1. INTRODUCTION TO ARTIFICIAL INTELLIGENCE

INTRODUCTION

Artificial Intelligence is one of the booming technologies of computer science, which is ready to create a new revolution in the world by making intelligent machines. AI is now all around us. It is currently working with a variety of subfields, ranging from general to specific, such as self-driving cars, playing chess, proving theorems, playing music, painting etc. AI holds a tendency to cause a machine to work as a human.

What is Artificial Intelligence?

Artificial Intelligence is composed of two words Artificial and Intelligence, where Artificial defines "man-made," and intelligence defines "thinking power", hence AI means "a man-made thinking power."

So, we can define AI as, "It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions."

Why Artificial Intelligence?

- With the help of AI, we can create such software or devices which can solve real-world problems very easily and with accuracy such as health issues, marketing, traffic issues, etc.
- With the help of AI, we can create your personal virtual Assistant, such as Cortana, Google Assistant etc.
- With the help of AI, we can build such Robots which can work in an environment where survival of humans can be at risk.
- AI opens a path for other new technologies, new devices, and new Opportunities.

Goals of Artificial Intelligence: Following are the main goals of Artificial Intelligence:

1. Replicate human intelligence
2. Solve Knowledge-intensive tasks
3. An intelligent connection of perception and action
4. Building a machine which can perform tasks that requires human intelligence such as:
   - Proving a theorem
   - Playing chess
   - Plan some surgical operation
5. Driving a car in trafficCreating some system which can exhibit intelligent behavior, learn new things by itself, demonstrate, explain, and can advise to its user.

Definition: “AI is the study of how to make computers do things at which, at the moment, people are better”.

Advantages of Artificial Intelligence:

1. High Accuracy with fewer errors: AI machines or systems are prone to less errors and high accuracy as it takes decisions as per pre-experience or information.
2. High-Speed: AI systems can be of very high-speed and fast-decision making; because of that AI systems can beat a chess champion in the Chess game.
3. High reliability: AI machines are highly reliable and can perform the same action multiple times with high accuracy.
4. Useful for risky areas: AI machines can be helpful in situations such as defusing a bomb, exploring the ocean floor, where to employ a human can be risky.
5. Digital Assistant: AI can be very useful to provide digital assistant to the users such as AI technology is currently used by various E-commerce websites to show the products as per customer requirement.
6. Useful as a public utility: AI can be very useful for public utilities such as a self-driving car which can make our journey safer and hassle-free, facial recognition for security purpose, Natural language processing to communicate with the human in human-language, etc.

Disadvantages of Artificial Intelligence:

1. High Cost: The hardware and software requirement of AI is very costly as it requires lots of maintenance to meet current world requirements.
2. Can't think out of the box: Even we are making smarter machines with AI, but still they cannot work out of the box, as the robot will only do that work for which they are trained, or programmed.
3. No feelings and emotions: AI machines can be an outstanding performer, but still it does not have the feeling so it cannot make any kind of emotional attachment with human, and may sometime be harmful for users if the proper care is not taken.
4. Increase dependency on machines: With the increment of technology, people are getting more dependent on devices and hence they are losing their mental capabilities.
5. No Original Creativity: As humans are so creative and can imagine some new ideas but still AI machines cannot beat this power of human intelligence and cannot be creative and imaginative.

HISTORY OF AI

Year 1943: The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of artificial neurons.

Year 1949: Donald Hebb demonstrated an updating rule for modifying the connection strength between neurons. His rule is now called Hebbian learning.

Year 1950: The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes "Computing Machinery and Intelligence" in which he proposed a test. The test can check the machine's ability to exhibit intelligent behavior equivalent to human intelligence, called a Turing test.

Year 1955: An Allen Newell and Herbert A. Simon created the "first artificial intelligence program"Which was named as "Logic Theorist". This program had proved 38 of 52 Mathematics theorems, and find new and more elegant proofs for some theorems.

Year 1956: The word "Artificial Intelligence" first adopted by American Computer scientist John McCarthy at the Dartmouth Conference. For the first time, AI coined as an academic field.

Year 1966: The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.

Year 1972: The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

Year 1974-1980: The duration between years 1974 to 1980 was the first AI winter duration. AI winter refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches.

Year 1980: After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert.

Year 1987-1993: The duration between the years 1987 to 1993 was the second AI Winter duration. Again Investors and government stopped in funding for AI research as due to high cost but not efficient result.

Year 1997: In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.

Year 2002: for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.

Year 2006: AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

Year 2011: In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.

Year 2012: Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.

Year 2014: In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."
Year 2018: The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.

Now AI has developed to a remarkable level. The concept of Deep learning, big data, and data science are now trending like a boom. Nowadays companies like Google, Facebook, IBM, and Amazon are working with AI and creating amazing devices. The future of Artificial Intelligence is inspiring and will come with high intelligence.

INTELLIGENT SYSTEMS

AI is the combination of computer science, physiology and philosophy. AI is a broad topic consisting of different fields from Machine vision to expert systems. John McCarthy was one of the founders of AI field who stated that "AI is the science and engineering of making intelligent machines, especially intelligent computer programs".

Other Definitions of AI:

1. AI is the study of mental faculties through the use computational models.
2. The art of creating machines that performs functions which require intelligence when performed by people.
3. AI is a field of study that seeks to explain & emulate intelligent behavior in terms of computational processes.
4. **AI is the study of how to make computers do things at which, at the moment, people are better.**
5. AI is the study of the computations that make it possible to perceive reason & act.
6. AI is the branch of computer science that is concerned with the automation of intelligent behavior.

ELIZA: ELIZA was a program that conversed with user in English. The program was able to converse about any subject, because it stored subject information in data banks. Another feature of ELIZA was its ability to pick up speech patterns from user's questions & provide responses using those patterns. The following is a passage that contains dialogue between Eliza & a teenage girl

ELIZA: Hello. I am Eliza.
GIRL: Can you help me?
ELIZA: Don't you believe that I can help you?
GIRL: Yes.
ELIZA: I see.

GIRL: I feel depressed.

ELIZA: How long have you been feeling depressed?

GIRL: Since last month.

ELIZA: That is quite interesting.

GIRL: What is interesting in this?

ELIZA: Have you asked anyone else?

GIRL: NO.

ELIZA: Are you saying no just to be negative?

GIRL: NO I did not ask anyone.

ELIZA: Why no?

GIRL: I do not believe other people.

ELIZA: Say, do you have any psychological problems?

GIRL: I do not know.

ELIZA: Tell me more.

Since then a lot many versions of ELIZA have been developed & are available on the internet. The basic philosophy & characteristics in all these programs are same.

Characteristics of ELIZA:

1. Simulation of Intelligence
2. Quality of Response
3. Coherence
4. Semantics

Simulation of Intelligence: Eliza programs are not intelligent at all in real sense. The do not understand the meaning of utterance. Instead these programs simulate intelligent behavior quite effectively by recognizing keywords & phrases. By using a table lookup, one of a few ways of responding question is chosen.

Quality of Response: It is limited by the sophistication of the ways in which they can process the input text at a syntactic level. For example, the number of templates available is a serious limitation. However, the success depends heavily on the fact that the user has a fairly restricted notion of the expected response from the system.

Coherence: The earlier version of the system imposed no structure on the conversation. Each statement was based entirely on the current input & no context information was used. More complex versions of Eliza can do a little better. Any sense of intelligence depends strongly on the coherence of the conversation as judged by the user.

Semantics: Such systems have no semantic representation of the content of either the user's input or the reply. That is why we say that it does not have intelligence of understanding of what we are saying. But it looks that it imitates the human conversation style. Because of this, it passed Turing test.

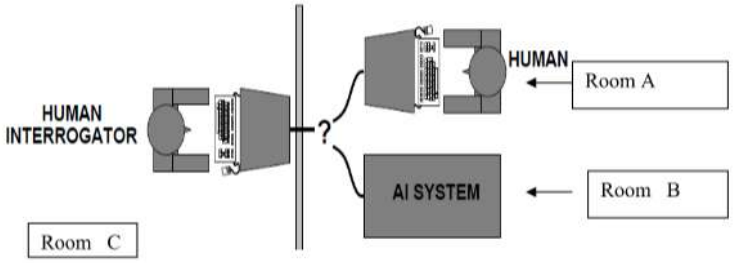
Categorization of Intelligent Systems:

- System that thinks like humans
- System that acts like humans
- System that thinks rationally
- Systems that acts rationally

1. System that thinks like humans: This requires cognitive modeling approaches. Most of the time, it is a black box where we are not clear about our thought process. One has to know the functioning of the brain & its mechanism for processing information.
2. System that acts like humans: This requires that the overall behavior of the system should be human like which could be achieved by observation. Turing test is an example.

   Turing Test: Turing Test was introduced by Alan Turing in 1950. A Turing Test is a method of inquiry in artificial intelligence (AI) for determining whether or not a computer is capable of thinking like a human.

   To conduct this test, we need two people and a machine to be evaluated. One person plays the role of an interrogator, who is in a separate room from the computer and the other person. The interrogator can ask questions of either the person or the computer by typing the questions and receiving typed responses. The interrogator knows them only as A and B and aims to determine which is the person and is a machine. The goal of the machine is to fool the interrogator into believing that the machine can think. If a large multiplication, for example, given the computer can give a wrong answer by taking a long time for calculation as a man can do, to fool the interrogator.



   Turing proposed that if the human interrogator in Room C is not able to identify who is in Room A or in Room B, then the machine possesses intelligence. Turing considered this is a sufficient test for

attributing thinking capacity to a machine. As of today, Turing test is the ultimate test a machine must pass in order to be called as intelligent test.

3. System that thinks rationally: This relies on logic rather than human to measure correctness. For example, given *John is a human and all humans are mortal* than one can conclude logically that *John is mortal*.
4. System that acts rationally: This is the final category of intelligent system whereby rational behavior we mean doing the right thing. Even if the method is illogical, the observed behavior must be rational.

Components of AI:

- Knowledge base
- Control strategy
- Inference mechanism

Knowledge base: Ai programs should be learning in nature & update its knowledge accordingly. Knowledge base generally consists of facts & rules & has the following characteristics:

- It is voluminous in nature & requires proper structuring
- It may be incomplete & imprecise
- It may be dynamic & keep on changing

Control Strategy: It determines which rule to be applied. To know this rule, some heuristics or thumb rules based on problem domain may be used.

Inference mechanism: It requires search through knowledge base & derives new knowledge using the existing knowledge with the help of inference rules.

FOUNDATIONS OF AI

The foundations of AI in various fields are as follows

- Mathematics
- Neuroscience
- Control theory
- Linguistics

Mathematics: AI systems use formal logical methods & Boolean logic, Analysis of limits to what to be computed, probability theory & uncertainty that forms the basis for most approaches to AI, fuzzy logic etc.

Neuroscience: This science of medicine helps in studying the function of brain. Recent studies use accurate sensors to correlate brain activity to human thought. By monitoring individual neurons, monkeys can now control a computer mouse using thought alone. Moore's law states that the computers will have as many gates as humans have neurons. Researchers are working to know as to how to have a mechanical brain. Such systems will require parallel computation, remapping and interconnections to a large extent.

Control Theory: Machines can modify their behaviour in response to the environment. Steam engine governor, thermostat & water flow regulator are few examples of Control Theory. This theory of stable feedback systems helps in building systems that transition from initial state to goal state with minimum energy.

Linguistics: Speech demonstrates so much of human intelligence. Analysis of human language reveals thought taking place in ways not understood in other settings. Children can create sentences they have never heard before. Languages & thoughts are believed to be tightly intertwined.

APPLICATIONS OF AI

1. Gaming: AI plays crucial role in strategic games such as chess, poker, tic-tac-toe, etc., where machine can think of large number of possible positions based on heuristic knowledge.
2. Natural Language Processing: It is possible to interact with the computer that understands natural language spoken by humans.
3. Expert Systems: There are some applications which integrate machine, software and special information to impart reasoning and advising. They provide explanation and advice to the users.
4. Vision Systems: These systems understand, interpret and comprehend visual input on the computer. For example,
   a. A spying aeroplane takes photographs, which are used to figure out spatial information or map of the areas.
   b. Doctors use clinical expert system to diagnose the patient.
   c. Police use computer software that can recognize the face of criminal with the stored portrait made by forensic artist.
5. Speech Recognition: Some intelligent systems are capable of hearing and comprehending the language in terms of sentences and their meanings while a human talks to it. It can handle different accents, slang words, noise in the background, change in human's noise due to cold, etc.
6. Handwriting Recognition: The handwriting recognition software reads the text written on paper by a pen or on screen by a stylus. It can recognize the shapes of the letters and converts it into editable text.
7. Intelligent Robots: Robots are able to perform the tasks given by humans. They have special sensors to detect physical data from the real world such as light, heat, temperature, movement, sound, bump, and

pressure. They have efficient processors, multiple sensors and huge memory, to exhibit intelligence. In addition, they are capable of learning from their mistakes and they can adapt to the new environment.

TIC-TAC-TOE GAME PLAYING

TIC-TAC-TOE is a two-player game with one player marking **O** & other marking **X**, at their turn in the spaces in a 3X3 grid. The player who succeeds in playing 3 respective marks in any horizontal, vertical or diagonal row wins the game.

Here we are considering one human player & the other player to be a computer program. The objective to play this game using computer is to write a program which never loses. Below are 3 approaches to play this game which increase in

- Complexity
- Use of generalization
- Clarity of their knowledge
- Extensibility of their approach

Approach 1:

Let us represent 3X3 board as nine elements vector. Each element in a vector can contain any of the following 3 digits:

- 0 - representing blank position
- 1 - indicating X player move
- 2 - indicating O player move

It is assumed that this program makes use of a move table that consists of vector of $\mathbf{3^9}$ (19683) elements.

Index	Current Board Position	New Board Position
0	000000000	000010000
1	000000001	020000001
2	000000002	000100002
3	000000010	002000010
	. . .	

All the possible board positions are stored in Current Board Position column along with its corresponding next best possible board position in New Board Position column.

Algorithm:

- View the vector (board) as a ternary number.
- Get an index by converting this vector to its corresponding decimal number.
- Get the vector from New Board Position stored at the index. The vector thus selected represents the way the board will look after the move that should be made.
- So set board position equal to that vector.

Advantage:

- Very efficient in terms of time.

Disadvantages:

- Requires lot of memory space to store move table.
- Lot of work is required to specify entries in move table manually.
- This approach cannot be extended to 3D TIC-TAC-TOE as 3^{27} board positions are to be stored.

Approach 2:

The board B[1..9] is represented by a 9-element vector.

- 2 - Representing blank position
- 3 - Indicating X player move
- 5 - Indicating O player move

In his approach we use the following 3 sub procedures.

- Go(n) – Using this function computer can make a move in square n.
- Make_2 – This function helps the computer to make valid 2 moves.
- PossWin(P) – If player P can win in the next move then it returns the index (from 1 to 9) of the square that constitutes a winning move, otherwise it returns 0.

The strategy applied by human for this game is that if human is winning in the next move the human plays in the desired square, else if human is not winning in the next move then one checks if the opponent is winning. If so then block that square, otherwise try making valid 2 in any row, column (or) diagonal.

The function PossWin operates by checking, one at a time, for each of rows/columns & diagonals.

- If PossWin(P)=0, then P cannot win. Find whether opponent can win. If so then block it. This can be achieved as follows:

- If (3*3*2=18) then X player can win as there is one blank square in row, column (or) diagonal.
- If (5*5*2=50) then player O can win.

Advantages:

- Memory Efficient
- Easy to understand

Disadvantages:

- Not as efficient as first one with respect to time
- This approach cannot be extended to 3D TIC-TAC-TOE

Approach 3:

In this approach, we choose board position to be a magic square of order 3; blocks numbered by magic number. The magic square of order n consists of n^2 distinct numbers (from 1 to n^2), such that the numbers in all rows, all columns & both diagonals sum to be 15.

8	1	6
3	5	7
4	9	2

Magic Square of Order 3

In this approach, we maintain a list of the blocks played by each player. For the sake of convenience, each block is identified by its number. The following strategy for possible win for a player is used.

- Each pair of blocks a player owns is considered.
- Difference D between 15 & the sum of the two blocks is computed.
    - If D<0 or D>9, then these two blocks are not collinear & so can be ignored. Otherwise if the block representing difference is blank (not in either list) then player can move in that block.
- This strategy will produce a possible win for a player.

Execution: Here, Human uses mind & Machine uses calculations. Assuming that Machine is the first player

Move 1: Machine: Takes the element 5.

Move 2: Human: Takes the element 8.

Move 3: Machine: Takes the element 4. So, the machine elements consists of 5, 4.

Move 4: Human: Takes the element 6. So, the human elements consists of 8, 6.

Move 5: Machine: From the above moves machine has the elements 5, 4. First machine checks whether it can win. 5+4=9, 15-9=6. Since the element 6 is already taken by the human, there is no possibility of machine winning in this move. Now machine checks whether human can win (or) not. The elements taken by human 8,

6. So, 8+6=14, 15-14=1. Since the element 1 is available the machine takes the element 1. Finally, the elements taken by the Machine are 5, 4, 1.

Move 6: Human: Takes the element 3. So, the human elements consist of 8, 6, 3.

Move 7: Machine: Considering all the elements taken by the 5, 4, 1 now 5+1=6, 15-6=9. Since the element 9 is available machine takes the element 9.

Therefore, 1+5+9=15 which leads to Machine winning state.

Advantage:

- This approach could extend to handle three-dimensional TIC-TAC-TOE.

Disadvantage:

- Requires more time than other 2 approaches as it must search a tree representing all possible move sequences before making each move.

DEVELOPMENT OF AI LANGUAGES

AI Languages have traditionally been those which stress on knowledge representation schemes, pattern matching, flexible search & programs as data. Examples of such languages are

- LISP
- Prolog
- Pop-2
- Machine Learning (ML)

LISP: LISP is a functional Language based on lambda Calculus.

Prolog: It is based on first-order predicate logic. Both the languages are declarative languages where one is concerned about 'what to compute' & not 'how to compute'.

Pop-2: It is based on stack-based language providing greater flexibility. It is similar to LISP.

Machine Learning (ML): It is an application of AI that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it learn for themselves. The primary aim is to allow the computers learn automatically without human intervention.

CURRENT TRENDS IN AI

- Hard Computing Techniques
- Soft Computing Techniques

Conventional Computing (Hard Computing) is based on the concept of precise modeling & analyzing to yield accurate results. Hard Computing Techniques works well for simple problems, but is bound by NP-complete set which include problems often occurring in biology, medicine, humanities, management sciences & similar fields.

Soft Computing is a formal computer Science area of study which refers to a collection of computational techniques in computer science, machine learning & some engineering disciplines, which study, model & analyze very complex phenomena. Components of Soft Computing include Neural Networks, Fuzzy Systems, Evolutionary Algorithms, Swarm Intelligence etc.

- Neural Networks have been developed based on functioning of human brains. Attempts to model the biological neuron have led to development of the field called Artificial Neuron Network.
- Fuzzy Logic (FL) is a method of reasoning that resembles human reasoning. The approach of FL imitates the way of decision making in humans that involves all intermediate possibilities between digital values YES and NO. The conventional logic block that a computer can understand takes precise input and produces a definite output as TRUE or FALSE, which is equivalent to human's YES or NO.
- Evolutionary techniques mostly involve meta-heuristic optimization algorithms such as evolutionary algorithms & Swarm Intelligence.
- Genetic algorithms were developed mainly by emulating nature & behavior of biological chromosome.
- Ant colony algorithm was developed to emulate the behavior of real ants. An ant algorithm is one in which a set of artificial ants (agents) cooperate to find the solution of a problem by exchanging information on graph edges.
- Swarm Intelligence (SI) is a type of AI based on the collective behavior of decentralized, self organized systems. Social insects like ants, bees, wasps & termites perform their tasks independent of other members of the colony. However, they are able to solve complex problems emerging in their daily lives by mutual cooperation. This emergent behavior of self organization by a group of social insects is known as Swarm Intelligence.
- Expert system continues to remain an attractive field for its practical utility in all walk of real life.
- Emergence of Agent technology as a subfield of AI is a significant paradigm shift for software development & break through as a new revolution. Agents are generally suited in some environment & are capable of taking autonomous decisions while solving a problem. Multi Agent Systems are designed using several independent & interacting agents to solve the problems of distributed nature.

2. PROBLEM SOLVING: STATE-SPACE SEARCH AND CONTROL STRATEGIES

INTRODUCTION

Problem solving is a method of deriving solution steps beginning from initial description of the problem to the desired solution. In AI, the problems are frequently modeled as a state space problem where the state space is a set of all possible states from start to goal states.

The 2 types of problem-solving methods that are generally followed include general purpose & special purpose methods. A general purpose method is applicable to a wide variety of problems, where a special purpose method is a tailor method made for a particular problem. The most general approach for solving a problem is to generate the solution & test it. For generating new state in the search space, an action/operator/rule is applied & tested whether the state is the goal state or not. In case the state is not the goal state, the procedure is repeated. The order of application of rules to the current state is called control strategy.

GENERAL PROBLEM SOLVING

Production System:

Production System (PS) is one of the formalisms that help AI programs to do search process more conveniently in state-space problems. This system consists of start (initial) state(s) & goal (final) state(s) of the problem along with one or more databases consisting of suitable & necessary information for a particular task. Production System consists of a number of production rules.

Example 1: Water Jug Problem

Problem Statement: There are two jugs a 4-Gallon one and 3-Gallon one. Neither has any measuring marker on it. There is a pump that can be used to fill the jugs with water. How can we get exactly 2- Gallon water into the 4- Gallon jug?

The state space for this problem can be described as the set of ordered pairs of integers (x, y) such that x = 0, 1, 2, 3 or 4 and y = 0, 1, 2 or 3. x represents the number of gallons of water in the 4- Gallon jug. y represents the number of gallons of water in the 3- Gallon jug. The start state is (0, 0). The goal state is (2, n) for any value of n.

S.No	Rule	Result	Rule Description
1.	(x, y) if x<4	(4, y)	fill the 4-g jug
2.	(x, y) if y<3	(x, 3)	fill the 3-g jug

3.	(x, y) if x>0	(x-d, y)	pour some water out of the 4-g jug
4.	(x, y) if y>0	(x, y-d)	pour some water out of the 3-g jug
5.	(x, y) if x>0	(0, y)	Empty the 4-g jug on the ground
6.	(x, y) if y>0	(x,0)	Empty the 3-g jug on the ground
7.	(x, y) if x + y ≥ 4 & y>0	(4,y-(4-x))	Pour water from the 3-g jug into the 4-g jug until the 4-g jug is full
8.	(x, y) if x + y>3 & x>0	(x-(3-y), 3)	pour water from the 4-g jug into the 3-g jug until the 3-g jug is full
9.	(x, y) if x + y ≤ 4 & y>0	(x+y,0)	Pour all the water from 3-g jug into 4-g jug.
10.	(x, y) If x + y ≤ 3 & x>0	(0,x+y)	Pour all the water from the 4-g jug into

One Solution to the Water Jug Problem:

Gallon in the 4-Gallon Jug	Gallon in the 3-Gallon Jug	Rule Applied
0	0	
		2
0	3	
		9
3	0	
		2
3	3	
		7
4	2	
		5/12
0	2	
		9/11
2	0	

Example 2: Water Jug Problem

Problem Statement: There are two jugs a 5-Gallon one and 3-Gallon one. Neither has any measuring marker on it. There is a pump that can be used to fill the jugs with water. How can we get exactly 4- Gallon water into the 5- Gallon jug?

The state space for this problem can be described as the set of ordered pairs of integers (x, y) such that x = 0, 1, 2, 3, 4 or 5 and y = 0, 1, 2 or 3. x represents the number of gallons of water in the 5- Gallon jug. y represents the number of gallons of water in the 3- Gallon jug. The start state is (0, 0). The goal state is (4, n) for any value of n.

Rule No	Left of rule	Right of rule	Description
1	$(X, Y \mid X < 5)$	(5, Y)	Fill 5-g jug
2	$(X, Y \mid X > 0)$	(0, Y)	Empty 5-g jug
3	$(X, Y \mid Y < 3)$	(X, 3)	Fill 3-g jug
4	$(X, Y \mid Y > 0)$	(X,0)	Empty 3-g jug
5	$(X, Y \mid X + Y \leq 5 \wedge Y > 0)$	(X + Y, 0)	Empty 3-g into 5-g jug
6	$(X, Y \mid X + Y \leq 3 \wedge X > 0)$	(0, X + Y)	Empty 5-g into 3-g jug
7	$(X, Y \mid X + Y \geq 5 \wedge Y > 0)$	(5, Y – (5 – X)) until 5-g jug is full	Pour water from 3-g jug into 5-g jug
8	$(X, Y \mid X + Y \geq 3 \wedge X > 0)$	(X – (3 – Y), 3)	Pour water from 5-g jug into 3-g jug until 3-g jug is full

Table: Production Rules for Water Jug Problem

One Solution to the Water Jug Problem:

Rule Applied	5-G Jug	3-G Jug
Start State	0	0
1	5	0
8	2	3
4	2	0
6	0	2
1	5	2
8	4	3
Goal State	4	0

Example 3: Missionaries & Cannibals Problem

Problem Statement: Three missionaries & three cannibals want to cross a river. There is a boat on their side of the river that can be used by either 1 (or) 2 persons. How should they use this boat to cross the river in such a way that cannibals never outnumber missionaries on either side of the river? If the cannibals ever outnumber the missionaries (on either bank) then the missionaries will be eaten. How can they cross over without eaten? Consider Missionaries as 'M', Cannibals as 'C' & Boat as 'B' which are on the same side of the river.

Initial State: ([3M, 3C, 1B], [0M, 0C, 0B]) Goal State: ([0M, 0C, 0B], [3M, 3C, 1B])

Production rules are as follows:

Rule 1: (0, M): One Missionary sailing the boat from Bank-1 to Bank-2.

Rule 2: (M, 0): One Missionary sailing the boat from Bank-2 to Bank-1.

Rule 3: (M, M): Two Missionaries sailing the boat from Bank-1 to Bank-2.

Rule 4: (M, M): Two Missionaries sailing the boat from Bank-2 to Bank-1.

Rule 5: (M, C): One Missionary & One Cannibal sailing the boat from Bank-1 to Bank-2.

Rule 6: (C, M): One Cannibal & One Missionary sailing the boat from Bank-2 to Bank-1.

Rule 7: (C, C): Two Cannibals sailing the boat from Bank-1 to Bank-2.

Rule 8: (C, C): Two Cannibals sailing the boat from Bank-2 to Bank-1.

Rule 9: (0, C): One Cannibal sailing the boat from Bank-1 to Bank-2.

Rule 10: (C, 0): One Cannibal sailing the boat from Bank-2 to Bank-1.

S.No	Rule Applied	Persons on River Bank-1	Persons on River Bank-2
1	Start State	3M,3C,1B	0M,0C,0B
2	5	2M,2C,0B	1M,1C,1B
3	2	3M,2C,1B	0M,1C,0B
4	7	3M,0C,0B	0M,3C,1B
5	10	3M,1C,1B	0M,2C,0B
6	3	1M,1C,0B	2M,2C,1B
7	6	2M,2C,1B	1M,1C,0B
8	3	0M,2C,0B	3M,1C,1B
9	10	0M,3C,1B	3M,0C,0B
10	7	0M,1C,0B	3M,2C,1B
11	10	0M,2C,1B	3M,1C,0B
12	7	0M,0C,0B	3M,3C,1B

State Space Search:

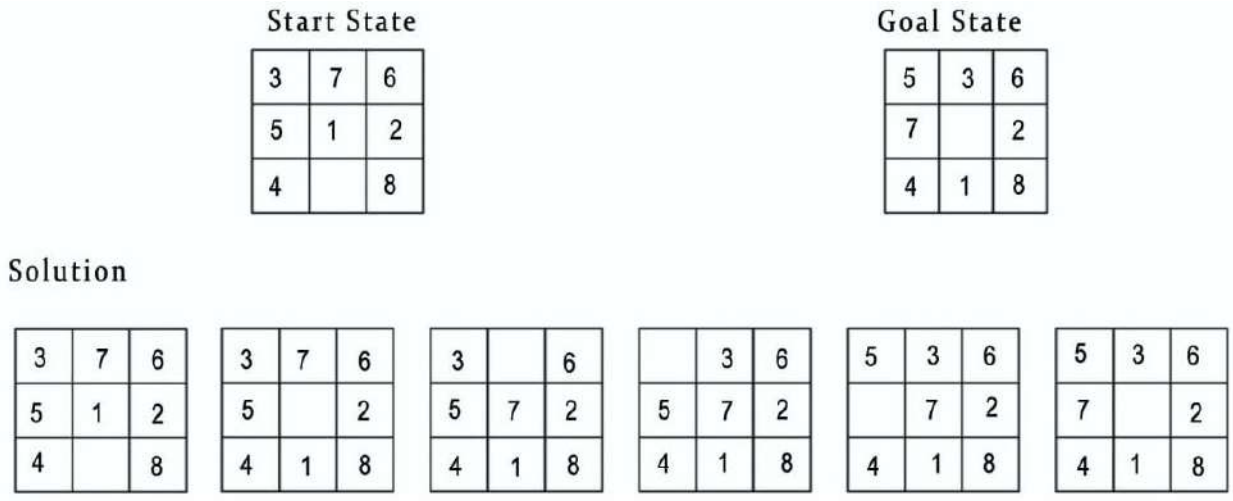
A State Space Search is another method of problem representation that facilitates easy search. Using this method, we can also find a path from start state to goal state while solving a problem. A state space basically consists of 4 components:

1. A set S containing start states of the problem.
2. A set G containing goal states of the problem.
3. Set of nodes (states) in the graph/tree. Each node represents the state in problem-solving process.
4. Set of arcs connecting nodes. Each arc corresponds to operator that is a step in a problem-solving process.

A solution path is a path through the graph from a node in S to a node in G. The main objective of a search algorithm is to determine a solution path in graph. There may be more than one solution paths, as there may be more than one ways of solving the problem.

Example: The Eight-Puzzle Problem

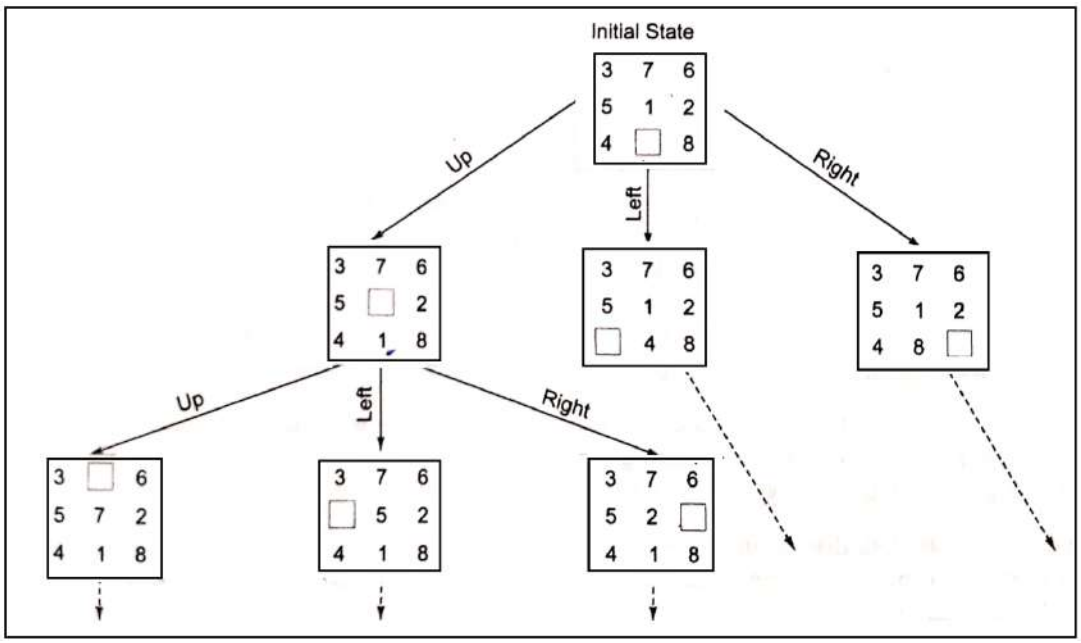
Problem Statement: The eight-puzzle problem has a 3X3 grid with 8 randomly numbered (1 to 8) tiles arranged on it with one empty cell. At any point, the adjacent tile can move to the empty cell, creating a new empty cell. Solving this problem involves arranging tiles such that we get the goal state from the start state.



A state for this problem should keep track of the position of all tiles on the game board, with 0 representing the blank (empty cell) position on the board. The start & goal states may be represented as follows with each list representing corresponding row:

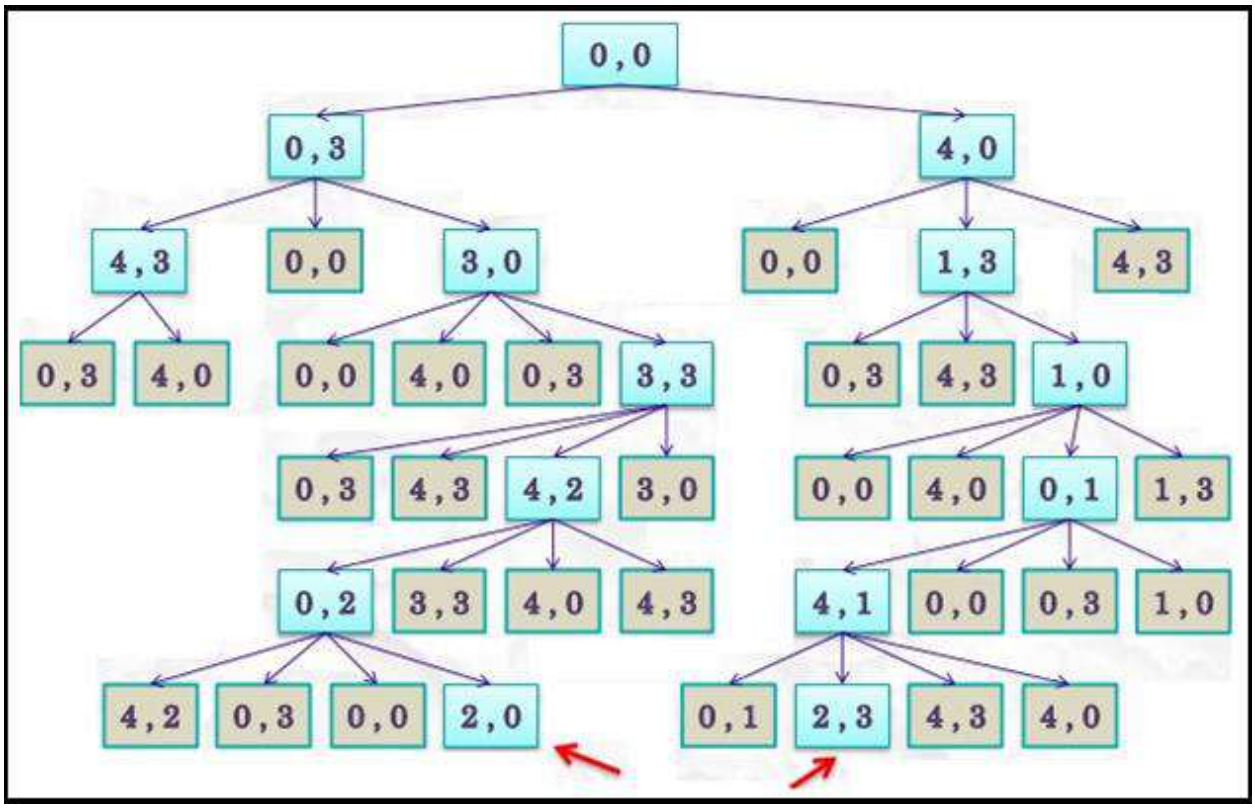
1. Start state: [[3, 7, 6], [5, 1, 2], [4, 0, 8]]
2. Goal state: [[5, 3, 6], [7, 0, 2], [4, 1, 8]]
3. The operators can be thought of moving {Up, Down, Left, Right}, the direction in which blank space effectively moves.

Solution: Following is a Partial Search Tree for Eight Puzzle Problem



The search will be continued until the goal state is reached.

Search Tree for Water Jug Problem:



Example: Chess Game (One Legal Chess Move)

Chess is basically a competitive 2 player game played on a chequered board with 64 squares arranged in an 8 X 8 square. Each player is given 16 pieces of the same colour (black or white). These include 1 King, 1 Queen, 2 Rooks, 2 Knights, 2 Bishops & 8 pawns. Each of these pieces move in a unique manner. The player who chooses the white pieces gets the first turn to play. The players get alternate chances in which they can move one piece at a time. The objective of this game is to remove the opponent's king from the game. The opponent's King has to be placed in such a situation where the king is under immediate attack & there is no way to save it from the attack. This is known as Checkmate.

For a problem playing chess the starting position can be described as an 8 X 8 array where each position contains a symbol standing for appropriate piece in the official chess opening position. We can define our goal as any board position in which the opponent does not have a legal move & his/her king is under attack. The legal moves provide the way of getting from initial state to goal state. They can be described easily as a set of rules consisting of 2 parts

- A left side that serves as a pattern to be matched against the current board position
- A right side that describes the change to be made to the board position to reflect the move

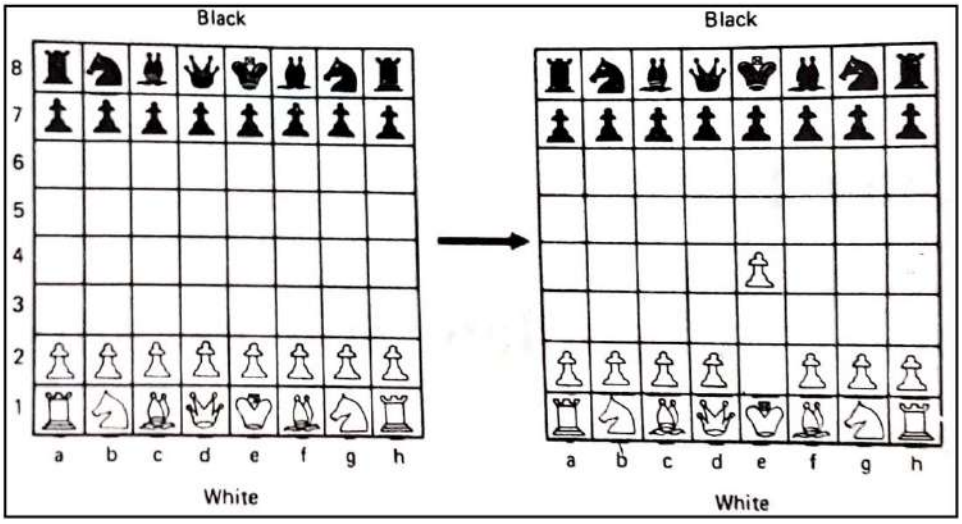


Fig: One Legal Chess Move

Note: There will be several number of Rules.

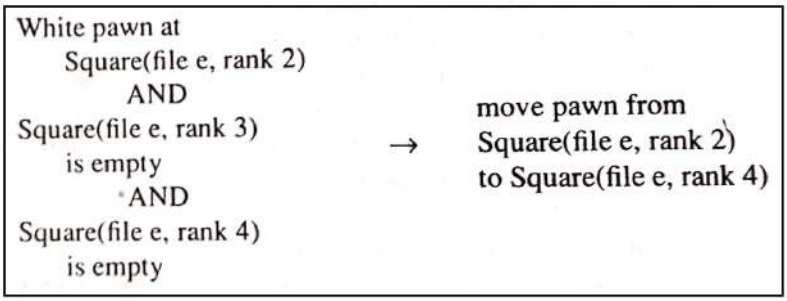


Fig: Another way to Describe Chess Moves

Control Strategies:

Control strategy is one of the most important components of problem solving that describes the order of application of the rules to the current state. Control strategy should be such that it causes motion towards a solution. The second requirement of control strategy is that it should explore the solution space in a systematic manner. Depth-First & Breadth-First are systematic control strategies. There are 2 directions in which a search could proceed

- Data-Driven Search, called Forward Chaining, from the Start State
- Goal-Driven Search, called Backward Chaining, from the Goal State

Forward Chaining: The process of forward chaining begins with known facts & works towards a solution. For example, in 8-puzzle problem, we start from the start state & work forward to the goal state. In this case, we begin building a tree of move sequences with the root of the tree as the start state. The states of next level of the tree are generated by finding all rules whose left sides match with root & use their right side to create the new state. This process is continued until a configuration that matches the goal state is generated.

Backward Chaining: It is a goal directed strategy that begins with the goal state & continues working backward, generating more sub-goals that must also be satisfied to satisfy main goal until we reach to start state. Prolog (Programming in Logic) language uses this strategy. In this case, we begin building a tree of move sequences with the goal state of the tree as the start state. The states of next level of the tree are generated by finding all rules whose right sides match with goal state & use their left side to create the new state. This process is continued until a configuration that matches the start state is generated.

Note: We can use both Data-Driven & Goal-Driven strategies for problem solving, depending on the nature of the problem.

CHARACTERISTICS OF PROBLEM

1. Type of Problems: There are 3 types of problems in real life,
    - Ignorable
    - Recoverable
    - Irrecoverable

Ignorable: These are the problems where we can ignore the solution steps. For example, in proving a theorem, if some lemma is proved to prove a theorem & later on we realize that it is not useful, then we can ignore this solution step & prove another lemma. Such problems can be solved using simple control strategy.

Recoverable: These are the problems where solution steps can be undone. For example, in Water Jug Problem, if we have filled up the jug, we can empty it also. Any state can be reached again by undoing the steps. These problems are generally puzzles played by a single player. Such problems can be solved by back tracking, so control strategy can be implemented using a push-down stack.

Irrecoverable: The problems where solution steps cannot be undone. For example, any 2-Player games such as chess, playing cards, snake & ladder etc are example of this category. Such problems can be solved by planning process.

2. Decomposability of a Problem: Divide the problem into a set of independent smaller sub-problems, solve them and combine the solution to get the final solution. The process of dividing sub-problems continues till we get the set of the smallest sub-problems for which a small collection of specific rules are used. Divide-And-Conquer technique is the commonly used method for solving such problems. It is an important & useful characteristic, as each sub-problem is simpler to solve & can be handed over to a different processor. Thus, such problems can be solved in parallel processing environment.

3. Roll of Knowledge: Knowledge plays an important role in solving any problem. Knowledge could be in the form of rules & facts which help generating search space for finding the solution.

4. Consistency of Knowledge Base used in Solving Problem: Make sure that knowledge base used to solve problem is consistent. Inconsistent knowledge base will lead to wrong solutions. For example, if we have knowledge in the form of rules & facts as follows:

If it is humid, it will rain. If it is sunny, then it is day time. It is sunny day. It is night time.

This knowledge is not consistent as there is a contradiction because ‘it is a day time’ can be deduced from the knowledge, & thus both ‘it is night time’ and ‘it is a day time’ are not possible at the same time. If knowledge base has such inconsistency, then some methods may be used to avoid such conflicts.

5. Requirement of Solution: We should analyze the problem whether solution require is absolute (or) relative. We call solution to be absolute if we have to find exact solution, where as it is relative if we have reasonable good & approximate solution. For example, in Water Jug Problem, if there are more than one ways to solve a problem, then we follow one path successfully. There is no need to go back & find a better solution. In this case, the solution is absolute. In travelling sales man problem, our goal is to find the shortest route, unless all routes are known, it is difficult to know the shortest route. This is the Best-Path problem; where as Water Jug is Any-Path problem. Any-Path problem is generally solved in reasonable amount of time by using heuristics that suggest good paths to explore. Best-Path problems are computationally harder compared with Any-Path problems.

EXHAUSTIVE SEARCHES (OR) UNIFORMED SEARCHES

- Breadth-First Search
- Depth-First Search
- Depth-First Iterative Deepening
- Bidirectional Search

1. Breadth-First Search (BFS):

- BFS is the most common search strategy for traversing a tree or graph. This algorithm searches breadth wise in a tree or graph.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a General-Graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Algorithm:

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
   a. Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
   b. For each way that each rule can match the state described in E do:
      i. Apply the rule to generate a new state.
      ii. If the new state is the goal state, quit and return this state.
      iii. Otherwise add this state to the end of NODE-LIST

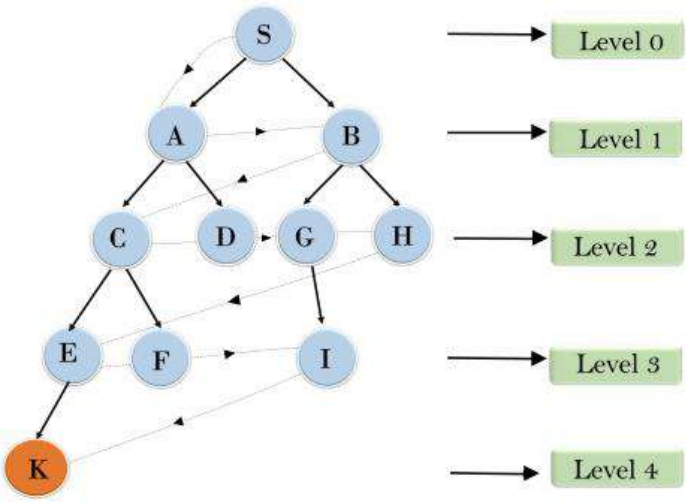
Advantages:

- BFS will provide a solution if any solution exists.
- If there is more than one solution for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

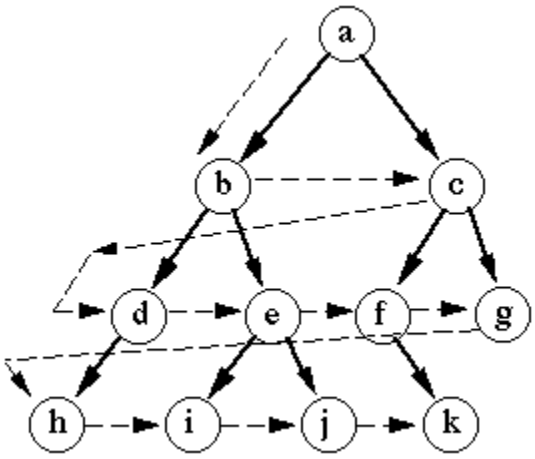
Disadvantages:

- BFS requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

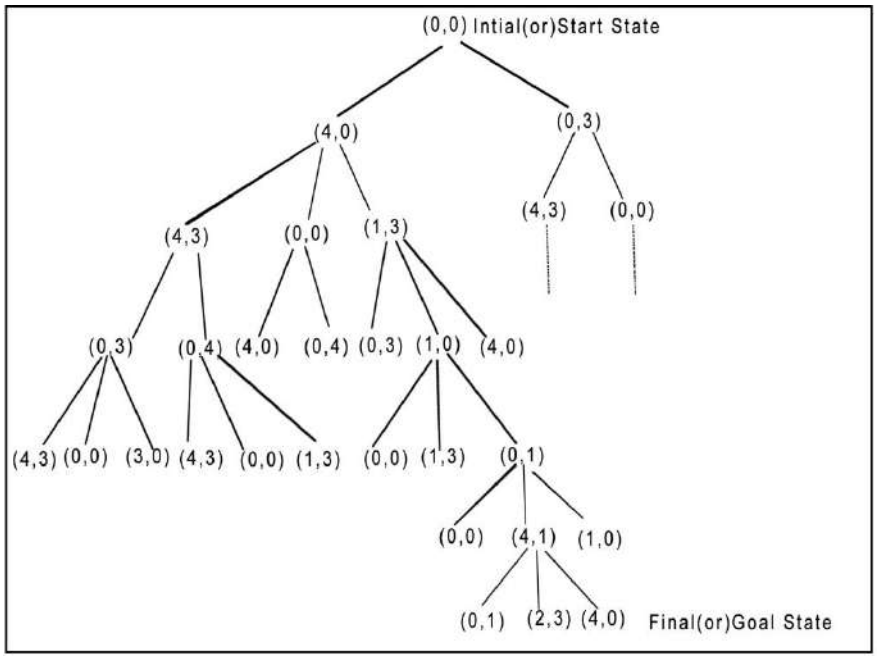
Example 1: S---> A--->B----->C--->D----->G--->H--->E----->F----->I----->K.



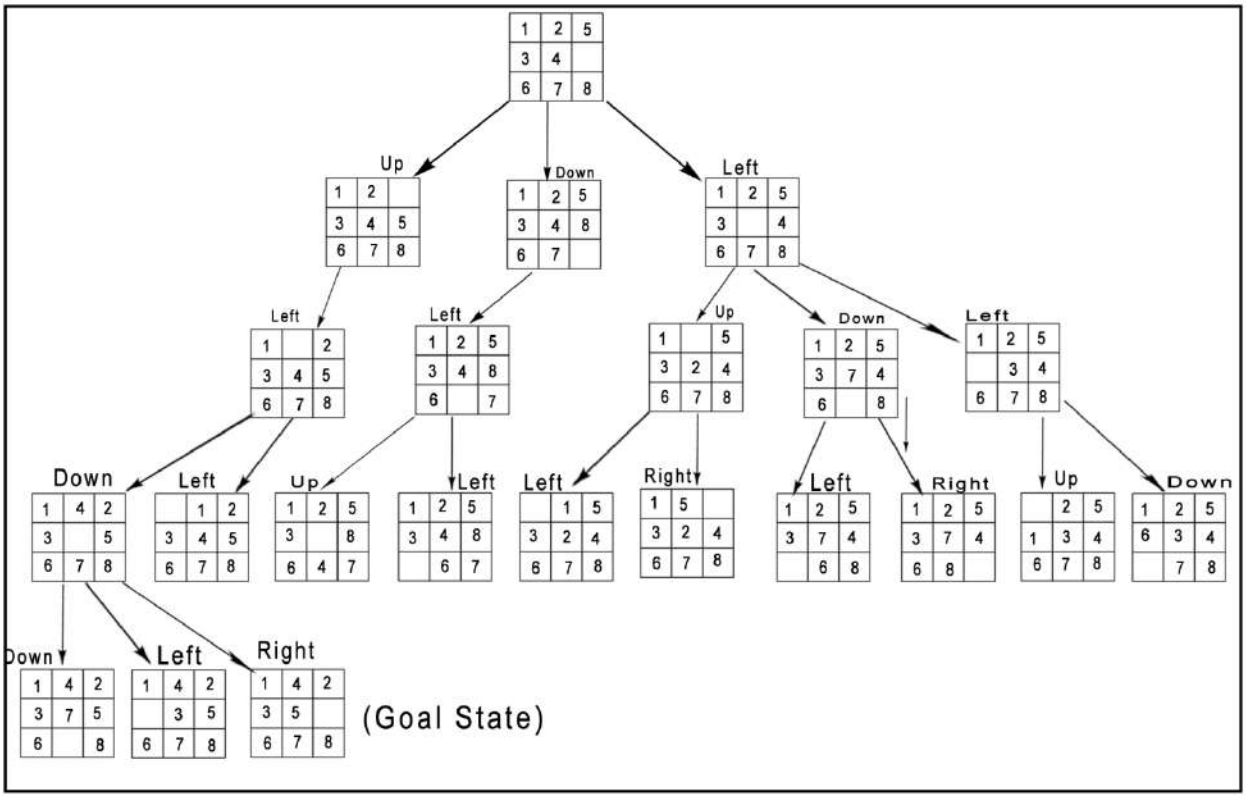
Example 2: a---> b---> c---> d---> e---> f---> g---> h---> i---> j---> k.



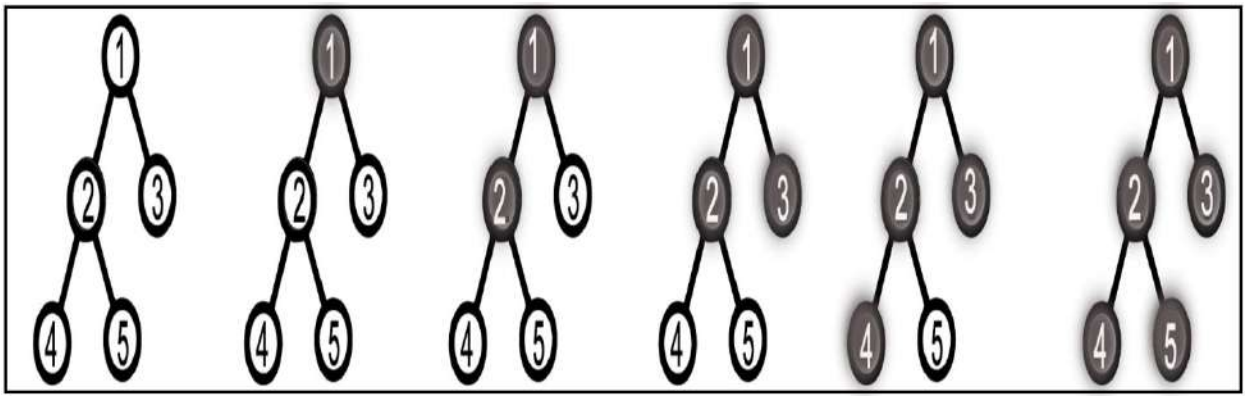
Example 3: BFS for Water Jug Problem



Example 4: 8-Puzzle Problem



Example 5:



2. Depth-First Search (DFS):

- DFS is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.

Algorithm:

1. If the initial state is a goal state, quit and return success.
2. Otherwise, loop until success or failure is signaled.

a) Generate a state, say E, and let it be the successor of the initial state. If there are no more successors, signal failure.

b) Call Depth-First Search with E as the initial state.

c) If success is returned, signal success. Otherwise continue in this loop.

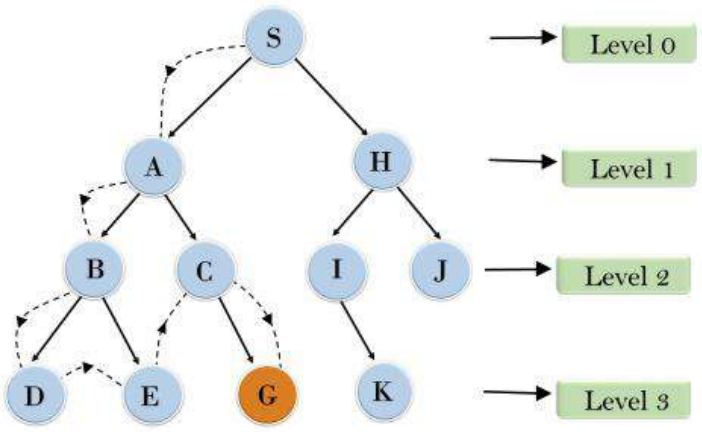
Advantages:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantages:

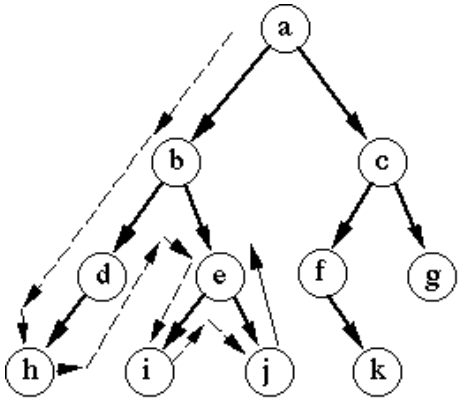
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example 1:

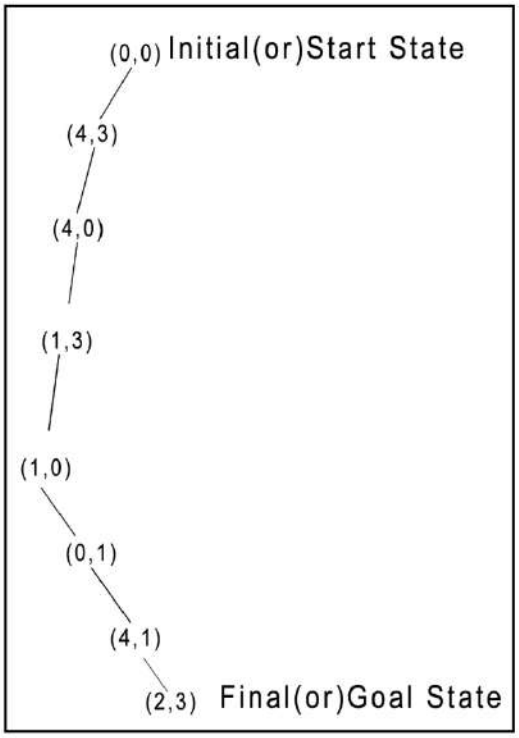


Note: It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

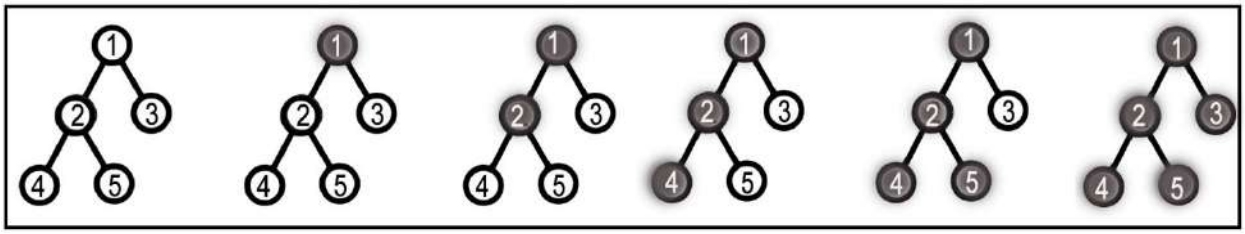
Example 2:



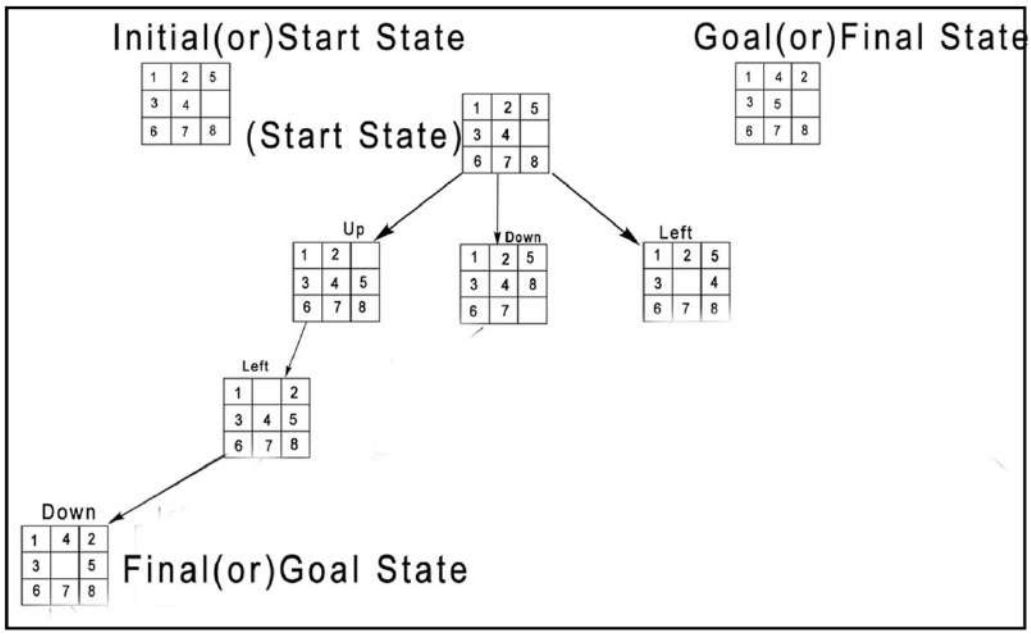
Example 3: Water Jug Problem



Example 4: 8- Puzzle Problem



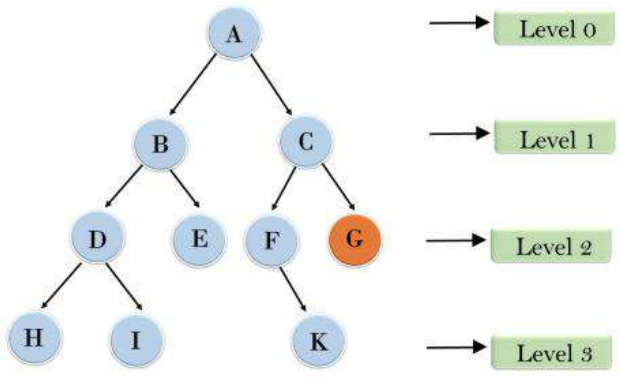
Example 5: 8- Puzzle Problem



3. Depth-First Iterative Deeping (DFID):

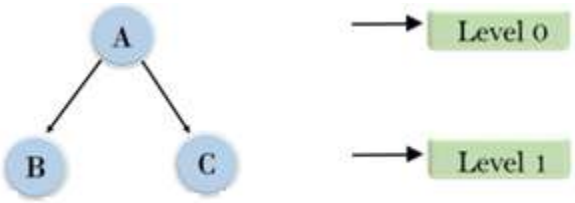
- DFID is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration 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.

Example:

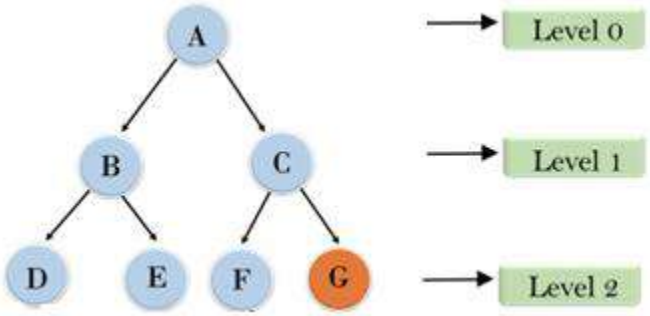


Iteration 1: A

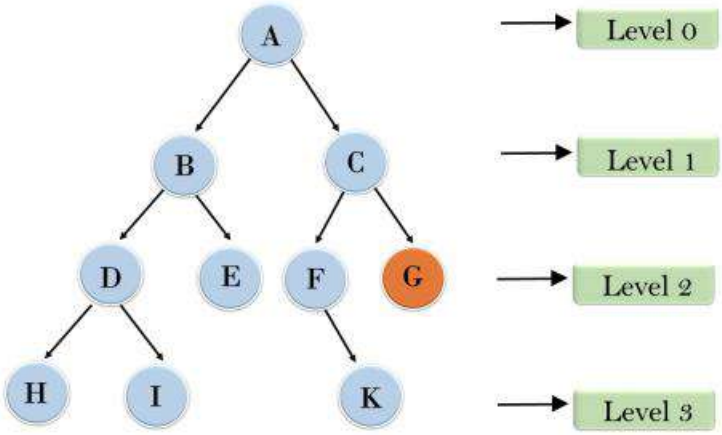
Iteration 2: A, B, C



Iteration 3: A, B, D, E, C, F, G



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



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

Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- Repeats all the work of the previous phase.

4. Bidirectional Search:

Bidirectional search is a graph search algorithm that runs 2 simultaneous searches. One search moves forward from the start state & other moves backward from the goal state & stops when the two meet in the middle. It is useful for those problems which have a single start state & single goal state.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

Example:

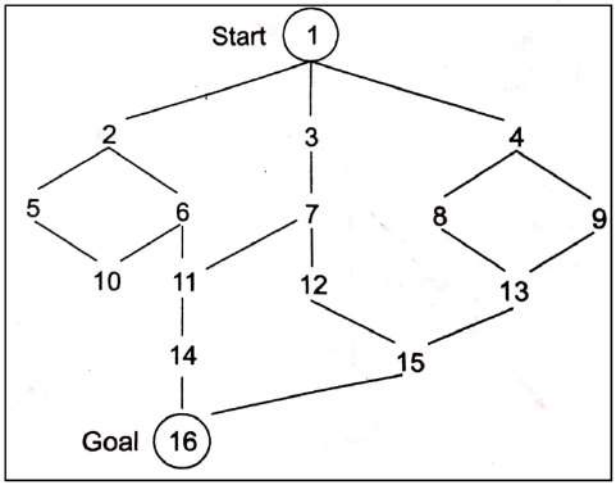
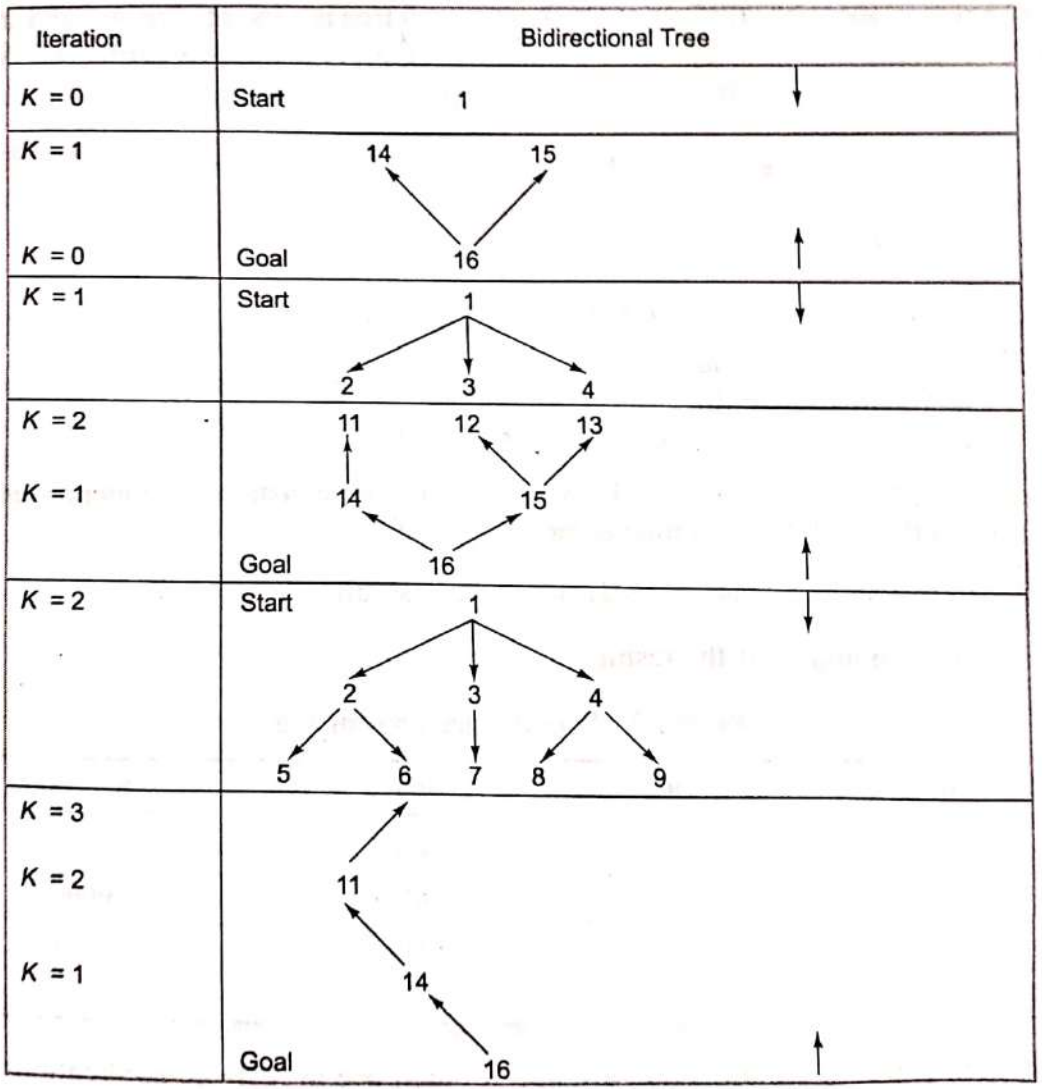


Fig: Graph to be Searched using Bidirectional Search

If match is found, then path can be traced from start to the matched state & from matched to the goal state. It should be noted that each node has link to its successors as well as to its parent. These links will be generating complete path from start to goal states.

The trace of finding path from node 1 to 16 using Bidirectional Search is as given below. The Path obtained is 1, 2, 6, 11, 14, 16.



5. **Analysis of Search Methods:**

Time Complexity: Time required by an algorithm to find a solution.

Space Complexity: Space required by an algorithm to find a solution.

Search Technique	Time	Space
BFS	$O(b^d)$	$O(b^d)$
DFS	$O(b^d)$	O(d)
DFID	$O(b^d)$	O(d)
Bidirectional	$O(b^{d/2})$	$O(b^{d/2})$

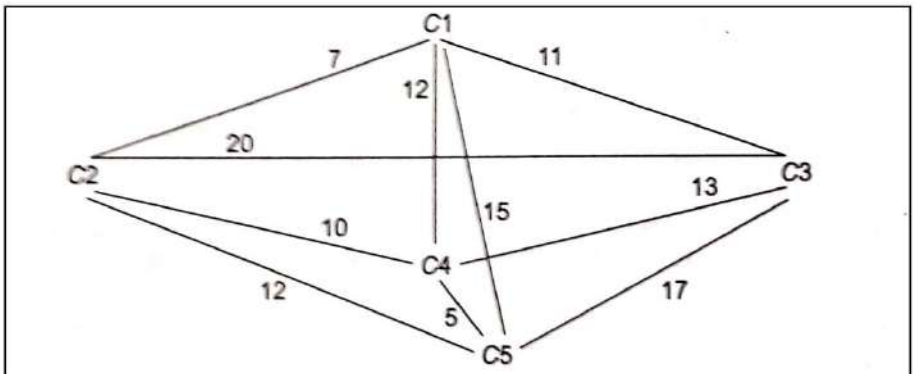
Table: Performance Comparison

<u>Travelling Salesman Problem</u> (TSP):

<u>Statement</u>: In travelling salesman problem (TSP), one is required to find the shortest route of visiting all the cities once & running back to starting point. Assume that there are 'n' cities & the distance between each pair of the cities is given.

The problem seems to be simple, but deceptive. All the possible paths of the search tree are explored & the shortest path is returned. This will require (n-1)! paths to be examined for 'n' cities.

- Start generating complete paths, keeping track of the shortest path found so far.
- Stop exploring any path as soon as its partial length becomes greater than the shortest path length found so far.



D(C1,C2) = 7; D(C1,C3) = 11; D(C1,C4) = 12; D(C1,C5) = 15; D(C2,C3) =20;
D(C2,C4) = 10; D(C2,C5) =12; D(C3,C4) =13; D(C3,C5) = 17; D(C4,C5) =5;

In this case, there will be 4!=24 possible paths. In below performance comparison, we can notice that out of 13 paths shown, 5 paths are partially evaluated.

Paths explored. Assume C1 to be the start city		Distance
1. C1 → C2 → C3 → C4 → C5 → C1 7 20 13 5 15 27 40 45 60	current best path	60 √ ×
2. C1 → C2 → C3 → C5 → C4 → C1 7 20 17 5 12 27 44 49 61		61 ×
3. C1 → C2 → C4 → C3 → C5 → C1 7 10 13 17 15 17 40 57 72		72 ×
4. C1 → C2 → C4 → C5 → C3 → C1 7 10 5 17 11 17 22 39 50	current best path, cross path at S.No 1.	50 √ ×
5. C1 → C2 → C5 → C3 → C4 → C1 7 12 17 13 12 19 36 49 61		61 ×
6. C1 → C2 → C5 → C4 → C3 → C1 7 12 5 13 11 19 24 37 48	current best path, cross path at S.No 4.	48 √
7. C1 → C3 → C2 → C4 → C5 7 20 10 5 37 47 52	(not to be expanded further) partially evaluated	52 ×
8. C1 → C3 → C2 → C5 → C4 11 20 12 5 37 49 54	(not to be expanded further) partially evaluated	54 ×
9. C1 → C3 → C4 → C2 → C5 → C1 11 13 10 12 15 24 34 46 61		61 ×
10. C1 → C3 → C4 → C5 → C2 → C1 11 13 5 12 7 24 29 41 48	same as current best path at S. No. 6.	48 √
11. C1 → C3 → C5 → C2 11 17 12 38 50	(not to be expanded further) partially evaluated	50 ×
12. C1 → C3 → C5 → C4 → C2 11 17 5 10 38 43 53	(not to be expanded further) partially evaluated	53 ×
13. C1 → C4 → C2 → C3 → C5 12 10 20 17 22 42 55	(not to be expanded further) partially evaluated	59 ×
Continue like this		

Table: Performance Comparison

HEURISTIC SEARCH TECHNIQUES

Heuristic: It is helpful in improving the efficiency of search process.

- Generate & Search
- Branch & Bound Search (Uniformed Cost Search)
- Hill Climbing
- Beam Search
- Best-First Search (A^* Algorithm)

Generate & Test:

The Generate and test strategy is the simplest of all approaches. This method generates a solution for the given problem and tests the generated solution with the required solution.

Algorithm:

Start

- Generate a possible solution
- Test if it is a goal
- If not go to start else quit

End

Advantage:

- Guarantee in finding a solution if a solution really exists.

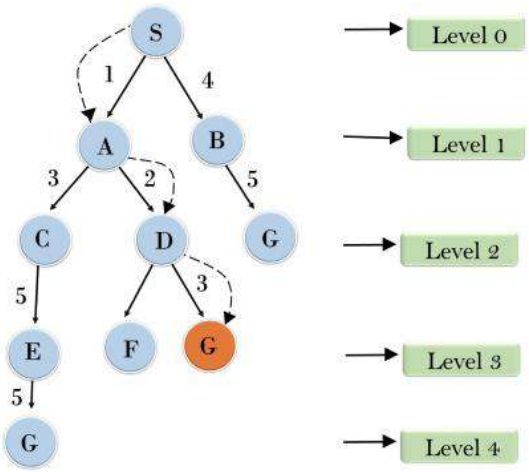
Disadvantage:

- Not suitable for the larger problems

Branch & Bound Search (Uniform Cost Search):

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. 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 from 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.

Example:



Advantage:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantage:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Hill Climbing:

- Simple Hill Climbing
- Steepest-Ascent Hill Climbing (Gradient Search)

Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.

Algorithm:

Step 1: Evaluate the initial state, if it is goal state then return success and Stop.

Step 2: Loop Until a solution is found or there is no new operator left to apply.

Step 3: Select and apply an operator to the current state.

Step 4: Check new state:

- If it is goal state, then return success and quit.
- Else if it is better than the current state then assign new state as a current state.
- Else if not better than the current state, then return to step2.

Step 5: Exit.

Steepest-Ascent Hill Climbing (Gradient Search):

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors.

Algorithm:

Step 1: Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.

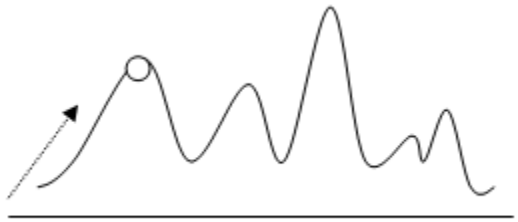
Step 2: Loop until a solution is found or the current state does not change.

a) Let SUCC be a state such that any successor of the current state will be better than it.
b) For each operator that applies to the current state:
    - Apply the new operator and generate a new state.
    - Evaluate the new state.
    - If it is goal state, then return it and quit, else compare it to the SUCC.
    - If it is better than SUCC, then set new state as SUCC.
    - If the SUCC is better than the current state, then set current state to SUCC.

Step 5: Exit.

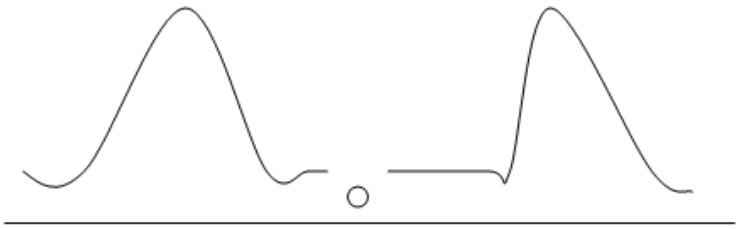
Disadvantages of Hill Climbing:

1. Local Maximum: It is a state that is better than all its neighbours but not better than some other states which are far away. From this state all moves looks to be worse. In such situation backtrack to some earlier state & try going in different direction to find a solution.



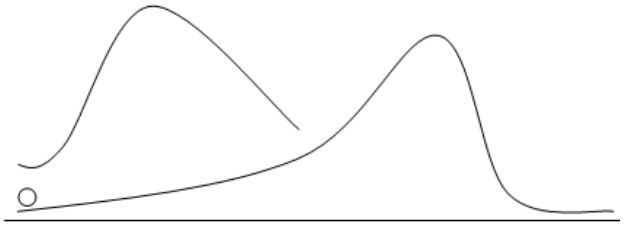
Local Maximum

2. Plateau: It is a flat area of the search space where all neighbouring states has the same value. It is not possible to determine the best direction. In such situation make a big jump to some direction & try to get to new section of the search space.



Plateau

3. Ridge: It is an area of search space that is higher than surrounding areas but that cannot be traversed by single moves in any one direction. It is a special kind of Local Maxima.



Ridge

Beam Search:

Beam Search is a heuristic search algorithm in which W number of best nodes at each level is always expanded. It progresses level-by-level & moves downward only from the best W nodes at each level. Beam Search uses BFS to build its search tree. At each level of tree it generates all its successors of the states at the current level, sorts them in order of increasing heuristic values. However it only considers W number of states at each level, whereas other nodes are ignored.

Here, W - Width of Beam Search

B - Branching Factor

There will only be W * B nodes under consideration at any depth but only W nodes will be selected.

Algorithm:

1. Node=Root_Node; Found= false
2. If Node is the goal node, then Found=true else find SUCCs of Node, if any with its estimated cost and store in OPEN list;
3. While (Found=false and not able to proceed further) do

   {

   - Sort OPEN list;
   - Select top W elements from OPEN list and put it in W_OPEN list and empty OPEN list
   - For each NODE from W_OPEN list

   {

- If NODE=Goal state then FOUND=true else find SUCCs of NODE. If any with its estimated cost and store in OPEN list;

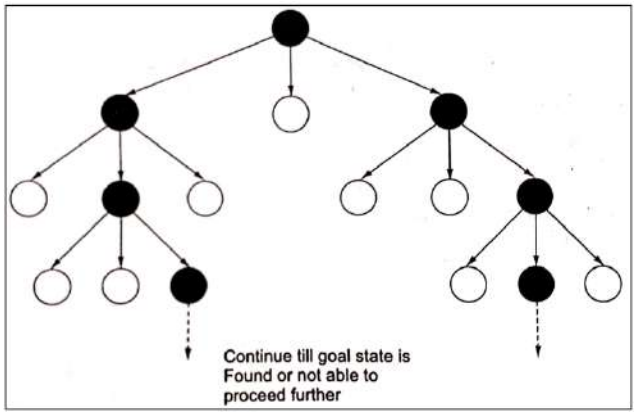
}

}

4. If FOUND=true then return Yes otherwise return No;
5. Stop

Example:

Below is the search tree generated using Beam Search Algorithm. Assume, W=2 & B=3. Here black nodes are selected based on their heuristic values for further expansion.



Best-First Search:

It is a way of combining the advantages of both Depth-First and Breadth-First Search into a single method. At each step of the best-first search process, we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic function to each of them.

In some cases we have so many options to solve but only any one of them must be solved. In AI this can be represented as OR graphs. In this among all available sub problems either of them must be solved. Hence the name OR graph.

To implement such a graph-search procedure, we will need to use two lists of nodes.

OPEN: This list contains all the nodes those have been generated and have had the heuristic function applied to them but which have not yet been examined. OPEN is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function.

CLOSED: This list contains all the nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree.

Algorithm:

1. Start with OPEN containing just the initial state
2. Until a goal is found or there are no nodes left on OPEN do:
    a) Pick the best node on OPEN
    b) Generate its successors
    c) For each successor do:
        i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.
        ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Example:

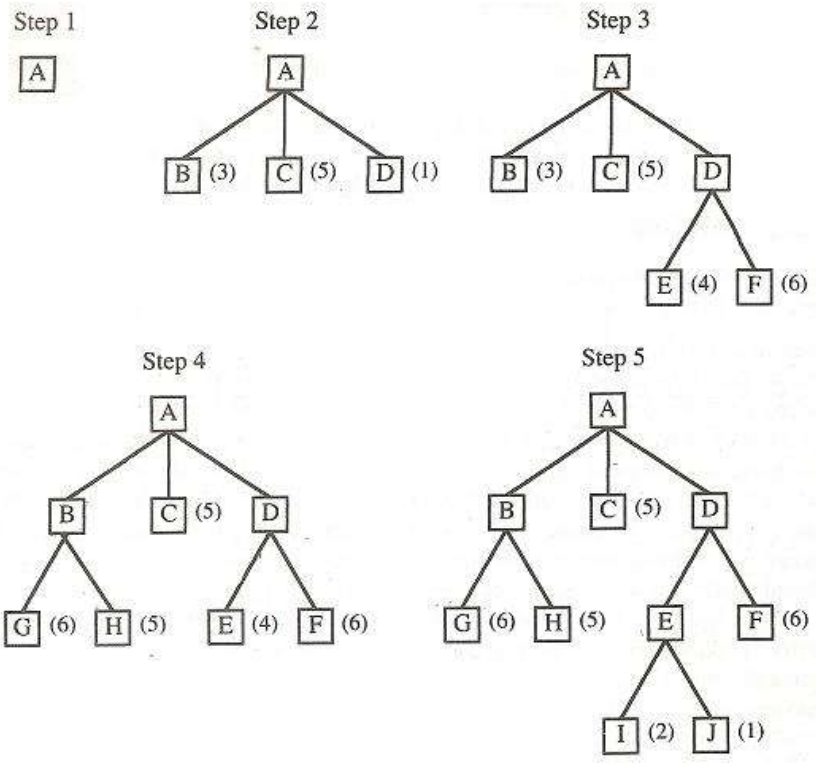
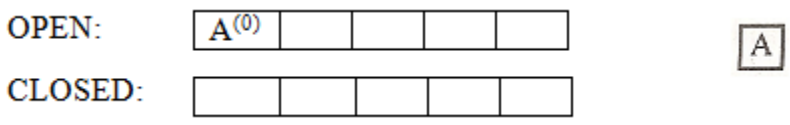
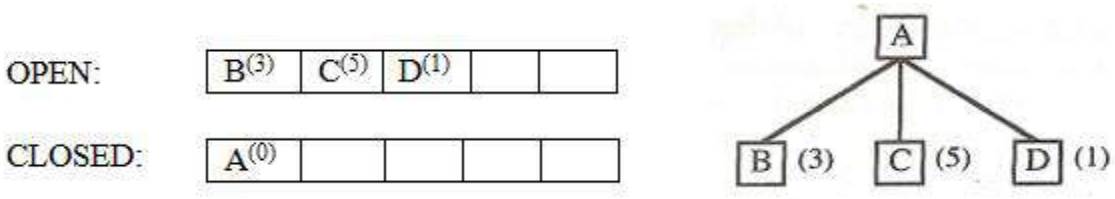


Fig: A Best-First Search

Step 1: At this level we have only one node, i.e., initial node A

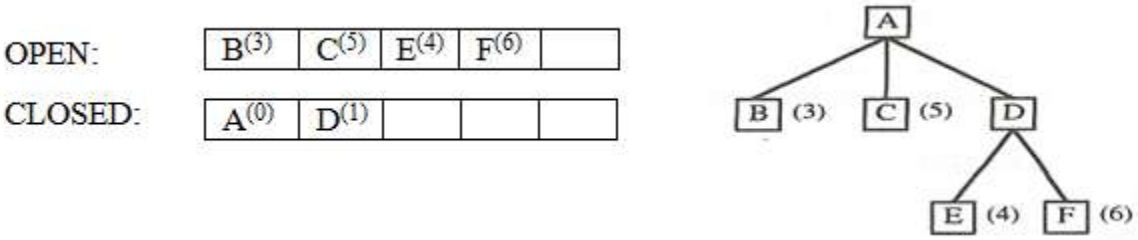


Step 2: Now we generate the successors A, three new nodes are generated namely B, C, and D with the costs of 3, 5 and 1 respectively. So these nodes are added to the OPEN list and A can be shifted to CLOSED list since it is processed.

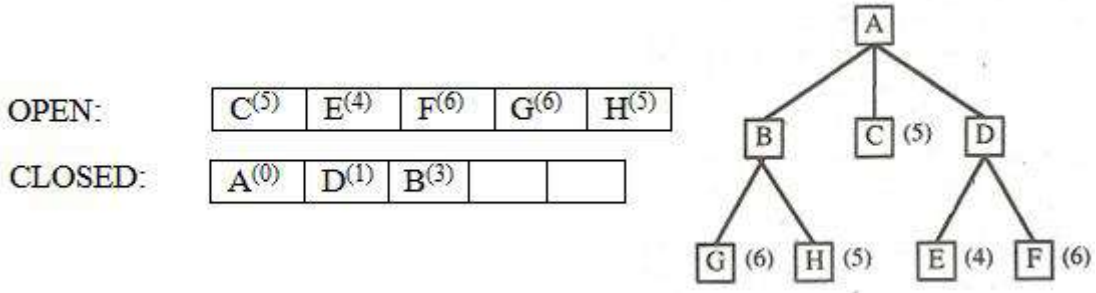


Among these three nodes D is having the least cost, and hence selected for expansion. So this node is shifted to CLOSED list.

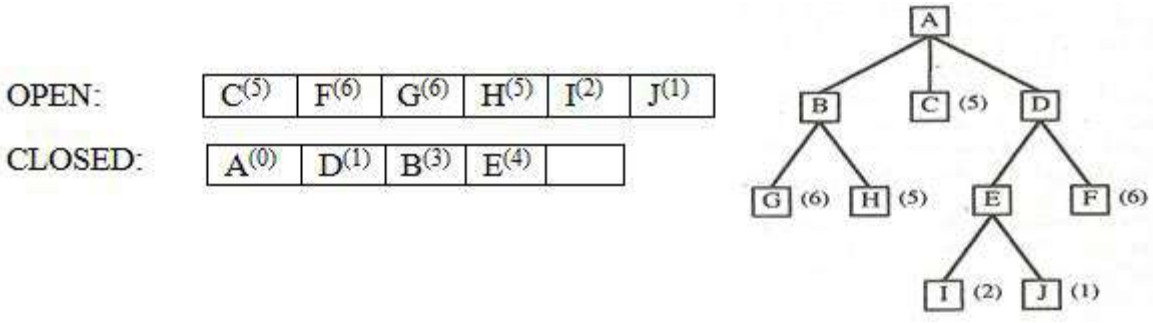
Step 3: At this stage the node D is expanded generating the new nodes E and F with the costs 4 and 6 respectively. The newly generated nodes will be added to the OPEN list. And node D will be added to CLOSED list.



Step 4: At this stage node B is expanded generating the new nodes G & H with costs 6 and 5 respectively. The newly generated nodes will be added to the OPEN list. And node B will be added to CLOSED list.



Step 5: this stage node E is expanded generating the new nodes I & J with costs 2 and 1 respectively. The newly generated nodes will be added to the OPEN list. And node E will be added to CLOSED list.



A* Algorithm:

A* is a Best First Search algorithm that finds the least cost path from initial node to one goal node (out of one or more possible goals)

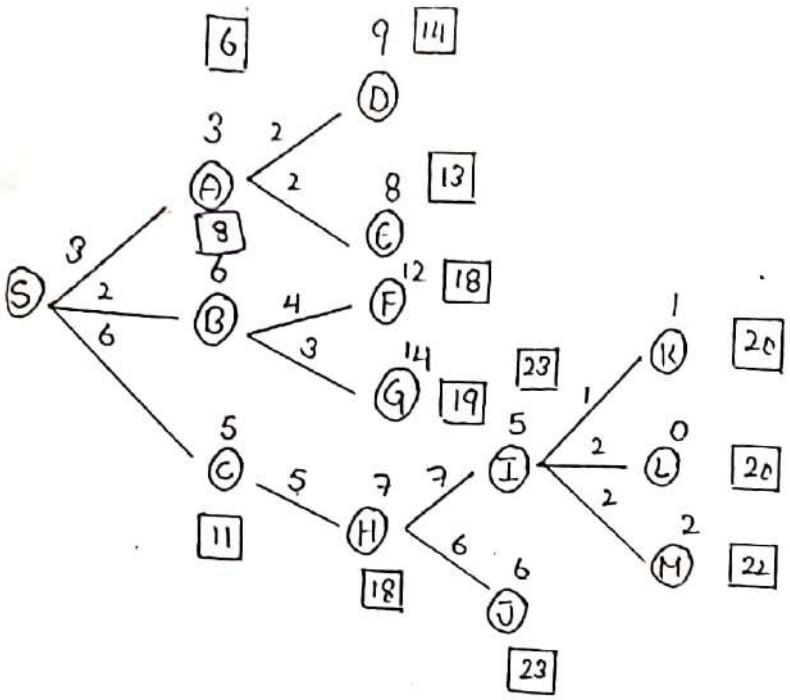
$$\mathbf{f^{l}(x) = g(x) + h^{l}(x)}$$

Where, f^{l} = estimate of cost from initial state to goal state, along the path that generated current node.

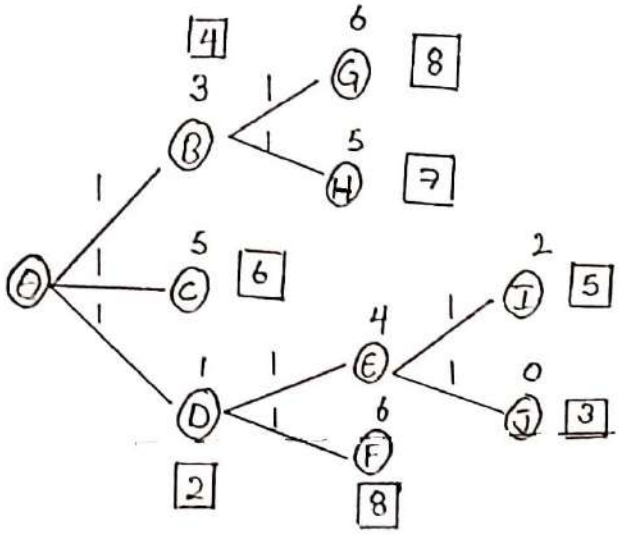
g = measure of cost getting from initial state to current node.

h^{l} = estimate of the additional cost of getting from current node to a goal state.

Example 1:



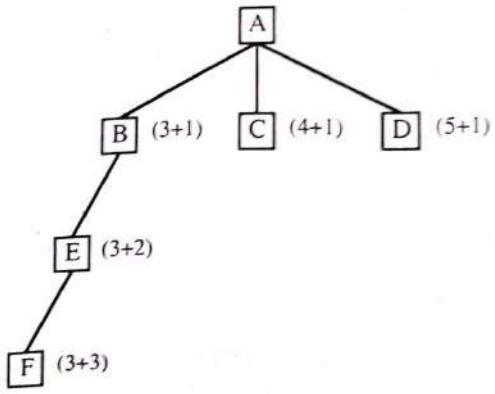
Example 2: Assuming all the values (arcs) as '1'



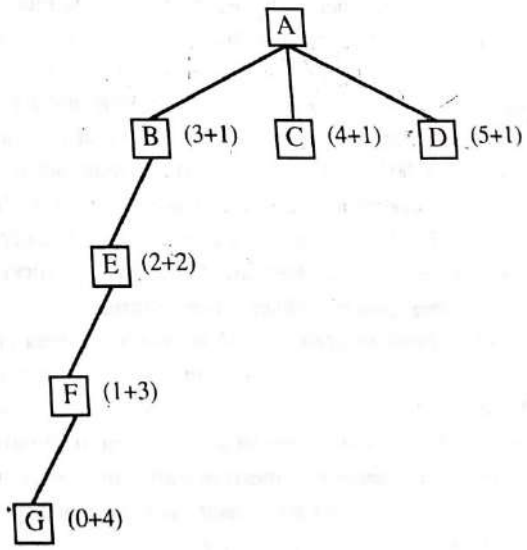
Optimal Solution by A* Algorithm:

- Underestimation
- Overestimation

Underestimation: From the below start node A is expanded to B, C & D with f values as 4, 5 & 6 respectively. Here, we are assuming that the cost of all arcs is 1 for the sake of simplicity. Note that node B has minimum f value, so expand this node to E which has f value as 5. Since f value of C is also 5, we resolve in favour of E, the path currently we are expanding. Now node E is expanded to node F with f value as 6. Clearly, expansion of a node F is stopped as f value of C is not the smallest. Thus, we see that by underestimating heuristic value, we have wasted some effort by eventually discovered that B was farther away than we thought. Now we go back & try another path & will find the optimal path.



Overestimation: Here we are overestimating heuristic value of each node in the graph/tree. We expand B to E, E to F & F to G for a solution path of length 4. But assume that there is direct path from D to a solution giving a path of length 2 as h value of D is also overestimated. We will never find it because of overestimating h(D). We may find some other worse solution without ever expanding D. So, by overestimating h, we cannot be guaranteed to find the shortest path.



Admissibility of A*:

A search algorithm is admissible, if for any graph, it always terminates in an optional path from start state to goal state, if path exists. We have seen earlier that if heuristic function 'h' underestimates the actual value from current state to goal state, then it bounds to give an optimal solution & hence is called admissible function. So, we can say that A* always terminates with the optimal path in case h is an admissible heuristic function.

ITERATIVE-DEEPING A*

IDA* is a combination of the DFID & A* algorithm. Here the successive iterations are corresponding to increasing values of the total cost of a path rather than increasing depth of the search. Algorithm works as follows:

- For each iteration, perform a DFS pruning off a branch when its total cost (g+h) exceeds a given threshold.
- The initial threshold starts at the estimate cost of the start state & increases for each iteration of the algorithm.
- The threshold used for the next iteration is the minimum cost of all values exceeded the current threshold.
- These steps are repeated till we find a goal state.

Let us consider as example to illustrate the working IDA* Algorithm as shown below. Initially, the threshold value is the estimated cost of the start node. In the first iteration, threshold=5. Now we generate all the successors of start node & compute their estimated values as 6, 8, 4, 8 & 9. The successors having values greater than 5 are to be pruned. Now for next iteration, we consider the threshold to be the minimum of the pruned nodes value, that is, threshold=6 & the node with 6 value along with node with value 4 are retained for further expansion.

Advantages:

- Simpler to implement over A* is that it does not use Open & Closed lists
- Finds solution of least cost or optimal solution
- Uses less space than A*

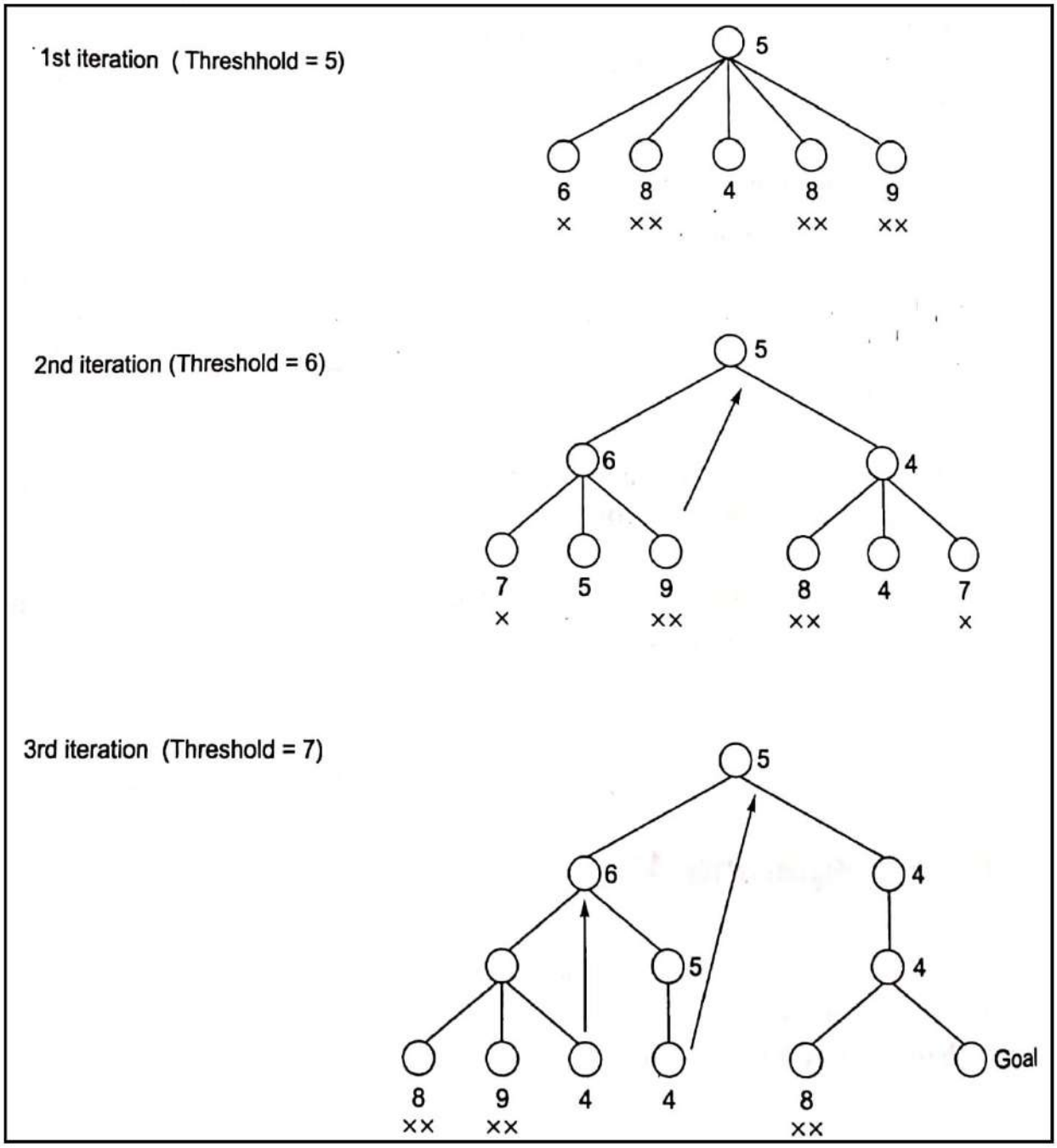


Fig: Working of IDA*

CONSTRAINT SATISFACTION

Many problems in AI can be viewed as problems of constraint satisfaction in which the goal is to discover some problem state that satisfies a given set of constraints instead of finding a optimal path to the solution. Such problems are called Constraint Satisfaction (CS) problems. Constraint satisfaction is a two step process.

- First, constraints are discovered and propagated as far as possible throughout the system. Then, if there is still not a solution then the search begins. A guess about something is made and added as a new constraint
- Propagation then occurs with this new constraint.

Start State: all the variables are unassigned.

Goal State: all the variables are assigned which satisfy constraints.

Rules:

- Values of variables should be from 0 to 9
- No 2 variables have same value

Algorithm:

1. Propagate available constraints. To do this, first set OPEN to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty:
   a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.
   b) If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to OPEN all objects that share any constraints with OB
   c) Remove OB from OPEN.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, then return failure.
4. If neither of the above occurs, then it is necessary to make a guess at in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated
   a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
   b) Recursively invoke constrain satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

Example:

```
  S E N D
+ M O R E
---------
M O N E Y
---------
```

Let us consider M=1, because by adding any 2 single digit number, at maximum we get one as carry.

	S	E	N	D
+	1	O	R	E
1	c3	c2	c1	
M	O	N	E	Y

Assume that S=8 (or) 9, S + M = 0 (or) S + M + C3 = 0

If S = 9, S + M = 9 + 1 =10 (with no carry)

If S = 8, S + M + C3 = 8 + 1 + 1 = 10 (with carry). So, we get O value as 0.

Therefore, M = 1, S = 9 & O = 0.

	9	E	N	D
+	1	O	R	E
1	c3	c2	c1	
1	0	N	E	Y

So, here E + 0 = N. Then E =N (It is not possible because no 2 variables should have same value).

E + O + c2 = N

E + 0 + 1 = N which gives E + 1 = N

Estimate E value from the remaining possible numbers i.e.., 2, 3, 4, 5, 6, 7, 8. So, from our estimation the E & N values satisfies at E = 5. So, E + 1 = 5 + 1 = 6 i.e.., N = 6 (E + 1 = N)

Therefore, M = 1, S = 9, O = 0, E = 5 & N = 6.

	9	5	N	D
+	1	0	R	E
1	c3	1	c1	
1	0	6	E	Y

So, here N + R + C1 = E. We already know that E = 5. So, the E value is satisfied by taking R = 8.

6 + 8 + 1 = 15.

Therefore, M = 1, S = 9, O = 0, E = 5, N = 6 & R = 8

	9	5	6	D
+	1	0	8	E
1	c3	1	1	
1	0	6	5	Y

.

Here, D + E = Y. It has to satisfy C1 = 1, so that D = 7.

Then, 7 + 5 = 12. So, Y = 2.

Final Values are M = 1, S = 9, O = 0, E = 5, N = 6, R = 8, D = 7 & Y = 2.

By substituting all the above values,

```
  S E N D          9 5 6 7
+ M O R E        + 1 0 8 5
---------        ---------
M O N E Y        1 c3 1 1
---------        ---------
                 1 0 6 5 2
```

Some other Examples of Constraint Satisfaction are as below,

Example 2:

```
  B A S E          7 4 8 3
+ B A L L        +7 4 5 5
---------        ---------
G A M E S        1 4 9 3 8
```

Example 3:

```
  R O A D S        9 6 2 3 3
+ C R O S S      + 6 2 5 1 3
-----------      -----------
D A N G E R      1 5 8 7 4 6
```

Example 4:

```
  D O N A L D        5 2 6 4 8 5
+ G E R A L D      + 1 9 7 4 8 5
-------------      -------------
R O B E R T        7 2 3 9 7 0
```

Example 5:

```
  T W O          7 3 4
+ T W O        + 7 3 4
---------      -------
F O U R        1 4 6 8
```

Example 6:

```
  L O G I C          9 0 4 5 2
+ L O G I C        + 9 0 4 5 2
-----------        -----------
P R O L O G        1 8 0 9 0 4
```

2. PROBLEM REDUCTION AND GAME PLAYING

INTRODUCTION

An effective way of solving a complex problem is to reduce it to simpler parts & solve each part separately. Problem Reduction enables us to obtain an elegant solution to the problem. The structure called AND-OR graph (or tree) is useful for representing the solution of complicated problems. The decomposition of a complex problem generates arcs which we call AND arcs. One AND arc may point to any number of successors, all of which must be solved.

PROBLEM REDUCTION (AND-OR Graph) (or) AO* ALGORITHM

In real-world applications, complicated problems can be divided into simpler sub-problems. The solution of each sub-problem may then be combined to obtain the final solution. A given problem may be solved in a number of ways. An AND-OR Graph provides a simple representation of a complex problem that leads to better understanding. For instance, if you wish to own a cellular phone then it may be possible that either someone gifts one to you or you earn money & buy one for yourself. The AND-OR graph which depicts these possibilities is as shown below

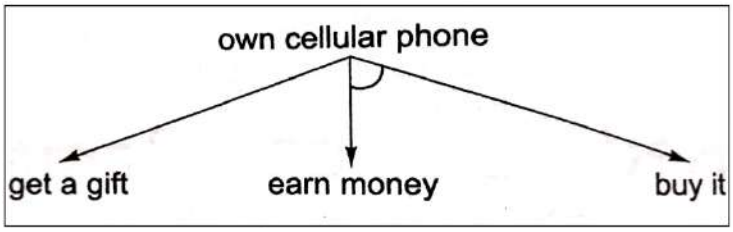
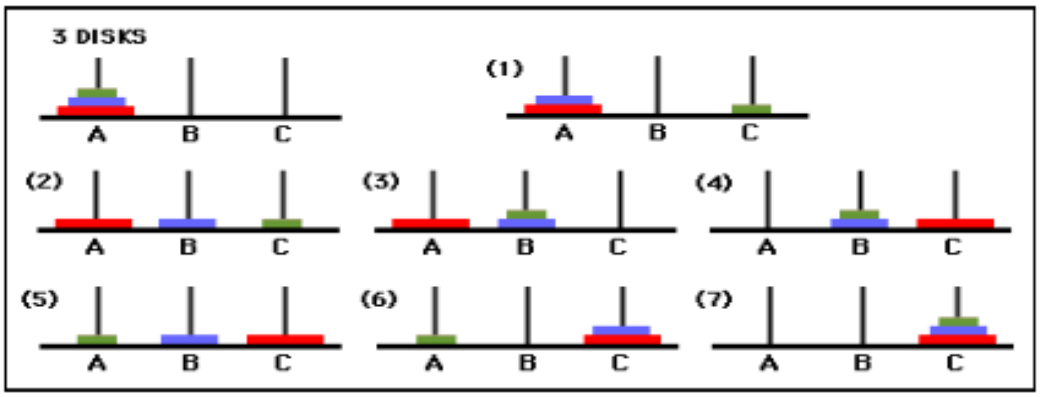


Fig: A simple AND-OR Graph

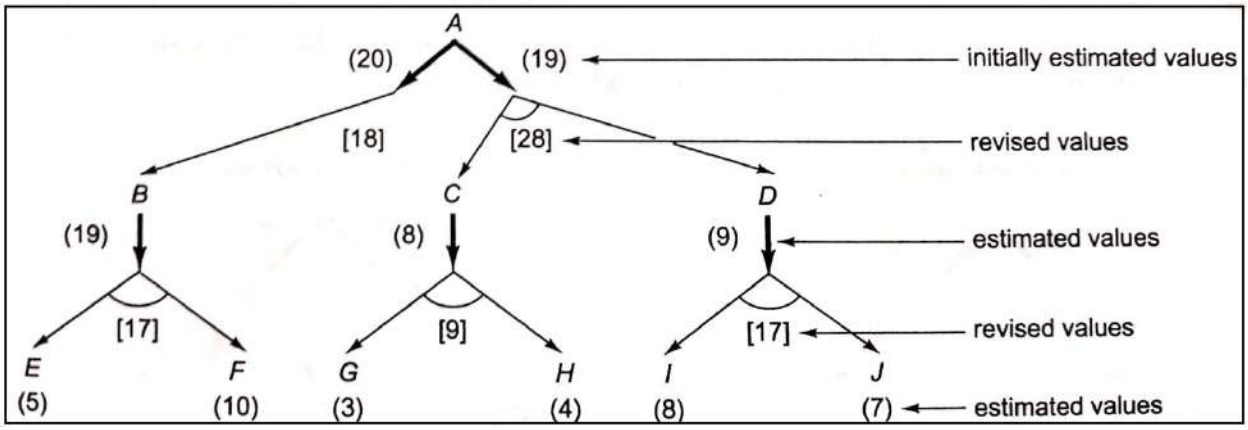
Let us consider a problem known as Towers of Hanoi to illustrate the need of problem reduction concept. The Tower of Hanoi is a mathematical game or puzzle. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks being stacked in descending order of their sizes, with the largest at the bottom of the stack & the smallest at the top thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod by using the following rules:

- Only one disk can be moved at a time.
- Each move consists of taking the uppermost disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No larger disk may be placed on top of a smaller disk.

With 3 disks, the puzzle can be solved in 7 moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks which is as follows



Consider an AND-OR graph where each arc with a single successor has a cost of 1. Let us assume that the numbers listed in parenthesis ( ) denote the estimated costs, while the numbers in the square brackets [ ] represent the revised costs of the path. Thick lines indicate paths from a given node.



Let us begin search from start node A & compute the heuristic values for each of its successors, say B & (C,D) as 19 and (8,9) respectively. The estimated cost of paths from A to B is 20 (19 + cost of one arc from A to B) & that from A to (C,D) is 19 (8 + 9 + cost of 2 arcs, A to C & A to D). The path from A to (C,D) seems to be better than that from A to B. So, we expand this AND path by extending C to (G,H) and D to (I,J). Now, heuristic values of G, H, I & J are 3, 4, 8 & 7 respectively which lead to revised costs of C & D as 9 & 17 respectively. These values are then propagated up & the revised costs of path from A to (C,D) is calculated as 28 (9 + 17+ Cost of arc A to C & A to D).

Note that the revised cost of this path is now 28 instead of the earlier estimation of 19. Thus, this path is no longer the best path now. Therefore, choose the path from A to B for expansion. After expansion we see that the heuristic value of node B is 17 thus making the cost of the path from A to B equal to 18. This path is the best so far. Therefore, we further explore the path from A to B. The process continues until either a solution is found (or) all paths lead to dead ends, indicating that there is no solution.

Algorithm for AND-OR Graphs:

1. Initialize graph with start node.

2. While (start node is not labeled as solved or unsolved through all paths)
   {
   - Traverse the graph along the best path & expand all the unexpanded nodes on it.
   - If node is terminal & heuristic value of the node is 0, label it as solved else label it as unsolved & propagate the status up to the start node.
   - If node is non terminal, add its successors with the heuristic values in the graph.
   - Revise the cost of the expanded node & propagate this change along the path till the start node.
   - Choose the current best path.

   }
3. If (start node = solved), the leaf nodes of the best path from root are the solution nodes, else no solution exists.
4. Stop.

GAME PLAYING

Game playing is a major topic of AI as it requires intelligence & has certain well-defined states & rules. A game is defined as a sequence of choices where each choice is made from a number of discrete alternatives. Each sequence ends in a certain outcome & every outcome has a definite value for opening player.

Games can be classified into 2 types

- Perfect Information Games
- Imperfect Information Games

Perfect Information Games are those in which both the players have access to the same information about the game in progress. Examples are Tic-Tac-Toe, Chess etc.

Imperfect Information Games are those in which players do not have access to the complete information about the game. Examples are Cards, Dice etc.

Game Problem versus State Space Problem:

In state space problems, we have a start state, intermediate states, rules or operators & a goal state. In game problems also we have a start state, legal moves & winning positions (goals).

State Space Problems	Game Problems
States	Legal board positions
Rules	Legal moves
Goals	Winning Positions

Table: Comparisons between State Space Problems & Game Problems

A Game begins from a specified initial state & ends in a position that can be declared a win for one, a loss for the other or possible a draw. A Game Tree is an explicit representation of all possible plays of the game. The root node is an initial position of the game. Its successors are the positions that the first player can reach in one move; their successors are the positions resulting from the second player's move & so on. Terminal or leaf nodes are represented by WIN, LOSS (or) DRAW.

Game theory is based on the philosophy of minimizing the maximum possible loss and maximizing the minimum gain. In game playing involving computers, one player is assumed to be the computer, while the other is a human. During a game, 2 types of nodes are encountered, namely MAX & MIN. The MAX node will try to maximize its own game, while minimizing the opponent's (MIN) game. Either of the 2 players MAX & MIN can play as first player.

Status Labelling Procedure in Game Tree:

Status labelling procedure for a node with WIN, LOSS or DRAW in case of a game tree is as follows:

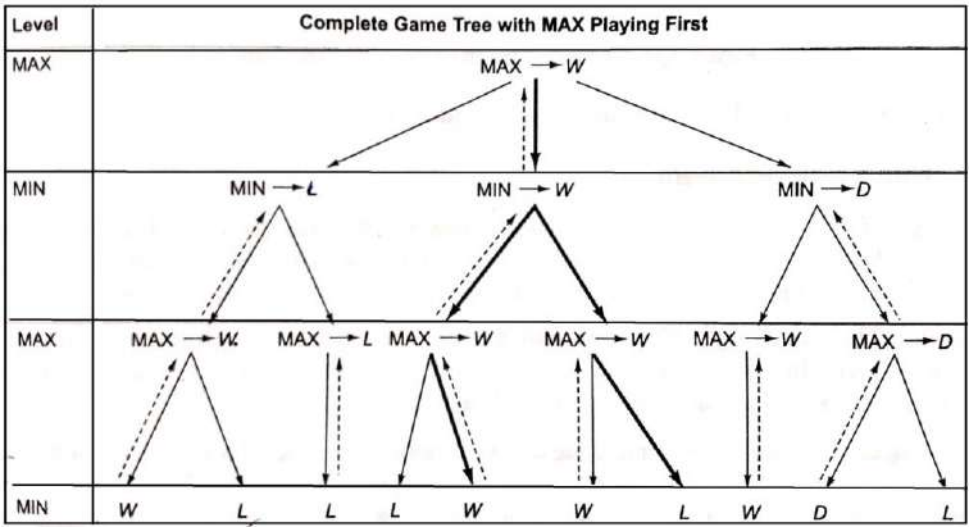
- If j is a non-terminal MAX node, then

$$\text{STATUS}(j) = \begin{cases} \text{WIN}, & \text{if any of } j\text{'s successor is a WIN} \\ \text{LOSS}, & \text{if all } j\text{'s successor are LOSS} \\ \text{DRAW}, & \text{if any of } j\text{'s successor is a DRAW and none is WIN} \end{cases}$$

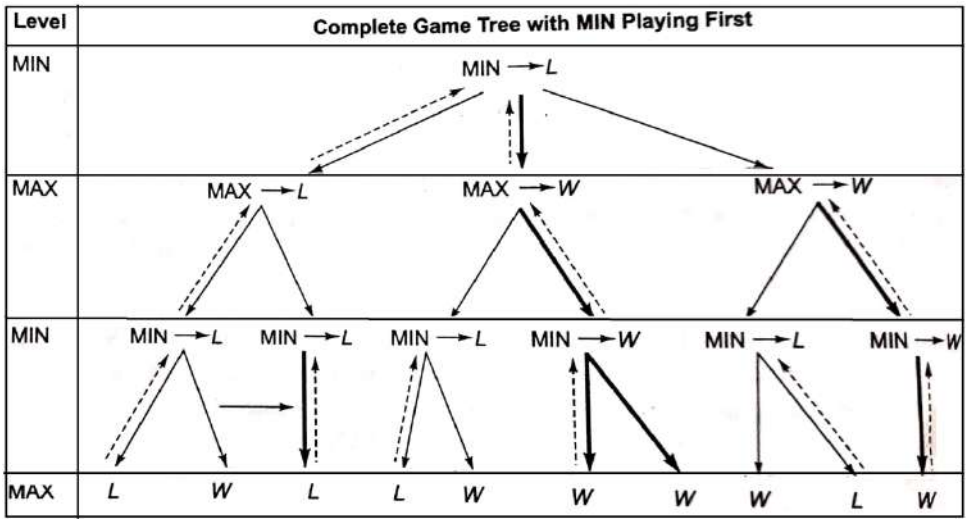
- If j is a non-terminal MIN node, then

$$\text{STATUS}(j) = \begin{cases} \text{WIN}, & \text{if all } j\text{'s successor are WIN} \\ \text{LOSS}, & \text{if any of } j\text{'s successor is a LOSS} \\ \text{DRAW}, & \text{if any of } j\text{'s successor is a DRAW and none is LOSS} \end{cases}$$

A Game Tree in which MAX Plays First:



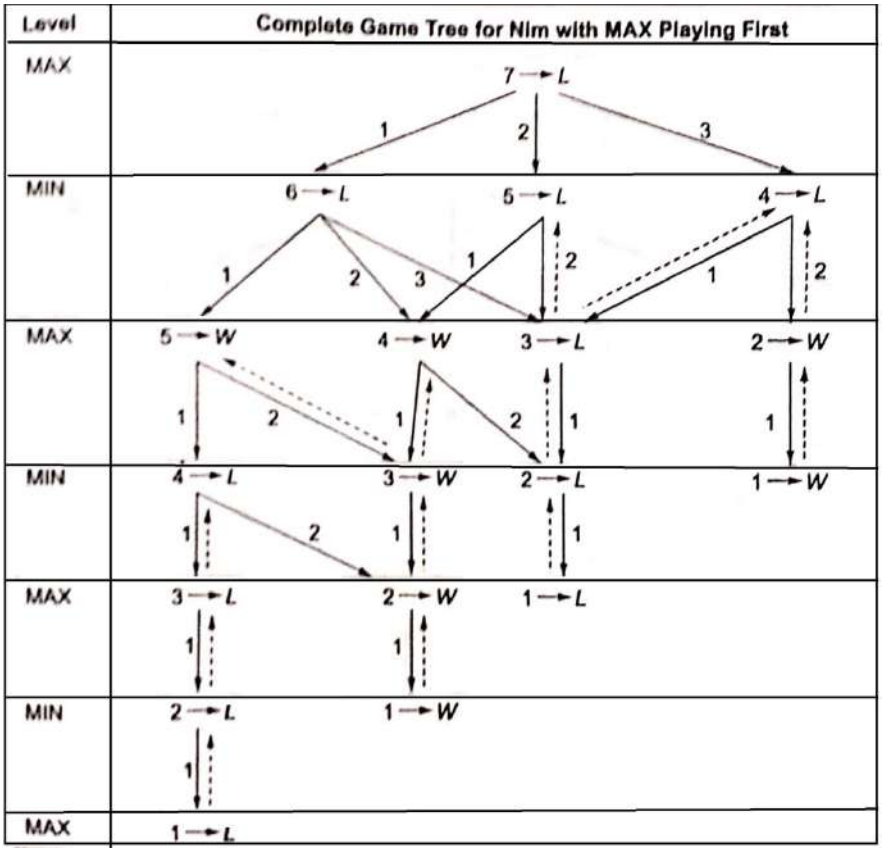
A Game Tree in which MIN Plays First:



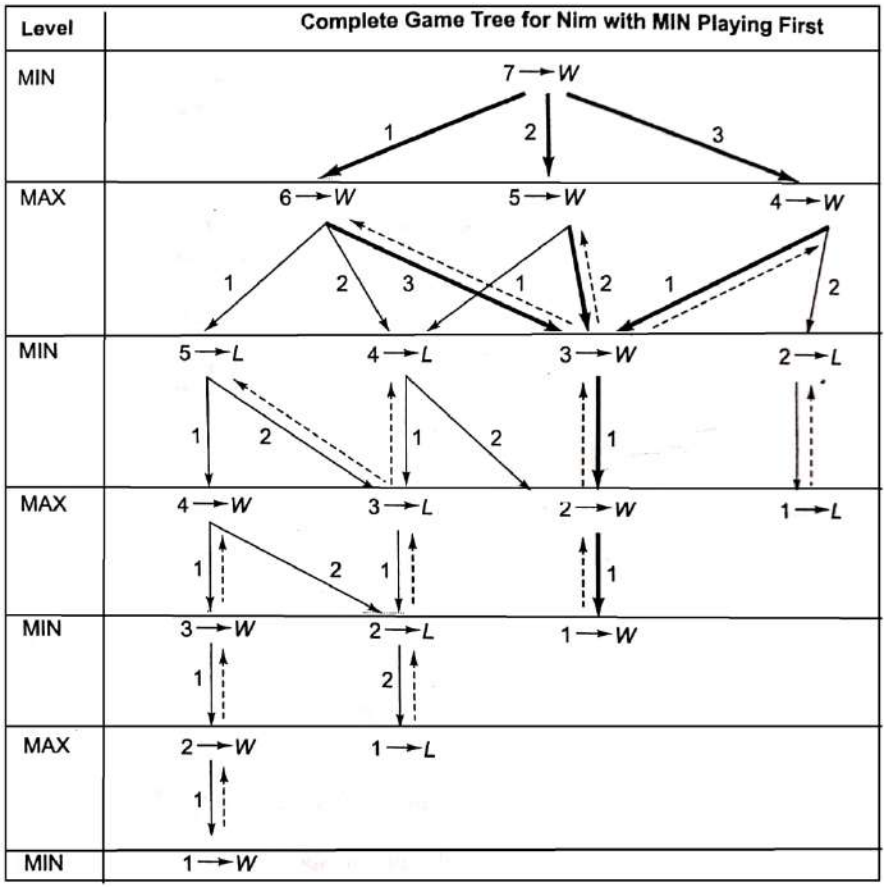
Nim Game Problem: Nim game was developed by Charles L. Bouton of Harward University at china in 1901.

The Game: There is a single pile of matchsticks (>1) and two players. Moves are made by the players alternately. In a move, each player can pick up a maximum of half the number or matchsticks in the pile. Whoever takes the last match sticks loses. Let us consider for explaining the concept with single pile of 7 matchsticks

The convention used for drawing a game tree is that each node contains the total number of sticks in the pile and is labeled as W or L in accordance with the status labeling procedure. The player who has to pick up the last sticks loses. If a single stick is left at the MAX level then as a rule of the game, MAX node is assigned the status L, whereas if one stick is left at the MIN level, then W is assigned to MIN node as MAX wins. The label L or W have been assigned from MAX's point of view at leaf nodes. Arcs carry the number of sticks to be removed; Dotted lines show the propagation of status. The complete game tree for Nim with MAX playing first is as follows



The complete game tree for Nim with MIN playing first is as follows



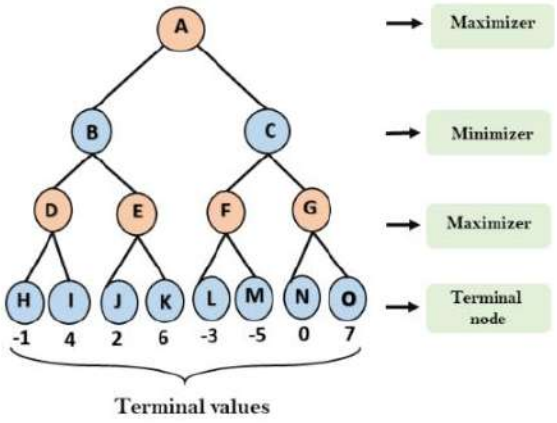
MINIMAX PROCEDURE

MINIMAX procedure is a recursive or backtracking algorithm which is used in decision-making and game theory. This is mostly used for game playing in AI, such as Chess, Checkers, tic-tac-toe & various tow-players game. In this 2 players play the game, one is called MAX and other is called MIN, where MAX will select the maximized value and MIN will select the minimized value. The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree. The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Working of MINIMAX Algorithm:

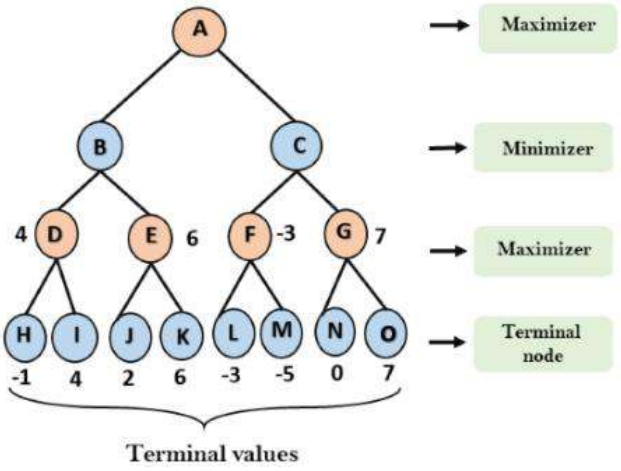
- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those values and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step 1: In the first step, the algorithm generates the entire game-tree. Let's take A is the initial state of the tree. Suppose maximizer takes first turn and minimizer will take next turn.



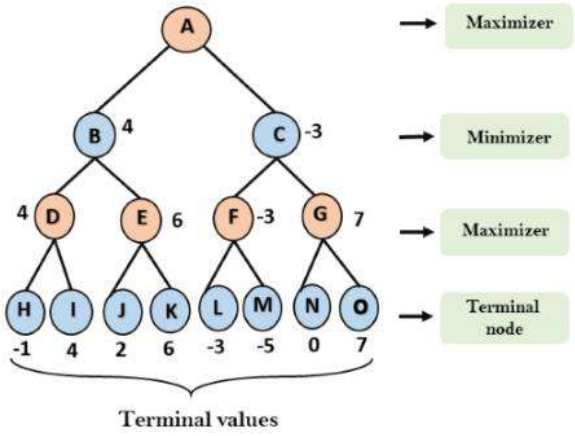
Step 2: Now, first we find the utilities value for the Maximizer. It will find the maximum among the all.

- For node D max(-1,4) = 4
- For Node E max(2, 6) = 6
- For Node F max(-3,-5) = -3
- For node G max(0, 7) = 7



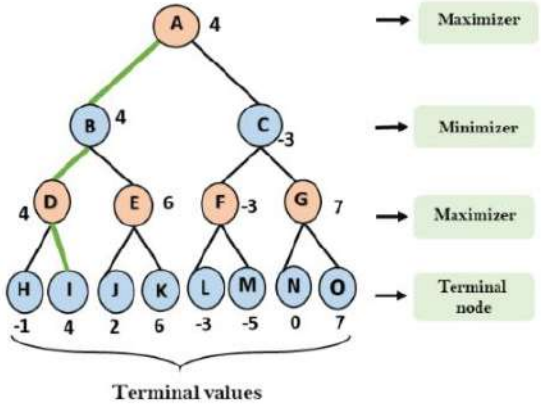
Step 3: In the next step, it's a turn for minimizer,

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3) = 4



Example 2:

Time Complexity: As it performs DFS for the game-tree, the time complexity of MINIMAX algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.

Space Complexity: Space complexity of MINIMAX algorithm is also similar to DFS which is **O(bm).**

Limitations of MINIMAX Algorithm:

- Gets really slow for complex games such as Chess, go, etc.
- This type of games has a huge branching factor, and the player has lots of choices to decide.

Note: This limitation of the MINIMAX algorithm can be improved from alpha-beta pruning.

Example 2:

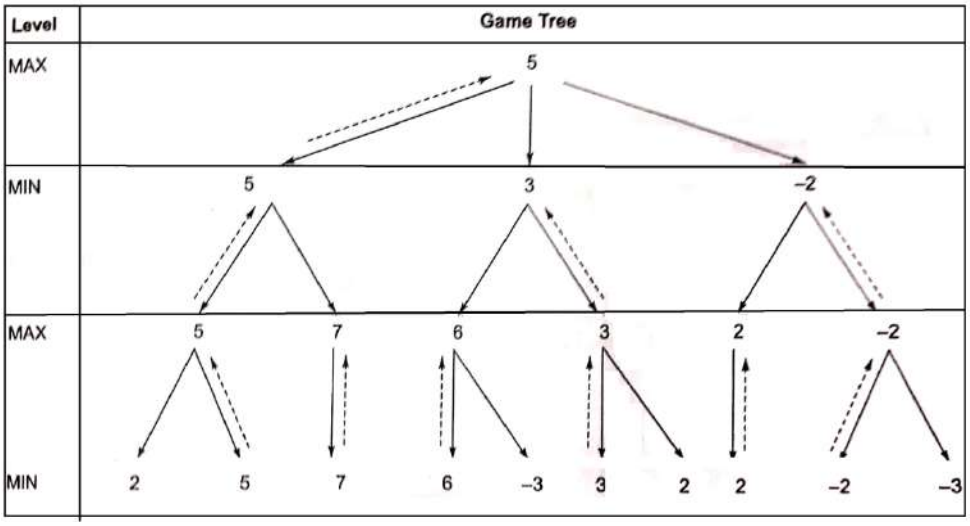


Fig: A Game Tree generated using MINIMAX Procedure

ALPHA-BETA PRUNING

Alpha-beta pruning is a modified version of the MINIMAX algorithm. It is an optimization technique for the MINIMAX algorithm. There is a technique by which without checking each node of the game tree we can compute the correct MINIMAX decision, and this technique is called Pruning. This involves two threshold parameter Alpha and beta for future expansion, so it is called ALPHA-BETA pruning.

- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
    a) Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

b) Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

- Pruning of nodes makes the algorithm fast.

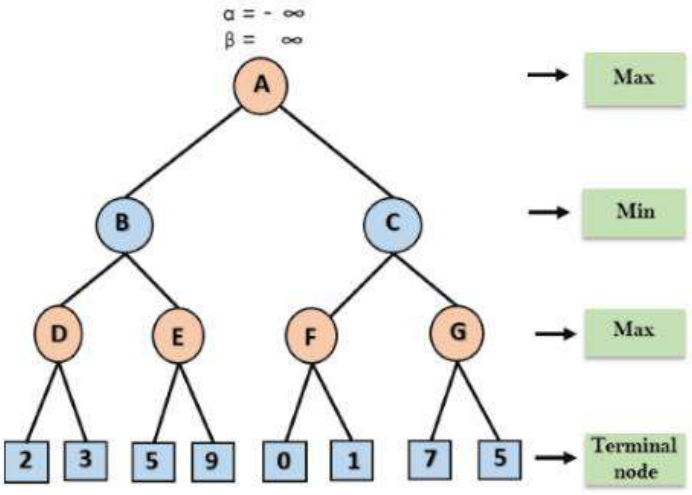
Condition for Alpha-beta pruning: **α>=β**

Key points:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

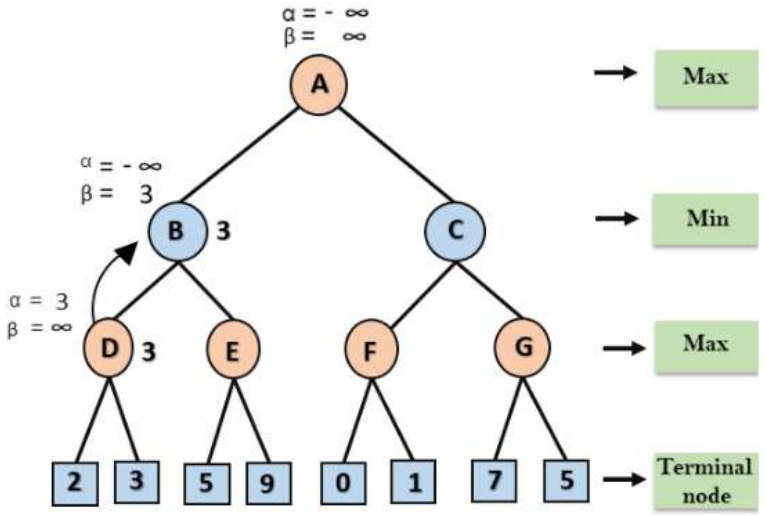
Working of Alpha-beta pruning:

Step 1**:** At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



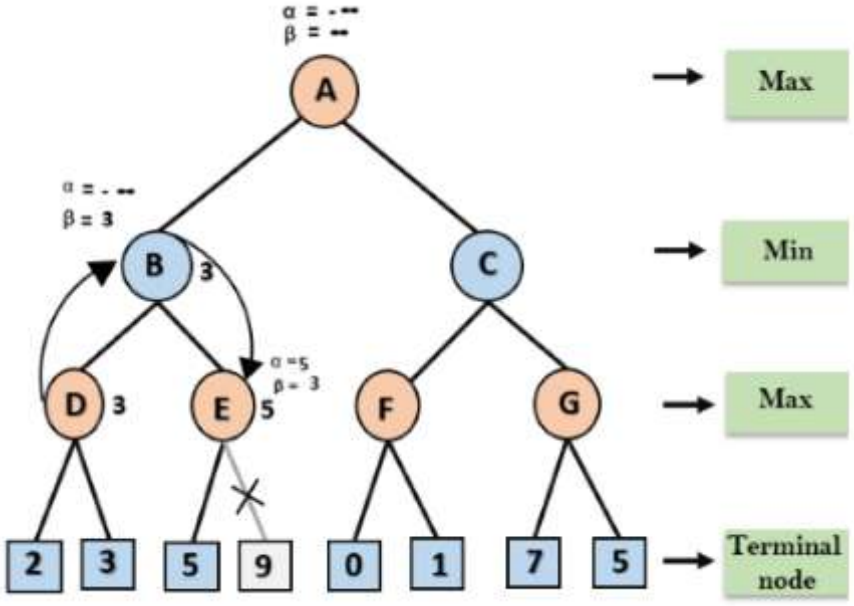
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtracks to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



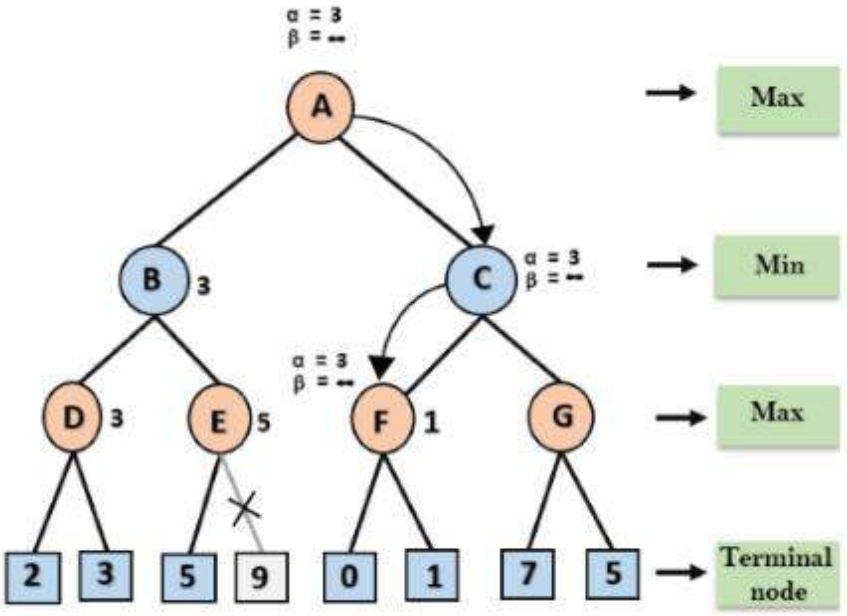
In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

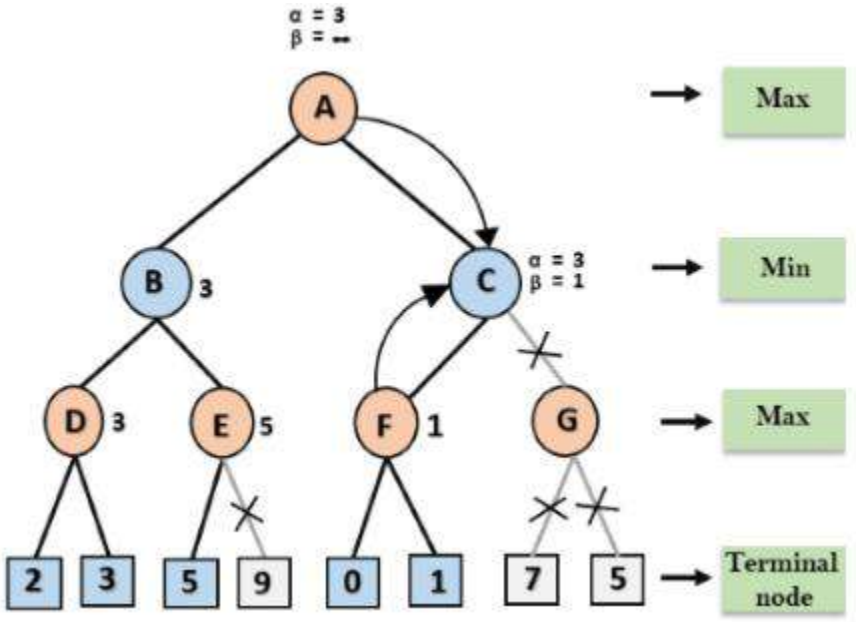


Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C. At node C, α=3 and β= +∞, and the same values will be passed on to node F.

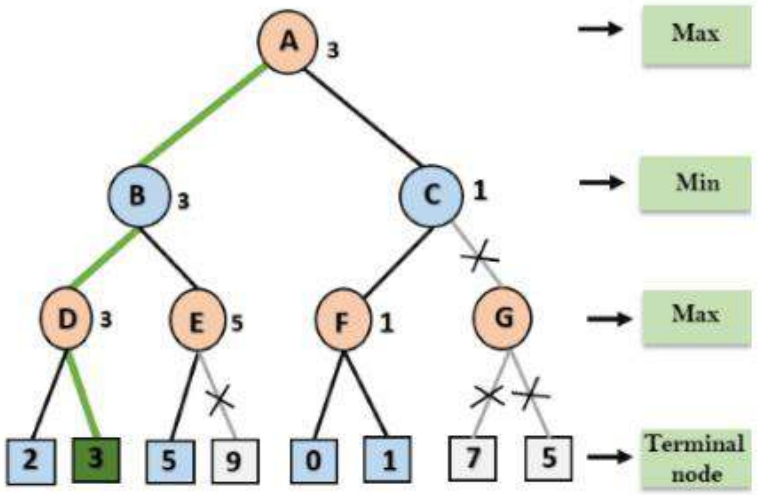
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



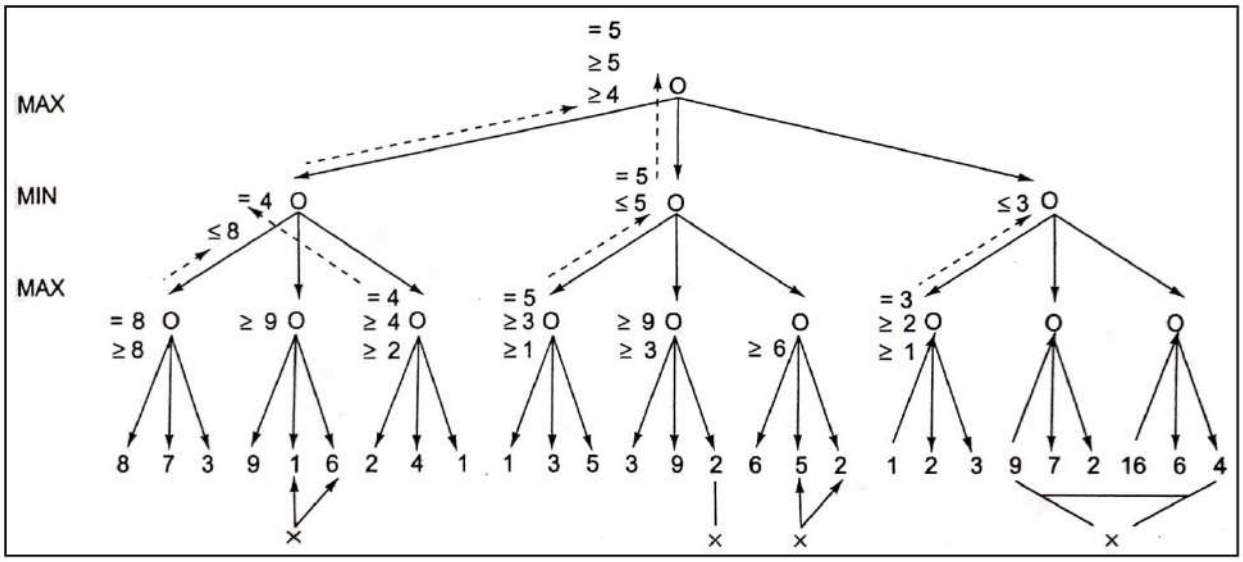
Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Example 2:



Time Complexity: O ($b^{m/2}$).

TWO-PLAYER PERFECT INFORMATION GAMES

- Chess
- Checkers
- Othello
- Go
- Backgammon

Chess: Chess is basically a competitive 2 player game played on a chequered board with 64 squares arranged in an 8 X 8 square. Each player is given 16 pieces of the same colour (black or white). These include 1 King, 1 Queen, 2 Rooks, 2 Knights, 2 Bishops & 8 pawns. Each of these pieces move in a unique manner. The player who chooses the white pieces gets the first turn to play. The players get alternate chances in which they can move one piece at a time. The objective of this game is to remove the opponent's king from the game. The opponent's King has to be placed in such a situation where the king is under immediate attack & there is no way to save it from the attack. This is known as Checkmate.

Checkers: Checkers (or) Draughts a 2 player game played on a chequered 8 X 8 square board. Each player gets 12 pieces of the same colour (dark or light) which are played on the dark squares of the board in3 rows. The row closest to a player is called the king row. The pieces in the king row are called kings, while others are called men. Kings can move diagonally forward as well as backward. Men can move only diagonally forward. A player can remove opponent's pieces from the game by diagonally jumping over them. When men pieces jump over king pieces of the opponent, they transform into Kings. The objective of this game is to remove all the pieces of the opponent from the board or by leading the opponent to such a situation where the opposing player is left with no legal moves.

Othello: Othello (or) Reversi is a 2 player board game which is played on an 8 X 8 square grid with pieces that have 2 distinct bi-coloured sides. The pieces typically are shaped as coins, but each possesses a light & a dark face, each face representing one player. The objective of this game is to make your pieces constitute a majority of the pieces on the board at the end of the game, by turning over as many of your opponent's pieces as possible.

Go: It is a strategic 2 player game in which the players play alternatively by placing black & white stone on the vacant intersections of a 19 X 19 board. The objective of the game is to control a larger part of the board than the opponent. To achieve this, players try to place their stones in such a manner that they cannot be captured by the opposing player. Placing stones close to each other helps them support one another & avoid capture. On the other hand, placing them far apart creates an influence across a larger part of the board. It is a strategy that enables players to play a defensive as well as an offensive game & choose between tactical urgency & strategic planning. A stone or a group of stones is captured & removed if it has no empty adjacent intersections, that is, it is completely surrounded by stones of the opposing colour. The game is declared over & the score is counted when both players consecutively pass on a turn, indicating that neither side can increase its territory or reduce that of its opponent's.

Backgammon: It is also a 2 player board game in which the playing pieces are moved using dice. A player wins by removing all of his pieces from the board. All though luck plays an important role, there is a large scope for strategy. With each roll of the dice a player must choose from numerous options for moving his checkers & anticipate the possible counter moves by the opponent. Players may raise the stakes during the game.

3. LOGIC CONCEPTS

PROPOSITIONAL CALCULUS (PC)

Propositional Calculus (PC) refers to a language of propositions in which a set of rules are used to combine simple propositions to form compound propositions with the help of certain logical operators. These logical operators are often called connectives. Examples of some connectives are not (~), and (Λ), or (V), implies (→) & equivalence (↔). A well defined formula is defined as a symbol or a string of symbols generated by the formal grammar of a formal language.

Properties of a well formed formula in PC:

- The smallest unit (or an atom) is considered to be a well formed formula.
- If α is a well-formed formula, then ~α is a well-formed formula.
- If α and β are well formed formulae, then (α Λ β), (α V β), (α → β), (α ↔ β) are also well-formed formulae.

Truth Table:

In PC, a truth table is used to provide operational definitions of important logical operators. The logical constants are true & false which are represented by T & F. Let us assume A, B, C, … are propositioned symbols.

A	**B**	**~A**	**A Λ B**	**A V B**	**A→B**	**A↔B**
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Example: Compute the truth value of α: (AVB) Λ(~B→A) using truth table approach.

Solution: Using the Truth Table approach, compute truth values of (AVB) and (~B→A) and then compute for the final expression (AVB) Λ(~B→A).

A	**B**	**A V B**	**~B**	**~B →A**	**α**
T	T	T	F	T	**T**
T	F	T	T	T	**T**
F	T	T	F	T	**T**
F	F	F	T	F	**F**

Equivalence Laws:

Equivalence relations (or laws) are used to reduce or simplify a given well-formed formula or to derive a new formula from the existing formula. Some of the important equivalence laws are:

Name of Relation	**Equivalence Relations**
Commutative Law	A V B ≅ B V A A Λ B ≅ B Λ A
Associative Law	A V (B V C) ≅ (A V B) V C

	A Λ (B Λ C) ≅ (A Λ B) Λ C
Double Negation	~ (~ A) ≅ A
Distributive Laws	A V (B Λ C) ≅ (A V B) Λ (A V C) A Λ (B V C) ≅ (A Λ B) V (A Λ C)
De Morgan's Law	~ (A V B) ≅ ~ A Λ ~ B ~ (A Λ B) ≅ ~ A V ~ B
Absorption Laws	A V (A Λ B) ≅ A A Λ (A V B) ≅ A A V (~A Λ B) ≅ A V B A Λ (~A V B) ≅ A Λ B
Idempotence	A V A ≅ A A Λ A ≅ A
Excluded Middle Law	A V ~ A ≅ T (True)
Contradiction Law	A Λ ~ A ≅ F (False)
Commonly used equivalence relations	A V F ≅ A A V T ≅ T A Λ T ≅ A A Λ F ≅ F A → B ≅ ~A V B A ↔ B ≅ (A → B) Λ (B → A) ≅ (A Λ B) V (~A Λ ~B)

Let us verify the absorption law A V (A Λ B) ≅ A using the Truth Table approach

A	**B**	**A Λ B**	**A V (A Λ B)**
T	T	T	**T**
T	F	F	**T**
F	T	F	**F**
F	F	F	**F**

PROPOSITIONAL LOGIC (PL)

Propositional Logic (or) prop Logic deals with the validity, satisfiability (also called consistency) and unsatisfiability (inconsistency) of a formula and the derivation of a new formula using equivalence laws. Each row of a truth table for a given formula α is called its interpretation under which the value of a formula may be true or false.

- A formula α is said to be a tautology if and only if the value of α is true for all its interpretations.

Now, the validity, satisfiability & unsatisfiability of a formula may be determined on the basis of the following conditions:

- A formula α is said to be valid if and only if it is a tautology.
- A formula α is said to be satisfiable if there exists at least one interpretation for which α is true.
- A formula α is said to be unsatisfiable if the value of α is false under all interpretations.

Example: Show that the following is a valid argument: “If it is humid then it will rain and since it is humid today it will rain”

Solution: Let us symbolize each part of the English sentence by propositional atoms as follows:

A: It is humid

B: It will rain

Now, the formula (α) corresponding to the given sentence: “If it is humid then it will rain and since it is humid today it will rain” may be written as

$$\alpha: [(A \rightarrow B) \wedge A] \rightarrow B$$

The truth table for $\alpha: [(A \rightarrow B) \wedge A] \rightarrow B$ is as follows

A	B	A → B = X	X Λ A = Y	Y→ B
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Truth table approach is a simple & straight forward method & is extremely useful at presenting an overview of all the truth values in a given situation. Although it is an easy method for evaluating consistency, in consistency (or) validity of a formula, the limitation of this method lies in the fact that the size of the truth table grows exponentially. If a formula contains **n** atoms, then the truth table will contain 2^n entries.

For example, if we have to show that a formula $\alpha: (A \wedge B \wedge C \wedge D) \rightarrow (B \vee E)$ is valid using the truth table approach, then we need to construct the table containing 32 rows & complete the truth values.

There are other methods which can help in proving the validity of the formula directly. Some other methods that are concerned with proofs & deductions are

- Natural Deductive System
- Axiomatic System
- Semantic Tableaux Method
- Resolution Refutation Method

NATURAL DEDUCTION SYSTEM (NDS)

Natural Deduction System **(**NDS) is thus called because of the fact that it mimics the pattern of natural reasoning. The system is based on as set of deductive inference rules. It has about 10 deductive inference rules. They are as follows

Rule Name	Symbol	Rule	Description
Introducing Λ	(*I*:Λ)	If $A_1, \ldots, A_n$ then $A_1 \Lambda \ldots \Lambda A_n$	If $A_1, \ldots, A_n$ are true, then their conjunction $A_1 \Lambda \ldots \Lambda A_n$ is also *true*.
Eliminating Λ	(*E*:Λ)	If $A_1 \Lambda \ldots \Lambda A_n$ then $A_i\ (1 \le i \le n)$	If $A_1 \Lambda \ldots \Lambda A_n$ is *true*, then any A_i is also *true*.
Introducing V	(*I*:V)	If any $A_i\ (1 \le i \le n)$ then A_1 V ... V A_n	If any $A_i\ (1 \le i \le n)$ is *true*, then A_1 V ... V A_n is also *true*.
Eliminating V	(*E*:V)	If A_1 V ... V $A_n, A_1 \rightarrow A, \ldots, A_n \rightarrow A$ then A	If A_1 V ... V $A_n, A_1 \rightarrow A, A_2 \rightarrow A, \ldots$, and $A_n \rightarrow A$ are *true*, then A is *true*.
Introducing →	(*I* : →)	If from $\alpha_1, \ldots, \alpha_n$ infer β is proved then $\alpha_1 \Lambda \ldots \Lambda \alpha_n \rightarrow \beta$ is proved	If given that $\alpha_1, \alpha_2, \ldots$, and α_n are *true* and from these we deduce β then $\alpha_1 \Lambda \ldots \Lambda \alpha_n \rightarrow \beta$ is also *true*.

Rule Name	Symbol	Rule	Description
Eliminating →	(*E*: →)	If $A_1 \rightarrow A, A_1$, then A	If $A_1 \rightarrow A$ and A_1 are *true* then A is also *true* This is called *Modus Ponen* rule.
Introducing ↔	(*I*: ↔)	If $A_1 \rightarrow A_2$, $A_2 \rightarrow A_1$ then $A_1 \leftrightarrow A_2$	If $A_1 \rightarrow A_2$ and $A_2 \rightarrow A_1$ are *true* then $A_1 \leftrightarrow A_2$ is also *true*.
Elimination ↔	(*E*: ↔)	If $A_1 \leftrightarrow A_2$ then $A_1 \rightarrow A_2, A_2 \rightarrow A_1$	If $A_1 \leftrightarrow A_2$ is *true* then $A_1 \rightarrow A_2$ and $A_2 \rightarrow A_1$ are *true*
Introducing ~	(*I*: ~)	*If from A infer* $A_1 \Lambda \sim A_1$ is proved then ~*A* is proved	If from *A* (which is *true*), a contradiction is proved then truth of ~*A* is also proved
Eliminating ~	(*E*: ~)	*If from* ~*A infer* $A_1 \Lambda \sim A_1$ is proved then *A* is proved	If from ~*A*, a contradiction is proved then truth of *A* is also proved

Example 1: Prove that A Λ (B V C) is deduced from A Λ B.

Solution: The theorem in NDS can be written as from A Λ B infer A Λ (B V C) in NDS. We can prove the theorem as follows

Description	Formula	Comments
Theorem	*from A Λ B infer A Λ (B V C)*	*To be proved*
Hypothesis (given)	*A Λ B*	1
E: Λ (1)	*A*	2
E: Λ (1)	*B*	3
I: V (3)	*B V C*	4
I: Λ (2, 4)	*A Λ (B V C)*	Proved

Example 2: Prove the theorem infer [(A→B) Λ (B→C)] → (A→C)

Solution: The theorem infer [(A→B) Λ (B→C)] → (A→C) is reduced to the theorem from (A→B), (B→C) infer (A→C) using deduction theorem. Further to prove 'A→C' we will have to prove a sub theorem from A infer C. The proof of the theorem is as follows

Description	Formula	Comments	
Theorem	*from* $A \rightarrow B, B \rightarrow C$ *infer* $A \rightarrow C$	To be proved	
Hypothesis 1	$A \rightarrow B$	1	
Hypothesis 2	$B \rightarrow C$	2	
Sub-theorem	*from A infer C*	3	
Hypothesis	A		3.1
E: → (1, 3.1)	B		3.2
E: → (2, 3.2)	C		3.3
I: → (3)	$A \rightarrow C$	Proved	

AXIOMATIC SYSTEM

The Axiomatic System is based on a set of 3 axioms & 1 rule called Modus Ponen (MP). Although it is minimal in structure, it is as powerful as truth table & NDS. In this system, only 2 logical operators not (~) & implies (→) are allowed to form a formula. It should be noted that other logical operators such as and (Λ), or (V) & equivalence (↔) can be easily expressed in terms of not (~) & implies (→) using equivalence laws. For example,

- A Λ B ≅ ~ (~A V ~B) ≅ ~ (A→ ~ B)
- A V B ≅ ~ A → B
- A ↔ B ≅ (A → B) Λ (B → A) ≅ ~ [(A → B) → ~ (B → A)]

In axiomatic system, α, β & γ are well-formed formulae.

Axiom 1: α → (β → α)

Axiom 2: (α →(β→γ)) →((α → β) → (α → γ))

Axiom 3: (~ α → ~ β) → (β → α)

Modus Ponen (MP) Rule: Hypotheses: α → β and α Consequent: β

Interpretation of Modus Ponen Rule: Given that α → β and α are hypotheses (assumed to be true), β is inferred (true) as a consequent.

Example 1**:** Establish that A → C is a deductive consequence of {A → B, B → C}, i.e.., {A→B, B→C} |- (A→ C)

Description	Formula	Comments
Theorem	$\{A \rightarrow B, B \rightarrow C\} \vdash (A \rightarrow C)$	*Prove*
Hypothesis 1	$A \rightarrow B$	1
Hypothesis 2	$B \rightarrow C$	2
Instance of Axiom 1	$(B \rightarrow C) \rightarrow [A \rightarrow (B \rightarrow C)]$	3
MP (2, 3)	$[A \rightarrow (B \rightarrow C)]$	4
Instance of Axiom 2	$[A \rightarrow (B \rightarrow C)] \rightarrow [(A \rightarrow B) \rightarrow (A \rightarrow C)]$	5
MP (4, 5)	$(A \rightarrow B) \rightarrow (A \rightarrow C)$	6
MP (1, 6)	$(A \rightarrow C)$	*Proved*

Example 2: Prove |- ~A → (A → B) by using deduction theorem.

Solution: If we can prove {~A} |- (A → B) then using deduction theorem, we have proved |- ~A → (A → B). The proof is a shown below

Description	Formula	Comments
Theorem	$\{\sim A\} \vdash (A \rightarrow B)$	*Prove*
Hypothesis 1	$\sim A$	1
Instance of Axiom 1	$\sim A \rightarrow (\sim B \rightarrow \sim A)$	2
MP (1, 2)	$(\sim B \rightarrow \sim A)$	3
Instance of Axiom 3	$(\sim B \rightarrow \sim A) \rightarrow (A \rightarrow B)$	4
MP (3, 4)	$(A \rightarrow B)$	*Proved*

Note: If ∑ is an empty set & α is deduced, then we can write |- α. In this case, α is deduced from axioms only & no hypotheses are used.

SEMANTIC TABLEAU SYSTEM IN PROPOSITIONAL LOGIC

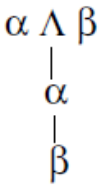
In both Natural Deduction & Axiomatic Systems, forward chaining approach is used for constructing proofs & derivations. In this approach we start proofs (or) derivations from a given set of hypotheses (or) axioms. In Semantic Tableau System, proofs follow backward chaining approach. In this method, a set of rules are applied systematically on a formula or a set of formulae in order to establish consistency or inconsistency.

Semantic Tableau Rules:

The Semantic Tableau Rules are given below where α & β are two formulae

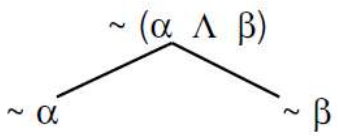
Rule 1: A tableau for a formula (α Λ β) is constructed by adding both α and β to the same path (branch).

α Λ β is true if both α and β are true



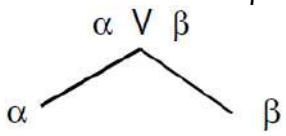
Rule 2: A tableau for a formula ~ (α Λ β) is constructed by adding two new paths, one containing ~ α and other containing ~ β.

~ (α Λ β) is true if either ~ α or ~ β is true



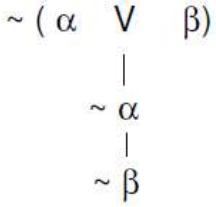
Rule 3: A tableau for a formula (α V β) is constructed by adding two new paths, one containing α and other containing β.

α V β is true if either α or β is true



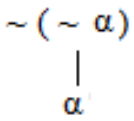
Rule 4: A tableau for a formula ~ (α V β) is constructed by adding both ~ α and ~ β to the same path.

~ (α V β) is true if both ~α and ~β are true



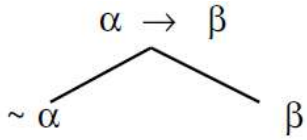
Rule 5: A tableau for ~ (~ α) is constructed by adding α on the same path.

~ (~ α) is true then α is true



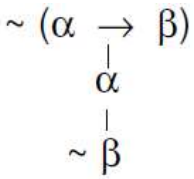
Rule 6: A tableau for a formula α → β is constructed by adding two new paths: one containing ~ α and the other containing β.

α → β is true then ~ α V β is true



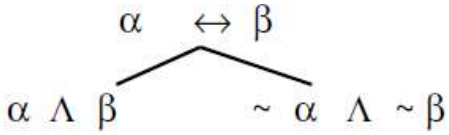
Rule 7: A tableau for a formula ~ (α → β) is constructed by adding both α & ~ β to the same path.

~ (α → β) is true then α Λ ~β is true



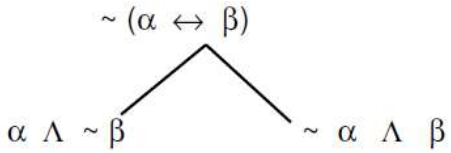
Rule 8: A tableau for a formula α ↔ β is constructed by adding two new paths: one containing α Λ β and other ~α Λ ~β which are furthur expanded.

α ↔ β is true then (α Λ β) V (~α Λ ~β) is true



Rule 9: A tableau for a formula ~ (α ↔ β) is constructed by adding two new paths: one containing α Λ ~β and the other ~α Λ β which are furthur expanded.

~ (α ↔ β) is true then (α Λ ~β) V (~α Λ β) is true



Example 1: Construct a Semantic Tableau for a formula (A Λ ~B) Λ (~B → C)

Description	Formula	Line number
Tableau root	$(A \wedge \sim B) \wedge (\sim B \rightarrow C)$	1
Rule 1 (1)	$A \wedge \sim B$	2
	$\sim B \rightarrow C$	3
Rule 1 (2)	A	4
	$\sim B$	5
Rule 6 (3)	$\sim(\sim B)$ C	6
Rule 3 (6)	B √(open)	
	× (closed) $\{B, \sim B\}$	

PREDICATE LOGIC

Logic: Logic is concerned with the truth of statements about the world. Generally each statement is either True or False. Logic includes

- Syntax
- Semantics
- Inference Procedure

Syntax: Specifies the symbols in the language about how they can be combined to form sentences. The facts about the world are represented as sentences in logic.

Semantic: Specifies how to assign a truth value to a sentence based on its meaning in the world. It specifies what facts a sentence refers to. A fact is a claim about the world, and it may be True or False.

Inference Procedure: Specifies methods for computing new sentences from an existing ones.

Predicate Logic: It is the study of individuals and their properties.

Conside the following set of sentences:

1. Marcus was a man.

2. Marcus was a Pompeian.
3. All Pompeians were Romans
4. Caesar was a ruler
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of WFF's (Well Formed Formula) in Predicate Logic as follows:

1. Marcus was a man.
   man (Marcus)
2. Marcus was a Pompeian.
   Pompeian (Marcus)
3. All Pompeians were Romans.
   ∀x: Pompeian(x) → Roman(x)
4. Caesar was a ruler.
   ruler (Caesar)
5. All Romans were either loyal to Caesar or hated him.
   ∀x: Roman(x)→loyal to (x, Caesar) V hate(x, Caesar)
6. Everyone is loyal to someone.
   ∀x: ∃y: loyal to(x, y)
7. People only try to assassinate rulers they are not loyal to
   ∀x: ∀y: person(x) ^ ruler(y) ^ try assassinate(x, y) → ¬ loyal to(x, y)
8. Marcus tried to assassinate Caesar.
   tryassassinate(Marcus, Caesar)

Now suppose we want to use these statements to answer the question

Was Marcus loyal to Caesar?

It seems that using 7 and 8, we should be able to prove that Marcus was not loyal to Caesar. Now let's try to produce a formal proof, reasoning backward from the desired goal:

¬ loyal to (Marcus, Caesar)

The attempt fails, however, since there is no way to satisfy the goal person(Marcus) with the statements available. We know that Marcus was a man; we do not have any way to conclude from that Marcus was a person. We need to add the representation of another fact to our system, namely:

9. All men are people. $\forall x : man(x) \rightarrow person(x)$

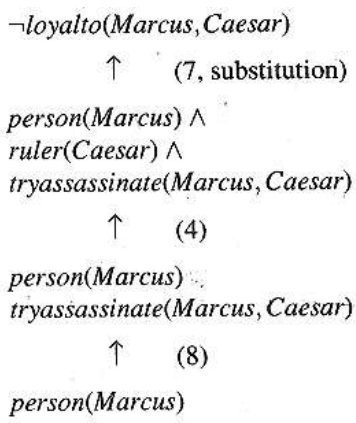


Fig: An Attempt to Prove ¬ loyalto (Marcus, Caesar)

Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar. From this example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones.

- Many English sentences are ambiguous. (Ex: 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
- There is often a choice of how to represent the knowledge (1 and 7). Simple representations are desirable, but they may preclude certain kinds of reasoning.
- Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand.

Representing *Instance* And *Isa* Relationships:

The below figure shows the five sentences of the last section represented in logic in three different ways. The first part of the figure contains the representations we have already discussed. In these representations, class membership is represented with unary predicates (such as Roman), each of which corresponds to a class. Asserting that p(x) is true is equivalent to asserting that x is an instance (or element) of P. The second part of figure contains representations that use the instance predicate explicitly.

The predicate instance is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit isa predicate. Instead, subclass relationships, such as that between Pompeians and Romans, are described as shown in sentence 3.The implication rule there states that if an object is an instance of subclass Pompeian then it is an instance of

superclass Roman. This rule is equivalent to subclass-superclass relationship. The third part contains representations that use both instance and isa predicates explicitly. The additional axiom describes how an instance relation and an isa relation can be combined to derive a new instance relation.

Statements as well-formed formulas (WFF's):

1. man (Marcus)
2. Pompeian (Marcus)
3. ∀x: Pompeian (x) → Roman(x)
4. ruler (Caesar)
5. ∀x: Roman (x) → loyal to (x, Caesar) V hate (x, Caesar)

Statement representation using INSTANCE predicate:

1. instance (Marcus, man)
2. instance (Marcus, Pompeian)
3. ∀x: instance (x, Pompeian) → instance (x, Roman)
4. instance (Caesar, ruler)
5. ∀x: instance (x, Roman) → loyalto (x, Caesar) V hate (x, Caesar)

Statements using ISA predicate:

1. instance (Marcus, man)
2. instance (Marcus, Pompeian)
3. isa (Pompeian, Roman)
4. instance (Caesar, ruler)
5. ∀x: instance (x, Roman) → loyalto (x, Caesar) V hate (x, Caesar)
6. ∀x: ∀y: ∀z: instance (x, y) ^ isa (y, z) → instance (x, z)

Computable Functions And Predicates:

In some cases we need to exempt some cases. For example that Paulus was a roman and he never hates Caesar. So in representing the statement that all roman was hated Caesar, only Paulus need to be exempted. And hence the statement will be as follows:

∀x: Roman (x) ^ ¬ equalto(x, Paulus) → loyalto(x, Caesar) V hate(x, Caesar)

These types of exceptions are very easy to represent in case of smaller number of statements. Suppose we want to express simple facts, such as the following greater-than and less-than relationships:

gt (1, 0) lt (0, 1)
gt (2, 1) lt (1, 2)
gt (3, 2) lt (2, 3) and so on.

If there is very large number of predicates like the above statements we can easily represent them and more over we can also simplify them if we have additional knowledge about them. That is we can infer new statements in smaller number by inferring new statements from the old.

1. Marcus was a man.
   Man (Marcus)
2. Marcus was a Pompeian.
   Pompeian (Marcus)
3. Marcus was born in 40 A.D.
   born (Marcus, 40)
4. All means are mortal.
   $\forall x$: man(x)$\rightarrow$ mortal(x)
5. All Pompeian's died when the volcano erupted in 79 A.D.
   Erupted (volcano, 79) ^ $\forall x$: [Pompeian (x) $\rightarrow$ died(x, 79)]
6. No mortal lives longer than 150yers.
   $\forall x$: $\forall t1$: $\forall t2$: mortal (x) ^ born (x, t1) ^ gt (t2-t1, 150) $\rightarrow$ dead (x, t2)
7. It is now 1991 A.D.
   now= 1991

Now we want to answer the question is "Is Marcus alive?" For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking. So we add the following facts:

8. Alive means not dead.
   $\forall x$: $\forall t$: [alive (x, t) $\rightarrow$ ¬dead (x, t)] ^ [¬dead (x, t) $\rightarrow$ alive (x, t)]

This is an example of the fact that rarely do two expressions have truly identical meanings in all circumstances.

9. If someone dies, then he is dead at all later times.
   $\forall x$: $\forall t1$: $\forall t2$: died (x, t1) ^ gt (t2, t1) $\rightarrow$ dead (x, t2)

This representation says that one is dead in all years after the one in which one died. Now let's attempt to answer the question "Is Marcus alive?" by proving:

¬alive (Marcus, now)

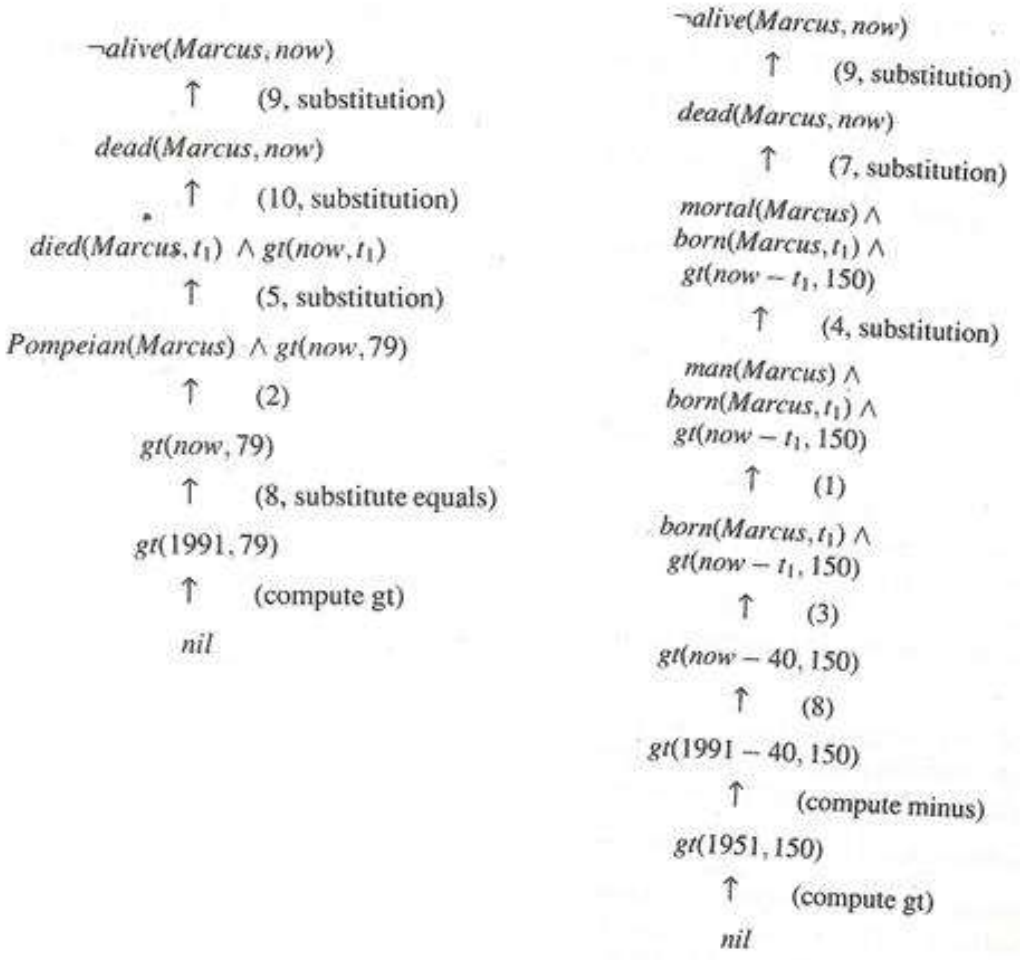


Fig: Two ways of Proving that Marcus Is Dead

RESOLUTION

Resolution is a procedure used in proving that arguments which are expressible in predicate logic are correct. Resolution is so far only defined for Propositional Logic. The resolution process can also be applied to predicate logic also. Resolution produces proofs by refutation. In other words, to prove the validity of a statement, resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., the statement is unsatisfiable).

The Basis of Resolution:

The resolution procedure is an iterative process; at each step, two clauses called the parent clauses containing the same literal, are compared (resolved), yielding a new clause called the resolvent, that has been inferred from them. The literal must occur in positive form in one clause and in negative form in the other. The resolvent is obtained by combining all of the literals of the parent clauses except the ones that cancel. If the clause that is produced is an empty clause, then a contradiction has been found.

Given: winter V summer

¬ winter V cold

We can conclude: summer v cold

In predicate logic, the situation is more complicated since we must consider all possible ways of substituting values for the variables. The theoretical basis of the resolution procedure in predicate logic is Herbrand's Theorem, which tells us the following.

- To show that a set of clauses S is unsatisfiable, it is necessary to consider only interpretations over a particular set, called the Herbrand universe of S.
- A set of clauses S is unsatisfiable if and only if finite subset of ground instances of S is unsatisfiable.

The second part of the theorem is important if there is to exist any computational procedure for proving unsatisfiability, since in a finite amount of time no procedure will be able to examine an infinite set. The first part suggests that one way to go about finding a contradiction is to try systematically the possible substitutions and see if each produces a contradiction.

So we observed here that to prove the statements using predicate logic the statements and well-formed formulas must be converted into clause form and the process will be as follows:

Conversion to Clause Form:

One method we shall examine is called resolution of requires that all statements to be converted into a normalized clause form .we define a clause as the disjunction of a number of literals. A ground clause is one in which no variables occur in the expression. A horn clause is a clause with at most one positive literal.

To transform a sentence into clause form requires the following steps:

1. Eliminate $\rightarrow$.
2. Reduce the scope of each $\neg$ to a single term.
3. Standardize variables so that each quantifier binds a unique variable.
4. Move all quantifiers to the left without changing their relative order.
5. Eliminate $\exists$ (Skolemization).
6. Drop $\forall$.
7. Convert the formula into a conjunction of disjuncts.
8. Create a separate clause corresponding to each conjunct.
9. Standardize apart the variables in the set of obtained clauses.

We describe the process of eliminating the existential quantifiers through a substitution process. This process requires that all such variables are replaced by something called Skolem functions. Sometimes ones with no arguments are called Skolem constants.

Example in Predicate logic: Suppose we know that **"all Romans who know Marcus either hate Caesar or think that any one who hates any one is crazy"**. We could represent that in the following wff:

$\forall x$: [Roman (x)$\wedge$know(x,marcus)] $\rightarrow$ [hate(x,Caesar) $\vee$ ($\forall$y: $\exists$z:hate(y,z) $\rightarrow$thinkcrazy(x,y))]

1. Eliminate implies symbol using the fact that $a \rightarrow b = \neg a \vee b$.

   $\forall x$:$\neg$[Roman(x)$\wedge$know(x,marcus)]$\vee$[hate(x,Caesar)$\vee$($\forall$y: $\neg$($\exists$z:hate(y,z) $\vee$ thinkcrazy(x,y))]

2. Reduce the scope of each $\neg$ to a single term.

   $\forall x$: [$\neg$Roman(x)$\vee\neg$know(x,marcus)]$\vee$[hate(x,Caesar)$\vee$($\forall$y:$\forall$z:$\neg$hate(y,z)$\vee$ thinkcrazy(x,y))]

   [$\neg$ ($\neg$ p) = p].[$\neg$(a $\wedge$b) = $\neg$a$\vee$ $\neg$ b]. [$\neg\exists$x: p(x) = $\forall$x: $\neg$ p(x)]. [$\neg\forall$x: p(x) = $\exists$x: $\neg$ p(x)].

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff.

   $\forall x$:P(x) $\vee$ $\forall x$:Q(x) would be converted into $\forall x$:P(x) $\vee$ $\forall$y:Q(y)

4. Move all quantifiers to the left of the formula with out changing their relative order.

   $\forall x$:$\forall$y:$\forall$z:[$\neg$Roman(x)$\vee\neg$know(x,marcus)]$\vee$[hate(x,Caesar)$\vee$ $\neg$hate(y,z) $\vee$ thinkcrazy(x,y))]

5. Eliminate existential quantifiers.

In our example, there are no existential quantifiers at step 4. There is no need to eliminate those quantifiers. If those quantifiers occur then use Skolem functions to eliminate those quantifiers. For ex:

$\exists$y: President (y) can be transformed into the formula President(S1)

$\forall x$: $\exists$y: father-of(y, x) would be transformed into $\forall x$: father-of (S2(x), x)

6. Drop the prefix. From (4).

   [$\neg$ Roman (x) $\vee\neg$know(x, Marcus)] $\vee$ [hate(x, Caesar) $\vee$ ($\neg$hate(y, z) $\vee$ thinkcrazy(x, y))]

7. Convert the matrix into a conjunction of disjuncts. In our problem that are no (ands) disjuncts. So use associative property. (a$\vee$b) $\vee$c = a$\vee$(b $\vee$c).

   $\neg$ Roman (x) $\vee\neg$know(x, Marcus) $\vee$ hate(x, Caesar) $\vee$ $\neg$hate(y, z) $\vee$ thinkcrazy(x, y).

   Ex: [winter V (summer ^ wearingsandals)] V [wearing boots V (summer ^ wearingsandals)]

   (winter V summer) ^

   (winter V wearingsandals) ^

   (wearing boots V summer) ^

   (wearing boots V wearingsandals)

8. Create a separate clause corresponding to each conjunct.

(winter V summer), (winter V wearingsandals), (wearing boots V summer), (wearing boots V wearingsandals)

Resolution in Propositional Logic:

We first present the resolution procedure for propositional logic. We then expand it to include predicate logic. In propositional logic, the procedure for producing a proof by resolution of proposition P with respect to a set of axioms F is the following.

Algorithm: Resolution Propositional

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until either a contradiction is found or no progress can be made:

a) Select two clauses. Call these the parent clauses.

b) Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both of the parent clauses with the following exception: if there are any pairs of literals L and ¬L such that one of the parent clauses contains L and other contains ¬L, then select one such pair and eliminate both L and ¬L from the resolvent.

c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

A Few Facts in Propositional Logic:

Given Axioms	Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

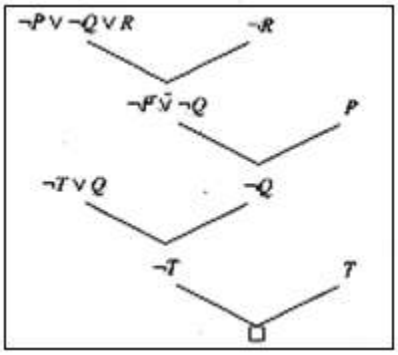


Fig: Resolution in Prepositional Logic

The Unification Algorithm:

In propositional logic, easily determine two literals and its contradictions. Simply look for L and **¬L** In predicate logic, this matching process is more complicated since the arguments of the predicates must be considered. For ex: man (John) and ¬man (John) is a contradiction. While man (John) and ¬man (John) is not. Thus in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. The recursive procedure that will do the above called the **unification algorithm**.

The basic idea of unification is very simple. To attempt to unify two literals, we first check,

1. If their initial predicate symbols are the same. If so, we can proceed, else we can't unify them. Ex. try assassinate (Marcus, Caesar). hate (Marcus, Caesar).
2. If the predicate symbols match, we must check the arguments, one pair at a time. If the first matches, we can continue with the second, and so on. To test each argument pair we can simply call the unification procedure recursively. The matching rules are
    - Different constants or predicates cannot match; identical ones can.
    - A variable can match another variable, any constant, or a predicate expression.

With the restriction that the predicate expression must not contain any instances of the variables being matched. For example, suppose we want to unify the expressions

P (x, x) and P (y, z)

The two instances of P match fine. Next we compare x and y, and decides that if we substitute y for x, they could match. We will write that substitute y for x → y/x.

The next match x and z, we produce the substitution z for x. → z/x. But we cannot substitute both y and z for x, we have not produced a consistent substitution. To avoid this we'll make the substitution y/x through out the literals.

P (y, y) and P (y, z) (substitution of y/x)

Now, we can attempt to unify arguments y and z which succeeds the substitution z/y

Algorithm: Unify (L1, L2)

1. If L1 and L2 are both variables or constants, then:
    a) If L1 and L2 are identical, then return NIL.
    b) Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return(L2/L1).
    c) Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL}, else return(L1/L2).
    d) Else return {FAIL}.
2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL}.

3. If L1 and L2 have a different number of arguments, then return {FAIL}.

4. Set SUBST to NIL.

5. For i ← 1 to number of arguments

   a) Call Unify with the ith argument of L1 and ith argument of L2, putting result in S.

   b) If S contains FAIL then return {FAIL}.

   c) If S is not equal to NIL then:

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

      ii. SUBST:= APPEND (S, SUBST).

6. Return SUBST.

Resolution in Predicate Logic:

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until either a contradiction is found or no progress can be made, or a predetermined amount of effort has been expended.
   a) Select two clauses. Call these the parent clauses.
   b) Resolve them together. The resolve will be the disjunction of all of the literals of both parent clauses with appropriate substitutions performed and with the following exception: if there is one pair of literals T1 and ¬T2 such that one of the parent clauses contains T1 and other contains T2 and if T1 and T2 are unifiable, then neither T1 nor T2 complementary literals. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.
   c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

- Axioms in clause form:
  1. *man(Marcus)*
  2. *Pompeian(Marcus)*
  3. ¬ *Pompeian*(x_1) v Roman(x_1)
  4. *Ruler*(*Caesar*)
  5. ¬ *Roman*(x_2) v *loyalto*(x_2, *Caesar*) v *hate*(x_2, *Caesar*)
  6. *loyalto*(x_3, *f*1(x_3))
  7. ¬ *man*(x_4) v ¬ *ruler*(y_1) v ¬ *tryassassinate*(x_4, y_1) v *loyalto* (x_4, y_1)
  8. *tryassassinate*(*Marcus, Caesar*)

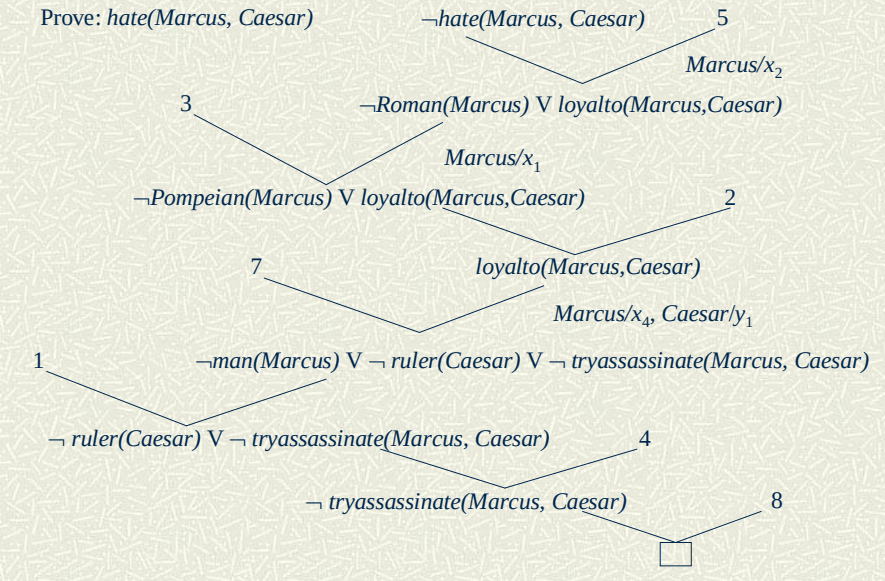


Fig: A Resolution Proof

9. persecute (x, y) → hate (y, x)

10. hate (x, y) → persecute (y, x)

Converting to clause form we get

9. ¬persecute (x_5, y_2) V hate (y_2, x_5)

10. ¬hate (x_6, y_3) V persecute (y_3, x_6)

Now to detect that there is no contradiction we must discover that the only resolvents that can be generated before. In other words, although we can generate resolvents, we can generate no new ones.

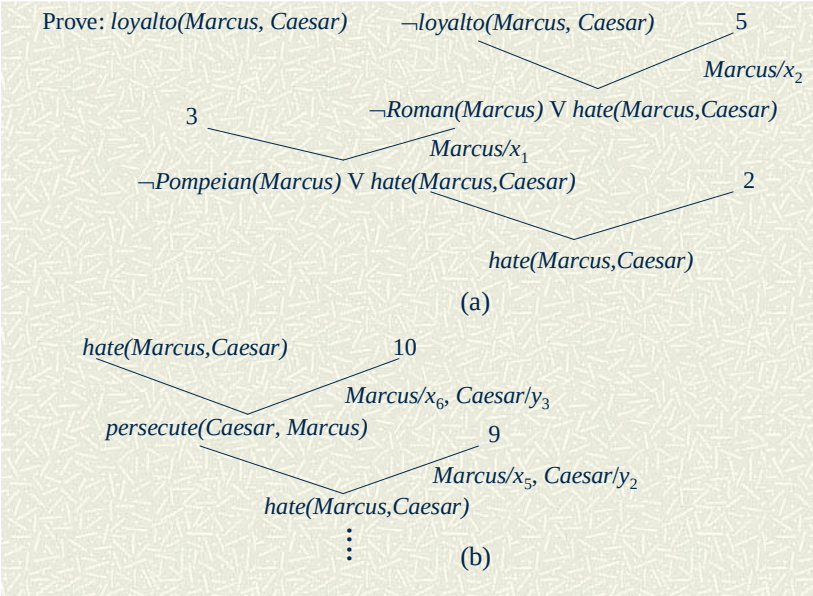


Fig: An Unsuccessful Attempt at Resolution

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\neg man(x_1) \vee mortal(x_1)$
5. $\neg Pompeian(x_2) \vee died(x_2, 79)$
6. $erupted(volcano, 79)$
7. $\neg mortal(x_3) \vee \neg born(x_3, t_1) \vee \neg gt(t_2 - t_1, 150) \vee dead(x_3, t_2)$
8. $now = 1991$

9a. $\neg alive(x_4, t_3) \vee \neg dead(x_4, t_3)$

9b. $dead(x_5, t_4) \vee alive(x_5, t_4)$

10. $\neg died(x_6, t_5) \vee \neg gt(t_6, t_5) \vee dead(x_6, t_6)$

Prove: $\neg alive(Marcus, now)$

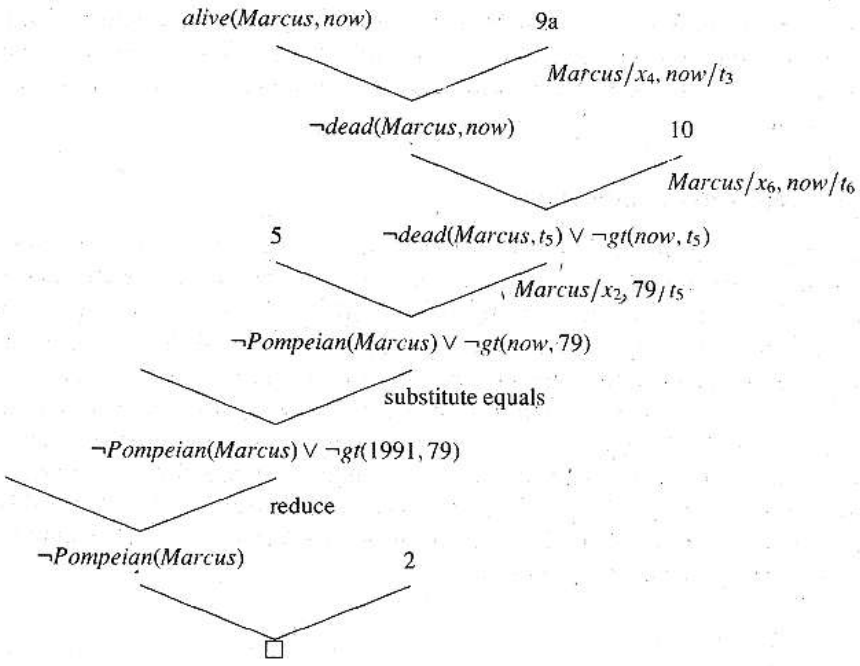


Fig: Using Resolution with Equality and Reduce

4. KNOWLEDGE REPRESENTATION

INTRODUCTION

Knowledge representation is an important issue in both cognitive science & AI. In cognitive science, it is concerned with the way in which information is stored & processed by humans, while in AI, the main focus is on storing knowledge or information in such a manner that programs can process it & achieve human intelligence. In AI, knowledge representation is an important area because intelligent problem solving can be achieved and simplified by choosing an appropriate knowledge representation technique. Representing knowledge in some specific ways makes certain problems easier to solve. The fundamental goal of knowledge representation is to represent knowledge in a manner that facilitates the process of inferencing (drawing conclusions) from it.

Any knowledge representation system should possess properties such as learning, efficiency in acquisition, representational adequacy & inferential adequacy.

- Learning: refers to a capability that acquires new knowledge, behaviours, understanding etc.
- Efficiency in acquisition: refers to the ability to acquire new knowledge using automatic methods wherever possible rather than relying on human intervention.
- Representational adequacy: refers to ability to represent the required knowledge.
- Inferential knowledge: refers to the ability of manipulating knowledge to produce new knowledge from the existing one.

APPROACHES TO KNOWLEDGE REPRESENTATION

Knowledge structures represent objects, facts, relationships & procedures. The main function of these knowledge structures is to provide expertise and information so that a program can operate in an intelligent way. Knowledge structures are usually composed of both traditional & complex structures such as semantic network, frames, scripts, conceptual dependency etc.

1. Relational Knowledge: Relational knowledge comprises objects consisting of attributes & associated values. The simplest way of storing facts is to use a relational method. In this method, each fact is stored in a row of a relational table as done in relational database. A table is defined as a set of data values that is organized using a model of vertical columns identified by attribute names & horizontal rows identified by the corresponding values.

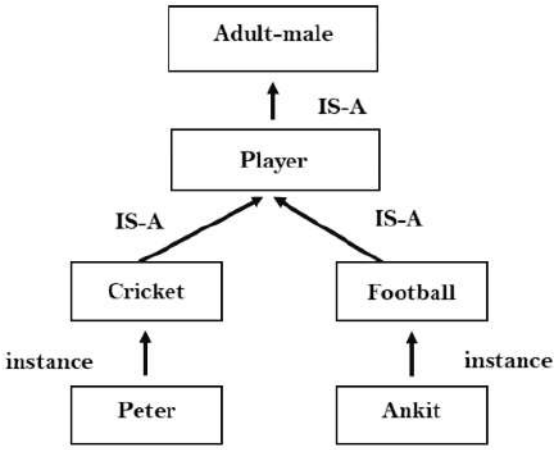
Name	Age (in years)	Sex	Qualification	Salary (in Rupees)
John	38	Male	Graduate	20000
Mike	25	Male	Undergraduate	15000
Mary	30	Female	Ph D	25000
James	29	Male	Graduate	18000

From the above table we can easily obtain the answers of the following queries:

- What is the age of John?
- How much does Mary earn?
- What is the qualification of Mike?

However, we cannot obtain answers for queries such as “Does a person having PhD qualification earn more?” So, inferencing new knowledge is not possible from such structures.

2. Inheritable Knowledge: In the inheritable knowledge approach, all data must be stored into a hierarchy of classes. Elements inherit values from other members of a class. This approach contains inheritable knowledge which shows a relation between instance and class, and it is called instance relation. Every individual frame can represent the collection of attributes and its value. Objects and values are represented in Boxed nodes. We use Arrows which point from objects to their values. For example



3. Inferential Knowledge: Inferential capability can be achieved if knowledge is represented in the form of formal logic. It guaranteed correctness. For example, Let's suppose there are two statements:
    a. Marcus is a man
    b. All men are mortal

    Then it can represent as;

    man(Marcus)

    ∀x = man (x) ----------> mortal (x)s
4. Procedural Knowledge: It is encoded in the form of procedures which carry out specific tasks based on relevant knowledge. For example, an interpreter for a programming language interprets a program on the basis of the available knowledge regarding the syntax & semantics of the language. The advantages of this approach are that domain-specific knowledge can be easily represented & side effects of actions may also be modeled. However there is a problem of completeness (all cases may not be represented) & consistency (all deductions may not be correct).

KNOWLEDGE REPRESENTATION USING SEMANTIC NETWORK

The basic idea applied behind using semantic network is that the meaning of concept is derived from its relationship with other concepts; the information is stored by interconnecting nodes with labeled arcs. For example, consider the following knowledge:

- Every human, animal & birds are living things who can breathe & eat. All birds can fly. Every man & woman are human who have 2 legs. A cat has fur & is an animal. All animals have skin & can move. A giraffe is an animal & has long legs & is tall. A parrot is a bird & is green in colour.

We can represent such knowledge using a structure called as Semantic Network (or) Semantic Net. It is conveniently represented in a graphical notation where nodes represent concepts or objects & arcs represent relation between 2 concepts. The relations are represented by bold directed links.

- **isa**: This relation connects 2 classes, where one concept is a kind or subclass of the other concept. For example, "Man isa Human" means Man is a subclass of the Human class.
- **inst**: This relation relates specific members of a class, such as John is an instance of Man.

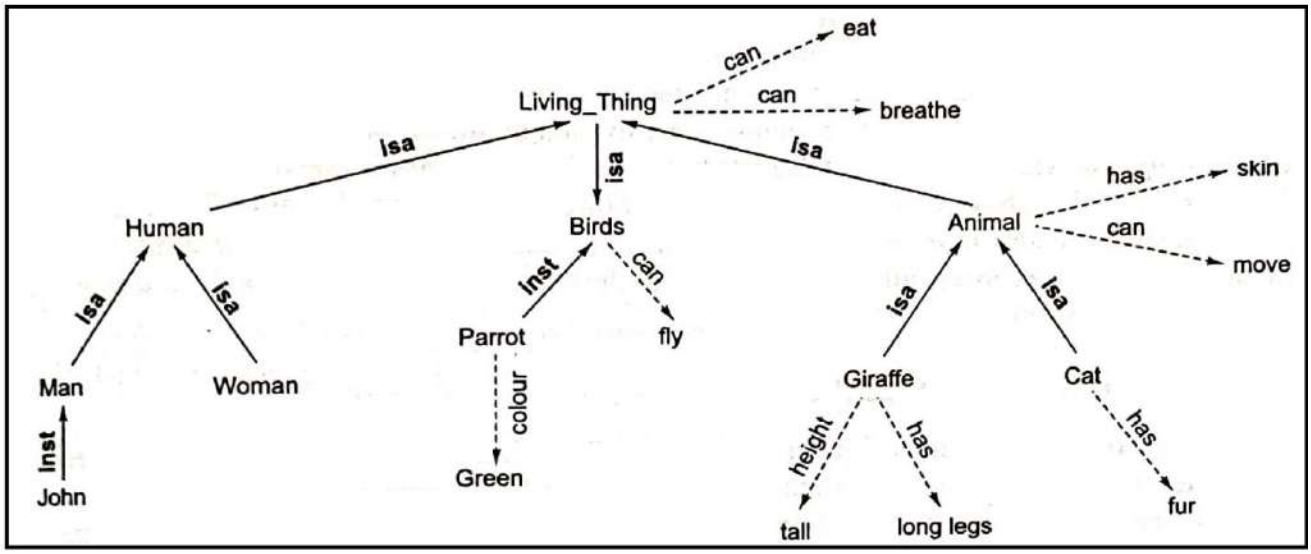


Fig: Knowledge Representation using Semantic Network

Other relations such as {can, has, colour, height} are known as property relations which are been represented by dotted lines. For example, the query "Does a parrot breathe?" can be easily answered as 'Yes' even though this property is not associated directly with parrot. It inherits this property from its super class named Living_Thing.

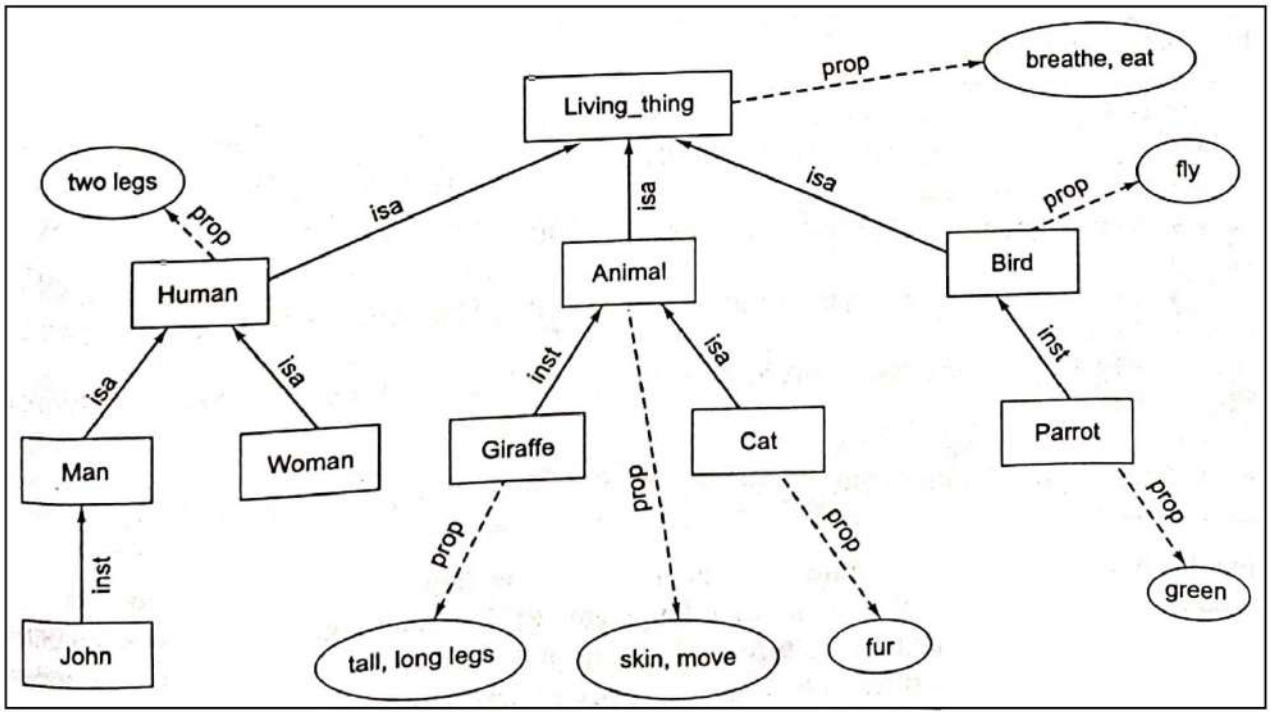


Fig: Concepts connected with Prop Links

Inheritance in Semantic Net:

Hierarchical structure of knowledge representation allows knowledge to be stored at the highest possible level of abstraction which reduces the size of the knowledge base. Since semantic net is stored as hierarchical structure, the inheritence mechanism is in-built & facilitates inferencing of information associated with nodes in semantic net.

Algorithm:

Input: Object & property to be found from Semantic Net:

Output: returns yes, if the object has the desired property else returns false.

Procedure:

- Find an object in the semantic net;
- Found=False:
- While [(object ≠ root) OR Found] Do

  {

  - If there is an attribute attached with an object then Found = True:
    else {object = isa(object,class) or object=inst(object.class)}

  };
- If Found=true then report 'Yes' else report 'No';

Semantic Net can be implemented in any programming language along with inheritance procedure implemented explicitly in that language. Prolog language is very convenient for representing an entire semantic structure in the form of facts & inheritance rules.

Prolog Facts: The facts in Prolog would be written as shown below

Isa facts	Instance facts	Property facts
isa(living_thing, nil).	inst(john, man).	prop(breathe, living_thing).
isa(human, living_thing).	inst(giraffe, animal).	prop(eat, living_thing).
isa(animals, living_thing).	inst(parrot, bird)	prop(two_legs, human).
isa(birds, living_thing).		prop(skin, animal).
isa(man, human).		prop(move, animal).
isa(woman, human).		prop(fur, bird).
isa(cat, animal).		prop(tall, giraffe).
		prop(long_legs, giraffe). prop(tall, animal). prop(green, parrot).

Table: Prolog Facts

Inheritance Rules in Prolog: We know that in class hierarchy structure, a member subclass of a class is also a member of all super classes connected through isa link. Similarly, an instance of a subclass is also an instance of all super classes connected by isa link. Similarly, a property of a class can be inherited by lower sub-classes.

```
Instance rules
instance(X, Y)    :-    inst(X, Y).
instance (X, Y)   :-    inst(X, Z), subclass(Z,Y).
Subclass rules
subclass(X, Y)    :-    isa(X, Y).
subclass(X, Y)    :-    isa(X, Z), subclass(Z, Y) .
Property rules
property(X, Y)    :-    prop(X, Y).
property(X, Y)    :-    instance(Y, Z), property(X, Z).
property(X, Y)    :-    subclass(Y, Z), property(X, Z).
```

Various queries can be answered by the above inheritance program as follows

English query	Prolog goal	Output
Is john human?	?- instance(john, humans).	Yes
Is parrot a living thing?	?- instance (parrot, living_thing).	Yes
Is giraffe an animal?	?- instance (giraffe, animal).	Yes
Is woman a subclass of living thing?	?- subclass(woman, living_thing).	Yes
Does parrot fly?	?- property(fly, parrot).	Yes
Does parrot have fur?	?- Does parrot have fur?	No
Does john breathe?	?- property (john, breathe).	Yes
Does cat fly?	?- property(fly, cat).	No

Table: Various Queries for Inheritance Program

EXTENDED SEMANTIC NETWORKS FOR KR

Logic and semantic networks are two different formalisms that can be used for knowledge representations. Simple semantic network is represented as a directed graph whose nodes represent concepts or objects and arcs represent relationships between concepts or objects. It can only express collections of variable-free assertions. The English sentences *"john gives on apple to mike and john and mike are human"* may be represented in semantic network as shown

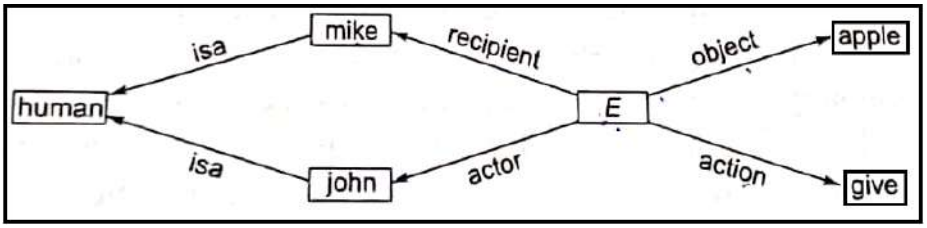


Fig: Semantic Net

Here 'E' represents an event which is an act of giving, whose actor is *John*, the object is an *apple*, and recipient is *mike*. It should be noted that semantic net can hold semantic information about situation such as actor of an event *giving is john* and object is *apple*, in the sentence *john gives an apple to mike*. The relationships in the network shown can be expressed in clausal form of logic as follows:

object(E, apple).

action(E, give).

recipient(E, mike).

isa(john, human).

isa(mike, human).

Predicate relations corresponding to labels on the arcs of semantic networks always have two arguments. Therefore, the entire semantic net can be coded using binary representation. Such representation is advantageous when additional information is added to the given system. For example, in the sentence *john gave an apple to mike in the kitchen*, it is easy to add *location(E,kitchen)* to the set of facts given above.

In first-order predicate logic, predicate relation can have n arguments, where n>=1. For example, the sentence *john gives an apple to mike* is easily represented in predicate logic by *give(john, mike, apple)*. Here , john, mike, and apple are arguments, while *give* represents a predicate relation. The predicate logic represention has greater advantages compared to semantic met representation as it can express general propositions, in addition to simple assertions. For example, the sentence *john gives an apple to everyone he likes* is expressed in predicate logic by clause as follows:

give(john, X, apple) ← likes (john, X)

Here, the symbol X is a variable representing any individual. The arrow represents the logical connective *implied by*. The left side of ← contains conclusion(s), while the right side contains condition(s).

Despite all the advantages, it is not convenient to add new information in an n-ary representation of predicate logic. For example, if we have 3-ary relationship give(john, mike, apple) representing *john gives an apple to mike* and to capture an additional information about kitchen in the sentence *john gave an appe to mike in the kitchen*, we would have to replace the 3-ary representation *give(john, mike, apple)* with a new 4-ary representation such as *give(john, mike, apple, kitchen)*.

Further, a clause in logic can have several conditions, all of which must hold for the conclusion to be true. For example,

- The sentence *if john gives something he lkes to a person, then he also likes that person* can be expressed in clausal representation in logic as

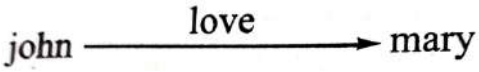
  likes(john,X)←give(john,X,Y),likes(john,Y)

- The sentence *every human is either male or female* is expressed by the following clause

  male(X), female(X) ← human(X)

In coventional semantic network, we cannot express clausal form of logic. To overcome this, R Kowalski and his colleagues (1979) proposed an Extended Semantic Network(ESNet) that combines the advantages of both logic and semantic networks. ESNet can be interpreted as a variant syntax for the clausal form of logic. It has the same expressive power as that of predicate logic with well-defined semantics, inference rules, and a procedural interpretation. It also incorporates the advantages of using binary relation as in semantic network rather than n-ary relations of logic.

Binary predicate symbols in clausal logic are represented by labels on arcs of ESNet. An *atom* of the form love(john.mary) is an arc labeled as *love* with its two end nodes representing *john and mary*. The direction of the arc (link) indicates the order of the arguments of the predicate symbol which labels the arc as follows:



Conclusions and conditions of clausal form are represented in ESNet by different kinds of arcs. The arcs denoting conditions are drawn with dotted arrow lines. These are called denial links (-------->), while the arcs denoting conclusions are drawn with continuous arrow lines. These are known as assertion links(→). For example, the clausal representation *grandfather(X,Y) ← father(X, Z), parent(Z, Y)* for grandfather in logic can be represented in ESNet as given below. Here, X and Y are variables; *grandfather(X, Y)* is the consequent(conclusion), and *father(X, Z)* and *parent(Z, Y)* are the antecedents (conditions).

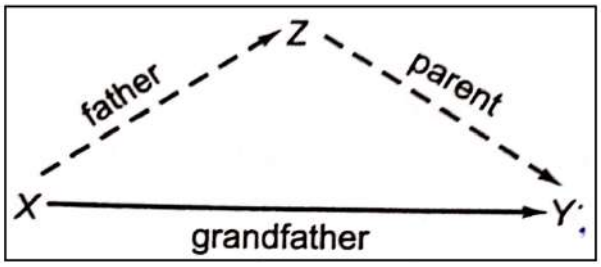


Fig: ESNet Representation

Similarly, the causal rule *male(X), female(X) ← human(X)* can be represented using binary representation as *isa(X, male), isa(X, female) ← isa(X, human).*

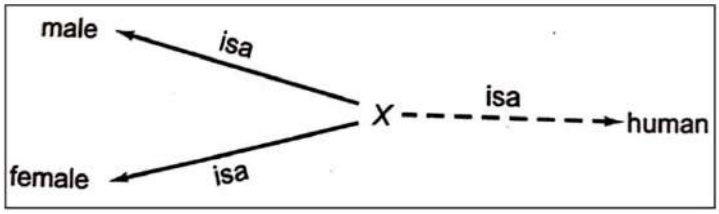


Fig: ESNet Representation

Inference Rules:

- The representation of the inference for every action of *giving*, there is an action of *taking* in clausal logic is *action(f(X)), take) ← action(X, give)*. The interpretation of this rule is that the event of *taking* action is a funtion of the event of *giving* action. In the ESNet representation, functional terms, such as f(X), are represented by a single node. The representation of the statement *action(f(X)), take) ← action(X, give).*

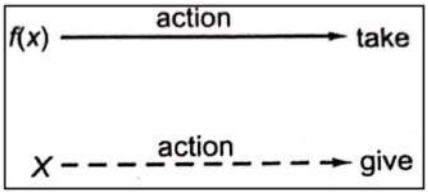


Fig: ESNet Representation

- The inference rule that an actor who performs a *taking* action is also the *recipient* of this action and can be easily represented in clausal logic; ESNet as given below. Here , E is a variable representing an event of an action of taking.

recipient(E, X) ← action(E, take), actor(E,X)

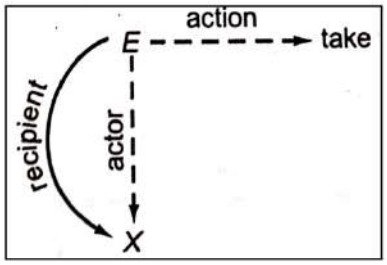


Fig: ESNet Representation

- The contradiction in ESNet can be represented as shown. Here *P* part_of *X* is conclusion and *P* part_of *Y* is condition, where *Y* is linked with *X* via *isa* link. Such kind of representation is contradictory and hence there is a contradiction in ESNet.

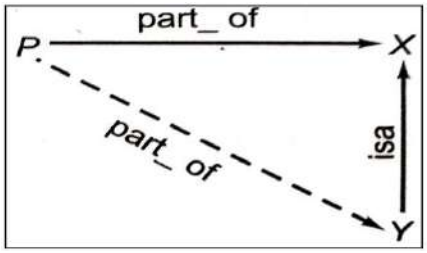


Fig: Contradiction in ESNet

Deduction in Extended Semantic Networks:

- Forward reasoning inference mechanism (bottom-up appoach)
- Backward reasoning inference mechanism (top-down approach)

Forward reasoning inference mechanism:

Given an ESNet, apply the following reduction (resolution) using modus ponen rule of logic {i.e., given (A←B) and B, then conclude A}. For example consider the following set of clauses:

isa(X,human)←isa(X,man)

isa(john,man)

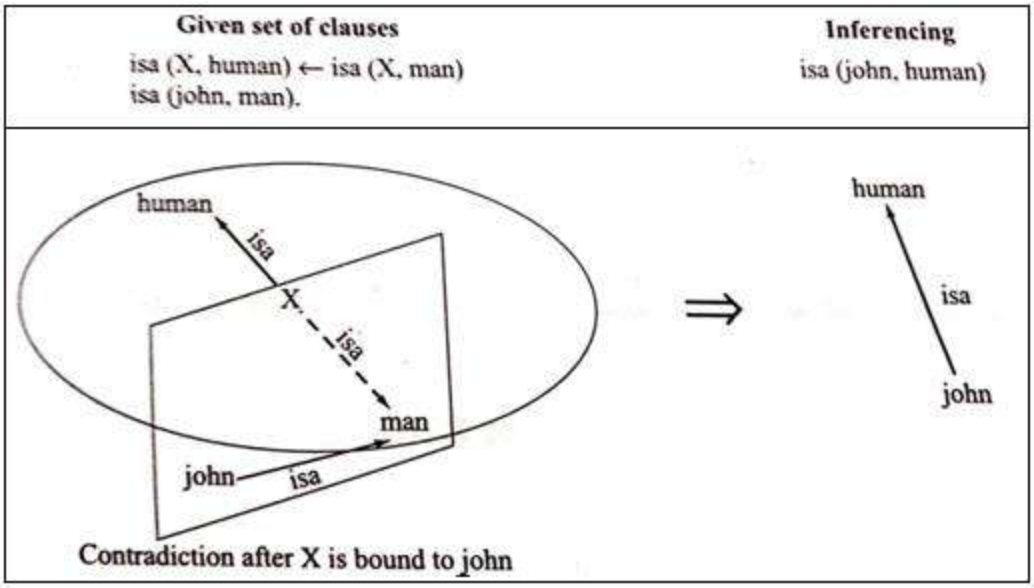


Fig: Forward reasoning inference mechanism

Backward reasoning inference mechanism:

In this mechanism, we can prove a conclusion (or) goal from a given ESNet by adding the denial of the conclusion to the network and show that the resulting set of clauses in the network gives contradiction.

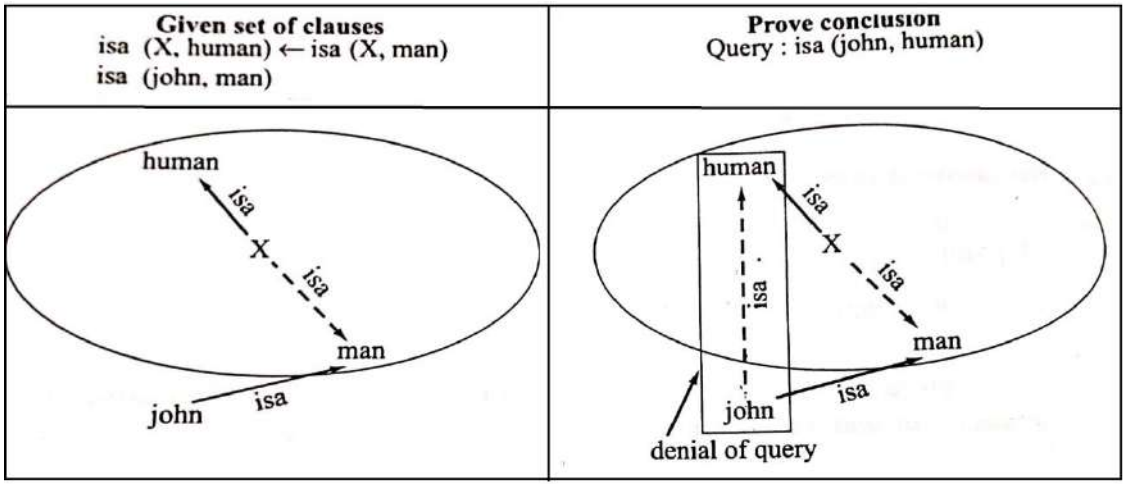


Fig: Backward reasoning inference mechanism

After adding denial link in ESNet, we get the reduction in ESNet as follows

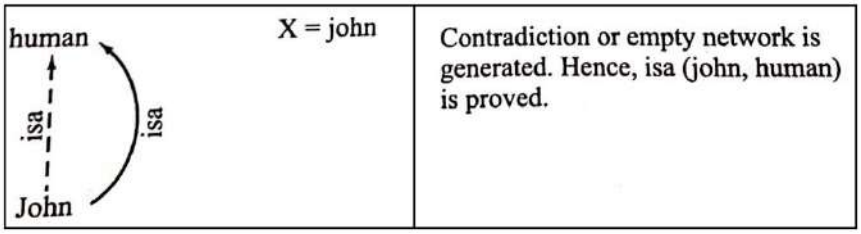


Fig: Reduction of ESNet

KNOWLEDGE REPRESENTATION USING FRAMES

A frame is a collection of attributes (usually called slots) and associated values (and possible constraints on values) that describes some entity in the world.

Frames as Sets and Instances:

Set theory provides a good basis for understanding frame system. Each frame represents either a class (a set) or an instance (an element of a class).The frame system is shown as

Person	
isa :	*Mammal*
cardinality :	6,000,000,000
* *handed* :	*Right*
Adult-Male	
isa :	*Person*
cardinality :	2,000,000,000
* *height* :	5-10
ML-Baseball-Player	
isa :	*Adult-Male*
cardinality :	624
* *height* :	6-1
* *bats* :	equal to handed
* *batting-average* :	.252
* *team* :	
* *uniform-color* :	
Fielder	
isa :	*ML-Baseball-Player*
cardinality :	376
* *batting-average* :	.262
Pee-Wee-Reese	
instance :	*Fielder*
height :	5-10
bats :	*Right*
batting-average :	.309
team :	*Brooklyn-Dodgers*
uniform-color :	*Blue*
ML-Baseball-Team	
isa :	*Team*
cardinality :	26
* *team-size* :	24
* *manager* :	
Brooklyn-Dodgers	
instance :	*ML-Baseball-Team*
team-size :	24
manager :	*Leo-Durocher*
players :	{*Pee-Wee-Reese*, ...}

In this example, the frames Person, Adult-Male, ML-Baseball-Player, Fielder, and ML-Baseball-Team are all classes. The frames Pee-Wee-Reese and Brooklyn-Dodgers are instances.

Hence, the isa relation is used to define the subset relation. The set of adult males is a subset of the set of people. The set of baseball players is a subset of the set of adult males, and so forth.

Our instance relation corresponds to the relation element-of. Pee-Wee-Reese is an element of the set of fielders. Thus he is also an element of all of the supersets of fielders, including baseball players and people.

Both the isa and instance relations have inverse attributes, which we call subclasses and all-instances. Because a class represents a set, there are two kinds of attributes that can be associated with it. There are attributes about the set itself, and there are attributes that are to be inherited by each element of the set. We indicate the difference between these two by prefixing the later with an asterisk (*).

For example, consider the class ML-Baseball-Player. We have shown only two properties of it as a set: It is a subset of the set of adult males. And it has cardinality 624. We have listed five properties that all ML-Baseball-Player have (height, bats, batting-average, team, and uniform-color), and we have specified default values for the first three of them. By providing both kinds of slots, we allow a class both to define a set of objects and to describe a prototypical object of the set.

We can view a class as two things simultaneously. A subset (isa) of a large class which also contains its elements and instances of class subsets from which it inherits it set-level properties.

To distinguish between two classes whose elements are individual entities and Meta classes (special classes) whose elements are themselves classes. A class in now an element of a class as well as subclass of are one or own class. A class inherits the property from the class of which it is an instance. A class passes inheritable properties down from its super classes to its instances.

4. ADVANCED KNOWLEDGE REPRESENTATION TECHNIQUES

CONCEPTUAL DEPENDENCY (CD)

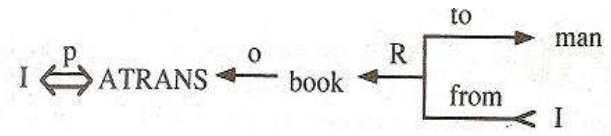
Conceptual dependency is a theory of how to represent the kind of knowledge about events that is usually contained in natural language sentences. The goal is to represent the knowledge in a way that

- Facilitates drawing inferences from the sentences.
- Is independent of the language in which the sentences were originally stated.

Representing Knowledge:

Because of the above two concerns mentioned, the **CD** representation of a sentence is built not out of primitives corresponding to the words used in the sentence, but rather out of conceptual primitives that can be combined to form the meanings of words in any particular language. It was first propose by SCHANK. Unlike Semantic nets, CD provides both a structure and a specific set of primitives, out of which representations of particular pieces of information can be constructed. For ex, we can represent the sentence.

"I gave the man a book"



where the symbols have the following meanings:

- Arrows indicates direction of dependency.
- Double arrow indicates two way links between actor and action.
- P indicates past tense.
- ATRANS is one of the primitive acts used by the theory. It indicates transfer of possession.
- indicates the object case relation.
- R indicates the recipient case relation

In CD, representations of actions are built from a set of primitive acts. Examples of Primitive Acts are:

ATRANS – Transfer of an abstract relationship (**Eg: give)**

PTRANS – Transfer of the physical location of an object (**Eg: go)**

PROPEL – Application of a physical force to an object (**Eg: push)**

MOVE – Movement of a body part by its owner (**Eg: kick)**

GRASP– Grasping of an object by an actor (**Eg: clutch)**

INGEST – Ingestion of an object by an animal (**Eg: eat)**

EXPEL – Expulsion of something from the body of an animal (**Eg: cry)**

MTRANS– Transfer of mental information (**Eg: tell)**

MBUILD – Building new information out of old (**Eg: decide)**

SPEAK – production of sounds (**Eg: say)**

ATTEND – Focusing of a sense organ toward a stimulus **(Eg: listen)**

A Second set of CD building blocks is the set of allowable dependencies among the conceptualizations described in a sentence. There are four primitive conceptual categories from which dependency structures can be built. These are

ACTs	Actions
PPs	Objects (picture producers)
AAs	Modifiers of actions (action aiders)
Pas	Modifiers of PPs (picture aiders)

Rule 1: describes the relationship between an actor and the event he or she causes. This is a two-way dependency since neither actor nor event can be considered primary. The letter p above the dependency link indicates past tense.

Ex: John ran.

Rule 2: describes the relationship between a PP and a PA that is being asserted to describe it. Many state descriptions, such as height, are represented in CD as numeric scales.

Ex: John is tall.

Rule 3: describes the relationship between two PPs, one of which belongs to the set defined by the other.

Ex: John is a Doctor

Rule 4: describes the relationship between a PP and an attribute that has already been predicated of it. The direction of the arrow is toward the PP being described.

Ex: A nice boy.

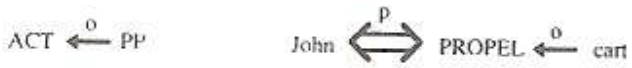
Rule 5: describes the relationship between two PPs, one of which provides a particular kind of information about the other. The three most common types of information to be provided in this way are possession (shown as POSS-BY), location (shown as LOC), and physical containment (shown as CONT). The direction of the arrow is again toward the concept being described.

Ex: John's dog.

5. 

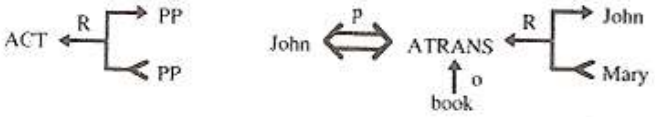
Rule 6: describes the relationship between an ACT and the PP that is the object of that ACT. The direction of the arrow is toward the ACT since the context of the specific ACT determines the meaning of the object relation.

Ex: John pushed the cart.

6. 

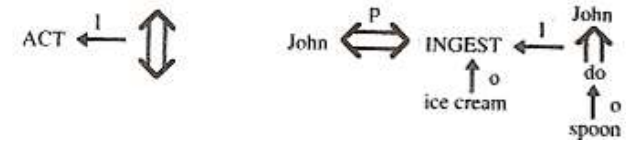
Rule 7: describes the relationship between an ACT and the source and the recipient of the ACT.

Ex: John took the book from Mary.

7. 

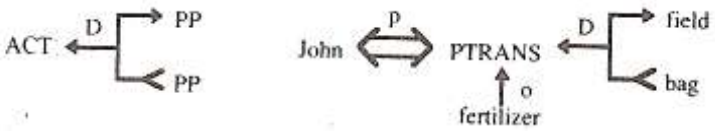
Rule 8: describe the relationship between an ACT and the instrument with which it is performed. The instrument must always be a full conceptualizations (i.e., it must contain an ACT), not just a single physical object.

Ex: John ate ice cream with a spoon.

8. 

Rule 9: describe the relationship between an ACT and its physical source and destination.

Ex: John fertilized the field.

9. 

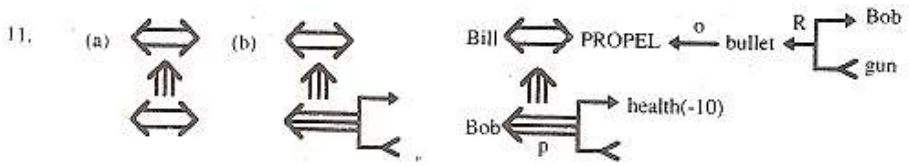
Rule 10: describe the relationship between a PP and a state in which it started and another in which it ended.

Ex: The Plants grew

10. 

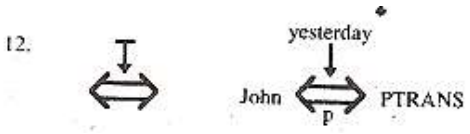
Rule 11: describe the relationship between one conceptualization and another that causes it. Notice that the arrows indicate dependency of one conceptualization on another and so point in the opposite direction of the implication arrows. The two forms of the rule describe the cause of an action and the cause of a state change.

Ex: Bill shot Bob.



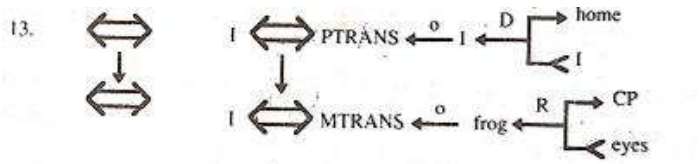
Rule 12: describes the relationship between a conceptualization and the time at which the event it describes occurred.

Ex: John ran yesterday.



Rule 13: describes the relationship between one conceptualization and another that is the time of the first. The example for this rule also shows how CD exploits a model of the human information processing system; see is represented as the transfer of information between the eyes and the conscious processor.

Ex: While going home, I saw a frog



Rule 14: describes the relationship between a conceptualization and the place at which it occurred.

Ex: I heard a frog in the woods.

The set of **conceptual tenses** proposed by SCHANK includes

- **p** – Past
- **f** – Future
- **t** – Transition
- **t_s**– start transition
- **t_f**– finished transition
- **k** – continuing
- **?** – Interrogative
- **/** – Negative
- **Delta** – Timeless
- **c** – Conditional

SCRIPT STRUCTURE

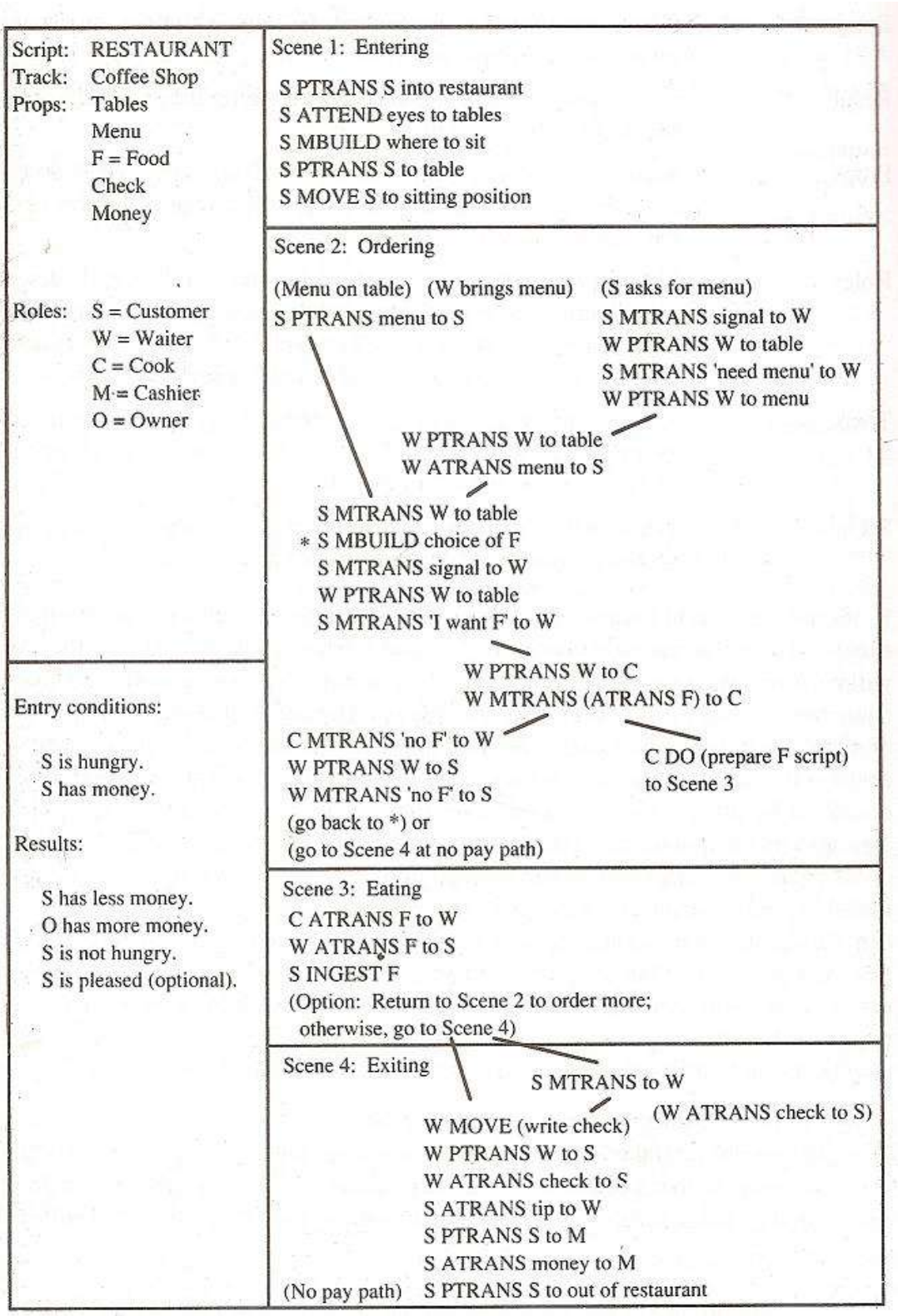
A script is a structure that describes a stereotyped sequence of events in a particular context. A script consists of a set of slots. Associated with each slot may be some information about what kinds of values it may contain as well as a default value to be used if no other information is available. Script and Frame structures are identical. The important components of a Script are:

- Entry Conditions: these must be satisfied before events in the script can occur.
- Result: Conditions that will, in general, be true after the events described in the script have occurred.
- Props: Slots representing objects that are involved in the events described in the script.
- Roles: Persons involved in the events.
- Track: Variations on the script. Different tracks of the same script will share many but not all components.
- Scenes: The actual sequences of events that occur. Events are represented in conceptual dependency formalism.

Example 1: Going to a Theatre

Script : Play in theater	Various Scenes
Track: Play in Theater **Props:** • Tickets • Seat • Play **Roles:** • Person (who wants to see a play) – P • Ticket distributor – TD • Ticket checker – TC **Entry Conditions:** • P wants to see a play • P has a money **Results:** • P saw a play • P has less money • P is happy (optional if he liked the play)	***Scene 1: Going to theater*** • P PTRANS P into theater • P ATTEND eyes to ticket counter ***Scene 2: Buying ticket*** • P PTRANS P to ticket counter • P MTRANS (need a ticket) to TD • TD ATRANS ticket to P ***Scene 3: Going inside hall of theater and sitting on a seat*** • P PTRANS P into Hall of theater • TC ATTEND eyes on ticket POSS_by P • TC MTRANS (showed seat) to P • P PTRANS P to seat • P MOVES P to sitting position ***Scene 4: Watching a play*** • P ATTEND eyes on play • P MBUILD (good moments) from play ***Scene 5: Exiting*** • P PTRANS P out of Hall and theater

Example 2: Going to a Restaurant



Example 3: Robbery in Bank

Script: ROBBERY	*Track: Successful Snatch*
Props: G = Gun, L = Loot, B= Bag, C = Get away car.	*Roles*: R = Robber, M = Cashier, O = Bank Manager, P = Policeman.
Entry Conditions: R is poor. R is destitute.	*Results*: R has more money. O is angry. M is in a state of shock. P is shot.
Scene 1: Getting a gun R PTRANS R into Gun Shop R MBUILD R choice of G R MTRANS choice. R ATRANS buys G (go to scene 2)	
Scene 2 Holding up the bank R PTRANS R into bank R ATTEND eyes M, O and P R MOVE R to M position R GRASP G R MOVE G to point to M R MTRANS "Give me the money or ELSE" to M P MTRANS "Hold it Hands Up" to R R PROPEL shoots G P INGEST bullet from G M ATRANS L to M M ATRANS L puts in bag, B M PTRANS exit O ATRANS raises the alarm (go to scene 3)	
Scene 3: The getaway M PTRANS C	

Advantages:

- Ability to predict events.
- A single coherent interpretation may be build up from a collection of observations.

Disadvantage:

- Less general than frames.

CYC THEORY

The CYC is a theory designed for describing the world knowledge (commonsense knowledge) to be useful in AI applications. The CYC is more comprehensive, where as CD is more specific theory for representing events. CYC is invented by Lenat & Guha for capturing commonsense knowledge. The CYC

structure contains representations of events, objects, attitudes, space, time, motion etc and tends to huge in structure. CYC contains large Knowledge Bases (KBs). Some of the reasons for large KBs are as follows:

1. Brittleness: Specialized knowledge bases are brittle. It is hard to encode new situations and there is degradation in the performance. Commonsense-based knowledge bases should have a firmer foundation.
2. Form & Content: Knowledge representation so far seen may not be sufficient for AI applications where main focus is comprehension. Commonsense strategies could point out where difficulties in content may affect the form and temporarily focus on content of KBs rather than on their form.
3. Shared Knowledge: Small knowledge base system should allow greater communication among themselves with common bases and assumptions.

It is a huge task to build such a large KB. There are some methods & languages in AI which can be used for acquiring this knowledge automatically. Special language based on frame based system is called CYCL using CYC knowledge is encoded. CYCL generalizes the notion of inheritance so that properties can be inherited along any link rather than only "isa" and "instance" links. In addition to frames, CYCL contains a constraint language that allows the expression of arbitrary first-order logical expressions.

CASE GRAMMARS

Case Grammar theory was proposed by the American Linguist Charles J. Fillmore in 1968 for representing linguistic knowledge that removed the strong distinction between syntactic and semantic knowledge of a language. He initially introduced 6 cases (called thematic cases or roles):

- AGENTIVE (Agent)
- OBJECTIVE (Object)
- INSTRUMENTAL (Instrument)
- DATIVE (which covers EXPERIENCER)
- FACTIVE (which covers result of an action)
- LOCATIVE (Location of an action)

The ultimate goal of case grammar theory was to extract deep meanings of sentences & express in the form of cases mentioned above. Different meanings would lead to different case structures, but different syntactic structures with same meaning would map to similar structure. For example, in the sentences, the door was broken by John with hammer, using hammer John broke the door, John broke the door with hammer, the hammer (instrument), John (actor) and the door (object) play the same semantic roles in each of the sentences. Here the act is of "breaking of door" and will have the same case frame.

The case frame contains semantic relation rather than syntactic ones. The semantic roles such as agent, action, object & instrument are extracted from the sentence straightway & stored in case frame which represents

semantic form of a sentence. For example, the sentences such as John ate an apple and An apple was eaten by John, will produce the same case frame. Some of optional cases such as TIME, BENEFICIARY, FROM_LOC (Source), TO_LOC (Destination), CO_AGENT & TENSE are introduced to capture more surface knowledge.

- AGENT (instigator of action)
- Object (Identity on which action is performed, For Eg: The door broke, door is the object)
- Dative (Animate entity affected by the action, For Eg: John killed Mike, Mike is Dative case)
- Experiencer (Animate subject in an active sentence with no agent, For Ex: in sentences John cried, Mike laughs, John & Mike fill Experiencer case)
- Beneficiary (animate who has benefitted by action, For Ex: in the sentence I gave an apple to Mike, Mike is beneficiary case whereas in I gave an apple to Mike for Mary, Mary is Beneficiary case & Mike is Dative case)
- Location (place of action, For Ex: The man was killed in garden, garden is location case & Consider another Ex: John went to school from home, home is Source_Loc case, whereas School is Destination_Loc case)
- Instrument (entity used for performing an action, For Ex: John ate an ice-cream with spoon, spoon is instrument cae)
- Co-Agent (In some situations, where 2 people perform some action together, then second person fills up Co_Agent case. For example, John and Mike lifted box or John lifted box with Mike, Here, John fills Agent case & Mike fills Co_Agent case)
- Time (The time of action, For Ex: John went to market yesterday at 4 O' clock, 4 O clock is Time case)
- Tense (Time of event, i.e.., present, past or future)

Let us generate case frame for a sentence using case structure. The case frame for "John gave an apple to Mike in the kitchen or Mike was given an apple by John in the kitchen" is as follows

Case Frame	
Cases	Values
Action	Give
Agent	John
Objective	Apple
Beneficiary	Mike
Time	Past
Location	Kitchen

Table: Sample Case Frame

5. EXPERT SYSTEM AND APPLICATIONS

INTRODUCTION

Expert systems (ES) are one of the prominent research domains of AI. It is introduced by the researchers at Stanford University, Computer Science Department.

Definition: "The expert systems are the computer applications developed to solve complex problems in a particular domain, at the level of extra-ordinary human intelligence and expertise."

Characteristics:

1. Expertise: An ES should exhibit expert performance, have high level of skill, and possess adequate robustness. The high-level expertise and skill of an ES aids in problem solving & makes the system cost effective.
2. Symbolic Reasoning: Knowledge in an ES is represented symbolically which can be easily reformulated & reasoned.
3. Self Knowledge: A system should be able to explain & examine its own reasoning.
4. Learning Capability: A system should learn from its mistakes & mature as it grows. Flexibility provided by the ES helps it grow incrementally.
5. Ability to provide Training: Every ES should be capable of providing training by explaining the reasoning process behind solving a particular problem using relevant knowledge.
6. Predictive Modelling Power: This is one of the important features of ES. The system can act as an information processing model of problem solving. It can explain how new situation led to the change, which helps users to evaluate the effect of new facts & understand their relationship to the solution.

Advantages: The Expert Systems are capable of the following:

- Advising
- Instructing and assisting human in decision making
- Demonstrating
- Deriving a solution
- Diagnosing
- Explaining
- Interpreting input
- Predicting results
- Justifying the conclusion
- Suggesting alternative options to a problem

Disadvantages: The Expert Systems are incapable of the following:

- Substituting human decision makers
- Possessing human capabilities
- Producing accurate output for inadequate knowledge base
- Refining their own knowledge

PHASES IN BUILDING EXPERT SYSTEMS

- Identify Problem Domain
- Design the System
- Develop the Prototype
- Test & Refine the Prototype
- Develop & Complete the Expert System
- Maintain the System

Identify Problem Domain:

- The problem must be suitable for an expert system to solve it.
- Find the experts in task domain for the ES project.
- Establish cost-effectiveness of the system.

Design the System:

- Identify the ES Technology.
- Know and establish the degree of integration with the other systems and databases.
- Realize how the concepts can represent the domain knowledge best.

Develop the Prototype:

- From Knowledge Base, the knowledge engineer works to
    - Acquire domain knowledge from the expert.
    - Represent it in the form of If-THEN-ELSE rules.

Test & Refine the Prototype:

- The knowledge engineer uses sample cases to test the prototype for any deficiencies in performance.
- End users test the prototypes of the ES.

Develop & Complete the ES:

- Test & ensure the interaction of the ES with all elements of its environment, including end users, databases, and other information systems.
- Document the ES project well.
- Train the user to use ES.

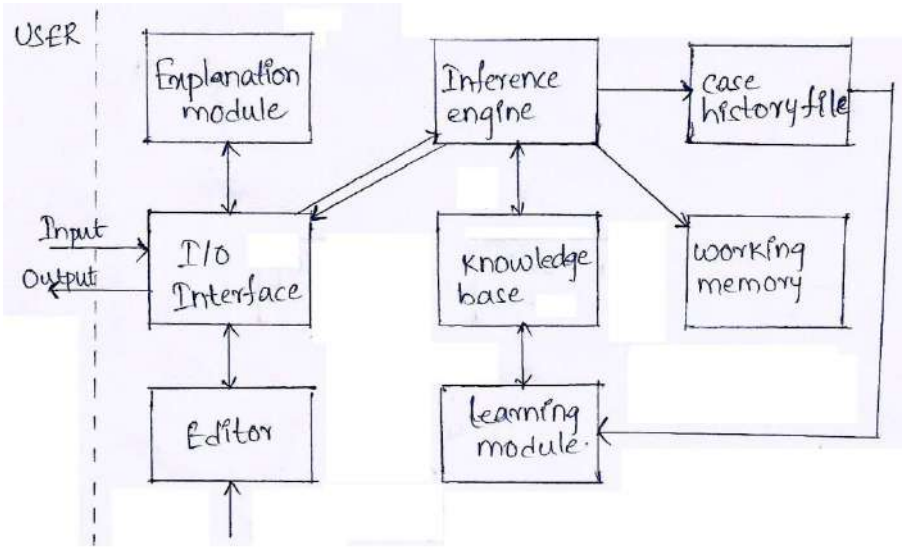
Maintain the System:

- Keep the knowledge base up-to-date by regular review and update.
- Cater for new interfaces with other information systems, as those systems evolve.

EXPERT SYSTEMS VERSUS TRADITIONAL SYSTEMS

Expert System	Traditional System
The entire problem related expertise is encoded in data structures only, not in programs.	Problem expertise is encoded in both program and data structures.
The use of knowledge is vital.	Data is used more efficiently than knowledge.
These are capable of explaining how a particular conclusion is reached and why requested information is needed during a process.	These are not capable of explaining a particular conclusion for a problem. These systems try to solve in a straight forward manner.
Problems are solved more efficiently.	Not as efficient as an Expert System.
It uses the symbolic representations for knowledge i.e. the rules, different forms of networks, frames, scripts etc. and performs their inference through symbolic computations	These are unable to express in symbols. They just simplify the problems in a straight forward manner and are incapable to express the "how, why" questions.
Problem solving tools are present in Expert System	No problem solving tools in specific.
Solution of the problem is more accurate.	Solution of the problem may not be more accurate.
Provide a clear separation of knowledge from its processing.	Do not separate knowledge from the control structure to process this knowledge.
Process knowledge expressed in the form of rules and use symbolic reasoning to solve problems in a narrow domain.	Process data and use algorithms, a series of well-defined operations, to solve general numerical problems.
Permit inexact reasoning and can deal with incomplete, uncertain and fuzzy data.	Work only on problems where data is complete and exact.
Enhance the quality of problem solving by adding new rules or adjusting old ones in the knowledge base. When new knowledge is acquired, changes are easy to accomplish.	Enhance the quality of problem solving by changing the program code, which affects both the knowledge and its processing, making changes difficult.

ARCHITECTURE OF EXPERT SYSTEM (or) COMPONENTS OF EXPERT SYSTEM (or) RULE BASED EXPERT SYSTEMS



I/O Interface:

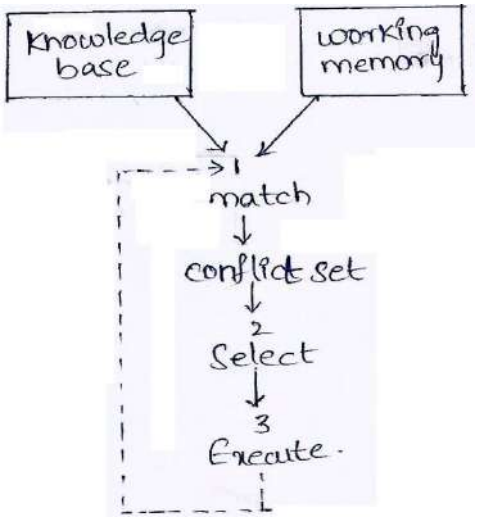
The I/O Interface permits the user to communicate with the system in a simple way.

Knowledge Base:

- The Knowledge Base is one of the important components of Expert System.
- The Knowledge Base consists of a set of production rules & facts (already proven results) along with its solutions.
- For Example, if a student fails to get ≥66% in attendance then declare that student as detained.

Inference Engine:

- Inference Engine is also called as Rule Interpreter.
- It performs the task of matching, from the responses given by the user & the rules.
- Finally it picks up a suitable rule.
- The process is as follows



Explanation Module:

- Explanation Module consists of the Answers for the Questions of 'How' & 'Why'.
- To respond to a 'How' query, the Explanation Module traces the chain of rules.
- To respond to a 'Why' query, the Explanation Module must be able to explain why certain information is needed to complete a step.

Working Memory:

- These are the one which are handled at the current situation by the system.
- It is also called as Temporary Storage Area.

Case History File:

- Case History File consists of all the Input & Output transactions.
- This consists of all the inputs given by various users & the outputs given by the system based upon the input.

Learning Module:

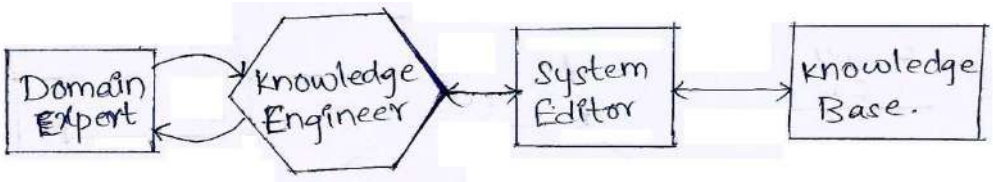
When the data is concerned with Case History File then, the system performs the learning Module.

Editor:

Editor module is used to insert (or) delete the data in the knowledge base.

Knowledge Acquisition & Validation:

One of the most difficult tasks in building knowledge base is in the acquisition & encoding of requisite domain knowledge.



Expert System Shell:

Each Expert System that was build from scratch was done by LISP programming language. After several systems have built in this way it is clear that these systems often use a lot of common things. It was possible to separate the interpreter from the domain specific knowledge & thus to create a system that would be used to construct new Expert Systems by adding new knowledge corresponding to the new problem domain.

One of the important features that a shell must provide is an easy way to interface between an Expert System & a programming environment.

Example: Graphical games.

MYCIN Expert System:

MYCIN is one of the oldest Expert Systems. It was developed at Stanford University in 1970s. It was developed as an ES that could diagnose & recommend treatment for certain blood infections. MYCIN was

developed for exploring the ways in which human experts make guesses on the basis of partial information. In MYCIN, the knowledge is represented as a set of if-then rules. In MYCIN rule can be written in English as follows

- IF the infection is primary-bacteraemia AND the site of the culture is one of the sterile sites AND the suspected portal of entry is the gastrointestinal tract, THEN there is suggestive evidence (0.7) that infection is bacteriod

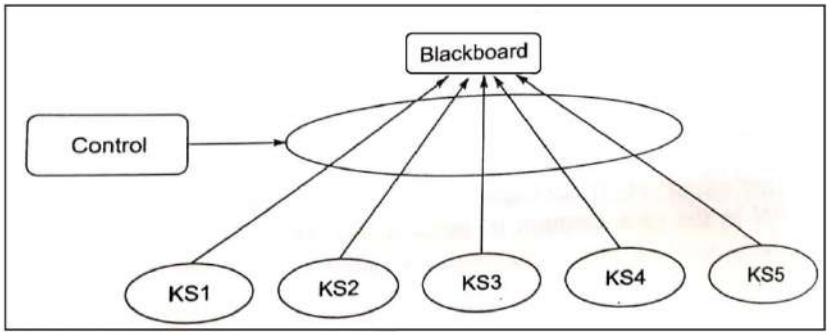
The 3 stages of conversation with MYCIN:

- In First stage, initial data regarding the case was gathered to enable the system to come up with a very broad diagnosis.
- In Second stage, the questions that were asked to test specific hypotheses were more direct in nature.
- In Final stage, a diagnosis was proposed.

The First version of MYCIN had a number of problems which were remedied in later. One of them was that the rules of domain knowledge were often mixed. EMYCIN was the first Expert shell developed from MYCIN. A new ES called PUFF was developed using EMYCIN in the new domain of heart disorders. To train Doctors, the system called NEOMYCIN was developed, which takes them through various sample cases, check their diagnoses, determines whether their conclusions are right & explains where they went wrong.

BLACK BOARD SYSTEMS

Blackboard Systems were first developed in the 1970s to solve complex, difficult & ill-structured problems in a wide range of application areas. Blackboard architecture is a way of representing & controlling the knowledge bases; using independent groups of rules called Knowledge Sources (KSs) that communicate through a data control database called a Blackboard.



The main modules of Blackboard System are as follows. It consists of 3 main components:

- Knowledge Sources
- Blackboard
- Control Shell

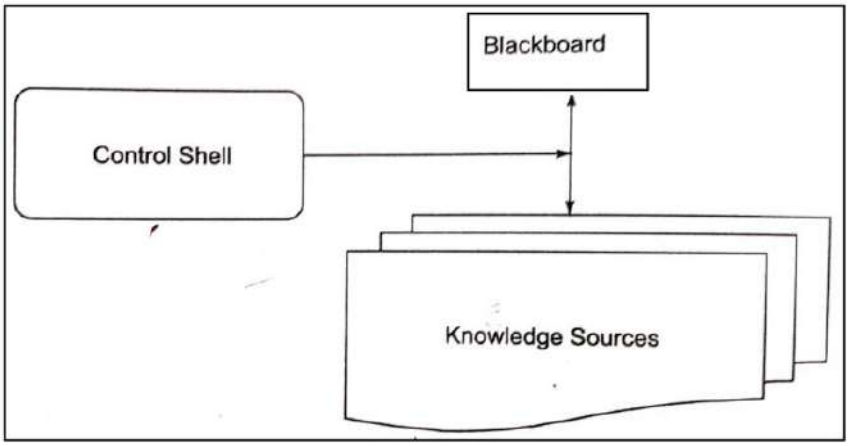


Fig: Blackboard System

Knowledge Sources:

- A KS is regarded to be a specialist at solving certain aspects of the overall application and is separate and independent of all other KSs in the blackboard system.
- Once it obtains the information required by it on the Blackboard, it can proceed further without any assistance from other KSs.
- It is possible to add additional KSs to the Blackboard system and upgrade or even remove existing KSs.
- Each KS is aware of the conditions under which it can contribute toward solving a particular problem. This knowledge in problem solving is known as triggering condition.

Blackboard:

- Blackboard represents a global data repository & shared data structure available to all KSs.
- It contains several important constituents such as raw input data, partial solutions, alternatives & final solutions, control information, communication medium used in various phrases in problem solving.
- An advantage of the blackboard system is that the system can retain the results of problems that have been solved earlier, thus avoiding the task of re-computing them later.

Control Component:

- The control Shell, directs the problem-solving process by allowing KSs to respond to changes made to the black board.
- In a classical blackboard system control approach, the currently executing KS Activation (KSA) generates events as it makes changes on the blackboard.
- These events are ranked & maintained until the executing KSA is completed.

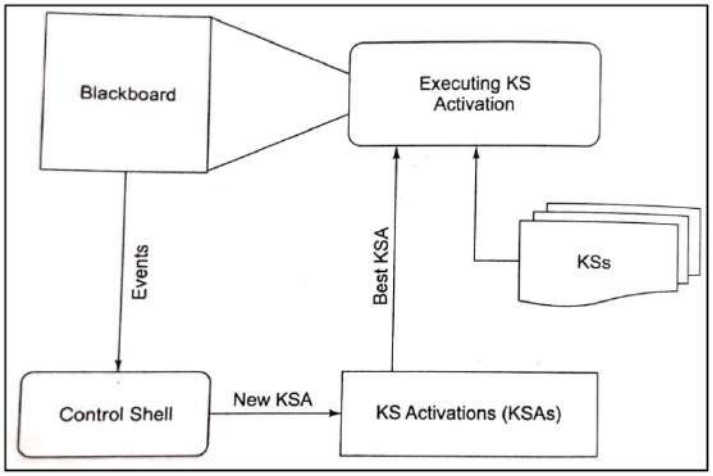


Fig: Blackboard System Control Cycle

Issues in Blackboard Systems for Problem Solving:

- Specialized representation: KSs may only operate on a few classes of blackboard objects.
- Fully general representation: All aspects of blackboard data are understood by all KSs.

Blackboard System versus Rule-based System:

A major difference between a Blackboard System and a Rule-based System lies in the size & scope of rules when compared to the size & complexity of KSs.

TRUTH MAINTENANCE SYSTEMS

Truth Maintenance System (TMS) is a structure which helps in revising set of beliefs & maintaining the truth every time new information contradicts information already present in the system. It is developed by Doyle in 1979. TMS maintains the beliefs for general problem-solving systems. All TMS manipulate proposition symbols & the relationships between different proposition symbols.

- A Monotonic TMS manipulates propositional symbols & Boolean constraints.
- A Non-Monotonic TMS allows for heuristic (or) non-monotonic relationships between proposition symbols such as ‘whenever P is true Q is likely’ (or) ‘if P is true then unless there is evidence to the contrary Q is assumed to be true’.

Monotonic System & Logic:

In Monotonic Systems, once a fact (or) piece of knowledge stored in the knowledge base is identified, it cannot be changed during the process of reasoning. In other words, axioms are not allowed to change as once a fact is confirmed to be true, it must always remain true & can never be modified.

- If a formula is a theorem for a particular formal theory, then that formula remains a theorem for any augmented theory obtained by adding axioms to the theory.

For instance, if a property P is a theorem of T & if T is augmented to T_1 by additional axioms, then P remains a theorem of T_1. Further, if an axiom A is added to a theory T to build a theory T_1, then all the theorems of T are also theorems of T_1.

In Monotonic Reasoning, the world of axioms continually increases in size & keeps on expanding. An example of monotonic form of reasoning is predicate logic.

Non-Monotonic System & Logic:

In non-monotonic systems, truths that are present in the system can be retracted whenever contradictions arise. Hence, the number of axioms can increase as well as decrease. The system is continually updated depending upon the changes in knowledge base. In non-monotonic logic, if a formula is a theorem for a formal theory, then it need not be theorem for an augmented theory. Common sense reasoning is an example of non-monotonic reasoning.

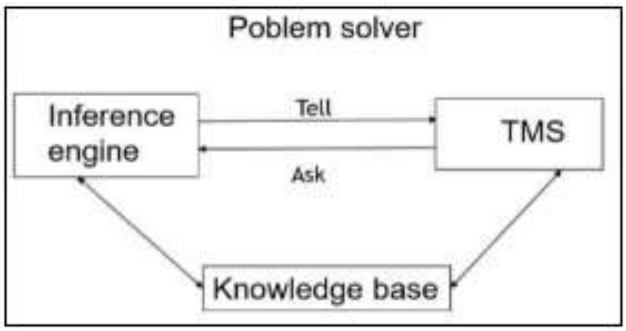


Fig: Interaction between different components in Problem Solver

- The term truth maintenance is synonymous with the term knowledge base maintenance and is defined as keeping track of the interrelations between assertions in knowledge base.
- The main job of TMS is to maintain consistency of the knowledge being used by problem solvers.
- The Inference Engine (IE) solves domain problems based on its current belief set.

Monotonic TMS:

A Monotonic TMs is a general facility for manipulating Boolean constraints on proposition symbols. The constraint has the form $P \rightarrow Q$ where P & Q are proposition symbols.

Functionality of Monotonic TMS:

- Add_constraint
- Follow_from
- Interface functions

Add_constraint: This interface functions adds a constraint to the internal constraint set. Once a constraint has been added, it can never be removed.

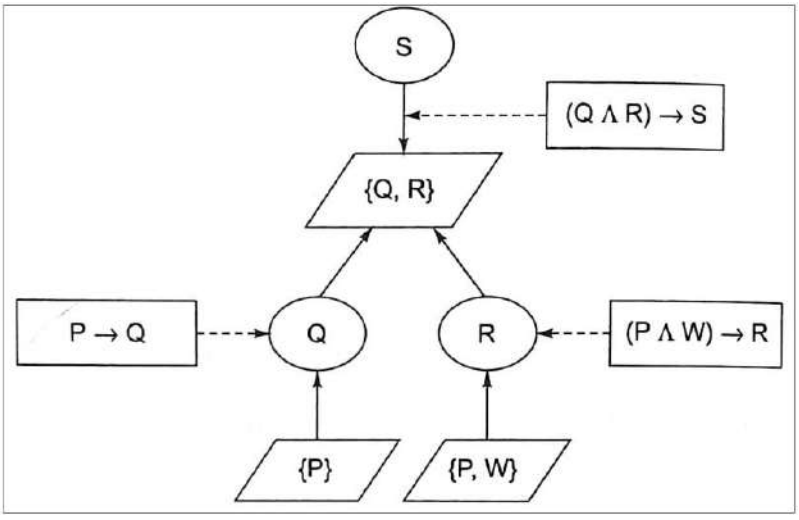
Follow_from: This function takes 2 arguments namely, a literal (L) and a premise set (Σ) and returns the values yes, no or unknown. If yes is returned, then the TMS guarantees that L follows from the Σ and internal constraints. If no is returned, then the TMS guarantees that L does not follows from the Σ and internal constraints. If TMS is unable to determine, then it returns unknown.

Interface Functions: these compute justifications. There are 2 interface functions used to generate such proofs namely justifying literals & justifying constraints.

Let us consider an example with a premise set Σ = {P, W} and an internal constraint set {P → Q, (P Λ W) → R, (Q Λ R) → S}.

Justification Literals	Derived Literals	Justifying Constraints
{P, W}	R	(P Λ W) → R
{P}	Q	P → Q
{Q, R}	S	(Q Λ R) → S

The Justification Tree is as follows



Non-monotonic TMS:

The basic operation of a TMS is to attach a justification to a fact. A fact can be linked with any component of program knowledge which is to be connected with other components of program information.

Support List is defined as SL (IN-node) (OUT-node), where IN-node represents a list of all IN-nodes (propositions) that support the considered node as true.

- Here, IN means that the belief is true. OUT-node is a list of all OUT-nodes that do not support the considered node as true.
- OUT means that the belief is not true for considered node.

For example

Node Number	Facts/assertions	Justification (justified belief)
1	It is sunny	SL(3) (2,4)
2	It rains	SL() ()
3	It is warm	SL(1) (2)
4	It is night time	SL() (1)

Table: Justification of Facts

In this case

- An empty list indicates that its justification does not depend on the current belief or disbelief.
- Node 1 assumes that it is sunny, provided that it is warm and it does not rain, and it is not night time.
- Node 2 has both empty lists indicating that its justification does not depend on current beliefs or disbeliefs.
- Node 3 assumes that it is warm given that it is sunny & it does not rain.
- Node 4 does not depend on the list in its (IN-node) part.

APPLICATIONS OF EXPERT SYSTEMS

Application	Description
Design Domain	Camera lens design, automobile design.
Medical Domain	Diagnosis Systems to deduce cause of disease from observed data, conduction medical operations on humans.
Prediction	Perform the task of inferring the likely consequences of a situation like weather prediction for rains, storms, prediction of crops, share market etc.
Monitoring Systems	Comparing data continuously with observed system or with prescribed behavior such as leakage monitoring in long petroleum pipeline.
Process Control Systems	Controlling a physical process based on monitoring.
Knowledge Domain	Finding out faults in vehicles, computers.
Finance/Commerce	Detection of possible fraud, suspicious transactions, stock market trading, Airline scheduling, cargo scheduling.

LIST OF SHELLS & TOOLS

The following are the list of popular shells & tools

- ACQUIRE
- ARITY
- ART
- CLIPS

ACQUIRE: It is primarily a knowledge-acquisition system & an ES shell, which provides a complete development environment for the building & maintenance of knowledge-based applications. ACQUIRE SDK is a software development kit, which provides callable libraries for systems such as MS-DOS, Windows, Windows NT, Windows 95 and Win 32.

ARITY: Expert Development Package is an ES that was developed by ARITY Corporation.

ART: It is called Automated Reasoning Tool that is an ES shell, which is based on LISP. It supports rule-based reasoning, hypothetical reasoning and case-based reasoning.

CLIPS: It is used to represent C Language Integrated Production System. It is a public domain software tool that is used for building Expert Systems. CLIPS is probably the most widely used ES tool because it is fast, efficient and free.

FLEX: Flex is a hybrid ES which is implemented in PROLOG. It supports forward & backward chaining, multiple inheritance & also possesses an automatic question & answer system.

GENSYM'S G2: It offers a graphical, object oriented environment for the creation of intelligent applications that are able to monitor, diagnose & control dynamic events in various environments.

GURU: It is an ES developed environment & offers a wide variety of information processing tools combined with knowledge-based capabilities such as forward chaining, backward chaining, mixed chaining, multi-value variables & fuzzy reasoning.

HUGIN SYSTEM: It is a software package for construction of model-based Expert System. It is easy-to-use.

KNOWLEDGE CRAFT: It is an ES development tool kit for scheduling, design & configuration applications.

K- VISION: It is a knowledge acquisition & visualization tool. It runs on Windows, DOS & UNIX workstations.

MAILBOT: It is a personal e-mail agent that reads an e-mail message on standard input & creates an e-mail reply to be sent to the sender of the original message. It provides filtering, forwarding, notification & automatic question-answering capabilities.

TMYCIN: It is an acronym for Tiny EMYCIN & is an ES shell modeled after the EMYCIN shell that was developed at Stanford. It is especially useful for student exercises, although real ES have been written using it. TMYCIN is written in common LISP & is fairly small.

6. UNCERTAINTY MEASURE: PROBABILITY THEORY

PROBABILISTIC REASONING

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty. We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.

In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today," "behavior of someone for some situations," "A match between two teams or two players." These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Need of probabilistic reasoning in AI:

- When there are unpredictable outcomes.
- When specifications or possibilities of predicates becomes too large to handle.
- When an unknown error occurs during an experiment.

In probabilistic reasoning, there are two ways to solve problems with uncertain knowledge:

- Bayes Rule
- Bayesian Statistics

Probability: Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

- $0 \leq P(A) \leq 1$, where P(A) is the probability of an event A.
- P(A) = 0, indicates total uncertainty in an event A.
- P(A) =1, indicates total certainty in an event A.

We can find the probability of an uncertain event by using the below formula.

$$\textbf{Probability of occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

- P(¬A) = probability of a not happening event.
- P(¬A) + P(A) = 1.

Event: Each possible outcome of a variable is called an event.

Sample space: The collection of all possible events is called sample space.

Random variables: Random variables are used to represent the events and objects in the real world.

Prior probability: The prior probability of an event is probability computed before observing new information.

Posterior Probability: The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.

Conditional probability: Conditional probability is a probability of occurring an event when another event has already happened. Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

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

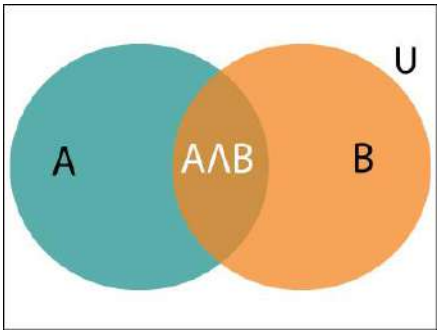
Where, P(A∧B)= Joint probability of A and B

P(B)= Marginal probability of B.

If the probability of A is given and we need to find the probability of B, then it will be given as:

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

It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of P(A∧B) by P(B).



Example: In a class, there are 70% of the students who like English and 40% of the students who likes English and mathematics, and then what is the percent of students those who like English also like mathematics?

Solution:

Let, A is an event that a student likes Mathematics.

B is an event that a student likes English.

$$P(A|B) = \frac{P(A \wedge B)}{P(B)} = \frac{0.4}{0.7} = 57\%$$

Hence, 57% are the students who like English also like Mathematics.

BAYES THEOREM

Bayes' theorem is also known as Bayes' rule, Bayes' law, or Bayesian reasoning, which determines the probability of an event with uncertain knowledge. In probability theory, it relates the conditional probability and marginal probabilities of two random events. Bayes' theorem was named after the British mathematician Thomas Bayes. The Bayesian inference is an application of Bayes' theorem, which is fundamental to Bayesian statistics.

In monotonic reasoning, at any given moment, the statement is either believed to be true, believed to be false or not believed to be either. In statistical reasoning, it is useful to be able to describe belief's that are not certain but for which there is some supporting evidence.

When we have uncertain knowledge one way to express confidence about an event is through probability that expresses the chance of happening or not happening. The general characteristics of probability theory is

1. Probability of a statement is always ≥0 =0(total uncertainty) and ≤1 (=1 total certainty)
2. Probability of a sure proposition is unity (1).
3. P (A U B) = P (A) + P (B), if A & B are mutually exclusive.
4. P (~A) = 1- P (A).

The fundamental notation of Bayesian statistics is that of conditional probability P (H/E), i.e., the probability of Hypothesis H, given that we have observed evidence E.

P (Hi/E) = the probability that hypothesis Hi is true given evidence E.

P (E/Hi) = the probability that we will observe evidence E given that hypothesis i is true.

P (Hi) = the a priori probability the hypothesis i is true in the absence of any specific evidence. These probabilities are called prior probabilities or priors.

k = the number of possible Hypotheses. Bayes theorem states that

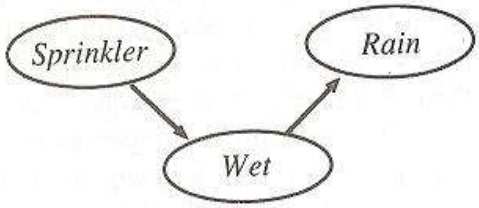
$$P(H_i|E) = \frac{P(E|H_i) \cdot P(H_i)}{\sum_{n=1}^{k} P(E|H_n) \cdot P(H_n)}$$

BAYESIAN BELIEF NETWORKS

Rule 1: If the sprinkler was on last night, then there is suggestive evidence 0.9, that the grass will be wet this morning.

Rule 2: If the grass is wet this morning then there is suggestive evidence 0.8 that is rained last night.

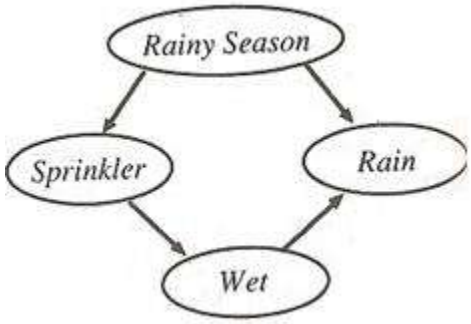
We can draw the network above two rules. These networks represent interaction among events. The main idea is that to describe the real world, it is not necessary to use a huge joint probability in which we list the probabilities of all the conceivable combination of events.



But, here the constraints flow out, incorrectly from sprinkler 'on' to rain last night. There are two different ways that propositions can influence the likelihood of each other.

1. That causes influence the likelihood of their symptoms.
2. That observing a symptom affects the likelihood of all of its possible causes.

Bayesian network structure makes a clear distinction between these two kinds of influence. For this, we construct a directed acyclic graph that represents casualty relationships among variables. The variables may be propositional or take on values of some other type. A casualty graph for the wet graph example is



The new node tells whether it is currently the rainy season or not. The graph shows the casualty relationship that occurs among its nodes. In order to use this as a basis for probabilistic reasoning we need conditional probability also.

Attribute	*Probability*
$p(Wet\|Sprinkler, Rain)$	0.95
$p(Wet\|Sprinkler, \neg Rain)$	0.9
$p(Wet\|\neg Sprinkler, Rain)$	0.8
$p(Wet\|\neg Sprinkler, \neg Rain)$	0.1
$p(Sprinkler\|RainySeason)$	0.0
$p(Sprinkler\|\neg RainySeason)$	1.0
$p(Rain\|RainySeason)$	0.9
$p(Rain\|\neg RainySeason)$	0.1
$p(RainySeason)$	0.5

For example, to represent the casual relationships between the propositional variables x1, x2….x6 from the following figure, one can write the joint probability P (x1, x2……x6) by inspection of as a product of conditional probabilities.

CERTAINTY FACTOR THEORY

This approach was adopted in the 'MYCIN' System. MYCIN is an expert system that can recommend appropriate therapies for patients with bacterial infection. It interacts with the physician to acquire the clinical data it needs.

MYCIN represents its diagnostic knowledge as a set of rules. Each rule has associated with a certainty factor CF, which is a measure of the extent to which the evidence that is described by the antecedent of the rule, supports the conclusion that is given in the rule's consequent. If

1. The stain of the organism is gram-positive and
2. The Morphology of the organism is coccus and
3. The growth confirmation of the organism is clumps then there is suggestive evidence (0.7) that the identity of the organism is staphylococcus.

This means if the antecedents 1, 2, 3 are 100% certain that the identity of the organism only 70% certain to be staphylococcus.

A certainty factor CF [h, e] is defined in two terms.

1. MB [h, e]: A measure of believe in hypothesis h given the evidence e. i.e. MB measures the extent to which the evidence supports the hypothesis. It is 0, if the evidence fails to support the hypothesis.
2. MD [h, e]: A measure of disbelieve in hypothesis h given the evidence e. i.e. MD measures the extent to which the evidence supports the negation of hypothesis. If it is 0, the evidence supports the hypothesis.

From these two we can define the certainty factors **CF [h, e] = MB [h, e] - MD [h, e**]

When multiple pieces of evidence and multiple rules to be applied to a problem, these certainty factors need to be combined. We have the following:

1. Several rules A, B provide evidence to a single hypothesis C
2. Consider a believe, when several propositions are taken together
3. The output of one rule provides the input to another.

<u>Several rules provide evidence to a single hypothesis</u>:

The measures of believe and disbelieve of a hypothesis given two observations s1 and s2 are computed from

$$MB[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MD[h, s_1 \wedge s_2] = 1 \\ MB[h, s_1] + MB[h, s_2] \cdot (1 - MB[h, s_1]) & \text{otherwise} \end{cases}$$

$$MD[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MB[h, s_1 \wedge s_2] = 1 \\ MD[h, s_1] + MD[h, s_2] \cdot (1 - MD[h, s_1]) & \text{otherwise} \end{cases}$$

Certainty factor of a combination of Hypothesis:

The formulas MYCIN uses for the MB of the conjunction and the disjunction of two Hypotheses are:

$$MB [h1 \wedge h2, e] = \min [MB [h1, e] + MB [h2, e]]$$

$$MB [h1 \vee h2, e] = \max [MB [h1, e] + MB [h2, e]]$$

MD can be computed analogously.

Ex: Consider the production rule, if there is enough fuel in the vehicle and the ignition system is working correctly and the vehicle does not start then fault lies in the fuel flow (CF = 0.75)

Every 'if' part is known with 100% certainty then consequence CF is 0.75. If the 'if' part is not known with the 100% certainty then the following rules are used to estimate the value of MB and MD

$$MB [h, e] = MB1 [h, e] + \max [0, CF [h, e]]$$

$$MD [h, e] = MD1 [h, e] + \max [0, CF [h, e]]$$

Whenever, if you want to solve the problem using the third rule, these rules are also adapted.

1. The CF of the conjunction of several facts is taken to be the minimum of CF's of the individual facts.
2. The CF for the conclusion is obtained by multiplying the CF of the rule with the minimum CF of the 'if' part.
3. The CF for a fact produced as the conclusions of one or more rules is the maximum of the CF's produced.

Rule 1: If p & q & r then z CF = 0.65

Rule 2: If u & v & w then z CF = 0.7

p	q	r	u	v	w
0.6	0.45	0.3	0.7	0.5	0.6

I from rule 1: min [p, q, r] = 0.3

II from rule 2: CF [Z] = 0.3 * 0.65 = 0.195

I from rule 2: min [u, v, w] = 0.5

II from rule 2: CF (Z) = 0.5 * 0.7 =0.35

DEMPSTER-SHAFER THEORY

Draw backs in Bayesian theory as a model of uncertain reasoning:

1. Probabilities are described as a single numeric value; this can be distortion of the precision that is actually available for supporting evidence.

Ex: There is a probability 0.7, that the dollar will compact to the Japanese yen over the next 6 months, when we say like this; the probability chance is 0.6 or 0.8

2. There is nowhere to differentiate between ignorance and uncertainty, these are distinct different concepts and should be treated as same concepts in our Bayesian theory.

3. Belief and disbelief are functional opposites w. r. to the classical probability theory. If p (A) =0.3 then P (~A) =0.7. But in our certainty factor concept measure of believe is 0.3 then measure of disbelief is 0, this assignment is conflicting.

To avoid this problems Aurther Dempster and has student Glenn Shafer proposed Dempster Shafer theory. Here considering the sets of propositions, we assign an interval

[Belief, Plausibility]

i.e., Belief denotes Bel which measures strength of evidence in favor of a set of propositions. Plausibility is denoted by Pl which measures the extent to which evidence in favor of ~S leaves room for belief in S. i.e.,

Pl (S) = 1- Bel (~S)

This ranges from 0 to 1. If you have certain evidence in favor of ~S then Bel (~S) will be 1 and Pl (S) will be 0. Initially Bel is a measure of our belief in some hypothesis given some evidence. We need to start with an exhaustive universe of mutually exclusive hypothesis. We will call this as the frame of discernment and we will write it as Θ. For example, in diagnosis problem might consist of set {All, Flu, Cold, Pneu}:

All: allergy

Flu: flu

Cold: cold

Pneu: Pneumonia

1. Our goal is to attach, some measure belief elements to Θ
2. Not all evidences is directly support on individual elements.
3. Often it supports sets of elements (i.e., subsets of Θ)

Here fever might supports {All, Flu, Cold, Pneu}. Since the elements Θ are mutually exclusive, evidence in favor of some may have an effect all over belief others.

In a purely Bayesian systems, we can handle the both of the phenomenon, by visiting all of the combinations of conditional probabilities. But Dempster Shafer theory handles interactions by manipulating sets of hypothesis directly. We use m- the probability density function define for all subsets of Θ including singleton subsets i.e., individual elements.

m (P) – measures the amount of belief that is currently assigned to exactly the set P (hypothesis)

If Θ contains n elements, then there are 2^n subsets of Θ. We must assign m, so that the sum of all the m values assigned to the subsets of Θ is 1.

1. Initially, suppose we have no evidence for anyone of the four hypotheses.

We define m as { Θ } = 1.0

All other values of m are 0.

2. Suppose the evidence is fever (at a level of 0.6). so the correct diagnosis is in the set {All, Flu, Cold, Pneu}

{All, Flu, Cold, Pneu}=0.6

{ Θ } = 0.4

Now Bel (P) is the sum of the values of m for the set P and for all of its subsets. Thus Bel (P) is over all belief that the correct answer lies somewhere in the set P.

This theory can combine any two belief functions, whether they represent multiple sources of evidence for a single hypothesis, or multiple sources of evidence for different hypothesis. Suppose m1, m2 are two belief functions.

X – sets of subsets of Θ to which m1 assigns a non zero value.

Y – Corresponding set for m2

then the combination of m3 of m1 and m2 is

$$m_3(Z) = \frac{\sum_{X \cap Y = Z} m_1(X) \cdot m_2(Y)}{1 - \sum_{X \cap Y = \emptyset} m_1(X) \cdot m_2(Y)}$$

We can apply this any set Z (Θ)

Case 1: When intersection of X and Y generate non-empty sets, suppose m1 is our belief after observing fever

$\{Flu, Cold, Pneu\}$	(0.6)
Θ	(0.4)

Case 2: m2 is our belief after observing running nose

$\{All, Flu, Cold\}$	(0.8)
Θ	(0.2)

Now we can compute m3 from the combination of m1 and m2 is

		$\{A, F, C\}$	(0.8)	Θ	(0.2)
$\{F, C, P\}$	(0.6)	$\{F, C\}$	(0.48)	$\{F, C, P\}$	(0.12)
Θ	(0.4)	$\{A, F, C\}$	(0.32)	Θ	(0.08)

Since, there are no empty factors then the scaling factor is 1 in case 1.

Case 2: When empty sets are generated in this case, after producing m3.

We consider

$\{Flu, Cold\}$	(0.48)
$\{All, Flu, Cold\}$	(0.32)
$\{Flu, Cold, Pneu\}$	(0.12)
Θ	(0.08)

Now suppose that m4 is our belief, that patient goes on a trip.

$\{All\}$	(0.9)
Θ	(0.1)

We can apply the numerator of the combination rule to produce

		$\{A\}$	(0.9)	Θ	(0.1)
$\{F,C\}$	(0.48)	$\emptyset$	(0.432)	$\{F,C\}$	(0.048)
$\{A,F,C\}$	(0.32)	$\{A,F,C\}$	(0.288)	$\{A,F,C\}$	(0.032)
$\{F,C,P\}$	(0.12)	$\emptyset$	(0.108)	$\{F,C,P\}$	(0.012)
Θ	(0.08)	$\{A\}$	(0.072)	Θ	(0.008)

Scaling factor = 1- (0.432+0.108) =1- 0.540 =0.46.

Total Belief for Θ is 0.46 is associated with outcome, that are in fact impossible.

FUZZY SETS AND FUZZY LOGIC

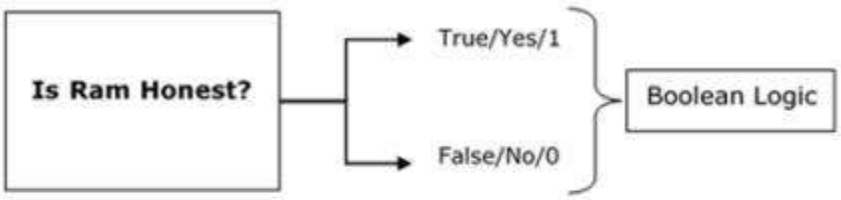
FUZZY LOGIC

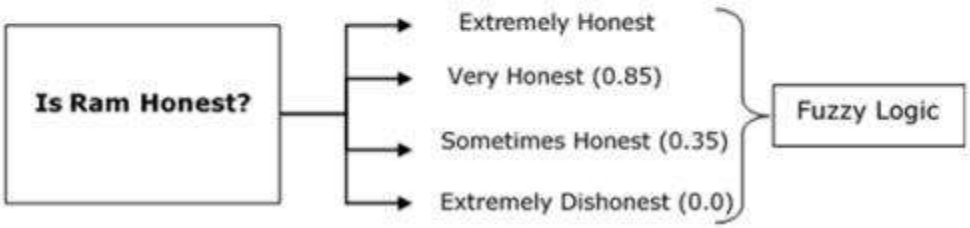
The word fuzzy refers to things which are not clear or are vague. Any event, process, process, or function that is changing continuously cannot always be defined as either true or false, which means that we need to define such activities in a Fuzzy manner.

What is Fuzzy Logic?

Fuzzy Logic resembles the human decision-making methodology. It deals with vague and imprecise information. This is gross oversimplification of the real-world problems and based on degrees of truth rather than usual true/false or 1/0 like Boolean logic.

Take a look at the following diagram. It shows that in fuzzy systems, systems, the values are indicated by a number in the range from 0 to 1. Here 1.0 represents absolute truth and 0.0 represents absolute falseness. The number which indicates the value in fuzzy systems is called the truth value.





In other words, we can say that fuzzy logic is not logic that is fuzzy, but logic that is used to describe fuzziness. There can be numerous other examples like this with the help of which we can understand the

concept of fuzzy logic. Fuzzy Logic was introduced in 1965 by Lofti A. Zadeh in his research paper "Fuzzy Sets". He is considered the father of Fuzzy Logic.

Fuzzy Logic - Classical Set Theory:

A set is an unordered collection of different elements. It can be written explicitly by listing its elements using the set bracket. If the order of the elements is changed or any element of a set is repeated, it does not make any changes in the set.

Example:

- A set of all positive integers.
- A set of all the planets in the solar system.
- A set of all the states in India.
- A set of all the lowercase letters of the alphabet.

Mathematical Representation of a Set: Sets can be represented in two ways

- Roster or Tabular Form
- Set Builder Notation

1. Roster (or) Tabular Form: In this form, a set is represented by listing all the elements comprising it. The elements are enclosed within braces and separated by commas. Following are the examples of set in Roster or Tabular Form:
    - Set of vowels in English alphabet, A = {a,e,i,o,u}
    - Set of odd numbers less than 10, B = {1,3,5,7,9}

2. Set Builder Notation: In this form, the set is defined by specifying a property that elements of the set have in common. The set is described as A = {x:p(x)}

    Example 1: The set {a,e,i,o,u} is written as

    $$A = \{x:x \text{ is a vowel in English alphabet}\}$$

    Example 2: The set {1,3,5,7,9} is written as

    $$B = \{x:1 \leq x < 10 \text{ and } (x\%2) \neq 0\}$$

    If an element x is a member of any set S, it is denoted by $x \in S$ and if an element y is not a member of set S, it is denoted by $y \notin S$.

    Example: If S = {1,1.2,1.7,2}, $1 \in S$ but $1.5 \notin S$

Cardinality of a Set:

Cardinality of a set S, denoted by |S||S|, is the number of elements of the set. The number is also referred as the cardinal number. If a set has an infinite number of elements, its cardinality is ∞∞.

Example: |{1,4,3,5}| = 4,|{1,2,3,4,5,…}| = ∞

If there are two sets X and Y, |X| = |Y| denotes two sets X and Y having same cardinality. It occurs when the number of elements in X is exactly equal to the number of elements in Y. In this case, there exists a bijective function 'f' from X to Y.

- |X| ≤ |Y| denotes that set X's cardinality is less than or equal to set Y's cardinality. It occurs when the number of elements in X is less than or equal to that of Y. Here, there exists an injective function 'f' from X to Y.
- |X| < |Y| denotes that set X's cardinality is less than set Y's cardinality. It occurs when the number of elements in X is less than that of Y. Here, the function 'f' from X to Y is injective function but not bijective.
- If |X| ≤ |Y| and |X| ≤ |Y| then |X| = |Y|. The sets X and Y are commonly referred as equivalent sets.

TYPES OF SETS

Sets can be classified into many types; some of which are finite, infinite, subset, universal , proper subset, singleton set, empty (or) null, single (or) unit, equal, equivalent, overlapping, disjoint.

Finite Set: A set which contains a definite number of elements is called a finite set.

Example: S = {x|x ∈ N and 70 > x > 50}

Infinite Set: A set which contains infinite number of elements is called an infinite set.

Example: S = {x|x ∈ N and x > 10}

Subset: A set X is a subset of set Y (Written as X ⊆ Y) if every element of X is an element of set Y.

Example 1: Let, X = {1,2,3,4,5,6} {1,2,3,4,5,6} and Y = {1,2}. Here set Y is a subset of set X as all the elements of set Y is in set X. Hence, we can write Y⊆X.

Example 2: Let, X = {1,2,3} {1,2,3} and Y = {1,2,3}. {1,2,3}. Here set Y is a subset (not a proper subset) subset) of set X as all the elements of set Y is in set X. Hence, we can write Y⊆X.

Proper Subset: The term "proper subset" can be defined as "subset of but not equal to". A Set X is a proper subset of set Y (Written as X ⊂ Y) if every element of X is an element of set Y and |X| < |Y|.

Example: Let, X = {1,2,3,4, {1,2,3,4,5,6} and Y = {1,2}. Here set Y ⊂ X, since all elements in Y are contained in X too and X has at least one element which is more than set Y.

Universal Set: It is a collection of all elements in a particular context or application. All the sets in that context or application are essentially subsets of this universal set. Universal sets are represented as U.

Example: We may define U as the set of all animals on earth. In this case, a set of all mammals is a subset of U, a set of all fishes is a subset of U, a set of all insects is a subset of U, and so on.

Empty (or) Null Set: An empty set contains no elements. It is denoted by Φ. As the number of elements in an empty set is finite, empty set is a finite set. The cardinality of empty set or null set is zero.

Example: $S = \{x|x \in N \text{ and } 7 < x < 8\} = \Phi$

Singleton Set (or) Unit Set: A Singleton set or Unit set contains only one element. A singleton set is denoted by {s}.

Example: $S = \{x|x \in N, 7 < x < 9\} = \{8\}$

Equal Set: If two sets contain the same elements, they are said to be equal.

Example: If A = {1,2,6} {1,2,6} and B = {6,1,2}, {6,1,2}, they are equal as every element of set A is an element of set B and every element of set B is an element of set A.

Equivalent Set: If the cardinalities of two sets are same, they are called equivalent sets.

Example: If A = {1,2,6} and B = {16,17,22}, they are equivalent as cardinality of A is equal to the cardinality of B. i.e. |A| = |B| = 3.

Overlapping Set: Two sets that have at least one common element are called overlapping sets. In case of overlapping sets:

$$n(A \cup B) = n(A) + n(B) - n(A \cap B)$$

$$n(A \cup B) = n(A - B) + n(B - A) + n(A \cap B)$$

$$n(A) = n(A - B) + n(A \cap B)$$

$$n(B) = n(B - A) + n(A \cap B)$$

Example: Let, A = {1, 2, 6} and B = {6, 12, 42}. There is a common element '6', hence these sets are overlapping sets.

Disjoint Sets: Two sets A and B are called disjoint sets if they do not have even one element in common. Therefore, disjoint sets have the following properties:

$$n(A \cup B) = n(A) + n(B) - n(A \cap B)$$

$$n(A \cup B) = n(A - B) + n(B - A) + n(A \cap B)$$

$$n(A) = n(A - B) + n(A \cap B)$$

$$n(B) = n(B - A) + n(A \cap B)$$

$$n(A \cap B) = \phi$$

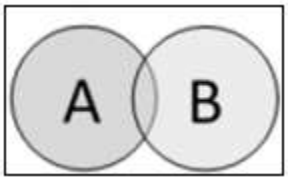
Example: Let, A = {1, 2, 6} and B = {7, 9, 14}, there is not a single common element, hence these sets are Disjoint sets.

OPERATIONS ON CLASSICAL SETS

Set Operations include Set Union, Set Intersection, Set Difference, Complement of Set, and Cartesian Product.

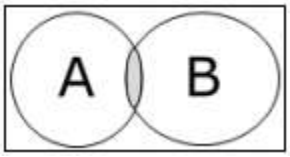
1. **Union:** The union of sets A and B (denoted by A ∪ B,A ∪ B) is the set of elements which are in A, in B, or in both A and B. Hence, A ∪ B = {x|x ∈ A OR x ∈ B}.

   Example: If A = {10,11,12,13} and B = {13,14,15}, then A ∪ B = {10,11,12,13,14,15}. The common element occurs only once. A ∪ B can be shown as



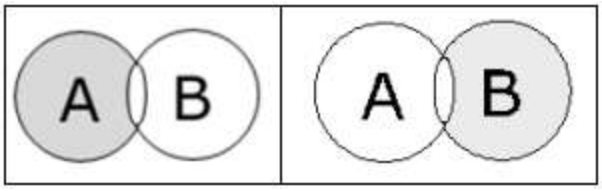
2. **Intersection:** The intersection of sets A and B (denoted by A ∩ B) is the set of elements which are in both A and B. Hence, A ∩ B = {x|x ∈ A AND x ∈ B}.

   Example: If A = {10,11,12,13} and B = {13,14,15}, then A ∩ B = {13}. A ∩ B can be shown as



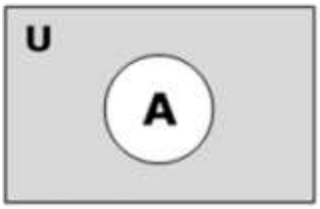
3. **Difference/Relative Complement:** The set difference of sets A and B (denoted by A–B) is the set of elements which are only in A but not in B. Hence, A − B = {x|x ∈ A AND x ∉ B}.

   Example: If A = {10, 11, 12, 13} and B = {13, 14, 15}, then (A − B) = {10,11,12} and (B − A) = {14,15}. Here, we can see (A − B) ≠ (B − A). Here, A – B & B – A can be shown as



4. **Complement of a Set:** The complement of a set A (denoted by A′) is the set of elements which are not in set A. Hence, A′ = {x|x ∉ A}. More specifically, A′ = (U−A) where U is a universal set which contains all objects.

   Example: If A = { 1, 2, 3, 4} and Universal set = U = { 1, 2, 3, 4, 5, 6, 7, 8}. Complement of set A contains the elements present in universal set but not in set A. Elements are 5, 6, 7, 8. Therefore, A complement = A′ = { 5, 6, 7, 8}. Hence, A = {x|x belongs to set of odd integers} then A′ = {y|y does not belong to set of odd integers}.



5. **<u>Cartesian Product / Cross Product</u>:** The Cartesian product of n number of sets $A_1,A_2,\ldots A_n$ denoted as $A_1 \times A_2 ... \times A_n$ can be defined as all possible ordered pairs $(X_1,X_2,\ldots X_n)$ where $X_1 \in A_1, X_2 \in A_2, \ldots Xn \in A_n$

   <u>Example</u>: If we take two sets A = {a,b} and B = {1,2},

   The Cartesian product of A and B is written as, $A \times B = \{(a,1),(a,2),(b,1),(b,2)\}$

   And, the Cartesian product of B and A is written as, $B \times A = \{(1,a),(1,b),(2,a),(2,b)\}$

PROPERTIES OF CLASSICAL SETS

Properties on sets play an important role for obtaining the solution. Following are the different properties of classical sets

1. <u>Commutative Property</u>: Having two sets A and B, this property states

   $A \cup B = B \cup A$

   $A \cap B = B \cap A$

2. <u>Associative Property</u>: Having three sets A, B and C, this property states

   $A \cup (B \cup C) = (A \cup B) \cup C$

   $A \cap (B \cap C) = (A \cap B) \cap C$

3. <u>Distributive Property</u>: Having three sets A, B and C, this property states

   $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

   $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

4. <u>Idempotency Property</u>: For any set A, this property states

   $A \cup A = A$

   $A \cap A = A$

5. <u>Identity Property</u>: For set A and universal set X, this property states

   $A \cup \varphi = A$

   $A \cap X = A$

   $A \cap \varphi = \varphi$

   $A \cup X = X$

6. <u>Transitive Property</u>: Having three sets A, B and C, the property states

   If, $A \subseteq B \subseteq C$, then $A \subseteq C$

7. Involution Property: For any set A, this property states

$$\overline{\overline{A}} = A$$

8. De Morgan's Law: It is a very important law and supports in proving tautologies and contradiction. This law states

$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$

TYPES OF MEMBERSHIP FUNCTIONS

- Triangular Membership Function
- Trapezoidal Membership Function
- Gaussian Membership Function
- Generalized Bell Membership Function
- Sigmoid Membership Function

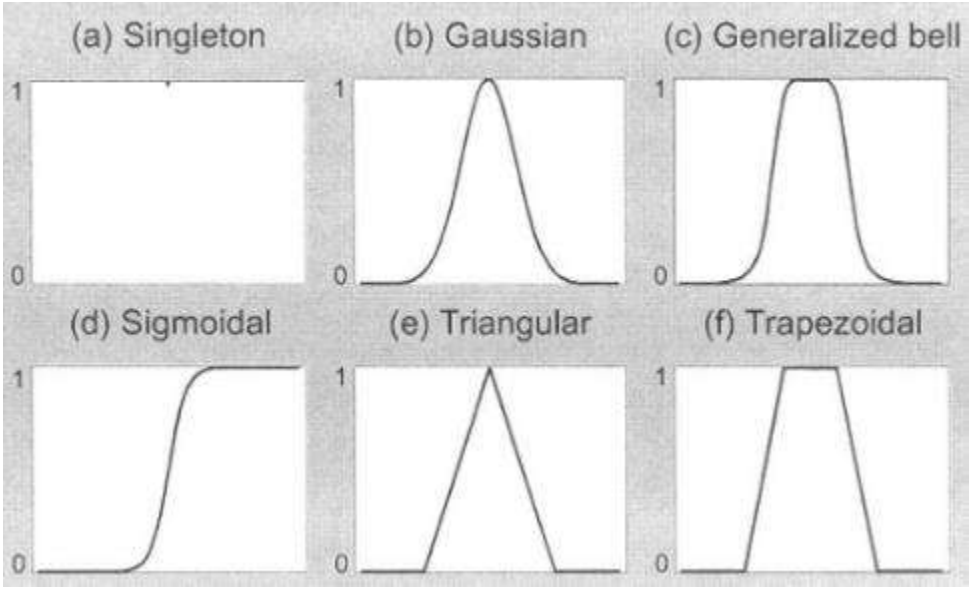


Fig: Various Types of Fuzzy Membership Functions

1. Triangular Membership Function: A Triangular Membership Function is specified by three parameters {a, b, c} as follows:

$$\text{triangle}(x; a, b, c) = \begin{cases} 0, & x \le a. \\ \frac{x-a}{b-a}, & a \le x \le b. \\ \frac{c-x}{c-b}, & b \le x \le c. \\ 0, & c \le x. \end{cases}$$

By using min and max, we have an alternative expression for the preceding equation:

$$\text{triangle}(x; a, b, c) = \max\left(\min\left(\frac{x-a}{b-a}, \frac{c-x}{c-b}\right), 0\right)$$

The parameters {a, b, c} (with a < b < c) determine the x coordinates of the three corners of the underlying Triangular Membership Function. Below figure illustrates a Triangular Membership Function defined by triangle (x; 20, 60, 80).

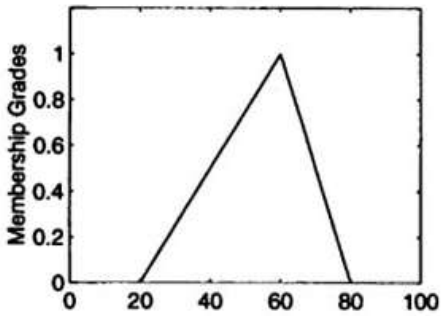


Fig: Triangular Membership Function

2. Trapezoidal Membership Function: A Trapezoidal Membership Function is specified by four parameters {a, b, c, d} as follows:

$$\text{trapezoid}(x; a, b, c, d) = \begin{cases} 0, & x \le a. \\ \frac{x-a}{b-a}, & a \le x \le b. \\ 1, & b \le x \le c. \\ \frac{d-x}{d-c}, & c \le x \le d. \\ 0, & d \le x. \end{cases}$$

An alternative concise expression using min and max is:

$$\text{trapezoid}(x; a, b, c, d) = \max\left(\min\left(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}\right), 0\right).$$

The parameters {a, b, c, d} (with a < b <= c < d) determine the x coordinates of the four corners of the underlying Trapezoidal Membership Function. Below figure illustrates a Trapezoidal Membership Function defined by trapezoid (x; 10, 20, 60 95). Note that a Trapezoidal Membership Function with parameter {a, b, c, d} reduces to a Triangular Membership Function when b is equal to c.

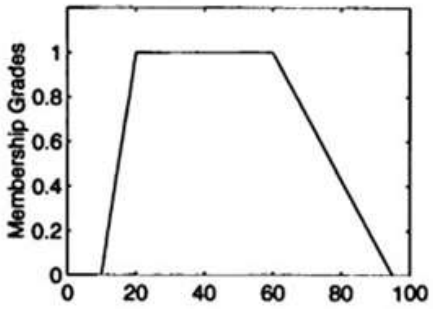


Fig: Trapezoidal Membership Function

Due to their simple formulas and computational efficiency, both triangular MFs and trapezoidal MFs have been used extensively, especially in real-time implementations. However, since the MFs are composed of straight line segments, they are not smooth at the corner points specified by the parameters. In the following we introduce other types of MFs defined by smooth and nonlinear functions.

3. Gaussian Membership Function: A Gaussian Membership Function is specified by tow parameters

$$\text{gaussian}(x; c, \sigma) = e^{-\frac{1}{2}\left(\frac{x-c}{\sigma}\right)^2}.$$

A Gaussian Membership Function is determined complete by c and σ; c represents the MFs centre and σ determines the MFs width. Below figure plots a Gaussian Membership Function defined by Gaussian(x; 50, 20).

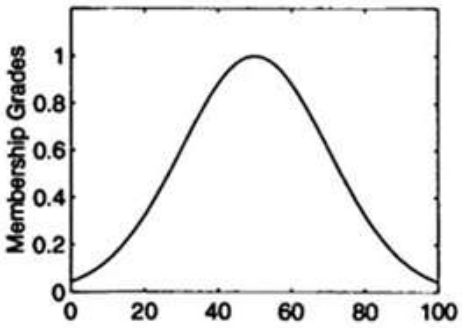


Fig: Gaussian Membership Function

4. Generalized Bell Membership Function**:** A Generalized Bell Membership Function (or) Bell-shaped Function is specified by three parameters {a, b, c}:

$$\text{bell}(x; a, b, c) = \frac{1}{1 + \left|\frac{x-c}{a}\right|^{2b}},$$

Where, the parameter b is usually positive. (If b is negative, the shape of this MF becomes an upside-down bell.) Note that this MF is a direct generalization of the Cauchy distribution used in probability theory, so it is also referred to as the Cauchy MF.

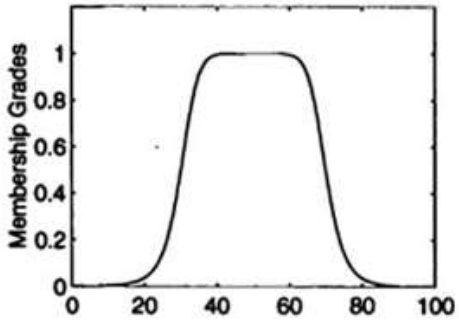


Fig: Generalized Bell Membership Function

Because of their smoothness and concise notation, Gaussian and Bell MFs are becoming increasingly popular for specifying fuzzy sets. Gaussian functions are well known in probability and statistics, and they possess useful properties such as invariance under multiplication (the product of two Gaussians is a Gaussian with a scaling factor) and Fourier transform (the Fourier transform of a Gaussian is still a Gaussian). The bell MF has one more parameter than the Gaussian MF, so it has one more degree of freedom to adjust the steepness at the crossover points. Although the Gaussian MFs and Bell MFs achieve smoothness, they are unable to specify asymmetric MFs, which are important in certain applications.

5. Sigmoid Membership Function: A Sigmoid Membership Function has two parameters: a responsible for its slope at the crossover point x = c. The membership function of the sigmoid function can be represented as Sigmf (x: a, c) and it is

$$sigmf(x;a,b,c) = \frac{1}{1+e^{-a(x-c)}}$$

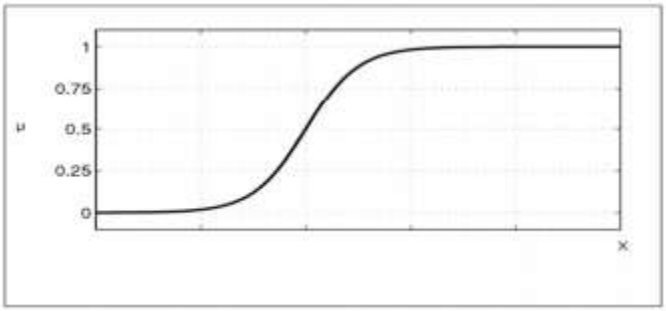


Fig: Sigmoid Membership Function

A Sigmoid Membership Function is inherently open right or left & thus, it is appropriate for representing concepts such as “very large” or “very negative”. Sigmoid Membership Function mostly used

as activation function of Artificial Neural Networks (NN). A NN should synthesize a close MF in order to simulate the behavior of a fuzzy inference system.

TYPES OF FUZZY PROPOSITIONS

1. Unconditional and Unqualified propositions: The canonical form of this type of fuzzy proposition is p:V is F. Where, V is a variable which takes value v from a universal set U. F is a fuzzy set on U that represents a given inaccurate predicate such as fast, low, tall etc. For example:

p: Speed (V) is high (F)

T(p) = 0.8, if p is partly true

T(p)=1, if p is absolutely true

T(p)=0, if p is totally false

Where, T(p) = μF(v) membership grade function indicates the degree of truth of v belongs to F, its value ranges from 0 to 1.

2. Unconditional and Qualified propositions: The canonical form of this type of fuzzy proposition is p:V is F is S. Where, V and F have the same meaning and S is a fuzzy truth qualifier. For example

P: Speed is high is very true

3. Conditional and Unqualified propositions: The canonical form of this type of fuzzy proposition is p: if X is A, then Y is B. Where, X, Y are variables in universes U1 and U2. A, B are fuzzy sets on X, Y. For example:

p: if speed is High, then risk is Low

4. Conditional and Qualified Propositions: The canonical form of this type of fuzzy proposition is p: (if X is A, then Y is B) is S. Where, all variables have same meaning as previous declare. For example:

p: if speed is high than risk is low is true.

FUZZY INFERENCE SYSTEM

Fuzzy Inference System is the key unit of a fuzzy logic system having decision making as its primary work. It uses the "IF…THEN" rules along with connectors "OR" or "AND" for drawing essential decision rules.

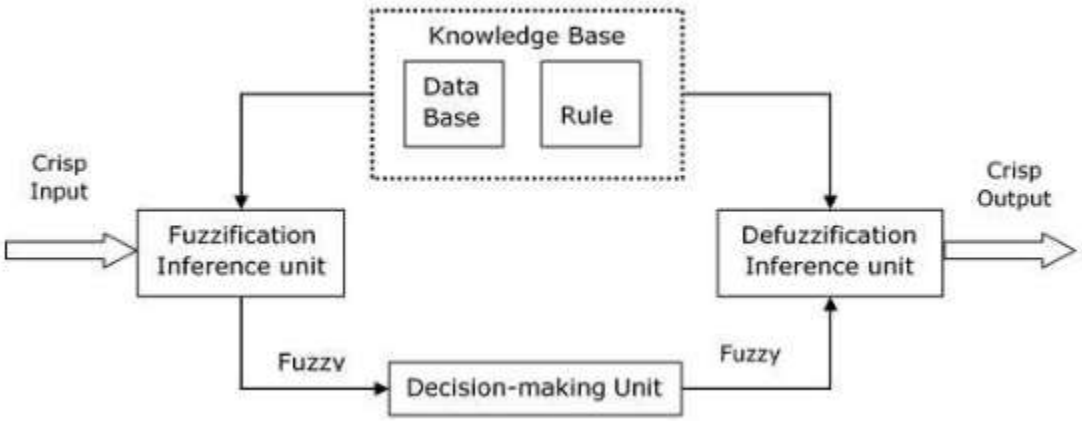
Characteristics of Fuzzy Inference System:

- The output from FIS is always a fuzzy set irrespective of its input which can be fuzzy or crisp.
- It is necessary to have fuzzy output when it is used as a controller.
- A defuzzification unit would be there with FIS to convert fuzzy variables into crisp variables.

Functional Blocks of FIS: The following are the functional blocks of FIS

- Rule Base: It contains fuzzy IF-THEN rules.

- Database: It defines the membership functions of fuzzy sets used in fuzzy rules.
- Decision-making Unit: It performs operation on rules.
- Fuzzification Interface Unit: It converts the crisp quantities into fuzzy quantities.
- Defuzzification Interface Unit: It converts the fuzzy quantities into crisp quantities. Following is a block diagram of fuzzy interference system.



Working of FIS:

- A Fuzzification unit supports the application of numerous Fuzzification methods, and converts the crisp input into fuzzy input.
- A knowledge base - collection of rule base and database is formed upon the conversion of crisp input into fuzzy input.
- The defuzzification unit fuzzy input is finally converted into crisp output.

Methods of FIS: Following are the two important methods of FIS, having different consequent of fuzzy rules:

- Mamdani Fuzzy Inference System
- Takagi-Sugeno Fuzzy Model (TS Method)

Mamdani Fuzzy Inference System: This system was proposed in 1975 by Ebhasim Mamdani. Basically, it was anticipated to control a steam engine and boiler combination by synthesizing a set of fuzzy rules obtained from people working on the system. Following steps need to be followed to compute the output from this FIS:

1. Set of fuzzy rules need to be determined in this step.
2. In this step, by using input membership function, the input would be made fuzzy.
3. Now establish the rule strength by combining the fuzzified inputs according to fuzzy rules.
4. In this step, determine the consequent of rule by combining the rule strength and the output membership function.
5. For getting output distribution combine all the consequents.
6. Finally, a defuzzified output distribution is obtained.

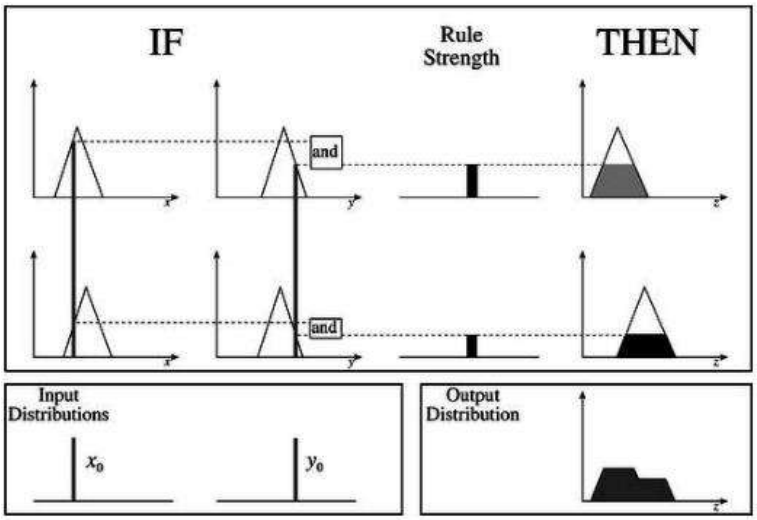


Fig: Block Diagram of Mamdani Fuzzy Interface System

Takagi-Sugeno Fuzzy Model (TS Method): This model was proposed by Takagi, Sugeno and Kang in 1985. Format of this rule is given as

IF x is A and y is B THEN Z = f(x,y)

Here, AB are fuzzy sets in antecedents and z = f(x,y) is a crisp function in the consequent. The fuzzy inference process under Takagi-Sugeno Fuzzy Model (TS Method) works in the following way

1. Fuzzifying the inputs − Here, the inputs of the system are made fuzzy.
2. Applying the fuzzy operator − In this step, the fuzzy operators must be applied to get the output.

Comparison between the two methods: The comparison between the Mamdani System and the Sugeno Model is

- Output Membership Function: The main difference between them is on the basis of output membership function. The Sugeno output membership functions are either linear or constant.
- Aggregation and Defuzzification Procedure: The difference between them also lies in the consequence of fuzzy rules and due to the same their aggregation and defuzzification procedure also differs.
- Mathematical Rules: More mathematical rules exist for the Sugeno rule than the Mamdani rule.
- Adjustable Parameters: The Sugeno controller has more adjustable parameters than the Mamdani controller.