axiom of the kinship domain. The axioms define the Mother function and the Husband, Male, Parent, Grandparent, and Sibling predicates. This is a natural way in which we can build up the representation of a domain.

Axioms can also be "just plain facts," such as Male(Jim) and Spouse(Jim, Laura) etc. Not all logical sentences about a domain are axioms. Some are theorems that is, they are entailed by the axioms. For example, consider the assertion that siblinghood is symmetric:

∀ x, y Sibling(x, y) ⇔ Sibling(y, x)

7.4.3 Numbers, Sets:

I. Numbers:

Numbers are the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of natural numbers or non-negative integers. We will use a predicate NatNum that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor). The Peano axioms define natural numbers and addition. Natural numbers are defined recursively:

NatNum(0) .

That is, 0 is a natural number and

∀ n NatNum(n) ⇒ NatNum(S(n))

Means for every object n, if n is a natural number, then S(n) is a natural number. So the natural numbers are 0, S(0), S(S(0)), and so on.

We also need axioms to constrain the successor function:

$\forall$ n 0 $\neq$ S(n)

$\forall$ m, n m $\neq$ n $\Rightarrow$ S(m) $\neq$ S(n)

Now we can define addition in terms of the successor function:

$\forall$m NatNum(m) $\Rightarrow$ + (0,m) = m

Above axiom says that adding 0 to any natural number m gives m itself.

$\forall$ m, n NatNum(m) $\wedge$ NatNum(n) $\Rightarrow$ + (S(m), n) = S(+(m, n))

We can also write S(n) as n+ 1, so the second axiom becomes

$\forall$ m, n NatNum(m) $\wedge$ NatNum(n) $\Rightarrow$ (m + 1) + n = (m + n) + 1

This axiom reduces addition to repeated application of the successor function.

Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms.

ii. Set:

The domain of sets is also fundamental to mathematics as well as to commonsense reasoning. We want to be able to represent individual sets, including the empty set. We need a way to build up sets by adding an element to a set or taking the union or intersection of two sets. We will want to know whether an element is a member of a set and we will want to distinguish sets from objects that are not sets.

1. The empty set is a constant written as {}.
2. There is one unary predicate, Set, which is true of sets.
3. The binary predicates are x$\in$ s (x is a member of set s) and s1 $\subseteq$ s2 (set s1 is a subset of set s2).
4. The binary functions are s1 $\cap$ s2 (the intersection of two sets), s1 $\cup$ s2 (the union of two sets), and {x|s} (the set resulting from adjoining element x to set s).

One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:

$\forall$ s Set(s) $\Leftrightarrow$ (s={}) $\vee$ ($\exists$ x, s2 Set(s2) $\wedge$ s={x|s2})

2. The empty set has no elements adjoined into it. In other words, there is no way to decompose {} into a smaller set and an element:

¬∃ x, s {x|s}={}

3. Adjoining an element already in the set has no effect:

∀ x, s x∈ s ⇔ s={x|s}

4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s2 adjoined with some element y, where either y is the same as x or x is a member of s2:

∀ x, s x∈ s ⇔ ∃y, s2 (s={y|s2} ∧ (x=y ∨ x∈ s2))

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:

∀ s1, s2 s1 ⊆ s2 ⇔ (∀x x∈ s1 ⇒ x∈ s2)

6. Two sets are equal if and only if each is a subset of the other:

∀ s1, s2 (s1 =s2) ⇔ (s1 ⊆ s2 ∧ s2 ⊆ s1)

7. An object is in the intersection of two sets if and only if it is a member of both sets:

∀ x, s1, s2 x∈ (s1 ∩ s2) ⇔ (x∈ s1 ∧ x∈s2)

8. An object is in the union of two sets if and only if it is a member of either set:

∀ x, s1, s2 x∈ (s1 ∪ s2) ⇔ (x∈ s1 ∨ x∈s2)

7.4.4 The Wumpus World:

The first-order axioms for Wumpus world are much more concise, capturing in a natural way exactly what we want to say. Wumpus agent receives a percept sequence 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 action 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:

∀ t, s, g, m, c Percept ([s, Breeze, g,m, c], t) ⇒ Breeze(t) ,

∀ t, s, b, m, c Percept ([s, b, Glitter,m, c], t) ⇒ Glitter (t) ,

and so on. Notice the quantification over time t. In propositional logic, we would need copies of each sentence for each time step.

Simple "reflex" behavior can also be implemented by quantified implication sentences.

For example,

∀ 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.

We have represented the agent's inputs and outputs; now it is time to represent the environment itself.

Objects are squares, pits, and the wumpus. We could name each square $Square_{1,2}$ and so on but then the fact that $Square_{1,2}$ and $Square_{1,3}$ are adjacent would have to be an "extra" fact, and we would need one such fact for each pair of squares. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term [1, 2]. Adjacency of any two squares can be defined as

∀ x, y, a, b Adjacent ([x, y], [a, b]) ⇔(x = a ∧ (y = b − 1 ∨ y = b + 1)) ∨ (y = b ∧ (x = a − 1 ∨ x = a + 1))

We will use a unary predicate Pit that is true of squares containing pits. Since there is exactly one wumpus, we will use a constant Wumpus.

The agent's location changes over time, so we write At(Agent, s, t) to mean that the agent is at square s at time t. We can fix the wumpus's location with

∀t At(Wumpus, [2,2], t)

We can then say that objects can only be at one location at a time:

∀ x, s1, s2, t At(x, s1, t) ∧ At(x, s2, t) ⇒ s1 = s2

Given its current location, the agent can infer properties of the square from properties of its current percept. For example, if the agent is at a square

and perceives a breeze, then that square is breezy:

$\forall$ s, t At(Agent, s, t) $\wedge$ Breeze(t) $\Rightarrow$ Breezy(s)

Having discovered which places are breezy, the agent can deduce where the pits are. Whereas propositional logic necessitates a separate axiom for each square and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:

$\forall$ s Breezy(s) $\Leftrightarrow$ $\exists$r Adjacent (r, s) $\wedge$ Pit(r)

Similarly, in first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step. For example, the axiom for the arrow becomes

$\forall$ t HaveArrow(t + 1) $\Leftrightarrow$ (HaveArrow(t) $\wedge\neg$Action(Shoot, t))

7.5 KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

The process of constructing a knowledge-base in first-order logic is called as knowledge- engineering. In knowledge-engineering, someone who investigates a particular domain, learns important concept of that domain, and generates a formal representation of the objects, is known as knowledge engineer.

7.5.1 Knowledge Engineering Process:

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. Identify the task:

The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents.

2. Assemble the relevant knowledge:

The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know by a process called knowledge acquisition. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works. For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. For real domains, the issue of relevance can be quite difficult.

3. Decide on a vocabulary of predicates, functions, and constants:

Translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering style. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the ontology of the domain. The word ontology means a particular theory of the nature of being or existence.

4. Encode general knowledge about the domain:

The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5. Encode a description of the specific problem instance:

If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

6. Pose queries to the inference procedure and get answers:

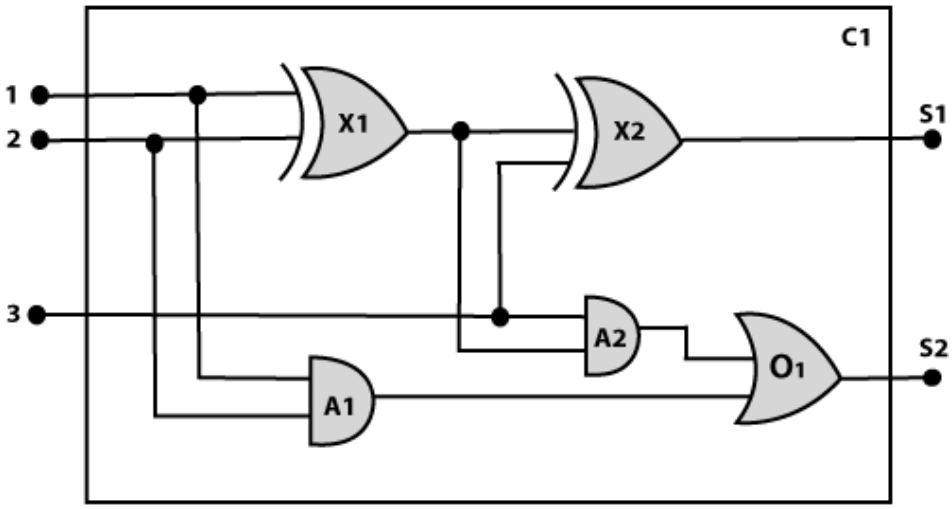
We can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.

7. Debug the knowledge base:

The answers to queries will be correct on the first try. More precisely, the answers will be correct for the knowledge base as written, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensure that no axioms are missing.

7.5.2 The Electronic Circuit Domain:

In this topic, we will understand the Knowledge engineering process in an electronic circuit domain. This approach is mainly suitable for creating special-purpose knowledge base. Following are some main steps of the knowledge-engineering process. Using these steps, we will develop a knowledge base which will allow us to reason about digital circuit (One-bit full adder) which is given below.



1. Identify the task:

The first step of the process is to identify the task, and for the digital circuit, there are various reasoning tasks. At the first level or highest level, we will examine the functionality of the circuit:

i. Does the circuit add properly?

ii. What will be the output of gate A2, if all the inputs are high?

At the second level, we will examine the circuit structure details such as:

i. Which gate is connected to the first input terminal?

ii. Does the circuit have feedback loops?

2. Assemble the relevant knowledge:

In the second step, we will assemble the relevant knowledge which is required for digital circuits. So, for digital circuits, we have the following required knowledge:

i. Logic circuits are made up of wires and gates.

ii. Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.

iii. In this logic circuit, there are four types of gates used: AND, OR, XOR, and NOT.

iv. All these gates have one output terminal and two input terminals (except NOT gate, it has one input terminal).

3. Decide on vocabulary:

The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates. Firstly, we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, Gate(X1). The functionality of each gate is determined by its type, which is taken as constants such as AND, OR, XOR, or NOT.

i. Circuits will be identified by a predicate: Circuit (C1).

ii. For the terminal, we will use predicate: Terminal(x).

iii. For gate input, we will use the function In(1, X1) for denoting the first input terminal of the gate, and for output terminal we will use Out (1, X1). The function Arity(c, i, j) is used to denote that circuit c has i input, j output.

iv. The connectivity between gates can be represented by predicate Connect(Out(1, X1), In(1, X1)).

v. We use a unary predicate On(t), which is true if the signal at a terminal is on.

4. Encode general knowledge about the domain:

To encode the general knowledge about the logic circuit, we need some following rules:

i. If two terminals are connected then they have the same input signal, it can be represented as

∀ t1, t2 Terminal (t1) ∧ Terminal (t2) ∧ Connect (t1, t2) → Signal (t1) = Signal (t2)

ii. Signal at every terminal will have either value 0 or 1, it will be represented as:

∀ t Terminal (t) →Signal (t) = 1 ∨Signal (t) = 0

iii. Connect predicates are commutative:

∀ t1, t2 Connect(t1, t2) → Connect (t2, t1)

iv. Representation of types of gates:

∀ g Gate(g) ∧ r = Type(g) → r = OR ∨r = AND ∨r = XOR ∨r = NOT

v. Output of AND gate will be zero if and only if any of its input is zero.

∀ g Gate(g) ∧ Type(g) = AND →Signal (Out(1, g))= 0 ⇔ ∃n Signal (In(n, g))= 0

vi. Output of OR gate is 1 if and only if any of its input is 1:

∀ g Gate(g) ∧ Type(g) = OR → Signal (Out(1, g))= 1 ⇔ ∃n Signal (In(n, g))= 1

vii. Output of XOR gate is 1 if and only if its inputs are different:

∀ g Gate(g) ∧ Type(g) = XOR → Signal (Out(1, g)) = 1 ⇔ Signal (In(1, g)) ≠ Signal(In(2, g))

viii. Output of NOT gate is invert of its input:

∀ g Gate(g) ∧ Type(g) = NOT → Signal (In(1, g)) ≠ Signal (Out(1, g))

ix. All the gates in the above circuit have two inputs and one output (except NOT gate).

∀ g Gate(g) ∧ Type(g) = NOT → Arity(g, 1, 1)

∀ g Gate(g) ∧ r =Type(g) ∧ (r= AND ∨r= OR ∨r= XOR) → Arity (g, 2, 1)

x. All gates are logic circuits:

∀ g Gate(g) → Circuit (g)

5. Encode a description of the problem instance:

Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought. This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.

For the given circuit C1, we can encode the problem instance in atomic sentences as below: Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

For XOR gate: Type(x1)= XOR, Type(X2) = XOR

For AND gate: Type(A1) = AND, Type(A2)= AND

For OR gate: Type (O1) = OR.

And then represent the connections between all the gates.

6. Pose queries to the inference procedure and get answers:

In this step, we will find all the possible set of values of all the terminal for the adder circuit. The first query will be:

What should be the combination of input which would generate the first output of circuit C1, as 0 and a second output to be 1?

∃ i1, i2, i3 Signal (In(1, C1))=i1 ∧ Signal (In(2, C1))=i2 ∧ Signal (In(3, C1))= i3 ∧ Signal (Out(1, C1)) =0 ∧ Signal (Out(2, C1))=1

7. Debug the knowledge base:

Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base. In the knowledge base, we may have omitted assertions like 1 ≠ 0.

7.6 SUMMARY

- First-order logic a representation language that is far more powerful than propositional logic.
- First-order logic makes use of objects and relations and thereby it gains more expressive power.
- The syntax of first-order logic is built upon propositional logic. It adds terms to represent objects, and has universal and existential quantifiers.
- A possible world, or model, for first-order logic includes a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations among objects, and function symbols to functions on objects.
- Use of First-Order logic is various domain such as kinship domain, number and sets and in Wumpus world.
- Developing a knowledge base in first-order logic requires a careful process of analyzing the domain, choosing a vocabulary, and encoding the axioms required to support the desired inferences.

7.7 UNIT END QUESTIONS

1. Why is First-Order Logic used over Propositional Logic?
2. What is First-Order Logic? Discuss the different elements used in first order logic.
3. Explain Universal and Existential quantifier with example.
4. Define following terms.

   a. Predicate

   b. Function (in logic)

   c. Model in First-order Logic

   d. First-order Logic
5. Explain Knowledge Engineering process in detail.
6. Translate the following sentences into first-order logic:

   a. Understanding leads to friendship.

   b. Friendship is transitive.

Define all predicates, functions, and constants you use.

7. Write the following assertions in first-order logic:

   a. Emily is either a surgeon or a lawyer.

b. Joe is an actor

c. All surgeons are doctors.

d. Joe does not have a lawyer (i.e., is not a customer of any lawyer).

e. Emily has a boss who is a lawyer.

f. Every surgeon has a lawyer.

7.8 LIST OF REFERENCES

1. A First Course in Artificial Intelligence, First Edition, Deepak Khemani, Tata McGraw Hill Publisher
2. Artificial Intelligence: A Modern Approach, Third Edition, Stuart Russel and Peter Norvig, Pearson Publisher

7.9 BIBLIOGRAPHY

1. Ayorinde, I. and Akinkunmi, B. (2013). Application of First-Order Logic in Knowledge Based Systems. African Journal of Computing & ICT
2. https://www.javatpoint.com/ai-knowledge-engineering-in-first-order-logic

*****

8

INFERENCE IN FIRST-ORDER LOGIC

Unit Structure

8.0 OBJECTIVES

After going through this chapter, you will learn:

- Inference rule for quantifiers.
- First-Order logic Inference rule.

- Unification for finding substitutions that make different logical expressions look identical.
- Inference using forward chaining & backward chaining.
- Translation of sentences into Conjunctive Normal Form (CNF) for first-order logic.
- Resolution inference rule using which two clauses that share no variables, can be resolved if they contain complementary literals.
- Resolution strategies that help find proofs efficiently.

8.1 INTRODUCTION

Inference in First-Order Logic is used to derive new facts or sentences from existing sentences. In inference we define effective procedures for answering questions posed in first- order logic. In propositional logic we studied how inference can be achieved for propositional logic. In this chapter, we extend those results to obtain algorithms that can answer any answerable question stated in first-order logic. We will introduce inference rules for quantifiers and show how to reduce first-order inference to propositional inference. Then we introduce the idea of unification, showing how unification can be used to construct inference rules that work with first-order sentences. Forward chaining, backward chaining and logic programming systems are also covered in this chapter. Forward and backward chaining can be very efficient, but those are applicable only to knowledge bases that can be expressed as sets of Horn clauses. General first-order sentences require resolution-based theorem proving, which is also discussed in this chapter.

8.2 PROPOSITIONAL Vs. FIRST-ORDER INFERENCE

There are some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead to the idea that first-order inference can be done by converting the knowledge base to propositional logic and then using propositional inference.

8.2.1 Inference Rules for Quantifiers:

As propositional logic we also have inference rules in first-order logic. Following are some basic inference rules in First-Order logic:

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

8.2.1.1 Universal Instantiation:

- Universal instantiation is also called as universal elimination or UI is a valid inference rule. It can be applied multiple times to add new sentences.

- The new knowledge base is logically equivalent to the previous knowledge base.
- As per universal instantiation, we can infer any sentence obtained by substituting a ground term (a term without variables) for the variable.
- The universal instantiation rule state that we can infer any sentence P(c) by substituting a ground term c (a constant within domain x) from ∀ x P(x) for any object in the universe of discourse.
- It can be represented as $\frac{\forall x\, P(x)}{P(c)}$

Example:1

IF "Every person like ice-cream"=> ∀x P(x). So we can infer that

"John likes ice-cream" => P(c)

Example: 2

Let's take a famous example,

"All kings who are greedy are Evil." So let our knowledge base contain this detail as in the form of First-Order Logic:

∀x king(x) ∧ greedy (x) → Evil (x),

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

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

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

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

Here, the substitutions are {x/John }, {x/Richard }, and {x/Father (John)}.

8.2.1.2 Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.
- It can be applied only once to replace the existential sentence.
- The new knowledge base is not logically equivalent to old knowledge base, but it will be satisfiable if old knowledge base was satisfiable.
- This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c.
- The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.
- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Example:

From the given sentence:

∃x Crown(x) ∧ OnHead(x, John),

We can infer:

Crown(K) ∧ OnHead(K, John), as long as K does not appear in the knowledge base.

The above used K is a constant symbol, which is called Skolem constant. The Existential instantiation is a special case of Skolemization process.

8.2.1.3 Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as $\forall x\, P(x)$.
- It can be represented as: $\frac{P(c)}{\forall x P(x)}$
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.

Example:

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

8.2.1.4 Existential Generalization:

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as: $\frac{P(c)}{\exists x P(x)}$

Example:

Let's say that,

"Priyanka got good marks in English."

"Therefore, someone got good marks in English."

8.2.2 Reduction to Propositional Inference:

Once we have defined rules for inferring non-quantified sentences from quantified sentences, it is possible to reduce first-order inference to propositional inference.

The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of all possible instantiations. For example, suppose our knowledge base contains just the sentences

$\forall$ x King (x) $\wedge$ Greedy (x) $\Rightarrow$ 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) $\wedge$ Greedy (John) $\Rightarrow$ Evil (John)

King (Richard) $\wedge$ Greedy (Richard) $\Rightarrow$ Evil (Richard)

We discarded the universally quantified sentence. Now, the knowledge base is propositionalized. We have King (John), Greedy (John), Evil(John), King(Richard) as proposition symbols. Now, we can apply any of the propositional algorithms to obtain conclusions such as Evil(John).

Every First-Order Logic knowledge base (KB) can be propositionalized so as to preserve entailment. A ground sentence is entailed by new KB if it is entailed by the original KB.

Idea of the inference is that first propositionalize knowledge base (KB) and query. After that apply resolution and then return result.

Problems with propositionalization:

Propositionalization generates lots of irrelevant sentences.

Example, from:

$\forall$x King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)

King(John)

King(John)

$\forall$y Greedy(y)

Brother(Richard,John)

It seems obvious that Evil(John), but propositionalization produces lots of facts such as Greedy(Richard) that are irrelevant.

8.3 UNIFICATION AND LIFTING

The inference of Evil(John) from the below sentences seems completely obvious to a human being. We now show how to make it completely obvious to a computer using inference rules.

∀ x King(x) ∧ Greedy(x) ⇒ Evil(x)

King(John)

Greedy(John)

8.3.1 A First-Order Inference Rule- Generalized Modus Ponens Rule:

For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens. It raises Modus Ponens from ground (variable-free) propositional logic to first-order logic. The key advantage of lifted inference rules over propositionalization is that they make only those substitutions that are required to allow particular inferences to proceed.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences pi, pi', q. Where there is a substitution θ such that SUBST (θ, pi',) = SUBST(θ, pi), it can be represented as:

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

Example:

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

Here let say,

p1' is king(John) p1 is king(x)

p2' is Greedy(y) p2 is Greedy(x)

θ is {x/John, y/John} q is Evil(x)

SUBST(θ, q) is Evil(John) .

8.3.2 Unification:

Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process. It takes two literals as input and makes them identical using substitution.

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 algorithm takes two sentences and returns a unifier for them if one exists:

UNIFY(p, q)= θ where SUBST(θ, p)= SUBST(θ, q) .

Let us look at some examples of how UNIFY should behave. Suppose we have a query

AskVars (Knows (John , x)): whom does John know?

Answers to this query 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} .

There is one more complication: we said that 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. It turns out that, for every unifiable pair of expressions, there is a single most general unifier (or MGU) that is unique up to renaming and substitution of variables. (For example, {x/John} and {y/John} are considered equivalent, as are {x/John, y/John} and {x/John, y/x}.) In this case it is {y/John, x/z}.

8.3.3 Storage and Retrieval:

The TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some sentence in the knowledge base. The problem we used to illustrate unification for finding all facts that unify with Knows(John, x) is an example of FETCH.

The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works. We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have some chance of unifying. For example, there is no point in trying to unify Knows(John, x) with Brother (Richard, John). We can avoid such unifications by indexing the facts in the knowledge base. A simple scheme called predicate indexing puts all the Knows facts in one bucket and all the Brother facts in another. The buckets can be stored in a hash table for efficient access. Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol.

Sometimes, however, a predicate has many clauses. For example, suppose that the tax authorities want to keep track of who employs and we use a predicate Employs(x, y). This would be a very large bucket with millions of employers and tens of millions of employees. Answering a query such as Employs(x,Richard) with predicate indexing would require scanning the entire bucket. For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key. Then we could simply construct the key from the query and retrieve exactly those facts that unify with the query.

For other queries, such as Employs(IBM , y), we would need to have indexed the facts by combining the predicate with the first argument. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with.

Given a sentence to be stored, it is possible to construct indices for all possible queries that unify with it. For the fact Employs(IBM ,Richard), the queries are

Employs(IBM ,Richard) Does IBM employ Richard?

Employs(x,Richard) Who employs Richard?

Employs(IBM , y) Whom does IBM employ?

Employs(x, y) Who employs whom?

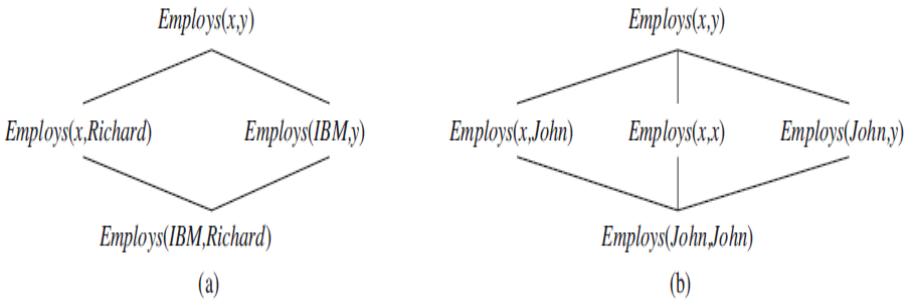


Figure (a) The subsumption lattice whose lowest node is Employs(IBM, Richard).

Figure(b) The subsumption lattice for the sentence Employs (John, John).

These queries form a subsumption lattice, as shown in Figure(a). The lattice has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution and the "highest" common descendant of any two nodes is the result of applying their most general unifier. The portion of the lattice above any ground fact can be constructed systematically. A sentence with repeated constants has a slightly different lattice, as shown in Figure(b).

The scheme we have described works very well whenever the lattice contains a small number of nodes. If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices. We can respond by adopting a fixed policy, such as maintaining indices only on keys composed of a predicate plus each argument, or by using an adaptive policy that creates indices to meet the demands of the kinds of queries being asked.

8.4 FORWARD AND BACKWARD CHAINING

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:

1. Forward chaining
2. 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 knowledge base in the form of the first-order definite clause.

- **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. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose conclusion is a single positive literal. The following are first-order definite clauses:

$King(x) \wedge Greedy(x) \Rightarrow Evil(x)$

$King(John)$

$Greedy(y)$

- **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 $p \wedge q \rightarrow k$.

8.4.1 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 bottom-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 First-Order Logic:

1. 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)

2. 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)

3. All of the missiles were sold to country A by Robert.

   Missiles(p) ∧ Owns (A, p) ⇒ Sells (Robert, p, A) .....(4)

4. Missiles are weapons.

   Missile(p) ⇒ Weapons (p) .....(5)

5. Enemy of America is known as hostile.

   Enemy(p, America) ⇒Hostile(p) .....(6)

6. Country A is an enemy of America.

   Enemy (A, America) .....(7)

7. Robert is American.

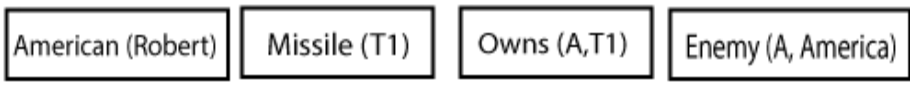
American(Robert) .....(8)

This knowledge base contains no function symbols and therefore it is an instance of the class Datalog knowledge bases. Datalog is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements which are typically made in relational databases.

Forward chaining proof:

Step-1:

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



Step-2:

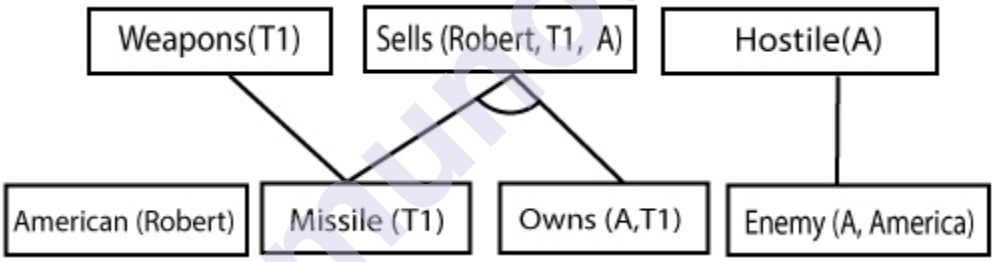
At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

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

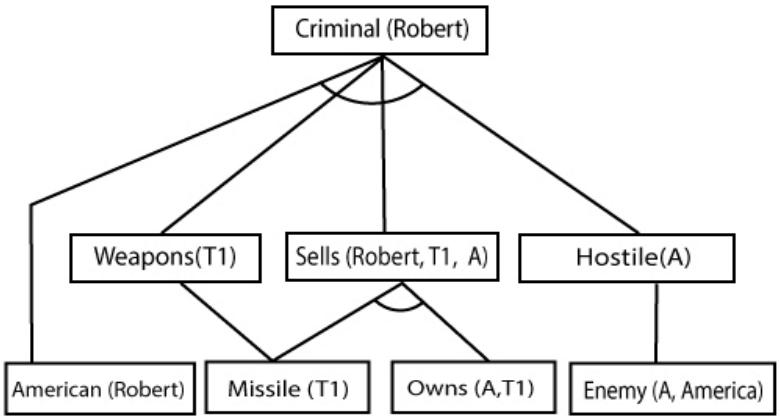
Rule-(4) satisfy with the substitution {p/T1}, so Sells (Robert, T1, A) is added, which infers from the conjunction of Rule (2) and (3).

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



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.



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

8.4.2 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. In this section we will study backward chaining example and then we describe how it is used in logic programming, which is the most widely used form of automated reasoning.

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.

8.4.2.1 Backward Chaining Example:

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

1. 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)

2. 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)

3. All of the missiles were sold to country A by Robert.

   Missiles(p) ∧ Owns (A, p) ⇒ Sells (Robert, p, A) .....(4)

4. Missiles are weapons.

   Missile(p) ⇒ Weapons (p) .....(5)

5. Enemy of America is known as hostile.

   Enemy(p, America) ⇒Hostile(p) .....(6)

6. Country A is an enemy of America.

   Enemy (A, America) .....(7)

7. Robert is American.

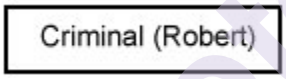
   American(Robert) .....(8)

Backward-Chaining proof:

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

Step-1:

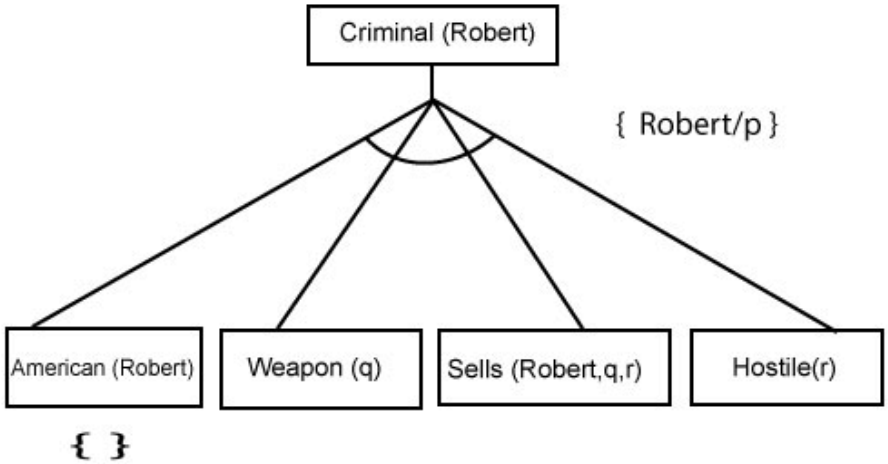
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So, our goal fact is "Robert is Criminal," so following is the predicate of it.



Step-2:

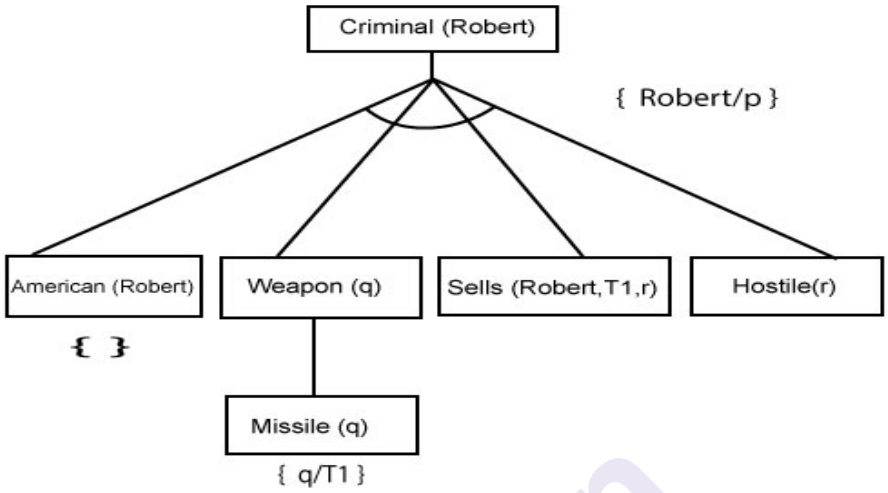
At the second step, we will infer other facts form goal fact which satisfies the rules. As we can see in Rule-1, the goal predicate Criminal(Robert) is present with substitution {Robert/P}. So, we will add all the conjunctive facts below the first level and will replace p with Robert.

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



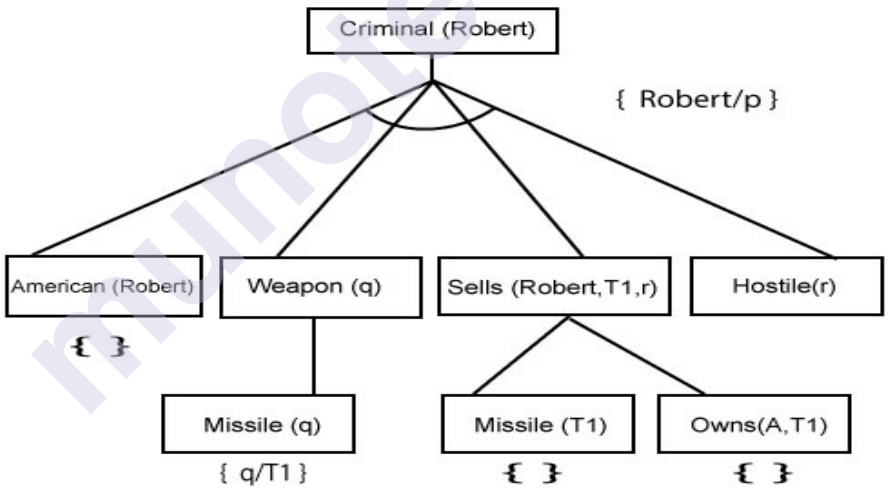
Step-3:

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.



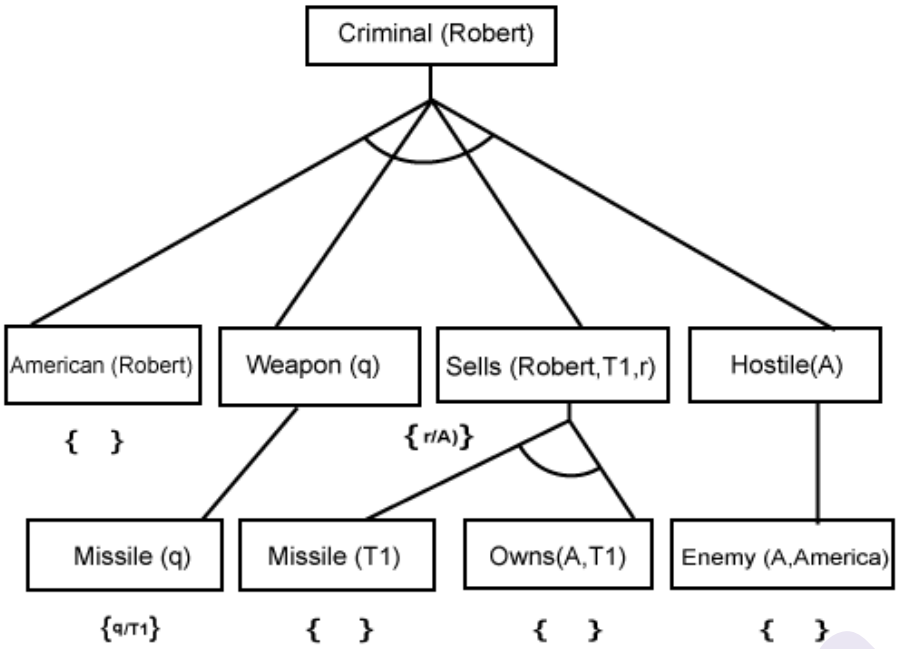
Step-4:

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.



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.



8.4.2.2 Logic Programming:

In Logic programming the system is constructed by expressing knowledge in a formal language and that problem is solved by running inference processes on that knowledge. Prolog is the most widely used logic programming language. It is used primarily as a Rapid prototyping language and for symbol-manipulation tasks such as writing compilers and parsing natural language. Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation which is different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for Constants which is the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written "backwards"; instead of

$A \wedge B \Rightarrow C$

In Prolog we have

C :- A, B.

Here is a typical example:

criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).

The notation [E|L] denotes a list whose first element is E and whose rest is L. Here is a Prolog program for append(X,Y,Z), which succeeds if list Z is the result of appending lists X and Y:

append([],Y,Y).

append([A|X],Y,[A|Z]) :- append(X,Y,Z).

In English, we can read these clauses as

1. appending an empty list with a list Y produces the same list Y and

2. [A|Z] is the result of appending [A|X] onto Y, provided that Z is the result of appending X onto Y.

We can ask the query

append(X,Y,[1,2]): what two lists can be appended to give [1,2]?

We get the solutions

X=[] Y=[1,2];

X=[1] Y=[2];

X=[1,2] Y=[]

The execution of Prolog programs is done through depth-first backward chaining. Some aspects of Prolog fall outside standard logical inference:

- Prolog uses the database semantics rather than first-order semantics, and this is apparent in its treatment of equality and negation.
- There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved" by executing code rather than doing further inference. For example, the goal "X is 4+3" succeeds where X bound to 7. On the other hand, the goal "5 is X+Y" fails, because the built-in functions do not do arbitrary equation solving.
- There are built-in predicates that have side effects when executed. These include input-output predicates and the assert/retract predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones.
- Dynamic programming is the one in which the solutions to subproblems are constructed incrementally from smaller subproblems and are cached to avoid re-computation. We can obtain a similar effect in a backward chaining system using memoization that is, caching solutions to subgoals as they are found and then reusing those solutions when the subgoal recurs, rather than repeating the previous computation.
- This is the approach taken by tabled logic programming systems, which use efficient storage and retrieval mechanisms to perform memoization. Tabled logic programming combines the goal-directedness of backward chaining with the dynamic programming.
- Prolog uses database semantics. There is no way to assert that a sentence is false in Prolog. This makes Prolog less expressive than first-order logic, but it is part of what makes Prolog more efficient and

more concise. If given problem can be described with database semantics, it is more efficient to reason with Prolog or some other database semantics system, rather than translating into FOL and reasoning with a full FOL theorem prover.

8.4.3 Forward Chaining Vs. Backward Chaining:

Sr. No	Forward Chaining	Backward Chaining
1	Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal.	Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
2	It is a bottom-up approach	It is a top-down approach
3	Forward chaining is known as data-driven inference technique as we reach to the goal using the available data.	Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
4	Forward chaining reasoning applies a breadth-first search strategy.	Backward chaining reasoning applies a depth-first search strategy.
5	Forward chaining tests for all the available rules	Backward chaining only tests for few required rules.
6	Forward chaining is suitable for the planning, monitoring, control, and interpretation application.	Backward chaining is suitable for diagnostic, prescription, and debugging application.
7	Forward chaining can generate an infinite number of possible conclusions.	Backward chaining generates a finite number of possible conclusions.
8	It operates in the forward direction.	It operates in the backward direction.
9	Forward chaining is aimed for any conclusion.	Backward chaining is only aimed for the required data.

8.5 RESOLUTION

Resolution is a valid inference rule producing a new clause implied by two clauses containing complementary literals. A literal is an atomic symbol or its negation, i.e., P, ~P. 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. A Knowledge Base is actually a set of sentences all of which are true, i.e.,

a conjunction of sentences. To use resolution, translate knowledge base into conjunctive normal form (CNF), where each sentence is written as a disjunction of (one or more) literals.

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. Propositional resolution using refutation is a complete inference procedure for propositional logic. Here, we describe how to extend resolution to first-order logic. Resolution is a single inference rule which can efficiently operate on the conjunctive normal form or clausal form.

8.5.1 Conjunctive Normal Form for First-Order Logic:

First-order resolution requires that sentences be in conjunctive normal form (CNF) that is, a conjunction of clauses, where each clause is a disjunction of literals.

Clause: Disjunction of literals is called a clause. It is also known as a unit clause.

Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$\forall$ x American(x) $\wedge$Weapon(y) $\wedge$ Sells(x, y, z) $\wedge$ Hostile(z) $\Rightarrow$ Criminal (x)

in CNF form it can be written as

$\neg$American(x) $\vee$ $\neg$Weapon(y) $\vee$ $\neg$Sells(x, y, z) $\vee$ $\neg$Hostile(z) $\vee$ Criminal (x)

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.

The procedure for conversion to CNF is similar to the propositional case. The principal difference arises from the need to eliminate existential quantifiers. We illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

$\forall$ x [$\forall$ y Animal(y) $\Rightarrow$ Loves(x, y)] $\Rightarrow$ [$\exists$ y Loves(y, x)]

The steps for translating the sentences in CNF are as follows:

1. Eliminate implications:

$\forall$ x [$\neg\forall$ y $\neg$Animal(y) $\vee$ Loves(x, y)] $\vee$ [$\exists$ y Loves(y, x)]

2. Move $\neg$ inwards:

In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$\neg\forall$x p becomes $\exists$ x $\neg$p

$\neg\exists$x p becomes $\forall$ x $\neg$p

Our sentence will go through the following transformations:

∀ x [∃ y ¬(¬Animal(y) ∨ Loves(x, y))] ∨ [∃ y Loves(y, x)]

∀ x [∃ y ¬¬Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ y Loves(y, x)]

∀ x [∃ y Animal (y) ∧¬Loves(x, y)] ∨ [∃ y Loves(y, x)]

Notice how a universal quantifier (∀ y) in the premise of the implication has become an existential quantifier. The sentence will now be read as "Either there is some animal that x doesn't love, or someone loves x." The meaning of the original sentence has been preserved.

3. Standardize variables:

For sentences like (∃xP(x))∨(∃xQ(x)) which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

∀ x [∃ y Animal (y) ∧¬Loves(x, y)] ∨ [∃ z Loves(z, x)]

4. Skolemize:

Skolemization is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule: translate ∃x P(x) into P(A), where A is a new constant. However, we can't apply Existential Instantiation to our sentence above because it doesn't match the pattern ∃x P(x); only parts of the sentence match the pattern.

Thus, we want the Skolem entities to depend on x and z:

∀ x [Animal (F(x)) ∧¬Loves(x, F(x))] ∨ Loves(G(z), x)

Here F and G are Skolem functions. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears.

5. Drop universal quantifiers:

At this point, all remaining variables must be universally quantified and the previous sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can now drop the universal quantifiers:

[Animal (F(x)) ∧ ¬Loves(x, F(x))] ∨ Loves(G(z), x)

6. Distribute ∨ over ∧:

[Animal (F(x)) ∨ Loves(G(z), x)] ∧ [¬Loves(x, F(x)) ∨ Loves(G(z), x)]

This step may require flattening out nested conjunctions and disjunctions.

The sentence is now in CNF and consists of two clauses.

8.5.2 The Resolution Inference Rule:

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule. Resolution can resolve two clauses if they contain complementary literals, which are assumed to be standardized apart so that they share no variables.

Propositional literals are complementary if one is the negation of the other. First-order literals are complementary if one unifies with the negation of the other. Thus, we have

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

Where, UNIFY(l_i,¬ m_j)=θ.

For example, we can resolve the two clauses

[Animal (F(x)) ∨ Loves(G(x), x)] and [¬Loves(u, v) ∨ ¬Kills(u, v)]

by eliminating the complementary literals Loves(G(x), x) and ¬Loves(u, v), with unifier

θ={u/G(x), v/x}, to produce the resolvent clause

[Animal (F(x)) ∨ ¬Kills(G(x), x)]

This rule is called the binary resolution rule because it resolves exactly two literals.

Steps for Resolution:

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

8.5.3 Example Proof:

We will consider an example in which we will apply resolution.

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 First-Order Logic

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)

Step-2: Conversion of First-Order logic into CNF

In First order logic resolution, it is required to convert the First-Order logic into CNF as CNF form makes easier for resolution proofs.

i. Eliminate all implication (→) and rewrite

a. ∀x ¬ food(x) V likes(John, x)

b. food(Apple) Λ food(vegetables)

c. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)

d. eats (Anil, Peanuts) Λ alive(Anil)

e. ∀x ¬ eats(Anil, x) V eats(Harry, x)

f. ∀x¬ [¬ killed(x)] V alive(x)

g. ∀x ¬ alive(x) V ¬ killed(x)

h. likes(John, Peanuts)

ii. Move negation(¬) inwards and rewrite

a. ∀x ¬ food(x) V likes(John, x)

b. food(Apple) Λ food(vegetables)

c. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)

d. eats (Anil, Peanuts) Λ alive(Anil)

e. ∀x ¬ eats(Anil, x) V eats(Harry, x)

f. ∀x ¬killed(x)] V alive(x)

g. ∀x ¬ alive(x) V ¬ killed(x)

h. likes(John, Peanuts)

iii. Rename variables or standardize variables

a. ∀x ¬ food(x) V likes(John, x)

b. food(Apple) Λ food(vegetables)

c. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)

d. eats (Anil, Peanuts) Λ alive(Anil)

e. ∀w¬ eats(Anil, w) V eats(Harry, w)

f. ∀g ¬killed(g)] V alive(g)

g. ∀k ¬ alive(k) V ¬ killed(k)

h. likes(John, Peanuts)

iv. 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.

v. 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.

a. ¬ food(x) V likes(John, x)

b. food(Apple)

c. food(vegetables)

d. ¬ eats(y, z) V killed(y) V food(z)

e. eats (Anil, Peanuts)

f. alive(Anil)

g. ¬ eats(Anil, w) V eats(Harry, w)

h. killed(g) V alive(g)

i. ¬ alive(k) V ¬ killed(k)

j. likes(John, Peanuts)

vi. Distribute conjunction Λ over disjunction V:

This step will not make any change in this problem.

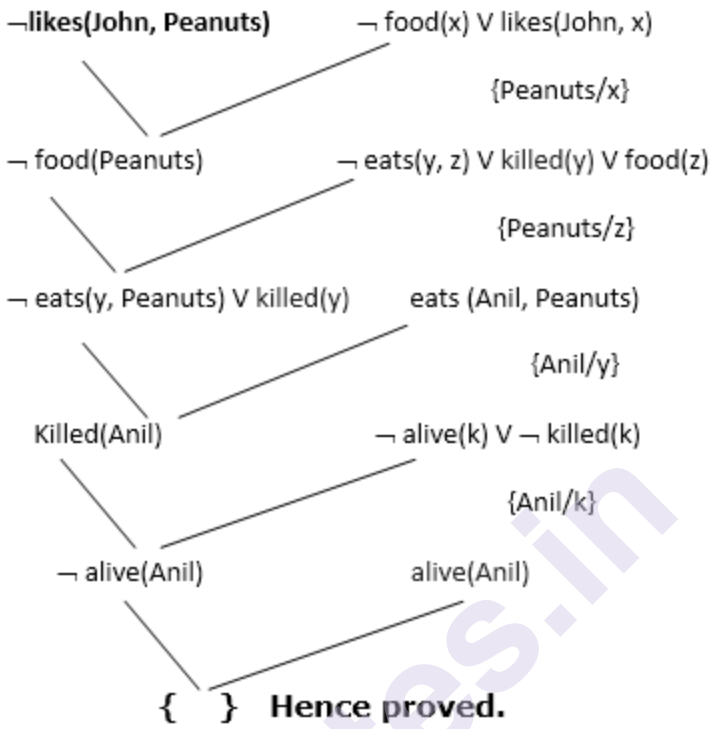
Step-3: Negate the statement to be proved:

In this step, we will apply negation to the conclusion statement, 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:



Hence, the negation of the conclusion has been proved as a complete contradiction with the given set of statements.

Explanation of Resolution graph:

a. In the first step of resolution graph, ¬likes(John, Peanuts) , and likes(John, x) get resolved(cancelled) by substitution of {Peanuts/x}, and we are left with ¬ food(Peanuts).

b. In the second step of the resolution graph, ¬ food(Peanuts) , and food(z) get resolved (cancelled) by substitution of { Peanuts/z}, and we are left with ¬ eats(y, Peanuts) V killed(y).

c. 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) .

d. 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).

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

8.5.4 Equality:

None of the inference methods described so far handle an assertion of the form x = y. Three distinct approaches can be taken. The first approach is to write down sentences about the equality relation in the knowledge base. Equality is reflexive, symmetric, and transitive and we can substitute equals for equals in any predicate or function. So, we need three basic axioms, and then one for each predicate and function:

$\forall x\ x=x$

$\forall\ x, y\ x=y \Rightarrow y=x$

$\forall\ x, y, z\ x=y \wedge y=z \Rightarrow x=z$

$\forall\ x, y\ x=y \Rightarrow (P1(x) \Leftrightarrow P1(y))$

$\forall\ x, y\ x=y \Rightarrow (P2(x) \Leftrightarrow P2(y))$

...

$\forall\ w, x, y, z\ w=y \wedge x=z \Rightarrow (F1(w, x)= F1(y, z))$

$\forall\ w, x, y, z\ w=y \wedge x=z \Rightarrow (F2(w, x)= F2(y, z))$

...

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations. However, these axioms will generate a lot of conclusions, most of them will not be helpful to a proof. So there has been a search for more efficient ways of handling equality. One alternative is to add inference rules rather than axioms.

The simplest rule, demodulation, takes a unit clause x=y and some clause α that contains the term x, and yields a new clause formed by substituting y for x within α. It works if the term within α unifies with x; it need not be exactly equal to x.

Note that demodulation is directional; given x = y, the x always gets replaced with y, never vice versa.

Example:

Given,

Father (Father (x)) = PaternalGrandfather (x)

Birthdate(Father (Father (Bella)), 1926)

we can conclude by demodulation

Birthdate(PaternalGrandfather (Bella), 1926) .

1. Demodulation: For any terms x, y, and z, w here z appears

somewhere in literal m_i and where UNIFY(x, z) = θ,

$$\frac{x = y, m_1 \vee \ldots \vee m_n}{SUB(SUBST(\theta,x), SUBST(\theta,y), m_1 \vee \ldots \vee m_n)}$$

Where, SUBST is the usual substitution of a binding list, and SUB(x, y,m) means to replace x with y everywhere that x occurs within m.

The rule can also be extended to handle non-unit clauses in which an equality literal appears.

3. **Paramodulation:** For any terms x, y, and z, where z appears somewhere in literal m_i ,

and where UNIFY(x, z) = θ,

$$\frac{l_1 \vee \ldots \vee l_k \vee x = y, \qquad m_1 \vee \ldots \vee m_n}{SUB(SUBST(\theta,x), SUBST(\theta,y), SUBST(\theta, l_1 \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_n))}$$

For example, from

P(F(x,B), x) ∨ Q(x) and F(A, y)= y ∨ R(y)

we have θ =UNIFY(F(A, y), F(x,B))= {x/A, y/B}, and we can conclude by paramodulation the sentence

P(B,A) ∨ Q(A) ∨ R(B)

Paramodulation yields a complete inference procedure for first-order logic with equality.

3. Equational unification:

A third approach handles equality reasoning entirely within an extended unification algorithm. That is, terms are unifiable if they are provably equal under some substitution, where "provably" allows for equality reasoning. For example, the terms 1 + 2 and 2 + 1 normally are not unifiable, but a unification algorithm that knows that x + y=y + x could unify them with the empty substitution. Equational unification of this kind can be done with efficient algorithms designed for the particular axioms using commutativity, associativity, and so on rather than through explicit inference with those axioms.

8.5.5 Resolution Strategies:

We know that repeated applications of the resolution inference rule will eventually find a proof if one exists. In this subsection, we examine strategies that help find proofs efficiently.

1. Unit preference:

This strategy prefers to do resolutions where one of the sentences is a

single literal (also known as a unit clause). The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses. Resolving a unit sentence (such as P) with any other sentence (such as ¬P ∨¬Q∨R) always yields a clause (in this case, ¬Q ∨ R) that is shorter than the other clause. This strategy leads to a dramatic speedup, making it feasible to prove theorems that could not be handled without the preference. Unit resolution is a restricted form of resolution in which every resolution step must involve a unit clause. Unit resolution is incomplete in general, but complete for Horn clauses.

2. Set of support:

Preferences that try certain resolutions first are helpful, but in general it is more effective to try to eliminate some potential resolutions altogether. For example, we can insist that every resolution step involve at least one element of a special set of clauses called the set of support. The resolvent is then added into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced. The set-of-support strategy has the additional advantage of generating goal-directed proof trees that are often easy for humans to understand.

3. Input resolution:

In this strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The space of proof trees of this shape is smaller than the space of all proof graphs. Linear resolution is complete.

4. Subsumption:

The subsumption method eliminates all sentences that are subsumed by (that is, more specific than) an existing sentence in the KB. For example, if P(x) is in the KB, then there is no sense in adding P(A) and even less sense in adding P(A) ∨ Q(B). Subsumption helps keep the KB small and thus helps keep the search space small.

8.5.6 Practical Uses Of Resolution Theorem Provers:

1. Theorem provers can be applied to the problems involved in the synthesis and verification of both hardware and software. Thus, theorem-proving research is carried out in the fields of hardware design, programming languages, and software engineering not just in AI.

2. In the case of hardware, the axioms describe the interactions between signals and circuit elements. Logical reasoners designed specially for verification have been able to verify entire CPUs, including their timing properties.

3. The AURA theorem prover has been applied to design circuits that are more compact than any previous design.

4. In the case of software, reasoning about programs is quite similar to reasoning about actions, axioms describe the preconditions and effects of each statement.

5. Similar techniques are now being applied to software verification by systems such as the SPIN model checker. For example, the Remote Agent spacecraft control program was verified before and after flight.

6. The RSA public key encryption algorithm and the Boyer–Moore string-matching algorithm have been verified this way.

8.6 SUMMARY

- In this chapter we analyzed logical inference in first-order logic.
- Inference rules (universal instantiation and existential instantiation) can be used to propositionalize the inference problem.
- The use of unification to identify appropriate substitutions for variables eliminates the instantiation step in first-order proofs, making the process more efficient.
- A lifted version of Modus Ponens uses unification to provide a natural and powerful inference rule, generalized Modus Ponens.
- The forward-chaining and backward chaining algorithms can be used for inference.
- The generalized resolution inference rule provides a complete proof system for first-order logic, using knowledge bases in conjunctive normal form.
- Several strategies exist for reducing the search space of a resolution system without compromising completeness. One of the most important issues is dealing with equality; we studied how demodulation and paramodulation can be used.
- Efficient resolution-based theorem provers have been designed to prove interesting mathematical theorems and to verify and synthesize software and hardware.

8.7 UNIT END QUESTIONS

1. What is Unification? Explain in brief about Unification.
2. Explain Conjunctive Normal Form (CNF) in First-Order logic.
3. Give the outline of simple forward chaining algorithm.
4. Explain in detail backward chaining algorithm.
5. Give comparison between forward chaining and backward chaining.
6. What is Resolution? Explain resolution steps with example.

7. Explain various resolution strategies in detail.

8. From "Horses are animals," it follows that "The head of a horse is the head of an animal." Demonstrate that this inference is valid by carrying out the following steps:

a. Translate the premise and the conclusion into the language of first-order logic. Use three predicates: HeadOf (h, x) (meaning "h is the head of x"), Horse(x), and Animal (x).

b. Negate the conclusion, and convert the premise and the negated conclusion into conjunctive normal form.

c. Use resolution to show that the conclusion follows from the premise.

8.8 LIST OF REFERENCES

1. A First Course in Artificial Intelligence, First Edition, Deepak Khemani, Tata McGraw Hill Publisher

2. Artificial Intelligence: A Modern Approach, Third Edition, Stuart Russel and Peter Norvig, Pearson Publisher

8.9 BIBLIOGRAPHY

- Ayorinde, I. and Akinkunmi, B. (2013). Application of First-Order Logic in Knowledge Based Systems. African Journal of Computing & ICT
- https://www.javatpoint.com/ai-resolution-in-first-order-logic
- https://www.javatpoint.com/forward-chaining-and-backward-chaining-in-ai

*****

UNIT V

9

PLANNING

Unit Structure

9.0 DEFINITION OF CLASSICAL PLANNING

1. How to reach goal from initial state is nothing but planning.
2. In order to achieve goal a sequence of actions is needed such kind of behaviour is called as planning.
3. During the execution after planning the agent will need to do prediction of the future so that it can move accordingly.
4. In simple words we can call planning as decision making system that tries to achieve its goal.
5. The agent can be any machine which is intelligent in nature.

9.0.1 Planning assumptions:

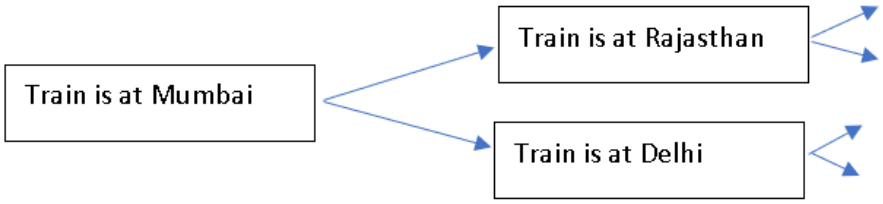
GOAL, observations, actions, deterministic nature, events etc.

9.1 ALGORITHMS FOR PLANNING AS STATE SPACE SEARCH

1. Information is needed to predict something; all these information is given by state.
2. This information is used to apply actions and decision can be made if the state is a goal state.
3. There are 3 state space for a problem:
   a. Initial state: The first state from which the action is applied and begins
   b. Goal state: Here the objective is satisfied
   c. Solution: Target is achieved here
4. There are 2 types of state space search.
   a. Forward state space
   b. Backward state space.

9.1.1 Forward State Space:

1. In this type of planning the agent will start from initial state and will go till the final i.e., target state.
2. The aim is to find the final path or full path which will give the target.
3. In the below example if you will notice then according to the forward state space, the initial state is Train is at Mumbai, now when the train moves it is getting two states after the action, one is train is in Rajasthan and another state is the train is at Delhi.
4. Imagine the goal is to reach Delhi then we can say that the target reached from initial till goal with the help of Forward state space.

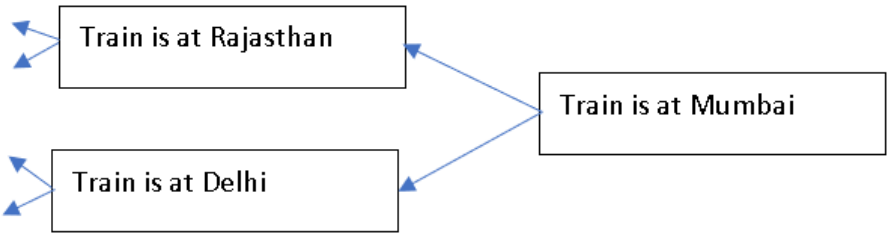


9.1.2 Backward State Space:

1. Backward state space planning is opposite to forward state space planning.

2. In this state space planning the state will start from the goal state the it will go towards the initial state.

3. In the below example if the target is Delhi, then the first state from which it will start is Train is at Delhi then it will move towards the initial state i.e., Train is at Mumbai.



9.2 PLANNING GRAPHS

1. This a technique in which plan is represented graphically to make work easy and to get the better an accurate planning.

2. All planning methods that is done can be rechecked or can be converted into planning graphs and checked.

3. There are many algorithms available which can be utilized to create this plan graph. One of the algorithms is called as GRAPHPLAN algorithm which is used in creation of notion and actions of the graph.

4. There is also parallelism which is supported by plan graph it is used to represent many states and how the particular state runs independently is represented

5. In the below graph you will come across 3 categories

   a. Action level nodes: The actions that can be executed in that particular period

   b. State level nodes: The state that can be true in that particular period

   c. Edge: The pre-condition which is applied in it and the effect

6. In the below example S_0 is the initial state

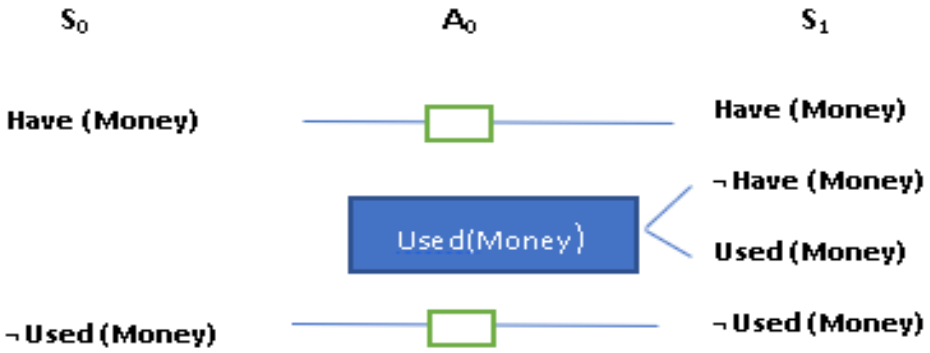
7. Every state will have its associated actions that can be true in that particular period.

8. If any conflict occurs anywhere then that can be represented using mutual exclusion links.

9. The level A_0 contains all the actions that can happed after the state S_0

10. The small box that is present above the edges represents facts.

11. Fact is nothing but it says that the literals that is getting used will not be modified.



9.2.1 Algorithm to build a plan graph:

1. Start at S0
2. Initialize the variable
3. Find the all-possible actions (A)that can be applied in that particular state, by incrementing it A++
4. Check if there is any mutex present anywhere.
5. Move ahead with next State by incrementing it S++
6. If mutex is present then compute that as well
7. If solution found then return success and exit.
8. If no solution found or possible then return failure

9.3 OTHER CLASSICAL PLANNING APPROACHES

1. Classical planning as Boolean satisfiability

- Boolean satisfiability is also called as propositional satisfiability problem
- It will check and let you know whether the plan is satisfying or not satisfying.
- As the name is Boolean it uses two values i.e. true or false so by using this approach if it returns true then we can say that the plan is satisfying otherwise not.
- It is also called as SAT.

2. Planning as constraint satisfaction

- In this mathematical query are applied for a set of objects and its state should agree with the constraints.
- Constraint Satisfaction Problem (CSP)
- It is a planning where it ignores the fact of partially ordered plan where many problems are not dependent.

- It searches through space not through state of the plan.

3. Planning as first order logical deduction: situation calculus

- The language called as PDDL (Planning Domain Definition Language) is used to balance and tackle by operating the high complex algorithm.
- The stating state is called as situation then action is applied on this situation. After applying particular set of action result is achieved. Even the result is regarded as situation.
- Fluent is the one which occurs when the relation is different from the situation
- On the basis of precondition, the action is executed.

9.4 ANALYSIS OF PLANNING APPROACHES

- Planning = Search + Logic
- From the above formula we can say that planning is a combination of searching and logic.
- Planner is a program that helps us to find the solution.
- GRAPHPLAN can be used as it will save or it will help in finding the difficult interactions which are due to mostly by mutex
- Even SATPLAN deals with the same kind of mutex relations.
- Subgoal is another task which is used to reach goal.

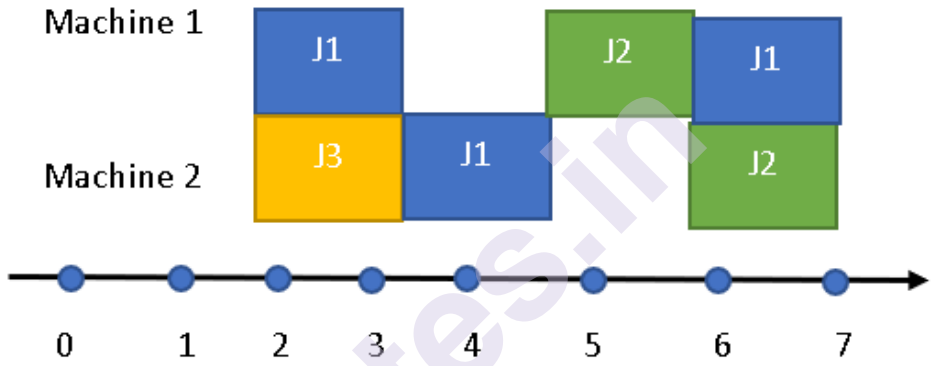
9.5 TIME, SCHEDULES AND RESOURCES

- Planning is done first, scheduling is done later
- With the help of planning the different types of actions that can be applied is decided again with the help of these set of actions that was decided during planning, out of this a suitable action is chosen and that action is scheduled to get the result.
- According the need of action resources are provided to it.

9.5.1 Representing Temporal and resource constraints:

- **Job Shop Scheduling (JSS)** or **Job Sop Problem** is used to do operational research and get the optimized results out of it.
- The main role of Job Shop Scheduling is to assign a job to the particular resource in that particular period of time.
- It has a particular section each section has its particular task associated with it.

- When it comes to Job Shop Scheduling machine can do only one work at a time.
- It follows non-pre-emptive approach i.e., if the job starts with one task then it has to complete fully, it can't stop in between.
- In the below example J1, J2 and J3 represents job and at the left end Machines are present with a number line starting at zero and going till 7.
- So, from the diagrammatic representation we can say that it takes around 7 for all the jobs to finish.
- It should be noted that the values by this is not always optimal in nature it may vary on the basis of machine, resource, kind of job etc.



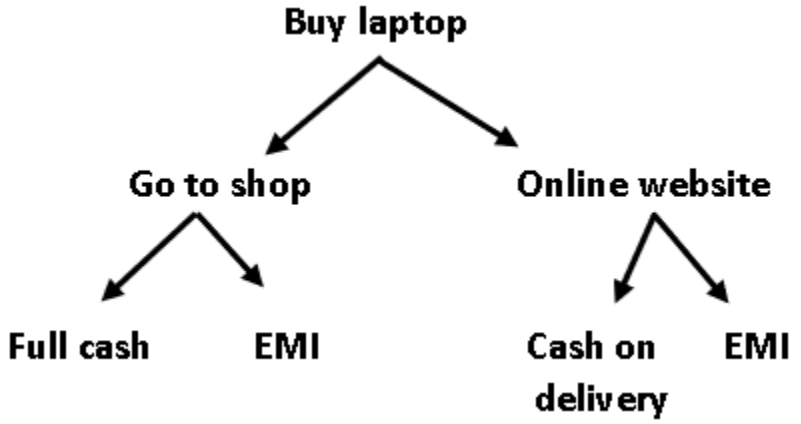
9.5.2 Aggregation:

- Aggregation is one in which the variable represented in the following manner like Man(3) instead of Man(l1), Man(l2) and Man(l3).
- The main aim of aggregation is nothing but groupism i.e., it may happen that there is a need of whole object to be used together in such scenarios we may use aggregation instead of calling each object individually.

9.5.3 Solving Scheduling Problems:

- In order to lower the time duration of planning phase we must start the plan and actions at the earliest so that it can be considered on the basis of ordering of the constraints and move on.
- With the help of directed graph, the orders of your plan and actions can be analysed.
- For efficient use of the above graph, CPM which stands for Critical Path Method can be applied to get the possible outcome for where to start and end.

9.6 HIERARCHICAL PLANNING



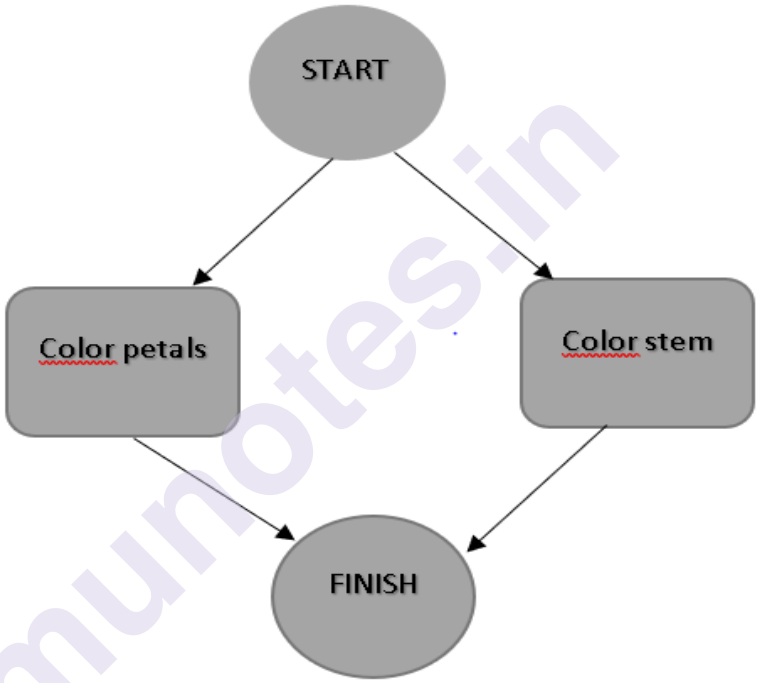
- In Hierarchical planning the importance is given to goal by ignoring all other actions that is also possible along with it.
- It is also known as plan decomposition
- The main plus point of hierarchical planning is that it's giving more important to the final target-based goal and ignoring the other things.
- Initially at the abstract level a plan is taken and applied by considering it.
- Then a particular solution on the basis of abstract consideration is applied and achieved and moved ahead by taking it as an input.
- Also, the abstract level can be single or multiple levels based on the need the selection and the process is done.
- Hierarchical planning involves to methods such as HTN also called as Hierarchical Task Network and POP called as Partial Order Planning.
- We will look at each types below:

9.6.1 Hierarchical Task Network (HTN):

- In the higher-level a less selection of actions is done and level by level it gets lessen at the lower level.
- HTN is regarded as an act of selecting the goal using a particular actions, to do this a proper selection of action is done at the top level this is called as ACT, then implementation of this act is followed in the lower level as well until the goal is achieved.
- When it comes to HTN, the first plan is regarded as very big level as many planning and actions has to be considered at this phase.
- As soon as the action is chosen and applied in that particular level, the decomposition of the level takes place into a partial one and this partial one is called as lower level.

9.6.2 Partial Order Planning (POP):

- In this the planning and action is done at the partial level i.e.; we will go little deeper in our scenario and try to achieve the goal.
- POP doesn't follows the action sequences, if you have 2 action then any of the one will be chosen and executed the main aim is to reach the goal.
- To understand POP let's take an example of colouring a flower which has a stem and petals
- So, imagine we coloured the petal first, then we coloured the stem.
- At the end the goal is achieved called as colouring.



9.6.3 POP one level planner:

- As the name is suggesting one level it does the same when it comes to application i.e., it will plan level by level.
- Let's take an example and understand this, so if you want to go travel to a particular destination then the first thing that you will do is buying a ticket so to do that you need to go to ticket counter stand in queue then wait for your turn to come then fill the ticket slip, submit id proof and at the end you will get the ticket. Such kind of planning and achieving the goal is nothing but one level planner.

9.7 PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS

- Nondeterministic domains are those in which nothing is known prior.

- In this plan has to be done with very less or no details
- Because of unknown details uncertainty occurs
- As the environment is less available or not available at all in this case perception is the best useful way to handle the situation, so that if the agent faces any sudden situation or need to do something in this case it can use the perception that was done earlier to apply particular action in that situation.
- The good solution to these kinds of problems is nothing but strategies.
- Knowledge is the key for every situation.

9.7.1 Some strategies are listed below:

- **Sensor less planning**

➔ In this there will be partial observability or no observability at all

➔ The agent as to use belief state which means that the agent needs to believe that something is present and move accordingly.

➔ It is also called as conformant planning

➔ It is not done on the basis of any perception.

➔ The only moto of this strategy is to reach goal

- **Contingent Planning:**

➔ It is also called as conditional planning

➔ On the basis of condition, the agent makes the plan, applies the action and executes it to reach the goal.

➔ The plan made on the basis of environment which can be partially observable or fully observable is appropriate.

➔ The variable which is used in contingent should be existential quantifier in nature.

➔ In this it can also use belief state if needed but this belief state must satisfy the condition, it may use formula for the same.

- **Online replanning:**

➔ In this particular technique a continuous checking is done which is nothing but the monitoring.

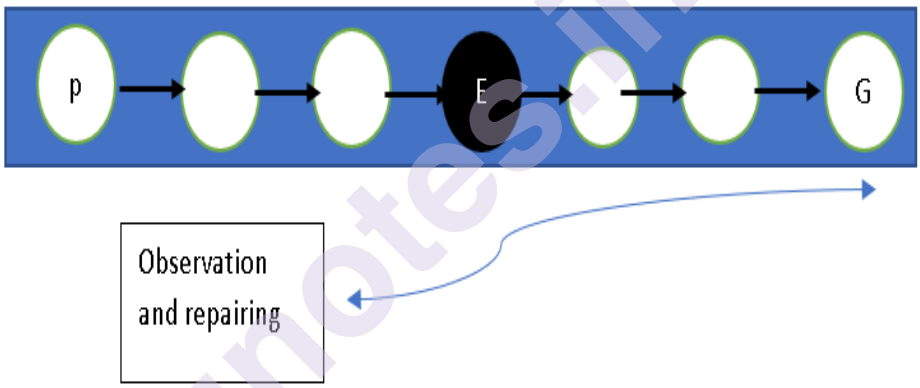
➔ Replanning is done by execution of it and a new plan is created.

➔ It is not always necessary that the whole plan has to be replanned, sometimes replanning may be done in particular part as well according to the need.

➔ In the below figure P is nothing but the plan that has been applied using this plan the agent will move ahead and side by side it will also observe the whole execution and once it gets to know that in the particular state say E the execution or the action that has been applied is not according to the need to reach the goal it will change the plan which is nothing but replanning or we can also call it as repairing. Now after replanning again the new actions are applied and executed to reach the goal in a very efficient manner.

➔ 3 monitoring are supported

1. **Action monitoring:** here the agent will check whether the actions are according to the precondition.
2. **Plan monitoring:** In this before running the action the agent will verify if it is according to the plan and whether it will give success.
3. **Goal monitoring:** Betterment of the goal is checked here by checking it.



9.8 MULTIAGENT PLANNING

- In multiagent each agent has its own goal
- A agent's main role is to plan, act and sense
- When there are multiagent in the environment each agent has its own goal and planning which leads to problem.

Categories:

- **Multieffector planning:** An agent having multiple effector associated with it is called as multieffector planning agent. To understand this, consider the following example: A person can sing and dance at the same time
- If these effectors are divided then we will get multibody.

Planning with multiple simultaneous actions:

- **Multiactor:** We will consider all multiagent, multibody and multieffector as one. So by doing this we are going to call all these as multiactor.
- Here the actor is pointing to agents, models and body.
- In this case the action is not always individual, over here the joint action is decided by all or many agents.
- Given by joint action(a1, a2, ….an)
- **Cooperation and Coordination:** The problem that arise is, as we say that joint decision has to be made then if there is any decision arises through any of the agent then each agent should agree with it. Achieving such agreement by every agent is what the main problem is. For this the agents should cooperate and coordinate with each other.
- The one solution that is possible is convention.
- If convention is not present then the substitution to it is communication between the agents.
- For example, the person who wants someone to play with them can call them and make team join in there team e.g.: "Join my team, John!"
- With evolution convention can arise
- One of the good examples of cooperative process is fish school where the group of fish swim together in particular fashion.

9.9 SUMMARY

- In this chapter how are what kind of planning has to be implemented to achieve goal is getting explained in a very brief manner.
- In forward chaining a initial phase is considered and it will move further till the final state whereas in backward chaining vice a versa is applied.
- Graphical representation of plan is done using GRAPHPLAN
- Decomposition of plan is done to get the other solutions that is possible in the graph such a way of finding the solution is called as hierarchical plan approach.
- Planning in nondeterministic approach is the one in which the details like environment or data are not known prior.
- Online planning is the one in which continuous monitoring is emphasised and replanning is made again to fine the most apt solution.

9.10 UNIT END QUESTIONS

- Explain replanning in detail.
- What do you mean by subgoals?
- What are nondeterministic domains? Explain planning adaption for non-deterministic domains
- Explain general characteristics of uncertain environments.
- Write a short note on conditional planning or contingency planning
- Explain online replanning in detail.
- Write a short note on how planning is done with Multiple simultaneous actions.
- Write a short note one sensorless planning or conformant planning

9.11 REFERENCES

- Artificial Intelligence: A modern approach by Stuart Russel and peter Norvig

  Publisher: Pearson 3rd edition year is 2015

- A first course in artificial intelligence by Deepak Khemani, TMH publisher frist edition 217
- Artificial intelligence: A ration approach by Rahul Deva, Shroff publishers, 1st edition with the year as 2018
- Artificial Intelligence by Elaine Rich, Kevin Knight and Shivashankar Nair 3rd edition 2018
- Artificial Intelligence and soft computing for beginners by Anandita das Bhattacharjee

*****

10

KNOWLEDGE REPRESENTATION

Unit Structure

10.0 CATEGORIES AND OBJECTS

- Facts are nothing but objects.
- Pink colour pen falls under object property.
- In this way many properties of an object can be grouped together and we can make them fall under categories.
- All dogs are mammals so these mammals can be called as category.
- With the help of inheritance, we can make it easy by using categories.
- We can show categories in two ways:

➢ One is using objects and predicates of FOL

Eg: Dog(x)

➢ Second way is by transforming the proposition into objects.

- Next important thing to consider is Subclass, subcategory.
- To understand subcategory let's understand the following example:

Let's take football, this football falls under category Ball, so now we can say that football is subcategory of ball.

- It can be represented as Football is subset of Balls.

- One another thing is consider this, sentence1: if its cloudy then it will rain, sentence 2: If it's raining then use umbrella, from these 2 sentences we can say that if its cloudy then use umbrella. This kind of creating new sentences or predicting something using the given data is nothing but inheritance.

10.1 EVENTS

- In the above topic we saw that the things were working on the basis of situation, but in this topic the event is done on the basis of time.
- It uses fluent and object to proceed further.
- Fluent is something where the behaviour is changed according to the time.
- We may represent fluent in this way:

  On(dress, flower pic)

To show that the fluent is true in that particular time, The alphabet T is used as follows:

T(On(dress, flower pic)

- In the above manner each predicate can be represented in there own unique way on the basis of there facts, let's see some example of the same:

➢ When the fluent becomes true after some gap, we can call it as restoration, so we will represent this as follows:

Restored(f,i)

Where f: fluent

i: interval

➢ When the fluent is true particular time, it is represented as follows:

T(f,t)

Where, T: true

f: fluent

t: time

➢ If it has to end then the word terminates is use to represent it, below is an example:

Terminates(e,f,t)

Where: e: Event

f: fluent

t: time

10.2 MENTAL EVENTS AND MENTAL OBJECTS

- In this deducing of knowledge happens.
- In our life also many times it happens that we want to know about something. We want to gain knowledge of something new. This new knowledge about something can be gained by asking a query from someone.
- This kind of gaining knowledge from someone's mind is called as mental objects and the way by changing these objects is called as mental events.

10.3 REASONING SYSTEMS OF CATEGORIES

- In order to apply reasoning in the categories we need to represent it in any of the one way from this;

➢ Semantic network

➢ Efficient algorithms

10.3.1 Semantic networks:

➢ It is also called as existential graphs.

➢ It is made up of nodes and edges.

➢ It uses propositional information to create graph so it is also called as propositional network.

➢ It is 2d dimension

➢ The 2d is on the basis of knowledge.

➢ The graph is made up of nodes, links and labels.

➢ Objects are represented by nodes.

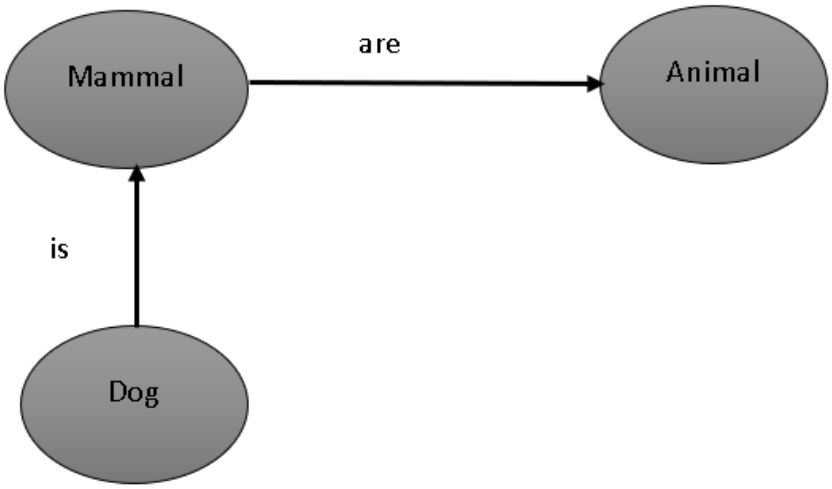
➢ Link represents the relationships.

➢ Nodes can be represented using oval or circle symbol.

➢ Directed link with "IS" or "has" etc representing the relationships

Eg: Dog is mammal

Animals are mammal



10.5.2 Description logics:

- It is an advance version of semantic network.
- It represents the knowledge in a diagrammatic way.

10.4 REASONING WITH DEFAULT INFORMATION

- It is non monotonic in nature
- Non monotonic is nothing but additional information that can get added in between and can lead to earlier conclusions.
- Changed notion on the truth is applied.

10.4.1 Circumscription and default logic:

- The meaning of circumscription is restriction of anything within some limits.
- A model may substitute by another if it has any unwanted or abnormal things inside it.
- The rule looks like this:

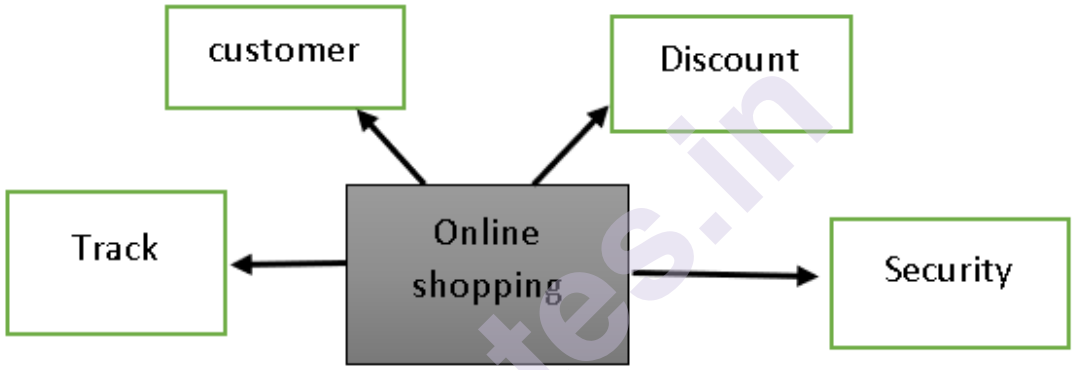
  Dog(i): barks(i)/barks(i)

- Based on the above rule we can say that the Dog(i) if true if it barks(i) .
- Which is leading to barks(i).

- If any probability interferes in the rule then it may be replaced or substituted.

10.5 INTERNET SHOPPING WORLD

- Almost all the present shopping websites and applications are using artificial intelligence.
- Using AI we can even judge which shopping website has the maximum visitors and accordingly ranking can be done.
- Also, if a particular website has customers who visit frequently then a personalized offers can be given.
- World wide web is the environment where the agent.



10.5.1 Following links:

- On all the modules of the website the links are added between the modules.
- This module will be able to interact and make it work with each other easily
- These re artificial intelligence based.
- A categorization is also done under this.
- All product details, customer details are available and stored.

10.6 SUMMARY

- In this chapter we saw that how the we categories the data.
- There are subcategories that can be applied. It uses fluent and objects.
- Knowledge gaining from someone else mind is explained in which mental object comes into picture.
- Reasoning system of categories is of two types one is semantic network and second one is efficient algorithms both helps us to represent the facts and knowledges through graphs and algorithms.

- Artificial Intelligence plays one of the important roles in internet shopping world as almost everything from creating the account till the product/service delivery requires intelligence.

10.7 UNIT END QUESTIONS

1. What do you mean by classical language?
2. What are categories with respect to Artificial Intelligence?
3. Write a short note on Mental Events
4. Write a short note on Events.
5. With the help of example explain in detail about semantic networks.
6. Explain briefly about internet shopping world

10.8 REFERENCES

- Artificial Intelligence: A modern approach by Stuart Russel and peter Norvig Publisher: Pearson 3rd edition year is 2015
- A first course in artificial intelligence by Deepak Khemani, TMH publisher frist edition 217
- Artificial intelligence: A ration approach by Rahul Deva, Shroff publishers, 1st edition with the year as 2018
- Artificial Intelligence by Elaine Rich, Kevin Knight and Shivashankar Nair 3rd edition 2018
- Artificial Intelligence and soft computing for beginners by Anandita das Bhattacharjee

*****